FULLY DYNAMIC HIERARCHICAL CLUSTERING

Anonymous authors
Paper under double-blind review

ABSTRACT

Hierarchical clustering provides a multiscale representation of similarity data and is a fundamental tool in machine learning. While static hierarchical clustering has been extensively studied, maintaining a high-quality hierarchy under continuous changes in the underlying similarity graph remains challenging. In the fully dynamic setting, repeated reconstruction is often too expensive when vertices and their incident edges are inserted or removed. Existing dynamic approaches mainly focus on incremental maintenance, linkage-based merge rules, or graph summaries. These techniques do not directly maintain an explicit hierarchy whose global Dasgupta cost remains competitive after arbitrary vertex insertions and deletions. We propose a fully dynamic algorithm for explicit hierarchical clustering with approximation guarantees for Dasgupta cost. Our method is based on a local invariant that combines a deletion-adjusted cut lower bound with a counter that tracks changes in cluster membership. This enables the algorithm to preserve valid parts of the hierarchy and repair only affected subtrees. By organizing edges according to their lowest common ancestors, reconstruction can be restricted to the relevant induced subproblems. Given a finite-ρ oracle for $1/4$-balanced cut and any fixed $\Gamma > \rho$, our algorithm maintains a $1/5$-balanced hierarchy with a $\frac{15}{2}\Gamma$-approximation guarantee. The running time is bounded by the direct update work and the cost of rebuilding affected subtrees. A separate, explicitly stated regularity condition yields a compact amortized bound. The approximation and event-based time bounds do not use it. Queries for the exact cost take $O(1)$ time, while reporting the complete hierarchy takes $\Theta(1 + n_t)$ time, where n_t is the current number of active vertices. For a rebuild involving s vertices, the stable set recourse is $O(s)$ and the stable mass recourse is $O(s \log(2 + s))$.

1 INTRODUCTION

Hierarchical clustering represents a dataset by a sequence of nested partitions, allowing its structure to be examined at different levels of resolution (Laber & Batista, 2024; Ran et al., 2023). For data given by pairwise similarities, a natural requirement is that strongly connected points remain together until a fine level of the hierarchy. Dasgupta cost formalizes this requirement by charging each similarity edge its weight times the size of the smallest cluster containing its endpoints (Dasgupta, 2016). It provides a common criterion for comparing different hierarchies and makes it possible to study their approximation guarantees.

In many applications, the underlying dataset changes as records arrive or are removed. Recomputing a clustering structure from scratch after every update provides a direct way to inherit the guarantee of a static algorithm, but its running time depends on the entire current dataset. This motivates the fully dynamic setting, where the goal is to maintain a solution under updates with substantially smaller update costs. A standard model considers point insertions and deletions, where an update modifies one data point together with its associated information while preserving the rest of the instance. Fully dynamic algorithms have been extensively studied for several clustering objectives, including center-based clustering (Fichtenberger et al., 2021; Bateni et al., 2023) and graph clustering (Braverman et al., 2025), demonstrating that approximation guarantees can be maintained without rebuilding the entire solution from scratch. These developments build on a broader literature on maintaining graph structures and clustering-related objects under updates, where the main challenge is to localize the effect of changes rather than reconstruct the whole instance (Holm et al., 2001).

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

We first discuss fully dynamic flat clustering, where the maintained object is typically a single partition or a set of representative centers. For k-center clustering, early dynamic algorithms established constant-factor approximations under point updates, and subsequent work improved the approximation and update-time tradeoffs (Chan et al., 2018; Fichtenberger et al., 2021). More recent results extend these ideas to other center-based objectives: fully dynamic k-median algorithms achieve improved approximation and update guarantees through local modifications and coresets (Bhattacharya et al., 2025a), while dynamic k-means algorithms maintain compact summaries that support efficient updates (la Tour et al., 2024). For k-center, Bateni et al. (2023) obtained a $(2+\varepsilon)$-approximation with $O(k\,\mathrm{polylog}(n,\Delta))$ amortized update time against an oblivious adversary, where n bounds the number of active points and Δ is the metric aspect ratio. The closest recourse comparisons are flat-clustering data structures: Bhattacharya et al. (2024) study dynamic k-clustering with small recourse, and Bhattacharya et al. (2025b) give an ICML 2025 k-center structure with $O(1)$ amortized recourse and $\widetilde{O}(k)$ update time. These results optimize a center set or flat assignment. They do not maintain the nested cuts and Dasgupta objective considered here. Our comparison with them is therefore about the update/recourse setting, not a claim that their guarantees transfer to hierarchies.

Dynamic graph clustering studies the maintenance of graph partitions and summaries under edge or vertex updates. Prior work has considered several update models, including adaptive edge-label flips (Braverman et al., 2025) and vertex streams with insertions and random deletions (Cohen-Addad et al., 2024). These methods generally maintain a single partition or a restricted set of graph structures, and their objectives are defined on partitions or local graph properties. In contrast, our problem requires maintaining an entire hierarchy and the corresponding Dasgupta cost, where the contribution of each pair is determined by the level at which the pair is separated.

Dynamic hierarchical clustering is substantially more challenging because the maintained structure must satisfy consistency constraints across multiple resolutions. Incremental methods such as GRINCH and GraphGrove use local tree rearrangements to incorporate arriving points (Monath et al., 2019; 2023), but they do not provide fully dynamic approximation guarantees for Dasgupta cost. Dynamic single-linkage methods maintain linkage information through minimum spanning forests (Tseng et al., 2022; De Man et al., 2025). The closest dynamic hierarchical baseline is DynHAC: it maintains a $(1+\varepsilon)$-approximate average-linkage dendrogram under point insertions and deletions by repairing affected partition subgraphs (Yu et al., 2025). Its guarantee is with respect to approximate linkage merges, whereas ours is a cut-based guarantee for Dasgupta cost and an explicit $1/5$-balanced recursive hierarchy. For average linkage, TeraHAC computes approximate merge sequences in distributed settings (Dhulipala et al., 2023). A complementary direction uses dynamic graph sketches or cut sparsifiers to construct approximate hierarchies from compressed representations (Agarwal et al., 2022), but the maintained object is not an explicitly repaired hierarchy after each update. These distinctions explain why we use DynHAC and the dynamic cut-sparsifier work as closest comparisons rather than as interchangeable implementations.

Our problem can also be viewed as dynamically maintaining the recursive cuts underlying static hierarchical clustering. The recursive partitioning method of Dasgupta (2016) and the approximation analysis of Charikar & Chatziafratis (2017) provide the foundation for static algorithms. These results motivate retaining historical splits while their recorded balance and cut information remains valid, and rebuilding locally when an update invalidates that information. Vertex membership alone is insufficient for this decision: deletions can remove edges that supported an earlier cut, while insertions can add substantial crossing weight to a retained split. Reconstructing an affected subtree can also require edge information that extends beyond the subtree boundary. Our algorithm addresses both issues by storing validity state at each internal node and maintaining the edge ownership information needed to rebuild an affected subtree efficiently.

1.1 Our Contributions

We address these difficulties with a fully dynamic local-repair method. This is the first fully dynamic algorithm that repairs only affected parts of an explicit hierarchy while providing a Dasgupta cost approximation guarantee under a finite-factor balanced-cut oracle and both vertex insertions and deletions. The challenge is to decide when a historical split can still be trusted after both membership and edge changes, without recomputing a balanced cut throughout the tree. Each internal node therefore keeps a validity record from the time it was created, together with counters for subsequent membership changes and cumulative deleted internal edge weight. As long as the resulting local

balance and cut-ratio conditions remain valid, the split is retained. After an update, the algorithm examines only the affected ancestor chain and rebuilds the highest node whose record has become invalid. All unaffected parts of the hierarchy are preserved. We give the detailed invariant and reconstruction procedure in Section 3.

Theorem 3.5 gives the following guarantees. Given a finite-ρ approximation oracle for $1/4$-balanced cut and any fixed $\Gamma > \rho$, the algorithm maintains an explicit $1/5$-balanced hierarchy T_t with $\mathrm{cost}_{G_t}(T_t) \le \frac{15}{2}\Gamma\ \mathrm{OPT}(G_t)$ after every insertion and deletion. Thus $\Gamma = (1+\varepsilon)\rho$ gives an $O(\rho)$ approximation for fixed $\varepsilon > 0$. If every induced s-vertex subgraph has at most κs edges and $B_\kappa(s)$ bounds a complete static construction, an update incident to d_t edges uses $O(d_t + \log N)$ work outside reconstruction, and rebuilding an s-vertex subtree adds $B_\kappa(s)$. The resulting event-based total-time bound charges the actual subtrees rebuilt rather than a full reconstruction after every update, although a single repair may still rebuild the entire hierarchy. The current root handle and exact Dasgupta cost are maintained in $O(1)$ query time.

We also give an amortized analysis that separates membership-triggered rebuilds from ratio-triggered rebuilds. Membership-triggered rebuilds are charged to accumulated vertex changes, whereas ratio-triggered rebuilds are charged to inserted edge weight and deleted internal weight. An optional reconstruction regularity condition is used only to translate these weight-based charges into reconstruction time; it is not needed for the approximation guarantee or for the event-based total-time bound. Under this condition and a uniform bound on the total incident edge weight of each update, the amortized overhead becomes polylogarithmic when the remaining parameters are bounded.

The local-repair strategy also controls structural changes outside the region being rebuilt. Reconstructing an s-vertex subtree changes only $O(s)$ clusters and incurs $O(s\log(2+s))$ total cluster mass. This locality, however, does not imply that the rebuilt region is always small. Explicit constructions show that even when each update modifies only a constant-size star of incident edges, the repair rule may select an $\Omega(n)$-size subtree and may do so repeatedly under an adaptive update sequence. Thus, additional assumptions are needed to obtain uniformly small reconstruction costs for this repair strategy.

The main contributions are summarized as follows.

- We give a fully dynamic local-repair algorithm that maintains an explicit $1/5$-balanced hierarchy under vertex insertions and deletions, while guaranteeing a $\frac{15}{2}\Gamma$ approximation to Dasgupta's unrestricted optimum whenever the stated finite-factor cut-oracle interface is available.
- We give an explicit implementation whose total update time is parameterized by the direct update work and the sizes of the subtrees actually rebuilt, while supporting constant-time queries for the root handle and the exact clustering cost. We further establish polylogarithmic amortized overhead under the optional regularity condition in Remark 3.6, a sufficiently efficient static cut oracle, and a uniform bound $\omega_t \le W_\star$ on the total incident edge weight of each update. We also provide recourse guarantees outside rebuilt regions and examples showing that this local-repair strategy may repeatedly rebuild $\Omega(n)$-size subtrees.

2 PRELIMINARIES

At time t, let $G_t = (V_t, E_t, w_t)$ be the current nonnegative weighted undirected similarity graph, where zero-weight pairs are treated as nonedges, and let $n_t = |V_t|$. We identify data points with vertices of this similarity graph. Each vertex update either inserts one new vertex together with all of its positive incident edges or deletes one active vertex together with all of its incident edges. We call these incident edges the *update star*. The similarity between two vertices that remain active throughout an update does not change. Reusing a deleted label creates a new vertex instance. For the stated update-time bounds, incident-edge endpoints are supplied as stable vertex records, meaning persistent records whose identities survive tree repairs. We write d_t for the number of positive edges processed by update t and ω_t for their total weight. For an update sequence of length Q, define $N = \max\{2, n_0, n_1, \ldots, n_Q\}$ and $H_N = \left\lceil \log_{5/4} N \right\rceil + 1$. Let $\mathcal{W}^+ = \sum_{t:\,\mathrm{insertion}} \omega_t$ and $\mathcal{W}^- = \sum_{t:\,\mathrm{deletion}} \omega_t$ denote the total inserted and deleted star weight, respectively.

A hierarchy T is a rooted full binary tree whose leaf set, denoted $\mathrm{Leaf}(T)$, consists of the active vertices, and T_t denotes the hierarchy maintained after update t. For any node x, let S_x be the set of leaves in its subtree, let $s_x = |S_x|$, and let $T[x]$ denote the subtree rooted at x. For an internal node x, let L_x and R_x be the leaf sets of its two children. We write $\mathcal{I}(T)$ for the internal nodes of T. For a weighted graph $G = (V, E, w)$, write $w_G(u, v) = w(\{u, v\})$ on edges and extend it by zero to nonedges. In particular, $w_t(u, v) = w_{G_t}(u, v)$. Set $w_G(A, B) = \sum_{u \in A,\, v \in B} w_G(u, v)$ for $A, B \subseteq V$. The Dasgupta cost of T is

$$\mathrm{cost}_G(T) = \sum_{\{u,v\} \in E} w_G(u, v) \left| S_{\mathrm{LCA}_T(u,v)} \right|.$$

Here $\mathrm{LCA}_T(u, v)$ denotes the lowest common ancestor (LCA) of u and v in T. For an edge $e = \{u, v\}$, write $\mathrm{LCA}_T(e) = \mathrm{LCA}_T(u, v)$. For an empty vertex set, the hierarchy is empty and has cost zero. Let $\mathrm{OPT}(G) = \min_T \mathrm{cost}_G(T)$, where the minimum ranges over all hierarchies without a balance constraint. We say that T is β-balanced if every internal node satisfies $\min\{|L_x|, |R_x|\} \geq \beta s_x$. For a subset $S \subseteq V$ and $\eta \in (0, 1/2]$, define its induced η-balanced cut value by

$$\mathrm{BC}_G^{\eta}(S) = \min_{\substack{A \dot\cup B = S \\ \min\{|A|, |B|\} \geq \eta |S|}} w_G(A, B).$$

The static constructor is given a ρ-approximation oracle, with $\rho \geq 1$, that returns a $1/4$-balanced partition (A, B) of every nontrivial input S. Following the oracle-based method used in prior work (Agarwal et al., 2022), this oracle is an algorithmic interface that can be instantiated by any existing approximation algorithm for the required balanced-cut primitive. The oracle formulation separates dynamic maintenance from the choice of static cut routine: a better balanced-cut implementation immediately improves the resulting approximation and update bounds. If $c = w_G(A, B)$ is the returned cut value, then

$$\mathrm{BC}_G^{1/4}(S) \leq c \leq \rho\, \mathrm{BC}_G^{1/4}(S). \tag{1}$$

Throughout the paper, we fix $\Gamma > \rho$.

The approximation analysis does not require a sparsity assumption. Sparsity is used only for the running-time bounds. We call a graph sequence *hereditarily κ-sparse* if $|E_t[S]| \leq \kappa |S|$ for every time t and every subset $S \subseteq V_t$, where $E_t[S] = \{\{u, v\} \in E_t : u, v \in S\}$ is the induced edge set. Let $B_\kappa(s)$ be a nondecreasing upper bound on a complete static construction on any induced graph permitted by the update model with at most s vertices, including the explicit input processing and LCA registration described in Appendix C.

3 Fully Dynamic Hierarchical Clustering

The local-repair strategy must address two issues that are easy to conflate. First, a split that was good when a node was created may cease to be useful after later insertions and deletions: membership changes can destroy balance, and deleting internal edges can invalidate the lower bound used to justify the old cut. Second, even after identifying a subtree that should be replaced, rebuilding its induced instance must not require scanning all edges incident to that subtree. We handle the first issue with a deletion-adjusted record and the second with an LCA-based edge representation.

Concretely, a node remembers the cut value from its base state, counts the vertex updates that affect its leaf set, and accumulates internal edge weight removed by deletions. These quantities define local membership, balance, and ratio conditions. An insertion can make only the old ancestors of its chosen anchor vertex dirty, and a deletion can make only the surviving old ancestors of the deleted vertex dirty. The update scans that chain bottom-up and rebuilds the highest dirty node, so the new subtree simultaneously replaces all lower dirty nodes. For reconstruction, each edge is owned by the lowest common ancestor of its endpoints. Hence the buckets inside a subtree contain exactly its induced edges, while boundary edges remain owned above the rebuilt root and need not be scanned or reassigned. The implementation uses the dynamic LCA interface of Cole & Hariharan (2005) and maintains subtree cost aggregates so that the exact global objective is available at the root. The remainder of this section formalizes the record, proves that clean nodes imply the global approximation guarantee, and then analyzes updates, reconstruction, amortized time, and recourse, meaning changes in the represented cluster family.

For the dynamic analysis, we associate each internal node with the state of its subtree when that node is created. A lifetime of a node begins when this base state is initialized and ends when the node is suppressed or when a rebuild of the node or one of its ancestors starts. Reusing the same memory address starts a new lifetime. Let B_x be the base leaf set, let $b_x = |B_x|$, and let J_x^0 be the graph induced by B_x at creation time. These base objects are needed only for the analysis. The algorithm stores the cut value c_x^0 obtained at creation and sets $\lambda_x = c_x^0/\rho$. Feasibility of the returned cut gives $\rho^{-1}\,\mathrm{BC}_{J_x^0}^{1/4}(B_x) \le \lambda_x$, while the oracle guarantee establishes the *base validity condition*

$$\lambda_x \le \mathrm{BC}_{J_x^0}^{1/4}(B_x). \tag{2}$$

The local conditions below use this value through c_x^0/ρ. A separate copy of λ_x is not required.

During the lifetime of x, the algorithm maintains its current size s_x, its current cut weight $c_x = w_{G_t}(L_x, R_x)$, a membership counter q_x, and the cumulative deleted internal weight E_x. Each insertion or deletion affecting the subtree increments q_x once at every surviving old ancestor. Newly created nodes are initialized with $q_x = E_x = 0$ after the update has been completed, and a parent that is directly suppressed by a deletion receives no terminal increment. We call each such increment of q_x a *membership touch* of x. If a vertex v is deleted, every surviving old ancestor x receives

$$E_x \leftarrow E_x + \sum_{u \in S_x^- \setminus \{v\}} w_{t-1}(v, u),$$

where S_x^- is the leaf set of x immediately before the deletion. Thus E_x records all internal weight lost from the subtree during the lifetime, including edges incident to vertices inserted after the base state was created. Insertions do not increase E_x.

Definition 3.1. For an internal node x, define the following local validity conditions.

1. *Membership.* $16q_x \le b_x$.
2. *Balance.* $5\min\{|L_x|, |R_x|\} \ge s_x$.
3. *Ratio.* $c_x \le \Gamma \max\{0, \lambda_x - E_x\}$.

A node is *clean* if all three conditions hold and *dirty* if at least one fails. A hierarchy is clean if every internal node is clean. Dirtiness is a local maintenance signal. The algorithm restores the maintained conditions by rebuilding the highest dirty node before the update returns.

The data structure explicitly tests Condition 2, although for a maintained node it follows from Condition 1. Condition 1 simultaneously preserves balance and controls the projection back to the base instance. E_x accounts for deleted base-edge weight.

Lemma 3.2. *Let x be a surviving internal node whose base split is $1/4$-balanced and whose value λ_x satisfies the base validity condition* (2)*. If Condition 1 holds, then Condition 2 holds and $\mathrm{BC}_{G_t}^{1/3}(S_x) \ge \max\{0, \lambda_x - E_x\}$. Consequently, every clean node maintained by the algorithm satisfies $c_x \le \Gamma\,\mathrm{BC}_{G_t}^{1/3}(S_x)$.*

Proof. Let $D_x \subseteq B_x$ be the deleted base vertices and let A_x^+ be the remaining vertices inserted after the node was created. During the lifetime of x, rebuilding or suppressing a strict descendant preserves the leaf set of x, and suppressing a strict ancestor while x survives only changes the attachment of the surviving branch containing x. Operations outside the subtree also leave S_x unchanged. Hence insertions and deletions are the only operations that change S_x, so $S_x = (B_x \setminus D_x) \,\dot\cup\, A_x^+$ and $|D_x| + |A_x^+| \le q_x$. Write $d = |D_x|$ and $a = |A_x^+|$. Each side of the base split contained at least $b_x/4$ vertices and therefore currently contains at least $b_x/4 - d$. Since $a + d \le b_x/16$ and $|S_x| = b_x - d + a$,

$$5(b_x/4 - d) - |S_x| = b_x/4 - 4d - a \ge 0,$$

so the current split is $1/5$-balanced.

Now fix any current $1/3$-balanced cut (A, B) of S_x and remove all vertices of A_x^+ from its two sides. Each restricted side contains at least

$$\left\lceil \frac{b_x - d + a}{3} \right\rceil - a \ge \frac{b_x - d - 2a}{3} \ge \frac{b_x - 2(a+d)}{3} \ge \frac{7b_x}{24} > \frac{b_x}{4}$$

surviving base vertices. Since these are integer counts, each is at least $\lceil b_x/4 \rceil$. Assigning the deleted base vertices to either side then gives a partition $(\widetilde{A}, \widetilde{B})$ with both sides of size at least $b_x/4$, hence a feasible $1/4$-balanced cut of B_x.

Surviving base edges keep both their weights and their crossing status under this projection. Every missing base edge is counted in E_x when its first endpoint is deleted, while edges incident to vertices inserted after the node was created have nonnegative weight. Hence the restored base cut has weight at most the current cut weight plus E_x. By the base validity condition (2),

$$\lambda_x \le \mathrm{BC}^{1/4}_{J^0_x}(B_x) \le w_{J^0_x}(\widetilde{A}, \widetilde{B}) \le w_{G_t}(A, B) + E_x.$$

Minimizing over current $1/3$-balanced cuts gives $\mathrm{BC}^{1/3}_{G_t}(S_x) \ge \lambda_x - E_x$. Nonnegative cut weights yield the stated maximum with zero. The final claim follows from Condition 3. □

The proof uses the gap between the base $1/4$-balance requirement and the current $1/3$-balanced comparison cut to project the changing leaf set back to the base graph. The only part of the historical lower bound that may disappear is the internal weight removed by deletions, which is precisely the quantity recorded by E_x.

When a rebuild is triggered, we distinguish two cases according to the violated condition. If Condition 1 fails, we call it a membership-triggered rebuild. Otherwise, if Condition 3 fails, we call it a ratio-triggered rebuild. By Lemma 3.2, Condition 1 implies Condition 2, and therefore these two cases cover all possible rebuild events. If both conditions fail, we assign the update to the membership case.

3.1 Local Stability Under Dynamic Updates

A clean node preserves a stable representation of its original partition under vertex insertions and deletions. This invariant is the basis of our dynamic analysis; lifetime details are provided in the appendix.

For an internal node x, let B_x denote the set of leaves present when x is created, let q_x count the accumulated membership updates affecting x, and let D_x and A^+_x denote, respectively, the deleted original vertices and the inserted vertices currently contained in the subtree of x. The maintained invariant implies the following decomposition.

Lemma 3.3. *For every surviving internal node x,*

$$S_x = (B_x \setminus D_x) \,\dot\cup\, A^+_x, \qquad |D_x| + |A^+_x| \le q_x, \qquad \big||S_x| - |B_x|\big| \le q_x.$$

Moreover, if the original split of x is $1/4$-balanced and Condition 1 holds, then the current split remains $1/5$-balanced.

Lemma 3.3 controls membership changes during a node's lifetime. A retained node is tested using Conditions 1–3, and any selected rebuild is classified according to a violated validity condition. The algorithm preserves the parts outside the selected subtree, although the selected subtree can be the entire hierarchy. The full induction over node lifetimes and the accounting of insertions and deletions are provided in Appendix B.

3.2 From Local Validity Conditions to Global Cost

The local validity conditions control the cut at each internal node. Applying the following lemma to the maintained $1/5$-balanced tree turns the nodewise $1/3$-balanced-cut bounds into a global Dasgupta cost bound.

Lemma 3.4 (Charikar & Chatziafratis, 2017)**.** *Let G be a nonnegative weighted graph and let T be a β-balanced hierarchy, $\beta \in (0, 1/2]$. If every internal node satisfies $w_G(L_x, R_x) \le \gamma\, \mathrm{BC}^{1/3}_G(S_x)$ for $\gamma \ge 0$, then $\mathrm{cost}_G(T) \le \frac{3\gamma}{2\beta}\, \mathrm{OPT}(G)$.*

The argument associates a node of size s with maximal clusters of an unrestricted optimal hierarchy whose sizes are at most $2s/3$. These clusters induce a $1/3$-balanced cut of S_x, and every edge crossing this cut has optimal LCA size larger than $2s/3$. If $a_x = \min\{|L_x|, |R_x|\} \ge \beta s_x$, charging

Algorithm 1 Update(T, op)

Require: A clean hierarchy T and one vertex update operation op
Ensure: The updated hierarchy and exact cost

if op is an insertion **then**
    Choose an anchor vertex a and record the affected ancestor path
    Insert the new vertex as a sibling of a and assign new edges to LCA buckets
else
    Record the ancestor path of the deleted vertex and remove incident edges
    Suppress the deleted leaf and update affected LCA buckets
end if
for each affected ancestor x from bottom to top **do**
    Update (s_x, q_x, c_x, E_x) and recompute the subtree cost
    if x violates the local validity conditions **then**
        Mark x as the highest dirty node
    end if
end for
if a highest dirty node exists **then**
    Rebuild the corresponding subtree
end if
return the updated root and exact cost

x to the integer interval $\{s_x - a_x + 1, \ldots, s_x\}$ makes the intervals disjoint along every ancestor chain, while incomparable nodes use disjoint induced edge sets. The resulting threshold charges sum to at most $(3/2)\,\mathrm{OPT}(G)$, which yields $\mathrm{cost}_G(T) \le \frac{3\gamma}{2\beta}\,\mathrm{OPT}(G)$. The full proof is given in Appendix B.

Combining Lemma 3.4 with Lemma 3.2 gives the following guarantee for any clean hierarchy:

$$\mathrm{cost}_{G_t}(T_t) \le \frac{15}{2}\Gamma\ \mathrm{OPT}(G_t).$$

It remains to show how the data structure restores the local validity conditions after every update.

3.3 Vertex Updates and Subtree Repair

Initialization recursively applies the cut oracle. Each subsequent update first changes the graph and one ancestor path, and then rebuilds at the highest dirty node under the rebuild-case classification above. Dirtiness is tested using Conditions 1–3. At each internal node, the data structure stores the fields defining the local invariant, an LCA edge bucket $\mathcal{E}_x$ containing the current edges whose endpoints have LCA x, and the exact subtree cost C_x. Here, $\mathrm{left}(x)$ and $\mathrm{right}(x)$ denote the two children of an internal node x:

$$C_x = s_x c_x + C_{\mathrm{left}(x)} + C_{\mathrm{right}(x)}. \tag{3}$$

Leaves have size one and cost zero. A new static node has $b_x = s_x$, $c_x = c_x^0$, and $q_x = E_x = 0$, and is immediately clean.

For an insertion of a new vertex v, choose an existing anchor vertex a: use an endpoint of the insertion star if available, and any leaf otherwise. Insert v as a sibling of a under a new parent p. Old edges retain their LCAs. A new edge $\{v, u\}$ belongs to p when $u = a$, and otherwise to $\mathrm{LCA}_{T_{t-1}}(a, u)$. Apart from p, all new edge owners and all affected old nodes lie on the old ancestor chain of a. We bucket the star using these LCAs before changing the topology, install the new records, then update old ancestors from bottom to top. Each receives one unit of size and membership count, together with the weight of its own new crossing edges. The new node with two leaves uses $b_p = s_p = 2$, $c_p^0 = c_p = w_t(a, v)$, and $q_p = E_p = 0$, which makes it clean even when a and v are nonadjacent.

For a deletion, let the path from the old parent of v to the root be $x_0 = p, x_1, \ldots, x_r$, where x_r is the old root. Bucket each incident edge by its stored owner and let Δ_i be the total removed weight owned by x_i. Deleting the leaf and suppressing p preserves all LCAs of surviving edges. For each

$1 \le i \le r$, set $s_{x_i} \leftarrow s_{x_i} - 1$, $q_{x_i} \leftarrow q_{x_i} + 1$, $c_{x_i} \leftarrow c_{x_i} - \Delta_i$, and $E_{x_i} \leftarrow E_{x_i} + \sum_{j=0}^{i} \Delta_j$. The prefix sum equals the deleted vertex's predeletion internal weighted degree in $S^-_{x_i}$. Thus E_x is maintained with one star scan and one path scan, rather than a separate neighbor scan at each ancestor.

The update has three stages. It records the incident star in LCA-owned buckets, updates the affected ancestor path and selects the highest dirty node, and then replaces at most that one subtree. All validity fields and cost aggregates are updated before reconstruction, so the replacement sees one consistent checkpoint. The full pseudocode, including edge ownership and empty-tree cases, is given in Appendix C.

Boundary edges need not be scanned. Every edge is stored at the LCA of its endpoints, so $E_t[S_x] = \dot{\bigcup}_{y \in \mathcal{I}(T[x])} \mathcal{E}_y$. This identity permits collection of the rebuild graph in $O(s_x + |E_t[S_x]|)$ time. An edge with exactly one endpoint in S_x has its LCA strictly above x. Replacing the subtree on the same leaf set does not change that owner. Endpoints refer to stable vertex records, whose leaf pointers are refreshed by the new build. Boundary edges therefore need not be updated. A postorder pruning schedule unregisters a subtree and, when reversed, registers its replacement using a linear number of supported LCA operations. Appendix C provides complete pseudocode, including memory ownership and the representation invariant.

3.4 Approximation and Update Complexity

We use standard notation for dynamic algorithms to keep the main bounds compact. For a sequence whose maximum active size is N, let $u(N)$ denote the amortized update time excluding initial construction. Let $q_{\mathrm{cost}}(N)$ denote the time to return the exact Dasgupta cost. Returning the current root handle has the same $O(1)$ bound. Let $q_{\mathrm{tree}}(n_t)$ denote the time to explicitly report the maintained hierarchy. All fixed oracle and regularity parameters are suppressed from the arguments of these functions.

Theorem 3.5. *Assume the cut oracle satisfies* (1) *and fix $\Gamma > \rho$. Suppose the graph sequence is hereditarily κ-sparse and the dynamic LCA interface supports the required tree operations in worst-case $O(1)$ time. Let $B_\kappa(s)$ be a nondecreasing upper bound on a complete static construction on at most s active vertices, including input processing and LCA registration. After every legal vertex update,* DynamicHC *maintains an explicit full binary hierarchy T_t such that*

$$\mathrm{cost}_{G_t}(T_t) \le \frac{15}{2}\Gamma \ \mathrm{OPT}(G_t),$$

and T_t is $1/5$-balanced. At most one subtree is rebuilt per update. If $S^{(1)}, \ldots, S^{(R)}$ are the leaf sets of all rebuilds during Q updates, the total running time is

$$O\left(B_\kappa(n_0) + \sum_{t=1}^{Q} (d_t + H_N) + \sum_{j=1}^{R} B_\kappa(|S^{(j)}|) \right), \tag{4}$$

where $H_N = O(\log N)$ bounds the ancestor-path work per update. Thus a single update costs $O(d_t + H_N + B_\kappa(N))$ in the worst case, while work outside rebuilding is $O(d_t + H_N)$. Moreover, $q_{\mathrm{cost}}(N) = O(1)$ and $q_{\mathrm{tree}}(n_t) = \Theta(1 + n_t)$.

The event bound is unconditional apart from the stated implementation assumptions: it charges only the subtrees actually rebuilt and does not assume that rebuilds are small. The complete proof, including the exact H_N terms and the parent-pointer fallback for LCA queries, is given in Appendix C.

To obtain a compact amortized bound, we separate membership-triggered rebuilds from the ratio case, as defined by Conditions 1 and 3. The former are paid for by accumulated vertex changes. The latter are charged to inserted edge weight and deleted internal weight over node lifetimes. Converting this weight charge into reconstruction work needs the following additional condition.

Remark 3.6. There exists a fixed $\mu > 0$ such that every node lifetime actually selected in the ratio case satisfies $\mathrm{BC}^{1/4}_{J^0_x}(B_x) \ge \mu b_x$. This is a checkable property of the realized node lifetimes, used only for the compact amortized corollary below. It is not part of the correctness statement.

Table 1: Sliding-window results at $h = 64$. D/F denotes dynamic repair/full reconstruction. U is mean update time (ms), C is mean Dasgupta cost, C_D/C_F is the mean pointwise ratio on positive-cost states, $\widehat{\rho}_{\max}$ is the largest finite feasible-cut/certified-lower-bound ratio, zero-LB is the percentage of states containing a zero certificate, and T_{tree} is mean full-tree reporting time (μs).

Dataset	U_D	U_F	C_D	C_F	C_D/C_F	$\widehat{\rho}_{\max}$	zero-LB	$T_{\text{tree},D}$	$T_{\text{tree},F}$
Census	51.31	104.35	11.2	11.2	1.0000	1.0	0	55.30	56.52
KDD-Cup	60.79	122.92	254.6	254.1	1.0012	2.0	0	55.68	57.16
Song	66.89	174.13	359.6	359.7	1.0008	172.5	0	55.86	56.76
Drift	62.00	104.51	56365.4	56365.6	1.0000	5.27×10^6	2.72%	55.07	56.28
OptDigits	43.93	104.43	69.5	69.4	1.0013	17.8	0	56.10	57.35

The regularity condition 3.6 is used only for the amortized time bound. Neither the approximation guarantee nor (4) requires it. The full charging argument and its exact total time expression are deferred to Appendix D.

Theorem 3.7. *Under the assumptions of Theorem 3.5 and the regularity condition 3.6, suppose each static cut call on an explicit s-vertex, m-edge graph takes $O(K(s+m)L(N))$ time, where $K \geq 1$ and $L(N) \geq 1$ are uniform over recursive calls. Fix $\varepsilon > 0$, set $\Gamma = (1+\varepsilon)\rho$, and assume every updated star has total edge weight at most $W_\star$. Define $\bar{d} = Q^{-1}\sum_{t=1}^{Q} d_t$. Then, for $Q \geq 1$ updates,*

$$u(N) = \widetilde{O}\left(\bar{d} + K(1+\kappa)L(N)\left(1 + \frac{\Gamma W_\star}{\mu\varepsilon}\right)\right).$$

Hence, when the displayed parameters are bounded and $L(N)$ is polylogarithmic, the amortized overhead beyond reading the update star is polylogarithmic. Hereditary sparsity alone does not imply the regularity condition 3.6 or a fast cut oracle, so the unconditional statement remains (4).

4 Experimental Results

We evaluate a practical instantiation of our local-repair method on sliding windows of up to 64 vertices. Dynamic repair and full reconstruction use the same balanced-cut routine, implemented with a spectral graph-partitioning method (Spielman & Teng, 2007) and exact enumeration on small subproblems. Table 1 summarizes the primary setting $h = 64$, where h is the maximum active-window size. The reported $\widehat{\rho}_{\max}$ is a diagnostic ratio between a feasible cut and a certified lower bound, not an empirical approximation ratio to the optimum; a small spectral lower bound can therefore make this quantity large even when the two hierarchy costs are close. Full protocol and implementation details are in the appendix.

5 Conclusion

In this paper, we present DynamicHC, a local-repair algorithm that maintains an explicit hierarchy under vertex insertions and deletions. To the best of our knowledge, this is the first fully dynamic hierarchical clustering algorithm with a Dasgupta cost approximation guarantee. It retains valid splits, repairs only affected subtrees, and rebuilds each affected subproblem from its induced edges. With a ρ-approximation oracle for $1/4$-balanced cut and any $\Gamma > \rho$, the algorithm maintains a $1/5$-balanced hierarchy with cost at most $\frac{15}{2}\Gamma\,\mathrm{OPT}$. Experiments show that local-repair strategy reduces update time relative to full reconstruction while keeping objective values close.

AI use statement

We used generative AI tools to assist with language polishing of the manuscript and assistance in drafting validation and diagnostic code. All AI-assisted mathematical arguments were checked by the authors, experimental results were obtained by running the authors' implementations, and cited references were manually verified. The authors reviewed all AI-assisted content and take responsibility for the final text, claims, proofs, experiments, and results presented in this work.

Ethics statement

This paper studies hierarchical clustering in a fully dynamic setting and does not involve human-subject studies or the collection of new personal data. Our experiments use publicly available benchmark datasets. We do not foresee ethical concerns arising from this work.

Reproducibility statement

All datasets used in this work are publicly available benchmark datasets. We provide detailed descriptions of dataset sources, preprocessing procedures, experimental settings, evaluation metrics, and implementation details in the paper and appendix to facilitate reproduction of our results. The source code is included in the submission.

References

Arpit Agarwal, Sanjeev Khanna, Huan Li, and Prathamesh Patil. Sublinear algorithms for hierarchical clustering. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, pp. 3417–3430, 2022.

MohammadHossein Bateni, Hossein Esfandiari, Hendrik Fichtenberger, Monika Henzinger, Rajesh Jayaram, Vahab Mirrokni, and Andreas Wiese. Optimal fully dynamic k-center clustering for adaptive and oblivious adversaries. In *Proceedings of the 34th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 2677–2727, 2023.

Sayan Bhattacharya, Silvio Lattanzi, and Nikos Parotsidis. Efficient and stable fully dynamic facility location. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, pp. 23358–23370, 2022.

Sayan Bhattacharya, Martín Costa, Naveen Garg, Silvio Lattanzi, and Nikos Parotsidis. Fully dynamic k-clustering with fast update time and small recourse. In *Proceedings of the 65th IEEE Annual Symposium on Foundations of Computer Science*, pp. 216–227, 2024.

Sayan Bhattacharya, Martín Costa, and Ermiya Farokhnejad. Fully dynamic k-median with near-optimal update time and recourse. In *Proceedings of the 57th Annual ACM Symposium on Theory of Computing*, pp. 1166–1177, 2025a.

Sayan Bhattacharya, Martin Costa, Ermiya Farokhnejad, Silvio Lattanzi, and Nikos Parotsidis. Almost optimal fully dynamic k-center clustering with recourse. In *Proceedings of the 42nd International Conference on Machine Learning*, pp. 4196–4209, 2025b.

Vladimir Braverman, Prathamesh Dharangutte, Shreyas Pai, Vihan Shah, and Chen Wang. Fully dynamic adversarially robust correlation clustering in polylogarithmic update time. In *Proceedings of the 28th International Conference on Artificial Intelligence and Statistics*, pp. 1477–1485, 2025.

T.-H. Hubert Chan, Arnaud Guerquin, and Mauro Sozio. Fully dynamic k-center clustering. In *Proceedings of the 27th International World Wide Web Conference*, pp. 579–587, 2018.

Moses Charikar and Vaggos Chatziafratis. Approximate hierarchical clustering via sparsest cut and spreading metrics. In *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 841–854, 2017.

Vincent Cohen-Addad, Silvio Lattanzi, Andreas Maggiori, and Nikos Parotsidis. Dynamic correlation clustering in sublinear update time. In *Proceedings of the 41st International Conference on Machine Learning*, pp. 9230–9274, 2024.

Richard Cole and Ramesh Hariharan. Dynamic LCA queries on trees. *SIAM Journal on Computing*, 34(4):894–923, 2005.

Sanjoy Dasgupta. A cost function for similarity-based hierarchical clustering. In *Proceedings of the 48th Annual ACM Symposium on Theory of Computing*, pp. 118–127, 2016.

Quinten De Man, Laxman Dhulipala, and Kishen N Gowda. Fully-dynamic parallel algorithms for single-linkage clustering. In *Proceedings of the 37th ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 397–412, 2025.

Laxman Dhulipala, Jakub Łącki, Jason Lee, and Vahab Mirrokni. TeraHAC: hierarchical agglomerative clustering of trillion-edge graphs. *Proceedings of the ACM on Management of Data*, 1(3): 221:1–221:27, 2023.

Hendrik Fichtenberger, Monika Henzinger, and Andreas Wiese. On fully dynamic constant-factor approximation algorithms for clustering problems, 2021.

Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM*, 48(4):723–760, 2001.

Max Dupré la Tour, Monika Henzinger, and David Saulpic. Fully dynamic k-means coreset in near-optimal update time. In *Proceedings of the 32nd Annual European Symposium on Algorithms*, pp. 100:1–100:16, 2024.

Eduardo S. Laber and Miguel Batista. On the cohesion and separability of average-link for hierarchical agglomerative clustering. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, pp. 48710–48739, 2024.

Silvio Lattanzi and Sergei Vassilvitskii. Consistent k-clustering. In *Proceedings of the 34th International Conference on Machine Learning*, pp. 1975–1984, 2017.

Nicholas Monath, Ari Kobren, Akshay Krishnamurthy, Michael R. Glass, and Andrew McCallum. Scalable hierarchical clustering with tree grafting. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 1438–1448, 2019.

Nicholas Monath, Manzil Zaheer, and Andrew McCallum. Online level-wise hierarchical clustering. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 1733–1745, 2023.

Xingcheng Ran, Yue Xi, Yonggang Lu, Xiangwen Wang, and Zhenyu Lu. Comprehensive survey on hierarchical clustering algorithms and the recent developments. *Artificial Intelligence Review*, 56(8):8219–8264, 2023.

Daniel A. Spielman and Shang-Hua Teng. Spectral partitioning works: planar graphs and finite element meshes. *Linear Algebra and its Applications*, 421(2–3):284–305, 2007.

Tom Tseng, Laxman Dhulipala, and Julian Shun. Parallel batch-dynamic minimum spanning forest and the efficiency of dynamic agglomerative graph clustering. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 233–245, 2022.

Shangdi Yu, Laxman Dhulipala, Jakub Łącki, and Nikos Parotsidis. DynHAC: fully dynamic approximate hierarchical agglomerative clustering. In *Proceedings of the 25th SIAM International Conference on Data Mining*, pp. 252–260, 2025.

A Additional Dynamic Invariant Analysis

A.1 Recourse and the Limits of Full Replacement

For update t, let $W_t = V_{t-1} \cap V_t$ and compare the two hierarchies after restricting both to W_t. Stable set recourse counts changed internal clusters, while stable mass recourse weights each changed cluster by its size. Direct sibling insertion and deletion with suppression have zero stable recourse. If DynamicHC rebuilds a subtree on s current vertices, then

$$\mathrm{Rec}^{\mathrm{stab}}_{\mathrm{set}}(t) = O(s), \qquad \mathrm{Rec}^{\mathrm{stab}}_{\mathrm{mass}}(t) = O(s \log(2+s)).$$

The logarithmic factor in the second bound is the height of the balanced subtree. Sparse adaptive examples show that updates involving a constant number of edges can still force large changes to clusters of remaining vertices and require a complete-subtree-replacement strategy to perform repeated $\Omega(n)$ work. The first construction lower-bounds changes in represented clusters, whereas the second applies to the complete-subtree-replacement strategy analyzed here. Formal definitions, constants, and proofs are in Appendix E.

A.2 Local Validity During a Node Lifetime

The local validity conditions depend on both the current split and the changes recorded since the node was created. We use the same notation as in the main text: B_x is the leaf set at creation, q_x counts membership updates, c_x is the current cut weight, and E_x is the cumulative deleted internal weight. In particular, c_x is not an update counter, and λ_x is not a bound on the number of updates in a lifetime.

Lemma A.1. *For every surviving internal node x, changes outside its subtree preserve the fields used in the local validity conditions. If Condition 1 holds, then Condition 2 holds and $\mathrm{BC}^{1/3}_{G_t}(S_x) \geq \max\{0, \lambda_x - E_x\}$. After every update returns, all internal nodes satisfy Conditions 1–3.*

Proof. Lemma B.1 shows that updates outside the subtree do not change its leaf set or the sides of its split. Such updates do not increment q_x or E_x, and the base fields remain fixed during the node lifetime. Lemmas B.2 and B.4 give the implications of Condition 1. The update checks the affected ancestor path and rebuilds at its highest dirty node, if one exists. Lemma C.6 then restores all three local conditions. □

We distinguish two rebuild cases according to the violated condition. The membership case is accounted for using q_x, whereas the ratio case is accounted for using inserted edge weight and E_x in Appendix D. Rebuilding initializes new base fields and counters rather than transferring the old ones. All local validity conditions are restored before an update returns.

B Local Validity and Approximation Proofs

This section establishes the bounds for local validity conditions and the resulting global objective guarantees. A node is analyzed within one logical lifetime, which ends after rebuilding the node or any ancestor. Replacement nodes do not inherit base values or accumulated counters. Let B_x denote the base leaf set, $b_x = |B_x|$, and J^0_x the induced graph at creation.

B.1 Membership Stability and Deletion-Adjusted Validity

Lemma B.1. *Let x be a surviving internal node. Every base vertex that survives continuously throughout the lifetime remains in S_x and on its original side of the split. A vertex that lies outside B_x at creation and remains active cannot enter S_x. If $D_x \subseteq B_x$ is the set of deleted original vertex instances and A^+_x is the set of vertices inserted after creation that currently belong to S_x, then $S_x = (B_x \setminus D_x) \dot\cup A^+_x$, $|D_x| + |A^+_x| \leq q_x$, and $|s_x - b_x| \leq q_x$.*

Proof. A sibling insertion adds only the new vertex to each surviving ancestor of its anchor. A deletion removes only the designated vertex. Suppressing a strict descendant of x preserves the

remaining leaf set on that side of x, and rebuilding a strict descendant preserves its entire leaf set. If a strict ancestor of x is directly suppressed while x survives in the other child branch, the operation only reattaches that surviving branch and leaves S_x and both sides of x's own split unchanged. Operations outside the subtree do not change its membership. These cases exhaust all structural operations that can occur while the lifetime of x survives, so the two stability assertions follow by induction over the operations in the lifetime. Rebuilding x or an ancestor, or suppressing x, terminates the lifetime and therefore requires no induction step for the old local validity conditions.

Each deleted original vertex instance accounts for a distinct membership touch of x caused by deletion. Each vertex in A_x^+ accounts for a distinct membership touch caused by insertion. Thus their total number is at most q_x. Vertices inserted and subsequently deleted can only increase q_x further. The set identity gives $s_x = b_x - |D_x| + |A_x^+|$, proving the final bound. Treating a reinserted external label as a new vertex instance is essential for this accounting. □

Lemma B.2. *If the base split of x is $1/4$-balanced and Condition 1 holds, then its current split is $1/5$-balanced.*

Proof. Write $b = b_x$, $d = |D_x|$, and $a = |A_x^+|$. By Lemma B.1 and Condition 1, $a + d \le b/16$ and $s_x = b - d + a$. Each side retains at least $b/4 - d$ base vertices. Consequently, $5\min\{|L_x|, |R_x|\} - s_x \ge 5(b/4 - d) - (b - d + a) = b/4 - 4d - a \ge b/4 - 4(d + a) \ge 0$. The side sizes are integers, so each is at least $\lceil s_x/5 \rceil$. The argument uses a single bound on the total number of insertions and deletions. □

Lemma B.3. *Suppose Condition 1 holds. Restrict either side of any current $1/3$-balanced cut of S_x to the surviving base vertices. Both restricted sides contain at least $\lceil b_x/4 \rceil$ vertices. Distributing the vertices of D_x arbitrarily between these sides therefore produces a $1/4$-balanced cut of B_x.*

Proof. Use a, d, b as above. A current side has at least $\lceil (b - d + a)/3 \rceil$ vertices, so removing all a new vertices leaves at least $\lceil (b-d+a)/3 \rceil - a \ge (b-d-2a)/3 \ge (b-2(a+d))/3 \ge 7b/24 > b/4$ surviving base vertices. Integrality yields at least $\lceil b/4 \rceil$ vertices. Adding deleted base vertices to either side cannot decrease its size. □

Lemma B.4. *Let x be a surviving internal node satisfying the base validity condition* (2)*. At every checkpoint where Condition 1 holds, $\mathrm{BC}_{G_t}^{1/3}(S_x) \ge \max\{0, \lambda_x - E_x\}$.*

Proof. Fix a current $1/3$-balanced cut (A, B) of S_x. Project it to the surviving base vertices and restore the deleted base vertices as in Lemma B.3, obtaining a feasible base cut $(\widetilde{A}, \widetilde{B})$. Any base edge whose endpoints both survive has unchanged weight: the input model permits changes to old edges only through endpoint deletion. It also has the same crossing status in the two cuts.

Consider a base edge with at least one deleted endpoint. When its first endpoint was deleted, the other endpoint still belonged to the cluster of x, by Lemma B.1. Its weight was therefore included in the predeletion internal weighted degree added to E_x at that update. Distinct base edges account for distinct terms in these degree sums. Hence the total weight of all disappeared base edges is at most E_x. The counter E_x may also include deleted edges incident to post-base vertices, which only enlarges this upper bound.

The restored base cut contains the same crossing surviving original edges as (A, B) and at most E_x additional weight from disappeared base edges. Current edges incident to vertices inserted after the node was created are nonnegative. Therefore,

$$\lambda_x \le \mathrm{BC}_{J_x^0}^{1/4}(B_x) \le w_{J_x^0}(\widetilde{A}, \widetilde{B}) \le w_{G_t}(A, B) + E_x.$$

Minimizing over (A, B) and using nonnegative cut weights proves the lemma. The increment to E_x must use the internal weighted degree before deletion. Using the graph after deletion would omit exactly the removed edges. □

Corollary B.5. *Every clean internal node maintained by the algorithm satisfies $c_x \le \Gamma\, \mathrm{BC}_{G_t}^{1/3}(S_x)$.*

Proof. Every maintained internal node is created either by the static constructor or by the explicit initialization on two vertices, so its value λ_x satisfies the base validity condition (2). Conditions 1 and 3 hold by Definition 3.1. Apply Lemma B.4. □

Lemma B.6. *Let p be the new parent formed by inserting v as a sibling of the leaf for a. The initialization $b_p = s_p = 2$, $c_p^0 = c_p = w_t(a, v)$, $\lambda_p = c_p^0/\rho$, and $q_p = E_p = 0$ satisfies the base validity condition* (2) *and makes p clean, including when $w_t(a, v) = 0$.*

Proof. The only nontrivial cut of the graph on two vertices has weight $w_t(a, v)$ and is $1/2$-balanced. Because $\rho \geq 1$, its weight divided by ρ is a valid lower bound on its balanced cut optimum. Conditions 1 and 2 hold immediately, while $c_p = \rho\lambda_p \leq \Gamma\lambda_p$ proves Condition 3. No division by an edge weight is used. □

B.2 From Local Cuts to the Global Objective

The next argument compares the maintained hierarchy with an unrestricted optimal hierarchy. The balance requirement applies only to the maintained tree.

Lemma B.7 (Charikar & Chatziafratis, 2017)**.** *Let $G = (V, E, w)$ have nonnegative edge weights. Suppose that a hierarchy T is β-balanced for $\beta \in (0, 1/2]$ and every internal node x satisfies $w_G(L_x, R_x) \leq \gamma\, \mathrm{BC}_G^{1/3}(S_x)$ for some $\gamma \geq 0$. Then $\mathrm{cost}_G(T) \leq \frac{3\gamma}{2\beta}\, \mathrm{OPT}(G)$.*

Proof. The conclusion is immediate for $|V| \leq 1$. Otherwise let $n = |V|$ and fix an optimal hierarchy $T^\star$. For each edge e, let $r_e = |S_{\mathrm{LCA}_{T^\star}(e)}|$ be the size of its LCA cluster in $T^\star$. For a real threshold $\tau \geq 0$, define $\phi^\tau(e) = w(e)\mathbf{1}\{r_e > \tau\}$, where $\mathbf{1}$ is the indicator function. An edge of optimal LCA size r_e contributes at most $3r_e/2$ times to the following sum, because $|\{j \in \{1, \ldots, n\} : 2j/3 < r_e\}| = \min\{n, \lceil 3r_e/2 \rceil - 1\} \leq 3r_e/2$. Thus

$$\sum_{j=1}^{n} \sum_{e \in E} \phi^{2j/3}(e) \leq \frac{3}{2}\, \mathrm{OPT}(G). \tag{5}$$

Fix an internal node x and put $s = s_x \geq 2$. Among all clusters of $T^\star$ of size at most $2s/3$ that intersect S_x, take those that are maximal under inclusion. They are pairwise disjoint and cover S_x: every singleton in S_x is eligible, and the clusters of a hierarchy form a laminar family, meaning that any two clusters are disjoint or nested. Their nonempty intersections with S_x therefore partition S_x into blocks of size at most $2s/3$. Some union of whole blocks has size in $[s/3, 2s/3]$. Indeed, use a block already in that interval if one exists. Otherwise every block has size less than $s/3$. Accumulating blocks until the total first reaches $s/3$ produces a total strictly below $2s/3$.

The union and its complement both have integral size at least $\lceil s/3 \rceil$, so they form a $1/3$-balanced cut. Every edge crossing this cut has $r_e > 2s/3$: if its LCA cluster in $T^\star$ had size at most $2s/3$, that cluster would be contained in one of the selected maximal clusters, placing both endpoints in the same block. Hence $\mathrm{BC}_G^{1/3}(S_x) \leq \sum_{e \in E[S_x]} \phi^{2s_x/3}(e)$.

Let $a_x = \min\{|L_x|, |R_x|\}$ and associate with x the integer interval $\mathcal{J}_x = \{s_x - a_x + 1, \ldots, s_x\}$. Balance gives $a_x \geq \beta s_x$, and ϕ^τ is nonincreasing in τ. Therefore

$$\begin{aligned} s_x\, w_G(L_x, R_x) &\leq \frac{\gamma}{\beta} a_x \sum_{e \in E[S_x]} \phi^{2s_x/3}(e) \\ &\leq \frac{\gamma}{\beta} \sum_{j \in \mathcal{J}_x} \sum_{e \in E[S_x]} \phi^{2j/3}(e). \end{aligned} \tag{6}$$

If y is a strict descendant of x, then $s_y \leq s_x - a_x$, so $\mathcal{J}_y$ and $\mathcal{J}_x$ are disjoint. Incomparable nodes have disjoint leaf sets and hence disjoint induced edge sets. Consequently each pair (j, e) appears at most once when the right-hand side of (6) is summed over all internal nodes. Using the split decomposition of Dasgupta cost and (5),

$$\mathrm{cost}_G(T) = \sum_{x \in \mathcal{I}(T)} s_x\, w_G(L_x, R_x) \leq \frac{\gamma}{\beta} \sum_{j=1}^{n} \sum_{e \in E} \phi^{2j/3}(e) \leq \frac{3\gamma}{2\beta}\, \mathrm{OPT}(G).$$

All inequalities remain valid when the edge set is empty. □

Corollary B.8. *If every internal node of T_t is clean, then* $\mathrm{cost}_{G_t}(T_t) \le \frac{15}{2}\Gamma\ \mathrm{OPT}(G_t)$.

Proof. All base splits are $1/4$-balanced, including the splits at nodes with two leaves. Lemma B.2 makes the returned tree $1/5$-balanced, and Corollary B.5 gives the local guarantee with $\gamma = \Gamma$. Substitute $\beta = 1/5$ into Lemma B.7. □

C Implementation and Representation Correctness

This section specifies an implementation of the dynamic algorithm using explicit sparse graph records. A subtree replacement must recover its induced graph, remove the obsolete LCA representation, register the replacement, and restore the exact objective value, without scanning boundary edges.

C.1 Records, Lifetimes, and Operation Contracts

Every active vertex instance has one stable record containing its external label, its current leaf pointer, an incidence list, and a temporary partition mark in $\{\mathsf{unset}, \mathsf{left}, \mathsf{right}\}$. The dictionary Active maps active external labels to these records. Reusing a deleted external label creates a new vertex instance. Rebuilding a subtree preserves the records of its remaining vertices and updates their leaf pointers.

Every positive edge has one record containing two stable endpoint pointers, its immutable weight, an owner pointer, two incidence handles, and a handle for the owner bucket. An edge is linked once at each endpoint and once in the bucket of its owner. At every completed checkpoint,

$$\mathrm{owner}(e) = \mathrm{LCA}_T(e), \qquad \mathcal{E}_x = \{e : \mathrm{owner}(e) = x\}. \tag{7}$$

In particular, graph records point to vertices, rather than directly to tree leaves. This distinction is necessary for replacing a subtree without repairing its boundary edges.

A node stores an explicit leaf/internal tag together with logical parent and child pointers. A leaf additionally points to its stable vertex record. Every node stores its current leaf count s_x and cost of the induced subtree C_x, with $s_x = 1$ and $C_x = 0$ at a leaf. An internal node also stores the immutable base size b_x and base cut c_x^0, the current cut c_x, membership count q_x, cumulative deleted internal weight E_x, and owner bucket $\mathcal{E}_x$. We do not store a separate copy of $\lambda_x = c_x^0/\rho$. Temporary fields consist of a list of edge pointers $\mathcal{P}_x$ and an accumulator τ_x. Both are empty or zero whenever an update returns. The global pointer root is null exactly when the hierarchy is empty. An attachment slot is either the global root pointer or one of a parent node's child-pointer locations.

A node's logical lifetime begins after its base fields have been initialized and ends when the node is suppressed or its containing subtree begins logical replacement. For a node created during an update, the base graph is the graph after that update's complete star has been applied. No historical base graph or base vertex set is stored. Old ancestors that survive the direct insertion or deletion receive one increment to the membership count. A directly suppressed parent receives no final increment to either q_x or E_x and has no local validity condition to maintain after deletion.

We use the standard unit-cost word random-access machine (word-RAM) model for pointer and weight arithmetic. The dictionary of active labels supports lookup, insertion, and deletion in $O(\log N)$ worst-case time. As stipulated in the input model, incident-edge endpoints are supplied as stable vertex records. The dictionary operation for the updated vertex's label is included in the $O(H_N)$ path work, since $H_N = \Theta(\log N)$. Resolving d_t endpoint labels separately would add $O(d_t \log N)$ time.

Lists have independent headers, tails, and lengths. Given a saved handle, unlinking a list cell takes constant time. Insertion and removal at either end also take constant time. Traversal is linear in the number of cells. No algorithm uses constant-time indexed access to a linked list. Temporary lists of pointers to vertices, edges, and nodes use their own cells, distinct from the persistent incidence and owner handles. Popping a temporary cell frees the cell but preserves the referenced record. A move transfers ownership of a container. Each expression for an empty list creates a fresh container.

The static constructor consumes its input pointer lists. Its cut subroutine reads the input and returns two independently enumerable partition descriptions. Partitioning creates disjoint child input lists, clears all vertex marks, and frees the parent input cells and descriptions before recursing. Destroying an obsolete tree node frees only that node and its empty internal containers: it does not recurse, destroy graph records, or read or reset the pointer from a vertex record to its leaf. The last restriction matters because a replacement leaf may already have installed a new pointer in the same stable vertex record.

Following Cole & Hariharan (2005), we assume the dynamic LCA operation contract stated below. The logical hierarchy and the auxiliary tree used by the dynamic LCA data structure are distinct. A registered logical node uses its own record as the handle of the corresponding auxiliary node. Thus every saved pruning triple names the exact auxiliary nodes used by its operations. The module is assumed to support, in worst-case constant time, leaf insertion and deletion, subdivision of a tree edge by a new internal node, suppression of a named unary node, and LCA queries. Leaf deletion does not automatically suppress its parent. A permanent dummy node $\perp$, outside the hierarchy, is the auxiliary parent of the logical root. The implementation below uses only these primitives. It assumes neither arbitrary subtree reparenting nor a constant-time operation that links or cuts an entire subtree. Registration and unregistration may temporarily leave the external parent unary, but they issue no LCA query. External queries remain suspended until the replacement is fully registered and the full binary topology is restored.

C.2 Extraction of Induced Edges and LCA Registration

The registration module separates logical tree manipulation from the dynamic LCA representation. PRUNE records how to collapse a subtree to one surviving leaf. UNREGISTER executes that schedule to remove the old auxiliary LCA topology. REGISTER then reverses the schedule after the new logical tree has been built. Thus no boundary edge needs to be revisited during a replacement.

Lemma C.1. *If (7) holds, then, for every current subtree root x, writing $E_t[S_x]$ for the current edges with both endpoints in S_x, we have $E_t[S_x] = \dot{\bigcup}_{y \in \mathcal{I}(T[x])} \mathcal{E}_y$. Its induced graph and tree nodes can therefore be collected in $O(s_x + |E_t[S_x]|)$ time. Replacing $T[x]$ in the same attachment slot by a tree with the same leaf set leaves the owner of every boundary edge unchanged.*

Proof. Both endpoints of an edge lie in S_x if and only if their LCA lies in $T[x]$. Each edge has exactly one owner, proving the identity as a disjoint union. A depth-first traversal visits $2s_x - 1$ nodes and enumerates their buckets, without accessing incidence lists. For an edge with exactly one endpoint in S_x, the LCA is a strict ancestor of x. The attachment slot and leaf set determine the same ancestor after replacement. Stable endpoint records remain valid when new leaves are created, so neither its owner nor its endpoints require modification. □

Lemma C.2. *A full binary subtree with $s \geq 1$ leaves admits a sequence of $s - 1$ triples (ℓ, p, r), each of which deletes a current leaf ℓ and suppresses its parent p after the deletion leaves p with the single child r. The sequence is computable in $O(s)$ time. Removing the final survivor unregisters the entire subtree. Reversing the sequence registers the same subtree. Both use $O(s)$ allowed LCA primitives.*

Proof. In postorder, a leaf returns itself. At an internal node p, the recursive calls return survivors a, b for its two children. Append (b, p, a) and return a. When the triple is executed, both child subtrees have already collapsed to these leaves. Thus deleting b makes p unary, and suppression is legal. There is one triple per internal node. To reverse the process, first insert the last survivor under the external parent. For each triple in reverse order, subdivide the edge above r with p, then insert ℓ under p. These operations exactly invert the pruning. Deleting the final survivor during unregistration may temporarily leave the external parent unary. The interface explicitly permits that state, and the inverse registration restores its missing child before any LCA query is issued. Iteration uses the ends of a list, with no random indexing. The logical pointers remain unchanged throughout, so the same postorder schedule can describe a detached, newly constructed logical tree. □

Algorithm 2 Prune(x, F)

Input: A full binary subtree root x and a list F to which entries are only appended.
Output: The surviving leaf and a postorder pruning schedule appended to F.

if x is a leaf **then**
  Return x.
end if
Set $a \leftarrow$ PRUNE$(\mathrm{left}(x), F)$ and $b \leftarrow$ PRUNE$(\mathrm{right}(x), F)$.
Append the pruning triple (b, x, a) to F.
Return a as the surviving leaf.

Algorithm 3 Register(r, g)

Input: A logical subtree root r and an external auxiliary parent g.
Output: The subtree registered below g, with all logical pointers unchanged.

Initialize an empty list F, and call the PRUNE(r, F) algorithm to obtain the final survivor z.
Insert the auxiliary leaf z below g.
while $F \neq \varnothing$ **do**
  Remove the last pruning triple (ℓ, p, a) from F.
  Subdivide the auxiliary edge above a by p, and insert ℓ below p.
end while
Release the temporary list F.
Return with the subtree rooted at r registered below g and all logical pointers unchanged.

Algorithm 4 Unregister(F, z)

Input: A pruning schedule F and its final survivor z.
Output: The subtree unregistered, with F empty and all logical pointers unchanged.

while $F \neq \varnothing$ **do**
  Remove the first pruning triple (ℓ, p, a) from F.
  Delete the auxiliary leaf ℓ, and suppress the auxiliary unary node p.
end while
Delete the final survivor z from the auxiliary LCA topology.
Return with the subtree unregistered, $F = \varnothing$, and all logical pointers unchanged.

C.3 STATIC CONSTRUCTION, INITIALIZATION, AND EXACT AGGREGATES

BUILD follows the static recursion. It obtains one balanced cut, places crossing edges in the current node's LCA bucket, and recursively processes the two induced subgraphs. The base fields are initialized before recursion and the cost aggregate is pulled after both children have been built.

The helper NEWLEAF(v) allocates a leaf x, sets its parent and child pointers to null, initializes $s_x = 1$ and $C_x = 0$, and makes $v.\mathsf{leaf}$ point to x. We denote a call to the oracle in (1) by

$$\textsc{ApproxBalancedCut}_{\rho}^{1/4}(S, F),$$

where S is the vertex list and F is the list of induced edges. For an internal node, PULL(x) evaluates

$$C_x = s_x c_x + C_{\mathrm{left}(x)} + C_{\mathrm{right}(x)}. \tag{8}$$

It sets $C_x = 0$ at a leaf and does nothing on a null pointer. It does not change the size or the fields used in the local validity conditions. The predicate DIRTY(x) returns true exactly when at least one of Conditions 1–3 fails, as specified in Definition 3.1. Condition 3 is implemented without division by testing $\rho c_x \leq \Gamma \max\{0, c_x^0 - \rho E_x\}$. The predicate is called only on internal nodes with two nonempty children and correct current sizes.

The helper that creates the partition lists checks that each vertex is marked exactly once and that the two lengths sum to s. The oracle's valid partition guarantees coverage, and these checks verify partition bookkeeping only. Edges are assigned according to the two endpoint marks, and all marks are cleared before either recursive call. At entry to BUILD, edge owners are null and owner handles are detached. Their incidence handles may already be linked and are left unchanged.

Algorithm 5 Build(S, F)

Input: A vertex list S and a list of induced edges F with detached owner handles.
Output: A clean logical tree that consumes S and F while preserving all incidence links.

Set $s \leftarrow |S|$.
if $s = 0$ **then**
    Assert that $F = \varnothing$, release S and F, and return null.
end if
if $s = 1$ **then**
    Assert that $F = \varnothing$, remove the first vertex v from S, and release S and F.
    Call the NEWLEAF(v) algorithm to create a new leaf for v, and return this leaf.
end if
Call the APPROXBALANCEDCUT$_\rho^{1/4}(S, F)$ oracle to obtain a partition (A_0, B_0).
Form independent lists A and B from A_0 and B_0, and mark their vertices as left and right, respectively.
Partition F into $F_A, F_B, F_\times$ according to the endpoint marks.
Clear all endpoint marks, and release A_0, B_0 together with the cells of the input lists S, F.
Create an internal node x with a null parent and empty buckets $\mathcal{E}_x$ and $\mathcal{P}_x$.
Set $b_x \leftarrow s_x \leftarrow s$, $c_x^0 \leftarrow c_x \leftarrow \sum_{e \in F_\times} w(e)$, and $q_x, E_x, \tau_x \leftarrow 0$.
while $F_\times \neq \varnothing$ **do**
    Remove the first edge e from $F_\times$, and assert that its owner handle is detached.
    Set $\mathrm{owner}(e) \leftarrow x$, and link its owner handle into $\mathcal{E}_x$.
end while
Release the temporary crossing-edge list $F_\times$.
Call the BUILD$(\mathrm{move}(A), \mathrm{move}(F_A))$ algorithm to construct $\mathrm{left}(x)$.
Call the BUILD$(\mathrm{move}(B), \mathrm{move}(F_B))$ algorithm to construct $\mathrm{right}(x)$.
Set the parent pointers of both children to x, and call the PULL(x) algorithm.
Return the clean subtree rooted at x.

Initialization first creates the stable vertex records with empty incidence lists, unset marks, and dictionary entries. It then creates one record per input edge, links its two incidence handles, and puts its pointer in an independent edge list. We then invoke BUILD on independent lists of these vertices and edges. A nonempty returned root is assigned a null logical parent and registered under $\perp$ by Algorithm 3. A graph with no vertices returns a null root. This construction includes record creation, all recursive calls, and LCA registration in its static construction bound.

Proposition C.3. *Suppose every cut call on an explicit s-vertex, m-edge graph costs $O(K(s+m)L(N))$, where $K \geq 1$ and $L(N) \geq 1$ are uniform over recursive calls. On a hereditarily κ-sparse graph, the complete construction, including record initialization and registration, admits*

$$B_\kappa(s) = O(K(1+\kappa)(1+s)L(N)\log(2+s)).$$

This time bound is conditional on the stated implementation of the cut subroutine.

Proof. Every static split is $1/4$-balanced, so recursion depth is $O(\log(2+s))$. At any depth, vertex sets are disjoint, and their internal edge sets are disjoint subsets of the original edges. At that depth, computing the cuts and partitioning the inputs costs $O(K(s+m)L(N))$. Summing over the recursion depth and using $m \leq \kappa s$ gives the bound. The linear registration work is absorbed. Sparsity alone does not make the cut routine near-linear. For example, exhaustive exact enumeration of balanced cuts satisfies the correctness contract but yields an exponential construction bound. □

C.4 STAR BUCKETING AND VERTEX UPDATES

Let ℓ_v denote the current leaf of vertex record v. The helper ANCESTORS(x) returns an independent list $x, \mathrm{parent}(x), \ldots$ in bottom-to-top order, stopping before null. Discarding this list frees its cells, not its nodes.

Lemma C.4. *For insertion, choose any old vertex a and make the new leaf ℓ_v a sibling of ℓ_a under a new parent p. Then*

$$\mathrm{owner}(\{v,u\}) = \begin{cases} p, & u = a, \\ \mathrm{LCA}_{T_{t-1}}(a,u), & u \neq a. \end{cases} \tag{9}$$

Only p and the old ancestors of ℓ_a can have changed local fields. For deletion, write the path from the old parent to the root as $p = x_0, x_1, \ldots, x_r$, where x_r is the old root, define $\Delta_i = \sum_{e \ni v:\, \mathrm{owner}(e)=x_i} w(e)$, and let $D_i = \sum_{j=0}^{i} \Delta_j$. For every surviving ancestor x_i with $1 \le i \le r$, the local cut decreases by Δ_i and E_{x_i} increases by D_i. All other surviving old nodes retain their local fields.

Proof. Sibling insertion leaves the LCA of any two old leaves unchanged. For $u \neq a$, the new leaf's path first meets the path from u at the old LCA of a, u, proving (9). Only anchor ancestors gain the new member. Before deletion, a neighbor u belongs to S_{x_i} exactly when the owner of $\{v, u\}$ lies at or below x_i on this path. Thus D_i is precisely the weighted degree of v inside S_{x_i} immediately before deletion, including edges to vertices outside the base set. The edges lost from x_i's own split have total weight Δ_i. Suppressing p after deleting its leaf child does not change any LCA of surviving leaves: an LCA within its sibling subtree stays there, and an LCA with one endpoint outside lies strictly above p. □

Algorithm 8 accepts the complete positive star and a previously inactive external label. If the star is nonempty, its first endpoint is a convenient anchor. Otherwise, descending to any old leaf costs one tree-height traversal. Algorithm 6 first creates and buckets the new edge records, whose persistent handles remain detached throughout this phase. Consequently all LCA queries are made in the unchanged old topology. Algorithm 7 then installs the three persistent links of each new edge. It empties the supplied pointer list without releasing its container.

The insertion pseudocode therefore keeps the old topology fixed while bucketing the new star, then inserts the sibling pair, and only afterward updates ancestor records. This ordering makes every owner of a new edge unambiguous and leaves one path on which dirtiness can be tested.

Algorithm 6 Bucket$(v, \ell_a, \mathcal{N})$

Input: A new record v, an anchor leaf ℓ_a, and a complete star $\mathcal{N}$, with all pending ancestor buckets empty.
Output: A detached parent bucket F_p, its total weight w_p, and the pending ancestor buckets.

Initialize an empty parent bucket F_p and set $w_p \leftarrow 0$.
for each $(u, w) \in \mathcal{N}$ **do**
  Create an edge record $e = (v, u, w)$ with a null owner and detached handles.
  if $\ell_u = \ell_a$ **then**
    Append e to F_p, and update $w_p \leftarrow w_p + w$.
  else
    Let z be the lowest common ancestor of ℓ_a and ℓ_u, append e to $\mathcal{P}_z$, and update $\tau_z \leftarrow \tau_z + w$.
  end if
end for
Return (F_p, w_p).

Algorithm 7 Link(F, x)

Input: A list of new edges F with detached handles and a designated owner x.
Output: All persistent owner and incidence links installed, with F empty but retained.

while $F \neq \varnothing$ **do**
  Remove the first edge e from F, and set $\mathrm{owner}(e) \leftarrow x$.
  Link e into $\mathcal{E}_x$ and into the incidence lists of both endpoints.
end while
Return with all persistent owner and incidence links installed and $F = \varnothing$.

The new parent is based on the complete updated graph on two vertices. The pair weight is the zero-extended weight $w_t(a, v)$, including the possibility of a nonedge. It is immediately clean: its split is balanced, $q_p = E_p = 0$, and $c_p = \rho\lambda_p \le \Gamma\lambda_p$. Old ancestors receive their complete numerical update before a replacement begins. In particular, no lifetime is reset midway through processing an insertion star.

Algorithm 8 Insert$(label, \mathcal{N})$

Create a stable record v for the new vertex instance, with a null leaf pointer, an empty incidence list, and an unset mark.
if root $=$ null **then**
    Assert that $\mathcal{N} = \varnothing$, and set root $\leftarrow$ NewLeaf(v).
    Set Active$[label] \leftarrow v$, call the Register(root, $\perp$) algorithm, and return.
end if
Choose an anchor leaf ℓ_a, set $g \leftarrow \mathrm{parent}(\ell_a)$, and save the logical attachment slot of ℓ_a.
Let $\mathcal{A}$ be the ancestor list of g, and assert that $\mathcal{P}_x = \varnothing$ and $\tau_x = 0$ for every $x \in \mathcal{A}$.
Call the Bucket$(v, \ell_a, \mathcal{N})$ algorithm to obtain (F_p, w_p).
Call the NewLeaf(v) algorithm to create a new leaf ℓ_v, and create a parent p with children ℓ_a and ℓ_v.
Set $b_p \leftarrow s_p \leftarrow 2$, $c_p^0 \leftarrow c_p \leftarrow w_p$, and $q_p, E_p, \tau_p \leftarrow 0$.
Create empty buckets $\mathcal{E}_p$ and $\mathcal{P}_p$, and set the new logical parent pointers.
Subdivide the auxiliary edge above ℓ_a by p, and insert ℓ_v below p.
Attach p in the saved logical slot, and set root $\leftarrow p$ when $g =$ null.
Set Active$[label] \leftarrow v$, call the Link(F_p, p) algorithm, and release F_p.
for each $x \in \mathcal{A}$ **do**
    Call the Link$(\mathcal{P}_x, x)$ algorithm to install the pending edges owned by x.
end for
Call the Pull(p) algorithm, and initialize the highest dirty ancestor as $r \leftarrow$ null.
for each $x \in \mathcal{A}$ from bottom to top **do**
    Update $s_x \leftarrow s_x + 1$, $q_x \leftarrow q_x + 1$, and $c_x \leftarrow c_x + \tau_x$.
    Call the Pull(x) algorithm, and set $r \leftarrow x$ if x is dirty.
    Reset the pending weight accumulator by setting $\tau_x \leftarrow 0$.
end for
Discard $\mathcal{A}$, and call the Rebuild(r) algorithm if $r \neq$ null.

The accumulator D in Algorithm 9 includes the contribution at the suppressed parent even though that parent receives no counter update. Consequently every strict ancestor receives its full internal weighted degree before deletion. All edges owned by p join the deleted leaf to its sibling subtree and are removed by the incidence scan, justifying the assertion that the bucket is empty. The auxiliary leaf is deleted exactly once before p is suppressed, and the obsolete records are destroyed afterward. Costs are recomputed only after logical suppression. The sibling's own subtree, fields, and cost are unchanged.

Deletion uses the dual order. It first removes the incident edge records and accumulates owner weights, applies prefix sums to the surviving ancestors, suppresses the deleted leaf's parent, and finally pulls costs and repairs the highest dirty ancestor.

C.5 Rebuilding at the Highest Dirty Node and Answering Queries

Rebuild collects exactly the induced subgraph, detaches only its internal LCA buckets, unregisters the old auxiliary LCA topology, and invokes Build on the same vertex and edge records. It then installs the fresh root in the saved slot and pulls the unchanged ancestors. This explains why replacement work is local to the selected subtree plus one ancestor path.

The collector in Algorithm 10 performs an ordinary stack traversal: it appends every visited node to a node list, appends each leaf's vertex record to a vertex list, and, at each internal node, appends the bucket entries to a list of edge pointers and pushes both children. It allocates independent pointer cells and frees the traversal stack afterward. Lemma C.1 identifies exactly the graph collected.

Internal edges retain their incidence links throughout replacement. They are detached only from old owner buckets, then assigned to new buckets by the static constructor. Old nodes remain allocated while their pruning schedule and replacement are processed, but each is released exactly once afterward. The input leaf set has size at least two, so the replacement root is nonnull. Strict ancestors retain their sizes, cut values, base values, membership counts, and deleted internal weight. Only their cost aggregates need to be pulled again.

Algorithm 9 Delete(v)

Set $\ell \leftarrow v.\mathsf{leaf}$ and $p \leftarrow \mathrm{parent}(\ell)$.
if $p = \mathsf{null}$ **then**
    Assert that $\ell = \mathsf{root}$ and that v has no incident edges.
    Delete the auxiliary leaf ℓ, and remove v from Active.
    Destroy ℓ and v, set $\mathsf{root} \leftarrow \mathsf{null}$, and return.
end if
Let r be the sibling of ℓ, set $g \leftarrow \mathrm{parent}(p)$, save the logical attachment slot of p, and let $\mathcal{A}$ be the ancestor list of p.
Initialize $\tau_x \leftarrow 0$ for every $x \in \mathcal{A}$.
while v has an incident edge **do**
    Let e be the first incident edge of v, and set $z \leftarrow \mathrm{owner}(e)$.
    Update $\tau_z \leftarrow \tau_z + w(e)$, unlink all three handles of e, and destroy e.
end while
Initialize the prefix accumulator as $D \leftarrow 0$.
for each $x \in \mathcal{A}$ from bottom to top **do**
    Update the prefix accumulator by setting $D \leftarrow D + \tau_x$.
    if $x \neq p$ **then**
        Update $s_x \leftarrow s_x - 1$, $q_x \leftarrow q_x + 1$, $c_x \leftarrow c_x - \tau_x$, and $E_x \leftarrow E_x + D$.
    end if
    Reset the pending weight accumulator by setting $\tau_x \leftarrow 0$.
end for
Assert that $\mathcal{E}_p = \varnothing$, delete the auxiliary leaf ℓ, and suppress the auxiliary unary node p.
Attach the logical sibling r in the saved slot of p, set $\mathrm{parent}(r) \leftarrow g$, and set $\mathsf{root} \leftarrow r$ when $g = \mathsf{null}$.
Remove v from Active, discard $\mathcal{A}$, and destroy ℓ and p before destroying v.
Set $z \leftarrow g$, and initialize the highest dirty ancestor as $h \leftarrow \mathsf{null}$.
while $z \neq \mathsf{null}$ **do**
    Call the PULL(z) algorithm, and set $h \leftarrow z$ if z is dirty.
    Move to the next ancestor by setting $z \leftarrow \mathrm{parent}(z)$.
end while
Call the REBUILD(h) algorithm if $h \neq \mathsf{null}$.

The query for the exact cost returns zero for a null root and otherwise returns C_{root}. The hierarchy query returns the explicit root handle, also in constant time. Enumerating the entire hierarchy costs $\Theta(1 + n_t)$. The handle query does not perform that enumeration. Handles to tree nodes need not survive an update. Stable vertex handles remain valid for the entire corresponding vertex instance.

C.6 Closure of the Invariants and Maintenance Bounds

Lemma C.5. *After initialization and after every legal update returns, the following statements hold simultaneously: the logical hierarchy is full binary and its leaves are in bijection with the active vertex records. Every vertex points to its current leaf. The LCA topology equals the logical topology below $\perp$. Each edge has exactly two correct incidence links and one correct owner link. All stored sizes, cut weights, subtree costs, membership counters, and counters for deleted internal weight agree with their definitions. All temporary lists, accumulators, and vertex marks are empty or cleared.*

Proof. We induct on the completed updates and verify each constituent operation.

In initialization and static construction, each edge receives a unique record and two incidence links. At any recursive split, its endpoint marks send it to exactly one of the two recursive subproblems or the crossing bucket. A crossing edge has the current node as its LCA, so owner buckets partition the edge set correctly. Leaf creation installs the vertex mapping, and base fields are initialized from the current induced graph. Equation (8), evaluated after both children, gives the exact objective by the split decomposition of Dasgupta cost. Lemma C.2 shows that registration installs precisely

Algorithm 10 Rebuild(x)

Assert that x is internal, set $g \leftarrow \mathrm{parent}(x)$, and save the logical attachment slot of x.
Collect the vertices S, internal edges F, and obsolete nodes Z from $T[x]$.
Initialize an empty pruning schedule D, and call the PRUNE(x, D) algorithm to obtain its final survivor z.
for each $e \in F$ **do**
    Unlink e from $\mathcal{E}_{\mathrm{owner}(e)}$, and set $\mathrm{owner}(e) \leftarrow \mathsf{null}$.
end for
Call the UNREGISTER(D, z) algorithm, and release D.
Detach x from its saved slot, or set $\mathsf{root} \leftarrow \mathsf{null}$ when $g = \mathsf{null}$.
Set $\mathrm{parent}(x) \leftarrow \mathsf{null}$.
Call the BUILD$(\mathrm{move}(S), \mathrm{move}(F))$ algorithm to obtain the replacement root y.
if $g = \mathsf{null}$ **then**
    Set $\mathsf{root} \leftarrow y$ and $\mathrm{parent}(y) \leftarrow \mathsf{null}$, and call the REGISTER$(y, \perp)$ algorithm.
else
    Attach y in the saved slot, set $\mathrm{parent}(y) \leftarrow g$, and call the REGISTER$(y, g)$ algorithm.
end if
while $Z \neq \varnothing$ **do**
    Remove the first obsolete node u from Z, and destroy u together with its empty containers.
end while
Release the list Z.
while $g \neq \mathsf{null}$ **do**
    Call the PULL(g) algorithm, and move to the next ancestor by setting $g \leftarrow \mathrm{parent}(g)$.
end while

this logical tree. Input consumption and mark clearing restore the assertions about temporary fields, including at empty and singleton base cases.

For insertion, Lemma C.4 gives the correct owner for every new edge and preserves every old owner. Each new edge is linked at its two endpoints exactly once. The new parent is based on the updated pair graph. Each old anchor ancestor gains one member and one membership touch, and its cut gains precisely the weight of the new edges assigned to its bucket. Nodes outside that path retain their members and fields. Pulling the new parent followed by the old ancestors in bottom-to-top order always uses current child aggregates. All pending cells and accumulators are cleared before any replacement begins.

For deletion, the incidence scan visits every deleted edge once, accumulates its weight at its unique owner, and removes all three persistent links before destroying its record. The prefix scan implements precisely the degree identity before deletion in Lemma C.4, giving the specified increments to E_x at surviving ancestors. The parent's bucket becomes empty, and deletion followed by suppression preserves all surviving edge owners. After the topology change, bottom-to-top pulls restore the costs. The suppressed parent's lifetime ends without requiring a state after deletion. The vertex record is destroyed only after its incidence list has emptied.

For replacement, Lemma C.1 gives all and only the internal edges. Their old owner handles are detached before construction assigns new owners. Boundary owners and endpoint records stay valid. New leaf creation updates each vertex mapping, and destroying an obsolete leaf never overwrites that mapping. The pruning and inverse registration schedules use valid LCA primitives and leave the new topology registered. The list of old nodes contains each old node once, and all its owned buckets are empty when it is freed. Strict ancestors retain all fields used in the local validity conditions and receive new exact cost aggregates. These cases establish every claimed invariant and complete the induction. □

Lemma C.6. *After each update all internal nodes are clean, and at most one subtree is rebuilt.*

Proof. Static nodes are clean because their balanced-cut guarantee holds and both the membership count and deleted internal weight are zero. The newly inserted pair node is clean by its explicit initialization. By Lemma C.4, every old node that can become dirty lies on one ancestor chain. The bottom-to-top scan overwrites the candidate whenever it encounters a dirty node, so the final

candidate is the highest dirty node on that chain. Rebuilding there replaces all lower dirty nodes by clean static nodes. Its strict ancestors were already clean, and replacement changes none of their fields used to test the local validity conditions. All other nodes are unchanged. Thus no second scan or replacement is needed. If the scan finds no dirty node, the entire tree is already clean. □

Proposition C.7. *Under the stated implementation conditions, the direct work of update t outside a subtree replacement is $O(d_t + H_N)$. If a replacement has leaf set S, its additional work is $O(B_\kappa(|S|))$. The $O(H_N)$ cost updates at strict ancestors are included in the base work. Let $S^{(1)}, \dots, S^{(R)}$ be the leaf sets of the R replacement events. Then*

$$O\left(B_\kappa(n_0) + \sum_{t=1}^{Q}(d_t + H_N) + \sum_{j=1}^{R} B_\kappa(|S^{(j)}|)\right) \tag{10}$$

is a valid total bound. Queries for the exact cost and the root handle take $O(1)$ time.

Proof. Clean nodes are $1/5$-balanced by the argument for changes to the original leaf set. Thus a returned tree has height at most $\lceil \log_{5/4} \max\{1, n_t\} \rceil$. Inserting a sibling adds at most one temporary level, while deletion and suppression add none. Every path list, dirty scan, cost pull, and cleanup therefore uses $O(H_N)$ time. An insertion edge causes at most one LCA query and a constant number of record and handle operations. Deletion reads stored owners directly and processes each removed edge once. Choosing an anchor from a nonempty star takes constant time. Choosing an anchor for an empty star takes one downward path. This proves the base bound.

Collecting a replacement subtree, detaching its owners, unregistering its old topology, and freeing its old records costs $O(|S| + |E_t[S]|)$ by Lemmas C.1 and C.2. This work is absorbed by the complete construction bound, which is at least linear in the explicit input. Strict ancestors require only one further path of pulls. Summing proves (10). In particular a complete update is bounded by $O(d_t + H_N + B_\kappa(N))$, whereas $O(d_t + H_N)$ alone is only the base maintenance bound.

Finally, Lemma C.5 and (8) establish the query guarantees. □

If the dynamic LCA interface is unavailable, the same logical algorithm can answer insertion LCA queries by walking parent pointers. Each query then costs $O(H_N)$, so insertion base work becomes $O((d_t + 1)H_N)$. Deletion still uses persistent owner pointers and retains $O(d_t + H_N)$ base work. This fallback removes the constant-time dynamic LCA premise without changing the approximation or local validity arguments.

D Amortized Rebuilding Analysis

We charge only selected rebuild roots. Dirty descendants covered by a selected root are not separate events. We use the membership and ratio cases determined by Conditions 1 and 3. Throughout, $N = \max\{2, n_0, \dots, n_Q\}$ and $H_N = \lceil \log_{5/4} N \rceil + 1$. The nondecreasing function $B_\kappa(s)$ bounds a complete static construction on an induced graph permitted by the update model with at most s vertices. The linear work of replacing the old subtree is absorbed into a constant multiple of this bound.

D.1 Accounting for Membership-Triggered Rebuilds

Let $\mathcal{G}_N$ be the family of integer sequences $(r_0, \dots, r_p)$ with $p \geq 0$, $2 \leq r_0 < \cdots < r_p \leq N$, and $r_{i+1} \geq (75/68)r_i$ for every $0 \leq i < p$. The bound sums the construction cost per vertex over these geometrically increasing base sizes.

$$\Xi_\kappa(N) = \max_{(r_0,\dots,r_p)\in\mathcal{G}_N} \sum_{i=0}^{p} \frac{B_\kappa(\min\{2r_i, N\})}{r_i}. \tag{11}$$

The truncation applies to the argument of the construction bound, since a realized subtree never contains more than N vertices.

Theorem D.1. *Over Q vertex updates, the total work of membership-triggered rebuilds, where Condition 1 fails, is $O(Q\Xi_\kappa(N))$.*

Proof. Consider the lifetime of a node eventually selected for a membership-triggered rebuild, and write $b = b_x$ for its base size. The node is clean immediately before the triggering update and receives at most one membership touch during that update. At the rebuild checkpoint, write $q = q_x$ for its membership count and $s = s_x$ for its current size. Then $b/16 < q \le b/16 + 1$ and $s \le b + q \le 2b$. The last inequality uses $b \ge 2$. Distribute the reconstruction cost $B_\kappa(\min\{2b, N\})$ equally to the q membership touches in this lifetime. Each membership touch receives a charge of $O(B_\kappa(\min\{2b, N\})/b)$.

Fix an update and consider all lifetimes that can charge it, including lifetimes that end at later updates. Because the algorithm returns a clean tree after every update, each charged node lies on the clean ancestor chain immediately before the update that supplied that membership touch. Newly created nodes receive no membership touch in their creation update, and a directly suppressed node receives no terminal membership touch. If c and p are adjacent internal nodes on the ancestor chain before the update, with c the child of p, then $s_p \ge (5/4)s_c$ and $15b_y/16 \le s_y \le 17b_y/16$ for $y \in \{c, p\}$. Consequently, $b_p \ge (16/17)s_p \ge (16/17)(5/4)(15/16)b_c = (75/68)b_c$. The charged base sizes form a subsequence of this chain, and hence satisfy the constraints in (11). The total charge to one update is therefore $O(\Xi_\kappa(N))$. Every reconstruction cost is distributed exactly once, among the membership touches of its own lifetime. Lifetimes terminated by an ancestor replacement incur no separate rebuild charge. Summing over updates proves the claim. □

D.2 Inserted and Deleted Edge Weight Across Node Lifetimes

All counters in this subsection refer to logical node lifetimes rather than reusable memory addresses. For each lifetime of an internal node x, define an analysis-only counter P_x for inserted crossing weight and initialize it to zero. At the checkpoint after an insertion and before any rebuilding, charge each new edge to its unique LCA owner if that owner existed before the update, adding the edge weight to P_x. An edge owned by the newly created node with two leaves belongs to its base graph and receives no such charge. Existing edges in a rebuilt subtree likewise enter the new base graphs without carrying over insertion counters. If the insertion triggers a rebuild, its charges to old owners are applied before their lifetimes end.

While x survives, rebuilding a strict descendant preserves both sides of its split, and a deletion can only remove crossing edges. Every inserted edge that newly crosses this fixed split has x as its unique owner and is charged once to P_x. Therefore, at every rebuild checkpoint for a surviving node,

$$c_x \le c_x^0 + P_x = \rho\lambda_x + P_x. \tag{12}$$

Let $\mathcal{L}$ contain all lifetimes of internal nodes occurring in the finite prefix, including those still alive at its end. Write $\overline{P}_x, \overline{E}_x$ for their final counter values. If a replacement ends a lifetime, take these values after the current star update and before logical removal. If the parent is directly suppressed, use its existing counter values without adding the internal weight removed by the final deletion. For a lifetime still active at time Q, use its current values. These values are used only in the analysis and need not be stored by the algorithm. Recall that $\mathcal{W}^+ = \sum_{t:\,\text{insertion}} \omega_t$ and $\mathcal{W}^- = \sum_{t:\,\text{deletion}} \omega_t$, where ω_t is the total weight of the inserted or deleted star.

Lemma D.2. *If a node x is selected in the ratio case and $\lambda_x > 0$, then $P_x + \Gamma E_x > (\Gamma - \rho)\lambda_x$.*

Proof. When $E_x < \lambda_x$, failure of Condition 3 and (12) give $\rho\lambda_x + P_x \ge c_x > \Gamma(\lambda_x - E_x)$, which rearranges to the desired inequality. When $E_x \ge \lambda_x$, use $P_x \ge 0$ to obtain $P_x + \Gamma E_x \ge \Gamma\lambda_x > (\Gamma - \rho)\lambda_x$, where the strict inequality follows from $\rho > 0$ and $\lambda_x > 0$. □

Lemma D.3. *The final counter values satisfy $\sum_{x\in\mathcal{L}} \overline{P}_x \le \mathcal{W}^+$ and $\sum_{x\in\mathcal{L}} \overline{E}_x \le H_N\mathcal{W}^-$.*

Proof. Each inserted edge increases at most one old owner's insertion counter. Subsequent rebuilding makes it a base edge of new lifetimes and never charges its insertion again. Summing its single possible contribution proves the first inequality.

An edge is processed for deletion only when its first endpoint is deleted. Its weight is added to counters for deleted internal weight only at surviving nodes on the owner-to-root chain before deletion. Immediately before the deletion, the current hierarchy is clean: initially by static construction, and after any preceding update by Lemma C.6. Condition 2 at every clean internal node makes that

hierarchy $1/5$-balanced. Therefore every root-to-leaf path contains at most $\lceil \log_{5/4} \max\{1, n_{t-1}\} \rceil$ internal ancestor steps, and the owner-to-root chain is bounded by H_N. Omitting the terminal increment at a directly suppressed parent can only reduce its contribution. Since counters are not copied into new lifetimes, each deleted edge contributes at most $H_N\, w(e)$ to the entire sum of final counters for deleted internal weight. Summation proves the second inequality. □

Theorem D.4. *Let $\mathcal{R}_+$ denote the node lifetimes selected in the ratio case with $\lambda_x > 0$. Then*

$$\sum_{x \in \mathcal{R}_+} \lambda_x \le \frac{\mathcal{W}^+ + \Gamma H_N \mathcal{W}^-}{\Gamma - \rho}.$$

Proof. A selected lifetime ends with its rebuild and thus occurs at most once in $\mathcal{R}_+$. Its terminal counters include the triggering update. Applying Lemmas D.2 and D.3, with weak inequalities when summing, gives

$$(\Gamma - \rho) \sum_{x \in \mathcal{R}_+} \lambda_x \le \sum_{x \in \mathcal{R}_+} (\overline{P}_x + \Gamma \overline{E}_x) \le \mathcal{W}^+ + \Gamma H_N \mathcal{W}^-.$$

Divide by $\Gamma - \rho > 0$. The same argument includes an empty set of rebuilds in the ratio case. □

D.3 Bounding Reconstruction Work Using the Stored Values

The bound on the sum of the stored values does not itself control the sizes of rebuilt subtrees. The following additional condition supplies this conversion.

For the conversion below, we invoke the regularity condition 3.6. It is an additional property of the realized execution, applied only to nodes selected in the ratio case. The preceding correctness and event-based reconstruction bounds are independent of this condition. Feasibility of the base cut gives

$$\lambda_x = \frac{c_x^0}{\rho} \ge \frac{\mathrm{BC}_{J_x^0}^{1/4}(B_x)}{\rho} \ge \frac{\mu}{\rho} b_x > 0. \tag{13}$$

In particular, the regularity condition 3.6 excludes the ratio case with $\lambda_x = 0$. Equivalently for the analysis below, one may directly assume $\lambda_x \ge \nu b_x$ for all nodes selected in the ratio case and some $\nu > 0$. Then every occurrence of ρ/μ is replaced by $1/\nu$. Set $\ell_\kappa(N) = \max_{2 \le s \le N,\, s \in \mathbb{Z}} B_\kappa(s)/s$.

Theorem D.5. *Under the implementation and static construction conditions stated above and the regularity condition 3.6, the total running time, including initialization, is*

$$\begin{aligned} O\Bigg(& B_\kappa(n_0) + \sum_{t=1}^{Q} (d_t + H_N) + Q \Xi_\kappa(N) \\ & + \frac{\rho \ell_\kappa(N)}{\mu(\Gamma - \rho)} \big(\mathcal{W}^+ + \Gamma H_N \mathcal{W}^- \big) \Bigg). \end{aligned} \tag{14}$$

For $Q \ge 1$, let $\bar{d} = Q^{-1} \sum_{t=1}^{Q} d_t$. Excluding the initial construction, the amortized update time is

$$\begin{aligned} O\Bigg(& \bar{d} + H_N + \Xi_\kappa(N) \\ & + \frac{\rho \ell_\kappa(N)}{\mu(\Gamma - \rho)} \frac{\mathcal{W}^+ + \Gamma H_N \mathcal{W}^-}{Q} \Bigg). \end{aligned} \tag{15}$$

Including initialization in the average adds $O(B_\kappa(n_0)/Q)$. For $Q = 0$, only the initialization bound applies.

Proof. For a node selected in the ratio case, Condition 1 holds. Membership stability and (13) yield $s_x \le 17 b_x/16 \le 17\rho\lambda_x/(16\mu)$ and $B_\kappa(s_x) \le \ell_\kappa(N) s_x \le 17\rho\ell_\kappa(N)\lambda_x/(16\mu)$. Every node selected in the ratio case has $\lambda_x > 0$ under the regularity condition 3.6, so Theorem D.4 bounds the sum of these reconstruction costs by the last term in (14). Adding Theorem D.1, the $O(d_t + H_N)$ base maintenance per update, and the initialization cost proves (14). Removing initialization and dividing by Q proves (15). □

Corollary D.6. *Suppose the regularity condition 3.6 holds. Fix $\varepsilon > 0$, set $\Gamma = (1+\varepsilon)\rho$, and assume that every update satisfies $\omega_t \le W_\star$. Suppose also that every static cut call on s vertices and m edges costs $O(K(s+m)L(N))$, where $K \ge 1$ and $L(N) \ge 1$ are uniform over all recursive calls. Then the amortized update time, excluding initialization, is*

$$O\left(\bar{d} + K(1+\kappa)L(N)\left(1 + \frac{\Gamma W_\star}{\mu\varepsilon}\right)\log^2 N\right).$$

Proof. Write $A = K(1+\kappa)L(N)$. A static construction has logarithmic recursion depth because every cut is $1/4$-balanced. At a fixed depth, the vertex sets and the internal edge sets of recursive inputs are disjoint. Consequently, the total work per depth is $O(K(s+m)L(N))$, and hereditary sparsity gives $B_\kappa(s) = O(A(1+s)\log(2+s))$. Thus $\ell_\kappa(N) = O(A\log N)$. Every sequence in (11) has $O(\log N)$ terms, each at most $2\ell_\kappa(N)$, so $\Xi_\kappa(N) = O(A\log^2 N)$. Moreover, $\mathcal{W}^+ + \Gamma H_N \mathcal{W}^- \le Q\Gamma H_N W_\star$ because $\Gamma \ge 1$ and $H_N \ge 1$, while $\rho/(\Gamma-\rho) = 1/\varepsilon$. Substitute these estimates into (15) and use $H_N = O(\log N)$ and $A \ge 1$. □

The rebuild term is polylogarithmic when the displayed parameters are bounded appropriately and $L(N)$ is polylogarithmic. The input-reading term $\bar{d}$ remains explicit. The static-oracle factor $L(N)$ also remains explicit, since sparsity controls graph size rather than the complexity of the cut routine.

Lemma D.7. *If at least two vertices are present before an insertion, the algorithm rebuilds a subtree during that update.*

Proof. Let g be the old parent of the anchor leaf. One child of g has one leaf. Membership stability and cleanliness before the update imply $1 \ge b_g/4 - q_g \ge 3b_g/16$. Hence $b_g \le 16/3 < 16$, and the integer condition $q_g \le b_g/16$ forces $q_g = 0$. No membership touch has occurred in this lifetime, so its current sets on each side equal its base sets on each side. The base $1/4$-balance now implies $b_g = s_g \le 4$. The insertion increases q_g to one, violating Condition 1. Thus g is dirty. The selected highest dirty root can be an ancestor of g, but a rebuild must occur. □

Lemma D.7 concerns rebuild frequency rather than rebuild size. Every insertion into a state with at least two active vertices triggers one rebuild. The charging analysis controls aggregate construction work by charging each reconstruction cost, according to the rebuilt subtree size, to membership changes or failure of Condition 3. The triggered subtree may therefore be small even when a rebuild occurs.

E Recourse on Remaining Vertices and Replacement Lower Bounds

E.1 Restriction and the Recourse Upper Bound

For a hierarchy T, let $\mathcal{C}(T) = \{S_x : x \text{ is an internal node of } T\}$, including the root cluster when the root is internal. For $W \subseteq \mathrm{Leaf}(T)$, the restriction $T|_W$ removes all other leaves and empty branches and suppresses unary nodes. Left–right order is immaterial. Restrictions to sets with zero and one leaf are, respectively, the empty and singleton trees. For update t, put $W_t = V_{t-1} \cap V_t$ and define

$$\mathcal{D}_t^{\mathrm{stab}} = \mathcal{C}(T_{t-1}|_{W_t}) \triangle \mathcal{C}(T_t|_{W_t}),$$
$$\mathrm{Rec}_{\mathrm{set}}^{\mathrm{stab}}(t) = |\mathcal{D}_t^{\mathrm{stab}}|, \qquad \mathrm{Rec}_{\mathrm{mass}}^{\mathrm{stab}}(t) = \sum_{C \in \mathcal{D}_t^{\mathrm{stab}}} |C|.$$

Here $\triangle$ denotes symmetric difference. All labels identify vertex instances. Omitting restriction in these definitions gives raw set and raw mass recourse. These quantities measure changes in the represented cluster family.

Lemma E.1. *If $|S_x \cap W| \ge 2$, then $S_x \cap W$ is an internal cluster of $T|_W$.*

Proof. Set $Y = S_x \cap W$ and let $y = \mathrm{LCA}_T(Y)$ denote the lowest common ancestor of all leaves in Y. The node y lies in $T[x]$, and both of its child branches contain a leaf of Y. Since $Y \subseteq S_y \subseteq S_x$, we have $S_y \cap W = Y$. Both child branches remain nonempty under restriction, so y is not suppressed and represents the cluster Y. □

Theorem E.2. *A direct sibling insertion or deletion with suppression has zero stable set and stable mass recourse. If the update subsequently rebuilds a subtree on $s \geq 2$ current vertices, then*

$$\mathrm{Rec}^{\mathrm{stab}}_{\mathrm{set}}(t) \leq 2(s-1), \qquad \mathrm{Rec}^{\mathrm{stab}}_{\mathrm{mass}}(t) = O(s \log(2+s)).$$

When no rebuild occurs, raw set recourse is $O(H_N)$ and raw mass recourse is $O(N)$.

Proof. Restricting the tree after a sibling insertion to the old leaves removes the new leaf and suppresses its new parent, recovering the old tree. Direct deletion with suppression produces exactly the restriction of the old tree to the surviving leaves. Thus the direct operations have zero stable recourse.

Let S be the rebuilt leaf set, $Y = S \cap W_t$, and $r = |Y|$. Strict ancestors contain the same stable leaves before and after the replacement, and the other subtrees are unchanged. All cluster changes after restriction are therefore confined to Y. If $r \leq 1$, there is no internal cluster change. Otherwise, Lemma E.1 identifies the relevant restricted subtrees. Each has $r-1$ internal nodes, so their symmetric difference has size at most $2(r-1) \leq 2(s-1)$.

The selected root existed before the update, since newly created nodes are clean. Before an insertion, its subtree had $s-1$ leaves. Before a deletion, it had $s+1$ leaves. In either case that subtree was $1/5$-balanced, with height $O(\log(2+s))$. Sibling insertion increases height by at most one. Deletion, suppression, and restriction never increase height. The new subtree is $1/4$-balanced and its restriction also has height $O(\log(2+s))$. For a tree with at most s leaves and height H, $\sum_{C \in \mathcal{C}(T)} |C| \leq sH$, because each leaf contributes to at most H internal ancestors. Applying this bound to both restricted subtrees proves the mass bound.

Without a rebuild, raw changes occur only on the corresponding old and new update paths, including the inserted or suppressed parent. There are $O(H_N)$ such clusters. Along each affected path in the old and new hierarchies, cluster sizes decrease by a factor of at most $4/5$ at each edge. The sum of their sizes is therefore $O(N)$ on each path, proving the raw mass bound. □

E.2 Adaptive Lower Bounds for Balanced Outputs

The next statements compare the output cluster families of arbitrary algorithms. For a current graph G and output hierarchy T, we call T α-approximate when $\mathrm{cost}_G(T) \leq \alpha\,\mathrm{OPT}(G)$. The comparator has no balance constraint.

Lemma E.3. *For $0 < \beta \leq 1/3$, a β-balanced full binary tree exists for every positive integer leaf count. For $1/3 < \beta \leq 1/2$, such a tree exists if and only if its number of leaves is a power of two.*

Proof. Recursively split each leaf count as evenly as possible. For every integer $s \geq 2$, $\lfloor s/2 \rfloor \geq \lceil s/3 \rceil$, proving the first assertion. For the second, use strong induction on the leaf count. A singleton is the base case. Both children of an internal root must themselves be β-balanced, and hence have 2^i and 2^j leaves by induction. If $i \neq j$, the smaller side contains at most one third of the total leaves, contradicting $\beta > 1/3$. The two sizes are therefore equal and their sum is a power of two. Conversely, recursively halving a set of 2^r leaves yields a $1/2$-balanced and hence β-balanced full binary tree. □

Theorem E.4. *Let $0 < \beta \leq 1/3$, $n \geq 6$, and $1 \leq \alpha < n/5$. For any dynamic algorithm that returns a β-balanced, α-approximate hierarchy after every update, there is a legal insertion at time $t = 1$ into an edgeless graph on $n-1$ vertices such that*

$$\mathrm{Rec}^{\mathrm{stab}}_{\mathrm{set}}(1) \geq 2\left\lceil \log_{1/\beta} \frac{n}{5\alpha} \right\rceil, \qquad \mathrm{Rec}^{\mathrm{stab}}_{\mathrm{mass}}(1) \geq \lceil \beta n \rceil - 1.$$

The resulting graph has just two edges of unit weight and satisfies hereditary 1-sparsity. The adversary chooses the insertion after observing the preceding output tree. For randomized algorithms this statement applies to an adversary allowed to observe that output.

Proof. Observe the root split $A_0 \mid B_0$ of the old tree T_0, and choose $u \in A_0$ and $w \in B_0$. Insert a new vertex v with precisely the two edges of unit weight $\{u, v\}$ and $\{v, w\}$. A hierarchy that first merges u, v, then adds w, and then attaches the isolated vertices has cost $2 + 3 = 5$. Thus $\mathrm{OPT}(G_1) \leq 5$.

Let y be the LCA of u, v, w in the new tree T_1. Its split separates at least one of the two positive edges: otherwise all three vertices would lie in one child of y, contradicting its definition. This edge has LCA y, so $|S_y| \le \mathrm{cost}_{G_1}(T_1) \le \alpha\,\mathrm{OPT}(G_1) \le 5\alpha < n$. Write $y = x_0, x_1, \ldots, x_k = \mathrm{root}(T_1)$ for the ancestor path. Since x_i is the child on the root path, β-balance gives $|S_{x_i}| \ge \beta |S_{x_{i+1}}|$, and hence

$$n \le \beta^{-k}|S_y| \le \beta^{-k} 5\alpha, \qquad k \ge \left\lceil \log_{1/\beta} \frac{n}{5\alpha} \right\rceil. \tag{16}$$

For $0 \le i < k$, define $C_i = S_{x_i} \setminus \{v\}$. Each contains u, w and is an internal cluster of $T_1|_{V_0}$ by Lemma E.1. At every ancestor step the path child already contains v, so its nonempty sibling subtree contains only old vertices. Hence $C_0 \subsetneq C_1 \subsetneq \cdots \subsetneq C_{k-1}$. They are proper clusters because the other child of the new root is nonempty and also contains only old vertices. No proper cluster of T_0 contains both u and w, since these vertices lie on opposite sides of its root split. Thus all k clusters are new. Both restricted trees are full binary trees on the same $n-1$ leaves and therefore have exactly $n-2$ distinct internal clusters. Consequently the number of removed clusters equals the number of new clusters and is at least k, proving the set bound.

Finally, x_{k-1} is a child of the new root. Balance gives $|C_{k-1}| = |S_{x_{k-1}}| - 1 \ge \lceil \beta n \rceil - 1$. This one new cluster proves the mass bound. Every induced subgraph of the star with two edges has at most as many edges as vertices, establishing the stated sparsity promise. □

Corollary E.5. *Fix $0 < \beta \le 1/3$ and $n \ge 6$, and use the same adaptive insertion of a vertex with two incident edges at time $t = 1$. Suppose the returned hierarchy is β-balanced and every internal node satisfies $w_{G_1}(L_x, R_x) \le \Gamma\,\mathrm{BC}^{1/3}_{G_1}(S_x)$ for some finite Γ. Then, without an additional global approximation premise,*

$$\mathrm{Rec}^{\mathrm{stab}}_{\mathrm{set}}(1) \ge 2\left\lceil \log_{1/\beta} \frac{n}{4} \right\rceil, \qquad \mathrm{Rec}^{\mathrm{stab}}_{\mathrm{mass}}(1) \ge \lceil \beta n \rceil - 1.$$

For the proposed algorithm, $\beta = 1/5$, giving, respectively, $2\lceil \log_5(n/4) \rceil$ and $\lceil n/5 \rceil - 1$.

Proof. Let y again be the smallest cluster containing u, v, w, and put $r = |S_y|$. If $r \ge 5$, choose $a = \max\{3, \lceil r/3 \rceil\}$. For integer $r \ge 5$, $a \le \lfloor 2r/3 \rfloor$. Placing the three nonisolated vertices and $a-3$ isolated vertices on one side produces a $1/3$-balanced cut of weight zero. Hence $\mathrm{BC}^{1/3}_{G_1}(S_y) = 0$. But the actual split at y separates a positive edge and has weight at least one, contradicting the local invariant. Therefore $r \le 4$. Replace 5α by 4 in (16). The same argument for the restricted clusters proves both claims. □

Corollary E.6. *Under the assumptions of Theorem E.4, for every positive integer R_{cyc}, an adaptive sequence of $2R_{\mathrm{cyc}}$ vertex updates has total stable set recourse and total stable mass recourse at least R_{cyc} times their respective lower bounds in Theorem E.4. Under the local invariant of Corollary E.5, its bounds apply instead. For fixed β, the average stable set recourse and average stable mass recourse are, respectively, $\Omega(\log(n/\alpha))$ and $\Omega(n)$. Under the local invariant the average stable set recourse is $\Omega(\log n)$.*

Proof. Repeat the following cycle: observe the current output on the edgeless graph, insert a new vertex adjacent to one vertex on each side of its root split, and then delete that vertex. Deletion restores the edgeless graph on the same old vertices. The next insertion can therefore invoke the same lower bound against the newly observed tree. Each deletion has nonnegative recourse, so summing the insertion bounds proves the claim. When n/α approaches its lower endpoint 5, the ceiling in the bound on the chain length remains at least one, which also covers the claimed logarithmic bound of constant order in that regime. □

E.3 Repeated Linear Work for Complete Root Replacement

Proposition E.7. *For every $n \ge 6$, there is a legal adaptive update sequence with at most two edges present at any time and star degree at most two per update, on which every insertion forces the proposed algorithm to perform $\Omega(n)$ work collecting and rebuilding the entire root subtree. This is a lower bound for its complete replacement rule.*

Proof. Initialize on the edgeless graph with $n-1$ vertices. The root has $c_{\mathrm{root}} = \lambda_{\mathrm{root}} = 0$. Observe its split, choose old vertices u and w on opposite sides, and insert a vertex v joined only to these two vertices. Whichever side contains the anchor also contains the new vertex, so at least one new edge crosses the old root split. At the rebuild checkpoint, $c_{\mathrm{root}} > 0$ while $\max\{0, \lambda_{\mathrm{root}} - E_{\mathrm{root}}\} = 0$. Condition 3 fails at the root, which is therefore the highest dirty node. The rebuild procedure collects and rebuilds all n current vertices, taking $\Omega(n)$ work.

In the rebuilt graph, put the three nonisolated vertices on the same side and add isolated vertices until that side has $a = \max\{3, \lceil n/4 \rceil\}$ vertices. For $n \geq 6$, $a \leq n - \lceil n/4 \rceil$, so this is a feasible $1/4$-balanced cut of weight zero. Every cut oracle with a finite multiplicative approximation factor must return a zero-weight root cut. Thus the new root again has $\lambda_{\mathrm{root}} = 0$.

Both root children after the static construction have at least $\lceil n/4 \rceil \geq 2$ leaves, so v's direct parent is not the root. Deleting v cannot directly suppress the root. If the root survives without rebuilding, its value λ_{root} remains zero. If it is rebuilt, its new base graph is edgeless and its new value λ_{root} is also zero. Replacing a strict descendant preserves λ_{root} as well. The deletion therefore restores the induction hypothesis, allowing arbitrarily many cycles.

All intermediate graphs are forests with at most two edges of unit weight. They satisfy hereditary 1-sparsity, constant star degree, and constant star weight. The work bound follows from explicitly collecting and replacing the root subtree, independently of the semantic recourse measure. □

Proposition E.7 shows that sparsity and bounded star weight alone do not control the cost of complete subtree replacement, motivating the additional density condition used in the amortized bound.

F Additional Experimental Details

F.1 Experimental Setup

Datasets and update sequence. We evaluate on five public datasets: Census, KDD-Cup, Song, Drift, and UCI OptDigits. We use 5,000 points from each dataset; when subsampling is required, points are selected without replacement with seed 0 and retained in their original file order. Each run starts from an empty hierarchy and follows a sliding-window stream with maximum active size $h = 64$: points are inserted in order, the oldest point is deleted once the window is full, and the remaining points are removed at the end. Thus, each run contains 10,000 vertex updates. We use 50 evenly spaced query checkpoints.

Graph construction. Features are standardized before graph construction. For squared Euclidean distances d_{ij}^2, we set $\sigma^2 = \mathrm{median}\{d_{ij}^2 : d_{ij}^2 > 0\}$ and use the RBF similarity $s_{ij} = \exp(-d_{ij}^2/(2\sigma^2))$ with zero diagonal. Census, KDD-Cup, Song, and OptDigits use the symmetric 10-nearest-neighbor graph, while Drift uses the complete RBF graph. Similarities are precomputed once and preprocessing time is excluded from the reported update times.

Cut routine and implementation. For subproblems with $|S| \leq 14$, we enumerate all feasible $1/4$-balanced cuts up to complementary symmetry and return the minimum cut. Larger subproblems use a normalized-Laplacian Fiedler sweep (Spielman & Teng, 2007). For a selected cut of value c, minimum weighted degree $d_{\min}$, and second normalized-Laplacian eigenvalue λ_2, we use $\ell = \frac{1}{2}\lambda_2 \lceil |S|/4 \rceil d_{\min}$ as the lower certificate; numerically negligible certificates are recorded as zero. Dynamic repair and full reconstruction use the same cut routine.

Experiments were run in Python 3.8.18 on an Intel Xeon Platinum 8458P with BLAS/OpenMP restricted to one thread. The two methods process identical update sequences, and their execution order is alternated to reduce timing bias. Update time includes window maintenance, subtree reconstruction, and cached-cost maintenance, but excludes preprocessing, validation, plotting, recourse computation, and file output. Query times are measured at the 50 checkpoints using repeated timings and the median at each checkpoint.

F.2 Additional Results

Table 1 contains the aggregate results for the primary window size $h = 64$. Dynamic repair is faster than full reconstruction on all five datasets; the ratio of their mean update times averages 2.14×. The mean pointwise cost ratios remain within 0.14% of one. The large finite values of $\widehat{\rho}_{\max}$ on Song and Drift arise when the spectral lower certificate is small, and should be interpreted as certificate looseness rather than as an empirical approximation ratio to the optimum.

Figure 1 illustrates the Dasgupta cost trajectories over the complete update streams. Across all five datasets, our dynamic repair method closely mirrors the full reconstruction baseline. Crucially, this high fidelity is maintained even during transient high-cost regions, confirming that localized repairs preserve global clustering quality without requiring full recomputation.

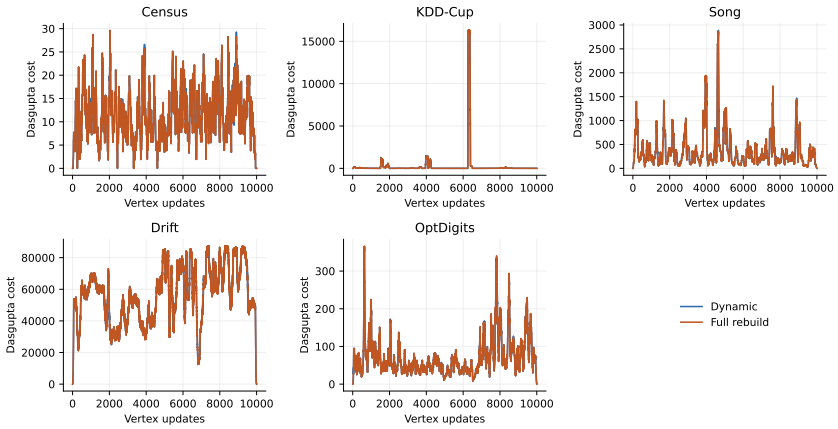

Figure 1: Dasgupta cost over the 10,000 vertex updates at $h = 64$.

Figures 2 and 3 demonstrate that the proposed dynamic repair strategy substantially reduces update time compared to full reconstruction, without compromising query efficiency.

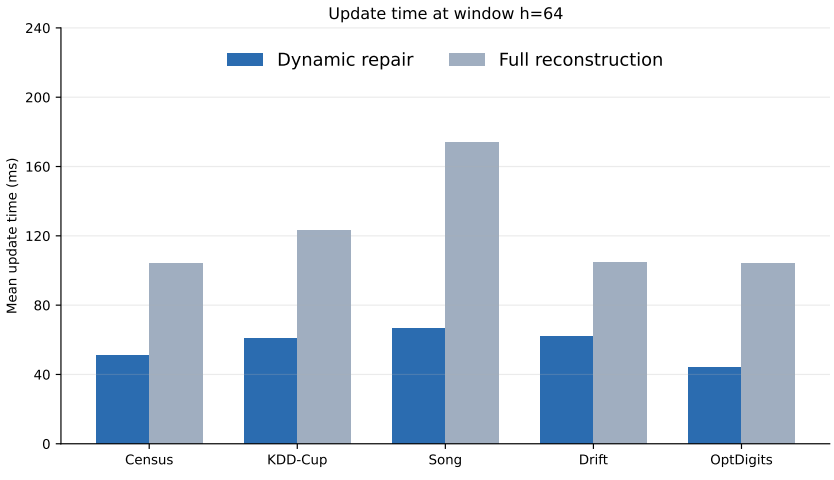

Figure 2: Mean update time at $h = 64$ for dynamic repair and full reconstruction.

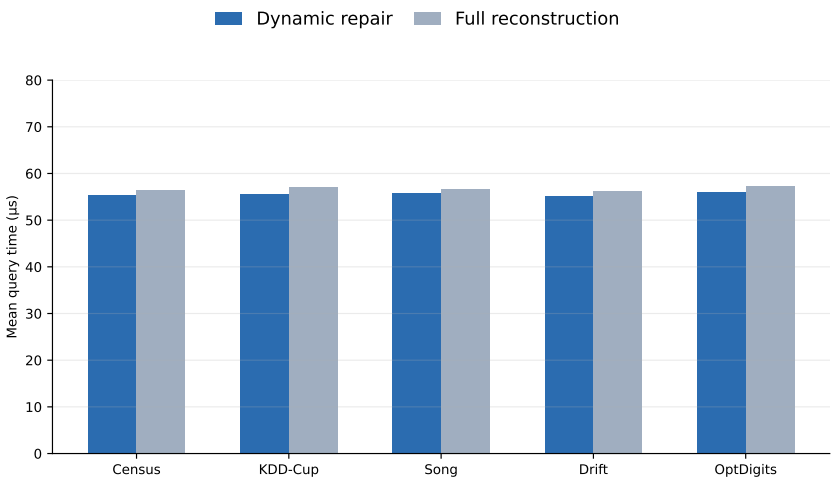

Figure 3: Cached-cost and full-tree query times at $h = 64$.

F.3 Recourse

We also evaluate stable recourse, following related work on dynamic clustering (Bhattacharya et al., 2022; Lattanzi & Vassilvitskii, 2017; Bhattacharya et al., 2024; 2025a;b). For each update, the old and new cluster families are compared only on vertices that remain present. Stable set recourse counts changed clusters, while stable mass recourse weights each changed cluster by its size. Recourse computation is excluded from update time.

Table 2: Mean stable recourse per vertex update at $h = 64$. D: dynamic repair; F: full reconstruction.

Dataset	Set (D)	Set (F)	Mass (D)	Mass (F)
Census	30.506	72.652	197.007	535.448
KDD-Cup	35.518	76.116	211.620	563.524
Song	22.769	54.316	164.855	450.839
Drift	27.924	50.849	186.241	362.539
OptDigits	24.151	64.414	176.934	532.293

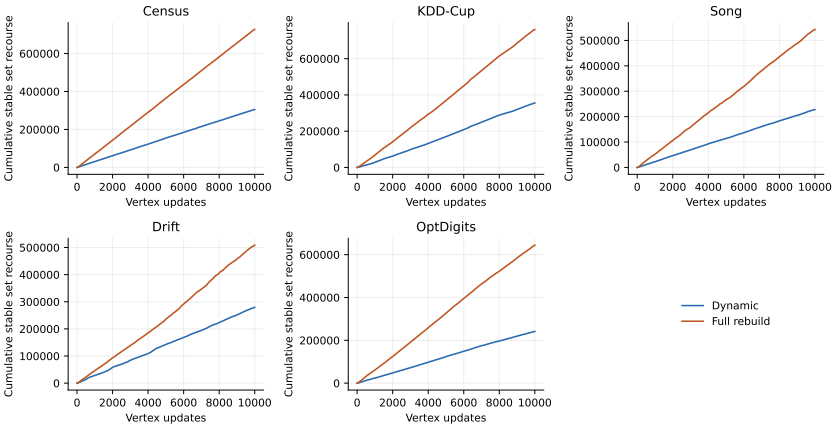

Figure 4: Cumulative stable set recourse at $h = 64$.