西安工业大学
XI'AN TECHNOLOGICAL UNIVERSITY

# 本科毕业设计(论文)

## 题目：大规模分布式分析型数据库中的Pipeline执行引擎实践研究


院　（系）：计算机科学与工程学院

专　　业：计算机科学与技术

班　　级：18060314

学　　生：赵长乐

学　　号：18030513121

指导教师：杨国梁


2023 年 04 月

西安工业大学
XI'AN TECHNOLOGICAL UNIVERSITY

# 本科毕业设计(论文)

## 题目：大规模分布式分析型数据库中的Pipeline执行引擎实践研究


院　（系）：计算机科学与工程学院

专　　业：计算机科学与技术

班　　级：18060314

学　　生：赵长乐

学　　号：18030513121

指导教师：杨国梁


2023 年 04 月

# 大规模分布式分析型数据库中的 Pipeline 执行引擎实践

## 摘 要

在面对海量数据增长的互联网新业态，OLAP（联机分析处理）数据库对于互联网企业的重要性愈发凸显。它通过多维数据模型、数据立方体计算和聚合查询优化等技术，为用户提供高效的数据分析和决策支持能力。在实践中，相对于OLTP（联机事务处理）场景中重要的高并发点查，OLAP 类数据库的使用场景往往更为复杂，需要处理大表、宽表下的复杂聚合查询。查询场景通常涉及多个表，需要对特定列进行全表扫描。因此，OLAP 类数据库通常使用列式存储代替传统 OLTP 场景中使用的行式存储，通过提供多种聚合优化、实现物化视图、表达式复用、Runtime Filter 等方式优化性能。

查询引擎是数据库系统运转的核心。随着 OLAP 场景需要处理的数据量快速攀升，传统的 Volcano 引擎由于无法灵活调度查询、难以实现资源隔离，加之本身因为无法规避的虚函数调用而产生的额外性能开销等原因，已经成为了业务性能瓶颈。这种前提下，新的 Pipeline 查询执行引擎遵照“小数据块的数据驱动”原则，将查询计划树拆分为多个非阻塞的数据流，高度契合现代 CPU 架构的缓存与指令流水线加速需求，实现了更好的查询性能和更加灵活的调度控制，尤其适用于大规模数据集的分析查询。

本论文实现了一种高性能、可靠的 Pipeline 引擎及其优化，并成功应用于 Apache Doris 这一著名的开源 OLAP 数据库中，取得了显著的性能改善和功能增强。

依靠本论文实现的 Pipeline 执行引擎带来的优化，Apache Doris 在 OLAP 数据库领域事实性能标准 Clickbench 上取得了世界第一的性能表现。

**关键词：数据库；OLAP；Pipeline 执行引擎；资源隔离；SIMD**

# Pipeline execution engine practice in large-scale distributed analytical database

## Abstract

In the face of the new Internet business model with massive data growth, the importance of OLAP (Online Analytical Processing) databases for Internet companies has become increasingly prominent. It provides users with efficient data analysis and decision support capabilities through technologies such as multidimensional data models, data cube calculations, and aggregation query optimization. In practice, compared with the important high-concurrency point query in OLTP (online transaction processing) scenarios, the usage scenarios of OLAP databases are often more complicated, and complex aggregation queries under large and wide tables need to be processed. Query scenarios often involve multiple tables and require full table scans for specific columns. Therefore, OLAP databases usually use columnar storage instead of row-based storage used in traditional OLTP scenarios, and optimize performance by providing multiple aggregation optimizations, materialized views, expression reuse, and Runtime Filter.

The query engine is the core of the operation of the database system. With the rapid increase in the amount of data that needs to be processed in OLAP scenarios, the traditional Volcano engine has become a business performance engine because it cannot flexibly schedule queries, it is difficult to achieve resource isolation, and it has additional performance overhead due to unavoidable virtual function calls. bottleneck. Under this premise, the new Pipeline query execution engine follows the principle of "data driven by small data blocks", and splits the query plan tree into multiple non-blocking data streams, which is highly in line with the cache and instruction pipeline acceleration requirements of modern CPU architectures. It achieves better query performance and more flexible scheduling control, especially suitable for analysis queries of large-scale data sets.

This thesis implements a high-performance, reliable Pipeline engine and its optimization, and successfully applies it to Apache Doris, a well-known open source OLAP database, and has achieved significant performance improvement and function enhancement.

Relying on the optimization brought by the Pipeline execution engine implemented in this paper, Apache Doris has achieved the world's first performance on Clickbench, the de facto performance standard in the OLAP database field.

**Key Words: database; OLAP; pipeline execution engine; resource isolation; SIMD**

# 目 录

# 1 绪论

## 1.1 研究背景及意义

随着大数据时代的到来，互联网以及依托互联网技术而发展的企业。正在面对越来越高强度的信息处理需求。部分企业每日所需处理的新增数据量，从以往的 GB 级别一跃提升为 PB 级别。面对如此繁重的信息处理压力，对高效的数据分析处理工具的需求已经成为制约行业发展与科技进步的关键环节。

根据前瞻产业研究院《2019 中国大数据行业研究报告》显示，2013 年全球大数据储量为 4.3ZB，2018 年 全球大数据储量达到 33.0ZB，同比增长 52.8%。随着越来越多的企业走向在线平台，企业的生产运营转向数字化管理，极大的刺激了全球大数据市场需求。同时在云计算、人工智能、物联网、信息通信等技术交织应用驱动经济和生活数字化发展趋势下，大数据市场仍将保持较快增长。[1]

根据中国信息通信研究院 2023 年 1 月 4 日发布的《大数据白皮书（2022 年）》研究显示，我国大数据行业产业规模已达 1.3 万亿，大数据相关市场主体超过 18 万家。现阶段是我国进入全面建设社会主义现代化国家开局起步的关键阶段，也是我国大数据实现跨越式发展的重大战略机遇期。[2]

在这种发展趋势下，以往的单机、少量数据的即时查询工具已经不能满足业界的绝大部分需求。能够针对海量数据高效完成数据查询及分析的数据库引擎是市场及产业发展急需的工具。这也就催生了 OLAP（On-line Analytical Processing，联机分析处理）型数据库的市场。相比于传统的如 Mysql、Oracle 等数据库，OLAP 型数据库大多采用了强大的分布式并行架构和存储、聚合优化等技术，极大地提升了分析型查询语句的执行效率，使得企业进行大规模数据分析成为了实际可能。

根据行业发展报告，OLAP（联机分析处理）数据库市场目前正处于快速发展的阶段，并且在未来拥有广阔的前景。主要体现在以下几个特点：

1. 市场规模扩大：随着大数据的兴起和企业对数据分析和洞察力的需求增加，OLAP 数据库市场规模不断扩大。企业越来越重视数据驱动决策，需要强大的分析能力和高性能的查询处理，从而推动了 OLAP 数据库的市场需求。

2. 技术创新和竞争加剧：在 OLAP 数据库市场中，存在着众多的技术供应商和解决方案提供商。这促使企业不断进行技术创新和功能扩展，以满足不断变化的用户需求。例如，引入新的查询引擎、增强数据可视化功能、提供云端部署和服务等。竞争的加剧推动了市场的发展和创新。

3. 行业多样性：OLAP 数据库市场适用于各个行业，包括零售、金融、制造、物流、电信等。不同行业的企业都需要进行数据分析、业务智能和决策支持，从

而促进了 OLAP 数据库的广泛应用和市场发展。

在这种背景下，我们可以看到，OLAP 数据库市场目前处于快速发展的阶段，并且面临着广阔的前景。随着数据驱动决策的重要性增加、大数据和实时分析的需求增加以及云计算和分布式处理技术的兴起，OLAP 数据库将在各个行业中得到广泛应用，并持续发展和创新。具体来讲，以下几点体现了 OLAP 数据库的重要性：

1. 数据驱动决策的重要性增加：随着互联网、物联网和人工智能的发展，企业面临着更多、更复杂的数据。对数据驱动决策的需求将继续增加，因此 OLAP 数据库作为重要的数据分析工具，将持续发挥关键作用，并有望获得更广泛的应用。

2. 大数据和实时分析的需求增加：随着大数据时代的到来，企业需要处理和分析海量数据，以从中提取价值和洞察力。同时，实时数据分析的需求也在增加，企业需要快速获取实时数据，并进行即时的分析和决策。OLAP 数据库具备处理大规模数据和实时查询的能力，因此在这个领域有很大的发展潜力。

3. 云计算和分布式处理的兴起：云计算和分布式处理技术的发展，为 OLAP 数据库提供了更大的部署灵活性和弹性扩展性。云原生 OLAP 数据库服务的出现使得企业能够更轻松地部署和管理 OLAP 数据库，减少了硬件和维护成本，并提供了强大的计算资源。因此，云计算和分布式处理的兴起将进一步推动 OLAP 数据库市场的发展。

基于以上的业务背景可以发现，数据库市场需求正在逐渐转向复杂化、规模化。因此，OLAP 数据库，尤其是基于大规模分布式架构的大型 OLAP 数据库对于整个互联网行业的发展具有着重大作用。而对于这样的一个高性能的 OLAP 数据库，查询引擎是其中最为关键的核心部件，是整个数据库的核心与灵魂。它的性能与可靠性直接决定了一款 OLAP 数据库的性能与可靠性。经过数据库领域的多年发展，现已得到证明的是：基于数据的流式处理的 Pipeline 引擎在可靠性、性能等方面明显优于传统的 Volcano 执行引擎。因此，对于一款高性能的 OLAP 数据库，Pipeline 引擎的实现至关重要。可以说，它奠定了一款 OLAP 数据库的竞争力基础。

## 1.2 OLAP 数据库技术与市场发展现状

从技术上讲，OLAP 数据库与传统的 OLTP（On-line Transaction Processing，联机事务处理）主要有以下不同[3]：

表 1.1 OLTP（联机事务处理）与 OLAP（联机分析处理）异同

| | OLTP | OLAP |
|---|---|---|
| 主要使用场景 | 在线业务服务 | 数据分析、挖掘、机器学习 |
| 涉及数据量 | 当前少量业务数据 | 可能涉及较广时间范围的大量存档数据 |
| 事务和数据完整性 | 对事务和数据一致性要求高 | 不要求事务能力 |
| 功能使用要求 | 简单的增删改查操作，极短时间内响应 | 复杂的聚合、外部数据源查询、多表关联查询；时间要求不高 |
| 并发要求 | 高并发 | 低并发 |
| 关键技术 | 事务、索引、存算耦合等 | 列式存储、物化视图、存算分离等 |
| 可用性要求 | 非常高 | 相对较低 |
| 数据模型 | 关系模型、3NF 范式 | 维度模型、关系模型、一般不要求范式 |
| 技术典范 | MySQL、Oracle | Clickhouse、Doris |

就具体技术而言，针对 OLAP 数据库所面对的复杂查询场景特点——即大多数情况下需要进行全表扫描、查询大多数情况下涉及少数特定列，OLAP 数据库常常采用列式存储而非行式存储的方式来提升效率；由于经常涉及到一些特定复杂模式的查询，OLAP 数据库大多实现了 Rollup 表（上卷表）以及物化视图、多表物化视图来加速这些特定查询；对于大型的分布式场景，由于企业级用户所需要存储、查询的数据量特别巨大，如何实现存算分离、冷热分离也是当前 OLAP 数据库研究的重点。

其中，由于一款 OLAP 数据库系统一般需要支撑整个业务系统的数据分析需求，如何进行负载均衡——即在大查询（指运行所需时间较长的查询）和小查询（指运行所需时间较短的查询）同时存在的混合负载（有一定的大查询持续运行的同时，大量高并发小查询发起查询请求）下，保证大查询稳定运行的同时小查询不被长时间阻塞成为了一个关键问题。这一类问题在传统 Volcano 引擎下实现较为困难、不够灵活，推动了全新的 Pipeline 引擎的产生。

此外，由于商业活动中对成本和架构灵活性的需求，将计算和存储资源解耦，使其能够独立扩展和管理的存算分离技术也占据了重要的技术版图。传统的数据库系统通常将计算和存储功能紧密集成在一起，这样的架构在面对大规模数据和高并发负载时可能存在性能瓶颈和扩展困难。而云原生的存算分离能力通过将计算和存储分开，为互联网商业领域带来了一系列重要的优势和前景。

其次在市场层面，得益于互联网高速发展带来急速膨胀的数据量与数据分析需求，OLAP 数据库，尤其是云原生 OLAP 数据库的市场规模也在快速增长。根据中信证券研究部援引 IDC 的相关数据，在云化趋势下，预计未来云操作型数据库

和云数仓都将保持 25%以上的高速增长。根据 IDC 数据，2019 年全球操作型数据库市场规模为 325 亿美元，其中云操作型数据库规模 81 亿美元。IDC 预计 2024 年全球全球操作型数据库市场规模将达到 482 亿美元，2019-2024 年 CAGR 为 8.2%；其中云操作型数据库市场规模将达到 253 亿美元，2019-2024 年 CAGR 为 25.6%。[6]

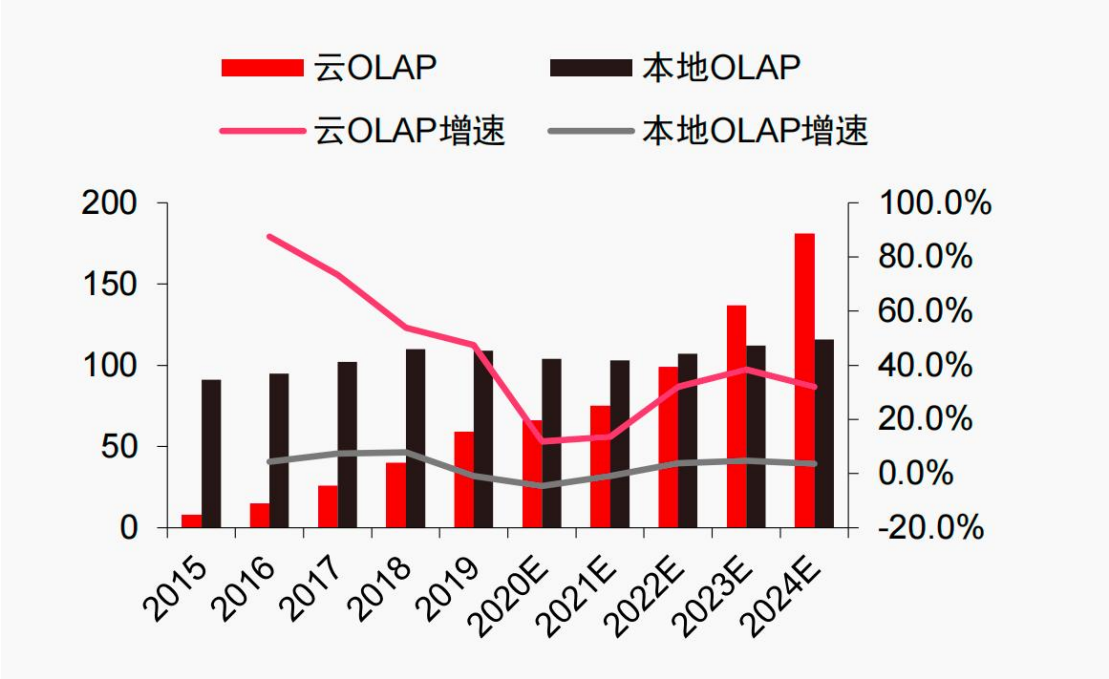


图 1.1 全球 OLAP 市场规模及增速

目前数据库领域，国产替代存量市场规模大于 600 亿元；与此同时，分析型的增量市场正以 30%以上的年均复合增速（2022-2025）高速扩张。[7] 因此，OLAP 领域的技术研究具有重要的技术与市场价值。

## 1.3 论文的研究内容与组织结构

在本文中，作者将首先介绍文中所实现的 Pipeline 执行引擎所属的完整数据库系统——即 Apache Doris。这是一款基于 MPP（大规模分布式架构）的全开源高性能 OLAP 数据库。基于这一数据库，本文将首先介绍 OLAP 数据库的基本概念与技术，阐明 Pipeline 执行引擎的重要性及其在技术架构中所担当的角色。基于其采用的 MPP 模型与实际业务压力，本文所实现的 Pipeline 引擎将会得到严苛的实际检验。

之后，本文将详细讲述 Apache Doris 中 Pipeline 执行引擎相比于 Volcano 引擎的改进、实际性能提升与整体的优缺点，并详细介绍实现原理、具体实现方式与性能优化，全方位描述这款引擎的实现方式与细节。

最后，本文将使用业界公认的事实标准，如 Clickbench、TPC-H、SSB 对 Pipeline 执行引擎进行全面的性能与稳定性测试，并展示和对比 Doris 数据库上应用 Pipeline 引擎前后的测试结果。

# 2 Apache Doris 数据库

## 2.1 发展历史

Apache Doris 是一款基于 MPP（大规模分布式架构）的全开源高性能 OLAP 数据库，在分析型数据库领域占据着重要的市场份额。在百度、美团、腾讯等大型互联网公司内部都得到了广泛应用，支撑了如百度凤巢报表系统在内的大量核心互联网商业活动，应用场景非常广泛。

作为一款 OLAP 数据库，它在大量互联网企业的数据驱动决策、业务智能和策略规划、用户行为分析和个性化推荐、实时监控和预测分析、数据可视化和报表展示领域发挥着重要作用。

它的前身是百度内部的 Doris（PALO）数据库，起源可以追溯到 2008 年。当时百度内部主要收入来源是广告业务，为了支撑广告主的多维度查询需求，基于 MySQL Sharding 的解决方案已经在性能上出现了明显不足。于是 Doris1 应运而生了，当时的架构比较简单：

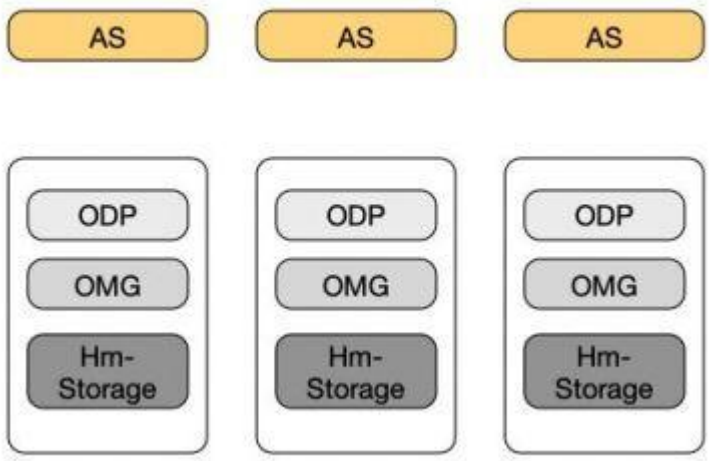


图 2.1 Doris1 架构

之后，经过多次迭代，Doris1 不断升级成为 Doris3，其间引入了 Key 列与 Value 列分离、分区、Schema Change 等大量功能，支撑了百度内部的大量业务线需求。

经过多次的迭代升级与重构，直到 2015 年，Doris 升级迭代成为 PALO2，采取了 FE（前端）+BE（后端）的多机多进程分布式架构，并兼容 MySQL 协议，通过 MySQL 作为查询接口：

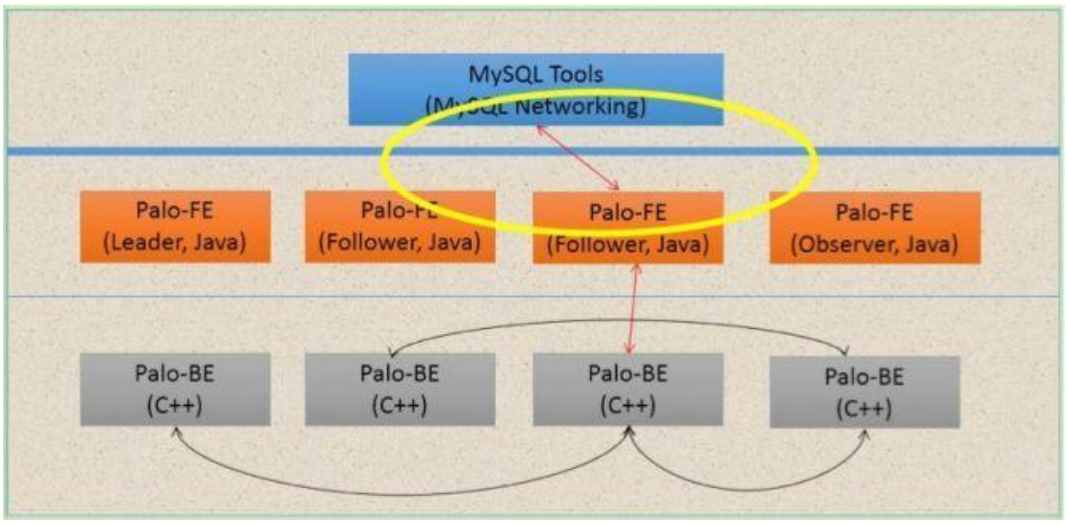


图 2.2 PALO2 架构

这种架构从 PALO 到开源成为 Apache 基金会顶级项目 Apache Doris，一直延续至今。此时系统架构本身变得相当简洁，并且不需要任何依赖。因为这种架构的简洁，后续提供公有云服务以及私有化部署变得相当简单；另一方面，其他用户也能够通过较低的门槛来搭建使用 PALO。

Palo 于 2017 年正式在 GitHub 上开源，并且在 2018 年贡献给 Apache 社区，并更名为 Apache Doris(incubating)进行正式孵化。〔4〕

四年后，全球最大的开源软件基金会 Apache 软件基金会于美国时间 2022 年 6 月 16 日宣布，Apache Doris 成功从 Apache 孵化器毕业，正式成为 Apache 顶级项目（Top-Level Project，TLP）。

## 2.2 应用场景

时至今日，Doris 的性能与稳定性已经在诸多商业场景中得到了验证，如：

a. 当前 Apache Doris 在小米内部已经具有数十个集群、总体达到数百台 BE 节点的规模，其中单集群最大规模达到近百台节点，拥有数十个实时数据同步任务，每日单表最大增量 120 亿、支持 PB 级别存储，单集群每天可以支持 2W 次以上的多维分析查询。

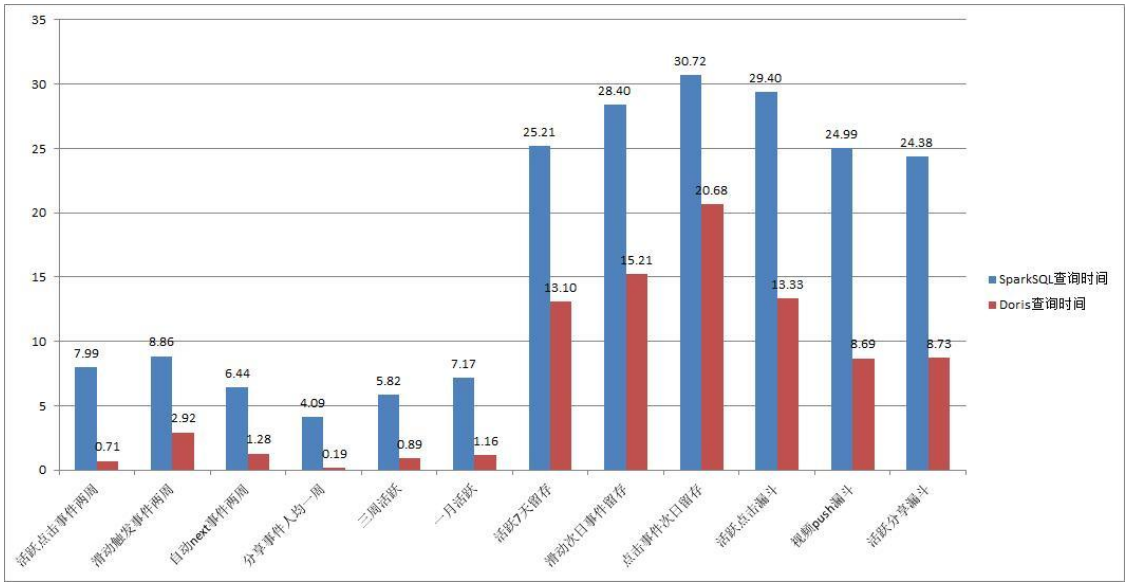


图 2.3 小米使用 Doris 后的业务性能提升

基于 Doris 的架构从数据源获取埋点数据后，数据接入后写入 Apache Doris，可以直接查询结果并在前端进行显示。真正实现了通过 Doris 统一了计算、存储，和资源管理 yarn 相关工具。

通过选取日均数据量大约 10 亿的业务，分别在不同场景下对 Doris 进行性能测试，其中包含 6 个事件分析场景 3 个留存分析场景以及 3 个漏斗分析场景。经过与 SparkSQL+Kudu+HDFS 的旧方案对比后，可以发现：在事件分析的场景下，平均查询所耗时间降低了 85%；在留存分析和漏斗分析场景下，平均查询所耗时间降低了 50%。[5]

b. 在美团的业务中，以前使用的基于 Kylin 的 MOLAP 架构存在着计算耗时大、无法查询明细数据等许多问题，在使用 Doris 后，整体在查询效率不变的情况下，生产耗能及存储成本都有较大收益。

使用 20BE+3FE 的 Doris 环境，美团在业务中支撑了数据分析产品数十个以上，整体响应达到 ms 级；支持百万、千万级大表关联查询；进行维表关联的雪花模型可以实现秒级响应；日级别，基于商家明细现场计算，同时满足汇总及下钻明细查询，查询时效基本都可以控制在秒级。[8]

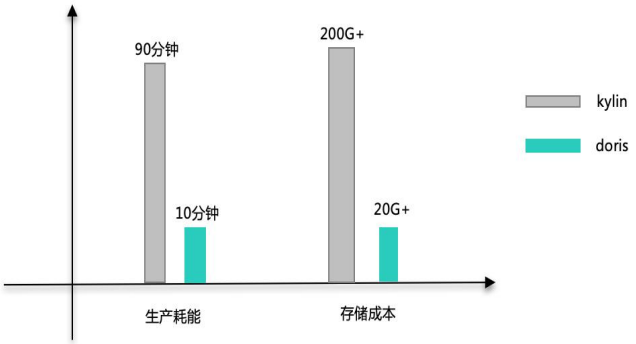


图 2.4 美团使用 Doris 后的业务性能提升

## 2.3 基础架构——MPP 架构

Apache Doris 采用了大规模分布式架构（MPP 架构）：

整体上来说，它由 FE（Frontend，前端）和 BE（Backend，后端）两类节点组成，FE 主要负责用户请求的接入、查询解析规划、元数据的管理、节点管理相关工作。BE 主要负责数据存储、查询计划的执行。

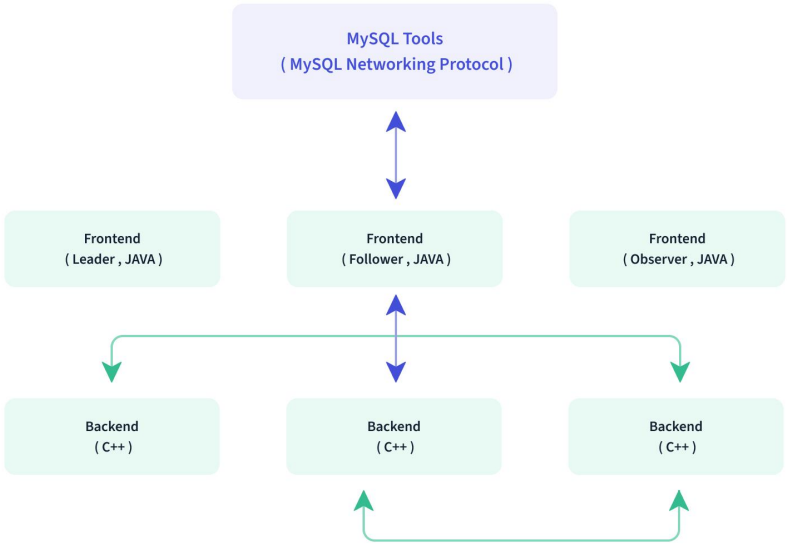


图 2.5 Doris 整体架构

这两类节点，都支持近乎无限的横向拓展，以提供无限增强的查询性能以及高可用能力。

所有 FE 节点被分为 Follower 和 Observer 两种身份（Leader 本质上是一种 Follower）。Observer 节点仅从 Leader 节点进行元数据同步，不参与选举。能够横向扩展以提供元数据的读服务的扩展性。当 FE 节点 Follower 数量达到 3 个时，集群进入高可用（HA）状态，此时可以提供读、写双方面的高可用能力。在 FE 高可用部署模式下，使用复制协议选主和主从同步元数据，所有的元数据修改操作，由 FE Leader 节点完成，FE Follower 节点可执行读操作。

在同步方面，FE 节点之间使用 BDBJE（Berkeley database java edition）管理元数据，BDBJE 本身实现了一种类 raft 分布式一致性协议。它提供的分布式同步能力保证了 FE 节点之间的数据同步。BE 节点之间不需要使用一致性协议完成分布式同步，这种能力完全通过 FE 节点提供。写操作被限制在 FE Leader 上执行，相关的日志将会按顺序保存并同步到 Follower 上，因此元数据的读写满足顺序一致性。

FE 的节点数目采用 2n+1 模式，其中最大可在容忍 n 个节点故障的情况下提

供正常服务。当 FE Leader 故障时，集群将从现有的 Follower 节点中重新进行选举操作，实现故障切换。

在数据可靠性方面，可以通过多 BE 部署以及同时对数据进行多副本（Replica）存储的方式进行保证。

除此之外，Doris 中存在一类可选的读取远端数据的专门进程，称为 Broker 节点。Broker 通过提供一个 RPC 服务端口来提供服务，是一个无状态的 Java 进程，负责为远端存储的读写操作封装一些类 POSIX 的文件操作。Broker 仅作为一个数据通路，并不参与任何计算，因此仅需占用较少的内存。通常一个 Doris 系统中会部署一个或多个 Broker 进程。[9]

## 2.4 关键技术

### 2.4.1 Nereids 优化器

现代查询优化器面临更加复杂的查询语句、更加多样的查询场景等挑战。同时，用户也越来越迫切的希望尽快获得查询结果。旧优化器的陈旧架构，难以满足今后快速迭代的需要。基于此，Doris 研发了现代架构的全新查询优化器 Nereids。在更高效的处理当前 Doris 场景的查询请求同时，提供更好的扩展性，为处理今后所需面临的更复杂的需求打下良好的基础。

Nereids 结合了 RBO（Rule-Based Optimization，基于规则的优化）和 CBO（Cost-Based Optimization，基于代价的优化）两种优化模式，具备以下特点：

更智能：新优化器将每个 RBO 和 CBO 的优化点以规则的形式呈现。对于每一个规则，新优化器都提供了一组用于描述查询计划形状的模式，可以精确的匹配可优化的查询计划。基于此，新优化器更好的可以支持诸如多层子查询嵌套等更为复杂的查询语句。

同时新优化器的 CBO 基于先进的 Cascades 框架，使用了更为丰富的数据统计信息，并应用了维度更科学的代价模型。这使得新优化器在面对多表 Join 的查询时，更加得心应手。

更健壮：新优化器的所有优化规则，均在逻辑执行计划树上完成。当查询语法语义解析完成后，变转换为一颗树状结构。相比于旧优化器的内部数据结构更为合理、统一。以子查询处理为例，新优化器基于新的数据结构，避免了旧优化器中众多规则对子查询的单独处理。进而减少了优化规则逻辑错误的可能。

更灵活：新优化器的架构设计更合理，更现代。可以方便地扩展优化规则和处理阶段。能够更为迅速的响应用户的需求。

在采用了 Nereids 优化器后，不经过任何手动优化，即可产生接近最优的查

询计划，极大地减轻了 DBA（数据库管理员）的工作负担。

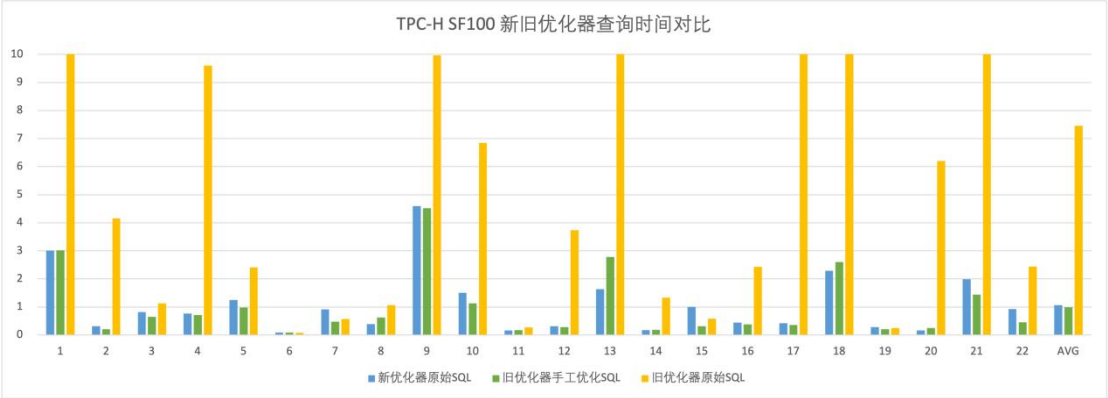


图 2.6 Nereids 与旧优化器在 TPC-H 100G 上性能对比

### 2.4.2 物化视图

物化视图是将预先计算（根据定义好的 SELECT 语句）好的数据集，存储在 Doris 中的一个特殊的表。它出现主要是为了满足用户既能对原始明细数据的任意维度分析，也能快速的对固定维度进行分析查询的需求。

对于那些经常重复使用相同子查询的查询，性能能够得到大幅提升。因为在命中了物化视图的情况下，大查询将退化为点查，从而几乎不消耗任何时间。

在没有物化视图功能之前，用户一般都是使用 Rollup 功能通过预聚合方式提升查询效率的。但是 Rollup 具有一定的局限性，他不能基于明细模型做预聚合。物化视图则在覆盖了 Rollup 的功能的同时，还能支持更丰富的聚合函数。所以物化视图其实是 Rollup 的一个超集。

在使用上，物化视图的创建类似于 MySQL 中的普通视图：

```
create materialized view store_amt as select store_id, sum(sale_amt) from sales_records group by store_id;
```

之后我们可以通过查询确认物化视图的存在：

```
    SHOW ALTER TABLE MATERIALIZED VIEW FROM db_name;
    +-------+--------------+---------------------+---------------------+--------------+----------------
+----------+--------------+-----------+-----------------------------------------------------------------------------------
-------------------------------------------------+----------+---------+
    | JobId | TableName    | CreateTime          | FinishedTime        |
BaseIndexName | RollupIndexName | RollupId | TransactionId | State    | Msg
| Progress | Timeout |
    +-------+--------------+---------------------+---------------------+--------------+----------------
+----------+--------------+-----------+-----------------------------------------------------------------------------------
-------------------------------------------------+----------+---------+
    | 22036 | sales_records | 2020-07-30 20:04:28 | 2020-07-30 20:04:57 |
sales_records | store_amt    | 22037    | 5008          | FINISHED |
| NULL     | 86400    |
```

```
    +-------+--------------+-------------------+-------------------+--------------+----------------
+----------+--------------+-----------+-----------------------------------------------------------------------------
-----------------------------------------------+----------+---------+
```

从而在查询时，这一已经创建的物化视图就可以在被查询包含的情况下被自动命中。可以通过 EXPLAIN 命令确认：

```
    EXPLAIN SELECT store_id, sum(sale_amt) FROM sales_records GROUP BY
store_id;
    +-----------------------------------------------------------------------------------------+
    |                                    Explain                                    String
|
    +-----------------------------------------------------------------------------------------+
    |                      PLAN                      FRAGMENT                      0
|
    |                                                  OUTPUT                  EXPRS:
|
    |                 <slot  4>  `default_cluster:test`.`sales_records`.`mv_store_id`
|
    |                                                                  <slot              5>
sum(`default_cluster:test`.`sales_records`.`mva_SUM__`sale_amt``)
|
    |                                        PARTITION:              UNPARTITIONED
|
    |
|
    |                                                VRESULT                  SINK
|
    |
|
    |                                                                    4:VEXCHANGE
|
    |                                                                offset:          0
|
    |
|
    |                      PLAN                      FRAGMENT                      1
|
    |
|
    |                     PARTITION:       HASH_PARTITIONED:       <slot       4>
`default_cluster:test`.`sales_records`.`mv_store_id` |
    |
|
    |                                      STREAM            DATA            SINK
|
```

```
|                                                    EXCHANGE ID: 04                                                    |
|                                                  UNPARTITIONED                                                        |
|                                                                                                                       |
|                                3:VAGGREGATE (merge finalize)                                                          |
|                                |  output: sum(<slot 5> sum(`default_cluster:test`.`sales_records`.`mva_SUM__`sale_amt``))  |
|                                |  group by: <slot 4> `default_cluster:test`.`sales_records`.`mv_store_id`                 |
|                                |  cardinality=-1                                                                          |
|                                                                                                                       |
|                                                  2:VEXCHANGE                                                          |
|                                                offset: 0                                                              |
|                                                                                                                       |
|                PLAN FRAGMENT 2                                                                                        |
|                                                                                                                       |
|                PARTITION: HASH_PARTITIONED: `default_cluster:test`.`sales_records`.`record_id`                        |
|                                                                                                                       |
|                STREAM DATA SINK                                                                                       |
|                EXCHANGE ID: 02                                                                                        |
|                HASH_PARTITIONED: <slot 4> `default_cluster:test`.`sales_records`.`mv_store_id`                        |
|                                                                                                                       |
|                1:VAGGREGATE (update serialize)                                                                        |
|                |  STREAMING                                                                                           |
|                |  output: sum(`default_cluster:test`.`sales_records`.`mva_SUM__`sale_amt``)                           |
```

```
|
    |           |       group   by:   `default_cluster:test`.`sales_records`.`mv_store_id`
|
    |                                           |                                   cardinality=-1
|
    |                                                                                           |
|
    |                                                                                 0:VOlapScanNode
|
    |                           TABLE:     default_cluster:test.sales_records(store_amt),
PREAGGREGATION: ON                          |
    |                 partitions=1/1,   tablets=10/10,   tabletList=50028,50030,50032   ...
|
    |                               cardinality=1,     avgRowSize=1520.0,     numNodes=1
|
    +-----------------------------------------------------------------------------------------+
```

观察到 VOlapScanNode 没有扫描实际物理表，而是扫描到了我们创建的物化视图 store_amt，说明这一物化视图被该查询成功命中了。

在这一基础上，Doris 的物化视图具备自动更新、自动命中最佳物化视图的能力，极大地减轻了数据库使用者的负担。这一功能在经常进行相关数据查询的 OLAP 领域具有非常重要的意义。

### 2.4.3 Runtime Filter

RuntimeFilter 指的是在进行某些 Join 操作时，将右表的数据计算出一个过滤条件，并将该过滤条件下推给左表扫描，以减少左表的数据扫描量。一般在左表很大而右表很小的情况下会有比较显著的作用。

与谓词下推、分区裁剪等优化不同的是，RuntimeFilter 所使用的过滤条件是查询执行时根据右表数据动态生成的。

假设有这样一条 sql：

```
SELECT * FROM A where A.k1 = B.k1;
```

此时如果左表 A 是大表，而右表 B 很小，而 A 的 k1 是 key 列，则 RuntimeFilter 可以先扫出 B 的全部数据，并将其作为过滤条件下推至 A 的 Scan 算子内，于是 Scan 算子读取数据时就可以仅读取很少的行。

经过实际测试，在开启 RuntimeFilter 后，Doris 查询速度在 TPC-DS 1T 数据集上产生了超过 20%的整体提升。

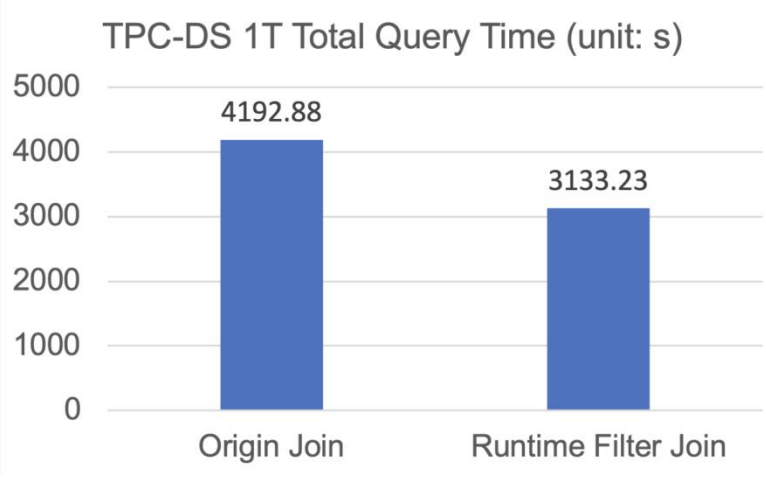


图 2.7 开启 RuntimeFilter 后在 TPC-DS 1T 上的性能变化

### 2.4.4 Colocate Join

Colocation Join 功能，是将一组拥有相同 CGS 的 Table 组成一个 CG。并保证这些 Table 对应的数据分片会落在同一个 BE 节点上。使得当 CG 内的表进行分桶列上的 Join 操作时，可以通过直接进行本地数据 Join，减少数据在节点间的传输耗时。

例如：

```
CREATE TABLE `tbl1` (
`k1` date NOT NULL COMMENT "",
`k2` int(11) NOT NULL COMMENT "",
`v1` int(11) SUM NOT NULL COMMENT ""
) ENGINE=OLAP
AGGREGATE KEY(`k1`, `k2`)
PARTITION BY RANGE(`k1`)
(
PARTITION p1 VALUES LESS THAN ('2019-05-31'),
PARTITION p2 VALUES LESS THAN ('2019-06-30')
)
DISTRIBUTED BY HASH(`k2`) BUCKETS 8
PROPERTIES (
"colocate_with" = "group1"
);

CREATE TABLE `tbl2` (
      `k1` datetime NOT NULL COMMENT "",
      `k2` int(11) NOT NULL COMMENT "",
      `v1` double SUM NOT NULL COMMENT ""
) ENGINE=OLAP
AGGREGATE KEY(`k1`, `k2`)
```

```
DISTRIBUTED BY HASH(`k2`) BUCKETS 8
PROPERTIES (
    "colocate_with" = "group1"
);
```

对于 tbl1 和 tbl2 这样两个表，由于指定了它们的 colocate 组相同，则在插入数据时，对应的数据将会在同一个分片中存储，这样在它们之间进行 Join 操作就避免了 Exchange 的网络传输开销。

如果 colocate join 正确生效，则查看查询计划

```
DESC SELECT * FROM tbl1 INNER JOIN tbl2 ON (tbl1.k2 = tbl2.k2);
```

结果为：

```
+----------------------------------------------------+
| Explain String                                     |
+----------------------------------------------------+
| PLAN FRAGMENT 0                                    |
|   OUTPUT EXPRS:`tbl1`.`k1` |                       |
|     PARTITION: RANDOM                              |
|                                                    |
|     RESULT SINK                                    |
|                                                    |
|     2:HASH JOIN                                    |
|     |   join op: INNER JOIN                        |
|     |   hash predicates:                           |
|     |   colocate: true                             |
|     |       `tbl1`.`k2` = `tbl2`.`k2`              |
|     |   tuple ids: 0 1                             |
|     |                                              |
|     |----1:OlapScanNode                            |
|     |       TABLE: tbl2                            |
|     |       PREAGGREGATION: OFF. Reason: null      |
|     |       partitions=0/1                         |
|     |       rollup: null                           |
|     |       buckets=0/0                            |
|     |       cardinality=-1                         |
|     |       avgRowSize=0.0                         |
|     |       numNodes=0                             |
|     |       tuple ids: 1                           |
|     |                                              |
|     0:OlapScanNode                                 |
|         TABLE: tbl1                                |
|         PREAGGREGATION: OFF. Reason: No AggregateInfo |
|         partitions=0/2                             |
|         rollup: null                               |
|         buckets=0/0                                |
```

```
|          cardinality=-1                                                |
|          avgRowSize=0.0                                                |
|          numNodes=0                                                    |
|          tuple ids: 0                                                  |
+--------------------------------------------------------+
```

可以看到，HASH JOIN NODE 显示 colocate: true，证明这两张表 join 时 colocate join 优化成功生效。

# 3 Pipeline 查询执行引擎

## 3.1 产生背景——Volcano 引擎的缺点

在 Pipeline 引擎出现之前，大多数数据库执行模型采取的是 Volcano 模型，即火山模型。这种模型执行的本质是在整棵执行计划树上，每个节点递归地调用子节点的 next()函数，逐层拉取数据上来。

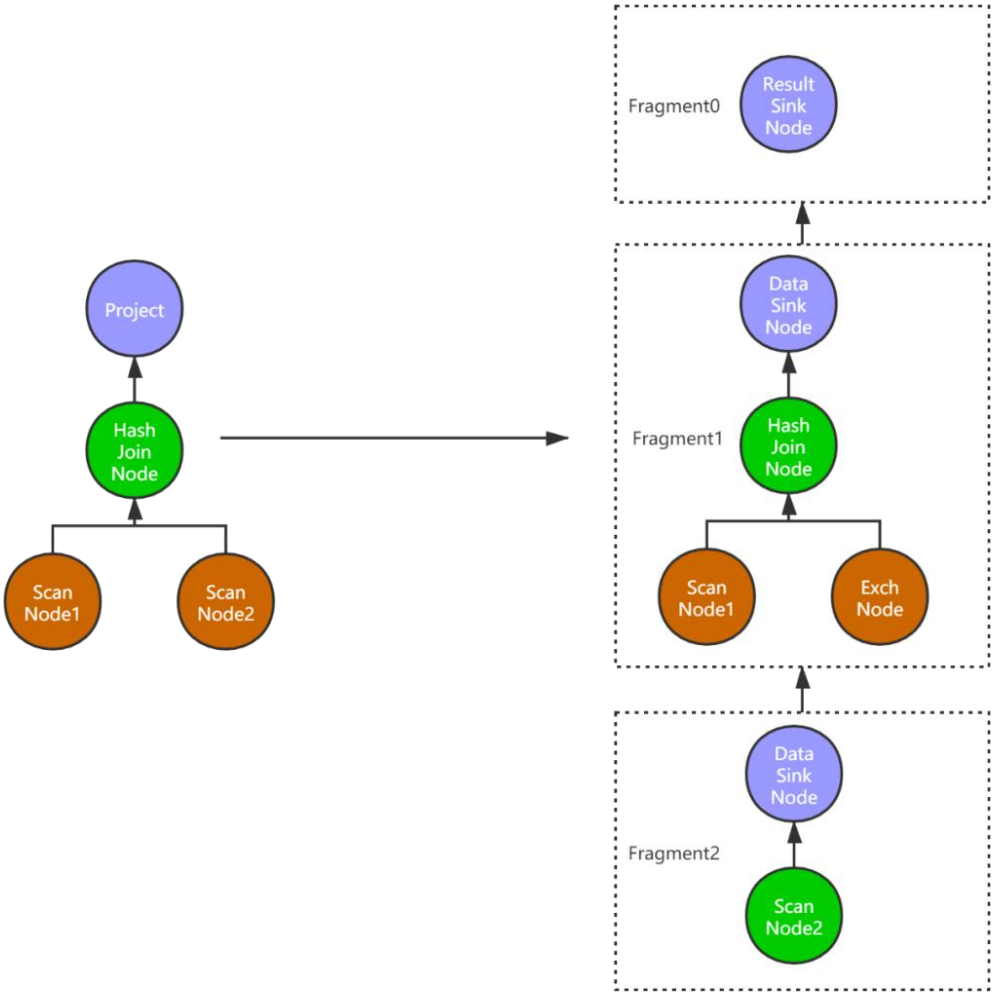


图 3.1 Fragment 拆分

在 Doris 中，我们首先会把整个 Plan（执行计划）拆分为多个 Fragment，然后将 Fragment 发送到对应的 BE 进行实际执行。

在具体执行时，数据流的本质是拉取的，这带来了两方面的额外性能开销[11]：

a. 由于需要上层拉取下层的数据流，程序的控制流也需要不断的进行跳转。这种跳转导致代码的局部性很差，CPU 缓存的命中率较低。额外的控制流跳转本

身也带来了很大的性能开销；

b. Volcano 模型中节点拉取所使用的 next()函数本质上是虚函数调用。这种调用无法在编译时被内联展开，同时也意味着局部需要发生难以预测的跳转，这对 CPU 的指令预取和超前执行都是一种负担。CPU 指令流水线的执行很可能在这种情境下被频繁打断，导致程序的运行效率很不理想。

对于对执行速度要求极高的 OLAP 场景来说，Volcano 模型的这些问题无法忽视。除了代码本身，Volcano 模型本身不够灵活，带来了其他方面的性能问题：

a. 无法充分利用多核计算能力提升查询性能。多数场景下进行性能调优时需要手动设置并行度，在生产环境中几乎很难进行设定。

b. 单机查询的每个 instance 对应线程池的一个线程，这会带来额外的两个问题：

a) 线程池一旦打满。Doris 的查询引擎会进入假性死锁，对后续的查询无法响应。同时有一定概率进入逻辑死锁的情况：比如所有的线程都在执行一个 instance 的 probe 任务。

b) 阻塞的算子会占用线程资源，而阻塞的线程资源无法让渡给能够调度的 instance，整体资源利用率受限。

c. 阻塞算子依赖操作系统的线程调度机制，线程切换开销较大（尤其在系统混布的场景中）

d. CPU 的资源管理困难，很难做到更细粒度的资源管理，多查询的混合并发做到合理的资源调度：

a) 大查询生成海量 instance 之后，线程池被打满。小查询几乎得不到调度的机会，导致混合负载下，小查询的时延较高。

b) 在混合部署的集群之间，用户间的 CPU 资源无法得到好的隔离，几乎是处于无管控状态。

## 3.2 设计与功能

要解决以上问题，在 Volcano 框架下是比较困难的，这是它本身的架构模式所限制的。因此，我们必须引入一种全新的、更加灵活的执行框架，这就是 Pipeline 执行引擎。

Pipeline 执行引擎的核心在于，将执行计划树递归式拉取数据的模式，改为从底层主动向上推送数据。整个执行引擎使用固定数量的线程池，各个 pipeline 被调度的逻辑是由数据驱动的。

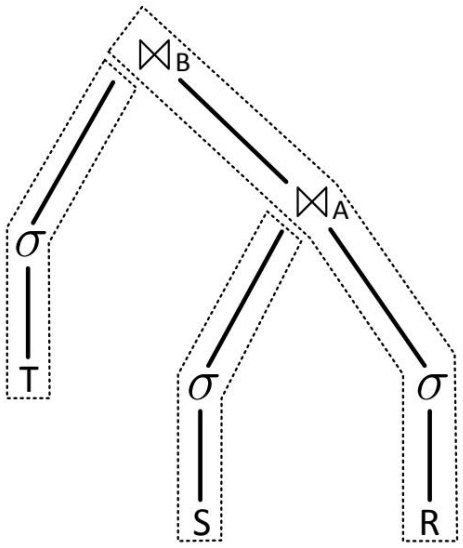


图 3.2 执行计划树拆分为 Pipeline

例如上图中所呈现的执行计划树，Join 操作的阻塞逻辑天然形成了 Pipeline 之间的拆分点。在对应节点进行拆分后，最右侧的 Pipeline 因为依赖其他两条 Pipeline 而自然阻塞。而其他两条 Pipeline 是无阻塞的。每个 pipeline 是否占有线程资源进行调度，取决于前置的数据是否 Ready，所以每个 Pipeline 的调度核心点是数据进行驱动的，未 ready 的 Pipeline 需要主动释放线程资源。

可以看出，这二者之间的主要不同在于：Pipeline 模型的数据流是下层主动向上推送的——下层 pipeline 执行完毕后，将会唤起依赖该 pipeline 的上层 pipeline 继续执行。而 Volcano 模型是基于 next()函数的递归实现的，数据流是上层递归式地从下层拉取上来的。

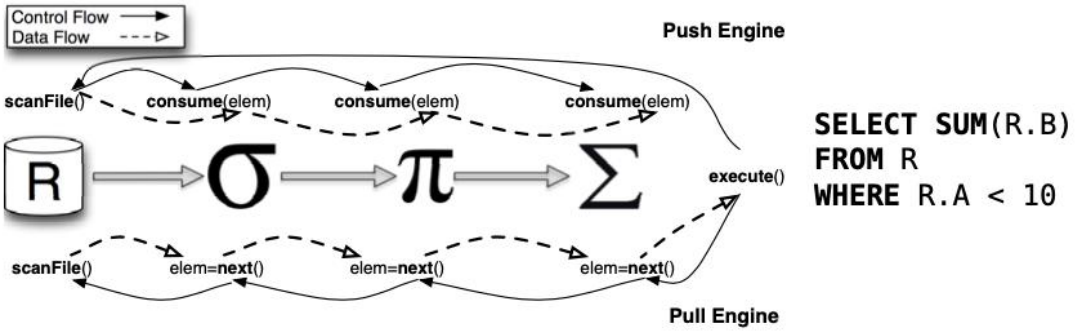


图 3.3 推送与拉取机制对比

## 3.3 优点

采用了这样的一种拆分设计后，Pipeline 模型成功使得阻塞算子实现异步化，不再阻塞整个查询执行。具体优点有以下几点：

a. 充分利用多核能力。当前服务器级别 CPU 核数快速增长，并发能力愈发提高。Pipeline 模型从数据流的角度向上推送数据，实现了最大可能的并发。多个 Pipeline 之间只要数据没有依赖，就可以并发执行。

b. 解决了线程池死锁问题和线程频繁切换的开销。MLFQ 类似于用户态的执行调度器，它在切换任务时只需要更换 PipelineTask 而不需要进入内核态切换线程，减少了大量开销。

c. 为 CPU 资源隔离打下了基础。能够解决大小查询的公平调度问题，多用户之间的资源隔离问题可以利用 Pipeline 框架解决。

## 3.4 缺点

相比原始的火山模型，Pipeline 模型让整个执行引擎的控制流和数据流做了拆解。这会带来工程复杂度和问题诊断的复杂度的提升，同时 Pipeline 之间的阻塞被拆解成逻辑依赖关系了，这里设计的不慎同样会带来概率性的逻辑死锁问题，这类问题衍生出来的性能和死锁问题都是很难定位的。

# 4 系统实现与优化

## 4.1 算子生成

在非 Pipeline 的代码中，Doris 已经实现了各个执行节点（Node），它们承载了 SQL 执行的实际具体功能。每个具体的执行节点类型都使用 ExecNode 作为它的基类。ExecNode 实现了完成一个执行计划所需的最基本的功能：

```
class ExecNode {
public:
    // Init conjuncts.
    ExecNode(ObjectPool* pool, const TPlanNode& tnode, const
DescriptorTbl& descs);

    virtual ~ExecNode();
    [[nodiscard]] virtual Status init(const TPlanNode& tnode, RuntimeState*
state);
    [[nodiscard]] virtual Status prepare(RuntimeState* state);
    [[nodiscard]] virtual Status open(RuntimeState* state);
    [[nodiscard]] virtual Status alloc_resource(RuntimeState* state);
    [[nodiscard]] virtual Status get_next(RuntimeState* state,
vectorized::Block* block, bool* eos);
    [[nodiscard]] virtual Status pull(RuntimeState* state, vectorized::Block*
output_block, bool* eos) {
        return get_next(state, output_block, eos);
    }
    [[nodiscard]] virtual Status push(RuntimeState* state, vectorized::Block*
input_block, bool eos) {
        return Status::OK();
    }
    [[nodiscard]] virtual Status sink(RuntimeState* state, vectorized::Block*
input_block, bool eos);
    virtual Status close(RuntimeState* state);
    virtual void release_resource(RuntimeState* state);
    [[nodiscard]] static Status create_tree(RuntimeState* state, ObjectPool*
pool, const TPlan& plan, const DescriptorTbl& descs, ExecNode** root);
protected:
    void release_block_memory(vectorized::Block& block, uint16_t child_idx =
0);
    Status do_projections(vectorized::Block* origin_block, vectorized::Block*
output_block);
```

```
        vectorized::VExprContextSPtrs _conjuncts;
        std::vector<ExecNode*> _children;

    private:
        friend class pipeline::OperatorBase;
        bool _is_closed;
        bool _is_resource_released = false;
        std::atomic_int _ref; // used by pipeline operator to release resource.
    };
```

可以看到，对于所有执行节点而言，pull()是其最基础的功能，它实现了Volcano 模型中父节点递归从子节点拉取一个 batch 的数据（即一个 Block）的功能。一般来说，子节点会通过实现 get_next()来具体实现该节点的功能，例如VScanNode 的一组实例调用链如下：

```
    Status VScanNode::get_next(RuntimeState* state, vectorized::Block* block,
bool* eos) {
        Defer drop_block_temp_column {[&]() {
            auto all_column_names = block->get_names();
            for (auto& name : all_column_names) {
                if (name.rfind(BeConsts::BLOCK_TEMP_COLUMN_PREFIX, 0) == 0)
{
                    block->erase(name);
                }
            }
        }};

        if (state->is_cancelled()) {
            if (_scanner_ctx) {
                _scanner_ctx->set_status_on_error(Status::Cancelled("query
cancelled"));
                return _scanner_ctx->status();
            } else {
                return Status::Cancelled("query cancelled");
            }
        }

        if (_eos) {
            *eos = true;
            return Status::OK();
        }

        vectorized::BlockUPtr scan_block = nullptr;
        RETURN_IF_ERROR(_scanner_ctx->get_block_from_queue(state,
&scan_block, eos, _context_queue_id));
```

```
        if (*eos) {
            DCHECK(scan_block == nullptr);
            return Status::OK();
        }

        // get scanner's block memory
        block->swap(*scan_block);
        _scanner_ctx->return_free_block(std::move(scan_block));

        reached_limit(block, eos);
        if (*eos) {
            // reach limit, stop the scanners.
            _scanner_ctx->set_should_stop();
        }

        return Status::OK();
    }
```

它的实际数据通过 VScannerContext 传递。具体的 Scanner（如 NewOlapScanner）会通过 ScannerContext::append_blocks_to_queue 将读取的数据存入 ScannerContext::_blocks_queue，对应 VScanNode 就可以利用其保有的 Scanner 读取：

```
    Status ScannerContext::get_block_from_queue(RuntimeState* state,
vectorized::BlockUPtr* block, bool* eos, int id, bool wait) {
        std::unique_lock l(_transfer_lock);
        if (has_enough_space_in_blocks_queue() && _num_running_scanners == 0)
{
            _num_scheduling_ctx++;
            _scanner_scheduler->submit(this);
        }
        // Wait for block from queue
        if (wait) {
            SCOPED_TIMER(_scanner_wait_batch_timer);
            while (!(!_blocks_queue.empty() || _is_finished
|| !_process_status.ok() ||
                        state->is_cancelled())) {
                _blocks_queue_added_cv.wait(l);
            }
        }

        if (state->is_cancelled()) {
            _process_status = Status::Cancelled("cancelled");
        }
```

```
        if (!_process_status.ok()) {
            return _process_status;
        }

        if (!_blocks_queue.empty()) {
            *block = std::move(_blocks_queue.front());
            _blocks_queue.pop_front();
            auto block_bytes = (*block)->allocated_bytes();
            _cur_bytes_in_queue -= block_bytes;
            _queued_blocks_memory_usage->add(-block_bytes);
            return Status::OK();
        } else {
            *eos = _is_finished;
        }
        return Status::OK();
    }
```

对于 Pipeline 引擎而言，我们需要基于 Volcano 模型已有的执行节点进行升级，实现 pipeline 中的算子。在节点到算子的升级改造中，每个算子所实现的实际功能与之前的节点并无不同。因此，我们可以继续复用对应逻辑与结构，在这之上实现对应的 pipeline 模型中的算子。例如，pipeline 中的 ScanOperator 是这样实现的：

```
    class ScanOperatorBuilder : public OperatorBuilder<vectorized::VScanNode> {
    public:
        ScanOperatorBuilder(int32_t id, ExecNode* exec_node);
        bool is_source() const override { return true; }
        OperatorPtr build_operator() override;
    };

    class ScanOperator : public SourceOperator<ScanOperatorBuilder> {
    public:
        ScanOperator(OperatorBuilderBase* operator_builder, ExecNode*
scan_node);

        bool can_read() override; // for source

        bool is_pending_finish() const override;

        bool runtime_filters_are_ready_or_timeout() override;

        std::string debug_string() const override;

        Status try_close() override;
    };
```

其中涉及到的几个基类的实现如下：

```
template <typename OperatorBuilderType>
class SourceOperator : public StreamingOperator<OperatorBuilderType> {
public:
    using NodeType =
std::remove_pointer_t<decltype(std::declval<OperatorBuilderType>().exec_node())>;

    SourceOperator(OperatorBuilderBase* builder, ExecNode* node)
            : StreamingOperator<OperatorBuilderType>(builder, node) {}

    ~SourceOperator() override = default;

    Status get_block(RuntimeState* state, vectorized::Block* block,
                     SourceState& source_state) override {
        auto& node = StreamingOperator<OperatorBuilderType>::_node;
        bool eos = false;
        RETURN_IF_ERROR(node->get_next_after_projects(
                state, block, &eos,
                std::bind(&ExecNode::pull, node, std::placeholders::_1,
std::placeholders::_2,
                          std::placeholders::_3)));
        source_state = eos ? SourceState::FINISHED :
SourceState::DEPEND_ON_SOURCE;
        return Status::OK();
    }
};

template <typename NodeType>
class OperatorBuilder : public OperatorBuilderBase {
public:
    OperatorBuilder(int32_t id, const std::string& name, ExecNode* exec_node
= nullptr)
            : OperatorBuilderBase(id, name),
_node(reinterpret_cast<NodeType*>(exec_node)) {}

    ~OperatorBuilder() override = default;

    const RowDescriptor& row_desc() override { return _node->row_desc(); }

    NodeType* exec_node() const { return _node; }

protected:
    NodeType* _node;
};
```

```
template <typename OperatorBuilderType>
class StreamingOperator : public OperatorBase {
public:
    using NodeType =
std::remove_pointer_t<decltype(std::declval<OperatorBuilderType>().exec_node())>;

    StreamingOperator(OperatorBuilderBase* builder, ExecNode* node)
            : OperatorBase(builder),
_node(reinterpret_cast<NodeType*>(node)) {}

    ~StreamingOperator() override = default;

    Status prepare(RuntimeState* state) override {
        _node->increase_ref();
        _use_projection = _node->has_output_row_descriptor();
        return Status::OK();
    }

    Status open(RuntimeState* state) override {
        RETURN_IF_ERROR(_node->alloc_resource(state));
        return Status::OK();
    }

    Status sink(RuntimeState* state, vectorized::Block* in_block,
                SourceState source_state) override {
        return _node->sink(state, in_block, source_state ==
SourceState::FINISHED);
    }

    Status close(RuntimeState* state) override {
        if (is_closed()) {
            return Status::OK();
        }
        if (!_node->decrease_ref()) {
            _node->release_resource(state);
        }
        _is_closed = true;
        return Status::OK();
    }

    Status get_block(RuntimeState* state, vectorized::Block* block,
                     SourceState& source_state) override {
        DCHECK(_child);
```

```
            auto input_block = _use_projection ? _node->get_clear_input_block() :
block;
            RETURN_IF_ERROR(_child->get_block(state, input_block,
source_state));
            bool eos = false;
            RETURN_IF_ERROR(_node->get_next_after_projects(
                    state, block, &eos,
                    std::bind(&ExecNode::pull, _node, std::placeholders::_1,
std::placeholders::_2,
                              std::placeholders::_3),
                    false));
            return Status::OK();
        }

        Status finalize(RuntimeState* state) override { return Status::OK(); }

        bool can_read() override { return _node->can_read(); }

    protected:
        NodeType* _node;
        bool _use_projection;
    };
```

这样我们很容易发现，在 pipeline 中使用的 ScanOperator 其本质就是对相关的 VScanNode 的封装。从 SourceOperator::get_block()可以看出，它的实际执行过程还是调用了对应 VScanNode 的 get_next_after_projects()实现。

对于一个查询的执行而言，我们在实际执行（execute）之前，会通过 prepare()、open()等函数进行相应的资源准备，这种操作在 pipeline 逻辑中，我们也通过给 Operator 实现对应的函数来传递这一过程。

## 4.2 执行计划拆分

将整个执行计划树拆分为 pipeline 的逻辑在 PipelineFragmentContext::_build_pipelines() 中实现，它是在 PipelineFragmentContext 的 prepare 过程中，从根节点开始递归实现的：

1. 首先，我们需要根据 FE 传递过来的执行计划创建对应的执行节点和他们组成的执行计划树，这与非 Pipeline 模型一致：

```
    RETURN_IF_ERROR(ExecNode::create_tree(_runtime_state.get(),
_runtime_state->obj_pool(),request.fragment.plan, *desc_tbl, &_root_plan));
```

ExecNode::create_tree 本质上是一种递归的过程：

```
Status ExecNode::create_tree(RuntimeState* state, ObjectPool* pool, const TPlan&
```

```
plan,
                                        const DescriptorTbl& descs, ExecNode** root) {
    if (plan.nodes.size() == 0) {
        *root = nullptr;
        return Status::OK();
    }

    int node_idx = 0;
    RETURN_IF_ERROR(create_tree_helper(state, pool, plan.nodes, descs, nullptr,
&node_idx, root));

   if (node_idx + 1 != plan.nodes.size()) {
            return Status::InternalError(
                "Plan tree only partially reconstructed. Not all thrift nodes were
used.");
    }
    return Status::OK();
}

Status ExecNode::create_tree_helper(RuntimeState* state, ObjectPool* pool,
                                    const std::vector<TPlanNode>& tnodes,
                                    const DescriptorTbl& descs, ExecNode*
parent, int* node_idx,ExecNode** root) {
    // propagate error case
    if (*node_idx >= tnodes.size()) {
        // TODO: print thrift msg
        return Status::InternalError("Failed to reconstruct plan tree from thrift.");
    }
    const TPlanNode& tnode = tnodes[*node_idx];

    int num_children = tnodes[*node_idx].num_children;
    ExecNode* node = nullptr;
    RETURN_IF_ERROR(create_node(state, pool, tnodes[*node_idx], descs,
&node));

    if (parent != nullptr) {
        parent->_children.push_back(node);
    } else {
        *root = node;
    }

    for (int i = 0; i < num_children; i++) {
        ++*node_idx;
        RETURN_IF_ERROR(create_tree_helper(state, pool, tnodes, descs, node,
```

```
node_idx, nullptr));

            if (*node_idx >= tnodes.size()) {
                return Status::InternalError("Failed to reconstruct plan tree from
thrift.");
            }
        }

        RETURN_IF_ERROR(node->init(tnode, state));

        for (int i = 1; i < node->_children.size(); ++i) {
            node->runtime_profile()->add_child(node->_children[i]->runtime_profile(),
true, nullptr);
        }

        if (!node->_children.empty()) {
            node->runtime_profile()->add_child(node->_children[0]->runtime_profile(),
true, nullptr);
        }

        return Status::OK();
}
```

2. 之后，我们将会从已经生成的执行计划树拆分构建 pipeline：

```
        _root_pipeline = fragment_context->add_pipeline();
        RETURN_IF_ERROR(_build_pipelines(_root_plan, _root_pipeline));
        if (_sink) {
            RETURN_IF_ERROR(_create_sink(request.local_params[idx].sender_id,
                                         request.fragment.output_sink,
_runtime_state.get()));
        }
```

按照算子是否阻塞，将执行计划树逐算子拆分、合并为 pipeline 的逻辑可以大致拆分为两种处理方式：

a. 非阻塞型算子。

非阻塞算子的实现比较简单。根据我们的 pipeline 设计，非阻塞的算子不需要进行 pipeline 层面的拆分，否则会带来额外的开销。因此，这种算子只需要将其添加至当前 pipeline 即可。以 exchange 算子为例：

```
            case TPlanNodeType::EXCHANGE_NODE: {
                OperatorBuilderPtr operator_t =
std::make_shared<ExchangeSourceOperatorBuilder>(next_operator_builder_id(),
node);
                RETURN_IF_ERROR(cur_pipe->add_operator(operator_t));
                break;
```

```
        }
```

可以看到，其中所实现的仅仅是将算子连接到当前 pipeline：

```
    Status Pipeline::add_operator(OperatorBuilderPtr& op) {
        if (_operator_builders.empty() && !op->is_source()) {
            return Status::InternalError("Should set source before other
operator");
        }
        _operator_builders.emplace_back(op);
        return Status::OK();
    }
```

b. 阻塞型算子。

对于阻塞型算子，由于它们需要对下层数据进行部分或者全量物化，如果将其作为一条完整 pipeline 中的中间节点，那么当其某个子节点将数据推送上来时，如果其他子节点还未完成，必然会导致 pipeline 内部的阻塞等待，这与我们 pipeline 的设计不符。因此，对于这一类需要阻塞、局部物化的节点（例如 Sort、Aggregation、Join 节点），我们需要在这些节点处拆分 pipeline，为其添加 source 节点，使之成为一条新 pipeline 的起点。

以 HashJoinNode 为例：

```
        case TPlanNodeType::HASH_JOIN_NODE: {
            auto* join_node = assert_cast<vectorized::HashJoinNode*>(node);
            auto new_pipe = add_pipeline();
            if (join_node->should_build_hash_table()) {
                RETURN_IF_ERROR(_build_pipelines(node->child(1), new_pipe));
            } else {
                OperatorBuilderPtr builder =
std::make_shared<EmptySourceOperatorBuilder>(
                        next_operator_builder_id(), node->child(1)->row_desc(),
node->child(1));
                new_pipe->add_operator(builder);
            }
            OperatorBuilderPtr join_sink =

std::make_shared<HashJoinBuildSinkBuilder>(next_operator_builder_id(),
join_node);
            RETURN_IF_ERROR(new_pipe->set_sink(join_sink));
            new_pipe->disable_task_steal();

            RETURN_IF_ERROR(_build_pipelines(node->child(0), cur_pipe));
            OperatorBuilderPtr join_source =
std::make_shared<HashJoinProbeOperatorBuilder>(
                    next_operator_builder_id(), join_node);
            RETURN_IF_ERROR(cur_pipe->add_operator(join_source));
```

```
            cur_pipe->add_dependency(new_pipe);
            break;
        }
```

可以看到，此时我们将 HashJoin 的右表通过_build_pipelines()继续递归创建为一条新的 pipeline，并设置 sink 节点，连接到当前 pipeline 的 source 节点。并且通过 add_dependency()为左表所在的 pipeline 设置右表对应 pipeline 的依赖：

```
    void add_dependency(std::shared_ptr<Pipeline>& pipeline) {
        pipeline->_parents.push_back(shared_from_this());
        _dependencies.push_back(pipeline);
    }
```

在 PipelineScheduler 进行调度时，将会检测相关的 dependency 是否已经完成，从而决定是否调度相关 PipelineTask。

3. 在 pipeline 拆分完成之后，它们仍然不足以独立形成被调度的实体，因此下一步是将所有 Pipeline 组成 PipelineTask：

```
    RETURN_IF_ERROR(_build_pipeline_tasks(request));
```

这一过程的核心操作是为 Pipeline 中的 Operator 创建真正的 ExecNode，然后设定正确的 child 关系，以及附加对应的 sink 节点：

```
Status PipelineFragmentContext::_build_pipeline_tasks(
        const doris::TPipelineFragmentParams& request) {
    _total_tasks = 0;
    for (PipelinePtr& pipeline : _pipelines) {
        // if sink
        auto sink = pipeline->sink()->build_operator();
        sink->init(request.fragment.output_sink);

        Operators operators;
        RETURN_IF_ERROR(pipeline->build_operators(operators));
        auto task =
                std::make_unique<PipelineTask>(pipeline, _total_tasks++,
_runtime_state.get(),
                                               operators, sink, this,
pipeline->pipeline_profile());
        sink->set_child(task->get_root());
        _tasks.emplace_back(std::move(task));
        _runtime_profile->add_child(pipeline->pipeline_profile(), true, nullptr);
    }

    for (auto& task : _tasks) {
        RETURN_IF_ERROR(task->prepare(_runtime_state.get()));
    }
```

```
    // register the profile of child data stream sender
    for (auto& sender : _multi_cast_stream_sink_senders) {
        _sink->profile()->add_child(sender->profile(), true, nullptr);
    }

    return Status::OK();
}

Status Pipeline::build_operators(Operators& operators) {
    OperatorPtr pre;
    for (auto& operator_t : _operator_builders) {
        auto o = operator_t->build_operator();
        if (pre) {
            o->set_child(pre);
        }
        operators.emplace_back(o);
        pre = std::move(o);
    }
    return Status::OK();
}

#define OPERATOR_CODE_GENERATOR(NAME, SUBCLASS)
\
    NAME##Builder::NAME##Builder(int32_t id, ExecNode* exec_node)         \
            : OperatorBuilder(id, #NAME, exec_node) {}                    \

\
    OperatorPtr NAME##Builder::build_operator() {                         \
        return std::make_shared<NAME>(this, _node);                        \
    }
\


\
    NAME::NAME(OperatorBuilderBase* operator_builder, ExecNode* node) \
            : SUBCLASS(operator_builder, node) {};
```

至此，我们就从执行计划树生成了一组等价的、可以直接被调度执行的PipelineTask。

## 4.3 调度与执行逻辑

Pipeline模型相比于Volcano模型的一个重要优势就是实现了用户级别的任务调度。我们所调度的对象就是之前实现拆分的 PipelineTask。而在调度器的选择

上，Doris 实现了类似于 Linux 操作系统中的调度算法，通过多级反馈队列（Multi-Level Feedback Queue，MLFQ）来实现 PipelineTask 的调度执行。多级反馈队列是目前实践意义上几乎最好的调度算法。

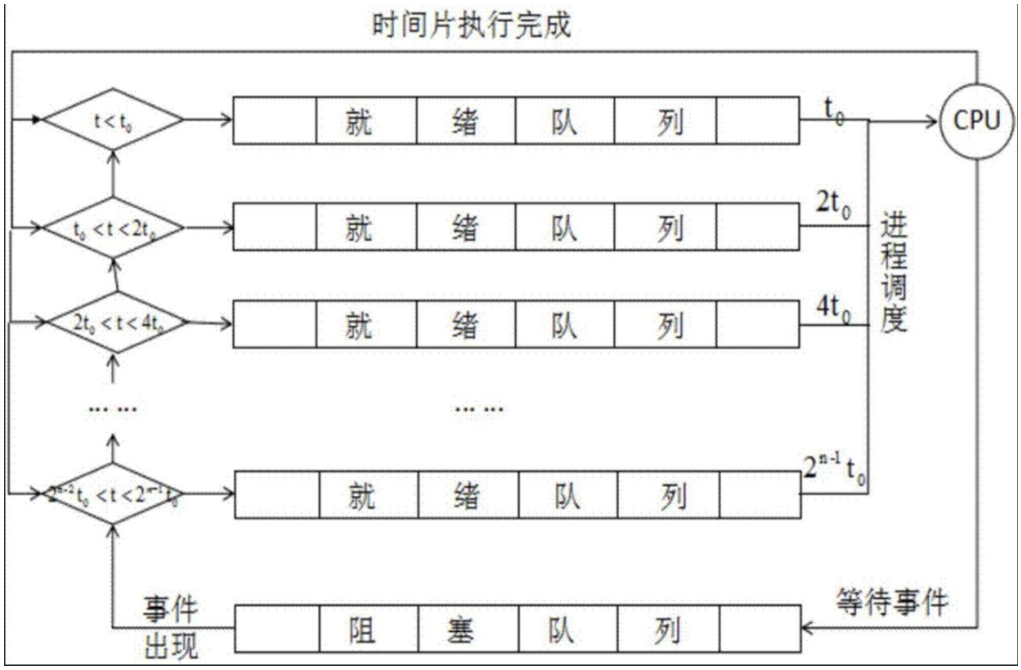


图 4.1 多级反馈队列调度

在 MLFQ 中，新进入的 PipelineTask 将被放入最高一级的子队列。当调度器需要从 MLFQ 中拉取任务执行时，总是由高级别队列向低级别队列遍历，选取第一个遇到的 PipelineTask。每个 PipelineTask 每次执行都有一个时间片限制，如果执行超出时间片限制仍未结束，任务将会被放回队列。放回时，PipelineTask 将会进入它被取到的队列的下一级队列。每级队列的时间片限制是相同的，总是相邻上一级队列的 2 倍。

具体实现细节上，首先是 MLFQ 本身的实现，这里我们实现了 TaskQueue、SubTaskQueue、PriorityTaskQueue 三个核心类型，以及两个为资源隔离环境进行任务调度的上层封装类 MultiCoreTaskQueue、TaskGroupTaskQueue：

```
class TaskQueue {
public:
    TaskQueue(size_t core_size) : _core_size(core_size) {}
    virtual ~TaskQueue();
    virtual void close() = 0;
    // Get the task by core id.
    virtual PipelineTask* take(size_t core_id) = 0;

    // push from scheduler
    virtual Status push_back(PipelineTask* task) = 0;

    // push from worker
```

```
        virtual Status push_back(PipelineTask* task, size_t core_id) = 0;

        virtual void update_statistics(PipelineTask* task, int64_t time_spent) {}

        virtual void update_tg_cpu_share(const taskgroup::TaskGroupInfo&
task_group_info,
                                         taskgroup::TaskGroupPtr task_group) =
0;

        int cores() const { return _core_size; }

protected:
        size_t _core_size;
        static constexpr auto WAIT_CORE_TASK_TIMEOUT_MS = 100;
};

class SubTaskQueue {
        friend class PriorityTaskQueue;

public:
        void push_back(PipelineTask* task) { _queue.emplace(task); }

        PipelineTask* try_take(bool is_steal);

        void set_level_factor(double level_factor) { _level_factor = level_factor; }

        double get_vruntime() { return _runtime / _level_factor; }

        void inc_runtime(uint64_t delta_time) { _runtime += delta_time; }

        void adjust_runtime(uint64_t vruntime) { this->_runtime = vruntime *
_level_factor; }

        bool empty() { return _queue.empty(); }

private:
        std::queue<PipelineTask*> _queue;
        // depends on LEVEL_QUEUE_TIME_FACTOR
        double _level_factor = 1;

        std::atomic<uint64_t> _runtime = 0;
};

// A Multilevel Feedback Queue
```

```
class PriorityTaskQueue {
public:
    explicit PriorityTaskQueue();

    void close();

    PipelineTask* try_take_unprotected(bool is_steal);

    PipelineTask* try_take(bool is_steal);

    PipelineTask* take(uint32_t timeout_ms = 0);

    Status push(PipelineTask* task);

    void inc_sub_queue_runtime(int level, uint64_t runtime) {
        _sub_queues[level].inc_runtime(runtime);
    }

private:
    static constexpr auto LEVEL_QUEUE_TIME_FACTOR = 2;
    static constexpr size_t SUB_QUEUE_LEVEL = 6;
    SubTaskQueue _sub_queues[SUB_QUEUE_LEVEL];
    // 1s, 3s, 10s, 60s, 300s
    uint64_t _queue_level_limit[SUB_QUEUE_LEVEL - 1] = {1000000000,
3000000000, 10000000000,
                                                        60000000000,
300000000000};
    std::mutex _work_size_mutex;
    std::condition_variable _wait_task;
    std::atomic<size_t> _total_task_size = 0;
    bool _closed;

    // used to adjust vruntime of a queue when it's not empty
    // protected by lock _work_size_mutex
    uint64_t _queue_level_min_vruntime = 0;

    int _compute_level(uint64_t real_runtime);
};

// Need consider NUMA architecture
class MultiCoreTaskQueue : public TaskQueue {
public:
    explicit MultiCoreTaskQueue(size_t core_size);
```

```
    ~MultiCoreTaskQueue() override;

    void close() override;

    // Get the task by core id.
    PipelineTask* take(size_t core_id) override;

    Status push_back(PipelineTask* task) override;

    Status push_back(PipelineTask* task, size_t core_id) override;

    void update_statistics(PipelineTask* task, int64_t time_spent) override {
        task->inc_runtime_ns(time_spent);

_prio_task_queue_list[task->get_core_id()].inc_sub_queue_runtime(task->get_queu
e_level(),

time_spent);
    }

    void update_tg_cpu_share(const taskgroup::TaskGroupInfo& task_group_info,
                             taskgroup::TaskGroupPtr task_group) override {
        LOG(FATAL) << "update_tg_cpu_share not implemented";
    }

private:
    PipelineTask* _steal_take(size_t core_id);

    std::unique_ptr<PriorityTaskQueue[]> _prio_task_queue_list;
    std::atomic<size_t> _next_core = 0;
    std::atomic<bool> _closed;
};

class TaskGroupTaskQueue : public TaskQueue {
public:
    explicit TaskGroupTaskQueue(size_t);
    ~TaskGroupTaskQueue() override;

    void close() override;

    PipelineTask* take(size_t core_id) override;

    // from TaskScheduler or BlockedTaskScheduler
    Status push_back(PipelineTask* task) override;
```

```
    // from worker
    Status push_back(PipelineTask* task, size_t core_id) override;

    void update_statistics(PipelineTask* task, int64_t time_spent) override;

    void update_tg_cpu_share(const taskgroup::TaskGroupInfo& task_group_info,
                             taskgroup::TaskGroupPtr task_group) override;

private:
    template <bool from_executor>
    Status _push_back(PipelineTask* task);
    template <bool from_worker>
    void _enqueue_task_group(taskgroup::TGEntityPtr);
    void _dequeue_task_group(taskgroup::TGEntityPtr);
    taskgroup::TGEntityPtr _next_tg_entity();
    int64_t _ideal_runtime_ns(taskgroup::TGEntityPtr tg_entity) const;
    void _update_min_tg();

    // Like cfs rb tree in sched_entity
    struct TaskGroupSchedEntityComparator {
        bool operator()(const taskgroup::TGEntityPtr&, const
taskgroup::TGEntityPtr&) const;
    };
    using ResouceGroupSet = std::set<taskgroup::TGEntityPtr,
TaskGroupSchedEntityComparator>;
    ResouceGroupSet _group_entities;
    std::condition_variable _wait_task;
    std::mutex _rs_mutex;
    bool _closed = false;
    int _total_cpu_share = 0;
    std::atomic<taskgroup::TGEntityPtr> _min_tg_entity = nullptr;
    uint64_t _min_tg_v_runtime_ns = 0;
};
```

这里我们主要分析完成基本功能的 SubTaskQueue 和 PriorityTaskQueue。PriorityTaskQueue 本身是一个完整的 MLFQ，SubTaskQueue 则是其中具体的一级队列：

```
    static constexpr auto LEVEL_QUEUE_TIME_FACTOR = 2;
    static constexpr size_t SUB_QUEUE_LEVEL = 6;
    SubTaskQueue _sub_queues[SUB_QUEUE_LEVEL];
    // 1s, 3s, 10s, 60s, 300s
    uint64_t _queue_level_limit[SUB_QUEUE_LEVEL - 1] = {1000000000,
3000000000, 10000000000, 60000000000, 300000000000};
```

可以看到，它首先包含了一组 SubTaskQueue 作为其子队列。每一个 SubTaskQueue 构成一个级别，每级之间具有逐渐递增的 level_factor：

```
PriorityTaskQueue::PriorityTaskQueue() : _closed(false) {
    double factor = 1;
    for (int i = 0; i < SUB_QUEUE_LEVEL; ++i) {
        _sub_queues[i].set_level_factor(factor);
        factor *= LEVEL_QUEUE_TIME_FACTOR;
    }
}
```

而_queue_level_limit 限定了每级反馈队列允许的执行时间：

```
int PriorityTaskQueue::_compute_level(uint64_t runtime) {
    for (int i = 0; i < SUB_QUEUE_LEVEL - 1; ++i) {
        if (runtime <= _queue_level_limit[i]) {
            return i;
        }
    }
    return SUB_QUEUE_LEVEL - 1;
}

Status PriorityTaskQueue::push(PipelineTask* task) {
    if (_closed) {
        return Status::InternalError("WorkTaskQueue closed");
    }
    auto level = _compute_level(task->get_runtime_ns());
    std::unique_lock<std::mutex> lock(_work_size_mutex);

    // update empty queue's    runtime, to avoid too high priority
    if (_sub_queues[level].empty() &&
        _queue_level_min_vruntime > _sub_queues[level].get_vruntime()) {
        _sub_queues[level].adjust_runtime(_queue_level_min_vruntime);
    }

    _sub_queues[level].push_back(task);
    _total_task_size++;
    _wait_task.notify_one();
    return Status::OK();
}
```

可以看出，在 PipelineTask 被放入或重新放回队列时，我们会通过统计的执行时间计算出它应该被放入哪一级队列。

在多级反馈队列的上层，实现调度逻辑的是 PipelineScheduler，其中 BlockedTaskScheduler 用于调度阻塞的 PipelineTask，TaskScheduler 用于存放非阻塞的 PipelineTask：

```
class BlockedTaskScheduler {
public:
    explicit BlockedTaskScheduler(std::shared_ptr<TaskQueue> task_queue)
            : _task_queue(std::move(task_queue)), _started(false),
_shutdown(false) {}

    ~BlockedTaskScheduler() = default;

    Status start();
    void shutdown();
    void add_blocked_task(PipelineTask* task);

private:
    std::shared_ptr<TaskQueue> _task_queue;

    std::mutex _task_mutex;
    std::condition_variable _task_cond;
    std::list<PipelineTask*> _blocked_tasks;

    scoped_refptr<Thread> _thread;
    std::atomic<bool> _started;
    std::atomic<bool> _shutdown;

    static constexpr auto EMPTY_TIMES_TO_YIELD = 64;

private:
    void _schedule();
    void _make_task_run(std::list<PipelineTask*>& local_tasks,
                        std::list<PipelineTask*>::iterator& task_itr,
                        std::vector<PipelineTask*>& ready_tasks,
                        PipelineTaskState state =
PipelineTaskState::RUNNABLE);
};

class TaskScheduler {
public:
    TaskScheduler(ExecEnv* exec_env, std::shared_ptr<BlockedTaskScheduler>
b_scheduler,
                  std::shared_ptr<TaskQueue> task_queue)
            : _task_queue(std::move(task_queue)),
              _exec_env(exec_env),
              _blocked_task_scheduler(std::move(b_scheduler)),
              _shutdown(false) {}
```

```
    ~TaskScheduler();

    Status schedule_task(PipelineTask* task);

    Status start();

    void shutdown();

    ExecEnv* exec_env() { return _exec_env; }

private:
    std::unique_ptr<ThreadPool> _fix_thread_pool;
    std::shared_ptr<TaskQueue> _task_queue;
    std::vector<std::unique_ptr<std::atomic<bool>>> _markers;
    ExecEnv* _exec_env;
    std::shared_ptr<BlockedTaskScheduler> _blocked_task_scheduler;
    std::atomic<bool> _shutdown;

private:
    void _do_work(size_t index);
    void _try_close_task(PipelineTask* task, PipelineTaskState state);
};
```

在 TaskScheduler 开始工作后，每个 CPU 核心都会持续调度对应的 PipelineTask：

```
Status TaskScheduler::start() {
    int cores = _task_queue->cores();
    // Must be mutil number of cpu cores
    ThreadPoolBuilder("TaskSchedulerThreadPool")
            .set_min_threads(cores)
            .set_max_threads(cores)
            .set_max_queue_size(0)
            .build(&_fix_thread_pool);
    _markers.reserve(cores);
    for (size_t i = 0; i < cores; ++i) {
        _markers.push_back(std::make_unique<std::atomic<bool>>(true));
        RETURN_IF_ERROR(

_fix_thread_pool->submit_func(std::bind(&TaskScheduler::_do_work, this, i)));
    }
    return _blocked_task_scheduler->start();
}
```

而核心的调度逻辑在 TaskScheduler::_do_work 中实现：

```
void TaskScheduler::_do_work(size_t index) {
    const auto& marker = _markers[index];
    while (*marker) {
```

```
            auto* task = _task_queue->take(index);
            if (!task) {
                continue;
            }
            task->set_task_queue(_task_queue.get());
            auto* fragment_ctx = task->fragment_context();
            signal::query_id_hi = fragment_ctx->get_query_id().hi;
            signal::query_id_lo = fragment_ctx->get_query_id().lo;
            bool canceled = fragment_ctx->is_canceled();

            auto check_state = task->get_state();
            if (check_state == PipelineTaskState::PENDING_FINISH) {
                DCHECK(!task->is_pending_finish()) << "must not pending close " <<
task->debug_string();
                _try_close_task(task,
                                canceled ? PipelineTaskState::CANCELED :
PipelineTaskState::FINISHED);
                continue;
            }
            DCHECK(check_state != PipelineTaskState::FINISHED &&
                   check_state != PipelineTaskState::CANCELED)
                    << "task already finish";

            if (canceled) {
                // may change from pending FINISH，should called cancel
                // also may change form BLOCK, other task called cancel

                // If pipeline is canceled caused by memory limit, we should send
report to FE in order
                // to cancel all pipeline tasks in this query
                fragment_ctx->send_report(true);
                _try_close_task(task, PipelineTaskState::CANCELED);
                continue;
            }

            DCHECK(check_state == PipelineTaskState::RUNNABLE);
            // task exec
            bool eos = false;
            auto status = Status::OK();

            try {
                status = task->execute(&eos);
            } catch (const Exception& e) {
                status = Status::Error(e.code(), e.to_string());
```

```
            }

            task->set_previous_core_id(index);
            if (!status.ok()) {
                LOG(WARNING) << fmt::format("Pipeline task failed. reason: {}, task:
\n{}",
                                            status.to_string(),
task->debug_string());
                // exec failed，cancel all fragment instance

fragment_ctx->cancel(PPlanFragmentCancelReason::INTERNAL_ERROR,
status.to_string());
                fragment_ctx->send_report(true);
                _try_close_task(task, PipelineTaskState::CANCELED);
                continue;
            }

            if (eos) {
                // TODO: pipeline parallel need to wait the last task finish to call
finalize
                //    and find_p_dependency
                status = task->finalize();
                if (!status.ok()) {
                    // execute failed，cancel all fragment

fragment_ctx->cancel(PPlanFragmentCancelReason::INTERNAL_ERROR,
                                                "finalize fail:" + status.to_string());
                    _try_close_task(task, PipelineTaskState::CANCELED);
                } else {
                    task->finish_p_dependency();
                    _try_close_task(task, PipelineTaskState::FINISHED);
                }
                continue;
            }

            auto pipeline_state = task->get_state();
            switch (pipeline_state) {
            case PipelineTaskState::BLOCKED_FOR_SOURCE:
            case PipelineTaskState::BLOCKED_FOR_SINK:
            case PipelineTaskState::BLOCKED_FOR_RF:
            case PipelineTaskState::BLOCKED_FOR_DEPENDENCY:
                _blocked_task_scheduler->add_blocked_task(task);
                break;
            case PipelineTaskState::RUNNABLE:
```

```
                _task_queue->push_back(task, index);
                break;
            default:
                DCHECK(false) << "error state after run task, " <<
get_state_name(pipeline_state);
                break;
            }
        }
}
```

这里的核心逻辑是，从 MLFQ 中每次取出一个 PipelineTask 执行。而 MLFQ 本身向外提供了 take()接口，内部实现了正确的调度方式。通过这种调度，我们成功保证了 pipeline 引擎的调度要求。

## 4.4 资源隔离

对于一个 OLAP 引擎来说，它的实际应用场景往往需要支撑一个甚至多个完整的实际业务系统的分析需求。在这种情况下，单纯的“快速查询”已经不能满足业务系统中的全部需求。在实际业务中，我们往往需要提供以下几点能力：

a. 对于某些重要的查询任务或者业务组（在数据库系统中对应到具体的用户或用户组），我们需要优先保证他们的执行响应时间；

b. 对于用户来说，需要设置资源访问权限以及硬件资源的使用权限；

c. 对于特定任务，我们需要划分一定的资源实现定向保障；

d. 在线和离线业务混布场景，需要在在线查询压力较低时部署较高的离线查询以避免浪费 CPU 资源，在在线查询压力较大时又需要降低离线查询压力以免阻塞在线业务。

这些需求指向了对一个重要基础能力——资源隔离的需求。这在能够实现动态、无感知扩容的云原生数据库领域显得尤为重要。资源隔离的目的是实现 CPU、内存等资源的拆分与灵活控制，可以限制查询任务对计算资源的消耗，目标是让不同租户的查询任务在同一集群执行能兼顾多方响应时间并且保证集群资源利用率。

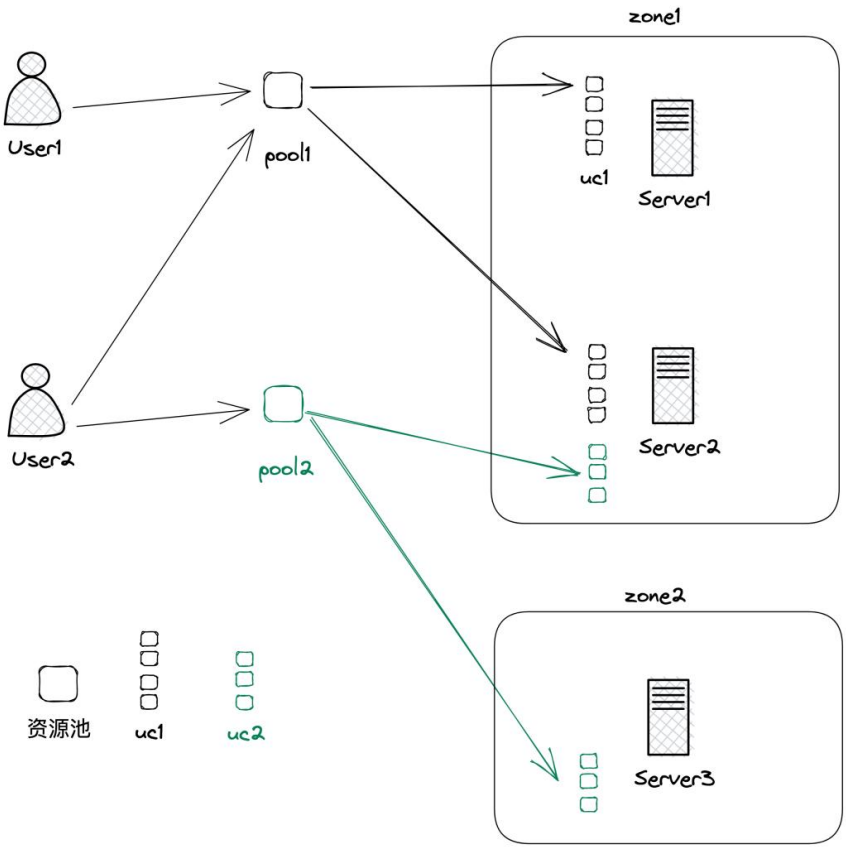


图 4.2 资源隔离

因此，在 Doris 的整个系统中，我们也实现了资源隔离的功能，允许将某些资源绑定为一个资源组，并将资源组开放给特定用户使用。这样，通过管控用户级别的权限，就实现了资源的管控隔离。

## 4.5 Scan 池化

在 Doris 的存储层设计中，VScanNode 本身并不会直接去进行数据的读取操作，而是通过持有的 Scanner 去实现这一过程。在传统的设计中，这一过程会导致大量的 Scanner 创建和销毁、对应的数据也有重复扫描导致浪费的可能。于是我们引入了 Scan 池化的逻辑，让 Scanner 也组成一个类似线程池的“Scanner 池”。这样，就避免了 Scanner 重复创建和销毁的开销、降低了 Scan 任务来临时的等待时间，以及提供了统一监测、管理 Scan 压力的方法和能力。

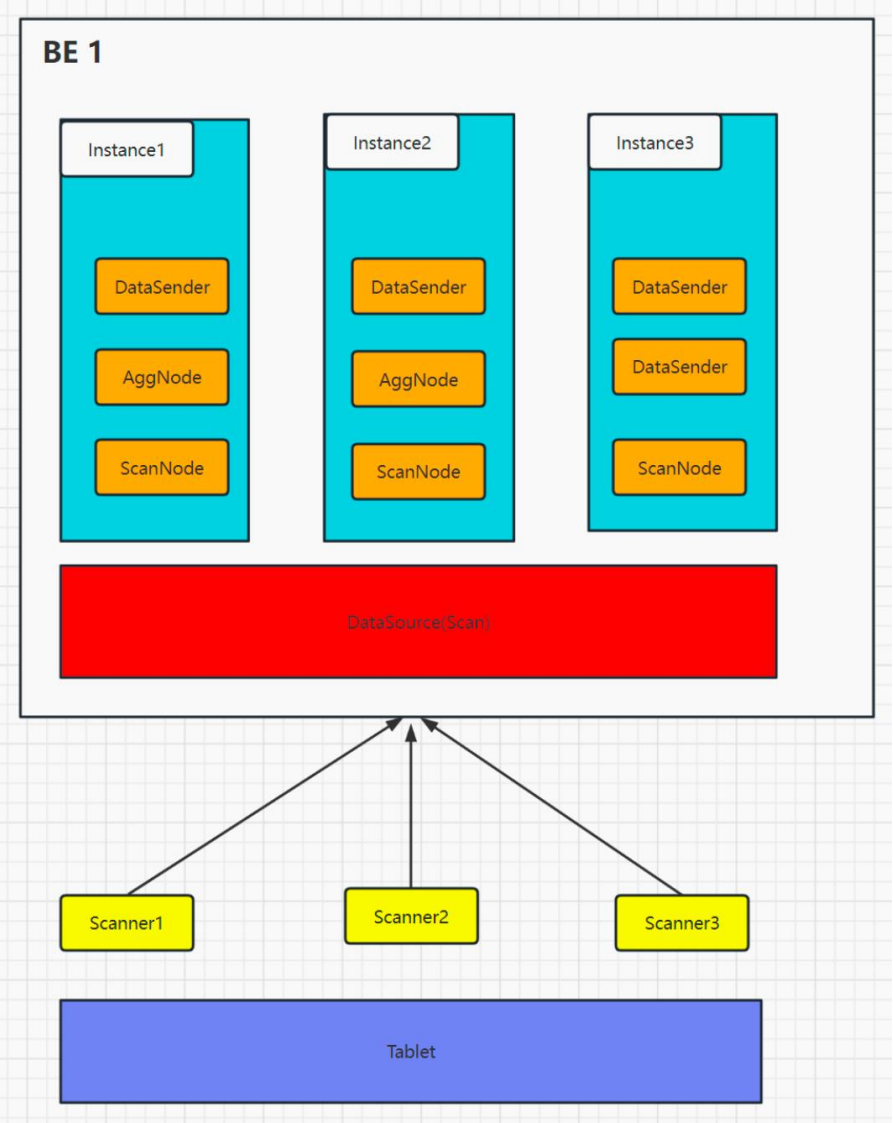


图 4.3 Scan 池化

```
class SharedScannerController {
public:
    std::pair<bool, int> should_build_scanner_and_queue_id(int my_node_id) {
        std::lock_guard<std::mutex> lock(_mutex);
        auto it = _scanner_parallel.find(my_node_id);

        if (it == _scanner_parallel.cend()) {
            _scanner_parallel.insert({my_node_id, 0});
            return {true, 0};
        } else {
            auto queue_id = it->second;
            _scanner_parallel[my_node_id] = queue_id + 1;
            return {false, queue_id + 1};
        }
    }
```

```
        void set_scanner_context(int my_node_id,
                                 const std::shared_ptr<ScannerContext>
scanner_context) {
            std::lock_guard<std::mutex> lock(_mutex);
            _scanner_context.insert({my_node_id, scanner_context});
        }

        bool scanner_context_is_ready(int my_node_id) {
            std::lock_guard<std::mutex> lock(_mutex);
            return _scanner_context.find(my_node_id) != _scanner_context.end();
        }

        std::shared_ptr<ScannerContext> get_scanner_context(int my_node_id) {
            std::lock_guard<std::mutex> lock(_mutex);
            return _scanner_context[my_node_id];
        }

    private:
        std::mutex _mutex;
        std::map<int /*node id*/, int /*parallel*/> _scanner_parallel;
        std::map<int /*node id*/, std::shared_ptr<ScannerContext>>
_scanner_context;
    };
```

在开启 Scan 池化以后，可以看到，通过 Clickbench 测试集进行测试，使用涉及到大表扫描的查询语句 select sum(CounterID),OS from hits group by OS order by OS 进行测试，结果取得了显著的性能改善：

表 4.1 开启 Scan 池化后 Doris 的性能比较

| | 1-instance | 16-instance |
|---|---|---|
| 开启 Scan 池化，并行 | 0.75s | **0.12s** |
| 开启 Scan 池化，非并行 | 0.75s | **0.43s** |
| 不开启 Scan 池化 | 0.72s | 0.7s |
| 某友商 | 0.57s | 0.62s |

使用 Clickbench-100G 测试集在 1tablet 分布下进行完整测试，开启 Scan 池化的结果显著优于不开启 Scan 池化的结果：

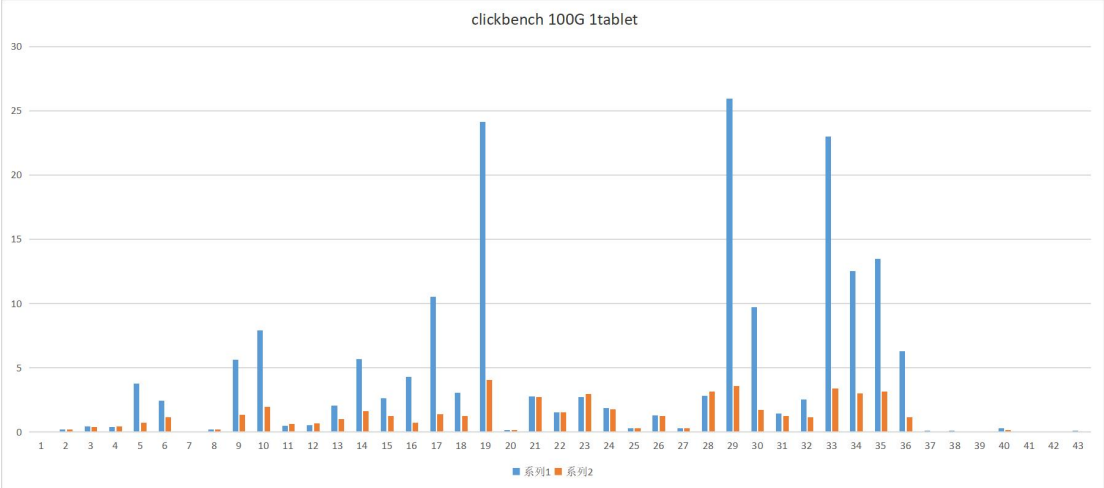


图 4.4 Scan 池化后 Clickbench 100G 测试结果

由此我们可以确认，Scan 池化的确能够充分现代多核 CPU 的能力，提高实际 IO 操作的并行性。

## 4.6 性能优化

### 4.6.1 列式存储

在实现 Pipeline 引擎替代 Volcano 引擎的基础上，我们还使用了多种技术手段进行性能优化，其中最基础的一个就是列式存储功能。

相比于 OLTP 场景经常面对的高并发点查，OLAP 场景下，更多的使用情景是仅查询某个表中的特定部分列，且一般会涉及全表数据的分析。此时如果使用行式存储，几乎总是会形成全表扫描，且所拉取的大多数数据都不会被当前 SQL 使用。这种存储方式会造成 CPU 资源极大的浪费。

因此，以列而非行为单位存储和使用数据，对于 OLAP 场景而言是非常重要的优化。

在 Doris 中，对于一个 Block 而言，它实际所存储的数据是这样的：

```
class Block {
        ENABLE_FACTORY_CREATOR(Block);

private:
        using Container = ColumnsWithTypeAndName;
        using IndexByName = phmap::flat_hash_map<String, size_t>;
        Container data;
        IndexByName index_by_name;
        std::vector<bool> row_same_bit;

        int64_t _decompress_time_ns = 0;
```

```
    int64_t _decompressed_bytes = 0;

    mutable int64_t _compress_time_ns = 0;
    ……

struct ColumnWithTypeAndName {
    ColumnPtr column;
    DataTypePtr type;
    String name;
    ……
```

可以看出，Block 中实际存储的对象是 ColumnPtr 也就是指向列的指针，而不是行。

列式存储也为向量化引擎采取 SIMD 优化提供了非常大的便利。由于相同类型的同列数据在内存上是相邻排列的，因此 SIMD 更有机会在列式存储的数据库中对查询执行的操作生效。

### 4.6.2 SIMD 优化

SIMD（Single Instruction Multiple Data，单指令多数据）是现代 CPU 指令集提供的一种强力优化技术。普通的 CPU 指令，单条指令只能处理一组寄存器；但 SIMD 指令集中提供的指令可以在单条指令的周期内对大量数据（如 16 组、32 组等）重复执行相同的指令。只用略微多于以往一组指令的执行开销，完成对大量数据的重复数学运算。对于密集的简单计算、赋值、比较等操作及其组合，使用 SIMD 指令进行优化，可以实现巨大的性能提升。

多数时候，对于较为简单的操作，主流 C++编译器（如 GCC、CLANG）都能完成自动生成 SIMD 代码的操作（称为 auto-vectorization，自动向量化），例如：

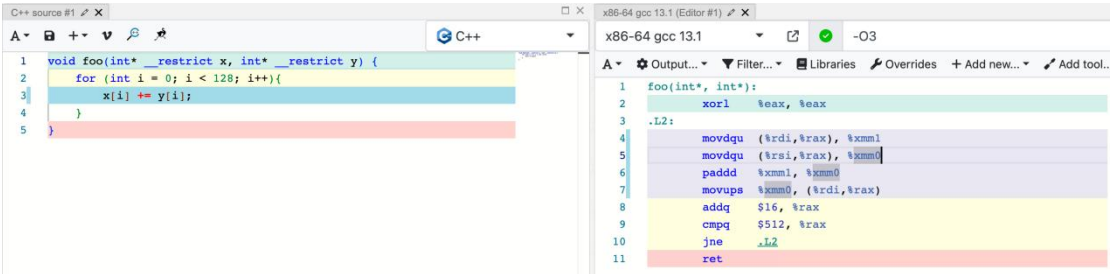


图 4.5 编译器自动向量化

但是对于更为复杂的场景，尤其是需要应用某些先验信息进行优化的情况，auto-vectorization 就显得无能为力了。此时，就需要我们使用对应平台（如 x86、arm）提供的 SIMD 指令接口，手动实现 SIMD 优化。例如：

```
template <typename Char>
int memcmp_small_allow_overflow15(const Char* a, size_t a_size, const Char* b,
size_t b_size) {
```

```
    size_t min_size = std::min(a_size, b_size);
    for (size_t offset = 0; offset < min_size; offset += 16) {
        uint16_t mask = _mm_movemask_epi8(
                _mm_cmpeq_epi8(_mm_loadu_si128(reinterpret_cast<const
__m128i*>(a + offset)), _mm_loadu_si128(reinterpret_cast<const __m128i*>(b +
offset))));
        mask = ~mask;

        if (mask) {
            offset += __builtin_ctz(mask);
            if (offset >= min_size) break;
            return detail::cmp(a[offset], b[offset]);
        }
    }

    return detail::cmp(a_size, b_size);
}
```

可以看到，这时我们通过_mm_cmpeq_epi8 手动指明使用 SIMD 指令进行 128 位的 8bits 比较，理论上局部代码可以提升约 16 倍的性能。

## 4.7 性能测试

使用 Clickbench-100G 数据进行性能测试，结果显示在切换至 Pipeline 查询引擎之后，Doris 的性能表现已经远超使用 Volcano 模型时的表现。对比行业内较为突出的友商，整体性能表现也有 6%左右的优势。

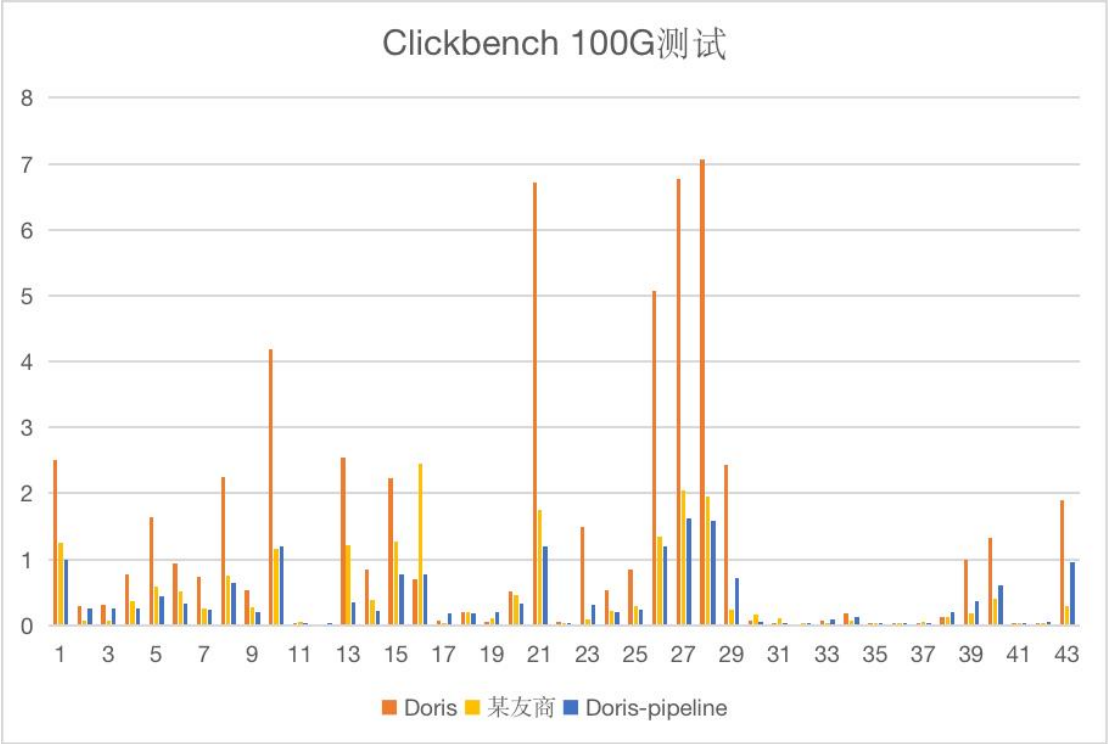


图 4.6 Clickbench 100G 测试结果

# 5 总结与展望

本文首先总结了 Apache Doris 这一成熟的商业化 OLAP 数据库架构、设计及其性能优化技巧。在这一基础上，本文介绍并实现了一款 Pipeline 数据库查询执行引擎。经过详细的分析发现，传统数据库 Volcano 引擎由于设计较为落后，存在着大量根本性问题制约性能提升。主要包括阻塞操作过多、数据局部性不佳、虚函数调用难以避免等。而 Pipeline 引擎由于从数据角度出发的设计，始终把握“小数据块的数据驱动”，从根本上避免了 Volcano 引擎的大多数原生缺点。在 Apache Doris 数据库中使用 Pipeline 引擎替代后，经过测试发现，Volcano 引擎的原有问题得到了显著的改善。

经过多款 OLAP 领域内公认事实标准的测试，Apache Doris 数据库在应用了 Pipeline 执行引擎之后，性能表现得到显著提升，目前稳居世界前列。其中在单表场景测试集 Clickbench 上稳定排名世界第一，相比于应用前取得了巨大的性能提升。在多核大型服务器上，能够在复杂大查询上取得数倍至数百倍的性能提升。最终测试结果表明，Pipeline 引擎的设计和实现是成功的，可以期待它在实际商业应用中取得更大的成绩。

同时，基于 Pipeline 引擎提供的对查询执行的灵活调度能力，我们实现了资源隔离功能，使得 Apache Doris 数据库在商业使用上更加可靠，能够适应更多复杂场景的实际业务需求。

接下来，应当基于该引擎取得的结果，针对复杂多表聚合场景的优化进行更进一步的优化，实现全面的性能领先。同时，Pipeline 引擎引入的机制在操作系统的线程调度之上增添了一套用户态的调度策略，使得系统调试、分析变得非常复杂。未来应当在 Pipeline 的执行分析、展示方向继续发力，产出更加友好的分析、调试信息，使 Doris 易用性进一步提高。

就 Pipeline 引擎自身而言，它仍然存在一定无法避免的虚函数调用即函数跳转，未来随着行业发展、数据量进一步增大，仍然可能成为性能瓶颈。未来仍可能采取编译执行模型等方式，将 SQL 语句实际编译成对应的完整二进制代码进行执行，可以达到进一步消除虚函数、函数调用等带来的性能开销的目的，有希望取得更深层次的全面性能改善。

# 参考文献

[1] 前瞻产业研究院. 2019 中国大数据行业研究报告. http://pdf.dfcfw.com/pdf/H3_AP201911251371103072_1.pdf, 2019

[2] 中国信息通信研究院. 大数据白皮书. http://www.caict.ac.cn/english/research/whitepapers/202303/P020230316608528378472.pdf, 2023-01

[3] 一个会写诗的程序员. 主流的 OLAP 引擎介绍. https://cloud.tencent.com/developer/article/1924583, 2021-12-24

[4] Apache Doris. Doris 简史-为分析而生的 11 年. https://xie.infoq.cn/article/4bdf3da72bc868ad78cf6bf4b, 2021-03-24

[5] 魏祚. 最佳实践: Apache Doris 在小米数据场景的应用实践与优化. https://doris.apache.org/zh-CN/blog/xiaomi_vector/#新旧架构性能对比, 2022-12-08

[6] 梁程加, 陈俊云, 许英博. 数据库：企业数字化支撑，大数据时代基石. 中信证券前瞻研究系列报告, 2021(86)

[7] 中金公司. 数智中国之二：数据库商业市场五问五答. https://research.cicc.com/frontend/recommend/detail?id=3100, 2022-06-08

[8] 朱良. Apache Doris 在美团外卖数仓中的应用实践. https://tech.meituan.com/2020/04/09/doris-in-meituan-waimai.html, 2022-04-09

[9] Apache Doris. Doris 介绍. https://doris.apache.org/zh-CN/docs/dev/summary/basic-summary/, 2023

[10] QIN. 物理执行引擎之火山引擎. https://www.qin.news/wu-li-zhi-xing-yin-qing-zhi-huo-shan-yin-qing/, 2022-02-20

[11] OceanBase. 数据库查询引擎的进化之路. https://zhuanlan.zhihu.com/p/41562506, 2018-08-15

[12] caroly. 分布式数据库（九）. https://caroly.fun/archives/分布式数据库九, 2021-05-15

[13] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. 2014 ACM SIGMOD International Conference, 2014, 743–754.

[14] G. Graefe. Volcano— An Extensible and Parallel Query Evaluation System.

IEEE Trans, 1994, 120–135.

[15] G. Graefe,W. J. McKenna. The Volcano optimizer generator: extensibility and efficient search. IEEE 9th International Conference on Data Engineering, 1993, 209-218

[16] Thomas Neumann. . Efficiently compiling efficient query plans for modern hardware. VLDB Endow, 2011, 539-550.

[17] Hector Garcia Molina, Jeffery D. Ullman, Jennifer Widom. 数据库系统全书. 岳丽华, 杨冬青, 黄育昌等. 北京：机械工业出版社，2003.

[18] Abraham Silberschatz, Henry F.Korth, S.Sudarshan . 数据库系统概念. 杨冬青, 李红燕, 唐世渭. 北京：机械工业出版社，2012.

[19] 陈海波, 夏虞斌等. 现代操作系统原理与实现.北京：机械工业出版社，2020

# 致　谢

在此，作者要衷心感谢本人的导师姚红革副教授，他是作者在科研领域中的领路人与引导者。感谢他在作者本科期间给予的关怀与指导，尤其是极尽耐心的包容。

同时，作者还要感谢本人在 Apache Doris 社区开发期间的 mentor，虽然因为商业原因作者无法在这里提及他的姓名，但与其共事、受之指导的日子，是作者在技术上获得最大进步的时光。作为一名已经证明过自己的、饱受行业赞誉的卓越工程师，相信在他的引领下，Apache Doris 会在技术上始终保持世界一流水准。

最终，作者还要感谢飞轮、美团、百度、腾讯等公司的前辈和同事们，以及其他优秀的自由开发者们，是你们的才华与奋斗，造就了 Apache Doris 这一天才般的杰作。相信在我们的共同努力下，Doris 将会在大数据领域的历史舞台上绽放更耀眼的光芒。

# 毕业设计（论文）知识产权声明

本人完全了解西安工业大学有关保护知识产权的规定，即：本科学生在校攻读学士学位期间毕业设计（论文）工作的知识产权属于西安工业大学。本人保证毕业离校后，使用毕业设计（论文）工作成果或用毕业设计（论文）工作成果发表论文时署名单位仍然为西安工业大学。学校有权保留送交的毕业设计（论文）的原文或复印件，允许毕业设计（论文）被查阅和借阅；学校可以公布毕业设计（论文）的全部或部分内容，可以采用影印、缩印或其他复制手段保存毕业设计（论文）。

（保密的毕业设计（论文）在解密后应遵守此规定）

毕业设计（论文）作者签名：

指导教师签名：

日期：

# 毕业设计（论文）独创性声明

秉承学校严谨的学风与优良的科学道德，本人声明所呈交的毕业设计（论文）是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，毕业设计（论文）中不包含其他人已经发表或撰写过的成果，不包含他人已申请学位或其他用途使用过的成果。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了致谢。

毕业设计（论文）与资料若有不实之处，本人承担一切相关责任。

毕业设计（论文）作者签名：

指导教师签名：

日期：

附录

西安工业大学
XI'AN TECHNOLOGICAL UNIVERSITY

# 毕业论文外文资料

院 （系）：计算机科学与工程学院

专　　业：计算机科学与技术

班　　级：18060314

姓　　名：赵长乐

学　　号：18030513121

指导教师：杨国梁

外文出处：SIGMOD 2014

附　　件：1.译文；2.原文

2023 年 4 月

# 附件 1 译文

# 小块数据驱动并行： 面向多核时代的 NUMA 有感知的查询评估框架


## 摘要

随着现代计算机体系结构的发展，有两个问题阻碍了最先进的并行查询执行方法：（1）为了利用多核的优势，所有的查询工作必须平均分配到（很快）数百个线程中，以达到良好的速度，然而（2）由于现代失序核的复杂性，即使有准确的数据统计，平均分配工作也很困难。因此，现有的 ″计划驱动 ″的并行方法遇到了负载平衡和上下文切换的瓶颈，因此不再有规模。多核架构面临的第三个问题是内存控制器的分散化，这导致了非统一内存访问（NUMA）。

作为回应，我们提出了 ″Morsel 驱动 ″的查询执行框架，其中调度成为一个细粒度的运行时任务，是 NUMA 感知的。Morsel 驱动的查询处理采用小块的输入数据（″小块″），并将这些数据调度给工人线程，这些线程运行整个运算器管道，直到下一个管道中断。并行性的程度并没有在计划中体现出来，而是可以在查询执行过程中弹性地改变，因此，调度员可以根据不同小块的执行速度重新行动，也可以根据工作负载中新到来的查询动态地调整资源。此外，调度器知道 NUMA 本地模块的数据位置和操作者的状态，这样，绝大多数的执行都是在 NUMA 本地内存上进行的。我们对 TPC-H 和 SSB 基准的评估显示了极高的绝对性能，在 32 个核心的情况下，平均速度提高了 30 多倍。


## 1 引言

如今，硬件性能提升的主要动力来自于多核并行性的提高，而不是单线程性能的加速[2]。到 SIGMOD 2014，英特尔即将推出的主流服务器 Ivy Bridge EX，可以运行 120 个并发线程。我们用多核这个词来形容这种具有几十或几百个核的架构。

同时，每台服务器的主内存容量增加到几 TB，导致了主内存数据库系统的发展。在这些系统中，查询处理不再受 I/O 限制，多核的巨大并行计算资源可以得到真正的利用。不幸的是，将内存控制器移到芯片中的趋势，以及因此而导致的内存访问的分散化--这是扩展巨大内存吞吐量所需要的--导致了非均匀内存访问（NUMA）。从本质上讲，计算机本身已经成为一个网络，因为数据项的访问成本根据数据和访问线程所在的芯片而不同。因此，多核并行化需要考虑到 RAM 和高速缓存的层次结构。特别是，必须仔细考虑 RAM 的 NUMA 划分，以确保线程在 NUMA 本地的数据上工作（大部分）。

20 世纪 90 年代对并行处理的大量研究导致大多数数据库系统采用了火山[12]模型中的一种并行形式，其中操作者基本上不知道并行。并行性被所谓的 ″交换 ″操作符所封装，这些操作符在多个线程之间路由元组流，每个线程执行查询计划中相同的流水线段。Volcano 模型的这种实现可以称为计划驱动：优化器在查询编译时静态地确定应该运行多少个线程，为每个线程实例化一个查询操作计划，并将其与交换操作符连接起来。

在本文中，我们提出了自适应的小鼠驱动的查询执行框架，这是我们为我们的主内存数据库系统 HyPer[16]设计的。我们的方法在图 1 中对三向连接查询 R IA S IB T 进行了描述。并行化是通过在不同的核心上并行处理每个管道来实现的，如图中的两个（上/红和下/蓝）管道所示。其核心思想是一种调度机制（″调度器″），允许灵活地并行执行运算器管道，甚至在查询执行过程中也能改变并行程度。一个查询被划分为若干段，每个执行段取一个输入图元（如 100,000）并执行这些图元，在下一个管道断路器中对结果进行匹配。如图中的颜色编码所示，Morsel 框架实现了 NUMA 本地处理： 一个线程对 NUMA 本地输入进行操作，并将其结果写入 NUMA 本地的存储区域中。我们的调度器运行固定的、与机器相关的线程数量，这样即使有新的查询到来，也不会出现资源超额占用的情况，而且这些线程被钉在核心上，这样就不会因为操作系统将线程移到不同的核心上而发生 NUMA 本地性的意外损失。

蚕食驱动调度的关键特征是，任务分配是在运行时完成的，因此是完全弹性的。这允许实现完美的负载平衡，即使面对不确定的中间结果的大小分布，以及难以预测的现代 CPU 内核的性能，即使它们得到的工作量是相同的，也会发生变化。它是有弹性的，因为它可以处理在运行时发生变化的工作负载（通过减少或增加已经在飞行中执行的查询的并行性），并且可以很容易地整合一个机制，以不同的优先级运行查询。

蛀虫驱动的想法从单纯的调度延伸到一个完整的查询执行框架，即所有的物理查询操作者必须能够在其所有的执行阶段（例如，哈希构建和探测）以蛀虫方式并行执行，根据阿姆达尔定律，这是实现多核可伸缩性的关键需求。morsel-wise 框架的一个重要部分是对数据位置性的认识。这从输入莫西尔和莫西尔化输出缓冲区的位置开始，但延伸到操作者可能创建和访问的状态（数据结构，如哈希表）。这种状态是共享数据，有可能被任何内核访问，但确实有高度的 NUMA 定位。因此，小块的调度是灵活的，但强烈倾向于最大化 NUMA 本地执行的调度选择。这意味着，只有在处理每个查询的几个摩西时，才会发生重新的 NUMA 访问，以实现负载平衡。通过主要访问本地 RAM，内存延迟得到了优化，跨插槽的内存流量也降到了最低，这可能会拖累其他线程。

在一个纯粹的基于 Volcano 的并行框架中，并行性被隐藏在运算符中，共享状态被避免，这导致计划在交换运算符中进行即时数据分割。我们认为，这并不总是导致最佳的计划（因为分割的努力并不总是有回报的），而通过即时分割实现的局部性可以由我们的局部感知的调度器来实现。其他的系统提倡每个操作者

的并行化[21]，以实现执行的灵活性，但这导致在一个管道段的操作者之间需要较少的同步。尽管如此，我们相信 morsel-wise 框架可以被整合到许多现有的系统中，例如，通过改变交换操作符的实现来封装 morsel-wise 调度，并引入例如哈希表共享。我们的框架也适用于使用及时（JIT）代码编译的系统[19, 25]，因为计划中出现的每个流水线的生成代码，随后可以进行分食式调度。事实上，我们的 HyPer 系统使用了这种 JIT 方法[25]。

在本文中，我们提出了一些相关的想法，以实现高效、可扩展和弹性的并行处理。主要的贡献是一个包含以下内容的查询引擎的架构蓝图：

- Morsel 驱动的查询执行是一个新的并行查询评估框架，它与传统的 Volcano 模型有很大的不同，因为它使用工作窃取的方式在线程之间动态地分配工作。这可以防止由于负载不平衡而导致的 CPU 资源未被使用，并允许弹性，即 CPU 资源可以在任何时候在不同的查询之间重新分配。

- 一套针对最重要的重理性化运算符的快速并行算法。

- 将 NUMA 意识整合到数据库系统中的系统性方案。

本文的其余部分组织如下。第 2 节专门详细讨论了管道并行化和数据碎片化的问题。在第 3 节中，我们讨论了调度器，它将任务（流水线作业）和小块（数据片段）分配给工作线程。调度器实现了完全的弹性，允许在任何时候改变工作在特定查询上的并行线程的数量。第 4 节讨论了并行连接、聚合和排序操作者的算法和同步细节。第 5 节通过整个 TPC-H 查询套件来评估该查询引擎的优点。在讨论了相关工作以指出我们的并行查询引擎架构的新颖性之后，我们在第 6 节中总结了本文。

## 2 小数据块驱动执行

改编自介绍中的激励性查询，我们将在以下查询计划的例子中展示我们的并行管道查询执行：

σ...(R) IA σ...(S) IB σ...(T)

假设 R 是最大的表（过滤后），优化器将选择 R 作为探测输入，并建立（团队）其他两个 S 和 T 的哈希表。由此产生的代数查询计划（由基于成本的优化器获得）由图 2 左侧所示的三个管道组成：

1. 扫描、过滤和建立基础关系 T 的哈希表 HT(T)、

2. 2. 扫描、过滤和建立论据 S 的哈希表 HT(S)、

3. 扫描、过滤 R 和探测 S 的哈希表 HT(S)，探测 T 的哈希表 HT(T)并存储结果图元。

HyPer 使用即时编译（JIT）来生成高效的机器代码。每个流水线段，包括所有操作器，都被编译成一个代码片段。这实现了非常高的原始性能，因为传统查询评估器的解释开销被消除了。此外，管道中的操作者甚至没有实现他们的中间结果，这仍然是由 Vectorwise[34]的已经非常高效的矢量一次评估引擎完成的。

代数计划中的小数据块驱动的执行是由一个所谓的 QEPobject 控制的，它将可执行的管道传输给一个调度器--参见第 3 节。QEPobject 的责任是观察数据的

依赖性。在我们的查询例子中，第三条（探测）管道只能在两个哈希表建立后执行，也就是在前两条管道完全执行后。对于每个管道，QEPobject 都会分配临时存储区域，执行该管道的并行线程会将其结果写入其中。在整个流水线完成后，临时存储区域被逻辑地重新分割成同等大小的小块；这样一来，后续的流水线就从新的相同大小的小块开始，而不是在流水线之间保留小块的边界，这很容易造成小块大小的偏差。任何时候在任何管道上工作的并行线程的数量都受处理器的硬件线程数量的限制。为了在本地写入 NUMA，并在写入中间结果时避免同步，QEPobject 为每个可执行管道的每个这样的线程/核分配了一个存储区域。

图 3 显示了过滤 T 和建立哈希表 H T（T）的流水线的并行处理。让我们专注于管道的第一阶段的处理，即过滤输入的 T 并将 ″幸存的 ″图元存储在临时存储区。

在我们的图中，显示了三个并行线程，每个线程一次在一个莫西干上操作。由于我们的基础关系 T 是在 NUMA 组织的内存中 ″以摩西为单位 ″存储的，调度器尽可能地分配一个位于线程执行的同一套接字上的摩西。这在图中用颜色表示：在红色套接字的核心上运行的红色线程被分配到处理一个红色的碎片，也就是位于红色套接字上的基础关系 T 的一个小片段的任务。一旦，线程完成了对所分配的摩西的处理，它可以被委托（派发）到一个不同的任务，或者它获得另一个摩西（相同的颜色）作为其下一个任务。由于线程一次处理一个果子，系统是完全弹性的。在处理查询的过程中，可以在任何时候（更确切地说，在果子的边界）减少或增加并行度（MPL）。

(1)扫描/过滤输入 T 和(2)建立哈希表的逻辑代数管道实际上被分解成两个物理处理管道，在图的左侧标记为阶段。在第一阶段，过滤后的图元被插入到 NUMA 本地存储区，也就是说，为了避免同步，每个内核都有一个独立的存储区。为了在进一步的处理阶段保持 NUMA-locality，一个特定内核的存储区域被本地分配在同一个套接字上。

在所有的基表碎片被扫描和过滤后，在第二阶段，这些存储区域被扫描--同样是由循环在相应内核上的线程进行扫描--并将指针插入哈希表中。将逻辑哈希表的构建管道分为两个阶段，可以完美地确定全局哈希表的大小，因为在第一阶段完成后，″存活 ″对象的确切数量是已知的。这个（完美大小的）全局哈希表将被位于 NUMA 系统不同套接字上的线程探测；因此，为了避免竞争，它不应该驻留在一个特定的 NUMA 区域，因此被交错（分散）在所有套接字上。由于许多并行线程争相向该哈希表插入数据，因此无锁实现是必不可少的。哈希表的实现细节将在第 4.2 节描述。

在这两个哈希表构建完毕后，可以安排探测管道。探测管道的详细处理过程如图 4 所示。同样，一个线程向打补丁器请求工作，打补丁器在相应的 NUMA 分区中分配一个摩尔。也就是说，一个位于红色 NUMA 分区的核心上的线程被分配了一个位于核心上的基础关系 R 的摩西。

响应的 ″红色 ″NUMA 套接字。探究管道的结果再次存储在 NUMA 本地存储区

域，以便为进一步的处理保留 NUMA 定位（在我们的样本查询计划中没有出现）。

总的来说，Morsel 驱动的并行性是并行地执行多个管道，这与 Vol- cano 模型的典型实现相似。然而，与 Volcano 不同的是，这些管道并不独立。也就是说，它们共享数据结构，操作者意识到平行执行，必须执行同步（通过有效的无锁机制--见下文）。另一个区别是，执行计划的线程数量是完全弹性的。也就是说，如图 2 所示，线程数量不仅在不同的流水线段之间可能不同，而且在查询执行过程中，在同一流水线段内也可能不同--如下面所述。

## 3 调度器——并行 Pipeline 任务调度

调度器是控制和分配计算再源到并行管道的。这是通过将任务分配给工人线程来完成的。我们为机器提供的每个硬设备线程（pre-）创建一个工人线程，并将每个工人线程永久地绑定到它。因此，一个特定查询的并行程度不是通过创建或终止线程来控制的，而是通过给它们签署可能不同的查询的特定任务来控制的。分配给这样一个工人线程的任务包括一个管道作业和一个特定的磨盘，该管道必须在其上执行。任务的抢占发生在果壳的边界，从而消除了潜在的昂贵的中断机制。我们通过实验确定，在即时弹性调整、负载平衡和低维护开销之间，约有 100,000 个图元的摩西尔大小可以产生良好的权衡。

将任务分配给在特定核心上运行的线程有三个主要目标：

1. 通过将数据元组分配给分配元组的内核来保持(NUMA-)位置性
2. 关于特定查询的并行性水平的充分弹性
3. 负载平衡要求所有参与查询管道的核心在同一时间完成它们的工作，以防止（快）核等待其他（慢）核 1。

图 5 是调度器的结构简图。它维护一个待处理的管道作业的列表。这个列表只包含前提条件已经被处理的管道作业。例如，对于我们正在运行的示例查询，构建输入管道首先被插入到待处理作业列表中。只有在这两个构建管道完成后，才会插入探测管道。如前所述，每个活动查询都由一个 QEPob- ject 控制，该 QEPob- ject 负责将可执行管线传输给调度器。因此，调度器只维护所有从属管道已经被处理的管道工作的列表。一般来说，调度器队列将包含不同查询的待决管道作业，这些作业是平行执行的，以适应查询间的并行性。

### 3.1 弹性

通过 ″一次一粒 ″的工作调度实现的完全弹性并行，允许根据服务质量模型对这些查询间的并行管道工作进行智能调度。它可以优雅地降低长期运行的查询 Q1 在任何处理阶段的并行程度，以便优先处理可能更重要的交互查询 Q+。一旦优先级较高的查询 Q+完成，钟摆就会摆回到长期运行的查询上，将所有或大多数核心分配给长期运行的查询 Q1 的任务。在第 5.4 节中，我们通过实验来证明这种动态弹性。在我们目前的实现中，所有的查询都有相同的优先级，所以线程被平均分配到所有活动的查询上。基于优先级的调度组件正在开发中，但超出了本文的范围。

对于每一个流水线作业，调度器都会维护该流水线作业仍需执行的待处理小块的列表。对于每个核来说，都有一个单独的列表，以确保例如核 0 的工作请求返回一个与核 0 分配在同一套接字上的摩西。一旦核心 0 处理完所分配的食物，它就会请求一个新的任务，这个任务可能来自同一个管道工作，也可能不是。这取决于不同管道作业的优先级，这些作业来自于正在执行的不同查询。如果一个高优先级的查询进入系统，可能会导致当前查询的并行度降低。碎片化处理允许重新将核心分配给不同的流水线工作，而不需要任何激烈的中断机制。

## 3.2 实现综述

为了说明问题，我们在图 5 中为每个内核展示了一个（长的）链式列表。在现实中（即在我们的实现中），我们为每个核/套接字维护存储区域边界，并根据需要将这些大的存储区域分割成小块；也就是说，当一个核向调度器请求一个任务时，特定套接字上的管道参数存储区域的下一个小块被 ″切出来″。此外，在图 5 中，Dispatcher 看起来像是一个独立的线程。然而，这将招致两个缺点：（1）调度器本身需要一个核心来运行，或者可能预留查询评估线程；（2）它可能成为争论的来源，特别是如果小数据块量被配置得相当小。因此，调度器仅作为一个无锁数据结构来实现。然后，调度器的代码由请求工作的查询评估线程本身执行。因此，调度器被自动地在这个工作线程的（否则未使用的）核心上执行。依靠无锁数据结构（即流水线作业队列以及相关的小数据块队列），即使多个查询评估线程同时请求新的任务，也能减少争论。类似地，通过观察数据依赖关系（例如，在执行探测管道之前建立哈希表）来触发特定查询进展的 QEPobject 被实现为一个无源状态机。每当一个管道工作被完全执行时，该代码就会被调度员调用，因为在工作请求时无法找到新的小数据块。同样，这个状态机是在最初向调度器请求新任务的工人线程的其他未使用的核心上执行。

除了能够在任何时候将一个核心分配给不同的查询--称为弹性--外，″小数据块″式处理还保证了负载平衡和抗偏斜。所有在同一流水线上工作的线程都以 ″拍照完成 ″的方式运行：它们被保证在处理一个小数据块的时间段内到达终点。如果由于某种原因，一个核心在其特定的套接字上完成了所有的小数据块，调度器将从另一个核心 ″偷工″，也就是说，它将把小数据块分配到不同的套接字。在一些 NUMA 系统中，并不是所有的套接字都是相互直接连接的；在这种情况下，先从较近的套接字上偷工是值得的。在正常情况下，从远程套接字中窃取工作的情况很少发生；然而，有必要避免空闲线程。而且，无论如何，写入临时存储区的工作将在 NUMA 本地存储区完成（也就是说，如果在从红色套接字上的核心窃取工作的过程中，一个红色的小数据块被一个蓝色的核心处理过，那么它就会变成蓝色）。

到目前为止，我们已经讨论了管线内的并行性。我们的并行化方案也可以支持总线并行，例如，我们的例子中 ″过滤和建立 T 的哈希表 ″和 ″过滤和建立 S 的哈希表 ″这两条管道是独立的，因此可以并行执行。然而，这种形式的并行性的作用是有限的。独立管线的数量通常比内核的数量小得多，而且每个管线的工

作量通常是不同的。此外，丛式并行会因为减少缓存定位而降低性能。因此，我们目前避免从一个查询中并行执行多个管道；在我们的例子中，我们首先执行管道 T，只有在 T 完成后，管道 S 的工作才被添加到管道工作列表中。

除了弹性，小数据块驱动处理还允许简单而优雅地实现查询的取消。一个用户可能放弃了她的查询请求，查询中发生了异常（例如，数字溢出），或者系统的 RAM 耗尽。如果这些事件中的任何一个发生了，涉及的查询就会在调度器中被标记。每当该查询的一个片段完成时，就会检查该标记，因此，很快所有的工作线程就会停止在该查询上工作。与强迫操作系统杀死线程相比，这种方法允许每个线程进行清理（例如，释放分配的内存）。

## 3.3 数据块大小

与 Vectorwise[9]和 IBM 的 BLU[31]等系统相比，这些系统使用向量/步长在操作者之间传递数据，如果一个小数据块不适合进入缓存，则不会有性能上的损失。Morsels 被用来将一个大的任务分解成小的、大小不变的工作单元，以方便工作的窃取和抢占。因此，小数据块的大小对性能不是很关键，它只需要大到足以摊销调度开销，同时提供良好的再响应时间。为了显示食物大小对查询性能的影响，我们测量了在 Nehalem EX 系统上使用 64 个线程从 R 中选择 min(a)的查询的性能，这在第 5 节有描述。这个查询非常简单，所以它尽可能地强调了偷工减料的数据结构。图 6 显示，在开销可以忽略不计的情况下，小数据块大小应该被设置为最小的可行值，在这种情况下，应该设置为高于 10,000 的值。最佳设置取决于硬件，但可以很容易地通过实验来确定。

在多核系统中，任何共享数据结构，即使是无锁的，最终也会成为一个瓶颈。然而，在我们的偷工减料数据结构的情况下，有许多方面可以防止它成为一个可扩展性问题。首先，在我们的实现中，总的工作最初在所有线程之间分配，这样，每个线程暂时拥有一个本地范围。因为我们对每个范围进行了缓存线的调整，所以在缓存线层面上的冲突是不可能的。只有当这个本地范围用尽时，一个线程才会试图从另一个范围偷取工作。其次，如果有多个查询被同时执行，数据结构的压力就会进一步减少。最后，总是有可能增加小数据块的大小。这将导致对偷工减料数据结构的访问减少。在最坏的情况下，过大的小数据块大小会导致线程利用不足，但如果有足够的并发查询正在执行，则不会影响系统的吞吐量。

# 附件 2 原文

# Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age


## ABSTRACT

With modern computer architecture evolving, two problems con- spire against the state-of-the-art approaches in parallel query exe- cution: (i) to take advantage of many-cores, all query work must be distributed evenly among (soon) hundreds of threads in order to achieve good speedup, yet (ii) dividing the work evenly is difficult even with accurate data statistics due to the complexity of modern out-of-order cores. As a result, the existing approaches for “plan- driven” parallelism run into load balancing and context-switching bottlenecks, and therefore no longer scale. A third problem faced by many-core architectures is the decentralization of memory con- trollers, which leads to Non-Uniform Memory Access (NUMA).

In response, we present the “morsel-driven” query execution framework, where scheduling becomes a fine-grained run-time task that is NUMA-aware. Morsel-driven query processing takes small fragments of input data (“morsels”) and schedules these to worker threads that run entire operator pipelines until the next pipeline breaker. The degree of parallelism is not baked into the plan but can elastically change during query execution, so the dispatcher can re- act to execution speed of different morsels but also adjust resources dynamically in response to newly arriving queries in the workload. Further, the dispatcher is aware of data locality of the NUMA-local morsels and operator state, such that the great majority of execu- tions takes place on NUMA-local memory. Our evaluation on the TPC-H and SSB benchmarks shows extremely high absolute per- formance and an average speedup of over 30 with 32 cores.


## 1. INTRODUCTION

The main impetus of hardware performance improvement nowa- days comes from increasing multi-core parallelism rather than from speeding up single-threaded performance [2]. By SIGMOD 2014 Intel’s forthcoming mainstream server Ivy Bridge EX, which can run 120 concurrent threads, will be available. We use the term many-core for such architectures with tens or hundreds of cores.

At the same time, increasing main memory capacities of up to several TB per server have led to the development of main-memory database systems. In these systems query processing is no longer I/O bound, and the huge parallel compute resources of many-cores can be truly exploited. Unfortunately, the trend to move memory controllers into the chip and hence the decentralization of mem- ory access, which was needed to scale throughput to huge mem- ories, leads to non-uniform memory access (NUMA). In essence, the computer has become a network in itself as the access costs of data items varies depending on which chip the data and the access- ing thread are located. Therefore, many-core parallelization needs to take RAM and cache hierarchies into account. In particular, the NUMA division of the RAM has to be considered carefully to en- sure that threads work (mostly) on NUMA-local data.

Abundant research in the 1990s into parallel processing led the majority of database systems to adopt a form of parallelism in- spired by the Volcano [12] model, where operators are kept largely unaware of parallelism. Parallelism is encapsulated by so-called “exchange” operators that route tuple streams between multiple threads each executing identical pipelined segments of the query plan. Such implementations of the Volcano model can be called plan-driven: the optimizer statically determines at query compile- time how many threads should run, instantiates one query operator plan for each thread, and connects these with exchange operators.

In this paper we present the adaptive morsel-driven query execu- tion framework, which we designed for our main-memory database system HyPer [16]. Our approach is sketched in Figure 1 for the three-way-join query R IA S IB T. Parallelism is achieved by processing each pipeline on different cores in parallel, as indi- cated by the two (upper/red and lower/blue) pipelines in the fig- ure. The core idea is a scheduling mechanism (the “dispatcher”) that allows flexible parallel execution of an operator pipeline, that can change the parallelism degree even during query execution. A query is divided into segments, and each executing segment takes a morsel (e.g, 100,000) of input tuples and executes these, mate- rializing results in the next pipeline breaker. The morsel frame- work enables NUMA local processing as indicated by the color coding in the figure: A thread operates on NUMA-local input and writes its result into a NUMA-local storage area. Our dispatcher runs a fixed, machine-dependent number of threads, such that even if new queries arrive there is no resource over-subscription, and these threads are pinned to the cores, such that no unexpected loss of NUMA locality can occur due to the OS moving a thread to a different core.

The crucial feature of morsel-driven scheduling is that task dis- tribution is done at run-time and is thus fully elastic. This allows to achieve perfect load balancing, even in the face of uncertain size distributions of intermediate results, as well as the hard-to-predict performance of modern CPU cores that varies even if the amount of work they get is the same. It is elastic in the sense that it can handle workloads that change at run-time (by reducing or increas- ing the parallelism of already executing queries in-flight) and can easily integrate a mechanism to run queries at different priorities.

The morsel-driven idea extends from just scheduling into a com- plete query execution framework in that all physical query opera- tors must be able to execute morsel-wise in parallel in all their ex- ecution stages (e.g., both hash-build and probe), a crucial need for achieving many-core scalability in the light of Amdahl’ s law. An important part of the morsel-wise framework is awareness of data locality. This starts from the locality of the input morsels and ma- terialized output buffers, but extends to the state (data structures, such as hash tables) possibly created and accessed by the opera- tors. This state is shared data that can potentially be accessed by any core, but does have a high degree of NUMA locality. Thus morsel-wise scheduling is flexible, but strongly favors scheduling choices that maximize NUMA-local execution. This means that re- mote NUMA access only happens when processing a few morsels per query, in order to achieve load balance. By accessing local RAM mainly, memory latency is optimized and cross-socket mem- ory traffic, which can slow other threads down, is minimized.

In a pure Volcano-based parallel framework, parallelism is hid- den from operators and shared state is avoided, which leads to plans doing on-the-fly data partitioning in the exchange operators. We argue that this does not always lead to the optimal plan (as parti- tioning effort does not always pay off), while the locality achieved by on-the-fly partitioning can be achieved by our locality-aware dispatcher. Other systems have advocated per-operator paralleliza- tion [21] to achieve flexibility in execution, but this leads to need- less synchronization between operators in one pipeline segment. Nevertheless, we are convinced that the morsel-wise framework can be integrated in many existing systems, e.g., by changing the implementation of exchange operators to encapsulate morsel-wise scheduling, and introduce e.g., hash-table sharing. Our framework also fits systems using Just-In-Time (JIT) code compilation [19, 25] as the generated code for each pipeline occurring in the plan, can subsequently be scheduled morsel-wise. In fact, our HyPer system uses this JIT approach [25].

In this paper we present a number of related ideas that enable efficient, scalable, and elastic parallel processing. The main contri- bution is an architectural blueprint for a query engine incorporating the following:

- Morsel-driven query execution is a new parallel query eval- uation framework that fundamentally differs from the tra- ditional Volcano model in that it distributes work between threads dynamically using work-stealing. This prevents un- used CPU resources due to load imbalances, and allows for elasticity, i.e., CPU resources can be reassigned between dif- ferent queries at any time.
- A set of fast parallel algorithms for the most important re- lational operators.
- AsystematicapproachtointegratingNUMA-awarenessinto database systems.

The remainder of this paper is organized as follows. Section 2 is devoted to a detailed discussion of pipeline parallelization and the fragmentation of the data into morsels. In Section 3 we discuss the dispatcher, which assigns tasks (pipeline jobs) and morsels (data fragments) to the worker threads. The dispatcher enables the full elasticity which allows to vary the number of parallel threads work- ing on a particular query at any time. Section 4 discusses algorith- mic and synchronization

details of the parallel join, aggregation, and sort operators. The virtues of the query engine are evaluated in Section 5 by way of the entire TPC-H query suite. After discussing related work in order to point out the novelty of our parallel query engine architecture in Section 6, we conclude the paper.

## 2. MORSEL-DRIVEN EXECUTION

Adapted from the motivating query of the introduction, we will demonstrate our parallel pipeline query execution on the following example query plan:

σ...(R) IA σ...(S) IB σ...(T)

Assuming that R is the largest table (after filtering) the optimizer would choose R as probe input and build (team) hash tables of the other two, S and T . The resulting algebraic query plan (as obtained by a cost-based optimizer) consists of the three pipelines illustrated on the left-hand side of Figure 2:

1. Scanning, filtering and building the hash table HT(T) of base relation T ,

2. Scanning, filtering and building the hash table HT(S) of ar- gument S,

3. Scanning, filtering R and probing the hash table HT(S) of S and probing the hash table HT(T) of T and storing the result tuples.

HyPer uses Just-In-Time (JIT) compilation to generate highly efficient machine code. Each pipeline segment, including all oper- ators, is compiled into one code fragment. This achieves very high raw performance, since interpretation overhead as experienced by traditional query evaluators, is eliminated. Further, the operators in the pipelines do not even materialize their intermediate results, which is still done by the already much more efficient vector-at-a- time evaluation engine of Vectorwise [34].

The morsel-driven execution of the algebraic plan is controlled by a so called QEPobject which transfers executable pipelines to a dispatcher – cf. Section 3. It is the QEPobject’s responsibility to observe data dependencies. In our example query, the third (probe) pipeline can only be executed after the two hash tables have been built, i.e., after the first two pipelines have been fully executed. For each pipeline the QEPobject allocates the temporary storage areas into which the parallel threads executing the pipeline write their results. After completion of the entire pipeline the tempo- rary storage areas are logically re-fragmented into equally sized morsels; this way the succeeding pipelines start with new homoge- neously sized morsels instead of retaining morsel boundaries across pipelines which could easily result in skewed morsel sizes. The number of parallel threads working on any pipeline at any time is bounded by the number of hardware threads of the processor. In order to write NUMA-locally and to avoid synchronization while writing intermediate results the QEPobject allocates a storage area for each such thread/core for each executable pipeline.

The parallel processing of the pipeline for filtering T and build- ing the hash table H T (T ) is shown in Figure 3. Let us concentrate on the processing of the first phase of the pipeline that filters in- put T and stores the “surviving” tuples in temporary storage areas.

In our figure three parallel threads are shown, each of which op- erates on one morsel at a time. As our base relation T is stored “morsel-wise” across a

NUMA-organized memory, the scheduler assigns, whenever possible, a morsel located on the same socket where the thread is executed. This is indicated by the coloring in the figure: The red thread that runs on a core of the red socket is assigned the task to process a red-colored morsel, i.e., a small frag- ment of the base relation T that is located on the red socket. Once, the thread has finished processing the assigned morsel it can either be delegated (dispatched) to a different task or it obtains another morsel (of the same color) as its next task. As the threads pro- cess one morsel at a time the system is fully elastic. The degree of parallelism (MPL) can be reduced or increased at any point (more precisely, at morsel boundaries) while processing a query.

The logical algebraic pipeline of (1) scanning/filtering the input T and (2) building the hash table is actually broken up into two physical processing pipelines marked as phases on the left-hand side of the figure. In the first phase the filtered tuples are inserted into NUMA-local storage areas, i.e., for each core there is a sep- arate storage area in order to avoid synchronization. To preserve NUMA-locality in further processing stages, the storage area of a particular core is locally allocated on the same socket.

After all base table morsels have been scanned and filtered, in the second phase these storage areas are scanned – again by threads lo- cated on the corresponding cores – and pointers are inserted into the hash table. Segmenting the logical hash table building pipeline into two phases enables perfect sizing of the global hash table be- cause after the first phase is complete, the exact number of “surviv- ing” objects is known. This (perfectly sized) global hash table will be probed by threads located on various sockets of a NUMA sys- tem; thus, to avoid contention, it should not reside in a particular NUMA-area and is therefore is interleaved (spread) across all sock- ets. As many parallel threads compete to insert data into this hash table, a lock-free implementation is essential. The implementation details of the hash table are described in Section 4.2.

After both hash tables have been constructed, the probing pipeline can be scheduled. The detailed processing of the probe pipeline is shown in Figure 4. Again, a thread requests work from the dis- patcher which assigns a morsel in the corresponding NUMA parti- tion. That is, a thread located on a core in the red NUMA partition is assigned a morsel of the base relation R that is located on the cor

responding “red” NUMA socket. The result of the probe pipeline is again stored in NUMA local storage areas in order to preserve NUMA locality for further processing (not present in our sample query plan).

In all, morsel-driven parallelism executes multiple pipelines in parallel, which is similar to typical implementations of the Vol- cano model. Different from Volcano, however, is the fact that the pipelines are not independent. That is, they share data struc- tures and the operators are aware of parallel execution and must perform synchronization (through efficient lock-free mechanisms – see later). A further difference is that the number of threads exe- cuting the plan is fully elastic. That is, the number may differ not only between different pipeline segments, as shown in Figure 2, but also inside the same pipeline segment during query execution – as

described in the following.

# 3. DISPATCHER: SCHEDULING PARALLEL PIPELINE TASKS

The dispatcher is controlling and assigning the compute re- sources to the parallel pipelines. This is done by assigning tasks to worker threads. We (pre-)create one worker thread for each hard- ware thread that the machine provides and permanently bind each worker to it. Thus, the level of parallelism of a particular query is not controlled by creating or terminating threads, but rather by as- signing them particular tasks of possibly different queries. A task that is assigned to such a worker thread consists of a pipeline job and a particular morsel on which the pipeline has to be executed. Preemption of a task occurs at morsel boundaries – thereby elimi- nating potentially costly interrupt mechanisms. We experimentally determined that a morsel size of about 100,000 tuples yields good tradeoff between instant elasticity adjustment, load balancing and low maintenance overhead.

There are three main goals for assigning tasks to threads that run on particular cores:

1. Preserving (NUMA-)locality by assigning data morsels to cores on which the morsels are allocated

2. Full elasticity concerning the level of parallelism of a partic- ular query

3. Loadbalancingrequiresthatallcoresparticipatinginaquery pipeline finish their work at the same time in order to prevent (fast) cores from waiting for other (slow) cores1.

In Figure 5 the architecture of the dispatcher is sketched. It maintains a list of pending pipeline jobs. This list only contains pipeline jobs whose prerequisites have already been processed. E.g., for our running example query the build input pipelines are first inserted into the list of pending jobs. The probe pipeline is only inserted after these two build pipelines have been finished. As de- scribed before, each of the active queries is controlled by a QEPob- ject which is responsible for transferring executable pipelines to the dispatcher. Thus, the dispatcher maintains only lists of pipeline jobs for which all dependent pipelines were already processed. In general, the dispatcher queue will contain pending pipeline jobs of different queries that are executed in parallel to accommodate inter-query parallelism.

## 3.1 Elasticity

The fully elastic parallelism, which is achieved by dispatching jobs “a morsel at a time”, allows for intelligent scheduling of these inter-query parallel pipeline jobs depending on a quality of service model. It enables to gracefully decrease the degree of parallelism of, say a long-running query Ql at any stage of processing in order to prioritize a possibly more important interactive query Q+. Once the higher prioritized query Q+ is finished, the pendulum swings back to the long running query by dispatching all or most cores to tasks of the long running query Ql. In Section 5.4 we demonstrate this dynamic elasticity experimentally. In our current implemen- tation all queries have the same priority, so threads are distributed equally over all active queries. A priority-based scheduling com- ponent is under development but beyond

the scope of this paper.

For each pipeline job the dispatcher maintains lists of pending morsels on which the pipeline job has still to be executed. For each core a separate list exists to ensure that a work request of, say, Core 0 returns a morsel that is allocated on the same socket as Core 0. This is indicated by different colors in our architectural sketch. As soon as Core 0 finishes processing the assigned morsel, it requests a new task, which may or may not stem from the same pipeline job. This depends on the prioritization of the different pipeline jobs that originate from different queries being executed. If a high-priority query enters the system it may lead to a decreased parallelism de- gree for the current query. Morsel-wise processing allows to re- assign cores to different pipeline jobs without any drastic interrupt mechanism.

## 3.2 Implementation Overview

For illustration purposes we showed a (long) linked list of morsels for each core in Figure 5. In reality (i.e., in our implementation) we maintain storage area boundaries for each core/socket and segment these large storage areas into morsels on demand; that is, when a core requests a task from the dispatcher the next morsel of the pipeline argument’s storage area on the particular socket is “cut out”. Furthermore, in Figure 5 the Dispatcher appears like a sep- arate thread. This, however, would incur two disadvantages: (1) the dispatcher itself would need a core to run on or might pre- empt query evaluation threads and (2) it could become a source of contention, in particular if the morsel size was configured quite small. Therefore, the dispatcher is implemented as a lock-free data structure only. The dispatcher’s code is then executed by the work- requesting query evaluation thread itself. Thus, the dispatcher is au- tomatically executed on the (otherwise unused) core of this worker thread. Relying on lock-free data structures (i.e., the pipeline job queue as well as the associated morsel queues) reduces contention even if multiple query evaluation threads request new tasks at the same time. Analogously, the QEPobject that triggers the progress of a particular query by observing data dependencies (e.g., build- ing hash tables before executing the probe pipeline) is implemented as a passive state machine. The code is invoked by the dispatcher whenever a pipeline job is fully executed as observed by not being able to find a new morsel upon a work request. Again, this state machine is executed on the otherwise unused core of the worker thread that originally requested a new task from the dispatcher.

Besides the ability to assign a core to a different query at any time – called elasticity – the morsel-wise processing also guaran- tees load balancing and skew resistance. All threads working on the same pipeline job run to completion in a “photo finish”: they are guaranteed to reach the finish line within the time period it takes to process a single morsel. If, for some reason, a core finishes processing all morsels on its particular socket, the dispatcher will “steal work” from another core, i.e., it will assign morsels on a dif- ferent socket. On some NUMA systems, not all sockets are directly connected with each other; here it pays off to steal from closer sock- ets first. Under normal circumstances, work-stealing from remote sockets happens very infrequently; nevertheless it is necessary to avoid idle threads. And the

writing into temporary storage will be done into NUMA local storage areas anyway (that is, a red morsel turns blue if it was processed by a blue core in the process of steal- ing work from the core(s) on the red socket).

So far, we have discussed intra-pipeline parallelism. Our par- allelization scheme can also support bushy parallelism, e.g., the pipelines “filtering and building the hash table of T ” and “filtering and building the hash table of S” of our example are independent and could therefore be executed in parallel. However, the useful- ness of this form of parallelism is limited. The number of indepen- dent pipelines is usually much smaller than the number of cores, and the amount of work in each pipeline generally differs. Fur- thermore, bushy parallelism can decrease performance by reducing cache locality. Therefore, we currently avoid to execute multiple pipelines from one query in parallel; in our example, we first exe- cute pipeline T , and only after T is finished, the job for pipeline S is added to the list of pipeline jobs.

Besides elasticity, morsel-driven processing also allows for a simple and elegant implementation of query canceling. A user may have aborted her query request, an exception happened in a query (e.g., a numeric overflow), or the system is running out of RAM. If any of these events happen, the involved query is marked in the dispatcher. The marker is checked whenever a morsel of that query is finished, therefore, very soon all worker threads will stop work- ing on this query. In contrast to forcing the operating system to kill threads, this approach allows each thread to clean up (e.g., free allocated memory).

## 3.3 Morsel Size

In contrast to systems like Vectorwise [9] and IBM’ s BLU [31], which use vectors/strides to pass data between operators, there is no performance penalty if a morsel does not fit into cache. Morsels are used to break a large task into small, constant-sized work units to facilitate work-stealing and preemption. Consequently, the morsel size is not very critical for performance, it only needs to be large enough to amortize scheduling overhead while providing good re- sponse times. To show the effect of morsel size on query per- formance we measured the performance for the query select min(a) from R using 64 threads on a Nehalem EX system, which is described in Section 5. This query is very simple, so it stresses the work-stealing data structure as much as possible. Fig- ure 6 shows that the morsel size should be set to the smallest pos- sible value where the overhead is negligible, in this case to a value above 10,000. The optimal setting depends on the hardware, but can easily be determined experimentally.

On many-core systems, any shared data structure, even if lock- free, can eventually become a bottleneck. In the case of our work- stealing data structure, however, there are a number of aspects that prevent it from becoming a scalability problem. First, in our imple- mentation the total work is initially split between all threads, such that each thread temporarily owns a local range. Because we cache line align each range, conflicts at the cache line level are unlikely. Only when this local range is exhausted, a thread tries to steal work from another range. Second, if more than one query is executed concurrently, the pressure on the data structure is further

reduced. Finally, it is always possible to increase the morsel size. This re- sults in fewer accesses to the work-stealing data structure. In the worst case a too large morsel size results in underutilized threads but does not affect throughput of the system if enough concurrent queries are being executed.