缓存关联对数据库系统性能的影响及其优化策略*)

冯 柯 陈 刚 董金祥
(浙江大学计算机科学与工程学系 杭州310027)

摘 要 研究了缓存关联对数据库系统性能的影响,并提出了一种优化策略,大大降低了数据访问时对缓存的争用。该优化技术实现简单,对已有系统改动小,已经实现并应用于开放源代码数据库系统 Postgres 和 MySQL,以及自主开发的 CoreBase 数据库系统中,测试结果表明:数据库的查询速度得到了极大的提高。

关键词 缓存关联,缓存争用,热区

Cache Associativity on the Performance of Database Systems: Problems, and Optimization Strategies

FENG Ke CHEN Gang DONG Jin-Xiang
(Department of Computer Science and Engineering, Zhejiang University, Hangzhou310027)

Abstract In this paper, we studied the impact of cache associativity on the performance of database systems, and presented a new optimization technique, which can reduce the cache conflict miss during the data access significantly. This technique is very simple, and requires only a trivial modification on current database systems. We have already implemented this technique in Postgres, MySQL, two famous open-source database systems, and also CoreBase, a database system developed by ourselves. Experiment results have shown that, the query performance of all above systems were improved greatly.

Keywords Cache associativity, Conflict miss, Hotspot

计算机的内存容量正在并将继续快速增加,根据著名的 Asilomar[1]报告的预测:在今后十年,典型的数据库服务器所配备的内存将达到1Tb,除最大的数据表之外,其它所有数据都能够被存放于内存之中;内存容量的增加减少了数据访问中引起的磁盘 I/O 次数,从而大大提高了数据库系统的性能。另一方面,CPU 和内存访问的速度差距正在迅速拉大,在上个十年中,CPU 速度每年提升60%～70%,而内存访问速度总共大约只提高了一半[2];在现代硬件平台上,完成一次内存访问通常需要数十、甚至上百个时钟周期,内存访问已经成为数据库系统新的性能瓶颈[2～4]。

解决这一问题的途径在于使用缓存(cache)。缓存位于内存和 CPU 之间,当 CPU 需要存取某数据时,它先访问缓存,如果数据位于缓存中(称为缓存命中,cache hit),该数据被直接读取到相应的寄存器里;否则(称为缓存未命中,cache miss),该数据被首先从内存读取(映射)到缓存中。访问缓存(指 L2缓存,下同)一般要比访问内存快5～10倍[4]。通过将最常用的数据存放在缓存中,可以有效地减小因内存变慢对系统性能造成的影响。但缓存容量有限(通常比内存容量小2～3个数量级),因此如何充分地利用缓存就成为提高缓存性能的关键。

提高系统的缓存性能是当前数据库研究的热点,学术界对此已经进行了大量的工作。如文[3]对几个商用数据库系统进行了性能测试,结果表明这些系统在运行时将因为缓存未命中而浪费近50%的 CPU 时间,这一结果有力地揭示了内存变慢对数据库系统性能的影响;文[5]通过将页面内的数据进行属性分解,并按照属性分组存放,来减少区域查询(range selection)时需要访问的数据量,大大提高了数据库系统在运行 OLAP 应用时的性能;文[6,7]对几种常用的数据库索引的缓存性能进行了分析,并针对树状索引(B^+Tree)提出了改进结构,改进后的索引中的每个节点能够存放更多的数据,从而降低了索引深度,减少了搜索时的内存访问次数;文[2,4]分析并比较了数据库连接操作常用的几种实现的缓存性能,并提出了一些改进算法,类似的工作还包括文[8]。

本文研究缓存关联(cache associativity)对数据库系统性能的影响。缓存关联是缓存的重要特性之一,它使得任意一个内存对象只能被映射到缓存中的特定区域。本文的分析表明:由于缓存关联的限制,目前数据库系统中所广泛采用的分页数据存放结构将导致在访问过程中产生严重的缓存未命中,从而极大地影响了系统性能。本文针对这一问题进行了深入的分析,并在此基础上给出了优化方案。

1 基本定义

缓存按照一定长度被分成多个缓存块(cache line),所有缓存块通常又被分为若干个大小相等的缓存组。

定义1(缓存) 缓存是(U,K,L)的三元组。

上述定义表示该缓存包含 U 个缓存组,每个缓存组包含 K 个缓存块(K 也被称为关联度),每个缓存块的长度为 L。相应地,内存也按照同样长度被分成多个内存块,当 CPU 需要访问内存中的数据时,该数据所在的内存块被整块读入(映射)到特定的缓存块中。缓存块的选择取决于缓存的映射函数(也称为缓存关联)。

定义2(缓存关联) 缓存关联是$C \rightarrow (U_C, K_C)$的函数,记

*)本文研究得到863项目(2003AA4Z3010)资助。冯 柯 博士生,主要研究领域为内存数据库和实时数据库。陈 刚 副教授,主要研究领域为对象数据库和安全数据库。董金祥 教授,博士生导师,主要研究领域为 CAD 和工程数据库。

作 $\pi(C)$。

映射函数 $\pi(C)$将内存块 C 映射到缓存组 U_C 中的第 K_C 个缓存块。映射一般分两步进行：首先将内存块映射到缓存组，称为组间映射；再在该缓存组中选择最终要映射的缓存块，称为组内替换，如果该缓存块已经映射了某内存块，则该内存块被替换，绝大多数系统都采用 LRU 策略来实现组内替换。为简化讨论，本文以下所指的映射函数将只反映组间映射。

常用的(组间)映射函数是通过将内存块地址对缓存组总数取模来获得要映射的缓存组，即：

$$\pi(C) = C \% U$$

定义3(程序) 程序是一组内存访问的有限序列，记作 $P = \{C_1, C_2, \cdots, C_N\}$。

其中 C_i 表示程序 P 的第 i 次内存访问的内存块地址。缓存优化的目标，就是使得程序 P 在缓存(U, K, L)上运行时产生的缓存未命中数最低[1]。为了达到这一目标，充分利用有限的缓存是非常重要的，缓存利用率表示程序在执行过程中所有映射的缓存块占缓存块总数的比例，缓存利用率可用于对程序中的随机访问进行缓存性能分析[2]。为便于讨论，本文引入缓存组利用率：

定义4(缓存组利用率) 对于程序 $P = \{C_1, C_2, \cdots, C_N\}$，其缓存组利用率 $\mu(P) = \|\{\pi(C_1), \pi(C_2), \cdots, \pi(C_N)\}\| / U$。

其中 $\|S\|$表示集合 S 中包含的不同元素的个数。缓存组利用率反映了程序 P 对缓存组的利用情况，它是缓存利用率的一种近似(当 K 较小时)。本文使用 $\mu(P)$来衡量应用随机访问的缓存性能。

2 问题的提出

2.1 缓存争用

缓存关联使得任一内存块只能被映射到一个特定的缓存组中，这一限制在实际使用时将引发缓存争用(conflict miss)，下面的例子将说明缓存争用是如何产生的：

考虑程序 $P = \{C_0, C_1, C_0\}$，假设 $K = 1$，且 $\pi(C_0) = \pi(C_1) = U_0$，$P$ 首先访问内存块 C_0，C_0被映射到缓存组 U_0中，再访问内存块 C_1，C_1也被映射到 U_0中，由于 $K = 1$，C_0将被替换，此时继续访问 C_0将发生缓存未命中，从而导致缓存争用。需要指出的是：这种缓存争用并不是因为缓存容量不够而引起的，而是由于缓存关联的限制使得程序对内存的访问无法充分利用缓存，因此增加缓存容量无助于解决这一问题。

如果能将 C_0和 C_1映射到不同的缓存块中，则上述争用就不会发生，因此提高缓存的关联度显然可以减少缓存争用发生的概率。但是关联度的增加将大大降低组内替换(LRU)算法的性能，同时使硬件成本变得昂贵，因此绝大多数实际系统中的缓存都采用了较小的关联度(通常位于1～8之间)。在此前提下，如果应用需要随机访问多个(远远大于关联度)将被映射到同一缓存组的内存块时，提高关联度并不能消除大多数的缓存争用。本文以下的讨论将证明这一点。

2.2 数据库系统中的热区

分页结构是绝大多数数据库系统所采用的数据存放结构，在这一结构中，数据以页面为单位从磁盘读取到内存中，页面长度通常是磁盘扇区长度的整数倍。通过将那些需要经常访问的页面存放在内存缓冲区中，能够有效地减少磁盘 I/O 次数，从而大大提高了系统性能。

数据库中的页面可以被分为两种：用于存放表记录的数据页面，和用于存放索引的索引页面。无论是在哪种页面中，都存在有一些需要经常被访问的区域，称为热区(hotspot)。

对于数据页面，大多数数据库系统都采用了一种被称为 NSM(N-ary Storage Model)的页面存储模式[5,8,9]。这种存储模式将记录从每页的开始位置连续存放，每页的尾部则存放一个用于维护该页面中各记录在页面内起始位置信息的记录偏移表(如图1(a))。为完成记录的定位，数据库首先根据记录的地址计算出记录所在的页面，以及其在记录偏移表中的偏移量，再从记录偏移表的对应入口中获得该记录在页面内的起始位置。NSM 的优点在于各记录在页面内可以自由移动而不改变其地址，因此能够高效地实现页面内的在线空间重整，在不降低系统性能的同时大大提高了数据库的空间利用率。很明显，由于每次访问记录之前都必须访问其所在页面的记录偏移表，记录偏移表所在的内存块就成为数据页面中的热区，称为记录热区。

对于索引页面，B^+-Tree 是大多数数据库系统所采用的主要索引形式，B^+-Tree 在每个索引页面内连续存放多个〈键值，该键值对应的记录地址〉索引对，并利用二分法查找进行页面内定位。二分法查找可以有两种实现方式：一是只在有效的索引对内进行查找(如图1(b))；此时必须在每个页面内保存该页面当前所有有效的索引对的计数，该计数所在的内存块就是索引页面的热区，称为索引热区。二是不保存计数，而是对所有的索引对都进行查找，索引对的数目可以通过页面长度除以索引对长度来求得，无效的索引对的键值则被设置为最大值(如图1(c))；这种处理虽然可能增加比较次数，但由于索引对的数目是固定的，因此可以利用循环展开(loop unrolling)[10]技术对查找过程进行优化；分析二分法查找过程可以发现，该算法总是从要查找的数据集的中间开始查找，因此此时索引页面中间位置所在的内存块就成为新的索引热区。

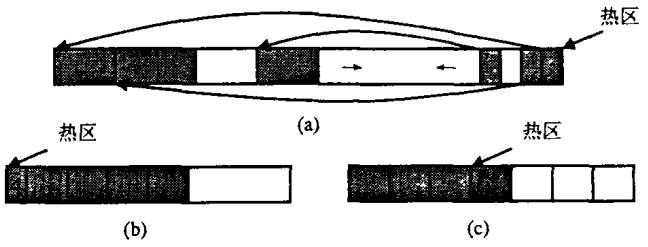

图1 页面结构示意图

2.3 热区对性能的影响

根据以上讨论，数据库系统中的热区具有如下特点：1)对热区的访问频率要远远高于页面中的其它部分；2)同一种热区(记录热区或索引热区)在其所在页面中的相对位置是相同的。这些特点使得对热区的访问将产生严重的缓存争用。

设 H 是数据库中某一种热区的所有集合，可以把程序 P 中所有对 H 的访问构成该程序的一个热区序列，该序列是 P

1 缓存在使用中还存在其它的开销，如 TLB miss 等，但这些开销对实际系统的性能影响不大[3]。

2 对于随机访问，如果需要访问的内存块数大于可映射的缓存块数，缓存未命中将是严重的，此时缓存利用率直接反映了系统的缓存性能，缓存利用率越低，随机访问的性能也就越差；而对于顺序访问，由于已访问过的内存块将不再被映射，因此其性能与缓存利用率无关。

的子序列，记作 P_H。由于对热区的访问远比其它数据频繁，因此热区序列的缓存性能将直接决定整个程序的性能。设每个数据库页面可包含 N 个内存块，则 P_H 中的任一访问 C_H 可以记作：

$$C_H = B_H * N + \alpha, 0 \leqslant \alpha < N$$

其中 B_H 表示 C_H 的(内存)页面编号，α 为 C_H 在该页面内的相对位置，由于同一种热区在其所在页面中的相对位置是相同的，因此 α 是一个常量。设 T 是 N 和 U 的最大公约数，则有：

结论1 $\mu(P_H) \leqslant 1 / T$

证明过程如下：设 $N = N_0 * T, U = U_0 * T, \alpha = \alpha_0 * T + \beta (0 \leqslant \beta < T)$，则：

$$C_H = B_H * N + \alpha = (B_H * N_0 + \alpha_0) * T + \beta$$

$$\pi(C_H) = C_H \% U = ((B_H * N_0 + \alpha_0) * T + \beta) \% (U_0 * T) = (B_H * N_0 + \alpha_0) \% U_0 * T + \beta$$

记 $M = (B_H * N_0 + \alpha_0) \% U_0$，则 P_H 中的任一访问 C_H 将只能被映射到形如 $M * T + \beta$ 的缓存组中(β 为常量)，而满足这一条件的所有缓存组的容量总和不会超过缓存总量的 $1 / T$，因此结论1成立。

在绝大多数系统中，U 总是能被 N 整除($U >> N$，且 U、N 均为2的整数次幂)。这意味着，当对上述热区进行访问时，访问过程最多只能利用 $1 / N$ 的缓存($T = N$)，对于典型的系统配置(页面长度4K～8K 字节，缓存块长度32～128字节)，其平均缓存利用率不到1%。这意味着：对数据库系统中热区的随机访问[3]将导致严重的缓存争用，从而大大地影响了系统性能。另外，随着磁盘带宽的不断提升，数据库系统中的页面长度将会不断增加[11]，这使得以上问题将变得更加严重。

3 优化策略

3.1 基本思想

解决上述问题的关键在于提高系统对热区访问时的缓存利用率。一种直观的优化方案是使得各页面中位于同一相对位置的内存块能够被映射到不同的缓存组中，这种方法的优点在于无须改变页面的结构，因此对应用是透明的。

修改映射函数可以实现上述目标，但为了保证顺序访问的性能，根据内存块的高位地址来定位要映射的缓存组是不可行的，因此修改后的映射函数必然是非线性的，这增加了映射开销，也使得硬件设计成本大大提高。也可以考虑修改 U 或者 N，使得 U 与 N 尽量互质($T = 1$)，但修改前者意味着映射开销的增加[4]；而修改后者则意味着页面长度的改变(改变已有缓存的缓存块长度是不可想象的)，这将使页面长度不再是磁盘扇区长度的整数倍(改变已有磁盘的扇区长度同样是不可想象的)，在进行磁盘 I/O 操作时，这将导致严重的性能问题。

本文提出的优化方案是将不同页面中的热区分布到其所在页面的不同位置中，具体的做法是，各热区在页面内的位置可以通过其所在页面的编号对 N 取模来求得。

热区的位置在页面创建的同时被确定下来，为了保证后续访问的正确性，页面编号必须保持不变，在数据库系统中，可以选择该页面在磁盘(文件)中的编号，称为外存页面编号[5]。这样，对 P_H 中的任一内存访问 C_H，设其外存页面编号为 D_H，则其优化后 C_H 的位置可通过如下公式得出：

$$C_H = B_H * N + D_H \% N$$

经过这一变换，不同页面内的热区将由于其所在页面的外存页面编号不同而被分布到页面内的不同位置，这些内存块在访问时将被映射到不同的缓存组中，从而最大限度地利用了缓存。上述方法的优点在于对硬件没有特殊要求，同时由于只改变页面内结构，因此只需要修改数据库系统中与页面内操作有关的部分代码，对已有系统的改动很小。

3.2 实现

如果将重新分布后的热区的地址作为新的页面(逻辑页面)的起始位置，则上述优化技术实际上是将所有数据在原有的页面(物理页面)内移动了一个特定的偏移量。因此原有系统中所有与页面内操作有关的基本算法(包括数据恢复)和数据结构都可以保持不变，只需要在实际数据定位时，将数据地址(逻辑页面内的偏移量)加上逻辑页面起始位置在物理页面内的偏移量，来计算出该数据在物理页面中的位置，这种修改的开销非常小，因此并不会影响顺序访问及其它操作的性能。当然，修改后的系统必须保证任何记录(或索引对)都不会跨越物理页面的边界。

对于数据页面，可以在页面创建的同时在记录偏移表中分配一个“哨兵入口”，其对应记录的位置处于物理页面的边界，长度为0，该记录不能被应用存取；以后的操作则同优化前完全相同(如图2(a))。此时，页面的空闲部分被划分成了两块，且不会被合并，这就保证了以后创建的所有记录都不会跨越物理页面的边界。由于记录的长度通常远远小于页面长度，这种结构对系统的空间利用率也没有显著的影响。

对于索引页面，在根据上述优化技术计算出热区新的位置之后，还必须根据索引对的长度对其进行适当的调整，以保证每个索引对都不会跨越物理页面的边界(如图2(b)(c))。由于索引对的长度通常远远小于页面长度，因此调整后的各热区在物理页面中的位置仍然可以认为是随机分布的。

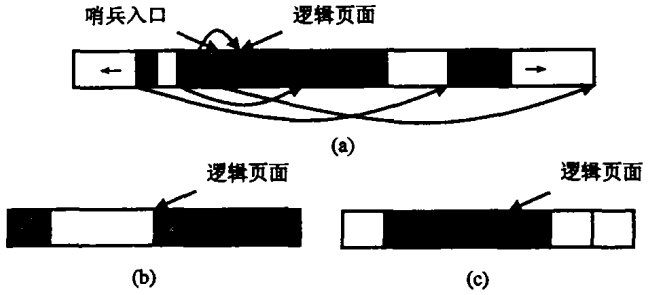

图2 优化后的页面结构示意图

4 性能测试

本文提出的优化技术已经在由我们自主开发的内存数据库系统——CoreBase 当中得到了实现，该数据库系统在863和航天工业总公司国防预研等多个项目中都得到了成功的应

3 在传统的 OLTP 数据库应用中，随机访问是大量存在的，可以参见 TPC-C 中的例子。

4 当 U 是2的整数次幂时，映射函数实际上只是将内块地址右移，其开销远远小于一般的取模操作。

5 选择内存页面编号 B_H 是不行的，在系统运行时，一个页面可能因为内存缓冲区的容量不足而被替换出去，当再次访问该页面时，系统无法保证将其读入到原来的内存页面中，因此页面的内存页面编号是经常变化的。

用。

本节首先来测试优化技术对 CoreBase 的性能改进。测试的硬件平台采用曙光 R220S，该平台包含两个1GHz 的 CPU，两个(4K，4，32)L2缓存和2Gb 内存，操作系统采用 Redhat Linux 7.1，页面长度被设置为4Kb。

测试过程如下：

1)在数据库中创建关系表 R，R 包含三个属性，其中 a1为主键[6]。

```
create table R (a1 integer primary key,
                a2 integer,
                a3 varchar(100))
```

2)产生一组记录并插入到 R 中，每条记录中 a1的值 key 从0开始顺序增长。

```
insert into R(a1)values(key)
```

3)随机生成一组固定数目(1000000个)的键值，对每个键值 key，通过索引在 R 中查找 a1的值为 key 的记录，并返回该记录 a2的值。所有的 key 都位于0和记录数目之间，以保证上述查找过程一定能成功。

```
select a2 from R where a1 = key
```

最后来考查完成上述查询过程的时间与记录数目的关系，其中记录数目在100～100000条之间变化[7]。为进行比较，测试过程分两遍进行，分别运行优化前后的系统。图3给出了测试结果。

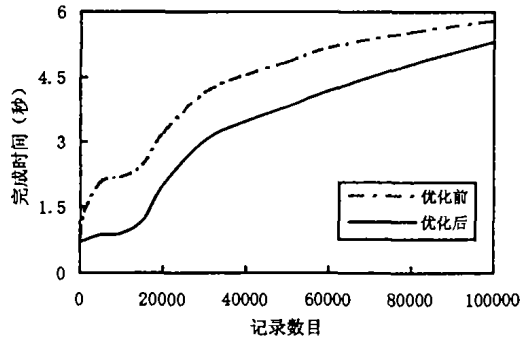

图3 特定查询的性能比较

测试结果表明：当记录数目较少(小于15000条)时，缓存争用构成了系统的主要开销，此时优化后的系统由于消除了绝大多数的缓存争用而使得其性能远远高于优化前的系统(提高50%～150%)；当记录数目较大时，由于缓存容量有限而造成的缓存未命中开始成为系统的主要开销，此时两者的性能都显著降低，其中优化后的系统的性能要比优化前的系统提高15%～25%，并随着记录数目的增加逐渐趋近于后者。

我们同时还改写了目前最常用的开放源代码数据库管理系统 Postgres 和 MySQL 中的相应代码，以便在这些系统中使用本文提出的优化技术(其中 MySQL 的数据存储模式与标准的 NSM 略有不同，但仍存在访问热区)，优化后的系统在执行上述查询时，性能平均提高了10%～15%。所有优化后的系统(包括 CoreBase)在执行其它查询(基于索引的简单连接操作、排序操作等)时的性能也有不同程度的提高。

结束语 本文的研究结果清楚地表明了缓存关联对数据库系统性能所具有的重要影响。本文详细地分析了在分页结构中，访问热区所造成的缓存争用问题，并在此基础上提出了一种优化技术。该技术实现简单，对已有系统改动小，并已经在实际系统中取得了很好的优化效果。

接下来的工作将通过运行一些更为复杂的性能测试基准(如 TPC-C 等)来考察优化技术对数据库系统的整体性能的改进；同时还将研究缓存关联对其它应用性能的影响，并提出相应的解决方案。

参 考 文 献

1 Bernstein P A，Brodie M L，Ceri S，et al. The Asilomar Report on Database Research. In：Ling Liu eds. SIGMOD Record. ACM，1998. 74～80

2 Boncz P A，Manegold S，Kersten M L. Database Architecture Optimized for the New Bottleneck：Memory Access. In：Atkinson M P，Orlowska M E，eds. Proc. of the Very Large Data Bases Conf. Edinburgh：Morgan Kaufmann，1999. 54～65

3 Ailamaki A，Dewitt D J，Hill M D，et al. DBMSs on a Modern Processor：Where Does Time Go? In：Atkinson M. P.，Orlowska M. E，eds. Proc. of the Very Large Data Bases Conf. Edinburgh：Morgan Kaufmann，1999. 266～277

4 Manegold S，Boncz P A，Kersten M L. What Happens During a Join? Dissecting CPU and Memory Optimization Effects. In：Abbadi A E，Brodie M L，eds. Proc. of the Very Large Data Bases Conf. Cairo：Morgan Kaufmann，2000. 339～350

5 Ailamaki A，Dewitt D J，Hill M D，et al. Weaving Relations for Cache Performance. In：Apers P. M. G.，Atzeni P，eds. Proc. of the Very Large Data Bases Conf. Roma：Morgan Kaufmann，2001. 169～180

6 Rao J，Ross K A. Cache Conscious Indexing for Decision-Support in Main Memory. In：Atkinson M. P.，Orlowska M. E，eds. Proc. of the Very Large Data Bases Conference. Edinburgh：Morgan Kaufmann，1999. 78～89

7 Rao J，Ross K A. Making B$^+$-Trees Cache Conscious in Main Memory. In：Chen W，Naughton J F，eds. Proc. of the ACM SIGMOD Conf. Dallas：ACM，2000. 475～486

8 Shatdal A，Kant C，Naughton J F. Cache Conscious Algorithms for Relational Query Processing. In：Bocca J. B.，Jarke M，eds. Proc. of the Very Large Data Bases Conf. Santiago：Morgan Kaufmann，1994. 510～521

9 Ramakrishnan R，Gehrke J. Database Management Systems. WCB/McGraw-Hill，2nd Edition，2000

10 Lomet D B. The Evolution of Effective B-tree：Page Organization and Techniques：A Personal Account. In：Ling Liu ed. SIGMOD Record. ACM，2001. 64～69

11 Gray J，Graefe G. The Five-Minute Rule Ten Years Later，and Other Computer Storage Rules of Thumb. In：Ling Liu ed. SIGMOD Record. ACM，1997. 63～68

6 CoreBase 在创建表 R 的同时会为属性 a1创建索引(B$^+$-Tree)以保证主键值的唯一性，其中索引页面内的二分法查找采用第二种实现方法，即比较所有索引对。

7 绝大多数实际系统中的表所包含的记录的数目都处于这个范围之中。