HiBase:一种基于分层式索引的高效 HBase 查询技术与系统

葛 微[1),3)] 罗圣美[2),4)] 周文辉[1),3)] 赵 頔[1),3)] 唐 云[1),3)]
周 娟[1),3)] 曲文武[2)] 袁春风[1),3)] 黄宜华[1),3)]

[1)](南京大学 计算机软件新技术国家重点实验室 南京 210046)
[2)](中兴通讯股份有限公司 南京 210012)
[3)](江苏省软件新技术与产业化协同创新中心 南京 210046)
[4)](清华大学计算机科学与技术系 北京 100084)

摘 要 大数据时代,众多应用领域的数据量爆炸式增长,迫切需要研究和寻找有效的大数据存储管理方法,提供实时或准实时的大数据查询分析能力.Hadoop HBase 系统为大数据的存储管理提供了一种具有高可扩展性的技术方法和系统平台.然而 HBase 只有主键索引,不支持非主键索引,这导致 HBase 的数据查询效率较低,难以满足数据实时或准实时查询需求.为此,在 HBase 基础上提供面向非主键的快速查询能力,是目前 Hadoop 环境下急需研究和解决的一个重要问题.该文研究提出了一种基于分层式 HBase 非主键索引的查询模型和方法,该模型和方法首先建立基于 HBase 的持久性索引.然后,为了利用内存提升查询性能,该文进一步提出了一种索引热点数据缓存技术和一种高效的热度累积缓存替换策略,以降低对 HBase 索引表的磁盘访问开销.热度累积缓存替换策略克服了最近最少使用(LRU)算法的局限性,考虑数据访问的累积热度和时间局部特性,从而更准确地捕获数据访问的特征.为了使索引热点数据缓存内存层具有良好的可扩展性,HiBase 设计了基于一致性哈希的分布式内存缓存,支持高效的基于非主键的单点查询和范围查询.最终,该文设计实现了完整的分层式索引和查询系统 HiBase.在千万至十亿条记录规模数据集上的测试结果表明,HiBase 冷查询响应时间比标准 HBase 快 65 倍(大结果集)到 3000 多倍(小结果集);而引入基于查询热度累积算法的内存索引缓存方法后,热查询性能可在 HiBase 冷查询基础上再提升 5～15 倍,使得总体查询性能比标准 HBase 快 300 多倍(大结果集)到 1.7 万倍(小结果集),比开源的 Hindex 系统快 5～20 倍.

关键词 HBase;非主键索引;查询处理;分层式索引;缓存替换策略;大数据
中图法分类号 TP311 **DOI 号** 10.11897/SP.J.1016.2016.00140

HiBase: A Hierarchical Indexing Mechanism and System for Efficient HBase Query

GE Wei[1),3)] LUO Sheng-Mei[2),4)] ZHOU Wen-Hui[1,3)] ZHAO Di[1),3)] TANG Yun[1),3)]
ZHOU Juan[1),3)] QU Wen-Wu[2)] YUAN Chun-Feng[1),3)] HUANG Yi-Hua[1),3)]

[1)](*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing* 210046)
[2)](*ZTE Corporation, Nanjing* 210012)
[3)](*Collaborative Innovation Center for Novel Software Technology and Industry of Jiangsu Province, Nanjing* 210046)
[4)](*Department of Computer Science, Tsinghua University, Beijing* 100084)

Abstract Nowadays we enter the big data era. The amount of data is growing explosively in many business areas. There is an urgent need for efficient storage and management of big data to provide realtime or near-realtime query for data analysis. Hadoop HBase provides a technical method and system with excellent scalability for storing and querying big data. However, HBase

收稿日期:2015-01-18;在线出版日期:2015-04-07.本课题得到国家自然科学基金专项基金(61223003,61362006)和中兴通讯产学研合作项目资助.**葛 微**,女,1979 年生,博士研究生,中国计算机学会(CCF)会员,主要研究方向为查询处理、查询优化、分布式和并行计算.E-mail:gloria.w.ge@gmail.com.**罗圣美**,男,1971 年生,硕士,高级工程师,中国计算机学会(CCF)会员,主要研究方向为云计算、云存储、大数据等技术领域.**周文辉**,男,1987 年生,硕士,主要研究方向为并行计算、大数据分布式一致性.**赵 頔**,男,1990 年生,硕士研究生,主要研究方向为分布式和并行计算.**唐 云**,男,1991 年生,硕士研究生,主要研究方向为大数据分布式系统和并行计算.**周 娟**,女,1990 年生,硕士研究生,主要研究方向为分布式和并行计算.**曲文武**,男,1979 年生,博士,主要研究方向为大数据查询优化.**袁春风**,女,1963 年生,教授,中国计算机学会(CCF)高级会员,主要研究领域为体系结构与并行计算、多媒体文档处理、Web 信息抽取与集成等.**黄宜华**(通信作者),男,1962 年生,教授,中国计算机学会(CCF)高级会员,主要研究领域为大数据处理技术、计算机体系结构、分布式与并行计算.E-mail:yhuang@nju.edu.cn.

only provides the row key indexing and does not support non-key indexing, which makes it insufficient to meet the need of realtime or near-realtime applications. In this paper, we proposed a hierarchical secondary indexing model and method for HBase. It built the permanent layer of secondary index for non-key columns in HBase table to speed up the query process. Furthermore, we presented the Hotscore Algorithm with hot-index cache mechanisms and an efficient cache replacement policy, to reduce the disk access overhead for index data. The Hotscore Algorithm overcame the limitations of the Least Recently Used (LRU) policy. To differentiate the hot and cold index data more precisely and fit in the time locality of data accesses, the Hotscore Algorithm presented a new method by accumulating the access frequency of records and reducing the accumulation variable exponentially and periodically. Additionally, we designed the distributed memory cache protocol based on consistent hashing to provide excellent scalability for the hot-index cache layer. Finally, we implemented a hierarchical indexing system HiBase. The experimental results on datasets ranging from 10 million to one billion records show that, the HiBase cold query (the cache-missed query) outperforms the standard HBase by 65 times (for large result sets) to more than 3000 times (for small result sets) respectively. Further, the HiBase hot query (the cache-hit query) after adopting the Hotscore Algorithm cache replacement policy can achieve extra 5—15 times speedup compared to the HiBase cold query, making the overall performance speedup more than 300 times (for large result sets) to 17000 times (for small result sets) compared to the standard HBase and speedup 5—20 times compared to the open-source Hindex secondary indexing system.

Keywords　HBase; secondary index; query processing; hierarchical index; cache replacement policy; big data

1 引　言

继 Google 在 2006 年发表论文阐述 BigTable[1] 数据模型后，为了有效应对海量数据的存储与查询管理，人们越来越多地用 NoSQL 分布式数据存储系统替代关系数据库管理系统(RDBMS). 在迫切的企业大数据应用需求推动下，目前出现了一些NoSQL非关系数据存储系统，例如，Apache 社区的 HBase①，Facebook 的 Cassandra[2]，Amazon 的 Dynamo[3] 以及支持高效数据查询的内存数据存储系统 Redis② 等等. 这些数据存储都采用了 key-value 数据模型. 在 key-value 数据存储系统中，HBase 的使用最为广泛.

HBase 在主键上建立了类 B^+ 树索引，可以高效地支持基于主键的快速数据查询. 然而，由于 HBase 缺少非主键索引能力，在面对以非主键为条件的查询请求时，需要对全表进行扫描，导致查询效率低下，难以满足需要快速响应的数据查询和统计分析场景. 在传统关系数据库领域，面对这样的问题，通常会借助索引来解决. 为了优化非主键查询的响应时间，降低查询代价，本文研究提出了一种基于 HBase 的非主键索引和查询模型与技术.

随着内存性能的持续提升和价格的不断下降，内存在支持实时大数据处理的商业需求中扮演着越来越重要的角色. 目前，用内存计算提升大数据处理性能已成为普遍公认的发展方向[4]. 在大数据查询分析过程中，数据访问通常具有一定的 80/20 分布特征：即 20%的数据通常会被频繁访问，我们把这些数据称之为“热数据”；而 80%的数据很少被访问，我们称之为“冷数据”. 为了提高查询性能，低延迟、高带宽的内存被普遍用做热数据缓存来填补 CPU 和磁盘之间的性能鸿沟.

为了优化 HBase 上非主键属性查询的性能，本文研究实现了基于 HBase 的分层式索引和查询系统 HiBase(Hierarchical-indexed HBase). 在索引存储模型上，HiBase 分为两层：① 基于 HBase 的持久化存储层. 用来存储 HBase 用户表对应的所有非主键索引数据；② 基于分布式内存的缓存层. 为了进一步提高索引访问的速度，本文研究提出了一种基于分布式内存的索引热点数据缓存替换策略和技术，将部分索引热点数据缓存在内存中，大幅提高数据查询访问的效率. 在 HiBase 中，我们进一步研究

① Apache HBase. http://hbase.apache.org/
② Redis. http://redis.io/

实现了热度累积的缓存替换策略,它克服了最近最少使用算法(LRU)只关注数据最近一次访问时间、不关注数据访问频繁程度不同的局限性,考虑数据访问的累积热度,并对早前时间的热度累积进行指数衰减,从而更准确地捕获数据访问的特征.同时在缓存空间未满的时候,采用"访问即插入"的缓存插入策略,保证缓存空间得到充分利用,缓存命中率在数据加载阶段可以得到快速提升并趋于稳定.而当缓存充满以后,热度累积的缓存替换策略会根据记录的热度累积评分选择"牺牲者"淘汰出内存,选择获得热度高分的记录保存在缓存中.通过实验对比算法的命中率和系统的查询响应时间,验证了热度累积的缓存替换策略可以有效地优化非主键索引上的查询性能.

本文第 2 节介绍相关工作;第 3 节给出 HiBase 的非主键索引及其存储模型;第 4 节提出 HiBase 索引热点数据缓存替换策略;第 5 节阐述 HiBase 的系统设计与实现;第 6 节对实验结果进行分析评估;第 7 节是总结与展望.

2 相关工作

通常,检索 HBase 表中的数据有 3 种方式:指定单个行键查询、指定行键的范围查询以及扫描(Scan).HBase 以字节数组的字典序对行键进行排序(这种存储格式更适合于记录的顺序读取,充分利用磁盘传输带宽),支持高效的指定行键的单点查询和指定行键范围的范围查询.扫描操作主要用于对非主键属性的查询,它允许指定扫描的数据范围,可以在查询过程中指定限定条件来过滤目标结果集.扫描操作允许对任意一个数据属性进行查询,因而比基于行键的查询更灵活,但是对非主键属性的扫描操作代价很大.基于行键检索的时间复杂性是 $O(\log N)$,甚至如果使用 Bloom filter 可以达到 $O(1)$,而扫描操作的时间复杂性是 $O(N)$.大数据应用的数据记录规模在千万至亿级以上,对全表进行非主键扫描的时间开销是不可接受的,这导致了 HBase 在很多实际应用中难以满足高实时性的查询需求.为此,一些研究工作致力于改善 HBase 非主键查询的性能.这些工作按照研究目标可以分为如下几个方面:

(1)面向选择查询优化的非主键索引方法

针对大数据上基于非主键属性的选择查询(包括单点查询和范围查询)需求,华为公司研发并开源了非主键索引查询系统 Hindex①.它采用基于Region 的局部索引模型,通过对 HBase 表的每个 Region 建立单独的索引表来提高非主键查询的效率.查询请求发送到每个 Region 服务器,然后通过 Region 上的索引表将结果集过滤并返回.Hindex 查询需要访问所有的 Region,由于绝大多数检索任务的目标记录数量相对较少,在分布式集群中并行地执行该任务往往造成很多未存储任何目标记录的计算节点也触发了检索过程,而最终却返回空集.在检索任务频繁的情况下,这一并行执行过程会耗费大量不必要的计算资源,最终将降低系统的并发查询吞吐量和系统的查询效率.

NGDATA 公司的 HBase-indexer② 也提供了 HBase 上的非主键索引.HBase-indexer 将 HBase 的更新数据异步发送到索引服务器上,在索引服务器上分析数据并生成对应的索引数据.索引服务器会定期将索引数据推送到 SolrCloud③ 服务集群上,查询则通过访问 Solr 服务来定位 HBase 上的内容.这种索引机制周期性地对索引进行异步更新,索引的时效性稍差,在面向实时应用时难以有效地满足需求.

(2)面向范围查询优化的非主键索引方法

Interval Index[5] 是希腊 Patras 大学提出的基于 HBase 的非主键范围查询索引,它通过 MapReduce 构造 Segment Tree[6] 索引结构并保存在内存中,同时将索引树中存储的每段范围的上界保存在 HBase 表中,以支持有效的范围查询.Interval Index 的可扩展性好,但是对单点查询,线段树退化成二叉树,索引的空间开销和维护代价都很大.

中国科学院研究团队提出在分布式有序表上提供高性能、低空间开销和高可用的基于非主键的多维范围查询索引方案 CCIndex,并分别在 HBase 和 Cassandra 上实现了原型系统[7-8].其主要思想是:充分利用数据的多个副本,在每个副本上分别建立基于不同非主键属性的聚簇索引,将非主键查询中大量的随机读转换成基于索引表主键的顺序扫描.CCIndex 在多维非主键范围查询上取得了显著的性能提升,同时优化了索引的空间开销.

Bloom filter 是 Bloom 在 1970 年提出的二进制向量数据结构,它具有很好的空间和时间效率,被用来检测一个元素是不是集合中的一个成员.Bloom filter 的局限性是它仅可以应用于单值查询(point

① Huawei hindex. https://github.com/Huawei-Hadoop/hindex

② NGDATA hbase-indexer. https://github.com/NGDATA/hbase-indexer

③ Apache Solr. https://lucene.apache.org/solr/

query). 在文献[9]中，微软提出了一种新的数据结构 Adaptive Range Filter (ARF). 从本质上来说，ARF 是适用于范围查询的 Bloom filter，可以判定出一个集合中的全部元素都不在特定的域值空间内. ARF 继承了 Bloom filter 的优点：快速、节省空间，它可以为具有可排序特性的属性列建立索引，并可以探测一个属性列上是否有满足范围查询需求的记录，以此来优化范围查询的性能.

(3) 采用内存缓存的查询优化方法

随着内存和固态硬盘性价比的持续提升，大数据上的缓存算法成为近几年的研究热点. TBF[10] 在 HBase 数据存储上提出了基于固态硬盘的缓存替换策略. TBF 算法结合 CLOCK 和 Bloom filter 的优点，充分利用 HBase 数据存储行键有序的特点，有效地降低缓存替换算法的两个元数据("数据记录是否被缓存"和"最近访问信息")的空间开销. TBF 算法的空间效率高，但它无法避免 CLOCK 算法固有的缺陷，不能量化数据最近被访问的频率，并且不能保存一个 CLOCK 周期以外的数据访问信息.

高效、准确地区分冷热数据是缓存算法的基础. 将频繁访问的热数据缓存，可以大幅度提高查询的性能. 微软在文献[11]中提出了如何区分持续数据访问中的冷热数据的方法. 通过指数平滑算法将数据记录的访问频率量化，并动态调整冷热数据的门限至最终收敛. 由于针对的是离线日志的数据访问分析，所以微软在这篇文章里提出了回溯(backward)算法和并行回溯算法，回溯算法只适用于日志数据的离线分析，因为只有这样才可以高效地从新时间到旧时间逆向分析日志，使冷热门限提前收敛.

本文研究提出了一种分层式非主键索引方案：采用全局索引模型的非主键持久化索引层，为索引机制提供良好的可扩展性和可靠性；内存热点索引数据缓存层通过热度累积的缓存替换策略为非主键查询提供实时、准确的热点数据缓存；同时，基于一致性哈希的分布式内存缓存结构为 HiBase 缓存层提供了良好的可扩展性和容错性.

3　分层式索引存储机制

3.1　非主键持久化索引存储模型

为了避免对 HBase 非主键查询时的全表扫描，提供快速的非主键查询能力，HiBase 为保存在 HBase 用户表中的非主键属性建立索引表，并将索引表保存在 HBase 中，借助 HBase 获得良好的可扩展性和容错性. 每个 HiBase 索引表用来存储管理用户表中的某个待查询非主键属性的索引. 我们为用户表中待建立索引的非主键属性定义如下格式的索引表主键：

〈用户表索引列名(简短别名)，
用户表索引列值，用户表主键〉

其中，用户表索引列名是用户表中被索引属性的名称. 通过将列名映射到一个简短的别名，如"Age"映射到"a"，可减少索引表主键存储空间的开销. 在索引表主键中存储用户表主键有两个作用：一是保证了索引表主键的唯一性；二是提供 HBase 用户表中被索引记录的地址，通过用户表主键，可快速获得用户表中被索引的记录.

图 1 给出了一个 HBase 用户表示例以及以 Age 属性为索引主键的索引模型和索引表示例. 该示例中，索引表的主键形如"a,12,Tom"，其中，a 为 Age 属性的简短别名；12 是用户表数据记录 Tom 的 Age 值，也就是索引列值；Tom 是该记录在用户表中对应的主键. 与关系数据库中的部分索引(partial index)一样，HiBase 以查询属性为索引表主键，索引表包含用户表的部分字段，通常只将查询中可能需要辅助性访问的字段存放在索引表的非主键属性中. 这是为了方便对查询中需要访问的辅助性字段提供快读访问，避免再次访问用户表而导致的二次磁盘访问. 本例中，索引表中的 value 部分所保存的如{Income：500}，表示查询 Age 时可能需要访问 Income 列.

用户表

Name	Age	Income
Bob	21	3000
Jerry	30	10 000
Ron	30	20 000
Simon	42	22 000
Tom	12	500

索引表

key	value
a,12,Tom	{Income:500}
a,21,Bob	{Income:3000}
a,30,Jerry	{Income:10 000}
a,30,Ron	{Income:20 000}
a,42,Simon	{Income:22 000}

图 1　HiBase 的用户表和索引表结构

以上描述了单个非主键属性索引的情形. 在实际应用中存在多个非主键属性组合查询的需求，为此，类似于数据库中的多字段索引，需要构建多个非主键属性列的组合索引. 对于多个非主键属性列的组合查询情况，HiBase 会基于多个查询属性列建立索引. 举例来说，当查询请求的条件中同时包含"Age"和"Income"时，我们建立以这两个非主键列为主键的索引表. 索引表的主键构成形如"a,12,i,500,Tom". 其中 a 是列名"Age"的别名，i 是列名"Income"的别名. 在这里，我们使用逗号做分隔符(HiBase 的分隔符是可配置的，对于整数、浮点数类

型和固定长度字符串类型,系统指定使用'\0'作为分隔符.对于变长字符串类型,用户需自定义分隔符).在多个属性列上建立了索引后,组合查询就转换成基于索引表主键的查询,索引过程和查询过程与单属性查询基本相同.

3.2 分层式索引存储模型与索引缓存

上述索引表将为 HBase 表实现索引数据的持久化存储.由于索引数据是存放在 HBase 中,每次查询访问 HBase 表会涉及到很多磁盘访问,我们进一步考虑把索引中那些访问频率高的索引数据作为热点数据缓存在内存中,形成基于 HBase 和分布式内存的分层式索引存储和查询机制,进一步提高索引的查询速度.HiBase 的分层式索引存储模型如图 2 所示.

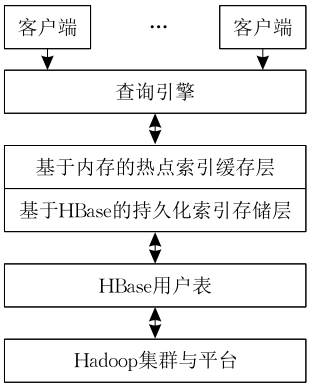


图 2 HiBase 分层式索引存储模型

HiBase 中的内存热点索引数据缓存格式不同于持久化存储中的索引格式,内存缓存索引的主键格式为

〈用户表索引列名(简短别名),用户表索引列值〉

其中,用户表索引列名和用户表索引列值的含义与持久化索引存储层中的相同.内存索引构建的基本思路类似于倒排索引,内存索引缓存层中的每个索引主键对应着一个具有相同索引列值的索引记录集合,该集合包含了与该索引值对应的所有索引表数据记录.与持久化索引存储层一样,集合中也包含了可能需要访问的其他非主键属性.因此,完整的内存索引数据格式如下:

索引主键:〈用户表索引列名(简短别名),用户表索引列值〉

索引集合:{〈用户表主键,{〈频繁访问列名,频繁访问列值〉}〉}

图 3 是内存缓存层的索引结构示例,该例中缓存了 Age 属性下的两个年龄值"21"和"30",分别对应于索引内存缓存层的主键"a,21"和"a,30",其中,a 是 Age 的简短别名.当以年龄为 30 进行查询时,根据索引内存缓存层主键"a,30"的哈希值找到对应索引在内存中的存储地址,可以得到该年龄值所对应的索引集合{〈Jerry,{Income:10 000}〉,〈Ron,{Income:20 000}〉}.由该集合对应的用户表主键到用户表中可获得相应的用户数据记录.在实现上索引内存缓存层数据存储在 Redis 内存数据库中,并由 Redis 自动完成上述哈希和快速查询过程,这一层全内存化的查询相当高效.

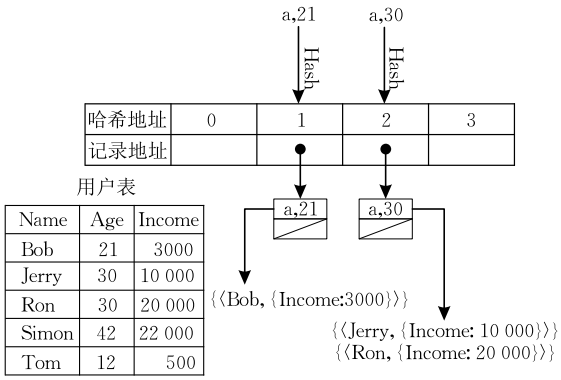


图 3 内存缓存层的索引结构示意图

分层式索引存储模型的查询过程是:首先到索引内存缓存层查询热点索引数据,若缓存中没有命中,则将查询转发到索引持久化存储层进行检索.可以看出,通过将索引热点数据缓存在内存中,部分查询可以直接在内存中命中结果集,从而降低了磁盘访问开销,提高整体查询性能,这对于具有倾斜的数据访问分布特性的应用来说尤为有效.

3.3 基于一致性哈希的分布式内存缓存

HiBase 利用 HBase 服务器节点上的分布式内存来存储管理所有的索引热点数据.我们引入了一致性哈希[12]来完成索引热点数据在分布式内存中的存储管理.一致性哈希是 1997 年由麻省理工学院的 David Karger 提出,目的是为了修正简单哈希算法在分布式环境中存在的节点更新带来的巨大开销等问题,使得哈希算法可以在分布式环境中真正得到应用.一致性哈希在许多分布式系统中得到了广泛的应用,如 Dynamo[4]、Cassandra[3].

一致性哈希为 HiBase 的索引内存缓存层提供了良好的可扩展性.索引内存缓存层的可扩展性是指当索引缓存层的内存使用率偏高时,通过加入新的服务节点即能够实现索引缓存层容量的动态增加.在动态变化的内存环境中,单调性是可扩展性的可靠保证[12].单调性是指如果已经有一些内容通过哈希方法分配到对应节点的缓存,此后又有新的节

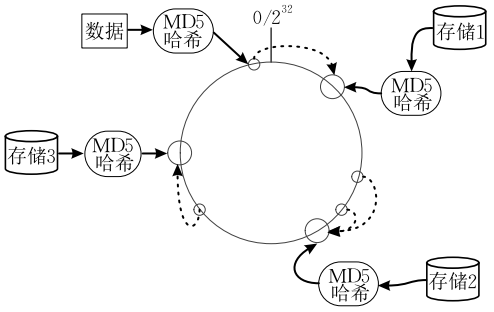
点加入到系统中，那么哈希的结果应能够保证原有已分配的内容要么被映射到新节点的缓存中，要么仍然被映射到原先所在节点的缓存中，以尽量减少数据迁移带来的开销. 例如，对于最简单的线性哈希：

$$Address=ax+b \bmod (N),$$

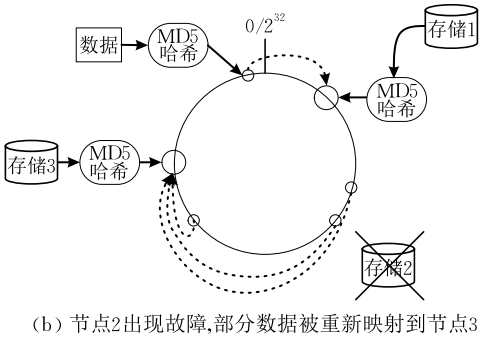
其中，N 表示全部缓存区的个数，在本文的分层式索引系统中即表示索引缓存层的节点数量.

倘若不使用一致性哈希管理分布式的索引缓存，由于服务进程的加入和退出而使得缓存节点个数 N 发生变化时，原来所有的哈希结果均会发生变化，意味着所有的映射关系需要在系统内全部更新. 节点的加入和退出将带来极大的计算和传输开销.

在 HiBase 分布式内存缓存中，使用一致性哈希来确定数据所在的服务器节点. 一致性哈希的基本原理如下：使用某种哈希函数将所有数据映射到圆环边上的某一点上（如果使用 32 位地址空间，那么圆环上总共存在 2^{32} 个点）. 同时使用另一个相同或不同的哈希函数将每个存储节点映射到该圆环边上的伪随机分布点上. 当查找数据所在的节点时，一致性哈希算法会从圆环上该数据的映射点开始，沿顺时钟方向进行搜索，找到的第 1 个存储节点即为数据所在的节点. 在节点发生变化时（如节点失效或节点加入），只有和变化节点相邻的节点数据需要迁移，从而可减少节点的加入和退出带来的计算和数据传输的开销. 例如，当图 4(a)中存储节点 2 出现故障时，原先映射到该节点的数据会映射到顺时钟方向遇到的第 1 个存储节点上，如图 4(b)所示部分数据被重新映射到存储节点 3. 而增加存储节点的时候，数据映射关系的变化与上述的过程正好相反，新节点将会负责管理其哈希地址到它逆时针方向上第一个存储节点之间的所有数据.



(a) 一致性哈希的存储节点映射



(b) 节点2出现故障,部分数据被重新映射到节点3

图 4　基于一致性哈希的分布式索引内存缓存存储机制

可以看出，当存储节点发生变化时，只需要迁移小部分的数据即可. 而且通过将存储节点映射到圆环的伪随机分布点上，能够有效地保证各存储节点之间的负载均衡.

HiBase 的内存缓存需要通过两次哈希来找到对应索引记录的真实位置：第 1 次通过图 4 所示的一致性哈希找到索引数据所在的服务器节点；第 2 次则通过 Redis 的哈希机制找到节点内的索引数据地址.

3.4　支持基于内存缓存范围查询的值表

HiBase 在索引持久化存储层中存储值表，用来有序地记录和存储索引属性所有取值的集合，以支持高效的基于索引内存缓存的范围查询，如图 5(a)所示.

通常情况下，一个索引属性值经常会对应多条用户表记录. 在持久化存储层的索引表中，HiBase 在索引表主键中加入用户表主键以维护其唯一性（如 3.1 节所示）. 在值表中，我们只保存索引列值，因此值表的记录条数会比用户表小得多. 用户对分层式索引存储系统进行范围查询时，值表会被频繁访问，因此值表会被底层文件系统缓存在内存中，可以大大提高值表的查询效率. 这个缓存由文件系统管理.

用户表

Name	Age	Income
Bob	21	3000
Jerry	30	10 000
Ron	30	20 000
Simon	42	22 000
Tom	12	500

值表

Age
12
21
30
42

(a) HiBase的Age列值表

用户表

Name	Age	Income
Bob	21	3000
Jerry	30	10 000
Ron	30	20 000
Simon	42	22 000
Tom	12	500

值表

Income
0−5000
10000−15000
20000−25000

(b) HiBase的粒度值表优化方案: Income列值表

图 5　HiBase 的索引列值表

当索引列的基数（即索引列取值的实例个数）不大时，上述机制保证值表能有效地工作. 当索引列取值很多或无限多（比如取值为范围变化大的整数、字

符串或浮点数)时,值表会带来大量的空间开销.因此我们提出粒度值表优化方案:将索引列取值进行粒度分段,每个分段范围内的多个索引列值对应一条值表记录,如图 5(b)所示.

HiBase 将一条值表记录对应的索引列取值范围内的索引记录存储在同一个内存缓存索引集合(如 3.2 节所述的内存缓存索引结构)中.因此,采用粒度值表优化方案后,仍然是一条值表记录对应一个 Redis 集合.

HiBase 在处理范围查询时,首先会查询值表,得到落在查询范围内、满足查询条件的所有值表记录.然后以值表记录值为请求发起批量的单值查询,并汇总查询结果.如前所述,粒度值表优化方案中仍然是一条值表记录对应一个 Redis 集合,因此不影响范围查询过程.只是查询结果可能是 Redis 集合的部分子集,需要过滤筛选.依赖于 Redis 高效的查询机制,以及可配置的值表粒度,过滤筛选是非常高效的.详细的查询执行过程见 5.3 节数据查询过程中的范围查询.

4 索引热点数据缓存替换策略

缓存是数据查询优化常用和行之有效的方法,缓存技术一直是数据存储和查询的研究热点,通过将部分数据保存在内存中,可以提高随后对数据重复访问的效率.通常缓存的容量远远小于保存全部数据的磁盘数据库的容量,所以当缓存满了之后需要选择合适的牺牲者淘汰出缓存,这就是缓存替换策略.

HBase 也有自己的缓存策略,即 HBase 的块缓存(BlockCache).HBase 在读数据时会以块(Block)为单位进行缓存,用来提升读性能,块的默认大小是 64 KB.读请求先到写缓存(MemStore)中查询,查询失败则继续到块缓存中查询,再查询失败就会到磁盘上的 HBase 表中读取,同时会把读取的结果放入块缓存.在配置文件中会指定块缓存的容量,当块缓存的容量已经被使用 85%以上的时候开始采用缓存替换策略淘汰数据,淘汰过程在块缓存容量占用比为 75%的时候停止.然而,HBase 的缓存策略是面向系统的读写性能提升的,而不是面向特定的数据表的.

HBase 采用的是最近最少使用(Least Recently Used)的缓存替换策略来淘汰块.LRU 算法是基于如下假定:最近被访问的数据在近期最有可能被重复访问,而如果数据很长时间未被使用,则在最近的未来被访问的概率很低.LRU 算法只关注数据项最近一次访问时间的差异,没有关注数据项访问的频繁程度不同,也就是说,LRU 算法没有考虑数据的累积热度.

4.1 HiBase 索引热点数据缓存基本思想

在 HiBase 中,我们提出了热度累积的缓存替换策略.它基于与 LRU 算法相同的假设:最近被访问的数据在最近的未来最有可能被重复访问.其基本设计思想是周期性地累积缓存索引集合被访问的次数,并将访问次数周期性地累积成热度保存在缓存元数据中.进而,对所有记录的累积热度排序,选择累积热度 TOP-K 的索引记录缓存到内存中,这就是热度累积的缓存替换策略.

图 6 给出了 HiBase 的分层式数据存储结构.用户表、索引表和值表全部存储在基于 HBase 的索引持久化存储层中.在内存缓存中,我们保存索引热点数据.索引缓存存储结构是基于 Redis 的集合(Set),Redis Set 也是以〈key,value〉格式来组织数据.热点数据的索引主键(如 3.2 节描述)做 Redis Set 的 key,而索引集合(如 3.2 节描述)作为 Redis Set 的 value 保存在内存缓存中.显然,具有相同索引列值的记录被绑定在同一个集合中,基于索引列值的查询命中是以集合为单位的.同时,它们也是热度累积的基本单位,每个集合都会累积它在一个计算周期内的访问次数.

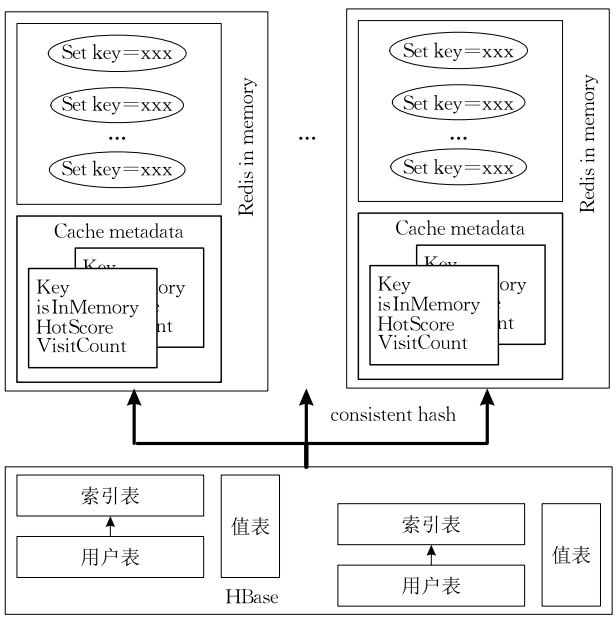


图 6 HiBase 的分层式数据存储

4.2 热度累积的缓存替换策略

在 HiBase 中,热度累积缓存替换策略的热度

计算公式如下：

$$score_n = \alpha \times \frac{visitCount}{countPeriod} + (1-\alpha) \times score_{n-1},$$

其中 $0 \leq \alpha \leq 1$. 式中的 $countPeriod$ 即热度计算周期，$visitCount$ 指当前热度计算周期中，该索引集合被访问的次数. 历史热度 $score_{n-1}$ 则反映集合累积的历史热度. 参数 α 是衰减系数，用来确定当前周期累积的热度和历史热度在 $score_n$ 中各自所占的权重. α 越大，则最近的访问在数据访问热度中所占的权重越大，历史访问记录对数据热度的影响越小，反之亦然. 集合的历史热度在本计算周期内以系数 $(1-\alpha)$ 的速率衰减，经过多次迭代，更早计算周期的累积热度经过了更多次衰减，所以早期的累积热度对数据热度的影响不断降低.

为了降低热度计算带来的计算和更新开销，在执行查询请求时，内存缓存的服务进程将会对访问到的每条索引数据记录本周期内的访问次数，此时并不对内存缓存的数据进行替换. 直到查询请求次数达到 $countPeriod$，即到达热度计算周期时，服务进程触发缓存的更新替换. 按照热度累积公式对所有的记录计算热度，根据热度排序，将热度排序 TOP-K 的集合记录缓存到内存中，集合中包含的记录条数是不固定的，所以选择 TOP-K 时，根据缓存空间能够容纳的记录条数限制计算出热度门限，高于门限的集合被缓存到内存中.

然而，在系统初始阶段，缓存是大量空闲的. LRU 算法在系统初始阶段的命中率提升很快，这是由于 LRU 算法中数据记录是访问即插入缓存，最长时间没有被访问的数据记录会在缓存充满后被淘汰. 所以 LRU 可以快速地进入稳定状态. 而热度累积的访问如果在系统初始阶段通过周期性地计算热度，等被访问数据记录的热度累积到门限时才可以插入缓存的话，那么初始预热阶段的缓存利用率低. 所以我们的热度累积算法在缓存空闲阶段做了优化，只要缓存有空闲，我们就采用“访问即插入”的策略，将所有访问到的记录都插入缓存. 而当缓存充满以后，热度累积的缓存替换策略根据记录的热度累积评分选择“牺牲者”淘汰出内存，选择获得热度高分的记录保存在缓存中.

热度累积的缓存替换策略不仅考虑了数据的访问时间远近，同时考虑了数据的访问频率，所以比 LRU 更准确. 从实验结果看出，热度累积的缓存替换策略明显优于 LRU 算法，和不使用内存缓存策略相比，可以提升 5～15 倍的查询性能.

5　HiBase 的系统设计与实现

5.1　系统总体架构

基于以上的非主键索引模型和技术方法，我们设计并实现了一个基于 HBase 的分层式非主键索引查询系统 HiBase，该系统实现了基于 HBase 的持久化索引存储，基于内存的索引热点数据缓存，以及热度累积的缓存替换策略. HiBase 提供了基于分层式非主键索引的数据查询方法，并对范围查询进行了并行化优化.

HiBase 将整个分层式索引存储系统划分为以下几个模块，系统功能模块划分如图 7 所示.

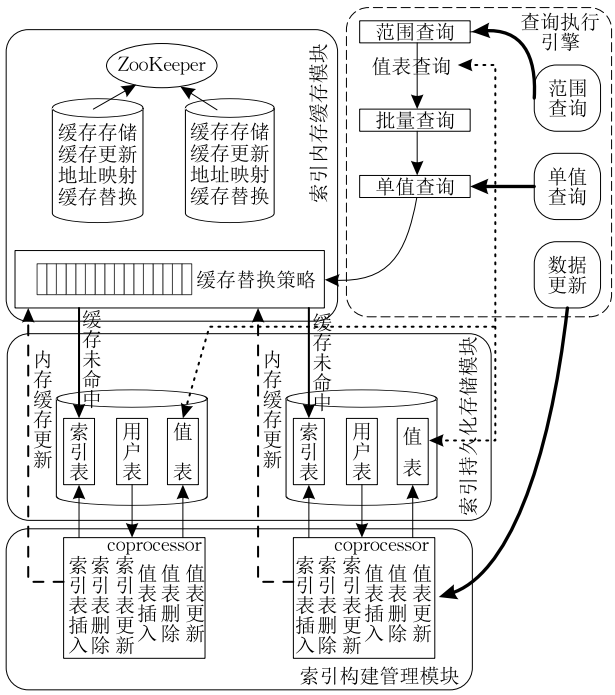


图 7　HiBase 系统架构

(1) 索引构建管理模块. 管理索引的元数据(记录用户表对应的索引表名称、索引列等信息)，并实现针对 HBase 的流式数据和静态数据两种不同特性数据的索引构建方法，包括支持索引表和值表的插入、删除、更新操作.

(2) 持久化存储管理模块. 提供索引表和值表的持久化存储，HBase 为持久化存储数据提供可扩展性和容错性.

(3) 索引内存缓存模块. 管理索引热点数据的缓存存储、更新和地址映射，实现热度累积缓存替换策略，使得最近频繁访问的数据能缓存到内存中.

(4) 查询执行引擎. 将用户的查询请求翻译成系统识别的命令，调用对应的方法执行查询，并将查

询结果汇总返回给客户端.

为了提供索引内存缓存的高可用性,HiBase使用了 Hadoop 环境中的分布式协调管理系统ZooKeeper来可靠地检测分布式内存节点上服务进程的存活状态.索引内存缓存层每个内存节点服务进程会分别建立与 ZooKeeper 的会话,并创建临时Znode来表示自身的存活状态.每个内存节点服务进程从 ZooKeeper 系统镜像中可以观察到其他节点进程的存活状态.系统通过对分布内存节点状态的监测实现内存节点的失效检测和上线处理,从而实现索引内存缓存层的高可用性.

5.2 索引构建过程

根据数据输入方式的不同,索引构建可分为面向流式数据和面向批处理数据的索引构建.索引构建过程都是读取用户表的一条记录,在非主键属性上生成一条索引记录,如果满足缓存条件,同时生成内存缓存层的索引数据.最后将索引数据分别更新到持久化存储层与内存缓存层,并更新值表.

对于流式数据索引构建方法,HiBase 利用Observer类型的 Coprocessor 来构建相关的索引.具体来说是使用 HBase 提供的 RegionObserver 接口的回调函数 *prePut*,插入一条数据之前会被触发调用. *prePut* 方法首先根据索引信息对用户发起的Put 操作进行分析,如果 Put 操作的数据包含有索引列,即包含要索引的数据,则触发索引数据的插入.

由于静态数据一般相对较大,为了能够加快静态数据索引的构建速度,本文利用 Hadoop MapReduce来并行化执行静态数据索引构建. MapReduce 任务首先得到输入〈*Row*,*Result*〉,其中 *Row* 为用户表的行键,*Result* 是通过 *Row* 获得的 HBase 记录.然后根据索引信息生成其对应的索引数据,并将索引数据插入到分层式索引中.整个过程不需要Reduce阶段即可完成,同时由于 HBase 用户表记录之间是相互独立的,所以可以充分利用 MapReduce 提供的并行化处理能力来加速索引构建过程.

5.3 数据查询过程

(1)单值查询

HiBase 通过建立非主键属性上的索引,支持高效的非主键列上的单值查询(point query)和范围查询(range query).

本文所说的单值查询,即查询请求中条件属性被限定取值唯一的查询.例如,对于图 1 中给出的实例,查询年龄为 30 岁的所有记录.对于单值查询,客户端的基本流程如下:

① 从配置文件获取 ZooKeeper 的地址,建立与ZooKeeper 的连接,获取注册的所有服务进程,确定当前提供内存缓存的所有服务进程的位置信息.

② 向内存缓存层的服务进程发起查询请求.若内存缓存层命中,则返回内存缓存层给出的结果,结束查询.

③ 若内存缓存层未命中,则向 HBase 的索引表发起查询请求.获取结果后,返回查询得到的结果,结束查询.

可以看出,如果在内存缓存层命中,整个查询流程都不会访问到磁盘,减少了磁盘访问开销,能够大幅度提高响应速度.另外,对 ZooKeeper 访问获取的内存缓存层服务进程信息会被缓存在客户端,在后续的访问中会进一步减少数据查询的通信开销.

(2)范围查询

本文所说的范围查询,即查询请求中条件属性的取值是范围的查询.例如,对于图 1 中给出的示例,查询年龄在 15～35 岁的所有记录.对于范围查询,内存缓存层通过一致性哈希的方法将数据分布到各个存储节点上来提高查询性能,而哈希破坏了索引列的有序性,所以我们需要记录索引表主键的所有取值,保存在值表中.通过该值表,我们可以获得索引主键某个范围内所有存在的列值.

对于范围查询,客户端的基本流程如下:

① 从配置文件获取 ZooKeeper 的地址,建立与ZooKeeper 的连接,获取注册的所有服务进程,确定当前提供内存缓存的所有服务进程位置信息.

② 从 HBase 中的值表获取查询范围内所有的查询列值.

③ 依次发起单值查询请求.将查询结果汇总并返回.

为了进一步提升查询效率,HiBase 对范围查询进行如下的优化和改进:(i)如果范围查询中的某些索引主键列值是由同一节点管理的,则将这些列值汇总,然后向该节点发起一次批量查询请求;(ii)向范围查询涉及到的多个节点并发地发起查询请求;(iii)当索引缓存层的数据未命中时,节点上的内存索引服务进程并不返回未命中标志,而是将查询请求转发给基于 HBase 的持久存储层,并将持久存储层的查询结果返回给客户端.

优化的范围查询具体流程如下:

① 从配置文件获取 ZooKeeper 的地址,建立与ZooKeeper 的连接,获取注册的所有服务进程,确定当前提供内存缓存的所有服务进程位置信息.

② 根据范围查询的条件，客户端从 HBase 值表中获取范围之间存在的所有索引列值.

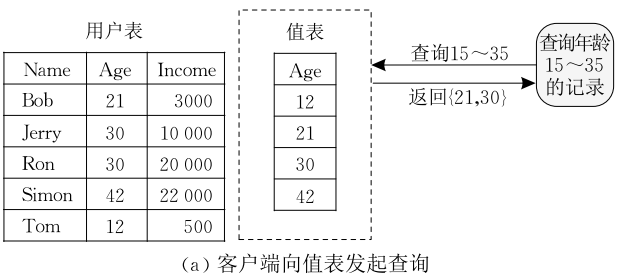
③ 对于所有存在的索引列值，根据一致性哈希算法计算出存储节点地址，从而将所有存在的索引列值与相关的节点地址一一对应起来.

④ 并发地对相关节点发起查询请求，其中，对同一节点发起的多个查询请求将会合并成一个批量请求.

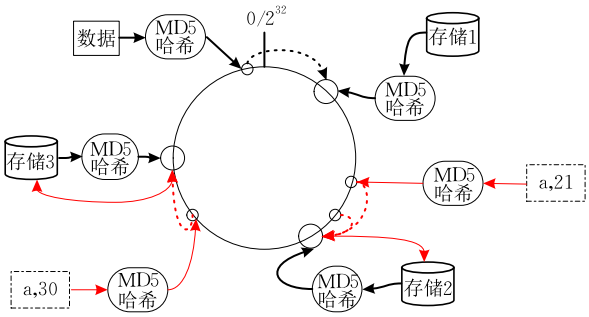
⑤ 各节点上的内存索引服务进程对查询请求进行响应，如果查询的内容在内存中，则直接返回内存中的数据；否则，服务进程将发起对持久存储层的查询，并返回查询结果.

⑥ 客户端汇总从各服务节点返回的查询结果.

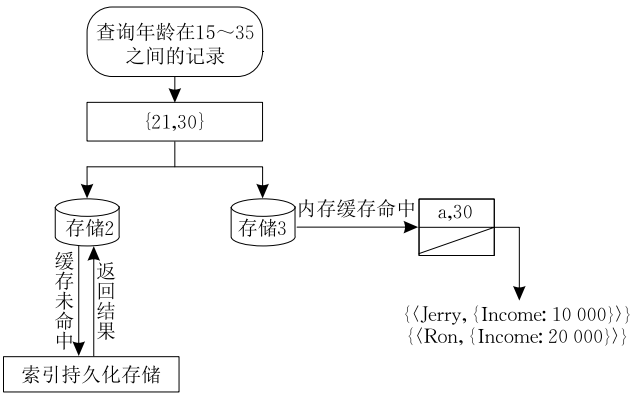
下面给出一个范围查询的实例：查询年龄在 15～35 岁的所有记录，来完整地描述整个基于分层式索引的范围查询流程，查询过程如图 8 所示.



(a) 客户端向值表发起查询



(b) 根据一致性哈希算法计算索引缓存的内存节点



(c) 客户端并发地向内存节点发查询请求

图 8　基于分层式索引的范围查询流程

① 图 8(a)是客户端根据查询条件向值表发起查询的过程. 首先客户端发起值表查询，HBase 返回值表中存在的满足查询条件的 3 个索引列值，表明用户表的年龄属性列中在 15～35 之间的值有 21 与 30.

② 客户端根据一致性哈希算法计算返回的索引列值与索引缓存层中节点之间的映射关系，如图 8(b)所示，21 映射到内存索引服务进程存储 2 上，30 映射到内存索引服务进程存储 3 上.

③ 客户端并发地向服务进程存储 2 与存储 3 发起查询. 如图 8(c)所示，客户端同时向存储 2 与存储 3 发起了对 21 与 30 的查询. 在存储 3 中，内存缓存直接命中“a，30”，于是直接返回结果. 而在存储 2 中，内存缓存并未存储索引记录“a，21”，此时存储 2 将此查询请求转发到基于 HBase 的持久存储层，最终将 HBase 的结果返回给客户端.

④ 最后，客户端汇总来自存储 2 与存储 3 的查询结果，完成此次范围查询的流程.

与单值查询一样，缓存命中率的提高会降低范围查询的响应时间. 范围查询中对值表的访问是 HiBase 为哈希索引付出的代价，但是由于每次范围查询只需要访问一次值表，其访问代价占整个范围查询的比重是较小的.

6　实验与性能分析评估

6.1　实验环境与数据集

我们从数据查询性能、数据扩展性、集群规模扩展性分别测试了 HiBase 的性能. 我们的集群测试环境共有 10 个节点，集群的节点配置参见表 1.

表 1　计算节点配置信息

属性	配置信息
CPU	2 路 4 Core Intel Xeon E-5620 2.4GHz
Memory	24GB
Disk	6 TB 7200RPM SATA II
Network	Bandwidth 1Gbps
OS	Red Hat Enterprise Linux Server 6.0
JVM Version	Java 1.6.0
Hadoop Version	Hadoop 1.2.1
HBase Version	HBase 0.94.14
ZooKeeper Version	ZooKeeper 3.4.5

实验采用的数据集包括布朗大学的大数据基准测试和国内电信运营商的电话用户上网记录数据.

首先，我们用电话用户上网记录数据测试不同系统的查询响应时间，该数据集共 1 072 868 115 条记录，每条数据记录不等长，平均约是 200 个字节，

10 个属性. 查询需求包括下面 4 个：

① 根据用户号码 MDN 查询某一个时间段内的详单列表.

② 根据 IP 查询某个时间段内使用该 IP 进行上网的用户列表.

③ 根据 IP＋PORT 查询某个时间段内使用该 IP＋PORT 进行上网的用户列表.

④ 根据 URL 查询某个时间段内登录该 URL 的用户列表.

以上查询需求都是时间段上的范围查询，其中，需求 1 是在用户表主键上查询，不用建立索引；需求 2 和 4 是单个属性上的非主键查询，建立查询属性上的索引；需求 3 是两个属性上的组合查询，建立组合索引. 针对以上查询需求，我们一共设计了 10 个测试用例，每个查询测试用例分别测试缓存未命中的冷查询响应时间和缓存命中的热查询响应时间.

然后，我们再进行布朗大学的大数据基准测试实验. 布朗大学的大数据基准测试[①]是布朗大学在文献[13]中为了对比关系数据库和 Hadoop 在大数据上的查询性能而提出的基准测试. 我们用该基准测试在 4 个节点上对数据集进行 10000 次符合齐夫分布的持续查询(包括点查询和范围查询)，测试在不同的数据缓存比例下缓存的命中率和查询响应时间.

6.2 实验结果及其分析

(1) 非主键索引查询性能对比实验与结果分析

我们用 10 亿条电话用户上网记录数据对 HiBase 实现的非主键索引的性能进行对比测试. 对比如下 3 种情况：① 不带任何非主键索引的标准 HBase 表上进行扫描的查询性能；② 华为公司的开源 HBase 非主键索引系统 Hindex 的查询性能；③ HiBase 基于热度累积缓存算法的非主键索引的查询性能. 图 9 给出了 4 个测试用例的查询响应时间性能对比. 图 9 中的横坐标分别是查询获得的结果集大小，纵坐标是查询响应时间，由于 HBase 和 HiBase 的查询响应时间数量级相差太大，无法用等比例坐标，在这里使用指数指标.

实验结果可以看出，HiBase 的查询响应时间大大优于无非主键索引的标准 HBase 和开源的 Hindex 非主键索引系统的查询性能. 对于结果集很小(极端情况下，结果集大小为 0)的查询，HiBase 和无非主键索引的标准 HBase 的扫描方法相比，冷查询的性能提升达到 116912 倍，热查询的性能提升可以达到 131732 倍. 对于结果集较小(结果集为 1155)的查询，冷查询的性能提升达到 3082 倍，热查询的性能提升达到 17751 倍. 对于结果集较大(结果集为 97515)的查询，冷查询的性能提升是 65 倍，热查询的性能提升是 343 倍. 和开源的 Hindex 非主键索引系统相比，HiBase 热查询的性能提升可以达到 5～20 倍. 从 HiBase 和标准 HBase 性能对比来看，结果集小的查询性能提升效果更明显，这是因为对于标准 HBase，不论结果集大小，非主键属性上的查询都需要遍历全部数据，而 HiBase 上的非主键索引可以直接定位到记录，查询响应时间随着结果集的大小而增长.

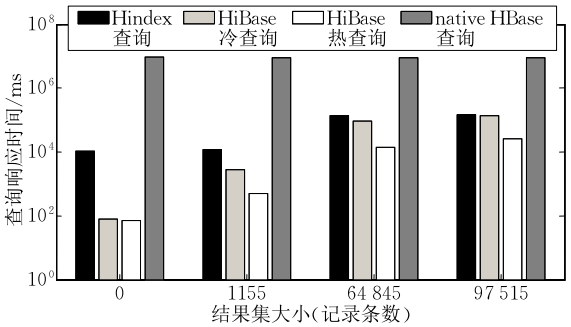


图 9 非主键索引的查询性能对比

(2) 缓存命中率查询性能对比实验与结果分析

缓存命中率是指在批量查询中，在缓存中获得结果集的数据查询个数占全部数据查询个数的比率. 提高缓存命中率会减少访问磁盘的次数，有效降低查询响应时间，因此命中率是缓存算法最重要的性能评价指标. 命中率越高，缓存系统的查询性能越好. 我们用布朗大学的大数据基准测试对 HiBase 进行 10000 次符合齐夫分布的持续查询来测试缓存性能，数据规模为 10000000 条记录. 图 10 给出了 HiBase 的热度累积缓存算法和 LRU 算法的命中率对比，图 11 给出了查询响应时间对比.

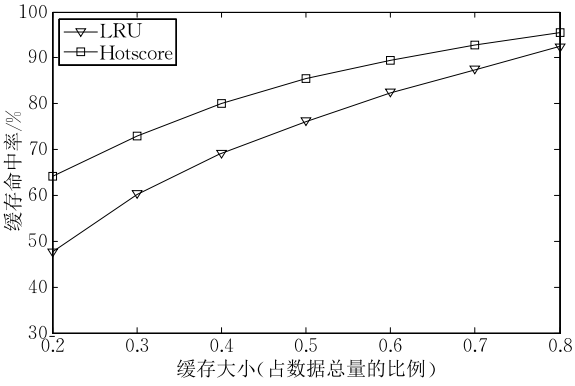


图 10 热度累积缓存算法和 LRU 的命中率对比

① Benchmark of Brown University. http://database.cs.brown.edu/projects/mapreduce-vs-dbms/

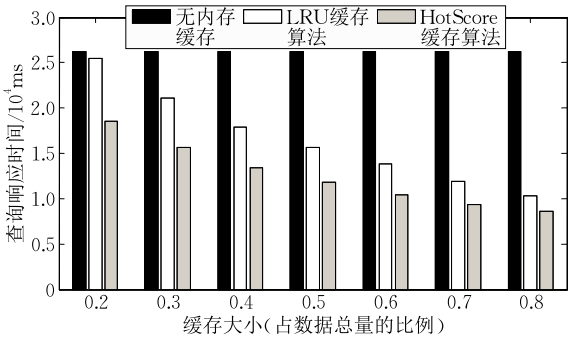


图 11　缓存算法的查询响应时间对比

从图 10 中可以看出，缓存空间增加，命中率提高. 热度累积缓存算法的命中率高于 LRU，尤其是在数据缓存比例较低的情况下. 如数据缓存比例为 0.2 时，热度累积缓存算法的命中率大约比 LRU 的命中率高 16%，因为热度累积缓存算法的热度累积机制能够更精确地记录数据的冷热程度，将热点数据缓存到内存中. 对于大数据应用，缓存空间受内存限制，热度累积缓存算法在缓存比例不高的时候正好可以充分发挥优势，提高大数据下的查询性能.

HiBase 是为了实现大数据上的高效非主键索引而实现的系统，我们在 HiBase 上做了无内存缓存、LRU 缓存替换策略、热度累积的缓存替换策略的 10000 次符合齐夫分布的持续查询，并记录查询响应时间，如图 11 所示. 可以看出，随着缓存空间的增大，命中率不断提升，使用两种替换策略的索引查询时间都在随之降低. 由于热度累积的缓存替换策略在命中率上的优势，查询响应时间明显优于使用 LRU 替换策略的查询性能，提升幅度从缓存比例为 0.2 时的 30%到缓存比例为 0.8 时的 67%.

(3)冷热查询对比实验与结果分析

我们在 10 亿条电话用户上网记录数据上做了 10 个测试用例的冷热查询性能对比实验. 每个测试用例测试了第一次冷查询和随后的缓存命中热查询的查询响应时间. 性能对比实验结果如图 12 所示，这里我们的查询响应时间是单次查询的性能指标，横坐标是查询得到的结果集大小(结果集包含的记录条数).

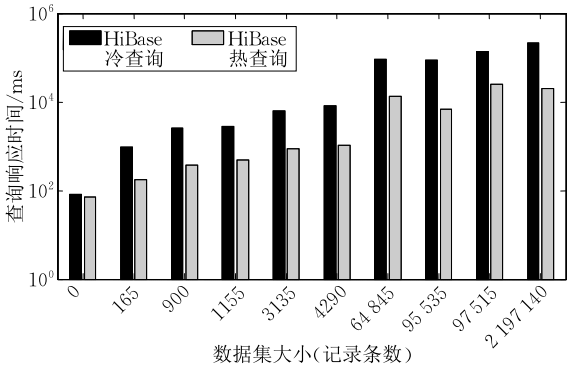


图 12　电话用户上网记录数据查询响应时间

可以看出，每个用例的第 1 次冷查询的时间都是最长的，第 2 次热查询由于缓存命中，所以查询响应时间缩短 5～15 倍(注：图 12 中的查询响应时间是指数坐标).

(4)扩展性实验与结果分析

我们通过布朗大学的大数据基准测试验证了 HiBase 在不同数据规模下的查询性能，如图 13 所示. 可以看到，部署在 10 个节点上的 HiBase 在数据规模从 10000000 扩展到 50000000 条数据记录时，查询时间保持线性增长，这验证了 HiBase 在数据可扩展性方面有着良好的表现. 查询响应时间和结果集大小成正比，因为在查询响应时间中，主要开销是对查询结果集的访问. 在数据集中，数据是符合均匀分布的，随着数据行总数的增长，查询结果集也随之增加，因此查询响应时间会与数据行总数的大小成正比.

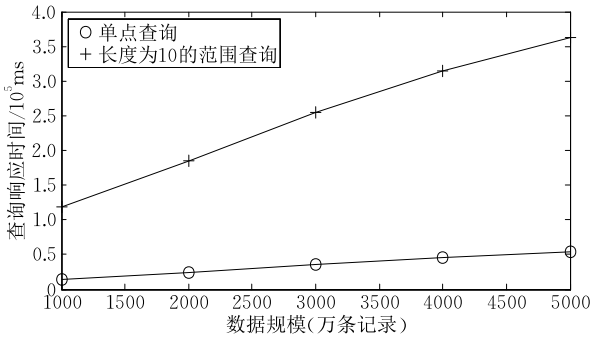


图 13　数据规模增大的查询响应时间对比

同时，我们在 20000000 条数据记录上进行了 10000 次范围长度为 10 的范围查询可扩展性测试，如图 14 所示. 本实验不使用单值查询的原因是：在进行单值查询时，没有并发的过程，所以节点的变化几乎不影响查询时间. 可以看出，随着节点数量的增

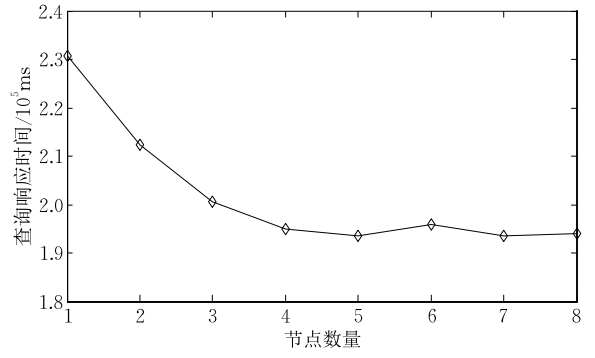


图 14　节点数量增大的查询响应时间对比

加,查询响应时间逐渐降低.在范围查询中,查询会根据范围内存在的索引列值向各节点发送查询请求,查询被并行执行后汇总结果,所以查询响应时间会随着节点的增加而降低.另一方面,索引列值是通过一致性哈希算法映射到各个节点上的,我们的测试实例范围固定为 10,所以当节点个数增加到一定程度之后,查询响应时间趋于平缓.

(5)索引构建的时间和空间开销分析

HiBase 非主键索引系统是在数据导入阶段创建索引,我们在电话用户上网记录数据上测试 HiBase 系统的数据导入时间来衡量系统的索引构建时间开销.同时我们测试了 HBase 的数据导入时间作为对比.数据导入时间的对比实验结果如表 2 所示.

表 2 HiBase 和 HBase 的数据导入时间对比

数据规模(记录行数)	数据集大小/GB	HBase 数据导入时间/ms	HiBase 数据导入时间/ms
6501311	1.4	113352	116768
1072868115	225.0	120382424	135297604

在电话用户上网记录实验数据集上我们分别做 1.4GB 部分数据集(6501311 条记录)和 225GB 全部数据集(1072868115 条记录)的数据导入时间对比测试.可以看出,HiBase 的数据导入时间分别是 HBase 的 1.03 和 1.12 倍,因为 HiBase 在将用户表插入到 HBase 的同时,触发 HBase Coprocessor 并行地构建索引记录并插入到索引表中.实验结果表明,HiBase 为建立索引所带来的时间开销是完全可以接受的.

空间开销上,索引数据与原始用户表数据的占比不是绝对固定的,而是依赖于用户表的结构以及索引属性列值的占用空间.在本文的电话用户测试数据中,225GB 全部数据集的用户表和索引表空间开销总和为 575GB,其中 225GB 文本数据导入到 HBase 后的用户表空间开销为 238GB.针对 6.1 节中提到的查询需求 1～4(其中查询需求 1 是用户表主键查询),我们在 HiBase 中为查询需求 2～4 建立非主键索引.索引的空间开销总和为 337GB,单个索引的平均开销约为 112.3GB,故单个非主键索引表的空间开销约为用户表的 47%.对于采用廉价的分布式存储、具有优异的存储空间可扩展性的 Hadoop 和 HBase 系统来说,构建一个十多个节点的 Hadoop 集群即可获得数十 TB 的存储容量.因此,通过为每个非主键索引提供约一半用户表的空间开销而达到数十至数百倍以上的查询性能提升是完全值得的.

7 总结和展望

为了提高 HBase 上非主键查询的效率,本文提出了一种基于 HBase 的分层式非主键索引存储模型,并设计实现了分层式索引查询系统 HiBase.HiBase 在 HBase 上持久化存储用户表和索引表,并将索引表的热点数据缓存在内存中.同时我们研究提出并实现了一种基于热度累积的缓存替换策略,通过高效的缓存替换策略,HiBase 获得了优于 LRU 的缓存命中率,进一步提升了 HBase 上非主键查询的性能.

下一步,我们会针对非主键属性上的聚集查询和连接查询进行研究并提出高效的解决方案.同时,引入 SSD 尝试多层热点缓存存储方案.另外,在 HiBase 中为了支持基于内存缓存的范围查询我们引入值表,下一步我们会针对值表做自适应粒度的存储优化,保证值表中的记录对应的集合大小基本均衡,从而提供性能稳定的范围查询.

参 考 文 献

[1] Chang F, Dean J, Ghemawat S, et al. Bigtable: A distributed storage system for structured data//Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Seattle, USA, 2006: 205-218

[2] Lakshman A, Malik P. Cassandra—A decentralized structured storage system. ACM SIGOPS Operating Systems Review, 2010, 44(2): 35-40

[3] DeCandia G, Hastorun D, Jampani M, et al. Dynamo: Amazon's highly available key-value store//Proceedings of the 21st ACM Symposium on Operating Systems Principles. Stevenson, USA, 2007: 205-220

[4] CCF TFBD. 2013 White paper on chinese big data technology and industry development. December, 2013(in Chinese)
(中国计算机学会大数据专家委员会. 2013 年中国大数据技术与产业发展白皮书, 2013)

[5] Sfakianakis G, Patlakas I, Ntarmos N, Triantafillou P. Interval indexing and querying on key-value cloud stores//Proceedings of the 29th IEEE International Conference on Data Engineering. Brisbane, Australia, 2013: 805-816

[6] Bentley J L. Solutions to Klee's rectangle problem. Technical Report, Carnegie-Mellon University, Pittsburgh, 1977

[7] Zou Yong-Qiang, Liu Jia, Wang Shi-Cai, et al. CCIndex: A complemental clustering index on distributed ordered tables for multi-dimensional range queries//Proceedings of the Network and Parallel Computing. Zhengzhou, China, 2010: 247-261

[8] Feng Chen, Zou Yong-Qiang, Xu Zhi-Wei. CCIndex for cassandra: A novel scheme for multi-dimensional range queries in cassandra//Proceedings of the 7th International Conference on Semantics, Knowledge and Grid. Beijing, China, 2011: 130-136

[9] Alexiou K, Kossmann D, Larson P A. Adaptive range filters for cold data: Avoiding trips to siberia//Proceedings of the VLDB Endowment. Hangzhou, China, 2014: 1714-1725

[10] Ungureanu C, Debnath B, Rago S, Aranya A. TBF: A memory-efficient replacement policy for flash-based caches//Proceedings of the 29th IEEE International Conference on Data Engineering. Brisbane, Australia, 2013: 1117-1128

[11] Levandoski J J, Larson P A, Stoica R. Identifying hot and cold data in main-memory databases//Proceedings of the 29th IEEE International Conference on Data Engineering. Brisbane, Australia, 2013: 26-37

[12] Karger D, Lehman E, Leighton T. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web//Proceedings of the 29th Annual ACM Symposium on Theory of Computing. New York, USA, 1997: 654-663

[13] Pavlo A, Paulson E, Rasin A, et al. A comparison of approaches to large-scale data analysis//Proceedings of the ACM SIGMOD Conference. Providence, USA, 2009: 165-178

GE Wei, born in 1979, Ph. D. candidate. Her research interests include query processing, query optimization, distributed and parallel computing.

LUO Sheng-Mei, born in 1971, M. S., senior engineer. His research interests include cloud computing, cloud storage, and big data processing.

ZHOU Wen-Hui, born in 1987, M. S. His research interests include parallel computing, distributed consistency of big data.

ZHAO Di, born in 1990, M. S. His research interests include distributed and parallel computing.

TANG Yun, born in 1991, M. S. His research interests include big data distributed system and parallel computing.

ZHOU Juan, born in 1990, M. S. Her research interests include distributed and parallel computing.

QU Wen-Wu, born in 1979, Ph. D. His research interests focus on big data query optimization.

YUAN Chun-Feng, born in 1963, professor. Her research interests include computer system architecture, multimedia document processing, Web information extraction and integration.

HUANG Yi-Hua, born in 1962, professor. His research interests include big data parallel processing and cloud computing, computer architecture, distributed and parallel computing.

Background

HBase is an open-source key-value storing system in Hadoop Ecosystem of Apache Foundation. Data in HBase are composed of much smaller entities in format of key-value pairs, and HBase organizes the small records into large storage files and stores them on HDFS to provide well scalability and fault tolerance. HBase is designed for fast point queries and sequential scans on row keys. Each data split maintains key-value pairs with ranges of rows sorted by row keys, so it excels at providing key-based or sequential range access. Point queries are supported by providing a row key to find what you are looking for. When faced to non-key query requirement, HBase takes a measurement of sequential scan, that is, scanning data from start to end one by one. It is suitable for reading adjacent key-value pairs and is optimized for block disk access operations that can make full use of disk transfer channels. Whereas, sequential scan misses the basic features of key-value retrieval, such as selection of row keys based on regular expressions. Thus the approach of non-key query leaves much to be desired. Some research works try to provide a solution by building secondary index in HBase.

HiBase is our high-performance storing and management system based on HBase to provide efficient query on non-key column. It builds secondary index according to global secondary index model, which outperforms local secondary index model (adopted by Hindex) for it saves much unnecessary computation overhead. In local secondary index query process accesses index table in all Regions, but some Regions return null result set. Furthermore, HiBase provides efficient memory caching policy, so as to reduce the index's disk access overhead, and as a result, query performance is improved in a large degree. Thus, HiBase offers the realtime or near-realtime capability for big data analysis.

This work is supported by the National Natural Science Foundation of China under Grant Nos. 61223003, 61362006 and the ZTE Foundation of Education Combined with Production and Research.