文章编号: 1001- 2486(2009) 04- 0068- 06

基于共享 Cache 多核处理器的 Radix 聚集连接优化*

邓亚丹, 景 宁, 熊 伟, 吴秋云
(国防科技大学 电子科学与工程学院, 湖南 长沙 410073)

摘 要: 基于目前主流的多核处理器, 研究了数据库 Radix-Join 算法中的聚集连接优化。针对多线程聚集连接执行时, 线程 Cache 访问缺失严重的问题, 采用预读线程预先将聚集连接线程需要访问的聚集对从内存读入 L2-Cache, 提高了线程的 Cache 访问性能。并根据聚集连接执行时的代价模型, 优化了聚集连接执行框架和各种线程参数。在实验中, 基于内存数据库 EaseDB 实现了本文提出了算法, 实验结果表明, 聚集连接性能得到较大提高。

关键词: 多核处理器; radix-join; 聚集连接
中图分类号: TP301 文献标识码: A

Radix Cluster Join Optimization Based on Shared Cache Chip Multi-processor

DENG Ya-dan, JING Ning, XIONG Wei, WU Qin-yun
(College of Electronic Science and Engineering, National Univ. of Defense Technology, Changsha 410073, China)

Abstract: Based on Chip Multi-Processor(CMP), this paper presents optimization of cluster join in Radix-Join algorithm. In order to solve the problem of serious cache may miss during the multithreaded cluster join execution, and to improve the performance of cache access, we adopt preload thread to read the clusters whose thread will access from memory to L2-Cache. Furthermore, based on the cost model of cluster join execution, the framework of cluster join execution and various thread parameters have been optimized. In the experiments, we implement the algorithm in EaseDB. The results show that cluster join performance is improved.

Key words: chip multi-processor; radix-join; cluster join

由于处理器访问内存的速度相对于 Cache 和寄存器差距很大, 处理器等待 Cache 从上一级存储器取数据造成的延迟成为数据库系统的主要瓶颈之一[1]。与此同时, 处理器的发展趋势已经从高主频的单核处理器转向片上多线程处理器, 特别是多核处理器(CMP, Chip Multi-Processor) 的普及, 使得并行计算成为程序性能提升的引擎。CMP 普遍采用的共享 Cache 体系结构可以提高 Cache 利用率, 但 CMP 的共享 Cache 体系结构给性能带来提升的同时, 也可能因共享 Cache 访问冲突导致线程的 Cache 访问性能下降, 因此很多学者已经展开基于 CMP 的数据库优化研究[1-6]。

1 相关研究现状

目前基于片上多线程处理器的各种数据库操作优化算法如下:

(1) 基于 CMP 的数据库优化。Parallel Buffer[2] 适用于多核处理器中, 多线程条件下的并发内存访问和管理; 文献[3] 基于 MTA- 2 多核处理器, 针对选择和连接运算, 测试多线程条件下的性能; 文献[4] 提出了基于 CMP 的自适应聚集算法; Pipelined Hash Join 算法[5] 基于多核处理器, 优化了采用流水线方式执行的 Hash 连接; 文献[6] 提出了多核处理器中的 MergeSort 算法优化。

(2) 基于 SMT 的数据库优化。文献[7] 基于 SMT 将页面中的元组分为奇偶两类在两个线程之间划分, 并行执行数据库操作, 并提出主线程和 Helper 线程相互协作模式, Helper 线程将主线程需要访问的数据从内存数据预读至 Cache。由于 SMT 只能同时执行两个线程, 所以其相关研究成果不适用于处理

* 收稿日期: 2009- 03- 19
基金项目: 国家 863 高技术研究发展计划重点资助项目(2007AA120400); 国家自然科学基金资助项目(40801160)
作者简介: 邓亚丹(1981—), 男, 博士生。

器核心数超过两个的 CMP。而且目前 CMP 已取代 SMT, 成为片上多线程处理器的主流, 所以基于 SMT 的数据库优化研究较少。

由于片上多核处理器成为市场主流的时间尚短, 因此相应的数据库优化研究较少。而已有的基于 CMP 的研究成果还未涉及 Radix 聚集连接的优化。

2 基于共享 Cache 多核处理器的聚集连接优化

表 1 给出了本节所涉及参数的意义。

表 1 常用参数

Tab. 1 Common symbol

参数	描述	参数	描述
C	Cache 容量(bytes)	L, R	参与连接的表, L.size, L.tuple 表示元组大小和元组数
B	Cache Line 大小(bytes)	*HitCost*	L2-Cache 命中时处理器访问延迟代价(Cycles)
N	处理器核心数	*MissCost*	L2-Cache 缺失时处理器访问延迟代价(Cycles)
ClusterSize	聚集大小(bytes)	*ComputeCost*	键值比较计算代价(Cycles)
ItemSize	聚集中数据项的大小(bytes)	*CJTNum*	聚集连接线程个数

2.1 Radix-Join 算法简介

Radix-Join 算法[8]提出通过 P 次划分将表分成 H ($H=\prod_{i=1}^{P}H_i$) 个聚集(聚集是指根据连接列 Hash 值的二进制位划分后形成的数据子集), 每个聚集一般都小于 L2-Cache 的容量, 从而大大减少 Cache Thrashing。在 P 次划分中, 从最左端位开始, 处理 Hash 值的 D 位数据($D=\sum_{i=1}^{P}D_i$)。每一次划分将一个聚集划分成 $H_P=2^{D_P}$ 个新聚集。根据聚集的大小决定两个连接表的聚集之间采用嵌套循环连接(NLJ)或 Hash 连接。现给出定义如下:

定义 1 *ClusterSet*(聚集集合) *ClusterSet*=(CL_1, CL_2, …, CL_m), 表示连接表在聚集划分后产生的聚集集合。*ClusterSetL*[0..*LNum*]和 *ClusterSetR*[0..*RNum*]分别表示参与连接的外表和内表对应的聚集集合。其中 CL=(*radixvalue*, *size*), 为聚集的元数据, *radixvalue* 和 *size* 分别表示该聚集的 *Radix Value* 值和聚集大小(byte)。聚集中存储的数据项 *Item* 为(*KeyValue*, *TID*), 分别表示连接的键值和元组 ID。

定义 2 *ClusterPairSet*(聚集对集合) *ClusterPairSet*={(l_1, r_1), (l_2, r_2), …, (l_q, r_q)}, 表示满足 *ClusterSetL*.*radixvalue*=*ClusterSetR*.*radixvalue* 条件的所有聚集 ID 对的集合。

2.2 基于多核处理器的聚集连接优化

一般情况下, 聚集数据量大大地超过了 Cache 容量, 使得聚集线程执行时 Cache 访问缺失严重。因此需要通过一个单独的线程负责将聚集线程未来一段时间内将要访问的数据从内存取到 Cache。

2.2.1 多线程聚集连接执行框架

以聚集连接在共享 L2-Cache 四核处理器中执行为例, 对应的多线程聚集连接执行框架如图 1 所示, 其中有两类线程: 聚集连接线程和 Preload 线程(预读线程)。聚集连接线程执行聚集对的 NLJ 运算, 预读线程则负责

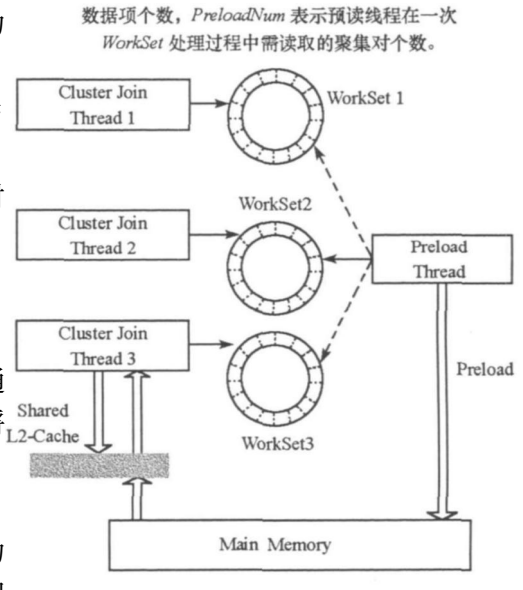

图 1 多线程聚集连接执行框架

Fig. 1 Multithreaded cluster join framework

将聚集连接线程将要访问的聚集对从内存取到L2-Cache。每个聚集连接线程对应一个 *WorkSet*。*WorkSet* 的定义如下:

定义3 *WorkSet*(预读线程工作集) *WorkSet* = (*DataItem*, *ItemNum*, *PreloadNum*), *DataItem* 为 *WorkSet* 数据项存储数组, *DataItem* = (*ClusterPairId*, *Status*), *ClusterPairId* 和 *Status* 分别表示需要预读线程读取的聚集对在 *ClusterPairSet* 中的标识和该聚集对是否已被预读线程读取。*ItemNum* 表示该 *WorkSet* 中存储的数据项个数, *PreloadNum* 表示预读线程在一次 *WorkSet* 处理过程中需读取的聚集对个数。

由于 *WorkSet* 被线程频繁访问,所以其结构及访问必须非常简单,因此 *WorkSet* 定义为数组结构,但按照环形方式进行访问。*ClusterPairSet* 中的聚集被均匀分配给各个聚集连接线程。*ClusterJoinThreadFun* 为对应的聚集连接线程的线程函数。

```
Algorithm 1 ClusterJoinThreadFun
    Input: ClusterPairSet, Start, End, ClusterSetL, ClusterSetR, WorkSet
    Output: TupleIDSet
BEGIN
    1. i= 0; j= 0; CPS= ClusterPairSet;
    2. L= ClusterSetL; R= ClusterSetR; Status= UnPreload;
    3. For i= 0 to WorkSet.ItemNum- 1
    4.    put (CPS[Start+ i], Status) to WorkSet.DataItem[i];
    5. PreloadPos= Start+ Workset.ItemNum;
    6. WSPos= 0; PLPos= Start+ WorkSet.ItemNum;
    7. For j= Start to End
    8.    If (PreloadPos< = End)
    9.       PreloadPos= PutDataToWS(PLPos, CPS, WorkSet);
    10.          Nested Loop Join L[CPS[j].l] and R[CPS[j].r];
    11.       add tuple ID result to TupleIDSet;
    End
```

ClusterJoinThreadFun 处理从 *ClusterPairSet*[*Start*] 至 *ClusterPairSet*[*End*] 中的聚集对连接。在聚集连接开始前,先将 *WorkSet* 填满,以供预读线程处理。在每次执行NLJ前, *PutDataToWS* 函数将 *WorkSet*.*PreloadNum* 个需要预读线程读取的聚集对ID放入 *WorkSet* 中。该函数第一次执行时从 *WorkSet*.*DataItem*[0]开始查找,如果存在 *WorkSet*.*DataItem*[*i*].*Status* 等于 *Preloaded*,则可将 *ClusterPairSet*[*PreloadPos*]存入 *WorkSet*.*DataItem*[*i*],如果找到 *WorkSet*.*PreloadNum* 个满足条件的 *DataItem* 则完成一次调用,并记录下一次查找的开始位置WSPos。当 *PutDataToWS* 再次执行时,从 *WorkSet*.*DataItem*[*WSPos*]开始查找,如果在 *WorkSet*.*DataItem*[*WSPos*]至 *WorkSet*.*DataItem*[*WorkSet*.*ItemNum*]中没有 *WorkSet*.*PreloadNum* 个满足条件的 *DataItem*,则继续遍历 *WorkSet*.*DataItem*[0]至 *WorkSet*.*DataItem*[*PreloadPos*- 1]中的数据形成环形访问方式,该函数每次调用最多只需一次完整的 *WorkSet*.*DataItem* 遍历。为了避免访问 *WorkSet* 的锁开销,聚集连接线程和 *Preload* 线程在访问 *WorkSet* 的数据项前无需申请加锁。

由于文献[7]指出处理器支持的预取指令(*Prefetch* 系列指令)存在性能缺陷,因此预读线程与文献[7]一样通过赋值语句达到将聚集从内存读取至L2-Cache的目的。具体为,令表 *L* 对应的聚集集合为 *ClusterSet* = (CL_1, CL_2, …, CL_m)。假设 CL_i 的起始地址为 p,预取时只需通过执行 *temp* = * ((*int**)*p*),即可将 p 所在的 B 个字节大小的数据取到 *Cache* 的某个 *Cache Line* 中。对于一个聚集 CL_i,只需要 $\lceil CL_i.size/B \rceil$ 次上述赋值操作,便能将 CL_i 中数据取到L2-Cache中。预读线程同样以环形方式访问 *WorkSet*,处理状态标识为UnPreload的聚集。在一次循环中处理所有聚集连接线程对应的 *WorkSet*,每次 *WorkSet* 遍历只需要读取 *WorkSet*.*PreloadNum* 个聚集对即可。如果预读线程当前处理的聚集ID号为分配给某个聚集连接线程的最后一个聚集对,完成该聚集对的读入操作后,预读线程即可无须处理该聚集连接线程对应的 *WorkSet*。当所有的聚集连接线程都完成处理时,预读线程即可结束。为了算法表达的简洁,

上述处理在算法中未体现。

2.2.2 多线程聚集连接代价模型

该模型表示聚集连接线程和预读线程处理一个聚集对时所需处理时间的比值。聚集连接线程对两个聚集执行 NLJ 运算时认为基本为 Cache 访问命中, 其执行代价取决于嵌套循环执行中的数据访问次数, 每次访问都需要一定的 Cache 命中代价和处理器计算代价, 可得

$$JTCost = \blacktriangledown LClusterSize/ ItemSize \lrcorner \cdot \blacktriangledown RClusterSize/ ItemSize \lrcorner \cdot (HitCost + ComputeCost) \quad (1)$$

预读线程处理一个聚集时, 则基本为 Cache 访问缺失, 其执行代价取决于内存读取次数和赋值语句的处理器执行时间, 可得

$$PTCost = (\ulcorner LClusterSize/ B \blacktriangle + \ulcorner RClusterSize/ B \blacktriangle) \cdot (MissCost + 1) \quad (2)$$

因此可得多线程聚集连接时的代价模型 *TimeRatio* 为

$$TimeRatio = JTCost/ PTCost \quad (3)$$

2.2.3 基于代价模型的多线程聚集连接执行框架优化

结合 *TimeRatio* 和处理器的核心数 N 优化线程的执行模式和预读线程参数。

线程执行模式优化 所谓线程执行模式是指启动的聚集连接线程和预读线程个数, 基本原则是使得预读线程预读聚集的进度超过聚集连接的处理速度(Algorithm1 中聚集连接线程首先将 *WorkSet* 填满即为此目的), 但聚集连接的执行线程也不能过少, 以免过度降低聚集连接执行的并行程度。$N = 2$ 时情况比较简单, 由图 4 的结果可知, 令 $JTNum = 1$, $PTNum = 1$ 即可。以下的讨论基于 $N > 2$。首先聚集连接线程数一定要大于或等于预读线程数, 以保证聚集连接操作有足够的并行性, 即

$$JTNum \geqslant PTNum \text{ 且 } JTNum + PTNum = N \quad (4)$$

由下节实验 3 可知, 线程 Cache 访问性能的改善可以大大提高线程的性能, 因此同时又需要保证有足够的预读提前量。由 *TimeRatio* 的定义可知, 在只有一个预读线程情况下, 预读线程处理一个 *WorkSet* 时可以预读 $TimeRatio/ JTNum$ 个聚集对, 当有 $PTNum$ 个预读线程时, 每次可以预读的聚集对平均为 $TimeRatio \cdot PTNum/ JTNum$。为使得预读线程预读的进度超过聚集连接的处理速度, 可得

$$TimeRatio \cdot PTNum/ JTNum \geqslant 1 \quad (5)$$

下面分为 3 种情况, 根据处理器的核心数和 *TimeRatio* 值设置具体的线程执行模式策略:

(1) $TimeRatio < 1$ 时。式(5) 成立必须使得 $PTNum > JTNum$, 而由实验 3 可知, 聚集连接执行的并行程度的下降会严重抵消 Cache 访问性能改善带来的优势, 此时式(4) 成立的优先级要高于式(5), 比如 $TimeRatio = 0.9$, $N = 4$, 可设置 $JTNum = 2$, $PTNum = 2$。

(2) *TimeRatio* 较大, N 较小时。比如 $TimeRatio = 9$, $N = 4$, 结合式(4) 和式(5), 可计算出 $JTNum = 3$, $PTNum = 1$。

(3) N 较大时, 比如六核、八核处理器。当 $TimeRatio \leqslant N$ 时, 比如 $TimeRatio = 5$, $N = 8$, 可得 $JTNum = 6$, $PTNum = 2$。在上述计算过程中, $JTNum$ 的优先级高于 $PTNum$。比如 $JTNum = 5$, $PTNum = 3$ 时也满足式(4) 和式(5), 但需选择 $JTNum$ 较大的执行模式。当 $TimeRatio > N$ 时, 表明一个 *Preload* 线程即可满足 $N - 1$ 个聚集连接线程执行所需要的提前量, 一般可令 $JTNum = N - 1$, $PTNum = 1$。

预读线程参数设置 对于预读的参数 *WorkSet*. *PreloadNum*, 由式(5) 的分析可得 $WorkSet.PreloadNum = \blacktriangledown TimeRatio \cdot PTNum/ JTNum \lrcorner$。

2.2.4 *WorkSet* 设置优化

在目前内存带宽的条件下, 对数据库这种数据密集型程序而言, 在一个线程时间片段内(10~100ms), 聚集连接执行框架中的所有线程从内存读入 L2-Cache 的数据接近或者超过目前 L2-Cache 容量。因此需要设置合理的 *WorkSet*. *ItemNum* 值, 以防止预读线程预读过多数据, 减少多线程执行时的共享 Cache 访问冲突。$JTNum$ 个 *WorkSet* 对应的聚集对数据量 $TotalDataSize = JTNum \cdot WorkSet.ItemNum \cdot B \cdot \ulcorner ClusterPairSize/ B \blacktriangle$。由实验 2 可知, 如果 *WorkSet*. *ItemNum* 能够满足预读线程的需要, 聚集连接的性能便会趋于平稳。由于目前的 L2-Cache 容量较大, 令 $TotalDataSize < Ratio \cdot C$ ($Ratio < 1$), 便可满足预读线

程的需求,而且即使预读线程在短时间内预读完所有 *WorkSet* 中的聚集对,从内存读入 L2-Cache 的数据量 *TotalDataSize* 也不会超过 L2-Cache 的容量,从而减少 Cache 访问冲突。具体的 *Ratio* 取值见实验 2。

3 实验结果与分析

3.1 实验设置

本文的研究和实验基于共享 Cache 双核和四核处理器展开,Cache 容量有 3MB 和 4MB 两种。实验数据库平台为内存数据库 EaseDB[11],实验时执行两个表的连接运算,测试两个表对应聚集的连接性能。数据设置与文献[12]相同,两个表中的列都为整型,数据由随机函数生成。实验数据为 $DataSet^i = (L.tuple, R.tuple) = \{a(5.26e+10^5, 7.0e+10^5), b(1.17e+10^6, 1.4e+10^6), c(1.87e+10^6, 2.8e+10^6), d(4.0e+10^6, 5.6e+10^6), e(7.43e+10^6, 1.12e+10^7)\}$, $i = 1, \cdots, 5$, $ItemSize = 8B$。

3.2 实验内容及结果分析

实验 1 测试 *WorkSet* 访问方式性能比较。执行的 *WorkSet* 访问方式有两种:(A)本文提出的访问方式;(B)访问 *WorkSet* 中数据项时,聚集连接线程和预读线程都需要执行加锁和解锁操作。由图 2 可知,模式 A 由于避免了频繁的加锁/解锁开销,虽然可能存在"脏数据"读取,但性能仍优于模式 B。

实验 2 测试 *WorkSet* 参数性能测试。首先测试不同 *WorkSet.ItemNum* 取值时聚集连接的性能,处理器为 4MB 共享 Cache 四核处理器。执行的聚集连接为 $DataSet^i$ ($i = 3, \cdots, 5$),对应的平均聚集对大小 $ClusterSize^i = (218, 252, 304)$,对应的线程执行模式均为 $JTNum = 3$, $PTNum = 1$。如图 3 所示,如果 *ItemNum* 过小,则会导致预读的提前量减小,聚集连接线程的 Cache 缺失随之加大;而如果 *ItemNum* 过大,*TotalDataSize* 则会接近或者超过 L2-Cache 容量,随着 *TotalDataSize* 的增大,Cache 访问冲突也随之加剧,导致性能下降比较严重。如果 *ItemNum* 取值适中,聚集连接性能趋于平稳,与 *ItemNum* 大小基本无关。各个聚集连接执行时,聚集连接性能平稳时对应的 *TotalDataSize* 大概为 L2-Cache 容量的一半,所以 2.2.4 节中的 $Ratio = 0.5$。

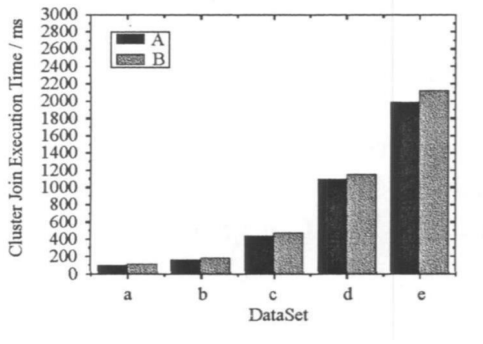

图 2 *WorkSet* 访问方式性能比较

Fig. 2 Performance comparison of Workset pattern

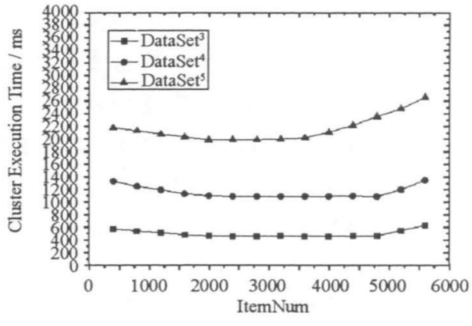

图 3 *WorkSet* 参数性能测试

Fig. 3 Performance test of Workset parameter

实验 3 测试各种线程执行方式的性能。首先测试数据预读对聚集连接性能的影响,为了突出聚集连接线程因 Cache 访问性能带来的性能提升,测试所用处理器为 3M 共享 Cache 双核处理器。测试时执行三种线程模式:(A)一个聚集连接线程,一个预读线程;(B)2 个聚集连接线程;(C)单个聚集连接线程,执行的聚集连接为 $DataSet^i$ ($i = 1, \cdots, 5$)。如图 4 所示,在双核处理器中,模式 A 中的聚集连接线程的 Cache 访问性能明显要优于模式 B,尽管模式 B 的聚集连接并行程度高于模式 A,但由于其聚集连接线程的 Cache 访问性能较差,导致聚集连接性能有所下降。而模式 C 的性能要明显低于其他两种模式。接下来测试四核处理器中聚集连接的性能。执行的聚集连接为 $DataSet^i$ ($i = 3, \cdots, 5$),平均聚集对大小 $ClusterSize^i = (218, 252, 304)$,由式(3)计算出 $TimeRatio = (7.1, 9.4, 10.1)$,根据 2.2.3 节的分析可知,对应的多线程执行模式(*JTNum*, *PTNum*)都为(3,1)。测试时执行的多线程模式(*JTNum*, *PTNum*)为(A)(3,1);(B)(2,2);(C)(4,0);(D)(1,3)。如图 5 所示,模式 B 和模式 D 由于分配了过多的计算资源给预

读线程, 导致部分处理器核心大部分时间处于空闲状态, 聚集连接的并行程度下降, 性能较模式 A 有较大下降。模式 C 即采用传统的多线程执行模式, 未利用预读线程进行 Cache 优化, 虽然其聚集连接的并行程度最高, 但模式 C 执行时聚集连接线程的 Cache 缺失访问严重, 导致其性能较模式 B 仍有一定差距。

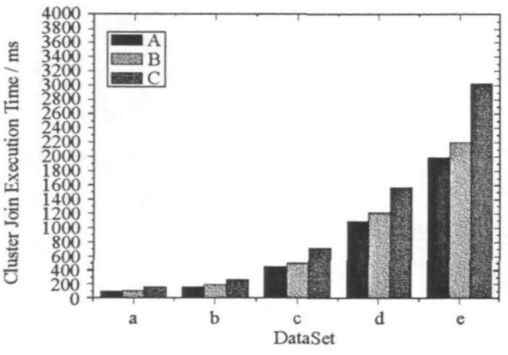

图 4 双核处理器聚集连接性能测试
Fig. 4 Performance test of cluster join

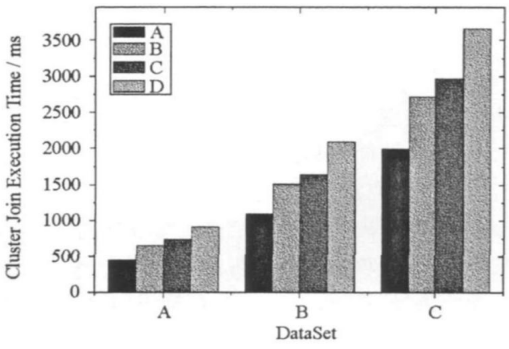

图 5 多线程执行模式性能比较
Fig. 5 Performance comparison of multithreaded pattern

4 结 论

针对共享 Cache CMP, 提出了 Radix 聚集连接的多线程执行框架。通过预读线程将聚集线程要访问的聚集对从内存读入 Cache, 从而大大减少了聚集连接线程执行时的 Cache 访问冲突; 并基于聚集连接的代价模式, 设置合理的线程执行模式, 优化线程执行参数。在实验中, 基于内存数据库 EaseDB 实现上述框架和各种优化措施。实验表明, 本执行框架能够充分利用多核处理器的计算优势, 提高聚集连接线程执行时的 Cache 访问性能, 较大地提高了聚集连接的性能。

参 考 文 献:

[1] Hardavellas N, Pandis I, Johnson R. Database Servers on Chip Multiprocessors Limitations and Opportunities[C]// 3rd Biennial Conference on Innovative Data Systems Research, On-Line Publication, www. crdrdb. org, 2007: 79- 87.

[2] Cieslewicz J, Ross K A, Giannakakis I. Parallel Buffer for Chip Multiprocessors[C]// 3th International Workshop on Data Management on New Hardware, NY: ACM, 2007: 1- 10.

[3] Cieslewicz J, Berry J, Hendrickson B, et al. Realizing Parallelism in Database Operations: Insights from a Massively Multithreaded Architecture [C]// 2th International Workshop on Data Management on New Hardware, NY: ACM, 2006.

[4] Cieslewicz J, Ross K A. Adaptive Aggregation on Chip Multiprocessors[C]// Proceedings of the 33th International Conference on Very Large Databases, . NY: ACM, 2007: 339- 350.

[5] Garcia P, Korth H F. Pipelined Hash-join on Multithreaded Architectures[C]// 3th International Workshop on Data Management on New Hardware, NY: ACM, 2007.

[6] Chhugani J, Macy W, Baransi A, et al. Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture[C]// The International Conference on Very Large Database, NY: ACM, 2008: 1313- 1324.

[7] Zhou J R, Cieslewicz J, Ross K. Improving Database Performance on Simultaneous Multithreading processors[C]// Proceedings of the 31st VLDB Conference, NY: ACM, 2005: 49- 60.

[8] Boncz P, Manegold S, Kersten M K. Database Architecture Optimized for the New Bottleneck: Memory Access[C]// Proceedings of the 25th VLDB Conference, NY: ACM, 1999: 231- 246.

[9] Deng Y D, Jing N, Xiong W. Hash Join Optimization Based on Shared Cache Chip Multi-processor[C]// 14th International Conference of Database Systems for Advanced Applications, Berlin: Springer, 2009: 293- 307.

[10] He B S, Luo Q. Cache-oblivious Nested Loop Joins[C]// Proceedings of the 2006 ACM International Conference on Information and Knowledge Management, NY: ACM, 2006: 718- 727.

[11] He B S, Luo Q. Cache-oblivious Database: Limitations and Opportunities[J]. ACM Transactions on Database Systems, 2008, 33(2).

[12] He B S, Luo Q. Cache-oblivious Hash Joins[R]. HKU Technical Report, 2006.