2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 979-8-3503-8711-7/24/$31.00 ©2024 IEEE | DOI: 10.1109/IPDPS57955.2024.00049

To Store or Not to Store: a graph theoretical approach for Dataset Versioning

Anxin Guo*†
Computer Science Department
Northwestern University
Evanston, USA
anxinbguo@gmail.com

Jingwei Li*†‡
Industrial Engineering and Operations Research
Columbia University
New York City, USA
jl6639@columbia.edu

Pattara Sukprasert†‡
Databricks
San Francisco, USA
pattara.sk127@gmail.com

Samir Khuller†
Computer Science Department
Northwestern University
Evanston, USA
samir.khuller@northwestern.edu

Amol Deshpande
Department of Computer Science
University of Maryland
College Park, USA
amol@umd.edu

Koyel Mukherjee
Adobe Research
Bangalore, India
komukher@adobe.com

***Abstract*—Dataset Versioning is extremely important for ensuring the reproducibility of results, tracking data changes over time, maintaining quality measures, enabling collaboration, and ensuring legal compliance. In this work, we study the *cost efficient data versioning problem*, where the goal is to optimize the storage and reconstruction (retrieval) costs of data versions, given a graph of datasets as nodes and edges capturing edit/delta information. One central variant we study is MinSum Retrieval (MSR) where the goal is to minimize the total retrieval costs, while keeping the storage costs bounded. This problem (along with its variants) was introduced by Bhattacherjee et al. [VLDB'15]. While such problems are frequently encountered in collaborative tools (e.g., version control systems and data analysis pipelines), to the best of our knowledge, no existing research studies the theoretical aspects of these problems.**

We established, in the full version of this work[1], that the previous best heuristic, LMG (introduced in [VLDB'15]) can perform arbitrarily badly in a simple worst case. Moreover, we show that it is hard to get $o(n)$-approximation for MSR on general graphs even if we relax the storage constraints by an $O(\log n)$ factor. Similar hardness results are shown for other variants. Meanwhile, we propose poly-time approximation schemes for tree-like graphs, motivated by the fact that the graphs arising in practice from typical edit operations are often not arbitrary. As version graphs typically have low treewidth, we further develop new algorithms for bounded treewidth graphs.

Furthermore, we propose two new heuristics and evaluate them empirically. First, we extend LMG by considering more potential "moves", to propose a new heuristic LMG-All. LMG-All consistently outperforms LMG while having comparable run time on a wide variety of datasets, i.e., version graphs. Secondly, we apply our tree algorithms on the minimum-storage arborescence of an instance, yielding algorithms that are qualitatively better than all previous heuristics for MSR, as well as for another variant BoundedMin Retrieval (BMR).

***Index Terms*—Version Control Systems, Data Management, Approximation Algorithm, Combinatorial Optimization.**

I. Introduction

The management and storage of data versions has become increasingly important. As an example, the increasing usage of online collaboration tools allows many collaborators to edit an original dataset simultaneously, producing multiple versions of datasets to be stored daily. Large number of dataset versions also occur often in industry data lakes [1] where huge tabular datasets like product catalogs might require a few records (or rows) to be modified periodically, resulting in a new version for each such modification. Furthermore, in Deep Learning pipelines, multiple versions are generated from the same original data for training and insight generation. At the scale of terabytes or even petabytes, storing and managing all the versions is extremely costly in the aforementioned situations [2]. Therefore, it is no surprise that data version control is emerging as a hot area in the industry [3–8], and even popular cloud solution providers like Databricks are now capturing data lineage information, which helps in effective data version management [9].

In a pioneering paper, Bhattacherjee et al. [10] proposed a model capturing the trade-off between *storage* cost and *retrieval* (recreation) cost. The problems they studied can be defined as follows. Given dataset versions and a subset of the *"deltas"* between them, find a compact representation that minimizes the overall storage as well as the retrieval costs of the versions. This involves a decision for each version: either we *materialize* it (i.e., store it explicitly) or we store a "delta" and rely on edit operations to retrieve the version from another materialized version if necessary. The downside of the latter is that, to retrieve a version that was not materialized, we have to incur a computational overhead that we call *retrieval cost*.

Figure 1, taken from Bhattacherjee et al. [10], illustrates the central point through different storage options. (i) shows the

*These authors contributed equally to this work.
†Supported by a gift from Adobe Research.
‡Part of the work was done while J. Li was an undergraduate student at Northwestern University, and while P. Sukprasert was a Ph.D. candidate at Northwestern University.
[1]https://arxiv.org/abs/2402.11741

1530-2075/24/$31.00 ©2024 IEEE
DOI 10.1109/IPDPS57955.2024.00049

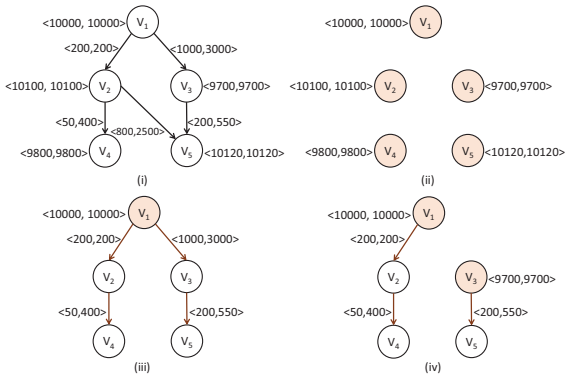

Fig. 1: (i) A version graph over 5 datasets – annotation $\langle$a, b$\rangle$ indicates a storage cost of a and a retrieval cost of b; (ii, iii, iv) three possible storage graphs. The figure is taken from [10]

input graph, with annotated storage and retrieval costs. If the storage size is not a concern, we should store all versions as in (ii). From (iii) to (iv), it is clear that, by materializing v_3, we shorten the retrieval costs of v_3 and v_5.

This retrieval/storage trade-off leads to combinatorial problems of minimizing one type of cost, given a constraint on the other. Moreover, as an objective function, the retrieval cost can be measured by either the maximum or total (or equivalently average) retrieval cost of files. This yields four different optimization problems (Problems 3-6 in Table I). While the first two problems in the table are easy, the other four turn out to be NP-hard and hard to approximate, as we prove in the full version of this paper[2].

Problem Name	Storage	Retrieval
MINIMUM SPANNING TREE	min	$R(v) < \infty,\ \forall v$
SHORTEST PATH TREE	$< \infty$	min $\{\max_v R(v)\}$
MINSUM RETRIEVAL (MSR)	$\leq \mathcal{S}$	min$\{\sum_v R(v)\}$
MINMAX RETRIEVAL (MMR)	$\leq \mathcal{S}$	min $\{\max_v R(v)\}$
BOUNDEDSUM RETRIEVAL (BSR)	min	$\sum_v R(v) \leq \mathcal{R}$
BOUNDEDMAX RETRIEVAL (BMR)	min	$\max_v R(v) \leq \mathcal{R}$

TABLE I: Problems 1-6. Here, $R(v)$ is the retrieval cost of version v, while $\mathcal{R}, \mathcal{S}$ are the retrieval and storage constraints, respectively.

There are some follow-up works on this model [11–13]. However, those either formulate new problems in different use cases [12–14] or implement a system incorporating the feature to store specific versions and deltas [13, 15, 16]. We will discuss this in more detail in Section I-B.

A. Our Contributions

We provide the first set of *approximation algorithms* and *inapproximability results* for the aforementioned optimization problems under various conditions. Our theoretical results also give rise to practical algorithms which perform very well on real-world data.

Due to space constraints, we focus on presenting and providing intuitions for our algorithms in this paper. Please refer to our full paper for proof of the hardness results that are summarized below. In line with our hardness results, we also show in the full paper that a simple path structure causes LMG, the previously best-performing heuristic for MSR [10], to perform arbitrary poorly.

Problem	Graph type	Assumptions	Inapproximability
MSR	arborescence	Triangle inequality $r = s$ on edges[3]	1
	undirected		$1 + \frac{1}{e} - \epsilon$
	general		$\Omega(n)$[4]
MMR	undirected		$2 - \epsilon$
	general		$\log^* n - \omega(1)$
BSR	arborescence		1
	undirected		$(\frac{1}{2} - \epsilon) \log n$
BMR	undirected		$(1 - \epsilon) \log n$

TABLE II: Hardness results

MMR and BMR. We prove that it is hard to approximate MMR within $\log^* n$ factor[5] and it is hard to approximate BMR within $\log n$ factor on general inputs. Meanwhile, in Section III we give a polynomial-time dynamic programming (DP) algorithm for the two problems on *bidirectional trees*, i.e., digraphs whose underlying undirected graph[6] is a tree. These inputs capture the cases where new versions are generated via edit operations.

We also briefly describe an FPTAS (defined below) for MMR, analogous to the main result for MSR in Section IV.

MSR and BSR. We prove that it is hard to design $\big(O(n), O(\log n)\big)$-bicriteria approximation[7] for MSR or $O(\log n)$-approximation for BSR. It is also NP-hard to solve the two problems exactly on trees.

On the other hand, we again use DP to design a fully polynomial-time approximation scheme[8] (FPTAS) for MSR on *bounded treewidth graphs*. These inputs capture many practical settings: bidirectional trees have width 1, series-parallel graphs have width 2, and the GitHub repositories we use in (Section VI) all have low treewidth.[9]

[2]https://arxiv.org/abs/2402.11741

[2]Both are assumptions in previous work [10] that simplify the problems. We also note that our algorithms function even without these assumptions.

[3]This is true even if we relax $\mathcal{S}$ by $O(\log n)$.

[5]$\log^* n$ is "iterated logarithm", defined as the number of times we iteratively take logarithm before the result is at most 1.

[6]The undirected graph formed by disregarding the orientations of the edges.

[7]An (α, β)-bicriteria approximation refers to an algorithm that potentially exceeds the constraint by α times, in order to achieve a β-approximation of the objective. See Section II for an example.

[8]An algorithm that takes the instance and a parameter ε, and in $\text{poly}(n, \varepsilon^{-1})$ time, outputs a $(1 + \varepsilon)$-approximation.

[9]`datasharing`, `styleguide`, and `leetcode` have treewidth 2,3, and 6 respectively.

Graphs	Problems	Algorithm	Approx.
General Digraph	MSR	LMG-All	heuristic
Bounded Treewidth	MSR & MMR	DP-BTW	$1+\epsilon$
	BSR & BMR		$(1, 1+\epsilon)$
Bidirectional Tree	MMR	DP-BMR	exact
	BMR		

TABLE III: Algorithms summary.

New Heuristics. We improved LMG into a more general LMG-All algorithm for solving MSR. LMG-All outperforms LMG in all our experiments and runs faster than LMG on sparse graphs.

Inspired by our algorithms on trees, we also propose two DP heuristics for MSR and BMR. Both algorithms perform extremely well even when the input graph is not tree-like. Moreover, there are known procedures for parallelizing general DP algorithms [17], so our new heuristics are more practical than existing algorithms, which are all sequential.

B. Related Works

*1) **Theory**:* There was little theoretical analysis on the exact problems we study. The optimization problems are first formalized in Bhattacherjee et al. [10], which also compared the effectiveness of several proposed heuristics on both real-world and synthetic data. Zhang et al. [11] followed up by considering a new objective that is a weighted sum of objectives in MSR and MMR. They also modified the heuristics to fit this objective. There are similar concepts, including *Light Approximate Shortest-path Tree (LAST)* [18] and *Shallow-light Tree (SLT)* [19–24]. However, this line of work focuses mainly on undirected graphs and their algorithms do not generalize to the directed case. Among the two problems mentioned, SLT is closely related to MMR and BMR. Here, the goal is to find a tree that is **light** (minimize weight) and **shallow** (bounded depth). To our knowledge, there are only two works that give approximation algorithms for directed shallow-light trees. Chimani and Spoerhase [25] gives a bi-criteria $(1+\epsilon, n^{\epsilon})$-approximation algorithm that runs in polynomial-time. Recently, Ghuge and Nagarajan [26] gave a $O(\frac{\log n}{\log\log n})$-approximation algorithm for *submodular tree orienteering* that runs in quasi-polynomial time. Their algorithm can be adapted into $O(\frac{\log^2 n}{\log\log n})$-approximation for BMR. For MSR, their algorithm gives $\big(O(\frac{\log^2 n}{\log\log n}), O(\frac{\log^2 n}{\log\log n})\big)$-approximation. The idea is to run their algorithm for many rounds, where the objective of each round is to *cover as many nodes as possible*.

*2) **Systems**:* To implement a system captured by our problems, components spanning multiple lines of works are required. For example, to get a graph structure, one has to keep track of history of changes. This is related to the topic of data provenance [27, 28]. Given a graph structure, the question of modeling "deltas" is also of interest. There is a line of work dedicated to studying how to implement `diff` algorithms in different contexts [29–33].

In the more flexible case, one may think of creating deltas without access to the change history. However, computing all possible deltas is too wasteful, hence it is necessary to utilize other approaches to identify similar versions/datasets. Such line of work is known as dataset discovery or dataset similarlity [1, 34–37].

Several follow-up works of Bhattacherjee et al. [10] have implemented systems with a feature that saves only selected versions to reduce redundancy. There are works focusing on version control for relational databases [13, 15, 16, 38–42] and works focusing on graph snapshots [14, 43, 44]. However, since their focus was on designing full-fledged systems, the algorithms they proposed are rather simple heuristics with no theoretical guarantees.

*3) **Usecases**:* In a version control system such as git, our problem is similar to what `git pack` command aims to do.[10] The original heuristic for `git pack`, as described in an IRC log, is to sort objects in particular order and only create deltas between objects in the same window.[11]It is shown in Bhattacherjee et al. [10] that git's heuristic does not work well compared to other methods.

SVN, on the other hand, only stores the most recent version and the deltas to past versions [45]. Other existing data version management systems include [4–8], which offer git-like capabilities suited for different use cases, such as data science pipelines in enterprise setting, machine learning-focused, data lake storage, graph visualization, etc.

Though not directly related to our work, recently, there has been a lot of work exploring algorithmic and systems related optimizations for reducing storage and maintenance costs of data. For example, Mukherjee et al. [2] proposes optimal multi-tiering, compression and data partitioning, along with predicting access patterns for the same. Other works that exploit multi-tiering to optimize performance include e.g., [46–49] and/or costs, e.g., [49–54]. Storage and data placement in a workload aware manner, e.g., [49, 55, 56] and in a device aware manner, e.g., [57–59] have also been explored. [47] combine compression and multi-tiering for optimizing latency.

II. Preliminaries

In this section, the definition of the problems, notations, simplifications, and assumptions will be formally introduced.

In the problems we study, we are given a directed *version graph* $G = (V, E)$, where vertices represent *versions* and edges capture the *deltas* between versions. Every edge e is associated with two weights: storage cost s_e and retrieval cost r_e.[12] The cost of storing e is s_e, and it takes r_e time to retrieve v once we retrieved u. Every vertex v is associated with only the storage cost, s_v, of storing (materializing) the version. Since there is usually a smallest unit of cost in the real world, we will assume $s_v, s_e, r_e \in \mathbb{N}$ for all $v \in V, e \in E$.

[10]https://www.git-scm.com/docs/git-pack-objects

[11]https://github.com/git/git/blob/master/Documentation/technical/pack-heuristics.txt

[12]If $e = (u, v)$, we may use $s_{u,v}$ in place of s_e and $r_{u,v}$ in place of r_e.

To retrieve a version v from a materialized version u, there must be some path $P = \{(u_{i-1}, u_i)\}_{i=1}^{n}$ with $u_0 = u, u_n = v$, such that all edges along this path are stored. In such cases, we say that v is retrieved from materialized u with retrieval cost $R(v) = \sum_{i=1}^{n} r_{(u_{i-1},u_i)}$. In the rest of the paper, we say v is "retrieved from u" if u is in the path to retrieve v, and v is "retrieved from materialized u" if in addition u is materialized.

The general optimization goal is to select vertices $M \subseteq V$ and edges $F \subseteq E$ of *small* size (w.r.t. storage cost s), such that for each $v \in V \setminus M$, there is a *short* path (w.r.t retrieval cost r) from a materialized vertex to v. Formally, we want to minimize (a) total storage cost $\sum_{v \in M} s_v + \sum_{e \in F} s_e$, and (b) total (resp. maximum) retrieval cost $\sum_{v \in V} R(v)$ (resp. $\max_{v \in V} R(v)$).

Since the storage and retrieval objectives are negatively correlated, a natural problem is to constrain one objective and minimize the other. With this in mind, four different problems are formulated, as described by Problems 3-6 in Table I. These problems are originally defined in Bhattacherjee et al. [10], although we use different names for brevity. Since the first two problems are well studied, we do not discuss them further.

Finally, we note that once we have an algorithm for MSR (resp. MMR), we can turn it into an algorithm for BSR (resp. BMR) by binary-searching over the possible values of the constraint. Due to the somewhat exchangeable nature of the storage and constraints in these problems, it's worth considering (α, β)-bicriteria approximations, where we relax the constraint by some β factor in order to achieve a α-approximation. For example, an algorithm A is (α, β)-bicriteria approximation for MSR if it outputs a feasible solution with storage cost $\leq \alpha \cdot \mathcal{S}$ and retrieval cost $\leq \beta \cdot OPT$ where OPT is the retrieval cost of an optimal solution.

III. Exact Algorithm for MMR and BMR on Bidirectional Trees

As discussed previously, we can use an algorithm for BMR to solve for MMR via binary search. Hence, it suffices to focus on BMR, namely, when we are given maximal retrieval constraint $\mathcal{R}$ and want to minimize storage cost.

Let $T = (V, E)$ be a bidirectional tree, i.e., a digraph with two directed edges $(u, v), (v, u) \in E$ corresponding to each edge $\{u, v\} \in E_0$, on some underlying undirected tree (V, E_0). Let $\mathcal{R}$ be the maximum retrieval cost constraint. We can pick any vertex v_0 as root, and orient the tree such that v_0 has no parent, while all other nodes have exactly one parent.

For each $v \in V$, let $T_{[v]}$ denote the subtree of T rooted at v. If v is retrieved from materialized u, we use p_v^u to denote the parent of v on the unique $u - v$ path to retrieve v. We write $p_v^v = v$. We now describe a dynamic programming (DP) algorithm DP-BMR that solves BMR exactly on T.

DP variables. For $u, v \in V$, let DP$[v][u]$ be the minimum storage cost of a *partial solution* on $T_{[v]}$, which satisfies the following: all descendants of v are retrieved from some node in $T_{[v]}$, while v is retrieved from some materialized version u, which is *potentially outside the subtree* $T_{[v]}$. See Figure 2 for an illustration.

Algorithm 1: DP-BMR

```
Input: T, a tree, and R, the max retrieval constraint;
Orient T arbitrarily;
Sort V in reverse topological order;
DP[v][u] ← ∞ for all v, u ∈ V;
for v in V do
    for u in V such that R(u, v) ≤ R do
        if u = v then
            DP[v][u] ← s_v;
        else
            DP[v][u] ← s_{p_v^u, v}, where p_v^u is the node
              preceding v on the path from u to v;
        for w that is a child of v do
            if w in the path from u to v then
                DP[v][u] ← DP[v][u] + DP[w][u];
            else
                DP[v][u] ←
                  DP[v][u] + min{OPT[w], DP[w][u]};
        OPT[v] ← min{DP[v][w] : w ∈ V(T_[v])};
return OPT[v_root];
```

Importantly, when calculating the storage cost for DP$[v][u]$, if u is not a part of $T_{[v]}$, the incident edge (p_v^u, v) is involved in the calculation, while other edges in the $u - v$ path, or the cost to materialize u, are not involved.

Base case. We iterate from the leaves up. Let $R(u, v)$ denote the retrieval cost of the $u - v$ path. For a leaf v, we set DP$[v][v] = s_v$, and DP$[v][u] = s_{(p_v^u, v)}$ for all $u \neq v$ with $R(u, v) \leq \mathcal{R}$. Here, p_v^u is just the parent of v in the tree structure. All choices of u, v such that $R(u, v) > \mathcal{R}$ are infeasible, and we therefore set DP$[v][u] = \infty$ in these cases.

Recurrence. For convenience, we define helper variable OPT$[v]$ to be the minimum storage cost on the subproblem $T_{[v]}$, such that v is either materialized or retrieved from one of its materialized descendants.[13] Formally,

$$\text{OPT}[v] = \min\{\text{DP}[v][w] : w \in V(T_{[v]})\}.$$

For recurrence on DP$[v][u]$ such that $R(u, v) \leq \mathcal{R}$, there are three possible cases of the relationship between v and u (see Figure 2). In each case, we outline what we add to DP$[v][u]$.

Case 1. If $u = v$, we materialize v. Each child w of v can be either materialized, retrieved from their materialized descendants, or retrieved from the materialized v. Note that the storage cost on $T_{[w]}$ is exactly $\min\{\text{OPT}[w], \text{DP}[w][v]\}$, which we will add to the total value of DP$[v][v]$.

Case 2. If $u \in V(T_{[v]}) \setminus \{v\}$, we would store the edge (p_v^u, v). Note that p_v^u is a child of v and hence is also retrieved from the materialized u, so we must add DP$[p_v^u][u]$. We then add $\min\{\text{OPT}[w], \text{DP}[w][u]\}$ for all other children w of v.

[13] In other words, the case where v is retrieved from u outside of $T_{[v]}$, or case 3 in Figure 2, is not considered in this helper variable.

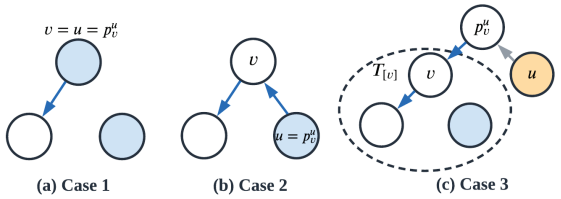

Fig. 2: The 3 cases of DP-BMR, where $u = v, u \in V(T_{[v]})$, and $u \notin V(T_{[v]})$ respectively. The blue nodes and edges are stored in the partial solution.

Case 3. If $u \notin V(T_{[v]})$, we store the edge (p_v^u, v), where p_v^u is now the parent of v in the tree structure. We then add $\min\{\text{OPT}[w], \text{DP}[w][u]\}$ for all children as before.

Output. We output $\text{OPT}[v_{root}]$, the storage cost of the optimal solution. To output the configuration achieving this optimum, we can use the standard procedure where we store the configuration in each DP variable.

Theorem 3.1: BOUNDEDMAX RETRIEVAL is solvable on bidirectional tree instances in $O(n^2)$ time.

The time complexity follows from the observation that each calculation of $DP[v][u]$ in the recurrence takes $O(\deg(v))$ time, and $\sum_u \sum_v \deg(v) = \sum_u O(n) = O(n^2)$. The optimality of this DP can be shown inductively from leaves up, and is ommited due to space limitations.

We note that by binary-searching the constraint value $\mathcal{S}$, this algorithm also solves MINMAX RETRIEVAL on trees.

IV. FPTAS FOR MSR VIA DYNAMIC PROGRAMMING

In this section, we work on MINSUM RETRIEVAL and present a fully polynomial time approximation scheme (FPTAS) on digraphs whose *underlying undirected graph* has bounded treewidth. Similar techniques can be applied to MMR, but we will focus on MSR due to space constraints.

We start by describing a dynamic programming (DP) algorithm on trees in Section IV-A. In Section IV-B, we define all notations necessary for the latter subsection. Finally, in Section IV-C, we show how to extend our DP to the bounded treewidth graphs.

A. Warm-up: Bidirectional Trees

As a warm-up to the more general algorithm, we present an FPTAS for bidirectional tree instances of MSR via DP. This algorithm also inspired a practical heuristic DP-MSR, presented in Section V-B.

Again, we assume the tree has a designated root v_{root} and a parent-child hierarchy. We further assume that the tree is binary, via the standard trick of vertex splitting and adding edges of zero weight if necessary.

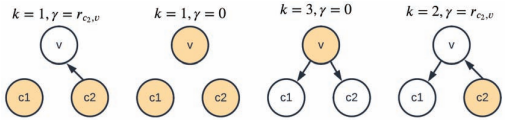

Fig. 3: An illustration of DP variables in Section IV-A

DP variables. We define $\text{DP}[v][k][\gamma][\rho]$ to be the minimum *storage* cost for the subproblem with constraints v, k, γ, ρ such that (with examples illustrated in Figure 3)

1) *Root for subproblem* $v \in V$ is a vertex on the tree; in each iteration, we consider the subtree rooted at v.
2) *Dependency number* $k \in \mathbb{N}$ is the number of versions retrieved from v (including v itself) in the subproblem solution. This is useful when calculating the extra retrieval cost incurred by retrieving v from its parent.
3) *Root retrieval* $\gamma \in \mathbb{N}$ represents the cost of retrieving the subtree root v, if it is retrieved from a materialized descendant. This is useful when calculating the extra retrieval cost incurred by retrieving the parent of v from v. Note that the root retrieval cost will be discretized, as specified later.
4) *Total retrieval* $\rho \in \mathbb{N}$ represents the total retrieval cost of the subsolution. Similar to γ, ρ will also be discretized.

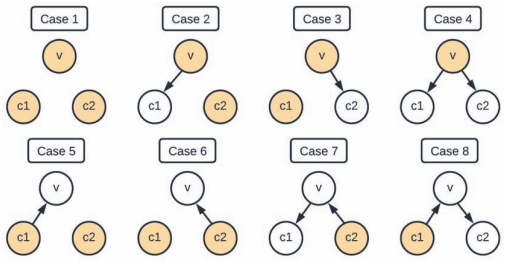

Fig. 4: Eight types of connections on a binary tree. A node is colored if it is either materialized or retrieved from a node *outside* the chart. Otherwise, an uncolored node is retrieved as illustrated with the arrows.

Discretizing retrieval costs. Let $r_{\max} = \max_{e \in E}\{r_e\}$. The possible total retrieval cost ρ is within range $\{0, 1, \ldots, n^2 r_{\max}\}$. To make the DP tractable, we partition this range further and define *approximated retrieval cost* $r'_{u,v}$ for edge $(u, v) \in E$ as follows:

$$r'_{u,v} = \lceil \frac{r_{u,v}}{l} \rceil \quad \text{where } l = \frac{n^2 r_{\max}}{t(\epsilon)}, \; t(\epsilon) = \frac{n^4}{\epsilon},$$

and $t(\epsilon)$ is the number of "ticks" we want to partition the retrieval range $[0, n^2 r_{\max}]$ into. The choice for $t(\epsilon)$ will be specified in the proof for Theorem 4.2. We will work with r' in the rest of the subsection. By an abuse of notation, we still use r for the discretized retrieval for ease of representation.

Base case. For each leaf v, we let $DP[v][1][0][0] = s_v$.

Recurrence step. On each iteration at node v, we consider each target configuration $\text{DP}[v][k][\gamma][\rho]$ under each connection types as illustrated in Figure 4. For each configuration, we go over all corresponding *compatible* partial solutions on $T_{[c_1]}$ and $T_{[c_2]}$.

The recurrence relation for all cases is given in our full paper. Here, we select representative cases and explain the details of calculation below:

*1) **Dealing with dependency**:* When we decide to retrieve any child from v, as in case 4 of Figure 4, the children c_1, c_2 along with all their dependencies now become dependencies of v. The minimum storage cost in case 4 (given $v, k, \gamma = 0, \rho$) is:

$$S_4 = s_v + s_{v,c_1} + s_{v,c_2} - s_{c_1} - s_{c_2} \tag{1}$$
$$+ \min_{\substack{\rho_1+\rho_2=\rho \\ k_1+k_2=k-1}} \Big\{ DP[c_1][k_1][0][\rho_1 - k_1 r_{v,c_1}] \tag{2a}$$
$$+ DP[c_2][k_2][0][\rho_2 - k_2 r_{v,c_2}] \Big\} \tag{2b}$$

In Eq. (2a), v is required to have dependency number k and root retrieval 0. For each $k_1 + k_2 = k-1$, we must go through subproblems where c_1 has dependency number k_1 and c_2 has that of k_2.

Also in Eq. (2a), the choice of ρ_1, ρ_2 determines how we are allocating retrieval costs budget ρ to c_1 and c_2 respectively. Specifically in Eq. (2a) and Eq. (2b), the total retrieval cost allocated to subproblem on $T_{[c_1]}$ is $\rho_1 - k_1 \cdot r_{v,c_1}$ since an extra $k_1 \cdot r_{v,c_1}$ cost is incurred by the edge (v, c_1), as it is used k_1 times by all versions depending on c_1. Similar applies to the subproblem on $T_{[c_2]}$.

Next, we highlight the idea of "invisible" dependency here: in case 2 on $T_{[v]}$, the diffs (v, c_1) and (v, c_2) was not available in any previous recurrence, since v has just been introduced. Therefore, the compatible solution for the subproblems on $T_{[c_1]}$ and $T_{[c_2]}$ have to materialize nodes c_1 and c_2 to ensure they can be retrieved. This explains the $-s_{c_1} - s_{c_2}$ terms in Eq. (1), since these costs are no longer present.

When generalizing the DP onto graphs with bounded treewidth, similarly, the restriction of a global solution does not always result in a feasible partial solution due to the existence of dependencies invisible to the subproblems. We will resolve them using similar ideas.

*2) **Dealing with retrieval**:* In contrast with dependencies, this refers to the case where v is retrieved from one of its children. We take case 5 as an example: given $v, k = 0, \gamma, \rho$,

$$S_5 = s_{c_1,v} + \min_{\rho_1 \le \rho} \Big\{ \min_{k_1} \{ DP[c_1][k_1][\gamma - r_{c_1,v}][\rho_1 - \gamma] \} + \min_{k_2, \gamma'} \{ DP[c_2][k_2][\gamma'][\rho - \rho_1] \} \Big\}$$

In contrast to case 4, here we simply take minimum over the dependency numbers. Since v is retrieved from c_1, the retrieval cost for c_1 has to be $\gamma - r_{c_1,v}$. Note importantly that now we are counting the retrieval cost for v in ρ_1, and so the retrieval cost budget for $T_{[c_1]}$ is now $\rho_1 - \gamma$.

Similarly, we take a minimum on all other unused parameters to get the best storage for case 5.

*3) **Combining the ideas**:* We take case 8 as an example where both retrieval and dependencies are involved. In case 8, v is retrieved from child c_1 (retrieval), and child c_2 is retrieved from v (dependency). Given v, k, γ, ρ, we claim that:

$$S_8 = s_{c_1,v} + s_{v,c_2} - s_{c_2} + \min_{\rho_1+\rho_2=\rho} \Big\{ \min_{k'} \{ DP[c_1][k'][\gamma - r_{c_1,v}][\rho_1 - \gamma] \} + DP[c_2][k-1][0][\rho_2 - (k-1) \cdot (r_2 + \gamma)] \Big\}$$

Note that the c_1 side is identical to that for case 5. In combining both dependency and retrieval cases, there is slight adjustment in the dependency side: since v now might also depend on nodes further down c_1 side, the total extra retrieval cost created by adding edge (v, c_2) becomes $(k-1) \cdot (r_2 + \gamma)$ instead of $(k-1) \cdot (r_2)$.

Output. Finally, with storage constraint $\mathcal{S}$ and root of the tree v_{root}, we output the configuration that outputs the minimum ρ which achieves the following

$$\exists k \le n, \gamma \in \mathbb{N} \quad \text{s.t.} \quad DP[v_{root}][k][\gamma][\rho] \le \mathcal{S}$$

We shall formally state and prove the FPTAS result below.

Lemma 4.1: The DP algorithm outputs a configuration with total retrieval cost at most $\text{OPT} + \epsilon r_{max}$ in $\text{poly}(n, 1/\epsilon)$ time.

Proof: By setting $t(\epsilon) = \frac{n^4}{\epsilon}$, we have $l = \frac{n^2 r_{max}}{t(\epsilon)} = \frac{\epsilon r_{max}}{n^2}$. Note that we can get an approximation of the original retrieval costs by multiplying each r'_e with l. This creates an estimation error of at most l on each edge. Note further that in the optimal solution, at most n^2 edges are materialized, so if ρ^* is the minimal discretized total retrieval cost, we have the total retrieval of the output $\le l\rho^* \le \text{OPT} + n^2 l \le \text{OPT} + \epsilon r_{max}$. ∎

Now we prove the main theorem of this subsection:

Theorem 4.2: For all $\epsilon > 0$, there is a $(1+\epsilon)$-approximation algorithm for MINSUM RETRIEVAL on bidirectional trees that runs in $\text{poly}(n, \frac{1}{\epsilon})$ time.

Proof: Given parameter ϵ, we can use the DP algorithm as a black box and iterate the following for up to n times:

1) Run the DP for the given ϵ on the current graph. Record the output.
2) Let (u, v) be the most retrieval cost-heavy edge. We now set $r_{(u,v)} = 0$ and $s_{(u,v)} = s_v$. If the new graph is infeasible for the given storage constraint, or if all edges have already been modified, exit the loop.

At the end, we output the best out of all recorded outputs. This improves the previous bound when $r_{max} > \text{OPT}$: at some point we will eventually have $r_{max} \le \text{OPT}$, which means the output configuration, if mapped back to the original input, is a feasible $(1+\epsilon)$-approximation. ∎

B. Treewidth-Related Definitions

We now consider a more general class of version graphs: any $G = (V, E)$ whose *underlying undirected graph*[14] G_0 has treewidth bounded by some constant k.

Definition 4.3 (Tree Decomposition [60]): A tree decomposition of undirected $G_0 = (V_0, E_0)$ is a tree $T = (V_T, E_T)$,

[14]As before, this means that $(u, v), (v, u) \in E$ for each undirected edge $\{u, v\} \in E_0$ in G_0.

where each $z \in V_T$ is associated with a subset ("bag") S_z of V_0. The bags must satisfy the following conditions:

1) $\bigcup_{z \in V_T} S_z = V_0$;
2) For each $v \in V_0$, the bags containing v induce a connected subtree of T;
3) For each $(u, v) \in E_0$, there exists $z \in V_T$ such that S_z contains both u and v.

The *width* of a tree decomposition $T = (V_T, E_T)$ is $\max_{z \in V_T} |S_z| - 1$. The *treewidth* of G_0 is the minimum width over all tree decompositions of G_0.

It follows that undirected forests have treewidth 1. We further note that there is also a notion of directed treewidth [61], but it is not suitable for our purpose.

We will WLOG assume a special kind of decomposition:

Definition 4.4 (Nice Tree Decomposition [62]): A nice tree decomposition is a tree decomposition with a designated root, where each node z is one of the following types:

1) A **leaf**, which has no children and whose bag has size 1;
2) A **forget node**, which has one children c, and $S_z \subset S_c$ and $|S_c| = |S_z| + 1$.
3) An **introduce node**, which has one children c, and $S_z \supset S_c$ and $|S_c| + 1 = |S_z|$.
4) A **join**, which has children c_1, c_2, and $S_z = S_{c_1} = S_{c_2}$.

Given a bound k on the treewidth, there are multiple algorithms for calculating a tree decomposition of width k [63–65], or an approximation of k [66–69].

For our case, the algorithm by Bodlaender [64] can be used to compute a tree decomposition in time $2^{O(k^3)} \cdot O(n)$, which is linear if the treewidth k is constant. Given a tree decomposition, we can in $O(|V_0|)$ time find a nice tree decomposition of the same width with $O(k|V_0|)$ nodes [62].

C. Generalized Dynamic Programming

Here we outline the DP for MSR on graphs whose underlying undirected graph G_0 has treewidth at most k.

*1) **DP States**:* Similar to the warm-up, we will do the DP bottom-up on each $z \in V_T$ in the nice tree decomposition T.

Before proceeding, let us define some additional notations. For any bag $z \in V_T$, let $T_{[z]}$ be the induced subtree of T rooted at z. We define $V_{[z]} = \bigcup_{z' \in V(T_z)} S_{z'}$ be the set of vertices in the bags of $T_{[z]}$, including S_z. Following that, $G_{[z]}$ is the induced subgraph of G by vertices $V_{[z]}$.

We now define the *DP states*. At a high level, each state describes some number of *partial solutions* on the subgraph induced by $V_{[z]}$, $G_{[z]}$. When building a complete solution on G from the partial solutions, the state variables should give us *all* the information we need.

Each DP state on $z \in V_T$ consists of a tuple of functions

$$\mathcal{T}_z = (\mathrm{Par}_z, \mathrm{Dep}_z, \mathrm{Ret}_z, \mathrm{Anc}_z)$$

and a natural number ρ_z:

(i) *Parent function* $\mathrm{Par}_z : S_z \mapsto V_{[z]}$ describing the partial solution on $G_{[z]}$, restricted on S_z. If $\mathrm{Par}_z(v) \neq v$ then v will be retrieved through the edge $(\mathrm{Par}_z(v), v)$. If $\mathrm{Par}_z(v) = v$ then v will be materialized.

(ii) *Dependency function* $\mathrm{Dep}_z : S_z \mapsto [n]$. Similar to the dependency parameter in the warm-up, $\mathrm{Dep}_z(v)$ counts the number of nodes in $V_{[z]}$ retrieved through v.

(iii) *Retrieval cost function* $\mathrm{Ret}_z : S_z \mapsto \{0, \ldots, nr_{\max}\}$. Similar to the root retrieval parameter in the warm-up, $\mathrm{Ret}_z(v)$ denotes the retrieval cost of version v in the partial solution on $G_{[z]}$.

(iv) *Ancestor function* $\mathrm{Anc}_z : S_z \mapsto 2^{S_z}$. If $u \in \mathrm{Anc}_z(v)$, then u is retrieved in order to retrieve v in this partial solution, i.e., v is dependent on u. We need this extra information to avoid directed cycles.

(v) ρ_z, the total retrieval cost of the subproblem according to the partial solution. Similar to its counterpart in the warm-up, all retrieval costs would be discretized by the same technique that makes the approximation an FPTAS.

A feasible state on $z \in V_T$ is a pair $(\mathcal{T}_z, \rho_z)$ which correctly describes some partial solution on $G_{[z]}$ whose retrieval cost is exactly ρ_z. Each state is further associated with a storage value $\sigma(\mathcal{T}_z, \rho_z) \in \mathbb{Z}^+$, indicating the minimum storage needed to achieve the state $(\mathcal{T}_z, \rho_z)$ on $G_{[z]}$.

We are now ready to describe how to compute the states.

*2) **Recurrence on leaves**:* For each leaf $z \in V_T$, the only feasible partial solution is to materialize the only vertex v in the leaf bag. We can easily calculate its state and storage cost.

*3) **Recurrence on forget nodes**:* This is also easy: for a forget node z with child c, we have $G_{[z]} = G_{[c]}$, and hence the states on z are simply the restrictions of states on c.

*4) **Recurrence on introduce nodes**:* At introduce node z with child c, we have $S_z = S_c \cup \{v_0\}$ for some "introduced" v_0. Each feasible state $(\mathcal{T}_z, \rho_z)$ on z must correspond to some state $(\mathcal{T}_c, \rho_c)$ on c, which we can calculate as follows:

We first initialize $\mathcal{T}_c$ to be the respective functions in $\mathcal{T}_z$ restricted on S_c. For instance, $\mathrm{Par}_c = \mathrm{Par}_z|_{S_c}$, the restriction of Par_z on domain S_c.

If v_0 is retrieved through $u \in S_c$ according to $\mathcal{T}_z$ ($\mathrm{Par}_z(v_0) = u$), then we remove the dependencies related to v_0 and the retrieval cost incurred on edge (u, v_0). Specifically:

(i) Decrease the value of Dep_c by 1 on all vertices in $\mathrm{Anc}_z(u)$.
(ii) Decrease ρ_c by $\mathrm{Dep}_z(v_0) \cdot \mathrm{Ret}_z(v_0)$.
(iii) Remove $\mathrm{Anc}_z(u)$ from the ancestor functions of all descendants of z.

If v_0 has some child w according to $\mathcal{T}_z$ (namely, $\mathrm{Par}_z(w) = v_0$), then we reverse the *uprooting* process in the warm-up, such that vertex w, which was not a root in $\mathcal{T}_z$, is now a root in $\mathcal{T}_c$. Specifically:

(i) Let $\mathrm{Par}_c(w) = w$.
(ii) Remove v_0 from the ancestor function of w and all its descendants.
(iii) Decrease the retrieval cost function of w and its descendants by $\mathrm{Ret}_z(w)$.
(iv) Decrease ρ_c by $\mathrm{Ret}_z(w) \times \mathrm{Dep}_z(w)$.

Since v could have multiple children, the last procedure is potentially repeated multiple times.

*5) **Recurrence on joins**:* Suppose we are at a join z with children a, b, where $S_z = S_a = S_b$. On a high level, for each state $(\mathcal{T}_z, \rho_z)$ on $G_{[z]}$, we want to find all pairs of states $(\mathcal{T}_a, \rho_a)$ and $(\mathcal{T}_b, \rho_b)$ such that the partial solutions they describe can combine into a partial solution on $G_{[z]}$, as described by $(\mathcal{T}_z, \rho_z)$. The pseudocode of the following functions can be found in the full paper.

Compatibility. The algorithm COMPATIBILITY decides whether $\mathcal{T}_a, \mathcal{T}_b$ are indeed how $\mathcal{T}_z$ looks like when restricted to $G_{[a]}$ and $G_{[b]}$ respectively. If the algorithm returns true, we proceed to calculate the correct value of $\rho_a + \rho_b$ based on this particular restriction.

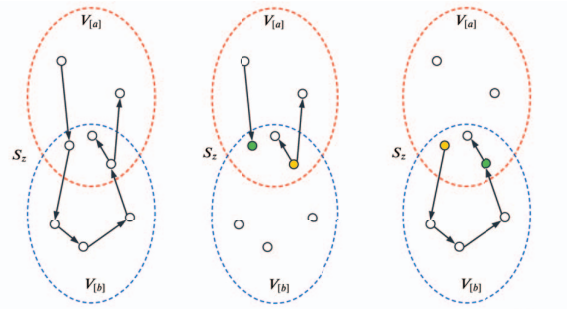

Fig. 5: Illustration for compatibility. Figures (b) and (c) show a pair of compatible configurations on $\mathcal{T}_a$ and $\mathcal{T}_b$ with the configuration on $\mathcal{T}_z$ in (a). The configurations of yellow nodes and green nodes are analyzed in EXTERNAL-RETRIEVAL and EXTERNAL-DEPENDENCY respectively.

Resolving external retrieval. COMPATIBILITY first deals with the vertices that are retrieved from outside S_z. For example, each $v \in S_z$ retrieved from $V_{[a]} \setminus S_z$, like the yellow node in (c) of Figure 5, is instead materialized from $\mathcal{T}_b$'s perspective. To check whether $\mathcal{T}_a$ and $\mathcal{T}_b$ resolve all such cases correctly, we define subroutine EXTERNAL-RETRIEVAL to loop through S_z topologically and calculate the correct $\text{Par}, \text{Ret}, \text{Anc}$ functions for both $\mathcal{T}_a$ and $\mathcal{T}_b$.

Resolving external dependency. The next step in COMPATIBILITY is to check whether the functions $\text{Dep}_a, \text{Dep}_b$ are compatible with Dep_z. Specifically, nodes in S_z could have *external dependencies* in $V_{[a]} \setminus S_z$ and $V_{[b]} \setminus S_z$, an example being the green nodes in Figure 5 and Figure 6. The specific definition of $\text{ExtDep}_a(v)$ is the number of descendants that v have outside S_z, to whom v is the *closest* ancestor in S_z, according to $\mathcal{T}_a$. To see an example, note that only four red nodes are counted towards $\text{ExtDep}_a(A)$ in Figure 6. The functions ExtDep_b and ExtDep_z are defined similarly according to $\mathcal{T}_b$ and $\mathcal{T}_z$.

We need that $\text{ExtDep}_a(v) + \text{ExtDep}_b(v) = \text{ExtDep}_z(v)$ for all $v \in S_z$ in order for $(\mathcal{T}_a, \mathcal{T}_b)$ to be compatible with $\mathcal{T}_z$. To check this, we call EXTERNAL-DEPENDENCY on $\mathcal{T}_z, \mathcal{T}_a, \mathcal{T}_b$ as a subroutine of COMPATIBILITY. We note that this is similar to distributing the dependency number k to the two children in case 4 of Figure 4.

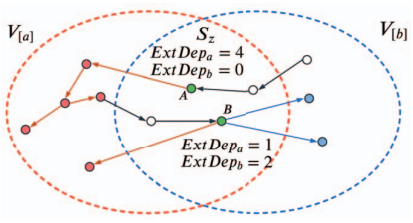

Fig. 6: Illustration for external dependency. Green nodes A and B both have non-zero external dependency, as labeled in the figure.

Calculating ρ. Given that $(\mathcal{T}_a, \mathcal{T}_b)$ are compatible with $\mathcal{T}_z$, we want to find the objective, $\sigma(\mathcal{T}_z, \rho_z)$, with the recurrence relation involving $\sigma(\mathcal{T}_a, \rho_a) + \sigma(\mathcal{T}_b, \rho_b)$ for suitable ρ_a and ρ_b. However, we cannot simply take $\rho_a + \rho_b = \rho_z$ due to the complicated procedure of combining $\mathcal{T}_a, \mathcal{T}_b$ into $\mathcal{T}_z$. We thus implement DISTRIBUTE RETRIEVAL to calculate ρ_Δ such that $\rho_a + \rho_b = \rho_z - \rho_\Delta$ and then iterate through all such ρ_a and ρ_b.

Recurrence relation. Finally, we have all we need for the recurrence relation. For each feasible $(\mathcal{T}_z, \rho_z)$, we take

$$\sigma(\mathcal{T}_z, \rho_z) = \min\{\sigma(\mathcal{T}_a, \rho_a) + \sigma(\mathcal{T}_b, \rho_b) - \textit{uproot} - \textit{overcount}\}$$

where the minimum is taken over all $(\mathcal{T}_a, \mathcal{T}_b)$ that are compatible with $\mathcal{T}_z$ and all $\rho_a + \rho_b = \rho_z - \rho_\Delta$, and where

$$\textit{uproot} = \sum_{v \in U_a} (s_v - s_{\text{Par}_z(v),v}) + \sum_{v \in U_b} (s_v - s_{\text{Par}_z(v),v}), \text{ and}$$

$$\textit{overcount} = \sum_{v \in S_a \cap S_b} s_{\text{Par}_z(v),v}.$$

If k is constant, then the recurrence relation takes $\text{poly}(n)$ time. This is because there are $\text{poly}(n)$ many possible choices of $\mathcal{T}$ and ρ on a, b, z,and it takes $\text{poly}(n)$ steps to check the compatibility of $(\mathcal{T}_a, \mathcal{T}_b)$ with $\mathcal{T}_z$ and compute ρ_Δ.

Output The minimum retrieval cost of a global solution is just $\min\{\rho_z : \exists \mathcal{T}_z, \sigma(\mathcal{T}_z, \rho_z) \le \mathcal{S}\}$ over all feasible $(\mathcal{T}_z, \rho_z)$, where z is the designated root of the nice tree decomposition.

We conclude this section with the following theorem.

Theorem 4.5: For a constant $k \ge 1$, on the set of graphs whose undelying undirected graph has treewidth at most k, MINSUM RETRIEVAL admits an FPTAS.

To see that our algorithm above is an FPTAS for MSR, the proof is almost identical to the proof of Theorem 4.2 (Section IV-A3) once we note that the number of partial solutions on each z is $\text{poly}(n)$.

An FPTAS for MMR arises from a similar procedure. When the objective becomes the maximum retrieval cost, we can use ρ_z to represent the maximum retrieval cost in the partial solution. We then modify $\text{Dep}_z(v)$ to represent the highest retrieval cost among all the nodes that are dependent on v. The recurrence relation is also changed accordingly. One can note

that, like before, the new tuple $\mathcal{T}_z$ contains all the information we need for a subsolution on $G_{[z]}$.

The same algorithms extend to $(1, 1+\epsilon)$ bi-criteria approximation algorithms for BSR and BMR naturally, as the objective and constraint are reversed.

V. Heuristics on MSR and BMR

In this section, we propose three heuristics that are inspired by empirical observations and theoretical results.

A. LMG-All: Improvement over LMG

Here we provide a brief description of the greedy heuristic LMG [10] and the improved LMG-All. We refer to the full paper for pseudocodes and formal definitions.

On a high level, LMG does the following:

1) Find a configuration that minimizes total storage cost.
2) Let V_{active} be the set of vertices not yet materialized, and, if materialized, does not cause the configuration to exceed storage constraint $\mathcal{S}$. If $V_{active} = \varnothing$, output the current configuration.
3) For each $v \in V_{active}$, calculate the cost and benefit of materializing v: storage cost increases by some $S(v)$, but total retrieval cost decreases by some $R(v)$.
4) From all such v, materialize the one that maximizes $\frac{R(v)}{S(v)}$. Go to step 2 and repeat.

Our improved heuristic LMG-All enlarges the scope of the search on each greedy step. Instead of searching for the most efficient version to *materialize*, we explore the payoff of *modifying any single edge*:

1) Find a configuration that minimizes total storage cost.
2) Let $\text{Par}(v)$ be the current parent of v on retrieval path. In addition to V_{active}, Define edge set E_{active} to be the edges that (a) does not cause the configuration to exceed storage constraint $\mathcal{S}$, and (b) does not form cycles, if $(u, v) \in E_{active}$ were to replace $(\text{Par}(v), v)$ in the current configuration. If $V_{active} = E_{active} = \varnothing$, output the current configuration.
3) Calculate cost and benefit of each $v \in V_{active}$ and $e \in E_{active}$. Materialize or store the most cost-effective node or edge. Go to step 2 and repeat.

While LMG-All considers more edges than LMG, it is not obvious that LMG-All always provides a better solution since it is still greedy-based.

B. DP on extracted bi-directional trees

We propose DP heuristics for both MSR and BMR, as inspired by algorithms in Sections III and IV. To ensure a reasonable running time, we extract bidirectional trees[15] from input graphs and run the DP for treewidth 1 on the extracted graph, with the steps below:

1) Fix a node v_{root} as the root. Calculate a minimum spanning arborescence A of the graph G rooted at v_{root}. We use the sum of retrieval and storage costs as weight.
2) Generate a bidirectional tree G' from A. Namely, we have $(u, v), (v, u) \in E(G')$ for each edge $(u, v) \in E(A)$.
3) Run the proposed DP for MSR and BMR on directed trees (see Section IV-A and Section III) with input G'.

In addition, we also implement the following modifications for MSR to further speed up the algorithm:

1) Total *storage* cost (instead of retrieval) is discretized and used as DP variable index, since it has a smaller range than retrieval cost.
2) Geometric discretization is used instead of linear discretization, reducing the number of discretized "ticks."
3) A pruning step is added, where the DP variable discards all subproblem solutions whose storage cost exceeds some threshold. This reduces redundant computations.

All three original features are necessary in the proof for our theoretical results, but in practice, the modified implementations show comparable results but significantly improves the running time.

VI. Experiments for MSR and BMR

In this section, we discuss the experimental setup and results for empirical validation of the algorithms' performance, as compared to previous best-performing heuristic: LMG for MSR, and MP for BMR.[16]

In all figures, the vertical axis (objective and run time) is presented in *logarithmic scale*. Run time is measured in *milliseconds*.

A. Datasets and Construction of Graphs

As in Bhattacherjee et al [10], we experiment on real-world GitHub repositories of varying sizes as datasets. We construct version graphs as follows. Each commit corresponds to a node with its storage cost equal to its size in bytes. Between each pair of parent and child commits, we construct bidirectional edges. The storage and retrieval costs of the edges are calculated, in bytes, based on the actions (such as addition, deletion, and modification of files) required to change one version to the other in the direction of the edge. We use simple `diff` to calculate the deltas, hence the storage and retrieval costs are proportional to each other. Graphs generated this way are called "**natural graphs**" in the rest of the section.

In addition, we also aim to test (1) the cases where the retrieval and storage costs of an edge can greatly differ from each other, and (2) the effect of tree-like shapes of graphs on the performance of algorithms. Therefore, we also conduct experiments on modified graphs in the following two ways:

Random compression. We simulate compression of data by scaling storage cost with a random factor between 0.3 and 1, and increasing the retrieval cost by 20% (to simulate decompression). The resulting storage and retrieval costs are potentially very different.

ER construction. Instead of the naturally constructing edges between each pair of parent and child commits, we

[15]Recall this means a digraph whose underlying undirected graph is a tree, as in Section IV

[16]Our code can be found at https://anonymous.4open.science/r/Graph-Versioning-7343/README.md.

Dataset	#nodes	#edges	avg. cost s_v	avg. cost s_e
datasharing	29	74	7672	395
styleguide	493	1250	1.4×10^6	8659
996.ICU	3189	9210	1.5×10^7	337038
freeCodeCamp	31270	71534	2.5×10^7	14800
LeetCode	246	628	1.7×10^8	1.2×10^7
LeetCode (0.05)	246	3032	1.7×10^8	1.0×10^8
LeetCode (0.2)	246	11932	1.7×10^8	1.0×10^8
LeetCode (1)	246	60270	1.7×10^8	1.0×10^8

TABLE IV: Natural and ER graphs overview.

construct the edges as in an Erdős-Rényi random graph: between each pair (u, v) of versions, with probability p both deltas (u, v) and (v, u) are constructed, and with probability $1 - p$ neither are constructed. The resulting graphs are much less tree-like.[17] We construct ER graphs from the repository LeetCode because it has a moderate size and is the least tree-like. [18]

B. Results in MSR

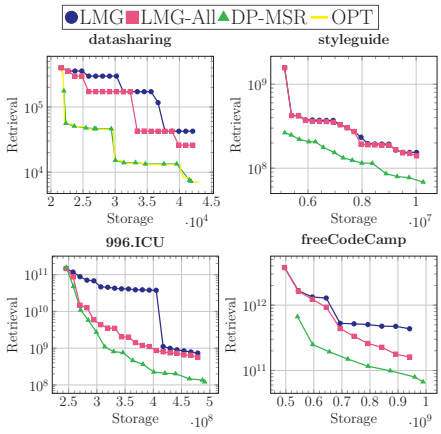

Fig. 7: Performance of MSR algorithms on natural graphs. OPT is obtained by solving an integer linear program (ILP) using Gurobi [71]. ILP takes too long to finish on all graphs except datasharing.

Figure 7, Figure 8, and Figure 9 demonstrate performance of the MSR heuristics on natural graphs, compressed natural graphs, and compressed ER graphs. The running times for the algorithms are shown in Figure 8 and Figure 9. Run time for non-ER graphs exhibit similar trends across most datasets, so only representatives are shown here due to space limits. Also note that DP-MSR generates all data points in a single run, so its running time is shown as a horizontal line over the full range for storage constraint.

[17]ER graphs have treewidth $\Theta(n)$ with high probability if the number of edges per vertex is greater than a small constant [70].

[18]On LeetCode, the average unnatural delta is 10 times more costly than a natural delta. This ratio is around 100 for other repositories.

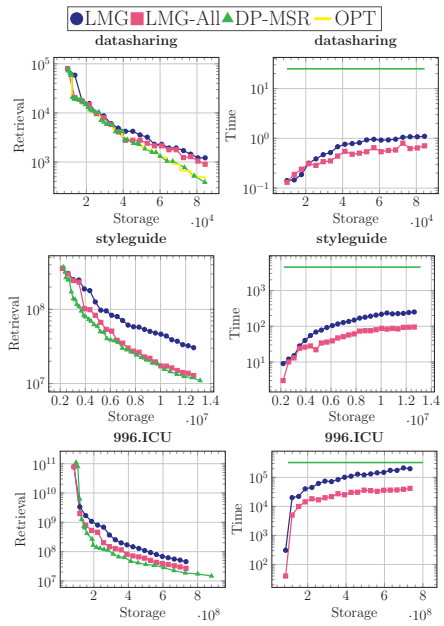

Fig. 8: Performance and run time of MSR algorithms on compressed graphs.

We run DP-MSR with $\epsilon = 0.05$ on most graphs, except $\epsilon = 0.1$ for freeCodeCamp (for the feasibility of run time). The pruning value for DP variables is at twice the minimum storage for uncompressed graphs, and ten times the minimum storage for randomly compressed graphs.

Performance analysis. On most graphs, DP-MSR outperforms LMG-All, which in turn outperforms LMG. This is especially clear on natural version graphs, where DP-MSR solutions are near 1000 times better than LMG solutions on 996.ICU. in Figure 7. On datasharing, DP-MSR almost perfectly matches the optimal solution for all constraint ranges.

On naturally constructed graphs (Figure 7), LMG-All often has comparable performance with LMG when storage constraint is low. This is possibly because both algorithms can only iterate a few times when the storage constraint is almost tight. DP-MSR, on the other hand, performs much better on natural graphs even for low storage constraint.

On graphs with random compression (Figure 8), the dominance of DP in performance over the other two algorithms become less significant. This is anticipated because of the fact that DP only runs on a subgraph of the input graph. Intuitively, most of the information is already contained in a minimum spanning tree when storage and retrieval costs are proportional. Otherwise, the dropped edges may be useful. (They could have large retrieval but small storage, and vice versa.)

Finally, LMG's performance relative to our new algorithms is much worse on ER graphs. This may be due to the fact that LMG cannot look at non-auxiliary edges once the minimum arborescence is initialized, and hence losing most of the information brought by the extra edges (see Figure 9).

Run time analysis. For all natural graphs, we observe that

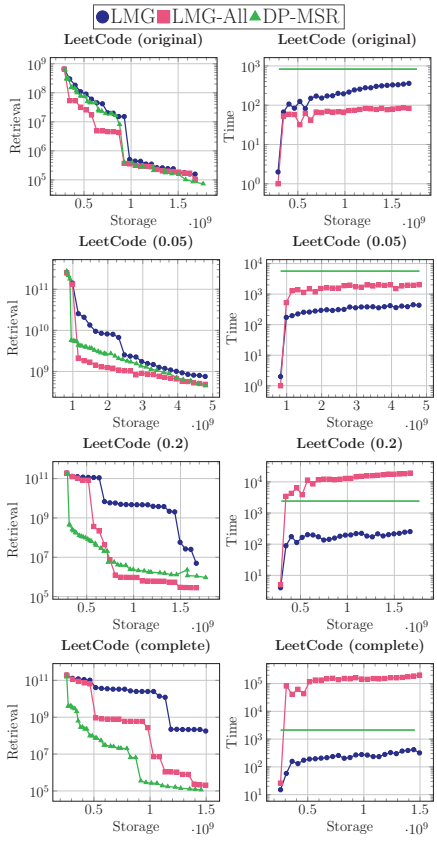

Fig. 9: Performance and run time of MSR algorithms on compressed ER graphs.

LMG-All is generally faster than LMG (Figure 8), especially on large natural graphs, which was surprising considering the almost identical structures in their implementations. Possibly, this could be due to LMG making bigger, more expensive changes on each iteration (materializing a node with many dependencies, for instance) as compared to LMG-All.

As expected, though, LMG-All takes much more time than the other two algorithms on denser ER graphs (Figure 9), due to the large number of edges.

DP-MSR is often slower than LMG, except when ran on natural constructions of large graphs (Figure 8). However, unlike LMG and LMG-All, the DP algorithm returns a whole spectrum of solutions at once, so it is difficult to make a direct comparison. We also note that the run time of DP heavily depends on the choice of ϵ and the storage pruning bound. Hence, the user can trade-off the run time with solution's qualities by parameterize the algorithm with coarser configurations.

C. Results in BMR

As compared to MSR algorithms, the performance and run time of our BMR algorithms are much more predictable. They exhibit similar trends across different graph constructions, including the non-tree-like ER graphs, surprisingly. Here, we present the results on natural graphs only (Figure 10) due to space limitation.

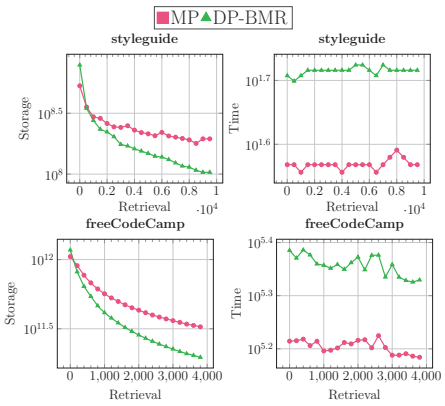

Fig. 10: Performance and run time of BMR algorithms on natural version graphs.

Performance analysis. For every graph we tested, DP-BMR outperforms MP on most of the retrieval constraint ranges, and the performance gap increases as the retrieval constraint increases. We also observe that DP-BMR performs worse than MP when the retrieval constraint is zero. This is because the bidirectional tree have fewer edges than the original graph. (Recall that the same behavior happened for DP-MSR when running on compressed graphs)

We also note that, unlike MP, the objective value of DP-BMR solution monotonically decreases with respect to retrieval constraint. This is again expected since these are optimal solutions of the problem on the bidirectional tree.

Run time analysis. For all graphs, the run times of DP-BMR and MP are comparable within a constant factor. This is true with varying graph shapes and construction methods in all our experiments, and representative data is exhibited in Figure 10. Unlike LMG and LMG-All, their run times do not change much with varying constraint values.

Overall Evaluation For MSR, we recommend always using one of LMG-All and DP-MSR in place of LMG for practical use. On sparse graphs, LMG-All dominates LMG both in performance and run time. DP-MSR can also provide a frontier of better solutions in a reasonable amount of time, regardless of the input.

For BMR, DP-BMR usually outperforms MP, except when the retrieval constraint is close to zero. Therefore, we recommend using DP in most situations.

VII. Conclusion

In this paper, we developed fully polynomial time approximation algorithms for graphs with bounded treewidth. This often captures the typical manner in which edit operations are applied on versions. For practical use, we extracted the idea behind this approach as well as previous LMG approach, and developed heuristics which significantly improved both the performance and run time in experiments, while potentially allowing for parallelization.

References

[1] F. Nargesian, E. Zhu, R. J. Miller, K. Q. Pu, and P. C. Arocena, "Data lake management: Challenges and opportunities," *Proc. VLDB Endow.*, vol. 12, no. 12, pp. 1986–1989, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol12/p1986-nargesian.pdf

[2] K. Mukherjee, R. Shah, S. K. Saini, K. Singh, Khushi, H. Kesarwani, K. Barnwal, and A. Chauhan, "Towards optimizing storage costs on the cloud," in *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 2023, pp. 2919–2932. [Online]. Available: https://doi.org/10.1109/ICDE55515.2023.00223

[3] "Git," https://github.com/git/git, 2005, last accessed: 13-Oct-22.

[4] "Pachyderm," https://github.com/pachyderm/pachyderm, 2016, last accessed: 13-Oct-22.

[5] "DVC," https://github.com/iterative/dvc, 2017, last accessed: 13-Oct-22.

[6] "TerminusDB," https://github.com/terminusdb/terminusdb, 2019, last accessed: 13-Oct-22.

[7] "LakeFS," https://github.com/treeverse/lakeFS, 2020, last accessed: 13-Oct-22.

[8] "Dolt," https://github.com/dolthub/dolt, 2019, last accessed: 13-Oct-22.

[9] P. Roome, T. Feng, and S. Thakur, "Announcing the availability of data lineage with unity catalog," https://www.databricks.com/blog/2022/06/08/announcing-the-availability-of-data-lineage-with-unity-catalog.html, 2022, last accessed: 13-Oct-22.

[10] S. Bhattacherjee, A. Chavan, S. Huang, A. Deshpande, and A. G. Parameswaran, "Principles of dataset versioning: Exploring the recreation/storage tradeoff," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1346–1357, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol8/p1346-bhattacherjee.pdf

[11] Y. Zhang, H. Liu, C. Jin, and Y. Guo, "Storage and recreation trade-off for multi-version data management," in *Web and Big Data - Second International Joint Conference, APWeb-WAIM 2018, Macau, China, July 23-25, 2018, Proceedings, Part II*, ser. Lecture Notes in Computer Science, Y. Cai, Y. Ishikawa, and J. Xu, Eds., vol. 10988. Springer, 2018, pp. 394–409. [Online]. Available: https://doi.org/10.1007/978-3-319-96893-3_30

[12] B. Derakhshan, A. R. Mahdiraji, Z. Kaoudi, T. Rabl, and V. Markl, "Materialization and reuse optimizations for production data science pipelines," in *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Z. G. Ives, A. Bonifati, and A. E. Abbadi, Eds. ACM, 2022, pp. 1962–1976. [Online]. Available: https://doi.org/10.1145/3514221.3526186

[13] S. Huang, L. Xu, J. Liu, A. J. Elmore, and A. G. Parameswaran, "$\varvec{\textsc {Orpheus}}$db: bolt-on versioning for relational databases (extended version)," *VLDB J.*, vol. 29, no. 1, pp. 509–538, 2020. [Online]. Available: https://doi.org/10.1007/s00778-019-00594-5

[14] N. N. Manne, S. Satpati, T. Malik, A. Bagchi, A. Gehani, and A. Chaudhary, "CHEX: multiversion replay with ordered checkpoints," *Proc. VLDB Endow.*, vol. 15, no. 6, pp. 1297–1310, 2022. [Online]. Available: https://www.vldb.org/pvldb/vol15/p1297-malik.pdf

[15] S. Wang, T. T. A. Dinh, Q. Lin, Z. Xie, M. Zhang, Q. Cai, G. Chen, B. C. Ooi, and P. Ruan, "Forkbase: An efficient storage engine for blockchain and forkable applications," *Proc. VLDB Endow.*, vol. 11, no. 10, pp. 1137–1150, 2018. [Online]. Available: http://www.vldb.org/pvldb/vol11/p1137-wang.pdf

[16] M. E. Schüle, L. Karnowski, J. Schmeißer, B. Kleiner, A. Kemper, and T. Neumann, "Versioning in main-memory database systems: From musaeusdb to tardisdb," in *Proceedings of the 31st International Conference on Scientific and Statistical Database Management, SSDBM 2019, Santa Cruz, CA, USA, July 23-25, 2019*, C. Maltzahn and T. Malik, Eds. ACM, 2019, pp. 169–180. [Online]. Available: https://doi.org/10.1145/3335783.3335792

[17] A. D. Stivala, P. J. Stuckey, M. G. de la Banda, M. V. Hermenegildo, and A. Wirth, "Lock-free parallel dynamic programming," *J. Parallel Distributed Comput.*, vol. 70, no. 8, pp. 839–848, 2010. [Online]. Available: https://doi.org/10.1016/j.jpdc.2010.01.004

[18] S. Khuller, B. Raghavachari, and N. E. Young, "Balancing minimum spanning and shortest path trees," in *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA*, V. Ramachandran, Ed. ACM/SIAM, 1993, pp. 243–250. [Online]. Available: http://dl.acm.org/citation.cfm?id=313559.313760

[19] G. Kortsarz and D. Peleg, "Approximating shallow-light trees (extended abstract)," in *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, 5-7 January 1997, New Orleans, Louisiana, USA*, M. E. Saks, Ed. ACM/SIAM, 1997, pp. 103–110. [Online]. Available: http://dl.acm.org/citation.cfm?id=314161.314191

[20] M. T. Hajiaghayi, G. Kortsarz, and M. R. Salavatipour, "Approximating buy-at-bulk and shallow-light *k*-steiner trees," *Algorithmica*, vol. 53, no. 1, pp. 89–103, 2009. [Online]. Available: https://doi.org/10.1007/s00453-007-9013-x

[21] B. Haeupler, D. E. Hershkowitz, and G. Zuzic, "Tree embeddings for hop-constrained network design," in *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, S. Khuller and V. V. Williams, Eds. ACM, 2021, pp. 356–369. [Online]. Available: https://doi.org/10.1145/3406325.3451053

[22] M. V. Marathe, R. Ravi, R. Sundaram, S. S. Ravi,

D. J. Rosenkrantz, and H. B. H. III, "Bicriteria network design problems," *J. Algorithms*, vol. 28, no. 1, pp. 142–171, 1998. [Online]. Available: https://doi.org/10.1006/jagm.1998.0930

[23] R. Ravi, "Rapid rumor ramification: Approximating the minimum broadcast time (extended abstract)," in *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*. IEEE Computer Society, 1994, pp. 202–213. [Online]. Available: https://doi.org/10.1109/SFCS.1994.365693

[24] M. R. Khani and M. R. Salavatipour, "Improved approximations for buy-at-bulk and shallow-light k-steiner trees and (k, 2)-subgraph," *J. Comb. Optim.*, vol. 31, no. 2, pp. 669–685, 2016. [Online]. Available: https://doi.org/10.1007/s10878-014-9774-5

[25] M. Chimani and J. Spoerhase, "Network design problems with bounded distances via shallow-light steiner trees," in *32nd International Symposium on Theoretical Aspects of Computer Science, STACS 2015, March 4-7, 2015, Garching, Germany*, ser. LIPIcs, E. W. Mayr and N. Ollinger, Eds., vol. 30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 238–248. [Online]. Available: https://doi.org/10.4230/LIPIcs.STACS.2015.238

[26] R. Ghuge and V. Nagarajan, "Quasi-polynomial algorithms for submodular tree orienteering and other directed network design problems," in *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, S. Chawla, Ed. SIAM, 2020, pp. 1039–1048. [Online]. Available: https://doi.org/10.1137/1.9781611975994.63

[27] P. Buneman, S. Khanna, and W. C. Tan, "Data provenance: Some basic issues," in *Foundations of Software Technology and Theoretical Computer Science, 20th Conference, FST TCS 2000 New Delhi, India, December 13-15, 2000, Proceedings*, ser. Lecture Notes in Computer Science, S. Kapoor and S. Prasad, Eds., vol. 1974. Springer, 2000, pp. 87–93. [Online]. Available: https://doi.org/10.1007/3-540-44450-5_6

[28] Y. L. Simmhan, B. Plale, D. Gannon *et al.*, "A survey of data provenance techniques," 2005.

[29] J. J. Hunt, K. Vo, and W. F. Tichy, "Delta algorithms: An empirical analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 7, no. 2, pp. 192–214, 1998. [Online]. Available: https://doi.org/10.1145/279310.279321

[30] R. C. Burns and D. D. E. Long, "In-place reconstruction of delta compressed files," in *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998*, B. A. Coan and Y. Afek, Eds. ACM, 1998, pp. 267–275. [Online]. Available: https://doi.org/10.1145/277697.277747

[31] W. Xia, H. Jiang, D. Feng, L. Tian, M. Fu, and Y. Zhou, "Ddelta: A deduplication-inspired fast delta compression approach," *Perform. Evaluation*, vol. 79, pp. 258–272, 2014. [Online]. Available: https://doi.org/10.1016/j.peva.2014.07.016

[32] J. MacDonald, "File system support for delta compression," Ph.D. dissertation, Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkley, 2000.

[33] D. T. N. M. T. Suel, "zdelta: An efficient delta compression tool," 2002.

[34] Y. Jayawardana and S. Jayarathna, "DFS: A dataset file system for data discovering users," in *19th ACM/IEEE Joint Conference on Digital Libraries, JCDL 2019, Champaign, IL, USA, June 2-6, 2019*, M. Bonn, D. Wu, J. S. Downie, and A. Martaus, Eds. IEEE, 2019, pp. 355–356. [Online]. Available: https://doi.org/10.1109/JCDL.2019.00068

[35] R. C. Fernandez, Z. Abedjan, F. Koko, G. Yuan, S. Madden, and M. Stonebraker, "Aurum: A data discovery system," in *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 2018, pp. 1001–1012. [Online]. Available: https://doi.org/10.1109/ICDE.2018.00094

[36] D. Brickley, M. Burgess, and N. F. Noy, "Google dataset search: Building a search engine for datasets in an open web ecosystem," in *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, L. Liu, R. W. White, A. Mantrach, F. Silvestri, J. J. McAuley, R. Baeza-Yates, and L. Zia, Eds. ACM, 2019, pp. 1365–1375. [Online]. Available: https://doi.org/10.1145/3308558.3313685

[37] A. Bogatu, A. A. A. Fernandes, N. W. Paton, and N. Konstantinou, "Dataset discovery in data lakes," in *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 2020, pp. 709–720. [Online]. Available: https://doi.org/10.1109/ICDE48307.2020.00067

[38] J. Brown and N. Weber, "DSDB: an open-source system for database versioning & curation," in *ACM/IEEE Joint Conference on Digital Libraries, JCDL 2021, Champaign, IL, USA, September 27-30, 2021*, J. S. Downie, D. McKay, H. Suleman, D. M. Nichols, and F. Poursardar, Eds. IEEE, 2021, pp. 299–307. [Online]. Available: https://doi.org/10.1109/JCDL52503.2021.00044

[39] A. Chavan and A. Deshpande, "DEX: query execution in a delta-based storage system," in *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, Eds. ACM, 2017, pp. 171–186. [Online]. Available: https://doi.org/10.1145/3035918.3064056

[40] M. Maddox, D. Goehring, A. J. Elmore, S. Madden, A. G. Parameswaran, and A. Deshpande, "Decibel: The relational dataset branching system," *Proc. VLDB Endow.*, vol. 9, no. 9, pp. 624–635, 2016. [Online]. Available: http://www.vldb.org/pvldb/vol9/p624-maddox.pdf

[41] A. P. Bhardwaj, S. Bhattacherjee, A. Chavan,

A. Deshpande, A. J. Elmore, S. Madden, and A. G. Parameswaran, "Datahub: Collaborative data science & dataset version management at scale," in *Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org, 2015. [Online]. Available: http://cidrdb.org/cidr2015/Papers/CIDR15_Paper18.pdf

[42] A. Seering, P. Cudré-Mauroux, S. Madden, and M. Stonebraker, "Efficient versioning for scientific array databases," in *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, A. Kementsietsidis and M. A. V. Salles, Eds. IEEE Computer Society, 2012, pp. 1013–1024. [Online]. Available: https://doi.org/10.1109/ICDE.2012.102

[43] T. Ying, H. Chen, and H. Jin, "Pensieve: Skewness-aware version switching for efficient graph processing," in *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, Eds. ACM, 2020, pp. 699–713. [Online]. Available: https://doi.org/10.1145/3318464.3380590

[44] U. Khurana and A. Deshpande, "Efficient snapshot retrieval over historical graph data," in *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, C. S. Jensen, C. M. Jermaine, and X. Zhou, Eds. IEEE Computer Society, 2013, pp. 997–1008. [Online]. Available: https://doi.org/10.1109/ICDE.2013.6544892

[45] W. Nagel, "Subversion: not just for code anymore," *Linux Journal*, vol. 2006, no. 143, p. 10, 2006.

[46] A. Kougkas, H. Devarajan, and X. Sun, "Hermes: a heterogeneous-aware multi-tiered distributed I/O buffering system," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2018, Tempe, AZ, USA, June 11-15, 2018*, M. Zhao, A. Chandra, and L. Ramakrishnan, Eds. ACM, 2018, pp. 219–230. [Online]. Available: https://doi.org/10.1145/3208040.3208059

[47] H. Devarajan, A. Kougkas, L. Logan, and X. Sun, "Hcompress: Hierarchical data compression for multi-tiered storage environments," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA, May 18-22, 2020*. IEEE, 2020, pp. 557–566. [Online]. Available: https://doi.org/10.1109/IPDPS47924.2020.00064

[48] H. Devarajan, A. Kougkas, and X. Sun, "Hfetch: Hierarchical data prefetching for scientific workflows in multi-tiered storage environments," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA, May 18-22, 2020*. IEEE, 2020, pp. 62–72. [Online]. Available: https://doi.org/10.1109/IPDPS47924.2020.00017

[49] Y. Cheng, M. S. Iqbal, A. Gupta, and A. R. Butt, "CAST: tiering storage for data analytics in the cloud," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2015, Portland, OR, USA, June 15-19, 2015*, T. Kielmann, D. Hildebrand, and M. Taufer, Eds. ACM, 2015, pp. 45–56. [Online]. Available: https://doi.org/10.1145/2749246.2749252

[50] A. Erradi and Y. Mansouri, "Online cost optimization algorithms for tiered cloud storage services," *J. Syst. Softw.*, vol. 160, 2020. [Online]. Available: https://doi.org/10.1016/j.jss.2019.110457

[51] M. Liu, L. Pan, and S. Liu, "To transfer or not: An online cost optimization algorithm for using two-tier storage-as-a-service clouds," *IEEE Access*, vol. 7, pp. 94 263–94 275, 2019. [Online]. Available: https://doi.org/10.1109/ACCESS.2019.2928844

[52] ——, "Keep hot or go cold: A randomized online migration algorithm for cost optimization in staas clouds," *IEEE Trans. Netw. Serv. Manag.*, vol. 18, no. 4, pp. 4563–4575, 2021. [Online]. Available: https://doi.org/10.1109/TNSM.2021.3096533

[53] W. Si, L. Pan, and S. Liu, "A cost-driven online auto-scaling algorithm for web applications in cloud environments," *Knowl. Based Syst.*, vol. 244, p. 108523, 2022. [Online]. Available: https://doi.org/10.1016/j.knosys.2022.108523

[54] R. Kinoshita, S. Imamura, L. Vogel, S. Kazama, and E. Yoshida, "Cost-performance evaluation of heterogeneous tierless storage management in a public cloud," in *Ninth International Symposium on Computing and Networking, CANDAR 2021, Matsue, Japan, November 23-26, 2021*. IEEE, 2021, pp. 121–126. [Online]. Available: https://doi.org/10.1109/CANDAR53791.2021.00024

[55] A. Anwar, Y. Cheng, A. Gupta, and A. R. Butt, "Taming the cloud object storage with MOS," in *Proceedings of the 10th Parallel Data Storage Workshop, PDSW 2015, Austin, Texas, USA, November 15, 2015*, A. R. Butt and J. F. Lofstead, Eds. ACM, 2015, pp. 7–12. [Online]. Available: https://doi.org/10.1145/2834976.2834980

[56] ——, "MOS: workload-aware elasticity for cloud object stores," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2016, Kyoto, Japan, May 31 - June 04, 2016*, H. Nakashima, K. Taura, and J. Lange, Eds. ACM, 2016, pp. 177–188. [Online]. Available: https://doi.org/10.1145/2907294.2907304

[57] L. Vogel, A. van Renen, S. Imamura, V. Leis, T. Neumann, and A. Kemper, "Mosaic: A budget-conscious storage engine for relational database systems," *Proc. VLDB Endow.*, vol. 13, no. 11, pp. 2662–2675, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p2662-vogel.pdf

[58] R. Lasch, T. Legler, N. May, B. Scheirle, and K. Sattler, "Cost modelling for optimal data placement in heterogeneous main memory," *Proc. VLDB Endow.*, vol. 15,

no. 11, pp. 2867–2880, 2022. [Online]. Available: https://www.vldb.org/pvldb/vol15/p2867-lasch.pdf

[59] R. Lasch, R. Schulze, T. Legler, and K. Sattler, "Workload-driven placement of column-store data structures on DRAM and NVM," in *Proceedings of the 17th International Workshop on Data Management on New Hardware, DaMoN 2021, 21 June 2021, Virtual Event, China*, D. Porobic and S. Blanas, Eds. ACM, 2021, pp. 5:1–5:8. [Online]. Available: https://doi.org/10.1145/3465998.3466008

[60] U. Bertelè and F. Brioschi, "On non-serial dynamic programming," *J. Comb. Theory, Ser. A*, vol. 14, no. 2, pp. 137–148, 1973. [Online]. Available: https://doi.org/10.1016/0097-3165(73)90016-2

[61] T. Johnson, N. Robertson, P. D. Seymour, and R. Thomas, "Directed tree-width," *J. Comb. Theory, Ser. B*, vol. 82, no. 1, pp. 138–154, 2001. [Online]. Available: https://doi.org/10.1006/jctb.2000.2031

[62] H. L. Bodlaender, "A partial *k*-arboretum of graphs with bounded treewidth," *Theor. Comput. Sci.*, vol. 209, no. 1-2, pp. 1–45, 1998. [Online]. Available: https://doi.org/10.1016/S0304-3975(97)00228-4

[63] ——, "A linear time algorithm for finding tree-decompositions of small treewidth," in *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, S. R. Kosaraju, D. S. Johnson, and A. Aggarwal, Eds. ACM, 1993, pp. 226–234. [Online]. Available: https://doi.org/10.1145/167088.167161

[64] S. Arnborg, D. G. Corneil, and A. Proskurowski, "Complexity of finding embeddings in a k-tree," *SIAM Journal on Algebraic Discrete Methods*, vol. 8, no. 2, pp. 277–284, 1987. [Online]. Available: https://doi.org/10.1137/0608024

[65] F. V. Fomin, I. Todinca, and Y. Villanger, "Large induced subgraphs via triangulations and CMSO," *SIAM J. Comput.*, vol. 44, no. 1, pp. 54–87, 2015. [Online]. Available: https://doi.org/10.1137/140964801

[66] M. Belbasi and M. Fürer, "Finding all leftmost separators of size $\le k$," in *Combinatorial Optimization and Applications - 15th International Conference, COCOA 2021, Tianjin, China, December 17-19, 2021, Proceedings*, ser. Lecture Notes in Computer Science, D. Du, D. Du, C. Wu, and D. Xu, Eds., vol. 13135. Springer, 2021, pp. 273–287. [Online]. Available: https://doi.org/10.1007/978-3-030-92681-6_23

[67] F. V. Fomin, D. Lokshtanov, M. Pilipczuk, S. Saurabh, and M. Wrochna, "Fully polynomial-time parameterized computations for graphs and matrices of low treewidth," in *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, P. N. Klein, Ed. SIAM, 2017, pp. 1419–1432. [Online]. Available: https://doi.org/10.1137/1.9781611974782.92

[68] T. Korhonen, "A single-exponential time 2-approximation algorithm for treewidth," in *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*. IEEE, 2021, pp. 184–192. [Online]. Available: https://doi.org/10.1109/FOCS52979.2021.00026

[69] U. Feige, M. T. Hajiaghayi, and J. R. Lee, "Improved approximation algorithms for minimum-weight vertex separators," in *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, H. N. Gabow and R. Fagin, Eds. ACM, 2005, pp. 563–572. [Online]. Available: https://doi.org/10.1145/1060590.1060674

[70] Y. Gao, "Treewidth of erdős-rényi random graphs, random intersection graphs, and scale-free random graphs," *Discret. Appl. Math.*, vol. 160, no. 4-5, pp. 566–578, 2012. [Online]. Available: https://doi.org/10.1016/j.dam.2011.10.013

[71] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2022. [Online]. Available: https://www.gurobi.com