软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(5):2202−2234 [doi: 10.13328/j.cnki.jos.007483] [CSTR: 32375.14.jos.007483]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于数据分布的移动对象学习索引及查询算法*

王撷阳[1], 巢 成[1], 金 鑫[1], 许建秋[1], 高云君[2]

[1](南京航空航天大学 计算机科学与技术学院/软件学院, 江苏 南京 211106)

[2](浙江大学 计算机科学与技术学院, 浙江 杭州 310058)

通信作者: 许建秋, E-mail: jianqiu@nuaa.edu.cn

摘 要: 移动对象的来源丰富、获取简单、运动频繁, 导致数据量呈现爆发式增长, 高效管理移动对象数据的需求日益增加, 使得移动对象数据的索引及查询成为亟待解决的热点问题. 传统的移动对象索引基于空间划分, 能够有效地处理对象的空间位置和时间变化, 但由于移动对象的动态特性需要频繁更新索引, 在对象数量庞大时会导致维护成本显著增加. 学习索引作为新型索引技术, 可以运用机器学习方法提高查询效率, 降低存储成本, 但学习索引并不适用于具有多维特性的移动对象数据. 为此, 提出了一种基于非均匀网格降维的学习索引 NUGC_LI, 使用类似 B+树的递归层次模型结构. 该学习索引分为根节点、内部节点和叶子节点这 3 个部分, 使用多阶段线性模型对灵活划分后的数据分布进行拟合学习, 并在叶子节点中设置有空隙的数组和节点关键值范围, 提高节点更新和查询效率. 同时, 对真实出租车轨迹、系统仿真火车轨迹和随机生成轨迹数据集分别建立了 B+树、RMI、ALEX、NUGC_LI、3DR 树与 TB 树索引. 真实数据集、仿真数据集和随机数据集中涉及的轨迹点分别约 917 000 个、51 544 个和 5 222 752 个. 通过对比实验与伸缩性测试, 在索引构建上, NUGC_LI 相较于 TB 树、3DR 树、B+树、RMI 和 ALEX 分别降低了约 91.45%、89.63%、90.38%、87.46% 及 13.71% 的构建时间; 在更新操作上, 其更新时间降低至少 93.76%. 基于 NUGC_LI 的范围查询、最近邻查询和相似轨迹查询在大数据量条件下均显示出显著优势, 查询时间分别至少比 ALEX 降低 8.74%、30% 和 16.07%; 比 RMI 降低 29.38%、77.44% 和 25.24%; 比 B+树降低 52.72%、92.44% 和 70.5%; 比 3DR 树降低 53.09%、91.2% 和 67.58%; 比 TB 树降低 52.67%、90.43% 和 67.47%. NUGC_LI 索引在多任务负载下不仅具备较高的扩展性, 而且在构建、更新以及查询操作中均实现了显著的性能提升.

关键词: 学习索引; 机器学习; 非均匀网格; 查询算法; 移动对象

中图法分类号: TP311

中文引用格式: 王撷阳,巢成,金鑫,许建秋,高云君.基于数据分布的移动对象学习索引及查询算法.软件学报,2026,37(5):2202–2234. http://www.jos.org.cn/1000-9825/7483.htm

英文引用格式: Wang XY, Chao C, Jin X, Xu JQ, Gao YJ. Moving Object Learned Index and Query Algorithm Based on Data Distribution. Ruan Jian Xue Bao/Journal of Software, 2026, 37(5): 2202–2234 (in Chinese). http://www.jos.org.cn/1000-9825/7483.htm

Moving Object Learned Index and Query Algorithm Based on Data Distribution

WANG Xie-Yang[1], CHAO Cheng[1], JIN Xin[1], XU Jian-Qiu[1], GAO Yun-Jun[2]

[1](College of Computer Science and Technology/College of Software, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China)

[2](College of Computer Science and Technology, Zhejiang University, Hangzhou 310058, China)

Abstract: The abundance of sources, ease of acquisition, and frequent movement of moving objects have led to exponential growth in data volume. The growing need for efficient management of moving object data has made indexing and querying such data a pressing

* 基金项目: 国家自然科学基金 (62472217, U23A20296)
收稿时间: 2024-11-28; 修改时间: 2025-04-02; 采用时间: 2025-06-06; jos 在线出版时间: 2025-10-29
CNKI 网络首发时间: 2025-10-31

issue. Traditional moving object indexes, based on spatial partitioning, can effectively handle changes in the spatial position and temporal dynamics of objects. However, due to the dynamic nature of moving objects, which require frequent index updates, maintaining these indexes becomes costly with large datasets. Learned indexes, as an emerging indexing technique, have the potential to improve query efficiency and reduce storage costs by leveraging machine learning methods. Nevertheless, learned indexes are not well-suited for data with multidimensional characteristics. To address this limitation, the proposed learned index uses the non-uniform grid code algorithm (NUGC_LI). It employs a recursive hierarchical model structure similar to the B+-tree, divided into root, internal, and leaf nodes. The learned index uses a multi-phase linear model to adapt to the flexibly divided data distribution, setting arrays with gaps and node key value ranges in the leaf nodes to improve node update and query efficiency. At the same time, B+-tree, RMI, ALEX, NUGC_LI, 3D R-tree, and TB-tree indexes are constructed for real taxi trajectories, system simulation train trajectories, and randomly generated trajectory datasets for comparison. The number of trajectory points in the real, simulated, and random datasets is approximately 917 000, 51 544, and 5 222 752, respectively. Through comparative experiments and scalability tests, NUGC_LI reduces the index construction time by approximately 91.45%, 89.63%, 90.38%, 87.46%, and 13.71% compared to TB-tree, 3D R-tree, B+-tree, RMI, and ALEX, respectively. For update operations, the update time is reduced by at least 93.76%. Range queries, nearest neighbor queries, and similar trajectory queries based on NUGC_LI show significant advantages under large-scale data conditions, with query times reduced by at least 8.74%, 30%, and 16.07% compared to ALEX; 29.38%, 77.44%, and 25.24% compared to RMI; 52.72%, 92.44%, and 70.5% compared to B+-tree; 53.09%, 91.2%, and 67.58% compared to 3D R-tree; and 52.67%, 90.43%, and 67.47% compared to TB-tree. The NUGC_LI index not only demonstrates high scalability under multi-task loads but also achieves significant performance improvements in construction, updates, and query operations.

Key words: learned index; machine learning; non-uniform grid; query algorithm; moving object

基于空间位置的应用服务已广泛融入日常生活的各个领域, 移动对象数据处理因此成为亟待解决的关键问题. 随着物联网位置获取技术的持续发展, 研究者获取移动对象数据的方式变得日益简便. 例如, 微信朋友圈定位、智能移动设备定位、出租车与公交车定位等应用需求[1]均是将移动对象在其移动过程中所记录的经纬度信息按时间顺序排列, 形成一系列具有时间属性的多维坐标. 由于移动对象数据来源广泛、获取便捷且采集频率较高, 导致数据呈现爆发式增长, 成为数据库领域的研究热点之一.

当数据增长的速度已经远超学习和处理的速度时, 数据量的存储将会趋于极限, 这就要求索引结构的设计能够减少物理空间的占用量和存储成本, 同时能够充分保留移动对象数据的时空相似性. 近年来, 人工智能、机器学习等技术迅猛发展, 数据库管理技术也随之发生变化, 研究学者越来越多地引入机器学习方法来构建学习索引[2], 索引结构也不断被调整和扩展.

学习索引有效规避了传统索引中忽略数据的分布特性、树状索引结构深度过大、磁盘 I/O 冗余操作等缺陷, 将索引看作一种模型, 通过机器学习模型来拟合数据分布, 将查询键作为模型输入值, 通过预测可以返回查询键对应记录所在的位置, 有效提高了查询效率、减小了索引文件的存储成本. 因此, 在传统索引结构的基础上, 引入学习索引, 贴合实际应用场景, 优化索引的更新方法, 从而提升数据库查询的效率是十分必要的.

当前学习索引的应用对象仍以一维数据为主, 所以在面向移动对象构建学习索引之前, 应先对移动对象数据进行降维处理. 数据降维是研究者们在处理和存储多维度数据时, 为解决其空间占用问题而提出的, 旨在将高维数据映射到低维数据空间, 但是移动对象数据与其他多维数据相比, 具有明显的空间特性, 如何在降维过程中充分保留移动对象数据的空间位置特征仍是值得研究的问题.

移动定位技术和无线通讯技术的广泛应用, 使得移动对象数据的相关研究变得越发重要. 为了提高移动对象复杂操作的效率, 节省数据存储的占用率和索引文件的更新频率, 开展了面向移动对象数据的学习索引与查询算法研究, 其意义可以概括为以下 3 点.

(1) 对于海量移动对象数据的处理更加友好. 对移动对象数据进行基于空间位置关系的高效降维有利于在压缩存储空间的同时保留位置特征. 索引构建时, 基于降维后数据的分布特点进行数据划分, 运用机器学习模型充分学习数据的分布规律, 有效避免复杂的运算操作, 利用简单的线性回归模型直接预测需查找数据的位置.

(2) 有效提高查询操作的效率. 传统索引基于树的数据结构每次查询都需要从树的根节点开始遍历搜索, 直到抵达叶节点, 找到数据存储的位置. 随着数据量的增大, 索引结构过深, 查询性能也随之降低. 但是学习索引使用学

习模型, 并非通过遍历查询数据, 而是通过训练得到模型, 模型根据输入值得到预测位置, 学习索引在以预测位置为中心的误差区间内进行遍历查询, 将查询的时间复杂度从原本的 O(log$n$), 降低为 O(C); 索引文件的空间成本降低, 传统索引在存储时通常以文件形式存储, 但是对于学习索引的叶子节点只需要存储模型参数, 将存储的空间复杂度从 O(n), 降低为 O(C); 针对现有学习索引缺少高效更新机制的问题, 设计与研究了针对移动对象学习索引的插入更新机制, 提高索引的可扩展性.

(3) 学习索引可应用于多种移动对象查询, 实用性高. 针对所提出的学习索引, 实现了在移动对象查询领域应用最为广泛的范围查询、最近邻查询和相似轨迹查询算法, 以验证学习索引在查询应用中的准确性和高效性, 同时提高移动对象学习索引的实用性.

高效的降维算法和学习索引的设计与研究, 对于优化移动对象数据库中的查询操作具有十分重要的现实意义, 可以应用到数据量庞大且响应速度要求较高的位置服务领域中, 有利于提高移动对象数据库的管理能力. 考虑传统的索引结构在处理海量复杂的移动对象时存在较大难度, 而基于机器学习模型构建的学习索引, 能够充分学习移动对象数据和存储位置的关系, 训练后的模型在查询性能和存储成本上优于传统的索引结构, 体现了数据分布的特点. 目前已有的学习索引多是针对基于单个机器学习模型而构建[3], 存在功能局限性, 因此本研究根据移动对象数据分布特点与数据库查询操作要求, 构建适应大规模移动对象数据集、在更低的空间占用下实现更高查询性能的学习索引. 研究工作概括如下.

(1) 设计基于移动对象数据\空间位置关系的均匀网格编码 (uniform grid code, UGC) 算法, 提出具有数据分布感知的降维算法, 即根据数据分布优化网格布局, 称为非均匀网格降维算法 (non-uniform grid code algorithm, *NUGC*), 从而支持非均匀数据分布优化存储空间, 在压缩移动对象数据存储空间的同时充分保留其空间位置关系的特点, 通过非均匀网格降维得到的一维值为学习索引构建和查询研究提供数据基础.

(2) 提出基于空间位置关系降维的一维非均匀网格学习索引 (non-uniform grid code learned index, NUGC_LI), 该索引结构为递归层次模型, 含根节点、内部节点和叶子节点这 3 部分, 每个节点均含有线性回归模型, 将查询输入值转化为数据位置的预测值, 并在预测值的误差范围内展开搜索. 移动对象数据具有动态更新的特点, 因而对设计的学习索引制定插入更新机制, 使其具备支持历史数据插入、删除的功能. 同时, NUGC_LI 支持节点的动态分裂, 能够更好地满足移动对象的更新需求.

(3) 设计移动对象数据查询领域应用较为广泛的查询算法, 实现使用学习索引对移动数据进行查询的算法研究, 并测试查询操作的效率与准确率, 在不同查询场景下将非均匀网格学习索引和传统索引、经典学习索引进行性能对比, 分别在真实采集轨迹数据、模拟仿真轨迹数据和随机生成轨迹数据等不同数据集上验证其性能优势.

综上所述, 本研究旨在通过构建学习索引提高移动对象数据的查询效率. 基于真实的全球定位系统 (global positioning system, GPS) 得到采集数据, 对多维数据进行数据准备和基于位置的降维处理, 然后利用机器学习模型及时学习和更新能力的特点, 构造出能够提高查询性能、降低空间占用的学习索引结构. 该索引根据查询参数直接确定查询结果所在数据块, 避免中间节点的访问, 并对参数调优以适应数据特点与任务需求.

1 相关工作

国内外针对移动对象数据的研究已发展多年, 取得了丰富的研究成果. 空间索引作为移动对象数据库的重要组成部分, 被国内外学者广泛研究, 很多通用高效的索引算法被相继提出. 但是随着目前物联网技术和智能应用的不断发展, 各种新型数据不断涌现, 新型索引和查询算法需要适应新应用下用户的需求. 尤其是在现实应用场景下要求系统能够在海量数据中迅速找到对用户有用的信息, 要求实现的索引及查询操作更具通用性, 这也要求针对移动对象数据索引及查询的研究工作能够更加深入并不断创新[4,5].

学习索引利用机器学习模型, 输入查询键值, 预测相应数据记录存储位置. 由于学习索引对模型进行训练以反映数据模式的相关性, 数据需具备有序性, 所以目前的研究趋势仍以一维数据为主, 对于多维空间数据的学习索引建立则是先通过降维算法将多维数据转化为一维数据, 再使用机器学习方法进行处理[6,7].

1.1 时空数据索引结构

在移动对象数据库中数据访问与查询的频率高, 由于移动对象的动态性, 迅速定位数据位置成为数据库的基础操作之一, 这对于提升查询操作的总体性能至关重要. 空间索引是移动对象数据库中用于提升查询和存储性能的数据结构, 是依据移动对象的位置信息或空间关系等特征, 按照一定的顺序存储数据的结构. 通过空间索引可以快速锁定数据位置范围, 能够有效地提高空间查询操作的效率[8,9]. 现有的空间索引主要分为以下 4 类.

1.1.1 基于二叉树的空间索引

二叉树是应用最为广泛的树形数据结构之一, 其存储结构及算法都较为简单, 每个树节点最多有两棵子树, 以左右区分. 在移动对象数据库中, 基于二叉树的索引结构以 KD 树[10]为典型代表, KD 树是一种二分索引树结构, 即 K 维的二叉树, 是一种空间划分的数据结构, 其中的每一个节点都是 K 维的数据, 主要功能是对多维数据进行索引, 常用作空间划分及最近邻查询. 2019 年, Ram 等人[11]以随机分区树为基础, 针对高维度移动对象数据集, 提出了基于 KD 树的近似搜索方案, 克服了 KD 树长期以来不适合用于精确最近邻搜索的难题, 并通过实验验证了该方案的搜索准确性和查询时间保证. 虽然 KD 树是一种对 K 维空间的移动对象点检索快速的树形数据结构, 但 KD 树针对空间关系的查询效率非常低.

研究者们为了优化 KD 树, 在此基础上做了很多探索. 2018 年, Nurgaliev 等人[12]通过在区块链系统中引入加密签名的树结构, 实现了对时空区块链数据的高效查询处理, 维护了数据存储和完整性, 并通过实验综合评价了所提方法的通用性和有效性. 此外, 研究者们还提出了 KDB 树, 将 KD 树与 B 树结合的索引结构, 目的是将 KD 树存储到外存中. 2013 年, Li 等人[13]提出一种工作负载自适应的在线算法, 该方法在闪存上实现了 KDB 树, 将树节点表示为日志集合, 称为日志记录条目, 以有效处理节点的细粒度更新, 提高查询性能. 2017 年, Liu 等人[14]提出基于加密的安全框架以保障空间众包分配时工作人员的安全隐私问题, 为了提高分配效率, 提出了一种基于 SKD 树的新型安全索引技术, 在集合索引时以查询对象的中心点作为查询键.

1.1.2 R 树及其主要变体

基于 R 树的空间索引最早是由 Guttman[15]提出的, 是通过普通关系数据库中的 B 树索引改进而来, 适用于多维空间非零数据对象的动态索引结构. R 树中的每个非叶子节点存放的是数据集空间中的一个矩形和一个指针, 该矩形是包含其所有子节点的最小外包矩形 (minimum bounding rectangle, MBR), 该指针指向的是下一层节点. R 树的叶子节点则存放了指向查询对象位置的指针. 2020 年, Qi 等人[16]提出了一种基于空间填充曲线的 R 树堆积策略, 满足实际和最坏情况下数据的多样访问, 该策略会在最坏情况下为窗口查询生成具有渐近最佳 I/O 复杂度的 R 树, 实验表明该 R 树在查询不同分布的真实数据和合成数据方面都非常高效.

R 树具有很好的灵活性, 能够保持较高的空间利用率, 允许兄弟节点所对应的空间区域相互重叠, 这就使得 R 树的搜索路径多样化, 导致查询效率降低, 且高频更新导致索引重构开销过大. 为了改进这一不足, 研究者们还陆续提出了 3DR 树、TB 树、TPR 树、TPR*树、SETI、R+树和 Cell 树等数据结构, 来提高查询效率并节省访问时间. 1996 年 Theodoridis 等人[17]提出的 3DR 树将时间视作第 3 维度, 与空间坐标一同构建三维 MBR, 通过三维分裂策略最小化体积增量, 既支持实时位置更新, 也天然保留历史轨迹, 但由于三维分裂与批量重构开销较大, 更新延迟和树膨胀问题仍需通过重建策略加以缓解. 2000 年 Pfoser 等人[18]提出一种基于 R 树的空间索引结构 TB 树. 其叶节点将同一轨迹的各个分段组织在一起, 从而支持高效的轨迹检索. 但由于 TB 树在处理空间聚集性较差的区域时, 可能导致不属于同一轨迹但空间上相近的线段被分散存储在不同的节点中, 从而影响了其空间利用效率与查询性能. 同年, Šaltinis 等人[19]提出的 TPR 树假设移动对象以线性速度运动, 引入速度和时间概念, 其 MBR 会随着时间推移扩大, 查询时将目标时间代入预测边界进行剪枝, 从而大幅减少因频繁位置变化带来的更新成本, 适合速度较稳定的在线服务场景, 但对非线性运动的支持有限. 2003 年, Tao 等人[20]提出的 TPR*树对 TPR 树进行了改进, 通过引入更高效的节点分裂策略和边界扩展算法, 有效减少了查询时的重叠区域和更新开销, 从而在对象运动状态有轻微波动的情形下获得更优的查询性能. 同年, Chakka 等人[21]提出的 SETI 索引在分布式环境中以空间网格与时间分段对时空域进行分片, 在每个分区中的轨迹线段用一个 R 树来索引, 并通过全局目录路由与并行检

索实现线性扩展. 2009 年, Beckmann 等人[22]提出了一种改进的 R*树结构, 适合在数据库管理系统中运行, 可以保证仅在单个路径中插入, 为了使特定的数据分布和插入顺序更具鲁棒性, 研究者们重新设计了子树选择和拆分算法, 实验证明该方法下 R*树的创建速度更快, 处理二维和三维数据查询所需的 I/O 成本平均提高了 30% 以上. 2020 年, Alamri 等人[23]提出一种基于单元格的索引结构即 Cell 树, 用于有效地分组和管理室内空间中移动物体的更新, Cell 树可以有效服务于室内空间查询、邻接查询和基于密度的查询.

1.1.3 基于哈希网格技术的空间索引

基于哈希的网格技术是将整个地理空间划分为网格表示, 然后把划分在相同网格的移动对象存储在同一区域中, 通过网格坐标计算得到该处的地址, 然后对网格中的对象进行索引. 网格索引的典型代表有 Grid file 以及 R-file 等算法, 更加适用于二维或三维移动对象. 2005 年, Hastings 等人[24]对空间区域的哈希技术进行了扩展及优化, 针对空间哈希的碰撞检测功能, 研究者们优化模拟了多个方面, 包括移动物体碰撞、物体–地形碰撞、物体和地形渲染等, 并通过实验给出仿真结果. 2015 年, Ding 等人[25]提出了一种基于自适应网格划分策略的遥感图像认证感知哈希算法, 可以在细粒度级别对信息密集区域进行身份验证. 2020 年, Pâris 等人[26]在 Merkle 树的基础上提出了 Merkle 哈希网格, 将传输和存储成本降低了 50%, 且加快了行列间的搜索速度, 能实现快速定位.

1.1.4 基于空间目标排序的空间索引

空间目标排序法是将地理空间划分为网格后, 为每个网格进行唯一性编码, 网格中对象的地址由它本身和与它相交的网格编码组合形成, 从而将高维空间映射至一维空间. 空间目标排序的典型代表有 location keys、Z-order 等. 2001 年, Kriegel 等人[27]提出了一种高效、动态且可扩展的方法来管理数据库系统中的一维间隔序列, 该方法符合空间填充曲线的概念, 基于关系区间树, 易于嵌入到现代可扩展索引框架中, 并在可用性、并发性和执行性能方面明显优于线性四叉树. 2014 年, Zhang 等人[28]为解决多维树索引结构复杂的问题, 提出了一种基于映射的大小分离索引方法, 将数据和查询映射到一维空间中, 该方法包括尺寸分离、数据分布转换和高效映射算法, 通过广泛实验表明, 该方法比以往基于映射的索引查询效率高出两个数量级, 且比 R 树更为有效. 2017 年, Kumar 等人[29]提出了一个索引和数据分发框架 M-Grid, 提供高效的数据分发、预警机制以及多维数据查询. 在建立索引阶段, 研究者们使用基于 Hilbert 空间填充曲线的技术, 通过保留数据的局部特点有效地管理键值索引, 实验表明 M-Grid 的查询效率与基准结构相比实现了 3 个数量级的性能提升.

1.2 学习索引技术

机器学习模型具有学习数据模式并根据趋势预测数据的优势特点, 所以近年来研究者们考虑使用机器学习方法来解决数据库的热点问题. 2018 年, Kraska 等人[30]提出了学习索引 (learned index, LI), 通过递归模型索引 RMI 提高检索效率. RMI 由多个模型构成包括神经网络、线性回归等, 每个模型以键值作为输入, 返回位置信息. 学习索引使用机器学习模型生成索引预测值, 将查找范围确定在与预测值距离固定的有限范围中. 现有的大量研究中, 研究者们按照所处理数据的维度特点对学习索引进行分类.

1.2.1 一维学习索引

将一维值输入模型并完成训练, 因一维值便于排序, 所以实现便捷且高效. 2019 年, Wu 等人[31]提出了二级索引 Hermit, 它利用隐藏在列中的软函数依赖关系构建了二级索引, 这样的结构有效地提高了与非主键特征相关的查询操作性能. Hermit 使用分层回归搜索树 (tiered regression search tree, TRS) 挖掘隐藏在数据列之间的依赖关系, 为索引键的访问清除了冗余结构, 同时因为 TRS 是机器学习强化的数据结构, 可以实现数据曲线的快速拟合, 自适应地动态捕获列数据的相关性和异常值. 2020 年, Kipf 等人[32]为解决学习索引构建速度较慢的问题, 介绍了一种学习索引 RadixSpline, 使得数据可以在单次传递中被构建, 并且在空间占用和查询性能方面具有突出优势, 通过实验评估表明该结构在各类数据集中均得到较好结果. 同年, Tang 等人[33]提出了专为快速查询而设计的并发有序索引 XIndex, 该索引同样使用学习模型来优化索引效率, 其能够利用细粒度同步和新的两阶段压缩方案有效地处理并发写入而不影响查询性能, XIndex 还能根据运行时工作负载特征调整其结构以支持动态工作负载. Wang 等人[34]则提出了一个并发学习索引 SIndex, 它可以作用于可变长度的字符串键, 解决了学习索引多集中在整数键

工作负载而忽略了可变字符串键的问题, SIndex 使用共享前缀对键进行分组, 并使用每个键的唯一部分进行模型训练, 有效降低了模型推理和数据访问的成本. Ding 等人[35]提出了一种基于内存可更新的学习型索引方法 ALEX. ALEX 通过引入固定数据布局和就地插入策略, 解决了传统索引在动态插入操作中的效率问题. ALEX 采用自适应的 RMI 结构, 能够处理新键的插入操作, 并且通过在初始化时执行根模型来对键空间进行分区. 每个非根节点被分配一个固定数量的分区, 当分区大小合适时, 叶节点以间隙数组或压缩内存数组的形式创建以支持插入操作. Ferragina 等人[36]提出了一个有最坏情况保证的学习型索引结构 PGM-index, 采用分段线性模型并通过增量缓冲区插入策略支持动态更新. PGM-index 通过计算最优分段并递归构建, 实现了稳定的查询性能. Marcus 等人[37]将 RMI 与其他几种经典索引结构进行对比. 2021 年, Zhang 等人[38]提出了 CARMI (cache-aware recursive model index) 索引, 固定每个节点大小, 确保数据查询的内存访问次数为最小最优值, 有效提升查询效率的同时, 具有更相似的空间使用量. 同年, Wu 等人[39]提出了一种精确的键到位置映射学习索引 LIPP, 通过避免误预测后的局部搜索来提升查询性能, 其核心在于采用核化线性模型确保映射均匀分布, 并利用最快最小冲突度算法 (fastest minimum conflict degree, FMCD) 优化冲突处理. 与 ALEX 相似, LIPP 也采用了就地插入策略, 在动态数据更新时展现出高效性. 2023 年, Li 等人[40]利用线性回归模型构建了一种基于数据分布的索引结构 DILI, 保证了动态数据分布下的高效就地插入.

1.2.2 多维学习索引

多维学习索引与一维学习索引最大的不同在于排序难度显著提升. 在一维学习索引中, 只有单一的属性值, 可以简单地对数据对象进行排序, 但多维数据存在逻辑顺序不强的问题, 这会影响学习模型对查找键预测的准确性. 现有的多维学习索引可以划分为: 映射一维、网格划分以及自定义布局. 在一维学习索引已经奠定坚实基础的情况下, 研究者们认为对于多维数据的处理同样可以先转化为一维数据, 在此一维序列上建立学习索引, 根据索引预测值在多维数据的误差范围内展开搜索.

2019 年, Wang 等人[41]提出了 ZM 索引, 借助 Z-order 空间填充曲线对移动对象数据进行降维, 并得到与空间位置关系相关的有序数列, 构建机器学习模型来学习数据分布, 预测查询键位置. ZM 索引有效提升了查询效率和存储占用率, 但其在实现查询过程中会对大量冗余点执行操作. 为解决这一问题, 2020 年, Davitkova 等人[42]提出了多维度学习 (multidimensional learned, ML) 索引, 由两个重要组件组成, 首先是索引构建, 使用聚类算法得到参考点, 基于距离函数和参考点将数据点降至一维, 得到排列后的一维值; 然后使用递归层次模型中的线性回归模型对排列后的一维值分布进行学习. 实验证明, 此方法在范围查询时具有优势. 2022 年, Wang 等人[43]基于动态框架提出时空跳跃连接模型 (spatio-temporal skip-connection model, STSM), 将时间和空间信息正确地结合起来. STSM 中包含时间模块和空间模块, 以及一个跳跃连接到原始输入融合时间、空间和全球信息的数据. 对比实验结果表明, STSM 不仅优于单独的时间或空间模块, 而且比其他传统方法预测更准确.

将多维数据映射为一维的方法, 舍弃了部分空间特征, 为解决这一问题, 产生了网格划分多维数据的方法. 2020 年, Ding 等人[44]在 Flood 基础上提出了 Tsunami 索引, Tsunami 在构造的网格树的各分支中, 构造增强网格对与该分支区域内相关的查询操作进行性能优化. 2021 年, Zhang 等人[45]提出了基于网格索引的空间插值函数 (spatial interpolation function based grid index, SPRIG), 采用了最近点剪枝技术和基于枢轴的过滤来提高最近邻问题的查询性能, 通过在真实的数据集上实验得到, SPRIG 在查询时所用时间比 ZM 索引高出一个数量级, 在索引建立、范围查询和 KNN 查询方面分别比 Flood 快 2.7 倍、3 倍和 9 倍, 但是消耗的空间更多.

虽然上述方法在查询时间和空间成本上都进行了一系列的优化, 但都不支持插入, 索引插入的位置将直接影响查询性能. 自定义数据布局的含义是在初始多维数据的基础上, 更改数据的布局, 以解决索引插入的问题. 2020 年, Li 等人[46]提出了 LISA (learned index structure for spatial data), 在外存中将任意数据集布局为可查询状态, LISA 使用机器学习模型建立可支持动态更新的数据布局, 实现了数据更新、插入和删除功能. 同时, 其还支持范围查询, 并结合范围查询实现了最近邻查询, 通过实验可知, LISA 的 I/O 消耗仅为 R 树的 80% 左右, 同时占用磁盘存储空间仅为 R 树的 90%. 与 LISA 相似, Qi 等人[47]同年提出一个基于磁盘的可更新学习型空间索引 RSMI, 支

持动态插入. RSMI 通过采用基于排序的排名空间映射, 将多维点映射到一维空间, 从而克服了 Z-order 空间填充曲线的经验累积分布函数不均匀间隙问题. RSMI 将数据点按升序排列, 按照预定义的块大小将数据打包到一个块中, 并且在数据集较大的情况下, 采用递归分区方法, 每个分区会训练一个机器学习模型来进行映射. 这些模型将搜索关键字映射到磁盘块 ID, 从而加速查询过程. 2023 年, Ding 等人[48]提出基于基数表、样条点和倒排列表技术构建的学习索引, 以有效地处理空间文本数据, 并使用 Morton 编码将高维坐标转换为一维坐标. 同时, 为了实时处理数据的插入、删除和更新, 采用间隙数组存储底层数据, 研究者们还设计了以样条点为单位的空间重分配策略, 并基于该索引提出查询处理算法, 高效处理不同的空间关键词查询. 同年, Li 等人[49]提出的 BMT 索引克服了传统空间填充曲线 (space-filling curve, SFC) 方法的局限性, 传统的 SFC 使用固定的位合并模式, 无法考虑数据分布或查询工作负载的变化. 其采用树形结构, 每个叶子节点对应一个子空间, 通过为不同维度的每个内部节点分配比特值来从根到叶的路径创建位合并模式. 为了提高 BMT 的构建效率, 采用了强化学习技术来学习最优的构建策略, 并将贪心策略与蒙特卡洛树搜索结合使用. 同年, Gao 等人[50]提出的 LMSFC 则聚焦于学习一个给定数据和查询工作负载的参数化单调 Z-order. 通过学习特定实例的最优参数, LMSFC 能够最小化查询处理成本. 优化问题通过贝叶斯优化方法来求解. 优化后, 采用基于密度的成本函数将数据点更高效地打包到磁盘页面中, 尽量减少 MBR 的空白空间. 然后, 使用如 PGM-index 等学习索引来查找与给定 Z 值对应的页. 与其他学习索引类似, LMSFC 通过原地插入策略支持动态插入操作. BMT 和 LMSFC 都在多维数据索引领域提供了先进的优化技术, 能够根据特定数据和查询分布动态优化数据布局和查询处理操作, 显著改善了查询性能.

学习索引作为数据库研究热点已经取得了一系列的成果, 但目前关于多维学习索引的动态更新、支持处理更多查询类型、与图数据库的结合等方面还有许多关键问题尚未解决, 需要更进一步的探索研究[51].

1.2.3 数据划分方法

均匀划分策略是一种常见的方法[52], 通过将数据均匀地分配给学习模型再进行拟合. 这种方法没有将数据特点和模型融合, 较难得到最优划分结果. ALEX 提出了一种自上而下划分策略, 通过多次访问数据集依靠代价模型估计进行划分. 还有学者提出了贪心算法[53]进行数据划分, 旨在降低模型的误差范围.

1.3 查询算法

移动对象数据库中存放的数据量规模巨大, 且数据的表现形式比关系型数据库更为复杂. 所以, 移动对象数据库在存储、索引以及查询等方面的技术难度比关系型数据库更大. 同时, 因为移动对象的数据含有时间、空间属性, 所以在移动对象的查询处理方面, 需要指明时间和空间请求, 现有针对移动对象的经典查询技术可以分为以下几个类型.

1.3.1 范围查询

移动对象数据库中最基本、应用最广泛的查询类型之一, 由用户给定一个时间区域和一个空间区域, 数据库查询得到所有满足该时空范围条件的移动对象并返回给用户. 范围查询在许多智慧城市应用中均有体现, 例如交通轨迹识别. 2019 年, Yu 等人[54]提出一种分布式混合索引, 将全局网格索引和局部 VR 树索引相结合, 将其部署在服务器集群上, 以维护海量移动对象数据和连续的范围查询. 2023 年, Baride 等人[55]为解决难以选择合适距离阈值的问题, 提出了一种用于共置模式挖掘的范围查询, 识别了搭配模式的若干结构属性, 使得查询条件从单个距离阈值更改为距离间隔, 并通过使用多类型数据集, 证明该算法的优越性.

1.3.2 最近邻查询

GPS 技术和嵌入式设备的迅速发展使得最近邻查询受到越来越广泛的关注, 它是指用户给定一个需查询的移动对象, 数据库在当前数据集中查询与该指定移动对象距离最近的指定个数对象, 作为查询结果返回. 最常见的最近邻查询是 k 近邻查询 KNN, 也就是查询距离指定移动对象最近的 k 个移动对象, 其中 k 为随机指定的数字. 例如, 乘客使用打车软件时, 应用程序会根据查询位置返回距离该乘客最近的网约车并自动派单, 以保证服务质量. 2016 年, Yi 等人[56]研究移动对象在近似 k 近邻查询中保留其位置和查询隐私的问题, 提出了建立在 Paillier 公钥密码系统上的通用解决方案, 可以应用于基于位置的私有查询的多个离散类型属性. 2021 年, Levchenko 等人[57]提

出了用于大型时序数据库上评估 k 近邻查询的并行解决方案, 根据查询质量和时间性能的各种度量对其进行比较, 并提供了使用应用程序数据特征来确定为该应用程序选择哪种算法及参数的工具, 并在 Spark 上实现, 在节点集群上进行了评估实验.

1.3.3 相似轨迹查询

根据查询轨迹特征, 在对轨迹进行降维之后, 使用标记特征的索引方法, 查询符合相似度定义的轨迹数据. 相似轨迹查询包含基于阈值的查询, 即给定一个相似值, 求解轨迹数据集中所有与查询对象的相似值小于阈值的轨迹. 还包括 Top-k 相似性查询, 即对基于阈值的相似性查询结果进行排序并进行筛选. 因为相似轨迹查询所用实验数据集较为庞大, 查询代价较高, 现有轨迹相似性查询算法主要基于分布式集群环境. 2006 年, Frentzos 等人[58]通过定义相似度的度量方法解决移动对象相似性搜索的问题, 提出了近似方法降低其计算成本, 并开发了新的度量和启发式方法来支持移动对象数据库中的 k 相似轨迹搜索. 2017 年, Xie 等人[59]提出分布式查询框架, 用于处理大量移动对象数据上的轨迹相似性搜索, 在分布式数据处理引擎 Spark 中实现该框架, 支持豪斯多夫距离和弗雷歇距离查询. 此外, 还有连续查询、并行查询、最短路径查询等各类查询, 不同的应用场景和用户需求催生了不同的查询方法.

2 问题定义

基于非均匀网格降维构建学习索引, 为验证查询性能, 引入了 3 种查询方法: 范围查询、最近邻查询与相似轨迹查询, 本节将对数据降维、学习索引及查询算法的相关定义进行说明. 本节定义涉及符号说明如表 1 所示.

表 1 缩略符号定义表

符号	说明	符号	说明
MP	移动对象	*Loc*	移动对象位置点
t	时间	*x*	经度
y	纬度	*f*	位置降维函数
RMP	降维轨迹数据	*RangeMO*	范围查询
Bbox	矩形边界框	*NUGC*	非均匀网格降维算法
CDF	累积分布函数	P_i	移动对象查询点
NNMO	最近邻查询	*Loc-MP*	轨迹相似性度量
MP-MP	轨迹相似性度量	*Traj-Sim*	轨迹间的相似性度量
MP-Sim	轨迹点距离另一轨迹的相似性度量	*Traj-SQ*	轨迹相似性查询

2.1 数据降维基础定义

定义 1. 移动对象 *MP*.

一个移动对象由一组移动对象位置点和时间组成, 在数据库系统中用数据类型 *MP* 来表示:

$$MP = \{mp_1, mp_2, \ldots, mp_n\} \tag{1}$$

$$mp = \langle (t_1, Loc_1), (t_2, Loc_2), \ldots, (t_n, Loc_n) \rangle \tag{2}$$

$$Loc = \{(x, y) \mid x, y \in \mathbb{R}\} \tag{3}$$

其中, *Loc* 为移动对象位置点, x 为经度, y 为纬度, t 为时间, 移动对象位置点按照时间顺序排列.

定义 2. 位置降维函数 *f*.

$$f : Loc \to Loc' \tag{4}$$

利用非均匀网格降维算法, 针对数据分布情况, 通过网格密度计算数据密集程度, 并根据网格密度采取平衡措施, 将移动对象位置点 *Loc* 映射为一维编码后得到对象 *Loc*′. 非均匀网格降维算法具体内容详见第 3 节.

定义 3. 降维轨迹数据 *RMP*.

经降维函数 *f* 作用后得到:

$$RMP=\{mp'_1,mp'_2,\ldots,mp'_n\} \tag{5}$$

$$mp'=\langle(t_1,Loc'_1),(t_2,Loc'_2),\ldots,(t_n,Loc'_n)\rangle \tag{6}$$

其中, mp' 由降维后的移动对象位置点与时间组成二元组, 按照时间顺序排列而成.

2.2 数据查询相关定义

定义 4. 空间范围查询 *RangeMO*.

给定一个空间查询矩形框 *Bbox*, 返回经过该区域的移动对象, 空间范围查询 *RangeMO* 定义为:

$$RangeMO(Bbox,MP)=\{MP'\mid MP'\subseteq MP,\ \forall mp\in MP',\ \exists Loc\in mp\wedge Loc\in Bbox\} \tag{7}$$

定义 5. 最近邻查询 *NNMO*.

非均匀网格降维算法可以使相邻单元格可以被快速定位, 具体说明可见第 5.2 节, 所以基于学习索引的最近邻查询问题被简化为了多个点查询, 定义如下:

$$NNMO(P_Q,MP)=\{P_j\mid P_j\in Query(P_{Q\text{-neighbor}}),\ \forall P_i\in Query(P_{Q\text{-neighbor}}),\ dis(P_j,P_Q)\leqslant dis(P_i,P_Q)\} \tag{8}$$

其中, P_Q 是最近邻查询的查询键, $P_{Q\text{-neighbor}}$ 是对查询键所在单元格相邻一圈网格的点查询操作, 该函数首先定位查询键的 P_Q 位置, 并获取该网格内存储的移动对象点数组, 依次遍历并计算得到最近邻居, 若该区域只有一个网格, 则在网格中寻找最接近该数据点的邻居点; 否则对其所在网格外围一周网格内的数据点进行遍历计算, 以查找最近邻居.

定义 6. *Loc-MP* 轨迹相似性度量.

移动对象是一组按照时间顺序排列而成的位置点, 通过离散采样表示对象点连续移动的趋势. 为了在多维空间中较为方便地比较不同轨迹的相似程度, 定义了一种 *Loc-MP* 轨迹相似性度量方式.

给定移动对象位置点 *Loc*, 移动对象 *MP*, 则 *Loc-MP* 轨迹相似性为:

$$Sim(Loc,MP)=\max_{P_i\in MP}(dis(Loc,P_i)) \tag{9}$$

其中, $dis(Loc,P_i)$ 为移动对象位置点 *Loc* 与移动对象 *MP* 中第 i 个位置点的欧氏距离, 由此可知将给定点 *Loc* 与给定移动对象点 $P_i\in MP$ 的最大欧氏距离值定义为 *Loc-MP* 轨迹相似性.

定义 7. *MP-MP* 轨迹相似性度量.

考虑到对于不同的轨迹而言, 采样的频率和起始终止时间并不总是完全一致, 所以任意两条轨迹间的相似性度量需要分别选取轨迹代表点, 以代表点的相似程度来衡量轨迹的相似程度. 代表点的选择可以基于多种策略, 如等间隔采样、基于特征的采样、基于聚类的采样等, 根据轨迹的不同特性进行相应选择, 为确保查询效率, 选取等间隔采样的方式, 在固定时间间隔内选取位置点平均值作为代表点.

给定两条起始终止时间相同的移动对象片段 MP_1 和 MP_2, 通过等间隔采样与均值计算分别得到 n 个代表点, MP_1 和 MP_2 两条轨迹之间的相似性为:

$$MP\text{-}Sim(MP_1,MP_2)=\frac{\sum_{Loc_i\in MP_1}Sim(Loc_i,MP_2)}{2\times n}+\frac{\sum_{Loc_j\in MP_2}Sim(Loc_j,MP_1)}{2\times n} \tag{10}$$

$$Traj\text{-}Sim(MP_1,MP_2)=\mathrm{e}^{-MP\text{-}Sim(MP_1,MP_2)} \tag{11}$$

其中, $MP\text{-}Sim(MP_1,MP_2)$ 是根据定义 7 计算的轨迹中所有给定点距离另一轨迹的相似性度量, 即两条轨迹间的相似性度量, $Traj\text{-}Sim(MP_1,MP_2)$ 则是对轨迹间的相似性度量值进行归一化处理, 从公式 (11) 易知 $Traj\text{-}Sim(MP_1,MP_2)\in(0,1]$, 且值越大, 轨迹相似性程度越高.

定义 8. 轨迹相似性查询 *Traj-SQ*.

轨迹相似性查询为给定轨迹 MP_Q 计算其与其他所有轨迹的相似性程度, 并返回相似性程度最高的轨迹 MP_j, 定义如下:

$$Traj\text{-}SQ(MP_Q)=\{MP_j\mid\forall MP_i\in MP,\ Traj\text{-}Sim(MP_j,MP_Q)\geqslant Traj\text{-}Sim(MP_i,MP_Q)\} \tag{12}$$

其中, MP_Q 是轨迹相似性查询的查询键, $Traj\text{-}Sim(MP_j, MP_Q)$, $Traj\text{-}Sim(MP_i, MP_Q)$ 是定义 8 中轨迹相似性程度计算操作, 根据计算得到移动对象 MP_j 与查询键 MP_Q 的轨迹相似性程度最高, 为相似轨迹查询结果.

3 非均匀网格降维算法

移动对象数据查询时, 需要先进行数据降维, 现有降维算法中大多保留数据局部结构特征. 移动对象数据的空间位置有均匀分布和非均匀分布两种情况, 且具有随时间变化、多粒度表示等特征, 仍是研究的难点. 因此, 提出了一种均匀网格降维算法. 在处理移动对象数据时, 均匀网格是传统的划分方法之一, 假设空间可以均匀地划分为多个网格单元, 以便于后续的数据分析和处理. 然而, 均匀网格在处理具有高度不均匀分布特征的移动对象数据时存在明显的局限性. 具体来说, 均匀网格假定每个网格单元包含相同数量的数据点, 但移动对象往往呈现出不均匀的空间分布. 在某些区域, 移动对象的聚集程度较高, 而在其他区域则可能没有任何对象. 这种不均匀分布会导致空间的浪费, 尤其是在一些网格单元内没有移动对象经过时, 造成了不必要的内存和计算资源的消耗, 还影响了数据处理的效率和准确性. 单纯依赖均匀网格方法会低效地利用空间资源, 并且未能有效捕捉数据的真实分布. 因此, 根据移动对象的分布特点, 进一步优化得到非均匀的网格降维算法.

3.1 均匀网格降维算法

常见的墨卡托投影在高纬度地区出现严重畸变, 限制了其在全球范围内的精确应用. 通用横轴墨卡托投影虽然适用于大范围区域, 但在跨带数据处理时需要额外的转换, 增加了复杂性和误差累积的风险. 而等距圆锥投影虽然在一定区域内表现良好, 但其精度仅限于特定区域, 难以适应更广泛的地理范围[60]. 而高斯-克吕格投影[61]在局部区域内表现出更高的精度和计算效率, 能够有效减少变形, 尤其适用于需要高精度的区域性应用. 由于其能够在较小范围内保持最小畸变, 便于后续基于 GeoHash 编码的空间数据降维处理, 因此成为降维过程中合适的选择. 将移动对象点的坐标点按照高斯-克吕格投影方式进行转换, 通过编码生成二进制字符串, 以实现数据降维. 均匀网格编码算法首先通过遍历得到移动对象轨迹的区域边界, 将该区域矩形递归地进行二等分得到更多的矩形, 并按照编码长度要求终止划分. 通过均匀网格编码算法得到的编码存在定理 1 所述性质.

定理 1. 给定移动对象位置 Loc_1 和 Loc_2, 均匀网格编码可计算出 $Code(Loc_1)$ 和 $Code(Loc_2)$. 若 $Pre(Code(Loc_1)) = Pre(Code(Loc_2)) = pre$, $|Code(Loc_1)| \geqslant pre$, $|Code(Loc_2)| \geqslant pre$, 则 Loc_1 和 Loc_2 均在编码为 pre 的网格中.

证明: 对于具有相同编码前缀 pre 的两个数据点 Loc_1 和 Loc_2, 假设这两个数据点不在同一个网格中. 因为 $Pre(Code(Loc_1)) = Pre(Code(Loc_2)) = pre$, 所以两个数据点 Loc_1 和 Loc_2 经过二分编码后均被划分到编码为 pre 的网格. 这与不在同一网格的假设矛盾, 不成立. 如果两个数据点前缀相同, 那么它们一定都在公共网格中.

均匀网格编码算法的具体实现步骤为: 首先将移动对象点的坐标存储在 MP 中; 其次将所有数据点进行高斯-克吕格投影, 找出投影区域最小点和最大点; 最后对每个数据点编码, 奇数位为经度编码, 若当前数据点经度小于平均值, 则编码左移 1 位, 末位赋 0, 否则赋 1. 将经度范围更新为当前数据点所在网格的经度范围值, 偶数位为纬度编码, 按照上述方法类推至编码长度达到给定精度要求. 根据上述描述与定理 1 的编码特点, 两个移动对象点距离越小, 通过网格降维后的公共前缀就会越长. 当编码 $Code_1$ 以编码 $Code_2$ 为前缀, 则 $Code_1$ 对应的移动对象点一定落在前缀码 $Code_2$ 所对应的区域中.

3.2 非均匀网格降维算法

移动对象具有分布广、聚集多、非均匀的特点. 如果不考虑数据特点, 对所有数据均采用均匀网格划分会增加存储空间, 降低处理效率. 本文通过网格密度计算数据密集程度, 并根据网格密度采取平衡措施, 针对数据分布情况提出了非均匀网格降维算法. 一方面避免了存储大量空网格和低密度网格, 节省编码所需的存储空间; 另一方面在查询阶段, 有效减少了读取操作代价, 对不符合范围条件的空网格和低密度网格进行有效过滤.

3.2.1 网格密度

为便于介绍网格密度值 (grid density, *Dens*), 先给出相关变量符号, 如表 2 所示.

表 2 均匀网格符号定义表

符号	说明
M	移动对象总数
Ul	单元格长度
Uw	单元格宽度
Num	单元格总数
PN_i	网格 i 中移动对象点的个数

给定移动对象的数据集中共有 M 个移动对象采样点, 区域边界均匀划分后得到 Num 个均匀网格, 第 i 个网格的密度值 $Dens_i$ 定义为:

$$Dens_i = \frac{Num \times PN_i}{M} \tag{13}$$

定义网格密度值计算网格中移动对象点的密集程度. 图 1 给出了具有不同级别密度值的网格.

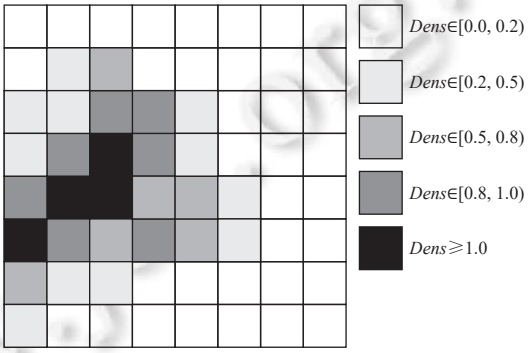


图 1 移动对象均匀网格分布示意图

3.2.2 非均匀网格构建

观察移动对象在均匀网格中的布局, 可以看出许多网格中仅包含少量移动对象点, 有些网格甚至没有. 如果区域中移动对象数据均匀分布, 每个网格的 $Dens$ 值为 1. 实际场景下不同网格的 $Dens$ 值差距显著, 图 1 给出了真实数据集下的网格密度值.

本文提出了一种名为非均匀网格降维算法 $NUGC$ 的方法, 旨在平衡每个单元格中移动对象点的数据量. 非均匀网格的数据划分如图 2 所示.

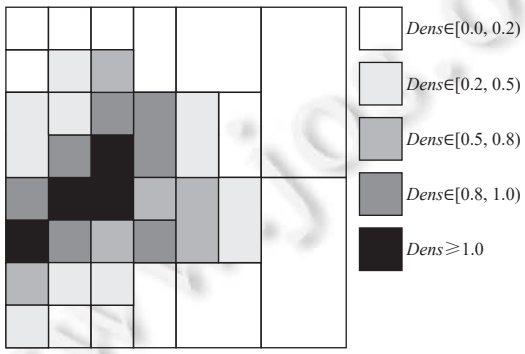


图 2 移动对象非均匀网格分布示意图

该算法首先设定一个网格密度阈值, 并对那些未达到此阈值的网格进行合并处理. 具体步骤包括, 首先计算网格密度平均值, 通过移动对象点总数 M 除以均匀网格总数 Num 得到; 其次, 将这个平均值作为决定是否合并网格

的标准. 这种方法有效地合并了空网格和数据点极少的网格, 从而节约了存储空间. 通过采用交叉编码技术, 合并网格时不需要更改网格标识或创建新的列表来记录合并动作, 将二进制编码右移一位从而与邻近网格合并. 根据定理 1, 因为合并后的网格代表上一级的公共网格, 所以提取公共前缀作为新的编码.

鉴于所有网格编码都是唯一的二进制字符串, 并且具有递归前缀, 可以在降维后的数据上构建 B 树. 定义标记位, 0 表示左子树, 1 表示右子树, 每个叶节点对应一个网格, 一个网格对应多个数据点.

在处理移动对象的动态更新时, 需要将新出现的移动对象点整合到现有的非均匀网格中, 以实现降维. 更新过程中的降维步骤包括: 首先确定移动对象所在网格; 然后对比该网格更新后的密度值与预设的最高值, 如果密度值≤最高值, 选择当前网格编码作为降维后的编码; 否则, 再次划分网格并使用一个 8 位的字段记录两次划分的局部编码. 如果划分次数超过 8 次导致局部编码溢出, 进行全局更新, 重新构建非均匀网格. 非均匀网格算法能够支持更新操作, 适用于更新频率不高、密度分布与历史移动对象相似的场景.

3.2.3 时间复杂度分析

由算法 1 可知在初始的均匀网格编码算法中, 采用了双层循环结构, 外层循环遍历移动对象点的投影值, 获取每个点在两个维度上的空间位置, 内层循环则负责从最低位到最高位依次计算并赋值编码, 进行位移操作. 均匀网格算法的时间复杂度为 $O(M\times L)$, 其中 M 为移动对象点个数, L 为编码长度.

在非均匀网格编码降维算法中, 原始实现涉及两个主要循环, 外部循环遍历所有移动对象计算每个均匀网格的密度; 第 2 个循环则逐个检查均匀网格的密度是否达到阈值, 若未达到, 则执行网格合并操作, 并记录合并后网格的更新值. 重复该过程直到所有网格合并满足相应条件. 该方法的时间复杂度为 $O(M)+O(Num)$, 其中网格数 $Num=2^L$.

为提升降维效率, 将需要合并的网格中轨迹点一维编码右移一位, 设置原始轨迹点标签属性, 表示该点所在网格的移动对象点总数. 遍历一次移动对象点, 完成网格合并, 合并过程采用位运算, 因此时间复杂度降低到 $O(M)$.

算法 1. 非均匀网格降维算法 *NUGC*.

输入: 网格密度集合数组 *Dens*, 移动对象点数组 *MP*, 移动对象点个数 *M*, 均匀网格总数 *Num*, 一维编码长度 *L*;
输出: 一维非均匀编码 *C*.

```
1. Dens ← ∅
2. C ← 均匀网格降维算法 (MP)
3. average(Dens) ← get(Num, M)
4. for each iter ← 1 to Num do
5.    C_i ← 遍历网格得到每个网格编码
6.    Dens ← 计算每个网格密度值
7.    while Dens_i < average(Dens) do
8.       C_i ≫ 1
9. return 降维后的一维非均匀网格编码数组 C
10. function 均匀网格降维算法 (MP)
11. for each iter ← 1 to |MP| do
12.    GKpoints[i] ← GKprojection(MP[iter])
13. P_min, P_max ← GoThrough(GKpoints)
14. for each iter ← 1 to M do
15.    get X_median and Y_median
16.    for each iter ← 1 to L do
17.       计算并获得数据点的经度和纬度编码
```

因为移动对象往往包含多个位置点, 尤其是时间周期长的对象, 所以时间复杂度 O(M) 代价依然很高. 例如, 采样频率为 10 s, 一天的移动对象具有 M=8640 个位置点. 为此, 设计了两种优化方法: (1) 前缀编码分组; (2) 子轨迹估计表示. 采用前缀编码分组使得所有移动对象点根据前 5 次二分后所在网格的编码前缀被分配到不同的数组中. 通过数组规模判断聚集程度高或低的网格, 避免遍历密度显著高于密度阈值的分组, 降低访问开销. 子轨迹估计表示通过最小矩形框 *Bbox* 表示时空范围, 通过不同时间区间划分, 用 *Bbox* 的中心点代表指定区间内的所有数据点, 减少移动对象点数量, 将时间复杂度从 O(M) 降低到 O(M/m), 其中 m 是子轨迹 *Bbox* 中含轨迹点数量的平均值.

4 非均匀网格学习索引

学习索引的核心理念在于索引不再是静态的数据结构或文本文件, 而是一个可以预测查询键位置的机器学习模型, 模型可以被训练, 因而数据库索引也可以被训练和学习. 但是现有的学习索引多面向静态只读数据或一维数据, 本工作所构建的非均匀网格降维学习索引面向移动对象, 基于通过 *NUGC* 降维后的值进行查询键预测. 同时, 考虑到移动对象位置的更新特点, 如果使用传统的学习索引, 每插入新的位置数据就需要重新训练索引模型, 会导致查询成本显著提高, 因此 NUGC_LI 还支持插入、更新和删除操作. 为应对数据更新与新增数据引起的冲突, 节点内预留了足够的空隙, 并设定节点阈值, 以便在数据插入时能够直接在预测位置附近进行微调, 减少因全局重训练带来的开销. 采用局部数据平移以及扩展分裂两种策略, 以避免每一次数据更新时都进行大范围的模型重训练, 从而降低更新成本, 提高索引的实时性. 基于预测位置误差控制方法, 确保更新过程中的查询准确性, 并通过统计误差信息动态调整节点分裂策略, 从而兼顾高效查询和低更新延迟, 能够初步满足在线场景的需求.

4.1 相关定义

定义 9. 误差区间 *ErrINR*.

$$ErrINR = [PrePos - err, PrePos + err] \tag{14}$$

其中, *PrePos* 为学习索引预测的查询值所在数组中的位置, *err* 为误差值, 每次通过学习索引预测到查询值的位置后, 在误差区间内用二分查找算法进行遍历搜索, 以保证查询准确性.

定义 10. 非均匀网格学习索引 NUGC_LI.

NUGC_LI 学习索引由 3 部分模型组成, 第 1 部分由一个线性回归模型构成的根节点模型; 第 2 部分是若干个可扩展的内部节点, 每个节点中存在一个线性回归模型; 第 3 部分是叶子节点, 也可以被视为数据节点模型.

$$\text{NUGC\_LI} = \{Root_{\text{model}}, Expandable\ Int_{\text{model}}, Leaf_{\text{model}}\} \tag{15}$$

指定一个查询键, NUGC_LI 中的第 1 部分是一个回归模型用来预测下一部分应选择哪个子模型, 第 2 部分的所有内部节点都是回归子模型, 用来预测符合查询键所在范围的叶子节点, 第 3 部分是叶子节点用来预测有序数组中查询键所在的准确位置. 这 3 层模型均是选择适应数据分布的机器学习模型, 通过数据训练得到较高的预测准确率.

定义 11. 学习索引节点阈值 *NodeTHR*.

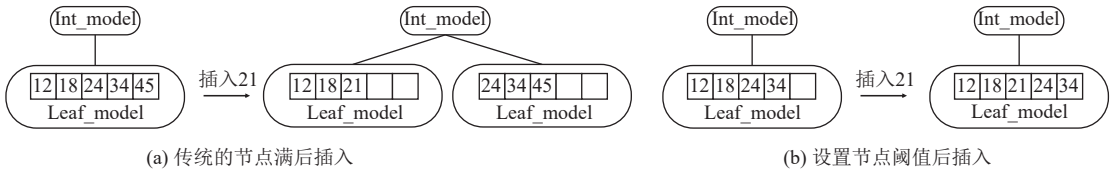
因为学习索引支持数据插入及更新, 所以为每个节点设置节点阈值, 当该节点中包含的数据位置信息已经超过阈值, 则该节点模型需要进行分裂, 分裂机制见第 4.4 节索引插入机制. 节点阈值计算公式如下:

$$NodeTHR = \frac{M}{Count(LR_{\text{leaf}})} \times X,\ X \in (0,1) \tag{16}$$

其中, M 为移动对象点个数, 即查询数据集中对象总数, LR_{leaf} 是作为叶子节点的回归模型, $Count(LR_{\text{leaf}})$ 则是作为叶子节点的回归模型的个数. 同时, 因为索引在更新和插入阶段, 为保持移动对象的空间位置关系会造成其他数据的移动和存储位置变换, 键值及指针数组中需要保留一定的间隙空间, 因此需乘以一个变量 X 使得节点总是未被完全插满, 为数据的移动操作提供空间.

在移动对象数据库中, 例如城市车辆位置跟踪的场景下, 数据需要频繁更新和插入. 传统索引结构在节点满时通过分裂来适应新的数据插入, 在静态数据集上表现良好, 但在当前问题场景下, 调低阈值允许索引在构建阶段就

为未来的数据插入预留空间, 新数据可以直接在节点内部进行排序, 无需触发节点分裂, 减少了数据移动和分裂的代价, 如图 3 所示.



(a) 传统的节点满后插入 (b) 设置节点阈值后插入

图 3 数据插入示意图

这一方法保证了与现有技术方法的一致性, 保留了现有的阈值分裂机制, 并在第 6.2.1 节根据本文的问题场景进行了合适的阈值选择.

定义 12. 叶子关键值范围 *LeafBound*.

对每个叶子节点设置关键值数值区间:

$$LeafBound = [key_{\min}, key_{\max}] \tag{17}$$

这是因为在范围查询过程中, 基于非均匀网格降维的一维值具有处于同一网格中的点前缀相似的特点, 所以在形成对数据的约束条件之后, 需要遍历有序列表中所有前缀符合约束条件的键值, 这会导致冗余点也被纳入. *LeafBound* 的设置可以直接通过核对区间值剔除查询冗余点, 有效避免了时间浪费. 上述定义中 $key_{\min}$ 是该节点中最小查询键, $key_{\max}$ 是最大查询键, 在范围查询遍历过程中, 若叶子节点在粗过滤范围内, 则细化查询过程中先比较 *LeafBound* 与实际查询区域 *Bbox* 是否存在包含或交叉关系, 若有则执行遍历, 若无则直接跳过当前叶节点, 有利于提高范围查询效率.

4.2 非均匀网格学习索引结构设计

学习索引是一个具有层次结构的递归模型索引, 根据数据特点的需要选择对应的机器学习模型, 并将这些模型组织成树形结构, 该树的分支数与深度均可自定义设置, 树的根节点中包含一个具有预测功能的学习模型, 通过学习数据分布预测当前查询在哪一个孩子节点中, 然后跳至该预测节点, 迭代此过程直到进入叶子节点, 叶子节点中存有最终的预测模型.

经典的学习索引为两层模型, 将数据集均匀划分为若干子集后使用机器学习模型学习数据分布, 但是两层模型并不能支持高效的插入和更新, 因为模型需要重新学习插入新数据后的分布情况. 此外, 经典的 RMI 结构在使用神经网络进行数据拟合时, 依靠全连接神经网络对输入的查询键进行学习, 并使用 ReLU 激活函数来进行非线性的数据变换, 最后使用均方差作为损失函数, 但是这种做法会导致极大的模型训练代价, 在执行查询操作前需要大量时间来训练模型. 而 FITing-Tree 的实现使用线性回归模型作为内部节点, 只需要存储线性回归模型的两个参数: 斜率和截距, 降低了索引的存储开销, 且实验证明该索引的性能更优.

综合考虑上述问题, 为实现面向移动对象的高效查询, 提出了非均匀网格学习索引 NUGC_LI, 该索引模型的结构如图 4 所示. 继承了经典学习索引的递归层次模型结构, 基于第 3 节中降维后的有序一维值进行索引, 且在数据划分后, 降维数据的线性特征更为明显, 但使用神经网络来拟合数据间复杂的线性关系是较为困难的, 特别是当神经网络规模小、数量多的时候. 因此选择使用多阶段的回归模型来取代需要多次训练的神经网络, 单调约束效果更佳、所需参数更少、训练时间更短, 能够有效提高索引的构建效率.

第 1 部分的根节点有且仅有一个, 其内部存储了一个线性回归模型和一个指针数组指向孩子节点, 其中存储线性回归模型即使用两个 64 位浮点数存储斜率和截距两个值, 另外还需要一个整数来记录误差值 *err*. 根节点下连接的孩子节点个数必须是 2 的指数幂, 使用线性回归模型计算得到数组位置, 然后得到指向孩子节点的指针. 第 2 部分的内部节点同样包含线性回归模型和孩子指针数组, 并且每个子树的深度会因节点分裂的影响导致各不相同, 因此节点中还需包含记录层数的整型值 *level*.

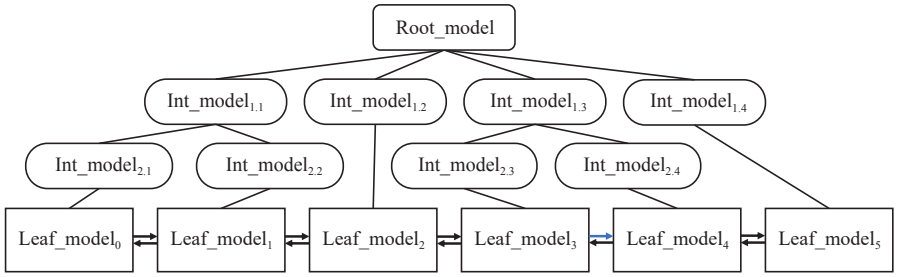


图 4 NUGC_LI 结构图

第 3 部分的每个叶子节点是数据节点, 如图 5 所示存储线性回归模型、键值数组、数据指针数组和一个 64 位 bitmap, 这个线性回归模型将把一个查询键映射为一个数据存储位置. 键值数组与数据指针数组大小相同且一一对应, 且整个数组并非完全填满的, 而是均匀分布元素, 即数组中存在均匀分布的空元素, 这是为了在插入和更新数据阶段, 使得新插入的数据能够更大概率地插到模型预测的位置, 有利于拟合原本的数据分布, 从而提高查询的效率. 同时, 为了能支持数据插入导致的索引更新, 叶子节点还支持扩展分裂, 因此包含一个分裂函数, 该函数的输入参数为预计分裂的孩子节点个数, 为保证分裂过程顺利, 该输入参数必须为 2 的指数幂, 分裂后的节点视具体情况决定是否需要生成新的内部节点.

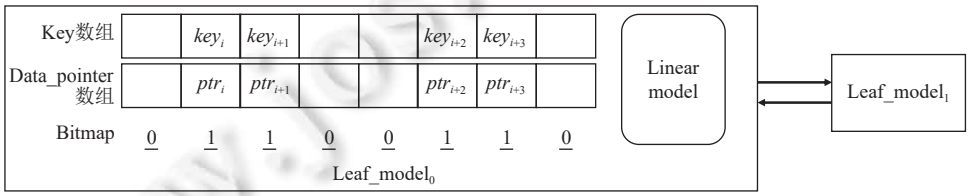


图 5 NUGC_LI 叶子节点图

此外, 为了使得数据节点能正常执行插入、更新等操作, 应充分避免数据节点中的键值数组和数据指针数组被插满, 因此根据实际情况, 对数据节点设置了一个节点阈值 *NodeTHR*, 当数据节点中的数组元素超过该阈值时则开始节点分裂. 叶子节点中的 bitmap 则是用来记录数组的使用情况, 逆序记录数组该位置若已有键值或数据指针, 则该位置为 1, 否则该位置为 0.

考虑到索引的构建是为了提升查询的效率, 叶子节点除了与其父节点相连接外, 还与相邻叶子节点相连接, 便于范围查询操作的执行, 如图 4 所示. 所以叶子节点中均包含指向前一个叶子节点和指向下一个叶子节点的两个指针来保证节点间的顺序连接.

4.3 非均匀网格初始化建立

在明确所需搭建索引模型的各部分结构后, 就可以根据移动对象数据构建出通过学习数据分布实现精准预测的学习索引模型, 如第 3 节所述将移动对象数据通过 *NUGC* 算法降至一维, 并按照编码二叉树的深度优先遍历顺序得到有序数列, 将移动对象数据处理成为符合学习索引构建需求的数据格式. NUGC_LI 使用线性回归模型通过最小化线性函数的平方误差对数据分布进行学习, 模型预测查询键对应的数据记录位置近似于累积分布函数 (cumulative distribution function, *CDF*), 用于描述变量的概率分布. 提出针对移动对象数据的累积分布函数, 给定一个查询键 *key*, 定义对应的数据位置表示为:

$$PrePos = CDF(key) \times Count(key) \tag{18}$$

$$CDF(key) = Pr(random_{key} \leqslant key),\ CDF(key) \in [0,1] \tag{19}$$

其中, *PrePos* 表示有序数据数组中的预测位置, *CDF*(*key*) 是数据估计的累积分布函数, 该函数计算 *Pr*, 其值表示任意键 $random_{key} \leqslant key$ 的概率, *Count*(*key*) 是键值的个数.

CDF 函数存在统计效应, 以随机数据集 (数据集具体内容详见第 6.1 节) 为例进行分析. 如图 6(a) 所示, 针对整个数据集的 *CDF* 曲线趋势平缓且光滑规则, 但是如果仅选择数据集中的一段区间如图 6(b) 就会发现数据趋势具有明显的分段性、随机性, 这就增加了对于数据拟合的难度, 而多阶段的线性回归模型能够更有效地从数据集的完整区域中将预测缩小至数千个区域.

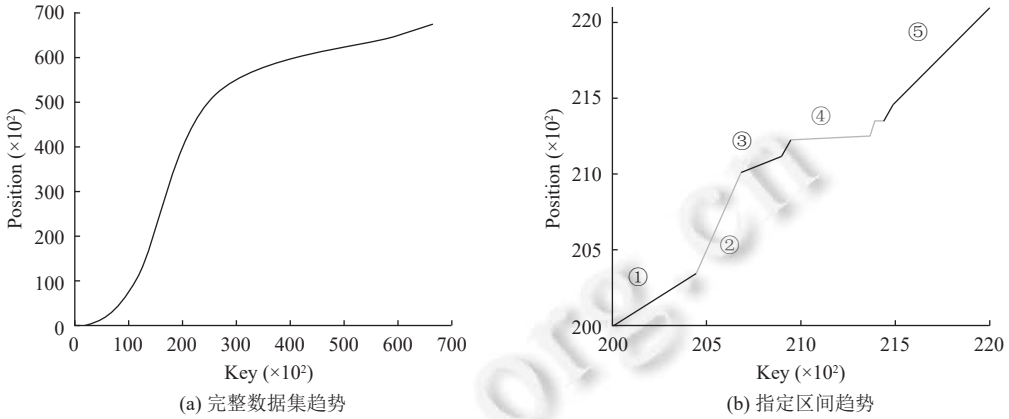


图 6 *CDF* 预测键值位置示意图

NUGC_LI 会根据 *CDF* 的概率分布曲线对数据集进行划分, 这个过程中就会造成第 2 部分内部节点各子树的不同生长形态, 这一步骤与经典的 RMI 学习索引结构有所不同, RMI 对数据集进行均匀划分, 然后对各个数据子集进行模型学习和训练, 但是 NUGC_LI 不追求数量上的平均划分, 会根据数据集的数据特点更加灵活地划分数据. 例如图 6(b) 中的数据区间 [20 000, 22 000], 如果按照均匀划分将生成 [20 000, 20 500]、[20 500, 21 000]、[21 000, 21 500]、[21 500, 22 000] 这 4 个数据节点, 但是在 NUGC_LI 的划分原则中, 因为 [20 500, 21 000] 区间中存在明显的趋势变换, 因此在进行数据划分时, 会将原本的 [20 500, 21 000] 数据节点替换为一个内部节点, 该内部节点存在两个均为数据节点的孩子节点, 分别存储图中的区间②和③, 这样的划分规则能够使得生成的数据节点①②③④⑤大致符合同一线性趋势, 便于线性回归模型准确拟合该节点键值. 按照上述节点生长规律, 如算法 2 所示.

算法 2. 学习索引 NUGC_LI 构建算法.

输入: 键值数量 *num_keys*, 键值数组 *value_keys*, 预训练线性模型 *pretrained_model*, 节点阈值 *NodeTHR*, 代价模型 *cost*;

输出: 基于当前数据构建的学习索引 NUGC_LI.

1. *keys_remaining* ← *num_keys*
2. **while** *keys_remaining* **do**
3. **if** *node.cost* < *best.cost* **then** /*自下而上计算出 *value_keys* 的最优划分*/
4. *num_leaf_nodes* ++
5. *leaf_node_bulkload*()
6. NUGC_LI ← *leaf_node*
7. *find_best_partition_top_down*(*value_keys*);
8. **if** *keys_remaining* > *NodeTHR* 或 *node.cost* > *best.cost* **then** /*判断该节点是内部节点还是数据节点*/
9. *num_internal_nodes*++ /*键值数量超过阈值或代价较高时, 节点设置为内部节点*/
10. **if** 当前子树深度一致, 内部节点分布均匀 **then**

```
11.          node_depth ← compute_level()
12.        node_model ← pretrained_model
13.        node_fanout ← 1 ≪ node_depth
14.        node_children ← allocate_node(node_fanout)
15.        实例化所有子节点并递归
16.        NUGC_LI ← internal_node
17.        更新 keys_remaining
18.    else /*其他情况下节点设置为叶子节点*/
19.        num_leaf_nodes ++
20.        leaf_node_bulkload()
21.        NUGC_LI ← leaf_node
22.        更新 keys_remaining
23. return 基于当前数据构建的学习索引 NUGC_LI
```

在算法 2 中, 学习索引从根节点开始依次向下构建, 在每个节点根据所需存储的数据趋势判断该节点是内部节点还是叶子节点, 其中内部节点的孩子节点数需保持为 2 的幂次方, 且判断时引入代价模型, 防止因为数据的非线性特点导致划分区间过小, 单一子树层度过深; 或者因为数据的线性特点导致划分空间过大, 单一数据节点存储数据量过多, 从而影响索引的查询效率. 在判断完节点类型后, 如果该节点是叶子节点, 则将数据放入该节点, 如果是内部节点, 则指向两个孩子节点, 并将数据按照线性特点划分为两部分, 分别放入两个孩子节点对应的数据节点中, 如此循环往复, 直到所有的数据都被放入索引模型中.

对于构建好的学习索引 NUGC_LI, 从模型的根节点开始查询该键值, 迭代地使用模型来计算在指针数组中的位置, 然后根据指针节点得到下一层中的孩子节点, 循环往复直到选择到一个数据节点, 中间节点有极高的准确性且没有搜索代价. 使用叶子节点中的模型来预测查询键在键数组中的位置, 然后使用二分查找来寻找键值的实际位置. 如果键值被找到, 就根据键值找到查询内容并返回查询结果, 否则返回未查找到结果.

4.4 非均匀网格索引插入机制

移动对象数据随着物体位置的变换会不断更新, 产生大量动态数据, 因此为了满足移动对象数据处理的实时性, 学习索引应具备高效插入新键值且保持预测精度不受影响的能力, 因此对 NUGC_LI 索引设计了不同情况下的数据插入机制, 结合索引结构中的若干变量与数据结构开展不同情况下的数据插入工作.

因为学习索引的本质是根据输入查询键预测到数据存储的位置, 所以在进行数据插入时, 首先将待插入数据作为查询键输入 NUGC_LI 索引模型中, 通过模型预测得到该数据应插入的位置, 根据待插入位置节点的不同情况做出对应数据插入机制的选择. 较为理想的情况是, 插入数据时, 索引的叶子节点未满, 当前位置恰好为空元素, 则可以直接插入, 同步更新键值数组与 bitmap 数组; 若当前预测位置已有元素, 则需要对叶子节点上的数据进行移动. 如图 7 所示, 首先查找到距离当前预测位置最近的一个空元素位置 *EmptyPos*, 然后将该 *EmptyPos* 与预测位置相比较, 若 *EmptyPos* 在当前预测位置的后面, 则将预测位置到 *EmptyPos* 之间所有键值元素往后平移一位, 然后将新键值插入到预测位置; 反之若 *EmptyPos* 在当前预测位置的前面, 则将 *EmptyPos* 到预测位置前一位的所有键值元素往前平移一位, 然后将新键值插入到预测位置的前一位.

若待插入叶子节点的数据已满, 则需要对叶子节点进行扩展分裂操作, 值得注意的是因为需要为叶子节点中的数据预留数据移动的空间, 所以不能直至叶子节点中的数组全部被填满时才进行扩展操作, 设置了节点阈值 *NodeTHR*, 当该节点中的数据量达到阈值时即认为该节点已满, 需要进行扩展分裂操作. 扩展分裂操作分为两种类型, 一种是保持原有结构不变, 在叶子节点中构造一个容量更大的键值数组和数据指针数组, 将原有数组中的元素转存入这个更大的键值数组和数据指针数组中, 使得扩展后的数组密度值为预先设定的扩展密度值, 这就使得节

点的密度小于节点阈值, 符合新数据插入的基本条件. 但是, 因为数组容量的变更导致新数组无法与已训练的线性回归模型相匹配, 所以需要对当前的线性回归模型重新进行训练. 将新数组创建时间、数组元素移动时间、线性回归模型重训练时间和当前节点中键查询的平均时间之和作为此次节点扩展的时间代价. 另一种则是对节点的分裂, 将当前已经超出阈值的叶子节点分裂为两个相同大小的新叶子节点, 这种情况下需要判断, 当前叶子节点的父节点是否已经存在两个孩子节点, 若不存在, 则新增叶子节点直接作为兄弟节点与当前父节点相连, 更新与相邻叶子节点间的顺序连接关系; 若存在, 则当前叶子节点转化为一个内部节点下分支两个叶子节点. 分裂操作后的两个叶子节点平均分配原有键值和数据指针, 将创建节点时间、线性回归模型重训练时间和当前节点中键查询的平均时间之和作为此次节点分裂的时间代价. 在实际选择扩展分裂操作类型时, 优先使用时间代价较小的方式. 这样的插入机制使得插入键值并不总是存储在缓冲区或某一特定位置, 而是每次都插入在预测位置的附近, 不必因插入新数据而重新训练模型, 因为数据总在模型预测的误差范围之内, 有效保证了索引预测的准确性. 新数据始终插入模型预测位置附近, 而非全局重分布. 这一机制保证了插入操作的时间复杂度与节点内数据量线性相关而非全局重建, 单次插入的均摊代价较低, 适合高频增量场景. 通过 *NodeTHR* 阈值提前触发节点扩展或分裂而非等待节点完全填满, 预留插入空间以缓冲突发负载. 此外, 扩展与分裂策略优先选择时间代价较小的操作, 进一步减少高频插入的累积开销. 由于插入位置严格限制在模型预测误差范围内, 数据分布不会因高频更新剧烈偏移, 避免了频繁的模型重训练, 仅在节点扩展或分裂时触发局部模型更新, 确保了长期预测精度. 若插入频率极高且数据分布持续突变, 如节点频繁分裂, 可能导致分裂操作的时间代价累积. 节点扩展或分裂的代价包含创建节点时间、线性回归模型重训练时间和当前节点中键查询的平均时间, 此类操作的时间代价虽通过阈值控制得到减小, 但仍可能成为极端场景下的性能瓶颈.

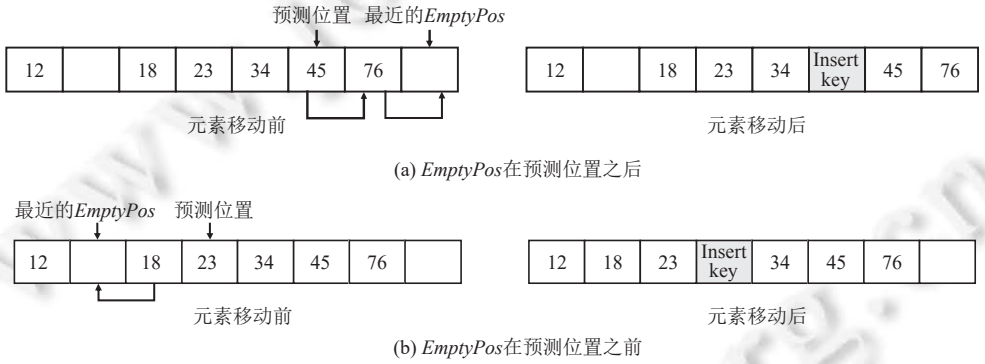


图 7 叶子节点未满时的索引数据插入机制示意图

4.5 非均匀网格索引删除及更新

因移动对象具有随时间快速变化位置的特点, 在实际的应用场景下, 会造成数据库中索引记录的频繁更新. 对于时间跨度较久、存储价值较低的数据记录和不再追踪的移动对象, 数据库均需删除其对应的索引数据. 在 NUGC_LI 学习索引中, 删除一个索引键的操作, 就是通过查询得到该键所在的位置, 然后将它和对应的数据记录的指针从索引数组和数据存储中同步删除. 这一删除操作应同时更新与之相关联的 bitmap, 以确保 bitmap 能够准确反映当前索引结构的状态. 如果该位置之前在 bitmap 中被标记为 1, 则在删除后需要将其更新为 0, 有助于后续插入操作准确判断该位置是否可用, 从而避免出现数据覆盖或混乱的问题. 删除操作仅是将数组中该位置的元素信息清空, 该位置的清空并不会导致学习模型中对于其他查询键位置预测准确性的影响, 因此索引不需要像插入操作时那样, 对于数组中其他已有的键值进行位置的移动, 操作十分简单快速. 但是, 若该键值删除后, 叶子节点中的键值数组和数据指针数组为空数组, 则需要进行节点删除工作, 即在父节点中删除指向该节点的指针, 并将该节点与相邻叶子节点间的双向指针断开, 将这两个相邻节点相互联结, 最后释放当前空的数据节点, 这一操作是为了降低索引的空间占用, 也能避免后续在范围查询等操作时因空叶子节点的存在造成冗余的遍历时间.

索引的更新操作是需要对某索引数据进行修改, 存在两种可能的情况, 一种是虽然该数据记录和索引需要重新赋值, 但是更新前后的变化较小, 并未影响到数组的有序排列, 在这种情况下仅需修改对应的数组元素, 不需要进行其他操作; 另一种是该数据的更新会打破当前数组的有序性, 这种情况就需要进行删除旧键和插入新键的组合操作, 首先通过查询操作找到需要更新的索引位置, 与其前后元素比较, 若更新值将破坏当前的有序性, 则将当前位置的索引值和对应的数据记录清空, 然后再次执行查询操作, 找到待更新值应插入的位置, 执行插入操作, 具体插入步骤可见第 4.4 节.

5 基于非均匀网格索引的查询算法

移动对象数据在支持位置服务的若干应用场景下, 主要的功能作用是为用户提供相关的查询服务, 因此基于学习索引 NUGC_LI 提出高效的查询处理技术也是重要研究内容. 本节介绍基于 NUGC_LI 的范围查询、最近邻查询和相似轨迹处理技术, 并通过实验验证了它们的有效性与实用性.

5.1 基于非均匀网格的范围查询

在现有面向移动对象的范围查询工作中, 查询区域通常被设置为矩形或圆形窗口, 通过经纬度信息查找通过该范围的所有移动对象, 并保证该区域中生成的移动对象的时间戳也在查找时间间隔内, 最终将查询结果返回输出. 例如, 通过将用户地理位置输入具有位置感知能力的应用程序中, 查询用户可以找到的附近相关的兴趣点, 如便利店等, 如算法 3 所述.

基于 NUGC_LI 的范围查询, 首先根据查询区域的范围边界值查找得到最小值所在的叶子节点; 然后依次扫描直到查询到范围的最大值. 在叶子节点的遍历过程中, 使用 bitmap 来跳过数组中的空元素, 如果查询范围较大涉及多个叶子节点, 则使用叶子节点间的指针直接完成跳转.

学习索引的引入使得范围查询在定位查询边界所在的叶子节点时, 有效减少了搜索时间, 叶子关键值范围 *LeafBound* 的设置也避免了对冗余节点的遍历, 弥补了 *NUGC* 的不足, 叶子节点间的指针连接使得范围查询在确定最小值所在节点之后能够直接跳至下一叶子节点, 顺序遍历有序数列, 无需再通过父节点进行跳转, 显著提升了查询效率. 在后续实验部分将与传统索引和经典学习索引进行对比, 凸显基于 NUGC_LI 的范围查询的效率优势.

算法 3. 基于 NUGC_LI 的空间范围查询 *RangeMO*.

输入: 查询域对角点 *ss*, 查询域对角点 *se*, 学习索引 NUGC_LI;
输出: 符合范围查询条件的移动对象集合 *Res*.

1. $Res \leftarrow \varnothing$
2. $ss' \leftarrow NUGC(ss)$
3. $se' \leftarrow NUGC(se)$
4. $prefix \leftarrow GetCommonPrefix(ss', se')$
5. $SearchRange \leftarrow get(prefix)$
6. $N \leftarrow GetRoot$(NUGC_LI)
7. **if** N 是空节点 **then**
8. **return** *Res*
9. **else**
10. $P \leftarrow$ NUGC_LI.*Pointer*(N)
11. 根据 *SearchRange* 查询叶子节点, 更新 P
12. $P \leftarrow$ NUGC_LI.*Pointer*(min(*SearchRange*))
13. **while** $P \neq$ null **do**
14. **if** *Intersect*(*P.LeafBound*, *SearchRange*) **then** /*通过叶子关键值范围判断是否需要遍历叶子节点*/

```
15.        for each iter ← 0 to L(keyarray) do
16.          if bitmap[iter] = 1 then /*通过 bitmap 跳过空元素*/
17.            if keyarray[iter] then 在查询范围内
18.              Res ← keyarray[iter] 指向的所有数据点
19.          else if keyarray[iter] > max(SearchRange) then
20.            return Res
21.      移动至下个叶子节点并更新 P
22. return Res
```

5.2 基于非均匀网格的最近邻查询

面向移动对象的最近邻查询则更为关注具有时间依赖性的最近邻查询研究, 如最快到达的出租车. 基于NUGC_LI 的最近邻查询算法是移动对象数据库中较为重要的查询操作之一, 需要查找与指定对象距离最近的移动对象.

最近邻查询在数据库的实际应用中还有一个重要变体, 即 k 近邻查询, 它的含义是连续查找 k 个距离指定对象最近的移动对象. 在经典的最近邻查询算法中, 研究者们提出了大量类似于剪枝界定深度优先的算法, 希望通过缩小搜索范围提高搜索效率. 但是, 所提出的最近邻查询算法与上述传统最近邻算法相比存在明显优势, 因为存储的一维值由基于位置关系的 *NUGC* 算法生成, 所以对查询点的搜索区域可以限定在指定网格中, 而学习索引 NUGC_LI 的引入使得对于网格的查询时间有效降低, 如算法 4 所示.

算法 4. 基于 NUGC_LI 的最近邻查询 *NNMO*.

输入: 查询点 Q, 学习索引 NUGC_LI;
输出: 符合范围查询条件的移动对象集合 *Res*.

```
1. Res ← ∅
2. SearchQ ← NUGC(Q) /*对查询点进行降维, 得到降维后一维网格编码*/
3. N ← GetRoot(NUGC_LI)
4. if N 是空节点 then
5.    return Res
6. else
7.    P ← NUGC_LI.Pointer(N)
8.    根据 SearchQ 查询到对应叶子节点更新 P
9.    P ← NUGC_LI.Pointer(SearchQ)
10.   遍历 SearchQ 网格中所有移动对象点
11.   Q_min ← MinDistance(Q,网格中任意点)
12.   根据指针 P 找到相邻的网格
13.   for each iter ← 1 to 相邻网格数量 do
14.      Distance(Q, 相邻网格中任意点)
15.      if Distance.value < MinDistance.value then
16.         更新 Q_min
17.   Res ← Q_min
18. return Res
```

算法 4 实现的最近邻算法同样为“粗过滤–细筛查”的二阶段查询过程, 首先在进行非均匀网格降维的基础上,

通过构建的学习索引 NUGC_LI 找到查询点 Q 对应单元格查询键所在的叶子节点, 并找到查询键位置, 其次通过数据指针数组找到该单元格中包含移动对象点数组, 依次遍历并计算单元格中所有移动对象与查询点 Q 的欧氏距离, 得到最近邻居, 若区域单元格数量为 1, 则寻找数据点的最近邻居, 否则在相邻单元格内的数据点继续搜索可能的最近邻居, 最后结束遍历, 返回最近邻居点. 该算法支持向 k 近邻的扩展. 在细筛查阶段, 可将单一最小值比较替换为可容纳 k 个元素的最大堆, 动态维护当前的前 k 个最近邻的候选集合. 当候选集合未满或相邻网格中存在距离更近的潜在点时, 算法可基于当前第 k 近邻的距离阈值, 自动扩展搜索范围至更远层级的相邻网格, 直至满足 k 个结果的覆盖性要求. 若某网格到查询点 Q 的最小距离大于距离阈值, 则可直接跳过该网格的遍历, 避免无效计算.

5.3 基于非均匀网格的相似轨迹查询

由于移动对象数据模型与数据采集现状, 传统的轨迹相似性计算与分析一般基于移动对象位置点距离的不同形式的组合, 割裂了移动对象位置点之间内在的连续性, 忽略了移动对象位置点之间的隐含的数据分布信息, 在比较实际轨迹时, 通过计算整条轨迹的相似性可以得到一个相似度数值. 然而, 真实移动对象的轨迹通常具有不同的长度, 时空起始点与终止点存在间隔, 采样频率与时长也各不相同, 整体相似性受轨迹跨越不同区域的影响较大.

经典轨迹相似性计算方法一般通过对时间翘曲或拉伸来实现采样点之间的匹配, 改变了移动对象数据的时空特性, 并不利于轨迹相似情况的对比分析, 而非均匀网格编码方法能够反映出轨迹在不同粒度下的相似特性. 从大量点距离的计算到基于连续有序段的字符串编码的匹配分析, 有效简化相似性度量步骤, 加快查询效率. 如算法 5 所示.

算法 5. 基于 NUGC_LI 的相似轨迹查询 *NL-TSR*.

输入: 查询轨迹 MP_Q, 学习索引 NUGC_LI;
输出: 符合相似轨迹查询条件的移动对象集合 *Res*.

```
1. Res ← ∅
2. max_sim ← 0
3. 将 MP_Q 中每个位置点转换为 NUGC 编码, 生成编码序列 Code_Q
4. N ← GetRoot(NUGC_LI)
5. if N 是空节点 then
6.     return Res
7. else
8.     P ← NUGC_LI.Pointer(N)
9.     while P ≠ null do
10.        for each 轨迹 Traj in P.LeafTrajs do /*遍历当前叶子节点存储的轨迹*/
11.            Code_i ← Traj 的 NUGC_LI 编码
12.            计算 Traj-StrSim(Code_Q, Code_i) /*基于定义 13 计算轨迹相似度*/
13.            if Traj-StrSim(Code_Q, Code_i) > max_sim then
14.                max_sim ← Traj-StrSim(Code_Q,Code_i)
15.                Res ← Traj /*重置为当前最大相似轨迹*/
16.            else if Traj-StrSim(Code_Q, Code_i) = max_sim then
17.                Res ← Res∪Traj /*保留相同最大值的轨迹*/
18.        移动至下一叶子节点并更新 P
19. return Res
```

第 2.2 节的定义 7 和定义 8 中已经对轨迹相似性度量及查询进行了描述, 但是基于非均匀网格降维后的编码,

轨迹的相似性度量与查询均得到改进. 移动对象位置点的编码可以反映出空间邻近关系, 将相似轨迹查询从大量点距离计算转化为简单的编码最长公共前缀匹配. 不同移动对象数据编码中的任意一对节点编码的相同前缀长度代表着对应移动对象数据的相似程度.

定义 13. 基于 NUGC_LI 的轨迹相似性度量.

移动对象数据在降至一维后, 定义轨迹相似度:

$$Traj\text{-}StrSim(MP_1, MP_2) = \frac{MP\text{-}StrSim(MP_1, MP_2)}{length} \tag{20}$$

$$MP\text{-}StrSim(MP_1, MP_2) = \frac{\sum_{Loc_i \in MP_1} StrSim(Loc_i, MP_2)}{2\times n} + \frac{\sum_{Loc_j \in MP_2} StrSim(Loc_j, MP_1)}{2\times n} \tag{21}$$

$$StrSim(Loc, MP) = \min_{P_i \in MP} Co\text{-}Prefix(Loc, P_i) \tag{22}$$

其中, $StrSim(Loc, MP)$ 为给定位置点 Loc 与移动对象 MP 的相似性度量, $Co\text{-}Prefix(Loc, P_i)$ 为给定位置点 Loc 与 MP 中第 i 个点编码的最长公共前缀长度. $MP\text{-}StrSim(MP_1, MP_2)$ 是轨迹中所有给定点距离另一轨迹的相似性度量, 即两条轨迹间的相似性度量, n 为每条轨迹中的代表点个数. $Traj\text{-}StrSim(MP_1, MP_2)$ 则是对轨迹间的相似性度量值进行归一化处理, $length$ 为初始编码长度. 从公式 (20) 易知 $Traj\text{-}StrSim(MP_1, MP_2) \in [0, 1]$, 且值越大, 轨迹相似性程度越高.

定义 14. 基于 NUGC_LI 的轨迹相似性查询 *NL-TSR*.

提出的轨迹相似性查询基于非均匀网格降维编码, 并在编码上构建学习索引 NUGC_LI, 基于 NUGC_LI 的轨迹相似性查询定义如下:

$$NL\text{-}TSQ(MP_Q) = \{MP_j \mid \forall MP_i \in MP,\ Traj\text{-}StrSim(MP_j, MP_Q) \geqslant Traj\text{-}StrSim(MP_i, MP_Q)\} \tag{23}$$

其中, MP_Q 是轨迹相似性查询的查询键, $Traj\text{-}StrSim(MP_j, MP_Q)$, $Traj\text{-}StrSim(MP_i, MP_Q)$ 是定义 14 中轨迹相似性的计算, 根据计算得到与查询键 MP_Q 轨迹相似性程度最高的 MP_j, MP_j 所组成的轨迹为相似轨迹查询结果.

6 实　验

为了评估所提出 NUGC_LI 索引的查询性能和动态更新效率, 使用 C++编译语言在 Ubuntu 20.04 环境下进行代码实现, 并将结果与 B+树和经典学习索引 RMI 进行对比, 实验硬件环境为: Intel(R) Core(TM) i7-5600U CPU @ 2.60 GHz, RAM 16.0 GB, Linux 5.15.0-76-generic.

6.1 实验设置

实验数据集包括真实采集的出租车移动轨迹、系统模拟仿真的火车移动轨迹与随机生成的移动轨迹这 3 类, 通过对不同类型数据集的测试保证了实验的完整性与结果的准确性. 轨迹数据集基本属性如表 3 所示, 其中真实采集的出租车移动轨迹为 2013 年 10 月 22 日 0–24 时期间 302 辆出租车的轨迹片段, 涉及移动对象点数量为 917 006 个; 系统模拟仿真的火车移动轨迹为 2003 年 11 月 20 日 6–9 时期间 562 辆火车的轨迹片段, 涉及移动对象点数量为 51 544 个; 随机生成的是以北京市某地坐标 (116.231 7, 39.542 7) 为期望数组, 维数为 2, 协方差矩阵为 [1, 1.5], [1.5, 4] 的随机函数生成的数据点, 涉及数据点数量为 1 000 000 个. 深圳市数据集密度分布呈现显著的空间聚集性, 网格最大密度值 47, 对应城市热点区域, 网格密度标准差为 2.4, 占总网格数 22% 的高密度网格 ($Dens \geqslant 1$) 覆盖了 83% 的移动对象点; 柏林市数据集密度沿预设铁路线呈带状分布, 网格最大密度值 8.8, 网格密度标准差为 0.5, 分布相对均匀; 随机数据集服从二维高斯分布, 密度峰值位于期望坐标 (116.231 7, 39.542 7), 密度值随距离衰减, 网格最大密度值为 3, 网格密度标准差为 1.2, 且约 20% 的网格覆盖了 65% 的数据点. 在本节中所使用的数据均建立在第 3 节非均匀网格降维的基础上, 分别将上述 3 个数据集进行非均匀网格降维, 然后分别在这 3 个降维后的一维非均匀网格编码数据集上建立 B+树、RMI、ALEX 和 NUGC_LI 索引结构, 同时在未进行编码的 3 个原始数据集上建立 3DR 树与 TB 树.

表 3 轨迹数据集属性表

数据集	时间范围	轨迹条数	轨迹点数	采样频率 (s)	网格密度标准差
深圳市	2013年10月22日0–24时	562	917006	30	2.4
柏林市	2003年11月20 日6–9时	302	51544	120	0.5
随机	2008年2月2日15时–8日15时	3203	5222752	300	1.2

经典 RMI 结构采用两层模型, 第 1 层为 1 个前馈神经网络, 第 2 层为依据数据集划分数据生成的若干个前馈神经网络, 神经网络中的隐藏层均为 2 层, 激活函数为 ReLU, 神经元数量固定为 100. RMI 在查询过程中体现出学习索引的典型特点, 即索引输入为 1 个查询键, 经第 1 层神经网络预测后, 得到第 2 层模型的编号, 进入预测指定的第 2 层神经网络, 随后再次经过神经网络预测, 得到预测的查询值位置, 在数据存储列表中的误差范围内开展遍历查询. 但是, RMI 结构并未设置支持动态更新的模型扩展及分裂机制, 仅存在模型中固定大小的缓冲空间, 插入此结构的数据和键值不能自由选择位置只能插入指定的缓冲空间, 会影响后续对该查询键的位置预测准确性, 同时, 因为缺乏扩展分裂机制, 在缓冲空间被填满后, 该模型需要重新进行训练, 对新的数据集排序划分子集后, 再次循环若干子模型, 构造出新的 RMI 结构.

ALEX 作为可更新自适应学习索引, 通过融合学习索引的分布感知能力与传统 B+树的结构优势, 在动态工作负载场景中实现了性能突破. 首先, 采用模型导向的插入策略, 基于线性回归模型预测数据分布, 通过贪心插入算法优化键值存储位置; 其次, 设计自适应节点扩展机制, 根据数据增长模式动态选择节点分裂或就地扩展策略, 结合间隙预留技术平衡空间利用率与更新效率; 最后, 构建混合存储布局, 在内部节点集成数组存储与间隙控制. 但索引性能对数据分布平稳性具有较强依赖性, 在面对非平稳数据流时需引入在线模型重训练机制, 带来额外计算开销, 在强实时更新系统中仍需权衡性能与维护成本.

6.2 非均匀网格索引实验

NUGC_LI 索引构建结构及所用函数在上述部分已有说明, 误差值 *err* 为 10, 误差区间 *ErrINR* 为 20, 根节点中线性模型的斜率为 $1.0/(key_{\max}-key_{\min})$, 截距为 $-key_{\min}/(key_{\max}-key_{\min})$, 根节点与中间节点的线性模型输出均为 *CDF* 函数, 输出值为 [0, 1] 之间的概率值, 由概率值对应归一化后的模型编号或数据位置, 就能确定下一阶段应选节点, 各节点最大容纳空间为 16 MB, 根据键值大小 (64 KB), 理论最大扇出数为 256.

6.2.1 学习索引节点阈值对比

NUGC_LI 索引节点中设置有节点阈值 *NodeTHR* 为后续判断节点是否需要扩展或分裂提供参考, 对 *NodeTHR* 参数的不同选择对索引性能的影响设置了对比实验, *NodeTHR* 的计算公式在定义 11 中已作详细说明, 易知公式 (16) 中 *X* 的设置直接决定了 *NodeTHR*, 因此下列对 *X* 的不同取值下索引的性能进行了实验分析, 对 *X* 进行 50%–90% 区间内的不同取值, 实验结果如图 8 所示.

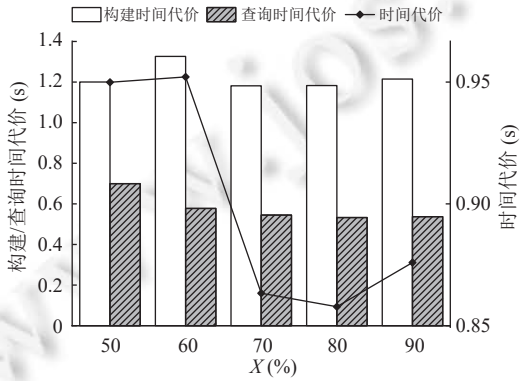


图 8 不同节点阈值性能对比图

对不同节点阈值下索引的构建时间和查询时间进行统计分析, 并对这两者取相同权重进行加权计算得到综合时间代价, 即图 8 折线数据点所示, 通过折线图趋势, 发现当 X 取 80% 时, 索引节点的综合时间代价是 0.85 s 为最优值, 故取该值为节点阈值的实验参数.

6.2.2 索引构建时间对比

对深圳市、柏林市和随机数据集分别构建了 B+树、RMI、ALEX、NUGC_LI、3DR 树和 TB 树索引, 构建时间结果如图 9 所示. 因为在深圳市数据集和随机数据集中, 基于 3DR 树和 TB 树索引构建时间比其他索引大两个数量级, 为便于展示, B+树、RMI、ALEX 与 NUGC_LI 索引的构建时间刻度为左纵轴, 3DR 树和 TB 树索引构建时间刻度为右纵轴. 在真实轨迹、仿真轨迹和随机数据集上, 6 种索引的构建时间呈现出同一趋势, TB 树索引耗费时间最多, 其次分别是 3DR 树、B+树、RMI 和 ALEX. NUGC_LI 所用时间最少, 比 TB 树降低至少 91.45%, 比 3DR 树降低至少 89.63%, 比 B+树降低至少 90.38%, 比 RMI 降低至少 87.46%, 比 ALEX 降低至少 13.71%. 此外, 以数据量最大的随机数据集为例, 采用逐步增加数据规模的方法, 分别以 25%、50%、75% 和 100% 的比例进行伸缩性测试. 如后文图 10 所示, 在不同数据比例下, NUGC_LI 索引的性能始终优于 B+树、RMI、ALEX、3DR 树以及 TB 树索引, 且随着数据量的逐渐增大, 构建时间不会产生较大的抖动, 所以 NUGC_LI 可以很好地扩展到更大的数据集.

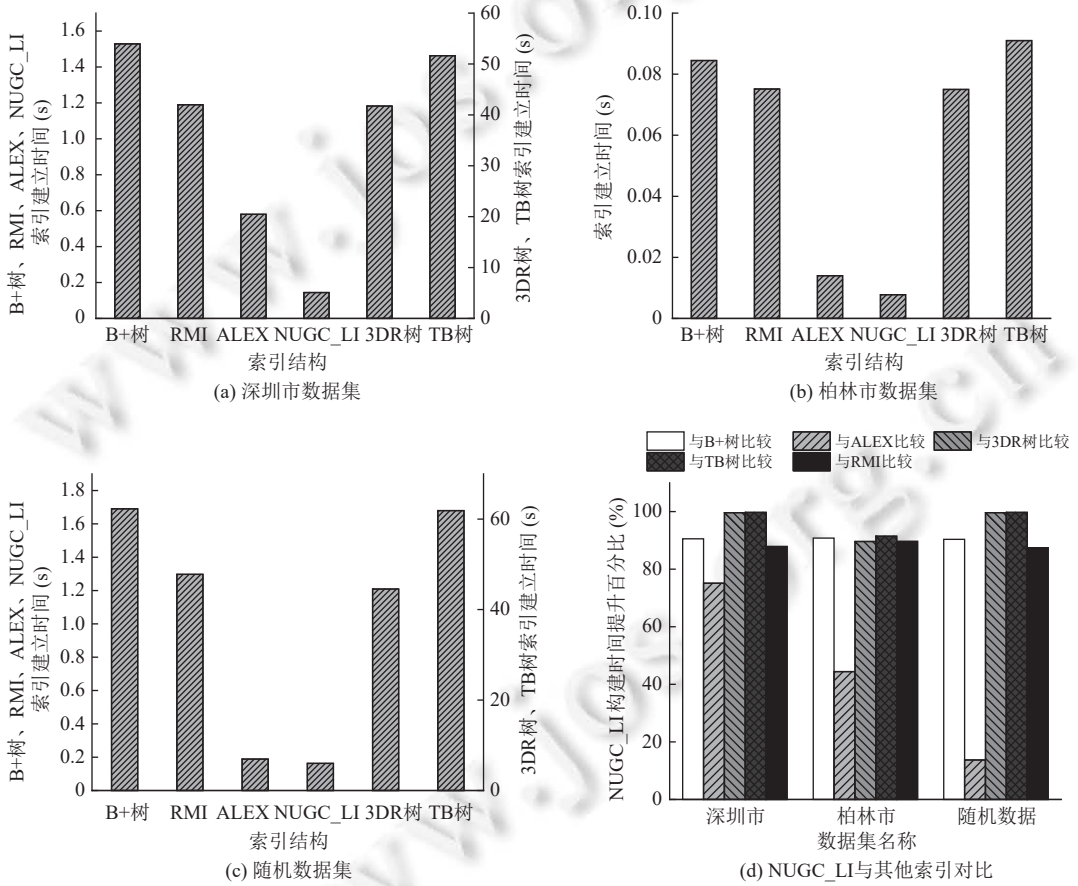


图 9 B+树、RMI、ALEX、NUGC_LI 不同索引建立时间对比图

6.2.3 索引更新时间对比

此外, 对不同数据集在 6 种索引上的更新操作进行了对比实验, 索引更新时间的实验结果如图 11 所示. 在 B+树、RMI、ALEX、NUGC_LI、3DR 树和 TB 树 6 种不同的索引结构上, 这 3 个数据集的更新时间有所不同,

尤其是 NUGC_LI 索引的更新时间较其他索引减少了 3 个数量级, 优势明显. 综合比较下, B+树索引在更新时所耗费的时间最多, 其次分别是 RMI、ALEX、3DR 树和 TB 树. RMI 需要重建, ALEX 在非均匀轨迹数据上性能下降, 反而传统移动对象在部分非均匀的数据上更新效果更好. NUGC_LI 所花费的更新时间比其他索引减少了至少 93.76%. 此外, 以数据量最大的随机数据集为例, 采用逐步增加数据规模的方法, 分别以 25%、50%、75% 和 100% 的比例进行伸缩性测试. 如后文图 12 所示, 在不同数据比例下, NUGC_LI 索引的性能始终优于 B+树、RMI、ALEX、3DR 树以及 TB 树索引, 且随着数据量的逐渐增大, 更新时间不会产生较大的抖动, 所以 NUGC_LI 可以很好地扩展到更大的数据集.

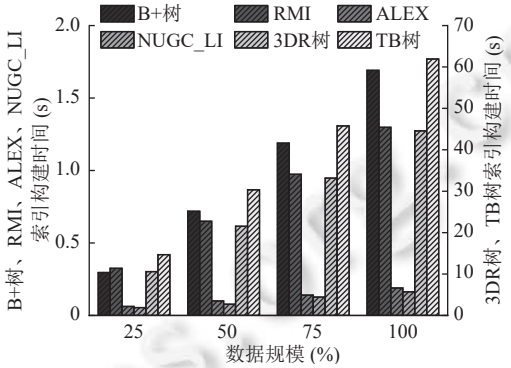


图 10 可变数据量比例下不同索引构建时间伸缩性对比图

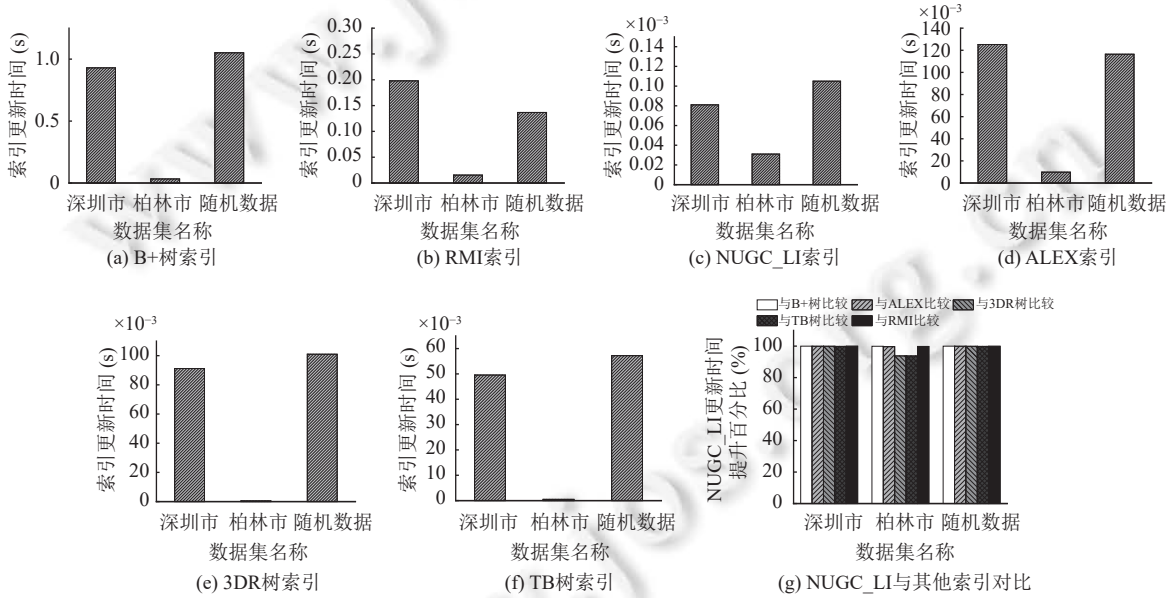


图 11 不同索引更新时间对比图

6.3 查询实验

6.3.1 范围查询

对不同索引结构的数据集进行范围查询, 查询时间结果如图 13 所示. 在进行范围查询时, 通过随机函数生成随机查询点, 然后以该点为区域最小点 $(X_{\min}, Y_{\min})$, 设置固定范围长度 10, 故查询区域为 $(X_{\min}, X_{\min}+10, Y_{\min}, Y_{\min}+10)$, 查询轮数为 5 轮, 在每一轮中随机生成查询点, 取 5 轮查询时间的平均值作为实验结果.

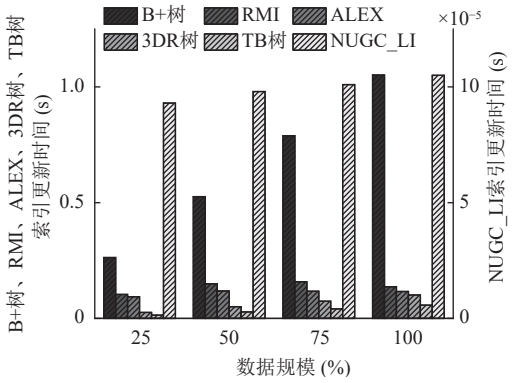


图 12 可变数据量比例下不同索引更新时间伸缩性对比图

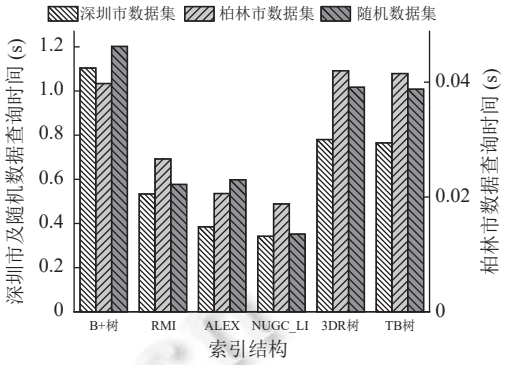


图 13 不同索引范围查询时间对比图

因为柏林市数据集规模较小, 查询时间较另外两个数据集相差两个数量级, 因此在图 13 中, 深圳市和随机数据集的查询时间对应左纵轴刻度, 柏林市数据集的查询时间对应右纵轴刻度. 通过查询时间对比, 易知基于 NUGC_LI 索引的范围查询具有明显优势, 除 3DR 树和 TB 树外, 在柏林市数据集上性能提升最少, 仅比 ALEX 提升 8.74%, 比 RMI 提升 29.38%, 比 B+树提升 52.72%. 除 3DR 树和 TB 树外, 在随机数据集上性能提升最高, 比 ALEX 提升 20.21%, 比 RMI 提升 39%, 比 B+树提升 70.73%. NUGC_LI 索引相比于 3DR 树和 TB 树索引在 3 个数据集上的提升相对稳定, 提升均在 50% 左右.

为了进一步研究基于 NUGC_LI 的范围查询在不同区域窗口下的性能区别, 选择了深圳市真实出租车数据集和随机生成数据集, 对这两个数据集分别进行划分, 各生成 4 种不同大小的区域窗口进行查询. 查询时间结果如图 14 所示.

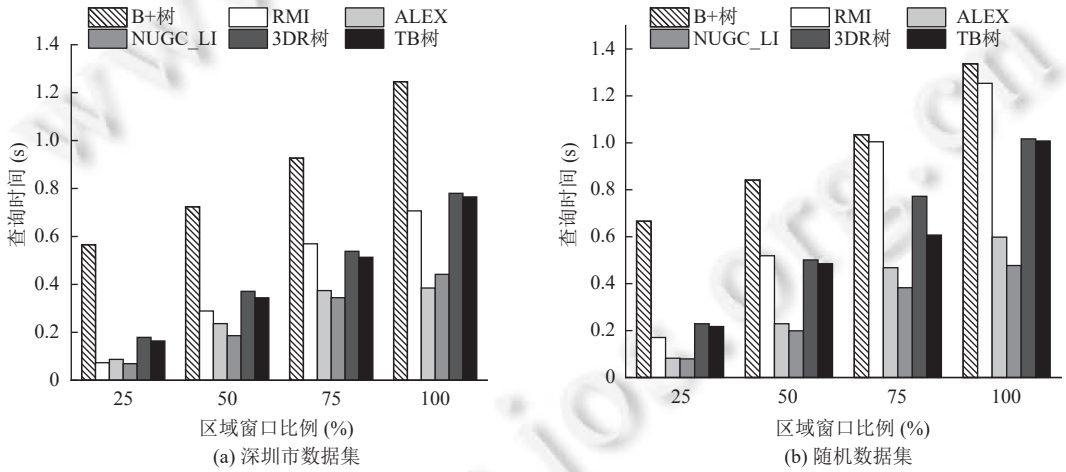


图 14 不同区域窗口查询时间对比图

通过查询时间比较分析可知, 在深圳市数据集中, 当区域窗口为 25% 时, NUGC_LI 索引查询时间较 B+树降低 87.75%, 较 RMI 降低 5.55%, 较 ALEX 降低 20.37%, 较 3DR 树降低 61.3%, 较 TB 树降低 57.76%; 当区域窗口为 50% 时, 较 B+树降低 74.27%, 较 RMI 降低 35.63%, 较 ALEX 降低 21.3%, 较 3DR 树降低 49.85%, 较 TB 树降低 45.92%; 当区域窗口为 75% 时, 较 B+树降低 62.82%, 较 RMI 降低 39.5%, 较 ALEX 降低 7.87%, 较 3DR 树降低 35.96%, 较 TB 树降低 32.83%; 当区域窗口为 100% 时, 较 B+树降低 64.53%, 较 RMI 降低 37.47%, 较 ALEX 降低 11.15%, 较 3DR 树降低 56.14%, 较 TB 树降低 55.28%. 在随机生成数据集中, 当区域窗口为 25% 时, NUGC_LI

索引查询时间较 B+树降低 88.1%, 较 RMI 降低 22.64%, 较 ALEX 降低 3.75%, 较 3DR 树降低 65.4%, 较 TB 树降低 63.48%; 当区域窗口为 50% 时, 较 B+树降低 76.36%, 较 RMI 降低 36.14%, 较 ALEX 降低 13.2%, 较 3DR 树降低 60.32%, 较 TB 树降低 36.96%; 当区域窗口为 75% 时, 较 B+树降低 63.02%, 较 RMI 降低 36.5%, 较 ALEX 降低 18.22%, 较 3DR 树降低 50.44%, 较 TB 树降低 26.96%; 当区域窗口为 100% 时, 较 B+树降低 64.31%, 较 RMI 降低 36.56%, 较 ALEX 降低 20.21%, 较 3DR 树降低 53.08%, 较 TB 树降低 52.67%. 可见在不同查询区域窗口下, NUGC_LI 索引的性能始终优于 B+树、RMI、ALEX、3DR 树和 TB 树索引, 且在随机生成数据集中更为稳定.

6.3.2 最近邻查询

在不同索引结构的 3 种不同数据集上进行最近邻查询实验, 查询时间的实验结果对比如图 15 所示.

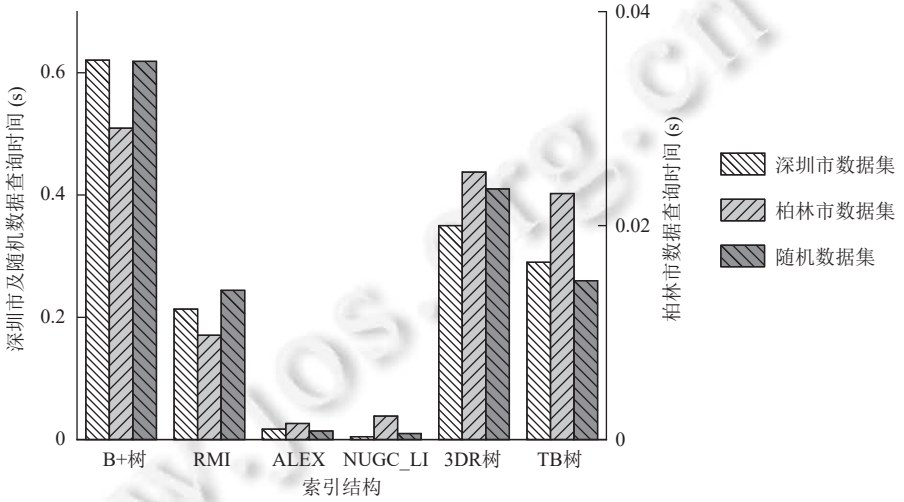


图 15 不同索引最近邻查询时间对比图

在进行最近邻查询时, 会在每一轮由随机函数生成一个随机查询点, 在不同索引结构中查找距离该查询点最近的邻居点, 共设置 10 轮, 每一轮生成一个随机查询点, 将 10 次最近邻查询的平均时间作为实验结果.

与范围查询一样, 因为柏林市数据集规模较小, 查询时间较另外两个数据集相差两个数量级, 因此图 15 中, 深圳市和随机数据集的查询时间对应左纵轴刻度, 柏林市数据集的查询时间对应右纵轴刻度. 可以看出, 基于 NUGC_LI 索引的最近邻查询与其他索引结构下的查询相比, 查询时间明显降低. 除 ALEX 索引外, 对于不同数据集的查询性能提升效果则各有不同, 在柏林市数据集上性能提升最少, 但也比 RMI 提升了 98.81%. 在其他数据集上, 查询性能提升均达 90% 以上. 相比于 ALEX 索引, 由于柏林市数据集较小, 同时轨迹分布相对均匀, 所以 NUGC_LI 和 ALEX 索引查询性能相当, 随着数据集增大和轨迹分布更加复杂, NUGC_LI 相比于 ALEX 索引在深圳市数据集和随机数据集上分别提升 71.76% 和 30%.

为了使得基于 NUGC_LI 的最近邻查询更符合实际应用场景的需求, 在最近邻查询的基础上, 拓展查询方式为 k 近邻查询, 即在不同索引结构中查找距离查询点最近的 k 个邻居点, 通过新建一个可容纳 k 个元素的最大堆实现. 选择了深圳市真实出租车数据集和随机生成数据集, 取 k 的值分别为 1、5、10 和 15, 并开展实验. 实验的查询时间结果如图 16 所示.

因为基于 ALEX 和 NUGC_LI 索引的 k 近邻查询时间比其他两种索引小两个数量级, 为便于展示, B+树、RMI、3DR 树与 TB 树索引的查询时间刻度为左纵轴, ALEX 和 NUGC_LI 索引查询时间刻度为右纵轴. 通过查询时间比较分析可知, 在深圳市数据集中, 随着 k 值的增长, NUGC_LI 索引的查询时间较 B+树索引降低至少 99.87%, 较 RMI 索引降低至少 97.56%, 较 ALEX 索引降低至少 38.13%, 较 3DR 树索引降低至少 99.09%, 较 TB 树索引降低至少 96.17%, 但降低幅度均呈现逐渐下降趋势. 在随机生成数据集中, 随着 k 值的增长, NUGC_LI 索引的查询时间较 B+树索引降低至少 99.84%, 较 RMI 索引降低至少 98.16%, 较 ALEX 索引降低至少 4.96%, 较

3DR 树索引降低至少 96.23%, 较 TB 树索引降低至少 94.68%. 可见在进行 k 近邻查询时, 无论 k 取何值, 除 ALEX 索引外, NUGC_LI 索引都具有明显优势, 且在不同数据集中查询性能都十分稳定. 相比于 ALEX 索引, NUGC_LI 在 k 为 1 时优势明显, 在深圳市数据集和随机数据集上分别降低 71.76% 和 30%, 然而随着 k 值增加, 优势逐渐下降, 但最终仍优于 ALEX 索引.

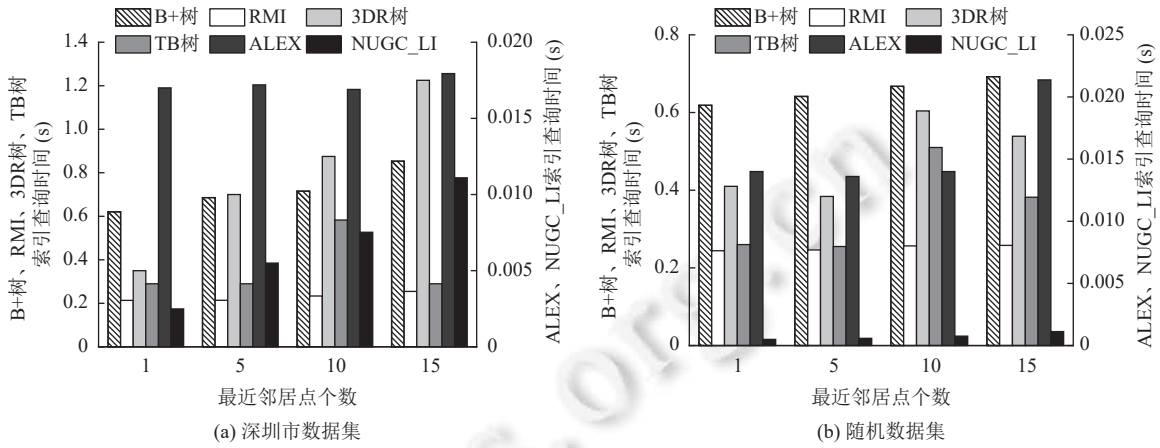


图 16 不同 k 值的最近邻查询时间对比图

此外, 考虑到移动对象数据还具有分布不均匀的特点, 还随机抽取了深圳市数据集中, 不同密度网格中的点作为查询点进行最近邻查询, 研究网格密度对于最近邻查询性能的影响. 图 17 是对网格密度处于 $0<Dens_{Q1}<0.5$、$0.5<Dens_{Q2}<1$、$1<Dens_{Q3}<5$、$Dens_{Q4}>5$ 这 4 个区间的查询点进行实验后得到的结果.

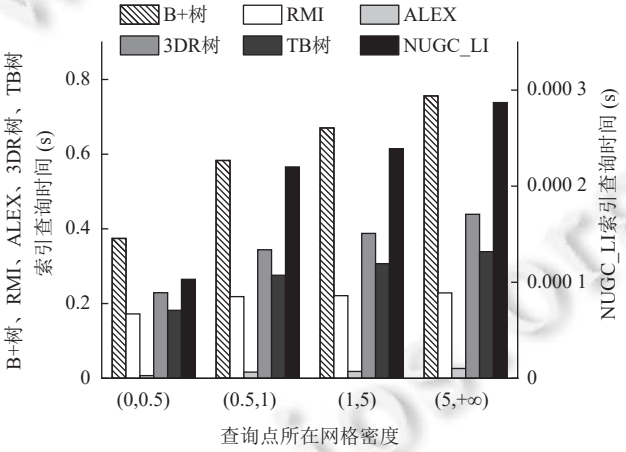


图 17 不同网格密度的最近邻查询时间对比图

同样因为基于 NUGC_LI 索引的最近邻查询时间比其他两种索引小 3 个数量级, 为便于展示, B+树、RMI、ALEX、3DR 树和 TB 树索引的查询时间刻度为左纵轴, NUGC_LI 索引查询时间刻度为右纵轴. 通过查询时间比较分析可知, 在不同网格密度的最近邻查询中, NUGC_LI 索引的性能始终优于其他索引, 至少可比 B+树提升 99.96%, 比 RMI 提升 99.07%, 比 ALEX 提升 98.55%, 比 3DR 树提升 99.93%, 比 TB 树提升 99.92%, 但随着网格密度的升高, NUGC_LI 的查询优势逐渐减小.

6.3.3 相似轨迹查询

每条移动对象轨迹都包含特定时间段内的所有行程, 为了取不同轨迹在同一时间段内的代表点, 从而以代表

点的相似程度来衡量轨迹的相似程度, 采用等间隔取样的方式, 分别对数据集以 1、5、10、15、30 min 这 5 种时间区间进行轨迹划分. 最终如图 18 所示, 通过综合性能对比, 选定 15 min 为子轨迹划分时间阈值. 在不同索引结构的 3 种不同数据集上进行相似轨迹查询实验, 查询时间的实验结果对比如图 19 所示.

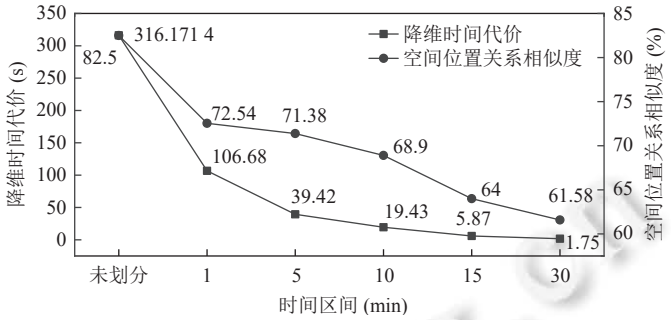


图 18 不同时间区间对性能的影响

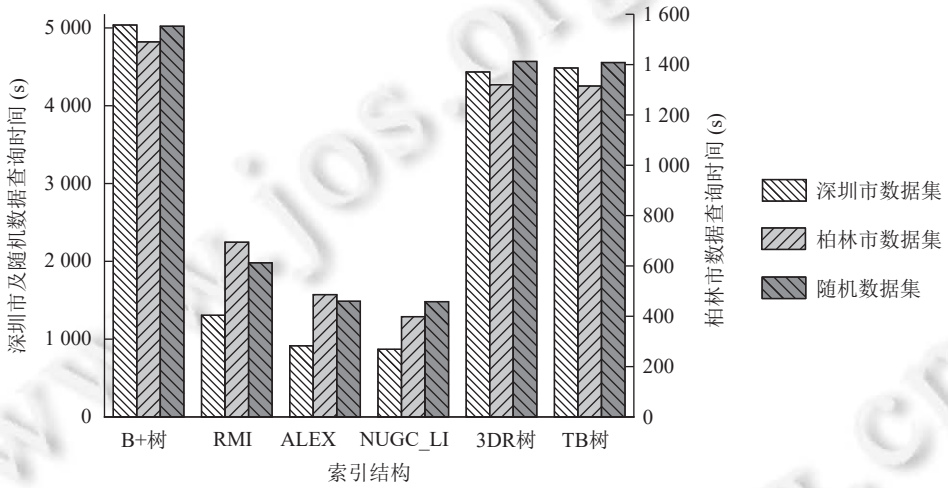


图 19 不同索引相似轨迹查询时间对比图

与上述查询实验一样, 因为柏林市数据集规模较小, 查询时间较另外两个数据集相差较大, 因此图 18 中, 深圳市和随机数据集的查询时间对应左纵轴刻度, 柏林市数据集的查询时间对应右纵轴刻度. 通过查询时间对比, 可知基于 NUGC_LI 索引的相似轨迹查询与其他索引结构下的查询相比均具有优势, 在随机数据集上性能提升最少, 仅比 ALEX 提升 3.77%, 比 RMI 提升了 25.24%, 比 TB 树提升 67.47%, 比 3DR 树提升 67.58%, 比 B+树提升 70.5%, 在深圳市和柏林市数据集上, 均比 ALEX 提升 10% 以上, 比 RMI 提升 30% 以上, 比 3DR 树和 TB 树提升 69% 以上, 比 B+树提升 73% 以上.

此外, 将基于 NUGC_LI 的相似轨迹查询拓展为 k 条相似轨迹查询, 即在不同索引结构中查找与查询轨迹最相似的 k 条轨迹, 通过新建一个可容纳 k 个元素的最大堆实现. 选择了深圳市真实出租车数据集和随机生成数据集, 取 k 的值分别为 1、3、5 和 7, 并开展实验. 实验的查询时间结果如图 20 所示.

因为基于 RMI 与 NUGC_LI 索引的相似轨迹查询时间比 B+树、3DR 树和 TB 树索引降低较为明显, 为便于展示, B+树、3DR 树和 TB 树索引的查询时间刻度为左纵轴, RMI、ALEX 与 NUGC_LI 索引查询时间刻度为右纵轴. 通过查询时间比较分析可知, 在深圳市数据集中, 随着 k 值的增长, NUGC_LI 索引的查询时间较 B+树索引降低至少 82.85%, 较 RMI 索引降低至少 33.42%, 较 3DR 树索引降低至少 75.41%, 较 TB 树索引降低至少 75.65%, 且降低幅度较为稳定, 随 k 值影响变化不大. 在随机生成数据集中, 随着 k 值的增长, NUGC_LI 索引的查询时间

较 B+树索引降低至少 71.52%, 较 RMI 索引降低至少 20.19%, 较 3DR 树索引降低至少 64.39%, 较 TB 树索引降低至少 63.92%. 可见除 ALEX 索引外, 无论 k 取何值, NUGC_LI 索引都具有明显优势, 且在不同数据集中查询性能都十分稳定. 相比于 ALEX 索引, NUGC_LI 索引在 k 为 1 时, 在深圳市数据集和随机数据集上表现相当, 但随着 k 值增大, NUGC_LI 索引优势逐渐增大, 查询时间分别最高降低 21.99% 和 16.06%.

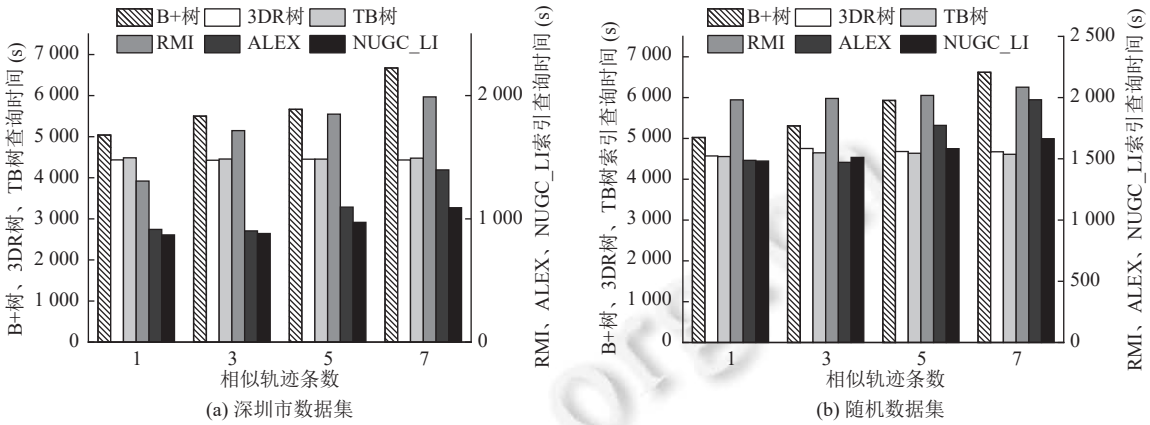


图 20 不同 k 值的相似轨迹查询时间对比图

7 总结与展望

以学习索引研究为出发点, 提出了基于空间位置关系的降维算法 *NUGC*、学习索引 NUGC_LI 以及 3 种应用广泛的查询技术, 通过构建高效索引提升移动对象数据的查询效率, 支持移动对象数据的频繁更新, 以满足现实场景下对于移动对象数据的处理任务需求, 提供更为快速、高效的查询服务. 但是对于移动对象数据库领域而言, 仍然存在许多难题和挑战需要攻克, 学习索引在移动对象数据领域的应用还处于初步阶段, 并未在现实场景下被广泛部署应用.

References

[1] Han SY, He Q, Yu ZQ, Tong XR, Zheng BL. Double layer index for continuous *k*-nearest neighbor queries on moving objects. Ruan Jian Xue Bao/Journal of Software, 2023, 34(6): 2789–2803 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6492.htm [doi: 10.13328/j.cnki.jos.006492]

[2] Antol M, Ol'ha J, Slanináková T, Dohnal V. Learned metric index—Proposition of learned indexing for unstructured data. Information Systems, 2021, 100: 101774. [doi: 10.1016/j.is.2021.101774]

[3] Marcus R, Zhang E, Kraska T. CDFShop: Exploring and optimizing learned index structures. In: Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. Portland: ACM, 2020. 2789–2792. [doi: 10.1145/3318464.3384706]

[4] Cai P, Zhang SM, Liu PR, Sun LM, Li CP, Chen H. An overview of learned index technologies for intelligent database. Chinese Journal of Computers, 2023, 46(1): 51–69 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2023.00051]

[5] Chai MK, Fan J, Du XY. Learnable database systems: Challenges and opportunities. Ruan Jian Xue Bao/Journal of Software, 2020, 31(3): 806–830 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5908.htm [doi: 10.13328/j.cnki.jos.005908]

[6] Li GL, Zhou XH, Sun J, Yu X, Yuan HT, Liu JB, Han Y. A survey of machine learning based database techniques. Chinese Journal of Computers, 2020, 43(11): 2019–2049 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2020.02019]

[7] Chao C, Pu FF, Xu JQ, Gao YJ. Efficient dimensionality reduction and query algorithm of trajectory data based on spatial position relation. Journal of Computer Research and Development, 2024, 61(7): 1771–1790 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.202330609]

[8] Tong YX, She JY, Ding BL, Wang LB, Chen L. Online mobile micro-task allocation in spatial crowdsourcing. In: Proc. of the 32nd IEEE Int'l Conf. on Data Engineering. Helsinki: IEEE, 2016. 49–60. [doi: 10.1109/ICDE.2016.7498228]

[9] Tong YX, Pan XC, Zeng YX, Shi YX, Xue CB, Zhou ZM, Zhang XF, Chen L, Xu Y, Xu K, Lv WF. Hu-Fu: Efficient and secure spatial

queries over data federation. Proc. of the VLDB Endowment, 2022, 15(6): 1159–1172. [doi: 10.14778/3514061.3514064]

[10] Bentley JL. Multidimensional binary search trees used for associative searching. Communications of the ACM, 1975, 18(9): 509–517. [doi: 10.1145/361002.361007]

[11] Ram P, Sinha K. Revisiting KD-tree for nearest neighbor search. In: Proc. of the 25th ACM SIGKDD Int'l Conf. on Knowledge Discovery & Data Mining. Anchorage: ACM, 2019. 1378–1388. [doi: 10.1145/3292500.3330875]

[12] Nurgaliev I, Muzammal M, Qu Q. Enabling blockchain for efficient spatio-temporal query processing. In: Proc. of the 19th Int'l Conf. on Web Information Systems Engineering. Dubai: Springer, 2018. 36–51. [doi: 10.1007/978-3-030-02922-7_3]

[13] Li GH, Zhao P, Yuan L, Gao S. Efficient implementation of a multi-dimensional index structure over flash memory storage systems. The Journal of Supercomputing, 2013, 64(3): 1055–1074. [doi: 10.1007/s11227-011-0679-0]

[14] Liu BZ, Chen L, Zhu XQ, Zhang Y, Zhang CQ, Qiu WD. Protecting location privacy in spatial crowdsourcing using encrypted data. Advances in Database Technology-EDBT, 2017, 3: 478–481. [doi: 10.5441/002/EDBT.2017.49]

[15] Guttman A. R-trees: A dynamic index structure for spatial searching. In: Proc. of the 1984 ACM SIGMOD Int'l Conf. on Management of Data. Boston: ACM, 1984. 47–57. [doi: 10.1145/602259.602266]

[16] Qi JZ, Tao YF, Chang YC, Zhang R. Packing R-trees with space-filling curves: Theoretical optimality, empirical efficiency, and bulk-loading parallelizability. ACM Trans. on Database Systems, 2020, 45(3): 14. [doi: 10.1145/3397506]

[17] Theodoridis Y, Vazirgiannis M, Sellis T. Spatio-temporal indexing for large multimedia applications. In: Proc. of the 3rd IEEE Int'l Conf. on Multimedia Computing and Systems. Hiroshima: IEEE, 1996. 441–448. [doi: 10.1109/MMCS.1996.535011]

[18] Pfoser D, Jensen CS, Theodoridis Y. Novel approaches in query processing for moving object trajectories. In: Proc. of the 26th Int'l Conf. on Very Large Data Bases. San Francisco: Morgan Kaufmann Publishers Inc., 2000. 395–406.

[19] Šaltinis S, Jensen CS, Leutenegger ST, López MA. Indexing the positions of continuously moving objects. In: Proc. of the 2000 ACM SIGMOD Int'l Conf. on Management of Data. Dallas: ACM, 2000. 331–342. [doi: 10.1145/342009.335427]

[20] Tao YF, Papadias D, Sun JM. The TPR*-tree: An optimized spatio-temporal access method for predictive queries. In: Proc. of the 29th Annual Int'l Conf. on Very Large Data Bases. Berlin: Morgan Kaufmann Publishers Inc., 2003. 790–801. [doi: 10.1016/B978-012722442-8/50075-6]

[21] Chakka VP, Everspaugh AC, Patel JM. Indexing large trajectory data sets with SETI. In: Proc. of the 2003 CIDR Conf. California: cidrdb.org, 2003.

[22] Beckmann N, Seeger B. A revised R*-tree in comparison with related index structures. In: Proc. of the 2009 ACM SIGMOD Int'l Conf. on Management of Data. Providence: ACM, 2009. 799–812. [doi: 10.1145/1559845.1559929]

[23] Alamri S, Taniar D, Nguyen K, Alamri A. C-tree: Efficient cell-based indexing of indoor mobile objects. Journal of Ambient Intelligence and Humanized Computing, 2020, 11(7): 2841–2857. [doi: 10.1007/S12652-019-01397-W]

[24] Hastings EJ, Mesit J, Guha RK. Optimization of large-scale, real-time simulations by spatial hashing. In: Proc. of the 2005 Summer Computer Simulation Conf. Berlin: Springer, 2005. 37(4): 9–17.

[25] Ding KM, Zhu CQ, Lu FQ. An adaptive grid partition based perceptual hash algorithm for remote sensing image authentication. Geomatics and Information Science of Wuhan University, 2015, 40(6): 716–720, 743 (in Chinese with English abstract). [doi: 10.13203/j.whugis20130567]

[26] Pâris JF, Schwarz T. Merkle hash grids instead of Merkle trees. In: Proc. of the 28th Int'l Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems. Nice: IEEE, 2020. 1–8. [doi: 10.1109/MASCOTS50786.2020.9285942]

[27] Kriegel HP, Pötke M, Seidl T. Interval sequences: An object-relational approach to manage spatial data. In: Proc. of the 7th Int'l Symp. on Spatial and Temporal Databases. Redondo Beach: Springer, 2001. 481–501. [doi: 10.1007/3-540-47724-1_25]

[28] Zhang R, Qi JZ, Stradling M, Huang J. Towards a painless index for spatial objects. ACM Trans. on Database Systems, 2014, 39(3): 19. [doi: 10.1145/2629333]

[29] Kumar S, Madria S, Linderman M. M-Grid: A distributed framework for multidimensional indexing and querying of location based data. Distributed and Parallel Databases, 2017, 35(1): 55–81. [doi: 10.1007/s10619-017-7194-0]

[30] Kraska T, Beutel A, Chi EH, Dean J, Polyzotis N. The case for learned index structures. In: Proc. of the 2018 Int'l Conf. on Management of Data. Houston: ACM, 2018. 489–504. [doi: 10.1145/3183713.3196909]

[31] Wu YJ, Yu J, Tian YY, Sidle R, Barber R. Designing succinct secondary indexing mechanism by exploiting column correlations. In: Proc. of the 2019 Int'l Conf. on Management of Data. Amsterdam: ACM, 2019. 1223–1240. [doi: 10.1145/3299869.3319861]

[32] Kipf A, Marcus R, van Renen A, Stoian M, Kemper A, Kraska T, Neumann T. RadixSpline: A single-pass learned index. In: Proc. of the 3rd Int'l Workshop on Exploiting Artificial Intelligence Techniques for Data Management. Portland: ACM, 2020. 5. [doi: 10.1145/3401071.3401659]

[33] Tang CZ, Wang YY, Dong ZY, Hu GS, Wang ZG, Wang MJ, Chen HB. XIndex: A scalable learned index for multicore data storage. In: Proc. of the 25th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. San Diego: ACM, 2020. 308–320. [doi: 10.1145/3332466.3374547]

[34] Wang YY, Tang CZ, Wang ZG, Chen HB. SIndex: A scalable learned index for string keys. In: Proc. of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems. Tsukuba: ACM, 2020. 17–24. [doi: 10.1145/3409963.3410496]

[35] Ding JL, Minhas UF, Yu J, Wang C, Do J, Li YN, Zhang HT, Chandramouli B, Gehrke J, Kossmann D, Lomet DB, Kraska T. ALEX: An updatable adaptive learned index. In: Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. Portland: ACM, 2020. 969–984. [doi: 10.1145/3318464.3389711]

[36] Ferragina P, Vinciguerra G. The PGM-index: A fully-dynamic compressed learned index with provable worst-case bounds. Proc. of the VLDB Endowment, 2020, 13(8): 1162–1175. [doi: 10.14778/3389133.3389135]

[37] Marcus R, Kipf A, van Renen A, Stoian M, Misra S, Kemper A, Neumann T, Kraska T. Benchmarking learned indexes. Proc. of the VLDB Endowment, 2020, 14(1): 1–13. [doi: 10.14778/3421424.3421425]

[38] Zhang JY, Gao YH. CARMI: A cache-aware learned index with a cost-based construction algorithm. Proc. of the VLDB Endowment, 2022, 15(11): 2679–2691. [doi: 10.14778/3551793.3551823]

[39] Wu JC, Zhang Y, Chen SM, Wang J, Chen Y, Xing CX. Updatable learned index with precise positions. Proc. of the VLDB Endowment, 2021, 14(8): 1276–1288. [doi: 10.14778/3457390.3457393]

[40] Li PF, Lu H, Zhu R, Ding BL, Yang L, Pan G. DILI: A distribution-driven learned index. Proc. of the VLDB Endowment, 2023, 16(9): 2212–2224. [doi: 10.14778/3598581.3598593]

[41] Wang HX, Fu XY, Xu JL, Lu H. Learned index for spatial queries. In: Proc. of the 20th IEEE Int'l Conf. on Mobile Data Management. Hong Kong: IEEE, 2019. 569–574. [doi: 10.1109/MDM.2019.00121]

[42] Davitkova A, Milchevski E, Michel S. The ML-Index: A multidimensional, learned index for point, range, and nearest-neighbor queries. In: Proc. of the 23th Int'l Conf. on Extending Database Technology. Copenhagen: OpenProceedings.org, 2020. 463–473.

[43] Wang JN, Chen C, Zheng ZB, Chen LN, Zhou YR. Predicting high-dimensional time series data with spatial, temporal and global information. Information Sciences, 2022, 607: 477–492. [doi: 10.1016/j.ins.2022.06.021]

[44] Ding JL, Nathan V, Alizadeh M, Kraska T. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. Proc. of the VLDB Endowment, 2020, 14(2): 74–86. [doi: 10.14778/3425879.3425880]

[45] Zhang SN, Ray S, Lu RX, Zheng YD. SPRIG: A learned spatial index for range and kNN queries. In: Proc. of the 17th Int'l Symp. on Spatial and Temporal Databases. Springer, 2021. 96–105. [doi: 10.1145/3469830.3470892]

[46] Li PF, Lu H, Zheng Q, Yang L, Pan G. LISA: A learned index structure for spatial data. In: Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. Portland: ACM, 2020. 2119–2133. [doi: 10.1145/3318464.3389703]

[47] Qi JZ, Liu GL, Jensen CS, Kulik L. Effectively learning spatial indices. Proc. of the VLDB Endowment, 2020, 13(12): 2341–2354. [doi: 10.14778/3407790.3407829]

[48] Ding XF, Zheng YT, Wang Z, Choo KKR, Jin H. A learned spatial textual index for efficient keyword queries. Journal of Intelligent Information Systems, 2023, 60(3): 803–827. [doi: 10.1007/s10844-022-00752-2]

[49] Li JN, Wang Z, Cong G, Long C, Kiah HM, Cui B. Towards designing and learning piecewise space-filling curves. Proc. of the VLDB Endowment, 2023, 16(9): 2158–2171. [doi: 10.14778/3598581.3598589]

[50] Gao J, Cao X, Yao X, Zhang G, Wang W. LMSFC: A novel multidimensional index based on learned monotonic space filling curves. Proc. of the VLDB Endowment, 2023, 16(10): 2605–2617. [doi: 10.14778/3603581.3603598]

[51] Zhang Z, Jin PQ, Xie XK. Learned indexes: Current situations and research prospects. Ruan Jian Xue Bao/Journal of Software, 2021, 32(4): 1129–1150 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6168.htm [doi: 10.13328/j.cnki.jos.006168]

[52] Galakatos A, Markovitch M, Binnig C, Fonseca R, Kraska T. FITing-Tree: A data-aware index structure. In: Proc. of the 2019 Int'l Conf. on Management of Data. Amsterdam: ACM, 2019. 1189–1206. [doi: 10.1145/3299869.3319860]

[53] Pai S, Mathioudakis M, Wang YH. WaZI: A learned and workload-aware Z-index. In: Proc. of the 27th Int'l Conf. on Extending Database Technology. Paestum: OpenProceedings.org, 2024. 559–571. [doi: 10.48786/EDBT.2024.48]

[54] Yu ZQ, Xhafa F, Chen YH, Ma K. A distributed hybrid index for processing continuous range queries over moving objects. Soft Computing, 2019, 23(9): 3191–3205. [doi: 10.1007/s00500-017-2973-0]

[55] Baride S, Saxena AS, Goyal V. Efficiently mining colocation patterns for range query. Big Data Research, 2023, 31: 100369. [doi: 10.1016/j.bdr.2023.100369]

[56] Yi X, Paulet R, Bertino E, Varadharajan V. Practical approximate k nearest neighbor queries with location and query privacy. IEEE Trans. on Knowledge and Data Engineering, 2016, 28(6): 1546–1559. [doi: 10.1109/TKDE.2016.2520473]

[57] Levchenko O, Kolev B, Yagoubi DE, Akbarinia R, Masseglia F, Palpanas T, Shasha D, Valduriez P. BestNeighbor: Efficient evaluation of kNN queries on large time series databases. Knowledge and Information Systems, 2021, 63(2): 349–378. [doi: 10.1007/s10115-020-01518-4]

[58] Frentzos E, Gratsias K, Theodoridis Y. Index-based most similar trajectory search. In: Proc. of the 23rd Int'l Conf. on Data Engineering. Istanbul: IEEE, 2007. 816–825. [doi: 10.1109/ICDE.2007.367927]

[59] Xie D, Li FF, Phillips JM. Distributed trajectory similarity search. Proc. of the VLDB Endowment, 2017, 10(11): 1478–1489. [doi: 10.14778/3137628.3137655]

[60] Snyder JP. Map Projections: A Working Manual. Washington: U.S. Government Printing Office, 1987. [doi: 10.3133/pp1395]

[61] Fang J. Transverse Mercator projection and Gauss-Kruger projection. Acta Geographica Sinica, 1955, 21(1): 87–99 (in Chinese with English abstract). [doi: 10.11821/xb195501008]

附中文参考文献

[1] 韩士元, 何清, 于自强, 童向荣, 郑渤龙. 面向移动对象连续 k 近邻查询的双层索引结构. 软件学报, 2023, 34(6): 2789–2803. http://www.jos.org.cn/1000-9825/6492.htm [doi: 10.13328/j.cnki.jos.006492]

[4] 蔡盼, 张少敏, 刘沛然, 孙路明, 李翠平, 陈红. 智能数据库学习型索引研究综述. 计算机学报, 2023, 46(1): 51–69. [doi: 10.11897/SP.J.1016.2023.00051]

[5] 柴茗珂, 范举, 杜小勇. 学习式数据库系统: 挑战与机遇. 软件学报, 2020, 31(3): 806–830. http://www.jos.org.cn/1000-9825/5908.htm [doi: 10.13328/j.cnki.jos.005908]

[6] 李国良, 周煊赫, 孙佶, 余翔, 袁海涛, 刘佳斌, 韩越. 基于机器学习的数据库技术综述. 计算机学报, 2020, 43(11): 2019–2049. [doi: 10.11897/SP.J.1016.2020.02019]

[7] 巢成, 蒲非凡, 许建秋, 高云君. 基于空间位置关系的轨迹数据高效降维和查询算法. 计算机研究与发展, 2024, 61(7): 177117) 与发展法[doi: 10.7544/issn1000-1239.202330609]

[25] 丁凯孟, 朱长青, 卢付强. 基于自适应格网划分的遥感影像感知哈希认证算法. 武汉大学学报 (信息科学版), 2015, 40(6): 716–720, 743. [doi: 10.13203/j.whugis20130567]

[51] 张洲, 金培权, 谢希科. 学习索引: 现状与研究展望. 软件学报, 2021, 32(4): 1129–1150. http://www.jos.org.cn/1000-9825/6168.htm [doi: 10.13328/j.cnki.jos.006168]

[61] 方俊. 横轴墨卡托投影和高斯-克吕格投影. 地理学报, 1955, 21(1): 87–99. [doi: 10.11821/xb195501008]

作者简介

王撷阳, 博士生, CCF 学生会员, 主要研究领域为移动对象数据库.

巢成, 硕士生, 主要研究领域为移动对象数据库.

金鑫, 博士生, 主要研究领域为数据管理.

许建秋, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为数据软件, 空间数据库, 移动对象数据库, 可扩充数据库.

高云君, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为数据库, 大数据管理与分析, DB 与 AI 融合.