计算机科学
COMPUTER SCIENCE

Truster:面向高效查询的自动驾驶轨迹数据聚类存储方案

王征权, 彭智勇

引用本文

王征权, 彭智勇. Truster:面向高效查询的自动驾驶轨迹数据聚类存储方案[J]. 计算机科学, 2025, 52(11): 62-70.

WANG Zhengquan, PENG Zhiyong. Truster:Efficient Query-oriented Clustered Storage Solution for Autonomous Vehicle TrajectoryData [J]. Computer Science, 2025, 52(11): 62-70.

相似文章推荐（请使用火狐或 IE 浏览器查看文章）

Similar articles recommended (Please use Firefox or IE to view the article)

基于细粒度缓存与学习型索引的LSM树键值存储系统性能优化

Performance Optimization of LSM-tree Based Key-Value Storage System Based on Fine-grained Cache andLearned Index

计算机科学, 2025, 52(2): 33-41. https://doi.org/10.11896/jsjkx.240200001

结合时空关键字的轨迹范围查询混合索引结构

Hybrid Index Structure for Trajectory Range Query Combined with Spatio-Temporal Keywords

计算机科学, 2024, 51(11A): 240200114-8. https://doi.org/10.11896/jsjkx.240200114

LayerLSB:基于分层局部敏感B树的最近邻搜索

LayerLSB:Nearest Neighbors Search Based on Layered Locality Sensitive B-tree

计算机科学, 2023, 50(4): 32-39. https://doi.org/10.11896/jsjkx.220600078

基于时空轨迹数据的异常检测

Anomaly Detection Based on Spatial-temporal Trajectory Data

计算机科学, 2021, 48(6A): 213-219. https://doi.org/10.11896/jsjkx.201100193

基于近似匹配的移动边缘计算缓存管理方法

Cache Management Method in Mobile Edge Computing Based on Approximate Matching

计算机科学, 2021, 48(1): 96-102. https://doi.org/10.11896/jsjkx.200800215

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.241100052

Truster:面向高效查询的自动驾驶轨迹数据聚类存储方案

王征权[1]　彭智勇[1,2]
1 武汉大学计算机学院　武汉 430000
2 武汉大学大数据研究院　武汉 430000
(zq_wang@whu.edu.cn)

摘　要　自动驾驶轨迹数据具有重要的研究与应用价值,其存储与查询技术得到了广泛的研究与关注。然而,现有轨迹数据管理方案主要针对一般轨迹数据设计,无法支持高频采样的自动驾驶轨迹数据的高效写入,且动态环境下高昂的索引维护开销使其难以满足动态更新与实时查询的需求。针对自动驾驶场景下高采样频率、高实时性的轨迹数据如何实现高频写入、动态更新、实时查询的问题,提出一种面向高效查询的自动驾驶轨迹数据聚类存储方案 Truster。该方法设计了编码器和嵌入器,为原始轨迹生成空间感知键并提取特征向量;设计了基于日志结构合并树的存储结构 CLSM 树,以实现相似轨迹的集中存储;设计了 LCC 合并策略,在有序字符串表进行合并的同时,利用基于局部敏感哈希的分桶方法进行快速聚类;设计了轨迹查询算法,利用多粒度缓存和分桶映射快速筛选搜索空间。Truster 不仅支持高频写入,还能适应动态工作负载的索引维护,且查询效率更高。在真实自动驾驶轨迹数据集 Argoverse 上进行的对比实验表明,与现有方法相比,Truster 在写入操作上取得了 20%～200%的加速,在查询操作上取得了 20%～100%的加速。

关键词:轨迹数据;轨迹存储;轨迹查询;日志结构合并树;局部敏感哈希
中图分类号　TP392

Truster:Efficient Query-oriented Clustered Storage Solution for Autonomous Vehicle Trajectory Data

WANG Zhengquan[1] and PENG Zhiyong[1,2]
1 School of Computer Science,Wuhan University,Wuhan 430000,China
2 Big Data Institute,Wuhan University,Wuhan 430000,China

Abstract　Autonomous vehicle trajectory data holds significant research and practical value,attracting extensive attention to its storage and querying technologies. However,existing trajectory data management solutions are primarily designed for general trajectory data and exhibit notable limitations in efficiently writing autonomous driving trajectory data with high sampling frequency. Additionally,the high cost of index maintenance in dynamic environments makes it challenging to meet the demands of dynamic updates and real-time queries. To address the challenges of achieving high-frequency writes,dynamic updates,and real-time queries for high-sampling-rate and high-real-time trajectory data in autonomous driving scenarios,this paper proposes Truster,an efficient query-oriented clustered storage solution for autonomous vehicle trajectory data. This method includes the design of an encoder and embedder to generate position-aware keys for raw trajectories and extract feature vectors;a storage structure based on a Log-Structured Merge tree(LSM-tree)called the CLSM-tree to achieve clustered storage of similar trajectories;an LCC compaction strategy that leverages locality-sensitive hashing(LSH)for rapid clustering during the compaction of Sorted-String Tables(SSTables);and a trajectory query algorithm that uses multi-granularity cache and bucket mapping to quickly narrow down the search space. Truster not only supports high-frequency writes but also maintains index adaptability to dynamic workloads while offering enhanced query efficiency. Comparative experiments on the real-world autonomous vehicle trajectory dataset Argoverse demonstrate that Truster achieves a 20% to 200% improvement in write performance and a 20% to 100% improvement in query performance compared to existing methods.

Keywords　Trajectory data,Trajectory storage,Trajectory query,LSM-tree,LSH

1　引言

自动驾驶技术的快速发展推动了边缘设备数量的飞速增长[1]。据分析,在未来几年,数十亿移动设备将产生以泽字节(ZB)为单位的轨迹数据[2]。存储和分析这些轨迹数据可以获得丰富的信息,用于路径规划和安全评估等[3]。例如,通过分析车辆在复杂交叉路口的轨迹数据,自动驾驶系统可以识别出潜在的危险区域和高风险行为,以此优化路径规划和

到稿日期:2024-11-07　返修日期:2025-02-16
通信作者:彭智勇(peng@whu.edu.cn)

碰撞检测算法。

区别于普通轨迹，自动驾驶轨迹数据具有3个关键特征。1)与普通轨迹数据相比，自动驾驶轨迹数据具有更高的采样频率。通常每秒收集数十个样本，例如自动驾驶轨迹数据集Argoverse[4]和NuScenes[5]的采样频率为10Hz。因此，轨迹写入操作必须足够灵活，避免阻塞其他操作。2)由于自动驾驶实时决策对时间非常敏感，因此轨迹查询需要在短时间内快速响应。3)自动驾驶轨迹数据是由移动设备或嵌入式设备收集的，这意味着CPU和存储资源受到限制，因此要保持尽量少的资源占用。

目前轨迹存储与索引技术得到广泛研究[6-13]，但它们因为一些原因不适用于自动驾驶的轨迹数据管理。1)在存储层面，一些轨迹管理方案[6-7]使用基于B树的存储引擎，而B树采用原地更新(In-place Update)策略，更新开销较大，不适用于高采样频率的自动驾驶轨迹数据。2)在索引层面，一些轨迹索引[7-11]在处理静态数据集时性能优秀，但难以应对动态变化的工作环境。例如，AsterixDB等[7-9]使用R树作为空间索引，而在面对高频更新的自动驾驶轨迹数据时，R树需要重新计算最小外接矩形，并执行节点合并或分裂操作以调整树结构，消耗大量时间。再例如，Torch[10]首先需要对轨迹进行地图匹配(Map Matching)，再基于此建立空间索引，而这个过程时间复杂度较高。3)在空间属性层面，空间属性作为轨迹数据的一项重要特征，在轨迹查询中，尤其是近邻查询和相似性查询中，能够用于快速过滤搜索空间，显著提升查询效率。而调研的所有轨迹管理方案[6-13]在存储轨迹时普遍忽略了轨迹的空间特性。例如，在分段存储方案[12]中，为了提升系统的可拓展性，采用存储与索引分离的设计架构，仅在其构建的空间索引中体现轨迹间的空间相关性。

针对自动驾驶轨迹数据的特征和现有轨迹数据管理方案的不足，本文设计并实现了一种更利于高效查询的自动驾驶轨迹数据聚类存储方案——Truster(Trajectory Clustered Storage)。Truster的核心设计思路如下：

1)设计了CLSM树(Cluster LSM-tree)作为底层存储结构。一方面，为了适应自动驾驶轨迹数据的高采样频率，CLSM树基于写入性能优秀的日志结构合并树(Log-structured Merge Tree，LSM-tree[14])进行改造，它保留了LSM树的异地更新(Out-of-place Update)、批量加载(Bulk Loading)的特性，充分发挥了磁盘高效顺序写的特性，提供卓越的写入性能。另一方面，为了充分利用轨迹的空间特性，该结构为轨迹生成空间感知键(Position-aware Key)，并在CLSM树内部按键的大小顺序排列数据，从而确保逻辑上相邻的轨迹在物理存储上也相邻。

2)设计了LCC(LSH-based Clustering while Compaction)合并策略。为了更充分地利用轨迹之间的相关性，在CLSM树的每次刷盘(Flush)和合并(Compaction)过程中，Truster会基于轨迹的特征向量进行一次基于局部敏感哈希(Locality-Sensitive Hashing，LSH[15-16])的快速聚类分桶，并将聚类结果作为固定单元持久化存储于有序字符串表(Sorted-String Table，SSTable[17])中。这种轨迹聚类存储的方式通过将相似的轨迹存储在一起，不仅有利于轨迹数据的压缩，还能通过快速过滤搜索空间，加速轨迹查询。

与现有轨迹数据管理方案相比，本文的核心贡献如下：

1)首次提出了针对自动驾驶场景下轨迹数据存储的系统解决方案，充分考虑了自动驾驶轨迹数据的高频更新和高实时性等特点。

2)创新性地利用轨迹的空间相关性优化轨迹存储，利用局部性原理加速轨迹查询。

3)结合LSM树的设计思想，提出了一种适用于动态工作负载的高效空间索引方法，能够在数据频繁更新的情况下实现更低的索引开销与更高的查询性能。

2 预备知识与相关工作

2.1 轨迹相关概念与定义

定义1(轨迹点) 一个轨迹点 $P=\{x,y,t,info\}$ 包含时间戳 t 及其在坐标系中的位置 (x,y)，以及其他属于该点的额外信息 $info$。

定义2(轨迹) 一条轨迹 $T=\{P_1,P_2,\cdots,P_n,info\}$ 是指一系列按照时间戳顺序排列的轨迹点的集合，以及其他属于该轨迹的额外信息 $info$。

定义3(范围查询) 给定一个轨迹集 $D=\{T_1,T_2,\cdots,T_n\}$ 以及一个查询窗口 $Q_r=\{x_1<x<x_2,y_1<y<y_2,t_1<t<t_2\}$，范围查询将返回 D_s，并且满足：$D_s\subseteq D$；$\forall T\in D_s$，$\exists P_i\in T,P_i\in Q_r$。

定义4(k 近邻查询(kNN)) 给定一个轨迹集 $D=\{T_1,T_2,\cdots,T_n\}$ 以及一个查询轨迹 T_q，k 近邻查询将返回 D_s，并且满足：$D_s\subseteq D$；$|D_s|=k$；$\forall T_i\in D_s$，$\forall T_j\in D-D_s$，$d(T_i,T_q)\leqslant d(T_j,T_q)$，其中 $d(\cdot,\cdot)$ 为计算两个轨迹之间距离的函数。若将 $d(\cdot,\cdot)$ 替换为计算两个轨迹相似度的函数，则近邻查询升级为相似度查询。

2.2 Z型曲线

Z型曲线(Z-order Curve)是一种空间填充曲线，它能够将多维数据映射为一维，同时保持数据点的局部性。例如，对于一个坐标点 (x,y)，通过交错其坐标值的二进制得到对应的 z 值。通过这种方式，具有相似 z 值的坐标点在地理空间上往往也相邻。

在执行范围查询时，Z型曲线通过将多维的查询窗口 $\{x_1<x<x_2,y_1<y<y_2,t_1<t<t_2\}$ 映射为一维 z 值范围 $(z_1,z_2)\cup(z_3,z_4)\cup\cdots\cup(z_{n-1},z_n)$，从而实现高效的范围查询。

2.3 LSM树

LSM树因其优秀的写性能，被广泛应用于现代高性能数据库存储系统中，如LevelDB，BigTable[17]和RocksDB等。LSM树的核心思想是采用异地更新(Out-of-Place Update)和仅追加(Append-Only)策略，将所有操作(包括插入、更新和删除)都转化为顺序写入，而不是直接覆盖旧记录。相比随机I/O，这种方式充分发挥了磁盘顺序I/O的优势，显著提高了写入性能。

图1展示了LSM树的基本架构，包括一个内存组件 L_0 和一系列磁盘组件 L_1-L_n。LSM树在 L_0 层的跳表(Skip-List)中积累并排序持续写入数据，这一组件被称为内存表(MemTable)。当内存表达到容量上限时，它会转化为不可变

内存表，并触发刷盘（Flush）操作，将其转换为 L_1 层的 SSTable。随着 L_1 层中 SSTable 的数量的不断增加并达到容量上限，系统会触发合并（Compaction）操作，将这些 SSTable 合并到 L_2 层。该过程持续进行，直至到达 L_n 层。

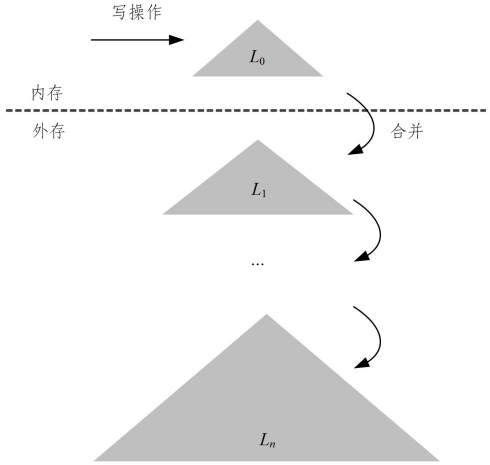


图 1 LSM 树结构图

Fig. 1 Structure of LSM-tree

由于采用异地更新策略，LSM 树中可能会同时存储同一数据的多个版本。因此，在 LSM 树上进行查询时，通常需要遍历多个层级的组件。此外，随着 SSTable 的逐渐累积，查询时需要检索的 SSTable 数量也不断增加，导致查询性能不断下降。因此，提高 LSM 树的查询性能成为一个研究热点。一些研究分别将 B+树、R 树和 RUM 树 LSM 化（LSM-ify），提出了 bLSM[18]，LSM R 树[7]和 LSM RUM 树[8]，以应对高频更新的数据。然而，二级索引的空间占用问题不可忽视，在最坏的情况下，二级索引大小可能达到原始数据的数倍[19]。因此，如何设计一种轻量高效的方式来加速 LSM 树上的查询，成为当前面临的一项重大挑战。

2.4 LSH

LSH 是一种用于解决高维空间近似 k 近邻问题的技术。其核心思想是通过多个哈希函数将数据点映射到哈希桶中，使得在高维空间中相近的数据点以较高概率落入相同的哈希桶。这一方法可以在保证查询准确性的同时，实现亚线性时间复杂度的近似 k 近邻查询。

面对静态数据集时，LSH 展现出优异的性能。然而，LSH 由于以下原因不适用于动态环境下：1）随着数据不断更新，数据分布不断改变，原有的哈希桶划分可能会变得不再适用，导致索引需要频繁重建，这一过程复杂且耗时；2）LSH 的性能高度依赖于参数的设置（如哈希桶的大小），但对于动态工作负载，固定的参数无法有效适应动态数据的变化。

3 整体设计

图 2 展示了 Truster 的整体框架，该系统由以下 3 个模块组成。

1）预处理模块。如图 2（a）所示，该模块利用编码器（Encoder）为输入的每条轨迹 T 生成其空间感知键 k 和序列化后的值 v，并通过嵌入器（Embedder）提取其特征向量 $\boldsymbol{f}$，得到一个三元组 $(k, v, \boldsymbol{f})$，并传递给 CLSM 树进行存储。

2）CLSM 树。如图 2（b）所示，三元组 $(k, v, \boldsymbol{f})$ 首先在 CLSM 树的内存表中进行排序。随着内存表中三元组的积累，当其容量达到上限时，CLSM 树会执行 LCC 刷盘，生成一个磁盘上的 SSTable，并将分桶映射信息持久化存储在 SSTable 的桶块（Buckets Block）中。此后，磁盘上 SSTable 的维护都通过如图 2（d）所示的 LCC 合并策略完成。

3）查询模块。如图 2（c）所示，Truster 设计了多粒度缓存和相应的查询算法来加速查询。具体地，元数据块缓存 C_M 通过减少磁盘 I/O 加速键值对查询，而 LSH 缓存 C_L 和树缓存 C_T 通过快速缩小搜索空间加速范围查询和 k 近邻查询。

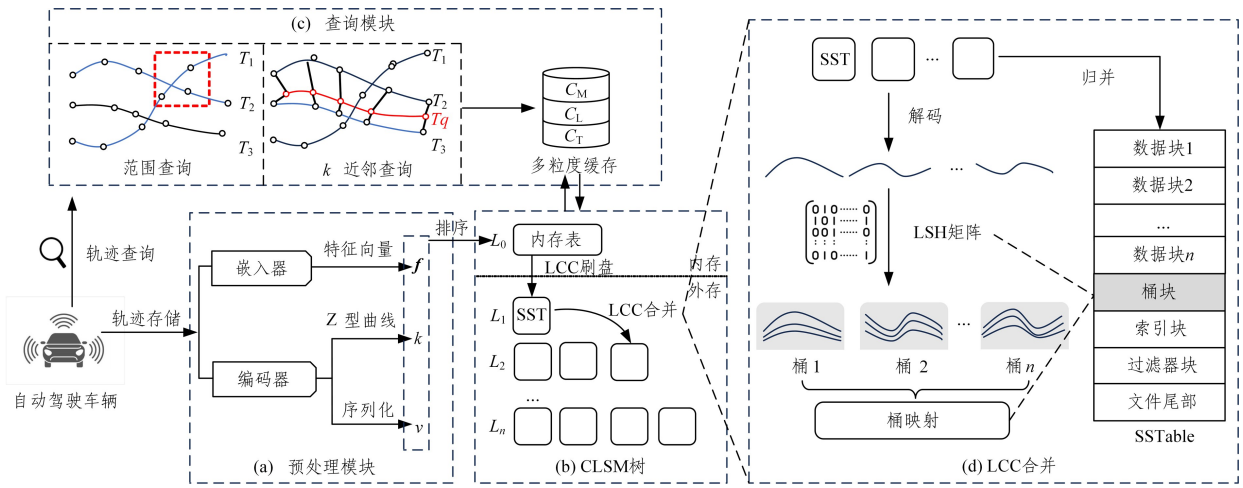


图 2 Truster 框架图

Fig. 2 Framework of Truster

4 轨迹预处理模块

4.1 轨迹编码

CLSM 树的内存表根据键的顺序对数据进行排序。为充分利用这种排序机制，Truster 设计了一个轨迹编码器，用于生成空间感知键。该键包含轨迹的基本位置信息，使轨迹在 CLSM 树中实现初步排序和分类，为后续聚类分桶奠定基础。

定义 5（轨迹编码） 给定一条轨迹 T，轨迹编码指将 T 映射成一个键值对 (k, v) 的过程，其中 k 为每条轨迹唯一的空间感知键，v 封装了 T 的完整信息。具体而言，$k = Z(x_0, y_0) +$

r_0，其中 $Z(\cdot,\cdot)$ 表示 Z 型曲线映射函数，而 x_0，y_0，r_0 分别表示 T 的最小外接圆（Minimum Bounding Box，MBB）的圆心坐标和半径。

4.2 轨迹嵌入

为便于后续 LCC 合并中对轨迹的快速聚类，Truster 设计了轨迹嵌入器，用于从轨迹 T 中提取相关信息并生成特征向量 $\boldsymbol{f}$。

定义 6（轨迹嵌入） 给定一条轨迹 T，轨迹嵌入指将 T 映射成一个低维特征向量 $\boldsymbol{f}=(f_0,f_1,\cdots,f_n)$ 的过程，该向量能够保留 T 的关键信息。具体地，从以下 3 个方面提取 $\boldsymbol{f}$：

1）地理位置信息。包含轨迹 T 的起始坐标、终点坐标以及 MBB 圆心坐标。这些特征提供了轨迹的地理分布与覆盖范围。

2）几何形态信息。包含轨迹 T 的长度、MBB 半径以及轨迹的空间紧凑性（凸包面积）。这些特征揭示了轨迹的整体形状与几何特征。

3）动态运动信息。包含轨迹点的数量与密度，以及最大/最小速度和方向变化次数。这些特征有助于揭示轨迹的动态行为与运动模式。

当然，这些特征指标并非绝对的，不同维度的指标在自动驾驶领域的不同上层任务中发挥着不同的作用。在实际应用中，应根据具体任务的需求调整特征维度的选择和权重。例如：在进行路径优化时，地理位置信息和几何形态信息更为关键；在进行风险评估或碰撞检测时，动态运动信息则更加重要。

定义 7（标准化） 给定一个特征向量 $\boldsymbol{f}=(f_0,f_1,\cdots,f_n)$，标准化指将 $\boldsymbol{f}$ 映射成 $\boldsymbol{f}'=(f_0',f_1',\cdots,f_n')$ 的过程，其中 $f_i'=\frac{f_i-\mu}{\sigma}$，$i\in[0,n]$，$\mu$ 和 σ 分别表示 f_i 的均值和标准差。

在生成 $\boldsymbol{f}=(f_0,f_1,\cdots,f_9)$ 后对其进行标准化，以避免单一分量 $f_i(i\in[1,9])$ 对 $\boldsymbol{f}$ 的整体表示产生过大影响。该过程通过减去每个特征的均值并除以其标准差，使得每个 $f_i(i\in[1,9])$ 均被缩放至均值为 0、标准差为 1 的范围内。

5 CLSM 树

5.1 CLSM 树结构

如图 2(b)所示，CLSM 树是一个内外存混合存储结构，它主要包含以下两个组件。

1）内存表。内存表是内存上的数据组织格式，由跳表（Skip List）实现。当轨迹 T 被转换为三元组 $(k,v,\boldsymbol{f})$ 后，将被暂存在内存表中并按照键的大小进行排序。当内存表的大小达到设定阈值时，将触发 LCC 刷盘操作将内存表中的数据写入磁盘。

2）SSTable。SSTable 是磁盘上的数据组织格式，分布在从 L_1 到 L_n 的不同层级。每个 SSTable 均由刷盘或合并操作生成。如图 2(d)所示，Truster 的 SSTable 与常规 LSM 树的 SSTable 相比，增加了一个桶块，用于记录 LCC 合并过程中哈希分桶的映射信息。

5.2 LCC 策略

在 Truster 的设计中，为了加速轨迹查询，Truster 将相似的轨迹集中存储，即在轨迹存储过程中实现类似轨迹聚类的效果。然而，轨迹聚类是一个计算密集型的过程。为避免对持续写入操作的影响，Truster 提出了 LCC 策略。具体而言，它从以下 3 个方面进行了优化。

1）基于 LSH 的快速聚类。传统轨迹聚类算法的时间复杂度为 $O(n^2)$，计算开销较大。为了提高聚类效率，Truster 将轨迹 T 映射为低维特征向量 $\boldsymbol{f}$，并进一步利用 LSH 方法将其映射到相应的桶中，从而将轨迹聚类的时间复杂度降低至 $O(nB)$，其中 B 为桶数量。

2）聚类合并同步执行。合并的过程涉及在后台线程中进行的数据解码和排序。Truster 对合并算法进行了改造，使其在执行合并的同时，能够同步完成轨迹聚类与空间索引的构建。通过将聚类过程嵌入合并中，Truster 能够将静态的 LSH 方法应用于动态工作环境中，从而在面对频繁更新的自动驾驶轨迹数据时仍然维持高效的查询响应速度。

3）合并选择策略。传统的合并选择策略通常考虑键的重叠程度和当前层级的容量占用比等因素。在此基础上，Truster 设计了一种更适用于 LCC 策略的合并选择策略。该策略旨在将不均匀分桶的 SSTable 加入合并过程，减少不均匀的 SSTable，从而提升 LSH 的准确性和高效性。

5.2.1 基于 LSH 的快速聚类

基于 LSH 的快速轨迹聚类的具体流程如算法 1 所示。首先，根据轨迹规模和桶容量 β 动态调整分桶数量（第 4 行）。随后，通过 Superbit 方法[20]生成随机超平面矩阵（第 5 行）。接着，将特征向量 $\boldsymbol{f}$ 正交投影到其补空间中，得到轨迹的特征签名，并将其映射得到相应的桶编号（第 7 行）。被映射至相同桶的轨迹被视为同一簇，并记录在桶集合 B 中（第 8 行）。聚类完成后，遍历所有桶以检查是否存在分桶不均的情况，对于此类桶，构建 KD 树并记录到 T 中，以避免查询过程中冗余的相似性计算（第 9－12 行）。

算法 1 基于 LSH 的快速聚类

输入：轨迹特征向量集 $\mathbf{F}$，桶容量 β

输出：LSH 分桶映射 B，LSH 矩阵 $\mathbf{M}$，树映射 T

1. B={}；
2. T={}；
3. d=|**F**.dimensions|；// **f** 的维度
4. $n=\frac{|\mathbf{F}|}{\beta}$；// 分桶数量
5. **M**=initLSH(d,n,β)；// 生成互相垂直的随机超平面，组成 LSH 矩阵
6. Foreach **f**∈**F** do：
7. $\quad b_{id}$=LSH(**f**,**M**)；
8. $\quad$ B.get(b_{id}).add(**f**)；
9. Foreach ⟨b_{id},cluster⟩∈**B** do：
10. $\quad$ If |cluster|>β then：
11. $\quad\quad$ kdTree=buildKD(cluster)；
12. $\quad\quad$ T.put(b_{id},kdTree)；
13. Return B,**M**,T.

5.2.2 聚类合并同步执行

在 2.4 节中提到，在动态环境下，LSH 索引稳定性较低，可能导致桶内轨迹的失配，从而需要花费大量开销频繁重建索引。为解决这一问题，Truster 将 LSH 索引的生成过程与 CLSM 树的合并过程相结合，即实现了 LSH 的 LSM 化，使聚类和合并能够同步执行。在此过程中，通过 CLSM 树提供的

动态合并机制，使 LSH 索引在自动驾驶轨迹数据不断更新的动态环境中能够自适应地进行调整，从而克服静态 LSH 索引的局限。

聚类合并同步执行算法的具体流程如算法 2 所示。首先在目标层级L_i生成一个空的 $SSTable_j$（第 1 行）；对参与合并的所有 SSTable，采用归并的思想，每次取其中最小的键对应的键值对（第 3－4 行），将数据解码并计算出特征向量 $\boldsymbol{f}$（第 5 行）；得到所有特征向量后，执行基于 LSH 的快速轨迹聚类（第 8 行）；接着，更新缓存（第 9－12 行）；最后，检查此次合并的结果是否触发级联合并（第 13 行）。此算法着重展示如何将聚类嵌入合并过程同步执行，因此屏蔽了大量细节。例如，第 7 行往新 SSTable 中添加键值对时，涉及数据块、过滤器块、索引块的维护；第 12 行描述的 SSTable 整体刷盘，在实际执行时采用的是逐块写满刷盘策略，包括桶块在内的元数据块将按照固定顺序进行刷盘。

算法 2 聚类合并同步执行

输入：SSTable 集合 S，目标层级L_i，桶容量 β，缓存 C＝{元缓存C_M，LSH 缓存C_L，树缓存 C_T}

输出：新生成的 SSTable sst

1. ⟨j, sst⟩＝initSST(L_t)；
2. **F**＝{}；
3. while |S. data|＞0 do：
4. ⟨k, v, **f**⟩＝S. popMin()；
5. **f**＝embed(v)；
6. **F**. add(**f**)；
7. sst. add(k, v)；
8. B, **M**, T＝LSHBasedClustering(**F**, β)；
9. C_M. update(j, sst)；
10. C_L. update(j, ⟨B, **M**⟩)；
11. C_T. update(j, T)；
12. sst. flush()；
13. 检查是否触发递归合并；
14. Return sst.

5.2.3 合并选择策略

合并是 LSM 树的一种数据维护机制，旨在将多个小型 SSTable 合并为更大的 SSTable，从而减少 SSTable 之间的重叠，减少查询过程中需要扫描的 SSTable 数量，从而提高查询性能。在 CLSM 树中，不均匀的分桶会严重影响查询性能。因此，Truster 对 CLSM 树中的合并选择算法进行了改进，具体流程见算法 3。该算法包含两个核心步骤。第一步是确定进行合并的层级L_t（第 1－11 行）。该过程通过计算每个层级的分数来完成，分数最大的被选定为L_t。分数的计算依据两项因素：第一项为该层的总数据规模，对于L_1，体现为 SSTable 的数量（第 4 行），对于其他层级，则体现为该层总大小与其总容量的比值（第 6 行）；第二项为该层中不均匀分桶的数量（第 7 行）。算法的第二步是选择需要合并的 SSTable 集合 S。如果参与合并的是L_1，则直接将L_1的所有 SSTable 加入待考虑集合（第 13 行），否则找到L_t中不均匀桶最多的 SSTable 加入待考虑集合（第 15－16 行）。随后，将L_t和L_{t+1}的所有 SSTable 划分为两部分 S 和 S'（第 17 行），算法会持续挑选 S'中的的 SSTable 加入 S，直至L_t和L_{t+1}的所有 SSTable 被划分为两个完全不重叠部分（第 18－22 行）。

算法 3 合并选择

输入：CLSM 树的所有层级 L＝{L_1, L_2, …, L_n}，分数权重 w_1, w_2，触发合并的分数阈值 θ

输出：目标层级L_t，SSTable 集合 S

// 第一步：寻找目标层级L_t

1. M＝{}；// 记录每个层级的分数
2. Foreach L_i∈L do：
3. If i＝＝1 then：
4. $sc_1 = \frac{L_i\text{中 SSTable 的数量}}{4}$；
5. else：
6. $sc_1 = \frac{L_i\text{中 SSTable 大小之和}}{L_i\text{容量}}$；
7. $sc_2 = \frac{L_i\text{中不均匀桶的数量}}{L_i\text{中桶总数}}$；
8. M. put(L_i, $sc_1 * w_1 + sc_2 * w_2$)；
9. ⟨L_t, sc_{max}⟩＝M. getMax()；
10. If $sc_{max} < \theta$ then：
11. Return －1, ∅；

// 第二步：寻找 SSTable 集合 S

12. If L_t＝＝L_1 then：
13. S. addAll(L_1)；
14. else：
15. ss t_0＝L_t中不均匀桶最多的 SSTable；
16. S. add(ss t_0)；
17. $S' = L_t \cup L_{t+1} - S$；
18. While hasOverlap(S, S′) do：
19. Foreach sst∈S′ do：
20. If hasOverlap(S, sst) then：
21. S. add(sst)；
22. S′. remove(sst)；
23. Return L_t, S.

6 搜索模块

6.1 多粒度缓存

性质 1 在 LSM 树中的查询操作总是从L_0层开始，逐层向L_n层推进，直至找到目标键。

推论 1 在 LSM 树中，SSTable 所在的层级越高，其被访问的频率越大。

性质 2 在单个 SSTable 中的查找操作总是始于其元数据块，而元数据块在整个 SSTable 中仅占极小的一部分。

基于上述性质和推论，Truster 设计了多粒度缓存策略，它将L_1－L_3的部分内容缓存在内存中，以优化查询性能。具体地，该缓存结构包括以下 3 个部分。

1）元数据块缓存C_M。元数据块缓存用于缓存每个 SSTable 的尾部、过滤器块和索引块。C_M能够在后续访问同一个 SSTable 时，避免从磁盘中重复读取与构建 SSTable，从而极大地提升查询效率。

2）LSH 缓存C_L。LSH 缓存用于缓存每个 SSTable 的 LSH 矩阵及分桶映射信息。在进行轨迹查询时，以 k 近邻查询为例，需要利用在 LCC 过程中生成的随机超平面对查询轨迹T_q进行投影，从而定位到与T_q相似度最高的桶中，然后取出桶内的轨迹进行进一步的比较。因此，C_L能够在轨迹查询时避免重新读取磁盘构建 LSH 矩阵和分桶映射。

3）树缓存C_T。树缓存用于缓存每个不均匀桶的 KD 树。在进行轨迹查询时，Truster 通过 LSH 映射快速定位到相应的分桶，从而有效缩小搜索空间。然而，当定位到的分桶较大时，就无法达到筛选搜索空间的效果。因此，Truster 针对每个不均匀分桶构建了 KD 树，以加速不均匀分桶内的轨迹查询。

6.2 查询算法

Truster 的查询算法的核心在于利用分桶进行搜索空间的快速筛选。算法 4 以 k 近邻查询为例，详细描述了这一过程。该算法分为两步：第一步利用分桶快速缩小搜索空间（第 2－23 行）；第二步在候选空间中进行精确计算以寻找与查询轨迹T_q相似度最高的 k 条轨迹（第 24－29 行）。

首先提取查询轨迹T_q的特征向量 $\boldsymbol{f}$（第 2 行）。对于每个层级中的每个 SSTable，从 LSH 缓存C_L中获取对应的分桶映射信息 B 和 LSH 矩阵 $\boldsymbol{M}$（第 8 行），并利用 $\boldsymbol{M}$ 将 $\boldsymbol{f}$ 映射成相应的桶编号b_{id}（第 9 行）。如果b_{id}对应的是一个不均匀分桶，则利用树缓存C_T获取其 KD 树（第 13 行），并执行 KD 树上的查询（第 14 行），否则将b_{id}中的所有轨迹加入候选集中（第 16 行）。在 CLSM 树的扫描完成一遍以后，如果候选集中轨迹的数量仍小于 k（第 5 行），则通过调整桶号偏移量 ε 来将相邻分桶的轨迹纳入候选集。具体地，设定一个初始值为 0 的桶号偏移 ε（第 3 行），并在每次循环后自增（第 24 行），从而在下一次循环中获取相邻的桶（第 10－11 行）。该循环会持续进行，直到候选集 S 中的元素个数达到 k 个（第 5 行）。

随后，初始化一个容量为 k 的自定义优先队列 P（第 25 行），该队列根据查询需求定义的相似度计算函数进行内部排序。通过将候选键集合 S 中的所有轨迹依次插入该优先队列进行筛选（第 26－29 行），最终保留的相似度最高的 k 条轨迹即为查询的最终结果。

算法 4 k 近邻查询

输入：k，查询轨迹T_q，CLSM 树的所有层级 $L=\{L_1,L_2,\cdots,L_n\}$，缓存 $C=\{C_M,C_L,C_T\}$

输出：目标轨迹集合 R

```
1. R={};
// 第一步：从桶中挑选足够的候选轨迹
2. f=embed(T_q);
3. ε=0;// 桶号偏移
4. S={};// 候选轨迹的键集合
5. While |S|<k do:
6.     Foreach L_i∈L do:
7.         Foreach sst∈L_i do:
8.             ⟨B,M⟩=C_L.get(sst);
9.             b_id=LSH(f,M);
10.            b_id1=b_id−ε;
11.            b_id2=b_id+ε;
12.            If b_id1∈C_T then:
13.                kd=C_T.get(b_id1);
14.                S′=kd.get(f,k);
15.            else:
16.                S′=B.get(b_id1);
17.                S.addAll(S′);
18.            If b_id2∈C_T then:
19.                kd=C_T.get(b_id2);
20.                S′=kd.get(f,k);
21.            else:
22.                S′=B.get(b_id2)
23.                S.addAll(S′);
24.                ε++;
// 第二步：在候选轨迹中进行精确筛选
25. P={};// 容量为 k 的优先级队列
26. Foreach key∈S do:
27.     value=lookup(key);
28.     T=deserialize(value);
29.     P.add(T);
30.     R←P;
31. Return R.
```

7 实验设计

7.1 实验设置

本实验采用 Argoverse 1 运动预测数据集[4]。Argoverse 1 是 Argo AI 发布的首个开源自动驾驶数据集，专为自动驾驶汽车的 3D 跟踪、运动预测和其他感知任务的训练与测试设计。该数据集包含高精度地图、3D 追踪和运动预测 3 部分数据。本实验采用其中的运动预测数据集，该部分数据包含自动驾驶车辆在匹兹堡和迈阿密两座城市中采集的轨迹数据，覆盖迈阿密 204 公里和匹兹堡 86 公里的车道网络，包含 324557个车辆轨迹场景，总持续时长超过 320 小时。每个场景持续 5 秒，采样频率为 10 Hz，包含每个被跟踪对象的 2D 鸟瞰质心。平均每条轨迹对应 50 个点。

本实验将 Truster 与两种最先进的系统进行对比，即 Spatialite 和 Dragon[9]。其中，Spatialite 是 SQLite 的时空拓展版本，它使用基于 B^* 树的存储结构和基于 R 树的空间索引；Dragon 是一个基于 Spark 的计算分析引擎，由于其并不提供外部存储功能，因此实验中使用 RocksDB 作为其存储引擎来支持数据持久化。实验采用的数据规模如表 1 所列。实验中各项参数的设置如表 2 所列，其中加下划线的参数值为默认设置。

表 1 数据规模

Table 1 Data size

轨迹数量/条	轨迹点数量/个	原始数据大小/MB
103	3165	0.3
1002	22907	2.2
10007	227687	22.4
100004	2270725	222.9

表 2 参数设置

Table 2 Parameter settings

参数名	值
采用的轨迹条数$\lvert D\rvert$	10^2，10^3，$\underline{10^4}$，10^5
近邻查询参数 k	1，<u>10</u>，50，100
范围查询窗口宽度 d/米	1，<u>10</u>，50，100
桶容量 β	10，<u>20</u>，30，40

运行环境为：本实验运行在 Nexus 5 的 Android 12.0 Google Play 模拟器上。该模拟器配置了 1.5 GB 的 RAM 和基于 x86_64 架构的 Qualcomm Snapdragon 800 CPU。此外，实验使用 Android Gradle 3.5.4 和 Gradle 7.4 进行配置。

7.2 存储性能

7.2.1 插入速度

图 3 展示了 Truster 和两个对比基准分别插入不同规模轨迹所需的时间。从图中可见，Truster 的写入性能优于其他基准系统。具体地，Truster 的写入时间比 Spatialite 快 40%～50%，这主要得益于其 CLSM 树比 B 树更高效的写入性能。与同样基于 LSM 树的 Dragon 相比，Truster 的加载速度依然提升了近 10%，这一优势主要源于 Truster 在构建空间索引时比 Dragon 的 R 树更快。

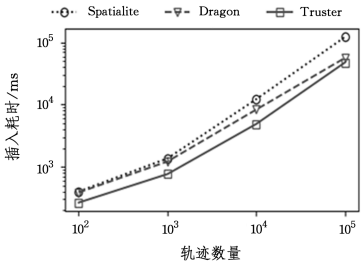


图 3 不同轨迹条数 $|D|$ 的写入耗时

Fig. 3 Insertion time with different $|D|$

7.2.2 内外存占用

图 4 与图 5 分别展示了加载不同规模轨迹时各系统的内存与外存占用情况。

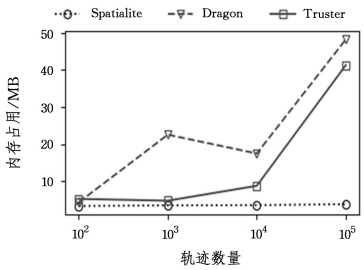


图 4 不同轨迹条数 $|D|$ 的内存占用

Fig. 4 Memory cost with different $|D|$

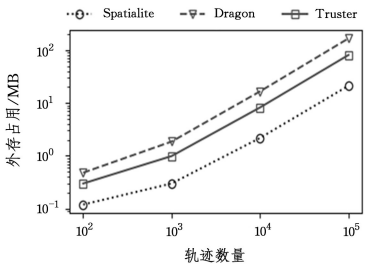


图 5 不同轨迹条数 $|D|$ 的外存占用

Fig. 5 Storage cost with different $|D|$

从图中可见，在内外存占用方面，Truster 优于 Dragon，但相较于 Spatialite 略显不足。具体而言，在内存占用方面，由于 Truster 设计了多粒度缓存以提升查询效率，因此其内存占用略高属于合理情况；而 Spatialite 采用延迟加载技术，仅在需要时加载必要的部分，内存占用保持相对稳定。在外存占用方面，Spatialite 作为关系型数据库，通过预定义表结构实现了更为紧凑的编码方式，优化了外存利用效率。

7.2.3 索引开销

图 6 展示了在加载不同规模数据时各系统空间索引的构建耗时。从图中可见，Truster 在索引构建时间上明显优于 Spatialite 和 Dragon，实现了近 10 倍的加速。

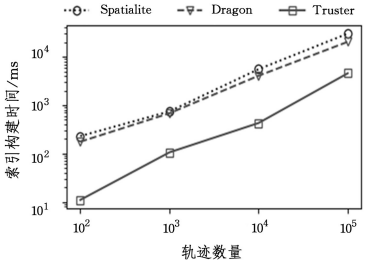


图 6 不同轨迹条数 $|D|$ 的索引构建时间

Fig. 6 Index construction time with different $|D|$

7.3 查询性能

7.3.1 范围查询

图 7 和图 8 分别展示了在不同轨迹规模 $|D|$ 和不同窗口宽度 d 下，各系统执行范围查询的平均查询时延。从图中可见，Truster 的范围查询时延始终优于其他基准系统。在图 7 中，随着 $|D|$ 的增大，Truster 范围查询的延迟随之上升。这是 $|D|$ 的增大导致 SSTable 的数量、层级和大小的增加，从而在查询时需要更多时间进行不同层级间结果的合并。图 8 中，在 d 较小时，Truster 的性能优势更显著。这是因为此时查询窗口更为精确，结合空间感知键的设计，目标轨迹往往集中于极少数 SSTable 中，从而显著缩短了不同层级间结果合并的耗时。

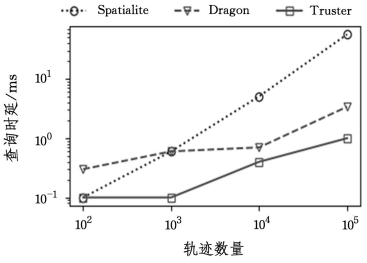


图 7 不同轨迹条数 $|D|$ 下的范围查询时延

Fig. 7 Latency of range query with different $|D|$

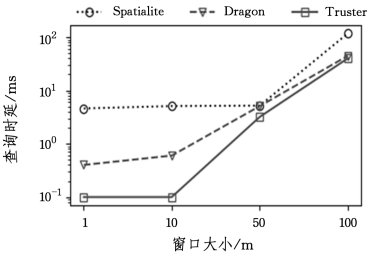


图 8 不同窗口大小 d 下的范围查询时延

Fig. 8 Latency of range query with different d

7.3.2 k 近邻查询

图 9 和图 10 分别展示了在不同轨迹规模 $|D|$ 和不同 k 下，各系统执行 k 近邻查询的平均查询时延。从图中可见，Truster 的 k 近邻查询性能比较稳定，不随着 $|D|$ 的增加而显著变化，且受 k 的影响较小。在图 9 中，当 $|D|$ 较小时，Truster 的 k 近邻查询时延略逊于 Spatialite。这是因为 Truster 在查询过程中不可避免地需要执行至少 k 次轨迹解码和相似度

排序操作。然而，随着$|D|$的增大，Truster 能够快速筛除大量无关的搜索空间，从而展示出更明显的性能优势。图 10 中，当 k 增大时，k 近邻查询时延略微增大。这是由于当 k 超过单个桶的容量时，需要合并多个桶中的数据并进行相似度排序，从而增加查询延迟。

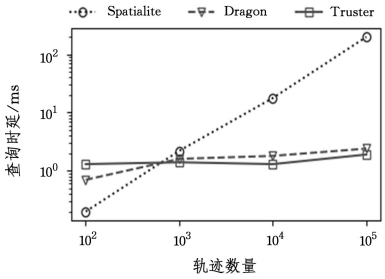


图 9　不同轨迹条数$|D|$下的 k 近邻查询时延

Fig. 9　Latency of kNN query with different $|D|$

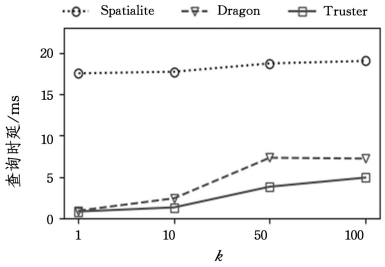


图 10　不同 k 下的 k 近邻查询时延

Fig. 10　Latency of kNN query with different k

7.3.3　复合查询

复合查询指一次查询中含有多个查询条件，这是实际应用场景中极为常见的查询类型。图 11 展示了在不同轨迹规模$|D|$下，各系统执行复合查询的平均查询时延。由图可知，Truster 的复合查询时延始终优于其他基准系统。

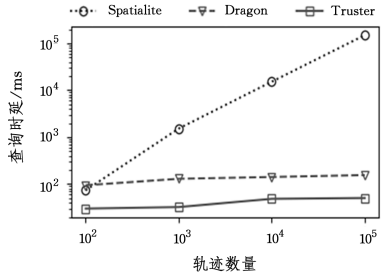


图 11　不同轨迹条数$|D|$下的复合查询时延

Fig. 11　Latency of composite query with different $|D|$

7.3.4　参数调优

为了验证树缓存C_T的有效性，在箱型图 12 中展示了启用与禁用C_T时 k 近邻查询时延的分布情况。从图中可见，在启动C_T时，k 近邻查询时延的分布更集中，性能更平稳；而在不启用C_T且 β 较大时，k 近邻查询时延出现显著波动，在最坏的情况下，查询时延甚至可能超过最佳性能的 10 倍。这是由于不均匀的大桶在相似度排序时需要消耗大量时间。相较之下，启用C_T为数据密集的大桶构建 KD 树，能够显著提高最坏情况下的查询性能。

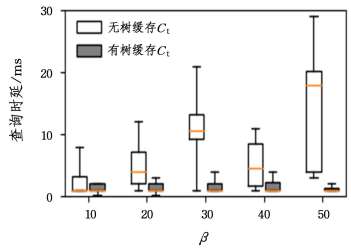


图 12　不同桶大小 β 下的 k 近邻查询时延

Fig. 12　Latency of kNN query with different β

为了在索引构建成本与查询时延之间进行权衡，在图 12 与图 13 中分别展示了不同桶大小 β 下 k 近邻查询的时延和索引构建的时间与内存开销。从图中可见，将 β 设定为 20 可以在获得较高查询性能的同时，维持较低的索引开销。

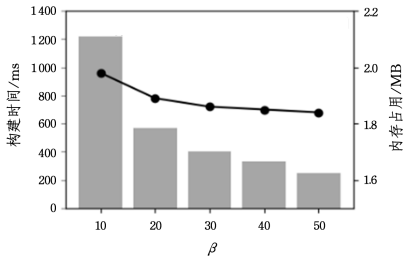


图 13　不同桶大小 β 下的索引开销

Fig. 13　Index overhead with different β

7.4　消融实验

为了验证 Truster 各模块的独立作用，本节通过消融实验对各模块的影响进行综合评估。实验结果如图 14 所示，从中可以得出以下结论：1）启用缓存虽然会牺牲一定的内存空间，但能获得更快的写入速度与大幅提升的查询速度；2）启用空间感知键与 LCC 策略虽然会牺牲一定的写入速度与内存空间，但能获得查询速度的大幅提升。

禁用C_M后，虽然内存占用有所减少，但查询时延显著增加。这是因为在 CLSM 树中，所有查询总是从高层的元数据块开始。当禁用C_M后，每次查询都必须重新从磁盘上加载对应的元数据块，这显著增加了磁盘 I/O 开销。

同理，C_L缓存了 LSH 映射矩阵和分桶信息，C_T缓存了不均匀分桶的 KD 树。禁用这些模块在一定程度上可以减小内存占用，但每次查询仍需从磁盘加载相关数据，而磁盘访问的代价较高，导致查询时延成倍增加。

此外，禁用缓存一定程度上会造成写入耗时的增加。这是因为写入过程包含 SSTable 的合并过程，而这个过程涉及数据的解码与重排序。当不启用缓存时，每次都需要重新从磁盘加载相应的数据才能继续正常访问 SSTable，从而增加了写入的延迟。

禁用空间感知键在一定程度上会降低查询性能。这是因为失去了空间感知键对空间的初步划分后，轨迹在整个 CLSM 树中的分布将更加分散，所以 LCC 策略在轨迹筛选时的效果减弱。

禁用 LCC 策略，即不进行 LSH 映射分桶，将显著增加 k 近邻查询的时延。这是因为此时 k 近邻查询需要遍历整个数据集以找到和查询轨迹最相似的 k 条轨迹。而在 LCC 策略下，通过分桶可以快速定位到目标桶或者其相邻

分桶中，从而有效筛选查询空间。

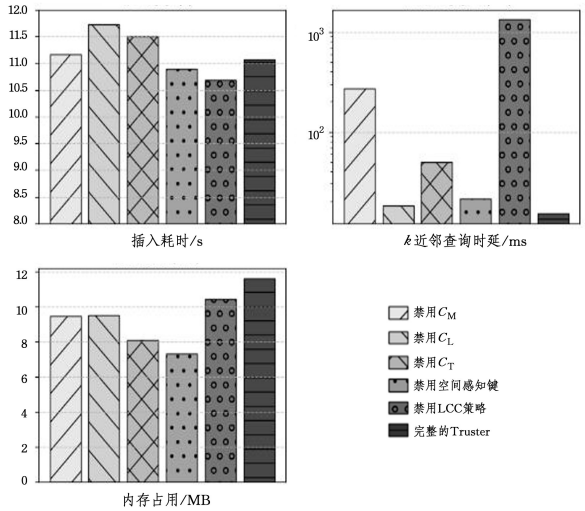


图 14 禁用不同模块对 Truster 性能的影响

Fig. 14 Impact of module removal on Truster's performance

结束语 本文提出了一种面向自动驾驶轨迹数据的存储方案 Truster，旨在通过聚类存储提高查询效率。具体地，Truster 设计了 CLSM 树和 LCC 合并策略，通过将相似的轨迹集中存储，从而快速过滤搜索空间，加速轨迹查询。在真实自动驾驶轨迹数据集上的实验结果表明，Truster 的写入速度、查询时延和索引开销均优于基准系统。

未来，将优化 Truster 的内存占用，并应用轨迹压缩技术以降低外存消耗，使 Truster 更加适用于资源受限的自动驾驶场景。

参 考 文 献

[1] ZEUCH S, CHATZILIADIS X, CHAUDHARY A, et al. NebulaStream: Data Management for the Internet of Things[J]. Datenbank-Spektrum, 2022, 22(2): 131-141.

[2] SCHELTER S, LANGE D, SCHMIDT P, et al. Automating large-scale data quality verification[C]// Proceedings of the VLDB Endowment. 2018: 1781-1794.

[3] VANB J, O'BRIEN M, GRUYER D, et al. Autonomous vehicle perception: The technology of today and tomorrow[J]. Transportation Research Part C: Emerging Technologies, 2018, 89: 384-406.

[4] CHANG M F, LAMBERT J, SANGKLOY P, et al. Argoverse: 3d tracking and forecasting with rich maps[C]// Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2019: 8748-8757.

[5] CAESAR H, BANKITI V, LANG A H, et al. nuscenes: A multimodal dataset for autonomous driving[C]// Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2020: 11621-11631.

[6] ZIMÁNYI E, SAKR M, LESUISSE A. MobilityDB: A mobility database based on PostgreSQL and PostGIS[J]. ACM Transactions on Database Systems, 2020, 45(4): 1-42.

[7] ALSUBAIEE S, BEHM A, BORKAR V, et al. Storage management in AsterixDB[C]// Proceedings of the VLDB Endowment. 2014: 841-852.

[8] SHIN J, WANG J, AREF W G. The LSM RUM-tree: a log structured merge R-tree for update-intensive spatial workloads [C]// 2021 IEEE 37th International Conference on Data Engineering. IEEE, 2021: 2285-2290.

[9] FANG Z, CHEN L, GAO Y, et al. Dragoon: a hybrid and efficient big trajectory management system for offline and online analytics[J]. The VLDB Journal, 2021, 30: 287-310.

[10] WANG S, BAO Z, CULPEPPER J S, et al. Torch: A search engine for trajectory data[C]// The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval. 2018: 535-544.

[11] LAN H, XIE J, BAO Z, et al. Vre: a versatile, robust, and economical trajectory data system[C]// Proceedings of the VLDB Endowment. 2022: 3398-3410.

[12] BAO Y, HUANG Z, GONG X, et al. Optimizing segmented trajectory data storage with HBase for improved spatio-temporal query efficiency[J]. International Journal of Digital Earth, 2023, 16(1): 1124-1143.

[13] KIM Y S, KIM T, CAREY M J, et al. A comparative study of log-structured merge-tree-based spatial indexes for big data [C]// 2017 IEEE 33rd International Conference on Data Engineering(ICDE). IEEE, 2017: 147-150.

[14] O'NEIL P, CHENG E, GAWLICK D, et al. The log-structured merge-tree(LSM-tree)[J]. Acta Informatica, 1996, 33: 351-385.

[15] KOGA H, ISHIBASHI T, WATANABE T. Fast agglomerative hierarchical clustering algorithm using Locality-Sensitive Hashing[J]. Knowledge and Information Systems, 2007, 12: 25-53.

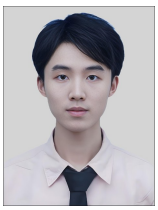
[16] SANCHEZ I, AYE Z M M, RUBINSTEIN B I P, et al. Fast trajectory clustering using hashing methods[C]// 2016 International Joint Conference on Neural Networks(IJCNN). IEEE, 2016: 3689-3696.

[17] CHANG F, DEAN J, GHEMAWAT S, et al. Bigtable: A distributed storage system for structured data[J]. ACM Transactions on Computer Systems, 2008, 26(2): 1-26.

[18] SEARS R, RAMAKRISHNAN R. bLSM: a general purpose log structured merge tree[C]// Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. 2012: 217-228.

[19] LI P, LU H, ZHENG Q, et al. LISA: A learned index structure for spatial data[C]// Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. 2020: 2119-2133.

[20] JI J, LI J, YAN S, et al. Super-bit locality-sensitive hashing [C]// Proceedings of the 26th International Conference on Neural Information Processing Systems. 2012: 108-116.



WANG Zhengquan, born in 2000, postgraduate, is a member of CCF (No. N4586G). His main research interests include mobile database and storage management.

PENG Zhiyong, born in 1963, Ph.D supervisor, is a member of CCF (No. 06132F). His main research interests include advanced database management and Web information management, etc.

(责任编辑：李亚辉)