软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(12):5821−5850 [doi: 10.13328/j.cnki.jos.007411] [CSTR: 32375.14.jos.007411]

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

NUMA-conscious 外键连接优化技术*

韩瑞琛[1,2,3], 张延松[1,2,3,4], 刘 专[5], 张 宇[6], 焦 敏[3], 王 珊[1,2,3]

[1](数据库与商务智能教育部工程研究中心, 北京 100872)

[2](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)

[3](中国人民大学 信息学院, 北京 100872)

[4](中国人民大学 中国调查与数据中心, 北京 100872)

[5](英特尔 (中国) 有限公司, 北京 100190)

[6](国家卫星气象中心, 北京 100081)

通信作者: 张延松, E-mail: zhangys_ruc@hotmail.com

摘　要: NUMA (non-uniform memory access) 是现代多核、多路处理器平台上主流的内存访问架构, NUMA 访问延迟对数据库的查询性能有较大影响, 因此如何降低查询处理中跨 NUMA 节点的访问延迟是现代内存数据库查询优化的热点问题之一. 不同的处理器在 NUMA 架构、NUMA 延迟等方面差异较大, 因此 NUMA 优化技术需要与硬件特性相结合. 基于内存数据库执行代价最高和对数据局部性依赖最强的内存外键连接算法, 面向代表性的 ARM、Intel CLX、Intel ICX、AMD Zen2 和 AMD Zen3 这 5 个处理器 NUMA 架构和延迟特征, 探索了不同 NUMA 优化方法, 包括 NUMA-conscious 和 NUMA-oblivious 实现技术. 在数据存储、数据分片、连接中间结果缓存等方面采用不同的优化方案, 比较了不同处理器架构上的算法性能, 实验结果表明, NUMA-conscious 优化策略需软、硬件相结合, 其中 Radix Join 对 NUMA 延迟敏感度为中性, 在 5 个不同的处理器平台上, NUMA 优化性能收益稳定在 30% 左右, NPO 算法对 NUMA 延迟敏感度较高, 在不同平台 NUMA 优化性能收益在 38%–57%, Vector Join 算法对 NUMA 延迟敏感但影响幅度较小, NUMA 优化性能收益在 1%–25% 之间, 且在算法性能特征上, Vector Join 受 cache 效率影响比 NUMA 延迟影响更大; NUMA-conscious 优化技术在 ARM 平台差异较大, 在 x86 平台差异极小, NUMA-oblivious 算法复杂度更低, 具有较好的通用性. 从处理器硬件发展趋势来看, 降低 NUMA 访问延迟可以有效地降低不同 NUMA-conscious 优化算法的性能差异, 简化连接算法的复杂度, 提高连接操作性能.

关键词: NUMA 架构; NUMA 感知优化; 非 NUMA 感知实现; 向量连接; 连接基准

中图法分类号: TP311

中文引用格式: 韩瑞琛, 张延松, 刘专, 张宇, 焦敏, 王珊. NUMA-conscious外键连接优化技术. 软件学报, 2025, 36(12): 5821–5850. http://www.jos.org.cn/1000-9825/7411.htm

英文引用格式: Han RC, Zhang YS, Liu Z, Zhang Y, Jiao M, Wang S. NUMA-conscious Foreign Key Join Optimization Technique. Ruan Jian Xue Bao/Journal of Software, 2025, 36(12): 5821–5850 (in Chinese). http://www.jos.org.cn/1000-9825/7411.htm

NUMA-conscious Foreign Key Join Optimization Technique

HAN Rui-Chen[1,2,3], ZHANG Yan-Song[1,2,3,4], LIU Zhuan[5], ZHANG Yu[6], JIAO Min[3], WANG Shan[1,2,3]

[1](Engineering Research Center of Database and Business Intelligence, Ministry of Education, Beijing 100872, China)

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

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

* 基金项目: 国家重点研发计划 (2023YFB4503600); 国家自然科学基金 (U23A20299, 62172424, 62276270, 62322214)
收稿时间: 2024-09-29; 修改时间: 2024-12-22, 2025-01-25; 采用时间: 2025-02-12; jos 在线出版时间: 2025-07-17
CNKI 网络首发时间: 2025-07-18

[4](National Survey Research Center, Renmin University of China, Beijing 100872, China)
[5](Intel China Research Center Ltd., Beijing 100190, China)
[6](National Satellite Meteorological Center, Beijing 100081, China)

Abstract: Non-uniform memory access (NUMA) is the mainstream memory access architecture for state-of-the-art multicore and multi-way processor platforms. Reducing the latency of cross-NUMA node accesses during queries is a key issue for modern in-memory database query optimization techniques. Due to the differences in NUMA architectures and NUMA latency across various processors, NUMA optimization techniques should be combined with hardware characteristics. This study focuses on the in-memory foreign key join algorithm, which has high cost and strong locality of data dependency in in-memory databases, and explores different NUMA optimization techniques, including NUMA-conscious and NUMA-oblivious implementations, on five platforms featuring ARM, Intel CLX/ICX, and AMD Zen2/Zen3 processors. The study also compares the performance of the algorithms across different processor platforms with strategies such as data storage, data partitioning, and join intermediate result caching. Experimental results show that the NUMA-conscious optimization strategy requires the integration of both software and hardware. Radix Join demonstrates neutral sensitivity to NUMA latency, with NUMA optimization gains constantly around 30%. The NPO algorithm shows higher sensitivity to NUMA latency, with NUMA optimization gains ranging from 38% to 57%. The Vector Join algorithm is sensitive to NUMA latency, but the impact is relatively minor, with NUMA optimization gains varying from 1% to 25%. For algorithm performance characteristics, cache efficiency influences the Vector Join performance more than NUMA latency. NUMA-conscious optimization techniques show significant differences on ARM platforms, while the differences are minimal on x86 platforms. The less complex NUMA-oblivious algorithms exhibit greater generality. Given hardware trends, reducing NUMA latency can effectively reduce performance gaps in NUMA-conscious optimization techniques, simplify join algorithm complexity, and improve join operation performance.

Key words: NUMA architecture; NUMA-conscious optimization; NUMA-oblivious implementation; vector join; join benchmark

当前主流服务器通常采用 NUMA (non-uniform memory access, 非一致性内存访问) 架构, 在 NUMA 架构中 CPU 通过集成的内存控制器低延迟访问本地内存, 而 CPU 之间通过 QPI 通道高延迟访问远程内存. 在 4 路、8 路以及 16 路 CPU 服务器中, 远程内存访问需要经过更多 QPI 通道, 从而产生更高访问延迟. NUMA 架构中本地内存与远程内存访问的延迟差异较大, 使得 NUMA 内存访问局部性与 cache 局部性共同成为内存查询优化的重要影响因素, 最大化线程对内存访问的 NUMA 局部性是 NUMA 优化的重要目标. 因此, 在数据布局和算法设计上需要根据不同 CPU 的 NUMA 架构和负载特征进行针对性的优化设计.

本文研究在 NUMA 架构上内存数据库中执行代价较高的外键连接操作优化技术. 外键连接是 OLAP (on-line analytical processing, 联机分析处理) 中重要的算子, 在主键表 R (假设主键为 keyP) 与外键表 S (假设主键为 keyF) 之间执行基于等值条件的连接操作 (R.keyP=S.keyF), S 表记录外键列 keyF 上的每个值都可以在 R 表主键列 keyP 上找到唯一的值与之对应. 外键连接是数据仓库负载的基础操作, 事实表通过外键与多个维表执行连接操作. 外键连接操作的主要实现技术有哈希连接[1]、排序归并连接[2]、向量连接[3]等, 相关研究表明, 哈希连接性能优于排序归并连接, 因而成为当前内存连接算法的主要研究对象. 哈希连接算法又主要分为 hardware-conscious 的 Radix 分区哈希连接算法 (parallel Radix join optimized, PRO) 和 hardware-oblivious 的无分区哈希连接算法 (NPO, no partitioning join optimized), 前者需要根据 CPU 硬件参数 (如 TLB 大小等) 设计分区趟数等关键参数, 将较大的表划分为较小的分区实现面向 CPU 私有 cache 的 in-cache 哈希连接, 后者主要依赖持续增长的共享 L3 cache 容量和多核并行访问能力, 通过简单共享哈希表提供硬件自适应的哈希连接算法. 两种算法具有不同的 cache 局部性特征, 还具有其他不同的内存局部性特征: PRO 算法在分区阶段将两个输入表划分为适合 cache 大小的分区, 将数据写到不同 NUMA 节点上的分区时, 会产生 NUMA 访问延迟, 随后在较小分区上 cache 执行哈希表创建和探测阶段操作时需要处理线程与分区所在的 NUMA 节点绑定, 避免跨 NUMA 节点的数据访问延迟; NPO 算法跨 NUMA 节点创建共享哈希表, 在哈希探测阶段跨 NUMA 节点访问共享哈希表. 哈希表是一种无索引等值连接算法, 依赖哈希表完成等值连接操作. 向量连接算法则是一种索引连接算法, 也是一种面向 OLAP 负载的连接算法. 相对于普通的主键-外键约束, 向量索引算法需要主键表 R 使用代理键索引 (surrogate key, 即连续的 1, 2, 3, …序列) 将主键表映射为维轴, 主键扩展了地址映射约束, 使外键表 S 的外键成为连接索引 (join index), 从而将外键值直接映射为主键表的偏移地址, 实现无哈希化的键值-地址映射连接. 向量连接算法将 R 表记录映射为以代理键索

引为下标的内存向量 (数组), 因此相对于哈希表而言更小, 具有更高的 cache 局部性, 从而能够更好地适应当前 CPU L3 cache 容量持续增长的硬件发展趋势; 另一方面, 键值-地址映射消除了哈希探测的哈希值计算、记录遍历和键值匹配代价, 使得 CPU cycle 消耗更低, 从而降低了 CPU 对复杂指令性能的依赖, 能更好地自适应当前 CPU 核心数量快速增长的趋势. 向量连接算法与 NPO 算法相似, 均依赖对共享数据结构 (共享向量) 的访问, 但 NPO 算法使用较大的哈希表从而具有较高的哈希探测代价, 而向量连接算法具有更好的 cache 局部性和内存访问性能, 在 NUMA 架构下具有趋势相近但幅度不同的性能特征. 在硬件方面, 不同 CPU NUMA 架构的差异对算法性能及 NUMA 优化技术的需求不同, 例如 ARM 和 AMD CPU 的跨 NUMA 访问延迟相对于 Intel CPU 较高, 不同代的 CPU 跨 NUMA 访问延迟也有较大的差异, 相同的 NUMA 优化技术对不同算法、不同 CPU 架构、不同的负载有不同的性能特征. 本文设计了 NUMA-conscious (NUMA 感知) 和 NUMA-oblivious (非 NUMA 感知) 外键连接算法库, 对不同的数据布局策略、不同的哈希表、向量创建方法、NUMA-conscious 协同分区 (co-partition) 方法等进行深入研究, 探索不同 NUMA 架构下的外键连接优化算法的性能特征. 本文的贡献主要体现在以下几个方面.

• NUMA-conscious 优化技术受多种硬件因素影响, 如不同 CPU 架构跨 NUMA 访问延迟、cache 大小和效率等. 本文对比了 5 个不同的 CPU 平台, 验证了不同 NUMA 优化技术在不同 CPU 平台上的性能特征, 提供了不同架构、不同硬件配置 CPU 平台上不同的 NUMA 优化算法选择策略, 为数据库查询优化器提供了第一手的连接算法性能数据.

• 研究揭示了不同外键连接算法对 NUMA 架构有不同的依赖强度. 如 NPO 算法对 NUMA 架构有强依赖性, PRO 算法对 NUMA 架构有中等依赖性, 向量连接算法对 NUMA 架构依赖性最弱.

• 提出一种新的连接基准. 与相关研究通常采用|R|=|S|或|S|=10×|R|的相对比例不同的是, 本文所提出的连接基准采用固定 S 表长度 (如|S|=SF×6M, 本文中设置 SF=200, 对应 12 亿条记录) 与 TPC-H 和 SSB 基准中事实表数据量相当, |R|大小设置为 2^5–2^{29} 行, 模拟代表性数据库基准 TPC-H、SSB、TPC-DS 中维表长度的变化范围, 覆盖 OLAP 负载中代表性的维表和事实表数据范围, 使连接操作性能可以映射到代表性基准查询中. 连接基准测试揭示了连接性能与 CPU 架构、cache 大小、NUMA 延迟的相关性, 实验结果表明, 良好的模式设计能够简化 NUMA 优化的复杂度, 提高连接性能.

实验结果给出如下结论: NUMA-conscious 数据布局能够更充分利用 NUMA 架构的内存带宽性能; 向量连接算法不仅具有 hardware-oblivious 硬件自适性能, 又具有 NUMA-oblivious 特性, 相对于 NUMA 延迟而言 cache 效率对性能有更显著的影响, 当跨 NUMA 访问延迟降低时, NUMA-conscious 优化收益降低; 通用的哈希连接算法受 NUMA 架构影响较大, 在 NUMA 架构上设计轻量 SN NUMA 系统 (shared-nothing NUMA) 能够更有效地降低跨 NUMA 访问延迟对性能的影响, 提高 NUMA 内存访问的局部性.

1 技术背景

本节首先介绍两个哈希连接算法和向量连接算法, 然后介绍不同 CPU NUMA 架构下的数据分区及连接基准设计.

1.1 连接实现技术

本文主要聚焦于两类连接算法: 哈希连接算法和向量连接算法. 哈希连接的 NPO 算法和 PRO 算法来源于文献 [1] 的源码, 向量连接算法基于 NPO 源码扩展设计. NPO 哈希连接算法包含两个微操作: 一是具有弱局部性特征的顺序表扫描操作 (外键表 S), 另一个是具有强局部性特征的哈希表操作 (主键表 R 上创建哈希表). 弱局部性的表扫描操作依赖内存带宽性能, 需要将数据分布在不同 NUMA 节点并由本地 NUMA 节点的 CPU 线程访问, 来提高内存访问局部性. 强局部性的哈希表操作既涉及 cache 局部性也涉及内存局部性, cache 局部性取决于哈希表相对于 L3 cache 大小. 当哈希表较小或 L3 cache 较大时, cache 的自动缓存机制保证哈希表访问具有较好的 cache 局部性, 受 NUMA 架构影响较小. 当哈希表超过 L3 cache 大小时会产生跨 NUMA 访问延迟, 优化策略需要平衡跨 NUMA 访问, 或者通过额外的分区操作来提高数据访问的 NUMA 局部性, 从而减少跨 NUMA 访问延迟. 如图 1

所示, NUMA-conscious 优化可以分为 NUMA 分区和 NUMA 协同分区两类: NUMA 分区策略将大表 (S) 按 NUMA 节点分区存储, 小表 (R) 根据处理线程动态分配内存页, 从而创建共享哈希表. 当哈希表探测时, S 表分片上的本地线程并行探测共享哈希表, 此策略只保证 S 表扫描的 NUMA 局部性, 无法保证哈希表创建与探测操作的 NUMA 局部性; NUMA 协同分区策略首先将 R 表和 S 表按 NUMA 节点数量使用同一哈希函数划分不相交的分区, 保证每对分区存储在相同 NUMA 节点内, 从而可以保证连接操作在 NUMA 分区内执行. 这种无共享数据分区策略可以获得最大的 NUMA 局部性, 其代价是动态数据分区的空间和时间开销, 或者依赖表存储模型的静态数据分区策略.

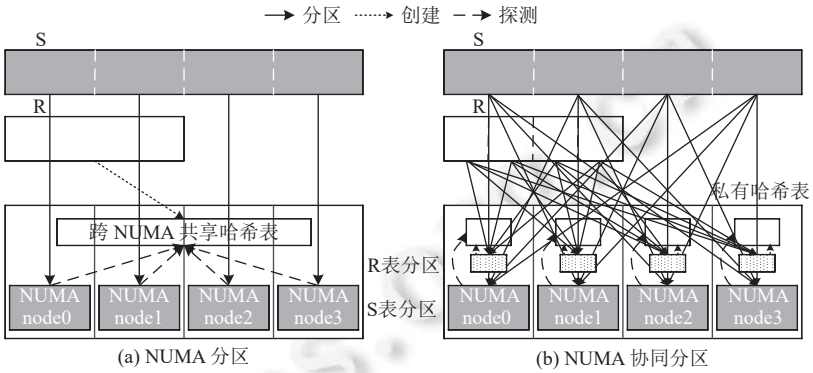
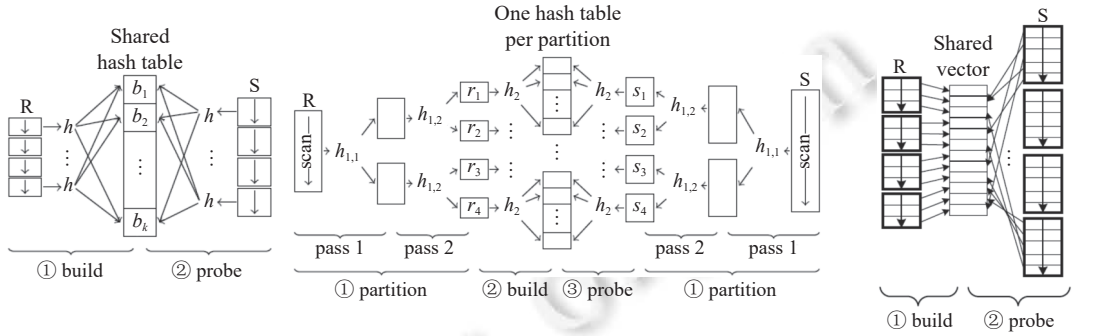


图 1 NUMA-conscious 分区策略

本文使用的 NPO 和 PRO 源码中缺省的执行模式作为 NUMA-oblivious 模式基准, 该模式采用随机内存分配模式, 在连接中产生远程 NUMA 访问延迟. --basic-numa 执行模式作为 NUMA-conscious 执行模式基准, 在 NUMA 节点平衡地分配内存. 该优化选项保证每个线程在相同的 NUMA 节点生成并扫描 R 表和 S 表数据, 创建 NPO 算法中共享哈希表和 PRO 算法中的分区. 该 NUMA-conscious 优化模式在动态生成数据集时隐式地由各线程分别生成 NUMA 节点的本地数据分片. 在真实内存数据库应用场景中 NUMA-conscious 分区策略通常由显式的静态数据 NUMA 分布存储策略和 NUMA-conscious 扫描操作来支持. NPO 算法需要创建跨 NUMA 节点的共享哈希表, 在探测阶段需要跨 NUMA 节点访问该哈希表, 可以在 NUMA 节点中复制 R 表创建私有哈希表来优化算法, 以保证哈希探测阶段的 NUMA 局部性, 但代价是增加了 R 表复制存储空间开销和创建多个哈希表的计算代价. NUMA-conscious PRO 算法的进一步优化[4]采用分块 (chunked) 并行 Radix 连接方法, 与原始的跨 NUMA 分区、NUMA 节点内创建哈希表、NUMA 节点内哈希探测的方法不同, 采用了 NUMA 内分区、跨 NUMA 创建哈希表和跨 NUMA 哈希探测策略. 该方法性能有 20% 的提升, 主要原因是将原始分区阶段跨 NUMA 随机写记录的代价转换为从远程 NUMA 节点分块顺序读代价, 降低了 NUMA 访问延迟. Radix 连接采用的是面向 cache 的动态分区哈希连接方法, 当采用面向 NUMA 节点的分区存储策略时, 动态分区简化为静态分区存储和 NUMA 节点内 Radix 连接执行, 能够较好地消除跨 NUMA 访问延迟, 将该优化技术融合于 NUMA 数据分区技术中, 因此 PRO 算法的 NUMA 优化技术本文不作进一步的研究与比较, 研究重点聚焦于对 cache 和 NUMA 局部性更加敏感的 NPO 算法和向量连接算法.

如图 2 所示, 基于哈希的 NPO 连接算法和 PRO 连接算法是通用的等值连接方法, 没有发挥数据库模式优化的优势, 如模式特征等. 向量连接算法则是面向 OLAP 数据库模式特征而设计的优化连接算法[3]. 向量连接算法在 R 表主键约束的基础上面向 OLAP 多维数据模型增加了地址映射约束, 使用连续的整数型代理键 (1, 2, 3,…) 作为 R 表主键, R 表创建向量代替传统的哈希表, 由 PAYLOAD 向量构成. 向量单元偏移地址作为默认的主键, 不需要显式存储, 消除了传统哈希表键值和溢出桶的存储开销. S 表外键直接用作连接索引, 将外键值映射为 R 表向量对应偏移地址单元完成连接操作, 消除传统哈希探测阶段的计算代价, 包括哈希值计算、哈希值比较和哈希记录遍历, 并且将 NPO 算法中的复杂哈希桶数据结构 (包含多个 R 表元组 (一个元组通常由一个主键值和一个

PAYLOAD 值组成) 和一个整数型值 (记录该哈希桶中元组的数量)) 设计简化为向量结构 (通常是一个整数型数组, 存储对应 R 表元组中的 PAYLOAD 值). 代理键是数据仓库和 OLAP 中普遍使用的技术, 如 TPC-H、SSB 和 TPC-DS 大部分表均使用了代理键作为主键. 向量连接算法可以看作是一种简化的 NPO 算法, 向量相对于哈希表更小 (如在 SSB 及 TPC-H 中, 向量可以压缩到 1 字节宽度), 具有更高的 cache 局部性. 向量连接算法性能优于传统的哈希连接算法[4,5] (也称为 array-join), 在内存效率方面优于 NPO 算法的哈希表, 也优于 PRO 算法额外的分区代价, 是一种内存利用率较高的 hardware-oblivious 连接算法. 因此向量连接算法也可以作为面向 OLAP 模式和负载的定制化连接优化技术.



(a) NPO 连接算法[1] (b) PRO 连接算法[1] (c) Vector Join 向量连接算法

图 2 NPO 连接算法、PRO 连接算法和向量连接算法示例

NUMA 节点通过 QPI 连接, QPI 带宽低于本地 NUMA 带宽, 跨 NUMA 数据访问延迟是 NUMA 优化的主要目标. 传统内存连接算法采用集中式设计, 在算法执行中优化 NUMA 本地访问, 当跨 NUMA 访问延迟足够高时, 需要针对 NUMA 节点的 SN (shared-nothing 无共享) 微集群做表存储的分布存储优化, 静态或动态地将连接表在 NUMA 节点上协同分片, 提高连接操作的 NUMA 局部性. 这种轻量 NUMA SN 集群并行连接优化策略是一种存储与计算一体化的优化设计, 需要在存储引擎和查询处理引擎进行协同优化.

1.2 CPU 架构及性能

CPU 架构和 NUMA 架构的差异化对连接优化设计有较大影响. 表 1 列出了 5 种代表性的 CPU 架构, 包括 Intel CLX、Intel ICX、AMD Zen2 和 AMD Zen3. CLX(28) 为 6 通道内存, 内存带宽低于其他 4 个 CPU 型号. AMD Rome 和 Milan CPU 采用小芯片 (chiplet) 架构, 每个小芯片中集成了 8 个核心, 其中, Rome Zen2 小芯片中每 4 个核心共享 16 MB L3 cache, Milan Zen3 小芯片中每 8 个核心共享 32 MB L3 cache, L3 cache 总容量均为 256 MB, 平均每核心 4 MB L3 cache slice. 与 AMD 架构不同, Intel x86 的两个型号 CLX(28) 和 ICX(38) 均采用 MESH 架构 L3 cache, 全部核心共享访问统一的 L3 cache 容量, 尽管 L3 cache 总容量低于 AMD CPU, 但核心有效 L3 cache 容量较 AMD 小芯片结构上的有效 L3 cache 容量更大. L3 cache 容量和缓存效率对 hardware-oblivious 的 NPO 算法和向量连接算法性能有较大的影响, 也间接影响了 NUMA 优化的效果.

表 1 代表性 CPU 架构及性能

CPU型号	类别	主频 (GHz)	核心	缓存	内存
华为鲲鹏920 7200	2代ARM v8	2.6	64	64 KB L1, 512 KB L2, 64 MB L3, 2×32 MB 1 MB/核	8通道, 带宽190.7 GB/s
Intel Xeon Gold 6258R	2代Cascade Lake (CLX)	2.7	28	32 KB L1, 1 MB L2, 38.5 MB L3	6通道, 带宽131.13 GB/s
AMD EPYC7742	2代Rome Zen2	2.24	64	32 KB L1, 512 KB L2, 256 MB L3, 8×32 MB 16 MB/4核	8通道, 带宽190.7 GB/s
Intel Xeon Platinum 8368	3代Ice Lake (ICX)	2.4	38	48 KB L1, 1.25 MB L2, 57 MB L3	8通道, 带宽200 GB/s
AMD EPYC 7763	3代Milan Zen3	2.45	64	32 KB L1, 512 KB L2, 256 MB L3, 8×32 MB 32 MB/8核	8通道, 带宽204.8 GB/s

除 CPU 和 L3 cache 架构差异外, 远程 NUMA 访问延迟也有较大差异. 我们模拟了 NUMA 内及远程 NUMA 哈希表访问 (预设 2^{29} 哈希桶) 操作来评估跨 NUMA 访问延迟, 如表 2 所示, 每个 ARM CPU 有 2 个 NUMA 节点, 其他 4 个 x86 每 CPU 有一个 NUMA 节点. 哈希表采用全表扫描, 更新操作模拟远程哈希表的读写操作, 以本地 NUMA 节点哈希表访问延迟作为基准延迟 (数值为 1), 远程 NUMA 访问延迟与本地基准访问延迟比率作为远程 NUMA 访问延迟的度量值, 该值越大则跨 NUMA 访问延迟越高. 总体而言, Intel x86 CLX(28) 和 ICX(38) 的远程 NUMA 访问延迟显著低于 AMD x86 Rome Zen2 和 Milan Zen3. ARM CPU 内部 NUMA 节点访问延迟较低, 但跨 CPU NUMA 节点的访问延迟较高, 最大延迟高于 AMD CPU.

表 2　不同平台的 NUMA 访问延迟

CPU架构	读 (NUMA0→目标节点) 延迟比率				写 (NUMA0→目标节点) 延迟比率			
	0	1	2	3	0	1	2	3
ARM(64)	1	1.24	1.91	1.76	1	1.18	1.75	1.64
CLX(28)	1	1.29	—	—	1	1.33	—	—
Rome Zen2(64)	1	1.94	—	—	1	1.74	—	—
ICX(38)	1	1.36	—	—	1	1.31	—	—
Milan Zen3(64)	1	1.73	—	—	1	1.58	—	—

综上所述, 不同 CPU 架构 NUMA 延迟的差异性使 NUMA 优化方法的性能收益和复杂度产生差异化, ARM CPU 具有多 NUMA 节点的特征, 这增加了 NUMA 优化算法的复杂性或数据分布代价. 但 CPU 内部 NUMA 延迟较低可以将 NUMA 逻辑合并, 从而达到与 x86 CPU NUMA 优化方法的兼容, 并且降低多 NUMA 节点上的数据分布或复制代价.

1.3 连接基准

连接是数据库重要的基础算子, 在数据仓库和 OLAP 应用中主要实现事实表与维表之间等值匹配连接, 其中事实表庞大且数据量持续增长, 维表较小且缓慢增长, 维表数据量相对事实表数据量占比较低 (通常维表数据量占总数据量的 1% 以下). 近年内存连接优化技术研究中主要模拟大表连接与小表连接性能, 如两表固定大小[1,6,7]或两表固定比例|R|=10×|S|和|R|=|S|[2,4,8], 这种固定负载固化了连接优化的覆盖范围, 不同的表大小及选择率都极大地影响连接性能特征 (如文献 [7] 中在 TPC-H 负载中的连接性能特性出现偏差). 另一方面, |R|=|S|或|S|=10×|R|的连接负载只关注了软件维度上连接表的大小关系, 没有考虑连接表大小与 CPU 的 LLC (last level cache, 最后一级 cache, 通常指 L3 cache) 的关系, 虽然 NPO 算法及向量连接算法在优化技术上的 hardware-oblivious 特性不需要面向 CPU 硬件参数配置调优, 但算法表现为 hardware-adaptive 特性, 即具有与 LLC 容量与效率自适应的硬件优化特征. 在设计连接负载时需要关注硬件技术的发展 (LLC 容量的持续增长) 与连接算法性能的相关性.

基于上述分析, 本文设计了 micro join benchmark (MJB), 主要从软件、硬件、应用这 3 个维度设计了连接负载, 可以更加全面测试 OLAP 负载中连接性能, 具体设计如下.

(1) R 表和 S 表采用主外键连接方法. R 表主键为连续整数代理键, 以 shuffle 乱序存储, 模拟异位更新场景及非聚集存储. S 表外键乱序存储.

(2) |S|=SF×6M, S 表模拟 TPC-H 和 SSB 中最大的事实表, SF 代表扩展因子.

(3)| R|= 2^5 – $2^{\log_2|S|}$, 取消|R|=|S|负载 (数据仓库等值连接的一一映射通常采用物化策略), 在相对 R 表大小上变化范围更加全面, 相对 LLC 大小的动态变化负载更优. 在绝对 R 表大小上覆盖了代表性的数据库基准 TPC-H、SSB 和 TPC-DS 中各连接表大小.

如本文实验中 SF=200, 模拟 12 亿行事实表上的连接操作, R 表变化范围从 2^5 – 2^{29} 行, 能够覆盖最大 17 TB SSB 数据集、3.5 TB TPC-H 数据集和 100 TB TPC-DS 数据集上的连接负载.

PRO 算法通过分区操作实现 L1/L2 cache 中的哈希连接操作, 对 LLC 不敏感. NPO 算法和向量连接算法通过

LLC 缓存哈希表/向量, CPU LLC 大小及访问性能对连接性能有直接的影响. 当前 CPU 发展的一个显著趋势是持续增大 LLC 大小, 如 ICX(38) 的 L3 cache 容量达到 57 MB, ARM(64) 达到 64 MB, AMD Zen2/Zen3 达到 256 MB, Fujitsu A64FX 集成了 32 GB 的 HBM2 以及 Intel Sapphire Rapids 集成 64 GB HBM2 扩大缓存容量. 前期研究固定的连接负载, 因而无法揭示连接性能受硬件技术发展的影响, 也无法体现在真实应用场景下连接负载的需求 (如通过物化 (SSB) 及冗余外键 (TPC-DS) 消除大表连接代价, 真实的维表缓慢及非线性增长等).

本文提出的 MJB 连接基准不仅用于评估不同连接算法的性能特征, 还能够进一步揭示不同连接算法性能与不同 CPU 架构之间的相关性. 随着晶体管集成度的持续提升, CPU LLC 容量增幅迅速, 连接算法因相对于不同 cache 层次的依赖性可能导致性能相对优势发生变化. MJB 连接基准不仅能客观全面地给出不同连接算法的性能特征, 还能进一步给出连接算法性能和硬件技术的相关性, 从硬件发展的维度预测未来连接算法性能的趋势, 为数据库提供更加全面的性能预测方法.

2 NUMA-conscious 连接算法设计

本节讨论 NUMA-conscious 连接算法实现技术, 在连接算法上聚焦于 NPO 算法和向量连接算法, 相对于 PRO 算法内存消耗更低, 而且可以更好地支持现代流水线处理和向量化处理模型, 在 OLAP 多表连接场景的查询处理中具有更大的应用价值.

2.1 NUMA-conscious 分区

NUMA 架构上理想的存储策略是将输入表均匀分布在各 NUMA 节点之间, 本地线程访问本地 NUMA 节点分区数据. 为避免 NUMA 节点内存分配的偏斜, 将 numactl 的内存分配策略设置为--interleave=all, 使操作系统在各 NUMA 节点之间采用轮询均匀分配内存. 这种分配策略适用于 NUMA-oblivious 算法, 在 NUMA-conscious 算法中可能存在线程跨 NUMA 节点的内存访问延迟.

NUMA-conscious 分区需要磁盘-内存一体化设计策略, 来保证表数据加载到内存时在 NUMA 节点间均匀分布. 图 3 显示了该分区策略, 原始表为行存储结构, 加载到内存转换为列存储结构, 即将行存储表划分为固定记录行数的行组 (row group) 并将行组转换为列存储结构, 不同的连接场景需要不同的 NUMA 分区策略.

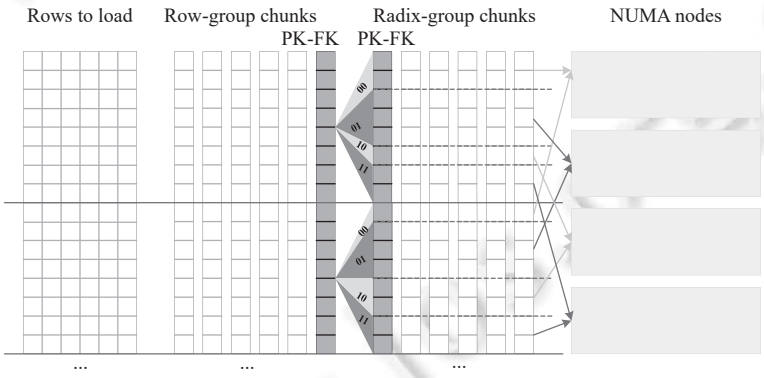


图 3 NUMA-conscious 分区方法

2.1.1 单表分区策略

当连接操作中 R 表很小而 S 表很大时, 可以采用简单的跨 NUMA 共享哈希表或共享向量连接算法, 此时仅需要将 S 表在 NUMA 节点间均匀分布, 最大化利用内存带宽性能. 这种 NUMA 分区场景适用于典型数据仓库和 OLAP 应用中的星形模型, 如 SSB 和 TPC-DS 中单一庞大的事实表和多个较小的维表模式. 在数据加载阶段, 事实表数据以列结构行组为粒度均匀分布在 NUMA 节点间, 同一 NUMA 节点的 CPU 线程访问行组列数据分片, 最大化 S 表扫描的 NUMA 局部性. 基于单表分区策略的跨 NUMA 共享哈希表或共享向量连接算法见算法 1.

算法 1. 基于单表分区策略的跨 NUMA 共享哈希表或共享向量连接算法.

数据预处理: 将 S 表水平切割成 N 份数据副本 (N 对应于 NUMA 节点数), 并分别存储在对应 NUMA 节点的内存中;
Input: R 表, S_k 表, $k \in [0, N)$, N 是 NUMA 节点数, 共享哈希表 (sht) 或共享向量 (sv);
Output: sum: 连接成功计数.

1. build 阶段
2. **for** $thread_k$ **in** all threads **do**
3. start ← $startoffset_k$ **in** R 表
4. i ← 1, location ← i + start
5. **repeat**
6. key ← R.tuple[location].key
7. index ← (Hash(key) OR key)
8. sht[index] ← R.tuple[location] OR sv[index] ← R.tuple[location].payload
9. i ← i + 1, location ← i + start
10. **until** i > R_$slice_length_k$
11. **end for**
12. probe 阶段
13. **for** $thread_k$ **in** all threads **do**
14. numaid ← $get\_numa\_id(thread_k)$
15. start ← $startoffset_k$ **in** S_{numaid} 表
16. i ← 1, location ← i + start
17. sum ← 0
18. **repeat**
19. key ← S_{numaid}.tuple[location].key
20. index ← (Hash(key) OR key)
21. sum ← sum + (sht[index].payload OR sv[index])
22. i ← i + 1, location ← i + start
23. **until** i > S_{numaid}_$slice_length_k$
24. **end for**
25. **return** sum

2.1.2 协同 NUMA 分区策略

当表 R 和 S 较大时, 远程 NUMA 访问增加了连接延迟, 可以通过协同 NUMA 分区策略将 R 表和 S 表按连接键 (R 表 PK 主键和 S 表 FK 外键) 划分为与 NUMA 节点数量相匹配的分区, 并将对应表分片存储在相同的 NUMA 分区中, 实现 NUMA 内 R 表和 S 表分区的本地化连接和跨 NUMA 分区数据的并行连接. 分区采用连接键 Radix 基数分区, 即通过连接键列数据末尾 n 位进行分区, 如 n=2 时将连接表划分为 2^n 个分区. 如图 3 所示, R 表和 S 表行组执行 n=2 的 Radix 分区, 相同行组中的记录划分为 4 个数据子集, 对应 4 个 NUMA 节点, R 表和 S 表具有相同 Radix 值的分区存储到相同 NUMA 节点中, 执行 NUMA 内的 R 表和 S 表分区连接操作. 当 NUMA 节点数量比分区数据大或小时, 如 2^2 分区在 2 或 4 个 NUMA 节点的平台上通过扩展或收缩 Radix 位数来扩展或合并 NUMA 分区.

协同分区策略适用于带有大表连接的应用场景, 如 TPC-H 中的 ORDERS⋈LINEITEM 或 PARTSUPP⋈LINEITEM 连接负载.

内存数据库在磁盘上的数据存储布局通常只考虑 I/O 访问效率, 在内存上存储布局则需要考虑在不同 NUMA 架构下通过行组内的 NUMA-conscious Radix 分区来提高数据访问的局部性.

2.2 NUMA-conscious 共享中间结果集复制策略

NPO 算法和向量连接算法使用共享哈希表和向量, 创建哈希表/向量及探测操作中会产生跨 NUMA 访问延迟, 为减少跨 NUMA 访问延迟, 设计了 3 个共享哈希表/向量复制策略, 通过在 NUMA 节点复制较小 R 表生成的哈希表/向量实现 NUMA 内的哈希/向量连接操作.

2.2.1 细粒度复制策略 (fine-grained replication)

如图 4(a) 所示, 每个 NUMA 节点中 R 表分区中的记录在各 NUMA 节点并发地创建哈希表/向量, 从而在探测阶段使每个 NUMA 节点中的 S 表分区记录可以执行 NUMA 内的哈希/向量连接. 该策略通过牺牲哈希表/向量创建阶段的跨 NUMA 写代价来消除探测阶段的跨 NUMA 读延迟, 适用于连接探测阶段的代价远高于创建阶段的应用场景. 基于细粒度复制策略的 NUMA 节点内私有哈希表或私有向量连接算法如算法 2 所示.

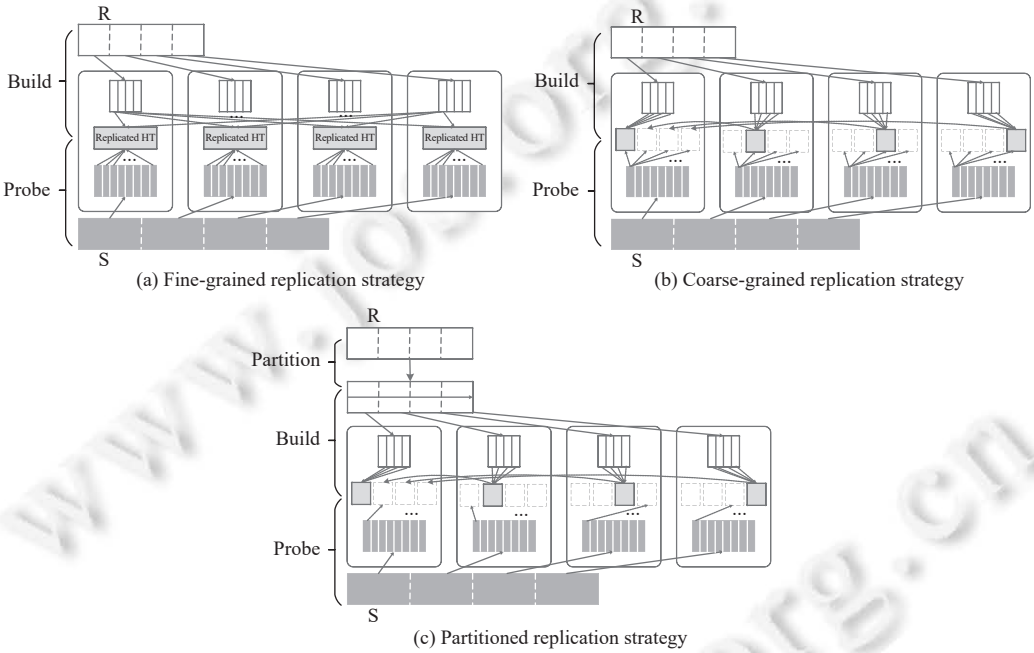


图 4 NUMA-conscious 连接实现技术

算法 2. 基于细粒度复制策略的 NUMA 节点内私有哈希表或私有向量连接算法.

Input: R 表, S 表, 私有哈希表 (pht_k) 或私有向量 (pv_k), $k \in [0, N)$, N 对应于 NUMA 节点数;
Output: sum: 连接成功计数.

1. build 阶段
2. **for** thread$_k$ **in** all threads **do**
3. start ← $startoffset_k$ **in** R 表
4. i ← 1, location ← i + start
5. **repeat**
6. key ← R.tuple[location].key
7. index ← (Hash(key) OR key)

8. **for** $j \leftarrow 1$ **to** N **do**
9. pht_j [index] ← R.tuple[location] **OR** pv_j [index] ← R.tuple[location].payload
10. **end for**
11. $i \leftarrow i + 1$, location ← i + start
12. **until** $i >$ R_slice_length$_k$
13. **end for**
14. probe 阶段
15. **for** $thread_k$ **in** all threads **do**
16. start ← $startoffset_k$ **in** S 表
17. $i \leftarrow 1$, location ← i + start
18. sum ← 0
19. **repeat**
20. key ← S.tuple[location].key
21. index ← (Hash(key) OR key)
22. numaid ← *get_numa_id*($thread_k$)
23. sum ← sum + (pht_{numaid} [index].payload **OR** pv_{numaid} [index])
24. **until** $i >$ S_$slice_length_k$
25. **end for**
26. **return** sum

2.2.2 粗粒度复制策略 (coarse-grained replication)

如图 4(b) 所示, 每个 NUMA 节点内的 R 表分区独立创建哈希表/向量, 然后将其复制到其他 NUMA 节点中, 从而提供多入口的哈希表/向量探测. 该策略创建 NUMA 局部的哈希表/向量, 通过粗粒度的复制降低跨 NUMA 写操作延迟, 探测阶段保证了 NUMA 访问的局部性, 但增加了多哈希表/向量探测代价, 适用于创建阶段跨 NUMA 写代价较大的连接负载. 基于粗粒度复制策略的 NUMA 节点内私有哈希表或私有向量连接算法如算法 3 所示.

算法 3. 基于粗粒度复制策略的 NUMA 节点内私有哈希表或私有向量连接算法.

Input: R 表, S 表, 私有哈希表 (pht_{k_m}) 或私有向量 (pv_{k_m}), k, $m \in [0, N)$, N 对应于 NUMA 节点数;
Output: sum: 连接成功计数.

1. build 阶段
2. **for** $thread_k$ **in** all threads **do**
3. start ← $startoffset_k$ **in** R 表
4. $i \leftarrow 1$, location ← i + start
5. **repeat**
6. key ← R.tuple[location].key
7. index ← (Hash(key) OR key)
8. numaid ← *get_numa_id*($thread_k$)
9. $pht_{\text{numaid\_numaid}}$ [index] ← R.tuple[location] **OR** $pv_{\text{numaid\_numaid}}$ [index] ← R.tuple[location].payload
10. $i \leftarrow i + 1$, location ← i + start
11. **until** $i >$ R_$slice_length_k$
12. **end for**

13. **for** $i \leftarrow 1$ **to** N **do**
14. **for** $j \leftarrow 1$ **to** N **do**
15. **if** $j = i$ **then**
16. **continue**
17. **end if**
18. $pht_{j\_i} \leftarrow pht_{i\_i}$ **OR** $pv_{j\_i} \leftarrow pv_{i\_i}$
19. **end for**
20. **end for**
21. probe 阶段
22. **for** $thread_k$ **in** all threads **do**
23. start $\leftarrow$ $startoffset_k$ **in** S 表
24. $i \leftarrow 1$, location $\leftarrow i$ + start
25. sum $\leftarrow 0$
26. **repeat**
27. key $\leftarrow$ S.tuple[location].key
28. index $\leftarrow$ (Hash(key) OR key)
29. numaid $\leftarrow$ $get\_numa\_id(thread_k)$
30. **for** $j \leftarrow 1$ **to** N **do**
31. **if** pht_{numaid_j} [index] ≠ NULL **OR** pv_{numaid_j} [index] ≠ NULL **then**
32. sum $\leftarrow$ sum + (pht_{numaid_j} [index].payload **OR** pv_{numaid_j} [index])
33. **Break**
34. **end if**
35. **end for**
36. **until** $i >$ S_$slice_length_k$
37. **end for**
38. **return** sum

2.2.3 分区复制策略 (partitioned replication)

如图 4(c) 所示, 该策略为向量连接的优化算法. 首先将 R 表记录按 PK 主键排序后分区, 每个分区主键值递增且不相交, 在各 NUMA 上独立创建连接向量, 再复制到其他 NUMA 节点. 在探测阶段每个 S 表分区上的记录的 FK 键值映射到单一向量入口进行探测. 该策略增加了创建向量阶段 R 表的排序代价. 基于分区复制策略的 NUMA 节点内私有哈希表或私有向量连接算法如算法 4 所示.

算法 4. 基于分区复制策略的 NUMA 节点内私有哈希表或私有向量连接算法.

数据处理: 将 R 表中各元组按 key 值升序排列并生成新数据副本 R_n;
Input: R 表, S 表, 私有哈希表 (pht_k) 或私有向量 (pv_k), $k \in [0, N)$, N 对应于 NUMA 节点数;
Output: sum: 连接成功计数.

1. build 阶段
2. **for** $thread_k$ **in** all threads **do**
3. start $\leftarrow$ $startoffset_k$ **in** R 表
4. $i \leftarrow 1$, location $\leftarrow i$ + start

```
5.    repeat
6.      key ← R_n.tuple[location].key
7.      index ← (Hash(key) OR key)
8.      numaid ← get_numa_id(thread_k)
9.      pht_numaid [index] ← R.tuple[location] OR pv_numaid [index] ← R.tuple[location].payload
10.     i ← i + 1, location ← i + start
11.   until i > R_slice_length_k
12. end for
13. for i ← 1 to N do
14.   for j ← 1 to N do
15.     if j = i then
16.       continue
17.     end if
18.     memcpy(pht_j + start_j, pht_i + start_j, R_slice_length_j)
19.   end for
20. end for
21. probe 阶段
22. for thread_k in all threads do
23.   start ← startoffset_k in S 表
24.   i ← 1, location ← i + start
25.   sum ← 0
26.   repeat
27.     key ← S.tuple[location].key
28.     index ← (Hash(key) OR key)
29.     numaid ← get_numa_id(thread_k)
30.     sum ← sum + (pht_numaid [index].payload OR pv_numaid [index])
31.   until i > S_slice_length_k
32. end for
33. return sum
```

综上所述, 为减少连接操作中跨 NUMA 访问代价, 需要权衡哈希表/向量创建阶段和探测阶段的代价. NUMA 分区内的创建或探测提高了 NUMA 访问的局部性, 但需要付出跨 NUMA 复制、多入口探测或数据预处理代价. 总体性能需要评估总体代价, 连接的综合性能受 NUMA 延迟、哈希/向量探测延迟、cache 效率等多种因素影响.

2.3 NUMA 集群连接实现技术

当跨 NUMA 访问延迟较高时, 可以将 NUMA-aware 存储与 NUMA-conscious 连接相结合, 即将 CPU 平台看作 NUMA SN (shared nothing, 无共享) 集群架构. 通过存储模型上 NUMA-aware 的优化策略实现 NUMA 计算的本地化, 通过存储引擎与计算引擎的协同设计 (co-design) 方法提高 NUMA-conscious 连接算法性能. 当两个连接表均为大表时, 采用基于 PK-FK 的连接键协同分区策略, 按 NUMA 节点数量 radix 基数分区, 将 R 表和 S 表分布在不同的 NUMA 节点上, 保证 R 表与 S 表相同的连接键在相同的 NUMA 分区, 实现 NUMA 分区上的本地化连接操作. 当 R 表非常小而 S 表较大时, S 表可以采用 Radix 分区、范围分区、分块分区等方法分布在不同的 NUMA 节点存储. R 表采用全复制方法创建各 NUMA 节点上的数据副本, 实现 NUMA 节点内 S 表数据子集上的并行连

接处理. 面向 NUMA 架构的存储策略需要数据库在存储引擎设计中支持 NUMA-aware 分区策略, 通过静态数据分布存储策略消除跨 NUMA 数据访问延迟, 通过原生设计的 NUMA-aware 存储引擎支持 NUMA-conscious 的查询处理引擎.

如第 2.1 节所述, 协同分区存储模型使用 PK-FK 上的 Radix 位进行 NUMA 分区, 创建不相交的 R 表和 S 表分区, 支持 NUMA 本地连接操作. 当存储引擎不支持 NUMA-aware 分区存储时, 需要将 NUMA 分区作为数据预处理, 付出额外的时间与存储空间代价. 该策略也适用于 PRO 算法, 通过 1+n 趟 Radix 分区首先使用 $\log_2$|NUMA node|位将 R 表和 S 表划分到不同的 NUMA 分区中, 然后在本地 NUMA 节点中执行 n 趟 Radix 分区, 实现 NUMA 节点内的分区和哈希连接操作.

典型的 OLAP 数据库采用单一事实表 (如 SSB), 通过事实表物化策略消除大表连接. 或者采用多事实表 (如 TPC-DS) 中冗余事实表外键来独立访问共享维表, 减少大事实表之间的连接操作. OLAP 数据库模式上的优化策略简化了 NUMA 集群上的并行连接算法设计. TPC-H 模式采用 3NF 结构, 有 3 个大事实表 LINEITEM, ORDERS 和 PARTSUPP, 面向 NUMA 集群进行协同分区时, 三表关联产生较大的数据冗余, 将 3 个事实表物化为 SSB 模式中的单一事实表 LINEORDER 可以优化 NUMA 集群连接算法, 但增加了冗余存储代价和计算代价 (如 LO_ORDTOTALPRICE 冗余存储, 需要去重计算). 一种更加灵活的策略是只将 PARTSUPP 表中用于度量计算的 SUPPLYCOST 列物化到 LINEITEM 表中, 消除 LINEITEM 表与 PARTSUPP 表之间利润计算时的连接需求. PARTSUPP 表可以与共享 PART 表和 SUPPLIER 表构成独立的星形模型进行 NUMA 分区, LINEITEM 表和 ORDERS 表采用协同 NUMA 分区, 使不同的连接负载具有较好的 NUMA 局部性.

综上所述, 从软件维度来看, 数据库的查询优化是一个系统性的问题, 计算层的设计与存储层密切关联, 模式层上的优化工作可以简化查询层的复杂度和效率. 从硬件维度来看, 传统的分布式系统以节点为粒度, 随着内存容量和处理器核心数量的持续增长, NUMA 节点数量也呈增长趋势, 如 AMD EPYC 9004 可以配置为 1 个或 4 个 NUMA 节点, 为 NUMA 访问延迟敏感算法提供了不同的硬件解决方案. NUMA 架构的多样性需要数据库在存储引擎和查询处理引擎上有灵活的软件架构相匹配, 传统集中式计算架构依赖动态分区策略, 从而带来较大的计算和存储空间开销. 将分布式系统节点间 SN 并行计算架构下沉到节点内, 实现 NUMA 集群并行计算架构, 从而能够更好地适应多 NUMA 节点处理器平台上的高效查询处理. 基于 NUMA 集群连接实现技术的私有哈希表或私有向量连接算法如算法 5 所示.

算法 5. 基于 NUMA 集群连接实现技术的私有哈希表或私有向量连接算法.

数据预处理: 对 R 表和 S 表使用 PK-FK 上的 Radix 位进行 NUMA 分区, 分割成 N 份数据副本 $R_{0...N-1}$, $S_{0...N-1}$ (N 对应于 NUMA 节点数), 并分别存储在对应 NUMA 节点的内存中;
Input: $R_{0...N-1}$ 表, $S_{0...N-1}$ 表, 私有哈希表 (pht_k) 或私有向量 (pv_k), $k \in [0, N)$, N 是 NUMA 节点数;
Output: sum: 连接成功计数.

```
1. build 阶段
2. for thread_k in all threads do
3.     numaid ← get_numa_id(thread_k)
4.     start ←startoffset_k in R_numaid 表
5.     i ← 1, location ← i + start
6.     repeat
7.         key ← R_numaid.tuple[location].key
8.         index ← (Hash(key) OR key)
9.         pht_numaid [index] ← R_numaid.tuple[location] OR pv_numaid [index] ← R_numaid.tuple[location].payload
10.        i ← i + 1, location ← i + start
```

```
11.    until i > R_numaid_slice_length_k
12. end for
13. probe 阶段
14. for thread_k in all threads do
15.    numaid ← get_numa_id(thread_k)
16.    start ← startoffset_k in S_numaid 表
17.    i ← 1, location ← i + start
18.    sum ← 0
19.    repeat
20.       key ← S_numaid.tuple[location].key
21.       index ← (Hash(key) OR key)
22.       sum ← sum + (pht_numaid [index].payload OR pv_numaid [index])
23.    until i > S_numaid_slice_length_k
24. end for
25. return sum
```

2.4 NUMA 节点合并策略

与 x86 架构不同, ARM 处理器包含两个 NUMA 节点, 相同芯片内跨 NUMA 节点访问延迟远低于不同芯片间跨 NUMA 访问延迟. NUMA 节点数量决定了不同 NUMA 节点上的复制及动态数据分区代价, 将芯片内 NUMA 节点合并使 NUMA 连接算法设计与 x86 保持一致, 减少算法设计的复杂度, 也可以极大地优化连接性能.

3 NUMA-conscious 连接实现技术

本文对比了 13 种 NUMA 优化连接算法, 深入探索不同的连接优化技术在不同 CPU 架构上的性能特征, 通过性能分析为不同 NUMA 架构 CPU 平台的连接算法优化选择提供参考.

3.1 基准连接实现技术

基准连接算法为 NPO 算法、PRO 算法和向量连接算法. 默认的算法不考虑 NUMA 优化, 由操作系统分配线程资源进行数据处理, 可能产生线程与数据所在 NUMA 点不一致的情况, 产生跨 NUMA 访问延迟. 算法提供了 NUMA 优化选项--basic-numa, 用于绑定线程访问相同 NUMA 节点内的数据, 降低跨 NUMA 访问延迟. 两种 NUMA 选项验证连接算法在 NUMA-oblivious 和 NUMA-conscious 模式下的性能差异, 可以进一步揭示不同连接算法对 NUMA 优化的敏感性以及不同 CPU 架构上 NUMA 优化对连接算法性能的影响.

3.2 NUMA-conscious 连接算法汇总

表 3 显示了不同的连接算法. 除 3 个基准连接算法外, 其余算法均为本文面向 NUMA 架构设计的不同优化算法实现, 覆盖了不同跨 NUMA 复制粒度和 NUMA 布局优化技术, 与 MJB 连接微基准结合给出完整负载下的连接性能曲线, 可以量化对比任何算法在任一平台任何负载下具体的性能指标.

FRHNPO 连接算法为每个 NUMA 节点创建本地哈希表, 所有线程将记录并发地插入多个 NUMA 节点的哈希表中, 使得在哈希表创建阶段跨 NUMA 节点的细粒度记录写操作延迟和哈希表并发访问控制延迟增加, 保证在探测阶段每个 NUMA 节点只访问本地哈希表.

FRVVJ 连接算法在创建阶段在每个 NUMA 节点创建连接向量, 与 FRHNPO 不同之处在于每个 R 表记录映射到唯一的向量位置, 消除了哈希表记录创建时的并发访问延迟, 细粒度跨 NUMA 节点的向量写操作增加了向量创建延迟. 探测阶段实现本地化向量连接操作.

CRVJ 连接算法采用粗粒度复制方法, 在创建阶段每个本地 NUMA 线程首先在本地 NUMA 节点创建连接向量, 然后将整个连接向量复制到其他 NUMA 节点上. 在探测阶段, 每个 S 表记录探测本地 NUMA 节点中的多个连接向量完成连接操作. CRVJ 算法通过粗粒度复制优化了跨 NUMA 访问延迟, 但在探测阶段增加了多次向量探测延迟.

表 3 NUMA-conscious 连接算法描述

算法分类	算法缩写	算法描述
基准连接算法 (默认和--basic-numa优化选项)	NPO	无分区哈希连接算法, no-partitioning hash join (with --basic-numa option)
	PRO	Radix分区哈希连接算法, Radix join (with --basic-numa option)
	Vector Join	向量连接算法, vector index lookup based join (with --basic-numa option)
NUMA-conscious连接算法	FRHNPO	细粒度哈希表复制NPO算法, fine-grained replicated hash table based no partition hash join
	FRVVJ	细粒度复制向量连接算法, fine-grained replicated vector based vector join
	CRVJ	粗粒度复制向量连接算法, coarse-grained replicated vector join
	SVCRVJ	基于排序的粗粒度复制向量连接算法, sorted vector based coarse-grained replicated vector join
NUMA SN集群并行连接算法	PSNNPO	基于分区的NUMA SN集群NPO连接算法, partitioned shared-nothing NPO
	PSNVJ	基于分区的NUMA SN集群向量连接算法, partitioned shared-nothing vector join
	RRNPO	基于R表全复制的NUMA SN集群NPO连接算法, replicated R table NPO
NUMA合并连接算法	FRHNPONM	NUMA合并细粒度哈希表复制NPO算法, fine-grained replicated hash table based no partition hash join for NUMA merge
	FRVVJNM	NUMA合并细粒度复制向量连接算法, fine-grained replicated vector based vector join for NUMA merge
	CRVJNM	NUMA合并粗粒度复制向量连接算法, coarse-grained replicated vector join for NUMA merge

SVCRVJ 算法是一个定制化的向量连接优化技术. 在算法的数据生成器中 R 表主键为连续代理键, 但乱序存储, 每个 NUMA 节点上 R 表数据分片创建的向量都覆盖全部的连接向量数值范围, 因此产生连接向量冗余创建和访问代价. SVCRVJ 算法首先对 R 表按连接键排序, 数据均匀分布到不同的 NUMA 节点形成连续且不相交的数据子集, 然后 NUMA 节点内的线程创建本地连接向量, 连接向量复制到其他 NUMA 节点构成物理独立逻辑连接的连接向量, NUMA 节点内的 S 表记录根据键值映射到唯一的连接向量分片上完成向量连接操作. 相对于 CRVJ 算法, SVCRVJ 算法通过 R 表有序的连接键消除了 S 表探测多个连接向量的代价. 数据库系统通常为主键创建聚集索引, 采用 SVCRVJ 算法时可以消除预排序代价, 降低创建连接向量代价.

3.3 NUMA SN 集群连接实现技术

PSNNPO 算法将 R 表和 S 表按主键和外键 Radix 位划分为 2^n 个分区, 分布式存储在 2^n 个 NUMA 节点上. 每个 NUMA 节点作为轻量 SN 集群, 执行本地 R 表和 S 表分片上的 NPO 连接算法. PSNVJ 算法与 PSNNPO 算法类似, NUMA 分区后执行 NUMA 节点间并行的向量连接算法. RRNPO 算法仅对 S 表 NUMA 分区, R 表采用全复制方法冗余存储在各 NUMA 节点上, 适合于 R 表较小的连接负载.

3.4 NUMA 节点合并连接实现技术

FRHNPONM 算法和 FRVVJNM 算法是应用于 ARM 平台的 NUMA 节点合并连接算法. 以 FRHNPO 和 FRVVJ 连接算法为基础, 将一个物理芯片上的两个逻辑 NUMA 节点合并为一个逻辑 NUMA 节点, 降低跨 NUMA 并发写操作数量. CRVJNM 算法是面向 ARM 平台定制化的 CRVJ 算法, 通过合并 NUMA 节点, 可将粗粒度连接向量的复制和探测的数量减少 50%, 其代价是在创建阶段增加了一部分芯片内跨 NUMA 访问延迟. ARM 和新一代 AMD CPU 支持片上多 NUMA 节点, 基于 NUMA 节点的 SN 并行连接算法, 所需的 NUMA 节点间复制的内存开销和计算代价更高. 该算法用于评估是否可以通过减少逻辑 NUMA 节点数量, 从而获得更高的存储空间利用率

和性能收益.

4 NUMA-conscious 连接性能评估

我们在实验部分测试了不同 NUMA-conscious 连接算法性能. 首先测试无 NUMA 优化和使用--basic-numa 优化选项的 NPO 算法、PRO 算法和向量连接算法性能作为 NUMA-oblivious 和 NUMA-conscious 连接算法基准性能, 对比不同架构 CPU 平台上连接算法对 NUMA 优化的性能收益和 NUMA 硬件敏感性. 通过 NUMA-conscious 连接算法性能测试揭示不同连接算法实现技术在不同 CPU 架构下的性能特征, 为数据库面向不同 CPU 架构的连接算法优化选择提供依据.

4.1 硬件配置

实验使用了 5 台双路服务器, CPU 配置为具有代表性的 ARM、Intel 和 AMD 架构的 5 个处理器 (如表 1 所示). 实验结果不仅能够展现 3 种不同架构 CPU 之间的性能特征与差异, 也能够展现相同架构不同代 CPU 之间的性能变化, 展现 CPU 硬件技术发展对连接性能带来的影响. 配置 Intel 和 AMD 处理器的服务器运行 CentOS Linux (内核版本为 4.18.0) 操作系统, ARM 处理器系统运行 Ubuntu Linux (内核版本为 4.19.9), gcc 版本为 7.3.1. 内存超过 512 GB, 支持大数据内存连接测试负载. 测试时首先对连接负载进行预加载, 选择 5 次连续连接执行时间中去掉预热执行时间后最短时间作为代表性的内存连接运行时间, 消除数据加载等系统开销对连接算法性能的影响 (连续执行时间相对稳定, 取最短时间最小化系统其他负载干扰).

4.2 基准连接性能

图 5 显示了 NPO 算法、PRO 算法和向量连接算法的基准性能. 虚线表示无 NUMA 优化的连接算法性能, 实线表示使用--basic-numa 选项的 NUMA 优化连接算法性能, 面积图表示 NUMA 优化连接算法相对无 NUMA 优化连接算法性能提升百分比. 通过性能测试观察到如下规律.

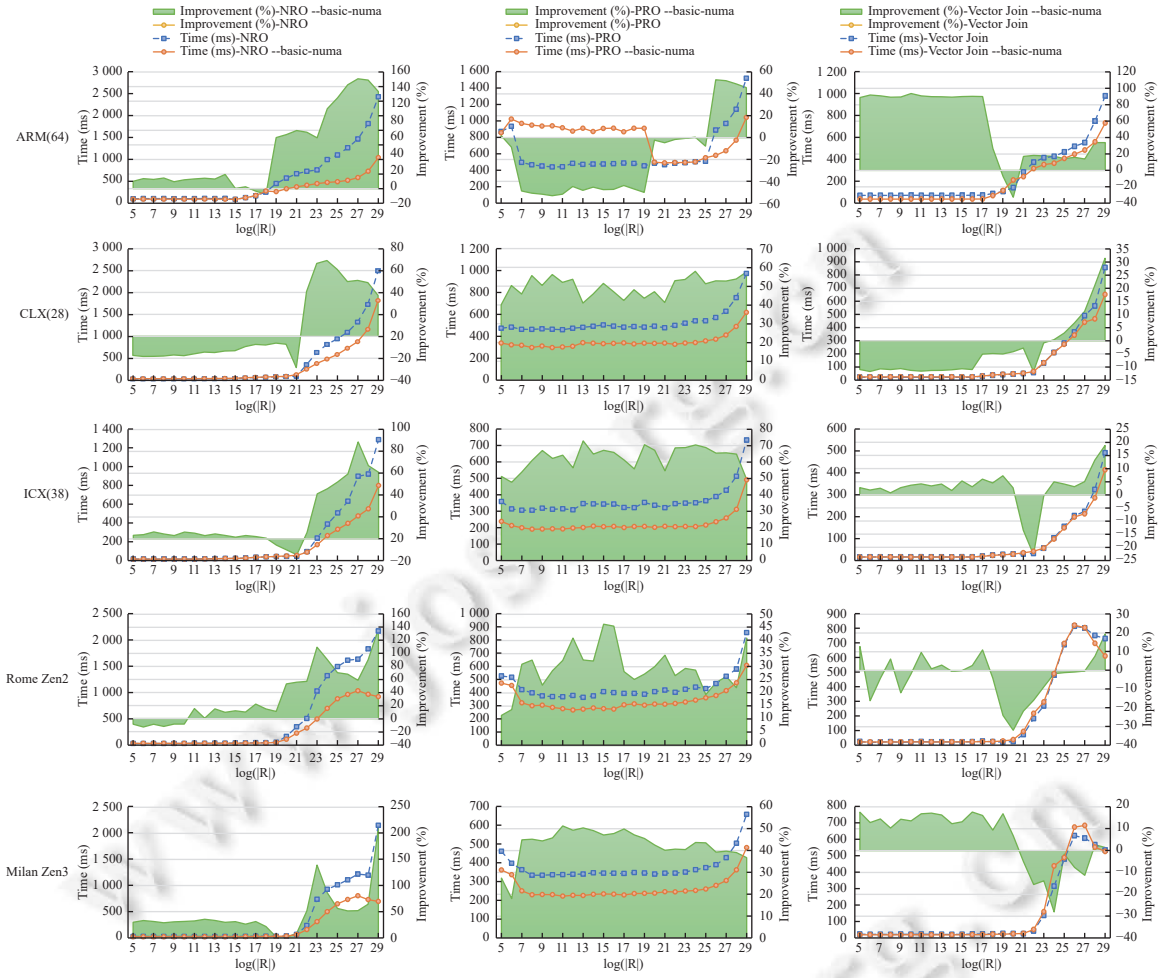
• NPO 连接算法性能对 NUMA 架构敏感度较高, NUMA 优化收益较其他两个连接算法更高. 当 R 表记录少于 2^{20} 行时, 连接性能主要受 cache 的影响, NUMA 优化对性能的影响较小. 当 R 表较大时产生大量内存哈希表访问, NUMA 延迟对连接的性能影响幅度逐渐增长. CXL(28) 平台上 NUMA 优化与无 NUMA 优化性能差异最小, ICX(38)、Milan Zen3 平台上性能差异相似, Rome Zen2 较 Milan Zen3 有更大的性能差异, ARM(64) 的性能差异最大. 不同 CPU 平台 NUMA 优化性能差异反映了表 2 跨 NUMA 访问延迟的规律, 说明 CPU 跨 NUMA 访问延迟性能对数据库连接算法性能具有直接的影响.

• NUMA 优化与无优化 PRO 算法之间的差异相对 NPO 算法较小, NUMA 优化和无 NUMA 优化算法性能曲线相对平滑且差异较为稳定. PRO 算法使用基于私有 cache 的分区连接优化技术, 无法使用 L3 cache 的优化资源消减小表连接时的 NUMA 延迟. 从性能曲线来看, AMD 的 Milan Zen3 和 Rome Zen2 的性能曲线较 Intel CLX(28) 和 ICX(38) 更加接近, 性能差异较小.

• 向量连接算法在 NUMA 优化和无 NUMA 优化模式下的性能差异最小, 算法性能对 NUMA 架构敏感度最低, 呈现出 NUMA-oblivious 性能特征. x86 的 4 个 CPU 平台上向量连接 NUMA 优化与无优化算法性能差异较小, ARM 平台两种算法性能差异相对较大, 反映了 ARM 平台跨 NUMA 访问延迟较大, 对向量连接算法性能产生影响. 相对于 NPO 算法, 向量连接算法使用更小的向量代替哈希表, 向量探测较哈希探测有更低的延迟和 CPU cycle 消耗. 在性能曲线上与 NPO 算法相似, 但相对 NPO 算法有更高的 L3 cache 利用效率, 即使在向量超过 L3 cache 大小时仍有较低的访问延迟, cache 效率对向量连接算法性能的影响超过 NUMA 访问延迟.

综上所述, NUMA 延迟性能主要影响内存访问阶段, 在 PRO 算法中分区阶段涉及跨 NUMA 读写访问延迟, 可以通过 NUMA 节点内分区缓存批量写操作, 来进一步降低 NUMA 访问延迟. 在探测阶段则通过 in-cache 探测消除 NUMA 访问延迟. NPO 算法和向量连接算法是 hardware-oblivious 算法, 即采用共享哈希表/向量的硬件自适应连接算法, 但在 NUMA 优化性能上有较大的差异. NPO 算法是 NUMA 性能高敏感型连接算法, 而向量连接算

法则是 NUMA 性能低敏感型连接算法. 向量连接算法较高的 cache 效率和较低的 NUMA 敏感性可以降低 NUMA 优化需求, 通过较低的算法复杂度达到较高的性能.



(a) NPO 连接算法 (b) PRO 连接算法 (c) Vector Join 向量连接算法

图 5 ARM(64)、CLX(28)、ICX(38)、Rome Zen2 和 Milan Zen3 CPU 平台上连接算法基准性能

NPO 算法、PRO 算法和向量连接算法在 5 个平台的连接性能如图 6 所示. ARM(64) 平台上 PRO 算法没有 NPO 算法在 4 个 x86 平台所表现出来的性能优势. 当 R 表较小时, NPO 算法与向量连接算法性能差异较 x86 平台更大. 当 R 表较大时, NPO 算法性能与向量连接算法性能差异较其他 4 个 x86 平台更小. 在 CLX(28) 和 ICX(38) 平台上, 当 R 表记录超过 2^{23} 行时, PRO 算法超过 NPO 算法性能, 随 R 表记录行数的增长性能差异进一步增大. 在 R 表行数低于 2^{20} 时, 向量连接算法与 NPO 算法性能相近. 当 R 表行数增长时性能优势增大, 并且在较大 R 表连接负载时也优于 PRO 算法. 向量连接算法具有较高内存利用率 (无分区内存开销)、算法复杂度较低且性能收益较高. 与 ARM(64) 类似, NPO 算法和向量连接算法在 Rome Zen2 和 Milan Zen3 平台上与在 CLX(28) 和 ICX(38) 平台上相比性能差异更小, 并行计算收益可以在一定程度上抵消向量连接 cache 延迟更低的性能优势. PRO 算法在 R 表记录超过 2^{23} 时超过向量连接算法性能, 相对 NPO 更早获得性能优势. Rome Zen2 和 Milan Zen3 采用 chiplet 上较小 L3 容量 (Rome Zen2 每 4 核共享 16 MB L3 cache, Milan Zen3 每 8 核共享 32 MB L3 cache), L3 cache 延迟与 CLX(28) 和 ICX(38) 的集中式 L3 cache 延迟更低. R 表行数在 2^{16}–2^{20} 之间时, Rome Zen2 的向量连接算法性能超过 CLX(28), R 表行数在 2^{18}–2^{21} 之间时, Milan Zen3 向量连接算法性能超过 ICX(38) 平台, R 表行数为 2^{29} 时 Rome

Zen2 和 Milan Zen3 向量连接算法性能超过 CLX(28) 但低于 ICX(38), 向量连接算法在 Intel 架构 CLX(28) 和 ICX(38) 上比在 AMD 架构 Rome Zen2 和 Milan Zen3 上有更大的性能优势区间.

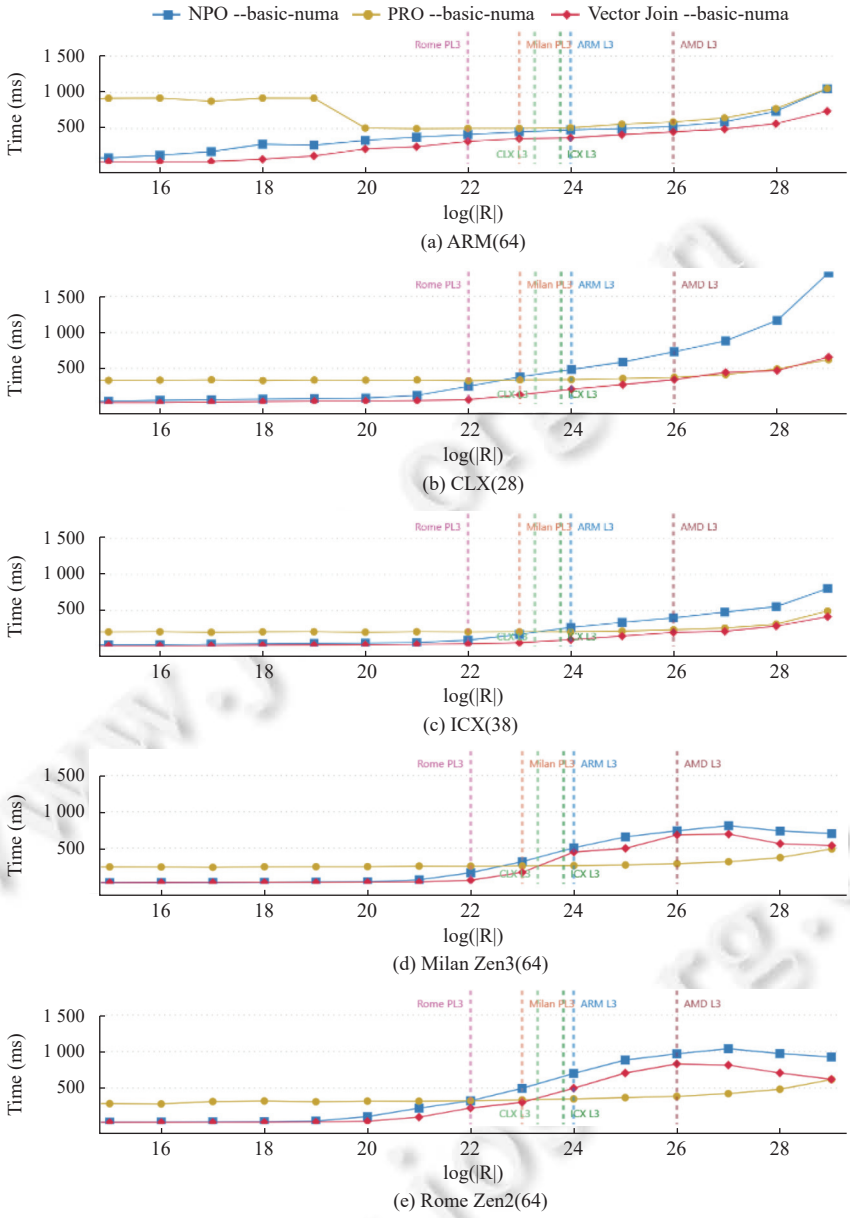


图 6 ARM(64)、CLX(28)、ICX(38)、Milan Zen3、Rome Zen2 平台 NUMA 优化 NPO、PRO 和向量连接算法性能

4.3 NUMA-conscious 连接性能

NUMA-conscious NPO 连接算法性能如图 7 所示. FRHNPO 连接算法为每个 NUMA 节点并发地创建哈希表, R 表较小时, FRHNPO 连接算法在各 CPU 平台算法综合性能低于 NPO 算法和 NPO --basic-numa 实现技术. 当 R 表记录增长时, FRHNPO 算法性能在 ARM(64), CLX(28) 和 ICX(38) 平台上性能仍然低于 NPO 算法和 NPO --basic-numa 实现技术, 但在 Rome Zen2 和 Milan Zen3 平台上优于 NPO 算法性能. 在中等大小数据集上性能与

NPO --basic-numa 实现技术相近.

NUMA-conscious 向量连接算法性能如图 8 所示. 在 ARM(64), CLX(28) 和 ICX(38) 平台上, 当 R 表记录低于 2^{26} 行时, FRVVJ 算法性能优于 Vector Join 向量连接算法[9]和 Vector Join --basic-numa 优化算法. 当 R 表记录增大时, FRVVJ 性能逐渐低于 Vector Join 算法和 Vector Join --basic-numa 优化算法, ARM(64) 平台上的性能差距高于 CLX(28) 和 ICX(38) 平台, 性能差距的大小主要受跨 NUMA 访问延迟的影响. 在 Rome Zen2 和 Milan Zen3 平台上, 当 R 表行数超过 2^{28} 时, FRVVJ 算法超过 Vector Join 算法和 Vector Join --basic-numa 优化算法. AMD 平台具有较大的跨 NUMA 访问延迟, FRVVJ 算法通过创建多个 NUMA 本地向量, 来降低跨 NUMA 向量访问延迟, 因此 FRVVJ 算法的性能更优. 在 Rome Zen2 和 Milan Zen3 平台上, Vector Join 算法性能上的差距较 FRVVJ 算法更小, 当 CPU 的跨 NUMA 访问延迟减少时, FRVVJ 性能优化收益有降低的趋势.

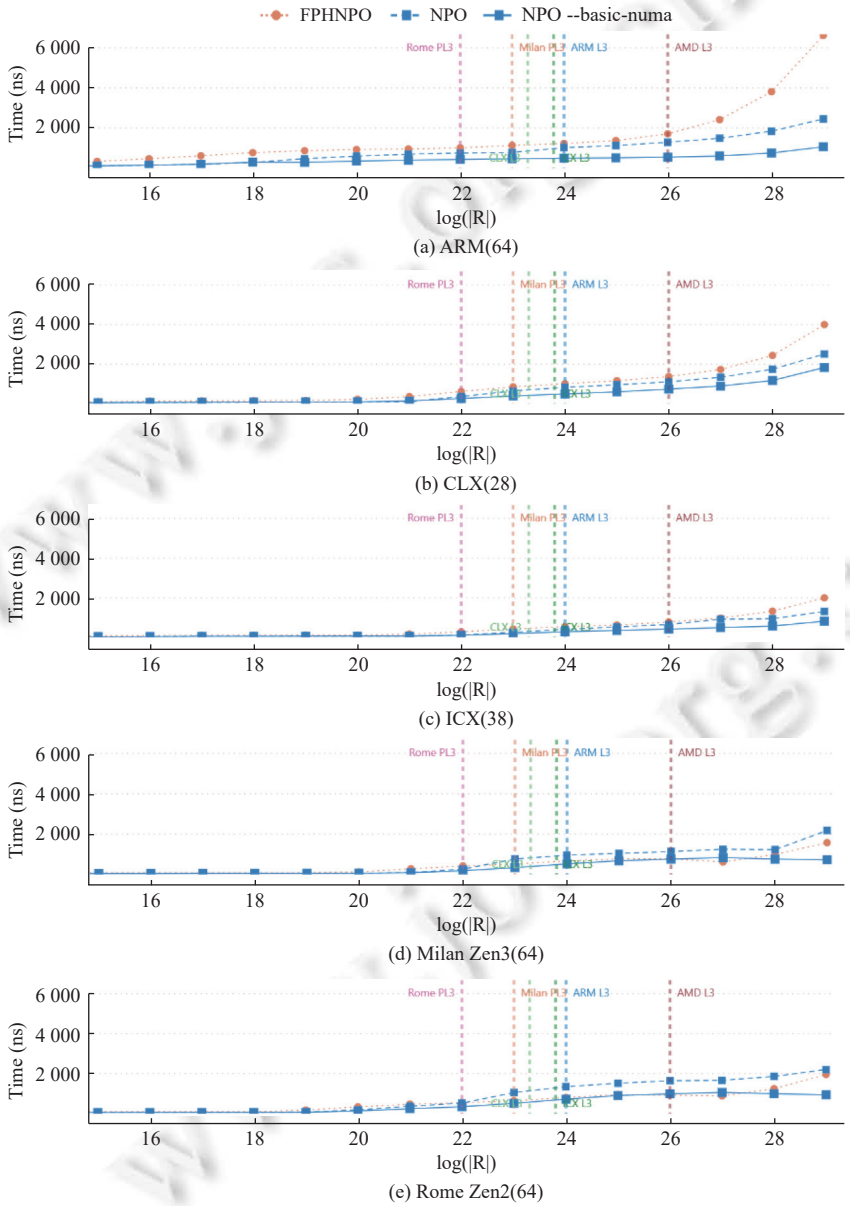


图 7 NPO 和 FRHNPO 连接算法性能比较

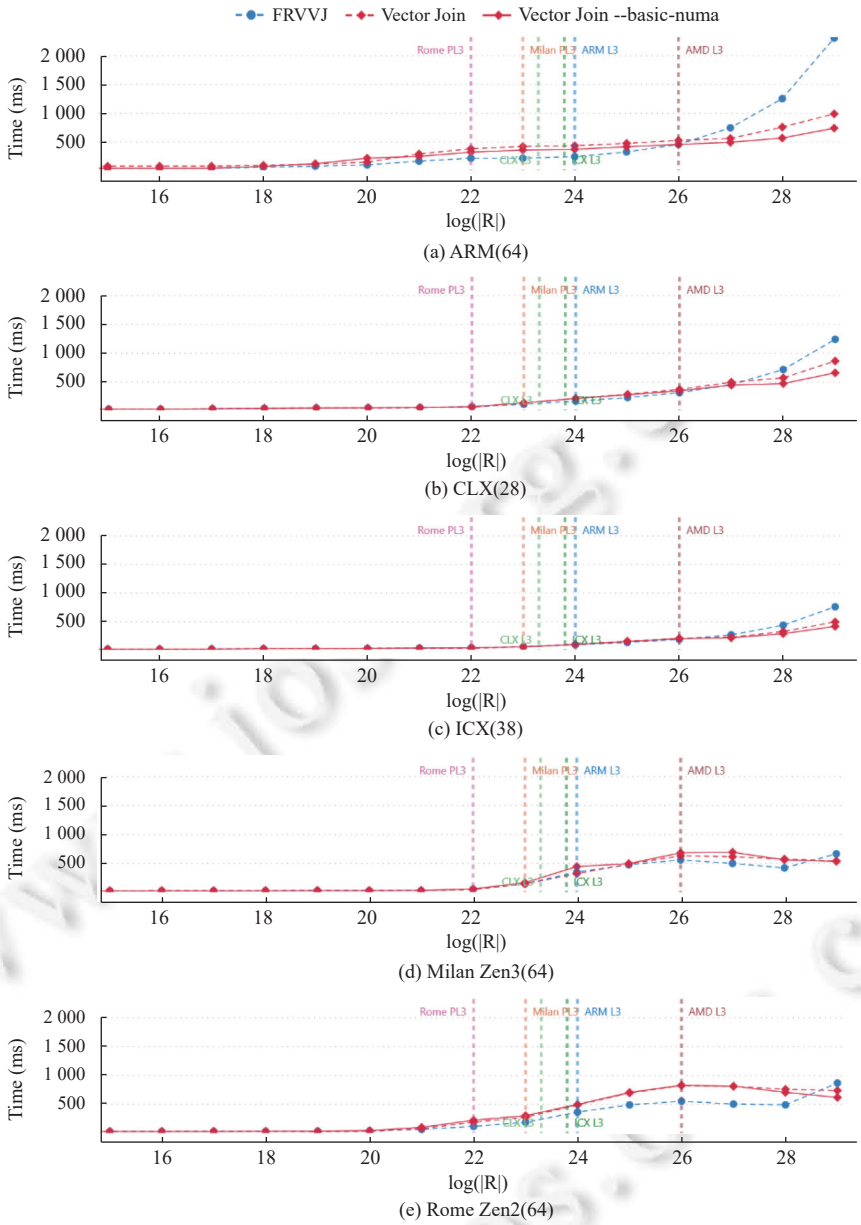


图 8 向量连接和 FRVVJ 连接算法性能比较

图 9 对比了细粒度向量复制连接算法 FRVVJ, 粗粒度向量复制连接算法 CRVJ 和基于排序的粗粒度向量复制连接算法 SVCRVJ 的性能, 各连接算法性能在不同 CPU 平台上呈现出不同的性能特征.

在 ARM(64) 平台上, 当 R 表记录数低于 2^{22} 行时, FRVVJ 算法与 SVCRVJ 算法性能相近, 但与 CRVJ 算法性能有较大差异. 当 R 表记录数量超过 2^{26} 行时, CRVJ 算法性能优于 SVCRVJ 算法, 且 SVCRVJ 算法与 FRVVJ 算法性能之间的差异较其他 4 个 x86 平台更大, 其主要原因是 ARM 平台具有较大的 cache 和内存访问延迟对性能产生影响.

在 CLX(28) 和 ICX(38) 平台上, 当 R 表记录数量低于 2^{24} 时, FRVVJ 算法和 SVCRVJ 算法的性能相近, 而 CRVJ 算法执行时间约为 FRVVJ 算法和 SVCRVJ 算法的 2 倍. 主要原因是当 R 表较小时, 连接性能的主要影响因素为向量探测延迟, CRVJ 算法的多入口探测延迟对连接性能影响较为显著. 随着 R 表增大, SVCRVJ 算法执行时

间逐渐增大, 而 CRVJ 算法执行时间逐渐向 FRVVJ 算法靠拢, 与 SVCRVJ 算法性能差距逐渐加大, 显示了大表排序代价影响权重逐渐增加而内存访问延迟影响权重逐渐降低的趋势.

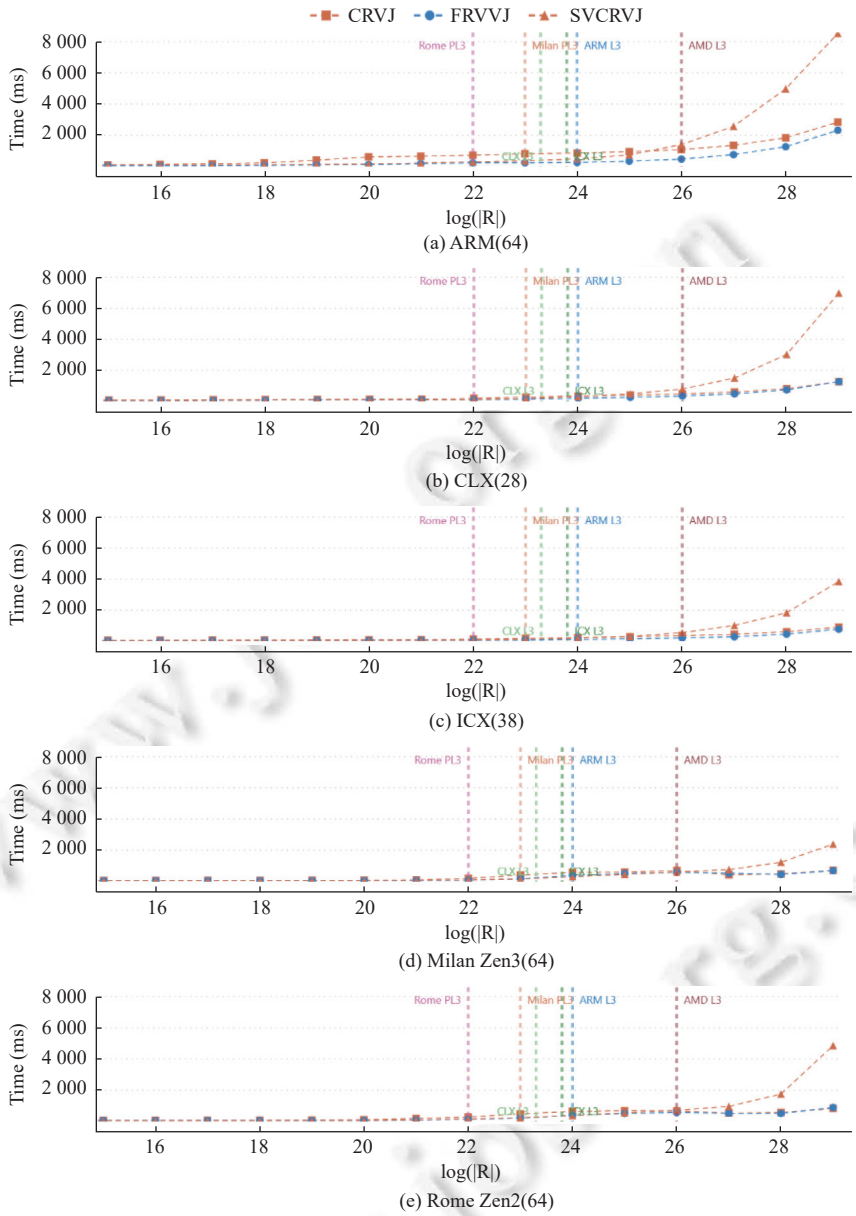


图 9 FRVVJ, CRVJ 和 SVCRVJ 连接算法性能比较

在 Rome Zen2 和 Milan Zen3 平台上, 当 R 表记录数量低于 2^{26} 行时, SVCRVJ 算法与 FRVVJ 算法性能接近, 当 R 表增大时性能逐渐拉开差距. 在 Rome Zen2 和 Milan Zen3 平台上 FRVVJ 算法和 CRVJ 算法的性能差距较 CLX(28) 和 ICX(38) 平台上的性能更低, 两代 CPU 的核心性能差异更小.

综上所述, NUMA-conscious 的局部哈希表/向量优化策略通常只有较小的优化区间, AMD 的 Rome Zen2 和 Milan Zen3 平台上优化区间相对较大. 相对简单的细粒度复制优化策略比粗粒度复制和基于排序的粗粒度复制策略的综合性能更高, 探测阶段的访问延迟对性能的影响高于创建阶段.

4.4 NUMA SN 集群连接性能

NUMA SN 集群连接算法提供了两个 NPO 算法和一个向量连接算法的实现技术, 采用动态数据分布技术提高 NUMA 访问局部性. 当与 NUMA-aware 存储技术相结合时, 采用静态 NUMA 分区技术可以消除查询的数据分区代价, 进一步提高连接性能收益.

如图 10 所示, PSNNPO 算法将 R 表和 S 表在 NUMA 节点间按连接键值分区存储以提高连接操作的 NUMA 数据访问局部性. 当 R 表较小时, PSNNPO 算法性能低于 NPO --basic-numa 算法. 当 R 表较大时, PSNNPO 算法可以有效地消除跨 NUMA 访问延迟, 性能优于 NPO --basic-numa 算法. 两个算法性能曲线的交汇点由 L3 cache 的效率决定, 即当哈希表有较高 cache 局部性时, NPO --basic-numa 算法性能优于 PSNNPO 算法, 反之则 PSNNPO 算法优于 NPO --basic-numa 算法. 在性能曲线交汇点的横坐标上 ARM(64)<Rome Zen2<Milan Zen3<CLX(28)<ICX(38), 反映了各 CPU L3 cache 的有效容量的特征. RRNPO 算法将 R 表复制到各 NUMA 节点, 实现 NUMA 本地化的连接操作. 全复制策略消除了 S 表分区代价但增加了 R 表冗余存储代价, 相对 PSNNPO 算法而言也增加了连接探测代价. 从性能曲线特征来看, RRNPO 算法性能仅在最大表连接时 (ARM (64) 平台$|R|>2^{27}$, CLX (28) 平台, ICX (38) 平台, Rome Zen2 和 Milan Zen 平台$|R|>2^{28}$) 性能低于 NPO --basic-numa 算法. 此时, 相同 CPU 系统中 3 代 CPU 与 2 代 CPU 性能差距更小, 其主要原因在于全复制策略增强了哈希表探测的 NUMA 内存访问局部性, 并且仅能通过本地 NUMA 节点的 cache 访问完整的 R 表的哈希表, NPO --basic-numa 算法的哈希表更小 (1/|NUMA 节点数量|×|R 哈希表|) 且可以利用多个 NUMA 节点的 cache.

图 11 显示了基于 NUMA 分区的向量连接算法 PSNVJ 算法和 Vector Join --basic-numa 基准算法的性能对比曲线图. 在 Intel 的 CLX(28) 和 ICX(38) 平台上, PSNVJ 算法的性能几乎均低于 Vector Join --basic-numa, 数据分区代价抵消了 NUMA 本地向量连接性能的收益. 在 ARM(64), Rome Zen2 和 Milan Zen3 平台上, PSNVJ 算法和 Vector Join --basic-numa 算法有相似的性能特征, PSNVJ 算法在 R 表较小和较大时性能均低于 Vector Join --basic-numa 算法, 在 R 表范围较大的区间内性能低于 Vector Join --basic-numa, 如在 ARM(64) 平台上 PSNVJ 算法的性能优势区间为 $2^{21}<|R|<2^{28}$, 在 Rome Zen2 的性能优势区间为 $2^{22}<|R|<2^{29}$, 在 Milan Zen3 的性能优势区间为 $2^{23}<|R|<2^{29}$. 当 R 表向量大小超过 L3 cache 时 (Rome Zen2 片上 L3 cache 大小为 16 MB, Milan Zen3 片上 L3 cache 大小为 32 MB, 向量宽度为 4 B), Vector Join 算法失去 cache 局部性, 会产生跨 NUMA 内存访问延迟, 向量探测时间高于 PSNVJ 算法. 当 R 表足够大时, PSNVJ 算法需要在 NUMA 本地访问完整的连接向量, 而 Vector Join 算法在各 NUMA 节点中访问 1/|NUMA 节点|×|连接向量|大小的向量, cache 访问效率逐渐提升, 连接性能逐渐优于 PSNVJ 算法.

NUMA SN 集群连接算法的动态数据分区代价较高. 如在 ICX(38) 平台上最小连接负载$|R|=2^5$ 和最大连接负载$|R|=2^{29}$ 时, 分区时间在连接总时间中占比为 82.5% 和 26.2%. PSNVJ 算法执行时间为 Vector Join --basic-numa 执行时间的 5.65 倍和 1.21 倍. 当采用静态 NUMA 数据预分区策略时, PSNVJ 算法消除动态分区代价后, 其执行时间为 Vector Join --basic-numa 执行时间的 0.99 倍和 0.89 倍, 静态分区策略的性能收益不显著. 存储引擎中的静态分区策略适合两个大表连接场景, 如 TPC-DS 基准中 store_sales 和 store_returns 双事实表按连接键进行 NUMA 分区存储. 当数据库中存在多个具有连接关系的大表时, 如 TPC-H 中 lineitem 表, orders 表和 partsupp 表, 多维 NUMA 连接分区的存储和计算代价较高, 具有一定的局限性.

综上所述, 当跨 NUMA 访问延迟较高时, 可以将 NUMA 节点看作轻量 SN 集群, 通过典型的全复制或分区复制策略提高 NUMA 节点内访问的数据局部性. 但连接表在 NUMA 节点上的协同分区需要一定的存储和数据预处理代价, 当 CPU 的 NUMA 延迟性能提升 (如 ICX(38) 跨 NUMA 延迟较低) 及连接算法性能提升 (如向量连接 cache 效率较高) 时, NUMA SN 集群并行连接算法的性能收益降低, 硬件及软件性能的提升降低了复杂优化技术的必要性.

4.5 NUMA 合并策略连接性能

NUMA 合并策略连接算法是面向 ARM 片上多 NUMA 节点所定制的优化策略. 由于片上 NUMA 间访问延

迟远低于片间 NUMA 访问延迟, 因此合并片上的 NUMA 节点可以减少 NUMA 分区及复制的数量, 提高内存存储效率和连接性能.

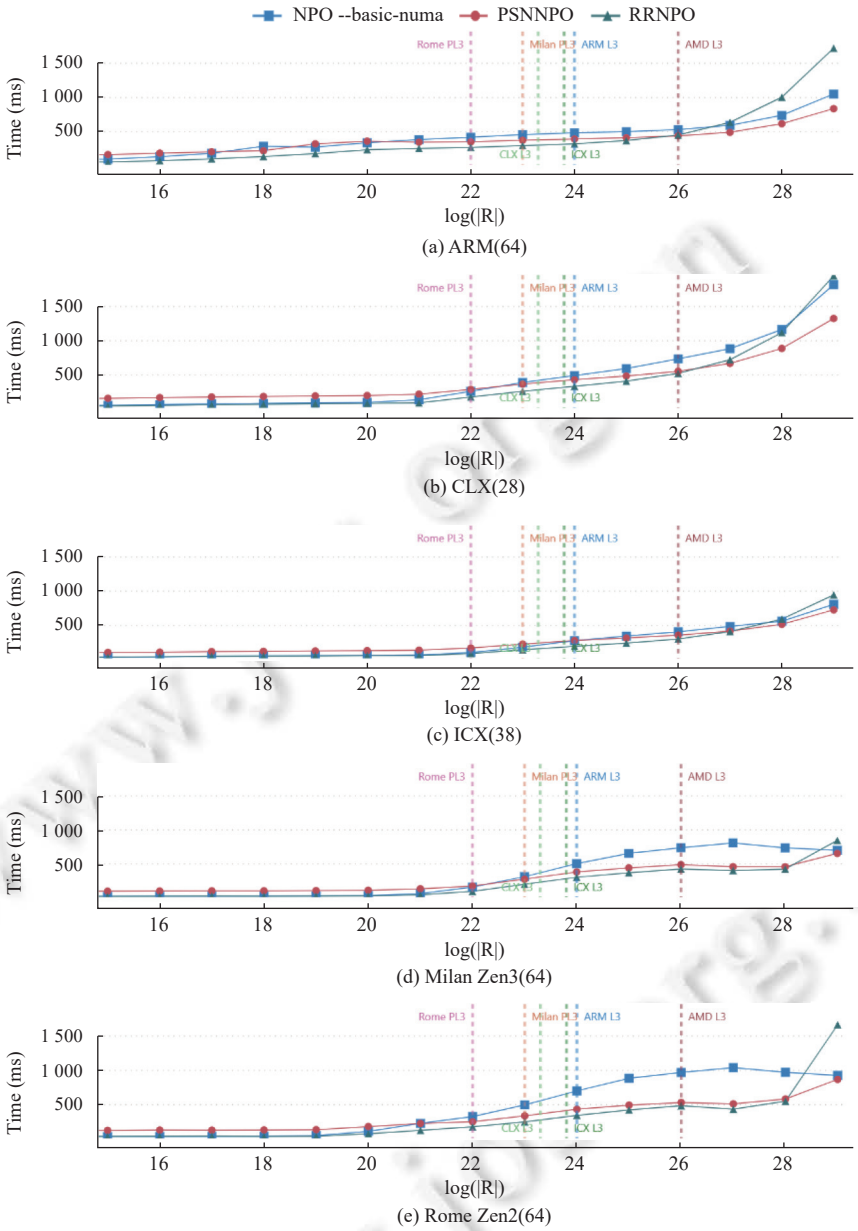


图 10 NUMA SN 集群 NPO 连接算法性能比较

图 12 对比了 3 种 NPO 算法: 一是在不合并 NUMA 分区细粒度哈希表复制连接算法 (FRHNPO), 二是合并 NUMA 分区细粒度哈希表复制连接算法 (FRHNPONM), 三是 baseline 算法的性能 (NPO --basic-numa).

在 ARM(64) 平台上, FRHNPO 算法和 FRHNPONM 算法的性能均低于 NPO --basic-numa 基准算法性能, 主要原因在于较大的 cache 容量减少了内存访问数量, 较高的跨 NUMA 访问延迟增加了跨 NUMA 哈希表访问代价. 但在 R 表较大的连接负载中 FRHNPONM 算法性能优于 FRHNPO, 主要原因是合并的 NUMA 节点减少了 NUMA 创建哈希表的数量, 降低了跨 NUMA 创建哈希表的代价.

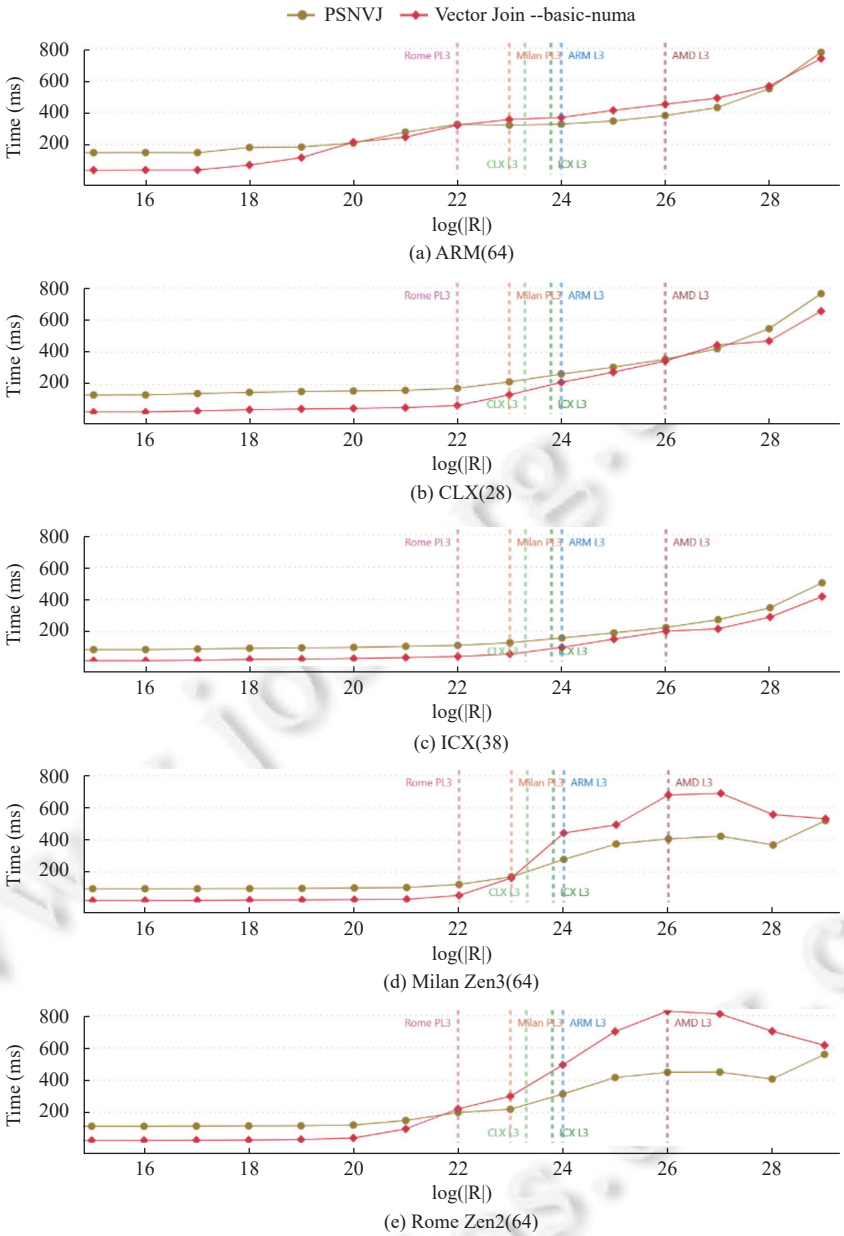


图 11 NUMA SN 集群向量连接算法性能比较

在向量连接算法中, 实线代表 NUMA 合并策略连接算法, 虚线代表按原始 NUMA 节点分区的连接算法. 图 13 中观察到以下现象: 基于粗粒度复制策略的 CRVJNM 算法相对于 CRVJ 算法性能有所提升, 性能收益的主要原因是 NUMA 分区合并策略减少了 NUMA 节点间的复制代价; 当 R 表较小时, FRVVJNM 算法基于细粒度复制策略, 其性能低于 FRVVJ 连接算法. 当 R 表行数大于 2^{28} 行时, FRVVJNM 算法性能优于 FRVVJ 算法, 主要原因在于 R 表较小时, FRVVJ 算法具有较低的内存访问延迟以及 L3 cache 缓存效率, 从而提高了算法整体性能; 当 R 表较大时, 相同芯片内两个 NUMA 节点需要争用相同的 L3 cache, 从而增加了连接操作的内存访问延迟, 降低了连接算法的整体性能. 基于排序的 SVCRVJ 算法在 NUMA 分区合并与不合并两种策略下性能相近, 排序后的向量在每个 NUMA 节点中虽然有多个向量片断, 但探测阶段根据外键值映射到唯一的向量片断中, NUMA 分区数量对探

测操作延迟的影响极小; 从连接算法整体性能来看, 在大多数连接负载中, 面向 NUMA 节点采用基于数据复制策略的连接算法, 其性能低于 Vector Join --basic-numa 基准算法性能,

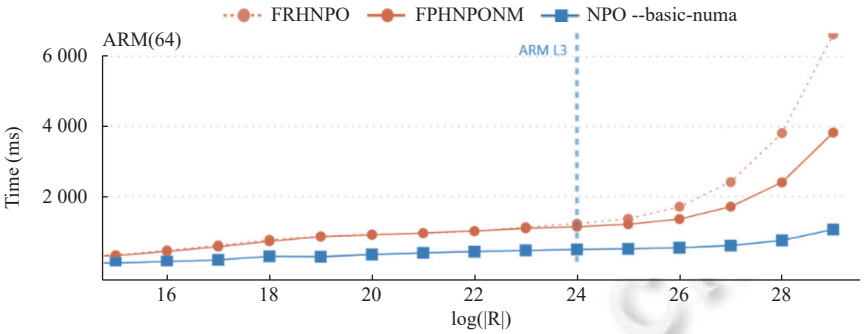


图 12 ARM(64) 平台上 NUMA 合并策略 NPO 连接算法性能比较

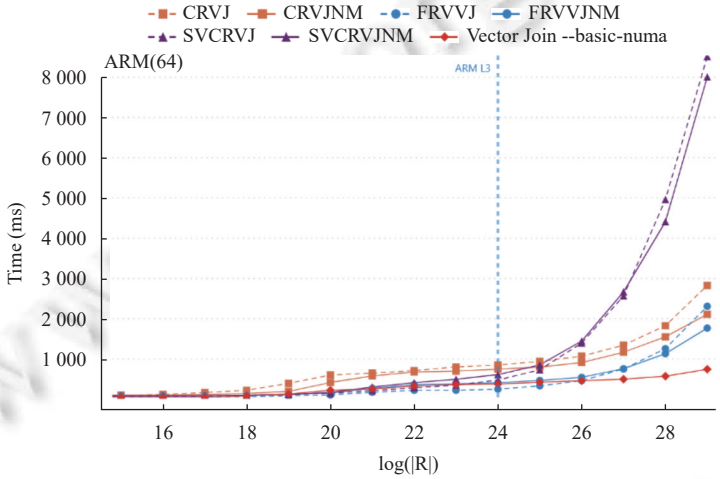


图 13 ARM(64) 平台上 NUMA 合并策略向量连接算法性能比较

综上所述, 如何平衡 NUMA-conscious 连接算法的数据复制代价和数据访问延迟是性能优化的重要影响因素. 新型 CPU 采用片上多 NUMA 架构 (4th Gen AMD EPYC 9004 series processor (https://www.amd.com/system/files/documents/4th-gen-epyc-processor-architecture-white-paper.pdf)) 和分离式 L3 cache 架构, 在内存、cache 微架构上局部性数据访问延迟和粒度差异增大, 需要进一步深入研究其优化策略.

4.6 基于 NUMA-conscious 外键连接优化技术实现的 SSB 基准负载性能

SSB 基准负载是麻省州立大学波士顿校区的研究人员定义的基于现实商业应用的数据模型, 主要模拟决策支持类应用, 被工业界和学术界广泛应用于数据仓库和 OLAP (在线分析处理) 系统性能评估. 为体现 NUMA 优化技术在基于现代多核处理器的软件优化中的重要性和必要性, 本节实验选取了分别配备 Intel 最新 Intel® Xeon® 6 产品系列中的 Intel Xeon 6780E 和 AMD 最新 EPYC 9004 Series 产品系列中的 AMD EPYC 9654 的双路服务器作为测试平台, 详细的 CPU 介绍和硬件参数如表 4 所示, 并基于向量连接算法和本文中的 NUMA-conscious 外键连接优化技术实现的 NUMA-conscious-SSB 和 NUMA-oblivious-SSB 基准测试工具来模拟和探究 NUMA-conscious 外键连接优化技术在现实世界数据库系统负载的性能收益和重要性.

数据规模为 SF=100 的 SSB 基准测试结果如图 14 所示, 其测试时间由两部分组成: (1) 面向元数据管理的维向量生成计算和 (2) 面向事实表外键列的多表连接分组聚集计算, 首先可看到无论是在 Intel 平台还是在 AMD 平

台的 NUMA-oblivious-SSB 的平均查询时间组成中, 多表连接的时间占比都超过 84%, 所以在数据库负载的场景下探究 NUMA-conscious 外键连接优化技术是必要且重要的. 其次, Intel 和 AMD 平台上的 NUMA-conscious-SSB 在每条查询上性能都优于 NUMA-oblivious-SSB, 由于本节实验所用到的 CPU cache 容量较大, 其中 AMD EPYC 9654 的 L3 cache 容量高达 383 MB, 而 SF=100 的 SSB 基准测试中生成最大向量大小仍处于 cache 容量内, 受 cache 访问数据延迟低的影响, 缩小 NUMA-conscious 外键连接优化技术带来的性能优化空间, 但在 Intel 平台上的平均性能优化可达 24.6%, 在 AMD 平台上可达 14.4%, 且由图 5 可推知, 随着数据量进一步增大, NUMA-conscious 外键连接优化技术所带来的性能提升将随之扩大, 所以在面临海量数据分析处理需求挑战的当下, 数据库系统如何面向现代处理器的 NUMA 架构特性做连接算法优化是非常重要且亟待解决的.

表 4 执行 SSB 基准测试的 CPU 架构及硬件参数

CPU型号	类别	主频 (GHz)	核心	缓存	内存	NUMA节点数
Intel Xeon 6780E	Intel® Xeon® 6	2.2	144	64 KB L1, 2 MB L2, 108 MB L3	8通道, 带宽490.8 GB/s	1
AMD EPYC 9654	EPYC 9004 Series	2.4	96	64 KB L1, 2 MB L2, 384 MB L3	12通道, 带宽434.7 GB/s	2

图 14 NUMA-conscious-SSB 和 NUMA-oblivious-SSB 性能对比

5 相关工作

连接是关系数据库中最重要的操作之一, 在数据仓库和 OLAP 中对性能的影响尤为重要, 庞大的事实表与多个维表之间的连接代价是 OLAP 查询中最大的代价. 当前内存数据库基于多核处理器和大内存平台, 连接算法性能研究是一个重要的研究热问题. 什么样的连接算法具备最高的性能近年来持续更新中, 新型硬件架构的差异性增加了问题研究的维度和难度, 硬件架构的特征对连接算法性能研究具有重要的影响作用.

面对硬件技术的飞速发展和硬件差异性的增大, 连接算法性能是 hardware-conscious 还是 hardware-oblivious, 哪个更优的讨论起源于文献 [6], 最初的结论是简单的 hardware-oblivious 连接算法在性能上比更为复杂和依赖硬件的 hardware-conscious 连接算法有竞争力. 因此鼓励数据库系统开发时采用更简单的无分区哈希连接算法和一些自动优化技术, 如自动预取等来提升内存连接算法性能. 无分区哈希连接算法需要创建全局共享访问哈希表, 在创建阶段受多线程并发访问控制代价影响, 在探测阶段受 L3 cache 大小和缓存访问效率影响, 在硬件架构的亲和性上更倾向于较大共享 L3 cache 架构, 其性能主要受共享哈希表相对于 L3 cache 大小的影响, 内存利用率相对较高. 连接性能的结论被 Balkesen 等人[1]反转, 提出了面向硬件特性的优化与调优策略使 hardware-conscious 的 Radix 分区连接算法性能优于 hardware-oblivious 的无分区哈希连接算法. 其优化技术的基本假设基于连接性能最主要的影响因素是数据访问的 cache 局部性, cache 容量不足产生会内存访问延迟, 因此需要通过 Radix 分区技术将大数据集划分为适合 cache 大小的数据集, 来实现 in-cache 连接操作, 从而可以降低连接操作的数据访问延迟. Radix 分区连接算法是以 CPU 核心的私有 cache 容量为粒度, 进行“分而治之”的优化策略, 可以实现在哈希表创建阶段无锁化以及在哈希探测阶段降低内存访问延迟. 但分区操作加倍了内存空间消耗, 产生大量内存读写操作代价. Lang 等人[8]进一步探索了 NUMA 架构对连接性能的影响, 提出了 NUMA-aware 无分区哈希连接算法, 其性能在 4 个 NUMA 节点 64 线程的硬件平台上显著优化 Radix 分区连接算法. Albutiu 等人[10]提出了 MPSM (massively parallel sort-merge join) 连接算法, 基于 NUMA 友好的排序归并算法提高了跨 NUMA 访问效率, 提升了性能. Balkesen 等人[2]进一步优化了无分区哈希连接算法、Radix 分区连接算法和排序归并算法, 实验结果显示哈希连接算法相对于排序归并连接算法仍然有较好的性能优势. 随着连接优化研究工作的深入, 丰富的连接优化技术给出了各不相同的连接性能结论, Schuh 等人[4]对 13 个基于 NUMA 架构进行优化的连接算法进行了深入的性能对比, 通过统一的连接基准揭示全面的连接算法性能特征, 研究结果表明在现代多 NUMA 架构计算平台上, hardware-conscious 的 Radix 分区连接算法普遍优于 hardware-oblivious 的无分区哈希连接算法性能, 该研究还指出在 TPC-H 查询负载中只有较小部分代价发生在连接操作上.

哈希连接性能研究从 CPU 平台扩展到硬件加速器平台, 如 GPU 和 FPGA 平台. He 等人[11]研究了集成 CPU-GPU 处理器架构上的哈希连接优化技术和性能, Halstead 等人[12]设计了 FPGA 平台上无分区哈希连接实现技术, Kara 等人[13]开发了 FPGA 平台 Radix 分区哈希连接算法. Rui 等人[14]研究了 GPU 上基于排序归并的连接算法, Sioulas 等人[15]的研究揭示了相似的结论, 即 GPU 平台上 Radix 分区哈希连接算法面向硬件特性进行优化, 其性能较无分区哈希连接算法同样有较大的性能优势. 从多核 CPU 平台到 GPU 和 FPGA 平台, 通用的结论是 hardware-conscious 的 Radix 分区连接算法性能优于 hardware-oblivious 的无分区哈希连接算法. 但该结论忽略了无分区哈希连接算法性能依赖于 CPU L3 cache 容量, 随硬件技术迅速提升所产生的性能优势越来越明显. 而优势的 Radix 分区连接算法由于内存空间利用率低以及物化连接策略难以适应现代流水线处理模型, 因此带来综合性能损失.

哈希连接算法研究工作通常使用定制的连接负载, 相对于现实世界的 SSB、TPC-H、TPC-DS 等数据库连接负载相比, 在应用中不仅要考虑性能维度, 还需要综合考虑更多的优化因素. Shanbhab 等人[16]在 GPU 平台上, SSB 基准负载实现技术中不再使用 Radix 分区连接算法, 主要原因是在 GPU 平台的连接物化导致显存利用率降低并且无法支持优化的流水处理模型. 传统的哈希连接和排序归并连接是无索引的等值连接算法, 连接算法的核心功能是根据外键值在连接表上的等值匹配操作, 没有数据库模式特征的支持, 没有使用面向连接的索引技术, 如连接索引[17]来加速外键值查找性能. 在 OLAP 负载中, 当主键使用代理键时, 外键成为连接索引, 即将 R 表的哈希

表简化为向量结构, S 表外键值可以直接映射为 R 表向量的偏移地址, 实现基于值-地址映射的直接访问. 从而可以简化哈希表结构, 优化哈希表大小并消除传统哈希探测过程中较高哈希值计算、键值匹配、溢出桶遍历等操作的 CPU 计算代价[9]. 这种基于简单高效向量结构的算法可以应用于数据库的分组聚集和多表连接操作, 来提升数据库基础算子的性能[18]. 需要特别指出的是, 相对于文献 [4] 指出连接代价占查询总代价比例偏低的结论, SSB 查询基于连接索引, 其连接操作执行代价占比达到 85% 以上, 从而使连接操作的优化在查询处理中获得最大收益. 最新研究[7]指出, 在学术界广泛关注和研究的 Radix 分区哈希连接算法在真实系统 TPC-H 负载上只有极为有限的性能收益. 从理论研究与应用实践相结合的视角来看, 无分区哈希连接算法及向量连接算法的优化工作需要进一步强化. 我们在文献 [19] 中深入研究了在不同 CPU 平台上, 数据库细粒度算子性能基准与宏基准的性能特征, 通过 SSB 数据集模拟了无分区哈希连接算法、Radix 分区哈希连接算法和向量连接算法的多表连接性能, 实验结果显示向量连接和无分区哈希连接在性能上远优于 Radix 分区哈希连接算法. 因此本文研究工作的重点放在前两个连接算法, Radix 分区连接算法仅用作基准性能对比.

在相关连接优化研究工作中, NUMA 优化通常作为一种辅助优化手段. CPU 不同硬件技术路线发展趋势加大了 NUMA 硬件的差异性, 如降低整体跨 NUMA 访问延迟和增加每 CPU 内部 NUMA 节点数量、减少内部 NUMA 节点访问延迟等技术路线. 从而使 NUMA 优化从阶段性走向全局性, 从连接算法维度的优化走向结合存储引擎优化的维度, 在算法设计上从集中式模式走向微观 NUMA SN 集群模式, 丰富了 NUMA 研究的应用场景. 当前学术界对 NUMA 优化的连接算法研究缺乏整体技术框架, 本文基于不同的 CPU 硬件架构提出了一个系统性的 NUMA-conscious 连接优化技术框架, 为新型硬件平台 NUMA 优化技术提供一个全面深入的算法库、基准测试方法和研究框架. 与本文探索静态的面向 NUMA 节点特性的数据放置策略, 以较小的系统调优代价来最大化利用硬件性能和数据库负载在 NUMA 架构下的性能收益的立足点不同的是, Dominico 等人[20]提出了一种基于抽象模型的弹性多核动态分配机制为操作系统提供特定 NUMA 节点中的局部最佳核心数量来减轻跨 NUMA 节点访问数据代价, 但这种机制理论上并不能发挥出硬件资源 (如所有核心) 的最佳性能. Fogli 等人[21]基于分布式数据库负载研究进一步发现在 chiplet-based 架构的 CPU 上, chiplet-aware 的优化技术相对于 NUMA-aware 的优化技术有更大的性能收益, 所以我们在下一步的工作中将参考其结论研究集中式内存数据库在 chiplet-based 架构上的优化技术.

6 结 论

本文聚焦于代表性的 5 款、两代 (二代和三代)、两种架构 (ARM 和 x86)、3 种类型 (ARM, Intel, AMD) CPU 上, 面向 NUMA 架构的连接优化技术研究. 发现连接性能没有单一的答案, 而是一个多维度的问题. 从算法维度而言, hardware-oblivious 的向量连接算法在 ARM, CLX 和 ICX 平台上优于 hardware-conscious 的 Radix 分区连接算法, 主要原因是其较大的 L3 cache 容量和效率. 从 NUMA-conscious 算法设计的维度来说, 不同连接算法对 NUMA-conscious 优化技术的敏感度不同, 无分区哈希连接算法通常对 NUMA-conscious 优化技术有较强及多样化的敏感性, 适合 NUMA-conscious 优化技术的应用场景较多且在方法的选择上具有多样性; Radix 分区哈希连接算法对 NUMA-conscious 优化技术的敏感性中等且较为稳定, 而向量连接算法对 NUMA-conscious 优化技术的敏感性较低, 从而使其具备 hardware-oblivious 和 NUMA-oblivious 的特性, 可以在保证较高性能的基础上简化数据库连接算法的设计. 从硬件维度来看, 不同连接优化技术在不同架构的 CPU 平台呈现不同的性能特征, NUMA-conscious 优化技术的性能收益在 Intel 架构的 CLX 和 ICX 平台上低于 AMD 架构的 Rome Zen2 和 Milan Zen3 以及 ARM 平台, 其较低的跨 NUMA 访问延迟降低了 NUMA 优化的重要性, 而较高的跨 NUMA 访问延迟特性则强化了 NUMA 优化的权重. 从 CPU 硬件技术发展趋势来看, 单 CPU 单 NUMA、大容量集中式 cache、低内存访问延迟、高带宽、更多并行核心等硬件特性可以通过 NUMA-oblivious 算法获得较高的性能, 而单 CPU 多 NUMA、分散式小容量 L3 cache、高内存访问延迟等硬件特性则需要与 NUMA-conscious 优化技术深度结合以发挥更高的性能. 从数据库模式优化的维度来说, 通过模式优化技术提高频繁访问数据集的大小以提高数据访问局部性、通过部分物化或冗余连接降低大表连接代价能够更好地利用当前 CPU cache 容量的发展趋势, 简化数据库连接算

法设计的复杂度, 提高连接算法性能.

从提高 NUMA 架构下数据库连接操作性能的目标来看, 在 OLAP 查询算法上优先选择具有良好 cache 局部性的向量连接算法, 最大化 cache 缓存效率并减少 NUMA 延迟的影响; 在 OLAP 数据库模式设计上, 通过模式优化降低连接表大小, 使其连接向量尽量位于 cache 优化区间 (最新 Intel 144 核处理器 L3 cache 容量达到 108 MB, 可以支持最大 1 亿行表记录的 cache 内向量连接), 通过 NUMA-oblivious 连接算法降低 NUMA 延迟对性能和算法复杂度的影响; 在 CPU 的硬件架构设计上, 数据库连接负载的强局部性特征需要大容量统一 L3 cache、降低 NUMA 访问延迟等硬件特性以应用简单的自适应 NUMA-oblivious 连接算法, 多 NUMA 的 chiplet 架构会增加 NUMA 连接算法优化的复杂性, 同时需要数据库在内存存储模型上面向多 NUMA 架构设计与之匹配的 NUMA 分区存储策略, 将集中式内存数据库扩展为轻量 SN 架构内存数据库.

随着新一代 CPU 技术的发展, CPU 硬件架构的差异性进一步扩大, 原有连接优化技术的结论在新型架构上可能会产生偏差, 本文通过设计系统性的算法库、测试基准和性能对比方法为 NUMA 连接优化提供一个可参考的研究框架.

致谢 感谢英特尔 (中国) 实验室邓刚、唐曦、钟涛等英特尔高级工程师对本文的硬件设备和技术支持.

References:

[1] Balkesen C, Teubner J, Alonso G, Özsu MT. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In: Proc. of the 29th IEEE Int'l Conf. on Data Engineering (ICDE). Brisbane: IEEE, 2013. 362–373. [doi: 10.1109/ICDE.2013.6544839]

[2] Balkesen C, Alonso G, Teubner J, Özsu MT. Multi-core, main-memory joins: Sort vs. hash revisited. Proc. of the VLDB Endowment, 2013, 7(1): 85–96. [doi: 10.14778/2732219.2732227]

[3] Zhang YS, Zhang Y, Wang S, Lu JH. Fusion OLAP: Fusing the pros of MOLAP and ROLAP together for in-memory OLAP. IEEE Trans. on Knowledge and Data Engineering, 2019, 31(9): 1722–1735. [doi: 10.1109/TKDE.2018.2867522]

[4] Schuh S, Chen X, Dittrich J. An experimental comparison of thirteen relational equi-joins in main memory. In: Proc. of the 2016 Int'l Conf. on Management of Data. San Francisco: ACM, 2020. 1961–1976. [doi: 10.1145/2882903.2882917]

[5] Zhang YS, Zhou X, Zhang Y, Zhang Y, Su MC, Wang S. Virtual denormalization via array index reference for main memory OLAP. IEEE Trans. on Knowledge and Data Engineering, 2016, 28(4): 1061–1074. [doi: 10.1109/TKDE.2015.2499199]

[6] Blanas S, Li YN, Patel JM. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In: Proc. of the 2011 ACM SIGMOD Int'l Conf. on Management of Data. Athens: ACM, 2011. 37–48. [doi: 10.1145/1989323.1989328]

[7] Bandle M, Giceva J, Neumann T. To partition, or not to partition, that is the join question in a real system. In: Proc. of the 2021 Int'l Conf. on Management of Data. ACM, 2021. 168–180. [doi: 10.1145/3448016.3452831]

[8] Lang H, Leis V, Albutiu MC, Neumann T, Kemper A. Massively parallel NUMA-aware hash joins. In: Proc. of the 1st and 2nd Int'l Workshops on in Memory Data Management and Analysis. Riva del Garda: Springer, 2015. 3–14. [doi: 10.1007/978-3-319-13960-9_1]

[9] Zhang YS, Zhang Y, Zhou X, Lu JH. Main-memory foreign key joins on advanced processors: Design and re-evaluations for OLAP workloads. Distributed and Parallel Databases, 2019, 37(4): 469–506. [doi: 10.1007/s10619-018-7226-4]

[10] Albutiu MC, Kemper A, Neumann T. Massively parallel sort-merge joins in main memory multi-core database systems. Proc. of the VLDB Endowment, 2012, 5(10): 1064–1075. [doi: 10.14778/2336664.2336678]

[11] He J, Lu M, He BS. Revisiting co-processing for hash joins on the coupled CPU-GPU architecture. Proc. of the VLDB Endowment, 2013, 6(10): 889–900. [doi: 10.14778/2536206.2536216]

[12] Halstead RJ, Absalyamov I, Najjar WA, Tsotras VJ. FPGA-based multithreading for in-memory hash joins. 2014. https://kipdf.com/fpga-based-multithreading-for-in-memory-hash-joins_5ab7c0ac1723dd329c647c31.html

[13] Kara K, Giceva J, Alonso G. FPGA-based data partitioning. In: Proc. of the 2017 ACM Int'l Conf. on Management of Data. Chicago: ACM, 2017. 433–445. [doi: 10.1145/3035918.3035946]

[14] Rui R, Tu YC. Fast equi-join algorithms on GPUs: Design and implementation. In: Proc. of the 29th Int'l Conf. on Scientific and Statistical Database Management. Chicago: ACM, 2017. 17. [doi: 10.1145/3085504.3085521]

[15] Sioulas P, Chrysogelos P, Karpathiotakis M, Appuswamy R, Ailamaki A. Hardware-conscious hash-joins on GPUs. In: Proc. of the 35th IEEE Int'l Conf. on Data Engineering (ICDE). Macao: IEEE, 2019. 698–709. [doi: 10.1109/ICDE.2019.00068]

[16] Shanbhag A, Madden S, Yu XY. A study of the fundamental performance characteristics of GPUs and CPUs for database analytics. In:

Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. Portland: ACM, 2020. 1617–1632. [doi: 10.1145/3318464.3380595]

[17] Zhang YS, Wang S, Lu JH. Improving performance by creating a native join-index for OLAP. Frontiers of Computer Science in China, 2011, 5(2): 236–249. [doi: 10.1007/s11704-011-9181-3]

[18] Chavan S, Hopeman A, Lee S, Lui D, Mylavarapu A, Soylemez E. Accelerating joins and aggregations on the oracle in-memory database. In: Proc. of the 34th IEEE Int'l Conf. on Data Engineering (ICDE). Paris: IEEE, 2018. 1441–1452. [doi: 10.1109/ICDE.2018.00163]

[19] Liu Z, Han RC, Zhang YS, Zhang Y, Tang X, Deng G, Zhong T, Dementiev R, Lu YF, Que MJ. Exploring fine-grained in-memory database performance for modern CPUs. IEEE Trans. on Parallel and Distributed Systems, 2023, 34(6): 1757–1772. [doi: 10.1109/TPDS.2023.3262782]

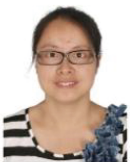
[20] Dominico S, de Almeida EC, Meira JA, Alves MAZ. An elastic multi-core allocation mechanism for database systems. In: Proc. of the 34th IEEE Int'l Conf. on Data Engineering. Paris: IEEE, 2018. 473–484. [doi: 10.1109/ICDE.2018.00050]

[21] Fogli A, Zhao B, Pietzuch P, Bandle M, Giceva J. OLAP on modern chiplet-based processors. Proc. of the VLDB Endowment, 2024, 17(11): 3428–3441. [doi: 10.14778/3681954.3682011]

韩瑞琛(1997－), 男, 博士生, 主要研究领域为内存数据库, 新硬件数据库.

张宇(1977－), 女, 博士, 高级工程师, 主要研究领域为数据仓库, OLAP.

张延松(1973－), 男, 博士, 副教授, 主要研究领域为内存数据库, GPU 数据库, 新硬件数据库技术.



焦敏(1975－), 女, 博士, 高级实验师, CCF 专业会员, 主要研究领域为数据仓库, GPU 数据库.

刘专(1996－), 男, 硕士, 主要研究领域为 GPU 数据库, 内存数据库.

王珊(1944－), 女, 教授, CCF 会士, 主要研究领域数据库, 数据仓库, 大数据管理.