软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(3):1225−1239 [doi: 10.13328/j.cnki.jos.007467] [CSTR: 32375.14.jos.007467]

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

JIT 编译技术在可插拔存储引擎数据库中的应用*

袁巩生[1,2], 杜晨路[1], 朱阅岸[3], 陈育兴[4], 唐 秀[1,2], 姚 畅[1,2], 陈 刚[5]

[1](浙江大学 软件学院, 浙江 宁波 315100)

[2](杭州高新区 (滨江) 区块链与数据安全研究院, 浙江 杭州 310059)

[3](嘉应学院 计算机学院, 广东 梅州 514215)

[4](深圳市腾讯计算机系统有限公司, 广东 深圳 518063)

[5](浙江大学 计算机科学与技术学院, 浙江 杭州 310027)

通信作者: 朱阅岸, E-mail: zhuyuean@jyu.edu.cn

摘　要: 数据库系统作为数据存储与处理的基础设施, 其性能对现代社会的运行效率具有重要影响. 随着内存技术的进步及 SSD (solid state drive) 的广泛应用, 磁盘数据库的性能瓶颈逐渐转向 CPU 的利用率和内存管理的优化. 现代数据库系统通常采用解释执行的方式处理查询, 造成了大量虚函数调用、上下文切换和高速缓存未命中, 不能充分发挥现代 CPU 的流水线和缓存机制, 导致低效的查询执行效率, 尤其是在大数据量和复杂查询的场景中表现更为明显. 为了解决上述问题, 针对传统数据库解释执行提供若干 JIT (just-in-time) 编译优化方案, 并在 MySQL 数据库中进行验证. 首先给出利用 LLVM (low level virtual machine) 编译器将 SQL 谓词在运行时转换为机器码的方案代替解释执行, 以减少虚函数调用和系统上下文切换的开销. 接着提出混合编译与解释执行的方案, 扩展了 JIT 编译执行的适用范围. 最后针对日益流行的可插拔数据库系统架构设计了一种将 JIT 机器码推送至存储引擎层的查询下推方案, 避免不必要的数据传输和计算开销. 实验结果表明, 启用 JIT 编译后, MySQL 的查询性能显著提升, 尤其在处理复杂查询和大数据量的场景, JIT 编译系统能够有效降低解释执行带来的开销, 显著提高系统的响应速度和吞吐量. 在类 TPC-H 的测试中, 对比原生 MySQL, 采用 JIT 编译执行的系统性能提升可达 148%.

关键词: JIT 编译; 混合编译执行; 查询优化; 谓词下推; 可插拔架构优化

中图法分类号: TP311

中文引用格式: 袁巩生, 杜晨路, 朱阅岸, 陈育兴, 唐秀, 姚畅, 陈刚. JIT编译技术在可插拔存储引擎数据库中的应用. 软件学报, 2026, 37(3): 1225–1239. http://www.jos.org.cn/1000-9825/7467.htm

英文引用格式: Yuan GS, Du CL, Zhu YA, Chen YX, Tang X, Yao C, Chen G. Applications of JIT Compilation Technology in Databases with Pluggable Storage Engines. Ruan Jian Xue Bao/Journal of Software, 2026, 37(3): 1225–1239 (in Chinese). http://www.jos.org.cn/1000-9825/7467.htm

Applications of JIT Compilation Technology in Databases with Pluggable Storage Engines

YUAN Gong-Sheng[1,2], DU Chen-Lu[1], ZHU Yue-An[3], CHEN Yu-Xing[4], TANG Xiu[1,2], YAO Chang[1,2], CHEN Gang[5]

[1](School of Software Technology, Zhejiang University, Ningbo 315100, China)

[2](Hangzhou High-tech Zone (Binjiang) Institute of Blockchain and Data Security, Hangzhou 310059, China)

[3](School of Computer Science, Jiaying University, Meizhou 514215, China)

[4](Shenzhen Tencent Computer System Co. Ltd., Shenzhen 518063, China)

[5](College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China)

Abstract: Database systems serve as the foundational infrastructure for data storage and processing, with their performance playing a critical role in the efficiency of modern society. With the advancement of memory technologies and the widespread adoption of SSDs

* 基金项目: 国家重点研发计划 (2023YFC3603103); 浙江省“尖兵领雁+X”研发攻关计划 (2024C01019)
收稿时间: 2025-01-09; 修改时间: 2025-03-03; 采用时间: 2025-05-08; jos 在线出版时间: 2025-12-17
CNKI 网络首发时间: 2025-12-25

(solid state drives), the performance bottleneck in disk-based databases has shifted towards optimizing CPU utilization and memory management. However, current database query execution often relies on interpreted methods, leading to numerous virtual function calls, context switches, and cache misses. This limits the ability of modern CPUs to fully utilize their pipelines and cache mechanisms, resulting in inefficient query execution, particularly in scenarios involving large datasets and complex queries. To address these issues, this study proposes several just-in-time (JIT) compilation optimization strategies for traditional interpreted database execution, validated through experiments on MySQL. First, an approach is presented wherein the LLVM (low level virtual machine) compiler is used to convert SQL predicates into machine code at runtime, replacing the interpretation method to reduce the overhead of virtual function calls and context switching. Next, a hybrid compilation and interpretation approach is introduced to extend the applicability of JIT execution. Finally, a query pushdown strategy is designed for pluggable database system architectures, enabling the transfer of JIT-compiled machine code to the storage engine layer to reduce unnecessary data transfer and computational overhead. Experimental results show that enabling JIT compilation significantly enhances MySQL's query performance. Notably, for complex queries and large datasets, the JIT-compiled system reduces CPU load and memory usage, leading to substantial improvements in system response speed and throughput. In TPC-H-like tests, compared to the native MySQL version, the optimized system shows performance gains of up to 148%.

Key words: just-in-time (JIT) compilation; hybrid compilation and execution; query optimization; predicate pushdown; pluggable architecture optimization

随着信息技术的迅猛发展, 数据的规模与复杂性呈现出指数级增长, 尤其是在大数据、人工智能和物联网等领域, 数据的生成与处理已成为技术革新的核心. 数据库作为数据存储与处理的核心, 其性能和效率直接关系到信息化社会的运行效率. 在数据处理过程中, 磁盘 I/O 往往是系统性能瓶颈. 得益于近些年来存储技术的飞速发展, 内存容量显著提升, SSD (solid state drive) 固态硬盘价格大幅下降从而逐渐替代传统硬盘, 从根本上缓解了 I/O 带来的问题. 因此, 数据库的性能瓶颈逐渐转向了 CPU 利用和内存管理方面的效率问题. 现代数据库系统要应对这一挑战, 需要更高效地利用计算资源、优化数据访问和处理方式. 目前数据库的查询执行大多仍采用 Volcano 模型[1]的解释执行方式. 系统将查询语句转换成由操作算子通过 GetNext 方法组合形成的查询执行树, 执行器采用迭代器机制, 通过 GetNext 方法按需从当前方法获取下一批数据. 每个运算符会向下游的子节点请求数据, 直到获取到满足条件的数据或子节点返回"无数据". 这个查询处理机制导致了大量的虚函数调用、上下文切换和高速缓存未命中[2]. 在现代硬件环境中, 特别是在处理复杂查询时, 其性能瓶颈愈发明显. 每次执行查询时, 系统会频繁地递归调用 GetNext 方法拉取数据, 执行相关计算. 频繁的虚函数调用带来的上下文切换使得系统无法充分利用现代 CPU 的流水线和缓存机制, 造成查询执行效率的低下, 尤其在大规模数据集上, 性能问题尤为突出. 因此, 如何减少虚函数调用, 优化计算过程, 成为现代数据库引擎亟待解决的问题之一.

近些年, JIT (just-in-time) 编译技术在数据库中愈发流行[3−7]. JIT 编译通过将查询操作转化为机器码, 在查询执行过程中动态生成优化后的可执行代码, 避免了传统解释执行带来的指令开销和函数调用开销. 与静态编译不同, JIT 编译能够根据代码实际执行路径和硬件环境的特点进行针对性优化, 如选择最优查询执行计划、内存优化、提升跨平台兼容等, 从而在查询执行时提升代码的执行效率. 这种优化方式不仅能有效减少解释执行带来的开销, 还能针对具体的查询逻辑和硬件特性进行定制化优化, 显著提高复杂查询的处理性能[8]. JIT 编译技术尤其适用于内存数据库和计算密集型任务, 能大幅提高数据库系统的吞吐量和响应速度.

鉴于以上研究背景, 本文针对传统 Volcano 引擎数据库提出若干 JIT 编译优化方案. 并在 MySQL 数据库[9]上验证了技术可行性. 主要贡献如下.

(1) 基于 LLVM (low level virtual machine) 编译器[10], 提出一种非侵入式 JIT 实时代码生成技术, 并保留系统原有的解释执行机制, 使得解释执行与 JIT 编译能够共存. 优化器根据查询执行代价自动选择最合适的执行方式, 为事务型与分析型场景融合提供一个可行的解决方案.

(2) 提出一种混合执行模式, 允许在同一个查询中同时存在编译与解释执行模式. 对不支持的查询树子树以函数调用的方式嵌入到生成代码中, 规避朴素 JIT 代码生成机制在子节点包含不支持的函数或数据类型的时候切换回解释执行模式, 浪费前置查询树节点的代码生成时间.

(3) 可插拔特性为数据库系统提供了灵活的数据存储选择, 但额外引入的中间层接口给数据传输带来不可忽

略的代价. 针对可插拔引擎架构, 设计将 JIT 编译生成的谓词机器码下推至存储引擎层的近数据节点计算 (近数计算) 方案. 在存储引擎执行过滤, 去除不符合条件的数据, 从而减少数据传输和中间层的句柄函数调用开销.

(4) 在开源数据库系统 MySQL 中实现上述提出的 JIT 代码生成技术, 称为 MySQLJ, 并对系统进行了详尽的测试. 通过 Microbench 和 TPC-H 基准测试[11], 对比了不同数据量、多种数据类型、不同查询选择率下开启 JIT 编译选项前后系统的性能表现. 实验结果表明, MySQLJ 能够在不同的业务场景下自动选择最优的执行方案, 显著提升数据库性能.

1 相关工作

在过去十多年中, JIT 编译技术已成为加速查询处理的关键技术. 本节将探讨 JIT 编译技术在数据库系统中的最新研究进展.

查询发送到系统后, 由优化器生成物理查询执行计划. 编译器在该物理查询执行计划基础上使用代码生成方法来生成本地代码. 总体来说存在两类方法来生成查询请求对应的代码. 一类是称为源到源的转译方法, 例如 HIQUE[12]. 对于给定的查询计划, 转译生成一个实现查询执行的 C/C++程序, 其中包括所有谓词和类型转换. 然后使用现成的编译器 (例如 gcc) 将代码转换为共享对象, 将其链接到数据库管理系统 (database management system, DBMS) 进程, 并调用 *exec* 函数来执行查询.

另一类方法是 JIT 即时编译, 使用 JVM、LLVM 等编译基础设施生成查询的中间表示 (intermediate representation, IR), 然后快速编译成本地机器码. 近年来, JIT 编译技术在数据库系统中的实现路径呈现多样化, 但受限于设计目标与架构约束, 现有方案在适用场景与工程扩展性上存在显著差异. HyPer 的作者[13]认为使用转译的方法存在编译速度慢、性能开销大等弊端[2]. 因此 HyPer 转而使用即时编译技术. 作为这个技术的先驱, HyPer 自 2011 年引入 JIT 编译技术, 主要关注扫描和处理大规模列式数据, 特别是在数据仓库和商业智能的场景中的应用. 最初 HyPer 利用 LLVM 编译器将查询操作编译成机器码, 摒弃传统的解释型执行流程. 为了克服 JIT 代码生成带来的巨大开销, HyPer 研究人员提出使用自适应执行框架, 该框架通过预估算子剩余执行时间可以动态地在解释执行和编译执行之间切换[14]. HyPer 的演进版本——Umbra 数据库系统, 持续优化代码生成的开销. 研究人员[15]提出了快速代码生成框架 Tidy Tuples. 它仅需单次遍历查询树即可生成代码, 并引入轻量级 IR 以支持快速代码生成和编译. 类似地, Funke 等人[16]通过引入 Flounder IR 避免传统 IR 的复杂性与代码生成的耗时. Flounder IR 是专为数据库查询处理设计的中间表示. 这种简化减少了编译过程中需要执行的分析和优化步骤, 从而缩短了编译的时间. 然而, HyPer 的优化高度依赖内存计算模型, 难以直接迁移至磁盘数据库或混合负载场景, 且其自研的 Tidy Tuples 框架需重构执行引擎, 导致与现有数据库生态兼容性不足. 相比之下, 本文提出的 JIT 编译方案基于通用的火山模型设计, 通过非侵入式代码生成 (见第 3.3 节) 适配 MySQL 数据库, 无须重写存储或执行层, 避免了其他研究中自底向上构建新引擎或重写编译器后端的复杂工作, 该方案面向通用的数据库执行器.

PostgreSQL[17]是一个广泛使用的关系型数据库, 在 2018 年发布的版本 11 中同样通过使用 LLVM 编译器基础设施引入了 JIT 编译[18], 目的是提升计算密集型操作的性能. 但其实现与存储引擎紧耦合, 无法适配存算分离或可插拔架构的算子下推需求. 例如, PostgreSQL 的 JIT 编译代码仅在服务层执行过滤, 无法将机器码下推至存储引擎层提前过滤数据, 导致冗余 I/O. 而本文提出的机器码下推方案 (见第 3.4 节) 通过字段裁剪与近数据计算, 在存储层直接过滤非匹配记录, 减少了数据传输开销. 尽管 HyPer 和 PostgreSQL 都采用了 JIT 编译来提升查询性能, 但它们的 JIT 编译实现并未充分考虑现代数据库架构中的一些新兴需求, 尤其是在可插拔引擎和存算分离架构中的适应性.

近年来, 不断有新的 JIT 编译框架与方法提出. Grulich 等人[19]提出了一个名为 Nautilus 的框架, 旨在结合查询解释的易用性和查询编译的性能. 允许根据工作负载特性动态选择不同的编译后端, 以平衡编译时间和执行性能. 并通过追踪技术, Nautilus 能够在运行时动态优化热代码路径, 减少编译时间并提高执行效率. Haffner 等人[20]提出了一种查询编译架构, 利用 WebAssembly 作为中间表示, 并使用 Google 的 V8 引擎作为后端. V8 提供了两级

编译: 快速编译 (Liftoff) 和优化编译 (TurboFan), 这使得在执行过程中可以逐渐替换代码, 实现从非优化代码到完全优化代码的动态替换, 从而在不牺牲性能的同时显著减少编译时间. WebAssembly 是一个新兴的技术, 它不提供标准库, 这意味着需要为每个查询定制所有必要的算法和数据结构的代码, 不如传统的编译基础设施 (如 LLVM) 成熟. 并且该架构主要针对特定的硬件架构 (x86 体系) 进行了优化. 对于不同的硬件架构, 可能需要额外的工作来确保代码的高效执行. 得益于项目的模块化设计, 上述编译器框架可以作为本文代码生成技术的候选, 替换代码生成相关 LLVM C++ API 即可.

与本文工作最相关的是华为云 GaussDB for MySQL 以及 AWS Aurora 针对云数据库的优化[21,22], 将 SQL 查询的过滤操作编译成字节码然后下推到存储层, 达到近数计算效果 (华为使用 LLVM 将 SQL 查询转换成字节码, AWS Aurora 研究团队没有公布采用的技术细节). 上述两家云计算厂商在现有成熟的数据库系统上将 SQL 转换成字节码, 使用优化器动态确定解释执行或者是编译执行过滤操作. 他们的技术也可以推广到可插拔引擎架构下. 但是他们并未给出相关的技术实现细节, 也不支持混合执行模式.

2 系统整体架构

针对 Volcano 执行器的不足以及日益流行的可插拔数据库系统架构, 本文相应给出若干 JIT 编译优化方案, 并在 MySQL 数据库中进行验证, 我们称其为 MySQLJ.

图 1 展示了 MySQLJ 的整体框架与对应的论文内容组织, 包含以下 4 个重要模块.

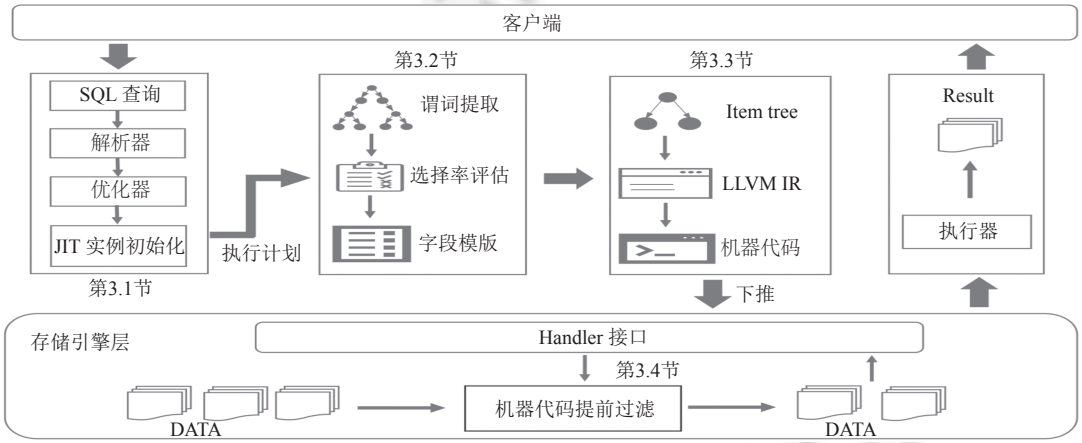

图 1 JIT 编译系统的整体架构设计示意图

(1) JIT 编译准备模块. 该模块主要功能是根据查询特点决定使用编译执行还是系统的解释执行. 首先, 模块分析 SQL 查询所涉及的元组数目和谓词表达式数量, 并与预设的阈值进行比较, 判断是否选择 JIT 编译. 若使用编译执行, 接下来编译模块将构建一个适配目标机器的 JIT 编译实例. 具体来说, 模块首先会创建执行控制对象来管理编译过程中的资源和执行环境. 然后, 生成适用于目标机器的编译环境, 并获取目标机器相关信息例如 CPU 型号、内存布局等, 以确保生成的代码与硬件兼容. 最后, 编译模块会创建执行会话和相关管理工具, 确保编译后的代码能够顺利加载并执行. 我们将在第 3.1 节介绍 MySQLJ 编译执行的适用场景及 JIT 编译实例的初始化过程.

(2) 表达式筛选模块. 该模块的主要功能是筛选出适合下推至存储引擎的谓词表达式. 随着可插拔架构的日益流行, 将查询条件下推到存储引擎在很多场景中可以带来实质性收益[23]. 因此, 将 JIT 编译系统引入条件下推, 扩展了优化范围, 特别是在复杂查询中, 条件下推能够有效减少计算开销. 在筛选条件时, 模块结合优化器的统计信息, 评估各个谓词的选择率, 优先下推选择率较低的谓词, 以减少数据量传输和接口调用. 为了避免无效下推, 只有在谓词选择率低于给定阈值时, 才会将查询条件下推至存储引擎. 筛选出的条件会构建成一棵新的表达式树, 并将运行时所需的过滤条件相关的字段编号和条件数量的信息一起下推至存储引擎. 剩余条件通过解释执行的方式处

理. 如何筛选下推表达式将在第 3.2 节进行详细探讨.

(3) 表达式编译模块. 该模块的主要功能是将筛选出来的谓词表达式转换为能够直接执行的机器码, 这个过程是 JIT 编译系统的核心. 首先, 表达式编译模块以自底向上的方式遍历表达式树, 利用 LLVM C++ API 将其转化为 LLVM IR (LLVM intermediate representation). LLVM IR 是编译过程中的核心表示形式, 它比机器码更接近高级语言, 且与目标平台无关. 为了提升编译效率和避免重复计算, 编译模块会执行一些常见优化操作, 如常量折叠、常量传播等. 最后, 表达式编译模块会根据不同硬件架构的特性, 将 LLVM IR 编译为目标机器码并链接到原本的查询执行进程中, 机器码将在查询执行时动态加载和执行. 本文第 3.3 节将给出表达式编译具体算法流程.

(4) 引擎层过滤模块. 该模块利用下推的机器码, 在存储引擎层对数据进行提前过滤. 以 InnoDB 引擎为例, 磁盘中存储的数据通常采用 InnoDB 格式. 当进行全表扫描时, 传统方法需要将整条记录从磁盘读取并转换为 MySQL 格式, 这一过程涉及大量的格式转换和数据复制开销. 引擎层过滤模块通过下推的字段编号, 仅转换查询所需的字段数据, 避免了对整条记录的格式转换. 接着, 模块根据下推的机器码对字段数据进行表达式求值, 只有满足谓词条件的记录才会被进一步处理并转换为 MySQL 格式. 经过过滤后的数据会加入缓冲池中, 而不符合条件的记录则会被过滤掉. 当下推的谓词整体选择率较低时, 这一处理方式能够有效避免将无用数据加载到缓冲池中, 减少了内存和 I/O 开销从而提升性能. 后续的实验验证了这一点. 最终, 缓冲池中符合条件的数据会被上传到服务层进行后续的操作. 在第 3.4 节阐述存储层过滤操作的实现细节.

3 基于 LLVM 编译器的 JIT 编译方法

本节将详细介绍 JIT 编译技术各个模块的实现细节, 包括使用 JIT 编译的考量、下推表达式的筛选以及物理查询执行计划之上生成代码的算法细节等.

3.1 JIT 编译准备模块

本节首先介绍解释执行和编译执行各自的优缺点, 然后介绍 MySQLJ 优化器如何选择解释执行还是编译执行, 最后介绍 JIT 编译实例的初始化过程.

3.1.1 MySQL 中的表达式求值

表达式求值在 SQL 查询中无处不在, 涵盖了过滤、聚合、投影, 甚至连接操作等多个方面. 本文主要聚焦于 SQL 查询中的谓词表达式. 谓词表达式由 n ($n\geqslant 0$) 个 AND/OR 逻辑操作组合条件判断, 形成谓词表达式子句. 在某些场景下, 谓词表达式的计算开销可能占据大部分查询执行时间, 成为系统瓶颈. 图 2 展示了在执行 TPC-H 基准查询 Q6 时, 谓词表达式计算所带来的 CPU 开销.

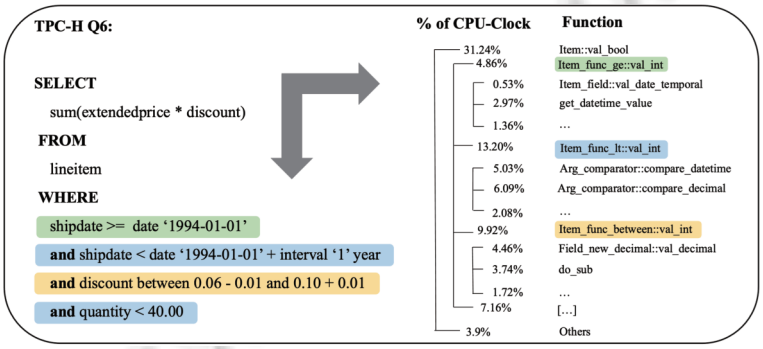

图 2 TPC-H 查询 Q6 及其谓词表达式求值 CPU 开销

实际上, 在 TPC-H 数据集的测试中, 复杂表达式的低效计算已被视为数据库查询的主要瓶颈. 图 2 右侧展示了在解释执行方式下应答 TPC-H 查询 Q6 时谓词表达式求值相关函数的 CPU 调用开销. 从图中可以看到大部分

CPU 开销 (31.24% + 3.9% = 35.14%) 花费在谓词表达式求值上, 并且这种虚函数调用, 需要在运行时动态查找函数地址, 导致频繁的 L1 高速缓存未命中, 成为数据库性能的瓶颈.

3.1.2 JIT 编译的选择

本质上, JIT 编译方法是将原本的解释执行时间 $t_J(N)$ 替换为编译时间 t_C 与优化后机器码执行时间 $t_E(N)$ 的总和, 其中 N 代表查询处理的数据量. 编译过程中的优化使得生成的机器码更加高效, 并避免了频繁的函数调用开销, 因此 $t_E(N) < t_J(N)$ 始终成立. 但为了使 JIT 编译具有优势, 需要满足 $t_C + t_E(N) < t_J(N)$, 即查询计划的解释执行时间必须大于编译和执行优化后机器码的总开销.

是否满足上述条件, 主要取决于查询处理的数据量和查询复杂度. 对于每一条记录, 解释执行与编译执行的差值都是 $t_J(i) - t_E(i)$, 因此数据量的大小直接决定了 $t_J(N) - t_E(N)$ 的差值. 当数据量较小时, $t_J(N) - t_E(N) < t_C$, 此时 JIT 编译方法反而会导致性能下降, 因此不适合使用 JIT 编译. 另一个影响因素是查询的复杂度. 如第 3.1.1 节中所述, 在 TPC-H Q6 查询中, 谓词表达式求值的 CPU 开销占比为 35.14%, 而在一些 OLTP 类查询中, 谓词表达式求值的 CPU 开销只有不到 10%, 此时 $t_J(N) - t_E(N)$ 的差值会大幅缩小. 尽管查询复杂度降低, 编译的时间开销也会有所减少, 但编译时间 t_C 存在一个基础值, 该基础值主要为环境初始化与动态链接的时间开销. 因此, 在复杂度较低的查询中, 由于 $t_J(N) - t_E(N)$ 的差值大幅缩小和 t_C 的固定开销, 导致 $t_C + t_E(N) > t_J(N)$, 此时 JIT 编译会导致性能下降.

在编译准备模块中, 是否启用 JIT 编译需综合评估数据规模 (N) 与查询复杂度 (M). 其中, N 表示待处理的数据量, M 为查询中谓词表达式的数量. 二者的乘积反映了查询的总计算开销: 数据量越大、谓词表达式越多, 解释执行的函数调用开销也越大, JIT 编译的效果越显著. 为了避免在小数据量场景或低查询复杂度场景下 JIT 编译开销大于性能收益, 导致性能下降的问题, MySQLJ 通过优化器统计信息动态计算 $N \times M$, 并与预设阈值 K 比较, 仅当满足 $N \times M > K$ 时触发 JIT 编译. 阈值 K 的设定基于编译开销与典型查询解释执行成本的权衡, 具体值通过实验校准 (见第 4.4 节).

3.1.3 JIT 实例初始化

JIT 实例的初始化过程主要包括设置执行会话、目标机器、数据布局和编译过程. 首先, 系统创建了执行控制对象, 并基于此创建了 ExecutionSession, 负责管理 JIT 编译的整个生命周期, 包括内存管理和机器执行的管理. 接着, 系统构建了目标机器, 目标机器是执行编译后代码的核心组件. 成功创建目标机器后, 系统会获取平台的数据布局 (DataLayout), 该布局定义了数据在内存中的存储和访问方式. 在获取数据布局后, 系统创建了一个工具来管理动态调用, 确保在代码执行时所有的调用都能正确处理. 接下来, 编译过程被分为几个层次: ObjectLayer 负责管理生成的对象文件, CompileLayer 负责将中间代码编译成目标机器码, OptimizeLayer 对代码进行优化, 以提升执行效率. 最后, 系统通过创建 MainJD 动态库并将其与当前进程结合, 确保编译出的代码可以被动态加载并执行.

3.2 表达式筛选模块

在可插拔引擎架构的数据库系统中, 服务层与存储引擎层之间的频繁数据交互可能导致大量冗余数据传输, 从而显著影响查询效率. 传统优化方案如索引条件下推 (index condition pushdown, ICP) 通过将索引相关条件下推[23]至存储引擎层, 在数据访问阶段提前过滤记录, 以减少服务层与存储引擎之间的数据传输开销. 然而, ICP 存在明显局限性: (1) 仅支持索引条件且无法处理子查询; (2) 当索引条件选择率较高时过滤效果有限; (3) 难以适配存算分离架构 (如云存储场景), 因存储层与计算层分离时需额外考虑网络传输与数据转换成本.

为了解决上述问题, 本文提出一种全量谓词条件下推机制. 与 ICP 仅下推索引条件不同, 该机制通过动态代码生成技术, 将查询中低选择率的谓词条件 (无论是否涉及索引) 编译为机器码并下推至存储引擎层直接执行. 其实现方式借鉴了 ICP 的代码下推思想, 但进行了显著改进以突破其限制: 该机制不再依赖索引, 支持任意类型条件的下推, 包括非索引字段和复杂表达式; 通过预编译机器码减少存储层与计算层之间的原始数据传输, 特别适用于网络带宽受限的云环境; 并基于选择率动态判定下推条件, 避免高选择率谓词条件引入额外开销. 具体来说, 在

MySQL 中, 表达式通过由多个 C++类 Item 的实例组成的 Item 树来表示, 每个节点都表示表达式的不同组成部分, 如操作符、常量值和字段等. 在该模块中, 首先会遍历 Item 树, 并结合优化器中的统计信息评估各个谓词的选择率. 只有当谓词的选择率低于设定的阈值时, 才会将该谓词加入下推条件中. 此外, 还需要判断 Item 节点是否属于支持的操作符、字段或常量类型. 如果当前节点尚未支持, 则后续采用混合执行方式, 即同时使用编译执行和解释执行, 以确保查询的正确性. 对于符合下推条件的谓词, 模块会构造一个新的 Item 树并构建一个包含下推条件字段编号、条件数量等信息的模板 (template). 该模板与机器码一起下推至存储引擎, 以确保存储引擎能够正确识别并执行下推条件. 对于不符合下推条件的谓词, 模块会重新构造一个剩余条件的 Item 树, 确保查询在服务层的后续处理能够保持完整性和准确性.

3.3 表达式编译模块

MySQLJ 的实现目标是将查询中的谓词表达式转换为高效的机器码, 使得生成的机器码比现有系统生成的指令更少, 并且生成的机器码能够在同一进程中执行. MySQLJ 会将每个 Item 树转换为一个独立的 LLVM IR 模块, 并利用 LLVM 的优化模块进行后续处理. 然后即时编译为本地机器码, 动态链接到当前进程中.

3.3.1 JIT 编译的时机选择

MySQLJ 采用了一种类似于“插件”的非侵入式方法在 MySQL 中实现 JIT 编译系统, 这种方法不会对原有的 Volcano 模型有任何影响, 遵循现有的过滤操作实现逻辑.

如前所述, MySQL 的解析器将输入查询的谓词表达式转化为一个或多个 Item 组成的表达式树, 这些表达式树作为表达式的内部表示存储在查询块中. 在查询优化过程中, 优化器会根据每个执行路径的代价估算来选择最优执行路径. 当查询计划确定后, 执行路径会被转换为具体的存取访问迭代器. 这些迭代器负责执行实际操作, 如表扫描、索引扫描和排序等. 对于谓词表达式, 它会被转换为过滤迭代器. 过滤迭代器会在执行过程中, 通过调用 *Item*::*val_int* 评估每一行数据是否满足条件. 在查询优化器最终确定执行计划并生成执行路径之后, 应用 JIT 编译生成相应的代码. 此时, 所有优化工作已经完成, 查询的执行路径已经固定, 表达式树不再发生变化, 避免了重复编译. 因此, JIT 编译的操作放在访问路径选择之后, 且在创建迭代器时刻进行编译. 通过这种方式, JIT 编译的机器码可以与迭代器一起被使用. 在查询执行时, 存取访问迭代器将直接使用这些编译后的代码来判断每一行是否满足谓词表达式的条件, 从而减少了传统解释执行的开销.

3.3.2 代码生成

我们为每种 Item 类型实现了相应的 CodeGen 方法, 用于生成对应的 LLVM IR. 例如, 对应 Item_int, 系统会调用 LLVM C++ API getInt64 生成表示 64 位整数的 LLVM IR; 对应 Item_and, 系统会调用相应的 LLVM C++ API CreateAnd 生成处理逻辑与操作的 LLVM IR. 大部分操作符、字段和常量类型都实现了各自的 CodeGen 方法. 在调用 CodeGen 时, 每个 Item 会根据其类型生成相应的 LLVM IR, 其子节点递归地调用 CodeGen, 最终生成整个函数体的 LLVM IR. 通过这种方式, 实现了为整棵查询执行树根据其节点类型动态生成 LLVM IR.

为了更清楚地解释代码生成机制, 这里给出一个 WHERE 语句的条件表达式“cond: ($a<1$ AND $b>2$) OR $c>=3$”作为例子演示代码生成步骤. 如图 3 所示, 该条件表达式 cond 按照树形结构组织, 代码生成按照自底向上的方式进行. 每一个算术表达式和过滤条件 e 都被视为一个单元, 编译到单独一个函数 f. 在触发评估表达式的时候, 只需调用一次相应的函数 f 即可. 首先, 转化逻辑将谓词表达式 p_1 看作一个编译单元 e_1, 将其编译成字节码<p_1(%a)>. 根据字节码的执行结果 %p_1, 程序利用分支跳转指令 *br* 跳转到分支 %l_0 或者 %l_2. 按照逻辑 AND 的语义, 程序可能直接跳转 %l_2 省略对谓词 p_2 的字节码<p_2(%b)>执行. 在分支 %l_0, 编译单元 e_2 被编译成字节码<p_2(%b)>. 此时逻辑函数 AND 的语义构建完成. 注意, 从逻辑函数 OR 的角度来看, 逻辑 AND 也是一个转换单元, 记为 e'. 依据转换单元 e'的字节码执行结果 %p_2, 程序跳转到分支 %l_1 或者分支 %l_2, 接着生成占位符Ⓣ与Ⓕ的代码. 按照逻辑 OR 的语义, 占位符Ⓣ直接生成返回 true 的指令, 而占位符Ⓕ则需要评估谓词 p_3 的字节码<p_3(%c)>的值, 然后返回相应的结果. 至此, 逻辑函数 OR 最后一个参数谓词 p_3 字节码构建完成, 那么逻辑函数 OR 的代码生成完成, 条件表达式 cond 的代码生成工作同时也结束.

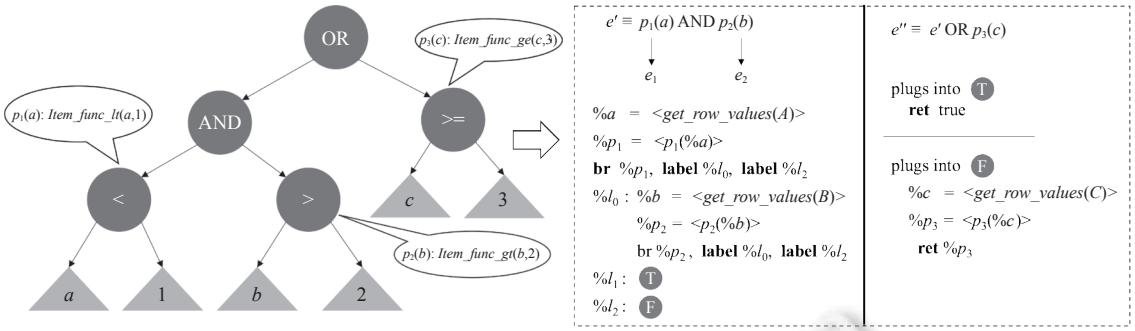

图 3 谓词表达式代码生成示例

3.3.3 混合编译

MySQL 支持多种数据类型和功能, 其中许多类型 (如字符串、日期和复杂几何类型) 在表达式求值时引入了复杂的处理逻辑. 例如, 字符串类型支持不同的字符集和排序规则, 需要根据标准进行正确的处理; JSON 或 GIS 类型的求值则依赖动态链接库实现空间计算或数据解析, 这类操作在解释执行时通过 MySQL 内置函数逐行处理, 但在 JIT 编译中面临显著挑战. 以 JSON 字段的提取操作为例, 其底层需调用 JSON_EXTRACT 函数解析字符串并构建内存中的树状结构, 这一过程涉及动态内存分配和递归遍历, 难以直接映射到静态的 LLVM IR. 因此, 考虑到支持所有类型、运算符和函数的实现复杂性, 并且这些功能在本研究中并不会带来实质性提升, MySQLJ 将重点支持常见的数据类型和运算符 (如 INT、CHAR、VARCHAR、DATE 和 DECIMAL) 以及基本的逻辑运算符 (AND、OR、NOT) 和比较运算符 (>=、<=、BETWEEN、!=、= 和 LIKE). 这些类型和运算符覆盖了大多数实际业务场景, 能够有效验证 JIT 编译技术对 MySQL 查询性能的提升效果.

对于未支持的数据类型或运算符, MySQLJ 采用了混合编译的方式. 具体来说, 对于 Item 树中不支持的数据类型, 我们不会对其进行完整的编译, 而是通过外部调用机制来处理这些不支持的类型. 这意味着, 当编译生成的代码在动态执行时遇到这些节点时, 它会回调虚函数 (如 *val_int*()) 来完成表达式求值. 这种处理方式虽然没有完全消除该节点的函数调用开销, 但可以确保其他支持的节点能够通过编译执行, 从而避免使用解释执行. 通过这种混合编译方法, 即便在遇到不支持的类型时, 系统依然能够正常编译执行, 尽管存在少数未支持的类型需要回退到虚函数调用, 但整体查询性能依然能得到显著改善.

3.4 引擎层过滤模块

在 MySQL 中, 不同的存储引擎采用不同的存储和访问方式. 以 InnoDB 引擎为例, 其采用了聚簇索引存储格式, 数据行存储在 B+ 树的叶子节点中. 当进行全表扫描时, 整条记录需要从磁盘读取并转换为 MySQL 格式, 这一过程涉及大量的格式转换和数据复制开销. 为了减少这一开销, 存储引擎层的过滤模块通过下推的字段编号, 仅转换查询所需的字段数据, 避免了对整个数据行的转换. 接下来, 模块通过下推的机器码对字段数据进行表达式求值, 只有符合条件的记录会被进一步处理并转换为 MySQL 格式.

经过筛选后的数据会加入缓冲池中, 而不符合条件的记录则会被过滤掉. 当下推谓词的选择率较低时, 这种处理方式能够显著减少无效数据的加载, 降低内存和 I/O 开销. 最终, 缓冲池中的数据会被传输到服务层做进一步处理. 通过在存储引擎层提前过滤数据, 不仅避免了不必要的数据加载, 还有效减少了存储引擎与服务层之间的数据传输 I/O 开销.

4 实验评价

本节主要对采用 JIT 编译的 MySQLJ 与原生 MySQL 进行对比实验. 首先, 利用 sysbench 工具测试单表查询在不同数据量、数据类型以及是否包含主键这 3 种场景下开启 JIT 编译系统前后的数据库性能表现. 随后, 测试

混合编译情况下 MySQLJ 的性能表现. 最后, 使用 TPC-H 数据集, 测试不同查询命中率以及是否启用下推这两种场景下 MySQLJ 的性能表现, 并对以上实验结果进行评估与分析.

4.1 实验环境

为了确保实验结果的客观性与准确性, 所有实验均在同一集群环境下进行. 集群的软硬件环境如表 1 所示.

表 1 实验测试集群软硬件环境配置

类别	参数
CPU	Intel® Xeon® Gold 6240R CPU @ 2.40 GHz 16物理核心, 16逻辑核心
内存	64 GB DRAM
操作系统	Ubuntu 20.04.6 LTS
MySQL版本	MySQL 8.0.31
测试工具	sysbench 1.0.17
TPC-H	TPC-H 3.0.1

4.2 单表查询测试

本节通过 Microbench 基准测试展示单表查询场景下启用 JIT 编译系统前后的性能表现. 表 2 列出了用于测试的查询示例 (以 INT 类型为例), 其余数据类型的测试均基于相同的 SQL 查询模板进行.

表 2 Microbench 基准测试查询示例 (INT)

类型	查询
不含主键	SELECT *a*, *b* FROM *test* WHERE (*a* > [int] AND *a* < [int]) AND (*b* > [int] AND *b* < [int]) AND (*a* != *b*);
含有主键	SELECT *a*, *b* FROM *test* WHERE (*id* > [int] AND *id* <[int]) AND (*a* > [int] and *a* < [int]) AND (*b* > [int] and *b* < [int]) AND (*a* != *b*);

4.2.1 数据量与数据类型对 JIT 编译的影响

本节旨在展示不同数据量和数据类型对 JIT 编译系统的影响. 测试涵盖 INT、DECIMAL、DATE 和 VARCHAR 这 4 种数据类型, 查询数据量分别为 10 万、50 万、100 万、400 万、800 万和 1 600 万, 使用表 2 中不含主键的 SQL 查询.

实验结果表明, JIT 编译在小数据量场景下可能导致性能下降. 如图 4 所示, 当数据量为 10 万时, 各个数据类型在使用 JIT 编译后执行时间下降 3%–9% 不等. 这是由于在小数据量场景下查询整体的执行时间较短, 而编译时间占总查询时间的比例较大, 导致边际效益为负, 无法满足第 3.1.2 节所述的 $t_C + t_E(N) < t_I(N)$. 因此, 在此情况下启用 JIT 编译后, 执行时间反而有所增加. 当数据量增加到 50 万时, INT 类型和 DATE 类型在启用 JIT 编译后, 相较于解释执行, 执行时间略有减少, 而 DECIMAL 类型和 VARCHAR 类型的测试结果则几乎与解释执行的结果持平. 随着数据量的增长, 执行时间显著降低. 数据量达到 1 600 万时, INT 类型的执行时间通过 JIT 编译降低了 24.48%, DATE 类型降低 35.69%, 而 DECIMAL 和 VARCHAR 类型分别降低约 10%, 这一结果验证了 JIT 编译的有效性. 然而, 随着数据量进一步增加, 执行时间的减幅呈递减趋势. 例如, INT 类型从 100 万增至 400 万时执行时间减少 13.24%, 而从 800 万增至 1 600 万仅减少 2.25%. 这表明数据量扩大导致编译时间在总查询时间中占比减少, 边际效益降低, 最终趋于饱和. 此时, 执行时间的减少比例可按公式 (1) 计算, 并最终趋近于该值:

$$Improvement = \frac{t_J(N) - t_E(N)}{t_E(N)} \tag{1}$$

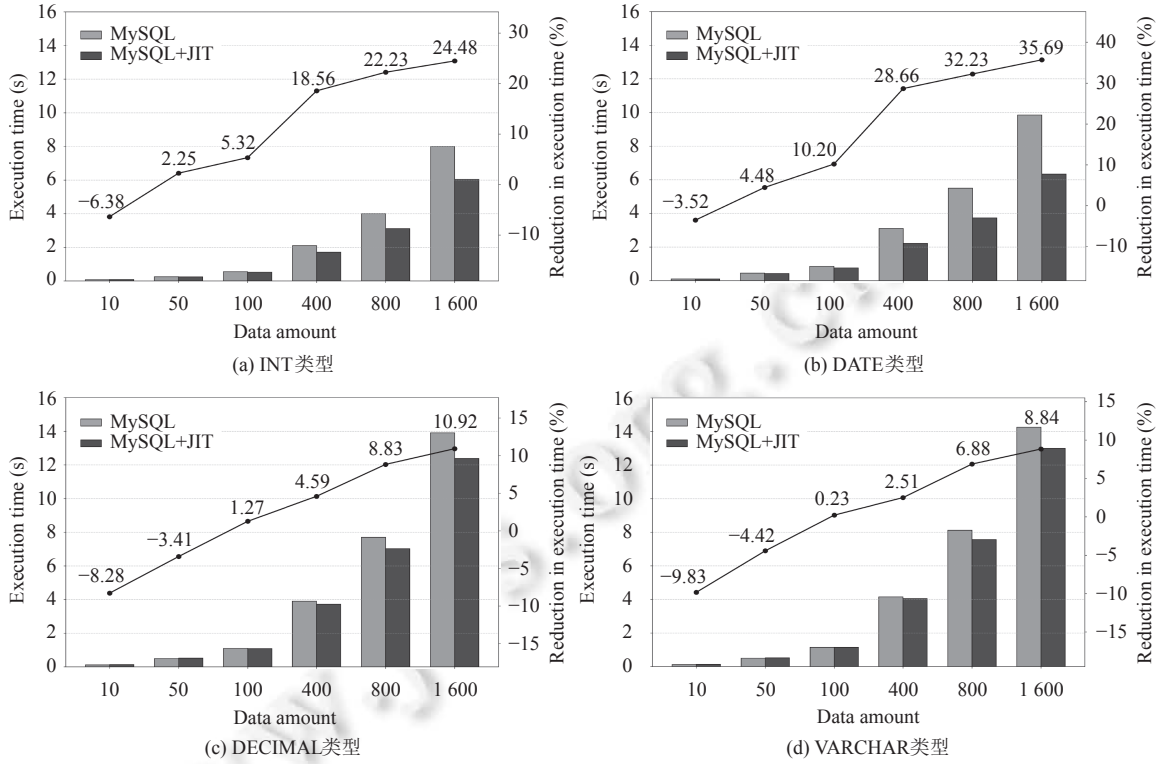

图 4 Microbench 基准测试结果

对于不同的数据类型, 启用 JIT 编译系统后执行时间减少的幅度各不相同. 实验结果显示, 在 1 600 万数据量场景下, DATE 类型减幅最显著 (35.69%), 其次是 INT 类型 (24.48%), DECIMAL 和 VARCHAR 类型则相对较小 (分别为 10.92% 和 8.84%). 对于 DECIMAL 和 VARCHAR 类型, 由于工作量较大, 本研究仅编译了调用栈底层的若干关键函数. 当使用 JIT 编译执行时, 仍有部分函数需要通过 MySQL 的 API 调用来处理, 限制了 JIT 编译对这两类数据类型的性能提升. 因此, DECIMAL 和 VARCHAR 类型在启用 JIT 编译后, 执行时间减幅相对有限. 而 DATE 类型优化效果突出的原因在于其解释执行过程比 INT 类型更复杂, 需调用 *Field_newdate*::*get_date_internal*() 进行逐层解析: 从存储引擎读取的日期值以 3 字节压缩格式存放, 再通过位掩码和移位操作将日期值拆分为年、月、日, 最后将结果打包为整数, 这一过程涉及多次虚函数调用. JIT 编译的优化策略主要为通过函数内联和指令融合, 将分散的操作合并为高效指令序列, 消除了虚函数调用开销. INT 类型的解释执行主要通过 *val_int*() 获取值并直接比较. JIT 优化主要体现在内联 *compare_int_unsigned*() 及相关函数, 将比较逻辑直接编译为硬件指令. 由于 INT 操作本身已接近硬件效率, 其优化空间小于 DATE 类型. 因此, 在查询复杂度相同的情况下, DATE 类型解释执行与编译执行之间的时间差距也更大. 根据公式 (1) 可知, 这意味着 DATE 类型的执行时间减幅将显著大于 INT 类型.

综上所述, JIT 编译系统在不同数据类型和不同数据量下的性能提升效果存在显著差异. JIT 编译在处理大规模数据和复杂查询时能够显著减少执行时间, 但在处理小规模数据时, 由于编译开销较高, 可能导致执行时间增加. 同时, DATE 类型由于其复杂的求值过程和较高的函数调用开销, 受益最大; INT 类型次之; DECIMAL 和 VARCHAR 类型由于部分函数仍需通过 API 调用, 执行时间减幅较为有限. 因此, 合理选择启用 JIT 编译的条件对于优化整体系统性能至关重要.

4.2.2 有无主键对 JIT 编译的影响

本节旨在展示谓词表达式中是否包含主键对 JIT 编译系统性能的影响. 实验采用表 2 中的 SQL 查询模板, 与原生 MySQL 进行对比. 实验结果如图 5 所示, 当数据量为 10 万时, DATE 类型在包含主键的情况下, JIT 编译相比于解释执行性能提升有 1.8%, 而不含主键时 JIT 编译性能是低于解释执行的. 当数据量为 50 万时, INT 类型和 DATE 类型的性能提升分别由 2.25% 和 4.48% 提升到了 9.98% 和 12.57%, 进一步表明当查询中含有主键时, 在小数据量场景下, JIT 编译也能有一定的性能提升.

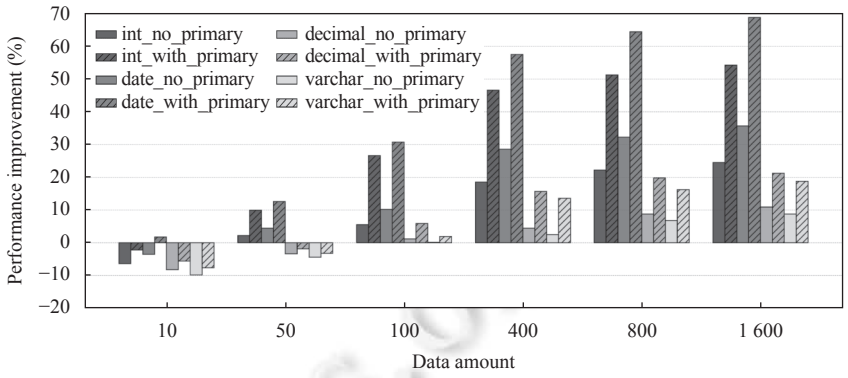

图 5 有无主键场景下的性能提升

随着数据量的增加, 主键对 JIT 编译的优化效果更为明显, 在 1 600 万条数据的情况下, 包含主键的 INT 类型查询通过 JIT 编译实现的性能提升达 54.20%, 而 DATE 类型的性能提升更为显著, 达到 68.89%. DECIMAL 和 VARCHAR 类型在启用 JIT 编译后, 性能提升幅度也分别达到了约 20%. 值得注意的是, 随着数据量的增加, 性能提升的比例逐渐趋于饱和, 这一趋势与第 4.2.1 节的研究结果相一致.

当查询中包含主键时, MySQL 能够利用主键索引快速定位目标数据行, 避免了全表扫描. 这种优化使得上传到服务层进行处理的数据量显著减少, 从而降低了 Handle 接口的 I/O 开销. 由于数据传输和 I/O 操作的减少, 过滤操作在总查询处理中的占比相对提高, 这为 JIT 编译系统提供了更大的优化空间.

4.3 混合编译测试

本节将展示混合编译的性能表现. 测试将涵盖 INT、DECIMAL、DATE、VARCHAR 和 TINYTEXT 这 5 种数据类型, 查询数据量分别为 10 万、50 万、100 万、400 万、800 万和 1 600 万. 下面为测试所用的 SQL 查询示例.

```
SELECT *
FROM test
WHERE (a > [int] AND a < [int])
AND (b > [date] AND b < [date])
AND (c > [decimal] AND c < [decimal])
AND (d > [varchar] AND d < [varchar])
AND (e > [tinytext] AND e < [tinytext]);
```

SQL 查询所涉及的数据类型中, TINYTEXT 是目前尚未支持的数据类型, 因此整体的 JIT 编译会使用混合编译的方式. 当 JIT 编译后的机器码实际执行时, TINYTEXT 的表达式求值是通过调用 *val_int*() 实现的, 本质上是解释执行的方式. 测试结果如图 6 所示, 在小数据量场景下, 混合编译的性能依旧不如解释执行. 但随着数据量的增加, 混合编译的优势逐渐显现. 特别是在数据量达到 1 600 万时, 执行时间减少了 20.89%. 尽管存在 TINYTEXT 类型的求值需要回退到虚函数调用, 在一定程度上影响了 JIT 编译的性能, 但查询执行时间依然能得到显著降低.

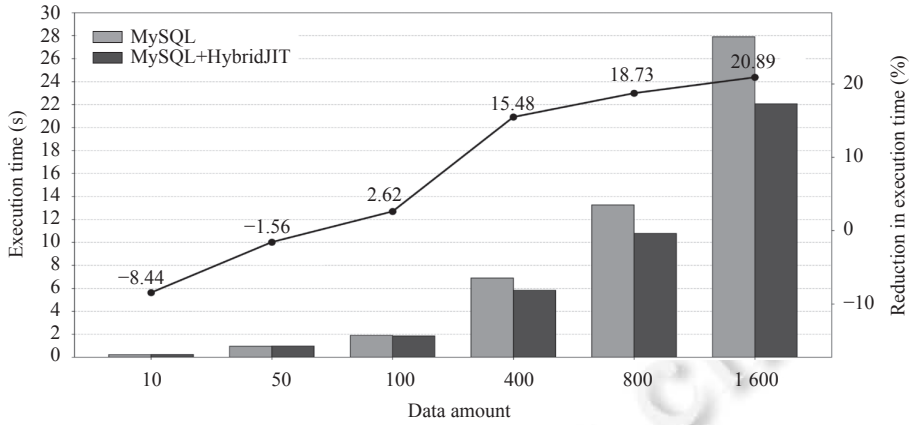

图 6 混合编译测试结果

4.4 阈值 *K* 的选取与验证

如本文第 3.1.2 节所述, 阈值 K 是 JIT 编译决策的核心依据. 为了验证阈值 K 的合理性, 明确其临界范围并为实际部署提供参考依据. 本节基于 TPC-H 基准测试框架, 通过动态调整 Q6 查询的复杂度与数据量, 系统化分析阈值 K 的临界范围及其对性能的影响. 实验通过修改 Q6 的 WHERE 子句中的谓词表达式数量构造不同复杂度的查询, 同时调整数据量使 $N\times M$ 的乘积覆盖 1×10^5 至 1×10^7, 观测不同 $N\times M$ 乘积对 JIT 编译性能的边际效益.

如图 7 所示, 当 $N\times M$ 低于 3×10^6 时, JIT 编译因编译开销占比过高导致性能收益为负; 当 $N\times M$ 高于 3×10^6 时, 性能的提升趋于稳定, 这表明此时编译开销可被大规模数据处理的效率优势完全覆盖. 这一临界范围的确定为实际部署提供了重要参考, 确保编译决策始终适配当前负载特征, 从而在保障稳定性的同时最大化 JIT 编译的收益边界.

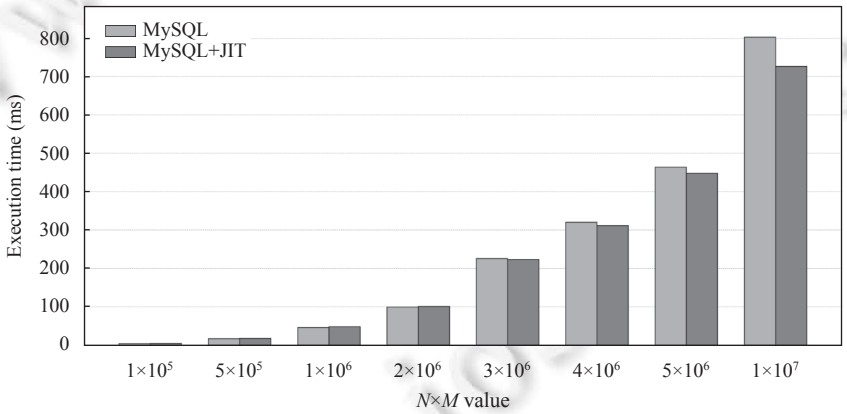

图 7 阈值 *K* 的临界值决策

4.5 TPC-H 测试

本节将展示 TPC-H 基准测试的实验结果. 在第 4.2.1 节中已证明, 数据量的增大与 JIT 编译性能提升成正比. 因此, 本节固定 TPC-H 的比例因子为 4 (即 4 GB 数据), 重点分析在复杂 OLAP 类查询中启用 JIT 编译系统的性能提升效果. 此外, 我们还将探讨在结合谓词下推的场景下, 查询条件的选择率对 JIT 编译系统性能的影响.

4.5.1 谓词下推对 JIT 编译的影响

本节将展示 JIT 编译与谓词下推结合的效果. 如图 8 所示, 仅启用 JIT 编译即可使大多数查询的执行时间减

少 5%–30%, 这表明我们的 JIT 编译系统能够有效提升 OLAP 类查询的谓词表达式求值效率. 某些查询如 Q8 和 Q13 在单独使用 JIT 编译时并未表现出显著的执行时间减少. 这是因为这两个查询中的谓词表达式相对简单, 且其性能瓶颈不在于过滤操作. Q8 的瓶颈在于连接操作, Q13 则在于聚合操作, 因此 JIT 编译相对于解释执行的优势不明显. 相反地, 查询 Q6 和 Q21 由于其性能瓶颈在过滤操作上, 因此在启用 JIT 编译后执行时间明显降低. 图 9 展示了查询 Q6 和 Q21 在解释执行与 JIT 编译执行下的 CPU 开销对比. 如图所示, 查询 Q6 的 CPU 开销从 35.14% 下降至 14.19%, 查询 Q21 的 CPU 开销则从 28.89% 下降至 10.29%. 编译执行尽管增加了一定的编译开销, 但查询的过滤操作 CPU 开销却大幅下降.

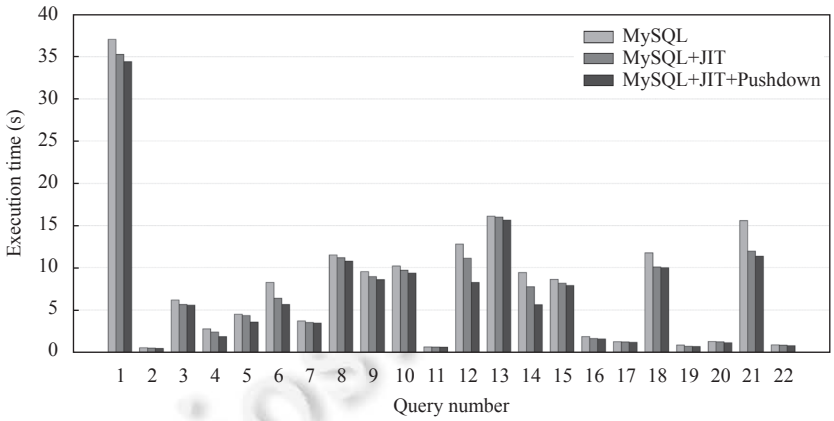

图 8 TPC-H 查询性能提升

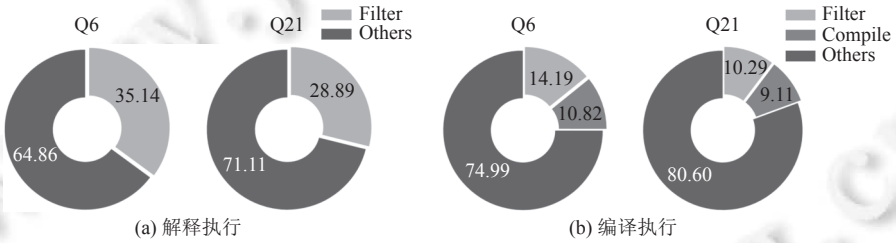

图 9 解释执行与编译执行 CPU 开销对比 (%)

进一步结合谓词下推技术, 我们观察到不同查询的表现有所不同. 尽管 Q8 和 Q13 在结合谓词下推后执行时间有一定的减少, 但由于其瓶颈仍不在过滤操作上, 减幅较小. 查询 Q1 由于其过滤条件的选择率非常高, 即使结合谓词下推, 也无法有效提前过滤数据, 导致执行时间的减幅不明显. 相比之下, 查询 Q4、Q12 和 Q14 不仅在过滤操作上存在明显的性能瓶颈, 且选择率较低, 通过谓词下推能够提前过滤大量数据, 执行时间得到显著的降低.

综上所述, JIT 编译系统在 TPC-H 基准测试中对复杂 OLAP 类查询的性能提升效果显著, 特别是在查询过滤操作成为性能瓶颈的情况下. 结合谓词下推技术, 系统进一步优化了查询性能, 尤其是对于那些具有低选择率过滤条件的查询, JIT 编译系统展现出了更大的优化潜力.

4.5.2 查询选择率对 JIT 编译的影响

本节重点分析在 JIT 编译结合谓词下推的场景下, 查询选择率对性能提升的影响. 为此, 我们选取了 3 个具有代表性的 TPC-H 查询, 分别是 Q1、Q4 和 Q8, 以全面评估选择率变化对 JIT 编译系统性能的具体影响.

如图 10 所示, 在启用 JIT 编译结合谓词下推功能后, 随着查询选择率的增加, 3 个查询的性能提升百分比均呈现出显著下降的趋势. 这一趋势表明, 当查询选择率较高时, 存储引擎层能够提前过滤的数据量减少, 性能提升更多地依赖于编译执行减少的函数调用开销, 而无法充分发挥谓词下推的优化优势.

在第 4.5.1 节的测试中, 启用谓词下推功能后, Q1 的性能提升为 7.64%. 当查询选择率降低至 1% 时, Q1 的性

能提升显著增加至 148%. 相比之下, 查询 Q8 的性能提升仅为 25%. 这种差异主要源于两个查询的性能瓶颈所在不同. 查询 Q1 的性能瓶颈主要集中在过滤操作上, 降低选择率使得存储引擎层能够有效地过滤大量数据, 从而显著减少上传到服务层的数据量和 I/O 开销. 查询 Q8 的性能瓶颈在于连接操作. 因此, 即使降低选择率, Q8 的性能提升仍然有限. 查询 Q4 的表现介于 Q1 和 Q8 之间, 进一步验证了选择率与性能提升之间的关系. Q4 的过滤条件具有中等的选择率和较高的谓词表达式开销, 因此在启用 JIT 编译结合谓词下推后, 能够实现中等程度的性能提升. 此外, 查询 Q13 虽然未在本次分析中详细讨论, 但其表现类似于 Q8, 因其性能瓶颈也不在于过滤操作上, 因此在启用 JIT 编译结合谓词下推后, 性能提升同样不显著.

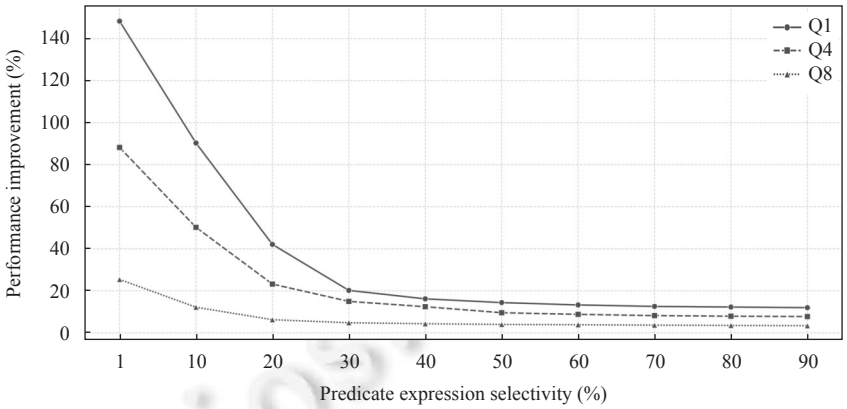

图 10 查询选择率对性能提升的影响

综上所述, 在 JIT 编译结合谓词下推的场景中, 查询选择率与谓词表达式的开销占比共同决定了性能提升的效果. 高选择率导致存储引擎层提前过滤数据的能力减弱, 性能提升主要依赖于编译执行减少的函数调用开销; 而低选择率则能够充分利用谓词下推技术, 提前过滤大量数据, 显著提升查询性能.

5 总 结

本文针对传统数据库迭代器执行模型的不足, 提出并实现了一种基于 JIT 编译的优化方案, 以提升数据库查询执行性能, 特别是在处理复杂查询和大数据量的场景. 通过将 SQL 查询中的谓词条件动态转换为机器码代替解释执行, 旨在减少虚函数调用和系统上下文切换的开销, 提高条件验证效率. 此外, 日益流行的可插拔数据库系统架构使得近数计算功能需求变得迫切. 本文将 JIT 编译与谓词下推结合的优化策略, 扩展了谓词下推的适用范围, 支持下推非索引条件, 并将 JIT 编译生成的机器码下推至存储引擎层进行提前过滤, 减少了不必要的数据传输和计算开销, 进一步提升了性能. 实验结果表明, 启用 JIT 编译后, MySQL 数据库在处理大数据量和复杂查询时性能显著提升, 尤其在降低 I/O 开销和优化资源利用方面.

未来研究将探索如何对聚合运算进行编译和下推, 并针对聚合操作的特性 (如分组、排序) 进行优化, 以进一步提升数据库性能. 其次, 我们还将探索如何实现高效的机器码缓存和复用机制, 减少重复查询场景中的编译开销, 确保在频繁执行的简单查询中能够有效平衡性能和成本.

References

[1] Padmanabhan S, Malkemus T, Jhingran A, Agarwal R. Block oriented processing of relational database operations in modern computer architectures. In: Proc. of the 17th Int'l Conf. on Data Engineering. Heidelberg: IEEE, 2001. 567–574. [doi: 10.1109/ICDE.2001.914871]

[2] Neumann T. Efficiently compiling efficient query plans for modern hardware. Proc. of the VLDB Endowment, 2011, 4(9): 539–550. [doi: 10.14778/2002938.2002940]

[3] Schulze R, Schreiber T, Yatsishin I, Dahimene R, Milovidov A. ClickHouse-lightning fast analytics for everyone. Proc. of the VLDB Endowment, 2024, 17(12): 3731–3744. [doi: 10.14778/3685800.3685802]

[4] Paek Y, Ahn M, Cho D, Kim T. Efficient embedded code generation with multiple load/store instructions. Software: Practice and Experience, 2007, 37(11): 1133–1159. [doi: 10.1002/spe.801]

[5] Butterstein D, Grust T. Precision performance surgery for CostgreSQL: LLVM-based expression compilation, just in time. Proc. of the VLDB Endowment, 2016, 9(13): 1517–1520. [doi: 10.14778/3007263.3007298]

[6] Pantilimonov M, Buchatskiy R, Zhuykov R, Sharygin E, Melnik D. Machine code caching in PostgreSQL query JIT-compiler. In: Proc. of the 2019 Ivannikov Memorial Workshop (IVMEM). Velikiy Novgorod: IEEE, 2019. 18–25. [doi: 10.1109/IVMEM.2019.00009]

[7] Datta D, Stoodley M, Ray S. Towards just-in-time compilation of SQL queries with OMR JitBuilder. In: Proc. of the 31st Annual Int'l Conf. on Computer Science and Software Engineering. Toronto: IBM, 2021. 256–261.

[8] Tetzel F, Lehner W, Kasperovics R. Efficient compilation of regular path queries. Datenbank-Spektrum, 2020, 20(3): 243–259. [doi: 10.1007/s13222-020-00353-9]

[9] Letkowski J. Doing database design with MySQL. Journal of Technology Research, 2014, 6: 1–15.

[10] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. of the 2004 Int'l Symp. on Code Generation and Optimization. San Jose: IEEE, 2004. 75–86. [doi: 10.1109/CGO.2004.1281665]

[11] Barata M, Bernardino J, Furtado P. An overview of decision support benchmarks: TPC-DS, TPC-H and SSB. In: New Contributions in Information Systems and Technologies, Vol. 1. Cham: Springer, 2015. 619–628. [doi: 10.1007/978-3-319-16486-1_61]

[12] Krikellas K, Viglas SD, Cintra M. Generating code for holistic query evaluation. In: Proc. of the 26th IEEE Int'l Conf. on Data Engineering (ICDE 2010). Long Beach: IEEE, 2010. 613–624. [doi: 10.1109/ICDE.2010.5447892]

[13] HyPer—A hybrid OLTP&OLAP high performance DBMS. 2025. https://hyper-db.de/

[14] Kohn A, Leis V, Neumann T. Making compiling query engines practical. IEEE Trans. on Knowledge and Data Engineering, 2021, 33(2): 597–612. [doi: 10.1109/TKDE.2019.2905235]

[15] Kersten T, Leis V, Neumann T. Tidy tuples and flying start: Fast compilation and fast execution of relational queries in Umbra. The VLDB Journal, 2021, 30(5): 883–905. [doi: 10.1007/s00778-020-00643-4]

[16] Funke H, Mühlig J, Teubner J. Low-latency query compilation. The VLDB Journal, 2022, 31(6): 1171–1184. [doi: 10.1007/s00778-022-00741-5]

[17] PostgreSQL: The World's most advanced open source relational database. 2025. https://www.postgresql.org/

[18] Manegold S, Kersten ML, Boncz P. Database architecture evolution: Mammals flourished long before dinosaurs became extinct. Proc. of the VLDB Endowment, 2009, 2(2): 1648–1653. [doi: 10.14778/1687553.1687618]

[19] Grulich PM, Lepping AP, Nugroho DPA, Pandey V, del Monte B, Zeuch S, Markl V. Query compilation without regrets. Proc. of the ACM on Management of Data, 2024, 2(3): 165. [doi: 10.1145/3654968]

[20] Haffner I, Dittrich J. A simplified architecture for fast, adaptive compilation and execution of SQL queries. In: Proc. of the 26th Int'l Conf. on Extending Database Technology (EDBT). 2023. 1–13.

[21] Lin S, Marathe AP, Larson PÅ, Chen C, Sun C, Lee P, Yu WD, Li JW, Meng JC, Lin RL, Chen XY, Zhu QP. Near data processing in Taurus database. In: Proc. of the 38th IEEE Int'l Conf. on Data Engineering (ICDE). Kuala Lumpur: IEEE, 2022. 1662–1674. [doi: 10.1109/ICDE53745.2022.00170]

[22] Amazon Aurora. 2025. https://aws.amazon.com/cn/rds/aurora/

[23] Yan C, Lin Y, He YY. Predicate pushdown for data science pipelines. Proc. of the ACM on Management of Data, 2023, 1(2): 136. [doi: 10.1145/3589281]

作者简介

袁巩生, 博士, 副研究员, CCF 专业会员, 主要研究领域为数据库, 量子计算, 数据挖掘.

杜晨路, 硕士, 主要研究领域为数据库.

朱阅岸, 博士, CCF 专业会员, 主要研究领域为高性能数据库系统, 大数据技术.

陈育兴, 博士, 腾讯云数据库高级研究员, CCF 专业会员, 主要研究领域为数据库事务处理和存储优化.

唐秀, 博士, "百人计划"研究员, 博士生导师, CCF 专业会员, 主要研究领域为数据库查询优化, 数据库测试, 数据智能.

姚畅, 博士, 研究员, 博士生导师, CCF 专业会员, 主要研究领域为数据库, 自然语言处理, 医疗信息技术, 医疗大模型.

陈刚, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为数据库系统, 大数据技术, 数据智能计算.