# 学习型多维索引

摘要:多维表的扫描和过滤是现代分析型数据库引擎上的关键操作。为了优化这些操作的性能，数据库通常在单维或多维索引（例如 R 树）上创建聚集索引，或使用复杂的排序顺序（例如 Z 排序）。然而，这些方案通常很难调整，并且它们的性能在不同的数据集和查询中不一致。在本文中，我们介绍了Flood，这是一种多维内存读取优化索引，它通过联合优化索引结构和数据存储布局来自动适应特定的数据集和工作负载。与真实数据集和工作负载上最先进的多维索引或排序顺序相比，Flood 使用**谓词**进行范围扫描的性能提高了三个数量级。 我们的工作是构建端到端学习型数据库系统的基石。

1. 介绍

扫描和过滤是所有分析型数据库引擎的基础，近几年来针对数据库扫描和过滤性能已经取得了一些进步. 最重要的是，列存储[7]被建议延迟或完全避免访问与查询不相关的列（即属性）。相似地，还有很多技术略过那些并不匹配查询过滤器的记录。例如，事务数据库系统在单个属性上创建聚集 B 树索引，而列存储通常按单个属性对数据进行排序。两者背后的想法是相同的：如果数据是根据查询过滤器中存在的属性进行组织的，则执行引擎可以分别遍历 B 树或使用二分搜索，以快速将搜索范围缩小到该属性中的相关范围。 我们将这两种方法称为聚集列索引。

如果必须按多个属性过滤数据，则可以使用二级索引。不幸的是，它们的巨大存储开销和追逐指针引起的延迟使得它们仅适用于相当狭窄的用例，即当索引属性上的谓词具有非常高的选择性时；一种可能的解决方法是使用多维索引；这些可能是基于树的数据结构（例如，k-d 树、R 树或八叉树）或针对多个属性的专门排序顺序（例如，像 Z 排序或手工挑选的分层排序这样的空间填充曲线）。许多最先进的分析数据库系统使用多维索引或排序顺序来提高具有多列谓词的查询的扫描性能。例如，Redshift [1] 和 SparkSQL [4] 都使用 Z 排序来布局数据； Vertica 可以定义多个列的排序顺序（例如，先是年龄，然后是日期），而 IBM Informix 以及其他空间数据库系统则使用 R 树

然而，多维索引仍然有很明显的缺点。首先，这些方法很难调整。Vertica 能够对多个属性进行分层排序，这需要管理员仔细选择排序顺序。因此，管理员必须知道哪些列被一起访问以及它们的选择性，以便做出明智的决定。其次，其次，没有一种方法（即使调整得当）可以支配所有其他方法。正如我们的实验将表明的那样，最佳多维索引根据数据分布和查询工作负载而变化。第三，大多数现有技术无法完全针对特定的数据分布和查询工作负载进行定制。虽然它们都提供可调参数（例如页面大小），但它们不允许对特定数据集和过滤器访问模式进行更细粒度的定制。

为了解决这些缺点，我们提出了 Flood，这是第一个学习的多维内存索引。Flood 的目标是通过自动共同优化特定数据和查询分布的数据布局和索引结构，比现有索引更快地定位与查询过滤器匹配的记录。

Flood 的核心有两个关键思想。 首先，Flood 使用**示例查询过滤器工作负载来了解某些维度的使用频率、哪些维度一起使用以及哪些维度比其他维度更具选择性。**根据这些信息，Flood 自动定制整个布局，以优化给定工作负载的查询性能。其次，Flood 使用**经验 CDF 模型将多维且可能倾斜的数据分布投影到更均匀的空间中。**这种“扁平化”步骤有助于限制搜索点的数量，并且是实现良好性能的关键

Flood 基于学习的布局优化方法使其有别于其他多维索引结构。它允许 Flood 将其性能对准特定的查询工作负载，避免困扰某些索引的索引大小的超线性增长 [9]，并快速定位相关记录，而无需 k-d 树和超八叉树带来的高遍历时间，特别是对于较大范围的扫描 。

虽然 Flood 的技术是通用的，并且可能使广泛的系统受益，从 OLTP 内存事务处理系统到基于磁盘的数据仓库，但本文重点关注**提高多维索引性能（即减少不必要的扫描和过滤开销） 内存中的列存储。**由于 RAM 价格较低 [20] 以及可放入单台机器的主内存数量不断增加 [8, 19]，内存存储越来越受欢迎。此外，Flood 针对读取（即查询速度）进行了优化，但以写入（即增量索引更新）为代价，使其最适合静态分析工作负载，尽管我们的实验表明，调整到新的查询工作负载相对较快。我们设想 Flood 可以作为多维内存中键值存储的构建块，或者集成到商业内存（离线）分析加速器中，例如 Oracle 内存数据库 (DBIM) [35]。自我优化的能力使 Flood 的性能比其他最先进的技术高出三个数量级，同时存储开销通常要小得多。但更重要的是，Flood 全面实现了最优性：与我们所有数据集和工作负载上的次快索引技术相比，Flood 具有更好或至少达到同等性能。例如，在真实的销售数据集上，Flood 比经过调整的聚集列索引提高了 3 倍，比 Amazon Redshift 的 Z 编码方法提高了 72 倍。在源自 TPC-H 的不同工作负载上，Flood 比聚集列索引快 61 倍 但仅比 Z 编码快 3 倍。我们做出以下贡献：

1. 我们在内存列存储上设计并实现了 Flood，这是第一个学习的多维索引。Flood通过学习示例过滤器谓词分布来针对特定工作负载确定其布局。
2. 我们在一个合成数据集和三个真实世界数据集上评估各种多维索引，其中一个数据集的工作负载来自一家主要分析数据库公司的实际销售数据库。
3. 我们表明，Flood 在不同的过滤谓词和数据大小上实现了查询加速，并且其索引创建时间与现有的多维索引具有竞争力。
4. 相关工作

有丰富的多维索引工作集，许多商业数据库系统已经转向多维索引方案。例如，Amazon Redshift 按 Z 顺序组织点 [29]，它将多维点映射到单个维度上进行排序 [1, 34, 47]。对于空间维度，SQL Server 允许 Z 排序 [27]，而 IBM Informix 使用 R 树 [15]。 其他多维索引包括 K-d 树、八叉树、R\* 树、UB 树（也使用 Z 顺序）等（有关调查，请参阅 [32, 41]）。Flood 的底层索引结构可能与网格文件 [31] 最相似，网格文件有许多变体 [13,14,42]。但是，网格文件不会自动调整以适应查询工作负载，从而导致性能较差（§7）。 事实上，即使对于均匀分布的数据，网格文件的索引大小也往往会出现超线性增长[9]。

Flood 也不同于其他自适应索引技术，例如database cracking [16,17,38]。cracking的主要目标是通过每个观察到的查询增量分区数据来构建查询自适应增量索引。然而，cracking仅产生单维聚类索引，并且不能联合优化多个属性的布局。这限制了它在具有多维过滤器的查询中的有用性。 此外，cracking不考虑数据分布，仅适应查询； 另一方面，Flood 会适应查询和底层数据。

可以说与这项工作最相关的是**自动索引选择**[3,25,45]。然而，这些方法主要侧重于创建二级索引，而 Flood 针对给定的工作负载和数据分布优化存储和索引本身。对于聚合查询，**数据立方体**[11]是索引的替代方案。 然而，单独的数据立方体不足以满足任意过滤范围的查询，并且它们不能支持对查询记录的任意操作（例如，返回记录本身）。

最后，学习模型被用来取代/增强传统的 B 树 [5, 10, 23] 和二级索引 [21, 46]。 自我设计系统使用学习的成本模型来综合数据结构的最佳算法，从而产生一系列可能的设计，形成数据结构的“周期表”[18]。Flood以两种方式扩展了这些工作。首先，Flood 学习用于索引多个维度的模型。由于多个维度的点没有自然的排序顺序，因此 Flood 需要专门针对多维数据进行设计。其次，先前的工作仅专注于构建数据模型，而没有考虑查询。 Flood 还通过学习查询工作负载来优化其布局。 与[18]不同的是，Flood 将模型嵌入到数据结构本身中。

SageDB[22]提出了学习多维索引的想法，但没有描述任何细节。

1. 索引概览

Flood 是一种多维聚集索引，可加快选择一个或多个属性范围的关系查询的处理速度。 例如：

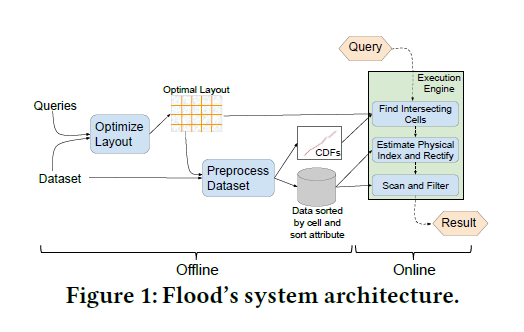
SELECT SUM(R.X)

FROM MyTable

WHERE (a ≤ R.Y ≤ b) AND (c ≤ R.Z ≤ d)

请注意，R.Z == f 形式的等式谓词可以重写为 f ≤ R.Z ≤ f。典型的选择通常还包括析取（即 OR 子句）。但是，这些可以分解为对不相交属性范围的多个查询； 因此我们关注 AND。

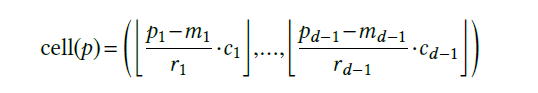
Flood 由两部分组成：(1) 离线预处理步骤，选择最佳布局，根据该布局创建索引；(2) 在线组件，负责在查询到达时执行查询（见图 1）。



从高层次上看，Flood 是基本网格索引的变体，它将 d 维数据空间划分为连续单元格的 d 维网格，以便每个单元格中的数据存储在一起。我们在第 3.1 节和第 3.2 节中描述了 Flood 的网格布局和在线操作。然后我们讨论了 Flood 的中心思想：如何针对特定查询工作负载自动优化网格布局的参数（第 4 节）。 本文的其余部分可互换地使用术语“属性”和“维度”，以及术语“记录”和“点”。

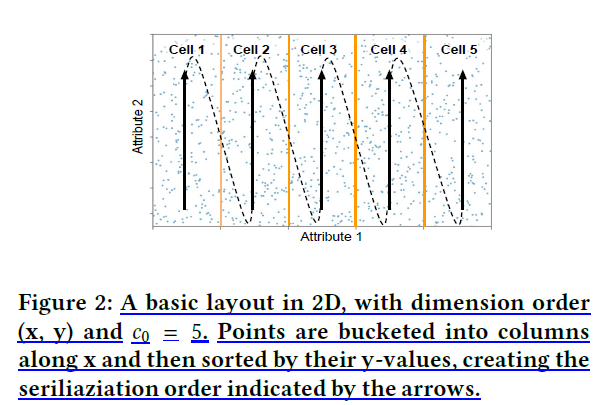
3.1数据布局

考虑 d 维度上的索引。 与单维情况不同，多维中的点没有自然的排序顺序。 我们的首要目标是对数据进行排序。我们首先对 d 属性进行排序。 如何选择排序的详细信息将在第 4 节中讨论，但出于说明目的，我们假设已给出排序。接下来，我们使用排序中的前 d − 1 维在数据上覆盖 (d−1) 维网格，其中排序中的第 i 个维度在其最小值和最大值之间分为 ci 等间隔的列。每个点都映射到该网格中的特定单元，即具有 d -1 属性的元组。 特别地，如果 Mi 和 mi 是数据沿第 i 维度的最大值和最小值，则将该维度的范围定义为 ri =Mi −mi +1。 那么点 p =(p1,...,pd ) 的单元格为：



请注意，单元仅由前 d−1 维确定； 第 d 个维度（排序维度）将用于对单元格内的点进行排序。

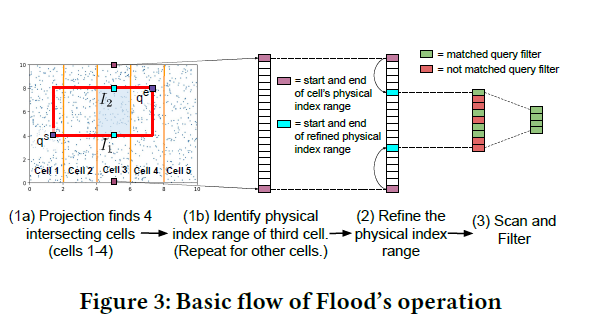
Flood**使用沿维度顺序对单元格进行深度优先遍历来对点进行排序**，即单元格按元组中的第一个值排序，然后是第二个值，依此类推。在每个单元格内，点按其在第 d 维度中的值排序。 图 2 说明了具有两个属性的数据集的排序顺序。



然后 Flood 通过这次遍历对数据进行排序。 换句话说，单元格 0 中的点（按排序维度排序）首先出现，然后是单元格 1，依此类推。平局是任意打破的。

* 1. 基本操作

Flood 接收一个过滤谓词作为输入，该过滤谓词由一个或多个属性的范围组成，并通过 AND 连接。 这些范围的交集定义了一个超矩形，Flood 的目标是准确找到并处理该超矩形内的点（例如，通过聚合它们）。 在较高层面上，Flood 执行以下工作流程（图 3）：



1. 投影：识别网格布局中与谓词的超矩形相交的单元格。 对于每个这样的单元格，确定存储中的位置范围，即包含该单元格的点的物理索引范围
2. 细化：**如果适用，利用每个单元内点的排序来缩短（或细化）必须扫描的每个物理索引范围**（第 3.2.2 节）。
3. 扫描：对于每个细化的物理索引范围，扫描并处理符合过滤器的记录。
   * 1. **投影。** 为了确定哪些点与过滤器匹配，Flood 首先确定哪些单元格包含匹配点。由于查询在 (d−1) 维网格中定义了一个“超矩形”，因此计算交集很简单。假设查询中的每个过滤器对于每个索引维度 i 都是 [qis ,qie ] 形式的范围。 如果查询中不存在索引维度，我们只需将范围的起点和终点分别设为−∞和+∞。 相反，如果查询包含索引中没有的维度，则在查询处理的此阶段将忽略该过滤器。

超矩形的“左下”角为 qs = (q0s, ... , qd-1s) ，“右上角”角 qe 也同样。两者如图 3 所示。然后，我们定义 相交单元的集合为 {Ci |cell（qs）i≤Ci≤cell（qe）i}。 Flood保存一个单元格表，记录每个单元格中第一个点的物理索引。 了解相交的单元格就可以轻松转换为一组要扫描的物理索引范围。

* + 1. **细化。** 当查询包含针对排序维度的过滤器时，Flood 利用每个单元格中的点按排序维度排序的事实来进一步细化要扫描的物理索引范围。特别是，假设查询包括对排序维度 R.S 的过滤器，其形式为 a ≤ R.S ≤ b。 对于每个单元，Flood 都会找到 R.S ≥a 的第一个点 I1 和 R.S ≤b 的最后一个点 I2 的物理索引。这将该单元格的物理索引范围缩小到 [I1,I2]。 查找 [I1,I2] 的最简单方法是在 C 内对排序维度中的值执行二分搜索。这是可能的，因为 C 中的点按排序维度按排序顺序连续存储。我们在5.2节讨论了一个使用模型的更快的细化方式。如果查询未过滤排序维度，Flood 会跳过细化步骤。

1. 优化网格

Flood 的网格布局有几个可以调整的参数，即构成网格的每个 d−1 维度分配的列数，以及使用哪个维度作为排序维度。调整这些参数是 Flood 发挥作用的关键方式 优化给定查询工作负载的性能。 我们发现 d−1 网格维度的排序不会显着影响性能。

在每个维度中添加更多列允许 Flood 扫描更紧密地限制真实查询过滤器的矩形，从而减少必须扫描的点的数量（图 4）。 然而，添加更多列也会增加子范围的数量，这会增加投影和细化的额外成本。 要达到正确的平衡，需要选择每个维度都有最佳列数的布局。

Flood还可以选择排序维度。 排序维度很特殊，因为它不会产生扫描开销； 给定一个查询，Flood 会在细化步骤中找到要扫描的精确子范围，以便保证扫描点的排序维度中的值位于所需的范围内。 另一方面，网格维度确实会产生扫描开销，因为某个列可能仅部分位于查询矩形内。 因此，排序维度的选择会对性能产生重大影响。

在每个维度中选择最佳列数是很困难的，因为它取决于许多相互作用的因素，包括该维度上查询过滤的频率、该维度上过滤器选择性的平均值和方差，以及与两个维度中其他维度的相关性。 数据和查询工作负载。 由于类似的原因，最优排序维度也很难选择。 因此，我们使用基于成本模型的方法优化布局参数。 我们首先描述成本模型，然后介绍 Flood 用于优化布局的过程。

4.1 成本模型

将 d 维度上的布局定义为L = (O, {ci }0≤i<d−1), 其中 O 是 d 维度的排序，其中第 d 维度是排序维度，{ci }0≤i<d−1 是剩余 d−1 网格维度中的列数。

给定数据集 D 和布局 L，我们将任何查询 q 的查询时间建模为三个部分的总和，这对应于第 3.2 节中的查询流的步骤。 每个部分由一些可测量的统计量 N 组成，该统计量乘以变量权重 w（数据集 D、查询 q 和布局 L 的函数），以生成该步骤所需时间的估计：

1. 投影对查询时间贡献为 wpNc，其中Nc是落在查询矩形内的单元格的数量，wp是对单个单元格执行投影的平均时间。 在所有数据集、查询和布局中，weightwp 并不是恒定的。 例如，识别沿单个网格维度（在线性存储介质上相邻）的单元块比识别沿不相邻的多个网格维度的超立方体单元更快。
2. 细化会给查询时间贡献 wrNc，其中 wr 是对单元格执行细化的平均时间。 如果查询 q 未在排序维度上进行筛选，则将跳过细化并且 wr 为零。 此外，如果单元较小，则 wr 较低，因为该单元的分段线性 CDF（第 5.2 节中解释）可能不太复杂，并且可以更快地进行预测。
3. Scan 对查询时间贡献 wsNs，其中 Ns 是扫描点的数量，ws 是执行每次扫描的平均时间。 权重取决于过滤的维度数量（维度越少意味着每个扫描点的查找次数越少）、扫描的运行长度（运行时间越长，局部性越好）以及有多少扫描落在精确的子范围内（第 7.1 节中进行了解释） ）。

把上述放在一起，我们模型的查询时间是

Time(D,q,L)=wpNc +wrNc +wsNs (1)

给定数据集 D 和查询工作负载 {qi }，我们找到最小化等式 2 的平均值的布局 L。 1 对于所有 q ∈ {qi }。

**4.1.1 校准成本模型权重** 由于四个权重参数 w = {wp,wr ,ws } 根据数据、查询和布局而变化，因此 Flood 使用模型来预测 w。这些权重模型的特征是在具有特定布局的数据集上运行查询时可以测量的统计数据。 这些统计信息包括 N = {Nc ,Ns }、单元格总数、可过滤单元格大小的平均值、中位数和尾分位数、查询过滤的维度数、每个单元格中访问点的平均数量以及精确子范围内访问的点的数量。

正如我们在第 7.7 节中所示，权重模型在不同的数据集和查询工作负载中都是准确的。 特别是，当新数据到来或查询分布发生变化时，Flood只需要评估现有模型，而不需要训练新模型。