计 算 机 研 究 与 发 展
Journal of Computer Research and Development

DOI: 10.7544/issn1000-1239.202111055
60(7): 1661−1677, 2023

HyperTree: 高并发 B+树索引加速器

吴婧雅[1,2] 卢文岩[1] 鄢贵海[1] 李晓维[1]

[1](处理器芯片全国重点实验室(中国科学院计算技术研究所) 北京 100190)

[2](中国科学院大学 北京 100049)

(wujingya@ict..ac.cn)

HyperTree: High Concurrent B+tree Index Accelerator

Wu Jingya[1,2], Lu Wenyan[1], Yan Guihai[1], and Li Xiaowei[1]

[1] (*State Key Lab of Processors*(*Institute of Computing Technology*, *Chinese Academy of Sciences*), *Beijing* 100190)

[2] (*University of Chinese Academy of Sciences*, *Beijing* 100049)

Abstract B+tree is the most commonly used index structure to improve query performance in relational databases. It maintains the order of key attributes in the database by building a balanced tree. The index improves database query performance but introduces index maintenance overheads, which include index updating, index deletion and index insertion, because of the high sequential order of the B+tree. Additionally, the performance of the B+tree will be further damaged by the huge amount of data under the circumstance of big data. Therefore, it is significant to balance index performance between the requirement of query and maintenance in B+tree systems while improving the index operation efficiency. To tackle these problems in big data applications, a domain-specific B+tree accelerator, called HyperTree, is proposed to boost the performance of B+tree operations including both index query and index maintenance. HyperTree co-optimizes both memory bandwidth and computation efficiency to fully utilize resources on the domain-specific accelerator. Specifically, to improve memory bandwidth utilization, the structures of the node and the B+tree are designed based on the characteristic of high bandwidth utilization in memory burst read/write; to improve computation parallelism and concurrency, multiple homogeneous computing engines and multiple data channels are configured to fully utilize hardware resources and reduce control overheads. In addition, to eliminate the block of index maintenance operations, a sub-tree system is proposed to uncouple the nodes in B+tree. The results of experiments show that compared with CPU, the FPGA-based B+tree accelerator achieves 6.84 times boosts in the throughput of query transactions and 29.14 times boosts in the throughput of index insertion.

Key words B+tree; FPGA; accelerator; high throughput; high concurrency; database query

摘 要 B+树是关系型数据库中用来加速查询的常用索引结构,通过构建平衡树维护关键属性的顺序.索引提升了数据库查询性能,但其严格的有序关系增加了数据库表的维护开销.特别是在大数据场景下,数据量激增使得索引查询和维序性能进一步下降.如何平衡B+树的查询和维序性能,以及在大数据场景下提升索引查询和维序的效率,对提升索引系统性能具有重要意义.由此设计了一种专用的B+树索引加速系统,对存储和计算进行协同优化,均衡提升索引查询和维序性能.利用内存突发读写高带宽的特性设计规则的树和节点存储格式以提升内存带宽利用效率,设计高效的同构计算架构和多数据通道以提升索引

收稿日期: 2021-10-27; **修回日期**: 2022-08-24

基金项目: 国家自然科学基金项目(62002340,61872336,61572470); 中国科学院青促会基金项目 (Y404441000)

This work was supported by the National Natural Science Foundation of China (62002340,61872336,61572470) and the Program of the Youth Innovation Promotion Association, CAS (Y404441000).

通信作者: 鄢贵海(yan@ict.ac.cn)

操作并行度. 同时设计解耦合的子树结构缓解索引维护时的树读写冲突. 实验结果表明, 相比于 CPU, B+树索引加速系统能够提升系统查询性能超过 6.84 倍, 提升索引维序性能提升超过 29.14 倍.

关键词 B+树; 现场可编程门阵列; 加速器; 高吞吐量; 高并发; 数据库查询

中图法分类号 TP302

索引是数据库中用来加速数据查询的一种常用数据结构. 其中, B+树凭借其独特优势成为关系型数据库中最常用的索引结构. 相较于其他索引结构, B+树具有 3 个优势:

1)数据存储结构规则. 利用关键字数据结构以平衡树维护, 能够提升数据查询效率. 且树的所有内部节点只存储索引关键字(即键值), 索引信息更密集, 降低查询 I/O 开销.

2)查询开销稳定. 关键字指向的实际数据只在叶子节点中存储. 由此, 查询时需要逐层遍历各层内部节点直到叶子节点结束, 查询开销稳定.

3)提升范围查询性能. 叶子节点以有序链表相连, 支持范围查询依序访问叶子节点, 避免对树的多次遍历, 使范围查询更高效.

然而, B+树索引的高度有序性使得大数据场景下索引查询效率降低、索引维序开销增加, 成为限制数据库性能提升的瓶颈. 总结大数据场景下 B+树性能提升面临的 3 个挑战:

1)访存性能低. 关键字查询通过逐层定位节点实现. 由于节点内关键字比较结果的不确定性导致对下一层子节点的访问是随机的. 这种随机内存访问使内存带宽利用异常低效.

2)并行性差. 由于查询需要逐层遍历节点才能完成, 中间结果的不确定性导致逐层遍历难以实现并行, 索引带来的优势被削弱.

3)树的维序难度大. 索引维序(索引删除、插入和修改, 其中 1 次修改可以看作是 1 次删除和 1 次插入的组合)过程会影响树中间各层节点键值的存储状态. 索引维序导致叶子节点内数据增加或减少, 并改变叶子节点内关键字顺序; 取决于节点内存储空间容量的限制, 也有可能导致中间节点结构的改变. 这种不确定的节点状态改变, 使叶子节点内关键字维序和中间节点更新复杂且效率低.

由此, 如何平衡 B+树的查询和维序性能, 以及在大数据场景下进一步提升索引的查询和维序效率, 成为索引优化的关键.

随着通用计算平台性能提升放缓[1], 许多研究提出利用 GPU 和 FPGA 等异构计算平台加速 B+树的查询和维护操作[2-4]. 为充分发挥 GPU 异构系统的算力, 多数研究从提升索引查询性能和缓解内存带宽瓶颈等角度提出解决方法. 在 SIMD 架构的平台中, 文献 [5−7] 提出了提升多条查询语句并行效率的策略以提升 GPU 对 B+树的查询效率. 文献 [8−9] 则优化了索引节点的存储结构, 以缓解 GPU 的内存带宽瓶颈. 这些研究将重点放在提升索引查询的性能. 利用 FPGA 这一类可重构计算架构, 文献 [10−11] 利用 FPGA 分别设计了支持索引的查询和更新操作的专用计算引擎, 并评估了利用 FPGA 加速不同体量索引对系统整体性能的影响. 但是文献 [10−11] 方法欠缺对索引查询和更新的协同优化, 没有提出 B+树索引系统的均衡优化方案.

为均衡提升大数据场景下数据库索引的查询和维护性能, 本文系统研究了 B+树索引的查询和维序的操作特性, 提出针对索引访存和计算的协同优化加速器架构 HyperTree, 在大数据场景中, 兼顾索引的查询和维序性能的提升. 本文的主要贡献点:

1)提出一种同构的流式计算多引擎架构, 既能够提升单计算引擎的效率, 又支持多索引查询、更新和维序任务的并行;

2)提出一种规则的 B+树节点存储格式, 同时设计专用的内存管理系统以支持低开销的并行访存, 进一步提升内存带宽利用效率;

3)提出一种高效灵活的计算控制系统, 支持多指令并行和多数据并行, 在提升计算引擎性能的同时, 不会引入额外开销;

4)提出一种多子树结构, 将复杂的 B+树拆分为几棵小的子树, 避免叶子节点的更新冲突, 以提升索引的维序效率.

实验结果表明, 相比于 CPU, B+树索引加速系统 HyperTree 能够提升系统查询性能 6.84 倍; 范围查询性能提升 14.53 倍, 且相比单值查询性能损失很小, 平均为单值查询性能的 75.58%; 引入子树存储管理系统后, 索引维序性能提升超过 29.14 倍.

1 研究背景

典型的 B+树结构如图 1 所示. 内部节点称为索引节点, 按照升序排列存储关键字; 叶子节点既存储

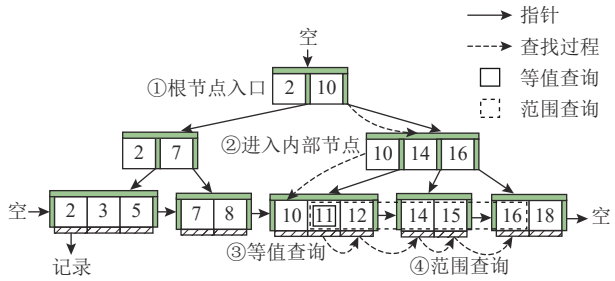


Fig. 1 Keyword search in B+tree

图 1 B+树的关键字查询

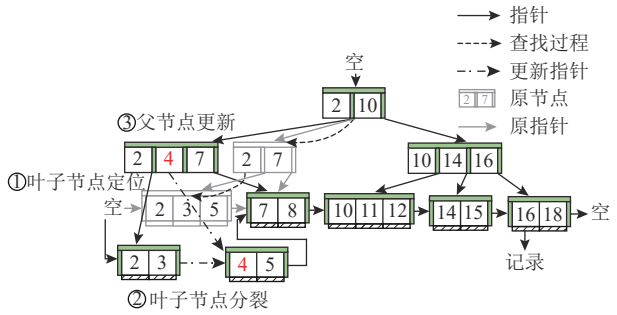


Fig. 2 Keyword insertion in B+tree

图 2 B+树的关键字插入

关键字又记录关键字指向的数据信息.

B+树查询操作从根节点开始，自顶向下逐层比较内部节点的索引关键字范围，直到找到包含查询内容的叶子节点. 以图 1 所示的等值查询目标关键字 11 为例. 从根节点开始，将 11 与关键字 2 和 10 比较，定位下一层节点应指向根节点最右侧子节点. 再重复上述范围比较，定位下一层应指向当前节点最左侧子节点. 此节点为 B+树的叶子节点，在叶子节点中定位关键字 11 指向的原数据库表，结束本次查询. 范围查询可以通过等值查询定位的叶子节点间的指针依次比较关键字得到查询结果.

B+树查询性能取决于树的高度(逐层查找)、节点大小(逐个数据比较)及关键字匹配速度. 例如：高度为 l 的 B+树所需查询时间为 l 次节点读取的 I/O 延时和 $m_i \times l$ 次计算引擎的关键字比较处理时间(m_i 为各节点关键字比较的次数). 由此，索引节点的 I/O 开销和关键字比较成为限制索引查询性能的瓶颈.

B+树的维序过程由数据库表的插入、删除和更新操作引起. 以索引插入为例，首先通过查询定位关键字待插入的叶子节点位置，然后插入关键字. 插入前需要判断插入新关键字后叶子节点是否超过规定长度. 若没有超过则结束本次索引插入，否则将该叶子节点平均分裂为 2 个新叶子节点，并用新分裂的节点信息更新其父节点. 更新时，将新分裂的右侧叶子节点的第 1 个索引关键字插入父节点原关键字，并更新叶子节点间指针. 叶子节点更新结束后，依层向上重复节点更新过程，直到不需要拆分父节点或找到根节点，结束索引插入. 删除与插入过程类似，区别在于删除可能导致节点合并，索引更新由删除和插入组成. 以图 2 所示插入关键字 4 为示例. 首先定位关键字 4 应写入的叶子节点位置. 此时插入新关键字后的叶子节点超过规定长度，将该叶子节点分裂为 2 个新节点，依大小顺序在父节点插入索引值 4，同时更新父节点指针. 更新后父节点长度未超过规定长度，结束更新过程.

数据表的更新，需要首先对叶子节点定位，再修改关键字. 由于节点存储空间的限制还有可能向上分裂或合并中间节点. 不确定的多层节点操作会引入随机 I/O 读写，使得数据表维序的开销相对于查询来说大大增加.

2 相关工作

在大数据时代，内存数据库的出现一定程度上解决了传统硬盘数据库的读写延迟问题[2]. 然而通用计算却不足以提供有效算力支持以充分利用内存读写带宽资源[1]. 研究人员转而利用异构计算平台，实现全卸载或半卸载的内存数据库加速方案提升数据库的计算效率[2-4].

为提升 B+树的查询效率，大量研究基于 GPU 异构计算平台设计高效的并行架构[2]，主要优化方向为加速索引查询[5-6,8]及索引结构优化[7,9,12]. GPU 加速索引操作的核心思想是改进查询的计算模式以充分适配 GPU 的 SIMD 架构. 文献 [5] 基于 AMD 的 CPU+GPU 异构系统，提出了粗细粒度 2 种并行策略以加速 B+树的查询. 细粒度并行策略将节点的二分查找替换为 K-ary 查找，从而使计算引擎执行相同模式的查询操作；粗粒度并行策略利用同构的计算引擎同时执行多条查询. 访存优化主要是调整 B+树内部节点为规则的数据结构，从而充分利用 GPU 的高访存带宽. 文献 [7] 设计了一种将索引关键字和子节点指针分布存储的改进树形结构，以充分利用 GPU 的内存带宽；并利用 SIMD 计算方法设计基于排序的并行查询方法，进一步缓解查询时的内存带宽瓶颈. 文献 [12] 优化节点内部存储结构，解决访存瓶颈，进一步提升查询并行计算效率.

充分发挥 GPU 的并行性能极强地依赖于计算规则且访存对齐[2]. 然而索引操作天然具有计算并行难度大、访存随机性等问题，导致利用 GPU 加速索引

操作的性能提升效果有限，从而有一些文献提出利用可重构硬件加速器设计更加灵活高效的方案. 文献 [10] 利用 FPGA 设计了加速 B+树索引查询的专用体系结构. 利用片上 BRAM 存储部分内部节点地址以达到快速访问内部节点的目的，同时设计专用并行查询计算单元，能够支持节点内关键字的快速定位；而叶子节点和一些低层内部节点的查询及全部索引维序依靠 CPU 实现. 文献 [11] 补充了文献 [10] 的研究工作，设计并行的节点更新计算单元，同时在 CPU 端增加 1 个调度单元部分卸载索引更新至 FPGA，即 FPGA 仅处理索引更新的部分操作，一定程度上提升了 FPGA 加速 B+树索引的效果. 文献 [10−11] 利用 FPGA 作为通用计算 CPU 的补充：CPU 负责数据和指令的卸载，将根节点和部分上层内部节点卸载到 FPGA 上加速查询和更新. 但这种方法是半卸载的方案，CPU 仍然需要负责主要的计算，FPGA 仅完成部分协同工作；且该方法的实现对 FPGA 的计算资源和存储空间利用效率很低，指令并行性差，B+树索引的性能提升有限. 此外，文献 [10−11] 将优化的 CBS+树结构作为数据库的索引格式，虽然降低数据传输开销及 FPGA 端地址计算复杂度，但给系统引入了额外的索引格式转换开销.

近年来，随着 FPGA 板卡内存空间和带宽性能的不断提升，利用 FPGA 加速内存数据库性能提升显著[2,4]. 鉴于 FPGA 结构更加灵活、资源可配置等特性，本文基于 FPGA 异构计算平台，设计并实现了全卸载 B+树索引加速系统 HyperTree——将索引的存储，B+树构建，索引查询，索引插入、删除和修改全部写到 FPGA 执行. 在本文所提出的 CPU-FPGA 构成的异构计算系统中，CPU 只负责查询计划的生成和优化以及加速器的启动运行和管理.

3 B+树性能瓶颈分析

基于第 1 节对 B+树典型操作的描述，本节将对 B+树各操作性能瓶颈进行深入分析.

3.1 数据随机访存造成存储带宽利用率低下

B+树查询时内存带宽利用非常低效：一是由于节点的随机访存；二是由于树的访存管理复杂. 在 B+树索引查询时，下层节点的访存地址由当前层节点内部关键字的比较结果决定. 结果的不确定性导致每层节点的访存是随机的，且中间节点在内存中依靠指针相连，每个节点的数据结构不完全相同，节点的物理地址也是随机的，进一步增加访存随机性. B+树的存储管理需要对内存空间动态维护. 当执行索引插入和删除操作时，可能需要动态申请或释放内存空间. 为维护节点的树形结构，需要动态修改对应的内存空间状态，该过程的存储访存也是随机的. 此外，数据量的增加导致实际应用中 B+树标准格式规定的节点存储空间无法完整保存 1 个数据表内的全部索引信息. 从而维护 1 个数据表索引需要管理多棵 B+树，进一步加剧了内存空间管理的困难. 这种数据访存随机性导致的问题无论是 CPU 还是 GPU 都无法直接解决，也是导致 GPU 的高访存带宽无法被直接有效利用的根本原因.

为验证随机内存访问带宽的性能，通过实验在实际数据库中测试 DDR4-3200 内存访存特性（其标称内存带宽为 204.8 Gbps）. 随着读写地址空间范围的增加，实际读写带宽有一定提升，但只有访存达到突发读写长度时，理论最大带宽才能被充分利用. 如图 3 的实验结果显示，单字读写、随机读写带宽实际只能将近标称性能的 60%，甚至更低.

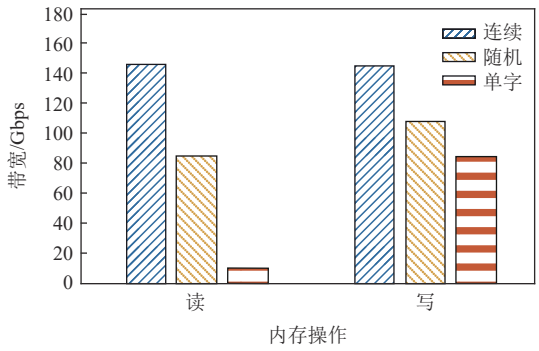


Fig. 3 Practical efficient read/write bandwidth of DDR4

图 3 DDR4 的实际有效读写带宽

3.2 查询并行度低下影响查询性能

单节点的查询和比较难以并行，是大数据量下查询性能下降的关键原因. 在 B+树设计中，为优化中间节点访存带宽，可以通过尽可能扩大中间节点容量（节点内数据连续存储）的方法实现. 然而，中间节点数据量的增加加剧了节点内关键字查找的难度. 有研究表明，利用 GPU 加速索引操作，即使是改进的算法，例如顺序查找、二分查找、并行查找，单条查询的性能提升仍然十分有限[6].

多查询语句并行是进一步提升系统性能的关键. 但现有的 CPU-GPU 异构系统由于无法在随机访存时充分利用内存带宽，且查询时计算结果随机性强，从而难以提升并行度[5-6,12]. 对于树的访存来说，有限的内存带宽无法满足多处理引擎对数据读写速度的需求；且每次访存的地址空间范围有限，进一步降低

读写效率. 对于计算来说，处理引擎的比较和查找能力决定了单条查询语句的效率，而处理引擎的数目决定了查询的并行度上限. 当单引擎的计算性能受限时，并行带来的性能提升十分有限.

3.3 更新维序复杂大大降低插入性能

索引提升了数据查询的效率，但树形结构的高度有序性使索引维序操作复杂. 修改数据库表记录，包括记录的增加、删除或更新，需要同时修改索引. 索引的维序对修改数据库表记录造成额外开销. 以索引的插入为例，受限于节点容量的限制，关键字插入可能引起节点分裂. 节点内数据量一旦超过容量阈值，节点的分裂可能自叶子节点出现，并向上逐级影响父节点直到根节点. 但这一过程是否出现取决于节点内原有关键字的容量和分裂后的节点关键字，难以提前预判计算和预取数据，导致索引插入的执行效率很低. 这种计算的不确定性对 GPU 的 SIMD 架构极不友好.

索引的维序会阻塞对同一棵树的其他操作，影响指令并行性能. 在本条维序操作结束前，节点内数据容量的差异和树形结构的数据依赖性导致无法预测其结构变化. 索引维序过程对正在执行的树具有独占性，即阻塞对同一棵树的其他操作，从而当索引维序请求发生时，无法实现多指令并行，进一步降低 B+树索引维序的性能.

4 B+树索引的优化

第 3 节的分析表明：访存效率低和计算并行难是造成 B+树查询性能差的主要原因；维序的复杂操作以及维序操作的数据依赖使得加速索引维序更加困难. 本节将从访存效率提升、单节点计算效率提升、计算并行优化及解除维序的数据依赖 4 个方面，提出提升 B+树查询和维序性能的设计思想.

访存优化的关键是利用主存 DRAM 突发读写带宽高的特性来提升带宽利用率. DRAM 存储器的特点是连续地址读写性能高效，即可以利用突发读写机制实现对 B+树各节点的读写. 在 B+树中，节点内在关键字比较时被反复访问，这也意味着 B+树中间节点数据量越大，数据读写效率越高. 因此，中间节点的数据量大小、存储格式以及读写方式是本文优化的重点.

然而单节点的数据量增加意味着在查找过程中需要进行更多次数据比较才能找到指向下一级满足条件的节点指针，对计算引擎的查询性能提出了更高的要求，从而提升单节点计算效率成为提高计算性能的首要保证. 为进一步提升查询性能，语句级并行是提升系统性能的另一个重点. 由此提出设计一种高性能的同构流式计算处理引擎架构以提升单查询语句的执行效率，并进一步提升多查询语句的并行度.

大节点引入使得索引维序的过程更加复杂. 一方面，定位待修改的关键字位置的查询过程延迟变长；另一方面，节点内数据量增加导致更高的数据修改开销. 上述 2 步骤的延迟和开销进一步加剧了索引维序导致的数据依赖，阻塞其他对同一棵树的索引操作. 因此，提出解除语句间数据依赖以提升索引维序性能的方法，其核心思想是将单棵 B+树拆分成多棵子树，能够支持对单棵 B+树操作转为对不同子树的操作. 由于子树间彼此独立，从而实现多插入更新操作并行. 具体地，我们采取了 5 个关键优化.

4.1 节点访存优化

为了提升节点数据的访存效率，首先要确定树节点的存储格式. 树节点存储格式设计需要解决 2 个核心问题：1）单节点的访存效率，节点大小要符合 DDR 控制器的特性；2）存储可扩展性设计满足不同大小树的存储需求.

设计节点大小的原则以 DDR 控制器所支持突发（burst）长度为基准，规定单个节点的大小为突发长度的整数倍. 比如 Xilinx FPGA 板卡突发传输为 512 B，可以设计节点大小为 512 B 或 1024 B 或其他 1 次突发读写大小的整数倍空间. 但一般来说，为降低单节点存储可能造成的存储空间浪费，单节点的设计不宜过大. 同时规定每次内存读写都以 1 个节点大小为单位，从而保证在对索引存储节点的每次数据读写都充分利用内存带宽.

为简化 B+树的存储管理，规定以内存块的方式维护单棵 B+树，一个完整的索引可以由单棵树或者多棵树构成，树的组织结构在内存块中完成. 每个内存块（即 1 棵树）包含多个节点，每个节点存储多个索引关键字. 多树的可扩展性设计的关键是进行树的动态维护. 规定内存空间的动态申请和回收也按照块进行，即创建新 B+树时，按块申请内存空间构建索引；删除 B+树，也相应地按块释放内存空间. 以内存块为最小访存管理对象，使得内存管理更加简洁高效.

4.2 高效的访存管理

确定了树的存储结构，接下来要解决的是如何实现处理引擎对树和节点的数据访问，即如何快速实现树和节点从逻辑编号到物理存储的映射，以及

如何高效地将内存数据加载到处理单元. 访存管理需要解决 2 个关键问题：1)逻辑树和节点到物理存储的映射，包括物理空间的申请和释放；2)为多处理引擎提供有效的计算数据.

规则的节点设计能够简化 B+树节点的地址空间管理. 定义一棵 B+树在内存中存储的数据格式为树表，每棵树表对应特定的内存块. 树表的最小数据组织单位为行，行大小一致，每行表示 1 个节点. 计算引擎直接对节点内的索引关键字进行操作，其访存过程首先确定树表的内存空间，然后定位节点的内存地址. 由于索引的查询和维序操作均在 1 个或多个节点内完成，为充分利用内存读写带宽，每次 I/O 读写均以 1 个节点为单位. 然而，由于索引维序时节点的动态修改对每个节点的地址管理仍然很复杂，需要解决新节点物理地址的申请或原节点物理空间的释放.

执行查询操作时，内存访问均以 B+树根节点开始，并依层次顺序访问内部节点，直到叶子节点结束. 为降低节点内存访问的控制成本，提出构建专用的树表存储管理单元(tree-table memory manage unit, TMMU)实现快速从 B+树逻辑地址到树表节点内存物理地址的映射，以降低内存空间维护的复杂度，并提升计算访存的效率. TMMU 记录 B+树和内部节点的相关状态信息，保证在树或节点创建或删除时，能够动态维护树和节点的有效性.

4.3 流式计算架构

利用流式计算的原则设计专用处理引擎从而提升节点内关键字查询的性能. 相比于其他批量计算，流式计算有 2 个主要的优势：1)无状态执行，控制简单，数据驱动有利于多任务并行；2)全流水执行，效率高，部分数据到达即可进行处理. 图 4 示例了流式计算与批量计算的区别. 与传统的批量计算不同，流式计算每次只对规定窗口大小的数据进行计算，而无需等待全部数据输入到计算节点. 图 4 示例表明，流式计算的控制简单，计算仅以数据流为驱动，没有复杂的控制逻辑结构.

4.4 多查询语句并行

规则的节点存储结构和高效的数据管理单元为多语句并行执行提供了快速有效的原始计算数据，解决了带宽与计算性能不匹配的问题. 数据管理单元配合内存控制单元，能够提供连续不间断的数据访存通路，为多计算引擎提供计算所需的数据.

实现多查询语句并发的另一个关键条件是指令间的调度和控制，即查询指令如何映射到多处理单元. 为最大化多语句并行度，并简化控制逻辑，提出

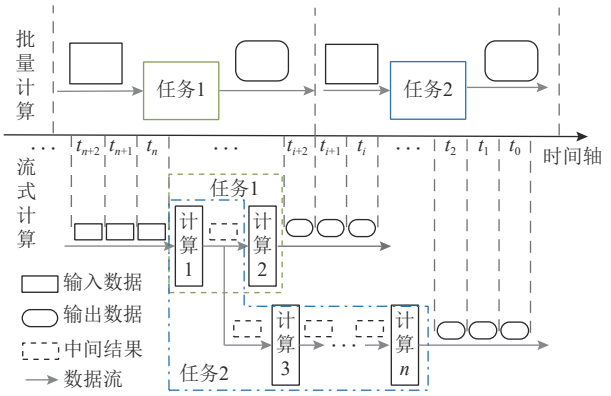


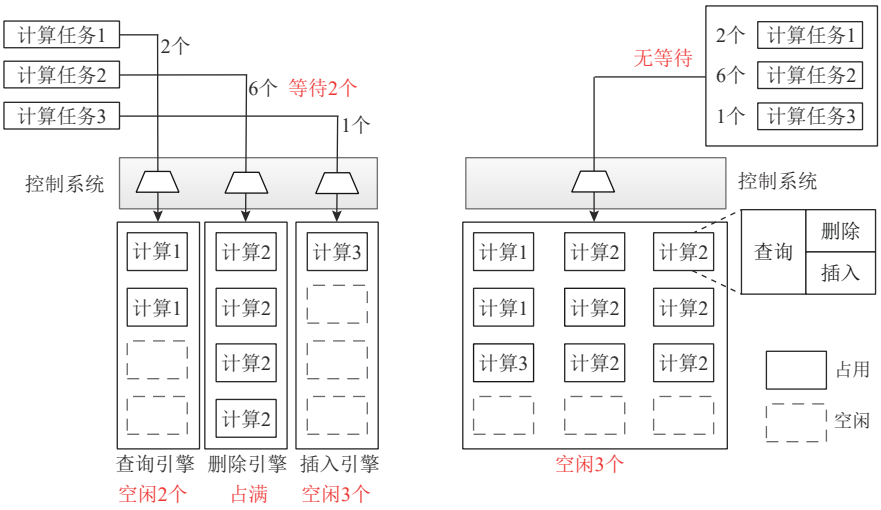
Fig. 4 Difference between streaming computing and batch computing

图 4 流式计算与批量计算的区别

同构的流式计算引擎阵列设计. 同构计算引擎，即所有处理单元架构一样，每个处理单元可以支持包括查询、删除、插入和更新在内的全部索引操作. 同时，由于各处理单元采用流式计算实现，进一步降低其控制逻辑的复杂度. 基于上述设计，多查询指令的并发管理变得异常简单高效，当有新的计算指令时，可以快速分配到任一空闲处理引擎执行，而不需要对各处理单元分类别管理. 同构的处理引擎设计也会避免计算任务和处理单元类型不匹配而导致资源浪费. 相比于如图 5 所示的异构设计，采用同构设计处理单元存在一定的冗余而使得硬件资源开销增加，我们也在降低开销上做了很多精细设计，详细见第 5 节.

4.5 提升索引维序性能的子树结构设计

为缓解索引维序的独占性而导致的性能损失，提出解耦合的子树结构设计，以解决由于索引维序导致的多语句并行阻塞现象. 子树是将 1 棵完整的 B+树按照一定的方式拆分为几棵小的子树，且规定每一棵子树的构建严格遵循 B+树索引的结构要求. 为支持子树查询和维序的并行，内存管理模块也需要进行相应的修改：查询和维序操作的起始访问从内存空间的入口地址(即大树的根节点)变为子树的根节点对应的内存地址. 具体地，创建新树时仍然申请 1 个新的内存块；但是在构建树的过程中，按照某种规则选择内存块的某些地址作为子树的根节点地址，并将树构建的过程平均至每一棵子树，直到完成 1 棵树的创建. 相应地，查询操作的起始地址将变为子树入口地址(即子树根节点)，而不再是原始的大树入口地址(根节点). 这样设计的优势在于，由于子树间没有数据依赖，对同一棵大树但不同子树的多条插入和更新操作可以并行执行，即实现将对 1 棵大树的串行操作转化为对多棵子树的并行操作.



(a) 异构计算引擎 (b) 同构计算引擎

Fig. 5 Difference between homogeneous and heterogeneous computing units

图 5 同构计算单元与异构计算单元的区别

子树的划分粒度是子树设计中需要考量的重要设计参数. 划分子树时, 划分粒度越细, 索引更新可以在更多棵子树中并行, 数据并行度增加, 从而提升计算并行效率. 但划分越细粒度, 子树根节点的入口地址维护会越复杂, 且划分越细将丢失越多索引结构信息, 对树的索引结构利用效率降低; 反之, 子树划分粒度越粗, 子树棵树越少, 数据依赖越多, 计算并行提升有限. 但粗粒度的划分方式能够更好地利用索引结构信息, 且子树控制逻辑简单, 能够缓解硬件资源开销.

5 系统实现

按照第 4 节中提出的优化策略, 本节主要介绍基于 FPGA 实现的 B+树索引加速系统 HyperTree 的具体实现和系统运行流程. HyperTree 统包含 3 个主要的子系统, 如图 6 所示: 存储管理子系统、索引计算子系统和指令控制子系统. 其中, 存储管理子系统包含 TMMU 和内存控制单元. 在得到激活控制信号后, TMMU 负责将指令中的逻辑编号转换为内存物理地址. 内存控制单元将 TMMU 输出的物理地址中的内存数据传递到计算引擎, 或按照计算引擎给出的结果写回相应的内存空间. 指令控制子系统负责解析索引操作指令, 提取操作码和操作数, 并根据操作码和操作数依次激活指令执行单元并打开内存数据通道, 负责指令并行控制. 索引计算子系统由多个同构的计算引擎构成, 负责并行执行索引的查询、插入、

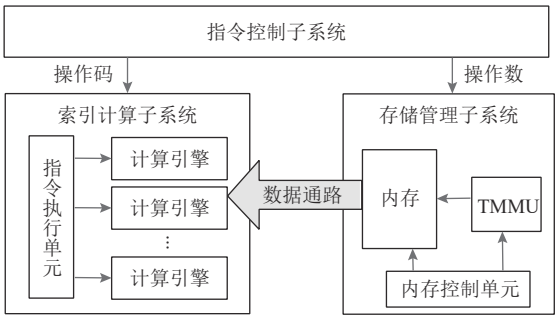


Fig. 6 System overall architecture

图 6 系统整体结构

删除和更新指令. 其中, 存储管理子系统和指令控制子系统是实现多语句并行的直接控制逻辑单元, 索引计算子系统是执行计算并行的核心功能部件. 接下来分别对各部分关键模块的实现进行介绍.

5.1 节点存储格式设计

根据第 4 节中节点的设计原则, 在微结构实现时以 512 B 为单位定义节点大小. 以 128 MB 内存块大小为单元设计树表, 则树表共包含 2^{18} 行(即节点), 在存储块内的标号为 0～2^{18}−1. 树的大小可以通过改变单位节点大小或树表容量相应调整. 给定固定容量的 B+树, 如果 1 棵树满, 可以通过动态申请存储块, 创建 1 棵新树以实现数据库表的完整索引结构. 该过程利用软件指令显式维护. B+树的节点格式规则定义如图 7 所示. B+树包含 3 种不同的节点类型: 内部节点、叶子节点和等值节点. 其中, 内部节点只存储关键字, 叶子节点和等值节点存储关键字和卫星数据(卫星数据指向数据表中的一条数据记录). 两者

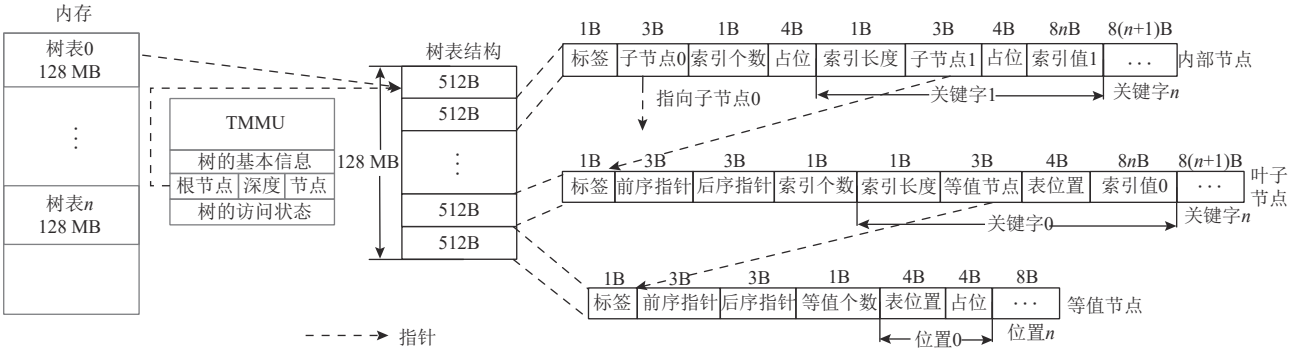


Fig. 7 Data structure definition of tree-table

图 7 树表的数据结构定义

的区别在于,叶子节点指向不同关键字对应的记录,而等值节点指向相同关键字对应的记录. 等值节点由叶子节点维护,叶子节点内的等值索引指针若为 0,则原始表位置信息有效,直接指向原始数据库表中的记录;否则,等值索引指向等值节点. 为简化索引关键字的寻址过程,设计 8B 对齐的节点存储格式. 3 类节点的格式组织基本相同,都是由节点头部和节点数据 2 部分组成. 节点头部主要标识节点类型及记录指针;数据部分主要存储索引关键字和指向子节点或表逻辑地址的指针.

规则的节点设计及以节点为单位的访存规则有效地提升了系统的内存带宽和内存空间的利用率. 以突发读写长度为单位设计节点容量,使计算引擎在处理节点内数据时,能够充分利用突发读写的高带宽,提升内存读写效率. 节点指针以逻辑地址的形式存储,所占内存空间更小,且标识节点特征的部分仅占节点头部的 1/4,提升了节点对索引关键字及指针的存储效率,确保最大程度利用内存空间. 此外,规则的节点格式设计简化了计算引擎对数据解析的过程. 在处理单节点内部信息时,进一步提升计算性能.

5.2 存储管理单元

为充分利用内存带宽,降低树表地址维护开销,提出专用树表存储管理单元 TMMU,其主要实现 2 部分功能:一是实现逻辑地址到物理地址的快速映射;二是高效地将数据传递到计算引擎.

快速的地址映射能够简化软件对内存地址的管理,并提升节点存储空间的存储效率. 在实际应用中,软件不直接管理 B+树的节点地址,从而维护索引物理地址被转换为硬件开销. 通常计算指令的操作数所携带的信息是 B+树的标识 *tree_ID*;根据树表的设计原则,所得到的节点指针也只存储其逻辑地址(以 *Num* 表示). TMMU 的主要任务之一就是实现 B+树的标识 *tree_ID* 及树内节点逻辑地址与 B+树根节点入口地址及节点物理地址的映射,并维护树表基本信息和节点的有效性. TMMU 的地址映射由图 8 所示.

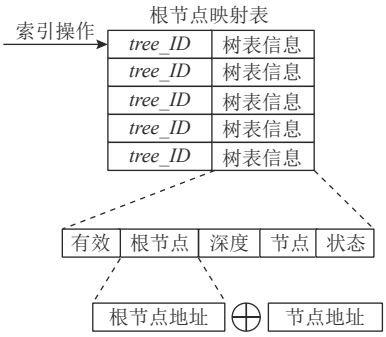


Fig. 8 Address mapping in TMMU

图 8 TMMU 的地址映射

TMMU 中维护地址映射,为充分利用缓存读写带宽,映射表的 1 行与缓存单次最大读写位宽相同. 表中的一行记录树的标识编号 *tree_ID*、根节点入口地址(记为 *Addr*)、树深度、节点个数等信息. 根节点映射表中以有效位 1 表示该条记录有效,否则该位置可以被改写. 当指令输入 *tree_ID*, TMMU 通过树表映射表确定树表的根节点入口及相应信息;再依据节点的逻辑地址,根据式(1)中给出的映射关系计算其物理地址,其中单位寻址空间的单位为 B. 节点逻辑地址到物理地址的映射计算简单,硬件计算资源开销很小.

$$\text{物理地址} = Addr + \frac{Num \times 512}{\text{单位寻址空间}}. \tag{1}$$

TMMU 可以实现对树表的动态维护. 新建或删除 B+树时,TMMU 根据对内存空间动态申请或释放的结果,在树表映射表中根据有效位信息增加或删除相应的 *tree_ID* 与树表信息的记录. 指令拆分或合并节点时,TMMU 负责修改树表中树的深度和节点个数信息. 为简化树表管理,规定增加节点时逻辑地址相应加 1,删除节点时不修改原有节点的逻辑地址,

下次增加节点仍然在现有逻辑地址的基础上递增. 这样的实现方式虽然造成节点删除时存储空间的浪费(这种空间损失代价很小，每一个节点仅占树表结构的 $1/2^{18}$)，但能够极大地简化树表管理的控制逻辑.

TMMU 的另一个主要功能是充分利用内存读写带宽，实现高效的内存读写. 随着指令控制单元依次解析指令队列中的指令内容，TMMU 接收到节点逻辑地址后立即计算物理地址，内存管理单元打开内存通道，将对应内存空间内的数据传递到指定的计算单元. TMMU 对数据的处理是连续的，即只要接收到指令流的逻辑地址，则执行物理地址的计算并打开数据通道，对内存块内的数据进行缓存. 该设计相当于实现了计算引擎的数据预取，消除计算时数据读写所引入的 I/O 访问延迟. TMMU 通过缓存机制实现内存的连续访问并形成数据通道. 每个计算引擎独占 1 个数据通道，TMMU 以轮询的方式依次打开并完成数据传输，能够充分利用内存的高读写带宽.

5.3 流式计算引擎设计

对于单计算节点，提出设计流式计算架构以支持高效的索引计算. 流式处理引擎不需要复杂的控制电路，计算以数据流为驱动，即只要在输入端有数据则计算会持续进行，不需要复杂的状态控制机制. 流式计算引擎能够最大程度提升单节点的计算性能. 单个计算引擎的内部结构如图 9 所示. 计算引擎内有 1 个地址缓存单元和 1 个节点缓存单元，分别用于暂存 TMMU 传递到计算引擎的节点地址和节点内容. 读数据缓存 FIFO 按照 8B 对齐的原则读取数据缓存单元内的节点数据，以指针和关键字组的顺序依次写入. 只要读数据缓存 FIFO 不为空，则比较器执行待比较关键字与节点内关键字的比较. 读写控制单元由指令操作码决定其打开或关闭. 如果操作码指示了关键字的插入或删除，读写控制单元打开，节点地址缓存当前节点的地址，写数据缓存 FIFO 依次读取节点内的数据内容. 当定位到待比较关键字的位置，写数据缓存 FIFO 根据指令控制插入或删除待比较关键字并更新指针内容. 节点内关键字全部写入写缓存 FIFO 后，读写控制打开内存通道，根据地址缓存中的地址更新节点内容. 由于索引维序的前序依赖是关键字查询，在同构计算引擎设计中，硬件资源的冗余开销只有读写控制单元和写数据缓存 2 部分. 相较于异构计算引擎的复杂状态控制来说，这部分硬件资源冗余几乎可以忽略.

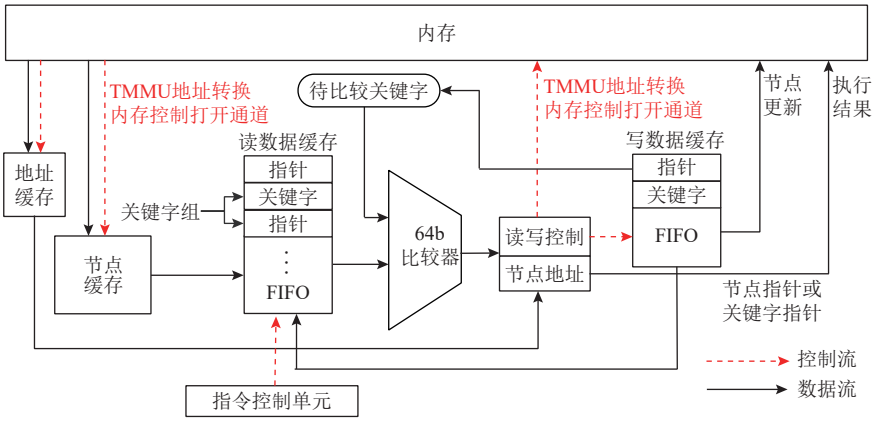


Fig. 9 Single computing engine architecture

图 9 单个计算引擎结构

计算引擎的执行过程以流水线方式进行，工作流程按照查询和维序过程分为 2 大类. 查询时，读写控制无效，仅有读数据缓存、比较和结果写回 3 个阶段. 当比较器得到待查询关键字期望位置时，输出关键字所在位置的指针信息. 期望位置是指，叶子节点内关键字相等，或内部节点中关键字大于前一个关键字且小于后一个关键字. 此时，如果叶子节点没有相等关键字，则输出不存在该数据. 输出的指针信息为叶子节点的卫星数据或等值节点逻辑地址，以及内部节点的叶子节点的逻辑地址. 插入或删除时，读写控制有效，包括读数据缓存、比较、写数据缓存和结果写回 4 个阶段. 指令打开读写控制单元，写数据缓存将读数据缓存内数据和待比较关键字根据比较结果的期望顺序依次读入. 期望顺序是指大于前一个关键字且小于后一个关键字，或存在与原有关键字相等的等值节点则输出指向等值节点的逻辑地址.

为简化节点的地址管理，降低节点动态维护的开销，对节点的合并和拆分进行规定：当且仅当节点内数据为空才执行节点的删除，当且仅当节点内数据量超过 512 B 时才执行节点的创建. 节点删除需要

删除上层父节点中的指针和索引关键字. 节点的插入对当前层节点进行修改，写数据缓存将前一半指针和关键字组写入当前节点，将后一半指针和关键字组写入新的节点地址空间，并在上层父节点中插入关键字和指针信息. 这 2 种操作需要同时更新 TMMU 中树表的深度、节点数等信息.

5.4 指令控制子系统

指令控制子系统主要包括指令解析模块、计算控制单元和内存控制单元，其结构如图 10 所示. 其中，指令解析单元分离指令中的操作数和操作码，解析索引计算类型及节点和树表逻辑地址. 计算控制单元查询处理引擎的执行状态，启动可执行的计算. 内存控制单元将逻辑地址写入 TMMU 完成地址映射，判定编号为 *tree_ID* 树的执行情况，并打开可访问的编号为 *tree_ID* 树的内存地址，完成处理引擎对数据的缓存.

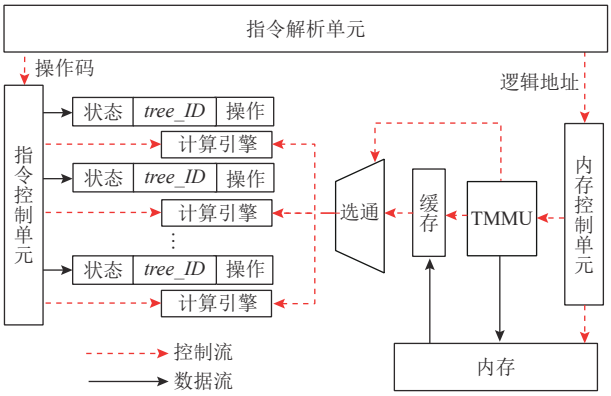


Fig. 10 Instruction control system

图 10 指令控制系统

为提升系统性能，设计多个同构计算引擎以实现指令集并行. 计算引擎的并行由指令控制单元和内存控制单元共同管理，其结构如图 11 所示. 计算控制单元为每个处理引擎维护一个寄存器，寄存器中包含处理引擎是否空闲标志位 *Dest_ID*、处理引擎正在执行的操作类别及处理引擎在处理的 *tree_ID* 信息. 计算控制单元获取指令队列中的首条指令，查询处理引擎寄存器信息，找到空闲的处理引擎，并判断当前指令及所有正在执行状态的处理引擎中是否存在对该指令 *tree_ID* 的更新操作. 有则阻塞指令并发；否则发送指令操作码到该空闲的计算引擎，并修改寄存器相关信息. 最大指令并行数与计算引擎数目相同. 当计算引擎状态寄存器 *Dest_ID* 空间被赋值，说明计算引擎已经完成对本条指令的执行，此时可以修改寄存器标识为当前计算引擎为空闲，并清除执行信息.

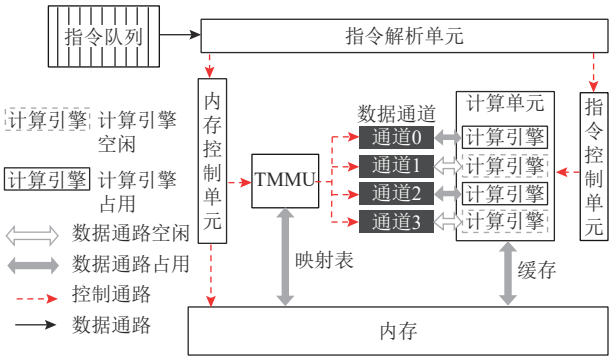


Fig. 11 Design of homogenous fully functional multi-processing engines

图 11 同构的全功能多处理引擎设计

5.5 指令设计

为配合系统的执行，设计支持的索引操作的指令如表 1 所示. 指令集包含 2 类指令：基本指令和附加指令. 其中基本指令是直接规定索引和树操作的指令，包括树的操作和关键字操作 2 类；附加指令用于部分基本指令的重定义和操作控制，包括范围查询和关键字更新 2 类附加指令.

Table 1 Design of Instruction Set

表 1 指令集设计

类型	指令名称	作用
基本指令	tree-creat	创建新树
	tree-delet	删除树
	key-query	单值查询
	range-query	范围查询
	key-update	关键字更新
	key-delete	关键字删除
	key-insert	关键字插入
附加指令	low-key	范围查询：区间下界
	high-key	范围查询：区间上界
	old-key	关键字更新：待修改的旧值
	new-key	关键字更新：应改为的新值

5.6 B+树子树系统

由于索引维序操作与其他操作之间存在强数据依赖，设计一种子树结构以解除这种数据依赖，其结构如图 12 所示. 子树结构使得对 1 棵大树的一次性访问可以拆分为多棵子树的多路访问. 也就是说，操作指令将不再以原始大树的根节点作为源操作数，而是以子树所在连续空间的起始地址作为访存的入口地址，只要读写不同子树空间，则对同棵大树操作的多条指令可以并行.

为降低对子树入口地址管理的复杂度，并提升子树划分方式的灵活性，将 1 棵大树的内存地址空

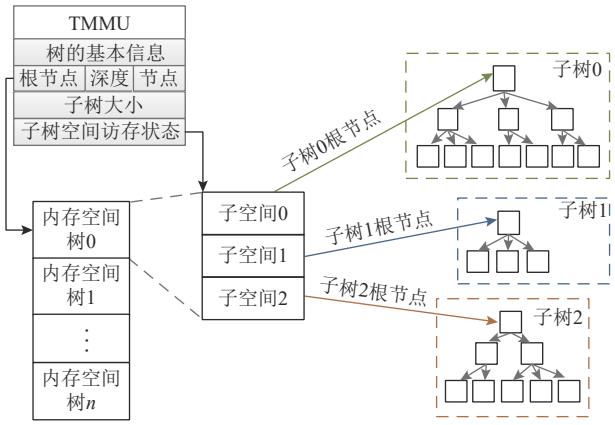


Fig. 12 Sub-tree structure and TMMU management

图 12 子树结构和 TMMU 管理

间(即树表)等分为几段连续子空间,以每部分子空间的起始地址作为子树的入口地址.子树入口地址为

$$tree\_address + i \times \frac{128}{subtree\_space}, \quad (2)$$

其中 i 表示在大树中的第 i 棵子树(i 从 0 开始计数),子树空间 *subtree_space* 的单位为 MB.

对齐的子树空间设计降低了内存地址管理的复杂性,也降低了地址映射计算资源的开销.子树的大小可以由地址空间范围划分方法灵活决定,即子树空间划分粒度可以根据实际应用需求设定,这种设计方式实现了灵活性与高效性的权衡.从而,在不同数据量和并发查询请求需求下,有效提升 B+树的并行读写效率,避免引入复杂的子树空间管理开销.子树划分对查询操作性能和资源的影响在第 6 节中具体评估.

为维护子树的状态,在 TMMU 的根节点映射表中加入与逻辑地址对应的子树的层数及内部节点数的记录.在插入新的索引关键字时,为均衡子树的访问频率,在 TMMU 中加入一个专用的轮询仲裁器管理子树空间的访存顺序:按照子树的内存空间顺序依次检查子树空间的访问状态,在可访问时申请对子树的访问,否则对下一棵子树进行检查.

显然,子树的引入改变了 B+树索引构建的方式.为配合子树结构,提出支持查询和维序的优化设计,其优化进程如图 13 所示.在 B+树索引构建及新关键字插入时,计算引擎按照 B+树创建指令的要求从内存中申请开辟一段连续的空间用于存放树表,即逻辑结构上的 1 棵完整 B+树.随后,将此段空间平均分配为 n 段连续子空间,并在 TMMU 中分别记录这几棵子树的信息.在当前树表中,每插入 1 个新的关键字,则以轮询的方式依次分配 1 段子空间入口地址,依照关键字插入的顺序依次在子树中插入索引关键字.按照这样的规则,子树仍然保持 B+树的特征,但子树构成的大树不维持逻辑索引结构.引入子树结构能够提升数据维护时树的并行读写效率.根据 TMMU 的设计原则,对树的管理也不会由于引入子树空间而过于复杂.

子树系统改变了索引指令的寻址方式,影响了指令的执行过程.关键字插入可以按轮询的方式选择子树.但是,查询和删除需要定位关键字所在的叶子节点,不能以轮询的方式选择子树,而大树没有逻辑上的索引结构,需要引入一种快速定位到包含关

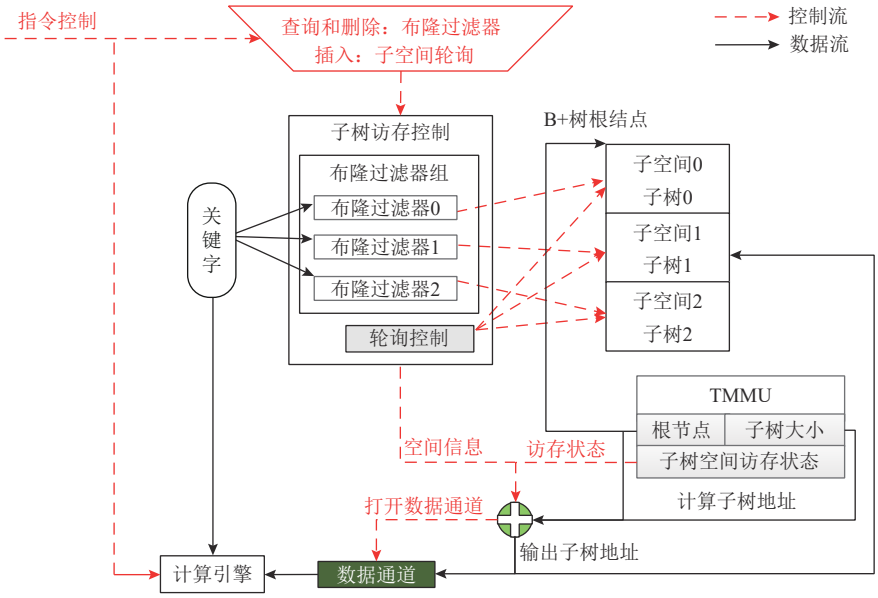


Fig. 13 Sub-tree structure and query optimization

图 13 子树结构和查询优化

键字的子树的机制. 为准确定位查询和删除操作中关键字所在的子树，在子树空间选择前加入 1 级布隆过滤器[13]. 布隆过滤器是一种快速判断某元素是否在集合中的功能部件. 利用布隆过滤器提前过滤 1 次关键字的位置，就可以快速定位关键字所在的子树空间. 将该子树的入口地址发送到空闲的计算引擎，子树内执行查找和删除与大树完全相同. 子树系统使得索引查询引入了 1 级布隆过滤器的计算开销，计算只需要几个周期，对性能的影响几乎可以忽略.

5.7 系统工作流程：多索引查询指令

基于 5.1～5.6 节所述的系统实现，我们以多条索引查询指令为例，介绍系统运行的过程，如图 14 所示. 当用户发出多指令请求时，指令按照顺序依次进入指令队列. ①指令处理：指令处理单元将按照队列顺序依次处理队列中的各条指令. ②指令解析：指令处理单元将得到的指令逐条解析，得到操作数和操作码. ③激活 TMMU：操作数中对 B+树的访存地址传递到 TMMU. TMMU 查询子树的访存状态，进行逻辑地址与物理地址的映射，得到子树的入口物理地址. ④打开数据通道：按照子树的内存地址轮询结果，内存管理单元打开可用的内存数据通道，使得节点内容能够到达相应的计算节点. ⑤激活指令控制单元：操作码和操作数中的关键字被传递到指令执行单元，指令执行单元管理计算引擎. ⑥计算引擎：指令控制单元依次查询计算引擎状态，激活空闲计算单元，修改相关寄存器内容. 若没有空闲计算引擎则等待，直到找到空闲计算引擎. 其中①②随指令流的控制连续不断地执行；③④和⑤⑥为异步控制. ⑦执行计算：计算引擎按照操作数和操作码的指令，启动查询或节点修改的执行流程. ⑧结果写回：查询时，仅输出查询关键字指向的卫星数据的地址. 更新时，则按照写数据缓存内容和地址缓存中的地址修改相应节点内容. 如果叶子节点需要删除，无需写回数据，只修改父节点及 TMMU 中的相关信息. 如果节点需要拆分，则将数据均等分为 2 部分，分别写入 2 个节点，并修改父节点和 TMMU 相关信息.

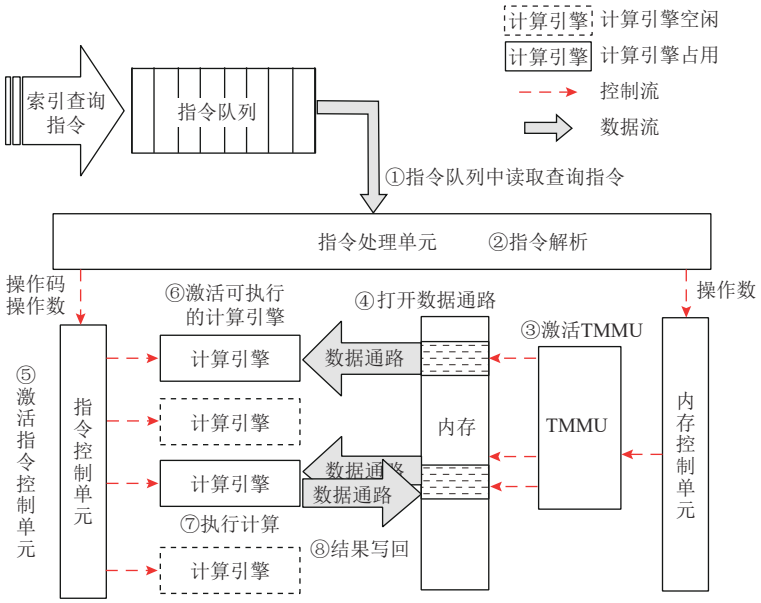


Fig. 14 Index query of B+tree accelerating system

图 14 B+树加速系统的索引查询

6 实验和评估

6.1 实验搭建

实验的硬件平台利用 PCIe 接口搭建 CPU 服务器与 FPGA 板卡互联的异构计算系统. CPU 型号为 Intel Xeon Gold 5218(@2.30GHz)配备 22 MB 的 L3 缓存和 64 GB 的内存(DDR4-2667). FPGA 板卡选用 Xilinx Alveo 的 U200, U50, U280 数据中心加速卡. CPU 与 FPGA 的互联接口为 PCIe3.0 ×16 总线. 实验分别测试 3 种不同资源配置(即异构系统选择不同 FPGA 板卡)下本方案的性能. FPGA 加速板卡具体资源配置情况如表 2 所示.

3 个板卡的主要区别在于其硬件逻辑资源和内存带宽限制不同. U200 逻辑资源充足但内存带宽受限，其上仅配备了 DDR4 作为存储资源，可提供 76.8 GBps

Table 2 FPGA Hardware Resource Configuration

表 2 FPGA 硬件资源配置

资源类型	参数	U200	U50	U280
存储	DDR4 空间/GB	64		32
	DDR4 带宽/GBps	77		38
	HBM2 空间/GB		8	8
	HBM2 带宽/GBps		316*	460
	BRAM	2 142	1 344	2 016
	URAM	960	640	960
SRAM	SRAM 空间/MB	43	28	43
	SRAM 带宽/TBps	37	24	35
逻辑资源	LUT	118.2 万	87.2 万	130.4 万
	寄存器	236.4 万	174.3 万	260.7 万
	DSP 单元	6 840	5 952	9 024

注：“*”标称 201 GBps，峰值 316 GBps.

存储带宽；U50 是一种轻量级的板卡，其硬件逻辑资源是三者之中最少的，但它使用了第二代高带宽内存（high bandwidth memory, HBM）资源 HBM2，内存带宽能够达到 316 GBps；U280 则是三者之中性能最高的，其硬件逻辑资源最多，且内存使用 DDR4 和 HBM2，既保证了存储空间大且带宽高达 460 GBps.

为评估数据库索引的查询和维序性能，实验的数据和程序选择数据库基准测试程序 TPC-C 中提供的标准数据库表生成数据和联机事务处理（on-line transaction processing, OLTP）程序，以充分测试数据库表记录的查询、删除、插入和更新的实际性能[14]. TPC- C 数据集用 9 个相关的数据表记录了某类仓库在不同地区的存货量、存储的不同商品库存以及地区消费者信息和货物订单等信息. 在这 9 个数据表中，选择最大数据表 order_line 作为本实验测试用数据. 测试数据表的查询、插入、删除和更新性能，对应 TPC-C 基准测试中订单信息查询、提交客户新交易申请、交易结束删除订单信息，以及订单修改申请. 实验运行所选择的关系型数据库为典型的 MySQL 数据库. 在本实验测试中，测试用的数据库表数据仍然存在 CPU 主机上，仅在 FPGA 板上内存预先存储需要构建索引的部分表信息. 此时认为 HyperTree 可以屏蔽与 CPU 进行数据传输的开销；HyperTree 可以单独完成指令要求的索引操作，即从索引构建并存储 B+树索引数据，到完成相应的事务处理的全过程. 实验的比较基准选择 CPU 并行执行 TPC-C 数据库索引查询和维序的性能. 进一步评估实验结果，选择目前具有最优效果的基于 GPU 实现的 B+树索引加速系统结果[5,7] 以及基于 FPGA 实现的 B+树加速器结果[10-11] 进行对比和分析.

在 6.2 和 6.3 节中，以 Alveo U200 板卡作为 B+树索引加速系统 HyperTree 的硬件实现原型. 此配置方法能够均衡考虑片上硬件计算资源和带宽资源的限制，设计 32 个并行的计算节点和 32 路数据通道实现对资源的充分利用（具体细节评估见 6.4 节），配置信息见表 3. 在这样的系统配置环境下，能够充分利用 FPGA 板卡的内存带宽资源，且不会引入冗余的计算引擎，引入不必要的硬件开销.

Table 3 Hardware Configuration of B+tree Index Accelerator Using Alveo U200 FPGA Board

表 3 基于 Alveo U200 FPGA 板卡的 B+树索引加速器硬件配置

配置指标	配置信息
计算引擎数目	32
带宽/GBps	51.2

6.2 索引查询性能

本节主要评价多查询语句的并行性能. 以 6.1 节中介绍的实验环境和实验数据为基础，实现 B+树索引加速系统 HyperTree. 实验评估指标有 2 个：一是以每秒执行查询指令数目条数 QPS（query per second）作为衡量系统性能的指标，其中实验结果 QPS 实际值的数量级为百万条每秒；二是以指令平均响应时间衡量每条指令的执行时间，用于验证 OLTP 是否满足应用系统的实时需求.

评估数据库表的数据量对系统性能的影响. 为充分释放处理引擎的并行度，指定实验中测试的并发指令数目为 32. 在实验中按照树表格式的设计规则，将数据库表大小的不同直接换算为 B+树索引中节点数目的不同，从而可以以节点数目的差别描述不同容量的 B+树. 由于比较基准测试中节点格式不同，节点可存储的关键字数量存在差别. 后序实验为公平对比，每组实验均基于具有相同节点数目的 B+树完成. 事实上，基于 CPU 和文献 [10] 中的方法所生成的 B+树要比 HyperTree 基于规则节点格式生成的 B+树更小. 由此，为公平地评估不同 B+树索引加速系统的性能随数据量的变化，实验结果的对比保持 B+树的节点数目的数量级一致，且在树结构的限制下尽可能保证数量相近. 实验评估时，以 B+树索引加速系统 HyperTree 生成的 B+树的容量作为数据量数量级的基准，图 15“数据量”指 HyperTree 生成的树. 此外，由于树存储格式的设计具有明显区别，后续的实验结果对比也仅考虑节点数目相同或维持同一数

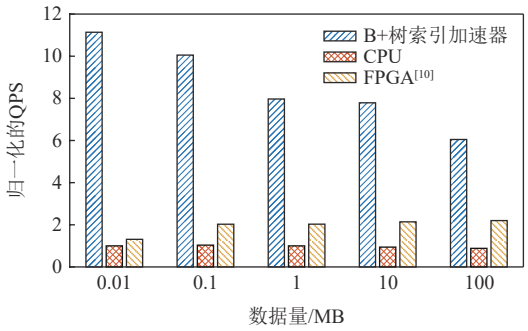


Fig. 15 Normalized throughput of B+tree accelerator

图 15 B+树加速器的归一化吞吐量

量级，而不直接以树的大小作为实验性能对比的指标.

图 15 显示了系统在 B+树的数据量变化下指令查询的归一化性能(以 CPU 单值查询为单位 1). 当在小数据量测试环境下时系统的吞吐量大约是 CPU 的 11.14 倍；当在大数据量的情况时，系统的查询性能提升大约为 CPU 的 6.84 倍. 小数据量时获得显著性能提升的原因主要是：紧凑的节点存储使得进行 B+树查询时层间跳数更少，且节点规则格式设计使系统对内存带宽利用更加高效. 大数据量时，B+树索引系统 HyperTree 在进行关键字查询时需要更多层内部节点来构建索引，从而增加了内部节点层间的跳数，使得查询平均延迟增加. 实验结果表明，B+树索引加速系统相比 CPU 能够获得显著的性能提升. 这一结果远超于文献 [10] 中利用 FPGA 实现的 B+树索引查询方案所获得的结果，相较于 CPU 性能仅平均提升 1.31～2.24 倍.

在多指令并发查询的实验中，以 TPC-C 数据集提供的 order_line 表为模板，按照规定的数据格式随机生成数据表，每个表包含 6.5～26.2 万行记录. 这样，利用其中待查询的关键字段构建 B+树索引可以保证单棵树内的节点全部有效，从而节点的数量级维持在 10^5，大约要求树表有效节点存储空间的容量为 10～128 MB. 实验设计单值查询和范围查询 2 类查询指令，单次并发指令数目从 10 条开始以 10 倍量级递增至 100 万条. 图 16 显示了系统在不同指令并发状态下归一化后的系统性能(以 CPU 单值查询为单位 1). 当并发指令数目为 10 时，计算引擎没有被占满，此时系统的性能明显低于其他多指令并发，但仍是 CPU 性能的 3.21 倍. 随着并发指令数的增加，由于查询关键字在节点内的位置具有不确定性，系统性能具有一定波动性，但与 32 条指令并发的最优性能大体一致. 与 CPU 在范围查询时性能显著下降有所不同，B+树索引加速系统 HyperTree 的 32 长度的范围查询的性能大约是其单值查询的 75.58%，性能损失不明显，能够达到 CPU 性能的 14.53 倍. 这是由于 B+树索引加速系统的规则大节点设计以及高效的并发读写机制，在范围查询时进行叶子节点读写过程中能够以较小的节点跳树实现大范围的查询. 实验结果表明，B+树索引加速系统能够在多指令并发的情况下维持高效的系统查询性能.

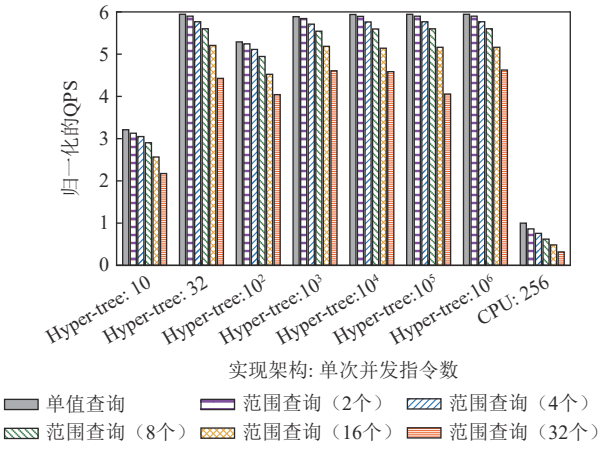


Fig. 16 Normalized throughput of parallel data query instructions

图 16 归一化的并行数据查询指令吞吐量

子树的引入对索引查询无显著影响. 一方面，子树的引入只要求增加 1 层布隆过滤器计算，FPGA 实现该计算过程只需要几个周期；另一方面，即使增加到 32 棵子树，树表容量的数量级仍然维持在 10^5. 因此，子树的引入对索引查询性能的影响很小，几乎可以忽略不计，所以这一部分实验不再评估子树的引入对索引查询性能的影响.

将该结果与 GPU 优化的 B+树进行对比. 参考文献 [5,7] 中利用 GPU 优化 B+树存储结构的多语句并行查询结果：GPU 实现的树的容量为 128 MB(只存储关键字，不存储卫星数据)时，相较于 CPU 实现的标准 B+树，单值查询性能提升 20.41 倍，范围查询性能提升 23.91 倍. 然而，由于 GPU 优化的树结构中只保留存储关键字的中间节点，这一方案也只能定位关键字在哪一个叶子节点，并没有最终找到数据库中的卫星数据，因而与本实验实现的环境难以直接进行对比. 一旦加入对卫星数据的寻址过程，会极大地损失 GPU 的查询性能. 同时，范围查询的性能提升要比单值查询更显著的原因是，CPU 实现范围查询时存在一定性能损失，其单值查询(比较基准)性能比范围查询具有 2.36 倍的增益. 如果以 CPU 单值查询的性能为基准，GPU 实现的 B+树加速系统的范围

查询性能提升大约为 11.22 倍，仅为单值查询性能提升结果的 50% 左右. 而本文所提出的 B+树索引加速系统 HyperTree 能够有效支持范围查询，性能损失大约 80%.

6.3 索引维序性能

本节主要评价系统的索引维序性能. 在 B+树索引加速系统中，除了设计多数据通道和并行计算引擎以加速索引查询外，还针对索引维序时由于数据冲突所导致的索引维序性能瓶颈提出子树的设计思想. 以 6.1 节中介绍的实验环境和 6.2 节中 B+树索引占满树表空间的环境为基础，搭建 B+树索引加速系统实验环境. 通过增加子树的棵数，评估子树系统设计对索引维序性能带来的提升. 实验以索引插入指令为例，子树棵数从 1 棵开始(即 1 棵大树，不实现子树系统)，以 2 的幂次增长，直到 32 棵为止. 实验结果以单棵树的插入性能(QPS)为归一化的单位 1. 如图 17 所示，实验结果表明随着子树棵数的增加，索引插入的并行性能取得显著提升：当子树棵数增加到 32 时，系统性能提升约 29.14 倍；且由于资源占用而导致的计算阻塞延迟也显著降低. 相比文献 [11] 基于 FPGA 实现的 B+树索引查询系统对索引更新操作会引入大约 70% 性能下降这一结论，HyperTree 能够有效解决这一问题. 另一方面，比较基准文献 [11] 所提出的 CPU-FPGA 优化架构并没有实现显著的索引维序性能提升，而仅通过优化调度 CPU 和 FPGA 的计算执行，使一些不适合利用 FPGA 加速的维序计算由性能更高的 CPU 取代，由此并没有充分利用 FPGA 计算性能.

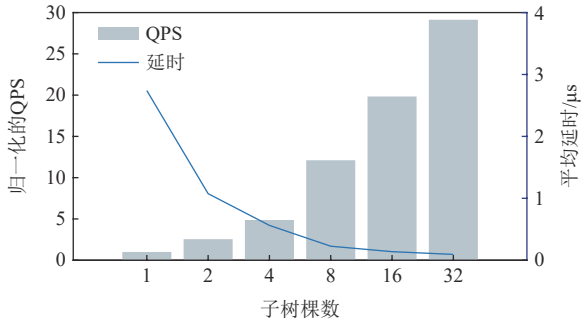


Fig. 17 Parallelism of sub-tree

图 17 子树并行性

将该结果与 GPU 优化的 B+树进行对比. 参考文献 [5,7] 中利用 GPU 实现的一种优化的 B+树存储结构的关键字插入结果：实验执行 GPU 实现的 B+树容量为 128 MB(只存储关键字，不存储卫星数据)时，其更新性能大约仅为 CPU 实现的标准 B+树的 72%. 这也进一步说明 B+树的维序操作是索引性能损失的关键因素. 而 HyperTree 的子树解耦合的数据存储方法能够解决由于索引维序对数据的独占性而导致的计算并行困难的问题，从而有效提升索引维序的性能，均衡提升 B+树查询和维序性能.

6.4 计算架构的可扩展性

本节评估 B+树索引加速系统的可扩展性. 计算架构的可扩展性可以衡量一种计算架构设计在增加计算节点时，硬件资源需求的增加情况以及实际性能的提升倍率. 利用不同硬件平台实现 B+树索引加速系统，可以衡量该计算架构设计是否能够充分利用板卡资源，以较小的边际硬件资源成本获得系统性能提升. 在本节实验中，选用 Xilinx Alveo U200, U50, U280 这 3 个不同资源配置的 FPGA 板卡来实现 B+树索引加速器系统 HyperTree. 这 3 个板卡中限制系统性能的因素不同：U200 的带宽资源受限，U50 的硬件逻辑单元受限，U280 则没有资源受限(在本实验的实现架构设计情况下).

评价计算架构的可扩展性，可以通过增加系统中的计算引擎数目，评估系统硬件资源的使用情况及系统带宽性能. 在实验中，设计系统的计算引擎数目从 4 开始，以 2 的幂次依次增长，直到 256 个截止(受硬件逻辑资源限制，U50 仅支持 128 个计算引擎). 本测试实验所实现的子树棵数与处理引擎数目相同，指令并发数目与处理引擎数目也相同. 假设一种理想的实验环境，即满足处理引擎能够无空闲地执行索引计算. 所获得的 QPS 值为理想条件下的结果，即没有数据冲突且处理引擎全部并发. 实验结果能够综合评估处理引擎、数据存储和子树系统的性能提升，结果如表 4 所示.

U200 板卡的内存带宽标称性能为 76.8 GBps，为最大化利用内存带宽资源，当计算引擎数目为 32 时，内存带宽资源可以达到 41.8 GBps，基本实现了带宽的最优利用效率，相对于 CPU 归一化的 QPS 可以达到 174.78. 计算引擎再翻倍后，受物理内存带宽的限制，系统整体性能不会进一步增长，反而造成计算引擎的冗余，即由于缺少计算所需数据而出现计算引擎空闲，造成硬件逻辑资源的浪费. U50 板卡由于使用 HBM 资源能够提供 316 GBps 的理论内存带宽. 为充分利用这一高内存带宽资源，实验结果表明，配置 128 个计算引擎能使系统资源的利用性能达到最优. 此时，限制计算性能的因素是 U50 片上的其他硬件计算资源的数目无法支持更多计算引擎. 而 U280 板卡是三者之中带宽性能和硬件逻辑资源都最丰富的，其标称内存带宽可以达到 460 GBps. 在片上实现 256

Table 4 The Scalability of B+tree Accelerator

表 4 B+树加速器的可扩展性

计算引擎数量	FPGA 板卡资源 U200 LUT 数量	寄存器数量	BRAM 数量	性能 实际带宽/GBps	QPS (CPU=1)
4	33 052	37 252	80	6.4	26.78
8	67 088	70 368	96	12.5	52.26
16	112 304	123 524	120	23.3	97.96
32	186 048	217 740	168	41.8	174.78
64	327 708	397 636	264	68.8	
128	588 368	741 904	456	75.8	
256	1 098 488	1 413 788	840	76.2	
总资源	1 182 240	2 364 480	2 142	76.8	

计算引擎数量	FPGA 板卡资源 U50 LUT 数量	寄存器数量	BRAM 数量	性能 实际带宽/GBps	QPS (CPU=1)
4	33 052	37 252	80	6.4	26.68
8	67 100	70 384	96	12.6	52.46
16	112 352	123 540	120	23.4	98.32
32	186 060	217 756	168	42.0	174.87
64	327 708	397 644	264	69.2	289.12
128	588 372	741 916	456	120.2	512.92
总资源	871 875	1 743 750	1 344	316	

计算引擎数量	FPGA 板卡资源 U280 LUT 数量	寄存器数量	BRAM 数量	性能 实际带宽/GBps	QPS (CPU=1)
4	33 052	37 252	80	6.4	26.64
8	67 084	70 356	96	12.6	52.45
16	112 288	123 504	120	23.2	96.56
32	186 032	217 728	168	42.0	174.80
64	327 660	397 992	264	76.1	316.72
128	588 328	741 872	456	152.4	634.28
256	1 098 488	1 413 788	840	302.3	1 258.12
总资源	1 303 680	2 607 360	2 016	460	

个计算引擎, 系统带宽性能可以提升至 302.3 GBps, 相对于 CPU 归一化的 QPS 可以达到 1 258.12. 实验结果表明, B+树索引加速系统具有很好的可移植性, 能够充分利用板卡资源, 扩展设计时只需引入很小的硬件资源开销即可实现索引查询和更新性能的线性提升.

7 结 论

为解决大数据环境下, 数据库查询和更新的性能, 本文提出一种支持 B+树索引加速的系统, 从访存和计算 2 方面协同加速数据库索引操作. 为提升内存读写带宽的利用效率, 设计规则的树表存储结构和高效的节点存储格式, 规定读写控制以单节点为单位, 使读写带宽可以达到 68.8 GBps. 为提升系统执行性能, 设计同构的多计算引擎, 支持语句级并行. 相比 CPU, B+树索引加速系统能够实现 14.53 倍的查询性能提升. 同时, 为缓解索引插入和删除操作的数据依赖, 设计一种子树存储结构, 支持多子树插入及更新的并行, 降低索引删除的数据依赖, 进一步提升索引的维序性能. 相较于单棵大树, 子树系统能够提升索引插入性能 29.14 倍.

作者贡献声明: 吴婧雅负责整理相关文献, 提出问题解决方法, 完成实验验证及撰写论文; 卢文岩提出设计实验方案并实现优化方案; 鄢贵海和李晓维提供实验环境支持, 指导论文撰写及修改论文.

参 考 文 献

[1] Hennessy J, Patterson D. A new golden age for computer architecture[J]. Communications of the ACM, 2019, 62(2): 48−60

[2] Papaphilippou P, Luk W. Accelerating database systems using FPGAs: A survey [C]//Proc of the 28th Int Conf on Field Programmable Logic and Applications. Piscataway, NJ: IEEE, 2018: 125−130

[3] Breß S, Heimel M, Siegmund N, et al. GPU-accelerated database systems: Survey and open challenges[G]// LNCS 8920: Proc of the 17th East European Conf on Advances in Databases and Information Systems. Berlin: Springer, 2014: 1−35

[4] Wu Linyang, Luo Rong, Guo Xueting, et al. Partitioning acceleration between CPU and DRAM: A case study on accelerating Hash Joins in the big data era[J]. Journal of Computer Research and Development, 2018, 55(2): 289−304 (in Chinese)
(吴林阳, 罗蓉, 郭雪婷, 等. CPU和DRAM加速任务划分方法: 大数据处理中Hash Joins的加速实例[J]. 计算机研究与发展, 2018, 55(2): 289−304)

[5] Daga M, Nutter M. Exploiting coarse-grained parallelism in B+ tree searches on an APU[C]//Proc of the Workshop of the Performance Computing, Networking Storage and Analysis (SC Companion). Los Alamitos, CA: IEEE Computer Society, 2012: 240−247

[6] Fix J, Wilkes A, Skadron K. Accelerating braided B+ tree searches on a GPU with CUDA [C/OL]// Proc of the 2nd Workshop on Applications for Multi and Many Core Processors: Analysis, Implementation, and Performance (in Conjunction with 38th Annual Int Symp on Computer Architecture). 2011[2021-08-27]. https://hgpu.org/?p=6508

[7] Yan Zhaofeng, Lin Yuzhen, Lu Peng, et al. Harmonia: A high throughput B+tree for GPUs [C]// Proc of the 24th Symp on Principles and Practice of Parallel Programming. New York: ACM, 2019: 133−144

[8] Zhang Werihua, Yan Zhaofeng, Lin Yuzhe, et al. A high throughput

B+tree for SIMD architectures[J]. IEEE Transactions on Parallel and Distributed Systems, 2020, 31(3): 707−720

[9] Shahvarani A, Jacobsen H A. A hybrid B+tree as solution for in-memory indexing on CPU-GPU heterogeneous computing platforms [C]//Proc of the Int Conf on Management of Data. New York: ACM, 2016: 1523–1538

[10] Heinrich D, Werner S, Stelzner M, et al. Hybrid FPGA approach for a B+ tree in a semantic web database system [C/OL]// Proc of the 10th Int Symp on Reconfigurable Communication-centric Systems-on-Chip. Piscataway, NJ: IEEE, 2015[2022-06-12].https://ieeexplore.ieee.org/document/7238093

[11] Heinrich D, Werner S, Blochzite Z, et al. Search & update optimization of a B+ tree in a hardware aided semantic web database system [C]// Proc of the 7th Int Conf on Emerging Databases. Singapore: Springer, 2018: 172−182

[12] Kaczmarski K. B+-tree optimized for GPGPU [G] // LNCS 7566: Proc of the on the Move to Meaningful Internet Systems: OTM Conf. Berlin: Springer, 2012: 843−854

[13] Cho J, Choi K. An FPGA implementation of high-throughput key-value store using Bloom filter [C/OL] //Proc of the Int Symp on VLSI Design, Automation and Test. Piscataway, NJ: IEEE, 2014[2021-10-08].https://ieeexplore.ieee.org/document/6834868

[14] Transaction Processing Performance Council (TPC). TPC-C benchmark [CP/OL]. 2013 [2021-09-12]. http://www.tpc.org/tpcc

Wu Jingya, born in 1994. PhD. Member of CCF. Her main research interests include domain-specific computer architecture and heterogeneous computing system optimization.

吴婧雅, 1994 年生. 博士. CCF 会员. 主要研究方向为专用计算体系结构和异构计算系统优化.

Lu Wenyan, born in 1990. PhD, associate professor, master supervisor. Member of CCF. His main research interests include deep learning accelerator, database accelerator, domain-specific computer architecture and heterogeneous computing system optimization.

卢文岩, 1990 年生. 博士, 副研究员, 硕士生导师. CCF 会员. 主要研究方向为深度学习加速器、数据库加速器、专用计算体系结构和异构计算系统优化.

Yan Guihai, born in 1982. PhD, professor, PhD supervisor. Member of CCF. His main research interests include computer architecture, domain-specific accelerator design and intelligent chip architecture.

鄢贵海, 1982 年生. 博士, 研究员, 博士生导师. CCF 会员. 主要研究方向为计算机体系结构、专用加速器设计和智能芯片体系结构.

Li Xiaowei, born in 1964. PhD, professor, PhD supervisor. Fellow of CCF. His main research interests include VLSI design, reliability design, fault-tolerant computing.

李晓维, 1964 年生. 博士, 研究员, 博士生导师. CCF 会士. 主要研究方向为超大规模集成电路设计、可靠性设计和容错计算.