文章编号: 1001- 2486(2010) 03- 0053- 07

一种支持高效并发访问的移动对象索引*

赵 亮, 陈 荦, 景 宁, 钟志农
(国防科技大学 电子科学与工程学院, 湖南 长沙 410073)

摘 要: 针对移动对象当前及未来位置索引不能有效支持多用户并发访问的问题, 提出了一种支持高效并发访问的移动对象索引 CS^{2}B$^+$-tree(Concurrent Space-filling curve enabled Cache Sensitive B$^+$-tree) 。该索引结合了 B^x-tree 和 CSB$^+$-tree 的特点, 因而能够支持对移动对象进行预测查询且具有缓存敏感特性。重点研究了一种针对 CS^{2}B$^+$-tree 的两层锁并发访问机制, 特别是设计了一种网格锁备忘录结构, 使得索引能够支持多任务并发执行。基于并发访问机制, 分别提出了 CS^{2}B$^+$-tree 的并发更新算法及并发预测范围查询算法。实验表明, 相对于 B^x-tree, CS^{2}B$^+$-tree 的并发访问的吞吐量提高了 15. 1%, 响应时间减少了 14. 9%。

关键词: 移动对象索引; 并发访问; 缓存敏感
中图分类号: TP392 文献标识码: A

An Efficient Moving Object Index that Supports Concurrent Access

ZHAO Liang, CHEN Luo, JING Ning, ZHONG Zhi-nong
(College of Electronic Science and Engineering, National Univ. of Defense Technology, Changsha 410073, China)

Abstract: Current literature on indexing current and future positions of the moving objects lacks the mechanisms on concurrent access. To solve this problem, the current research proposed an efficient moving object index that supports concurrent access, also called CS^{2}B$^+$-tree(Concurrent Space-filling curve enabled Cache Sensitive B$^+$-tree). CS^{2}B$^+$-tree combines the characteristics of the B^x-tree and CSB$^+$-tree, thus it can support querying the predicted future positions of the moving objects and is cache sensitive. Focus was put on studying a concurrent access mechanism to CS^{2}B$^+$-tree which resulted in a two-level lock mechanism and particularly a lock memo structure was designed. Based on the concurrent access mechanism, a CS^{2}B$^+$-tree concurrent location update algorithm and a concurrent predicted range query algorithm were proposed respectively. Experimental results show that, compared with B^x-tree, the throughput of the CS^{2}B$^+$-tree improves by 15. 1%, and the response time decreases by 14. 9%.

Key words: moving object index; concurrent access; Cache sensitive

位置服务的广泛应用催生了对移动对象数据管理的研究, 过去十年中出现大量移动对象数据管理的研究工作, 由于索引技术在查询处理中的基础作用, 所以在移动对象数据管理研究成果中尤以移动对象索引技术为多。然而目前位置服务的主要应用在于导航和定位服务, 绝大部分位置服务仅支持对静态空间对象的查询处理。基于移动对象的位置服务应用较少, 与移动对象理论研究成果形成鲜明反差。就移动对象索引技术来说, 我们认为存在以下问题极大地限制了其广泛应用: (1) 大部分提出的索引结构都是基于单用户环境展开研究, 基本没有考虑多用户的并发访问情况, 与实际应用极不相符; (2) 大部分提出的索引结构都是基于磁盘存储, 同时采用内存中的缓存替换算法减少磁盘 I/O, 然而缓存替换策略不能保证对于内存的最优利用, 因而造成基于磁盘索引的查询处理性能(主要指查询处理时间) 不高。

目前, 随着内存价格的下降以及容量的不断增大, 将所有的数据放入内存进行处理将成为现实; 64 位处理器逐渐成为主流, 极大程度地增加了计算机内存的可用容量[1]。开源的 MonetDB、商业的 TimesTen都是比较典型的关系型内存数据库系统[2], 在内存数据库理论和实际应用方面进行了开创性研究。内存数据库技术的发展和应用, 意味着对位置服务这种查询密集型应用来说, 将所有数据放入内存

* 收稿日期: 2009- 12- 25
基金项目: 国家 863 高技术研究发展项目(2008AA12A211) ; 国家自然科学基金资助项目(40801160)
作者简介: 赵亮(1982—), 男, 博士生。

进行处理以提高移动对象数据库查询性能已成为现实[3]。此外,在移动对象的应用中,由于其频繁更新,对移动对象的经常性、持久化存储是非常耗时且不必要的。

基于上述认识,我们研究了移动对象当前及未来位置的内存索引,提出一种支持高效并发访问的移动对象索引 CS^2B^+-tree(Concurrent Space-filling curve enabled Cache Sensitive B^+-tree)。

1 CS^2B^+-tree

本文提出一种支持高效并发访问的移动对象索引 CS^2B^+-tree。它是一种移动对象当前及未来位置内存索引结构,用以支持对移动对象进行预测查询。它的特点正如其名字所示:一是支持高效的并发访问,二是缓存敏感。所谓缓存敏感,主要是指针对处理器 Cache 参数,比如 Cache 容量、Cache line 大小和映射维度等,优化算法中的数据结构和算法执行,减少 Cache 访问缺失,提高 Cache 利用率。对于 CS^2B^+-tree 来说,其缓存敏感性体现在其底层采用了类似 CSB^+-tree 的结构,而为了使其支持并发访问我们设计了 CS^2B^+-tree 的并发访问机制。

1.1 CS^2B^+-tree 结构

CS^2B^+-tree 采用类似于 B^x-tree 的构建思想,即使用网格(Grid)将移动对象存在的空间进行划分,而后将单元格中移动对象的位置信息、速度矢量信息和更新时间信息通过空间填充曲线映射到一维空间中。两者的不同之处在于前者的底层采用多棵 CSB^+-tree 索引一维键值。

CS^2B^+-tree 的原理及结构如图1所示。

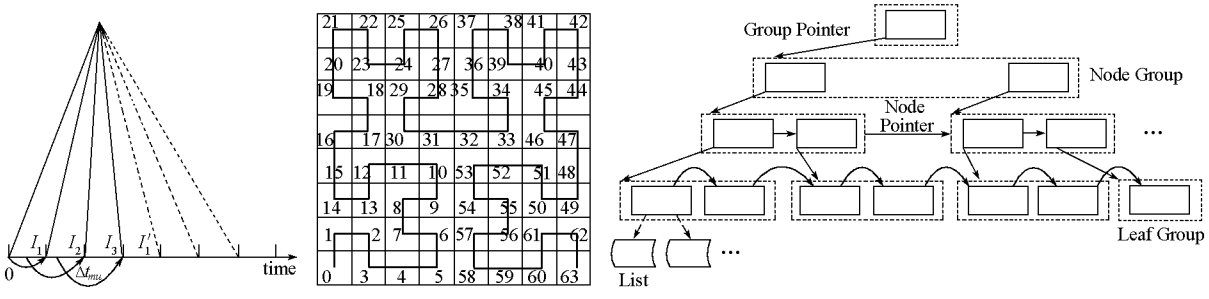


图1 CS^2B^+-tree 的原理及结构

Fig. 1 The principles and structure of the CS^2B^+-tree

具体来说,移动对象 $O(\boldsymbol{x}, \boldsymbol{v}, t_u)$ 在其更新时刻 t_u 的索引值为:

$$value(O, t_u) = index\_partition \oplus x\_rep \tag{1}$$

式(1)中 *index _partition* 表示根据移动对象更新时间将其划分到的索引号(每个索引号对应一棵 CSB^+-tree),*x _rep* 表示将移动对象的位置信息利用空间填充曲线转换为一维空间上的值。⊕表示将 *x _rep* 存储到 *index _partition* 对应的树中。下面说明 *index _partition* 和 *x _rep* 这两个值的计算方法。

如图1左侧所示,假设 Δt_{mu} 为任意一个移动对象的最大更新时间间隔。首先将时间轴划分为多个 Δt_{mu},对每个 Δt_{mu},再划分为 n 个(图中 $n=2$)等长的子间隔,称之为阶段 *phase*。移动对象的任一更新时刻 t_u 映射到一个标签时间戳 $t_{lab}=\lceil t_u+\Delta t_{mu}/n \rceil$,$\lceil x \rceil$ 表示离 x 最近的未来时间戳,每个时间戳是阶段的整数倍。显然,通过这一映射,使得整个索引中包含 $n+1$ 个 CSB^+-tree,将不同时刻更新的移动对象划分到不同的索引树上。在图1中,$t_u=0$ 时刻更新的移动对象将被索引到 I_1,$0<t_u\leqslant 1/2\Delta t_{mu}$ 时刻更新的移动对象被索引到 I_2。当 $t_u=0$ 时刻更新的移动对象再次更新时,由于最大更新间隔为 Δt_{mu},所以这些移动对象被更新到 I_3 或 I_1 无效之后(因为 I_1 中索引的移动对象最多在 Δt_{mu} 时间后会全部更新)的新索引 I'_1。*index _partition* 的计算如式(2)所示。对于移动对象的位置信息和速度信息,与其他索引结构类似假设其呈线性运动,根据其速度矢量计算该移动对象在 t_{lab} 时刻的位置信息。再用典型的空间填充曲线 Hilbert 曲线将其映射到一维空间中,得到式(3)表示的 *x _rep* 值。

$$index\_partition = [t_{lab}/(\Delta t_{mu}/n)] \bmod (n+1) \tag{2}$$

$$x\_rep = hilbert\_value(\boldsymbol{x} + \boldsymbol{v} \times (t_{lab} - t_u)) \tag{3}$$

Δt_{mu} 的划分个数 n 对查询性能和 CS^{2}B-tree 存储空间都有影响。n 较大时会导致多个 CSB$^+$-tree 索引并占用更多的存储空间; n 较小时会导致较差的查询性能。因此, 在最后的实验中设置 $n = 2$。

经过上述处理, 可以对移动对象根据其更新时刻存储到不同的 CSB$^+$-tree 中, 得到如图 1 右侧所示的索引结构。在索引这些一维值时, 采用类似 CSB$^+$-tree 的缓存敏感机制。在我们设计的 CS^{2}B-tree 中, 底层采用的 CSB$^+$-tree 是一种多路平衡搜索树。它将子节点存储在给定的连续空间中, 称为节点组, 父节点仅仅记录指向第一个子节点的指针。节点组中的节点通过相对于第一个节点的偏移量来访问。中间节点存储的数据包括: 节点中 *key* 的个数, 指向第一个子节点的指针, *key* 数组及指向其右侧第一个兄弟节点的指针。叶节点则包含一组 *id* 和 *key* 的键值对, 键值对的个数以及指向兄弟节点的指针。此外, 由于空间填充曲线将每个单元格中的移动对象空间位置信息映射为相同的一维键值, 因此, 在 CS^{2}B-tree 叶节点中还存储了对应于每个键值的移动对象列表的指针, 该列表则存储移动对象实际的位置信息。

综上, 虽然 CS^{2}B-tree 通过采用 B^x-tree 的构建思想并继承 CSB$^+$-tree 的功能具备了缓存敏感性, 但是其结构和上述两种结构有以下不同:

(1) B^x-tree 将 *index_partition* 作为 x_rep 值的二进制前缀, 并最终转化为十进制值, 因此 B^x-tree 中不同时刻更新的移动对象存储在树的不同部分。而在 CS^{2}B-tree 中, 不同时刻更新的移动对象存储在不同的树中, 因此一个 x_rep 值唯一对应一个叶节点中的键值和数据列表, 为接下来的并发访问机制设计提供了可能。

(2) CS^{2}B-tree 底层采用的 CSB$^+$-tree 中, 非叶节点存储一个指向兄弟节点的指针, 为了是在 CSB$^+$-tree 中使用传统的 B^{link} 并发访问[6]。

1.2　CS^{2}B-tree 并发访问机制

下面来设计 CS^{2}B-tree 的并发访问机制。由于利用空间填充曲线和 CSB$^+$-tree 对空间位置信息进行索引时, 一个空间查询往往与多个单元格相交(2.2 节会有详细的查询算法讨论), 因而一个空间查询会对应于多个一维范围查询。在查询时, 被查询覆盖的单元格都不应允许插入操作。为此设计了 CS^{2}B-tree 的并发访问机制如表 1 所示。该并发访问机制由两层锁协议组成, 在单元格层需要包括读写两种锁, 单元格层的读操作可并发执行, 读写操作之间是互斥的关系, 访问相同索引树的写操作之间是互斥的, 两个写操作访问不同索引树时可以并发执行; 在节点层, 对节点的写操作设置了写锁。

表 1　CS^{2}B-tree 并发机制

Tab. 1　The concurrency mechanism of CS^{2}B-tree

		Cell-level		Node-level
		Write-lock	Read-lock	Write-lock
Cell-level	Write-lock	Exclusive/Compatible	Exclusive	/
	Read-lock	Exclusive	Compatible	/
Node-level	Write-lock	/	/	Exclusive

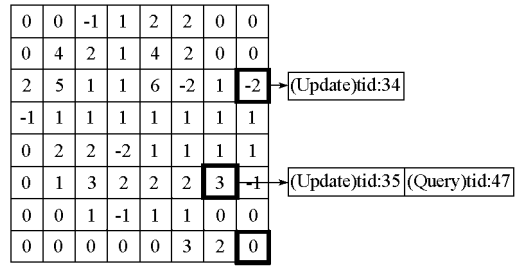


图 2　lock memo 原理

Fig. 2　The principle of lock memo

CS^{2}B-tree 并发访问机制的核心就是设计了一种称为 Lock Memo(锁备忘)的数据结构, 支持对单元格层的并发访问。

Lock Memo 的原理可用图 2 进行阐释。为了防止查询时对单元格中数据的更新操作, 需要对单元格设置读写锁。每个单元格中维护的数据结构包括: 一个查询线程计数器和一个更新线程计数器和一个等待线程队列。其中查询线程计数器以正整数来统计当前网格中查询线程的个数(读线程可以并发执行), 如图 2 中加粗的单元格中的数字“3”表明当前有 3 个线程正在对该单元格进行读操作。更新线程计数器以负整数来统计当前网格中更新线程的个数, 如图中加粗的单元格中的数字“-2”表明当前有 2 个线程正在对该单元格进行写操作。因为 CS^{2}B-tree 中存在多棵 CSB$^+$-tree, 所以对不同索引树进行写操

作的线程可以并发执行。图2中假设CS^2B-tree中有2棵索引树,因而负的最小值为-2。线程等待队列存储当前正等待对该单元格进行访问的线程,如图中箭头所指向的数据结构。由于对单元格的读写操作是互斥的,因此2个计数器不可能同时不等于0。当查询线程计数器大于0或2个线程计数器都等于0时,可启动查询线程;当2个线程计数器都等于0或更新线程计数器为-1且与到达的更新线程针对不同索引树操作时,可启动更新线程。

2 CS^2B-tree并发访问算法

本节研究基于并发访问机制的CS^2B-tree并发更新及并发预测范围查询算法。

2.1 并发更新算法

移动对象位置更新是从索引的数据列表中删除过时数据,在新的数据列表中插入新数据。需要注意的是一条移动对象的记录唯一映射在一个单元格,而一个单元格唯一映射为一个叶节点中的键值;反过来,一个叶节点存储多个键值对应于多个单元格,一个单元格对应于多个移动对象的空间位置。因此在更新操作中考虑两种情况:(1)移动对象的原位置不是原来所在单元格对应的最后一条位置数据,同时新位置所在单元格对应的键值已经存在于索引树的叶节点中。此时索引树的结构不需要修改,只需将数据列表中的数据更新。(2)移动对象新的位置在一个新的没有被索引的单元格中,或者过时数据是原单元格的最后一条数据。此时需要对索引树中的节点进行结构调整及写操作。

算法1: Location Update(*old_loc*, *new_loc*, *T*, *LM*)

Input: *old_loc*: location to be deleted, *new_loc*: location to be inserted, *T*: CS^2B-tree, *LM*: Lock Memo

Output: *T*: Updated CS^2B-tree

```
1.  c_old = x_rep(old_loc);
2.  c_new = x_rep(new_loc);
3.  n_old = T.search(c_old);
4.  n_new = T.search(c_new);
5.  T.writeLock(n_new and n_old);
6.  LM.writelock(c_old and c_new);
7.  if(c_new.size> 0)
8.     T.unWriteLock(n_new);
9.  if(c_old.size> 1 or c_old == c_new)
10.    T.unWriteLock(n_old);
11. if(c_old.size == 1 and c_old != c_new)
12.    n_old.removeEntry(c_old);
13.    if(n_old.underflow == true)
14.       n_old.merge();
15.    T.unWriteLock(n_old);
16. if(c_new.size == 0)
17.    n_new.addEntry(c_new);
18.    if(n_new.overflow == true)
19.       n_new.split();
20.    T.unWriteLock(n_new);
21. ListDelete(n_old.entry(c_old), old_loc);
22. ListInsert(n_new.entry(c_new), new_loc);
23. LM.unWriteLock(c_old and c_new);
24. return T;
```

按照上述思想给出移动对象索引更新算法如算法 1 所示。算法的输入为移动对象原位置信息, 新位置信息, 索引 CS^2B-tree, 网格锁备忘录 Lock Memo; 输出为更新后的移动对象索引。

2.2　并发预测范围查询算法

CS^2B-tree 对移动对象当前及未来位置进行索引, 支持对移动对象的预测查询处理。本小节研究基于 CS^2B-tree 及其并发访问机制的预测范围查询算法。

CS^2B-tree 的多棵 CSB^+-tree 中存储着移动对象在不同 t_{lab} 时刻的未来位置, 因而对于预测范围查询 $R_q=(\boldsymbol{x}_1, \boldsymbol{x}_2)$ (即该查询希望查找 t_q 时刻落在 R_q 范围内的移动对象) 来说, 按照移动对象的最大速度扩张到 t_{lab} 时刻才能保证不丢失可能的查询结果, 因为 t_q 时刻在 R_q 范围的移动对象不会运动出扩张后的范围 R'_q。如图 3 所示, 移动对象 o_1, o_2 在索引中的数据分别为黑点所示的位置, $t_q < t_{lab}$, 在 t_{lab} 时刻 o_1 不在 R_q 范围内而 o_2 在 R_q 范围内, 在 t_q 时刻情况相反, 因此通过查询扩张可以确保不丢失可能的查询结果。采用内存中二维网格直方图统计每个单元格中移动对象的最大运动速度, 以此为依据进行查询的扩张, 得到如下计算查询扩张的公式:

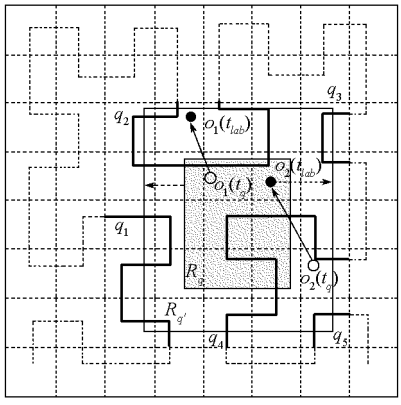


图 3　CS^2B-tree 中范围查询扩张

Fig. 3　Query Enlargement in CS^2B-tree

$$R'_q=(\boldsymbol{x}_1-\boldsymbol{max}_v\cdot|t_{lab}-t_q|, \boldsymbol{x}_2+\boldsymbol{max}_v\cdot|t_{lab}-t_q|) \quad (4)$$

将查询扩张后得到如图 3 加粗部分所示的多个一维范围, 进而在 CS^2B-tree 中进行多次一维范围查询获得可能的查询结果集, 而后计算移动对象在 t_q 时刻的位置得到最终的查询结果。

按照上述思想给出范围查询算法如算法 2 所示。算法的输入为预测范围查询及其查询时间, 索引及 Lock Memo; 输出为查询结果。

算法 2: *Predicted Range Query*(q, t_q, T, LM)

Input: q: Spatial query, t_q: time slice, T: CS^2B-tree, LM: Lock Memo

Output: *result _set*

```
1.    S = ∅; // candidate result set
2.    L = ∅; // lock set
3.    for i ←0 to n
4.      if T_i of the CS²B-tree is valid at t_q
5.        q' ← QueryEnlarge(q, t_q);
6.        calculate start and end entries i_1, …, i_2m for q';
7.        for k ← 1 to m
8.          locate leaf node containing entry i_2k-1;
9.          do
10.           LM.ReadLock(Cell_2k-1);
11.           L = L + Cell_2k-1;
12.           store candidate objects in S;
13.           follow the right pointer to the sibling node;
14.         until node with point i_2k is reached;
15.   for each object in S
16.     if the object's position at t_q is inside q
17.       add the object to the result _set;
18.   LM.unReadLock(L);
19.   return result _set;
```

3 实验结果与分析

实验的软硬件环境为: Intel Pentium Dual E2200 2.2GHz CPU, 1MB L2 高速缓存, 1GB 内存, 操作系统采用 Windows XP SP2。

实验数据的产生与文献[4]类似, 均采用人工合成的数据。移动对象在 1000km × 1000km 的空间内均匀分布, 初始位置随机分布, 运动方向随机选择, 速度在 10~ 30m/s 之间随机选择。在 CS^2B-tree 的构建中, 使用 8 阶 Hilbert 曲线进行数据映射, 3 棵 CSB^+-tree 的节点大小设为 512B[7]。为了进行吞吐量测试, 还构造了由查询和更新组成的工作负载。具体来说, 实验数据如表2 所示, 加粗值表示默认值。

表 2 实验参数设置

Tab. 2 Experimental Parameters

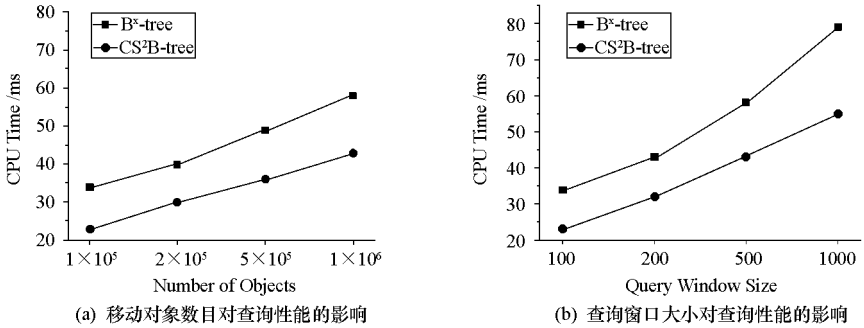
参数	值
移动对象数目	**1×10^5**, 2×10^5, 5×10^5, 1×10^6
最大更新间隔	120
最大查询预测时间间隔	**60**, 120
查询窗口大小	**100**, 200, 500, 1000
并发操作负载	**1000**, 2000
负载中更新操作的百分比	**80%**
并发访问的线程数	2, 4, **8**, 16

在实验中我们以 C++ 语言在 .Net 2005 开发环境中实现了提出的索引结构及并发访问算法。此外, 为了与其他方法比较, 我们把 B^x-tree 实现在内存中, 并利用文献[4]中的方法实现对 B^x-tree 的并发访问。

需要说明的是, 当移动对象数据量为 1×10^6 时, CS^2B-tree 占用空间约为 26MB, 这对于当前的服务器内存配置来说并不算难以承受。

3.1 索引查询性能测试

首先在不考虑并发执行的情况下, 测试了索引在不同数据量及不同查询空间范围时的查询性能, 以验证 CS^2B-tree 的缓存敏感性。如图 4 所示, 在两种测试下, CS^2B-tree 的查询性能均优于 B^x-tree, 因为缓存敏感的索引具有较好的 Cache 访问性能, 能够极大地提高查询性能。从图 4(a)中看到, 就单个查询来说, 随着移动对象数据量的增长, CS^2B-tree 和 B^x-tree 之间的查询性能差距变化并不明显。这一现象表明采用类似索引机制的两个索引具有很好的性能扩展性, 均对移动对象的数据不敏感。当查询数目较多时, CS^2B-tree 的性能优势才能逐步体现。图 4(b)表明 B^x-tree 受查询窗口影响更大, 体现为图中斜率更大。这是因为, 窗口越大, 需要进行的一维查询就越多, 这时 B^x-tree 与 CS^2B-tree 的性能差距就越明显。



(a) 移动对象数目对查询性能的影响 (b) 查询窗口大小对查询性能的影响

图 4 索引查询性能

Fig. 4 Query performance

3.2　吞吐量及响应时间测试

其次, 我们用与文献[4]中相同的方法, 采用不同线程数来模拟多用户并发执行的情况。由于底层 CSB^+-tree 具有非常好的数据更新性能, 体现为在 CS^2B-tree 中更新数据的性能比查询的性能好一个数量级。因此, 在工作负载中我们将更新操作比例控制为 80%, 避免程序执行时间过长。图 5 显示了采用不同线程数并发执行工作负载时索引访问的吞吐量及响应时间。随着线程数的增多, 并发访问需求增加, CS^2B-tree 的并发访问优势体现得更为明显。总的来说, 由实验结果可以计算出, 在不同线程并发执行情况下, CS^2B-tree 相对于 B^x-tree 来说, 吞吐量平均提高了 15.1%, 响应时间减少了 14.9%, 证明了论文提出的并发访问机制的高效性。

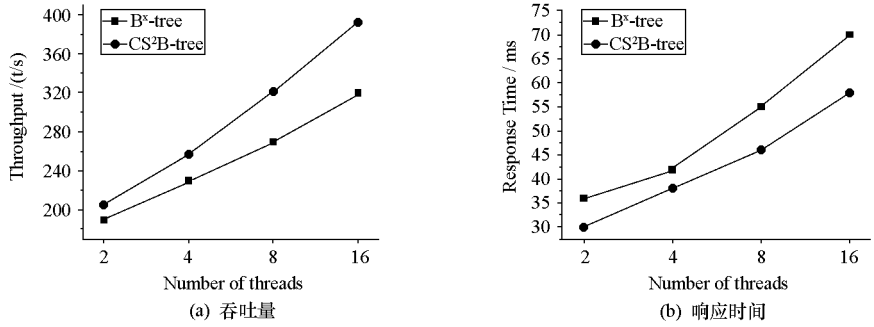


图 5　吞吐量及响应时间

Fig. 5　Throughput and the response time

4　结 论

本文针对传统移动对象当前及未来位置索引技术不能有效处理多个更新和查询操作并发执行的情况, 提出了一种支持高效并发访问的移动对象索引。该索引的缓存敏感特性提高了处理单个查询的性能, 同时, 相对于传统的 B^x-tree, 我们设计的两层并发访问机制及 Lock Memo 结构能有效地提高并发操作的吞吐量, 减少单个操作的响应时间。通过实验, 我们认为当今的服务器能够支持基于内存的移动对象索引及并发查询处理。如果需要进行移动对象数据的持久化存储以支持数据挖掘或历史信息查询等服务, 可以采用周期性将内存数据备份到磁盘上的方法。另外, 我们注意到响应时间随着并发线程数的增加而提高, 表明多线程之间存在资源冲突, 由于多核处理器平台能有效支持多线程并发执行, 因此下一步的研究工作需要基于多核处理器, 分析多线程并发访问时的 Cache 访问冲突、处理器资源使用冲突等有可能影响性能的地方, 进一步优化并发访问性能, 使得在多线程访问提高吞吐量的同时, 减少响应时间。

参 考 文 献:

[1]　刘大为, 栾华, 王珊, 等. 内存数据库在 TPC-H 负载下的处理器性能[J]. 软件学报, 2008, 19(10): 2573- 2584.

[2]　王珊, 肖艳芹, 刘大为, 等. 内存数据库关键技术研究[J]. 计算机应用, 2007, 27(10): 2353- 2357.

[3]　Johnson R, Hardavellas N, Pandis I, et al. To Share or Not to Share[C]// Proceedings of the International Conference on Very Large Databases VLDB, 2007: 351- 362.

[4]　Jensen C, Lin D, Beng C O. Query and Update Efficient B^+-tree Based Indexing of Moving Objects[C]// Proceedings of the International Conference on Very Large Databases VLDB, 2004: 768- 779.

[5]　Rao J, Ross K R. Making B^+-trees Cache Conscious in Main Memory[C]// Proceedings of the ACM SIGMOD, 2000: 475- 486.

[6]　Srinivasan V, Carey M J. Performance of B^+-tree Concurrency Control Algorithms[J]. VLDB Journal, 1993(2): 361- 406.

[7]　Hankins R A, Patel J M. Effect of Node Size on the Performance of Cache conscious B^+-tree[C]// Proceedings of the SIGMETRICS, 2003: 283- 294.