DBCC-Join：一种新的高速缓存敏感的磁盘连接算法

韩希先[1)] 杨东华[2)] 李建中[1)]

[1)](哈尔滨工业大学计算机科学与技术学院 哈尔滨 150001)

[2)](哈尔滨工业大学基础与交叉科学研究院高性能计算中心 哈尔滨 150001)

摘 要 随着 CPU 和内存的性能差距越来越大，系统设计者在 CPU 寄存器和内存之间插入高速缓存来弥补这个差距.高速缓存的数据存取速度远高于内存，所以数据库操作要获得更好的性能就必须考虑充分利用高速缓存.基于磁盘的连接操作是一种常用并且耗时的数据库查询操作，可是大多数传统的连接算法在设计时都没有考虑高速缓存的使用，从而使得这些连接算法无法充分利用 CPU 的能力.文中分析了传统的连接算法在高速缓存利用方面的问题，并且提出了一种新的可以充分利用高速缓存的磁盘连接算法 DBCC-Join.连接位置索引对表 JPIPT 是用到的数据结构，说明了每个连接结果元组在各自表中的位置索引对.DBCC-Join 的执行包括两个阶段：JPIPT 构建阶段和结果输出阶段.JPIPT 构建阶段对列存储化的连接属性执行高速缓存敏感的算法来构建连接位置索引对表.利用获得的 JPIPT，结果输出阶段只需要对数据表执行一遍顺序扫描就可以获得结果.该文是第一篇提出利用高速缓存的磁盘连接算法的文章.实验表明，和传统磁盘连接算法相比，DBCC-Join 算法可以获得一个数量级的加速比.

关键词 DBCC-Join；JPIPT 构建阶段；结果输出阶段；缓存敏感算法

中图法分类号 TP311 **DOI 号**：10.3724/SP.J.1016.2010.01500

DBCC-Join：A Novel Cache-Conscious Disk-Based Join Algorithm

HAN Xi-Xian[1)] YANG Dong-Hua[2)] LI Jian-Zhong[1)]

[1)](*School of Computer Science and Technology, Harbin Institute of Technology, Harbin* 150001)

[2)](*Center for High Performance Computing, The Academy of Fundamental and Interdisciplinary Sciences, Harbin Institute of Technology, Harbin* 150001)

Abstract System designers exploit cache to make up for performance gap between CPU and main memory. Since data access speed of cache is much faster than that of memory, it is important for database operations to take maximum advantage of cache to obtain higher performance. Disk-based join operation is a common but time-consuming database operation. Unfortunately, most of traditional join algorithms do not take cache into consideration. This paper analyzes low cache utilization problem in traditional join algorithms and proposes a disk-based cache-conscious join algorithm DBCC-Join. Join positional index pair table (JPIPT) is a data structure which specifies the positional index pairs of join tuples in each table. The execution of DBCC-Join consists of two stages: JPIPT construction stage and result output stage. JPIPT construction stage performs cache-conscious construction algorithm on join attributes which are kept in column-oriented model, to obtain join positional index pair table of join results. The obtained JPIPT is used in result output stage to retrieve results in a one-pass sequential scan on each table. To the best of our knowledge, this paper is the first to exploit cache to improve performance of disk-based join algorithm. Experimental results show that compared to traditional join algorithms, DBCC-Join can be improved by a factor of an order of magnitude.

收稿日期：2010-06-11.本课题得到国家“九七三”重点基础研究发展规划项目基金(2006CB303005)、国家自然科学基金(60903016，60533110，60773063)、新世纪优秀人才支持计划(NCET-05-0333)、黑龙江省教育厅科学技术研究项目(11531276)和 NSFC-RGC of China (60831160525)资助.**韩希先**，男，1981 年生，博士研究生，主要研究方向为海量数据管理和数据密集型计算. E-mail：xxhan1981@163.com. **杨东华**，男，1976 年生，博士，讲师，主要研究方向为海量数据管理和数据密集型计算等. **李建中**，男，1950 年生，教授，博士生导师，主要研究领域为数据库和传感器网络等. E-mail：lijzh@hit.edu.cn.

Keywords　DBCC-Join；JPIPT construction stage；result output stage；cache-conscious algorithm

1　引　言

摩尔定律预测 CPU 的处理速度每 18 个月增长一倍，主存的存取速度也有类似的定律，可是其增长速度却比 CPU 的要慢得多（每 6 年提高一倍）. 如图 1 所示[1]，这使得 CPU 和内存之间的性能差异越来越大（每 21 个月差距增大一倍）. 系统设计者在 CPU 寄存器和主存之间插入高速缓存来弥补这个差距. 高速缓存的数据存取速度远高于内存，如果数据存储在高速缓存中，引用该数据需要几个 CPU 时钟周期；如果数据存储在主存，则需要几十个甚至几百个时钟周期才可以访问该数据. 所以，数据库操作要获得更好的性能，就必须考虑充分利用高速缓存[2].

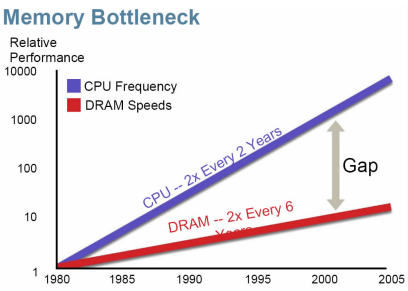


图 1　CPU 和内存的性能差距

连接操作是一种常用的关系数据库查询操作，它可以根据公共信息将两个或多个表的元组组合起来. 连接操作也是数据库操作中最耗时的操作之一，其性能直接影响着数据库的整体性能. 连接操作已经得到了广泛和深入的研究[3]，并且已经提出了一些经典的连接算法：nested-loop 方法、Sort-merge 连接方法和基于 hash 的连接方法.

大多数传统的连接算法在设计时没有考虑高速缓存的使用，从而使得这些连接算法无法充分利用 CPU 的能力[4-5]. 现有的研究工作提出了新的利用高速缓存的内存连接算法[6]，可是这些方法只适用于内存数据库. 而在实际应用中，用户需要更多的是基于磁盘的连接操作.

本文分析了传统磁盘连接算法在高速缓存利用中存在的问题，并且提出一种新的高速缓存敏感的磁盘连接算法 DBCC-Join（Disk-Based Cache-Conscious Join）. 该算法解决了现有连接算法的低缓存利用率问题. 在本文中，数据表根据列存储模式维护. 连接位置索引对表 JPIPT（Join Positional Index Pair Table）是 DBCC-Join 用到的数据结构，说明了每个连接结果元组在各自数据表中的位置索引对. DBCC-Join 的执行分为两个阶段：JPIPT 构建阶段 JCS（JPIPT Construction Stage）和结果输出阶段 ROS（Result Output Stage）. JCS 类似于普通的连接操作，可是该阶段只处理连接属性. 本文提出一种缓存敏感的构建算法来快速构建 JPIPT. ROS 的执行包括 3 个连续操作：JPIPT 划分、外关系扫描和内关系扫描. 我们设计了元组抽取算法来保证只需要对关系表执行一次顺序扫描就可以获得结果. 本文还提供了 DBCC-Join 算法的正确性证明. 需要注意的是，JCS 要求连接属性列存储化，可是在 ROS 中，只要存储模型支持利用位置索引来抽取元组，DBCC-Join 就可以应用于该存储模型. 这实际上包括了目前的所有存储模型（行存储和列存储）.

本文作者设计了较全面的实验分别从元组数量、表宽度、输出属性数量和连接选择度 4 个方面来评价 DBCC-Join 的性能. 实验表明，和传统磁盘连接算法相比，DBCC-Join 算法可以获得一个数量级的加速比.

本文的主要贡献如下：

（1）本文分析了传统磁盘连接算法在高速缓存利用中存在的问题，提出一种新的高速缓存敏感的磁盘连接算法 DBCC-Join. 据我们所知，本文是第一篇提出利用高速缓存的磁盘连接算法的文章.

（2）本文提出一种高速缓存敏感的 JPIPT 构建算法来获得连接位置索引对表.

（3）利用获得的 JPIPT，本文提出元组抽取算法只需要对数据表执行一遍顺序扫描来获得结果，并且提供理论分析证明了算法的正确性.

（4）本文通过较全面的实验来评价 DBCC-Join 的性能. 实验结果表明，和传统的磁盘连接算法相比，DBCC-Join 算法可以获得一个数量级的加速比.

本文第 2 节介绍高速缓存结构；第 3 节分析传统 Join 算法在利用高速缓存方面的问题；DBCC-Join 算法在第 4 节中详细描述；第 5 节是性能评价部分；相关工作和结论分别在第 6 和第 7 节中给出.

2　高速缓存结构

计算机读取数据的顺序是先查看需要的数据是否在寄存器，如果不在就继续检查是否在高速缓存，如果也不在就继续检查是否在内存和磁盘. 如果

CPU 请求的数据存储在高速缓存，称为缓存命中(cache hit)，否则称为缓存不命中(cache miss)．高速缓存包括 L1 高速缓存和 L2 高速缓存．本文主要考虑 L2 高速缓存，因为 L2 cache 比 L1 cache 要大得多，并且 L1 data cache miss 对性能的影响要远小于 L2 data cache miss 的影响[5]．图 2 表示了存储器的层次结构．高速缓存由昂贵的静态 RAM 构成，所以其容量一般不大，一个桌面系统的高速缓存一般不会超过几 M 字节．高速缓存被划分成一组固定大小的高速缓存行(cache line)，其大小在 16～256 字节．当高速缓存从主存中读取数据时，即使只需要一个字节，高速缓存也会从主存中读取可以填满整个 cache line 的数据．

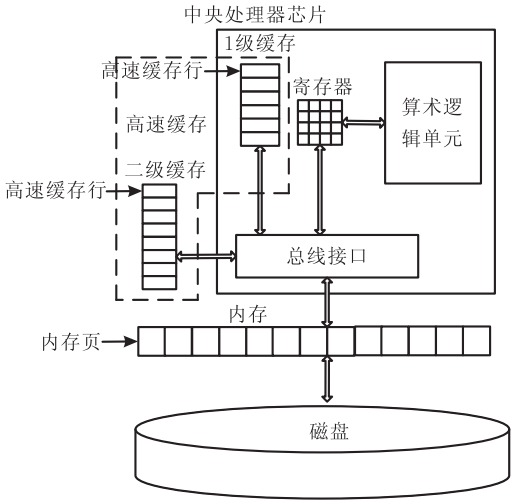


图 2 存储器层次结构

如果计算机程序具有较好的局部性，即程序倾向于访问相同的数据项集合，或者倾向于访问临近的数据项集合，则系统就可以很好地利用缓存来提高程序的性能．与主存和磁盘不同，人们无法直接利用软件来控制哪些数据放置到高速缓存，因为高速缓存是由硬件控制的．所以要充分利用高速缓存，我们的程序就必须具有良好的局部性，根据高速缓存的数据读取特点来设计程序操作．本文我们主要考虑如何利用缓存来提高磁盘连接操作的性能．

3 当前连接算法利用高速缓存的问题

连接操作是数据库系统中的一个重要并且耗时的操作，其性能对于整个数据库系统的性能有着很大的影响．人们提出了很多的连接算法：nested-loop join、sort-merge join 和 hash join 算法等．Hash join 由于其较好的性能更多地被人们所采用，所以本文以 hash join 为例来讨论连接算法．本文讨论的 hash join 的形式是 GRACE hash join[3]，一方面因为 GRACE join 具有严格区分的划分阶段和连接阶段，方便我们对性能的分析，另一方面由于我们处理的数据量较大，hybrid hash join[3] 和 GRACE hash join 的性能差距不大．本节中，我们分析 hash join 算法在利用高速缓存方面存在的问题．

(1) Tuple-oriented 处理模式．当前主流数据库采用行存储模式，同一元组的数据连续存储，并且采用 tuple-oriented 处理模式．在执行查询处理时，tuple-at-a-time 迭代器每次读取一个元组，提供给上层的关系操作符进行处理．假设高速缓存的 cache line 大小是 W_{cl}，关系的元组宽度是 W_{tuple}，在 tuple-at-a-time 迭代器模型下，高速缓存读取 tuple 的过程如图 3 所示．高速缓存每次执行主存读时，会同时获得 W_{cl} 字节的数据放入 cache line，加快接下来的读取操作．所以，执行一次内存读，高速缓存会读取 W_{cl}/W_{tuple} 个元组．典型的 cache line 大小是 64 字节，而数据表的属性数量一般都较大，几十个属性的数据表是很常见的．一般来说 $W_{cl}<W_{tuple}$，使得每次读取下一个元组都需要执行主存读操作，所以 tuple-oriented 处理模型并没有很好地利用高速缓存．

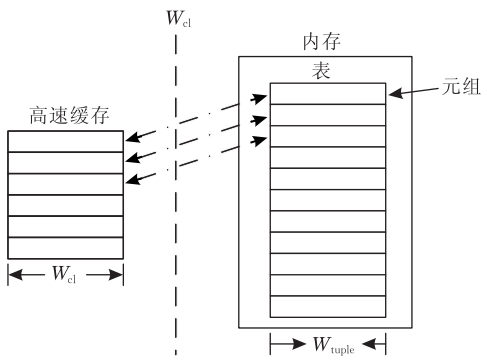


图 3 tuple-at-a-time 迭代模型的 cache 行为

(2) 较大的内关系块．Hash join 划分阶段把内关系划分成多个块，使得每个块都小于内存的容量，再对外关系执行相应的划分，划分后内关系的每个块通常比高速缓存大得多．内关系构建 Hash 表完毕后，外关系的元组开始探测 Hash 表以得到连接结果．内关系构建 Hash 表时，为保证外关系探测 Hash 表的效率，Hash 表的桶数较大．由于 Hash 函数的随机性以及较大的内关系块，当前的内关系元组映射到的 Hash 桶所在的数据页一般不在高速缓存，即每次内关系元组插入 Hash 表的操作通常引起内存读．同样，在外关系探测阶段，被探测桶所在的数据页一般也不在高速缓存中，从而使得每次探

测 Hash 表的操作都会引起一个内存读. 基于同样的原因，hash join 的 Hash 表构建和探测过程中，当前需要读取的数据页很可能没有存储在 TLB（翻译后备缓冲器，存储最近处理过的 64 个虚拟页-物理页地址影射，如果当前请求的虚拟页地址转换没有存储在 TLB，则说发生一次 TLB miss 操作），从而使得每次主存读写操作还需要执行虚拟页-物理页地址转换，进一步增加了其执行费用.

我们知道，具有良好局部性的程序比局部性差的程序运行得更快. 局部性通常有两种形式：时间局部性和空间局部性. 时间局部性指的是重复引用相同的数据项集合，而空间局部性指的是倾向于引用当前访问数据附近的数据项集合. 很明显，连接操作不具有良好的时间局部性，因为它处理完一个元组后就继续处理下一个元组. 我们利用步长为 k 的引用模式来讨论空间局部性，具有步长为 k 的引用模式指的是顺序访问连续空间第 k 个字节的数据，k 表示跳过的字节数. 步长越小，则空间局部性就越好，在存储器中以大步长访问不同位置的数据的程序不具有良好的空间局部性. 处理每个元组时，连接操作具有步长为 W_{tuple} 的访问模式，W_{tuple} 的值一般较大，则关系的顺序扫描操作不具有良好的空间局部性. Hash 函数的随机性以及较大的内关系块使得构建和探测 Hash 表具有较大的步长，所以构建和探测 Hash 表的操作不具有良好的空间局部性.

总的来看，现有的 hash join 算法不具有良好的局部性，从而无法充分利用高速缓存的能力. 下一节中，我们提出一种新的高速缓存敏感的磁盘连接算法 DBCC-Join，该算法可以充分利用高速缓存以提高磁盘连接操作的性能.

4　DBCC-Join 算法

通过分析，我们知道传统连接算法较低的高速缓存利用率的主要原因在于：较大的引用步长和较大的内关系分片. DBCC-Join 通过减少引用步长和内关系分片大小的方法来获得较好的高速缓存利用率. 减小引用步长的一个方法是不采用面向元组的处理模式，DBCC-Join 算法采用列存储策略[7]存储关系，关系不是一行行地连续存储，而是把关系的每个属性的值顺序存储到单独的文件. 采用列存储的好处不但可以减小扫描操作的引用步长，在查询处理阶段，还可以只处理相关的属性，而忽略其他的属性.

DBCC-Join 的执行分为两个阶段：JPIPT 构建阶段（4.1 节）和结果输出阶段（4.2 节）. 4.1 节介绍如何利用高速缓存来快速构建 JPIPT. 利用获得的 JPIPT，4.2 节介绍了如何获得最终的连接结果.

4.1　JPIPT 构建阶段 JCS（JPIPT Construction Stage）

假设 T_1 和 T_2 是参与连接操作的数据表. 在执行阶段，T_1 作为外关系而 T_2 作为内关系. 用户提交的查询 Q 的形式为“select attribute-list from T_1，T_2 where $T_1.key=T_2.key$”. 我们首先给出本文用到的一些定义.

定义 1（位置索引）. 给定表 T，元组 $t\in T$ 的位置索引 PI（Positional Index）是 i，如果 t 是 T 的第 i 个元组.

我们用 $T(i)$ 来表示表 T 中位置索引为 i 的元组.

定义 2（连接位置索引对表）. 给定表 T_1 和 T_2，$T_1.key_1$ 和 $T_2.key_2$ 分别是 T_1 和 T_2 的连接属性，$T_1\bowtie T_2$ 包括 N 个结果，JPIPT 是 $T_1\bowtie T_2$ 的连接位置索引对表，如果 JPIPT 满足条件：(1) $|\text{JPIPT}|=N$，(2) $\forall i,j,(PI_1:i,PI_2:j)\in\text{JPIPT}$，当且仅当 $T_1(i).key_1=T_2(j).key_2$.

在本文中，我们把 JPIPT 看作具有两个属性（PI_1，PI_2）的表. DBCC-Join 把 T_1 和 T_2 的连接属性存储为两个独立的列文件：C_1 和 C_2. 每个列文件都包含一个隐含的属性：PI. C_1 和 C_2 的模式可以分别看作 $C_1(key_1,PI_1)$ 和 $C_2(key_2,PI_2)$，其中，key_1 和 key_2 分别是 T_1 和 T_2 的连接属性，PI_1 和 PI_2 分别表示 key_1 和 key_2 在表 T_1 和 T_2 的位置索引. JPIPT 的构建操作就是执行如下的连接操作：“select PI_1，PI_2 from C_1，C_2 where $C_1.key_1=C_2.key_2$”.

传统的磁盘 hash join 并没有考虑高速缓存，从而导致了较低的高速缓存和 CPU 利用率. 本文提出高速缓存敏感的 JPIPT 构建算法 C3-JPIPT 算法（Cache-Conscious Construction of JPIPT），该算法充分利用高速缓存来更快地构建 JPIPT，其执行过程如算法 1[8]所示. 注意：本文涉及的所有算法的伪代码都包含在文献[8]中.

C3-JPIPT 的执行包括两个步骤：划分和连接. 划分把每个表划分成 $N_{p1}=\dfrac{2\times(S_{c_1}+S_{c_2})}{M}$ 个子表 $\{C_{1i}\}$ 和 $\{C_{2i}\}$（$1\le i\le N_{p1}$）. 其中，S_{C_1} 和 S_{C_2} 分别表示列文件 C_1 和 C_2 的大小，M 表示 C3-JPIPT 利用的内存容量. C3-JPIPT 产生比传统 Hash 连接操作更多的分片数量，这是因为该算法的连接阶段要求同时把内关系和外关系的单个子表对读入内存. 在划分时，C3-JPIPT 实例化每个元组的位置索引值. 注意：如果和外关系相比，内关系非常小，那么 JCS 就采用传统的连接算法而不是 C3-JPIPT.

当 C3-JPIPT 进入连接操作时，每一对子表 C_{1i} 和 C_{2i} 依次读入内存，并且划分为 $N_{p2}=\frac{S_{C_{2i}}}{C}$ 个分片，使得每个内关系子表的分片可以放入高速缓存. 其中，$S_{C_{2i}}$ 表示读入内存的内关系子表大小，C 表示高速缓存的大小. 如果 $N_{p2}>64$，则该划分操作采用多轮划分方法来减少 TLB miss. 每一轮划分的分片数量少于 64，并且前一轮获得的分片进一步划分成多个子分片. 该过程继续执行直到获得的分片数量达到 N_{p2}. C3-JPIPT 采用的多轮划分方法类似于 Radix-cluster 算法[6].

经过第二次的划分，C_{1i} 和 C_{2i} 分别被划分为两个含有 N_{p2} 个分片的集合. Hash 连接操作在 C_{1i} 和 C_{2i} 的每个分片对上执行. 因为当前的内关系的分片可以放入高速缓存，所以构建和探测 Hash 表的操作都可以在高速缓存中执行. C_{1i} 和 C_{2i} 的第 j 分片对的连接结果是第 j 分片对对应的连接位置索引对表 $JPIPT_{ij}$. 我们在内存中维护 $JPIPT_{ij}$ 以方便接下来的操作.

定理 1. 产生的 $JPIPT_{ij}$ 根据 PI_1 域排序.

证明. $\forall k_1,k_2(k_1<k_2),(PI_1:m_{k_1},PI_2:n_{k_1})\in JPIPT_{ij},(PI_1:m_{k_2},PI_2:n_{k_2})\in JPIPT_{ij}$. 假设在 $JPIPT_{ij}$ 中，$(PI_1:m_{k_1},PI_2:n_{k_1})$ 的出现要早于 $(PI_1:m_{k_2},PI_2:n_{k_2})$. 这意味着，读取 $T_1(m_{k_1})$ 的时间不能晚于读取 $T_1(m_{k_2})$ 的时间. 已知在连接操作中，T_1 作为外关系，并且 T_1 的元组按顺序读取来探测 Hash 表，所以 $m_{k_1}\leq m_{k_2}$. 证毕.

C_{1i} 和 C_{2i} 的连接操作产生 N_{p2} 个连接位置索引对表. 根据定理 1，每个 $JPIPT_{ij}$ 根据 PI_1 域排序. 我们利用一次多路合并排序操作来合并 $JPIPT_{ij}$ $(1\leq j\leq N_{p2})$ 获得 $JPIPT_i$，$JPIPT_i$ 输出到磁盘. 类似的操作在每个 C_{1i} 和 $C_{2i}(1\leq i\leq N_{p1})$ 上执行，获得 N_{p1} 个 $JPIPT_i$ 文件. 此时，再一次执行多路合并排序来合并 $JPIPT_i(1\leq i\leq N_{p1})$ 获得最终的 JPIPT.

4.2 结果输出阶段 ROS(Result Output Stage)

$(PI_1:i,PI_2:j)\in$JPIPT 意味着 $T_1(i)$ 和 $T_2(j)$ 满足连接条件. 利用 JPIPT，一个直观的输出结果的想法是根据 PI_1 和 PI_2 域的值，抽取表 T_1 和 T_2 的指定位置索引的元组，分别存储到文件 R_1 和 R_2. R_1 和 R_2 中具有相同的新位置索引(R_i 的第一个元组具有新位置索引 1)的元组对构成一个连接元组. 这个想法比较简单，可是其效率却较低. JPIPT 的 PI_1 域是排序的，对 T_1 的一次顺序扫描就可以获得需要的 T_1 元组. 可是，JPIPT 的 PI_2 域却不是排序的，根据 PI_2 的域指定的位置索引来抽取 T_2 元组会引起大量的磁盘 seek 操作，执行效率较低.

在本节讨论时用到的参数定义如表 1 所示. 在结果输出阶段中，有 3 个步骤顺序执行：JPIPT 划分、外关系扫描和内关系扫描. JPIPT 划分过程把 JPIPT 划分为多个分片来保证外关系扫描和内关系扫描的内存需求. 外关系扫描利用 PI_1 域来抽取 T_1 元组，内关系扫描则利用 PI_2 域来抽取 T_2 元组，当然只返回 attribute-list 涉及到的属性. 接下来，我们依次介绍 3 个步骤的执行过程.

表 1 参数符号

符号	意义
M	内存容量
C	高速缓存容量
S_{JPIPT}	JPIPT 占用的磁盘空间大小
S_{JT}	JPIPT 元组大小
N	连接操作结果数量
S_{T_2}	attribute-list 涉及到的 T_2 元组大小
N_1	T_1 表的元组数量
N_2	T_2 表的元组数量

4.2.1 JPIPT 划分 JP(JPIPT Partition)

在 DBCC-Join 中，外关系扫描要求每个 JPIPT 分片可以放入内存，内关系扫描则要求每个 JPIPT 分片和对应的 T_2 元组都可以放入内存. JP 独立地把 JPIPT 划分成两个不同的分片集合：PS_1 和 PS_2. 其中 PS_1 包括 $m=\left\lceil\frac{S_{\text{JPIPT}}}{M}\right\rceil$ 个分片，PS_2 包括 $n=\left\lceil\frac{S_{\text{JPIPT}}+N\times S_{T_2}}{M}\right\rceil$ 个分片. PS_1 的第 k 个元素 $PS_1(k)=\left\{(i,j):(i,j)\in\text{JPIPT},i\in[PI_{1_{k-1}},PI_{1_k}],PI_{1_k}=k\times\frac{N_1}{m},1\leq k\leq m\right\}$. $[PI_{1_{k-1}},PI_{1_k}]$ 是 T_1 的位置索引的一个连续子范围(为方便讨论，假设 T_1 和 T_2 的连接属性在其值域内均匀分布). $[PI_{1_{k-1}},PI_{1_k}]$ 满足 $\forall i,j(i\neq j),[PI_{1_{i-1}},PI_{1_i}]\cap[PI_{1_{j-1}},PI_{1_j}]=\varnothing$，$\bigcup_{i=1}^{m}[PI_{1_{i-1}},PI_{1_i}]=[1,N_1]$. 类似的，$PS_2$ 的 k^{th} 元素 $PS_2(k)=\left\{(i,j):(i,j)\in\text{JPIPT},i\in[PI_{2_{k-1}},PI_{2_k}],PI_{2_k}=k\times\frac{N_2}{n},1\leq k\leq n)\right\}$，$[PI_{2_{k-1}},PI_{2_k}]$ 是 T_2 位置索引的一个连续子范围. $[PI_{2_{k-1}},PI_{2_k}]$ 满足 $\forall i,j(i\neq j),[PI_{2_{i-1}},PI_{2_i}]\cap[PI_{2_{j-1}},PI_{2_j}]=\varnothing$，$\bigcup_{i=1}^{n}[PI_{2_{i-1}},PI_{2_i}]=[1,N_2]$. JPIPT 划分过程的子范围划分如图 4 所示.

JPIPT 构建阶段最后的多路合并排序操作和 JPIPT 划分操作结合为连续的操作，从而节省

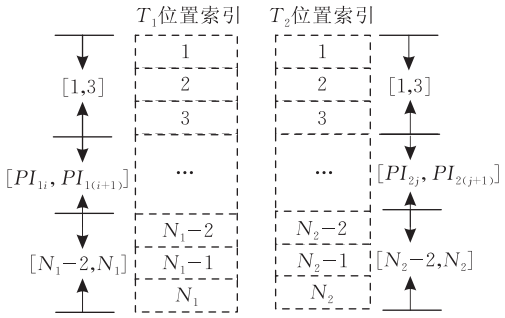


图 4　JPIPT 划分操作的子范围划分

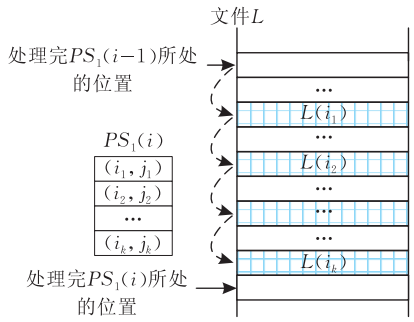


图 5　外关系扫描示例

JPIPT 的一次读和一次写操作. JPIPT 划分只需要对 JPIPT 进行一次扫描就可以把 JPIPT 元组分配到对应的 PS_1 和 PS_2 元素中，其执行过程如算法 2[8] 所示，该过程具有步长为 8 的引用模式.

4.2.2　外关系扫描 ORS(Outer-Relation Scan)

这部分介绍如何利用 PS_1 来抽取 T_1 元组. 在 ORS 过程中，PS_1 从第一个元素 $PS_1(1)$ 开始遍历. 根据定义，$PS_1(1)$ 中 PI_1 的值在范围 $[PI_{1_0}, PI_{1_1}]$，并且根据 PI_1 的值排序. ORS 从涉及的 T_1 表的起始位置开始扫描，抽取 $PS_1(1)$ 中 PI_1 值指定的 T_1 元组. 注意，获得的结果元组不是输出到单个文件，而是根据该元组对应的 JPIPT 元组的 PI_2 值输出到 n 个文件的某一个文件. 给定 JPIPT 元组 (PI_1, PI_2) 指定的 T_1 元组 t，如果 $PI_2 \in [PI_{2_{j-1}}, PI_{2_j}]$，元组 t 输出到文件 $file_1\_1\_j(1 \leq j \leq n)$).

继续遍历 PS_1 的元素，得到 $PS_1(2)$ 的数据. 根据 PS_1 的划分规则，$PS_1(2)$ 中的 PI_1 的值在范围 $[PI_{1_1}, PI_{1_2}]$. $\forall i_0, i_1, j_0, j_1, (i_0, j_0) \in PS_1(1), (i_1, j_1) \in PS_1(2)$，则必然有 $i_0 \leq i_1$. 所以 ORS 可以从当前 T_1 的偏移位置继续扫描，而不需要重新从头开始扫描. 该处理过程类似于对 $PS_1(1)$ 的讨论，根据对应的 JPIPT 元组的 PI_2 值，T_1 元组输出到对应文件 $file_1\_2\_j(1 \leq j \leq n)$).

该过程继续执行直到 PS_1 的元素都处理完毕. 图 5 给出了 ORS 的一个示例. 此时，我们获得一组文件 $FS_1 = \{file_1\_i\_j, 1 \leq i \leq m, 1 \leq j \leq n\}$. ORS 的执行过程如算法 3[8] 所示. ORS 执行采用批量读写机制，当需要读取连续的大块数据时，每次读操作不是读取单个元组，而是一次性读取多个数据块，从而分摊了磁盘寻道和旋转时间. 但是，当一次性读取的数据块增长到一定程度的时候，批量读取带来的增量收益就会变得很小. 根据实验的机器配置，本文的批量读操作一次性读取 4 个数据块. 类似的，每个输出文件维护一个输出缓冲区，当该缓冲区满的时候才真正执行磁盘写操作.

4.2.3　内关系扫描 IRS(Inner-Relation Scan)

这部分介绍如何利用 PS_2 来抽取 T_2 元组. 由于 JPIPT 的 PI_2 域不是排序的，所以 IRS 的执行过程比 ORS 更加复杂. 在 IRS 过程中，PS_2 从第一个元素 $PS_2(1)$ 开始遍历. 根据定义，$PS_2(1)$ 中 PI_2 的值在范围 $[PI_{2_0}, PI_{2_1}]$. 要对 T_2 执行顺序扫描来获得结果元组，$PS_2(1)$ 先根据 PI_2 的值执行一次排序操作. 因为 $PS_2(1)$ 的大小通常都要大大超过高速缓存的容量，并且常用的排序算法具有较大的引用步长，本文提出一种新的缓存敏感的排序算法 C2-Sort 来更快地完成排序操作，其执行代码如算法 4[8] 所示. C2-Sort 的基本思想是：不同于一次性对大数据块执行排序操作，我们先把数据块划分成多个子块，使得每个子块可以放入高速缓存，然后依次对每个子块执行排序操作，最后利用一次多路归并排序来合并排好序的子块. 在每一个执行步骤中，C2-Sort 都可以较好地利用高速缓存从而获得较好的性能.

我们排序 $PS_2(1)$ 的方法不是一次性把它读入内存，而是每次读入 C/S_{JT} 个 JPIPT 元组并且对这些元组执行排序. 排好序的元组在内存中维护. 该过程持续进行直到所有的 $PS_2(1)$ 的数据都读入内存，此时一次合并操作就可以把排好序的子块合并为单个排好序的 $PS_2(1)$.

$PS_2(1)$ 排序完毕后，IRS 从涉及的 T_2 表的起始位置开始扫描，因为此时 PI_2 的值是排序的，所以顺序扫描足够抽取 $PS_2(1)$. PI_2 指定的 T_2 元组. IRS 在内存中缓存获得的元组. 当所有的 $PS_2(1)$ 指定的 T_2 元组都获得后，IRS 利用 C2-Sort 根据 $PS_2(1)$：PI_1 对缓存的 T_2 元组执行第二次排序操作，来保证结果的正确性(正确性证明在 4.2.4 节). 每个排好序的 T_2 元组根据对应的 JPIPT 元组的 PI_1 的值来决定输出到 m 个文件中的某一个. 如果 PI_1 的值落入范围 $[PI_{1_{i-1}}, PI_{1_i}]$，则对应的 T_2 元组输出到文件 $file_2\_i\_1(1 \leq i \leq m)$.

继续遍历 PS_2 的元素，得到 $PS_2(2)$ 的数据. 根据定义，$PS_2(2)$ 中的 PI_2 的值在范围 $[PI_{2_1}, PI_{2_2}]$.

$\forall i_0, i_1, j_0, j_1, (i_0, j_0) \in PS_2(1), (i_1, j_1) \in PS_2(2)$，则必然有 $j_0 \leq j_1$. 所以 IRS 可以从当前 T_2 的偏移位置继续扫描，而不需要重新从头开始扫描，该处理过程类似于对 $PS_2(1)$ 的讨论，此时连接结果根据对应的 JPIPT 元组的 PI_1 值输出到文件 $file_2\_i\_2 (1 \leq i \leq m)$. IRS 的执行过程见文献[8]算法 5.

该过程继续执行直到 PS_2 的元素都处理完毕. 我们获得一组文件 $FS_2 = \{file_2\_i\_j, 1 \leq i \leq m, 1 \leq j \leq n\}$.

此时，ROS 阶段执行完毕. 我们获得 $m \times n$ 对文件：$file_1\_i\_j$ 和 $file_2\_i\_j (1 \leq i \leq m, 1 \leq j \leq n)$，在每对文件中，具有相等的新位置索引的元组构成连接操作的结果元组.

4.2.4 正确性证明

这部分证明 ROS 的正确性. 我们首先证明定理 2 的正确性. 在定理 2 中，没有内存限制的 ROS 不包括 JPIPT 划分阶段，即 $m=1$ 和 $n=1$.

定理 2. 没有内存限制的 ROS 生成的结果是正确的.

证明. 假设 BUF_T_1 保存 ORS 操作获得的 T_1 元组，BUF_T_2 保存 IRS 操作获得的没有经过第二次排序的 T_2 元组. JPIPT 的 PI_1 域的数据可以看作序列 $S_1 = \{i_1, i_2, \cdots, i_N\}$，$PI_2$ 域的数据可以看作序列 $S_2 = \{j_1, j_2, \cdots, j_N\}$. 由于 JPIPT 根据 PI_1 排序，所以 $i_1 \leq i_2 \leq \cdots \leq i_N$，$BUF\_T_1$ 保存的元组具有新的位置索引 $P_1 = \{1, 2, \cdots, N\}$，并且 P_1 的元素和 S_1 的元素具有 1-1 对应关系（虽然 S_1 可能存在重复值，可是这些重复值表示相同的 T_1 元组，所以其顺序不会影响结果的正确性）. 我们把 S_1 和 P_1 结合而获得一个新的序列 $SP_1 = \{(i_1, 1), (i_2, 2), \cdots, (i_N, N)\}$. 类似的，我们获得 $SP_2 = \{(j_1, (i_1, 1)), (j_2, (i_2, 2)), \cdots, (j_N, (i_N, N))\}$，其中 $(j_k, (i_k, k))$ 表示 $T_2(j_k)$ 是 $JPIPT(k)$ 对应的连接结果的一部分. 因为 SP_2 不是根据 j_k 排序的，为方便对 T_2 的顺序扫描来获得结果，IRS 对 SP_2 根据 j_k 排序，获得 $SP'_2 = \{(j'_1, (i_{s_1}, s_1)), (j'_2, (i_{s_2}, s_2)), \cdots, (j'_N, (i_{s_N}, s_N))\}$，$j'_1 \leq j'_2 \leq \cdots \leq j'_N$，$\{s_1, s_2, \cdots, s_N\}$ 是 P_1 的一个排列. 很明显，BUF_T_1 和 BUF_T_2 中具有相同的新位置索引的元组不是连接操作的结果元组，因为通常 $s_k \neq i_{s_k}$. 要保证结果的正确性，BUF_T_2 需要执行另一次根据 s_k 排序的操作. 由于 S_1 和 P_1 的元素有 1-1 对应关系，所以 BUF_T_2 也可以根据 i_{s_k} 排序，即根据对应的 PI_1 排序. 该证明过程如图 6 所示. 证毕.

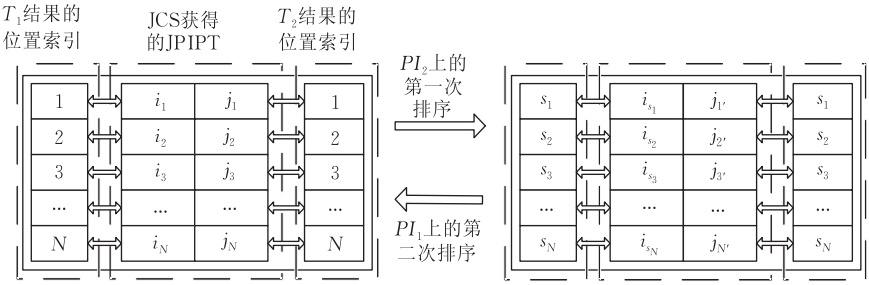


图 6 定理 2 证明实例

由于 PI_1 和 PI_2 根据各自范围进行划分，PS_1 和 PS_2 的元素不存在直接的关系. 我们用矩阵 $\boldsymbol{S}$ 来构建 PS_1 和 PS_2 的元素之间的联系，$\boldsymbol{S}$ 的第 i 行对应于 $PS_1(i)$，$\boldsymbol{S}$ 的第 j 列对应于 $PS_2(j)$，$\boldsymbol{S}(r,c) = \{(i,j): (i,j) \in \text{JPIPT}, i \in [PI_{1_{r-1}}, PI_{1_r}], j \in [PI_{2_{c-1}}, PI_{2_c}], 1 \leq r \leq m, 1 \leq c \leq n\}$. 注意，$\boldsymbol{S}$ 并没有像 PS_1 和 PS_2 那样被实例化，只是虚拟地维护指定数据的范围.

定理 3. ROS 生成的结果是正确的.

证明. ROS 阶段产生 $m \times n$ 对文件：$file_1\_i\_j$ 和 $file_2\_i\_j (1 \leq i \leq m, 1 \leq j \leq n)$. $\boldsymbol{S}$ 的元素 $S(i,j)$ 对应文件对 $file_1\_i\_j$ 和 $file_2\_i\_j$，这是因为 $\forall a, b, (a,b) \in S(i,j)$ 满足如下条件：(1) $a \in [PI_{1_{i-1}}, PI_{1_i}]$，(2) $b \in [PI_{2_{j-1}}, PI_{2_j}]$. $S(i,j)$ 是 JPIPT 的子序列，已知 JPIPT 根据 PI_1 排序，则 $S(i,j)$ 也根据 PI_1 排序. 定理 2 证明 $file_1\_i\_j$ 和 $file_2\_i\_j$ 的正确性，所以 $\forall i, j$，$file_1\_i\_j$ 和 $file_2\_i\_j$ 都产生正确的结果. 证毕.

5 性能评价和分析

我们通过实验来评价 DBCC-Join 的性能. 由于现有的利用高速缓存的连接算法[4,6,9-11]只适用于内存数据库，而本文的重点是海量数据上的磁盘连接算法，所以本文的实验部分比较 DBCC-Join 和传统的 hash join 算法. 我们用 Java 实现 hash join 和 DBCC-Join，jdk 版本是 jdk-6u17-windows-x64，实验程序在 HP xw8600 workstation（8×2.8GHz Xeon CPU+6MB L2 cache+32GB 内存+1.4TB 硬盘+64bit windows）上运行. 我们用 tpc-h 生成元组定长的实验数据，采用的数据表是 lineitem 和 orders. 1sf 的 lineitem 和 orders 表分别含有 6M 和 1.5M 条元组，lineitem 的每个元组为 160 字节，orders 的每个元组为 128 字节，1sf 数据表的大小分别是 0.89GB

和 0.18GB. 实验中采用的查询 Q 如下所示：

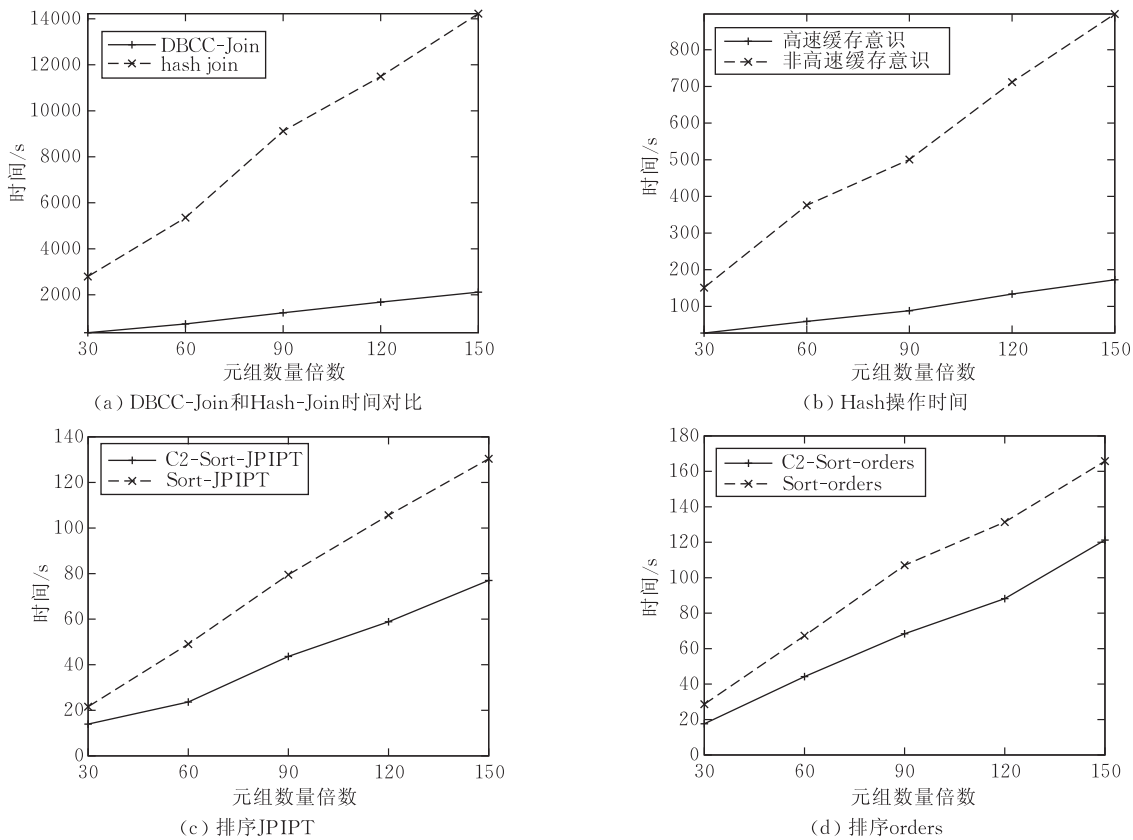
```
select attribute-list
from lineitem, orders
where l_orderkey=o_orderkey
```

实验从 4 个方面评价 DBCC-Join 算法的性能：元组数量、表宽度、输出属性数量和连接选择度. 考虑的元组数量是 30sf、60sf、90sf、120sf 和 150sf. 令 tpc-h 的默认表模式作为表宽度 1(默认值)，表宽度 2 则表示把 tpc-h 的表模式扩大一倍，即重复一遍初始表模式，以此类推得到表宽度为 3、4 和 5 的数据集. 输出属性数量考虑：表模式的 20%、40%、60%(默认值)、80%和 100%的属性. 本文考虑相对连接选择率，连接属性 *l_orderkey* 和 *o_orderkey* 在给定的值域范围(0，*MAX*)内随机分布. 默认情况下，$MAX=2\times N_1$，N_1 表示 lineitem 表的元组数量，假设当 $MAX=1\times N_1$ 时的选择率为 1，则默认选择率为 1/2. 讨论连接选择度的效果时，我们考虑的相对选择度分别为 1、1/2、1/3、1/4 和1/5. 在实验中，内存使用量限制为 8G.

5.1　实验 1：元组数量的效果

实验 1. 评价在元组数量变化的情况下 DBCC-Join 的性能，其中，表宽设置为 1，输出属性比例为 60%，相对选择率为 1/2.

如图 7(a)所示，在实验 1 中，DBCC-Join 的执行时间平均比 hash join 快 7.26 倍. 该加速比的获得一方面是由于 DBCC-Join 只需要对原数据表执行一次扫描就可以完成，而传统的 hash join 则需要对原数据表执行多次扫描操作，另一方面由于是 DBCC-Join 充分利用高速缓存而获得的更好的性能. JPIPT 构建阶段执行 Hash 表的构建和探测操作时，采用多次划分操作使得内关系子分片可以放入高速缓存，然后对每个子分片执行 Hash 操作，Hash 操作的结果在输出之前还需要执行归并排序操作，使得输出的位置索引对根据外关系的位置索引排序，如图 7(b)所示，即使需要如此多的步骤来执行内存内的 Hash 操作，DBCC-Join 的执行时间仍然比不考虑高速缓存的 Hash 操作要快 5.62 倍. 图 7(c)和(d)说明在内关系扫描阶段，采用 C2-Sort 算法比普通的排序算法运行得更快，分别获得了 1.79 和 1.51 倍的加速比.



(a) DBCC-Join和Hash-Join时间对比　(b) Hash操作时间

(c) 排序JPIPT　(d) 排序orders

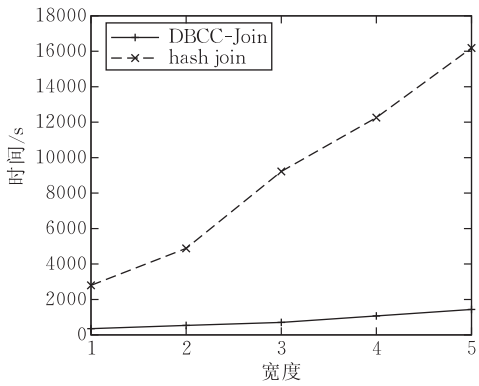
图 7　元组数量变化时执行时间的比较

5.2　实验 2：表宽的效果

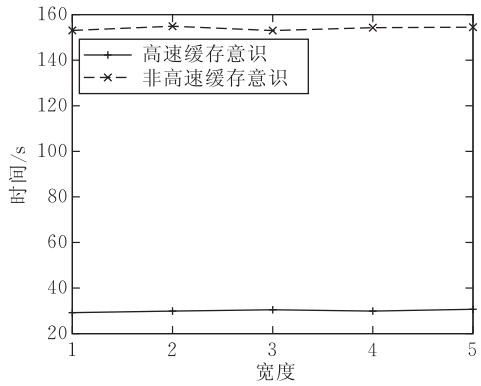
实验 2. 评价在表宽度变化的情况下 DBCC-Join 的性能，其中，sf 设置为 30，输出属性比例设置为 60%，相对选择率为 1/2.

如图 8(a)所示，在实验 2 中，DBCC-Join 的执行时间平均比 hash join 快了 10.58 倍. 通过实验我

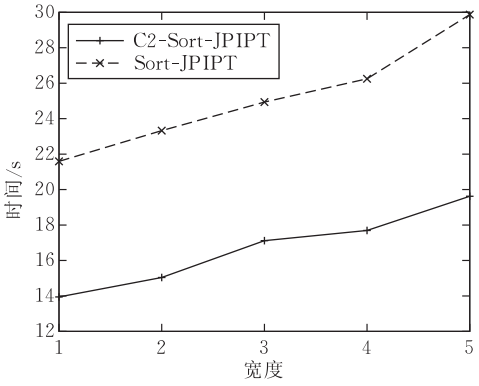
们发现,在其它条件不变的情况下,增加表宽度会提高 DBCC-Join 对 hash join 的加速比,这是由于在固定元组数量的情况下,JPIPT 构建阶段的执行时间不会随着表宽的增大而增加,而且如图 8(b)所示考虑高速缓存的 Hash 操作比不考虑高速缓存的 Hash 操作要快 5.31 倍,而 hash join 的划分和连接操作都会随着表宽的增大而增加. 图 8(c)和(d)表示在右扫描阶段中,采用 C2-Sort 算法比不考虑高速缓存的排序算法运行得更快,分别获得了 1.53 和 1.24 倍的加速比,和实验 1 相比,C2-Sort(orders)获得加速要小一些,这是由于随着表宽的增大,更多的属性需要执行排序操作,这使得合并操作的费用增加,从而减小了所获得的加速比.



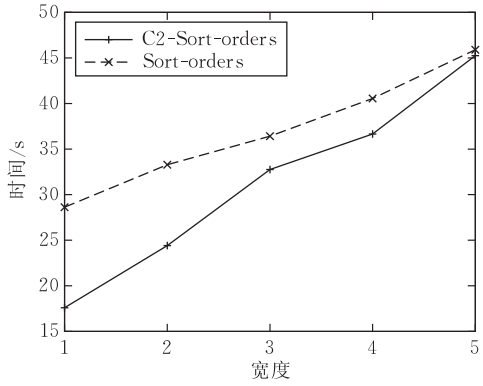
(a) DBCC-Join和hash join时间对比



(b) Hash操作时间



(c) 排序JPIPT



(d) 排序orders

图 8 表宽变化时执行时间的比较

5.3 实验 3:输出属性数量的效果

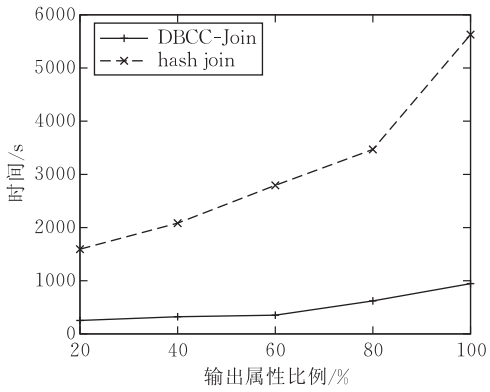
实验 3. 评价在输出属性数量变化的情况下 DBCC-Join 的性能,其中,sf 设为 30,表宽设置为 1,相对选择率为 1/2.

如图 9(a)所示,在实验 3 中,DBCC-Join 的执行时间平均比 hash join 快了 6.44 倍. 我们发现,在输出属性从 80%提高到 100%时,执行时间的增长幅度较大,这是由于 lineitem 和 orders 表的最后一个属性 *l_comment* 和 *o_comment* 比较长,该属性的字节长度分别占整个元组长度的 1/3 和 1/2. 在固定元组数量的情况下,JPIPT 构建阶段 Hash 操作的时间不会随着输出属性数量的变化而变化,如图 9(b)所示. 可以看到,考虑高速缓存的 Hash 操作比不考虑高速缓存的 Hash 操作仍然具有较大的优势. 随着输出属性的增加,如图 9(c)和(d),采用 C2-Sort 算法比不考虑高速缓存的排序算法运行得更快,分别获得了 1.98 倍和 1.63 倍的加速比.

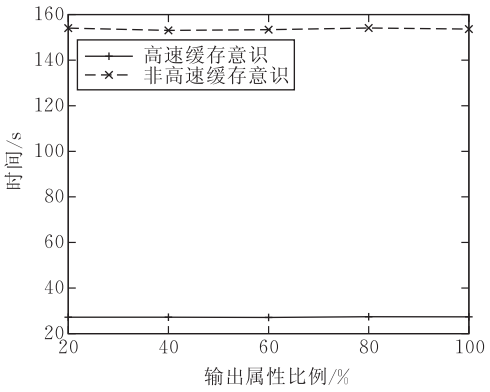
5.4 实验 4:连接选择度的效果

实验 4. 评价在输出属性数量变化的情况下 DBCC-Join 的性能,其中,sf 设为 30,表宽设置为 1,输出属性比例设置为 60%.

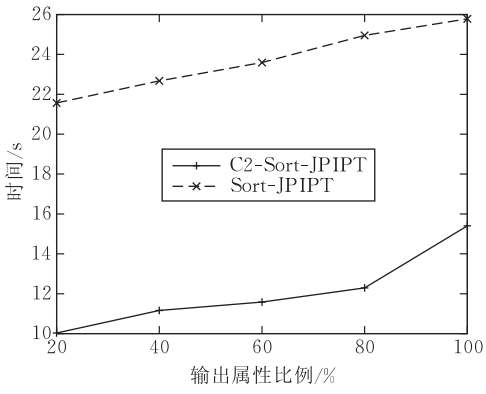
如图 10(a)所示,在实验 4 中,DBCC-Join 的执行时间平均比 hash join 快了 6.35 倍. 可以看到,随着相对选择度的降低,hash join 和 DBCC-Join 的执行时间也在下降,但是 DBCC-Join 的下降幅度更大. 这是由于,更小的选择度使得结果输出阶段所需要处理的数据越少,所以内关系扫描和外关系扫描所耗费的时间基本呈线性下降. DBCC-Join 的 70% 的时间花费在结果输出阶段,所以该阶段所耗费的时间随着选择度降低而线性下降,使得 DBCC-Join 的整体时间下降幅度较大. 而 hash join 的划分阶段不受选择度变化的影响,在连接阶段的Hash表构



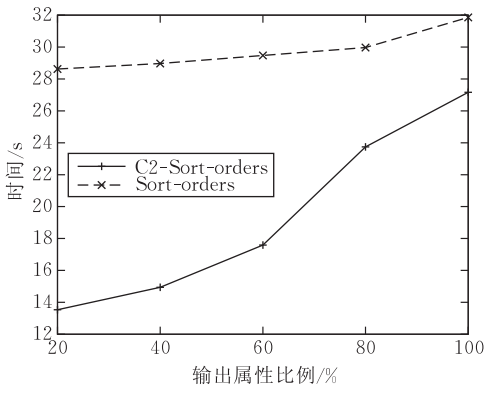
(a) DBCC-Join和hash join时间对比



(b) Hash操作时间

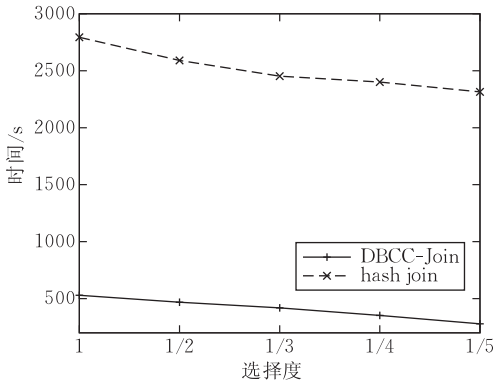


(c) 排序JPIPT

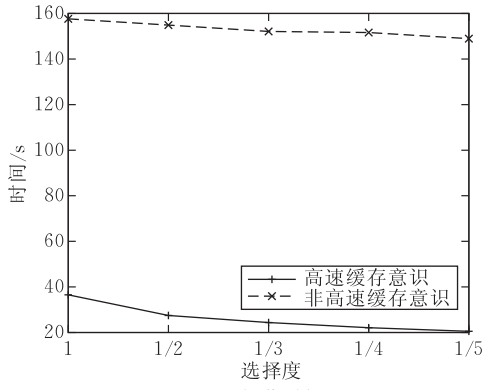


(d) 排序orders

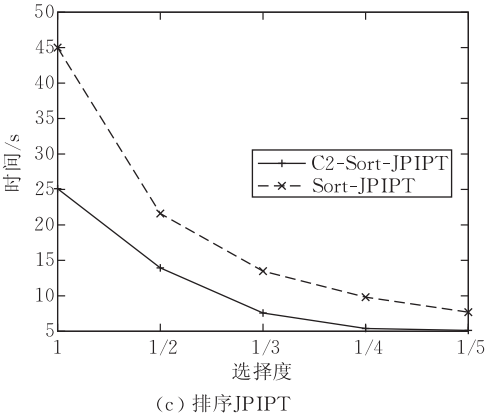
图 9　输出属性数量变化时执行时间的比较



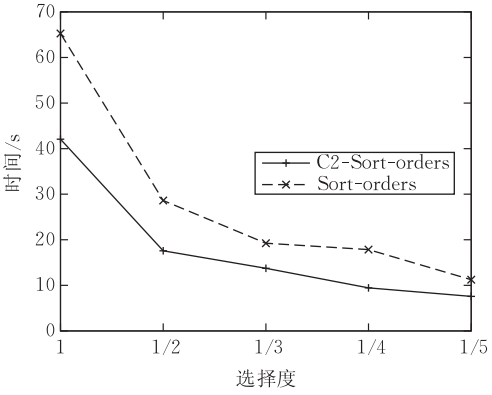
(a) DBCC-Join和hash join时间对比



(b) Hash操作时间



(c) 排序JPIPT



(d) 排序orders

图 10　连接选择度变化时执行时间的比较

建的费用也不变,选择度降低只是会使得 Hash 表的探测阶段的时间降低,所以选择度对 hash join 的影响没有对 DBCC-Join 那么大,所以 DBCC-Join 比 hash join 的性能优势逐渐变大.如图 10(b)所示,在执行 Hash 构建操作时,随着选择度的降低,其执行时间也在逐渐下降,但是和不考虑高速缓存的算法相比,高速缓存敏感的 Hash 算法仍然获得了 6.06 倍的加速比.同样,如图 10(c)和(d)所示,选择度降低使得内关系扫描需要排序的数据也减少,从而减少了执行时间,但是相对于普通的排序算法,C2-Sort 仍然获得了 1.69 和 1.59 倍的加速比.

6 相关工作

连接操作是数据库中的一种基本并且耗时的操作,所以得到了研究人员的广泛关注.Nested-loop 算法、Sort-merge 算法、Grace Hash join 和 Hybrid Hash join 是 4 种常用的连接算法[3,12].Bucket skip merge join[13]的方法假定关系表已经根据连接属性排序,并且虚拟地把数据表划分成多个分片,利用每个分片的连接属性的最大值和最小值信息来跳过不满足连接条件的元组及分片,从而提高了连接算法的性能.Graefe[14]利用 5 种方法来改进 Hybrid hash join 的性能:压缩、大规模的机群及多层递归、内关系和外关系的角色转换、利用数据分布的柱状图以及多表连接的算法等.Chen 等[15-16]利用预取机制来充分利用计算机的硬件特点,提高了 hash join 的性能.

本文利用位置索引对信息来减少所需要的 IO 代价,加快 join 查询的处理速度,其概念和 Valduriez[17]提到的连接索引类似.Lei 等[18]提出 stripe-join 方法处理 join 操作,可是该算法假设已经得到所需要的连接索引.

大多数现有的数据库查询算法并没有充分利用计算机硬件,从而没有充分发挥 CPU 的能力[4-5].一部分研究人员提出利用缓存的 join 算法.Shatdal 等[2]提出缓存划分算法,使得在划分阶段把内关系的每个分片划分得尽可能小,使得每个分片可以放入高速缓存,但是该方法引起昂贵的磁盘 IO 费用.Boncz 等[4]指出了数据库操作的新瓶颈是内存存取,而不是通常认为的 IO 存储,并且发现在主存数据库的划分操作中,划分分片过多时,会引起大量的 TLB miss,所以他们提出了 radix-clustering 算法[6],通过执行多遍划分的方法,使得每遍划分的分片数量小于 TLB 维护的映射数量,从而减少了 TLB miss 的数量.He 等[9-11]提出 cache oblivious 查询处理方法,该方法意识到多层存储结构的存在,但是不需要知道存储结构的具体参数,而是采用自动调节的方法获得较好的性能.和本文的方法不同,Boncz 和 He 等处理的是主存数据库的 join 操作,而不是磁盘 join 操作.Kim 等[19]根据 Radix-clustering 算法,并且结合了计算机 SIMD 和多核的特点实现了 join 算法,来比较同样充分利用计算机硬件特性的 sort-merge join 的性能.他们得出如下结论:当 SIMD 的宽度达到 512-bit 的时候,sort-merge join 就优于 hash join.Nyberg 等[20]指出要获得高性能的排序算法,必须考虑如何利用高速缓存,他们采用的方法也是利用划分操作把数据表数据分为多个分片,每个分片可以放入高速缓存,从而获得较快的 cache 执行效率,可是其划分过程存在的问题和文献[2]中的问题类似.

7 结 论

为弥补 CPU 和内存之间日益增大的性能差异,人们在 CPU 和内存之间插入了高速缓存.要提高 CPU 的利用率,应用程序必须充分利用高速缓存.本文分析了传统的磁盘连接算法在高速缓存利用方面的问题,并且提出了一种新的可以充分利用高速缓存的磁盘连接算法 DBCC-Join.连接位置索引对表 JPIPT 是用到的数据结构,说明了每个连接结果元组在各自表中的位置索引对.DBCC-Join 的执行包括两个阶段:JPIPT 构建阶段和结果输出阶段.JPIPT 构建阶段对列存储化的连接属性执行高速缓存敏感的算法来构建连接位置索引对表.利用获得的 JPIPT,结果输出阶段只需要对数据表执行一遍顺序扫描就可以获得结果.实验表明,和传统磁盘连接算法相比,DBCC-Join 算法可以获得一个数量级的加速比,从而强调了高速缓存在数据库查询处理方面的重要性.

参 考 文 献

[1] Todd Jobson's Blog Reflections. Santa Clara, CA, USA: Sun Microsystems, 2007

[2] Shatdal Ambuj, Kant Chander, Naughton Jeffrey F. Cache conscious algorithms for relational query processing//Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94). Santiago de Chile, Chile, Morgan Kaufmann, 1994: 510-521

[3] Mishra Priti, Eich Margaret H. Join processing in relational databases. ACM Computing Surveys, 1992, 24(1): 63-113

[4] Boncz Peter A, Manegold Stefan, Kersten Martin L. Database architecture optimized for the new bottleneck: Memory access//Proceedings of the 25th International Conference on Very Large Data Bases (VLDB'99). Edinburgh, Scotland, UK, Morgan Kaufmann, 1999: 54-65

[5] Ailamaki Anastassia, DeWitt David J, Hill Mark D, Wood David A. DBMSs on a modern processor: Where does time go? //Proceedings of the 25th International Conference on Very Large Data Bases (VLDB'99). Edinburgh, Scotland, UK, Morgan Kaufmann, 1999: 266-277

[6] Manegold Stefan, Boncz Peter A, Kersten Martin L. What happens during a join? Dissecting CPU and memory optimization effect//Proceedings of the 26th International Conference on Very Large Data Bases (VLDB'00). Cairo, Egypt, Morgan Kaufmann, 2000: 339-350

[7] Stonebraker Mike, Abadi Daniel J, Batkin Adam et al. C-Store: A colmn-oriented DBMS//Proceedings of the 31st International Conference on Very Large Data Bases (VLDB'05). Trondheim, Norway, ACM, 2005: 553-564

[8] Han Xixian, Yang Donghua, Li Jianzhong. DBCC-Join: A novel cache-conscious disk-based join algorithm. Harbin Institute of Technology, Harbin: Technical Report DBTR-1002, 2010

[9] He Bingsheng, Luo Qiong. Cache-oblivious nested-loop joins//Proceedings of the 2006 ACM CIKM International Conference on Information and Knowledge Management (CIKM'06). Arlington, Virginia, USA, ACM, 2006: 718-727

[10] He Bingsheng, Luo Qiong. Cache-oblivious query processing//Proceedings of th 3rd Biennial Conference on Innovative Data Systems Research (CIDR'07). Asilomar, CA, USA, 2007: 44-55

[11] He Bingsheng, Luo Qiong. Cache-oblivious databases: Limitations and opportunities. ACM Transactions on Database Systems (TODS), 2008, 33(2): 8

[12] Shapiro Leonard D. Join processing in database systems with large main memories. ACM Transactions on Database Systems (TODS), 1986, 11(3): 239-264

[13] Kamath Mohan, Ramamritham Krithi. Bucket skip merge join: A scalable algorithm for join processing in very large database using indexes. University of Massachusetts: Technical Report CS-TR-96-20, 1996

[14] Graefe Goetz. Five performance enhancements for hybrid hash join. University of Colorado at Boulder: Technical Report CU-CS-606-92, 1992

[15] Chen Shimin, Ailamaki Anastassia, Gibbons Phillip B, Mowry Todd C. Improving hash join performance through prefetching//Proceedings of the 20th International Conference on Data Engineering (ICDE'04). Boston, MA, USA, 2004: 116-127

[16] Chen Shimin, Ailamaki Anastassia, Gibbons Phillip B, Mowry Todd C. Improving hash join performance through prefetching. ACM Transactions on Database Systems (TODS), 2007, 32(3): 17

[17] Valduriez Patrick. Join indices. ACM Transactions on Database Systems (TODS), 1987, 12(2): 218-246

[18] Lei Hui, Ross Kenneth A. Faster joins using join indices. VLDB Journal, 1998, 8(1): 1-24

[19] Kim Changkyu, Sedlar Eric, Chhugani Jatin, Kaldewey Tim, Nguyen Anthony, Blas Andrea Di, Lee Victor, Satish Nadathur, Pradeep Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs//Proceedings of the 35st International Conference on Very Large Data Bases (VLDB'09). Lyon, France, ACM, 2009: 1378-1389

[20] Nyberg Chris, Barclay Tom, Cvetanovic Zarka, Gray Jim, Lomet Dave. AlphaSort: A cache-sensitive parallel external sort. VLDB Journal, 1995, 4(4): 603-627

HAN Xi-Xian, born in 1981, Ph. D. candidate. His research interests include massive data processing and data intensive computing.

YANG Dong-Hua, born in 1976, Ph. D., lecturer. His research interests include massive data processing and data intensive computing.

LI Jian-Zhong, born in 1950, professor, Ph. D. supervisor. His research interests include database and sensor network.

Background

This paper focuses on the research for query processing issues on massive data. Join processing is an important but time-consuming operation in database. Its performance directly affects overall performance of database. Cache is used for making up for performance gap between CPU and memory. Its access speed is much faster than memory. Efficient join algorithm should be designed to take maximum advantage of cache memory. But, current join algorithms considering cache are mainly used for main memory database. Some researchers propose cache-oblivious join algorithms which elaborately design algorithms without considering concrete parameters of hierarchical organization of memory system, but they are still used in memory database. None of existing disk-based join algorithm takes cache into consideration, which makes poor cache utilization rate for these join algorithms. This paper is the first to propose cache-conscious disk-based join algorithm DBCC-Join on massive data. Its execution consists of two stages: JPIPT construction stage and result output stage. JPIPT construction stage makes use of cache-conscious construction to generate JPIPT quickly. Result output stage exploits JPIPT obtained to perform one-pass sequential selective scan on each joined table to retrieve join results. Extensive experiments are conducted in this paper in terms of tuple number, table width, output attribute number and selectivity. Experimental results show that DBCC-Join obtains an order of magnitude speed-up comparing to existing join algorithms.

This work is supported in part by the National Grand Fundamental Research 973 Program of China, the Key Program of National Natural Science Foundation of China, the NSF of China and the NSFC-RGC of China. These foundations focus on the research of various areas of data intensive super computing. Our group has been working on the research of database for many years, and many good papers have been published in worldwide conferences and transactions, such as SIGMOD, VLDB, ICDE, KDD, INFOCOM, TKDE, VLDB Journal et al. This paper proposes an efficient disk-based join algorithm by cache-conscious operation. It gives a solution on topic of query processing on massive data.