软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(10):4661−4680 [doi: 10.13328/j.cnki.jos.006665]

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

基于图结构索引的分布式 OLAP 加速方法*

沈斯杰, 陈 榕, 陈海波, 臧斌宇

(上海交通大学 并行与分布式系统研究所, 上海 200240)

通信作者: 陈榕, E-mail: rongchen@sjtu.edu.cn

摘 要: 随着业务数据的规模增大, 一些重要的应用场景需要使用分布式在线分析处理 (OLAP) 支持大规模数据的分析, 例如商务智能 (BI), 企业资源计划 (ERP), 用户行为分析等. 同时, 分布式 OLAP 打破单机存储的限制, 可以将数据放在内存中以提升 OLAP 的处理性能. 然而, 基于内存的分布式 OLAP 在消除磁盘 I/O 后, 性能瓶颈转移到了连接操作. 连接操作是 OLAP 中的一种常用操作, 会进行大量的数据读取与计算操作. 通过对现有的几种连接操作方式进行分析, 提出了一种能够加速连接操作的图结构索引以及基于图结构索引的连接操作方式 LinkJoin. 图结构索引通过用户所指定的连接关系, 将数据在内存中的位置以图结构的形式进行存储. 基于图结构索引的连接方式, 不仅能够有等同于哈希连接的较低复杂度, 而且在执行过程中能减少数据读取与计算操作次数. 将目前先进的开源内存 OLAP 系统 MonetDB 从单机系统扩展成分布式系统, 并且在该系统上设计与实现了基于图结构索引的连接操作方式. 针对该系统的图索引结构, 列式存储以及分布式执行引擎这 3 个重要方面, 进行一系列设计与优化, 以提升系统的分布式 OLAP 处理性能. 测试结果表明, 在 TPC-H 标准测试中, 基于图结构索引的连接操作对于有连接操作的查询的平均性能提升达 1.64 倍 (最多达 4.1 倍). 对于这些查询中的连接操作, 性能提升达 9.8–22.1 倍.

关键词: OLAP 系统; 分布式系统; 连接操作; 索引技术; 图结构

中图法分类号: TP311

中文引用格式: 沈斯杰, 陈榕, 陈海波, 臧斌宇. 基于图结构索引的分布式OLAP加速方法. 软件学报, 2023, 34(10): 4661–4680. http://www.jos.org.cn/1000-9825/6665.htm

英文引用格式: Shen SJ, Chen R, Chen HB, Zang BY. Accelerating Distributed OLAP with Graph Structure Indexing. Ruan Jian Xue Bao/Journal of Software, 2023, 34(10): 4661–4680 (in Chinese). http://www.jos.org.cn/1000-9825/6665.htm

Accelerating Distributed OLAP with Graph Structure Indexing

SHEN Si-Jie, CHEN Rong, CHEN Hai-Bo, ZANG Bin-Yu

(Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai 200240, China)

Abstract: As the scale of business data increases, distributed online analytical processing (OLAP) is widely performed in business intelligence (BI), enterprise resource planning (ERP), user behavior analysis, and other application scenarios to support large-scale data analysis. Moreover, distributed OLAP overcomes the limitations of single-machine storage and stores data in memory to improve the performance of OLAP. However, after the in-memory distributed OLAP eliminates disk input/output (I/O), the join operation becomes one of its new performance bottlenecks. As a common practice in OLAP, the join operation involves a huge amount of data accessing and computation operations. By analyzing existing methods for the join operation, this study presents a graph structure indexing method that can accelerate the join operation and a new join method called LinkJoin based on it. Graph structure indexing stores the in-memory position of data in the form of a graph structure according to the join relationship specified by users. The join method based on graph structure indexing reduces the amount of data accessing and computation operations with a low complexity equivalent to that of the hash join. This study expands the state-of-the-art open-source in-memory OLAP system called MonetDB from a single-machine system to a

* 基金项目: 国家自然科学基金面上项目 (61772335)
收稿时间: 2021-11-03; 修改时间: 2021-12-17; 采用时间: 2022-03-03; jos 在线出版时间: 2023-01-04
CNKI 网络首发时间: 2023-01-05

distributed one and designs and implements a join method based on graph structure indexing on it. A series of designs and optimizations are also conducted in the aspects of graph indexing structure, columnar storage, and distributed execution engine to improve the distributed OLAP performance of the system. The test results show that in the TPC-H benchmark tests, the join operation based on graph structure indexing improves the performance on queries with join operations by 1.64 times on average and 4.1 times at most. For the join operation part of these queries, it enhances the performance by 9.8–22.1 times.

Key words: online analytical processing (OLAP) system; distributed system; join operation; indexing; graph structure

随着业务数据的不断增长, 分布式在线分析处理 (online analytical processing, OLAP) 应用在商务智能 (business intelligence, BI), 企业资源计划 (enterprise resource planning, ERP), 用户行为分析等不同场景[1,2]. 同时, 由于分布式系统能够打破单机系统在内存上的限制, 分布式 OLAP 系统能够将主要数据保留在内存中, 加速分析的性能, 提高查询的时效性. 因此, 一批分布式数据库与内存数据库已投入研究和使用以支撑内存场景的数据处理[2–7].

相较于传统基于磁盘的 OLAP 处理系统, 基于内存的分布式 OLAP 系统虽然规避了由于读写磁盘 I/O 而产生的性能瓶颈, 但是又出现了一些新的性能挑战. 连接操作 (join operation) 就是其中的一个典型. 连接操作是用来将多张数据库表进行笛卡尔积并且筛选出匹配一定条件数据的操作. 在 OLAP 处理中, 用户常用通过连接操作来得到跨数据库表的信息. 例如, 在用户行为分析中, 将用户表和订单表进行连接操作, 能得到用户的订单情况 (如用户平均消费等). OLAP 的基准测试 TPC-H[8]提供了 22 个标准查询, 其中有 20 个查询都直接或间接地使用了连接操作[9]. 连接操作是数据库中标准操作, 而且也是一个耗时的操作. 在基于磁盘的数据库中, 连接操作会导致大量的读写磁盘 I/O, 因此这些系统会通过减少 I/O 对连接操作进行优化, 例如优化的内嵌循环连接 (nested loop join), 哈希连接 (hash join), 排序归并连接 (sort-merge join)[10–13].

尽管分布式内存 OLAP 系统能够避免磁盘 I/O, 连接操作却仍旧是一个耗时操作, 此时连接操作的性能瓶颈已经转变为筛选匹配数据的操作. 尤其是对于一类稀疏连接 (sparse join) 的查询, 匹配选出的数据占所有数据的比例很低. 例如在 TPC-H 数据集[8]中, 与一条订单所对应的订单条目平均只有 4 条, 而订单条目的总数却在上百万条, 并且会随着伸缩因子 (scale factor) 上升而增大. 因此, 有必要对于分布式内存 OLAP 系统中连接操作进行进一步的性能优化.

近年来, 随着大数据技术的发展以及数据模型的复杂化, 为了表示数据之间的关联, 图结构被常用来存储这些关联数据, 例如 RDF 图, 社交网络图等. 同时, 一些用于查询与分析图结构数据的图处理系统也出现了, 它们将数据维护成顶点和边以体现数据之间的关联. 而对于关系型数据库, 实体关系模型 (entity-relationship model) 与数据库模式 (schema) 本身也是图结构. 我们观察到, 通过数据库表之间的关系 (relationship) 构建一个专门用于连接操作的图结构索引, 相较于使用传统的单表上的关系型索引 (例如 B+树索引, 哈希索引等), 能够有数量级上的性能提升 (实验见第 5.4 节). 因此, 图结构索引可以通过预处理的数据结构, 提升连接操作时间占比大的 OLAP 处理性能.

然而, 由于现有的业务数据大量基于关系型数据库, 使用关系型数据模型进行存储[10,14,15], 因此直接使用图处理系统或者图数据库 (如 Neo4j[16]等), 会带来数据格式转换的跨系统开销和额外的维护成本. 同时, 在关系型数据库上长期积累的优化经验 (例如列式存储模型及其执行优化), 也未在现有图数据库上得到完整移植. 而在关系型数据库中, 直接使用关系模型实现图结构, 也不能充分利用图结构的性能优势[17]. 因此, 如何在关系型数据库上进行图结构的设计, 使关系型数据在连接操作上得到图结构的性能, 是本文研究的主要内容.

本文提出了一个基于图结构索引的分布式 OLAP 连接方法 LinkJoin, 通过在关系型数据库上建立图结构索引以提升分布式内存 OLAP 系统中对于关系型数据的连接操作性能. 为了对基于图结构索引的连接方法 LinkJoin 进行验证, 我们首先将一个目前先进的开源内存 OLAP 系统 MonetDB (常被作为代表系统用于研究与分析[18–21]) 从单机扩展成了分布式系统, 并且使用了 LinkJoin 的连接方式 (本文将该系统简称为 LinkJoin 系统). 为了保证 OLAP 处理的性能, 系统使用了列式存储以利用 OLAP 处理过程中的数据局部性 (尽管 LinkJoin 系统在存储上使用了列式存储以加速 OLAP 处理, 但对于数据模型仍使用关系模型, 因此按照 C-Store[12], MonetDB[4]等相关系统的定义, 仍称为关系型数据库). 然而, 在使用图结构索引时, LinkJoin 和系统的设计将会面临性能方面的挑战, 例如如何实现一个高效的图索引结构, 如何保证图索引结构的使用和对数据结构的改造不影响其他操作的性能, 以及如何对

分布式执行引擎进行改造.

针对这些方面的挑战, LinkJoin 系统通过对图结构索引, 列式存储以及分布式执行引擎进行优化与改造, 充分利用数据的物理位置信息, 减少数据查询过程中的内存访问以及跳转. 我们在一个 4 台机器的集群中进行了系统的测试. 测试使用 TPC-H[8]标准测试, 通过增加图索引结构, LinkJoin 相比 MonetDB 原有的连接方式能够在带有连接的查询中平均性能提升 1.64 倍 (取决于连接操作占整个查询中的比重), 对于连接操作比重较大的查询, 整个查询的性能提升最多达到 4.1 倍. 对于这些查询中的连接操作, 使用 LinkJoin 所带来的性能提升达 9.8–22.1 倍. 我们将 LinkJoin 系统集成到一个分布式内存混合负载系统 VEGITO[22]中 (代码地址: https://github.com/SJTU-IPADS/vegito).

概括起来, 本文的贡献与组织结构如下.

(1) 通过对现有基于内存的分布式 OLAP 系统中的连接操作方式的分析 (第 2 节), 提出新型的基于图结构索引的连接方式 LinkJoin, 并给出其复杂度分析.

(2) 给出基于 LinkJoin 技术的分布式 OLAP 系统 (即 LinkJoin 系统) 的框架 (第 3 节). 针对系统设计的挑战, 提出 3 个方面 (图索引结构, 列式存储结构以及分布式执行引擎) 的优化和改造技术, 以实现一个高效的基于图结构索引的内存 OLAP 系统 (第 4 节).

(3) 将先进的 MonetDB 从单机扩展到分布式系统, 并进行了一组证明 LinkJoin 技术和系统有效性的实验, 以验证图索引结构在 OLAP 系统中能够加速分析查询 (第 5 节).

(4) 给出了 LinkJoin 及系统实现上的讨论与展望 (第 6 节), 以及其他相关工作分析 (第 7 节).

1 背景知识

1.1 在线分析处理 (OLAP)

在线分析处理 (OLAP) 是一种常见的数据库系统对于数据处理的模式, 常用于数据的查询分析, 例如商务智能 (BI), 企业资源计划 (ERP), 用户行为分析等 [1,2]. OLAP 的特征是对数据的操作以只读操作为主, 而对于数据的修改 (包括增加, 修改, 删除) 等操作会使用离线的批量加载方式. 由于这种特征, 对于 OLAP 中的数据处理能够避免读写冲突, 因此能够减少读写并发控制方面的考虑, 而查询处理的执行时间是 OLAP 系统性能优化的一个主要目标, 并且可以通过一些预分配的数据结构进行查询的加速优化 [1,5,6,23–27].

对于传统基于磁盘的 OLAP 处理系统, 磁盘的 I/O 开销是查询处理的一个主要性能瓶颈. 由于分布式系统能够打破单机内存的限制, 并且随着内存容量的扩大以及内存计算的发展, 传统基于磁盘的 OLAP 处理正在向着内存 OLAP 进行转变. 内存 OLAP 将分析所需要使用的数据存在内存中, 在分析过程中仅需要从内存中读取数据, 而完全避免磁盘 I/O 的开销. 目前, 已经有不少面向内存 OLAP 的数据库系统用于商业或研究使用, 例如 MonetDB[4], SingleStore[11], HyPer[18], SAP HANA[23]等.

基于内存的分布式 OLAP 处理系统不仅了消除了磁盘 I/O 的开销, 同时也使用了不同于磁盘数据库的新型内存技术, 通过充分发挥硬件特性以进一步提升分析处理的性能, 例如列式存储能够充分利用数据的局部性特征以最大化查询过程中的 CPU 缓存 (cache) 利用率 [4,11,12]; 向量化执行能够充分使用 CPU 的向量指令集加速对列式存储的数据计算 [24]; 根据现代处理器的多核特性使用并行技术缩短查询的执行时间和提高查询处理的吞吐 [26]; 以及减少中间结果的网络通信 [11]等.

1.2 连接操作 (join operation)

在关系型数据库中, 连接操作是用来组合两张数据库表信息的操作. 如图 1 所示, 我们简化了一个标准测试 TPC-H[8]中所使用的数据集: 在关系型数据库中有两张数据库表 ORDERS 与 LINEITEM, 分别存放订单和每个订单所属的条目. 订单表 ORDERS 会记录订单号 O_ID 与订单日期 O_ENTRY_D 等信息, 订单条目表 LINEITEM 会存放订单条目的主键 L_ID 以及所属的订单号 L_O_ID. 一个常用的查询会检查某类订单所属的条目, 这就需要用连接操作将两张表的信息进行组合. 连接操作的语义是将两张数据库表的信息进行笛卡尔积 (数据库表 R 与 S

的笛卡尔积, 即把 R 的每一行与 S 的每一行进行组合得到结果集, 该结果集的属性数 (列数) 是 R 与 S 的属性数之和, 结果集的元组数 (行数) 是 R 与 S 的元组数之积) 的操作, 然后再根据条件 (如 WHERE 从句后的条件) 进行筛选, 得到结果集.

```
SELECT    COUNT(*)
FROM      ORDERS, LINEITEM
WHERE     O_ID = L_O_ID AND O_ENTRY_D in […]
```

ORDERS

O_ID	O_ENTRY_D
…	…
1001	2021-08-07
1002	2021-08-07
1003	2021-08-08

⋈

LINEITEM

L_ID	L_O_ID
…	…
10001	1002
10002	1027
10003	1002
10004	1003

=

O_ID	O_ENTRY_D	L_ID	L_O_ID
…	…	…	…
1002	2021-08-07	10001	1002
1002	2021-08-07	10003	1002
1003	2021-08-08	10004	1003

图 1 关系型数据库的连接操作示例

由于连接操作在做笛卡尔积后的中间结果行数会等于原来两个表行数乘积, 然后再进行筛选, 因此一般在数据库中, 会将筛选和组合在一起完成, 以减少中间结果集的大小. 在实现上, 常用的 OLAP 处理系统, 如 SAP HANA[10], SingleStore[11], C-Store[12], Hyrise[13], MonetDB[4]中, 通常支持内嵌循环连接, 排序归并连接, 哈希连接等连接方式, 以及它们的优化和变种. 表 1 给出了这些典型 OLAP 系统所支持的不同连接方式, 这些连接方式可以供系统的用户根据不同的场景和性能要求进行选择.

表 1 典型 OLAP 系统对于不同连接方式的支持

连接方式	SAP HANA[10]	SingleStore[11]	C-Store[12]	Hyrise[13]	MonetDB[4]	Vectorwise[28]
内嵌循环连接	√	√	√	√		
排序归并连接		√	√	√		√
哈希连接	√	√	√	√	√	√

(1) 内嵌循环连接 (nested loop join, NLJ). NLJ 是和连接操作的语义最接近的实现方式, 通过嵌套的循环来遍历两张需要连接的表, 并且在内层循环中根据匹配条件进行筛选. 这种方式支持的筛选条件广泛 (例如相等, 不等, 字符串匹配等条件或者它们的组合条件). 对于基于磁盘的关系型数据库, 其主要的性能瓶颈在于数据从磁盘上读取的 I/O 开销. 以图 1 的查询为例, 外循环是 ORDERS, 内循环是 LINEITEM. 对于每一条 ORDERS 的数据, 都需要读取完整的 LINEITEM 数据, 即把这个表的磁盘页面依次读出. 设 ORDERS 的行数为 R(O), ORDERS 与 ORERLINE 的磁盘页面数分别为 P(O) 与 P(OL), 则 I/O 次数至少为 P(O)+R(O)×P(OL). 因此, 一些优化方式往往通过充分利用每次被读入的磁盘页面 (如通过分块的方式减少 I/O 次数), 或者针对新型的存储介质 (如 SSD) 进行特定的优化[29].

(2) 排序归并连接 (sort-merge join, SMJ). SMJ 是针对筛选条件中的列可排序, 并且是表内的顺序是按照该列进行排序的. 对于已经排序的表而言, 连接过程是一个简单的归并过程, 因此两张表只需各遍历一遍, 其 I/O 开销为 P(O)+ P(OL). 然而, 这种连接方式的问题在于: 其一, 对于没有排序的表而言, 需要按照筛选条件中的列进行排序, 引入额外的排序开销; 其二, 对于无法排序的列和筛选条件, 无法使用, 因此在语义上受限.

(3) 哈希连接 (hash join, HJ). 在连接操作的筛选条件是值相等的情况下, 为了进一步提高查询效率与减少查询过程中的不必要 I/O 开销, HJ 使用哈希结构加速连接. HJ 分为准备阶段与匹配阶段. 在准备阶段, HJ 使用了一个辅助的哈希划分, 通过尽量减少不必要的读取来提升连接速度. 在这一阶段中, 会将两张表中需要比对的列使用一

个哈希函数进行计算哈希值, 并且根据该哈希值将两张表的行分为不同的数据划分. 由于两个值相等的必要条件是它们的哈希值相等, 因此匹配的两行只可能是在对应的数据划分中进行. 所以, 在第 2 个阶段, 即匹配阶段中, 仅需要在对应的数据划分中进行匹配连接.

在以上所介绍的连接方法中, 对于两张表的连接操作, 都可以看作有两层主要的循环来进行数据的匹配和筛选, 这两层循环分别对应着两张表的顺序. 首先遍历第 1 张表, 对于第 1 张表的每一行, 通过不同的匹配方式 (例如嵌套循环中的遍历操作, 排序归并中的归并操作, 哈希连接中的哈希匹配操作等) 去筛选第 2 张表的数据. 因此, 在连接操作中, 由于上述的第 1 张表会出现在外层循环中, 会称之为外层表 (outer table). 与此对应, 第 2 张表成为内层表 (inner table).

1.3 索引 (index)

索引是数据库系统中用于加速数据查询的一种经典方法. 在关系型数据库系统中, 为了避免逐条查询某条数据, 会维护一种称为索引的数据结构, 将数据的某个属性 (列) 的值作为键, 通过索引映射到这个属性值所对应的数据的物理位置或主键. 因此, 根据应用的工作负载来建立合适的索引, 能够提升数据库的查询效率.

查询一般是针对数据的某个属性或者多个属性进行筛选. 根据筛选条件的形式可以将查询分为两类: 点查询与范围查询. 点查询给定的筛选条件一般是等值查询, 即筛选出某一个属性 (或者多个属性) 等于某个值的数据行. 与之相反, 范围查询的筛选条件是某一个范围 (例如数值的大于, 小于, 以及字符串的匹配操作 like 等).

与之相对, 索引结构主要有两类索引用以加速不同类型的查询: 哈希索引与树状索引. 哈希索引由于其 O(1) 的查找复杂度, 对于点查询有很好的支持, 然而却不能支持范围查询, 对于使用哈希索引进行范围查询, 将遍历范围内的每一个值从而转换为点查询, 这不仅在查找效率上会变差, 而且一些范围查询 (如浮点数的范围或者大范围的整数) 将无法进行转化. 树状索引 (例如 B+树, 以及其变种如跳表等) 单次查找的复杂度是 O(log$n$), 但是树状索引由于其顺序结构, 能够在给定最大值或最小值后选定一部分范围进行范围查询. 本文将这些关系型数据库中所使用的索引 (包括树状索引和哈希索引等), 称为关系型索引 (relational index).

因此, 一般而言, 在关系型索引中, 哈希索引对点查询更为适合, 而树状索引对于范围查询更为适合. 由于这些查询能够加速对于数据的查询, 因此也会用于加速连接操作 (例如在嵌套循环连接中, 对内层表使用索引进行筛选). 然而, 如何设计一个本身针对连接场景的索引, 仍旧是一个开放问题.

2 现有连接方式分析

相比基于磁盘的 OLAP, 内存 OLAP 的性能瓶颈已经不是原来的 I/O 开销, 而是转移为内存读取与复杂计算. 连接操作则是一个典型的代表, 在复杂的分析场景中, 跨表的查询十分常见. 以 OLAP 的基准测试 TPC-H[8]为例, 其 22 个查询中有 20 个查询使用了连接操作[9]以进行跨表的数据分析. 仅有 2 个查询 (Q01 与 Q06) 是进行单表内的数据分析而不需要进行连接操作. 而在我们的测试中 (实验配置: 4 台机器, 每台机器部署 20 个工作线程, 每台机器 SF=10), 连接操作的时间占比甚至会达到 80% 的时间 (详见图 2, 柱上标识连接操作占总查询时间的百分比). 因此, 需要对目前常用的连接方式进行算法层面 (第 2.1 节) 与实际运行层面 (第 2.2 节) 的分析, 以提出对于连接操作的优化方法, 从而对优化整个查询的性能有所助益.

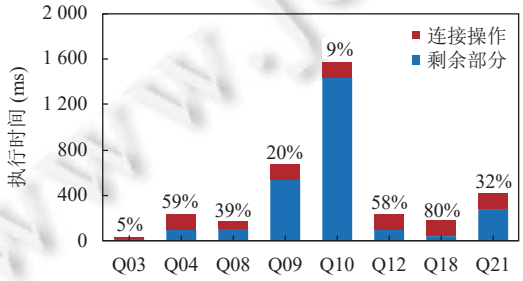


图 2 使用 MonetDB 对 TPC-H 中部分查询的查询时间拆解

2.1 常用连接方式的算法复杂度

在基于磁盘的数据库研究中, 对于连接操作的性能研究会从连接操作的 I/O 开销 (cost) 进行考虑, 例如对于不同连接操作所需要加载的磁盘页面数 (详见第 1.2 节). 对于内存数据库而言, 由于这样的 I/O 开销已经去除, 因此计算 I/O 开销以评估连接方式性能的方法已不合适. 因此, 我们需要通过研究不同连接方式的算法复杂度, 来评估它们在内存计算时的性能.

我们将不同的连接方式的使用分为先后两个阶段:准备阶段与匹配阶段. 第 1 阶段 (准备阶段) 是连接之前准备特定的数据结构, 例如排序归并连接中的排序以及哈希连接中的构建哈希表的操作. 第 2 阶段 (匹配阶段) 是按照连接条件进行对应行筛选的过程. 同时为了进行有效分析, 我们假设连接操作的两张表是“一对多 (one to many)”的关系, 即外层表的一条数据可能与内层表的多条数据匹配, 而内层表的一条数据至多与外层表的一条数据匹配[30]. 若没有这个假设, 则这些常用连接方法的复杂度在最坏情况下为 $O(m\cdot n)$, 例如外层表的所有数据都与内层表的所有数据匹配.

表 2 的第 2–4 列给出了内存中, 使用 3 种连接方式, 即内嵌循环连接, 排序归并连接以及哈希连接在两个阶段的时间复杂度[30–32]. 表 2 中, 进行连接操作的外层表与内层表的行数分别为 m 和 n. 对于排序归并连接, 表中给出的初始化复杂度为两张表是未排序时的复杂度, 未排序的表在进行归并操作前需要进行排序.

表 2 内存中不同连接方式的时间复杂度分析

阶段	内嵌循环连接	排序归并连接	哈希连接	内嵌循环连接 (树状索引)	内嵌循环连接 (哈希索引)	图结构索引连接LinkJoin
准备	—	$O(m\log m+n\log n)$	$O(n)$	—	—	—
匹配	$O(m\cdot n)$	$O(m+n)$	$O(m+n)$	$O(m\cdot\log n+n)$	$O(m+n)$	$O(m+n)$

通过表 2 的第 2–4 列可以看到, 相比于内嵌循环连接, 排序归并连接与哈希连接能够减少匹配阶段的时间复杂度, 但是需要不少时间进行数据结构的初始化准备, 而且对于归并-排序连接, 这种初始化时间可能会比匹配时间更长, 而哈希连接的实际匹配时间也与哈希函数的选择以及哈希桶的数目相关. 内嵌循环连接, 是不需要第 1 阶段进行数据结构的初始化的. 因此, 在数据库中, 也会使用内嵌循环连接以避免初始化开销, 其中一种常用的优化即基于索引的内嵌循环连接[33,34], 在 MySQL, SQL Server, PostgreSQL, SAP HANA 等商用数据库中也得以使用.

基于索引的内嵌循环连接是对于内嵌循环连接的一种有效优化, 该优化复用数据库中已有的索引结构, 因此不需要准备阶段. 其进行连接的复杂度与所选用的索引结构有关. 使用索引进行连接操作时, 类似于内嵌循环连接使用两层主要的循环, 其中外循环仍需要遍历外层表的每一项, 而内层表的遍历则变成了使用外层表的属性去匹配索引. 树状索引和哈希索引一次查找的复杂度为 $O(\log n)$ 与 $O(1)$, 同时最坏情况下需要遍历一次所有内层表的数据, 因此这两种索引结构下的连接操作复杂度分别为 $O(m\cdot\log n+n)$ 与 $O(m+n)$, 如表 2 的 5–6 列所示.

如表 2 所示, 在目前常用的连接方式中, 使用哈希索引的连接方式在内存计算时的算法复杂度最小, 其复杂度取决于外层表的大小. 而表 2 中的最后一列中, 使用基于图结构索引的连接方式 LinkJoin, 能够获得同样最小的复杂度 (详见第 3.2 节), 在此基础上, 我们进行系统层面的优化, 达到相比其他连接方式更优的连接性能.

2.2 连接操作的实际用时占比分析

为了进一步确定连接操作在内存 OLAP 处理中所占的时间比重, 我们选择使用一个先进的开源内存数据库 MonetDB[4]进行测试分析. MonetDB 在近年关于内存 OLAP 的研究中常被作为代表系统用于研究或者进行分析[18–21]. 我们在 MonetDB 上进行连接操作的实际用时占比分析, 并且建立了哈希索引进行连接操作的加速. 实验针对 OLAP 的基准测试 TPC-H 进行纯内存查询, 对于每个查询进行拆解分析, 即测量每个查询所用时间以及其中连接操作所占用的时间.

如图 2 所示, 我们选取了连接操作用时占整个查询用时的比例不小于 5% 的查询作为连接操作占比明显的查询进行分析. 可以看到在选取的 8 个查询中, 连接操作占整个查询时间的比重最高可以达到 80% (Q18). 在 22 个查询中, 有 7 个查询的连接操作用时比重超过 20%. 其中, 有 3 个查询的连接操作用时比重超过 50%, 成为整个查

询中的主要性能瓶颈. 可以发现, 即使使用算法复杂度较小的基于哈希索引优化的内嵌循环连接方式, 连接操作仍旧是在内存查询处理中耗时的操作之一. 因此, 对于一些连接操作用时占比较大的查询而言, 优化连接操作的时间能够对于优化整个内存查询的性能起到关键的作用.

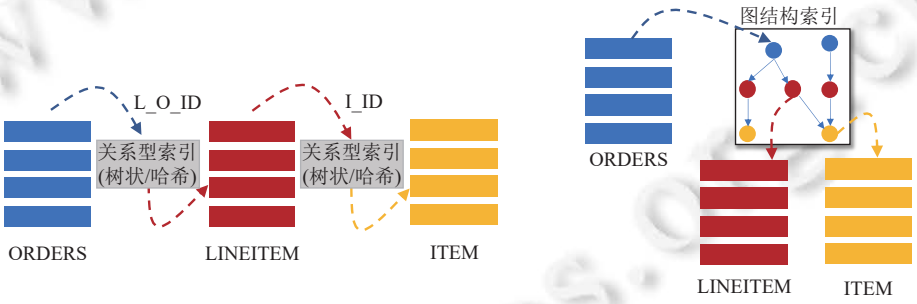
3 研究动机与系统架构

3.1 图结构索引

近年来, 随着大数据技术的不断发展, 数据不仅是从体量上增大, 数据之间的关系也越发复杂, 为了更好地刻画这些数据之间的关系, 相比原来关系型数据库中的关系模型, 用图结构存储数据能够更好地表达这些数据之间的关联. 使用图结构存放数据时, 数据分为点 (vertex) 和边 (edge) 两类数据, 点用于存放一些实体信息, 例如人员信息, 商品信息等, 而边存放点之间的关系, 例如购买情况等.

然而, 对于关系型数据库, 在数据库层缺少表与表之间的关系信息. 如图 1 中的示例, 订单 ORDERS 与订单条目 LINEITEM 的连接匹配只能通过值匹配来进行. 即使是使用了索引结构来加速连接操作也是需要通过索引结构的查询.

图 3(a) 给出了在关系型数据库中使用连接操作的执行过程. 示例中的连接查询使用了 3 张表: 订单表 (ORDERS), 订单条目表 (LINEITEM) 以及物品信息表 (ITEM). 一个常用的查询是, 通过订单查询订单中所购买的物品信息, 因此需要连接 ORDERS, LINEITEM 与 ITEM 这 3 张表. 在连接过程中, ORDERS 中每一条订单信息通过订单条目的编号 (L_ID) 来找到 LINEITEM 表中对应的订单条目, 订单条目再通过货品编号 (I_ID) 来找到 ITEM 表中对应的货品信息. 为了提升连接的性能, 在对应的属性列 (LINEITEM 的 L_ID 列, ITEM 的 I_ID 列) 上建立了关系型索引结构 (如树状或哈希索引). 对于使用索引进行连接操作, 对于每一行表中的数据, 都需要通过索引查询到另一张表中对应数据所在的位置. 如果是多张表的连接, 则需要在筛选出部分结果后继续进行索引的查询. 可以看到, 在关系型数据库中的索引本身是用于查询数据的加速, 而并不是为了连接操作而设计的, 只能减少连接操作中不必要的读取和筛选. 在图 3(a) 的示例中, 为了筛选一条结果集中的数据, 都需要经历两次索引结构的查询.



(a) 使用关系型索引结构 (b) 使用图结构索引

图 3 关系型数据库中 3 张表的连接操作示例

对于关系型索引而言, 进行连接操作的匹配是通过值的查询和匹配所完成的, 因此需要引入查询的开销. 而数据库在制定表的结构 (schema) 时, 所建立起的语义信息 (例如外键), 是缺失的. 而图结构能够在表达数据之间的关联, 提供更丰富的语义. 因此, 我们观察到是引入一个图结构的索引能够加速关系型数据库的连接操作, 减少连接过程中查询索引所带来的开销 (实验见第 5.4 节).

一个图结构的索引如图 3(b) 所示, 该实例中仍旧进行的是 ORDERS, LINEITEM 和 ITEM 这 3 张表之间的连接, 但是提供了一个图结构的索引. 图结构索引通过存储数据库数据之间的拓扑结构来加速连接操作. 在该索引中, 图的顶点表示数据库中的一行数据 (例如示例中的蓝色顶点表示 ORDERS 中的一行), 这个顶点仅存放这这条数据的位置信息 (例如同一台计算节点中的内存地址或偏移, 跨计算节点的节点编号和内存偏移等) 和类型, 而不

会存放实际的属性. 而图结构中的边, 则表示的是数据库数据之间的关系, 例如 ORDERS 所对应的顶点会有出边指向 LINEITEM 一个顶点 (红色顶点), 表示一条订单项. 以此类推, 一条 OERDERLINE 的数据也会指向一个 ITEM (黄色顶点). 由于顶点中存放着数据的位置信息, 因此, 可以根据出边直接在连接操作时定位对应的数据. 在图 3(b) 的示例中, 当进行 3 张表的连接操作时, 首先通过 ORDERS 中的一行数据找到它在图结构索引中所对应的节点, 然后通过其出边找到它的订单项, 并通过位置可以获取该订单项的实际数据. 同时, 可以通过订单项的出边, 找到 ITEM 的信息.

通过以上示例可以看出, 原有建立在数据库表上的关系型索引 (例如树状索引, 哈希索引等) 主要是为了直接加速单表查询而设计的, 而连接操作复用这些索引虽然能够减少一定的查询时间, 但是仍旧有一定的查询开销 (通过对多表进行连接操作的视图后建立关系型索引, 也是一种在效果上建立多表索引的方式, 然而对多表建立这种视图会产生大量的存储开销) 而图结构索引保存着数据之间的拓扑关系, 它的设计是为了加速连接操作, 与为了加速单表查询的索引相比, 图结构索引在加速连接操作时有以下几个优势: (1) 减少索引的查询开销. 由于图结构索引的节点中直接存放对应数据的位置信息, 因此可以通过 O(1) 的时间复杂度直接进行定位与查询. 尤其是当数据在其他计算节点上, 可以通过位置信息减少远端计算节点上的计算操作 (如数据查找和筛选) 而直接获取数据. (2) 提供数据之间的拓扑结构信息, 优化特定连接操作的执行. 例如, 对于 COUNT 操作能够直接通过出边的数量而直接返回. 对于没有匹配数据的也能够减少不必要的索引查询. (3) 减少多张表连接的索引查询次数. 当需要多张表进行连接操作时, 对于每一条结果集的数据, 索引的查询只需要遍历一次图结构即可完成, 而在为单表查询建立的索引需要在每张表 (除了最外层表) 上进行索引的查询.

3.2 复杂度分析

本节我们将从算法复杂度的角度来分析基于图结构的连接操作. 与基于索引的内嵌循环连接一样, 索引是在数据加载时离线计算的, 因此也不需要额外的准备阶段, 可以在进行连接操作时直接使用. 因此, 仅需考虑其匹配阶段的复杂度即可.

算法 1 给出了使用图结构索引进行连接操作的主要步骤, 设连接操作的外层表与内层表分别为 R 与 S. 与一般的基于索引的内嵌循环连接类似, 基于图结构索引的连接操作对于外层表的每一行进行遍历. 不同的是, 需要通过 locate_vertex 过程获取该行在图结构索引中所对应的顶点 (第 1–2 行). 由于图结构索引在建立时, 会将与该顶点的数据有匹配关系的数据作为该顶点的邻居进行存储, 因此内循环对该顶点的邻居进行遍历 (第 3–5 行). 由于为了空间节省, 在顶点的邻居中保存的是数据的标识符 (即算法中的 *s*_id) 而非数据本身, 因此需要通过 locate_row 过程进行定位 (第 4 行). 然后, 再将两张表中匹配的数据进行处理与输出 (第 5 行).

算法 1. 基于图结构索引的连接操作.

```
R: 连接操作的外层表
S: 连接操作的内存表
1. foreach r in R:
2.     v = locate_vertex(r)
3.     foreach s_id in v.neighbors:
4.         s = S.locate_row(s_id)
5.         process_and_output(r, s)
```

根据算法 1 主要步骤, 我们可以对该索引方法进行算法复杂度分析. 与第 2.1 节和表 2 一致, 我们设外层表和内层表的行数分别为 m 和 n. 并且为了进行有效分析, 同样假设连接操作的形式是两张表“一对多”的形式. 对于外层表的每一行数据, 需要通过其图结构索引得到与其匹配的其他表中的数据. 这个操作, 在图结构上的形式是根据数据的标识符 (id) 找到其对应的顶点, 然后通过边找到其邻居. 在图结构索引中, 定位一个数据的顶点 (locate_vertex), 定位一个顶点的邻居 (neighbors), 以及根据数据的标识符定位数据 (locate_row), 这些关键操作可以在

O(1) 时间完成 (详见第 4.1 节), 并且“一对多”的形式在最坏情况下能够保证至多对内层表的每条数据遍历一次. 因此, 与基于哈希索引的内嵌循环连接类似, 使用图结构索引能够在 O(m+n) 的时间内完成连接操作.

因此, 对于使用图结构的连接操作, 在算法复杂度上保证与复杂度为线性的常用连接方式有同样的复杂度. 并且, 在该图结构索引的设计与实现上, 能够减少其他索引结构中复杂的计算操作以及减少内存访问次数, 能够有比哈希索引等更好的连接操作性能.

3.3 图结构索引的构建

图结构索引的构建与关系型索引一样, 是离线进行的. 为了构建图结构索引, 需要对该索引所涉及的表进行预先的一次连接操作. 图结构索引仅保存连接的关系, 而不是所有连接的结果. 例如在图 3(a) 中, 对于 ORDERS 与 LINEITEM 之间的图结构索引, 需要根据连接条件 (即两个表中对应列的匹配) 进行连接. 并且, 在这个连接过程中, 并不需要产生最终结果, 而只是将对应的数据的位置记录下来, 保存在图结构中即可.

构建阶段的连接操作可以使用高效的哈希连接进行, 其构建的复杂度为 O(m+n), 并且由于不需要产生最终结果, 只需要记录数据的位置构成图结构即可. 在空间使用上, 图结构索引可以避免哈希连接中由于空桶产生的空间浪费 (实验见第 5.4 节).

3.4 分布式系统架构

为了对图结构索引进行进一步验证, 我们设计了一个基于图结构索引及其连接方式的分布式内存 OLAP 系统, 并且通过给关系型内存数据库提供图结构索引, 以提升内存中连接操作的处理性能. 该系统的架构如图 4 所示.

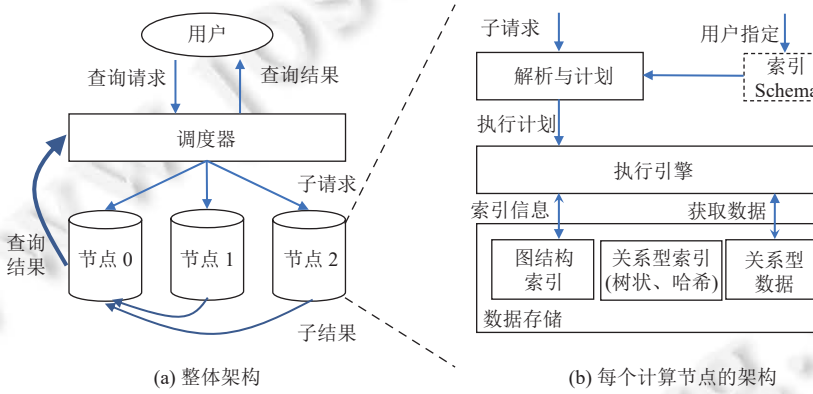


图 4 基于 LinkJoin 的分布式 OLAP 系统 (LinkJoin 系统) 架构

图 4(a) 给出了基于 LinkJoin 技术的分布式内存 OLAP 系统的整体架构, 系统称为 LinkJoin 系统. 该系统由一个分布式调度器以及若干节点所组成的, 所有的数据会划分成若干个部分 (partition), 每台机器会负责维护一部分数据, 使用零分享 (shared-nothing) 存储架构. 当用户发送请求后, 系统的调度器将会接收到请求, 并按照数据划分将请求拆解为子请求发给涉及的节点, 同时会设定一个节点作为处理该请求的主节点, 其他节点为从节点. 主节点和从节点都需要负责执行一部分子请求 (例如筛选数据的不同部分), 而主节点需要负责将各个从节点的子结果归并起来作为查询的结果返回给调度器, 调取器会将该查询结果返回给发起该请求的用户.

对于每一个节点内部的架构, 如图 4(b) 所示, 每个节点中系统分为 3 层: 解析层, 执行层与存储层.

解析层负责将用户的请求进行解析, 通过解析器与计划器等部件, 将用户的查询请求转换为执行计划. 在转化的过程中, 除了进行语言解析与常规的查询优化之外, 还需要根据用户定义的图索引结构 (schema), 来进行优化. 由于本工作的重点不在优化本身, 因此对于图结构索引, 我们选择在连接操作时能够使用图结构索引的, 均使用图结构索引. 算法 2 给出了 LinkJoin 系统中增加的一个 SQL 接口, 用于建立图结构索引. 其中, 图结构以有向边的形式连接两张关系表 out_table_name 与 in_table_name, 匹配关系由对应的列决定, 当对应列的数据匹配 (即值相等) 时, 进行边的连接. 有向边从 out_table_name 指向 in_table_name, 为了实现效率考虑, 一般在“一对多”关系下,

这两个表名分别对应“一”和“多”的表. 可以看出, 图结构索引与常规的关系型数据库的索引的最大区别在于, 普通关系型索引是建立在一张表上的, 而图结构索引建立在表与表之间的关系上.

算法 2. 建立图结构索引的 SQL 接口.

```
CREATE GRAPH INDEX index_name
    FROM out_table_name(column1, column2, ...)
    TO in_table_name(column1, column2, ...);
```

执行层负责执行由计划器所生成的执行计划. 为了提升系统性能, 如之前系统的设计[18,35], 执行层使用会话 (session) 的模式对查询请求进行并行处理. 当执行层收到请求后, 会把一个查询放在一个会话中, 该会话会调用多个线程并行处理一个查询请求. 同时, 多个会话也可以并行执行. 一般而言, 由于物理计算资源的限制, 并行的会话数越少, 一个会话中可以使用更多的并行线程处理查询, 减少该查询的执行时间.

存储层是一个内存存储, 负责存储关系型数据 (包括行式存储和列式存储), 关系型索引 (哈希索引和树状索引等) 以及用于连接操作的图结构索引. 所有的数据和索引将都会存储在本地或者远端节点的内存中. 为了尽量避免网络传输的开销, LinkJoin 系统遵循一般分布式 OLAP 系统的原则 (如 SAP HANA[23]和 SingleStore[11]), 在执行分布式查询时, 尽量让每个节点只处理本节点的本地数据, 只有在最后归并阶段再进行结果的合并. 对于每个数据划分都需要使用到的只读数据, 以备份的形式在每个节点中出现, 避免执行交换大量数据的全局连接操作.

3.5 系统设计的挑战

在对 LinkJoin 系统进行设计时, 会面临以下几个挑战.

首先, 需要实现一个高效的图结构索引. 图结构索引能够在连接操作的算法复杂度上带来和哈希索引一样带来最低的复杂度, 然而在使用图结构索引时, 如何避免使用复杂的计算与减少内存访问, 决定了图结构索引是否能够有良好的性能. 如果图结构索引在使用过程中, 在关键过程 (如定位, 查找, 筛选) 中使用了复杂的计算和额外的内存访问, 则会退化成哈希索引的性能, 甚至会更差. 因此, 需要设计在连接场景下的高效图结构索引.

其次, 图索引结构的设计不会引起其他操作性能的额外开销. 对于图结构索引的设计, 需要保证其结构不影响关系型数据本身的存储. 由于在 OLAP 处理系统中, 列式存储等优化常用于提升查询性能, 图结构索引需要保证能与这些关系型数据的优化正交, 保证这些常用的优化效果.

最后, 在关系型查询的执行过程中, 需要突破原来关系型数据执行的限制, 能够使用图结构索引进行连接操作的加速. 因此, 需要对优化器和执行引擎需要进行额外的设计, 以保证查询能够对图结构索引进行使用.

4 设计与实现

本节将描述一个提供图结构索引和 LinkJoin 技术支持的分布式内存 OLAP 系统——LinkJoin 系统, 并且提供列式存储的支持, 针对上文 (第 3.4 节) 所描述的系统设计的挑战, 本节从图结构索引 (第 4.1 节), 列式存储 (第 4.2 节) 以及执行引擎 (第 4.3 节) 来描述 LinkJoin 系统的设计与实现.

4.1 图索引的数据结构

为了提升连接操作的性能, 实现一个高效的图结构作为索引对于 LinkJoin 而言是十分必要的. 对于图结构的设计, 主要是从顶点的存储方式与边的存储方式进行考虑. 图 5(b)–图 5(d) 给出了图 5(a) 这个图结构的几种数据结构设计方式.

如图 5(b) 所示, 对于顶点的存储, 与关系型数据库中存储数据的方法类似, 顶点信息存在顶点表 (vertex table) 中. 顶点表中每一条数据表示一个顶点, 具体存储着顶点编号 vid (对应关系型数据库中的主键), 顶点类型 type (示例中包括订单 O, 订单条目 L 与物品 I), 以及其他需要存储的一系列属性.

对于图中边的存储, 如图 5(c) 与图 5(d) 中给出了两种主要的设计方式. 图 5(c) 给出了一种, 使用边表 (edge table) 存储边的方式 (如 SAP HANA[10], SQL Server[35], Virtuoso[36]等). 以出边的存储为例, 图中的一条出边, 存储着

起始顶点的编号 sid, 目标顶点的编号 tid, sid 与 tid 都对应顶点表中的顶点编号 vid. 此外, 还存储边的类型与其他所需要存储的属性. 此外, 图 5(d) 给出了基于邻接表 (adjacency list) 来存储边的方式 (如 Neo4j[16], WuKong[37], LiveGraph[17], RisGraph[38]等). 以出边的邻接表为例, 表中的每一项至少保留两个信息: 一个是存每个起始顶点的编号 (vid), 另一个是每个起始顶点会存一个链表, 即出边链表 (oe). 这个链表中记录的是该顶点的出边邻居 (即出边的目标顶点).

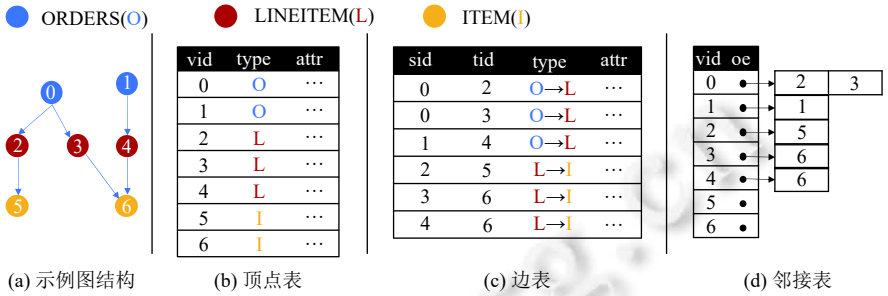


图 5 图结构的主要设计方式

为了使用图结构进行连接操作的加速, 顶点表中的属性需要存储该顶点所对应数据的内存地址. 而在边的设计上, 如果使用图 5(c) 中边表的设计, 会造成找邻居的过程仍旧变成两个关系型表的连接操作, 造成问题的递归. 而使用图 5(d) 中邻接表的设计, 能够在 O(1) 的时间复杂度内, 找出给定顶点的邻居. 因此, 作为关系型数据库的图结构索引, 使用基于邻接表的形式来存储边的信息更为合适.

然而, 如何找到该顶点对应的条目仍是个问题. 为了避免通过查找索引定位某个顶点在顶点列表中的条目, LinkJoin 中使用了一种优化, 即基于位置的邻接表.

LinkJoin 中观察到, 在关系型数据库中, 每一条数据中可以通过物理层的表名与行号的二元组, 即 (table_id, row_id) 二元组进行确定, 如果需要进一步定位到某条数据的某个属性 (字段), 则需要通过列的标识符 col_id 进行确定. 其中, 数据的行号是按照数据的物理存储自然地从 0 开始增长的. 因此, LinkJoin 的图结构索引将结合关系型数据库的这一数据存储特性进行存储. 对于顶点的存储, 在 LinkJoin 中直接复用原来关系型数据库的数据存储, 不会使用额外的数据结构与存储空间进行图中的顶点存储. 因此, 在一张表中, 顶点的编号 vid 将与数据的行号 row_id 一一对应. 对于边的存储, 在顶点列表中不会存储顶点的编号 vid, 而是直接存指向邻居的指针, 并且通过数据的行号进行索引.

图 6 给出了 LinkJoin 中的图索引结构的示意图. 与通用的图数据存储结构相比, 图索引结构使用关系型数据库中数据的物理位置信息 (即数据在每张表中的行号 row_id) 来减少存储和查询的开销. 在图 6 中, 以 ORDERS 与 LINEITEM 表为例. ORDERS 表上建立了到 LINEITEM 表的图索引, 因此在 LinkJoin 仅需要在 ORDERS 表上建立一个新的列 L 用来存储出边信息, 此外无需存储额外的顶点表与边表. 而在出边的邻接表中, 由于其排列的顺序与 ORDERS 的每一行数据在物理上的存储相同, 因此无需额外的一列存储顶点编号.

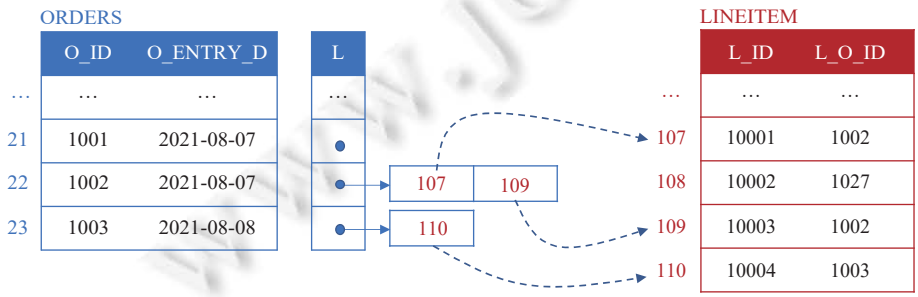


图 6 LinkJoin 中的图索引结构

在每条数据的出边表中, 存储着目标数据的物理行号. 例如对于 O_ID 为 1002 的 ORDERS 数据, 有两条 LINEITEM 的数据是与其匹配的即 L_ID 为 10001 与 10003 的两条数据, 这两条数据在 LINEITEM 表中的行号分别为 107 与 109. 因此, 通过存储物理偏移, 在 ORDERS 与 LINEITEM 进行连接操作时, 每条数据仅需要通过行号进行一次内存访问即可访问到对应的数据, 而不需要通过查询索引结构的复杂计算. 同时, 考虑到数据的局部性, 出边表的设计基于数组, 有利于数据的连续访问, 而是用链表会造成由于局部性和解引用造成的开销 (实验见第 5.4 节).

4.2 列式存储

为了进一步提升内存数据库中连接操作的性能, LinkJoin 使用了基于列式存储的数据存储设计. 在 OLAP 查询中, 其查询模式往往是针对全表中的某些列进行查询计算, 如计算数目, 平均值等操作. 而传统在 OLTP 数据库中所使用的行式存储 (row store), 即数据一行接着一行的存储方式, 无法充分利用 CPU 的缓存 (cache) 以及数据局部性. 因此, 对于 OLAP 数据的处理, 数据按照一列接着一列的存储方式, 能够提升分析处理的效率, 这种存储方式称为列式存储 (column store), 并且在 C-Store[12], MonetDB[4], SingleStore[11], SQL Server[39]等支持 OLAP 的系统中得到使用. 在这些系统中, 列式存储或作为一种全局存储的格式, 或作为在行式存储上建立列式存储的索引, 对 OLAP 查询进行加速.

列式存储是数据的一种物理存放方式, 不影响数据在逻辑上的表达. 逻辑上, 数据仍旧是按照行的关系型存储. 因此, 数据仍有行的概念, 只是一行的数据记录在不同的位置. LinkJoin 中所使用的列式存储是完全的列式存储, 所有表中数据的存储格式均为列式存储. 每一个列可以看做是一个连续的数组, 不同行的同一个属性依次排列 (例如图 6 中的 ORDERS 表中的所有 O_ID 存在同一个数组中), 而它们的排列顺序都是按照逻辑上的行依次排的. 在顺序上的设计能够保证同一行的不同属性虽然记录在不同的数组中, 但是它们在数组中的索引位置都是同一个. 由于不同列的数据大小 (即列宽) 是不同的, 因此内存地址 vaddr 要通过列的基地址 col_base, 列的宽度 (即属性的长度) col_width 与行号 row_id 决定: vaddr = col_base + col_width × row_id. 其中, col_width 在设计数据库表的结构时即已经决定, 对于定长的属性, col_width 即定长属性的长度, 对于变长属性考虑两种模式: 如果对于数据局部性要求高的列, 则列的宽度为该变长长度的最大值, 此外情况下存储指针即可.

LinkJoin 的列式存储中, 支持以下几种操作.

(1) 扫列 (scan). 这是在 OLAP 处理中一种常见的操作, 这种操作会将数据库中某个表的一列或者几列完整地读一次, 以获取该列的数据. 扫列的结果会做进一步的聚集操作等分析. 扫列可以作为读取数据的方式单独出现在查询中, 也可以与条件筛选等方式结合出现. 扫列操作能够充分利用数据的局部性, 也是列式存储最为适合的场景.

(2) 点查询 (point query). 点查询作为条件筛选查找的一种形式, 在某个列上进行等值查询, 例如在 TPC-H 基准测试中的查询会选取某个属性名为某个字符串的数据.

(3) 范围查询 (range query). 作为另一种条件筛选形式, 范围查询相对点查询而言, 不是进行值相等的筛选, 而是在某一个范围内 (例如大于, 小于, 字符串的 like 操作等) 进行筛选.

为了高效支持点查询与范围查询, LinkJoin 提供了哈希索引结构与 B+树索引, 分别支持点查询与范围查询.

4.3 分布式执行引擎

LinkJoin 系统中使用工作线程的模式进行查询的并行处理, 每一个 CPU 的物理线程对应一个处理查询的工作者 (worker). 并且, 使用会话 (session) 机制作为单个查询的并行度控制.

LinkJoin 系统中的并行度分为查询内并行与查询间并行. 查询间并行通过会话数目直接进行控制, 表示系统内同时有多少查询正被处理. 而查询内并行, 是通过会话数隐式控制的. 在极端情况下, 会话数为 1 则表示所有的线程处理同一个查询, 而会话数为最大值 (即可供查询的最大线程数), 那么每个查询是单线程进行处理的.

由于 OLAP 的查询均为只读操作, 因此查询间并行无需考虑并发控制的问题, 直接进行线程的划分即可. 而对于查询内的并行, 需要对查询中可并行部分进行合适的分解与划分. 我们观察到, 在 OLAP 的查询中, 大多都是先从扫表开始, 然后再进行条件筛选, 连接等操作. 因此, LinkJoin 中的并行从扫表入手, 将表的数据按照表的长度平

均分给每个负责这个查询的工作线程. 以下选取部分操作举例说明 LinkJoin 中的查询内并行的策略.

扫列, 筛选与聚集操作. 对于扫列与聚集操作, LinkJoin 将需要扫的列进行等长度的划分, 分给不同的查询线程, 每个查询线程在自己负责的范围内进行扫列, 并根据筛选条件进行数据的筛选, 然后将筛选出的数据进行聚集函数的计算, 形成子结果. 当每个线程完成自己负责的区域后, 会由一个线程将各个线程的子结果进行汇总计算出查询的结果.

连接操作. 由于连接操作涉及多张表, 对于连接操作的并行, LinkJoin 使用的并行划分是对最外层的表进行均等的划分. 对于每个工作线程, 将这些划分后的子表作为最外层表, 与内层的其他表进行连接操作, 连接后所得到的结果在并行阶段完成之后进行合并. 为了进一步提升并行度, 可以使内层表也进行划分, 然而这样的划分会增加合并结果时的开销以及实现的复杂度, 因此目前 LinkJoin 仅支持对于最外层表的并行划分.

分组操作. 对于分组操作的并行, 是在前文所描述的操作 (扫列, 筛选, 连接等) 的基础上进行的. 在每个工作线程执行这些操作时, 会在最后对中间结果进行分组操作. 当并行阶段结束后, 一个主线程会对所有的分组操作进行合并. 由于分组之后一般需要再一次的筛选和聚集等, 分组的中间结果需要保留必要的信息. 例如, 在计算分组平均值的时候, 需要保留每个工作线程在每个组内的计数与总和, 才能够保证最后主线程能够通过这些中间结果合并出正确的最终结果.

分布式查询. 对于分布式查询, 可以看作是一个部分工作线程在远端机器的并行化查询. 在执行引擎层面与上述的查询内并行类似, 然而并行的粒度为集群中的不同机器. 系统前端的调度器已经将查询任务按照数据并行拆分成不同的子任务了, 而子任务的拆分与将查询并行化的过程相同, 只是一部分任务交由远端的工作线程完成. 为了减少中间结果传输所带来的网络开销, LinkJoin 对于任务结果的归并, 分为两个阶段. 第 1 阶段将每台机器内的不同工作线程的结果归并并且发送给一个调度器所指定的主节点. 这一阶段的目的是为了避免所有工作线程发送中间结果给主节点. 第 2 阶段是主节点将收集到的结果进一步归并为查询的最终结果. 同时, 在进行分布式的连接操作时, 我们通过数据划分和建立副本的方式, 尽量规避需要传输大量数据的情况 (如符合条件的内层表数据散落在不同计算节点上, 该情况的讨论见第 6 节).

5 系统评测

为了测试 LinkJoin 利用图索引的连接操作对于分布式内存 OLAP 的性能提升, 我们在一个小型分布式集群上进行性能测试 (实验配置见第 5.1 节). 在测试中, 我们希望能够回答以下几个问题.

(1) 使用图索引的连接操作能够对分析处理整体带来多少性能提升? (第 5.2 节)

(2) 使用图索引能够对连接操作带来多少性能提升以及查询中其余部分的开销是多少? (第 5.3 节)

(3) 图索引结构的优化效果对于连接操作的性能提升如何? (第 5.4 节)

5.1 实验配置

硬件配置. 我们在 4 台装有 24 核 Intel Xeon E5-2650 处理器的机器上进行实验, 每台机器装有 128 GB 的内存, 以及两张 ConnectX-4 100 Gb/s IB 网卡. 每台机器上分配 20 个线程作为处理 OLAP 的工作线程, 剩余的线程作为监听消息, 转发查询等辅助线程.

测试基准. 我们使用一个常用的 OLAP 标准测试 TPC-H[8]作为测试基准. TPC-H 标准测试模拟了商务智能的分析场景, 总共有 22 个分析查询, 用于订单分析, 库存分析等不同的分析场景. 在分析处理过程中, 每一轮分析会将 22 个查询按照顺序依次进行. 其数据集的大小可以根据伸缩因子 (scale factor, SF) 进行调整, 一个 SF 对应的数据集大小约为 1 GB. 默认配置下, 每台机器所使用的数据量为 SF=10, 即整个集群的数据量为 SF=40.

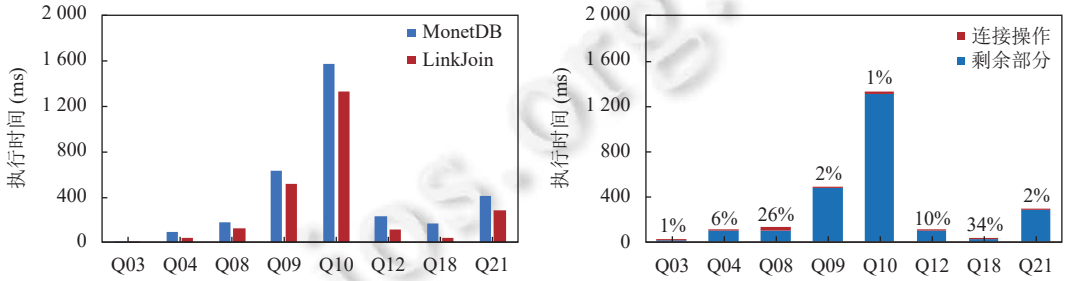
各系统与对比对象配置. 我们首先选取了一个先进的开源内存 OLAP 系统 MonetDB[4]作为基准, 并且把该系统从单机系统扩展成分布式 OLAP 系统. 对于分布式系统的架构, 我们根据分布式内存 OLAP 系统 SingleStore[11]的架构进行设计. 然后, 我们在该系统上实现了基于图结构的连接操作技术 LinkJoin. 为了对比的公平性, 我们需要消除查询计划所带来的对于查询执行的性能影响. 并且在执行时间上, 只统计查询执行的时间. 因此, 该系统在

使用基于图结构的索引操作以外, 其余的操作的查询计划与 MonetDB 使用相同的查询计划. 并且在测试 MonetDB 时, 使用的线程数和数据量也与 LinkJoin 保持一致.

5.2 整体性能

为了测试基于图索引结构的连接方式 LinkJoin 对于系统的性能提升, 我们在系统上分别对没有加上该技术 (以 MonetDB 表示) 和加上该技术 (以 LinkJoin 表示) 的情况分别进行系统性能测试.

我们对 LinkJoin 进行了 TPC-H 的整体性能测试并且与 MonetDB 的性能进行比较, 其测试结果如图 7 所示. 我们选取了 LinkJoin 能够进行加速的 8 个 TPC-H 查询: Q03, Q04, Q08, Q09, Q10, Q12, Q18, Q21. 由于 LinkJoin 的执行引擎是基于 MonetDB 的, 其余查询的表现与 MonetDB 的性能表现相同. 我们使用的会话数与工作线程数相同, 即每个查询线程独立处理一个查询会话. 我们测了这些查询在 MonetDB 与 LinkJoin 下的执行时间. 其中, MonetDB 和 LinkJoin 在这些查询中的几何平均执行时间分别为 207 ms 与 127 ms, LinkJoin 带来的加速比为 1.64 倍.



(a) 可加速查询的执行时间对比

(b) 可加速查询中各操作时间分布
(柱上标识连接操作占总查询时间的百分比)

图 7 使用 TPC-H 进行 MonetDB 与 LinkJoin 的整体分析性能比较 (SF=40)

对于每个查询, LinkJoin 带来的相对于 MonetDB 的加速比在 1.1 倍 (如 Q03) 到 4.1 倍 (如 Q18) 不等. 对于不同查询的加速效果, 主要取决于不同查询中的连接操作所占的比重, 连接操作占比越大, 则 LinkJoin 中的基于图结构的连接操作的加速效果越是显著. 各个查询中连接操作的所占比例如图 2 所示, 其中可以看到, 查询 Q03 中连接操作所占的比例为 5%, 因此加速连接操作对于整体查询的加速较少. 相反, 查询 Q18 中, 连接操作在执行时间中所占的比例为 80%, 因此加速连接操作能够获得 4.1 倍的加速比.

同时, 我们对于大数据集的情况下也进行了测试. 我们在每台机器上部署数据集大小为 SF=80 (即集群 SF=320), 测试结果如图 8 所示. 由于结果集随着数据规模的扩大而同比例增长, 查询内部各个部分的耗时比例也大致相同, LinkJoin 在加速表现上受数据规模影响较小, 其加速比达到了 1.57 倍, 与小数据集近似.

5.3 性能拆解分析

为了进一步分析 LinkJoin 对于连接操作的加速效果, 我们对这些查询进行了拆解测试, 通过测试其连接操作部分的执行时间以显示 LinkJoin 的加速效果, 其测试结果如图 9 所示. 对于连接操作的加速, LinkJoin 相对 MonetDB 的加速比在 9.8–22.1 倍. 对于不同查询, 加速比取决于连接操作时匹配的比例. 一般来说, 匹配的比例越小, 即连接越稀疏, LinkJoin 中图索引对于连接操作的加速情况更好. 反之, 如果匹配占比越大, 那么基于图索引的连接操作的加速比就相对变小了.

比较特殊的查询是 Q08, 其加速比只有 2.1 倍. 这是由于该查询中的连接操作具有对字符串筛选 (例如字符串的 like 操作), 不能够直接使用图结构索引进行加速. 我们对查询 Q08 进行了进一步的拆解, 分为了可以加速部分与不能加速部分的连接操作. 实验表明, 可加速部分的连接操作相比原来能够有 9.2 倍的加速比.

5.4 使用不同索引结构的连接性能

图 10 给出了一个基于 TPC-H 的微观性能测试, 在该微观测试中, 我们选取了 ORDERS 与 LINEITEM 进行一

对多的连接操作, 同时保证涉及的数据全部已经加载到内存, 以测试不同的索引结构给内存连接操作带来的性能(执行时间) 影响, 以及不同索引结构构建时间和构建所占用内存大小. 我们使用基于索引的内嵌循环连接 (该连接关系在关系型数据库的标准测试 TPC-C (在线联机事务的标准测试)[40]与 CH-benCHmark (混合负载的标准测试)[41]中的部分连接关系相同, 例如这两个测试数据集的 ORDERS 表与 ORDERLINE 表), 并且在内层循环中选取了 4 种索引结构的实现方式. 树状索引: 基于常用的 C++库 TLX 中的 B+树进行实现[42]; 哈希索引: 基于 MonetDB 中的实现方式; 图索引 (链表): 使用图结构索引, 并且在邻接表中使用链表的方式进行组织; 图索引 (数组): 使用连续的数组实现邻接表.

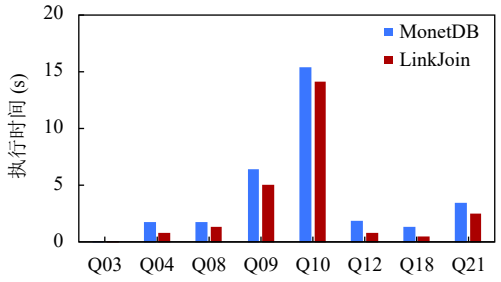


图 8 使用 TPC-H SF=320 规模下, MonetDB 与 LinkJoin 的分析性能比较

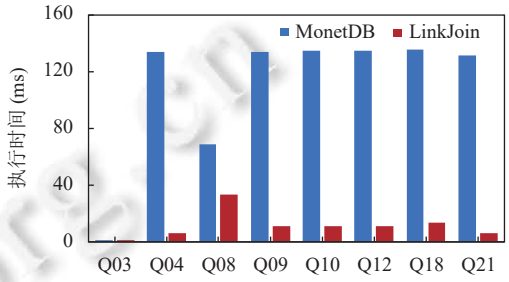


图 9 使用 TPC-H 进行 MonetDB 与 LinkJoin 的连接操作性能比较

执行性能表现. 由于哈希表的查询复杂度是常数, 而 B+树是对数复杂度 (如表 2 所示), 因此使用哈希索引进行连接操作的性能是 B+树索引的 1.9 倍. 而尽管图结构索引在算法复杂度上与哈希索引相同, 但是基于图索引的两种实现方式都能比树状索引结构与哈希的索引结构带来性能提升. 相比哈希索引, 链表实现的图结构与数组实现的图结构分别能都带来 4.4 倍与 6.5 倍的性能提升. 而相比于 B+树的索引结构, 这两种实现方式分别能带来 8.5 倍与 12.7 倍的性能提升. 这是由于, 在使用图结构索引时, 能够避免哈希值的计算操作, 以及减少内存跳次数. 相对于使用链表实现的图结构, 使用数组实现的图结构能够提升性能约 49%, 这是由于, 数组实现的图结构能够减少内存的解引用操作以及有更好的内存局部性. 该测试表明, 使用基于图的索引结构能够相比原有的针对单表查询的索引结构进一步提升内存中连接操作的性能.

构建性能表现. 在本实验中, 我们也给出了不同索引结构在构建时的时间成本和空间成本, 其结果如图 10 表示 (图中树状索引的构建时间与构建内存均超过纵轴最大值, 分别约为 11 s 与 16 GB). 在索引构建时间方面, 与最快的哈希索引的构建时间相比, 使用链表的图结构索引构建时间和使用数组的图结构索引构建时间分别为其 1.8 倍和 1.1 倍. 使用链表的实现方式, 主要的性能开销在于频繁分配链表中的节点. 而数组由于一次分配, 因此与哈希索引的构建时间近似 (且与使用哈希索引执行连接操作的时间接近). 在内存使用方面, 基于数组的图索引结构在内存使用上, 是哈希索引的 57%, 这是由于哈希索引中存在一定的空桶造成空间的浪费, 而在一对多连接中, 每个索引中的键值对至多出现一次, 不会造成浪费. 而图结构索引的内存使用, 会随着连接的复杂度而上升, 这是由于图结构中的边会增多. 而基于链表的图索引结构会因为链表中存指针的开销, 空间占用是基于数组的图结构索引的 1.2 倍, 哈希索引空间占用的 72%.

5.5 可扩展性

为了测试 LinkJoin 的可扩展性, 我们在系统中运行一个 OLAP 会话, 并且通过改变每台机器上处理该会话的工作线程数测试其执行 OLAP 请求的执行时间变化, 以表现该系统随着工作线程数变化的可扩展性. 由于 TPC-H 有 22 个标准查询, 我们按照 TPC-H 的标准文档[8]使用了这些查询的几何平均执行时间作为其性能表征. 同时, 在保证总体数据量相同的情况下, 我们也进行了分布式 (4 台机器) 与单机的性能对比测试. 该测试的结果如图 11 所示.

分布式系统性能. 通过测试可以发现, LinkJoin 的处理能力能够随着工作线程数的上升而提升. 这是由于, 在

提升工作线程数时, 各个工作线程会分摊对于扫表, 筛选以及一部分的连接操作. 相比单线程, 使用 2 个线程的性能是其 1.7 倍, 当使用 20 个线程时提升至 7.3 倍. 然而这个可扩展行不是完全线性的, 这是由于, 在 LinkJoin 系统中随着工作线程数上升后, 性能瓶颈从扫表转变为中间结果的合并, 并且中间结果和合并的工作量会随着线程数增大而上升. 对于这部分操作的并行与优化, 我们将作为后续工作进一步进行性能的优化.

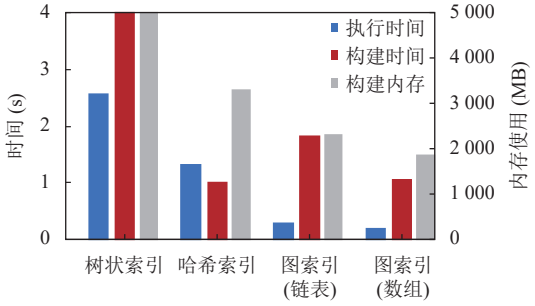


图 10 使用不同索引结构下的内存连接操作执行时间, 索引构建时间以及构建内存使用

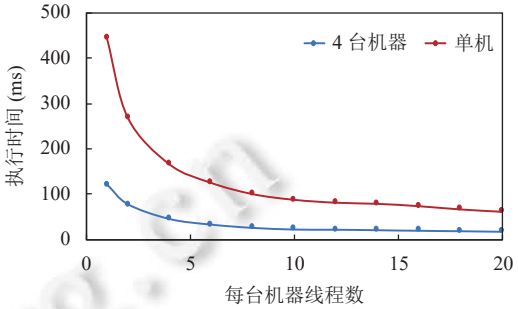


图 11 相同数据量, 使用不同线程数时的 OLAP 处理执行时间 (几何平均执行时间)

分布式与单机性能对比. 与单机性能相比, 使用 4 台机器的分布式系统, 在相同的总体数据量情况下, 4 台机器的分布式系统能够带来平均 3.7 倍的加速比. 为了进一步分析分布式场景下的开销, 我们继续在单机上使用分布式场景下每台机器的数据量 (即使用 1/4 的总体数据), 测试表明, 用于分布式的网络开销占整体查询时间的比例为 9%.

6 讨论与展望

图结构索引的动态调整. 由于对于 OLAP 系统而言, 数据的更新常常以定时批量的方式所执行的, 因此对数据的更新往往是以批量加载 (bulk load) 的形式进行更新, 而近年来又有数据进行实时更新的趋势[43]. 为了系统的通用性, 对于 LinkJoin 中的图索引结构而言, 也需要对这两种形式的更新进行一定的考虑. 对于批量加载, 由于是在离线时所进行的, 因此不需要考虑数据在加载时的读取情况. 因此, 通过对于数据增量 (delta) 的分析 (通常由数据源系统提供), 会触发图索引对修改的数据进行定位, 然后针对该数据对应顶点的邻接表进行修改即可. 对于修改较大的邻接表, 可以使用重新分配的方式减少碎片. 然而, 对于数据的实时更新而言, 对于图结构索引的实时结构会对并发控制带来一定的挑战, 可以结合原子操作与多版本并发控制 (MVCC) 减少图结构索引的更新对只读的分析处理造成的干扰. 这也是本工作的未来研究的方向之一.

完全分布式连接操作. 所谓的完全分布式连接操作, 是指在进行连接时需要将一个节点上完整的数据交由其他节点进行处理, 因此会造成数据的大量传输, 造成网络通信的开销增大. 对于完全分布式连接操作, LinkJoin 系统参照了 SingleStore[11], SAP HANA[1]等分布式 OLAP 系统的架构, 通过数据备份的方式以尽量避免分布式连接. 因此, 在 LinkJoin 系统中尚未考虑对分布式图索引的使用, 每台机器上的图结构索引是只针对本机器上的数据而建立的. 系统可以通过和普通数据相同的方式 (即增加备份), 减少分布式去查找图结构索引的开销. 如何高效地支持完整的完全分布式连接, 也是一个开放问题. 目前的策略是, 通过数据划分和备份等方式, 尽量减少这种耗时的完全分布式连接操作的出现[1,11].

数据的并行处理. 对于数据的并行处理, 这里主要讨论更为复杂的查询内并行. 目前实现的基于 LinkJoin 的 OLAP 系统中使用的并行方法, 是根据 MonetDB 中原有的并行方式, 即对于数据进行划分的数据并行. 数据并行有两个主要的问题可以进一步考虑, 一个是数据划分的负载均衡, 另一个是归并阶段的进一步并行化. 负载均衡问题是指, 由于数据的冷热程度 (即被访问或筛选的频率) 不同, 造成不同工作线程的实际处理的数据量有差异. 例如, 尽管给两个工作线程分配相同数量的数据, 但是经过一个筛选之后, 一个工作线程接下去要处理的数据远大于

其他线程, 就会造成负载不均的现象, 导致一个工作线程过载, 其他工作线程空转的情况, 造成资源浪费. 另一个问题是, 目前 LinkJoin 系统的归并阶段由一个线程所完成, 可进一步考虑归并阶段的并行性. 这些问题的解决和优化, 都是与目前在 LinkJoin 系统上的图索引与列存等技术所正交的.

图结构索引的其他使用场景. 图结构索引技术除了前文介绍的场景以外, 也同样可以使用在完全分布式连接操作和外存系统中. 在完全分布式连接操作中, 其使用思路是, 在全局建立统一的图结构索引, 在索引结构不太大的情况下可以通过副本的形式保存在不同计算节点上, 也可以通过图划分算法[44]将该结构划分在不同计算节点上. 对于外存数据库, 图结构索引能够相比视图减少存储开销, 因此能够将图结构索引缓存在内存中, 减少磁盘的读取开销, 从而能够加速外存数据库中连接操作的性能.

7 相关工作

连接操作的优化. 目前, 有一系列的数据库连接的加速方法以应对不同的工作负载和不同的软硬件环境. 面对磁盘等存储介质, 连接操作的主要瓶颈在于 I/O 操作的开销, 因此会使用索引减少筛选范围, 减少读取磁盘的次数[45]. 同时, 随着存储介质的变化, 一些工作会针对不同的存储介质来优化连接操作, 例如在 SSD 上的连接操作需要考虑 SSD 基于块的存储特性来设计分块连接以延迟 SSD 的使用寿命[29], DigestJoin 连接框架[46]通过分析闪存(flash) 的随机读写特性, 增加一定的随机读来减少中间结果集的大小. 针对内存数据库, 李梁等人[47]提出了基于跳表的优化方法, Begley 等人[48]提出一种优化的哈希连接方式 MCJoin, 旨在在连接过程中减少对于内存的使用. Arroyuelo 等人[49]提出了针对三元组连接操作的优化方法, 通过使用一种“环 (ring)”的数据结构作为索引, 保存三元组中属性的环状关系, 加速三元组连接的执行. 薛翔等人[50]提出了在分布式场景下, 使用数据备份重建索引, 加速范围查询的方法.

图系统与关系型数据库的结合. 近期有一些工作观察到在现有的关系型数据库上直接执行图计算是低效的, 因此为了兼容现有关系型数据, 进行了图系统与关系型数据库结合的尝试[10,14,33,51]. 赵展浩等人[14]将图数据转换为关系型数据, 并且利用关系型数据库的接口进行图算法的计算. Paradies 等人[10]在关系型数据库 SAP HANA 上通过用户在 SQL 语句中指定顶点表和边表, 在关系型数据上提供图处理的接口. Bornea 等人[33]为了提升在关系型数据上执行图算法的性能, 提出的 G-SQL 系统[52]在原有的关系型内存数据库上, 通过外键关系增加图的拓扑结构信息, 并且提出一种 SQL 方言提供图查询能力, 使得用户能够在关系型数据上能够高效执行图算法. SQLGraph 系统[35,51]也通过在 MySQL 和 HBase 等关系型数据库上提供图抽象, 以执行图算法. 与之设计目标不同, 本文所提出的基于 LinkJoin 的分布式 OLAP 系统旨在使用图结构提升关系型查询本身.

此外, 对于分布式内存 OLAP 系统的设计方法, 还有其他优化的方向. 例如对于列式存储以及列式存储上的连接方法的优化[12,25], 在多核场景下的并行优化[26], 以及在分布式场景下的减少网络通信的优化[11]等.

8 结 论

本文提出了一个基于图结构索引的分布式 OLAP 加速方法 LinkJoin, 通过对现有连接方式的分析, 提出了在关系型数据上建立一个用于连接操作的图结构索引以提升连接操作的性能. 为了应对 LinkJoin 技术在系统设计上所带来的性能挑战, 系统通过对图结构索引, 列式存储结构以及分布式执行引擎这 3 方面的改造与优化, 实现了高效的基于图结构索引的连接方式. 在 TPC-H 标准测试中, LinkJoin 相比 MonetDB (扩展为分布式系统) 的原有连接方式, 对有连接操作的查询带来平均 1.64 倍的加速比, 最多达到 4.1 倍. 其中, 对于这些查询中的连接操作部分, 能够带来 9.8–22.1 倍的加速比.

References:

[1] May N, Böhm A, Lehner W. SAP HANA —The evolution of an in-memory DBMS from pure OLAP processing towards mixed workloads. In: Mitschang B, Nicklas D, Leymann F, Schöning H, Herschel M, Teubner J, Härder T, Kopp O, Wieland M, eds. Datenbanksysteme für Business, Technologie und Web (BTW 2017). Bonn: Gesellschaft für Informatik, 2017.

[2] Rantung VP, Kembuan O, Rompas PTD, Mewengkang A, Liando OES, Sumayku J. In-memory business intelligence: Concepts and performance. IOP Conf. Series: Materials Science and Engineering, 2018, 306: 012129. [doi: 10.1088/1757-899X/306/1/012129]

[3] Neumann T, Mühlbauer T, Kemper A. Fast serializable multi-version concurrency control for main-memory database systems. In: Proc. of the 2015 ACM SIGMOD Int'l Conf. on Management of Data. Melbourne: ACM, 2015. 677–689. [doi: 10.1145/2723372.2749436]

[4] Idreos S, Groffen FE, Nes NJ, Manegold S, Mullender KS, Kersten ML. MonetDB: Two decades of research in column-oriented database. IEEE Data Engineering Bulletin, 2012, 35(1): 40–45.

[5] Shanbhag A, Tatbul N, Cohen D, Madden S. Large-scale in-memory analytics on Intel® Optane™ DC persistent memory. In: Proc. of the 16th Int'l Workshop on Data Management on New Hardware. Portland: ACM, 2020. 4. [doi: 10.1145/3399666.3399933]

[6] Daase B, Bollmeier LJ, Benson L, Rabl T. Maximizing persistent memory bandwidth utilization for OLAP workloads. In: Proc. of the 2021 Int'l Conf. on Management of Data. China: ACM, 2021. 339–351. [doi: 10.1145/3448016.3457292]

[7] Wei XD, Dong ZY, Chen R, Chen HB. Deconstructing RDMA-enabled distributed transactions: Hybrid is better. In: Proc. of the 13th USENIX Conf. on Operating Systems Design and Implementation. Carlsbad: USENIX Association, 2018. 233–251.

[8] TPC Becnhmark™ H Standard Specification (Revision 3.0.0). Trans. Processing Performance Council (TPC). 2021. https://www.tpc.org/information/benchmarks5.asp

[9] Boncz P, Neumann T, Erling O. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In: Proc. of the 5th TPC Technology Conf. on Performance Characterization and Benchmarking. Trento: Springer, 2013. 61–76. [doi: 10.1007/978-3-319-04936-6_5]

[10] Paradies M, Kinder C, Bross J, Fischer T, Kasperovics R, Gildhoff H. GraphScript: Implementing complex graph algorithms in SAP HANA. In: Proc. of the 16th Int'l Symp. on Database Programming Languages. Munich: ACM, 2017. 13. [doi: 10.1145/3122831.3122841]

[11] SingleStore. 2022. http://www.singlestore.com/

[12] Stonebraker M, Abadi DJ, Batkin A, Chen XD, 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.

[13] Grund M, Krüger J, Plattner H, Zeier A, Cudre-Mauroux P, Madden S. HYRISE: A main memory hybrid storage engine. Proc. of the VLDB Endowment, 2010, 4(2): 105–116. [doi: 10.14778/1921071.1921077]

[14] Zhao ZH, Huang FR, Wang XL, Lu W, Du XY. SQL-based solution for fast graph similarity search. Ruan Jian Xue Bao/Journal of Software, 2018, 29(3): 689–702 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5449.htm [doi: 10.13328/j.cnki.jos.005449]

[15] Xin JC, Wang GR, Li GH, Gao YJ, Zhang ZQ. State of the art data model and its research progress. Ruan Jian Xue Bao/Journal of Software, 2019, 30(1): 142–163 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5649.htm [doi: 10.13328/j.cnki.jos.005649]

[16] Neo4j Graph Database. 2022. http://neo4j.org/

[17] Zhu XW, Feng GY, Serafini M, Ma XS, Yu JP, Xie L, Aboulnaga A, Chen WG. LiveGraph: A transactional graph storage system with purely sequential adjacency list scans. Proc. of the VLDB Endowment, 2020, 13(7): 1020–1034. [doi: 10.14778/3384345.3384351]

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

[19] Boncz PA, Zukowski M, Nes N. MonetDB/X100: Hyper-pipelining query execution. In: Proc. of the 2005 CIDR Conf. 2005. 225–237.

[20] Dreseler M, Boissier M, Rabl T, Uflacker M. Quantifying TPC-H choke points and their optimizations. Proc. of the VLDB Endowment, 2020, 13(8): 1206–1220. [doi: 10.14778/3389133.3389138]

[21] Zhu YA, Zhang YS, Zhou X, Wang S. Column-oriented query execution engine for OLAP based on triplet. Ruan Jian Xue Bao/Journal of Software, 2014, 25(4): 753–767 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4568.htm [doi: 10.13328/j.cnki.jos.004568]

[22] Shen SJ, Chen R, Chen HB, Zang BY. Retrofitting high availability mechanism to tame hybrid transaction/analytical processing. In: Proc. of the 15th USENIX Symp. on Operating Systems Design and Implementation. USENIX Association, 2021. 219–238.

[23] Lee J, Moon S, Kim KH, Kim DH, Cha SK, Han WS. Parallel replication across formats in SAP HANA for scaling out mixed OLTP/OLAP workloads. Proc. of the VLDB Endowment, 2017, 10(12): 1598–1609. [doi: 10.14778/3137765.3137767]

[24] Kersten T, Leis V, Kemper A, Neumann T, Pavlo A, Boncz P. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. Proc. of the VLDB Endowment, 2018, 11(13): 2209–2222. [doi: 10.14778/3275366.3284966]

[25] Wang M, Lu XC, Le JJ. Column-oriented join order optimization in column store systems. Journal of Computer Research and

Development, 2013, 50(7): 1473–1483 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.2013.20110927]

[26] Jiao M, Zhang YS, Wang S, Chen H. Research on multicore parallel query processing techniques for main-memory OLAP. Chinese Journal of Computers, 2014, 37(9): 1895–1910 (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2014.01895]

[27] Liu DW, Luan H, Wang S, Tan B. Main memory database TPC-H workload characterization on modern processor. Ruan Jian Xue Bao/Journal of Software, 2008, 19(10): 2573–2584 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/19/2573.htm [doi: 10.3724/SP.J.1001.2008.02573]

[28] Zukowski M, van de Wiel M, Boncz P. Vectorwise: A vectorized analytical DBMS. In: Proc. of the 28th Int'l Conf. on Data Engineering. Arlington: IEEE, 2012. 1349–1350. [doi: 10.1109/ICDE.2012.148]

[29] Roh H, Shin M, Jung W, Park S. Advanced block nested loop join for extending SSD lifetime. IEEE Trans. on Knowledge and Data Engineering, 2017, 29(4): 743–756. [doi: 10.1109/TKDE.2017.2651803]

[30] Mahajan SM, Jadhav VP. Analysis of execution plans in query optimization. In: Proc. of the 2012 Int'l Conf. on Communication, Information & Computing Technology. Mumbai: IEEE, 2012. 1–5. [doi: 10.1109/ICCICT.2012.6398216]

[31] Helmer S, Moerkotte G. Evaluation of main memory join algorithms for joins with set comparison join predicates. In: Proc. of the 23rd Int'l Conf. on Very Large Data Bases. San Francisco: Morgan Kaufmann Publishers Inc., 1997. 386–395.

[32] Shin DK, Meltzer AC. A new join algorithm. ACM SIGMOD Record, 1994, 23(4): 13–20. [doi: 10.1145/190627.190633]

[33] Bornea MA, Vassalos V, Kotidis Y, Deligiannakis A. Double index nested-loop reactive join for result rate optimization. In: Proc. of the 25th IEEE Int'l Conf. on Data Engineering. Shanghai: IEEE, 2009. 481–492. [doi: 10.1109/ICDE.2009.101]

[34] Shahvarani A, Jacobsen HA. Parallel index-based stream join on a multicore CPU. In: Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. 2020. 2523–2537. [doi: 10.1145/3318464.3380576]

[35] Microsoft. SQL graph architecture. 2022. https://docs.microsoft.com/en-us/sql/relational-databases/graphs/sql-graph-architecture?view=sql-server-ver15

[36] Erling O. Virtuoso, a hybrid RDBMS/graph column store. IEEE Data Engineering Bulletin, 2012, 35(1): 3–8.

[37] Shi JX, Yao YY, Chen R, Chen HB, Li FF. Fast and concurrent RDF queries with RDMA-based distributed graph exploration. In: Proc. of the 12th USENIX Conf. on Operating Systems Design and Implementation. Savannah: USENIX Association, 2016. 317–332.

[38] Feng GY, Ma ZX, Li DX, Chen SQ, Zhu XW, Han WT, Chen WG. RisGraph: A real-time streaming system for evolving graphs to support sub-millisecond per-update analysis at millions Ops/s. In: Proc. of the 2021 Int'l Conf. on Management of Data. New York: ACM, 2021. 513–527. [doi: 10.1145/3448016.3457263]

[39] Dziedzic A, Wang JJ, Das S, Ding BL, Narasayya VR, Syamala M. Columnstore and B+ tree—Are hybrid physical designs important? In: Proc. of the 2018 Int'l Conf. on Management of Data. Houston: ACM, 2018. 177–190. [doi: 10.1145/3183713.3190660]

[40] TPC Becnhmark™C standard specification (Revision 5.11). Trans. Processing Performance Council (TPC). 2010. https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf

[41] Cole R, Funke F, Giakoumakis L, Guy W, Kemper A, Krompass S, Kuno H, Nambiar R, Neumann T, Poess M, Sattler KU, Seibold M, Simon E, Waas F. The mixed workload CH-benCHmark. In: Proc. of the 4th Int'l Workshop on Testing Database Systems. Athens: ACM, 2011. 8. [doi: 10.1145/1988842.1988850]

[42] TLX Library. 2018. https://panthema.net/2018/0528-tlx-library/

[43] Pareek A, Khaladkar B, Sen R, Onat B, Nadimpalli V, Lakshminarayanan M. Real-time ETL in Striim. In: Proc. of the 2018 Int'l Workshop on Real-time Business Intelligence and Analytics. Rio de Janeiro: ACM, 2018. 3. [doi: 10.1145/3242153.3242157]

[44] Chen R, Shi JX, Chen YZ, Chen HB. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In: Proc. of the 10th European Conf. on Computer Systems. Bordeaux: ACM, 2015. 1. [doi: 10.1145/2741948.2741970]

[45] Mofidpoor M, Shiri N, Radhakrishnan T. Index-based join operations in Hive. In: Proc. of the 2013 IEEE Int'l Conf. on Big Data. Silicon Valley: IEEE, 2013. 26–33. [doi: 10.1109/BigData.2013.6691768]

[46] Li Y, On ST, Xu JL Choi B, Hu HB. Optimizing nonindexed join processing in flash storage-based systems. IEEE Trans. on Computers, 2013, 62(7): 1417–1431. [doi: 10.1109/TC.2012.86]

[47] Li L, Wu G, Wang GR. In-memory skiplist optimization technologies based on data feature. Ruan Jian Xue Bao/Journal of Software, 2020, 31(3): 663–679 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5902.htm [doi: 10.13328/j.cnki.jos.005902]

[48] Begley S, He Z, Chen YPP. MCJoin: A memory-constrained join for column-store main-memory databases. In: Proc. of the 2012 ACM SIGMOD Int'l Conf. on Management of Data. Scottsdale: ACM, 2012. 121–132. [doi: 10.1145/2213836.2213851]

[49] Arroyuelo D, Hogan A, Navarro G, Reutter JL, Rojas-Ledesma J, Soto A. Worst-case optimal graph joins in almost no space. In: Proc. of the 2021 Int'l Conf. on Management of Data. New York: ACM, 2021. 102–114. [doi: 10.1145/3448016.3457256]

[50] Xue X, Shen SJ, Chen R. Range search method based on indexed backup in distributed data store. Journal of Chinese Computer Systems,

2018, 39(8): 1781–1786 (in Chinese with English abstract). [doi: 10.3969/j.issn.1000-1220.2018.08.026]

[51] Sun W, Fokoue A, Srinivas K, Kementsietsidis A, Hu G, Xie GT. SQLGraph: An efficient relational-based property graph store. In: Proc. of the 2015 ACM SIGMOD Int'l Conf. on Management of Data. Melbourne: ACM, 2015. 1887–1901. [doi: 10.1145/2723372.2723732]

[52] Ma HB, Shao B, Xiao YH, Chen LJ, Wang HX. G-SQL: Fast query processing via graph exploration. Proc. of the VLDB Endowment, 2016, 9(12): 900–911. [doi: 10.14778/2994509.2994510]

附中文参考文献:

[14] 赵展浩, 黄斐然, 王晓黎, 卢卫, 杜小勇. 基于SQL的图相似性查询方法. 软件学报, 2018, 29(3): 689–702. http://www.jos.org.cn/1000-9825/5449.htm [doi: 10.13328/j.cnki.jos.005449]

[15] 信俊昌, 王国仁, 李国徽, 高云君, 张志强. 数据模型及其发展历程. 软件学报, 2019, 30(1): 142–163. http://www.jos.org.cn/1000-9825/5649.htm [doi: 10.13328/j.cnki.jos.005649]

[21] 朱阅岸, 张延松, 周烜, 王珊. 一个基于三元组存储的列式OLAP查询执行引擎. 软件学报, 2014, 25(4): 753–767. http://www.jos.org.cn/1000-9825/4568.htm [doi: 10.13328/j.cnki.jos.004568]

[25] 王梅, 陆戌辰, 乐嘉锦. 列存储系统面向列的连接顺序优化研究. 计算机研究与发展, 2013, 50(7): 1473–1483. [doi: 10.7544/issn1000-1239.2013.20110927]

[26] 焦敏, 张延松, 王珊, 陈红. 内存OLAP多核并行查询优化技术研究. 计算机学报, 2014, 37(9): 1895–1910. [doi: 10.3724/SP.J.1016.2014.01895]

[27] 刘大为, 栾华, 王珊, 覃飙. 内存数据库在TPC-H负载下的处理器性能. 软件学报, 2008, 19(10): 2573–2584. http://www.jos.org.cn/1000-9825/19/2573.htm [doi: 10.3724/SP.J.1001.2008.02573]

[47] 李梁, 吴刚, 王国仁. 面向数据特征的内存跳表优化技术. 软件学报, 2020, 31(3): 663–679. http://www.jos.org.cn/1000-9825/5902.htm [doi: 10.13328/j.cnki.jos.005902]

[50] 薛翔, 沈斯杰, 陈榕. 一种使用索引式备份的范围查询方法. 小型微型计算机系统, 2018, 39(8): 1781–1786. [doi: 10.3969/j.issn.1000-1220.2018.08.026]

沈斯杰(1995－), 男, 博士生, 主要研究领域为并行与分布式系统.

陈海波(1982－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为操作系统, 并行与分布式系统.

陈榕(1981－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为操作系统, 并行与分布式系统.

臧斌宇(1965－), 男, 博士, 教授, 博士生导师, CCF 会士, 主要研究领域为操作系统.