文章编号:1000-5641(2014)05-0192-15

内存数据索引:以处理器为核心的性能优化技术

董绍婵, 周敏奇, 张 蓉, 周傲英
(华东师范大学 软件学院;数据科学与工程研究院,上海 200062)

摘要: 随着单机内存容量的持续上升,内存数据库技术逐渐取代传统磁盘数据库为数据管理提供更快速的支持.本文分析了设计内存索引结构所需要考虑的基本要素;对目前的内存索引结构进行了分类总结,并分析各结构的优缺点;针对当前应用发展趋势,指出内存索引未来发展的机遇与挑战;最后介绍了我们正在研发的分布式集群感知内存数据库(CLAIMS)中的内存索引结构.

关键词: 内存索引; cache 利用率; 分布式内存数据库; 索引压缩
中图分类号: TP392 **文献标识码**: A **DOI**:10.3969/j.issn.1000-5641.2014.05.017

In-memory index: Performance enhancement techniques leveraging on processors

DONG Shao-chan, ZHOU Min-qi, ZHANG Rong, ZHOU Ao-ying
(*Institute for Data Science and Engineering, Software Engineering Institute, East China Normal University, Shanghai* 200062, *China*)

Abstract: As main memory capacities grows larger and larger, the memory era has arrived and in-memory databases have taken the place of traditional disk-based databases to provide efficient data management. In this paper, we analyzed the fundamental elements in in-memory index designing: summarized and evaluated the existing index structures, pointing out the future opportunities and challenges based on the development trend of current applications. Finally, we introduced our on-going distributed in-memory index studies on the Cluster Aware In-Memory System (CLAIMS).

Key words: in-memory indexing; cache utility; distributed in-memory database; index compressing

0 引 言

在 20 世纪 70—80 年代,由于硬盘技术的不成熟,导致计算机磁盘空间增长缓慢,数据

收稿日期:2014-06
基金项目:国家自然科学基金(61332006)
第一作者:董绍婵,女,硕士研究生,研究方向为内存数据库系统. Email:scdong510@163.com
通信作者:周敏奇,男,副教授,硕士生导师,研究方向为内存数据库系统. Email:mqzhou@sei.ecnu.edu.cn.

访问性能低下. 当时数据库系统所处理的数据规模并不大,所以有人曾设想过在数据库处理过程中,将磁盘上的数据全部常驻在内存[1],以提高数据访问和处理的速度. 1980 年,IBM 提出“薄膜”磁头技术,这为进一步减小硬盘体积、增大容量提高读写速度提供了可能;1980 年代末期发明了磁阻磁头,使得盘片的存储密度较以往提高了数十倍;随后推出了首个容量超过 1 GB 的硬盘“IBM 3380 直连存储设备(DASD)”(每个容量为 2.5 GB)[2]. 从此,磁盘容量的增长速率远远超过了内存容量的增长,“内存数据库”的想法看似已无可能. 对内存数据库系统的研究也止步不前;研究人员或者转向实时系统领域,或者致力于对磁盘数据库性能的改进.

在传统的磁盘数据库中,数据从磁盘加载到内存的过程成为提高整个数据库系统性能的瓶颈所在. 为了减少数据访问时的 I/O 开销,各种基于磁盘的数据索引结构应运而生;其中,由于对数据的增删改有非常好的性能支持,本身结构可以进行范围查询等优势,B + 树[3]索引得到了广泛的应用.

近十几年来,随着 RAM 价格的不断下降,计算机内存的容量飞速上升. 在当今的主流服务器中,TB 级的内存容量已随处可见. 这使得完全可以将所有数据常驻内存,以实现快速的数据访问和更新,基于内存的数据库系统再次得到人们的重视.

在内存数据库系统中,所有的数据操作任务都可直接在内存中得到响应;因此,困扰传统数据库的 I/O 瓶颈不复存在. 类似于内存-磁盘间的性能差异,CPU 与内存对数据的不同处理能力导致内存数据库的瓶颈转变为 CPU-memory 代价. 由于处理器厂商与内存厂商相互分离的产业格局,导致内存技术与处理器技术发展的不同步. 图 1 给出了 20 世纪最后 20 年间 CPU 和 DRAM 性能增长的趋势;从图中可明显看出,在这段时间内,处理器的性能每年大约有 60%的提升,而内存性能的提升速度则每年只有 10%左右. 这种不均衡的发展使得当前内存的存取速度严重滞后于处理器的计算速度,内存访问瓶颈导致高性能处理器难以发挥应有的效率,并且直接制约了内存数据库系统操作性能的提升. 事实上,早在 1994 年就有研究人员分析和预测了这一问题,并将这种严重阻碍处理器性能发挥的内存瓶颈命名为"内存墙"(Memory Wall)[4]. 在很长一段时间内,对内存索引的研究都围绕着如何减弱“内存墙”问题对数据获取的延迟进行. 一系列“缓存敏感性”[5]索引的提出(例如 CSB-Tree[6]、CST-Tree[7]等)使得该问题得到了缓解.

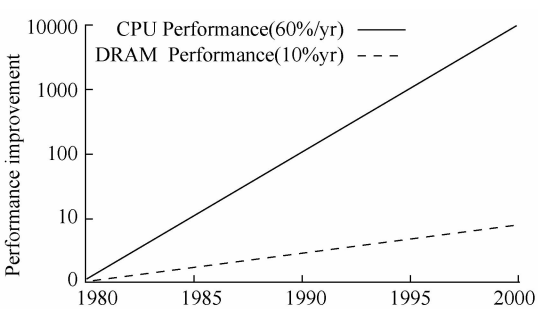

图 1　CPU 与 DRAM 性能增长对比[6]

Fig. 1　Comparison of perfomance increases for CPU and DRAM

在现今的 CPU-Cache-Memory 处理架构下,已有实验结果表明[8]:数据库执行时间的

一半用于等待内存,而且 90%的等待时间是由二级数据 cache 失效和一级指令 cache 失效造成的——这就意味着数据放置即数据模型对二级数据 cache、一级指令 cache 有重要的影响.因此,如何有效地利用 cache 的特征设计高效算法,成为内存数据库系统中亟待解决的一个重要问题.类比于传统数据库系统中对 I/O 瓶颈的索引处理方式,在内存数据库中,同样需要建立针对内存数据的索引结构,以避免对其进行全部扫描或大量随机读取,从而加快数据访问与处理的速度,尽可能减弱"内存墙"问题对数据库处理性能的影响.

除此之外,新的硬件架构也对内存索引的结构设计提出了各种要求.由于多核技术的发展及内存容量的增大,单台计算机中的内存被分配给了不同的处理器,从而催生了非一致性内存访问(Non-Uniform Memory Access,NUMA)[9]系统的出现.由于处理器访问直接挂载内存与远端内存的时间消耗存在差异,针对内存中数据的索引结构可增添新的标识信息以使得数据尽量被其所在内存直接挂载的处理器所处理,从而加快数据的访问处理时间.另外,随着处理器与内存间缓存层数的增加,各处理器对一级、二级缓存的私有化导致数据访问过程中的相干性冲突增加,在多线程处理环境下,当多个处理器同时对某一特定的索引结构进行遍历搜索时,如何有效的避免对特定结点访问所造成的相干性冲突也是提升内存数据访问效率的一个有效手段.

除了在单机环境下要解决的"内存墙"问题之外,大数据时代的到来,它所具有的"3V"(即海量(Valume)、高速(Velocity)、多样(Variety)[10])特性,对数据库的分析和处理能力提出了新的挑战.传统的单机内存数据库系统并不能够很好地满足要求,于是分布式数据管理系统受到重视并被广泛研究.基于 Key-Value Store 的分布式内存数据管理系统(例如 Spark[11]、RAMCloud[12]等)已经取得较好的应用.分布式内存数据库系统的研究也如火如荼.SAP 的 HANA[13]已经有相对稳定高效的分布式版本;MinSQL[14]作为一个实验室产物,是用 JAVA 搭建的,一个较为完善的分布式内存数据库系统.在分布式环境下,源数据的存储策略直接影响数据在集群中的分布;而之前的基于单机内存的索引结构,也必然要针对分布式的数据存储作出新的调整.整个数据在集群中数据结点间的分布式存储策略与在每个数据结点内部的部分数据的局部索引的结合,将导致分布式环境下索引的多层架构.目前,针对内存数据的分布式索引结构的研究并不是很成熟,仍有很多的工作值得去做.

图 2 列出了内存索引技术近 30 年来的开展.从 20 世纪 80 年代开始,内存索引结构被首次提出并开始针对其独立于传统的磁盘索引结构的特征进行研究改进.随着计算机硬件的不断变化,内存索引结构也产生了相对应的改变.从传统的基于 B 树类型的索引到 Trie 结构的引进,从对磁盘 hash 结构的直接应用到针对 cache 及 TLB 的特性进行的敏感性改进,直到后来针对不同的索引结构特性进行复合利用,逐步提升内存索引对数据扫描的性能影响.

在过去的 30 多年中,已有很多学者致力于研究如何对内存数据建立高效的索引结构,但是这部分的研究工作主要针对集中式环境,对分布式内存数据库系统并没有作相关的索引工作探讨.本文尝试全面回顾内存数据库索引工作在近 30 年来的研究进展与已有成果,并展望其未来可能的发展方向.此外,华东师范大学数据科学与工程院在分布式内存数据库的数据存储及索引建立方面也做了一些探索,本文也会给出简要的介绍.

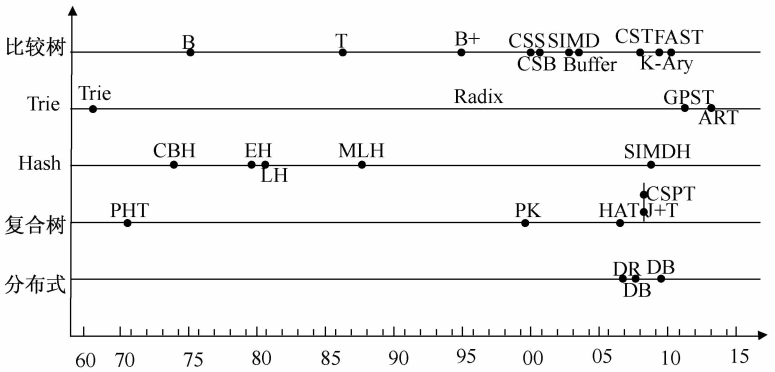

图 2　内存数据库系统索引发展概览

Fig. 2　Summarization of indexing techiques in in-memory databases

本文的后续内容组织如下:第 1 节介绍内存索引结构需要考虑的基本要素;第 2 节和第 3 节分别介绍单机和分布式环境下内存数据库索引结构的研究进展;第 4 节总结当今数据及应用的发展趋势并展望未来内存大数据索引的发展方向,同时简介华东师范大学数据科学与工程研究院在 CLAIMS 系统中的分布式内存索引研究;最后,第 5 节总结全文.

1　内存索引结构要考虑的基本要素

已有的研究成果表明,在传统的磁盘行数据库系统中,最大的性能瓶颈在于对大量磁盘数据的访问和获取.因此,产生了各种磁盘数据索引方式,用以避免对原始数据的全盘扫描,精准定位数据;然而,随着内存容量的上升及列存储方式的出现,基于内存的数据库系统可以将查询所需的原始数据全部缓存在内存中以便在查询到来时可以快速地获取到对应于查询的数据.

由图 3 给出的计算机中各级存储介质数据访问效率的对比可看出,当传统的磁盘瓶颈消失之后,内存访问延迟将会变成制约内存数据库系统查询处理性能提升的新瓶颈,如何加速数据从内存获取到 CPU 处理的过程已经成为内存数据库性能提升的一个重要研究方向.总结上述情况并结合近年来出现的新的硬件架构形式及分布式环境的发展,以下给出在设计内存数据索引时需要重点考虑的基本要素.

架构层次	数据获取时间
Core	- -
一级缓存	0.5 ns
二级缓存	7 ns
内存	0.25 ms
固态硬盘	1 ms
磁盘	10 ms

图 3　各级存储介质的访问延迟说明

Fig. 3　The access delay for different storages

1.1 提升 cache 利用率

类似于传统磁盘数据库中内存 buffer 与磁盘的关系,在内存数据库中数据 cache 可看作是内存中原始数据的更高一层存储 buffer. cache 与内存之间数据的交换以 cache line 为单位;每次不论实际所需的数据占用多少空间,一旦在 cache 中不能找到所需数据,访问内存获取的数据规模必定填充满一个完整的 cache line. 因此,尽量提高 cache 中的空间利用率,使得每次加载到 cache 中的索引数据能够全部有效地用于查询处理中对结果集的 prune 或直接指示,是设计内存索引结构的一个主要考虑因素.

以传统的 B 树[15]类索引为例,虽然该类索引在磁盘数据库中得到了非常广泛的应用并且也被证明可以取得很好的性能提升,但是显然这类结构并不能直接用于对内存数据进行索引. Ross 提出了"缓存敏感(cache conscious)"的概念,形式化地说明了对 cache line 利用率的优化问题,并在文献[5]中以传统 B 树结构为基础,在 OLAP(在线分析系统,On-Line Analysis Processing)环境下提升 cache 利用率,通过对其进行缓存敏感性改进使得其变体结构最终能被应用到内存数据索引结构中.

1.2 减少 cache 失效及 TLB 失效

通过已有的实验已经知道,数据库执行时间的一半是 CPU 在等待内存,而 90%的等待时间是由二级数据 cache 失效和一级指令 cache 失效造成的. 因此,减少二级数据 cache 失效的发生概率显得尤为重要. 对于树型索引,如果整个结构没有被加载到 cache,那么每次层间偏移都对应一次数据 cache 失效. 因此如何提高其结点的扇出分支数目,降低树高,减少索引搜索过程中 cache 失效发生的概率是内存索引要考虑的另一个重要因素.

对 hash 类索引而言,数据的定位只需一次 hash 值的计算,理想情况下搜索代价为 O(1). 但是由于 TLB 空间局限性,对于大规模分散 hash 而言,绝大多数数据桶的物理地址都不在 TLB 中,这使得在数据定位过程中 TLB 失效频发,严重影响索引效率. 针对这种情况,如何组织 hash 的层次以减少在寻址过程中出现的 TLB 失效问题是设计内存 hash 类索引要考虑的一个重要方面. 其中,将 Radix[16]技术引入索引结构而提出的分层 hash 算法得到广泛的认可.

1.3 索引结构压缩

对于传统的行存储磁盘数据库而言,表的所有属性以记录为单位按行存储,在没有索引的情况下对任一列的查询都将引发整个数据表的全部扫描,这极大地增加了系统的 I/O 开销,不利于查询性能的提升. 传统磁盘索引通过避免对数据的全部扫描来提高查询性能,其压缩比率以一列的索引空间对比整个完整表数据;因此,即使没有任何压缩技术,索引的压缩率也非常高.

当前,针对常驻内存的大量列数据建立索引结构,其压缩性能就成为一个非常重要的考量因素:首先,新的列存储数据组织形式使得原始表中的数据按属性进行组织存储,这样一来在针对某列操作时可以有针对性的只扫描该列的原始数据,大大缩小了对原始数据扫描的范围(相比行数据库而言). 因此,索引结构的压缩比率也由对比之前行存储的整个数据表变成仅与所要建立索引结构的列进行对比. 这种情况下,由于索引中添加了许多指示数据位置的信息,原来未经压缩的稠密索引所占用的空间必然大于原始该列数据所占用空间;因此,对于索引结构的精简和压缩在列存储的环境下就必不可少. 其次,由于源数据已全部在

内存中放置,其全部扫描相比磁盘而言本身已经非常迅速,要使索引结构发生效力,必定通过查询索引结构直接定位数据要快于有 cache 预取加速下的数据顺序扫描. 而如上一小节所说,不论是树型结构还是基于 hash 的索引结构,其在访问过程中都会不可避免的产生 cache 失效和 TLB 失效,类比于传统磁盘数据库的内存维护,最好的处理策略是能够让针对磁盘的索引结构直接维护在 cache 中,这样对索引结构的访问不会产生任何 cache 失效问题,整个数据的访问过程只会产生直接获取数据一次 cache 失效. 为了使整个索引结构能够在 cache 中维护,该结构所占用的空间量必须足够小,针对大规模数据集,索引的压缩就显得尤为重要.

1.4　新的硬件特性带来的索引结构改进

在传统磁盘数据库中,系统的性能瓶颈主要在磁盘 I/O 上,而单位时间内系统能从磁盘上读取的数据量与配置磁头数目,磁盘转速等物理因素直接相关,这导致只能通过改变物理架构来提升 I/O 吞吐量. 而在内存中,随着当今计算机硬件的发展,多核、多线程的处理模式完全能够支持处理性能的成倍提升. 图 4 给出了计算机存储架构示意图,结合新的结构特性及技术发展,为了充分利用硬件性能,内存中的索引结构应充分考虑到以下几个因素.

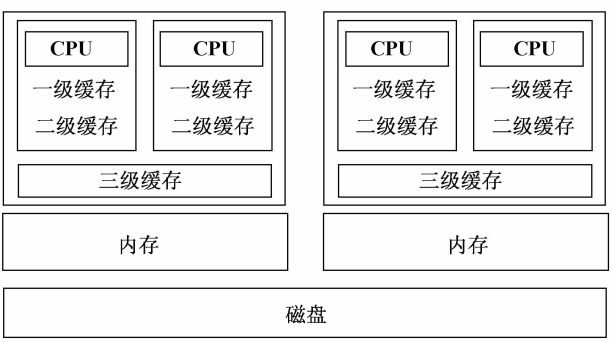

图 4　各级存储层架构示意图

Fig. 4　Hierarchical structure of in-memory

(1) 随着 cache 层数的增加以及多核技术的使用,在多线程环境下,核与核间 L3 cache 对内存中数据访问产生的相干性冲突,以及核内不同处理器间更高层 cache 对 L3 cache 中数据访问的相干性冲突,都会导致访问过程中处理器的等待,使得最终的总体性能下降;

(2) 由于内存容量的上升以及多核技术的应用,“非一致性内存访问”[9]结构使得 CPU 访问不同内存中的数据所需要的时间不尽相同;如何设计索引算法,使得在数据搜索过程中 CPU 所遍历的数据恰好在其直接挂载的内存中,这也是提升性能的一个有效途径;

(3) 同时单指令多数据流(Single Instruction Multiple Data,SIMD)[17]技术的合理利用也能够使索引的搜索性能得到进一步的提升;传统的单个 key 值比较指令可直接扩展到对整个数据结点的处理,从而节约 key 值在索引结点中比较过程的时间消耗,进一步加快索引访问的速度.

1.5　分布式内存集群特性

从图 2 中对近 30 年来的索引结构的初步总结可看出,内存索引结构的研究一直都孤立于内存数据库系统之外单独存在;并且几乎所有的索引结构所注重的只是在单机内存中的

性能优化策略,对于现今应用越来越广泛的分布式集群系统并没有太多的研究.

随着数据库所处理的数据规模的急剧增大,实时处理需求的不断提升,对集中式环境下服务器结点的要求也日益提高.相较于配置极高的单台服务器,基于分布式架构的集群更易被接受.内存数据处理系统发展的下一阶段必然会向分布式环境拓展,目前已经有许多较为成熟的分布式内存系统:基于 Key-Value 存储的 Spark、RamCloud;基于 Hadoop 的存储结构拓展而来的 Impala;以及分布式的内存数据库系统 memSQL、HANA 等等.相较于分布式内存系统的蓬勃发展,在其上的索引技术并没有随之兴起.而由于网络间内存访问的额外开销,分布式内存索引技术的研究也对传统的单机内存索引结构提出了新的要求:建立分布式索引必须结合考虑数据在分布式系统中的存储策略;索引结构与原始数据的一致性存储与否对数据访问时间的影响不可忽略.而在分布式环境下应用广泛的基于 HASH 的数据分布策略与传统的比较性索引结构的结合效率并不理想.因此,如何在分布式环境下建立高效的内存索引结构需要考虑的因素更加复杂.

2 现有的内存索引结构

回顾内存索引结构的发展,其每一次改进都与 CPU-Cache-Memory 架构的发展变化密切相关.由于源数据的内存存放,使得处理器及 memory 的架构对索引结构的影响逐渐增大.从传统的单核单处理器单线程架构到 SIMD 指令集的出现,到多线程多处理器多核时代的到来,每次 CPU-Cache-Memory 架构的改进都引起针对内存索引结构的改革.

2.1 基于树型结构的内存索引

鉴于 B-tree 一族的索引结构在传统磁盘数据库索引中取得良好的效果,目前大部分内存索引结构也是以 B-Tree 类的 Key 值比较树为原型,再针对 CPU-Cache-Memory 架构下的新特性进行性能优化.

(1)"内存索引"的首次提出——T-Tree

在 20 世纪 90 年代,B-Tree 作为一种被广泛接受的磁盘索引结构被直接用于内存索引;但是,由于当时 cache 的容量较小,B-Tree 的结点结构中指针数据过多,导致整个结构太过庞大;并且索引结构对 cache 的利用率明显不高,不能很好地满足 Cache-Memory 架构对索引的新要求.

针对 B-Tree 的不足,结合 AVL 树[18]和 B 树[15]的各自特性进行改进得到的 T-Tree[19]被认为是最早出现的针对内存数据建立的索引结构.该结构由威斯康星大学的 Lehman 和 Carey 共同提出,并在后来被应用于 MonetDB[20]中索引更新区的数据,取得了较好的查询性能提升.但是,由于其平衡的调整仍然借鉴 AVL 树的结点旋转规则,导致其失衡后的再平衡过程非常繁琐;并且插入的上溢出及删除的下溢出算法过于复杂,也导致 T-Tree 更新操作的平均消耗过大.

(2)"cache 敏感性"概念提出及索引结构的改进

尽管 T-Tree 的改进探索并不是很成功,但 T-Tree 的提出使得人们认识到,由于 Cache-Memory 架构与 Memory-Disk 架构之间的性能及容量等各方面的差异,所以内存索引结构并不能完全照搬磁盘索引结构.

当数据从磁盘全部转移到内存中后,cache 的结构特性对内存数据索引的建立影响重大,其涉及的最重要的一个问题是 cache 的敏感性.CPU 及 cache 对数据处理和访问速度的

差异成为内存数据库的主要性能瓶颈.已有实验结果表明,在内存数据库系统中,半数的CPU 等待时间消耗在二级 cache 的数据失效和一级 cache 的指令失效处理上[8].所以,如何合理地利用 cache 特征设计 cache 敏感算法已经成为内存数据库系统中要解决的一个重要问题.

针对内存数据索引和 cache 的架构特性,人们提出了"缓存敏感(Cache-Sensitive)技术"[5].研究表明,对传统的磁盘索引进行缓存敏感改造得到的内存索引结构能够较为显著地提升内存数据的查询性能."缓存敏感性"主要解决的问题是针对 cache 的容量较小、以 cache Line 为数据交换的最小单位以及 cache 失效所带来的巨大的时间消耗等特性;通过尽量提高 cache Line 的利用率来降低在索引搜索过程中 Cache Miss 发生的概率,从而提高查询的性能.

针对 Cache 敏感性问题,Ross 等人对传统面向磁盘的 B+-Tree 类索引结构作了一系列的改进,提出了新的缓存敏感的内存索引结构:基于 OLAP 应用的 CSS-Tree[5]及添加 OLTP(在线事务系统,On-Line Transaction Processing)的更新操作的 CSB+-Tree[6].其中 CSS-Tree 与 B+-Tree 的索引结构类似,利用索引结点的连续顺序存放特性,容易得出任意索引结点的任意子结点所存放的位置,从而完全省略 B+Tree 中用于指示子结点位置的指针空间,最大化索引结点的扇出分支数目,达到提高 cache 利用率的最终目的.并对索引结点大小与 cache line 的对应关系的改变导致的索引结构遍历过程中产生的 cache 失效次数给出了详细的理论论证,从而确立了索引结点大小与 cache line 容量相等的最优索引结点配置.

在当时的硬件环境下,针对 OLAP 应用,CSS-Tree 已经达到了接近最优的查询性能优化;然而,为此付出的更新代价极大:由于该索引结构中没有任何指针数据的存在,所有的结点按顺序连续存放,使得该索引结构的更新变得非常困难;在 OLAP 应用的定期批量更新操作中,索引结构只能通过重新构建的方式进行维护,并不具备实时插入更新的能力.为了解决 OLTP 系统中的实时增删改操作,必须对 CSS-Tree 的结构进行合理化的改进——适当地容许位置标示指针的存在.针对该应用需求,结合传统 B+Tree 在处理增删改操作时的规范,CSB+-Tree 在每个索引结点组(Node Group)中增加了一个位置指针,使得整个索引结构只需以结点组为单位连续存放即可;增加了索引结构的位置灵活性,并借鉴 B+Tree 的增删调节算法加以改进,实现了实时的动态索引项增删操作.改进后的 CSB+Tree 在提升 OLTP 应用中的数据查找性能上展示了较大的优势.在文献[6]中,除了提出针对基于内存的 OLTP 应用有较好性能提升的 CSB+Tree 索引结构外,对其索引结点组的规模也进行了形式化的讨论:通过 Segmented CSB+Tree 的结点块划分来调节整个索引结点组的空间占用大小及数量规模,从而调节索引项增删造成结点或结点组分裂时的维护开销.

在当时的 CPU-Cache-Memory 架构形式下,充分考虑 cache 敏感性问题所提出的 CS-类索引无疑极大提升了内存数据查找效率,最大化地发挥了内存索引在数据检索中的作用.到 2007 年,韩国的一组研究人员对 T-Tree 做了缓存敏感性改进,提出基于 T-Tree 的 CST-Tree[7],以改善数据所消耗的索引树层数较大且没有进行 cache line 对齐等问题;与此同时,合理避免了传统 T-Tree 在查找过程中可能发生结点误载入 cache 的情况;从而使经过改进的 CST-Tree 在数据查询的时间性能上大幅度优于之前的 CSB+Tree.

(3) 针对内存索引的优化技术

① SIMD 指令技术

当代计算机的 CPU 提供"单指令流多数据流"(SIMD)的指令集支持,以实现一条指令同时作用于多个操作数,从而加快数据处理的速度. 文献 [21]给出了将传统数据库操作进行 SIMD 推广的实现,并说明 SIMD 指令集对索引结构的支持给不同索引的构建或遍历带来性能提升.

有了 SIMD 指令集在数据库索引结构中的应用基础,2009 年 Schlegel 等人提出了 K-Ary Search 算法[22]. 该算法在传统的二分查找的基础上,对其进行 SIMD 拓展,使其每个比较指令针对 k 个操作数进行,将要查找的原始有序数据分为 $k+1$ 个区间;对比于原来的折半查找,单条指令过滤掉的数据成倍增加,从而加快有序数据的查找效率. 随后,由 Kim 等人提出的 FAST(Fast Architecture Sensitive Tree)[23]索引结构正式将 SIMD 技术应用于内存索引中:该结构以 K-Ary Search 为基础,充分考虑 cache 层的架构特性,平衡索引页块、cache line 块及 SIMD 指令块三者之间的关系,通过 page blocking-cache line blocking-SIMD blocking 三层架构的索引树存储方式,使得在兼顾、cache 敏感性问题的同时将 SIMD 指令运用于内存索引结构.

② 查询缓存技术[24]

在建立内存索引结构的基本考虑因素中除了,尽量提升 cache 利用率和考虑 cache 敏感性之外,减少 cache 失效所带来的额外时间损耗也是提升性能的有效方法之一. 在对海量数据建立内存索引时,并不能保证整个索引结构在任意时刻都在 cache 中缓存;因此在对基于树型结构的索引进行搜索时,也需要尽量避免 cache 失效所带来的性能损失. Kenneth A. Ross 团队以 B-Tree 和 CSB+ Tree 为基础,提出的针对批量查询的缓存技术,在很大程度上减少了系统冷启动或索引结构被替换出 cache 时对其搜索产生的 cache 失效次数. 该算法基于已有的内存索引结构,在结点(或结点组)上新加用于缓存搜索键值的数组空间. 当大量的查询来临时,逐层进行索引结构的查找将搜索键值存放于该层目标索引结点(所在结点组)对应的缓存数组中,只有当用于缓存键值的数组存满溢出时才会触发索引结构向下一层进一步查找;同样的,只有叶子结点(组)对应的缓存数组溢出时才会返回查询结果.

当批量查找操作到来时,假定数据索引结构不在 cache 中缓存,该算法将原来单独的索引结构中针对每个键值查找的 cache 失效次数进行了有效合并,使得一组键值的查找只会产生一次 cache 失效,大大减少了 cache 失效带来的时间消耗,从而提升数据搜索的效率. 需要注意的是,该算法的原始结构并不能保证批量查询的返回结果顺序与查询到来顺序一致,需要额外的保序算法;另外,该方案增大了索引结构所占用的空间大小(用于增加键值缓存数组),因此对键值缓存数组的选取需进行较为完善的空间代价分析.

2.2 基于字典(trie)结构的内存索引

文献[25]首次形式化的给出了 trie(也称数字搜索树 digital search tree[16])结构的概念——一种类似于字典的数据索引方式,直接通过对要查找的原始数据的每一位进行索引来组织整个索引结构,常被用于对字符串数据建立索引. 在 2011 年 SIGMOD Programming Contest 中,德累斯顿工业大学的 Kissinger 等人组成的团队所提出的针对内存数据的广义前缀搜索树(Generalized Prefix Search Tree GPS-Tree)[26,27]索引结构,取得了很好的成绩. 该结构以传统 trie-tree 为基础,用原始数据(限定为数值类型)对应的二进制表示形式建立

索引结构,并提出 bypass structure 和 dynamic,prefix-based expansion 算法压缩索引树高度从而取得了较好的查询性能.

文献[16]提出的 GPS-Tree 索引结构,在树高压缩上已经臻于完美;然而当索引数据长度较大并且值域空间分布较小时,其固定的结点结构使得结点中索引值浪费的情况非常普遍;并且其特定的针对二进制数据进行索引的限制条件也极大缩小了该结构能够处理的数据类型种类.为了解决这些问题,Leis 等人提出了 adaptive radix tree(AR-Tree)[28].在 AR-tree 中针对 GPS-tree 的问题作了一系列的改进:(1)索引结点的大小并不固定,而是根据其子结点的数目动态变化,从而尽可能地减少结点载入 cache 时导致的利用率下降问题;(2)形式化地定义了对传统 trie-tree 高度进行压缩的两种算法(path compression 和 lazy expansion)并将其应用于 AR-tree 中;(3)给出了常用的数据类型及其复合形式与二进制串的对应转换关系,拓展了 AR-tree 可用于索引的数据类型范围.经过如上的算法改进,AR-tree 在内存索引的查找性能相较于 GPS-tree 取得了进一步的提升.

2.3　Hash 及复合型结构的内存索引

(1) 基于 hash 结构的内存索引

传统的 Hash 索引结构也被广泛的用于磁盘数据库中,其索引结构不需要额外的存储空间,并且能够在 O(1)的时间复杂度下准确定位到所查找的数据.将磁盘数据库中的数据查找时间代价优化至最小.从 chained bucket hash[26]的提出到后来 extendible hash[29]、linear hash[30]、modified linear hash[19]的改进,只是对映射冲突的处理、桶分裂的策略进行了性能优化,并没有针对内存索引的结构特性进行任何算法改进,直至 2007 年,由 Ross 提出的基于现代处理器的 Hash 预取算法[31]才将 SIMD 指令集融入到 hash 算法中,从内存索引的角度提升传统 hash 算法的索引构造过程中数据组织的效率.

回顾 hash 算法的发展史可看出,针对 hash 算法的内存改进与传统磁盘 hash 相比没有明显的区别,单独的基于 hash 的内存索引结构在内存数据库系统中并不常见,大多被用作分布式环境下原始数据分布的一种平衡性策略.

(2) 复合型结构的内存索引

由于其对于字符串类型数据的完美索引方式,使得 Trie 索引结构及其变型总能达到相对较优的性能;但 Trie 的先天性结构问题导致在内存索引中对 cache 的利用率比较低下.因此,以 Trie 结构为基础,添加各种辅助结构优化其针对内存数据的索引能力成为复合型内存索引结构的主要组成方式.

Trie 与 hash 结构的结合:众所周知,trie 结构对字符串类型的数据有较好的索引效率,而 hash 可以根据数据的不同特征将原始数据划分为不同的模块.正如之前所说,由于 Trie 的基本结构中结点大小固定,当索引数据长度较大并且取值空间分布较小时,结点中索引项浪费情况明显;而结合 hash 先对数据进行分组,使得同一组中的数据范围相对缩小,可以有效的提升 trie 结构中索引结点的有效利用率.文献[32,33]中分别对 trie 和 hash 结构进行了结合,相较于原始的结构有效的提升了索引的查找效率.

Trie 与比较树类型索引结构的结合:由于 trie 结构本身的不保序性导致其无法应对范围查询,而在现实应用中绝大多数情况下会面临范围查询的需求.因此借鉴比较树类型索引的特性对 trie 进行改进,使其能够支持范围查询也得到广泛的研究:文献[34,35]中给出了 trie 与 B/T-Tree 结合的算法;文献[36]中给出了 trie 与 CSB+ Tree 结合产生的索引结构,

在满足范围字符串查询的基础上综合考量了缓存敏感性问题,对内存索引更具针对性.

3 分布式内存数据库中索引的应用

随着大数据时代的到来,数据处理系统所要处理的数据规模日益增大.传统的单机式系统已逐渐无法满足应用对处理性能的所提出的要求,内存数据处理系统也逐渐向分布式方向发展.随之而来的针对内存数据建立的索引结构必然要针对分布式数据存储进行有效的调整,使其在分布式环境下仍然能够高效进行数据查找过程中的过滤,快速精准的定位数据位置.

目前,对分布式内存数据库的研究还刚刚开始,尚未有非常完善的分布式内存数据库系统. memSQL[37] 作为一个分布式内存数据库系统,完全遵从传统的 ACID 事务准则;利用无锁的数据结构和即时编译器(Just-In-Time Compiler,JIT Compiler)处理高负荷的工作流,使得数据处理性能得到很大的提升;然而,其对数据的获取并没有添加任何基于索引算法的加速,仍旧采取简单的顺序扫描策略.除此之外,以 SAP 公司开发的商业内存数据库系统 HANA[13] 为例,虽然有分布式版本,但是相较于其单机版本的性能优化仍有一定的差距;并且 SAP 开发人员并没有提供对 HANA 的主数据建立内存索引的功能,只是在其用于处理增删改等 OLTP 操作的 deta 内存区默认建立了 CSB + Tree 的索引结构以便快速定位已被修改的数据.而对整个的未被修改的原始数据的搜索,其认为在内存环境下的列存储数据,顺序扫描的速率已经足够迅速,可以满足日常应用对数据查询的性能要求.

由于分布式内存数据库系统的研究并不成熟,相应的分布式内存数据索引结构的探索也并没有系统的展开.如何结合当前硬件的 CPU-Cache-Memory 架构对已有的单机内存索引结构进行分布式拓展,使其能够无缝的融入已有的分布式内存数据库系统从而大幅度提升数据查找过程的性能,成为内存索引研究领域的难点问题之一.

4 未来展望

当今是网络极具膨胀发展的时代,数据产生量日益增多,大规模数据的有效管理问题臻待解决,大数据的管理技术也已经成为当前和未来一段时间内计算机领域的重要研究课题之一[41].

4.1 数据及应用的发展趋势

随着云时代的来临,大数据也吸引了越来越多的关注. 2008 年 Nature 上出现"大数据"专刊[42],而时至今日大数据时代早已开启.尽管传统的基于磁盘的数据库系统在处理传统的应用时表现出了优异的性能,但在不同的时代所要处理的数据规模也大不相同.例如在 Web 1.0 时代,整个互联网的数据总规模较小,数据管理较为简单;而随着 Web 2.0、Web 3.0 时代的到来,所有的互联网用户均可在互联网上产生数据记录.以社交媒体为例,Face-Book 的日数据产生量已经达到 TB 级.日益庞大的网络数据规模对数据的有效管理提出了新的挑战,目前大数据处理中最大的困难并非是如何存储规模庞大的数据,而是如何从这些海量数据中高效检索出所需要的数据,并对其进行处理.

随着数据规模的日益增大,数据处理系统也逐步面临更新改革.在最初的数据库诞生之际,基于单机的简单的磁盘数据库管理系统能够充分满足当时数据处理的需求;随着时间的推移逐步出现了分布式数据库系统.当今各个不同企业或组织的应用不同,系统设计的目

标、所面向的负载也不尽相同,传统的面向磁盘的数据库系统所支持的数据分析处理功能受到前所未有的挑战,类似于 Hadoop[43],HBase[44],Cassandra[45]等完全颠覆数据库的 ACID 事务准则[46]而基于 key-value 的分布式存储处理系统也得到广泛的应用,并在海量数据的并行处理中取得了非常好的性能提升.另一方面,随着硬件技术的革新,内存价格迅速下降导致单机特别是高性能服务器的内存容量日趋增大,基于内存的数据处理系统也应运而生,Spark、Shark[47]等对 Hadoop 的内存拓展系统也得到广泛的应用.同时,如何将传统的磁盘数据库进行内存拓展也引起越来越多人的兴趣.与传统的磁盘数据库相比,内存数据库并不是单纯的增加了缓冲区的大小.当传统的磁盘 I/O 瓶颈消失后,CPU 与内存间的数据获取时间差异也成为新的内存数据库中的系统瓶颈.于是,对内存中数据快速有效的获取问题成为分布式内存数据库领域中新的研究热点.随着内存数据库的发展和分布式数据管理技术的提高,针对分布式的内存数据库系统的研究也成为新的领域热点问题.如何针对不同的数据类型、分布规律、查询处理频率等信息建立相对应的高效的分布式内存索引结构引起许多业内人士的讨论研究.

崭新的应用需求是推动数据库技术发展的源动力.企业中实际的应用需求决定了数据库系统发展及性能提升的研究方向.现今时代,数据量爆炸,对数据的实时性分析要求越来越高,例如证券交易所中对股票行情的实时展示,金融机构所提供的人们信用卡的近期消费情况统计分析等,这些应用导致企业对能够处理大数据量的实时分析系统的需求日益紧迫.毫无疑问,新生的分布式内存数据库系统在分布式数据处理效率上取得了极大的提升,并且依旧兼容传统的 SQL 查询方式,对上层的应用接口并未进行任何改变使得其更容易被企业所接收.同时,在传统数据库中极大提升搜索速率的磁盘索引结构也被借鉴到内存数据库中,针对内存数据的有效索引既要考虑分布式集群内存数据的分布情况,又要结合 CPU-Cache-Memory 的新型架构,使得内存数据索引的结构与传统的磁盘索引不尽相同,针对不同类型的应用需求都可以考虑建立相对应的查询加速索引结构.

4.2　未来内存索引研究的机遇与挑战

索引的有效 cache:Cache-Memory 间的访问延迟相较于 Memory-Disk 间更加微小(图 3),如何利用有效的索引结构在这更小的访问延迟中加速数据的访问效率,这是内存索引所要考虑的重要问题之一.由于在数据访问过程中 cache 失效所带来的时间消耗很大,利用有效策略使得在访问内存索引结构的过程中尽量避免 cache 失效成为提升整个数据访问性能的关键技术.随着数据规模的增大,如何设计高效的索引压缩算法,以尽可能减少索引所占用的空间,从而使得整个或者绝大部分的索引结构能够常驻在 L3 cache 中,在索引的访问过程中消除 cache 失效所产生的性能损耗,这些是未来设计内存索引结构所要面对的重要问题.

硬件变化:由于 cache 结构的特殊性所带来的 cache 敏感性,使得当今索引在设计的时候都要充分考虑所针对集群的具体架构;并且为了使得索引的搜索效率达到最大化,类似于 SIMD 指令技术及 NUMA 架构的访问性能等都做了针对性的优化,这使得索引结构在不同配置集群间的可移植性受到限制,从而影响整个内存数据库系统的集群可移植性.

分布式索引拓展:从本文第 2、3 节可看出,虽然内存索引的研究已经历经了三十多年,但其研究领域依局限于单机的环境,对现今流行的分布式环境并没有给出较好的拓展策略.而随着大数据时代的到来,海量的数据规模对单机的处理器性能提出了非常高的要求,使得

绝大多数企业都选择了分布式的处理环境从而减轻对单机的性能限制. 类似于 Hadoop、Shark、Spark 等基于 Key-Value 存储方式的分布式数据处理系统得到广泛的应用. 而内存数据库系统从原来的单机版本(传统的 MonetDB[20])向分布式版本拓展也变得尤为迫切. 针对单机的内存索引结构势必要进行分布式的改进,以便在分布式集群环境下继续有效的提升数据查找过程中的效率.

海量数据的地址表示:在传统的磁盘数据库中,除非外力进行干预(如人工进行数据迁徙)否则原始数据表格在磁盘上的位置保持不变;因此,磁盘索引的叶子结点可以直接用数据所在磁盘的逻辑地址来指示数据位置. 而在内存数据库系统中,不论内存容量如何增大,系统运行过程中总是无法保证有足够大的内存空间可以使得所有的表数据都完全被载入. 因此,内存数据库系统中也会存在数据在内存与磁盘间的交换(swap). 一旦内存中原始数据被 swap 到磁盘,当其重新被载入内存时,操作系统所分配给它的内存地址几乎不可能与之前的地址完全一致;因此,在内存索引结构中,叶子结点索引数据时并不能直接使用其在内存中的逻辑地址. 通常情况下,对小规模的数据而言该问题的解决办法为用数据相对于某些特定标志位置的偏移来指示数据在内存中的位置,这样只要原始数据的排序保持不变,无论其是否因内存不足被交换到磁盘过,只要重新被载入内存,都可以根据其相对目标位置的偏移准确的进行定位. 随着数据规模的增大,使得存储相对目标偏移所需的空间随之越来越大,索引结构叶子结点的规模也急剧上升,非常不利于整个索引结构在 L3 cache 中的常驻存储,因此,当数据规模庞大时,如何有效地表示数据所处的位置从而高效地定位数据也是未来内存索引要考虑的一个重要因素.

4.3 分布式内存索引探索@CLAIMS[48]

当今时代数据规模急剧增长,对数据的实时处理需求日益提高,即使当今单台服务器结点的性能可以达到非常高的标准,不论是处理器数目还是内存容量都相当可观,集中式的内存数据库系统仍然极大的限制了其数据处理的能力及规模. 内存数据库的分布式拓展显得尤为重要. 而目前并没有相对成熟的分布式内存数据库系统及比较成熟的内存索引结构. 华东师范大学数据科学与工程学院在此类系统的研发上已经开展了近两年的工作,致力于设计与实现一个集群感知的基于内存的分布式数据库系统(Cluster-Aware In-Memory System, CLAIMS). 该系统中也已经设计了针对内存数据的分布式索引结构,该索引基于传统的 B-tree[15],充分考虑 cache 的敏感性问题;并在此基础上结合列存储及分布式数据组织的特点,对该结构作进一步的优化:

(1) 考虑"cache 敏感性"问题,对传统的 B-tree 索引进行内存改进:消除子结点指针、数据指针分块存储、结点 cache line 对齐,得到 CSB-tree 结构.

(2) 结合分布式 hash 和 NUMA 技术对数据进行以 chunk 为单位的分布式内存 bounding,并对每个 chunk 建立 CSB-tree 构成分布式环境下的 Clustered CSB-trees.

(3) 利用 SIMD 指令集对 Clustered CSB-Trees 查询算法进行加速,以期达到更高的访问效率.

(4) 根据索引数据的值域进行结点压缩,进一步增大索引结点扇出,降低树高减少层间偏移产生 cache miss 的次数.

经过一系列的优化,我们所提出的 clustered enhanced CSB-trees 能够高效地应用于分布式内存数据索引,并在 CLAIMS 中得到实验验证.

5　小　　结

随着内存容量的大幅度增加及价格的急剧下降，使得单机的内存配置快速提升. 内存数据库相较于传统数据库的瓶颈的改变，导致之前在传统磁盘数据库中的所有性能优化策略都要在内存数据库系统中重新考虑. 本文对近 30 年来内存索引结构的技术发展，作了详细的回顾，在此过程中也发现由于内存索引的特殊性，所以其结构的发展与计算机 CPU-Cache-Memory 架构的发展有着密切的联系. 并且给出大数据时代到来对内存结构分布式拓展的挑战，在分布式环境下，无论是内存数据库系统还是索引结构都还有长足的发展空间. 本文最后还简单介绍了华东师范大学数据科学与工程研究院在分布式内存数据库系统及分布式内存索引方面正在进行的探索性工作.

[参　考　文　献]

[1] AMMANN A C, HANRAHAN M B, KRISHNAMURTHY R. Design of a memory resident DBMS[C]//Proc IEEE COMPCOM Conf, 1985.

[2] 微硬盘磁头弹性臂的研究[R/OL]. http://www.docin.com/p-270914249.html

[3] JANNINK J. Implementing deletion in B+-trees[J]. SIGMOD Record, 1995, 24(1).

[4] WOLF W A, MCKEE S A. Hitting the memory wall: Implications of the obvious[C]//ACM SIGARCH, 1995.

[5] RAO J, ROSS K A. Cache conscious indexing for decision-support in main memory[C]//Proceedings of the 25th VLDB Conference, 1999.

[6] RAO J, ROSS K A. Making B+-trees cache conscious in main memory[C]//SIGMOD, 2000.

[7] LEE I, SHIM J, LEE S, et al. CST-Trees: Cache Sensitive T-Trees[C]//DASFAA 2007, LNCS 4443, 2007: 398-409.

[8] AILAMAKI A, et al. DBMSs on a modern processor: Where does time go. In Proceedings of the 25th VLDB Conference, 1999.

[9] WIKIPEDIA. Non-uniform memory access [EB/OL]. [2014-08-31]. http://en.wikipedia.org/wiki/Non-uniform_memory_access.

[10] LANEY D. 3D data management: controlling data volume, velocity and variety[R]. Technical Report, Meta Group, 2001.

[11] Amplab spark-lightning-fast cluster computing: https://amplab.cs.berkeley.edu/projects/spark-lightning-fast-cluster-computing.

[12] RAMCloud[EB/OL]. https://ramcloud.stanford.edu/wiki/display/ramcloud/RAMCloud.

[13] Homepage for SAP HANA Database: http://www.saphana.com/welcome.

[14] SWART G. MinSQL: a simple componentized database for the classroom[C]//PPPJ, 2003:129-132.

[15] BAYER R, MCCREIGHT E M. Organization and maintenance of large ordered indices[J]. Acta Inf, 1972(1): 173-189.

[16] KNUTH D E. The art of computer programming Volume I: fundamental algorithms[M], 3rd ed. Addison-Wesley, 1997.

[17] Single Instruction Multiple Data: http://en.wikipedia.org/wiki/SIMD

[18] AHO A, HOPCROFT J, ULLMAN J D. The design and analysis of computer algorithms, Addison-Wesley Publishing Company, 1974.

[19] LEHMAN T J, et al. A study of index structures for main memory database management systems[C]//Proceedings of the 12th VLDB Conference, 1986.

[20] Homepage for MonetDB[EB/OL]. http://www.monetdb.org/Home

[21] ZHOU J, ROSS K A. Implementing database operations using SIMD instructions[C]//SIGMOD, 2002.

[22] SCHLEGEL B, GEMULLA R, LEHNER W. k-Ary search on modern processors[C]//DaMoN, 2009.
[23] KIM C, CHUGANI J, SATISH N, et al. FAST: fast architecture sensitive tree search on modern CPUs and GPUs[C]//SIGMOD, 2010:339-350.
[24] ZHOU J, ROSS K A. Buffering accesses to memory-resident index structures[J]. VLDB, 2003.
[25] FREDKIN E. Trie Memory[J]. Communications of the ACM, 1960, 3(9):490-499.
[26] BOEHM M, SCHLEGEL B, VOLK P B, et al. Efficient in-memory indexing with generalized prefix trees[J]. BTW, 2011:52-55.
[27] KISSINGER T, SCHLEGEL B, BOEHM M, et al. A high-throughput in-memory index, durable on flash-based SSD[J]. SIGMOD, 2012,41(3):40-50.
[28] LEIS V, KEMPER A, NEUMANN T. The adaptive radix tree: aRTful indexing for main-memory databases[J]. ICDE, 2013.
[29] FAGIN R, NIEVERGELT J, PIPPENGER N, et al. Extendible hashing—A fast access method for dynamic files [J]. ACM Trans Database Syst, 1979, 4(3):315-344.
[30] LARSON P. Linear hashing with separators-a dynamic hashing scheme achieving one-access retrieval[J]. ACM Trans Database Syst, 1988,13(3):366-388.
[31] ROSS K A. Efficient hash probes on modern processors. In ICDE, 2007.
[32] COFFMAN E G, EVE J. File structures using hashing functions[J]. Commun ACM, 1970, 13(7):427-433.
[33] ASKITIS N, SINHA R. HAT-Trie: a cache-conscious trie-based data structure for strings. In ACSC, 2007.
[34] BOHANNON P, MCILROY P, RASTOGI R. Main-memory index structures with fixed-size partial keys. In SIGMOD, 2001.
[35] LUAN H, DU X, WANG S. Prefetching J+-tree: a Cache-optimized main memory database index structure[J]. J Comput Sci Technol, 2009,24(4):687-707.
[36] BINNIG C, HILDENBRAND S, FARBER F. Dictionary-based order preserving string compression for main memory column stores[C]//SIGMOD, 2009.
[37] Homepage for memsql: http://www.memsql.com.
[38] AGUILERA M K, GOLAB W, SHAH M A. A practical scalable distributed b-tree[C]//VLDB, 2008.
[39] WU S, JIANG D, OOI B C, et al. Efficient b-tree based indexing for cloud data processing[C]//Proc VLDB Endow, 2010,3(1): 1207-1218.
[40] MOUZA C, LITWIN W. Sd-rtree: A scalable distributed rtree[C]//ICDE, 2007: 296-305.
[41] 宫学庆,金澈清,王晓玲,等. 数据密集型科学与工程:需求和挑战[J]. 计算机学报,2012, 35(8): 1563-1578.
[42] Big Data Section in Nature: http://www.nature.com/news/specials/bigdata/index.html.
[43] WHITE T. Hadoop 权威指南[M]. 周敏奇,王晓玲,金澈清,钱卫宁译. 2 版. 北京:清华出版社,2011.
[44] Apache HBASE[EB/OL]. http://hbase.apache.org.
[45] Apache Cassandra[EB/OL]. http://cassandra.apache.org.
[46] HAERDER T, REUTER A. Principles of transaction-oriented database recovery[C]//ACM Computing Srveys, 1983, 15(4): 287-317.
[47] XIN R S, ROSEN T, ZAHARIA M, et al. Shark: SQL and rich analytics at scale[C]//SIGMOD, 2013: 13-24.
[48] CLAIMS Homepage at Github[EB/OL]. https://github.com/scdong/Claims/wiki.

(责任编辑 王善平)