面向Select和Sort的数据库算子缓存的设计与实现

蔡万里[1)] 王新硕[1)] 胡卉芪[1)] 蔡 鹏[1)] 周 烜[1)] 屠要峰[2)]

[1)](华东师范大学数据科学与工程学院 上海 200062)

[2)](中兴通讯股份有限公司 南京 210012)

摘 要 缓存是数据库中提高查询性能的一种常用技术.目前,现有数据库缓存主要有两个方向:查询结果缓存和存储层块缓存.查询结果缓存是利用数据库查询执行的最终结果或中间结果(如子查询),而存储层块缓存则缓存查询涉及的底层数据块.本文从另外一个角度"缓存中含有的计算量"来重新审视缓存在查询优化中的应用,并以此为基础进一步划分数据库缓存方式.在查询执行过程中,数据库查询被转换成一系列操作(例如选择、排序等)的集合,而算子对应操作.查询处理中算子输出的数据为中间结果,含有部分计算量,我们将这部分数据进行缓存并加以利用.我们将这种缓存部分计算量的缓存方式称为算子缓存,即缓存每个操作执行后的结果.由于不同查询之间可能会存在相同算子,对相近数据执行相同计算,因此利用算子缓存加速查询执行性能具有相当大的潜力.本文的新颖之处在于从缓存含有的计算量角度出发,提出并研究算子缓存如何在查询优化中应用.本文以Filter、Sort算子为例,针对缓存复用提出了一种基于语义树的匹配算法,用于快速匹配缓存中的结果集.同时,针对复用缓存可能劣化查询性能的情况,提出使用基于成本的代价优化器防止使用缓存劣化查询性能.最后,本文基于开源分析型数据库ClickHouse实现了Filter、Sort算子缓存的原型,并对提出的算子缓存方案进行了大量的实验测试.结果表明,相比块缓存、物化视图方式,本文提出的算子缓存方案在本地SSD部署下最大能够分别提升9倍以及1.5倍的查询响应速度,在云环境下部署能够分别提升30倍以及2倍的查询响应速度.

关键词 数据库;查询执行;查询优化;算子缓存;联机分析处理

中图法分类号 TP315 **DOI号** 10.11897/SP.J.1016.2024.02084

Design and Implementation of Database Operator Cache for Select and Sort

CAI Wan-Li[1)] WANG Xin-Shuo[1)] HU Hui-Qi[1)] CAI Peng[1)] ZHOU Xuan[1)] TU Yao-Feng[2)]

[1)](*School of Data Science and Engineering, East China Normal University, Shanghai* 200062)

[2)](*ZTE Corporation, Nanjing* 210012)

Abstract Caching is a commonly used technique in databases to enhance query performance. Currently, existing database caching primarily falls into two directions: result caching and block caching. Result caching involves utilizing the final or intermediate results (such as subqueries) obtained during the execution of database queries, while block caching stores the underlying data blocks involved in the queries. This paper takes a different perspective, focusing on the 'computational load' contained within the cache, to reexamine the application of caching in query optimization. Building on this, the paper further classifies database caching methods. During query exe-

收稿日期:2023-10-13;在线发布日期:2024-06-21.本课题得到国家自然科学基金(No. 92270202)、上海市自然科学基金(No. 23ZR1418300)资助、中兴通讯研究基金(编号 HC-CN-20220721010)资助.**蔡万里**,博士研究生,主要研究领域为OLAP数据库的查询优化、查询执行引擎.E-mail:52265903008@stu.ecnu.edu.cn.**王新硕**,硕士研究生,主要研究领域为大数据、数据库.**胡卉芪**(通信作者),博士,副教授,主要研究领域为分布式事务、分布式一致性理论、基于新硬件的云数据库.E-mail:hqhu@dase.ecnu.edu.cn.**蔡 鹏**,博士,高级研究员,主要研究领域为高性能内存事务处理、高可用机制.**周 烜**,博士,教授,主要研究领域为高性能数据库和信息检索.**屠要峰**,博士,研究员,主要研究领域为大数据、数据库、机器学习、云计算.

cution, a database query is transformed into a collection of operations (such as selection, sorting, etc.), with each operation corresponding to an operator. The data output by operators during query processing forms intermediate results, containing a portion of the computational load. We cache and leverage this subset of data. This caching approach, which caches a portion of the computational load, is termed 'operator caching', specifically caching the results of each operation execution. Due to the potential presence of common operators across different queries, performing similar computations on similar data, leveraging operator caching holds significant potential for accelerating query execution performance. The novelty of this paper lies in its exploration of how operator caching can be applied in query optimization, viewed from the perspective of the computational load contained in the cache. Using Filter and Sort operators as examples, we propose and investigate how operator caching can be employed in query optimization. For cache reuse, we introduce a semantic tree-based matching algorithm designed to efficiently match result sets in the cache. Simultaneously, to address the potential degradation of query performance caused by reusing the cache, we suggest the use of a cost-based optimizer to prevent the degradation of query performance. Finally, based on the open-source analytical database ClickHouse, this paper implements a prototype of the Filter and Sort operator caching and conducts extensive experimental testing of the proposed operator caching scheme. The results indicate that, compared to block caching and materialized view approaches, the operator caching solution proposed in this paper can achieve a maximum improvement of 9 times and 1.5 times in query response speed when deployed on a local SSD. When deployed in a cloud environment, the operator caching approach can achieve respective improvements of 30 times and 2 times in query response speed.

Keywords　database; query execution; query optimization; operator cache; OLAP

1 引　言

随着云计算、云存储等技术的发展,OLAP 数据库架构也逐渐转向云架构[1].云架构下数据库通常采用分层或解耦的模型,其中弹性计算层可访问远程云存储,例如 Amazon S3[2]和 Azure Blobs[3].由于强大的弹性伸缩能力,云存储受到企业的青睐,但也给 OLAP 数据库带来了新的挑战[4].考虑到远程云存储相对较高的延迟和较低的带宽,缓存数据变得非常重要.因此,云环境下针对 OLAP 数据库的缓存技术重新获得了关注,例如 Alluxio[5]和 Snowflake[6]等,都是云环境下优化 OLAP 数据库的缓存技术.本文的研究旨在使用缓存提高 OLAP 数据库在云环境下的查询处理性能.

按照缓存作用的方式,数据库缓存主要分为查询结果缓存和存储层块缓存两大类.存储层块缓存贴近存储设备,查询结果缓存贴近用户层.块缓存主要利用数据块访问的空间局部性和时间局部性来减少存储访问的 I/O 开销.查询结果缓存则是缓存数据库所执行的查询语句以及该语句的结果集,缓存命中时会直接从内存中返回查询结果,无需再重新执行整个或局部查询,不仅省略了存储访问的 I/O 开销,还省略了大量内存与 CPU 消耗的数据计算.因此查询结果缓存在缓存命中的前提下对查询性能会有显著提升.

基于上述对现有缓存方案的观察与理解,本文从"缓存中含有的计算量"这个角度来进一步划分数据库缓存方式.计算量是指数据库系统在处理和执行查询所需的计算资源量.它包括对数据库中存储的数据进行选择、投影、连接、聚合等操作所涉及到 CPU 与内存等各方面资源.在数据库中,通常查询优化器会通过代价模型来估算算子的执行代价,它利用代价模型分别计算各个算子的资源消耗代价.而查询中包含的所有算子的代价总和即代表了该查询所包含的计算量.

为了减轻查询执行包含的计算量,用户可以通过缓存查询结果的方式避免重复计算.例如,针对图 1 右上角的查询,如果缓存最终结果,当查询重复执行时,数据库可以在不做任何计算的情况下直接返

回结果，意味着它节省了整个查询的所有计算量（包括子查询中的连接、过滤及外部查询的分组、排序、聚合算子的计算）. 现有方法也可以通过物化视图的方法缓存中间结果，即图中子查询结果（包含连接、过滤算子的计算），那么缓存命中时，数据库只需在缓存数据的基础上继续执行分组、排序以及聚合操作即可获得最终结果.

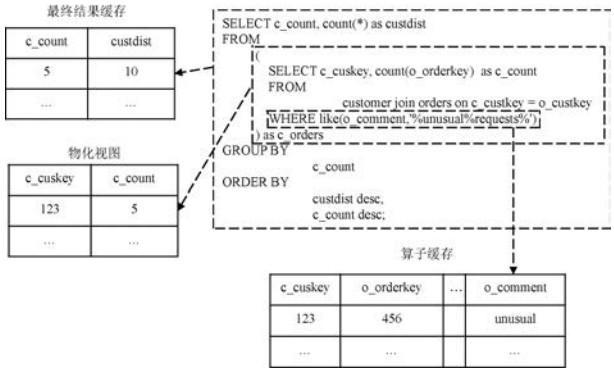

图 1 缓存中计算量

算子是指数据库中对数据进行特定处理的操作的集合，通常数据库内部都会实现一系列基础算子，例如面向谓词条件过滤的算子，面向排序的算子，面向 Limit 裁剪的算子等. 而数据库查询可以看成算子的集合，由一系列算子按照特定顺序串联组成. 查询处理中算子输出的数据为中间结果，含有部分计算量，如果将这部分数据进行缓存并加以利用，便可以在算子粒度上复用缓存数据. 这种缓存算子处理结果的方式，我们称之为算子缓存. 算子缓存的核心思想是：缓存查询中算子（例如，Filter、Sort）的处理结果，当新查询包含相同算子时便可直接复用缓存结果. 之所以选择研究算子缓存，主要因为其包含单个算子的计算量，在缓存复用方面会更灵活. 相比查询结果缓存，算子缓存复用条件更加宽松，对于算子缓存而言，复用缓存只要求查询中包含相同算子并且算子内具有相同或相近谓词条件即可，因此缓存的数据更容易被查询复用. 相比块缓存，算子缓存包含一定计算量，因此缓存更加高效. 尤其在云环境中，数据库通常采用存算分离架构，算子缓存高效的缓存方式会为查询优化带来巨大潜力.

图 2 展示了算子缓存在缓存分类中的位置. 最底部是存储层块缓存，面向存储设备，数据库中的缓冲区（Buffer Pool）、大数据领域的 Alluxio 等加速方案都属于该范畴. 往上就是查询结果缓存方案，包括最终结果缓存、物化视图以及并发查询共享等方式. 各种缓存方案自底向上观察，呈现出“含有的计算量”逐渐增多趋势，自顶向下观察则呈现出越来越贴近存储设备的特征. 算子缓存属于中间结果缓存的一种实现方式，介于块缓存和最终结果缓存之间，缓存中包含的计算量比其他查询结果缓存更少. 在图 1 中算子缓存只会缓存子查询中过滤算子的结果，当复用缓存时只需在算子缓存结果的基础上执行剩余操作（在本例中包括子查询中的连接、过滤以及外部查询的分组、排序、聚合）即可获得最终查询结果.

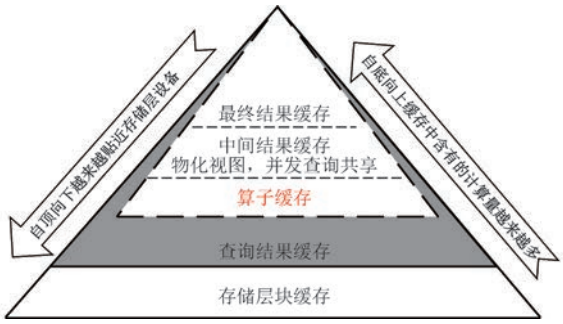

图 2 算子缓存在缓存分类中的位置

在算子缓存实现中，本文主要选取 Filter（其中 Filter 算子是关系代数 Select 的具体实现①）以及 Sort 算子. 选择 Filter 和 Sort 算子是因为该算子的广泛应用性，尤其是在全面云化的背景下，许多 OLAP 引擎都选择云上存储作为主要存储手段，算子缓存将发挥巨大的加速效果；其次，选择 Sort 算子的另外一个原因是出于 Sort 算子缓存潜在的巨大加速效果和期望探索更高级算子缓存的意愿.

总结来说，在设计并实现 Filter、Sort 算子缓存时需要面对和解决以下几个困难与挑战：

（1）如何表示、存储与使用算子缓存. 算子缓存的表示是一个非常重要的问题，关系到后续数据处理是否方便、数据是否能够只进行较少的处理就直接返回，直接决定了算子缓存的工作效率. 算子缓存数据的存储格式也关系到工作效率，如果格式转换的时间比较长，那么就很难高效地工作. 当把算子的输出进行缓存后，如果每个算子只能对相同的 SQL 做出反应，那么本质上就与查询结果缓存没有区别. 因此需要探索新的缓存匹配算法，尽可能复用不同 SQL 之间 Filter 和 Sort 算子的结果，为大幅度减少网络、计算开销提供可能.

（2）如何防止算子缓存劣化查询执行性能. 尽管算子缓存在极大多数情况下能够加速查询性能，但

① 在无歧义的情况下本文将用 Filter 算子表示论文标题中的数据库 Select 算子.

还是存在极端情况使得算子缓存劣化查询.这种情况下使用算子缓存中的数据进行过滤、反转然后返回反而会比直接从底层存储中读取数据更耗时.所以,并不是任何时候算子缓存都具有正向收益,需要一个自动化的内嵌入数据库引擎内部的策略来防止算子缓存劣化查询执行性能.

针对上述问题本文深入研究并总结了目前多种查询加速方案,并在此基础上实现了 Filter 和 Sort 算子缓存.本文主要贡献以及工作可以总结为以下几点:(1)使用语义树表示算子计算结果,并设计、实现基于语义树的缓存匹配算法,解决了缓存结果的表示以及复用问题;(2)基于算子缓存的特点,设计并实现基于成本的代价优化器,通过增加算子缓存代价计算逻辑、修改代价优化器解决算子缓存劣化查询性能的问题;(3)基于分析型数据库 ClickHouse 实现了 Filter 和 Sort 的算子缓存原型,并进行大量测试实验.实验结果表明相比块缓存以及物化视图方式,算子缓存方案能够分别提高最多 9 倍以及 1.5 倍的查询执行效率,在云存储应用场景下能够分别提高最多 30 倍以及 2 倍的查询执行效率.

本文第 2 节介绍数据库缓存研究的相关工作;第 3 节给出算子缓存设计的总体架构图,并详细介绍各部分的作用;第 4 节介绍本文的核心技术,缓存匹配算法以及基于成本的代价优化器;第 5 节通过实验验证了本文提出的算子缓存的加速性能;第 6 节讨论算子缓存对其他算子的扩展;第 7 节总结全文.

2　相关工作

过去数年学术界已有许多在查询缓存上的研究工作,按照缓存的作用方式可以将缓存分为两大类:查询结果缓存以及存储层块缓存.查询结果缓存核心思想在于利用重复计算,缓存查询过程中产生的部分或最终结果,并通过语义匹配的方式复用缓存.存储层块缓存的核心思想在于数据缓存,减少从磁盘加载数据开销,将查询相关数据块保存在内存中,确保在查询执行过程中相关数据都已加载到内存从而加速查询,属于一种通用且普适的查询加速方案.表 1 总结了有关各缓存方式的相关工作,接下来将对各类缓存方式进行详细介绍.

表 1　缓存相关工作

缓存方法	缓存方式	构建方式	相关文献
查询结果缓存 — 最终结果缓存	缓存查询最终结果	数据库自动构建	文献[7-12]
查询结果缓存 — 物化视图	缓存虚拟表	根据业务知识手动构建	文献[13-23]
查询结果缓存 — 并发查询共享	缓存并发查询公共子表达式	少部分数据库支持自动构建	文献[24-32]
存储层块缓存	缓存底层数据块	数据库自动构建	文献[33,34]

2.1　查询结果缓存

查询结果缓存将查询和对应查询结果进行映射,将查询结果保存在内存中并通过语义匹配的方式复用缓存.按照实施方式不同,查询结果缓存又可以分为:最终结果缓存和中间结果缓存.接下来对这两种缓存方式进行详细介绍.

查询(最终)结果缓存:最终结果缓存最早在 Postgres[7] 中提出,旨在缓存当前查询的结果,以加速重复查询.查询结果缓存可以自动决定哪些结果需要缓存,且大小不超过预算,决策通常采用基于成本的策略.该策略需要将视图的几个属性考虑在内,如结果大小、访问频率、物化成本等.对于重复查询而言查询结果缓存特别有效,但其缺点也很明显,只能对相同查询生效,因此缓存结果的可重用性很低.为了提高查询结果缓存的可重用性,部分研究[8-12]考虑了对查询结果缓存的重叠,即允许缓存被作为后续查询结果集的一部分来应答缓存中不可用的查询.Ahmad 等人[8]提出通过语义描述来维护缓存数据,并对数据划分合适的语义区域.查询执行时通过语义匹配的方式返回合适数据.而 Ahmad 等人的方法中将查询结果集分割为重叠和非重叠区域,使缓存能够使用传统的替换策略来管理非重叠区域.然而这种方法会产生结果集碎片,导致处理上的巨大开销,维护非重叠结果集的代价十分昂贵,因为访问重叠的结果集可能会导致结果集分割和重写缓存.为解决这一问题,后继者提出基于块的结果缓存[9],将“超域”分割成大量独立于查询的区域,极大程度上减少了结果集碎片的产生,从而降低结果集碎片的管理开销.

中间结果缓存:由于缓存最终结果只针对重复查询,可重用性不高,因此已有研究探索重用中间结果而不是查询最终结果的想法.对查询中间结果缓存至少包含两个研究方向:物化视图和并发查询共享数据.接下来详细介绍这两种缓存技术.

物化视图:在关系数据库管理系统中,视图是应用于表的虚拟化技术,表示数据库查询结果的虚拟

表.在设计架构时常常使用视图来表示数据的子集、汇总数据(例如聚合或转换数据),或者用来简化跨多个表的数据访问.使用数据仓库时,视图可以为一些商业智能工具简化来自多个表的聚合数据访问.视图提供易用性和灵活性,但无法加快数据访问速度,在应用程序每次访问视图时数据库系统都必须评估代表该视图的底层查询.在性能至关重要的应用中,数据工程师会转而使用物化视图.

物化视图(Materialized View)是一个包含查询数据的数据库对象[13-15],是对视图的缓存.它不是在运行时构建和计算数据集,而是在创建的时候预先计算、存储和优化数据访问,物化视图如同常规表数据一样,随时可供查询使用,缺点是在底层数据更新时不会刷新.

物化视图极大地改善查询处理时间,特别是对于大型表上的聚合查询,要实现这种潜力,查询优化器必须知道如何利用物化视图.Goldstein J[16]提出了一种算法用于确定是否可以从物化视图中计算出查询的部分或全部结果,并描述了如何将其整合到基于代价的优化器中.当前版本处理的视图只由选择、连接和最终的聚合函数组成,优化仍然是完全基于成本模型.与此同时 Unger C 等人[17]提出了在多表连接的情况下使用物化视图计算出分组和聚合结果的新算法.他们说明了在查询具有分组和聚合而视图不存在分组和聚合的情况下,只有在视图和查询的一部分之间存在重叠时,视图才可用来响应查询.其次,当视图具有分组和聚合时,就需要确定视图中显示的聚合信息足以执行查询中所需的聚合计算的条件.物化视图的构建和使用往往需要借助数据管理人员丰富的业务知识,这进一步限制了物化视图的应用.因此,已有许多研究[18-20]利用人工智能技术自动推导并构建物化视图.

物化视图实现多集成在数据库内部,但随着系统的用户数和访问量的提升,数据库会收到越来越多的并发请求.数据库是从磁盘中读取数据,性能较低,随着请求数的增多,数据库受到的压力会越来越大.而应用访问数据库的连接数有限,当数据库的处理能力跟不上请求数时,新的请求将排队等待,从而导致后台程序阻塞.当并发请求数持续增大时,数据库甚至会出现宕机情况.为了分担数据库压力,将缓存管理从数据库中解耦,数据库研究人员提出了中间件缓存,即将热点数据建模为物化视图,并将数据保存在第三方内存中.由于缓存是将数据存入内存中,读取速度非常快,在成功提升性能的同时还替数据库分担大量压力.针对数据库中间件缓存的优化工作有很多[21-23],最具有代表性的工作便是 Goldstein P 等人提出的 MTCache[23].MTCache 是一个基于 SQL Server 构建的中间件缓存解决方案原型,其将查询过程中经常访问的数据建模为物化视图并存储在远端内存中.当查询执行时,优化器会根据缓存中的数据将查询改写为 UnionAll 形式,确保最大可能利用缓存数据.

并发查询共享数据方案:数据库查询通常访问一些公共关系,或者共享一些公共子表达式.因此,也有研究人员提出在并发查询之间共享计算.核心思想是缓存多个并发查询之间公共子表达式计算的结果,多条并发查询之间只需要执行一次计算即可.基于这一思想,Lee Tan 等人[24]在论文中提出一种称为按需缓存(cache-on-demand,CoD)的框架来研究缓存问题.CoD 框架将现有查询的中间结果视为后续查询可以利用的缓存.这种方法成本确定、回报已知,而且缓存肯定会被重用.Tang 等人[25]在此基础上观察到许多应用程序以一种间歇性的、但在很大程度上可预测的模式获取数据,而面对决定如何更新正在进行的查询时,现有系统往往会忽略数据是如何到达的.为了解决这一缺点,他们提出了一种新的查询处理范式——间歇查询处理(Intermittent Query Processing,IQP),他们将查询执行和策略结合起来以确定何时更新结果以及为确保查询快速更新应该分配多少资源.对于查询系统会提供了一个初始结果,当策略要求时该结果将被刷新.在间歇数据到达之间,IQP 通过新记录的到达模式,有选择地释放一些在正常执行中占用的资源,从而使查询执行处于不活跃状态.

然而,以上提出的这些工作只会在大量并发查询之间共享中间结果,对重叠查询的时间局域性施加了限制.为此,其他研究工作[26-32]尝试去存储公共表达式的中间结果,以便在后续查询中重用它们.

2.2 存储层块缓存

存储层块缓存做为一种通用且普适的缓存方法已经普遍应用于各种场景,在数据库领域中已被广泛应用.例如,传统数据库的页缓冲区缓存最近经常访问的数据块,减少从磁盘加载数据的开销;在云数据仓库中,经常将远程存储中的数据缓存在分布式内存中.Alluxio[33]是一个开源的基于内存的分布式缓存系统,可以对外提供缓存加速功能以及简单的数据处理功能.因此,可以将热点数据加载到 Alluxio 中进行管理,而数据库在进行数据访问时可以

直接从 Alluxio 中读取数据,不必跨网络从远程存储中加载数据,减少大量网络和磁盘开销从而提高查询性能. Alluxio 虽然可以缓存数据,但仍然避免不了网络数据传输. 为了降低数据传输带来的延迟,T. Neumann 等人[34]提出 Crystal. Crystal 是一种新型智能缓存存储系统架构,其客户端是具有下推谓词的特定于数据库的数据源,本质上与 DBMS 类似,集成了查询处理和优化组件,专注于高效缓存和服务称为区域的单表超矩形. Crystal 主要针对 Filter 操作缓存,利用云存储系统提供的谓词下推功能将谓词发送到远程存储避免检索所有数据从而减少大量网络传输开销. 完成过滤后的数据被缓存到本地 SSD 中,缓存粒度以区域为单位,每个区域对应列的谓词过滤条件,当缓存命中时内存所加载的数据基本都是满足谓词条件的数据,因此该缓存方式有很高的缓存利用率.

2.3　缓存加速方案横向对比

本文将主要从缓存中包含的计算量、构建方式以及缓存复用三个方面分析算子缓存与其他缓存方案的不同之处.

从缓存包含的计算量角度而言,算子缓存含有的计算量介于最终结果缓存和存储层块缓存之间,处于中间结果缓存最底端. 因此,在相同内存消耗下会比块缓存更加高效.

从缓存的构建方式上看,最终结果缓存以及块缓存都是在数据库中自动构建,中间结果缓存实现方案,如物化视图,则需要借助额外工具,即预先定义要物化的局部查询,手动构建. 相比而言,本文实现的算子缓存可以自动构建,能够减少数据管理人员的工作负担.

就缓存复用而言,现有查询结果缓存只能针对重复或相近查询,而算子缓存并不局限于查询本身,当查询中具有相同算子并且算子包含的谓词条件相近时就可复用缓存数据. 因此,在缓存复用上会比其他缓存方案具有更高的缓存命中率.

综上,算子缓存是在缓存计算量和缓存复用之间一个较好的权衡.

3　算子缓存总体架构

算子缓存包括缓存匹配、代价优化器以及缓存池三个部分,如图 3 所示算子缓存的实现位于执行引擎,包括语义树构建、语义树匹配、代价优化器、物化/拉取数据. 缓存匹配负责将谓词表达式转换成语义树并将其同缓存池中的结果进行匹配;代价优化器在匹配到结果后执行,评估缓存复用代价防止缓存劣化查询性能;缓存池则用于存储数据块以及对应语义树. 具体而言,当前端发出一条 SQL 查询后,经过一系列处理最终转换为一条可执行的流水线计划,当查询包含过滤或排序操作时便会触发算子缓存逻辑. 首先,在算子内部会将谓词条件转换为语义树形式(3.1),然后同缓存中的数据进行语义匹配(3.2). 如果匹配为真,说明过滤操作的结果在缓存中,此时代价优化器会对复用缓存进行代价估算(3.3). 如果使用缓存能带来正向收益,则复用缓存池中维护的缓存数据(3.4),反之则按原查询计划执行(3.5). 若在查询匹配阶段失败,则将此次过滤结果进行物化并将其存储在缓存池中(3.6). 以上便是算子缓存的整体执行流程,接下来对每个部分进行详细介绍.

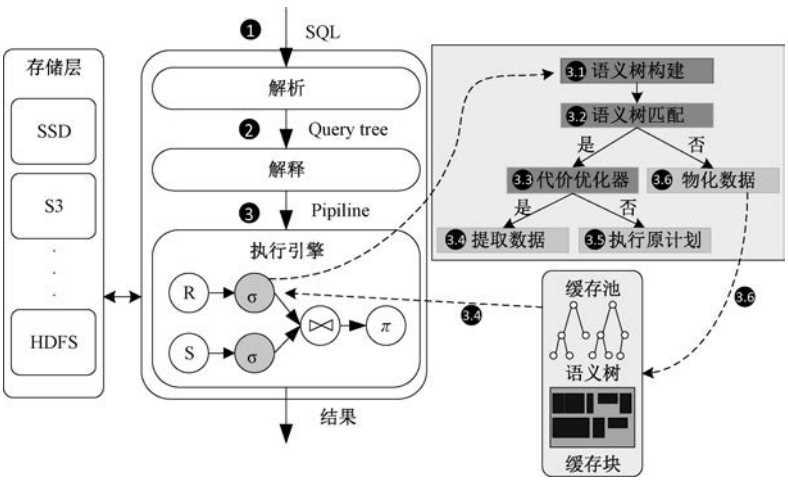

图 3　算子缓存架构

缓存匹配是算子缓存能否发挥作用的关键，采用语义匹配方式匹配缓存结果，包括语义树构建、语义树匹配两个阶段. 前者会提取出算子的谓词条件，并为其构建语义树. 后者则是利用前者产生的语义树进行结果集匹配，识别可以复用的结果集. 语义树本质上是对谓词条件的结构化表示，表达谓词语义信息的同时也代表了算子的计算结果. 因此，缓存匹配的实现依托于语义树匹配算法，通过比较语义树之间的关系就可以获取结果集之间的匹配关系（详见 4.1 小节）.

代价优化器是为了防止使用缓存劣化查询性能而设计. 考虑以下两种情况：1）缓存中的数据范围很广，而复用的有效数据只占缓存数据的极少一部分；2）缓存池中 Sort 算子的计算结果根据时间排序，而当前查询中包含的 Sort 算子按主键排序. 缓存中的数据包含查询中的 Filter 或 Sort 计算结果，但并不完全匹配. 此时要复用缓存需要添加额外（过滤、重排序等）操作. 如果额外操作的代价很大，甚至超过了使用缓存带来的收益，那么使用缓存就会降低查询性能. 代价优化器的设计便是为了解决这种情况. 优化器会根据系统内部的统计信息计算出原查询计划和使用缓存时的成本，通过比较两者之间的成本大小来决定最终是否使用缓存，从而避免因额外操作成本过大而劣化查询性能的情况（详见 4.2 小节）.

缓存池是一个有限的内存缓存，存储之前查询的算子的结果，包括算子处理结果的数据块以及表示该结果的语义树. 缓存池提供了一个灵活实现，用户可以根据实际情况选择自己的缓存组件. 缓存数据的格式也需要重点关注，这关系到后续缓存复用的效率. 在 OLAP 场景下数据多会采用列存储，而 Parquet 是大数据领域一个较为通用且高效的列存格式，因此本文将采用 Parquet 格式来缓存数据. 这样的设计方式主要有三点考虑：(1) Parquet 是一种压缩存储格式，在数据传输上会节省大量网络开销；(2) 许多其他分析型数据库或计算引擎原生集成了对 Parquet 格式的支持，因此即便在不同数据平台上，只要执行的 SQL 具有相同或相似算子时也能复用缓存数据，最大程度地提高了缓存复用的可能性；(3) 分析型数据库大多有其内部的列存格式，从列到列格式的转换具有较高转换效率. 这种设计是 CPU 和网络传输、内存之间的一种权衡，压缩存储节省网络传输、内存的同时，也带来了数据格式转换的额外 CPU 开销. 在查询执行的过程中，如果在算子处理逻辑中数据格式的转换操作会消耗大量 CPU、内存资源，将会严重影响整个查询的执行性能.

算子缓存工作机制：假设某个查询的关系代数如图 4 所示. 表示从表 A，B 中选择数据并在排序后连接结果，最后将数据排序并聚合后输出. 查询首次执行时会将 Sort 算子的计算结果进行缓存，同时继续执行剩余步骤. 当有其他查询执行时（查询中包含 Filter 或 Sort 算子，这里假设都包含），若 Filter 算子的谓词可以匹配（在 4.1 节中将对缓存匹配方式进行详细介绍），则说明当前查询可以复用 Sort 算子的计算结果. 然而，根据前文描述可知算子缓存的复用可能会劣化查询性能，在复用缓存前需要进行代价评估. 若缓存收益大于重计算开销，则查询可直接从缓存中获取数据（浅蓝色 Cache 中），并在此基础上继续执行后续处理步骤，无需从存储层读取数据，也不用重复计算（红色方框部分的算子）.

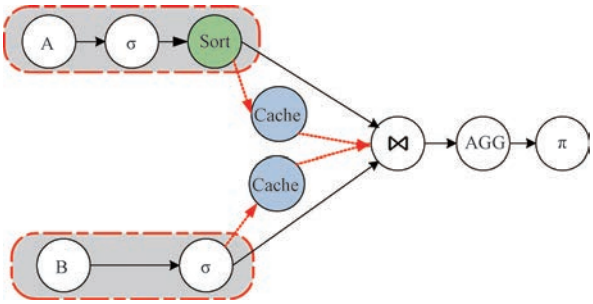

图 4 算子缓存工作机制

算子缓存能够消除大量不必要的重复计算，但算子缓存的使用也有一定限制. 目前，算子缓存的工作机制只支持在单条流水线中使用，且 Sort 算子与 Filter 算子之间不能包含聚合、连接等复杂操作. 例如，在本例中 Join 算子之后的 Sort 算子就不支持缓存（如果 Join 算子后面存在 Sort 算子），因为此时要复用 Sort 算子缓存需要保证 Join 算子之前的所有算子均要相等（谓词条件、执行路径完全一致），本质上等同于查询结果缓存.

4 算子缓存实现

上节中介绍了算子缓存的整体架构设计，接下来本节将会详细介绍算子缓存的实现细节. 4.1 节将会介绍基于语义树的缓存匹配实现，包括语义树构建以及语义树匹配算法，4.2 节则主要介绍代价优化器部分的实现.

4.1 基于语义树的缓存匹配

缓存匹配是算子缓存能否发挥作用的关键，本

文采用基于语义树的匹配方式来判断缓存数据的匹配关系.语义树已在数据库中得到广泛使用,例如:数据库会将 SQL 查询转换为关系代数的抽象语义树,具有等价关系的语义树(优化器查询改写依据的原理)代表同一个查询,因此产生的查询结果相同.本文采用同样的思想,将谓词建模为语义树,并使用语义树表征查询结果.谓词条件产生的查询结果具有唯一性,等价的谓词条件产生的查询结果相同.若缓存结果被表征的语义树之间存在关联关系,则缓存结果之间也必然存在相同的关联关系.因此,可以将缓存结果匹配问题转换为语义树匹配问题.

依据上述原则,本文将算子谓词条件转换为语义树,用语义树表征算子处理结果,并在缓存复用时通过匹配语义树的方式匹配对应缓存结果.缓存匹配主要包含语义树构建以及语义树匹配两个步骤,接下来将对两个步骤进行详细介绍.

4.1.1　语义树构建

缓存结果表示是一个非常重要的问题,关系到后续数据处理是否方便、数据是否只进行较少的处理就直接返回,直接决定了算子缓存的工作效率.文献[15,34]中分别采用规范化和树字符串的表示方式,虽然能够满足快速匹配的需求,但无法完全表达出谓词条件的语义信息.例如,对于谓词条件 E_1:$\sigma(1<A<10)(E)$ 和 E_2:$\sigma(10>A>1)(E)$ 具有相同语义,但上述两种方法却解析成不同比较操作.

为了解决上述问题,本文采用语义树来表示算子结果,并通过匹配语义树的方式来确定其代表的数据集之间的关系.

定义 1. 语义树. 语义树是一棵二叉树,用于表示谓词条件的语义关系.它是一种抽象语法树的变体,不仅包含语法结构,还包含了谓词条件的语义关系.在语义树中,叶子节点表示谓词条件中的列属性以及立即值,内部节点则表示谓词条件中的逻辑运算符、函数以及对应参数,节点之间的连接则用于表示语义关系.

语义树用于表示算子的谓词条件,也用于表征该算子的计算结果.树中每个节点表示谓词条件的不同部分,记录节点类型、节点数据以及孩子节点信息.其中,逻辑运算符是一个二元操作,对应谓词条件中的与(AND)和或(OR);比较函数是数据库列值的比较操作,包括大于(GREATER)、大于等于(GREATEROREQUAL)、等于(EQUAL)、小于(LESS)以及小于等于(LESSOREQUAL);数据提取函数作用于数据库列值,例如日期、字符函数等;参数记录函数运算涉及的列值及其数据类型.

语义树构建遵循以下步骤:首先,从字符串形式的谓词表达式中提取出关键字信息;其次,将其转换为语义树的节点.为此,本文采用了编译原理中的词法分析方法,将字符串组织为记号(TOKEN)序列(词法分析可参考附录 A),获取字符串中的逻辑运算符、比较函数、数据提取函数以及参数四类关键字;然后使用语法解析工具"上下文无关文法"解析字符流的语义信息,并将其转换成语义树节点(语法解析规则可参考附录 B).

查询 Q1:查询 1993 年折扣大于 1 且销售数量少于 25 的所有订单的销售总额

```
SELECT
  SUM(PRICE * DISCOUNT)
FROM       LINEORDER
WHERE
  TOYEAR(ORDERDATE) = 1993
AND        DISCOUNT≥1
AND        QUANTITY<25
ORDERBY    QUANTITY
```

图 5 展示了为查询 Q1(WHERE 部分)构建的语义树.首先,提取出 Filter 算子的谓词条件(左下角圆角方框),然后词法分析程序提取关键字信息(逻辑运算符、比较操作等),最后语法解析程序按照语法规则将其转换为语义树节点并最终形成右半部分的语义树.该语义树表征了 Filter 算子的计算结果.

4.1.2　语义树匹配

算子缓存中最关键的部分就是进行缓存匹配.缓存池中使用语义树来表示结果数据集,通过比较树之间的关系就能判断出其代表的结果集之间的关系.在介绍语义树匹配算法之前,首先介绍下结果集之间的匹配关系.对于 Filter 算子,假设目标查询产生的结果集为 r_x,缓存中候选结果集为 r_y,则 r_x 和 r_y 之间的匹配关系主要考虑以下三种情况:

(1)精准匹配. r_x 结果区域的谓词条件完全等价于 r_y 的谓词条件,r_x 和 r_y 具有完全相同的结果区域,即 $r_x=r_y$,$r_x\subseteq r_y$ and $r_y\subseteq r_x$;

(2)包含.候选结果集 r_y 是当前查询算子结果 r_x 的一个超集,且有更宽松的约束条件,能够直接或间接计算得出满足 r_x 谓词条件的所有结果,即 $r_x\subseteq r_y$;

(3)部分匹配.候选结果集 r_y 中包含当前查询算子结果 r_x 的部分数据,即 $r_x\cap r_y\neq\varnothing$.

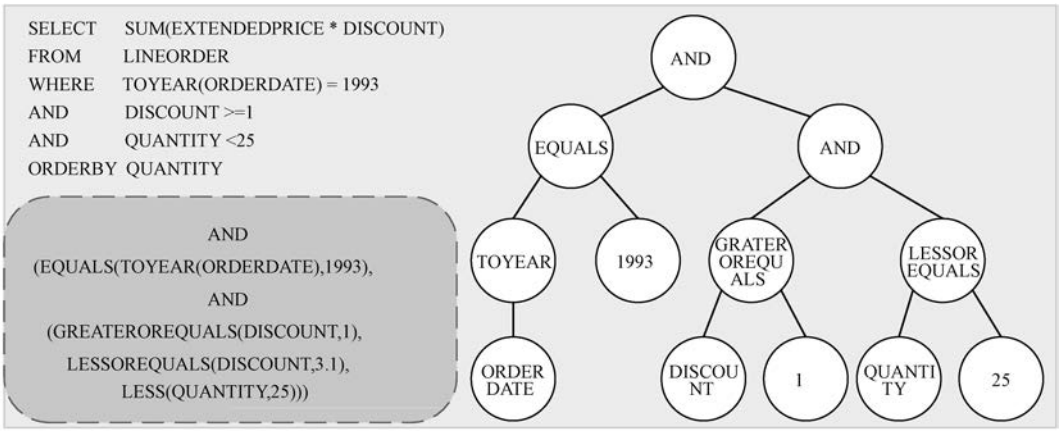

图 5 语义树构建

精准匹配是最简单的一种处理情况，不需要进一步对缓存结果进行处理，直接返回对应的缓存结果，因此无需重新计算该算子以及所有之前算子. 而对于包含和部分匹配两种情况，则需要进一步对缓存结果进行处理.

包含情况的检测相比精准匹配要复杂一些，需要满足以下条件：首先需要检测“$NULL$ 值”约束是否存在冲突；其次，检测缓存区域的(min, max)区间是否包含目标算子区域；最后，检测缓存区域的约束条件是否更宽松.

假设 Filter 算子的谓词过滤条件：

e_1:$\sigma(10 < x < 30)(E) \vee \sigma(20 < y < 50)(E)$,

e_2:$\sigma(10 < x < 30)(E) \vee \sigma(20 < y < 40)(E)$,

e_3:$\sigma(10 < x < 30)(E)$.

对应 3 条不同查询中 Filter 的计算结果. 相比 e_1,e_2 中 y 具有更小的上界，在 e_1 的基础上补充“y <40”获取 e_2 的结果，即 $e_2 \equiv \sigma(y < 40)(e_1)$，因此 $e_1 \supseteq e_2$；相比 e_1,e_3 缺失了“20<y<50”约束条件，e_1 的计算结果可以等价表示为 $e_1 \equiv \sigma(20 < y < 50)(e_3)$，因此 $e_3 \supseteq e_1$.

部分匹配需要检测缓存区域和目标算子的约束条件是否重叠，检测过程类似包含检测.

语义树匹配算法用于检测结果集之间的匹配关系，支持精准匹配和包含关系检测. 部分匹配的检测十分复杂，需要遍历所有缓存数据同时记录具有交集的缓存结果并在该结果中找出所有符合谓词条件的数据，包含数据合并、去重等操作，当缓存结果巨大时这些操作带来的开销将会严重影响查询执行性能.

语义树之间的关系遵循结果集匹配，但也有不同之处，针对语义树之间的匹配本文有如下定义.

定义 2. 语义树精准匹配. 若两棵不同语义树精准匹配，则其每个节点满足：(1)具有相同操作类型，例如都是逻辑运算符、数值等；(2)具有相同参数，例如具有相同的比较数值；(3)具有相同数量的子节点.

定义 3. 语义树包含. 假设有语义树 R_x 和 R_y，若 $R_y \supseteq R_x$，则 R_x、R_y 满足以下两个条件之一：(1) R_x 是 R_y 的子树；(2) R_x 比 R_y 具有更小范围的列值.

根据上述定义，很容易得到语义树之间的匹配规则.

规则 1. 假设存在语义树 Q、P，若 P、Q 每个节点相等或具有等价关系，并且具有相同的树结构，那么 P 和 Q 等价，即 $P \equiv Q$；

规则 2. 假设存在语义树 Q、P，若 Q 为 P 的子树，并且 P 包含 Q 中所有节点，那么 P 包含 Q，即 $Q \supseteq P$；

规则 3. 假设存在语义树 Q、P，若 P 中每个节点的比较条件大于等于 Q，那么 P 包含 Q，即 $P \supseteq Q$.

规则 1 对应缓存结果精准匹配情况，即匹配结果完全相等；规则 2、3 对应包含匹配，但缓存结果匹配和语义树匹配相反，即语义树之间存在包含关系($Q \supseteq P$)，则缓存结果之间的包含关系相反($P \supseteq Q$).

根据上述语义树的定义和匹配规则本文实现了语义树匹配算法，算法采用深度优先遍历方式遍历所有节点并进行比较，不同类型的节点有不同比较方式. 对于非数值型属性节点的比较需要值相等，而对于数值型属性则根据前文中提到的结果集匹配关系进行判断. 算法分为两部分，语义树比较以及节点比较，前者用于遍历树节点返回匹配结果，后者则是

判断每个节点之间的关系.鉴于篇幅原因,本文只展示了包含关系的算法,精准匹配的实现同包含关系类似.

算法 1.语义树比较算法.

输入:缓存结果语义树 p,目标树 q.

输出:匹配结果(布尔值)

```
1. IF q = NULL THEN
2.    RETURN true;
3. IF p = NULL && q≠NULL THEN
4.    RETURN false;
5. IF p=NULL && q≠NULL THEN
6.    IF ! NodeCompare(p,q) THEN
7.       RETURN false;
8.    FOREACH q_child : q→child DO
9.       exist←false;
10.       FOREACH p_child : p→child DO
11.          IF contain(p→child,q→child) THEN
12.             exist←true;
13.             IF ! exist THEN
14.                RETURN false;
15.    RETURN true;
```

(1)语义树比较算法

假设缓存中有候选集$\{p_1,p_2,\cdots,p_n\}$,其中,每个数据集由一棵语义树表示,算法将目标查询产生的语义树同候选集依次进行比较并返回一个最合适的匹配结果.语义树比较伪代码见算法 1.

算法 1 用于判断语义树之间是否存在包含关系.算法会在 Filter 算子处理逻辑中执行,自顶向下遍历整棵树,并在缓存池中查找匹配项.具体而言,首先进行特例判断,比如两树节点都为空或目标树为空而候选树不为空,此时说明候选树包含目标树(对应定义 3 中的条件 1),反之则必然没有包含关系(第 1～4 行).接下来处理两树节点非空的情况,首先调用 *NodeCompare* 函数来判断节点之间的关系,如果都不是叶子节点只需要简单地比对一下节点值,如果一个是叶子节点一个非叶子节点,那么必然不存在包含关系,如果都是叶子节点则进入叶子节点的判定逻辑(第 5～7 行).完成节点比较后递归调用该算法,比较所有剩余子节点(第 8～15 行).

(2)语义树节点比较算法

语义树节点比较算法用于确定节点之间的关系,不同类型的节点所比较的方式不同.语义树节点比较伪代码见算法 2.

算法 2.语义树节点比较算法.

输入:目标节点 p,候选节点 q.

输出:比较结果(布尔值)

```
1. IF p→type != q→type THEN
2.    RETURN false;
3. IF p = function && p = string THEN
4.    RETURN p→data==q→data;
5. IF p = compare THEN
6.    IF p→data=LE || LEOREQUAL THEN
7.       RETURN p_child→data<=q_child→data;
8.    IF p→data=GE || GEOREQUAL
9.       RETURN p_child→data>=q_child→data;
10.    RETURN false;
```

算法 2 用于判断节点之间是否存在包含关系.具体而言,算法会根据节点类型来选择一种比较方式,如果节点之间类型不同则说明两节点之间没有任何关系(第 1～2 行).对于数据提取函数以及字符串类型节点的比较,需要确保其值完全相同(第 3～4 行).而对于比较函数类型节点的比较,需要确定节点间值域的大小(第 5～10 行).例如,对于 LESS 或者 LESSOREQUAL 节点,实际上需要比较两节点之间哪个节点具有更小值域,反之则是比较哪个节点具有更大值域.具有更小下界或更大上界值域的节点将包含另外一个节点(对应定义 3 中的条件 2).

4.1.3　缓存复用

如前文所述,本文采用语义树来表示算子处理结果,通过匹配语义树来确定查询之间的关系.由于匹配关系有多种情况,因此在匹配到多个计算结果时需要选择一个最优结果.本文的缓存复用遵循以下策略:(1)当 Filter 和 Sort 算子同时存在时,优先使用 Sort 算子缓存;(2)优先使用精准匹配结果,并且当精准匹配时停止搜索;(3)否则,将匹配结果按数据量大小进行排序,并选用数据量最小的作为复用结果.

实际上,为了减少匹配结果消耗的时间,可以指定一个成本阈值,当算法匹配到满足阈值的结果集时便停止搜索.

4.2　代价优化器

算子缓存面临的一大问题是:缓存中的数据并不一定完全符合当前查询中算子的计算结果,该情况下复用缓存需要额外的过滤、排序等处理.如果这些额外处理的代价很高,甚至超过了缓存带来的收益,那么使用算子缓存反而会劣化查询性能.

针对上述问题本文提出了修改代价优化器,增加算子缓存的代价计算逻辑,使代价优化器自动选

择是否使用缓存,实现自适应算子缓存功能.通过引入代价优化器,可以保证在算子缓存有效时正确启用缓存,而不会由于引入缓存导致性能下降.

代价优化器的实现借鉴了 Volcano 优化器,在 Volcano 的基础上修改了成本计算模型以及搜索空间.原生 Volcano 会在所有执行路径中选择成本最小的执行路径,没有考虑将缓存添加到搜索空间.例如,对于包含 Sort 算子的查询(假设 Sort 的等价物理操作包括 InMemorySort、MergeSort 两种),Volcano 模型只会在 Sort 的等价操作中选择成本较低的操作添加到执行路径中(Volcano 的详细细节可参考文献[35,36]).为了考虑缓存带来的影响,我们在优化器搜索空间中增加了缓存复用操作(Sort 等价物理操作为 InMemorySort、MergeSort 以及 GetFromCache,其中 GetFromCache 表示 Sort 数据从缓存中获取).这样,在进行路径搜索时也会将缓存复用考虑在内.

要计算是否复用缓存,首先需要知道原始查询计划的执行成本以及复用缓存后的执行成本.由于算子缓存会复用原执行计划的一部分计算,因此只需要比较缓存算子前的计算成本和复用缓存的成本.例如,对于查询 2 而言,只需要比较 Scan、Filter 的总成本和 Sort 算子复用的成本.

如上文所述,我们增加了代价计算逻辑,分别计算原查询计划成本以及使用缓存时的成本.对于不同算子,其消耗的资源也不同,因此需要分别计算每个算子的成本.为了方便计算,本文将算子涉及的计算资源统一分为四类,并为每个算子分别计算对应的成本.这四类计算成本分别是:CPU 成本,内存成本,I/O 成本以及网络成本.成本计算类似文献[37],依赖于魔法数的设定,并且成本最终计算出来是一个数值.其中,CPU 成本根据函数调用次数计算,I/O 成本根据从磁盘加载的数据量确定,内存成本则是计算过程中的内存总消耗,网络成本计算方式类似于 I/O,根据远程数据传输量进行计算.有关统计信息可以通过元数据获取,因此,在计算成本时我们只需要设定各计算资源的权重系数(魔法数)即可.

查询 Q2:

```
SELECT    *
FROM      LINEORDER
WHERE     TOYEAR(ORDERDATE) = 1993
AND       DISCOUNT ≥ 1
AND       QUANTITY < 25
ORDERBY   DISCOUNT;
```

查询 Q3:

```
SELECT    *
FROM      LINEORDER
WHERE     TOYEAR(ORDERDATE) = 1993
AND       DISCOUNT ≥ 1
AND       QUANTITY < 25
ORDERBY   QUANTITY;
```

针对不同算子,其计算量估计都是从上述四个方面进行综合评估,区别在于每个算子具有不同负载,因此计算的侧重点不同.例如,对于 Sort 缓存复用,主要涉及 CPU、内存以及网络开销,因此其 I/O 成本为 0.假设一次元组比较操作消耗 CPU 的成本为 0.0025,排序算法平均需要比较 $n*log_2 n$(算法的时间复杂度,其中 n 表示元组个数,假设 $n=100$)次元组完成排序,因此 CPU 成本为 2.5.假设每兆内存占用成本为 0.005,每兆数据传输成本为 0.075.若计算过程中共消耗 10M 内存以及传输 10M 数据,则其内存成本和网络成本分别为 0.05 和 0.75(本文的权重系数参考 ClickHouse 配置).

根据前文所述,本文将原查询计划执行的总成本 $cost(R)$ 定义为

$$cost(R)=\sum_i c_i(m)\cdot r \tag{1}$$

其中 $c_i(m)$ 表示第 i 个算子的成本向量(m 表示数据量),记录算子的函数调用次数、返回结果大小、网络传输以及磁盘 I/O. r 是可配置的成本系数,通过调整 r 可以控制各资源的权重.

缓存命中时,算子缓存将直接从缓存池中读取数据.若非精准匹配,还需要将数据进行处理再返回,这部分开销也要计入成本.针对缓存复用成本 $cost(C)$ 本文将其定义为

$$cost(C)=cost_{match}(x)+\mathrm{cost}_{scan}(x)+R+\sum_i cost(c_i(x_i))\cdot r \tag{2}$$

$cost_{match}(x)$ 表示语义树匹配所消耗 CPU 成本,由于无法预先知道匹配次数,因此这里通常会根据经验设置一个常数;$cost_{scan}(x)$ 表示从缓存中读取结果的成本,代表网络传输成本;R 表示格式转换成本,在内存中本文采用 Parquet 格式组织数据,因此数据格式转换的 CPU 成本也要包含在内;$\sum_i cost(c_i(x))$ 表示对缓存数据进行额外过滤、排序的成本之和,算子的成本估算同前文.

为了方便理解,本文给出 Sort 算子复用实例.

查询 Q2,Q3 均包含 Sort 算子操作,Q3 可以复用 Q2 中 Sort 算子的结果,但需要按键 QUANTITY 重新排序.假设 Q2 中 Sort 算子缓存 1000 万条数据(约 500M 数据,转换成 Parquet 格式后约 100M),约 500 个块.为了方便描述,这里假设成本向量 $r=(0.01,1,2,1)$,分别对应 CPU、内存、网络传输、磁盘 I/O 成本.根据公式 2 可知复用 Sort 结果的代价为:匹配算法、将 Parquet 数据格式转换为原数据格式消耗的 CPU 成本、扫描数据的网络、内存成本以及重排序消耗的 CPU 成本.其中,匹配算法成本 $cost_{match}(x)$ 设置为常数 50(在我们的微基准测试中算法匹配成本设置为 50 有较好结果,测试结果参考附录 C);扫描数据成本 $cost_{scan}(x)=100\times2+500\times1=700$(100 代表网络传输数据量,500 代表转换格式后内存占用);$cost(c_i(x_i))$ 表示排序成本,假设这里采用快速排序,则排序成本为 $cost(c_i(x_i))=1000log1000\times0.01=30$;格式转换成本 $R=1000\times0.01=10$(1000 为函数调用次数).因此缓存复用成本 $cost(C)=50+700+30+10=790$.对于查询总成本的计算,本例中只需计算 Filter 算子和 Sort 算子的成本.根据成本系数,Filter 算子成本为 $cost(F)=500\times1+1000\times0.01+500\times1=1010$(I/O 成本、CPU 成本以及内存占用成本).因此,总成本 $cost(R)=1010+30=1040$.通过对成本比较,可以发现使用缓存的执行计划所包含的代价低于原执行计划,因此在本例中会选择使用缓存方式执行查询.

5　实验与分析

本文基于 ClickHouse 实现了 Filter 算子以及 Sort 算子缓存原型,并使用 Redis 充当外部缓存系统.本节将设计实验对缓存匹配算法、Filter 算子缓存、Sort 算子缓存进行性能测试.

5.1　实验设置

实验环境由两台机器组成,一台作为远程存储服务器,一台部署 ClickHouse 服务.其中,使用 Redis 作为远程存储服务,缓存大小设置为总数据量的 20%(压缩后数据,约 5GB),使用 Redis 默认缓存替换策略.各机器以及软件详细信息参见表 2.

实验主要采用 SSB(Star Schema Benchmark)和 TPC-H(Transaction Processing Performance)标准.SSB 由事务处理性能委员会发布的 TPC-H 标准改进而来,它在 TPC-H 的基础上将雪花模型改为星型模型,基准查询由 TPC-H 的复杂 Ad-Hoc 查询改为结构更固定的 OLAP 查询.本文主要针对 OLAP 应用场景,因此 SSB 标准更适用于本文实验.SSB 标准包含四类查询,共有 13 条查询.其中 Q1,Q2 类查询分别包含 Q1.1、Q1.2、Q1.3 以及 Q2.1、Q2.2、Q2.3 各 3 条查询.Q1 类查询均为单表带有 Filter 算子的查询,Q2 类查询均为单表带有 Filter 以及 Sort 算子的查询,符合算子缓存的应用场景,因此本次实验主要选用该两类查询.

表 2　测试环境

名称	型号(版本)
CPU	Intel(R) Xeon(R) Gold 6240R CPU@2.40 GHz
MEM	374 GB
DISK	7.4 TB
NET	Ethernet 1000 Mb/s
	Infiniband 100000 Mb/s
OS	CentOS Linux release 7.5.1804 (Core)
Kernel	Linux 3.10.0−862.el7.x86_64
ClickHouse	20.10
GCC	9.3.0
Redis	6.2.7

同时,为了证明算子缓存的适用性,我们还采用 TPC-H 进行了算子缓存的综合性能测试.

5.2　语义树匹配算法性能测试

在测试算子缓存的性能之前首先需要评估缓存匹配算法的性能.另外由于本文方案采用外部内存存储系统,所以网络时延也不可忽略.因此,本节实验会分别测试本地存储以及远程存储下算法的匹配效率.实验使用 SSB 中 Q1.1 作为目标查询,将其他查询作为匹配对象,测试不同数量下算法的匹配效率.

图 6 展示了语义树匹配算法的实验结果.结果表明算法的匹配速率大致稳定,随着匹配数目线性

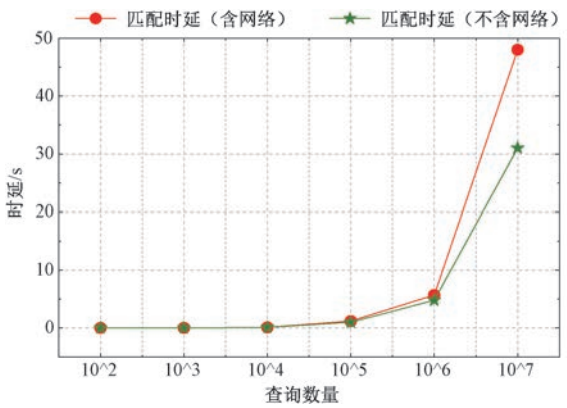

图 6　语义树匹配算法

增长.只有当匹配数量达到 10^5 级别时算法执行时延以及网络延迟急剧上升,此时算法带来的延迟将变得不可接受.在实际应用场景中,单次查询缓存的数据量通常在 MB 或 GB 级别,缓存中的查询数量大约在 20～500 之间[34,38],很难达到拐点数量,并不会产生很大的时延占比.因此使用远程缓存的方式有利于发挥远端缓存大容量的特点且不会对性能造成很大影响.

5.3 Filter 算子缓存性能测试

本节实验选用 SSB 中 Q1.1、Q1.2、Q1.3 验证 Filter 算子缓存的性能,这三条查询都是 Fliter 操作结合少量的聚合操作,可以准确地反应出 Filter 算子缓存的加速效果.实验中 Scale factor 设置为 100,生成大约 100GB 数据.同时,还测试了连续执行一组查询来模拟真实的 Ad-hoc 查询下算子缓存的效果.查询遵循以下模板,其中"XXX"表示可变参数,通过配置不同参数和修改查询列来模拟不同查询,分别记录算子缓存、查询结果缓存、物化视图以及块缓存的查询时延.

查询模板:

```
SELECT
AVG(lo_extendedprice) ,
AVG(lo_discount),
AVG(lo_tax)
FROM lineorder_flat WHERE
AND lo_shipdate>=XXX
AND lo_extendedprice>=XXX
AND lo_extendedprice<=XXX
AND lo_tax>=XXX AND lo_tax<=XXX;
```

实验主要和块缓存、物化视图以及查询结果缓存进行对比.为了确保实验具有说服力,重复多次实验并取平均值来作为最终实验结果.在对比实验中物化视图会将查询中选择性最高的谓词条件得到的结果进行物化,同时修改查询语句使查询命中物化视图.例如,对于查询模板我们将查询改写为如下所示物化视图模板.其中,view 表保存了符合谓词条件 lo_extendedprice 和 lo_tax 的结果.

物化视图模板:

```
SELECT
AVG(lo_extendedprice) ,
AVG(lo_discount),
AVG(lo_tax)
FROM view
WHERE
AND lo_shipdate>=XXX
```

图 7(a)展示了 Q1.1、Q1.2、Q1.3 的执行结果.实验开始先执行一遍查询预热缓存,然后再次执行相同查询.结果表明当缓存命中时查询结果缓存(缓存最终结果)效率最高,算子缓存次之.相比块缓存提升约 8.3 倍,比物化视图方式提升约 1.1 倍性能.本实验中的物化视图建立的中间表已经涵盖查询中所有符合谓词条件的数据,即便如此算子缓存仍然有微小优势.这是因为算子缓存直接缓存 Filter 算子的结果,不用再次执行 Filter 算子,所以能够节省大量重复计算时间.

图 7(b)展示了执行一组具有相似结构 Ad-hoc 查询时 Filter 算子的性能.本文直接选用文献[31]中的查询,查询中每个合取谓词条件包含 10%～15%的数据,共返回 8%～13%的数据.每次实验随机选取 100 条查询,查询结果取多次实验的平均值.由于这些查询都改变了参数结构,因此该应用场景下查询结果缓存无法生效.对于物化视图缓存方式,我们控制谓词条件的选择性,使缓存数据在 5GB 左右.而算子缓存由于其缓存的粒度更细,针对特定的计算步骤,因此即便查询不同也能够复用查询中的公共算子.

实验在无预热缓存的情况下进行,分别连续执行 100 条 SQL 查询.结果表明在冷缓存情况下,块缓存只会缓存少量的数据块,因此性能提升效果甚微.物化视图缓存中间结果数据,当查询命中时也能够节省部分重复计算,在本次实验中物化视图方案有超过 20 条查询缓存命中,总体查询时延较原始查询降低 29%,较比块缓存降低 16%.反观 Filter 算子缓存,随着查询不断进行,缓存复用率越来越高.在我们的实验中,100 条查询中有超过 30 条查询缓存命中,总体查询时延较原始查询降低 38%,较比块缓存降低 21%,较比物化视图方式降低 13%.

查看缓存数据时发现,执行 100 条查询后的算子缓存数据约占 4.85GB,没有超过预设内存限制.为了进一步观察随着查询次数增多,缓存数据量的变化,我们收集了查询过程中内存消耗的变化情况.图 7(c)展示了各缓存方式下的内存消耗,可以观察到,块缓存方式随着查询的执行迅速消耗内存.物化视图方案将热点数据物化成中间表(物化视图通过手动方式预先为查询热点数据建表),所以在查询执行过程中内存消耗不再变化.而算子缓存随着查询不断执行,内存消耗缓慢增加.在相同内存限制下,算子缓存略优于物化视图.

实际应用场景中,缓存通常都有内存限制,而有限的内存会极大程度上影响缓存的命中率.

因此,本文还测试了在不同内存限制下各缓存的查询命中率(查询命中率=利用缓存的查询数/总查询数).图 7(d)展示了不同内存限制条件下物化视图以及算子缓存的缓存命中情况.可以看到,在具有相同内存的情况下,块缓存命中率最高,但其查询延迟最高,说明块缓存的数据比较普适,但有效数据很少.对于查询结果缓存而言,由于实验中的查询具有不同参数,因此查询结果缓存在这种情况下几乎无法使用.相同情况下,算子缓存命中情况比物化视图方式要高.而当内存足够大时,物化视图带来的缓存效果要优于算子缓存.这是因为,物化视图主要是将查询中的子查询或部分查询进行预计算并将结果物化,是一种空间换时间的方案.相比而言,算子缓存会自动缓存那些已经过滤、排序等处理后的数据,仅消耗少量内存就可以达到相同甚至更好的缓存效果.

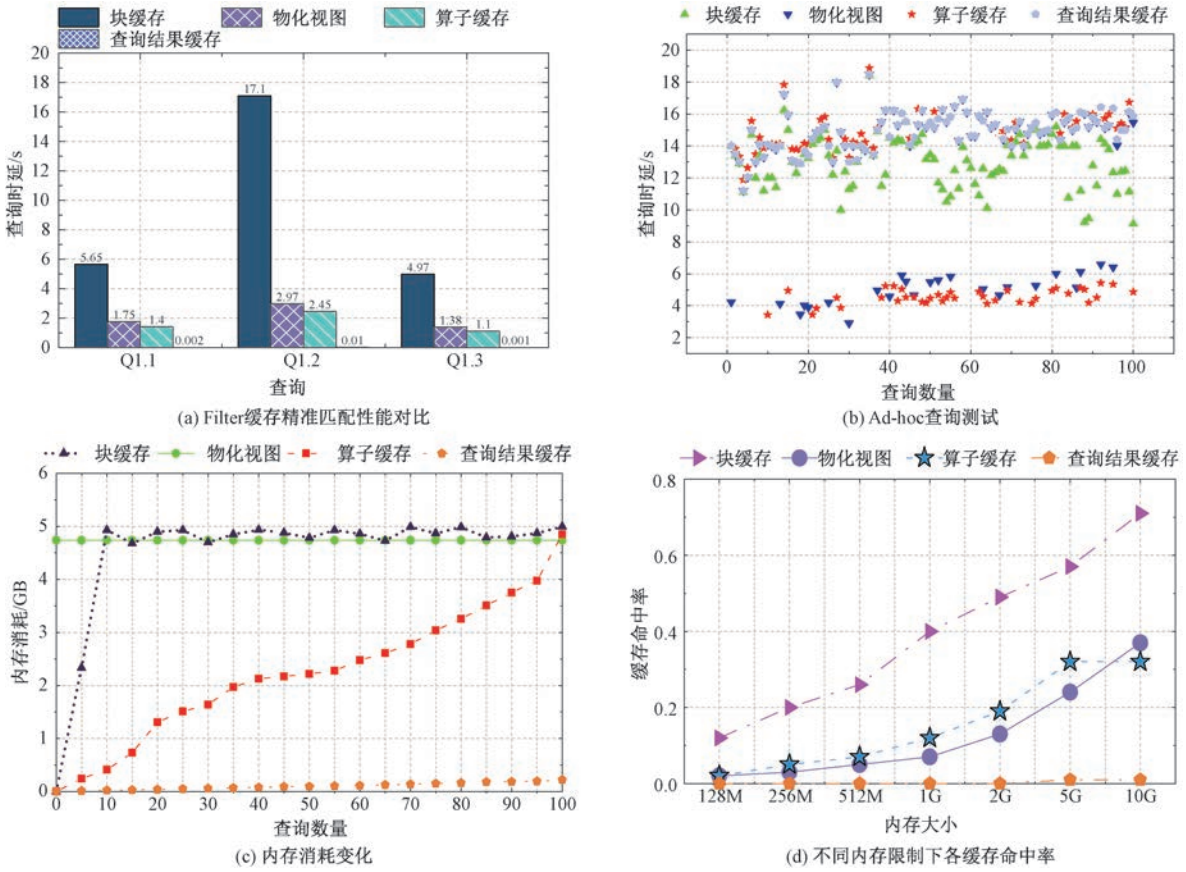

图 7　缓存性能对比

总结而言,在缓存命中时,查询结果缓存具有最高效率,但缓存复用十分受限.物化视图虽然缓解了这一问题,但其缓存中包含的计算量依旧在查询级别,并且在缓存数据时需要预先计算中间结果表.相比来说算子缓存只包含单一算子的计算量(算子级别),在查询执行期间自动缓存数据,不会有太多额外开销,并且在缓存复用上也更加灵活.相比块缓存,算子缓存在相同内存条件下能够缓存更多有效数据.因此,算子缓存能够更加高效地缓存数据,在云环境下将会发挥巨大潜能.

5.4　Sort 算子缓存性能测试

本节实验测试了 SSB 中 Q2.1、Q2.2、Q2.3(该三条查询都包含了 Sort 操作)用于验证 Sort 算子缓存的查询效率.首先,测试谓词条件完全命中的情况下 Sort 缓存带来的性能提升,对应精准匹配情况.其次,通过调整谓词参数以及排序键,使查询覆盖缓存数据从 20%至 100%,此时缓存中的结果不能直接返回,需要经过过滤、重排序处理后才能使用,对应包含情况.在这种情况下,有的查询会触发缓存功能,而有的不会触发,主要取决于优化器对于使用算子缓存收益的判断.此外,前文中已经证明查询结果缓存在缓存命中时具有最高效率,但只针对重复查询,因此后续实验将只对比物化视图和块缓存方案.

图 8(a)展示的是精准匹配下 Sort 的缓存加速性能,实验前先执行一遍查询用于预热缓存.实验结果表明,相比块缓存 Sort 算子缓存能够提升超过 5 倍的查询执行效率,相比物化视图也能提升近一倍的查询效率.这是因为 Sort 算子层级更高,能够省略更多算子的执行过程.

图 8(b)展示了包含情况的匹配,复用缓存数据需要将数据进行过滤、排序才能返回.实验结果表明,使用 Sort 算子缓存的查询时延并没有一直比关闭缓存时低,反而在缓存中包含的数据量低于 50%时比没有使用缓存性能更差.出现这种情况可以归因于算子缓存的额外操作.因为 Sort 算子缓存相对于 Filter 算子缓存来说需要更多的裁剪、反转操作,这些操作虽然都是内存操作但仍然非常耗时,证明了前文猜想的正确性.而当加入优化器之后,查询总是能够选择代价最低的执行方式,因此具有总体最佳性能.

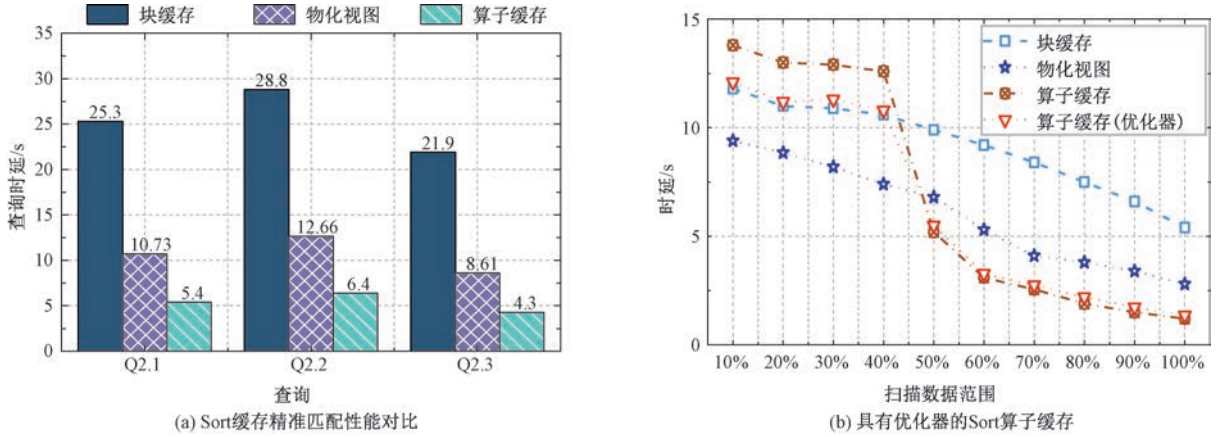

图 8 Sort 缓存性能对比

5.5 TPC-H 综合性能测试

TPC-H 是国际事务处理性能委员会(TPC,Transaction Processing Performance Council)于 1994 年制定的标准,是一款面向商品零售业务的决策支持系统测试基准,共包含 8 张表,22 个查询.通过 TPC-H 测试,可以更好体现出算子缓存的优势.表 3 给出了 22 条查询的结果(实验使用 100GB 数据量),在 ClickHouse 中部分复杂查询会出现内存限制错误,对于 ClickHouse 不支持的查询不在表中列出.

表 3 列出了算子缓存在 TPC-H 基准下的测试,共有 11 条查询.查询结果列表示查询返回的数据行数,聚合表示查询用于计算整体信息,只返回一条汇总结果.查询类型说明查询包含的算子.实验共分为两组,首先测试无缓存情况下每条查询的执行时间,执行多次并记录平均结果.其次,开启算子缓存,执行多次并记录平均结果.

实验结果表明,Sort 缓存命中时带来的收益十分明显.通过查看每条查询的执行计划可知:Project 算子(投影)在最底层,而 Sort 算子总是在 Project 算子上一步.因此,在 Sort 算子缓存命中时,后续只需要对缓存结果进行映射后就能获取最终的查询结果.Q6、Q14 只包含 Filter 算子,并且都为聚合查询.缓存命中后,还需要对缓存数据进行聚合运算.查看缓存数据可知,对于 Sort 算子,由于其处于执行计划的最底层,通常缓存的结果数据少,Filter 算子缓存的数据远远多于 Sort 算子.因此,Q6 和 Q14 的缓存收益远不如其他查询.

表 3 TPC-H 测试

查询	无缓存(秒)	算子缓存(秒)	查询结果	查询类型
Q1	39.71	1.99	<10	Sort+Filter
Q2	21482.26	27.86	<10	Sort+Filter
Q4	127.87	1.25	<10	Sort+Filter
Q6	15.33	4.66	聚合	Filter
Q11	10.85	0.18	<10	Sort+Filter
Q12	21.37	0.72	<10	Sort+Filter
Q13	75.1	0.22	<10	Sort+Filter
Q14	17.83	0.31	聚合	Filter
Q16	10.27	1.74	>20000	Sort+Filter
Q18	197.82	22.57	<1000	Sort+Filter
Q22	91.61	1.57	<10	Sort+Filter

总结而言,Sort 算子处于查询执行计划的底层,包含大部分计算,缓存的结果一般数据量很小并且十分接近最终结果;而 Filter 算子处于执行计划的较前端,缓存结果的数据量大并且还需要执行后续诸多算子才能获取最终结果.因此,Sort 算子缓存带来的收益远远大于 Filter 算子.但 Sort 算子处于更底层,因此相比 Filter 算子而言也更难复用(查询计划包含越多算子越难匹配结果).

前文中算子缓存和物化视图的对比实验中仅分析了简单查询,为了使实验更具普适性,我们还使用了 TPC-H 进行测试. 查询选用 Q2、Q4、Q11、Q13、Q16、Q18、Q22,这些查询都包含子查询以及复杂连接操作. 我们将这些查询的子查询进行物化,这些子查询涵盖连接、聚合等复杂操作,更能全面、客观地比较两种缓存方式. 图 9 展示了实验结果. 就物化视图而言,查询 Q4、Q11、Q13、Q22 的效果最好. 分析查询可知,这些查询的子查询都包含连接或聚合操作,是查询的主要瓶颈. 而将这些子查询进行物化后,查询性能得到极大提升. 即便如此,算子缓存仍然有微小的优势,足以说明算子缓存有巨大应用前景.

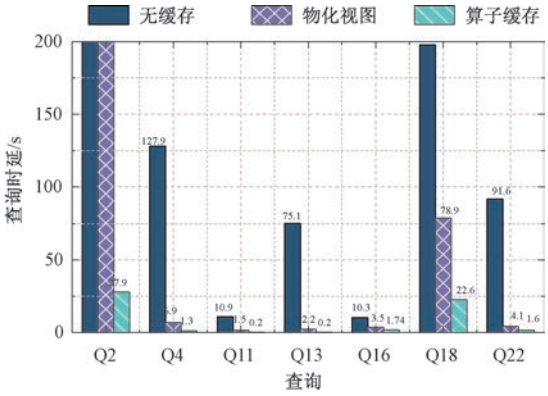

图 9　TPC-H 下各缓存方案性能对比

5.6　S3 作为存储的算子缓存性能测试

目前许多分析型数据库都会采用 S3、OSS 等云上对象存储来降低存储成本,这给分析型数据库的查询执行带来了新的挑战,需要考虑不同存储的时延开销与带宽成本. 本文的主要目的就是优化以云上存储服务为主要存储介质的分析型数据库的查询. 因此,本小节进行了面向 S3 作为存储的算子缓存性能测试.

测试采用的查询、配置与 5.3、5.4 相同,实验结果采用查询的平均延迟. 同时,出于成本考量本文采用 Minio 模拟云上 S3 的实验环境,云主机带宽限定在 200MB/s,时延在 10ms 量级.

图 10 展示了以 S3 为存储的场景中算子缓存. 紫色圆点连线代表了数据在 S3 上存储的查询时延,蓝色五角星折线代表了数据在本地 SSD 存储的查询时延,绿色三角形折线代表了算子缓存命中时查询时延(此时查询会从 Redis 中直接读取算子计算结果).

通过比较可以看出在以 S3 为存储的场景中算子缓存能带来更大的性能优势,尤其是当查询数据越大时算子缓存带来的性能提升越明显. 这是因为当数据存储在 S3 上时,查询执行需要从 S3 读取全

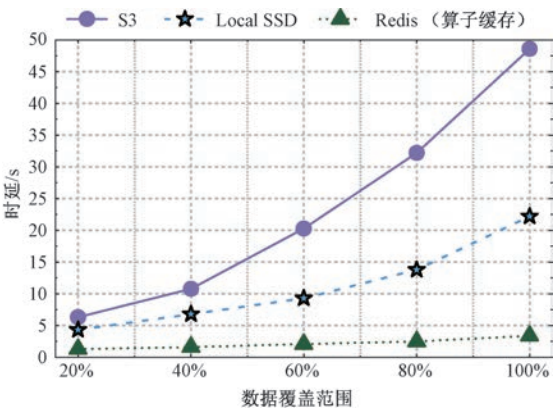

图 10　S3 下算子缓存性能测试

量数据,而由于云上存储的网络和带宽限制,此时 I/O 将成为主要瓶颈. 而缓存中的数据包含一定的计算量(经过 Filter 算子处理后的数据会提前过滤大部分部分无效数据),既能减少网络传输也能减少重复计算. 因此相比本地存储,算子缓存在云环境中能带来更加可观的性能提升.

除了上述实验外,我们还测试了 S3 下各缓存方式的加速比. 加速比是数据库中衡量查询性能的常用指标,表示优化前查询执行时延与优化后查询执行时延之比. 通过加速比可以准确地反映出查询实际的加速效果. 图 11 展示的是在以 S3 为存储的场景下各缓存方案的加速比. 从实验结果中可以看出在 S3 作为存储的场景下,算子缓存有更大的加速比,并且随着缓存数据覆盖范围的增大加速效果也呈现逐渐上升趋势,说明算子缓存在云环境的背景下能够大幅提升查询性能.

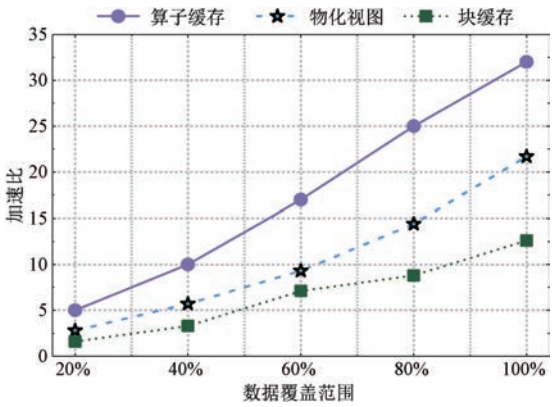

图 11　S3 下各缓存方式加速比

6　关于其他算子的讨论

本文在深入研究并总结目前国内外工业界、学术界提出的多种数据库查询优化技术的基础上,主

要针对 OLAP 型数据库的 Filter、Sort 算子的性能提升，并为此设计了算子缓存的工作方式.

我们的算子缓存的实现方法不仅只针对 Filter 和 Sort 算子，也能够推广到其他类型的算子，属于一种通用性缓存方法. 接下来我们对数据库的其他主要算子投影和连接展开讨论，并简要分析它们之间的异同.

就 Project(投影)算子而言，其缓存方式类似 Filter 算子，但在语义树构建和语义匹配上会更加简单，因为只需要在语义树中维护列属性即可. 缓存复用时同样可以借助代价优化器来判断使用缓存是否会带来正向收益. 但要注意成本模型会有略微不同，对于 Project 算子只需要关注其网络传输和磁盘 I/O 开销即可. 对于 Join(连接)算子，情况则较为复杂. 首先，语义树不仅要维护连接本身的谓词信息，还要维护在连接之前所有算子的谓词信息. 因为在连接之前任何处理存在差别都有可能引起结果变化，导致缓存失效. 其次，由于语义树要维护更多信息，在匹配时需要考虑其他算子带来的影响，这对语义树匹配算法提出更高要求. 最后，对于缓存复用也需要修改其成本模型. Join 属于计算密集型任务，因此需要重点关注其 CPU 和内存开销. 对于其他数据库算子在缓存上具有相似过程，鉴于篇幅原因本文不再继续讨论.

本文中之所以会选择这两个算子而非其他，主要有以下考量：在 OLAP 场景下过滤和排序分析十分常见；其次，Filter 算子通常处于查询计划的最前端，且很多云存储也支持 Filter 下推，因此 Filter 算子会被频繁复用；就 Sort 算子而言，其通常处于查询计划的最底端，缓存该算子回答来巨大收益. 针对其他算子(Project、Join 等). 缓存 Project 算子在以列存为主的 OLAP 数据库中意义没有 Filter 与 Sort 算子大. 因为本身以列存为主的数据库在加载数据期间就会过滤无关列，缓存投影算子所带来的收益并不高；对于 Join 算子而言，在 OLAP 应用场景中有更好处理方式(通常对涉及多表连接的查询，会将多个表合并成一张大宽表，避免在查询中执行连接操作)，并且缓存 Join 算子结果会消耗大量内存同时还不易于复用结果. 综合考虑，我们主要选取 Filter 和 Sort 算子为切入点研究算子缓存.

7 结束语

缓存是数据库领域中最为常见也是最有效的查询优化方法，一个好的缓存方案能给数据库带来巨大性能提升，尤其在云环境下存算分离的数据库中缓存结果变得更为重要. 本文结合实际应用，从缓存中包含的计算量角度出发，研究并探讨了以 Filter 和 Sort 为代表的算子缓存，解决了缓存匹配以及使用缓存可能劣化查询的问题. 整个算子缓存方案都在开源 OLAP 型数据库 ClickHouse 上实现并验证了算子缓存的性能，实验结果表明算子缓存相比块缓存、物化视图在本地 SSD 上分别有 2～9 倍以及 1.1～1.5 倍的性能提升，在云存储下分别有 5～30 倍以及 1.5～2 倍的性能提升.

综合比较，算子缓存能够更加高效地缓存数据，因此更容易复用缓存数据. 而在云环境下显得格外重要，因为复用缓存意味着能大幅减少从远程云存储中加载数据，而数据传输带来的延迟往往是云数据库的重要瓶颈.

参 考 文 献

[1] Wang Xiaoyan, Chen Jinchuan, Du Xiaoyong. Survey on OLTP application oriented data distribution in cloud computing. Chinese Journal of Computers, 2016, 39(02):253-269 (in Chinese)
(王晓燕，陈晋川，杜小勇. 云计算环境中面向 OLTP 应用的数据分布研究. 计算机学报，2016，39(02)：253-269)

[2] Amazon S3 Cloud Storage. https://aws. amazon. com/s3/ 2023,03,16

[3] Azure-Storage-Secure-cloud-storage. https://azure. microsoft. com/en-us/services/storage 2023,03,16

[4] Shi Yingjie, Meng Xiaofeng. A survey of query techniques in cloud data management systems. Chinese Journal of Computers, 2013,36(2):210-222 (in Chinese)
(史英杰，孟小峰. 云数据管理系统中查询技术研究综述. 计算机学报，2013，36(2)：210-222)

[5] Alluxio-Data Orchestration for the Cloud. https://www. alluxio. io/ 2023,03,16

[6] Dageville B, Cruanes T, Zukowski M, et al. The snowflake elastic data warehouse//Proceedings of the 2016 International Conference on Management of Data. New York, USA, 2016: 215-226

[7] Liu Q, Islam B, Governatori G. Towards an efficient rule-based framework for legal reasoning. Knowledge-Based Systems, 2021, 224: 107082

[8] Ahmad M, Qadir M A, Rahman A, et al. Enhanced query processing over semantic cache for cloud based relational databases. Journal of Ambient Intelligence and Humanized Computing, 2023, 14: 5853-5871

[9] Wei X, Hu H, Zhou X, et al. A chunk-based hash table caching method for in-memory hash joins//Proceedings of the

Web Information Systems Engineering. Amsterdam, The Netherlands, 2020: 376-389

[10] Mouna M C, Bellatreche L, Boustia N. Selecting subexpressions to materialize for dynamic large-scale workloads//Proceedings of the International Conference on Big Data Analytics and Knowledge Discovery. Vienna, Austria, 2021: 39-51

[11] Michiardi P, Carra D, Migliorini S. In-memory caching for multi-query optimization of data-intensive scalable computing workloads//Proceedings of the Workshops of the EDBT/ICDT 2019 Joint Conference (EDBT/ICDT 2019). Lisbon, Portugal, 2019: 1-8

[12] Hao Xiao-Wei, Zhang Tao, Li Lei. Optimization technology of query processing based on logic rules in semantic caching. Chinese Journal of Computers, 2005,28(7):1096-1103 (in Chinese)
(郝小卫,章陶,李磊. 基于逻辑规则的语义缓存查询处理优化技术. 计算机学报,2005,28(7):1096-1103)

[13] Ahmed R,Bello R,Witkowski A, et al. Automated generation of materialized views in oracle. Proceedings of the VLDB Endowment, 2020,13(12): 3046-3058

[14] Müller S, Nica A, Butzmann L, et al. Using object-awareness to optimize join processing in the SAP HANA aggregate cache//Proceedings of the International Conference on Extending Database Technology. Brussels, Belgium, 2015: 557-568

[15] Gosain. A, Sachdeva K. Selection of materialized views using stochastic ranking based backtracking search optimization algorithm. International Journal of System Assurance Engineering and Management, 2019,10: 801-810

[16] Goldstein J, Larson P Å. Optimizing queries using materialized views: A practical, scalable solution//Proceedings of the Special Interest Group on Management of Data. New York, USA, 2001, 30(2): 331-342

[17] Unger C, Bühmann L, Lehmann J, et al. Template-based question answering over RDF data//Proceedings of the 21st International Conference on World Wide Web. New York, USA, 2012: 639-648

[18] Han Y, Li G, Yuan H, et al. An autonomous materialized view management system with deep reinforcement learning//Proceedings of the 2021 IEEE 37th International Conference on Data Engineering (ICDE). Chania, Greece, 2021: 2159-2164

[19] Nakandala. S, Kumar A, Papakonstantinou Y. Query optimization for faster deep CNN explanations//Proceedings of the Special Interest Group on Management of Data. New York, USA, 2020: 61-68

[20] Yuan H, Li G, Feng L, et al. Automatic view generation with deep learning and reinforcement learning//2020 IEEE 36th International Conference on Data Engineering (ICDE). Texas, USA, 2020: 1501-1512

[21] Glasbergen B, Langendoen K, Abebe M, et al. Chronocache: Predictive and adaptive mid-tier query result caching//Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. New York, USA, 2020: 2391-2406

[22] Bornhövd C, Altinel M, Mohan C, et al. Adaptive database caching with DBCache. IEEE Data Engineering Bulletin, 2004, 27(2): 11-18

[23] Goldstein P A L J, Guo H, Zhou J. Mtcache: Mid-tier database caching for sql server. IEEE Data Engineering Bulletin, 2004, 1001: 35

[24] Tan K L, Goh S T, Ooi B C. Cache-on-demand: Recycling with certainty//Proceedings 17th International Conference on Data Engineering. Heidelberg, Germany, 2001: 633-640

[25] Tang D, Shang Z, Elmore A J, et al. Intermittent query processing. Proceedings of the VLDB Endowment, 2019, 12(11): 1427-1441

[26] Dursun K, Binnig C, Cetintemel U, et al. Revisiting reuse in main memory database systems//Proceedings of the 2017 ACM International Conference on Management of Data. New York, USA, 2017: 1275-1289

[27] Ivanova M G, Kersten M L, Nes N J, et al. An architecture for recycling intermediates in a column-store. ACM Transactions on Database Systems (TODS), 2010, 35(4): 1-43

[28] Jindal A, Karanasos K, Rao S, et al. Selecting subexpressions to materialize at datacenter scale. Proceedings of the VLDB Endowment, 2018, 11(7): 800-812

[29] Jindal A, Qiao S, Patel H, et al. Computation reuse in analytics job service at microsoft//Proceedings of the 2018 International Conference on Management of Data. New York, USA, 2018: 191-203

[30] Nagel F, Boncz P, Viglas S D. Recycling in pipelined query evaluation//Proceedings of the 2013 IEEE 29th International Conference on Data Engineering (ICDE). Brisbane, Australia, 2013: 338-349

[31] Perez L L, Jermaine C M. History-aware query optimization with materialized intermediate views//Proceedings of the 2014 IEEE 30th International Conference on Data Engineering. Chicago, USA, 2014: 520-531

[32] Wei Jianhao, Xia Yefeng, Gong Xueqing. Review of research on multi-query sharing technology. Journal of Software, 2021,32(10):3176-3202 (in Chinese)
(危剑豪,夏烨峰,宫学庆. 多查询共享技术研究综述. 软件学报,2021,32(10):3176-3202)

[33] Li H, Ghodsi A, Zaharia M, et al. Tachyon: Reliable, memory speed storage for cluster computing frameworks//Proceedings of the ACM Symposium on Cloud Computing. Seattle, USA, 2014: 1-15

[34] Durner D, Chandramouli B, Li Y. Crystal: A unified cache storage system for analytical databases. Proceedings of the VLDB Endowment, 2021, 14(11): 2432-2444

[35] Ding B, Chaudhuri S, Narasayya V. Bitvector-aware query optimization for decision support queries//Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. New York, USA, 2020: 2011-2026

[36] Zhan C, Su M, Wei C, et al. AnalyticDB: Real-time OLAP

database system at Alibaba cloud. Proceedings of the VLDB Endowment, 2019, 12(12): 2059-2070

[37] Chatterjee S, Jagadeesan M, Qin W, et al. Cosine: A cloud-cost optimized self-designing key-value storage engine. Proceedings of the VLDB Endowment, 2021, 15(1): 112-126

[38] Armenatzoglou N, Basu S, Bhanoori N, et al. Amazon Redshift re-invented//Proceedings of the 2022 International Conference on Management of Data. New York, USA, 2022: 2205-2217

附 录

A. 语法解析规则(产生式)定义

语法规则定义如下所示,由五部分组成:基本构成元素为 param,对应数据库列值和立即值;func 定义数据提取函数的解析规则,由函数名关键字和 param 构成;judge 定义完整的比较函数表达式(例如,age<25 或 func(year)>1993)基本组成元素为 func;hyper 定义了 AND 与 OR 操作的级联,例如条件 20<age<25 可以看成一个 hyper(由 20<age and age<25 两个 judge 构成);logic 表示一个完整的谓词条件表达式,基本组成元素为 hyper,体现在语法树结构上就是将多个逻辑条件的 AND/OR 操作拆分成多个级联的 AND/OR 操作.

```
logic: T_AND T_LEFT_BRACKET hyper
       T_RIGHT_BRACKET
       | T_OR T_LEFT_BRACKET hyper
       T_RIGHT_BRACKET;
hyper: judge T_COMMA hyper
       | judge;
judge: T_LESS T_LEFT_BRACKET func
       T_COMMAfunc T_RIGHT_BRACKET
       | T_LESS_OR_EQUAL T_LEFT
       _BRACKETfunc
       T_COMMAfunc T_RIGHT_BRACKET
       | T_EQUAL T_LEFT_BRACKETfunc T
       _COMMA
       func T_RIGHT_BRACKET
       | T_NOT_EQUAL T_LEFT_BRACKETfunc
       T_COMMA func T_RIGHT_BRACKET
       | T_GREATER T_LEFT_BRACKETfunc
       T_COMMA
       func T_RIGHT_BRACKET
       | T_GREATER_OR_EQUAL T_LEFT
       _BRACKET
       func T_COMMA func T_RIGHT_BRACKET;
func: T_TO_YEAR T_LEFT_BRACKET param
      T_RIGHT_BRACKET |param;
param: T_INT | T_FLOAT | T_PARAMETER;
```

B. 词法分析

词法分析表如表 4 所示,根据表中定义可以提取字符串形式的谓词条件中的关键字信息.

表 4 词法分析表

字符串	记号(Token)
[0-9]+\.[0-9]+	T_FLOAT
[0-9]	T_INT
"("	T_LEFT_BRACKET
")"	T_RIGHT_BRACKET
","	T_COMMA
"AND"	T_AND
"OR"	T_OR
"LESS"	T_LESS
"LESSOREQUALS"	T_ LESSOREQUALS
"EQUALS"	T_EQUALS
"GREATE"	T_ GREATE
"GREATEOREQUALS"	T_ GREATEOREQUALS
"TOYEAR"	T_ TOYEAR
[[:ALNUM]_]	T_ PARAMETER

C. 匹配次数测试

在实现 Sort 算子缓存中,由于无法提前预知缓存匹配次数,通常需要按照经验设置一个常数(与硬件相关).因此,我们增加了一个匹配次数的微基准测试,主要测试在不同常数下优化器的效果.我们给优化器设置不同常数,分别在缓存包含不同范围数据下执行查询并记录整体查询时延.基准测试结果如图 12 所示,当匹配次数设置为 10 时,匹配代价太小导致优化器总是会选择缓存方式.当匹配次数设置超过 200 时,匹配代价太大导致优化器总是会选择原执行方式.而当匹配次数设置为 50 时优化器具有较好的性能.因此,在本文中我们将匹配常数设置为常数 50.

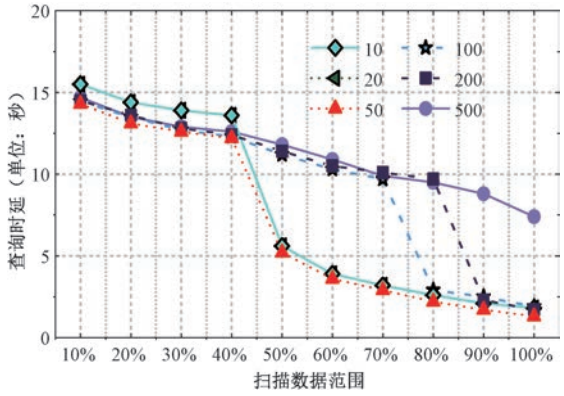

图 12 最佳匹配常数测试

CAI Wan-Li, Ph. D. candidate. His research interests include database, OALP and Query Optimization.

WANG Xin-Shuo, master. His research interests include big data, database and OLAP.

HU Hui-Qi, Ph. D., associate professor. His research interests include distributed transaction, distributed consensus theory, cloud-based database and database for new hardware.

CAI Peng, Ph. D., professor. His research interests include high performance memory transaction processing and high availability mechanism.

ZHOU Xuan, Ph. D., professor. His research interests high performance database and information retrieval.

TU Yao-Feng, Ph. D., researcher. His research interests include big data, database and machine learning.

Background

As technologies such as cloud computing and cloud storage continue to advance, database architectures are gradually shifting towards cloud-based models. In cloud architectures, databases typically adopt layered or decoupled models. However, the layered model increases data access latency, resulting in decreased OLAP query performance. Therefore, optimizing OLAP database query performance in a cloud environment presents new challenges. Considering the relatively high latency and low bandwidth of remote cloud storage, caching data becomes crucial. This paper aims to explore how to leverage caching to optimize the query performance of OLAP workloads. Currently, caching primarily includes query result caching and storage layer block caching. The former offers the highest caching efficiency but has very low cache utilization, while the latter, being a general caching method, improves cache utilization but suffers from low cache efficiency. Given this context, we propose operator caching, a caching method that lies between the two. Operator caching primarily caches the results of individual operators within a query, thereby improving both cache hit rate and cache efficiency. In the context of OLAP applications, the Filter and Sort operators are relatively common. Hence, our research focuses mainly on caching the Filter and Sort operators. This paper makes two main contributions: 1. Proposing the use of semantic trees to represent cached results, matching cached results through semantic tree matching, and introducing algorithms for constructing and matching semantic trees; 2. Designing and implementing a cost-based optimizer by incorporating operator caching cost calculations and modifying the cost optimizer to address the query performance degradation problem caused by operator caching.

Our research team has been dedicated to the study of OLAP database query optimization and has published a series of works on query optimization in high-quality journals and international conferences, such as Chinese Journal of Computer, Journal of Software, ICDE, and VLDB. This work was supported in part by the National Natural Science Foundation of China, under Grant No. 92270202, in part by Natural Science Foundation of Shanghai, under Grant No. 23ZR1418300 and ZTE Research Fund, under Grant HC-CN-20220721010.