华南理工大学

South China University of Technology

本科毕业设计（论文）

面向 ARM 架构的 openGauss IndexScan 算子优化

学　　院	软件学院
专　　业	软件工程
学生姓名	董梓键
学生学号	202030480184
指导教师	汤德佑
提交日期	2024 年 5 月 23 日

华南理工大学

学位论文原创性声明

本人郑重声明：所呈交的论文是本人在导师的指导下独立进行研究所取得的研究成果。除了文中特别加以标注引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写的成果作品。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律后果由本人承担。

作者签名：董梓键　　日期：：2024 年 5 月 23 日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，即：学校有权保存并向国家有关部门或机构送交论文的复印件和电子版，允许学位论文被查阅；学校可以公布学位论文的全部或部分内容，可以允许采用影印、缩印或其它复制手段保存、汇编学位论文。本人电子文档的内容和纸质论文的内容相一致。

作者签名：董梓键　　日期：2024 年 5 月 23 日

指导教师签名：冯德俊　　日期：2024 年 5 月 23 日

作者联系电话：18318788700　电子邮箱：822153562@qq.com

摘 要

随着信息存储技术的不断发展，数据规模爆炸式增长，人们对数据库的查询性能提出了更高的要求。索引扫描作为数据库基础查询中一种高效的性能提高手段，在许多主流数据库系统中都进行了实现，但是在数据库系统发展趋势偏向大数据和分布式计算的同时，索引扫描可能需要具备高效率、高吞吐等特点。

本文针对 openGauss 行存储模型下的非聚集索引的索引扫描给出了一种基于排序索引扫描和同步并行 I/O 的优化算法。排序索引扫描是指将索引扫描的若干次请求按照对应数据元组所在数据页在磁盘上的顺序进行排列。同步并行 I/O 与传统同步 I/O 流程类似，但是每次对一组 I/O 请求集进行处理。本文针对 openGauss 的索引扫描算法进行调研，并对其热点函数进行分析。为实现上述优化方案，引入 openGauss 的 tid 扫描算子作为排序索引扫描的基础，修改索引扫描的算子流程，并添加对应的可处理多个缓冲区的一系列 I/O 函数，通过将原本对磁盘的随机扫描变成分段的顺序扫描，并尽可能利用 SSD 的内部并行性，以提高索引扫描的查询性能。

最后，本文基于 TPC-H 对算法进行测试，结果表明本文实现的基于排序索引扫描和同步并行 I/O 的索引扫描算法平均性能在性能上要优于原生的索引扫描算法约 4%，优化程度基本满足要求，对数据库性能优化研究有重要帮助。

关键词：索引扫描；tid 扫描；排序索引扫描；并行同步 I/O

Abstract

With the continuous development of information storage technology and the explosive growth of data size, people have put forward higher requirements for database query performance. Index scanning, as an efficient means of performance improvement in database base query, has been implemented in many mainstream database systems, but while the development trend of database systems is biased towards big data and distributed computing, index scanning may need to be characterized by high efficiency and high throughput.

In the thesis, an optimization algorithm based on sorted index scanning and synchronous parallel I/O is given for index scanning of non-aggregated indexes under openGauss row storage model. Sorted index scanning means that a number of requests for index scanning are arranged in the order of the data pages on disk where the corresponding data tuples are located. Synchronous parallel I/O is similar to the traditional synchronous I/O process, but processes a set of I/O request sets one at a time. In this thesis, we investigate the index scanning algorithm for openGauss and analyze its hotspot function. In order to realize the above optimization scheme, the tid scan operator of openGauss is introduced as the basis of sequential index scanning, the operator flow of index scanning is modified, and a corresponding series of I/O functions that can handle multiple buffers are added, so as to improve the query performance of index scanning by turning the original random scanning of disks into segmented sequential scanning, and utilizing the internal parallelism of SSDs as much as possible.

Finally, the thesis tests the algorithm based on TPC-H, and the results show that the average performance of the index scanning algorithm based on sequential index scanning and synchronous parallel I/O implemented in this thesis is better than the native index scanning algorithm by about 4% in performance, and the degree of optimization basically meets the requirements, which is an important help to the research of database performance optimization.

Keywords: Index scan； Tid scan； Sort index scan； Parallel synchronous I/O

目 录

第一章 绪论

1.1 课题背景及意义

数据库系统是一种用于存储并管理数据的工具，具有广泛的应用空间，信息技术飞速发展，企业和组织所处理的数据量爆炸式增长，数据库业务逻辑随之日益复杂，人们对数据库系统的处理能力提出了更高的要求。与此同时，CPU、内存等硬件技艺的进步也为数据库提供了底层技术的支持。在这种情况下，如何提高 SQL 的查询性能是数据库系统设计时面临的巨大挑战。

索引扫描作为数据库系统中加速查询操作的高效手段，具有重要的研究价值。对索引扫描算法进行优化，从应用程序角度来说，可以减少非必要的数据读取处理，降低查询操作的响应时间，提升系统的整体效率，改进数据库应用的性能；从用户角度来说，快速、高效的查询可以提高用户的数据库使用体验，同时，高性能的数据处理性能也可以让用户及时进行分析与决策。

1.2 课题提出

openGauss 是华为公司开源的一款关系型数据库管理系统，具有良好的兼容性和可扩展性，有着非常广泛的应用前景。ARM 架构具有低功耗、高性能的特点，越来越多地应用于数据中心和云计算领域。研究 ARM 架构下 openGauss 数据库中 SQL 算子的性能优化，可以作为后续数据库系统 SQL 性能调优研究的基础。

本文旨在研究在 ARM 架构下对 openGauss 数据库系统的索引扫描算子，给出合理的优化方案并进行测试，以提高索引扫描的性能。

1.3 国内外研究现状

针对 SQL 性能提升的研究可以粗略地分为以下几个方向：修改算子实现以支持外部高性能设备、针对特殊数据分布调整算子实现等。下面将从这三个方面简要介绍目前国内外基于 SQL 性能优化的研究状况。

1.3.1 DBMS 内部实现优化

直接选择优化 DBMS 内核算子的具体实现是最直接但又最困难的一种性能优化方式，通过修改算子流程提高算子在面对各种情况下的平均性能，可以在最大程度上沿用其稳定性。

在数据库查询过程中往往会占用部分内存进行操作，其中选择算子 SELECT 和排序算子 SORT 在数据量较大时会占用更多的内存，Wang 等人针对 SELECT 算子设计了 Filter 算子缓存，针对 SORT 算子也设计了自适应 SORT 算子缓存，将算子的执行结果缓存到磁盘中，在其后的执行过程中磁盘中的缓存中的执行结果进行过滤、排序等操作后再返回上层算子，减少 I/O 开销[1]。Lothar 等人通过实验发现索引扫描性能对缓冲区大小和每个键值存储的数据记录非常敏感，通过推导具体性能模型得到索引扫描性能公式，为索引扫描的性能优化做出重大贡献[2]。

数据库内核在为计划树中的算子选择实际的运行策略是根据扫描表的统计数据调整的，也就是说，优化器在很大程度上依赖于统计数据创建高效的查询计划，在运行过程中，内核虽然会为各种操作赋予一个合理的比例因子以减少其对统计数据的影响，但是固定的因子无法适应所有情况。Reneta 等人主张通过数据的统计信息在运行中实时调整算子的选择，通过设计并实现一种称为平滑扫描的自适应访问路径选择操作符解决计算过程中出现的统计数据过时或失效等情况，使 SQL 语句的运行不受统计数据信息的影响[3]。

Fazai 等人提出了一个 SQL 模型，该模型允许检查与预定义规则、SQL 基准、SQL 调优策略和数据冻结相关的查询性能，并对 SQL 查询指定一套系统化的规则，通过规范化提高 SQL 的稳定性以及执行的高效性[4]。

1.3.2 面向外部设备的优化策略

随着大数据、视觉设计行业的发展，更高算力更高存储的外部设备的应运而生，而 DBMS 的算子算法更多的是面向普适性的基础设备的设计，这些设计在更高性能的外部设备中的表现可能并不理想，因为算法并没有为高性能设备进行特殊化支持，无法高效利用这些设备来提供更快速更高效的数据库查询分析操作。

使用计算型存储设备加速 SQL 查询就是一种有效的途径，罗晓等人在计算型存储设备上提出了基于表数据特征的过滤算子动态卸载和执行优化方案[5]。方案提出根据目标表的大小等参数估算计算型存储设备卸载过滤算子的计算代价，使用目标表的元组过滤比例估算卸载过滤算子过程中减少的数据传输量，最终动态选择是否将原生的过滤算子下放到计算型存储设备处理，并确定对应的处理程度，以最大程度利用计算型存储设备并保证整个算子运行流程的高效。

以 GPU 为计算核心也是提升 SQL 性能的可行方法。传统的通用数据库，如 openGauss，以 CPU 为计算核心，由于功耗墙、内存墙等原因，在执行 SQL 查询时可能会出现性能不佳的情况。GPU 是一种异构计算硬件，具有高吞吐、高并行的特点，陈现森等人通过将 PG-Strom 移植到 openGauss 并提出基于分块读取和按键分发的 CPU-GPU 协同的异构并行方案，设计实现了可嵌入 openGauss 列向量执行引擎的自定义算子框架[6]；谭国龙面向 CPU-GPU 架构设计实现多查询批流处理的关系算子流计算方法[7]。

围绕 SSD 的高性能提升 SQL 速度也是一个重要途径。不同于传统的机械硬盘，SSD 的随机访存性能更优，且 SSD 内部支持并行。Dasom 等人针对传统机械提出了排序索引扫描方法，将元组获取列表按照其在磁盘上的顺序排列，可以顺序读磁盘，减少随机访存带来的 I/O 时间消耗[8]。Roh 等人则提出一种名为同步并行 I/O 的 I/O 策略，通过将 I/O 请求组织起来，作为整体进行 I/O 请求，充分利用 SSD 的内部并行性以提高 SQL 性能[9]。

1.3.3 针对特殊数据或语句的算子流程优化

特殊的数据分布，如倾斜度高的数据集具有一些特殊的性质，DBMS 常规的算子流程面对这些数据分布的性能可能会变得非常差，而面对复杂子查询语句时，DBMS 经常会出现负载太大的情况，所以针对特殊的数据或语句需要特殊的算子处理方法。

司一鸣针对 Shared-Nothing 架构下静态数据的分布式哈希连接（Dist-HJ）进行优化[10]。通过设计基于 DBMS 原生表统计信息的倾斜值统计方法，其中倾斜值统计以及频率估计嵌入在表统计数据收集过程中，在分布式数据库 Cockroach DB 对分治-广播算法进行实现，在倾斜度较高的数据集也能表现较好的性能。

Carlos 等人针对 SQL 线性递归查询进行研究[11]。利用图的传递闭包和邻接矩阵的幂矩阵等为其建模，通过研究五种常见的线性递归查询语句，评估两种 SQL 实现算法：seminaive 和 direct，提出谓词相同时的早期策略选择可以显著加快线性递归查询的计算速度。

1.4 本文主要工作

本文主要工作如下：

（1） 调研 openGauss 索引扫描算子和 tid 算子算法流程以及外存存储部分流程。

（2） 使用 Vtune 工具分析索引扫描算子的热点函数。

（3） 仿照 tid 扫描的实现修改索引扫描执行流程，以实现排序索引扫描。

（4） 在磁盘管理模块添加一系列可处理多缓冲的 I/O 函数，以实现同步并行 I/O。

（5） 使用 TPC-H 工具对优化后的索引扫描进行性能评估并分析结果。

1.5 本文组织

本文整体框架共分为五章：第一章介绍本文研究课题的背景及意义，并介绍现今国内外对 SQL 优化的研究情况，并给出论文整体框架；第二章主要给出算法相关的理论基础，对 openGauss 的存储引擎基础模型、排序索引扫描和并行同步 I/O 的原理进行解释；第三章结合两种优化方法给出针对索引扫描的优化方案的原理；第四章根据优化方案给出核心的具体实现细节并对优化方案的性能提升测试，结合图表给出实验结论；结论章是对整体方案的实验结论，并给出未来可发展工作方向。

第二章 相关理论和基础

本章主要介绍与 openGauss 索引扫描相关的理论，首先会介绍 openGauss 的行存储引擎，并简要地说明 openGauss 的几种扫描方式，围绕索引扫描和后续优化算法使用到的 tid 扫描进行整体介绍，然后会谈到索引的一个重要特性——聚集性，本文主要探究的是非聚集索引上的扫描的优化，最后会对两种性能优化方法——排序索引扫描和同步并行 I/O 的原理进行介绍。

2.1 openGauss 的行存储引擎

本文设计的优化算法基于 openGauss 的行存储引擎，接下来介绍 openGauss 存储引擎的基本类型以及行存储引擎存储单元的底层结构模型。

2.1.1 openGauss 的三种存储引擎

openGauss 的存储引擎包括以下三种类型：行存储引擎，列存储引擎和内存引擎。

行存储引擎中，数据按元组行按行进行存储，主要面向 OLTP 场景设计，例如订货、发货、银行交易系统等。

列存储引擎中，数据按照属性列地进行存储，主要面向 OLAP 场景设计，例如一些数据统计、报表分析等。

行存储引擎和列存储引擎都属于外存存储引擎，而内存引擎会把元组存放在内存中，读写速度相较于其余两种读写会经过大量磁盘 I/O 开销的引擎，主要面向性能要求高的场景设计。

2.1.2 行存储引擎的基础模型

行存储引擎的元组结构和页面结构，是行存储引擎的 DML 实现、可见性判断、行存储等功能和管理机制的基础。

行存储引擎是基于磁盘的，所以行存储的格式是遵循更加亲和磁盘存储的段页式设计，存储结构以页为单位，便于和操作系统内核以及文件系统的接口进行交互。图 2-1 是一个基本的堆页面的示意图：

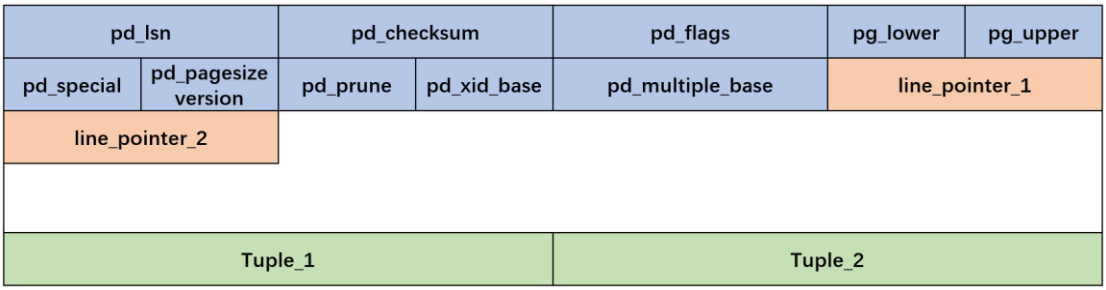

图 2-1 openGauss 行存储引擎的堆页面模型

页头是整个页面的首部信息，包含页面公用的属性信息，如地址信息、校验和等。除此之外，页头还包含若干个 line_pointer，line_pointer 指向数据元组实际存储的地址。

对于每一个元组 Tuple，系统会保存其在管理系统中的唯一标识：ctid，ctid 是 ItemPointer 类型的，存储元组所在的页面号以及元组对应的 line_pointer 的偏移量，即第几个 line_pointer。根据 ctid，系统可以迅速定位到对应的数据页的 line_pointer，然后就可以根据 line_pointer 找到对应的元组的实际数据。

2.2 openGauss 的扫描算法

openGauss 内核支持以下几种扫描方式：顺序扫描（Seq Scan）、索引扫描（Index Scan）、纯索引扫描（Index Only Scan）、位图扫描（Bitmap Scan）和 tid 扫描（Tid Scan）。在输入 SQL 查询语句后，openGauss 内核会在语义分析生成 SQL 计划树后，根据表的基数、选择性、磁盘的顺序 I/O 代价、随机 I/O 代价等，为执行计划中的每一个步骤选择理论上最合适的算子。本文将围绕索引扫描和 tid 扫描的具体实现展开。

索引扫描类似全表顺序扫描，但是顺序扫描是按序扫描整个数据表上的元组，而索引扫描是扫描在数据表属性上建立的索引，由于索引表比数据表小，相比于全表扫描，索引扫描具有 I/O 次数少，I/O 效率高的优点，缺点是当 SQL 语句的选择度高时，索引扫描的性能会急剧下降，反而不如全表扫描。

Tid 扫描也是一种行扫描。在 Btree 索引中，除了会存储元组的值，还会存储对应的 ctid。Tid 是一个 ItemPointer 类型的指针，指向一个具体的数据元组，tid 中存储元组所在的数据页号和数据块号，所以根据 tid 进行查找时可以直接定位元组所在的物理地址，查找速度非常快，缺点是使用 tid 扫描只适用于单点查询，对于范围查询、联表查询来说性能比较一般。

2.3 索引的聚集性

索引的聚集性是指表数据在磁盘空间上的物理排列的方式，是影响索引扫描的重要因素之一。根据聚集性，索引可以分为聚集索引和非聚集索引两种。其中，聚集索引是指数据表中的数据元组在磁盘上的排列顺序和数据表索引上的数据项的一致的索引，相反，如果数据表中的数据元组在磁盘上的排列顺序和数据表索引不一致，则称为非聚集索引。

由于数据表在磁盘上实际的排列方式只能有一种，所以一个数据表至多只能有一个聚集索引。在聚集索引上进行查询扫描时，DBMS 会按磁盘排列顺序依次获取数据页，可以高效地进行访问磁盘。

但是如果数据表上的索引是非聚集的，按照数据表索引进行遍历扫描时，访问磁盘上的数据页就不是顺序的，而是随机的，也就是说，遍历非聚集索引的时候会出现反复读取同一个数据页的情况。在服务器资源较为宽裕的时候，性能消耗可能尚且不明显，但是在服务器缓冲空间有限时，大量重复的数据页会导致缓冲利用率低，进而降低了索引扫描的查询性能。

2.4 排序索引扫描

排序索引扫描对比传统的索引扫描，会在访问磁盘上的数据页之前，将索引项按照元组对应的磁盘的数据页的顺序进行排序，这样可以让非聚集索引扫描像聚集索引一样，只需按顺序最多读取一遍磁盘上的数据页。使用排序索引扫描，可以大幅度地减少扫描所需的 I/O 次数，从而达到提高性能的目的。

虽然使用排序索引扫描进行扫描，可以将随机的索引项顺序转换为磁盘上的排列顺序，但在最后获取查询结果的时候，返回的结果集是按照磁盘上的排列顺序而非索引项顺序，也就是说结果集是乱序的。这种乱序可以通过 DBMS 内部的排序算法进行规避，不会造成太大影响。

2.5 并行同步 I/O

并行同步 I/O 的执行流程和传统的同步 I/O 类似，但是不同于传统同步 I/O 每次只操作单个 I/O 请求，并行同步 I/O 以 I/O 请求数组作为操作单元。并行同步 I/O 利用服

务器底层的 SSD 存储，可以利用 SSD 队列处理 I/O 的时间非常短的特性，可以让多个 I/O 在较短时间内进行传递，提升 I/O 请求的效率。

实现并行同步 I/O 需要满足下列三点要求：

1. I/O 请求发送：并行同步 I/O 一次只向磁盘发送一组 I/O 请求，并一次性取回所有的处理结果，也就是说，同一时间只能有一组 I/O 请求集被处理，其他的 I/O 请求集需要先被挂起，等待收到前一个 I/O 请求集的处理结果。
2. I/O 请求处理：I/O 请求集在从用户空间内陷到内核空间的过程中，会被视为一个整体，也就是说，在这个过程中，请求集中的每个请求在整个请求集都被发送到内核空间之前都会进行等待；
3. I/O请求结束：并行同步I/O要求I/O请求集处理过程中，对应的进程应该阻塞，直到 I/O 请求列表中的所有请求都处理完毕。

同时满足上述三点要求的 I/O 方案并不存在，只能尽最大程度满足。

2.6 本章小结

openGauss 的存储引擎分为三类：行存储、列存储和内存存储。行存储基于 ctid，通过 ctid 定位数据元组所在数据页和指针偏移量。openGauss 有多种扫描算法，其中索引扫描和 tid 扫描均为相对高效的行扫描算法。索引的聚集性是指索引表和元组表在磁盘上的排列顺序一致与否，是否聚集会影响扫描时访问磁盘的方式，进而影响磁盘性能。排序索引扫描通过将待扫描元组按 tid 进行排序，通过顺序访问磁盘减少 I/O 开销。同步并行 I/O 类似传统同步 I/O，但是一次处理多个 I/O，可以利用到 SSD 的内部并行性。

第三章 Index Scan 算子优化研究

本章主要围绕索引扫描算子的优化算法展开，首先对 openGauss 的扫描算子的流程进行剖析；然后对索引扫描过程中的热点函数进行分析，确定通过将 I/O 请求组织起来并按照磁盘顺序进行排序来提升查询的性能。

3.1 openGauss 原生扫描算子研究

openGauss 的扫描算子的基础运行流程是一致的，都是通过策略选择、执行器执行到具体扫描算子的顺序执行的。不同的扫描算子的扫描状态中包含的参数是不同的，具体实现中对于扫描算子的初始化和扫描函数都是不同的。本节首先介绍所有扫描算子的通用基础运行流程，然后针对索引扫描和 tid 扫描具体介绍它们的特化实现，最后再简要介绍一下扫描算法会调用到的底层 I/O 函数。

3.1.1 openGauss 扫描算子基础运行流程

openGauss 对于扫描算子的实现集中在/executor/runtime/nodexxxScan.cpp 中。openGauss 将 SQL 语句解析生成 SQL 计划树后到具体算子的执行，如图 3-1 所示，会经过三个阶段：策略选择、执行器执行、具体算子执行。接下来分别对三个阶段的具体实现进行阐释。

3.1.1.1 策略选择阶段

openGauss 解析 SQL 语句生成 SQL 计划树，计划树的节点为 PlanState*类型，节点除了存放左右子节点等信息之外，还存放了节点的策略选择，这里的策略选择区别于后面要谈及的算子不同，只是根据 SQL 语句的 SELECT、ORDER 等谓语选择使用哪一种执行策略，如扫描策略、排序策略、聚集策略等。内核通过调用 PortalStart()和 PortalRun()初始化并进行策略选择，在选定了各个步骤使用的策略和对应可能的运行参数后，就会转到执行器执行阶段。

3.1.1.2 执行器阶段

在执行器阶段，内核会调用 ExecutorRun、standard_Executor、ExecutePlan 等一系列函数为上一步选定的策略选定具体的算子实现，例如，在策略选择阶段内核根据计

划树选定了 Scan 策略后，在过滤条件的属性上有索引约束且不含有其他特殊条件的情况下，内核会优先选择索引扫描策略。在选定了具体算子后，会在设置算子运行所需的表达式上下文信息，包括运行时的约束条件如筛选条件、过滤条件等、资源条件如运行时地址空间的分配信息等以及一些标志用来判断算子的运行逻辑，然后会将算子选择标志传递给 ExecuteProcNode 函数，ExecuteProcNode 是一个中间函数，负责根据 ExectePlan 函数传递的信息调用具体算子对应的执行函数。

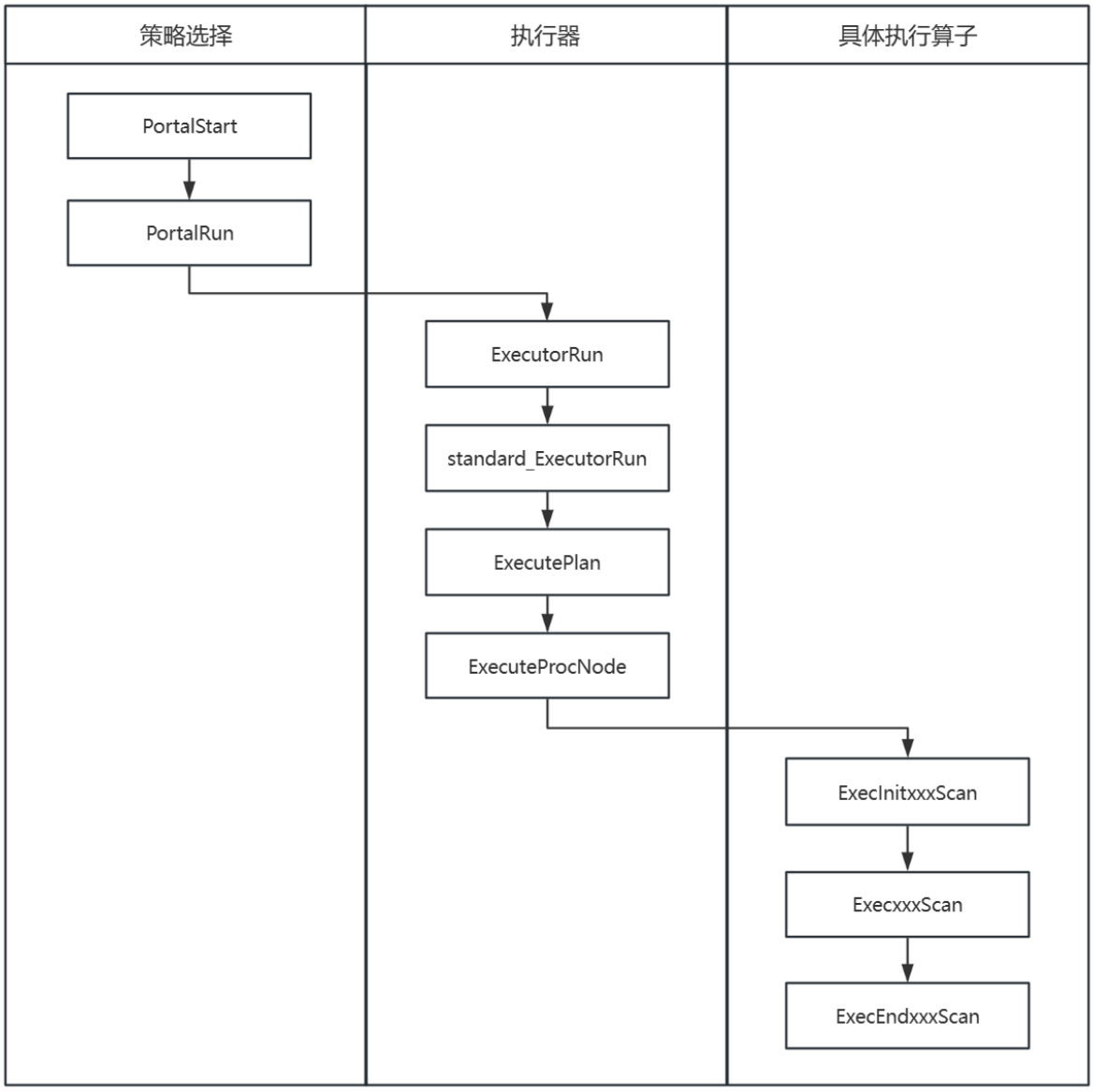

图 3-1 openGauss 扫描算子的基本流程

3.1.1.3 算子具体执行阶段

在具体的扫描算子执行阶段，所有扫描算子都会拥有一个类型为 xxxScanState（这

里的 xxx 是指具体的扫描算法）的状态，xxxScanState 类包含一个基础的 ScanState 类参数，包含所有扫描需要的基础信息，如扫描表位置，结果表输出位置，扫描方向等。除此之外，xxxScanState 会根据具体扫描算子的需求，加入对应的成员变量，例如，对于索引扫描，还会加上索引过滤条件，扫描键，运行时键等信息。

所有扫描算子执行时，都会先调用 ExecInitxxxScan 通过 Estate，即当前执行器状态对 xxxScanState 进行初始化，对表达式上下文进行初始化操作。同时，ExecInitxxxScan 函数会打开将要被扫描的基表并获取对应的合适的锁，并分配结果槽的空间。

随后，openGauss 内核调用 ExecxxxScan 扫描函数，ExecxxxScan 是一个中间函数，内部调用 ExecScan 进行具体的扫描。ExecScan 包含三个主要参数：node，accessMtd，recheckMtd，类型分别是 ScanState*，ExecScanAccessMtd，ExecScanRecheckMtd，node 参数负责传递当前扫描的运行参数，accessMtd 参数用于指定扫描算子扫描数据表的访问函数，recheckMtd 参数用于指定扫描算子检查取得的数据元组的重查函数。不同扫描算子会在各自的文件中实现各自算子对应的访问函数和重查函数。ExecScan 函数会首先从 node 参数中提取过滤条件、投影信息以及扫描所需的表达式上下文信息，然后对条件进行统一处理，具体流程见下图，最后通过循环调用 ExecScanFetch 函数获取数据元组，根据约束条件调用 ExecQual 进行条件过滤，调用 ExecProject 进行投影操作，处理结果元组，最后存放到指定的 TupleTableSlot 类型的数据结果槽中返回。

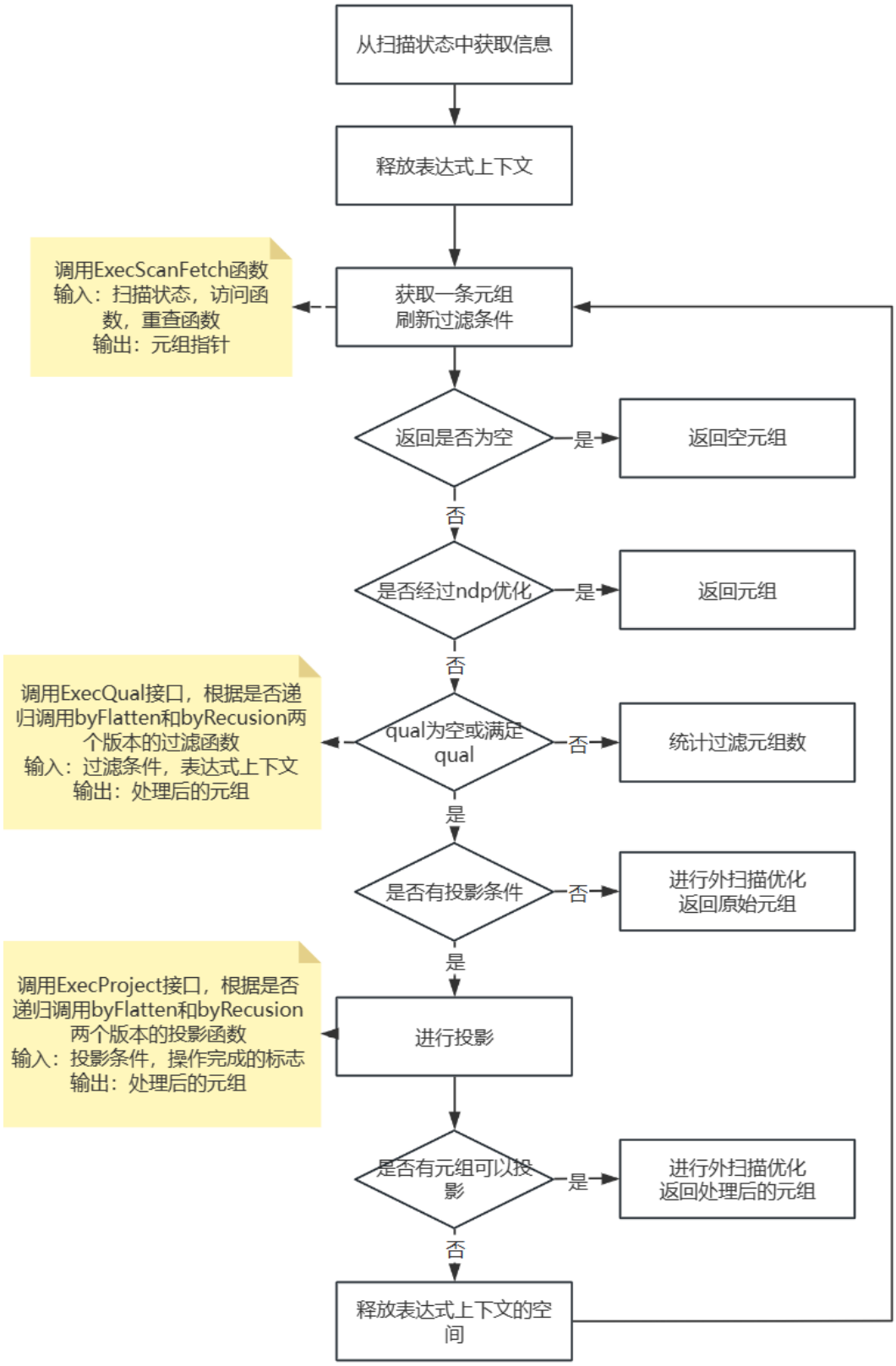

图 3-2 openGauss ExecScan 函数流程图

ExecScanFetch 参数与 ExecScan 相同，主要功能是对当前的运行状态进行判断，选择是否调用重查函数进行元组的检查，若没有查重的必要，会调用访问函数获取一条

数据元组。

在执行完 ExecScan 函数获得结果槽后，内核会调用 ExecEndxxxScan 函数对资源进行回收。函数首先会清空存放扫描表和结果表的槽，然后关闭使用到的表，释放初始化阶段施加的锁以及其他使用到的空间。

3.1.2 openGauss 扫描算子具体实现

3.1.2.1 索引扫描具体实现

在 IndexScanState 中，除了基础的 ScanState 状态外，还添加了索引原生过滤条件以及扫描键，排序键和运行时键等状态变量。索引原生过滤条件（indexqualorig）就是 SQL 语句中带有索引的属性的过滤条件的列表，主要是用于后续的 ExecQual 函数判断获取到的元组是否符合过滤条件准备。扫描键是索引原生过滤条件经过处理后的过滤条件集的子集，一个扫描键对应着一个过滤条件。排序键和扫描键类似。运行时键是比较特殊的一类键，需要在算子具体运行时用表达式上下文进行评估，所以 IndexScanState 中还需要为运行时键保存需要用到的表达式上下文。

IndexNext 函数是索引扫描实现的访问函数，函数参数仅有一个 IndexScanState 的扫描状态参数，返回 TupleTableSlot 指针类型的结果槽。如图 3-3 所示，函数执行过程中，会从 IndexScanState 中获取当前的执行状态、扫描方向、扫描描述符、表达式上下文和扫描基表存放的槽信息。然后，会根据索引的排列方向和执行状态中指定的扫描方向调整索引扫描实际的方向，随后使用循环逐一获取元组，随后调用 scan_handler_idx_getnext 处理器函数在堆上获取一个元组，在 scan_handler_idx_getnext 函数中含有多个版本 idx_getnext 函数用来获取元组，其中有为存储在分配了桶的堆上的索引使用的 Hbkt_idx_getnext 等，所以调用 scan_handler_idx_getnext 后原先的 IndexScanDesc 可能会由于桶的一些哈希操作导致内置的用于存放取到的元组的缓冲区对应的下标索引值被修改，需要更新当前的 IndexScanDesc 的值，随后调用 ExecStoreTuple 将暂存在 IndexScanDesc 内的 xs_mbuf 缓冲内的元组存储到准备好的结果槽中，如果索引是有损的，还需要将取到的元组和过滤条件进行匹配，调用 ExecQual 验证查重。最后，返回获取的结果槽。

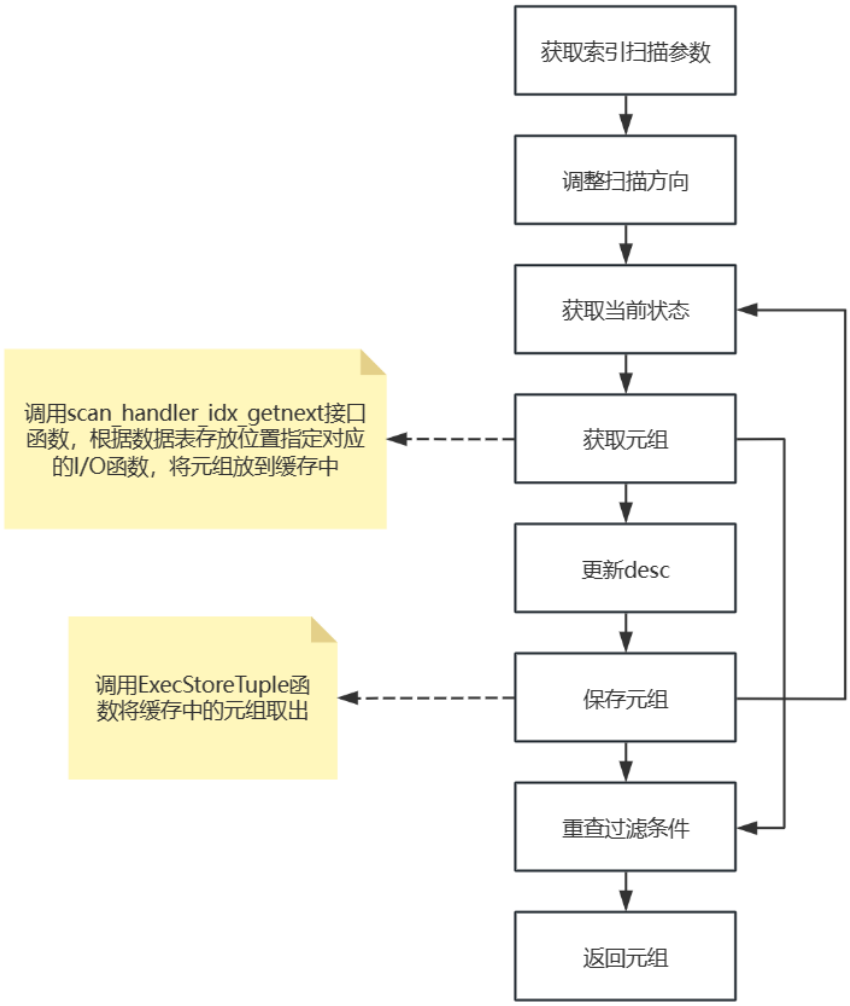

图 3-3 openGauss IndexNext 函数流程图

在 scan_handler_idx_getnext 函数中，若要扫描的数据表不是以桶的方式存储的，会调用最基础的版本 idx_getnext，idx_getnext 函数需传入 IndexScanDesc 扫描描述符和当前的扫描方向，函数调用 index_getnext_tid 函数获取下一个 tid，根据 tid 的状态进行特殊化处理后，调用 IndexFetchTuple 函数获取元组并返回，IndexFetchTuple 函数内会调用底层宏 pgstat_count_heap_fetch 在 IndexScanDesc 指定的数据表中获取一条元组记录并返回。

3.1.2.2 openGauss Tid 扫描具体实现

与索引扫描不同，除了基础的 ScanState 外，TidScanState 需要维护一个 tid 表达式列表，tid 的数量，tid 指针和一个 tid 列表。其中 tid 表达式列表类似 IndexScanState 中的过滤条件列表，是 SQL 中对于 tid 的过滤条件的集合，一个 tid 过滤条件最终会指定

一个元组。Tid 表达式列表主要用于 Tid 列表的初始化和作为 ExecQual 判断元组是否满足 tid 过滤条件的依据。Tid 列表在使用 tid 过滤条件初始化时会进行类似 STL 中 vector 的扩容，所以 tid 列表空间中不一定装满了 tid，所以用一个 tid 计数器存储 tid 列表中 tid 的数量。Tid 指针，其实是一个 int 类型的数据，用于在 tid 扫描过程中作为下标遍历 tid 列表。

由于 Tid 扫描不像常规的扫描算子一样需要对数据表进行扫描，所以对应的访问函数的具体实现有明显差异。TidNext 函数同样仅有一个 TidScanState 类型的入参，返回一个 TupleTableSlot 类型的结果槽。如图 3-4 所示，函数执行过程中，从扫描状态中获取扫描相关信息后，会先调用 TidListCreate 函数计算当前扫描需要的 tid 列表，TidListCreate 遍历 tid 过滤条件列表，调用 ExecEvalExprSwitchContext 函数找到对应的 tid，存放到 tid 列表中。随后调用 TidFetchTuple 函数获取 tid 列表上对应的元组，在 TidFetchTuple 中，调用函数获取元组，最后逐层递归返回结果槽。

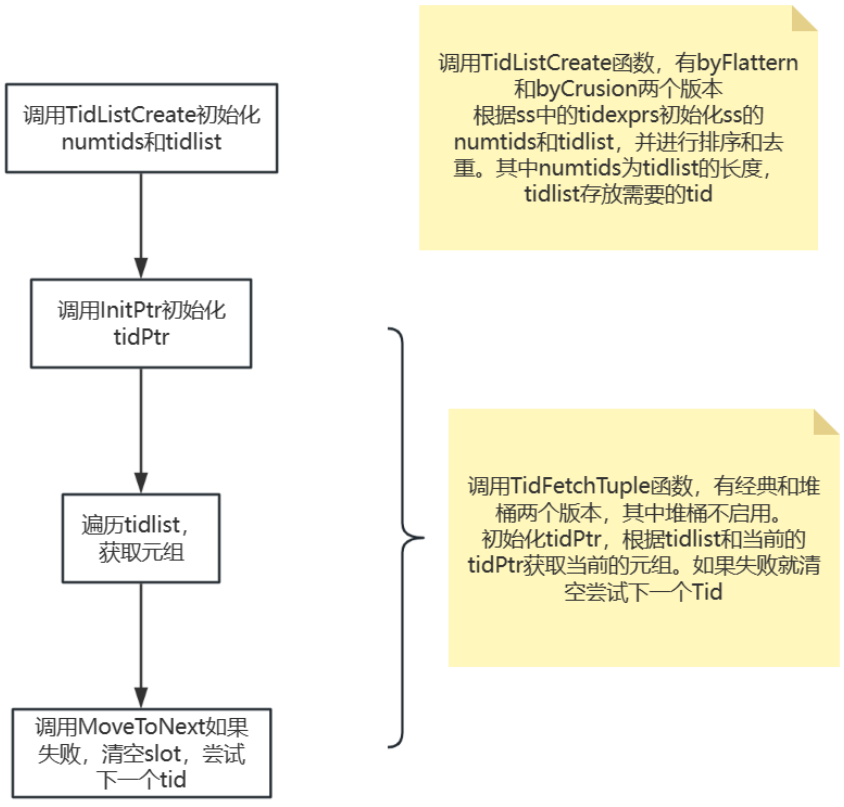

图 3-4 openGauss TidNext 函数流程图

3.1.2.3 部分底层 I/O 函数

heap_fetch 函数接收一个 tid，返回一个 HeapTuple 类型的元组，用于根据 tid 找到元组所在的数据页并获取该元组。heap_fetch 函数从数据块逐步调用函数到最后获取元组，按照如下的顺序：首先调用 ReadBuffer 读取 tid 对应元组所在的数据块；调用 BufferGetPage 函数获取元组所在的数据页；调用 GetOffsetNumber 获取 line_pointer 再数据页中的偏移量；调用 PageGetItemId 根据 page 和 offset 获取 line_pointer，并检查 line_pointer 是否有效；最后调用 PageGetItem 根据 line_pointer 获取元组。元组获取后保存在 Buffer 类型的缓冲中并返回。

3.2 索引扫描热点函数评测

为了优化 openGauss 索引扫描算子的性能，需要先对其热点函数进行研究。TPC-H 的测试语句为多种 SQL 操作的整合，不利于单独测试索引扫描。非聚集索引因为其索引顺序和磁盘排列顺序不同，所以必然是稠密索引，且不能为表的主索引。本文使用 TPC-H 的 lineitem 表，在 L_SHIPMODE 上建立索引。针对 1G 数据集下的 lineitem 表使用 Vtune 的热点函数分析工具进行测试，可以得到图 3-5 所示的结果。

Hotspots

INTEL VTUNE PROF

Analysis Configuration | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Flame Graph | Platform

Grouping: Module / Function / Call Stack

Module / Function / Call Stack	CPU Time	Module	Function (Full)	Source File	
gaussdb	16.587s				/home/omm/c
extent_recycle	1.871s	gaussdb	extent_recycle	extent.c	/home/omm/c
WaitLatchOrSocket	0.940s	gaussdb	WaitLatchOrSocket(Latch v...	pg_latch.cpp	/home/omm/c
MemoryContextAllocZeroDebug	0.897s	gaussdb	MemoryContextAllocZeroDe...	mcxt.cpp	/home/omm/c
drainSelfPipe	0.840s	gaussdb	drainSelfPipe(void)	pg_latch.cpp	/home/omm/c
GenericMemoryAllocator::AllocSetAlloc<(bool)1, (bool)0, (bool)0:	0.696s	gaussdb	GenericMemoryAllocator::Al...	aset.cpp	/home/omm/c
GenericMemoryAllocator::AllocSetCheck	0.679s	gaussdb	GenericMemoryAllocator::Al...	aset.cpp	/home/omm/c
heapgettup	0.639s	gaussdb	heapgettup(HeapScanDesc...	heapam.cpp	/home/omm/c
witness_assert_not_owner	0.530s	gaussdb	witness_assert_not_owner	witness.h	/home/omm/c
HeapTupleSatisfiesNow	0.310s	gaussdb	HeapTupleSatisfiesNow(He...	heapam_v...	/home/omm/c
nocachegetattr	0.300s	gaussdb	nocachegetattr(HeapTupleD...	heaptuple....	/home/omm/c
FunctionCallInfoData::FunctionCallInfoData	0.270s	gaussdb	FunctionCallInfoData::Functi...	fmgr.h	/home/omm/c
FunctionCall2Coll	0.240s	gaussdb	FunctionCall2Coll(FmgrInfo*...	fmgr.cpp	/home/omm/c
tsd_witness_tsdp_get	0.224s	gaussdb	tsd_witness_tsdp_get	tsd.h	/home/omm/c
hash_search_with_hash_value	0.209s	gaussdb	hash_search_with_hash_val...	dynahash....	/home/omm/c
UDFInfoType::UDFInfoType	0.180s	gaussdb	UDFInfoType::UDFInfoType...	fmgr.h	/home/omm/c
get_curr_candidate_nums	0.140s	gaussdb	get_curr_candidate_nums(C...	pagewriter...	/home/omm/c
_GLOBAL__sub_I_IndirectCallPromotionAnalysis.cpp	0.130s	gaussdb	_GLOBAL__sub_I_Indirect...		/home/omm/c
witness_tsdn_tsd	0.128s	gaussdb	witness_tsdn_tsd	witness.h	/home/omm/c
hash_any	0.120s	gaussdb	hash_any(unsigned char co...	hashfunc....	/home/omm/c
InvalidBaseEntry::Init	0.120s	gaussdb	InvalidBaseEntry::Init(void)	knl_locals...	/home/omm/c
buf_hash_operate<(HASHACTION)0>	0.119s	gaussdb	buf_hash_operate<(HASHA...	dynahash....	/home/omm/c
witness_tsdn_null	0.117s	gaussdb	witness_tsdn_null	witness.h	/home/omm/c

表 3-5 Vtune 性能分析结果

分析结果中，extent_recycle、WaitLatchOrSocket、drainSelfPipe 等函数分别为垃圾

回收以及 openGauss 针对 Latch 机制的处理函数，消耗性能占比高，主要原因是测试语句只针对索引扫描，较为单一，再加上数据量偏小，导致时间占比大的是 SQL 启动和收尾阶段的函数。但是可以看到，heapgettup 函数依然拥有不小的性能消耗占比。heapgettup 函数用于在存储数据表的堆上获取元组的函数，在索引扫描的 IndexNext 函数底层的数据访问层被调用。

综上可以看到，在索引扫描算子的流程中，对堆页面的读取相关的函数消耗性能较大。在索引扫描中，IndexNext 函数每次调用只会从堆中读取一个元组，这会导致程序需要反复地在用户空间和内核空间中切换，会带来许多额外的时间开销。

3.3 算法概述

根据并行同步 I/O 原理，本文尝试改进索引扫描使用的 IndexNext 函数，使其可以一次性从磁盘中读取多个元组，同时，基于排序索引扫描的工作原理，在 IndexNext 发出 I/O 请求的时候，I/O 请求会按照元组所在的数据页的顺序进行排序，得到一个临时列表，然后按照临时列表的顺序依次访问对应的数据页。开启同步并行 I/O 时，openGauss 的缓冲区管理器会将请求的页面放在临时 I/O 缓冲区列表中，并在接收整个 I/O 请求集的结果后返回改临时缓冲区列表。

下图是 openGauss 原生索引获取元组的示意图，下方的方框表示数据页号，中间的线表示索引项对应的元组在哪一个数据页，假设当前 SQL 语句请求了五个元组，那么会根据索引的顺序依次访问（6，1，3，0，4）五个数据页，且每次只访问一个数据页。

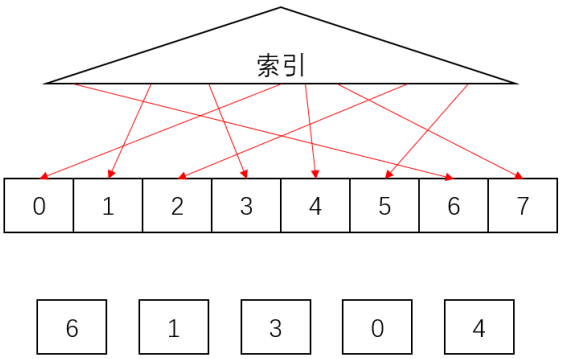

图 3-6 原生索引扫描获取元组

下图是经过优化后索引扫描获取元组的示意图，同样是请求五个元组的场景，利用

并行同步 I/O，可以将五次元组的请求组合成一个集合，一次性进行请求，并且，在获取到对应的 tid 后，不会直接请求，而是先根据 tid 的大小进行排序，然后再以一个整体的形式统一进行一次 I/O 请求。

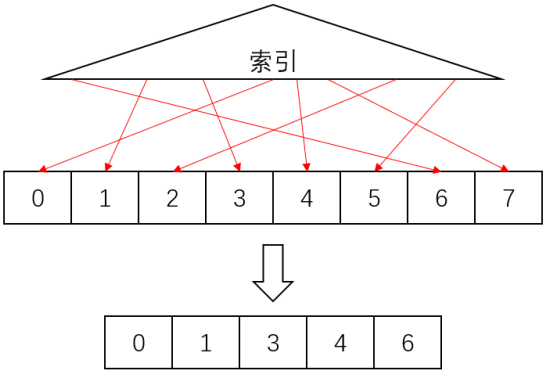

图 3-7 优化后索引扫描获取元组

3.4 排序索引扫描细节设计

openGauss 的 tid 扫描和排序索引扫描工作的原理较为相似，遂选用 tid 扫描算法作为排序索引扫描的基础，但是直接套用 tid 扫描本身实现有一定缺陷。在 2.4.2.1 小节提到，tid 扫描获取元组需要先调用 TidListCreate 函数获取元组的 tid，然后对 tid 按照磁盘排列顺序进行排序，然后再按顺序获取对应的数据元组。在创建 tid 列表后，排序索引扫描算法如果沿用索引扫描，会调用 index_fetch_heap 函数获取元组对应的 tid，但是 index_fetch_heap 其实会完整地获取整个数据元组，而后排序索引扫描会根据获取到的 tid 再去获取对应的元组，也就是说，排序索引扫描中会重复读取两次数据元组[10]。但是其实完全可以在不获取完整数据元组的情况下就获取元组的 tid，本文参考 Dasom 等人对于 TidNext 在排序索引扫描方向的改进，通过修改排序索引扫描中的函数，改进扫描流程，具体的函数实现细节请见 4.1 小节。

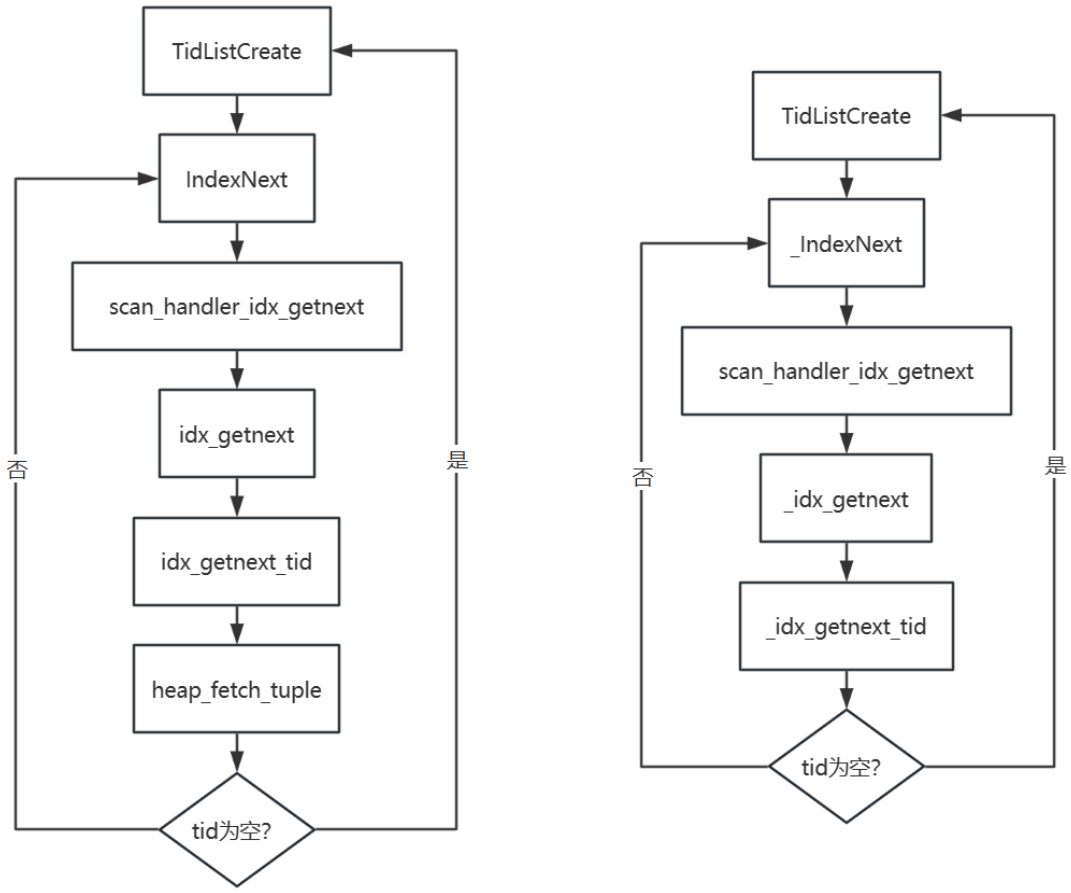

图 3-8 优化前后 tid 列表维护流程

3.5 并行同步 I/O 细节设计

本文使用 Linux 提供的直接 I/O 和异步 I/O 库 libaio 实现并行同步 I/O，以一次处理多个 I/O 请求。除此之外，针对优化后的索引扫描，设置了全局变量 MAX_LIBAIO，用以限制元组集中的元组数量。并添加了几个函数模拟 openGauss 中读取缓冲区数据页的函数，但是相较于原生函数，添加的函数可以一次处理多个缓冲区。为了尽量满足 2.5 节提到的三个要求，添加 md_read_start、md_read_insert 函数以符合要求 1，添加 md_aio_read、md_aid_end 函数以符合要求 2，要求 3 暂不满足。详细实现请见 4.1.2。

3.6 本章小结

基于排序索引扫描和同步并行 I/O 的索引扫描通过将多个待扫描元组请求组织起来，内部按照 tid 顺序排序，再一次性交给内核处理，充分利用存储顺序访存的优势，并利用 SSD 存储的内部并行性，提高 SQL 性能。排序索引扫描流程与 tid 扫描类似，设计

时仿照 tid 扫描的 tid 列表存储元组请求。同步并行 I/O 利用 Linux 的 libaio 库在磁盘管理器模块添加用于处理多个缓冲区的 I/O 接口。

第四章 Index Scan 算子优化算法实现及性能测试

4.1 算法实现

4.1.1 应用层

4.1.1.1 IndexScanState

根据排序索引扫描的需求，IndexScanState 添加了以下变量：

表 4-1 IndexScan 新增参数

变量名	数据类型	功能
xs_mbuf	Buffer[]	获取元组的缓冲区
iss_htup	HeapTupleData[]	用来存储元组
iss_cur_tuple	int	用于遍历缓冲区
iss_TidPtr	int	用于遍历 tid 列表
iss_mbufNum	int	buffer 大小
iss_remain_tid	int	tid 列表中可用 tid 数量
iss_fidx_state	FidxState	当前的扫描状态

xs_mbuf 是 openGauss 原生变量，但是原生版本下是 Buffer 类型，用于在调用 heap_fetch 函数时作为出参保存获取从堆上获取到的元组。iss_htup 同理。

其中 FidxState 为枚举类，表示排序索引扫描的中间状态，主要是便于循环获取元组时可以根据扫描状态及时打断。

4.1.1.2 _IndexNext 函数

_IndexNext 函数大部分仿照 TidNext 函数的流程。在 Tid 列表非空的情况下，遍历 tid 列表，遍历次数取决于 remain_tid 参数，remain_tid 为 tid 列表剩余可用的 tid 和请求集最大请求数 MAX_LIBAIO 的较小值。接着函数循环调用 heap_fetch_multiple 函数，一次性获取 remain_tid 个元组，存储到 xs_mbuf 中，然后通过 cur_tuple 遍历 xs_mbuf，循环调用 ExecStoreTuple 函数将缓冲区中的元组保存到 iss_htup 中。

4.1.1.3 qsort 函数

为了对 tid 列表进行排序，本文沿用 tid 扫描中为 tid 列表排序使用的 qsort 函数，并根据元组的数据结构，添加了排序使用的比较函数。

4.1.2 数据访问层

4.1.2.1 _idx_getnext 函数

不同于传统的索引扫描的 idx_getnext 函数需要通过获取实际的元组然后再提取出对应的 tid 信息，由于获取元组的功能由 heap_fetch 函数接管，idx_getnext 只需返回所需的下一个 tid，因此，_idx_getnext 函数的逻辑结构相对简单，只需要通过调用 index_getnext_tid 函数获取下一个 tid 并返回即可。

4.1.2.2 multiple 版本 I/O 函数

openGauss 原生的 I/O 函数，如 ReadBuffer 系列、heap_fetch 等函数仅针对单个 I/O 请求设计，无法处理 3.2 节所述的 I/O 请求集，于是，本文对 openGauss 原生 I/O 函数稍加修改，使其可以处理多个 I/O 请求。各个 Multiple 版本的 I/O 函数修改思路大致相同，本质上是通过循环遍历 I/O 或 tid 列表中的访存请求，然后调用原生 I/O 函数中对应的函数进行处理。接下来简单介绍以下 heap_fetch 函数的修改思路。

heap_fetch_multiple 函数和 heap_fetch 函数功能类似，都是根据 tid 从存储在堆上的表中获取元组，并通过缓冲区出参返回元组，不同的是 heap_fetch_multiple 函数一次处理获取多个元组，获取元组数为_IndexNext 传入的 mbufNum。

heap_fetch_multiple 函数首先调用 ReadBufferMultiple 函数通过传入的 tid 列表获取元组所在的数据块，保存在给定的缓冲数组中，然后遍历缓冲数组，调用 heap_hot_search_buffer 函数，该函数是 heap_fetch 函数功能的子集，可以获取元组所在的数据页和 line_pointer 对应的偏移量，然后通过调用 openGauss 的 Postgresql 内核提供的 pgstat_count_heap_fetch 宏获取 line_pointer 指向的元组。

4.1.2.3 smgr 下的函数

本文在 smgr 中添加了几个用于同步并行 I/O 的中间函数接口，其实现相对简单，只是对相关的函数进行了包装或者简单的调用，底层的实现主要集中在 md.cpp 文件中。smgr 中涉及的函数按照实际执行流程包括 smgr_read_start、smgr_read_insert、smgr_aio_read、smgr_read_end 等。接口通过函数指针的形式，指定下层 md 中实现的具体函数进行执行。

4.1.2.4 md 下的函数

md 文件中针对 smgr 中添加的四个函数进行特化处理。利用 libaio 库中的 io_submit、io_getevents 等函数实现。四个函数及具体功能见下表：

表 4-2 磁盘管理器下新增的 I/O 函数

函数	功能
md_read_start	为 md 读取操作初始化相关变量，分配空间
md_read_insert	提交 I/O 事件到 I/O 队列，等待处理
md_aio_read	从 I/O 队列中获取 I/O 事件，并进行处理
md_read_end	释放为 md 读取操作分配的空间

4.2 算法性能评测

4.2.1 测试环境

4.2.1.1 硬件环境

本文实验使用到的服务器的硬件信息如表 4-3 所示：

表 4-3 硬件环境

服务器	厂商	Huawei
	系列	TaiShan 200(Model 2280)
处理器	型号	Kunpeng 920-4826
	CPU	2 个
	每个 CPU 核心	48 个
	逻辑核心	96 个
内存	制造商	Hynix
	组合	32GB*8
	总大小	256GB
	类型	DDR4*8
磁盘	型号	INTEL SSDPE2KX040T8
	大小	4TB

4.2.1.2 软件环境

表 4-4 软件环境

操作系统	发行版本	openEuler 20.03(LTS)
	内核版本	Linux version 4.19.90-2003.4.0.0036.oel.aarch64
编译器	版本	gcc version 7.3.0 (GCC)
	优化选项	0
	架构选项	-march=aarch64
数据库	版本	openGauss LST 5.0.0
性能分析工具	版本	Vtune 2024.1.0
测试集	版本	TPC-H version 3.0.1

4.2.2 实验设计

4.2.2.1 测试数据集

TPC-H 是 TPC 组织提供的一套标准，用于进行 OLAP 数据库测试，它模拟了供应商和采购商之间的交易行为。TPC-H 的 22 条查询测试语句主要检验的是数据库如下的数据分析能力：聚合、连接、表达式计算、子表达式以及并行并发。在本文涉及的测试中，主要考察数据库索引扫描性能的提升，根据各个测试语句的执行计划筛选出其中使用到索引扫描的语句，后续的性能测试也仅会对这些语句进行测试。下面列出筛选后的测试语句和涉及的索引。

在筛选索引扫描相关的测试语句后，还需要对语句中索引扫描部分进行转换，这是因为本文对索引扫描的优化是基于 tid 扫描的，需要将索引扫描转换为对应的 tid 扫描语句，语句的转换依据为 Lee[12]提供的转换方法。

表 4-5 筛选后的测试语句

语句编号	索引扫描算子涉及索引
Q2	partsupp、part、supplier、nation、region 的主键
Q4	lineitem 的主键
Q5	customer、lineitem、supplier 的主键
Q7	nation、customer、lineitem 的主键
Q8	customer、lineitem、part、supplier 的主键
Q9	lineitem、partsupp、supplier 的主键
Q10	customer、lineitem 的主键
Q12	lineitem 的主键
Q15	supplier 的主键
Q18	customer、lineitem 的主键
Q20	partsupp 的主键
Q21	lineitem、supplier 的主键

```
/* 转换前 */
SELECT  *
FROM    tab
WHERE   a BETWEEN min AND max;

/* 转换后 */
SELECT  *
FROM    tab
WHERE   ctid = ANY (ARRAY (
                    SELECT  ctid
                    FROM    tab
                    WHERE   a BETWEEN min AND max)
                );
```

图 4-1 转换方法

实验数据由 TPC-H 提供的 dbgen 数据生成工具提供。

4.2.2.2 测试设计

对算法的测试主要围绕以下几个方面展开：数据量、选择度以及 NDP 优化开启情况。测试过程中使用 openGauss 的 timing 开关记录语句的运行时间。

A. 数据量测试中，数据集按大小分为 1G、5G、10G，其中 1G 指 TPC-H 的 8 张表总数据量为 1G。实验在不同数据量情况下，针对各测试语句在 openGauss 原生算子和优化后算子进行对比测试，分别记录对应的运行时间，计算其性能优化比例。

B. 选择度测试中，本文选择 TPC-H 中行数较多的表 orders 表，针对其主键 o_orderkey 进行带 BETWEEN 的 SELECT 查询。Orders 表采用 dbgen 生成 5G 数据集中的数据表，其 o_orderkey 最大值为 30000000。选择度按大小分为 1%、5%、10%、20%、50%和 100%，其中 1%是指设定 BETWEEN 谓语范围为 0 到 300000。实验在不同选择度情况下，针对范围 SELECT 查询在 openGauss 原生算子和优化后算子进行对比测试，分别记录对应的运行时间，计算其性能优化比例。

C. NDP 优化测试中，本文通过 openGauss 的 ndp_mode 开关控制是否开启 NDP 优化。实验使用 5G 数据集，针对各测试语句在是否开启 NDP 优化下进行对比测试，分别记录对应的运行时间，计算其性能优化比例。

4.2.3 实验结果

4.2.3.1 实验数据

表 4-6 1G 数据量下各测试语句运行时间

测试语句	原生运行时间/ms	优化后运行时间/ms	性能提升/%
Q2	2203	2076	5.76
Q4	8645	8292	4.08
Q5	7205	7335	-1.81
Q7	10433	10430	0.03
Q8	7542	7393	1.98
Q9	28527	26733	6.29
Q10	12319	12192	1.03
Q12	10091	8989	10.92
Q15	14097	14814	-5.08
Q18	35147	32237	8.28
Q20	15661	14668	6.34
Q21	19467	18083	7.11

表 4-7 5G 数据量下各测试语句运行时间

测试语句	原生运行时间/ms	优化后运行时间/ms	性能提升/%
Q2	4122	3851	6.57
Q4	47896	46205	3.53
Q5	47038	48468	-3.04
Q7	88453	88586	-0.15
Q8	55292	54313	1.77
Q9	268349	251685	6.21
Q10	72892	73111	-0.30
Q12	58196	50939	12.47
Q15	83968	87946	-4.74
Q18	176734	166307	5.90
Q20	111636	104212	6.65
Q21	158997	146452	7.89

表 4-8 10G 数据量下各测试语句运行时间

测试语句	原生运行时间/ms	优化后运行时间/ms	性能提升/%
Q2	5967	5735	3.89
Q4	99685	95389	4.31
Q5	104315	105911	-1.53
Q7	151890	151738	0.10
Q8	132701	129012	2.78
Q9	642386	604549	5.89
Q10	161091	159948	0.71
Q12	115228	104466	9.34
Q15	196485	202380	-3.00
Q18	351470	344265	2.05
Q20	256901	238841	7.03
Q21	364244	333939	8.32

整理表 4-6、表 4-7、表 4-8 可得到如图 4-2 所示对比图：

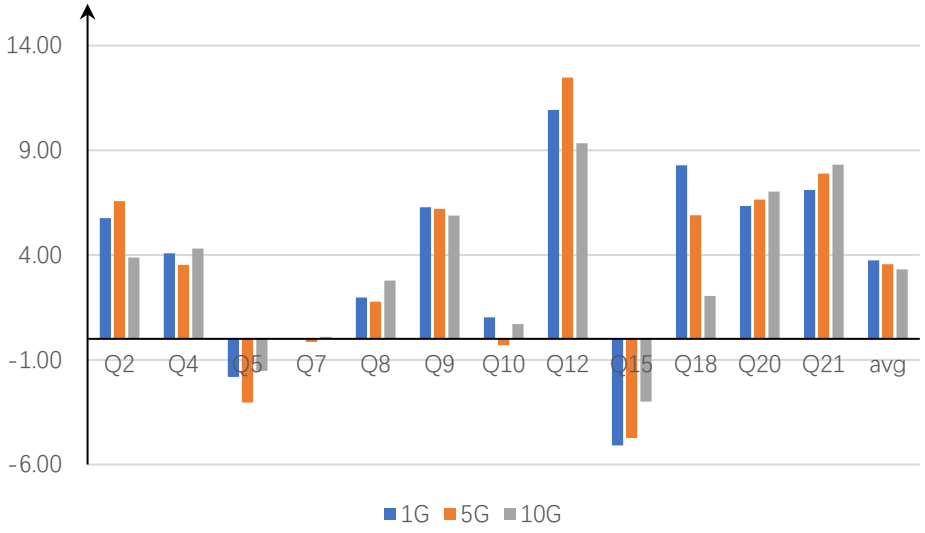

图 4-2 各测试语句优化比例对比图

除了 Q5 和 Q15，基于排序索引扫描和并行同步 I/O 的索引扫描相对于原生的索引扫描都有提升，总体平均也有性能上的提升，这是因为并行同步 I/O 通过将各个单个 I/O 请求组织起来，排序索引扫描将 I/O 扫描将 I/O 请求集进行排序，优化后的索引扫描在运行时可以尽可能利用服务器底层的 SSD 存储提供的内部并行性，通过并行减少索引扫描的运行耗时。

Q7 和 Q10 在各个数据量下的优化效果几乎可以忽略不计，通过 Explain Analyse 语句查看 Q7 和 Q10 实际的执行计划，可以看到这两个语句的计划树中并不包含索引扫描相关操作，而通过 Explain 语句可以看到 Q7 和 Q10 的执行计划中是包含索引扫描操作的，其中索引扫描操作在 Explain Analyse 的执行计划中被替换为了顺序扫描，这说明在测试语句实际运行的时候，SQL 优化器根据具体的算子运行代价对算子具体策略选择进行了优化，所以 Q7 和 Q10 两个语句并没有使用到索引扫描，无法使用优化后的索引扫描算法。

Q5 和 Q15 在各个数据量下都是负优化，但是总体的负提升比例并不大，主要的原因是 Q5 和 Q15 中索引扫描针对的索引都是在如 customer、supplier 等中等大小的表上。虽然都有针对 lineitem 表的索引扫描，但是由于扫描的选择度较高，索引扫描性能趋近于顺序扫描，性能消耗集中于对中等数据表的性能提升上。而在大小中等的表上，索引表的大小也是较小的，在这种情况下，基于排序索引扫描和并行同步 I/O 的索引扫描由于在扫描开始前需要进行额外的初始化，并且还需要维护 tid 列表进行 I/O 数组的请

求，消耗了部分空间和时间，所以优化后的索引扫描相较于原生的索引扫描并不占优。

横向对比 Q4 和 Q12，根据执行计划，这两个语句都只针对 lineitem 表上的主索引进行索引扫描操作，且整个语句仅有一个索引扫描操作，但是从图 4-2 可以看出两个语句的性能优化比例仍有差别，这是因为 Q4 中除了索引扫描之外还有排序、哈希聚合、顺序扫描等算子，索引扫描运行时间的占比相对 Q12 的显然较少，即使在索引扫描性能提升一致的情况下，整个语句的实际优化效果相对会差一些；不仅如此，Q12 语句中索引扫描操作的选择度比 Q4 的高，说明 Q4 在扫描过程中会获取更多的无用元组，这也在一定程度上带来了性能上的差异。

但是总体上看，测试语句的平均性能优化比例并不大，可能由以下原因影响：openGauss 针对运行时的计划树动态优化，体现在 Explain 生成的执行计划是系统根据各个算子的执行代价比例因子和基表元组数量综合选择总执行代价最小的计划[13]，但是 Explain Analyse 后可以发现部分原先的算子被替换了，如在一次语句测试中，读取相同的一张表，由于 SQL 缓存的存在，对表进行缓存，后续再次进行扫描可以减少磁盘 I/O 的代价，SQL 优化器可能会选择直接对 SQL 缓存中的表进行顺序扫描而不是再进行索引扫描。

表 4-9 不同选择度下运行时间

选择度/%	原生时间/ms	优化时间/ms	性能提升/%
1	5992	5146	14.13
5	8274	7993	3.40
10	9502	9894	-4.12
20	13029	14216	-9.11
50	23235	26148	-12.54
100	37437	46048	-23.00

综合 Q4 和 Q12、Q5 和 Q15 以及表 4-9，选择度的不同会影响到优化算法的性能提升，其中选择度越低，算法性能优化比例越大，在选择度高于 5%后，优化后算法较原生算法还会出现负优化的情况，原因是优化后的算法需要维护 I/O 列表的额外的空间时间开销已经大于优化算子时间。

表 4-10 NDP 优化前后各测试语句运行时间

测试语句	原生时间/ms	优化时间/ms	性能提升/%
Q2	4122	4121	0.02
Q4	47896	47987	-0.19
Q5	47038	46967	0.15
Q7	88453	89134	-0.77
Q8	55292	54988	0.55
Q9	268349	268081	0.10
Q10	72892	73176	-0.39
Q12	58196	58033	0.28
Q15	83968	83632	0.40
Q18	176734	178501	-1.00
Q20	111636	110977	0.59
Q21	158997	159538	-0.34

openGauss 的近数据处理（Neighbor Data Process，NDP）技术可以将 SQL 语句操作下推到页面存储中，在页面存储中就过滤掉非必要的数据，只将匹配的数据子集返回到数据库节点进一步处理，如要对数据表中的元组进行计数，那么这个计数操作就可以下推到数据页存储中，由存储来计数元组，并将计数而非实际的数据页返回数据库节点，使用 NDP 计数优化可以在特定情况下避免大量的网络流量，可以提升 SQL 语句的执行时间，对性能可能会产生一定影响，但是根据表 4-10 可以看出，在索引扫描算子中，NDP 优化开启后针对各测试语句的性能提升并不大，原因可能是虽然开启了 NDP 优化但是底层硬件或者数据库环境并不会使用到 NDP。

4.2.3.2 实验结论

优化后的索引扫描算法相较于原生的算法在 SQL 语句的选择度不低于 5%的情况为正提升，在 1G、5G、10G 数据量下对各测试语句的平均性能提升约为 3%-4%。

4.3 本章小结

使用 TCP-H 测试集在 1G、5G、10G 数据量下分别对 openGauss 原生和优化后的索引扫描算子进行测试，实验结果表明优化后的索引扫描的平均性能要优于原生算子。

总结与展望

1. 论文工作总结

近年来，数据库技术不断发展，人们不断追求更加高效更加快速更加安全的数据查询分析应用，而索引扫描作为 SQL 扫描中常用的算子，其性能高低与否对整个 SQL 查询效果影响是很大的。

本文主要研究在 ARM 架构下对 openGauss 数据库的索引扫描算子进行优化。本文结合 Dasom 等人提出的闪存敏感索引扫描[8]和 Roh 等人提出的并行同步 I/O 策略[9]，仿照其在 PostgreSQL 数据库中的实现，在 openGauss 数据库中对索引扫描算子流程进行改进。而后基于 TPC-H 官方提供的数据库以及 dbgen 工具生成的数据，使用 2.8 节筛选得到的与索引扫描相关的，优化后的索引扫描相较于原生索引扫描算子在运行性能上提升在-5%到 13%左右，平均性能提升在 3%到 4%左右。

优化后的索引扫描不同于原生的索引扫描一次只会发送一个 I/O 请求，而是设置一个大小至多为给定值的 I/O 请求列表，并且按照元组所在的数据页进行排序，然后根据排序后的 I/O 请求数组进行查询。每次处理 I/O 请求数组时都只需要按序扫一次磁盘即可，在随机访问代价比顺序访问代价大的存储设备，如机械硬盘上性能提升会更大。使用同步并行 I/O 可以更好地利用 SSD 内部的并行性，一定程度上提高 SQL 多并行环境下的运行性能。

2. 工作展望

虽然优化后的索引扫描总体上优化了执行性能，但是综合 4.2.3 节中各数据量下的平均性能优化比例来看，优化效果较为一般。未来的研究工作可以围绕以下几个方面进行：

A．针对 openGauss 特化优化

由于本文优化算法参考的两种优化方法[9][10]都是基于较早版本的 PostgreSQL 的，openGauss 内核在 PostgreSQL 的基础上进行了改进，所以参考的优化方法优化程度有限，通过优化当前算法的流程，利用 openGauss 针对自身数据库特化的优化策略或者更

高性能的接口函数、底层 I/O 函数等，也可以进一步提高算法性能。

B． 优化 SQL 访问路径选择

访问路径是指查询过程获取数据的方式，比如说全表扫描，索引扫描等。根据数据库的原生表统计信息等对访问路径进行优化[11]，亦或者通过 Reneta 等人[3]提出的称为平滑扫描的自适应访问路径选择运算符在算子运行过程中动态地调整当前的算子策略。

C．针对 ARM 架构进行优化

本文的优化策略并无针对 ARM 架构进行特殊化设计，而是 x86 和 ARM 架构都兼容的，后续的工作可以利用 ARM 架构的低功耗、高性能的特点进一步优化。

D．优化其他的扫描算子

本文提到的算法是基于 tid 扫描的，所以优化方案并不局限于索引扫描，对算法的流程略微调整，可以对顺序扫描、仅索引扫描等扫描算子也进行性能优化。

参考文献

[1] 王新硕. 面向 SELECT 和 SORT 的数据库算子缓存的设计与实现[D].华东师范大学,2023.

[2] Lothar F. Mackert and Guy M. Lohman. 1989. Index scans using a finite LRU buffer: a validated I/O model[J]. ACM Trans. Database Syst. 14, 3 (Sept. 1989), 401–424.

[3] R. Borovica-Gajic, S. Idreos, A. Ailamaki, M. Zukowski and C. Fraser, Smooth Scan: Statistics-oblivious access paths[C], 2015 IEEE 31st International Conference on Data Engineering, Seoul, Korea (South), 2015

[4] F. Mithani, S. Machchhar and F. Jasdanwala, A novel approach for SQL query optimization[C], 2016 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC), Chennai, India, 2016

[5] 罗晓. 基于计算型存储设备的 SQL 查询优化方法研究[D].华中科技大学,2022.

[6] 陈现森. 基于 openGauss 的异构算子加速技术[D].华东师范大学,2023.

[7] 谭国龙. 面向 GPU 的关系算子流计算方法研究[D].南昌大学,2023.

[8] D. -s. Hwang, W. -h. Kang, G. Oh and S. -w. Lee, Flash-aware index scan in PostgreSQL[C], 31st IEEE International Conference on Data Engineering Workshops, Seoul, Korea, 2015

[9] ongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, and Sangwon Lee, B+-tree Index Optimization by Exploiting Internal Parallelism of Flash-based Solid State Drives[J], Proceeding of the VLDB Endowment, Vol. 5, No.4

[10] 司一鸣. 面向倾斜数据的统计信息扩展和数据库连接算子优化[D].西安电子科技大学,2022.

[11] C. Ordonez, Optimization of Linear Recursive Queries in SQL[C], in IEEE Transactions on Knowledge and Data Engineering, vol. 22, no. 2

[12] Eun-Mi Lee, Sang-won Lee, and Sang-won Park, Optimizing Index Scan on Flash Memory SSDs[J], SIGMOD Record, December 2011, vol. 40, No.4

[13] P.G.Selinger, M.M.Astrahan, et.al. Access Path Selection in a Relational Database Management System, in SIGMOD, 1979.

致谢

首先，感谢汤德佑老师在这次毕业设计过程中对我耐心而专业的指导，使我得以顺利完成本次毕业论文。我认为一篇论文不能代表我在软件技术方面的水平，更不应该止步于此，而是要学习汤老师勤奋求真的治学态度，在软件技术领取开拓进取，学以致用，成为该领域的人才。

其次，感谢给我帮助的所有同学，在论文撰写出现困难时，很多同学都提供了高效的论文写作技巧，让我体会到同学之间互帮互助、团结奋进的真挚友情，这种温暖、团结、进取的学习气氛，让我感动，让我一生铭记。

最后感谢我的家人，是他们多年来对我学业的支持才让我走到这一步，才使我得以顺利完成学业。