ISSN 1004-9037,CODEN SCYCE4
Journal of Data Acquisition and Processing Vol. 38,No. 5,Sep. 2023,pp. 1035－1047
DOI:10. 16337/j. 1004-9037. 2023. 05. 003

http://sjcj. nuaa. edu. cn
E-mail:sjcj@nuaa. edu. cn
Tel/Fax:＋86-025-84892742

面向轨迹数据查询优化的缓存机制研究

金　鑫，吴冰雅，许建秋

（南京航空航天大学计算机科学与技术学院，南京 211106）

摘　要：移动对象数据库（Moving object database，MOD）管理随着时间不断改变位置的空间对象，其已经在广泛的应用中被研究。尽管索引和查询算法等许多与MOD相关的技术已经被提出，但缓存管理仍然被忽略，这对于数据库性能至关重要。传统的缓存方法忽视了数据本身的时空特性，无法实现良好的性能。本文提出从缓存层面充分挖掘轨迹数据的查询性能，首先针对轨迹数据特有的存储结构和读写过程，设计了适合MOD的缓存访问机制；然后对于MOD系统中缺少与应用场景和访问模式相关的缓存策略问题，定制了缓存替换方法；最后开发了缓存管理工具MOCache，以可视化的形式动态跟踪查询语句结束后缓存状态的变化。通过实验表明本文提出的缓存替换策略与传统的策略相比，命中率提高到76.56%，缩短了查询时间，并且使用缓存工具监控历史状态信息，能够更加全面地反馈和分析性能问题。

关键词：轨迹数据；缓存访问机制；缓存替换策略；缓存监控；查询优化

中图分类号：TP393.0　　**文献标志码**：A

Research on Cache Mechanism for Query Optimization of Trajectory Data

JIN Xin, WU Bingya, XU Jianqiu

(College of Computer Science and Technology, Nanjing University of Aeronautics & Astronautics, Nanjing 211106, China)

Abstract: Moving objects database (MOD) manage spatial objects that continuously change their locations over time and have been studied in a wide range of applications. Although a number of relevante techniques have been proposed such as indexing and query algorithms, cache management in MOD has been ignored. This is essentially important for database performance. Traditional cache methods ignore the spatial-temporal characteristics of data and cannot achieve good performance. This paper proposes to fully exploit the query performance of trajectory data from the cache level. Firstly, based on the unique storage structure and read/write process of trajectory data, a cache access mechanism suitable for MOD is designed. Then, due to the lack of cache policies related to application scenarios and access modes in MOD, a cache replacement strategy is customized. Finally, a cache management tool MOCache for trajectory data is implemented. By utilizing MOCache, dynamic tracking of cache state changes is visualized after each query statement. Compared with traditional algorithms, the proposed cache replacement strategy improves the hit ratio to 76.56% and reduces query time, and using the cache tool to monitor historical state information can facilitate comprehensive feedback and analyze performance problems.

基金项目：国家自然科学基金（61972198）。
收稿日期：2022-05-24；**修订日期**：2022-09-15

Key words: trajectory data; cache access mechanism; cache replacement strategy; cache monitoring; query optimization

引　言

过去20多年来,轨迹数据管理系统研究十分活跃,轨迹数据库不仅可对原始数据进行数据格式转换、面向多种查询处理操作,还能灵活拓展自定义的数据类型和功能模块。很多原型系统被开发并广泛应用,例如TrajStore[1]、SharkDB[2]和DITA[3],这些系统通过有限次数的磁盘访问高效检索特定时空区域中的所有数据,并且利用不同的分区方法来解决数据局部性问题。为了进一步优化系统性能,多种技术方向被提出[4],包括数据存储方式、索引结构优化、数据划分方法和查询算法设计等。然而在现存的轨迹管理系统中与底层缓存技术相关的潜在性能还没有得到充分挖掘,很少有系统会根据轨迹数据特有的时空访问特点分析和改进缓存机制。在移动对象数据库(Moving object database,MOD)管理系统中,针对增删改查的业务场景,有80%是关于查询,仅20%是数据变更操作,轨迹查询紧紧依赖着时间和空间属性,因此如何充分利用时空访问特性设计缓存机制使其能够在极其有限的存储容量中长期保留高价值的数据是非常值得研究的问题。由于数据规模的急剧增大,查询处理需求激增,当访问大规模数据时,一个合适的缓存管理机制可以大大减少磁盘访问开销,缩短查询响应时间,提高数据库性能。

缓存是数据库的基础组建,缓存管理通常包括:缓存内容、缓存时间、缓存位置和缓存方式等,这与数据特征和访问模式密切相关[5]。轨迹在系统中的不定长存储是一种独特的存储方式,存储管理各层级之间存在继承关系,数据在读写过程中的流动走向也遵循一定规则有序进行。在设计缓存访问机制时,应当首先考虑轨迹数据的查询应用场景,再充分利用MOD中现有的基础架构,探索出缓存层的系统接口,针对性地设计缓存组织结构和读写访问流程。

为了最大化缓存中查询数据的重用性,学术界提出并衍生了多种缓存替换策略[6-9]。从本质而言,这些策略旨在解决数据访问预测问题,目的是将价值高的对象保留在内存中而在空间不足时执行替换来加快访问速度,充分利用了数据的局部性原则。在查询语句执行过程中,当缓存空间溢出时,大多数MOD采用的是较传统的缓存替换算法,由于轨迹数据具有变化频繁、规模大、时空范围广等特点,传统算法无法考虑移动对象的时空特性、应用场景的匹配性以及不能顾及大规模数据批量操作时系统的性能,只是实现了单纯的换入换出,导致用户查询的等待时间较长,在大部分实际应用中命中率不高,存在大量的缓存污染情况。

在用户使用数据库或系统底层开发过程中,缓存管理监控工具十分重要。实际应用时经常需要定位系统出现的问题,而缓存模块基于内存开发,对用户而言相当于一个黑匣子,不利于及时反馈信息,因此应当开发缓存工具以提供反馈从而进行正确的系统调整。在轨迹数据库中,缓存管理监控工具可以可视化缓存状态变化,控制资源利用,提高查询效率。

本文主要从轨迹管理系统的缓存层出发,围绕3个技术方向展开研究和实践:轨迹数据的缓存访问机制、缓存替换策略和缓存管理监控工具,主要贡献总结如下:

(1)结合轨迹数据的存储特点设计了灵活的块级缓存结构,为了能够在缓存中高效查找以及动态管理缓存空间,实现了哈希页面检索结合双向链表的技术进行缓存块组织。

(2)提出一种在最近最少使用(Least recently used, LRU)算法基础上结合三维网格访问热度的缓存替换策略(LRU based on cell heat, LRU-CH),继承LRU算法优点的同时,充分挖掘了移动对象的时空特性,实验结果表明LRU-CH的缓存命中率最高,达到76.56%,减少了查询时间开销。

(3)为了高效地向用户和研究人员反馈缓存内部信息,开发了缓存管理监控工具MOCache,在查询结束时监视和收集缓存状态变化。

1　相关研究现状

1.1　轨迹数据管理系统

轨迹数据库的最基本任务是研究了一类支持时空对象的抽象数据类型(Abstract data type, ADT),如:移动点和移动区域,使其可以在系统中作为基本数据类型被使用。移动点在采集频率很高时近似于连续运动,但在数据库中仍然是离散形式,移动点按时间排序组成一条首尾相连的轨迹进行存储。作为轨迹数据管理领域最著名的研究学者,Güting等[10]设计了一个支持空间数据类型及其操作的数据库原型系统SECONDO,实现点、线、区域等数据类型,采用高斯-克吕格投影将经纬度坐标投影到笛卡尔坐标系中,可在图形用户界面中导入道路网将轨迹的运动过程动态展现出来。用户可以利用系统定义好的接口模块,自定义开发抽象数据模型包来扩展关系或对象关系模型,添加所需的数据库操作。

1.2　轨迹数据查询方法

不同于关系型数据库仅对数据本身的增删改查,MOD的查询更倾向于提供分析数据的功能[11],每一类查询都会聚集很多函数和操作符。时空范围查询是MOD最常用的查询方式,查询在某特定时间通过某空间的移动对象,可细分为时间点查询、时间段查询、空间位置点查询、空间区域查询和时空组合查询。在处理查询逻辑时,时空范围可以表示成二维或者三维的边界框,再计算出与之相交的移动轨迹。Xu等[12]实现了多属性轨迹的范围查询,每个对象的信息由时间戳位置和描述性属性组成,如查询在$[t_1, t_2]$时间段内是否有银色的宝马经过中心广场。将时空范围结合带有语义标签的属性进行查询更能明确表达对象活动轨迹。Lan等[13]研究的基于成本的优化器几乎被当前所有数据库系统所采用,其通过引入计划枚举算法来查找(子)计划,并使用成本模型选择成本最低的计划。

1.3　缓存技术优化方法

1.3.1　缓存替换策略

引入缓存是提高系统性能有效且常见的方式,通过设计良好的数据分块、预取、缓存替换等方法提高对缓存内容的命中率。由于缓存空间有限以及不同系统的数据访问模式不同,同一种缓存策略很难在不同的数据访问模式下均取得满意性能[7],因此学术界提出多种缓存策略以适应不同业务需求。

基于访问时间的替换。此类算法按各缓存项最近访问情况来组织缓存结构以决定替换对象,也称为时间上的邻近性,如经典的LRU策略[8],但当应用程序的工作集不适合缓存时,LRU会出现页面置换频繁的抖动现象而导致性能突降。为克服LRU的性能弊端,学术界针对不同的应用场景研究出了很多变体,如多倍最近最少使用(A multiple LRU, AM-LRU)[9],基于用户行为的最近最少使用(LRU based on user behaviour, LRU-UB)[13],基于计数过滤器的最近最少使用(LRU based on counting cuckoo filter, CCF-LRU)[14]等,这些算法主要从3个技术方向改进:将LRU的时间邻近性与其他技术结合;当工作集不适合缓存时采取一些手段保护部分工作集不被逐出,以避免抖动;按照访问模式划分为不同类型,对每一类进行不同的处理。文献[15]基于Belady算法改进缓存替换,解释了如何通过利用过去的缓存历史访问来学习Belady的算法,以预测未来的缓存替换决策。该文提出的新缓存替换策略由两部分组成,一是重构了Belady对过去缓存访问的最佳解决方案,二是实现一个预测器,它可以学习计算机系统过去的行为,以决定对未来负载的驱逐方式。此算法在满足历史与未来指令有密切联系时效果最佳。

基于访问频率的替换。这类算法用缓存项的被访问频率来组织缓存,通过维护每个缓存项的频率计数器并排序进而决定替换。最不经常使用(Least frequently used, LFU)算法是基于访问频率替换的传统算法,围绕LFU衍生了多种策略,著名的LRU-K和双队列算法(Two queues algorithm, 2Q)也是借助于访问频率思想的典范,LRU-K维护历史队列和缓存队列,历史队列保存着每次访问的页面,当页面访问次数达到了K次则调到缓存队列,若历史队列满了会根据一定的策略淘汰,若缓存队列溢出则淘汰第K次访问距离现在最久的页面。2Q算法具有恒定的时间开销,历史队列是采用先进先出(First input first output, FIFO)方式,更适用真实的场景,而且结构灵活无需过多调整。

混合策略。通过兼顾访问时间和频率,混合策略会在几个预先确定好的策略间进行动态选择,使得数据模式在变化时缓存策略仍有较好性能。多数此类算法具有一个可调或自适应参数,通过该参数的调节使缓存策略在基于访问时间与频率间取得平衡。

1.3.2 *缓存工具*

缓存属于系统底层应用,缓存局部优化是实现高性能的关键,这种优化的先决条件是获取缓存访问行为的性能数据。与应用层软件不同,用户和系统开发人员无法直接观察缓存状态,在需要依赖缓存性能数据来优化应用程序时显得尤为不便。因此,为了提高开发效率减少重复性工作,学术界研究了显性描述缓存内部信息的方法,如缓存管理、缓存监控和缓存可视化工具[16],可用于把控性能变化、检测信道攻击、预测系统开销等方向。

轻量级缓存框架CacheOptimizer[17]的出现是为了帮助开发人员提高以数据库为中心的Web应用程序的性能,首先利用详细的Web日志在工作负载和数据库访问之间创建映射关系,再使用有颜色的Petri Net找到最佳缓存配置,再自动将缓存配置添加到应用程序中,最后改进了内存占用和吞吐量开销,与默认缓存配置相比,CacheOptimizer带来了显著收益。CacheVisor[18]是一个用于可视化多核和多线程处理器中共享缓存的工具包,执行处理器模拟生成内存访问轨迹,以图形表示方式描述并发使用缓存的应用程序之间的缓存共享动态行为,以便设计出更优的缓存共享算法。名为CacheAudit[19]的决策支持工具将缓存配置条件作为输入,并跟踪命中和未命中情况以及执行时间,以基于观察缓存状态得到系统安全保证。CacheMAsT[20]旨在可视化网络内缓存管理策略的配置和性能,以高效地控制资源利用率。

这些缓存工具都为所处应用领域带来了显著的效果,但是,在现有的轨迹数据库中,缓存对研究人员而言仍是一个完全未知的部分,缓存性能对查询访问操作的影响优劣与否也不确定。因此,有必要在轨迹数据库系统设计一个工具跟踪每个查询语句的缓存状态。

2 轨迹数据的定义及存储

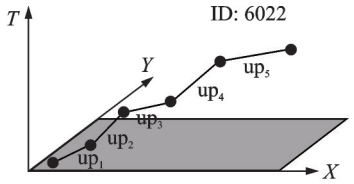
定义1(轨迹单元UPoint) 单个轨迹点称为Point,每两个时间相邻的Point以左闭右开区间首尾相连组成一个UPoint(Unit point)。存在两个轨迹点$\text{point}_1[\text{Longitude}_1, \text{Latitude}_1, \text{Time}_1]$,$\text{point}_2[\text{Longitude}_2, \text{Latitude}_2, \text{Time}_2]$,且$\text{Time}_1 < \text{Time}_2$。轨迹单元$\text{UPoint}_1$由$\text{point}_1$和$\text{point}_2$连接组成,$\text{UPoint}_1 = [\text{point}_1, \text{point}_2)$。

定义2(移动对象MPoint) 移动对象MPoint(Moving point)是一段由多个UPoint按照时间上不相交顺序连接起来而形成的完整运动轨迹。

$\text{MPoint} = \{ <\text{UPoint}_1>, <\text{UPoint}_2>, \cdots, <\text{UPoint}_N> \}$,其中$\text{UPoint}_1 = [\text{point}_1, \text{point}_2)$,$\text{UPoint}_2 = [\text{point}_2, \text{point}_3)$,$\text{UPoint}_N = [\text{point}_N, \text{point}_{N+1})$。

为了改善存储管理系统中包含大型对象时的存储过程,设计了Flob存储系统中规模较大的对象,

通常是持久化数据。Flob决定一个对象是否必须存储在磁盘上,以及如何将它分布在不同的记录文件中,Flob的设计巧妙地减少了大型对象的管理复杂性。变长数组DbArray是派生于Flob的子类,数组容量会随着数据动态增长,可使用索引下标随机访问数组中元素。DbArray最重要的应用是存储轨迹,每个索引下标对应存储的就是一个UPoint。图1以ID为6022的部分车辆轨迹为例展示了移动对象存储方式,其中$up_1 \sim up_5$是彼此相连的UPoint。在数据库中如何唯一地标识磁盘上读取的数据是一个挑战,唯一性是为了跟踪数据访问历史。这个目标可以通过数据库文件及记录中的偏移量来实现。MOD存储数据的基本单位是Record(记录),偏移量是数据项在Record中的起始位置,给定偏移量的值,就可唯一确定数据项。因此,Flob包含3个重要属性:FileId(唯一的文件标识符)、RecordId(记录标识符)、offset(偏移量),表示数据被存储于某个文件中的某条记录,偏移量是偏移于记录的起始地址,这3个变量组合起来为数据唯一确定在磁盘文件中的位置,可以从数据偏移序列中识别访问模式,也为后续缓存结构设计发挥重要作用。



DbArray : MPoint				
up1	up2	up3	up4	up5
ID : int				

ID : 6022
Type : Mpoint
Value:
(
(
("2008-02-02-17:33:17" "2008-02-02-17:47:10" TRUE FALSE)(116.50099 40.00638 116.50099 40.00638)) up1
(
("2008-02-02-17:47:10" "2008-02-02-17:48:22" TRUE FALSE) (116.50099 40.00638 116.50068 40.0058)) up2
(
("2008-02-02-17:48:22" "2008-02-02-17:53:24" TRUE FALSE)(116.50068 40.0058 116.48254 40.00453)) up3
(
("2008-02-02-17:53:24" "2008-02-02-17:57:14" TRUE FALSE)(116.48254 40.00453 116.46739 40.01027)) up4
(
("2008-02-02-17:57:14" "2008-02-02-17:58:26" TRUE FALSE) (116.46739 40.01027 116.46688 40.00766)) up5
)

图1　移动对象MPoint的存储方式

Fig.1　Storage method of a moving object Mpoint

3　面向轨迹数据的缓存管理机制

3.1　读写访问机制

轨迹数据管理系统中各层级间自顶向下,主要由3个部分组成:图形用户界面(Graphical user interface, GUI)连接器、优化器和内核层。在代数模块里进行逻辑处理时,存储引擎确定要从何处调用重要函数获取数据。在分析轨迹数据存储结构时,已确定了Flob和DbArray是存储的基础,图2展示了查询时涉及的数据访问过程,用户区发生读写活动需要访问数据,经内核对查询操作分析后从存储管理层获取数据,具体可分为读取和写入两种情况分析。

读操作。当读取数据时,存储管理层通过调用DbArray的Get()函数从数组的索引下标迭代读取UPoint组成MPoint,然后调用Flob的Read()函数唤醒FlobManager存储管理器的getData()函数触发缓存层。若通过某些特定计算能够在缓存空间查找到当前轨迹数据则表示命中,否则就从磁盘上读取数据,此时系统需要创建一块新的缓存区域,先读入进来,再复制到缓冲区供给外界读。

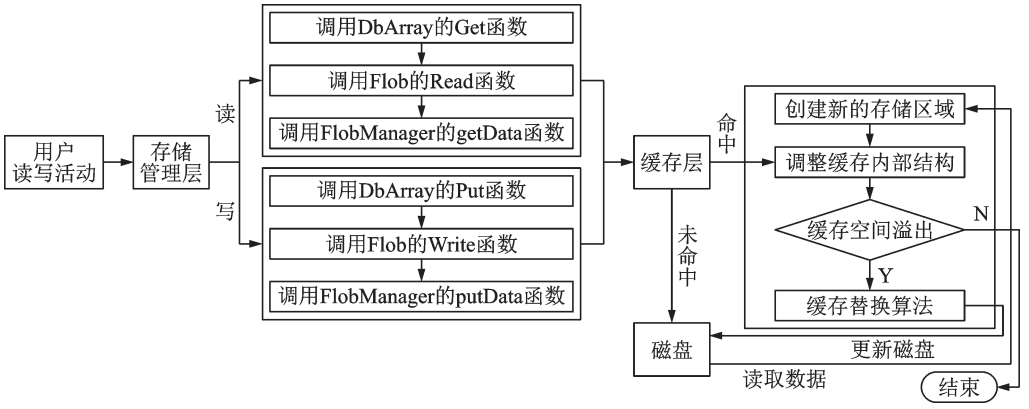


图2 轨迹数据读写访问过程

Fig.2 Trajectory data read/write access process

写操作。当数据发生写操作时,存储管理层调用DbArray的Put()函数按照索引下标依次将buffer缓冲区的UPoint数据写入数组对应位置,如果数组长度不够,则会自动分配增长空间。然后调用Flob的Write()函数唤醒FlobManager的putData()函数触发缓存层,将数据写入缓存。如果在缓存空间中查找到当前轨迹则表示命中,直接修改缓存块数据即可,否则系统会创建一块新的缓存区域,再将数据写入该区域,并同步写入磁盘以确保数据一致性。

由上述分析可知,在读写访问时,FlobManager负责最后一层函数调用,也是真正接触数据的一层,因此在FlobManager中建立接口去触发缓存层最为合适。由于缓冲区通常是在内存中申请,所以其总容量远远小于实际物理文件大小,缓冲区终究会被填满。将未缓存的页面从磁盘中加载进来的过程称为换入,将修改或淘汰的缓冲页刷写回磁盘称为换出。但不能随意地将缓存中的某页给置换出去,如果该缓存页上包含访问频繁的热点数据,则会导致大部分时间浪费在页面置换,磁盘输入输出开销增大,而中央处理器(Central processing unit, CPU)实际工作时间大幅降低,因此需要针对不同的情况设计不同的页面置换算法来淘汰缓存页。接下来的缓存组织结构设计就是为了方便在缓存替换时选择低价值的缓存块淘汰。

3.2 缓存组织结构

本文将缓存换入换出的基本单位设为缓存块,称为CacheEntry,因为轨迹数据的变长存储,一个缓存块存储一个轨迹,缓存块大小由当前轨迹所占字节数决定。考虑到查询时应当从缓存中快速定位数据的问题,即在查找时达到$O(1)$时间复杂度,本文借助哈希页面检索技术结合双向链表的形式进行缓存块组织。表1介绍了CacheEntry类的成员属性,其中变量flob记录了当前轨迹数据在磁盘上的位置,其包含的3个属性值(FileId、RecordId、offset)和slotNo唯一标识一个CacheEntry,CacheEntry*类型的两对前驱后继指针用于在哈希表和双向链表中高效查询、插入或删除数据。

表1 CacheEntry类的属性信息

Table 1 Attribute information of CacheEntry class

变量名	类型	含义
flob	Flob	该缓存块中移动对象的来源,包含FileId,RecordId,offset
slotNo	size_t	在flob中的插槽编号
tableNext	CacheEntry*	该缓存块在哈希表中的后继
tablePre	CacheEntry*	该缓存块在哈希表中的前驱
lruNext	CacheEntry*	该缓存块在双向链表中的后继
lruPre	CacheEntry*	该缓存块在双向链表中的前驱
mem	char*	起始地址
size	size_t	该缓存块总大小

为了保证数据库缓存变化的一致性,缓存

实体在数据库系统中只能实例化一个对象,该缓存对象控制整个缓存结构,因此需要以单例模式运行,由 FlobManager 存储管理层触发和关闭。表2介绍了缓存类的成员信息,hashTable 指向哈希表的起始地址,hashTable 的长度由缓存总空间决定。哈希表的 Value 域存储一系列 CacheEntry 类型的指针,first 和 last 指针分别指向缓存链表的头部和尾部,是为了在缓存块替换时能够直接定位在双向链表的首尾进行操作,usedSize 记录当前已用缓存空间,当 usedSize 大于 maxSize 时进行缓存替换。

表2　缓存类的属性信息

Table 2　Attribute information of cache class

变量名	类型	含义
maxSize	size_t	缓存总空间大小
usedSize	size_t	缓存已用空间大小
hashTable	CacheEntry**	哈希表的起始地址
tableSize	size_t	哈希表长度
first	CacheEntry*	双向链表头指针
last	CacheEntry*	双向链表尾指针

哈希表使用链地址法存储,链地址法的优势是当出现哈希冲突时不存在开放定址法更换地址的问题,无论有多少冲突,只需在当前双向链表增加结点即可。在 MOD 中,计算哈希表 Key 值的方式是利用 flob 的3个属性和 slotNo 唯一确定 CacheEntry 的特点,对哈希表总长度求余,余数相同即表示在同一个 Key 下的双向链表中,具体计算方式如式(1)所示。

$$Key=(fileId+recordId+offset+slotNo)\%tableSize \tag{1}$$

图3描述了详细的缓存组织结构,其中上方是 CacheEntry 的结构,包含1个数据域和4个指针域,黑色箭头是哈希表中链接缓存块的指针,蓝色箭头是缓存链表中链接缓存块的指针;下方是链式哈希表和双向链表的组合,当 first 和 last 同时为空时表示缓存中无数据。这样设计的目的是方便借助 LRU 思想,使用链接列表来存储数据,适应大规模查询时缓存块的灵活高效变化。新访问的数据通过式(1)计算出 Key 后存储于 hashTable[Key]对应的 tableList 的末尾,然后再利用 first 指针将其插入到 lruList 的头端,这一过程对应缓存块换入(page_in),lruList 里的缓存块次序会随着数据情况而经常变化。当 lruList 的空间溢出时,尾部的数据也就是 last 指向的缓存块数据将被淘汰,这一过程对应缓存块换出。

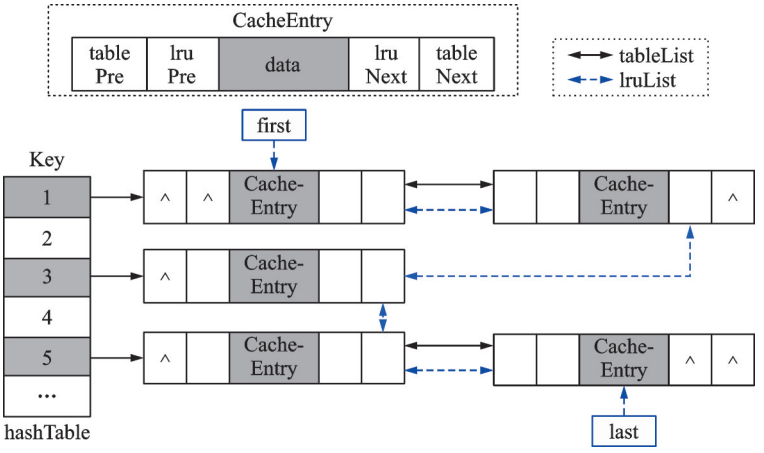


图3　缓存组织结构

Fig.3　Structure of cache

3.3　面向轨迹数据查询的缓存替换方法

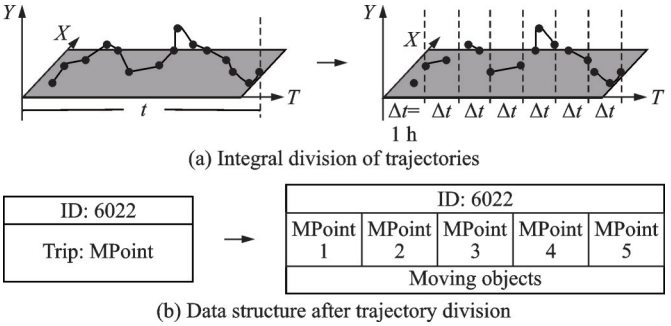
缓存替换的目的是提高特定场景下的数据命中率,降低磁盘IO开销。MOD 使用了较传统的替换算法,并没有考虑轨迹变化频繁、时空范围广以及数据访问的时空规律等特点。因此,提出一种结合三维网格访问热度的缓存替换策略 LRU-CH,在继承 LRU 算法优点的同时,充分挖掘了移动对象的时空特性。

3.3.1　三维空间网格划分与映射

在真实的数据采集场景中,需要通过全球定位系统等设备连续多天跟踪移动对象的轨迹,由于现

实世界的复杂性和数据采集的延迟性，获取的轨迹点多变且时空范围跨度都大，整条轨迹都会从数据库磁盘文件读取到缓存中，这将导致缓存溢出和替换的概率极大。所以缓存优化的第一步是需要进行轨迹划分以减少无用数据占据的缓存空间。

经对采集的原始数据集进行研究发现可以采取通用的整时划分方案，整时划分是从时间角度出发，按照轨迹点在每个完整小时内划分成子轨迹，时间起始点是整时刻。图4展示了轨迹划分前后数据及结构的变化。划分后一个MPoint变成若干条子轨迹MPoint，它们的共同ID都为6022，所以此时ID不再唯一代表一个轨迹，新增一个属性作为唯一标识。



(a) Integral division of trajectories

(b) Data structure after trajectory division

图4 轨迹划分

Fig.4 Trajectory partition

子轨迹具备3个基本属性：经度(X)、纬度(Y)、时间(T)，以[ID, TupleID, Trip]形式呈现给用户，其中ID是子轨迹隶属的原始轨迹，TupleID是子轨迹在系统里的唯一性编号，Trip是由轨迹点按照时间升序相连组成的MPoint类型。数据集里的所有轨迹分布在一个三维的网格，称为三维边界框。用户查询轨迹时聚焦于时空范围查询，查询条件对应的就是一个三维小网格，划分数据集三维空间的目的是将子轨迹与三维小网格进行映射，当访问到某个轨迹时根据映射关系直接定位到所属小网格，将网格热度递增，在缓存溢出时通过轨迹所属小网格的热度直接替换。

三维网格的划分如图5所示。首先统计出数据集的时间和空间范围，得到最大的三维矩形框。然后对于X和Y方向采用定点枚举的方式，时间复杂度为$O(mn)$，每枚举一次，就需要计算子轨迹的bbox与所有小网格是否相交，记录下来每个子轨迹跨越的网格数，最后可以统计出X、Y结果分布情况，找到最满足三维网格划分标准的X和Y值，至此就可确定3个维度各自划分比例为$[x, y, t]$，则三维网格总数目为$x\times y\times t$。最后计算出每个小网格的坐标范围，并分配唯一编号CellId。

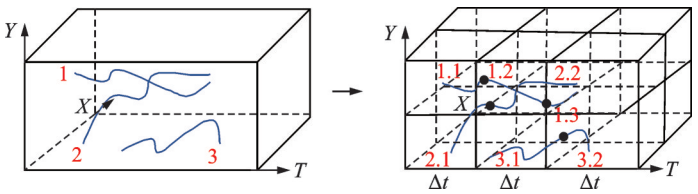


图5 三维空间网格划分

Fig.5 Partition of three-dimensional spatial grid

为了在缓存替换时将子轨迹与小网格高效匹配，需要完成二者之间的相互映射关系。数据由Flob的FileId、RecordId和Offset属性分配空间管理，缓存中的CacheEntry也是由这3个属性标识。在数据读写时，数据库引擎会通过FileId和RecordId找到对应数据，所以可以在数据集存入数据库时将轨迹的ID-TupleID-Trip-FileId-RecordId条目保存到日志文件里。然后再通过系统内部的BBox()函数计算出每个轨迹的三维边界框范围。三维空间划分后大部分子轨迹只跨越一个小网格，少部分与多个网格相交，当涉及多个网格时，采用3D bbox的范围计算出子轨迹的中心点，将中心点所在网格设定为子轨迹所属网格。最后即可将移动对象的FileId-RecordId与CellId互相绑定，建立映射关系。

3.3.2 基于三维空间网格热度的替换

本文缓存替换策略的核心是在LRU算法的基础上结合移动对象的时空特性进行改进，在执行查询时，最近访问的轨迹数据仍按照LRU方式被放置在链表头部，通过子轨迹与三维网格的映射文件可以

直接得到当前子轨迹所在网格的编号,将其热度增加,当计算出缓存容量溢出后,从LRU链表尾部向上寻找目标对象替换,判断当前对象所在网格热度是否偏高,若偏高,则继续向上查找至与first指针相遇,否则直接将该缓存块换出。

解决上述问题后,以数据读操作为例描述LRU-CH策略的详细过程,如算法1所示,通过flob和slotNo计算出子轨迹所在的哈希表位置index,先判断能否在hashTable[index]的tableList链表中遍历到该子轨迹所在的缓存块,若能找到,则直接将缓存块数据读入buffer中,再移动到lruList头部,否则,新创建一个缓存块newEntry,从磁盘读取该子轨迹,再读入buffer中,并将该newEntry挂载到tableList尾部和lruList头部。接着判断缓存空间是否溢出,若溢出则计算出三维网格的第K大热度值heatValue,从lruList尾部寻找所在网格的热度值低于heatValue的对象进行替换。

算法1　LRU-CH替换策略

输入:轨迹所在的flob,插槽位置slotNo,缓存总空间maxSize,已用空间usedSize

输出:读取数据存进buffer

```
(1)   index ← ( fileId + recordId + offset + slotNo) % tableSize;//计算轨迹在哈希表中位置
(2)   IF hashTable[index] = 0 ‖ (hashTable[index] != 0 &&在tableList中查找不到该轨迹)
(3)     新建一个缓存块newEntry存储轨迹;
(4)     将newEntry添加到hashTable[index]链表尾部;
(5)     将数据读入buffer;
(6)     usedSize += newEntry.size;
(7)     将newEntry直接链接到lruList的头部;
(8)     IF usedSize > maxSize
(9)       heatValue ← RandomSelect(Arr, p, r, K);//计算出第K大热度值;
(10)        从lruList尾部寻找网格热度值低于heatValue的子轨迹替换掉;
(11)      END IF
(12)    ELSE
(13)      entry ∈ hashTable[index];//缓存命中
(14)      将数据读入buffer;
(15)      将entry移动到lruList的头部;
(16)    END IF
(17)    RETURN buffer
```

3.4　缓存管理工具MOCache的设计与实现

本文开发了用于轨迹数据库的缓存管理工具MOCache,其主要目标是:(1)缓存状态信息的完整性;(2)实用高效性。该工具可用于分析不同缓存策略对数据库性能的影响,在查询操作期间分析底层缓存访问过程,在每个查询命令结束之后监视缓存的状态变化,以观察命中情况,并根据实际需要调整缓存机制。

在MOD中嵌入缓存状态查询系统表“CACHERST”,用于跟踪系统缓存的状态,实现的难点在于设计数据库底层系统表的逻辑结构,并在每条语句结束时实现及时更新,过程如图6所示。首先设计CacheRST类将缓存信息的属性名称追加进系统嵌套列表NList,该类继承数据库的InfoTuple抽象接口,可以在系统中实现Relation和Tuple的转化。再设计CacheRSTRel类初始化缓存属性的系统数据类型,继承数据库的SystemInfoRel类,系统中的对象标识符可以添加到此类的构造函数中。查询语句

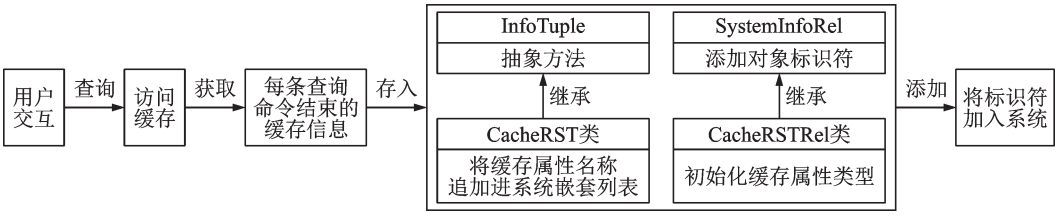


图6 缓存状态标识符“CACHERST”的设计过程

Fig.6 Design process of cache status identifier “CACHERST”

在执行时访问缓存,CacheRST实例会采集属性值,需要注意的是这些属性值应在缓存模块定义,因此应设为全局变量便于引用和修改。将一条语句查询结束后的所有属性值封装进系统的NList嵌套列表结构中,每条语句的结果依次存进系统Vector结构。最后将“CACHERST”标识符插入系统表中,用户在数据库查询窗口输入“query CACHERST”,会展示本轮查询过程中系统状态的变化,也可结合feed、filter对目标命令过滤查询。

4 实验与性能测试

本节在轨迹数据库中设计了详细的实验方法,实现LRU-CH和多种常用替换策略,对命中率和执行时间进行了比较。

4.1 实验准备

实验环境:使用开源的轨迹数据库SECONDO V4.3.0系统,Intel(R) Core(TM) i5-10500 CPU @ 3.01 GHz处理器,Ubuntu 18.04 LTS操作系统。

实验之前需要先了解关于查询操作的基本理论知识。在SECONDO中存储移动对象的查询主体称为关系,如Trains (Id: int, Line: int, Trip: mpoint)是一个描述柏林地铁轨迹的关系。从磁盘上读取关系,利用feed操作符转化成一组元组流,再通过filter选择元组流的子集,最后再通过consume将结果元组流收集到关系中,存储为持久性关系用于查找和索引。数据集:使用公开的北京市出租车轨迹数据集进行实验。数据共包括919条轨迹,1 465 846个轨迹点,数据属性包含车辆ID、时间T、经度坐标X和纬度坐标Y这4个属性。时间范围(2008-02-02-13:31:07,2008-02-08-17:39:19),经度范围(98.920 25,127.016),纬度范围(23.439 87,46.321 69)。该数据集由于是在真实世界采集,环境复杂,原始数据会产生大量脏数据。首先需要处理脏数据,最终得到1 200 000个有效点。

建立索引。在MOD中,轨迹数据具有时间和空间属性,应选择可以充分利用轨迹时空关系的索引结构。本节选择建立R-tree索引来访问数据,轨迹可以通过系统中的bbox操作符生成表示时空范围的三维矩形框(Minimum bounding rectangle, MBR)属性,因此很适合建立索引项。

设计查询语句。在用户实际查询中,时空范围查询应用广泛,因此本节随机选取符合用户查询规范的10个时间段和10个北京地点,进行笛卡尔积组合成100个<时间,地点>条目。为了模拟真实查询场景下用户会对某些时间和地点进行多次查询的情况,本节将前10条重复100次,第11~50条重复10次,第51~100条只重复一次,最后生成1 450条时空范围查询语句。查询语句格式如下:

```
let result1 = BJTaxisMBR_rtree BJTaxisMBR windowintersects [bbox([const upoint value (("2008-02-04-11:00:00" "2008-02-04-12:00:00" TRUE TRUE) (116.402884 39.05438 116.405651 39.913826))])] consume;
```

实验基准。以经典通用的LRU替换算法作为基准,在SECONDO中分别实现LRU、2Q、LFU以及本文提出的LRU-CH缓存替换策略。

4.2　三维网格划分结果

网格划分的目的是让子轨迹的外接三维矩形框包含于划分后的小网格中,根据三维网格的划分标准可知,网格划分是基于轨迹划分结果进行,轨迹划分在前,网格划分在后,网格划分需要分别在3个维度上进行。

以北京市出租车轨迹数据集为例,首先将包含的919条轨迹按照整时划分后获得119 157条子轨迹,再对整个数据集所在的三维网格进行划分,由于T已经按小时划分得到149段,因此只需对X和Y进行划分,划分结果如表3和图7所示,设n表示跨越网格数,当$X,Y=10,10$时,效果相对最佳,整个数据集所在的三维空间被划分成14 900个小网格,在119 157条子轨迹中有99.7%占据1或2个网格,且横跨1个网格的占比最多。

表3　X、Y的取值

Table 3　Values of X and Y

编号	X	Y
1	7	8
2	8	7
3	8	8
4	8	9
5	9	8
6	9	9
7	9	10
8	10	10
9	10	11
10	11	10

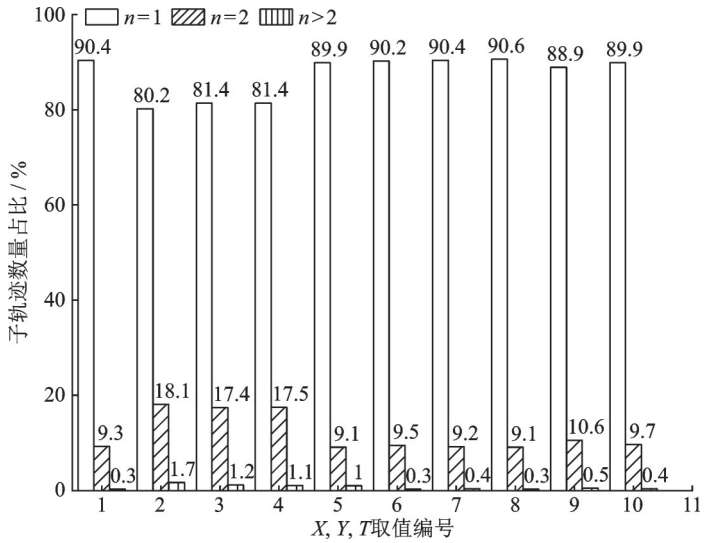


图7　网格划分3个维度取值比较

Fig.7　Comparison of three dimension values in cell partition

4.3　缓存替换性能比较

为了检验本文提出的LRU-CH策略的性能影响,在SECONDO系统中实现了LRU、2Q、LFU、LRU-CH四种缓存替换策略,利用北京市出租车数据集和1 450条时空范围查询语句,针对原始数据和轨迹划分后的数据分别进行实验,对查询命中率和执行时间进行了比较。

LRU-CH策略进行缓存块替换的依据是将位于网格热度低的移动对象淘汰掉,第K大的网格热度值被作为阈值,因此K的取值对替换过程有一定影响。实验设计中一共有100个<时间,地点>条目,对K值从10到100进行实验,得到相对最佳取值30,命中率最高,之后逐渐趋于平稳,因此最终将K值初步确定为30。

在确定K值后可进一步比较LRU-CH与其他策略。图8(a)显示了查询时间对比,可以明显看出经过轨迹划分后的数据,在每一种缓存替换策略下的时间都大大缩短,这也证明了轨迹划分对缓存性能有很大影响,并且轨迹划分后LRU-CH的查询时间最短,表明使用LRU-CH策略时缓存中持有的访问频率高的数据更多,磁盘输入输出次数相对较少。图8(b)展示了不同替换策略的查询命中率对比,可

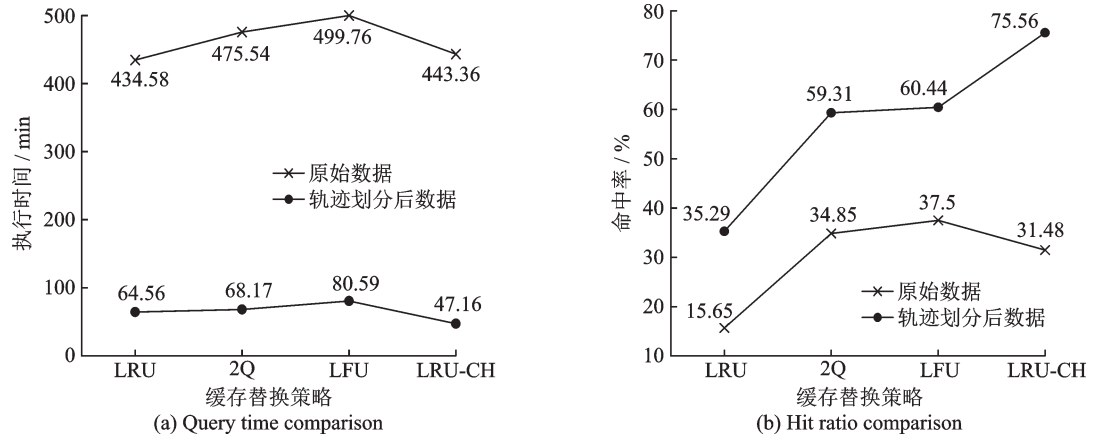


图8 不同缓存替换策略比较

Fig.8 Comparison of different cache replacement strategies

以发现不论是原始数据集还是轨迹划分后的数据集，LRU算法的命中率都最低。在原始数据集下LRU-CH的命中率提高并不明显，2Q和LFU的命中率虽然比较高，但它们的时间也相对较长，而移动对象轨迹经划分之后LRU-CH的命中率明显最高，达到了76.56%。

4.4 MOCache性能测试

测试遍历缓存链表对系统开销的影响，由表4可知，在缓存大小不同的情况下，每条语句遍历缓存块平均时间都不足1 s，遍历每个缓存块所占的时间微乎其微，因此对系统开销影响可忽略不计。

表4 遍历缓存块时间开销

Table 4 Time overhead for traversing cache blocks

缓存大小/MB	遍历缓存块总时间/s	每条语句遍历缓存块平均时间/s	遍历每个缓存块平均时间/s
64	1 308.2	0.902 2	0.000 001 8
32	695.376	0.479 6	0.000 001 9
16	364.237	0.251 2	0.000 001 9
8	223.36	0.154	0.000 002 4

5 结束语

本文为轨迹数据库系统设计了合适的缓存管理机制，结合存储管理层的基础架构设计了轨迹数据的缓存组织结构，然后考虑轨迹的查询应用场景和时空访问特性，提出了基于三维空间网格热度的缓存替换策略，最后嵌入开发了一个缓存管理监控工具MOCache。本文的缓存访问机制主要针对的是以DbArray作为存储载体的移动对象，适合MOD中的范围查询，可以细分为时间点、时间段、空间位置和空间区域等不同范围的查询；最近邻查询，即查询空间中最靠近查询点的指定个数移动对象；轨迹相似性查询，通常用于异常检测、轨迹分析和预测等领域。接下来可以拓展缓存结构使其能够接收多种类型的数据，同时支持时空聚集查询、连接查询等更为复杂的数据库查询操作。

参考文献：

[1] CUDREMAUROUX P, WU E, MADDEN S. TrajStore: An adaptive storage system for very large trajectory data sets[C]// Proceedings of 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010). Piscataway: IEEE, 2010: 109-120.

[2] WANG Haozhou, ZHENG Kai, XU Jiajie, et al. SharkDB: An in-memory column-oriented trajectory storage[C]//Proceedings of the 23rd ACM International Conference on Information and Knowledge Management. New York: ACM, 2014: 1409-1418.

[3] SHANG Zeyuan, LI Guoliang, BAO Zhifeng. DITA: Distributed in-memory trajectory analytics[C]//Proceedings of the 2018 International Conference on Management of Data. New York: ACM, 2018: 725-740.

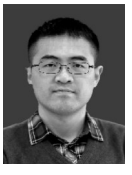
[4] BRISABOA N R, GAGIE T, GÓMEZ-BRANDÓN A, et al. An index for moving objects with constant-time access to their

compressed trajectories[J]. International Journal of Geographical Information Science, 2021, 35(7): 1392-1424.
[5] DURNER D, CHANDRAMOULI B, LI Y. Crystal: A unified cache storage system for analytical databases[J]. Proceedings of the VLDB Endowment, 2021, 14(11): 2432-2444.
[6] KHAN T A, ZHANG D, SRIRAMAN A, et al. Ripple: Profile-guided instruction cache replacement for data center applications[C]//Proceedings of the 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). Piscataway: IEEE, 2021: 734-747.
[7] VAKIL-GHAHANI A, MAHDIZADEH-SHAHRI S, LOTFI-NAMIN M R, et al. Cache replacement policy based on expected hit count[J]. IEEE Computer Architecture Letters, 2017, 17(1): 64-67.
[8] MORALES K, LEE B K. Fixed segmented LRU cache replacement scheme with selective caching[C]//Proceedings of the 2012 IEEE 31st International Performance Computing and Communications Conference (IPCCC). Piscataway: IEEE, 2012: 199-200.
[9] WU Xiang, CAI Delin, GUAN Shujie. A multiple LRU list buffer management algorithm[C]//Proceedings of IOP Conference Series: Materials Science and Engineering. Bristol: IOP Publishing, 2019: 052002.
[10] GÜTING R H, ALMEIDA V, ANSORGE D, et al. Secondo: An extensible dbms platform for research prototyping and teaching[C]//Proceedings of the 21st International Conference on Data Engineering (ICDE'05). Piscataway: IEEE, 2005: 1115-1116.
[11] HUANG Y K. Indexing and querying moving objects with uncertain speed and direction in spatiotemporal databases[J]. Journal of Geographical Systems, 2014, 16(2): 139-160.
[12] XU J, LU H, GÜTING R H. Range queries on multi-attribute trajectories[J]. IEEE Transactions on Knowledge and Data Engineering, 2017, 30(6): 1206-1211.
[13] LAN Hai, BAO Zhifeng, PENG Yuwei. A survey on advancing the dbms query optimizer: Cardinality estimation, cost model, and plan enumeration[J]. Data Science and Engineering, 2021, 6: 86-101.
[14] WANG Yinyin, YANG Yuwang, QIU Xiulin, et al. CCF-LRU: Hybrid storage cache replacement strategy based on counting cuckoo filter hot-probe method[J]. Applied Intelligence, 2022, 52(5): 5144-5158.
[15] JAIN A, LIN C. Back to the future: Leveraging Belady's algorithm for improved cache replacement[C]//Proceedings of the 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA). Piscataway: IEEE, 2016: 78-89.
[16] SHRIVASTAVA R K, NATU V, HOTA C. Code integrity verification using cache memory monitoring[J]. Information Security Journal: A Global Perspective, 2022, 31(2): 226-236.
[17] CHEN T H, SHANG Weiyi, HASSAN A E, et al. CacheOptimizer: Helping developers configure caching frameworks for hibernate-based database-centric web applications[C]//Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: Association for Computing Machinery, 2016: 666-677.
[18] EVTYUSHKIN D, PANFILOV P, PONOMAREV D. CacheVisor: A toolset for visualizing shared caches in multicore and multithreaded processors[C]//Proceedings of the International Conference on Parallel Computing Technologies. Heidelberg: Springer, 2011: 284-289.
[19] DOYCHEV G, KÖPF B, MAUBORGNE L, et al. CacheAudit: A tool for the static analysis of cache side channels[J]. ACM Transactions on Information and System Security (TISSEC), 2015, 18(1): 1-32.
[20] TUNCER D, SHERBORNE T, CHARALAMBIDES M, et al. CacheMAsT: Cache management analysis and visualization tool[C]//Proceedings of the 2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM). Piscataway: IEEE, 2017: 875-876.

作者简介:

金鑫(1981-),女,博士研究生,研究方向:数据管理,E-mail:jinxin@nuaa.edu.cn。

吴冰雅(1999-),女,硕士研究生,研究方向:移动对象数据库。



许建秋(1982-),**通信作者**,男,教授,博士生导师,研究方向:数据软件、空间数据库、移动对象数据库、可扩充数据库,E-mail:jianqiu@nuaa.edu.cn。

(编辑:夏道家)