基于三级存储器的 Join 算法*

李建中[1,2], 张冬冬[1+], 张艳秋[1]

[1](哈尔滨工业大学 计算机科学与技术学院,黑龙江 哈尔滨 150001)

[2](黑龙江大学 计算机科学与技术学院,黑龙江 哈尔滨 150080)

Join Algorithms Based on Tertiary Storage

LI Jian-Zhong[1,2], ZHANG Dong-Dong[1+], ZHANG Yan-Qiu[1]

[1](College of Computer Science and Technology, Harbin Institute of Technology, Harbin 150001, China)

[2](College of Computer Science and Technology, Heilongjiang University, Harbin 150080, China)

+ Corresponding author: Phn: 86-451-6415827, Fax: 86-451-6415827, E-mail: z_dd@sina.com.cn

http://db.hit.edu.cn

Received 2002-03-04; Accepted 2002-12-24

Li JZ, Zhang DD, Zhang YQ. Join algorithms based on tertiary storage. *Journal of Software*, 2003,14(5): 947~954.

http://www.jos.org.cn/1000-9825/14/947.htm

Abstract: The Join algorithms of massive relations in relational databases based on tertiary storage are studied in this paper. At present, Hash-Based Join algorithms are the best ones. However, the effect of tape locate time is not taken into consideration in these algorithms. It has great influence on the time complexity of the Join algorithms to locate positions on tertiary storages. For this reason, two new Join algorithms of massive relations in relational databases are proposed based on tertiary storage, Disk-Based-Hash-Join algorithm and Tertiary-Only-Hash-Join algorithm. Adopting disk buffer technique and the method of storing hashed data concentratedly, the cost of the random position locating on tertiary storage is much lower than other algorithms so that the proposed Join algorithms are more efficient. The analysis and experimental results show that the performance of this algorithms is superior to others, and thus they are suitable for massive database management.

Key words: massive data; tertiary storage; join algorithm

摘 要: 研究了基于三级存储器的海量关系数据库的 Join 算法.目前,在所有磁带数据 Join 算法中,基于 Hash 思想的算法是最优的.但是,这些算法没有考虑从第三级存储器中读取数据时,磁带定位时间对算法性能的影响.磁带的磁头随机定位耗时大,是影响基于三级存储器的数据操作算法时间复杂性的关键因素.针对这个问题,提出了两种新的基于三级存储器的海量关系数据库连接算法,即 Disk-Based-Hash-Join 算法和

* Supported by the National Natural Science Foundation of China under Grant No.60273082 (国家自然科学基金); the National High-Tech Research and Development Plan of China under Grant No.2001-AA-415-410 (国家高技术研究发展计划); the National Grand Fundamental Research 973 Program of China under Grant No.G1999032704 (国家重点基础研究发展规划(973)); the National Research Foundation for the Doctoral Program of Higher Education of China under Grant No.2000021303 (国家教育部博士点基金)

第一作者简介: 李建中(1950－),男,黑龙江哈尔滨人,教授,博士生导师,主要研究领域为数据库,并行计算.

Tertiary-Only-Hash-Join 算法.这两种算法采用了磁盘缓冲技术和散列数据集中存储方法,降低了算法的磁带磁头随机定位时间复杂性,提高了基于三级存储器的连接算法的性能.理论分析和实验结果表明,提出的基于三级存储器连接算法的性能高于目前所有同类算法的性能,可以有效地应用于海量数据管理系统.

关键词: 海量数据;三级存储器;连接算法

中图法分类号: TP311 **文献标识码**: A

最近几年,随着信息技术的发展,特别是 Internet 技术的发展,各行各业的信息量都呈爆炸性增长趋势,高于 10^{12} 字节的海量数据库已成为常见的数据库.目前 Internet 上已经拥有的数据量已达兆亿字节量级,并且还在不断扩大.我国海量数据库的数据量也在迅速增长.例如,黑龙江移动通信局 3 个月的信息量高达 1 000 亿字节;国家信息安全数据库 3 个月的数据量高达 2 000 亿字节;粒子碰撞实验每年产生的数据容量高达 300TB;描述一架飞机全机构件的信息量就高于 150G.显然,传统的基于二级存储器(主存储器和磁盘存储器)的数据库管理系统难以承担如此庞大的海量数据的存储与管理任务.海量数据需要使用基于三级存储器(主存储器、磁盘存储器和机器手磁带库等第三级存储器)的数据库管理系统来存储和管理.目前绝大多数数据库管理系统都是基于二级存储器的,难以管理海量数据.有些数据库系统把第三级存储器作为后援存储器来使用.这样的系统在处理第三级存储器上的数据时,先将第三级存储器上的数据复制到第二级存储器,然后使用基于二级存储器的数据库技术处理查询.当被查询的数据集合的大小大于第二级存储器的容量时,这种系统将无能为力.不言而喻,我们需要研究基于三级存储器的数据库技术.Join 操作是一种最常用也是最耗时的数据库操作.本文旨在研究海量数据的 Join 算法.

20 世纪 90 年代,数据库工作者曾经开展过一些基于磁带的海量数据 Join 算法的研究,并提出了一些算法.这些算法可以分为两类,一类是基于 Nested-Loop 的思想算法[1,2],另一类是基于 Hash 的思想算法[1,3~5],具有代表性的基于 Nested-Loop 思想的算法包括 Nested-Loop 算法[2]和 Concurrent-Nested-Loop 算法[2].具有代表性的基于 Hash 思想的算法包括 Grace-Hash 算法[1]和 Concurrent-Grace-Hash 算法[3].下面,我们简单介绍这 4 种具有代表性的算法.以下,设$|M|$表示总的可用内存空间字节数,$|M_r|$表示分配给 Join 关系 R 的内存空间字节数,$|M_S|$表示分配给 Join 关系 S 的空间字节数,关系 R 和 S 分别存储在两条不同的磁带上,且内存和硬盘的可用空间都不足以完全容纳任何关系的全部数据.

Nested-Loop(简称 N-L)算法是最直观也是最简单的算法.它把关系 R 和 S 全部数据逐步依次调入内存执行 Join 操作.对于每次读入内存中的关系 R 的$|M_r|$数据量,S 中的全部数据都要读入内存一次,并与之进行 Join.显然,N-L 算法的效率非常低.

为了改进 N-L 算法的性能,文献[2]提出了一种称为 Concurrent-Nested-Loop(简称 C-N-L)的算法.C-N-L 算法试图使 I/O 操作和 CPU 操作并发执行.

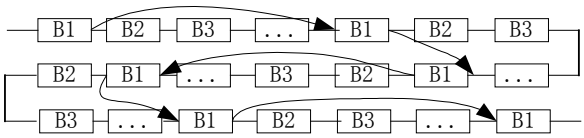

Fig.1 Reading buckets with the same hash value from the tape

图 1 从磁带上读取具有相同散列值的数据桶示例

为了解决基于 Nested-Loop 思想的磁带数据 Join 算法的低效问题,文献[1]提出了基于 Hash 方法的磁带数据 Join 算法,称为 Grace-Hash(简称 G-H)算法.G-H 算法首先将关系 R,S 中的数据分别散列到另一条磁带的数据区末尾形成各自的散列桶,然后将两个关系中具有相同散列值的散列桶进行 Join.与基于 Nested-Loop 思想的算法相比,G-H 算法的性能得到了很大的提高.但是,该算法有两个缺陷,一是需要在磁带之间额外传送两次相当于原关系大小的数据量,二是在从磁带上读取数据散列桶的过程中增加了磁带定位时间.在内存中完成数据散列操作的过程中,由于内存的空间小,内存中的散列桶会产生溢出,为了维持散列操作的正常实现,需要把将要溢出的散列桶提前回写到磁带上.然而,磁头在磁带上必须是顺序写数据的,所以造成了具有相同散列值的数据桶在磁带上不连续分布.紧接着下一步在完成散列桶之间 Join 的过程中,需要把具有相同散列值的数据桶都从磁带上读取到内存中,因此需要磁头不断地跳跃式定位到合适的磁带位

置上去读取这些数据桶(如图 1 所示),从而产生了系统额外的时间开销,增加了算法的整体执行时间.

为了进一步提高 G-H 算法的性能,文献[3]提出了一个称为 Concurrent-Grace-Hash(简称 C-G-H)的算法.C-G-H 算法是对 G-H 算法的改进,它试图在完成散列桶之间的 Join 过程中使 I/O 操作和 CPU 操作并发执行.C-G-H 连接算法首先将关系 R,S 中的数据分别散列到另一条磁带的数据区末尾形成各自的散列桶;然后把可用主存空间(大小为$|M|$)划分为 3 部分,第 1 部分(大小为$|M_r|$)用来存储关系 R 的散列桶,第 2 部分和第 3 部分作为关系 S 的双主存缓冲区(每个大小为$|M_s|/2$),记作 *Buff*[0]和 *Buff*[1],存放关系 S 的散列桶.在算法的执行过程中,当 M_r 中的散列桶与 *Buff*[0](或 *Buff*[1])中的散列桶进行 Join 的同时,从磁带上读取 S 的其余散列桶到 *Buff*[1](或 *Buff*[0]),准备与 M_r 中的散列桶进行 Join.C-G-H 算法在磁带定位时间开销方面存在着与 G-H 算法相同的缺陷.

虽然基于 Hash 思想的已有算法是目前最优的磁带数据 Join 算法,但是,这些算法没有考虑从第三级存储器中读取数据时,磁带定位时间对算法性能的影响.我们通过大量的实验发现,第三级存储器的随机定位时间是影响算法时间复杂性的关键因素.为此,本文深入研究了在三级存储器环境下有效地完成 Join 操作的问题,提出了两种基于三级存储器的 Join 算法,即 Disk-Based-Hash-Join 算法(简称 D-B-H-J 算法)和 Tertiary-Only-Hash-Join 算法(简称 T-O-H-J 算法).当磁盘可利用空间大的时候,D-B-H-J 算法可以取得很好的性能;反之,当磁盘可利用空间较小的时候,T-O-H-J 算法的性能好一些.与传统算法相比,本文提出的算法显著地降低了磁带定位时间.理论分析和模拟实验结果表明,本文提出的 Join 算法的性能高于已有的基于磁带的 Join 算法.

1 磁带模型

为了分析简单且易读,不失一般性,本文在进行算法的复杂性分析时,把锯齿型磁带作为三级存储器的第三级存储器.本文的算法和结论完全适合于其他类型的磁带.本节首先讨论锯齿型磁带的模型.

锯齿型磁带是顺序存取设备,其模型可以由一组参数来表示.表 1 给出了人们常用的锯齿型磁带的模型参数,这些参数来自于文献[6,7].

为了对算法的复杂性进行分析,我们还需要其他一些参数.表 2 给出了在分析算法性能过程中经常用到的参数的表示方法[4].

Table 1 Parameters of the tape model

表 1 磁带模型参数

Parameter	Value	Description	Parameter	Value	Description
LibraryType	Exabyte 220	Type of library	*TapeDriver*	Eliant 820	Type of driver
Modelname	8mm	Type of tape	$T_{apestartup}$	21s	Tape startup latency
M_{ount}	40s	Tape's mounting time	*Numofdriver*	2	Number of drivers
U_{nmount}	21s	Tape's unmounting time	X_T	1.5MB/s	Mean transfer rate of the tape
$R_{OBOTMOVE}$	10s	Robot's moving time	X_D	11MB/s	Mean transfer rate of the disk
Tapelocate	5MB/s	Tape's locating speed	*sizeoftape*	18.98GB	Available size of tape

Table 2 Other parameters used in the algorithms

表 2 算法分析中用到的其他参数

Parameter	Description	Parameter	Description
$\|R\|$	Size of relation R on tape	$\|S\|$	Size of relation S on tape
$\|M_r\|$	Size of available buffer for relation R	$\|M_s\|$	Size of available buffer for relation S
T_{probe}	Mean time to probe a tuple in relations	N_{bucket}	Numbers of buckets with different hash value
S_{tuples_r}	Size of a tuple in relation R	S_{tuples_s}	Size of a tuple in relation S
$\|B_{ucket}\|$	Size of a bucket	$\|D\|$	Size of available disk space
B_{tong_r}	Numbers of buckets with the same hash value in relation R and it is equal to $\|R\| \div \|B_{ucket}\| \div N_{bucket}$	B_{tong_s}	Numbers of buckets with the same hash value in relation S and it is equal to $\|S\| \div \|B_{ucket}\| \div N_{bucket}$

2 Disk-Based-Hash-Join 算法

2.1 算法描述

D-B-H-J 算法利用硬盘缓冲空间完成关系的 Join,并且使 I/O 操作和 CPU 操作并发执行.与 C-G-H 算法思想类似,D-B-H-J 算法把 M_r 这一部分主存空间分成两部分(每个大小为$|M_r|/2$),记作 *Buff*[0]和 *Buff*[1],用来存放关系 R 的散列桶.D-B-H-J 算法首先将关系 R 按照$|D|$大小进行分块,将第 1 块散列到磁盘.然后将关系 S 的数据分块依次散列到内存 M_s 中,分别与磁盘上关系 R 的散列桶进行 Join.接着把关系 R 的下一个数据块散列到硬盘,依次循环下去,直到关系 R 中的数据全部扫描到磁盘一遍.在算法实现过程中,当 M_r 中的关系 S 的散列桶与 *Buff*[0](或 *Buff*[1])中的关系 R 的散列桶进行 Join 的同时,从磁盘上读取 R 的其余散列桶到 *Buff*[1](或 *Buff*[0])中,准备与 M_r 中的关系 S 散列桶进行 Join.算法结束后,磁带上关系 R 的数据需要扫描一遍,而磁带上关系 S 的数据则需要扫描$|R|/|D|$遍.D-B-H-J 算法的形式描述如下:

(1) FOR i=1 To $|D|/|X_D|$ DO
(2) 读取磁带 R 的大小为$|D|$的部分数据 R',使用 Hash 函数 F 散列 R',在磁盘上建立 R'的 Hash 桶;
(3) FOR j=1 To $|S|/|M_S|$ DO
(4) 读磁带 S 的大小为$|M_S|$的部分数据 S',用 F 散列 S',在 M_s 中建立 S'的 Hash 桶;
(5) 读取磁盘中关系 R 的第 1 部分 Hash 桶到内存 *Buff*[0]中;
(6) WHILE 磁盘中关系 R 的 Hash 桶没有取尽 DO /* (7)和(8)并发执行 */
(7) 依次读取磁盘中关系 R 的其他部分 Hash 桶到内存 *Buff*[1](或 *Buff*[0])中;
(8) 对 *Buff*[0](或 *Buff*[1])和 M_S 中的 Hash 桶进行 Join;
(9) ENDWHILE
(10) ENDFOR
(11) ENDFOR

2.2 算法分析

该算法需要扫描一遍磁带上的关系 R,$|R|/|D|$遍关系 S.算法中循环体((2)~(10))执行一次所需的时间为

$$T_{dbhjtrans}=|D|/X_T+|D|/X_D+(|S|/|M_S|)\cdot\{|M_S|/X_T+\mathrm{Max}[|D|/X_D,\ N_{bucket}\cdot|M_S|/(N_{bucket}\cdot S_{tuple\_s})\cdot|D|/(N_{bucket}\cdot S_{tuple\_r})\cdot T_{probe}]\},$$

其中$|D|/X_T$表示关系 R 中$|D|$字节数据从磁带读入内存中的时间开销,$|D|/X_D$表示关系 R 的$|D|$字节数据在内存中散列后移动到硬盘上所需的时间或者从硬盘读取到内存的时间,$|S|/|M_S|$表示关系 S 被扫描一遍的过程中磁盘上关系 R 的相同数据散列桶从磁盘读入内存的次数,$|M_S|/X_T$ 表示关系 S 的$|M_S|$字节数据读入内存的时间,$\mathrm{Max}[|D|/X_D, N_{bucket}\cdot|M_S|/(N_{bucket}\cdot S_{tuple_s})\cdot|D|/(N_{bucket}\cdot S_{tuple_r})\cdot T_{probe}]$表示算法实现中 I/O 操作与 CPU 操作并发执行时的时间开销.因此该算法的数据传输的总时间代价为

$$C_{DBHJ\_trans}=(|R|/|D|)\cdot T_{dbhjtrans}=|R|/X_T+|R|/X_D+|R|\cdot|S|/(|D|\cdot X_T)+|(S|/|M_S|)\cdot(|R|/X_D). \quad (1)$$

关系 S 需要被扫描$|R|/|D|$遍,因此磁头需要有$|R|/|D|$次从磁带数据末尾定位到磁带数据首部,该算法磁头定位总时间开销为

$$C_{DBHJ\_locate}=(|R|/|D|)\cdot(|S|/Tapelocate). \quad (2)$$

从算法的描述可以容易地看到,D-B-H-J 算法的 CPU Join 操作时间复杂性为

$$C_{DBHJ\_join}=(|R|/|D|)\cdot(|S|/|M_S|)\cdot\{N_{bucket}\cdot[|D|\cdot(N_{bucket}\cdot S_{tuple\_r})]\cdot[|M_S|/(N_{bucket}\cdot S_{tuples\_s})]\}$$
$$=|R|\cdot|S|\cdot T_{probe}/(N_{bucket}\cdot S_{tuple\_r}\cdot S_{tuple\_s}). \quad (3)$$

3 Tertiary-Only-Hash-Join 算法

3.1 算法描述

T-O-H-J 算法把 G-H 算法中关系 S 的数据散列桶重新调整顺序,使具有相同散列值的数据散列桶连续地排放在磁带上(如图 2 所示).然后在关系 S 调序后的散列桶和关系 R 的对应散列桶之间进行 Join.T-O-H-J 算法使

I/O 操作和 CPU 操作并发执行.与 C-G-H 算法思想类似,T-O-H-J 算法用主存 M_r 来存放关系 R 的散列桶,同时把主存 M_S 分成两部分(每个大小为 $|M_S|/2$),记作 *Buff*[0]和 *Buff*[1],用来存放关系 S 的散列桶.当 M_r 中的散列桶与 *Buff*[0](或 *Buff*[1])中的散列桶进行 Join 的同时,从磁带上读取 S 的其余散列桶到 *Buff*[1](或 *Buff*[0])中,准备与 M_r 中的散列桶进行 Join.算法的形式描述如下:

(1) 建立关系 R 的 Hash 桶 *RH*,写入磁带 S 末尾;
(2) 建立关系 S 的 Hash 桶 *SH*,写入磁带 R 末尾;
(3) 取出磁带 *S*,装载新磁带 *K*;
(4) 将磁带 R 上的散列桶 *SH* 调整顺序,使具有相同散列值的 Hash 桶连续存放,复制到磁带 *K* 上;
(5) 取出磁带 *R*,装载磁带 *S*;
(6) FOR $i=1$ TO Hash 桶的种类数
(7) WHILE *RH* 中第 i 类 Hash 桶没有取尽 DO
(8) 读取 *RH* 中的第 i 类 Hash 桶中的部分 Hash 桶到内存 M_r 中;
(9) 读取 *SH* 中相应的第 i 类桶中的第 1 部分 Hash 桶到内存 *Buff*[0](或 *Buff*[1])中;
(10) WHILE *SH* 中的第 i 类 Hash 桶没有取尽 DO /* (11)和(12)并发执行 */
(11) 读取 *SH* 中相应的第 i 类 Hash 桶中的其他部分 Hash 桶到内存 *Buff*[1](或 *Buff*[0])中;
(12) 对 M_r 和 *Buff*[0](或 *Buff*[1])中的 Hash 桶进行 Join;
(13) ENDWHILE
(14) ENDWHILE
(15) ENDFOR

在此算法实现过程中,为了重新调整关系 S 的散列桶排放顺序,需要额外的一条磁带,增加了轮换磁带的时间开销.即便如此,该算法的性能仍然优于原有算法.原因是关系 R 的散列桶只需扫描一遍,关系 S 的散列桶需要扫描多遍,而关系 S 的散列桶具有一定的排放顺序,因此极大地减少了 Join 过程中读取关系 S 的散列桶时的磁带定位时间开销.

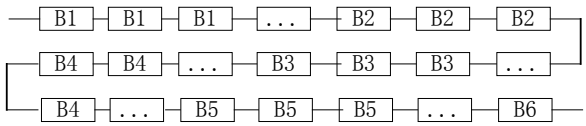

Fig.2 Relation S's buckets on the tape after rearranging
图 2 磁带上关系 S 的数据桶重新排列后的示例

3.2 算法分析

T-O-H-J 算法首先需要对数据进行散列操作,此过程的时间代价为 $T_{halfhash}=2\cdot(|R|+|S|)/X_T$.然后,T-O-H-J 算法需要交换磁带两次,每次的时间代价为 $T_{change}=U_{nmount}+R_{obotmove}+R_{obotmove}+R_{obotmove}+M_{ount}+Tapestartup$.

接着,T-O-H-J 算法需要重新调整散列后关系 S 的 Hash 桶排放顺序.这一过程的时间代价为 $T_{ordertape}=N_{bucket}\cdot[|S|/(X_T\cdot N_{bucket})+|S|/(X_T\cdot N_{bucket})]+T_{randlocate}$,其中 $T_{randlocate}$ 为读取关系 S 未经调整顺序的散列桶时的磁头定位时间代价(包括磁头跳跃式地读取同一类 Hash 桶的定位时间和磁头若干次从磁带数据散列桶尾部定位到磁带数据散列桶首部的时间).假设数据均匀分布,则磁带上具有相同散列值的散列桶按固定磁带间隔长度存放,因此可得 $T_{randlocate}=N_{bucket}\cdot(N_{bucket}-1)\cdot B_{tong\_s}\cdot(|B_{ucket}|/Tapelocate)+N_{bucket}\cdot T_{HL\_s}$.在完成 Hash 桶之间的 Join 过程中,算法实现时所需的时间代价为

$$T_{join}=N_{bucket}\cdot\{[|R|/(N_{bucket}\cdot|M_r|)]\cdot T_{innerjoin}+[|R|/(N_{bucket}\cdot|M_r|)]\cdot T_{partlocate}\}+N_{bucket}\cdot T_{HL\_r},$$

其中,$T_{partlocate}$ 为扫描关系 S 中的某同一类散列桶时磁头从数据散列桶尾部回绕到数据散列桶首部所需要的磁头定位时间,$T_{innerjoin}$ 为执行内层循环((8)~(13))一次所需要的时间.它们的时间代价分别为 $T_{partlocate}=B_{tong\_s}\cdot(|B_{ucket}|/Tapelocate)$ 和 $T_{innerjoin}=|M_r|/X_T+T_{locatetong}+[|S|/(N_{bucket}\cdot|M_S|/2)]\cdot T_{tongjoin}$. 上式中的 $T_{tongjoin}$ 表示第 2 层循环中在内存里具体执行某一次 Join 所需的时间,由于这一过程需要 M_r 中的 Hash 桶与 *Buff*[0](或 *Buff*[1])的 Hash 桶进行 Join,并且算法中实现了 I/O 操作与 CPU 操作的并发执行,所以它的时间代价

为 $T_{tongjoin}=\mathrm{Max}\{(|M_S|/2)/X_T, (|M_r|/S_{tuple\_r})\cdot[(|M_S|/2)/S_{tuple\_s}]\cdot T_{probe}\}$,$T_{locatetong}$表示读取关系 R 的具有相同散列值的部分散列桶到内存 M_r 中所产生的磁头定位时间,它的时间代价为

$$T_{locatetong}=(|M_r|/|B_{ucket}|)\cdot(N_{bucket}-1)\cdot(|B_{ucket}|/Tapelocate).$$

通过以上分析可知,在算法实现过程中,总的磁头定位时间代价为

$$C_{TOHJ\_locate}=T_{randlocate}+(|R|/|M_r|)\cdot T_{locatetong}+(|R|/|M_r|)\cdot T_{partlocate}+N_{bucket}\cdot T_{HL\_r}.$$
$$=(2\cdot N_{bucket}-1)\cdot(|R|/Tapelocate)+[2\cdot N_{bucket}-1+|R|/(N_{bucket}\cdot|M_r|)]\cdot(|S|/Tapelocate). \quad (4)$$

把上述分析中的所有含 $1/X_T$ 项的数据单独累加起来,可以得到磁带数据传输总时间开销为(包括交换磁带的时间)

$$C_{TOHJ\_trans}=2\cdot T_{change}+2\cdot(|R|+|S|)/X_T+2\cdot|S|/X_T+|R|/X_T+N_{bucket}\cdot[|R|/(N_{bucket}\cdot|M_r|)][|S|/(N_{bucket}\cdot|M_S|/2)]\cdot[(|M_S|/2)/X_T]$$
$$=2\cdot T_{change}+3\cdot|R|/X_T+4\cdot|S|X_T+[|R|/(N_{bucket}\cdot|M_r|)](|S|/X_T). \quad (5)$$

从算法的形式描述易知,T-O-H-J 算法的 CPU Join 操作时间复杂性为

$$C_{TOHJ\_join}=N_{bucket}\cdot[|R|/(N_{bucket}\cdot|M_r|)]\cdot[|S|/(N_{bucket}\cdot|M_S|/2)]\cdot(|M_r|/S_{tuple\_r})\cdot[(|M_S|/2)/S_{tuple\_s}]\cdot T_{probe}$$
$$=|R|\cdot|S|\cdot T_{probe}/(N_{bucket}\cdot S_{tuple\_r}\cdot S_{tuple\_s}). \quad (6)$$

4 理论分析对比

本节将对上述提出的算法与传统的算法进行分析对比. 由于 C-G-H 算法是效率最高的传统磁带数据 Join 算法,我们只需与该算法进行分析比较.

4.1 Concurrent-Grace-Hash算法的分析

在数据散列操作中,G-H 算法读写关系 R,S 各一次,整个时间开销为 $T_{halfhash}=2\cdot(|R|+|S|)/X_T$.在做散列桶之间的 Join 时,关系 R 的全部散列桶需要读入内存一次,而关系 S 的全部散列桶需要读入内存 $N_{hash_trans}=|R|/(N_{bucket}\cdot|M_r|)$次.从而总的 I/O 数据传输时间代价为

$$C_{CGH\_trans}=T_{hash}+|R|/X_T+N_{hash\_trans}\cdot(|S|/X_T)=2\cdot|S|/X_T+3\cdot|R|/X_T+|R|/(N_{bucket}\cdot|M_r|)\cdot(|S|/X_T). \quad (7)$$

假设磁带上的数据均匀分布,则在完成两个关系的某一类散列桶之间的 Join 过程中,当每次从磁带上读取一个散列桶时,磁头都需要从最近访问过的散列桶处跳跃 $N_{bucket}-1$ 个其他种类的散列桶,才能到达下一次需访问的散列桶位置.这一过程需要的定位时间代价为 $T_{getnext\_locate}=[(N_{bucket}-1)\cdot|B_{ucket}|]/Tapelocate$.其次,每次扫描完一遍某类散列桶之后,磁带 R 和 S 的磁头都需要从磁带数据散列桶末尾定位到磁带数据散列桶首部,这一过程的磁头定位时间代价分别为 $T_{HL\_r}=|R|/Tapelocate$ 和 $T_{HL\_s}=|S|/Tapelocate$.总共有 N_{bucket} 类散列桶,因此 G-H 算法实现过程中所需总的磁头定位时间代价为

$$C_{CGH\_locate}=N_{bucket}\cdot\{B_{tong\_r}\cdot T_{getnext\_locate}+[|R|/(N_{bucket}\cdot|M_r|)]\cdot B_{tong\_s}\cdot T_{getnext\_locate}+T_{HL\_r}+[|R|/(N_{bucket}\cdot|M_r|)]\cdot T_{HL\_s}\}$$
$$=(2\cdot N_{bucket}-1)\cdot(|R|/Tapelocate)+(2\cdot N_{bucket}-1)\cdot[|R|/(N_{bucket}\cdot|M_r|)]\cdot(|S|/Tapelocate). \quad (8)$$

因为总共有 N_{bucket} 类散列桶,所以易知 C-G-H 算法的 CPU Join 操作时间代价为

$$C_{CGH\_join}=N_{bucket}\cdot[|R|/(N_{bucket}\cdot|M_r|)]\cdot[|S|/(N_{bucket}\cdot|M_S|/2)]\cdot\{(|M_r|/S_{tuple\_r})\cdot[(|M_S|/2)/S_{tuple\_s}]\cdot T_{probe}\}$$
$$=|R|\cdot|S|\cdot T_{probe}/(N_{bucket}\cdot S_{tuple\_r}\cdot S_{tuple\_s}). \quad (9)$$

4.2 算法实现的总时间代价比较

C-G-H 算法、D-B-G-H 算法和 T-O-G-H 算法中均实现了 I/O 操作与 CPU 操作的并发执行,因此当数据量大的时候,CPU 操作时间开销已经几乎被 I/O 操作时间开销和磁带定位时间开销所掩盖.所以,不失一般性,本文把各自的磁带定位时间代价与 I/O 数据传输时间代价之和作为这 3 种算法实现时总的时间代价.

由式(4)、式(5)、式(7)和式(8)可知,C-G-H 算法与 T-O-H-J 算法的总时间代价之差为

$$C_{CGH-TOHJ}=(C_{CGH\_locate}+C_{CGH\_trans})-(C_{TOHJ\_locate}+C_{TOHJ\_trans})$$
$$=\{(2N_{bucket}-2)\cdot[|R|/(N_{bucket}\cdot|M_r|)]-(2\cdot N_{bucket}-1)\}\cdot(|S|/Tapelocate)-2\cdot(|S|/X_T)-2\cdot T_{change}.$$

由表 1 中的参数可知,$Tapelocate\approx 4\cdot X_T$,$2\cdot T_{change}\approx 224$(s).对于处理海量数据的 Join 操作来说,由轮换磁带造成的 $2\cdot T_{change}$ 时间开销可以忽略.因此

$$C_{CGH-TOHJ}\approx\{(2\cdot N_{bucket}-2)\cdot[|R|/(N_{bucket}\cdot|M_r|)]-(2\cdot N_{bucket}-1)-8\}\cdot(|S|/Tapelocate).$$

由于$|R|$远大于 $N_{bucket}\cdot|M_r|$,从而有$|R|/(N_{bucket}\cdot|M_r|)>[(2\cdot N_{bucket}-1)-8]/(2\cdot N_{bucket}-2)$,因此 $C_{CGH-TOHJ}>0$,即 C-G-H 算法的总时间代价比 T-O-H-J 算法的总时间代价要多.

由于$|D|>|M_r|$,所以$[(N_{bucket}\cdot|R|)/(N_{bucket}\cdot|M_r|)]\cdot(|S|/Tapelocate)>(|R|/|D|)\cdot(|S|/Tapelocate)$.由表 1 中的参数可知,$X_D\approx 2\cdot Tapelocate$,从而可得$(N_{bucket}-1)\cdot X_D>N_{bucket}\cdot Tapelocate$.若令$|M_r|=|M_S|$,可得

$$\{[(N_{bucket}-1)\cdot|R|]/(N_{bucket}\cdot|M_r|)\}\cdot(|S|/Tapelocate)>(|S|/|M_S|)\cdot(|R|/X_D).$$

易知$|R|/X_T>|R|/X_D$,于是由式(1)、式(2)、式(7)和式(8)可知,C-G-H 算法与 D-B-H-J 算法的总时间代价之差为

$$C_{CGH-DBHJ}=(C_{CGH\_locate}+C_{CGH\_trans})-(C_{DBHJ\_locate}+C_{DBHJ\_trans})$$
$$>\{(2\cdot N_{bucket}-1)\cdot(|R|/Tapelocate)+2\cdot|S|/X_T+|R|/X_T+[|R|/(N_{bucket}\cdot|M_r|)]\cdot(|S|/X_T)\}-(|R|/|D|)\cdot(|S|/X_T).$$

从上式可知,随着数据量的增大,$C_{CGH-DBHJ}>0$,即 C-G-H 算法的总时间代价比 D-B-H-J 算法的总时间代价要多.

5 模拟实验结果

本文算法的模拟实现环境为 PII450,64MB 主存,400MB 硬盘,第三级存储器是一个型号为 EXA 220 的机器手磁带库.假定关系 S 的大小为关系 R 大小的 2 倍,且均存放在两条不同的磁带上.

5.1 算法的执行时间

在相同的可利用硬盘空间环境下,图 3 中显示了随着关系数据量变化时各种算法性能的差异.从实验结果可以看出,算法 D-B-H-J 和算法 T-O-H-J 的响应时间明显少于传统连接算法.基于 Nest-Loop 思想的算法性能要逊于基于 Hash 思想的算法性能,这是因为前者的 CPU Join 操作时间开销和 I/O 读取次数都要比后者多.而 D-B-H-J 算法和 T-O-H-J 算法的优势体现在降低了磁头在磁带上读取数据桶时的随机定位时间开销.

5.2 算法局部时间开销对总时间开销的影响程度实验

下面考察随着关系数据量变化,磁带定位时间开销、数据传输时间开销和 CPU 的 Join 操作时间开销分别对系统总时间开销的影响程度对比情况,实验结果如图 4~图 6 所示.

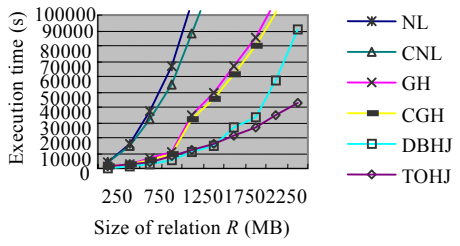

Fig.3 Comparison of execution time of Joins
图 3 各种 Join 算法的总时间开销比较

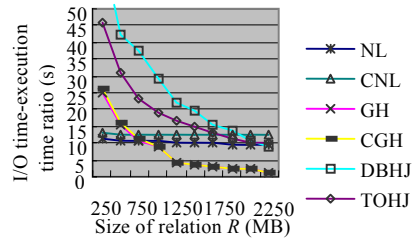

Fig.4 Comparison of ratio of transferring time to total time
图 4 I/O 开销占系统总时间开销比例的比较

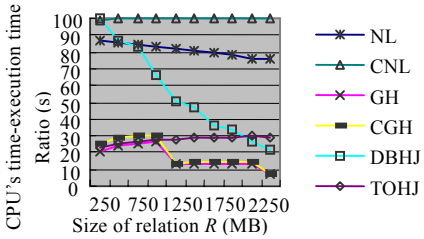

Fig.5 Comparison of ratio of CPU's time to total time
图 5 CPU 的时间开销占系统总时间开销的比例

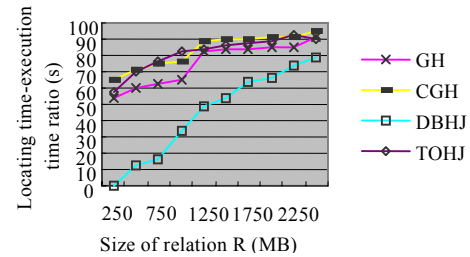

Fig.6 Comparison of ratio of locating time to total time
图 6 磁带定位时间开销占系统总时间开销的比例

如图 4 所示,随着数据量的增大,各种算法的数据传输时间开销占系统的总时间开销百分比均呈下降趋势,

最终低于 15%,说明数据传输时间开销对系统的总时间开销影响不大.由于在 C-G-H 算法、D-B-H-J 算法和 T-O-H-J 算法中均采用了 I/O 操作与 CPU 操作并发执行技术,因此随着数据量的增大,这 3 种算法中的 CPU 操作时间开销已经几乎被 I/O 操作时间开销和磁带定位时间开销所掩盖.但为了更清楚地比较各种算法中的 CPU Join 操作时间开销对系统总时间开销的影响程度,图 5 的实验结果中给出了各种算法的 CPU Join 操作开销占系统的总时间开销比值的对比情况.在基于 N-L 思想的算法中,CPU 操作时间开销几乎占系统总时间开销的 70%以上,相对来说,基于 Hash 思想的算法在这方面的比值要小一些,随着数据量的增大,达到 30%以下.原因是 N-L 算法要求一个关系中的每一个元组都要与另一个关系中的所有元组进行连接测试操作,从而造成了数据的冗余连接,也带来了数据的冗余传输.相比而言,基于 Hash 思想的算法对数据进行了散列预处理,因而只要求一个关系中的每一个元组与另一个关系中的部分元组进行连接测试操作,则调入内存的数据量大大减少.在基于 Hash 思想的算法中,D-B-H-J 算法和 T-O-H-J 算法的数据传输时间开销要比 C-G-H 算法和 G-H 算法多,这是因为 D-B-H-J 算法和 T-O-H-J 算法需要在 C-G-H 算法的基础上调整 Hash 桶的存放次序,这样就多了一些额外的数据传输时间开销.从图 6 中可以看出,随着数据量的增大,磁带定位时间开销与系统总时间开销的比值逐渐增大,最终达到了 80%以上.说明磁带定位时间开销对系统总时间开销的影响较大.

综上所述,各种算法的数据传输时间开销对系统总时间开销的影响不大,且随着数据量的增大,降到了 15%以下.对于基于 Nest-Loop 思想的算法来说,CPU 的 Join 操作时间开销对系统总时间开销的影响比较大,超过了 70%.对于基于 Hash 思想的算法来说,磁带定位时间开销对系统总时间开销的影响比较大,且随着数据量的增大,达到了 80%以上,而 CPU 的 Join 操作时间开销和数据传输时间开销分别对系统总时间开销的影响较小,且随着数据量的增大,降到了 30%以下.

6 结 论

本文提出了两种新的基于三级存储器海量关系数据 Join 算法,即 Disk-Based-Hash-Join 算法和 Tertiary-Only-Hash-Join 算法.与传统的算法相比,这两种算法考虑了磁带随机定位时间开销大,是影响基于三级存储器的数据操作算法时间复杂性的关键因素这一特点.新算法中在减少磁头定位时间方面作了一些改进,并取得了良好的性能.通过数学理论分析对比以及大量的实验考察,不仅充分说明新算法在相同内存环境下的系统响应时间少于其他传统 Join 算法,而且也说明了在研究海量数据的 Join 算法的性能时,仅使用 I/O 调度次数的多少来评价算法的性能好坏是片面的.因为从这些实验结果中可以看出,数据传输的时间开销对算法执行的总时间开销不是起决定性作用,起更大作用的是 CPU 的 Join 操作时间开销和磁头定位时间开销.从而,可以得出一个结论:在设计分析海量关系代数操作算法时,我们必须充分考虑 CPU 的 Join 操作时间开销、磁带定位时间开销和 I/O 数据传输时间开销等综合因素,尽量减少磁带定位时间开销.

References:

[1] Myllymaki J. Joins on tapes: project report [MS. Thesis]. Madison: University of Wisconsin-Madison, 1993.

[2] Kim W. A new way to compute the product and join of relation. In: Chen PP, Sprowls RC, eds. Proceedings of the 1980 ACM SIGMOD International Conference on Management of Data. Santa Monica, CA: ACM Press, 1980. 179~187.

[3] Myllymaki J, Livny M. Relational joins for data on tertiary storage. In: Alex G, Per-Ake L, eds. Proceedings of the 13th International Conference on Data Engineering. Birmingham: IEEE Computer Society, 1997. 159~168.

[4] Kraiss A, Muth P, Gillmann M. Tape-Disk join strategies under disk contention. In: Mike P, Calton P, eds. Proceedings of the 15th International Conference on Data Engineering. Sydney: IEEE Computer Society, 1999. 552~559.

[5] Myllymaki J, Livny M. Disk-Tape joins: Synchronizing disk and tape access. In: Satish T, Isi M, eds. SIGMETRICS. Ottawa: ACM Press, 1995. 279~290.

[6] Exabyte Corporation. SCSI Reference: EXB-8205 and EXB-8505 8mm Tape Drives for Standard and eXtended-Length configurations, 510503-004. 1685 38th St., Boulder, CO, 1994.

[7] Johnson T, Miller EL. Performance measurements of tertiary storage devices. In: Ashish G, Oded S, Jennifer W, eds. Proceedings of the 24th VLDB Conference. New York: Morgan Kaufmann Publishers, 1998. 50~61.