2017 届研究生硕士学位论文

分类号：_____ 学校代码：10269

密　级：_____ 学　　号：51141500038

華東師範大學

East China Normal University

硕士学位论文

MASTER'S　DISSERTATION

论文题目：面向分布式数据库的连接查询优化

院　　系：计算机科学与软件工程学院

专　　业：软件工程

研究方向：数据库查询优化

指导教师：周傲英　教授

学位申请人：王雷

2017 年 4 月 9 日

Dissertation for master's degree in 2017

University code:10269

Student ID: 51141500038

East China Normal University

Title: Join Optimization In Distributed Database System

Department:	**School of Computer Science and Software Engineering**
Major:	**Software Engineering**
Research direction:	**Database Query Optimization**
Supervisor:	**Prof. Zhou Aoying**
Candidate:	**Wang Lei**

April，2017

华东师范大学学位论文原创性声明

郑重声明：本人呈交的学位论文《面向分布式数据库的连接查询优化》，是在华东师范大学攻读硕士/博士（请勾选）学位期间，在导师的指导下进行的研究工作及取得的研究成果。除文中已经注明引用的内容外，本论文不包含其他个人已经发表或撰写过的研究成果。对本文的研究做出重要贡献的个人和集体，均已在文中作了明确说明并表示谢意。

作者签名： 王雷 **日期：** 2017 年 6 月 12 日

华东师范大学学位论文著作权使用声明

《面向分布式数据库的连接查询优化》系本人在华东师范大学攻读学位期间在导师指导下完成的硕士/博士（请勾选）学位论文，本论文的著作权归本人所有。本人同意华东师范大学根据相关规定保留和使用此学位论文，并向主管部门和学校指定的相关机构送交学位论文的印刷版和电子版；允许学位论文进入华东师范大学图书馆及数据库被查阅、借阅；同意学校将学位论文加入全国博士、硕士学位论文共建单位数据库进行检索，将学位论文的标题和摘要汇编出版，采用影印、缩印或者其它方式合理复制学位论文。

本学位论文属于（请勾选）

（ ）1.经华东师范大学相关部门审查核定的“内部”或“涉密”学位论文*，于 年 月 日解密，解密后适用上述授权。

（ ）2.不保密，适用上述授权。

导师签名 周傲英 本人签名 王雷

2017 年 6 月 12 日

* “涉密”学位论文应是已经华东师范大学学位评定委员会办公室或保密委员会审定过的学位论文（需附获批的《华东师范大学研究生申请学位论文“涉密”审批表》方为有效），未经上述部门审定的学位论文均为公开学位论文。此声明栏不填写的，默认为公开学位论文，均适用上述授权）。

王雷硕士学位论文答辩委员会成员名单

姓名	职称	单位	备注
张蓉	教授	华东师范大学	主席
张召	副教授	华东师范大学	
翁楚良	教授	华东师范大学	
钱卫宁	教授	华东师范大学	
沙朝锋	副教授	复旦大学	

摘　要

互联网环境下，随着数据规模的不断增大，查询请求的高并发，使得数据库系统的存储与计算的横向扩展能力显得非常重要。基于分布式存储的数据库，以其良好的可扩展性受到了工业界与学术界的广泛关注，本文在分布式存储架构下对连接查询进行研究，总结影响连接查询效率的三大因素：数据的本地提取、数据的网络传输以及连接算法的执行效率。针对这三个因素，在分布式存储架构的基础上，提出了分布式数据库连接查询优化框架，有效地降低了查询响应时间，提升了用户体验。

本文工作的主要贡献如下：

1. 提出了一个有效的分布式数据库连接查询优化框架。在分布式存储的系统架构下，针对影响连接查询效率的主要因素，提出了分布式数据库连接查询优化框架，从并行度、连接算子以及半连接操作三方面对连接查询进行优化，并在开源分布式数据库 OceanBase 上实现了连接查询优化的框架。
2. 在开源分布式数据库 OceanBase 上，设计并实现了并行的嵌套循环连接、哈希连接以及半连接操作。一方面对数据进行并行的请求与处理，加快数据的本地提取，使用半连接操作有效地减少了数据的网络传输；另一方面使连接算子能够充分利用系统的计算资源，采用并行计算技术快速响应连接操作，显著地提高了连接效率。
3. 通过大量实验，验证了分布式连接查询优化框架的可行性与效率。利用开源数据库性能评测工具 Sysbench，开展了充足的实验，实验结果表明：本文提出的连接查询优化框架能有效的降低响应时间，提升查询效率。

本文提出的分布式连接查询优化框架在分布式数据库 OceanBase 上的测试结果表明：从并行度、连接算子以及半连接操作三方面出发对连接查询进行优化，可以有效地减少连接查询的响应时间，并且提升连接查询效率。同时，本文提出的分布式连接查询优化框架对其他基于分布式存储的数据库有一定的借鉴意义，

也为将来的连接查询优化工作提供了参考。

关键词：分布式存储、分布式数据库、连接查询、查询优化、优化框架

ABSTRACT

Recent development of Internet has yielded the growing volumes of data. To meet the demands of processing high concurrent query requests towards those “big data”, the capability of scaling out storage and computation in database system becomes increasingly important. Distributed database with good scalability haveattracted the attention from both industry and academy. In this paper, we study the join query optimization under a scalable distributed storage architecture. We summarize three factors that affect the query efficiency of the join: local data extraction, data transmission across network and the efficiency of the join operators. Based on the distributed storage architecture and these three factors, we design a distributed database join optimization framework, which can effectively reduce the query response time and improve the user experience.

In particular, the main contributions of this paper are threefold:

1. This paper proposes an effective optimization framework for join query under a distributed database architecture. All join queries are improved via optimizing the parallel execution, join operator and semi join operation. In addition, we implement the framework based on OceanBase, which is an open source distributed database.
2. Based on the optimization framework and the open sourced system OceanBase, we design and implement several parallel join operators including nested loop join, Hash join and semi join operation. Leveraging the parallelism is a key pillar to improve join efficiency. On the one hand, pull and process of data in parallel accelerate the local data extraction, and also effectively reduces the network transmission in join operations(e.g. semi-join). On the other hand, it makes full use of computing resources.
3. We conduct comprehensive experiments to evaluate the validation and efficiency of the distributed query optimization framework. Based on a benchmark of Sysbench, experiment results illustrate the performance of the proposed optimization framework. The experimental results indicate that the parallel join operators can effectively reduce the response time and improve the query efficiency.

The testing results of the proposed join optimization framework demonstrate that three aspects

of parallelism, connection operator and semi join operation can be used to improve the performance of the system. The proposed optimization framework and parallel join operators can effectively reduce the response time and improve the query efficiency. In the same time, the proposed join optimization framework not only equips with reference meaning for other different types of scalable distributed storage architecture systems, but also provides the reference for the join query optimization.

Keywords: *Distributed Storage, Distributed Database, Join Query, Query Optimization, Optimization Framework*

目 录

插 图

第一章 绪论

1.1 研究背景

数据库的出现与发展已有几十年的历史。从早期的网状数据库[1]、层次数据库[2]到目前主流的关系数据库[3]，广泛应用于金融、教育、政府服务、航空航天等行业与研究领域。每一阶段的数据库形态、架构设计与同时代对数据管理的需求、数据结构、数据规模以及计算机硬件技术密不可分。随着数据形态的多元化，数据规模的不断增长以及计算机硬件技术的进步，数据库的架构设计也由传统的集中式计算、共享存储[4]的模式向分布式计算、非共享存储[5]的模式转变。新型架构数据库的出现主要有以下几点因素：

1. **数据规模的不断增长**：互联网产业的发展，加速推进了各行各业的互联网化进程，推出了各种互联网应用。如传统的金融行业推出互联网金融服务、网上银行、网上借贷；新兴的互联网企业推出各种社交软件、支付平台、网络媒体、网约租车等互联网应用。**We Are Social** 公司在 2014 年关于互联网统计数据的报告[6]中指出：全球活跃的互联网用户在同年 11 月底突破了 30 亿人次。如果按照国际数据公司（International Data Corporation，IDC）2012 年的白皮书中关于全球所有人每秒产生 1.7MB 数据[7]的情况进行计算，每年在全球范围内将会产生数以 ZB（10^{21} 字节）的数据。在如此数据规模下，传统的集中式数据库已然不堪重负，数据存储的可扩展性成为了目前数据库领域的热点研究方向。
2. **数据形态的多元化**：随着信息技术的不断进步、数据来源的不断变化、应用需求的不断更新，互联网环境下，出现了大量的非结构化数据，如视频网站的多媒体音频与视频、用于位置信息服务的空间数据、互联网公司的

网站日志信息、各网站的 HTML 与 XML 文档。以上非结构化数据有如下几个特点：数据规模大、富含隐藏价值、不易对数据进行查询，针对非结构化数据的存储与管理，各互联网公司相继推出满足其业务需求的新型架构数据库，如 Apache 的顶级开源项目 Hbase [8]，是以谷歌的 BigTable 技术为基础的列式分布式数据库；以及基于分布式文件存储的 MongoDB[9]。

3. **计算机硬件技术的发展：**摩尔定律[10]指出：当原材料价格不变时，集成电路上可以容纳的电子器件数量，每隔十几个月就会增加一倍，性能翻一番。事实证明，摩尔定律准确的预估了计算机硬件技术的革新。10 年来，随着 CPU 计算能力的不断增强、存储容量不断的增加，使单台 PC（Personal Computer）服务器的价格变得更加低廉，并且依然拥有良好的性能。为了控制成本和便于扩展，互联网公司更倾向于在廉价 PC 服务器上进行数据库系统架构的设计，相比于传统数据库架构中集中式计算、共享存储的模式，分布式计算、无共享的设计模式对数据库系统的可扩展性有更好的提升。

在以上三点因素的影响下，传统的集中式数据库已无法应付所有的应用场景，新型架构的数据库被迫切需求来弥补其他应用领域的空白。在上述背景下，很多基于分布式存储的数据库油然而生，这类数据库具有良好的扩展性，其计算节点与存储节点可以分别部署在不同的 PC 服务器上。如商用的开源数据库系统 OceanBase[11]、最新发表在 SIGMOD 大会上的 Tell 原型系统[12]都采用这种分布式数据存储的架构。

基于分布式数据存储架构的数据库有以下几点特性：

1. **数据分布式存储**。数据通过一定的切分策略，分散存储于数据库集群中的各个节点。数据设有多个副本，以提高系统的可用性，每个存储节点上的数据规模由系统负责负载均衡，避免单点容量过剩与负载过重的情况。存储节点之间一般情况下不会进行数据的交互，只提供简单的计算能力，主

要负责数据的本地提取、数据过滤、网络发送、本地存储等任务。由于存储节点都是无状态的，因此当数据规模增大时，可以向数据库集群中动态的添加存储节点，达到横向扩容的目的。

2. **计算与存储分离**。分布式存储架构的数据库，设有负责查询处理与事务管理的计算节点。计算节点与存储节点分别部署在不同的服务器上，便于各自的扩展。数据库的查询任务都落在集群中的计算节点，计算节点往往拥有相对较好的硬件配置，如高主频的多核 CPU、大容量内存[13]，以确保快速的完成计算任务。
3. **节点之间通过网络进行数据的交互**。数据库集群中的存储节点与计算节点之间通过网络进行数据交互，网络带宽以及网络延迟将成为影响查询的主要因素。特别地，数据库系统采取多集群跨地域部署，网络传输开销将极大的影响查询效率，使得某些复杂查询很难有很好的响应时间。

数据库在保障基本功能的同时，查询的性能也是一项巨大的挑战，无论是传统的集中式数据库，还是新型架构下的分布式数据库，连接查询的执行效率对响应时间都有很大的影响。特别是在海量结构化数据上，减少连接查询的响应时间能够很好的提升用户体验，因此如何减少连接查询的响应时间成为数据库查询优化的主要目标。

在分布式存储架构下，影响连接查询效率的因素包括以下几个方面：

1. **本地的数据提取**，底层数据存储策略由分布式存储系统决定，如何快速的将所需的数据从服务器本地磁盘加载到内存中，发送给计算节点是提升查询效率的关键因素之一。应充分利用多核 CPU 的计算资源，采用并行技术，将本地数据的提取任务，拆分成多个子任务并行执行，提高数据提取的效率。
2. **数据的网络传输**，由于分布式存储的架构设计，数据存储节点与查询处理节点被部署到不同的服务器上。查询处理节点对数据的访问从传统数据库的磁盘访问变为对远程数据存储节点的网络请求，数据到达查询处理

节点的速度直接影响最终的响应时间。

3. **连接算法的执行效率**，当查询处理节点请求的数据到位后，将采用不同的连接算法对两张表的结果集进行连接运算。不同的连接算法有与其相适应的应用场景，选用合适的连接算法对连接查询的效率提升有显著效果，如果选择的连接算法并不适用于当前场景，连接查询的效率则会有所下降。因此恰当的连接算法对应合适的应用场景，是提升连接查询效率的关键。通过对连接算法进行适当改进，将并行技术应用于连接算法的处理流程，进一步提升连接计算的效率。

本文在分布式存储架构的基础上，结合以上几点影响连接查询效率的因素，提出了分布式数据库连接查询优化框架，并在开源的分布式数据库系统 OceanBase 上进行实现，选用 OceanBase 作为原型系统的原因有以下两点：

1. OceanBase 是一个典型的基于分布式存储架构的数据库，其数据存储节点与查询处理节点分别部署于不同的 PC 服务器上，各节点之间通过网络进行数据的访问，且其存储节点与查询处理节点都是无状态的，可根据系统负载与应用需求的变化进行动态的调整，具有良好的扩展性。
2. OceanBase 同样面临着连接查询性能不足的问题，这与分布式存储架构下，影响连接查询效率的因素密切相关。特别地，当查询中涉及的表的数据量逐渐增大时，数据的网络传输开销逐渐增加，连接计算的效率逐渐降低，如何有效的减少数据的网络传输以及提高连接计算的效率，成为 OceanBae 亟待解决的问题。

基于以上两方面的考虑，本文将分布式数据库连接查询优化框架应用于 OceanBase，以期解决 OceanBase 连接查询性能不足的问题。下面详细的介绍一下本文的工作内容。

1.2 本文工作

本文通过对分布式数据库系统中连接查询相关技术的研究，对基于分布式存

储的数据库系统架构进行解析，总结影响连接查询效率的几点主要因素。在分布式存储架构的基础上，针对影响连接查询效率的因素，提出分布式数据库连接查询优化框架，有效的减少了连接查询的响应时间。设计并实现了并行的哈希连接、循环嵌套连接以及半连接操作，提升了连接计算的效率，有效的减少了网络数据的传输。

本文在分布式存储的架构下作出了如下几点贡献：

1) **提出了分布式数据库连接查询优化框架**。包括以下几方面：
   a) 针对影响连接查询效率的因素，结合服务器计算资源，提出了从**并行度**、**连接算子**、**半连接操作**等三个方面对连接查询进行动态调整的优化策略。**并行度**主要关注利用并行技术结合系统的硬件计算资源提高数据提取速度与连接算法的执行效率；**连接算子**，为哈希连接、嵌套循环连接、排序归并连接的物理实现，主要关注连接算法的适用场景；**半连接操作**主要关注数据的网络传输开销。
   b) 在分布式存储架构的基础上，结合优化策略，提出了分布式数据库连接查询优化框架。
   c) 在开源分布式数据库 OceanBase 上对连接查询优化框架进行实现。
2) **连接算子与半连接操作在分布式数据库 OceanBase 上的并行实现**。在 OceanBase 上，设计并实现了并行的哈希连接、循环嵌套连接以及半连接操作。使用并行技术，一方面对数据进行并行的请求与处理，加快数据的本地提取速度，并使用半连接操作有效的减少数据的网络传输；另一方面使连接算子充分利用系统的计算资源，对数据进行快速的连接计算，显著的提高了连接计算的执行效率。
3) **通过实验，验证了连接查询优化框架的可行性与效率**。使用 Sysbench 的数据生成器构建实验数据集，从连接算子的执行效率以及半连接操作对查询效率的影响两方面出发，进行了全面的实验，验证了本文提出的连接查询优化框架能有效的降低响应时间，提升查询效率。

1.3 本文结构

本文章节组织如下：

第二章，主要介绍本文的相关工作，包括对分布式数据库进行概述，以及介绍了通用的查询优化过程，最后对几种常用的连接算法与半连接操作进行详细的介绍。

第三章，对基于分布式存储的数据库系统架构进行详细的介绍，分析并总结了此架构下影响连接查询效率的主要因素，然后对 OceanBase 进行系统的解析，并对 OceanBase 存储引擎 ChunkServer 与查询处理引擎 MergeServer 进行详细的介绍。

第四章，针对第三章总结的影响连接查询效率的主要因素，提出了具体的连接查询优化策略，包括三个方面：并发度、连接算子和半连接操作。最后在分布式存储架构的基础上，提出了分布式数据库连接查询优化框架，并对框架进行了详细的介绍，最后给出了连接查询优化框架中主要模块的详细设计方案，并进行了实现。

第五章，主要介绍了哈希连接、嵌套循环连接在 OceanBase 上的并行实现设计方案与具体的实现流程。然后从三个方面对并行的半连接操作进行介绍，包括：半连接操作整体的执行流程、任务线程的处理流程、数据过滤方式。

第六章，首先介绍了实验软硬件环境与实验数据设置，然后开展了充足的实验，验证连接查询框架的可行性与效率。

第七章，则是对本文工作的总结以及对未来的展望。

第二章 相关工作

分布式数据库系统，作为应对大规模数据存储与管理的有效手段，无论是在学术界还是在工业界都备受关注。本章首先介绍基于传统数据库搭建的分布式数据库系统 Mysql Sharding 以及典型的 SQL-on-Hadoop 数据库 Trafodion[14]，并分析其连接查询技术的优缺点。然后介绍通用的查询优化过程，在下文中，将会涉及到对通用的查询优化过程的改进。最后对几种常用的连接算法以及半连接操作进行详细的介绍，分析每种连接算法的优缺点以及适用场景。通过本章内容的介绍，可以对基于分布式存储架构的数据库系统与数据库领域常用的查询优化手段进行简单的了解。

2.1 分布式数据库

2.1.1 分布式数据库简介

分布式数据库[15]是一种逻辑上完整、物理上分散的数据库系统：

1. **逻辑上完整**，体现在其有与传统关系数据库类似的完整的体系结构[16]，如对外提供统一的查询接口，内部管理层有由 SQL 解析器（SQL Parser）、查询优化器（SQL Optimizer）、缓存（Cache）等组成的查询引擎[17]和各个功能模块，存储层有存储引擎[18]和存放数据的文件系统[19]，通过以上各组件的协同合作，组成了完整的数据库系统；
2. **物理上分散**，主要体现在其各个功能组件分散在不同的节点上，各个节点由网络进行连接，并由主控节点负责管理和控制系统的稳定运行。

如图 2.1 所示为基础的分布式数据库架构，其中客户端（Client）通过应用层

服务器（Application Servers）向主数据库（DBServer Master）发送请求并得到响应，DBserver Master 与备数据库（DBServer Slave）之间则通过 Zookeeper Cluster[20]进行选主，保证高可用。DBServer 上包含了连接池（Connection Pool）、管理系统与工具（Management Service & Utilities）、查询处理引擎（Query Engine）、存储引擎（Storage Engine）等组件。

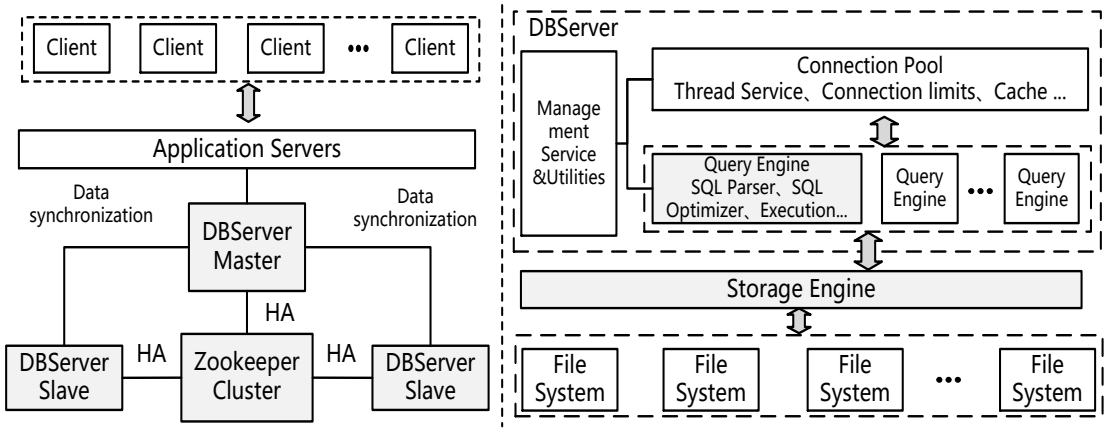


图 2.1 分布式数据库整体架构示意图

分布式数据库具有以下两种特性[21]：

1. **数据的分布性**，与传统数据库的数据集中式存储不同之处在于，分布式数据库的数据存储也是分布式的，即数据存放在不同的物理节点上，各个节点通过网络连接，优点是可以根据存储数据的总量动态地调整集群中数据存放节点的数量。
2. **逻辑的关联性**，也就是管理的集中性，为了确保数据的一致性与完整性，需要对集群中各个节点进行统一管理，时刻掌集群中各个节点的状态变化，在个别节点出现故障时能及时地进行应急处理来保证数据库的正常运行。

分布式数据库系统是现有数据库系统技术与计算机网络[22]相结合的产物，其具有建设规模大，硬件成本低，海量数据存储等特点。满足指定业务需求并特定设计的分布式数据库系统，对于互联网公司有着重大的现实意义，对我国推进“互联网+”计划具有重大的战略意义。

2.1.2 基于分布式存储架构的数据库简介

随着数据量不断的增加，读操作与写操作给传统数据库带来了巨大的压力。为了对传统关系数据库进行扩展，缓解数据存储的压力，常用的做法是先在应用层对数据进行切分，分布式存储在多个数据库实体中，然后在这些数据库实体之上进行封装，这层封装称之为数据库中间层[23]。一方面对应用层隐藏具体的存储细节，另一方面通过中间层对数据库实体资源进行统一的分配与调度。这种基于多个数据库实体存储数据，并在中间层进行统一服务的模型称之为分区数据库体系[24]，如图 2.2 所示。

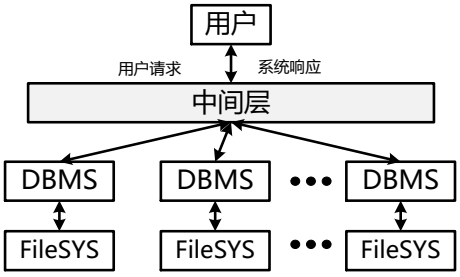


图 2.2 分区数据库体系

MySQL Sharding[25]为分区数据库体系的典型代表，也具有典型的分布式数据存储架构，图 2.3 所示为 MySQL Sharding 的架构图：

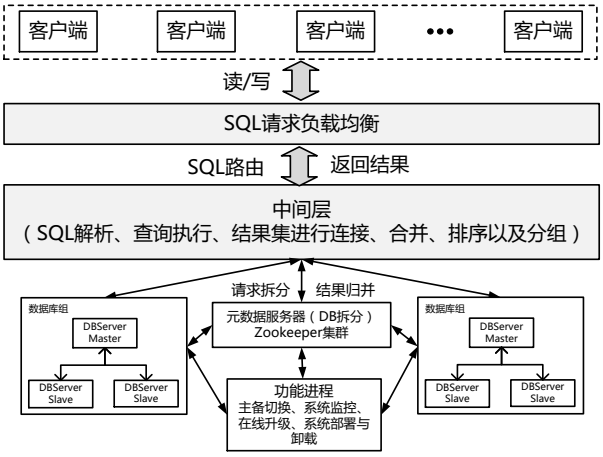


图 2.3 MySQL Sharding 架构

主要包括中间层、数据库集群、元数据服务器、功能进程等组件，下面分别对这些组件进行简单介绍：

1. **中间层**：解析客户端的 SQL 请求，将请求转发到数据库集群，最后对数据库集群返回的结果集进行合并、排序和分组等处理，并将结果返回客户端。
2. **数据库集群**：包含多个数据库组，每个数据库组中包含多个数据库实体，数据库实体中存放着应用层按一定规则所切分的数据。一般情况下，数据库组中包含主库与备库两种角色，主库负责所有写请求以及强一致读请求，并将相应操作以 BinLog[26]的形式同步到备库。
3. **元数据服务器**：主要负责维护数据的拆分规则以及用于数据库组内数据库实体之间进行选主，数据库组内通过主备同步的方式使得数据分片存在多个副本来进行容灾，因此当主库无法提供服务时，元数据服务器从新从备库中选择一个成为主库继续提供服务。数据库组内部实现高可用的同时，元数据库服务器同样使用 Zookeeper 集群来实现高可用。
4. **功能进程**：功能进程部署在数据库集群中的数据库实体上，作为常驻进程实现了诸如：系统监控、主备切换、系统部署与卸载、在线升级等功能。

在 MySql Sharding 的架构设计中，中间层主要负责解析客户端的 SQL 请求，并将数据请求转发到数据库集群，最后数据库集群将数据通过网络传输回中间层，中间层对返回的结果集进行合并、排序以及分组等操作。

MySQL Sharding 的优点在于对用户的服务是透明的，并可以存储海量的数据。缺点则是需要集群管理员（DBA）与开发人员针对应用层的业务逻辑来对数据进行人工切分，工作量巨大。特别当业务逻辑改变时，底层数据存储策略也要相应的发生改变，维护成本高，不易于扩展，并且不同分库之间无法执行连接操作。本文后续介绍的基于分布式存储架构的数据库，具有良好的扩展性，可以分布式存储海量的数据，存储节点与计算节点均为无状态节点，可以根据需求的变化动态的进行增加与删除。下面介绍另一种基于分布式存储的数据库 Trafodion。

Trafodion 是一个典型的 SQL-on-Hadoop 数据库，也是基于分布式存储架构的关系型数据库，它不仅支持完整的 ANSI SQL，还能保证事务的 ACID[27,28]（Atomicity、Consistency、Isolation、Durability）属性，和传传统数据库的不同之处在于，它利用 Hadoop 作为底层存储，使数据存储节点拥有良好的横向扩展能力。Trafodion 利用 Hbase 的扩展性，通过增加普通 PC 服务器，就可以很好的提高系统的计算与存储能力。

值得一提的是，Trafodion 的查询优化器拥有良好的优化效果，可以快速的给出最优的查询计划。Trafodion 的优化器是一个采用 Cascades 框架[29]的基于成本的优化器，并且 Cascades 框架的便于扩展，使得相关技术人员可以添加新的规则来应付某些特定的应用场景。优化器通过各种规则生成等价的查询计划，并通过自顶向下的搜索方法，利用 branch-and-bound 算法[30]将不需要深入的分支减掉。Trafodion 优化器通过多年总结出来的经验式规则进一步的缩小搜索空间，提高优化器的效率。下面对传统的查询优化过程进行详细的介绍，并分析其所面临的的问题。

2.2 查询优化过程

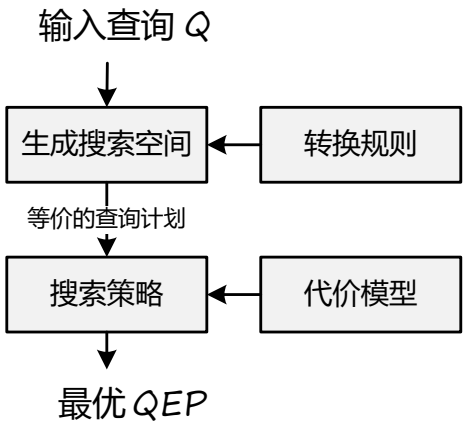


图 2.4 查询优化过程

查询优化过程是指生成查询执行计划（Query Execution Plan，QEP）[31]的过

程，其中 QEP 应使目标函数最小化，分布式环境下目标函数取值为查询执行所需的时间，本文主要关注连接操作。数据库的查询优化器[32]由三部分组成：搜索空间、代价模型[33]、搜索策略，如图 2.4 所示。

1. **搜索空间**：查询请求的可选执行计划的集合，这些执行计划可以产生相同的结果，每个查询计划特有的执行顺序，以及各种操作的不同实现方式，导致了执行计划的性能也有所不同。执行计划一般被抽象成一颗运算树，树上的节点是操作，树的形状决定了操作的执行顺序，这些运算树是查询请求经过转换规则产生的，这些查询运算树是等价的，因为它们可以产生相同的结果集。
2. **代价模型**：查询优化器的代价模型包括代价函数、数据统计信息、中间结果集预估工具等，执行代价的主要衡量标准为查询的执行时间。
3. **搜索策略**：使用代价模型来探测搜索空间中的执行计划，通过动态规划法或者模拟退火策略过滤不会产生最优解的执行计划，对需要检查的计划根据代价模型预估其响应时间，综合考虑所有执行计划，最终得出最佳执行计划。

其中搜索空间阶段不需要考虑数据库系统是集中式还是分布式，因为执行计划是按照一定的转换规则生成的，转换规则在任何情况下均是适用的。而代价模型与搜索策略则需要根据系统环境的不同，对细节进行相应的处理，因为分布式情况下数据库信息的统计以及代价函数的设置都需要考虑集群的部署与系统的设计架构，不同的系统设计架构也会导致搜索策略的改变，提高搜索策略的实施难度。

查询处理中的各种操作中，连接操作的关注度和复杂程度最高，原因在于连接操作往往涉及多张表之间的连接操作，随着表数量的增加，搜索空间中执行计划的数量呈爆炸性增长，搜索策略会浪费大量的时间用于无效执行计划的筛选，导致查询优化时间增加。而如果不对连接操作进行优化，最终的执行计划在执行过程中可能产生更为庞大的中间结果集，虽然减少了优化时间，但是增加了执行时间，并会导致最终的结果集大小是无法估计的。连接查询的难度在分布式环境

下也有所增加，分布式环境下，数据往往都是分散存储于集群中的各个服务器上，数据的访问可能是跨机房、跨地域的，这种情况下数据的传输开销将会对最终的响应时间产生很大的影响，本文接下来会介绍分布式环境下用于减少网络数据传输的常用方法：半连接操作。

2.3 连接算法与半连接操作介绍

数据库常用的连接算法主要包括：嵌套循环连接（NestLoop-Join）[34]、哈希连接（Hash-Join）[35]、排序归并连接（Merge-Join）[36]。三种算法都有其特定的应用场景，查询优化器的一个重要职责就是判断采用哪种连接算法，提升查询的效率。下面分别介绍一下这三种连接算法的原理与适用场景。

2.3.1 嵌套循环连接

嵌套循环连接是传统关系数据库常用的连接算法之一，原因在于其逻辑清晰、实现简单，并且最终结果集以 Pipeline[37]的方式返回客户端，而无需等待所有结果计算完成。基本的嵌套循环连接的伪代码与步骤如图 2.5 所示，假设关系 R 和 S 都是非空的。

```
For S 中的每个元组 s Do
    For R 中的每个元组 r Do
        IF r 与 s 满足连接条件，并形成新的元组 t Then
            Output t
```

图 2.5 嵌套循环连接伪代码

1. 首先由查询优化器决定哪张表作为驱动表，假设选择 S 表作为驱动表，则 S 表用于外层循环，R 用于内层循环。
2. 取出 S 表中第一条元组与 R 表中的所有元组判断是否符合连接条件，如果符合连接条件则组成新的元组 t 响应客户端，否则继续取出 S 表中第

二条元组，重复上一动作，直至 S 表中的数据都遍历完成。

以上所示为磁盘环境下基于元组的嵌套循环连接算法，一般情况下选择数据量较小的表作为驱动表用于外层循环，另一张表则称之为被连接表，用于内层循环。此外还有诸如基于块的嵌套循环连接算法以及基于索引的嵌套循环连接算法，基于块的嵌套循环连接算法旨在减少磁盘访问的次数，而基于索引的嵌套循环连接算法则进一步减少对表数据全量扫描的代价。

嵌套循环连接相比于下面将要介绍的排序归并连接的优点在于不用对数据进行排序，当数据量庞大的时候这个优点尤为明显。但是相比于哈希连接，嵌套循环连接中涉及到的两张表的每条记录两两之间都要进行判断是否符合连接条件，数据规模越大，则占用服务器的计算资源越多。而传统的单机数据库计算资源有限，虽然可以将数据规模较大的表拆分成几部分分别与驱动表进行嵌套循环连接，以达到并行的效果，然而在当时有限的计算资源下，还是显得有些杯水车薪。

总结来讲，当需要连接的两表数据规模较大的情况下，使用嵌套循环连接可以通过避免对数据进行排序来减少响应时间，然而数据规模的增加会对连接操作施加计算方面的压力，被连接表中的连接条件列是否有索引、是否为主键也会对数据的本地提取速度产生影响。因此嵌套循环连接的适用场景应该包含两个主要因素：

1. 驱动表的数据规模不能太大或者是过滤后的结果集非常小；
2. 被连接表中的连接条件列应该是主键或者有索引，这也是主流的商业数据库 Oracle 与 DB2 在判断是否适用嵌套循环连接时的重点考虑因素[38]。

2.3.2 哈希连接

哈希连接主要用于等值连接查询，目前主流的哈希连接分为三个阶段：分区阶段、索引表构建阶段以及探查阶段，并将其称之为有分区的哈希连接[39]。

1. 分区阶段的引入，一方面原因在于内存的不足，另一方面在于查询优化器

可以根据服务器硬件的参数，如 Cache 的大小和翻译后备缓冲器（Translation Lookaside Buffer，TLB）[40]的大小，在分区阶段决定分区的个数，降低 Cache 与 TLB 的缺失比率，加快哈希表的构建速度；

2. **哈希表构建阶段**根据分区阶段产生的分区依次构建每个分区上的哈希表；
3. **探查阶段**则利用分区阶段相同的哈希函数对另一张表进行分区操作，然后依次与对应分区上的哈希表进行匹配。

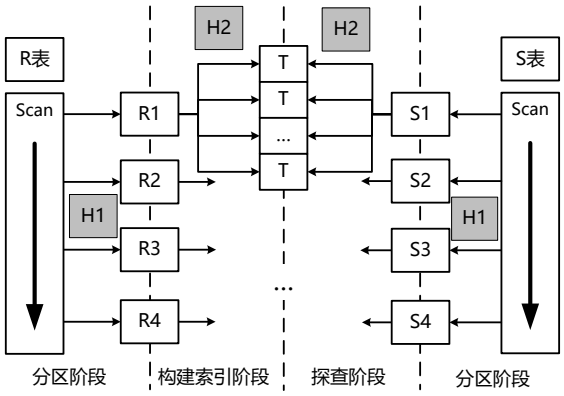


图 2.6 有分区的哈希连接

如图 2.6 所示，有两张表 R 表与 S 表，分区阶段中 R 表被哈希函数 H1 切分成 R1、R2、R3、R4 四个分区，针对每个分区使用哈希函数 H2 构建各个分区上的哈希表，S 表在分区阶段同样使用哈希函数 H1 将数据切分成 S1、S2、S3、S4 四个分区；探查阶段中，对 S 表的各个数据分区使用与 R 表相同的哈希函数 H2 进行数据的探查，完成数据的匹配。

有分区的哈希连接还包括 Radix 连接与并行 Radix 连接[41]两种算法：

1. Radix 连接目的是减少 TLB 的访问缺失，TLB 是 Linux 操作系统中 CPU 经常访问的硬件之一，存储着内存页的虚拟地址到物理地址的映射关系，这些映射关系在 TLB 中称为 TLB 实体。为了减少 TLB 的访问缺失，应该使分区的个数控制在 TLB 实体个数的范围之内。基于以上分析，Radix 连接算法在分区阶段多加了一层分区，在第一层分区的基础上再次进行

分区，使分区个数与 TLB 实体的个数保持一致。

2. 并行 Radix 连接是基于多核 CPU 架构提出的多线程分区技术，来提高分区阶段的速度，但是同时也面临着数据同步的问题，因为在有多核共享缓存的情况下，会造成缓存一致性的缺失，有些核心可能会拿到错误的数据。总体而言，Radix 连接与并行 Radix 连接算法都考虑了对硬件的使用情况，使硬件资源与数据库系统有机融合在一起。

Oracle 的连接算法[42]在 7.3 版本以前主要以排序归并连接与嵌套循环连接为主，自 7.3 版本之后开始使用哈希连接在某些应用场景下来替代以上两种连接算法，进一步提高连接查询的效率。然而哈希连接对 CPU 的消耗也是个不容忽视的问题，CPU 资源的消耗主要体现在临时哈希表的创建以及哈希计算上，因此当系统负载过高，CPU 资源不足时应酌情使用哈希连接。

下面为哈希连接的优缺点分析：

1. 哈希连接的优点是当处于连接状态的两张表中缺少可用的索引时，其整体性能要优于嵌套循环连接与排序归并连接，原因在于嵌套循环连接中被连接表的数据在有索引的情况下是通过检索 B 树[43]来获取，而哈希连接则是通过扫描全表然后通过内存中的哈希表来进行匹配，这就避免了嵌套循环连接中大量随机读的问题。相比于排序归并连接，则是减少一次排序。
2. 哈希连接的缺点有两点：一是只能用于等值连接；二是对于 CPU 资源的消耗比较严重。因此如果使用哈希连接，应该满足两个基本要求：连接中的两张表缺少可用的索引、当前数据库系统负载不高。

2.3.3 排序归并连接

排序归并连接是传统关系数据库系统中使用最为频繁的连接算法，内存不足时使用两阶段多路归并排序算法对数据进行排序，内存充足则使用内排算法，然后对排好序的两张表使用归并连接算法计算，得出最终结果集。以基于内排算法

的排序归并连接为例，其流程如图 2.7-1 所示，首先 R 表与 S 表的数据经过排序后，通过归并连接算法的计算，将最终结果集返回给上一层调用关系。图 2.7-2 中所示为简单的归并连接示例，R 表与 S 表的数据排好顺序后，依次按升序进行连接操作，如果满足连接条件就形成新的元组，直到两张表中其中一张表的数据全部处理完毕。

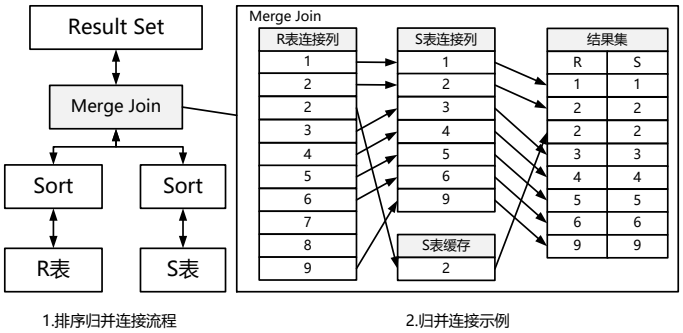


图 2.7 排序归并连接

排序归并连接算法对每张表排好序后的数据只扫描一次，当其中一张表扫描结束后，算法也即刻终止，无需对另一张表剩余的数据继续进行扫描，因此排序归并连接算法的效率与表的中间结果集大小密切相关。制约排序归并连接算法效率的因素主要有两方面：内排算法的性能、两张表在连接属性上的重复值数量。

在图 2.7-2 的例子中，当 R 表有重复值的情况下，需要额外一张表来维护重复的记录，如果重复值较多，会额外增加内存的开销。以上所提到两种制约条件在数据量增加的情况下对算法的效率影响尤为突出。

综上所述，归并排序连接较之嵌套循环连接的优点在于对每张表都只扫描一次，减少了磁盘 I/O 时间开销[44]。缺点在于需要对两张表进行排序，当数据量增加时，算法的效率会显著下降。

2.3.4 半连接操作

半连接（Semi-Join）操作[45]的基本思想是减少无用数据（不会产生新的连

接关系的数据）的网络传输，降低数据的传输代价，以下为半连接操作的基本定义。

假设有关系 R 和关系 S，则其在属性 A 上的半连接操作有公式：

$$R \ltimes S = \pi_R(R \bowtie S) = R \bowtie \pi_A(S)$$

其中 R 位于服务器 1，S 位于服务器 2，属性 A 为投影属性，半连接 $R \ltimes S$ 的流程如下：

1. 首先在服务器 2 上得到关系 S 在属性 A 上的投影$\pi_A(S)$；
2. 然后将投影结果集$\pi_A(S)$通过网络传输至服务器 1 上与关系 R 进行连接判断得到新的关系R'；
3. 最后将R'从服务器 1 发送至服务器 2 与关系 S 进行连接得到最终的结果集。

以应用广泛的 SDD-1[46]算法为例，该算法利用半连接的原理将通信代价降到最低，优化思想来源于一个分布式查询处理算法，名为“Hill-Climbing Algorithm”[47]。 Hill-Climbing 算法属于贪心算法的一种，这类算法的思想是从一个初始的可行方案着手，然后迭代的加以改进，计算当前收益，并使得收益最高，最终得到最优解决方案。Hill-Climbing 算法也存在一定的问题，一方面：算法在初始阶段可能会过滤掉那些初始代价较高，但是最后能够得到很好的整体收益的计划，另一方面：算法可能会对局部代价最小的方案进行深层次的优化，继而得不到全局最优的解决方案。之后 SDD-1 使用半连接操作对 Hill-Climbing 算法进行了改进，主要包括两点：1.使用半连接操作代替直接连接，2.使用以全部的通信时间来表达的目标函数。SDD-1 算法忽略了数据的本地处理时间和消息的传递时间，因为分布式环境下通信时间占主导地位。

在某些场景下，半连接操作可以有效的减少数据的网络传输，降低查询的响应时间，本文后续会在开源分布式数据库 OceanBase 上进行半连接操作的并行实现，有效的降低数据的网络传输开销。

第三章 基于分布式存储的数据库架构

第二章对基于分布式存储架构的数据库系统 Mysql Sharding 以及 Trafodion 进行了简单的介绍，并对通用的查询优化过程以及常用的几种连接算法与半连接操作进行了详细描述，分析了每种连接算法的优缺点与适用场景。接下来，本文将对基于分布式存储的数据库架构进行详细的解析，首先介绍系统的基础架构，然后对架构中涉及的各个功能模块所负责的具体任务进行详细的描述，通过对架构模型的进一步精简，总结影响连接查询效率的几点主要因素：数据的本地提取、连接计算的效率和数据的网络传输。最后对本文的原型系统 OceanBase 进行详细的介绍，内容包括：OceanBase 的系统架构、OceanBase 的存储引擎 ChunkServer 和 OceanBase 的查询处理引擎 MergeServer，结合具体的实现情况，分析 OceanBase 连接查询方面存在的问题。

3.1 系统架构

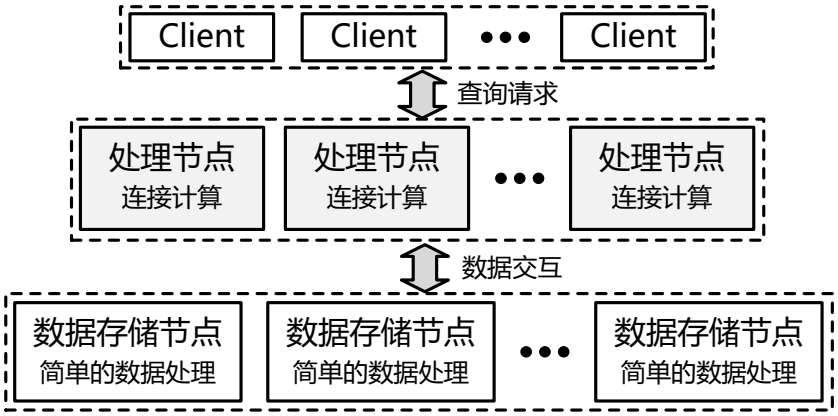


图 3.1 基于分布式存储的数据库架构

图 3.1 所示为基于分布式存储的数据库架构，其中所有复杂的连接计算任务

都由处理节点负责，处理节点相当于数据库的查询处理引擎，其主要负责数据请求与数据的处理工作，底层的数据存放节点只负责简单的数据提取与发送。一般情况下处理节点会被部署到一台性能较好的服务器上，性能较好主要体现在超大的内存（目前最大可达 24TB）、读写速度快的硬盘（如 SSD）、多核心 CPU 以及万兆网卡，这样的硬件配置下使得单台服务器有很强的数据处理能力。

通常，查询处理引擎会为每个客户端分配一个服务线程，每个服务线程单独为用户提供查询请求服务，服务线程首先解析 SQL 语句，然后制定逻辑计划，根据一定的优化策略进行逻辑计划的调优，确定最终执行的物理计划，交由查询执行器来执行。分布式数据库系统中，所有查询处理引擎共同享有所有存储节点，底层数据可以被所有查询处理引擎访问。因此在一个分布式数据库集群中，会部署很多查询处理引擎来增加查询请求的并发量，因为单台服务器的硬件资源有限，则要求查询处理引擎是无状态的，可以在任意时刻上线或者下线。图 3.1 中的处理节点在分布式数据库集群中可能部署有很多个，之所以称之为处理节点是指其制定的物理执行计划无法进行拆分并由其他服务器的查询处理引擎分担查询任务，无论请求中的数据位于集群中的哪些服务器上，最后都需要将数据通过网络传输至处理节点进行连接计算，这是本文对分布式存储架构下的计算节点做出的限制。

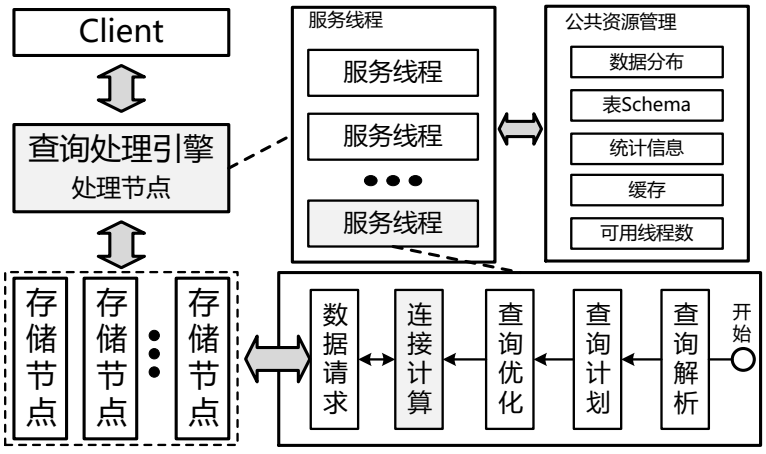


图 3.2 基于分布式存储的数据库架构细节

图 3.2 是更为细节的基于分布式存储的数据库架构，上文提到过在一个分布式数据库集群中可能会部署很多的查询处理引擎以增加并发。每个查询处理引擎也会有很多功能线程支撑起内部的处理逻辑，其中服务线程则是最为重要的一类，它们可以单独为用户提供查询功能。服务线程包含了 SQL 解析、查询优化、连接计算、数据请求等模块。为了充分利用服务器的计算资源：多核 CPU，查询优化模块与数据请求模块，要根据连接计算与数据请求任务的多少，动态的分配系统的计算资源。在本文提到的分布式存储架构中，查询处理引擎无法将查询计划分发给集群中的其他查询处理引擎，进行并行的处理，因此需要充分利用本地服务器多核 CPU 的计算资源，并行处理更多的计算任务与数据处理任务。

图 3.2 中，**查询优化模块**需要根据**公共资源管理模块**提供的数据分布、统计信息、缓存大小和可用线程数等信息，确定数据请求与连接计算的并行度，制定较优的执行计划，确保充分利用硬件的计算资源，快速的响应用户。**查询优化模块**利用统计信息预测结果集的大小，**连接模块**根据当前预测的结果集大小，向公共资源申请相应数量的线程并行完成连接计算任务，**数据请求模块**根据查询优化制定的数据提取方式，向数据存储节点发送多个请求任务，然后向公共资源申请相应个数的线程进行数据的接收与处理任务。

数据的提取分为两种情况：

1. 根据表的主键来获取数据，通常情况下，主键列数据都会设有索引，可以使用索引结构快速的定位数据，因此不需要发送多个请求来获取数据。发送多个请求的目的是为了减少数据存储节点等待数据就位的时间，通常有多少个请求任务，存储节点就会分配相应个数的线程进行本地数据的提取，多个请求并行执行，数据就并行传回数据请求模块，减少数据的提取时间。
2. 需要提取数据的表上没有任何过滤条件，或者条件为非主键、非索引列。在这种情况下，存储节点会将表的所有数据或者部分数据从磁盘扫描进内存，本文称之为范围扫描。而此时就需要数据请求模块发送多个请求任务，使存储节点并行的获取硬盘上的数据，减少数据本地提取的时间。

综上所述，基于分布式存储架构的数据库的优点在于其良好的扩展性，根据需求对计算节点与存储节点进行不同程度的扩展；缺点在于查询处理引擎硬件压力较大，主要体现在 CPU 的消耗上，为了提高并行度加快数据提取的速度与连接计算的效率，就需要充分利用服务器 CPU 多核的资源，其次要求数据库有较强的数据统计功能来帮助预测硬件资源的使用情况，以此来调整后续查询的执行策略。

3.2 影响连接查询效率的因素

接下来本文对分布式存储架构进一步精简，来分析影响连接查询效率的主要因素。如图 3.3 所示：

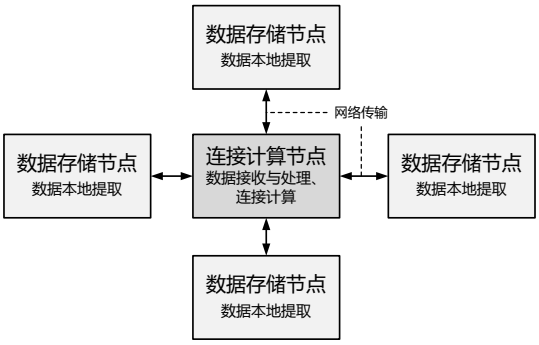


图 3.3 分布式存储架构的精简模型

连接计算节点主要有两方面的任务：**数据接收与处理**、**连接计算**，数据存储节点主要负责**数据本地提取**以及其他一些简单的数据过滤任务。整个连接查询处理的过程中主要包括四个任务：**数据接收与处理**、**连接计算**、**数据本地提取**、**数据网络传输**。查询执行流程如下：

1. 连接计算节点向数据存储节点发送数据请求；
2. 数据存储节点进行数据本地提取，由网络传输回连接计算节点；
3. 连接计算节点等到数据全部就绪后选择适当的连接算法进行连接计算，最后将结果集返回客户端。

根据以上执行流程，得出连接查询的响应时间：$T_{响应时间}$ 由如下几部分组成：

$$T_{响应时间} = T_{数据接收与处理} + T_{连接计算} + T_{数据本地提取} + T_{数据网络传输}$$

其中数据接收与处理任务与数据本地提取任务的完成时间主要由数据本地的提取速度决定，因此影响连接查询效率的因素主要有三方面：数据的本地提取、连接算法的执行效率、数据的网络传输。下面介绍一下这三方面影响连接查询效率的因素：

1. **数据的本地提取**：连接查询处理中最重要的一环是对数据的处理。数据的本地提取主要是将连接计算节点请求的数据从本地磁盘加载到内存，经过解压、过滤、序列化等一系列操作将数据通过网络发送到连接计算节点，其中涉及到两种数据提取的方式：索引定位和范围扫描。上一节中提到，如果是有主键与索引的过滤方式，则不需要太大的并行度，但如果是范围扫描的方式，则需要根据数据量的大小适当提高数据提取的并行度。
2. **连接计算的效率**：同样由连接计算节点来完成，前文已经介绍了传统数据库中常用的三种连接算法，并分析了其优缺点。想要快速的完成连接计算的任务，根据不同场景选取合适的连接算法，利用服务器多核 CPU 的计算资源，适当的增加连接计算的并行度。
3. **数据的网络传输**：在分布式环境下，减少数据的网络传输开销所用到的技术主要为半连接操作，在连接的两张表的数据量都很大的情况下，可以使用二次半连接操作[48]，进一步对两张表的数据进行过滤，减少数据的传输量。

本小节在分布式存储架构的基础上，对连接查询效率的影响因素进行了总结，针对以上几点影响因素，提出有效的优化方法，提高连接查询的效率，降低响应时间，提升用户体验是本文的研究重点。接下来对满足分布式存储架构的开源数据库 OceanBase 进行详细的介绍。

3.3 OceanBase 介绍

OceanBase 是阿里巴巴集团自主研发的可扩展的关系型数据库，其数据库集群可以存储数百 TB 的数据。到 2012 年 8 月份，OceanBase 支持了淘宝收藏夹、天猫评价等多项线上业务，线上数据处理总量已达千亿条。OceanBase 使用主流的普通 PC 机来搭建数据库集群，替代传统商业数据库、高端服务器、高端存储相结合的解决方案，降低建设成本。并且 OceanBase 数据库集群的扩容与缩容成本也因为廉价的 PC 机得到了很好的控制。在 2013 年的“神棍节”双 11 期间创下了数百亿元人民币的交易额，平均每分钟将近 90 万并发量的成绩[49]。

OceanBase 的优点在于：

1. **其一**，OceanBase 是一款分布式**关系型数据库**。分布式数据库如代表性的 NoSQL 数据库[50]：HBase、MongoDB，以及以 VoltDB 为代表的 NewSQL 数据库[51]，它们都具有高吞吐、低延迟、可扩展、高可用等优点，但是其优秀的数据库性能表现，是以牺牲数据的高度一致性为代价。相比于 NoSQL 与 NewSQL 数据库，对于数据一致性要求高的金融行业，分布式关系型数据库系统更能满足其需求。通过调研，互联网公司的业务不仅仅只是包含如网站日志、超链接、图片、视频等等大量的非结构化数据，还包括很多结构化的数据，如支付宝的移动支付，各电商网站的广告计费，网上购物等等商务金融交易，以上这些业务都需要依赖数据库的事务处理能力，因此传统的关系型数据库必是不可少的。
2. **其二**，OceanBase 具有分布式存储架构下良好的扩展性。OceanBase 架构中查询处理节点与数据存储节点可以分别部署于不同的服务器上，并且查询处理节点与数据存储节点的都是无状态的，可以根据业务需求进行横向的扩展。

OceanBase 是一款成长中的数据库，其配套的外围工具也在不断的补充，如数据库常用的监控系统，ETL 工具[52]等等。并且系统在某些场景下的的连接查

询效率也亟待提高，下面几个小节将详细介绍 OceanBase 的设计架构，以及连接查询技术的优缺点。

3.3.1 OceanBase 系统架构

OceanBase 的整体架构如图 3.4 所示，OceanBase 由以下几个部分组成：

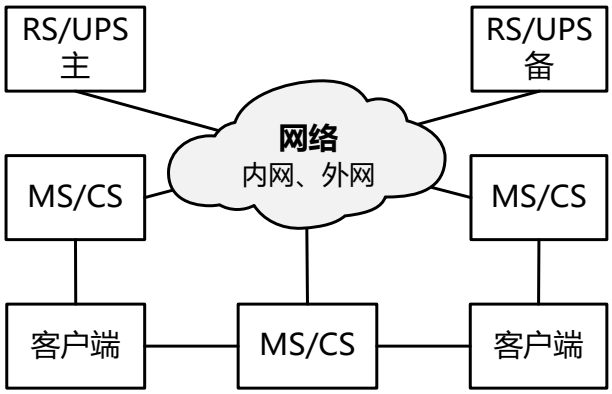


图 3.4 OceanBase 整体架构图

1. **客户端**：用户使用 OceanBase 的方式与使用 MySQL 的方式一样，因为其实现了 MySQL 客户端的大部分协议，以往部署在 MySQL 上的应用或者基于 MySQL 开发的工具都可以无缝迁移到 OceanBase 上，并且还支持 JDBC 以及 C 客户端的访问方式。
2. **RootServer（RS）**：负责集群中所有服务器的管理、数据分布的控制以及副本的管理， 一般情况下一个 OceanBase 集群中会有一主一备两台服务器分别部署 RootServer，并且主备之间数据强同步，来保证系统的高可用性。
3. **UpdateServer（UPS）**：查询请求中的写操作产生的增量数据主要保存在 UpdateServer 上，因此 UpdateServer 也作为 OceanBase 存储引擎之一，一般情况下 UpdateServer 采用一主一备的部署方式，并且主备 UpdateServer 之间的数据同步方式可以根据实际情况进行改变，另外，RootServer 与

UpdateServer 通常被部署于同一台服务器上。

4. **ChunkServer（CS）**：OceanBase 数据存储引擎的另一部分，存储系统的基线数据。OceanBase 中数据以两种形式存在，一种是存储于 UpdateServer 服务器内存中的增量数据，另一种是存储于 ChunkServer 服务器磁盘中的基线数据。基线数据通过系统的定时合并机制得来，并存有多个副本，一般情况下会存储为三份，副本个数可根据数据的规模进行动态的调整，数据的负载均衡由 RootServer 来控制。
5. **MergeServer（MS）**：相当于传统关系数据库中的查询处理引擎，其负责接收并解析用户的 SQL 请求，经过一系列的此法解析、语法解析、查询优化等操作后产生可执行的物理计划，物理执行计划向相应的 ChunkServer 以及 UpdateServer 请求数据，有些操作可以在 ChunkServer 上执行后返回结果，如投影、选择等，而像聚合、连接等复杂的运算则需要在 MergeServer 上执行。一般情况下，数据的请求不会只针对一台 ChunkServer，因此 MergeServer 还需要将多台 ChunkServer 返回的数据进行合并与排序。MergeServer 与客户端之间采用和 MySQL 相同的通信协议，MergeServer 可以解析大部分 MySQL 自带的命令，用户可以通过 MySQL 客户端来直接访问 MergeServer。

由以上 OceanBase 各个组件的介绍中可知，OceanBase 是分布式存储架构的典型，OceanBase 的查询处理引擎 MergeServer 对应着计算节点，在不考虑更新请求的情况下，OceanBase 的存储引擎 ChunkServer 对应着存储节点，并且 MergeServer 与 ChunkServer 可以根据应用的需求进行横向扩展，以提高请求的并发量与存储的容量。

接下来将详细介绍一下 OceanBase 中 MergeServer 的工作原理，并对 ChunkServer 进行简单的介绍，因为其存放着系统数据量庞大的基线数据，数据的访问方式对于连接查询优化起到重要的作用。

3.3.2 OceanBase 数据存储引擎之 ChunkServer

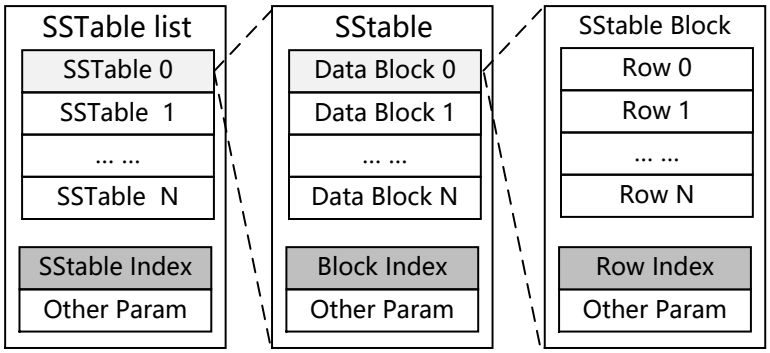


图 3.5 OceanBase 底层数据的索引结构概览图

OceanBase 的数据以主键范围切分，分布式存储于各个 ChunkServer 节点。并且数据的访问采用三级索引结构，如图 3.5 所示为 OceanBase 的底层数据的索引结构概览图，行记录在每个 Data Block 中以主键序，顺序存储。实体表与静态有序表（SStable,static sorted table）是一对多的关系，在数据请求过程中，首先根据 SStable index 获取对应的 SStable，继而通过 SStable 中的 Data Block index 获取需要访问的 Data Block，Data Block index 由每个 Block 中最后一行的主键组成，最后通过 Data Block 中的 Row index 定位到具体某一行，以上构成了数据访问的三级索引结构。

MergeServer 利用数据分布信息确定待请求的数据存放在哪些 ChunkServer 上，并向多个 ChunkServer 同时发送数据请求信息，ChunkServer 根据收到的请求信息确定需要将哪些 Data Block 加载到内存中，然后再根据过滤条件获取具体的行记录，最后将结果返回给 MergeServer。

OceanBase 的基线数据是以主键顺序存储，其存储结构中的 Data Block 是通过主键为索引来获取的，因此数据的请求如果是根据主键来过滤数据，待请求的数据会根据索引结构快速的加载到 ChunkServer 的内存中，但是如果过滤条件不是主键，则需要将所有的 Data Block 加载到内存中进行筛选，相比于主键提取会增加磁盘 I/O 的开销，导致数据响应缓慢。特别地，在半连接操作中，最重要的

环节就是对右表数据的过滤阶段，在已知过滤数据的情况下，针对不同的实际环境，选用适当的过滤方式，快速的将需要的数据从磁盘加载到内存中，可以有效的提高半连接操作的效率。

3.3.3 OceanBase 查询处理引擎之 MergeServer

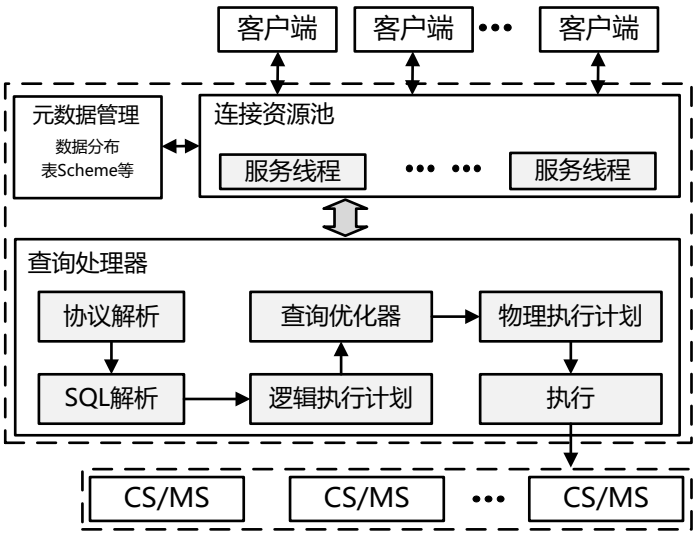


图 3.6 MergeServer 查询处理流程

图 3.6 所示为 MergeServer 的查询处理流程，OceanBase 的 MergeServer 有传统关系数据库查询引擎的所有功能模块，包括**连接资源池**、**元数据与公共资源管理**、**查询处理器**等等。OceanBase 的客户端与 MergeServer 之间的通信协议采用 MySQL 原生协议，其查询处理流程如下所示：

1. MergeServer 使用 MySQL 原生协议从客户端发送过来的查询请求，获取具体的 SQL 语句；
2. 进行词法、语法解析，生成逻辑执行计划与物理执行计划，物理执行计划树中包含各种一元、二元甚至与多元操作符，这些操作符有的负责数据提取，有的负责数据过滤，有的负责数据排序；
3. 根据数据分布信息，向相关的 ChunkServer 发起数据请求，ChunkServer 提取本地数据到内存中，经过过滤条件的筛选，将满足条件的数据发送给

MergeServer。

4. MergeServer 接收到数据后，进行连接、排序、聚合等运算，最后将结果返回客户端。

OceanBase 数据库中的数据存在两种状态：一种是存放在 UpdateServer 中的增量数据，一种是存放在 ChunkServer 上的静态数据。当用户发起查询请求时 MergeServer 需要将静态数据与增量数据进行合并，并将合并后的结果发送给客户端。

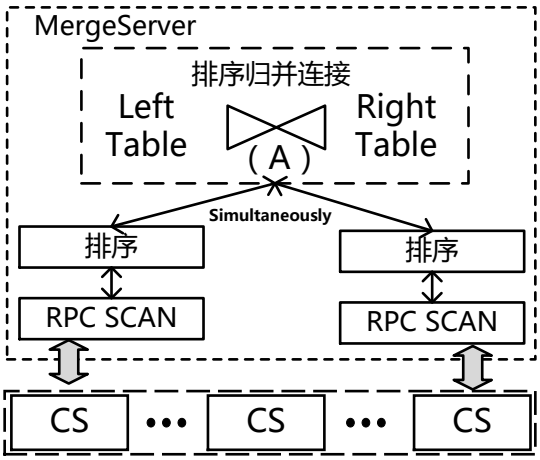


图 3.7 OceanBase 排序归并连接流程

图 3.7 所示为 OceanBase 内连接的处理流程，使用的连接算法为排序归并连接，也是 OceanBase 0.4.2 版本中唯一的连接算法。OceanBase 连接查询流程如下：

1. 排序归并连接算法处理模块同时驱动两个排序操作符获取数据；
2. 排序操作符驱动“RPC SCAN”（远程过程调用）[53]操作符进行远程数据的请求；
3. ChunkServer 接收到数据请求后，将数据从磁盘加载到内存中，并根据过滤条件对数据进行筛选，最后将满足条件的数据通过网络发送回排序操作符，排序操作符对数据进行排序，排好序的数据等待排序归并连接算法处理模块的调用；

4. Left Table 与 Right Table 将排好序的结果集，通过排序归并连接算法进行连接计算，筛选出满足条件的元组，响应客户端。

OceanBase 的排序归并连接流程中对于左右两张表的数据请求是同时发起的，数据到位的时间是大表的数据本地提取时间与网络传输时间之和，数据到位指的是数据都到达 MergeServer。现实中的业务需求中除了统计类查询、定时任务查询、报表类查询不可避免的涉及到大表的连接，结果集的规模通常也很大以外，其他的普通查询最终结果集都在百条或者千条之内。按照 OceanBase 的连接查询执行流程，意味着在即使查询最终的结果集较小，而连接查询涉及到的都是大表的情况下，也要将所有数据通过网络传回 MergeServer，这就造成无用数据的网络传输，对连接查询的响应时间产生影响。

3.4 本章小结

本章首先对基于分布式存储的数据库架构进行了详细的介绍，然后分析了该架构下影响连接查询效率的三个因素：连接计算、数据本地提取、数据网络传输，最后对分布式存储架构的典型分布式数据库 OceanBase 进行详细的介绍，分析其连接查询方面的缺点。下一章将会针对影响连接查询效率的三个因素，提出具体的优化方法，并结合特定场景进行分析，最后在分布式存储架构的基础上，提出分布式数据库连接查询优化框架，并在 OceanBase 上进行实现。

第四章 分布式数据库连接查询优化框架

回顾前几章提到的查询处理方面的技术，第二章介绍了传统数据库查询优化中常用的三种连接算法，对通用的查询优化过程和基于半连接操作算法的简单介绍，其中**嵌套循环连接算法**、**哈希连接算法**以及**查询优化过程**与**半连接操作**等技术被用于本文提出的连接查询优化框架中。

第三章对基于分布式存储的数据库架构进行了详细的介绍，总结在分布式存储架构下影响连接查询效率的几点因素，并对计算与存储架构的典型分布式数据库 OceanBase 进行详细的分析。

综上所述，本文提出的分布式连接查询优化框架，以基于分布式存储架构的数据库为技术背景，使用诸如**嵌套循环连接算法**、**哈希连接算法**以及**查询优化过程**与**半连接操作**等数据库查询方面的技术，从影响连接查询效率的三个因素出发，有效的提高连查询的效率。

本章首先利用查询处理方面的技术，针对几点影响连接查询效率的因素，提出具体的优化策略，然后在分布式存储架构的基础上，结合连接查询优化策略，提出普遍适用的分布式连接查询优化框架，并在 OceanBase 上进行实现。

4.1 连接查询优化策略

本文提出的连接查询优化策略由两方面组成：面向分布式存储架构的查询优化过程、具体的优化策略。即在面向分布式存储架构的查询优化过程的基础上，针对影响连接查询效率的因素，提出了具体的优化策略。下面分别对这两方面进行详细的介绍。

4.1.1 面向分布式存储架构的查询优化过程

第二章介绍了通用的查询优化过程，其中的代价模型作为评估查询计划可行性与执行效率的主要模块，它需要先借由搜索策略过滤掉搜索空间中数量庞大的查询计划，然后对需要进行评估的查询计划进行执行代价的计算工作，在分布式环境下，代价函数的结果是查询总的响应时间。

通用的查询优化过程采用静态优化的策略，其首先根据 SQL 请求生成一组等价的查询计划，然后经过搜索策略与代价模型的过滤与评估，找出最佳的查询计划。这种做法的优点，是在执行之前就确定了执行过程中所需要的操作类型并规定了操作顺序，所以在执行过程中就不用考虑执行效率波动的问题。缺点在于搜索空间的生成将会耗费大量的时间，特别是在连接关系很多的情况下，搜索空间中的查询计划将呈爆炸式的增长，影响搜索策略模块与代价模型对查询计划的评估效率，最终就会导致查询响应时间的变长。鉴于以上问题，本文在通用查询优化过程的基础上进行了改进，提出了面向分布式存储架构的查询优化过程。

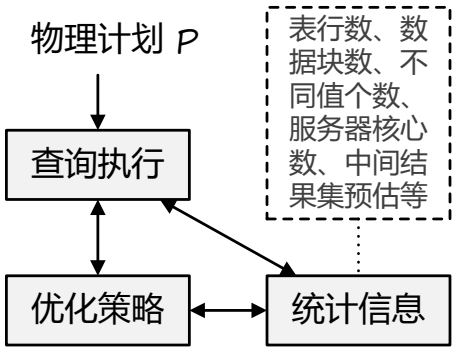


图 4.1 面向分布式存储架构的查询优化过程

图 4.1 为面向分布式存储架构的查询优化过程，其中包括：**查询执行**、**优化策略**、**统计信息**三个模块。优化策略为针对影响连接查询的因素，提出的指导查询优化的策略，统计信息包括：表的行数、表的数据块数、索引字段的行数、不同值的个数、服务器的 CPU 核心数等等。

与通用查询优化过程的不同点如下：

1. 通用的查询优化过程在生成搜索空间时，会产生所有等价的查询计划，而本文改进后的查询优化过程去掉了搜索空间、转换规则、搜索策略、代价模型等环节，只产生一个确定的查询计划，并且查询计划中的连接方式为内连接。这样改动的前提是：在生成查询计划之前已经对除连接算子之外的其他方面进行了优化，其他方面如：关系转换、操作符下压、使用索引和表达式调优等等。

   图 4.2 为内连接查询计划树的几种不同树形，图 4.2-1 中T_1与T_2进行连接产生的结果集与T_3进行连接，产生的结果集再与T_4连接，生成最终结果集，这种树形称为左深树，其规定了查询中各个连接操作的执行顺序为从左到右依次执行，图 4.2-2 为右深树，其规定了查询中各个连接操作的执行顺序为从右到左依次执行，图 4.2-3 则为其他众多树形中的一种。在本文提出的面向分布式存储架构下的查询优化过程中，查询计划树是一颗左深树。

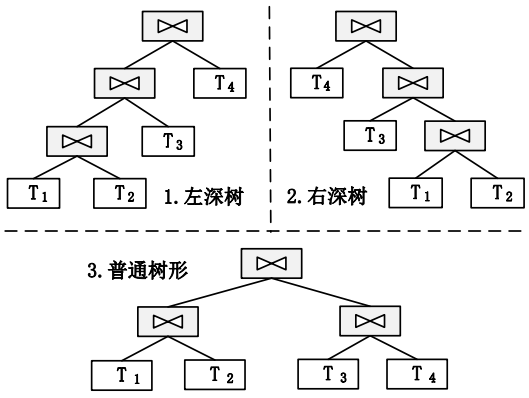


图 4.2 查询计划树

2. 在通用的查询优化过程中，查询计划的执行是确定的，即在执行过程中不会再对其进行任何方面的优化，而在本文提出的查询优化过程中，查询计划的执行伴随着动态的调优，即在查询执行的过程中，利用优化策略与统计信息对查询计划树中的每个连接操作进行动态的调整，从连接算法的

执行效率、数据的本地提取、数据的网络传输等三方面对连接查询进行优化。

面向分布式存储架构的查询优化过程有两个特点：

1. 只产生一个确定的查询计划，这个查询计划是在进入查询优化流程之前根据请求 SQL 生成的查询计划框架，即这个查询计划中的连接操作符类似于一个模板，至于具体的连接算子与其他物理操作符需要在执行过程中根据优化策略以及统计信息的进行动态的调整。连接方式为内连接，查询计划的树形为左深树，在进行优化之前明确规定了连接执行的顺序，这样处理的优点在于避免在查询执行过程中对连接操作的执行顺序进行调整，增加查询优化的复杂性与难度；缺点在于由查询处理引擎生成的查询计划，可能不具备最优的执行顺序。
2. 在查询执行的过程中，利用优化策略与统计信息对查询计划进行动态的调优。查询计划中的每个连接操作所需使用的连接算子以及计算资源都有所不同，在执行过程中对其进行动态的调整，可以有效的利用系统的计算资源，发挥出各连接算法在不同应用场景的优势，还可以适当的选用半连接操作减少数据的网络传输。

图 4.1 中物理计划 P 在执行过程中，优化策略模块需要根据统计信息模块提供的信息与执行过程中产生的中间结果，对物理计划 P 后续的执行操作进行动态的优化。下一节将对具体的优化策略进行详细的介绍。

4.1.2 优化策略介绍

第三章对计算与储存分离架构下影响连接查询效率的三点因素进行了分析，包括：数据的本地提取、数据的网络传输和连接算法的执行效率。

本文针对以上影响连接查询效率的因素，提出了具体的优化策略，从**并行度、连接算子、半连接操作**等三方面对连接查询进行优化，下面分别介绍一下并行度、连接算子、半连接操作所关注的主要问题：

1. **并行度**：主要关注对服务器计算资源的利用情况，多核架构下的 CPU，

其计算能力较之单核 CPU 有了很大的提升，随着 CPU 核心数目的不断增多，并行计算成为了有效加快任务处理速度的技术。针对数据的本地提取速度与连接算法执行效率的问题，本文采用并行技术，一方面使数据的请求与处理并行执行，提升数据的本地提取速度，另一方面加快连接算法的执行效率，降低了连接查询的响应时间。

2. **连接算子：**主要关注连接算法的适用场景。连接算子为哈希连接、嵌套循环连接、排序归并连接等连接算法的物理实现。基于对内存、数据量、连接方式等方面的考量，针对不同场景选择合适的连接算子，充分利用服务器硬件的计算资源，可以有效的提高连接查询的效率。
3. **半连接操作:**主要关注数据的网络传输开销。在分布式环境下，数据的网络传输开销作为影响连接查询效率的主要因素，一直是重点优化的对象，目前较为有效的优化手段就是使用半连接操作。半连接操作的核心思想是通过左表已知的结果集，通过其连接属性列上的数值来对右表的数据进行过滤，已达到减少无用数据网络传输的目的。本文对半连接操作进行了相应的设计，使其可以利用并行技术，将左表结果集拆分成多份，并行的对右表数据进行过滤，进一步提高数据过滤的效率，同时也变相的加快了数据的本地提取速度。

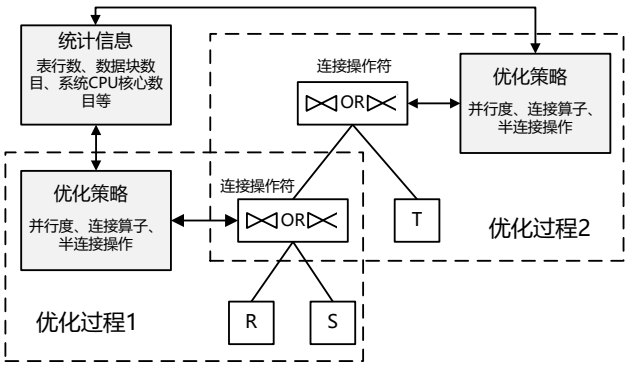


图 4.3 优化过程

本文提出的优化策略核心思想为：首先确定使用哪种连接算子，针对不同的应用场景选择合适的连接算子，可以提高连接查询的效率；然后判断是否使用半

连接操作，用来减少数据的网络传输；其次确定半连接操作在执行过程中，数据过滤的并行度，选用恰当的并行度，不仅可以充分利用服务器的计算资源，还可以提高存储节点本地数据的提取速度；最后确定连接算子中连接计算的并行度，主要包括：数据处理的并行度和连接计算的并行度，数据处理的并行度主要关注对数据进行快速的拆分，计算的并行度是在数据分区的基础上，对每份数据进行单独的连接运算，提高连接查询整体性能。图 4.3 为引入优化策略的查询优化过程，以优化过程 1 为例，优化策略的具体流程如下：

1. **判断使用哪种连接算子（Hash-Join or Nestloop-Join or Merge-Join）：**
   a) 根据连接类型选择连接算子，如果是等值连接初步考虑使用 Hash-Join 或者 Merge-Join，否则直接使用 Nestloop-Join，如果使用 Nestlooop-Join 跳到步骤 2；
   b) 通过统计信息获取 R 表与 S 表的数据行数，判断 R 表与 S 表的数据行数是否大于给定阈值：“Merge_or_Hash”， “Merge_or_Hash”为根据不同的输入总结得出的实验数据。如果大于给定阈值则使用 Hash-Join 否则使用 Merge-Join；
2. **判断是否使用半连接操作（Is_use_SemiJoin）：**
   a) 连接操作符驱动下层操作符获取 R 表结果集；
   b) 根据 R 表结果集行数与 S 表的行数，以及给定阈值：“Is_use_SemiJoin”判断是否使用半连接，给定阈值为 R 表结果集与 S 表数据量的比例，例如：Is_use_SemiJoin = 0.1，即如果 R 结果集为 S 表数据量的十分之一，则使用半连接。如果不使用半连接跳到步骤 4；
3. **确定半连接操作并行度：**
   根据 R 表结果集的行数:ResultSet_Size（R）、当前系统可用线程数:Idle_Thread_Num、单线程最高数据处理总量：Thread_Capacity，确定数据过滤的并行度，对 S 表的数据进行并行的过滤，提高数据提取效率。如果系统负载不高，则并行度可设置为 Idle_Thread_Num，否则根

据 ResultSet_Size（R）、Thread_Capacity、Idle_Thread_Num 三个参数对并行度进行调整，例如：ResultSet_Size（R）= 10000、Thread_Capacity = 5000、Idle_Thread_Num = 20，则并行度为 ResultSet_Size（R）/ Thread_Capacity = 2，在可用线程个数范围内。

4. **确定连接算子并行度：**

根据 R 表与 S 表在执行过程中产生的中间结果集行数：ResultSet_Size（R）和 ResultSet_Size（S）、当前系统可用线程数：Idle_Thread_Num，确定连接算子的计算并行度，采用相应的哈希函数，将数据进行分区，分区的个数与并行度相同，所有数据并行处理以提高连接计算的执行效率。

如：ResultSet_Size（R）和 ResultSet_Size（S）的平均值为 5 万，并且 Idle_Thread_Num = 10，则并行度可以设置为 5。

优化过程 2 在优化过程 1 的基础上，继续使用优化策略来指导查询的执行，直到所有连接操作处理完毕。优化策略中涉及到的所有阈值均为实验数据，具体数值需要根据具体情况进行改动，具体实现细节需要根据原型系统进行详细的考虑。

本文提出的优化策略主要目的是用于指导查询执行过程中对连接操作的动态调整。调整包括：连接算子的选取、半连接操作的使用与否、连接算子与半连接操作的并行度。本文所介绍的优化策略内容只是对于优化方向上的一些建议，对于连接算子的选取、半连接操作的使用和并行度的考量，需要根据具体的应用需求进行相应的调整。优化策略模块是一个相对独立的模块，其中的优化策略是根据实践经验总结出来的有效方法，本文提出的优化策略并不能涵盖所有情况，特别地，对于连接查询中涉及的表数量较多，并且每张表的数据量较大，而最终的结果集较小的情况，优化策略可以有效的减少查询响应时间。传统数据库基于成本的查询优化器，会缓存最优的查询计划，进一步加快查询的效率，本文提出的优化策略相当于缓存的查询计划，并且这个查询计划还可以通过对查询执行过程中的连接操作进行动态的调整来适应更多的情况。

4.2 连接查询优化框架

基于面向分布式存储架构的查询优化过程与具体的优化策略，在分布式存储架构的基础上，本文提出了分布式连接查询优化框架。

图 4.4 为分布式数据库连接查询优化框架的具体内容，接下来本文按照从左到右的顺序依次对框架中的各个模块进行介绍：

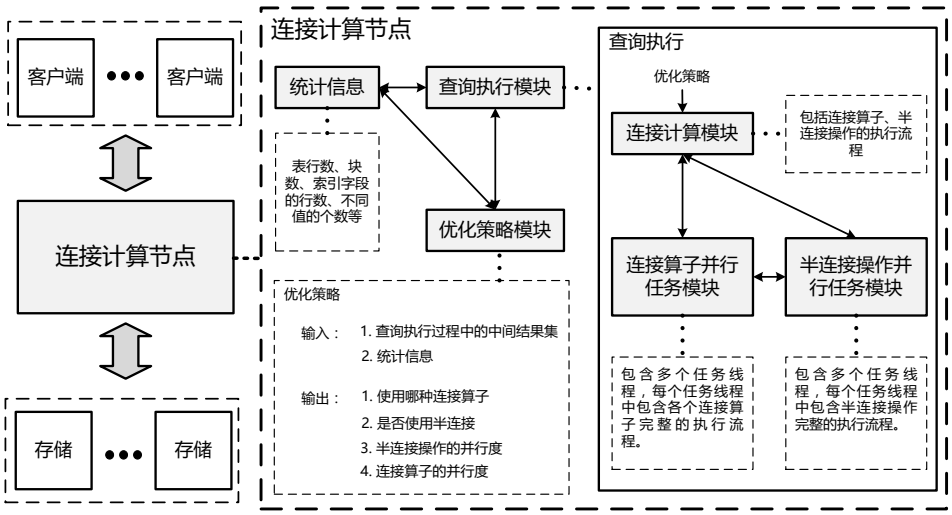


图 4.4 分布式数据库连接查询优化框架

1. 最左边为基于分布式存储架构的数据库简图，客户端向连接计算节点发送查询请求，连接计算节点产生查询计划，向存储层请求数据，最后连接计算节点将处理后的结果集发送回客户端。在基于分布式存储架构的数据库中，计算节点的处理能力决定了连接查询的响应时间，因此需要充分利用系统的硬件资源，采用并行技术，提高连接查询的执行效率。
2. 中间部分为连接计算节点的实现细节，包括查询执行模块、优化策略模块、统计信息模块。查询执行模块作为驱动节点，根据优化模块与统计信息模块提供的优化策略、数据库系统的统计信息与服务器的 CPU 核心数目等硬件资源，对查询计划中的连接操作所涉及的连接算子进行动态的调整，对服务器的硬件资源进行动态的分配（主要为计算资源）。

其中，统计信息中表的行数、CPU 的核心数等信息，用于确定连接算子与半连接操作的并行度。优化策略模块包括两个输入信息和四个输出信息，具体地，输入信息为：查询执行过程中的中间结果集信息与统计信息，通过对这些数据进行分析，提出具体的优化策略。优化策略模块的输出为具体的优化策略：

a) 明确指出使用哪种连接算子；
b) 明确指出是否使用半连接操作；
c) 给出半连接操作的并行度；
d) 给出连接操作的并行度。

查询执行模块根据具体的优化策略，进行下一步的连接计算与资源的分配。对于单次连接查询，其中可能包括多个连接操作任务，因此对于每个连接操作都需要由统计信息与优化策略模块提出具体的优化建议，使连接查询的每个环节都得到动态的调整，提升连接查询的效率。

3. 最右边为查询执行的具体流程与各个功能模块，包括：连接计算模块、连接算子并行任务模块、半连接操作并行任务模块。连接计算模块中封装了各种连接算法以及半连接操作的物理实现，负责任务的分发、结果集的汇总、连接的计算等任务。连接算子并行任务模块与半连接操作并行任务模块主要功能如下：

a) 根据统计信息中系统计算资源的使用情况，动态的对任务线程的个数进行调整，使得查询处理节点的线程使用情况处于均衡的状态，避免资源利用过度；
b) 提供任务线程的管理服务，如线程的挂起、分配、回收、启动等；
c) 任务线程的处理模块包括所有连接算子以及半连接操作的执行流程，每个任务线程完成部分任务并返回中间结果；

以上为连接查询优化框架中各模块功能的详细介绍，本文提出的连接查询优

化框架结合分布式存储的系统架构、连接查询的优化策略以及对系统计算资源的充分利用，采用并行技术从影响连接查询效率的因素出发，在查询执行的过程中对连接查询的各个连接操作进行动态的调优，减少了连接查询的响应时间，提高了连接查询的效率。下一小节介绍连接查询框架在 OceanBase 的上的详细设计方案，并付诸实现。

4.3 连接查询优化框架中各模块的详细设计

通过对 OceanBase 架构中连接查询相关方面的调研，结合本文提出的连接查询优化框架，对以下四方面进行详细设计：

1. 优化策略模块；
2. 连接算子并行任务模块；
3. 半连接操作并行任务模块；
4. 连接计算模块。

优化策略模块、连接算子并行任务模块、半连接操作并行任务模块是相对独立的功能模块，只有对连接计算模块的设计，需要考虑 MergeServer 的具体实现，下面分别介绍各模块的设计方案。

4.3.1 优化策略模块

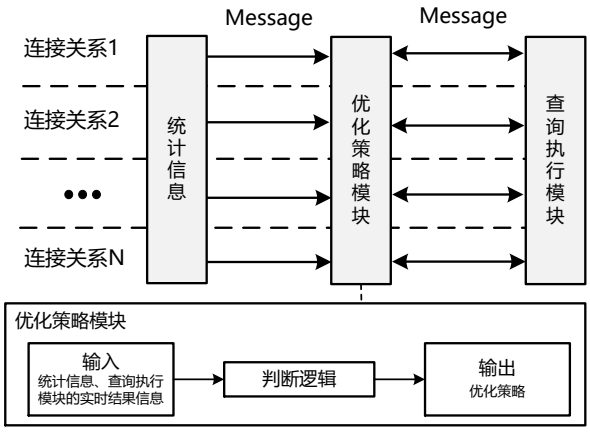


图 4.5 优化策略模块设计图

单个连接查询中包括多个连接关系，如：R S ⋈ T，其中 R ⋈ S 为一个连接关系，R ⋈ S 产生的结果集 ResultSet(R ⋈ S)与 T 进行连接，ResultSet(R ⋈ S) ⋈ T 也是一个连接关系，因此 R ⋈ S ⋈ T 有两个连接关系，每个连接关系称为一个连接操作。

图 4.5 为优化策略模块的设计方案，包括与统计信息模块、查询执行模块的信息交互情况，对于一个连接查询请求，优化策略模块需要与统计信息模块、查询执行模块进行多次的信息交互，对每个连接关系进行动态的调优。

针对一个连接关系，优化策略模块与统计信息模块、查询执行模块的信息交互细节如图 4.6 所示：

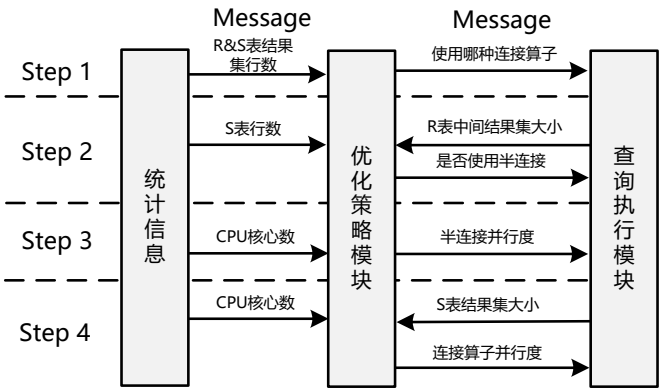


图 4.6 优化策略模块信息交互细节

优化策略模块实现判断逻辑，输入为统计信息与查询执行模块的实时中间结果信息，输出为具体的优化策略，下面为判断逻辑的四个处理阶段：

Step1：优化策略模块根据统计信息提供的 R 表与 S 表的中间结果集行数，判断使用哪种连接算子，将结果发送给查询执行模块。统计信息提供的中间结果集行数是根据：R 表行数、S 表行数、查询计划关于 R 表与 S 表的过滤条件等信息预估的结果。

Step2：优化策略模块根据统计信息提供的 S 表的行数、查询执行模块提供的 R 表中间结果集行数，判断是否使用半连接，并将判断结果发送给查询执行模块，

其中 R 表中间结果集行数为查询执行过程中的真实结果。

Step3：优化策略模块根据统计信息提供的 CPU 核心数、R 表中间结果集行数，计算半连接操作的数据过滤并行度，发送给执行模块。

Step4：优化策略模块根据统计信息提供的 CPU 核心数、查询执行模块提供的 R 表和 S 表的中间结果集行数，计算连接算子的并行度，并发送给执行模块。

以上判断逻辑内容为在本文在实际工作环境中提出的判断逻辑，具有一定的狭隘性，但是优化策略的思想具有普遍的适用性，可以为连接查询的优化提供良好的建议。

4.3.2 连接算子与半连接操作并行任务模块

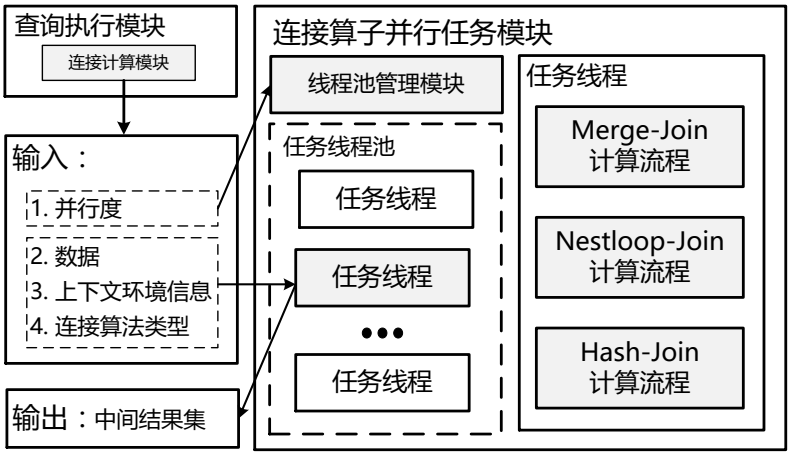


图 4.7 连接算子并行任务模块

图 4.7 为连接算子并行任务模块的设计方案，包括两个重要的组件：线程池管理模块、任务线程。

线程池管理模块负责线程池的创建、资源的分配、线程的唤醒与挂起、动态的线程创建与销毁、系统 CPU 的负载情况监控。任务线程负责具体的计算任务，内部封装了 Merge-Join、Hash-Join、Nestloop-Join 的具体实现。任务线程根据连接算法的类型、输入的数据、上下文环境信息，对数据进行连接处理。

连接算子并行任务模块的输入由查询执行的连接计算模块提供，分为两个部

分：并行度为线程池管理模块的输入；数据、上下文环境信息、连接算法类型为任务线程的输入。线程池管理模块根据请求的并行度、当前空闲的任务线程、CPU的负载，为连接计算模块提供相应数目的任务线程。

图 4.8 为半连接操作并行任务模块的设计方案，与连接算子并行任务模块拥有相同的组件：线程池管理模块与任务线程。不同之处有两点：

1. 半连接操作并行任务模块的输入参数由“连接算法类型”变为“左表部分结果集”，输出参数由“中间结果集”变为“右表部分数据”。
2. 任务线程的任务类型由连接算法的计算任务，变为根据左表部分结果集，构造用于过滤右表数据的数据结构，并将数据结构发送到相应存储节点，进行数据的过滤。

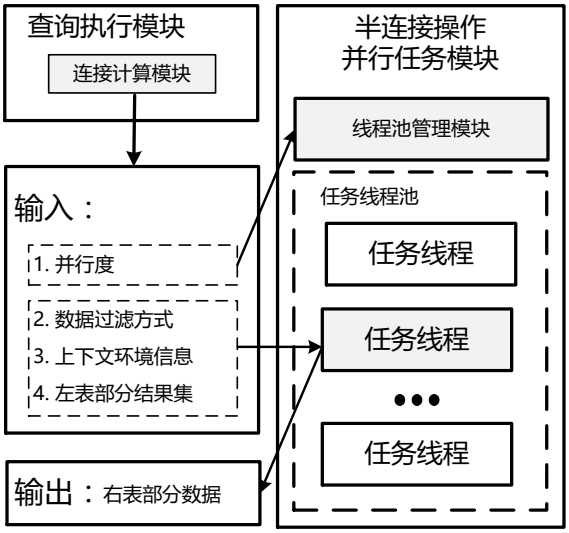


图 4.8 半连接操作并行任务模块

由于 OceanBase0.4.2 版本缺少对 Hash-Join、Nestloop-Join 的支持，因此连接算子并行任务模块保留了 Hash-Join、Nestloop-Join 的实现接口。并且 OceanBase0.4.2 版本也没有半连接操作的相关实现，因此半连接操作并行任务模块同样预留了半连接操作流程的实现接口。Hash-Join、Nestloop-Join、半连接操作的相关实现细节将在下一章进行详细的介绍。

4.3.3 连接计算模块

OceanBase 上的连接计算任务由 Merge-Join 操作符完成，Merge-Join 负责数据的请求与连接算法的具体实施。OceanBase 的连接计算流程如图 4.9 所示，涉及到**投影**、**排序**、“**RPC Scan**”三个物理操作符。**投影**操作符负责数据的投影与过滤；**排序**操作符使用内外排混合的排序算法对数据进行排序处理；“**RPC Scan**”操作符负责从远程服务器拉取数据。

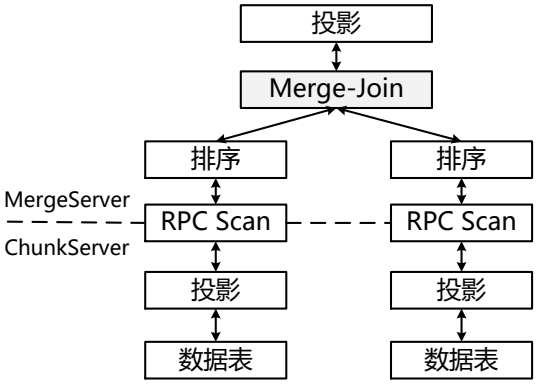


图 4.9 OceanBase 的连接计算流程

OceanBase 的连接计算流程中，Merge-Join 操作是连接计算的核心，它驱动其他物理操作符进行数据请求、数据排序、数据过滤等任务。然而在本文提出的连接查询优化框架中，连接算子是在执行过程中确定的，因此需要对 OceanBase 的连接计算流程进行改进，改进后的连接查询流程如图 4.10 所示：

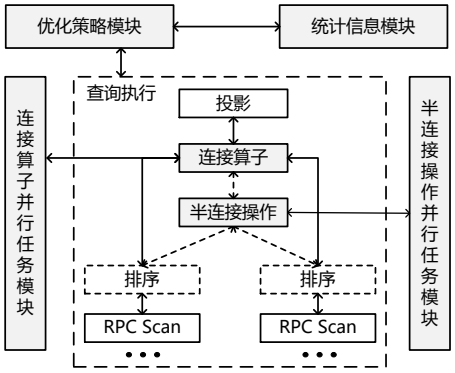


图 4.10 改进后的 OceanBase 连接计算流程

在原流程的基础上，新增优化策略模块、统计信息模块、连接算子并行任务模块、半连接操作并行任务模块、半连接操作符，查询执行中的 Merge-Join 操作符变成连接算子操作符。虚线表示**排序**操作符与半连接操作是根据优化策略的调优，动态的添加到计算流程当中的。连接算子作为连接查询的驱动模块，调动优化策略模块、连接算子并行任务模块以及半连接操作并行任务模块，对连接查询进行动态的优化。

对于策略模块、连接算子并行任务模块、半连接操作并行任务模块，前文已经给出了详细的设计方案，连接算子与半连接操作的并行实现将在下一章进行详细的介绍。

4.4 本章小结

本章首先提出面向分布式存储架构的查询优化过程，在此优化过程的基础上，针对影响连接查询的几个因素，提出具体的优化策略，从并行度、连接算子、半连接操作三方面对连接查询进行优化。

然后结合优化策略与优化过程，在分布式存储架构的基础上，提出分布式连接查询优化框架。最后在开源分布式数据库 OceanBase 上进行框架中各个功能模块的详细设计，并最终在 OceanBase 进行了实现。

第五章 连接算子与半连接操作的并行实现

第四章首先提出了分布式数据库连接查询优化框架，针对影响连接查询的几点因素，有效的提高了连接查询的效率。然后对连接查询优化框架中的各个模块进行详细的设计，并在 OceanBase 上进行实现。然而在 OceanBase 实现方案中，缺少对哈希连接、嵌套循环连接与半连接操作的支持，如果将这三方面技术与并行计算的思想相结合，则可有效的提高连接计算的效率，减少数据的网络传输。因此本章将详细的介绍哈希连接、嵌套循环连接与半连接操作的并行实现方案，并对优化框架中服务器硬件资源的使用情况进行简单的分析。

5.1 哈希连接与嵌套循环连接的并行实现

哈希连接的并行实现：

在上一章节中，本文对连接算子并行任务模块与连接计算模块进行了详细的设计，并在连接算子并行任务模块预留了实现哈希连接计算流程的接口。在连接计算模块的基础上，介绍哈希连接的并行实现设计方案与具体的实现流程。

在 OceanBase 架构下，哈希连接的并行实现设计方案如图 5.1 所示，设计方案中包括：连接算子并行任务模块、连接算子操作符、优化策略模块。优化策略模块负责判断使用哪种连接算子；连接算子操作符负责任务的分发与结果的整合，最后将结果集返回客户端；连接算子并行任务模块根据连接算子操作符分发的任务，分配相应数量的任务线程来进行任务的处理并返回数据。为方便描述，连接关系设定为 $R \bowtie S$。

按照时间线，从左到右介绍哈希连接的并行实现流程：

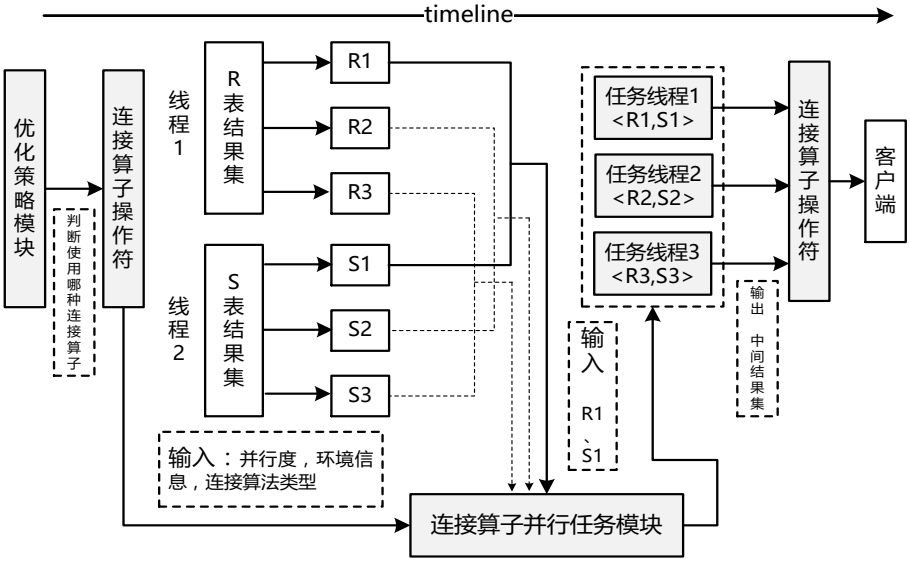


图 5.1 哈希连接的并行实现设计方案

1. 优化策略模块根据统计信息与查询执行模块的中间结果，利用判断逻辑提出使用哈希连接作为连接算子；
2. 连接算子操作符分配两个线程：线程 1 与线程 2，使用相同的哈希函数，并行的对 R 表与 S 表的结果集根据连接属性列进行分区处理。由于 R 表与 S 表的结果集大小不同，分区所消耗的时间也不相同，因此连接算子操作符线程要阻塞的等待两张表的数据都分区完成。
3. R 表数据分区完成后，分为 R1、R2、R3 三个结果集。S 表数据分区完成后，分为 S1、S2、S3。其中分区的个数由哈希函数确定，R 表的 R1 对应 S 表的 S1，以此类推，每一对分区如：<R1，S1>为一个任务线程的输入。
4. 连接算子操作符将并行度、环境信息、连接算法类型（哈希连接）、分区对如：<R1，S1>，作为输入，提交给连接算子并行任务模块。
5. 连接算子并行任务模块为每个分区对提供单独的任务线程：任务线程 1（<R1，S1>）、任务线程 2（<R2，S2>）、任务线程 3（<R3，S3>）。三个线程并行的对各自的数据分区进行处理。
6. 任务线程中的处理流程：以任务线程 1 为例，
   a) 根据分区 R1 中的连接属性列，使用哈希函数构造哈希表 H；

b) 分区 S1 使用相同的哈希函数对哈希表 H 进行探测；

c) 如果 S1 的连接属性列有数据落在哈希表上，则进行连接操作，生成新的结果集；

d) 处理完成后将生成的中间结果交由连接算子操作符。

e) 任务线程的任务结束，释放内存资源，清除环境信息，重新挂起并等待新的任务。

7. 连接算子操作符接收任务线程陆续发送过来的中间结果，并将结果放入操作符内部的缓存区，当所有任务线程完成处理后，连接算子操作符将缓存区中的数据发送给客户端。

以上为哈希连接的并行实现流程，并行体现在两方面：对 R 表与 S 表进行并行的分区处理；对每一对分区<R1,S1>并行的构建哈希表，并行的进行数据的探测，并行的返回中间结果。

嵌套循环连接的并行实现：

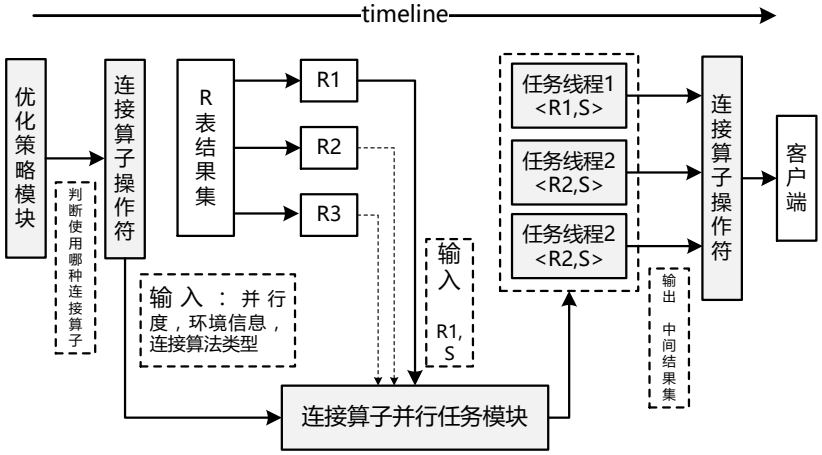


图 5.2 嵌套循环连接的并行实现设计方案

图 5.2 所示为嵌套循环连接的并行实现设计方案，其中包括：连接算子并行任务模块、连接算子操作符、优化策略模块，各模块的功能与哈希连接的设计方案中一致。不同之处在于接算子并行任务模块的输入信息，嵌套循环连接的实现流程如下：

1. 优化策略模块根据统计信息与查询执行模块的中间结果，利用判断逻辑提出使用嵌套循环连接作为连接算子。
2. 连接算子操作符线程，将 R 表结果集分为等份的几部分：R1、R2、R3。具体分为几等份，取决于 R 表结果集的大小与可用的任务线程个数。R 表的数据分区 R1、R2、R3 分别对应 S 表的结果集，分区对为<R1，S>、<R2，S>、<R2，S>。
3. 连接算子操作符将并行度、环境信息、连接算法类型（嵌套循环连接）、分区对如：<R1，S>，作为输入，提交给连接算子并行任务模块。
4. 连接算子并行任务模块为每个分区对提供单独的任务线程：任务线程 1（<R1，S>）、任务线程 2（<R2，S>）、任务线程 3（<R3，S>）。三个线程并行的对各自的数据分区进行处理。
5. 任务线程中的处理流程：以任务线程 1 为例，
   a) R1 作为驱动表，S 作为被驱动表，R1 作为外层循环，S 作为内层循环；
   b) 将 R1 中的记录逐条与 S 表的所有记录进行比对，如果满足连接条件，则生成新的元组，持续这一过程，直到 R1 的数据遍历完毕。
   c) 处理完成后将生成的中间结果交由连接算子操作符。
   d) 任务线程的任务结束，释放内存资源，清除环境信息，重新挂起并等待新的任务。
6. 连接算子操作符接收任务线程陆续发送过来的中间结果，并将结果放入操作符内部的缓存区，当所有任务线程完成处理后，连接算子操作符将缓存区中的数据发送给客户端。

5.2 半连接操作的并行实现

半连接操作的思想是用左表结果集中的连接属性列，对右表进行过滤，筛选出可以产生连接关系的结果集，通过网络发送回查询处理节点。因此本地数据过

滤的速度是影响半连接操作效率的主要因素，选用哪种过滤方法直接关系到数据本地提取的速度。本文结合 OceanBase 的具体实现细节，给出了半连接操作的并行实现设计方案。

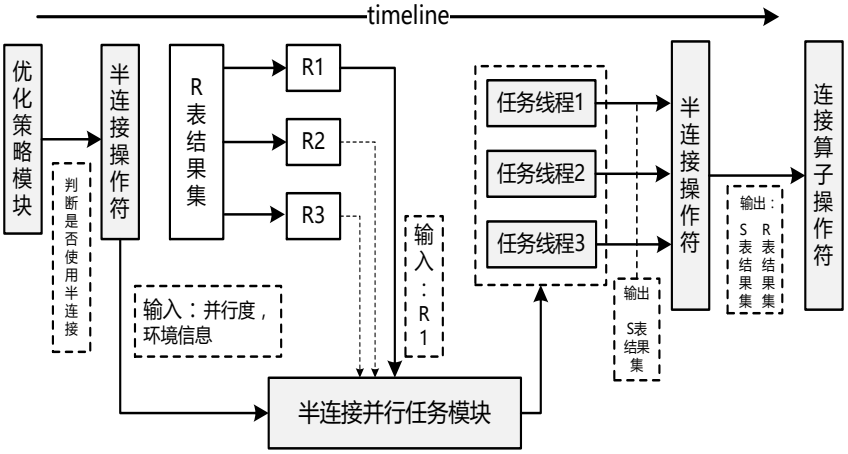


图 5.3 半连接操作的并行实现设计方案

如图 5.3 所示，半连接操作的并行实现设计方案包括：半连接操作符、连接算子操作符、半连接并行任务模块、优化策略模块。半连接操作符负责任务的分发，结果的汇总；连接算子操作符为半连接操作符的上层操作符，其负责向半连接操作符索取数据；半连接并行任务模块提供具体执行的任务线程，任务线程中实现了半连接操作的具体逻辑；优化策略模块给出是否使用半连接的优化建议。

本文从三个方面对并行的半连接操作进行介绍，包括：半连接操作整体的执行流程、任务线程的任务流程、数据过滤方式。

首先，连接操作整体的执行流程如下：

1. 优化策略模块根据统计信息与查询执行模块的中间结果，利用判断逻辑提出是否使用半连接操作。
2. 半连接操作符线程，将 R 表结果集分为：R1、R2、R3 几等份。
3. 半连接操作符将并行度、环境信息、R 表数据分区：R1、R2、R3 作为输入，提交给半连接操作并行任务模块。
4. 半连接操作并行任务模块为 R 表结果集的每个分区提供单独的任务线程。

5. 任务线程负责数据的过滤任务。
6. 半连接操作符接收任务线程陆续发送过来的中间结果，并将结果放入操作符内部的缓存区，当所有任务线程完成处理后，连接算子操作符将缓存区中的数据发送给连接算子操作符。

其次，任务线程的具体执行流程如图 5.4 所示，根据时间线由上到下进行介绍：

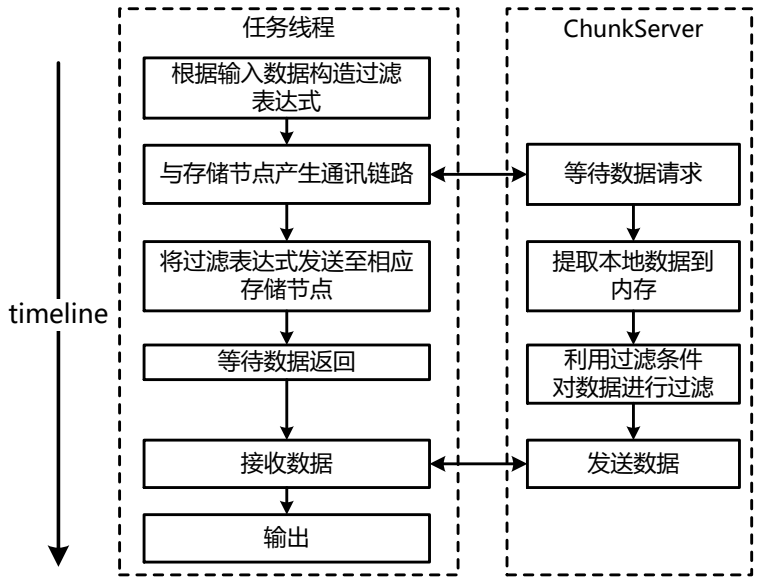


图 5.4 半连接任务线程具体流程

1. 任务线程根据输入的 R 表部分结果集 R1 构造过滤表达式，用于对 S 表的数据进行过滤。
2. 与存储节点 ChunkServer 建立通讯链路，用于消息与数据的传输。同时 ChunkServer 进入等待数据请求状态。
3. 任务线程将过滤表达式通过通讯链路，发送到相应的 ChunkServer，并进入数据等待状态。ChunkServer 将本地数据加载进内存，利用过滤表达式对数据进行筛选。
4. ChunkServer 将符合条件的数据通过通讯链路发送回任务线程，任务线程同步接收数据，并将数据加入缓存。

5. 任务线程将 S 表的部分数据返回给半连接操作符。

一次半连接操作，会申请多个任务线程并行的对 S 表的数据进行过滤，不同的过滤方式使得 ChunkServer 本地数据的提取速度也不相同。针对 OceanBase 的具体实现细节，本文主要考虑两种数据过滤方式：IN 表达式与 Between 表达式。下面介绍两种过滤方式的具体执行流程与优缺点。

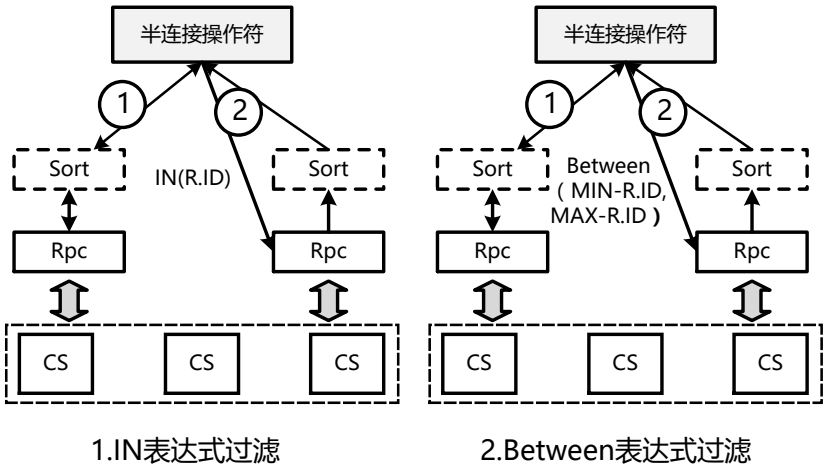


图 5.5 IN 表达式与 Between 表达式执行流程

以 IN 表达式的过滤方式为例，如图 5.5-1 中的圆圈标号所示，数据的过滤分为两个阶段：

1. 半连接操作符驱动其下一层操作符获取左表结果集，并将其连接属性列 ID 的所有值封装成 IN 表达式。
2. 半连接操作符将 IN 表达式序列化化后发送给右表的 RPC 模块，进行右表数据的过滤，最后将 S 表经过过滤后的数据返回半连接操作符。

Between 表达式的过滤流程与 IN 表达式相同，下面对两种表达式的优缺点进行分析。

IN 表达式优缺点：

OceanBase 中 IN 表达式的原理是：首先根据 IN 表达式的左值，即要被过滤的表所对应的的属性列，定位具体 Data Block，能定位 Data Block 的前提是属性

列为主键或者主键前缀；然后在 ChunkServer 上将对应的 Data Block 加载到内存中，利用 IN 表达式的过滤流程对数据进行过滤。

IN 表达式的优点在于其可以定位到具体的 Data Block，不需要将所有的数据都扫描进内存。缺点在于当需要过滤的数据变得很多时，定位到的 Data Block 会有大量的重复，意味着 ChunkServer 可能会将同一个 Data Block 多次的加载到内存中，随着数据量的增加，Data Block 的重复率也会增加，此时磁盘 I/O 的开销可能比全表扫描的 I/O 开销还要大。

因此 IN 表达式适合过滤较少的数据，当数据量增多时可以采取并行过滤的方式，将过滤数据拆分成多个 IN 表达式并行的对右表数据进行过滤。

Between 表达式优缺点：

Between 表达式的过滤方式相当于对指定范围内的数据进行扫描，通过 Between 表达式中的最大值与最小值，确定需要扫描的初始 Data Block 与结束 Data Block，然后将 Data Block 顺序的扫描进内存。

Between 表达式的优点在于顺序的将一定范围内的数据快速的加载进内存当中，并且可以有效的控制磁盘 I/O。但是其缺点在于，如果左表的连接属性列只有几个数值，但是在这几个数值范围内的右表数据却有几万、几十万甚至于几百万，这种情况下会有大量的无用数据被传回 MergeServer，造成数据网络传输开销的增加。

因此 Between 表达式适用于左表连接属性列的数据较多，并且右表连接属性列的值相对分散的情况，避免将大量无用的数据加载进内存。

5.3 连接查询优化框架中服务器硬件资源使用情况

内存与 CPU 的使用情况主要分为两类：计算节点的 CPU 和内存使用情况以及存储节点的 CPU 和内存使用情况。下面分别对这两类进行分析：

计算节点的 CPU 和内存使用情况：

计算节点上的主要任务为数据请求与处理以及数据的连接运算。这两个任务

将会频繁的使用 CPU。对 CPU 的使用情况表现在，不同的查询类型对于数据的请求处理任务与连接计算任务的并行度要求也不相同。而并行度主要体现在对 CPU 核心的利用上，并行度高说明各种任务线程的个数多。一旦任务线程的总体数目超过服务器 CPU 的核心数目时，就会发生资源争用与解决冲突的情况，这种情况下即使增加任务线程的个数，任务执行的效率也不会上升，并且可能由于系统调度的因素导致执行效率的下降。因此对于表的数据量较大、连接关系较多，并对查询效率有很高要求的查询请求，相关技术人员应为这类查询指定计算节点，将其与其他类型的查询分开处理。

对于内存的使用情况，主要体现在各个关系的结果集的大小，因为查询请求中所有的连接操作都要在同一个计算节点上进行处理，当结果集的大小超过了服务器内存的容量，就要将部分结果集暂时存存储到磁盘上，此时磁盘 I/O 的开销，会影响连接查询的效率。因此建议计算节点尽可能配置大的内存，还要求数据库系统对内存进行妥善的管理，防止内存碎片以及内存泄漏等情况的发生，影响内存的利用效率。

总之，计算节点对于 CPU 与内存的使用会随着不同的查询以及不同的负载动态的变化，系统硬件有一定的计算压力，因此对于计算节点所在服务器的 CPU 与内存的要求相对于其他节点更为严苛。

存储节点的 CPU 和内存使用情况：

存储节点的主要任务为数据提取与过滤。存储节点上会维护一个数据请求队列，用于接收所有计算节点的数据请求，同时维护一个数据处理任务线程池，用于具体的数据提取与发送，存储节点面临着与计算节点同样的问题，当存储节点的数据请求增多时，CPU 的计算能力成为影响响应时间的瓶颈。但是分布式数据库一般会保存多个数据副本，并存放在不同服务器上，因此对于单张表的查询压力就会分散到其他数据副本上，这样就会减少单台存储节点的任务处理压力。对于内存的使用情况，主要关注维护热点数据的缓存以及热点的索引结构。

总之，存储节点上的 CPU 与内存使用情况也会受到当前一段时间内的业务

类型以及负载的影响。因此在进行系统各个功能模块以及计算模块的设计时，要考虑到 CPU 与内存的使用情况，进行有效的负载均衡，防止单次查询过度的使用系统的硬件资源，导致其他查询的响应时间增加。

5.4 本章小结

本章给出了并行哈希连接与并行嵌套循环连接的设计方案，并对其实现流程进行详细的介绍。接着提出了并行半连接操作的设计方案，从半连接操作整体的执行流程、任务线程的执行流程、数据过滤方式等三方面对其进行详细的介绍。

并行哈希连接与嵌套循环连接可以快速的对大量的数据进行连接运算，并行的半连接操作可以有效的减少数据的网络传输，本文下一章将对连接查询优化框架的有效性从三个角度进行验证，包括：不同的数据过滤方式对响应时间的影响、不同并行度下连接算子的执行效率、不同数据量下半连接操作对连接查询效率的影响。

第六章 实验分析

本章采用从局部到整体的测试方式，首先，是不同的数据过滤方式对响应时间的影响；其次，对各连接算子的执行效率进行测试；然后，在不同的数据过滤方式下，测试半连接操作的响应时间；最后针对连接查询优化框架中的优化策略，从连接算子、半连接操作以及并行度三方面，进行综合的测试。

6.1 系统软件环境

本文实验使用 5 台服务器，OceanBase 的实验版本为 0.4.2，其中主控节点 RootServer（RS）与 UpdateServer（UPS）共用一台服务器，其余 4 台服务器分别部署 ChunkServer（CS）与 MergeServer（MS）。整个 OceanBase 数据库集群中有 3 台数据存储节点（CS），1 台查询处理引擎（MS）。实验环境的 OceanBase 集群物理拓扑如图 6.1 所示：

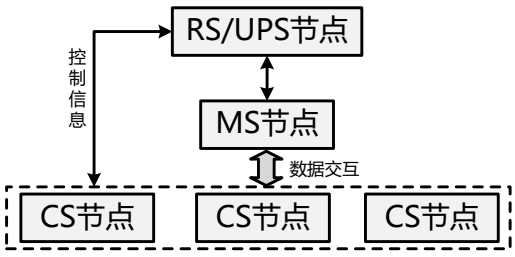


图 6.1 OceanBase 实验环境物理拓扑

本文实验的所有查询请求都由 MS 节点进行处理，三台 CS 节点上存储实验数据，RS 与 UPS 负责集群的管控与事务处理。

6.2 系统硬件环境

图 6.2 为服务器的硬件配置：

角色	CPU	内存	磁盘	网络
CS	6核12线程（Intel（R） Xeon（R） CPU E5-2620 V2 @ 2.10GHz）*2	64 GB	3T SSD	千兆网
UPS/RS	6核12线程（Intel（R） Xeon（R） CPU E5-2650 V3 @ 2.30GHz）*2	165 GB	1.5T SSD	千兆网
MS				

图 6.2 集群服务器配置

6.3 实验数据设置

本文测试用到的所有数据表的 schema 如图 6.3 所示：

属性名称	是否为主键	数据类型	数据大小（Byte）
ID	是	Int	4
Col1	否	Varchar	64
Col2	否	Varchar	64

图 6.3 测试表的 schema

所有实验数据都是由 Sysbench 的数据生成器生成。实验一共涉及到七张表，数据分布以及数据量如下：

表名	数据分布（主键）	数据量（行数）
R-1	连续	10万
R-2	连续	100万
R-3	连续	1000万
S-1	连续	10万
S-2	连续	100万
S-3	连续	1000万
S-4	离散	1000万

图 6.4 测试数据表信息

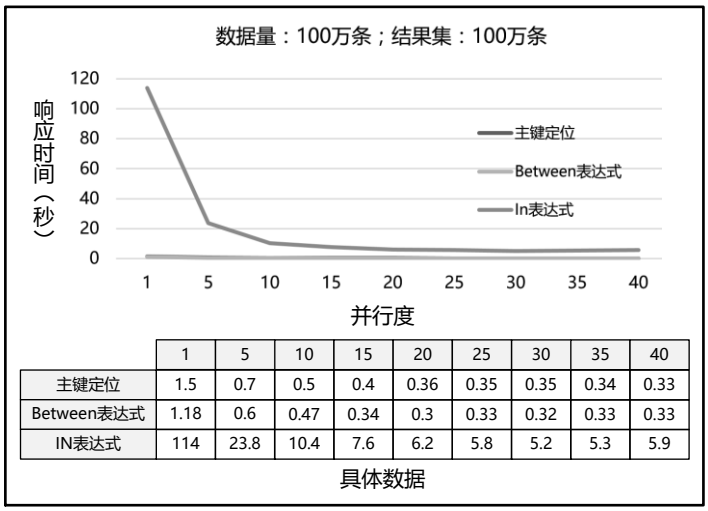
以 R-1 为例，数据分布的连续性是指 R-1 表的主键列 ID 的数据是按照升序排列的，离散性是指 ID 的值是无规律的，取值范围在 1~1 亿之间。

6.4 不同的数据过滤方式对响应时间的影响

实验目的：单表情况下并行度以及数据过滤方式对于查询响应时间的影响。

数据设置：使用表 R-2 作为测试表，结果集为 100 万，并发度分别为 1、5、15、20、25、30、35，数据过滤方式为主键定位、IN 表达式、Between 表达式。

实验结果：



	1	5	10	15	20	25	30	35	40
主键定位	1.5	0.7	0.5	0.4	0.36	0.35	0.35	0.34	0.33
Between表达式	1.18	0.6	0.47	0.34	0.3	0.33	0.32	0.33	0.33
IN表达式	114	23.8	10.4	7.6	6.2	5.8	5.2	5.3	5.9

具体数据

图 6.5 单表数据过滤的响应时间

实验结果表明基于主键定位与 Between 表达式的数据过滤方式在响应时间上要远远优于基于 IN 表达式的数据提取方式。原因在于，主键定位是根据

OceanBase 的三级索引结构直接定位数据块，Between 表达式则是在一定范围内对表的数据进行扫描。假设一张表有 100 个 Data Block，Between 表达式可能将其中 80 个 Data Block 加载到内存中，其数据过滤效率甚至比主键定位要高，因为主键定位的方式可能会导致缓存缺失而将同一块 Data Block 重复的加载到内存中。至于 IN 表达式所花费的时间：一方面可能需要将所有涉及到的 Data Block 重复的加载进内存中，对于重复率，本文实验没有进行统计，实验结果表明，100 万条数据规模下的 Data Block 重复加载率还是很高的，IN 表达式效率的问题与 OceanBase 的实现有关，因此这也是一点需要改进的地方；另一方面，OceanBase 中实现的 IN 表达式需要将所有待过滤的数据与 IN 表达式中的数据进行一一比对，如果 IN 表达式中有 1 万条数据，待过滤的数据有 1 万条，则要比对 1 亿次，这个计算代价是很高的。因此本文未来的工作中会有对 IN 表达式进行改造。

从图 6.5 可以得出，随着并行度的增加，响应时间的总体呈下降趋势，因此通过并行的方式来提高数据的过滤速度，对减少连接查询的响应时间是有效的。

6.5 连接算子的执行效率

实验目的：在不同数据量上，测试连接算子的执行效率对响应时间的影响。

数据设置：使用表 R-1、R-2、R-3、S-1、S-2、S-3 作为测试表。查询语句形如：SELECT * FROM R JOIN S ON R.ID=S.ID。连接计算的并行度设置为 1、10、20、30。

实验结果：

图 6.6 所示为在不同的并行度下，各连接算法的执行效率。在并行度为 1 的情况下，Nestloop-Join 在处理 100 万条数据的连接计算时花费了 78 秒左右的时间，由于与其他连接算子的响应时间相差太大，因此没有在图中完全显示。随着并行度的增加，各连接算子的响应时间都有所降低，当数据量在 10 万条左右时，三种连接算子的执行效率区别不大。但是当数据量在百万级别时，Nestloop-Join 的效率明显下降，因为随着数据量的增多，连接运算中的判断次数也呈爆炸式增

长，100 万条数据要进行一亿次判断。

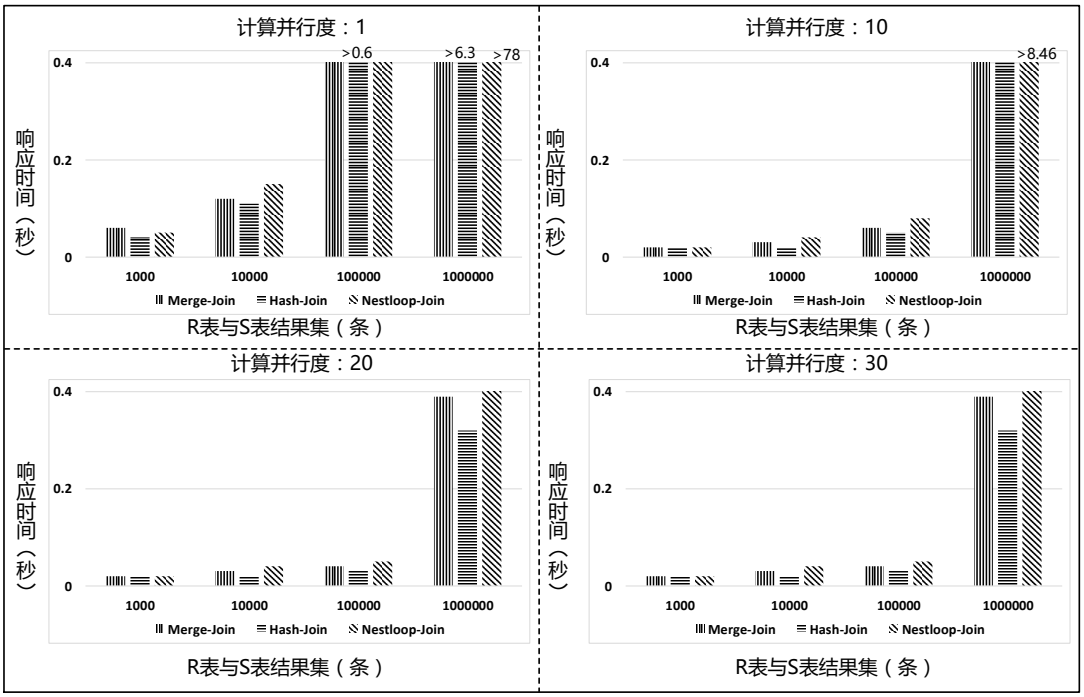


图 6.6 不同并行度下连接算法的执行效率

实验结果表明，当数据量在 10 万条左右的情况下，三种连接算子的执行效率相差不是很大。但随着数据量的不断增加，Nestloop-Join 的执行效率显著下降，然而对于非等值连接，Nestloop-Join 又是必不可少的的连接算子。不同的连接算子有其特定的应用场景，并且在并行技术的帮助下，连接算子的执行效率进一步提升，可以有效的减少查询的响应时间。

6.6 半连接对查询效率的影响

名词解释：Merge-Join 为 OceanBase 原生的连接计算操作，Seimi-In-Join 为改造后使用半连接并且数据过滤方式为 IN 表达式的 Merge-Join，同理 Semi-Between-Join 为改造后使用半连接并且数据过滤方式为 Between 表达式的 Merge-Join。

密度（Density）：|Result-Set(R)| 表示 R 表结果集的数据量，

|Between(MIN(R.ID),MAX(R.ID))|表示 S 表的 ID 列在 MIN(R.ID)与 MAX(R.ID)范围之间的数据量。

$\mathbf{Density} = \frac{|\text{Result-Set(R)}|}{|\text{Between}(\text{MIN(R.ID)},\text{MAX(R.ID)})|}$表示 S 表理论上需要过滤的数据量与实际返回 MergeServer 的数据量之比。例如 Density=0.1，并且 R 表经过处理后的结果集|Result-Set(R)|为 100 条数据，也就是 R 表 ID 的值有 100 个或者少于 100 个， S 表就需要将 ID 列在 MIN(R.ID)与 MAX(R.ID)范围之间的所有数据返回 MergeServer，如果过这个数据量是 1000，说明 S 表实际返回 1000 条数据，而只有不到 100 条记录可以产生连接关系，此时的 Density 是 0.1。这说明密度越小，传回 MergeServer 的无用数据就越多。

实验目的：在不同数据量下，对比 OceanBase 原 Merge-Join 与引入半连接操作后的 Merge-Join 在响应时间上的差距。并且在连接算子为 Merge-Join 的前提下，测试 IN 表达式与 Between 表达式的过滤方式，在不同密度的情况下，对半连接操作效率的影响。

数据设置：使用 R-1、S-1、S-2、S-3、S-4 作为测试表。为了与 OceanBase 原 Merge-Join 进行性能比较，数据提取以及连接计算方面的并发度都默认为 1，连接算法采用排序归并连接。

查询形如：“SELECT * FROM R JOIN S ON R.ID=S.ID WHERE R.ID<=？”，其中结果集的大小是通过过滤条件 R.ID≤(100 or 1000 or 10000 or 100000) 来控制的，此外 S 表上无任何过滤条件。

实验结果：

图 6.7 所示为 S 表数据量为 1 万、10 万、100 万和 1000 万，并且 R 表结果集分别为 100、1000、10000、10000 时，三种连接操作的响应时间。实验结果表明 Merge-Join 的响应时间与 R 表的结果大小和 S 表的数据量的大小都有所关联，随着 R 表结果集与 S 表数据量的增加，Merge-Join 的响应时间也在显著提升，原因在于 OceanBase 原生的 Merge-Join 并没有使用 R 表的结果集对 S 表的数据进行过滤，导致 S 表中很多不能产生连接关系的数据也通过网络传回了 MergeServer，

造成了响应时间的增加。而 Semi-In-Join 与 Semi-Between-Join 则利用半连接操作以及 IN 表达式和 Between 表示的过滤方式，先将不会产生连接关系的数据在 ChunkServer 上过滤掉，然后将可以产生连接关系的数据传回 MergeServer，减少了网络的传输，降低了响应时间。

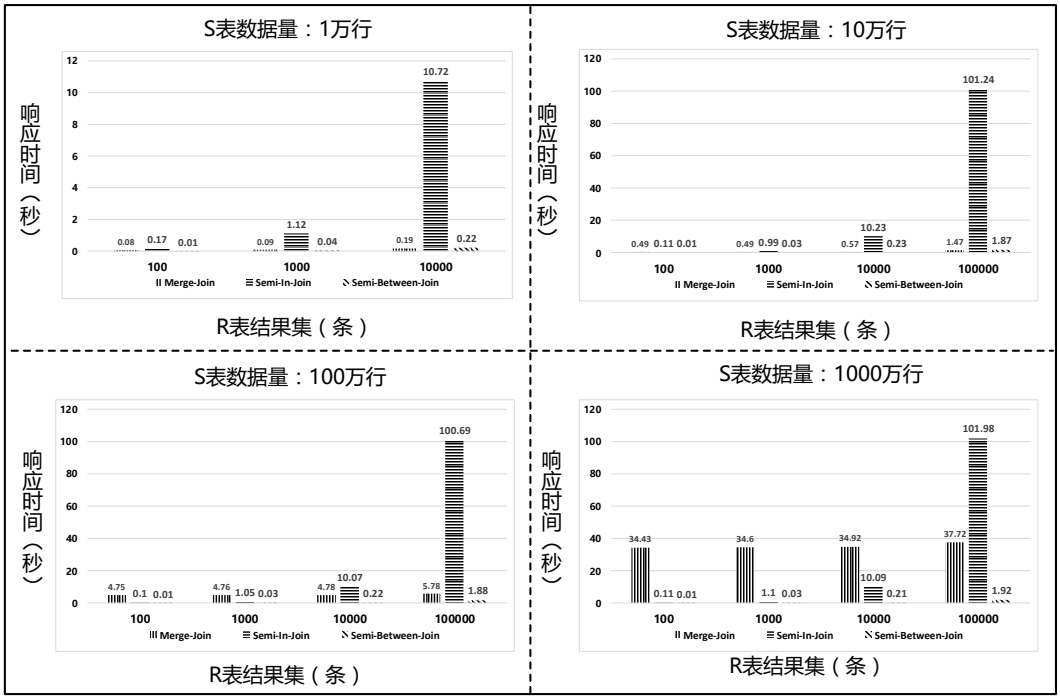


图 6.7 Merge-Join、Semi-In-Join 与 Semi-Between-Join 响应时间对比

图 6.7 显示，在 S 表数据量大于 100 万，R 表结果集小于 1000 的情况下，Merge-Join 的响应时间要高于 Semi-In-Join 与 Semi-Between-Join。同时 Semi-Between-Join 的响应时间要低于 Semi-In-Join 的响应时间，原因在于 Semi-In-Join 需要将 IN 表达式中的所有数据所涉及的 Data Block 都加载到内存中，可能发生缓存未命中的情况，继而将相同的 Data Block 重复的加载内存中。而 Semi-Between-Join 则只需要把一定范围内的 Data Block 按照顺序加载进内存即可，从磁盘访问的角度上来看，Semi-Between-Join 要比 Semi-In-Join 少很多，特别地，当需要过滤的数据增加时，Semi-In-Join 的过滤效率会直线下降。

但是 Semi-Between-Join 在密度减小的情况下，响应时间就会明显高于 Semi-In-

Join，图 6.8 中显示了在不同密度下 Semi-In-Join 与 Semi-Between-Join 的响应时间对比情况。其中密度设置为 0.1、0.01、0.001 和 0.0001，实验使用表 S-4，并且由于 S 表的数据量限制，在密度为 0.001 与 0.0001 时有些数据无法显示。结果表明，随着密度的不断减小，Semi-In-Join 的性能要远远优于 Semi-Between-Join，原因在于随着密度的减小，不会产生连接关系的数据就越多，最终导致响应时间的增加。因此选择哪种数据过滤方式，应根据具体的场景来动态的调整，如果数据比较连续则使用 Between 表达式进行过滤，如果比较离散则使用 In 表达式进行过滤。

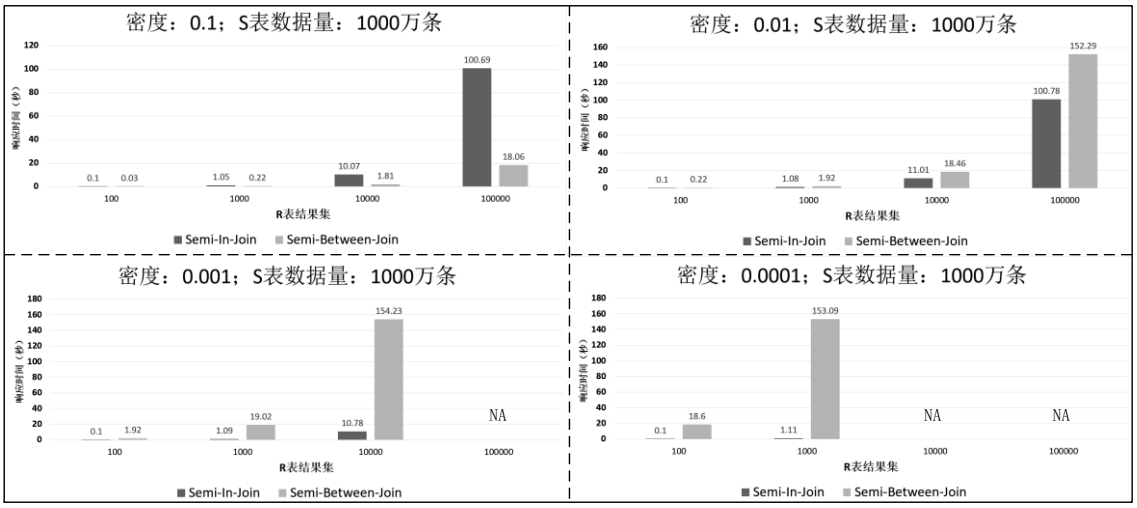


图 6.8 不同密度下 Semi-In-Join 与 Semi-Between-Join 的响应时间对比

综上所述，半连接操作可以有效的减少数据的网络传输，无论是使用 IN 表达式的数据过滤方式，还是 Between 表达式的数据过滤方式，随着 S 表数据量的不断增加，采用半连接操作的 Semi-In-Join 与 Semi-Between-Join，比 Oceanbase 原有的 Merge-Join 在查询效率上都有显著的提高。

6.7 综合性能测试

实验目的：通过对并行度、连接算子、半连接操作这三方面的组合测试，验证这三方面对连接查询效率的影响。

数据设置：本节测试包括两张表：R-3 和 S-3，结果集设置为 100 条、1000 条、1 万条 10 万条。查询语句形如：“SELECT * FROM R JOIN S ON R.ID=S.ID WHERE R.ID<=？”，因此结果集的大小主要通过控制过滤条件来实现。

实验内容：分为两个，综合测试 1 与综合测试 2。

综合测试一的测试内容为：在使用半连接的前提下，随着结果集的增加，并行度的提高，连接算子（Hash-Join 与 Merge-Join）以及不同的数据过滤方式对响应时间的影响。

综合测试二的测试内容为：在使用半连接的前提下，随着结果集的增加，并行度的提高，连接算子（Nestloop-Join 与 Merge-Join）以及不同的数据过滤方式对响应时间的影响。

实验结果：

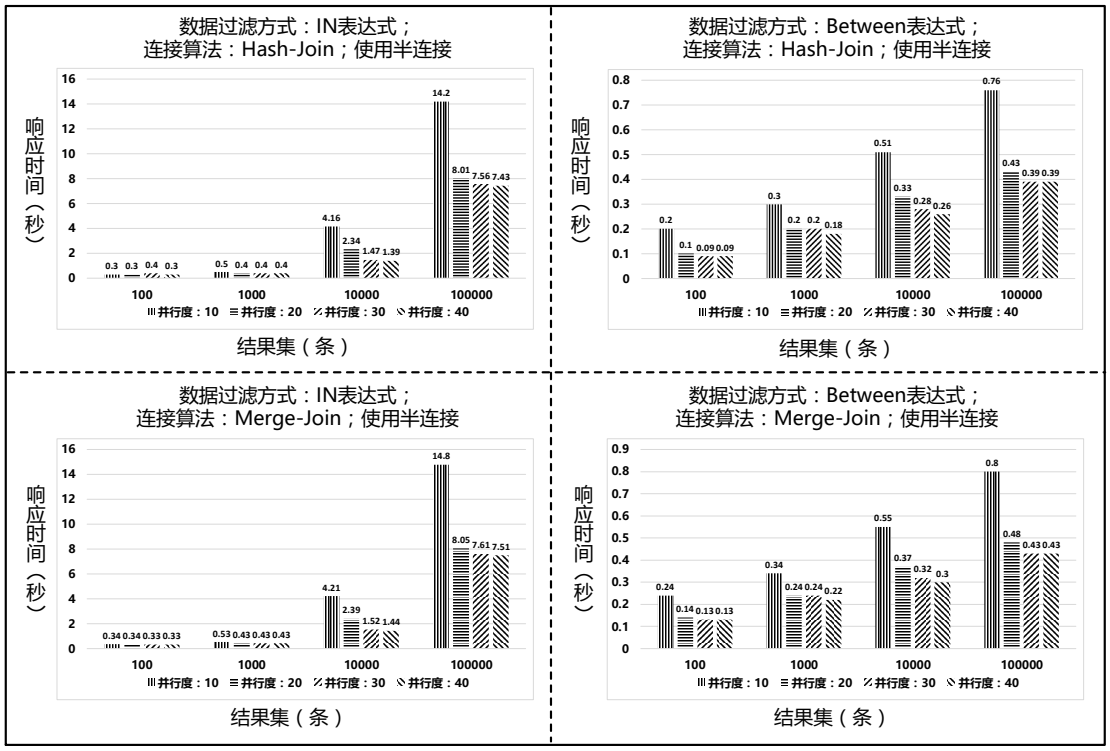


图 6.9 综合测试一

图 6.9 为综合测试一的实验结果汇总，可以看出随着结果集的增加，数据库

连接查询总体的响应时间也是在不断增加的，接下来从三个方面对连接查询的效率进行分析：并行度、数据过滤方式、连接算子。

并行度，随着并行度的提高，Hash-Join 与 Nestloop-Join 的响应时间都在逐渐降低，特别地，当结果集不断变大时响应时间的下降速度也随之加快，但是当并行度超过 20 后响应时间的变化就变得不是非常明显，原因在于数据库系统的计算能力受制于服务器 CPU 的核心数目，而用于本文实验的服务器，在采用超线程技术后，可用的核心数目为 24 个，当并行度超过 20 后就有明显的调度以及资源争用问题，一方面原因在于 CPU 核心数目的限制，另一方面本文也没有将任务线程与相应的 CPU 核心进行绑定，因此可能出现多个任务线程共用一个核心的情况。

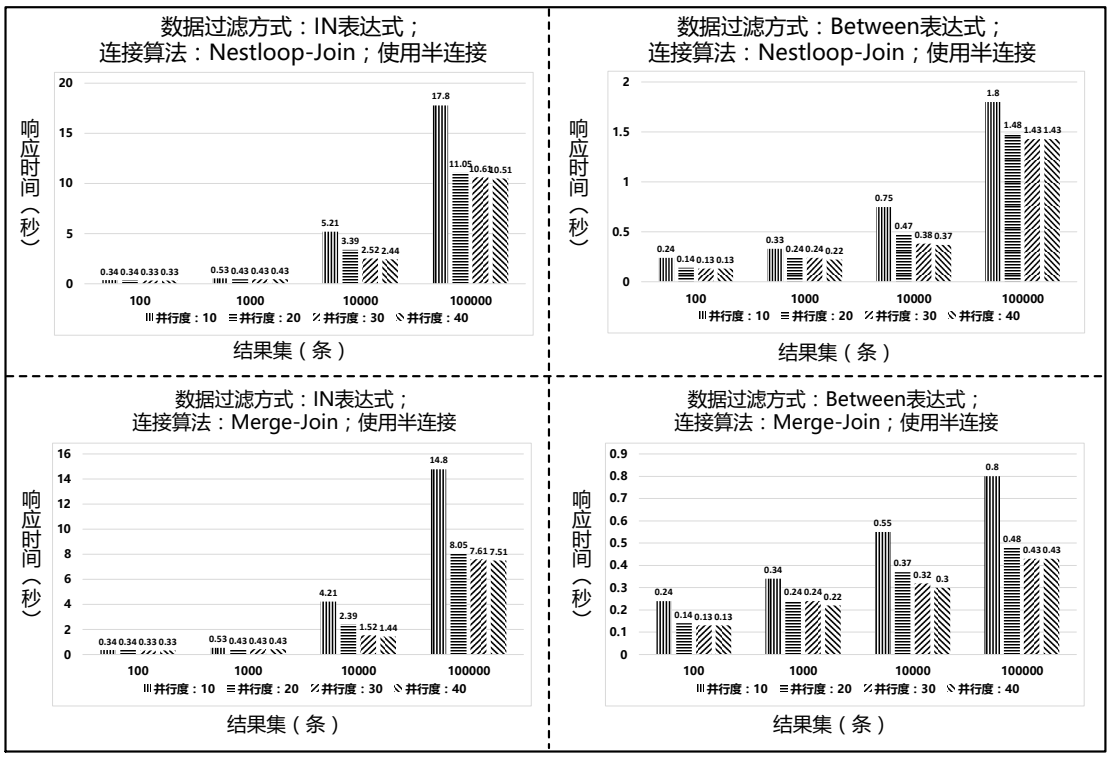


图 6.10 综合测试二

数据提取方式，可以看出 IN 表达式的数据过滤方式与 Between 表达式的数据过滤方式在结果集处于 1000 条左右的范围内时，连接查询执行效率是相差无

几的。但是一旦超过 1000 条记录，IN 表达式的数据过滤方式使得响应 时间急剧上升，虽然随着并行度的增加，响应时间有一定程度的下降，但是相比于 Between 表达式还是有很大的差距。而 IN 表达式过滤的优点在于当过滤数据量在 1000 条到 10000 条之间时，并且数据分布离散的情况下，其响应时间整体上要优于 Between 表达式，因为同样数据量的情况下，Between 表达式过滤的数据范围可能会被无限的放大，例如：两个记录 1 和 10000，使用 IN 表达式只需将 1 与 10000 过滤出来即可，但是 Between 则需要将 1 到 10000 之间的所有数据加载进内存，以上种情况下，通过 IN 表达式来过滤数据是比较好的选择。

连接算子，选择合适的连接算子并通过并行技术提高连接计算的效率，是连接查询优化框架中的一个重要优化策略，由于实验的结果集设置等因素，用于连接计算的数据量较小，在数据过滤方式相同的情况下，Hash-Join 与 Merge-Join 的响应时间相差无几，然而通过实验结果可以预见，随着结果集大小的增加，连接的计算量也在增加，因此 Hash-Join 的执行效率要好于 Merge-Join。

图 6.10 为综合测试二的实验结果汇总，综合测试一的不同之处在于连接算子使用了 Nestloop-Join，与连接算子的执行效率实验结果一致。Nestloop-Join 在数据量增加的情况下，连接效率也显著下降，特别地，当数据过滤方式为 IN 表达式时，连接查询的响应时间会进一步增加。

实验结果表明，对比 OceanBase 原有的 Merge-Join，连接查询优化框架中提到的优化策略对于连接查询的效率提升有很大的帮助，不同的并行度、不同的连接算子、是否使用半连接对查询的响应时间也有不同程度的影响。

6.8 本章小结

本章中，对本文提出的连接查询优化框架中的优化策略模块所涉及到三点优化方向，进行了充分的实验。首先测试不同的数据过滤方式对响应时间的影响，得出了随着并行度的增加，响应时间呈下降趋势的结论。其次对几种连接算子的执行效率进行测试，得出在并行技术的帮助下，连接算子的执行效率会进一步提

升。然后通过对比引入半连接后的各种连接算子，在响应时间上的差距，得出随着右表数据量的不断增加，采用半连接操作的 Semi-In-Join 与 Semi-Between-Join，比 Oceanbase 原有的 Merge-Join 在查询效率上有显著的提高。最后本文对并行度、连接算子、半连接操作这三点优化方向进行综合的测试，得出不同的并行度、连接算子以及半连接操作的选用对查询的响应时间都有不同的影响，因此在本文提出的连接查询优化框架中，优化策略模块中的优化策略应根据具体的应用进行动态的调整，以满足特定的性能需求。

第七章 总结与展望

数据库系统经过几十年的发展，已经渗透到社会的各行各业，随着科技的进步，互联网逐渐走进了人们的生活。互联网应用的背后是支撑其庞大数据存储与管理需求的数据库系统，然而在数据量爆发式增长的今天，传统数据库在面对 TB、ZB 级别的数据规模时已不堪重负。基于分布式存储的数据库，以其良好的扩展性，有效的解决了海量数据的存储问题。但是由于其架构设计的特点，数据分散存储于集群中的各节点，节点之间通过网络进行数据的交互，数据访问可能是跨机房、甚至是跨地域的，在这种情况下，网络传输开销将极大的影响查询的效率，对于某些复杂查询，尤其是连接查询，数据库很难有好的性能表现。

因此本文在分布式存储架构下，对连接查询进行充分的研究，总结出影响连接查询效率的几点因素：数据的本地提取、数据的网络传输以及连接算法的执行效率。然后针对这三方面影响因素，在分布式存储架构的基础上，提出了分布式数据库连接查询优化框架，可以有效的降低查询响应时间。下面为本文详细的研究内容：

1. **相关背景介绍**。首先对传统的分区数据库进行简单的介绍，以 Mysql Sharding 架构为例，分析其优缺点，并且对 SQL-on-Hadoop 类型的基于分布式存储的开源关系型数据库 Trafodion 进行简单的介绍，着重强调了其查询处理方面的特点。然后介绍了传统数据库中的查询优化过程是如何对查询进行优化的。最后介绍了目前几种主流的连接算法，以及在分布式环境下，用于减少网络传输开销的半连接操作。
2. **基于分布式存储的数据库架构与典型分布式数据库 OceanBase**。首先给出了基于分布式存储的数据库架构，并对架构中的各个模块进行详细的介绍，然后在分布式存储架构的基础上，分析影响连接查询效率的几点因

素。最后对典型的分布式存储架构数据库 OceanBase 以及其查询处理节点 MergeServer 进行了详细的介绍，分析其连接查询方面的优缺点。

3. **提出了有效减少响应时间的分布式连接查询优化框架**。针对影响连接查询效率的几点因素，提出具体的优化策略，从并行度、连接算子、半连接操作三方面对连接查询进行优化，然后提出了分布式连接查询优化框架，利用优化策略，采用并行技术，可以有效的减少响应时间。最后在 OceanBase 的基础上，给出优化框架中各模块详细的设计方案，并付诸实现。
4. **设计并实现了并行的哈希连接、嵌套循环连接以及半连接操作**。分别给出详细的设计方案和具体的实现流程，分析数据过滤方式：IN 表达式与 Between 表达式的优缺点，并给出其适应场景。
5. **在自定义数据上进行了充分的实验**，验证了分布式数据库连接查询优化框架的有效性与效率。

以上为本文的主要研究内容与贡献，以下两点为本文的未来工作：

1. 本文提出的分布式连接查询优化框架对优化过程中的连接方式进行了限制，规定为内连接，并且查询计划的树形规定为左深树。原因在于内连接中表的连接顺序可以任意的改变，可以使用结果集较小的表来过滤数据量较大的表；查询计划为左深树，就将每个连接操作的顺序进行了固定，优点在于在优化过程不需要考虑连接操作的顺序对优化效果的影响，缺点在于可能在最初选择了不能产生最优查询计划的树形。未来工作，将研究如何在外连接的情况下，针对不同的树形，在优化策略中增加相应的处理流程，动态的对连接查询进行优化。
2. 本文设计并实现的各种连接算子以及半连接操作中，用到了并行计算的技术，但是没有对计算资源的合理分配做进一步的研究，可能导致资源的浪费、甚至于过度使用计算资源使得单点负载过高，对查询的整体效率与并发度产生影响，因此在未来工作中，将研究如何做到资源的合理利用，既

能保证单次查询的响应时间，又能相应的提高系统整体的并发度。

总之，在未来的工作中，我们将会对连接查询优化框架中的细节进行完善，使之能适应更多的应用场景，并设计更多的优化策略，进一步提升连接查询的效率，扩大连接查询优化框架的适用范围，最终使之成为适用性更强的优化框架，普遍应用于实际的生产环境。

参考文献

[1] Tautges T J. Mesh Oriented dataBase[J]. 2004.

[2] 周宇葵, 杜方冬.现代图书情报技术. 数据库发展之现状[J], 2000, 第 16 期，58-59.

[3] Codd E F. A relational model of data for large shared data banks[M]// Pio neers and Their Contributions to Software Engineering. Springer Berlin Hei delberg, 2001:377-387.

[4] Adve S V, Gharachorloo K. Shared Memory Consistency Models: A Tutorial[J]. Computer, 1996, 29(12):66-76.

[5] Borr, Andrea. Robustness to Crash in a Distributed Database: A Non Shared-Memory Multi-Processor Approach.[J]. Proc Vldb, 1984:445-453.

[6] 2016 global Internet, social media, mobile device penetration Report.2016. http://wearesocial.com/uk/[Online;accessed 23-march-2017]

[7] 2012 white paper.International Data Corporation.http://www.idc.com.cn/ [Online;accessed 23-march-2017]

[8] Vora M N. Hadoop-HBase for large-scale data[C]// International Conference on Computer Science and Network Technology. IEEE, 2011:601-605.

[9] Chodorow K. MongoDB: The Definitive Guide[M]. O'Reilly Media, Inc. 2010.

[10] Moore G E. Cramming More Components Onto Integrated Circuits[J]. Proceedings of the IEEE, 1998, 86(1):82-85.

[11] 阳振坤.OceanBase 关系数据库架构.华东师范大学学报:自然科学版，(5):141-148,2014

[12] Loesing S, Pilman M, Etter T, et al. On the Design and Scalability of Distributed Shared-Data Databases[C]// Eidgenössische Technische Hochschule Zürich,

2015:663-676.

[13] Jin B G. Computer hardware: US, USD432533[P]. 2000.

[14] Trafodion .http://trafodion.incubator.apache.org/.[Online;accessed 23-march-2017]

[15] Ozsu M T, Valduriez P. Principles of distributed database systems[M]. 清华大学出版社, 2002.

[16] Rio Yunanto S K, weni, wulandari, et al. Database architecture[M]// Inside Relational Databases with Examples in Access. Springer London, 2007:168-177.

[17] Natkovich O, Cao J, Gates A. Clustered query support for a database query engine[J]. 2010.

[18] Gupta A W, Fachan N, Mckelvie S J, et al. Database system with database engine and separate distributed storage service[J]. 2016.

[19] Ghemawat S, Gobioff H, Leung S T. The Google file system[C]// ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, Ny, Usa, October. DBLP, 2003:29-43.

[20] Junqueira F, Reed B. ZooKeeper: Distributed Process Coordination[M]. O'Reilly Media, Inc. 2013.

[21] 分布式数据库. 作者：郑振楣. 出处：科学出版社 1998

[22] 汤子瀛. 计算机网络技术及其应用[M]. 电子科技大学出版社, 1999.

[23] Amza C. Database Middleware[M]. Springer US, 2009.

[24] Dwork C, Nissim K. Privacy-Preserving Datamining on Vertically Partitioned Databases[C]// Advances in Cryptology - CRYPTO 2004, International Cryptologyconference, Santa Barbara, California, Usa, August 15-19, 2004, Proceedings. DBLP, 2004:528-544.

[25] Widenius M, Axmark D P. Mysql Reference Manual[J]. Dec 2009 - World Bank, Washington, 2002(4).

[26] Seidman C, Smith P. MySQL: The Complete Reference[M]. McGraw-Hill, Inc.

2003.

[27] Haerder T, Reuter A. Principles of transaction-oriented database recovery[J]. Acm Computing Surveys, 1983, 15(4):287-317.

[28] Gray J. The transaction concept: virtues and limitations (invited paper)[C]// International Conference on Very Large Data Bases. VLDB Endowment, 1981:144-154.

[29] Chang L, Karin M. Mammalian MAP kinase signalling cascades[J]. Nature, 2001, 410(6824):37-40.

[30] Lawler E L, Wood D E. Branch-And-Bound Methods: A Survey[J]. Operations Research, 1966, 14(4):699-719.

[31] Kołaczkowski P, Rybiński H. Automatic Index Selection in RDBMS by Exploring Query Execution Plan Space[J]. Studies in Computational Intelligence, 2009, 223:3-24.

[32] Jarke M, Koch J. Query Optimization in Database Systems[J]. Acm Computing Surveys, 1984, 16(16):111-152.

[33] Farrar C M, Leslie H A, Celis P, et al. Database query cost model optimizer: US, US 6330552 B1[P]. 2001..

[34] 邓亚丹, 景宁, 熊伟. 多核处理器中基于 Radix-Join 的嵌套循环连接优化[J]. 计算机研究与发展, 2010, 47(6):1079-1087.

[35] Imai H, Nakano M, Kitsuregawa M. Consideration on Parallel Hashjoin in Distributed Shared Memory Architecture[C]// Information Processing Society of Japan (IPSJ), 1996:15-16.

[36] SUN Wenjun LI Jianzhong Institute of Information Research Heilongjiang University Harbin. SortMergeJoin Algorithm Revisited[J]. Journal of Software, 1999.

[37] Lander G C, Stagg S M, Voss N R, et al. Appion: An integrated, database-driven

pipeline to facilitate EM image processing[J]. Journal of Structural Biology, 2009, 166(1):95-102.

[38] 杨晓强, 朱卫东. ORACLE 的查询优化[J]. 计算机系统应用, 1998, 7(8):41-43.

[39] Nakayama M, Kitsuregawa M, Takagi M. Hash-Partitioned Join Method Using Dynamic Destaging Strategy[C]// Fourteenth International Conference on Very Large Data Bases, August 29 - September 1, 1988, Los Angeles, California, Usa, Proceedings. DBLP, 1988:468-478.

[40] Smith A J, Saavedra R H. Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes[J]. IEEE Transactions on Computers, 1995, 44(10):1223-1235.

[41] DENG YaDan, JING Ning, XIONG Wei,等. Hash Join Query Optimization Based on Shared-Cache Chip Multi-Processor 基于共享 Cache 多核处理器的 Hash 连接优化[J]. 软件学报, 2010, 21(6):1220-1232.

[42] 刘光霆. ORACLE 中 SQL 查询优化技术[J]. 微型电脑应用, 2008, 24(5):59-61.

[43] 李毅. 数据库管理系统中基于 B 树索引系统算法[D]. 武汉大学, 2003.

[44] Blanco-Cano X, Russell C T, Strangeway R J. The Io mass-loading disk: Wave dispersion analysis[J]. Journal of Geophysical Research Atmospheres, 2001, 106(A11):26261-26276.

[45] Valduriez P, Gardarin G. Join and Semijoin Algorithms for a Multiprocessor Database Machine[J]. Acm Transactions on Database Systems, 1984, 9(1):133-161.

[46] Chuan L I. Research on & Improvement of SDD-1 Algorithm[J]. Journal of Xian Aerotechnical College, 2012, 303(7):H784-H794..

[47] Russell S J, Norvig P. Artificial Intelligence: A Modern Approach[M]. 人民邮电出版社, 2002.

[48] Gardarin G, Valduriez P. Join and semijoin algorithms for a multiprocessor database machine[C]// ACM Trans. on Database Systems. 1984:133-161.

[49] 杨传辉. OceanBase 高可用方案[J]. 华东师范大学学报(自然科学版), 2014(5):173-179.

[50] Han J, Haihong E, Le G, et al. Survey on NoSQL database[C]// International Conference on Pervasive Computing and Applications. IEEE, 2011:363-366.

[51] Kumar R, Gupta N, Charu S, et al. Manage Big Data through NewSQL[C]// National Conference on Innovation in Wireless Communication and NETWORKING Technology. 2014.

[52] Kimball R, Caserta J. The Data Warehouse ETL Toolkit[J]. 2004. On the Design and Scalability of Distributed Shared-Data Databases.

[53] Srinivasan R. RPC: Remote Procedure Call Protocol specification: Version 2[J]. Sun Microsystems, 1995, 11(3):5531.

致谢

时光荏苒，岁月如梭，三年的硕士生涯已接进尾声，而入学时的场景还历历在目。回顾这几年的研究生时光，既漫长又短暂，其中充满了辛酸，更多的是收获和成长。三年来，感谢陪我一起度过美好时光的每位尊敬的老师和亲爱的同学，正是由于同学们无私的帮助，我才能迎难而上，正是老师们殷切的教导，我才能冲破生活与学习上的迷茫，勇敢前行。研究生三年是艰辛的三年，是快乐的三年，是拼搏的三年，是丰收的三年。研究生三年凝聚了我的汗水，见证了我的拼搏。更加庆幸的是我来到了一个很好的学习环境，遇到了许许多多的良师益友，给我了很多的指引和帮助，使我能够顺利地完成学业，再此谨向他们表示最由衷的感谢。

首先，我要特别感谢我的恩师高明教授。在过去的三年里，高明老师无论是在生活上还是科研学习上都给予我很大的帮助，犹记得当初我发表第一篇科研论文时，高明老师孜孜不倦的指导我修改论文，从思想到行文，一步步，手把手教我如何完成一篇高质量的论文，这是我人生中一笔宝贵的财富。高明老师为人师表、以身作则，教导我为人处世的道理，教给我正确的人生观、价值观，教会我从不同的角度分析问题、解决问题。我所取得的所有成绩都凝聚着恩师的汗水和心血，恩师开阔的视野、严谨的治学态度、精益求精的工作作风，深深地感染和激励着我，在此谨向高明教授致以衷心的感谢和崇高的敬意。

其次，我要感谢我的论文指导老师胡卉齐老师，胡老师认真负责，并指导我对论文进行细致的修改，从论文的选题到论文的结构组织都非常用心、事无巨细，本篇论文就是在胡老师的悉心指导下完成的，胡老师无论是在学业上还是生活上都给予我无微不至的关怀，在此我表示衷心的感谢与祝福。

同时，我还要感谢实验室的其他老师们。感谢宫学庆老师在实习过程给予的

生活上与科研上的帮助，感谢张蓉老师在科研项目开发过程中的指导与生活中的帮助，感谢周傲英老师每次振聋发聩的演讲与法人深省的谆谆教诲，感谢王晓玲、何晓丰、金澈清、蔡鹏等各位老师的知遇之恩与传道受业解惑之情。

另外，我还要感谢与我一同生活、学习的小伙伴们。感谢他们为我们的科研项目作出的努力，感谢他们帮助我顺利的完成毕业论文。感谢刘柏众、祝君、余晟隽、熊辉、钱招明等交行项目组的小伙伴，一年的生活与工作使我们之间建立起深厚的友谊与战友之情。感谢郭进伟博士、李宇明博士、周欢博士、朱涛博士、储佳佳博士、朱燕超博士、段慧超博士等实验室的小伙伴，感谢他们平时的帮助与科研上的指点。三年研究生生活中因为有了你们，我的经历才变的丰富多彩，有了你们我的生活才充满了乐趣。

最后，我要感谢的我的父母和姐姐，以及所有给予我支持与关心的家人们，感谢父母一直以来的教诲，感谢姐姐一直以来的教导，使我在有限的人生中少走了很多弯路，没有你们的鼓励与帮助就没有今天的我，谢谢你们。

谨以此文向所有关系和帮助我的人们表示衷心的感谢！

王雷

二零一七年四月九日

发表论文和科研情况

■ 已发表或录用的论文

[1] 论文：**Lei Wang**, Ming Gao, Rong Zhang, Cheqing Jin, Aoying Zhou. Computing Probability Threshold Set Similarity on Probabilistic Sets. Web-Age Information Management. 2015(16).

[2] 论文：**王雷**, 钱招明, 郭进伟, 张蓉，周敏奇，高明，钱卫宁. OceanBase 数据库监控系统[J]. 中国数据库学术会议，计算机应用, 2016, 36(S1):237-239.

[3] 论文：钱招明, **王雷**, 余晟隽, 宫学庆. 分布式系统中 Semi-Join 算法的实现[J]. 华东师范大学学报(自然科学版), 2016(5):75-80.

■ 参与的科研课题

[1]国家自然科学基金，集群环境下基于内存的高性能数据管理与分析，2014-2018，参加人

[2]国家高技术研究发展计划（863 计划）课题，基于内存计算的数据库管理系统研究与开发，2015-2017，参加人