面向内存数据库的类字典树索引综述与性能比较

储召乐[1)] 罗永平[1)] 金培权[1),2)]
[1)](中国科学技术大学计算机科学与技术学院 合肥 230027)
[2)](中国科学院电磁空间信息重点实验室 合肥 230027)

摘 要 如何快速存取海量数据是大数据时代数据库系统面临的重大挑战.利用大内存构建内存数据库系统是实现大数据实时存取的可行途径.在此背景下,用于加速内存数据存取的内存数据库索引成为近几年国内外的研究热点.但是,内存数据库索引也面临着诸多挑战.以常见的内存 B+树索引为例,第一个问题是索引的空间效率较低,这是因为内存 B+树索引的节点内部存在较大的空间浪费;第二个问题是索引的查询复杂度较高,B+树的查询复杂度受限于数据规模,随着数据规模的扩张,索引的搜索效率也会下降;第三个问题是变长数据支持弱,B+树对于变长键的支持比较差,往往难以适应实际应用的需要.近年来,由于字典树具有空间代价低、查询效率与数据规模无关、支持变长键等优点,逐步成为了内存数据库索引研究中的一个主要方向.本论文围绕面向内存数据库的类字典树索引,首先介绍了字典树的概念、特点和历史,然后系统梳理和总结了类字典树索引的现状和最新进展,之后提出了一种全新的分类方法对类字典树索引进行了分类.在此基础上,论文对主流的六种类字典树索引进行了实验,在多个数据集和负载上进行了性能对比,并基于实验结果讨论了类字典树索引的设计和使用建议,最后展望了未来类字典树索引的发展方向.

关键词 内存数据库;字典树索引;性能对比
中图法分类号 TP311 **DOI号** 10.11897/SP.J.1016.2024.02009

Review and Performance Comparison of Trie-like Indexes for Main-Memory Databases

CHU Zhao-Le[1)] LUO Yong-Ping[1)] JIN Pei-Quan[1),2)]
[1)](*School of Computer Science and Technology, University of Science and Technology of China, Hefei* 230027)
[2)](*Key Laboratory of Electromagnetic Space Information, Chinese Academy of Sciences, Hefei* 230027)

Abstract Accelerating query performance over massive data has become a major challenge in database systems in the big data era. Building main-memory database systems on top of large memory has been a feasible way to support real-time big data access. Accordingly, main-memory indexes for accelerating data access have been a research focus in recent years. However, main-memory indexes face a few challenges. Taking the main-memory B+tree as an example, the first issue is its low space efficiency stemming from the space overheads incurred by the internal nodes. Second, the B+tree's query complexity directly correlates with data size, leading to diminishing search efficiency as dataset volumes escalate. Third, the B+tree cannot deal with variable-length keys efficiently, failing to meet the real requirements of applications. In recent years, trie-like indexes (also called trie indexes) have emerged as a focal point in main-memory index research due to their advantages of low space cost, high data-scale-independent query efficiency,

收稿日期:2023-05-17;在线发布日期:2024-04-24.本课题得到了国家自然科学基金面上项目"面向异构混合内存的NVM感知索引及自适应学习方法研究"(No.62072419)的资助.**储召乐**,博士研究生,主要研究领域为学习索引、面向NVM的数据库系统.E-mail:czle@mail.ustc.edu.cn.**罗永平**,博士研究生,主要研究领域为数据库索引、面向新型硬件的键值存储系统.**金培权**(**通信作者**),博士,副教授,博士生导师,计算机学会(CCF)杰出会员,主要研究领域为面向新型硬件的数据库技术、大数据存储与管理、移动对象数据库.E-mail:jpq@ustc.edu.cn.

and supporting variable-length keys. This paper focuses on trie-like indexes for main-memory databases. We first introduce some foundational aspects of trie indexes, including their concept, characteristics, and history. Then, we summarize the current achievements and advances of trie-like indexes. Next, we provide a new taxonomy to classify trie-like indexes, based on which we implement six prominent trie-like indexes and evaluate their performance on various datasets and workloads. Further, we present some suggestions for the design and use of trie-like indexes based on the experimental results. Finally, we discuss the future development of trie-like indexes.

Keywords main-memory database; trie index; performance comparison

1 引 言

随着大数据时代的来临，人类社会产生的数据规模大幅度增长，上层应用对数据访问的性能需求持续攀升，这对数据库系统提出了诸多挑战. 一方面，数据库存储系统需要保证数据安全可靠的存储，另一方面需要为数据库使用者提供更高的读写性能和更低的访问延时. 对于后者，数据库存储系统中引入数据库索引技术，为数据提供快速的定位功能. 在传统的数据库系统中，数据库一般存储在磁盘或者SSD等介质中，数据库索引的引入能够显著减少数据存取时的I/O代价，从而降低访问数据的时间代价. 随着现代内存技术的发展，内存的性价比在逐渐提升；在过去的30年内，内存的价格每五年降低10%[1]，单机内存容量从数十GB提升至TB级别，使得数据和索引全部放置在内存中成为可能，因此也催生了内存数据库和内存索引技术的研究.

相较于传统的磁盘索引技术，内存数据库索引的设计在访问代价组成、内存效率以及并发性能等多个方面都面临着全新的挑战. 首先，由于内存数据库索引完全存放在内存中，没有对磁盘等I/O设备的访问，因此内存访问成为了索引读写的主要代价；其次，尽管现代计算机系统的内存容量在不断增加，但是相较于传统的磁盘存储设备，内存依旧十分宝贵，所以内存数据库索引的空间占用成为了需要关注的设计重点；最后，在如今的计算机系统中，处理器的核数在不断提升，系统的并行度在不断增加，如何支持大规模的并发读写也成为了评价内存数据库索引性能的重要方面.

按照所采用的技术机理，内存数据库索引大致可以被分为三类：哈希表类、B+树类以及字典树类.

(1)哈希表类索引. 哈希表是一种被广泛使用的索引结构，其优点在于高效的点查询性能. 哈希表能够实现$O(1)$的操作时间复杂度，其插入和查询效率通常能够达到B+树索引的4到5倍[2]，这对数据库存取效率的提升起到了至关重要的作用. 然而，由于哈希表并不是有序结构，所以无法支持高效的范围查询，这一缺陷也限制了哈希表的应用范围，目前哈希表被广泛应用于内存键值存储系统中，比如Redis、Memcached等，但对于关系型数据库如MySQL、SQL Server、PosgreSQL来说，B+树索引的使用率仍然远远高于哈希表.

(2)B+树类索引. 相较于哈希表，另一种更常使用的是B+树索引，其拥有稳定的点查询性能，并且能够支持高效的范围查询，因而在传统数据库和内存数据库中都得到了广泛应用. 早在20世纪80年代，T-Tree[3]在内存数据库中，但是由于内存容量的限制，该种索引不会在节点中存放数据，而是存放指向实际数据的指针，这样的结构导致了大量的指针追逐，使得内存访问效率十分低下. 随着内存和处理器技术的发展，T-Tree已经不再适用于现代的数据库系统，研究者也将目光转向提升索引的内存访问效率和并发性能，一系列提升索引缓存效率和并发性能的B+树索引随之提出，比如CSB+Tree[4]、Pb+Tree[5]和Bw-Tree[6,7]等. 其中CSB+Tree设计了缓存友好的索引结构提升了索引的缓存效率，其通过将节点进行分组的方式，减少了指针代价，使得内部节点的扇出更大，从而降低了内存B+树的树高，减少了索引查询过程中的缓存缺失数量；Pb+Tree使用缓存预取操作大大提升了搜索效率；而Bw-Tree则通过设计无锁的索引结构提升了索引的并发性能. 但是B+树也面临着几个问题：首先就是索引的内存效率问题，B+树的内部节点不会存放实际的数据，只会用于索引叶子节点，同时

由于需要支持快速的平衡操作,其内部节点和叶子节点都会留出一些空位,因而索引的内存效率不佳,而内存效率对于内存数据库来说是一个十分关键的问题;其次,B+树的查询复杂度取决于数据规模,由于内存 B+树的节点大小要远小于磁盘 B+树(通常设置为 256 字节),当数据规模过大时,索引高度的增加会引发严重的性能下降,随着现代社会数据规模的不断增长,需要一种更有效的索引结构进行处理;最后,B+树难以有效支持变长数据的索引,一种常用的方法是使用字典表编码的方式将字符串数据进行编码以提升索引性能[8],但是这种方法会引入额外的内存和编码代价,在一些应用场景下,使用者通常会倾向于使用简单的键值存储,并不会引入编码机制,所以 B+树无法进行高效的处理.

(3)字典树索引.为了解决变长数据的高效检索问题,研究者们提出了字典树索引.字典树又称前缀树、数字搜索树,是一种有序的树形索引.字典树结构在检索变长数据方面具有天然的优势,且其查询复杂度与数据规模无关.字典树拥有比 B+树更高效的点查询性能,同时又能支持哈希表无法支持的范围查询,所以近年来受到越来越广泛的关注.字典树的概念在 20 世纪 50 年代被正式提出[9],起初只被用来检索字符串数据,后来的研究者将字典树进行了泛化,提出了众多类字典树索引,使其成为能快速检索任意数据类型的数据结构.进入 21 世纪,随着内存和处理器技术的飞速发展,研究者们提出了一系列内存高效且性能优异的类字典树结构,比如 Masstree[10]、ART[11]、HOT[12]、Wormhole[13]、HydraList[14]、CuckooTrie[15]等,这些索引通过设计缓存高效的索引结构和高效的并发机制取得了良好的性能.

随着近几年内存数据库研究方向的不断发展,系统梳理类字典树索引的研究进展已经十分必要,但是以往还没有这方面的综述工作.本论文聚焦于基于类字典树的内存数据库索引(在后续文字中,如果不特别解释,"索引"均指"内存数据库索引").论文对已有的类字典树索引技术进行了归纳和分析,并给出了未来发展展望.与以往综述相比,本论文的主要特色如下:

(1)本文提出了一种全新的类字典树索引的分类方法,对主流的类字典树索引进行了全面的分类和细致的总结对比,并且对近十年来最受关注的几种类字典树索引进行了详细分析和实验对比,为类字典树索引的研究方向提供了参考.

(2)本文完成了对近十年提出的六种类字典树索引较全面的性能评测,并给出了类字典树索引的使用和设计建议.

(3)本文讨论了类字典树索引的未来发展趋势,给出了值得研究的研究方向,为未来开展类字典树索引的研究与应用提供了新的建议.

本文后续内容安排如下:第二节简要介绍了字典树索引的基本概念、结构、历史和特点;第三节给出了类字典树索引的分类和对比;第四节对主流的类字典树索引进行了实验和性能对比,并基于性能对比结果给出了类字典树索引的设计和使用建议;第五节给出了类字典树索引的未来发展展望;最后第六节将对全文进行了总结.

2　字典树索引概述

2.1　字典树索引的基本概念与特点

字典树[9,16](Trie)又称前缀树,是一种有序的树形数据结构,主要用于索引变长数据.图 1 展示了一棵简单的字典树结构.与 B+树、红黑树等数据结构不同,字典树的每个节点只会存放键的部分片段(如字符串数据中的一个字符),不会将完整的键存放在节点中.在查询过程中,用户只需要使用与节点位置相匹配的键分片进行比较,就可以定位到下层的节点,这样做避免了代价高昂的完整键比较,也是字典树特别适合索引变长数据的原因所在.

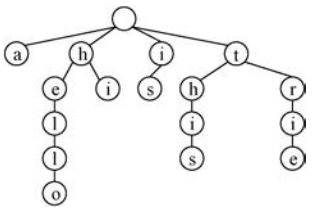

图 1　一棵简单的字典树

字典树的另一个优点是其稳定的结构,因为其结构完全取决于键空间以及键分布,与键的插入顺序无关,所以无需进行类似 B+树的平衡操作.此外,字典树还有一个优点,即与数据规模无关的操作复杂度,与 B+树完全依赖于数据规模的 $O(\log n)$ 操作复杂度不同,字典树的操作复杂度只取决于键的长度,在键长度有限的场景下,字典树能够达到媲美哈希表的常数操作时间.

字典树的一个特点在于其结构隐式地表示了完整的键,用户能够根据查询路径重构出完整的键,因而无需额外存储键.但是,字典树的结构设计需要使

表 1 类字典树索引总结

索引名称	结构	并发机制	存储场景	设计特点
Kiss-Tree[17]	纯 trie	RCU 机制	内存	三层结构,无锁更新
ART[11,18]	纯 trie	OLC/ROWEX	内存	动态调整节点结构
HOT[12]	纯 trie	ROWEX	内存	动态调整节点跨度
FST[19]	纯 trie	无	内存	两种编码方式混合, 上层节点侧重性能, 下层侧重内存效率
Hyperion[20]	纯 trie	无	内存	内部节点两层字典树, 内存高效的编码
Burst trie[21]	trie+其他	无	内存	trie 和其他数据结构混合, 达到时间和空间的平衡
HAT-trie[22]	trie+哈希表	无	内存	使用哈希表优化 Burst trie, 提升缓存效率
Masstree[10]	trie+B+树	乐观并发控制	内存	内部节点使用 B+树, 降低树高,提升缓存效率
Wormhole[13]	trie+哈希表+B+树	叶子节点读写锁, 哈希表使用 RCU 机制	内存	哈希表存储 trie 节点, 叶子节点使用 B+树节点
HydraList[14]	trie+B+树	内部节点 ROWEX, 叶子节点乐观并发控制	内存	数据层与搜索层解耦合,搜索层使用 ART, 数据层使用 B+树节点, 适应 NUMA 架构
CuckooTrie[15]	trie+哈希表	乐观并发控制	内存	哈希表存储 trie 节点, 预取操作提升缓存效率
WOART[23]	纯 trie	OLC/ROWEX	非易失内存	对非易失内存进行适配
ROART[24]	trie+叶子数组	OLC	非易失内存	叶子数组,选择性持久化
PACTree[25]	trie+B+树	内部节点 ROWEX, 叶子节点乐观并发控制	非易失内存	对非易失内存进行适配
ERT[26]	trie+哈希表	细粒度读写锁	非易失内存	trie 节点使用哈希表
SMART[27]	纯 trie	内部节点无锁, 叶子节点乐观并发控制	分离式内存	无锁内部节点, 读取委派,写入聚合

用大量的指针操作,降低了字典树的内存效率.此外,初始字典树只用于索引字符串数据,这使得其应用范围受到了限制.

字典树的一个重要概念是节点跨度(span).节点跨度指的是节点内存放的部分键的长度.举例来说,初始字典树每个节点只会存放字符串的一个字符,所以它的节点跨度就是 1 字节(或者说 8 比特).节点跨度决定了整棵树的高度以及节点的最大扇出(即每个节点的分支数目),索引的设计者可以设定索引的节点跨度.将字典树的节点跨度设为 1 比特时,得到的泛化字典树即可存储任意数据类型的键,一般将这种字典树称为二进制字典树(binary trie/bitwise trie),二进制字典树解决了初始字典树只能索引字符串数据的问题.另一个等价概念是基数,指的是每个节点的最大分叉数,对节点跨度为 1 比特的字典树来说,其基数即为 2,因此节点跨度与基数具有等价性,在后文的叙述中我们统一使用节点跨度的概念.

2.2 字典树索引的历史

字典树的思想最早在 1912 年被初次提出[28];1959 年,René de la Briandais 首次在计算机领域提出了字典树的概念[9];1960 年,Edward Fredkin 又独立提出了相同的概念,并确定了“trie”的名称[16],该名称取自于“信息检索”(information retrieval)中的“检索”(retrieval)这一单词.1963 年,E. H. Sussenguth,Jr. 提出了一种混合结构的字典树,其认为当拥有共同前缀的单词个数较少时,将这些数据存放在一个数据结构(如数组)中能够减少字典树的节点数目,提升索引的性能,例如图 1 中,共同前缀为“h”的单词只有两个:hi 和 hello,所以这两个单词可以存放在同一个数组之中,这种方法减少了字典树的节点数目,提升了索引的内存效率,同时降低了树的整体高度,有助于性能的提升,该种思想被众多的后续工作所借鉴[13,14,21,22,24,25].

由于需要存储大量的指针,字典树的内存效率较低,为此研究者提出了一种压缩字典树:基数树(radix tree).基数树的最主要特点就是其将字典树中只有一个孩子的节点合并到父节点中.这一策略带来了两点好处:一方面,合并节点使得总的节点数减少,节点使用的指针数量会减少,因此空间代价也

会随之降低;另一方面,将只有一个孩子的节点进行合并,使得整体树高降低,在插入和查询操作过程中避免了大量的指针追逐操作,从而提高了操作效率.如今的类字典树索引基本都采纳了这种思想.

1968 年,Donald R. Morrison 提出了一种新型的字典树:帕特里夏树 (Patricia)[29].该种字典树可以看作一种节点跨度为 1 比特的基数树.为了提升内存效率,Patricia 的节点只会存放特定比特位的位置,这里的特定比特位指的是不同键发生分歧的比特位.Patricia 这样的存储特点使得其内存效率得到了提高,但是也需要付出相应的代价:由于完整的键无法通过查询路径重构出来,所以查询不存在的键时也可能返回正确结果,因此在查询过程的最后阶段需要进行完整键的比较以确保正确性.

2.3　类字典树索引介绍

现代的类字典树索引依旧延续着初始字典树索引的基本思想,并通过结构上的设计以提升索引的内存效率和时间性能,同时适应各种不同的存储场景,表 1 总结了现在的一些类字典树的结构特征、并发机制、面向的存储场景以及设计特点.本节我们将简要介绍这些索引的一些基本特点,并在第 3 节中对这些索引进行全面的分类和比较.

2.3.1　类字典树索引介绍

本节将简要介绍现代类字典树索引的特点.

Kiss-Tree[17]是一种无锁的类字典树索引,其用于存储不长于 4 字节的键.Kiss-Tree 使用了三层结构,并使用 RCU(read-copy-update)技术实现无锁更新操作.

ART(Adaptive radix tree)[11]是一种可以动态调整节点扇出(fanout)的基数树,被用于内存数据库 Hyper[30]中.ART 的节点跨度为固定的 1 字节,但是其节点扇出可以是 4、16、48 或 256,这四种类型的节点性能和内存效率各不相同.ART 通过四种节点类型之间的自适应转换达到了时间与空间性能的平衡.ART 在索引分布较为密集的数值型数据时拥有十分高的插入和查询效率[12-15],但是真实世界的字符串数据往往分布较为稀疏,ART 结构中会存在大量扇出较小的节点,从而影响了索引的查询和插入性能.为了避免因为数据分布对索引性能的影响,研究者提出了一种可以进行动态节点平衡操作的字典树索引:HOT(Height Optimized Trie)[12].HOT 使用了一种可以根据数据分布动态调整节点跨度的技术,使得索引节点的扇出稳定在 32 左右,并且使用了一种精心设计的物理节点结构提升索引的内存效率和缓存效率,HOT 还使用了缓存预取操作和 SIMD 指令进一步提升索引性能.

FST[19]和 Hyperion[20]是一类内存极其高效的类字典树索引,它们通过对节点结构进行编码的方式,以牺牲性能为代价提升索引的内存效率.FST 的设计基于这样一项观察:字典树索引的上层节点数目较少,但是被频繁访问,下层节点数目较多,但是访问较少.于是 FST 提出在索引上层使用侧重性能的编码方式,下层则使用侧重内存效率的编码方式,使得整个索引在空间高效的同时不会损失过多性能.Hyperion 则将索引的节点跨度提升至 16 比特,每个节点依然是一个两层的字典树,但是通过压缩编码的方式将其存储为一个字节数组,并且设计了新的内存分配器提升索引的内存效率.

Burst trie[21]是一种混合结构的字典树,用于存放字符串数据.Burst trie 将字典树的叶子节点变成可以存放多条数据的其他数据结构,称为容器,通过这种方式可以降低索引的高度,提升索引的内存效率和时间性能.Burst trie 的容器可以是任意一种可以用于有效存储字符串数据的数据结构,比如链表、二叉搜索树、哈希表等.HAT-trie[22]则是对 Burst trie 的一种优化,由于链表、二叉搜索树等容器结构的搜索效率十分低下,会对索引的整体性能造成影响,所以 HAT-trie 使用一种缓存感知的哈希表作为容器,提升索引的性能.

HydraList[14]则是一种结合 B+树和字典树的混合索引,和 Burst trie 的思想类似,其叶子节点设置为可以存放多条数据的 B+树叶子节点结构,HydraList 将整个索引解耦成上下两层,上层由一棵 ART 构成(称之为搜索层),叶子节点则采用 B+树的叶子节点结构(称之为数据层),上层的 ART 索引每个叶子节点的最小值.HydraList 采用数据层与搜索层解耦合的策略来提升索引的并发性能.同时 HydraList 在不同的 NUMA 节点中都会维护一份搜索层的副本,尽可能减少跨节点的内存访问以适应 NUMA 架构.

Masstree[10]另一种字典树与 B+树的混合结构,被用于内存数据库 Silo[31]中.与同样是字典树与 B+树结合的 HydraList 不同,Masstree 将节点跨度设置为 8 字节,然后将每个节点组织成一棵内存 B+树,用来实现节点内的快速搜索.为了提升索引的性能,Masstree 使用了缓存行预取等操作提升缓存效率.Wormhole[13]则是一种结合了哈希表、B+树和字典树三种结构的混合索引.Wormhole 的

叶子节点使用了 B+树的叶子节点结构，每个叶子节点通过锚点键被上层的索引检索，而上层的索引则是字典树与哈希表的混合结构. Wormhole 将字典树索引的所有节点存放在一个哈希表中，消除节点之间的依赖关系，并使用一种改良的遍历算法达到了 O(log L)的查询复杂度(其中 L 表示键的长度). 而 CuckooTrie[15]是一种结合哈希表和字典树的混合索引，与 Wormhole 类似，CuckooTrie 将字典树索引的所有节点存放在哈希表中，具体的结构我们在第 3.4 节中进行介绍. CuckooTrie 通过使用哈希表的方式，一方面消除了指针的代价，另一方面消除了节点之间的依赖关系，使得索引能够同时访问多个层级的节点. 利用这一特性，CuckooTrie 使用缓存预取操作大大提升了索引的内存级并行度，实现了更高的查询效率.

2.3.2 面向新存储场景的类字典树索引

前面所涉及到的类字典树索引均为传统计算机系统结构下的内存数据库索引，随着计算机技术的发展，还出现了一些新的存储介质和新的存储结构，一些工作面向这些新的存储场景设计了类字典树索引.

非易失内存是一种近年来兴起的新型存储介质，其拥有字节寻址和持久性存储的特性，针对这一新型存储设备，研究者也提出了一系列类字典树索引设计. WOART[23]将 ART 移植到了非易失内存上，其整体结构与 ART 一致，由于非易失内存拥有持久存储的特性，面向该种介质的索引均需要保证失败原子性，WOART 提出了一种新的保证失败原子性的范式，并将 ART 的节点结构进行了修改以达到此目标. ROART[24]也是一种对 ART 的优化，其通过将 ART 的多个叶子节点合并到一个叶子数组的方式，达到降低树高以及提升范围查询性能的目的，ROART 也对非易失内存进行了适配. PACTree[25]将 HydraList 移植到非易失内存，整体结构并无差别，只是对非易失内存进行了特定的适配. ERT[26]将哈希表与字典树结合，但是结合方式与 Wormhole 和 CuckooTrie 并不相同，ERT 将字典树的每一个节点替换为哈希表，在降低树高、提升节点扇出的同时保证节点内部的搜索效率，从而获得更高的性能，与此同时，ERT 中的哈希表使用类似 CCEH[32]的结构以适应非易失内存的特点.

分离式内存系统是近几年兴起的一种新型存储系统，目的是提升数据中心的内存资源利用率，同时避免内存容量不足造成的系统错误[33]. 在分离式内存系统中，内存资源与计算资源被解耦，服务器节点被分为计算节点和内存节点，节点之间通过新型高宽带低延时网络硬件进行连接通信. 为了适应这种新型存储架构，SMART[27]对 ART 的节点结构进行了重新设计，使其内部节点支持无锁操作，并通过一系列与系统结构相关的优化提升索引的整体性能，如读取委托、写入聚合技术等.

2.3.3 并发控制

随着现代计算机系统处理器数目的增长和处理器核数的不断增加，索引的并发性能成为索引性能最重要的评估指标之一，因此并发控制机制的选择也成为索引设计至关重要的一部分[18,34]. 本节简要讨论类字典树索引的并发控制机制.

为了索引性能的考虑，如今的内存数据库索引基本都会选择乐观并发控制(乐观锁)[34,35]. 乐观并发控制假定索引操作没有过多地写入竞争，因而读者无需对索引节点进行加锁操作. 如今的类字典树索引基本都会选择使用版本号机制实现乐观并发控制，如 ARTOLC[18]、Masstree[10]、HydraList[14]、CuckooTrie[15]等. 在版本号机制的具体实现中，每个节点会维护一个版本号，写线程在完成修改操作后会修改该版本号，读线程则会在进行读取操作之前会保存版本号的快照，在读取完成之后检测版本号是否发生改变，如果发生改变则说明在读取过程中节点发生了修改，需要重新进行读取操作. 在实现该机制时，系统一般会将锁位和版本号编码存储在一个 64 比特的变量中，然后读者只需在读取操作之前保存版本号快照并在读取完成之后进行验证，无需对锁位进行加锁操作，而写者依然需要获取互斥锁以保证同一时刻只有一个写线程在进行修改操作.

每种索引使用乐观锁的方式可能有所不同，一种比较常用的技术为乐观锁耦合(OLC). 锁耦合[36]是一种内存 B+树的标准并发控制技术，该技术保证在遍历树的过程中任意时刻最多只持有两个锁，在遍历过程中，对父节点的锁会维持到孩子节点成功获取锁之后. 乐观锁耦合的运行流程与锁耦合十分相似，只是将锁替换为版本号[18]. 这种机制已被证明可以有效实现并发控制[35].

乐观锁在写入冲突较少的负载环境中能够实现非常好的并发性能，但是对于写入较多的负载，读操作可能会发生大量重启现象，这会对索引的并发性能造成巨大的影响. 为了避免这种现象，Viktor Leis[18]提出了另一种读优化写互斥机制(ROWEX). 这是一

种介于无锁机制与锁机制之间,又与乐观锁不同的并发控制机制.在 ROWEX 机制下,读操作并不会获取节点锁,也不会检查节点版本号,所以读操作一定会成功.而写操作则需要获取互斥锁,阻塞其他的写者对节点的修改.为了保证读操作的正确性,写者需要确保读线程读到的一定是正确的内容,不会读到节点修改的中间状态,因此使用该机制需要对索引结构进行适当的修改.目前使用该机制的类字典树索引包括 ARTROWEX[18]以及 HOT[12].

除了以上两种并发控制机制外,还有一些类字典树索引会使用 RCU 机制或者读写锁[37].比如 Kiss-Tree[17]对第三层节点的更新会使用 RCU 机制,在更新节点时首先创建新的节点副本,并将新数据插入到该私有副本中,在该节点副本准备好时则会原子地替换原有节点.ERT[26]则使用细粒度的读写锁机制保证索引的并发性能.Wormhole[13]是一种混合结构,所以其并发控制也使用了两种不同的机制,其叶子节点使用了读写锁保护,其哈希表结构则使用了 RCU 机制.

3　类字典树索引优化

近年来研究者针对字典树索引提出了多种优化策略.本节将重点讨论相关的类字典树索引优化技术.为了更系统、清晰地总结各类优化技术,我们首先提出了一种类字典树索引的分类方法,然后基于此分类方法对主流的类字典树索引进行分析和对比.

3.1　分类框架

(1)内存效率.内存是现代服务器中最昂贵的组件之一[20],而数据库的索引会占据大量的内存资源,比如 H-Store 在运行 TPC-C 时,索引可能会占据内存容量的 55%[38],因而内存效率是评价内存数据库索引的重要指标.现代的类字典树索引基本都采用了基数树的结构,使用路径压缩技术降低了索引的内存占用,但是随着数据规模的进一步增长,一些结构上的优化技术被用于进一步提升索引的内存效率,比如动态调整节点结构、利用操作系统特性降低内存空间、使用压缩算法对节点进行编码等.

(2)树高.类字典树索引的高度决定了遍历索引时的随机内存访问数量,因而直接影响索引的综合性能.类字典树索引的高度主要取决于数据的长度,尽管使用路径压缩等技术能够降低树高,但是其效果很大程度上取决于数据分布情况,研究者们提出了一些更进一步的优化技术用于降低索引的高度,包括增加叶子节点容量、扩大节点跨度、引入平衡操作降低树高等.

(3)内部遍历速度.让我们考虑类字典树索引的搜索过程,搜索开始于根部节点,经过内部节点的遍历到达叶子节点才算一次搜索的结束,所以内部节点的遍历速度会影响索引的性能表现.同样的,有一系列工作研究如何降低搜索过程中内部节点遍历的时间开销,包括降低内存访问开销、降低搜索复杂度等.

(4)范围查询性能.字典树索引是一种有序结构,可以支持范围查询,但是因为字典树索引的范围查询会涉及到跨层级的访问,导致大量的随机内存访问,所以字典树索引的范围查询性能不佳.研究者们也提出了多种优化技术提升类字典树索引的范围查询性能,包括增加叶子节点的容量、使用指针连接叶子节点等.

(5)适配新型内存.近年来出现了一些非易失内存、分离式内存等新型内存技术.面对这些新的内存介质及系统架构,类字典树索引也需要进行特定的适配和优化,目前已经有相当多的工作对其进行研究.

围绕优化内存数据库索引的目标,本文提出了一种新的内存数据库索引分类方法,如图 2 所示.基于这一分类方法,我们对主要的类字典树索引进行了分类和总结,结果如表 2 所示.下面我们对各种优化技术进行详细的介绍和分析.

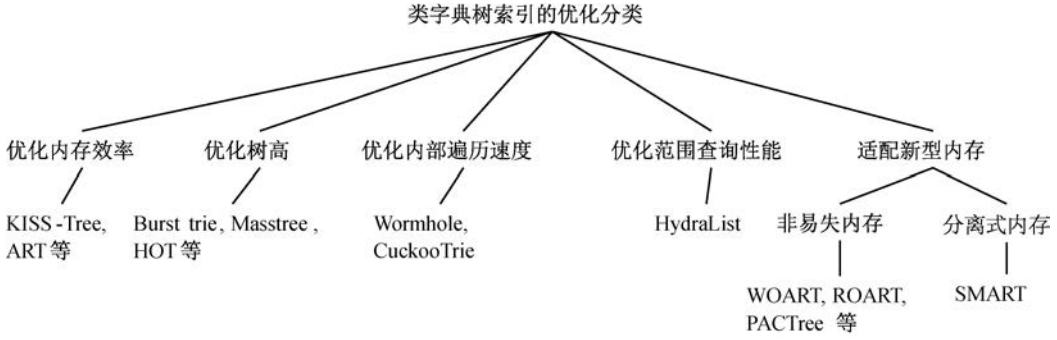

图 2　类字典树索引的优化分类

表 2 类字典树索引分类(♠表示主分类,△表示次级分类)

索引名称	优化内存效率	优化树高	优化内部遍历速度	优化范围查询性能	适配新型内存
ART[11,18], Kiss-Tree[17], FST[19], Hyperion[20]	♠				
Burst trie[21], HAT-trie[22], Masstree[10]		♠			
ERT[26]		♠			△
HOT[12]	△	♠			
Wormhole[13], CuckooTrie[15]			♠	△	
HydraList[14]				♠	
WOART[23]					♠
ROART[24], PACTree[25]				△	♠
SMART[27]					♠

3.2 优化内存效率

优化类字典树索引内存效率的一个直接方法是降低树高,因为降低树高能够减少索引的节点总数,从而降低索引的内存占用.例如将多个叶子节点存放在同一个数据结构中、使用类似基数树的路径压缩技术等.本论文中我们将优化树高作为索引优化的一个直接目标,不会将其看作优化内存效率的一个手段,所以这一节我们不对其进行讨论,本节将关注在节点结构上优化内存效率的工作.

3.2.1 动态调整节点结构

动态调整节点结构的代表性索引是 ART.ART 的设计与基数树密切相关.对于基数树,一种简单的实现方法是使用指针数组表示基数树的内部节点.假设基数树的节点跨度为 s,内部节点使用一个大小为 2^s 的指针数组表示,在查询时只需使用 s 比特的数据作为数组下标就可以直接定位到下一层节点.从性能方面考虑,越大的节点跨度意味着更低的树高和更高的性能.但是,由于数据分布的特性,内部节点的指针数组可能会存在大量的空指针项,造成大量的空间浪费.因此,ART 提出了一种可以进行动态转换的节点结构,在保证性能的前提下提升索引的内存效率.

ART 的节点跨度固定为 1 字节,但是内部节点的扇出会根据实际的数据分布进行动态调整.ART 的结构如图 3 所示,它设计了四种不同类型的节点:Node4、Node16、Node48 以及 Node256.这四种节点的扇出各不相同,性能和空间消耗都随着扇出的增大而增加.利用这四种节点结构,ART 可以根据实际的数据分布对节点类型进行调整.例如,当节点存放的键分片少于 4 时,则使用 Node4 类型的节点,从而降低空间代价;当键分片数量增加到大于 16 小于 48 时,则将节点转换为 Node48 类型,在不增加

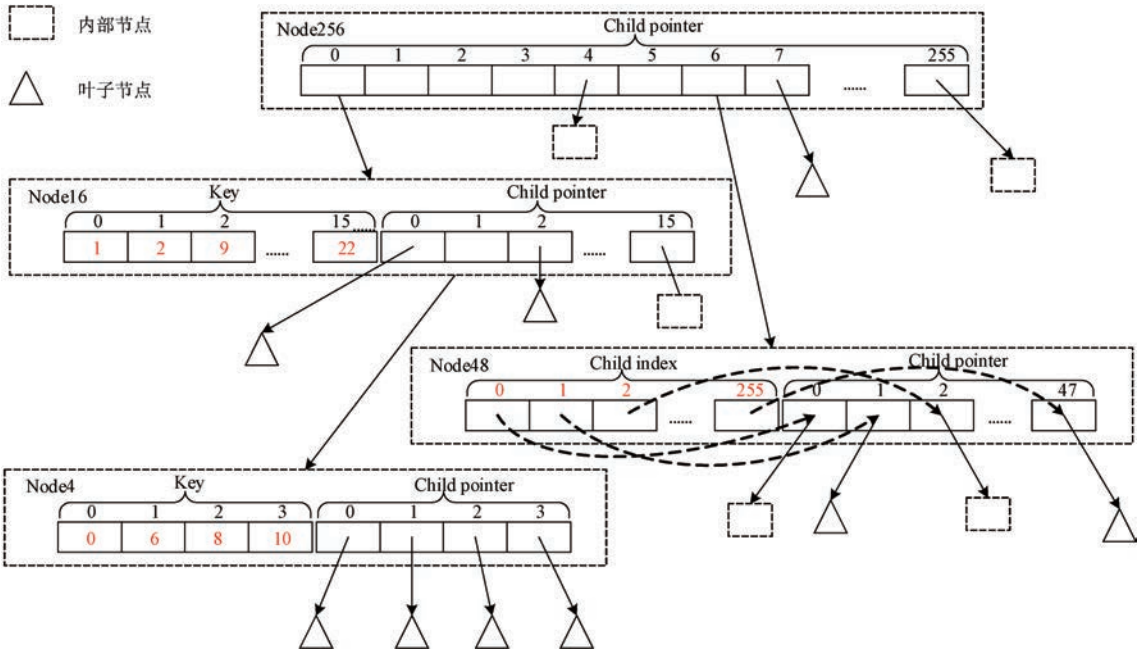

图 3 ART 索引结构

过多空间代价的前提下提升性能;类似的,当键分片数量大于 48 时,ART 将节点转换为 Node256 类型.通过这种内部节点的自适应转换过程,ART 实现了索引的性能与内存效率之间的平衡.

3.2.2　按需分配内存

另一种提升内存效率的方法是按照实际内存需求为索引分配物理内存空间.这一方法利用了操作系统分配内存的特点,即分配的虚拟内存只有在发生缺页中断时才会分配实际的物理内存.

Kiss-Tree[17]是一种三层结构的类字典树索引,其只能索引 32 比特的数据.Kiss-Tree 的第一层索引数据的前 16 比特,但是这一层并不是真实存在的,而是直接根据数据的前 16 比特计算出第二层节点的地址,这种方法一方面减少了一次内存访问,另一方面则也降低了索引的空间代价.第二层索引数据后续的 10 比特,该层节点会按照需求分配实际的物理内存,具体来说在构建索引时会向操作系统申请容纳第二层节点所需要的全部虚拟内存,但此时并不会分配实际的物理内存,只有当数据插入到第二层的某一节点时,才会向操作系统请求分配该节点的物理内存,这种方法进一步降低了索引的内存空间占用.Kiss-tree 的最后一层则使用数据的最后 6 比特数据定位到具体位置.

3.2.3　节点压缩编码

使用压缩编码方式对索引节点进行编码也是一种提升索引内存效率的高效方法,可以有效降低索引节点的空间占用.

HOT[12]可以看作一种泛化版本的帕特里夏树(Patricia),其通过平衡操作使得整个索引的高度保持在稳定的高度,具体的设计将在第 3.3 节中进行介绍.HOT 的每个节点就是一棵帕特里夏树,为了降低节点的空间代价,HOT 设计了独特的节点物理结构,通过将每个节点进行串行化,形成一个压缩的位串,降低了节点的空间代价.同时 HOT 还会通过 SIMD 指令加速节点内的搜索过程.

FST[19]的设计源于一项观察:字典树索引的上层节点数目较少,但是被频繁访问,下层节点数目较多,但是访问较少.于是 FST 提出一种混合编码的方式,上层节点使用侧重性能的编码,下层节点则使用侧重内存效率的编码.FST 基于 LOUDS(Level-Ordered Unary Degree Sequence)编码算法[39],提出了侧重性能的 LOUDS-Dense 编码和侧重内存效率的 LOUDS-Sparse.上层和下层的界限为手动设置,使用者可以根据实际情况进行设置以达到性能和内存效率的平衡.

Hyperion[20]同样基于节点编码的方式提升索引的内存效率,其节点跨度为 2 字节.Hyperion 的节点使用容器进行存储,每个节点从概念上依然是一棵两层字典树,为了提升内存效率,Hyperion 将该字典树按照前序遍历的顺序进行压缩编码并存储为一个字节数组.字节数组会随着节点元素增加而动态扩张,每次以 32 字节的粒度增长,由于使用常规的内存分配器会产生严重的内存碎片问题,所以作者提出了一种与 Hyperion 高度耦合的内存分配器,提升索引的内存效率.

3.2.4　总　结

内存效率是评估内存数据库索引的重要指标之一.本节中我们讨论了三种优化内存效率的方法:动态调整节点结构、按需分配内存以及节点压缩编码.ART 动态调整节点结构的方法简单且高效,在保证性能的前提下提升索引的内存效率,后续工作也证明了该种方法的有效性.ART 也成为了众多后续工作的基础,如 HydraList、WOART、ROART、SMART 等工作都是在 ART 的基础上进行的进一步优化.Kiss-Tree 利用其虚拟的第一层和按需分配的第二层优化了索引的内存效率,但是其只支持最长 32 比特的数据存取,实用性较低.压缩编码是一种非常有效的提升内存效率的方式,但是使用该种方法同样需要在性能和内存效率之间进行权衡.HOT 使用的编码在保证性能的前提下尽可能降低空间开销,而 FST 和 Hyperion 的目标则侧重于内存效率,在保证空间高效的前提下尽可能提升性能.设计者需要根据面对的应用场景选择合适的编码算法.

3.3　优化树高

本节将讨论类字典树中对于树高进行优化的工作,由于路径压缩技术已经被广泛使用,本节中我们将不对其进行深入的研究.

3.3.1　增加叶子节点容量

将多个叶子节点进行合并以增加节点容量是一种常用的降低索引高度的方法.

Burst trie[21]是由字典树和一系列容器构成的混合结构.对于字典树索引来说,其下层节点通常较为稀疏,所以 Burst trie 将这些稀疏的子树合并到一个容器中进行存储,从而降低了索引的整体高度.Burst trie 的容器类型包含链表、二叉搜索树、哈希表等.当索引的容器中元素数量达到一定阈值时,Burst trie 会进行爆发(Bursting)操作,将该容器分裂为一系列的字典树节点和容器节点,Burst trie 使

用启发式算法确定爆发的时机，保证索引的性能. HAT-trie 则是对 Burst trie 的进一步优化，其将容器替换为一种缓存感知的哈希表，提升索引的缓存效率，从而提升索引的性能.

ROART[24]也使用了类似的方法对 ART 进行了优化. ROART 提出叶子压缩的概念将多个叶子指针压缩到一个叶子数组中，每个叶子数组最多存放 m 个叶子指针，当子树的叶子数目小于 m 时，该子树就会被压缩到一个叶子数组中. 通过这种压缩的方式，ROART 降低了索引的高度并减少了操作过程中的指针追逐. ROART 的其他设计集中在对非易失内存场景的优化，我们将在第 3.6.1 节中进行介绍.

3.3.2 扩大节点跨度

降低节点高度的另一种方法就是扩大节点跨度. 假设索引的节点跨度为 s，对于一个长度为 k 比特的字符串，为了存储该字符串索引需要构建 $\lceil k/s \rceil$ 层内部节点，因而节点跨度越大，索引高度就越低. 但是节点跨度的提升会带来新的挑战，因为节点跨度越大节点内的元素也就越多，如果无法进行有效的节点内搜索，索引的性能可能会产生下降，所以需要提升节点内部搜索的效率. 目前有两种方法应对这种挑战：第一种是 Masstree 使用的方法，内部节点构造为一棵 B+树以提升节点内部的搜索效率；另一种则是 ERT，内部节点使用哈希表提升搜索效率.

Masstree[10]使用 8 字节的节点跨度，每个节点的最大扇出为 2^{64}，为了提升节点内的搜索效率，Masstree 使用 B+树组织节点内的数据. Masstree 将 8 字节的键分片视为 8 字节的整数进行处理，如果分片长度不足 8 字节，填充 0 进行处理. Masstree 还使用了预取操作提升性能，在查询过程中 Masstree 会并行地将需要访问的节点全部预取到缓存中，降低内存访问的代价以提升性能.

ERT[26]则使用 4 字节的节点跨度，并利用可扩展哈希表组织节点内的数据，以提升节点内搜索的速度. 尽管哈希表能够提供十分高效的点查询性能，但是无法保证数据的有序性，为了保证索引对范围查询的支持，ERT 不会对键分片进行哈希操作，而是使用键分片的原始位串进行索引. 尽管使用原始位串进行索引可能会导致大量的冲突，但是使用路径压缩技术可以减少这种冲突的发生[26]. 由于 ERT 是一种面向非易失内存的索引，所以其哈希表使用了类似 CCEH[32]的结构提升索引的性能.

3.3.3 平衡操作降低树高

最后一种降低类字典树索引高度的方法是使用类似 B+树的平衡操作，避免过于倾斜的数据分布影响树高. 这方面的代表性工作是 HOT[12].

大部分类字典树索引的节点跨度都是静态的，比如 ART 是 1 字节、Masstree 是 8 字节. 静态的节点跨度导致索引的性能和内存消耗很大程度上取决于数据的分布情况. 例如对于 ART 来说，其在数值型数据上的性能表现十分优异，但是在真实的字符串数据集下性能表现并不理想[12]. 出现这一问题的原因在于，真实数据集的分布较为稀疏，使用路径压缩技术无法有效降低索引高度，如图 4a 所示，在真实的稀疏数据集场景下，ART 的下层会产生大量扇出较小、性能较差的节点，从而影响了索引的性能.

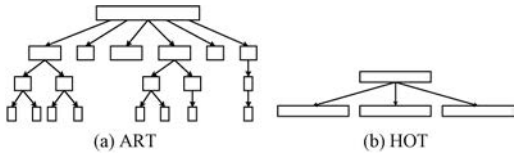

图 4 ART 与 HOT 在稀疏数据分布下的结构对比

为了避免类似 ART 这种索引在稀疏数据集下性能的不稳定，HOT 提出了一种自适应调整节点跨度的方式，使得节点扇出稳定在一个较大的值，并通过一种类似 B+树的平衡操作对整棵树的高度进行了有效压缩.

HOT 可以看作一棵泛化版本的 Patricia，它将 Patricia 的若干个节点合并成一个混合节点，这是 HOT 实际使用的节点类型. 每个混合节点可以容纳的元素数目被设定了一个阈值，当元素数目到达阈值时则对节点进行结构调整. 结构调整的过程类似于 B+树的平衡操作，主要目的是对数据进行重新分配，从而降低树的平均高度，得到的结果就是混合节点的节点跨度各不相同. HOT 的结构调整有三种不同的情况，根据实际情况，结构调整过程可能会生成新的孩子节点，可能会生成新的内部节点，也可能会影响父亲节点的结构. 结构调整的基本原则为尽可能不增加树的高度，这也是 HOT 的目标和提升性能的关键. 图 4 展示出了 HOT 与 ART 之间的结构区别，可以看到 HOT 的结构更加扁平，不会受到数据分布的影响.

3.3.4 总 结

索引的高度是影响索引性能的重要因素. 本节讨论了三种降低树高的方法：增加叶子节点容量、扩大节点跨度以及引入平衡操作. 增大叶子节点容量

是一种十分常用的方式,许多类字典树索引都采用了这一策略,包括 Burst trie、HAT-trie、ROART 等.扩大节点跨度的方式有助于降低树高,但它可能降低内存效率,比如 Masstree 会在键分片长度不足 8 字节时填充 0 进行处理,这就引入了额外的空间代价.平衡操作降低树高是一种很有效的方式,但在结构设计和代码实现方面难度较高.

3.4　优化内部遍历速度

对于树形索引来说,内部节点的遍历占了整个索引操作时间的大部分[26],因为从根部定位到叶子节点的过程中会涉及到大量的随机内存访问.降低树高能够有效减少随机内存访问的数量,除此之外,本节将继续探究另外一些优化内部遍历速度的技术.

3.4.1　降低搜索复杂度

Wormhole 通过将字典树、哈希表和 B+树结合的方式,降低了索引查询的复杂度,从而提升索引的性能. Wormhole 的设计主要分成两个部分:首先,Wormhole 将字典树与 B+树结合,把 B+树的内部节点替换成了一棵字典树,从而将索引的查询复杂度从 O(log N) 降到了 O(L)(其中 N 表示数据规模,L 表示数据长度);其次,Wormhole 通过将字典树存放在哈希表中的方式,进一步将查询复杂度降低到了 O(log L).

Wormhole 的第一个想法源于对 B+树索引的思考:B+树的内部节点仅仅被用于快速定位到存储数据的叶子节点,不会存放数据,因而可以使用更加高效的结构替代 B+树中的非叶子节点的结构. Wormhole 选择使用字典树作为替代(替换的字典树称为 MetaTrie),好处在于字典树的查询复杂度只取决于数据的长度,不取决于数据规模,因而索引整体的查询复杂度得到了降低.

但是仅仅使用字典树作为替代还不够,因为现实世界中的数据特别长时,O(L) 的时间复杂度也是不可接受的,于是 Wormhole 使用了哈希表来进一步降低索引搜索的复杂度.在 Wormhole 中,字典树的节点所代表的所有前缀都被存放在一个哈希表中(称为 MetaTrieHT),这样做的好处在于打破了字典树的层级式结构,使得索引搜索不再需要进行层级式的遍历,这一思想也被后续工作 CuckooTrie[15] 所采纳.

引入哈希表结构可以加速搜索过程. Wormhole 搜索过程实际上分成两个阶段,在第一个阶段中执行查询键与锚点键的最长前缀匹配过程,此过程中如果在未到达叶子节点前就发生了不匹配,则需要开始第二阶段:利用兄弟节点找到查询键可能存在的叶子节点,并在叶子节点中进行局部搜索返回最终结果.在 Wormhole 中,因为锚点键的所有前缀都存放在哈希表中,所以查询过程中可以使用任意长度的前缀进行匹配,以找到最长的前缀匹配.根据这一特点,Wormhole 使用类似二分查找的算法进行搜索,无需进行层级式的遍历.这种优化后的搜索算法将查询复杂度从 O(L) 降到了 O(log L),从而提升了索引的查询效率.

3.4.2　提升内存级并行度

现代处理器可以通过乱序执行的方式同时执行多条指令,因此多个内存访问以及计算操作可以并行执行.内存级并行度是指同一时刻并行执行的内存访问数目.对于内存数据库索引来说,内存访问造成的延迟要远高于处理器的计算延迟,因此在内存数据库索引的操作过程中,内存访问的耗时占据了单次操作的大部分时间[15].更高的内存级并行度意味着更多的内存访问延迟能够被其他计算操作所覆盖,所以提升索引的内存级并行度对索引性能有着重要影响.

CuckooTrie 使用哈希表和字典树的混合结构来提升索引的内存级并行度,其设计源自于对内存数据库索引的一项观察:现有的内存数据库索引基本都是层级式的索引结构,节点之间的依赖关系限制了性能发挥.具体来说,对于一个树形索引结构,想要获取下一层节点的信息就首先获取该层节点的信息,下层节点对上层节点存在着数据依赖,使得索引的内存访问只能以串行的方式进行.为了克服这一缺陷,CuckooTrie 使用了哈希表结构来打破节点之间的依赖关系,图 5 展示了如何将一棵基于指针的传统字典树转变为使用哈希表的 CuckooTrie.从图中可以看到,在 CuckooTrie 中,哈希表的每个表项是一个键值对(key-value pair);值(value)对应的是传统字典树中的一个节点,键(key)对应的则是字典树节点所代表的字符串前缀,称为节点名(node's name).在 CuckooTrie 的搜索过程中,对子节点的搜索不再依赖于父节点,比如搜索节点"ston"时,可以利用该前缀的哈希值在哈希表中找到对应的表项,无需通过节点"sto"的信息进行查找. CuckooTrie 利用了该种结构的特点,并使用缓存预取操作实现了更高的内存级并行度.

3.4.3　总　结

本节讨论了两种优化内部遍历速度的方法:降

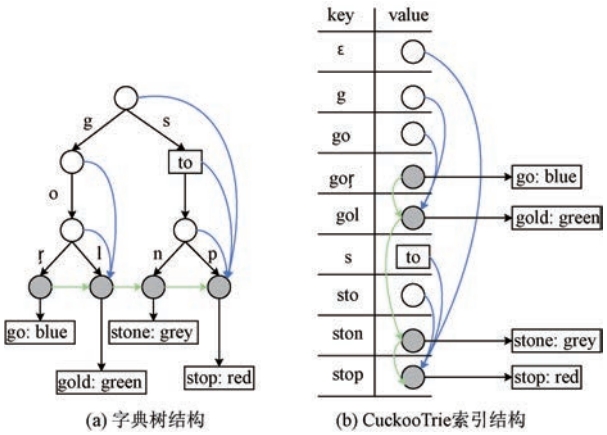

图 5 将字典树与哈希表结合得到 CuckooTrie

低搜索复杂度以及提升内存及并行度. Wormhole 引入哈希表,解除了字典树索引节点之间的依赖关系,降低了索引搜索的理论复杂度. 而 CuckooTrie 采用类似的思想,利用哈希表存储字典树,提升了索引搜索过程中的缓存效率,从而达到提升性能的目的.

3.5 优化范围查询性能

字典树索引的范围查询通常要差于 B+树索引[7],因此,优化字典树索引的范围查询性能十分必要. 一种优化方法是提升叶子节点容纳数据的能力,如第 3.3 节中提到的 Burst trie、HAT-trie 以及 ROART 等,从而支持叶子节点内的快速扫描. 但是,当叶子节点内扫描的结果满足不了查询条件时依旧需要通过回溯的方法寻找后继节点,因此范围查询性能的优化效果有限.

CuckooTrie[15]则采用了链表来提升范围查询性能,它的每个叶子节点存放有指向下一个叶子节点的信息,因此范围查询只需要定位到起始键,之后利用该信息向后进行搜索即可,无需进行回溯搜索. 但是,CuckooTrie 的叶子节点依然只存放一条数据,因此依然会存在大量的指针追逐现象.

Wormhole[13]和 HydraList[14]则完全采用 B+树的叶子节点结构,每个叶子节点存放多条数据,节点之间也通过指针进行连接,整个查询过程类似 B+树的范围查询,通过这种方式可以进一步提升索引的范围查询性能.

从上述讨论可以看出,优化类字典树索引范围查询的关键所在就是后继元素的查找方式,因此一种有效的方法就是使用类似 B+树的节点结构将叶子节点使用链表的方式连接起来. 这一点也在第 4 节的实验结果中得到了验证.

3.6 适配新型内存

非易失内存、分离式内存等新型内存技术的出现对类字典树索引的优化提出了新的挑战,因此近年来有较多的索引优化工作集中在这一方向. 本节将重点介绍针对非易失内存和分离式内存的类字典树索引优化技术.

3.6.1 非易失内存

非易失内存是近年来逐渐兴起的一种新型存储介质[40],其既具有传统磁盘持久性存储的特性,又有着与内存相同的字节寻址特性以及与内存十分相近的访问延迟,是一种十分理想的持久性存储介质. 和 DRAM 不同,非易失内存中存放的数据掉电不会丢失,这就要求索引保证在系统崩溃之后能够恢复到一个一致性状态,即保证索引的崩溃一致性(又称失败原子性). 如今的索引通常使用缓存行刷写指令(如 clwb、clflush)和内存屏障指令(mfence)来保证失败原子性,使用该方法引入的代价被称为持久化代价.

面向非易失内存的索引研究主要分为两个阶段,以 2019 年英特尔发布第一款商用非易失内存产品为界,前一阶段主要使用模拟器进行研究,后一阶段则是针对英特尔的傲腾持久性内存进行适配研究.

WOART[23]是第一个将 ART 移植到非易失内存的工作,其提供了一套新的范式保证 ART 在非易失内存上的失败原子性. 首先其保证节点分裂时的失败原子性,在节点发生分裂时,按照节点创建时间的反方向更新节点指针,确保分裂过程中的失败原子性. 其次,为了保证路径压缩过程中的失败原子性,其增加了节点信息,通过给定的公式检查索引是否发生了不一致现象. 最后,WOART 对 Node4 和 Node16 的结构进行了相应的修改,不再对节点中的数据分片进行排序操作,从而保证节点操作过程中的失败原子性.

ROART[24]是针对傲腾持久性内存产品进行设计优化的类字典树索引,其结构特点已经在第 3.3 节中进行了介绍,ROART 还针对非易失内存的特点提出了几项设计以减少索引操作的代价. 由于如今的计算机系统中,指针只有低 48 位是有效的[41],所以可以把一些信息编码到指针中优化索引的读写过程,ROART 在两个地方使用了这种方法. 首先是叶子节点,ROART 的叶子节点中存放的是指向数据的指针,这意味着在叶子节点中进行查询比较时依然需要通过指针访问原始数据,为了减少这一代

价,ROART 使用了指纹技术[42],对存储的数据进行哈希得到 16 比特的指纹,并将 16 比特的指纹嵌入到指向该数据的指针中,使用这一方法可以有效减少对非易失内存的访问.其次是内部节点,ROART 的内部节点依旧是 ART 的结构,如图 3 所示,该结构中 Node4、Node16 和 Node48 都将键分片和指针分别存放在两个数组中,ROART 则通过将键分片嵌入到孩子节点指针的方式减少了持久性代价.最后,ROART 还使用了选择性持久化的方式降低索引的持久化代价,节点锁、位图等元数据不会被显式地进行持久化.

PACTree[25]将 HydraList 移植到了傲腾持久性内存上,其结构与 HydraList 完全相同.为了适应非易失内存的特点,PACTree 对上层的 ART 进行了重新设计以实现无日志的持久化.

ERT[26]同样是面向傲腾持久性内存设计的类字典树索引,其结构为字典树与哈希表的结合,我们在 3.3 节中已经进行了相应的介绍.同样的,ERT 针对持久性内存也进行了相应的优化设计.首先,ERT 使用了类似 WOART 的机制,在头部增加深度信息以保证失败原子性;其次 ERT 的哈希表使用了类似 CCEH[32]的结构,将桶聚合到一个段中,从而降低了目录的大小,使得目录能够被保留在缓存中,降低索引操作对非易失内存的访问次数.

3.6.2　分离式内存

分离式内存系统是近几年兴起的一种新型内存系统,其目的是提升数据中心的内存资源利用率,同时避免内存容量不足造成的系统错误[33].该种系统的主要特点是将内存资源解耦合,将节点划分为内存节点和计算节点,节点之间通过新型高带宽低延时网络硬件(如 RDMA)进行连接通信,通过这种解耦合的方式,使用者可以使用大量的内存节点来构建一个拥有超大内存的内存系统,从而满足大规模数据的存储需求.

SMART[27]是一种面向分离式内存系统的类字典树索引,作者指出 ART 比 B+树更适合分离式内存系统,因为 ART 的节点规模更小,所以读写放大更小.SMART 主要解决了 ART 在分离式内存系统上的三个挑战.

首先,基于锁机制的并发控制在分离式内存系统上会严重限制索引的写入性能.为此 SMART 的内部节点实现了无锁机制,保证索引的写入性能.SMART 利用现代计算机系统中指针只有 48 位有效数据的特点[41],将键分片嵌入到指向孩子的指针中,从而实现了无锁并发机制.SMART 的叶子节点依然使用乐观并发控制,写者之间互斥,读者无需获取锁,通过校验和机制保证读者的正确性.

其次,跨节点的网络访问会导致冗余的读写 I/O.例如同一计算节点上的多个客户端可能会读取同一内存节点中的相同数据,但是每个客户端都会发送独立的 RDMA 读取指令,这就造成了冗余的数据读取,因为一份相同的数据可以被同一计算节点的客户端共享,SMART 提出了读委托(Read delegation)机制解决这一问题:当同一节点的多个客户端需要读取同一份数据时,只有第一个客户端需要去内存节点进行读取,其他客户端只需等待该客户端返回的结果即可.写入操作也有相似的问题,SMART 使用聚合写的方式减少冗余的写入操作,即在计算节点内维护聚合一个写缓存,对同一远程地址的更改会首先存储在本地写缓存中,之后通过聚合的方式一起写入到远程地址.

最后,由于网络访问的延迟要远高于内存访问,如今分离式内存上的索引都选择在节点本地保留缓存以加速索引操作,而 ART 的路径压缩和动态节点结构会对缓存设计造成挑战,因为插入数据时导致的路径解压缩和节点类型变动对计算节点的索引缓存并不可见,这就需要研究者设计一套新的缓存失效机制发现这种变化.在 SAMRT 中有三种缓存失效:第一种是由于父子节点之间的关系变化,比如在父节点中更新了某个孩子指针,这种更新操作无法被本地缓存探测到,所以 SMART 增加了一个反转指针变量,即在子节点中存放指向父亲节点的指针,用来探测这种缓存失效.第二种是由于节点类型变化导致的缓存失效,SMART 在节点头部增加了类型变量以检测这种失效.最后一种则是节点被删除导致的缓存失效,当内部节点被删除时,SMART 会将节点类型变为 0,叶子节点被删除则会将其置为无效节点,通过这种方式 SMART 能够发现由于节点被删除导致的缓存失效.

3.6.3　总　结

新型内存的引入要求类字典树索引进行特定的优化和适配.本节总结了一些面向新型内存的类字典树索引优化技术.在非易失内存场景中,索引优化的目标主要在于如何保证失败原子性的前提下尽可能减少持久化的代价;而在分离式内存场景中,索引优化的目标则在于如何减少网络通信的代价,尽可能地有效利用网络带宽.

4 性能评测及分析

为了进一步揭示和对比各种类字典树索引在实际负载下的性能，本节通过多个数据集和负载对Masstree、ART、HOT、Wormhole、HydraList 以及CuckooTrie 等进行较全面的实验性能评测. 我们选择这六种索引进行测评主要基于以下几点考虑：首先，Patricia 是一种经典的字典树索引结构，其思想被后续的索引广泛接纳，而 HOT 基于该索引进行了优化，所以不需要将其纳入比较中；其次，Burst trie 和 HAT-trie 的时间较为久远，且其所使用的技术也被 Wormhole 和 HydraList 所延续，所以我们将其排除在外；Kiss-Tree 只支持最长 4 字节的数据存储，并不具有实用性；FST 和 Hyperion 针对的是内存极端受限的场景，这两种索引牺牲性能换取内存效率，所以我们将这两种索引排除在性能评测之外；最后，由于面向新型存储场景的类字典树索引需要特殊设备，如非易失内存、RDMA 网卡，目前这些设备没有被广泛地应用于系统之中，所以本文不对这些索引进行评测，这些索引评测将留在后续工作中进行. 此外我们在实验中将内存 B+树索引和哈希索引也纳入了实验对比中，以更好地展示出类字典树索引的优点与缺陷.

4.1 实验配置

(1)运行环境. 所有实验在一台配备 256 GB 内存的 Linux 服务器上运行，具体硬件配置如表 3 所示，服务器有两个 socket 节点，两颗处理器，每颗处理器拥有 20 个物理核. 实验运行的系统版本为 Ubuntu 20.04.4，代码使用 g++9.4.0 编译，开启 -O3 优化选项. 所有索引使用 jemalloc 内存分配器分配内存，避免内存分配成为索引的性能瓶颈. 默认情况下所有索引运行在同一个节点，且内存也分配在同一个节点上，避免 NUMA 效应对索引性能的影响. 在探究 NUMA 效应对索引性能的影响时，我们将会使用到两个节点上的 CPU 和内存.

表 3 服务器配置

配置项	描 述
CPU	Intel Xeon Gold 6242 CPU Dual-socket with 40 cores at 3.1 GHz
L1 cache	32 KB iCache & 32 KB dCache (per core)
L2 cache	1 MB (per core)
L3 cache	36 MB (per socket)
内存	256 GB (2 (socket)×4 (channel)×32 GB)

(2)数据集. 实验使用六种数据集进行索引的性能评估，包括三份人工生成数据集 random8、random32 和 random128，以及三份真实数据集 reddit、titles 和 url，其中 random8 为随机生成的 64 位整型数据，random32 和 random128 为随机生成的最大长度分别为 32 和 128 的变长字符串数据，而 reddit 数据集包含了社交网站 reddit 上所有 2018 年的账号名称①，titles 数据集则是维基百科所有页面的标题数据②，urls 包含了 100M 网页信息③. 这些数据集已经被使用在以往多个类字典树索引的性能评测中. 表 4 给出了数据集的详细统计信息.

表 4 实验数据集

数据集	平均长度(字节)	最大长度(字节)	规模(百万)
random8	8	8	200
random32	18.1	32	200
random128	65.9	128	200
titles	19.5	255	41.4
reddit	10.9	32	71.6
urls	104.2	2048	105.9

(3)实验负载. 实验使用类似 YCSB[43] 的负载进行评测，每种负载使用不同的操作比例，实验负载的具体信息如表 5 所示. 所有的查询使用 Zipfian 0.99 分布.

表 5 实验负载信息

实验负载	描 述
只写	100% 插入
读写均衡	50% 插入，50% 查询
读密集	5% 插入，95% 查询
只读	100% 查询
范围查询	100% 范围查询
读-修改-写	50% 查询，50% 读-修改-写

实验对比的索引均使用其开源实现版本，其中 ART 包含其两种并发实现代码：ARTOLC 和 ARTROWEX④，我们使用了 GitHub 上以及相关论文中使用较多的 ARTOLC 进行评测；对于 Masstree，我们使用了其官方版本代码⑤，且将其设置为支持最长 4096 字节的数据. 对于 HOT⑥，我们选择其并发版本 HOTROWEX 进行评测. 对于 HydraList⑦，由于其开源代码仅支持最长 32 字节的数据插入，我们对其代码进行了有限的修改以使其能够支持更长

① https://files.pushshift.io/reddit/RA—2018—09.gz.
② https://dumps.wikimedia.org/enwiki.
③ https://law.di.unimi.it/webdata/uk—2007—05.
④ https://github.com/flode/ARTSynchronized.
⑤ https://github.com/kohler/masstree-beta.
⑥ https://github.com/speedskater/hot.
⑦ https://github.com/cosmoss-jigu/hydralist.

字符串数据的查询插入操作.对于 Wormhole①,其官方开源代码提供了三种不同类型的接口:whsafe、wormhole 以及 whunsafe,其中 whsafe 接口能够保证最强的线程安全性;wormhole 接口也保证线程安全,但是需要一些处理来避免死锁等特殊情况的发生,其性能优于 whsafe 接口;whunsafe 接口则不保证线程安全.为了在保证索引的正确性的同时不会损失过多的性能,我们选择 wormhole 接口进行评测.对于 CuckooTrie②,由于其结构本质上是一个静态哈希表,且其开源代码并不支持哈希表的扩容,所以我们将哈希表的大小设置为构建索引的数据规模的五倍,以确保其能够正确运行.此外,在对比的这些索引中,除了 Masstree 可以直接使用 char * 类型的字符串进行操作之外,其他索引均需要构建特定的数据格式以进行操作,为了尽可能保证实验的公平性,实验过程中会首先构建出符合索引要求的键数组,然后进行评测.由于在实验过程中我们发现 HOT 和 HydraList 无法保证 255 字节以上字符串的插入正确性,所以在 urls 数据集的实验中我们将排除这两种索引.对于对比使用的 B+树索引,我们选择使用 BTreeOLC③,该版本使用 OLC 并发机制,并在众多后续工作中被用作对比对象[7,14,35].对于哈希索引,我们选择被广泛使用的 Libcuckoo④ 作为对比对象[44,45],该种哈希索引使用细粒度锁机制实现并发控制,由于哈希索引并不支持范围查询操作,所以在测试范围查询性能时我们将排除该索引.

(4)性能指标.实验主要考查以下的指标:

①读写吞吐.吞吐量指的是索引在单位时间完成的操作数,它是衡量索引性能的重要指标.在实验中我们将评测索引在不同数据集和不同负载下的吞吐量.

②尾延迟.尾延迟反映了索引的稳定性,而索引的稳定性会影响到数据库系统查询的稳定性[46,47].在实验中我们将评测索引在纯读负载、范围查询负载以及纯写负载下的尾延迟.

③扩展性.为了达到更高的并行度,现代计算机往往会配备多个 NUMA 节点以容纳更多的处理器和内存[48,49],而跨节点内存访问的代价要高于本地内存的访问,探究 NUMA 效应对索引性能的影响能够更好地展示出索引的扩展性.在实验中我们将比较索引跨 NUMA 节点访问时的读写性能.

④内存效率.索引结构占用的内存大小是内存数据库索引的重要评判标准,如何达到内存效率与时间性能的平衡也是众多内存数据库索引一直在探究的方向[50].在实验中我们将对比索引在不同数据集下的内存效率.

4.2　实验结果及分析

4.2.1　吞吐对比

在该实验中,我们首先插入一半的数据构建索引,之后使用不同负载测试索引的吞吐.为了避免 NUMA 效应对实验结果的影响,所有线程和内存都将被分配在同一个节点上,由于使用的 CPU 为 20 核 40 线程,所以当线程数超过 20 时将使用超线程进行实验.图 6～图 11 展示了这八种索引在不同数据集、不同负载以及不同线程数下的吞吐性能,从图中可以观察到,没有索引能够在所有数据集以及所有负载下都明显超越其他索引.下面按照负载类型分别进行分析.

(1)只写负载.图 6～图 11(f)展示了索引在只写负载下的吞吐.在三个随机数据集下,ARTOLC 的插入性能要远高于其他类字典树索引,这是因为 ART 是纯字典树结构,而其他类字典树索引则混合了额外的数据结构(比如链表、哈希表),这使得 ART 的插入过程相较于其他索引结构而言相对简单.此外,随机数据集的数据分布较均匀,使得 ART 的结构十分扁平,这也使得 ART 在随机数据集下呈现了高的插入性能.但在三个真实数据集下,ART 的插入性能并没有取得太大的优势,尤其在 url 数据集下,ART 的插入性能甚至弱于 Masstree.这是因为真实数据集的分布往往更加稀疏,且共同前缀较少,使得 ART 的高度增加,产生大量的小结点,影响了查询性能.Wormhole 的并发插入性能要明显弱于其他索引,且随着线程数的增加出现了性能下降,这说明 Wormhole 的并发控制性能较差.HashTable 在随机数据集下插入性能较差,但是在真实数据集下性能远高于其他索引,这是因为随机数据集的分布较为均匀,哈希冲突较多,这意味着 HashTable 在插入过程中会产生了大量的锁竞争,从而导致索引性能不佳,但是真实数据集的分布较为稀疏,哈希冲突较少,所以 HashTable 在真实数据集下的性能表现较好.

(2)只读负载.图 6～图 11(c)展示了所有索引在只读负载下的吞吐.在随机数据集下,ARTOLC 都展示出了优异的性能,因为随机数据集的分布

① https://github.com/wuxb45/wormhole.
② https://github.com/cuckoo-trie/cuckoo-trie-code.
③ https://github.com/wangziqi2016/index-microbench.
④ https://github.com/efficient/libcuckoo.

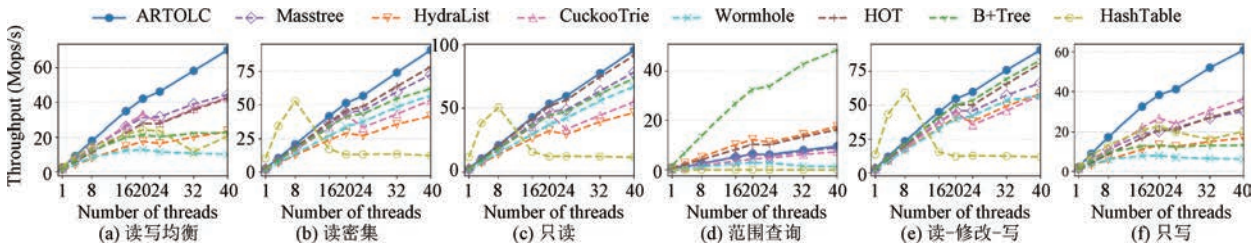

图 6 random8 数据集索引吞吐

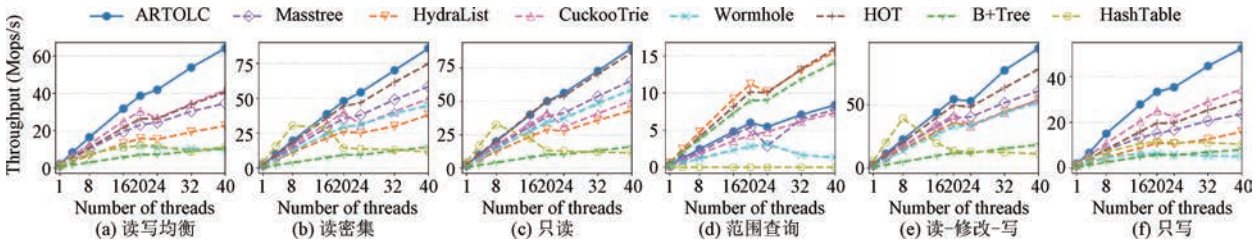

图 7 random32 数据集索引吞吐

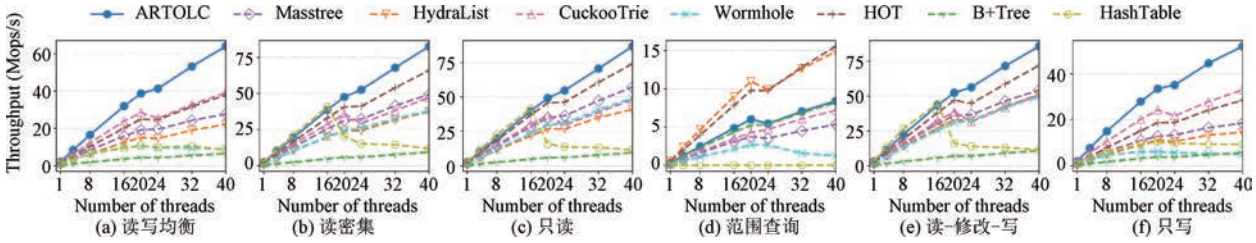

图 8 random128 数据集索引吞吐

较为集中，且数据中存在较多的共同前缀，使得 ART 能够形成扁平的索引结构，从而获得优异的查询性能. 但在真实数据集下，ART 的出现了较为严重的性能下降，这是因为真实数据集的分布较为稀疏，使得索引产生了大量扇出更小且性能更差的节点，从而影响了索引的查询性能. 相比之下，HOT 的性能要稳定许多，这得益于其动态自适应的节点结构. HOT 在各种数据集下都能保证稳定的索引结构，进而获得稳定的性能. 此外，在 url 数据集下，Wormhole 的查询性能要远高于其他索引，这是因为 url 数据集的键平均长度较长，其他索引无法进行有效的前缀压缩，而 Wormhole 使用锚点键索引下层节点，能够降低键长度，从而提升索引的查询性能. 值得注意的是 HashTable 在 8 线程或者 16 线线程时就达到了最高性能，之后发生了性能下降，这与 HydraList[14] 论文中的测试结果一致，主要原因是 HashTable 使用了锁机制实现并发，而实验中的查询操作遵从 zipfian 0.99 分布，于是查询过程中产生了大量的锁竞争，这导致 HashTable 的性能下降，我们在第 4.2.5 节中对不同倾斜程度的查询负载进行了测试，实验结果也证明了这一结论.

读写均衡负载：图 6～图 11(a)展示了索引在读写均衡负载下的吞吐. 由于该负载包含了 50%的插入操作，所以各个索引的性能变化趋势与只写负载相似.

读密集负载：图 6～图 11(b)展示了索引在读密集负载下的吞吐. 该负载中 95%的操作为查询，所以各个索引的性能变化趋势与只读负载相似.

范围查询负载：图 6～图 11(d)展示了索引的范围查询性能. HydraList 的范围查询性能始终优于除了 B+树的其他索引，这得益于 HydraList 链表式的叶子节点结构，使得 HydraList 能够很轻易地进行扫描操作. 除了 Wormhole 的其他索引均需要进行跨层级的遍历才能进行有序的扫描. 尽管 Wormhole 的叶子节点结构同样是链表，但它在扫描节点时会对节点进行加锁，因此限制了索引的扩展性. HOT 的范围查询性能略弱于 HydraList，在 random128 数据集下与 HydraList 相当，这得益于 HOT 提出的平衡操作. 这种平衡操作使得 HOT 能够形成更稳定的结构和更低的树高，使得范围查询的遍历操作无需跨越过多的层级，从而获得更好的性能. 在 url 数据集下，由于数据的平均长度较长，

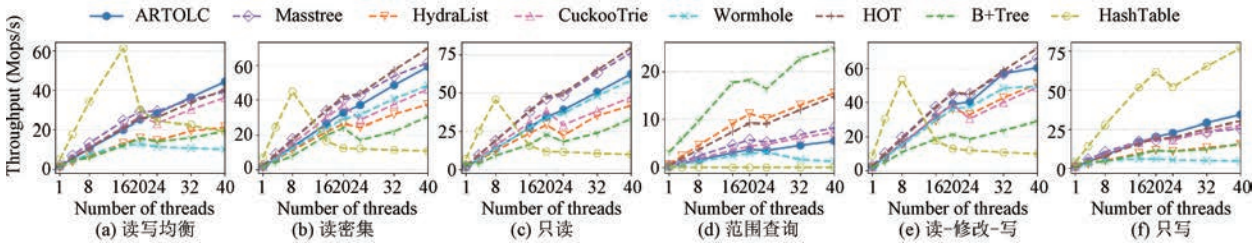

图 9　reddit 数据集索引吞吐

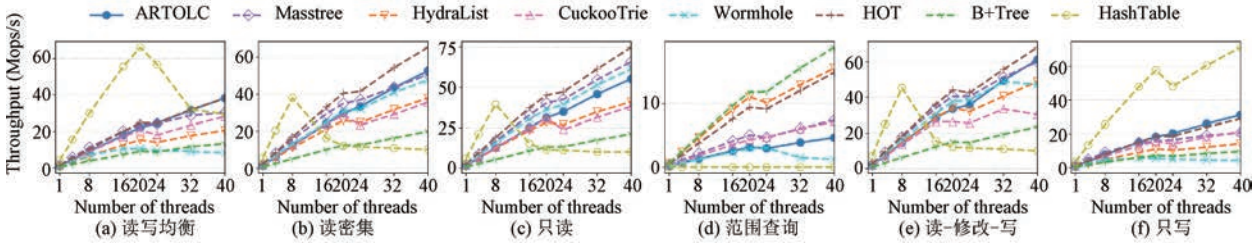

图 10　titles 数据集索引吞吐

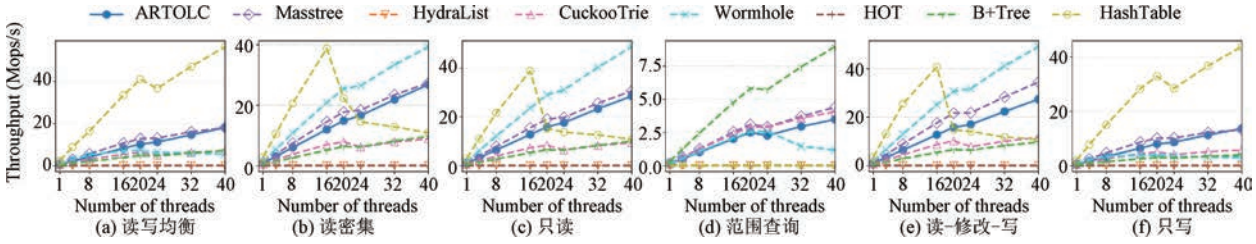

图 11　urls 数据集索引吞吐

所以各个索引的范围查询性能都受到了较大的影响. CuckooTrie 由于使用了链表式的叶子节点结构,所以在该数据集下展现出了较好的性能. B+树的范围查询性能在数值型数据要远高于其他类字典树索引,这证明了 B+树在范围查询负载下的优势;而当数据长度增加时,B+树的范围查询性能逐渐下降,在 random32 和 random128 数据集下都弱于 HydraList 和 HOT,这是因为数据长度的增加使得 B+树只能使用指针间接存储数据,额外的指针访问导致了性能的下降.

读一修改一写负载:图 6～11(e) 展示了索引在读一修改一写负载下的吞吐. 由于该负载不包含插入操作,所以各个索引的性能表现与只读负载相似.

当线程数从 20 增加到 24 时,所有的索引都显示出了扩展性变差的现象,其中 CuckooTrie 甚至出现了性能下降的情况. 这主要由于是超线程的影响. 在吞吐实验中我们只使用了一个 CPU,该 CPU 有 20 个物理核,当线程数超过 20 时就会开启超线程. 一方面,超线程时每个线程的计算能力要更弱;另一方面,超线程会对 CPU 核的私有缓存命中率造成影响,由于每个物理核运行了两个线程,但是只有一块私有缓存,所以运行在同一个物理核的两个线程会互相影响缓存效率. 因此,索引扩展性在 20 线程之后出现了下降的现象. 由于 CuckooTrie 会使用大量的预取操作,缓存命中率下降会对性能造成非常大的影响,所以超线程时出现了性能下降的现象.

从吞吐实验结果可以看出,在不同的数据集和不同的负载下,各个索引都会有不同的性能表现. 总结来说,ART 结构比较适合稀疏的随机数据,但是其无法适应真实的变长数据,且其范围查询性能有很大的局限性;HydraList 的范围查询性能始终优于除 B+树的其他索引,但是其读写性能偏弱,且代码实现不完备;Masstree 的读写性能都比较稳定,且代码十分完备,能够支持多种应用需求;Wormhole 的并发性能有很大的局限性,不适合有高并发性能写入需求的系统;CuckooTrie 在随机数据集下有可观的性能,但是在真实数据集下,其性能并未表现出优势,而且由于其使用了大量的预取操作,其性能很大程度上会受到内存带宽的影响,此外,因为该索引本质上是一张静态哈希表,所以在实际应用中有很大的局限性;最后,得益于索引引入平衡操作的

设计，HOT 在各个数据集和负载下都表现得十分稳定，且性能十分优秀，但是现在的版本并不支持 255 字节以上的数据操作. 综上所述，当应用场景为数值型数据且大部分查询为点查询操作时，ART 索引是较好的选择. 但对于宽泛的应用场景，如果数据长度有限，我们推荐使用 HOT 索引，若数据长度不限，则 Masstree 索引更为合适.

4.2.2 尾延迟

尾延迟是衡量索引性能的一个重要指标，它关系到整个系统的稳定性. 本节对索引在真实数据集 reddit 和 titles 下的尾延迟进行了评测. 实验测试了索引在单线程和 20 线程下的读延迟和写延迟，选择这种线程设置的原因在于：使用单线程可以尽可能保证索引的性能不受到其他因素的影响；而单颗 CPU 的物理核数为 20，设置成 20 线程可以避免 NUMA 效应和超线程对索引的性能影响. 读延迟的评测使用了随机分布的查询键，以减少缓存带来的影响.

从图 12(a)和图 13(a)可以看出，除 HydraList 外，其他索引的查询延迟基本相当. HydraList 在 99.99%情况下要明显高于其他索引，这是因为 HydraList 使用了特殊的叶子节点结构. 在 HydraList 的查询过程中，索引需要通过上层的字典树索引才能定位下层的叶子节点，之后还需要在叶子节点中进行搜索才能找到最终结果. 在特殊情况下，HydraList 需要搜索的范围将超过一个节点，故其尾延时较高. 从图 12(d)和图 13(d)可以看出，在 20 线程下，除了 HashTable 外的其他索引查询延迟都有所降低，而 HydraList 的查询延迟也降低至其他索引的同一水平. HashTable 查询尾延迟的突然升高源于多线程的锁竞争，这与吞吐测试结果基本一致.

图 12(b)和图 13(b)展示了索引范围查询的尾延迟. 可以看出，与吞吐性能不同的是，HydraList 的 99.9%尾延迟要明显高于其他索引，而 B+树的范围查询尾延迟则始终保持最低，这展示了 B+树在范围查询负载下的巨大优势.

从图 12(c)和图 13(c)可以看出，HydraList 和 Wormhole 的 99%尾延迟要明显高于其他索引，这是因为这两种索引的插入可能会导致叶子节点的分裂，导致索引操作的延迟上升；ARTOLC 的 99.99%延迟明显升高，这可能是由于节点的类型转换；而 HOT、CuckooTrie 以及 HashTable 的尾延迟则一直比较稳定，这是因为 HOT 一直在进行规模较小的平衡操作，使得索引结构较为稳定，CuckooTrie 的结构为一个静态哈希表，不会发生大规模的结构变化，而 HashTable 在真实数据集下哈希冲突较少，所以延迟较低. 而图 12(f)和图 13(f)可以观察到，在 20 线程下，Wormhole 的 99%延迟要明显高于单线程，这是因为 Wormhole 特殊的上层结构和并发控制机制的设计问题，在索引的叶子节点发生分裂进而导致上层结构的更改时，Wormhole 需要锁住整个哈希表，导致多线程插入时操作延迟的明显升高.

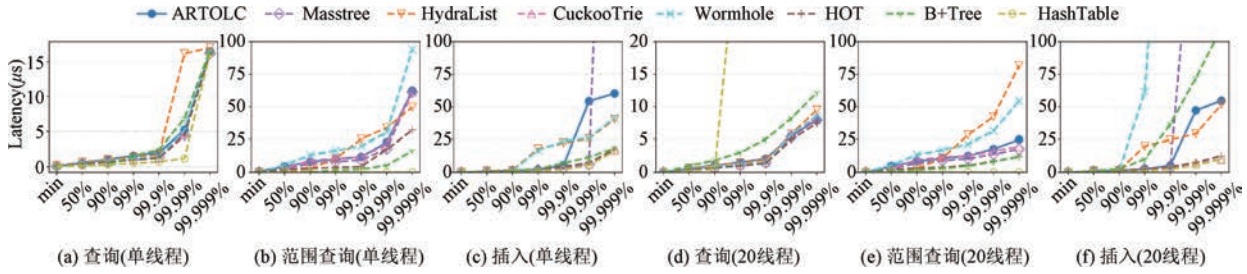

图 12 reddit 数据集索引尾延迟

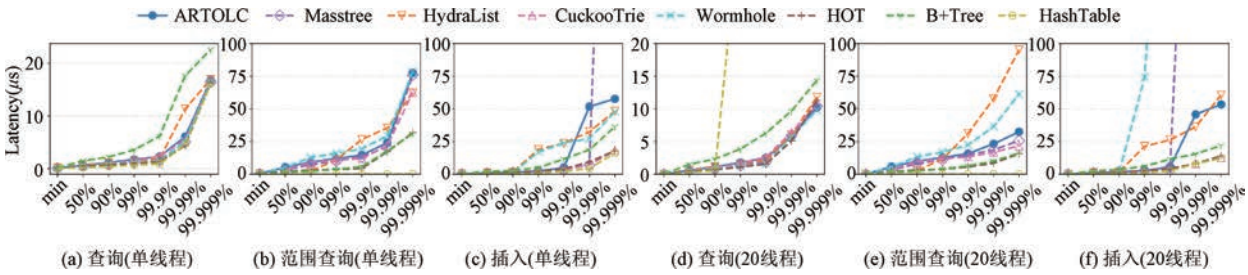

图 13 titles 数据集索引尾延迟

综上所述,HydraList 的尾延迟始终比较高,这表明其稳定性不佳,Wormhole 在高线程竞争环境下的尾延迟会显著提升,HashTable 在多线程查询时由于锁的原因尾延迟明显升高,B+树的范围查询延迟优秀,但是其他负载下的尾延迟表现不佳,而 HOT 和 CuckooTrie 则展现出了很好的稳定性.

4.2.3　扩展性

本节实验主要探究 NUMA 效应对索引性能的影响.实验使用了两个节点中的两个处理器.为了更清晰地展示 NUMA 效应对性能的影响,我们将不同的内存分配方式和线程分配方式进行组合.具体来说,内存分配有两种模式:单一节点分配和多节点交织分配;线程分配方式也有两种:固定在单一节点和分布在多节点.其中多节点交织分配内存的含义是使用轮询(round-robin)的方式在不同的节点中进行内存分配.此时,内存会随机分配在指定的 NUMA 节点中,这可以通过 Linux 系统的 numactl 指令完成设置.在实验中,我们将使用以下几种设置进行评测:

(1)S1.内存分配固定在 0 号节点上,线程分布在另一个节点上,线程超过处理器的物理核数时使用超线程.

(2)S2.内存分配固定在 0 号节点上,线程数小于处理器的物理核数时使用 0 号节点的处理器,超出物理核数时将超出的线程固定在另一个节点的处理器上.

(3)S3.内存使用交织分配方式,线程固定在 0 号节点上.

(4)S4.内存使用交织分配方式,线程分布与 S2 配置相同.

实验中我们将使用 titles 数据集测试索引 1—40 线程的性能,由于 HydraList 针对 NUMA 效应进行了特定的优化,在实验中我们也将使用该设置,并比较该种设置在不同内存分配方式下产生的实际效果.

图 14 展示了在不同设置下索引的性能表现,可以看到在 S1 设置下,因为所有的内存都分配在远程节点,内存访问几乎全都是远程访问,所以索引性能相比其他设置下产生了明显的下降,由此也可以看出远程内存访问对性能的影响.

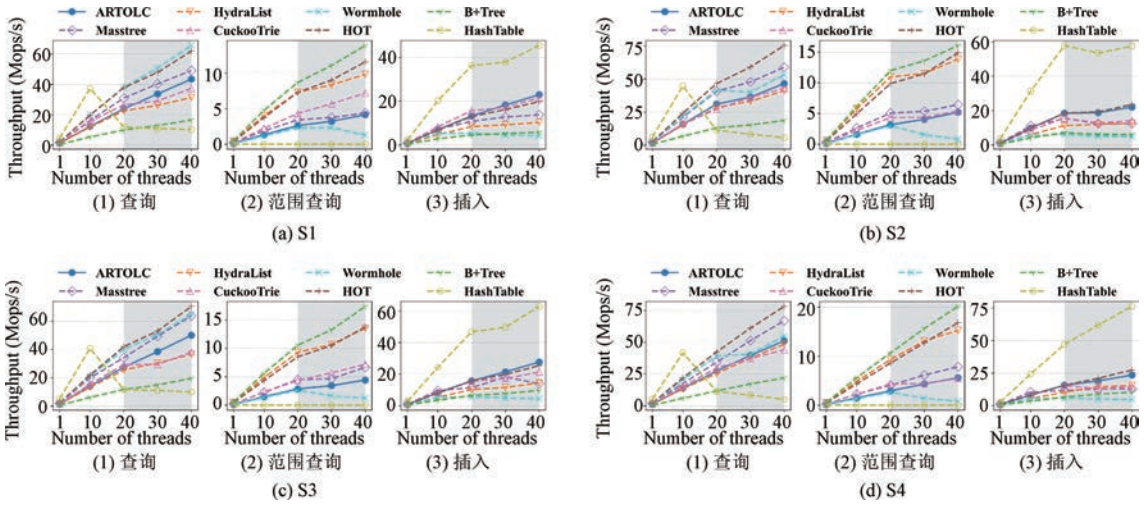

图 14　NUMA 效应对索引性能的影响

在 S2 设置下,可以看到线程数小于 20 时所有索引都展现出了良好的扩展性,而当线程数超过 20 时,由于有一半的线程被分配到远程节点上,所以索引的性能提升困难,甚至出现了性能下降的现象.

在 S3 设置下,由于内存采用交织分配的方式,但是线程全部分布在 0 号节点,所以程序有一定的概率进行远程内存访问,可以看到该设置下的索引性能相较于 S2 设置出现了明显的下降.

在 S4 设置下,可以看到几乎所有索引在查询负载下都展现出了十分良好的扩展性,即使线程被分配在两个节点之上,这是受到内存分配模式和 Linux 系统的 AutoNUMA[51] 设置的影响.因为内存随机分配在不同 NUMA 节点中,所以查询操作就会有更大的概率访问本地节点,同时又因为 Linux 默认的 AutoNUMA 设置会倾向于将内存迁移到对其访问最频繁的节点上,进一步增加了本地访问的概率.由此可以看出,使用一些合适的系统设置就可以使得索引取得良好的扩展性.

从实验结果可以看出,NUMA 效应会对内存数据库索引的读写性能产生较大的影响,但是通过

使用合适的内存分配机制和操作系统的 NUMA 策略可以很大程度上缓解 NUMA 效应对索引性能的影响，并获取良好的扩展性，类似 HydraList 这样对 NUMA 架构进行专门优化的内存数据库索引并没有想象中的那样有效. 由此可以得出结论：在现代操作系统已经对 NUMA 效应进行适配的前提下，对内存数据库索引专门进行 NUMA 架构的适配没有特别大的必要性.

4.2.4 内存效率

在本实验中，我们比较各个索引在不同数据集下的内存效率. 我们将把数据集所包含的全部数据插入索引中，之后统计索引所占的内存大小. 由于 CuckooTrie 实际上是一个静态哈希表，且开源代码并不支持哈希表的扩容，所以在本实验中不展示 CuckooTrie 的结果.

图 15 展示了各种索引在不同数据集下的内存消耗量. 为了更清晰地展示出不同索引的内存消耗区别，我们将图例上限设为 40 GB，超出 40 GB 的内存消耗我们使用倍数进行显示. 图中的 1.1x 表示内存消耗为(1.1×40) GB. 可以看到，在不同数据集下 HOT 的内存消耗都是最少的，这得益于 HOT 空间高效的节点结构设计和稳定的树高. ART 在随机数据集下内存消耗基本相同，并没有因为数据长度的增加而升高，这也反映了随机数据集下 ART 的稳定结构. Wormhole 在三个真实数据集下的内存效率与 ART 相当或者优于 ART，在三个随机数据集下的内存效率则要弱于 ART. Masstree 的内存消耗随着数据长度增加而增长，在多数数据集下仅次于 HydraList 和 B+树，因为其拥有与 B+树十分相似的节点结构，会造成大量的空间浪费. 由于 HydraList 在代码实现中固定了键的长度，且没有在内存效率上进行特定的优化，所以其内存效率在类字典树索引中最差. B+树索引的内存效率随着键长度的增加而降低，且在键较长时其内存效率较差.

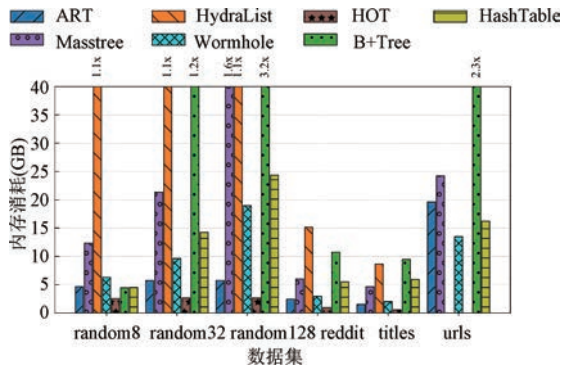

图 15 内存消耗对比

从实验结果可以看出，HOT 的内存效率最高，如果研究者需要空间高效的索引结构，HOT 将是一个很好的选择.

4.2.5 其他负载测试

本节将对比类字典树索引在一些其他负载下的性能，以进一步探究这些索引在各种应用场景下的表现. 实验使用 titles 数据集并在 40 线程下进行测试，所有的内存和线程都分布在同一个节点中以避免 NUMA 效应对性能的影响.

(1)倾斜负载. 本实验使用纯更新负载进行测试，倾斜参数设置为 0.1 到 0.99，实验结果如图 16(a)所示. 当倾斜程度小于 0.9 时，所有索引的性能都比较平稳；当倾斜程度从 0.8 增加到 0.9 时，HashTable 的性能出现了剧烈下降，其他索引性能变化依然较小；而从 0.9 增加到 0.99 时，HashTable 的性能进一步下降，除了 B+树和 Hydralist 的其他索引也出现了性能下降. 由于 HashTable 使用的是细粒度锁，所以在负载倾斜度较小时其性能非常优异，但是当倾斜程度非常高时，其性能发生了剧烈抖动，这一现象充分说明了并发机制对索引性能的影响. 使用了乐观并发控制的索引尽管在偏斜度更高的负载下发生了性能下降，但是下降幅度远远小于使用锁机制的 HashTable.

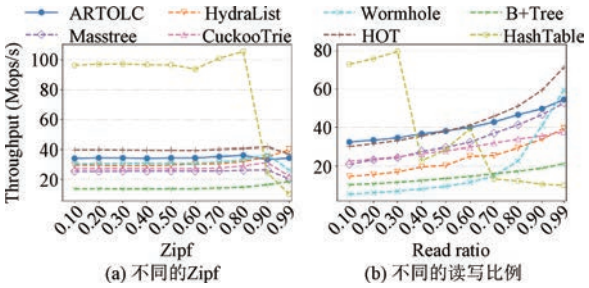

图 16 不同倾斜度和读写比例的负载

读写混合负载：本实验将探究不同读写比例对索引性能的影响. 此处我们将查询操作与插入操作进行混合，读操作比例从 0.1 增加到 0.99，读操作遵从 Zipfian 0.99 分布，观察索引的性能变化. 从图 16(b)中可以看到，随着读操作比例的增加，除了 HashTable 之外的所有索引的性能都在上升，其中 Wormhole 上升幅度最为明显，从最差提升到了第二，这说明 Wormhole 的多线程插入性能非常差. 对于 HashTable，由于 titles 数据集的分布较为稀疏，进行插入操作时产生的哈希冲突较少，所以其插入性能十分优秀，但是当查询操作较多时，索引性能产生了明显的下降，这是因为读操作都是倾斜分布，而 Hash-

Table 使用细粒度锁进行并发控制,于是读操作之间发生了大量的线程竞争,从而导致索引性能的下降.

4.3 实验总结与思考

(1)没有一种类字典树索引能适应所有负载.

表 6 总结了在不同数据集不同评价指标下表现最佳的索引,从表中可以看出,没有一种类字典树索引能够适应所有负载. ART 在随机数据集和数值型数据集下保持了十分优秀的读写性能,因此十分适用于这两种应用场景. HOT 无论在随机数据集还是在真实数据集下都保持了十分稳定且十分优秀的性能表现和稳定性,但是其只支持最长 255 字节的数据操作,因此在长度较短的数据集下使用 HOT 是一个比较好的选择. HydraList 的范围查询性能十分优秀,但是在其他负载下的性能和内存效率都十分不理想,且性能稳定性较差,因此不适合真实系统的使用.

表 6　实验结果总结:在不同负载下表现最好的索引

数据集 / 性能指标	random8	random32	random128	reddit	titles	urls
点查询	ART	ART	ART	HOT	HOT	Wormhole
插入	ART	ART	ART	HashTable	HashTable	HashTable
范围查询	B+Tree	HydraList	HydraList	B+Tree	B+Tree	B+Tree
内存效率	HOT	HOT	HOT	HOT	HOT	Wormhole

(2)索引结构越简洁,插入性能越高.

从只写负载的性能表现可以看出,ART 的写入性能始终优于除 HashTable 以外的其他索引,尽管在真实数据集下其性能下降明显,但还是优于大多数其他索引. 这一现象表明简洁的结构有利于插入性能的提升. 这是因为插入过程通常会导致大量的节点结构改变,简洁结构的更改所涉及的节点范围更小,而复杂结构的整个更改过程会更复杂,从而导致插入性能下降.

(3)使用链表式的叶子节点能够提升索引的范围查询性能.

HydraList 的范围查询在大部分情况下均优于除了 B+树的其他索引,而 B+树在真实数据集下取得了最优范围查询性能,这展示出了链表式的叶子节点结构在范围查询负载下的优势. 因此,如果想获得优秀的范围查询性能,需要使用链表式的叶子节点结构,避免跨层级的节点扫描.

(4)平衡的数据结构能够获得提升索引性能的稳定性.

红黑树、AVL 树、B+树等传统的数据结构已经证明了平衡操作是取得稳定性能的关键所在,HOT 的读写性能表现进一步证明了平衡操作的重要性. 因此如果想要获得在各种数据集各种负载下都能保持性能稳定的字典树索引,必须将平衡操作引入索引的构建过程中,使得整个索引结构保持稳定.

(5)特意适配 NUMA 架构的内存数据库索引设计重要性不高.

现代操作系统已经对 NUMA 架构进行了很好的适配[51],特意设计针对 NUMA 架构进行优化的内存数据库索引可能并不会取得预想中的效果,因此在设计内存数据库索引的过程中对 NUMA 效应的考虑可能并不是那么重要.

(6)字典树结构适合使用乐观并发控制,不合适的并发机制会使索引的性能受到极大的影响.

因为字典树的节点规模通常较小,线程之间的竞争粒度较小,所以适合使用乐观并发控制. 本文中测评的绝大部分类字典树索引均使用了乐观并发控制机制,并且取得了良好的并发性能,而未使用该机制的 Wormhole 和 HashTable 在高竞争负载下性能表现较差. 由此也可以看出并发控制机制对内存数据库索引的性能具有较大的影响.

5 研究展望与未来趋势

随着内存数据库的发展,类字典树索引也越来越受到关注. 对于未来的类字典树索引发展趋势,我们认为以下几个方面是值得关注的方向.

5.1 面向非易失内存的类字典树索引

非易失内存是近年来逐渐兴起的一种新型存储介质[40]. 它既具有传统磁盘持久性存储的特性,又有着与内存相同的字节寻址特性以及与内存十分相近的访问延迟,是一种较为理想的持久性存储介质. 有关非易失内存的索引研究已经持续多年,主要集中在 B+树索引研究[42,47,52-59],其中也有部分针对字典树索引的研究(见第 3.6.1 节). 尽管 2022 年英特尔宣布终止生产傲腾持久性内存,但是非易失内存技术依旧吸引着研究者,得益于持久性存储、低功

耗等特点，非易失内存在物联网等领域依旧存在着十分广阔的应用前景. 除了已经停产的傲腾持久性内存使用的 3D Xpoint 技术，目前还有其他研究中的非易失内存技术，如电阻式随机存取存储器(ReRAM)、磁性随机存储器(MRAM)、铁电随机存取存储器(FRAM)和相变存储器(PCM)等，有报告称 MRAM 的总收入到 2033 年预计可以达到 983 亿美元[60]，这意味着未来会有更多的非易失内存设备被应用到实际生产环境中，因此面向非易失内存的索引设计依旧存在必要性. 但是目前面向非易失内存的索引设计大多依赖于英特尔傲腾持久性内存的硬件特性(例如 256 字节的存取单位)，而未来的非易失内存产品和傲腾持久性内存可能存在硬件特性上的差异，比如访问粒度、存储密度、读写延迟差距等. 因此，在未来的索引设计中，一方面要考虑一些普适性的优化策略，针对非易失内存产品的持久存储、字节寻址等共同的设备特点提出一些非针对性的优化方法；另一方面，在面向实际的产品设备时，也要针对产品的硬件特性调整优化策略.

5.2 类字典树索引与学习索引的结合

学习索引是一种新型索引结构，也是将机器学习技术应用于数据库优化的典型代表. 学习索引的概念在 2018 年 SIGMOD 上被 Tim Kraska 等人正式提出[61]. 学习索引的提出来自于一项观察：数据库中常用的 B+树索引与机器学习模型具有相似的特征，使用机器学习模型能够实现与 B+树相同的功能. 学习索引创造性地将机器学习技术与数据库索引结合在一起，取得了良好的查询性能和内存效率，并引发了新一轮数据库索引领域的研究热潮. 近几年来出现了大量工作聚焦于机器学习模型与各种数据库索引的结合，包括主索引、辅助索引、多维索引以及布隆过滤器等. 这些工作主要从模型拟合、数据插入处理以及并发控制机制等方面对学习索引进行了全方面的研究[62-68]. 但是，如今的学习索引还未解决变长数据的索引问题，目前仅有 RSS(Radix-StringSpline)[69]、SIndex[70]和 SIA[71]三份工作对变长数据索引问题进行了研究. 其中 RSS 使用了字典树与学习索引结合的结构，但是其并不支持数据插入；Sindex 则性能表现不佳，无法满足变长数据索引的性能需求；SIA 则针对 Sindex 进行了优化，Sindex 将变长字符串视为高维的向量数据，模型的拟合和预测通过矩阵运算进行，SIA 通过对矩阵运算算法的优化，减少了模型训练的代价，从而提升索引的插入性能，但是 SIA 并没有减少模型预测过程中的矩阵运算代价，因而其查询性能无法超越传统的类字典树索引. 因此，能否将字典树与学习索引进行更加紧密的结合，解决索引的插入和并发问题，以获得性能更优秀、稳定性更强且空间代价更低的变长数据索引，该领域还需要大量的探索工作.

5.3 面向分离式内存的字典树索引

分离式内存系统是近几年兴起的一种新型存储系统，近年来已经有若干在分离式内存上利用索引构建存储系统的工作[72-81]，这些工作使用的索引包含 B+树[72,73,75,77,80]、哈希表[76,81]、Masstree[74]以及学习索引[75,78,79]等. 目前该方向工作较少，且以 B+树索引的研究为主，我们在第 3.6.2 节中对面向分离式内存系统的类字典树索引 SMART 进行了介绍，随着分离式内存系统的持续发展，能否利用字典树索引构建出高性能的分离式内存系统，该领域仍然有很大的探索空间.

6 结束语

字典树作为一种经典的数据结构，经过多年的发展，已经日渐成熟. 因为其高效的查询性能和内存效率，字典树索引及其变种结构受到了数据库领域越来越多的关注. 本论文综述了类字典树索引的研究进展，给出了一种新的分类方法并对类字典树索引进行了分类和讨论，从而为读者提供一个类字典树索引的完整视角. 本文还对六种类字典树索引进行了性能评测，给出了较为详尽的对比测试结果和分析比较，为后续相关研究提供了参考. 最后，论文对类字典树索引的未来研究方向进行了展望.

参 考 文 献

[1] Faerber F, Kemper A, Larson P Å, et al. Main memory database systems. Foundations and Trends® in Databases, 2017, 8(1-2):1-13

[2] Richter S, Alvarez V, Dittrich J. A seven-dimensional analysis of hashing methods and its implications on query processing. Proceedings of the VLDB Endowment, 2015, 9(3): 96-107

[3] Lehman T J, Carey M J. A study of index structures for main memory database management systems//Proceedings of the 12th International Conference on Very Large Data Bases. San Francisco, USA, 1986: 294-303

[4] Rao J, Ross K A. Making b+− trees cache conscious in main memory //Proceedings of the 2000 International Conference on Management of Data. Dallas, USA, 2000: 475-486

[5] Chen S, Gibbons P B, Mowry T C. Improving index performance through prefetching//Proceedings of the 2001 International Conference on Management of Data. Santa Barbara, USA, 2001:235-246

[6] Levandoski J J, Lomet D B, Sengupta S. The bw-tree: A b-tree for new hardware platforms//Proceedings of the 29th International Conference on Data Engineering. Brisbane, Australia, 2013: 302-313

[7] Wang Z, Pavlo A, Lim H, et al. Building a bw-tree takes more than just buzz words//Proceedings of the 2018 International Conference on Management of Data. Houston, USA, 2018: 473-488

[8] Zhang H, Liu X, Andersen D G, et al. Order-preserving key compression for in-memory search trees//Proceedings of the 2020 International Conference on Management of Data. Portland, USA, 2020: 1601-1615

[9] De La Briandais R. File searching using variable length keys//Proceedings of the 1959 Western Joint Computer Conference. San Francisco, USA, 1959: 295-298

[10] Mao Y, Kohler E, Morris R T. Cache craftiness for fast multicore key-value storage//Proceedings of the 7th ACM European Conference on Computer Systems. Bern, Switzerland, 2012: 183-196

[11] Leis V, Kemper A, Neumann T. The adaptive radix tree: Artful indexing for main-memory databases//Proceedings of the 29th International Conference on Data Engineering. Brisbane, Australia, 2013: 38-49

[12] Binna R, Zangerle E, Pichl M, et al. Hot: A height optimized trie index for main-memory database systems//Proceedings of the 2018 International Conference on Management of Data. Houston, USA, 2018:521-534

[13] Wu X, Ni F, Jiang S. Wormhole: A fast ordered index for in-memory data management//Proceedings of the 14th ACM European Conference on Computer Systems. Dresden, Germany, 2019: 1-16

[14] Mathew A, Min C. Hydralist: A scalable in-memory index using asynchronous updates and partial replication. Proceedings of the VLDB Endow, 2020,13(9):1332-1345

[15] Zeitak A, Morrison A. Cuckoo trie: Exploiting memory-level parallelism for efficient dram indexing//Proceedings of the 28th ACM Symposium on Operating Systems Principles. Virtual Event, Germany, 2021: 147-162

[16] Fredkin E. Trie memory. Communications of the ACM, 1960, 3(9):490-499

[17] Kissinger T, Schlegel B, Habich D, et al. Kiss-tree: Smart latch-free in-memory indexing on modern architectures//Proceedings of the 8th International Workshop on Data Management on New Hardware. Scottsdale, Arizona, 2012: 16-23

[18] Leis V, Scheibner F, Kemper A, et al. The art of practical synchronization//Proceedings of the 12th International Workshop on Data Management on New Hardware. San Francisco, California, 2016: 1-8

[19] Zhang H, Lim H, Leis V, et al. Surf: Practical range query filtering with fast succinct tries//Proceedings of the 2018 International Conference on Management of Data. Houston, USA, 2018: 323-336

[20] Mäsker M, Süß T, Nagel L, et al. Hyperion: Building the largest in-memory search tree//Proceedings of the 2019 International Conference on Management of Data. Amsterdam, The Netherlands, 2019: 1207-1222

[21] Heinz S, Zobel J, Williams H E. Burst tries: A fast, efficient data structure for string keys. ACM Transactions on Information Systems, 2002, 20(2):192-223

[22] Askitis N, Sinha R. Hat-trie: A cache-conscious trie-based data structure for strings//Proceedings of the 13th Australasian Conference on Computer Science. Ballarat, Australia, 2007: 97-105

[23] Lee S K, Lim K H, Song H, et al. Wort: Write optimal radix tree for persistent memory storage systems//Proceedings of the 15th USENIX Conference on File and Storage Technologies. Santa Clara, USA, 2017: 257-270

[24] Ma S, Chen K, Chen S, et al. ROART: Range-query optimized persistent ART//Proceedings of the 19th USENIX Conference on File and Storage Technologies. 2021: 1-16

[25] Kim W H, Krishnan R M, Fu X, et al. Pactree: A high performance persistent range index using pac guidelines//Proceedings of the 28th ACM Symposium on Operating Systems Principles. Germany, 2021: 424-439

[26] Wang K, Yang G, Li Y, et al. When tree meets hash: Reducing random reads for index structures on persistent memories. Proc. ACM Manag. Data, 2023, 1(1):1-26

[27] Luo X, Zuo P, Shen J, et al. SMART: A high-performance adaptive radix tree for disaggregated memory//Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation. Boston, USA, 2023: 553-571

[28] Knuth D. The art of computer programming volume 3: Sorting and searching. 2nd ed. Boston, USA: Addison-Wesley, 1997

[29] Morrison D R. Patricia—practical algorithm to retrieve information coded in alphanumeric. Journal of ACM, 1968, 15(4):514-534

[30] Kemper A, Neumann T. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots//Proceedings of the 27th International Conference on Data Engineering. Hannover, Germany, 2011: 195-206

[31] Tu S, Zheng W, Kohler E, et al. Speedy transactions in multicore in-memory databases//Proceedings of the 24th ACM Symposium on Operating Systems Principles. Farmington, USA, 2013: 18-32

[32] Nam M, Cha H, ri Choi Y, et al. Write-Optimized dynamic hashing for persistent memory//Proceedings of the 17th USENIX Conference on File and Storage Technologies. Boston, USA, 2019: 31-44

[33] Calciu I, Imran M T, Puddu I, et al. Rethinking software

runtimes for disaggregated memory//Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. USA, 2021: 79-92

[34] Böttcher J, Leis V, Giceva J, et al. Scalable and robust latches for database systems//Proceedings of the 16th International Workshop on Data Management on New Hardware. Portland, Oregon, 2020: 1-8

[35] Shi G. Robust optimistic locking for memory-optimized indexes[Master Thesis]. Simon Fraser University, British Columbia, Canada, 2023

[36] Bayer R, Schkolnick M. Concurrency of operations on b-trees. Acta informatica, 1977, 9(1):1-21

[37] McKenney P E, Slingwine J D. Read-copy update: Using execution history to solve concurrency problems//Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Systems. 1998: 509-518

[38] Kallman R, Kimura H, Natkins J, et al. H-store: A high-performance, distributed main memory transaction processing system. Proceedings of the VLDB Endowment, 2008, 1(2):1496-1499

[39] Jacobson G. Space-efficient static trees and graphs//Proceedings of the 30th Annual Symposium on Foundations of Computer Science. USA, 1999: 549-554

[40] Yang J, Kim J, Hoseinzadeh M, et al. An empirical guide to the behavior and use of scalable persistent memory//Proceedings of the 18th USENIX Conference on File and Storage Technologies: volume 20. Santa Clara, USA, 2020: 169-182

[41] Zhang Z, Chu Z, Jin P, et al. Plin: A persistent learned index for non-volatile memory with high performance and instant recovery. Proceedings of the VLDB Endowment, 2022, 16(2):243-255

[42] Oukid I, Lasperas J, Nica A, et al. Fptree: A hybrid SCM-DRAM persistent and concurrent b-tree for storage class memory// Proceedings of the 2016 International Conference on Management of Data. San Francisco, USA, 2016: 371-386

[43] Cooper B F, Silberstein A, Tam E, et al. Benchmarking cloud serving systems with ycsb//Proceedings of the 1st ACM Symposium on Cloud Computing. Indianapolis, USA, 2010: 143-154

[44] Fan B, Andersen D G, Kaminsky M. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing//Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation. Lombard, USA, 2013: 371-384

[45] Li X, Andersen D G, Kaminsky M, et al. Algorithmic improvements for fast concurrent cuckoo hashing//Proceedings of the 9th ACM European Conference on Computer Systems. Amsterdam, The Netherlands, 2014: 1-14

[46] Zhang Y, Xiong X, Balmau O. Tone: Cutting tail-latency in learned indexes//Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems. Rennes, France, 2022: 16-23

[47] Chen Y, Lu Y, Fang K, et al. uTree: A persistent B+-tree with low tail latency. Proceedings of the VLDB Endowment, 2020, 13(11):2634-2648

[48] David T, Guerraoui R, Trigonakis V. Asynchronized concurrency: The secret to scaling concurrent search data structures//Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems. Istanbul, Turkey, 2015:631-644

[49] Wang Q, Lu Y, Li J, et al. Nap: Persistent memory indexes for numa architectures. ACM Transactions on Storage, 2022, 18(1):1-35

[50] Böhm M, Schlegel B, Volk P B, et al. Efficient in-memory indexing with generalized prefix trees//Proceedings of the 2011 Datenbanksysteme für Business, Technologie und Web. Kaiserslautern, Germany, 2011:227-246

[51] Moura D, Mossé D, Petrucci V. Performance characterization of autonuma memory tiering on graph analytics//Proceedings of the 2022 International Symposium on Workload Characterization. Austin, USA, 2022: 171-184

[52] Cha H, Nam M, Jin K, et al. B3-tree: Byte-addressable binary b-tree for persistent memory. ACM Transactions on Storage, 2020, 16(3):17:1-17:27

[53] Chen S, Jin Q. Persistent b+-trees in non-volatile main memory. Proceedings of the VLDB Endowment, 2015, 8(7):786-797

[54] Arulraj J, Levandoski J J, Minhas U F, et al. Bztree: A high-performance latch-free range index for non-volatile memory. Proceedings of the VLDB Endowment, 2018, 11(5):553-565

[55] Liu J, Chen S, Wang L. Lb+-trees: Optimizing persistent index performance on 3dxpoint memory. Proceedings of the VLDB Endowment, 2020, 13(7):1078-1090

[56] Yang J, Wei Q, Wang C, et al. Nv-tree: A consistent and workload-adaptive tree structure for non-volatile memory. IEEE Transactions on Computers, 2016, 65(7):2169-2183

[57] Kim W, Seo J, Kim J, et al. clfb-tree: Cacheline friendly persistent b-tree for NVRAM. ACM Transactions on Storage, 2018, 14(1):5:1-5:17

[58] Hwang D, Kim W, Won Y, et al. Endurable transient inconsistency in byte-addressable persistent b+-tree//Proceedings of the 16th USENIX Conference on File and Storage Technologies. Oakland, USA, 2018: 187-200

[59] Venkataraman S, Tolia N, Ranganathan P, et al. Consistent and durable data structures for non-volatile byte-addressable memory//Proceedings of the 9th USENIX Conference on File and Storage Technologies. San Jose, USA, 2011: 61-75

[60] Tom C, Jim H, Arthur S. Emerging memories branch out. Los Gatos: Coughlin Associates & Objective Analysis, Technical Report: No. TR-001, CA,USA, 2023

[61] Kraska T, Beutel A, Chi E H, et al. The case for learned in-

dex structures//Proceedings of the 2018 International Conference on Management of Data. Houston, USA, 2018: 489-504

[62] Zhang Z, Jin P, Xie X. Learned indexes: Current situations and research prospects. Journal of Software, 2021, 32(4): 1129-1150(in Chinese)
(张洲,金培权,谢希科. 学习索引:现状与研究展望. 软件学报,2021,32(4):1129-1150)

[63] Ding J, Minhas U F, Yu J, et al. Alex: An updatable adaptive learned index//Proceedings of the 2020 International Conference on Management of Data. Portland, USA, 2020: 969-984

[64] Wu J, Zhang Y, Chen S, et al. Updatable learned index with precise positions. Proceedings of the VLDB Endowment, 2021, 14(8):1276-1288

[65] Vaidya K, Chatterjee S, Knorr E, et al. Snarf: A learning-enhanced range filter. Proceedings of the VLDB Endowment, 2022, 15(8):1632-1644

[66] Ferragina P, Vinciguerra G. The pgm-index: A fully-dynamic compressed learned index with provable worst-case bounds. Proceedings of the VLDB Endowment, 2020, 13(8):1162-1175

[67] Zhang Z, Jin P Q, Wang X L, et al. Colin: A cache-conscious dynamic learned index with high read/write performance. Journal of Computer Science and Technology, 2021, 36(4):721-740

[68] Zacharatou E T, Kipf A, Sabek I, et al. The case for distance-bounded spatial approximations//Proceedings of the 11th Annual Conference on Innovative Data Systems Research. Chaminade, USA, 2021: 1-7

[69] Spector B, Kipf A, Vaidya K, et al. Bounding the last mile: Efficient learned string indexing. arXiv preprint arXiv: 2111.14905, 2021

[70] Wang Y, Tang C, Wang Z, et al. Sindex: A scalable learned index for string keys//Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems. Tsukuba, Japan, 2020: 17-24

[71] Kim M, Hwang J, Heo G, et al. Accelerating string-key learned index structures via memoization-based incremental training. Proceedings of the VLDB Endowment,2024,17(8): 1802-1815

[72] Mitchell C, Montgomery K, Nelson L, et al. Balancing cpu and network in the cell distributed b-tree store//Proceedings of the 2016 USENIX Annual Technical Conference. Denver, USA, 2016: 451-464

[73] Ziegler T, Tumkur Vani S, Binnig C, et al. Designing distributed treebased index structures for fast rdma-capable networks//Proceedings of the 2019 International Conference on Management of Data. Amsterdam, The Netherlands, 2019: 741-758

[74] Kalia A, Kaminsky M, Andersen D G. Datacenter rpcs can be general and fast//Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation. Boston, USA, 2019:1-16

[75] Wei X, Chen R, Chen H. Fast rdma-based ordered key-value store using remote learned cache//Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation. Berkeley, USA, 2020: 117-135

[76] Zuo P, Sun J, Yang L, et al. One-sided RDMA-Conscious extendible hashing for disaggregated memory//Proceedings of the 2021 USENIX Annual Technical Conference. 2021: 15-29

[77] Wang Q, Lu Y, Shu J. Sherman: A write-optimized distributed b+tree index on disaggregated memory//Proceedings of the 2022 International Conference on Management of Data. Philadelphia, USA, 2022: 1033-1048

[78] Li P, Hua Y, Zuo P, et al. ROLEX: A scalable RDMA-oriented learned Key-Value store for disaggregated memory systems//Proceedings of the 21st USENIX Conference on File and Storage Technologies. Santa Clara, USA, 2023: 99-114

[79] Li P, Hua Y, Zuo P, et al. A high-performance rdma-oriented learned key-value store for disaggregated memory systems. ACM Transactions on Storage, 2023, 19(4):30:1-30:30

[80] An H, Wang F, Feng D, et al. Marlin: A concurrent and write-optimized b+−tree index on disaggregated memory//Proceedings of the 52nd International Conference on Parallel Processing. Salt Lake, USA, 2023: 695-704

[81] Min X, Lu K, Liu P, et al. Sephash: A write-optimized hash index on disaggregated memory via separate segment structure. Proceedings of the VLDB Endowment, 2024, 17(5): 1091-1104

CHU Zhao-Le, Ph. D. candidate. His research interests include learned indexes and NVM-based database systems.

LUO Yong-Ping, Ph. D. candidate. His research interests include database indexes and key-value storage for novel hardware.

JIN Pei-Quan, Ph. D., associate professor. His research interests include database systems for new hardware, big data storage and management, and moving-objects databases.

Background

This paper mainly focuses on the development of the main-memory database index technology, more specifically,

the development of trie indexes. In the era of big data, the database faces significant challenges in storing, managing, and analyzing massive data. With the decreasing price and growing size of memory, main-memory databases have been a research hotspot in the field of big data management. While the index structure performance is a critical bottleneck for main-memory databases, it's important to design a fast and stable main-memory index.

There are numerous works on the design of main-memory indexes, including B+ tree, hash table, and trie. However, there is no comprehensive and in-depth evaluation of the trie index, and we can not tell which trie index is the best. Therefore, in this paper, we reviewed the history of the trie index and conducted comprehensive experiments on the most recent trie indexes. We then reported and analyzed the result and concluded some lessons on the design and usage of the trie index.

This work is supported by the National Science Foundation of China under the grant no. 620724190.