软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.006203]

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

多查询共享技术研究综述*

危剑豪, 夏烨峰, 宫学庆

(华东师范大学软件工程学院,上海 200062)
通讯作者: 宫学庆, E-mail: xqgong@sei.ecnu.edu.cn

摘 要:传统的数据库系统围绕单次查询的模型构建,独立地执行并发查询.由于该模型的限制,传统数据库无法一次对多个查询进行优化.多查询共享技术旨在共享查询之间的公共部分,从而达到提高系统整体响应时间和吞吐量的目的.本文将多查询执行模式分为两类,介绍了各自的原型系统——基于全局查询计划的多查询原型系统和以运算符为中心的多查询原型系统,并且讨论了两种系统的优势以及所适用场景.在之后的内容中,我们将多查询共享技术按照查询的各个阶段分为查询编译阶段中的多查询共享技术以及查询执行阶段中的多查询共享技术两大类.以这两个方向为线索,梳理了多查询计划的表示方法,多查询表达式合并,多查询共享算法,多查询优化等各种方向的研究成果,在此基础上,我们还介绍了共享查询技术在关系数据库和非关系数据库中的应用.最后,分析了共享查询技术面临的机遇和挑战.

关键词:多查询;共享查询;查询优化;数据库;高并发
中图法分类号: TP311

中文引用格式:危剑豪,夏烨峰,宫学庆.多查询共享技术研究综述.软件学报,2020 http://www.jos.org.cn/1000-9825/6203.htm
英文引用格式:Wei JH, Xia YF, Gong XQ.Review of research on multi-query sharing technology. Ruan Jian Xue Bao/Journal of Software, 2020 (in Chinese). http://www.jos.org.cn/1000-9825/6203.htm

Review of Research on Multi-Query Sharing Technology

WEI Jian-Hao, XIA Ye-Feng, GONG Xue-Qing
(School of Software Engineering, East China Normal University, Shanghai 200062, China)

Abstract: Traditional database systems are built around a model of query-at-a-time, and concurrent queries in the context are executed independently. Due to the limitations of this model, traditional databases cannot optimize multiple queries at a time. Multi-query sharing technology is designed to share the common part between queries to improve the overall response time and throughput of the system.This paper divides the multi-query execution mode into two categories and introduces their respective prototype systems:the multi-query prototype system based on the global query plan and on demand simultaneous pipelining. Also, we discussed the advantages of the two systems and the applicable scenarios. In the following content, we divide the multi-query sharing technology into multiple query sharing technologies in the query compilation phase and query execution phase according to the various stages of the query. There are two major types of multi-query sharing technologies. Taking these two directions as clues, the research results in various directions such as the multi-query plan representation method, multi-query expression combination, multi-query sharing algorithm, and multi-query optimization are reviewed here. On the basis, we also introduce the shared query technology in relational database and non-relational database. Application. Finally, it analyzes the opportunities and challenges facing shared query technology.
Key words: multi-query; query sharing; query optimization; database; high concurrency

对于一个数据库管理系统来说, 长期以来都是采取一次执行一个查询(one-query-at-a-time, 以下称"单次查询")[1]的执行模式, 即系统在处理单次查询的时候, 为该查询创建一个线程, 各个查询线程之间互不影

* 基金项目: 国家重点研发计划(2019YFB2102600); 国家自然科学基金(61572194); 国家自然科学基金(61672233)
Foundation item: National Key Research and Development Project (2019YFB2102600), National Natural Science Foundation of China (61572194),National Natural Science Foundation of China (61672233)

收稿时间: 2020-03-27; 修改时间: 2020-06-29, 2020-08-10; 采用时间: 2020-09-11; jos 在线出版时间: 2020-12-01

响.各查询从查询编译、生成查询计划、检查缓存、执行查询计划到返回结果等各个步骤, 均是独立完成. 当多个查询同时执行时, 各个物理计划之间会竞争相应的磁盘 I/O 访问资源和 CPU 计算资源. 这导致尽管现代系统可以有效地优化和执行单个复杂 OLAP[2](On line Transaction Processing)或 OLTP[3](On line Transaction Processing)查询, 但是当多个复杂查询同时运行时, 它们的性能将显著下降[4]. 为了提高并发环境下的性能(其核心是提高资源利用率以减少资源争用), 解决方案通常包含三种: 1)构建缓存池以共享部分数据[5], 这种数据被动的共享模式往往存在缓存命中率低. 2) 利用不同查询之间访问数据重叠的特性构建物化视图[6][7], 物化视图利用了不同查询之间的共性, 但却潜在着大量维护视图的成本的问题. 3) 针对相似的查询缓存查询结果和查询计划, 此类方法适用的场景较窄, 可扩展性不高, 仅适合于查询语句单一, 查询结果变化较小的场景.

由于单查询模式的局限性, 不同查询在执行过程中相互隔离, 使得很多显而易见的优化机会无法实现[4, 8].并发查询往往会存在很多重复的算子, 例如多个查询扫描同一个数据集, 多个查询对同一个中间结果进行排序等等. 这些可以共同执行的机会在单查询模式下只能被冗余执行, 导致系统的吞吐量降低. 基于该瓶颈, 很多学者开始研究一次执行多个查询(multi-query-at-a-time, 以下称为多查询)的模式.

多查询共享技术的发展历史

早在上世纪 80 年代, 就有团队开始研究多查询共享技术. 早期的多查询共享技术主要集中于查询编译中的查询重写阶段[9,10], 即对并发查询的公共子表达式进行检测与合并, 从而减少多个查询之间的冗余计算. 2000 年 Jianjun Chen 提出了一种多查询计划的概念[11], 即在查询编译中的查询优化阶段将并发查询下的多个查询计划合并为一个全局查询计划, 查询引擎只需执行一次该全局计划就能得到多个查询结果. 陆续的研究表明, 最优全局查询计划问题为 NP-hard 问题[12]. 因此之后对于多查询计划的研究主要集中在根据不同的场景设计启发式的全局查询优化方案. 到 2012 年, Giannikis 结合之前的研究, 提出了一种通用的全局查询计划的生成方案[12], 该方案能够适应场景的变化, 确保查询执行的时间复杂度为$O(n^2)$. 该方案在 SharedDB[4]上实现, 能够为上千并发查询生成全局查询计划, 大大地提高了系统的吞吐量. 在查询优化阶段的另一个研究方向是对各运算符共享算法的优化, 该方向的研究贯穿整个多查询共享研究的历史, 主要集中在常用且开销较大的运算符上, 如顺序扫描[12, 13]、连接[14, 15]、分组[16]、排序[17,18]等. 多查询共享技术虽然能显著地提高系统的吞吐量, 但其并非在任何场景下均适用. 首先多个查询在共享某一运算符或中间结果时, 往往会导致单个查询的响应时间受到影响. 其次[19]中的研究结果表明, 即便是在高并发且查询较为相似的所谓“高共享”场景下, 多查询的吞吐量依然受到硬件环境, 查询选择性, 查询的语义等因素的影响.

随着多查询共享研究工作的不断展开, 支持多查询的数据库系统相继问世, 包括 QPipe[20], Datapath[21], CJoin[22], SharedDB[4]等. 这些系统的实现不尽相同, 对于各种并发环境的优化也是各有所长. 随着这些系统的问世, 多查询共享技术的应用领域也逐渐拓宽. 经过 30 多年的发展,多查询共享技术也从最开始应用于数据仓库和连续查询系统[11, 21,22,23, 24,25], 拓展到了图数据库系统[26,27,28,29]、以 GPU 计算的查询系统[30,31]、基于 Map-Reduce 计算框架的批处理系统[32,33,34, 35,36,37]、以及数据流分析查询系统[38,39, 40]等多个领域. 本文根据查询的执行流程将多查询共享技术分为两类, 分别为查询编译阶段的多查询共享以及查询执行阶段的多查询共享, 并对这两类技术的相关文献进行了梳理与评价. 本文还总结了两种经典的支持多查询共享的原型系统, 并且分析了它们的优势与不足. 本篇文章的主要工作和行文安排如下:

第一节介绍了多查询共享的相关概念以及总结了多查询共享技术的研究方向.

第二节介绍了两种支持多查询的原型系统.

第三节梳理了查询编译阶段中的多查询共享技术.

第四节梳理了查询执行阶段中的多查询共享技术.

第五节梳理了多查询共享技术在各个领域的应用.

第六节进行总结和展望.

1 多查询共享描述

1.1 相关概念

通常数据库处理一条查询语句包含**查询编译**和**查询执行**两个阶段. 查询编译阶段又可分为**查询分析**、**查询重写**、**查询优化**三个步骤, 最终生成**物理查询计划**. 查询执行阶段则是执行查询编译生成的物理查询计划. 我们以关系数据库为例, 关系数据库引擎[41]下的单查询执行流程如图 1 所示, 不同的颜色代表不同的线程. 其中解析器、转换器和优化器位于查询编译阶段, 执行引擎位于查询执行阶段. 在整个查询过程中, 每个客户端所提交的查询实例始终作为一个独立的线程在程序中运行, 不同客户端提交的查询之间互不影响.

数据库中的**并发**通常是指当前时刻正在执行的工作单元,比如当前时刻正在执行 10 个查询,则并发量为10.并发是数据库系统中非常重要的属性,是最通用的负载定义.**吞吐量**即单位时间能够处理的查询数量.**响应时间**即查询的处理时间与等待时间之和.并发的高低直接影响了一个数据库系统的吞吐量和平均响应时间. 随着系统并发量的提升, 吞吐量和平均响应时间均逐渐增加而达到瓶颈.传统数据库由于其受限于单次查询的执行模式, 其性能受到并发的严重制约. 而本文所提到多查询的执行模式旨在减少高并发对数据库系统所带来的负面影响.

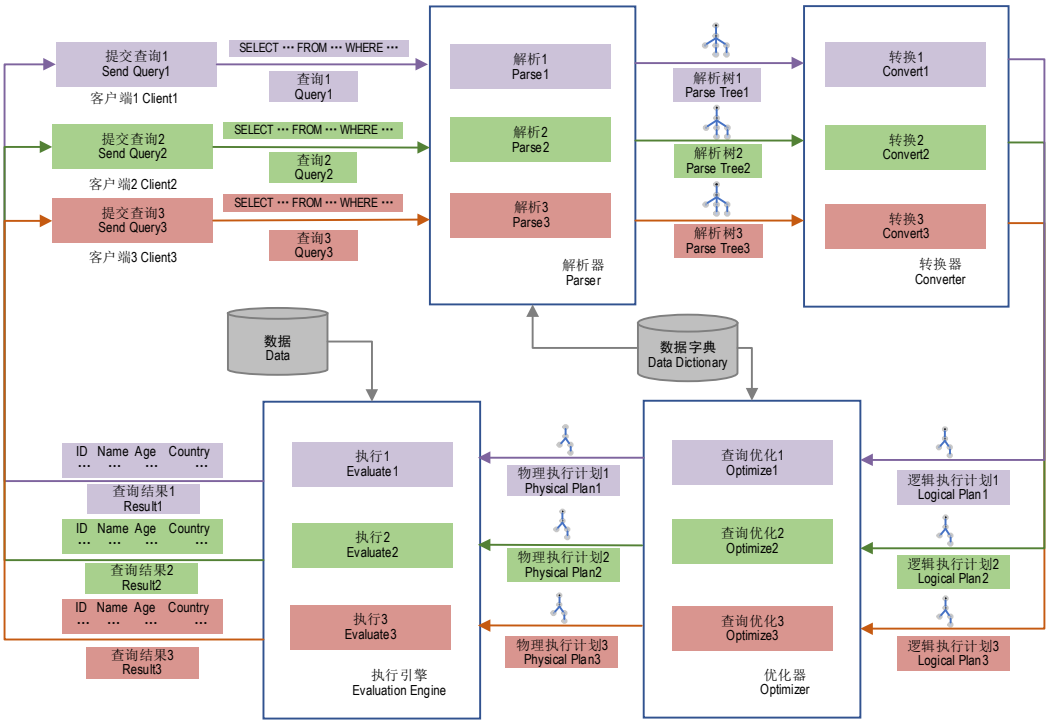

Fig.1 Query execution process of relational database engine

图 1 关系数据库引擎下的查询执行流程

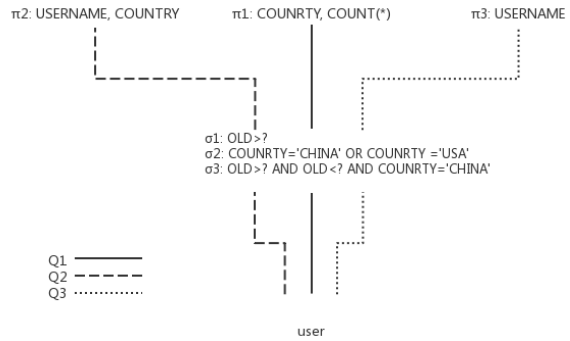

Fig.2 A multiple query plan tree for Q1, Q2, Q3

图 2 Q1,Q2,Q3 的多查询计划树

Table 1 Relation of user, order and product

表 **1** 常见的用户-订单-产品的关系

关系	属性
用户(user)	用户编号(user_id), 名字(username), 年龄(age), 国籍(country)
订单(order)	订单编号(order_id), 产品编号(item_id), 用户编号(user_id), 状态(status), 交易日期(date)
产品(item)	产品编号(item_id), 库存(available), 价格(price), 类别(category)

Table 2 Seven ordinary queries

表 **2** 常见的 7 个查询

查询	语句
Q1	SELECT COUNTRY, COUNT(*) FROM USERS GROUP BY COUNTRY WHERE AGE> ?;
Q2	SELECT USERNAME, COUNTRY FROM USERS WHERE COUNTRY = 'CHINA' OR COUNTRY = 'USA';
Q3	SELECT USERNAME FROM USERS WHERE COUNTRY = 'CHINA' AND AGE>? AND AGE< ?;
Q4	SELECT * FROM USERS U, ORDERS O WHERE U.USER_ID = O.USER_ID AND U.USERNAME = ? AND O.STATUS = 'OK';
Q5	SELECT * FROM USERS U, ORDERS O, ITEMS I WHERE U.USER_ID = O.USER_ID AND O.ITEM_ID = I.ITEM_ID AND I.AVAILABLE < ?;
Q6	SELECT * FROM ORDERS O, ITEMS I WHERE O.ITEM_ID = I.ITEM_ID AND O.DATE > ? ORDER BY I.PRICE;
Q7	SELECT * FROM ITEMS I WHERE I.CATEGORY = ? ORDER BY I.PRICE;

为了后文的讨论, 我们将数据库系统中的**查询语句**称为**查询表达式**, 查询表达式中的一部分被称为**子表达式**, 被多个查询表达式所共有的查询表达式称为**公共查询子表达式**. 查询语句的树结构表达方式称为**查询树,** 又称为查询计划树. 在查询优化阶段, 通过查询搜索算法找出的代价最小的查询计划树称为**最优查询路径**. 将最优查询路径中的节点替换为物理运算符后, 生成的查询计划树即**物理查询计划**.

多查询共享是指多个并发查询在查询执行的各个阶段, 合并相似的子表达式, 共用相同的查询计划, 共

享相同的访问数据或者相同的计算的过程. 通常查询共享, 共享查询, 多查询优化在没有特别说明的情况下都指的是多查询共享. 从计算机程序的角度来说, 多查询共享是指多个查询或者多个查询的子表达式在一个线程执行, 这与“单次查询”有着本质的不同.

接下来描述一个多查询共享实例, 三个常见关系分别是用户(user), 订单(order), 产品(item)如表 1 所示, 查询 Q1, Q2, Q3 如表 2 所示. 这三个查询语句均是访问 USERS 表, 因此存在共享机会. 先在查询编译期间合并三者的公共子表达式, 生成一棵多查询树, 该查询树将三个 SQL 语句的选择谓词合并到了同一个算子中, 如图 2 所示. 之后查询引擎会通过该多查询计划树中的运算符选择合适的多查询算法, 从而生成物理查询计划. 最后查询引擎执行该查询计划, 并将各自的结果返回给客户端. 多查询共享的策略有很多, 且贯穿于查询执行的各个阶段, 本例描述的策略只是其中一种.

1.2 多查询共享研究问题分类

目前所有关于多查询共享的研究都围绕着两种原型系统展开, 即基于全局查询计划的多查询原型系统(Multi-query prototype system based on global query plan, 以下简称 GQP 系统)和以运算符为中心的多查询原型系统(Operator-centric Multi-query prototype system, 以下简称 OMP 系统), 我们将会在下一节对它们进行详细的描述. 基于此两种多查询系统架构, 研究内容主要可以分为: 1) 查询编译阶段的多查询共享技术, 包括多查询计划的表示方法、合并规则、路径搜索以及运算符的共享算法等研究内容; 2) 查询执行阶段中的共享查询技术, 包括缓存管理、执行模块之间的调度; 3)不同类型数据库系统中的多查询共享技术.

1) 两种原型系统
   a) 基于全局查询计划的多查询原型(GQP)系统: GQP 系统应用广泛, 大部分支持多查询共享的系统均采用该系统的设计模式. GQP 系统将来自多个客户端的多个查询解析树进行合并, 之后生成全局查询计划, 最后执行该全局计划并返回结果给相应的客户端. 该系统的共享时机是在生成全局查询计划时产生的.
   b) 以运算符为中心的多查询原型系统(OMP): 针对 GQP 设计上的不足,团队[20]设计了 OMP, 该系统以运算符(算子)为中心,在查询解析与转换时与 GQP 类似,之后生成的解析树通过包分派器(packet dispatcher)将查询拆分为不同的运算符, 并发送给若干个执行器, 每个执行器执行特定的运算符, 生成的结果发送给其他执行器或返回给用户.该系统的共享的时机是在执行器执行相同或相似运算符时产生的.
2) 查询编译阶段的多查询共享技术
   a) 多查询计划的表示方法:该技术主要为 GQP 模式的系统提供全局查询计划. 在 GQP 模式下,当查询语句进入查询分析和重写阶段时, 查询引擎会对多个查询语句解析, 通过词法分析和语义分析生成多查询计划图[9]. 在该阶段主要研究内容是设计有效的数据结构将多查询计划抽象出来.目前主流的多查询计划表示方式是基于图的数据结构.一个良好的多查询抽象方法, 一方面要清楚地表示出查询中共享的算子, 另一方面能够适应查询数量和种类的变化.
   b) 多查询表达式合并: 该研究主要是根据特定的规则, 在不同的场景下,通过签名,哈希等技术快速识别多个查询的公共表达式, 并且将其合并[11]. 如上节所描述, GQP 和 OMP 模式均包含合并模块(Merger), 因此多数多查询表达式合并的研究均适用于以上两种模式.
   c) 多查询共享算法, 该阶段为最优查询计划树中的每个运算符节点选择最佳多查询算法, 将逻辑多查询计划翻译为物理多查询计划. 主要的研究内容即为各个运算符(扫描, 选择, 连接, 排序, 分组等)设计共享查询算法. 扫描和选择运算符逻辑较为简单, 共享算法较为单一. 连接和排序等运算符逻辑较为复杂, 算法种类繁多兼之其具有使用频繁, 开销较大等特点, 因此更多的文献集中于研究这些运算符的共享算法. 由于 GQP 和 OMP 两种模式的算子存在差异, 因此一种算法往往只能适应一种执行模式.
   d) 多查询优化: 与单查询优化相似, 多查询优化的目的是为了产生多查询计划. 早期的多查询共享系统通常在查询重写之后, 跳过最佳查询路径搜索阶段, 直接将逻辑查询计划转化为物理查询计划, 这导致大量运算符和中间结果无法共享. 2000 年之后, 基于成本的启发式多查询优化在一些场景下的成功[11, 42], 吸引了更多学者从事关于多查询路径搜索的研究.良好的多查询计划方案通常包括以下几点, 一是

多查询计划的代价模型, 该模型必须准确地估计出多查询计划的开销并且具有较强的健壮性; 其二是多查询计划的生成算法, 该算法利用上述代价模型, 快速选出一个多查询计划, 同时保证查询计划的高效性. 近年来多查询共享技术被广泛应用到图查询引擎, MapReduce 执行引擎等各领域, 几乎所有的文章中都会提到多查询计划的优化方案. 目前大多数的多查询优化方案都是基于 GQP 模式, 而针对 OMP 的查询优化研究较少, 主要是执行模式所限: 由于 GQP 模式存在优化器(Optimizer)模块, 因此先关研究可以很好地在该模块中进行, 而 OMP 在查询合并之后, 就立即进入执行阶段, 因此该模式不存在传统意义上的查询优化阶段.

3) 查询执行阶段的多查询共享优化

a) 多查询缓存管理: 在查询执行的过程中,由于多查询的运算符一次处理的元组比单查询要多出许多, 这就需要消耗更多的内存. 也有不少文献研究多查询的缓存管理技术, 旨在提高查询中间结果的利用率.

b) 多查询执行模块之间的调度: 该研究方向大多基于OMP 系统. 该系统的执行模块由若干个运算符执行器构成, 数据在各执行器之间相互传递, 这就需要良好的调度机制使各执行器之间能够合理地分配资源, 以减少阻塞, 保证程序的正常运行. 因此也有相关文章对该方向进行探讨.

4) 多查询共享技术的应用

目前多查询共享技术应用较多领域有数据仓库, 连续查询, MapReduce[43]计算框架等. 该研究方向大致是将自身查询引擎, 数据存储, 以及查询的特点与共享查询技术结合起来, 进行特殊的优化并设计一套基于本系统的多查询共享方案. 此类研究往往与某个领域的兴衰息息相关, 例如MapReduce 计算框架的流行也激励着许多学者为MapReduce 设计多任务共享方案; 半结构化存储引擎的研究渐渐衰落也导致多查询共享技术在该领域应用的研究越来越少.

2 支持多查询的原型系统

在关系数据模型之下, 几乎所有的多查询共享技术都围绕着以下两种原型系统进行设计. 在本节中, 我们将详细介绍这两种原型系统的架构以及它们各自的优缺点.

2.1 基于全局查询计划的多查询原型系统

基于全局查询计划的多查询原型系统(Multi-query prototypesystem based on global query plan, 以下简称GQP系统)目前较为主流,大部分支持多查询共享的系统均采用该原型系统的设计模式,例如早期的CScan[23]到最近的SharedDB[4]和DataPath[21].

GQP 的核心思想如下, 首先将多个并发查询根据表达式合并规则进行合并, 生成多查询解析树, 之后通过多查询优化生成全局查询计划, 最后执行该全局查询计划并将结果返回给相应的客户端. 该系统的查询执行流程如图 3 所示, 不同的颜色代表不同的线程.

在该系统中, 多个查询语句首先在查询分析和重写阶段进行公共子表达式的合并, 生成一个全局查询计划图, 在查询路径选择过程生成一个最优或次优全局查询计划图. 查询执行器执行该计划中的每个运算符.生成的结果根据元组标记算法返回给各客户端. 我们为表 2 中的查询构建一个全局查询计划如图 4 所示. Q1, Q2, Q3 共享对用户表的选择运算, Q5 先与 Q6 共享对订单表和项目表的连接运算, 然后再与 Q4 共享对用户表和订单表的连接运算. Q6 在于 Q5 共享之后, 再与 Q7 共享排序运算.

GQP 的优点在于查询执行流程较为简单, 易于扩展, 使其能够很好的兼容各种多查询共享技术, 因此大部分多查询共享技术均是基于该模式. 适用场景一般为高并发下的 OLAP 查询, 对于运算符的相似度、类型则没有具体的限制. GQP 的缺点在于 1) 查询往往需要等待, 单个查询延迟高; 2) 生成全局查询计划开销较大.

经过多年的研究, GQP 执行模式也在不断的发展和完善, 早期 GQP 手动生成全局查询计划, 当查询提交进来时, 直接执行, 计算后的结果通过标记的 queryId 返回, 因此适用的场景非常有限. 随着多查询共享技术的发展, 之后基于 GQP 的系统大多实现多查询路径搜索算法, 使得多个查询可以动态地生成全局查询计

划. 目前主要的优化方向为更多利用直接的共享机会, 尽量减少查询的等待时间以及多查询计划的搜索时间. 类似的系统[21]在内存中维护着一个全局查询计划, 当有 query 提交时, 无需等待, 直接将该 query 拆分为若干个算子, 将该算子与查询计划中存在的共享算子合并, 若系统中不存在该算子的共享算子, 则系统将会为该算子创建新的计划树节点.

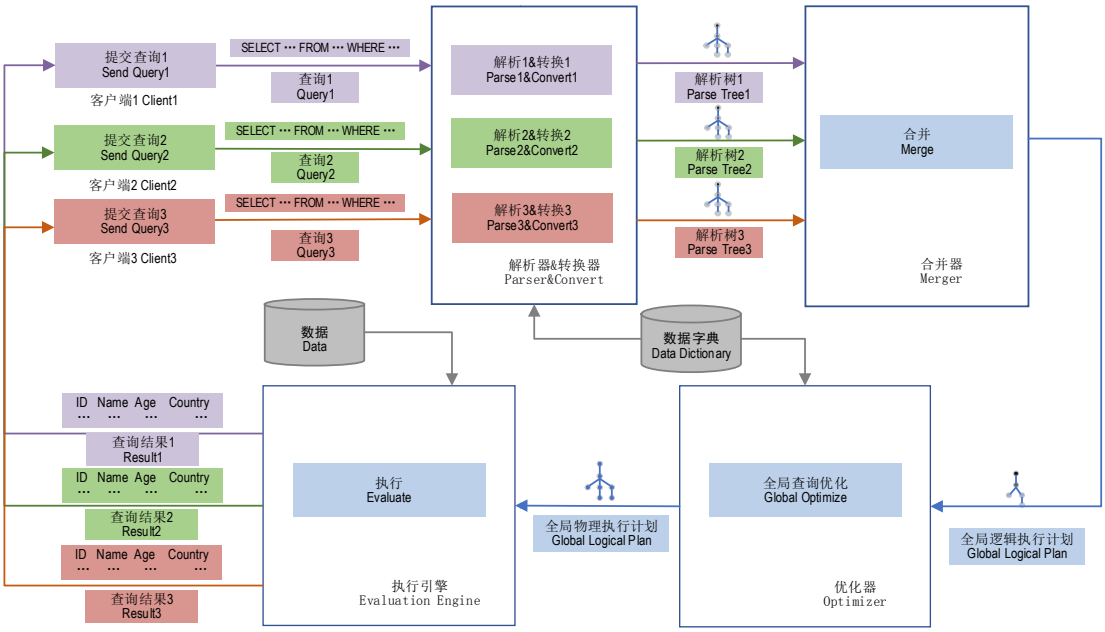

Fig.3 Query execution process based on GQP system

图 3 基于 GQP 系统的查询执行流程

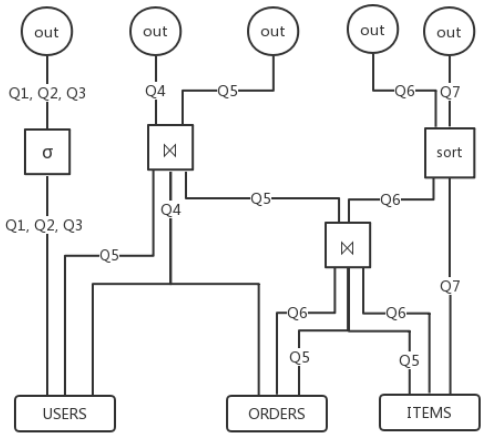

Fig.4 Global query plan tree

图 4 全局查询计划树

2.2 以运算符为中心的多查询原型系统

由于上述查询模式存在着不足, 人们开始寻求一个共享机会更多且更易捕捉, 同时能够避免复杂的多查询计划路径搜索带来的额外开销的多查询执行架构. 2005 年 Stavros 的论文[20]奠定了以运算符为中心的多查询原型系统(Operator-centric Multi-query prototype system,以下简称 OMP 系统)的基础.

对于 OMP 系统来说, 如果两个或多个并发查询在其查询计划中包含相同的关系运算符, 并且该运算符对于所有的查询来说都输出相同的元组(或元组的投影), 则称这些运算符为"可共享运算符". 当查询运行时, 系统动态的利用这些共享运算符, 每一个共享运算符将执行一次, 并且其输出的数据将同时通过流水线管道传输到所有父算子节点. 该系统的查询执行流程如图 5 所示, 不同的颜色代表不同的线程.

在该系统中, 每个运算符都被提升为一个独立的微型引擎(μEngine). 系统输入的是一个个预编译的单查询计划. 查询计划通过 packet 分派器(该分派器会创建与查询树中的节点一样多的 packet)分配给相应的 μEngines, 每个 μEngine 都有一个 packet 请求队列. 属于该 μEngine 的辅助线程从队列中取出 packet 并执行它. packet 里包含输入和输出元组缓冲区位置以及关系运算符的参数(例如, 排序属性, 谓词等). 所有 μEngine 并行处理以评估查询. 评估模型类似于基于 push 式的执行设计[8], 其中每个运算符独立生成元组, 直到填满父级缓冲区为止. 例如, 排序 μEngine 仅接受对元组进行排序的请求. 请求本身必须指定需要排序的内容以及需要将结果放入哪个元组缓冲区. 系统通过将一个 μEngine 的输出链接到另一个 μEngine 的输入来完成查询, 从而在参与的 μEngine 之间建立生产者－消费者关系.

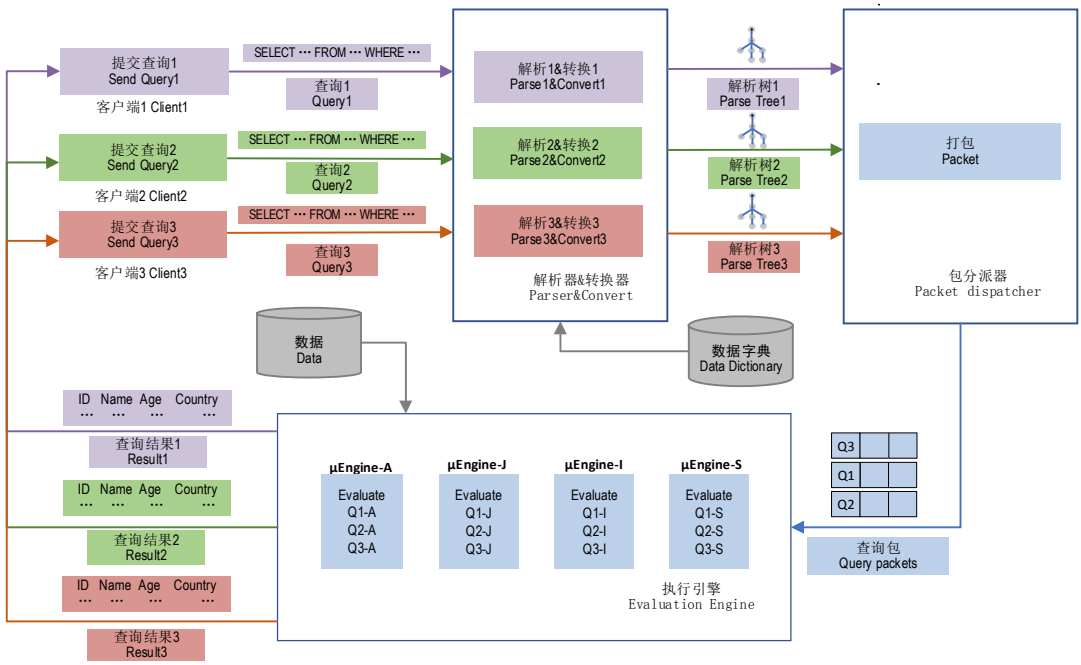

Fig.5 Query execution process based on OMP system

图 5 基于 OMP 系统的查询执行流程

查询 packet 表示查询在给定的 μEngine 上需要执行的工作. 每当新的 packet 在 μEngine 中排队时, 系统都会扫描队列中的 packet, 以检查是否有重叠的工作, 即对每个 packet 编码参数列表(在查询通过 packet 分派器时产生的)的快速检查. 找到重叠的 packet 后, 检查当前算子的哪个阶段可以重用. 每个 μEngine 都采用不同的共享机制, 具体取决于算子的共享时机. μEngine 包含 OMP 协调器和死锁检测器. OMP 协调器负责将新的 packet 与正在执行或未执行的 packet 进行合并, 并将输出的元组同时通过管道传递给所有参与的查询. OMP 协调器还可对原始查询的评估策略做一些必要的调整, 如创建一个附加数据包来完成算子非重叠部分. 死锁检测器则是确保同时执行流水线的调度时实现无死锁.

OMP 的共享机会主要有以下四种:

(1) **线性重叠**, 表示可以始终能够利用正在进行的相同操作的未完成部分, 其成本节省从 0 到 100%不等, 具体取决于 Q2 加入 Q1 的时间, 对应的操作有文件扫描, 索引扫描

(2) **步骤重叠**, 只要尚未生成第一个输出元组, 则可以完全相互利用的并发操作(节省 100%的成本), 对应的操作有嵌套循环连接, 排序合并连接的合并阶段, hash 连接的探测阶段

(3) **完全重叠**, 正在进行中的操作的整个生命周期内始终可以节省 100%的成本. 相应的扫描有排序, 聚合, 排序合并 join 的排序阶段, hash 连接的构建阶段

(4) **尖峰重叠**, 只有在同一时间开始才能重叠的操作. 相应的操作有文件和索引的顺序扫描,

OMP 执行模式的好处在于它无需指定复杂的多查询计划, 共享机会较多,可以充分利用查询之间重叠的计算及数据, 查询无需等待, 到达的查询可以立即执行. 适用的场景主要具有如下特征: 高并发环境下, 查询选择性相似且存在大量聚合排序(共享机会为完全重叠)的查询.

OMP 执行模式的缺点也是很明显的, 1) 一个 μEngine 执行完成后, 会按照查询的数量将产生的结果发给父算子所在的 μEngine, 这无疑会产生大量的数据冗余. 在极端情况下, 数据的冗余是呈指数增长的. 因此该模型对内存的开销极大; 2) 数据需要在各个 μEngine 之间传递, 会产生一定的延迟; 3) 由于某个 μEngine 接收数据的时间是不确定的, 有可能导致各 μEngine 之间负载不均衡.

对该执行模式主要的优化包括 1) 在查询编译时对表达式进行预处理, 合并公共子表达式以减少 μEngine 的负担, 2) 设计协调策略使 μEngine 的负载更加均衡.

2.3 小结

这两种原型系统的优点和缺点都很明显, 这也从一方面说明了多查询共享并不适用于所有高并发场景. 2007 年 Ryan Johnson 发现在硬件条件为 32 核 CPU, 使用 QPipe[20]作为查询引擎, 运行共享机会较多的工作负载时, 并没有得到很好的吞吐量结果. 进一步实验表明, 导致共享效率降低的原因有很多, 如共享的时机, 查询的种类, 硬件条件, 查询选择性, 负载的并行性等等. 因此, 目前大多数共享查询系统仅仅针对某些高并发场景进行优化.

3 查询编译阶段中的多查询共享技术

多查询优化器的工作是: 1) 寻找共享计算的可能性, 对每个运算符采用最优共享算法; 2) 利用优化器搜索策略找到全局最优计划. 以上两项任务都是至关重要的, 但它们是正交的. 换句话说, 搜索策略的细节并不取决于运算符共享算法的有效程度. 本节梳理了近 30 年来主流的多查询共享优化技术.

3.1 多查询计划的表示方法

1982 年 Sheldon Finkelstein 在其文[9]中将查询用一种无向图来表示, 名为查询图(query graph). 查询图实例如图 6 所示, 与查询树不同的是, 该查询图的节点中存储了除连接关系以外的所有信息, 包括所属关系, 谓词, 投影等信息, 而连接关系则用边来表示. 这种用图来表示查询的方法有利于查询之间的合并, 文中的共享策略为将多个相似的查询图合并为临时查询图, 并为每个查询图产生一个快照. 当不可预知的连续查询对数据库进行访问时, 可以先访问查询图, 从而共享查询图所对应快照中的数据.

随着研究的深入, 图数据结构渐渐被用来表示多查询计划, 查询引擎将多个公共运算符节点合并, 从而生成一个包含多个查询的图结构. 2000 年 Prasan 在其文中[42]将查询用一种有向无环图来表示, 名为 and-or DAG, 该 DAG 的起始节点表示关系, 其他节点则表示运算符(包括扫描, 选择, 连接, 排序等). 而 2003 年 Nilesh 文中[17]提出的 plan-DAG(如图 7 所示)则是对 and-or DAG 的改进, plan-DAG 的实线边表示数据以流水线的方式从一个节点(运算符)传递到另一个节点(运算符), 虚线边则表示数据以固化的方式从一个节点(运算符)传递到另一个节点(运算符). 用 DAG 表示多查询计划的优势在于执行引擎可以并行地从多个根节点开始执行该查询计划. 在最近的研究中,shareDB 与 Datapath, AStream 等系统也使用 DAG 来表示全局查询计划. 与 plan-DAG 不同的是, sharedDB 等系统处理数据的方式为 push, 即自底向上将元组推入查询计划中, 而由于 plan-DAG 中数据的传递方式包含固化和流水线两种, 因此能够采用 pull 的数据处理方式.

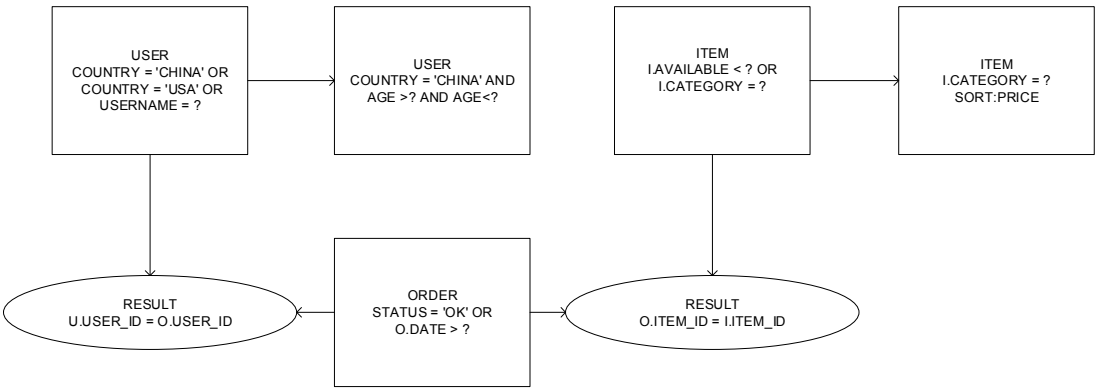

Fig.6 An example of query graph

图 6 查询图实例

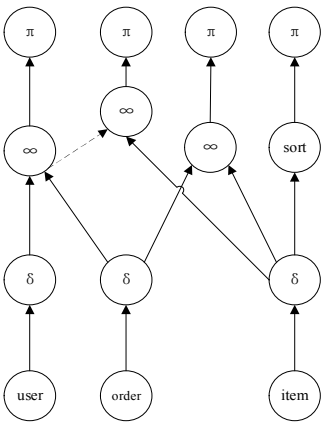

Fig.7 An example of plan DAG

图 7 Plan DAG 实例

3.2 多查询表达式合并

2000 年 Jianjun Chen 提出通过签名(signature)将多个查询表达式进行合并的技术. 在其文中[11]提出了多种签名策略. 首先, 用占位符替换谓词中出现的常量, 为选择谓词创建表达式签名. 该签名是一种树形结构(如图 8 所示), 父节点存储运算符, 左叶子节点存储公共表表达式, 右叶子结点存储变化的常量. 在此基础上, 提出签名组概念(所谓签名组就是所有查询的通用表达式签名(如表 3 所示)),从而使得在合并表达式时能够记录常量所属的查询.该签名组为二维表结构, 第一列表示常量值, 第二列表示查询 id.

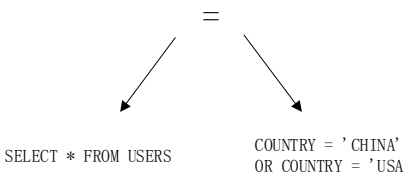

Fig.8 The signature of Q2

图 8 Q2 的签名

Table 3 Signature groups of Q1, Q2, Q3
表 **3 Q1, Q2, Q3** 的签名组

常量	查询 id
AGE> ?	Q1
COUNTRY = 'CHINA' OR COUNTRY = 'USA	Q2
COUNTRY = 'CHINA' AND AGE>? AND AGE< ?	Q3

2007 年 Jingren Zhou 提出了一种表签名技术[44], 用来快速标记多个查询的公共部分, 该表签名可以表示为$S_e = [G_e; T_e]$(其中 e 为选择、投影、连接、分组等表达式, S_e为表签名, G_e是一个布尔值, 表明 e 是否包含 groupby 运算, T_e是表达式 e 中表的集合). 例如 Q1, Q2, Q3 的表签名分别表示为S_{Q1}=[T;{User}], S_{Q2}=[F;{User}], S_{Q3}=[F;{User}]. 表签名充当表达式的高级摘要, 用作检测潜在可共享表达式的快速筛选器.

2012 年 Yasin N. Silva 也提出了类似识别公共子表达式的技术[25], 称之为 FINGERPRINTS(FP). 简单地来说, FP 是一种查询表达式的摘要, 首先作者将每个查询转化成查询树, 然后将树中的操作节点与相应的操作 id 对应, 再将树中的表节点与文件 id 对应, 通过数学计算得到相应的 FP. 具有相同 FP 的两个表达式很可能相等, 具有不同 FP 的两个表达式不相等. 源自根节点 R 的表达式 E 的 FP 表示为F_E,其计算方式如下:

(1) 如果 R 代表直接从数据文件读取的操作,则

$$F_E = R.FIleID \ mod \ N$$

(2) 否则,

$$F_E = \left(R.OpID \oplus F_{R.child[i]}\right) mod \ N \quad (0<=i<=k)$$

在该定义中, ⊕是 XOR 操作; N 是足够大的质数, 可以防止 FileID 和 OpID 的值之间发生冲突; FileID 是数据文件的唯一标识符; OpID 是操作的唯一标识符, 例如, 所有分组操作都具有相同的 OpID.

2018 年 Alekh Jindal 对公共子表达式的选择问题进行了基于整数线性规划(integer linear programming)的定义,并将该问题抽象为二分图标记(bipartite graph labeling)问题[45].文中[46]将逻辑查询计划抽象为$G = (V_Q, V_S, E)$的二分图,Q 表示查询,S 表示子表达式,每个顶点$v_{q_i} \in V_Q$,代表一个查询$q_i \in Q$;并且每个顶点$v_{s_i} \in V_S$,代表一个子表达式$s_j \in S$;查询顶点与表达式顶点相互连接,代表查询与表达式之间的关系.基于以上定义,文中阐述了一种并行的公共子表达式选择算法 BIGSUBS.该算法本质是一种迭代方法,每个迭代包括两个步骤：首先为每个查询的子表达式分配标签, 以最大化每个查询的效用,同时注意子表达式的交互作用.在第二次迭代中,确定在顶点标注步骤中选择的子表达式, 然后遵循等式中的优化目标, 通过标记与该查询相邻的边来确定各查询中的公共子表达式.

2018 年 Albert Jonathan 在其文[40]中采用有向无环图(DAG)来表示单个查询的逻辑查询计划,以查询顶点(运算符)为中心, 以拓扑顺序遍历顶点,从源顶点拓扑上将一个新查询与每个现有查询进行比较,如果顶点相等,则继续遍历,值到找出所有可共享的节点,并将其合并.如果两个顶点不相等, 则通过定义, 它们的下游顶点均不相等, 因此不需要对其进行比较.该算法可以高效地将非共享的查询区分出来,从而更快地定位相似性高的查询(但相似性高并不意味着查询能够共享,需要进一步分析).该算法在子图匹配算法[47]的基础上进行了改进,并将其运用到共享查询的匹配上,使得原本只能匹配单个查询的算法可以同时匹配多个查询.

3.3 多查询共享算法

运算符的共享计算也是多查询共享技术研究的另一大方向, 主要的研究内容即针对各个运算符设计共享算法, 如选择, 连接, 排序以及分组等. 选择运算符的多查询共享算法主要分为两种: 1) 是在查询编译阶段, 将多个查询的谓词表达式合并, 在扫描一条元组的同时, 进行多个查询谓词的评估, 最后根据查询识别算法[4]将中间结果或最终结果返回; 2) 直接在查询执行阶段, 将多个可共享的选择运算符放入一个队列中, 对每一个元组进行流水线评估, 结果直接返回给查询下一阶段. 两种策略的性能相仿, 第一种策略可以对查询表达式进行更多的优化, 从而减少冗余计算, 但需要依赖查询识别算法分派结果. 第二种策略存在更多的

冗余计划, 但可以直接处理查询结果. 更多共享算法的研究集中于排序、分组和连接运算符, 这些运算符使用频繁, 而且开销较大, 有很高的共享价值. 同时由于单查询模式下, 这些运算符大多具有几种以上的处理算法, 从而衍生出了相应的多查询处理算法. 本节将列出一些有代表性的多查询共享算法的研究.

2004 年 Sailesh Krishnamurthy 等人在其文中[13]指出, 简单的将查询谓词合并, 会导致查询执行时产生大量冗余元组及不必要的计算, 对此, 他们提出了一种叫做 TULIP(Tuple Lineage in Plans)的精确共享查询方案, 该方案能最大程度地减少查询执行带来的冗余元组. 精确共享需要满足以下两个条件 a: 对于每个处理的元组或者元组的副本, 任何给定的算子最多可以应用一次. b 任何算子都不得产生“僵尸”元组; 也就是说, 一个元组, 在数据流中是否存在对任何查询的结果都没有影响. 不满足 a 的计划会遭受冗余开销. 不满足 b 的算子会导致浪费的生产并伴随消除僵尸元组的工作. TULIP 方案首先为每一个中间结果元组添加一个叫做元组谱系的向量结构, 该向量由两部分组成: (1) 引导向量, 用于描述数据流中已访问和将要访问的运算符. (2)完成向量, 用于描述系统中对该元组“无效”的查询, 即该元组无法满足的查询. 其次, 该方案还在内存中维护着当前系统所注册的谓词和连接的索引, 当它收到一个新的元组时, 它会有效地探查索引以识别失败的所有已注册子句. 然后, 将所有这些失败记录在元组的沿袭矢量(lineage vector)中. 如果在处理元组结束时, 有任何对元组的实时查询(即仍然可以满足的查询), 则仍然元组将输出. 实验结果显示, 使用 TULIP 方案的系统在查询重叠较大的负载之下, 平均响应时间要比没有使用的系统少 20%到 30%.

2007 年 Marcin Zukowski 等人在其文中提出了运算符 CScan[12], 是扫描运算符的一种协作变体. Cscan 通过活动缓冲区管理器(ABM)追踪每个 Cscan 算子, 找到算子之间公共部分, 并尝试安排磁盘进行读取, 以便使多个并发扫描重用同一数据块. 此外, 为了在高并发查询场景下获得良好的带宽, 扫描处理通常使用称为块的大型的 io 单元. 相比于传统扫描, 协作扫描框架实现了一种叫做相关性扫描的策略, 该策略通过使用一组相关性函数来进行调度决策. 当每次需要处理大量数据时, 系统会调用 choiceAvailableChunk 函数, 该函数访问搜索仍然需要由查询处理的所有缓存块, 同时 useRelevance 会“推荐”出尽可能被较少查询访问的块, 这样一来, 不太活跃的块将会尽快被消耗, 从而加入新的活跃的块. 协作扫描框架还维护了一个查询优先级队列, 较短的查询会有的较高的优先级, 这样就能保证短查询的响应时间, 但是系统也会优先执行将等待时间较长的长查询. 实验结果表明, 协作扫描框架在上亿数据集的 OLAP 工作负载之下, 吞吐量和整体响应时间都要优于使用传统扫描框架的查询引擎. 不足之处在于, 协作扫描不能很好地保证单查询的响应时间, 由于块长度过大, 使得该框架必须运行在高内存设备上.

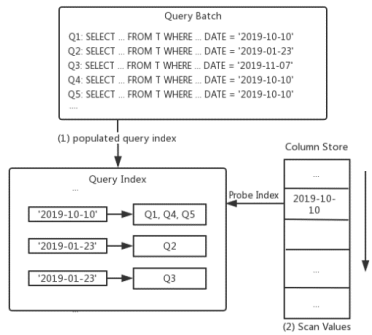

Fig.9 An index of multiple query

图 9 多查询索引

2013 年 G. Alonso 在其文中[14]提出了一套在列存数据库上共享扫描的算法, 该算法如下, 首先遍历列的集合, 对于每一个列, 为该列的每一个值标记它的位置,以及它所对应的查询 id, 从而为每列生成一个位置数组, 最后, 各查询扫描后的结果集根据位置数组生成. 该算法采用两种方法来避免冗余计算 a) 对选择性高的谓词尽量优先进行评估, 从而减少中间结果的数量; b) 为查询谓词添加索引, 索引的 key 为高选择性谓

词的值, value 则是该谓词所对应的查询 id 集合. 整个索引的生成过程如图 9 所示, 该想法早在[48]中提出, 但适用性较差(查询开销必须普遍高于构建查询索引的开销), 因此之后的研究鲜有提及. 作者将该方案应用到列存数据库引擎中, 能够提高探测索引的效率, 但并构建索引的代价较大, 尤其是在高并发场景下, 而作者并未对此作出优化.

2016 年 Duy-Hung 在其文中[49]提出了一种针对聚合查询的共享查询算法, 名为“自顶向下拆分”(Top-Down Splitting)算法, 该算法可扩展到数百甚至数千个查询, 并且可以快速有效地生成优化的查询执行计划. 首先,作者假定表 T 有 m 列属性.设$S = \{s_1, s_2, \dots, s_n\}$为 n 个查询的分组(grouping)集合, s_i表示表 T 的一个或多个属性.Q_i代表一个分组查询:

$$Q_i: Select\ s_i, Count(*)\ From\ T\ Group\ By s_i$$

而$Att = \{a_1, a_2, \dots, a_m\} = \mathrm{U}_{i=1}^{\infty} s_i$为 S 中包含的所有属性的集合.作者构建了一个名为搜索 DAG 的有向无环图$G = (V, E)$来表示该多查询实例.V 是集合S再加上根节点 T 的集合,根节点(rootnode)为潜在地数据输入节点, 其他节点则称为终端节点(terminalnode).从节点 u 到节点 v 的边$e = (u, v) \in E$表示分组查询的执行路径.该分组多查询图的实例如图 10 所示:

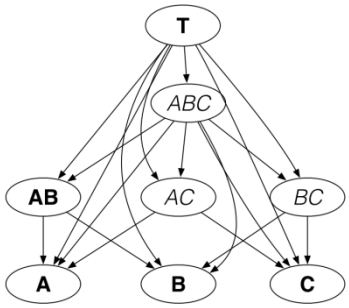

Fig. 10: An example of a search DAG

图 10 搜索 DAG 实例

该算法的基本思想如下: 第一步是构建仅由终端节点和根节点组成的初步方案树. 文中以终端节点的基数降序对终端节点进行排序,在遍历终端节点的过程中,将它们添加到初步解决方案树G'中.添加所有终端节点后，系统将更新节点 u 及其子节点中扫描/排序之间的关系.本质上,该过程可以找到 u 的子节点，并对相同的扫描和排序算子进行预合并处理.第二步以该初步方案树为输入, 从根节点自顶向下地递归寻找子节点的最优化方案,然后生成全局逻辑查询计划,最后生成全局物理查询计划并执行.作者的实验结果表明, 与未优化的计划相比, 作者的算法最多可将查询执行时间减少 34%.

2017 年 Jizhou Sun 在其文章[16]提出了一种分批分组和共享中间结果的技术, 类似思路也在[50, 51, 52]中提及. 该技术利用分组查询中计算的重叠, 从而达到中间结果的共享, 以减少大数据做分组操作, I/O 开销过大的问题. 设 R(A, B, C, D), 现对 A 和 D 属性排序或分组, 当属性 D 有序时, 属性 A 部分有序, 这时若利用该结果对 A 属性进行排序或者分组, 则可以减少相当一部分计算. 而在批处理分组设置中, 不同的分组顺序可能出现更多的计算重叠. 根据分组键之间的不同关系, 作者提出了三种不同的共享级别, 分别是：完全共享, 部分共享和常规共享. 为了最小化 I/O 成本, 作者将组顺序调度问题形式化为一个可以在 NP-hard 中证明的优化问题, 然后提出一种启发式算法--- HEGOS(启发式分组调度)算法. 该 HEGOS 算法核心思想是, 由于完全共享减少了后续任务的整个工作量, 因此作者尽可能地充分利用完全共享, 然后以贪心地方法逐一安排其余类型的共享. 作者并未在真实的数据库系统下进行实验, 而是编写程序模拟了数据库分组计算的过程. 因此文中所提出的算法是否能适用于真实的系统环境, 还有待商榷.

2018 年, Darko Makreshanski 等人在其文中[15]提出了一种针对哈希连接运算符的共享算法, 名为 MQJoin. 该算法的思路大致如下, 首先将关于左表的 where 子语句筛选出来设为组 A, 再将剩下的右表的

where 子语句设为组 B. 分别对左右两表进行扫描, 过滤出满足条件 A 和 B 的左右表元组集合. 然后对两组元组集合进行哈希连接. 该算法为每个中间结果添加长度为查询数量的位图向量, 用来识别每个元组所对应的查询. 该算法的缺点在于每个中间结果均包含一个冗余的位图向量, 该向量大小与查询数量成正比, 性能往往受 CPU 高速缓存大小的限制. 为了提升算法的连接效率, 作者采用 12 核 AMD Opteron 6174 'Magny-Cours'处理器, 使得带有位图向量的元组可以一次性读入 CPU 缓存中以增加连接探测的效率. 作者在 Crescando 上实现了 MQJoin, 对于基于 TPC-H[53]的工作负载, 该系统的吞吐量要比 Vectorwise[54]高出 2-5 倍, 并且相应时间更稳定.

3.4 多查询优化

1988 年 Timos[55]提出识别正在运行的查询集中的常见子表达式, 生成有效的多查询执行计划. 他的关注点在于构建最佳的访问路径[56], 文中提出了两种多查询访问路径的优化方案. 一种是基于计划合并的方法, 主要思路是为每个查询生成最佳访问计划, 并将这些计划合并为一个多查询访问路径. 但这种方式并不适用于多连接的 OLAP 负载, 因为当单个查询确定了连接的顺序之后, 查询之间因为连接结果的不一致而导致无法共享. 第二种更复杂的方法是基于全局优化器, 在该方案中, 作者将优化问题表述为状态空间搜索问题, 并针对该问题提出了一种算法, 但该算法的时间复杂度较高, 导致生成多查询计划的时间远远超过单查询计划. 类似的策略还包括[54, 57] , 这些优化策略专注于为少量查询找到最佳的全局计划, 通常适用于查询事先给出或少量类似的查询会在很短的时间间隔内进入系统的场景. 但每当添加新查询时, 系统往往需要重新优化所有的查询, 这种方法对于大型动态环境显然是不可接受的.

2000 年 Jianjun Chen 在其文中提出了一种启发式的多查询优化策略[11]. 该策略的核心思想如下, 首先根据现有查询的签名为它们创建组, 这些签名代表查询之间的相似结构. 每个组允许共享两个或多个查询的公共部分. 查询组中的每个单独查询共享执行组计划的结果. 提交新查询时, 组优化器会将现有组视为潜在的优化选择. 新查询将合并到与其签名匹配的现有组中, 因此现有查询并不会重新分组. 该策略可能无法产生最优多查询计划, 但它显著降低了优化的成本, 更重要的是, 它在动态环境中具有很高的可伸缩性. 由于查询组中经常添加和删除连续查询, 可能使当前组执行的效率变得非常低下. 在之后的研究中, 该团队又提出了"动态重新分组"技术, 该技术定期或在系统性能下降到某个阈值以下时重新组织部分或全部查询.

2000 年 Prasan 在其文章中首次证明了启发式的多查询优化是切实可行的, 并且具有明显的优势. 文中提出了三种基于成本的启发式算法：Volcano-SH 和 Volcano-RU, 以及贪心启发式算法. 三种均是算法基于 AND-OR DAG[58, 59], 并且算法的思路大致也相同, 而启发式规则略有差异. 算法大致思路如下, 通过启发式规则将公共子表达式合并, 生成多个 AND-OR DAG. 然后对每一个 DAG 进行深度优先遍历, 找出等价操作节点 e, 然后设 cost(e)为节点 e 的计算开销, numuses(e)为 e 操作重叠的次数, matcost(e)为节点 e 物化的开销, reusecost(e)为重用物化后 e 的结果开销. 如果满足 cost (e)+matcost(e)+reusecost(e)*(numuses(e)-1) < numuses(e) * cost(e) 则对该节点的结果进行物化, 从而与其他相同操作的节点共享. 当对 DAG 遍历完成后, 计算出所有 DAG 的开销, 找出开销最低的 DAG. 作者将传统 Vocalno 与 Volcano-SH, Volcano-RU, 以及贪心启发式算法进行对比试验, 随着数据量的增加, 三种算法的成本与收益比明显增加. 但是, 当处理非共享的工作负载时, 三种算法的性能就不及传统 Vocalno 算法, 贪心启发式算法的性能下降最多, 高达 25%.

2003 年 Nilesh 在其文中证明了找到最低成本的多查询计划是 NP-hard 问题[17], 并提出了一种启发式的贪心算法用以搜索最优查询计划(多查询计划以由 plan-DAG 构建). 作者首先对谓词相同的节点合并, 以生成全局的多查询计划图, 该图的节点之间以有向边连接, 有向边代表数据从一个节点(运算符)传递到另一个节点(运算符), 传递方式包括流水线和固化两种. 算法的思想就是将读取成本最高的边进行固化, 以共享公共元组, 剩下的边进行流水线化,直到无法将边进行流水线化为止, 从而达到局部成本最优的解.

2007 年 Jingren Zhou 设计了一个实用, 统一且具有扩展性的探测和利用相似子表达式的方案[44], 子表达式分为选择(select), 投影(projection), 连接(join), 分组(group)四种类型(SPJG). 该方案首选通过对关系表签名(table signature)机制快速找到可能共享的 SPJG 子表达式组, 所谓表签名即通过将子查询中的 SPJG 信息

结构化并存入二元组中. 在探测到相似子表达式集合后, 系统按照启发式规则同时生成多个名为 Covering SubExpression (CSE)的数据结构, 每个CSE对应同类子表达式所要求的元组集合, 这样的CSE集合称为候选CSE. 然后优化器根据所定义的代价模型, 排除开销较大以及冗余的CSEs, 并将剩下的CSEs加入查询计划, 被选中的 CSEs 对所有的元组进行评估并产生相应的查询结果. 实验表明, CSE 有效降低了生成查询计划所需的时间.

2012 年 Yasin N. Silva 指出[25], 作者提出了一个基于识别公共子表达式的查询优化框架, 该查询优化策略如图 11 所示.

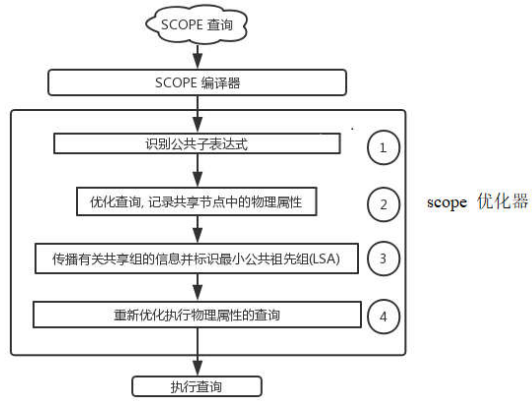

Fig. 11 SCOPE query processing flow to implement multiple query sharing technology

图 11 实现多查询共享技术的 SCOPE 查询处理流程

1) 识别公共子表达式: 系统事先将所有的子表达式分类, 并对其进行hash, 结果映射为一张hash表, 新来的query 被拆分为多个子表达式, 而公共子表达式可根据 hash 表进行快速识别, 并分为若干共享组. 2) 记录物理性质: 常规优化器被扩展从而记录步骤一中被标识的物理属性. 作为共享子表达式组的根节点, 各个查询物理属性存储在一个链表中. 3) 传播有关共享组的信息并标识最小公共祖先组(LCA). 在步骤 4 开始重新优化之前, 立即执行此步骤. 有关共享组的信息从下至下从共享组传播到根. 该过程还为每个共享的子表达式S标识其使用者的LCA. 组或节点L的LCA是DAG中从根到L中任何组的每条路径所遍历的最低组. 有关共享组和 LCA 的信息用于指导最终的优化步骤. 4) 重新优化执行物理属性的查询: 这是添加新阶段以利用常见子表达式重新优化脚本. 此步骤重新优化了在共享组中强制执行物理属性的查询. 当优化器处理不是LCA的组G时, 优化过程将照常继续. 当找到作为共享表达式S的使用者的LCA的组G时, 该过程将重新优化以G为根的子表达式, 从而传播在S被优化时使用的一组物理属性.

2014 年 Georgios 在其文中提出动态生成全局查询计划的策略[12]. 全局查询优化器将整个工作负载作为输入, 并生成执行计划, 从而最小化评估它所需的工作总量. 该策略中的多查询计划仍然是以图为基础构建的, 文中把多查询路径搜索问题转化为运算符之间的排序问题, 提出了全局查询计划的启发式的规则: 利用观察结果, 分析了连接的顺序以及中间结果的利用等维度. 将共享工作问题抽象为非凸双线性全局优化问题, 并提出了基于分支定界优化的代价模型. 该方案也存在着不足, 例如没有将问题抽象出最佳子结构, 意味着无法将其分解为更小的子问题; 对于更新操作较多的 OLTP 负载, 也并没有很好的支持.

2018年Pietro Michiardi在其文中提出了一种的多查询优化方法[60], 该方法使用内存中的缓存原语来提高数据密集型可扩展计算框架(例如 Apachespark[61])的效率.该法将输入大量使用 SparkSQL API 编写的查询作为输入, 并对其进行分析以找到共同的(子)表达式, 从而导致基于覆盖表达式的备选执行计划的构建, 该表达式包含了每个查询所需的单独工作.其生成全局查询计划的流程如下,首先通过修改哈希树以快速识别常用子图, 以及枚举可行的常用表达式的算法,对查询表达式进行合并;然后作者将多查询优化问题转换为多选背包问题：每个可行的逻辑共享查询计划都与一个值（查询之间可以共享多少工作量）和一个权重（内

存压力）相关联,确保通过缓存公用数据来确定），并且目标是最佳填充给定容量（内存限制）的背包.总体而言,该多查询执行流程与 GQP 系统的执行流程是一致的,文中能将多查询优化问题转换为多选背包问题是本文的一大亮点,但影响多查询效率的因素不仅仅与文中所提到的两个自变量关(查询之间可共享的工作以及内存压力).所以该方案的可扩展性值得进一步地研究与试验.

2018 年 Albert Jonathan 设计出一种用于分布式数据流分析查询系统的多查询算法[40],该算法将广域网络(wide-areanetwork)带宽的可用性作为首要考虑因素,从而确定需要利用哪些共享机会,最后生成全局查询计划.具体而言,对于“输入-运算符共享”来说, 查询优化器需要确保顶点共享算子v_i所在的节点具有足够的出口带宽来传输其他输出流;在“输入共享”的情况下, 查询优化器需要进一步确保入口和出口链路中都有足够的带宽来分别传输附加的输入和输出流;在此基础上,优化器对所有流过该共享算子的数据进行运算,输出结果则发往下一个算子.该算法的缺陷在于当工作负载中的查询执行时间存在较大差异时,执行较慢的查询会严重拖慢执行较快的查询. 比如查询 A 含有 5 层 join、6 层聚合,其数据可能需要在 6 个以上的广域网络节点中互相传输;若查询 B 只含有 1 层 join、1 层聚合,一旦二者进行共享,则查询 B 的执行需要付出比原来多出 6 倍以上的广域网络传输代价,这显然是非常昂贵的.

2019 年 Jeyhun Karimov 在其文中提出了一种支持共享查询的分布式数据流处理框架(AStream)[39],主要面向的场景是流处理系统中的临时查询(Ad-hocquery).具体策略如下,AStream 内维护着一个名为”共享流运算中心(sharedstreamingoperator)”的模块,随着查询的提交,系统会先将可共享的查询合并,并且为合并后的查询创建若干个”共享选择、连接、聚合等实例”, 而数据流则会依次通过这些操作实例从而得到查询结果,最后系统将查询结果反馈.该框架主要借鉴了 sharedDB[4]全局查询计划的设计思路,并为选择、连接、聚合等算子设计了增量查询处理的共享算法,使得系统能够计算临时流聚合并以增量地方式进行连接.对于差异性较大的临时查询(Ad-hocquery)来说,Astream 生成的全局查询计划往往不是最优的,作者在文章末尾也提出,在未来的工作中, 他们计划使用基于成本的优化器和自适应查询处理技术来扩展 AStream,以及通过对相似的查询进行分组来生成更优化的查询计划.

3.5 小结

基于多查询计划的优化研究种类繁多, 像多查询计划的抽象模型, 子表达式合并规则, 中间结果的数据模型, 适应新型存储结构的多查询调度以及多查询计划搜索算法等都囊括在内.目前的研究主要集中在设计出适应性更强的多查询调度策略(这些策略大多是启发式的), 即为不断变化的负载动态地生成全局查询计划, 最大化地利用查询之间的共享机会, 并保证单查询的响应时间.

多查询共享算法的性能往往依赖于多查询优化器对查询的调度, 若是不能把握住稍纵即逝的共享机会, 那么再优秀的算法也会因错过共享时机而无法发挥. 很多的共享算法研究往往不考虑多查询优化过程, 而只是研究运算符之间如何进行共享, 这就导致所研究的内容与实际应用互相脱离, 文中的实验结果固然好看, 但无法兼容到现有的多查询引擎中. 因此, 目前排序和分组的多查询共享算法研究论文最多, 但在实际的多查询系统中, 往往只应用了针对选择和连接运算符的共享算法.

4 查询执行阶段中的多查询共享技术

4.1 多查询缓存管理

1982 年 Sheldon Finkelstein 提出了热点多查询标记算法[9], 首先筛选出热点查询, 然后分析多个热点查询的表达式, 将公共子表达式合并, 并将这些查询的结果固化, 达到共享中间结果的目的. 由于当时 io 的性能严重制约了关系数据系统性能, 不少学者希望通过反复利用内存中的数据及代码以达到减少 io 开销的负担, 而将频繁使用的查询计划或者中间结果存储在内存中, 也是之后许多研究并发查询解决方案的中心思想.

2001 年 Kian-Lee 在其文中[62]提出了按需缓存(COD)的概念, 即缓存肯定会被重用的结果. 为了确定缓存的可重用性, 而不是像传统的缓存策略那样预测未来, 作者将重点放在当前系统内的查询. 该方案思路大

致如下, 1) 系统检查系统中当前正在运行的查询, 确定传入查询和正在运行的查询之间的公共子表达式, 并标识可以被重新使用的查询的中间或最终结果, 这些中间或最终结果被放在虚拟缓存中; 2) 使刚提交的查询利用这些结果. 作者将研究重点放在如何查找候选虚拟缓存以及重用虚缓存上. 为此, 作者定义了两个数据结构来保存相应的信息: 1) VirtualCache(VC), 用来跟踪所有正在运行查询的虚拟缓存上的信息; 2) OperationExecutionStatus(DES)表用来保留正在运行的虚拟缓存(或算子)的执行状态. 另外作者还提出了两种基于 COD 框架的缓存策略, 即 Conform-COD 和 Scramble-COD.

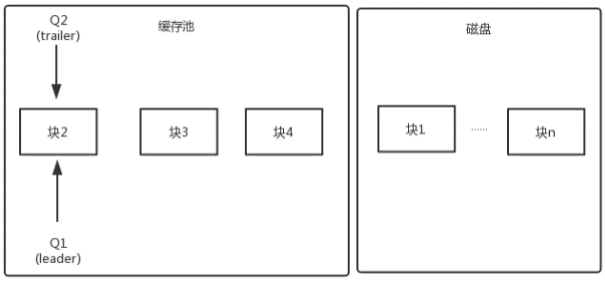

Fig.12 An example of scan group

图 12 扫描组实例

2007 年 Christian 在其文章中[63]指出, 数据库引擎在对表进行扫描的时候, 通常都是由存储引擎从第一个页面(first page)一直读取到最后一个页面(last page), 导致 buffer 的利用率低下, 因此 Christian 提出了一种协作扫描的方案, 该方案将具有相同扫描算子的查询整合到一起, 设为一个扫描组, 扫描组实例如图 12 所示. 第一个扫描算子, 称为 leader, 从起始 page 开始扫描, 之后的扫描称为 trailer, 从当前 buffer pool 中读取 page, 并记录 page 的位置, 当扫描算子再次读到标记位置的时候, 则扫描结束. 作者在 DB2 上实现了该方案, buffer 效率显著提升. 随着计算机内存带宽的提升, 越来越多的数据页可以缓存在内存中, 从而使得该方案的应用场景渐渐减少, 除非连续的查询相似性非常高, 否则, 该方案几乎没有优势. 图 11 显示了一个工作组的实例.

2008 年 Lin Qiao 等人指出[64], 内存带宽增长速度无法与 cpu 性能的增长相匹配, 而渐渐取代了磁盘 I/O, 成了查询处理的瓶颈. 他们提供了一个新颖的方案来提高查询的吞吐量, 该方案允许所有并发查询在执行基表 I/O 时共享 cpu 内核缓存, 即将查询计划分为多个批次, 每个批次聚集表(用于聚集操作的 hash 表)的工作集被写入 cpu 缓存中, 已达到共享该批次操作的目的. Lin Qiao 通过实验提出数据的选择性, 工作集大小是制约该方案的重要因素. 为了避免"快查询"和"慢查询"被分入同一批次而导致系统延迟, 以及工作集较大使得 cpu 缓存溢出等问题, 该方案提供了一种有效估计选择性和工作集大小的采样技术. 实验结果表明, 在 8 核平台上, 该方案在选择性波动较小的工作负载中的吞吐量相比同条件下主流商用数据库要高出 2 到 2.5 倍.

4.2 多查询执行模块之间的调度

2005 年 Stavros 在其文中[20]提出了协调 OMP 执行模式的策略, 并实现了 OMP 协调器. 一旦 OMP 协调器将一个或多个卫星数据包附加到主机数据包, 参与的查询之间便会形成" 1 个生产者, N 个消费者"的关系. QPipe 的中间缓冲区可调节数据流. 如果任何一个消费者的速度都比生产者慢, 则所有查询最终都会将其消费速度调整为最慢的消费者的速度. 作者在在后续工作中还研究了: 1) QPipe 如何应对频繁到达或离开的卫星扫描的负担; 2) OMP 协调器如何利用顺序敏感的重叠扫描; 3) QPipe 如何防止死锁; 4)QPipe 如何处理锁定请求和更新语句等问题.

2009 年 George Candea 等人设计的 CJoin 查询流水线模型[22], 每一个 filter 和 Aggr Operator 类似于一

个 μ-engine. 作者文中指出其面临以下优化问题：给定一个工作负载 Q 和一个 Q 的 CJOIN 管道, 确定筛选器的排序, 以使每个事实元组的探测预期数量最小. 一个复杂的因素是, 每个过滤器的选择性取决于工作负载Q, 因为过滤器会在特定的维表上对多个查询的连接谓词进行编码. 因此, 如果工作负载是不可预测的(如数据仓库中的临时分析查询), 则最佳顺序可能会随着查询组合的变化而变化. 通过观察该结果, 作者提出了一种在线方法, 用于优化 CJOIN 管道中的过滤器顺序. 这个想法是在运行时监视每个过滤器的选择性, 然后根据收集的统计信息优化顺序. 监视和重新优化的连续过程可以在 Pipeline Manager 线程内部异步执行.

2018 年 Albert Jonathan 为流分析查询系统专门设计了作业调度模式[40],一旦查询优化器生成了全局查询计划,作业调度器将广域网络带宽作为重要属性并对该查询计划进行部署.调度程序将根据其上游顶点的部署以拓扑方式将每个运算符放置并部署到物理执行计划中.在利用共享机会时, 调度程序将优先利用“输入-运算符共享”的机会,从而创建从共享顶点到两个顶点未共享的任何其他下游顶点的附加边来更新现有执行.当某个运算节点需要参与其他共享节点, 则作业调度程序无需部署该节点.另一方面, 具有“输入共享”的节点将与其对应的共享节点部署在同一站点上, 以减轻广域网络上的冗余数据传输.

4.3 小结

在多查询系统中, 数据的利用率与数据的共享性往往成正相关, 这使得多查询缓存的管理相较单查询而言更为复杂. 此外, 在大多数多查询系统中, 数据通常以push的方式自底向上推入各个运算符, 一方面增加了数据之间的共享机会, 但另一方面也导致较大的内存开销. 因此, 一个良好的多查询缓存策略还需考虑大量中间结果所带来的内存消耗问题.

由于 OMP 的执行引擎比较特殊, 如第 2 节所述, 其执行引擎的内部由若干个 μ-engine 组成, 不同 μ-engine 分别执行不同的计算, 相互之间通过 packet 传递数据. 因此多个 μ-engine 之间保持良好的通信, 相互之间协调的传递数据是 OMP 执行效率的关键.

5 多查询共享技术的应用

在多查询共享技术研究伊始, 就有不少基于该技术的系统问世, 这些系统基本上是用来解决数据仓库并发查询激增而带来的资源争用问题, 能够处理的共享运算符非常有限, 也没有较好的方案生成多查询计划. 随着多查询优化技术的不断发展----多查询计划的优化方法日趋成熟, 各种运算符的共享算法愈来愈完备, 包括图数据库, Map-Reduce, 半结构化数据库等多个领域开始尝试应用多查询共享技术来解决查询并发的问题. 本节将梳理多查询共享技术在这些领域中的应用.

5.1 关系型数据库系统中的应用

5.1.1 OLAP数据库中的多查询共享

在多查询共享技术的研究历史中, 大部分的技术都应用于处理 OLAP 负载的系统. 2000 年 Jianjun Chen 等人设计了 NiagaraCQ[11], 一个连续查询处理引擎, 旨在处理 web 查询场景下数百万并发查询. NiagaraCQ 通过归纳 Web 查询中的相似结构, 对查询分组, 将通用的计算放入内存中进行共享, 可以大大降低 I/O 成本. 此外, 对选择谓词进行分组可以消除大量不必要的查询调用. NiagaraCQ 在之前的分组优化技术基础上做了改进. 首先, 使用具有动态重新分组功能的增量分组优化策略. 新查询将添加到现有查询组中, 而无需重新分组已安装的查询. 其次, 使用查询拆分方案, 该方案只需要对通用查询引擎进行最少的更改. 第三, NiagaraCQ 以统一的方式对基于更改的查询和基于计时器的查询进行分组. 为了确保 NiagaraCQ 具有可伸缩性, 中文采用了其他技术, 包括连续查询的增量评估, 同时使用 pull 模型和 push 模型来检测异构数据源的变化以及内存缓存. 实验表明, 虽然 NiagaraCQ 的适用场景存在着局限性, 不过对于高并发的 web 查询场景, NiagaraCQ 的吞吐量要明显优于同类系统.

2007 年 Marcin Zukowski 等将协作扫描技术和多查询缓存管理技术应用到了数据仓库中, 在其文中描述了一个描述了“协作扫描”框架[23], 该框架由传统的(索引)扫描运算符的协作变体(称为 CScan)和活动缓冲

区管理器(ABM)组成. CScan 声明所需要扫描的数据范围或聚集索引的范围, 初始化后的 CScan 算子被放入 ABM 中. ABM 负责协调多 CScan 算子的执行, 以最大程度地提高 I/O 带宽重用, 同时确保良好的查询延迟. Marcin Zukowski 等人关注点在于实现一个良好的算子调度策略, 使平均查询成本最小化, 同时使最大查询执行成本保持稳定(即确保对所有查询的“公平”对待).

2009 年 George Candea 等人设计的 CJoin 查询流水线模型[22], 可以有效的评估星型设计下的并发连接查询. CJoin 应用了共享扫描技术, 优化了多查询连接算法, 其执行流程如图 13 所示:

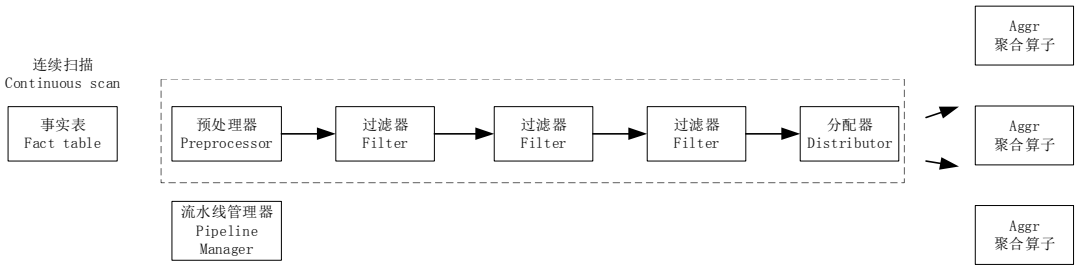

Fig.13 The Cjoin pipeline

图 13 Cjoin 查询流水线模型

$$\text{SELECT } A, Aggr_1, \ldots, Aggr_k$$

$$\text{FROM } F, D_{d_1}, \ldots D_{d_n}$$

$$\text{WHERE } \underset{1\le j\le n}{\wedge} F \bowtie D_{d_1} \text{ AND } \underset{1\le j\le n}{\wedge} \sigma_{c_j}\left(D_{d_j}\right) \text{ AND } \sigma_{c_0}(F)$$

$$\text{GROUP BY } B$$

Fig.14 Star query mode

图 14 星型查询模式

该模型专为星型查询所设计, 星型查询模式如图 14 所示. F 为事实表, D 为维度表. CJoin 引擎采用“始终在线”的单个物理计划, 该查询将事实表 F 与维度表D_i进行连接, 最后以属性 B 分组并做聚合运算. 在图 13 中, 系统不断扫描事实表, 将元组推入过滤器中, 元组在 filter 中对维度表进行 join 操作, 之后被推入聚合算子中. CJoin 流水线可同时进行多个查询, 在扫描时通过对 tuple 进行标记, 带有查询 id 的 tuple 被源源不断地推入 filter 中与维度表进行连接, 之后再推入聚合算子以生成一个多查询结果集, 最后再通过查询 id 得到查询所对应的元组. 实验中, CJoin 在高并发星型查询场景下, 吞吐量比传统商业系统要领先一个数量级. 但 Cjoin 的设计使得其支持的查询场景单一, 可扩展性不高; 并且对于设备内存要求较高, 因为其维度表必须存储在内存中, 否则事实表与维度表的连接操作将大大拖慢整个系统.

2010 年 Subi 利用各种多查询共享技术, 设计了一个以数据为中心的多查询原型数据库系统, 名为 Datapath[21]. 在 DataPath 中, 查询不请求数据, 取而代之的是, 系统自动将数据推送到 cpu 中进行计算, 一旦到达该相应的位置, 所有可以使用数据的运算符都将对它们进行所需的计算, 而 DataPath 中的共享机会在于将多个相似的运算符合并, 并在数据被推入运算符时达到共享计算的目的. Datapath 把全局查询计划中的每个运算符视为一个 waypoint, 新来的查询被拆分为若干运算符, 这些运算符要么与相同的 waypoint 的合并, 要么被创建为新的 waypoint 并加入到全局查询计划中. 系统每隔一段时间将数据 push 至各 waypoint 中.

Datapath 主要围绕两个问题进行研究: 路径网络和调度策略. 作者设计的显示调度策略如下: 调度程序为单线程, 主要对 work queue 进行操作. 当 I / O 子系统或 waypoint 产生一个块时, 该块和任何相关的元数据将被放入 work queue 中. 调度程序不断监视 work queue 中块的进出. 从 work queue 中提取块时, 调度程序有两个选项: 1) 系统可能没有足够的 CPU 资源来处理该块, 因此必须丢弃该块. 2) 调度程序可以将块提供

给相关的 waypoint. waypoint 本身在调度程序的线程上运行, 并且像调度程序一样, 它们实际上不执行任何数据处理. 取而代之的是, 一个 waypoint 将数据块打包到一个工作单元中, 该工作单元既包含数据块, 也需要处理该数据块所需的任何其他状态信息, 包括代码和元数据. 工作单元构造完成后, 调度程序将工作单元发送到工作线程, 由该线程执行工作单元. 调度程序通常为每个CPU内核维护一个工作线程. 作者设计的路径网络优化策略如下: 路径网络优化器类似于传统的基于成本的查询优化器. 系统将现有的路径网络作为输入, 然后以数据移动代价尽可能小的方式将新查询应用到路径网络中, 作者将路径网络优化视为基于成本的搜索问题, 对于一个waypoint(w)来说, 设t(w)为该waypoint所对应的算子需要执行的所有元组. 路径网络 P 的成本定义为：$\sum_{w\in P} t(w)$. 作者目前使用 A *style 搜索[19]实现将新查询集成到路径网络中的过程.

到目前为止, Datapath 的研究进展并不是很乐观, 最大的原因是当查询并发量非常大时, waypoint 的管理和维护成本也变得非常高, 新来的查询往往无法与原本的 waypoint 进行合并, 从而达不到共享的效果. 对此, sharedDB 在该方面进行了妥协, sharedDB 不允许在查询执行时将新来的运算符加入查询计划中, 在最初的设计中, sharedDB 甚至手动生成全局查询计划以处理已知的工作负载, 在之后的研究中, 多查询路径搜索技术以[12]及多连接优化算法[15]等技术被陆续应用到了 sharedDB 中. 此外, 该团队还专门为支持列存储的 sharedDB 版本[14]优化了共享算子.

2012 年 Yasin N. Silva 指出[25], 在早期利用共享表达式的共享查询系统中, 只是将结果视为一个集合, 通常对于不同用户的不同物理需求(如排序, 和结果集分区)需要分别处理, 对于某些查询来说, 加入这样一个共享查询显然是不划算的. 为了满足用户的各种需求且保证查询的响应时间, 作者提出了一个基于识别公共子表达式的查询框架, 该框架在 SCOPE[65]中实现. 期实验结果表明, 扩展的优化器生成的计划的成本降低了 21%至 57%.

2018 年 Albert Jonathan 将共享查询技术运用到分布式流分析查询系统中[40], 文中探讨了在广域流分析中应用多查询优化的机会,从而有效利用广域网络带宽, 同时实现高吞吐量和低延迟执行. 在这项工作中, 作者的团队研究了不同类型的共享机会, 并将其分为“输入-运算符共享”, “输入共享”两种,随后提出一种实用的在线算法, 该算法允许流分析查询逐步共享其执行的公共部分.该策略以有向无环图(DAG)来表示逻辑查询计划,查询优化器以顶点为中心的方式分析查询之间的共性,将广域网络(wide-areanetwork)带宽的可用性作为首要考虑因素,从而确定需要利用哪些共享机会,最后生成全局查询计划.之后文中介绍了基于该算法实现的原型系统 Sana,该系统底层为 Apache Flink [61](一款开源的数据流处理系统),作者团队已经将其部署并对其进行实验,实验结果表明,在相同条件下,Sana 的吞吐量要比没有任何共享的系统高出 21%,带宽利用率则提高了 33%.

2019 年 Jeyhun Karimov 在其文中提出了一种支持共享查询的分布式流处理框架(AStream)[39],该框架借鉴了 SharedDB[4],并对临时查询(ad-hocquery)负载做了优化.AStream 旨在通过共享资源和计算来提高数据流处理系统中临时查询的吞吐量和整体响应时间.值得一提的是,上文所提到的 Sana 也是支持共享查询的分布式数据流处理框架,二者最大的区别在于查询处理流程的不同.AStream 属于 GQP 系统,因为其需要生成全局查询计划,并为每个共享查询创建一个查询实例,而 Sana 则属于 OMP 系统,其查询处理是以运算符为中心的,集群中的每个节点拥有自己的计算实例(这类似于上文所述的 CJoin[22]),数据通过”推”(push)的方式进入各计算实例,系统不需要提前制定全局查询计划,每个 μ-engine 只需将当前提交过来的算子与正在进行的算子共享即可.由于二者均不开源,测试所用的查询负载以及数据集以均不相同,虽然均是在 Apache Flink 上实现,但我们也很难通过实验比较其性能差异.

5.1.2 OTLP数据库中的多查询共享

OLTP[3]负载存在大量更新操作, 对响应时间的要求较高, 有大量点查询等共享机会不高的查询. 这些都是共享查询技术鲜有支持 OLTP 负载的原因. 近来逐渐有团队试图解决这些不利因素, 2018 年 Robin 等人在其文中[8]系统地研究了大量OLTP工作负载, 旨在得出实际OLTP工作负载中的共享潜力. 他们围绕 1) 典型的OLTP应用程序使用了多少个不同的查询语句? 2) 哪种查询句式最常用？这两个问题, 对来自卡内基梅

隆数据库应用程序目录(CMDBAC)[57]中的大量 OLTP 工作负载进行分析, 结果 89%的应用程序只有 10 个不同的语句字符串或更少, 50%的应用程序只使用单个不同的语句字符串, 100%程序仅使用单语句事务, 并且超过 80%的语句是简单的基于键的查找查询. 从作者的结果中可以看出, OTLP 工作负载也具有相当的共享潜力. 在之后的工作中, 作者提出了一种名为 OLTPShare 的执行策略, 该策略实现了 OLTP 工作负载的查询共享方案. 该方案大致如下, OLTPShare 使用基于队列的方法批量处理所有输入的事务, 根据公共子表达式合并规则对查询进行合并, 在最初的设计中, OLTPShare 并未对合并后的全局查询进行优化. 作者在文章最后提到, 未来的工作会为 OLTPShare 实现多查询计划的优化策略. 就系统本身而言, OLTPShare 适用的场景较为苛刻, 根据作者的实验, OLTPShare 处理读写混合负载的性能仅是读负载的 50%.

5.1.3 小结

多查询共享技术被广泛地应用于基于关系模式的数据库系统中, 其中不乏 sharedDB, Qpipe, Cjoin, DataPath 等经典多查询系统. 目前几乎所有的系统都遵循 3.1 所描述的两种查询模式, 基于全局查询计划的多查询执行模式性能稳定, 可扩展性强, 应用范围更广, 依赖于良好全局查询搜索算法; 以运算符为中心的多查询执行模式共享机会更多, 但可扩展性, μEngine 之间的调度还有待优化.

5.2 非关系型数据库系统中的应用

5.2.1 图数据库中的多查询共享

2012 Wangchao 在其文中[26]将多查询优化的场景搬入 RDF/SPARQL 中, 他们指出, SPARQL 的多查询优化是一个 NP-hard 问题, 在总结了 SPARQL 查询的结构相似性, 并考虑了 SPARQL 的独特属性之后, 作者提出了一种启发式多查询优化算法, 将输入的查询分为若干组, 以便可以一起优化每组查询. 该优化的重要组成部分包括一个基于查找常见子查询结构的算法来发现多个 SPARQL 查询的通用子结构, 以及一个有效的成本模型来比较候选执行计划. 算法的基本思想是: 1) 将输入的查询划分为多个组, 同一组中的查询更有可能通过查询重写而共享公共子表达式; 2) 将每组中的类型 1[66]查询重写为其对应的具有成本效益的类型 2[66]查询; 3) 执行重写的查询并分发查询结果返回到原始输入查询. 作者的实验结果显示, 带有多查询共享的 SPARQL 查询系统要比没有多查询共享的吞吐量高出 40%~75%.

2016 年 Xuguang Ren 指出[28]现有的子图同构搜索算法不能很好的支持多查询, 在其中文中, 他们介绍了 Tri-vertex 标签序列的概念, 并提出了两个查询图之间的新颖分组因子, 可用于有效滤除不共享的子图. 文章中还提出了新型的图分区及构建该分区的启发式算法, 在此基础上, 作者设计了一种结构来将常见子图的查询结果存储在内存中, 从而共享这些结果.

2019 年 Xintong Guo 指出[27]现有的基于 SPARQL 的查询优化方案存在诸多问题, 首先这些方法主要依靠子图同构算法来检测常见的子查询, 这增加了计算复杂性, 使得单个查询的响应时间大大增多. 此外, 由于数据集越来越大, 可伸缩性问题也变得越来越严重. 文章中构建了一种支持多查询优化的分布式 RDF[67]引擎, 该引擎使用特征集来探测公共子表达式, 并且沿用了[28]中的成本模型.

5.2.2 以GPU为计算核心的数据库中的多查询共享

2013 年 Kaibo Wang 对具有复杂软件优化和硬件配置的数据库查询进行了全面研究, 他们指出, 在当前的 GPU 数据库中, 系统使用专用的优化器来处理每个单独的查询, 并不支持在并发查询之间共享 GPU 资源. 在开源 GPU 查询引擎[31]上运行 OLAP 负载并进行性能分析后, 他们观察到主要 GPU 资源的利用率仅为 25%. 于是, 该团队在文章[30]中提出了一个原型系统, 该系统可以在多个查询之间共享 GPU 资源. 该系统主要依靠 GPU 查询调度策略来实现多查询共享, 该调度的实现的策略类似全局查询计划. 并且作者还提出了一个衡量指标, 通过该指标可以量化 GPU 查询的资源需求, 从而动态控制批处理查询的数量. 实验结果表明, 通过同时执行多个 GPU 查询, 与专用处理相比, 系统吞吐量最多可提高 55%.

5.2.3 Map-Reduce中的多查询共享

2008 年 Parag 等把共享查询的场景搬到了 Map-Reduce 中, 他们在文章中[34]指出现有的以短工作优先的常规调度技术无法满足多任务工作负载的调度, 对此该团队专门针对如何共享工作负载提出了新的调度策略.该调度策略是一种在线算法, 每次执行者空闲时都会被调用, 然后从输入队列中删除非空的作业子集, 将其打包为一个执行批处理, 然后将该批处理提交给执行程序. 策略仅有的共享时机是 Map 阶段多个作业对相同文件的扫描操作, 对到达作业的类型有着较高的要求. 实验表明, 该策略处理共享率不高的工作负载时, 性能明显不如非共享的调度策略.

2010 年 Tomasz 在其文中[33]提出了一种针对 MR 的共享框架, 名为 MRShare, 该框架可以自动地在 MR 中进行工作共享. 作者总结了 MR 工作中的三种共享时机, 分别是共享扫描, 共享 map 阶段的输出和共享 map 阶段的计算. MR 的核心组件是 grouping layer, 该组件使用基于 IO 的代价模型, 将批量工作负载作为输出, 并根据三种共享时机实现多查询的调度. 值得一提的是, 通过代价模型找出最佳计划的方法被证明为 NP-hard 问题, 因此文中所提到的 grouping layer 在实现中降低了优化标准. 实验结果表明, 相对于传统的 MR 执行引擎, MRShare 的性能无论从批处理的整体响应时间还是吞吐量, 都有了明显的提升.

2011 年 Xiaodan Wang 在其文中[32]提出了 Map-Reduce(MR)多任务调度方案, 名为 CoScan, 它消除了在 MR 计算环境中扫描大量数据的工作流中存在的冗余处理. 与[33]不同的是, 作者设计了一个启发式策略在制定计划决策时支持 join 和 no-join 工作流, 该算法支持在多个 MR 任务中共享公共部分的数据. 该调度算法是一种贪心算法, 根据作业的软截止时间对作业进行排序, 并在每次迭代中合并尽可能多的排序顺序扫描具有相同输入的作业, 得出可行的作业时间表. 实验表明, 应用扫描共享可以使调度程序显著提高资源使用率. 当许多作业争用少量文件时, 导致资源使用减少三倍. 但 CoScan 目前的成本估算器存在一些问题, 比如没法根据先前的性能预估新作业的执行情况.

2012 年 Joel Wolf 等在其文中[36]提出了一种通用的 MR 共享作业方法称为 circumflex, 该方案分为两个阶段, 在第一阶段, cyclic piggybacking 提供了一种自然而有效的技术来摊销共享扫描的成本. 作业被分解为多个子作业, 这些子作业与自然优先级约束相关. 与[23]类似, cyclic piggybacking 在系统中维护者一直在线的扫描运算符, 到达的作业在对文件进行扫描时, 起始块为上一个作业正在扫描的块, 然后依次扫描到该块的前一块为止. 在第二阶段, 针对各种度量标准解决了由此产生的优先级调度问题. 作者在文中描述了最近调度算法研究的概括, 用于在此 cyclic piggybacking 范式的背景下优化调度, 该调度可根据各种成本指标进行定制, 此类成本指标包括平均响应时间, 平均拉伸和任何 minimax-type 指标等总共 11 个单独的标准指标. 该方案的性能较 MRShare 和 Coscan 略有不及, 然而策略中各个作业立即执行, 无需等待, 因此对单个作业的影响确是三种策略中最小的.

2013 年 Guoping 等在其文章中[35]提出了两种用于 MapReduce 框架中的多作业共享优化技术. 其一是通用的分组技术, 该技术将多个作业合并到单个作业中, 从而使合并的作业能够共享输入文件的扫描以及公共 Map 阶段的输出. 其二是提出了一种基于成本的两阶段优化算法来优化批量 MR 执行计划, 在第一阶段, 作者为每个作业选择 Map 输出键, 以最大化这批作业之间的共享机会. 在第二阶段, 作者将一批工作划分为几组, 并为每组选择处理技术, 以最大程度地降低总评估成本. 值得一提的是, 文中提到的 MR 工作共享策略与 MRShare 十分类似, 相比之下, 二者都提出了计算批量共享任务的代价模型, 并且在代价模型上设计了启发式的查询计划算法. 不同之处在于, 本文在广义分组技术方面更加全面, 共享的时机也更多, 但导致了更复杂的优化问题(例如, 每个作业的 Map 输出键的顺序变得很重要). 而 MRShare 启发式算法的时间复杂度为$O(n^2)$, 而 MRShare 为$O(n^3)$. 根据作者在 Hadoop 上的实验结果表明, 本文提出的方法比 MRShare 的吞吐量提高了 107%.

2015 年 Chuan Lei 等人设计了一个为 MapReduce 基础架构中的重复工作量身定制的可扩展多查询共享引擎[37], 称为 Helix. 文中主要的研究内容如下, 介绍了切片窗口对齐策略, 用于对数据进行预处理并将其划分为较小的段. 该策略分析并且确定由查询执行的关注范围之间的差异, 使用一种逻辑窗口切片方法, 将

重复出现的查询窗口划分为对齐的切片, 对切片的查询处理将产生部分结果, 这些结果可用于以很少的开销来满足多个查询. 然后作者开发了一个 SLA 驱动的优化器, 该优化器将查询的属性(例如窗口语义和 SLA 约束)合并到交错的共享和调度算法中从而为给定的一组重复查询生成执行计划, 优化程序利用分支和边界搜索策略以及各种修剪策略, 可以从指数搜索空间中尽可能早地有效修剪次优解决方案, 从而在实践中使搜索变得容易. 相比于 MRShare, Helix 能够支持的共享机会更多, 并且支持在重复查询中指定窗口约束和 SLA 要求. 而与 CoScan 相比, Helix 适用于正在不断变化的数据集, 并且对特定的运算符进行了优化.

5.2.4 半结构化数据中的多查询共享

2003年Nicolas Bruno等人把共享查询技术的研究场景搬到了XML数据库系统中[68], 他们在其文中提出了两种多查询共享技术, 一种是基于索引的多查询共享技术, 名为索引过滤器index-filter, 可以针对XML文件评估多个XML路径查询, 索引过滤器采用了PathStack算法[69], 并利用XML路径查询集的前缀树表示形式在多个查询评估期间共享计算. 另一种是基于导航的多查询共享技术, 名为Y过滤(Y-filter). 主要思想是将输入查询的前缀树表示形式扩展为如下的不确定性有限自动机(NFA): (i) NFA标识由所有输入路径查询的并集定义的确切“语言”; (ii) 当达到输出状态时, NFA输出在该状态下接受的查询的所有匹配项. 与用于标识常规语言的NFA不同, 对XML文档的过滤要求继续进行处理, 直到达到所有可能的接受状态为止. 传入的XML文档一次被解析一个标签, 解析时, 开始标记令牌会触发NFA中的转换; 解析结束标记令牌后, 执行将回溯到紧接在相应开始标记之前的状态. 作者对两种策略分别进行了实验, 结果显示当并发查询在500个以下时, 索引过滤的性能要高于Y过滤的性能, 而并发查询在多余500个时, 则是Y过滤的性能更好.

2007年Mingsheng Hong 在其文章[70]提出了一种多查询联接处理技术, 名为”大规模多查询联接处理”技术, 用于有效评估XML文档流上的并发连接查询. 该策略的核心是将连接条件分解为树模式和值比较组件, 通过两阶段的处理方式在查询之间有效地共享存储和计算, 两阶段的处理模式如图15所示. 作者为第一个处理阶段的结果开发了一个紧凑的表示形式, 核心思想是存储XPath[71]查询块中涉及的所有变量绑定的有效组合. 作者在第二阶段提出了一个可扩展的连接处理器, 主要思想是将问题映射到一个关系框架中, 该框架有助于在不同查询之间共享连接运算. 实验表明, 采用该连接共享方案的系统吞吐量较原始系统提升了100%.

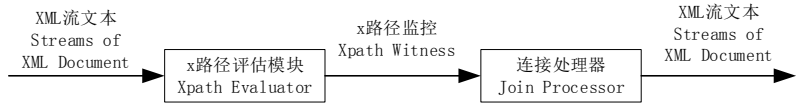

Fig.15 Two-stage processing mode in large-scale multi-query join processing
图15大规模多查询联接处理中的两阶段的处理模式

5.2.5 小结

多查询共享技术应用在此类系统的程度, 完全取决于该应用领域的兴衰, 如在本世纪 10 年代初期, Map-Reduce 的研究进行地如火如荼之际, 也正是大量出产 MapReduce 多任务方案研究文章的时期; 再如半结构化数据中的多查询共享技术也随着该领域热度地减少而停止了研究.

6 总结与展望

从上述研究结果可以看出, 共享查询技术的本质是将多个查询或任务的公共部分执行一次, 从而达到共享资源的目的. 本文对多查询共享优化进行了详细的阐述, 并梳理了基于多查询计划的各种优化技术以及多查询共享的各种算法, 之后还梳理了多查询共享优化在数据库系统领域的应用, 总结了两个通用的多查询执行模式. 总体来说, 多查询共享技术的研究历史较久, 应用范围较广, 虽未广泛地应用到工业中, 但相关技术正在渐渐成熟. 我们认为, 该领域还存在如下值得进一步研究的问题:

(1) 针对多查询计划的优化而言, 现有的研究往往是基于某个特定工作负载或硬件下的优化, 例如在[21]中, 多查询计划搜索算法根本无法适应变化的负载, 而[4]等通用的优化方案存在着单查询延迟高, 非并发

条件下性能较差等问题. 再如,变化的负载导致基于成本的全局查询计划的生成开销过大,从而无法适应某些数据流处理系统的并发场景.Jeyhun Karimov 在其文中[39]提到,在未来的工作中,他们计划使用基于成本的优化器和自适应查询处理技术[72]来扩展 AStream.我们为此类问题的解决总结出两种思路:

a) 改变现有查询流程, 从而避免全局查询计划的生成.研究团队不妨借鉴[20]所做的工作, 改进或重新设计现有的查询模式, 从而为多个查询创造更多的共享机会,这在 3 和 4 节已有详细说明.

b) 另一个是采用机器学习(MachineLearning,以下简称 ML)的方法,对查询和数据集进行训练,从而启发对多查询引擎的设计.如 2019 年, Maximilian Schleich 等人提出 LMFAO(Layered Multiple Functional Aggregate Optimization)[50],这是基于 ML 的执行引擎,用于在输入数据库上批量处理聚合运算.文中将大量查询数据集、表达式及查询计划作为训练集,并且考虑了并发,负载类型以及时间等因素, 训练的结果则作为查询优化器制定查询计划的主要依据.同样,类似的方式也可以用于多查询中.目前,使用 ML 的大规模数据分析为许多数据驱动的应用程序奠定了基础.数据管理社区已经为解决 ML 工作负载中出现的与数据管理相关的挑战进行了十多年的工作,并已建立了多个用于高级分析的系统[73, 74, 75]. Arun Kumarz 在其文中[76]全面介绍了此类系统，并分析了关键的数据管理挑战和技术.他们将研究重点专注于三个互补的方向：(1)将 ML 算法和语言与 RDBMS 等现有数据系统集成，(2)将以数据管理为核心的技术(例如查询优化，分区和压缩)适应以 ML 工作负载为目标的新系统,(3)将数据管理和 ML 思想相结合，以构建可改善与 ML 生命周期相关的任务的系统.基于以上观点,我们认为将 ML 方法与多查询共享技术结合,存在着类似的问题与挑战.

(2) 目前, 多查询共享在很多领域的应用还比较初步, 我们希望相关团队接下来会有更进一步地研究. 例如,在[8]中,作者对 OLTP 负载进行了大量研究与分析,发现 OTLP 负载也存在着较大的共享潜力.然而文中所设计的 OLTPShare 还存在着诸多不足,如支持的算子有限(目前只支持表扫描,两表连接以及聚合运算),不支持多查询写(插入,更新,删除)操作.未来可以先从 OLTP 负载中的写操作着手,研究其共享价值,并逐步设计出写操作的共享策略.再如,目前已经开始有团队[39, 40]将多查询技术应用于分布式数据流系统中,但其策略存在明显瑕疵(如 3.2 节所述).我们希望之后的研究能够朝着高可用性的方向前进.

(3) 由于大多数多查询系统(如 SharedDB,QPipe,CJoin,DataPath 等)不开源,导致后续研究无法开展,其他团队无法跟进,不少共享查询算法仅仅停留在模拟实验阶段.如 3.5 节所述, 很多的共享算法研究往往无法考虑多查询优化过程, 而只是研究运算符之间如何进行共享, 如[16],其实现仅以简单的 Java 程序来模拟数据库系统中的排序过程,而并没有将其算法在某个多查询系统中实现,因此其实验不具备太多参考价值.更为重要的是,其他的研究者无法对这些技术进行有效的横向对比,导致整个行业没有一个统一的性能评价体系.

在之后的研究中,我们将会开源地实现 GQP 以及 OMP 系统,并且在其上进一步实现各种多查询共享算法以及优化策略.我们希望能够在平台环境相当的基础上,对目前主流的多查询共享技术进行一次全面的实验与分析,旨在制定出一套该行业统一的性能评价标准;同时也希望更多的研究者参与我们的工作.

References:

[1] Garcia-Molina, Hector, Jeffrey D. Ullman, and Jennifer Widom. Database system implementation. Vol. 672. Upper Saddle River, NJ:: Prentice Hall, 2000.

[2] What is the definition of OLAP?. https://olap.com/olap-definition.

[3] What is OLTP?. https://database.guide/what-is-oltp.

[4] Giannikis, Georgios , G. Alonso , and D. Kossmann . "SharedDB: Killing One Thousand Queries With One Stone." In: Proceedings of the VLDB Endowment. 2012. 5(6): 526-537.

[5] Mingsheng Hong, Alan J. Demers, Johannes Gehrke, Christoph Koch, Mirek Riedewald, Walker M. White: Massively multi-query join processing in publish/subscribe systems. In: Proceedings of the 2007 ACM SIGMOD international conference on Management of data. 2007. 761-772.

[6] U..S. Chakravarthy and J. Minker. Multiple Query Processing in Deductive Databases using Query Graphs. In: VLDB. 1986. 384-391.

[8] Robin Rehrmann, Carsten Binnig, Alexander Böhm, Kihong Kim, Wolfgang Lehner, Amr Rizk: OLTPShare: The Case for Sharing in OLTP Workloads. In: Proceedings of the VLDB Endowment. 2018. 11(12): 1769-1780.

[9] S. Finkelstein. Common Expression Analysis in Database Applications. In:Proceedings of the SIGMOD. 1982. 235–245.

[10] A. Rosenthal and U. S. Chakravarthy. Anatomy of a Modular Multiple Query Optimizer. In: VLDB. 1988. 230-239.

[11] Jianjun Chen, David J. DeWitt, Feng Tian, Yuan Wang: NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In: Proceedings of the 2000 ACM SIGMOD international conference on Management of data. 2000. 379-390.

[12] Georgios Giannikis, Darko Makreshanski, Gustavo Alonso, Donald Kossmann:Shared Workload Optimization. In:Proceedings of the VLDB Endowment. 2014. 7(6): 429-440.

[13] Sailesh Krishnamurthy, Michael J. Franklin, Joseph M. Hellerstein, Garrett Jacobson: The Case for Precision Sharing. In: Proceedings of the Thirtieth international conference on Very large data bases. 2004. 972-986

[14] G. Alonso, D. Kossmann, T. Salomie, and A. Schmidt. Shared Scans on Main Memory Column Stores. In:Proceedings of Technical report/Systems Group, Department of Computer Science, ETH Zurich. 2012. no. 769.

[15] Darko Makreshanski, Georgios Giannikis, Gustavo Alonso, Donald Kossmann: Many-query join: efficient shared execution of relational joins on modern hardware. VLDB J. 2018. 27(5): 669-692.

[16] Jizhou Sun, Jianzhong Li, Hong Gao: Efficient Batch Grouping in Relational Datasets. In: International Conference on Database Systems for Advanced Applications. 2017. 376-390

[17] Nilesh N. Dalvi, Sumit K. Sanghai, Prasan Roy, S. Sudarshan: Pipelining in multi-query optimization. Journal of Computer and System Sciences. 2003. 66(4): 728-762.

[18] Yu Cao, RamadhanaBramandia, Chee-Yong Chan, Kian-Lee Tan: Sort-sharing-aware query processing. VLDB J. 2012. 21(3): 411-436.

[19] Ryan Johnson, Nikos Hardavellas, IppokratisPandis, NajuMancheril, Stavros Harizopoulos, KivancSabirli, AnastassiaAilamaki, Babak Falsafi: To Share or Not To Share? In: Proceedings of the VLDB Endowment.. 2007. 351-362.

[20] Stavros Harizopoulos, Vladislav Shkapenyuk, AnastassiaAilamaki: QPipe: A Simultaneously Pipelined Relational Query Engine. In: Proceedings of the 2005 ACM SIGMOD international conference on Management of data. 2005. 383-394.

[21] Subi Arumugam, Alin Dobra, Christopher M. Jermaine, Niketan Pansare, Luis Leopoldo Perez: The DataPath system: a data-centric analytic processing engine for large data warehouses. In: Proceedings of the 2010 ACM SIGMOD international conference on Management of data. 2010. 519-530.

[22] George Candea, NeoklisPolyzotis, Radek Vingralek: A Scalable, Predictable Join Operator for Highly Concurrent Data Warehouses. In: Proceedings of the 35th International Conference on Very Large Data Bases. 2009. 2(1): 277-288.

[23] Marcin Zukowski, SándorHéman, Niels Nes, Peter A. Boncz: Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In: Proceedings of the 33rd international conference on Very large data bases. 2007. 723-734.

[24] IraklisPsaroudakis, Manos Athanassoulis , Anastasia Ailamaki: Sharing Data and Work Across Concurrent Analytical Queries.In:Proceedings of the 39th International Conference on Very Large Data Bases. 2013. 6(9): 637-648.

[25] Yasin N. Silva, Per-Åke Larson, Jingren Zhou: Exploiting Common Subexpressions for Cloud Query Processing. In: 2012 IEEE 28th International Conference on Data Engineering. 2012: 1337-1348

[26] Wangchao Le, Anastasios Kementsietsidis, Songyun Duan, Feifei Li: Scalable Multi-query Optimization for SPARQL. In: 2012 IEEE 28th International Conference on Data Engineering. 2012. 666-677.

[27] Xintong Guo, Hong Gao, Zhaonian Zou: Leon: A Distributed RDF Engine for Multi-query Processing. In: International Conference on Database Systems for Advanced Applications. 2019. 742-759.

[28] Xuguang Ren, Junhu Wang: Multi-Query Optimization for Subgraph Isomorphism Search. In: Proceedings of the VLDB Endowment. 2016. 10(3): 121-132.

[29] U..S. Chakravarthy and J. Minker. Multiple Query Processing in Deductive Databases using Query Graphs. In: Proceedings of the VLDB Endowment. 1986. 384-391.

[30] Kaibo Wang, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, Xiaodong Zhang: Concurrent Analytical Query Processing with GPUs. In: Proceedings of the VLDB Endowment. 2014. 7(11): 1011-1022.

[31] Johns Paul, Jiong He, Bingsheng He: GPL: A GPU-based Pipelined Query Processing Engine. In: Proceedings of the 2016 International Conference on Management of Data. 2016.1935-1950.

[32] Xiaodan Wang, Christopher Olston, Anish Das Sarma, Randal C. Burns: CoScan: cooperative scan sharing in the cloud. In: Proceedings of the 2nd ACM Symposium on Cloud Computing. 2011.1-12.

[33] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, Nick Koudas: MRShare: Sharing Across Multiple Queries in MapReduce. In:Proceedings of the VLDB Endowment. 2010. 3(1): 494-505.

[34] Parag Agrawal, Daniel Kifer, Christopher Olston: Scheduling shared scans of large data files. In:Proceedings of the VLDB Endowment. 2008. 1(1): 958-969.

[35] Guoping Wang, Chee-Yong Chan: Multi-Query Optimization in MapReduce Framework. In:Proceedings of the VLDB Endowment. 2013. 7(3): 145-156.

[36] Joel L. Wolf, Andrey Balmin, Deepak Rajan, Kirsten Hildrum, Rohit Khandekar, Sujay Parekh, Kun-Lung Wu, RaresVernica: On the optimization of schedules for MapReduce workloads in the presence of shared scans. VLDB J. 2012. 21(5): 589-609.

[37] Chuan Lei, Zhongfang Zhuang, Elke A. Rundensteiner, Mohamed Y. Eltabakh: Shared Execution of Recurring Workloads in MapReduce. In:Proceedings of the VLDB Endowment. 2015. 8(7): 714-725.

[38] J Yang, Y Zhang, J Wang, C Xing: Distributed Query Engine for Multiple-Query Optimization over Data Stream. In: International Conference on Database Systems for Advanced Applications. 2019. 523-521.

[39] Jeyhun Karimov, TilmannRabl, Volker Markl: AStream: Ad-hoc Shared Stream Processing. In: Proceedings of the 2019 International Conference on Management of Data. 2019. 607-622.

[40] Albert Jonathan, Abhishek Chandra, Jon B. Weissman: In: Proceedings of the ACM Symposium on Cloud Computing. 2018. 412-425.

[42] Prasan Roy, S. Seshadri, S. Sudarshan, SiddheshBhobe: Efficient and Extensible Algorithms for Multi Query Optimization. In: Proceedings of the 2000 ACM SIGMOD international conference on Management of data. 2000. 249-260.

[44] Jingren Zhou, Per-Åke Larson, Johann Christoph Freytag, Wolfgang Lehner: Efficient exploitation of similar subexpressions for query processing. In: Proceedings of the 2007 ACM SIGMOD international conference on Management of data. 2007. 533-544.

[45] Diestel, Reinard: Graph Theory, Grad. Texts in Math. Springer. 2005. ISBN 978-3-642-14278-9.

[46] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, Hiren Patel: Selecting Subexpressions to Materialize at Datacenter Scale.In: Proceedings of the VLDB Endowment. 2018. 11(7): 800-812.

[47] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, Mario Vento: A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. IEEE Trans. Pattern Anal. Mach. Intell. 2004. 26(10): 1367-1372.

[48] T. K. Sellis. Multiple-Query Optimization. ACM Trans. Database Systems. 1988. 13(1):23–52.

[49] Duy-Hung Phan, Pietro Michiardi: A novel, low-latency algorithm for multiple Group-By query optimization. In: 2016 IEEE 32nd International Conference on Data Engineering. 2016. 301-312.

[50] Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen: A Layered Aggregate Engine for Analytics Workloads. In: Proceedings of the 2019 International Conference on Management of Data. 2019. 1642-1659.

[51] Ravindra Guravannavar, S. Sudarshan: Reducing Order Enforcement Cost in Complex Query Plans. In: 2007 IEEE 23rd International Conference on Data Engineering. 2007. 856-865.

[52] Thomas Neumann, Guido Moerkotte: A Combined Framework for Grouping and Order Optimization. In: Proceedings of the Thirtieth international conference on Very large data. 2004. 960-971.

[53] Raghunath Othayoth Nambiar, Nicholas Wakou, Forrest Carman, Michael Majdalany: Transaction Processing Performance

Council (TPC): State of the Council 2010. TPCTC. 2010. 1-9.

[54] Zukowski,M., van deWiel, M., Boncz, P.:Vectorwise: a vectorized analytical DBMS. In: 2012 IEEE 28th International Conference on Data Engineering. 2012. 1349–1350.

[55] Sellis, T. “Multiple Query Optimization.” ACM TODS. 1988. 13(1):23-52.

[56] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In: Proceedings of the 1979 ACM SIGMOD international conference on Management of data. 1979. 23–34.

[57] D. V. A. Zeyuan Shang and A. Pavlo. Carnegie Mellon Database Application Catalog (CMDBAC).

[58] N. Roussopolous. View indexing in relational databases. ACM Trans. on Database Systems. 1982. 7(2):258–290.

[59] Goetz Graefe and William J. McKenna. Extensibility and Search Efficiency in the Volcano Optimizer Generator. In: Intl. Conf. on Data Engineering, 1993.

[60] Pietro Michiardi, Damiano Carra, Sara Migliorini: Cache-based Multi-query Optimization for Data-intensive Scalable Computing Frameworks. In: Information Systems Frontiers. 2018. abs/1805.08650.

[61] SanketChintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Peng, Paul Poulosky: Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. In: 2016 IEEE international parallel and distributed processing symposium workshops. 2016.1789-1792.

[62] Kian-Lee Tan, Shen-Tat Goh, Beng Chin Ooi: Cache-on-Demand: Recycling with Certainty. In: Proceedings 17th International Conference on Data Engineering. 2001. 633-640.

[63] Christian A. Lang, Bishwaranjan Bhattacharjee, Timothy Malkemus, Sriram Padmanabhan, Kwai Wong: Increasing Buffer-Locality for Multiple Relational Table Scans through Grouping and Throttling. In: 2007 IEEE 23rd International Conference on DataEngineering. 2007. 1136-1145.

[64] Lin Qiao, Vijayshankar Raman, Frederick Reiss, Peter J. Haas, Guy M. Lohman: Main-memory scan sharing for multi-core CPUs. In: Proceedings of the VLDB Endowment. 2008. 1(1): 610-621.

[65] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, Jingren Zhou: SCOPE: easy and efficient parallel processing of massive data sets. In: Proceedings of the VLDB Endowment. 2008. 1(2): 1265-1276.

[66] Wangchao Le, Feifei Li: Query Access Assurance in Outsourced Databases. IEEE Trans. Services Computing. 2012. 5(2): 178-191.

[67] Eric Miller: An Introduction to the Resource Description Framework.Bulletin of the American Society for Information Science and Technology. 1998. 4(5).

[68] Nicolas Bruno, Luis Gravano, Nick Koudas, Divesh Srivastava: Navigation- vs. Index-Based XML Multi-Query Processing. In: Proceedings 19th International Conference on Data Engineering. 2003. 139-150.

[69] Enhua Jiao, Tok Wang Ling, Chee Yong Chan: PathStack : A Holistic Path Join Algorithm for Path Query with Not-Predicates on XML Data. In: International Conference on Database Systems for Advanced Application.s2005. 113-124.

[70] Mingsheng Hong, Alan J. Demers, Johannes Gehrke, Christoph Koch, Mirek Riedewald, Walker M. White: Massively multi-query join processing in publish/subscribe systems. In: Proceedings of the 2007 ACM SIGMOD international conference on Management of data. 2007. 761-772.

[71] J Clark, S DeRose: XML path language (XPath) version 1.0.

[72] DEWITT, D., GHANDEHARIZADEH, S., SCHNEIDER, D., HSIAO, H., BRICKER, A., AND RASMUSSEN, R. The Gamma database machine project. IEEE Transactions on Knowledge and Data Engineering. 1999. 2, 1.

[73] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, Xiaoqiang Zheng: TensorFlow: A System for Large-Scale Machine Learning. In: 12th symposium on operating systems design and implementation. 2016. 265-283.

[74] MertAkdere, UgurÇetintemel, Matteo Riondato, Eli Upfal, Stanley B. Zdonik: The Case for Predictive Database Systems:

Opportunities and Challenges. In:CIDR. 2011. 167-174.

[75] ArashAshari, Shirish Tatikonda, Matthias Boehm, Berthold Reinwald, Keith Campbell, John Keenleyside, P. Sadayappan: On optimizing machine learning workloads via kernel fusion. ACM SIGPLAN Notices. 2015. 173-182.

[76] Arun Kumar, Matthias Boehm, Jun Yang: Data Management in Machine Learning: Challenges, Techniques, and Systems. In: Proceedings of the 2017 ACM International Conference on Management of Data. 2017. 1717-1722.

附中文参考文献:

[7] 崔跃生,张勇,曾春,冯建华,邢春晓.数据库物理结构优化技术.软件学报, 2013, 24(4):761−780.

[41] 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.

[43] 覃雄派,王会举,杜小勇,王珊.大数据分析——RDBMS 与 MapReduce 的竞争与共生.软件学报,2012,23(1):32-45.