数据集版本管理：探讨恢复/存储的权衡问题

柯洲彬

摘要

协作式数据科学和分析的相对便利性导致了许多科学和商业领域中同一数据集的数千个或数百万个版本的激增，这些数据集是在许多用户的数据分析的不同阶段获得或构建的，而且往往是在很长一段时间内。管理、存储和重新创建这些数据集版本是一项非艰巨的任务。这里的基本挑战是存储-创建的权衡：我们使用的存储越多，重新创建或检索版本就越快，而我们使用的存储越少，重新创建或检索版本就越慢。本文探讨这种权衡：在不同的设置下提出了六个问题，并从延迟约束调度的技术和生成树文献中提出了一套廉价的启发式方法来解决这些问题。利用真实数据集进行实验，结果表明这些启发式方法在实际的数据集版本管理场景中提供了有效的解决方案。

关键词：多版本数据管理；存储和恢复的权衡；科学集数据管理

1 引言

科学领域的数据集在不断地扩大。例如下一代的望远镜天空观测，如大型同步观测望远镜（LSST）[1]，每年将产生 10 到 100PB 的图像和衍生数据；又例如地球微生物组计划[2]，预计将在他们的元基因组学工作中产生 2.4PB 的数据。因此，科学家需要强大的数据管理系统（DBMS）来帮助他们管理这些大规模数据。然而，对于科学家而言，一个数据库管理系统除了能高效存储庞大数据集外，还需要能够创建、归档和搜索数据集的不同版本。此外，在一个科研项目中，科研团队会对某一版本数据进行加工处理[3-4]，其反复分析得到的中间结果（即中间数据集）会被存储起来以供将来使用。因此，数据规模的日益增长导致了数据集管理和检索出现了严重的问题。数据库管理系统一方面必须能够跟踪数据集的版本，并且能够根据需要重新创建它们；同时，也必须通过减少冗余数据来尽量降低存储成本。

在本文中，我们关注的是存储成本和恢复成本的权衡问题。具体来说，本文所探讨的问题是：给定一个数据集的集合以及连接它们的有向版本图，如何布局这些数据集使得总体存储量以及检索他们的恢复成本最小。尽管探讨存储和检索之间的权衡是一个基本问题，但对这种权衡的讨论以及设计相应的技术来为特定数据集的有效存储提供解决方案方面，相关的工作并不多。虽然现有的版本控制系统（如 Git、SVN 等）被广泛应用，但其底层实现简单，且在管理大型数据集上存在很大的限制[5]。因此，本文将复现《Principles of Dataset Versioning: Exploring the Recreation/Storage Tradeoff》的工作。我们将在本文中研究分析如何利用数据集版本信息来解决存储量以及恢复成本之间权衡关系。

除了能够处理数据集或是版本数量等规模因素外，还有其他一些值得考虑的因素使得这个问题具有挑战性。首先，不同的应用场景和约束条件导致了均衡存储成本和检索成本这一问题上有许多问题定义，如表 1所示，其中 Δ 和 Φ 分别表示版本间的存储成本和恢复成本的差异。针对存储成本和恢复成本不同的约束，可归纳出 6 个问题， Bhattacherjee et al.[5]为其依次提供了对应的解决方案，我们将在第 3节具体讨论。其次，存储成本和恢复成本在不同环境下有不同的关系。例如，当系统性能的瓶颈在 I/O 或是网络通信上是，存储成本和恢复成本是正比关系；当系统性能的瓶颈在 CPU 计算时，上述关系又未必成立。最后，获取版本间的增量信息影响着最后的权衡结果。理想状况下，获取所有版

本之间的增量信息能构建最佳的解决方案，但这种方式因高昂的时间开销而显得不可行；另一方面，仅依赖版本图的增量信息是不够的，因为过少的信息容易遗漏数据集之间的重要冗余。

表 1: 不同约束条件和目标下的问题定义

	存储成本	恢复成本	无向图 ($\Delta = \Phi$)	有向图 ($\Delta = \Phi$)	有向图 ($\Delta \neq \Phi$)
问题 1	minimize$\{\mathcal{C}\}$	$\forall i, \mathcal{R}_i < \infty$	Minimum Spanning Tree		
问题 2	$\mathcal{C} < \infty$	minimize $\{max\{\mathcal{R}_i \vert 1 \leqslant i \leqslant n\}\}$	Shortest Path Tree		
问题 3	$\mathcal{C} \leqslant \beta$	minimize $\left\{\sum_{i=1}^{n} \mathcal{R}_i\right\}$	NP-hard, LAST Algorithm	NP-hard, LMG Algorithm	
问题 4	$\mathcal{C} \leqslant \beta$	minimize $\{max\{\mathcal{R}_i \vert 1 \leqslant i \leqslant n\}\}$		NP-hard, MP Algorithm	
问题 5	minimize$\{\mathcal{C}\}$	$\sum_{i=1}^{n} \mathcal{R}_i \leqslant \theta$	NP-hard, LAST Algorithm	NP-hard, LMG Algorithm	
问题 6	minimize$\{\mathcal{C}\}$	$max\{\mathcal{R}_i \vert 1 \leqslant i \leqslant n\} \leqslant \theta$		NP-hard, MP Algorithm	

值得注意的是，上述所描述的问题本质上应是“在线”的，因为新的数据集和版本通常是不断被创建并被添加到系统中。在本文中，我们专注于这个问题的静态、离线版本，即数据集数量已经固定的情况下。

总的来说，本工作正式定义和分析了数据集的版本问题，考虑了不同场景下的权衡存储成本和恢复成本的解决思路。本工作所提出的算法可以应用于数据库管理系统，以满足不同用户不同场景下的需求，以应对数据日益增长所导致的数据集管理以及检索方面所存在的问题。

2 相关工作

2.1 源代码版本控制系统

对于版本控制，最早提出并应用的是源代码版本控制系统，如 Git、SVN 等。这些系统在底层存储设计上简单，且被优化用于处理中等大小的源代码文件，其磁盘结构被优化用于处理基于行的差异，因而在处理大文件和大量版本时存在很大的局限性[6]。因此，如 git-annex[7]、git-bigfiles[8]等各种拓展被开发处理，使其能够合理地处理大文件。

2.2 支持版本控制的数据库管理系统

时序数据库中有许多关于管理线性版本链，以及检索特定时间点的版本（快照查询）的相关工作[9-12]。 Buneman et al.[13]提出了一种归档技术，所有版本的数据被合并到一个层次中。然而，这并不是一个成熟的版本控制系统，它代表了一个任意的版本图。而在阵列数据库[14-15]和图数据库[16]的背景下，快照查询也被研究。 Seering et al.[14]提出了一种类似于 MST 的技术，用于在科学数据库的背景下存储任意的版本树。他们还提出了几种启发式方法，用于选择在历史版本的访问频率分布下实现哪些版本。一些数据库支持 “时间查询” 功能（例如，Oracle Flashback，Postgres[17]），但是这些数据库无法构建存在分支的版本图。

2.3 存储文件

许多先前的工作已经研究了存储相关文件集合的总存储成本最小化的问题。这些工作通常不考虑存储成本和恢复成本之间的权衡。 Quinlan et al.[18]提出了一个归档的 “重复数据删除” 存储系统，该系统可以识别跨文件的重复块，只存储一次，以减少存储需求。 Zhu et al.[19]提出了几个关于基本主题的优化方案。 Douglis et al.[20]提出了几种技术来识别可以使用 delta 压缩有效存储的文件对，即使没有关于这两个文件的明确衍生信息；类似的技术可以用来更好地识别矩阵 Δ 和 Φ 的哪些条目在我们的场景中被揭示。 Burns et al.[21]提出了一种使用图论方法对 delta 压缩的文件进行就地重建的技术。

3 本文方法

3.1 本文方法概述

如表 1所呈现，对存储成本和恢复成本有不同的限制，对应着有不同的优化目标。对于问题 1、2，表格中已给出了具体的解决方案。本文主要为问题 3-6 提供对应的解决思路。

由于问题 3-6 的问题均是 NP-Hard 的，因此本文着重于提出高效的启发式算法。具体地，在 3.2中，提出 LMG，即局部移动贪婪算法，专门针对平均恢复成本有约束或者有优化目标的方案：因此，该算法适用于问题 3 和 5；在 3.3中，提出了 MP 算法，即修正的 Prim 算法，专门针对最大恢复成本有约束或有优化目标的方案：因此，该算法适用于问题 4 和 6。

此外，本文将另外介绍两种算法。在 3.2中，提出 LAST 近似算法；在 3.5提出 Git 的改进算法，即 GitH。这两种算法都是从已有的文献中改变而来的，以适应上述问题。上述提及的 4 个算法将在第 5节进行对比测试，并且这些算法都适用于有向版本图和无向版本图，以及对称和非对称的情况。

3.2 Local Move Greedy Algorithm

LMG 算法设计于当用户对版本数据集的平均恢复成本有约束时的情况。本文重点讨论对存储成本优约束的情况（问题 3）；没有这种约束的问题 5 可以通过对前一个问题的反复迭代来解决。

简单地说，该算法从最小生成树（MST）作为 G_S 开始，然后贪婪地增加最短路径树（SPT）中不存在于 MST 的边。该过程如此反复，直至 G_S 达到上限。下面对该算法做出更具体的描述。

首先根据版本图分别构建 MST 和 SPT，其中 MST 用 G_S 表示。令 ξ 为不存在于 G_S 中但存在于 SPT 的边的集合。在每一轮中，算法挑选使 ρ 最大化的边 $e_{uv} \in \xi$，并替换原本指向顶点 v 的边 $e_{u'v}$。挑选完毕后将 e_{uv} 从 ξ 中移除。其中，ρ 的定义如 (1) 所示。值得注意的是，在计算恢复成本减少的总和时需要包含 v 及其所有后代 $w \in G_S$。另一方面，存储成本的增加是指 e_{uv} 的权值减去 $e_{u'v}$ 的权值。只要存储成本总和不超过上限，并且 ξ 的集合不为空，这个过程就会重复执行。该算法的伪代码如 **Algorithm 1**所示。

$$\rho = \frac{\text{reduction in sum of recreation costs}}{\text{increase in storage cost}} \tag{1}$$

3.3 Modified Prim's Algorithm

接下来介绍一种基于 Prim 的最小生成树算法的启发式算法。该算法适用于问题 6，目标是在减少总存储成本的同时，同时控制每个版本的恢复成本都在阈值 θ 以内，问题 4 的解决方案也是类似。

简单地说，该算法是 Prim 算法的一个变种，贪婪地增添存储成本最小的版本和对应的边，最终构成生成树 T。与 Prim 算法不同的是，在 T 构建的过程中，T 中存在的边都有可能在未来的迭代中被删去。在任何时候，该算法都保持一个原则，即 T 中的所有版本的恢复成本都在 θ 之内。下面对该算法的过程做出更具体的描述。

对于每一个版本 V_i，都需要维护 $l(V_i)$ 和 $d(V_i)$ 两个指标，分别表示 V_i 的存储成本和 V_i 的恢复成本。初始化阶段，$l(V_i)$ 是 V_i 实例化的存储成本。X 表示当前生成树 T 中已经存在版本集合，最初 $X = \emptyset$。此外，优先队列 PQ 按照 $l(V_i)$ 的值从小到大进行维护。在每一次迭代中，从 PQ 中获取 $l(V_i)$ 最小的版本 V_i 并添加到生成树 T 中。当 V_i 被添加到 T 中时，更新所有与 V_i 相邻的版本 V_j 的存储成

Algorithm 1: Local Move Greedy Heristic

Input: Minimum Spanning Tree(MST), Shortest Path Tree(SPT), source vertex V_0, space budget W
Output: A tree T with weight $\leqslant W$ rooted at V_0 with minimal sum of access cost

Initialize T as MST.
Let $d(V_i)$ be the distance from V_0 to V_i in T, and $p(V_i)$ denote the parent of V_i in T. Let $W(T)$ denote the storage cost of T.
while $W(T) < W$ **do**
    $(\rho_{max}, e_{SPT}) \leftarrow (0, \emptyset)$.
    for $e_{uv} \in \xi$ **do**
        compute ρ_e.
        if $\rho_e > \rho_{max}$ **then**
            $(\rho_{max}, \bar{e}) \leftarrow (\rho_e, e_{uv})$
    $T \leftarrow T \backslash e_{u'v} \cup e_{uv}$; $\xi \leftarrow \xi \backslash e_{uv}$.
    if $\xi = \emptyset$ **then**
        return T

本和恢复成本。值得注意的是，在 Prim 算法中，我们并不需要考虑已经在 T 中的顶点和相邻节点的关系而在 MP 算法中，任何结点都可能因为新的邻居节点的加入，发现一条更有路径（即当前节点的恢复成本能更进一步减少）。例如，如果边 $\langle V_i, V_j \rangle$ 可以使 V_j 的存储成本降低，而 V_j 的恢复成本不会增加，则可以更新 $p(V_j) = V_i$ 以及 $d(V_j)$、$d(V_j)$ 和 T。即对于当次添加至 T 的结点 V_i 及其邻居 $V_j \notin T$，如果边 $\langle V_i, V_j \rangle$ 能使 V_j 的存储成本降低，并且 V_j 的恢复成本不超过 θ，则算法就更新 $d(V_j)$、$d(V_j)$ 和 $p(V_j)$。该算法的伪代码如 **Algorithm 2**所示。

Algorithm 2: Modified Prim's Algorithm

Input: Graph $G = (V, E)$, threshold θ
Output: Spanning Tree $T = (V_T, E_T)$

Let X be the version set of current spanning tree T; Initially $T = \emptyset, X = \emptyset$.
Let $p(V_i)$ be the parent of V_i; $l(V_i)$ denote the storage cost from $p(V_i)$, $d(V_i)$ denote the recreation cost from root V_0 to version V_i.
Initially $\forall i \neq 0, d(V_0) = l(V_0) = 0, d(V_i) = l(V_i) = \infty$.
Enqueue $\langle V_0, (l(V_0), d(V_0)) \rangle$ into priority queue PQ. (PQ is sorted by $l(v_i)$)
while $PQ \neq \emptyset$ **do**
    $\langle V_i, (l(V_i), d(V_i)) \rangle \leftarrow$top($PQ$), dequeue($PQ$).
    $T = T \cup \langle V_i, p(V_i) \rangle, X = X \cup V_i$.
    for $V_j \in$ (V_i*'s neighbors in* G) **do**
        if $V_j \in X$ **then**
            if $(\Phi_{i,j} + d(V_i)) \leqslant d(V_j)$ *and* $\Delta_{i,j} \leqslant l(V_j)$ **then**
                $T = T - \langle V_j, p(V_j) \rangle$.
                $p(V_j) = V_i$.
                $T = T \cup \langle V_j, p(V_j) \rangle$.
                $d(V_j) \leftarrow \Phi_{i,j} + d(V_i)$.
                $l(V_j) \leftarrow \Delta_{i,j}$.
        else
            if $(\Phi_{i,j} + d(V_i)) \leqslant \theta$ *and* $\Delta_{i,j} \leqslant l(V_j)$ **then**
                $d(V_j) \leftarrow \Phi_{i,j} + d(V_i)$.
                $l(V_j) \leftarrow \Delta_{i,j}; p(V_j) = V_i$
                enqueue $\langle V_j, (l(V_j), d(V_j)) \rangle$ in PQ

3.4 LAST Algorithm

LAST 算法是先前算法[22]的改进，该该算法能在 $\Delta = \Phi$ 以及 Φ 是对称的情况下找到一个存储和恢复成本有着良好平衡的树。

Khuller et al.[22]研究了无向图中平衡最小生成树和最短路径树的问题，在给定参数 α 的情况下，所得到的生成树 T 具有如下特性：（1）对于每个节点 V_i，T 中从 V_0 到 V_i 的路径成本是 G 中从 V_0 到 V_i 最短路径的 α 倍以内；（2）T 的总成本是 G 中最小生成树成本的 $1 + \frac{2}{(\alpha - 1)}$ 倍。具体地，该算法以参数 α 作为输入，从最小生成树开始，对其进行深度优先搜索（DFS）。在遍历过程中访问 V_i 时，如果发现 V_i 的恢复成本超过了 $\alpha\times$ 从 V_0 到 V_i 的最短路径的恢复成本，那么这个当前路径就会被替换成通往该节点的最短路径。该算法的伪代码如 **Algorithm 3**所示。

Algorithm 3: Balance MST and Shortest Path Tree

Input: Graph $G = (V, E)$, MST, SP
Output: Spanning Tree $T = (V_T, E_T)$

```
Initialize T as MST. Let d(V_i) be the distance from V_0 to V_i in T and p(V_i) be the parent of V_i in T.
while DFS traversal on MST do
    (V_i, V_j) ← the edge currently in traversal.
    if d(V_j) > d(V_i) + e_{i,j} then
        d(V_j) ← (d(V_i) + e_{i,j}).
        p(V_j) ← V_i.
    if d(V_j) > α * SP(V_0, V_j) then
        add shortest path (V_0, V_j) into T.
        d(V_j) ← SP(V_0, V_j).
        p(V_j) ← V_0.
```

3.5 Git Heuristic

GitH 是对 Git 目前使用的启发式算法的改编。该算法适用两个参数：w（窗口大小）和 d（最大深度）。下面对该算法的过程进行描述。

该算法按照版本大小非递增顺序考虑这些版本。在这个顺序下，第一个版本被存储为版本图的根，深度为 0（即当前版本实例化）。在存储版本的过程中，维护一个最多包含 w 个版本的滑动窗口。在第一个版本之后的每一个版本 V_i，让 V_l 表示当前窗口中的一个版本。通过计算 $\Delta'_{l,i} = \Delta_{l,i}/(d - d_l)$（其中 d_l 是 V_l 的深度），找到是这个值最低的版本 V_j，并让它作为 V_i 的父节点（只要 $d_j < d$）。接着，V_i 的深度被设置为 $d_j + 1$。处理完毕之后对滑动窗口进行修改，将 V_l 移到窗口的末端（使其停留时间更长），V_j 被添加到窗口中，而窗口开头的版本会被删除。

4 复现细节

4.1 与已有开源代码对比

复现过程中没有参考任何相关源代码，对[5]中的四个算法（LMG，MP，LAST，GitH）进行了复现。

4.2 实验数据集获取

本次复现所使用的数据集来自美国国家冰雪数据中心（NSIDC[23]），它是美国国家航空航天局 (NASA) 所建立的对地观测系统的一部分。我们从 NSIDC 中获取 3 个数据集，分别是 ILATM2[24]、

SPL2SMP[25]、NISE[26]。

除了获取数据集本身外，我们还需要获取版本间的 Δ 和 Φ 等数据。我们是用 Xdelta3[27]——一个用于 delta 压缩的开源工具，来计算和记录版本间的 Δ 以及 Φ 值。此外， Bhattacherjee et al.[5]所设计的启示类算法是为信息受限的情况下所服务的（因为实际应用中获取所有版本对的 Δ 和 Φ 非常耗时的）。因此，我们以 10 个版本为一组，仅获取 10 跳距离内的所有版本对的相关数据（增量文件大小，获取 Δ 的时间等），并且组间使用一对版本进行连接，从而构建出一个信息受限的版本图。值得注意的是，该版本图并没有反映版本间的衍生关系，而仅仅只是为了获取版本间的信息而服务的。

4.3 创新点

本次复现除了模拟原文的相关实验外，我还将讨论信息数目对版本数据存储的影响。如 4.2所述，我们只是获取了部分版本对之间的信息。这不由地联想，若我们获取的信息数目更多，那么会对最终的布局造成怎样的影响（存储成本、恢复成本等）。因而在 5.2.3，我们将通过实验的方式呈现这一影响，该额外实验也能为后续这方面的工作提供思路与启示。

5 实验结果分析

本部分对实验所得结果进行分析，详细对实验内容进行说明，实验结果进行描述并分析。

5.1 数据集分析

由 4.2所述获取数据集，这里各个数据集的信息，如 2所示。其中，MCA 和 SPT 分别表示最小生成树以及最短路径树。

表 2: 数据集信息

Datasets	ILATM2	SPL2SMP	NISE
Number of Versions	1017	1047	1035
Number of Deltas	9233	9506	9393
Average Version Size(KB)	83.7129	1389.09	2137
Average Delta Size(KB)	29.0932	960.316	93.7784
MCA-Storage Cost(KB)	29413	1005369	97381
MCA-Sum Recreation Cost(KB)	6665986	147951497	234517203
MCA-Max Recreation Cost(KB)	13207	287063	436190
SPT-Storage Cost(KB)	81778	1453505	2211795
SPT-Sum Recreation Cost(KB)	84141	1454372	2211795
SPT-Max Recreation Cost(KB)	87	1777	2137

由表可知，ILATM2、SPL2SMP、NISE 三个数据集，版本数目相当，且单个版本的平均存储成本依次增加（NISE 数据集的单个版本的平均出成本是 ILATM2 的近 3 倍）。但是，通过平均增量大小与单个版本的平均存储成本可以获知，SPL2SMP 数据集的增量压缩最差，ILATM2 其次，而单版本平均存储成本最大的 NISE 数据集却有着最高的增量压缩。同样地，三个数据集的 MCA 存储成本也不满足单个版本的平均存储成本变化规律：SPL2SMP 的 MCA 有着最大的存储成本，远大于 ILATM2 和 NISE 的 MCA 的存储成本。而 MCA 方案的总恢复成本以及单版本的最大恢复成本方面，ILATM2 的数值最低，NISE 的数值最高。最后来看 SPT，可以看到三个数据集的 SPT 存储成本与总恢复成本相

近，这是因为 SPT 方案下会导致绝大多数的版本实例化，而对于实例化版本而言，由于不计算增量，因而此时系统唯一出现的瓶颈是 IO 传输，即恢复成本与存储成本成正比。因而实例化版本下，其恢复成本等于其存储成本，从而 SPT 的存储成本与总恢复成本相近。此外，由于 SPT 是以恢复成本为指标来构建的，所以其总恢复成本和单版本的最大恢复成本都远小于 MCA，与之相对的，其存储成本远高于 MCA 的存储成本。

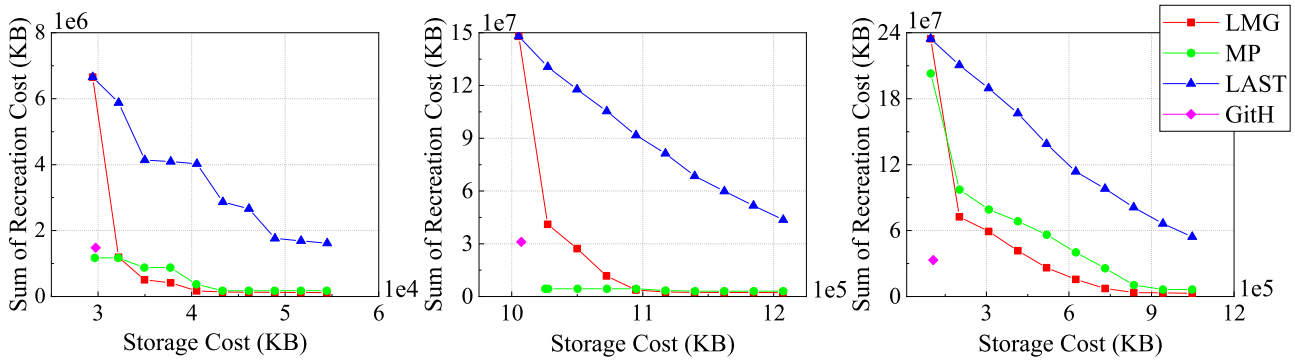
5.2 实验结果

接下来将呈现 4 个算法在 3 个数据集上的实验结果。其中，在对 MP 算法进行测试时，其最大恢复成本设置为 LMG 方案的最大恢复成本结果。这么做一方面验证 MP 能限制最大恢复成本的同时，也能比较两个方案在平均恢复成本上的差异。此外，我们用四种不同的窗口大小（500、250、125、100）和固定的深度 50 来运行测试 GitH 算法。最后，对于每一个数据集，所有版本实例化的存储成本为 S_{max}，MCA 的存储成本为 S_{min}，实验所设置的存储成本上限 $S_{constraint}$ 会从初始值 S_{min} 逐渐增大至 $(S_{min}+S_{max})/2$。

后续内容安排如下：在 5.2.1和 5.2.2，我们将分别呈现在有向图和无向图的情况下，各个方案的实验结果；在 5.2.3，我们将以 LMG 算法为例，分析信息数目的差异对最终版本布局的影响；在 5.2.4，我们将分析 LMG 算法的运行时间。

5.2.1 有向图

我们首先在有向图数据集上，对四个算法进行测试，绘制出了设置不同存储成本上限的情况下，不同算法的总恢复成本（图 1）和最大恢复成本（图 2）的变化。

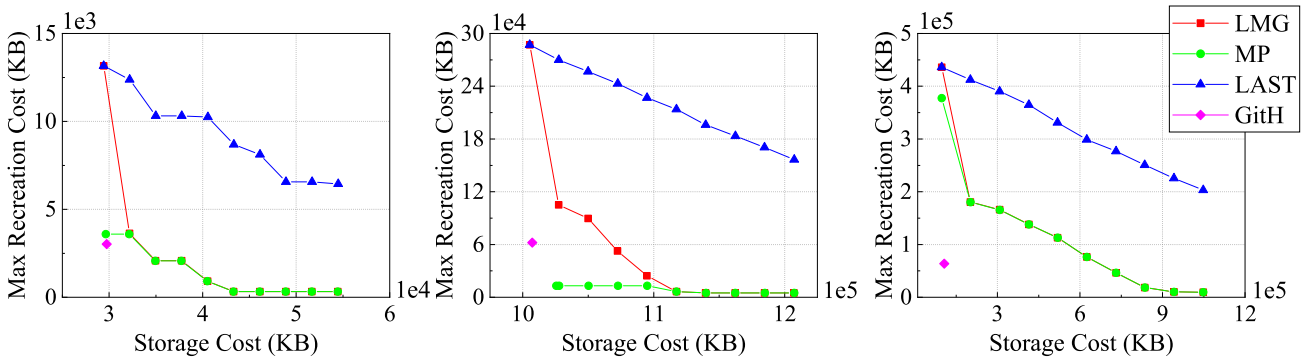


(a) ILATM2 (b) SPL2SMP (c) NISE

图 1: 有向图数据集实验结果：比较存储成本和总恢复成本的变化

由图 1，我们能观察到：当 $S_{constraint}$ 从 S_{min} 适当增加后，其总恢复成本能显著下降，这个特征在 LMG 和 LAST 体现上尤为明显（对于 LMG 方案，当 $S_{constraint}$ 在三个数据集上分别增加 9%、22%、100% 时，其恢复成本分别下降了 73%、63%、59%）。此外，也能观察到 LMG 算法的总恢复成本下降速度比 LAST 算法更快，这是因为同样是将 SPT 融入至 MCA 中，LMG 算法会优先考虑极大减少总恢复成本的版本，而 LAST 方案仅是通过 DFS 遍历的方式考查各个版本。即便如此，LMG 和 LAST 算法的初始总恢复成本远高于 MP。虽然他们均能达到存储成本的理论最小值，但此时只是单一地追求最低存储成本而完全忽略了恢复成本，从而导致了恢复成本过高。而反观 MP 算法，虽然该算法不能达到存储成本的最小值，但是其总恢复成本的初始值低显著低于 LMG 和 LAST。但随着 $S_{constraint}$

逐渐增大，其总恢复成本的下降速度也不如 LMG 算法快，因此在存储成本上限较大的情况下，LMG 算法的恢复成本低于 MP 算法。最后来看 GitH，该算法表现出了较低的总恢复成本，但是并不跟随 $S_{constraint}$ 的变化而变化。该算法是一个贪婪的启发式算法，它决策版本存储方式的核心是深度，而没有像其他方案那样同时兼顾了存储成本以及恢复成本。在之后的实验结果里，我们会看到 GitH 并不是总是拥有低恢复成本的结果，这种贪婪的策略会导致 GitH 的性能表现极不稳定。



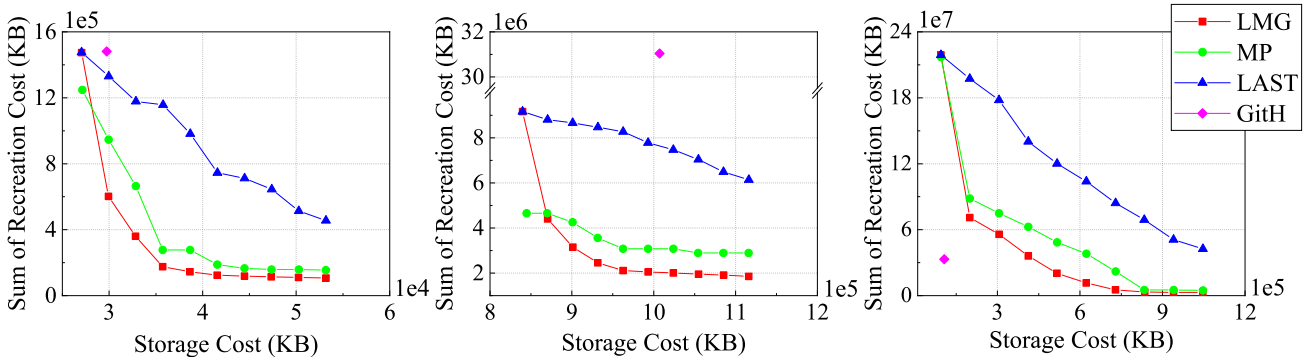
(a) ILATM2 (b) SPL2SMP (c) NISE

图 2: 有向图数据集实验结果：比较存储成本和最大恢复成本的变化

接下来来看最大恢复成本的结果图，如图 2所示。可以看到，与总恢复成本有着相似的趋势：LMG 和 LAST 在存储成本达到理论最小值时，其最大恢复成本显著高于 MP，但随着 $S_{constraint}$ 上升，最大恢复成本逐渐下降，并且 LMG 下降的更为显著。由于我们在对 MP 的最大存储成本的限制上是与 LMG 的结果一致，因此随着 $S_{constraint}$ 上升，二者的曲线合并为一条曲线。而 GitH 依然保持着较低的恢复成本。

5.2.2 无向图

接着，我们来看四个算法在无向图数据集上的测试效果，图 3和图 4分别呈现了算法的总恢复成本以及最大恢复成本。

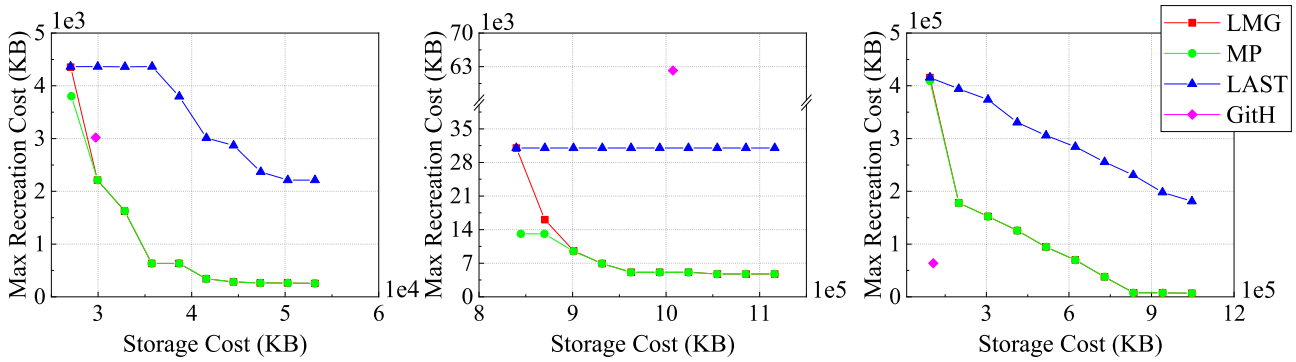


(a) ILATM2 (b) SPL2SMP (c) NISE

图 3: 无向图数据集实验结果：比较存储成本和总恢复成本的变化

由图 3，我们再次观察到与图 1相同的趋势：当 $S_{constraint}$ 逐渐增大时，LMG、MP 和 LAST 算法的总恢复成本逐渐下降，并且 LMG 的下降速度尤为显著；虽然 MP 算法的总恢复成本的初始值低于 LMG 算法，但是随着 $S_{constraint}$ 逐渐增大，LMG 的总恢复成本比 MP 算法更低。而 GitH 算法并没有展现出与图 1一样的情况，它在 3 个数据集上各有不同的表现：在 ILATM2 和 SPL2SMP 上，其恢复

成本超过其他算法（在 SPL2SMP 上表现得尤为糟糕）；仅在 NISE 上表现出极好的恢复成本和存储成本。由此不难看出，GitH 这种简单的设计方案在实际应用中并不稳定。



(a) ILATM2 (b) SPL2SMP (c) NISE

图 4: 无向图数据集实验结果：比较存储成本和最大恢复成本的变化

接着来看最大恢复成本的结果图，如图 4所示。可以看到，与图 2相比，LAST 算法存在的问题被更进一步放大。在 SPLSMP 中，无论 $S_{constraint}$ 如何变化，其最大恢复成本都不变。这是因为 LAST 通过 DFS 遍历来修改版本的存储方式，以此来改善恢复成本。但这也就存在着恢复成本最大的版本迟迟没有得到改善的可能，所以导致了图中 LAST 方案的最大恢复成本没有下降的情况。此外，LMG、MP 依旧保持着与 2一样的趋势，而 GitH 也如图 3一样，其性能表现极不稳定。

5.2.3 信息数目的影响

在这个部分，我们将比较信息数目对版本数据存储的影响。如 4.2所述，我们仅仅只是获取了部分版本对之间的信息。接下来，我们将以 LMG 算法为例，测试在获取所有版本对之间的信息的情况下，版本数据集的存储情况。具体地，我们将对比存储成本，总恢复成本、最大恢复成本以及增量数据获取时间的变化，结果如下所示。

数据集	ILATM2	SPL2SMP	NISE
部分信息	29413	1005369	97381
完整信息	29044 ↓	1004812 ↓	93709 ↓

表 3: 存储成本变化 (KB)

数据集	ILATM2	SPL2SMP	NISE
部分信息	6655471	147951497	234517203
完整信息	210646 ↓	10918194 ↓	45934303 ↓

表 4: 总恢复成本变化 (KB)

数据集	ILATM2	SPL2SMP	NISE
部分信息	13158	287063	436190
完整信息	511 ↓	17056 ↓	77839 ↓

表 5: 最大恢复成本变化 (KB)

数据集	ILATM2	SPL2SMP	NISE
部分信息	8	50	16
完整信息	870 ↑	5709 ↑	1817 ↑

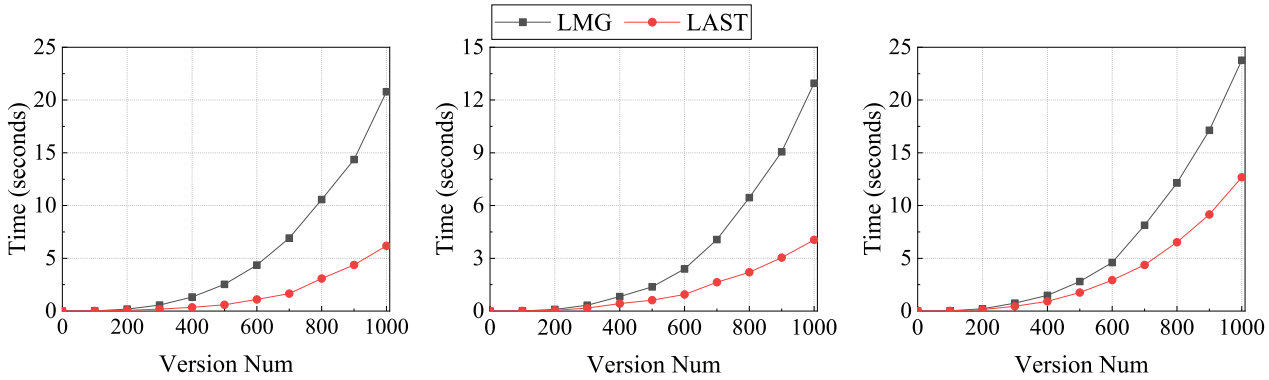
表 6: 增量数据获取时间变化 (分)

由上表所呈现，获取完整信息后会使存储成本、总恢复成本以及最大恢复成本下降。具体地，在三个数据集上，存储成本分别下降了 1%、1%、4%；总恢复成本分别下降了 97%、93%、80%；最大恢复成本下降了 96%、94%、82%。虽然在获取完整信息后，存储成本没有明显变化，但是恢复成本却有显著的下降。这说明了随着版本对的信息量上升，算法能对这些版本数据集有着更优的存储，使得存储成本以及恢复成本都有改善。然而，获取完整信息必然导致获取信息的时间上升，可以看到在三

个数据集上，增量数据的获取时间均增加了 ×100 倍。该结果也能推断，随着版本数目的上升，其倍数会更进一步扩大。因而在后续工作中，我们可以考虑时间利用方面的问题，如何充分利用时间，尽可能获取更多的增量数据。

5.2.4 运行时间

在这个部分，我们将评估各个算法的运行时间，如 5.2.4所示。



(a) ILATM2 (b) SPL2SMP (c) NISE

图 5: LMG 和 LAST 在三个数据集上的运行时间

首先，我们评估 LMG 和 LAST 的运行时间，二者均将 MCA 和 SPT 作为输入。可以看到，在三个数据集上，LMG 的运行时间都高过 LAST，这是因为 LMG 用 SPT 的变替换 MCA 的过程中需要计算整个存储树的恢复成本减少量，从中挑选出“性价比”最高的边并进行替换；而 LAST 仅仅只是依赖于 DFS 遍历，遍历的过程中进行替换。因而在将 SPT 融入 MCA 的阶段，LMG 的时间复杂度比 LAST 更高，耗时更长。另一方面，可以看到三个数据集中，NISE 的耗时最多，ILATM2 其次，SPL2SMP 耗时最少。这是因为三者的压缩率依次下降，需要从 SPT 中获取更多的边来满足存储上限。

对于 MP 算法，当最大恢复成本设置与 LMG 相同时，MP 在这些数据集上最多仅需要 0.02 秒；而 GitH 在这些数据集上的运行时间最多也仅是 0.03 秒。

总之，虽然 LMG 本质上是一个比 MP 和 LAST 更为耗时的算法，但是它在规模数据中的运行时间是合理的（由 5.2.3可知，获取增量数据的时间远大于算法本身运行的时间）。

6 总结与展望

对大数据集的协同工作以及迭代分析正在成为许多应用领域的常态。考虑到数据集之间存在着重叠，考虑使用 delta 压缩以用更为紧凑的方式来存储数据集是可行的。然而，单单追求极低的存储成本往往会导致检索特定的数据集或是记录时存在较高的延迟。本文中对这类问题进行定义，并提出了几种有效的算法，使得存储和恢复成本之间达到动态平衡。

在论文复现中，我并没有获取到原文献所使用的人工合成的数据集，因而转而使用别的数据集进行测试。在测试过程中，GitH 算法的结果与原文有出入。出现这样的现象主要是由于数据集的选取以及数据规模，因而以后的复现中，我会尽力获取并使用原文的数据集以及对应的规模。

最后，本文的主要工作围绕的是“静态”数据的情况下，该存储数据集，使得存储和恢复成本达到动态平衡。但事实上，实际应用中数据集是动态添加的，因而下一步我们可以思考当面临“动态添加”的数据时，上述算法能否使用，存在怎样的问题。此外，我们在 5.2.3中提及了信息数量对于最后

存储结果的影响，我们同样也能考虑，在什么情况下可以获取完整的数据，又或是在怎样情况下，我们能获取更多的数据，以更进一步改善恢复成本。

参考文献

[1] Large Synoptic Survey Telescope[Z]. https://www.lsst.org/.

[2] Earth microbiome project[Z]. https://earthmicrobiome.org/.

[3] HUANG S, XU L, LIU J, et al. Orpheusdb: Bolt-on versioning for relational databases[J]. arXiv preprint arXiv:1703.02475, 2017.

[4] BHARDWAJ A, BHATTACHERJEE S, CHAVAN A, et al. Datahub: Collaborative data science & dataset version management at scale[J]. arXiv preprint arXiv:1409.0798, 2014.

[5] BHATTACHERJEE S, CHAVAN A, HUANG S, et al. Principles of dataset versioning: Exploring the recreation/storage tradeoff[C]//Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases: vol. 8: 12. 2015: 1346.

[6] DESHPANDE A. Why Git and SVN Fail at Managing Dataset Versions[Z]. http://www.cs.umd.edu/~amol/DBGroup/2015/06/26/datahub.html.

[7] git-annex[Z]. https://git-annex.branchable.com/.

[8] git-bigfiles[Z]. http://caca.zoy.org/wiki/git-bigfiles.

[9] BOLOUR A, ANDERSON T L, DEKEYSER L J, et al. The role of time in information processing: a survey[J]. ACM SIGART Bulletin, 1982(80): 28-46.

[10] SNODGRASS R, AHN I. A taxonomy of time databases[J]. ACM Sigmod Record, 1985, 14(4): 236-246.

[11] OZSOYOGLU G, SNODGRASS R T. Temporal and real-time databases: A survey[J]. IEEE Transactions on Knowledge and Data Engineering, 1995, 7(4): 513-532.

[12] TANSEL A U, CLIFFORD J, GADIA S, et al. Temporal databases: theory, design, and implementation [M]. Benjamin-Cummings Publishing Co., Inc., 1993.

[13] BUNEMAN P, KHANNA S, TAJIMA K, et al. Archiving scientific data[J]. ACM Transactions on Database Systems (TODS), 2004, 29(1): 2-42.

[14] SEERING A, CUDRE-MAUROUX P, MADDEN S, et al. Efficient versioning for scientific array databases[C]//2012 IEEE 28th International Conference on Data Engineering. 2012: 1013-1024.

[15] SOROUSH E, BALAZINSKA M. Time travel in a scientific array database[C]//2013 IEEE 29th International Conference on Data Engineering (ICDE). 2013: 98-109.

[16] KHURANA U, DESHPANDE A. Efficient snapshot retrieval over historical graph data[C]//2013 IEEE 29th International Conference on Data Engineering (ICDE). 2013: 997-1008.

[17] STONEBRAKER M, KEMNITZ G. The POSTGRES next generation database management system[J]. Communications of the ACM, 1991, 34(10): 78-92.

[18] QUINLAN S, DORWARD S. Venti: A new approach to archival data storage[C]//Conference on File and Storage Technologies (FAST 02). 2002.

[19] ZHU B, LI K, PATTERSON R H. Avoiding the disk bottleneck in the data domain deduplication file system.[C]//Fast: vol. 8. 2008: 269-282.

[20] DOUGLIS F, IYENGAR A. Application-specific Delta-encoding via Resemblance Detection.[C]// USENIX annual technical conference, general track. 2003: 113-126.

[21] BURNS R C, LONG D D. In-place reconstruction of delta compressed files[C]//Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing. 1998: 267-275.

[22] KHULLER S, RAGHAVACHARI B, YOUNG N. Balancing minimum spanning trees and shortest-path trees[J]. Algorithmica, 1995, 14(4): 305-321.

[23] NASA. National Snow and Ice Data Center[Z]. https://nsidc.org/. 2022.

[24] STUDINGER M. IceBridge ATM L2 Icessn Elevation, Slope, and Roughness, Version 2[C]//. 2020.

[25] CHAN S, BINDLISH R, O'NEILL P E, et al. Assessment of the SMAP passive soil moisture product [C]//IEEE Transactions on Geoscience and Remote Sensing. 2016: 6046-6048.

[26] BRODZIK M J, STEWART. J S. Near-Real-Time SSM/I-SSMIS EASE-Grid Daily Global Ice Concentration and Snow Extent, Version 5[EB/OL]. NASA National Snow. 2016. https://nsidc.org/data/NISE/versions/5.

[27] MACDONALD J. Xdelta3[Z]. https://github.com/jmacd/xdelta. 2017.