ISSN 1673-9418 CODEN JKYTA8
Journal of Frontiers of Computer Science and Technology
1673-9418/2016/10(09)-1250-12
doi: 10.3778/j.issn.1673-9418.1507065

E-mail: fcst@vip.163.com
http://www.ceaj.org
Tel: +86-10-89056056

HashMap优化及其在列存储数据库查询中的应用*

母红芬[1],李 征[1+],霍卫平[2],金正皓[2]
1. 北京化工大学 计算机系,北京 100029
2. 北京东方国信科技股份有限公司,北京 100102

HashMap Optimization and Its Application in Column-Oriented Database Query*

MU Hongfen[1], LI Zheng[1+], HUO Weiping[2], JIN Zhenghao[2]
1. Department of Computer Science, Beijing University of Chemical Technology, Beijing 100029, China
2. Business-Intelligence of Oriental Nations Corporation, Beijing 100102, China
+ Corresponding author: E-mail: lizheng@mail.buct.edu.cn

MU Hongfen, LI Zheng, HUO Weiping, et al. HashMap optimization and its application in column-oriented database query. Journal of Frontiers of Computer Science and Technology, 2016, 10(9): 1250-1261.

Abstract: HashMap has been widely used to retrieve big data because of its constant level in average performance of dictionary operations. Block_HashMap (BHMap) is based on C++ HashMap, in which three optimizations are introduced: Hash function selection, conflict resolution and keyword matching. Conflict resolution is the core of optimization, where Block_list, a storage structure based on the chain address method, is proposed to use cache efficiently and save matching time by store hashcode. In the situation of limited bucket number and low data repetition rate, experiments show that although it consumes a small amount of memory, BHMap has a 3.5 times of C++ unordered_map and 10 times of Map in terms of query speed. In column-oriented database, group by and join are the most commonly used, in which bucket keywords, resolving conflict and matching keywords are all Hash based. Finally the application of BHMap in the query of column-oriented database is provided.

Key words: HashMap; group by; join; cache-conscious; cache-oblivious; column-oriented database; BHMap

* The National Natural Science Foundation of China under Grant Nos. 61170082, 61472025 (国家自然科学基金); the New Century Excellent Talents Foundation from MOE of China under Grant No. NCET-12-0757 (教育部新世纪优秀人才支持计划); the Scientific Research Foundation for the Returned Overseas Chinese Scholars, State Education Ministry of China under Grant No. LXJJ201303 (教育部留学回国人员科研启动基金).

Received 2015-07, Accepted 2015-10.

CNKI网络优先出版: 2015-10-29, http://www.cnki.net/kcms/detail/11.5602.TP.20151029.1704.002.html

摘　要: HashMap在基本字典操作中具有常数级别的平均算法时间复杂度,广泛应用于大数据的检索。Block_HashMap(BHMap)基于C++ HashMap,其优化包括三方面:哈希函数选取,冲突解决和关键字匹配。优化核心在于冲突解决时,以链地址法为基础,提出了一种高效利用高速缓存的存储结构Block_List来存储冲突的数据,并且预先缓存哈希值,节省匹配时间。实验证明,在桶数目充足的情况下,BHMap会多消耗少部分内存,但在桶数目有限,数据重复率比较低的情况下,时间性能上相对C++标准模板库中的Map提升10倍以上,比unordered_map快3.5倍以上,且消耗的内存与unordered_map相差不大。在列存储数据库分组和连接查询中,关键字的分桶、解决冲突和匹配操作也都涉及到基于哈希的技术,最终把BHMap应用到列存储数据库的关键查询中。

关键词: 哈希图;分组;连接;缓存感知;缓存不敏感;列存储数据库;BHMap

文献标志码: A　　**中图分类号:** TP311.132.3

1　引言

随着云计算与大数据的广泛应用,大数据集的实时动态分析成为研究热点。高效的数据查询、分析操作是实现该目标的重要技术手段。

在大数据技术中,HashMap查找速度快,查询、插入、删除操作的平均复杂时间度为 $O(1)$,属于常数级别,查询时间与数据集大小无关。在众多数据库如Oracle、MonentDB/X100[1]、C-Store、Microsoft SQL Server 2012[2]等都使用了HashMap。

HashMap是基于哈希表的数据结构,按照 *key* 值给每一个元素分桶,使用哈希函数对 *key* 值操作,返回哈希值,然后再将 (*key*,*value*) 存放到hashcode对应的桶中。哈希函数的选取直接影响分桶的效率,而对于不同的关键字,经过哈希函数分桶,出现了相同的哈希值,就会产生“冲突”。

传统解决冲突的方法是链地址法,即将冲突的数据以指针链表的方式存储在相应的桶后,当系统分配的桶数目不够时,进行rehash,重新申请桶。使用链表方式来存储 (*key*,*value*),在匹配查找过程中,会频繁地读取指针指向的内存。随着数据量的增大,时间消耗也急剧增大;同时rehash策略动态增加桶数目也会引起新的时间开销。

国内外对HashMap的研究主要集中在使用该结构体实现数据的高效查询和插入,大部分直接使用封装好的接口,很少对该结构进行优化。

本文针对上述情况,考虑分桶效率、缓存利用率和匹配效率,对HashMap提出以下优化策略:(1)在哈希函数分桶过程中,使用“按位与”操作代替传统的“取模”操作,缩短定址时间;(2)在解决冲突时,使用“块链地址法”取代现有的“链地址法”,提出存储结构Block_List,减少指针寻址时间;(3)在匹配查找时,在结构体中增加存储哈希值,将匹配字符串 *key* 值改为匹配数值型哈希值,节省查找时间。结合以上优化策略,提出一种数据结构BHMap(Block_HashMap),用其解决传统的HashMap在桶数目有限,数据 *key* 重复率较低情况下的性能瓶颈,最后将BHMap应用于列存储数据库中,以提高查询效率。

本文通过一个基准的计数应用在千万级数据集上验证BHMap的性能,并与C++标准库的Map和unordered_map进行比较,结果显示,当桶数有限,数值重复率较低时,BHMap只在增加少部分内存消耗的情况下,查询速度就能比unordered_map快3.5倍以上,比Map快10倍以上。本文给出了BHMap的优化方法以及调用API,通过在列存储数据库的分组和连接查询中使用该数据结构,说明其在大数据查询中的适用性。

2　背景及相关工作

哈希技术在大数据环境下应用广泛,能实现海量数据的高效插入和查询。本章主要介绍哈希技术的研究进展,哈希技术在列存储数据库分组和连接查询中的应用,以及Cache利用率对数据库查询的重要性。

2.1　哈希技术

哈希技术由于其高效性、不可逆性以及唯一性,

在信息系统的数据存储与访问中占有重要的地位[3-4]。哈希技术将关键字直接映射为存储地址，达到快速寻址的目的，即 $Addr=H(key)$，其中 H 为哈希函数。

文献[5]考虑到在大数据环境下节省内存，使用数组代替链表来提高定位速度，以减少遍历链表的时间开销。由于大多数情况下，数据变化未知，使用固定大小的数组要考虑最坏情况的冲突，会浪费许多不必要的空间，而使用动态数组要花费很多时间在内存分配上。本文提出的方法能够减少空间的浪费，同时保证定位速度。

2.2 哈希技术在列存储数据库查询中的应用

列存储数据库在联机分析处理（online analytical processing，OLAP）、商务智能、数据仓库等决策分析领域逐渐被应用，其将关系表中的数据按字段分开存储，执行查询时，仅从磁盘读取与当前查询相关的列，有效节省了I/O带宽，避免不相干数据的读入，从而能够极大程度地提高分析查询的效率。

在列存储数据库的查询操作中，分组（group by）和连接（join）是最主要的，同时也是最费时的操作。

2.2.1 分组和连接

计算分组函数中时间开销最大的部分是执行group by子句，它结合合计函数（max，min，avg，sum），根据一个或多个列对结果集进行分组。

分组主要有基于排序、基于哈希技术和基于嵌套循环3种方式，Albutiu[6]从响应时间、健壮性、资源利用率等方面比较了3者的区别，得出在数据量很大并且无序的情况下，基于哈希的方法比其他两种分组方式效率高。

在Oracle 10gR2中，分组由以前版本的sort group by改成了Hash group by，这种算法上的改进，取消了sort group by必须进行的排序操作。访问时间复杂度从 $O(\mathrm{lb}\,n)$ 降低到了 $O(1)$。

连接基于两个或多个表中列之间的关系执行查询操作。比较经典的连接算法有嵌套循环连接、归并排序连接和哈希连接。

哈希连接常用于大数据集连接[7]。使用两个表中较小的表，利用连接键在内存中建立哈希表，然后扫描较大的表并探测哈希表，找出与哈希表匹配的行。

通常情况下，哈希连接的效果要比嵌套循环连接和排序合并连接好[8-9]。由于列存储数据库的数据存储特性，一般采用哈希连接算法。

2.2.2 Cache利用率

Cache的高效访问也是数据库查询的一个重要方面[10]。文献[11]对4种商业数据库的分组和连接进行了测试，结果显示，CPU等待时间占整个处理时间的50%以上。分析原因，90%是由于L1指令Cache失效和L2数据Cache失效造成的。也就是说，数据库系统的执行时间有很大一部分都花费在Cache和主存之间的数据交换上[12]。

目前在加快数据处理操作方面，主要集中于对Cache感知[13]和Cache不敏感策略[14]的研究。本文借鉴COHJ（cache-oblivious Hash joins）[15]和CONLJ（cache-oblivious nested-loop joins）[16-17]的Cache不敏感策略，对数据库join查询进行优化，提出数据存储结构Block_List，将冲突数据存入Block中，保证Block中数据连续存放，并且设置合适的Block大小，使整个Block的内容能一次性存放在Cache中，提高命中率，减少CPU的等待时间。

3 HashMap优化

本文提出的HashMap优化策略包括：分桶时的哈希函数优化；用链块地址法解决冲突；增加存储 *hashcode*，提高 *key* 的匹配效率。

3.1 哈希函数分桶

哈希分桶的第一步是选取一个均匀的哈希函数，使数据进入每个桶的概率相等，这对于数据存储的平衡性及后期匹配效率都非常重要。

本文对哈希函数的改进参考Java HashMap的哈希函数，将HashMap的容量设置为2的整数次幂，把“mod”操作改为“位与”操作。

本文优化的哈希函数代码如图1所示。

该函数对 *key* 值进行操作，*bucket_num* 代表桶的数目，是2的整数次幂，具体大小由系统最大可分配桶数目决定。将 *hashcode* mod *bucket_num* 操作改为 *hashcode* & (*bucket_num* − 1)，不但对分桶的概率没有影响，还提高了运算速度。通过位与操作，得到

```
Function HashFunc(string key){
    for (size_t i=0;i<key.length ();++i)
        hashcode&= (bucket_num−1);
  return hashcode;
}
```

Fig.1 Code of Hash function

图1 哈希函数代码

key 值所对应的 *hashcode* ,接下来将 (*key*,*value*) 指针存到相对应的桶中。

3.2 块链地址法解决冲突

传统链地址法解决冲突是将冲突的数据以指针链表的方式存储在相应的桶后。因此链表的存储是不连续的,内存的申请和释放也是分段进行的,从而遍历链表时,需要频繁地访问内存,会造成很大的时间开销。

衡量哈希表的存储效率的一个因素是装载因子(load_factor,记作 α)[18],它表示一个链的平均存储元素数,即对于一个能存放 n 个元素的、具有 m 个桶的哈希表, α 为 n/m 。

在简单均匀哈希的假设下,对于链地址法解决冲突的哈希表,一次成功和不成功的查找所需要的时间都是 $\Theta(1+\alpha)$ [18]。

降低 α 可以加快检索速度,根据定义可以增加桶的数量来降低平均装载因子。但是动态增加桶的数量又会造成重新哈希,从而增加时间开销。而在桶数量不变的情况下,随着 n 的增加,平均装载因子必定增加,造成 *key* 值冲突的概率增大。有效的办法就是优化冲突化解机制,提高冲突解决的效率。

考虑遍历链表(List)的时间开销和Cache的敏感性,本文提出了链块地址法,定义一个新的结构Block_List来代替List,该结构转变如图2所示。

Block之间通过链表链接,Block中的数据连续存储。每个Block的内容如图3所示。

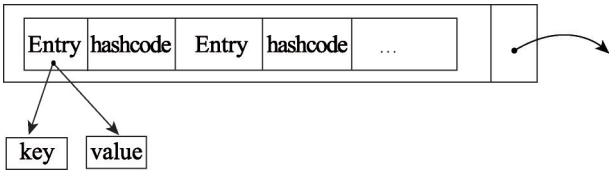

Fig.3 Contents of block

图3 Block存储内容

该结构中,每个Block存放一个结构体数组,在结构体中存放指向 (*key*,*value*) 的指针,Block之间使用链表方式链接。算法1是使用Block_List冲突化解的方法。

算法1 冲突化解

输入:原始数据 (*key*,*value*) 集合 *Sk* 和由哈希函数计

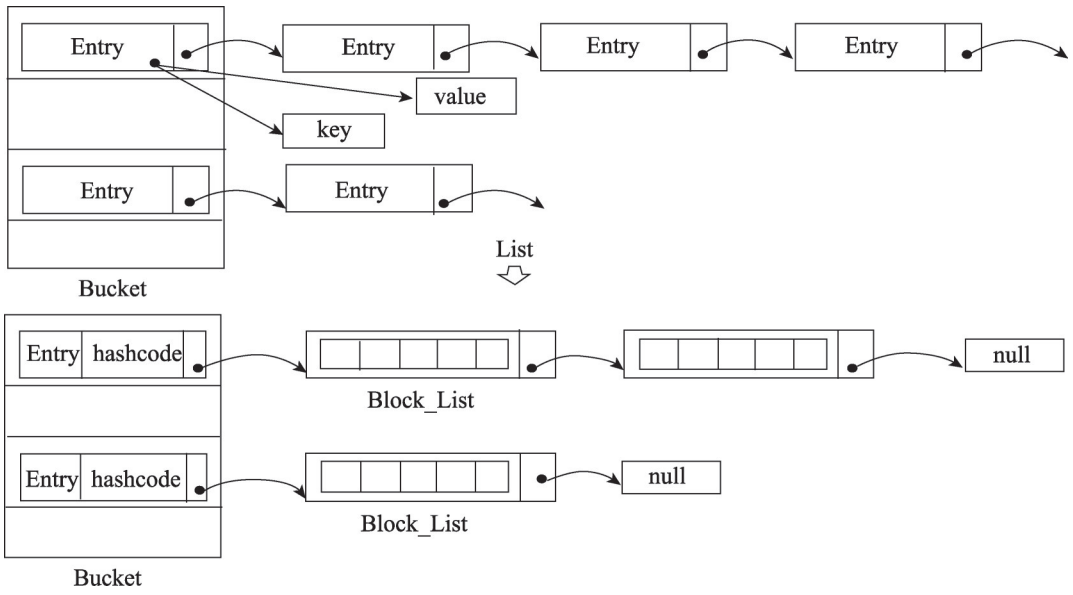

Fig.2 Block_List replacing List

图2 Block_List 代替 List

算得来的*hashcode*集合*Sh*。

输出:Block_List。

Function *conflick_resolution*(*Sk*, *Sh*)

1. For each *hashcode* ∈*Sh*
2. Read *hashcode*;
3. *m*=*hashcode* & *bucket_num*;
4. //根据*hashcode*计算出桶编号*m*, *bucket_num*是总的桶数目
5. if 所有Bucket没有存满
6. if *m*不存在;
7. 建立一个新桶,桶的编号为*m*;
8. else *m*存在,冲突产生
9. 将(*key*,*value*),*hashcode*存入对应结构体数组中
10. if 当前Block存满
11. newBlock=Malloc(sizeof(Block));
12. currentBlock->next= newBlock;
13. //创建新的Block,存在当前Block之后
14. If *Sk* ==null or *Sh* ==null;
15. All Block->next=null; //存完数据,所有 Block->next为null

对(*key*,*value*)集合以及由3.1节哈希函数计算得出的*hashcode*集合进行分桶,先求出桶编号m,如果系统分配的桶没有用完,且m不存在,则建立一个新桶;如果m已经存在,则表示有冲突,将(*key*,*value*)和*hashcode*存到相应的Block中,当Block存满之后,重新开辟Block空间。当所有数据都插入成功后,将所有Block->next置为null,表示结束。

本文将Block大小设置为一个接近α的整数,因为α是装载因子,表示一个链的平均存储元素数,读取Block的数据时,可以将一个链的冲突数据一次读入缓存,与文献[5]提出的方法相比,浪费的空间比较小,提高了堆区的使用效率。由于不用频繁地扫描链表,此方法相对传统的链地址法,在查询效率上有明显的优势,冲突数据连续存放,能高效地利用缓存。

3.3 预缓存hashcode提高匹配效率

列存储数据库的*key*值通常是不定长的字符串,当新的(*key*,*value*)值插入进来时,需要用新*key*值与Block中每一个*key*进行比较。

而字符串的存储机制是由一个字符指针指向存放字符串的地址,直接比较*key*值会造成频繁的读取内存。本文将每个*key*的*hashcode*也预读到Block中,在匹配时,先直接用*hashcode*比对,当Block中具有相同的*hashcode*时,才去*key*值所对应的内存中对比*key*字符串以及*value*值。

算法2是预缓存*hashcode*值匹配算法的具体实现。

算法2 预缓存*hashcode*值匹配算法

输入:(*key*,*value*), *hashcode*。

输出:*key_exist_tag*布尔值,该*key*值是否存在。

Function *key_exist*()

1. *Bucket_num*= *hashcode*&*m*;
2. if *Bucket_num*不存在
3. Return false;
4. else
5. for each Block
6. 比较当前*hashcode*与Block中存储的*hashcode*值
7. if *hashcode*存在
8. 比较当前*key*和*hashcode*对应的*key*
9. if *key*值相等
10. return true
11. else return false;
12. else *hashcode*不存在
13. if Block->next != null;
14. Block =Block->next
15. else Block->next ==null;
16. return false;

对于输入的*hashcode*值以及(*key*, *value*),要查看当前HashMap中是否存在该值。首先根据*hashcode*求得*Bucket_num*,如果*Bucket_num*不存在,则返回false,如果存在,则比较*hashcode*和该Bucket的第一个Block中存储的*hashcode*,如果不同,返回false,如果相同,对比两个*key*值。如果遍历完整个Block_List都没有匹配到,则返回false。

采用*hashcode*比对代替*key*值的比较,会增加少量的存储空间,但由于将字符串的比较改为数值的比较,对内存的访问次数相应减少了,比较速度也提高了。同时,每次从内存中读取一个Block大小的数据进行Cache,根据CPU访问数据的局部性,使缓存得到高效利用,总体上匹配效率还是提高了。

3.4 BHMap

结合以上优化策略,本文提出存储结构BHMap,该存储结构如3.2节中图2和图3所示。

首先,在选取哈希函数分桶的过程中,设置桶的大小为2的整数次幂,保证哈希分桶的均匀性;其次,使用位与操作能够快速定位桶编号,当新(*key*,*value*)插进来,如果发生冲突,将冲突数据存放在该桶之后的Block_List中,直到Block存满后,重新申请内存开辟下一块Block。设计Block大小的时候,考虑Cache敏感性,将Block大小设计为接近 α 的整数,每次从内存中读取一个Block的数据进Cache,使冲突数据能够连续存放,保证计算机的空间和时间局部性。

同时,在数据插入过程中,BHMap增加存储*hashcode*值,在Block_List中存储包含指向内存(*key*, *value*)的指针entry,以及该*key*值计算得到的*hashcode*,在匹配查找的过程中,先匹配数值型*hashcode*,如果相等再匹配*key*值,可用减少指针寻址时间以及长字符串的匹配时间。

4 实验设计与结果分析

根据提出的优化方法,实验设计部分将比较使用BHMap优化前后的查询性能。测试准则包括运行时间和运行时占用的内存大小。选取的基准测试案例如图4所示,test_map代表数据结构,运行时用BHMap、Map以及unordered_map等替代。

```
void ins_or_upd()
 {
   size_t n = str_vec.size();
   for (size_t i = 0; i < n; ++i)
      ++test_map[str_vec[i]]
 }
```

Fig.4 Benchmark test case

图4 基准测试案例

该案例实现一个单一的计数功能。采用该基准测试,可以避免其他因素对实验结果的影响。

本文将桶数目和*key*值的不重复数据量(KeyNoRepeatNum, β)作为可变参数来衡量结构体的性能及适用场景。当数据量确定时,桶数目会直接影响 α。参数 β 能够说明源数据的不确定性, β 越高,说明源数据重复率越低,对其进行碰撞的估计就越难,因此本文将 β 作为一个主要参数,进行以下3组实验。

实验1 比较3种数据结构在不同 α、β 下的性能:①Map,该结构体是C++标准的Map结构,实现原理是平衡二叉树;②unordered_map, 该结构在C++ Boost库中实现,用链表法解决冲突;③本文提出的BHMap。

实验2 优化效果测试。对于BHMap,比较3种情况:①只优化哈希函数(HashFunc);②优化哈希函数以及使用块链结构(Block_List,BList);③优化哈希函数,使用Block_List并且缓存了*hashcode*的BHMap。

实验3 适用场景验证。在不同的桶数目下,比较unordered_map、HashFunc、BList、BHMap在不同的 α、β 下的性能。

对于实验1、实验2、实验3,统计比较各种情况下的运行时间和占用内存情况。

实验使用CentOS release 6.5操作系统,Cache size为15 360 KB,每个数据文件包含10 000 000条记录,存储形式为字符串。每个文件的 β 近似呈指数增长,分别为99, 999, 9 999,99 997, 999 960, 9 950 474。

4.1 比较3种数据结构性能

对于3种数据结构Map、unordered_map和BHMap,首先比较在不同的桶数目下的性能。在桶数目为1 048 576和16 777 216时(即 α 为9.537和0.596,Block大小为10),测得其时间性能分别如图5、图6所示。

从图5中可以看出,在桶数目一定时,随着 β 的增加,查询消耗的时间也逐渐增加。在 β 为9 950 474,即接近于文件总记录数时,BHMap有明显的性能优势,其查询速度比unordered_map快3.5倍以上,比Map快10倍以上。从实验数据可以看出,BHMap适用于 β 比较大,即*key*值的重复率比较低的情况。

图6是桶数目比较多的情况,此时BHMap和unordered_map的性能差别不大。说明桶数目的大小对基于HashMap的结构性能影响比较大,桶越多,查询速度越快。

对比图5、图6中可以看出,桶数目直接影响unordered_map的性能。在桶数目比较大,即 α 比较

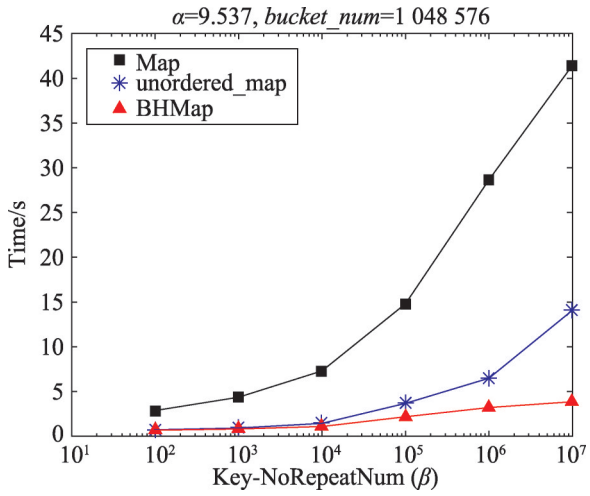

Fig.5 Time consumption of 3 data structures

图5 3种数据结构时间消耗比较

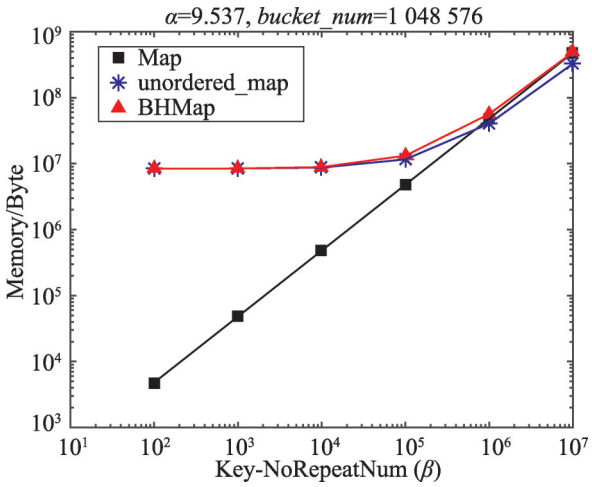

Fig.7 Memory consumption of 3 data structures

图7 3种数据结构内存消耗比较

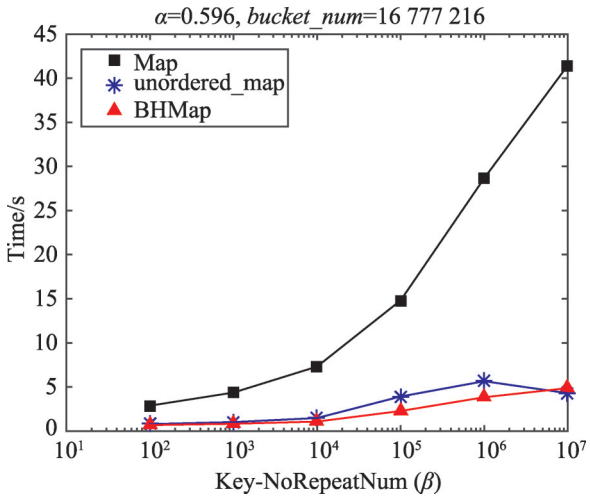

Fig.6 Time consumption of 3 data structures

图6 3种数据结构时间消耗比较

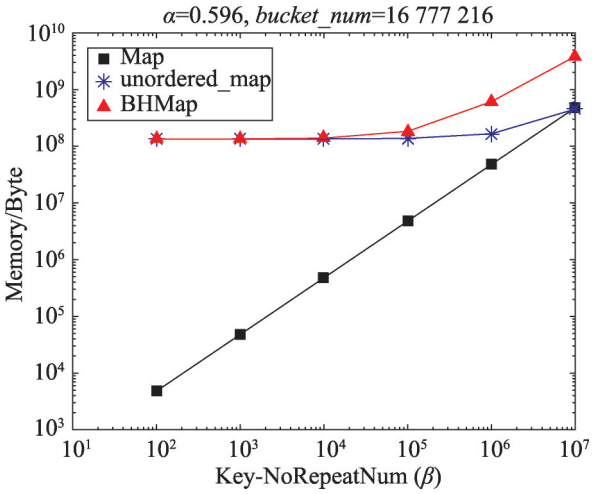

Fig.8 Memory consumption of 3 data structures

图8 3种数据结构内存消耗比较

小时，unordered_map也能发挥很好的性能，其运行时间与BHMap接近，同时证明了减小α是提高HashMap性能的一个有效方法。实验也可以证明，执行与顺序无关的操作时，HashMap的性能比Map好。

图7、图8对比了3个数据结构在不同桶数目下运行的内存使用情况。

从图7、图8中可以看出，在α比较高，β比较低的情况下，Map运行时占用的内存比unordered_map和BHMap少，在桶数目为1 048 576时，unordered_map和BHMap对内存使用的变化趋势相同。但是，在桶数目增加到16 777 216时，BHMap对内存的使用量明显增加，比unordered_map高出近一个数量级。

内存方面，BHMap在可用桶数目比较多的情况下，会比unordered_map多消耗内存。这一点也说明了Block_HashMap更适用于桶数目比较少的情况。

4.2 优化效果测试

对本文提出的3个优化方法HashFunc、BList、BHMap分别测试其运行时间和内存使用量。根据4.1节的结论，本实验的桶数目设置为1 048 576，即桶的数目比较少的情况。查询时间和内存使用情况如图9、图10所示。

图9说明，HashFunc优化哈希函数，使用“位与”

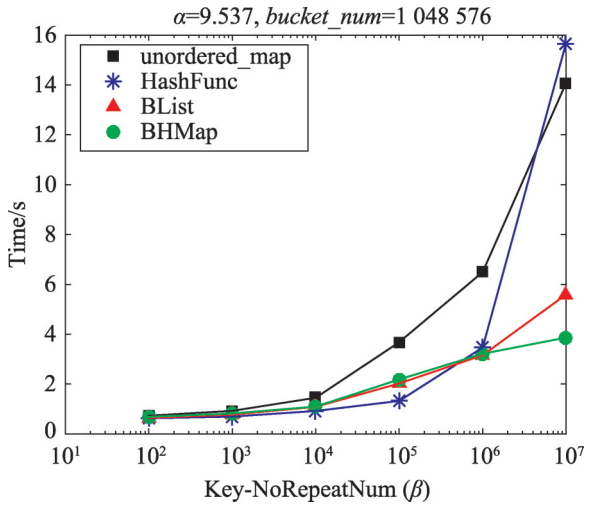

Fig.9 Time consumption comparison of optimization effect

图9 优化时间性能比较

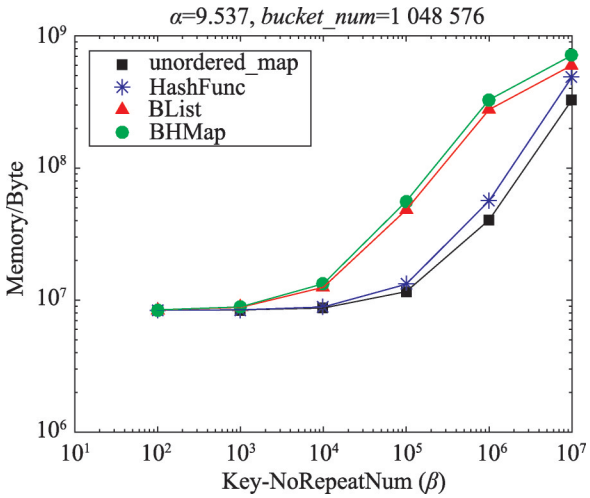

Fig.10 Memory consumption comparison of optimization effect

图10 优化内存使用比较

操作代替“取模”操作，在 β 比较低的情况下，能节省程序运行的时间。但是HashFunc仍然使用链表解决冲突，在 β 比较高的情况下，其运行时间超过了unordered_map，因为该版本的优化在桶数目不够时，需要动态开辟新的桶空间，这个过程消耗的时间比较多。BList优化版本实现了Block_List，用块链式结构来解决冲突，从图中可以看出，其性能明显提高。在BHMap版本中，预缓存了*hashcode*，使访问速度明显较BList版本又有所提高。

从图10中可以看出，使用Block_List结构在 β 比较低的情况下，会比unordered_map多消耗内存，但是，在 β 接近记录的总数时，unordered_map对内存的使用量也急剧上升，两者差别减小，这也说明了BHMap适用于*key*值重复率比较低的情况。

虽然BHMap较BList版本多存储了*hashcode*值，会多占用少量内存，但是由于*hashcode*值是整型数据，不会占用太多内存。从内存成本和时间效益来看实验结果也是可以接受的。

4.3 适用场景验证

基于4.2节的实验，本节讨论BHMap的适用场景。在不同的桶数目下，随着 β 的增大，unordered_map、HashFunc、BList、BHMap的运行时间和内存使用情况如图11、图12所示。

从图11中可以看出，对于unordered_map和BHMap，一般都是桶数目越多，碰撞越小，运行时间越少，效果越好。但是现实数据并不总是这样，在桶数目有限、数据量很大的情况下，要减少 α 来加速HashMap就很困难。前3个子图充分说明了这种情况，尤其是前两个子图，在 β 接近记录总数时，桶数目不同而引起的时间消耗差距已达到5倍左右。

本文介绍的优化方法，充分考虑到这种情况，在使用Block_List之后，运行时间对桶数目的敏感性降低。BHMap在数据的重复率比较低的情况下，优化后的数据结构在桶数目为1 048 576时，已经超过了桶数目为10 777 216时的性能。

图12说明，桶的数目越多，运行时占用的内存越多。相比之下，使用了Block_List的BHMap要比使用链表解决冲突unordered_map多占用内存。在桶数目为16 777 216的情况下，当数据的重复率比较低，β 接近记录总数时，内存使用量突然大幅度增加。从后两个子图可以看出，对于桶数目比较少的情况，优化的BHMap能够控制内存的急剧增加。

从以上3个实验，总结出了BHMap的适用场景，是对一般优化方法通过降低 α 来加快HashMap访问速度的补充。为了节约时间和空间，桶的数目一般固定，并且不会设置得太大。在动态增加桶的数量

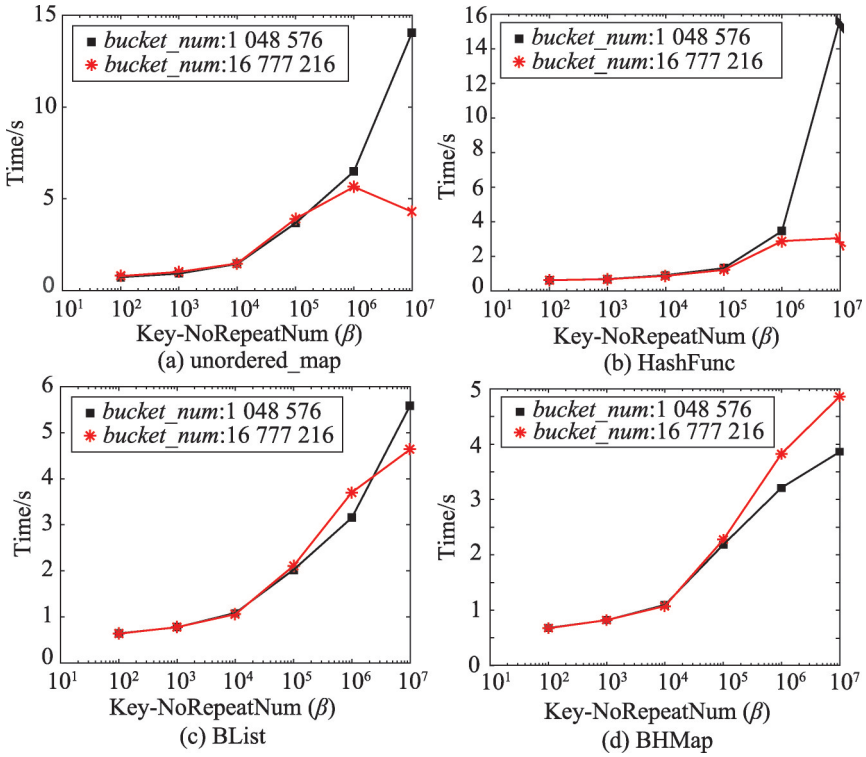

Fig.11 Time consumption comparison of optimization effect on different *bucket_num*

图11 不同桶数目下优化版本的时间消耗比较

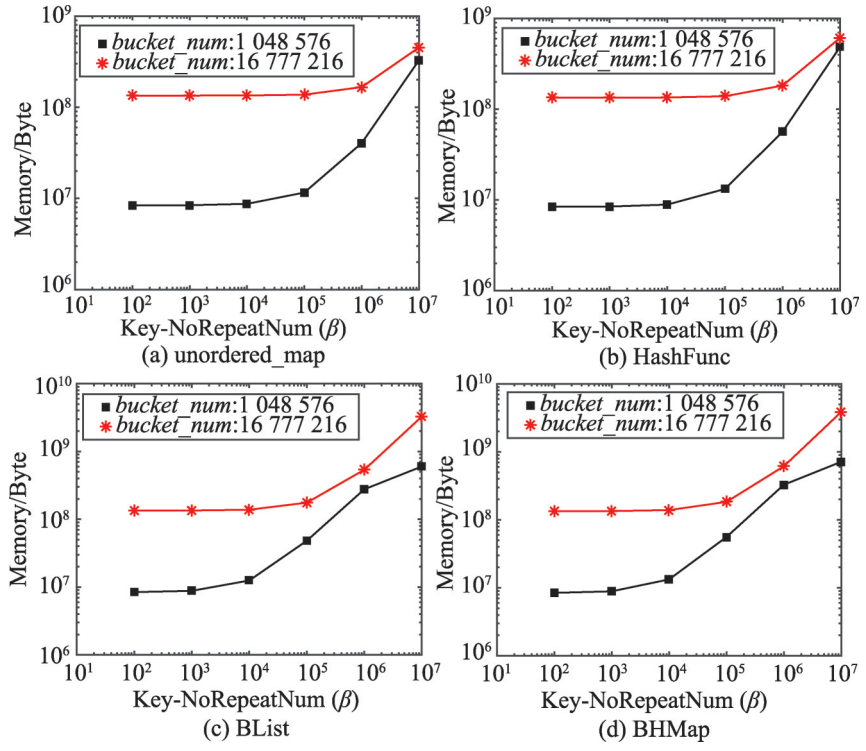

Fig.12 Memory consumption comparison of optimization effect on different *bucket_num*

图12 不同桶数目下优化版本的内存消耗比较

时，会造成重新哈希，从而增加时间开销。而在桶数量不变的情况下，随着 n 的增加，平均装载因子就会上升，从而造成 *key* 值冲突的概率增大。本文优化也能表现出很好的效率。

在处理的数据重复率比较高的情况下，unordered_map和BHMap的性能都比较好，但是对于数据重复率比较低的情况，BHMap的优势非常明显。

5 BHMap在列存储数据库中的应用

优化后的BHMap调用接口类似于unordered_map，可以在任何适用的场合方便地调用。本文以列存储数据库查询语句中两个重要的查询——分组和连接，来举例说明BHMap的使用。

5.1 分组查询实现

分组部分，主要实现group by语句，该语句结合合计函数，根据一个或者多个列进行分组。结合BHMap，本文定义适合的哈希分桶函数以及匹配函数，将要分组的列放在一个结构体key_set中，作为HashMap的 *key* 值，对其进行分桶和匹配。

group by算法实现原型：

```
SELECT column_name, Aggregate_function(column_name)
FROM table_name
WHERE column_name operator value
GROUP BY key_column
```

算法3 group by

输入：*column_file*；//列存储文件

输出：*Res_Set*; //BHMap结果集

Function *group_by*()

1. Read column_names to *Arrays*;
2. BHMap<key_set,value_set,hasher,keyEquer>*Res_Set*
3. *tmpresult*=Array.get(*index, value*);
4. *Res_Set*.inset(*tmpresult)*;
5. If insert succed
6. continue;
7. else
   insert failed

SELECT关键字之后存储列的结构体，从源文件的第index行，读出 *value* 值放在临时结果变量 *tmpresult* 中。再根据 *tmpresult.key_set* 作为键值分桶，如果桶已经存在，则计算合计函数。

5.2 连接查询实现

连接部分，用于根据两个或者多个表中列之间的关系，从这些表中查询数据。

hash join将两个表中较小的一个在内存中构造一个哈希表，扫描另一个表，与内存中的小表进行比较，找出与之匹配的行。

join算法实现原型：

```
SELECT result_column
FROM build_table,probe_table
WHERE build_table.keyset=probe_table.keyset
```

算法4 join

输入：*build_files*(小表)，*probe_files*(大表)

输出：*Res_Set* //BHMap结果集 Function *hash_join*()

1. BHMap<key_set,index,hasher,keyequel>*Res_Set*
2. read build_file column to *B_Array*, probe_file column to *P_Array*;
3. *tmpresult=B_Array*.get(*index*,*value*)
4. *Res_Set*.insert (*tmpresult*);
5. If insert succed
6. *P_Array*.find(*tmpresult.key_set*)
7. //从小表中取出 *tmpresult*，扫描大表，匹配
8. if find succed
9. *Res_Set*=index of *B_Array* and *P_Array*;

return *Res_ Set*

读入存放小表的文件build_files，将其存入数组 *B_Array* 中，与group by算法类似，将其以主键分桶；之后，对大表数据对应的 *P_Array* 中的每一个数据，从BHMap匹配小表中的数据，如果找到，记录其在 *B_Array* 和 *P_Array* 的下标，找出结果。

6 结束语

本文提出的BHMap结构适用于两种情况：桶数目一定，而 *key* 值重复率比较低；或者数据一定，但是可用的桶数目比较少。

BHMap的优化包括哈希分桶、冲突解决以及 *key* 值匹配的整个过程。在哈希分桶过程中，选取合适的哈希函数，用“位与”操作代替传统的“取模”操作

来得到桶编号,提高了运算效率。在冲突解决阶段,定义了存储结构Block_List来代替传统的链表,提高存储效率以及插入效率。在 *key* 值匹配过程中,预缓存 *hashcode* 值,匹配过程中用整型数据代替字符串,节省了对指针内存的遍历时间。

本文考虑到存储体系中Cache的命中率对CPU查询效率的影响,考虑缓存敏感性,相对于传统的链地址法,每个节点可以连续存储多个值,存储密度增加,CPU检索数据不需要频繁地进行指针寻址,查询速度明显提高。同时设计Block大小时,使其接近装载因子,使冲突数据能一次读入缓存,也可以减少空间的浪费。

在存储方面,由于在每个节点中多存储了 *hashcode* 值,相比较传统的unordered_map,会增加一定的存储空间。但是由于多存储的 *hashcode* 值是整型数据,运行时不会占用太多内存,在桶数目比较少的情况下,该消耗是可以接受的。

对于数据库中一些主要的查询操作,如分组、连接等比较耗时的语句,本文提出的优化方法可以作为unordered_map的补充,根据查询数据的特点,选取结构进行查询,可在某些应用场景下提高查询性能。

References:

[1] Boncz P A, Zukowski M, Nes N. MonetDB/X100: hyper-pipelining query execution[C]//Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research, Asilomar, CA, Jan 4-7, 2005: 225-237.

[2] Abadi D J. Query execution in column-oriented database systems[D]. Massachusetts Institute of Technology, 2008.

[3] Owolabi O. Empirical studies of some hashing functions[J]. Information & Software Technology, 2003, 45(2): 109-112.

[4] Ma Rulin, Jang Hua, Zhang Qingxia. An improved fast searching method of Hash table[J]. Computer Engineering & Science, 2008, 30(9): 66-68.

[5] Li Xiaotang, Zhan Feng. An improved hash map realization method[J]. Journal of Shenzhen Institute of Information Technology, 2010, 8(2): 80-83.

[6] Albutiu M C. Scalable analytical query processing[D]. München: Technische Universität München, 2013.

[7] Singhal R, Nambiar M. Extrapolation of SQL query elapsed response time at application development stage[C]//Proceedings of the 2012 Annual India Conference, Kochi, India, Dec 7-9, 2012. Piscataway, USA: IEEE, 2012: 35-41.

[8] Balkesen C, Alonso G, Teubner J, et al. Multi-core, main-memory joins: sort vs. hash revisited[J]. Proceedings of the VLDB Endowment, 2013, 7(1): 85-96.

[9] Graefe G. New algorithms for join and grouping operations[J]. Computer Science-Research and Development, 2012, 27(1): 3-27.

[10] Qin Xiongpai, Wang Huiju, Li Furong, et al. New landscape of data management technologies[J]. Journal of Software, 2013, 24(2): 175-197.

[11] Ailamaki A, Dewitt D J, Hill M D, et al. DBMSs on a modern processor: where does time go?[C]//Proceedings of the 25th International Conference on Very Large Data Bases, Edinburgh, UK, Sep 7-10, 1999. San Francisco, USA: Morgan Kaufmann Publishers Inc, 1999: 266-277.

[12] Han Xixian, Yang Donghua, Li Jianzhong. DBCC-join: a novel cache-conscious disk-based join algorithm[J]. Chinese Journal of Computers, 2010, 33(8): 1500-1511.

[13] Shatdal A, Kant C, Naughton J F. Cache conscious algorithms for relational query processing[D]. University of Wisconsin-Madison, Computer Sciences Department, 1994.

[14] He Bingsheng. Cache-oblivious query processing[D]. Hong Kong University of Science and Technology, 2008.

[15] He Bingsheng, Luo Qiong. Cache-oblivious hash joins[R]. Hong Kong University of Science and Technology, 2006.

[16] He Bingsheng, Luo Qiong. Cache-oblivious nested-loop joins[C]//Proceedings of the 2006 ACM CIKM International Conference on Information and Knowledge Management, Arlington, USA, Nov 5-11, 2006. New York: ACM, 2006: 718-727.

[17] Manegold S, Boncz P A, Kersten M L. Optimizing main-memory join on modern hardware[J]. IEEE Transactions on Knowledge and Data Engineering, 2002, 14(4): 709-730.

[18] Cormen T H, Leiserson C E, Rivest R L, et al. Introduction to algorithms[M]. Cambridge, USA: MIT Press, 2001.

附中文参考文献:

[4] 马如林, 蒋华, 张庆霞. 一种哈希表快速查找的改进方法[J]. 计算机工程与科学, 2008, 30(9): 66-68.

[5] 李晓堂, 詹峰. 一种改进的 hash map 实现方式[J]. 深圳信

息职业技术学院学报, 2010, 8(2): 80-83.
[10] 覃雄派, 王会举, 李芙蓉, 等. 数据管理技术的新格局[J]. 软件学报, 2013, 24(2): 175-197.
[12] 韩希先, 杨东华, 李建中. DBCC-join: 一种新的高速缓存敏感的磁盘连接算法[J]. 计算机学报, 2010, 33(8): 1500-1511.

MU Hongfen was born in 1990. She is an M.S. candidate at Beijing University of Chemical Technology, and the student member of CCF. Her research interests include database and query algorithm optimization.
母红芬(1990—),女,云南大理人,北京化工大学硕士研究生,CCF学生会员,主要研究领域为数据库,查询算法优化。

LI Zheng was born in 1974. He received the Ph.D. degree from King's College London in 2009. Now he is a full professor at Department of Computer Science, Beijing University of Chemical Technology, and the senior member of CCF. His research interest is software analysis and testing. He has published more than 30 papers, and has taken charge of 3 projects funded by National Natural Science Foundation of China.
李征(1974—),男,河北清苑人,2009年于英国伦敦国王学院获得博士学位,现为北京化工大学计算机系教授,CCF高级会员,主要研究领域为软件分析及测试。发表学术论文30多篇,先后主持3项国家自然科学基金项目。

HUO Weiping was born in 1970. He is the vice president and chief engineer of Business-Intelligence of Oriental Nations Corporation. His research interests include data analysis, data warehouse and business intelligence of telecommunications and financial industry, etc.
霍卫平(1970—),男,山西朔州人,北京东方国信科技股份有限公司副总经理、总工程师,主要研究领域为电信、金融行业的大数据分析、数据仓库、商业智能等。

JIN Zhenghao was born in 1971. He received the M.S. degree from University of Chinese Academy of Sciences. Now he is the vice president of R&D center in Business-Intelligence of Oriental Nations Corporation. His research interests include system analysis and development in big data of telecommunications industry.
金正皓(1971—),男,吉林长春人,中国科学院数学和系统科学研究院硕士,现为北京东方国信科技研发中心副总经理,主要研究领域为电信行业的大数据分析系统的规划和开发工作。