软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2020,31(3):663−679 [doi: 10.13328/j.cnki.jos.005902]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

面向数据特征的内存跳表优化技术*

李 梁[1], 吴 刚[1,3], 王国仁[2]

[1](东北大学 计算机科学与工程学院,辽宁 沈阳 110169)

[2](北京理工大学 计算机学院,北京 100081)

[3](南京大学 计算机软件新技术国家重点实验室,江苏 南京 210093)

通讯作者: 王国仁, E-mail: wanggr@bit.edu.cn

摘 要: 跳表作为数据库中被广泛采用的索引技术,优点在于可以达到类似折半查找的复杂度 $O(\log(n))$.但是标准跳表算法中,结点的层数是通过随机算法生成的,这就导致跳表的性能是不稳定的.在极端情况下,查找复杂度会退化到 $O(n)$.这是因为经典跳表结构没有结合数据的特征.一个稳定的跳表结构应该充分考虑数据的分布特征去决定结点层数.基于核密度估计的方式估计数据累积分布函数,预测数据在跳表中的位置,进而设计用于判定结点层数的跳表算法.另外,跳表的查找过程中,结点层数越大的结点被访问的概率越高.针对历史数据的访问频次,设计一种保证频繁访问的"热"数据尽可能地在跳表的上层,而访问较少的"冷"数据在跳表的下层的跳表算法.最后,基于合成数据和真实数据对标准跳表和 5 种改进的跳表算法进行了全面的实验评估并开源代码.实验结果表明,优化的跳表最高可以获取 60%的性能提升.这为未来的科研工作者和系统开发人员指出了一个很好的方向.

关键词: 内存索引;跳表;机器学习;密度估计

中图法分类号: TP18

中文引用格式: 李梁,吴刚,王国仁.面向数据特征的内存跳表优化技术.软件学报,2020,31(3):663−679. http://www.jos.org.cn/1000-9825/5902.htm

英文引用格式: Li L, Wu G, Wang GR. In-memory skiplist optimization technologies based on data feature. Ruan Jian Xue Bao/Journal of Software, 2020,31(3):663−679 (in Chinese). http://www.jos.org.cn/1000-9825/5902.htm

In-Memory Skiplist Optimization Technologies Based on Data Feature

LI Liang[1], WU Gang[1,3], WANG Guo-Ren[2]

[1](School of Computer Science and Engineering, Northeastern University, Shenyang 110169, China)

[2](School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China)

[3](State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China)

Abstract: Skiplist is a widely used indexing technology in the database systems. The advantage is that the complexity of skiplist is $O(\log(n))$. However, in the standard skiplist algorithm, the level of each nodes is generated by a random generator, thus, the performance of the skiplist is unstable. In extreme case, the searching complexity deceases to $O(n)$ which is similar to the list searching time. This is

* 基金项目: 国家自然科学基金(61872072, U1811262, 61572119, 61622202, 61672145, 61732003, 61572121, 61332006, 61332014, 61328202, 61702087); 中央高校基本科研业务费专项资金(N181605012, N171604007, N171904007); 中国博士后科学基金(2018M631358)

Foundation item: National Natural Science Foundation of China (61872072, U1811262, 61572119, 61622202, 61672145, 61732003, 61572121, 61332006, 61332014, 61328202, 61702087); Fundamental Research Funds for the Central Universities (N181605012, N171604007, N171904007); China Postdoctoral Science General Program Foundation (2018M631358)

本文由人工智能赋能的数据管理、分析与系统专刊特约编辑李战怀教授、于戈教授和杨晓春教授推荐.

收稿时间: 2019-07-19; 修改时间: 2019-09-10, 2019-11-25; 采用时间: 2019-12-18

CNKI 网络优先出版: 2020-01-10 13:34:37, http://kns.cnki.net/kcms/detail/11.2560.TP.20200110.1334.005.html

because the classic skiplist do not combine data features to generate its structure. It is believed that a stable skiplist structure should fully consider the distribution characteristics of the data to determine the number of node levels. This study estimates the data cumulative distribution function based on the kernel density estimation method, and predicts the position of the data in the skiplist, determines the number of node levels. In addition, it is found that the node with a higher level has a higher probability of being accessed. This study also focuses on the access frequency and the hot data of frequent access, make sure that the upper level of the skiplist is hot data, and access the less cold data in the lower level of skiplist. Finally, a comprehensive experimental evaluation of the six kinds of skiplist algorithms is performed based on the synthesis dataset and real dataset, besides, the source code is open. The results show that the best skiplist algorithm can achieve a 60% performance improvement, which points out a authentic direction for the future researchers and system developers.

Key words: in-memory index; skiplist; machine learning; density estimation

近年来,伴随着移动互联网和大数据的热潮,行业对数据处理能力,尤其是在线联机事务处理能力(OLTP)提出了更高的要求.一个比较典型的案例是在 2019 年的“双十一”,阿里巴巴当天创下了 6100 万次/s 处理峰值的纪录(https://developer.aliyun.com/article/727517).如此巨大的数据处理能力,与阿里自研数据库 OceanBase[1]息息相关的.在摩尔定律的作用下,计算机硬件(多核、大内存、固态硬盘等)的性能一直在快速发展,这很大程度上促进了数据库得以支撑类似“双十一”这种高吞吐率的应用.典型地,计算机处理器的性能从单方面提高频率转向增加核心数的方向转变.为了适配新硬件的特性及最大限度地发挥硬件的性能,数据库领域很多模块都要针对性地进行适配.索引技术作为数据库存储层的核心,同样需要提出适配.跳表(Skiplist[2])是数据库领域的一个重要索引技术,它具有结构简单、易于实现、无锁并发、$O(\log(n))$的查找复杂度等优点,因此在 LevelDB,MemSQL,Redis 等数据库中都有广泛应用.

跳表是基于线性链表改进而来的.传统的链表中,每个结点只包含一个结点指针域,而跳表的每个基础结点包含多个指针域.图 1 是理想情况下的跳表示意图,水平方向来看,每一层都是单向链表,而且越往上的层级,结点越稀疏,跳表通过维护每个结点的层数,保证每层结点个数都比下一层大致减半.跳表的查找操作是从最上层结点依次往下缩小查找区间,过程类似于折半查找.对于跳表的插入操作,先根据待插入 *key* 查找到待插入的位置,然后算法通过随机算法确定结点的层数,再插入结点.假设结点 *key* 的层数为 3,则维护下三层链表的指针.第 1.1 节将对跳表做详细介绍.

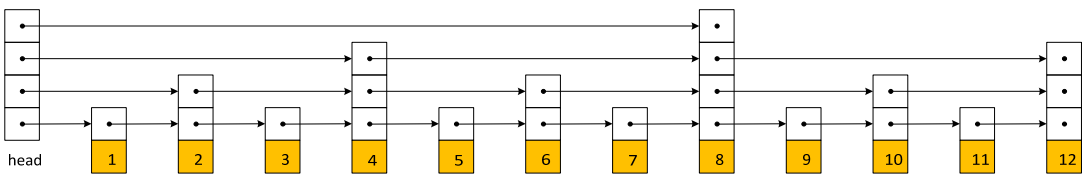


Fig.1 Example of skiplist in ideal case

图 1 理想情况的跳表示例

但我们认为,跳表不是一个稳定的数据结构.计算机科学中,经典的不稳定的数据结构是二叉查找树.二叉树的特性只需要保证左小右大的特质,一旦数据从小到大依次插入二叉树,二叉树便退化为线性表,此时的查找复杂度也从 $O(\log(n))$退化到 $O(n)$.因此提出了平衡二叉树,通过左旋和右旋操作,改进了二叉树的稳定性.

跳表结构的查找时间是通过多层链表指针域达到加速的效果.但跳表的层数随机算法导致它不是一个稳定的结构,具体来说,包含以下两点挑战.

- 数据偏斜.跳表的优点在于可以达到类似折半查找的复杂度 $O(\log(n))$,但是跳表的层数是通过随机算法生成的,这会导致性能的不稳定.在极端情况下,会生成类似于图 2 的跳表结构,此时的查找复杂度等价于链表的查找复杂度(都是 $O(n)$),跳表的优点便不复存在了.这主要是因为跳表的随机生成层数的算法并没有结合数据特征去生成跳表结构.我们认为,一个稳定的跳表结构应该充分考虑数据的分布特征,在插入 *key* 的同时,根据分布特征去决定层数,而不是随机生成;

- 热度数据.我们发现,跳表的查找过程中,结点层数越大的结点被访问的概率越高.比如图 1 中,查找结点 8,直接在第 1 层便找到了,便不需要继续一直下沉操作了.这促使我们可以充分考虑热度数据的特征,让热度数据的层数尽可能高,访问效率便更快一些.

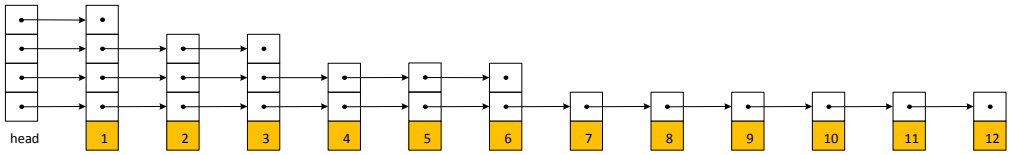


Fig.2 Example of skiplist in worst case

图 2 最坏情况的跳表示例

本文的贡献如下:

- 基于数据分布估计的跳表结构.本文基于核密度估计的方式去估计数据集合的累积分布函数,进一步预测数据在跳表中的位置和结点层数,从而达到跳表查找性能的优化目的;
- 基于冷热数据分层的跳表结构.本文针对历史数据的访问频次信息,设计了一种冷热分层的跳表算法,对频繁访问的热度数据保证尽可能的在跳表的上层,而访问较少的冷数据在跳表的下层,从而保证热度数据访问加速的效果;
- 跳表选型.本文基于合成数据和真实数据对标准跳表和 5 种改进的跳表算法进行了全面的实验评估并开源代码.实验结果表明,优化的跳表最高可以获取 60%的性能提升.

本文第 1 节对跳表和密度估计进行全面的介绍.第 2 节对相关工作进行总结.第 3 节根据数据分布特征进行跳表算法的优化设计.第 4 节根据热度数据特征对跳表进行优化算法设计.第 5 节通过合成和真实数据集对上述算法进行实验验证.第 6 节对全文进行总结,并提出对未来工作的设想.

1 预备知识

本节将完整介绍跳表的概念和基本操作,然后讨论标准跳表算法所遇到的问题.

1.1 跳表

跳表从结构上可以看成多层链表结构.它的结点结构和链表的结点结构很相似,不同的是,链表结点除了数据域之外只包含一个 next 指针域,而跳表结点包含多个指针域.每个结点所包含的指针个数(层数)是可变的.跳表结点一旦被创建,指针个数便确定.多个指针域按照“层”的方式,维护多个单向链表.通过随机函数(几何分布)控制每个结点的指针个数(层数),从而保证每一层的“链表”个数依次减半.因此,跳表是一种随机性算法.

算法 1 阐述了标准跳表中的随机生成层数的算法.原理相当于做一次抛硬币的(伯努利实验)实验,统计首次抛出反面的次数.如果遇到正面继续抛,遇到反面则停止,用实验中抛硬币的次数 K 作为结点的层数.显然,随机变量 K 服从参数 p=1/2 的几何分布.K 的期望值 $E[K]=1/p=2$.也就是说,跳表中元素的平均层数为 2 层,包含 N 个元素的跳表的指针域个数大致为 $2n$ 个.

算法 1. 基于伯努利实验的跳表层数决策算法.

输入:跳表最大层数 max*level*;

输出:待插入 *key* 的层数.

```
1:  Function random_level(maxlevel)
2:  level=1;
3:    While (random(0,1)<0.5)
4:      level++;
5:  Return min(level,maxlevel);
```

数据库索引一般需要提供 3 种访问操作:查找、插入、删除.跳表的查找操作是从最上层开始查找,依次按照范围往下查找,直到最下面一层为止.如图 3 所示,假设我们要查找结点 7.首先,我们从最上第 4 层开始找到结点 1,然后下沉到第 3 层,依次查找到结点 6;再下沉到第 2 层,查找到结点 11,发现 7 小于 11,回退到结点 6;然后再下沉到第 1 层,继续查找结点 7.如果在第 1 层还没有找到,则返回不存在该结点.数据查找的时间复杂度为 $O(\log(n))$,其中,n 为跳表中数据元素的个数.

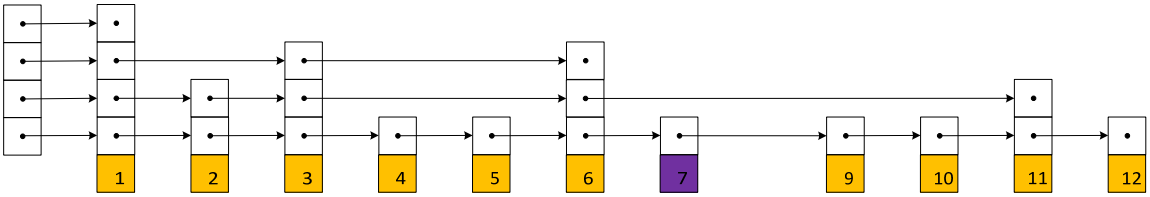


Fig.3 Example of searching node 7

图 3 查找结点 7 的示例

跳表的插入操作主要包含 3 个步骤:首先,通过类似查找的算法找到待插入的元素,并记录出查找过程中的下沉结点;然后,采用上面的随机算法决定层数,分配结点;最后,在每一层上修改插入结点的后继指针和下沉结点的后继指针.比如图 4,假设我们要插入结点 8,先通过查找操作,记录图中的结点,确定结点 8 的待插入位置;第 2 步,假设根据算法 1 生成结点 8 的层数为 2;最后,把图中指针域和结点 8 的指针域修改正确.数据插入的复杂度为 $O(\log(n))$.跳表的删除操作是类似于插入操作的逆操作,先查找到结点,然后修改指针域.图 5 展示了删除结点 2 的过程.数据删除的复杂度也为 $O(\log(n))$.

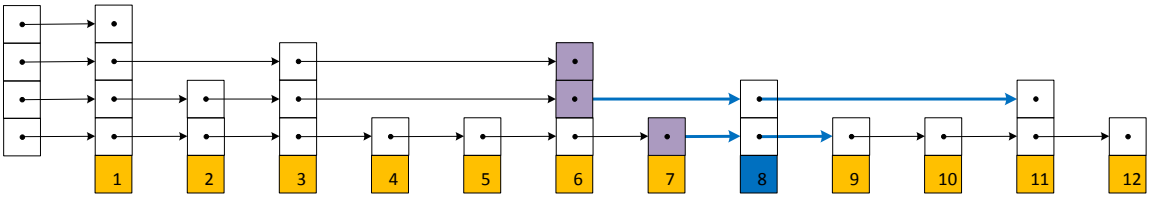


Fig.4 Example of insert node 8 in skiplist

图 4 插入结点 8 的示例

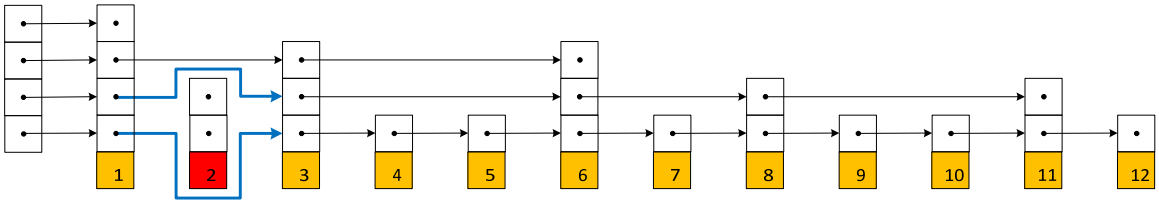


Fig.5 Example of delete node 2 in skiplist

图 5 删除结点 2 的示例

1.2 数据分布估计

在数据库领域,系统在运行过程中通常需要基于历史运行数据分析出数据的分布特征,这有利于加速或优化数据处理.比如,数据库管理系统的查询优化器模块需要采用直方图的方式进行统计数据分布情况,这部分数据通常被记录在数据字典内.直方图一般分为等宽直方图和等高直方图,但直方图的数据还不足够细致.

在人工智能领域,可以借助机器学习的手段进行密度函数估计(PDF)和累计分布函数(CDF)的估计.对于密度估计的手段,可以分为两类:一类是参数估计,需要假设数据服从特定的分布,然后估计分布的参数;另一类是

无参数估计(非监督学习),包含直方图估计、核函数估计、k 最近邻估计和神经网络估计等.下文我们简单介绍一种实用的技术:核函数估计.

假定随机变量 $x\in R$,概率密度函数为 $f(x)$.需要在给定的一组 x 的随机样本 $x_1,x_2,x_3,\ldots,x_n\in R$ 的基础上,计算 $f(x)$的一个估计 $\hat{f}(x)$, $\hat{f}(x)$ 需要充分接近真实的 $f(x)$.核函数估计基于所有的样本信息计算近似密度函数:

$$\hat{f}_h(x)=\frac{1}{nh}\sum_{i=1}^{N}\phi\left(\frac{x-x_i}{h}\right) \tag{1}$$

其中,n 表示样本的数量的个数,h 为带宽,$\phi(x)$被称为核函数.核函数需要满足下述 4 个性质.

(1) 对称性:$\phi(u)=\phi(-u)$;

(2) 标准化:$\int_{-\infty}^{\infty}f(x)\mathrm{d}x=1$;

(3) 递减性:$\phi'(x)$当 $u>0$;

(4) 期望为 0:$E(\phi)=0$.

常见的核函数包括高斯、余弦、均匀、三角形等.对于累计分布函数的估计,可以对公式(1)求积分得到.

2 相关工作

下面分 3 个方向介绍和本文直接相关的工作.

- 索引技术.

在过去的几十年中,已经提出了各种不同的索引结构[3],例如基于磁盘的 B+树[4]和面向内存系统的 T 树[5]以及平衡/红黑树[6,7].由于原始内存索引结构具有较差的缓存命中率,因此提出了针对缓存优化的 B+树变体,例如 CSB+树[8],它能够通过使用偏移量而不是结点之间的指针来定位数据.此外,最新的工作还有采用 SIMD 指令(比如 FAST[9])或 GPU[9−11]优化数据库索引的技术.对于字符串的索引结构也有大量的研究,例如字典树/前缀树[12−15].此外,还有针对索引存储空间进行优化设计的索引结构,如 wavelet 树[16,17]等.跳表[2]具有与树型索引大致相同的平均搜索复杂度,但跳表的实现难度小很多.即便是 lock-free 的跳表实现,通过使用 CAS 指令可以很容易地实现跳表[18],而这对于 B+树来说是非常困难的.Abraham 等人[19]结合跳表和 B 树来进行有效的查询处理.

- 密度估计.

对于数据分布的估计,最简单直接的方式是通过直方图统计(https://en.wikipedia.org/wiki/Histogram),直方图虽然简单,但被广泛应用在数据库查询优化系统中[20].核密度估计法[21]可以用来估计未知的密度函数,属于非参数检验方法之一,由 Rosenblatt 和 Parzen 提出,又叫 Parzen 窗(Parzen window).基本原理和直方图有些类似,是一种平滑的无参数密度估计法.在机器学习社区中对密度估计也进行了大量研究[22−25].还有基于深度学习的 DeepNADE 密度估计[26].密度估计可以应用于排名[27].如何最有效地模拟 CDF,仍然是一个值得进一步研究的问题.

- 智能数据库技术.

对于借助机器学习的方法优化数据处理的技术由来已久,我们主要列举 3 个类别的代表性工作:

(1) 比较有代表性的工作是 SIGMOD 18 上,Dean 等人[28]开创性的手段借助机器学习的技术优化 B+树的存储空间和 hash 索引的冲突问题,提出了 learned index 的概念.本文继承同样的思想,结合跳表的数据结构对跳表进行优化.自 Google 之后,2019 年出现了大量的关于 learned index 的研究[29].Galakatos 等人提出了 FITing-Tree[29],可以在查询精度和存储空间之间达到平衡.Ding 等人[30]提出了 ALEX 索引,进一步对 learned index 在动态数据集上进行优化.Wu[31]针对二级索引设计了 Succinct 类索引;

(2) Pavlo[32]和 Chaudhuri[33]提出了人工智能的方法还可以用于优化数据库运维的难度,降低 DBA 的难度.代表性工作主要是卡耐基梅隆大学提出的 OtterTune[34]和阿里巴巴达摩院的 iBTune[35], iBTune 被大规模应用在阿里巴巴线上系统中,主要用于数据库缓存大小的自动设置;

(3) 另外,在数据挖掘领域存在大量采用机器学习的手段学习位置敏感的哈希函数用于构建近似最近邻索引的工作,例如文献[36,37].

3 基于数据分布的跳表

本节首创性地采用数据分布特征去优化跳表结构,接下来我们将介绍 3 个优化算法.

3.1 *Cdf-list*

跳表是一种有序结构,对于给定的 *key* 在跳表中的排序位置 *location* 和数据的分布特征有很直接的关联.已知数据累计分布函数的情况下,对于包含 *N* 个元素的数据集,元素 *key* 和 *key* 的位置 *location* 满足公式(2):

$$CDF(key)=\frac{location}{N} \tag{2}$$

当插入 *key* 的时候,我们通过核密度估计的方式估计出数据的累计分布函数 $\widehat{CDF}(key)$,公式(3)可以预判 *key* 所在的位置 $\widehat{location}$ 估计:

$$\widehat{location}=\left\lceil \widehat{CDF}(key)\times N \right\rceil \tag{3}$$

一旦可以估计 *key* 的位置,可以通过位置信息决定层数,而不只是通过简单的 *random* 方式.

算法 2 陈述了基于 *CDF* 信息计算 *key* 层数的 *cdf-list* 算法.

算法 2. *Cdf-list* 的层数决策算法.

输入:数据集个数 *N*,数据累计分布函数 *CDF*,待插入的 *key*;

输出:待插入 *key* 的层数.

```
1:  Function get_level(key)
2:  location=CDF(key)×N;
3:  level=1;
4:  step=2;
5:  While (step<=location)
6:     If (location%step==0)
7:        level++;
8:     End If
9:     step=step<<1;
10: Return level;
```

与算法 1 中 *random* 的方式相比,算法 2 充分利用了数据的分布特征.算法 2 的输入为数据集个数 *N*、数据集的累积分布函数 *CDF*、待插入的 *key*.输出为待插入 *key* 的层数 *level*.首先,通过公式(3)计算出待插入 *key* 在跳表中从小到大的排序位置 *location*(第 2 行),注意:如果 *location* 的结果为小数,需要向上取整.为了创建出类似于图 1 的跳表结构,处在不同 *location* 的 *key* 的高度是可以被计算的(第 3 行~第 10 行).理想情况下,跳表中高度大于等于 2 的 *key* 对应的位置应该是 2^1 的倍数,高度大于等于 3 的 *key* 对应的位置应该是 2^2 的倍数,依此类推,高度大于等于 *k*+1 的 *key* 对应的高度应该是 2^k 的倍数.根据位置判定层数,实际上是通过判定 *location* 的二进制数字的末尾有几个零:如果有 *k* 个零,那么高度则为 *k*+1.例如,处在第 8(二进制 1000)个位置的数据高度应该是 4 层.下面用一个例子对跳表创建过程进行说明.

例 1:为便于理解,这里我们假设数据服从均匀分布,数据集是包含 12 个元素的集合(真实数据集会很大).数据集为{5,4,3,2,1,6,7,8,9,10,12,11}.初始时,*cdf-list* 为空,插入元素 5 的时候,虽然当前并没有插入 1,2,3,4 这 4 个结点,但可以预判元素 5 的位置.因为元素 5 的 *cdf* 值为 5/12,元素 5 应该所处的位置为 5(二进制 101),对应的跳表高度为 1.依此类推,可以创建出接近于图 1 所示的跳表结构.

3.2 *Bound-list*

公式(3)对于 *location* 的估计会存在误差,因为累计分布函数的估计存在偏差.这可能会导致多个连续的 *key* 判定到同一个 *location*,从而多个连续的 *key* 的高度会相同.如图 6 所示:比如说数据中元素 7,8,9 的 *location* 估

计的都是 8,这就导致判定的高度都为 4,这样的结构在查询过程中是不合理的.

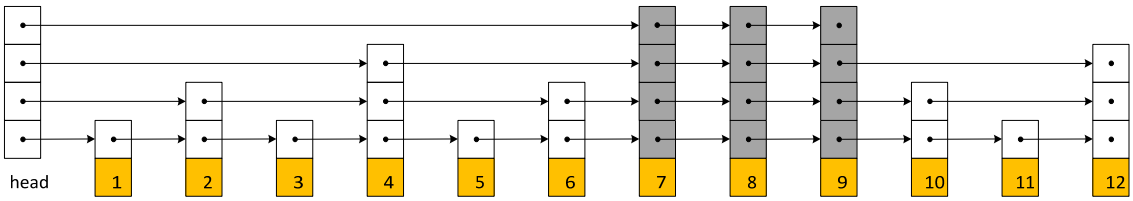


Fig.6 Unsuited structure caused by *CDF* error

图 6 *CDF* 误差导致的不合理结构

为了容忍位置估计的误差,我们提出了容忍偏差的算法.首先,我们采用一个长度为 *N* 的数组,该数组预先记录理想情况下每个位置的元素的高度.算法 3 给出了数组的创建过程.算法的输入是数据集的大小 *N*,输出为 *location* 和层数的映射数组 *level_array*.对于包含 *N* 个元素的数据集,首先分配一个 *N*+1 大小的数组(第 2 行),并初始化为 0(第 3 行).注意,*level_array*[0]代表的是 head 头结点.初始设置步长为 1(第 4 行),将 *level_array* 中每个元素累计加 1(第 6 行~第 8 行).然后步长 *step* 变为 2,偶数位置的 *level_array* 数值加 1.依此类推,*step* 每次乘以 2(第 9 行),直到 *step*<*N* 为止(第 5 行).

算法 3. 构造 *bound-list* 的层数数组.

输入:数据集个数 *N*;

输出:*location* 与层数的映射数组 *level_array*.

```
1:  Function build_array(N)
2:  level_array=new int[N+1]
3:  set_zero(level_array);
4:  step=1;
5:  While (step<=N)
6:      For (i=0; i<=N; i=i+step)
7:          level_array[i]++;
8:      EndFor
9:      step=step<<1;
10: Return level_array;
```

例如包含 12 个元素的数据集,构建的 *level_array* 数组为{4,1,2,1,3,1,2,1,4,1,2,1,3}.我们可以看到,*level_array* 数组和图 1 对应的跳表每个结点的高度一一对应.其中,*level_array*[0]=4 代表的是 head 头结点.然后引入额外的 *bound*,每次高度选取时,我们结合判定位置前后的 *bound* 区间内选取最大的 *level* 作为最终值.算法 4 给出了具体的细节.

算法 4. 容忍 *CDF* 估计误差的层数决策算法.

输入:数据集个数 *N*,数据累计分布函数 *CDF*,待插入的 *key*,误差限 *bound*,层数数组 *level_array*;

输出:待插入 *key* 的层数.

```
1:  Function get_level(key)
2:  location=CDF(key)×N;
3:  lower_bound=max(location-bound,1);
4:  upper_bound=min (location+bound, N);
5:  level=0;
6:  For (i=lower_bound; i<=upper_bound; i++)
7:      If (level_array[i]>=level)
```

8: *level*=*level_array*[*i*];
9: *level_array*[*i*]=1;
10: **End If**
11: **EndFor**
12: **Return** *level*;

算法 4 与算法 2 类似,都是根据 *CDF* 的值决定待插入 *key* 的层数.不同的是,算法 4 结合 *bound* 区间去判定层数,容忍了估计导致的误差.算法 4 的输入是数据集个数 *N*、数据估计出的累计分布函数 *cdf*、待插入的 *key* 和误差限 *bound* 以及算法 3 生成的数组 *level_array*.算法的输出是待插入 *key* 的层数.首先,根据估计的 *CDF* 值估计出 *key* 在跳表中的位置(第 2 行).*location* 的位置并不是精确的,第 3 行、第 4 行计算一个 *location* 的区间[*lower_bound*,*upper_bound*].然后,通过在 *level_array* 数组的区间[*lower_bound*,*upper_bound*]中选取最大值作为待插入 *key* 的 *level*,并把这个最大值所在数组位置的值设置为 1,从而保证在插入附近连续的 *key* 时,*key* 的高度不一样.

例 2:类似于例 1 的均匀数据集为{5,4,3,2,1,6,7,8,9,10,12,11}.初始时,*bound-list* 为空,通过算法 3 构建出 *level_array*.构建的 *level_array* 数组为{4,1,2,1,3,1,2,1,4,1,2,1,3}.插入元素 5 的时候,预判元素 5 的 *cdf* 值为 5/12,元素 5 应该所处的位置为 5.假设 *bound* 为 1,则我们从 *level_array*[5−1]到 *level_array*[5+1]之间选取最大值 3,即 *get_level*(5)=3.插入 5 之后,需要修改层数数组为{4,1,2,1,**1**,1,2,1,4,1,2,1,3}.接着插入元素 4,预判 *location* 为 4,结合 *bound*,对应层数为 1.依此类推,插入跳表中所有结点.

3.3 *Partition-list*

Cdf-list 和 *Bound-list* 需要提前知道数据集的大小,对于数据集大小未知的情况下,我们提出了 *partition-list*. *Partition-list* 把数据集 2^p-1 等分.可以通过下述公式判定 *key* 属于哪个分区:

$$partition=\lceil CDF(key)\times(2^p-1)\rceil \tag{4}$$

类似 *bound-list* 的方式,我们通过调用算法 3 的函数 *build_array*(2^p-1)构建长度为 2^p 的 *partition_array* 数组. *Partition-list* 的伪代码如算法 5.

算法 5. *Partition-list* 的层数决策算法.

输入:数据累计分布函数 *CDF*,待插入的 *key*,参数 *p*,层数数组 *partition_array*;

输出:待插入 *key* 的层数.

1: **Function** *get_level*(*key*)
2: *partition*=*CDF*(*key*)×(2^p-1);
3: **If** (*level_array*[*partition*]!=0)
4: *level*=*level_array*[*partition*]+max*level*−*p*;
5: *level_array*[*partition*]=0;
6: **Else**
7: *level*=*random_level*(max*level*−*p*);
8: **End If**
9: **Return** *level*;

算法 5 首先将 *partition-list* 划分成 2^p-1 个 *partition*,插入 *key* 时,第 1 个落入某个 *parition* 的 *key*1 的层数采用 *level_array* 去决策(第 4 行).若新插入的 *key*2 落入 *key*1 所属的同样的 *partition*,则需要算法 1 类似的随机方法计算层数(第 7 行).算法能够保证最上面的 *p* 层,在每一个 *partition* 里都有唯一的 *key*,并且最上面 *p* 层的 *key* 的高度和分布基本类似于图 1 的理想状况.

例 3:数据集为{6,7,8,9,10,5,4,3,2,1,14,13,12,11}.初始时,*partition-list* 为空,最大高度为 6.假设 *p*=3,我们把数据集分成 2^3-1 个分区.当 *key*=6 时,*partition*=3,对应的高度 *get_level*(5)=1+(6−3)=4.当 *key*=5 时,此时 *partition*=3,*partition*3 内已经有一个高度大于 3 的 *key*,所以 *key*=5 对应的高度应该通过 *random_level*(3)生成一个高度不大

于 3 的层数,如图 7 所示.

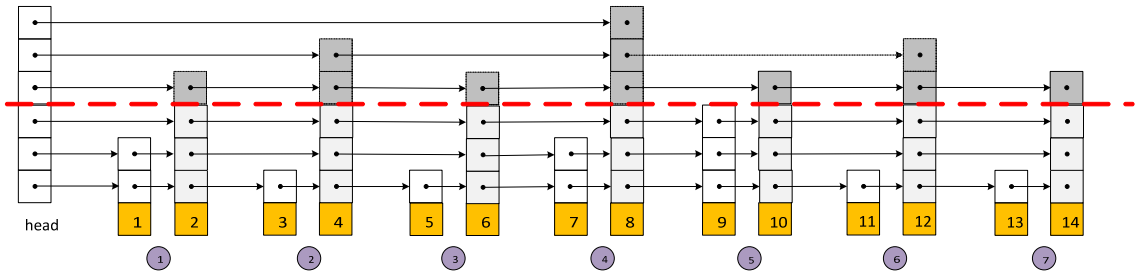


Fig.7 Example of partition-based skiplist

图 7 基于分区的跳表示例

4 结合访问热度的跳表

4.1 *Hot-list*

数据库 *key* 的查找过程中,我们是需要从上层结点依次向下层结点搜索查找的过程.引言部分指出,结点层数高的结点,查找的效率相对比结点低的结点要高.基于此,我们设计了一套针对热度数据优化的跳表 *hot-list*.

我们根据历史数据的访问频率,把数据集中访问频率较高的 2^h-1 个数据判定为热度数据.我们通过算法 6,想构建一个最上面 *h* 层都是热数据,max*level*–*h* 层及以下层级的都是冷数据.

算法 6. *Hot-list* 的层数决策算法.

输入:待插入的 *key*,参数 *h*;

输出:待插入 *key* 的层数.

1: **Function** *get_level*(*key*)

2: **If** (*key* is hot *key*)

3: *level*=*random_level*(*h*)+max*level*–*h*;

4: **Else**

5: *level*=*random_level*(max*level*–*h*);

6: **End If**

7: **Return** *level*;

算法 6 的输入是待插入的 *key* 和参数 *h*,总体依然是采用 *level* 的 *random* 方案.但我们对 *key* 做了一个分层.热度数据的层数高度都在高层(第 2 行、第 3 行),冷数据都在下层(第 4 行~第 6 行).通过上述代码,仍然可以保持每层从下往上依次减半的规律.并且 *h* 层以下都是冷数据,*h* 层以上都是热度数据.

例 4:类似于例 1 的均匀数据集为{5,4,3,2,1,6,7,8,9,10,12,11}.初始时,*hot-list* 为空,最大高度为 6.假设 1,2,3,4,5 这 5 个 *key* 被频繁访问,也就是属于热度数据.我们设置 *h* 为 3,对于 1,2,3,4,5 这几个 *key*,我们采用第 3 行代码的方式创建层数 *level*,*level* 一定大于 3.对于其他的 *key*,用第 5 行的方式创建 *level*,层数肯定不大于 3.这就保证热度数据层数大于等于 3,冷数据小于等于 3.加速热度数据的访问.

4.2 Mix-*list*

上述算法 *partition-list* 和 *hot-list* 可以结合优化成下述算法 7,同时,结合 *partition* 方案和冷热分层的方案进行设计,原理类似便不展开介绍.原则上,*bound-list* 同样可以和 *hot-list* 进行方案结合,但我们认为,*partition-list* 的适用性比 *bound-list* 要好(*bound-list* 需要知道数据集的大小),因此我们只考虑 *partition-list* 和 *hot-list* 结合的方案.

算法 7. 基于数据分布和热度的层数决策算法.

输入:数据累计分布函数 *CDF*,待插入的 *key*,误差限 *bound*,层数数组 *partition_array*,参数 *h* 和 *p*;

输出:待插入 *key* 的层数.

1: **Function** *get_level*(*key*)
2: *partition*=*CDF*(*key*)×(2^p−1);
3: **If** (*key* is hot *key*)
4: *level*=*random_level*(*h*)+max*level*−*h*;
5: *level_array*[*partition*]=0;
6: **Else If** (*level_array*[*partition*]!=0)
7: *level*=*level_array*[*partition*]+max*level*−*p*;
8: *level_array*[*partition*]=0;
9: **Else**
10: *level*=*random_level*(max*level*−*p*);
11: **End If**
12: **Return** *level*;

4.3 总结对比

表 1 对上述算法进行总结.标准跳表虽然需要的信息少,适用范围广,但却不够稳定.*cdf-list* 和 *bound-list* 可以基于估计的 *cdf* 信息以及数据集大小去判定层数.但是它们需要提前对数据集大小,削减了应用的范围. *partition-list* 只需要估计 *cdf* 函数便可以使用,对参数 *p* 的设置需要进一步评估.*hot-list* 结合数据的冷热信息进行判定层数,参数 *h* 同样会影响性能.mix-*list* 是 *partition-list* 和 *hot-list* 的结合.

Table 1 Summary

表 1 算法对比

算法	层数决策的特点	适用前提	性能参数
SkipList	集合分布,随机生成		
Cdf-list	根据 *cdf* 信息估计高度	*Cdf* 估计和数据集大小	
Bound-list	容忍 *cdf* 误差的估计方法	*Cdf* 估计和数据集大小	
Partition-list	基于数据分区的方式	*Cdf* 估计	*p*
Hot-list	基于冷热数据分层的随机	冷热数据信息	*h*
mix-*list*	结合 *partition-list* 和 *hot-list*	*Cdf* 估计、冷热数据信息	*p*, *h*

5 实验及分析

5.1 硬件环境

本文的实验基于刀片式 HP ProLiant DL380p Gen8 服务器进行性能评估.服务器搭配的处理器为 E5-2620,搭载 15MB 三级缓存,内存为 256GB 的 DDR4.运行系统为 CentOS 7 x86_64(linux 3.10).跳表基于 C++实现,g++ 4.8.5 编译,采用 CLion 2019.1 开发,Googletest 进行单元测试,Spdlog 进行日志打印,实现代码开源在“码云”平台(https://gitee.com/bombel/cdf_skiplist).本节首先对基于 *cdf* 优化的算法(*skiplist*,*cdf-list*,*bound-list*,*partition-list*)进行性能评测,然后针对热度数据优化的算法(*skiplist*,*hot-list*,mix-*list*)进行测评.默认情况下,*partition-list* 中的 *p*=13,*hot-list* 中的 *h*=20.

5.2 测试数据集

不同分布的数据集对 *CDF* 的特征有很明显的影响,本文首先针对不同分布下的合成数据集进行性能评估,包括均匀分布、正态分布、齐夫分布.具体介绍如下:

(1) 均匀分布:我们基于均匀分布随机数生成器生成测试数据集.为了反映数据集大小对不同跳表的性能

影响,分别生成包含 2^{15},2^{18},2^{21},2^{24} 个键值对的数据集,其中,*key* 和 *value* 都是浮点数类型;

(2) 正态分布:采用正态分布生成器生成 5 个不同方差(均值为 10、方差分别为 1,3,5,7,9)、大小都为 2^{21} 的数据集,因为方差反映了数据的稀疏程度,方差越大,数据越稀疏;

(3) 齐夫分布:齐夫分布是一种反映数据偏斜程度的数据集,广泛用在数据库的测试场景.我们分别生成参数为 0.1,0.3,0.5,0.7 的齐夫分布.数据集的大小都为 2^{21};

(4) 真实数据集.我们基于 Amazon 和 Youtube 的真实数据集(http://snap.stanford.edu/data/index.html)对算法性能进行评测.数据集大小分别为 334 863 和 1 134 890,每个数据项代表的是分别是 YouTube 用户 id 和 Amazon 用户的 id.

通过下述实验我们可以发现,数据的稀疏程度对跳表性能影响不大,因此对于亚马逊和 YouTube 的数据分布我们并没有去分析.数据集总结见表 2.

Table 2 Dataset

表 2 数据集

数据集	大小	参数
均匀分布	2^{15}, 2^{18}, $\mathbf{2^{21}}$, 2^{24}	
正态分布	2^{21}	1, 3, 5, 7, 9
齐夫分布	2^{21}	0.1, 0.3, 0.5, 0.7
Amazon	334 863	
Youtube	1 134 890	

5.3 CDF优化实验结果

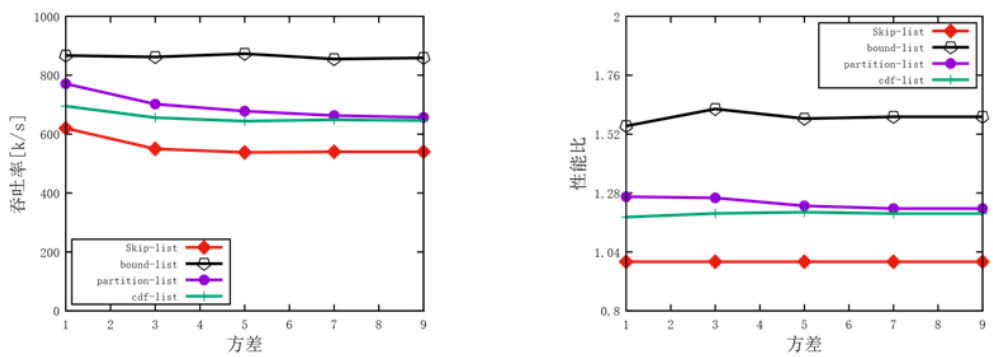
本节评估 CDF 优化算法的效果,主要测试查询吞吐率(QPS)和吞吐率性能比.图 8 和图 9 分别反映的是基于正态分布和齐夫分布数据集下的性能的结果.横轴分别是正态分布的方差和齐夫分布的参数,纵轴分别是查询的吞吐率性能(QPS,单位是 k/s)和 *cdf-list*,*bound-list*,*partition-list* 相对 *skiplist* 的吞吐率性能比.

从实验结果可以发现:

(1) 相同数据量的情况下,不管是正态分布还是齐夫分布,每个算法的性能基本不受分布参数的影响;

(2) 但在方差为 1 和齐夫分布参数为 0.7 时的数据集下,每个算法的性能都略有增加,这是因为实验中的跳表不容许重复 *key* 出现,上述两个参数情况下,会导致过多 *key* 重复,导致跳表结点数量下降,性能有所增加;

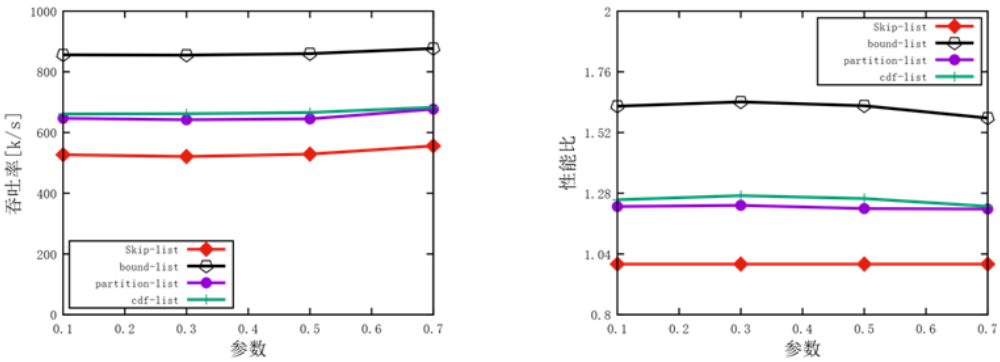
(3) 根据图 8(b),9(b),*bound-list* 可以提高 60%的性能,*cdf-list* 和 *partition-list* 的相比 *skiplist* 性能有大约 25%的提升.



(a) 吞吐率受方差的影响 (b) 性能比受方差的影响

Fig.8 Performance influenced by the parameter of normal distribution's variance

图 8 正态分布不同方差下,吞吐率的性能和性能比



(a) 吞吐率受齐夫参数的影响 (b) 性能比受齐夫参数的影响

Fig.9 Performance influenced by the parameter of zipfian distribution

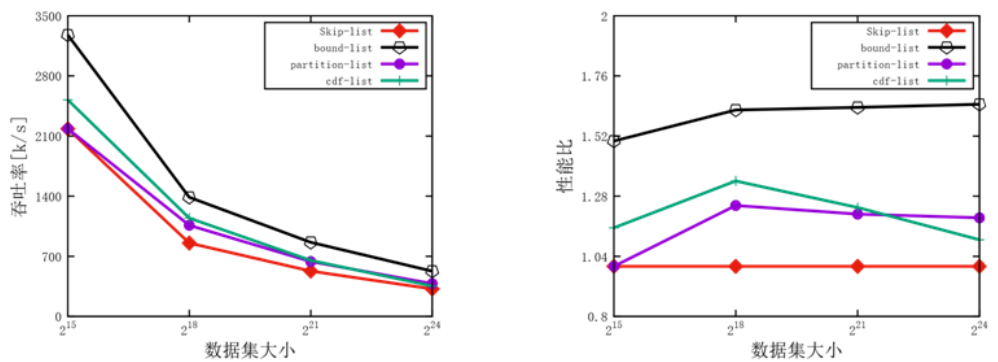
图 9 齐夫分布下不同参数吞吐率的性能和性能比

图 10 是均匀数据集下的实验.图 10(a)反映了数据集大小对不同跳表的查询吞吐率性能的影响,横轴对应的数据集大小,纵轴是查询吞吐率的性能;图 10(b)反映的是 *cdf-list*,*bound-list*,*partition-list* 相对 *skiplist* 的吞吐率性能比.实验结果反映 3 点.

(1) 整体上来看,所有跳表的查询性能都随着数据集大小的增大而下降.这是因为数据集越大,跳表的层数会越高,并且每次要对比的 *key* 的个数越多,导致性能下降.虽然 *skiplist* 和 B+树有区别,但这也是为什么 MySQL,Postgresql 等按 B+树组织表结构的数据库,建议单表的行数不要太大的原因;

(2) 不同算法之间对比.可以看到,性能最好的是 *bound-list*,其次是 *cdf-list* 和 *partition-list*,*skiplist* 最差;

(3) 其中有一个异常的结果,在小数据集下,*partition-list* 的性能很差,相比 *skiplist* 没有性能提升.这是因为参数 p=13 在此时的设置是不合适的,图 11 会详细介绍.



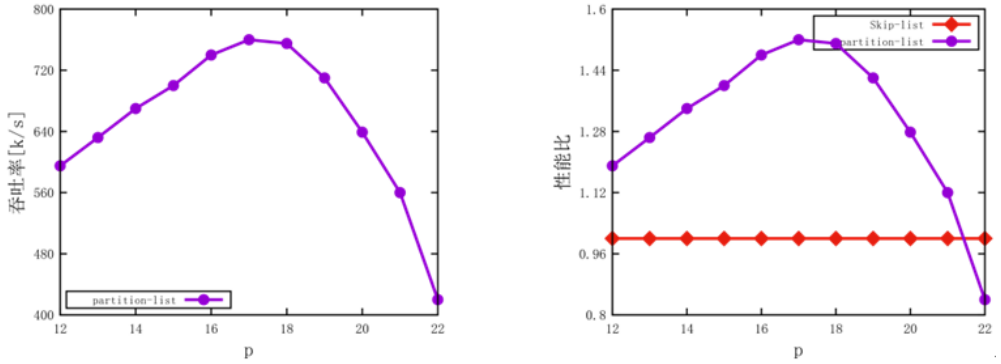
(a) 不同数据集大小下的查询吞吐率 (b) 不同数据集大小下的性能比

Fig.10 Performance influenced by dataset size with uniform distribution

图 10 均匀分布下,数据集大小对算法性能的影响

Partition-list 算法的性能受参数 p 的影响.图 11 展示了在 2^{21} 数据集大小的均匀分布数据集下,参数 p 对 *partition-list* 的吞吐率影响.实验结果表明,p 只有在一个合理的区间内,*partition-list* 的性能才会好,尤其是 p 的值接近于 $\log_2(N)$之后会发生急剧下降.我们给出了 p 的一个经验参考值:$\log_2(N)-5$.此外可以看出,*partition-list* 的吞吐率在 p=17 时和 *bound-list* 性能接近.由于 *Partition-list* 对于数据集的大小信息未知,这会导致 *partition-list* 的高度比 *bound-list* 要高.实际上,*bound-list* 和 *skiplist* 一样,平均高度都是 2.而对于 *partition-list* 的平均高度一

定大于 2.比如图 7 中,大于 max*level–p* 层(红线以上)的节点的平均高度是 2+max*level–p*(一定大于 2),而小于等于 max_*level-p* 的节点的平均高度是 2.所以整体的平均高度一定大于 2.一些不需要的高度原因导致了 *partition-list* 相对 *bound-list* 的性能损失.

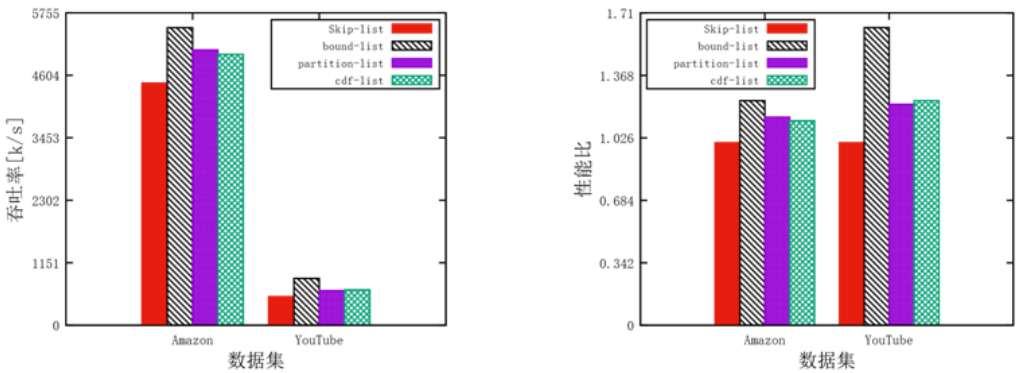


(a) 不同参数 *p* 下的查询吞吐率 (b) 不同参数 *p* 下的性能比

Fig.11 Performance influenced by parameter *p*

图 11 参数 *p* 对性能的影响

图 12 是真实数据集 Amazon 和 YouTube 的实验结果.实验结果表明:(1) 算法在小数据 Amazon 集合下性能加速不明显,在大数据 Youtube 下性能加速很明显,这与图 10 的结果一致;(2) 小数据下的优化效果不如大数据集下的优化效果.*skiplist* 的优化算法尤其时 *bound-list* 更加适合于大数据集.



(a) 真实数据集合下的查询吞吐率 (b) 真实数据集合下的性能比

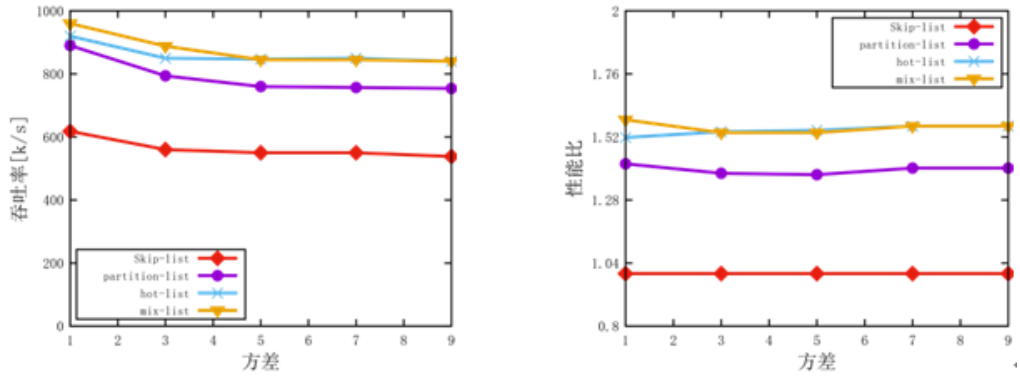
Fig.12 Performance under real dataset

图 12 真实数据集合下的性能

5.4 热度数据实验结果

本节对比 *skiplist*,*partition-list*,*hot-list*,mix-*list* 这几个跳表.为了反映数据库的访问热度特征,我们针对数据集中 1%的数据查询 80 次,其他的数据只查询 1 次的方式模拟冷热数据.

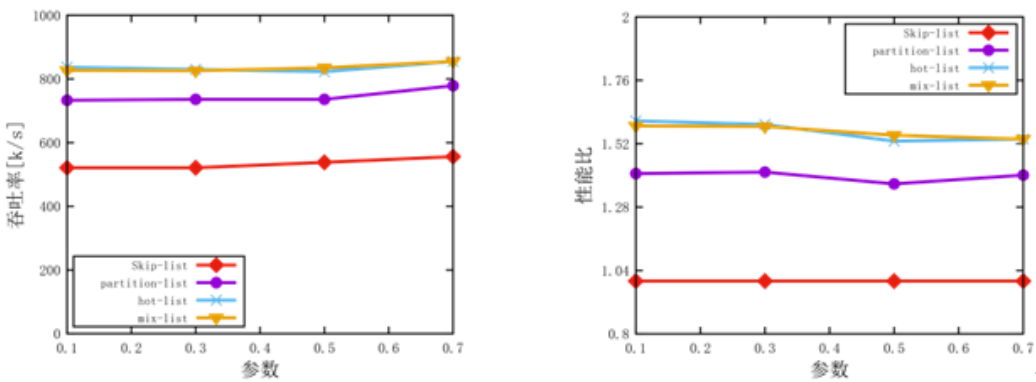
图 13 反映的是基于正态分布不同方差的数据集下的性能对比图,图 14 反映的是基于齐夫分布不同参数的数据集下的性能对比图.实验结果表明:(1) *hot-list* 可以获取大致 50%的性能提升;(2) *hot-list* 基本不受数据集分布的影响;(3) mix-*list* 相比 *hot-list* 几乎没有性能提升.



(a) 吞吐率受方差的影响 (b) 性能比受方差的影响

Fig.13 Performance influenced by the parameter of normal distribution's variance

图 13 正态分布下方差对性能的影响

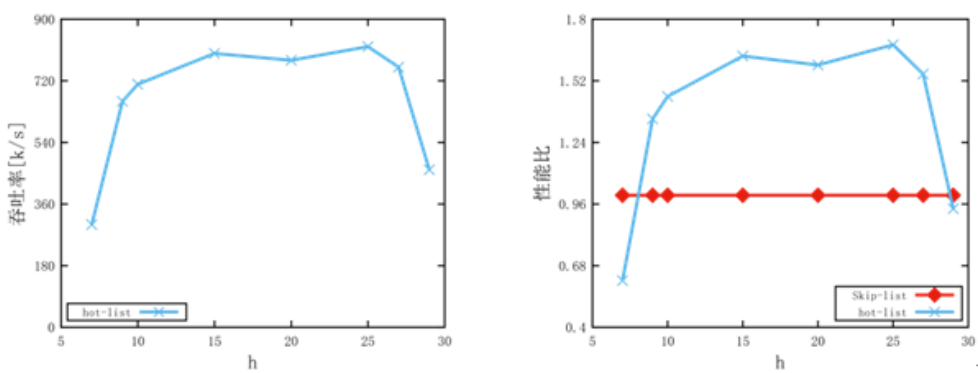


(a) 吞吐率受齐夫参数的影响 (b) 性能比受齐夫参数的影响

Fig.14 Performance influenced by the parameter of zipfian distribution

图 14 齐夫分布下参数对性能的影响

hot-list 的性能会受参数 *h* 影响,图 15 展示了在 2^{21} 数据集大小的均匀分布数据集下,参数 *h* 对 *hos-list* 性能比的影响.



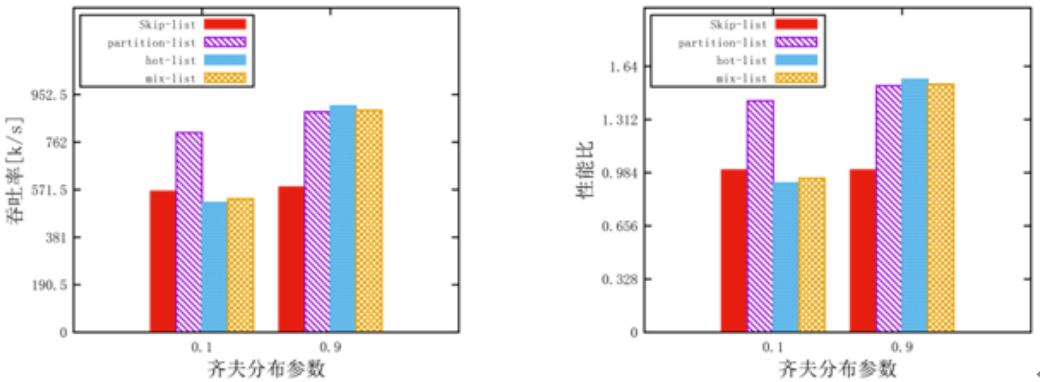
(a) 不同参数 *h* 下的查询吞吐率 (b) 不同参数 *h* 下的性能比

Fig.15 Performance influenced by parameter *h*

图 15 参数 *h* 对性能的影响

实验结果表明,*hot-list* 算法的 *h* 对性能有很大的影响,*h* 只有在一个合理的区间内才能有加速的效果.我们可以给出 *h* 的参考区间:[$\log_2(M)$,$\log_2(N)$],其中,*N* 为数据集的个数,*M* 为热度数据的个数.

此外,为了反映热度数据的频度特征对不同跳表算法性能的影响,我们采用服从齐夫分布、参数分别为 0.7 和 0.9 的两个分布分别模拟热度数据,图 16 展示了不同算法的性能结果,其实参数 *p*=17,*h*=10.



(a) 不同参数下的查询吞吐率 (b) 不同参数下的性能比

Fig.16 Performance under different zipf distribution dataset

图 16 访问频度(不同齐夫分布参数)对性能和性能比的影响

我们可以发现,在参数为 0.9 时,6 个算法的性能普遍好于 0.1 参数时的性能.这是因为数据高频访问时,cache 利用率更高.注意,*hot-list* 和 mix-*list* 只有在热度数据明显的时候(参数为 0.9),才有性能的提升.

6 结论及展望

本文重点研究数据库跳表索引优化这一经典问题.针对跳表由于随机性导致的不稳定性问题,提出了 3 个优化算法 *cdf-list*,*bound-list*,*partition-list*.针对热度数据需要被频繁访问的问题,采用冷热分层处理跳表节点的方式,提出了 *hot-list* 和 mix-*list* 跳表算法.在已知数据集大小的情况下,*bound-list* 的性能提升明显;如果对数据集大小不可知的情况下(一般情况),*partition-list* 也能获取理想的性能提升.实验结果表明,*partition-list* 可以获得最大 60%的性能提升.此外,*hot-list* 可以用作在冷热数据明显的场景,比如类似“双十一”热度商品大促的情形.以本文的 *partiton-list* 算法为代表的设计方案,可以给未来的科研人员和开发人员提供一个新的设计方向.

未来工作中,我们可以考虑基于人工智能的手段去优化 *cdf* 的学习过程和冷热数据学习的过程.本文的跳表结构并没有针对多核特征去做优化,未来可以结合多核的特性进一步思考.

References:

[1] Yang ZK. The architecture of OceanBase relational database system. Journal of East China Normal University (Natural Science), 2014,5:141−148+163 (in Chinese with English abstract).

[2] Pugh W. Skip lists: A probabilistic alternative to balanced trees. Communications of the ACM, 1990. 668−676.

[3] Graefe G, Larson PA. B-Tree indexes and CPU caches. In: Proc. of the ICDE. 2001. 349−358.

[4] Rudolf B, McCreight E. Organization and maintenance of large ordered indices. In SIGFIDET (Now SIGMOD).1970. 107−141.

[5] Tobin L, Carey M. A study of index structures for main memory database management systems. In: Proc. of the VLDB. 1986. 294−303.

[6] Rudolf B. Symmetric binary B-trees: Data structure and maintenance algorithms. Acta informatica, 1972. 290−306.

[7] Joan B, Larsen K. Efficient rebalancing of chromatic search trees. Journal of Computer and System Sciences, 1994. 667−682.

[8] Rao J, Ross K. Making B+-trees cache conscious in main memory. In: Proc. of the SIGMOD. 2000. 475−486.

[9] Changkyu K, Chhugani J, Satish N, Sedlar E, Nguyen A, Kaldewey T, Lee V, Brandt S, Dubey P. Fast: Fast architecture sensitive tree search on modern cpus and GPUs. In: Proc. of the SIGMOD. 2010. 339−350.

[10] Krzysztof K. B+-Tree optimized for GPGPU. In: Proc. of the OTM Confederated Int'l Conf. 2012. 843−854.

[11] Amirhesam S, Jacobsen HA. A hybrid B+-tree as solution for in-memory indexing on CPU-GPU heterogeneous computing platforms. In: Proc. of the SIGMOD. 2016. 1523−1538.

[12] Matthias B, Schlegel B, Volk PB, Fischer U, Habich D, Lehner W. Efficient in-memory indexing with generalized prefix trees. In: Proc. of the BTW. 2011. 227−246.

[13] Edward F. Trie memory. Commun. ACM, 1960. 490−499.

[14] Thomas K, Schlegel B, Habich D, Lehner W. Kiss-Tree: Smart latch-free in-memory indexing on modern architectures. In: Proc. of the DaMoN. 2012. 16−23.

[15] Viktor L, Kemper A, Neumann T. The adaptive radix tree: Artful indexing for main-memory databases. In: Proc. of the ICDE. 2013. 38−49.

[16] Roberto G, Gupta A, Vitter JS. High-Order entropy-compressed text indexes. In: Proc. of the SODA. 2003. 841−850.

[17] Roberto G, Ottaviano G. The wavelet trie: Maintaining an indexed sequence of strings in compressed space. In: Proc. of the PODS. 2012. 203−214.

[18] Maurice H, Shavit N. The Art of Multiprocessor Programming. Elsevier, 2012.

[19] Ittai A, Aspnes J, Yuan J. Skip B-trees. In: Proc. of the OPODIS. 2006. 366−380.

[20] Joseph H, Stonebraker M, Hamilton J. Architecture of a database system. In: Proc. of the Foundations and Trends in Databases. 2007. 141−259.

[21] Parzen E. On the estimation of a probability density function and mode. Annals of Mathematical Statistics, 1962, 1065−1076.

[22] Fukunaga K, Hostetler LD. Optimization of *k*-nearest neighbor density estimates. IEEE Trans. on Information Theory, 1973, 320−326.

[23] Magdonismail M, Atiya A. Neural networks for density estimation. Neural Information Processing Systems, 1998. 522−528.

[24] Bishop CM. Mixture Density Networks. 1994.

[25] Zhang SD. From CDF to PDF—A density estimation method for high dimensional data. arXiv: Machine Learning, 2018.

[26] Murray I, Larochelle H. The Neural autoregressive distribution estimation. Journal of Machine Learning Research, 2016, 1−37.

[27] Huang JC, Frey BJ. Cumulative distribution networks and the derivative-sum-product algorithm: Models and inference for cumulative distribution functions on graphs. Journal of Machine Learning Research, 2011, 301−348.

[28] Kraska T, Beutel A, Chi EH, Dean J. Neoklis polyzotis: The case for learned index structures. In: Proc. of the SIGMOD. 2018. 489−504.

[29] Galakatos A, Markovitch M, Binnig C, Fonseca R, Kraska T. A-Tree: A bounded approximate index structure. In: Proc. of the SIGMOD. 2019.

[30] Ding JL, Minhas UF, Zhang HT, Li YN, Wang C, Chandramouli B, Gehrke J, Kossmann D, Lomet DB. ALEX: An updatable adaptive learned index. 2019.

[31] Wu YJ, Yu J, Tian YY, Sidle R, Barber R. Designing succinct secondary indexing mechanism by exploiting column correlations. In: Proc. of the SIGMOD. 2019. 1223−1240.

[32] Pavlo A, Angulo G, Arulraj J, Lin HB, Lin JX, Ma L, Mowry PMTC, Perron M, Quah I, Saanturkar S, Toor ATS, Van Aken D, Wang ZQ, Wu YJ, Xian R, Zhang TY. Self-Driving database management systems. In: Proc. of the CIDR. 2017.

[33] Chaudhuri S, Narasayya V. Self-Tuning database systems: A decade of progress. In: Proc. of the VLDB. 2007.

[34] Van Aken D, Pavlo A, Gordon GJ, Zhang BH. Automatic database management system tuning through large-scale machine learning. In: Proc. of the SIGMOD. 2017. 1009−1024.

[35] Tan J, Zhang TY, Li FF, Chen J, Zheng QX, Zhang P, Qiao HL, Shi Y, Cao W, Zhang R. iBTune: Individualized buffer tuning for large-scale cloud databases. In: Proc. of the PVLDB. 2019. 1221−1234.

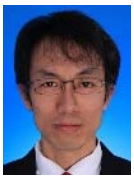
[36] Wang JD, Shen HT, Song JK, Ji JQ. Hashing for similarity search: A survey. 2014. [doi: CoRR,abs/1408.2927]

[37] Turcanik M, Javurek M. Hash function generation by neural network. In: Proc. of the NTSP. 2016. 1−5.

附中文参考文献:

[1] 阳振坤.OceanBase 关系数据库架构.华东师范大学学报(自然科学版),2014(5),141−148+163.

李梁(1992－),男,湖北黄冈人,博士,主要研究领域为内存数据库.



吴刚(1978－),男,博士,副教授,CCF 专业会员,主要研究领域为内存数据库,知识图谱.

王国仁(1966－),男,博士,教授,博士生导师,CCF 杰出会员,主要研究领域为不确定数据管理,数据密集型计算,可视媒体数据管理与分析,非结构化数据管理,分布式查询处理与优化技术(主要包括传感器网络和 P2P 对等计算),生物信息学.