软件学报 ISSN 1000-9825, CODEN RUXUEW
[doi: 10.13328/j.cnki.jos.007554] [CSTR: 32375.14.jos.007554]

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

SPBSpark: 高查询效能的分布式轨迹索引方法*

汤 娜, 蔡锶淇, 李晶晶, 罗凯原, 汤 庸

(华南师范大学 计算机学院, 广东 广州 510613)

通信作者: 李晶晶, E-mail: lijingjing@scnu.edu.cn

摘 要: GPS (global positioning system, 全球定位系统) 移动设备与 5G (5th generation mobile communication technology, 第 5 代移动通信技术) 互联网技术的普及催生了轨迹数据的飞速增长. 如何对海量轨迹数据进行高效地存储、管理和分析成为当前环境下的研究热点问题. 传统的单节点式轨迹索引受限于内存容量、磁盘 I/O 速度等问题已经无法胜任海量轨迹数据的管理. Spark 作为一种基于内存计算的分布式框架, 在处理海量数据时具备天然的优势. 因此, 提出了基于 Spark 平台的分布式轨迹数据索引以及相关的查询技术方案. 为了提升分布式集群中单个节点的数据存储能力和轨迹查询效率, 首先提出了一种轨迹编码技术 (Z-order trajectory encoding, ZTE), 该技术对轨迹 MBR (minimum bounding rectangle, 最小外接矩形) 所覆盖的最小相邻子空间进行编码, 可以表达不同粒度的轨迹以及轨迹的运动方向, 用于判断轨迹与查询空间的关系. 基于这一技术, 将轨迹的 ZTE 编码进一步组织成偏序结构, 设计了基于子空间偏序分支的 SPB 分支 (subspace partial-order branch, SPB) 并结合哈希映射表 IDMap 构建局部索引. 索引能够避免类 R 树索引中最小限定矩形堆叠形成死空间导致的效率低下问题, 实现快速剪枝. 为了支持海量轨迹数据的高效检索, 基于 SPB 分支的局部索引设计了分布式的轨迹索引 SPBSpark. SPBSpark 主要包括数据分区、局部索引和全局索引这 3 个部分. 该索引能有效支持时空范围查询、k 近邻查询、移动对象轨迹查询这 3 种查询. 最后, 选取了同样基于 Spark 框架的分布式轨迹索引 TrajSpark 和 LocationSpark 作为实验对照对象. 通过仿真实验对比分析, SPBSpark 索引的空间利用率在 LocationSpark 上改善了约 15%. 在查询性能上, 相较于 TrajSpark 和 LocationSpark, SPBSpark 拥有 2–3 倍的性能提升.

关键词: 分布式索引; Spark; 轨迹; 时空查询; kNN

中图法分类号: TP311

中文引用格式: 汤娜, 蔡锶淇, 李晶晶, 罗凯原, 汤庸. SPBSpark: 高查询效能的分布式轨迹索引方法. 软件学报. http://www.jos.org.cn/1000-9825/7554.htm

英文引用格式: Tang N, Cai SQ, Li JJ, Luo KY, Tang Y. SPBSpark: Distributed Trajectory Indexing Method with High Query Efficiency. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7554.htm

SPBSpark: Distributed Trajectory Indexing Method with High Query Efficiency

TANG Na, CAI Si-Qi, LI Jing-Jing, LUO Kai-Yuan, TANG Yong

(School of Computer Science, South China Normal University, Guangzhou 510613, China)

Abstract: The popularization of GPS mobile devices and 5G Internet technology has led to the rapid growth of trajectory data. How to efficiently store, manage, and analyze massive trajectory data has become a hot research issue in the current environment. The traditional single-node trajectory index is limited by memory capacity, disk I/O speed, and other factors, and is no longer capable of managing large-scale trajectory data. Spark, as a distributed framework based on in-memory computing, has natural advantages in processing massive data. Therefore, this study proposes a distributed trajectory data indexing and query scheme based on the Spark platform. To improve the data storage capacity of a single node in a distributed cluster and the efficiency of trajectory queries, a trajectory encoding technique, Z-order

* 基金项目: 国家重点研发计划 (2023YFC3341204); 国家自然科学基金 (62377014, 62407016)
收稿时间: 2025-05-28; 修改时间: 2025-06-24, 2025-08-22; 采用时间: 2025-09-19; jos 在线出版时间: 2026-01-07

trajectory encoding (ZTE), is proposed. This technique encodes the minimum adjacent subspaces covered by the trajectory minimum bounding rectangle (MBR), which can represent trajectories of different granularities and their movement directions, and is used to determine the relationship between a trajectory and the query space. Based on this technique, this study further organizes the ZTE codes of trajectories into a partial-order structure and designs a subspace partial-order branch (SPB). Combined with the hash mapping table IDMap, a local index is constructed. This index avoids the inefficiency caused by the dead space formed by the overlapping of minimum bounding rectangles in R-tree-like indexes and enables fast pruning. To support efficient retrieval of massive trajectory data, the study designs a distributed trajectory index named SPBSpark based on the SPB-branch local index. SPBSpark mainly consists of three components: data partition, local index, and global index. The proposed index effectively supports three types of queries: spatiotemporal range query, k-nearest neighbor query, and moving object trajectory query. Finally, the study selects the distributed trajectory indexes TrajSpark and LocationSpark, which are also based on the Spark framework, as comparison systems. Through comparative simulation experiments, the spatial utilization of the SPBSpark index is improved by about 15% compared with LocationSpark. In terms of query performance, SPBSpark achieves a 2–3 times performance improvement compared with TrajSpark and LocationSpark.

Key words: distributed index; Spark; Trajectory; spatio-temporal query; kNN

在当今数字化时代, 随着智能移动终端与 5G 技术的广泛普及, 基于位置的服务 (location based service, LBS)[1] 已成为支撑社会运行的核心基础设施, 涵盖地图服务、导航系统以及一系列基于位置的社交网络和推荐系统等[2], 这些应用和服务催生了轨迹大数据. 所谓轨迹数据, 是指移动对象在地理空间中运动时生成的一系列时空数据点, 这些数据点按照时间顺序排列形成的曲线被称为轨迹[3]. 轨迹作为移动对象运动行为的连续性时空表征, 由有序位置序列 $T_r=\{p_0,p_1,\ldots,p_n\}$ 构成. 其中每个采样点 $p_i=\{O_{\mathrm{id}},lon_i,lat_i,timestamp_i,attr_i\}$ 包含 5 维属性: 移动对象标识 O_{id} 唯一确定轨迹归属实体, 空间坐标 (lon_i,lat_i) 记录轨迹点的经纬度信息, 时间戳 $timestamp_i$ 标记采集时刻, 动态属性 $attr_i$ 描述瞬时运动状态 (如速度、方向等). 此类数据具有显著的大数据 4V 特征 (volume、variety、velocity、veracity): 规模上, 滴滴平台每日处理量超 70 TB; 速度上, 物流轨迹采样频率达秒级; 多样性上, 融合时空坐标、移动速度、方向等多维属性; 价值密度上, 需通过高效计算提取潜在规律. 传统单机系统受限于 I/O 吞吐量与计算瓶颈, 难以应对海量数据实时处理需求; 基于 Hadoop 的分布式方案因 MapReduce 的磁盘密集型计算范式, 无法满足低延迟查询要求. 在此背景下, 基于 Spark 的分布式轨迹索引技术凭借内存计算架构与弹性分布式数据集 (resilient distributed dataset, RDD) 特性[4], 为破解“存储-计算-查询”协同优化难题提供了新路径, 但其在编码策略的几何适配性、索引结构的动态维护成本及复杂查询支持能力等方面仍面临严峻挑战.

现有研究通过全局-局部混合索引与空间填充曲线实现了多维数据的高效分片与查询优化, 并在分布式系统中结合动态负载均衡策略以应对数据演化. 但针对轨迹等非点数据, 编码仍存在局限性, 传统空间填充曲线难以保留轨迹存储的局部邻近性与运动方向语义, 且 R 树索引结构易引发节点重叠与计算冗余[5]. 现有系统多依赖静态分区或经验性参数, 难以平衡复杂查询效率与动态负载适应性. 本文针对这些问题, 做出了以下贡献.

(1) 提出一种面向轨迹 MBR 的编码方法 (Z-order trajectory encoding, ZTE). 该方法基于 Z-order 曲线[6], 对轨迹 MBR 的最小相邻子空间 MAS (minimal adjacent subspace) 进行编码, 通过 MAS 的 3 种位置编码模式表征轨迹运动方向, 构建由空间位置、运动方向、分辨率层级组成的四元编码结构. 编码可以表达不同分辨率的轨迹以及轨迹的运动方向, ZTE 编码的核心作用是用于辅助判断轨迹段与查询空间之间的相交和包含关系.

(2) 提出轨迹索引结构 SPB (subspace partial-order branch) 分支, 通过轨迹 MBR 的 MAS 编码在二维平面的位置包含关系构建 MAS 偏序分支结构, 并提出序分支剪枝定理来进行快速剪枝. 轨迹的原始采样点数据未直接参与编码, 而是以轨迹片段集合的形式组织在局部哈希映射表 IDMap 中, IDMap 将移动对象 ID 映射至对应轨迹片段的时序采样点集合, 从而在保持轨迹空间局部邻近性的同时, 支持对轨迹点数据的快速访问与定位. 在查询阶段, SPB 分支可高效定位潜在匹配的轨迹, 再结合 IDMap 进行轨迹片段的精确过滤与返回.

(3) 采用均衡的分区方案, 在基于 R 树和双哈希表的全局索引结构中引入路径数量, 最终提出支持 k 近邻 (k-nearest neighbors, kNN) 查询、时空范围查询和移动对象轨迹查询这 3 种查询方式的分布式轨迹索引 SPBSpark. 实验表明, 相比选取 R 树或哈希结构作为索引方案的 TrajSpark 和 LocationSpark, SPBSpark 在查询方面拥有 2–3 倍的性能提升.

索引的设计具有以下特点: (1) 存储模型中实现了轨迹元数据 (轨迹的空间范围)-实体数据 (实体轨迹点数据) 分离. 轨迹点以片段集合的形式按时间顺序组织在 IDMap 中, 轨迹的 ZTE 编码以偏序存储在 SPB 分支中, 这种存储模型保证了轨迹点数据存储的空间局部邻近性, 避免了直接对轨迹点编码排序引发的空间打散问题. (2) SPB 基于轨迹 MAS 编码的偏序关系构建, 有效避免了 R 树中最小外接矩形堆叠带来的“死空间”, 其查询机制避免了查询过程中多路径遍历的问题, 支持快速轨迹过滤与高效剪枝. (3) 为提高 kNN 算法的效率, 本文在全局索引结构中引入了路径数量目录的辅助机制, 结合算法优化策略, 实现了通过全局索引对空间搜索范围的快速判定, 避免了传统方法中全局与局部索引之间多轮交替筛选.

本文第 1 节介绍相关工作. 第 2 节介绍如何设计索引, 包括索引整体架构的设计原理和思路. 第 3 节介绍构建索引、查询和更新索引的算法. 第 4 节介绍实验对比结果及分析. 第 5 节总结本文工作并对下一步工作做出展望.

1 相关工作

用于处理大规模空间数据的内存集群计算框架 GeoSpark[7]扩展 Spark 的 RDD 以支持空间数据, 并且支持网格划分、四叉树划分等多种空间索引, 为近年来基于 Spark 的轨迹索引方案提供了理论依据. 空间内存大数据分析系统 Simba[8]旨在为空间大数据提供高效查询和分析系统, 采用 IndexRDD 的全局索引和局部索引策略支持范围查询、距离查询、kNN 查询以及距离连接查询, 优化了空间查询, 但不支持时空查询. 除了 GeoSpark、Simba 外, Magellan[9]、SpatialSpark[10]等系统也支持 Spark 上的空间操作, 但查询偏斜、缺乏适应性以及过多且未优化的网络和 I/O 通信开销阻碍了系统的性能. 这些工作主要致力于构建基于 Spark 的大规模轨迹数据通用管理平台, 但其处理范围仅限于空间维度, 并未引入时间属性. 平台中通常采用如 R 树和基于网格 (grid-based) 的方法作为基本的数据索引结构. 不同平台之间的差异主要体现在是否支持更复杂的几何对象类型及空间操作, 从而提升其在多维空间数据处理中的灵活性与表达能力.

近年来, 学者针对上述问题展开了深化研究, 不仅通过新型索引结构 (KDB 树[11]、R 树[12]、四叉树[13]) 和计算范式显著优化了网络通信与计算效率, 更通过时空联合索引机制突破了传统空间计算系统在时序维度上的局限性. Wang 等人[14]通过 KDB 树与 R 树混合索引架构, 结合时空边缘校正权重缓存和定制序列化机制, 在分布式计算中实现了时空双重维度的优化. LocationSpark[15]引入查询调度器 sFilter, 通过双层 R 树索引与负载均衡策略同步解决空间查询效率与数据倾斜问题. DITA 系统[16]则通过 R 树全局索引与 Trie 树局部索引的异构融合, 辅以基于成本的负载均衡技术, 将优化范围进一步扩展至轨迹分析领域. Beast[17]通过 R*-Grove 分区技术, 引入最小分区大小比例和样本加权机制, 在静态分区中实现存储与计算的双重负载均衡. TMan[18]通过 TR 索引和 TShape 索引与分布式键值存储优化, 减少冗余存储并降低时空范围查询的检索次数, 利用并行查询下推策略突破传统空间计算系统在时序与空间联合检索中的性能瓶颈. 在既有研究围绕时空索引优化与负载均衡取得阶段性突破的基础上, 进一步聚焦轨迹数据的高效管理与动态适应性优化, 以应对大规模轨迹分析场景中的实时性与演化性需求. Dragoon 框架[19]通过 mRDD 模型实现历史轨迹与流式轨迹的统一管理, 利用 R 树构建全局-局部双层索引以加速历史轨迹查询, 同时通过可变 RDD 动态更新实时轨迹状态, 突破了传统静态 RDD 模型对时序数据连续演化的支持瓶颈. 针对轨迹数据的时空关联性与分布动态性设计的 TrajSpark 系统[20]通过 IndexRDD 封装轨迹数据与局部索引, 结合全局多层混合索引与时间衰减模型, 实时感知数据分布偏移并触发自适应分区重构, 在索引效率与系统鲁棒性之间实现动态平衡. 这些工作普遍采用多层 R 树索引结构作为空间划分与检索的核心机制, 但 R 树在高维或密集轨迹数据场景下, 容易出现最小外接矩形 (MBR) 之间的大量重叠, 形成“死空间”, 从而导致查询过程中需访问多个冗余节点, 降低检索效率与系统吞吐能力.

在分布式时空计算系统的演进中, 全局-局部混合索引与空间填充曲线 (space-filling curve, SFC)[21]的技术路径呈现协同趋势. Sagan[22]的理论研究为此类协同奠定了数学基础, 其通过定义多维至一维的局部性保持映射 (如 Hilbert[23]、Peano 曲线[24]), 提出以不规则性度量为核心的多维空间线性化质量评估框架, 为高维数据的高效压缩与检索提供了通用解决方案. Hughes 等人[25]进一步将 SFC 与分布式架构结合, 设计 GeoMesa 系统——以 Hilbert 曲线构建全局索引, 结合列式存储与局部索引, 实现矢量与栅格数据的统一管理, 并通过 SFC 的全局编码优化数据分片的局部性与负载均衡. TrajMesa[26]基于 GeoMesa 的 XZ2 索引策略, 构建分布式 NoSQL 轨迹存储引擎, 通过全局

空间范围索引表支持轨迹数据的空间范围查询、相似性查询与 kNN 查询. HGST[27]方法基于 Hilbert 空间填充曲线优化空间聚类特性, 结合 GeoSOT 模型构建时空联合索引机制, 通过自适应时空缩放编码策略动态平衡时空查询尺度, 并利用 Spark 分布式框架实现高效并行处理. 现有编码技术 (如 Z-order、Hilbert 曲线等), 虽广泛应用于空间点数据的压缩与索引构建, 但直接用于非点类对象 (如轨迹段、线段或多边形) 时, 普遍存在如下局限性: 首先, 这些编码方法无法反映轨迹的空间形状、方向信息及延展特征, 导致轨迹等具有几何结构的数据在编码过程中结构信息丢失; 其次, 在映射过程中, 同一个轨迹中物理空间中相邻的点可能被映射至编码空间中相距较远的位置, 破坏了轨迹的空间局部邻近性 (spatial locality), 从而削弱了索引在空间剪枝与过滤过程中的效率; 最后, 部分编码方法的计算复杂度随对象复杂度 (如轨迹采样点数量) 上升而显著增加, 不利于大规模分布式轨迹数据的处理与索引构建.

2 索引设计

2.1 索引架构

图 1 给出了 SPBSpark 的整体框架. SPBSpark 共包括 3 层: Apache Spark 层、轨迹预处理层、索引层.

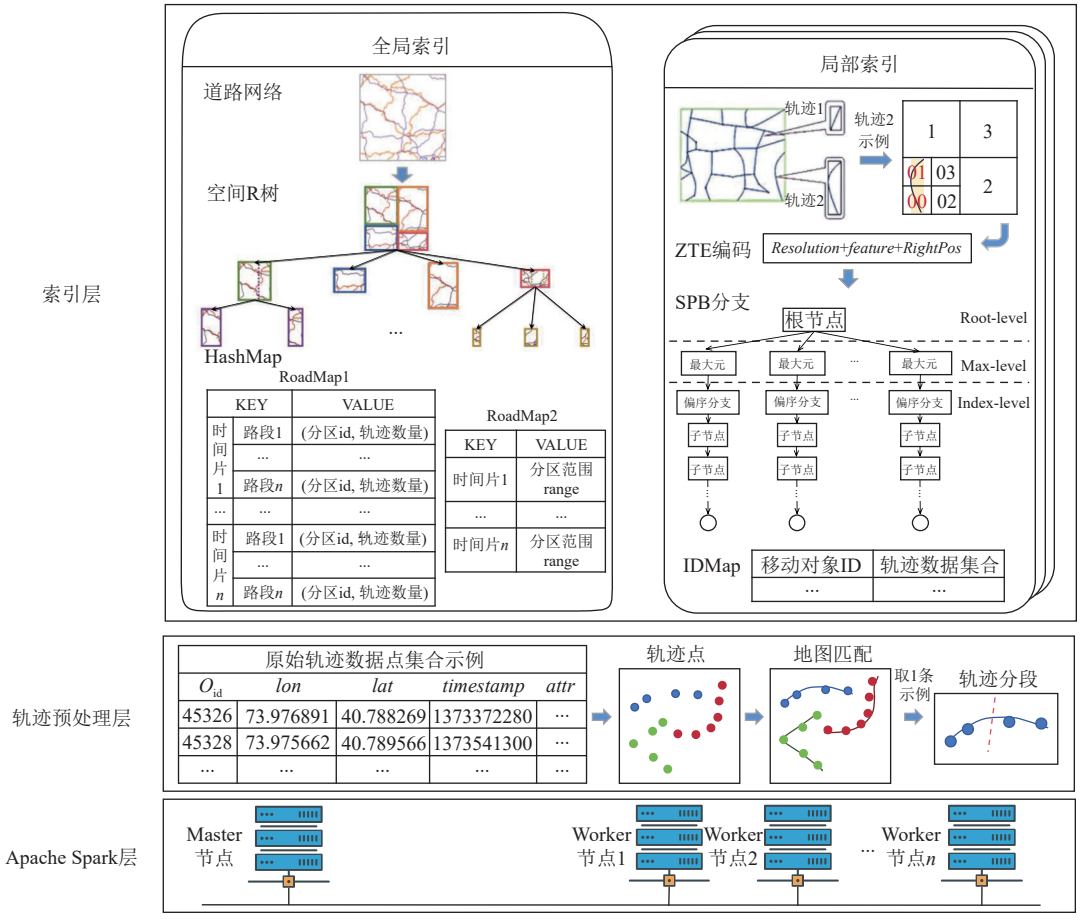

图 1 SPBSpark 整体框架示意图

(1) Apache Spark 层: 该层是由 Spark 提供的 RDD 以及其常用的操作构成的, 是整个索引的基础框架和计算引擎, 其能力直接继承自 Apache Spark 核心框架.

(2) 轨迹预处理层: 该层接收原始的轨迹点作为输入, 这些轨迹点来源于 GPS 终端设备对移动对象在不同时刻的位置进行连续记录, 形成一组按照时间顺序排列的数据点序列. 然后利用基于相互投票的轨迹地图匹配

(interactive voting-based map matching, IVMM) 算法[28]和分布式计算框架 Spark 执行轨迹分段[29]. 其中轨迹地图匹配将偏离道路网络的 GPS 点映射至道路上, 并过滤掉无法匹配道路的脏数据. 轨迹分段是指将一个移动对象的长轨迹划分成多个有意义的短轨迹段, 轨迹太长跨分区不利于存储和构建索引.

(3) 索引层: 该层对预处理过后的轨迹数据构建索引, 根据索引的类型可分为全局索引和局部索引. 全局索引构建在 Master 节点上, 全局索引由 R 树和两张哈希表 RoadMap1 和 RoadMap2 构成. 其作用有两个: 一是快速过滤掉不符合查询时间或者时空条件的分区, 二是在 kNN 查询中仅通过全局索引和优化算法就能快速确定查询的时空范围. 全局索引的上层是对整张道路网络构建的空间 R 树, 用于过滤出符合空间窗口的路段, 下层维护了两张哈希表: 第 1 张哈希表 RoadMap1 的作用是映射每一个时间分片不同路段的分区信息及该路段上的轨迹数量, 第 2 张哈希表 RoadMap2 的作用是映射不同时间分片下的分区范围; 局部索引构建在 Worker 节点上, 本文为 Worker 节点上的每一个数据分区构建了一个 SPB 分支和一张哈希映射表 IDMap 作为局部索引, 用于准确地筛选和匹配出符合查询条件的全部轨迹点数据. 其中 SPB 分支用于快速地剪枝, 筛选出符合条件的轨迹隶属的移动对象 id, 然后通过 IDMap 中的映射关系找到移动对象 id 对应的轨迹片段点数据集合, 最后对轨迹数据进行精准过滤.

本轨迹索引技术涵盖了从数据预处理、空间分区、局部索引构建、全局索引组织到多级查询处理的完整流程, 形成了面向大规模轨迹数据的分布式索引与查询优化框架. 首先, 系统需对原始轨迹数据进行预处理, 包括去噪、路网地图匹配以及轨迹分段等. 随后为保证分布式环境下的负载均衡性, 轨迹数据会基于空间与时间属性采用多级分区策略划分至不同计算节点. 针对每个分区内的数据, 系统进一步构建高效的局部索引结构, 以加快对局部数据的检索与剪枝过程. 同时为了支持全局范围内的快速分区过滤与查询调度, 还需在全局层面建立时间或时空混合特征的全局索引结构, 以减少无效分区访问.

SPBSpark 能够高效支持 3 种常用的轨迹查询类型, 包括时空范围查询、kNN 查询和移动对象轨迹查询. 在实际时空范围查询和移动对象轨迹查询过程中, 系统会先利用全局索引快速定位候选数据分区, 再结合局部索引对分区内轨迹数据进行高效检索与匹配. kNN 查询利用全局索引快速确定查询的时空范围, 最终将 kNN 查询转化为时空范围查询.

2.2 分区方案

本文先对轨迹数据集采用相同粒度的时间段进行切片, 将数据的时间属性按照固定长度的周期进行切分得到多个时序分片. 为了满足不同的查询需求, 可将时间片的长度设置为小时、天、周等. 对于每个时间分片下的轨迹点集, 运行排序-平铺-递归 (sort-tile-recursive, STR)[30]算法, 将轨迹数据递归划分成多个不相交的分区. 根据数据分布疏密的程度, 对空间进行不均等分割, 使得数据的数量自适应地平均划分, 保证了每个节点的负载均衡, 如后文图 2 所示. 在分区结束之后, 采用带有虚拟节点的一致性哈希算法映射数据分区到对应的物理节点上.

由于不同时间片下的轨迹数据在道路网络中的分布情况是不同的, 因此道路网络中的路段集合会随着轨迹分布变化在不同时间片下呈现不同的分区情况.

2.3 全局索引

全局索引由路段的 R 树+两张哈希表: RoadMap1 和 RoadMap2 构成. 在数据分区阶段, 轨迹数据集在每个时间分片下采用 STR 算法进行空间分区. 因此, 最终生成的空间 R 树数量等同于时序分片的数量. 然而, 若直接收集所有 R 树的根节点以构建全局索引, 存储和查询开销将极为庞大. 因此本文基于局部索引 RDD 收集的分区标识信息、轨迹数据路段信息及分区内轨迹数量信息, 构建了一种多层混合式全局索引, 如图 3 所示. 最终在全局索引的上层仅针对路网数据构建了一棵 R 树作为空间索引, 该方法能够利用空间查询窗口的参数, 快速定位符合查询范围的路段. 在全局索引的下层, 基于局部索引阶段收集的各分区涉及的路段信息, 构建了两张哈希表 RoadMap: (1) RoadMap1 用于映射每个时间分片中不同路段的分区信息, 主要服务于时空联合查询场景, 即在查询条件同时包含时间范围与空间范围时, 系统可根据指定时间片内轨迹所在的空间路段编号, 快速筛选出可能命中查询范围的目标分区, 从而减少无效分区访问. (2) RoadMap2 用于映射每个时间片的分区信息, 适用于仅基于时间条件的查询场景. 当查询请求仅指定时间范围、无空间约束时, 系统可通过该表直接确定与目标时间片相关的全部数据分

区, 无需进行空间层次的进一步过滤. 通过这两类时空维度的分层映射机制, 从而通过全局索引快速定位计算要涉及的分区. 引入了路径数量的辅助机制, 在 kNN 查询中仅通过全局索引就能快速地确定查询的时空范围, 避免了传统方法中全局与局部索引之间的多轮交替筛选.

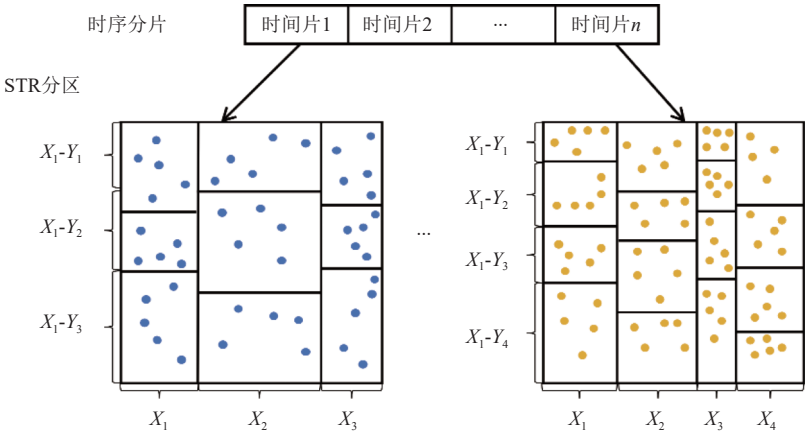

图 2 数据分区示意图

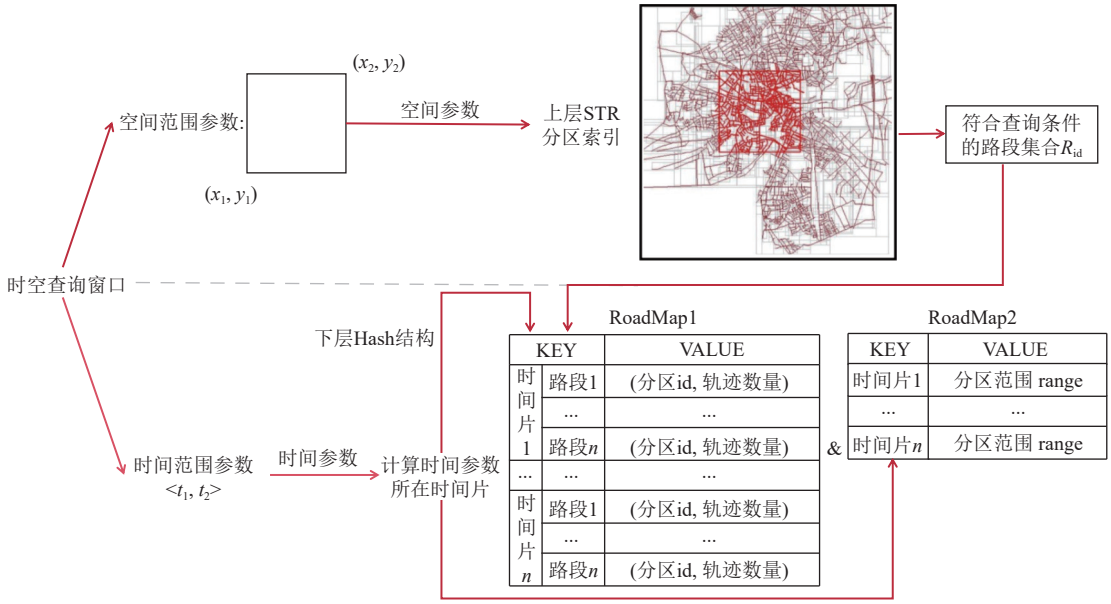

图 3 SPBSpark 全局索引示意图

2.4 局部索引

局部索引由 SPB 偏序分支和 IDMap 哈希表构成. 其中 SPB 分支用于快速地剪枝, 筛选出符合条件的轨迹隶属的移动对象 id, 然后通过 IDMap 中的映射关系找到移动对象 id 对应的轨迹片段点数据集合, 最后精准过滤. 经过数据划分之后, 轨迹数据集可以看作是一个按照时间信息、路段信息划分好的点对象构成的 RDD, 本文称之为点 RDD. 但在构建局部索引之前, 需要对轨迹数据进行一系列预处理以达到压缩数据的目的. 对于每一个分区内的数据, 首先将轨迹点按照移动对象唯一标识进行分组, 随后在每一个分组内按时间顺序对轨迹点排序, 生成连续的轨迹, 随后对较长的轨迹进行轨迹分段处理. 在经过轨迹分段之后, 会首先将每条轨迹段中全部轨迹点的经纬度排序, 并分别记录最大和最小的经度值、纬度值作为 MBR 的左下点和右上点坐标, 随后针对每条轨迹段的 MBR

对轨迹数据进行 ZTE 编码. 将生成的编码作为建立索引时的概要数据, 并在每个分区建立 SPB 分支作为局部索引, 以方便在查询时有效地剪枝. 接着需要对轨迹数据进行压缩处理. 其中同一个属性的值会被连续存放并压缩. 针对不同的数据类型, 可以采用不同的压缩形式, 例如数值型数据采用差值的形式压缩, 字符串类型采用 gzip 的形式压缩. 本文还为每个分区增加一个局部哈希索引 IDMap, 该索引用于将移动对象唯一标识和轨迹片段映射起来以实现针对移动对象的轨迹查询. 最后, 将整体打包成一个新的 RDD, 称之为局部索引 RDD, 并收集每个分区的唯一标识、分区数据的时空信息、路段信息等, 方便后续建立全局索引.

2.4.1 基于 Z 曲线的轨迹编码 ZTE

定义 1. ZTE 编码. 对于轨迹数据的矩形 MBR, 其左下角和右上角坐标分别为 (lat_l, lon_l) 和 (lat_r, lon_r). ZTE 编码通过对 MBR 所在空间平面进行 Z-order 递归划分来描述轨迹数据的空间位置. 递归划分的终止条件为: 当轨迹数据矩形的左下点坐标所在的子空间 s_1 与右上点坐标所在的子空间 s_2 不相邻或不重叠时, 划分过程结束. ZTE 编码表示为 (*Resolution*, *LB*, *RT*, *RightPos*), 其中, *Resolution* 表示分辨率, 对应递归划分的层次, 用于确定空间划分的粒度; *LB* (left-bottom) 表示矩形左下点坐标所在子空间通过 Z 曲线生成的编码, 并转换为十进制数, 用于唯一标识矩形左下点所在的子空间位置; 同理, *RT* (right-top) 表示右上点坐标经转换生成的十进制数, 用于唯一标识矩形右上点所在的子空间位置; 特征 $feature = (LB, RT)$ 形成一个用于描述矩形空间特征的组合表示; *RightPos* (right position) 称为右方子空间位置, 代表矩形右上点坐标所在的子空间相较于矩形左下点坐标所在子空间的位置关系, 若 s_2 与 s_1 相邻, 则 s_2 会出现在 s_1 的正上方、右斜上方、正右方这 3 种位置关系. 子空间的位置关系由 3 位的二进制编码表示, 根据矩形右上点坐标所在的子空间来确定 *RightPos* 的值, 共有 3 种情况, 分别为正上方 (100), 右上方 (010), 正右方 (001).

例 1: 以图 4 所示的矩形 tr_1 为例, 记录递归的次数为分辨率 (*Resolution*), 当分辨率为 1 时, 矩形 tr_1 的左下点坐标所在子空间 s_1 和右上点坐标所在子空间 s_2 均在编码为“0”的子空间内, 子空间相同, 不满足递归结束的条件; 当分辨率为 2 时, s_1 是编码为“00”子空间, s_2 是编码为“01”子空间, 子空间相邻, 递归继续; 当分辨率为 3 时, s_1 是编码为“001”子空间, s_2 是编码为“013”子空间, 显然是不相邻的子空间, 此时递归结束, 最终选择最小相邻子空间“00”和“01”表示矩形 tr_1, 且“01”子空间位于“00”子空间的正上方, 因此 tr_1 对应的 ZTE 编码为 (2, 0, 1, 100).

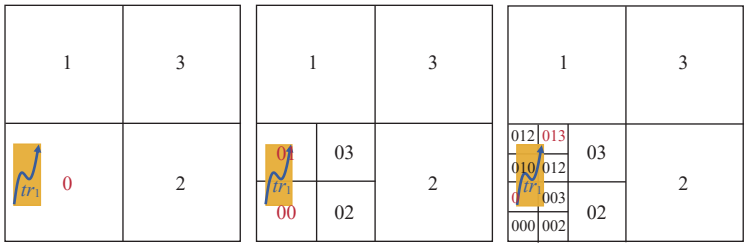

图 4 ZTE 编码

在 ZTE 编码中的递归划分终止条件判断, 我们采用以下形式化描述: 假设左下点 S_l 的四进制空间编码为 $A = a_1a_2,\ldots,a_n$, 其中 $a_i \in \{0,1,2,3\}$ 表示的是 A 中第 i 位的数字. 用 $Inc_k(A_{p:q})$ 表示 A 从第 p 位到第 q 位整体加 k, k 为整数. 特别的, $Inc_k(A)$ 表示四进制数 A 整体加 k. 因此, 仅需验证右上点 S_r 的空间编码是否位于 S_l 编码 A 的上 ($Up(A)$)、右 ($Right(A)$)、右上 ($UpR(A)$) 这 3 个方向的邻域范围内:

$$Up(A) = \begin{cases} Inc_1(A), & \text{if } a_n \in \{0,2\} \\ \varnothing, & \text{if } a_i \in \{1,3\},\ \forall 1 \leqslant i \leqslant n \\ Inc_1(A_{1:i-1}) \,\|\, (A_{i:n} - 1), & \text{otherwise} \end{cases} \tag{1}$$

$$Right(A) = \begin{cases} Inc_2(A), & \text{if } a_n \in \{0,1\} \\ \varnothing, & \text{if } a_i \in \{2,3\}, \forall 1 \leqslant i \leqslant n \\ Inc_2(A_{1:i-1}) \,\|\, (A_{i:n} - 2), & \text{otherwise} \end{cases} \tag{2}$$

$$UpR(A)=\begin{cases} Up(A)+Right(A)-A, & \text{if } Up(A)\neq\varnothing\wedge Right(A)\neq\varnothing \\ \varnothing, & \text{otherwise} \end{cases} \tag{3}$$

其中, $\varnothing$ 表示 A 到达该方向的边界, 则该方向无相邻空间.

定义 2. 最小相邻子空间 MAS. 设 s_1, s_2 分别为轨迹数据的矩形 MBR 通过 ZTE 编码的左下点和右上点坐标所在的子空间, 称轨迹的最小相邻子空间, 记作 $\mathrm{MAS}(s_1, s_2)$. 由于 MBR 的 ZTE 编码表示为 (*Resolution*, *LB*, *RT*, *RightPos*), 故 $\mathrm{MAS}.LB$ 和 $\mathrm{MAS}.RT$ 分别为子空间 s_1 和 s_2 的 Z-order 编码. $\mathrm{MAS}.feature = (\mathrm{MAS}.LB, \mathrm{MAS}.RT)$, $\mathrm{MAS}.res$ 表示子空间的分辨率, $(\mathrm{MAS}.x_{LB}, \mathrm{MAS}.y_{LB})$ 代表 MBR 左下角的坐标, $(\mathrm{MAS}.x_{RT}, \mathrm{MAS}.y_{RT})$ 代表 MBR 右上角的坐标.

MBR 作为二维的时空数据矩形, 若直接对其构建索引, 处理效率通常是比较低的. 本文基于 Z 曲线[22]提出针对轨迹数据矩形的 ZTE 编码技术. 首先, 抽取出轨迹数据矩形 MBR 的左下点和右上点坐标 (lat_l, lon_l, lat_r, lon_r), 然后按照类似 XZ-ordering[31]的方式对空间平面进行递归划分, 递归结束的条件是轨迹数据矩形左下点坐标所在的子空间 s_1 和右上点坐标所在的子空间 s_2 不相邻或不相同, 利用 Z 曲线平面中恰好能够包含轨迹 MBR 的最小相邻子空间来表示轨迹数据矩形.

2.4.2 最小相邻子空间的偏序分支结构

轨迹数据矩形 MBR 在经过编码之后, 所生成 ZTE 编码中的 *feature* 值所代表的最小相邻子空间 MAS 集合在不同分辨率下存在偏序关系并可以组成序的分支结构, 可形式化定义如下.

定理 1. MAS 偏序关系. 设 γ 为用于表示轨迹的最小相邻子空间 (MAS) 的数据集合, 存在 MAS_i、$\mathrm{MAS}_j \in \gamma$, $res_i = \mathrm{MAS}_i.resolution$, $res_j = \mathrm{MAS}_j.resolution$, 若 $\mathrm{MAS}_i \subseteq \mathrm{MAS}_j$, 即 $(\mathrm{MAS}_i.x_{LB} \geqslant \mathrm{MAS}_j.x_{LB}) \wedge (\mathrm{MAS}_j.x_{RT} \geqslant \mathrm{MAS}_i.x_{RT}) \wedge (\mathrm{MAS}_i.y_{LB} \geqslant \mathrm{MAS}_j.y_{LB}) \wedge (\mathrm{MAS}_j.y_{RT} \geqslant \mathrm{MAS}_i.y_{RT})$, 那么称 MAS_i 和 MAS_j 存在偏序关系, 记作 $\mathrm{MAS}_i \preccurlyeq \mathrm{MAS}_j$. 即若集合严格满足 $\left\{\frac{\mathrm{MAS}_i.LB}{4^{(res_i-1)}}, \frac{\mathrm{MAS}_i.RT}{4^{(res_i-1)}}\right\} \subseteq \left\{\frac{\mathrm{MAS}_j.LB}{4^{(res_j-1)}}, \frac{\mathrm{MAS}_j.RT}{4^{(res_j-1)}}\right\}$, 则满足 $\mathrm{MAS}_i \preccurlyeq \mathrm{MAS}_j$. 特别的, 存在 $UpR(\mathrm{MAS}_j.LB) = \mathrm{MAS}_j.RT$, 若满足 $\left\{\frac{\mathrm{MAS}_i.LB}{4^{(res_i-1)}}, \frac{\mathrm{MAS}_i.RT}{4^{(res_i-1)}}\right\} \subseteq \left\{\frac{\mathrm{MAS}_j.LB}{4^{(res_j-1)}}, \frac{\mathrm{MAS}_j.RT}{4^{(res_j-1)}}, Up(\mathrm{MAS}_j.LB), Right(\mathrm{MAS}_j.LB)\right\}$, 也满足 $\mathrm{MAS}_i \preccurlyeq \mathrm{MAS}_j$. 其中, $\preccurlyeq$ 是集合 γ 上满足自反性、反对称性、传递性的偏序关系.

证明: 以下将分别证明 MAS 偏序关系的自反性、反对称性和传递性.

● 自反性: 对于所有的 $\mathrm{MAS}_i \in \gamma$, 需要证明 $\mathrm{MAS}_i \preccurlyeq \mathrm{MAS}_i$. 根据条件, 因为 $(\mathrm{MAS}_i.x_{LB} \geqslant \mathrm{MAS}_i.x_{LB}) \wedge (\mathrm{MAS}_i.x_{RT} \geqslant \mathrm{MAS}_i.x_{RT}) \wedge (\mathrm{MAS}_i.y_{LB} \geqslant \mathrm{MAS}_i.y_{LB}) \wedge (\mathrm{MAS}_i.y_{RT} \geqslant \mathrm{MAS}_i.y_{RT})$, 即满足集合 $\left\{\frac{\mathrm{MAS}_i.LB}{4^{(res_i-1)}}, \frac{\mathrm{MAS}_i.RT}{4^{(res_i-1)}}\right\} \subseteq \left\{\frac{\mathrm{MAS}_i.LB}{4^{(res_i-1)}}, \frac{\mathrm{MAS}_i.RT}{4^{(res_i-1)}}\right\}$, 这是一个恒等关系, 所以 $\mathrm{MAS}_i \preccurlyeq \mathrm{MAS}_i$ 成立, 因此$\preccurlyeq$具有自反性.

● 反对称性: 假设 $\mathrm{MAS}_i \preccurlyeq \mathrm{MAS}_j$ 且 $\mathrm{MAS}_j \preccurlyeq \mathrm{MAS}_i$, 则 $\mathrm{MAS}_i = \mathrm{MAS}_j$. 根据定义, 这意味着 $(\mathrm{MAS}_i.x_{LB} \geqslant \mathrm{MAS}_j.x_{LB}) \wedge (\mathrm{MAS}_j.x_{RT} \geqslant \mathrm{MAS}_i.x_{RT}) \wedge (\mathrm{MAS}_i.y_{LB} \geqslant \mathrm{MAS}_j.y_{LB}) \wedge (\mathrm{MAS}_j.y_{RT} \geqslant \mathrm{MAS}_i.y_{RT})$以及 $(\mathrm{MAS}_j.x_{LB} \geqslant \mathrm{MAS}_i.x_{LB}) \wedge (\mathrm{MAS}_i.x_{RT} \geqslant \mathrm{MAS}_j.x_{RT}) \wedge (\mathrm{MAS}_j.y_{LB} \geqslant \mathrm{MAS}_i.y_{LB}) \wedge (\mathrm{MAS}_i.y_{RT} \geqslant \mathrm{MAS}_j.y_{RT})$, 可得 $(\mathrm{MAS}_i.x_{LB} = \mathrm{MAS}_j.x_{LB})$ 且 $(\mathrm{MAS}_i.x_{RT} = \mathrm{MAS}_j.x_{RT})$ 且 $(\mathrm{MAS}_i.y_{LB} = \mathrm{MAS}_j.y_{LB})$ 且 $(\mathrm{MAS}_i.y_{RT} = \mathrm{MAS}_j.y_{RT})$. 即满足集合 $\left\{\frac{\mathrm{MAS}_i.LB}{4^{(res_i-1)}}, \frac{\mathrm{MAS}_i.RT}{4^{(res_i-1)}}\right\} \subseteq \left\{\frac{\mathrm{MAS}_j.LB}{4^{(res_j-1)}}, \frac{\mathrm{MAS}_j.RT}{4^{(res_j-1)}}\right\}$, 且满足集合 $\left\{\frac{\mathrm{MAS}_j.LB}{4^{(res_j-1)}}, \frac{\mathrm{MAS}_j.RT}{4^{(res_j-1)}}\right\} \subseteq \left\{\frac{\mathrm{MAS}_j.LB}{4^{(res_i-1)}}, \frac{\mathrm{MAS}_j.RT}{4^{(res_i-1)}}\right\}$, 得 $\left\{\frac{\mathrm{MAS}_i.LB}{4^{(res_i-1)}}, \frac{\mathrm{MAS}_i.RT}{4^{(res_i-1)}}\right\} = \left\{\frac{\mathrm{MAS}_j.LB}{4^{(res_j-1)}}, \frac{\mathrm{MAS}_j.RT}{4^{(res_j-1)}}\right\}$, 则 $\mathrm{MAS}_i = \mathrm{MAS}_j$, 因此$\preccurlyeq$具有反对称性.

● 传递性: 假设 $\mathrm{MAS}_i \preccurlyeq \mathrm{MAS}_j$ 且 $\mathrm{MAS}_j \preccurlyeq \mathrm{MAS}_k$, 则 $\mathrm{MAS}_i \preccurlyeq \mathrm{MAS}_k$. 根据条件, $\mathrm{MAS}_i \subseteq \mathrm{MAS}_j$ 意味着 $(\mathrm{MAS}_i.x_{LB} \geqslant \mathrm{MAS}_j.x_{LB}) \wedge (\mathrm{MAS}_j.x_{RT} \geqslant \mathrm{MAS}_i.x_{RT}) \wedge (\mathrm{MAS}_i.y_{LB} \geqslant \mathrm{MAS}_j.y_{LB}) \wedge (\mathrm{MAS}_j.y_{RT} \geqslant \mathrm{MAS}_i.y_{RT})$, 即满足 $\left\{\frac{\mathrm{MAS}_i.LB}{4^{(res_i-1)}}, \frac{\mathrm{MAS}_i.RT}{4^{(res_i-1)}}\right\} \subseteq \left\{\frac{\mathrm{MAS}_j.LB}{4^{(res_j-1)}}, \frac{\mathrm{MAS}_j.RT}{4^{(res_j-1)}}\right\}$, $\mathrm{MAS}_j \preccurlyeq \mathrm{MAS}_k$ 意味着 $(\mathrm{MAS}_j.x_{LB} \geqslant \mathrm{MAS}_k.x_{LB}) \wedge (\mathrm{MAS}_k.x_{RT} \geqslant \mathrm{MAS}_j.x_{RT}) \wedge (\mathrm{MAS}_j.y_{LB} \geqslant \mathrm{MAS}_k.y_{LB}) \wedge (\mathrm{MAS}_k.y_{RT} \geqslant \mathrm{MAS}_j.y_{RT})$, 即满足 $\left\{\frac{\mathrm{MAS}_j.LB}{4^{(res_j-1)}}, \frac{\mathrm{MAS}_j.RT}{4^{(res_j-1)}}\right\} \subseteq \left\{\frac{\mathrm{MAS}_k.LB}{4^{(res_k-1)}}, \frac{\mathrm{MAS}_k.RT}{4^{(res_k-1)}}\right\}$. 由于 $\mathrm{MAS}_i.x_{LB} \geqslant \mathrm{MAS}_j.x_{LB}$ 且 $\mathrm{MAS}_j.x_{LB} \geqslant \mathrm{MAS}_k.x_{LB}$, 根据传递性, 有 $\mathrm{MAS}_i.x_{LB} \geqslant \mathrm{MAS}_k.x_{LB}$; 同理, $(\mathrm{MAS}_k.x_{RT} \geqslant \mathrm{MAS}_i.x_{RT}) \wedge (\mathrm{MAS}_i.y_{LB} \geqslant$

$\mathrm{MAS}_k.y_{LB}) \wedge (\mathrm{MAS}_k.y_{RT} \geqslant \mathrm{MAS}_i.y_{RT})$, 即满足 $\left\{\frac{\mathrm{MAS}_i.LB}{4^{(res_i-1)}}, \frac{\mathrm{MAS}_i.RT}{4^{(res_i-1)}}\right\} \subseteq \left\{\frac{\mathrm{MAS}_k.LB}{4^{(res_k-1)}}, \frac{\mathrm{MAS}_k.RT}{4^{(res_k-1)}}\right\}$, 则 $\mathrm{MAS}_i \preccurlyeq \mathrm{MAS}_k$, 因此$\preccurlyeq$具有传递性.

同理可得, 特例的自反性、反对称性、传递性也满足 $\mathrm{MAS}_i \preccurlyeq \mathrm{MAS}_j$, 证毕.

定理 2. MAS 相交定理. 设 γ 为用于表示轨迹的最小相邻子空间 (MAS) 的数据集合, 存在 $\mathrm{MAS}_i(LB_i, RT_i)$、$\mathrm{MAS}_j(LB_j, RT_j) \in \gamma$, 令 $\mathbf{z}_1, \mathbf{z}_2, \mathbf{z}_3, \mathbf{z}_4$ 分别为 LB_i, RT_i, LB_j, RT_j, 粒度对齐长度为:

$$Min_{res} = \min(\mathrm{MAS}_i.res, \mathrm{MAS}_j.res) \tag{4}$$

并定义截断编码为:

$$\mathbf{z}'_k = \frac{\mathbf{z}_k}{4^{(\mathrm{MAS}_q.res - Min_{res})}}, \text{ 其中, } q = \begin{cases} i, k \in \{1,2\} \\ j, k \in \{3,4\} \end{cases} \tag{5}$$

其中, $k \in \mathbb{N}^+$. 则 MAS_i 与 MAS_j 相交的充要条件为:

$$\exists m \in \{\mathbf{z}'_1, \mathbf{z}'_2\},\ n \in \{\mathbf{z}'_3, \mathbf{z}'_4\},\ m \cap n \neq \varnothing,$$

即截断编码集合 $\{\mathbf{z}'_1, \mathbf{z}'_2\}$ 和 $\{\mathbf{z}'_3, \mathbf{z}'_4\}$ 至少存在一个公共元素. 特别的, 存在 $UpR(\mathrm{MAS}_j.LB) = \mathrm{MAS}_j.RT$, 若满足 $m \cap \{n \cup Up(\mathbf{z}'_3) \cup Right(\mathbf{z}'_3)\} \neq \varnothing$, 也满足 $\mathrm{MAS}_i \preccurlyeq \mathrm{MAS}_j$.

证明: 以下将分别证明 MAS 相交定理的必要性和充分性.

● 必要性: 若 $\mathrm{MAS}_1(LB_1, RT_2)$ 和 $\mathrm{MAS}_2(LB_3, RT_4)$ 实际相交, 则存在至少一个空间点 p 同时属于两者的覆盖范围. 根据 Z 曲线编码的局部性, 点 p 的编码经粒度对齐转换后, 其高位部分 $\mathbf{z}'_{Min_{res}}$(即截断至最小粒度 Min_{res} 的编码) 必然同时属于 $\{\mathbf{z}'_1, \mathbf{z}'_2\}$ 和 $\{\mathbf{z}'_3, \mathbf{z}'_4\}$. 这是因为截断操作本质是将空间映射到更高粒度的划分块中, 而实际相交点 p 的存在表明其在粗粒度层级必然对应两个 MAS 的公共子块. 由于 $Up(\mathbf{z}'_3)$ 和 $Right(\mathbf{z}'_3)$ 扩大了交集的可能范围, 因此实际相交性必然蕴含截断编码集合存在公共元素.

● 充分性: 若截断编码集合 $\{\mathbf{z}'_1, \mathbf{z}'_2\}$ 和 $\{\mathbf{z}'_3, \mathbf{z}'_4\}$ 存在公共元素 z, 则表明两个 MAS 在 Min_{res}-粒度 下共享至少一个公共子块. 由于 Z 曲线编码的递归四分划分性质, 该子块可进一步细化为低位编码定义的具体区域, 而 $\mathbf{z}_1, \mathbf{z}_2, \mathbf{z}_3, \mathbf{z}_4$ 的完整编码是对该子块的扩展. 因此, 公共子块内必然存在未被截断的低位编码组合, 使得对应的更细粒度区域同时属于两 MAS 的覆盖范围. 因此, 截断编码集合的公共元素充分保证了原空间中的相交性. 同理可得, 特例的截断编码集合也满足该性质, 证毕.

定义 3. MAS 序分支. 对于一个 MAS 的集合 γ, 其偏序划分记作 $\Theta = \{M_1, M_2, \ldots, M_i, \ldots, M_n\}$, M_i 被称为 Θ 的一个 MAS 偏序分支 MPB (MAS partial-order branch), 集合 M_i 中的每个元素都满足"$\preccurlyeq$"的偏序关系, 每个 M_i 都是 Θ 偏序划分中的一个全序列分支, 且每个 MAS 元素仅属于一个 M_i.

本文基于序分支的结构提出了序分支的相交定理, 并且给出该定理的证明如下.

定理 3. 序分支相交定理. 对于每个序分支 $M_i = \{\mathrm{MAS}_1, \mathrm{MAS}_2, \ldots, \mathrm{MAS}_n\}$, 对于任意 MAS_q, 若存在 $\mathrm{MAS}_j \cap \mathrm{MAS}_q \neq \varnothing$, 则 M_i 中所有位于 MAS_j 前的子空间均与 MAS_q 相交, 即 $\mathrm{MAS}_1 \cap \mathrm{MAS}_q \neq \varnothing \wedge \mathrm{MAS}_2 \cap \mathrm{MAS}_q \neq \varnothing \wedge \ldots \wedge \mathrm{MAS}_{j-1} \cap \mathrm{MAS}_q \neq \varnothing$. 若存在 $\mathrm{MAS}_j \cap \mathrm{MAS}_q \neq \varnothing$, 则 M_i 中所有位于 MAS_j 后的子空间与 MAS_q 均不相交, 即 $\mathrm{MAS}_{j+1} \cap \mathrm{MAS}_q = \varnothing \wedge \mathrm{MAS}_{j+2} \cap \mathrm{MAS}_q = \varnothing \wedge \ldots \wedge \mathrm{MAS}_n \cap \mathrm{MAS}_q = \varnothing$.

证明: 根据定理 1 和定理 2, 若存在 $\mathrm{MAS}_j \cap \mathrm{MAS}_q \neq \varnothing$, $\forall j \geqslant k$, 有 $\mathrm{MAS}_j \preccurlyeq \mathrm{MAS}_k$, 则 $\mathrm{MAS}_j \subseteq \mathrm{MAS}_k$, $\left\{\frac{\mathrm{MAS}_j.LB}{4^{(res_j-1)}}, \frac{\mathrm{MAS}_j.RT}{4^{(res_j-1)}}\right\} \subseteq \left\{\frac{\mathrm{MAS}_k.LB}{4^{(res_k-1)}}, \frac{\mathrm{MAS}_k.RT}{4^{(res_k-1)}}\right\}$, MAS_j 和 MAS_k 的截断编码相同, 即 MAS_j 的坐标区间完全包含在 MAS_k 的坐标区间内. 由于 $\mathrm{MAS}_j \cap \mathrm{MAS}_q \neq \varnothing$, 存在公共点 $p \in \mathrm{MAS}_j \cap \mathrm{MAS}_q$, 因 $\mathrm{MAS}_j \subseteq \mathrm{MAS}_k$, 点 p 必然同时属于 MAS_k 和 MAS_q, 即 $\mathrm{MAS}_k \cap \mathrm{MAS}_q \neq \varnothing$. 同理可得, $\forall j < k$ 时, 若 $\mathrm{MAS}_j \cap \mathrm{MAS}_q = \varnothing$, 则 $\mathrm{MAS}_k \cap \mathrm{MAS}_q = \varnothing$. 证毕.

例 2: 图 5 可表示为由 4 个最小相邻子空间共同组成的 MAS 偏序分支, 编码分别是 $\mathrm{MAS}_1(0,1)$, $\mathrm{MAS}_2(4,5)$, $\mathrm{MAS}_3(17,20)$, $\mathrm{MAS}_4(69,80)$, 由于 $\mathrm{MAS}_1.feature = \{0,1\}$, 因此 4 个最小相邻子空间满足 $\left\{\frac{69}{4^{(4-1)}}, \frac{80}{4^{(4-1)}}\right\} = \left\{\frac{17}{4^{(3-1)}}, \frac{20}{4^{(3-1)}}\right\} = \left\{\frac{4}{4^{(2-1)}}, \frac{5}{4^{(2-1)}}\right\} = \{1,1\} \subseteq \mathrm{MAS}_1.feature$, 符合偏序关系定义, 且截断编码相交不为空, 而这种关系也使得其

在平面上的形状呈现完全包含关系, 如图 5 所示, 其中 MAS_1 是由 0 和 1 组成的黄色区域, 分辨率为 1; MAS_2 是由 10 和 11 组成的蓝色区域, 分辨率为 2; MAS_3 是由 101 和 110 组成的粉色区域, 分辨率为 3; MAS_4 是由 1011 和 1100 组成的粉色区域, 分辨率为 4.

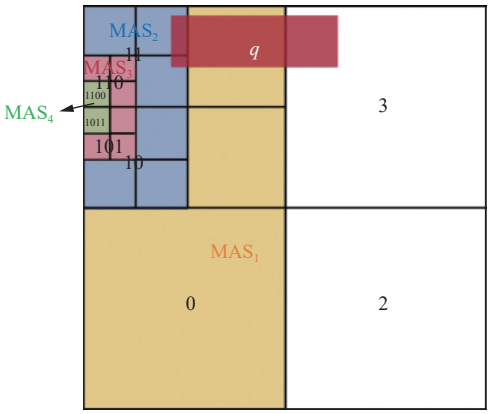

图 5 MAS 偏序分支相交示意图

定义 3 中的 MAS 序分支是依据定理 1 中定义的 MAS 偏序关系构建的, 用于组织轨迹 ZTE 编码的存储顺序. 即根据 ZTE 编码中的轨迹方向 *RightPos* 的值, 对整个数据集进行分组, 将 *RightPos* 值相同的轨迹划分到同一个组中, 随后对每一个分组内的轨迹段的 ZTE 编码构建多个符合 MAS 偏序关系的 MAS 序分支. 定理 2 所提出的 MAS 相交判定定理可用于高效判断轨迹的 MAS 是否与查询空间相交; 而定理 3 则进一步用于判定查询空间是否与 MAS 序分支相交, 以实现快速剪枝.

2.4.3 SPB 序列结构

为了提升轨迹数据的查询效率, 本文基于相邻子空间的偏序分支结构构建轨迹数据的索引结构 SPB 分支, 通过构建 Root-Max-Index 三级分层结构, SPB 将逻辑入口、空间范围元数据与实体轨迹数据进行解耦, 实现了查询过程中多层次渐进式过滤, SPB 分支的空间索引结构如图 6 所示.

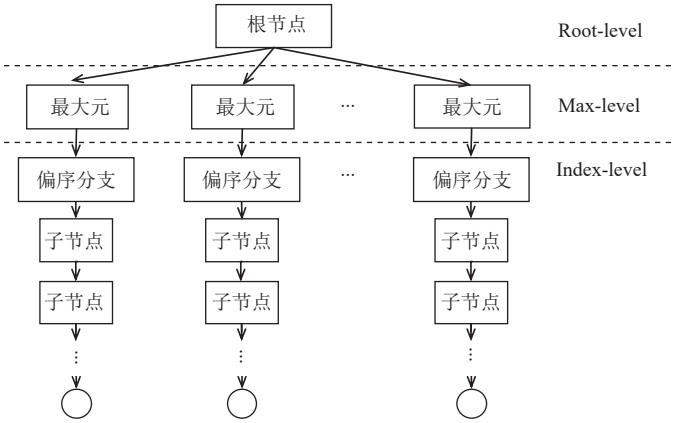

图 6 SPB 分支多叉索引结构示意图

(1) Root 层: 逻辑层, 表示数据操作的入口, 记录 SPB 分支的唯一标识 id, 当有数据操作时会通过 Root 层进入.

(2) Max 层: 维护 Index-level 分支内最大和最小子空间节点 ZTE 编码的边界信息, 在查询初始阶段即可运用

空间谓词 (相交、包含、覆盖等) 完成粗粒度剪枝: 当查询范围与分支最大子空间不存在包含或相交关系时, 可跳过该分支下所有 Index 节点的细粒度判断; 而当仅与最大子空间相交时, 则基于 ZTE 编码的偏序特性, 采用二分搜索算法在 O(log*N*) 时间复杂度内确定临界节点 (第 1 个不符合查询条件的节点), 查询对象为最大子空间节点至临界节点之间的所有节点.

(3) Index 层: 每个 Max-level 节点相对应的偏序分支节点构成, 且每个 Index 节点存放对应轨迹的移动对象等标识信息.

由于 R 树的兄弟节点的 MBR 存在着空间重叠的情况, 这种“死空间”在执行范围查询或 kNN 查询时会导致多路径遍历问题, 查询往往需要访问多个子树, 查询对象在磁盘上的位置分散造成磁盘访问的不连续性, 从而导致大量的磁盘随机 I/O (random I/O) 访问.

SPB 结构有效避免了这一问题. 其核心在于利用偏序关系构建序分支, 并通过多层级过滤机制进行剪枝, 具体机制如下.

(1) Max 层预过滤机制: 如果查询范围与每个序分支的最大 MAS 无包含或相交关系, 可直接跳过整个序分支中所有节点的判断. 如果包含最小 MAS 或者相交, 则需要访问分支中所有节点.

(2) 序分支内剪枝机制: 若查询范围与最大 MAS 存在相交, 则根据序分支相交定理进行剪枝, 遇到“临界节点”, 位于其后面的全部子节点可以被快速剪枝, 查询对象为从该分支的头节点至临界节点为止. 由于序分支中的节点是按 MAS 的偏序关系进行线性存储, 则查询对象的访问则是顺序 I/O (sequential I/O). 相比于随机 I/O 访问, 顺序 I/O 具有更好的磁盘访问效率.

SPB 分支范围查询算法见算法 1.

算法 1. SPB 分支范围查询算法.

输入: 空间范围 *spaceRange*, 根节点 *Root*;
输出: 符合时空范围的所有轨迹数据集合 *result.*

```
1. result ← NULL, temp_set = NULL
2. FOR each node in Root.MaxLevel DO
3.    min_zte ← node.min_node.zte // 最小子空间编码
4.    max_zte ← node.max_node.zte // 最大子空间编码
5.    branch ← node.IndexBranch // 对应的 Index 层分支
      // 阶段 1: 最大子空间与查询空间相交
6.    IF Intersect(max_zte, spaceRange) THEN
        // 剪枝条件 1: 查询空间是否与最小子空间相交
7.      IF Intersect(spaceRange, min_zte) THEN
          // 情况 1: 查询空间与最小子空间相交, 全量提取
8.        result.add(node.max_node.entries)
9.      ELSE
          // 情况 2: 二分查找找到首个不相交查询空间的临界节点
          // 临界节点 A (最大子节点, 即第 1 个子节点) 和 B (首个不相交的节点)
10.       A ← BinarySearch_FirstNode(branch)
11.       B ← BinarySearch_FirstDisjoint(branch, spaceRange)
          // 提取 A 剪掉 B 区域后的候选的分支节点
12.       result.add(A.entries–B.entries)
13. RETURN result
```

3 索引的构建和检索

3.1 索引的构建

SPBSpark 索引的构建流程如图 7 所示, 依照整体的构建过程可以分为数据划分、局部索引构建和全局索引构建这 3 部分. 在数据划分阶段要将数据进行分区, 随后将划分后的子分区分散到各个计算节点上以达到分布式计算的效果. 在分区时, 需要考虑负载均衡性以及可扩展性. 经过数据划分之后, 轨迹数据集可以看作是一个按照时间信息、路段信息划分好的点对象构成的 RDD, 称之为点 RDD. 再对此 RDD 经过分组、排序、分段等预处理操作后, 生成新的 RDD, 随后为每个分区中轨迹数据构建 SPB 分支作为局部索引用于时空范围查询和 kNN 查询, 并且在每个分区的头部增加一个局部哈希 IDMap, 将移动对象标识信息和轨迹片段映射起来, 用于移动对象轨迹查询, 并打包成新的 RDD, 称之为局部索引 RDD. 最后主节点收集每个分区的标识信息构建、路段信息、时间信息、分区中轨迹条数等信息构建多层混合全局索引.

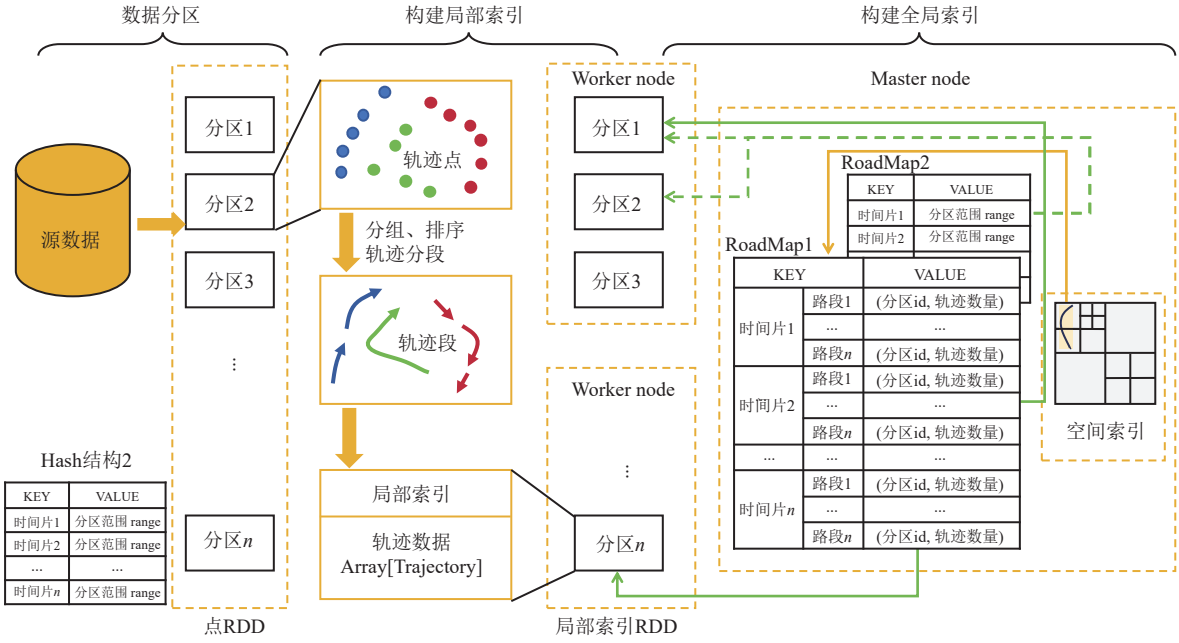

图 7 SPB 分支索引构建示意图

SPB 的构建算法如算法 2 所示, 该算法的平均时间复杂度为 $O(N\log N)$.

算法 2. SPB 的构建算法.

输入: 路段轨迹的数据集合 T;

输出: SPB 分支索引的根节点 *Root*.

1. $Root \leftarrow$ NULL, $resolution \leftarrow 1$
2. **FOR** $Traj \in T$ **DO**
3. S_l, S_r 分别为 *Traj* 轨迹数据矩形左下点坐标所在的子空间和右上点坐标所在的子空间
   // 遍历集合 T, 通过编码技术生成轨迹 ZTE 编码
4. **WHILE** $(S_l=S_r)\wedge(S_l$ nearby $S_r)$ **DO**
   // 递归划分平面, 直至子空间不相邻或不相同, 并生成编码
5. $S_l, S_r \leftarrow (S_l, S_r \,\&\, (1 << (resolution-1))) >> (resolution-1)$

```
6.      IF resolution = n DO BREAK
7.    RightPos ← generateRightPos(S_l, S_r) // 根据最小相邻子空间的位置关系, 生成 RightPos
8.    ZTE ← (resolution, feature ← Decimal(MAS), RightPos)
9.  all_node ← NULL, resolution_map 初始化为按 res 分组的字典
10. FOR new_node in ZTE DO
11.   all_node.append(new_node)
12.   resolution_map[new_node.res].append(new_node)
      // 更新当前分辨率层级的空间索引
13.   update_index(resolution_map[new_node.res], new_node)
      // 候选父筛选: 使用截断编码判断包含关系
14.   parent_candidates ← NULL
      // 检查更高或相同分辨率层级
15.   FOR res IN range(new_node.res, MAX_RESOLUTION+1) DO
        // 使用空间索引快速查询包含 new_node 的候选父节点
16.     candidates ← query_index(resolution_map[res], new_node)
17.     parent_candidates.extend(candidates)
18.   IF parent_candidates THEN
        // 动态维护有序候选列表: res 升序 > 范围降序
19.     best_parent ← find_best_parent(parent_candidates)
        // 链插入
20.     IF best_parent.child THEN
21.       new_node.child ← best_parent.child // 原子节点降级
22.     best_parent.child ← new_node
23.   ELSE
24.     Root.append(new_node) // 新分支根
25. RETURN Root
```

每条路段上的轨迹数据集合 T, 有两个关键步骤: (1) 通过编码技术对每条轨迹递归划分平面生成轨迹 ZTE 编码 (算法 2 中的第 5 行); (2) 候选父节点的查询 (算法 2 中的第 18 行).

● 步骤 (1) 时间复杂度. 对于生成轨迹 ZTE 编码, 假设 $|T| = n$, $resolution = m$, 每条轨迹生成编码的最坏时间复杂度为 $\mathrm{O}(mn)$, 最优时间复杂度为 $\mathrm{O}(n)$, 平均时间复杂度为 $\mathrm{O}\left(\dfrac{n\sum_{i=1}^{m} i}{m}\right) = \mathrm{O}\left(\dfrac{(m+1)n}{2}\right)$.

● 步骤 (2) 时间复杂度. 假设将节点按分辨率层级分为 L 个组, 每个层级平均节点数为 $n' = \dfrac{N}{L}$. 通过空间索引对每个层级进行高效查询, 单次候选父节点查询的时间复杂度可优化至 $\mathrm{O}(\log(n'))$. 由于每个新节点需要遍历所有可能包含它的更高或相同分辨率层级, 总查询时间复杂度为 $\mathrm{O}(L\log(n'))$, 所有节点总时间复杂度为 $\mathrm{O}\left(NL\log\left(\dfrac{N}{L}\right)\right)$. 当 L 为常数时, 这一时间复杂度简化为 $n\log(n)$.

在考虑轨迹数量作为大数据集的情况下, 其中轨迹数代表分辨率且为有限的整数值, 因此算法 2 的时间复杂度可被量化为 $\mathrm{O}(n\log(n))$.

3.2 索引的查询

SPBSpark 中支持的移动对象轨迹查询、时空范围查询和 kNN 查询[18]这 3 种查询方法.

3.2.1 基于移动对象标识的轨迹查询

基于移动对象标识的轨迹查询是指给定移动对象的标识 O_{id} 和时间范围<t_s, t_e>, 返回该移动对象在指定时间内的运动轨迹. SPBSpark 会首先通过时间窗口计算出所属时间切片, 在全局索引阶段通过 RoadMap2 的映射快速找到满足时间范围的每一个数据分区, 在每一个数据分区中通过局部索引中的 IDMap 哈希索引将移动对象的标识快速定位到移动对象的轨迹, 如算法 3 所示.

算法 3. 移动对象标识的轨迹查询算法.

输入: 移动对象 O_{id}, 时间范围 *timeRange*, 索引 *SPBSpark*;
输出: 移动对象在该时间范围内的轨迹 *Tr*.

// 通过全局索引找出时间范围和时间窗口有交集的全部分区, 路段参数置空
1. *partitionIds* ← *SPBSpark.globalIndex.RoadMap*2(*timeRange*)
2. **FOR** each *partitionId* in *partitionIds* **DO**
// 通过 Spark 的 API 找出每个分区中的数据内容
3. *PartitionPruningRDD*(*partitionId*)
// 根据 IDMap 获取移动对象 O_{id} 的轨迹片段
4. *trajSeg* ← *getTraj*(O_{id})
// 全局索引中的时间剪枝是粗粒度的, 但凡和时间切片有交集的分区都会被找出, 因此需要在局部索引时再次对时间做细粒度过滤
5. **IF** *partitionId.timeRange* OVERLAPS *timeRange* **THEN**
// 截取符合时间范围的轨迹片段
6. *trajSeg* ← *trajSeg.mapValues*(*clip*(*timeRange*))
// 合并所有的轨迹片段形成轨迹 *Tr*
7. *Tr* ← *trajSeg.reduceByKey*(*merge*).*collect*()
8. **RETURN** *Tr*

全局索引部分的效率为 O(log*N*), 局部索引部分的效率为 O(1).

3.2.2 基于时空范围窗口的轨迹查询

SPBSpark 采用了 SPB 分支作为基于时空范围窗口的轨迹查询的局部索引结构. SPBSpark 首先将查询窗口 Q 转换为时间范围参数 *timeRange* 和空间范围参数 *spaceRange*, SPBSpark 允许只给出时间参数或空间参数, 生成纯时间区间查询或者纯空间范围查询. 接着, SPBSpark 会通过全局索引中的 R 树搜索和指定空间范围有交集的全部路段, 随后根据时间范围换算其对应的时间切片, 并在哈希索引 RoadMap1 中找到对应的全部分区. 对于每一个分区内, 将时空范围查询窗口转换为基于 Z 曲线的编码 ZTE, 调用 SPB 分支的时空范围查询算法, 搜索出和查询窗口有交集的轨迹段, 对满足约束的片段进一步进行精确化比对. 最后, 将满足时空范围的轨迹片段按移动对象唯一标识分组、排序、合并得到完整轨迹. 基于以上步骤, 本文设计了基于移动对象标识的轨迹查询算法 (算法 4).

算法 4. 时空范围窗口查询算法.

输入: 时间范围 *timeRange*, 空间范围 *spaceRange*, 索引 *SPBSpark*;
输出: 在该时空范围内的轨迹 *Tr*.

// 通过全局索引找出空间范围和路网有交集的全部分区
1. *roadId* ← *SPBSpark.globalIndex.RtreeSearch*(*spaceRange*) // 通过全局索引找出空间范围和路网有交集的全部路段
2. *partitionId* ← *SPBSpark.globalIndex.RoadMap*1(*timeRange*, *roadId*) // 通过全局索引的哈希结构找出时间范围和对应路段下的全部分区

3. **FOR** each *partitionId* in *partitionIds* **DO**
   // 通过 Spark 提供的 API 找出每个分区中的数据内容
4. *PartitionPruningRDD*(*partitionId*)
   // 截取时空范围内的轨迹片段
5. **FOR** each *SPBranch* **DO**
6. *trajSeg* ← *SPBranch.tsRangeSearch*(*timeRange*, *spaceRange*)
   // 实际轨迹并不在查询的时间和空间窗口中, 因此要对查询结果进行精确化比对
7. **FOR** each *trajSeg* **DO**
8. *accurateSet* ∈ *trajSeg.mapValues*(*sub*(*timeRange*, *spaceRange*))
   // 合并所有的轨迹片段形成轨迹 *Tr*
9. *Tr* ← *trajSeg.reduceByKey*(*merge*).*collect*()
10. **RETURN** *Tr*

3.2.3 基于给定点的 kNN 轨迹查询

基于给定点的 kNN 查询是指, 给定查询点 q、时间范围 *timeRange*, 找出在时间范围内与之最近的 k 条轨迹. 传统的 kNN 查询算法的主要思想是不断扩展以查询点为中心的空间范围, 直到找到最近的 k 条轨迹为止. 但不断扩展空间范围也就意味着要对全局索引和局部索引进行反复遍历才能确定扩展的空间范围, 这种做法的效率不高. 为了能够更加高效地实现 kNN 查询, 本文在全局索引阶段就可以确定扩展的空间范围. 具体做法是: 首先, 根据查询点 q 的位置, 在空间 R 树中找到 MBR 包含 q 的路段的集合 *RoadMBRs*, 如果找不到, 则以 q 为中心不断扩展形成空间范围 S, 直到可以在空间 R 树中找到 MBR 和此空间范围相交的路段集合 *RoadMBRs*; 根据这些路段的 MBR 离 q 的最大距离进行排序, 找到距离最小的路段 *Rm*, 即 *Rm* 满足条件: $\min_{\mathrm{MBR}_i \in RoadMBRs} d_{\max}(\mathrm{MBR}_i, q)$. 由于在生成全局索引时, 收集了每个时间片上每条路段上的轨迹数量. 计算出给定时间范围对应的时间片, 通过全局索引中的 RoadMap1 哈希表找到 (时间片+*Rm* 路段) 对应的数据分区, 以及该分区中此路段的轨迹数量. 因此, 通过 RoadMap1 可以先判断该分区中的路段 *Rm* 上轨迹数量是否达到 k 个, 若没达到 k 个, 则继续查找距离次短的路段, 直到累计的轨迹数量达到 k 个为止, 并记录距离最大的路段的 MBR 离 q 最远的点 *MaxPoint* ($X_{\max}$, $Y_{\max}$). 即倘若此时对以上时间片范围内的路段集合所在的分区发起查询, 得到的轨迹数量至少可以达到 k 条, 但是这 k 条轨迹并不一定是距离查询点 q 最近的 k 条轨迹. 因此, 这一步骤称之为“模糊查询”, 这一步查询仅通过查询全局索引完成, 确定下来至少包括 k 条轨迹的大致范围.

其次, 需要找到可能比模糊查询的结果中距离 q 更近的路段及轨迹. 如图 8 所示, 以查询点 q 为圆心, 以 q 到最远点 *MaxPoint* ($X_{\max}$, $Y_{\max}$) 的距离为半径 (为了方便作图和阅读, 图 8 中的点代表每条路段 MBR 中距离 q 的最远点), 得到一个空间范围, 并使用该空间范围的外接 MBR 和时间范围进行时空范围查询, 找到符合查询条件的全部轨迹.

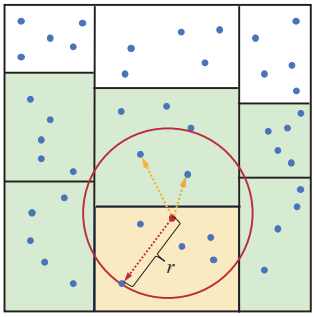

图 8 kNN 查询示意图

最后, 还需将搜集到的轨迹进行真实距离计算并排序, 返回距离查询点 q 最近的 k 条轨迹的结果候选集 *candidateTraj*.

基于以上描述, 本文给出基于给定点的 kNN 查询算法, 如算法 5 所示.

算法 5. 基于给定点的 kNN 轨迹查询算法.

输入: 查询点 q, 要返回的轨迹数目 k, 时间范围 *timeRange*, 索引 *SPBSpark*;
输出: 查询轨迹结果候选集 *candidateTraj*.

1. 初始化最大容量大于 k 的优先队列 *priQueue* 作为临时候选结果集
   // 根据空间 R 树找到与查询点 q 位置距离最近的路段, 并将路段加入 *AroundNode* 集合中
2. *Rm* ← *SPBSpark.globalIndex.RtreeSearch*(*q*)
3. *AroundNode* ← *Rm*
   // 判断路段 *Rm* 上轨迹数量是否达到 k 个, 若没达到 k 个, 则继续寻找距离 q 次近的路段
4. **WHILE** *SPB.globalIndex.RoadMap*1(*timeRange*, *AroundNode*).*count*() < *k* **DO**
5. *AroundNode.append*(*findNextAroundNode*(*Rm*))
   //依据离 $q(x, y)$ 最远点 *MaxPoint* (X_{max}, Y_{max}) 的 Haversine 距离 d 形成查询的空间范围
6. *spaceRange* ← (2*d*, 2*d*)
   // 通过全局索引找出空间范围和路网有交集的全部路段
7. *candidateRoad* ← *SPBSpark.globalIndex.RtreeSearch*(*spaceRange*)
   // 通过全局索引的哈希结构找出对应路段下的全部分区
8. *partitionId* ← *SPBSpark.globalIndex.RoadMap*1(*timeRange*, *candidateRoad*)
   // 通过 Spark 提供的 API 找出每个分区中的数据内容
9. *SPBSpark.localIndexRDD.PartitionPruningRDD*(*partitionId*)
10. **FOR** each *partition* **DO**
    // 发起局部时空范围查询, 将符合条件的轨迹加入 *priQueue*
11. *priQueue.collect*(*SPBranch.tsRangeSearch*(*timeRange*, *spaceRange*))
12. **WHILE** *priQueue.size*() < *k* **DO**
13. 调整 α 值, α 为扩大因子
14. *priQueue.collect*(*SPBSpark.tsRangeSearch*($\alpha \times$*spaceRange*, *timeRange*))
    // 利用 Spark 提供的 API 找到前 k 条数据
15. *IndexRDD* ← *priQueue*
16. *candidateTraj* ← *IndexRDD.takeOrdered*(*k*)
17. **RETURN** *candidateTraj*

3.3 索引的更新

在轨迹分析或历史数据处理系统中, 数据操作通常具备“一次写入、长期查询”的特征. 系统主要面临持续不断的新轨迹数据写入 (即插入操作或者追加更新操作) 以及高频次的查询请求. 此类应用中, 轨迹一旦生成即固定, 以“数据归档”的形式进行保存, 不会再修改单个轨迹的时空间信息. 同时, 删除操作主要用于周期性的历史数据清理 (如“仅保留近一个月的轨迹数据”), 一般通过批量移除时间分区实现, 而非针对单个轨迹的删除. 因此, 在此类场景下, 索引结构的设计核心在于高效支持轨迹数据的持续追加插入与快速查询响应, 所以本索引的更新算法就是特指数据的追加更新.

分布式轨迹索引 SPBSpark 中, 针对轨迹数据的更新不仅需要考虑局部索引的更新, 还需要考虑数据分区和全局索引的部分, 因此 SPBSpark 的轨迹索引更新算法如算法 6 所示, 在该算法中, 首先要对新增的轨迹数据进行轨

迹预处理, 其中包括轨迹地图匹配、轨迹分段, 得到基于路网的轨迹段集合. 接着, 根据新增轨迹段的时间参数换算得出其所属的时间分片, 同时调用 STR 空间分区算法将轨迹数据插入对应的数据分区集合中, 若插入新数据后该分区的轨迹数据量超过了阈值, 则应该进一步地切割该分区. 随后对于数据产生变化的全部分区, 应分为两种情况: (1) 针对插入新数据后没有超出阈值的分区, 调用局部索引 SPB 分支的更新算法, 对这些分区内的 SPB 分支进行更新. (2) 对于超出阈值重新生成的分区, 调用 SPB 分支的构建算法, 构建新分区的 SPB 分支. 最后, 针对被更新的分区及局部索引, 更新全局索引, 包括 RoadMap1 和 RoadMap2 两张哈希表中轨迹的数量以及新增分区的变化情况.

算法 6. SPBSpark 轨迹索引的更新算法.

输入: 轨迹段集合 *trajSet*, 更新前的 *SPBSpark*;
输出: 更新后的 *SPBSpark*.

1. *trajSeg* ← *trajSegment*(*trajSet*)
2. **FOR** each *trajSeg* ∈ *trajSet* **DO**
   // 更新分区
3. *node* ← 插入轨迹数据到对应的数据分区结点
4. **IF** *length*(*node.trajSet* > threshold) **THEN**
   // 切割一个新的分区
5. *splitNode*(*node*)
   // 对所有需要更新的分区更新局部索引
6. **FOR** *node* ∈ *updateNodes* **DO**
7. **IF** *node* ∉ *SPBSpark.partition* **THEN**
8. *localIndex* ← 构建 SPB 分支索引
9. **ELSE**
10. *localIndex* ← 更新 SPB 分支索引
11. // 更新全局索引的哈希表条目
12. **FOR** each *node* **DO**
    // 如果没有新增分区, 则更新全局索引哈希表中相应条目的轨迹数量记录, 如果新增分区, 则在哈希表中增加映射条目
13. *SPBSpark.globalIndex.RoadMap*1.*update*()
14. *SPBSpark.globalIndex.RoadMap*2.*update*()
15. **RETURN** *SPBSpark*

4 实验分析

本文实验的硬件环境是由阿里云的 ECS 虚拟计算节点 ecs.r7.xLBarge 搭建的 6 节点分布式集群环境, 每个节点配置有 4 核的 CPU 和 32 GB 内存. 实验的软件环境采用了 CentOS7.6 版本的 64 位操作系统. 为验证 SPBSpark 的性能, 本文采用的真实数据集[32]为 2013 年 10 月 1 日–31 日期间北京 13 007 辆出租车生成的真实数据集, 采样率约每分钟 1 个点, 包含 7 122 320 条轨迹, 每条轨迹长度大于 3 km, 大小为 186 GB, 并在 OpenStreetMap[33]中选取对应的北京市路网地图用于数据路网匹配.

在实验对象的选取方面, 选取了基于 Spark 平台构建的索引体系. 此外还需要同时能够支持时空范围查询、kNN 查询和移动对象轨迹查询, 并支持索引的追加更新. 对于历史轨迹数据的查询, Dragoon 和 LocationSpark 二者拥有相似的查询效率. 但 Dragoon 将更多精力投入到实时轨迹数据研究方面, 和本文的研究方向并不一致, 因此

本文最终选取了 TrajSpark 和 LocationSpark 作为实验对比的对象. TrajSpark 在全局索引结构中使用了 3 层混合索引结构, 包含粗粒度的时间索引、四叉树结构的空间索引以及一棵 B+树, 用于快速定位符合查询条件的分区. 在局部索引部分, TrajSpark 为每个分区头部增加了一个哈希结构用于映射移动对象和其对应的轨迹. LocationSpark 在全局索引处使用了空间 R 树, 并且 R 树的每一个叶子节点都对应一个分区, 并为每个分区构建一棵 R 树作为局部索引. 不仅如此, LocationSpark 还在每个节点中维护了一个空间位图过滤器 sFilter 来降低网络通信成本.

4.1 索引的构建实验对比

SPBSpark 索引构建的开销主要包括主节点中全局索引 R 树的构建和从节点局部索引 SPB 分支的构建. 图 9 显示了构建索引的时间随着数据量大小的变化情况, 实验选取随机生成的 50 组数据量一致的数据集. 由图 9 可得知, 随着数据量的增加, 3 个系统的空间开销都呈现线性上升的趋势, 但 SPBSpark 的索引构建的空间开销介于两者之间, 相较于 LocationSpark 实现了约 15% 的提升. 这主要是因为 LocationSpark 在全局和局部索引上采用了节点填充方式的 R 树, 其有效数据只存储于 N 个叶子节点当中, 而非叶子节点需要存储大量的中间无效数据, 而 SPBSpark 的空间开销主要在从节点的局部索引 SPB 分支当中, 其数据的存储方式相对紧凑, 并且每个节点都存储了有效数据, 因此 SPBSpark 的空间开销要优于 LocationSpark, 但 TrajSpark 虽然其在全局索引采用了 3 层索引结构, 但在局部索引阶段 TrajSpark 只构建了一个结构相对简单的哈希表用于映射移动对象和轨迹的关系, 因此在构建索引的时间开销方面会优于 SPBSpark.

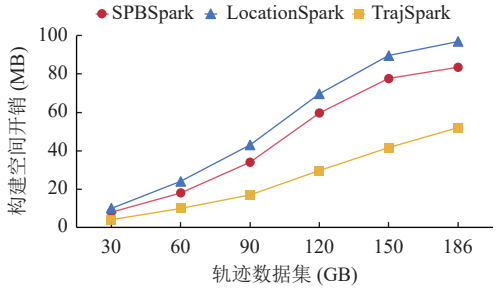

图 9 索引构建的空间开销对比图

4.2 索引的查询实验对比

4.2.1 时空范围查询实验对比

本节实验主要是针对时空范围查询算法的性能分析, 研究在不同数据量规模和不同查询窗口大小下的查询效率变化. 首先进行的是数据量变化对查询性能影响的实验, 其中时空范围查询的窗口取最大查询范围的 10%, 实验数据量从 30 GB 增长到 186 GB. 如图 10 所示, 随着轨迹数据量的增长, 这 3 个系统的查询时间开销虽有所增加, 但增加幅度并不大. 这是由于 3 个系统都是基于 Spark 架构而构建的分布式索引, 因而在内存充足的情况下, 查询时间开销不会大幅度地增长.

接下来分析查询窗口尺寸对查询时间开销的影响, 实验条件为相同数据集规模下, 查询窗口的空间和时间分别占其最大查询窗口的 10%–60%, 每个条件随机生成 500 组时空查询窗口进行对比查询实验, 并取平均值作为实验结果. 实验结果如图 11 所示, 随着时空窗口的不断扩大, 三者在查询时间上的消耗变化并不大, 但 SPBSpark 仍然比 TrajSpark 和 LocationSpark 的性能略佳. 这是由于 TrajSpark 并没有为时空范围查询在从节点上设置局部索引, 因此在经过全局索引获得可能与查询窗口有交集的分区之后, 系统需要对每一个分区上的轨迹数据全部扫描, 因此时间开销相对较大. 而 LocationSpark 和 SPBSpark 虽然都为时空范围查询构建了局部索引结构, 但是 LocationSpark 所构建的 R 树只把有效数据存储在叶子节点, 查询窗口越大, 其包含的子区域就越多, 需要递归遍历的次数也逐渐增加, 这种迭代遍历节点的查询方式的性能就会随着窗口的变大而显著降低. 但 SPBSpark 所采用的 SPB 树基于子空间编码的偏序关系进行构建, 根据其中的序分支结构剪枝定理可以快速剪枝掉与查询窗口不

相关的大量分支. 此外, 其检索过程中的时间花费主要是在窗口边界处的查询, 作为窗口内部的路段会直接输出其轨迹, 不需要经过复杂的索引, 因此窗口面积变化对 SPBSpark 的影响很小.

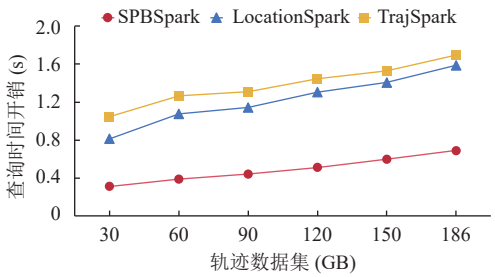

图 10 数据量变化对时空范围查询开销的影响对比图

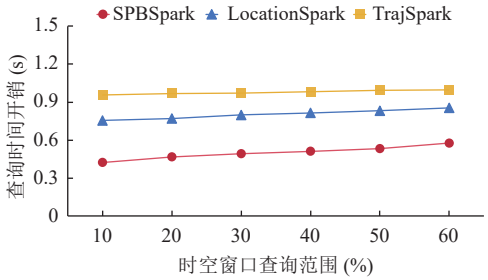

图 11 窗口大小变化对时空范围查询开销的影响对比图

如果在构建查询窗口时只输入空间范围参数就构成了空间范围查询. 如图 12 所示是纯空间范围查询的实验结果. SPBSpark 得益于在全局索引设置了对路网的 R 树索引, 可以先通过 R 树剪枝掉空间范围不符合的分区, 而且对于空间查询窗口内部的分区, 可以直接返回全部轨迹, 对于窗口边界处, 也可以通过遍历整棵 SPB 分支快速地剪枝掉不符合空间查询条件的分支. 而 TrajSpark 和 LocationSpark 在全局索引结构中有空间索引结构, 因此可以快速定位符合查询条件的分区, 但 TrajSpark 在分区内部并没有空间索引结构, 因此需要对数据全部扫描, 但 LocationSpark 在分区内构建 R 树作为空间索引, 因此查询效率略高于 TrajSpark.

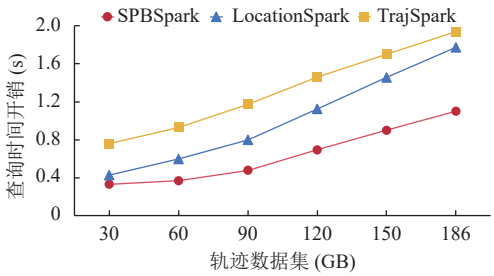

图 12 数据量变化对纯空间范围查询开销的影响对比图

4.2.2 kNN 实验对比

最近邻查询的实验主要分为两组, 分别是固定 k 值不变, 查看数据量变化对系统性能的影响; 以及固定数据量不变, 查看 k 值从 1 变化到 20 对系统性能的影响. 随机选取 100 组不同位置的查询点进行实验, 取查询时间的平均值作为实验结果. 从后文图 13 中可以看出, 随着数据量的增长, 3 个系统的时间开销增加的幅度都不大, 这是由于三者都是基于 Spark 的分布式架构, 数据量的变化只会影响节点参与计算的数量, 但由于 TrajSpark 在全局索引是对时间的粗粒度剪枝, 并且在局部索引阶段需要对轨迹数据全部扫描, 性能相对较差. LocationSpark 虽然能够使用全局索引和 sFilter 快速定位符合查询条件的分区, 但是在局部索引阶段仍需遍历整个 R 树. SPBSpark 优于其他两个系统的原因是其在全局索引时提供了两个剪枝边界, 因此只需要对空间 R 树遍历一次, 并且按时间片的细粒度划分方法能够有效对不符合查询条件的结果进行精确剪枝, 因此查询效率相对更高.

随后固定数据集的数据量不变, 将 k 值从 1 递增到 20, 时空查询点的选取和固定 k 值的情况相同, 同样取查询时间的平均值作为实验结果, 如后文图 14 所示, 随着 k 值逐渐增加, 查询时间开销略有增加. 这是因为随着 k 值的递增, 三者都需要遍历更多的子节点和分区, 需要更加频繁地与从节点进行通信, 导致 I/O 开销增加.

4.2.3 移动对象轨迹查询实验对比

本节在移动对象的轨迹查询方面对 3 个系统进行对比, 实验随机选取 100 个移动对象, 固定时间区间为最大

时间查询范围的 10%, 取查询时间的平均值作为实验结果. 如图 15 所示, SPBSpark 的查询效率和 TrajSpark 接近, 但略优于 TrajSpark. 这是由于 SPBSpark 和 TrajSpark 都在局部索引设计了映射移动对象 ID 和轨迹 ID 关系的哈希表, 但 SPBSpark 在全局索引阶段设计了更细粒度的时间切片的索引, 能够快速过滤掉不符合查询条件的全部分区, 相比之下 TrajSpark 在全局索引阶段只提供了粗粒度的时间索引, 全局获取的分区数量要多于 SPBSpark, 从而参与运算的从节点数量较多. 而 LocationSpark 并未对移动对象的轨迹查询专门设计索引, 因此在该查询中需要频繁遍历 R 树, 因此查询的时间开销相对较大.

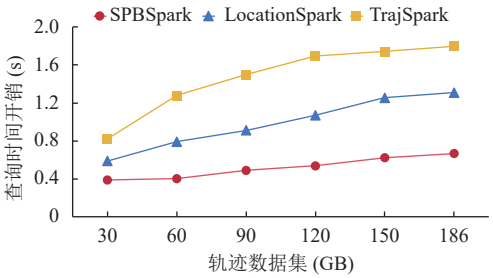

图 13 数据量变化对 kNN 查询开销的影响对比图

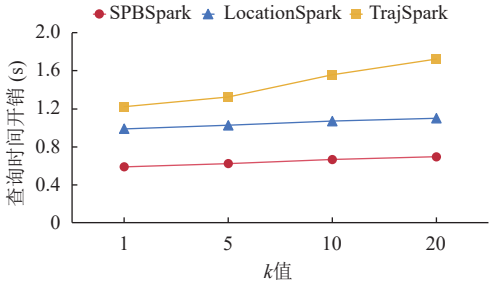

图 14 k 值变化对 kNN 查询开销的影响对比图

4.3 索引的更新实验对比

索引的更新主要分为追加更新、修改和删除这 3 类, 但由于 SPBSpark、TrajSpark 和 LocationSpark 都主要应用于历史轨迹数据的索引领域, 因此索引的更新只考虑追加更新. 本节专注于对比 SPBSpark、TrajSpark 和 LocationSpark 在索引追加更新方面的时间开销. 实验轨迹数据量从 30 GB 递增到 186 GB, 随机进行 100 次更新实验, 取实验结果的平均值进行研究. 如图 16 所示, 在索引追加更新的时间开销方面, 一开始 SPBSpark 的时间开销高于 TrajSpark 和 LocationSpark, 这是由于 SPBSpark 的索引结构比较复杂, 在索引更新时, 需要更新数据分区、局部索引和全局索引结构. 但随着数据量的不断增加, SPBSpark 的更新效率优于 LocationSpark, 这是因为 R 树会多次分裂节点以保持其平衡性, 而 SPB 分支只需要将新增的节点插入对应的位置上, 并不需要分裂节点来保持树的平衡性, 并且随着数据量的增加, SPB 分支的节点发生重复的概率提升, 因此不需要频繁地插入和更新 SPB 分支结构. 但 TrajSpark 索引的结构比较简单, 并且在局部索引中只需要更新哈希表结构, 因此更新时间开销较短.

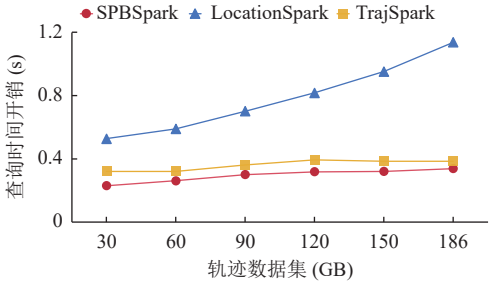

图 15 数据量变化对轨迹查询开销的影响对比图

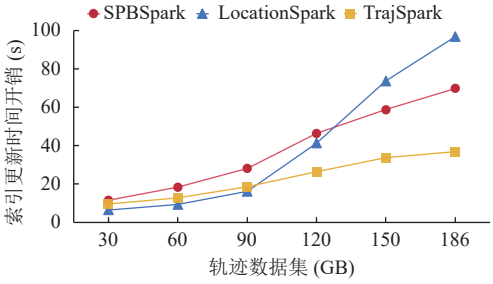

图 16 索引更新时间开销对比图

5 总结与展望

随着智能移动终端以及 5G 互联网技术的广泛普及, 基于位置的应用程序和服务已经完全充斥到人们的日常生活当中, 随之而来的是各行各业无时无刻不在产生的海量轨迹数据. 在此背景下, 本文研究成果和主要工作如下.

(1) 本文提出了一种适合表示轨迹数据的编码技术 ZTE, 该技术选取了 Z 曲线在不同递归分辨率下恰好完全

包含 MBR 的最小相邻子空间 MAS 的编码来表示轨迹, 并使用 MBR 右上点所在子空间相较于 MBR 左下点所在子空间的 3 种位置关系编码 (正上、右上、正右) 来记录轨迹的运动方向.

(2) 针对当前分布式索引支持的查询方式单一以及单机式索引无法处理海量轨迹数据的问题, 设计了基于 Spark 的分布式轨迹索引 SPBSpark. 本文结合了 Spark 大数据平台, 在 SPB 分支的基础上进一步提出了支持 kNN 查询、时空范围查询和移动对象轨迹查询这 3 种查询方式的分布式轨迹索引 SPBSpark. 本文还给出了 SPBSpark 的构建、查询和更新算法, 最终实验表明, 相比于选取了 R 树或哈希结构作为索引方案的 TrajSpark 和 LocationSpark, SPBSpark 在 kNN 查询、轨迹查询和时空范围查询方面拥有 2–3 倍的性能提升.

同时, 本文也通过大量实验和分析发现了一些有待思考和解决的问题. 首先, 在轨迹编码方案中, 本文提出了如何将二维的轨迹数据矩形 MBR 降至一维并生成对应的编码. 未来, 可探索利用多维曲线, 将时间与三维空间的信息进行融合编码. 其次, 本文支持了移动对象轨迹查询, kNN 轨迹查询和时空范围查询. 未来应继续考虑如何利用 SPBSpark 支持更多类型的查询, 如轨迹相似性查询. 最后, 本文的研究工作是基于历史数据展开的, 未来可以考虑支持实时轨迹查询的需求.

References

[1] Huang HS, Gartner G, Krisp JM, Raubal M, van de Weghe N. Location based services: Ongoing evolution and research agenda. Journal of Location Based Services, 2018, 12(2): 63–93. [doi: 10.1080/17489725.2018.1508763]

[2] Alam MM, Torgo L, Bifet A. A survey on spatio-temporal data analytics systems. ACM Computing Surveys, 2022, 54(10s): 219. [doi: 10.1145/3507904]

[3] Gao Q, Zhang FL, Wang RJ, Zhou F. Trajectory big data: A review of key technologies in data processing. Ruan Jian Xue Bao/Journal of Software, 2017, 28(4): 959–992 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5143.htm [doi: 10.13328/j.cnki.jos.005143]

[4] Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin MJ, Shenker S, Stoica I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proc. of the 9th USENIX Symp. on Networked Systems Design and Implementation. San Jose: USENIX Association, 2012. 15–28.

[5] Sidlauskas D, Chester S, Zacharatou ET, Ailamaki A. Improving spatial data processing by clipping minimum bounding boxes. In: Proc. of the 34th IEEE Int'l Conf. on Data Engineering (ICDE). Paris: IEEE, 2018. 425–436. [doi: 10.1109/ICDE.2018.00046]

[6] Morton GM. A computer oriented geodetic data base and a new technique in file sequencing. Technical Report, Ottawa: IBM Ltd, 1966.

[7] Yu J, Wu JX, Sarwat M. GeoSpark: A cluster computing framework for processing large-scale spatial data. In: Proc. of the 23rd SIGSPATIAL Int'l Conf. on Advances in Geographic Information Systems. Seattle: ACM, 2015. 70. [doi: 10.1145/2820783.2820860]

[8] Xie D, Li FF, Yao B, Li GF, Zhou L, Guo MY. Simba: Efficient in-memory spatial analytics. In: Proc. of the 2016 Int'l Conf. on Management of Data. San Francisco: ACM, 2016. 1071–1085. [doi: 10.1145/2882903.2915237]

[9] Magellan. 2021. https://github.com/harsha2010/magellan

[10] SpatialSpark. 2015. https://github.com/syoummer/SpatialSpark

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

[12] 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]

[13] Finkel RA, Bentley JL. Quad trees a data structure for retrieval on composite keys. Acta Informatica, 1974, 4(1): 1–9. [doi: 10.1007/BF00288933]

[14] Wang Y, Gui ZP, Wu HY, Peng DH, Wu JH, Cui ZS. Optimizing and accelerating space-time Ripley's K function based on Apache Spark for distributed spatiotemporal point pattern analysis. Future Generation Computer Systems, 2020, 105: 96–118. [doi: 10.1016/j.future.2019.11.036]

[15] Tang MJ, Yu YY, Mahmood AR, Malluhi QM, Ouzzani M, Aref WG. LocationSpark: In-memory distributed spatial query processing and optimization. Frontiers in Big Data, 2020, 3: 30. [doi: 10.3389/fdata.2020.00030]

[16] Shang ZY, Li GL, Bao ZF. DITA: Distributed in-memory trajectory analytics. In: Proc. of the 2018 Int'l Conf. on Management of Data. Houston: ACM, 2018. 725–740. [doi: 10.1145/3183713.3183743]

[17] Eldawy A, Hristidis V, Ghosh S, Saeedan M, Sevim A, Siddique AB, Singla S, Sivaram G, Vu T, Zhang YM. Beast: Scalable exploratory

analytics on spatio-temporal data. In: Proc. of the 30th ACM Int'l Conf. on Information & Knowledge Management. New York: ACM, 2021. 3796–3807. [doi: 10.1145/3459637.3481897]

[18] He HJ, Xu ZH, Li RY, Bao J, Li TR, Zheng Y. TMan: A high-performance trajectory data management system based on key-value stores. In: Proc. of the 40th IEEE Int'l Conf. on Data Engineering (ICDE). Utrecht: IEEE, 2024. 4951–4964. [doi: 10.1109/ICDE60146.2024.00376]

[19] Fang ZQ, Chen L, Gao YJ, Pan L, Jensen CS. Dragoon: A hybrid and efficient big trajectory management system for offline and online analytics. The VLDB Journal, 2021, 30(2): 287–310. [doi: 10.1007/s00778-021-00652-x]

[20] Zhang ZG, Jin CQ, Mao JL, Yang XL, Zhou AY. TrajSpark: A scalable and efficient in-memory management system for big trajectory data. In: Proc. of the 1st Int'l Joint Conf. on Web and Big Data. Beijing: Springer, 2017. 11–26. [doi: 10.1007/978-3-319-63579-8_2]

[21] Lawder JK, King PJH. Querying multi-dimensional data indexed using the Hilbert space-filling curve. ACM SIGMOD Record, 2001, 30(1): 19–24. [doi: 10.1145/373626.373678]

[22] Sagan H. Hilbert's space-filling curve. In: Sagan H, ed. Space-Filling Curves. New York: Springer, 1994. 9–30. [doi: 10.1007/978-1-4612-0871-6_2]

[23] Hilbert D. Über die stetige Abbildung einer Linie auf ein Flächenstück. In: Dritter Band: Analysis · Grundlagen der Mathematik · Physik Verschiedenes. Berlin: Springer, 1935. 1–2. [doi: 10.1007/978-3-662-38452-7_1]

[24] Albuquerque N, Bernal-González L, Pellegrino D, Seoane-Sepúlveda JB. Peano curves on topological vector spaces. Linear Algebra and its Applications, 2014, 460: 81–96. [doi: 10.1016/j.laa.2014.07.029]

[25] Hughes JN, Annex A, Eichelberger CN, Fox A, Hulbert A, Ronquest M. GeoMesa: A distributed architecture for spatio-temporal fusion. In: Proc. of the 2015 SPIE 9473, Geospatial Informatics, Fusion, and Motion Video Analytics V. Baltimore: SPIE, 2015. 94730F. [doi: 10.1117/12.2177233]

[26] Li RY, He HJ, Wang RB, Ruan SJ, He TF, Bao J, Zhang JB, Hong L, Zheng Y. TrajMesa: A distributed NoSQL-based trajectory data management system. IEEE Trans. on Knowledge and Data Engineering, 2023, 35(1): 1013–1027. [doi: 10.1109/TKDE.2021.3079880]

[27] Liu H, Yan JN, Wang JL, Chen B, Chen M, Huang XH. HGST: A Hilbert-GeoSOT spatio-temporal meshing and coding method for efficient spatio-temporal range query on massive trajectory data. ISPRS Int'l Journal of Geo-information, 2023, 12(3): 113. [doi: 10.3390/ijgi12030113]

[28] Liu ZD, Zhou YQ, Liu XS, Zhang HD, Dong YB, Lu DM, Wu KS. Learning road network index structure for efficient map matching. IEEE Trans. on Knowledge and Data Engineering, 2025, 37(1): 423–437. [doi: 10.1109/TKDE.2024.3485195]

[29] Bao Y, Huang Z, Gong XR, Zhang YY, Yin GM, Wang H. Optimizing segmented trajectory data storage with HBase for improved spatio-temporal query efficiency. Int'l Journal of Digital Earth, 2023, 16(1): 1124–1143. [doi: 10.1080/17538947.2023.2192979]

[30] Giao BC, Anh DT. Improving Sort-Tile-Recusive algorithm for R-tree packing in indexing time series. In: Proc. of the 2015 IEEE RIVF Int'l Conf. on Computing & Communication Technologies—Research, Innovation, and Vision for Future (RIVF). Can Tho: IEEE, 2015. 117–122. [doi: 10.1109/RIVF.2015.7049885]

[31] Böhm C, Klump G, Kriegel HP. XZ-ordering: A space-filling curve for objects with spatial extension. In: Proc. of the 6th Int'l Symp. on Advances in Spatial Databases. Hong Kong: Springer, 1999. 75–90. [doi: 10.1007/3-540-48482-5_7]

[32] Liu HP, Jin CQ, Zhou AY. Popular route planning with travel cost estimation. In: Proc. of the 21st Int'l Conf. on Database Systems for Advanced Applications. Dallas: Springer, 2016. 403–418. [doi: 10.1007/978-3-319-32049-6_25]

[33] OpenStreetMap. 2010. https://www.openstreetmap.org/

附中文参考文献

[3] 高强, 张凤荔, 王瑞锦, 周帆. 轨迹大数据: 数据处理关键技术研究综述. 软件学报, 2017, 28(4): 959–992. http://www.jos.org.cn/1000-9825/5143.htm [doi: 10.13328/j.cnki.jos.005143]

作者简介

汤娜, 博士, 副教授, CCF 专业会员, 主要研究领域为时态数据库, 大数据管理, 移动对象数据库, 知识发现.

蔡锶淇, 硕士生, CCF 学生会员, 主要研究领域为数据处理与挖掘.

李晶晶, 博士, 副教授, 主要研究领域为计算智能与优化.

罗凯原, 硕士, 主要研究领域为数据处理与挖掘.

汤庸, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为时间信息和数据管理, 计算机支持协作, 信息计算和服务, 大数据管理技术.