软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2021,32(3):859−885 [doi: 10.13328/j.cnki.jos.006175]

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

GPU 数据库核心技术综述*

裴 威[1,2], 李战怀[1], 潘 巍[1]

[1](西北工业大学 计算机学院,陕西 西安 710129)

[2](锦州医科大学,辽宁 锦州 121001)

通讯作者: 李战怀, E-mail: lizhh@nwpu.edu.cn

摘 要: GPU 以其超高速计算能力和超大数据处理带宽受到数据库厂商及研究人员的青睐,以 GPU 计算为核心的数据库分支(GDBMS)蓬勃发展,以其吞吐量大、响应时间短、成本低廉、易于扩展的特点,与人工智能、时空数据分析、数据可视化、商务智能交互融合能力,彻底改变了数据分析领域的格局.将对 GDBMS 的四大核心组件——查询编译器、查询处理器、查询优化器和存储管理器进行综述,希望促进未来的 GDBMS 研究和商业应用.

关键词: GPU;数据库;查询编译器;查询处理器;查询优化器;存储管理器

中图法分类号: TP311

中文引用格式: 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

英文引用格式: 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). http://www.jos.org.cn/1000-9825/6175.htm

Survey of Key Technologies in GPU Database System

PEI Wei[1,2], LI Zhan-Huai[1], PAN Wei[1]

[1](School of Computer Science, Northwestern Polytechnical University, Xi'an 710129, China)

[2](Jinzhou Medical University, Jinzhou 121001, China)

Abstract: In recent years, GPU is favored by database manufacturers and researchers for its ultra-high-speed computing capacity and huge data processing bandwidth. The database branch—GPU accelerating database or GPU database (GDBMS) is developing vigorously. With the characteristics of high throughput, low response time, high cost performance, and easy to expand, integrated with artificial intelligence (AI), business intelligence (BI), spatial-temporal data analysis, data visualization, GDBMS have the potential to change the world pattern of data analysis field. This study surveys the four core components of GDBMS: query compiler, query processor, query optimizer, and storage manager, hoping to promote the future research and commercial application of GDBMS.

Key words: GPU; database; query compiler; query processor; query optimizer; storage manager

近年来,GPU 已经从专业的图形处理器发展成为通用的计算处理器(general-purpose computing on graphics processing units,简称 GPGPU)[1],其超高速计算能力和超大数据处理带宽受到数据库厂商及研究人员的青睐.一方面,传统以 CPU 为计算中心的数据库技术面临"能耗墙""内存墙"的限制,单核 CPU 的性能提升已经接近物理极限,不得不借助并行计算、分布式技术来提升系统整体性能;另一方面,通用 GPU 每 10 年提升 1 000 倍的性能增长(Jensen'Law),使 GPU 在满足当今大数据需求方面具有得天独厚的优势.

* 基金项目: 国家自然科学基金(61732014, 61672432); 中央高校基本科研业务费专项资金(3102019DX1004)

Foundation item: National Natural Science Foundation of China (61732014, 61672432); Fundamental Research Funds for the Central Universities (3102019DX1004)

本文由"支撑人工智能的数据管理与分析技术"专刊特约编辑陈雷教授、王宏志教授、童咏昕教授、高宏教授推荐.

收稿时间: 2020-07-19; 修改时间: 2020-09-22; 采用时间: 2020-11-06; jos 在线出版时间: 2021-01-21

由于受能耗墙(“heat wall”)限制[2],简单地增加 CPU 的主频来获得性能提升走到了尽头.早在 2004 年,45 纳米制程下,CPU 的主频就可达到 3GHz;但时至今日,主流 CPU 制程已达到 14 纳米~7 纳米,主频却仍然在 3GHz~4GHz 左右(除了 2017 年 IBM z14 CPU 主频曾达到 5GHz)[3].与此同时,以超流水线、乱序执行、微指令(uops)技术在提升 CPU 每时钟周期内执行指令数上也收效甚微.比如将 CISC 指令变为 RISC 类似的微指令(uops)来提高并发度的方法,从 1995 年~2013 年,近 20 年间仅仅把最高并发微指令数从 3 提高到 4[3].尽管使用多核 CPU 一定程度上缓解了 CPU 性能提升的瓶颈,每年继续保持 15%~20%的增长.然而,CPU 已经无法跟上多源异构数据的爆炸性增长.

GPU 与 CPU 具有不同的体系结构和处理模式,将更多的片上空间用于计算单元,控制单元和缓存单元相对很少,使得单块 GPU 上拥有数千个并发计算核心.因此,在同样的主频下,GPU 能够取得更高的并发计算能力,也使 GPU 具有满足大数据处理需求的巨大潜能.与 CPU 不同,GPU 每两年取得性能上的突破,保持了近似摩尔定律的性能增长:以英伟达公司当年最顶级显卡的单精度浮点数计算峰值为例,2013 年,Titan Black 为 5 TFLOPS(每秒运行 5 兆次单精度浮点数计算指令);2015 年,Titan X 达到 7 TFLOPS;2017 年,Titan V 为 15 TFLOPS;2020 年,RTX 3090 达到 36 TFLOPS.与基于 CPU 的分析技术相比,基于 GPU 的数据库可实现数量级的加速,并具有更高的性价比.同时,由于单个 GPU 包含大量计算能力,因此扩展 GPU 数据库仅需要向服务器添加更多 GPU,而不是添加更多服务器.在时空数据 OLAP 分析任务和结果的可视化展示方面,几十个 GPU 的 GPU 数据库可以取得近千台普通 CPU 服务的数据库处理能力,而其响应时间甚至更短.比如:Tesla V100 GPU 提供 120 TFLOPS 的计算力,8 块互联即可以顶上 160 台双 CPU 的服务器!GPU 极大地加速了可以并行化的操作,即使数据集增长到数百万或数十亿条记录,GPU 数据库也可以在毫秒内返回复杂的查询.

除了极高的性能之外,GPU 数据库还在功能上日趋完善:基于商务智能 BI 的可视化交互式图表查询界面,让数据查询和探索更人性化;同时支持标准 SQL 查询语言,相较于其他大数据 OLAP 分析平台,分析周期大大缩短,往往只要几分钟就可以完成以往数天乃至一周的工作;时空数据分析功能,让基于位置的数据分析更高效、及时;GoAI(GPU open analytics initiative,GPU 开放分析计划)产业联盟和 GPU 数据帧(GPU data frame,GDF)构建了数据库库和人工智能应用程序之间数据交换标准,使数据科学家可以停留在 GPU 上的同时探索数据,训练机器学习算法并构建人工智能应用程序.现在,我们可以使用 GPU 数据库分析每周数十亿次的电视观看记录,以做出广告决策;根据移动定位推进时空思考、计算和工程设计,从自然灾害到能源可持续性,解决全球挑战,都需要了解现象在时间和空间上的联系.Covid-19 疫情以来,接触者追踪为 GPU 数据库带来新的挑战和机遇,通过将人口统计数据与运动数据集成在一起,并在地理和时间多个维度聚合数据,人们可以及时获得了疫情第一手资料,做出防疫决策.

纵观整个 GPU 数据库领域,不管是开源系统还是商业系统,其核心研究内容主要集中在查询编译器(query compilation)、查询处理器(query processing)、查询优化器(query optimizer)和存储管理器(memory management)等 4 大部件:查询编译器要将用户的 SQL 语言描述的查询需求转化为 CPU- GPU 异构计算环境下的查询计划;查询处理器需要应用 GPU 并行计算技术实现关系型数据处理的各种算子,挖掘 GPU 峰值计算能力;查询优化器利用机器学习乃至人工智能技术,结合 CPU-GPU 异构计算环境和查询计划特点以及数据分布特征,生成最优(次优)的异构查询计划;存储管理器需要在异构数据存储管理、数据存储格式、数据压缩、数据访问等问题上做出决策.本文余下部分将依次对 GPU 数据库的 4 大部件的关键技术进行综述.

1 GPU 数据库分类与层次

GDBMS(GPU database management system or GPU accelerated database management system),顾名思义,就是使用 GPU 进行或加速数据增删改查的数据库管理系统.从 GDBMS 的系统形态上,本文将其分为研究原型(R-GDBMS: for research)和商用系统(C-GDBMS: for commercial)两大类,如图 1 所示.

商用 GDBMS 中,进一步可以分为 3 类.

- 第 1 类是支持 GPU 计算的传统数据库.这类 GDBMS 在已有传统数据库基础之上,将特定算子部署到

GPU 上用以加速的查询处理,包括以 PostgreSQL 为基础的 PG-Storm 系统和 DB2 的扩展模块 DB2 BLU.这类数据库与现有数据库集成度高、周边工具完善、且具有一定的与 OLTP 系统集成的能力;

- 第 2 类是非内存型 GDBMS,使用 GPU 完成全部或者大部分数据库关系运算,可以分析超过 10TB 的数据量,包括 SQream 和 BlazingDB;
- 第 3 类是内存型 GDBMS,该类系统将数据全部驻留内存,以发挥 GPU 的全部潜在性能、提高数据处理速度,可以处理 1TB~10TB 的较小数据集,包括 OmniSci(原 MapD),Kinetica(原 GPUDB),MegaWise,Brytlyt.

一般来说,后两类 GDBMS 由于专为 GPU 计算设计,没有传统数据库中束缚性能的遗留冗余模块,所以性能更高.

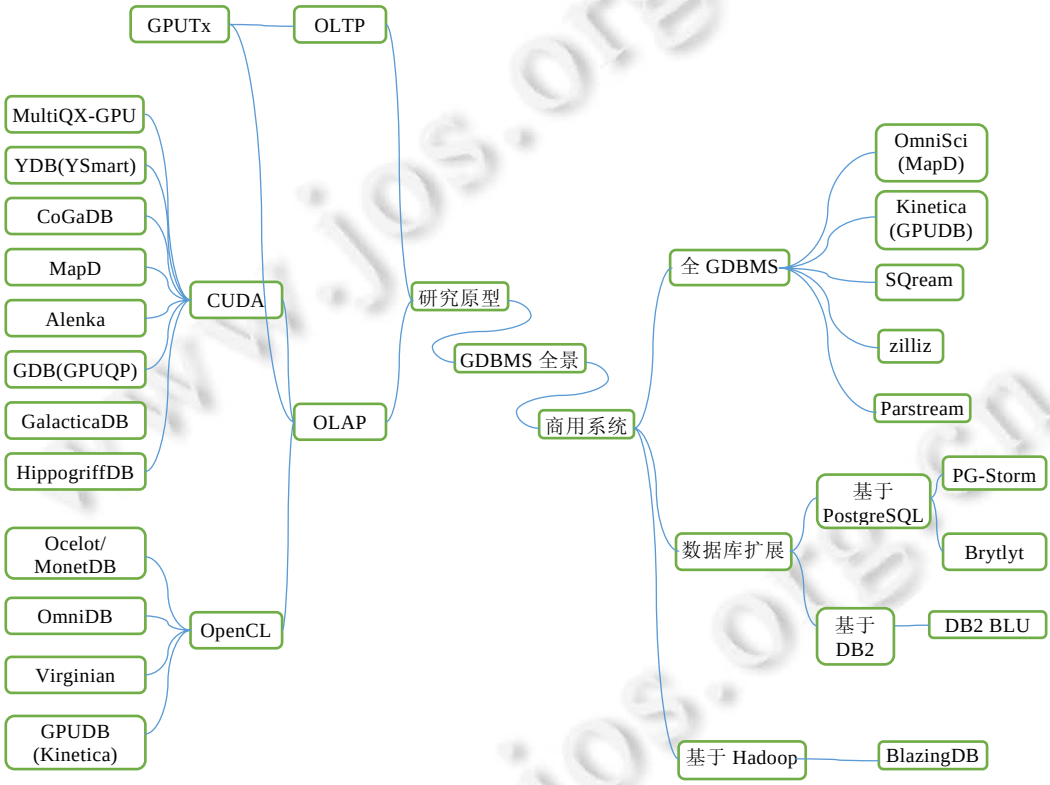

Fig.1 GDBMS landscape

图 1 GDBMS 系统全景图

目前,商用 GDBMS 普遍比基于 CPU 的解决方案在性能上有几个数量级的提升.比如:根据 OmniSci(MapD)最新的白皮书介绍[4],在十亿行出租车时空数据查询分析中,性能超过 PostgreSQL 达 722 倍~7 238 倍,更是比 Spark 快 1 884 倍~43 000 倍.在应用场景上,商用 GDBMS 以超高速端到端的时空数据分析和数据可视化为特色,结合机器学习与 AI 系统(H2O.ai 等),以一台普通服务器配备多个 GPU 显卡就能取得分布式大数据系统一个集群的处理能力.超大吞吐量、超低时延以及更低的成本,让 GDBMS 在 OLAP 业务上优势明显.

研究原型 GDBMS 将重点放在了全 GPU 计算上,研究数据可以在 GPU 显存上存储的情况下,GPU 加速所能获得的最高性能.根据支持显卡厂商,可分为专用 GDBMS 和通用 GDBMS:前者因为使用 CUDA 而只能在 Nvidia 显卡上运行,包括 MapD(OmniSci)[5],CoGaDB[6−15],GDB(GPUQP)[16,17],MultiQX-GPU[18],YDB(YSmart)[19],Alenka[20],GalacticaDB[21],HippogriffDB[22],G-SDMS[23,24];后者使用 OpenCL,在 Nvidia 卡和 AMD 卡上都可以运

行,实现了一定程度的平台无关性,包括 GPUDB(Kinetica)[19],Ocelot[13],OmniDB[25],Virginian[26−28]这 4 个数据库.绝大多数 GDBMS 研究原型针对 OLAP 业务,只有 GPUTx[29]系统实现了部分 OLTP 事务功能.文献[30]总结了 GDBMS 设计上的诸多原则,提出了 GDBMS 的共有系统层次架构,如图 2 所示.该文将 GDBMS 分为 7 层,分别为 SQL 解析和逻辑优化层、物理优化层、算子层、数据访问层、数据并发原语层、数据管理策略层和数据存储层.该文中为了突出 GDBMS 与传统的基于硬盘的数据库和内存数据库两者的区别,特别将 GDBMS 独有的模块标注为深色.本文认为:GDBMS 作为一个独立的数据库分支,其设计是为了适应 GPU 并行计算的特点,因而传统数据库中与 GPU 计算不适应的模块没有必要存在.因此,本文将 GDBMS 分为查询编译器、查询执行器、查询优化器和存储管理器这 4 个核心模块.对比文献[30]的 7 层结构,我们将其中 SQL 解析和逻辑优化层归属于查询编译器;算子层、数据访问层和数据并发原语层是顶层功能和底层实现的关系,在逻辑上属于一个统一的整体,对应于本文的查询处理器;物理优化层对应于查询优化器;而数据管理策略层和数据存储层密不可分,也应该视为一个模块(存储管理器).在异构查询编译、CPU\GPU 异构计算任务调度、异构查询优化、代价模型构建、GPU 关系数据并发算法设计、显存-内存异构存储体系管理等方面,GDBMS 面对与传统 CPU 数据库完全不同的问题和挑战,需要重新设计或扩展原有的功能模块,以充分发挥 GPU 的计算性能优势.

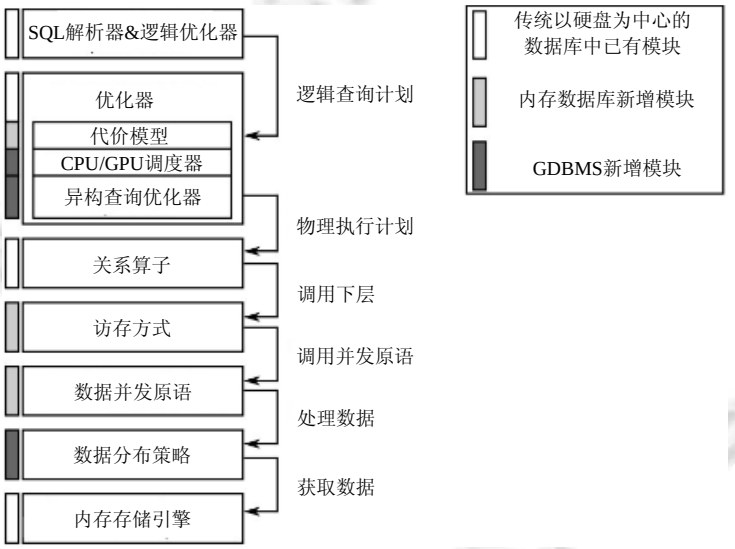

Fig.2 Layered architecture of GDBMS[30]

图 2 GDBMS 层次架构图[30]

2 查询编译器

数据库管理系统大都使用 SQL 或其他高层逻辑 API,而查询编译器作为将 SQL 转化为数据库内部执行逻辑并向用户返回结果的重要组件,处在 GDBMS 的前端,与查询处理器、查询优化器、存储管理器深度耦合协同工作,共同为用户提供查询服务.

传统数据库中,SQL 编译器的词法分析、大部分编译技术同样适用 GDBMS,所不同的是,GDBMS 需要应对的异构计算硬件环境和数据处理模式变化的双重挑战.

- 首先,查询编译器需要利用 CUDA\OpenCL 编译工具生成 GPU 可执行代码,同时要创新 SQL 编译模式,尽可能减少 SQL 编译的开销,使整体的性能最优;
- 其次,传统的“火山模型”不适合 GPU 计算,GDBMS 查询编译器面临着关系数据处理模式的变化,需要将向量化(vector-wise)、一次一算子(operator-at-a-time)和批量执行(bulk processing model)这 3 种策略结合起来.

- 向量化,原指将多个关系型数据以向量形式作为一个 SIMD(single instruction multiple data)指令的多个操作数来处理的方法,可以有效提高查询处理吞吐量.在 GPU 中,向量化思想可以用 GPU 的单指令多线程(single instruction multiple thread,简称 SIMT)进一步加大数据处理的吞吐量;
- “一次一算子”与“一次一数据”是数据处理的两种模式:前者就是先将所有数据同时完成第 1 个算子的处理,并保存中间结果,作为下一个算子的输入数据,直至全部处理完;而后者是指先让一条数据经过所有算子计算得出结果,再计算下一条数据的策略,是基于 CPU 计算常采用的策略;
- 批量执行就是尽可能一批处理更多的数据,通过提高单次处理数据的数量,弥补处理频次不高的缺点;

由于 GPU 由于编程模型和计算架构与 CPU 不同,GDBMS 借鉴了 CPU 计算中的向量化、“一次一算子”、批量执行这 3 种策略,并使之适应 GPU 大规模并发计算的特点,我们将在第 3.2 节详细介绍 GDBMS 编译器的数据处理模式.

2.1 GDBMS编译模型

传统数据库系统支持 SQL 作为查询语言,在大数据集合上进行 ad-hoc 查询.数据库系统直到运行时才能知道用户查询的语义信息,因此,DBMS 大多采用解释型的 SQL 前端,通过语法解析、语义解析模块将查询 query 解析为可为查询执行引擎使用的内部任务表示,即逻辑执行树.尽管传统的基于迭代的查询执行模型(即火山模型或 tuple-at-a-time)具有很高的灵活性,但是由于生成查询计划的过程必然经过多次虚函数的调用,这种在 CPU 环境中行之有效的编译方案,在 GPU 上执行效率却很低,不符合 GPU 大规模线程并发 SIMT 的编程范式,使查询编译消耗的时间日益成为高性能数据库查询处理时延的主要部分,进而使查询编译器处于数据库性能优化的关键路径.

为提高查询编译性能,GDBMS 采用解释型查询编译器,采用 JIT 即时编译技术,通过将常用的 query 子句预编译为可执行代码块,并在运行时组合调用,实践中可取得与编译原生代码几乎相同的性能.SQL 是非常适合 JIT 技术的语言,这方面技术解决方案也非常多.早在 1999 年,AT&T 实验室的 Daytona 数据管理系统(1999)支持 SQL 作为子集的高级查询语言(cymbal)[31],将高级语言编译为 C 语言,再在运行时将 C 编译为可执行代码模块.SQLite 使用类似虚拟机的技术,将 SQL 解释为虚拟机操作指令,而虚拟机指令是预先存储在硬件层面的,根据代码局部性原理可获得较高的执行效率.

因此,GDBMS 查询编译器采用 3 种不同的编译模型来解决异构环境下 SQL 编译难题,即 JIT 代码生成(并发原语)、SQL 虚拟机和适配器模式.

- 第一,MapD[5],CoGaDB[10]等系统使用 nvcc 编译工具链,基于底层虚拟机(LLVM)编译器框架来即时编译 SQL 代码,是目前 GDBMS 系统编译器实现的主流方法.该方法使用基于 LLVM[32]的 nvcc 编译器,将关系算子分为更小的原语 primitives,并把原语预编译为架构无关汇编代码 PTX,在运行时,由编译器只需完成 SQL 语言到算子的编译工作,而由查询执行器在优化器指导之下完成算子到并发原语的映射.这种方法通过预编译并发原语的方法,降低了实时编译 SQL 语言的工作负载,同时保留了交互式编译执行的灵活性,是 GDBMS 编译器的主流技术.此外,Google 公司推出了不同于 nvcc 的 gpucc[33]编译器.文献[34]提出了另外一种 CUDA 编译器,使用了核函数融合 kernel 的技术.借助对 CUDA 编译器的改进,预计将来,GDBMS 可以更好地使用 JIT 编译技术,进一步缩短 SQL 编译流程;
- 第二,在 SQL 虚拟机模式下[26−28],以 SQLite 为基础的 GDBMS 系统——Virginian 系统将 SQL 编译为操作码(opcode)作为中间表示,将整个 DB 系统视为一个虚拟机,Opcode 操作码对应的算子被预编译为 cudabin 可执行代码,直接发送到 GPU 端执行.虚拟机模式有效减少了运行时编译负载,让数据可以超过 GPU 显存容量而运行,缺点是所有算子只能在 GPU 上运行,缺少了基于并发原语方案中 CPU 和 GPU 一起执行查询的灵活性,进而在系统整体性能上略低于并发原语方案;同时,虚拟机模式放弃了 SQL 优化的机会,无法提供查询重写、基于代价的优化;
- 第三,在适配器模式主要是解决不同厂商 GPU 接口不兼容的问题.GPUDB 编译器直接使用代码生成

模块,将 SQL 直接编译为 CUDA 或 OpenCL 驱动能执行的代码,以算子为单位进行即时编译[19].适配器模式在运行时的编译负载会比较高,在提高了系统对显卡种类多样性的同时,牺牲了针对特定显卡的性能优化,需要结合查询并行、分布式计算等技术来提升性能.此外,为应对 GPU 硬件的多样性,尤其是为弥合 NVIDIA 显卡和 AMD 显卡两家处于竞争中的两种架构之间的不同,Ocelot[35]等系统使用 OpenCL 框架,避免为 GPU 不同架构分别编写代码造成的代码膨胀问题.

基于 LLVM 中间表示的 GPU 通用编译工具能够很好地隔离硬件多样性,做到编译各阶段彼此孤立,给 GDBMS 在编译的各个阶段进行优化提供了可能.未来,基于编译自动化工具的研究将极大提升 GDBMS 系统的性能.

2.2 GPU数据处理模型

数据库中,从数据处理模型来看,可分为 3 种:迭代模式(iteration)、批量模式(batching)或二者的混合.传统的 DBMS 往往采用一次一行的流式迭代模型,也就是著名的火山模型(volcano model)处理查询请求.时至今日,研究界和工业界提出了各种改进版的火山模型来规避其缺点,比如增加每次迭代的数据量、使用 SIMD 指令一次处理多个数据、推拉结合的数据获取方式等,目前仍然是数据库中的主流编译技术.批量模式是将每个查询编译为可执行代码,采用完全物化的方式处理所有数据.批量模式相较火山模型的迭代模式,在提高局部性、减少运行时解释开销、使用 SIMD 指令方面有很大优势,但在实现 ad-hoc 查询上,面临灵活度不够、物化存储空间要求过高的问题.因此,实践中将两者结合的方式更有优势,比如微批量化查询处理.该类方案使用不同的粒度作为数据处理的单元,仍然在逻辑上组织成树型结构,让数据自底向上流动完成查询操作,兼具迭代模型的灵活性和批处理的高吞吐量的优点.

GDBMS 普遍采用向量化一次一算子数据处理模式,并以此改造查询编译器.

- 首先,迭代模式并不适合 GDBMS,因为火山模型赖以存在的虚函数机制因为 GPU 缺乏对应的复杂逻辑控制模块,在 GPU 上不可实现或者引起严重的线程分支恶化问题.迭代模型的灵活性是“彼之蜜糖,我之毒药”,实际上会损害 GPU 的性能.GPU 的 SIMT 采用大规模线程并发的方式来提高数据处理的速度,批量执行可以有效降低生成计划的函数调用次数,将列数据细粒度分配给 GPU 线程,并用循环展开的方式,可有效减少控制指令总量,有效降低分支恶化的风险;
- 其次,列式处理更适合 GDBMS.一次一行的处理数据方式在代码上需要做大量的逻辑判断,而这正是 GPU 的劣势;一次一列来处理数据时,由于每列数据类型一致,可以用向量化方式处理,避免了分支判断劣化性能问题,更适合 GPU 计算.此外,有研究[36]证实:对于 OLAP 业务,按行为单位的处理模型即使行被合理分区并增加列索引等优化策略后,仍然不如列式处理高效.事实上,列式处理模型自 MonetDB[37]首次引入后,其后续系统 X100[38]将流水化(pipelining)引入列式处理模型中.GDBMS 系统普遍采用列式处理模型[30],比如 Ocelot[13],CoGaDB[10]等;
- 再次,由于 GPU 的大规模并行编程模型依赖于对数据的并行处理,很多算法想在 GPU 上运行必须适应单指令多线程(SIMT)的编程范式,所以需要对关系算子进行并行化改造,使得同一指令同时处理多个关系数据处理需求,充分利用 GPU 的并发编程优势.“一次一算子”的数据处理模式就是:让数据在 GPU 向量化算子间流动,每次采用完全物化的策略保存算子输出的中间结果,作为下一个算子的输入数据;
- 最后,为了降低物化代价,通过适当分区切分数据,可以使 GDBMS 兼具迭代模式的最大的优点——流水化处理数据的能力[39].为了加速数据处理以及利用合理分区数据,采用数据流水化处理(pipelining data processing),有效提高数据处理并行度.文献[40]通过细粒度划分数据,将处理整个列的算子切成更小的算子单元,在 GPU 上实现了相关算子间流水化处理数据.

3 查询处理器

GDBMS 查询处理引擎接受处理查询编译器输出的查询计划树 QEP(query execution plan)并执行查询返回结果,是利用 GPU-CPU 异构计算处理用户查询请求的核心模块.从功能角度来看,GDBMS 查询处理引擎面对的

核心问题就是如何利用 GPU 实现关系代数运算,即实现选择、投影、连接、聚合基本的关系算子,同时还需要实现的空间数据(geo-spatial data)处理、OLAP 聚合查询等功能复杂的算子,这是 GDBMS 查询处理引擎面临的功能挑战.在执行模式上,GPU 上执行的代码被称为 kernel 核函数,以核函数为基础的查询处理技术(KBE)是 GDBMS 查询处理引擎的必然选择.但是核函数的并发执行并不完全在程序的控制之下,如何在 GPU 高并发 SIMT 模式下以何种粒度切分关系查询树来最大化查询处理性能,是 GDBMS 面临的性能挑战.

面对查询功能挑战和性能挑战,GDBMS 采用了分而治之的策略,在逻辑查询树级别,用 KBE 融合或切分的策略,提升 GPU 的资源利用率和查询并发度;而在算子级别,则采用了设计专门针对 GPU 优化的算子的方法,即数据并发原语的方法.

3.1 GPU关系代数和并发原语

在 GPU 上实现选择、投影、连接等基本关系代数算子,是实现 GDBMS 数据库的基础.传统的数据库系统中,关系代数多由 CPU 算法实现,GDBMS 必须将关系算子用相应的 GPU 算法达到相同目的,而 GPU 算法在编程模型、并发执行、访存方式、控制逻辑上与 CPU 存在很大不同,这是制约 GDBMS 发展的难点之一.早期的 GDBMS 使用图形流式编程接口[1],采用图形接口(graphic API)来编写数据库内核,需要很深的计算机图形学基础,并且必须按照光线渲染流程编写代码,这严重制约了人们的创造力;而且专用显存(纹理显存等)容量小、访存方式复杂、读写保护机制严格,这些因素都严重制约了 GDBMS 处理大规模数据的能力.自 2006 年统一计算设备架构 CUDA(compute unified device architecture)和 OpenCL 异构计算框架相继推出之后,可用于高性能通用计算的 CPU-GPU 异构计算技术(GPGPU)被引入到 GDBMS 系统中来,并迅速占据主导地位.比如:GPUQP[16]系统早期采用图形化接口和通用 CUDA 共同完成数据库查询处理,但随后完成了向通用计算技术的转化.CUDA/OpenCL 赋予了程序员使用 C\C++代码控制 GPU 大规模并行环境的能力,简化了 GDBMS 开发的难度,促进了 GDBMS 的繁荣.

GDBMS 系统普遍借鉴了 GDB[17]分而治之的分层设计,将关系代数功能拆解为算子层(operator)和原语层(primitives)两部分,设计了一系列的适应 GPU 计算的数据并行原语(primitives),表 1 列出了大部分的 GPU 并发原语.在此基础之上,CoGaDB[10]通过调用 GPU 优化的高效数据结构库 Thrust 来实现相同的关系代数算法.这种使用 NVIDIA 官方程序库的方法具有性能高、升级成本低等优点,也是系统走向成熟的必经之路.此外,Diamos[41]等人于 2012 年提出了基于算法框架(algorithm skeleton)的处理行数据的关系代数原语实现,给我们提供了列数据之外的另一种解决方案.

Table 1 Data parallel primitives' description

表 1 并发原语功能说明

原语	功能简介	相关文献
1. Map 映射	对每个输入数据运行一个处理函数,根据条件过滤(predicate or filter),设置结果位.	[16,17]
2. Scatter 分散	按照特定序列散射输入	[43]
3. Gather 聚集	按顺序聚集数据	[43]
4. Reduce 归约	根据条件归约数据,得出聚合的结果,用于去掉不符合条件的数据.	[17]
5. Prefix Sum 前缀求和	求到指定位置的非零数据之和,用于计算非冲突的写入位置.	[17]
6. Split 划分	按划分函数将数据分区,用于 hash 表建立、排序等.	[16,17,42]
7. Sort 排序	排序	[16,17,44,45]
8. Scan 扫描	已废弃,类似于 Reduce	[17]
9. Product 乘积	两个元组每元素生成笛卡尔乘积对	[41]

采用并发原语机制具有如下优点[42]:首先,可以充分利用处理器间的通信机制,相比 CPU 解决方案获得 4 倍~10 倍的内存带宽提升;其次,并行原语在同步和分支负载很小,因此可以发挥 GPU 极限性能;再次,并行原语可以方便地扩展到大规模处理器上;最后,并行原语设计之初考虑到数据倾斜的影响,可以取得确定性的性能下限.比如:scatter 原语[43]通过分区散射操作,在每个 load-store 周期内处理一个分区的散射操作,增加了数据访问聚合读写,避免了随机读写模式下运行性能不可控的问题.很多算子映射成原语的单个或多个组合,能够充分利用 GPU 高并发计算能力.通过以上原语的排列组合,大部分简单的关系代数算子可以进行有效拆解.比如选择

算子可以用 filter 原语实现,同时,filter 原语是由 map 原语、prefix sum 原语和 scatter 原语实现的.

基于并发原语的算子实现方法可以在实现中充分利用 GPU 编程特点进行优化,如用写入位置不同解决并发冲突、合并访存方法、shared memory 优化、用计算隐藏数据传输时延、循环展开等技术,写出高效的 CUDA 程序.同时,并发原语策略也是一种分解合并问题的策略,采用了分层隔离的设计理念,未来还可根据 GPGPU 技术发展对原语进行升级而不影响上层应用.尽管如此,并发原语策略以增加算子处理步骤为代价,进而增加了物化中间结果的代价,存在一定的显存浪费.每个原语按需请求显存资源的方式给全局显存管理提出了挑战,增大了算子失败的概率.而算子一旦失败,会造成多个关联算子的级联失败,造成严重的性能问题.

3.2 复杂关系算子

相较于选择、投影、扫描等基本的关系代数算子,复杂的关系代数算子(join 连接算子、聚集函数)因其逻辑功能复杂,相对计算量大,更能发挥 GPU 大规模并发计算的优势,因此成为 GDBMS 优势所在.

3.2.1 Join 算子

Join 连接算子对于数据库来说至关重要,往往成为一个查询计划的性能瓶颈,是数据库查询优化的重点.Join 算子本身的计算量大,适合利用 GPU 大规模并发线程技术进行计算.文献[46]指出:尽管现有硬件(CPU)已经足够智能,对隐藏缓存失效、TLB 失配延迟做的足够好,但是针对特定硬件的优化仍然可以有效提升 join 算子的性能.文献[42]采用数据并发原语机制(scan,scatter,split)来实现最常用的 join 算子:有和无索引的 Nest Loop Join、归并连接 Merge Join 以及哈希连接 Hash Join.其后续研究成果[17]用 CUDA 重写了数据并发原语,并将 4 种 join 算法集成到 GDB 系统中.实验表明:对于 JOIN 算子,GPU 实现性能可以提升 2 倍~20 倍.而另一个 GDBMS 系统 CoGaDB[10]则将 join 算法实现为 3 种模式:通用 join 算子、主键-外键连接(适用于事实表和维表的连接)、预取 join(使用 invisible-join[36]算法).文献[47]在 Virginian 系统上实现了多表连接算子,将查询处理引擎视为执行 SQL 查询的虚拟机,使用代码生成技术将多表连接查询编译为 op 操作码,由 SQL 虚拟机完成 CUDA 计算任务映射.受 CUDA 核函数维度的限制,该方法同时最多只能做 3 个表的连接操作.文献[48]实现了高效的 4 种 Join 算法(无索引连接 NIJ、索引连接 IJ、排序索引连接 SIJ 和哈希连接 HJ),无索引情况下,首先对连接表划分子表,再对子表对做笛卡尔乘积;在有索引时,先分别对连接表进行索引划分,再进行连接.文献[24]在 G-SDMS[23]系统中改进了 hash-join 和 sort-merge-join,利用 CUDA 超大寄存器组来加速连接操作,并首次解决了连接表大小超过单块 GPU 显存的问题.

在 CPU-GPU 集成架构上,文献[49]利用集成显卡架构下 CPU 和 GPU 共享内存无需数据传输的特点,动态地在 CPU 和 GPU 间划分计算任务,并依据代价模型做负载均衡,创造性地提出了异构计算环境下的 hash join 新算法.该算法有效避免了 PCIe 总线传输瓶颈,但现阶段集成式 GPU 架构的计算能力比分离式低得多,造成了其整体性能不佳.受动态调度的启发,文献[50]提出适用于列式存储的异构并发 join 算法:利用 ICMD(improved coordinate module distribution)给数据划分成彼此查询正交的独立分区,并在 CPU-GPU 架构中动态调度来均衡负载,达到并行化 join 算子的目的;此外,该研究是基于 Ocelot[13]的,能够利用分离式 GPU 高效的计算性能.

实践表明:GDBMS 在处理 Join 算子上比 CPU 方案效果好得多,平均可以取得 2 倍~15 倍的加速比.这是由于 Join 算子是计算制约算子(compute-bounded)决定的,也是 GDBMS 主要的优势所在.未来,如何进一步优化 GPU 上 Join 算子算法以及如何调整连接顺序(join-order problem),仍是 GDBMS 领域收益最高的研究热点之一.

3.2.2 OLAP 聚集函数算子

聚集函数算子是又一个计算负载很高(compute-bounded)的关系代数算子,适合使用 GPU 加速.在 OLAP 分析工作任务中,切片(slicing)、切块(dicing)、上卷(Roll-up)、向下钻取(drill-down)以及数据立方(cube)函数是 OLAP 业务中经常用到的算子,结合 sum,average,maximum,minimum,median,count 等各类聚集函数,提供给用户强大的分类汇总复杂信息的能力.借助 GPU 高并发高性能计算能力加速聚集函数算子,可以有效提升 GDBMS 竞争力.

现有研究聚焦到如何用 GPU 的高并发计算能力和可编程的存储层次结构加速多维聚集算子.文献[51]提出了 MC-Cubing 算法,通过改进自底向上广度优先 cache 优化算法 CC-BUC,在多核环境下,充分利用 CPU-GPU

异构计算能力实现了高效的 Cube 算子,相比于 CC-BUC,取得了 6 倍的加速比.文献[52]提出了适用 GPU 的 OLAP 数据立方表示数据结构以及配套算法集合,在 GPU 上实现了高效的多维聚合操作.作为底层模块支持 OLAP 算子的运行,该算法可以通过组合 map,reduce,scatter 等原语实现,并通过预先计算的方式用空间换时间加快运行时查询性能、缩短总体响应时间.文献[53]提出一种压缩多维数据立方的 semi-MOLAP 模型,使用维坐标 ID 索引表数据,解决了稀疏数据的 GPU 存储和查询优化问题.文献[54]对比了 GPU 和多 CPU 的 OLAP cube 创建算法的性能、可扩展性和优化策略,通过实验证明,GPU 可以比 CUP 实现 10 倍的加速比.但是文献同时指出,多 GPU 的数据立方算法并没有预期的那么高,如何实现多 GPU 数据立方算法还有待解决.文献[55]通过实现聚合核函数(SUM\MAX\MIN)来加速简单聚合函数算子,实验表明,GPU 聚合核函数方案能将算法复杂度从线性降到对数级别,比对应 CPU 并行算法可提高平均 32 倍的加速比.文献[56]系统对比了 GPU 上实现 TopK 算子的各类可能算法,包括基于排序、堆数据结构、高位前缀算法,提出了性能最优的基于二进制位归并的 TopK 算子(bitonic top-*k*).

3.3 空间数据查询

随着移动互联网、物联网、车联网的发展,基于位置服务越来越普及,空间数据查询变得越来越重要.为应对大规模空间数据处理的需求,大数据处理平台开发出了一系列产品,比如 HadoopGIS,SpatialHadoop, SpatialSpark,GeoSpark,LocationSpark 等.综述文献[57]调研了基于 Hadoop,Spark 系列的处理空间数据的产品,系统对比了其功能性、性能指标、实现方法等方面的优缺点.但基于 Hadoop 系列的时空数据分析平台采用批处理方式进行查询,响应时间过长.而在传统的数据领域,各 DBMS 以 GIS 插件形式提供了空间数据的处理,比如 PostgreSQL 的 PostGIS、Oracle 的 Oracle Spatial、IBM 的 DB2 Spatial Extender、Informix 的 Spatial DataBlade 等等.PG-Storm 等系统基于 PostgreSQL,对 Geo 地理数据,早在 1988 年,研究人员就试图将关系代数的成功经验引入到地理位置信息系统 GIS 中去[58].同样,传统数据库的 GIS 插件,在处理数据量、响应时间、查询吞吐量上都差强人意.

GDBMS 作为新兴数据库分支,以超过传统 GIS 系统两到三个数量级的时空数据处理速度和数据可视化交互式查询接口,引领了时空信息系统的发展潮流.其中,OmniSci(MapD)与 Kinetica 数据库可以借助 GPU 优秀的高速并发处理和图形化渲染两大优势的结合,以接近实时的处理速度进行时空大数据量查询和可视化,取得了极佳的用户体验.GDBMS 原型系统 GalacticaDB[21]通过使用 CUDAset 作为 GPU 并发数据格式,处理大规模空间数据查询,其架构设计合理、实现代码开源有很大的参考价值.文献[59]利用 GPU 的大规模并发计算能力为曲线的每一条边界(line)分配线程并发查询,在分布式计算框架(impala)上,使用均衡分区方法实现了空间连接的分布式并行计算.空间数据处理业已成为 GDBMS 独有的杀手级应用,性能远超大数据平台和传统 DBMS.

在算法设计层面,综述文献[60]对空间连接(spatial join)的 3 个典型算法模块——数据分区技术、在子区间做空间连接、多边形相交检测进行深入详尽的调研.由于二维空间下很难定义大小偏序关系,因此 merge join 或 hash join 不适用于空间连接,因此,传统的空间连接实现方法包括嵌套循环连接(nest-loop-join)、平面删除算法(plane sweep)、多维删除算法(Z-Order)及其变种.

在空间索引方面,将历史数据构建驻留 GPU 的空间索引,能够处理大部分时空数据查询,是未来发展方向之一.文献[61]在 GPU 上实现了基于排序批量装载的 R-Tree,并详细对比了各种 GPU 上的 R-Tree 实现的性能.空间数据的可视化问题需要运行大量空间聚合函数查询,即将点映射到任意形状的区域并统计信息.文献[62]利用 GPU 渲染通道实现空间聚合查询,实现了任意形状的空间聚合查询功能,同时保证了处理数据的实时性.STIG[63]通过改进 GPU 多维 kd-tree,实现了包含时间信息的可交互空间数据查询(PIP,多边形范围内点查询).G-PICS[64]在 GPU 上实现了四叉树(quadtree)索引,较 STIG 查询并发度更高,且支持动态更新.

在 GPU 空间查询理论方面,文献[65]提出了一种全新的 GPU 友好的空间数据表示方法及空间代数,将点、线、面统一成一种称为“Canvas”的统一空间数据对象,并重新定义了针对 Canvas 对象的 5 种基本空间算子:形变(geometric transfer)、值变(value transfer)、选取(Mask)、相交(Blend)和分解(dissect).该文献将空间查询变成基于 Canvas 对象的代数几何运算,利用了 GPU 擅长处理二维图像数据的特点,实现了空间数据的范围选择、距

离选择、形状选择、空间连接、聚集函数和近邻查询.实验表明:基于 Canvas 的 GPU 空间数据查询相较于 CPU 方法,可以取得 100 倍~1 000 倍的加速比.

3.4 KBE查询执行引擎

在查询执行引擎实现方式上,由于GDBMS普遍采用的CUDA\OpenCL以核函数(kernel)为载体执行协处理器计算,所以大部分 GDBMS 使用基于核函数(kernel based execution,简称 KBE)[40]的查询执行引擎.一种自然的实现 KBE 的方法就是将每个关系算子以一个 kernel 函数执行,大部分 GDBMS 基于此实现自己的查询处理引擎,比如 CoGaDB,MapD 等.在此基础之上,已有研究在核函数的层次上进行合并融合或细致切分两种策略:针对数据量大或计算复杂的任务,通过将多个核函数融合(kernel fusion)到一起,共同完成查询处理;而对于相对简单的任务,则进一步切分核函数(mini-kernel),做到精细化管理,以合理利用 GPU 资源.

在核函数融合方面,文献[18]提出了 MultiQx-GPU(multi-query execution on GPU)框架,通过提高查询并发度来提升设备资源利用率,即:通过将多个查询请求编译到一个核函数中,达到核函数复用的效果,提升 GDBMS 的整体性能.GDBMS 系统 GPUQP[16]以 10 个算子组成的查询子树为一个核函数执行查询处理.文献[66]提出了基于核函数合并的查询编译优化方案 Kernel Weaver,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的核函数合并为一个,不但减少了整体核函数的数量,同时降低了数据在主机端和设备端的传输开销;最为重要的是,增加了编译器的对 GDBMS 用户查询的优化空间.文献[67]使用数据预取和 SIMD 指令并行机制,实现松弛融合算子技术(relaxed operator fusion),可以有效提升查询并发程度,并降低物化中间结果的负载.核函数融合技术通过增加 GPU 函数处理操作的数量来优先使用 GPU 资源,使得同样配置下更能发挥 GPU 高并发高速率的优势,同时减少核函数的数量,减低了系统调用和管理的开销,非常适合 GPU 多计算少逻辑控制的硬件特性.但是核函数融合技术并不能增加单位数据的计算强度,不能从根本上提高加速比;其次,核函数融合技术增加了单个核函数处理所需要的资源,以降低 GPU 资源重复利用率为代价;最后,核函数融合技术目前还不能有效地跟查询优化器结合起来,如何有效融入真实 GDBMS 系统中是个未解之题.

在核函数切分方面,文献[68]提出进一步切分核函数,并通过精细化调度来提升系统资源利用率.文献[40]提出了流水线化查询执行树的查询执行框架 GPL,充分利用核函数计算和 IO 指令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流水化技术,在 GPU 并发调度不可知的情况下,解决了核函数并发执行多个关系代数算子的问题,提高了 GPU 资源的利用率,使性能提升 48%.MapD(OmniSci)通过将数据分区,对分区数据执行粒度更精细的核函数执行,不但实现了超过 GPU 显存容量的查询技术,而且有效提高了数据并发度,实现分块数据流水化处理的效果.CoGaDB 使用硬件无关的优化器 HyPE 来进行并发查询时资源的优化利用[11],着重解决运行时处理器负载分配问题,即所谓的算子分配问题.而文献[14]则在 CoGaDB 上提出根据数据所在位置(内存或设备显存)驱动查询树切分的策略,减少数据在总线上的传输,从而消除 PCIe 瓶颈.GDB[17]系统首次引入数据并发原语的概念,将核函数切分为更细粒度的数据并发原语来实现关系代数功能,实现了理论上的突破,成为 GDBMS 处理关系代数的事实上的标准.核函数切分能够以更精细的粒度管理 GPU 资源,使流水化成为可能,提高了资源利用率的同时,降低了单个核函数的运行周期,可以进一步提高查询并发度.但是核函数切分是以更频繁的核函数调度为代价的,数据换入换出频率更高,受 PCIe 总线瓶颈影响更大,如果控制不当,很可能降低系统整体性能.

此外,为了适应未来 GPU 的性能扩展以及多厂商间编程接口的差异,使用核函数适配的方式可以赋予 GDBMS 一定的硬件无关性,减少系统开发和维护成本.GDBMS 系统 OmniDB[25]以 GPUQP 系统为蓝本,使用核函数适配的技术,对于同一个 SQL 请求,使用适配技术生成不同的代码来处理查询,统一了不同厂商间 GPU 编程接口的不同,是一种提升 GDBMS 应用场景的技术.但是现有适配器模式只能针对编译好的 SQL 存储过程,在 SQL 执行的关键路径上增加了编译开销,离商用 Ad-hoc 查询还有一定距离.

基于核函数的查询执行模式适应 GPU 编程的范式,是目前 GDBMS 采用的主流技术.该技术兼顾灵活性和实效性,通过将算子预编译为不同粒度的核函数,在运行时根据数据的规模启动不同维度的线程块(warp)执行查询,同时保留了进一步优化的空间.但是,基于核函数的执行策略还面临数据传输瓶颈的考验和核函数容易崩

溃的问题,当数据超过一定限度或者中间结果物化代价过大超过显存容量时,需要引入全局的优化策略避免核函数失败重做的代价.同时,KBE 策略普遍没有考虑数据规模的问题,依赖于在运行时启动多少核函数、分配多大显存的策略,在实践中,这样的策略往往过于复杂而采用全列运算来避归,付出了过大的计算代价.

3.5 GPU事务处理

保证事务的 ACID 属性,是支撑 OLTP 业务的关键.可惜,众多 GDBMS 并没有致力于解决并发事务处理的问题.GPUTx[29]在理论上讨论了 GPU 实现事务并发控制算法的可能,该文献假设事务的所有读写操作可以在一瞬间完成,因而在赋予事务全局唯一的时间戳并以数据为中心排序事务操作之后,形成的事务依赖图 T-dependency 是无环的,可以用简单的拓扑排序形成多个无冲突的事务集.在 K-Set 无冲突事务集前提下,GPUTx 实验了 3 种并发控制算法:两阶段锁、单分区单事务、K-Set 无冲突事务集,并在 GPU 上实现了以数据为中心对操作排序执行的高性能 GPU 事务处理模式,比 CPU 算法提升 4 倍~10 倍的吞吐量.但是 GPUTx 对事务的操作在同一时刻完成的假设过强,在实践中面临事务并发读写冲突、无法支撑 ad-hoc 查询、长事务执行、排序操作负载过大等问题.目前,如何在 GPU 上实现数据库事务的并发执行还是一个开放问题,需要在 GPU 并发控制算法、GPU 数据高效获取、日志和故障恢复机制、GPU 高效锁实现机制、乐观并发控制策略等方面进一步研究.

3.6 小 结

查询执行器是 GDBMS 所有计算真正执行的核心引擎,决定了 GDBMS 的几乎全部的功能特性,是真正为用户提供价值的重要组件.数据库技术发展到今天,很难有一体适用的数据库技术包打天下,而 GDBMS 作为后来者,尽管有很大的应用前景和性能提升潜力,其功能应该是 GDBMS 设计者应该关注的重点.GDBMS 查询执行引擎核心功能是实现关系代数算子,目前主流的方法是采用并发数据原语分解关系代数的功能

目前,GDBMS 更多地面向 OLAP 业务,在查询与报表工具、多维分析工具、统计工具、空间数据处理、数据可视化、人工智能系统等方面努力寻找商业应用场景,业已在残酷的商业化竞争中占有一席之地.但是我们也注意到,鲜有研究[29]关注 GDBMS 事务的 ACID,这导致 GDBMS 无法支撑 OLTP 业务,严重制约了 GDBMS 的应用场景,对此,笔者表示非常遗憾.

4 查询优化器

GDBMS 查询优化器按功能可分为代价估计系统、查询重写规则、任务调度系统这 3 大部分,以查询编译器输出的逻辑执行计划作为输入,以生成适应 CPU-GPU 异构计算环境的代价最优的异构查询树为目标,并指导查询处理引擎执行计划.目前,大多数的 GDBMS 或者缺乏统一的优化器,由查询执行器按规则决定如何执行查询操作;或者用任务调度来代替优化器的作用,这就造成了一旦 SQL 编译完成,其执行过程就已经固定,放弃了优化查询的机会.而研究表明,未经优化的查询计划与最优的查询计划之间的性能可能有数量级上的差距.未来,优化器将扮演越来越重要的角色,是 GDBMS 能否在 CPU-GPU 异构计算环境下高效执行、发挥全部硬件性能的关键,也是保障查询安全、提高系统健壮性的核心部件,值得深究:首先,如何建立 GDBMS 查询处理的代价模型,是查询优化的核心问题;其次,在代价模型指导下,构建适应 GPU 查询处理的启发式规则体系对查询树进行等价改写,是 GDBMS 优化器的重要问题;再次,GDBMS 查询优化问题在一定程度上可以抽象为算子在何种类型的计算核心(CPU or GPU)上进行处理的问题,即算子分配问题,解决了算子分配问题,就能最大程度地发挥 GDBMS 整体性能优势;最后,如何在真实系统中设计实时高效的查询优化器,与数据库系统的其他模块协同工作,是制约 GDBMS 发展的关键问题.

4.1 代价模型

代价是数据库内核对给定 SQL 任务执行成本的估计,在传统数据库中包括 CPU 执行代价和 IO 代价两部分,是逻辑查询计划向物理查询计划转化过程中选择最优物理执行计划的依据.构建 GDBMS 代价模型,需要考虑 GPU-CPU 混合计算和异构存储层次特点,特别是 PCI-E 总线瓶颈问题,使其仍是一个未解决的开放问题.下

面介绍构建 GDBMS 代价模型所面临的挑战以及两种常用的采用代价估计方法——基于代码分析的“白盒方法”和基于学习的“黑盒方法”,并简要介绍算子选择率的估计方法.

4.1.1 GDBMS 代价模型之难

SQL 优化过程可以分为逻辑优化和物理优化两个阶段:在逻辑优化阶段,优化器根据关系代数等价定律、算子下推启发式规则、子查询重写等规则对逻辑执行计划进行改写,以得到执行代价更小的物理执行计划;而在物理优化阶段,需要为逻辑查询计划中的算子选择特定的算法和数据访问方法,并选择其中代价最低的一条生成路径作为最终的查询计划.这里非常关键的一点是如何估算查询的代价.传统的以磁盘为中心的数据库在查询执行代价估计方面有很成熟的经验,一般以磁盘 IO 和 CPU 计算的加权平均和作为最终的查询执行代价;在分布式数据库系统中,通信代价也是非常重要的度量标准;在内存数据库中,由于数据尽量放在内存中,消除了磁盘 IO,代价估计的重点是内存访问.而在 GDBMS 中,查询代价的估计问题变得很不一样.

- 首先,GDBMS 的计算代价不仅仅包括 CPU 计算,还包括 GPU 计算,而 GPU 计算能力往往是 CPU 的 10 倍以上.传统的方法以查询涉及的 tuple 条目的数量之和来估计其计算代价,这是基于各 CUP 的计算能力大致相同的假设;但在 GDBMS 中,必须根据 tuple 计算的位置(在 CUP 中还是在 GPU 中)来分别计算其代价.在 GPU 中,计算的代价由于其速率更高所以必须乘以一个经验系数(比如 0.1);
- 其次,GDBMS 访存代价的估计变得更加复杂.与内存数据库类似,GDBMS 将数据尽可能地存放在内存中以消除磁盘 IO 瓶颈,甚至有的方案将数据完全放在 GPU 显存中,因此,GDBMS 中存储层次体系更加复杂——既包括传统的缓存-内存-硬盘三级存储管理,又包括主机内存与 GPU 设备内存的异构存储体系管理,这决定了 GDBMS 必须设计独有的、能够充分反映 GDBMS 存储特性的访存代价估计函数;
- 最后,GDBMS 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[69]等研究指出:PCIe 总线瓶颈是所有 GPGPU 算法不能忽视的性能开销,也是 GDBMS 的性能瓶颈所在.文献[20]对开源 Alenka 系统运行 TPC-H 基准测试进行了详尽的性能分析,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 环境下的优化查询性能仍然是一个未解难题.

以上因素共同作用,造成了 PCIe 总线数据传输开销成为了 GDBMS 代价估计的重点和难点.

4.1.2 算子代价估计的方法

目前,GDBMS 主要有两种方法获取查询计划的代价:基于代码分析的“白盒”方法(analytical)[17,40,43,49]和基于学习(learning-based)[11−13,15,70]的“黑盒”方法.基于代码分析的方法认为,查询的执行取决于开生成指令的代码,因此可以通过静态的代码分析方法将 GPU 上执行的关系代数算子的执行代价精确估计,包括传输代价、计算代价、传回代价及内存访问代价.这种方法依赖于对每个算子的精确估计.对于多 GPU 协同运算,文献[71]系统介绍了确定性计算任务的多 GPU 性能代价估计方法,假定每元素和每子集的开销在多 GPU 环境下具有不变性,同时考虑了 PCIe 通信开销.文献[17,42]采用分析型 cost 模型,对一个实际的 GDBMS 系统——GDB 分析其关系代数算子、Join 算子的执行 cost 代价,其对查询时间的估计由 3 部分组成:传输到 GPU 的时间、GPU 运算时间、传回到 host 端的时间.该方法针对特定的算子实现,其准确率可以达到 90%.但是由于采用了静态的“白盒”分析的方法,只能针对算子级别进行分析,没有考虑多查询并发情况下算子之间的相互影响以及系统运行时的负载情况,不可避免地将在运行时发生错误;而且随着系统的进化,任何代码上的微小改动都将对代价估计产生影响,对代价估计模块的维护成本也随之增高,在扩展性和可维护性上面临巨大挑战.

而基于学习的方法将算子视作黑盒,通过机器学习的方法,经过观测算子的过往行为,估计该算子的执行代价,并在运行时利用反馈机制修正算子代价.这种方法在一定程度上避免了静态方法的一些问题,但同样不够完美:首先,基于学习的方法需要一定的训练过程,这在实际生产环境下面临冷启动缺乏基础统计数据的问题,造成优化器可发挥作用的时效性低,切换任务后面临新的“训练空窗期”;其次,基于学习的方法对算子代价的估计

是不精确的——执行计划的代价估计需要考虑的空间过大,很难在训练阶段穷举所有的情况,这就造成了对算子代价的估计模型的训练上面临训练样本空间和测试样本空间不重叠、训练样本过少的问题,模型必然存在精度不高和泛化能力不足的问题;最后,现有基于学习的方法没有考虑多显卡、多计算节点分布式的应用场景,低估了 PCIe 总线瓶颈的影响.基于学习的 cost 模型中,文献[15]对比了 sort,indexed scan,update 这 3 种常用数据库算子的性能均衡点,证明特定算法在不同输入情况下,可以根据性能估计来选择合适的处理器来优化系统的性能.文献[72]在 PostgreSQL 平台下,针对 TPC-H 基准测试使用基于学习的方法,在查询和算子两个粒度下对查询的执行时间(query execution latency)进行预测,实现了线下分析和线上执行决策结合的查询性能预测.

在现今复杂多变的硬件环境、不断变化升级的异构计算图景(landscape)、分析任务的越来越复杂、与操作系统之间交互的不确定性等原因,分析型的查询代价估计在实践中很难做到高精确度,而且基于训练模型再预测的学习型代价估计系统在时效性、准确度、泛化能力上难以满足数据库系统的要求.因此,将两种方法结合的优化器 cost 模型在未来会很有前景,比如用分析模型为学习型 cost 系统设定初值,来解决学习型代价系统的冷启动问题.

HyPE[12,13]系统高效地解决了异构环境下算子分配问题、负载均衡问题,是不可多得的基于 cost 代价估计可移植物理查询优化框架.但是我们也应该看到:该系统的“学习”算法只在算子层级上决策算子在哪个计算核心(CPU OR GPU)上,从理论上无法保证选出最优的执行计划,仍存在较大的改进空间;同时,HyPE 还无法对多显卡架构下的并发环境进行优化,同时无法适应多计算节点的分布式计算环境,未来需要做的大量的研究工作.

4.1.3 算子的选择率估计

对于一个特定的关系算子来说,基数估计(cardinality estimation)就是对算子运行前后数据的变化量评估的技术,其中,选择率(selectivity)评估占据了重要的位置.选择率是指满足算子条件的数据数量与数据总量之间的比值.算子选择率的精确估计对中间结果大小估计、算子代价估计、最优 join 顺序等都有重要影响,不但直接决定了优化器的准确性,甚至对 GPU 内存管理产生直接影响.因此,选择率的快速而精准的估计对 GDBMS 来说十分重要.目前最好的解决方案是基于抽样的柱状图方法,即等宽或等值地抽取数据,事先建立多区间的统计条形图,查询时,结合查询条件及柱状图信息估计算子的选择率.

文献[73]提出了使用 GPU 加速的基于核函数(kernel density estimation)的多维数据过滤算子选择率估计方法,该方法使用数值优化 KDE 方法,在数据更改或查询执行时可动态调整选择率,并将该模型通过 GPU 加速提升算法速度.该方法提供了选择率估计的统计学方法,精度及适应性不如现有的直方图方法,使用 GPU 加速的方式也对 GDBMS 的性能提升帮助有限.

针对空间 join 的选择率估计问题,文献[74]提出了多维空间累计柱形图 histogram 结构和积分表技术的并发选择率计算方法,能够在 GPU 上使用较少的资源实现选择率的并发估计.该方法需要对 join 数据提前分析,不适应于运行时查询的需求.由于需要占用宝贵的 GPU 显存,在应用该方法前,需要仔细权衡利弊.

目前,由于现有选择率估计的精度和性能仍有待提高,无法根据其对数据的估计结果精确控制用以存储中间结果的缓冲区.但是,利用深度学习等智能算法计算选择率提升空间很大,值得进一步探索.比如,可以利用深度神经网络(DNN)来预测一个算子的选择率,而 GDBMS 中由于数据就在 GPU 上,可以进行本地训练 DNN,可以预见,其性能将比传统的 CPU 算法要高.

4.2 查询重写

查询重写是指优化器对逻辑查询树等价变形,以达到提高查询效率的目的.在 GDBMS 中,传统的查询改写策略同样适用,当然也有细微的不同之处,比如对于“下推算子”策略,由于投影操作在列式存储下可以直接用物理投影解决,因此不存在下推投影的问题.目前的研究主要致力于改进 Join 算子的性能和合理消减 copy 算子来控制数据 PCIe 总线传输总量上.

4.2.1 join 算子改写

文献[36]针对 SSBM 测试提出了 invisible join 技术,对星型模式下的事实表与多个维表外键连接(star join)进行优化.该技术通过将 JOIN 重写为事实表上的多个 Select 操作,并用布尔代数将多个选择的结果合并,以减少

后续 JOIN 阶段的整体数据量.CoGaDB[10]系统通过位图(bitmap)改进 Invisible-Join 技术,实现了 Star Join 查询.

对于OLAP业务中最常用的两个表间的PK-FK Join,CoGaDB则使用文献[42]提到的适应GPU的NLJ(nest-loop-join)算法,先对 PK 列排序,再给每个工作线程组(warp)分配一定数量的 FK 值,warp 用二分查找法在排序 PK 列中查找匹配值.该实现的主要改进在于跳过了线程各自计数匹配 tuple 数量的过程.对于超过显存的大表 PK-FK 连接,文献[75]通过将事实表放在主存、维表放在显存中,试图利用 PCIe 总线非对称性带宽传输特点,发挥出全部硬件的潜力,以提升 Star-Join 的性能.

表连接的顺序是查询性能优化的关键.在现有的 GDBMS 中,各系统放弃了连接顺序的优化,采用确定性的查询计划构建方法,留下了较大的优化空间.文献[76]提出一种端到端的表连接顺序的基准测试——JOB 测试.文献[77]在多核 CPU 框架下,用动态规划算法遍历所有可能的表邻接顺序,通过将数据分区,用多核的并发性处理多表连接顺序的指数增长,为优化器提供保留了所有可能的连接顺序的查询执行树空间,最多可以处理 25 个表的连接顺序问题.文献[78]讨论了用 GPU 加速多表连接顺序问题的动态规划算法所面临的挑战:首先,动态规划算法是顺序算法,不易并行化,现有的 CPU 顺序算法的上限可以解决 12 个表的连接顺序问题;而 GPU 拥有更大规模的并发处理能力,理论上有潜力彻底解决多表连接顺序问题,但是需要解决分支消除、传输瓶颈、优化访存、并行计算等问题.目前,多表连接顺序判定仍然是数据库优化领域开放问题之一,还没有一个可以利用 GPU 加速该问题求解的成熟方案.

4.2.2 减少 copy 算子

对于查询优化,可以通过查询重写技术减少 copy 算子的数量为目标,来减少在 PCIe 上来回反复传输的数据量,从而提升性能.文献[70]在算子序列化阶段,通过穷举 copy 算子位置不同的算子执行计划,并用贪心策略找到能使 GPU 算子尽可能多(两个 copy 中间算子尽可能长)的执行计划,以减少数据在 PCIe 总线上的传输,最大化利用 GPU 的计算能力.但是此方法只适用于单查询执行计划的情况,且没有考虑算子间的依赖关系,而其基于贪心的策略并不能保证生成最优的查询计划.

文献[14]提出一种根据数据位置驱动算子分配的优化策略,即:只在算子需要的数据已经在 GPU 上时,才将算子分配给该 GPU 执行计算.在数据管理上,该文将 GPU 显存划分为管理算子输入输出数据的 Cache 缓冲区和存储中间数据结构 Heap 堆栈空间,通过对并发查询占用资源的集中控制,来解决缓存抖动问题和数据反复迁移(数据 ping-pong)问题.

此外,前文提到的 KBE 核函数执行策略中,核函数融合技术能够有效减少 copy 算子的数目;而核函数切分则会显著增加 copy 算子数量,增加本就繁重的 PCIe 通信成本.综合应用核函数融合和切分技术,有望进一步提高 GDBMS 的查询性能.如何有效综合各种优化策略来减少 copy 算子,将成为未来 GDBMS 优化器设计的重中之重.

4.3 异构计算任务队列

GDBMS 与传统 CPU 数据库的不同之处在于,可以利用 CPU-GPU 异构计算平台并行处理数据.为保证多个计算设备(CPU 和 GPU)处于“繁忙”状态,保证系统整体性能[12],给每个计算核心维护一个工作队列,根据启发式规则,将查询计划树中相互独立的算子分配给工作队列并发执行.

以响应时间为优化目标下,可以使用 SRT(simple response time 简单响应时间最小策略)或 WTAR(waiting time-aware response time,停等时间最少)调度策略[7].对于 SRT,系统估计单个算子的执行时间,选择工作队列中执行该算子代价最小(时间最短)的队列分配即可.而在 WTAR 策略下,系统需要对整个工作队列中所有已经就绪算子的执行总时间以及算子各计算核心的执行时间来计算各任务队列的停等时间,以系统整体停等时间最短为原则分配算子.细致的实验对比下,WTAR 策略效果在所有情况下都效果最佳.

以系统整体吞吐率[11]为优化目标,则可以使用公平轮询(RR)、基于阈值(TBO)、基于概率(PBO)这 3 种调度策略.

- Round-robin(RR)策略将算子轮番均分给各个任务队列,以公平的策略进行调度;
- TBO(thread-based optimization)基于阈值,给每个计算核心 CU 一个阈值,超过阈值之后就不再往该任

务队列中分配任务.这样有两点好处:首先,TBO 解决了 SRT 策略负载不均衡的问题,避免了因并发算子过多引起的资源耗尽问题;其次,TBO 给了我们选择次优执行计划的能力,这在一定程度上避免了 cost 代价估计不准而造成的分配失效的问题;

- PBO 利用归一化指数概率函数 *Softmax* 计算最优计算核心 CU(CUP/GPU)的概率值,并在算子分配时按概率分配计算任务.PBO 策略可以并发使用多 CU 来提高系统吞吐率,同时以最大概率将算子分配给最快的 CU.

RR\TBO\PBO 都是在算子层面上对最佳 CU 选择上的调度策略,在保证响应时间接近最优的情况下,以负载均衡的方式提升系统整体的吞吐量.

现有研究表明:启发式调度策略解决算子分配问题,WTAR 策略是最优的方案.但是在更大的范围来看,GDBMS 算子分配问题是一个多目标的优化问题,启发式调度算子策略无法解决关联算子切换 CU 过程中引起的数据传输代价,还需要进一步改进以扩大优化视野.

4.4 真实GDBMS系统中的优化器

HyPE 优化器框架采用可移植分层设计,在很好地解决了查询性能预测(query performance prediction)和算子分配问题的基础上,可通过与特定数据库兼容的适配层,理论上有与任何 GDBMS 结合的可能性[7,8,10−13,70].异构查询处理优化引擎 HyPE(a hybrid query-processing engine)[12,70,79]采用灵活的分层设计,针对异构环境下查询物理优化.该系统采用基于学习的方法,由 3 个部分组成:代价估计器(cost estimator,简称 STEMOD)、算法选择器(device-algorithm selector,简称 ALRIC)、异构查询优化器(hybrid query optimizer,简称 HOPE).配合与特定数据库适配层,有与任何 GDBMS 配合使用的能力.通过与 CoGaDB[10]结合,证明了其与基于 CUDA 框架的 GDBMS 的结合能力;而 Ocelot[13]与 HyPE 的融合,再次证明了硬件无关优化器框架 HyPE 能够与所有基于 OpenCL 的 GDBMS 配合使用.图 3 描述了 HyPE 的工作过程.

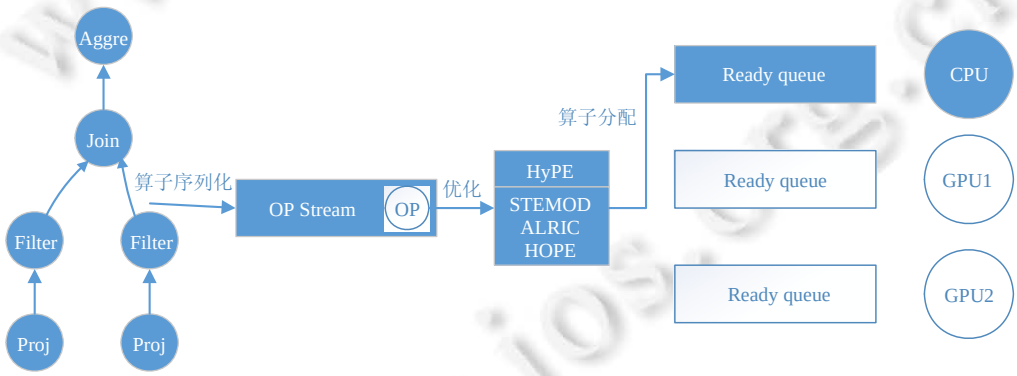

Fig.3 HyPE workflow

图 3 HyPE 工作流程图

- 首先,在 HyPE 中,假设算子跟其处理的数据是一个整体 *OP*(*A*,*D*),*OP* 表示算子、*A* 表示使用的算法、*D* 表示处理的数据,则物理执行计划可视为一个算子的流处理器.对于逻辑执行树进行拓扑排序来序列化算子,消除算子间的数据依赖,保证每个算子执行时其子算子均已执行完毕;
- 其次,算子流中的算子依次经过优化器 HyPE 中,经由代价估计器(cost estimator,简称 STEMOD)[15,72]、算法选择器(device-algorithm selector,简称 ALRIC)[15,72]、异构查询优化器(hybrid query optimizer,简称 HOPE)[70],为特定算子选择合适的实现算法,并根据启发规则为算子分配合适的处理器.

HyPE 的 3 个模块内部工作流程如图 4 所示,对于查询计划中的每个算子 *O*,在算法选择器中选出其对应的 CPU\GPU 算法 *A*,经由代价估计器,以测试数据集 *D* 和算子的参数 *P* 运行机器学习算法,估计对应算法的代价,最后由异构查询优化器按照启发式规则选择最终的算法和执行计算核心,并将选择结果反馈给系统作为后续

代价估计的依据.

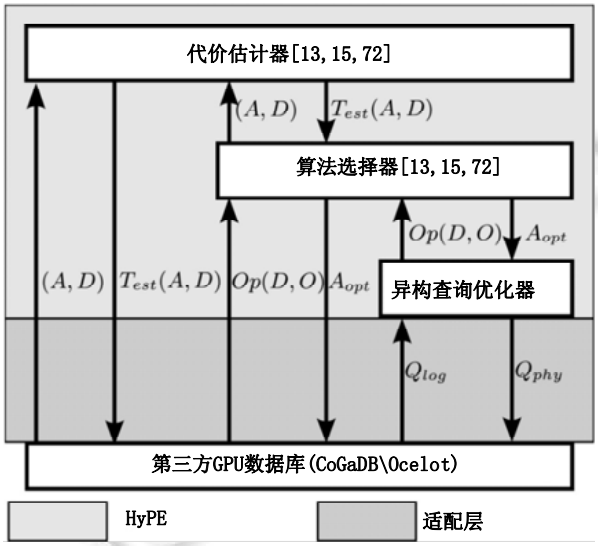

Fig.4 HyPE architecture[12]

图 4 HyPE 架构图[12]

尽管 HyPE 优化器基于机器学习的方法估计算子执行代价,并组合多种启发式规则选择算子的物理实现算法,但是将查询优化问题等价为在运行时处理算子分配问题的优化策略对 GDBMS 来说未必是最有效的.文献[80]指出了 HyPE 采用局部优化(local optimize)策略,直到优化期再考虑的算子分配问题,由于将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,由于缺乏全局视野容易陷入局部最优解,还可能会造成不必要的数据传输代价.该文提出了从编译期开始考虑算子分配问题的全局优化(global optimize)策略,该方法统筹考虑整个执行计划(QEP)各算子之间的依赖关系,鼓励算子间共享数据来消除可能的数据拷贝.全局视野下考虑算子分配问题,主要存在两个问题:首先,优化要考虑的问题空间变大了,使得算子分配的复杂度成指数增长;其次,物化中间结果的空间大小难以精确估计,也增大了算子级联失败的可能.

此外,PG-Storm[81],brytlyt/BrytlytDB[82]基于开源数据库 PostgreSQL,复用了 PostgreSQL 优化器模块,通过计算 cost,在查询执行加护中插入适合的 GPU 算子(Join、Scan、聚合等),将计算任务分配到 GPU 端加速,取得了较好的效果.这种编译期静态分配的任务分配策略,避免了 HyPE 优化器在运行时决策算子分配问题的负载,可以取得确定性的性能提升.但是这种方法只能对 Join 等 GPU 与对应 CPU 算子性能相差悬殊的一类算子起作用,放弃了大多数可用 GPU 加速查询的机会;同时,由于体系结构上的差距,无法结合列式存储等 OLAP 分析任务上行之有效的优化方法使用,整体性能提升不如纯粹的 GDBMS 高.Ocelot[35]克服了 PostgreSQL 行式处理的弊端,同时可以借助 MonetDB 的优化器进行查询优化,采用规则化的决策策略,尽可能将计算任务部署到 GPU 端来加速查询.Ocelot 优化方案存在如下弊端:首先,Ocelot 的优化器依赖于 MonetDB 查询优化器,并没有考虑异构环境下的查询优化问题;其次,Ocelot 采用静态分配算子策略,不如 HyPE 运行时动态分配算子高效.因此,有研究[13]将 Ocelot 与 HyPE 结合,引入了动态算子分配策略.此外,文献[80]进一步引入了全局优化的策略,尽管全局优化效果与局部优化提升有限.

4.5 小 结

传统数据库的经验表明,未经优化的执行计划和最优执行计划之间的性能差异是巨大的.考虑到异构计算环境特点、代价模型的变化、最优连接顺序问题、算子分配问题、PCIe 瓶颈问题、查询规模估计等难题,异构查询优化问题更加复杂.未来,人工智能赋能的异构查询优化器将成为研究的热点,在 GDBMS 的架构中发挥

核心关键作用.

5 存储管理

数据库管理系统的任务,本质上是在一定的存储介质上读写数据.因此,存储管理器成为其密不可分的关键模块.传统的 DBMS 与 GDBMS 在存储管理器上存在以下不同.

- 第一,从功能上说,面向磁盘的数据库主要包括内存管理和外存管理两部分;而对于 GDBMS 来说,还增加了 GPU 显存管理的任务.如何充分发挥异构存储体系的性能优势,是 GDBMS 数据管理最核心的问题;
- 第二,压缩数据能够有效减少需要处理的数据总量,进而成比例增加 GDBMS 的性能.如何在 GPU 异构显存体系下尽可能获得更大的压缩比,充满了挑战;
- 第三,索引能够加速以硬盘为中心的数据查询处理,但是在 GDBMS 大量物化的全量处理模型中是否还有存在的价值、如何发挥索引长处加速查询处理,也是 GDBMS 存储管理需要研究的重要问题.

本节将就 GDBMS 存储管理面对的挑战、GPU 数据压缩技术、GPU 数据索引技术以及提升 GDBMS 处理数据容量的软硬件技术进行深入细致的分析,希望对未来的 GDBMS 研究和开发有所启发.

5.1 GDBMS数据存储体系

在数据存储模型上,列式存储在 OLAP 业务中比行式存储在性能上有明显优势[27,36],业已成为绝大多数 GDBMS 的选择.

- 第一,列式存储更方便压缩.属性值之间值域有限、高相似度、等位宽等特点可以获得更高效的压缩解压缩算法、获得更高的压缩比,甚至可以直接用压缩格式进行查询处理.比如采用字典压缩的方式下,完全可以在压缩形式的数据上进行查询操作而无需引入解压缩的负载;
- 第二,列式存储减少了需要处理的数据容量.由于分析类查询往往只涉及记录中有限的属性列,列式存储可以实现“物理投影”,进而减少不必要的数据获取、处理开销;其次,使用尽可能晚的物化操作、通过“位置”等中间信息生成最终结果,而在查询处理过程中无需进行多余的行式结果构造,也对性能提升帮助很大,比如 CoGaDB 使用 tid 在 GPU 和 CPU 之间传递中间结果;
- 第三,列式存储更适合 GPU 处理.由于关系表中一个列内的数据在数据类型上一致,因此对列中每个元素的处理存在等价性,设计向量化算法非常适合 GPU 的大规模并发编程模型.

在此基础上,一些针对列式存储的优化方法(比如 invisible join[36])、GPU 显存的聚合操作特性,也是行存储无法比拟的.但是列式存储更适合 OLAP 业务,对于 OLTP 事务处理有天然的短板,列式存储将让 GDBMS 难以适应 OLTP 业务.

在数据存储位置上,由于 GDBMS 需要管理硬盘(包括机械键盘和 SSD 等)、内存(CPU 端)和 GPU 显存这 3 层存储结构,其中,GPU 显存包含了超过 8 种不同类型的存储空间——全局显存(global)、共享显存(shared)、常量显存(constant)、纹理显存(texture)以及各级缓存(cache),GDBMS 的存储管理需要考虑的空间变大了.文献[83]提出了一种度量因存储数据位置不同而造成程序性能不同的方法,帮助程序开发人员自动寻找最佳的数据存储布局方案,但该方案并不是针对关系型数据库的.该研究表明,GPU 指令重试、地址编码模式、访存指令队列等待延迟、各级缓存失效效应是造成 GPU 显存不同存储位置性能差异的主要原因.因此,GDBMS 的存储管理与传统的 DBMS 和内存数据库类似,数据需要存储在大而慢的硬盘存储器上,但在快而小的 GPU 显存中查询处理.不同之处在于:数据可以在主机内存和设备显存上分别计算,带来了优化的可能.

GDBMS 普遍采用内存驻留(memory-resident)的技术,将数据尽可能放在内存中(甚至放在 GPU 显存上)来避免磁盘 IO 瓶颈.文献[17]指出:GDB 系统对于稍大于内存的数据进行 TPC-H 测试时,相对于 CPU 加速方案,性能仅提高 23%;而对于全内存驻留数据,GPU 方案可提供多达 27 倍的加速比.来自 MapD[5]系统的数据表明:相对于访问驻留硬盘(1GB/s~2GB/s)上的数据,全内存(32GB~3TB)计算速度为 70 GB/s~120GB/s,加速比为 35~120;而如果只用 GPU 显存存放数据(容量可达 384GB),速度为 3000GB/s~5000GB/s,加速比可达

1500~5000.MapD 通过将渲染引擎融入数据库内核,使得数据可视化的速度接近 GPU 处理的极限,对于交互式商业智能图表、空间数据分析、聚合统计等无需返回原始数据的应用起到了极大的加速作用.但是对于传统的 ad-hoc 查询,由于 GPU 只能加速存放于内存的数据,GDBMS 的处理速度实际上还远未达到 GPU 显存的极限.一些 GDBMS 直接将整个数据库驻留在 GPU 显存中,由于 GPU 片内显存比 PCIe 总线速率高 16 倍,因此可以预见,这种方法可以有效提升性能.同时,这也简化了数据管理,不需要额外考虑如何保证内存和显存的数据一致性.但是,这极大限制了 GDBMS 的数据容量,因为即便使用多块显卡,总显存容量跟内存的容量相比仍然有数倍的差距.

如果只用内存完全不用硬盘,会极大增加部署 GDBMS 系统的成本,限制 GDBMS 能够处理的数据容量,并且内存断电后数据丢失的特性,也给数据安全带来挑战,在与其他大数据处理系统竞争中也将失去优势.商业的 GDBMS 普遍采用硬盘存储数据,在系统加载时,将数据尽可能部署到内存中,并利用基于代价的优化器或启发式规则,尽可能地选择 GPU 进行查询处理.PG-Storm[81]系统引入了 SSD 数据通过 PCIe 总线直接向 GPU 发送查询数据的功能,未来有望在不增加 I\O 瓶颈的前提下,将大容量非易失性存储引入到 GDBMS 体系当中.SPIN[84]则在文件系统层级上通过内存页面缓存、GPU 数据预读、轻量级硬盘地址转换,解决了 GPU 和 SSD 硬盘之间的点对点(P2P)数据传输问题.该方法建立了 GPU 版的 DMA 机制,无需 CPU 控制,降低了系统的管理负载;同时,不需要内存作为“跳板”,有效减少了 PCIe 总线瓶颈,对 GDBMS 的发展具有重要的启发意义.

现有研究型 GDBMS 原型系统,如 CoGaDB,GPUQP,GPUDB,OmniDB 等都针对单显卡系统,鲜有多显卡 GDBMS 的相关研究,限制 GDBMS 容量扩展的一个直观因素是显卡的容量.但是相应的 GPU 单块显卡的显存容量上的提升不像其计算能力的增长迅速,比如 K80 显卡通过集成两块 GPU 核心已经可以达到 24GB 的显存,最新一代 Nvidia 显卡单块容量普遍还是在 12G~16G 之间,虽然最高容量已经达到 64GB,但相应的价格也过于昂贵.商业 GDBMS 在这方面走在了研究界的前面.将多块 GPU 合并处理 SQL 请求,是扩展数据库处理能力、提高数据库容量的自然解决方案.DB2 BLU[85]数据库引入多显卡处理查询机制,通过统一管理多 GPU 显存,统计查询对显存的总需求,在运行时,根据各 GPU 资源使用情况决定在哪个 GPU 上执行查询任务.MapD[5]通过联合最多 8 块 GPU 同时处理查询请求,可以在 GPU 显存时处理 1.5TB~3TB 的数据,而内存中数据更是多达 15TB,以每秒 2 600 亿行的查询处理速度.随着 GPU 计算能力的超摩尔定律发展,我们相信,现在的 GBDMS 在数据处理速度上已经远超内存数据库.但是多 GPU 系统中,单台服务器 PCIe 总线接口限制了可集成的 GPU 数量.因此,单靠增加 GPU 数来增加 GDBMS 的数据容量的解决方案效果有限,需要引入分布式技术,用集群来扩充 GDBMS 容量.这也是未来 GDBMS 的一个重要发展方向.现有的 GDBMS 多针对 OLAP 业务,数据一定时间内保持不变,对于分布式横向扩展友好.比如:SQream DB[86]通过将存储引擎独立出来,采用多查询共享存储的架构,在一定程度上实现了数据库的横向扩展,提供从 10TB~1PB 的 OLAP 数据查询云服务.未来,结合分布式技术的多显卡系统,将成为 GDBMS 的发展方向.同时,单卡多核 GPU 架构的 scale up 扩展方案也值得关注,文献[87]提出一种评测 NUMA GPU 架构性能的 GPUJoule 系统,该系统可以模拟多达 32 个 GPU 核心的性能扩展和能源效率指标 EDPSE(energy-delay-product scaling efficiency).研究表明,多核 GPU 系统的设计难点在于 GPU 的本地显存(local memory)设计和 GPU 核心间通信网络的设计,因为目前单核 GPU 的显存带宽还不足 DRAM 理论上限的三分之一(300GB/S vs 900GB/S),提升空间巨大.可以预见:能够充分发挥多核 GPU 性能的 GDBMS,无论采用横向扩展(scale out)还是纵向扩展(scale up)的方式,都有巨大的性能潜力可以挖掘.

未来,随着 GPU 显存容量和速度的继续提升和分布式技术的引入以及分片分区策略的使用,相信 GDBMS 会在数据库容量上不断提升,逐步拓展应用场景,前景一片光明.

5.2 GPU数据压缩

数据压缩技术历史悠久,早在关系代数理论创立之初,人们就致力于将数据压缩用于到数据库管理系统中来.GDBMS 普遍采取压缩技术,以降低数据存储空间,节约宝贵的内存、显存资源.采用压缩技术存储数据,还可以降低设备与主机间必须传输的数据容量,缓解 PCIe 总线瓶颈问题.但是,利用压缩技术引入了压缩-解压缩步骤,增加了系统响应时延,在实践中必须做出权衡.

在传统的以 CPU 计算为中心的内存数据库系统环境下,文献[88]利用架构方法的调查缓存和主存系统中的数据压缩对近年来在缓存和内存上应用各类压缩算法,达到提高性能、减少能耗的目的.该文希望能帮助理解压缩算法的研究现状,激励研究者在设计现代计算系统时提升压缩算法的效率、扩大数据压缩的应用范围.在内存数据库领域,HANA[15],MonetDB[38],C-store[89]往往采用字典压缩、位图压缩等轻量级压缩算法,达到在压缩数据上直接查询的目的.文献[90]将数据压缩集成到列式数据库 C-Store 系统中,使用多种(null suppression, dictionary encoding,run-length encoding,bit-vector encoding,lempel-ziv encoding)压缩算法,并实现了多种压缩数据格式之上直接运行 SQL 查询,省去了解压缩的过程.但是该文采用传统的火山查询模型,引起的分支恶化问题并不适应 GPU 编程环境;其针对不同压缩格式定制特定算子的处理模式会引起算子数量剧增的问题,在算子匹配上也增加了优化调度的难度.

适用于 GDBMS 的压缩算法选择上必须遵守如下原则.

- 第一,必须是无损压缩算法,因为数据库业务的要求,数据如果压缩后有损失,那么势必影响查询的准确性,任何以损失业务正确性来提升性能的措施都是不可取的;
- 第二,一般采用轻量、上下文相关度低的压缩算法,这样压缩-解压缩过程的负载开销不大,不会造成严重的性能问题;
- 第三,解压缩算法应该易于并行化,方便利用 GPU 富裕的计算资源.相比于压缩过程,解压缩流程一般处于查询处理的关键路径上,对查询速率的影响更加关键;
- 第四,多种压缩算法可以组合使用,进一步提高压缩比.而如何选择合适的组合策略,将会成为将来研究的热点.

在 GPU 数据库中,Kinetica 为用户提供 snappy 等传统数据块压缩算法,在查询执行之前先解压再在未压缩数据上执行查询.这样虽然可以支持全部的 SQL 查询算子,但是代价也非常巨大.文献[91]分别在 GDB 以及 MonetDB 上组合使用 9 种轻量级压缩算法,提出部分压缩模式,并将数据压缩引入到代价估计模型中,通过优化器生成考虑数据压缩因素的异构 SQL 查询执行计划.研究表明:GDBMS 通过使用数据压缩技术,可以大幅减少 PCIe 总线传输数据量,进而将整个查询运行时间提升一个数量级.文献[92]用轻量级压缩算法 WAH 压缩位图索引以支持范围查询,数据首先以压缩形式发送给 GPU,再在 GPU 上以 DecompressVector 技术解决 WAH 压缩数据无法用 GPU 并行处理的问题,实现了在压缩数据上直接进行查询,性能较 CPU 并行算法最高提升了 87.7 倍.

在 GPGPU 研究领域,一些研究通过在现有的异构计算环境软硬件栈的不同位置引入压缩-解压缩缓冲层的方案,对加速 GDBMS 也许有借鉴价值.文献[93]提出了 CABA 方法,使用辅助线程族,利用 GPU 的空闲计算资源对数据进行压缩-解压缩操作,以消除显存带宽瓶颈.CABA 方法采用软硬件结合的设计理念,在 GPU 每个 SM 控制器中上增加了辅助线程代码存储器、控制器、数据缓冲区这 3 个硬件基础设施,使数据以压缩形式存储于显存中,并在调入 cache 之前进行解压缩.文献[94]提出了 Warped-Compression(WC)方法,利用同一线程族 warp 之内每个线程之间的寄存器内容相似度高的特点,对 GPU 中数量众多的寄存器文件进行压缩.文献[95]使用 GPGPU-sim 模拟器改变 GPU 显存控制器 MC,对来自 PCIe 总线的数据进行压缩解压缩操作,实现了浮点型数据的有损和无损压缩算法.HippogriffDB[22]综合运用多种压缩算法,用 GPU 的计算能力换取高效的数据传输速率.为解决不同查询适应不用压缩算法的问题,HippogriffDB 采用同表多压缩格式,并在查询时采用自适应压缩的方法;为解决 PCIe 传输瓶颈,HippogriffDB 使用 SSD 到 GPU 直接传输压缩数据的办法来提升整体性能.实验表明,HippogriffDB 可以取得比 MonentDB 快 100 倍、比 YDB 快 10 倍的加速比.

压缩算法的引入可以有效减少数据的总容积,同时在轻量级压缩上可以直接查询,消除了解压缩负载,进而全面提升 GDBMS 的性能.在未来的研究中,适应于 GPU 处理的易并行、压缩比更大的压缩算法将成为研究的重点.而在应用领域,轻量级压缩算法的使用会越来越多,并逐渐寻找大压缩比算法的适用场景.

5.3 GPU索引技术

传统数据库中,使用 B+树等索引结构减少访问数据时磁盘读写的负载.在 GDBMS 中,由于索引结构在访存特性上的随机性,不满足 GPU 对齐合并访问的特点,传统的索引结构照搬到 GPU 异构计算环境下性能不高;甚

至,由于 GDBMS 全内存存储和一次一算子全物化的查询处理方式,不使用索引一样可以达到很高的性能.因此,在 GDBMS 中,如果使用全显存存储数据,索引可能是多余的模块;但对于要直接从硬盘存取数据的 GDBMS,索引可在绝大多数情况下有效消除了磁盘 IO.比如:PG-Storm 由于要从磁盘读取数据,且无法改变 PostgreSQL 的存储引擎,因此选择用 BRIN-index 减少查询需要传输到 GPU 的数据量[81].

使用全内存存储的内存数据库中,索引查询成为新的性能瓶颈.文献[96]在内存 KV 系统中使用 GPU 加速索引查询来消除内存 KV 主要的性能瓶颈,提出一种驻留 GPU 显存的哈希表作为索引结构.此外,设计良好的索引结构也对连接算子的性能产生巨大的影响[42,48].在算法加速层面,一些 Join 算法[42,48]为了增加公平性,通过手动添加索引的方式实现了 GPU-Join 算子,并对比 CPU 版本取得了 10 倍的加速比.文献[15]在单个算子粒度下考虑索引读取数据(index scan)是否是对整个查询有利,并提出寻找性能均衡点(break-even point),对于是否走索引的优化决策起到关键作用.表 2 中列出了现有对 GPU 索引相关的研究工作及突出贡献.

Table 2 Research status of GPU index

表 2 GPU 索引研究现状

索引类型	实现细节	性能(每秒次查询,M=百万)
radix trees[97]	支持点查询和范围查询	100M~130M 点查询 600M~1000M 范围查询
CSS-Tree[42]	针对 NLJ 优化索引,使用定长且 cache 优化的数组	N/A
HB-Tree(异构 B+树)[98]	动态使用 CPU-GPU 异构计算资源,使用负载均衡策略	240M
FAST[99]	根据硬件特性自调整静态二叉树 (static binary-tree)结构,提升查询效率	50M CPU 85M GPU
R-Tree[61]	对多维空间数据进行查询	N/A
Hash index[96]	对 KV 数据存储进行索引	160+MOPS(KV get/set)
GPU LSM[100]	可更新日志结构树,对写优化	
GPU Btree[101]	实现可更新的 GPU B-Tree 索引	1000M 点查询 600M 范围查询

基于树的索引使用分层查找的策略,用尽可能访问少的内部节点而找到真正记录 tuple 的位置信息,进而将对整个数据空间的查找范围缩短为最多为树高的多个内部节点的查找.一般分为创建索引、索引搜索、更新索引这 3 个步骤.在索引查找步骤中,GPU 的实现方式重点在单个数组内找到特定值的指向位置,可安排线程块中每个线程记住内部节点的键,使用 map 原语进行比较,将结果用 reduce 原语统计,执行下一步查找.与 CPU 方案针对 Cache 块进行优化不同,GPU 中 shared memory 更大,对树节点大小要求不高,因而可以在 GPU 上实现树高更低的索引结构,加上 GPU 大规模并发计算能力的贡献,GPU 索引查询效率更高.FAST[99]通过根据硬件架构特性(页大小、cache 块、SIMD 指令位宽等)自调整节点大小,使用软件流水线、数据预取等技术手段,在查询计算的同时预取下一层节点的方式隐藏访存延迟,并尝试用压缩技术提升整体性能.研究表明:在小节点树上,CPU 的性能更高;而对大节点树上,GPU 性能更高,但是相对于 GPU 的峰值计算能力,索引计算的效果不佳.HB-Tree[98]则通过在 CPU-GPU 之间提供复杂均衡策略,利用内存和显存存储索引树结构,解决了索引树体积超过 GPU 显存的问题,取得了较好的索引查询性能,并支持批量更新索引操作.

以上使用 GPU 加速索引操作的研究取得了可喜的成果,但 GDBMS 普遍将数据存储在内存中,避免了传统数据库中最影响性能的磁盘 IO 瓶颈,因而索引是否是必要的模块有待研究.另一方面,OLAP 业务涉及数据量大,在数据获取上多采用全表扫描、全物化策略执行查询,这都进一步减少了索引存在的必要性.但是对于 OLTP 业务,单个事务的数据获取量少,GPU 索引就能发挥应有的作用.未来,GPU 加速索引查询的研究方向将围绕可更新索引、加速 Join 算子、高效空间数据索引这 3 个方向展开.

5.4 小 结

异构存储体系下 GPU 数据的存储管理、压缩和索引是 GDBMS 存储管理器的核心功能,一方面要高效利用“小而快”的 GPU 显存降低查询响应时延提高吞吐量;另一方面,又要利用 SSD、硬盘存储空间大、可持久存储的特点,增大 GDBMS 能够处理数据的总量和高可用性,同时还要尽可能减少 PCIe 上的数据迁移.

6 总 结

近 10 年间,尤其是 CUDA\OpenCL 异构统一计算语言的提出,使得 GPU 成为数据库可用的强大的可编程的通用协处理器,并涌现了一大批面向复杂查询的商用 GDBMS 或研究原型,这必将深刻改变高性能数据库系统的格局.数据库系统作为数据密集型系统应用软件,其性能的提升依赖于数据获取能力和计算能力的分别大规模提升:前者的提升依赖于大容量内存计算、分布式技术、SSD 磁盘阵列、NVM[102]新型存储介质以及与之配套的高性能数据结构和算法的开发;后者则受到 CPU 摩尔定律制约而发展缓慢.GDBMS 的兴起和发展,证明了通用 GPU 的大规模线程并发计算模式在关系型数据处理上是可行的和高效的,为数据库计算能力取得突破性进展提供了一种新的可能.

尽管如此,GPU 并不是专为关系型数据处理而开发,其硬件体系结构还有很多不适应 GDBMS 发展的地方,未来需要在如下几个方面进一步研究.

- 对于 GDBMS 查询编译器,一方面,以代码即时生成 JIT 技术和 LLVM 编译技术为依托,进一步压缩 SQL 的编译时间,是未来 GDBMS 查询编译器的研究方向.同时,如何充分利用不同架构、不同厂商 GPU 功能特性的同时,保证 SQL 编译器的稳定高效,也是研究热点之一.另一方面,在“一次一算子”编译模式下,以更多粒度批量编译查询请求,为不同规模的查询需求编译出规模适中的查询计划或多个查询计划的组合,避免因资源限制导致的查询失败,提高系统的稳定性,是 GDBMS 编译器在数据处理模式上的新挑战.同时,考虑数据所在位置(是否在 GPU 显存上)和系统运行状态生成“动态”的查询计划,是未来的研究热点之一.总之,GDBMS 查询编译器主要面临压缩异构查询计划的编译时间和根据不同数据查询规模、存储位置而生成高效查询计划两方面的挑战;
- 在 GDBMS 查询处理器领域,数据库算法的 GPU 改造将成为未来主要的研究方向,尤其是以 Join 为代表的复杂关系算子的 GPU 高效实现,将成为 GDBMS 性能提升的关键.此外,通过与查询编译器和查询优化器协同,KBE 对核函数的融合与切分的智能化、动态化乃至跨 GPU 改造,将成为 GDBMS 执行引擎的研究热点之一.而在功能上,GDBMS 对 OLTP 业务支撑离不开对事务 ACID 属性的支持,这将成为 GDBMS 亟待突破的难点,也成为最有潜力的研究方向之一;
- GDBMS 查询优化器的研究,将逐渐从以算子为中心到以查询计划树(QEP)为核心转变.以算子为单位的查询优化在异构查询代价模型、算子选择率、算子分配问题上取得了阶段性成果,尤其是以机器学习方法估计算子查询代价的 HyPE 框架的提出,给 GDBMS 查询优化器的设计提供了范本.但算子层次的优化缺乏全局视野,无法保证生成最优的查询计划.未来,以降低整个查询计划树的异构执行代价为目标的全局优化方案,是 GDBMS 查询优化器的研究重点.此外,多表连接的最佳顺序问题,仍是 GDBMS 优化器未解难题之一,其动态规划解决方案尚没有 GPU 版本的算法;
- 在 GDBMS 存储管理器方面,列式存储、全 GPU 计算、内存数据驻留带来的超高的计算性能仍是 GDBMS 的基石,但固态硬盘 SSD 的引入十分必要,在高速度和大容量之间的权衡,将是 GDBMS 存储引擎设计的主要问题.未来,在降低性能前提下提升存储容量,将成为 GDBMS 存储引擎未来的研究方向,SSD 与 GPU 数据直连、跨 GPU 分布式存储等技术非常有潜力.在数据压缩方面,研究降低解压缩成本为核心的轻量级 GPU 数据压缩算法,比更高压缩比的通用 GPU 压缩算法对 GDBMS 系统来说更为重要.对于 GPU 索引的研究,算法层面将向减少全局数据同步点、面向更新的数据结构、降低 PCIe 瓶颈方向努力;而在系统层面,将继续探索 GPU 索引技术在 GDBMS 中的应用场景.

可以预见:随着 GPU 性能随摩尔定律而提升和 GDBMS 四大核心模块技术的发展,GDBMS 一定会在技术上越来越成熟、性能优势越来越明显,从而在根本上改变当下数据处理系统软件的格局.

References:

[1] Owens JD, Luebke D, Govindaraju N, Harris M, Krüger J, Lefohn AE, Purcell TJ. A survey of general—Purpose computation on graphics hardware. Computer Graphics Forum, 2010,26(1):80−113. [doi: 10.1111/j.1467-8659.2007.01012.x]

[2] Fred P. New microarchitecture challenges in the coming generations of cmos process technologies. In: Proc. of the IEEE Computer Society1730 Massachusetts Ave. DCUnited States, 1999. 1–46. [doi: 10.1109/MICRO.1999.10004]

[3] Etiemble D. 45-Year CPU evolution: One law and two equations. In: Proc. of the 2nd Workshop on Pioneering Processor Paradigms. Vienne: CoRR, 2018. http://arxiv.org/abs/1803.00254

[4] OmniSci. Omnisci Technical White Paper: GPU-Accelerated Analytics-Big Data Analytics at Speed and Scal. 2020. https://www.omnisci.com/platform

[5] Root C, Mostak T. Mapd: A GPU-powered big data analytics and visualization platform. In: Proc. of the ACM SIGGRAPH 2016 Talks. Anaheim: ACM, 2016. 1–2. [doi: 10.1145/2897839.2927468]

[6] Broneske D, Breß S, Heimel M, Saake G. Toward hardware-sensitive database operations. In: Proc. of the 17th Int'l Conf. on Extending Database Technology (EDBT). Athens, 2014. [doi: 10.5441/002/edbt.2014.22]

[7] Breß S, Siegmund N, Bellatreche L, Saake G. An operator-stream-based scheduling engine for effective GPU coprocessing. In: Proc. of the 17th East European Conf. (ADBIS 2013). Berlin, Heidelberg: Springer-Verlag, 2013. 288–301. [doi: 10.1007/978-3-642-40683-6_22]

[8] Breß S, Heimel M, Siegmund N, Bellatreche L, Saake G. Exploring the design space of a GPU-aware database architecture. In: Proc. of the New Trends in Databases and Information Systems, Advances in Intelligent Systems and Computing. Switzerland: Springer Int'l Publishing, 2014. 225–234. [doi: 10.1007/978-3-319-01863-8_25]

[9] Breß S, Beier F, Rauhe H, Sattler KU, Schallehn E, Saake G. Efficient co-processor utilization in database query processing. Information Systems, 2013,38(8):1084–1096. [doi: 10.1016/j.is.2013.05.004]

[10] 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]

[11] Breß S, Siegmund N, Heimel M, Saecker M, Lauer T, Bellatreche L, Saake G. Load-Aware inter-co-processor parallelism in database query processing. Data Knowledge Engineering, 2014,93:60–79. [doi: 10.1016/j.datak.2014.07.003]

[12] Breß S, Saake G. Why it is time for a hype: A hybrid query processing engine for efficient GPU coprocessing in DBMS. Proc. of the VLDB Endowment, 2013,6(12):1398–1403. [doi: 10.14778/2536274.2536325]

[13] Breß S, Kocher B, Heimel M, Markl V, Saecker M, Saake G. Ocelot/hype: Optimized data processing on heterogeneous hardware. Proc. of the VLDB Endowment, 2014,7(13):1609–1612. [doi: 10.14778/2733004.2733042]

[14] Breß S, Funke H, Teubner J. Robust query processing in co-processor-accelerated databases. In: Proc. of the 2016 Int'l Conf. on Management of Data. San Francisco: ACM, 2016. 1891–1906. [doi: 10.1145/2882903.2882936]

[15] Breß S, Beier F, Rauhe H, Schallehn E, Sattler KU, Saake G. Automatic selection of processing units for coprocessing in databases. In: Proc. of the 16th East European Conf. on Advances in Databases and Information Systems. Poznan: Springer-Verlag, 2012. 57–70. [doi: 10.1007/978-3-642-33074-2_5]

[16] Fang R, He B, Lu M, Yang K, Govindaraju NK, Luo Q, Sander PV. GPUQP: Query co-processing using graphics processors. In: Proc. of the 2007 ACM SIGMOD Int'l Conf. on Management of data. Beijing: ACM, 2007. 1061–1063. [doi: 10.1145/1247480.1247606]

[17] He B, Lu M, Yang K, Fang R, Govindaraju NK, Luo Q, Sander PV. Relational query coprocessing on graphics processors. ACM Trans. on Database Systems (TODS), 2009,34(4):1–39. [doi: 10.1145/1620585.1620588]

[18] Wang K, Zhang K, Yuan Y, Ma S, Lee R, Ding X, Zhang X. Concurrent analytical query processing with GPUs. Proc. of the VLDB Endowment, 2014,7(11):1011–1022. [doi: 10.14778/2732967.2732976]

[19] Yuan Y, Lee R, Zhang X. The yin and yang of processing data warehousing queries on GPU devices. VLDB Journal, 2013,6(10): 817–828. [doi: 10.14778/2536206.2536210]

[20] 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 Int'l Workshop on Data Management on New Hardware. Chicago: ACM, 2017. 1–6. [doi: 10.1145/3076113.3076119]

[21] Yong KK, Ho WK, Chua MW, See S. A GPU query accelerator for geospatial coordinates computation. In: Proc. of the 2015 Int'l Conf. on Cloud Computing Research and Innovation (ICCCRI). IEEE Computer Society, 2015. 166–172. [doi: 10.1109/icccri.2015.26]

[22] Li J, Tseng HW, Lin C, Papakonstantinou Y, Swanson S. Hippogriffdb: Balancing I/O and GPU bandwidth in big data analytics. VLDB, 2016,9(14):1647−1658. [doi: 10.14778/3007328.3007331]

[23] Tu YC, Kumar A, Yu D, Rui R, Wheeler R. Data management systems on GPUs: Promises and challenges. In: Proc. of the 25th Int'l Conf. on Scientific and Statistical Database Management. Baltimore: ACM, 2013. 1–4. [doi: 10.1145/2484838.2484871]

[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: Association for Computing Machinery, 2017. Article 17. [doi: 10.1145/3085504.3085521]

[25] Zhang S, He J, He B, Lu M. Omnidb: Towards portable and efficient query processing on parallel CPU/GPU architectures. Proc. of the VLDB Endowment, 2013,6(12):1374−1377. [doi: 10.14778/2536274.2536319]

[26] Angstadt K, Harcourt E. A virtual machine model for accelerating relational database joins using a general purpose GPU. In: Proc. of the Symp. on High PERFORMANCE Computing. 127−134. [doi: https://hgpu.org/?p=13546]

[27] Bakkum P, Chakradhar S. Efficient Data Management for GPU Databases. Nec: Princeton, 2012. [doi: http://hgpu.org/?p=7180]

[28] Bakkum P, Skadron K. Accelerating SQL database operations on a gpu with cuda. In: Proc. of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units. Pittsburgh: ACM, 2010. 94−103. [doi: 10.1145/1735688.1735706]

[29] He B, Yu JX. High-Throughput transaction executions on graphics processors. Proc. of the VLDB Endowment, 2011,4(5):314−325. [doi: 10.14778/1952376.1952381]

[30] Breß S, Heimel M, Siegmund N, Bellatreche L, Saake G. GPU-Accelerated database systems: Survey and open challenges. In: Proc. of the Trans. Large-Scale Data- and Knowledge-Centered Systems. Berlin Heidelberg: Springer-Verlag, 2014. 1−35. [doi: 10.1007/978-3-662-45761-0_1]

[31] Greer R. Daytona and the foruth-generation language cymbal. In: Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD'99). Philadelphia: ACM, 1999. 525−526. [doi: 10.1145/304182.304242]

[32] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. of the Int'l Symp. on Code Generation and Optimization: Feedback-Directed and Runtime Optimization. Palo Alto: IEEE Computer Society, 2004. 75. [doi: 10.1109/CGO.2004.1281665]

[33] Wu J, Belevich A, Bendersky E, Heffernan M, Leary C, Pienaar J, Roune B, Springer R, Weng X, Hundt R. GPUCC: An open-source GPGPU compiler. In: Proc. of the 2016 Int'l Symp. on Code Generation and Optimization. Barcelona: ACM, 2016. 105−116. [doi: 10.1145/2854038.2854041]

[34] Nugteren C, Corporaal H. Bones: An automatic skeleton-based c-to-cuda compiler for GPUs. ACM Trans. on Archit. Code Optim., 2014,11(4):1−25. [doi: 10.1145/2665079]

[35] Saecker M, Saecker M, Pirk H, Manegold S, Markl V. Hardware-Oblivious parallelism for in-memory column-stores. Proc. of the VLDB Endowment, 2013,6(9):709−720. [doi: 10.14778/2536360.2536370]

[36] Abadi DJ, Madden SR, Hachem N. Column-Stores vs. Row-stores: How different are they really? In: Proc. of the 2008 ACM SIGMOD Int'l Conf. on Management of Data. Vancouver: ACM, 2008. 967–980. [doi: 10.1145/1376616.1376712]

[37] Boncz P. Monet, a next-generation DBMS kernel for query-intensive applications [Ph.D. Thesis]. Amsterdam: Universiteit van Amsterdam, 2002. 1−180. https://dare.uva.nl/search?identifier=aa4f797f-853d-4a75-8383-6d6a60948fac

[38] Boncz PA, Zukowski M, Nes NJ. Monetdb/x100: Hyper-pipelining query execution. In: Proc. of the 2005 CIDR Conf. Asilomar: Very Large Data Base Endowment (VLDB), 2005. 225–237.

[39] 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]

[40] Paul J, He J, He B. 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]

[41] Diamos GF, Wu H, Lele A, Wang J. Efficient relational algebra algorithms and data structures for GPU. In: Proc. of the CERS. Georgia, 2012.

[42] He B, Yang K, Fang R, Lu M, Govindaraju N, Luo Q, Sander P. Relational joins on graphics processor. In: Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD 2008). Vancouver: ACM, 2008. 511−524. [doi: 10.1145/1376616.1376670]

[43] He B, Govindaraju NK, Luo Q, Smith B. Efficient gather and scatter operations on graphics processors. In: Proc. of the 2007 ACM/IEEE Conf. on Supercomputing (SC 2007). ACM, 2007. 1−12. [doi: 10.1145/1362622.1362684]

[44] Arkhipov DI, Wu D, Li K, Regan AC. Sorting with GPUs: A survey. 2017. https://arxiv.org/abs/1709.02520

[45] Govindaraju N, Gray J, Kumar R, Manocha D. Gputerasort: High performance graphics co-processor sorting for large database management. In: Proc. of the 2006 ACM SIGMOD Int'l Conf. on Management of Data. Chicago: ACM, 2006. 325−336. [doi: 10.1145/1142473.1142511]

[46] 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 IEEE 29th Int'l Conf. on Data Engineering (ICDE 2013). 2013. 362−373. [doi: 10.1109/ICDE.2013.6544839]

[47] Angstadt K, Harcourt E. A virtual machine model for accelerating relational database joins using a general purpose GPU. In: Proc. of the Symp. on High Performance Computing. Alexandria: Society for Computer Simulation Int'l, 2015. 127−134. https://hgpu.org/?p=13546

[48] Yabuta M, Nguyen A, Kato S, Edahiro M, Kawashima H. Relational joins on GPUs: A closer look. IEEE Trans. on Parallel and Distributed Systems, 2017,28(9):2663−2673. [doi: 10.1109/TPDS.2017.2677451]

[49] He J, Lu M, He B. Revisiting co-processing for hash joins on the coupled CPU-GPU architecture. VLDB, 2013,6(10):889−900. [doi: 10.14778/2536206.2536216]

[50] D. Xiangwu, C. Jinxin. Optimizing parallel join of column-stores on heterogeneous computing platforms. In: Proc. of the 2016 IEEE Information Technology, Networking, Electronic and Automation Control Conf. 2016. 621−625. [doi: 10.1109/ITNEC.2016.7560435]

[51] Zhou G. Parallel cube computation on modern CPUs and GPUs. Journal of Supercomputing, 2012,61(3):394−417. [doi: 10.1007/s11227-011-0575-7]

[52] Lauer T, Datta A, Khadikov Z, Anselm C. Exploring graphics processing units as parallel coprocessors for online aggregation. In: Proc. of the ACM Int'l Workshop on Data Warehousing and OLAP. ACM, 2010. 77−84. [doi: 10.1145/1871940.1871958]

[53] Zhang Y, Zhang YS, Chen H, Wang S. GPU adaptive hybrid OLAP query processing model. Ruan Jian Xue Bao/Journal of Software, 2016,27(4):1246−1265 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4828.htm [doi: 10.13328/j.cnki.jos.004828]

[54] Kaczmarski K. Comparing GPU and CPU in olap cubes creation. In: Proc. of the Int'l Conf. on Current Trends in Theory and Practice of Computer Science. Nový Smokovec, 2011. 308−319. [doi: 10.5555/1946370.1946396]

[55] Dojchinovski D, Gusev M, Zdraveski V. Efficiently running SQL queries on GPU. In: Proc. of the 2018 26th Telecommunications Forum (TELFOR). 2018. 1−4. [doi: 10.1109/TELFOR.2018.8611821]

[56] Shanbhag A, Pirk H, Madden S. Efficient top-*k* query processing on massively parallel hardware. In: Proc. of the 2018 Int'l Conf. on Management of Data. Houston: ACM, 2018. 1557−1570. [doi: 10.1145/3183713.3183735]

[57] Pandey V, Kipf A, Neumann T, Kemper A. How good are modern spatial analytics systems? Proc. of the Vldb Endowment, 2018, 11(11):1661−1673. [doi: 10.14778/3236187.3236213]

[58] Güting RH. Geo-Relational Algebra: A Model and Query Language for Geometric Database Systems. Berlin Heidelberg: Springer, 1988.

[59] You S, Zhang J, Gruenwald L. High-Performance polyline intersection based spatial join on GPU-accelerated clusters. In: Proc. of the 5th ACM SIGSPATIAL Int'l Workshop on Analytics for Big Geospatial Data. Burlingame: ACM, 2016. 42−49. [doi: 10.1145/3006386.3006390]

[60] Jacox EH, Samet H. Spatial join techniques. ACM Trans. on Database Systems, 2007,32(1):7. [doi: 10.1145/1206049.1206056]

[61] You S, Zhang J, Gruenwald L. Parallel spatial query processing on GPUs using R-trees. In: Proc. of the 2nd ACM SIGSPATIAL Int'l Workshop on Analytics for Big Geospatial Data (BigSpatial 2013). 2013. 23−31. [doi: 10.1145/2534921.2534949]

[62] Zacharatou ET, Doraiswamy H, Ailamaki A, Silva CT, Freiref J. GPU rasterization for real-time spatial aggregation over arbitrary polygons. Proc. of the VLDB Endowment, 2017,11(3):352−365. [doi: 10.14778/3157794.3157803]

[63] Doraiswamy H, Vo HT, Silva CT, Freire J. A GPU-based index to support interactive spatio-temporal queries over historical data. In: Proc. of the IEEE Int'l Conf. on Data Engineering. Helsinki: IEEE, 2016. [doi: 10.1109/ICDE.2016.7498315]

[64] Nouri Z, Tu YC. GPU-Based parallel indexing for concurrent spatial query processing. In: Proc. of the 30th Int'l Conf. on Scientific and Statistical Database Management. Bozen-Bolzano: Association for Computing Machinery, 2018. Article 23. [doi: 10.1145/3221269.3221296]

[65] Doraiswamy H, Freire J. A GPU-friendly geometric data model and algebra for spatial queries. In: Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. Portland: Int'l Foundation for Autonomous Agents and Multiagent Systems, 2020. 1875−1885. [doi: 10.1145/3318464.3389774]

[66] Wu H, Diamos G, Cadambi S, Yalamanchili S. Kernel weaver: Automatically fusing database primitives for efficient GPU computation. In: Proc. of the IEEE/ACM Int'l Symp. on Microarchitecture. Vancouver: IEEE, 2012. 107−118. [doi: 10.1109/MICRO.2012.19]

[67] Menon P, Mowry TC, Pavlo A. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. Proc. of the VLDB Endowment, 2017,11(1):1−13. [doi: 10.14778/3151113.3151114]

[68] Zhong J, He B. Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling. IEEE Trans. on Parallel & Distributed Systems, 2014,25(6):1522−1532. [doi: 10.1109/TPDS.2013.257]

[69] 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 and Software. Austin: IEEE, 2011. 134−144. [doi: 10.1109/ISPASS.2011.5762730]

[70] Breß S, Geist I, Schallehn E, Mory M, Saake G. A framework for cost based optimization of hybrid CPU/GPU query plans in database systems. Control & Cybernetics, 2012,41(4):715−742.

[71] Schaa D, Kaeli D. Exploring the multiple-GPU design space. In: Proc. of the IEEE Int'l Symp. on Parallel&distributed Processing. Rome: IEEE, 2009. 1−12. [doi: 10.1109/IPDPS.2009.5161068]

[72] Akdere M, Çetintemel U, Riondato M, Upfal E, Zdonik SB. Learning-Based query performance modeling and prediction. In: Proc. of the 2012 IEEE 28th Int'l Conf. on Data Engineering (ICDE 2012). IEEE Computer Society, 2012. 390−401. [doi: 10.1109/icde.2012.64]

[73] Heimel M, Kiefer M, Markl V. Self-Tuning, GPU-accelerated kernel density models for multidimensional selectivity estimation. In: Proc. of the 2015 ACM SIGMOD Int'l Conf. on Management of Data. Melbourne: ACM, 2015. 1477−1492. [doi: 10.1145/2723372.2749438]

[74] Zhang J, You S, Gruenwald L. Parallel selectivity estimation for optimizing multidimensional spatial join processing on GPUs. In: Proc. of the 2017 IEEE 33rd Int'l Conf. on Data Engineering (ICDE). 2017. 1591−1598. [doi: 10.1109/ICDE.2017.236]

[75] Pirk H, Manegold S, Kersten ML. Accelerating foreign-key joins using asymmetric memory channels. In: Proc. of the 2nd Int'l Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS 2011) in Conjunction with VLDB 2011. VLDB, 2011. https://dare.uva.nl/personal/pure/en/publications/accelerating-foreignkey-joins-using-asymmetric-memory-channels(bd88a910-2b4f-4e7d-9743-656b56c63d41).html

[76] Leis V, Gubichev A, Mirchev A, Boncz P, Kemper A, Neumann T. How good are query optimizers, really? Proc. of the VLDB Endowment, 2015,9(3):204−215. [doi: 10.14778/2850583.2850594]

[77] Han WS, Kwak W, Lee J, Lohman GM, Markl V. Parallelizing query optimization. Proc. of the VLDB Endowment, 2008,1(1):188−200. [doi: 10.14778/1453856.1453882]

[78] Meister A, Saake G. Challenges for a GPU-accelerated dynamic programming approach for join-order optimization. In: Proc. of the 28th GIWorkshop GvDB. Berlin, 2016. 86−91. http://ceur-ws.org/Vol-1594/paper16.pdf

[79] Breß S, Mohammad S, Schallehn E. Self-Tuning distribution of DB-operations on hybrid CPU/GPU platforms. In: Proc. of the Grundlagen von Datenbanken. 2012. 89−94.

[80] Karnagel T, Habich D, Lehner W. Local vs. global optimization: Operator placement strategies in heterogeneous environments. In: Proc. of the Workshops of the EDBT/ICDT 2015 Joint Conf. 2015. 48−55. https://pdfs.semanticscholar.org/7cd1/50c5e35a65854572159d14b6dab42da0df0e.pdf

[81] Pg-strom v2.0 release technical brief. 2018. 1−46. https://heterodb.github.io/pg-strom/

[82] Brytlyt White Paper: Speed of Thought Analytics at Scale. 2019. https://www.brytlyt.com/

[83] Huang Y, Dong L. Performance modeling for optimal data placement on GPU with heterogeneous memory systems. In: Proc. of the 2017 IEEE Int'l Conf. on Cluster Computing (CLUSTER). Honolulu: IEEE, 2017. [doi: 10.1109/CLUSTER.2017.42]

[84] Bergman S, Brokhman T, Cohen T, Silberstein M. Spin: Seamless operating system integration of peer-to-peer dma between SSDS and GPUs. ACM Trans. on Comput. Syst., 2019,36(2):1–26. [doi: 10.1145/3309987]

[85] Meraji S, Schiefer B, Pham L, Chu L, Kokosielis P, Storm A, Young W, Ge C, Ng G, Kanagaratnam K. Towards a hybrid design for fast query processing in DB2 with BLU acceleration using graphical processing units: A technology demonstration. In: Proc. of the 2016 Int'l Conf. on Management of Data. San Francisco: ACM, 2016. 1951–1960. [doi: 10.1145/2882903.2903735]

[86] Sqream DB Architecture Whitepaper: A Next Generation Gpu Powered Analytics SQL Database. 2019. https://sqream.com/

[87] Arunkumar A, BolotinE, Nellans D, Wu CJ. Understanding the future of energy efficiency in multi-module GPUs. In: Proc. of the 2019 IEEE Int'l Symp. on High Performance Computer Architecture (HPCA). 2019. [doi: 10.1109/HPCA.2019.00063]

[88] Mittal S, Vetter JS. A survey of architectural approaches for data compression in cache and main memory systems. IEEE Trans. on Parallel & Distributed Systems, 2016,27(5):1524–1536. [doi: 10.1109/TPDS.2015.2435788]

[89] Stonebraker M, Abadi DJ, Batkin A, Chen X, Cherniack M, Ferreira M, Lau E, Lin A, Madden S, O'Neil E, O'Neil P, Rasin A, Tran N, Zdonik S. C-Store: A column-oriented DBMS. In: Proc. of the 31st Int'l Conf. on Very Large Data Bases. Trondheim: VLDB Endowment, 2005. 553–564. [doi: 10.1145/3226595.3226638]

[90] Abadi DJ. Integrating compression and execution in column-oriented database systems. In: Proc. of the 2006 ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD 2006). 2006. 671–682. [doi: 10.1145/1142473.1142548]

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

[92] Nelson M, Sorenson Z, Myre JM, Sawin J, Chiu D. GPU acceleration of range queries over large data sets. In: Proc. of the 6th IEEE/ACM Int'l Conf. on Big Data Computing, Applications and Technologies. Auckland: ACM, 2019. 11–20. [doi: 10.1145/3365109.3368789]

[93] Vijaykumar N. A case for core-assisted bottleneck acceleration in GPUs: Enabling flexible data compression with assist warps. In: Proc. of the Int'l Symp. on Computer Architecture. ACM, 2015. 41–53. [doi: 10.1145/2872887.2750399]

[94] Lee S. Warped-Compression: Enabling power efficient GPUs through register compression. In: Proc. of the Int'l Symp. on Computer Architecture. 2015. 502–514. [doi: 10.1145/2872887.2750417]

[95] Sathish V, Schulte MJ, Kim NS. Lossless and lossy memory I/O link compression for improving performance of GPGPU workloads. In: Proc. of the Int'l Conf. on Parallel Architectures and Compilation Techniques. Minneapolis: IEEE, 2017. 325–334. http://www.irisa.fr/alf/downloads/ADA/Sathish_LinkCompressionGPGPU_PACT12.pdf

[96] Zhang K, Wang K, Yuan Y, Guo L, Lee R, Zhang X. Mega-KV: A case for GPUs to maximize the throughput of in-memory key-value stores. Proc. of the VLDB Endowment, 2015,8(11):1226–1237. [doi: 10.14778/2809974.2809984]

[97] Alam M, Yoginath SB, Perumalla KS. Performance of point and range queries for in-memory databases using radix trees on GPUs. In: Proc. of the 2016 IEEE 18th Int'l Conf. on High Performance Computing and Communications; IEEE 14th Int'l Conf. on Smart City; IEEE 2nd Int'l Conf. on Data Science and Systems (HPCC/SmartCity/DSS). 2016. 1493–1500. [doi: 10.1109/HPCC-SmartCity-DSS.2016.0212]

[98] Shahvarani A, Jacobsen HA. A hybrid B+-tree as solution for in-memory indexing on CPU-GPU heterogeneous computing platforms. In: Proc. of the 2016 Int'l Conf. on Management of Data. San Francisco: ACM, 2016. 1523–1538. [doi: 10.1145/2882903.2882918]

[99] Kim C, Chhugani J, Satish N, Sedlar E, Nguyen AD, Kaldewey T, Lee VW, Brandt SA, Dubey P. Fast: Fast architecture sensitive tree search on modern CPUs and GPUs. In: Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD 2010). Indianapolis, 2010. 339–350. [doi: 10.1145/1807167.1807206]

[100] Ashkiani S, Li S, Farach-Colton M, Amenta N, Owens JD. GPU LSM: A dynamic dictionary data structure for the GPU. In: Proc. of the 2018 IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS). 2017. [doi: 10.1109/IPDPS.2018.00053]

[101] Awad MA, Ashkiani S, Johnson R, Farach-Colton M, Owens JD. Engineering a high-performance GPU B-tree. In: Proc. of the 24th Symp. on Principles and Practice of Parallel Programming. Washington: District of Columbia, Association for Computing Machinery, 2019. 145–157. https://doi.org/10.1145/3293883.3295706 [doi: 10.1145/3293883.3295706]

[102] Pan W, Li ZH, Du HT, Zhou CC, Su J. State-of-the-Art survey of transaction processing in non-volatile memory environments. Ruan Jian Xue Bao/Journal of Software, 2017,28(1):59−83 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5141.htm [doi: 10.13328/j.cnki. j0s.005141]

附中文参考文献:

[53] 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(4):1246−1265. http://www.jos.org.cn/1000-9825/4828.htm [doi: 10.13328/j.cnki.jos.004828]

[102] 潘巍,李战怀,杜洪涛,周陈超,苏静.新型非易失存储环境下事务型数据管理技术研究.软件学报,2017,28(1):59−83. http://www.jos.org.cn/1000-9825/5141.htm [doi: 10.13328/j.cnki.j0s.005141]

裴威(1988－),男,博士,讲师,CCF 学生会员,主要研究领域为数据库.

李战怀(1961－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为大数据管理技术,海量信息存储系统.

潘巍(1977－),男,博士,副教授,CCF 专业会员,主要研究领域为数据库理论与技术,大数据处理技术,分布式数据库,新型硬件数据管理.