PBPP:列存储系统中基于传递块缓冲区的流水线并行处理

丁祥武 张光辉
(东华大学计算机与科学技术学院 上海 201620)

摘 要 片上多核处理器(CMP)凭借其低功耗和低成本等优势迅速成为处理器市场的主角,它为多线程的实现提供了硬件支持。列存储技术在分析型应用中具有显著的优势。在列存储系统中,查询优化依然是最重要的问题之一。在列存储系统中,利用多核资源提高查询处理性能具有较大潜力。文中通过对查询执行器生成的物理查询树进行流水多线程设计,结合列存储的特点,建立传递块缓冲区,使主线程和辅助线程分别对传递块缓冲区读写,以提高查询性能。同时还提出使用操作系统中经典的"生产者和消费者"模式来解决线程之间的同步问题。提出的这些方法应用在实验室研发的列存储系统DWMS中,使用数据仓库基准测试集SSB验证了这些方法的有效性。实验结果表明,传递块缓冲区的设计使SQL的查询效率有了近50%的提升。

关键词 多线程,多核,列存储,传递块缓冲区,并行处理
中图法分类号 TP311 **文献标识码** A

PBPP:Pipelined Parallel Processing Based on Passing Buffer in Column-store System

DING Xiang-wu ZHANG Guang-hui
(School of Computer Science and Technology, Donghua University, Shanghai 201620, China)

Abstract Chip multiprocessor(CMP) with low-power dissipation, lowcost advantages becomes rapidly the leading role of the market, and it provides hardware support for multithread. Column-store has significant advantages in analytical applications. Query optimization is one of the key issues in column-store. In column-store, multi-core resources can improve performance of query processing. In order to improve query performance of column-stores, this paper established passing block buffer to make main thread and worker thread to read and write respectively different passing blocks, so parent node and child node of physical execution tree execute parallel. We used classic producer-consumer pattern to solve the problem of synchronization between the threads. In column-stores DWMS developed by our laboratory, experimental results on benchmark data set SSB show the effectiveness of this design, and it can improve 50% execution performance for some typical complex queries.

Keywords Multithread, Multicore, Column-store, Passing block buffer, Parallel processing

1 引言

数据分析型应用日渐普遍和重要。大量的研究表明[2-4]:列存储在分析型应用中具有显著的优势。列存储系统中的查询处理速度一直是用户关注的重点,是列存储研究的热点问题。

由于功耗和设计的限制,单纯地提高处理器主频已经非常有限。现在处理器的发展趋势已经从单核高频处理器转向了片上多核处理器(CMP),由指令级并行向多线程并行发展[5]。设计高效的多线程,充分发挥多核处理器的优势,能显著提高运算性能。在列存储系统的查询过程中,通过查询语句的不同操作和操作内使用多线程可以提高查询性能。本文的主要研究工作是通过采用传递块缓冲区来提高操作节点之间的并行性。

本文提出了一个基于传递块缓冲区的流水并行化设计,通过改变执行树中上下级操作节点传递数据的方式,提高列存储的查询效率。这种设计将上下级操作节点之间直接通过传递块传递数据的方式改变为通过传递块缓冲区传递数据,对传递块缓冲区的读写分别采用不同的线程进行,这样可以充分利用多核CPU多线程的优势,提高查询效率。另外对传递块大小、缓冲区大小及其缓冲区的数量进行优化设置,进一步提高查询优化的性能。

本文第2节主要介绍了多线程查询优化方面的相关工作;第3节主要介绍了本文所涉及的基本概念,尤其是传递块缓冲区的设计;第4节是本文的重点,详细设计了基于传递块缓冲区的查询执行;第5节进行相关实验验证;最后对本文的工作进行了总结。

到稿日期:2013-08-18 返修日期:2013-10-20 本文受"核高基"国家科技重大专项基金项目(2010ZX01042-001-003-004),国家自然科学基金项目(61070031,61070032),上海市自然科学基金项目(11ZR1401200)资助。
丁祥武(1963—),男,博士,副教授,主要研究方向为数据库、列存储、分布式处理技术等,E-mail:dingxw@dhu.edu.cn;张光辉(1986—),男,硕士生,主要研究方向为Java工作流、列存储等。

2 相关工作

列存储相对于行存储的众多优势使得它在分析型应用，甚至在大数据处理中逐步被采用。

在行存储系统的查询处理过程中，最常用的数据流模型是基于拉式(pull-based)迭代器模型，根节点向它的孩子节点调用“getNextTuple”，然后孩子节点又向它的孩子节点调用“getNextTuple”，直到树的叶子节点。一个节点“getNextTuple”之后数据被处理并传送到它的父节点作为“getNextTuple”调用的返回值。包括 C-Store[3] 在内的大多数列存储系统采用一次一块的处理方式替代一次一元组的方式，将“getNextTuple”的请求变为“getNextBlock”，这样一次函数调用得到的是一组数据(64k)[6]。如果将“getNextTuple”的请求变为“getNextAll”，即将全部列读取内存，这样肯定会导致中间结果过大而采用物化的方式，物化则会降低查询效率。C-Store 等系统每次处理一组数据的确使得查询性能有较大提升，但 C-Store 系统毕竟是基于单个线程的[6]。在这个单核 CPU 几乎被淘汰的今天，为了提高查询性能，多线程设计已成为必然。

列存储系统的按列存储按需索取数据的特点，使得列存储的查询处理有更多的并行机会。假设 A 表有 item1 和 item2两个属性，B 表有 item3 属性，C 表有 item4 属性。如果有一条查询语句涉及到 item1 与 item3 连接和 item2 与 item4 连接，则它们可以用两个线程分别进行处理，当然单个连接的不同阶段都可以采用多线程设计[7]，这都属于水平并行化。垂直并行化(即列存储中物理执行树上下操作节点的并行)则研究得较少，要想实现流水线并行操作，垂直并行化是必要的。

由于采用单进程单线程设计，类似于 C-Store 的列存储系统一般是拉式迭代器模型，暂时还达不到查询的流水线操作。为了实现垂直并行化以达到流水线并行化操作，这里我们建立了传递块缓冲区(Passing Buffer)。父节点向 Passing Buffer 调用“getNextBlock”的时候，首先会建立一个辅助线程，辅助线程再向下一个 Passing Buffer 调用“getNextBlock”，一直这样下去直到叶子节点。叶子节点向上返回数据，这时中间节点依靠辅助线程，将得到的块放入缓冲区，然后继续向下请求数据块。这样就形成了流水并行化操作，查询执行树的所有节点都处理忙碌的状态，流水线一直处于流动状态。

针对多核处理器进行查询优化有多种方式，优化共享 CACHE、建立并行缓冲区和对某些操作使用高效的并行算法等也都是比较有效的优化方法。

CACHE 是为了缓解 CPU 与内存速度之间的不匹配而设计的缓存结构。充分利用 CACHE 来提高查询效率已经得到很多学者的重视。文献[8]提出用主/辅线程机制来提高查询效率，其主要思想是将主线程要访问的内存通过辅助线程加载到 CACHE 中，减少主线程的 CACHE MISS。相当于将主线程的 CACHE MISS 转移到辅助线程，而辅助线程的 CACHE MISS 造成的性能损失不会影响到主线程。文献[9]提出了并行缓冲区，通过避免多个线程同时对一个缓冲区进行读写时的线程冲突，提高操作内的并行度。在我们的系统中建立了传递块缓冲区，使向传递块缓冲区写数据的操作节点与从中读取数据的操作节点并行执行，提高操作间的并行性。对缓冲区的操作只有两个线程，一个是主线程，另一个是辅助线程。主线程负责从缓冲区中读取数据，辅助线程负责将数据读入缓冲区。文献[10]针对 hash-join 操作提出了在多线程环境下的流水线设计，并对线程数量进行优化控制，同时将工作量划分到各个线程中，对缓冲区进行管理使 CACHE MISS 最小化。

多线程的使用必然会带来线程同步问题，操作系统中经典的“生产者和消费者”模型[12]能很好地解决这一问题。生产者向缓冲区放入数据，消费者则从缓冲区中读取数据，并且保证缓冲区中的数据不溢出。生产者消费者模型如图 1 所示。我们实际的设计中“生产者”和“消费者”都可以有多个，只是我们的查询处理父子节点之间的数据传递是明显的单个“生产者”和单个“消费者”。

图 1 生产者消费者模型

3 基本概念

3.1 传递块

传递块(PASS_BLK)是列存储系统的查询执行中使用的一种数据结构，存储子节点向父节点传递的一组数据。

在列存储系统的物理执行树中，从底层的数据扫描到最后的输出结果，查询处理能像流水线一样进行工作，子节点通过传递块向父节点传递数据。对传递块的设计，行存储和列存储有明显不同，在行存储中一条记录的所有数据聚合存储在一起，每次读取一条记录都将记录中所有值读取到内存，下级节点向上级节点传递的数据包含了操作需要的所有数据。而在列存储中则不同，列存储的特点是将所有记录中相同字段的数据聚合存储。查询处理时，只提取与操作相关的列，不相关的列则不读入内存。由于每列数据均以〈rowid,value〉的形式存储在磁盘上，因此在列存储中，传递块中传递的数据可以包括所需数据或者所需数据的行号。传递块结构如图 2 所示。

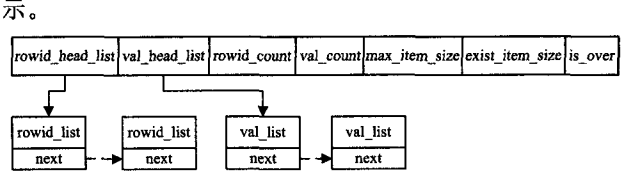

图 2 传递块结构

其中，rowid_head_list 是指向 rowid 列表的指针，val_head_list 是指向 val 列表的指针，rowid_count 是 rowid 列表中 rowid 的个数，val_count 是 val 列表中 val 的个数，max_item_size 是传递块中最大的 rowid 和 val 的个数，exist_item_size 是传递块中存在的 rowid 和 val 的个数。

行号列表和列值列表是分别存放行号和列值的数据结构。当子节点要向父节点传递数据时，先将子节点要传递的

行号和列值分别存放在行号列表和列值列表中，如有多个行号或者列值，则存放在多个行号列表和列值列表中。行号列表与行号列表、列值列表与列值列表之间都是 next 指针指向其下一个。而传递块中则存放了指向行号列表的 rowid_head_list和指向列值列表的 val_head_list 的指针。

3.2 传递块缓冲区

在介绍传递块缓冲区之前，我们先说明引入缓冲区的必要性。假设一个查询树的父节点从子节点中获取了一个传递块，子节点继续向父节点传递数据。如果这时父节点接受新传来的数据，就覆盖了旧数据，造成数据访问混乱；如果不接受，子节点的数据只能丢弃或者等待。即父子操作无法并行。如果为父子节点各自设置一个传递块结构，数据覆盖问题即可得到解决。与此相关的问题是如何解决这两个传递块按顺序传递，父子节点如何进行协调，这两个传递块是否够用等问题。这时缓冲区的优势就显现出来了。将多个传递块放入一个统一的缓冲区，然后对缓冲区进行协调管理，可以保证按正确顺序访问数据，保证正确的并发执行。

传递块缓冲区(PASS_BUF)是临时存放传递块的一片连续的内存区域，是为了使操作之间并行而设计的缓存结构。传递块缓冲区的结构如图 3 所示。

图 3 传递块缓冲区结构

其中，l_cur 为左游标，表示传递块数组中下一个将要被读取的传递块所在数组的位置；r_cur 为右游标，表示下一个将要存入的传递块在数组中的位置；num 为传递块的数量，表示当前数组中传递块的数量；PASS_ARRAY 为传递块数组，用来存放传递块；PASS_BLK 就是 3.1 节中的传递块。

3.3 主线程与辅助线程

父子节点分别对 3.2 节的传递块缓冲区进行读写，这样父子节点就可以分别使用线程对缓冲区进行读写。

如图 4 所示，这时的生产者是子节点，消费者是父节点。我们这里将父节点从传递块缓冲区中读取传递块数据的线程称作主线程(M_T)，将子节点向传递块缓冲区写入传递块的线程称作辅助线程(PRE_T)。

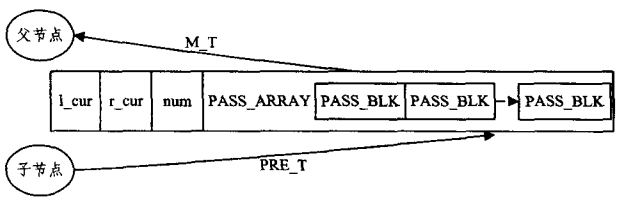

图 4 父子节点数据传递流程

4 流水线并行查询处理设计

4.1 已有列存储系统的查询处理

列存储系统的查询处理与行存储系统相似，首先对解析的 SQL 语句进行词法语法分析生成语法分析树，并对语法分析树进行预处理生成查询树；然后对查询树进行基于规则的优化生成初步的逻辑计划，将初步逻辑计划基于代价的优化转换为最终的逻辑执行计划；最后将逻辑执行计划转换为物理执行计划，也即是物理查询执行树。

对查询语句 Q1：

```
Q1: select sum(lineorder. extended_price * lineorder. discount) as revenues
from lineorder, date where lineorder. orderdate=date. datakey
and date. year=1993
and lineorder. discount >2
```

最终生成的物理执行计划可形象地用如图 5 所示的树结构表示。

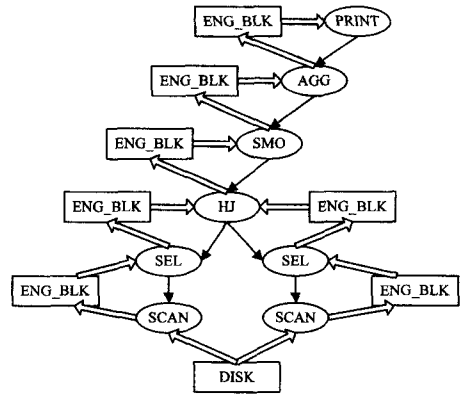

图 5 查询 Q 的物理执行树

SCAN：列扫描节点，从磁盘中读取数据。

SEL：选择节点，将上一节点传递过来的数据进行选择操作，将符合选择条件的数据或者符合条件的 rowid 以传递块的方式传递给上一节点。

HJ：hash-join 节点，在这里 hash-join 节点从左右孩子中提取数据，进行 hash-join 操作，将结果传递给上一节点。

SMO：select 语句中的数学操作，这里就是 lineorder. extended_price * lineorder. discount 操作。将操作的结果传递给上一聚集节点。

AGG：聚集节点，将下一操作节点传递过来的数据进行聚集操作。操作之后传递给打印节点。

PRINT：打印节点，输出最终的结果。

由图 5 可知，所有操作节点的数据传递都是通过传递块一块一块进行的，每一个操作节点从它的孩子节点接收传递块，从中取出数据进行运算，将运算之后的结果写入传递块中，并将传递块传给它的父亲节点。最终由 PRINT 节点输出结果。由于 hash-join 操作的特殊性，它将左右孩子中一个孩子的全部数据读入内存建立 hash 表，然后另一个孩子逐块与 hash 桶内的数据进行 hash。由于内存的限制，维表的数据一般较小，通常都是针对维表的数据建立 hash 表。

4.2 查询的并行化设计

由图 5 所示的查询执行树可知，数据是至底向上逐块地传递，操作节点从孩子节点请求得到一个传递块，操作节点运算之后再取下一传递块是典型的串行化执行方式，至始至终只有一个线程在工作，并不能充分利用现代多核 CPU 的计算能力。采用多线程设计能够充分发挥 CPU 能力，提高列存储的查询效率。

下面以查询树中的 hash-join 节点为例，介绍现有查询操

作的流程：

(1)hash-join 节点从左孩子节点中读取全部数据建立 hash 表；

(2)从右孩子请求得到一个传递块，并将该传递块的游标(l_blk_cur)重置为 0；

(3)然后从 l_blk_cur 位置开始，进行 hash-join 操作；

(4)在 hash-join 操作的过程中，如果 hash-join 的结果等于传递块的大小，则将结果以传递块的形式传递给 SMO 节点，并记录 l_blk_cur 的位置，继续(3)；如果 l_blk_cur 等于传递块的大小，则判断从右孩子得到的传递块是否是最后一个，是则继续(2)，否则结束。

整个流程如图 6 所示。其中 PASS_BUF 为图 1 中的传递块结构。

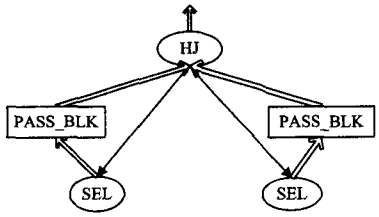

图 6　HJ 节点的操作流程

为了能进行流水线并行化操作，我们将 3.2 节中设计的传递块缓冲区用于以上查询设计。加入传递块缓冲区之后，查询处理过程发生了较大的变化。未加入缓冲区之前，操作节点直接从孩子节点请求获取传递块。加入缓冲区之后，父节点从传递块缓冲区中请求获取传递块，传递块缓冲区从孩子节点请求获取传递块。

下面以查询树中 hash-join 节点为例，介绍引入传递块缓冲区后的查询处理过程：

(1)hash-join 节点从左孩子节点中读取全部数据建立 hash 表。

(2)初始化传递块缓冲区，并将指针(pass_buf_p)指向该缓冲区，初始化信号量(FULL(0，PASS_ ARRAY_SUM)，EMPTY(PASS_ARRAY_SUM，PASS_ ARRAY_SUM))和互斥区(Mutex)，建立 PRE_T 线程，同时执行(3)和(10)。

主线程(M_T)：

(3)等待 FULL 信号量 wait(FULL)。

(4)等待互斥量 wait(Mutex)。

(5)初始化一个传递块 PASS_BUF，hash-join 节点从传递块缓冲区得到传递块 pass_array[pass_buf_p->l_cur]，并赋值给 PASS_BUF，并将该传递块游标(l_blk_cur)重置为 0。PASS_BUF 中左游标加 1(l_cur++)，如果 l_cur 等于最大值 PASS_ARRAY_SUM，则从最左端开始，即pass_buf_p->l_cur=(l_cur+1) % PASS_ARRAY _SUM。

(6)释放互斥量 signal(Mutex)。

(7)释放 EMPTY 信号量 signal(EMPTY)。

(8)从传递块的 l_blk_cur 位置进行 hash-join 操作。

(9)在 hash-join 操作的过程中，如果 hash-join 的结果等于传递块的大小，则将结果以传递块的形式传递给 SMO 节点，并记录 l_blk_cur 的位置，继续(8)。如果 l_blk_cur 等于传递块的大小，则判断从传递块缓冲区得到的传递块是否是最后一个，是则继续(3)，否则结束。

辅助线程(PRE_T)：

(10)等待 EMPTY 信号量 wait(EMPTY)。

(11)等待 Mutex 互斥量 wait(Mutex)。

(12)PASS_BUF 从右孩子节点请求得到一个传递块。然后右游标加 1(r_cur++)，如果 r_cur 等于 PASS_ARRAY_SUM，则从最左端开始，即 pass_buf_p->r_cur=(pass_buf_p->r_cur)% PASS_ARRAY_SUM。

(13)释放互斥量 signal(Mutex)。

(14)释放 FULL 信号量 signal(EMPTY)。

(15)如果传递块不是最后一个传递块则继续(10)，否则结束。

整个流程如图 7 所示。其中传递块缓冲区 PASS_BUF 的结构为图 4 中的 PASS_BUF。这里是简化的 PASS_BUF，其只保留了用来存放传递块的传递块数组 PASS_ARRAY。

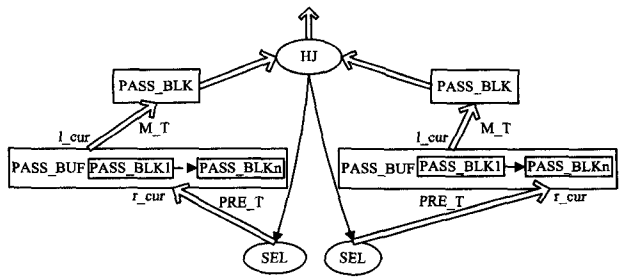

图 7　带有传递块缓冲区的 HJ 节点操作流程

这里主要以 HJ 节点为例介绍了基于传递块缓冲区的并行化设计，但这种设计不仅仅适用于 SEL 节点向 HJ 节点传递数据，HJ 节点向 SMO 节点传递数据的时候同样可以这样进行。任何两个节点的数据传递都可以采用这种设计方式进行，这里就不再累述。

4.3　相关参数对查询性能的影响

由于本文采用了传递块缓冲区的设计，这里我们重点考虑两个参数：一个是传递块的大小(max_item_size)，另一个是传递块缓冲区中传递块的数量(PASS_ARRAY_SUM)。关于 max_item_size，如果设置太小，无关操作甚至会超过 50%，并且在共享最后一级缓存的计算机中 cache miss 增加，查询效率将大幅下降。如果 max_item_size 设置太大，会占用大量的内存，查询性能不但没有提升反而会下降。关于 PASS_ARRAY_SUM，如果设置比较小，在 hash 操作比较费时的情况下，使 PRE_T 线程长期处于等待状态，并不能充分发挥缓冲区的作用。如果设置太大，会占用大量内存资源，传递块缓冲区中始终达不到满状态，如果传递块比较大，会造成大量的内存浪费。具体第 5 节中的实验将会详细说明。

5　实验

本节实验主要是验证传递块缓冲区的使用对查询性能的影响，也测试了一些参数对查询性能的影响。为了保证数据的真实有效性，本节实验中每一个数据均是进行多次实验取其平均值而得到的。

5.1 实验环境

本文实验基于列存储数据仓库管理系统 DWMS 进行，使用数据仓库基准测试集 SSB[19] 作为测试数据。硬件实验环境如表 1 所列。

表 1 实验硬件环境

硬件名称	硬件参数
处理器	Intel i3 380M 2.53GHz 2 核心 4 线程
CACHE	2 * 64kB(一级) 2 * 256kB(二级) 3MB(三级)
内存	2 * 2=4GB 1333MHz
磁盘	320GB 300MB/s(数据传输率) 8MB(缓存)

5.2 DWMS 与 SSB

DWMS 是我们实验室开发的数据仓库管理系统，底层采用列存储的形式存储数据，每列的数据都以(rowid，value)的形式呈现。

本文的实验采用星形模型 SSB 测试中定义的数据集进行验证。此模型是星形模式下的真实数据集，共有 5 张表组成，即 1 张事实表(lineorder)与 4 张维度表(date、part、customer 和 supplier)。

5.3 使用缓冲区前后对比

实验使用 SSB 测试集中比较具有代表性的查询语句 Q1.1 作为实验测试查询语句。

Q1.1：

```
select sum(lineorder.extended_price * lineorder.discount) as revenues
from lineorder,date
where lineorder.orderdate=date.datekey and date.year=1993
and lineorder.discount between 1 and 3
and lineorder.quantity < 25;
```

图 8 显示了使用传递块缓冲区前后查询语句执行时间的变化，这里取传递块大小为 30000 个(指每个传递块所指向的所有行号列表 rowid_list 和值列表 val_list 所能容纳的最大元组个数)，每个传递块缓冲区中传递块数量为 2。

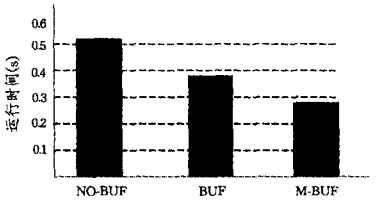

图 8 传递块缓冲区使用前后对比

NO-BUF：不使用传递块缓冲区；

BUF：仅使用单个传递块缓冲区；

M-BUF：使用多个传递块缓冲区。

由图 8 可知，使用传递块缓冲区之后 Q1.1 的查询效率有了很大程度的提高。使用单个缓冲区时大约比未使用缓冲区提高 30%，使用多个缓冲区比未使用缓冲区提高约 50%。主要原因是多传递块缓冲区有更好的并行性。

5.4 参数对查询性能的影响

(1)传递块缓冲区大小对查询的影响

图 9 显示了 Q1.1 查询语句不同的传递块缓冲区大小对查询的影响，这里使用多个传递块缓冲区，每个传递块大小为 30000 个。

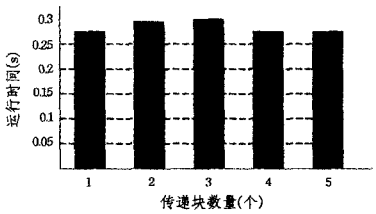

图 9 传递块缓冲区大小的影响

由图 9 可知，传递块缓冲区中传递块的数量对整个查询的影响不大，每个传递块缓冲区能装下一个或者两个传递块即可。传递块数据数量太多，会占用较多的内存，并且性能没有提升。

(2)传递块缓冲区所在位置对查询的影响

这里我们主要测试仅使用一个传递块缓冲区的时候，传递块所在节点之间的位置对查询性能的影响。实验结果如图 11 所示。

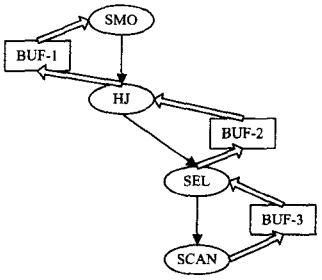

图 10 物理查询树部分节点

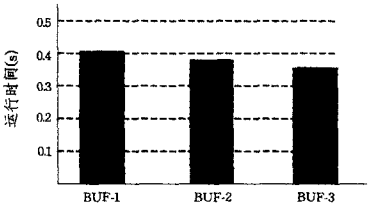

图 11 传递块缓冲区位置的影响

其中，图 11 中 BUF-1、BUF-2 和 BUF-3 在物理查询树中的位置分别与图 10 中 BUF-1、BUF-2 和 BUF-3 对应。

当我们采用单个缓冲区时，缓冲区离叶子节点越近查询效率越好。经分析可知，这主要是因为缓冲区离叶子节点越近，主线程的运算越多，等主线程运算完一个传递块，辅助线程已经将数据读入到缓冲区，这样缓冲区中已经存在传递块，主线程再次向缓冲区请求传递块数据的时候便立即得到一个传递块。相反，如果缓冲区离根节点越近，主线程处理一个传递块所用时间越少，处理完再次请求缓冲区时，辅助线程很可能还没有将传递块读入缓冲区，这样主线程就处于等待状态，影响查询效率。

(3)传递块大小对查询的影响

本节测试使用了 3 种模式(NO-BUF、BUF 和 M-BUF)来进行实验，在 3 种模式下分别测试了不同传递块大小对查询性能的影响，结果如图 12 所示。

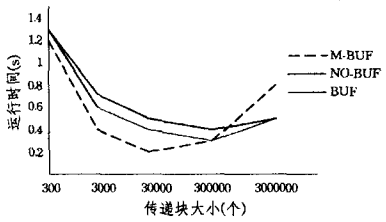

图 12 传递块大小的影响

由图 12 可知，无论是否采用传递块缓冲区，传递块大小对查询性能的影响都非常大。当传递块很少时，比如传递块大小为 300 个时，NO-BUF、BUF 和 M-BUF 的查询时间都急剧增加，这主要是因为传递块小时，CACHE MISS 增加，无关操作所占比例过大所致。而缓冲区过大对查询效率的提高不大，比如传递块大小为 300000 个，NO-BUF、BUF 和 M-BUF 并没有多大的区别。当传递块大小为 3000000 个时，M-BUF 的查询时间反而最大，这主要是因为传递块数据占用内存较多，当计算的数据超过缓存大小时查询效率反而因 CACHE MISS 下降。当传递块大小为 30000 个时，无关操作所占比例较小，也不容易发生 CACHE MISS，所以当传递块大小为 30000 个时查询语句的效率提升非常明显，M-BUF 比 NO-BUF 有近 50%的性能提升。

结束语 本文在列存储的研究基础之上，提出了传递块缓冲区的概念，并将其应用到了实际的列存储系统 DWMS 中，通过使用不同的线程读写传递块缓冲区，充分利用了现代多核 CPU 的优势，提高了查询效率，且对相关参数进行合理设置进一步提高了查询效率。第 5 节中的实验验证了我们设计的合理有效性。

现在我们更多的工作只关注了列存储在全共享单机多核 CPU 环境下的并行因素，未来为了达到更大的并行性，准备将列存储数据库系统部署到分布式集群计算机中，通过对数据进行划分，使集群计算机中每个节点分别对不同的数据划分块进行运算，以获取更大的并行度，提高列存储系统的查询效率。

参 考 文 献

[1] Copeland G P, Khoshafian S N. A decomposition storage model [C]// Proceedings of SIGMOD International Conference. Austin, Taxes: ACM, 1985: 268-279

[2] MacNicol R, French B. Sybase IQ multiplex-Designed for analytics[C]// Proceedings of the 30th Very Large Data Base Conference. Toronto, Canada: VLDB Endowment, 2004: 1227-1230

[3] Stonebraker M, Abadi D J, Batkin A, et al. C-Store: A column-oriented DBMS[C]// Proceedings of the 31st Very Large Data Base Conference. Trondheim, Norway: VLDB Endowment, 2005: 553-564

[4] Boncz P A. Monet: A next-generation DBMS kernel for query-intensive applications[D]. Amsterdam: Universiteit van Amsterdam, 2002

[5] Hennessy J L, Patterson D A. Computer Architecture(4th ed) [M]. Morgan Kaufman Publishers, 2007

[6] Abadi D J. Query Execution in Column-Oriented Database System[D]. MIT PhD Dissertation, 2008

[7] Blanas S, Li Yi-nan, Patel J M. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs[C]// Proceedings of the ACM SIGMOD Conference. Athens, Greece: ACM, 2011

[8] Zhou Jing-ren, Cieslewicz J, Ross K A, et al. Improving database performance on simultaneous multithreading processors[C]// Proceedings of the 31st international conference on Very large data bases. Trondheim, Norway: VLDB Endowment, 2005: 49-60

[9] Cieslewicz J, Ross K A, Giannakakis I. Parallel buffers for chip multiprocessors[C]// Proceedings of the 3rd International Workshop on Data Management on new Hardware. New York, USA: ACM, 2007

[10] Garcia P, Madison, WiHenry, et al. Pipelined hash-join on multithreaded architectures[C]// Proceedings of the 3rd international workshop on Data management on new hardware. NewYork, USA: ACM, 2007

[11] Nehme R, Bruno N. Automated Partitioning Design in Parallel Database Systems[C]// Proceedings of the ACM SIGMOD Conference. Athens, Greece: ACM, 2011: 1137-1148

[12] 汤子瀛，哲凤屏，汤小丹. 计算机操作系统[M]. 西安：西安电子科技大学出版社，2002

[13] Harizopoulos S, Shkapenyuk V, Anastassia. QPipe: a simultaneously pipelined relational query engine[C]// Proceedings of the 2005 ACM SIGMOD international conference on Management of data. New York, USA: ACM, 2005: 383-394

[14] Hardavellas N, Pandis I, Johnson R, et al. Database servers on chip multiprocessors: Limitations and opportunities[C]// CIDR. 2007: 79-87

[15] Manegold S, Boncz P, Nes N, et al. Cache-conscious radix-decluster projections[C]// Proceedings of the Thirtieth International Conference on Very Large Data Bases. VLDB Endowment, 2004: 684-695

[16] Rao Jun, Zhang Chun, Megiddo N, et al. Automating physical database design in a parallel database [C]// Proceedings of the 2002 ACM SIGMOD Interna-tional Conference on Management of data. New York, USA: ACM, 2002: 558-569

[17] Cieslewicz J, Mee W, Ross K A. Cache-conscious buffering for database operators with state [C]// Proceedings of the Fifth International Workshop on Data Management on New Hardware. New York, USA: ACM, 2009: 43-51

[18] 吴峻峰，许跃生，张永东，等. CC$：一种面向分布式众核平台的并行编程语言[J]. 计算机科学，2013，40(3)：128-132

[19] O'Neil P, O'Neil B, Chen Xue-dong. Star schema bench-mark Revision 3 June 5 [EB/OL]. http://www.cs.umb.edu/~poneil/, 2010-02