文章编号:1000-5641(2014)05-0252-11

# 批处理在内存数据处理系统中的应用

周 烜, 薛忠斌
(中国人民大学 数据工程与知识工程教育部重点实验室, 北京 100872)

**摘要**: 内存数据处理相较磁盘数据处理有明显的速度优势. 在基于磁盘的数据管理系统中,设计者往往会花很多精力对事务响应时间进行调优,以提高应用的用户体验. 在内存数据管理系统中,由于存储介质的改变,事务响应时间得到极大提升,甚至远远超出应用系统的需求. 因此,系统设计者将注意力转移到对吞吐率的优化上. 批处理技术的本质是通过牺牲响应时间换取吞吐率,它将在内存计算中得到广泛应用. 本文讨论批处理方式在内存数据管理系统中的应用,并以移动对象管理为实际案例,验证批处理在内存数据管理中的优化效果.

**关键词**: 内存数据库; 批处理; 吞吐率
**中图分类号**: TP392 **文献标识码**: A **DOI**:10.3969/j.issn.1000-5641.2014.05.022

## Batch processing for main memory data management

ZHOU Xuan, XUE Zhong-bin
(*DEKE MoE Key Laboratory, Renmin University of China, Beijing* 100872, *China*)

**Abstract**: Main memory based data processing is substantially faster than disk based data processing. When developing a traditional Disk Resident DBMS, various optimization techniques are required to ensure that query response time meet the requirements of general applications. This is less necessary for a Main Memory DBMS, whose response time usually goes far beyond the requirements of most applications, due to the superior speed of main memory. As a result, throughput becomes a more important concern for system design. The central idea of Batch Processing is to achieve improved throughput at by trading off response time. Therefore, we believe that batch processing will play an important role in main memory centered data processing. This paper attempts to provide some insight on how to apply the idea of batch processing to speedup Main Memory DBMS. A case study on in-memory moving object manage is used to demonstrate the effectiveness of batch processing.

**Key words**: main memory database; batch processing; throughput

## 0 引 言

内存计算近年来在工业界和学术界都备受关注. 随着存储技术和工艺的发展,内存的容

收稿日期:2014-06
基金项目:国家自然科学基金(61272138)
第一作者:周烜,男,副教授,研究方向为数据库与信息检索. E-mail: zhou.xuan@itlook.com.
通信作者:薛忠斌,男,博士研究生,研究方向为数据库. E-mail: zbxue@ruc.edu.cn.

量几乎以每年一倍的速度增长.如今的高端服务器已经可以配备 4TB 甚至更高容量的内存.在很多应用领域,几乎可以将全部业务数据放入内存,从而明显提升整个软件系统的性能.可以推测,在不久的将来,基于磁盘的传统数据库系统将逐渐被内存数据库系统取代.对内存数据库而言,针对磁盘 I/O 的优化策略将不再是系统设计的重点;如何提高内存的访问效率成为了提升系统性能的关键.

众所周知,CPU 访问内存的速度远远滞后于 CPU 访问寄存器的速度;CPU 与内存之间的多级缓存机制(即 L1、L2、L3 Cache)成为缓解这一速度差异的主要机制.因此,为了提升内存计算的性能,首先需要提升多级缓存的命中率.为了提升缓存命中率,就必须提升指令和数据访问的局部性.近期的一些研究表明[14],在去除 I/O 瓶颈之后,传统数据库系统的指令和数据访问局部性并不理想,存有较大优化空间.这促使研究人员开始重新思考数据库系统的设计方案.可以预见,代码和内存访问的效率会成为今后数据库系统研发的重点,并推动数据库性能的进一步提升.

本文着重探讨用于提升内存数据库性能的一类策略 — 批处理优化.常用于衡量数据库性能的指标包括响应时间和吞吐率.除一些特殊领域外(如金融领域的自动交易),大部分领域对响应时间的要求并不十分苛刻,只要它能跟上用户的反应速度即可;而普通应用往往对系统吞吐率的要求很高,希望系统能够承受尽量多用户带来的高强度工作负载,从而获取软硬件的最佳性价比.当内存取代磁盘成为数据的主要存储介质后,系统的性能得到大幅提升,大大缓解了响应时间方面的压力,使得吞吐率成为了衡量系统性能的主要指标.批处理作为一种优化系统吞吐率的常用方案就变得尤其有效.批处理优化的主要目标是将多个查询请求合并,同时处理;一方面,如果多个请求共享公共的数据操作,合并后可以减少这些操作的重复调用,减轻系统的负载;另一方面,系统可以调整批处理作业的执行次序,对位于同一区域的数据尽量一起访问,对位于同一区域的代码也尽量一并执行,这有利于提高数据和指令的局部性,提升系统效率.虽然批处理优化可能增加单个查询的响应时间,但能够有效提升系统的吞吐率,这与内存数据库对吞吐率的偏重相契合.

近年来的一些学术研究[3,5]已经开始关注批处理在内存数据库中的应用.ETH 的系统团队(System Team)开发的 SharedDB 系统和 EPFL 的数据库团队(DB Team)开发的 StagedDB 都是典型的例子.本文将对这些相关的研究成果作概要介绍,也提出一些作者本人对批处理优化的初步思考.本文还将介绍一个将批处理优化用于移动对象管理的案例,通过实际应用对批处理优化的有效性做初步验证.

## 1　相关研究

批处理在传统数据库中已经得到了一定程度的应用.一个典型例子是数据库的批量插入操作(用于将大量数据导入数据库).但类似的批处理操作与本文涉及的批处理优化有所区别.前者是可被单独调用的数据库操作,后者则是对多个操作的整体优化.传统数据库的查询结果缓存和多查询优化更符合批处理优化的模式.本节将对批处理优化的一些已知方法做简单的概括和分析.这些方法可以分为三大类:查询结果缓存、多查询优化、分阶段查询处理.

### 1.1　查询结果缓存

查询结果缓存是数据库系统的一项常见功能.当某查询执行完毕并返回结果后,系统一

般不急于丢弃查询结果,而是先将其在缓存中保留一段时间;若用户又提交相同的查询,在不违背时效性的前提下,系统无需重新执行查询,可直接返回缓存的结果给用户.对于数据访问方式有限且用户众多的应用,查询结果缓存的优化效果非常明显.因此,几乎所有的数据库系统都不同程度地实现了查询结果缓存的功能.

查询结果缓存可以有不同的实施粒度,各自的效果不同.最粗粒度的实施方法仅缓存整个查询的最终结果;在这种粒度下,只有在用户重复提交相同查询时,缓存的结果才可以被重用.更细粒度的实施方法是将查询的中间结果也一并缓存(中间结果可以是子查询的结果,也可以是查询计划);当后续查询涉及相同的中间结果时,缓存中的内容就可以被重用.此外,查询结果的重用机制也有多种.最直接的重用机制要求用户提交的查询或子查询与缓存中的查询或子查询完全一致,否则不执行结果重用.更复杂的机制则将缓存中的查询结果物化为视图;当用户新提交的查询与该物化视图不完全一致,但被该视图包含时,系统将新查询改写为可直接实施于该物化视图的查询,在一定程度上达到重用的目的.很明显,以上的各种查询结果缓存的实现方案各有优缺点.简单的方法可重用率低,复杂的方法本身也可能成为系统的负担.方案的执行效率往往取决于应用负载的特点.由于不同数据库厂商的考虑不同,采取的实施方法也不尽相同.

作为批处理优化的一类方法,查询结果缓存可以实现查询之间相同数据访问操作的重用,节省数据访问和计算的开销.这种方法同样适用于内存数据库.然而,这种重用机制的粒度较粗(一般以查询或子查询为单位,并无法达到指令级),并且不能针对性地提高指令和数据访问的局部性.方法虽然简单实用,但并不是一种彻底的批处理优化方式.

### 1.2 多查询优化

相比查询结果缓存,多查询优化(Multi-Query Optimization)[4]实现了更加彻底的查询间的操作重用和资源共享.多查询优化更加符合批处理的模式,它并不逐个执行用户提交的查询,而是先积累一批查询后再对其进行统一执行.通常,多查询优化会为若干查询构建一个统一的查询计划,通过执行这个查询计划完成所有查询.在统一的查询计划中,多个查询公用的数据尽量做到一次性访问,公用的代码尽量做到一次性执行,从而尽可能避免查询间的重复工作.这种重用机制比查询结果缓存机制粒度更细.

传统的多查询优化的研究成果并不多,其方法一般以优化 I/O 操作为主,且只针对于只读查询.新近的一些研究则开始将多查询优化向内存数据库扩展,并且逐步纳入了数据更新的操作.SharedDB[1,2,3]就是一个典型的实行多查询优化的内存数据库系统.这一系统不仅仅实现了传统多查询优化的计算资源共享,而且还纳入了数据流的查询方式 — 它根据应用需求生成一个统一的查询计划,用于满足应用中遇到的所有查询需求;这一查询计划常驻系统,每次都被用于对若干查询请求进行同时处理.图 1 为 SharedDB 针对 TPC-W 应用生成的一个查询计划,用于同时处理 TPC-W 中的若干查询.

实验证明,多查询优化在某些情况下可以明显提升系统的吞吐率.从理论上讲,这种方式也有利于提升指令和数据访问的局部性,从而提高 CPU 缓存的命中率.然而,现有研究成果对多查询优化中的操作重用问题探讨得更多,对指令和数据局部性问题涉及得不多.另外,多查询优化如何合理地利用多核处理器,还是一个有待深入探讨的问题.

### 1.3 分阶段查询处理

传统数据库系统对单个查询或事务的执行是线性的. 一个事务请求的执行通常经过多

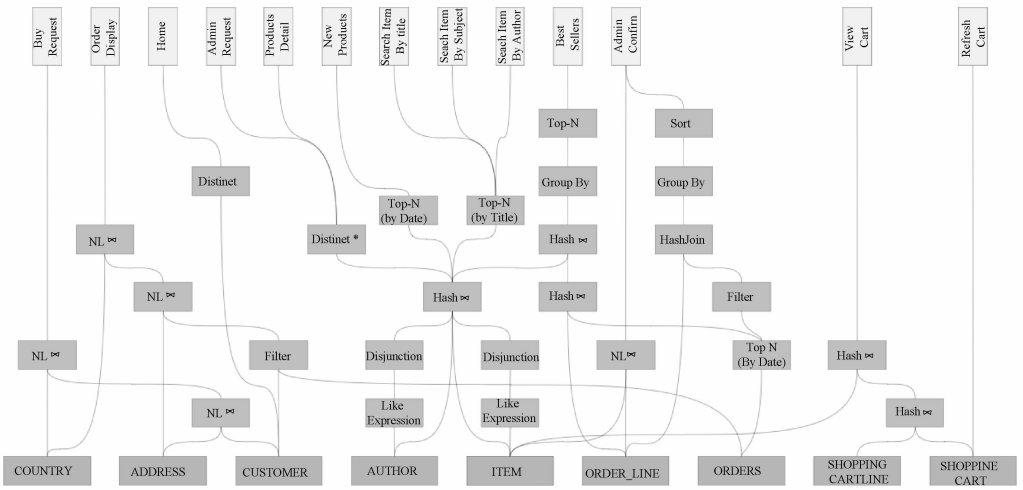


图 1　SharedDB 针对 TPC-W 生成的查询计划(摘自文献[3])

Fig. 1　Query plan generated by SharedDB for TPC-W

个阶段,包括查询解析、查询优化、以及查询执行过程中的选择、投影、连接等多个操作.传统数据库一般使用单线程依次执行这些阶段,执行完毕后立即将结果返回给用户.这种执行方式的指令局部性和数据访问局部性都是有限的,并不能很好地发挥 CPU 缓存的效率.而分阶段查询处理则将事务执行的多个阶段分配给不同的模块,每个模块享有独占的 CPU 和内存资源.这样,事务执行的过程不再是绝对线性的,而是将任务分配给各个模块;当所有模块完成任务后,事务即执行完毕.每个模块可以自行决定任务的执行方式,可以按照先进先出的方式顺序执行,也可以将多个任务按批处理的方式执行.总之,分阶段查询处理的方式有利于在模块内实现较高的指令和数据局部性.如果系统对模块的资源分配是合理的,就可以获得较高的缓存命中率和系统性能.分阶段查询处理虽然不是显式的批处理优化,但其使用的效果与批处理优化相似,都涉及多个查询的资源共享.

分阶段查询处理的思想最早在 CMU 的 StagedDB 项目[5-7]中提出.该方法除了有助于提升指令和数据访问的局部性之外,还具备模块化、并行度高等优点.虽然这些优势在理论上是成立的,但至今还没有实际系统完整地采用这种方法.因此,分阶段查询处理的实现技术还需要进一步探索,它的实用性也有待检验.

## 2　批处理优化面对的基本问题

从以上的相关研究成果,我们看到,批处理优化对数据库系统的性能提升可以归结于两个方面:首先,由于普通应用对数据库的访问模式是相对单一的,在时序上相邻的访问请求往往共享一定的数据库操作或中间结果;通过批处理优化,我们可以让不同的查询和事务共享这些操作和中间结果,减少数据库的工作量,提升系统效率;其次,通过使用批处理优化,系统可以灵活调整程序执行和数据访问的顺序,提高内存访问的局部性,提升缓存的利用率.批处理优化对吞吐率的提升是明显的,但会延长单个请求的响应时间,因为每个查询通常需要等待同一批查询全部完成后才能返回结果.这一点也是制约批处理优化应用的主要因素.在内存数据库中,由于查询和事务的响应时间已经被压缩得很短,这就扩大了批处理

优化的应用空间.因此,内存数据库适合更深入的批处理优化.

通过总结批处理优化的相关技术,我们认为批处理优化需要考虑以下问题:

(1) 查询相似性:批处理优化要求被同时处理的查询或事务具有一定的相似性,否则就难以达到操作共享或提升内存访问局部性的效果.查询和事务或者具有相似的数据访问模式(使得查询计划或指令可以共享),或者它们会访问相同或相邻的数据(使得数据操作可以共享).如果查询之间不具备以上的特性,批处理优化只能在一定程度上提升程序的指令局部性,优化效果有限.单一领域的应用对数据库的访问模式有限,查询之间一般具有较高的相似性,通常适合使用批处理优化.但对于一些复杂应用,由于其数据模式复杂且访问方式多样,查询可能表现出较弱的相似性,不一定适合批处理优化.

(2) 执行顺序改变带来的影响:批处理优化的核心步骤是改变原有的查询或事务执行次序,尽量将相同或相似的操作放在一起执行,从而达到操作共享的目的并提升内存访问局部性.执行顺序的改变使得传统数据库系统的很多设计方案需要做出调整,比如锁管理和并发控制的实现方法等,因为这些模块大都是按照单事务单线程模式实现的.与此对应,系统的一些优化技术也需做出相应调整化.如何对整个系统进行改造以适应批处理方式是批处理优化的关键问题.

(3) 多核利用率:传统数据库系统在大多数情况下使用顺序的事务执行模式,并发事务通过操作系统的线程调度被自动分配到不同的处理核心,从而自然地实现了多核扩展(虽然这样的多核扩展受到临界资源的种种限制).当使用批处理优化后,这样的多核扩展模式被打破,如何分配多核处理器资源,成为另一个关键问题.由于批处理可以比较自由地调整查询和事务的执行顺序,这有利于系统减少临界资源的竞争,从而获取更高的多核扩展性.然而,如果系统不能有效地控制负载均衡,多核资源也难以获得充分利用.部分使用批处理优化的系统(如 SharedDB 和 StagedDB)都给出了各自的多核并行方案.但这些方案都有待进一步验证.

我们认为,任何批处理优化方案都需要对以上问题做出回答,否则其优化效果就难以得到保证.当然,批处理优化在具体实施时还会遇到其他各式问题,本文无法一一列举.本文以空间移动物体数据管理为场景,对批处理优化做一些初步尝试,并汇报一些对该领域的研究可能有参考意义的实验结果.

## 3 批处理优化一个应用实例

我们将批处理优化应用到移动对象查询上.考虑与人群相关的移动对象,例如:手机用户或者车辆.移动对象周期性地向中央服务器报告位置,更新各自的位置信息.为了方便描述,我们将每个移动对象当做空间中的一个点,用 OID 进行标识,其位置为二维欧氏空间中一个坐标$(X,Y)$.用户对移动对象的查询主要包括范围查询和 KNN 查询两类.我们着重考虑范围查询.一个范围查询一个矩形区域,通过区域的左下角坐标$(X_{\text{low}}; Y_{\text{low}})$和右上角坐标$(X_{\text{high}}; Y_{\text{high}})$定义.查询结果为当前落在查询区域的移动对象.

移动对象在空间内自由移动,位置信息持续更新,范围查询持续不断的到来.对移动对象管理系统而言,这些都是较高的负载.为了保证查询的实时性,传统的方法是在移动物体上建立高效的索引,既有利于快速更新,也能够快速响应查询.然而,如此高性能的索引的设计难度是很大的.如果将移动对象信息完全存放在内存中,批处理优化则可以派上用场.

### 3.1　批处理优化的实施

为了利用批处理优化,将位置更新请求和范围查询请求看做两个数据流,分别称为更新流和查询流. 当更新和查询抵达时,系统并不立即对其做出响应,而是将它们缓存起来. 当一个时间窗口结束时,将所有缓存的更新以批处理的方式进行实施,然后再统一对缓存中的查询进行应答. 只要保证时间窗口的大小满足应用对实时性的要求,就可以达到批处理优化的效果.

由于我们的方法未使用索引,位置更新的批处理实现很简单,这里不再赘述. 我们主要考虑对多个范围查询进行批处理应答的方法. 假设移动对象和查询分别被存放在两张表中,整个批处理过程可以通过对两张表的一个连接操作完成. 连接的结果是将移动对象与它们所满足的查询进行配对. 空间对象连接算法的相关研究成果已经非常丰富. 参考这些研究成果,我们制定出以下的批处理方法.

批处理方法分为三个阶段. 第一个阶段为索引移动对象阶段,创建一个临时 Grid 索引结构,把移动对象用 Grid 进行索引. 第二阶段为索引查询阶段,用 Grid 结构索引所有的查询. 经过前两个阶段,对象和查询分别映射到相应的单元格中,在每个单元格中通过把移动对象和查询执行连接操作,得到查询结果.

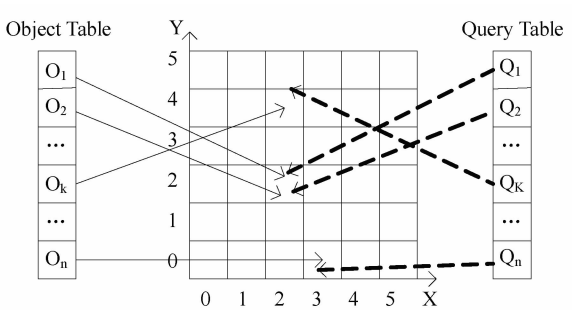


图 2　连接算法示例

Fig. 2　DSJ example

图 2 为一个查询示例,首先索引移动对象,对移动对象进行聚类操作. 如图 2 所示,对象 $O_1$ 和 $O_2$ 都位于格网索引的单元格(2, 2)中,通过聚类操作,把两个对象聚集在对象表的相邻位置. 然后处理查询. 对查询进行聚类操作时,算法在查询所涉及的每个单元格中保存一个副本,避免了执行过程中,跨单元格访问导致的 cache miss. 通过上面的两步操作,完成对对象和查询的聚类,提高了访问查询索引时数据的空间局部性,进而提高 cache 命中率. 例如处理 Q1 后,接着处理 Q2,则不会将单元格(2, 2)中的数据从 cache 中替换出去,从而提高 cache 的命中率.

### 3.2　并行方案

对于对象和查询的聚集操作,我们参考了文献[8]中的多核并行方案,通过使用数据分布直方图的方式,使操作可以充分并行执行,发挥了多核的性能. 执行过程中对数据进行两次顺序扫描,避免了使用锁机制,消除了执行过程中空间竞争. 对于连接操作,每个线程处理一个单元格的数据,多个线程并行执行,线程间不存在竞争,充分发挥了多核的性能. 除了范围查询,这种方式也适用于 KNN 查询,因为后者很容易从范围查询中得到结果.

用来索引移动对象和查询的 Grid 结构中,每个单元格作为一个数组结构,被放在不同

的内存页中.内存页频繁的换入换出,保存虚拟内存到物理内存映射的页表被保存在 TLB 中.文献[13]指出,划分会产生大量页表,若页表数量超过 TLB size,查找过程中会产生 TLB miss.因此,TLB size 限定了划分数目的上限.我们在 Grid 索引创建过程中为了消除 TLB miss,采用了层次 Grid 的方式.在第一层 Grid 中单元格的数目小于 TLB size,消除了 TLB miss.在构建第二层 Grid 时,确保单元格中数据能够完全放在内存中,消除 cache miss.

当 Grid 结构构建完成后,通过多线程并行执行的方式,索引移动对象.移动对象被存储在连续的数组中.在构建索引时,充分考虑 cache conscious 特性,把位于相同单元格中的对象放在一起,减少后续调用时 cache miss.

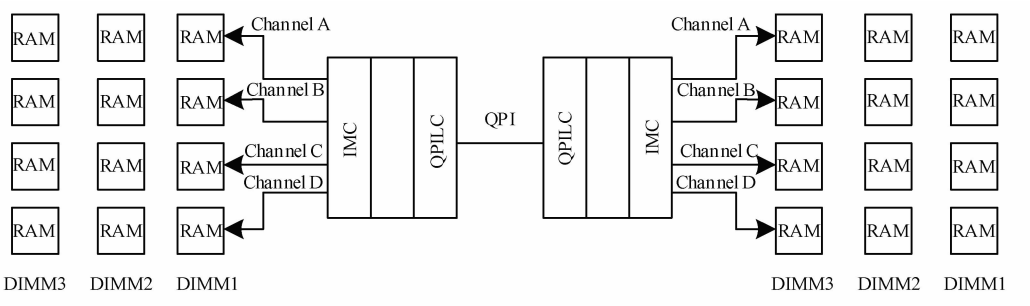


图 3 范围查询的划分(左)和赋值(右)

Fig. 3 Partition and assignment for range query

通过前两步操作,查询和移动对象分别存放在对应的单元格中,各个单元格中数据相互独立,可以多线程并行执行.在执行过程中,采用 round-robin 的方式,每个线程处理一个单元格,把单元格中的对象和查询执行连接操作,最后把结果汇总.在单元格中连接运算,为计算密集型操作.对于连接运算,我们首先采用了传统的算法:桶链连接算法(bucket-chaining-join)和嵌套循环连接(nested-loops-join)算法.桶链连接算法在文献[13]中提出,其主要思路是把单个元组串联形成一个桶结构,然后用数组位置作为指针(而不是内存中实际的指针),通过这种方式提高内存算法的效率.

由于移动对象的用户访问方式单一,查询的相似性高,适合使用批处理优化.我们提出的批处理优化方法使得系统在两个方面获得性能提升:首先,相邻的移动对象和查询被聚集到相同的 Grid 单元里,使得它们能够共享大量计算和数据访问操作,提升了内存访问的局部性;其次,更新的查询之间的冲突被规避,使得程序可以充分利用多核并行提升效率.

## 4 初步实验结果

### 4.1 数据集

在实验中,我们使用了基于德国实际路网生成的数据集.实际数据集中包含了整个德国的路网,由 3.8 亿个节点和 4 亿个路段构成,覆盖 641 km × 864 km 的面积.用开源的移动对象路径生成器 MOTO[10] 生成数据.MOTO 是在数据生成器 Brinkhoff[9] 的基础上形成的.MOTO 中采用了一个基于路网的对象布局方式,所有的移动对象都随机分布在一个给定的路网中,设定的移动对象的最大车速 Smax = 60 m/s = 216 km/h.数据生成器还按照现实中城市人口的分布进行了修改,一半的对象分布在 5 个主要的德国城市,因此能够确保

更新最频繁的区域同时也是查询最多的区域.

我们的算法用 C/C++实现,用 g++在最高的优化等级下进行编译. 实验运行在 32 核(4 Intel E5-2670@2.6GHz)的计算机上,使用了 SUSEOS 11 (64-bit)系统,有 256G RAM,片上的内存被所有的线程共享. 若无特别说明,所有的实验都在 32 核中完成.

4.2　批处理优化的性能表现

图 4 中显示不同数量的移动对象执行 500 百万个查询时,查询的响应时间. 横坐标表示移动对象的数量,纵坐标为查询的响应时间. 通过图中可以发现,随着数据量的增加,查询的响应时间呈线性增长. 在不同的移动对象的数据量下,所有查询的响应时间小于 2 秒. 对于一般应用而言,移动对象的数量都在百万级别. 即同时处理 500 万的查询,对大多数应用而言,都可以获得亚秒级的响应,完全可以应对大部分用户对响应时间的要求. 因此,移动对象应用是非常适合使用批处理优化的.

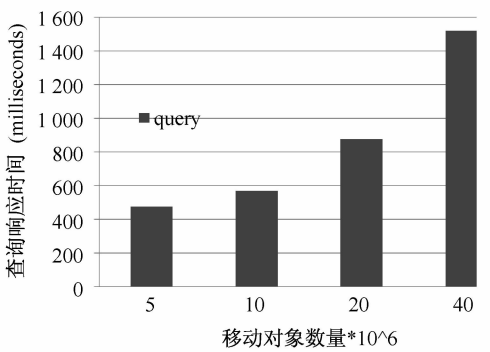


图 4　查询执行时间

Fig. 4　The response time of range query

图 5 显示了我们的批处理算法随线程数增加吞吐量的变化趋势. 算法在对对象和查询构建索引时,通过建立数据分布直方图,提前计算出对象所在位置,避免了线程间的空间竞争问题. 同时,每个查询所在单元格都保存一个副本. 当执行连接操作时,通过 round-robin 的方式,每个线程处理一个单元格中的数据,各个线程独立执行,避免了线程间的竞争. 通过图 5 可以看到,随线程数的增加算法的吞吐量呈线性增长. 当超过系统的物理核数 16 时,算法的吞吐量仍缓慢增长. 总之,算法对多核的利用是很充分的.

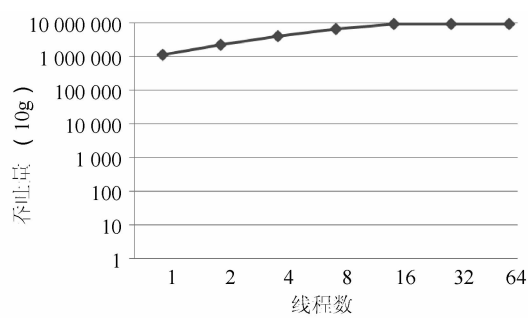


图 5　多核扩展性

Fig. 5　The scalability of multi-core

### 4.3 对比试验

我们进一步对比了批处理优化和传统非批处理方法的性能. 对比对象为 PGrid 算法[12] 和 TwinGrid 算法[11]. 两者将数据完全存储在内存中,且都使用动态索引,是目前文献中性能最好的算法.

图 6 对比了不同方法执行一批查询的响应时间. 通过图中可以看出,当执行 500 个查询时,TwinGrid 算法的查询总体响应时间最长. 在 TwinGrid 算法中,查询和更新分别在两个 Grid 结构中,每隔一定的时间间隔,需要把 write-store 复制到 read-store 中,复制操作占用大量的时间. 随着查询数量的增加,PGrid 算法的查询性能逐渐下降. PGrid 算法中,查询和更新在同一个 Grid 结构中,为了避免冲突,查询执行过程中采用了加锁机制,增加了查询的处理时间. 对于 TwinGrid 算法在执行过程中,更新和查询分别在两个 Grid 结构中,不存在冲突的问题. 当查询数量增加时,其性能优势逐渐明显. 对于批处理优化的方法,性能表现是最优的,随着查询数量的增加,优势更加明显. 我们的批处理优化采用了每次执行一组查询的方法,在执行过程中,通过索引查询的方式,增加了数据的局部性,实现查询内的并行,提高了算法的效率.

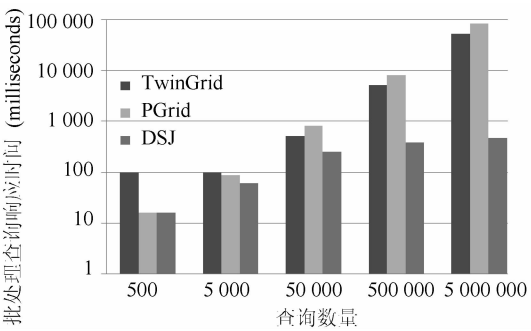


图 6 更改查询数量的性能表现

Fig. 6 Performance of changing the number of queries

图 7 中显示了查询范围变化时,两个算法的吞吐率表现. 随着查询范围的增加,需要涉及到更多的单元格和对象,对索引的性能会有影响. 图 7 中看到,TwinGrid 算法和 PGrid 算法随着查询范围的增加,吞吐量保持平稳. 批处理优化方法随着查询范围的增大,需要进行更多的比较,致使性能有所下降,但其吞吐量仍比 TwinGrid 算法和 PGrid 算法高一到两个数量级.

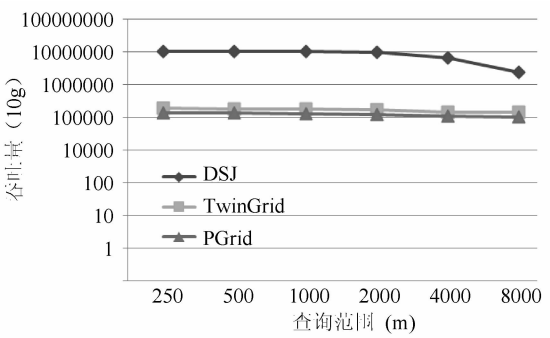


图 7 更改查询范围的性能表现

Fig. 7 Performance of changing the query range size of queries

图 8 显示改变更新和查询的比例，从 2000∶1到 1∶4时，PGrid 算法、TwinGrid 算法和批处理优化方法在执行过程中吞吐量变化. 随着查询数量的不断增加，需要进行更多的运算，三个算法的吞吐量都会下降. 通过图中可以看到，当查询增加时，PGrid 算法的吞吐量下降更快. PGrid 算法中采用锁机制去保持数据一致性，随着查询数量的增加，竞争更激烈，导致性能下降. TwinGrid 算法在执行过程中，查询和更新分别在两个结构中，执行过程中不存在冲突，提高了查询的效率. 批处理优化方法采用每次处理一组查询的方式，充分利用数据的局部性，提高了算法效率. 批处理优化方法将更新进行缓存，消除了查询和更新过程中冲突. 因此，当大数据量下，查询和更新大量涌入时，该方法也能保持较高的系统吞吐量.

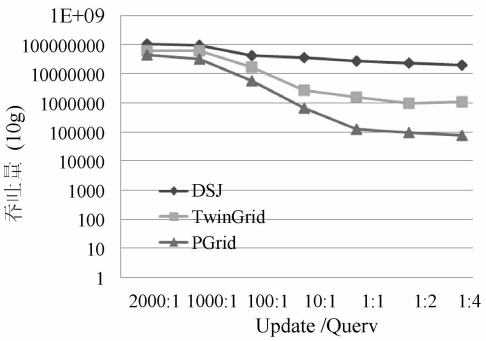


图 8　改变查询更新比率的性能表现

Fig. 8　Performance of changing the query update rates of quering

## 5　结　　论

当使用内存作为数据存储设备后，数据管理系统摆脱了 I/O 操作的性能瓶颈，并且迎来了新的性能优化空间. 我们认为，批处理优化对于内存数据库而言是一种行之有效的优化方法. 本文以移动对象管理的案例，对批处理优化在内存数据管理中的应用进行了初步探讨和验证. 还总结了批处理优化获得性能提升的两个基本途径：操作共享和内存访问局部性. 同时也分析了批处理优化的适用范围和实施过程中面临的潜在问题. 通过在移动对象管理上的一些初步实验，我们发现批处理优化可以带来相当明显的性能提升.

如何将批处理优化应用到通用的关系数据库是一个更加复杂且更具有实用价值的问题. 人们对此已经做了一些初步探索，但还存在大量未解决的问题，也尚未出现完整的系统原型. 我们认为，批处理优化将是一个会出现丰富实用成果的研究领域.

## [参　考　文　献]

[1] GIANNIKIS G, MAKRESHANSKI D, ALONSO G, et al. Shared workload optimization[C]//PVLDB 2014, 7(6): 429-440.

[2] GIANNIKIS G, MAKRESHANSKI D, ALONSO G, et al. DEMO: Workload optimization using sharedDB[C]//SIGMOD 2013, New York, 2013, USA, June 22-27:[s. n.]

[3] GIANNIKIS G, ALONSO G, KOSSMANN D. SharedDB: Killing one thousand queries with one stone[C]//PVLDB, 2012, 5(6): 526-537.

[4] SELLIS T K. Multiple-Query Optimization[J]. ACM Trans. Database Systems, 1988, 13(1): 23-52.

[5] HARIZOPOULOS S, AILAMAKI A. StagedDB: Designing Database Servers for Modern Hardware[J]. IEEE

Data Eng Bull, 2005, 28(2) pp. 11-16.

[6] HARIZOPOULOS S, SHKAPENYUK V, AILAMAKI A. QPipe: A simultaneously pipelined relational query engine[C]//Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland:[s. n.], 2005: 383-394.

[7] HARIZOPOULOS S, AILAMAKI A. Improving instruction cache performance in OLTP[J]. ACM Transactions on Database Systems, 2006, 31(3): 887-920.

[8] BALKESEN C, ALONSO G, TEUBNER J, et al. Main-memory hash joins on modern processor architectures [J]. IEEE Transactions on Knowledge and Data Engineering (TKDE), 2014.

[9] BRINKHOFF T. A framework for generating network-based moving objects. GeoInformatica 6.2 (2002): 153-180.

[10] DITTRICH J, BLUNSCHI L, VAZSALLES M A. Indexing moving objects using short-lived throwaway indexes. Advances in Spatial and Temporal Databases. Springer Berlin Heidelberg, 2009. 189-207.

[11] ŠIDLAUSKAS D. Thread-level parallel indexing of update intensive moving-object workloads. Advances in Spatial and Temporal Databases. Springer Berlin Heidelberg, 2011. 186-204.

[12] ŠIDLAUSKAS D, ŠALTENIS S, CHRISTIAN S. JENSEN S. Parallel main-memory indexing for moving-object query and update workloads[C]//Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. ACM, 2012.

[13] MANEGOLD S, BONCZ P A, KERSTEN M L. Optimizing main-memory join on modern hardware[J]. IEEE Trans Knowl Data Eng, vol. 14, no. 4, pp. 709-730, 2002.

[14] TÖZÜN P, GOLD B, AILAMAKI A. (2013) OLTP in Wonderland- Where do cache misses come from in major OLTP components? [C]//Proceedings of the 9th International Workshop on Data Management on New Hardware, pp. 8:1-8:6.

(责任编辑 王善平)