

学习型二级索引 HERMIT 的复现

刘欢

摘要

数据库管理员在数据表上构造辅助索引，以加速关系数据库管理系统（RDBMS）中的查询处理。根据数据统计，这些索引构建在最常查询的列的顶部。不幸的是，在同一数据库中维护多个辅助索引可能会非常消耗空间，由于内存空间的潜在耗尽，导致性能显著下降。在此课程报告中，我复现了 HERMIT，一种用于现代 RDBMS 的简洁的二级索引机制。HERMIT 明智地利用隐藏在列之间的软函数依赖关系，以删除索引键访问的冗余结构。与其构建一个完整的索引来存储键列中的每一个条目，HERMIT 将所有传入的键访问查询导航到建立在相关列上的现有索引。这是通过分层回归搜索树（TRS-TREE）实现的，这是一种简洁、ML 增强的数据结构，它执行快速曲线拟合，以自适应和动态地捕获列相关性和异常值。我们的实验证实，HERMIT 可以在有限的性能开销下显著减少空间消耗，尤其是在支持复杂范围查询时。

关键词：关系数据库管理系统；列相关性；二级索引；分层回归搜索树；HERMIT

1 引言

现代关系数据库管理系统（RDBMS）支持快速的辅助索引，有助于在事务和分析工作负载中加速查询处理。这些索引由数据库管理员或查询优化器自动创建，建立在最常查询的列之上，因此提供了通过这些列检索数据元组的有效方法。然而，管理数据库中的多个辅助索引会占用大量存储空间，这可能会由于内存空间的耗尽而导致性能严重下降。这个问题在现代主存 RDBMS 中并不罕见，因为内存空间是一种稀缺资源。

面对这个问题，数据库社区的研究人员提出了各种实用的解决方案来限制索引维护的空间使用。从数据库调整的角度来看，一些研究工作引入了智能性能调整顾问，该顾问可以在固定空间预算的情况下自动选择最有益的二级索引。在满足空间限制的同时，这些技术实质上限制了在表上构建的二级索引的数量，从而导致查找未索引列的查询性能不佳。从结构设计的角度来看，一组研究人员开发了节省空间的索引结构，与传统索引相比，这种索引结构消耗更少的存储空间。这些工作要么只存储列条目的子集，要么使用压缩技术来减少空间消耗。然而，这样的解决方案节省了有限的空间^[1]，并且可能导致查找操作的高开销。

2 Motivation

数据表中的许多列显示出相关关系或软函数依赖关系，其中一列的值可以用另一列的近似值来估计。部署这样的关系可以大大减少二级索引维护导致的内存消耗。具体来说，如果我们想在一个与另一个已建立索引的列 N 高度相关的列 M 上创建索引，我们可以简单地构造一个简洁的 ML 增强数据结构，以捕获 M 和 N 之间的相关性。使用多个简单的统计回归过程来拟合隐藏相关函数的曲线。为了对 M 执行查找查询，RDBMS 从新构造的数据结构中检索 N 的查找范围，并使用 N 的索引获取目标元组。这种机制就是 HERMIT。

HERMIT 在支持范围查询时具有竞争性的性能，而范围查询通常用于辅助键列访问。它还提出了计算和空间消耗之间的权衡。虽然避免构建完整的索引结构显著减少了空间消耗，但 HERMIT 要求任何传入的查询在检索目标元组之前都要经过额外的跳跃。然而，正如实验所表明的那样，这种开销实际上不会对性能产生实质性影响，而且当存储空间非常宝贵且稀缺时，例如在主存 RDBMS 中，它也会带来巨大的好处。

3 相关工作

3.1 树索引结构

B+-Tree^[2]是面向磁盘的 DBMS 的教科书索引，其结构经过精心设计以减少随机磁盘访问。随着主内存价格的降低，研究人员和从业者开发了内存友好的索引，可以有效利用更大的主内存和快速的随机访问速度。一些开创性的工作包括 T-TREE^[3]和缓存意识索引^[4]。所有这些索引都使用层次树结构来及时返回准确的查询结果。然而，这些解决方案会导致高内存消耗，从而对主内存 RDBMS 造成高压力。

3.2 简洁的索引结构

稀疏索引只存储指向父表中磁盘页（或列块）的指针，以及每个页（或列块）中的值范围，以减少空间开销。Column Sketch^[5]逐值索引表，但将值压缩为有损映射。权衡的结果是，这些结构会在查询时间中引入假阳性。BF-TREE^[6]是为有序或分区数据设计的近似索引。在生成未量化结果的同时，它可以通过只记录近似元组位置来大大减少空间消耗。学习索引^[7]通过使用机器学习技术利用数据分布提高了空间效率。它产生了良好的性能，但需要较长的训练阶段来生成数据结构。Zhang 等人^[8]为日志结构合并树提出了一种新的范围查询过滤机制。

Stonebraker^[9]引入了部分索引，该索引仅存储索引列中的条目子集，以减少叶节点的数量。Idreos 等人开发了一系列称为数据库破解的技术，以基于查询工作负载自适应地生成索引。具体地说，部分侧写^[10]引入了一个称为部分映射的索引，该索引由几个自组织块组成。可以根据剩余的存储空间自动删除或重新创建这些块，从而最大限度地利用可用空间。Athanasios 等人^[11]后来提出了 RUM 猜想，以捕捉读取、更新和内存开销之间的关系。

压缩技术丢弃冗余数据以节省存储空间。然而，这些技术需要额外的时间来提前压缩数据，并在查询时对数据进行解压缩。这会降低查询性能和索引维护速度。此外，他们仍然存储元组的指针，这样保存的内存量是有限的。

3.3 二级索引选择

一些工作还讨论了如何在固定的空间预算量下选择二级索引。微软的一组研究人员提出了一种分析 SQL 查询工作负载的机制，并提出了合适的索引^[12]。他们进一步提出了一种端到端的解决方案，以解决选择物质化视图和索引的问题。IBM 的研究人员将索引选择问题建模为背包问题的变体，并将索引推荐机制引入 DB2。最近，Pavlo 等人^[13]使用基于机器学习的方法研究了这个问题。

3.4 列相关性

BHUNT^[14]自动显示列对之间的代数约束。通过重新放松相关性，CORDS 使用采样来消除列之间的相关性和软函数相关性。此外，CORDS^[15]还推荐了保持某些简单联合统计数据的专栏组。研究人

员数据清理还致力于检测功能依赖，包括软依赖和近似依赖。CORADD^[16]提出了一种相关感知数据库设计器，为给定的数据库大小约束推荐最佳的材料化视图和索引集。Correlation Maps (CM)^[17]是一种数据结构，它表示加速未索引列访问的相关属性之间的映射。虽然共享了利用列相关性来节省空间的类似方法，但 HERMIT 不需要使用集群列；更重要的是，它的 ML 增强型 TRS-TREE 结构可以自适应地动态地对复杂相关性和异常值进行建模，因此在许多情况下会产生更好的性能。

3.5 基数估计

基数估计在 RDBMS 查询优化器中起着至关重要的作用。列相关性是阻碍估计的最常见原因。示例视图^[18]和 PSALM^[19]使用采样方法来检测列相关性。最近的项目开始将列语义视为黑盒，并使用机器学习模型从查询反馈中学习基数。Kipf 等人^[20]选择使用深度学习技术来学习联接查询的基数。

4 本文方法

4.1 本文方法概述

RDBMS 中不同列之间隐藏的相关性表明它们对应索引结构的高度相似性。观察到这一点，出现了一种简洁但快速的二级索引机制，名为 HERMIT^[21]，它利用列相关性来回答查询。

为了索引指定的列 M，HERMIT 需要两个组件：一个建立在目标列 M 上的分层回归搜索树的简洁数据结构 (TRS-TREE)，以及主机列 N 上预先存在的称为主机索引的完整索引。TRS-TREE 模拟了 M 和 N 之间的相关性：它利用逐步回归方法对 M 到 N 中的相关函数 $N = F_n(M)$ 进行分层曲线拟合，并使用树结构对一组回归函数进行索引，每个回归函数表示从 M 值范围到 N 值范围的近似线性映射。

为了处理查询，HERMIT 采用了三阶段搜索算法：(1) TRS-TREE 搜索；(2) 主机索引搜索；(3) 基表验证。具体来说，HERMIT 使用查询谓词来搜索 TRS-TREE，以便从 M 到 N 检索范围映射。然后，它利用主机索引来查找一组候选元组标识符。我们注意到这个候选集合是近似的，并且它包含不能满足原始谓词的假阳性。HERMIT 通过直接验证基表上的相应值来检测假阳性。

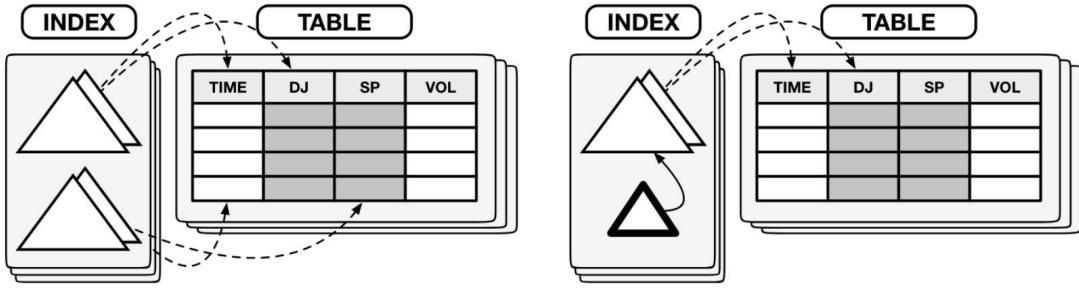
HERMIT 也适用于多列二级索引。假设表中的两个列 A 和 M 经常被查询在一起，所以 (A, M) 上的索引是可取的。她可以利用 (A, N) 上的主机索引以及 M 和 N 之间的相关性来回答关于 A 和 M 的查询。

在存在多列相关性的情况下（例如，(W, X) \rightarrow (Y, Z)，尽管 RDBMS 很少检测到），HERMIT 连接多个键并构建 TRS-TREE。

我们现在使用一个运行示例来演示 HERMIT 是如何工作的。让我们考虑一个数据表 STOCK-HISTORY 记录美国股市交易历史，有四个不同的列：TIME（即交易日期）、DJ（即道琼斯）、SP（即标准普尔 500）和 VOL（即总交易量）。数据库管理员已经在 (TIME, DJ) 上创建了索引。现在，它决定在 (TIME, SP) 上创建另一个索引，因为经常会出现这样的查询：

```
SELECT * FROM STOCK-HISTORY  
WHERE (TIME BETWEEN ? AND ?) AND (SP BETWEEN ? AND ?)
```

在收到索引创建语句时，RDBMS 采用 HERMIT 首先检查是否通过任何相关发现算法检测到包含 TIME 或 SP 的任何列相关。如果观察到 SP 中的值与 DJ 中的值高度相关，并且 (TIME, DJ) 上存在索引，则 RDBMS 将构建一个 TRS-TREE，以对从 SP 到 DJ 的相关映射进行建模。给定列 TIME 上的



(a) Conventional index

(b) HERMIT

图 1: 通过传统二级索引和 HERMIT。(双三角形表示传统的二级索引; 小的单三角形表示提议的 TRS-TREE 状结构)

查询范围 (T_{min}, T_{max}) 和列 SP 上的 (S_{min}, S_{max})，HERMIT 首先将 SP 范围 (S_{min}, S_{max}) 输入到构造的 TRS-TREE 中，以获取 DJ 上的相应范围 (D_{min}, D_{max})。然后，它使用 TIME 范围 (T_{min}, T_{max}) 和 DJ 范围 (D_{min}, D_{max}) 搜索 (TIME, DJ) 上的主索引，以找到所有令人满意的元组。为了过滤假阳性，RDBMS 从基表中读取 SP 的值并验证结果的正确性。

图 1 显示了在运行示例中检索元组时 HERMIT 与传统的二次索引机制的区别。与提供对元组标识符的直接访问的现有索引技术不同，HERMIT 需要两跳访问。虽然可能会导致点查询的更高访问开销，但 HERMIT 可以在二级索引中常见的范围查询中获得非常有竞争力的性能（正如我们将在实验中演示的那样）。当然，HERMIT 可以显著减少索引维护的空间消耗。HERMIT 可以支持正确性保证的插入、删除和更新操作。由于其近似特性，HERMIT 最适合于范围查询，这对于辅助关键字列非常常见，尤其是在数据分析中。此外，HERMIT 对内存 RDBMS 非常有益，因为内存空间不足

4.2 TRS-TREE

TRS-TREE 是一种简洁的树结构，它在数据库的同一数据表中对目标列和主机列之间的数据关系进行建模。它利用分层回归方法来适应和动态类型，以在相关函数 $N=F_n(M)$ 上进行曲线拟合。为了更精确，TRS-TREE 将复杂的曲线拟合问题分解为多个更简单的子问题，并使用线性回归方法精确地解决这些子问题。TRS-TREE 是自适应的，在这个意义上，它基于相关性复杂性来构建其内部结构；它也是动态的，这意味着它在运行时重组内部以确保最佳效率。在下一节中，我们介绍 TRS-TREE 的内部结构，在复现细节章节演示了它的构造、查找、维护的具体实现。

4.2.1 内部结构

TRS-TREE 是一种将目标列 M 中的值映射到主机列 N 中的值的树状结构。它的构造算法递归地将 M 的值范围划分为多个子范围，直到 M 到 N 中的每个条目对 (m, n) 可以通过基于简单线性回归的数据映射来很好地估计相应子范围所覆盖。作为一种基于树的数据结构，TRS-TREE 使用叶节点来维护详细的数据映射，其内部节点为这些叶节点提供快速导航。图 2 显示了在评估范围为 0 到 1024 的目标列上构建的 TRS-TREE 结构。

叶节点。TRS-TREE 中的叶节点与目标列 M 的一个分区关联。我们定义一个范围 r 有两个元素：下边界 lb 和上边界 ub 。给定一组被范围 r 覆盖的列 M^r ，叶节点试图提供从 M^r 到其在主机列 N 中的相应列项集合 N^r 的近似线性映射。这种映射使用线性函数 $n=\beta m+\alpha \pm \epsilon$ 来表示，其中 m 和 n 表示 M^r 和 N^r 列值， β 和 α 分别表示函数的斜率和截距， ϵ 表示置信区间。TRS-TREE 使用标准线性回归公式

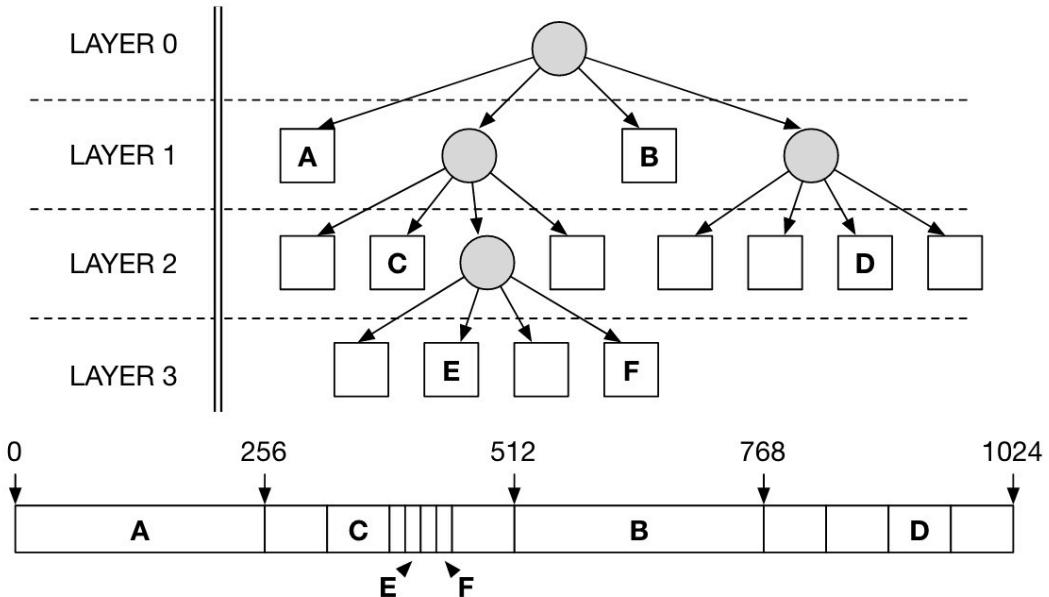


图 2: 目标列上的 TRS-TREE 数据结构, 值范围为 0 到 1024。节点扇出设置为 4。框表示叶节点, 圆圈表示内部节点。标尺条显示 TRS-TREE 如何划分目标列的范围。

计算 β 和 α :

与斜率和截距不同, 间隔 ϵ 的置信度可以根据用户定义的参数 (称为误差边界) 来计算。

函数 $n = \beta m + \alpha \pm \epsilon$ 捕获了 M 中子范围 r 下的列 M 和 N 之间的近似线性相关性。给定一个 M^r 中的 m , 它给出相应的 n 的边界在 $(\beta m + \alpha - \epsilon, \beta m + \alpha + \epsilon)$ 范围内。然而, 并非 M^r 和 N^r 的所有入口对 (m, n) 都必须被计算的线性函数覆盖。我们将这些条目称为异常值。叶节点将所有这些异常值保存在另一个外部缓冲区中, 该缓冲区被实现为从 m 映射到相应元组标识符的哈希表, 该标识符可以是主键或元组位置。

内部节点。TRS-TREE 中的内部节点用作导航器, 将查询路由到其目标叶节点。与叶节点类似, 每个内部节点都与目标列 M 中的某个范围相关联。然而, 内部节点不维护到主机列的任何映射, 而是只维护指向其子节点的固定数量的指针, 每个子节点可以是叶节点或其他内部节点。要执行查找, 内部节点可以轻松地将查询导航到其范围覆盖输入值的相应子节点。

4.3 HERMIT

TRS-TREE 查询只返回近似结果。为了获得输入查询的真实匹配, HERMIT 需要进一步删除所有假阳性结果。在本节中, 我们首先讨论现有 RDBMS 中的元组标识符方案, 以及 HERMIT 可以如何使用这些不同的方案并生成准确的查询结果。

4.3.1 元组标识符

建立在特定 (一组) 列上的二级索引提供了从列的键值到相应元组标识符的映射。根据 RDBMS 的性能要求, 元组标识符可以以两种不同的方式实现。HERMIT 的索引机制是通用的, 足以适用于这两种方案。

采用逻辑指针的 RDBMS 将对应元组的主键存储在每个辅助索引的叶节点中。逻辑指针背后的原理是, 元组位置的任何更新都不会影响二级索引。然而, 这种机制的缺点是 RDBMS 必须在每次进行辅助索引查找时对主索引执行额外的查找。像 MySQL 这样的流行 RDBMS 采用这种机制。

采用另一种标识符机制（称为物理指针）的 RDBMS 直接在每个辅助索引的叶节点中存储元组位置（在“块 ID+ 偏移量”的格式中）。虽然避免了在二次索引查找期间遍历主索引，但使用物理指针的 RDBMS 必须在任何一个位置更改后更新每个索引的对应叶节点。几个 DBMS（如 PostgreSQL）采用了这种方案

4.3.2 HERMIT 中的查询

HERMIT 为两种元组标识符方案生成精确的查找结果。图 3 显示了 HERMIT 查找机制的整个工作流程。我们列出了以下关键步骤：

步骤 1. TRS-TREE 查询-此步骤对 TRS-TREE 执行查找。结果是主机列上的一组范围和一组元组标识符。

步骤 2. 主机索引查找-此步骤使用返回的主机列范围作为输入对主机索引执行查找。结果是一组元组标识符，它与步骤 1 返回的标识符集进一步结合。

步骤 3. 主索引查找（可选）-只有当 RDBMS 采用逻辑指针作为元组标识符时，才会出现此步骤。它以返回的元组标识符集作为输入来查找主索引。结果是一组元组位置。

步骤 4. 基表验证-此步骤使用元组位置获取实际元组，并验证每个元组是否满足输入谓词。此步骤将所有合格结果返回到输入查询。

请注意，主索引也可以用作主机索引，在这种情况下，查找过程应相同。

与传统的二级索引方法相比，由于额外的主机索引查找阶段以及基表验证阶段，它的限制会带来额外的开销。当使用逻辑指针作为元组标识符方案时，开销会加剧，因为它涉及不必要的主索引查找以查找不符合条件的匹配。然而，在执行范围查询时，这种开销是微不足道的，这在索引中的辅助查询中很常见。这是因为与限定的元组相比，范围查询的误报数量非常少。此外，由于 TRS-TREE 极大地减少了空间消耗，它为现代主存 RDBMS 带来了巨大的好处，因为内存空间非常宝贵。

5 复现细节

本次论文复现工作没有参考任何相关源代码，该篇论文代码没有开源。在本节中，我将详细描述关键代码实现细节。

5.1 TRS-TREE 构建

RDBMS 可以根据用户的请求有效地构建 TRS-TREE。算法 1 详述了构建步骤。构造算法将基表 T、目标列 ID cidM、主机列 IDcidN 和目标列的完整范围 R 作为输入。范围 R 包含目标列中的最小值

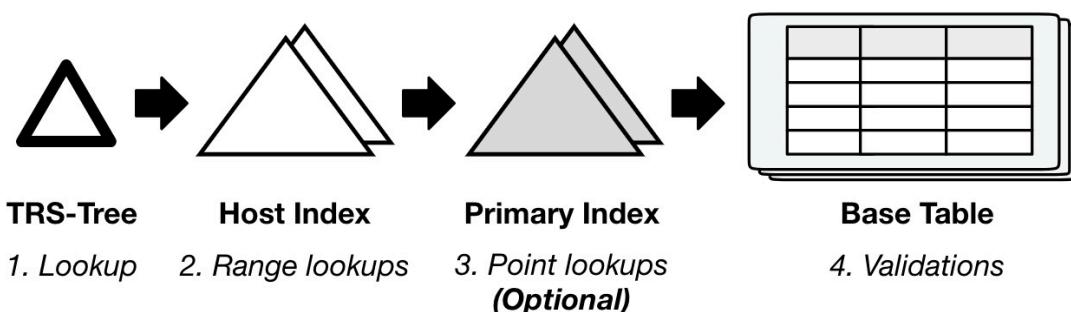


图 3: HERMIT 查找机制的工作流程

Procedure 1 Index construction in TRS-TREE

Input: base table T , target column ID cid_M , host column ID cid_N , value range R **Output:** TRS-TREE's root node $root$ Node $root(R)$;TmpTable $fullTmpTable = ProjecTable(T, cid_M, cid_N)$;FIFOQueue $queue$; $queue.Push(Pair(root, fullTmpTable))$;**while** $queue.IsEmpty()$ **do** Pair $pair = queue.Pop()$; Node $node = pair.GetKey()$; TmpTable $tmpTable = pair.GetValue()$; Compute($tmpTable$, $node$); **if** $\neg Validate(tmpTable, node)$ **then** Node[] $subNodes = SplitNode(node, nodeFanout)$; TmpTable[] $subTables = SplitTable(tmpTable, subNodes)$; **for** $i \in (0 \text{ to } nodeFanout - 1)$ **do** $| queue.Push(Pair(subNodes[i], subTables[i]))$; **end** **end** DeleteTmpTable($tmpTable$);**end****return** $root$;**Function** $Validate(tmpTable, node)$ **for** $entry \in tmpTable$ **do** **if** $entry.host \notin node.GetHostRange(entry.target)$ **then** $| node.outliers.Add(entry.target, entry.id)$; **end** **if** $node.outliers.Size() > outlierRatio * tmpTable.Size()$ **then** $| \text{return } false$; **end****end****return** $true$;

和最大值，可以很容易地从 RDBMS 的优化器统计信息中获得。该算法还需要一组 TRS-TREE 的预定义参数来进行计算。

该构建算法利用 FIFO（先进先出）队列以自顶向下的方式构建 TRS-TREE。FIFO 队列中的每个元素都是一对，包含一个 TRS-TREE 节点和节点对应的临时表。TRS-TREE 节点的临时表是 T 的子表，它选择节点范围内具有目标列的行，并仅投影目标列和宿主列以及每个元组的标识符。

TRS-TREE 的构建首先创建一个根节点，其范围设置为整个范围 R，然后将该根节点及其投影的临时表推送到 FIFO 队列中。然后，它重复执行以下步骤，直到 FIFO 队列为空：(1) 从 FIFO 队列中检索 ($tmpTable$, $node$) 对；(2) 计算节点的 β 、 α 和 ϵ ，并确定生成的线性映射是否能够很好地覆盖其对应的条目对；(3) 如果 (2) 返回 $false$ ，则将 $node$ 和 $tmpTable$ 分别拆分为多个子节点和子表，然后将相应的子节点和子表格对推回 FIFO 队列。

$compute$ 函数扫描节点的临时表以计算线性函数的参数。 $Validate$ 函数再次扫描临时表，并验证线性函数是否可以覆盖每对目标列值和主机列值。任何不符合条件的条目都将插入节点的异常值缓冲区。如果异常缓冲区超过预先定义的 $outlierRatio$ （异常缓冲区大小与节点覆盖的元组总数之比），则节点的线性函数被确定为不够好。在这种情况下，将触发步骤 (3)，该步骤将所有生成的内容丢弃，并将其拆分为固定数量（等于预先定义的 $nodeFanout$ 参数）的等范围子节点。用户可以通过设置参数

`maxHeight` 来限制-TREE 结构的最大深度。

TRS-TREE 的用户定义参数可以直接控制线性函数的置信区间以及离群值缓冲区大小，从而影响性能。

我们现在对算法 1.TRS-Trees 的 Compute 函数进行复杂性分析，以扫描每个树节点覆盖的元组，从而导出线性回归模型。如果他生成的 TRS-TREE 总是一个平衡的完整树，然后对相同高度的所有树节点运行线性回归需要对所有元组进行完整扫描。高度等于将执行全部扫描的 TRS-TREE。一个 TRS-TREE 受参数 `maxHeight` 的限制，我们可以得出平均和最坏情况下的复杂性为 $O(N)$ 。

5.2 TRS-TREE 查询

TRS-TREE 允许用户在目标列 M 上执行点和范围查找，以获得主机列 N 上的相应结果。TRS-TREE 的查找算法不返回与查询谓词完全匹配的结果，而是返回近似结果。HERMIT 将会对主机索引执行额外的查找，并进一步验证结果并生成精确的匹配。

算法 2 列出了 TRS-TREE 的查找算法的细节。该算法将 TRS-TREE 的根节点 $root$ 和目标列 M 上的查询谓词 P 作为输入。它生成主机列 N 上的一组值范围 RS 以及一组元组标识符 IS 作为输出。在不丧失一般性的情况下，我们认为 M 是一个具有两个元素的值范围：下边界带上边界带。点查询谓词的下限等于其上限。查找从根开始，并使用 FIFO 队列运行广度优先搜索。TRS-TREE 遍历队列中的每个节点，如果该节点是叶节点，则执行查找。在内部节点附近，TRS-TREE 检索其子节点，并检查每个子节点的范围是否与 P 重叠。任何重叠的子节点都将被推送到 FIFO 队列。查找算法继续迭代，直到队列为空。

TRS-TREE 通过执行以下步骤在叶节点上进行查找。首先，它计算查询谓词和节点值范围之间的

Procedure 2 Index lookup in TRS-TREE

```
Input: root node  $root$ , predicate  $P$ 
Output: range set  $RS$ , tuple identity set  $IS$ 
Set<Range>  $RS$ ;
Set<TupleID>  $IS$ ;
FIFOQueue  $queue$ ;
 $queue$ .Push( $root$ );
while  $queue$ .IsEmpty() do
     $node$  =  $queue$ .Pop();
    if  $node$ .IsLeaf() then
        Range  $r$  = Intersect( $node$ .range,  $P$ );
         $RS$ .Add( $node$ .GetHostRange( $r$ ));
         $IS$ .Add( $node$ .outliers.Lookup( $r$ ));
    else
        for  $child$  in  $node$ .children do
            if  $child$ .IsOverlapping( $P$ ) then
                 $queue$ .Push( $child$ );
            end
        end
    end
end
end
 $RS$  = Union( $RS$ );
return  $RS$  and  $IS$ ;
```

交集。区间结果是一个 value 区间。使用该范围，节点可以使用其线性函数来计算 N 上的估计范围，该范围涵盖了精确匹配。估算范围为 $(\beta \times r.lb + \alpha - \epsilon, \beta \times r.ub + \alpha + \epsilon)$ 或 $(\beta \times r.ub + \alpha - \epsilon, \beta \times r.lb + \alpha + \epsilon)$ ，取决于斜率 β 的正负号。并非所有匹配都被线性函数覆盖。因此，叶节点进一步从其 outlier 缓存区中检索一组元组标识符。这些标识符可用于直接从 RDBMS 中获取相应的元组，而无需查找主索引。在终止算法之前，TRS-TREE 在 RS 中的所有元素之间建立一个并集。这是因为不同叶节点生成的返回范围重叠。计算并集有助于减少 RS 中元素的数量

5.3 TRS-TREE 维护

在系统运行时，TRS-TREE 可以动态支持所有常用的数据库操作，包括插入、删除和更新。这使得 TRS-TREE 与现有的基于机器学习的解决方案大相径庭，后者依赖于长期的训练阶段从头开始重建索引结构。TRS-TREE 也可以在运行时重新组织 TRS-TREE 结构，以确保最佳查询性能。算法 3 演示了 TRS-TREE 如何处理插入和删除。

插入。TRS-TREE 中的元组插入快速执行，其内部结构在运行时几乎没有变化。给定要插入的元组的目标列值 m 、主机列值 n 和元组标识符 tid ，TRS-TREE 首先通过定位包含范围 m 的叶节点进行插入。之后，TRS-TREE 检查主机列的节点的对应范围是否可以覆盖（使用叶节点的线性函数）。如果没有，则 TRS-TREE 将该元组的需求插入异常值缓冲区。否则，插入算法将直接终止。某些叶节点的离群值缓冲区可能会变得太大，从而降低 TRS-TREE 的查找性能。在这种情况下，TRS-TREE[1] 结构重组以进一步拆分这些节点，正如我们将简短讨论的那样。

删除。TRS-TREE 中的元组删除算法与插入类似。然而，TRS-TREE 在定位叶节点后不执行任何计算。相反，它直接检查异常值缓冲区并删除相应的条目（如果存在）。频繁从索引中删除元组可能会导致空间利用率问题，这意味着 TRS-TREE 可能会使用更少的叶节点来准确捕获列相关性。TRS-TREE 也依靠结构重组来解决这个问题。

Procedure 3 Index insertion and deletion in TRS-TREE

```

Input: root node root, target column value m, host column value n, tuple ID tid
Function Insert(root, m, n, tid)
Node node = Traverse(root);
if n not in node.GetHostRange(m) then
| node.outliers.Add(m, tid);
|
end
Function Delete(root, m, n, tid)
Node node = Traverse(root);
node.outliers.Remove(m, tid);
Function Traverse(root, node, m)
if node.IsLeaf() then
| return node;
|
end
else
| for child in node.children do
| | if m in child.range then
| | | return Traverse(child)
| | |
| | end
| |
| end
|
end
```

6 实验结果分析

6.1 实验设置

测试数据集。实验测试中采用了合成的数据集，合成数据包含一个具有 3 个 8 字节数字列的表，分别为：colA, colB, colC, colB 的值由 colC 的某个相关函数产生，即 $\text{colB} = \text{Fn}(\text{colC})$ ，在 colB 上存在一个二级索引。在本次实验中，我们采用线性相关来作为两者间的相关性，元组数量为 2000 万个。

实验基准。我们使用传统 RDBMS 中的基于标准 B+-tree 的二级索引做对比，对范围查询、点查询、插入、内存消耗进行比较。B+-tree 的节点大小设置为 256 字节，我们将 HERMIT 的参数 nodeFanout、maxHeight、outlierRatio、erroBound 分别设置为 8, 10, 0.1, 2。

6.2 实验结果

范围查询。图 4 显示了具有不同查询选择性的范围查询吞吐量变化。结果表明，HERMIT 的表现非常接近基线方法的表现。在选择性为 0.01% 时，HERMIT 和基线方法分别每秒进行 1190, 1270 次操作。这个差距随着查询选择性的增加而逐渐减小。

点查询。图 5 显示了不同元组数量下的点查询吞吐量变化。尽管范围查询效率很高，但由于引入了误报，HERMIT 在点查找方面的性能有所下降。现在，我们通过增加数据库中元组的数量来衡量点查找吞吐量。可以观察到，HERMIT 的吞吐量比基线方法低 35%，这是因为 HERMIT 不仅要对二级索引和主索引的进行多次不必要的查找，还需要对基表的额外验证。

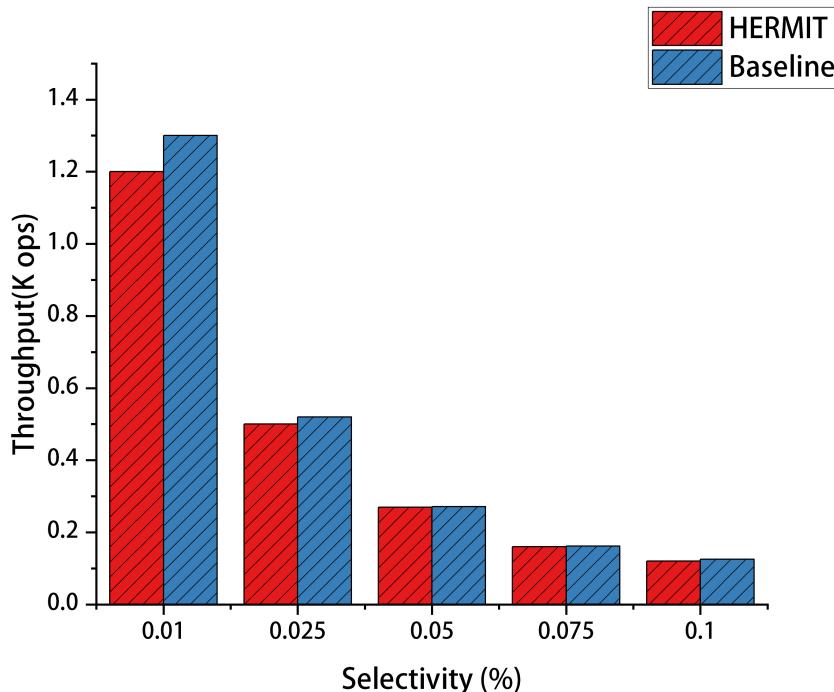


图 4: 具有不同查询选择性的范围查询吞吐量

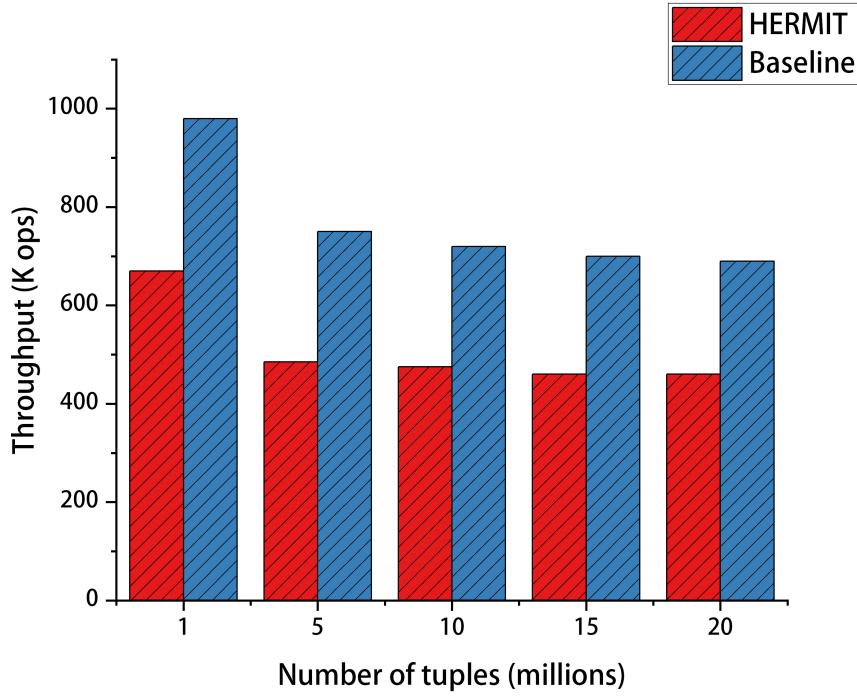


图 5: 具有不同元组数量的点查询吞吐量

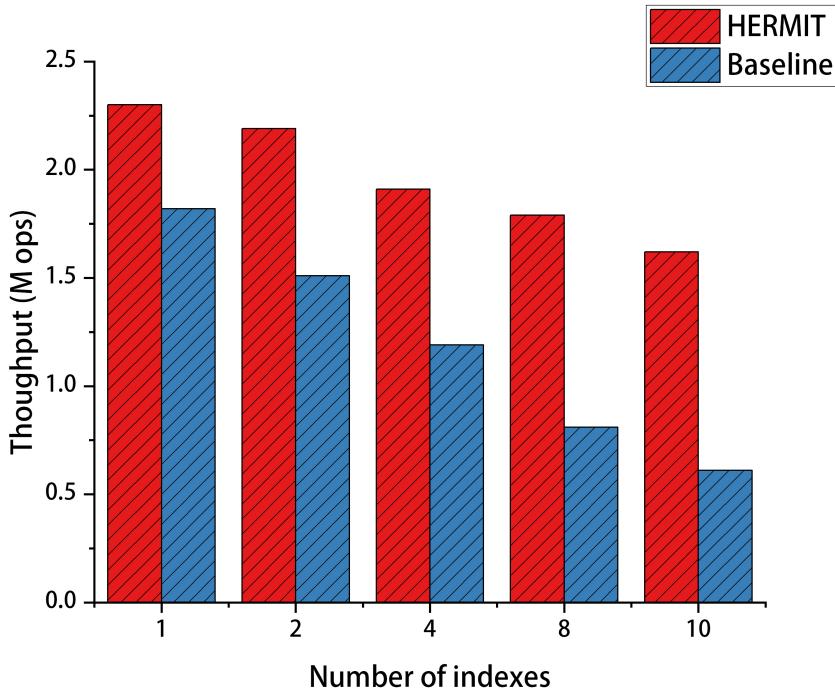


图 6: 具有不同索引数的插入吞吐量

插入。图 6 显示了不同索引数量下的插入吞吐量变化。结果表明,当将索引数设置为 10 时,HERMIT 每秒能处理 170 万次插入操作,这比传统的二级索引方案高 2.6 倍。主要原因是 HERMIT 的 TRS-TREE 需要在必要时更新叶节点的异常值缓冲区,这非常轻量级。使用基线方法,数据库应用程序必须花费 80% 以上的时间将元组插入到二级索引中。这表明了用于支持插入的传统索引方法的效率低下。

内存消耗。图 7 显示了不同索引数量下的总的内存消耗变化。当采用基线方法时,内存消耗量会随着新添加索引数量的增加而接近线性增长。具体来说,当支持 10 个辅助索引时,数据库使用了高达 8.5GB 的内存。相比之下,当采用 HERMIT 时,数据库仅消耗 2.4GB 内存,与基线方法相比,这

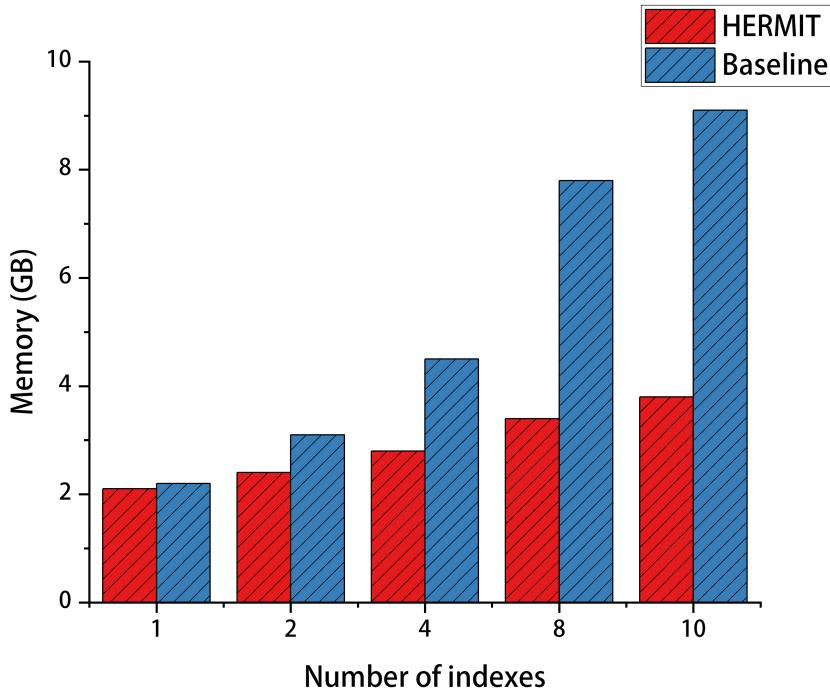


图 7: 具有不同索引数的总的内存消耗
在内存利用率方面有了显著提高。

7 总结与展望

本次课程复现了 2019 年 SIGMOD 上的一篇学习型二级索引的工作，该工作利用列之间的相关性或软函数依赖性进行建模，提出了一种新的二级索引结构——HERMIT。我们成功复现了该篇论文的工作，完成对 HERMIT 的代码实现，并通过数据集进行实验。目前实现过程中可以发现，HERMIT 旨在减少 RDBMS 中的存储空间大小，但相应的，会带来其查询性能的下降（下降较少，在可接受范围内）。因此，未来可着重研究如何在 HERMIT 工作的基础上，设计一种二级索引结构，既能节省存储空间消耗，又能保持甚至提升其查询性能。

参考文献

- [1] ZHANG H, ANDERSEN D G, PAVLO A, et al. Reducing the storage overhead of main-memory OLTP databases with hybrid indexes[C]//Proceedings of the 2016 International Conference on Management of Data. 2016: 1567-1581.
- [2] COMER D. Ubiquitous B-tree[J]. ACM Computing Surveys (CSUR), 1979, 11(2): 121-137.
- [3] LEHMAN T J, CAREY M J. A study of index structures for main memory database management systems [R]. University of Wisconsin-Madison Department of Computer Sciences, 1985.
- [4] RAO J, ROSS K A. Making B+-trees cache conscious in main memory[C]//Proceedings of the 2000 ACM SIGMOD international conference on Management of data. 2000: 475-486.
- [5] HENTSCHEL B, KESTER M S, IDREOS S. Column sketches: A scan accelerator for rapid and robust predicate evaluation[C]//Proceedings of the 2018 International Conference on Management of Data. 2018: 857-872.

- [6] ATHANASSOULIS M, AILAMAKI A. BF-tree: approximate tree indexing[C]//Proceedings of the 40th International Conference on Very Large Databases: CONF. 2014.
- [7] KRASKA T, BEUTEL A, CHI E H, et al. The case for learned index structures[C]//Proceedings of the 2018 international conference on management of data. 2018: 489-504.
- [8] ZHANG H, LIM H, LEIS V, et al. Surf: Practical range query filtering with fast succinct tries[C]//Proceedings of the 2018 International Conference on Management of Data. 2018: 323-336.
- [9] STONEBRAKER M. The case for partial indexes[J]. ACM Sigmod Record, 1989, 18(4): 4-11.
- [10] IDREOS S, KERSTEN M L, MANEGOLD S. Self-organizing tuple reconstruction in column-stores[C]//Proceedings of the 2009 ACM SIGMOD International Conference on Management of data. 2009: 297-308.
- [11] ATHANASSOULIS M, KESTER M S, MAAS L M, et al. Designing Access Methods: The RUM Conjecture.[C]//EDBT: vol. 2016. 2016: 461-466.
- [12] CHAUDHURI S, NARASAYYA V R. An efficient, cost-driven index selection tool for Microsoft SQL server[C]//VLDB: vol. 97. 1997: 146-155.
- [13] PAVLO A, ANGULO G, ARULRAJ J, et al. Self-Driving Database Management Systems.[C]//CIDR: vol. 4. 2017: 1.
- [14] BROWN P G, HAAS P J. BHUNT: Automatic discovery of fuzzy algebraic constraints in relational data [C]//Proceedings 2003 VLDB Conference. 2003: 668-679.
- [15] ILYAS I F, MARKL V, HAAS P, et al. CORDS: Automatic discovery of correlations and soft functional dependencies[C]//Proceedings of the 2004 ACM SIGMOD international conference on Management of data. 2004: 647-658.
- [16] KIMURA H, HUO G, RASIN A, et al. CORADD: Correlation aware database designer for materialized views and indexes[J]., 2010.
- [17] KIMURA H, HUO G, RASIN A, et al. Correlation maps: a compressed access method for exploiting soft functional dependencies[J]. Proceedings of the VLDB Endowment, 2009, 2(1): 1222-1233.
- [18] LARSON P A, LEHNER W, ZHOU J, et al. Cardinality estimation using sample views with quality assurance[C]//Proceedings of the 2007 ACM SIGMOD international conference on Management of data. 2007: 175-186.
- [19] ZHANG H, ILYAS I F, SALEM K. Psalm: Cardinality estimation in the presence of fine-grained access controls[C]//2009 IEEE 25th International Conference on Data Engineering. 2009: 505-516.
- [20] KIPF A, KIPF T, RADKE B, et al. Learned cardinalities: Estimating correlated joins with deep learning [J]. arXiv preprint arXiv:1809.00677, 2018.

- [21] WU Y, YU J, TIAN Y, et al. Designing succinct secondary indexing mechanism by exploiting column correlations[C]//Proceedings of the 2019 International Conference on Management of Data. 2019: 1223-1240.