软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(11):5205−5229 [doi: 10.13328/j.cnki.jos.006739]

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

GPU 数据库 OLAP 优化技术研究*

张延松[1,2], 刘 专[1,2], 韩瑞琛[1,2], 张 宇[3], 王 珊[1,2]

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

[2](中国人民大学 信息学院, 北京 100872)

[3](国家卫星气象中心, 北京 100081)

通信作者: 张宇, E-mail: yuzhang@cma.gov.cn

摘 要: GPU 数据库近年来在学术界和工业界吸引了大量的关注. 尽管一些原型系统和商业系统 (包括开源系统) 开发了作为下一代的数据库系统, 但基于 GPU 的 OLAP 引擎性能是否真的超过 CPU 系统仍然存有疑问, 如果能够超越, 那什么样的负载/数据/查询处理模型更加适合, 则需要更深入的研究. 基于 GPU 的 OLAP 引擎有两个主要的技术路线: GPU 内存处理模式和 GPU 加速模式. 前者将所有的数据集存储在 GPU 显存来充分利用 GPU 的计算性能和高带宽内存性能, 不足之处在于 GPU 容量有限的显存制约了数据集大小以及稀疏访问模式的数据存储降低 GPU 显存的存储效率. 后者只在 GPU 显存中存储部分数据集并通过 GPU 加速计算密集型负载来支持大数据集, 主要的挑战在于如何为 GPU 显存选择优化的数据分布和负载分布模型来最小化 PCIe 传输代价和最大化 GPU 计算效率. 致力于将两种技术路线集成到 OLAP 加速引擎中, 研究一个定制化的混合 CPU-GPU 平台上的 OLAP 框架 OLAP Accelerator, 设计 CPU 内存计算、GPU 内存计算和 GPU 加速 3 种 OLAP 计算模型, 实现 GPU 平台向量化查询处理技术, 优化显存利用率和查询性能, 探索 GPU 数据库的不同的技术路线和性能特征. 实验结果显示 GPU 内存向量化查询处理模型在性能和内存利用率两方面获得最佳性能, 与 OmniSciDB 和 Hyper 数据库相比性能达到 3.1 和 4.2 倍加速. 基于分区的 GPU 加速模式仅加速了连接负载来平衡 CPU 和 GPU 端的负载, 能够比 GPU 内存模式支持更大的数据集.

关键词: 混合 CPU-GPU 平台; GPU 加速 OLAP; OLAP GPU 内存; GPU 量化处理模型

中图法分类号: TP311

中文引用格式: 张延松, 刘专, 韩瑞琛, 张宇, 王珊. GPU数据库OLAP优化技术研究. 软件学报, 2023, 34(11): 5205–5229. http://www.jos.org.cn/1000-9825/6739.htm

英文引用格式: Zhang YS, Liu Z, Han RC, Zhang Y, Wang S. OLAP Optimization Techniques Based on GPU Database. Ruan Jian Xue Bao/Journal of Software, 2023, 34(11): 5205–5229 (in Chinese). http://www.jos.org.cn/1000-9825/6739.htm

OLAP Optimization Techniques Based on GPU Database

ZHANG Yan-Song[1,2], LIU Zhuan[1,2], HAN Rui-Chen[1,2], ZHANG Yu[3], WANG Shan[1,2]

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

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

[3](National Satellite Meteorological Center, Beijing 100081, China)

Abstract: Graphics processing unit (GPU) databases have attracted a lot of attention from the academic and industrial communities in recent years. Although quite a few prototype systems and commercial systems (including open-source systems) have been developed as next-generation database systems, whether GPU-based online analytical processing (OLAP) engines really outperform central processing unit (CPU)-based systems is still in doubt. If they do, more in-depth research should be conducted on what kind of workload/data/query

* 基金项目: 国家自然科学基金 (61772533, 61732014); 北京市自然科学基金 (4192066)
收稿时间: 2022-03-15; 修改时间: 2022-04-22, 2022-07-01; 采用时间: 2022-07-17; jos 在线出版时间: 2023-06-16
CNKI 网络首发时间: 2023-06-19

processing models are more appropriate. GPU-based OLAP engines have two major technical roadmaps: GPU in-memory processing mode and GPU-accelerated mode. The former stores all the datasets in the GPU device memory to take the best advantage of GPU's computing power and high bandwidth memory. Its drawbacks are that the limited capacity of the GPU device memory restricts the dataset size and that memory-resident data in the sparse access mode reduces the storage efficiency of the GPU display memory. The latter only stores some datasets in the GPU device memory and accelerates computation-intensive workloads by GPU to support large datasets. The key challenges are how to choose the optimal data distribution and workload distribution models for the GPU device memory to minimize peripheral component interconnect express (PCIe) transfer overhead and maximize GPU's computation efficiency. This study focuses on how to integrate these two technical roadmaps into the accelerated OLAP engine and proposes OLAP Accelerator as a customized OLAP framework for hybrid CPU-GPU platforms. In addition, this study designs three calculation models, namely, the CPU in-memory calculation model, the GPU in-memory calculation model, and the GPU-accelerated model for OLAP, and proposes a vectorized query processing technique for the GPU platform to optimize device memory utilization and query performance. Furthermore, the different technical roadmaps of GPU databases and corresponding performance characteristics are explored. The experimental results show that the vectorized query processing model based on GPU in-memory achieves the best performance and memory efficiency. The performance is 3.1 and 4.2 times faster than that achieved with the datasets OmniSciDB and Hyper, respectively. The partition-based GPU-accelerated mode only accelerates the join workloads to balance the workloads between the CPU and GPU ends and can support larger datasets than those the GPU in-memory mode can support.

Key words: hybrid CPU-GPU platform; GPU accelerated OLAP; GPU in-memory OLAP; GPU vectorized processing model

近年来随着基于 GPU 平台的主流高性能计算平台的兴起, GPU 数据库成为下一代高性能数据库的代表性技术之一. GPU 数据库的性能优势主要由大量 GPU 并行计算核心、高带宽内存 (HBM) 和先进的互联技术决定 (如 NVIDIA 最新的 A100 GPU 集成了 6912 个 cuda 核心, 80 GB 显存带宽超过 2 TB/s, GPU-GPU 之间的 NVLink3 通道带宽达到 600 GB/s[1]). 传统的 GPU 数据库研究受较小的 GPU 显存容量和较低的 PCIe 通道性能限制, 导致 CPU 与 GPU 之间基于数据传输的加速模式性能较低, 不能充分发挥 GPU 的性能优势, 因此需要设计新的技术路线更有效地提升 GPU 数据库性能.

GPU 加速型数据库将 GPU 看作是一个协处理器, 每个查询通过低带宽的 PCIe 通道将数据从 CPU 传输到 GPU 进行计算, GPU 显存用作 CPU 内存的缓存使用[2,3]. GPU 加速型数据库的工作原理与磁盘数据库类似, CPU 内存类似磁盘, GPU 显存类似缓冲区, PCIe 通道类似 I/O 通道, 基本假设是缓冲区 (GPU 显存) 不足以存储查询处理的全部数据, 需要通过磁盘 (CPU 内存) 和缓冲区 (GPU 显存) 之间的数据交换完成查询处理, 主要性能瓶颈是 I/O (PCIe).

GPU 内存数据库以 GPU 显存足够存储查询热数据集为基础假设 (OmniSciDB[4]), GPU 内存数据库可以看作是 GPU 端的内存数据库, GPU 不断增长的显存容量和多卡配置使显存能够存储查询的数据子集, 为 GPU 内存数据库提供了硬件支持[5].

当前基于 GPU 的数据库已有大量的研究和一系列的系统[6−10], 但以下几方面仍然存在着不足.

● 性能评估: GPU 数据库的性能研究需要通过更有代表性的基准测试评估当前最优的 GPU 数据库和内存数据库系统, 如 OmniSciDB (CPU 和 GPU 版本)[4]、Hyper[11]、Actian Vector [12]、MonetDB[13]、PG-Strom+Apache Arrow [14]等系统, GPU 数据库是重要的数据库性能加速技术, 但相对于最优的内存数据库还没有取得绝对的性能优势.

● 优化技术: 很多 GPU 数据库声明不使用索引 (“An End to Indexing”, OmniSci Core SQL), 主要原因是避免索引的存储和更新开销, 但连接索引[15]和代理键索引[16]可以在低存储和管理代价的前提下, 提供更高的性能, 而且 GPU 数据库并不是简单的内存数据库算法 GPU 化, 需要根据 GPU 的硬件特性, 设计更适合 GPU 的优化算法, 有效地发挥 GPU 的计算性能.

● GPU 处理模型: GPU 数据库的性能受很多因素的影响, 如果数据集适合 GPU 显存大小则需要 GPU 端内存计算模型, 否则需要设计 GPU 加速处理模型, 如何优化对数据集在 CPU 和 GPU 端分布, 如何优化 CPU 和 GPU 端负载来最小化 PCIe 传输代价, 什么样的模式/数据模型/查询处理模型更加适合 GPU 数据库, 这些问题仍然没有明确的答案.

对于联机分析处理 (online analytical processing, OLAP) 负载来说, 在数据量和性能之间的一个权衡策略是基于负载分区的 GPU 加速 OLAP 计算模型[17], 通过 GPU 加速数据库中数据量较小的计算密集型负载, 使用 CPU 处理高选择性但计算代价较低的数据密集型负载. 这种 OLAP 加速模型的目标是提高 CPU 和 GPU 端的计算局部性. Fusion OLAP 模型[18]是一种多维-关系融合的 OLAP 查询处理模型, 将多维索引计算定义为计算密集型负载, 可以通过 GPU、Phi 以及 FPGA 等硬件加速器提高其有限容量本地存储数据集上的计算性能, 同时将 CPU 端存储的大容量事实数据上的分组聚集计算定义为数据密集型负载, 通过向量索引提高内存数据访问性能, 通过分组向量提高分组聚集计算性能, 提高分组聚集计算负载的内存带宽利用率. CPU 与 GPU 端的数据交互限定在较小的维向量和压缩的向量索引, 能够最小化 CPU 与 GPU 之间 PCIe 通道传输的数据量, 降低 PCIe 瓶颈效应. 这种面向 GPU 定制化的计算模型的基础假设是 GPU 高带宽显存容量不足以容纳计算密度不同的全部数据集, 并且, 具有数据密集型特征的基于向量索引的分组聚集计算的内存带宽吞吐性能高于 PCIe 带宽性能.

本文设计了 OLAP Accelerator 计算框架, 研究如何通过索引、算法优化、查询处理模型优化、GPU 内存利用率、中间结果物化策略等技术探索 GPU 数据库的最优性能, 系统同时支持 GPU 内存计算模型和 GPU 加速处理模型, 以探索面向不同性能和应用场景下不同的 GPU 数据库技术路线. OLAP Accelerator 计算框架基于 3 个维度来设计: 在 OLAP 查询处理维度使用 Fusion OLAP 模型而不是传统的 ROLAP (关系 OLAP) 和 MOLAP (多维 OLAP) 模型, 把传统基于关系模型和关系操作的 OLAP 查询处理任务转换为多维计算; 从索引维度来看, 通过代理键索引在关系存储的数据上构建虚拟 n 维数据空间, 使用向量连接和向量分组聚集操作替代传统的哈希连接和哈希分组聚集操作, 优化 OLAP 算子的性能; 在查询处理模型维度, 设计了面向 CPU 和 GPU 平台的行式、列式和向量化查询处理模型, 尤其是面向 CPU 和 GPU 不同 cache 结构的向量化查询处理模型优化 CPU 和 GPU 平台的算法性能.

OLAP Accelerator 在现阶段支持星形模型上的 OLAP 查询处理, 支持星形模式 (star schema benchmark, SSB) 基准上的全部查询任务, 将来会扩展到多事实表结构的雪花形模型负载. OLAP Accelerator 支持 CPU 内存计算、GPU 加速和 GPU 内存计算模式的内存 OLAP 查询处理模型, 设计了基于向量索引的连接、星形连接和分组聚集算法替代传统数据库的算子实现. 从性能的角度而言, 向量化查询处理模型性能较优; 从 GPU 显存利用率的角度来看, 行式和向量化查询处理都有效地提高 GPU 显存利用率. 考虑到 CPU 与 GPU 硬件结构上的显著差异, 本文设计了面向 CPU 和 GPU 的向量化查询处理算法, 支持定长和压缩的向量索引技术. 本文的贡献主要体现在以下 3 个方面.

- 设计了面向 CPU-GPU 混合平台的 3 层 OLAP 计算模型, 按 OLAP 计算中不同的负载特性将 OLAP 负载分为 3 类: 管理密集型负载是维表上的多维元数据管理, 由 CPU 处理; 计算密集型负载是外键列上的多维计算, 可以由 GPU 加速; 数据密集型负载是 CPU 或 GPU 存储事实数据上的分组聚集计算. OLAP Accelerator 计算框架提供了灵活的 GPU 数据库实现技术路线, 根据数据集大小、GPU 显存大小为不同负载的数据集选择优化的存储和计算平台, 负载的划分由模式决定, 也就是说数据库模式层的优化技术可以决定 CPU-GPU 混合平台的数据分布与计算策略, 确立了模式决定计算 (schema defined computing) 的新型 OLAP 优化框架.
- 提出了基于向量索引的行式和向量化 OLAP 查询处理模型, 不同于大多数 GPU 数据库采用的一次一操作符 (operator-at-a-time) 计算模型, 本文通过利用 GPU 的共享内存 (shared memory) 优化算子间中间结果的物化访问和计算代价, 提高查询性能和 GPU 显存利用率. 通过不同 OLAP 模型、算法、查询处理模型等不同技术路线的系统之间的对比, 给出了当前最优内存数据库、GPU 数据库、GPU OLAP 实现技术在 SSB 基准测试中的性能, 为 GPU 数据库技术路线的选择提供了实验依据.
- 提出了基于数据/负载特征的 CPU-GPU 存储和计算策略, 通过全面的性能对比测试, 揭示了最优的内存 OLAP 实现技术与 GPU 数据库技术相比仍然有较好的竞争力, GPU 内存 OLAP 计算模型具有最优的性能, 但是数据处理容量受 GPU 显存容量的限制, GPU 加速模式牺牲了一定的性能但能够支持更大的数据集, 可以根据实际业务的数据和硬件条件进行负载划分, 选择合适的模型进行处理.

GPU 数据库的性能加速受多种因素的影响[19], 单一的 GPU 数据库技术难以满足不同场景下的高性能 OLAP

查询处理需求, 本文探索基于向量索引技术的统一 OLAP 计算框架提供面向异构计算平台的定制化优化技术, 以满足不同硬件配置、数据规模、性能需求下的 OLAP 查询处理实现技术.

1 背 景

1.1 CPU 与 GPU 硬件架构及相关查询优化技术

表 1 显示了当前 Intel 和 NVIDIA 最有代表性的 CPU 和 GPU 硬件配置, 多核 CPU 核心配置私有的寄存器、L1-L2 cache 和共享的 L3 cache, 在多线程查询处理时数据被逻辑或物理划分为多个分片, 每个线程处理一个数据分片, 通过私有 L1-L2 cache 优化线程私有数据集和中间结果, 通过 L3 cache 访问线程间共享数据集和中间结果. 当前内存数据库的优化技术以 CPU 的硬件架构为基础, 如 Hyper 通过 LLVM 实时编译[20]查询任务, 生成高效的机器码和面向寄存器优化的推模式查询执行计划, 提升代码执行性能. Vector[12]采用向量化查询处理模型将数据划分为适合 L1 cache 存储的向量数据分片 (向量长度通常为 1 024), 通过列向量执行提高代码性能和中间结果在 L1 cache 中的物化访问性能, 降低中间结果的内存物化和访问延迟, 提高查询整体性能. MonetDB 采用单指令多数据 (single instruction multiple data, SIMD) 优化技术和列式查询处理模型提高 CPU 计算效率和 cache 性能, 但内存物化策略产生较大的内存访问延迟. CPU 较少的核心数量和较大的私有 cache 结构简化了并行查询处理模型, 在向量化查询处理模型中可以支持较大的向量长度.

表 1 代表性 CPU 和 GPU 的硬件配置

处理器	核心	线程	Cache	带宽
Intel Xeon Platinum 8368	38@2.4 GHz	76	32 KB L1, 1.25 MB L2, 57 MB L3	–200 GB/s
NVIDIA A100 SMX	6912@1.41 GHz	—	192 KB L1/SMEM 40 MB L2	80 GB HBM @ 2 TB/s NVLink 600 GB/s PCIe Gen4 64 GB/s

GPU 由几十到上百个流处理器 (streaming multiprocessor, SM) 组成, 每个 SM 内部由一定数量的 cuda 核心组成. 每个 SM 共享统一的 L1 cache 和共享内存, 所有的 SM 共享 L2 cache. 与 CPU 相比, 每个 cuda 核心的私有共享内存远小于 CPU 的 L1-L2 cache. GPU 的线程组织为线程块 (thread block) 在 SM 上运行, 线程块进一步划分为 warp (通常 32 线程) 执行单指令多线程 (single instruction multiple threads, SIMT) 模式的处理, 每一线程在不同的数据上执行相同的操作. CPU 线程访问连续的数据通过私有 cache 缓存中间结果, 而 GPU 的 SM 内部的线程访问连续的数据通过共享内存缓存中间结果, 形成资源争用.

文献 [5] 指出, 内存带宽性能决定了很多关系操作性能的上限, GPU 配置的 HBM2 显存的带宽性能为数据库性能提供了强大的硬件支持, GPU 内存处理模式在性能上限上远高于 CPU 内存处理模式和 GPU 加速模式, 其优化技术的核心思想是, 如何更好地利用 GPU cache 和共享内存优化查询处理中各算子的算法实现和中间结果的物化访问, 提高查询性能, 提高 GPU 显存利用率.

1.2 OLAP 相关技术

GPU 数据库主要用于 OLAP 分析处理任务, 通过 GPU 的计算能力提高 OLAP 查询处理性能. 当前 OLAP 引擎主要使用关系数据库引擎, 研究中主要的内容是如何实现 GPU 端的关系数据库和优化 GPU 端的关系操作性能. 但 GPU 与 CPU 在硬件架构上有较为显著的差异, 需要扩展传统关系操作的框架, 设计更加适合 GPU 计算特性的算法和处理模型, 以更好地适应现代 CPU-GPU 混合架构上的 OLAP 计算, 更好地发挥 CPU 与 GPU 各自不同的性能优势. 本节从 OLAP 模型、数据库设计和算子算法设计 3 个方面讨论 OLAP 实现技术.

● OLAP 模型

当前主要 GPU 数据库使用的是 ROLAP 模型, 通过 GPU 计算技术提高关系数据库的分析处理性能, 研究目标主要限定在如何通过 GPU-conscious 的设计提高 GPU 端关系操作的性能. Fusion OLAP 模型[18]使用 MOLAP

模型构建逻辑的数据立方体, 使用多维 OLAP 操作, 与传统 MOLAP 模型的区别在于, Fusion OLAP 模型直接在存储效率更高的关系存储模型上执行多维计算, 消除传统 MOLAP 模型数据立方体预聚集和物化代价. 多维计算模型采用直接多维地址映射技术, 相对于传统关系操作复杂的算子更为简单高效, 但依赖于随机访问的性能, 不适合传统磁盘数据库平台, 更适合于现代的内存计算或 GPU 计算平台. Fusion OLAP 模型通过将 OLAP 查询处理过程划分为数据独立、计算独立的维向量映射、多维过滤和聚集计算 3 个阶段, 能够更灵活地支持在 CPU-GPU 平台的负载优化匹配, 也适合未来开放的异构计算平台上的应用.

当前 GPU 数据库在 OLAP 查询处理模型上研究较多, 总体可以看作是关系型 GPU 数据库在 OLAP 领域的优化技术, 如 OmniSciDB、SQream、Brytlyt、PG-Strom、RateupDB[21]等, 主要通过优化 GPU 平台上的关系数据库算子算法优化技术来提高数据库的 OLAP 性能, 在算法优化技术上更多是进行面向 GPU 硬件特性的优化技术, 对传统的哈希连接、排序连接、哈希分组和排序分组等算法进行 GPU 化的改造和优化.

● 数据库设计

数据库的核心功能是管理数据模型、元数据、存储模型和查询处理模型. 一个典型的 OLAP 查询任务可以看作是在元数据上的管理和在多维数据上的计算任务, 复杂的管理功能适合 CPU 端处理, 而大数据量上的多维计算任务更有利于发挥 GPU 的计算能力. 理想的情况下, 应该从传统的查询处理负载中将管理型负载和计算型负载进行分离, 并且将负载的特性与 CPU 和 GPU 的硬件特性进行优化匹配, 而不是用 CPU 或 GPU 平台全部执行这些特征不同的负载.

在 OLAP 数据集中, 维表可以看作是多维数据模型中的元数据, 它定义了维、层次、过滤器等 OLAP 操作的逻辑多维结构数据, 对应 SQL 命令中的 GROUP BY 和 WHERE 等子句, 用于 OLAP 查询中的分片、分块和分组等操作, 数据类型通常为较长的字符型、日期型等非数值型数据用于描述性属性的存储. 维表通常数量较多但数据量较小, 而且增长缓慢, 在较小维表上数据库的并发控制和更新机制效率较高而且只需要较低的并行处理能力. 事实表由外键列和度量列组成, 通常为简单的整型、浮点型数据, 数据宽度较小, 主要用于聚集计算任务. 事实表上的计算可以进一步分为两类: 计算密集型负载和数据密集型负载. 外键列的数量由模式决定, 数据量中等大小, 在 OLAP 查询中执行高强度的多表连接操作, 也是 OLAP 查询中主要的代价. 外键列上的连接操作在本文中被定义为计算密集型负载, 即在较小的数据集上较大计算强度的负载. 与之相对, 度量数据较大, 但在优化的 OLAP 查询处理模型中只执行基于向量索引的稀疏 (低选择率) 的数据访问和轻量的聚集计算任务, 主要计算类型是在庞大的数据集中执行稀疏的按位置访问操作, 对带宽性能的依赖度高于计算性能. 本文将事实表度量数据上的计算任务定义为数据密集型负载.

通过负载的划分, 数据/负载的分布策略可以根据负载的特性进行优化设计. 当计算密集型和数据密集型负载都分配给 GPU 时, GPU 的众核并行计算能力和 HBM 数据访问性能可以提供最优的性能并且最小化 PCIe 通道性能瓶颈的影响. 当 GPU 显存容量不足以容纳两类负载时, 传统 GPU 数据库主要采用数据分区、PCIe 传输优化、GPU 端缓存优化等技术实现 CPU-GPU 协同处理, 而本文采用按模式和负载计算强度特性划分数据集和负载的方法, 将计算密集型负载外键列上的多表连接操作驻留在 GPU 端, 将数据密集型的聚集计算驻留在 CPU 端, 这种 GPU 加速模式数据边界清晰, 可以最大化 CPU 与 GPU 端存储与计算的局部性, 最小化 CPU 与 GPU 之间 PCIe 通道的传输代价.

● 算子优化技术

OLAP 查询是从多维数据空间中通过切片或切块操作定位查询数据子集并进行分组聚集计算, 查询中核心的算子是连接、多表星形连接和分组聚集操作.

近年来连接优化技术是一个持续的学术热点问题, 以哈希连接在 CPU[22]和 GPU[23,24]上的优化技术为代表. 深入的研究表明, radix 分区哈希连接算法通过多趟 radix 分区技术将连接数据集优化为适合 cache 大小的数据子集, 从而优化哈希探测性能, 相对于不分区的哈希连接算法具有更好的 cache 局部性, 其性能在 CPU、GPU、Phi、FPGA 平台等均证明优于后者. 但从 OLAP 负载特征来看, OLAP 查询中最普遍的是庞大事实表与多个较小维表之间的星形连接操作, 而优化的 radix 分区哈希连接算法需要在每趟连接操作中物化中间结果, 增加了物化存储和

访问开销, 而且不能使用数据库优化的流水线处理模型, 在真实的负载中, 如 TPC-H 难以取得较好的性能[5,24]. 从连接算法优化技术来看, 哈希连接和排序归并连接算法是两种代表性的技术路线, 并随着 CPU 及 GPU 硬件技术的发展而不断提升性能. 在 CPU 平台, 排序归并连接算法主要通过 SIMD、NUMA 优化的并行排序归并算法、bitonic 归并网络等技术优化性能, 但性能仍然低于哈希连接算法. 在 GPU 端, 哈希连接算法与 CPU 端类似, radix 分区哈希连接仍然优于无分区哈希连接算法, 排序归并连接算法则通过 GPU 的归并路径优化技术取得较好的性能, 优于哈希连接算法. 但与 CPU 端连接优化技术类似, 在 OLAP 负载的多表连接场景下, 更为简单的无分区哈希连接算法通过流水线处理获得更高的性能[5], 在实际的 GPU 数据库系统中得到应用[21]. 分组聚集操作的算子实现与连接操作类似, 主要是哈希分组和排序分组, 基本策略是分组数较低 (低于 200 000, 哈希表小于 L2 cache 大小) 时哈希分组性能较优, 而较大分组时排序分组性能较高.

上述关系操作算法研究并未考虑模式特征, 而 OLAP 查询并不是单纯的关系操作算子的叠加, 而是一种基于多维数据模型的计算. 在理想的 OLAP 模型中, 维表定义了维和层次, 通过维和层次进行多维数据过滤 (切片和切块操作) 并分组, 最后执行多维空间数据检索和聚集计算. 传统的哈希连接及分组算法并不考虑模式的语义特征, 因此无法根据模式和 OLAP 负载的特征将多维分组下推到维表层, 只能通过连接操作实现后物化的分组操作. 另一方面, OLAP 负载通常包含上卷或下钻操作, 查询之间相似度较高, 查询优化的目标不仅是独立查询的优化, 还要考虑组查询的优化, 在查询中间结果的数据结构和物化策略上也有不同的优化方法.

总体而言, OLAP 查询优化不是单一的关系操作算子优化技术, 还可以根据其多维数据模型的特征定制优化的多维计算模型和索引, 提高查询性能和效率, 如基于维表映射机制的连接索引技术[18]、面向多维数据模型的向量连接和向量分组聚集技术、基于多维分组的 key vector[25]优化方法等. 与传统的基于哈希结构的连接和分组算法相比, 基于更加简单的多维地址映射机制的向量连接和向量分组聚集算法更适合 GPU 简单核心上的计算模式, 在计算性能和存储效率两方面均有较好的表现.

2 CPU-GPU 异构平台 OLAP 计算框架

本节讨论本文提出的面向 CPU-GPU 异构平台的 OLAP Accelerator 计算框架. 首先讨论逻辑的 OLAP Accelerator 计算框架技术, 然后讨论主要操作的实现技术和不同的查询处理模型, 最后讨论 CPU 内存计算、GPU 内存计算和 GPU 加速模式的查询处理模型.

2.1 OLAP Accelerator 计算框架

OLAP Accelerator 计算框架由数据管理模型和计算模型构成. 主流的 GPU 加速模型需要将数据从 CPU 内存传输到 GPU 端进行计算, GPU 显存用作查询处理过程中的缓存使用, GPU 端运行面向 GPU 硬件特性优化的算法来提高查询处理性能, 通常采用 PCIe 传输与 GPU 端计算的流水并行处理来减少数据传输代价, 其基本假设的 GPU 端的计算性能远高于 CPU 端. GPU 内存计算模型是将 GPU 显存作为工作数据集的长期存储设备, GPU 端存储的数据可以是原始数据的一个优化的副本, 如将 GPU 端存储的外键数据转换为基于代理键索引 (使用连续整数类型的主键) 机制的外键数据以支持更简单的向量连接, 消除连接操作中复杂的哈希表结构, 或者 GPU 存储压缩后的数据以优化存储效率等, 同时, 查询中间结果的物化策略也需要考虑 OLAP 组查询的共享数据结构的使用, 以减少 GPU 存储空间的消耗.

图 1 显示了 OLAP Accelerator 计算框架的 CPU-GPU 异构计算平台数据/负载分布模型. 以 SSB 基准为例 (SF=100, 事实表记录数量为 6 亿条), 星形模型是一个典型的偏斜的数据分布, 元数据 (4 个维表)、外键列、事实表度量列的数据量分别占数据总量的 1.03%、19.31% 和 79.66%, 而 3 个数据集上的计算时间占比则为 7.11%、86.43% 和 6.46%[26], 这种偏斜的数据量和负载特征为 GPU 加速和 GPU 内存计算模型提供了优化的数据与硬件平台匹配策略. GPU 内存计算模型提供事实表数据驻留 GPU 显存的 GPU 内存计算模式, GPU 加速模型则可以作为专用的外键列存储设计来加速执行代价最高的星形连接操作性能, 而且相对于 GPU 内存计算模型, GPU 加速模型能够加速 4 倍大小数据集. GPU 加速模型由模式、数据/负载的偏斜特征决定, 通过数据的预分区减少查询过

程中的数据移动代价, GPU 端的外键列执行星形连接操作, 输出选择率较低的向量索引, 可以作为一个硬件级的向量索引加速模块, 为数据库提供基于向量索引的 OLAP 加速技术.

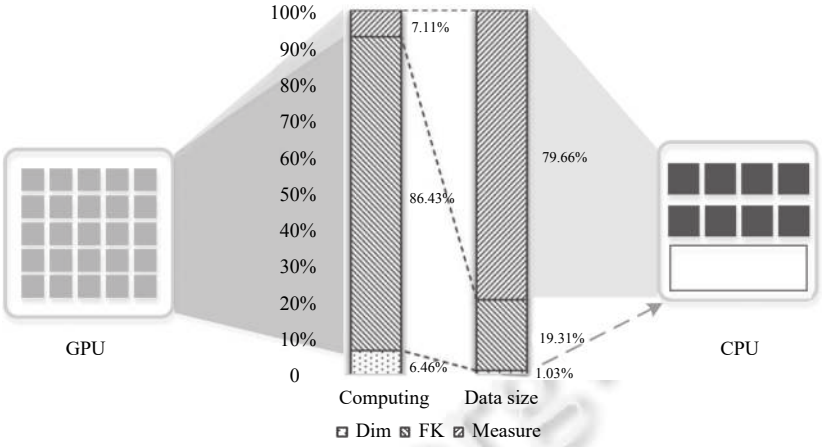

图 1 CPU-GPU 异构计算平台的数据/分布模型

本文 OLAP Accelerator 计算框架不同于已有 GPU 数据库计算框架的特征体现在以下 3 个方面.

(1) 元数据管理端的多维映射

维表是多维数据模型的元数据, 使用代理键索引作为主键将维表记录映射为多维数据的维度, 而且能够实现外键向维表记录基于地址映射的连接操作, 消除复杂的哈希表和哈希探测操作. GROUP BY 子句在维表端构建了一个分组立方体 (GCube), 进一步映射为分组向量 (*GVec*) 用于分组聚集计算, 将传统后物化的、位于连接操作后端的哈希分组聚集操作转换为“早分组, 晚聚集”策略, 实现将 OLAP 查询中 SPJGA (S: 选择, P: 投影, J: 连接, G: 分组, A: 聚集) 操作的 SPG 操作下推到较小的维表端. 在具体实现中为维表创建共享的向量索引, 空值表示不满足选择操作的记录, 非空值为满足选择操作的记录并且存储投影出 GROUP BY 属性的压缩编码, 各维表向量索引中存储的 GROUP BY 属性压缩编码构成了查询 GROUP BY 属性的分组立方体. 维表向量索引较传统的哈希表更小, 具有更好的 cache 局部性和 PCIe 传输延迟, 不同的查询仅改变向量索引中值的分布而使用共享的数据结构, 优化内存空间分配.

(2) 基于向量索引的计算

向量索引可以看作是一个多位位图索引, 支持高效的按位置访问. 维向量 (DimVec) 是元数据存储端用于查询改写和维映射的向量索引, 存储维表上的选择、投影和分组操作结果. 外键列可以用作连接索引, 通过向量连接算法将外键列的值映射到维向量相应的位置. 在外键列存储端, 向量索引用于存储连接中间结果, 即在多表连接过程中迭代计算映射的维向量中分组立方体的多维地址, 构建查询分组向量. 对于分组聚集操作, 向量索引通过与位图索引类似的功能直接访问满足连接过滤条件的事实表记录, 通过向量索引中存储的值将度量列的聚集表达式结果映射到分组向量对应单元中进行聚集计算.

(3) 灵活的计算模型

如图 2 所示, 基于向量索引的 OLAP 计算框架分为 3 个层次, 3 个层次可以灵活地与 CPU 或 GPU 硬件进行优化匹配. 查询改写和维映射是管理密集型负载, 适用使用 CPU 平台管理复杂多样的数据类型和查询管理任务. 多表连接是计算密集型负载, 在 20% 的数据集上产生 80% 的计算量[26], 适合 GPU 端的存储和计算. 聚集计算是数据密集型负载, 在 80% 的数据量上仅有 10% 的计算量. 第 1 个层次适合 CPU 端处理, 第 2、3 层次可以根据数据量大小、GPU 显存大小等硬件配置分配给 CPU 或 GPU 端存储和计算. 当 GPU (或多 GPU) 显存足够大时, 事实数据全部存储于 GPU 显存模式支持最高性能的 OLAP 计算性能, 当 GPU 显存容量不足时可以仅加速计算密集型的多表连接负载, 由 CPU 端完成数据密集型的聚集计算负载.

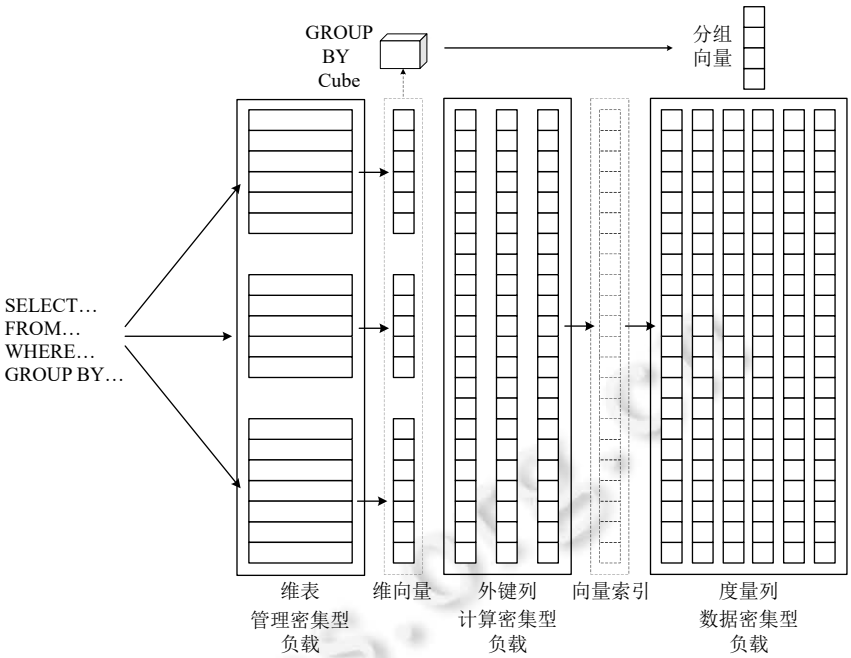

图 2 基于向量索引的 OLAP 计算模型

GPU 内存模式或 GPU 加速模式的选择策略是硬件配置和性能的权衡, 不同类型的负载需要灵活地与 CPU-GPU 异构计算平台优化匹配.

2.2 查询处理模型

本文以下面的 Q0 查询为例说明基于向量索引的 OLAP 查询处理算法实现.

SELECT *sum*(*exprice***discount*)

FROM *lineorder*, *date*

WHERE *lo_orderdate* = *d_datekey* AND *d_year* = 1993

AND *lo_discount* < 5 AND *lo_quantity* < 20

GROUP BY *d_season*;

如图 3 所示, GROUP BY 子句首先下推到维映射计算层, 通过对 date 表选择记录的 GROUP BY 属性进行压缩创建字典表并将压缩编码存储于维向量索引 (*DVec*) 中, 维向量索引是用来存储维表选择之后生成的结果, 记录了通过过滤的 GROUP BY 属性的字典压缩值, 在之后的图表和文字中均用 *DVec* 表示. 事实表位图 *bitmap* 用于存储 *lo_discount* < 5 AND *lo_quantity* < 20 选择表达式的结果, 然后过滤外键列 *lo_orderdate*, 通过外键值地址映射到维向量索引相应的位置获得分组编码, 并存储于向量索引 (*VecInx*) 中, 非空值用于映射访问相应的度量列, 例如 *exprice* 和 *discount* 记录进行聚集计算, 并将结果存储到向量索引值映射的分组向量 (*GVec*) 对应单元. Q0 代表了星形模型上典型的 OLAP 查询任务, 不同的查询生成不同的维向量索引、不同的分组编码, 可以共享相同的位图、向量索引、分组向量数据结构进行计算. 传统数据库通常使用 B+树索引、R 树索引、位图连接索引等技术加速查询性能, 索引的存储和管理代价较高. 当选择率较低时, 定长的位图和向量索引可以采用压缩结构, 如 OID 向量 (压缩位图索引) 或<OID, VALUE> (压缩向量索引).

图 3 中白色底纹列是基础数据列, 是查询中的冷数据, 用于顺序访问或按位置随机访问. 斜向底纹部分的位图索引、向量索引、分组向量可以在查询中动态创建, 可以被不同查询共享使用相同的数据结构, 在查询处理中用作中间结果存储数据结构, 是查询中的热数据, 在向量化查询处理中通过 L1 cache 进行优化. 横向底纹的维向量用于连接操作, 是线程间共享访问数据结构, 是查询中的暖数据, 通过 L3 共享 cache 进行优化访问. 为了优化查询性能和中间结果物化代价, 本文设计了不同的查询处理模型.

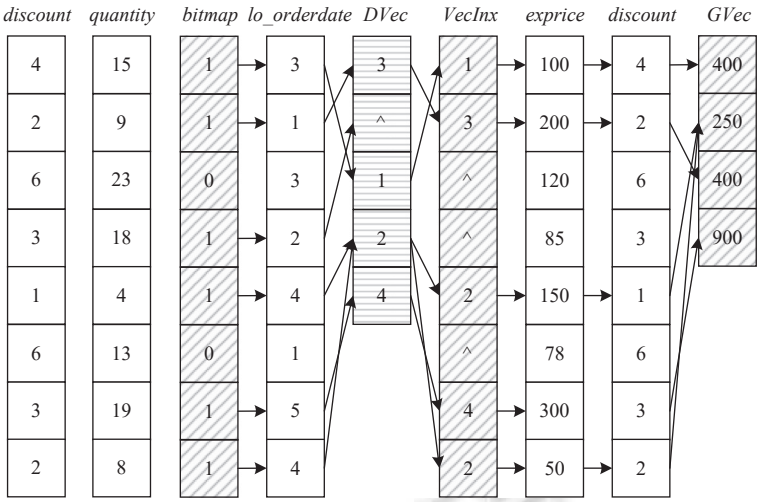

图 3 基于向量索引的 OLAP 查询处理示例

● 行处理模型

本文的行处理模型基于编译的推模型, 列存储数据在查询中以行方式访问记录并执行流水线查询处理, 不需要位图、向量索引等中间结果数据, 表中记录被顺序访问, 需要大量的分支语句判断记录的处理, 在低选择率时性能提升不显著.

● 列处理模型

列处理模型将流水线处理分解为一系列基于列数据的处理阶段, 每个列完成全部的数据处理, 代码执行效率和 cache 局部性较好, 但一次一列的处理模式需要将中间结果物化下来, 产生额外的物化数据存储和访问代价. 列处理模型中需要物化位图索引和向量索引, 尤其在 GPU 处理时消耗了额外的显存存储空间, 需要把中间数据存储到 GPU 的 global memory 全局内存. 对于低选择率的查询, 列处理模型通过位图和向量索引记录满足条件记录的 OID, 在下一列的扫描中可以消除大量不必要的数据访问, 能够显著提高查询性能.

● 向量化处理模型

向量化处理模型是一种 cache 内的列处理模型[27], 通过将数据划分为适合 L1 cache 大小的向量执行基于向量粒度的批量列式处理, 位图、向量索引等中间结果物化在 L1 cache 中, 既保持了列处理模型较高的代码执行效率, 又通过 L1 cache 的物化机制消除中间结果的内存物化和访问代价, 从整体上也体现了流水线处理模型的特征, 可以看作是行处理模型和列处理模型面向 cache 结构优化的混合处理模型. 向量化查询处理模型广泛应用于现代内存数据库系统, 如 Actian Vector、SQL Server、OmniSciDB 等, GPU 平台上的向量化查询处理技术也有了初步的探索[5], 但当前 GPU 数据库主要还是采用列式处理模型[21].

2.3 OLAP 查询处理模型在 CPU 和 GPU 端的实现技术

CPU 和 GPU 在核心和 cache 结构上有显著的不同, OLAP 查询处理模型在 CPU 和 GPU 端的实现和性能也有较大的不同, 本文重点讨论 CPU 和 GPU 端基于向量索引的向量化查询处理实现技术.

● CPU 端查询处理模型实现技术

图 4 和图 5 显示了 CPU 端基于向量索引的 OLAP 查询处理模型. 行式与列式处理模型将数据按线程数据划分为逻辑行组 (图中灰色分片), 每个线程独立处理数据分片上的 OLAP 查询处理任务, 再将线程间的查询结果归并为统一的查询结果. 向量化查询处理模型则将数据划分为适合 cache 大小的向量分片, 作为流水线处理的数据粒度. 行式处理模型由线程依次访问每一条记录, 流水执行选择、连接、分组、聚集等操作, 在分组向量 (*GVec*) 中记录当前记录的聚集结果. 列式处理和向量化处理模型需要物化位图索引和向量索引, 列式处理模型中每个线程一次处理完整的数据分片上的列, 在内存物化中间结果, 在向量化处理模型中, 线程一次处理适合 L1 cache 大小

的数据分片, 中间结果在 cache 中物化和访问, 主要区别是线程一次处理的列与向量的长度不同, 在微观的算法实现上是相似的.

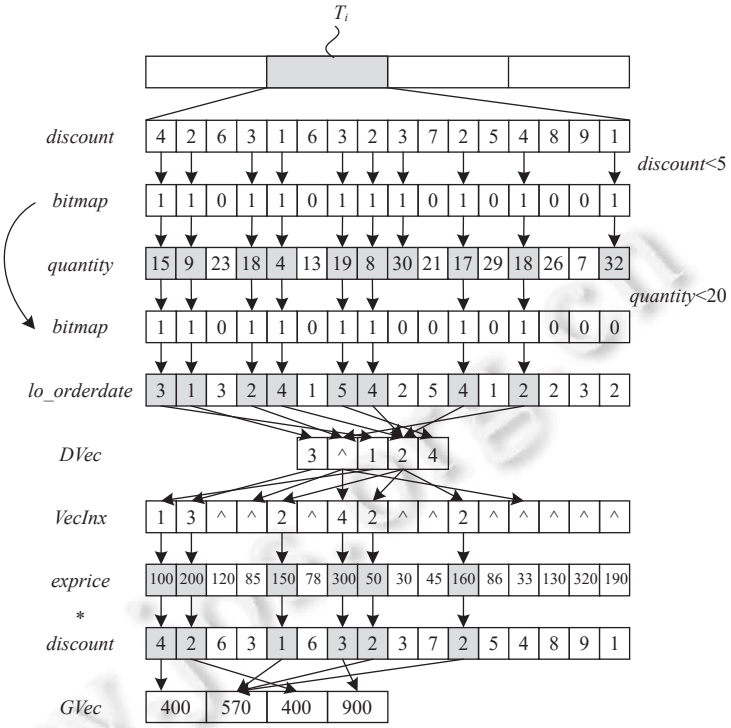

图 4 基于向量索引的 OLAP 查询处理

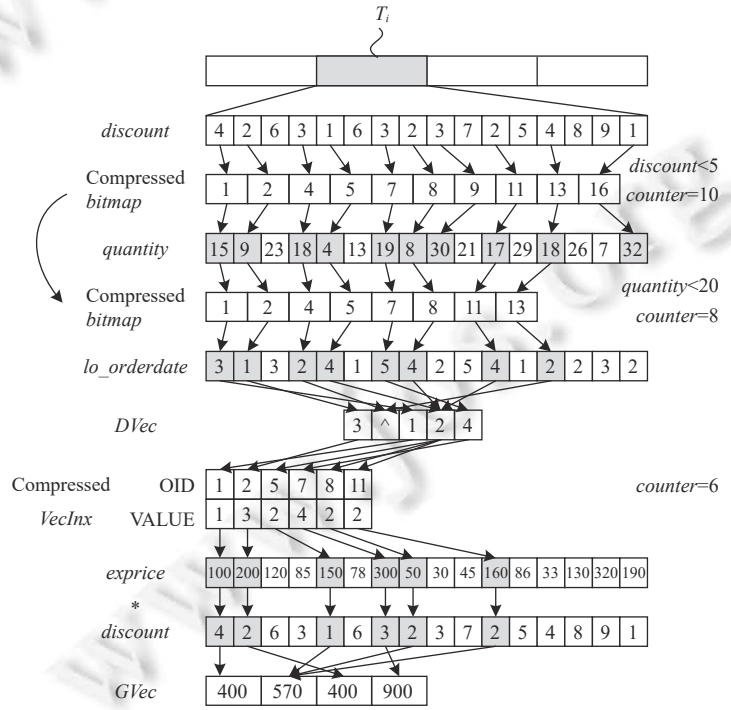

图 5 基于压缩向量索引的 OLAP 查询处理

图 4 的 OLAP 查询处理示例中位图与向量索引为定长结构, 选择操作 *discount*<5 的结果存储到位图索引中, 并通过位图索引按位置访问下一个选择列 *quantity*, 并根据选择操作 *quantity*<20 结果更新位图索引. 选择操作执行完后根据位图索引访问连接列 *lo_orderdate*, 通过向量连接操作访问维向量索引 *DVec*, 将连接操作结果存储在向量索引 *VecInx* 中, 最后按向量索引按位置访问度量列 *exprice* 和 *discount* 记录, 并将聚集表达式结果按向量索引存储的编码映射到分组向量 *GVec* 单元中聚集计算.

图 5 为基于压缩向量索引的 OLAP 查询处理过程示例. 主要区别在于使用压缩的位图和向量索引存储查询中间结果, 压缩位图、向量索引的长度, 消除查询中的分支操作, 提高代码执行效率. 使用压缩位图和向量索引时, 数据宽度增加, 如位图由 1 位增加到一个 int 型 OID, 向量索引由一个 int 值增加到一个 int 型 OID 和一个 int 型的 VALUE, 但总长度减少. 单查询处理时压缩的位图和向量索引在低选择率时能够节省存储空间, 但 OLAP 查询通常为组查询, 每个查询独立存储的压缩位图和向量索引无法共享并且产生较多的内存空间碎片, 降低内存利用率. 本文使用了定长结构的压缩技术, 即按 100% 选择率创建压缩位图和向量索引, 满足全部查询不同选择率的需求. 所有查询共享访问相同的压缩位图、向量索引内存结构, 满足条件的压缩数据顺序存储在压缩位图与数据结构的顶端, 支持其上的连续访问. 在列式处理模型中, 压缩位图和向量索引的存储开销大于定长的位图和向量索引, 增加了内存存储空间开销. 在向量化查询处理模型中, 压缩位图和向量索引以较小的向量形式物化在 cache 中, 内存开销较小, 共享的压缩位图和向量索引结构简化了内存空间分配, 提高了低选择率时的查询效率.

● GPU 端查询处理模型实现技术

GPU 显存相对于 CPU 内存容量更小、带宽更高, 增加 GPU 显存存储数据量、提高显存利用率与提高 GPU 计算性能同等重要. 本文采用的基于向量索引的向量连接和向量分组聚集算法使用较小而且可以查询间共享的向量结构, 从而可以预先确定 GPU 显存分配策略, 除维向量、位图索引、向量索引之外的显存均可分配给事实数据存储, 最大化 GPU 显存的数据存储能力.

在行处理模型中, 通过流水线处理模型可以消除位图索引和向量索引的存储开销, 维向量 *DVec* 和分组向量 *GVec* 用于存储查询输入数据. 维向量中存储的是维表查询中生成的 GROUP BY 属性压缩编码, 分组向量是 GROUP BY 属性压缩编码, 压缩技术实现了维向量和分组向量的最小化. OLAP 交互查询通常分组数量较小, 维表较小, 因此维向量和分组向量可以通过 GPU 的 L1 和 L2 cache 进行缓存优化访问. 假设 GPU 显存容量为 M, 则 GPU 可以存储的最大事实记录的行数为:

$$R=\frac{M-sizeof(DVecs)-\max(GVec)}{width(fact\_tuple)} \tag{1}$$

行存储模型最大化了 GPU 显存和 cache 利用率, 通过流水线处理模型消除中间结果的物化代价, 但在选择和连接操作处理过程中涉及较多的分支判断操作, 在低选择率查询时性能可能受到一定的影响.

列处理模型一次处理整个数据列, 适合 GPU 的 SIMT 计算模型, 可以最大化 GPU 的并行计算能力. 但列处理模型需要将查询中间结果物化在 GPU 显存, 通过物化的位图、向量索引来执行下一列上的操作. GPU 计算使用 BLOCK 和 THREAD_BLOCK 来管理线程对数据的访问, 以图 6 为例, 假设 BLOCK B_i 中的 THREAD_BLOCK 数量为 4, 包含线程 T_0–T_4, 每个 THREAD_BLOCK 中的线程访问连续的数据, 完整的数据列在 GPU 处理时可以映射为#BLOCK 个逻辑数值矩阵, 每个 BLOCK 处理一个矩阵, 每个线程处理物理上不相邻的矩阵列数据. 基于向量索引的 OLAP 算法使用两个中间结果矩阵, 位图矩阵和向量索引矩阵, 矩阵存储可以采用定长方案和压缩方案. 定长位图和向量索引每个线程处理相同长度的矩阵列数据, 选择和连接中间结果存储在定长的位图或向量索引中, 通过空值标识不满足选择或连接条件的记录, 在执行下一个操作时需要扫描定长的位图或向量索引并进行分支判断操作, 由于 GPU 的分支预测优化技术相对 CPU 较差, 对性能有一定的影响. 压缩位图和向量索引结构如图 7 所示, 位图矩阵增加宽度, 存储的不是状态值 0 或 1, 而是满足选择条件记录的 OID 连续序列, 同时为线程增加相应的计数器, 控制线程对压缩位图或向量索引的访问长度. 向量索引压缩时变成两个向量, 分别记录满足条件记录的 OID 和存储的分组编码值. 相对于定长位图和向量索引, 压缩技术采用定长压缩的位图和向量索引矩阵存储以满足不同的选择率, 增加了位图和向量索引矩阵的存储开销, 在计算时减少了分支语句, 提高了代码的执行效率.

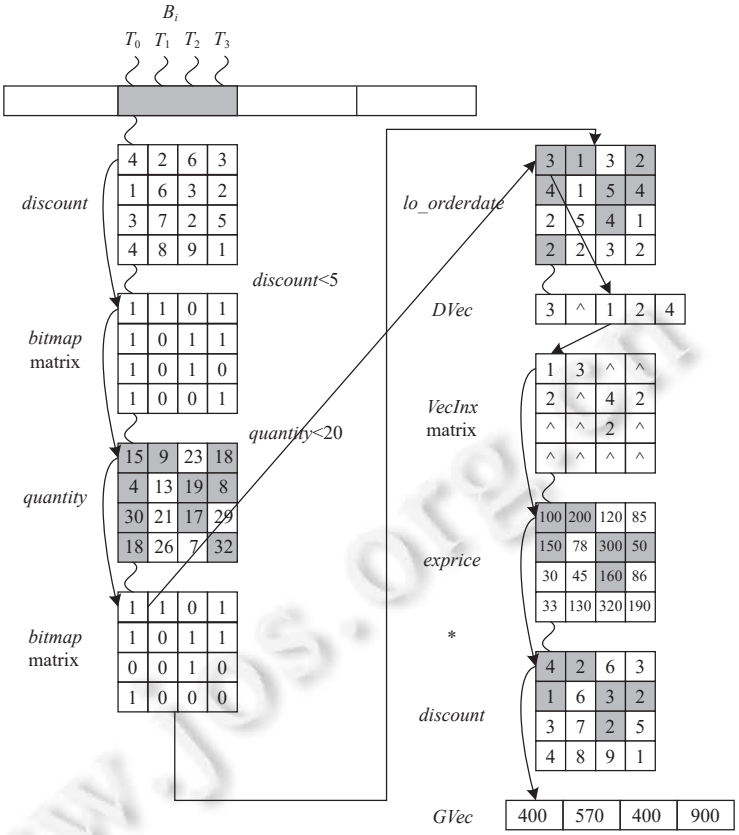

图 6 基于向量索引的 GPU OLAP 查询处理

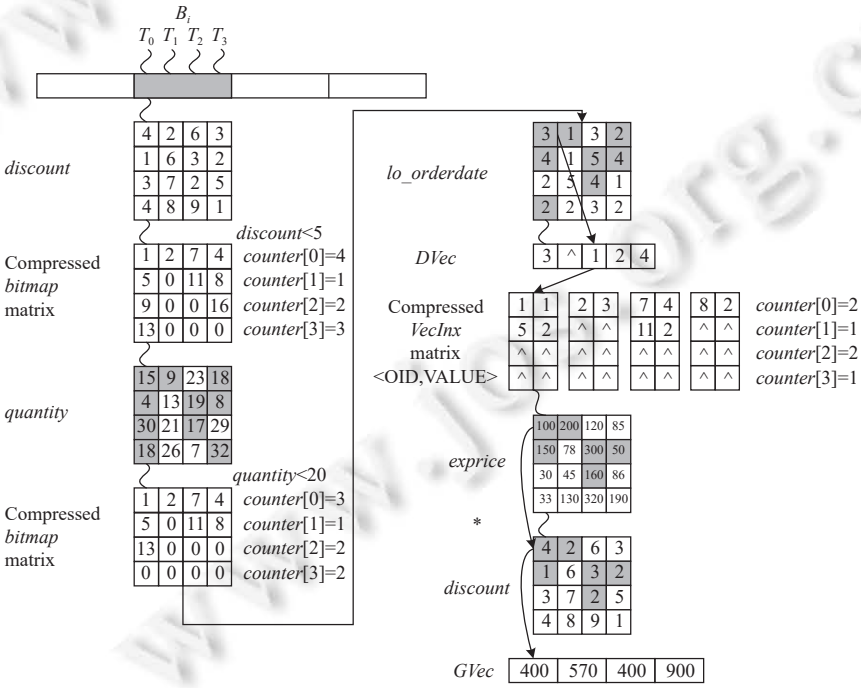

图 7 基于压缩向量索引的 GPU OLAP 查询处理

向量化处理模型把数据列划分为适合 cache 大小的向量分片, 每个分片由一个线程完成全部的查询处理任务, 查询产生的中间结果也以向量为单位物化在 cache 中, 从而最小化列处理模型中中间结果的内存物化代价. 向量化查询处理的方法与图 6 和图 7 类似, 不同之处在于需要根据 GPU 每个 SM 处理器共享内存的大小和 THREAD_BLOCK 参数确定优化的向量大小, 然后管理线程对数据的访问. 假设向量大小为 v, 通过循环控制使每次迭代处理中每个线程处理 v 个数据, 则每个 BLOCK 处理的数据形成一个 $n\times v$ 的矩阵, 每个线程对应长度为 v 的向量. 基于向量索引的 OLAP 算法根据可用的 GPU 共享内存大小和 THREAD_BLOCK 大小计算向量大小, 并通过实验测试确定最优的向量长度参数. 定长与压缩的位图和向量索引均存储在共享内存中, 消除 GPU 显存空间的占用并且优化访问性能.

以 NVIDIA V100 GPU 为例, 一块 V100 由 80 个 SM 组成, 每个 SM 内部有 64 个 cuda 核心和统一的 128 KB L1/共享内存, 其中 96 KB 可以用作共享内存. 在实际应用中, 我们可以使用 48 KB 的共享内存用于位图和向量索引存储, THREAD_BLOCK 设置线程数为 1 024, 位图使用 1 字节存储, 向量索引宽度为 2 字节, 压缩位图和向量索引的 OID 宽度为 4 字节, 值为 2 字节, 图 6 和图 7 所示的定长与压缩位图与向量索引算法中向量长度分别为 16 和 8, 实验测试最优参数为 16 和 6.

向量化查询处理中定长位图与向量索引方法中向量长度的计算公式为:

$$Vec\_len=\frac{Shared\_mem\_size/thread\#}{width(bitmap)+width(VecInx)} \tag{2}$$

向量化查询处理中压缩位图与向量索引方法中压缩位图的 OID 列可以与压缩向量索引的 OID 列复用, 向量长度的计算公式为:

$$Vec\_len=\frac{Shared\_mem\_size/thread\#}{width(ComVecInx)} \tag{3}$$

在向量化查询处理中, 每次迭代处理的向量分片数据量为 Vec_len×BLOCK×THREAD_BLOCK, 每个向量分片处理时对显存中的数据列只读访问, 位图和向量索引在更快的共享内存中迭代更新并用于对显存中的数据列进行索引访问, 优化位图和向量索引的存储空间开销.

综上所述, 向量化查询处理可以看作是行式流水处理和列式处理技术的融合, 通过适合 cache 或 GPU 共享内存大小的向量划分数据, 每个向量分片上执行基于向量粒度的流水线处理, 中间结果以向量形式物化在 cache 或共享内存中, 执行向量粒度的流水线处理. CPU 与 GPU 平台上向量化查询处理技术的主要区别在于: (1) CPU 线程处理连续的向量数据, GPU 线程处理不连续的向量数据; (2) CPU 的向量缓存在核心私有 cache 中, 由线程串行访问, 而 GPU 的向量由 SM 的 THREAD_BLOCK 中的线程在共享内存中并发访问; (3) CPU 较大的私有 cache 支持较大的向量长度, 而 GPU 多线程共享相同的共享内存方式支持的向量长度较小; (4) CPU 线程在查询处理时可以使用私有的位图、向量索引和分组向量用于查询处理, 而 GPU 可以使用私有的位图和向量索引, 分组向量则在 GPU 的 L1 cache 或共享内存中由各线程共同访问, 需要支持并发访问机制.

3 CPU 和 GPU 端算子实现技术

查询处理模型主要优化算子之间中间结果的物化访问, 而算子则决定关系操作的性能. 在基于向量索引的 OLAP 查询处理模型中, SPG 操作下推到较小的维表端执行, 主要由 CPU 完成, 在事实表数据上的 SJA 操作是 OLAP 查询性能的主要决定因素.

3.1 选择操作

较小维表上的选择操作集成到维向量映射操作中, 本文主要讨论事实表上的选择操作实现技术. 事实表上的选择操作主要为多列数据上的多谓词操作, 选择率依次降低, 选择操作的结果存储于位图或压缩位图中. 谓词处理可以采用分支判断 (branching, if-statement) 或无分支判断 (non-branching, branch-free predication) 方法, 如算法 1–算法 3, 分别为位图过滤、分支压缩位图过滤和无分支压缩位图过滤算法. 以算法 1 和谓词条件 $discout$<5 and $quantity$<20 为例说明选择操作的实现, 首先根据谓词条件 $discount$<5 过滤得到中间位图 $bimap$, $bitmap$ 中的每个

记录标记当前位置通过了过滤, 然后按照 *bitmap* 和 *quantity* 进行 *quantity*<20 的二次过滤, 生成最终位图 *bitmap*. 算法 2 和算法 3 在处理谓词判断和最终结果不同. 位图过滤算法使用定长的位图记录选择操作中间结果, 对不同选择率的查询都使用相同大小的、较小的位图数据结构, 适合于 GPU 每个线程较小私有 cache 的硬件配置. 压缩位图存储满足连接条件的 OID, 数据宽度大于定长位图, 当选择率较低时压缩位图的数据量小于定长位图, 但考虑到 OLAP 查询上卷、下钻操作的组查询模式包含较大范围的选择率变化, 本文采用定长的压缩位图方式减少压缩位图空间分配的碎片化, 但定长的压缩位图存储空间增大, 适合于向量化查询处理方法利用 L1 cache 进行缓存物化. 无分支压缩位图选择算法消除了分支操作, 更适合 GPU 处理的应用.

算法 1. 位图过滤.

输入: 谓词列 *discount* 和 *quantity*, 谓词列长度 *FRows*;
输出: 过滤位图 *bitmap*.

伪代码:

```
BEGIN
FOR i TO FRows DO
  IF discount[i]<5 THEN
    bitmap[i]=1;
  ELSE bitmap[i]=0;
  END IF
END FOR
FOR i TO FRows DO
  IF bitmap[i]=1 AND quantity<20 THEN
    bitmap[i]=1;
  ELSE bitmap[i]=0;
  END IF
END FOR
END
```

算法 2. 分支压缩位图过滤.

输入: 谓词列 *discount* 和 *quantity*, 谓词列长度 *FRows*;
输出: 压缩过滤位图 *c_bitmap*, 长度 *k*.

伪代码:

```
BEGIN
FOR i TO FRows DO
  IF discount[i]<5 THEN
    c_bitmap[j++]=i;
  END IF
END FOR
FOR i TO j DO
  IF quantity[c_bitmap[i]] <20 THEN
    c_bitmap[k++] = c_bitmap[i];
  END IF
```

END FOR
END

算法 3. 无分支压缩位图过滤.

输入: 谓词列 *discount* 和 *quantity*, 谓词列长度 *FRows*;
输出: 压缩过滤位图 *c_bitmap*, 长度 *k*.

伪代码:

```
BEGIN
FOR i TO FRows DO
    c_bitmap[j]=i;
    j+=discount<5;
END FOR
FOR i TO j DO
    c_bitmap[k]=c_bitmap[i];
    k+=quantity[c_bimap[i]]<20;
END FOR
END
```

行处理模型比较适合选择率较高的情况, 选择率较低时列式处理和压缩位图方法效率较高但物化的压缩位图增加了内存物化存储和访问开销, 向量化处理模型在 L1 cache 物化压缩位图, 降低了压缩位图的物化空间开销和访问代价. 对于显存容量较小的 GPU 来说, 行式处理和向量化处理都有较高的内存利用率.

3.2 连接和星形连接

如算法 4 所示, 本文采用向量连接算法实现 OLAP 数据库中事实表与维表之间基于主键-外键的等值连接操作. 向量连接算法是一种特殊的索引连接, 主键表使用代理键作为主键, 连续的键值不仅提供唯一性约束, 还提供了面向多维模型的地址映射, 支持基于外键值直接映射到主键表记录的偏移地址. 在 OLAP 查询中, 维表执行 WHERE 和 GROUP BY 子句, 将选择出的 GROUP BY 属性进行字典压缩并使用压缩编码作为该记录的输出值, 维表创建的向量索引存储了选择操作的结果 (空值和非空值)、分组操作的结果 (分组编码) 和连接映射 (维向量索引), 由外键列直接进行基于键值的地址映射访问. 向量连接使用定长的维向量索引, 存储的压缩编码有效降低了数据量, 具有更高的 cache 局部性, 对不同选择率的查询使用共享的数据结构, 节省了内存空间.

算法 4. 定长列式 OLAP 星形连接算子.

输入: 维度表主键 *PK*, 事实表外键列集合 *FK*, GROUP BY 字典表记录数量集合 *Card*;
输出: 向量索引 *VecInx*.

伪代码:

```
BEGIN
FOR tid TO tablenum DO
  FOR i TO len DO
    idx=PK[tid][FK[tid][i]]; //连接映射
    IF idx !=0 AND VecInx[i]!= –1 THEN
      VecInx[i]+= idx×Card[tid];
    ELSE VecInx[index]= –1; //没有通过连接
```

```
    END IF
  END FOR
END FOR
END
```

算法 4 仅代表定长列式的处理方法, 相对应还有行式, 向量化以及它们各自的压缩处理方法, 算法 4 中以 *VecInx* 作为最终的输出结果. 首先根据外键列映射到维向量以及之前已经连接的结果判断当前记录是否被淘汰, 如果未被淘汰, 根据维表分组字典值计算多维分组地址, 记录到 *VecInx* 中完成计算. *VecInx* 相当于通过新型向量索引技术对传统查询处理过程中的产生的中间结果进行压缩, 并起到与位图索引相同的索引访问作用. 向量索引既起到位图一样的连接过滤作用, 又作为多表连接时 GROUP BY 多属性分组编码计算的存储结构, 在算法设计上可以使用定长向量索引、基于分支操作压缩向量索引和基于无分支操作的压缩向量索引.

3.3 分组聚集计算

如前文图 2、图 3 所示, 本文采用了基于向量索引的分组聚集计算技术. 通过维表管理端维映射操作实现对 GROUP BY 属性的“早分组”操作, 将其压缩到维向量索引中, 通过多表星形连接操作迭代更新向量索引中的分组编码, 最后通过向量索引在事实表数据上执行“晚聚集”模式的分组聚集计算. 算法 5 描述了定长向量分组聚集的核心实现过程, *VecInx* 是前一节星形连接的最终结果, 定长向量索引, 里面存储了每个 GROUP BY 属性的分组地址, 所以只需要按照 *VecInx* 的值进行聚集计算存储到 *GVec* 即可. 也就实现了“早分组, 晚聚集”的目的, 与传统基于哈希的分组聚集方法等价, 实现了通过多维数组映射 GROUP BY 多属性.

算法 5. 向量分组聚集算子.

输入: 向量索引 *VecInx*, 长度 *len*, 度量列 *exprice* 和 *discount*;
输出: 分组向量 *GVec*.

伪代码:

```
BEGIN
FOR i TO len DO
  IF VecInx[i]!= –1 THEN
    GVec[VecInx[i]]+=exprice[i]×discount[i];
  END IF
END FOR
END
```

由于 CPU 和 GPU 在处理器结构和线程结构上的不同, 向量分组聚集计算的实现在 CPU 和 GPU 平台上有一定的差异. 图 8 显示了在 CPU 平台上向量分组聚集计算示例. 压缩的向量索引<OID, VALUE> 被 CPU 线程逻辑划分, 每个线程处理对应的压缩向量索引分片, 创建线程私有的分组向量 Private *GVec*, 在压缩向量索引中 OID 的映射下直接访问事实表度量列上满足选择、连接条件的记录, 将聚集表达式结果按 VALUE 值映射到分组向量中进行聚集计算. 线程在数据分片上是串行处理, 私有分组向量使用无锁结构, 每个线程生成自己线程数据分片上的局部分组向量, 最后通过线程间的归并计算生成查询的全局分组向量. 由于交互式 OLAP 查询中分组数量相对较小, CPU 较大的私有 L1、L2 cache 可以缓存较大的分组向量 (如 1 MB L2 cache 可以缓存 $2^{20}/2^3=2^{17}$ 个 double 或 big int 类型的分组, 满足大多数 OLAP 查询的分组需求), 因此分组聚集计算有较高的 cache 局部性, 性能较好.

GPU 每个 SM 中的 cuda 核心共享较小的 L1/共享内存 (如 V100 为 96 KB), 每个线程支持的私有分组向量太小, 因此采用 SM 的 THREAD_BLOCK 线程块共享分组向量. 如图 9 所示, 每个 THREAD_BLOCK 中的线程对应一个数值矩阵, 每个线程按向量索引中的 OID 访问其对应的列, 完成聚集计算, 并将结果更新到共享的分组向量

GVec 中, 分组向量的更新需要 GPU 端的 latch 机制保证其正确性, 最后各 SM 内的分组向量归并为统一的分组向量, 输出 OLAP 计算结果.

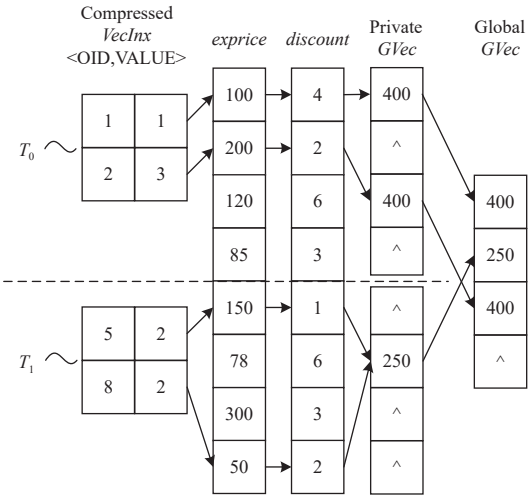

图 8 CPU 端向量分组聚集计算

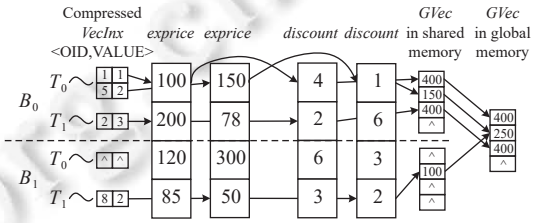

图 9 GPU 端向量分组聚集计算

综上所述, 在 OLAP 算子实现技术上通过向量索引简化了 GPU 端存储的数据类型、数据结构和算法设计, 消除了执行效率较低的分支操作, 通过分组压缩技术缩小了分组向量使其在 CPU 和 GPU 平台上具有更好的 cache 局部性.

4 实验分析

4.1 实验设计和配置

基于向量索引的 OLAP 计算模型将整个 OLAP 查询处理过程分解为 3 个阶段, 每一个处理阶段可以根据数据集大小、存储容量、性能需要等因素独立地分配给 CPU 或 GPU 平台. 本文在实验中评估了 CPU 内存计算、GPU 加速和 GPU 内存计算 3 种 OLAP 查询处理模型, 覆盖了 CPU-GPU 混合计算平台上的典型应用场景.

实验测试平台为一个 2U 的机架服务器, 配置两块 Intel Xeon Gold 6138 @ 2.00 GHz CPU, 512 GB DDR4 内存, 共有 40 个物理核心 80 个物理线程, 每个 CPU 的 L3 cache 为 27.5 MB. 服务器 Linux 内核版本为 3.10.0-1062.4.1.el7.x86_64. 服务器配置一块 NVIDIA Tesla V100 GPU, 集成了 5120 cuda 核心和 32 GB HBM2, 带宽性能达到 900 GB/s. V100 的每个 SM 包含 128 KB L1 和共享内存, L2 cache 容量为 6 MB. 我们开发了基于 C 和 cuda 10.0 的基于向量索引的 OLAP 计算引擎, 支持 CPU 内存计算、GPU 加速和 GPU 内存计算 3 种处理模型, 其中维表上的维向量映射均在 CPU 端执行, GPU 加速模型通过将事实表外键列存储在 GPU 高带宽显存中支持 GPU 上的多表星形连接, 用作硬件级的向量索引计算引擎, 通过 GPU 生成向量索引, 服务于 CPU 内存事实表上的分组聚集计算; GPU 内存计算模型则将全部事实表存储于 GPU 高带宽显存, 支持 GPU 事实表上的多维分析计算任务, 生成最终的分组向量. GPU 加速模型只在 PCIe 通道中传输较小的维向量和压缩向量索引, GPU 内存计算模型只传输维向量和分组向量, 实现了 PCIe 通道上数据传输的最小化.

本文使用 SSB 基准测试, 数据集大小为 SF=100, 数据集包含 4 个维表和一个事实表, 数据量分别为 2 555 (Date 表)、200 000 (Supplier 表)、1 400 000 (part 表)、3 000 000 (customer 表)、600 038 144 (lineorder 表). 整体事实表大小为 26 GB, 4 个外键列大小为 8.4 GB. 一块 V100 GPU 的 32 GB 显存可以容纳 SF=100 事实表的全部 GPU 内存计算任务, 或者支持 SF=330 数据集的事实表外键 (4 个外键列和 1 个向量索引列) GPU 向量索引计算加速任务.

本文在性能测试中使用了内存数据库 Hyper、OmniSciDB、Vector、MonteDB 和 PG-strom 作为对比对象, OmniSciDB 使用 GPU 内存计算模式. OmniSciDB 在 GPU 端的优化策略是尽可能将热数据驻留在 GPU 显存支持

GPU 内存计算, 在不同测试模式下性能差异较大. 我们测试了 OmniSciDB 的 CPU 和 GPU 版本在暖机模式和最短执行时间两种模式时的执行时间, 暖机模式通过预执行组内查询实现数据从磁盘向内存或从内存向 GPU 显存的加载, 查询平均时间不计入暖机时间, 以客观评估最优执行模式下的性能. CPU 版本无暖机批量执行模式查询执行时间是重复执行最短时间模式的 85 倍, 但暖机批量执行模式和重复执行取最短时间模式的平均查询时间相近. GPU 版本的重复执行最短时间和暖机批量查询执行时间相差 2.6 倍, 本文在测试中选择最短执行时间模式来评估 OmniSciDB 在 CPU 和 GPU 上的性能上限.

4.2 基于向量索引的 OLAP 计算实现技术性能对比测试

实验对比了基于向量索引的 CPU 内存计算、GPU 内存计算和 GPU 加速模型的 OLAP 查询处理性能. 图 10 显示了 3 种计算模型的平均查询时间, 每种计算模型又分为行处理、列处理和向量化处理 3 种实现方式, 行处理模型在 GPU 端实现了基于共享分组向量和私有分组向量两种实现技术, CPU 端只支持私有分组向量技术. 列处理和向量化处理模型分别实现了基于定长向量索引和压缩向量索引的实现技术. GPU 加速模式只在 GPU 端存储事实表外键列, 从 CPU 端获取维向量后执行 GPU 端的星形连接操作生成向量索引, 再将向量索引通过定长或压缩方式通过 PCIe 通道传回 CPU 端执行分组聚集计算. 实验结果给出了如下结论.

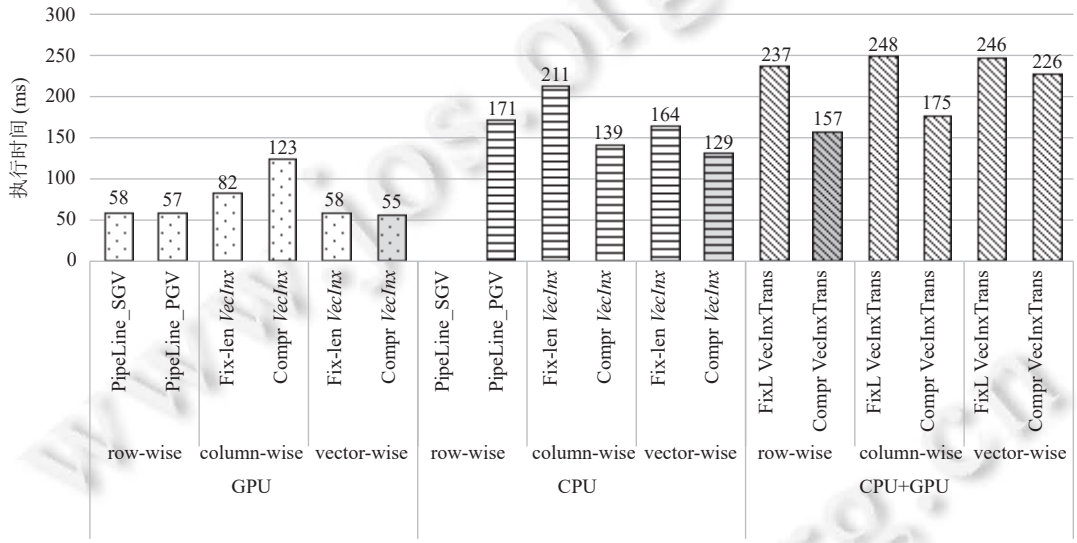

图 10 GPU 内存计算、CPU 内存计算和 GPU 加速模式 SSB 基准测试性能对比

(1) GPU 内存计算模型中向量化查询处理模型性能最优, 压缩向量索引技术略优于定长向量索引技术, 主要原因是 GPU 较大的线程数量和较小的共享内存支持较短的向量, 因此查询选择率的变化对性能影响降低. 行式处理模型性能与最优的向量化查询处理模型性能相近, 在实现技术上更为简单. 列处理模型需要在显存中物化向量索引, 较大的内存访问延迟降低了性能, 而且简单的定长向量索引模式性能优于压缩向量索引模式, 主要原因是压缩向量索引数据宽度增加, GPU 的线程块结构使不同线程的压缩向量索引存储在不连续的物理地址空间中, 难以发挥压缩向量索引的数据访问效率. 整体来看, 消除 GPU 端全局内存访问延迟是提高性能的重要条件, 行处理模型的流水线处理方法和向量化处理方法通过 GPU 的共享内存消除全局内存访问, 表现出较好的性能.

(2) CPU 内存计算模型中基于压缩向量索引的向量化查询处理模型性能优于压缩向量索引的列处理模型, 优于行处理模型, CPU 较大的私有 cache 和连续的压缩向量索引存储访问有效地提高了列处理和向量化查询处理模型的性能. CPU 优化的核心技术是提高查询处理时数据在 cache 的局部性, 向量化查询处理利用 L1 cache 缓存查询中间结果, 通过压缩向量索引提高低选择率查询处理时的数据访问局部性, 从而提高查询的整体性能.

(3) GPU 加速模型只执行外键列的连接操作, 访问的数据列减少时行处理模型性能较优, 当查询选择率低于 1.9% 时, 压缩向量索引方法可以提高 PCIe 传输性能, 当查询选择率高于 1.9% 时, 定长向量索引传输方法整体性

能更好. GPU 加速模型整体性能与 GPU 内存计算模型有较大差距, 相对于 CPU 内存计算模式也略有差距. GPU 内存计算模型在 V100 的 32 GB 显存下最大支持 SF=100 数据集, 但 GPU 加速模型可以支持的数据集超过 SF=300, 而 CPU 内存计算模型则可以支持 TB 级数据集.

实验平台为双路 CPU 和一块 GPU, 内存计算模型已获得最高性能, 但 GPU 加速模型只表现出最低性能, 通过扩展多块 GPU 卡, GPU 内存计算和 GPU 加速模型不仅可以支持更大的数据集, 而且可能通过多 GPU 卡的并行处理获得线性化的性能加速能力.

4.3 性能分析

在基于向量索引的 OLAP 计算模型中, 整个数据集划分为 3 个数据子集: 维表定义为多维元数据, 描述维、层次等结构信息, 数据类型多样化, 数据量较小; 事实表外键列定义为计算密集型数据集, 通过整数据映射维结构, 数据集中等大小但计算强度较大; 事实表度量列数据量较大, 由数值型数据组成, 主要用于聚集计算, 执行大数据量上稀疏的低强度计算. 在 3 种计算模型中, 维表都存储在 CPU 端执行查询的维向量映射任务, 生成查询对应的维向量, 星形连接和分组聚集操作可以分配给 CPU、GPU 或 CPU-GPU 混合平台. 本文分析 3 种计算模型中最优的实现算法, 在 CPU 和 GPU 端的向量化查询处理模型中, 连接和聚集操作作为一个整体计时, 在 GPU 加速模型中, 星形连接和向量索引的 PCIe 传输整体计时.

图 11 显示了 3 种计算模型平均查询时间的分段计时结果.

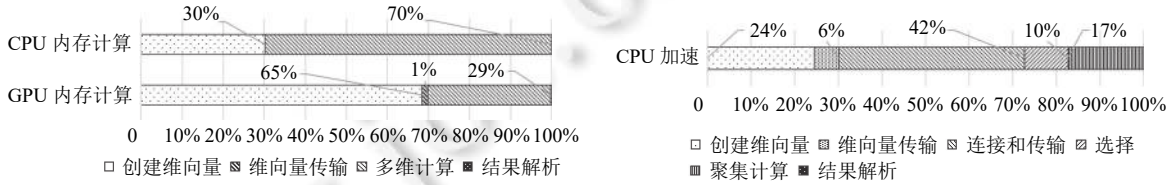

图 11 CPU 内存计算、GPU 内存计算和 GPU 加速模式分段计时

图 11 中 CPU 内存计算模型的维表元数据管理端的创建维向量执行时间占比约 30%, 向量化的连接及聚集计算执行时间占比约 70%, 查询代价主要集中在大事实表数据上的多表星形连接和分组聚集计算. GPU 内存计算模型中 GPU 端存储完整的事实表, 执行 GPU 上向量化的多表星形连接和分组聚集计算, 由于 GPU 强大的并行计算能力和显存的高带宽性能使 GPU 端的内存计算性能得到极大的提升, 平均查询时间是 CPU 内存计算模型的 42.6% (55 ms vs. 129 ms), 其中 GPU 端的计算时间是 CPU 内存计算模式该阶段执行时间的 20%. 维表元数据管理端的创建维向量执行时间在 CPU 内存计算模式下占比较低, 但在 GPU 内存计算模式下占比较高. 维表数据类型多样, 而且在维向量映射过程中需要通过多线程并发控制机制创建共享的 GROUP BY 分组字典表, 不适合 GPU 高并发计算特征, 而且多维元数据的维护包含通用的增、删、改操作, 需要完整的事务处理机制, 更适合 CPU 处理模型, 而事实表数据属于只读或只支持插入更新操作, 不需要复杂的数据管理机制, 更强调强大的计算能力, 适合 GPU 端处理.

图 11 中 GPU 加速模型中 CPU 端的维向量映射和分组聚集计算执行时间占比分别为 24% 和 17%, GPU 端的维向量传输、星形连接和向量索引传输时间占比为 48% (6%+42%). GPU 内存计算模型只需要传输较小的维向量, 而 GPU 加速模型还要额外传输较大的向量索引, PCIe 传输代价也相对较高, 但提高了 GPU 对大数据集计算加速的支持能力. 从 CPU 和 GPU 端的负载来看, 大约各占 50%, 因此在执行批量查询处理时可以充分利用 CPU 与 GPU 端独立的数据集和计算任务构建 CPU 和 GPU 设备间的流水并行处理模型, 使 Q_{i+1} 查询的维向量映射计算和 Q_{i-1} 查询的分组聚集计算任务在 CPU 端和 Q_i 的星形连接向量索引计算在 GPU 端并行处理, 提高 CPU 和 GPU 计算资源的利用率, 通过查询间的流水并行处理将查询执行时间缩短为原来的一半. 对于 CPU-GPU 混合平台的加速模型来说, 除提高 GPU 端计算性能外, 如何均衡 CPU 与 GPU 端的负载, 实现 CPU 与 GPU 设备间的并行处理, 提高计算资源利用率也是一个重要的研究课题. 本文的 GPU 加速模型实现了 CPU 与 GPU 端计算负载的相对平衡, 未来可以进一步探索 CPU-GPU 混合平台间的多查询并行处理.

从来未的硬件发展趋势来看, CPU 和 GPU 的性能持续提升, PCIe 4.0 和 NVLink 技术也持续提升 CPU-GPU 和 GPU-GPU 的数据传输性能. GPU 新的发展趋势是持续扩大显存容量, 最新的 NVIDIA A100 显存最大容量达到 80 GB[28], AMD Instinct MI250X 显存容量达到 128 GB[29], 专用 GPU 服务器配置 8–16 块 GPU 卡可以支持 1–2 TB

的 GPU 显存, 能够支持较大数据集上的高性能 GPU 内存 OLAP 查询处理.

综上所述, GPU 内存计算模型通过对事实表数据的 GPU 本地存储与计算能够获得最优的性能, 最大化 GPU 存储端的计算性能. GPU 加速模型可用于对内存数据库进行加速, 将 GPU 用作硬件级的向量索引存储和计算设备, 通过 GPU 本地存储和计算能力支持计算型的向量索引, 简化 OLAP 查询处理算法, 提高 OLAP 查询处理性能.

4.4 SSB 基准测试性能

BrytlytDB[30]和 OmniSciDB 是两种代表性的 GPU 数据库[31], OmniSciDB 是一种较高性能的开源数据库, 本文在实验中选择 OmniSciDB 作为 GPU 数据库的性能基准. PG-Strom+Arrow 是一个基于开源数据库 PostgreSQL 开发的 GPU 加速数据库, 通过开源的内存列存储引擎 Arrow 提高存储访问性能, 可以作为 GPU 加速数据库的性能基准. Hyper 是一个基于实时编译 (JIT) 技术的高性能内存数据库, Hyper 也被用在很多内存数据库的研究中作为基准[5,32]. MonetDB 是代表性的列存储内存数据库, Vector 是 MonetDB 的商业版, 其代表性的向量化查询处理技术已成为当前内存数据库的核心技术, 曾经在 TPC-H 榜单上持续排名第一. Hyper、 Vector、MonetDB 代表了当前内存数据库的 3 个主流技术 (JIT 实时编译, vector-at-a-time 向量化查询处理和 column-at-a-time 列式查询处理). OmniSciDB 有 CPU 版本, 支持主流的 JIT 实时编译和向量化查询处理技术, GPU 版本支持 GPU 存储本地化查询处理, 本文在测试中每个基准查询连续执行 3 次取最短时间作为查询最优性能, 评估各数据库系统在最理想状态下的性能上限.

图 12 显示了 GPU 内存计算 (GPU IM)、GPU 加速 (GPU Acce)、CPU 内存计算 (CPU-only) 和对比数据库在 SSB 数据集上的基准测试性能, 数据集 SF=100, 执行时间为平均查询时间, 性能加速比以 GPU 内存计算为基准 (1). 在内存数据库系统中, Hyper 是 OmniSciDB CPU 性能的 2.53 倍, 但仅为 OmniSciDB GPU 版本的 0.34 倍, OmniSciDB GPU 版本是 CPU 版本性能的 4.75 倍, OmniSciDB GPU 版本是 Vector、MonetDB、PG-Strom+Arrow 数据库性能的 10.58 倍、11.06 倍和 41.8 倍, 显示了 GPU 数据库的性能优势. 实验中仅使用了一块 V100 GPU 卡, 当使用多块 GPU 卡时, 支持的数据量及性能将得到进一步提升, 也将进一步显示 GPU 数据库的性能优势.

3 种基于向量索引的 OLAP 计算模型在 SSB 基准测试的性能上优于 Hyper、OmniSciDB、Vector、MonetDB 和 PG-Strom+Arrow, 其中最优的 GPU 内存计算模型性能为 OmniSciDB GPU 版本的 3.1 倍, CPU 内存计算模型是内存数据库 Hyper 的 1.79 倍, 是 OmniSciDB CPU 版本的 6.33 倍, GPU 内存计算模型达到了 CPU 内存计算模型的 2.4 倍, 显示了相同算法结构下 GPU 计算性能的优势. GPU 加速模型性能略低于 CPU 内存计算模型, 但可以通过扩展多 GPU 卡进一步提升性能, 在多查询 CPU 和 GPU 端流水并行处理模式下理想的性能预期达到 CPU 内存计算模型的 1.6 倍.

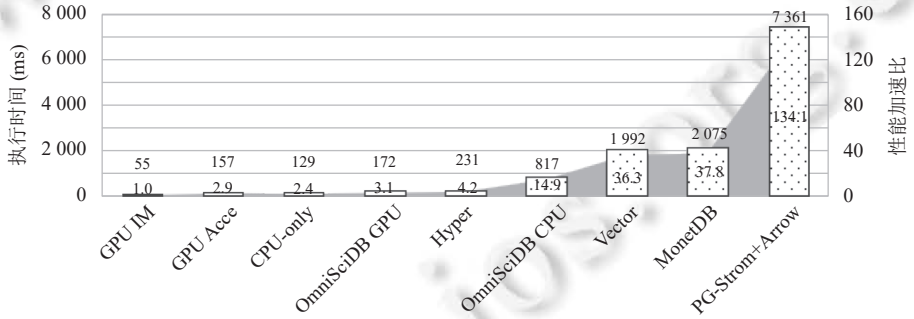

图 12 SSB 基准性能

图 13 显示了 3 种 OLAP 计算模型各阶段执行时间 (ms) 明细, 我们可以进一步分析 CPU 和 GPU 端的计算负载情况. CPU 内存计算模型维表上的处理时间较短, 主要负载集中在事实表上的连接和聚集计算, 当选择率降低时查询执行时间也显著降低. GPU 内存计算模型在 GPU 存储的事实表数据上执行的连接和分组聚集操作执行时间较短, 而在 CPU 端维表处理时间占比较高, GPU 端查询处理时间对选择率的敏感程度低于 CPU 内存计算模型. GPU 加速模型对选择率较敏感, 当选择率高时 PCIe 通道需要传输较大的压缩向量索引, 由于 PCIe 通道带宽性能远低于内存和 GPU 显存, 因此对查询整体性能造成较大的影响, 但另一方面, CPU 与 GPU 端计算负载较为均衡, 存在着通过多查询流水并行方式进一步优化查询性能的空间.

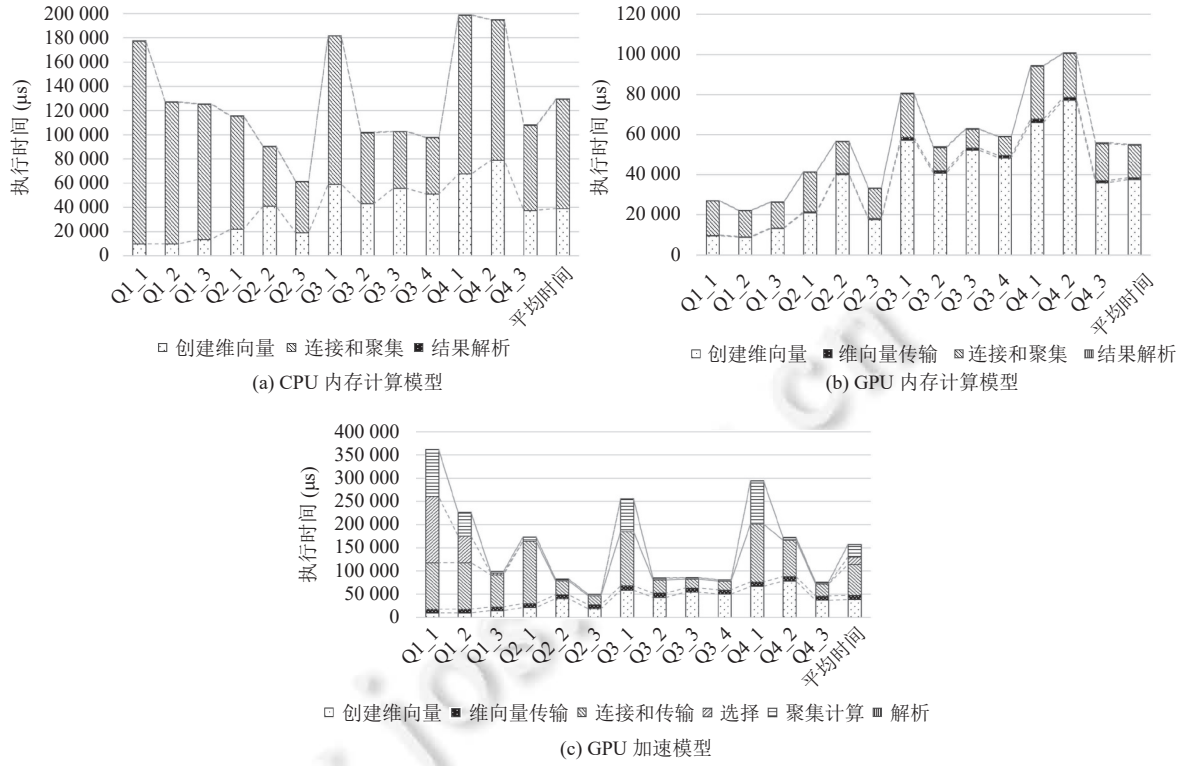

图 13 查询时间分解

4.5 讨 论

通过实验我们观察到以下现象.

● 典型的 GPU 数据库需要将数据从 CPU 内存传输到 GPU 端进行计算, 优化策略需要权衡 CPU 端计算性能和 PCIe 传输代价, 一些研究可以优化传输代价[33], 但是收益并不明显, 当 CPU 端计算性能较优时, PCIe 传输延迟会降低整体性能. 表 2 显示了 Hyper、CPU 内存计算模型和 OmniSciDB CPU 版本的平均查询时间, 并对应计算出在 PCIe3.0、PCIe4.0 和 NVLink 通道下查询时间内可以传输的列的数量. 在当前 PCIe3.0 和下一代 PCIe4.0 传输技术下, 性能较高的 Hyper、CPU 内存计算模型平均查询执行时间内仅能传输 3 个以下的列数据, 数据从 CPU 内存移动到 GPU 显存计算没有性能收益, 而 OmniSciDB CPU 的查询执行时间较长, 在 CPU 端查询执行时间内可以传输足够的数据, 通过 GPU 端计算性能的收益存在通过 GPU 加速优化 CPU 查询处理性能的可能性. 当数据通道性能足够高时, 如 NVLink 在 CPU 端查询执行时间内可以传输超过查询所需要的数据, 存在较大的 GPU 性能加速空间. 当前 NVLink 主要用于 GPU-GPU 之间的高速数据传输, 只有 IBM 的 Power9 在 CPU 和 GPU 之间采用了 NVLink 技术. 因此, GPU 数据库的技术路线与 GPU 和数据通道硬件技术存在着紧密的联系, 其优化策略也受硬件技术水平的影响而发生改变. 在当前硬件技术阶段, 我们不推荐基于实时数据传输策略的 GPU 数据库实现技术, 其性能可能低于优化的内存数据库性能, 面向未来大容量显存的 GPU 内存计算模型和基于数据分布策略 GPU 加速模型能够更好地提高 GPU 存储访问和计算的局部性.

表 2 传输与 GPU 计算并行优化策略分析

系统	平均查询时间 (ms)	PCIe3.0 (16 GB/s)下的列传输数量	PCIe4.0 (32 GB/s)下的列传输数量	NVLink (300 GB/s)下的列传输数量
Hyper	231	2	3	31
CPU内存计算	129	1	2	17
OmniSciDB CPU	817	6	12	110

● GPU 加速模型将事实表外键存储于 GPU 显存, 执行查询中的星表连接操作生成向量索引, 加速 OLAP 查询中执行时间最长的多表连接操作. 基于向量索引技术, 我们可以把 GPU 作为一个向量索引的硬件级存储和计算平台, 根据查询内容动态生成向量索引, 加速大事实表上的访问和聚集计算. 对于数据库系统而言, 可以通过增加类似 create GPU vector index…之类的命令创建 GPU 端向量索引, 将 OLAP 数据集上的复杂查询处理任务在 GPU 向量索引的支持下转换为简单的事实表上基于向量索引的聚集计算任务, 提高数据库的查询性能, 为数据库提供插件化的 GPU 硬件索引加速器.

● CPU 与 GPU 硬件结构的差异也显示了相同的查询处理模型在不同平台的性能特征. CPU 平台上向量化查询处理模型优于列式查询处理模型, 优于行式查询处理模型, 而 GPU 平台上行式查询处理模型与向量化查询处理模型的性能非常接近, 优于列式查询处理模型. 主要原因是 GPU 线程较小的向量长度难以发挥 CPU 平台较大向量长度对中间结果的优化效率, 列式查询处理模型中向量索引的内存物化对性能影响大于 CPU 平台. 因此, GPU 平台上的查询优化相对于 CPU 平台更强调通过流水线处理消除物化结果的优化策略, 算法效率在 GPU 大量并发线程的支持下对性能的影响降低. 同时, 行式处理模型和向量化处理模型可以消除向量索引在 GPU 显存的存储空间开销, 提高 GPU 显存的利用率.

● 当前 GPU 数据库在数据分布和存储策略上还不十分明确, 如 OmniSciDB 强调将热数据集缓存到 GPU 显存端提高 GPU 本地存储访问性能, 主要体现在缓存技术的优化策略上, 没有面向 OLAP 模式特征明确的优化策略. 本文提出的面向 OLAP 负载特征的数据分布策略是从 OLAP 模式特点、计算特点出发设计的静态数据分布策略, 通过模式定义优化的 CPU-GPU 分布存储和分布计算策略, 通过模式定义计算, 对硬件支持的数据量、最优硬件配置等都可以做出明确的需求.

5 相关工作

在 TOP500 和 GREEN500 榜单上 GPU 是主流的高性能计算平台. GPU 数据库也被认为是下一代高性能数据库的代表性技术[31]. GPU 数据库的算法与实现技术研究已持续多年, 主要思想是设计 GPU 硬件敏感的 GPU 数据库算法优化数据库算子性能[34], 设计适合 GPU 的查询处理模型以及优化面向 CPU-GPU 异构计算平台的查询处理模型, 主要技术路线区别是将 GPU 加速数据库还是 GPU 内存数据库.

哈希连接算法是近年研究较为深入的领域, 也是数据库性能最关键的算法之一. 主要的研究集中在两种哈希连接算法: 无分区哈希连接算法和 radix 分区哈希连接算法[22−24,35−38]. 当前主要的结论是 radix 分区哈希连接算法需要通过硬件特性参数进行优化设计, 其性能优于不考虑硬件结构的无分区哈希连接算法. 除算法优化技术外, 基于 MOLAP 模型的 AIR (array index referencing) 算法实现了将哈希连接转换为数组地址映射的实现技术, 在多核 CPU、Phi 和 GPU 平台证明其性能优于两种代表性的哈希连接算法. 除性能因素外, radix 分区哈希连接算法虽然性能较优, 但其分区过程需要额外的内存空间消耗, 使其在 Phi 和 GPU 等硬件加速器上的存储效率降低, 只能处理较小的数据集, 降低了硬件加速器的使用效率. 进一步的研究[5]表明, radix 分区哈希连接算法并不适合 OLAP 的多表连接场景, 无法执行流水线查询处理模型, 在通用数据库系统中使用 radix 分区哈希连接算法在 TPC-H 基准测试中也难以取得预期的性能收益[32].

GPU 数据库主要用于 OLAP 场景, 当前 GPU 数据库系统通过设计一个 GPU 版本的数据库[39], 通过 GPU 优化的算法提升数据库的分析处理性能. 另一个技术路线是为 GPU 定制专用的 OLAP 加速引擎而不是通用的数据库引擎, 从而最大化 GPU 的 OLAP 查询处理性能, 降低 GPU OLAP 引擎设计的复杂度. 本文在前期工作中设计了一种基于数组存储模型和数组计算的 OLAP 计算模型[16], 通过数组地址访问和多维数组聚集提高 OLAP 性能, 在 SSB 基准测试中性能优于当时最优的内存数据库 Hyper、Vectorwise 和 MonetDB. 进一步研究在融合 ROLAP 和 MOLAP 模型的 Fusion OLAP 模型[18], 通过在关系存储模型上的多维计算模型实现 ROLAP 模型较高的存储效率和 MOLAP 模型较高的性能, 同时通过 Fusion OLAP 模型的分层技术模拟了多维计算通过 Intel Xeon Phi 和 GPU 加速后为数据库性能带来的提升.

OmniSciDB 重要的优化策略是 GPU 端存储热数据集上的本地化计算, 文献 [5] 也建议数据集直接存储在

GPU 端的计算模式. 从 GPU 显存容量持续增长的硬件发展趋势来看, GPU 内存计算是一个重要的技术路线. 因此, GPU 数据库研究的基本假设从 GPU 缓存优化向 GPU 内存计算优化转变, 在 GPU 算法优化技术上不仅要重视性能, 还要重视算法的 GPU 内存利用效率问题.

内存数据库的向量化查询处理技术是一种重要的优化手段, GPU 数据库主要采用一次一操作符的执行方式, 需要 GPU 显存存储中间结果. GPU 端流水处理技术[40]在操作符之间设计了流水处理模型来提高多查询处理时 GPU 硬件资源的利用率. GPU 查询处理模型研究也扩展到向量化查询优化技术[5], 通过 GPU 的共享内存缓存中间结果来降低查询中间结果的物化代价, 但由于 CPU 与 GPU 在硬件结构和线程管理方面的差异, 其优化技术有显著的不同.

本文设计了基于向量索引的 OLAP 查询处理技术, 设计了 CPU 内存计算、GPU 内存计算和 GPU 加速 3 种不同的查询处理模型, 对应不同的应用场景, 探索 GPU 数据库面向 OLAP 负载的优化技术方案.

6 总 结

本文研究了基于向量索引的 OLAP 实现技术, 实现并对比了 CPU 内存计算、GPU 内存计算和 GPU 加速 3 种查询处理模型, 并通过 SSB 基准测试展示了不同查询处理模型的性能特征. GPU 内存计算模型展现了显著的性能优势, 相对主流的内存数据库和 GPU 数据库 GPU 加速模型和 CPU 内存计算模型也有较高的性能. GPU 内存计算模型充分发挥了 GPU 强大的并行计算能力和 HBM 存储访问性能, 在未来的大容量显存 GPU 平台可以提供强大的 OLAP 查询处理性能. GPU 加速模型更为灵活, 可以通过有限的硬件下加速 OLAP 关键算子, 为数据库系统提供硬件级的向量索引计算能力. 在查询处理模型研究上, 实现了 GPU 端事实表数据上的多表星形连接和分组聚集操作的向量化查询处理技术, 通过 GPU 的共享内存缓存查询中间结果, 减少对 GPU 显存的消耗和访问代价, 提高 GPU 端的计算性能和 GPU 显存利用效率. 研究发现, GPU 端简单的行处理模型和向量化查询处理模型在性能上较为接近, 可以通过简单的算法实现较高的性能, 这种特征与 CPU 平台不同. 对于实际的 OLAP 应用来说, GPU 数据库的优化策略受较多因素的影响, 如模式特点、负载特征、性能需求、数据集大小、GPU 显存大小、系统开发成本等, 需要灵活的 OLAP 计算框架满足不同的需求. 本文设计的 GPU 内存计算和 GPU 加速模型为 GPU 数据库提供了两种技术路线, 满足不同硬件配置、数据集大小的性能加速需求.

本文实现了基于数据仓库典型星形模型的 OLAP 实现技术, 目前主要支持单事实表结构上的高性能多维计算, 对于维表上的雪花模型同样支持 (维映射阶段对较小的维雪花结构进行映射), 对多事实表结构目前还不支持 (如 TPC-H), 我们将在未来的工作中继续探索对 TPC-H 复杂模式的 OLAP 优化技术.

References:

[1] NVIDIA. NVIDIA A100 tensor core GPU. 2022. https://www.nvidia.cn/data-center/a100/

[2] Funke H, Breß S, Noll S, Markl V, Teubner J. Pipelined query processing in coprocessor environments. In: Proc. of the 2018 Int'l Conf. on Management of Data. Houston: ACM, 2018. 1603–1618. [doi: 10.1145/3183713.3183734]

[3] Yuan Y, Lee R, Zhang XD. The Yin and Yang of processing data warehousing queries on GPU devices. Proc. of the VLDB Endowment, 2013, 6(10): 817–828. [doi: 10.14778/2536206.2536210]

[4] HeavyDB. Open source analytical database & SQL engine. 2022. https://www.heavy.ai/product/heavydb

[5] 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]

[6] Breß S. The design and implementation of CoGaDB: A column-oriented GPU-accelerated DBMS. Datenbank-Spektrum, 2014, 14(3): 199–209. [doi: 10.1007/s13222-014-0164-z]

[7] Funke H, Teubner J. Data-parallel query processing on non-uniform data. Proc. of the VLDB Endowment, 2020, 13(6): 884–897. [doi: 10.14778/3380750.3380758]

[8] Karnagel T, Mueller R, Lohman GM. Optimizing GPU-accelerated group-by and aggregation. In: Proc. of the 6th Int'l Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures. 2015. 13–24.

[9] Li J, Tseng HW, Lin CB, Papakonstantinou Y, Swanson S. HippogriffDB: Balancing I/O and GPU bandwidth in big data analytics. Proc. of the VLDB Endowment, 2016, 9(14): 1647–1658. [doi: 10.14778/3007328.3007331]

[10] Paul J, He BS, Lu SL, Lau CT. Improving execution efficiency of just-in-time compilation based query processing on GPUs. Proc. of the VLDB Endowment, 2020, 14(2): 202–214. [doi: 10.14778/3425879.3425890]

[11] Kemper A, Neumann T. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In: Proc. of the 27th IEEE Int'l Conf. on Data Engineering. Hannover: IEEE, 2011. 195–206. [doi: 10.1109/ICDE.2011.5767867]

[12] ACTIAN. Scalable, secure and flexible analytics database. 2022. https://www.actian.com/analytic-database/vector-analytic-database/

[13] Idreos S, Groffen F, Nes N, Manegold S, Mullender S, Martin K. MonetDB: Two decades of research in column-oriented database architectures. Bulletin of the Technical Committee on Data Engineering, 2012, 35(1): 40–45.

[14] KaiGai. SSDtoGPU direct SQL on columnar-store (Apache arrow). 2019. https://kaigai.hatenablog.com/entry/2019/05/01/004618

[15] 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]

[16] 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]

[17] Zhang Y, Zhang YS, Chen H, Wang S. GPU adaptive hybrid OLAP query processing model. Ruan Jian Xue Bao/Journal of Software, 2016, 27(5): 1246−1265 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4828.htm [doi: 10.13328/j.cnki.jos.004828]

[18] Zhang YS, Zhang Y, Wang S, Lu JJ. 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]

[19] 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]

[20] Neumann T. Efficiently compiling efficient query plans for modern hardware. Proc. of the VLDB Endowment, 2011, 4(9): 539–550. [doi: 10.14778/2002938.2002940]

[21] Lee R, Zhou MH, Li C, Hu SG, Teng JP, Li DY, Zhang XD. The art of balance: A RateupDB™ experience of building a CPU/GPU hybrid database product. Proc. of the VLDB Endowment, 2021, 14(12): 2999–3013. [doi: 10.14778/3476311.3476378]

[22] 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. Brisbane: IEEE, 2013. 362–373. [doi: 10.1109/ICDE.2013.6544839]

[23] 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]

[24] 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]

[25] Chavan S, Hopeman A, Lee S, Liu 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]

[26] 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]

[27] Zukowski M, Boncz PA, Nes NJ, Héman S. MonetDB/X100—A DBMS in the CPU Cache. IEEE Data Engineering Bulletin, 2005, 28(2): 17–22.

[28] NVIDIA. NVIDIA A100 80GB PCIe GPU product brief. 2022. https://www.nvidia.cn/content/dam/en-zz/Solutions/Data-Center/a100/pdf/PB-10577-001_v02.pdf

[29] AMD. AMD instinct MI250X accelerator. 2022. https://www.amd.com/en/products/server-accelerators/instinct-mi250x

[30] Brytlyt. The world's fastest analytics database. 2022. https://brytlyt.io/brytlyt-platform/database/

[31] Mark Litwintschik. Summary of the 1.1 billion taxi rides benchmarks. 2022. https://tech.marksblogg.com/benchmarks.html

[32] 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]

[33] Fang WB, He BS, Luo Q. Database compression on graphics processors. Proc. of the VLDB Endowment, 2010, 3(1–2): 670–680. [doi: 10.14778/1920841.1920927]

[34] Lahiri T, Chavan S, Colgan M, *et al*. Oracle database in-memory: A dual format in-memory database. In: Proc. of the 31st IEEE Int'l Conf. on Data Engineering (ICDE). Macao: IEEE, 2015. 1253–1258. [doi: 10.1109/ICDE.2015.7113373]

[35] 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]

[36] 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]

[37] Paul J, He BS, Lu SL, Lau CT. Revisiting hash join on graphics processors: A decade later. Distributed and Parallel Databases, 2020, 38(4): 771–793. [doi: 10.1007/s10619-019-07280-z]

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

[39] Raza A, Chrysogelos P, Sioulas P, Indjic V, Anadiotis AC, Ailamaki A. GPU-accelerated data management under the test of time. In: Proc. of the 10th Annual Conf. on Innovative Data Systems Research. Amsterdam, 2020.

[40] Paul J, He B, He BS. GPL: A GPU-based pipelined query processing engine. In: Proc. of the 2016 Int'l Conf. on Management of Data. San Francisco: ACM, 2016. 1935–1950. [doi: 10.1145/2882903.2915224]

附中文参考文献:

[17] 张宇, 张延松, 陈红, 王珊. 一种适应GPU的混合OLAP查询处理模型. 软件学报, 2016, 27(5): 1246–1265. http://www.jos.org.cn/1000-9825/4828.htm [doi: 10.13328/j.cnki.jos.004828]

[19] 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]

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.

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

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, 主要研究领域为数据库, 数据仓库, 大数据管理.

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