软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(3):774−796 [doi: 10.13328/j.cnki.jos.006441]

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

Geno: 基于代价的异构融合查询优化器*

屠要峰[1,2], 陈小强[2], 周士俊[2], 卞福升[2], 吴 非[2], 陈 兵[1]

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

[2](中兴通讯股份有限公司, 江苏 南京 210012)

通信作者: 屠要峰, E-mail: 13605151819@qq.com

摘 要: 新型硬件及其构建的环境改变了传统的计算、存储以及网络体系, 也改变了上层软件既往的设计假设,特别是通用处理器和专用加速器组成的异构计算架构, 改变了数据库系统的底层框架设计和查询优化的代价模型. 数据库系统需要针对新型硬件的特性做出适应性调整, 以充分发挥新硬件的潜力. 提出一种面向 CPU/GPU/FPGA 异构计算融合的基于代价的查询优化器 Geno, 可以灵活地调度并最优化地使用各类资源. 主要的贡献是: 发现根据系统环境硬件实际能力调整代价参数可以显著地提升查询计划的准确性, 并提出一种异构资源代价计算方法和校准工具; 通过对 GPU、FPGA 等异构硬件能力估算及对数据库系统硬件实际能力的校准, 建立异构计算环境下查询处理的代价模型; 实现了支持选择、投影、连接、聚合的 GPU 算子和 FPGA 算子, 实现了 GPU 算子融合及流水线设计、FPGA 算子流水线设计; 通过基于代价的评估解决算子分配和调度问题, 生成异构协同的执行计划, 实现异构计算资源的协同优化, 以充分发挥各异构资源的优势. 实验结果表明, 通过 Geno 校准后的参数值与实际硬件能力更加匹配. 相比于 PostgreSQL 和 GPU 数据库 HeteroDB, Geno 能够生成更加合理的查询计划. TPC-H 实验中,在行存表情况下, Geno 比 Postgresql 执行时长减少了 64%–93%, 比 Hetero-DB 执行时长减少了 1%–39%; 在列存表情况下, Geno 比 Postgresql 执行时间减少了 87%–92%, 比 Hetero-DB 执行时间减少了 1%–81%; Geno 列存与行存相比, 查询执行时间减少了 32%–89%.

关键词: 数据库; 查询优化; GPU; FPGA; 异构计算

中图法分类号: TP311

中文引用格式: 屠要峰, 陈小强, 周士俊, 卞福升, 吴非, 陈兵. Geno: 基于代价的异构融合查询优化器. 软件学报, 2022, 33(3): 774–796. http://www.jos.org.cn/1000-9825/6441.htm

英文引用格式: Tu YF, Chen XQ, Zhou SJ, Bian FS, Wu F, Chen B. Geno: Cost-based Heterogeneous Fusion Query Optimizer. Ruan Jian Xue Bao/Journal of Software, 2022, 33(3): 774−796 (in Chinese). http://www.jos.org.cn/1000-9825/6441.htm

Geno: Cost-based Heterogeneous Fusion Query Optimizer

TU Yao-Feng[1,2], CHEN Xiao-Qiang[2], ZHOU Shi-Jun[2], BIAN Fu-Sheng[2], WU Fei[2], CHEN Bing[1]

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

[2](Zhongxing Telecommunication Equipment Corporation, Nanjing 210012, China)

Abstract: The new hardware and its built environment have changed the traditional computing, storage and network systems, and also changed the previous design assumptions of the upper-level software. In particular, the heterogeneous computing architecture composed of general-purpose processors and dedicated accelerators has changed the design of the underlying framework of the database system and the cost model of query optimization. The database system needs to make adaptive adjustments to the characteristics of the new hardware to give full play to the potential of the new hardware. A cost-based query optimizer Geno for CPU/GPU/FPGA heterogeneous computing fusion is proposed, which can flexibly schedule and optimize the use of various computing resources. The main contribution is: finding

* 基金项目: 国家重点研发计划(2019YFB2102002); 江苏省重点研发计划(BE2019012)

本文由“数据库系统新型技术”专题特约编辑李国良教授、于戈教授、杨俊教授和范举教授推荐.

收稿时间: 2021-06-29; 修改时间: 2021-07-31; 采用时间: 2021-09-13; jos 在线出版时间: 2021-10-21

that adjusting the cost parameters according to the actual hardware capabilities of the system environment can significantly improve the accuracy of the query plan, and proposing a calculation method and calibration tool for the cost of heterogeneous resources; through the estimation of the capabilities of heterogeneous hardware such as GPU and FPGA and the calibration of the actual capabilities of the database system hardware, establishing a cost model for query processing in a heterogeneous computing environment; implementing GPU operators and FPGA operators that support selection, projection, join and aggregation, realizing GPU operator pipeline design and FPGA operator pipeline design; solving the operator assignment and scheduling through cost-based evaluation, and generating a heterogeneous collaborative execution plan to realize the collaborative optimization of heterogeneous computing resources to makes full use of the advantages of each heterogeneous resource. Experiments show that the parameter values calibrated by Geno are more compatible with the actual hardware capabilities. Compared with PostgreSQL and GPU database HeteroDB, Geno can generate a more reasonable query plan. In the TPC-H scenario, the execution time of Geno in the case of row storage is 64%–93% less than that of Postgresql, and 1% to 39% less than that of Hetero-DB; in the case of column storage, Geno's execution time is 87%–92% less than that of Postgresql, and 1%–81% less than that of Hetero-DB; Compared with row storage, Geno reduces query execution time 32%–89% in the case of column storage.
Key words: database; query optimization; GPU; FPGA; heterogeneous computing

21 世纪以来, 随着互联网、物联网和移动计算技术的发展, 人们能够获取的数据规模爆炸式增长, 大量新型应用应运而生, 对数据的实时分析需求越来越高. 摩尔定律增长变得缓慢, 传统以 CPU 为中心的计算技术面临"能耗墙"、"内存墙"的限制, 仅仅通过增加 CPU 主频来获得性能提升的方式走到了尽头. 高性能处理器技术进入多核时代, 多核 CPU 在一定程度上缓解了计算性能提升的瓶颈. 同时, 通用图形处理器(general purpose computing on graphics processing units, GPGPU)、现场可编程门阵列(field programmable gate array, FPGA)等专用处理器因为其具备大规模并行计算的能力, 在处理海量数据方面具有得天独厚的优势. 这使得"通用处理器+异构加速器"在很大程度上成为现代计算的发展趋势, 这些发展所产生的影响是服务器硬件朝着高度异构的方向转变. 因此, 高性能处理器和新型加速器的出现, 使得传统的数据管理与分析系统从单 CPU 架构开始向异构、混合处理架构转变[1,2].

数据库是高效组织、存储、管理数据的软件, 数据库技术是计算机信息系统和应用系统的重要支柱. 随着多样化应用的扩展与深入, 数据库管理的数据规模越来越大, 查询处理更加复杂庞大, 而且实时性要求越来越高. 对于数据库系统而言, 硬件技术是载体支撑, 决定了数据存取和查询等处理性能的物理极限; 软件的目标是优化算法与数据结构的设计以提高软件与硬件的契合度, 最大化硬件使用效率, 同时规避或减少硬件固有的限制[3,4]. 因此, 数据库系统要应对海量异构数据带来的巨大挑战, 必须充分利用多核 CPU 以及 GPU/FPGA 等异构硬件的高速并行处理能力, 将一部分 CPU 任务卸载到异构硬件上去执行, 比如计算密集型数据库操作, 如数据的 SCAN、谓词过滤、大量数据排序、大表 JOIN、聚集等, 以及 I/O 密集型操作如压缩/解压缩、加密/解密等, 以充分发挥各类计算资源的优势.

然而, 处理器架构的差异直接影响到数据库的工作效率, 数据库系统需要根据不断发展的处理器架构做出相应的优化. 根据 Stonebraker 等人[5]在 2014 年 VLDB 上发表的论文证明, 传统数据库的事务处理机制无法有效利用数十到上百个核处理能力. 新型异构计算架构也改变了数据库系统查询优化的代价模型, 查询优化设计与硬件特征的关联性越来越高, 所依赖的因素也更加复杂, 查询优化的难度进一步增加. 现有数据库查询优化在面对新型异构硬件时存在的挑战主要有:

① 如何利用异构硬件高效地实现关系代数运算和丰富的关系算子.

② 查询优化所依赖的路径代价计算直接影响到所选择查询计划的优劣: 首先是现有的代价模型缺乏异构硬件能力的评估方法和异构算子路径代价的计算方法, 部分现有基准代价与实际环境中硬件处理能力不匹配, 比如 I/O 基准代价没有考虑 SSD 和 HDD 实际能力的差异, 基于元组的 CPU 代价固定为 0.01; 其次, 统计信息不能及时准确捕捉数据分布的特征, 比如没有表的数据页在缓存中的数量等统计信息, 没有充分考虑内存大小对缓存命中率的影响, 而统计信息的准确度对代价估算模型中行数估算和代价估算至关重要.

③ 如何生成适应异构计算环境的代价最优的异构融合查询树, 指导查询处理引擎的执行, 实现各类计

算资源的协同优化.

支持异构融合的查询优化器是数据库系统在异构计算环境下高效执行、发挥全部硬件性能的关键, 也是保障查询高效、安全、提高系统健壮性的核心部件. 本文提出一种面向 CPU/GPU/FPGA 异构计算融合基于代价的查询优化器, 可以灵活调度并优化使用各类资源. 主要的贡献是: 发现根据系统环境硬件实际能力调整代价参数可显著提升查询计划的准确性, 并提出一种异构资源代价计算方法和校准工具; 通过对 GPU/FPGA 等异构硬件能力估算及对数据库系统硬件实际能力的校准, 建立异构计算环境下查询处理的代价模型; 实现了支持选择、投影、连接、聚合的 GPU 算子和 FPGA 算子, 实现了 GPU 算子融合及流水线设计、FPGA 算子流水线设计; 通过基于代价的评估解决算子分配和调度问题, 生成异构协同的执行计划, 实现异构计算资源的协同优化, 以充分发挥各异构资源的优势.

本文第 1 节介绍研究背景和相关工作. 第 2 节介绍 Geo 的设计, 重点阐述异构资源代价校准的方法、异构计算环境下查询处理代价模型的建立、基于代价评估的异构融合执行计划生成、异构算子融合和流水线设计. 第 3 节通过实验对异构资源代价校准的方法和工具进行验证, 设计异构算子对比实验分析验证 GPU/FPGA 的优势, 在 TPC-H 测试集上进行 Geno 与 CPU 数据库 PostgreSQL、GPU 数据库 Hetero-DB 的对比测试, 并对测试结果加以分析. 最后, 在第 4 节进行总结.

1 相关工作

查询优化是数据库系统最重要的任务之一, 数据库的性能非常依赖于查询优化器. 查询优化理论的诞生已有近 50 年, 学术界和工业界已经形成了多个比较完善的查询优化框架, 但是围绕查询优化的核心难题始终没变, 即如何利用有限的系统资源, 尽可能地为查询选择“好”的执行计划, 从而降低查询执行成本. 影响查询执行成本的因素包括关系的数量、通信成本、资源以及对大型分布式数据集的访问等, 查询优化被认为是 NP 难题[6].

从处理器发展的角度看, 随着硬件平台的变化, 数据库查询优化技术经历了以 I/O 代价为中心、以 Cache 数据局部性为中心、以多核并行处理性能为中心和以异构计算性能为中心的不同发展阶段. 不同发展阶段的查询优化目标具有显著的差异, 从减少面向磁盘的 I/O 数量, 到设计 cache-conscious 的数据结构和访问方法, 再到提高算法的并行处理性能[7]. 传统数据库的优化器是基于统计信息进行表连接规划, 随着新型处理器技术的发展, 处理器技术从多核走向众核, 在核心集成度、线程数量、缓存结构、访存技术等方面与传统的 x86 多核处理器架构有很大的不同, 查询优化的目标也转向 SIMD 优化技术[8]、面向 GPU、FPGA 等新型处理器和加速器的查询优化技术[9−11], 查询优化设计与硬件特征的依赖性越来越高. 在新硬件环境下, 数据库异构查询优化问题更加复杂, 除了传统优化器的基数估计问题和连接顺序问题等传统研究难题外, 还有异构计算环境带来的代价模型变化、异构存储层、PCIe 瓶颈等问题. 因此, 如何对这些异构系统的数据处理进行协同优化是一个极具挑战的研究问题.

(1) 代价估算和连接算法

基于代价的查询优化是基于采样信息和代价模型进行表连接规划, 对查询语句对应的待选执行路径进行代价估算, 从待选路径中选择代价最低的执行路径作为最终的执行计划. 传统模型的代价估算是将连续页面提取的成本、随机页面提取的成本、处理元组的 CPU 成本和执行操作的成本等多个因素结合起来估算的. 这些因素与查询所影响的数据基数强相关, 而且每个因子的权重必须调整. 在新型硬件环境下, 首先是要进行新型硬件的能力校准和异构算子的代价估算; 其次是传统代价模型无法刻画新硬件环境下各种操作的访问特征, 例如最主要的性能瓶颈已经从磁盘上顺序和随机 I/O 比例过渡到了 NVM 的读写比例; 再次, 异构计算架构下更深的缓存层级和非一致的缓存读写代价也进一步使得新型硬件环境下的访问代价估算复杂化、动态化. 因此, 如何确保新型硬件环境下代价模型的有效性和正确性, 是极具挑战性的研究工作.

Van Aken 等人[12]提出了一个通过人工智能的方法确定数据库系统的最佳配置参数系统 OtterTune,对不同的硬件、不同的负载确定更好的参数配置组合. 但主要是调整数据库系统已有的参数数值, 没有考虑异构硬

件. 同时, 该系统要实时地监控数据库运行状态, 某些参数的调整需要重启数据库才能生效.

快速并准确地估计多维度的选择率, 是现代关系查询优化器的重要组成部分. 该领域的最新技术是多维直方图, 它提供了良好的估计质量, 但构造复杂且难以维护. 内核密度估计(kernel density estimation, KDE)是一个有潜力的替代方法, 研究证明: KDE 比直方图收敛到基础分布更快, 且 KDE 模型易于构建和使用. 但现有的基于 KDE 的选择率估计器质量还需要提高. Heimel 等人[13]提出了一个在数值上快速并准确地估计多维度选择率的 KDE 模型, 能大大改善选择率估算, 并且该 KDE 模型能动态根据查询工作量变化而自动调整, 进而开发了 GPU 加速的自调整 KDE 模型, 使估算器能不断适应数据库和查询工作量变化. 同时, 为了快速估算选择率, 将选择率估算推送到 GPU 以提高性能. 但是优化的 KDE 模型只涉及表数据的选择率, 没有涉及表扫描和连接方式等优化以及估算, GPU 作用限于选择率估算, 不执行数据库操作符.

已处理查询的成功, 很大程度上取决于查询优化器所实施的搜索方法. 针对海量数据查询时经典的启发式方法(例如蚁群和遗传算法)无法覆盖所有搜索空间, 并可能导致陷入局部最小值的问题, 文献[14]采用量子启发式蚁群算法(QIACO)作为概率算法的一种混合策略, 以尽快找到使总执行时间最短的最佳连接顺序. 量子计算的多样化能力可以覆盖较大的查询空间, 这有助于选择最佳路径, 从而提高收敛速度, 避免陷入局部最优状态.

(2) 查询优化框架

Paul 等人[15]提出了一种新颖的流水线查询执行引擎 GPL, 指出现有的将 GPU 作为查询协处理器存在严重的资源效率不足, 给出了一个分析模型, 该模型可以确定参数的最佳配置, 以提高基于 GPU 查询协同处理的资源利用率. 执行引擎 GPL 通过确定 CPU 和 GPU 间高效地通信的最佳参数配置, 减少内存停顿, 进而提高 GPU 算子的性能.

Breβ[16]开发了一个异构查询处理引擎 HyPE, 结合 CPU 和 GPU 的优点进行基于代价的优化, 扩展了传统的查询优化器, 支持混合查询计划的生成, 以最大限度地提高系统性能. HyPE 采用学习方法估算算子的执行时间, 为算子分配最合适的处理器, 由 3 个部分组成: 代价估计器(cost estimator)、算法选择器(device-algorithm selector)、异构查询优化器(hybrid query optimizer). HyPE 优化器框架采用可移植分层设计, 可通过与特定数据库兼容的适配层, 理论上可以与任何 GDBMS 结合. 但是 HyPE 只在算子层级上决策算子在哪个计算核心上执行, 每个算子看成一个独立的个体, 采用贪心策略为算子分配处理器, 采用局部优化策略, 缺乏全局视野容易陷入局部最优解, 还可能造成不必要的数据传输代价. 同时, HyPE 无法对多计算节点的分布式计算环境进行优化.

Breβ 等人[17]提出一个负载感知的并行协同处理器 CoGaDB, 其核心是一个决策模型, 利用优化启发式算法 WTAR(waiting-time-aware response time), 根据多种协同器和 CPU 的工作负载及响应时间, 将数据库操作任务分配给各个异构设备, 获得最佳的计算资源利用率和总体性能. CoGaDB 的决策模型必须结合第三方优化器 HyPE 才能生成查询计划, 而且 CoGaDB 的协处理器限定在 GPU, 不支持 FPGA.

Zhang 等人[18]实现了一个基于异构计算和多种存储资源的下一代数据库系统 Hetero-DB, 它的 GPU 查询引擎通过两个关键组件确保高资源利用率和系统性能: 一个查询调度器, 在 GPU 上保持最佳的并发级别和工作负载; 一个数据交换机制, 最大限度地提高 GPU 设备内存的有效利用率. Hetero-DB 是从 GPU 内存空间的利用率等资源角度调度查询在 CPU 还是 GPU 上执行.

Owaida 等人[19]实现了一个 CPU-FPGA 混合架构的数据库系统 Centaur, 该系统提供一种与现有数据库架构兼容的使用 FPGA 加速 SQL 的解决方案, 为进一步探索基于 FPGA 的数据处理提供了可能性. Centaur 通过监视 FPGA 上的作业, 将查询执行计划中的算子动态调度到可用 FPGA 资源上, 多个算子在 FPGA 上流水线化执行, 这些算子流水线也可以跨 CPU 和 FPGA 混合运行. 同时, Centaur 完全兼容关系引擎,比如与流行的列存储数据库 MonetDB 无缝集成. Philippos 等人[2]做了一个 FPGA 加速在线分析处理的调查,指出其面临的两个主要挑战. (1) 主机内存和 FPGA 间数据吞吐量受限于 PCIe 总线带宽, 当前每个 PCIe 链接的速度约为 6.5 GB/s, 而现代 Intel 处理器的读取速度超过 80 GB/s, 写入速度达到 38 GB/s; (2) FPGA 加速效果受限于主机

内存和 FPGA 间数据传输时延. FPGA 加速器分为两类: 一类是框架, 包括 Centaur、DoppioDB 等专用框架和 MaxCompiler 通用框架; 另一类是算子加速, 包括用 FPGA 加速排序、选择、连接、字符串匹配、过滤等算子操作.

(3) 基于学习的查询优化

基于代价的查询优化存在着基数估计问题和连接顺序问题等传统研究难题. 最近几年, 随着人工智能技术的发展, AI4DB 成为热门研究方向, 数据库社区尝试利用机器学习模型来改进查询优化器. 基于学习的查询优化(AI based optimization, ABO)收集执行计划的特征信息, 借助机器学习模型获得经验信息, 进而对执行计划进行调优, 获得最优的执行计划. ABO 可以根据历史数据学习, 并根据系统现状在运行时进行动态调整.

文献[20]提出了一种查询优化的端到端学习方法 Neo (neural optimizer), 给定一组查询重写规则确保语义的正确性, Neo 学会做关于联接顺序、运算符和索引选择的决定. Neo 使用强化学习优化了这些决策, 根据用户的数据库实例, 并根据实际的查询延迟进行决策. Neo 的设计模糊了传统查询优化器的主要组件(基数估计、代价模型和计划搜索算法)之间的界限. Neo 没有明确估计基数估计及代价模型, 它将这两个功能组合为一个价值网络、一个神经网络, 采用部分查询计划并预测完成此部分计划可能产生的最佳预期运行时间; 通过价值网络的指导, Neo 对查询执行简单的搜索规划空间做决定; 随着 Neo 发现更好的查询计划, Neo 的价值网络也随之改善, 将搜索重点放在更好的计划上. 但是, Neo 仍然有许多重要的局限性: 首先, Neo 需要一个传统的查询优化器来引导它的学习过程; 其次, Neo 需要提前具备查询重写规则知识以保证正确性; 第三, Neo 查询优化器的特征是从一种特定的数据库中学习到的, 因此, Neo 不能从一种数据库泛化到另一种数据库, 不具备通用性.

针对代价和基数估计的挑战, 李国良等人[21]提出了一个基于树结构模型的端到端学习代价估算框架, 可以同时进行代价和基数的估算. 他们提出了一种有效的特征提取和编码技术, 在特征提取时, 同时考虑查询和物理执行, 将查询操作、元数据、查询谓词和一些样本编码到模型中. 他们把这些特征嵌入到树状结构模型中, 利用树状结构估计成本和基数. 模型包含了嵌入层、表示层和估计层, 能够有效地学习到每个子方案的表示, 并且这些表示用于成本和基数估计, 可以无缝取代传统的成本估计. 模型基于训练数据进行训练, 将更新的参数存储在模型中, 并估算新查询计划的成本和基数.

综上所述, 虽然近年来基于专用加速器对数据库进行加速的研究和应用取得了一定成果, 但还存在较大的局限和不足: 首先, 现有工作大都是使用单一异构算力针对特定场景的加速方法, 鲜有融合 CPU-GPU-FPGA 的加速研究, 也没有在通用数据库系统的成功应用; 其次, 针对异构计算环境下数据库查询优化问题的研究较少, 尚未出现有效解决新型硬件环境下数据库查询优化的解决方案. 已有的数据库加速研究都是采用蛮力或规则方式将定制 SQL 查询卸载到异构加速硬件上, 放弃了对异构算力的优化配合. 本文探讨了中兴通讯新一代数据库系统在新型硬件方向上的研究实践, 给出了建立支持新型硬件异构算力的代价模型的方法和工具, 并提出了支持新型硬件异构融合的查询优化器, 可以生成最优(次优)的异构融合执行计划, 充分发挥各类计算资源的优势.

2 Geno 的设计

SQL 是介于关系演算和关系代数之间的一种结构化查询语言, 它只关注结果而不关注过程, 即它只告诉数据库“想要什么”, 但是没有告诉数据库“如何获取”这个结果, 这个“如何获取”的过程是由数据库的“大脑”查询优化器来决定的. 在数据库系统中, 一个查询通常会有很多种获取结果的方法, 每一种获取的方法被称为一个执行计划, 优化器产生的执行计划的优劣, 直接决定数据库的性能. 因此, 查询优化在数据库系统中具有非常重要的作用.

通常数据库系统处理一条查询语句包含查询解析、查询优化和查询执行这 3 个阶段.

- 查询解析对查询语句作词法分析、语法分析和语义分析, 生成一个由关系算子组成的逻辑执行计划.
- 查询优化对逻辑执行计划进行优化, 最终生成物理执行计划. 传统的查询优化分为基于规则的优化

(rule based optimization, RBO)和基于代价的优化(cost based optimization, CBO)两种: CBO 首先对逻辑执行计划进行逻辑上的等价变换, 枚举出等价的执行计划; 然后, 查询优化器根据统计信息和代价模型为每个执行计划计算一个“代价”(这里的统计信息描述了数据的分布情况, 是由数据库统计模块自动产生的; 这里的代价通常是指执行计划的执行时间).

- 最后,查询优化器会在众多等价计划中选择一个“代价最小”的执行路径作为最终的执行计划,传给查询执行模块.

在包含 GPU/FPGA 等异构加速器的计算环境中, 为了充分发挥异构硬件的能力对数据库加速, 我们提出一种基于代价的异构融合查询优化器 Geno, 其主要模块和处理流程如图 1 所示. Geno 根据查询涉及的数据集大小、具体操作和硬件资源能力, 计算各种可能路径(包括异构执行路径)的代价, 选择代价最低的路径, 生成相应的计划节点及执行代码. 查询执行阶段依据生成的异构融合查询执行路径, 在各个计划节点执行指定的动态执行代码.

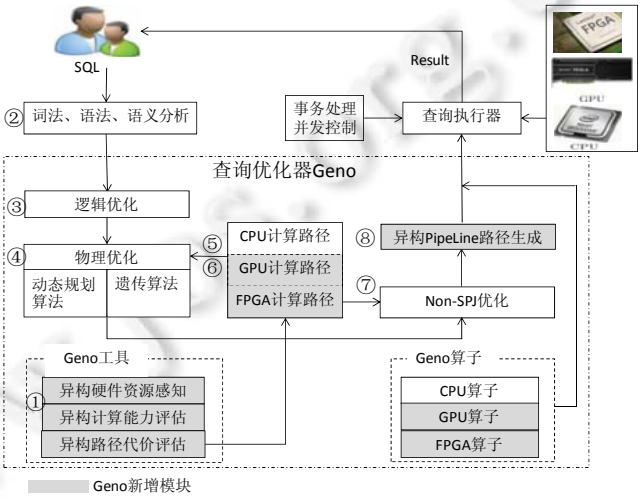


图 1 Geno 主要模块和处理流程

1) 与传统基于代价的优化器相比, Geno 新增的模块有:

(1) Geno 工具. 包括异构硬件资源感知、异构硬件计算能力评估、异构路径代价评估这 3 个部分. 异构硬件资源感知模块自适应感知和加载 GPU, FPGA 异构资源, 使数据库集成异构硬件计算能力. 异构硬件计算能力的评估完成不同异构硬件处理能力的比较折算, 根据硬件能力调整代价计算因子.异构路径代价评估完成不同异构硬件扫描路径、连接路径、聚合路径的代价评估.

(2) Geno 算子. 包括 GPU 算子、FPGA 算子: GPU 算子集成 GpuScan、GpuJoin、GpuAgg 等 GPU 支持的基本操作算子; FPGA 算子集成 FpgaScan、FpgaJoin、FpgaAgg 等 FPGA 支持的基本操作算子.

(3) GPU 计算路径. 为查询涉及的每个基表生成 GPUScan 扫描路径, 并调用异构路径代价评估模块, 计算其启动代价和总代价; 生成基表连接路径, 基于动态规划算法或遗传算法, 遍历基表之间各种可能的连接顺序及连接方式, 经过异构路径代价评估, 选择代价最低的连接路径, 生成 GpuJoin 连接路径以及该路径的启动代价和总代价. 在 Non-SPJ 优化阶段, 同时生成 GpuAgg 聚合路径, 得到其启动代价、总代价.

(4) FPGA 计算路径. 为查询涉及的每个基表生成 FpgaScan 扫描路径, 同样基于动态规划算法或遗传算法, 生成 FpgaJoin 连接路径, 在 Non-SPJ 优化阶段, 生成 FpgaAgg 聚合路径, 并调用异构路径代价评估模块, 计算 FPGA 各个路径的启动代价和总代价, 供优化器进行路径选择.

(5) 异构 PipeLine 路径生成. 为了减少主机与异构硬件之间的数据传输, 根据异构硬件特性, 采用算子融合、流水线设计等技术将多个 GPU 计算路径进行融合, 组合成一个新的组合算子, 支持 GpuScan+ GpuJoin+GpuAgg, GpuScan+GpuJoin, GpuScan+GpuAgg; FPGA 采用动态配置 SQL 流水线, 支持 FpgaScan+FpgaJoin+FpgaAgg.

2) Geno 处理流程:

(1) 运行异构计算能力评估工具, 计算并保存异构硬件的能力计算因子, 供 Geno 计算代价时使用. 数据库系统启动阶段, 调用异构硬件感知工具, 检测收集可用的异构硬件信息, 并进行初始化.

(2) SQL 语句经过数据库词法、语法、语义分析后生成查询树, 传递给 Geno 进行查询规划.

(3) Geno 首先对 SQL 查询进行基于规则的逻辑优化, 利用关系代数中一些等价的逻辑变换规则对查询进行等价变换, 如子链接和子查询提升、外连接消除、表达式预处理、连接条件和过滤条件下推, 生成新的等价约束条件等.

(4) 在逻辑优化生成新的查询树后, Geno 继续进行物理优化, 建立物理执行路径. Geno 默认使用动态规划算法来生成物理执行路径, 当查询涉及的基表数量大于设定的阈值时, Geno 选择遗传算法来生成路径, 这样可以保证在有限的时间内生成较优的查询计划.

(5) 路径生成第 1 阶段, 生成基表扫描路径及代价, Geno 根据收集的硬件信息, 调用 CPU 计算路径、GPU 计算路径、FPGA 计算路径模块中扫描路径生成接口, 为每个基表生成不同的扫描路径, 包括 SeqScan、IndexScan、GpuScan、FpgaScan, 根据 CPU 路径代价评估模型和异构计算路径代价评估模型计算每条路径的启动代价和总代价, Geno 为每个基表保存候选的扫描路径列表(总代价最优路径、启动代价最优路径、最优的排序路径).

(6) 路径生成第 2 阶段, 生成多表连接的路径及代价, Geno 采用动态规划算法或遗传算法, 遍历不同的连接顺序、连接方式, 调用 CPU 计算路径、GPU 计算路径、FPGA 计算路径模块中连接路径生成接口, 为连接表生成相应的连接路径及代价. 比较各个路径的代价, 淘汰较差的连接路径, 控制计划树规模. 同样地, Geno 为每个连接表保存候选的执行路径列表.

(7) 路径生成第 3 阶段进行 Non-SPJ 优化, 调用 CPU 计算路径、GPU 计算路径、FPGA 计算路径模块中聚合路径生成接口, 生成 Agg 路径及代价, 保存候选的路径列表.

(8) 在候选路径生成后, Geno 为了提高异构硬件执行效率, 减少主机与异构硬件之间不必要的数据传输, 调用异构 PipeLine 路径生成模块, 采用算子融合、流水线技术, 生成 PipeLine 路径及代价.

最后, Geno 根据以上生成的所有候选路径, 选择一条最优的查询路径生成混合查询计划树. 为提高算子的执行效率, Geno 利用 JIT 技术, 为每个计划节点生成专用执行代码, 将代码扁平化(inline), 直接调用对应的函数, 减少大量不必要的跳转和代码分支执行, 从而精减大量的执行指令. 查询执行器根据最终的查询计划树, 执行计划节点指定的操作, 通过多种异构硬件算子数据传输方法, 在多个计算资源上对数据进行读取、处理、存储等操作, 给用户返回最终查询结果.

2.1 异构加速算子的设计

2.1.1 GPU 算子设计

与常见的 GPU 数据库系统[22]一样, Geno 以核函数为载体执行 GPU 计算, 采用批处理、核函数融合等技术来设计 GPU 相关算子, 充分挖掘 GPU 的并行处理性能. Geno 将不同的 SQL 算子设计为 GPU kernel, 供 SQL 执行引擎调用. Geno 支持 GpuScan、GpuJoin (Hash, NestLoop)、GpuGroupBy、GpuAgg (MIN, MAX, SUM, COUNT)、GpuSortBy 等算子. GPU kernel 分 3 层设计: 第 1 层通用函数库层, 包括基本算术运算符、位操作运算符、数据类型转换、算术函数、三角函数、日期时间函数、文本函数、梯度求和、hash 等; 第 2 层为数据存取方法层, 包括行存、列存、数据块等; 第 3 层为关系运算符层, 包括选择、投影、排序、分组聚合和连接操作等.

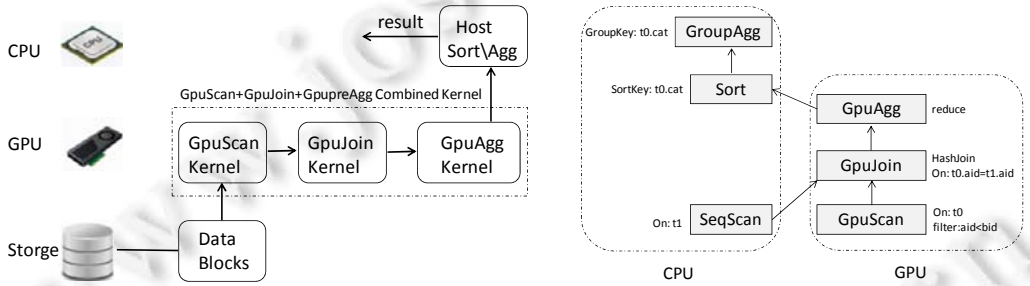
数据库系统利用 GPU 加速的性能瓶颈在于主机内存与 GPU 显存之间的数据拷贝, 可以类比于分布式数

据库中的网络通信开销, 这也是在代价估计中必须考虑的最重要的因素. 文献[23]等研究指出: PCIe 总线瓶颈是所有 GPU 算法不能忽视的性能开销, 也是 GDBMS 的性能瓶颈所在. 文献[24]对开源 Alenka 系统运行 TPC-H 基准测试并进行详尽的性能分析, 发现只有 5%用在了 GPU 计算上, 大部分开销都用在了数据传输上. 其次, 在多 GPU 环境的系统中, 由于多 GPU 往往共享 PCIe 总线, 因此多 GPU 下性能并不呈现简单的线性增长, 而且与之相关的管理成本和决策空间也相应变大. 因此, 如何在多 GPU 环境下优化查询性能仍然是一个难题. 以上因素共同作用造成了 PCIe 总线数据传输开销成为代价估算的重点和难点. Geno 为减少主机和 GPU 间数据传输, 采用了算子融合和流水线设计, 相应地, Geno 在 NO-SPJ 优化后, 增加异构 PipeLine 路径生成步骤.

1) GPU 算子融合设计

针对数据量大或者计算复杂的任务, Geno 通过分析核函数之间对数据的依赖性, 将承载处理相同数据的算子核函数合并融合在一起, 共同完成数据处理. 这样不但降低了数据在 CPU 和 GPU 之间的传输开销, 还可以减少整体核函数的数量, 简化查询处理的优化空间. Geno 支持 Scan+Join+Agg、Scan+Join、Scan+Agg 融合. 以 SELECT cat, count(*), avg(ax) FROM *t*0 NATURAL JOIN *t*1 WHERE aid < bid GROUP BY cat 为例, 展示 Scan+Join+Groupby 融合算子设计.

如图 2 所示, GpuScan 算子扫描过滤 *t*0 上的数据, 不经过主机直接传输给 GpuJoin 算子; GpuJoin 算子在 *t*0, *t*1 上运行 JOIN 操作, 也不经过主机直接传输给 GpuAgg 算子, GpuAgg 算子通过 GPU 设备上的 cat 列运行 GROUP BY 操作后才传输给主机.



(a) 融合算子设计 (b) 执行计划树

图 2 GPU 算子融合

2) GPU 算子流水线设计

有研究提出对核函数进一步进行切分, 通过更精细的粒度管理和流水线化调度, 提高 GPU 资源利用率. 但是核函数切分是以更频繁的核函数调度为代价, 数据传输代价和 PCIe 总线瓶颈的影响更大. Geno 是从数据执行粒度上精细化核函数的执行, 利用核函数计算和数据 I/O 交替使用不同硬件的特点, 通过切分核函数和核函数之间数据的流水化, 有效提高数据并发度.

如图 3 所示, 数据以 chunk 块为单位进行调度和处理.

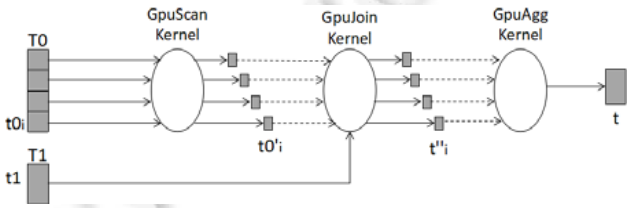


图 3 GPU 算子流水线

输入关系表(例如 *T*0 和 *T*1), 把 *T*0 表数据切分为 chunk 块 $t0_i$ 作为 GpuScan Kernel 的输入数据, 经过核函数扫描过滤后, 产生数据块 $t0'_i$ 并作为 GpuJoin Kernel 的输入数据, GpuJoin 核函数同时加载内表 *T*1 的数据进

行连接, 产生连接关系结果集 t_i'', 输入到 GpuPreAgg Kernel 进行分组操作, 产生 GPU 最终输出结果 t. 如果内表 $T1$ 超过 GPU 内存无法加载, 则采用 Grace Hash join 算法, 在连接前先对 $T0$/$T1$ 表进行 Hash 分片, 并将分片文件缓存到磁盘, 然后加载成对的 $T0$/$T1$ 表分片文件, 执行 GpuJoin 进行 Hash 连接.

2.1.2 FPGA 算子设计

FPGA 是一种新型异构计算硬件, 存在出错和失效的可能. 对于 FPGA 的使用, 无法像 CPU 一样透明无感知, 数据库引擎需要根据 FPGA 的状态来做出相应的管理和任务调度. 提出一种 FPGA 即服务(FPGA as service)架构, 将 FPGA 作为从数据库引擎中独立出来, 将 FPGA 的计算能力视作一种服务提供给 SQL 计算引擎, 从而降低 SQL 模块间的耦合性和整体的可用性. FPGA 即服务提供 FPGA 硬件本身的运行状态信息、FPGA 中计算资源的忙碌空闲状态、FPGA 中算子的计算能力信息等信息.

为了使用 FPGA 加速 SQL 处理, 需要将 SQL 算子下推到 FPGA 中进行处理. 因此, 需要设计 SQL 算子在 FPGA 执行的逻辑. 具体来说, 是要将 SQL 算子设计为 kernel, 以供 SQL 引擎调用. Geno 支持 fpgaScan、fpgaFilter、fpgaJoin (Hash, Merge)、fpgaSortBy、fpgaGroupBy、fpgaAgg (MIN, MAX, SUM, COUNT) 等常用 SQL 算子.

FPGA kernel 的设计自底向上分为 3 层: 最底层是通用基本算子层, 提供最基本的算法功能模块以供上层组合调用, 包括基本的压缩解压算法、排序算法、基本运算符、bloomfilter 和 hash 等; 中间层是 SQL 算子层, 利用通用基本算子层提供的基本功能组合出相应的 SQL 算子, 包括选择、投影、分类、分组聚合和连接操作等, 例如使用压缩解压算法、bloomfilter 等算法模块组合出带有解压数据和过滤功能的扫描算子, 使用 hash 和基本运算符模块组合出哈希连接算子等; 最上层是软件接口层, 向下对中间层的 SQL 算子进行抽象, 向上为用户提供友好易用的 API 调用接口. 整个设计自底向上层层抽象又相对独立, 具有较好的扩展性和易用性.

1) FPGA 算子动态可配置的 SQL 流水线设计

一般来说, 查询优化器会将某一个或者几个 SQL 算子的处理下推到 FPGA 中. 在面对复杂查询时, 数据在 FPGA 和主机间的频繁传输难以避免, 从而会大大增加了处理延迟和 I/O 的代价, 削弱了 FPGA 加速的效果. 为消除不必要的数据传递, 我们提出了一种动态可配置的 SQL 流水线设计.

该设计的思路是: 在 FPGA 硬件中配置一条 SQL 节点的流水线, 流水线上的节点使用与否可以通过参数动态地控制, 从而达到适配大部分查询并且消除与内存间不必要的数据交换的目的(如图 4 所示).

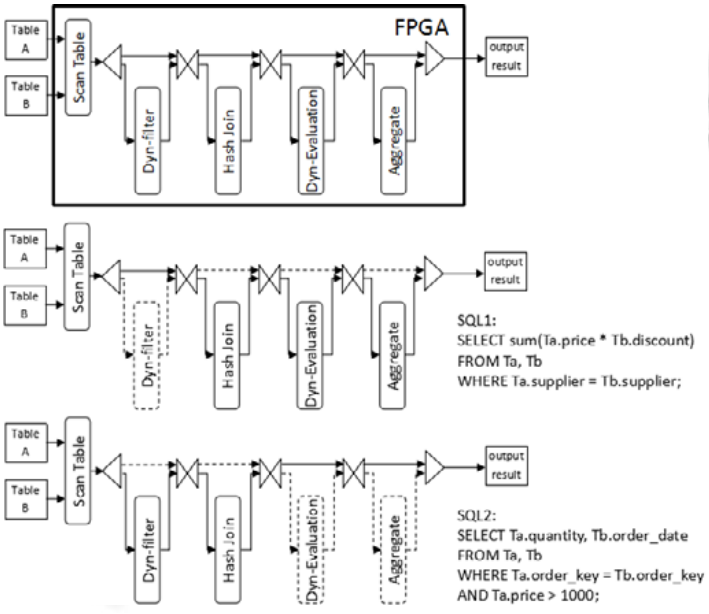


图 4 FPGA 流水线

2.2 异构计算代价模型

代价(*cost*)是数据库内核对给定 SQL 任务执行成本的估计, 是一个相对值, 以顺序读取一个页面的时间(*seq_page_cost*)为基准单位 1, 其他代价都是相对这个基准单位来计算. 一条 SQL 语句的代价估算主要从 I/O 代价、CPU 计算两方面考虑. 首先, 根据统计信息和查询条件, 估算出本次查询需要进行的 I/O 数量以及要获取的元组个数; 然后, 根据代价参数计算出 I/O 代价和 CPU 代价; 最后, 综合考虑 I/O 代价和 CPU 代价即可得到总代价. 代价估算依赖表和索引占据的磁盘块数、表的元组数、符合条件的元组数、各个属性出现的次数等统计信息, 还需要用到顺序读取一个页面的代价(*seq_page_cost*)、随机读写页面的代价(*random_page_cost*)、CPU 处理一条元组的代价(*cpu_tuple_cost*)、CPU 处理一条索引元组的代价(*cpu_index_tuple_cost*)、CPU 处理一个运算符的代价(*cpu_operate_cost*)、一个查询中所有表的可用缓冲大小(*effective_cache_size*)等代价参数. 因为表的扫描方式、表的连接顺序和表的连接方式不同, 相同的 SQL 语句有不同的执行路径, 对应的代价也不同, 路径的总代价是启动代价和执行代价两者之和. 查询优化器就是从众多可选的路径中选择一条总代价最低的路径.

传统数据库的代价模型没有考虑系统硬件如 HDD/SSD 的实际能力差异, 直接将 *random_page_cost* 等代价参数固定, 也没有充分考虑内存大小对缓存命中率的影响, 因此主机上 I/O 代价和 CPU 计算代价需要校准. 在异构硬件环境下, 计算代价不仅仅包括 CPU 计算代价, 还包括 GPU、FPGA 等异构硬件计算代价. 而对异构硬件的支持, 通常采用经验值进行固定配置. 在不同的硬件运行环境中, 固定配置与硬件实际表现出的能力相比差距比较大, 因此异构计算代价相对 CPU 计算代价必须根据实际硬件能力进行校准. 同时, 因为异构硬件的加入, 还带来了主机内存与异构硬件设备内存间的数据传输, 这也是在代价估算时必须考虑的重要因素.综合以上因素, Geno 提出了异构硬件环境下的代价估算模型, 如公式(1)所示.

$$TotalCost=newCost_{I/O}+newCost_{compute}+Cost_{trans} \tag{1}$$

即, 一个计划的总代价由根据实际存储介质校准后的 CPU、GPU、FPGA 等 I/O 代价、根据硬件能力校准后的 CPU、GPU、FPGA 等计算代价、主机内存与 GPU、FPGA 等异构硬件设备内存间的传输代价这 3 个部分组成.代价计算是一个自底向上的过程: 首先计算扫描算子的代价, 然后根据扫描算子的代价计算连接算子的代价以及 Non-SPJ 算子的代价.

2.2.1 异构计算代价校准

- 主机 I/O 代价校准: *seq_page_cost* 是顺序读取一个页面的代价. *seq_page_cost* 作为整个代价估算体系的基准代价, 是衡量其他操作代价的基准. 校准采用通用的磁盘 I/O 测试工具 fio, 测试顺序扫描数据的时间作为代价的基准时间 *T*0, 其他操作耗时以这个为基准进行换算. *random_page_cost* 是随机读取一个页面的代价, 校准方法和 *seq_page_cost* 相同.
- 主机计算代价校准: 包括 *cpu_operator_cost*、*cpu_tuple_cost*、*cpu_index_tuple_cost* 这 3 个代价参数的校准.
  - *cpu_operator_cost* 是 CPU 执行一个运算符或函数调用的代价, 该值为表达式计算的代价评估提供重要参考. 由于浮点数处理是衡量硬件性能的重要指标, 所以选择浮点数相乘作为测试方法. 校准方法采用在函数内循环调用浮点数相乘, 将单次运算符执行时间除以基准时间 *T*0, 得出运算符代价.
  - *cpu_tuple_cost* 是从主存中的页面读取并解析一个元组的代价, 与元组中字段数量和类型相关. 为了测试类型覆盖的全面性, 校准时创建包含 int, string, float 等常见类型的字段, 批量生成测试数据, 测试总解析时间, 用总时间除以元组数, 得出单个元组解析耗时, 单个元组解析耗时再除以基准时间 *T*0, 得出元组处理代价.
  - *cpu_index_tuple_cost* 是从主存中的页面读取并解析一个索引元组的代价. 除了表上有构建索引, 校准思路和方法与 *cpu_tuple_cost* 相同.
- GPU 计算代价校准: 主要校准 *gpu_ratio* 参数. *gpu_ratio* 表示 GPU 与 CPU 运算代价折算比, 用来估

算在 GPU 上执行同样操作的代价, 为选择 GPU 还是 CPU 执行操作提供重要参考. 此处同样选择浮点向量相乘作为测试标准. 校准采用 *A*, *B*, *C* 这 3 个浮点向量, 用 *A* 与 *B* 对应位置元素相乘存入 *C* 对应位置, 分别在 GPU 上和 CPU 上进行浮点运算, 用 GPU 耗时除以 CPU 耗时, 得出 GPU 与 CPU 运算能力的折算比.

- FPGA 计算代价校准: 主要校准 *fpga_ratio* 参数. *fpga_ratio* 表示 FPGA 与 CPU 运算代价折算比, 用来估算在 FPGA 上执行同样操作的代价, 为选择 FPGA 还是 CPU 执行操作提供重要参考. 同 GPU 计算代价校准, 选择 *A*, *B*, *C* 浮点数向量相乘作为测试标准, 分别在 FPGA 上和 CPU 上进行浮点运算, 用 FPGA 耗时除以 CPU 耗时, 得出 FPGA 与 CPU 运算能力的折算比.

2.2.2 索引扫描 I/O 代价校准

索引扫描会随机访问在磁盘上数据页, 如果需要访问的数据页已经存在于缓冲池中, 则不需要从磁盘上读取; 反之, 则会从磁盘上读取数据页, 产生磁盘 I/O 的代价. 索引扫描的性能在很大程度上取决于实际需要访问的物理磁盘的页面数. 对于存在于缓冲池中的数据页, 可能由于其他并发访问的会话需要加载其他数据页, 而把之前的数据页替换出去, 导致下一次访问需要从物理磁盘上重新加载该数据页, 并发访问数越大,对缓冲池资源的争抢也越激烈. 另外, 缓冲池越大, 系统可容纳的数据页越多, 数据页被替换出去的可能性也越低. 所以, 并发的请求数和缓冲池的大小成为决定物理页被访问次数的关键因素. 主流数据库中, 现有计算物理页访问次数的方法采用了 Mackert 等人[25]建立的模型.

1) 当 $T \leqslant b$ 时, 物理页访问次数为 $\min(2TNs/(2T+Ns), T)$.
2) 当 $T>b$ 且 $Ns \leqslant 2Tb/(2T-b)$时, 物理页访问次数为 $2TNs/(2T+Ns)$.
3) 当 $T>b$ 且 $Ns>2Tb/(2T-b)$时, 物理页访问次数为 $b+(Ns-2Tb/(2T-b))*(T-b)/T$.

其中, T 为当前表中页面数量, N 为表中元组数量, s 为约束条件产生的选择率, b 为当前表可用缓存大小.

b 的计算公式为 $b=effective\_cache\_size*T/total\_pages$, 其中, *effective_cache_size* 是指在一个查询中所有表的可用缓冲大小, 假设均匀分布, 那么就可以推定当前表的可用缓存大小; *total_pages* 是指当前查询中涉及的所有表的总页面数. 对于一个确定的查询来说, T 和 *total_pages* 都是确定值, *effective_cache_size* 的值的准确性决定了 b 值的准确性. 更进一步地, 在上述模型中, T, N, s 都是确定值, b 值的准确性决定了物理页访问次数的准确性. 因此, *effective_cache_size* 值的准确性决定了物理页访问次数的准确性. 主流数据库中, *effective_cache_ size* 使用固定值 524 288 (4 GB). 显然, 这不是一个准确的值, 没有体现缓冲区大小、并发查询数对估算物理页访问次数的影响, 导致估算的物理页访问次数也不可能准确.

Geno 通过参数校准工具, 在实验基础上总结提出公式(2).

$$effective\_cache\_size=a0+a1*X+a2*Y+a3*X^2+a4*X*Y+a5*Y^2 \tag{2}$$

其中, X, Y 分别代表 buffer pool 大小、并发查询数, 其余参数均为系数($a0$=4150, $a1$=0.001735, $a2$=−89.42, $a3$=−8.039e−05, $a4$=0.004912, $a5$=0.8386). 该公式描述了 *effective_cache_size* 和并发查询数、buffer pool 大小之间的关系, 在查询请求处理过程中, 输入 buffer pool 大小、并发查询数得到 *effective_cache_size* 最佳值, 这个值进一步提升了索引扫描代价估算的准确性.

2.2.3 异构计算传输代价估算

- *gpu_trans_cost*: 主机和 GPU 之间传输数据的代价. 因 GPU 并行计算能力强大, 计算代价小, 因此传输代价对 GPU 总代价影响巨大. 校准时, 将 chunk 块从 CPU 传到 GPU, 用总传输耗时除以基准时间 $T0$ 得出 GPU 传输代价.
- *fpga_trans_cost*: 主机和 FPGA 之间传输数据的代价. 和校准 GPU 和主机间的传输代价相似, 校准时, 将 chunk 块从 CPU 传到 FPGA, 用总传输耗时除以基准时间 $T0$, 得出 FPGA 传输代价.

2.3 异构融合查询优化

根据异构计算代价模型, Geno 利用校准后代价和异构硬件间传输代价进行路径的代价估算, 从众多可选的路径中选择一条总代价最低的路径, 并根据语句实际执行时间动态更新统计信息, 并用于后续自动重优化.

2.3.1 异构计算路径代价模型

Geno 在传统 CBO 优化器基础上增加了 GPU, FPGA 上运行的 Scan, Join, Agg 等路径的代价估算, 用于优化器筛选最优查询路径. 异构路径代价估算相关 GPU 参数表参见表 1.

表 1 异构路径代价估算相关参数表

符号	定义	来源
C_s	一条路径的启动代价	优化器
C_r	一条路径的执行代价	优化器
C	一条路径的总代价	优化器
K	GPU 核函数动态编译加载代价	校准参数配置
T	主机与 GPU 之间传输一个 chunk 块(64 MB)的代价	校准参数配置
R	GPU 与 CPU 计算能力折算比	校准参数配置
C_{op}	GPU 执行一个运算符的代价	校准参数配置
S_delay	GPU 处理首个 chunk 块代价, 即 GPU 启动延时代价	校准参数配置
N_c	GPU 扫描的 chunk 个数	优化器
N_t	GPU 扫描的记录条数	系统表
N_p	基表占用磁盘总页数	系统表
E_s	表达式启动代价	系统表
E_r	表达式一次执行代价	系统表
R_e	表达式条件的选择率	系统采样表
S_{8K}	顺序读一个 8K 页面的代价	校准参数配置
D_s	主机发送到 GPU 的数据大小(chunk 数)	优化器
O_s	外表扫描路径的启动代价	优化器
O_r	外表扫描路径的执行代价	优化器
O_t	外表的记录总数	优化器
O_c	外表数据大小(chunk 数)	优化器
I_load	内表数据预加载代价	优化器
num_inner	连接内表的个数	优化器
I_t	内表的记录数	系统表
I_{ck}	内表 K 扫描的总代价	优化器
I_{hk}	内表 K 建立 Hash 表的代价	优化器
I_{dk}	内表 K 的数据传输代价	优化器
C_h	Hash 连接操作的代价	优化器
J_t	连接表记录数	优化器
J_c	连接表数据大小	优化器
E_{js}	连接表达式启动代价	系统表
E_{jr}	连接表达式一次执行代价	系统表
E_{gps}	Groupby 分组表达式启动代价	系统表
E_{gpr}	Groupby 分组表达式一次执行代价	系统表
E_{aggs}	主机最终聚合表达式启动代价	系统表
E_{aggr}	主机最终聚合表达式一次执行代价	系统表
N_{gp}	分组 Key 的个数	SQL 语句
C_{gp}	GPU 分组操作总代价	优化器
C_{agg}	GPU 聚合操作总代价	优化器
A_t	GPU 聚合操作输出总记录数	优化器
A_c	GPU 聚合操作输出数据大小(chunk 数)	优化器
C_{outer_s}	FPGA 外表扫描路径的启动代价	优化器
K_{fpga}	启动 FPGA 的代价	优化器
C_{build}	内表数据传入 FPGA 建立 HASH 表的代价	优化器
C_{inner_s}	FPGA 内表扫描路径的代价	优化器
C_{inner_trans}	FPGA 内表数据传输代价	优化器
C_{hash}	FPGA 内表数据建立 hash 表的代价	优化器
C_{out_trans}	FPGA 外表数据传输代价	优化器
C_{probe}	FPGA 外表数据执行 probe 的代价	优化器
$inner\_tuples$	FPGA 的 join 算子中内表的元组数量	系统表
$hash\_cost$	FPGA 中单个 hash 单元执行 hash 操作的代价	校准参数配置
$hash\_concurrency$	FPGA 中并发执行的 hash 单元个数	优化器
$pcie\_speed$	FPGA 与主机间 PCIe 传输速度	校准参数配置
$outer\_tuples$	FPGA 的 join 算子中外表的元组数量	系统表

GPU 的异构计算路径代价新增了主机与异构硬件之间的数据传输代价、核函数动态编译加载代价、GPU 启动延时代价、GPU 并行计算代价等. GPU 启动延时代价是 GPU 处理首个 chunk 的代价 $S\_delay=C_r/N_c$, N_c 代表 GPU 处理数据的 chunk 总数.

1) GPUScan 路径代价计算

(1) 启动代价 C_s 是核函数动态编译加载代价 K 与启动延时代价 S_delay 之和.

(2) 执行代价 C_r 包括表数据加载 I/O 代价、过滤条件表达式计算代价、主机与 GPU 间数据传输代价这 3 部分, 如公式(3)所示.

$$C_r=S_{8K}*N_p+E_r*R*N_t+T*(D_s+D_s*R_e) \tag{3}$$

其中, S_{8K} 代表顺序读取一个 8K 页面的代价, N_p 代表基表占用磁盘总页数, E_r 代表表达式一次执行代价, R 代表 GPU 与 CPU 计算能力折算比, N_t 代表 GPU 扫描的记录条数, T 代表主机与 GPU 之间传输一个 chunk 块的代价, D_s 代表主机发送到 GPU 的 chunk 数, R_e 代表表达式条件的选择率.

GPU 支持通过 DMA 传输直接从 NVMeSSD 盘上加载表数据, 避免从主机内存拷贝数据至 GPU 设备内存, 节约总线带宽并显著降低传输代价. 以 DMA 方式扫描表的执行代价调整为 $C_r=S_{8K}*N_p+E_r*R*N_t+T*D_s*R_e$.

2) GPUHash 连接路径代价计算

(1) Hash 连接路径的启动代价 C_s 为核函数动态编译加载代价、外表扫描路径启动代价、内表数据预加载代价、选择投影等表达式的启动代价、GPU 启动延时代价之和. 内表数据预加载代价为所有内表数据扫描代价加上数据传输代价以及所有内表构建 HASH 表的代价, 如公式(4)所示.

$$C_s=K+O_s+I\_load+E_s+S\_delay \tag{4}$$

其中, K 为核函数动态编译加载代价; O_s 为外表扫描路径启动代价; E_s 为表达式启动代价; S_delay 为启动延时代价; I_load 为内表数据预加载代价, 计算式为 $\sum_{k=1}^{num\_inner}(I_{ck}+I_{dk}+I_{hk})$, I_{ck} 为内表 K 扫描的总代价, I_{dk} 为内表 K 的数据传输代价, I_{hk} 为内表 K 建立 Hash 表的代价.

(2) Hash 连接路径执行代价 C_r 包括外表扫描路径的执行代价、外表和连接表数据传输代价、Hash 连接操作代价、连接表投影表达式的 GPU 计算代价, 如公式(5)所示.

$$C_r=O_r+T*(O_c+J_c)+C_h+E_r*R*J_t \tag{5}$$

其中, O_r 代表外表扫描路径的运行代价; O_c 代表外表数据大小(chunk 数); J_c 代表连接表数据大小; C_h 代表 Hash 连接操作的代价; J_t 代表连接表记录数; T, E_r, R 同公式(3).

3) GPU NestLoop 连接路径代价计算

(1) NestLoop 连接路径启动代价 C_s 包括核函数动态编译加载代价、外表扫描路径的启动代价、所有内表数据预加载代价、投影和连接表达式的启动代价、GPU 启动延时代价, 如公式(6)所示.

$$C_r=K+O_s+I\_load+E_s+S\_delay \tag{6}$$

其中, K, O_s, E_s, I_load, S_delay 同公式(4).

(2) NestLoop 连接路径代价 C_r 包括外表扫描路径的执行代价、主机与 GPU 之间数据传输代价、NestLoop 连接 GPU 并行计算代价、投影表达式的执行代价, 如公式(7)所示.

$$C_r=O_r+T*(O_c+J_c)+E_{jr}*R*O_t*I_t+E_r*R*J_t \tag{7}$$

其中, O_r, T, O_c, J_c, R, E_r, J_t 同公式(5); E_{jr} 代表连接表达式一次执行代价; O_t 代表外表的记录总数; I_t 代表内表的记录总数.

4) GPU Agg 路径代价估算

(1) Agg 路径启动代价 C_s 包括核函数动态编译加载代价、主机与 GPU 数据传输代价、外表扫描路径的总代价、GPU 分组计算代价、GPU 聚合计算代价和主机上最终聚合操作计算启动代价, 如公式(8)所示.

$$C_r=K+T*(O_c+A_c)+(O_s+O_r)+C_{gp}+C_{agg}+E_{aggs} \tag{8}$$

其中, K, T, O_c, O_s, O_r 同上; A_c 代表 GPU 预聚合操作输出数据大小(chunk 数); C_{gp} 代表 GPU 分组操作总代价;

C_{agg}代表 GPU 聚合操作总代价; E_{aggs}代表主机最终聚合表达式一次执行代价.

(2) Agg 路径运行代价 C_r为 E_{aggr}与 A_t的乘积, 即 $C_r=E_{aggr}+A_t$, 其中, E_{aggr}代表主机最终聚合表达式一次执行代价, A_t代表 GPU 聚合输出记录条数.

5) GPU Pipeline 路径代价调整

上层 GPU 路径与下层 GPU 路径进行一次核函数融合, 启动代价就要减少一次核函数动态编译加载代价, 即, 启动代价 $C_s=C_s-K$. 执行代价减少一次 CPU 与 GPU 之间发送、接收数据的传输代价, 如公式(9)所示.

$$C_r=C_r-T*(O_c+J_c)\text{或 } C_r=C_r-T*(O_c+A_c) \tag{9}$$

其中, K, T, O_c, A_c同公式(8); J_c代表连接表数据大小.

6) FPGA 路径代价估算

由于以下原因, 针对 FPGA 算子的执行代价估算没有通用的估算标准, 需要根据算子的设计具体地分析.

(1) FPGA 的频率(300 MHz 左右)本身要比 CPU(GHz 级别)慢很多.

(2) FPGA 和 GPU 的加速原理不同, 不完全通过多核特性加速 SQL, 很大程度上依赖 HBM 的访问速度取胜(HBM 的访问速度要比 DDR 要高).

(3) FPGA 的加速效果也依赖于特定的电路的设计和优化, 同样的算法, 不同的设计和优化导致的表现会有很大差异, 和 FPGA 本身的能力不完全成正比. 因此, 计算特定 kernel 的代价更有意义.

下面以具体的 Hash Join 设计为例, 分析 FPGA 中 SQL 算子的计算代价公式的建立. 在图 5 所示的 FPGA 设计中, 整个 join 过程分为两个阶段: Build 阶段+Probe 阶段.

- Build 阶段: 将需要做 Hash 的内表通过 HBM 的多个通道并行导入, 根据 SQL 的连接条件分布到不同的 build 模块中并行地进行 Hash 操作, 快速地生成 Hash 表并存储到 HBM 中.
- Probe 阶段: 将外表数据通过 HBM 的多个通道并行导入, 同样根据 SQL 的连接条件分布到不同的 build 模块中并行地进行 Hash 操作, 通过计算到的 Hash 值在 HBM 上存储的 hash 表中进行匹配操作, 匹配记录输出到 collect 模块后, 再收集输出至 host.

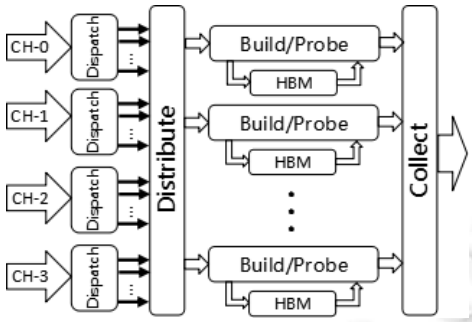


图 5 FPGAJoin 算子设计

由于内表数据和对应的 Hash 表需要存储在 FPGA 内部的 HBM 上, 因此内表的大小不能超过 HBM 的大小; 而由于外表数据只进行 Hash 计算和匹配, 因此理论上外表无大小限制.

针对以上 Hash Join 的设计, 将 Hash Join 路径代价分为启动代价 C_s和执行代价 C_r分别进行分析.

(1) 启动代价 C_s是外表扫描路径的启动代价 C_{outer_s}、启动 FPGA 的代价 K_{fpga}与内表数据传入 FPGA 建立 HASH 表的代价 C_{build}三者之和. C_{build}可以表示为内表扫描路径的代价 C_{inner_s}、内表数据传输代价 C_{inner_trans}与内表数据建立 hash 表的代价 C_{hash}之和. 而内表数据是通过 PCIe 总线传输到 FPGA 中的, 因此内表数据传输代价为 C_{inner_trans}为 $inner\_size/pcie\_speed$, 其中, $inner_size$是内表数据大小, $pcie_speed$是 PCIe 传输速度. 内表数据建立 hash 表是通过内部多条并行 hash 单元实现的, 因此, 内表数据建立 hash 表的代价如公式(10)所示.

$$C_{hash}=\frac{inner\_tuples*hash\_cost}{hash\_concurrency} \tag{10}$$

其中, *inner_tuples* 是内表的元组数量, *hash_concurrency* 是并发执行的 hash 单元个数, *hash_cost* 是单个 hash 单元执行 hash 操作的代价.

因此, Hash Join 路径的启动代价如公式(11)所示.

$$C_s = C_{outer\_s} + K_{fpga} + C_{inner\_s} + \frac{inner\_size}{pcie\_speed} + \frac{inner\_tuples * hash\_cost}{hash\_concurrency} \tag{11}$$

(2) 执行代价 C_r 是外表数据传入 FPGA 的代价与执行 probe 操作的代价之和. 外表数据同样是通过 PCIe 总线传输到 FPGA 中的, 因此外表数据传输代价 C_{outer_trans} 为 *outer_size*/*pcie_speed*, 其中, *outer_size* 是内表数据大小, *pcie_speed* 是 PCIe 传输速度. 而外表数据执行 probe 是通过内部多条并行 hash 单元实现的, 因此外表数据执行 probe 的代价如公式(12)所示.

$$C_{probe} = \frac{outer\_tuples * hash\_cost}{hash\_concurrency} \tag{12}$$

其中, *outer_tuples* 是外表的元组数量, *hash_concurrency*, *hash_cost* 同公式(10).

因此, Hash Join 路径的执行代价如公式(13)所示.

$$C_r = \frac{outer\_size}{pcie\_speed} + \frac{outer\_tuples * hash\_cost}{hash\_concurrency} \tag{13}$$

7) FPGA 算子流水线路径代价调整

针对 Hash Join 设计, 减少了内表和外表的传输代价.

(1) 启动代价 C_s 是外表扫描路径的启动代价 C_{outer_s}、启动 FPGA 的代价 K_{fpga} 与内表数据传入 FPGA 建立 Hash 表的代价 C_{build} 三者之和. C_{build} 可以表示为内表扫描路径的代价 C_{inner_s} 和内表数据建立 hash 表的代价 C_{hash} 之和. 因此, Hash Join 路径的启动代价如公式(14)所示.

$$C_s = C_{outer\_s} + K_{fpga} + C_{inner\_s} + \frac{inner\_tuples * hash\_cost}{hash\_concurrency} \tag{14}$$

(2) 执行代价 C_r 只包含执行 probe 操作的代价,如公式(15)所示.

$$C_r = \frac{outer\_tuples * hash\_cost}{hash\_concurrency} \tag{15}$$

2.3.2 异构融合计算路径生成

查询优化器的主要任务分成逻辑优化、物理优化、No-SPJ 优化这 3 个阶段. 逻辑优化阶段针对查询树做等价代数变换, 主要是提升子查询/子链接以及对常量/表达式计算等进行预处理等, 与硬件能力无关, Geno 不涉及此阶段. 物理优化阶段主要是生成基本查询语句最小代价的路径, 路径用一棵树来标识, 叶节点是基本表的扫描路径, 中间节点是表的连接路径. 基本表的扫描方式、两张表的连接方式和连接顺序都影响路径的代价, 在物理优化阶段采用动态规划或者遗传算法选择一条最优路径. 动态规划是一个自底向上逐层构建的过程, 以 *N* 张表为例, 构建步骤: 第 1 层, 针对每张基本表, 计算顺序扫描、索引扫描等代价, 生成最优的表扫描路径, 这样生成了一张基本表的扫描路径; 第 2 层, 选取第 1 层的两张表进行连接, 根据表的连接方式(嵌套循环连接、归并连接、Hash 连接)和连接顺序(左连接、右连接)计算代价, 生成最优的连接, 这样生成了两张表的最优连接路径; 第 3 层, 选取第 2 层的两张表或者第 2 层的一张表与第 1 层的一张表进行连接, 根据表的连接方式和连接顺序计算代价, 生成最优的连接, 这样生成了 3 张表的最优连接路径. 以此类推, 直到第 *N* 层, 确定全部 *N* 张表的最优连接路径. 动态规划算法的时间复杂度随着表数量指数级上升, 因此当连接表数量超过指定阈值后, 为减少生成路径的时间, 采用遗传算法. 遗传算法是一种启发式算法, 不一定能找到最好的路径, 只能找到相对较优的路径. 将一张表的编号作为一个基因, *N* 张表的一种可能连接路径作为一个染色体, 步骤如下: 随机初始化一个染色体(一条路径)种群, 计算每个染色体的适应值(一条路径的代价)并排名, 按照一定策略选择两个父代染色体 *A*, *B*, 父代染色体 *A*, *B* 进行杂交、变异, 生成新染色体 *C*, 计算新染色体 *C* 的适应值, 插入种群中, 并去掉排名最后的染色体. 达到一定的迭代次数后, 遗传算法终止, 选择适应值最小的染色体, 生成 *N* 张的最优连接路径顺序. NO-SPJ 阶段在物理优化生成路径基础上, 根据查询语句中 AGG、

Group by、Order by、distinct、limit 等子句中添加 Agg 路径(普通 Agg、排序 Agg、哈希 Agg)、Group by 路径、Order by 路径、limit 路径、distinct 路径等, 生成完整路径.

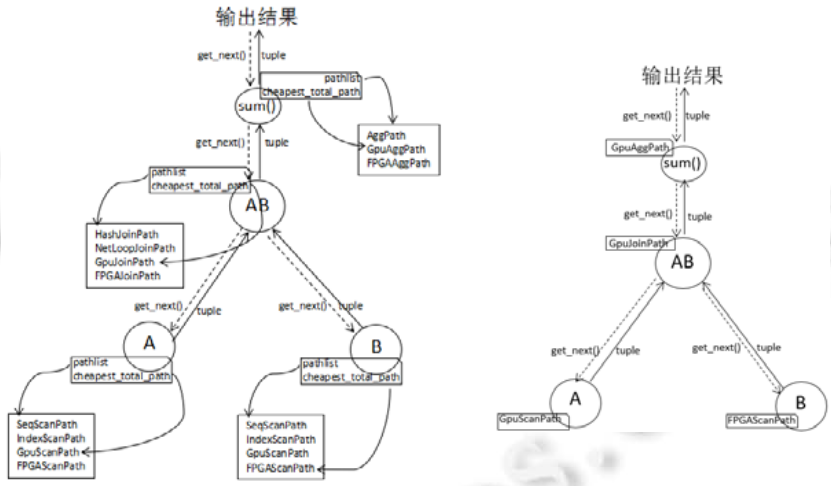
物理优化、No-SPJ 优化阶段都涉及代价估算和路径生成, 是需要 Geno 进行优化的阶段. 在扫描路径、连接路径、Agg 路径生成阶段, Geno 提供相应的钩子来注册、调用异构计算入口函数, 生成异构计算的路径及代价, 加入到对应的基表或连接表的候选路径列表中. Geno 根据多条路径的启动代价、总代价, 选择保存一条最优的查询路径. 如图 6(a)所示, *A* 表为行存表, *B* 表为压缩的列存表, 执行 SQL 语句 select *sum*(*col*2) from *a*, *b* where *a.col*1=*b.col*1.

Geno 在扫描路径生成阶段, 为 *A*, *B* 表分别生成本机 CPU 扫描路径 SeqScanPath、IndexScanPath, 以及 GPU 扫描路径 GpuScanPath、FPGA 扫描路径 FPGAScanPath 等候选路径, 并为每条路径计算启动代价和总代价,最后, 优化器根据各个候选路径的代价选择一条最优的扫描路径保存到对应基表的 *cheapest_total_path* 中. 在连接路径生成阶段, 优化器为 *AB* 连接表生成 CPU 连接路径 HashJoinPath、NestLoopJoinPath, GPU 连接路径 GPUJoinPath 和 FPGA 连接路径 FPGAJoinPath 等候选路径, 为每条连接路径计算启动代价和总代价, 选择一条代价最低的连接路径保存到该 *AB* 连接表的 *cheapest_total_path* 中. 在 Agg 路径生成节点生成本地 AggPath 路径、GPUAggPath 路径、FPGAAggPath 路径, 计算各路径的启动代价和总代价, 选择代价最低的路径保存到输出结果的 *cheapest_total_path* 中.

Geno 选择的最优查询路径如图 6(b)所示.

$$A(GPUScanPath)+B(FPGAScanPath)+AB(GPUJoinPath)+Agg(GPUAggPath).$$

Geno 根据以上生成的最优查询路径遍历整个查询路径, 依据每个路径节点类型生成对应的计划节点, 构建生成执行计划树, 最后调用执行引擎完成查询操作.



(a) 异构计算路径生成 (b) 最优路径

图 6 异构计算路径

2.3.3 动态统计信息及自动重优化

统计信息是计算路径代价的基石, 统计信息的准确度对代价估算模型中行数估算和代价估算至关重要, 直接影响查询优化器所确定的执行计划的优劣. 如果查询涉及的表缺失统计信息或者表上有多个谓词以及谓词包含有复杂操作, 以至于无法单独依赖于基表的统计信息, 将使得优化器不能准确估算基数. Geno 能够使用动态统计信息来进行补充, 动态统计信息允许查询优化器强化现有的统计信息以获取更加精确的基数估算, 不仅仅是为单表的访问, 而且也包含连接和分组谓词. Geno 利用动态统计信息进行自动重优化, 能够为那些反复执行的具有基数估算误差的查询改善查询计划.

Geno 利用一个 SQL 语句初次执行期间收集到的信息来进行自动重优化. 在一个 SQL 语句的初次执行结束时, 将它原来的基数估算和在执行期间观测到的实际基数进行比较, 如果估算值和实际值有显著差异, 它会将正确的值存储起来供后续使用. 当查询再次执行时, 优化器会使用纠正过的基数估算值, 而不是它原先的估算值来确定执行计划. Geno 在自动重优化阶段还会生成一些启发式规则, 使得其他 SQL 语句也能受益于这次初始执行中学到的信息. 例如, 当发生连接的两个表在连接列上有倾斜数据时, 这些规则可以指引优化器使用动态统计信息来获得更加精确的连接基数估算.

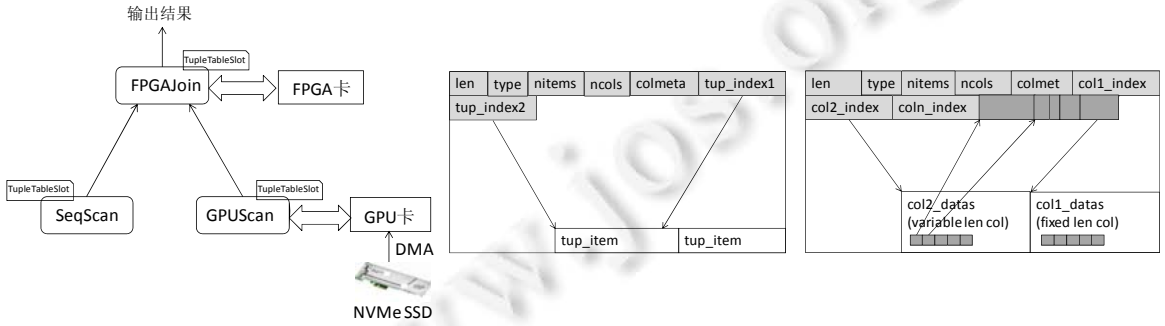
2.4 数据存储

异构计算环境下, 主机与异构硬件的数据传输和存储管理非常关键: 一方面, 要高效利用异构硬件内的内存资源来降低响应时延并提高吞吐量; 另一方面, 还要尽可能地减少 PCIe 上的数据迁移和传输.

图 7(a)展示了多个异构硬件计划节点间的数据传输. 每个计划节点在主机上有一个执行状态数据结构, 其中, TupleTableSlot 用来保存节点执行的输出数据, 同时也作为上层计划节点的输入数据. 执行节点从计划树下层节点 TupleTableSlot 获取数据, 批量拷贝到异构硬件算子的输入数据缓冲区执行算子, 并把结果数据拷贝回主机, 保存到执行计划节点的 TupleTableSlot 中.

行存储格式表按照行数据为基本存储单元进行存储, 一行中的所有列数据存储在一起. 行存储适合对表数据进行随机的增、删、改、查操作, 特别是需要频繁插入或更新的场景; 但是对于选择投影操作, 即使只涉及某几列, 所有数据也都会被读取. 列存储格式的表按照列数据为基本存储单元进行存储, 同一个列的数据存储在一起, 且数据类型一致, 可以针对列的数据类型、数据量选择不同的压缩算法, 对列数据进行压缩存储, 比行式存储更节省空间; 由于各列独立存储,查询过程中只读取涉及的列, 能够减少无关 I/O, 避免全表扫描; 列存储还可针对各个列的运算进行并发执行, 以降低查询响应时间. 通常, 行存储表适合于 OLTP 场景, 列存储适合于 OLAP 场景. Geno 优化器同时支持行存储格式和列存储格式数据在异构硬件上的加速, 能够满足多种业务场景, 具有更好的适应性. 针对行存储表和列存储表, 设计不同的主机与异构硬件之间数据传输缓冲区: 主机侧只负责加载磁盘原始数据, 不对数据进行处理; 异构硬件侧对原始数据进行解析过滤, 充分利用异构硬件并行计算优势.

行存储表的数据交换格式如图 7(b)所示, 缓冲区的头包含缓冲区长度 *len*、缓冲区类型 *type*、行记录数 *nitems*、字段个数 *ncols*、每个字段的类型信息 *colmeta*、每条原始记录在缓冲区的偏移量 *tup_index*. 记录 *tup_item* 从缓冲区尾部开始填充, 直至缓冲区满或全部数据加载完成. GPU 加载数据时, 以记录 *tup_item* 为单位并行处理, 每个 GPU core 一次读取一条记录进行处理, 根据缓冲区中各个字段的类型信息 *colmeta*, 解析出该条记录所有字段值, 执行选择、投影表达式, 完成对行存储表的扫描.



(a) 计划节点间数据交换 (b) 异构硬件数据交换行格式 (c) 异构硬件数据交换列格式

图 7 计划节点间数据交换和交换格式

列存储表数据交换格式如图 7(c)所示, 缓冲区头包含缓冲区长度 *len*、缓冲区类型 *type*、记录数 *nitems*、字段个数 *ncols*、每个字段的类型信息 *colmeta* 等元数据信息、每个字段批数据 *coln_datas* 在缓冲区中的偏移

量 *coln_index*. 对于定长数据类型字段, *coln_datas* 中直接存放相应的字段值, 对于变长数据类型字段, *coln_datas* 存放每个字段值的指针, 指向实际字段值存储的位置. 主机上只需要加载查询涉及的字段到缓冲区, GPU 加载数据时, 以字段为单位并行处理, 每个 GPU core 根据缓冲区中字段的类型信息 *colmeta* 及数据偏移, 一次读取一个字段值, 执行选择、投影表达式, 完成对该字段的扫描. 每个字段分配一组 GPU core 进行并行处理.

3 实验与分析

3.1 实验环境

实验环境配置见表 2.

表 2 实验环境配置

硬件名称	环境 *A* 规格参数	环境 *B* 规格参数
CPU	Intel(R) Xeon(R) Gold 5118@48 核 2.30 GHz	Intel(R) Xeon(R) E5-2630 v3@32 核 2.40 GHz
内存	125 GB	250 GB
硬盘	SSD MZ7LH480HAHQ-00005	SCSI DISK(MR9271-8i)
GPU	NVIDIA Tesla V100 32 GB	NVIDIA TITAN Xp 12 GB
FPGA	Xilinx Alveo U280	–

3.2 参数校准实验

3.2.1 计算相关代价参数校准实验

在环境 *A* 和环境 *B* 中运行参数校准工具, 得到校准后的代价参数见表 3. Geno 优化器根据校准后的代价参数来估算各种路径的启动代价和总代价, 并根据代价选择更优的执行计划.

表 3 校准参数

代价因子	Hetero-DB 固定值	环境 *A* 校准值 1	环境 *B* 校准值 2
seq_page_cost	1.000 00	1.000 00	1.000 00
random_page_cost	4.000 00	1.026 63	3.310 72
cpu_tuple_cost	0.010 00	0.011 16	0.002 15
cpu_index_tuple_cost	0.005 00	0.003 18	0.000 60
cpu_operator_cost	0.002 50	0.000 25	0.000 19
gpu_setup_cost	4 000.000 00	3 734	2 227
gpu_ratio	0.062 5	0.061 30	0.189 23
gpu_trans_cost	10.000 00	41	13

CPU 上, *random_page_cost* 参数的缺省值为 4, 环境 *A* 校准后的参数值为 1.026 63, 环境 *B* 校准后的参数值为 3.310 72. 由于 SSD 盘随机读性能接近于顺序读的性能, 校准后的参数值更精确反映了不同环境的 I/O 代价因子差异. 由于环境 *B* 的 CPU 性能强于环境 *A*, 环境 *B* 的 *cpu_tuple_cost*、*cpu_index_tuple_cost*、*cpu_operator_cost* 这几个 CPU 相关的因子校准值比环境 *A* 的小. 而环境 *A* 的 GPU 计算能力更强大, 所以环境 *A* 的 GPU 与 CPU 计算能力折算比例 *gpu_ratio* 比环境 *B* 的 *gpu_ratio* 要小. 主机与 GPU 之间数据传输代价相对于本地硬盘顺序读 8K 页面数据的代价比例来说, 由于环境 *B* 的硬盘性能相对较差, 因而环境 *B* 的 GPU 传输代价相对于磁盘顺序 I/O 代价比例小.

- 校准参数应用

下面我们对校准前后的环境参数进行验证对比. 在环境 *A* 中使用优化器默认值与校准参数 1, 在环境 *B* 中使用优化器默认值与校准参数 2, 分别运行最常用的等值查询和范围查询语句.

Q1:select *count*(*) from *t*0, *t*1 where *t*0.*id*=*t*1.*aid*;

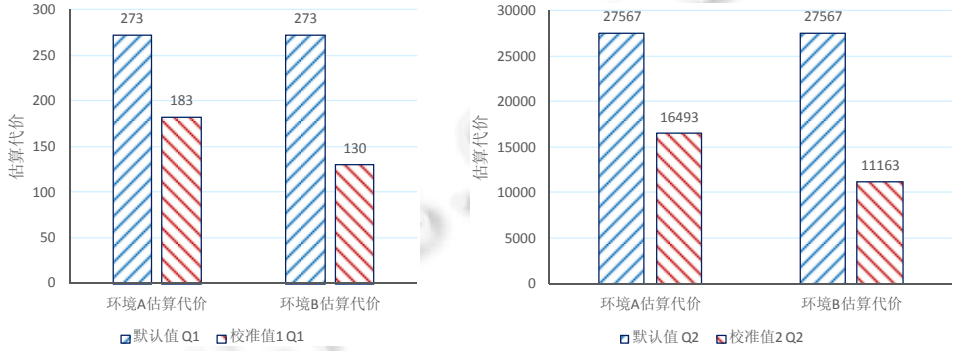
Q2:select *count*(*) from *t*1, *t*2 where *t1.aid*=*t*2.*bid* and *t*2.*bid*>700000.

测试结果如图 8 所示.

- 环境 *A* 上, Q1 查询校准前, *t*0 表采用顺序扫描, 然后与 *t*1 表进行 Hash 连接, 路径估算代价为 273; *t*0

表采用索引扫描, 然后与 $t1$ 表进行嵌套连接, 路径估算代价为 282. 因此, Geno 选择了代价相对较小的顺序扫描和 Hash 连接路径. 校准后, $t0$ 表采用顺序扫描, 然后与 $t1$ 表进行 Hash 连接, 路径估算代价校准为 211; $t0$ 表采用索引扫描并与 $t1$ 表进行嵌套连接, 路径估算代价为 183. 因此, Geno 选择了索引扫描和嵌套连接路径, 查询时间从 11.216 ms 减少到 6.934 ms;

- 环境 *B* 上, Q1 查询校准前后, Geno 都选择了顺序扫描和 Hash 连接路径, 因校准前后的代价参数不一样, 故路径估算代价不同, 但执行路径相同故实际执行时间相同.
- 环境 *A* 上, Q2 查询校准前, GpuJoin 路径估算代价为 27 567, MergeJoin 路径估算代价为 49 332, Geno 选择了 GpuJoin; 校准后, GpuJoin 路径估算代价为 19 421, MergeJoin 路径估算代价为 16 493, Geno 选择了 MergeJoin, 查询时间从 1 275.978 ms 减少到 931.862 ms;
- 而环境 *B* 上, Q2 查询校准后, Geno 也选择了路径估算代价最小的 MergeJoin 路径, 查询时间则从 1 114.571 ms 减少到 988.636 ms.



(a) 查询 1 参数校准实验 (b) 查询 2 参数校准实验

图 8 校准参数应用

通过以上实验分析, 可以得出校准后的参数值与硬件能力更匹配.

3.2.2 effective_cache_size 参数校准实验

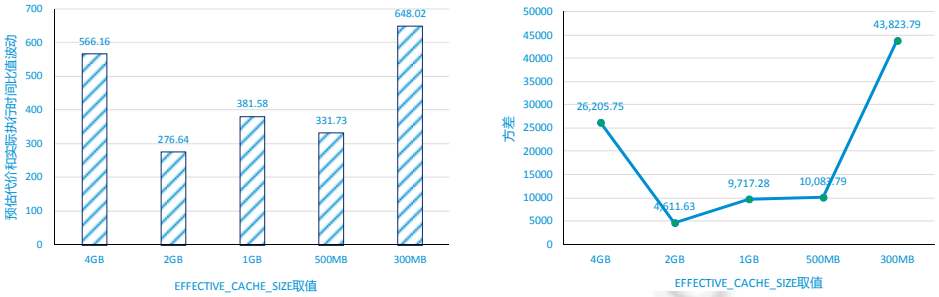
测试采用 Sysbench 工具, 通过调整 *effective_cache_size* 参数, 找出使得查询优化器估计值同实际值偏差最小的参数, 则该参数为最优参数. 由此检验实际测试的参数值是否同函数计算结果一致, 如果两者基本一致, 则函数有效.

验证过程: 设置共享缓冲区大小为 500 MB(根据实际情况调整), 同时, 数据库在做 TPCC 压力测试, 并发查询数为 60. *effective_cache_size* 取不同的值, 执行查询语句 select * from *test_b* where $a<N$. N 依次取值 500, 1 000, 3 000, 6 000, 10 000, 20 000, 40 000, 60 000, 80 000, 100 000, 记录估算出的总代价以及实际执行时间比值以及方差.

如图 9(a)所示, 相同数据条件下, *effective_cache_size* 参数取不同值: 4 GB, 2 GB, 1 GB, 500 MB, 300 MB, 分别测试上述 10 个 SQL, 得到 10 组预估代价和实际时间, 取两者的比值, 再从 10 组比值中取最大值和最小值的差值,可以获得预估代价和实际执行时间比值的波动值. 可以看出, 在各种场景中, 2 GB 情况的波动值是最小的.

如图 9(b)所示, 分别计算不同参数下比值的方差. 可以看出, *effective_cache_size* 参数取 2 GB 时, 优化器估计值与实际执行时间的比值方差最小. 索引扫描估算得到的代价与实际执行的时间之间的线性程度最好. 也就是说, 此时代价估算最准确. 将实时的并发数(60)和系统缓存大小(500 MB)代入公式(2), 计算得到的 *effective_cache_size* 值为 1.89 GB, 基本和实验得到的最佳参数值一致. 与系统默认值 4 GB 相比, 有效提升了索引代价估算的准确性.

对于连接查询, 其过程是依次获取外表的每一条记录, 在内表中通过索引扫描查找满足连接条件的记录. 也就是说, 会对内表进行多次串行的单表查询, 次数为外表记录条数. 而 *effective_cache_size* 实验中, 单表查询执行多次, 与连接查询过程类似, 所以对于连接查询, 和单表查询的效果是一样的.



(a) 预估代价和实际执行时间比值波动　　(b) 不同参数下比值的方差

图 9　*effective_cache_size* 参数校准实验

3.3 异构加速算子实验

Geno 支持了选择、连接、聚合、流水线等 GPU 算子和 FPGA 算子. 为验证其效果, 本节在环境 *A* 上使用校准参数配置值, 基于 TPC-H 基准测试模型的表和 300 GB 数据, 设计了 4 个典型 SQL 语句.

- 选择: SELECT *l_orderkey*, *l_partkey*, *l_suppkey*, *l_linenumber* FROM *lineitem*1 WHERE *l_linenumber*=1.
- 连接: SELECT *l_orderkey*, *s_name* FROM *supplier*1, *lineitem*1 WHERE *l_suppkey*=*s_suppkey* and *l_discount*=*l_tax**5.
- 聚合: SELECT *sum*(*l_extendedprice**0.8+*l_discount**0.2), *sum*(*l_discount**200+*l_tax**100), *sum*(*l_orderkey**0.8+*l_partkey**0.2) FROM *lineitem*1 where *l_orderkey*%100 between 94 and 100.
- 流水线: SELECT *sum*(*l_extendedprice**0.8+*l_discount**0.2), *sum*(*l_discount**200+*l_tax**100), *sum*(*l_orderkey**0.8+*l_partkey**0.2) FROM *lineitem*1, *supplier*1 WHERE *l_suppkey*=*s_suppkey* and (*l_orderkey*/3)%999=0 and (*l_linenumber*/6)%10000=0.

测试结果如图 10 所示, 与 CPU 相比, GPU 选择算子、连接算子、聚合算子和流水线算子的执行时长分别下降了 71%, 84%, 88%和 89%, FPGA 选择算子、连接算子、聚合算子和流水线算子的执行时长分别下降了 46%, 58%, 59%和 67%. 其中, 选择算子加速比例最小, 这是因为选择算子的执行时间中 I/O 占比高. 流水线算子加速最明显, 这是因为流水线算子包含聚合和连接等适合 GPU/FPGA 的计算密集型任务, 且多个算子融合减少了 CPU 与 GPU/FPGA 之间中间结果数据传输. FPGA 算子的执行时间是 GPU 的 2–3 倍, 这是因为本实验中 FPGA 的计算核心数和运行频率低于 GPU. 可见, Geno 提供的异构加速算子相对 CPU 算子, 处理性能显著提升, 但因为与 CPU 间存在数据传输瓶颈, GPU 和 FPGA 多核计算能力并未充分体现.

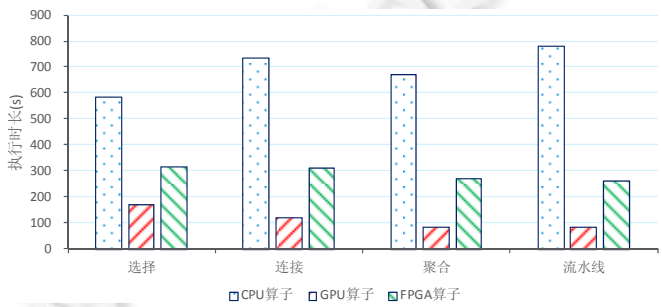


图 10　异构加速算子实验

3.4 TPC-H实验

基于 TPC-H 测试标准, 在环境 *A* 上使用校准参数 1 配置值, 预置行存表及列存表 300 GB 的测试数据, 对比分析 Postgresql、Hetero-DB 与 Geno 之间的性能差异.

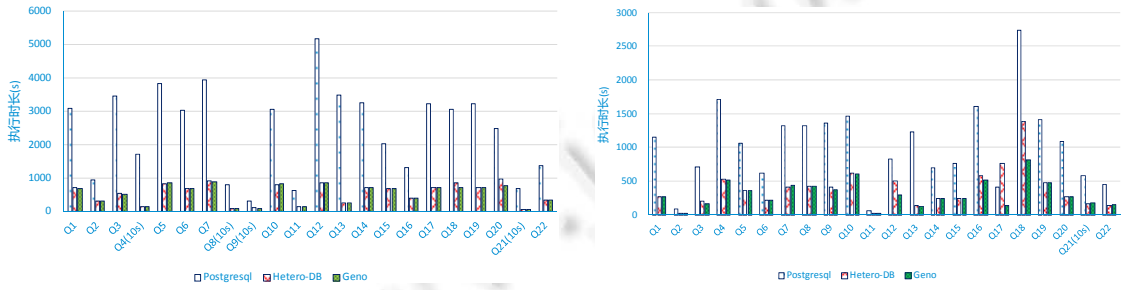
行存表实验结果如图 11(a)所示, 与 Postgresql 相比, Geno 执行时长减少了 64%–93%. 其中, Q3, Q4, Q8, Q13 减少 85%以上, 这是因为查询中包含了条件过滤、多表连接和分组聚合等适合 GPU 并行计算的操作, Geno 充分利用了 GPU 加速和多个 GPU 算子融合来提高查询执行效率; Q1, Q5–Q7, Q9–Q12, Q14, Q16–Q19, Q22 查询执行时长减少 75%以上, 这是因为查询中包含了单表查询或带有 From 子查询, 使用到了 GPU 加速; 而 Q2, Q15, Q20, Q21 执行时长只减少 65%左右, 这是因为 WHERE 条件中包含了复杂的子查询, 影响了 GPU 核利用率.

与 Hetero-DB 相比, 有 3 条查询上执行时长减少了 17%–39%, 其他查询执行时间相当. 其中, Q9 查询执行时间减少了 39%, 这是因为查询中对 lineitem 大表进行全表扫描, Geno 使用了 FPGA Scan, 由于 FPGA Scan 集成了解压和过滤功能, 特别适合大数据表的扫描; Q18, Q20 执行时长分别减少 17.8%和 19%, 这是因为查询中含有多表嵌套子查询和分组、排序查询, 并且过滤条件和分组、排序条件涉及的字段有索引, Geno 生成的计划扫描表时, 利用 CPU 进行索引字段扫描和过滤, 而 Hetero 强制使用了 GPU 对所有连接表进行全表扫描; 而 Q1–Q8, Q10–Q17, Q19, Q21, Q22 查询等多表扫描、连接操作时, 二者都选择了更优的 GPU 执行路径, 执行时间相当.

列存表实验结果如图 11(b)所示, Geno 与 Postgresql 相比, 执行时间减少 87%–92%, 这是因为在 TPC-H 查询语句中包含大量表扫描、条件过滤、多表连接和分组聚合等适合 GPU、FPGA 加速的操作.

Geno 与 Hetero-DB 相比, 有 5 条查询执行时间减少 9%–81%, 其他查询执行时间相当. 其中, Q17 查询对两个表进行连接查询, 执行时间减少 81%, 这是因为 Geno 选择了结果集比较小的表作为内表进行 Hash 连接; Q12 查询执行时间减少 38%, 这是因为 Geno 选择使用 FPGA 流水线执行查询 Scan, Join, Agg 和 Sort, 消除了 SQL 节点间不必要的数据交换; Q3, Q18 查询是多表连接, 执行时间分别减少 17%和 41%, 这是因为 Geno 先进行两小表连接, 再与大表进行连接; Q16 查询执行时间减少 9%, 这是因为 Q16 查询在 NOT IN 子句中包含子查询, 外表数据量相对较小, Hetero-DB 对外表扫描选择了 GpuScan, 而 Geno 选择了 CPU Scan; Q1, Q2, Q4–Q7, Q11, Q14, Q15, Q19–Q22 查询执行时间基本相近, 因为二者都选择了相似的 GPU 执行路径.

Geno 列存与行存相比查询执行时间减少 32%–89%, 其中, Q2, Q11, Q6 分别减少 89%, 79%, 67%, 这是因为 Q2, Q11, Q6 这 3 个查询中只涉及表的部分列字段, 查询列式表时只读取涉及的列, 消除了无关列的 I/O 开销及其列值解析开销, 减少了主机与 GPU 之间数据传输代价; 同时, GPU 内存能够容纳更多的有效数据, 提高了 GPU 的并行处理效率.



(a) 行存表测试结果　　(b) 列存表测试结果

图 11　TPC-H 实验

上述实验结果表明: Geno 优化器通过新型硬件能力校准, 更准确地评估异构计算路径的代价, 从而生成与硬件能力匹配的异构计算执行计划, 充分利用异构硬件计算能力有效提升系统性能.

4 总结与展望

高性能处理器和专用加速器、新型非易失存储器、高速互联设备，以及由它们催生的具有丰富硬件上下文的异构计算架构、以高速持久化为显著特征的混合存储环境和支持远程直接数据存取的高速互联结构正在快速改变传统数据库系统的底层载体支撑. 这些新型硬件及其构建的环境改变了传统的计算、存储以及网络体系, 也改变数据库系统既往的设计假设. 数据库系统需要针对新型硬件特性做出适配和调整, 以充分发挥新硬件的潜力, 同时避免新硬件自身约束导致的新瓶颈. 本文提出一种支持新型硬件异构融合的基于代价的查询优化器 Geno, 通过异构计算资源的代价校准, 建立异构计算环境下查询处理的代价模型; 在 GPU/FPGA 上实现了关系型数据处理的多种算子, 挖掘异构加速器强大的计算能力; 通过基于代价的评估解决算子分配问题并生成异构协同执行计划, 实现异构计算资源的协同优化. 本文工作应用在中兴新一代数据库系统, 配合中兴"通用处理器+专用加速器"算力方案, 提供数据库加速场景解决方案. 但是, 目前新型处理器和专用加速器发展的多样性, 使得新型硬件环境的构成复杂多变, 基于代价估计的 Geno 在实践中很难做到高精确度和灵活性的兼顾. 下一步, 我们将在基于代价估计的异构融合查询优化器的基础上引入人工智能技术, 综合异构计算环境特征、查询计算特征以及数据分布特征, 探索研究 Geno 和基于学习的查询优化技术结合, 形成动态的模型, 自适应地根据实际场景进行优化.

References:

[1] Pei W, Li ZH, Pan W. Survey of key technologies in GPU database system. Ruan Jian Xue Bao/Journal of Software, 2021, 32(3): 859−885 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6175.htm [doi: 10.13328/j.cnki.jos.006175]

[2] Papaphilippou P, Luk W. Accelerating database systems using FPGAs: A survey. In: Proc. of the 28th Int'l Conf. on Field Programmable Logic and Applications (FPL). 2018. 1312−1317.

[3] Bordawekar RR, Sadoghi M. Accelerating database workloads by software-hardware-system co-design. In: Proc. of the 2016 IEEE 32nd Int'l Conf. on Data Engineering (ICDE). IEEE, 2016. 1428−1431.

[4] Ailamaki A. Databases and hardware: The beginning and sequel of a beautiful friendship. Proc. of the VLDB Endowment, 2015, 8(12): 2058−2061.

[5] Yu XY, Bezerra G, Pavlo A, Devadas S, Stonebraker M. Staring into the abyss: An evaluation of concurrency control with one thousand cores. Proc. of the VLDB Endowment, 2014, 8(3): 209−220.

[6] Ibaraki T, Kameda T. On the optimal nesting order for computing *N*-relational joins. ACM Trans. on Database Systems, 1984, 9(3): 482−502.

[7] Bausch D, Petrov I, Buchmann A. Making cost-based query optimization asymmetry-aware. In: Proc. of the 8th Int'l Workshop on Data Management on New Hardware. ACM, 2012. 24−32.

[8] Balkesen C, Teubner J, Alonso G, Tamerözsu M. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In: Proc. of the 2013 IEEE 29th Int'l Conf. on Data Engineering (ICDE). IEEE, 2013. 362−373.

[9] He J, Zhang SH, He BS. In-cache query co-processing on coupled CPU-GPU architectures. Proc. of the VLDB Endowment, 2014, 8(4): 329−340.

[10] Cheng XT, He BS, Lu M, Lau CT, Huynh HP, Goh RSM. Efficient query processing on many-core architectures: A case study with Intel Xeon Phi processor. In: Proc. of the 2016 Int'l Conf. on Management of Data. ACM, 2016. 2081−2084.

[11] Werner S, Groppe S, Linnemann V, Pionteck T. Hardware-accelerated join processing in large semantic Web databases with FPGAs. In: Proc. of the Int'l Conf. on High Performance Computing & Simulation. IEEE, 2013. 131−138.

[12] Van Aken D, Pavlo A, Gordon GJ, Zhang BH. Automatic database management system tuning through large-scale machine learning. In: Proc. of the SIGMOD 2017. ACM, 2017. 1009−1024.

[13] Heimel M, Kiefer M, Markl V. Self-tuning, GPU-accelerated kernel density models for multidimensional selectivity estimation. In: Proc. of the 2015 SIGMOD Int'l Conf. on Management of Data. Melbourne: ACM, 2015. 1477−1492.

[14] Mohsin SA, Darwish SM, Younes A. QIACO: A quantum ant system for query optimization in relational database. IEEE Access, 2021, 9: 15833−15846.

[15] Paul J, He J, He BS. A GPU-based pipelined query processing engine. In: Proc. of the 2016 Int'l Conf. on Management of Data (SIGMOD 2016). New York: ACM, 2016. 1935−1950.

[16] Breß S. Why it is time for a HyPE: A hybrid query processing engine for efficient GPU coprocessing in DBMS. In: Proc. of the 39th Int'l Conf. on Very Large Data Bases. ACM, 2013. 1398−1403.

[17] Breß S, Siegmund N, Heimel M, Saecker M, Lauer T, Bellatreche L, Saake G. Load-aware inter-co-processor parallelism in database query processing. In: Proc. of the Data & Knowledge Engineering. ACM, 2014. 60−79.

[18] Zhang K, Chen F, Ding X, Huai Y, Lee RB, Luo T, Wang KB, Yuan Y, Zhang XD. Hetero-DB: Next generation high-performance database systems by best utilizing heterogeneous computing and storage resources. Journal of Computer Science and Technology, 2015, 30(4): 657−678. [doi: 10.1007/s11390-015-1553-y]

[19] Owaida M, Sidler D, Kara K, Alonso G. Centaur: A framework for hybrid CPU-FPGA databases. In: Proc. of the 2017 IEEE 25th Annual Int'l Symp. on Field-programmable Custom Computing Machines (FCCM). IEEE, 2017. 211−218.

[20] Marcus R, Negi P, Mao HZ, Zhang C, Alizadeh M, Kraska T, Papaemmanouil O, Tatbul N. Neo: A learned query optimizer. Proc. of the VLDB Endowment, 2019, 12(11): 1705−1718.

[21] Sun J, Li GL. An end-to-end learning-based cost estimator. Proc. of the VLDB Endowment, 2019, 13(3): 307−319.

[22] Shanbhag A, Madden S, Yu XY. A study of the fundamental performance characteristics of GPUs and CPUs for database analytics (extended version). In: Proc. of the Int'l Conf. on Management of Data (SIGMOD 2020). ACM, 2020. 1617−1632.

[23] Gregg C, Hazelwood K. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In: Proc. of the IEEE Int'l Symp. on Performance Analysis of Systems & Software. IEEE, 2011. 134−144.

[24] Furst E, Oskin M, Howe B. Profiling a GPU database implementation: A holistic view of GPU resource utilization on TPC-H queries. In: Proc. of the 13th Int'l Workshop on Data Management on New Hardware. Chicago: ACM, 2017. 1−6.

[25] Mackert LF, Lohman GM. Index scans using a finite LRU buffer: A validated I/O model. ACM Trans. on Database Systems, 1989, 14(3): 401−424.

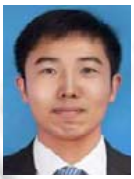
附中文参考文献:

[1] 裴威, 李战怀, 潘巍. GPU 数据库核心技术综述. 软件学报, 2021, 32(3): 859−885. http://www.jos.org.cn/1000-9825/6175.htm [doi: 10.13328/j.cnki.jos.006175]

屠要峰(1972－), 男, 博士生, 研究员, CCF 高级会员, 主要研究领域为大数据, 数据库, 机器学习, 云计算.

卞福升(1971－), 男, 学士, 主要研究领域为数据库.

陈小强(1975－), 男, 学士, CCF 专业会员, 主要研究领域为数据库, 异构计算.



吴非(1991－), 男, 硕士, 主要研究领域为数据库, 异构计算.

周士俊(1979－), 男, 学士, 主要研究领域为数据库, 云计算.

陈兵(1970－), 男, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为大数据, 云计算, 认知无线电网络.