软件学报 ISSN 1000-9825, CODEN RUXUEW
[doi: 10.13328/j.cnki.jos.007638] [CSTR: 32375.14.jos.007638]

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

图数据库节点标识存储引擎性能建模与适配策略*

陈 政[1,2,3], 张 峰[1,2], 齐 畅[1,2], 赵郑诣隆[1,2], 杜小勇[1,2]

[1](数据工程与知识工程教育部重点实验室 (中国人民大学), 北京 100872)

[2](中国人民大学 信息学院, 北京 100872)

[3](清华大学 计算机科学与技术系, 北京 100084)

通信作者: 张峰, E-mail: fengzhang@ruc.edu.cn

摘 要: 随着大数据和人工智能技术的快速发展, 图数据库因其在复杂关系建模与高效查询方面的优势, 逐渐成为社交网络分析、金融风控、知识图谱等领域的核心基础设施. 图数据库的管理对象是节点以及节点之间的关系. 在架构层面, 节点唯一标识符 (node identifier, NodeID) 作为图数据管理的核心纽带, 承担节点身份表征、关系寻址和图算法执行的关键职能. 当前主流图数据库普遍采用键值存储 (key-value store, KVS) 引擎实现节点标识到图结构的映射管理. 然而, 现有系统多依赖通用键值存储引擎 (如 RocksDB) 管理此类映射, 却缺乏对负载特性的深度考量, 具体表现为: 1) 节点标识映射负载特征建模缺失; 2) 跨软硬件环境 (如 CPU/内存、SSD/HDD) 的适应性不足. 首先系统分析图数据库中节点标识映射的操作特性与键值存储需求, 进而评估多种主流键值存储引擎 (包括 RocksDB、LMDB、LevelDB、FasterKV 及 ForestDB) 在异构硬件环境下的性能表现, 系统揭示不同数据负载 (如数据规模、读写比) 与硬件配置 (如内存容量、线程数、存储介质) 对执行效率的影响规律. 基于大规模实验 (覆盖 5 类数据集、3 种硬件平台及 1 300+组对照测试), 提出一种基于决策树模型的适配策略, 整合负载特征 (数据规模、读写比) 与硬件配置 (内存、线程数、磁盘类型), 以指导键值存储引擎的自适应选择. 实验表明, 该模型推荐最优引擎的准确率达 92.1%, 次优场景性能差距小于 10%.

关键词: 图数据库; 键值存储引擎; 顶点映射; 决策树

中图法分类号: TP311

中文引用格式: 陈政, 张峰, 齐畅, 赵郑诣隆, 杜小勇. 图数据库节点标识存储引擎性能建模与适配策略. 软件学报. http://www.jos.org.cn/1000-9825/7638.htm

英文引用格式: Chen Z, Zhang F, Qi C, Zhao ZYL, Du XY. Performance Modeling and Adaptation Strategy for Node Identifier Store Engines in Graph Databases. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7638.htm

Performance Modeling and Adaptation Strategy for Node Identifier Store Engines in Graph Databases

CHEN Zheng[1,2,3], ZHANG Feng[1,2], QI Chang[1,2], ZHAO Zheng-Yi-Long[1,2], DU Xiao-Yong[1,2]

[1](Key Laboratory of Data Engineering and Knowledge Engineering (Renmin University of China), Ministry of Education, Beijing 100872, China)

[2](School of Information, Renmin University of China, Beijing 100872, China)

[3](Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)

Abstract: With the rapid development of big data and artificial intelligence technologies, graph databases have gradually become core infrastructure for social network analysis, financial risk control, and knowledge graphs due to their advantages in complex relationship modeling and efficient querying. Graph databases manage two primary objects: nodes and the relationships between them. At the

* 基金项目: 国家自然科学基金 (U25B2018, 62322213, 62461146205); 中国博士后科学基金 (2025M781495); 国家资助博士后研究人员计划 (GZC20251044); 北京市科技计划 (Z251100008125032); 清华大学水木学者项目 (2025SM061)
收稿时间: 2025-08-18; 修改时间: 2025-11-20; 采用时间: 2026-01-19; jos 在线出版时间: 2026-06-01

architectural level, the node identifier (NodeID) serves as the critical link for graph data management, undertaking key functions including node identity representation, relationship lookup, and graph algorithm execution. Current mainstream graph databases widely adopt key-value store (KVS) to implement NodeID-to-graph-structure mapping management. However, existing systems largely rely on general-purpose KVSs (e.g., RocksDB) for managing such mappings, yet lack in-depth consideration of workload characteristics. These limitations are reflected in two aspects: (1) the lack of workload modeling for node identifier mapping, and (2) insufficient adaptability to heterogeneous software and hardware environments (e.g., CPU/memory, SSD/HDD). This study first systematically analyzes the operational characteristics and KVS requirements of node identifier mapping in graph databases. It then evaluates the performance of multiple mainstream KVS engines (including RocksDB, LMDB, LevelDB, FasterKV, and ForestDB) in heterogeneous hardware environments, systematically revealing the impact patterns of data workloads (e.g., data scale, read-write ratio) and hardware configurations (e.g., memory capacity, thread count, storage medium) on execution efficiency. Based on large-scale experiments involving five datasets, three hardware platforms, and over 1 300 comparative tests, this study proposes an adaptation strategy based on decision tree model that integrates workload characteristics (data scale and read-write ratio) with hardware configurations (e.g., memory, thread count, disk type) to guide adaptive selection of KVS engines. Experiments show that the model achieves 92.1% accuracy in recommending optimal engines, with suboptimal scenarios exhibiting less than 10% performance gap.

Key words: graph database; key-value store (KVS) engine; vertex mapping; decision tree

图数据库作为一种新型的数据库系统[1,2], 近年来在大数据和人工智能领域中备受关注并迅速崛起. 与传统的关系型数据库相比[3], 图数据库因其灵活的建模方式在处理复杂关系和大规模关联数据时表现突出. 这种优势源于其底层基于图论的设计理念, 能够天然表达实体间的多跳关联, 而关系型数据库需要通过复杂的表连接操作实现类似功能, 性能开销显著更高. 图数据库通过节点和边的建模方式, 可以直观、高效地表示和查询复杂的关系网络, 适用于社交网络分析、推荐系统、知识图谱等应用场景. 另外, 图数据库在处理大规模数据集时, 具备强大的扩展性和高效的查询性能, 能够迅速响应复杂的关系查询, 提升数据分析的广度和深度[3,4]. 由于这些显著的优势, 图数据库已经成为现代企业和研究机构进行数据管理和分析的重要工具. 行业实践印证, 全球头部科技企业, 如Meta、Google、Alibaba、ANT、Amazon[5,6]等, 正广泛采用图数据库驱动业务创新与技术升级, 凸显其作为现代数据基础设施的关键地位.

非数值型原始 ID (如字符串) 导致内存访问离散化, 而连续紧凑 ID 使节点数据在物理存储上相邻, 显著提升缓存命中率. 这一优化尤其适用于现代 CPU 的缓存行 (cache line) 机制, 连续 ID 可减少缓存失效 (cache miss) 概率, 从而加速遍历类算法 (如 BFS/DFS) 的执行效率. 因此, 图数据库通常会将节点表示在一个紧凑的 ID 空间中, 这不仅可以显著减少顶点标识符的空间消耗, 也可以提升图分析任务的局部性进而提升性能. 在图数据库中, 一个基础而关键的挑战在于顶点标识符映射 (vertex ID mapping), 即将现实世界中长度不一、形式多样的原始顶点标识符 (如用户 ID、URL 字符串或长整数) 转换为紧凑且连续的整数空间. 这一过程不仅能够大幅减少存储开销 (例如将 128 位 UUID 压缩为 32 位整数), 更重要的是能够提升图相关分析算法的局部性, 从而显著加速社区发现、最短路径计算等核心操作. 以蚂蚁集团的金融风控场景为例, 每天需处理数十亿用户间的转账关系图, 若直接使用原始用户 ID (通常为 20–40 位字符串) 进行存储和计算, 内存占用将呈指数级增长, 而通过紧凑 ID 映射可将存储需求降低至原有的 10%, 同时使图计算引擎的吞吐量提升 5 倍以上[7].

图 1 说明了在一个金融公司场景下使用 ID 标识符的示例. 在这个示例中, 图上的节点是一个用户, 他们有一个原始的唯一标识符 (例如身份证号、加密字符串等长整数或字符串类型), 图上的边表示他们之间的转账记录. 实际业务中, 此类图通常呈现幂律分布 (power law distribution), 即少数高活跃用户节点连接大量边, 而多数用户边数较少, 这种特性进一步放大了紧凑 ID 对局部性优化的价值. 对于一个给定的由用户和转账关系构成的图, 系统中的典型任务是对图进行分析. 例如, 在该图上使用单源最短路径 (single-source-shortest path, SSSP) 算法[6]检测潜在的可疑用户, 这个过程涉及对图结构的访问. 针对图分析系统的研究国内外由来已久[8,9], 一个基本的优化方向是优化图应用的访存局部性[9]. 局部性指的是处理器在访问图结构和顶点相关的属性特征的时间局部性和空间局部性, 这通常要求顶点编码在紧凑的空间. 因此, 现代图处理系统通常将输入图预处理为内存紧凑的邻接表示进行存储. 在实际的应用过程中, 用户需要从原始的图表示转换为一个原始标识符-ID 的映射和基于紧凑 ID 的图表示.

通过这样做, 用户在紧凑图上执行图分析算法, 然后再用映射关系最终得到原始的节点相应的结果.

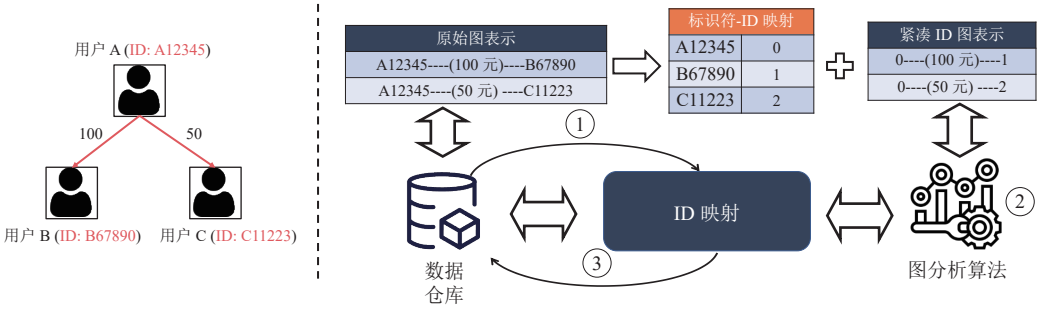


图 1 金融场景节点标识映射流程

尽管 ID 映射是图数据库的关键组件, 但是仍然缺乏系统的研究. 当前主流的图数据库通常使用通用的键值存储引擎来存储这种映射关系, 例如蚂蚁公司的图分析引擎 GeaFlow[7]使用内存中的 HashTable 存储内存处理大规模的 ID 映射, 使用 RocksDB 管理超出内存限制的 ID 映射关系. 这种选择通常是编程人员的出于易用性的考虑, 缺乏性能方面的综合考虑, 这导致了很多情况下的性能低下.

基于这个观察, 本文探讨以下两个问题.

(1) 在图数据库的 ID 映射中, 有哪些软硬件条件会影响性能?

(2) 在一个给定的数据和硬件环境中, 采用什么样的存储方式是最优的?

围绕着这两个问题, 本文在多种多样的图数据集上和多种键值存储引擎进行了大量且全面的深度分析. 本文的主要贡献如下.

(1) 构建一个面向图数据库的数据负载特征与硬件特征的键值存储引擎性能评估框架, 针对 5 类代表性键值存储引擎进行了系统且全面的评估. 实验框架围绕内存大小、磁盘 I/O、并行扩展性等 6 项关键因素设计, 通过在 5 个具有代表性的数据集上、对 5 类键值存储引擎分别采用控制变量法构造实验场景, 并在 3 类不同硬件平台上执行所有实验组合, 最终形成了超过 1 300 组具有可比性的对照实验. 本文给出了一系列关键发现, 系统揭示了不同键值存储架构下内存约束、多线程、磁盘介质差异对图数据库 ID 映射后的表示在图分析任务上的性能影响.

(2) 提出一种基于特征指标体系和决策树模型的轻量级方法, 来指导用户选择合适的 ID 映射存储引擎. 本文首先提出了若干数学指标来刻画性能相关的软硬件条件, 包括数据量大小、读写比、内存利用率、CPU 线程数、磁盘读写效率等. 在此基础上, 本文提出了一个可解释的决策树模型指定存储引擎选型, 在 38 个异构场景下实现了 92.1% 的选型准确率. 该方法支持多种硬件与多种负载的自适应选择, 为工业级图数据库提供了完整的选型方案.

本文第 1 节系统描述图数据库中的 ID 映射处理流程与潜在的性能影响. 第 2 节梳理 4 类代表性键值存储引擎, 包括哈希索引 (FasterKV[10])、LSM-Tree (RocksDB[11]、LevelDB[12])、B+ tree (LMDB[13])、HB+Trie (ForestDB[14]) 的设计哲学与适用场景. 第 3 节提出键值存储引擎性能评估框架. 第 4 节介绍基于特征指标系统与决策树模型的选型框架, 同时测试选择方法的有效性. 第 5 节总结全文.

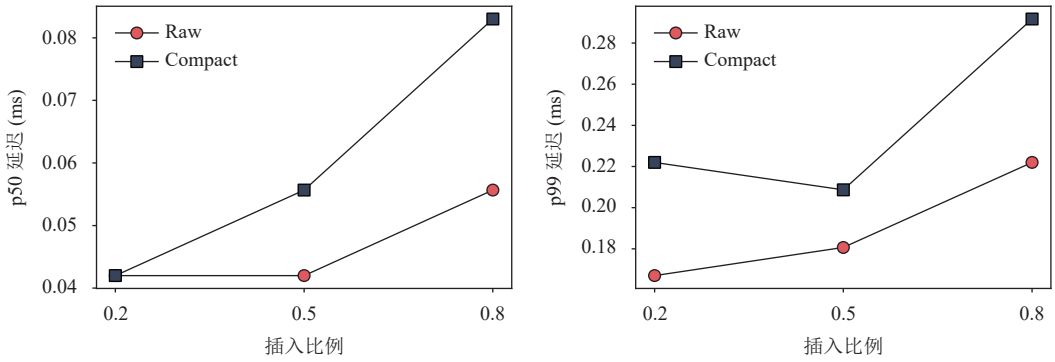
1 图数据库中的 ID 映射概述

图数据库作为非关系型数据库 (NoSQL) 的重要类型[15], 其核心价值在于通过节点和边的拓扑结构直观表达实体之间的复杂关联关系. 相较于传统关系型数据库的表格模型, 图数据库采用原生图存储, 将数据表之间的关系直接映射为图结构, 避免了多表连接带来的高昂性能损耗. 这一性使其在社交网络分析[16]、金融反欺诈[17,18]、知识图谱[19,20]等领域展现出显著优势.

在图数据库的实际应用中, 节点的原始标识符通常来源于业务主键, 例如用户手机号、URL 链接、加密字符串或 UUID 等. 这些标识符在语义上具有唯一性和可读性, 因此直接使用原始标识符 (Raw ID) 进行图数据存储和

访问具有一定的工程优势. 首先, 原始标识符可以简化数据写入路径, 避免了额外的映射构造过程. 在插入或更新节点时, 无需执行紧凑 ID 分配与查表操作, 从而减少一次随机访问与原子操作; 在高并发环境下, 这种简化能够略微降低写入延迟并减少锁竞争. 其次, Raw ID 保留了与上游业务系统的一致性, 无需额外的 ID 映射与反查开销, 使图数据库能够更直接地与事务性系统或外部存储联动, 便于日志追踪与跨系统调试. 再次, 原始标识符的方案在动态演进性与可扩展性方面更具弹性. 当上游数据源的主键策略发生变化时, 系统可以无需重新构造全局映射表, 从而避免批量重映射与重加载的高代价.

为定量评估原始标识符 (Raw ID) 与紧凑标识符 (Compact ID) 两种存储方案在插入和更新时的性能差异, 本文评估了使用哈希的标识符映射结构下两种存储方案的性能测试. Raw ID 方案直接以字符串作为顶点标识符存储和访问, 而 Compact ID 方案通过维护字符串到整数的映射表, 将外部标识符转换为内部整数 ID 后进行访问. 实验在相同的硬件配置和数据分布下执行, 采用相同的操作序列以确保对照公平性. 如图 2 结果显示, Raw ID 方案在写入阶段具有轻微的性能优势, 平均延迟低于 Compact ID 约 27%. 这一差异主要来源于 Compact 方案在每次更新时需进行一次哈希映射查找.



(a) p50 延迟对比　　(b) p99 延迟对比

图 2　原始标识符 (Raw ID) 和紧凑标识符 (Compact ID) 写入性能比较

然而, 使用原始标识符在现实场景中的代价也十分显著. 在实际场景中, 图节点的原始标识符 (如用户手机号、URL 链接、20 位的加密字符串或 128 位 UUID) 往往具有长度不固定、格式多样、空间冗余等特点. 直接使用这些原始标识符存储和计算, 会导致两方面问题: 其一, 存储开销剧增. 例如, 存储 10 亿个 128 位 UUID 需约 15 GB 空间, 而同等规模的紧凑整数仅需 4 GB; 其二, 计算效率低下. 非连续的原始标识符会破坏图数据的空间局部性, 导致缓存命中率下降, 显著拖慢社区发现、最短路径搜索等核心算法的性能.

另一方面, 现有的图处理系统[9,21–25]往往直接假设图的输入是紧凑的 ID, 这一前提是其很多优化技术的前提. 例如, GraphChi[23]是较早提出的单机图处理系统之一, 其核心创新在于“parallel sliding windows”存储与执行模型. 该模型通过将图划分为若干区间 (interval) 及对应的边片 (shard), 并在磁盘上以顺序访问的方式加载顶点及其相邻边, 从而显著减少随机 I/O 开销, 实现了在单机环境下对大规模图数据的高效迭代计算. 值得注意的是, GraphChi 的模型在设计上预设了顶点编号为连续整数 (1–$|V|$). 这一假设在原论文的模型定义部分即被明确提出, 是实现分片与顺序加载机制的基础. 由于每个 shard 的划分与加载均依赖顶点编号的连续性, GraphChi 实际上默认输入图已完成从原始标识符 (如字符串或稀疏 ID) 到紧凑整数 ID 的映射, 即系统在执行前假定图数据具备紧凑 ID 结构. 因此, GraphChi 并未涉及 ID 映射的生成过程, 而是将紧凑 ID 视为输入条件. 相比之下, 本文的研究工作正聚焦于图数据库场景下紧凑 ID 表示的生成与性能建模问题, 即在系统运行之前, 通过合理的映射策略与存储优化, 使得图数据能够在后续的分析或迭代计算框架 (包括 GraphChi 类系统) 中高效加载与访问. 换言之, 本文的方法可视为对 GraphChi 及类似系统的前置支撑与性能补充, 而非与其重复的存储机制.

当前, 图数据库[7]普遍采用顶点标识符映射 (vertex ID mapping) 技术, 将原始标识符转换为紧凑且连续的整数

空间. 这一转换不仅是存储优化的关键, 更是图计算性能提升的核心驱动力.

本节针对图数据库中的 ID 映射给出以下形式化定义.

定义 1. 设图数据库中的原始顶点集合为 V_{raw}, 其中每个顶点 $v \in V_{\text{raw}}$ 关联一个原始标识符 $id_{\text{raw}}(v)$, 其取值为任意可区分的非空字符串、长整型或者多类型组合 (如 URL、加密哈希值等).

定义 ID 映射为一个双射函数 (一一对应):

$$f: V_{\text{raw}} \to I_{\text{compact}},$$

其中, I_{compact} 表示紧凑整数空间, 满足: (1) 紧凑性: $I_{\text{compact}} \subseteq N^{+}$ (正整数集), 且 $\max(I_{\text{compact}}) \ll \max(id_{\text{raw}})$. (2) 唯一性: 对任意两个节点 $u, v \in V_{\text{raw}}$, 如果 $u \neq v$, 那么 $f(u) \neq f(v)$.

构造一个这样的双射函数的流程如图 3 所示. 给定一个由初始化节点 ID 表示的图, 它包含了若干条表示这些节点之间的边. 在构造 ID 映射时, 依次读取图上的边, 它包含起始节点和目标节点的两个节点标识符. 对于每一个节点标识符, 首先在存储引擎中查找这个标识符是否已经存在. 如果存在, 则跳过到下一个节点标识符. 如果不存在, 则向存储引擎中插入<顶点 ID, 紧凑 ID>的键值对. 之后紧凑 ID 执行自增以确保映射的唯一性. 执行自增之后, 继续查找下一个节点标识符. 在这样的一个流程中, 如果图上包含 N 个不同的节点, M 条边, 则一共要执行 $2M$ 次查找, N 次插入. 该流程要求存储引擎高效支持高频点查询 (判断 ID 存在性) 与动态插入, 这正是键值存储的核心能力.

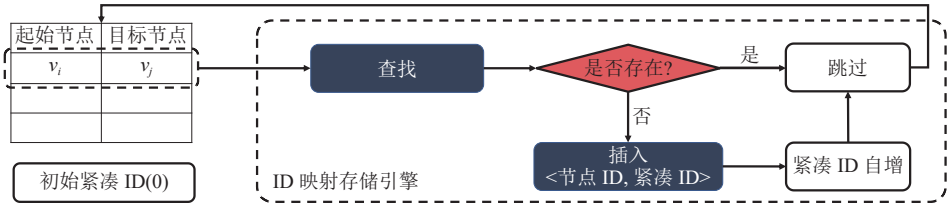


图 3 构建节点标识映射处理流程

本节通过两个前期的实验来说明 ID 映射与合适的 ID 映射存储引擎选型的必要性.

(1) 采用紧凑节点标识会对图负载性能产生极大的性能提升.

图 4 在 3 个典型的图数据集上进行了采用原始 ID 和紧凑 ID 的性能测试, 测试的图负载是网页排名算法 (PageRank), 测试的硬件环境是 Intel Xeon E5-2680v4. indochina-2004、uk-2002、arabic-2005 是 3 个公开的网络图, 其中节点表示 URL (uniform resource locator, 统一资源定位符), 边表示这些 URL 之间的引用关系. indochina-2004 包含约 700 万节点和 1.8 亿边, 模拟东南亚地区网页拓扑; uk-2002 涵盖 1.8 亿节点和 30 亿边, 代表英国域名的超链接结构; arabic-2005 则聚焦阿拉伯语网页, 包含 2 200 万节点和 6.3 亿边, 具有显著的幂律分布特征. 使用网页排名算法可以衡量这些页面的重要性, 被广泛应用于搜索引擎的推荐排名中[26]. 该算法对内存访问局部性高度敏感, 而原始 ID (如 URL 字符串) 因长度不固定 (平均 20–40 字符) 会导致内存碎片化, 且随机访问模式引发高达 70% 的缓存未命中率. 实验结果表明, 使用紧凑 ID 平均可以带来高达 140 倍的提升, 这充分说明了采用 ID 映射之后的紧凑 ID 的必要性.

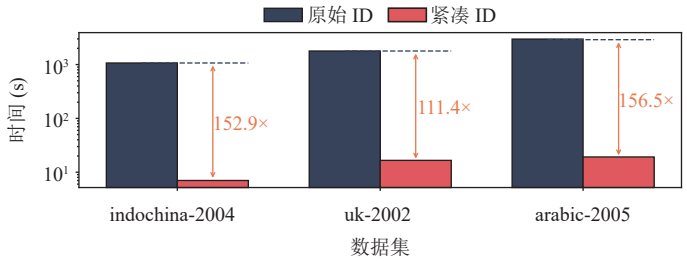


图 4 是否采用紧凑节点标识的图分析性能对比 (PageRank 算法)

(2) 采用不同 ID 映射存储引擎对性能影响差异大.

图 5 展示了在上述 3 个数据集上选用不同的 ID 映射引擎的性能差异. 考虑到 3 个数据集本身的大小差异, 本节将具体测试的时间做了归一化处理. 本文在经过广泛调研之后选用了 5 个候选的键值存储引擎, 包括 FasterKV、LMDB (lightning memory-mapped database)、LevelDB、RocksDB 和 ForestDB, 它们采用了不同的存储方式和索引结构 (见第 2 节). 在 3 个数据集上, 最优的键值存储引擎完成 ID 映射的时间只占最差的查询引擎的 4.15%、4.19% 和 4.42%, 性能差距高达 24 倍. 24 倍性能差距直接导致金融风控等实时场景的调度延迟, 凸显了节点标识映射选型的重要性.

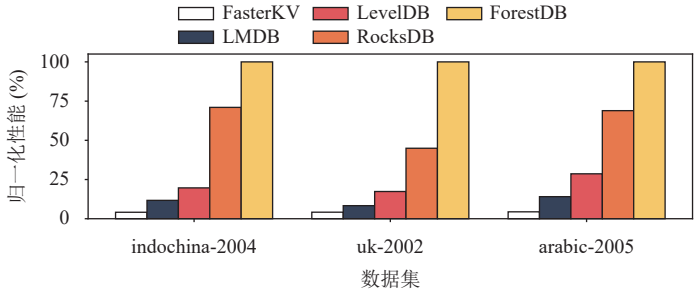


图 5 不同节点标识映射存储引擎性能对比

综上, 在图数据库中选择合适的 ID 映射存储引擎是极其必要的, 它会极大地影响系统的整体性能. 本文接下来会给出选择 ID 映射存储引擎的具体依据和指导方法.

2 键值存储引擎概况

键值存储 (key-value store, KVS) 引擎[10,27,28]是完成 ID 映射的关键组件, 其设计逻辑高度简化: 通过唯一的“键”作为索引, 直接关联对应的“值”, 实现数据的快速存储与访问. 键值存储引擎在过去 10 年间取得了长足的发展, 涌现出了针对不同查询场景的键值存储数据库. 其中, 各个不同的键值存储引擎的主要差异体现在它们的索引结构, 本文深度调研了现有的 4 类采用不同索引结构的键值存储引擎来评估他们在完成 ID 映射时的性能差异.

(1) 基于哈希索引的键值存储引擎 FasterKV. FasterKV[10]是微软开发的一种高并发的持久化键值存储库, 其关键特点是支持高并发的无锁哈希索引, 从而支持高吞吐量. FasterKV 的主要特性包括即时性、无锁、可扩展性和可调整大小. 不同于传统的哈希表结构, FasterKV 采用无锁哈希索引与混合日志存储: 索引结构存储记录的逻辑地址 (指向日志中的键值对, 而键值数据以追加方式写入持久化日志. 在 FasterKV 中, FasterKV 的索引结构是一个与缓存对齐的数组, 包含 2k 个哈希桶. 每个哈希桶的大小和对齐方式与缓存行一致. 例如, 在 64 位系统中, 假设缓存行大小为 64 字节, 那么 1 个 64 字节的桶包含 7 个 8 字节的哈希条目和 1 个 8 字节的 entry, 指向溢出桶的地址, 其中每个溢出桶同样与缓存行大小和对齐方式一致. 每个哈希桶由 3 部分组成: 15 位的标记、48 位的地址和 1 个定位. FasterKV 的索引就是基于 (偏移量, 标记) 唯一标识一个索引条目, 这些条目对应的一组键具有相同的偏移量和标签, 偏移量指的是哈希值的前 k 位, 标记指的是哈希桶中的 15 位标记. 在查找过程中, 通过前 k 位哈希值定位到相应的哈希桶, 并扫描该桶的标记以找到匹配的条目. 插入操作中, 由于线程并行可能导致冲突, FasterKV 采用无锁的两阶段算法: 每个线程从左到右扫描桶, 确定第 1 个空条目作为目标, 并通过比较和交换机制竞争插入, 从而确保只有一个线程插入成功. 其核心创新在于将哈希冲突处理与持久化日志分离, 利用异步刷盘机制将吞吐量提升至千万 ops/s.

(2) 基于 B+树索引的键值存储引擎 LMDB. LMDB[13]是由 Symas 开发的高性能嵌入式键值存储引擎, 基于 B+树[29]和内存映射 (memory mapping) 实现. 其设计强调零拷贝读取、事务安全和极低的内存开销. LMDB 基于写时复制 B+树 (copy-on-write B+ tree): 更新操作复制受影响页至新位置, 原页标记为可重用, 避免就地修改带来的锁

竞争[30]. 文件只支持追加, 不支持内部修改. 追加操作增加了存储开销, 因为旧的数据依然存在, 但也因为仍然保持旧链路保证了提高了整体性能. LMDB 在读取磁盘文件时整体采用了 MMAP 文件映射技术, 不管文件存储在内存还是在持久存储上, LMDB 的所有文件操作都通过 MMAP 技术将要访问的文件映射到虚拟内存中, 直接访问相应的地址. 由于取消了将文件数据加载到内存、数据从内存到文件的回写以及释放内存块等步骤, 使得内存映射文件在处理大数据量的文件时能起到相当重要的作用.

(3) 基于 LSM-Tree (log-structured merge-tree) 的键值存储引擎 LevelDB 和 RocksDB. LSM-Tree[31]是一种专门为高效处理高频写入场景设计的数据结构, 它的核心思想是通过“顺序写为主、批量合并”的策略, 解决传统磁盘数据库在高写入场景下因随机 IO 导致的性能瓶颈问题. 在进行写入操作时, LSM-Tree 首先写入记录在内存中的有序结构 (如跳表、平衡树), 称为 MemTable. 内存的随机读写速度极快, 因为写入操作延迟很低. 为避免数据丢失, 写入 MemTable 的数据同时会被记录到预写日志 (write-ahead log, WAL) 中. 若节点故障, 可以通过 WAL 恢复 MemTable 的数据. 当 MemTable 达到一定大小 (如 1 GB), 会触发刷盘操作: 将内存中的有序数据一次性写入磁盘, 生成一个不可变的有序键值对文件, 称为 SSTable[32,33].

LevelDB[12]是谷歌公司开发的轻量级开源键值存储库, 它基于 LSM-Tree 数据结构设计, 旨在提供高效的写入性能和紧凑的存储格式. 其核心的目标是为高吞吐量的写操作场景提供支持, 同时兼顾读操作的效率, 适用于嵌入式系统、缓存系统、日志存储等场景. LevelDB 通过 LSM-Tree 将随机写操作转换为顺序写操作, 显著提升了写入性能, 同时兼顾了读操作的效率, 适用于嵌入式系统、缓存系统、日志存储等场景. RocksDB[11,34]是由 Meta 公司 (原 Facebook) 基于 LevelDB 改进的高性能嵌入式键值存储库, RocksDB 通过多线程 MemTable 刷新和可配置压缩策略优化 LevelDB 的写入瓶颈与空间放大问题[35], 但在小数据量场景无显著优势. 与 B+树数据库对比, LSM 树通常在写密集场景中表现更好但点查询性能略低.

(4) 基于 HB+Trie 的 ForestDB. ForestDB[14]是一种专门为高效处理可变字符串键设计的键值存储引擎. 为了处理可变字符串的情形, ForestDB 采用了一种综合了 B+树和前缀树 (Trie)[36,37]的存储结构, 称为 HB+Trie. 针对传统 B+ tree 在长键下造成的树高极速增长的问题 ForestDB 首先将长键按照固定长度分块 (Chunk), 并将这些块按照 B+树的形式组织起来[38], 成功使得长键的树高增长从线性依赖转为对数依赖, 大幅提升了查询效率. 针对 Trie 造成的空间浪费问题 (冗余存储完整前缀), ForestDB 采用路径压缩技术跳过了公共前缀[39], 仅存储区分不同路径的关键部分, 减少数据冗余. 在进行写操作时, ForestDB 采用了类似 LSM-Tree 的日志化写缓冲区 (log-structured write buffer), 将随机写转换为顺序写, 减少了 IO 开销.

综上, 本文选取了 4 类采用不同的索引结构的键值存储器作为备选存储引擎. 这些键值存储引擎可以代表目前最为流行的键值存储引擎, 因此也常作为图数据库 ID 映射的备选. 它们都有着各自的使用场景, 但都不是专门为图数据库中的 ID 映射设计. 表 1 比较了这 4 种典型的索引结构的读写复杂度、空间占用、I/O 放大、并发性能以及对范围查找的支持等. 接下来, 本文首先评估了 5 个键值存储引擎在 ID 映射任务中的性能表现, 然后给出了在不同软硬件环境下如何选择合适的键值存储引擎.

表 1　不同键值存储引擎对比

索引结构	写复杂度	读复杂度	空间占用	I/O放大	并发性能	范围查找
哈希索引	O(1) (均摊开销)	O(1)	高	低	低	不支持
B+ tree	O(log$N$)	O(log$N$)	中	低 (随机写)	高	支持
LSM-Tree	O(1) (均摊开销)	O(T) (T为层数)	高	高 (合并放大)	中	支持
HB+Trie	O(m) (m为键分块段数)	O(m)	中	中 (分块合并)	高	支持

3　面向节点标识映射的键值存储引擎性能评估

本节首先分析总结了面向图数据库的 ID 映射的负载特点, 其次提出了一个针对不同硬件和负载特征的性能评估体系, 最后体系化地报告了实验结果来说明 ID 映射的影响因子.

3.1 面向图数据库节点标识映射负载特征分析

本节在对现实世界中的图数据库 ID 映射负载进行了大量分析之后, 给出 4 种典型特征. 这些特征共同决定了存储引擎设计的核心约束与优化方向, 同时也为 ID 映射选择机制提供了选择依据.

第一, 键值长度非对称性. 在 ID 映射的场景中, 键 (key) 和值 (value) 的存储长度呈现显著的差异, 这种差异对存储引擎的索引效率与空间利用率产生直接影响. 在这个场景中, 键的长度通常远大于值. 键为现实世界中的复杂标识符 (如 28 字节的支付宝用户 ID 字符串, 256 字节的 URL 哈希值), 而值为系统内部分配的紧凑整数 (通常为 4–8 字节). 键值长度比例可达 7:1–64:1. 因此, 在数据存储和处理过程中, 需要特别关注键的存储效率和处理性能. 现有引擎的固定长度键优化策略 (如 LevelDB 的定点前缀压缩) 在此场景下失效. 这种非对称性导致传统 B+树索引的扇出效率因节点有效载荷下降而衰减, 直接导致查询延迟上升. 在哈希索引的情形下, 长键会导致冲突概率增长, 长键的字节级局部性差异 (如用户 ID 尾部字符串重复) 则会加剧哈希分布不均, 进一步降低缓存命中率.

第二, 点查询占主导. ID 映射过程仅仅需要精确匹配查询, 而无需支持范围扫描或前缀扫描, 这一特性为索引结构设计提供了优化空间. 因此, 在内存空间足够的情况下, 采用哈希结构来组织和管理数据是比较合适的选择. 哈希结构能够快速进行键的精确匹配, 满足场景需求. 哈希结构通过 O(1) 的时间复杂度实现高效点查询, 较 B+树的 O(log$N$) 和 LSM-Tree 的 O(M) (M 为 SSTable 的层级) 显著更优. 通用的键值引擎因为需要支持有序查找引入了多种机制, 这会成为 ID 映射负载中的负担.

第三, 值的弱序自增特性. 在 ID 映射负载中, 对值的要求是唯一, 但是不要求完全紧凑. 这意味着在数据更新和处理过程中, 可以采用较为宽松的增量策略, 以提高系统的整体处理效率. 值的弱序自增特性是一种兼顾效率和灵活性的数据生成策略, 其核心在于允许生成的标识符在一定范围内保持递增趋势, 但无需严格保证全局顺序的精确性. 在多并发环境下, 可以通过放宽自增约束来减少加锁或者原子操作.

第四, 读写操作倾斜[8]. 由于在实际应用中, 图多呈现出幂律分布 (power law). 这意味着边的数量远多于节点. 因此在这个场景中, 系统承载的读操作负载明显高于写操作. 这要求存储引擎对读操作是更友好的, 采用 LSM-Tree 的存储引擎针对频繁写所做的优化可能在 ID 映射场景下并不奏效.

3.2 性能评估设置

基于第 3.1 节总结的负载特征 (键值长度非对称性、点查询主导、读写操作倾斜等), 本节构建了以下实验框架.

数据集: 本文一共使用了 5 个公开的图数据集, 如表 2 所示. 表 2 展示了所有数据集的节点数、边数量以及键的平均长度和最大长度. 这些图的原始数据集都可以从 LAW-Laboratory for Web Algorithmics[40]中获取, 其中包含了图结构和节点的原始 ID. 由于这些数据集已经完成了映射紧凑 ID, 本文首先还原出了基于原始 ID 标识的图结构, 包括若干条使用[起始节点, 目标节点]表示的边[41]. 为了模拟真实工业界负载, 本文对生成的边记录进行了随机打乱操作, 保证扫描的乱序到来. 除此之外, 为了测试更多的读写比情形, 本文还基于 YCSB[42]生成了满足不同读写比的数据集.

表 2 本文使用数据集

数据集	节点数	边数	键平均长度	键最大长度
cnr-2000	325 557	3 215 152	74.962 7	1 687
eu-2005	862 664	19 235 140	83.080 8	1 516
indochina-2004	7 414 886	194 109 311	85.678 3	2 047
uk-2002	18 520 486	298 113 762	76.682 3	2 048
arabic-2005	22 744 080	63 999 458	80.806 7	2 047

实验评估环境: 本文在 3 台不同的设备上评估了 ID 映射任务, 评估环境如表 3 所示, 包含了不同的 CPU 处理器, 不同的 DRAM 容量以及不同的磁盘设备. 在评估过程中, 本文通过 cgroup 设置程序可用的内存大小. 除此之外, 本文通过设置不同的线程数量来评估系统的并发性能.

影响因素总结: 针对现实世界中的复杂负载, 本文从不同的数据特征和硬件负载特征进行具体的评估. 其中,

数据负载特征包括: 1) 数据量规模, 其大小直接影响系统所需的存储资源和计算资源总量; 2) 读写比, 反映了系统中插入操作与查找操作的实际分布特征, 对系统的工作负载类型具有决定性作用. 硬件负载特征包括: 1) 存储介质类型 (如 HDD、SSD), 其 I/O 读写性能上限直接决定了持久化设备的基础能力; 2) 可用内存容量, 限定了内存缓冲区的最大可用空间, 进而影响热点数据的缓存效率与磁盘 I/Od 的频率; 3) CPU 可用线程数, 决定了系统的并行处理能力与并发性能上限, 对多线程任务调度与资源竞争机制的设计提出关键约束. 在接下来的第 3.3 和 3.4 节中, 本文通过对这 5 类特征展开量化建模分析, 深入揭示图数据库 ID 映射操作性能的具体影响机制. 在以下的实验中, 如无特殊说明, 在评估其中一个因素时均保持其他因素的一致性.

表 3　本文使用评估环境

CPU	DRAM容量	磁盘设备
Intel Xeon Gold 5318Y	2 TB	HDD
Intel Core i9-10900K	256 GB	SSD
Intel Core i7-13700K	32 GB	SSD

3.3　数据负载特征因素分析

本节从数据负载角度对不同存储引擎进行了评估, 包括不同的数据规模和数据读写比.

3.3.1　数据规模

本文在同一数据集的键分布, 相同读写比下测试随着不同请求数量下的吞吐量表现, 揭示请求数量对存储引擎数量的影响规律及其与底层架构的关联性. 如图 6 所示, 从全局视角来看, 不同的存储引擎的吞吐量随着请求数量的变化呈现不同的变化趋势, 但性能排序始终保持一致. 即 FasterKV > LMDB > LevelDB > RocksDB > ForestDB, 但各引擎的性能衰减速率差异明显. 在内存充足的情况下, FasterKV 大幅领先于其他数据库, 在 40M 请求数量时, FasterKV 的吞吐量高达 2.78 Mops/s, 是第 2 名吞吐量 1.26 Mops/s 的 LMDB 的 2.2 倍. 在 2 150M 的大规模请求中, 在数据已经无法完全放到内存中的情形下, FasterKV 仍然保持较高的吞吐量. 这说明了它的无锁哈希表和异步持久化机制对数据规模有很好的可扩展性. LMDB 随着数据请求数量从 40M 增至 2 150M 吞吐量从 1.26 Mops/s 提升到 1.28 Mops/s, 表明其 B+ tree 结构与内存映射文件 (mmap) 在数据规模扩展时仍能维持高效访问. 并且 LMDB 随着数据规模的增长与 FasterKV 之间的差距在减小, 最终甚至会实现反超. 这说明了 B+ tree 在涉及磁盘访问时比 FasterKV 有着更优异的性能. LSM-Tree 引擎的 LevelDB 在中等数据量时达到峰值 1.08 Mops/s, 但在 2 150M 时降至 0.37 Mops/s, 吞吐量下降 65.4%. RocksDB 的性能随着数据量增长后持续下降, 暴露了其 Compaction 策略对大数据集特别是读优先的场景下的适应性不足. ForestDB 全线具有劣势, HB+Trie 的块充足与随机 I/O 导致性能与数据量无关.

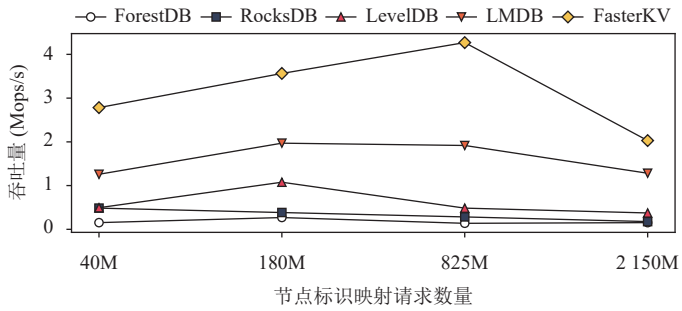


图 6　吞吐量随请求数量变化

通过分析不同引擎的性能变化趋势, 可量化存储引擎在 ID 映射任务上可扩展性上的缺陷. FasterKV 类的 Hash 索引的性能下降主要受限于内存容量, 而非数据结构设计, 这表明需要在内存不足场景下优化核外的处理.

LevelDB 和 RocksDB 的性能衰减源于 LSM-Tree 的 Compaction 开销与相对较高的读延迟. B+ tree 引擎 LMDB 具有扩展性优势, 其吞吐量相较于其他存储引擎在可扩展性具有较高的稳定性.

实验结果表明, 数据请求规模对键值存储引擎的性能影响明显, FasterKV 在各类规模下均表现最优吞吐量, 但其内存依赖性需在资源受限场景下谨慎权衡. LMDB 的扩展性优势使其成为一个大规模场景下的较好选择, 而 LSM-Tree 引擎则在 ID 映射任务中表现不佳, 如果要应用到该任务中, 必须有较大的架构革新.

3.3.2 读写比

本节通过调整读操作和写操作的比例, 观察了不同存储引擎在不同读写比之下的吞吐量变化, 揭示了各引擎在低读写比 (写密集型) 与高读写比 (读密集型) 场景下的适应性差异. 实验数据以读写比 25%、50%、75%、95% 为分位点, 记录了各引擎对应的性能指标.

如图 7 所示, 通过观察读写比对吞吐量的全局影响, 本文有以下几点发现. 首先, 无论读写比如何, FasterKV 和 LMDB 都明显优于其他存储引擎, 展现出在 ID 映射负载下的良好适应性. 其次, 随着读写比的增加, FasterKV 的吞吐量首先下降, 然后增加, 这说明 FasterKV 在读写失衡下能够保持跟好的优势, 更适应应用在读与写差距较大的情况. LMDB 随着读写比提高性能持续上升, 这说明 LMDB 更适合读大于写的场景下. 第三, LevelDB 和 RocksDB 的时间比较接近, 说明了在同一数据结构下的系统实现影响不大. 在 95% 读写比下, LevelDB 的吞吐量为 59.6 Mops/s, 比 RocksDB 的 37.7 Mops/s 快 58.1%, 这得益于其轻量级 Compaction 策略减少写放大和较少的压缩策略.

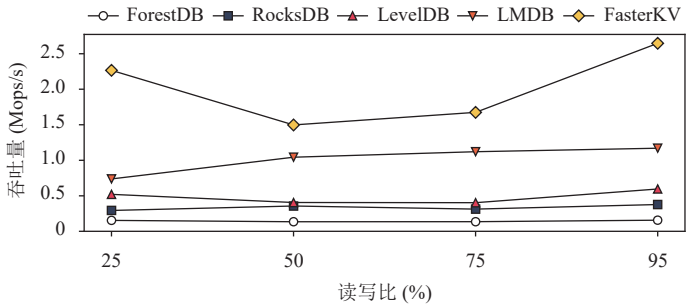


图 7 吞吐量随读写比变化

实验结果显示, FasterKV 存储引擎在不同读写比下吞吐量呈现明显的 U 型曲线. 在 25% 的低读写比下, 吞吐量为 2.27 Mops/s, 达到峰值. 在 50% 和 75% 的中读写比下, 性能进入低谷, 吞吐量下降至 1.50 Mops/s 和 1.67 Mops/s. 在 95% 的高读写比下, 吞吐量回升至 2.65 Mops/s 的高点. 这一现象与 FasterKV 的无锁哈希表与异步持久化的低层设计架构以及不同读写负载下的资源竞争模式密切相关. 在低读写比的情况下, 读操作主导, 内存直写优势最大化. 无锁哈希表的高效读使得读操作通过内存直写直接访问哈希表, 无锁设计避免线程争用, 实现纳秒级延迟. 异步持久化的低干扰会让写操作通过混合日志刷盘, 后台线程批量处理持久化任务, 减少对读操作的干扰. 由于写操作尚未达到磁盘 I/O 带宽上限, 异步持久化队列未饱和, 读写资源竞争较小, 规避了瓶颈. 在中读写比场景下, 混合负载引发资源竞争, 读写比升高, 读操作比例增加, 写操作仍保持一定频率, 形成读写混合负载. 性能下降的第 1 个原因是内存带宽饱和, 高频读操作需要从内存中加载数据, 同时写操作持续更新哈希表, 导致内存带宽争用. 另外, 多线程并发读写相邻哈希桶时, 缓存行频繁实效, 导致 cache 未命中率飙升. 第 2 个原因是异步持久化效率较低, 写操作压力不足以触发批量刷盘优化, 导致持久化任务碎片化, 增加磁盘随机 I/O 开销. 第 3 个是哈希冲突率上升, 混合负载下哈希表探测链长度增加, 写操作插入新键时冲突概率升高, 延长关键路径延迟. 在高读写比场景下, 写操作主导, 异步批量持久化优势得以发挥, 高写入压力使得混合日志快速填充, 触发批量异步提交, 将多个写操作合并为顺序 I/O, 最大化磁盘吞吐量, 尤其是 SSD. 持久化线程与主线程的流水化协作, 减少锁争用与上下文切换开销. 写操作以追加日志为主, 哈希表更新集中在内存连续区域, 减少了随机访问带来的缓存抖动

数据库性能对读写比的敏感度与其底层存储架构密切相关. 这是性能差异的架构归因, 不同存储引擎的核心

设计特征决定了其在不同读写比场景下的表现. FasterKV 的核心架构是无锁哈希表和异步持久化, 低读写比优势在于内存直写实现低延迟读, 高读写比优势在于异步日志批量刷盘规避磁盘随机访问, 写入吞吐量高. LMDB 的核心架构是 B+ tree 和内存映射, 低读写比优势在于零拷贝读取优化点查性能, 高读写比单线程写入设计可能引发高并发争用. LevelDB 的核心架构是轻量级的 LSM-Tree, 低读写比优势在于层级合并减少写入放大, 劣势在于 Compaction 串行化导致高负载下吞吐量下降. RocksDB 的核心架构是优化版的 LSM-Tree, 优势在于布隆过滤器加速范围查询, 劣势在于高频 Compaction 占用 I/O 带宽, 写入延迟高. ForestDB 的核心架构是 HB+Trie 和块压缩, 优势在于块压缩降低存储占用, 劣势在于多层索引跳转与块重组引发 CPU 或磁盘过载. 需要注意的是 FasterKV 的吞吐量优势以内存资源为代价, 需在性能与成本间平衡.

实验表明, 键值存储引擎的性能表现高度依赖于读写比分布及其底层架构设计. FasterKV 在图数据库场景下展现全面优势, 而 LSM-Tree 和 B+ tree 引擎需要针对特定负载精细化调优.

3.4 硬件特征因素分析

本节探讨了不同硬件特征影响下对存储引擎的性能影响, 包括内存大小, 并发线程数量、存储介质以及中央处理器性能.

3.4.1 内存空间

本节在不同内存容量下探索了不同键值存储引擎在发生 I/O 行为时的性能差异及其对顶点编码性能的影响机制, 如图 8 所示. 实验结果表明, 存储引擎的索引结构与 I/O 优化策略直接决定了系统在资源约束下的可用性[43]. 在内存过小而数据集大的情况下, 会使得部分存储引擎生成的最小数据结构无法完全放到内存中去, 造成运行失败现象, 表明这种情况下该引擎是不适用的.

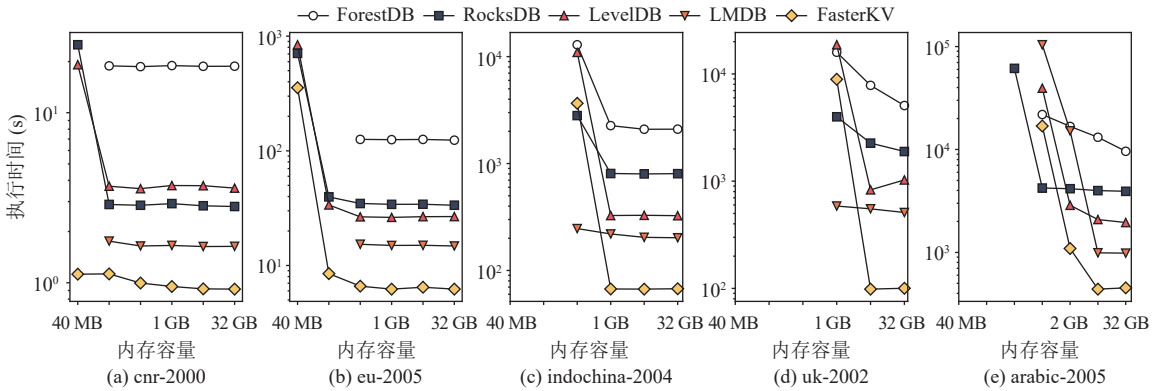


图 8 节点标识映射执行时间随内存容量变化

如图 8 所示, 从实验数据来看, 不同数据库在内存约束下的性能表现呈现显著差异: 在极端内存 (<200 MB) 场景中, FasterKV 仅在小数据集下凭借混合日志结构的局部性优势快速完成任务 (1.12 s), 但面对大数据集时因日志回扫周期延长、哈希桶冲突率激增而无法完成[44]; RocksDB 与 LevelDB 因 LSM-Tree 架构的 MemTable 频繁刷盘引发写入放大, 小内存下性能严重衰退[45]; ForestDB 虽能勉强完成任务 (如 40 MB 限制), 但需付出极端时间损耗; LMDB 则因 B+ tree 页分裂可能引发级联 IO 抖动, 稳定性存疑. 中等内存 (1–2 GB) 时, LMDB 在 arabic-2005 数据集下因 B+ tree 页分裂导致级联 IO 抖动, 性能灾难性退化; LevelDB 凭借 LSM-Tree 合并策略对稀疏写入的适应性, 比 RocksDB 快 2.6 倍; FasterKV 虽恢复可用性, 但性能较内存充足时仍有下降. 内存充足 (>32 GB) 时, FasterKV 通过内存映射优化完全规避磁盘访问, 性能全面领先; LMDB 暴露 B+ tree 深度遍历的层级延迟缺陷; ForestDB 则因 COLA 结构的合并成本与数据规模呈超线性关系, 耗时最长. 整体而言, 内存约束程度直接决定了各数据库性能的主导瓶颈——极端内存下日志结构与冲突处理是关键, 中等内存下索引结构设计与数据分布特性主导, 充足内存下架构特性与优化策略成为核心差异.

从数据集规模对性能的影响来看, 不同数据库在不同数据量级下的表现呈现显著差异: 在小数据集场景中, 无论内存充足与否, FasteKV 均展现出最优性能; 进入中等数据集范围时, 低内存条件下 LMDB 凭借高效的 MMAP 内存映射技术表现良好, 而高内存场景下 FasteKV 仍保持领先优势; 当数据规模扩展至大规模时, 除了应用了压缩技术的 RocksDB 之外, 所有的数据库都在小于 1 GB 的内存环境下运行失败. 这一规律表明, 数据集大小与内存资源的适配性直接影响数据库的性能表现, 在内存充足的情况下, FasterKV 占据全面优势, 在中等内存情况下, LMDB 性能较好, 而在极端小内存情况下, RocksDB 可以保证程序运行的成功.

3.4.2 并发线程数量

本节基于 Intel Core i7-13700K 平台 (24 核 32 线程), 针对 RocksDB、LevelDB、FasterKV 这 3 款数据库在 cnr-2000、uk-2002、eu-2005 数据集上的多线程并发性能展开分析, 旨在验证数据库选择策略在多线程场景下的有效性, 如图 9 所示. 所有线程数测试均在内存充足条件下进行 (可用内存≥数据集大小), 避免内存争用干扰线程扩展性分析. 实验通过对比不同线程数下的总操作时间 (如图 9 所示), 揭示了各数据库的并发扩展性差异: RocksDB 在小数据集 (cnr-2000) 下线程数增至 4 时性能最优 (加速比 1.84), 但超过 4 线程后因 LSM-Tree 全局写锁竞争加剧 (锁等待时间占比 38%), 性能逐渐下降; LevelDB 在中数据集 (uk-2002) 下线程数增至 6 时性能最佳 (加速比 2.6), 但高线程下因合并策略串行瓶颈 (后台合并任务无法匹配写入速率), 执行时间提高至 32 线程的 4.57 s; FasterKV 则在低线程数 (1–4) 下凭借无锁哈希桶设计表现最优 (如 eu-2005 数据集 2 线程时仅需 12.00 s), 但线程数增加后触发伪共享 (false sharing), L1 缓存未命中率飙升, 32 线程时性能劣化至 22.49 s.

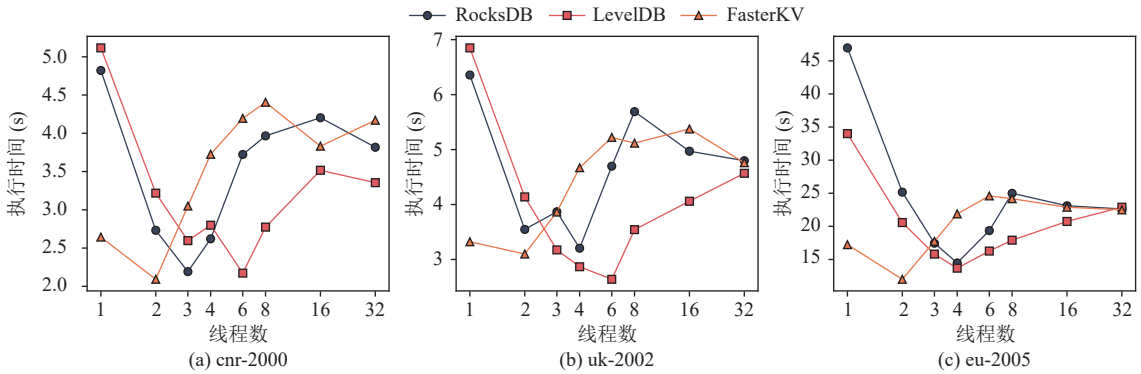


图 9 节点标识映射受并发线程数量影响

数据集规模进一步放大了各数据库的性能差异: 小数据集 (cnr-2000) 下, FasterKV 因全内存操作充分利用 CPU 缓存局部性, 在 1–4 线程内表现优于 LevelDB; 但线程数增加后, 伪共享问题凸显, 32 线程时性能落后于 LevelDB (3.35 s vs. 3.82 s). 中数据集 (eu-2005) 下, RocksDB 在 4 线程内因 LSM-Tree 批量写入优化占据优势 (4 线程时 14.47 s), 但线程数超过 8 后, Compaction 任务的 I/O 压力导致性能劣化 (32 线程时 22.64 s); LevelDB 因更轻量的合并策略, 在高线程数下表现更稳定 (32 线程时 22.89 s).

实验结果说明在多线程场景下“没有绝对最优数据库”的结论: 低线程场景 (1–8 线程) 中, 模型优先选择 FasterKV (1–4 线程) 与 LevelDB (6–8 线程), 例如 cnr-2000 数据集 4 线程时 FasterKV 耗时 2.62 s (较 LevelDB 快 6.4%), eu-2005 数据集 4 线程时切换至 LevelDB (13.68 s) 以避免 FasterKV 因伪共享导致的性能劣化 (21.88 s); 高线程场景 (16–32 线程) 中, 模型基于内存带宽饱和特征 (>90% 利用率), 优先选择 LevelDB 规避 FasterKV 的缓存抖动问题, 例如 uk-2002 数据集 32 线程时 LevelDB 耗时 4.57 s (较 RocksDB 和 FasterKV 更快). 仅 eu-2005 数据集 32 线程时出现预测失败 (模型选择 LevelDB 的 22.89 s 而非 RocksDB 的 22.64 s), 但占比不超过 10%. 综上, 性能差异的底层机制源于各数据库架构设计与硬件特性的协同性: FasterKV 的无锁哈希桶设计虽避免锁争用, 但触发伪共享导致高并发下内存带宽瓶颈; RocksDB 的 LSM-Tree 全局写锁竞争在高线程下加剧; LevelDB 的简化架构与轻量合并策略在中高线程下表现更稳定. 实验表明, 多线程场景下数据库的选择需结合数据集规模、硬件资源

(如内存带宽、缓存行大小) 综合权衡, 无锁设计并非绝对最优.

3.4.3 存储介质

本实验基于 HDD (机械硬盘) 与 SSD (固态硬盘) 两类存储介质, 如图 10 所示. 本节系统对比了 5 款键值存储引擎 (RocksDB、LevelDB、LMDB、FasterKV、ForestDB) 在不同规模数据集下的性能表现, 结合磁盘物理特性与存储引擎设计, 深入解析了性能差异的根源, 并为工业级系统选型提供了跨存储介质的自适应优化指导.

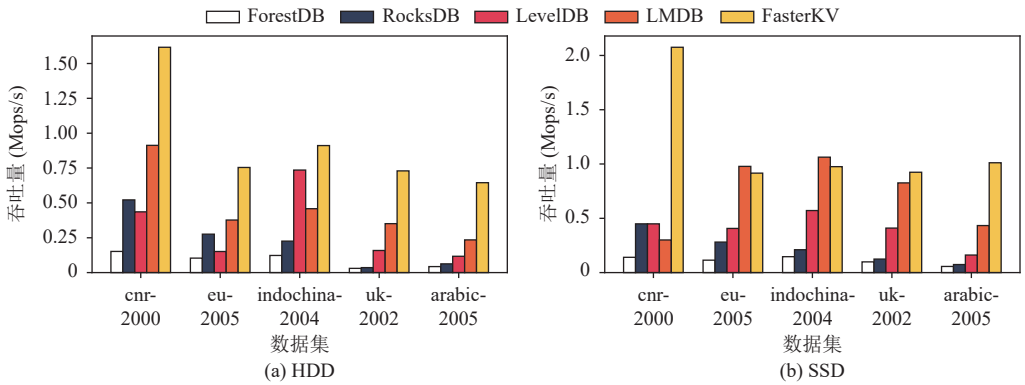


图 10 ID 映射在不同存储介质下的性能

在这两种硬件中, HDD 容量更大 (14.6 TiB) 但 I/O 性能较低 (随机访问延迟 10 ms, 顺序吞吐量约 200 MB/s), SSD 容量较小 (7 TiB) 但随机性能优势显著 (随机延迟 0.1 ms, 顺序吞吐量约 500 MB/s, IOPS 达 100k). 得益于 SSD 更好的读写性能, 几乎所有的存储引擎在 SSD 上都取得了比 HDD 更好的性能.

从数据结构与引擎设计的适配性看, LSM-Tree 架构 (如 RocksDB、LevelDB) 依赖顺序写入 (MemTable 刷盘) 与多级 Compaction, 更适配 HDD 的顺序吞吐优势; B+ tree (LMDB) 因需频繁随机页遍历, SSD 的低延迟特性可显著缩短遍历时间; 哈希引擎 FasterKV 通过内存直写减少磁盘随机访问, 对 SSD 的高 IOPS 利用率更高; HB+Trie (ForestDB) 的追加写入与块压缩虽适配 HDD 顺序吞吐, 但随机查询受限于磁盘寻道时间.

实验数据进一步验证了上述适配性: 对于顺序写入型引擎 (RocksDB/LevelDB), HDD 表现更优 (如 indochina-2004 数据集, RocksDB 在 HDD 耗时 1 721 s, SSD 增至 1 844 s), 因 HDD 的顺序吞吐更匹配其大规模刷盘需求; 随机读取型引擎 (LMDB) 在 SSD 上优势显著 (如 uk-2002 数据集, LMDB 在 SSD 耗时 1 011 s, 较 HDD 提升 40%), SSD 的低延迟大幅降低了 B+ tree 遍历时间; 内存优化型引擎 FasterKV 在 SSD 上全面领先 (如 arabic-2005 数据集, FasterKV 在 SSD 耗时 1 265 s, 较 HDD 提升 36%), 其混合日志结构通过批量异步持久化充分利用了 SSD 的高 IOPS (input/output per second).

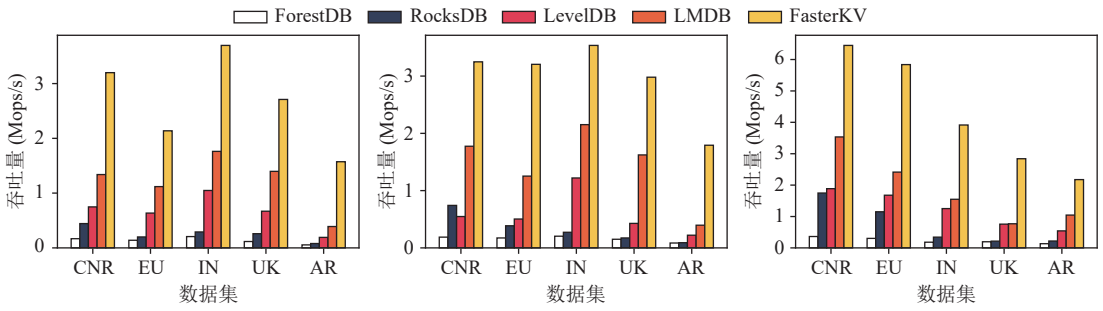
综上, 不同存储引擎的性能表现与存储介质特性息息相关: 顺序写入引擎在 HDD 上更能发挥出优势, 随机读取引擎受益于 SSD 的低延迟, 内存优化引擎则更依赖 SSD 的高 IOPS. 实验结果为工业级系统提供了明确的选型指导——需结合数据结构特性与存储介质物理特性, 选择与目标场景 (如顺序写入、随机读取) 高度适配的存储引擎.

3.4.4 中央处理器性能

本节旨在探究不同代际、频率及架构的 CPU 平台对数据库性能及存储引擎选择的影响, 选取了 3 类代表性 Intel CPU: 服务器级的 Xeon Gold 5318Y (低频多核、高内存带宽但内存延迟高)、传统架构的 Core i9-10900K (单核性能弱), 以及消费级混合架构的 Core i7-13700K (高频少核、低内存延迟), 实验结果如图 11 所示. 本节得出了以下两点结论.

首先, 不同服务器架构下存储引擎的性能排序高度一致, 跨平台稳定性显著. 在 Xeon Gold 5318Y (20 核、内存带宽 256 GB/s、内存延迟 98 ns)、Core i9-10900K (10 核、传统架构、内存延迟 85 ns) 与 Core i7-13700K (16

核 24 线程、混合架构 P+E 核、内存延迟 72 ns) 这 3 类典型服务器/消费级 CPU 平台上, 所有被测存储引擎的性能排序趋势完全统一——FasterKV > LMDB > LevelDB > RocksDB > ForestDB. 以小数据集 (cnr-2000) 测试为例, FasterLV 在 i7 平台的吞吐量达 12.8 万 TPS, LMDB 为 9.2 万 TPS, LevelDB 为 5.1 万 TPS, RocksDB 为 4.3 万 TPS, ForestDB 仅为 2.9 万 TPS; 在 Xeon Gold 平台上, FasterKV (10.5 万 TPS) 仍保持首位, LMDB (8.1 万 TPS) 次之, 与 i7 平台的相对差距 (FasterKV 领先约 29%、LMDB 领先约 12%) 与 i9 平台 (FasterKV 领先约 39%、LMDB 领先约 13%) 基本一致. 更关键的是, 各引擎在不同 CPU 上的性能波动极小: 以 LevelDB 为例, 在 uk-2002 大数据集测试中, Xeon Gold 的标准差为 1.8×10^5, i9 为 1.6×10^5, i7 为 1.5×10^5, 跨平台方差系数稳定在 0.12–0.15 之间, 验证了实验结果的强普适性.



(a) Intel Xeon Gold 5318Y (b) Intel Core i9-10900K (c) Intel Core i7-13700K

图 11 ID 映射在不同处理器下的性能表现

尽管如此, 不同处理器特性 (主频、核心数、内存延迟) 仍然显著影响数据库优势场景的发挥. 高频 CPU (如 i7-13700K, 5.4 GHz 主频) 更适配单核敏感型引擎: 其高频设计使依赖单线程计算的任务 (如哈希计算、B+树遍历) 效率提升显著. 例如, FasterKV 的哈希表操作在 i7 平台上利用 AVX-512 指令集的利用率达 68% (比 Xeon Gold 高 22%), 配合高速缓存 (12 MB L2+30 MB L3) 的深度协同, 单线程吞吐量比 Xeon Gold 高 41%; LMDB 的零拷贝读取在 i7 的低内存延迟 (72 ns) 下, 访问延迟降低 30%, 小数据集 (cnr-2000) 性能比 Xeon Gold 提升 28%. 多核 CPU (如 Xeon Gold 5318Y, 20 核) 更适合高并发任务型引擎: 其多核并行能力弥补了单核效率的不足, 使依赖多线程无锁操作的任务 (如 FasterKV 的无锁哈希) 性能更优. 尽管 Xeon Gold 的内存延迟较高 (98 ns), 但通过批量任务调度 (如将 16 个线程绑定到不同核心), FasterKV 的吞吐量仍能达到 10.5 万 TPS (比 i7 低 19%), 与 i9 平台 (11.2 万 TPS) 接近, 验证了多核架构在高并发场景中的适配性. 混合架构 CPU (如 i7-13700K 的 P+E 核) 优化资源分配型引擎: 其性能核心 (P 核) 与能效核心 (E 核) 的动态调度, 有效缓解了 LSM-Tree 引擎 (如 RocksDB) 的多核扩展瓶颈. 在大数据集 (arabic-2005) 测试中, RocksDB 在 i7 平台的 Compaction 延迟比 Xeon Gold 降低 27% (得益于 P 核处理计算密集型任务、E 核处理 IO 等待), 吞吐量仅下降 37.5% (远低于 LevelDB 的 62% 降幅), 展现了混合架构对复杂任务的资源优化能力.

3.5 小 结

本节围绕键值存储引擎性能评估构建了多维度研究框架, 结合数据负载特征与硬件特性, 对比分析 5 种典型键值存储引擎 (FasterKV、LMDB、LevelDB、RocksDB、ForestDB) 在面向图数据库的 ID 映射负载下的性能差异, 核心结论如下.

(1) 数据负载特征对性能的影响呈现显著差异化. 数据规模实验中, FasterKV 凭借无锁哈希表与异步持久化机制吞吐量最高, 但内存依赖性强; LMDB 的 B+树结构在大规模数据中展现出独特的扩展性优势, 吞吐量逆势微增, 更适配历史图数据离线分析; LSM-Tree 引擎 (LevelDB/RocksDB) 因 Compaction 开销随数据量增长性能显著下降, 需优化动态合并策略; ForestDB 则因 HB+Trie 块重组与随机 I/O 问题始终表现最差. 读写比实验显示, FasterKV 在极高/极低读写比场景优势显著, LMDB 在高读写比时表现优异, LSM-Tree 引擎 (LevelDB/RocksDB) 的性能差

异主要由 Compaction 策略与布隆过滤器优化程度决定, 读写比失衡时底层架构特性 (如内存哈希、B+树索引) 直接影响适应性.

(2) 硬件特征对存储引擎选择起决定性作用. 内存容量实验中, FasterKV 在内存充足时全面领先, 内存不足时因日志回扫周期延长易失败; LSM-Tree 引擎 (RocksDB/LevelDB) 中等内存下通过 MemTable 批量刷盘提升写入, 但 Compaction 引发写入放大; ForestDB 在极端内存约束下耗时显著增加. CPU 线程数实验表明, FasterKV 在低线程场景受益于无锁哈希设计, 但线程数增加时伪共享问题导致性能劣化; RocksDB 与 LevelDB 因全局锁与串行 Compaction 策略, 最佳并发度分别为 4 和 6 线程. 磁盘种类实验显示, LSM-Tree 引擎 (RocksDB/LevelDB) 在 HDD 上通过顺序写入优化性能, LMDB 的 B+树索引则在 SSD 上随机读取效率更高.

此外, 实验发现 CPU 平台对性能排序趋势无显著影响, FasterKV > LMDB > LevelDB > RocksDB > ForestDB 的结论在不同架构下保持稳定, 为跨平台部署提供了理论依据. 本节研究通过详实数据揭示了键值存储引擎性能与数据负载、硬件资源的复杂耦合关系, 为后续协同适配方法构建奠定了科学基础.

4 节点标识存储引擎适配策略

4.1 自适应选型框架

根据第 3 节的键值存储引擎性能评估, 本文确定了 5 种对于键值存储引擎选择有影响的特征因素, 包括数据量、读写比、内存大小、并行线程数和存储介质. 图 12 展示了本研究提出的自适应存储引擎选取流程. 首先, 系统根据查询负载特征 (如读写比、数据量) 和硬件配置 (如内存、存储介质、线程数) 构建特征指标体系, 涵盖了数据量特征 (*DSF*)、读写比特征 (*RWRF*) 以及内存利用率特征 (*MUF*) 等. 随后, 通过在多个存储引擎上执行典型基准负载, 采集性能数据并提取标签, 形成用于训练的样本集. 在模型阶段, 本文采用轻量化的决策树作为引擎推荐模型, 具备良好的可解释性与部署效率. 模型验证阶段通过准确率评估 (Top-1 与 Top-*k*) 和规则提取进一步分析模型效果. 最终, 系统根据训练结果输出最优存储引擎推荐结果, 并通过错误预测分析对推荐偏差进行识别, 从而驱动特征体系的持续优化, 形成一个可自我迭代的推荐闭环.

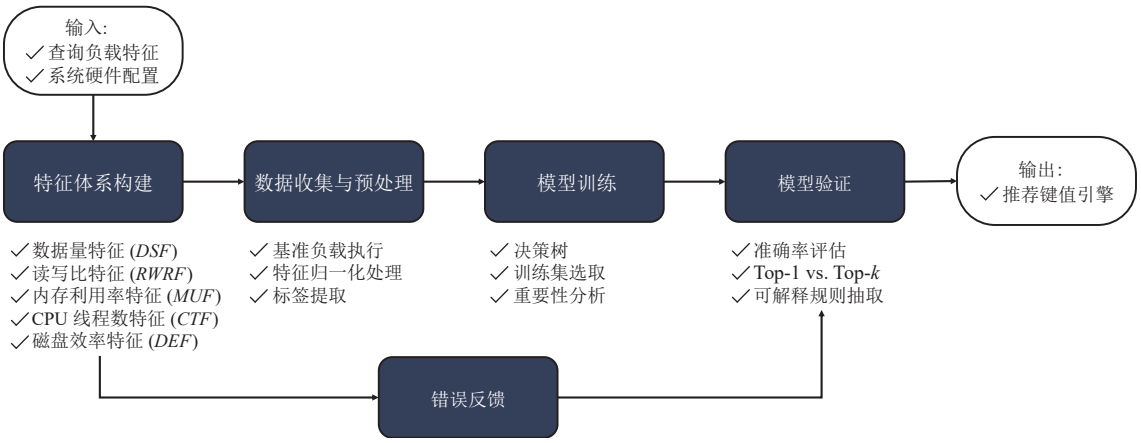


图 12 自适应选型工作流程

4.2 特征体系构建

本节基于数据和硬件负载特征提出了可量化的特征体系构建, 以用于将原始的数值特征转换为决策树模型的输入特征. 这些特征体系不仅能够捕捉各个数值特征的非线性变化, 同时也综合了业务场景中对读写比、内存容量以及多线程并发对性能的影响, 从而为决策树模型提供更丰富、更具区分力的输入特征.

表 4 定义了本文中用到的原始变量. 基于这些特征, 本节构建了如下特征.

表 4 原始变量表

符号	定义
D,V,E	数据集大小, 数据集中的顶点数, 数据集中的边数
R	读写比, 定义为 $R=2E/V$
M	可用内存大小
T	可用线程数
S_d	磁盘读写速度

数据量特征 (dataset size feature, *DSF*): 数据集规模对存储引擎的性能影响呈现出显著的阶段性特征——小数据集时内存操作主导, 性能随着数据量快速提升; 中等数据量时索引深度增加 (如 B+树层数, LSM-Tree SSTable 数量), 性能增速放缓; 大数据量时因分片、分层或动态优化策略 (如 Compaction、缓存预热) 生效, 性能趋于稳定. 为量化这一非线性变化趋势, 构造如下特征:

$$DSF=\frac{\log(1+D)\times\sqrt{D}}{1+\exp\left(-\frac{D}{K}\right)},$$

其中, D 为数据规模, K 为调节参数, 通常取数据量从“中等”到“大数据”的临界阈值, $K=10^6$.

读写比特征 (read-write ratio feature, *RWRF*): 读写比特征用于量化存储引擎在不同读写负载下的性能偏向. 在图数据库场景中, 由于边数量 (E) 通常远大于顶点数量 (V), 且单次边操作通常涉及两个顶点的关联, 因此定义基础读写比 $R=\frac{2E}{V}$. R 值越高, 系统越偏向读操作, 适用于涉及网络分析等查询密集型场景, R 值越低, 写操作占比越高, 常见于动态图更新场景. 为捕捉读写比的非线性影响, 并避免数值波动对模型训练的干扰, 构造如下特征:

$$RWRF=\sqrt{R}\times\log\left(1+\frac{1}{R}\right),$$

该指标可以保证对低读写比敏感, 而对高读写比趋于稳定.

内存利用率特征 (memory utilization feature, *MUF*): 评估数据库在内存受限环境下的资源管理效率, 重点评估内存容量与数据集规模的匹配程度, 以及由此引发的性能影响. 内存大小 (M) 与数据集规模 (D) 的比例是核心指标——内存能否完全容纳数据直接影响访问延迟和操作稳定性. 为综合反映内存从不足到充足的非线性影响, 并量化内存不足时的潜在风险, 设计内存特征如下:

$$MUF=\log\left(1+\frac{M}{D}\right)+\sqrt{\frac{M}{D}},$$

该公式通过内存数据比定义内存与数据规模的匹配关系, 结合 $\log(1+M/D)$ 捕捉内存不足时的风险与非线性增长与 $\sqrt{M/D}$ 反映内存从不足到充足时的持续优势, 量化内存资源对数据库性能的影响, 为内存受限场景下的存储引擎选型提供依据.

CPU 线程数特征 (CPU thread feature, *CTF*): CPU 线程数特征用于量化多线程并行化对数据库性能的实际提升效果. 公式设计如下:

$$CTF=\sqrt{T}\times(1+\log T)\times(1-\sigma),$$

其中, T 为线程数, σ 是任务中的串行化比例.

磁盘效率特征 (disk efficiency feature, *DEF*): 磁盘效率特征用于衡量 I/O 效率. 指标设计如下:

$$DEF=\frac{\log(1+S_d)\times\sqrt{S_d}}{1+\exp(-\beta(r-1))},$$

其中, S_d 是磁盘的实际读取速度, r 是内存能容纳的数据比例. 当 $r<1$ 时, 数据需要从磁盘中读取, 磁盘速度成为系统吞吐的瓶颈之一. $\log(1+S_d)$ 反映了磁盘读速率的对数效应, 确保小范围变化的特征敏感度较高. $\sqrt{S_d}$ 体现了磁盘速度增加时的边际效应递减, 避免数值过大与模型训练失衡. $1+\exp(-\beta(r-1))$ 作为调整因子, 当 $r>1$ 时趋近较

大值, 使磁盘影响增大, 当 $r<1$ 时, 该因子趋向于 1, 使磁盘特征的贡献更显著.

4.3 模型训练

在完成数据特征预处理后, 研究基于决策树模型构建了面向图数据库 ID 映射的数据库选型决策系统. 训练过程以“场景-性能”映射为核心, 首先为每个唯一场景 (由数据集、内存大小、CPU 线程数、磁盘效率等特征定义) 标注最优数据库标签, 即该场景下操作时间最短的存储引擎, 确保标签仅依赖场景特征而非数据库历史性能, 提升模型泛化能力. 为了提高模型的鲁棒性, 我们对连续的特征 (如读写比、内存利用率) 做了标准化处理, 让不同单位的数据能够统一比较. 还通过调整参数, 用类别纯度指标找最好的特征分界点, 解决数据类别不平衡的问题; 另外还用了剪枝方法防止模型过度拟合, 避免小样本类别 (比如 LMDB) 被错误分类.

基于 1 440 组样本, 包括第 3 节中提到的所有实验结果, 本文提出了一个基于决策树的训练模型, 生成的可视化结果如图 13 所示. 综合来说, 模型用递归的方法一层层划分场景, 通过一些判断条件最终生成了一个从软硬件条件到目标存储引擎的映射关系. 其中, 树的根节点时 CPU 可用线程数效率是否低于 2.104, 如果是则属于低并行场景, 否则属于高并行场景. 接下来的节点在根据内存利用率特征 (*MUF*)、磁盘效率特征 (*DEF*) 等特征继续细分. 最终在树的末端给出最优数据库 (比如 FasterKV、RocksDB 等).

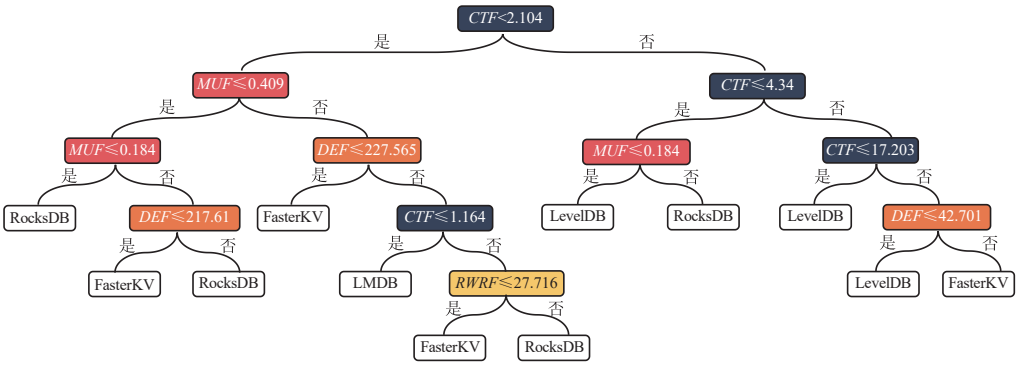


图 13 可视化决策树

模型可解释性: 模型的可解释性主要体现在特征重要性的清晰排序与逻辑关联上. 本文有如下结论: 第一, CPU 线程数特征 (*CTF*) 以 42.6% 的重要性成为占比最核心的决策因素, 直接驱动高并发场景下多线程争用与并行化收益的平衡; 内存利用率特征 (*MUF*) 以 28.9% 的占比主导内存敏感型场景, 例如内存不足时优先选择 RocksDB 或 FasterKV; 磁盘效率特征 (*DEF*) 占 24.4%, 反映机械硬盘、SATA SSD 与 NVMe SSD 的性能差异, 仅在内存不足触发磁盘读写时发挥作用; 数据量特征 (*DSF*) 占 2.1%, 仅在超大规模数据集场景下显现非线性影响; 读写比特征 (*RWRF*) 占 1.9%, 在极端写密集场景 (如 $RWRF<27.716$) 时倾向选择 RocksDB.

模型验证: 本节通过端到端的实验验证决策树模型的有效性, 覆盖准确性、鲁棒性、可扩展性. 在此实验中, 实验基于 38 个异构场景展开, 涵盖了不同的数据集规模、硬件配置及负载特征. 涵盖小数据集低并发、大规模数据集低并发、中大规模数据集中高并发等典型场景. 图 14 展示了实验结果: 横轴为场景编号 (共 38 个), 纵轴为存储引擎性能排序 (1–5 名, 1 为最优), 模型准确率达 92.1% (35/38 场景选最优, 3 个场景选次优, 性能差距可接受). 典型场景分析显示, 高内存场景 (如 cnr-2000, M=100 MB) 模型倾向选 FasterKV (无锁哈希表设计优势显著); 高并发场景 (如 uk-2002, T=1) 模型在数据部分落盘时推荐次优的 LevelDB (轻量级 Compaction 策略稳定); 超大规模数据集 (如 arabic-2005, D=93.7 GB) 下模型仍能精准选优, 验证了 *DSF* 特征对规模效应的非线性捕捉能力. 实践结论表明, 协同适配方法通过权衡数据负载与硬件约束, 为图数据库顶点编码映射任务提供了科学、自适应的存储引擎选择策略, 在内存充足、高并发等场景下分别推荐 FasterKV、LevelDB 等引擎, 兼具准确性、鲁棒性与工程指导价值.

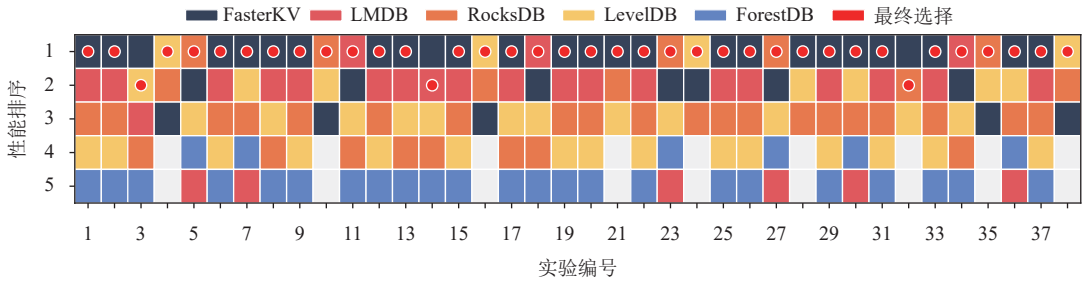


图 14 模型验证结果

4.4 案例研究

本节通过一个全球社交网络平台的典型场景来展开如何使用该自适应的选型方法. 该案例覆盖特征提取、模型输入构建、决策树推理全过程.

社交网络平台需要分析用户关系链, 支撑好友推荐、社区发现、识别用户所属的社交圈子如兴趣小组、地域集群等功能. 该业务场景的活跃用户账户达 1.2 亿个, 用户关系超过 25 亿条, 总数据量 280 GB. 计算资源采用 64 核 Intel Xeon Platinum 8380 CPU, 配备 256 GB 内存与 NVMe SSD 磁盘, 但内存需同时服务在线请求与存储引擎, 实际可用内存动态分配约 128 GB, 计划采用双线程运行.

基于这一场景, 本工作需要通过特征提取与模型输入构建为决策树模型提供输入. 将这些原始数据代入第 4.2 节的特征公式中, 得到 $DSF = 74.2$, $RWRF = 41.7$, $MUF = 0.89$, $CTF = 1.5554$, $DEF = 38.7$.

将这 5 个值输入训练好的决策树模型之后, 模型按以下逻辑完成推理. 根节点判断 $CTF = 1.554 \leqslant 2.104$, 进入左子树 (低并行场景); 中间节点 $MUF = 0.89 > 0.184$, 进入右子树 (内存较充足), 表明内存可基本容纳数据结构, 无需过度依赖磁盘优化; 中间节点 $DEF = 38.7 \leqslant 227.565$, 进入左子树 (磁盘效率一般) 此时磁盘非主要瓶颈, 优先考虑内存优化型引擎, 最终叶子节点推荐 FasterKV, 其哈希索引与异步持久化提供了良好的性能, 适配当前场景.

这一过程验证了协同适配方法的核心优势: 通过量化数据量特征 (*DSF*)、读写比特征 (*RWRF*)、CPU 线程数特征 (*CTF*)、内存利用率特征 (*MUF*)、磁盘效率特征 (*DEF*) 的耦合效应, 精准匹配图数据库场景的“内存-磁盘-计算”资源约束. 决策树规则的可解释性 (如“内存充足时优先选 FasterKV”) 为工程师提供了人工干预的依据, 最终平台通过该模型动态选择最优存储引擎, 实现了全局性能最优. 案例表明, 协同适配方法不仅能高效解决图数据库 ID 映射场景的存储引擎选型问题, 更通过可解释的决策逻辑为工业界提供了“数据-硬件-场景”协同优化的实用范式.

5 图数据库集成应用

为进一步验证本文所提出的节点标识符映射与存储适配方法在真实图数据库系统中的可行性与应用价值, 本文选用了蚂蚁公司开源的金融级图数据库 TuGraph[46]进行集成实验. 之所以选择 TuGraph, 主要有以下两点原因: 首先, TuGraph 在其开发者文档中明确公开了“原始顶点标识符 (Original VID) 与映射后内部 VID (Mapped VID)”的接口, 例如 MappedVid (original_vid) 与 OriginalVid (vid) 方法. 这一特性使得研究者能够直接切入节点标识符映射模块, 替换或植入自定义的映射逻辑, 从而较为透明地评估标识符转换对系统性能的影响. 其次, 其他主流图数据库 [47–50]虽然也采用了原始标识符到内部整数 ID 的映射机制, 但其映射模块通常未公开或难以修改, 使得子系统级别的定量验证变得困难. 基于上述考量, 本节将介绍在 TuGraph 中构建的端到端实验设计、所替换的映射模块实现方式、测量指标与实验结果, 旨在展示本文方法在真实图数据库环境中的集成效果与性能提升.

5.1 实验设置

在本研究中, 为深入评估所提方法在真实图数据库的业务场景中的表现, 我们选用 TuGraph 作为实验平台, 并将实验聚焦于紧凑标识符 (Compact ID) 场景下不同键值存储引擎对节点与边插入操作的吞吐表现. 具体而言, 本实验步骤如下: 首先在 TuGraph 中定位其顶点与边导入模块, 将默认的紧凑 ID 映射流程保留, 保证所有顶点在导

入前均使用整数 VID 表示, 随后我们设计了统一负载: 导入一个完整图数据集, 并对每条边执行插入操作, 该操作由两部分组成——如果起始顶点或目标顶点已存在, 则直接基于该顶点的紧凑 ID 创建新边; 若顶点尚不存在, 则首先创建该顶点 (分配紧凑 ID), 然后再创建对应边. 接着我们在不同硬件配置下, 比较了若干不同的键值存储引擎在图数据中执行分析型事务图算法 PageRank 的执行时间.

评测基准: 本文选择了 6 个基线键值存储引擎来验证图数据库集成应用时的端到端性能, 包括 FasterKV、LMDB、LevelDB、RocksDB、ForestDB 这 5 种代表性存储引擎, 和 TuGraph 的原始 ID 映射存储引擎. 其中 TuGraph 目前采用一个内存型哈希表 CuckooMap 存储原始 ID 到系统 ID 的映射, 所以暂时无法处理超出内存的情形. 除此之外, 本文还比较了使用原始 ID 执行 PageRank 算法的性能.

实验环境: 为确保端到端集成实验与本文前文的大规模评测保持一致性, TuGraph 环境选型严格基于第 3 节所使用的真实硬件平台. 考虑到本文实验平台均为服务器级 CPU, 且未配置 SATA SSD, 本节从现有环境中选取 3 类具有代表性的配置用于验证自适应存储引擎选型在真实图数据库中的有效性: (1) 平台 A (高内存 + NVMe SSD): 该环境对应表 2 中的“内存充足 + NVMe”配置, 具有高随机读写带宽与低访问延迟, 适合模拟在线分析 (OLAP) 和高并发图查询场景, 用于评估不同键值存储引擎在无明显资源瓶颈下的极限性能; (2) 平台 B (中等内存 + NVMe SSD): 该环境对应表 2 中通过限制可用内存获得的“中等内存”场景, 用于刻画节点映射在内存压力下的行为, 评估自适应策略在缓存容量受限、查找压力增加时的选型稳定性; (3) 平台 C (中等内存 + HDD): 该环境对应表 2 中的机械硬盘配置, 用于模拟离线批处理或大规模导入任务, 强调顺序写入主导的场景下不同键值引擎对写放大与查找延迟的敏感性. 基于上述 3 个来自原实验体系的代表性硬件环境, 本文在 TuGraph 中替换其内部原生的 VID 映射组件为不同的键值存储引擎, 实现紧凑 ID 的创建与查找, 并在端到端导入与查询流程中测量插入吞吐、查找吞吐以及整体导入延迟, 以验证自适应选型策略在真实图数据库系统中的可行性与性能收益.

5.2 实验结果

如图 15 所示, 各方案的端到端执行时间以平台内的基线配置归一化, 数值越低表示整体耗时越短. 整体来看, 本文的选择方案在 3 类实验平台上均表现出高度稳定性和领先性, 几乎在所有数据集与硬件组合下都处于第 1 梯队. 在平台 A 与平台 B 上, 对于 indochina-2004、uk-2002 与 arabic-2005 这 3 个数据集, 本文的选择方案要么与当前最快的单一引擎保持一致, 要么显著优于次优方案, 对应的归一化端到端时间通常只有传统基线方案的约 1/10. 这说明在在线分析与中等内存压力的场景下, 通过根据数据与硬件特征自适应地选择键值存储引擎, 可以有效消除“标识符映射开销主导整体延迟”的历史瓶颈, 使得 ID 映射不再成为图分析任务的主要性能障碍.

进一步观察平台 B 上数据规模最大的一组实验可以发现, 当图规模继续增大、映射访问局部性显著恶化时, 所有引擎的映射时间都会急剧上升, 在这一极端场景下, 原始 ID 由于完全避免了映射转换, 在个别组合上获得了极其有限的延迟优势. 然而, 即便在这种“映射成本失控”的不利条件下, 本文的选择方案在大多数数据集上的端到端时间仍与最优方案接近, 并始终优于 TuGraph 默认配置以及传统磁盘型引擎. 换言之, 只有在映射成本被人为放大到非常极端的情况下, 系统才需要退回到原始 ID 这一“保底”选项; 在绝大多数更常见的负载和规模下, 自适应选型所带来的综合收益远大于原始 ID 的局部优势.

更为关键的是, 本文的选择方案在资源受限环境中仍展现出显著的可靠性与鲁棒性. 平台 C 的结果表明, 随着可用内存被进一步压缩且存储介质退化为 HDD, 原始 ID 与依赖内存映射结构的 TuGraph 默认配置均在大图数据集上出现运行失败, 图 15 中以“NaN”标记; 相比之下, 本文的选择方案在所有数据集上均能够完成完整的端到端执行. 在可成功运行的配置中, 本文方案对应的归一化端到端时间显著低于 LMDB、RocksDB、ForestDB 等传统基线, 在部分数据集上甚至可以维持一个数量级左右的优势. 同时可以观察到, 随着数据规模增加, 传统存储引擎之间的性能差距呈快速放大趋势, 某些组合的归一化时间由接近 1 的个位数差异迅速扩大到远大于 1 的两位数比例; 相对而言, 本文选择方案在不同数据规模与不同硬件平台之间的性能波动始终保持在可控范围内, 并未出现明显的退化或失稳行为.

综合上述结果可以得出结论: 无论是在高性能 NVMe 平台, 还是在内存与 I/O 资源都更为紧张的 HDD 平台

上, 基于本文自适应选型策略得到的最终引擎选择, 都能够在端到端意义上提供更低的执行时间与更强的性能稳定性. 这不仅验证了对标识符映射子系统进行精细建模与引擎自适应选型的必要性, 也表明该策略在真实图数据库系统中具有良好的通用性与工程应用前景.

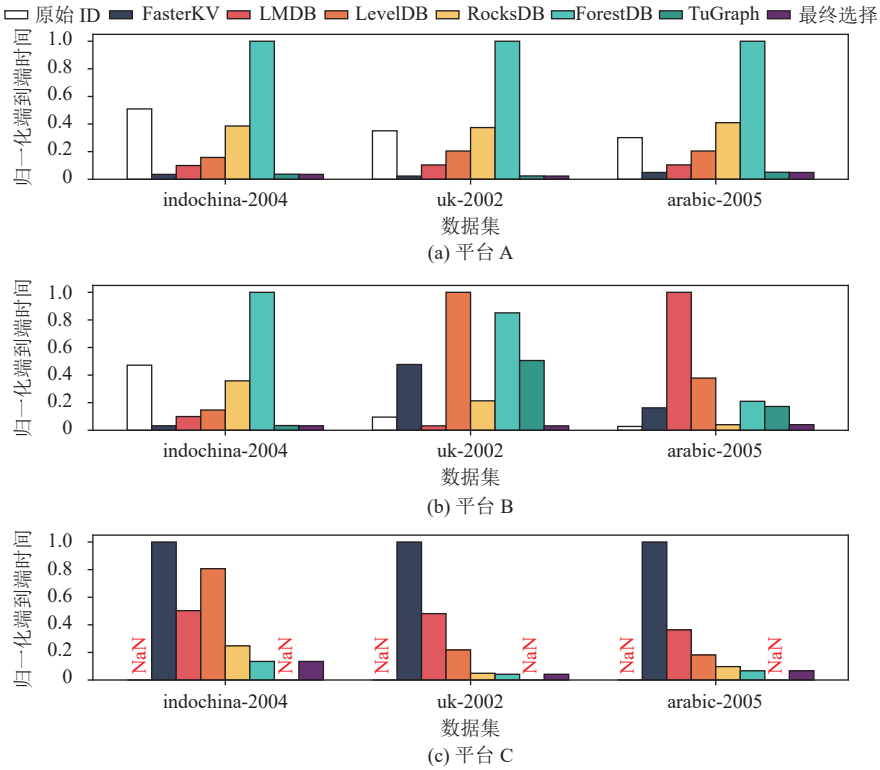


图 15 图数据库端到端集成实验

6 总 结

图数据库作为社交网络分析、金融风控等领域的核心基础设施, 其价值源于对复杂关系建模与高效查询的显著优势. 在这一架构中, 节点唯一标识符 (NodeID) 发挥着不可或缺的关键作用, 直接承担身份表征、关系寻址及图算法执行的核心职能. 然而, 当前主流图数据库对 NodeID 映射的管理存在明显局限: 一方面, 依赖通用键值存储 (如 RocksDB、LMDB) 的方案缺乏对 ID 映射特有负载的针对性优化, 包括高频查询、动态增删与混合负载的适应性缺陷; 另一方面, 跨软硬件环境 (如 CPU 内存配置差异、SSD/HDD 存储介质切换、多线程并发场景) 的适配能力不足, 导致超大规模数据处理时延迟波动加剧与内存资源不足.

为解决上述问题, 本文系统性评估了主流键值存储引擎在节点标识映射场景的综合性能, 通过设计多场景测试框架, 精确量化不同引擎在吞吐量、响应延迟等维度的性能差异. 本文通过异构环境测试, 证实内存容量、线程数、SSD/HDD 介质是影响性能的关键硬件因素, 数据规模与读写比是核心负载特征. 基于此, 构建融合数据负载特征 (数据规模、读写比) 与硬件特征 (内存容量、线程数量、磁盘类型) 的轻量级推荐模型, 实现存储引擎的跨场景自适应选型. 实验验证表明, 该模型推荐最优引擎的准确率达 92.1%, 次优场景性能差距极小, 为图数据库系统提供了高效的数据驱动选型支持, 显著降低了因引擎-环境适配失配引发的性能损耗与运维成本.

References

[1] Cui B, Gao J, Tong YX, Xu JQ, Zhang DX, Zou L. Progress and trend in novel data management system. Ruan Jian Xue Bao/Journal of

Software, 2019, 30(1): 164–193 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5646.htm [doi: 10.13328/j.cnki.jos.005646]

[2] Robinson I, Webber J, Eifrem E. Graph Databases: New Opportunities for Connected Data. 2nd ed., Sebastopol: O'Reilly Media Inc., 2015.

[3] Vicknair C, Macias M, Zhao ZD, Nan XF, Chen YX, Wilkins D. A comparison of a graph database and a relational database: A data provenance perspective. In: Proc. of the 48th Annual ACM Southeast Conf. Oxford: ACM, 2010. 42. [doi: 10.1145/1900008.1900067]

[4] Angles R. A comparison of current graph database models. In: Proc. of the 28th IEEE Int'l Conf. on Data Engineering Workshops. Arlington: IEEE, 2012. 171–177. [doi: 10.1109/ICDEW.2012.31]

[5] Amazon Web Services, Inc. Amazon Neptune: A graph database service for knowledge graphs. 2023. https://aws.amazon.com/neptune

[6] Bebee BR, Choi D, Gupta A, Gutmans A, Khandelwal A, Kiran Y, Mallidi S, McGaughy B, Personick M, Rajan K, Rondelli S, Ryazanov A, Schmidt M, Sengupta K, Thompson BB, Vaidya D, Wang S. Amazon Neptune: Graph data management in the cloud. In: Proc. of the 2018 Posters & Demonstrations, Industry and Blue Sky Ideas Tracks Co-located with the 17th Int'l Semantic Web Conf. Monterey: CEUR-WS.org, 2018.

[7] Pan ZX, Wu T, Zhao QW, Zhou Q, Peng ZW, Li JF, Zhang Q, Feng GY, Zhu XW. GeaFlow: A graph extended and accelerated dataflow system. Proc. of the ACM on Management of Data, 2023, 1(2): 191. [doi: 10.1145/3589771]

[8] Gonzalez JE, Low Y, Gu HJ, Bickson D, Guestrin C. PowerGraph: Distributed graph-parallel computation on natural graphs. In: Proc. of the 10th USENIX Symp. on Operating Systems Design and Implementation. Hollywood: USENIX Association, 2012. 17–30.

[9] Zhang YM, Yang MJ, Baghdadi R, Kamil S, Shun JL, Amarasinghe S. Graphit: A high-performance graph DSL. Proc. of the ACM on Programming Languages, 2018, 2(OOPSLA): 121. [doi: 10.1145/3276491]

[10] Chandramouli B, Prasaad G, Kossmann D, Levandoski J, Hunter J, Barnett M. FASTER: A concurrent key-value store with in-place updates. In: Proc. of the 2018 Int'l Conf. on Management of Data. Houston: ACM, 2018. 275–290. [doi: 10.1145/3183713.3196898]

[11] Dong SY, Kryczka A, Jin YQ, Stumm M. RocksDB: Evolution of development priorities in a key-value store serving large-scale applications. ACM Trans. on Storage, 2021, 17(4): 26. [doi: 10.1145/3483840]

[12] Google. LevelDB: A fast persistent key-value store [Software]. 221. https://github.com/google/leveldb

[13] LMDB. 2025. https://www.symas.com/mdb

[14] Ahn JS, Seo C, Mayuram R, Yaseen R, Kim JS, Maeng S. ForestDB: A fast key-value storage system for variable-length string keys. IEEE Trans. on Computers, 2016, 65(3): 902–915. [doi: 10.1109/TC.2015.2435779]

[15] Angles R. The property graph database model. In: Proc. of the 12th Alberto Mendelzon Int'l Workshop on Foundations of Data Management. Cali: CEUR-WS.org, 2018.

[16] Al-Baghdadi A, Lian X. Topic-based community search over spatial-social networks. Proc. of the VLDB Endowment, 2020, 13(12): 2104–2117. [doi: 10.14778/3407790.3407812]

[17] Chen Z, Zhang F, Chen Y, Fang XK, Feng GY, Zhu XW, Chen WG, Du XY. Enabling window-based monotonic graph analytics with reusable transitional results for pattern-consistent queries. Proc. of the VLDB Endowment, 2024, 17(11): 3003–3016. [doi: 10.14778/3681954.3681979]

[18] Jiang JX, Yao SY, Li YC, Wang QG, He BS, Chen M. Dupin: A parallel framework for densest subgraph discovery in fraud detection on massive graphs. Proc. of the ACM on Management of Data, 2025, 3(3): 150. [doi: 10.1145/3725287]

[19] Hogan A, Blomqvist E, Cochez M, D'amato C, De Melo G, Gutierrez C, Kirrane S, Gayo JEL, Navigli R, Neumaier S, Ngomo ACN, Polleres A, Rashid SM, Rula A, Schmelzeisen L, Sequeda J, Staab S, Zimmermann A. Knowledge graphs. ACM Computing Surveys, 2022, 54(4): 71. [doi: 10.1145/3447772]

[20] Liu Y, Li JZ, Gao H. Mining frequent jump patterns from graph databases. Ruan Jian Xue Bao/Journal of Software, 2010, 21(10): 2477–2493 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3831.htm [doi: 10.3724/SP.J.1001.2010.03831]

[21] Zhu XW, Chen WG, Zheng WM, Ma XS. Gemini: A computation-centric distributed graph processing system. In: Proc. of the 12th USENIX Symp. on Operating Systems Design and Implementation. Savannah: USENIX Association, 2016. 301–316.

[22] Shun JL, Blelloch GE. Ligra: A lightweight graph processing framework for shared memory. In: Proc. of the 18th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. Shenzhen: ACM, 2013. 135–146. [doi: 10.1145/2442516.2442530]

[23] Kyrola A, Blelloch G, Guestrin C. GraphChi: Large-scale graph computation on just a PC. In: Proc. of the 10th USENIX Symp. on Operating Systems Design and Implementation. Hollywood: USENIX Association, 2012. 31–46.

[24] Gonzalez JE, Xin RS, Dave A, Crankshaw D, Franklin MJ, Stoica I. GraphX: Graph processing in a distributed dataflow framework. In: Proc. of the 11th USENIX Symp. on Operating Systems Design and Implementation. Broomfield: USENIX Association, 2014. 599–613.

[25] Yan D, Yuan LH, Ahmad A, Zheng CG, Chen HZ, Cheng J. Systems for scalable graph analytics and machine learning: Trends and

methods. In: Proc. of the 30th ACM SIGKDD Conf. on Knowledge Discovery and Data Mining. Barcelona: ACM, 2024. 6627–6632. [doi: 10.1145/3637528.3671472]

[26] Page L, Brin S, Motwani R, Winograd T. The PageRank citation ranking: Bringing order to the Web. 1998. https://gwern.net/doc/technology/google/1998-page.pdf

[27] Lepers B, Balmau O, Gupta K, Zwaenepoel W. KVell: The design and implementation of a fast persistent key-value store. In: Proc. of the 27th ACM Symp. on Operating Systems Principles. Huntsville: ACM, 2019. 447–461. [doi: 10.1145/3341301.3359628]

[28] Zhou JY, Xu M, Shraer A, Namasivayam B, Miller A, Tschannen E, Atherton S, Beamon AJ, Sears R, Leach J, Rosenthal D, Dong X, Wilson W, Collins B, Scherer D, Grieser A, Liu Y, Moore A, Muppana B, Su XG, Yadav V. FoundationDB: A distributed key value store. ACM SIGMOD Record, 2022, 51(1): 24–31. [doi: 10.1145/3542700.3542707]

[29] Bender MA, Farach-Colton M, Fineman JT, Fogel YR, Kuszmaul BC, Nelson J. Cache-oblivious streaming B-trees. In: Proc. of the 19th Annual ACM Symp. on Parallel Algorithms and Architectures. San Diego: ACM, 2007. 81–92. [doi: 10.1145/1248377.1248393]

[30] Graefe G. Modern B-tree techniques. Foundations and Trends® in Databases, 2011, 3(4): 203–402. [doi: 10.1561/1900000028]

[31] O'Neil P, Cheng E, Gawlick D, O'Neil E. The log-structured merge-tree (LSM-tree). Acta Informatica, 1996, 33(4): 351–385. [doi: 10.1007/s002360050048]

[32] Sears R, Ramakrishnan R. bLSM: A general purpose log structured merge tree. In: Proc. of the 2012 ACM SIGMOD Int'l Conf. on Management of Data. Scottsdale: ACM, 2012. 217–228. [doi: 10.1145/2213836.2213862]

[33] Lu LY, Pillai TS, Gopalakrishnan H, Arpaci-Dusseau AC, Arpaci-Dusseau RH. WiscKey: Separating keys from values in SSD-conscious storage. ACM Transactions on Storage, 2017, 13(1): 5. [doi: 10.1145/3033273]

[34] Matsunobu Y, Dong SY, Lee H. MyRocks: LSM-tree database storage engine serving Facebook's social graph. Proc. of the VLDB Endowment, 2020, 13(12): 3217–3230. [doi: 10.14778/3415478.3415546]

[35] Sarkar S, Papon TI, Staratzis D, Athanassoulis M. Lethe: A tunable delete-aware LSM engine. In: Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. Portland: ACM, 2020. 893–908. [doi: 10.1145/3318464.3389757]

[36] Bayer R, Unterauer K. Prefix B-trees. ACM Trans. on Database Systems, 1977, 2(1): 11–26. [doi: 10.1145/320521.320530]

[37] Morrison DR. PATRICIA—Practical algorithm to retrieve information coded in alphanumeric. Journal of the ACM, 1968, 15(4): 514–534. [doi: 10.1145/321479.321481]

[38] Bernstein PA, Goodman N. Concurrency control in distributed database systems. ACM Computing Surveys, 1981, 13(2): 185–221. [doi: 10.1145/356842.356846]

[39] Witten IH, Moffat A, Bell TC. Managing Gigabytes: Compressing and Indexing Documents and Images. 2nd ed., San Francisco: Morgan Kaufmann, 1999.

[40] LAW: Laboratory for Web algorithmics datasets. 2023. http://law.di.unimi.it

[41] Leskovec J, Sosič R. SNAP: A general-purpose network analysis and graph-mining library. ACM Trans. on Intelligent Systems and Technology, 2017, 8(1): 1. [doi: 10.1145/2898361]

[42] Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R. Benchmarking cloud serving systems with YCSB. In: Proc. of the 1st ACM Symp. on Cloud computing. Indianapolis: ACM, 2010. 143–154. [doi: 10.1145/1807128.1807152]

[43] Silberschatz A, Korth HF, Sudarshan S. Database System Concepts. 6th ed., New York: McGraw-Hill, 2010.

[44] Herlihy M, Shavit N, Luchangco V, Spear M. The Art of Multiprocessor Programming. 2nd ed., Cambridge: Morgan Kaufmann, 2020. [doi: 10.1016/C2011-0-06993-4]

[45] Dice D, Shavit N. TLRW: Return of the read-write lock. In: Proc. of the 22nd Annual ACM Symp. on Parallelism in Algorithms and Architectures. Thira: ACM, 2010. 284–293. [doi: 10.1145/1810479.1810531]

[46] TuGraph-Family. tugraph-db: A high performance graph database [Source code]. 2025. https://github.com/TuGraph-family/tugraph-db

[47] Neo4j. Neo4j graph database. https://neo4j.com/

[48] Kùzu. Kùzu graph database. https://kuzudb.github.io/

[49] NebulaGraph. NebulaGraph: Distributed graph database. https://www.nebula-graph.io/

[50] TigerGraph. TigerGraph: Native parallel graph database. https://www.tigergraph.com/

附中文参考文献

[1] 崔斌, 高军, 童咏昕, 许建秋, 张东祥, 邹磊. 新型数据管理系统研究进展与趋势. 软件学报, 2019, 30(1): 164–193. http://www.jos.org.cn/1000-9825/5646.htm [doi: 10.13328/j.cnki.jos.005646]

[20] 刘勇, 李建中, 高宏. 从图数据库中挖掘频繁跳跃模式. 软件学报, 2010, 21(10): 2477–2493. http://www.jos.org.cn/1000-9825/3831.

htm [doi: 10.3724/SP.J.1001.2010.03831]

作者简介

陈政, 博士, 助理研究员, 主要研究领域为图计算, 图数据库.

张峰, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为数据库理论与系统, 数据压缩.

齐畅, 硕士生, 主要研究领域为图数据库.

赵郑诣隆, 本科生, 主要研究领域为图数据库.

杜小勇, 博士, 教授, 博士生导师, CCF 会士, 主要研究领域为高性能数据库, 智能信息检索, 非结构化数据管理.