FITing-Tree: 一种数据感知的索引结构

袁佳楠

摘要

索引是数据库系统检索数据的有力工具，良好的索引结构可以有效提升存储系统的性能。然而，随着数据规模的不断增大，传统索引将耗费极大的系统资源。一项研究显示，在现代数据库系统上运行 TPC-C 工作负载，仅索引就占了 55% 的内存空间。这个空间开销消耗了宝贵的内存资源，也限制了用来存储新数据或处理现有数据的空间，导致系统整体性能的下降。

本报告重点研究并复现一种名为 FITing-Tree 的数据感知的索引结构，它使用轻量级学习模型来拟合数据分布。本工作（1）复现了 FITing-Tree 的索引结构；（2）复现了 FITing-Tree 的数据分区算法；（3）复现了代价模型，用来确定 FITing-Tree 拟合数据的误差界限，以权衡索引的查询性能和存储开销；（4）对 FITing-Tree 的读写性能和存储开销开展了充分的实验，结果表明 FITing-Tree 取得了与传统索引相当的性能，同时极大降低了存储空间。

关键词：数据库；索引；数据感知；线性数据分区；代价模型

1 引言

索引是数据系统的核心组件之一。根据索引机制的不同，索引可以分为若干种，其中，树型索引在索引结构中占据关键一席，对分析型处理应用和事务型处理应用的性能起着重要的作用。然而，对于内存数据库，树状索引经常浪费了大量的内存。最近的一项研究表明，在最先进的内存数据库系统运行 TPC-C 工作负载，仅索引就占用了 55% 的内存空间[1]。这个空间开销限制了存储新数据的空间，也限制了用于处理现有数据的空间，导致了系统整体性能的下降。

为了降低树型索引（如 B+ 树）的存储开销，人们提出了多种索引压缩方案，通过减少键的冗余、减小索引节点中键的大小、使用先进的编码方式来压缩索引。虽然这些压缩方案都能减少索引结点的大小，但这些索引占用的内存空间，仍然会随着索引中键的数量增加而增长，在自动驾驶、物联网场景下，这种数据增长尤为明显。

针对这个问题，本报告着重研究并复现 FITing-Tree，这是一种数据感知的索引结构，使用分段线性函数拟合数据趋势。和传统索引在叶子节点使用固定大小的页来存放数据不同，FITing-Tree 使用分段线性函数来拟合底层的数据分布。相比于传统 B+ 树索引，FITing-Tree 利用数据的分布规律来建立索引，使得 FITing-Tree 对内存地占用显著降低。FITing-Tree 的关键参数是误差界限，代表任意一个键的预测位置和实际位置的最大差距。误差界限可以调节，以便满足不同场景下索引的查询性能和空间占用要求。本研究提出一个代价模型，通过指定查询延迟要求和存储预算，FITing-Tree 可以确定一个合适的误差界限。

使用线性函数拟合分布使得 FITing-Tree 成为一种学习索引。虽然使用分段线性函数来拟合数据并不新奇，但这种技术从没用在索引上，因此也没有考虑索引应该支持的操作。笔者复现的 FITing-Tree，既将分段拟合思想引入索引结构中，又实现了索引所应该支持的操作。多种真实数据集上的评估表明：FITing-Tree 取得了与传统 B+ 树索引相当的性能，同时数量级地减少了存储空间。

复现的主要工作如下：

- 实现了 FITing-Tree 线性划分数据段的算法。
- 实现了 FITing-Tree 的索引结构和相应的读写操作。
- 实现了一个代价模型来辅助确定误差界限的选取。
- 使用人造数据集和真实数据集，对 FITing-Tree 的读写性能和存储进行了充分的评估。

2 研究动机

索引是数据库系统中提升数据检索效率最有力的工具。目前的索引对下层数据是“无感”的。也就是说，不管底层数据的分布规律如何，传统索引结构都是将数据简单组织成树（如 B+ 树）的结构。例如，假设现有键的序列是 1 到 n 的连续自然数，使用扇出系数为 b 的 B+ 树来存储这些键，则空间复杂度为 $O(\log_b n)$。虽然上述空间复杂度是对数复杂度，然而，我们处在数据急速增长的时代，庞大的数据促使索引占据的空间势必越来越大。巨大的空间开销不仅增加了存储成本，还限制了存储新数据的空间，也限制了用于处理现有数据的空间，降低了系统的性能。

针对键在存储系统中有序分布的特点，我们提出可以使用学习的方式来拟合底层数据的分布。在上述例子中，假如索引感知到底层数据是 1 到 n 的连续自然数分布，则可以用 $pos(key) = key$ 函数来拟合数据分布，而无需存储底层数据。给定一个查询键 k，索引可以计算得到该键的存储位置，进而读取出数据记录。这种情况下，索引的空间复杂度为 $O(1)$。这种学习型索引仅需存储少量的参数，可以大大降低索引占用的存储空间。因此，本报告着重对数据感知型的学习索引进行研究。

3 相关工作

3.1 索引压缩

传统索引如 B+ 树会占据较多存储空间，于是，人们提出了索引压缩技术。为了压减 B+ 树索引的大小，人们通过前后缀截断、字典压缩和键规则化[2-4]，来压缩树型索引内部节点的大小。事实上，由于本工作复现的 FITing-Tree 内部节点也采用 B+ 树进行组织，因此上面提到的这些压缩方法也可以用到 FITing-Tree 中。与 B+ 树压缩类似，人们对 bitmap 索引的压缩也进行了研究[5-9]。然而，这些研究仅仅只是对 bitmap 这种索引结构起作用，不能推广到树型索引。

Correlation Maps[10]利用非聚簇属性和聚簇属性之间的相关关系来建立索引。FITing-Tree 索引不假设属性之间存在相关关系，并使用变长的页段来拟合数据分布。

FAST[11]也是一种树型索引，它利用现代硬件的特性，用更有效的压缩方式来存储数据。ART[12]索引利用 CPU 缓存来做内存内索引。LSM-tree[13]是为写密集工作负载而设计，Monkey[14]在 LSM-tree 的基础上进一步平衡了索引性能和内存占用。这些结构都可以被 FITing-Tree 采用，来更高效存储索引，或者用来优化读密集工作负载。

3.2 分区索引和自适应索引

分区索引目标在于降低存储开销，因为每个分区索引存储的仅仅是索引数据的一个子集。Tail Indexes[15-16]通过只存储至关重要的数据来压减存储空间。与之不同，FITing-Tree 索引的仍是全量数据。Database Cracking[17]基于历史查询数据，对数据表里面的一列数据进行重排列，使得在没有二级

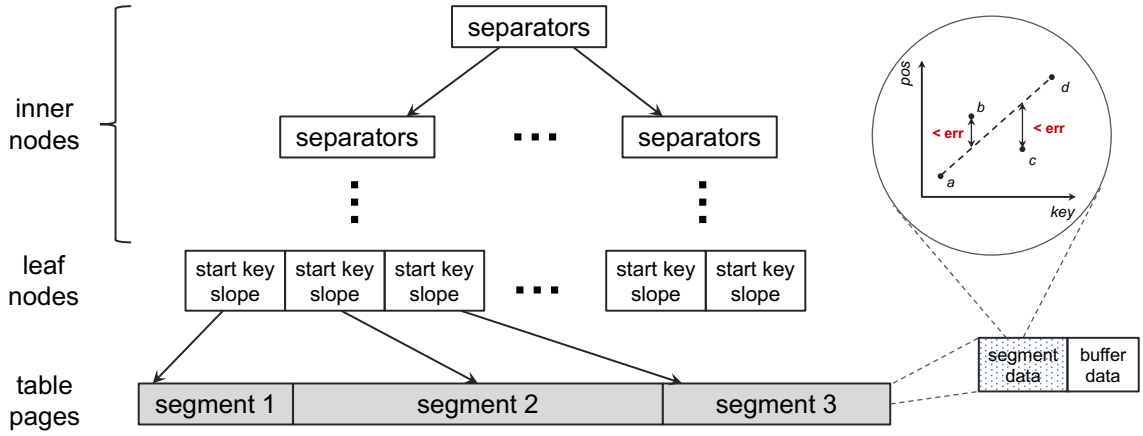


图 1: FITing-Tree 工作概览

索引的情况下，还能高效查询数据。与之不同，FITing-Tree 不考究历史查询，只对底层数据分布进行感知。

3.3 函数近似

用线性回归来拟合数据分布的做法不新奇，有很多的工作，如 ParaPFor[18]使用分段线性回归来逼近数据分布。与这些方法不同，FITing-Tree 只需要考虑单调递增函数和不相交的线性段。并且，过去这些线性拟合数据分布的算法，并没有用在索引上，更没用考虑到使用分段线性回归算法建立索引后，如何支持数据的查询和插入。

对于时序数据，依然可以用线性回归算法来拟合数据分布模式[19]，这与 FITing-Tree 原理相通。但不同的是，FITing-Tree 通过代价模型，对索引的查询性能和存储开销提供了保证。

4 本文方法

4.1 本文方法概述

从宏观层面来看，索引可以看成是一种将输入键映射到存储位置的函数。FITing-Tree 使用分段线性回归方法，将键空间划分为一系列不相交的线性段，如图 1所示。图的核心部分是 FITing-Tree 简要的数据结构，它分为内部节点和叶子节点，内部节点像 B+ 树一样存储，起导航作用；叶子节点不再存储实际数据，而是存储数据分段的起始键（$start\ key$）和对应分段的斜率（$slope$），有了这两个参数，就可以计算出键大致的存储位置。叶子节点指向底层页表存储的分段数据，每个分段都由线性分段数据和缓冲数据组成。其中，缓冲区是为了支持高效写入操作而设计的一块区域。在线性分段中查找目标键，最重要的一个参数是误差界限。一个键通过线性段预测出来的存储位置，和它实际存储的位置，不会超过这个界限值。

本工作着重对（1）数据分段算法；（2）FITing-Tree 结构和读写操作；（3）代价模型 三个方面进行研究和复现。

4.2 数据分布近似方法

在数据库系统中，主键是按序存储的，这种存储的规律性使得我们可以建立起一个将键映射到存储位置的模型。换句话讲，这个模型刻画了底层数据的分布规律。现实中数据的分布可能是复杂的，因此若想要精确将键映射到存储位置会花费较大的开销，这在现实系统中是不可行的。因此，FITing-Tree 的目标不在于精确预测数据的存储位置，而是预测数据的大致存储位置。

已有工作[20]表明：使用深度神经网络来拟合数据分布的开销较大。虽然理论上可以用其他回归模

型来拟合数据分布，但是分段线性模型的计算开销小很多，且存储参数少，从而有效减少了索引建立时的开销，提升了插入新数据的性能（无需重新训练整个模型，如神经网络）。因此，FITing-Tree 使用分段线性回归来拟合数据的分布。这种分段方式将键空间划分为若干个子空间，对每一个子空间使用线性回归算法来拟合数据分布，如图 1右侧圆圈内所示。

由于 FITing-Tree 是一种近似的索引，因此预测键位置和实际键位置存在误差，这个误差最大不超过误差界限。一个分段的误差界限定义如式 1所示。

$$error = max(|pred\_pos(k) - true\_pos(k)|) \quad \forall k \in keys \tag{1}$$

FITing-Tree 的一个线性分段覆盖了全部数据的一个连续子集，这个子集所有数据的存储位置都被限定在线性分段的上下限之内。线性分段过程产生一系列能够拟合数据分布的线性段。在分段结束之后，FITing-Tree 将每一线性段的起始键和斜率存进 B+ 树的叶子节点中，从而减少了索引的内存空间开销。

4.3 索引结构

本工作复现了 FITing-Tree 作为主键索引时的结构。前面提到，与传统 B+ 树不同，FITing-Tree 的叶子节点并不存储键数据，而是存储所索引线性段的斜率、起始键。而 FITing-Tree 的内部节点则和 B+ 树一样，存储的是卫星数据，起导航作用。对于一个插入或查询操作来说，最终都需要落到叶子节点做进一步的查询。当抵达叶子结点时，FITing-Tree 需要做额外的工作。对查询而言，需要结合叶子结点所索引的线性段的起始键和斜率来计算键大致的存储位置；然后在该存储位置的上下误差界限内，使用二分查找等高效查询算法来确定数据存储的最终位置。对插入而言，我们需要保证插入数据不会破坏原有线性分段的规则。本工作实现了异地插入更新的策略：新来的数据会暂存到一个线性段的缓冲区中，当缓冲区满，再对该线性段重新执行分段算法。

4.4 代价模型

影响 FITing-Tree 性能的关键参数是误差界限，因此，如何选择一个合适的误差界限是值得探讨的问题。对数据库索引而言，查询性能和存储开销是两个重要的性能指标。因此，我开发了一个代价模型来辅助选取合适的误差界限。

对于优先保障查询性能的情况，代价模型能够在满足查询性能的要求下，寻找到一个合适的误差界限值，使得索引占用的空间是最小的。对于优先保障存储空间的情况，开销模型能够在满足存储容量的限制下，寻找到一个合适的误差界限值，使得查询性能是最好的。

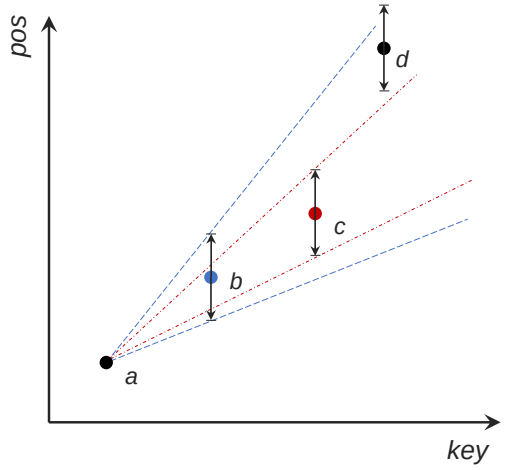


图 2: SHRINKING-CONE 算法

Procedure 1 SHRINKING-CONE 分段

Input: 按键值升序排序的键序列 *keys*
Output: 线性段集合 *segs*
$slope_{high} \leftarrow \infty$
$slope_{low} \leftarrow 0$
the first key is the first segmentation origin
for *every* $k \in keys$ **do**
    if *k is inside the cone:*
        $update\ slope_{high},\ slope_{low}$
    else
        $store\ the\ old\ segment\ in\ segs$
        $k\ is\ the\ origin\ of\ a\ new\ segment$
        $slope_{high} \leftarrow \infty$
        $slope_{low} \leftarrow 0$
end
return *segs*

5 复现细节

5.1 与已有开源代码对比

FITing-Tree 论文作者没有公开源代码，笔者从传统数据结构 B+ 树的代码开始修改，逐步改成 FITing-Tree 的代码。

5.2 线性分段 SHRINKING-CONE 算法

FITing-Tree 使用的分段算法是 SHRINKN-CONE 算法。从算法的名称可以推测，这个算法是使用一个“缩小的锥形”来拟合数据。如图 2所示。

图 2表示的是键的存储位置随键的变化关系，横轴是键（key），纵轴是键的存储位置（pos）。假设从点 a 开始拟合，当分段算法扫描到点 b 时，计算 b 位置的上下界，并将 b 的上下界分别与点 a 相连，即得到蓝色虚线围成的锥形，在这个锥形内的任意射线，都可以在误差界限 e 内找到 a 和 b。当分段算法扫描到点 c 时，计算 c 的上下界，并将 c 的上下界分别与点 a 相连，即得到红色虚线围成的锥形，原来蓝色的锥形缩小为现在红色的锥形。在这个红色锥形内的任意射线，都可以在误差界限 e 内找到 a、b 和 c。当点 d 加进来之后，发现 d 的上下界均不在红色锥形内部。因此，从点 d 开始，需要建立一个新的分段来拟合数据。

SHRINKING-CONE 算法的伪代码表示如算法 1所示。

5.3 索引操作

5.3.1 索引查询

FITing-Tree 索引的查询分为两个阶段。第一个阶段和 B+ 树的检索过程一致，通过树内部节点的卫星数据，从根节点定位到叶子节点，这个过程记为 $TraverseBPTree$。第二个阶段是线性段查询，为了计算输入键的存储位置，根据线性分段的斜率和分段的起始键，通过式 2计算得到键的近似存储位置。

$$pred\_pos = (k - s.start) \times s.slope \tag{2}$$

Procedure 2 索引查询

Input: 查询键 *k*
Output: 键 k 对应的数据记录 *val*
$seg \leftarrow TraverseBPTree(k)$ // $traverse\ B+\ Tree$
$pos \leftarrow (\ k\ -\ seg.start)\ \times\ seg.slope$
$val \leftarrow BinarySearch(seg.data, pos - err, pos + err, k)$
if val is null:
  $val \leftarrow BinarySearch(seg.buffer, k)$
return val

Procedure 3 异地更新索引插入

Input: 插入键 *k*
$seg \leftarrow TraverseBPTree(k)$
$seg.buffer.insert(k)$
if $seg.buffer.isFull()$:
  $keys$ = Combine-and-Reorder($seg.data, seg.buffer$)
  $segs$ = SHRINKING-CONE-SEGMENTATION($keys$)
for *every* $s \in segs$
do
  $tree.insert(s)$
end
$tree.remove(seg)$

由于 $pred\_pos$ 是近似存储位置，真实键的存储位置（若存在）与其预测位置相差不超过误差界限 e。因此，计算得到式 3的查询范围，在该范围内使用二分查找来查询目标键。

$$true\_pos \in [pred\_pos - e, pred\_pos + e] \tag{3}$$

查询算法的伪代码如算法 2所示。

5.3.2 索引插入

FITing-Tree 的插入策略有就地插入和异地插入更新两种方案，但由于就地插入更新会引发大量元素的搬迁操作，导致性能下降，因此，本工作重点针对异地插入策略进行复现。

异地插入的主要思想，是先将到插入数据所在的线性段，将数据存在该线性段的缓冲区里面。等到缓冲区满，将该线性段所管理的数据和缓冲区数据进行融合，重新执行分段算法，将新生成的线性段插入 FITing-Tree 中，并将旧的线性段从 FITing-Tree 中删除。

异地插入具体操作如算法 3所示。

5.4 代价模型

对于优先保障查询性能的情况，代价模型能够在满足查询时延的要求下，寻找到一个合适的误差界限值，使得索引占用的空间是最小的。我们用 L_{req} 来表示用户对查询时延的要求，表示查询时延不超过 L_{req}（纳秒），误差界限 e 如式 4所计算。

$$e = \arg \min_{\{e \in E | LATENCY(e) \leqslant L_{req}\}} (SIZE(e)) \tag{4}$$

其中，$LATENCY(e)$ 的计算如 5所示。

$$LATENCY(e) = c \times [\log_b (S_e) + \log_2 (e) + \log_2 (buf)] \tag{5}$$

式中 $\log_b (S_e)$ 表示在 FITing-Tree 搜索到目标线性分段的时间，b 是 FITing-Tree 的节点扇出系数，

$\log_2(e)$ 表示在目标线性分段所界定的误差范围内执行查询的时间，$\log_2(buf)$ 表示在异地插入更新策略下搜索缓冲区的执行时间，c 是一个常数，表示访问 CPU 片上缓存所需要的时间。

对于优先保障存储空间的情况，代价模型能够在满足存储容量的限制下，寻找到一个合适的误差界限值，使得查询性能是最好的。我们用 S_{req} 来表示用户对存储容量的限制，表示存储容量不超过 S_{req}（字节），误差界限 e 如式 6所计算。

$$e = \arg \min_{\{e \in E | SIZE(e) \leqslant S_{req}\}} (LATENCY(e)) \tag{6}$$

其中，$SIZE(e)$ 的计算如 7所示。

$$SIZE(e) = (\log_b(S_e) \times S_e \times 16B) + (S_e \times 24B) \tag{7}$$

式中 $\log_b(S_e) \times S_e \times 16B$ 是 FITing-Tree 内部节点的大小（内部节点存放 1 个 8 字节的卫星数据和 1 个 8 字节的指向下一个树节点的指针），$S_e \times 24B$ 是 FITing-Tree 叶子节点的大小（叶子节点存放 1 个 8 字节的斜率、1 个 8 字节的起始键和 1 个 8 字节的指向所管理线性分段的指针）。

5.5 实验设置

为了验证复现工作 FITing-Tree 的有效性，我们按照如下思路开展实验：

- 硬件和系统平台：11th Gen Intel(R) Core(TM) i7-11700K @ 3.60GHz（8 核），16GB RAM，50GB SSD，Ubuntu 20.04.3 LTS 64 位，cmake 3.9.0
- 数据集：4 个人造数据集（linear，segment-1%，segment-10%，normal），4 个真实数据集（books，fb，osmc，wiki）。其中，linear 是线性分布的数据；segment-1% 是每隔 1% 的数据就会发生一次跳变的数据集，segment-10% 同理；normal 是符合高斯分布规律的数据；books 是 Amazon 上书籍的销量数据；fb 是 Facebook 用户的 ID 数据；osmc 是在 OpenStreetMap 上采样得到的数据；wiki 是维基百科词条编辑的时间戳数据。
- 基线：B+ 树
- 性能指标：查询时延、p99 查询时延、读吞吐率、插入时延、p99 插入时延、写吞吐率、索引空间大小

5.6 代码使用

5.6.1 FITing-Tree 代码

- 进入到项目根目录，新建一个名为 build 的文件夹
- 进入 build 文件夹，执行 cmake -DCMAKE_BUILD_TYPE=Release ..；待生成 Makefile 文件后，执行 make -j 命令
- 回到项目根目录，chmod +x ./exp.sh 命令，再执行./exp.sh 脚本，即可运行测试

5.6.2 B+ 树基线代码

- 首先确保基线代码文件夹父路径和 FITing-Tree 代码文件夹父路径完全一致
- 进入到项目根目录，新建一个名为 build 的文件夹
- 进入 build 文件夹，执行 cmake -DCMAKE_BUILD_TYPE=Release ..；待生成 Makefile 文件

后，执行 make -j 命令

- 回到项目根目录，chmod +x ./exp.sh 命令，再执行./exp.sh 脚本，即可运行测试

5.7 创新点

- 使用机器学习线性回归的方式来替代传统索引方法
- 开发代价模型来选择合适的误差界限值
- 经过良好的设计，FITing-Tree 的内存空间远小于 B+ 树的内存空间

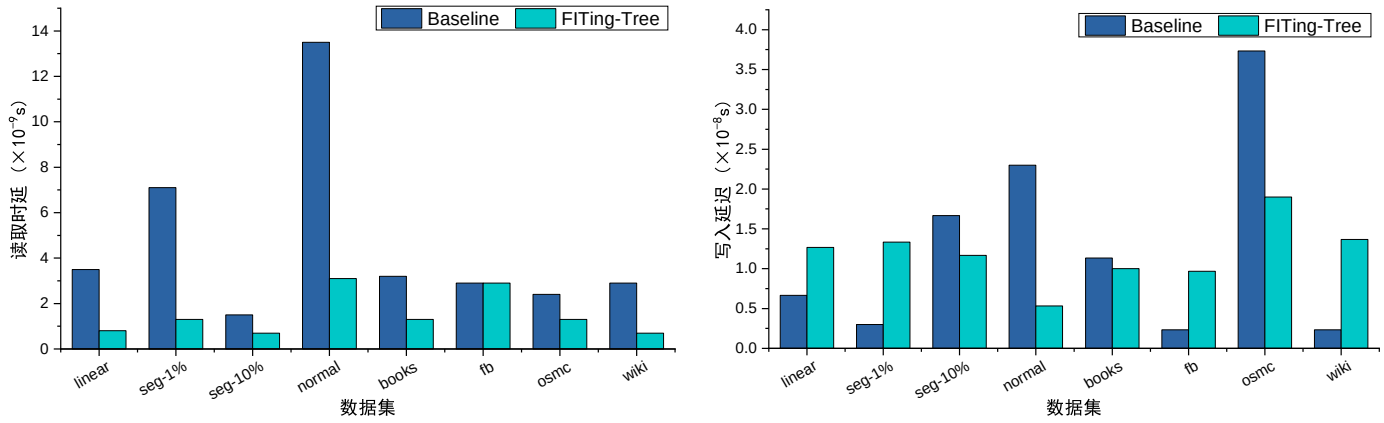
6 实验结果分析

6.1 平均查询时延

平均查询时延实验结果如图 3(a)所示。可以看到，在全部数据集上，FITing-Tree 的查询延迟和 B+ 树接近或有所下降。相较于 B+ 树，FITing-Tree 取得了 $1.0\times-5.5\times$ 查询性能上的提升。FITing-Tree 之所以可以在全部数据集上都取得良好的查询性能，是因为其对底层数据分布的规律进行了感知。相比于 B+ 树将全量数据都存储在 B+ 树（包括内部卫星节点和叶子数据节点）中，FITing-Tree 将键空间划分为若干个线性空间，再将每个线性段的代表键（即起始键）插入到树的内部。通过这种方式，一方面，FITing-Tree 无需再存储全量数据，树内部的卫星导航节点数量减少，树的高度降低，从根节点到叶子节点的导航时间缩短；另一方面，FITing-Tree 在搜索得到候选数据段后，通过线性计算得到目标数据的大致存储位置，再在该位置附近执行查找，这样就减少了查询范围，加快了查询速度。

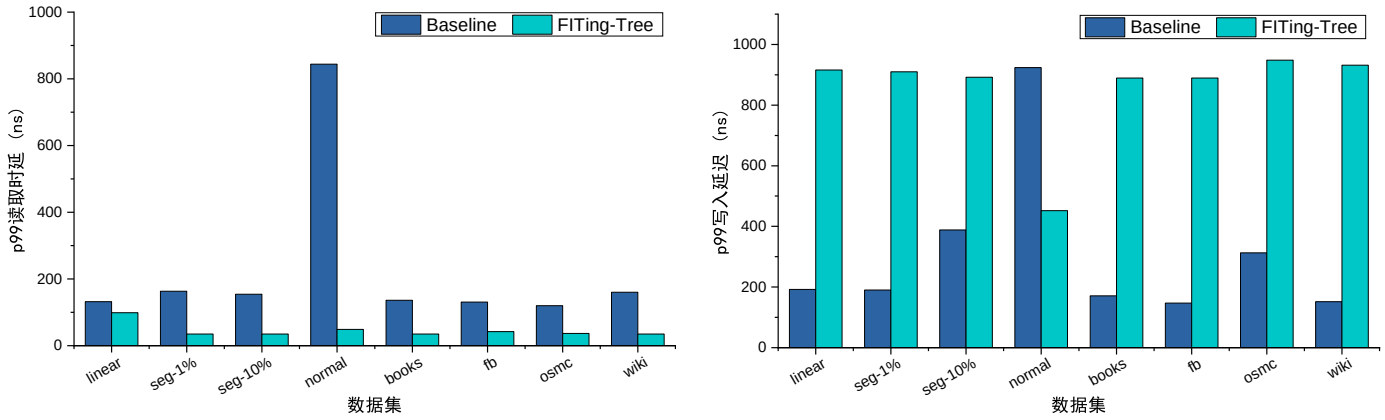
6.2 平均插入时延

平均插入时延的实验结果如图 3(b)所示。可以看到，FITing-Tree 在 seg-10%、normal、books、osmc 数据集上的写入延迟比 B+ 树低，在 linear、seg-1%、fb、wiki 数据集上的写入延迟比 B+ 树高。总的来说，FITing-Tree 的写入性能是 B+ 树的 $0.2\times-2.0\times$。FITing-Tree 的插入性能与原始数据分布和插入数据有关。当插入数据比较均匀地落在各个数据段上时，这个时候缓冲区能对插入数据起到较好的缓冲作用。相比于 B+ 树就地更新引发较多元素的移动，FITing-Tree 异地更新提高了插入性能。当插入数据比较集中地落在某些数据段上时，会导致缓冲区很快就达到存储限制，触发分段算法。相较于简单的插入操作，分段算法带来的时间开销更大，因此导致了插入性能的下降。



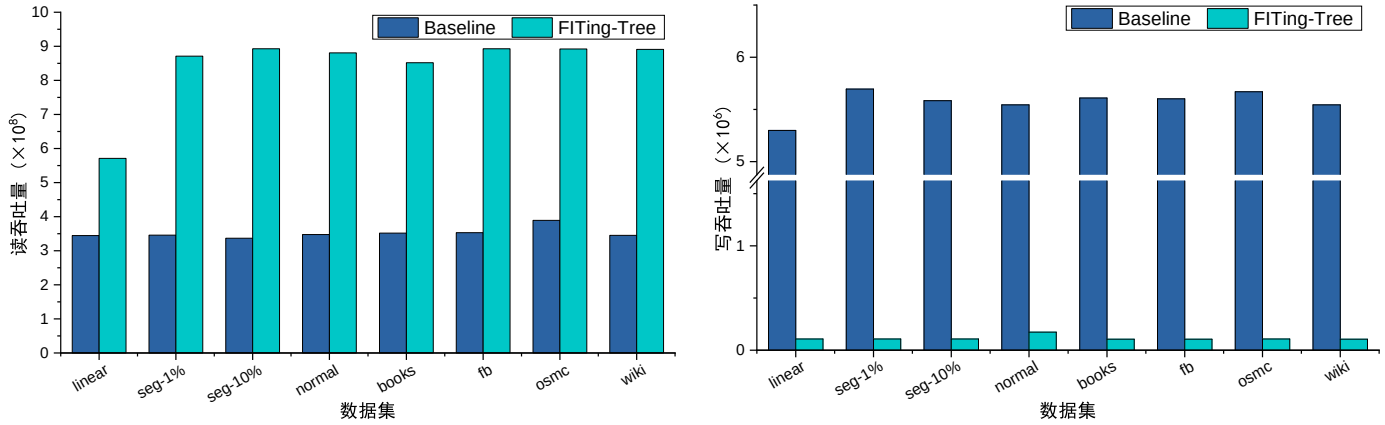
((a)) 平均查询时延　　((b)) 平均插入时延

图 3: 索引的平均读写时延



((a)) p99 查询时延　　((b)) p99 插入时延

图 4: 索引的 p99 读写时延



((a)) 读吞吐率　　((b)) 写吞吐率

图 5: 索引的吞吐率

6.3　p99 查询时延

p99 查询时延的实验结果如图 4(a)所示。可以看到，在所有数据集上，FITing-Tree 的 p99 查询时延都比 B+ 树低，相较于 B+ 树，FITing-Tree 的 p99 查询性能快了 $1.3\times-17.2\times$。这说明，FITing-Tree 不仅在平均程度上提升了查询性能，在最坏查询性能方面也有所提升。这种性能上的提升主要来自于（1）树内部导航过程 和（2）底层数据访问过程 的加速，与 6.1节所述原因一致。

6.4　p99 插入时延

p99 插入时延的实验结果如图 4(b)所示。可以看到，除了 normal 数据集，其他数据集上 FITing-Tree 的 p99 插入时延都比 B+ 树高，FITing-Tree 的 p99 插入延迟可能达到 B+ 树的 5 倍。这说明，FITing-Tree 在最坏插入性能方面比 B+ 树差。FITing-Tree 最坏插入性能的下降主要来自缓冲区满而触发分段算法重新执行，相比于简单的插入，重新执行分段算法带来的的时间延迟就显得十分可观。

6.5　读吞吐率

读吞吐率的实验结果如图 5(a)所示。可以看到，在所有数据集上，FITing-Tree 的查询吞吐率都比 B+ 树高，查询吞吐率达到了 B+ 树的 $1.7\times-2.6\times$。事实上，从 6.1节中 FITing-Tree 取得的低时延效果也可以推断：FITing-Tree 的查询吞吐率将达到一个较好的优化效果。查询吞吐率的提升来自于 FITing-Tree 学习化的索引方式，既能够在短时间内定位到目标数据分段，又能在短时间内定位到分段中目标数据的具体位置，因此提高了索引的读吞吐率。

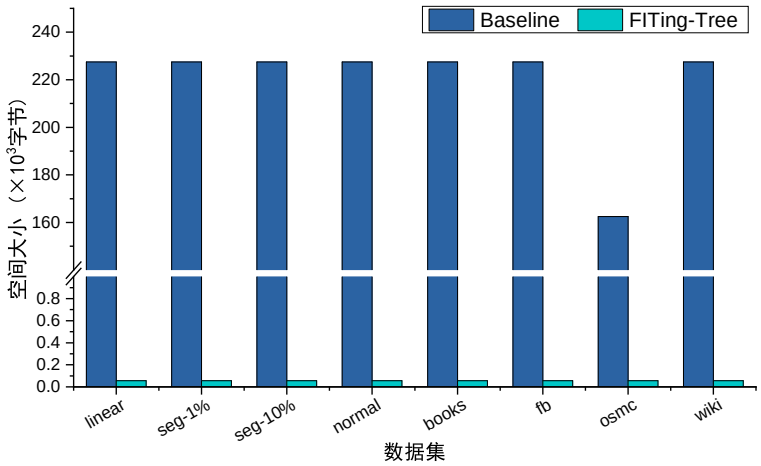


图 6: 索引占用的内存空间

6.6 写吞吐率

写吞吐率的实验结果如图 5(b)所示。可以看到，在所有数据集上，FITing-Tree 的插入吞吐率比 B+ 树低。虽然 FITing-Tree 采用了异地更新的插入算法，以避免插入数据时引起的较多的数据移动。然而，在缓冲区满、分段算法被重新执行时，这个过程可能带来比简单的就地插入更高的时间开销，这个开销将显著影响学习索引 FITing-Tree 的写入性能。事实上，学习索引的主要目标是对索引结构进行读性能上的优化，不支持索引写入和更新[20]。本篇复现的工作是早期为数不多的支持写入的学习索引，我们将提升 FITing-Tree 的写入性能作为一项未来的研究工作。

6.7 索引空间大小

索引空间大小实验结果如图 6所示。可以看到，在所有数据集上，FITing-Tree 对内存的占用都比 B+ 树少很多，普遍能以 10^3 这种数量级来降低内存存储空间。FITing-Tree 采用数据分段的方式，使用简单、轻量的线性参数来描述底层数据的分布，降低了叶子结点的存储空间；且 FITing-Tree 内部节点存放的卫星数据都是每一线性分段的代表数据，进一步降低了树内部结点的存储大小。因此，相比于 B+ 树，FITing-Tree 极大地降低了存储开销。

7 总结与展望

本报告针对一种数据感知型的索引——FITing-Tree 进行了研究和复现。FITing-Tree 是一种学习型索引，旨在降低内存数据库中索引占用的空间。通过对误差参数的调优和代价模型的辅助决策，FITing-Tree 可以协调查询性能和存储空间这两个性能指标。在多个人工数据集和真实数据集上的评估表明：与传统 B+ 树索引相比，在查询性能相差不多的情况下，FITing-Tree 数量级地降低了存储空间的占用。

目前的 FITing-Tree 只在叶子节点对底层数据分布进行拟合，其实，FITing-Tree 的内部节点也有其数据分布规律，也可以使用模型来刻画，很有希望可以进一步降低 FITing-Tree 的存储空间。此外，当前版本 FITing-Tree 的实现，用户需要对节点扇出系数、误差界限、缓冲区大小等参数进行单独的调优。如何在不同数据集或工作负载上实现这些参数的协同自动调优，协调提升索引的读写性能，是未来的一个研究方向。最后，当前版本 FITing-Tree 的实现针对的是内存数据库，如何将这种数据感知的学习型索引推广到磁盘数据库，也是未来值得探讨的课题。

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]//SIGMOD ’16. 2016: 1567-1581.

[2] GRAEFE G, LARSON P A. B-tree indexes and CPU caches[C]//ICDE ’01. 2001: 349-358.

[3] GOLDSTEIN J, RAMAKRISHNAN R, SHAFT U. Compressing relations and indexes[C]//ICDE ’98. 1998: 370-379.

[4] NEUMANN T, WEIKUM G. RDF-3X: A RISC-Style Engine for RDF[J]. PVLDB 2008, 1(1): 647-659.

[5] WU K, OTOO E J, SHOSHANI A. Optimizing Bitmap Indices with Efficient Compression[J]. ACM Trans. Database Syst., 2006, 31(1): 1-38.

[6] PINAR A, TAO T, FERHATOSMANOGLU H. Compressing bitmap indices by data reorganization[C]//ICDE ’05. 2005: 310-321.

[7] STABNO M, WREMBEL R. RLH: Bitmap Compression Technique Based on Run-Length and Huffman Encoding[C]//DOLAP ’07. 2007: 41-48.

[8] CHAN C Y, IOANNIDIS Y E. Bitmap Index Design and Evaluation[C]//SIGMOD ’98. 1998: 355-366.

[9] ATHANASSOULIS M, YAN Z, IDREOS S. UpBit: Scalable In-Memory Updatable Bitmap Indexing [C]//SIGMOD ’16. 2016: 1319-1332.

[10] KIMURA H, HUO G, RASIN A, et al. Correlation Maps: A Compressed Access Method for Exploiting Soft Functional Dependencies[J]. PVLDB 2009, 2(1): 1222-1233.

[11] KIM C, CHHUGANI J, SATISH N, et al. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs[C]//SIGMOD ’10. 2010: 339-350.

[12] LEIS V, KEMPER A, NEUMANN T. The adaptive radix tree: ARTful indexing for main-memory databases[C]//ICDE ’13. 2013: 38-49.

[13] O’NEIL P, CHENG E, GAWLICK D, et al. The log-structured merge-tree (LSM-tree)[J]. Acta Informatica, 1996, 33(4): 351-385.

[14] DAYAN N, ATHANASSOULIS M, IDREOS S. Monkey: Optimal Navigable Key-Value Store[C]//SIGMOD ’17. 2017: 79-94.

[15] CROTTY A, GALAKATOS A, ZGRAGGEN E, et al. The Case for Interactive Data Exploration Accelerators (IDEAs)[C]//HILDA ’16. 2016: 1-6.

[16] GALAKATOS A, CROTTY A, ZGRAGGEN E, et al. Revisiting Reuse for Approximate Query Processing[J]. PVLDB 2017, 10(10): 1142-1153.

[17] IDREOS S, KERSTEN M L, MANEGOLD S, et al. Database Cracking.[C]//CIDR ’07: vol. 7. 2007: 68-78.

[18] AO N, ZHANG F, WU D, et al. Efficient Parallel Lists Intersection and Index Compression Algorithms Using Graphics Processing Units[J]. PVLDB 2011, 4(8): 470-481.

[19] SHATKAY H, ZDONIK S. Approximate queries and representations for large data sequences[C]// ICDE ’96. 1996: 536-545.

[20] KRASKA T, BEUTEL A, CHI E H, et al. The Case for Learned Index Structures[C]//SIGMOD ’18. 2018: 489-504.