基于分布式内存数据库的移动对象全时态索引

周翔宇[1] 程春玲[1] 杨雁莹[2]
(南京邮电大学计算机学院 南京 210003)[1] (南京森林警察学院信息技术系 南京 210023)[2]

摘 要 针对现有移动索引仅对内存/磁盘两层结构进行优化,忽略了索引节点在内存中的缓存敏感性,提出一种基于分布式内存数据库的全时态索引结构DFTB^{x}树。该索引结构针对存储器Cache、内存和磁盘3层结构进行优化,根据Cache行、指令数量和TLB失配数等多个条件设计内存索引节点的大小。同时,根据磁盘数据页的大小设计历史数据迁移链节点的大小,使得Cache和内存能够一次读取索引节点和迁移链节点数据,避免多次读取数据带来的延迟。此外,构建历史数据迁移链,实现历史数据持久化,从而支持移动对象全时态索引。实验结果表明:与B^{x}树、B^{dual}树、TPR*树和STRIPES算法相比,DFTB^{x}树具有较高的查询和更新效率。

关键词 分布式内存数据库,移动对象,全时态索引,三层结构
中图法分类号 TP392 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2016.7.037

Full-temporal Index of Moving Objects Based on Distributed Main Memory Database

ZHOU Xiang-yu[1] CHENG Chun-ling[1] YANG Yan-ying[2]
(College of Computer, Nanjing University of Posts and Telecommunications, Nanjing 210003, China)[1]
(Department of Information and Technology, Nanjing Forest Police College, Nanjing 210023, China)[2]

Abstract Due to the traditional index of moving objects ignores the cache-conscious of index nodes, only the two-layer memory/disk hierarchy is optimized. Thus, this paper proposed a novel full-temporal index structure named DFTB^x-tree based on the distributed main memory database. The optimization of new index structure includes the Cache, the main memory and the hard disk. The size of index nodes is set according to many conditions such as Cache line, the number of instructions and the number of TLB mismatches. Meanwhile, the size of historical data migration nodes is designed a ccording to the size of the disk data pages. Therefore, the cache and the main memory can read the data of interior node or leaf node at a time, to avoid the delay caused by multiple data reads. Moreover, the full-temporal index of moving objects is supported by historical data which is linked through a migration chain. Compared with other algorithms, the experiment shows that DFTB^x-tree has higher efficiency in query and update operations.

Keywords Distributed main memory database, Moving objects, Full-temporal index, Three-level structure

1 引言

随着无线通信技术、GPS定位技术和移动计算技术的快速发展和普及,基于位置服务(Location-based Services, LBS)[1,2]得到了广泛的应用。在许多典型的LBS应用中,如交通调度控制、物流管理、智能导航以及个人位置服务等众多领域,需要对移动终端的位置进行追踪管理以提供相关的查询服务。因此,移动对象(Moving Objects)的数据管理引起了学术界和工业界的高度关注。

分布式内存数据库因其具有良好的可扩展性、高效的数据访问和可靠的负载均衡等特点,成为海量移动对象存储和管理的有效方式。然而,现有的移动对象索引仅针对内存和磁盘两层结构优化而来,即根据内存/磁盘上数据页大小设置索引节点大小。通过缓存数据预取技术,在内存开辟的缓冲区中存放磁盘上的数据页,使得一次I/O就可以完全载入数据节点,降低磁盘访问次数,提高数据访问效率。

为了解决CPU和内存的读写速度不匹配问题,在内存存储结构中引入高速缓存(Cache)。由内存存储结构的缓存敏感性(Cache-Conscious)[3]可知,内存索引节点大小的设置需要考虑Cache中缓存行(Cache Line)大小、指令数量和TLB失配数量等多个条件。当CPU读取数据时,Cache会从内存中读取多个缓存行大小,CPU可以直接从Cache读取数据实现对数据的访问,避免了从内存中多次读取数据带来的延迟。根据文献[4]的实验结果可知,具有缓存敏感性的索引节点大小设置为512Byte时,可取得最高的内存数据访问效率。然而,目前的操作系统数据页大小通常为4kB,因此,需要针对存储器多个层次优化索引结构,设计节点大小不同的索引。

在分布式内存数据库环境下,移动对象全时态索引结构

到稿日期:2015-06-22 返修日期:2015-09-23 本文受中央高校基本科研业务费专项资金项目(LGZD201502),国家自然科学基金(61373139,61403208)资助。
周翔宇(1990—),男,硕士生,主要研究方向为时空数据库;程春玲(1972—),女,教授,CCF会员,主要研究方向为数据管理、云计算中的资源管理和优化等,E-mail:chengcl@njupt.edu.cn;杨雁莹(1973—),女,副教授,主要研究方向为数据挖掘。

的设计需要考虑移动对象的数据划分、内存索引的缓存敏感性和历史数据访问。设计和实现高效的基于分布式内存数据库的移动对象全时态索引结构时，应该满足以下 4 个基本原则：

(1)尽可能保证空间和时间上邻近的移动对象在分布式网络节点上也是邻近的，从而降低不同网络节点间的访问频率。

(2)尽可能保证同一个网络节点中，空间和时间上邻近的移动对象存放于同一个数据页或者邻近的数据页中，从而减少磁盘 I/O 读取次数。

(3)尽可能保证同一个网络节点中，移动对象内存索引结构满足缓存敏感性，从而减少 Cache 失配数，提高内存数据访问效率。

(4)尽可能保证近期的移动对象数据存放于 Cache 和内存中，历史数据存放于持久化的磁盘中。

本文在分布式内存数据库环境下，提出了一种支持移动对象位置全时态索引的 DFTB^x 树(Distributed Full-Temporal B^x-Tree)。本文的主要贡献如下：

(1)DFTB^x 树索引设计为 3 层结构，分别为覆盖网络全局索引、节点内存索引和节点磁盘索引。

(2)DFTB^x 树的节点内存索引基于 B^x 树扩展而来，并根据缓存行、TLB 失配数等多个条件设计内存索引节点大小，使内存索引节点满足缓存敏感性，降低缓存失配率；同时根据数据页大小设计历史数据迁移链节点大小，降低磁盘 I/O 次数。

(3)构建历史数据迁移链，实现移动对象历史数据持久化，支持移动对象位置的全时态索引。

2 相关工作

2.1 移动对象索引

现有的移动对象索引技术主要以空间索引技术为基础，并对其进行扩展和改进。移动对象索引技术可以根据以下两种方式划分：基于数据划分和基于空间划分。基于数据划分的移动对象索引方案主要根据移动对象的分布来划分地理空间，其主要代表是基于 R 树的 TPR 树[5]及其变种 TPR* 树[6]。TPR 树使用时间参数化包围框来包含移动对象，其索引结构与 R 树类似。TPR 树索引节点不仅存储了移动对象的 MBR(Minimal Bounding Rectangle)，而且包含移动对象的速度矢量。TPR* 树改进了 TPR 树的动态操作算法，通过维护一个代价下降优先队列来保证插入路径选择的全局最优，保持 TPR 树 MBR 的紧致性以提高查询性能。然而，TPR 树及 TPR* 树的不足在于：首先，对于频繁更新的移动对象，其包围框出现重叠，需要依次对多个包围框进行查询，使得查询和更新效率较低；其次，上述两种索引的本质是在每个时间片上建立 R 树，以时间片为参数，存储代价较高。

基于空间划分的移动对象索引方案主要按照特定规则划分地理空间，其主要代表是基于 B+树的 B^x 树[7]、B^{dual} 树[8]、ST^{2}B 树[9]、STRIPES 索引[10]和 CS^{2}B 树[11]。B^x 树和 B^{dual} 树基于 B+树结构，利用空间填充曲线实现线性化，确保多维空间中邻近的点映射到一维空间中也是邻近的。ST^{2}B 树是基于 B^x 树的一种可自适应的索引，其针对移动对象的空间分布不均衡和不同的聚集性，采用类似 Voronoi 图的方式将空间进行不同粒度的划分以实现自我调节。STRIPES 索引利用四叉树来实现空间划分，将二维移动对象映射为四维点数据，并利用 PR 四叉树进行索引。然而，STRIPES 索引的不足在于：四叉树是非平衡的数据结构，当移动对象发生数据倾斜时，其查询和更新处理并不高效。CS^{2}B 树是一种支持高并发访问且具有缓存敏感性的索引。该索引通过结合 B^x 树和 CSB+树[12]，提出网格锁备忘录结构以支持多任务的并发访问。然而，该索引只能支持移动对象当前和未来位置查询，并不支持移动对象位置的全时态索引。

2.2 缓存敏感索引

Cache 敏感技术主要是根据处理器 Cache 参数，优化算法中的数据结构和执行过程。具有代表性的 Cache 敏感索引有：pB+树[13]、CSS 树[14]和 CSB+树[12]。pB+树基于 B+树结构，树中节点大小为缓存行的整数倍，通过硬件预取指令将节点一次装入 Cache 中。然而，这种索引的不足在于：整个节点占用多个缓存行空间，但是所有数据并不会全部使用，从而降低了 Cache 失配率。CSS 树通过对数据编号，将孩子节点保存在固定大小数组中，消除了指向子节点的指针。当 CSS 树执行更新操作时，需要重新创建整棵树，其更新代价较大。CSB+树基于 CSS 树和 B+树扩展而来，其具有近似 CSS 树的 Cache 性能和 B+树的更新操作。然而，CSB+树在节点插入和分裂时，需要为整个节点组重新分配空间，因此节点分配开销较大。

综上所述，上述的移动对象索引和内存索引通常针对内存与磁盘之间和 Cache 与内存之间两层结构优化索引，并没有针对 Cache、内存和磁盘 3 层结构优化索引。因此，本文提出针对存储器 3 层存储结构优化的移动对象全时态索引。

3 DFTB^x 树索引

3.1 DFTB^x 树索引框架

DFTB^x 树索引框架包含 3 层结构，如图 1 所示，其分别为覆盖网络全局索引、节点内存索引和节点磁盘索引。

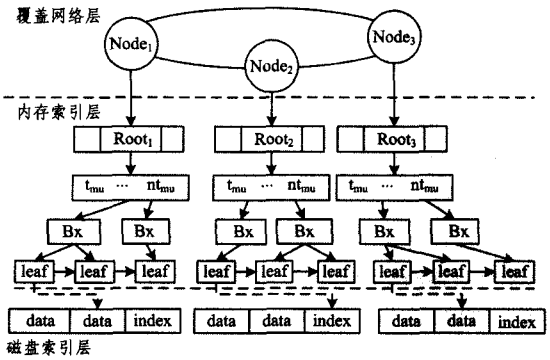

图 1 DFTB^x 树索引框架

覆盖网络全局索引，利用 Hilbert 空间填充曲线和 Chord 覆盖网络，尽可能将空间和时间上邻近的移动对象划分到同一个网络节点下的同一个数据页中，使得每个节点管理对应空间子区域内的移动对象。通过路由信息表来实现移动对象的节点定位。首先，利用 Hilbert 曲线将多维时空数据映射为一维数据；其次，利用 Chord 覆盖网络形成物理存储节点和移动对象空间范围的环形映射，保证了空间邻近的数据分布于同一个网络节点，实现海量移动数据分布式存储和定位。

节点内存索引，通过 DFTB^x 树实现对近期移动对象数据

的内存索引。首先，DFTB^x 树在 B^x 树的基础上进行改进，根据 Cache 缓存行、TLB 失配数等多个条件设计内存索引节点大小，使内存索引节点满足缓存敏感性，并根据数据页大小设计迁移链节点大小，从而降低磁盘 I/O 次数；其次，设计历史数据迁移链，将历史数据持久化于磁盘中，保证可能被频繁访问的近期移动对象存放于内存中。

节点磁盘索引，其为本地磁盘上的 B^x 树索引，主要实现对移动对象历史数据的访问。当内存数据占用空间达到内存设定的持久化阈值时，将历史数据迁移链持久化到本地磁盘上，实现了历史和近期移动对象数据的冷热分离，同时保证了移动对象全时态索引的实现。

3.2 内存索引结构

DFTB^x 树基于 B^x 树扩展而来，并针对内存节点缓存敏感性和构建历史数据迁移链两个方面优化 B^x 树。

定义 1（分布式内存索引结构） 分布式内存索引结构由每个节点内存索引 DFTB^x 树组成的森林{$DFTB_1^x$，$DFTB_2^x$，…，$DFTB_n^x$}表示。其中，任意一棵 $DFTB^x$ 树都对应一个 $Node_{id}$，$Node_{id}$ 表示覆盖网络下不同网络节点的唯一性标识符。在路由信息表中，通过 $Node_{id}$ 映射该网络节点下移动对象的 Hilbert 值空间范围信息。

定义 2（节点内存索引结构） 每个节点内存索引结构由多棵 m 阶平衡的 DFTB^x 树组成。该树满足以下特性。

(1)根节点分别包含指向 DFTB^x 树的 *Root* 指针及指向历史数据迁移链首尾的 *firstChain* 和 *lastChain* 指针。

(2)DFTB^x 树的 *Root* 节点至少有两个孩子，且 *Root* 节点的关键值由 Key_{Root} 表示：

$$Key_{Root}=(t_{lab}/(\Delta t_{mu}/n)-1) \bmod (n+1) \tag{1}$$

其中，t_{lab} 表示同一阶段中移动对象的标签时间戳，Δt_{mu} 表示任意移动对象最大更新时间间隔，n 表示最大更新时间间隔内的多个阶段。

(3)DFTB^x 树的每个内节点最多有 m 个孩子，有 $n(0<n\leqslant m)$ 个孩子的内节点必须有 n 个关键值，且关键值由 B^xValue 表示：

$$B^xValue(O,t_u)=[Key_{Root}]_2 \oplus [x\_rep]_2 \tag{2}$$

$$x\_rep=x\_value(\vec{x}+\vec{v}\cdot(t_{lab}-t_u)) \tag{3}$$

其中，$[x]_2$ 表示 x 的二进制形式，$\oplus$表示两个部分的连接，$\vec{x}$、$\vec{v}$和 t_u 分别表示移动对象的位置矢量、速度矢量和更新时刻，x_value 表示空间填充曲线对移动对象属性的计算值。

(4)所有的叶子节点均在同一层，其存储移动对象的具体数据，并且叶子节点指向其相邻的兄弟节点。

定义 3（DFTB^x 树节点组织形式） 一棵 m 阶的 DFTB^x 树不同类型节点的表现形式和分配空间大小如下。

(1)索引节点组成：索引节点由三元组〈B^xValue，*Pointer*，*histPtr*〉组成。其中，B^xValue 表示节点的关键值，*Pointer* 表示指向下一个索引或者叶子节点的指针，*histPtr* 表示指向迁移链节点的指针，用于访问持久化的历史数据。

(2)叶子节点组成：数据节点由三元组〈B^xValue，*Pointer*，*Sibling*〉组成。其中，B^xValue 表示节点的关键值，*Pointer* 表示指向叶子节点 *data* 域存放数据地址的指针，*Sibling* 表示指向其兄弟节点的指针。

(3)节点大小：内存索引内节点和叶子大小为 S_{cache}，历史数据迁移链节点大小为 S_{disk}。其中，S_{disk} 和 S_{cache} 大小关系满足：

$$S_{disk}=2^i \cdot S_{cache}, i=1,2,\cdots,n \tag{4}$$

其中，S_{cache} 表示符合缓存敏感性优化后的内存节点大小，S_{disk} 表示内存和硬盘之间基本数据传输单元数据页的大小。

3.3 数据插入和节点分裂

m 阶 DFTB^x 树的数据插入操作在叶子节点上进行，建立本地节点时，首先对叶子节点 $L1$ 分配 S_{cache} 大小的空间，并将移动对象的关键值 B^xValue 和相关信息$\vec{x}$、$\vec{v}$和 t_u 插入到 $L1$ 中。插入过程中，如果叶子节点未满，直接将该记录插入到叶子节点后结束操作。当插入的叶子节点 $L1$ 已满，DFTB^x 树执行基于关键值的节点分裂操作。节点的关键值分裂操作将小于或等于分裂关键值的数据存放到叶子节点 $L1$ 中，同时分配 S_{cache} 大小空间的叶子节点 $L2$ 和索引节点 $I1$；将大于分裂关键值的数据存放到叶子节点 $L2$ 中，并将关键值和指向新叶子节点 $L2$ 的指针构成一个索引项，插入到索引节点 $I1$ 中；同时，将叶子节点 $L1$ 中 *Sibling* 指针指向兄弟叶子节点 $L2$。索引节点中插入一个索引项的过程与上述过程相似，同样可能引起节点分裂，而且节点分裂可能向上传播，最坏情况下可能到达根节点，从而造成整个 DFTB^x 树的层数增高。

4 数据持久化

DFTB^x 树中叶子节点创建后，并不立即迁移到磁盘中，而是当内存空间占用达到设定阈值时，构建历史数据迁移链，并将历史数据迁移链持久化于磁盘中。DFTB^x 树中的历史数据迁移链通过根节点的两个指针访问：*firstChain* 和 *lastChain* 指针分别指向迁移链的初始和结束位置。

根据叶子节点关键值大小的顺序来构建历史数据迁移链。由于移动对象的关键值通过移动对象的时间戳、速度和位移等属性合并而成，因此历史数据的时间戳对应的关键值较小。历史数据迁移链构建过程中：首先，将关键值范围最小的叶子节点构成历史节点 $H1$，并将根节点的两个指针 *firstChain* 和 *lastChain* 分别指向历史节点 $H1$；其次，依次构建历史节点，*firstChain* 指针不移动，*lastChain* 指针按照 S_{disk} 空间大小向右移动，最终构建成一个按照时间戳顺序且节点大小为 S_{disk} 的迁移链；最后，将历史数据迁移链中迁移链节点按照顺序依次写入磁盘，保证内存中总是保存近期的移动对象数据。

历史数据迁移链由多棵 DFTB^x 树的叶子节点组成。由于每棵 DFTB^x 树的叶子节点中都包含指向其右兄弟节点的指针，迁移链构造过程中只需要查询相邻两棵 DFTB^x 树的叶子节点，并将前一棵树的最右叶子节点的指针指向后一棵树的最左叶子节点。因此，构建历史数据迁移链的时间复杂度为 $O(2\log_m N)$，其中 m 表示 DFTB^x 树的阶数，N 表示 DFTB^x 树所有的数据项。此外，迁移链构造过程中，迁移链节点仍为每棵 DFTB^x 树的叶子节点，并不需要分配额外的存储空间来存放迁移链，只需要分配两个指针空间用于存放迁移链的首尾指针。因此，构建历史数据迁移链的空间复杂度为 $O(1)$。

当内存索引节点需要访问磁盘中的历史数据时，可以通过索引节点的 *histPtr* 指针访问。迁移链构成如图 2 所示，其中带箭头实线表示 *Pointer* 和 *Sibling* 指针，带箭头虚线表示 *firstChain* 和 *lastChain* 指针，带箭头点线表示 *histPtr* 指针。

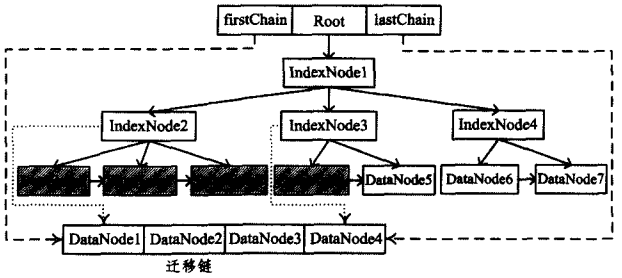

图 2　节点中的迁移链构建

5　查询处理及算法

5.1　范围查询

根据当前移动对象位置信息，查询该位置指定范围内所有的移动对象。其查询过程如下：

(1)根据移动对象当前位置和查询时间，计算该移动对象的关键值，并根据路由信息表返回符合该关键值的网络节点 N_{id}。若移动对象当前位置的周边范围超过节点管理的空间范围，需要根据节点路由表信息返回该节点的前驱和后继节点。

(2)根据返回的网络节点 N_{id}，利用节点本地 DFTB^x 树索引中扩展窗口查询方式，依据最大和最小速度矢量进行扩展查询移动对象更新时刻位置。

(3)根据移动对象位置和查询范围，返回扩展窗口内的不同移动对象 id，其算法伪代码如表 1 所列。

表 1　范围查询算法伪代码

```
Algorithm1 Range-Query(O,tq,q)
输入:移动对象 O、当前查询时间 tq 和查询范围 q
输出:返回移动对象 O 范围 q 内所有移动对象集合
1. BxValue(O,tq);//根据移动对象当前的位置和查询时间计算其关键值
2. return (x_rep+q<Nid? Nid:Collection(Nid));//返回该关键值周边范围
   的网络节点;
3. for(B+:Nid.DFTBx){//对每个网络节点中的索引子树进行查询;
4.     if(tq<tlab) //判定查询时间和标签时间戳之间的关系;
5.         Collection.add(Enlarge(O(tlab-tq)));//返回扩展窗口查询
           后的移动对象结果集;
6.     else
7.         Collection.add(Enlarge(O(tq-tlab)));
8. }
9. for(O:Collection(O(tq))){//遍历返回的移动对象集合;
10.    if(O(tq)∈q)//判定返回的移动对象是否在扩展窗口 q 中;
11.        Collection.add(O(tq));
12. retrun Collection(O(tq));//返回查询范围 q 内的移动对象集合;
```

5.2　KNN 查询

根据当前移动对象位置信息，查询该位置周边 K 个最邻近的移动对象。其查询过程如下。

(1)根据移动对象当前位置和查询时间计算该移动对象的关键值，并根据当前位置构建查询矩形，该矩形以当前位置 q 为圆心，以 $d=2*r=2*D_k/k$ 为边长，其中 $D_k=\frac{2}{\sqrt{\pi}}[1-\sqrt{1-\sqrt{k/N}}]$。

(2)根据路由信息表返回符合该关键值的网络节点 N_{id}。根据返回的网络节点 N_{id}，利用节点本地 DFTB^x 树索引中扩展窗口查询方式，依次扩展查询矩形大小，直到查询对象满足 K 个。

(3)返回扩展窗口内的 K 个不同移动对象 id，其算法伪代码如表 2 所列。

表 2　KNN 查询算法伪代码

```
Algorithm2 KNN-Query(O,tq,K)
输入:当前移动对象 O、查询时间 tq 和 K 值
输出:返回移动对象 O 邻近 K 个移动对象
1. BxValue(O,tq);//根据移动对象当前的位置信息计算其关键值
2. Rq1(q,d);//构建查询矩形,该矩形以 q 为圆心,d 为边长;
3. R'q1=Enlarge(Rq1,tq);//根据查询矩形构建其扩展查询矩形;
4. return Nid(KeyRange(x_rep));//根据路由信息表,返回该关键值周边的
   网络节点;
5. flag=true;
6. while(flag){
7.     for(B+:DFTBX){//对网络节点中的索引子树进行查询;
8.         if(i=1)
9.             Collection.add(Rq1);//返回扩展窗口查询后的移动对象
               集合;
10.        else
11.            Collection.add(R'q1-Rq1);
12.        if(Collection.count>=k)
13.            flag=false;
14.        else{//扩展矩形的边长 r 继续扩展,继续下一次矩形查询
15.            i++;Rqi=Enlarge(Rqi-1,rq);R'q1=Enlarge(Rqi,tq);
16.        }
17.    }
18.}
19. return Collection(O(tq));
```

5.3　全时态轨迹查询

本文移动对象的全时态轨迹模型通过不同更新时刻的移动对象线性插值模型来表示，即每个移动对象都是通过固定时间采样，采用直线段插值方式表示。移动对象轨迹可以表示为：$MOT=(m_1,m_2,\cdots,m_n)$，其中不同采样时刻移动对象可以表示为 $m_i=(oid,\vec{x}_i,\vec{v}_i,t_u)$。因此全时轨迹查询通过不同采样时刻移动对象的点查询来表示，即移动对象点查询集。其算法伪代码如表 3 所列。

表 3　全时态轨迹查询算法伪代码

```
Algorithm3 Trajectory-Query(O,t1,tq)
输入:移动对象 O 和查询时间范围[t1,tq]
输出:返回查询时间范围内不同时刻的移动对象 O
1. for(i=0;i<Nnumber;i++){//对每个网络节点进行遍历查询移动对象;
2.     if(t1>tlab)//根据时间下限查找内存中的移动对象数据;
3.         for(i=0;t1<tu+iΔtmu<tq;i++)//遍历不同时间戳的移动对象;
4.             Collection.add(lookingAt(oid,tu));
5.     else//查找持久化于硬盘上的迁移链中移动对象数据;
6.         Collection.add(searchHistLink(t1,tlab));
7. }
8. return Collection(O(tq));
```

6　实验结果与分析

6.1　实验内容与设置

本文实验采用文献[15]的数据集和评价指标。该数据集由数据生成器根据设定的参数生成空间均匀分布的数据。评价指标主要包括范围查询时间、范围查询 I/O 次数、更新时间、更新 I/O 次数和索引空间占用率。由于该数据集生成器和评价指标被相关的移动对象索引实验大量采用，形成一系列标杆测试数据，因此，具有一定的对比意义。

实验的软硬件环境为：采用分布式内存数据库 Redis3.0.1，该集群包含 5 个节点，操作系统为 Linux CentOS，以 Java 为编程语言，Jedis 为数据库访问接口，Eclipse 为开发环境。移动对象索引对比算法采用 C++语言在 Visual Studio2005 开

发环境中实现。

本实验模拟了 $1000 * 1000m^2$ 二维空间内移动对象的运动。在该运动空间中，移动对象的初始位置、速度和方向随机选取。在 $DFTB^x$ 树的构建中，采用 8 阶 Hilbert 曲线实现数据降维，设定移动对象的查询时间参数和更新频率参数为固定值，从而能够客观地比较相同时间点下的查询和更新代价。$DFTB^x$ 树的内存索引节点和历史数据迁移链节点大小分别设置为 512B[4] 和 4kB，其中内存数据持久化操作采用 Redis 的 AOF(Append-Only File)方式，内存持久化阈值采用 Redis 默认值。本实验测试结果为 20 轮测试的平均值，其他相关实验参数的设置如表 4 所列。

表 4　实验参数设置

参数	参数值
数据域	$1000 * 1000m^2$
移动对象数据集大小	10M、20M、50M、100M、200M、500M
范围查询窗口大小	100m
KNN 查询 K 值	10
磁盘索引开辟内存缓冲区大小	50 * 4kB
历史数据迁移链节点大小	4kB
内存索引节点大小	512B
内存数据库开辟空间大小	512MB

6.2　索引范围查询性能测试

本节实验比较了不同数据集下，$DFTB^x$ 树与 B^x 树、B^{dual} 树、TPR^* 树和 STRIPES 的范围查询性能，其评价指标分别为范围查询时间和范围查询 I/O 次数。首先，随机选择一个移动对象，并查询其周边范围 100m 内所有移动对象。其次，根据系统执行查询的初始和结束时间来计算范围查询平均时间。范围查询 I/O 次数表示范围查询时磁盘上数据页交换的次数。

如图 3 和图 4 所示，$DFTB^x$ 树具有较低的范围查询时间和最低的范围查询 I/O 次数。在数据集较小的情况下，内存数据库能够将数据全部存放于内存中。与硬盘磁头访问扇区方式相比，内存数据库通过内存寻址方式访问内存数据，消除了磁盘 I/O 多次读写，提高了数据访问效率。其他对比算法中，TPR^* 树具有最低的范围查询时间和较低的范围查询 I/O 次数。STRIPES 和 TPR^* 树索引有相近的范围查询时间，但是 STRIPES 索引的范围查询 I/O 次数最高。随着移动对象数量增多，STRIPES 中不平衡的四叉树索引结构导致树的高度越来越高，更新需要通过很长的路径才能访问到节点。B^x 树和 B^{dual} 树的范围查询时间较差。由于 B^x 树和 B^{dual} 树都是通过计算空间填充曲线值来查询相应位置的移动对象，并且 B^{dual} 树还需要消耗更多的时间计算 MOR，因此 B^x 树的范围查询时间低于 B^{dual} 树。

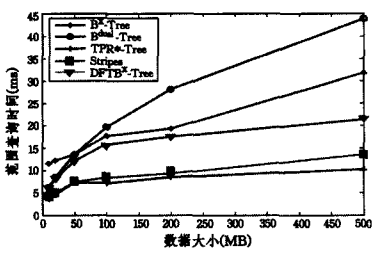

图 3　范围查询时间

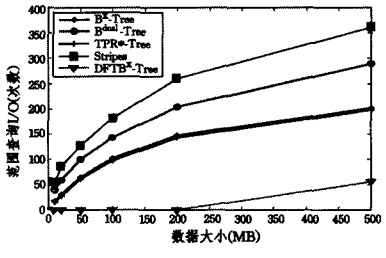

图 4　范围查询 I/O 次数

6.3　索引空间大小测试

本节实验比较了不同数据集下，$DFTB^x$ 树与 B^x 树、B^{dual} 树、TPR^* 树和 STRIPES 的索引空间大小。索引占用空间大小的计算过程如下：首先，遍历存放在内存缓冲区内的索引结构，并统计内节点和叶子结点数目(该叶子节点仅包含指向数据的指针和相关信息，并不包含具体数据)；其次，根据内存分配的节点大小，计算不同索引结构中索引节点分配的字节数。

如图 5 所示，STRIPES 索引占用的存储空间最大。这是由其不平衡的四叉树结构造成的。其他几种索引空间的大小近似。与 B^x 树和 B^{dual} 树相比，$DFTB^x$ 树索引空间占用较大。由于 $DFTB^x$ 树索引 Cache 中索引节点 S_{cache} 相对磁盘中索引节点 S_{disk} 较小，使得 $DFTB^x$ 树的高度较高，从而索引节点占用空间相对较大。

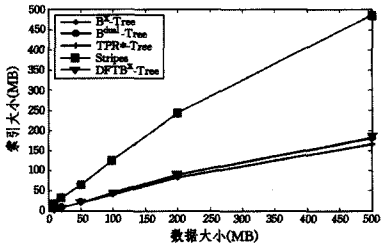

图 5　索引空间大小

6.4　索引更新性能测试

本节实验比较了不同数据集下，$DFTB^x$ 树与 B^x 树、B^{dual} 树、TPR^* 树和 STRIPES 的更新性能，其评价指标分别为更新时间和更新 I/O 次数。其中，更新时间表示多个移动对象更新的平均响应时间，更新 I/O 次数表示更新移动对象时磁盘上顺序读或者顺序写数据块的次数之和。

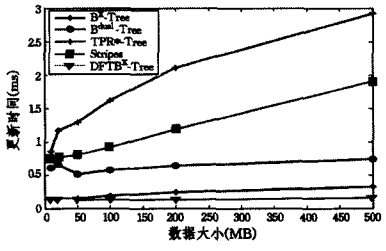

图 6　更新时间

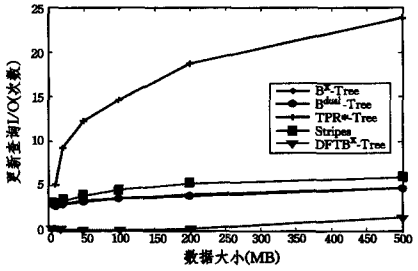

图 7　更新 I/O 次数

(下转第 216 页)

[13] Gendreau M, Potvin J Y. Handbook of Metaheuristics [M]. Springer, 2010: 475-513

[14] Hansen P, Mladenović N. Variable neighborhood search for the p-median[J]. Location Science, 1997, 5(4): 207-226

[15] Liu Qun. Chinese Lexical Analysis and Syntactic Parsing Technology Overview [C]// sWcL2002 Lecture. Beijing, 2002(in Chinese)
刘群. 汉语词法分析和句法分析技术综述[C]// 第 1 届学生计算语言学研讨会 (sWcL2002) 专题讲座. 北京, 2002

[16] Salehipour A, Sörensen K, Goos P, et al. Efficient GRASP+ VND and GRASP+ VNS metaheuristics for the traveling repairman problem[J]. 4OR, 2011, 9(2): 189-209

[17] Villegas J G, Prins C, Prodhon C, et al. GRASP/VND and multi-start evolutionary local search for the single truck and trailer routing problem with satellite depots[J]. Engineering Applications of Artificial Intelligence, 2010, 23(5): 780-794

[18] Schittekat P, Kinable J, Sörensen K, et al. A metaheuristic for the school bus routing problem with bus stop selection[J]. European Journal of Operational Research, 2013, 229(2): 518-528

[19] Wang Ya. Research on Method of Acquiring Commonsense Knowledge Based on Semantic Taxonomy [D]. Nanning: Guangxi Normal University, 2015(in Chinese)
王亚. 基于语义分类的常识知识获取方法研究[D]. 南宁: 广西师范大学, 2015

[20] Lu Chuan, Hou Rui-long, Dong Li-Ping. Basic Words of Modern Chinese[J]. Chinese Teaching in the World, 2000, 4(1): 11-24 (in Chinese)
鲁川, 侯瑞隆, 董丽萍. 现代汉语基本句模[J]. 世界汉语教学, 2000, 4(1): 11-24

[21] Yao X, Thill J C. How Far Is Too Far? -A Statistical Approach to Context-contingent Proximity Modeling[J]. Transactions in GIS, 2005, 9(2): 157-178

[22] Yao X, Thill J C. Neurofuzzy modeling of context-contingent proximity relations[J]. Geographical Analysis, 2007, 39(2): 169-194

[23] Vlassis N. A concise introduction to multiagent systems and distributed artificial intelligence[J]. Synthesis Lectures on Artificial Intelligence and Machine Learning, 2007, 1(1): 1-71

(上接第 207 页)

由图 6 和图 7 可知, DFTB^x 树具有最低的更新时间和更新 I/O 次数。由于内存通过内存寻址方式访问数据, 因此内存数据更新速度快于磁盘数据更新速度。对比算法中, TPR* 树的更新性能最差。随着移动对象数量增加, 移动对象的频繁更新使得 TPR* 树中 MBR 出现覆盖重叠, 导致存在多个查询路径, 降低了更新性能。与 B^x 树和 B^{dual} 树相比, STRIPES 具有较高的更新时间和更新 I/O 次数。由于 B^x 树和 B^{dual} 树采用B+树作为基础索引, B+树更新时, 只需要单路径查询, 因此, B^x 树和 B^{dual} 树取得了较好的更新 I/O 性能。

结束语 本文提出一种基于分布式内存数据库的移动对象全时态索引, 针对存储器 3 层结构, 优化索引节点大小。同时, 设计历史数据迁移链并持久化, 从而支持移动对象全时态索引。实验表明, 与 B^x 树、B^{dual} 树、TPR* 树和 STRIPES 算法相比, DFTB^x 树具有较高的查询和更新效率。本文主要研究的是不受空间约束环境下的移动对象索引。现实生活中, 人们真实的运动场景往往局限于道路交通网络中。因此, 下一步的研究工作主要考虑基于道路交通的移动对象时空索引和查询技术。

参 考 文 献

[1] Qiu P, Zhang J, Zeng J. Study on the mobile LBS development model[C]// 2012 IEEE International Conference on Computer Science & Service System (CSSS). 2012: 1070-1074

[2] Rao B, Minakakis L. Evolution of mobile location-based services [J]. Communications of the ACM, 2003, 46(12): 61-65

[3] Sun L M, Song B Y, Yu Y X, et al. Cache-conscious index mechanism for main-memory databases [J]. Wuhan University Journal of Natural Sciences, 2006, 2(1): 309-312

[4] Hankins R A. Effect of node size on the performance of cache-conscious B+-tree[J]. ACM SIGMETRICS Performance Evaluation Review, 2003, 31(1): 283-294

[5] Saltenis S, Jensen C S, Leutenegger S T, et al. Indexing the positions of continuously moving objects[C]// Proceedings of ACM SIGMOD International Conference on Management Data. 2000: 331-342

[6] Tao Y, Papadias D, Sun J. The TPR*-Tree: An optimized spatio-temporal access method for predictive queries[C]// Proceedings of the 29th International Journal on Very Large Data Bases (VLDB). 2003: 790-801

[7] Jensen C S, Lin D, Ooi B C. Query and update efficient B+-tree based indexing of moving objects[C]// Proceedings of the Thirtieth International Journal on Very Large Data Bases(VLDB). 2004: 768-779

[8] Man L Y, Tao Y, Mamoulis N. The B dual-tree: indexing moving objects by space filling curves in the dual space[J]. The VLDB Journal, 2008, 17(3): 379-400

[9] Chen S, Ooi B C, Tan K L, et al. ST^{2}B-tree: A self-tunable spatio-temporal B+-tree index for moving objects[C]// Proceedings of the ACM SIGMOD International Conference on Management of Data. 2008: 29-42

[10] Patel J M, Chen Y, Chakka V P. STRIPES: An efficient index for predicted trajectories[C]// Proceedings of the ACM SIGMOD International Conference on Management of Data. 2004: 635-646

[11] Zhao L, Chen L, Jing N, et al. An efficient moving object index that supports concurrent access[J]. Journal of National University of Defense Technology, 2010, 32(3): 53-59(in Chinese)
赵亮, 陈荦, 景宁, 等. 一种支持高效并发访问的移动对象索引[J]. 国防科技大学学报, 2010, 32(3): 53-59

[12] Rao J, Ross K A. Making B+- Trees Cache conscious in main memory[C]// Proceedings of the ACM SIGMOD International Conference on Management of Data. 2000: 475-486

[13] Chen S, Todd G P B. Improving index performance through prefetching[J]. ACM SIGMOD Record, 2001, 30(2): 235-246

[14] Rao J, Ross K A. Cache conscious indexing for decision-support in main memory[C]// Proceedings of the International Journal on Very Large Data Bases(VLDB). 1999: 78-89

[15] Chen S, Jensen C S, Lin D. A benchmark for evaluating moving objects indexes[C]// Proceedings of the International Journal on Very Large Data Bases(PVLDB). 2008: 23-28