B_Link 树结构的缓存机制在数据集成中的应用

陈受凯，刘雅正
（暨南大学 计算机科学系，广东 广州 510632）

摘 要：对于一些查询密集型的应用，查询操作的响应速度往往是决定其系统性能的关键因素，因此如何提高查询响应速度和系统吞吐率成为首要任务。经过实验证明，通过将查询数据缓存可以有效地解决这个问题。

关键词：数据集成；数据缓存；B_Link 树
中图分类号：TP319 **文献标识码**：A **文章编号**：1674-7720(2012)01-0001-03

Application of buffer memory based on B_Link tree in data integration

Chen Shoukai, Liu Yazheng
(Department of Computer Science, Jinan University, Guangzhou 510632, China)

Abstract: For some query-intensive applications, query operation response time sometimes is often a key factor to determine their performance, so how to improve query response speed and system throughput is a primary task. It's proved through experiment that cache the query data can effectively solve this problem.

Key words: data integration; buffer memory; B_Link tree

目前，在企业中由于开发时间或开发部门的不同，往往存在有多个异构的在不同软硬件平台上的信息系统同时运行，这些系统的数据源彼此独立、相互封闭，使得数据难以在系统之间交流、共享和融合，从而形成了“信息孤岛”问题。如何对这些数据进行有效的集成管理，屏蔽这些信息的异构部分，并给上层用户或应用提供一个统一透明的访问接口，以透明的方式访问各信息源，成为当今企业和组织所关心的问题。数据集成也就是在这样的情况下提出来的。

通过复制的方式，将各种异构数据源中的数据抽取到一个统一的数据源中（如数据仓库），同时维护数据源整体上数据的一致性。该方式实现了对物理数据库异构的屏蔽和数据访问的控制，以便于提供一个统一的访问接口、提高访问效率和系统的独立性。图 1 是一个典型的基于数据复制方式建立的数据仓库体系结构图。

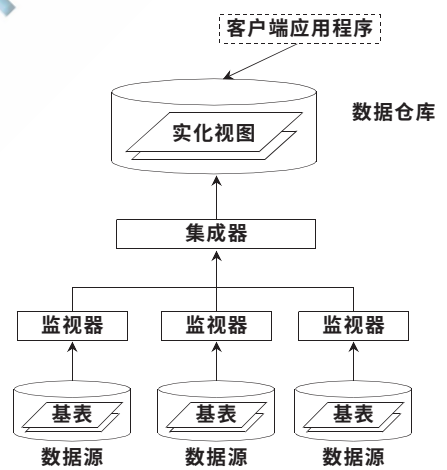

图 1 数据仓库体系结构图

此外，在把数据仓库应用于数据集成时，还需解决如何应对大量客户端应用程序对数据仓库进行访问时能够做出快速的响应。但是一个数据库的连接资源是有限的，大量的并发查询操作必定会导致查询效率的下降，导致客户端应用程序获取数据的延迟时间大大增加。这样的情况用户是不想看到的，甚至有些对时间有一定限制的应用因为不能及时获得所需的数据而无法正常工作而迫切需要解决这一问题。出现这个问题的原因是：由于连接数据库是一件耗时的工作，而且一般数据库服务器能同时提供的连接数也相当有限，出现了性能瓶颈。解决这个问题的有效方法可以通过将查询过的数据缓存起来，若下次有同样的查询时则不需再连接数

据库，而是直接从缓存中获得，这样不但节省了连接数据库的时间开销，而且省略了查询数据库所带来的时间和资源的消耗。但是对于数据缓存的组织不能简单了事，必须得经过精心的设计。为此，本文提出一种基于B_Link树的方法来管理缓存。下面将详细介绍如何组织和管理缓存。

1 数据缓存的体系结构

无论多么强大的服务器，其内存的数量都是有限的，故在考虑大数据量的缓存时，不可能将所有的数据都缓存在内存中，而要考虑使用二级缓存。基于B_Link树缓存总体结构如图2所示，为了解决内存有限的问题，采用了两层缓存结构，用一张哈希表将查询与索引表对应起来。若已经建立了缓存，则对应有一张缓存索引表，用于记录数据记录块的位置（数据记录块是指把数据记录打包成块进行保存和传输）。若数据在内存中则直接取出发送给客户端应用程序即可，否则需通过关键词去磁盘中获取。磁盘中的缓存是基于B_Link树组织起来的。

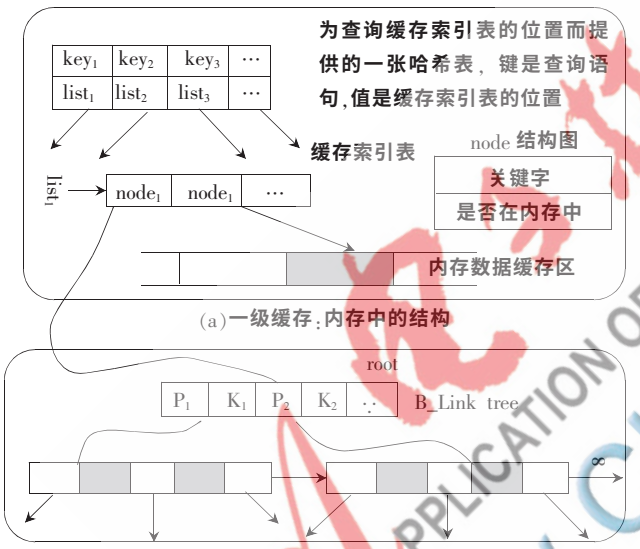

图2 基于B_Link树缓存总体结构图

2 B_Link树的基本结构和操作

2.1 数据结构

B_Link树是在B+树的基础上加以改进的，主要添加了一个高值域和非叶子结点指向其右兄弟结点的Link指针。B_Link树结构如图3所示，其中带下划线的表示高值域，主要用于提升并发操作的性能。Link指针可以使得每个结点至少可以有两条路径访问到，提高了查询速度。

2.2 B_Link树的查询操作

B_Link树的查询操作是比较容易的，具体操作可查阅参考文献[1]。其基本思路是：首先检查根结点，找到大于查询关键字V的最小搜索码值（假设搜索码值为k_i）；然后，顺着指针p_i到达另一个结点，如果查询关键字比该结点所有关键字都大，则遍历指针指向当前结点的右兄弟结点，在达到最后一层叶子结点时，若找到则返回该结点，否则返回空。

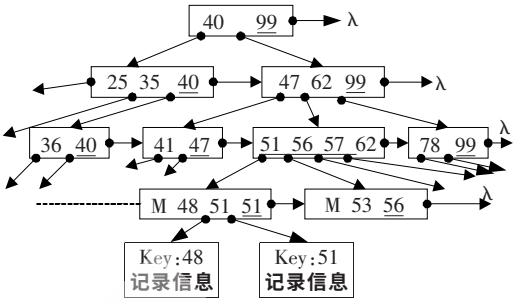

图3 B_Link树结构实例

2.3 B_Link树的插入操作

B_Link树的插入操作是一个比较复杂的过程，既要保证B_Link树的基本结构不受到破坏，还要保证并发操作时不会出现错误，所以有必要进行加锁处理。下面介绍几种操作：

（1）x←scannode(v，A)：扫描指针A指向的结点，找到能发现key值v的下一个节点并赋给x。

（2）A←get(current)：表示将current结点读入内存中并把其指针赋给A。

（3）A←node.insert(A，w，v)：把指针w和带关键字v插入指针A指向的结点。

（4）u←allocate(B)：为结点B在磁盘中分配一新的页，并将其指针赋给u。

（5）A，B←rearrange old A：把需要分裂的A指向的结点分裂成新的由A和B指向的结点。

（6）lock(current)：表示将current结点锁住，防止其他并发操作对该结点进行修改，但这并不会锁住查询操作。需要说明的是：如果查询某个结点时，这个结点进行了分裂操作，有可能会出现查询关键字大于高值域的情况，这时只要将当前指针指向右兄弟结点即可，并在父结点中添加一搜索码和指针指向右兄弟结点（细节可参考文献[1]）。

下面是插入算法的伪代码：

```
Procedure insert(v)
initialize stack;  //初始化一个堆栈，用于保存祖先结点
current←root;
A←get(current);
                   //将current读入内存中并将其指针赋给A
while current is not a leaf do
                   //从上至下遍历结点，直到叶子结点
Begin
    t←current;
    current←scannode(v，A);
    if new current was not link pointer in A then
        push(t);
    A ← get(current);
end;
lock(current);                          //将该结点锁住
```

```
A←get(current);
move.right;
                    //如果有必要将当前结点指向其右兄弟结点
if v is in A then stop
//如果原来的树中存在关键字为 v 的结点,则算法停止
w←pointer to pages allocated for record associated with v;
                                  //把与 v 相关联的指针赋给 w
Doinsertion:
if A not need to split                          //若结点无须分裂
Begin
    A←node.insert(A. w. v);
//把 w 指针和关键字 v 插入 A 指向的结点中,返回新的指针 A
    put(A, current);
                        //把指针 A 放入 current 结点适当的位置
    unlock(current);
end else begin
    u←allocate(1 new page for B);
    A,B←rearrange old A,adding v and w, to make
2 nodes;                  //把 A 指向的结点分裂成 2 个结点,
                                  //分别由 A 和 B 指向
(link ptr of A, link ptr of B)←(u, link ptr of old A);
                        //把分裂出的新结点的右兄弟结点指针
                                  //指向未分裂前 A 的右兄弟结点
    y←getmaxvalue(A);
//取出 A 指向的结点中的最大的值(注意不是高值域)
    put(B,u);
    put(A, current);          //开始将指针放入父结点中
    v ← y;
    w ← u;
current ← pop(stack);
                    //进行回溯处理,检测父亲是否需要继续分裂
    lock(current);
    A ← get(current);
    move.right;
    unlock(oldnode);
    goto Doinsertion;
end
```

其中的 move.right 操作表示将指针指向其右兄弟结点。对于删除算法,基本思路与插入算法类相同,要保持树结构和并发操作的正确性,具体可参阅文献[2]。此外,充分利用内存中的缓存也是很是重要的,在这里可以采用一种预读取的办法,即在磁盘缓存中检索到要读的结点时,在传输数据这段时间里可以将接下来的一些记录数据块读入内存缓存中,这样在下次取数据时可以减少一些磁盘 IO 操作,提高读取性能。

3 性能测试分析

测试环境如下:

数据库服务器端配置:数据库 Oracle 10g,操作系统 Linux 2.6,CPU 2.1 GHz X4,内存 4 GB,硬盘串口 500 GB。查询中间件服务器与数据库服务器是同样的配置,但不在同一台服务器上,对查询中间件的介绍可以参阅文献[3]。

客户端配置:CPU 2.6 GHz X2,内存 2 GB,硬盘串口 160 GB。表 1 和表 2 是在不同的并发查询数目和不同的查询规模下经过缓存和未经缓存所需时间的对比。表 1 中查询记录数固定为 10 万条,表 2 中并发查询数固定为 50 个。

表 1 对比不同并发查询数下的效率

并发查询数	5	10	15	30	50	75	100
未经缓存处理/s	3.87	6.01	11.65	21.98	30.36	40.83	79.68
缓存处理/s	0.61	0.97	1.52	1.59	1.89	2.06	2.24

表 2 对比不同的查询记录数下的效率

查询记录数/万条	0.5	1	10	15	20	30	40
未经缓存处理/s	3.62	8.25	30.56	40.56	59.54	85.21	115.45
缓存处理/s	0.58	1.87	1.84	2.24	2.59	3.25	4.01

通过对比直接查询数据库和经过缓存优化后进行查询的实验数据可知,通对对数据进行缓存处理,效率提升是非常明显的,尤其是在并发查询的数量比较大时,效率提升更为明显。

本文把数据缓存应用于基于数据仓库方式的数据集成中,而且,在组织缓存时采用了两级缓存结构,并采用了 B_Link 树来组织磁盘中的缓存中的结构。因此提高了并发查询数据的效率、缩短了客户端查询数据的响应时间、提高了工作效率。但本文方法还存在一些不足,例如,如何更有效地协调磁盘和内存中的缓存交互,这一设计的好坏也直接关系到查询的性能,这有待以后进一步完善和优化。

参考文献

[1] PHILIP L, LEHMAN S, Bing Yao. Efficient locking for concurrent operations on B-trees [M]. ACM Transactions on Database Systems,1981:655-663.

[2] 孙兆玉,黄宇光,朱鸿宇. 一种基于 B_Link 树结构的高性能缓存机制[J].高性能计算技术,2007,186:23-24.

[3] 房元平,许娇阳,葛珂.流水线处理技术在数据集成中的应用[J].微型机与应用,2010(24):67-69.

[4] JOHNSON T. A highly concurrent priority queue based on the B_Link tree[R]. Technical Report,University of Florida, 1991.

[5] CORMEN T H, CHARLES E. LEISERSON R L, et al. Introduction to algorithms (second edition)[M]. The MIT Press, 2009:434-453.

(收稿日期:2011-09-05)

作者简介:

陈受凯,男,1988 年生,硕士,主要研究方向:数据库,信息集成。

刘雅正,男,1987 年生,硕士,主要研究方向:数据集成,分布式应用。