计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.210700176

基于热点数据的持久性内存索引查询加速

刘高聪　罗永平　金培权
中国科学技术大学计算机科学与技术学院　合肥 230027
(cs1033@mail.ustc.edu.cn)

摘　要　非易失性内存(Non-Volatile Memory,NVM),也被称为持久性内存(Persistent Memory,PM),具有按位寻址、持久性、存储密度高、低延迟等特点。虽然 NVM 的延迟远小于闪存,但高于 DRAM(Dynamic Random Access Memory)。此外,NVM 还有读写不均衡、写次数有限等不足。因此,目前 NVM 还无法完全代替 DRAM。一种更为合理的方法是利用 NVM 构建基于 DRAM+NVM 的混合内存架构。文中针对 NVM 和 DRAM 构成的混合内存架构,着重研究了基于热点数据的持久性内存索引加速方法。具体而言,以数据访问中的倾斜性特征为基础,利用 DRAM 的低延迟和 NVM 的持久性与高存储密度,提出了在持久性内存索引的基础上增加基于 DRAM 的热点数据缓存,进而提出了可以根据热点数据的变化自动调整缓存的查询自适应索引方法。将所提方法应用到多种持久性内存索引上,包括 wBtree,FPTree 以及 Fast&Fair,并进行了对比实验。结果表明,当热点数据访问达到总访问次数的 80% 时,所提索引加速方法在 3 种索引上的查询性能分别取得了 52%,33%,37% 的提升。

关键词:非易失性内存;混合内存架构;热点数据;自适应索引
中图法分类号　TP311

Accelerating Persistent Memory-based Indices Based on Hotspot Data

LIU Gao-cong,LUO Yong-ping and JIN Pei-quan
School of Computer Science and Technology,University of Science and Technology of China,Hefei 230027,China

Abstract　Non-volatile memory(NVM),also known as persistent memory(PM),has the characteristics of bit-based addressing,durability,high storage density and low latency.Although the latency of NVM is much smaller than that of solid-state drives,it is greater than that of DRAM.In addition,NVM has shortcomings such as unbalanced reading and writing as well as short writing life.Therefore,currently NVM cannot completely replace DRAM.A more reasonable method is using NVM to build a hybrid memory architecture based on DRAM+NVM.Based on the observation that many data accesses in database applications are skewed,this paper focuses on the hybrid memory architecture composed of NVM and DRAM and proposes a hotspot data-based speedup method for persistent memory indices.Particularly,we utilize the low latency of DRAM and the durability and high storage density of NVM,and propose to add a DRAM-based hotspot-data cache for persistent memory indices.Then,we present a query-adaptive indexing method that can automatically adjust the cache according to the change of hotspot data.We apply the proposed method to several persistent memory indices,including wBtree,FPTree and Fast&Fair,and conduct comparative experiments.The results show that when the number of hotspot data visits accounts for 80% of the total visits,the proposed method can accelerate the query performance of the three indices by 52%,33% and 37%,respectively.

Keywords　Non-volatile memory,Hybrid memory architecture,Hotspot data,Adaptive index

1　引言

近年来,持久性内存,即非易失性内存,正在快速发展并受到广泛关注,如相变存储器(Phase-Change Memory,PCM)、自旋矩传输磁性存储器(Spin-Torque Transfer RAM,STT-RAM)、阻变存储器(Resistive RAM,RRAM)[1]。NVM 和传统的持久性存储器如固态硬盘(Solid State Disk,SSD)、硬盘(Hard Disk Drive,HDD)相比,具有按位寻址、低延迟的优点。而与 DRAM 相比,NVM 具有高存储密度、持久性的优点。表 1 列出了几种存储介质的对比。

鉴于 NVM 按位寻址、优秀的存取性能、持久性等特点,利用 NVM 构建内存数据库替代传统的磁盘数据库成为可能,但 NVM 还不能完全代替 DRAM。与 DRAM 相比,NVM 写寿命更短、延迟(特别是写延迟)更高。因此利用 NVM 构建基于 DRAM+NVM 的内存数据库系统是一个更好的选择,但同时也带来了许多新挑战,如排序算法[2]、

到稿日期:2021-07-19　返修日期:2022-02-27
基金项目:国家自然科学基金(62072419)
This work was supported by the National Natural Science Foundation of China(62072419).
通信作者:金培权(jpq@ustc.edu.cn)

连接处理[3]、索引[4]等。

表 1 不同存储介质的对比[1]

Table 1 Comparison of different storage media[1]

参数	读延迟	写延迟	字节寻址	持久性	存储密度	写次数
DRAM	25 ns	30 ns	Y	N	1×	10^{16}
NVM	50～70 ns	150～220 ns	Y	Y	2～4×	10^{10}
SSD	25 μs	300 μs	N	Y	4×	10^{5}
HDD	10 ms	10 ms	N	Y	N/A	10^{16}

由于 NVM 的写延迟高于读延迟,而且写次数有限,因此现有的持久性内存索引大都以减少 NVM 写次数为主要目标,例如 wB-tree[5]和 FPTree[6]均采用了无序叶节点的结构来降低数据写入叶节点时的额外写代价。如果采用传统 B+树的有序叶节点结构,当新的数据插入到叶节点时,必须执行叶节点内的排序操作,而且当叶节点填满时还需要执行叶节点分裂操作。对于持久性内存索引来说这些操作都会带来大量的 NVM 写操作,对持久性内存是不友好的。但是,wB-tree 和 FPTree 等持久性内存索引采用的无序叶节点设计带来了另一个问题,即查询性能的降低。这是因为当我们在无序叶节点上执行查询时,不得不扫描整个叶节点,这就带来了大量的 NVM 读操作。因此,wB-tree 等现有持久性内存索引在查询性能上都低于传统的 B+树。这对于许多读多写少的数据库应用来说显然是不友好的。因此,我们在设计持久性内存索引时,不仅应考虑减少 NVM 写操作次数,还需要兼顾查询性能的提升。

解决持久性内存索引查询性能优化的思路大致有两种:1)设计全新的读写优化索引结构;2)不改变现有持久性内存索引的结构,而是设计某种查询加速方法来提升索引查询性能。很显然,第一种方法更具有吸引力。但是,由于 NVM 读写不均衡以及写次数有限等特性的限制,目前在读写优化的持久性内存索引方面还没有提出突破性的方法。因此,本文采用第二种思路,即研究针对现有持久性内存索引的查询加速方法,以此来解决现有持久性内存索引查询性能不佳的问题。

基于上述背景,本文针对现有持久性内存索引的查询性能优化问题,提出了一种基于热点数据的持久性内存索引查询加速方法,以倾斜性数据访问[7-8]为前提,以基于 DRAM 和 NVM 的异构混合内存架构为基础,通过在 DRAM 上动态地建立热点数据的局部自适应索引,实现对持久性内存索引的查询加速。本文的主要贡献如下:

(1)针对现有持久性内存索引写快读慢的问题,提出了一种基于异构混合内存的热点数据感知的自适应索引结构,通过 DRAM 上的热点数据缓存和局部索引来提升现有持久性内存索引的查询性能。

(2)在 DRAM+NVM 的异构混合内存环境下,将基于热点数据的自适应索引加速策略应用到多种现有的持久性内存索引结构上,包括 wBtree[5],FPTree[6]和 Fast&Fair[9],并进行了对比实验。结果表明,当负载具有倾斜性时,本文提出的方法显著提升了 3 种索引的查询性能。

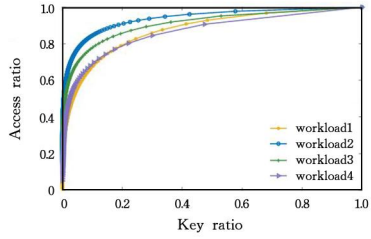
本文第 2 节介绍了相关工作;第 3 节给出了热点数据自适应索引的结构;第 4 节讨论了自适应索引应用在异构混合内存上的操作;第 5 节给出了实验结果与分析;最后总结全文并展望未来。

2 相关工作

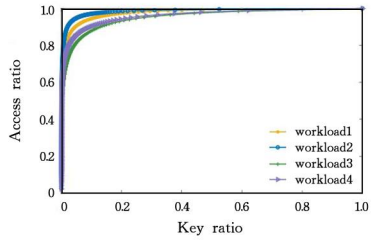
由于 NVM 写操作的延迟显著高于读延迟,并且具有有限的写寿命,因此,基于 NVM 的索引需要保证索引结构满足崩溃一致性(在系统崩溃后可以恢复到一致性的状态),并且需要降低一致性成本和写成本。

NVM 索引设计可以在传统内存索引上进行优化以适应 NVM 的特性,如对哈希索引优化的 Level Hash[10]和 Dash[11],对 B+树索引优化的 wBtree[5],FPTree[6]和 FAST&FAIR[9]等。由于 B+树索引已被广泛应用在数据库领域,因此目前的持久性内存索引大都以 B+树索引为基础进行设计。B+树是一种查询优化的索引结构,但它对于更新操作来说并不友好。因此,基于 B+树的持久性内存索引结构对 B+树的有序叶节点结构进行了修改,如 wBtree 和 FPTree 均将叶节点设计为无序结构,并采用追加写的方式将数据插入到叶节点。如此一来,数据写入叶节点时就不需要对叶节点进行排序、分裂等操作,从而可以有效减少索引结构的 NVM 写操作。但这样的设计会导致持久性内存索引的查询性能较差,因为无序叶节点上的查询需要扫描整个叶节点以及追加的数据。查询性能较差已经成为现有持久性内存索引的一个普遍问题。从这一角度来说,现有的持久性内存索引对于读多写少的数据库应用是不友好的,难以满足多样化的数据存取需求。

另一方面,现实世界中大多数的数据访问具有一定的倾斜性[12],这也是现有数据库系统中许多技术设计的基础,包括数据库缓存算法、查询连接算法等。热点数据访问在真实场景中是普遍存在的,尤其是在互联网领域[7]。图 1 是阿里巴巴不同业务下的数据访问分布情况[8],可以看到,50%(正常情况下)到 90%(极端情况下)的数据访问都集中在 1%的数据上。这表明热点问题在互联网大数据存储场景中具有现实意义。



(a)正常情况



(b)极端情况

图 1 阿里巴巴不同业务数据的访问比例[8]

Fig.1 Access ratio of different business data in Alibaba[8]

由于数据访问倾斜性的存在,如果对访问频度不同的数据采用统一的方式建立持久性内存索引,将无法区分热点

数据访问和冷数据访问。尤其在执行热点数据查询时,也不得不访问整个持久性内存索引。因此,本文提出了针对热点数据单独设计额外的 DRAM 缓存索引结构,使热点数据访问能够在 DRAM 中完成,从而提升持久性内存索引的查询性能。

3 热点数据自适应索引设计

3.1 设计目标

在数据访问负载严重偏斜的情况下,大量的查询将集中到小部分的热点数据上,如果将这小部分的热点数据识别出来并建立更小的索引,那么热点数据的查询可以快速得到响应。进一步地,在 DRAM 和 NVM 构成的异构混合内存系统中,我们可以将热点数据的索引构建在读性能更高的 DRAM 上,进一步提升查询性能。

但是,热点数据是由数据访问决定的。由于数据访问的倾斜性有可能随时间而变化,因此理论上热点数据也会随时间而改变。故热点数据索引必须能够探测热点数据的变化,并自适应地调整索引。下面给出了热点数据自适应索引的设计目标。

(1)能够根据数据访问的变化动态识别热点数据。

(2)具有较小的空间代价。热点数据不能针对大部分的数据构建索引(这种情况下热点数据索引就失去了价值),因此我们希望自适应索引能够针对小部分数据构建索引,需要的空间代价较小。

(3)具有较小的索引维护代价。如果热点数据的维护代价太高,其造成的性能增益会被额外的维护代价所抹平甚至产生负影响。

3.2 设计思路

热点数据自适应索引是根据查询特征动态建立的,对存储的热点数据的拷贝没有持久性要求。此外,自适应索引的维护需要进行写操作,因此将自适应索引存储在读写延迟更低的 DRAM 中。如图 2 所示,整体索引结构分为两部分:

(1)位于 NVM 中的全局索引(Global Index),即图 2 中的 globalIndex。由于热点数据不是连续的,只能用于加速点查询,因此全局索引采用 NVM 下的类 B+树索引,如 wBtree 和 FPTree,以保证较好的范围查询性能。在实现中,我们直接采用现有的 wBtree 等索引结构,使得本文提出的方法可以直接应用在现有的持久性内存索引之上。

(2)位于 DRAM 中的热点数据自适应索引(Adaptive Index)主要用于查询加速,即图 2 中的 localIndex。

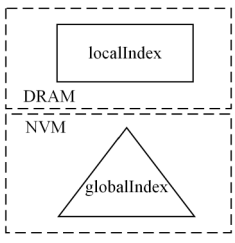
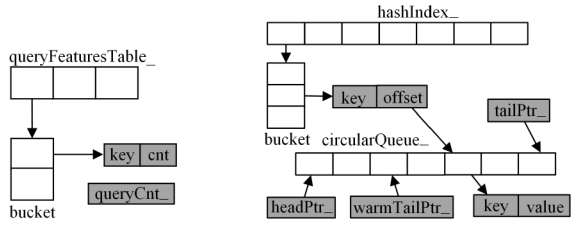


图 2 索引的总体设计思路

Fig.2 General idea of index

3.3 索引结构与实现

图 3 给出了用于识别热点数据的查询特征表以及索引存储结构。热点数据自适应索引的关键实现技术包括热点数据识别、热点数据索引以及存储、热点转移等。



(a)查询特征表 (b)索引和存储

图 3 热点数据自适应索引结构

Fig.3 Structure of hotspot data adaptive index

3.3.1 热点识别

根据数据的查询特征可以识别热点数据,但维护所有数据的特征空间代价太大。假设键值长均为 8 字节,一个 256 GB 的 NVM 可以存储 16 G 个键值对,即使每个键只用 1 字节存储查询特征,也会消耗 16 GB 的内存空间。为了减小空间代价,将查询按次数分为不同时期,如每百万次查询为一个时期。如图 3(a)所示,使用固定大小的静态哈希表 queryFeaturesTable_维护一个时期的查询特征。在一个较短的时期内,非热点数据出现多次的概率非常小,因此,如果一个数据在某个时期出现 K 次(K 和数据集规模以及时期大小有关),即可以认为它是热点数据。queryCnt_统计查询次数,区分查询时期,当下一个查询时期来临时,将 queryFeaturesTable_清空,以免当前时期热点数据的识别受到之前时期统计信息的干扰。此方案既减少了空间消耗(正比于一个查询时期的大小),又保证了热点数据的实时性。

3.3.2 索引和存储

由于热点数据只是部分数据,不支持范围查询,只支持点查询,因此我们选择哈希索引[10]来加快点查询,如图 3(b)所示。此外,我们使用固定规模的循环队列 circularQueue_来存储热点数据,以减少空间消耗。由于热点数据占比较小,固定规模可以取全部数据规模的 1%～10%,并且使用顺序表实现队列,以减少指针操作带来的性能消耗。

3.3.3 热点转移

热点数据可能随时间逐渐变化,而热点数据存储规模是固定的,因此会发生替换操作。数据替换时,如果根据访问次数替换,那么一般新进的热点数据会被替换掉,而之前时期访问较多的数据(最近时期不会用到)可能永远不会被替换。因此,我们不以访问次数作为替换参考,而采用一般的替换策略,如 FIFO(First in First out),LRU(Least Recently Used)[13]。

LRU 替换策略每次命中时,需要更新 LRU 队列,每次查询命中都需要进行数据转移,维护代价过高。FIFO 替换策略不需要维护,但会降低命中率。为此,本文综合了 LRU 和 FIFO 替换策略,提出了一种新的替换策略。

将热点数据分为两层,即热数据层和温数据层,两层的

替换策略都采用 FIFO。如图 4 所示,新进的热点数据写入热数据层,如果热数据层满了,热数据层中被替换的数据将被写入温数据层。如图 4(c)所示,插入 45 时,热数据层的队首元素 8 被替换掉,并写入温数据层。查找时,热数据层命中时不维护,温数据层命中时,将命中的元素插入到热数据层。如图 4(e)所示,命中 2 时,将 2 插入热数据层,热数据层队首元素 10 被替换并插入到温数据层中,温数据层队首元素 9 被替换掉,而温数据层中的元素 2 则失效。这样,温数据层命中时才维护,维护代价为 LRU 替换策略的 α(α为温数据层占所有热点数据的比例)。

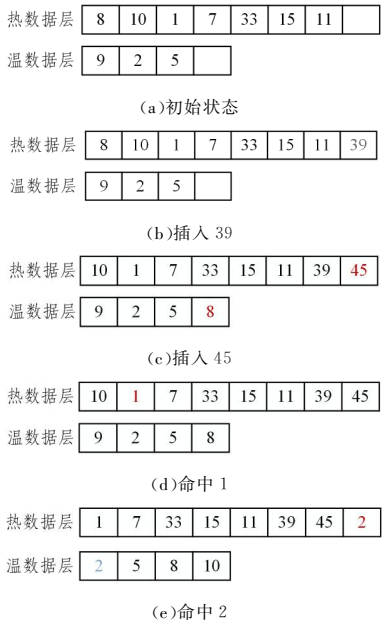


图 4 替换策略示例

Fig.4 Example of replacement strategy

如图 3(b)所示,我们采用 3 个指针将队列逻辑上分为两层:1)headPtr 到 warmTailPtr 为温数据层;2)warmTailPtr 到 tailPtr 为热数据层。当热数据层中被替换的数据写入温数据层时,只需要移动 warmTailPtr 指针,不需要转移数据。

本文提出的热点数据自适应索引能够识别热点数据,具有较小的空间代价和维护代价,并设计了查询特征表用于识别热点数据。为了减少空间代价,我们将查询分为不同时期,只记录当前时期的查询特征。为了减小维护代价并保证命中率,我们结合 FIFO 和 LRU,提出了一种新的替换策略,以较少的维护代价换取较高的命中率。

4 索引操作

本节讨论热点数据自适应索引上的操作算法,包括索引恢复、查询操作以及索引维护操作。

4.1 索引恢复

如果全局索引是纯 NVM 索引,如 wBtree,鉴于 NVM 的非易失性,则不需要恢复;如果全局索引是异构混合内存下的索引,如 FPTree,则对其进行恢复操作。而在 DRAM 中的自适应索引是根据查询特征动态建立的,不需要恢复,可以对其执行初始化操作。算法 1 实现了对自适应索引的初始化。

算法 1 自适应索引初始化 Initialize()

Input:无

Output:无

```
1.clear queryFeaturesTable_ and hashIndex_
2.resize queryFeaturesTable_ and circularQueue_
3.queryCnt_←0
4.headPtr_,warmTailPtr_,tailPtr_←0
```

初始化查询特征表和哈希索引,固定它们的大小(第 1—2 行)。由于热数据占比较小,循环队列的规模取所有数据的规模的 1%～10%。queryCnt_记录查询次数,以区分不同查询时期。headPtr_,warmTailPtr_,tailPtr_维护循环队列并且实现逻辑分层。

4.2 索引查询

算法 2 实现了对混合索引的查找操作。首先在局部自适应索引查找,如果查找成功,那么维护热点数据自适应索引并返回 true;如果在热点数据自适应索引中查找失败,则再去全局索引中查找,若查找成功,则维护热点数据自适应索引,此时 hit=false。若两个索引都查找失败,则返回 false。

算法 2 混合索引查找 HybridIndexFind (key,&value)

Input:查找键 key,值引用 &value

Output:如果存在 key,返回 true 并修改 value,否则返回 false

```
1.if adaptiveIndex.Find (key,value) then
2.    adaptiveIndex.Maintain(key,value,true)
3.    return true
4.end
5.if globalIndex.Find(key,value) then
6.    adaptiveIndex.Maintain(key,value,false)
7.    return true
8.return false
```

算法 3 实现了自适应索引的查找操作。如果哈希索引中存在 key,那么就可以得到在循环队列中的偏移量,从而访问该元素,得到 value 的值并返回 true,否则返回 false。

算法 3 自适应索引查找 Find(key,&value)

Input:查找键 key,值引用 &value

Output:如果存在 key,返回 true,并修改 value,否则返回 false

```
1.if key in hashIndex_ then
2.    value←ciclarQueue_[hashIndex_[key]].value
3.    return true
4.end
5.return false
```

4.3 索引维护

算法 4 实现了对自适应索引的维护操作。每次查找操作后,需要对自适应索引进行维护。如果之前命中了并且 key 在温数据层,那么将 key 和 value 插入到热数据层(第 1—4 行)。如果没有命中,则使用查询特征表维护 key 在这一时期的查询次数。如果 key 变成热数据则插入到热数据层(第 5—10 行)。更新查询次数,如果进入新时期,则清空查询特征表(第 11—14 行)。

算法 4 自适应索引维护 Maintain (key,value,hit)

Input:键 key,值 value,是否命中 hit

```
Output:无
1.if hit then
2.   offset ← hashIndex_[key].offset
3.   if offset > headPtr_ & offset < warmTailPtr_ then
4.      Insert (key,value)
5.   end
6.else
7.   use queryFeaturesTable_ to maintain the querynumber of key in
     this period
8.   if key becomes hot data then
9.      Insert (key,value)
10.  end
11.end
12.queryCnt_ ← queryCnt_+1
13.if queryCnt_=MAXQUERYCNT then
14.  queryCnt_ ← 0
15.  clear queryFeaturesTable_
16.end
```

算法5实现了对自适应索引的插入操作。如果当前队列是满队,那么就将队首元素从哈希索引中移除,队首指针后移(第1—4行)。将键值插入队尾和索引,维护队尾指针,并根据温数据和热数据占比调整温数据的队尾指针。

算法5 自适应索引插入 Insert (key,val)

```
Input:键 key,值 val
Output:无
1.if (tailPtr_+1)% circularQueue_size=head Ptr_ then
2.   remove circularQueue_[headPtr_].key from hashIndex_
3.   headPtr_←(headPtr_+1)% circularQueue_Size
4.end
5.insert (key,val) into circularQueue_[tailPtr_]
6.tailPtr_ ← (tailPtr_+1)% circularQueue_.Size
7.maintain warmTailPtr_ according to the ratio of hot data to warm
  data
```

此外,为了保证一致性,全局索引进行更新/删除操作后,热点数据自适应索引也要进行更新/删除操作。自适应索引的更新和删除操作比较简单,这里不再赘述。需要指出的是,由于热点数据自适应索引存储的是读热数据,因此插入操作并不影响热点数据自适应索引,只需要将数据插入到全局索引中。

5 实验与分析

本节讨论了实验的细节并对结果进行了分析。5.1节首先给出了实验设置,5.2节详述了实验结果,5.3节进行了总结。

5.1 实验设置

本文在配备了实际持久性内存的服务器上评估了热点数据自适应索引对3种NVM索引或异构内存索引(wBtree,FPTree,Fast&Fair)的加速效果。实验环境的详细配置如表2所列。服务器操作系统为Ubuntu,内核版本5.4.0。服务器包含256 GB DDR4 DRAM和512GB Intel Optane DC持久性内存[14]。GCC版本为9.3.0,使用PMDK 1.8管理PMem文件。wBtree和Fast&Fair模仿其开源版本实现,由于FPTree并未开源,实验使用了自行实现的FPTree。

表2 服务器环境配置

Table 2 Server configuration

服务器组件	说明
OS	Ubuntu 20.04.1 LTS
CPU	Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz
L1 cache	32kB iCache & 32kB dCache
L2 cache	1MB
L3 cache	24MB
DRAM	256GB DDR4 (2666MHz)
NVM	512GB Intel Optane DC
GCC version	9.3.0
PMDK version	1.8

实验使用2^{28}个键值对对索引进行预载,键长度为8字节,而值长度通常作为变量来测试不同工作负载下的性能,因此本实验也设置为8字节。为了模拟热点问题,测试负载分布特征为偏态(Skewed)分布,偏态分布的数据用开源实现[15]的Zipfian分布生成。该项目实现源自Gray等的工作[16],能生成一个整型Zipfian分布。实验中为了模拟不同时期的热点数据不同,zipfian分布的中心会根据时期随机移动,一个时期为100万次查询。实验中热点数据自适应索引规模为2^{23},热数据和温数据比例为20∶1。

5.2 实验结果

5.2.1 查找性能

热点数据访问达到80%时,查询次数分别取10×10^6,50×10^6,100×10^6,对6种索引进行测试,得到的吞吐量如表3所列。当查询次数为10×10^6时,热点数据自适应索引对wBtree,FPTree,Fast&Fair分别提升了61%,43%,53%。当查询次数为50×10^6时,热点数据自适应索引对wBtree,FPTree,Fast&Fair分别提升了57%,37%,46%。当查询次数为100×10^6时,热点数据自适应索引对wBtree,FPTree,Fast&Fair分别提升了52%,33%,37%。

表3 读吞吐量

Table 3 Read throughput

(单位:Mop/s)

索引\查询次数	10×10^6	50×10^6	100×10^6
wBtree	0.713	0.702	0.711
wBtree+adaptive index	1.154	1.106	1.082
FPTree	0.920	0.912	0.913
FPTree+adaptive index	1.317	1.250	1.213
Fast&Fair	0.811	0.805	0.808
Fast&Fair+adaptive index	1.245	1.179	1.106

在开始运行时,由于空间足够,热点数据直接储存而不需要替换。随着查询次数的增加和热点数据的转移,自适应索引的存储空间耗尽,此时存储热点数据会发生替换,带来额外的维护代价,加速效果减弱。当查询次数足够大时,替换次数占比接近1,此时加速效果将接近稳定。

查询次数与加速效果的关系如图5所示。当查询次数为10×10^6时,存储热点数据的空间还未耗尽,位于直线上;当查询次数为50×10^6以及100×10^6时,空间已经耗尽,处于曲线上。随着查询次数的增加,热点数据自适应索引对3种索引

的加速效果将趋于稳定,这个值接近查询次数为 100×10^6 时的加速效果。

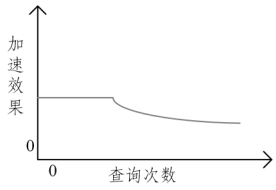


图 5 查询次数与加速效果的关系

Fig.5 Relationship between the number of queries and acceleration effect

5.2.2 偏态系数 skew 对吞吐量的影响

本文在查询次数取 100×10^6 的情况下,测试了偏态系数 skew 对 wBtree 和 wBtree 混合自适应索引吞吐量的影响,结果如图 6 所示。可以看出,当 skew 较小时,热点数据占比较小,维护代价大于热点数据命中的收益,此时性能反而下降。随着 skew 值的增大,数据冷热分布越明显,自适应索引对 wBtree 的性能提升越明显。当 skew 为 0.9 时,性能提升了 43%;当 skew 为 0.99 时,性能提升了 55%。

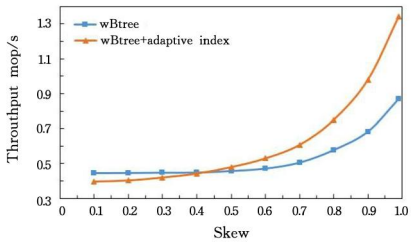


图 6 偏态系数对吞吐量的影响

Fig.6 Impact of skewness on throughput

5.2.3 命中率对结果的影响

命中率,即查询时直接在 DRAM 中的热点数据自适应索引命中的次数占总查询次数的比例。当负载变化时,命中率也会发生变化。图 7 给出了当查询次数为 100×10^6 时热点数据自适应索引命中率对性能加速比的影响。当命中率较低时,由于热点数据自适应索引有额外的维护代价,wBtree 混合自适应索引的性能相比 wBtree 有所下降。命中率为 0.09 时,性能下降了 11%;当命中率为 0.25 时,wBtree 混合自适应索引的性能和 wBtree 的性能相当。此后,随着命中率的提升,wBtree 混合自适应索引对 wBtree 性能的提升越明显。当命中率为 0.65 时,性能提升了 36%;当命中率为 0.84 时,性能提升了 86%。

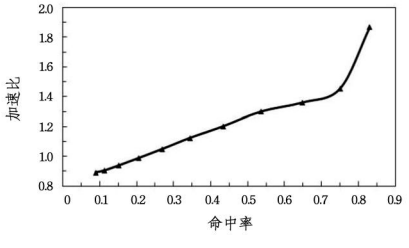


图 7 命中率对加速比的影响

Fig.7 Impact of hit rate on speedup

5.2.4 更新/删除

更新性能如表 4 所列。实验先以 zipfian 分布负载进行了 10×10^6 次查询操作,再分别用同一 zipfian 分布负载进行 10×10^6 次更新操作。可以看出,最差的情况下,读热数据同时也是写热数据,此时自适应索引使 wBtree,FPTree,Fast&Fair 的更新性能分别下降了 7%,11%,7.5%。一般情况下,数据写分布和数据读分布并不会完全一致,此时自适应索引对更新性能的影响应该更小。

表 4 更新吞吐量

Table 4 Update throughput

(单位:Mop/s)

索引结构	吞吐量
wBtree	0.538
wBtree+adaptive index	0.500
FPTree	0.612
FPTree+adaptive index	0.543
Fast&Fair	0.552
Fast&Fair+adaptive index	0.511

采用同样的步骤对删除性能进行测试,结果如表 5 所列。自适应索引更新操作需要先查询哈希表得到元素在队列中的偏移量,再根据偏移量读取元素并更新,而自适应索引删除操作只需要将键从哈希表中移除,因此自适应索引删除操作比更新操作更快。此时自适应索引使 wBtree,FPTree,Fast&Fair 的更新性能分别下降了 0.5%,3.5%,4%。

表 5 删除吞吐量

Table 5 Delete throughput

(单位:Mop/s)

索引结构	吞吐量
wBtree	0.537
wBtree+adaptive index	0.534
FPTree	0.735
FPTree+adaptive index	0.711
Fast&Fair	0.559
Fast&Fair+adaptive index	0.536

5.2.5 空间代价

本次实验中,数据集规模为 2^{28},查询特征表规模为 1×10^6,队列规模为 8×10^6,键值均为 8 字节,自适应索引的空间代价约为 270 MB,占总数据集空间(4 GB)的 6.6%。

5.3 实验总结

在真实 NVM 环境下,将热点数据自适应索引应用到 wBtree,FPTree 和 Fast&Fai 上,并进行了对比实验。实验结果表明,当热点数据访问次数占总访问次数的 80% 时,热点数据自适应索引仅消耗了 6.6% 的空间代价,对 3 种索引的查询性能分别提升了 52%,33%,37%。此外,我们还探究了偏态系数和命中率对查询性能的影响,评估了自适应索引对 NVM 索引的删除/更新性能的影响。结果显示,在最坏的情况下,热点数据自适应索引对 3 种索引的更新性能分别下降了 7%,11%,7.5%,对其删除性能分别下降了 0.5%,3.5%,4%。

结束语 本文针对热点问题,综合 DRAM 的存取性能、NVM 的持久性和高存储密度,提出了一种异构混合内存下的热点数据自适应索引,以对持久性内存索引进行查询加速。将热点数据自适应索引应用于 wBtree,FPTree 和 Fast&Fair 并进行了对比实验。实验结果表明,在热点数据访问占比

达到80%时,热点数据自适应索引在仅消耗6.6%左右的额外DRAM空间的情况下,wBtree,FPTree和Fast&Fair的查询性能分别提升了52%,33%,37%。

在未来的研究工作中,首先,我们将进一步研究更好的热点数据识别方案。目前提出的基于分时期统计的方案虽然能降低空间消耗,但对于热点数据的识别具有一定的延后性并且没有考虑热点数据不同时期的相关性。其次,我们希望实现热点数据自适应索引的并发版本。热点数据可能被大规模并发访问,因此支持并发操作是非常重要的。

参 考 文 献

[1] WU Z L,JIN P Q,YUE L H,et al.A survey on PCM-Based big data storage and management[J].Journal of Compute Research and Development,2015,52(2):343-361.

[2] LUO Y,CHU Z,JIN P,et al.Efficient sorting and join on nvm-based hybrid memory[C]// ICA3PP.2020:15-30.

[3] LIU Y,JIN P,WAN S.BF-Join:An efficient hash join algorithm for dram-nvm-based hybrid memory system[C]// ISPA.2019:875-882.

[4] VENKATARAMAN S,TOLIA N,RANGANATHAN P,et al.Consistent and durable data structures for non-volatile byte-addressable memory[C]// FAST.2011:61-75.

[5] CHEN S,JIN Q.Persistent B+-trees in non-volatile main memory[J].Proceedings of the VLDB Endowment,2015,8(7):786-797.

[6] OUKID I,LASPERAS J,NICA A,et al.FPtree:A hybrid scm-dram persistent and concurrent b-tree for storage class memory[C]// SIGMOD.2016:371-386.

[7] ATIKOGLU B,XU Y,FRACHTENBERG E,et al.Workload analysis of a large-scale key-value store[C]// SIGMETRICS.2012:53-64.

[8] CHEN J,CHEN L,WANG S,et al.Hotring:A hotspot-aware in-memory key-value store[C]// FAST.2020:239-252.

[9] HWANG D,KIM W H,WON Y,et al.Endurable transient inconsistency in byte-addressable persistent b+tree[C]// FAST.2018:187-200.

[10] ZUO P,HUA Y,WU J.Level hashing:A high-performance and flexible-resizing persistent hashing index structure[J].ACM Transactions on Storage,2019,15(2):13:1-13:30.

[11] LU B,HAO X,WANG T,et al.Dash:Scalable hashing on persistent memory[J].Proceedings of the VLDB Endowment,2020,13(8):1147-1161.

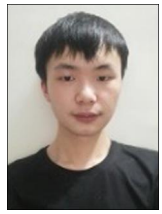
[12] JUNG J,KRISHNAMURTHY B,RABINOVICH M.Flash crowds and denial of service attacks:Characterization and implications for cdns and website[C]// WWW.2002:293-304.

[13] DAN A,TOWSLEY D.An approximate analysis of the LRU and FIFO buffer replacement schemes[C]// SIGMETRICS.1990:143-152.

[14] INTEL.Intel Optane Technology[EB/OL].https://www.intel.com/content/www/us/en/archit-ecture-and-technology/optane-dc-persistent-memory.html.

[15] GITHUB.Zipfian-distribution[EB/OL].https://github.com/ekg/dirtyzipf/blob/master/dirty_zipfian_int_distribution.html.

[16] GRAY J,SUNDARESAN P,ENGLERT S,et al.Quickly generating billion-record synthetic databases[C]// SIGMOD.1994:243-252.



LIU Gao-cong, born in 1999, postgraduate. His main research interests include database technologies for NVM and so on.

JIN Pei-quan, born in 1975, Ph D, associate professor, is a senior member of China Computer Federation. His main research interests include databases and big data.

(责任编辑:何杨)