计 算 机 研 究 与 发 展
Journal of Computer Research and Development

DOI: 10.7544/issn1000-1239.202330507
61(7): 1754−1770, 2024

基于查询编译的 SQL 执行技术研究进展

潘青峰[1] 徐 辰[1,2]

[1](华东师范大学数据科学与工程学院 上海 200062)

[2](上海市大数据管理系统工程研究中心(华东师范大学) 上海 200062)

(qfpan@stu.ecnu.edu.cn)

Advances in SQL Execution Techniques Based on Query Compilation

Pan Qingfeng[1] and Xu Chen[1,2]

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

[2] (*Shanghai Engineering Research Center of Big Data Management System* (*East China Normal University*), *Shanghai* 200062)

Abstract Information systems usually use data management systems to manage data, among which SQL has been the mainstream query language for data management because of its ease of use and flexibility, and users can write SQL statements and submit them to the data management system to get query results. The efficiency of the execution model determines whether the system can quickly respond to user queries. The existing execution models mainly adopt interpreted execution and compiled execution. Interpreted execution is used by most systems due to its scalability and maintainability. Unlike interpreted execution, compiled execution generates efficient custom code to speed up queries that should have been processed by interpreted execution, and the significant performance gains have attracted a number of database systems to implement the technology. However, generating the corresponding custom code for a query is a complex process that requires a number of considerations, even in some cases, the performance of using compiled execution may not be as good as the traditional volcano model. We provide a systematic review of the progress of compiled execution techniques from conceptual and technical perspectives. Firstly, we outline the basic concepts of query compilation and introduce the relevant terminology and background knowledge. Secondly, we introduce the relevant techniques from three perspectives: intermediate code generation, intermediate representation, machine code generation and running. Finally, we look at the future development direction of compiled execution technology in the context of current research trends in data management systems and recent research work.

Key words data management system; query execution; code generation; compiler; just-in-time compilation

摘 要 信息系统通常会借助数据管理系统来进行数据管理,其中 SQL 凭借良好的易用性和灵活性一直作为数据管理的主流查询语言,用户将编写的 SQL 语句交由数据管理系统执行后便可得到查询结果.执行模型的高效与否决定了系统能否快速响应用户的查询请求,现有执行模型主要采用解释执行和编译执行 2 种方式.解释执行具有良好的拓展性、可维护性等因而被大多数系统采用.不同于解释执行,编译执行为原本需要解释执行的查询生成高效的定制化代码来加速查询,带来的显著性能提升吸引了一众数据管理系统开始实现相应技术.然而,如何针对查询生成其对应的定制化代码是一个复杂的过程,在实现时需要考虑诸多方面,甚至在某些情况下,采用编译执行的查询性能可能还不及传统的火山模型.从概念、技术等角度系统地综述了编译执行技术的研究进展.首先,概述了编译执行的基本概念,对相关术语和背

收稿日期: 2023-06-15; **修回日期**: 2024-01-03

基金项目: 上海市自然科学基金项目(23ZR1419900)

This work was supported by the Natural Science Foundation of Shanghai(23ZR1419900).

通信作者: 徐辰(cxu@dase.ecnu.edu.cn)

景知识进行了介绍；其次，分别从中间代码生成、中间表示、机器码生成与运行3个角度介绍了相关技术；最后，结合当前数据管理系统的研究趋势以及近期研究工作展望了编译执行未来的发展方向.

关键词 数据管理系统；查询执行；代码生成；编译器；即时编译

中图法分类号 TP311.13

数据作为当今时代的重要能源，加速了传统产业的转型升级，推动着新型产业如生成式AI的蓬勃发展. 如何有效管理海量数据，充分挖掘数据中的潜藏价值，从而支撑各行业信息系统的精准服务是数字化转型时期的关键问题. 信息系统通常会借助数据管理系统来进行数据管理，其中，SQL凭借良好的易用性和灵活性一直作为数据管理的主流查询语言. 用户通过编写简单的SQL语句，便能够在存储的数据上高效地进行查询与分析. 用户在编写完SQL语句后，需要将其交给对应的数据管理系统，系统会对输入的SQL语句进行解析、优化和执行，最终返回结果给用户. 在上述过程中，最为耗时的环节往往是查询执行，因此执行模型对于整个系统性能有着举足轻重的影响，其高效与否决定了系统能否快速响应用户的查询请求，从而满足信息系统精准服务的需求.

在早期的数据管理系统中，火山模型由于具有良好的拓展性、可维护性而被广泛采用，如SQL Server, PostgreSQL. 然而，火山模型在执行过程中存在大量虚函数调用、元组变形（tuple deforming）等处理过程，存在高额的解释开销. 在早期磁盘I/O为主要性能瓶颈的时代背景下，解释开销对查询性能的影响并不显著. 但在面向如内存数据库中的查询等CPU计算为性能瓶颈的负载时，火山模型已经无法满足查询所需的高吞吐和低延迟需求.

为了解决火山模型性能不佳的问题，现有数据管理系统采用2种策略来提高查询执行效率，即向量化和查询编译. 二者是针对火山模型的不同优化思路，存在不同的优势和弱势[1]. 本文聚焦于综述查询编译的相关技术. 其中，向量化采用和火山模型类似的解释方式，反复调用查询计划中各算子的迭代器，以“一次一批”的形式来均摊虚函数调用等带来的开销，如MonetDB/X100[2], ClickHouse[3]等，与火山模型都属于解释执行. 查询编译则通过为查询即时生成相应执行代码来避免传统火山模型中高额的解释开销，又称编译执行. 生成的代码在运行时避免了大量的函数调用并且能够高效利用处理器，因而提升了查询执行性能. 值得注意的是，在查询编译技术诞生之前，查询编译一词通常指将SQL查询转换为物理执行计划[4]. 随着编译执行技术的普及，大多数文献以及本文将该词特指为为物理执行计划生成对应的可执行文件.

近些年来，查询编译技术飞速发展. 为了满足各种需求，关于查询编译的研究不断推陈出新. 与此同时，越来越多的工业系统也开始将相关技术融入到自身执行引擎中. 查询编译发展至今已有40余年，其思想的出现可以追溯至System R[5]时期. 如图1所示，本文将查询编译的发展历程划分为3个时期：

1）萌芽期. 以System R中编译思想的出现为标志，查询编译的雏形悄然诞生. 随后在I/O为性能瓶颈的背景下，其并未广泛研究，一直缓慢发展. 该时期相关技术的主要目标都是降低解释执行的开销，同时构建查询编译系统相关的基础组件. 期间诞生了诸如HIQUE, JAMDB等查询编译数据库系统原型，为随后相关技术的发展奠定了基础.

2）发展期. Hyper[6]中以数据为中心的代码生成的出现标志着查询编译迈入发展阶段，该思想被随后的查询编译器广泛采用. 该时期研究工作主要方向转变为提升查询编译器的扩展性，包括优化架构和支持新硬件等. 期间诞生了LegoBase[7], LB2[8]等数据库系统原型，对随后查询编译器的架构产生了重要指导作用.

3）成熟期. 2015年左右，Spark SQL等工业系统逐步开始采用查询编译相关技术，标志着查询编译步入成熟期. PostgreSQL[9], OceanBase[10], openGauss[11]等数据库系统纷纷开始支持表达式上的代码生成，图1中的虚线方框表明该系统仅支持编译表达式等简单对象. Spark SQL[12-13], SingleStore[14]等系统则面向整个查询进行代码生成. 相关技术在工业界中的落地促进了查询编译基础组件的完善与成熟，但与此同时，系统的低延迟需求、融合机器学习的复杂分析的出现推动着该技术仍在不断发展. 近些年来一些研究工作因而转向支持复杂查询的性能需求等，如Weld[15], Umbra[16]等系统通过构建自定义中间表示来支持更高效的查询.

虽然查询编译核心技术已经趋于成熟，但是在具体实现时仍有诸多方面需要考虑，包括如何构建高效的代码生成器、如何选择合适的中间表示、如

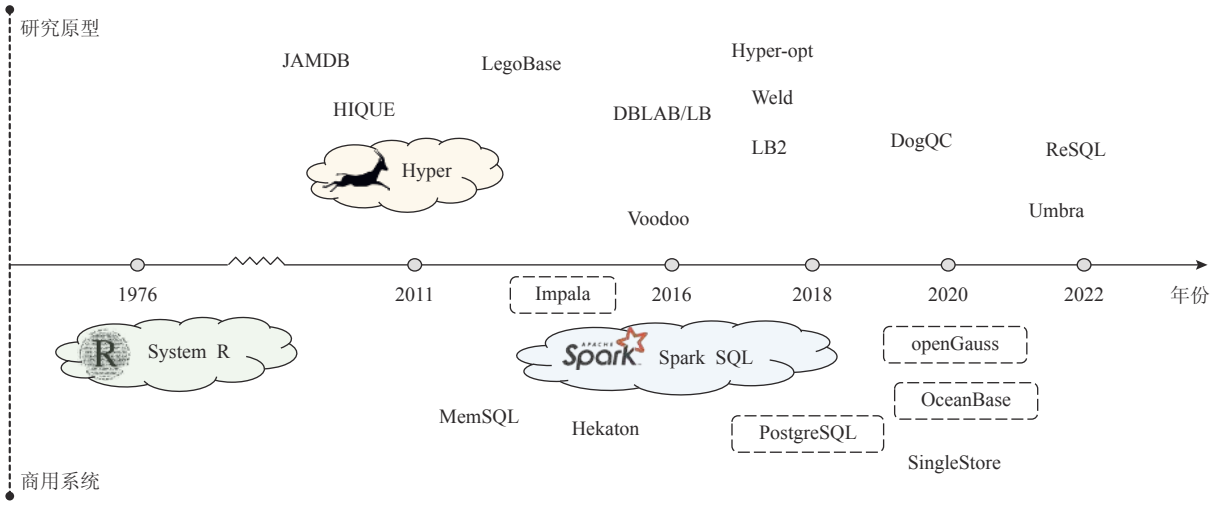


Fig. 1 History of query compilation development

图 1 查询编译发展历程

何避免额外的编译开销等，这些对于整个查询编译性能而言都有着至关重要的影响. 已经有相关综述文献对查询编译技术进行了介绍，文献 [17] 主要聚焦在单个系统上，文献 [18] 则综述了萌芽期的研究工作. 本文从查询编译的基本概念出发，按照 SQL 查询编译的执行流程综述了已有的查询编译技术，为相关研究人员提供参考.

1 查询编译概述

查询执行模型主要采用解释执行与编译执行 2 种方式，其中编译执行又可称为查询编译. 解释执行具有良好的拓展性和可维护性，且采用向量化后也能兼具高效的处理性能，因而被大多数数据管理系统采用. 查询编译因为能够减少查询执行时所需的指令数量，更适合于缓存常驻型负载[1]，也得到越来越多系统的关注与青睐. 二者同样作为系统的执行模块，如图 2 所示，既有相似部分，又有本质上不同的设计思想. 本节通过对比解释执行来介绍查询编译的基本概念，随后分析了查询编译的研究挑战和技术分类.

1.1 查询编译概念与原理

不同文献对于解释执行和编译执行的概念定义有所不同，为了更清晰准确地介绍相关技术，本文综合不同文献后，首先对二者进行如下定义. 解释执行是指以物理执行计划为输入，根据计划中算子间的调用关系连续地调用各算子预编译成可执行文件的通用算子计算单元来解释查询计划，查询执行过程

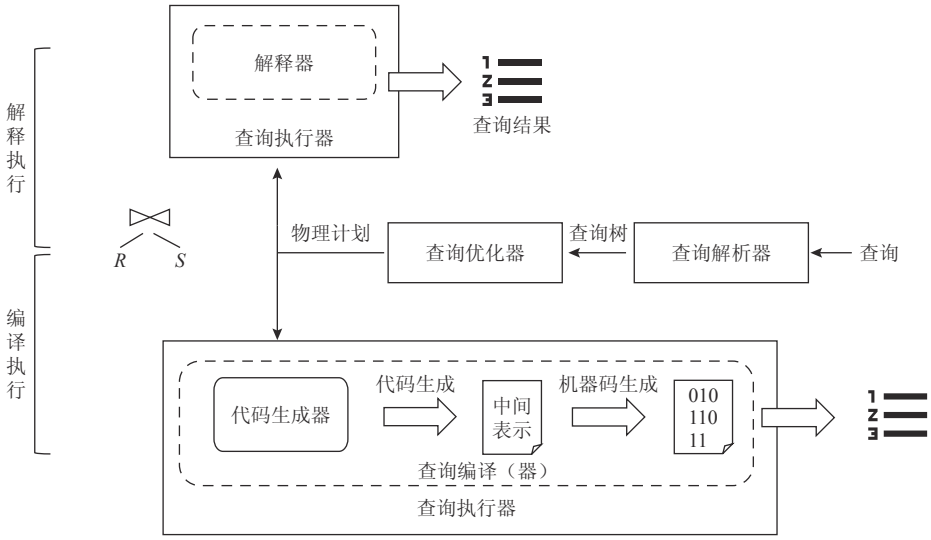
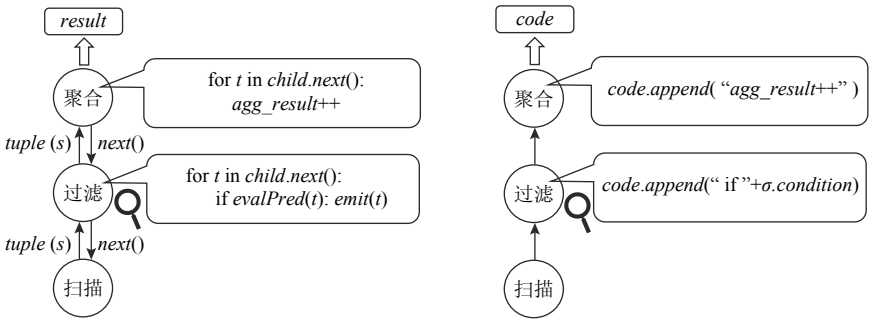


Fig. 2 Query processing

图 2 查询处理流程

中不会生成任何新的可执行文件. 编译执行是指以物理执行计划为输入, 根据计划中算子间调用关系依次调用各算子的代码生成单元, 为计划生成包含运算符语义、控制逻辑等的定制化代码, 执行时需将该代码即时编译为可执行文件.

在解释执行过程中, 上下游算子间通过调用彼此的 *next* 虚函数以“一次一元组”或“一次一批”的形式进行元组传递. 各个算子在获取到元组后, 根据预先编译好的可执行文件中的计算逻辑对其进行处理, 如图 3(a)所示. 这种抽象与封装降低了研发人员在开发系统时的复杂度, 也使执行模型具有良好的拓展性和可维护性.



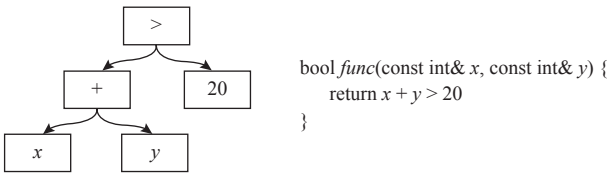
(a) 算子计算单元 (b) 代码生成单元

Fig. 3 Operator calls under two execution models

图 3 2 种执行方式下的算子调用

然而, 传统火山模型中存在的虚函数调用、元组变形等处理过程本身存在一定开销, 导致 CPU 无法被充分利用. 当待处理元组较多时, 频繁的虚函数调用等会显著影响查询性能. 虚函数调用本质是上下游算子间的逐级访问, 元组变形则是按照关系表中的字段类型等将元组转换成内存中的表示, 而这类信息在查询执行前是能够通过查询计划树结构和关系表结构等内容确定的. 查询编译因而结合此类信息为查询即时生成定制化的中间代码来避免上述开销, 其处理过程如图 2 下方部分所示, 分为代码生成阶段和机器码生成阶段. 在生成中间代码时, 查询编译采用的也是与解释执行类似的树形结构, 如图 3(b)所示. 不同的是调用每个算子时不再是直接计算, 而是以字符串拼接方式输出对应的逻辑代码, 包括计算和控制逻辑等, 所以在查询编译中查询执行代码是即时生成的.

以表达式计算为例, 在解释执行中, 用户输入的表达式通常被表示成如图 4(a)所示的树形结构. 树中诸如“>”等各种类型表达式的计算, 都是通过覆盖基类中的虚函数实现的, 在计算时又会递归地调用子表达式. 然而, 这些表达式的计算逻辑大多非常简单, 如 2 个数字相加减, 这导致调用虚函数等的开销甚至超过了计算本身的开销. 如果为用户输入的表达式生成如图 4(b)所示的调用函数并对其进行内联, 便可以避免虚函数调用开销. 同时, 内联也可以使得通用编译器进一步采用子表达式消除等优化手段, 进一步提高计算性能. 以 SELECT COUNT(*) FROM *tbl* WHERE $(x+y)>20$ 这条查询为例, 在 PostgreSQL 中, 表达式 $(x+y)>20$ 的计算在解释执行下占据了 56% 的执行时间. 而在编译执行下, 表达式计算仅为查询总执行时间的 6%, 整条查询的执行速度也提高了 2 倍. 对于更为复杂的查询, 如包含了连接等操作的 TPC-H1, 查询编译能给 PostgreSQL 带来 5 倍的性能提升. 对于包含了多表连接、聚合等复杂操作的 TPC-H3, 查询编译也能带来 2 倍的性能提升[19]. 但对于需要访问较大哈希表的查询而言, 采用向量化可能会比采用查询编译具有更快的执行速度[1].



(a) 表达式树 (b) 代码生成

Fig. 4 Expression evaluation under two execution models

图 4 2 种执行方式下的表达式计算

上述表达式计算的例子属于查询编译的简单应用, 总的来说, 查询编译通过代码生成器将用户输入的查询转换成高效且定制化的中间代码, 包括定制化的数据结构、执行逻辑等. 该代码随后被 GCC 等通用编译器编译成可执行文件并交由运行时运行, 运行时由于避免了原先火山模型中大量的函数调用等开销, 因而提高了查询执行性能, 甚至能够媲美专

业人员手写代码的性能. 如图 5 所示, 本文按照上述执行流程将中间代码生成、中间表示(intermediate representation, IR)、机器码生成与运行这 3 个层面的相关技术独立出来, 分别介绍各自的研究挑战与解决方案.

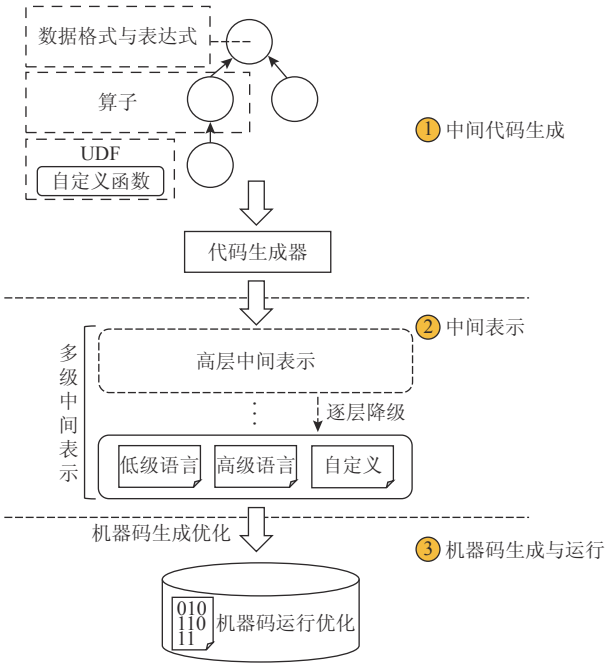


Fig. 5 Illustration of query compilation execution

图 5 查询编译执行示意图

1.2 研究挑战

本节从中间代码生成、中间表示、机器码生成与运行 3 个层面分析了查询编译引擎在设计与实现上的研究挑战. 代码生成、机器码生成与运行的研究挑战集中在如何提升系统吞吐或降低系统延迟. 中间表示作为承上启下的核心模块, 研究挑战在于如何提高系统的可维护性与拓展性等.

1)代码生成层面. 如何生成执行速度接近最优手写代码的中间代码, 同时保证代码生成器的通用性. 代码执行速度与代码逻辑息息相关, 手动地为查询编写高效的代码逻辑本身并非易事, 如何使生成的代码接近手写代码的执行速度则更加复杂. 此外, 生成的代码越高效, 对应的代码生成器的结构通常更为复杂, 以至于不适用于其他系统. 因此如何兼具良好的通用性也是构建代码生成器需要解决的关键问题.

2)中间表示层面. 如何在可接受的编译延迟范围内, 尽可能提高中间表示的表达能力. 中间表示作为查询编译的核心模块, 需要兼具高效的编译速度和良好的表达能力. 高效的编译速度是指生成的中间代码能够迅速编译出机器码, 以此降低查询编译的延迟. 良好的表达能力则意味着基于该中间表示, 能够方便地生成高质量的中间代码. 然而, 提升表达能力通常需要构建结构更复杂的中间表示, 而此类中间表示又会增大编译延迟. 因此, 在构建中间表示时需要考虑到二者间的制约关系.

3)机器码生成与运行层面. 如何在减小机器码生成带来的额外开销的同时, 尽量不影响甚至提升随后代码运行时的性能. 在大多数情况下, 查询编译能够给查询执行带来显著的性能提升. 但查询编译的一些特性产生的额外开销, 如即时将中间代码编译成机器码引入的机器码生成开销, 在某些情况下反倒会导致查询的整体性能不及传统的火山模型. 机器码生成的开销主要来自于优化过程, 但如果通过减少优化步骤来降低机器码生成开销又会影响代码运行时的性能. 如何兼顾快速的机器码生成以及随后高效的代码运行, 对于系统保证低延迟和高吞吐至关重要.

1.3 相关技术分类

如图 5 所示, 本节从 3 个层面归纳了查询编译的相关技术.

1)代码生成. 为了生成高效中间代码, 不同数据管理系统会选择对查询中的表达式、算子以及用户自定义函数等部分进行代码生成. 此外, 现有系统大多采用模板技术的变体生产与消费框架来构建自身的代码生成器, 以此保证生成器的通用性.

2)中间表示. 通用编程语言领域中的各类语言或表示如 C++, LLVM IR 等, 本身具有一定的表达能力, 因此大部分系统会选择在此类通用编程语言或表示上构建单层的中间表示. 然而基于这种架构的中间表示的表达能力较弱, 部分工作通过构建自定义表示在降低编译延迟的同时, 提高中间表示的表达能力. 另一部分系统则在可接受的延迟范围内, 通过构建多层中间表示来提高中间表示的表达能力.

3)机器码生成与运行. 为了在降低机器码生成开销的同时保证随后代码运行时的性能, 现有工作从机器码的生成和运行 2 个阶段入手进行优化. 针对机器码生成的优化聚焦于通过代码缓存、编译器化简等技术来降低机器码生成的开销, 并尽可能不影响代码运行性能. 针对机器码运行的优化则关注于在运行阶段通过通用或特定优化技术来提升代码运行时的性能.

总而言之, 查询编译给查询执行带来新机遇的同时也引入了一些新的挑战. 接下来的 3 节将分别从

代码生成、中间表示以及机器码生成与运行 3 个核心阶段入手，系统地介绍相关技术.

2 代码生成

代码生成阶段的核心内容是通过代码生成器将优化后的整个或部分物理执行计划转换成高效、紧凑的定制化代码. 其中存在 2 个关键问题：一是面向各类计划节点如何生成性能最优的中间代码；二是针对整棵计划树，如何设计代码生成器框架，使其能够像解释执行中的火山模型一样具有良好的通用性.

如图 6 所示，物理计划中包含元组变形与表达式、算子以及用户自定义函数 3 类计划节点. 针对每类节点，可构造的代码内容多种多样，能否使生成代码的性能接近最优对于系统实现高吞吐至关重要. 在确定各节点的代码生成策略后，需要通过一套框架将这些代码整合起来，进而生成完整的、逻辑正确的中间代码. 早期的一些系统通常采用简单的代码模板技术来构建代码生成器框架，但是这种方式实现的代码生成器往往缺乏清晰的架构，通用性不佳，因而难以拓展或不便于其他系统借鉴. 本节首先介绍了针对不同计划节点的代码生成相关工作，随后概述了面向整棵计划树如何构建代码生成器框架.

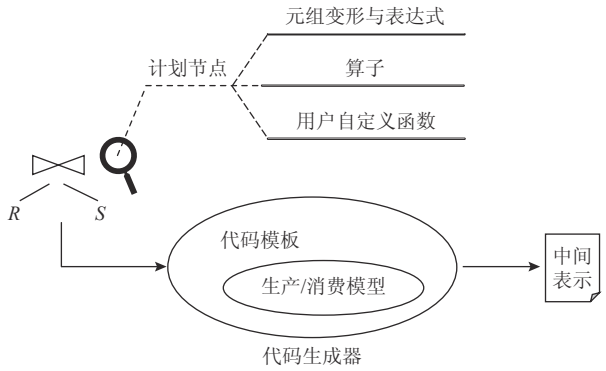


Fig. 6 Code generation workflow

图 6 代码生成工作流

2.1 面向计划节点的代码生成

一棵完整的物理计划树主要包括元组变形与表达式、关系算子以及用户自定义函数 3 类节点. 这 3 类节点在计算内容上差异较大，因此在代码生成时，需要根据节点类型来设计不同的代码生成策略.

2.1.1 元组变形与表达式

在算子或表达式计算前，需要先将待处理元组转换成内存中的表示. 类似于反序列化，该过程将元组的各个字段转换成相应数据类型如整型、布尔型等. 由于待处理字段的类型只有在用户向系统提交查询后才能得知，为了能够应对各种类型数据的转换，解释执行下的转换过程在设计与实现时往往非常通用. 然而，这种通用性导致元组的转换过程存在冗余指令，当待处理元组过多时，转换的高效与否就会直接影响查询的性能. 查询编译则根据输入查询的字段信息，即时生成特定的转换函数，如针对整型和浮点型字段生成特定的整型或浮点型转换函数. 该函数消除了运行时不必要的指令以及变量[20]，也可以进一步被内联，以此提高了查询性能.

表达式一般出现在选择、条件判断等子句中，如 1.1 节所述，其计算在解释执行模型中通过表达式树的递归调用实现. 但由于表达式的计算逻辑通常较为简单，这使得函数调用等成为了主要的性能开销. 编译表达式旨在通过代码生成消除这些冗余函数调用，并通过内联函数等进一步提高表达式的执行效率.

2.1.2 算 子

为表达式计算等进行代码生成虽然消除了运行过程中部分不必要的函数调用开销，但是并没有解决解释执行中算子间频繁的函数调用问题，所以需要进一步对算子计算进行代码生成. 现有工作采用的代码生成方式可以分为面向单算子和面向多算子 2 类. 前者孤立地为每个算子进行代码生成，未考虑算子间的联系；后者则打破了各算子边界，为多个关联的算子生成紧凑的执行代码.

面向单算子代码生成的 HIQUE[21] 通过在算子执行过程中内联物化结果来消除解释过程. 具体地，其通过类似 C++模板的代码模板技术，为查询中每个算子生成特定的执行代码，随后对多个算子的代码进行整合后，便形成了整个查询最终的执行代码. 该代码在运行时避免了大部分函数调用，减少了指令以及数据的访问，提高了缓存局部性，并且能够正常调用系统的其他模块如存储引擎等. 然而 HIQUE 并不适用于大规模的数据查询，因为当需要处理的数据量过大时，算子的中间结果可能无法全部放置于内存中，此时需要写盘操作，引入额外的开销. 此外，HIQUE 生成的代码中，各个算子是孤立的，这导致元组数据需要在各算子间传递，存在冗余的数据传递开销.

针对上述问题，Hyper[6] 提出的面向多算子（又称以数据为中心）的代码生成，根据流水线切分点（pipeline breaker）将查询切分成多条包含若干算子的流水线. 随后代码生成器会以流水线为单位，为其生成紧凑的执行代码，即流水线内部多个算子的计算

逻辑能够连续执行. 图 7 具体描述了以数据为中心的代码生成, 不同序号表示不同的流水线, 方框中的伪代码是该流水线对应的执行代码. 在每条流水线执行时, 元组首先被加载至 CPU 寄存器中, 只有在被流水线中所有算子逻辑处理完后才会物化至内存以供下一个流水线处理. 这种方式使得元组尽可能久地驻留在 CPU 寄存器中, 避免了原先算子间冗余的物化操作, 最大化数据局部性. 以数据为中心的查询编译技术显著提升了查询执行性能, 吸引了工业界和学术界中不少系统采用该方式来实现自己的查询编译引擎. 如 Spark SQL[12-13] 中的全阶段代码生成(whole-stage codegen)就是将多个物理算子融合到一个单独的 Java 函数中; 异构查询引擎 DogQC[22], πQC[23] 也是将多个算子逻辑封装至一个 CUDA 核函数内来进一步提高执行性能.

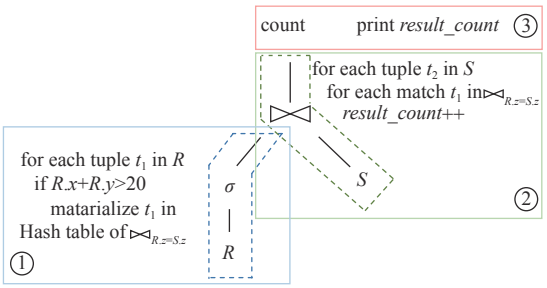


Fig. 7 Data-centric code generation

图 7 数据为中心的代码生成

2.1.3 用户自定义函数

用户自定义函数(以下简称函数)作为 SQL 的拓展组件, 通常被用来实现 SQL 难以表达的计算逻辑, 如数据挖掘、机器学习算法等. 在不同的应用场景下, 函数会借助不同语言来实现, 如常见的数据分析中, 函数会通过 Python 实现. 因此不少数据管理系统都支持 Python 的函数拓展, 以增强 SQL 的表达能力, 如 PostgreSQL 中的 PL/Python 等. 然而, 嵌有函数的查询在执行时需要横跨函数和 SQL 引擎这 2 个分离的运行环境, 进而导致 2 个问题: 一方面, 编写函数的语言的执行性能通常低于本地语言 C/C++等; 另一方面, 查询优化器将函数视为黑盒, 无法将函数与 SQL 进行全局优化, 且缺乏对函数本身的调优. 针对这 2 个问题, 现有工作主要通过通用编译器加速以及构建统一表示 2 种方式来解决.

动态语言 Python 性能往往不及 C/C++, 因而在通用编译器领域, 出现了不少能够加速 Python 执行的编译器, 如实现了即时编译成机器码的 PyPy、将 Python 转换成 C/C++的 Cython 等. 类似技术可以直接用来加速用户自定义函数执行, YeSQL[24] 利用 PyPy 构建了自身的即时编译器, Kläbe 等人[25] 则通过 Cython 提高了函数的执行性能.

为了打开函数这个黑盒, 不少工作通过将函数和 SQL 映射到统一表示上来进行全局优化. 根据表示抽象程度的不同, 现有技术可以分为基于高级统一表示的优化和基于低级统一表示的优化.

一些工作通过将函数转换成关系操作等高级表示来实现全局优化. Hirn 等人[26] 将用户定义的 PL/SQL 函数翻译成普通的 SQL 查询, 通过避免 PL/SQL 与 SQL 的上下文切换从而提升了查询性能. Froid[27] 将简单函数转换成关系算子, 统一了 SQL 与函数的运行环境. 不过这些工作底层的执行引擎仍是 PostgreSQL 或 SQL Server 等解释执行模型.

上述技术[26-27] 适用范围有限且难以应用更底层的优化, 因此另一些工作将 SQL 与函数映射到统一的低级表示上[15,28-30], 从而进行更深层次的全局优化. HorsePower[31] 将 SQL 查询和 MATLAB 函数编译到 HorseIR[28] 上, 基于编译器优化技术来进行全局优化. TUPLEWARE[32] 为函数生成对应的 LLVM IR, 通过分析 LLVM IR 来收集一些函数统计信息. 在代码生成阶段, 优化器会利用函数的统计信息来实施针对底层硬件的底层优化. 此外, Zhang 等人[33] 将文献 [34] 中自适应处理思想拓展至自定义函数中, 根据执行过程中动态变化的数据分布等, 自适应地调整执行代码以提高函数本身的执行速度.

2.2 面向计划树的代码生成

面向计划节点的代码生成主要聚焦于为每种类型的节点生成最优的执行代码, 但是代码生成器需要将上述代码生成策略整合起来, 并为整棵查询计划树生成一份完整的中间代码. 如何面向整棵计划树来设计一个通用的代码生成框架, 是代码生成器能否具有良好拓展性的关键所在. 现有工作本质上都是通过将查询计划中的操作映射到字符串形式的中间代码上, 随后通过拼接多段字符串实现的. 具体实现方式主要有 2 种, 分别是代码模版技术以及其变体, 即生产与消费(produce/consume)模型[6].

基于模板的代码生成类似于 C++中的模板技术, 通过将当前查询的模式信息代入到预先定义好的代码模板中, 经由模板产生对应的执行代码. 因为模板的内容需要预先定义, 因此该方式比较适用于语义相对固定的算子的代码生成. 采用此方式构建代码生成器的系统包括 HIQUE, Hekaton[35] 等, 但这种方式实现的代码生成器通用性不佳, 难以拓展.

为了提高代码生成器的通用性、降低其复杂度，Neumann 提出生产/消费模型. 该模型采用类似解释执行下迭代器的结构，每个算子都需要实现生产和消费 2 个函数. 其中，生产函数负责逻辑控制，消费函数负责输出算子运算逻辑对应的字符串形式，代码生成器通过往返遍历输入的物理执行计划来完成代码生成工作. 具体过程如图 8 所示，首先根节点会逐层向下调用子节点的生产函数，当遇到哈希表构建等流水线切分点时，会切断当前流水线并产生一条新流水线. 切分完成后会继续向下调用生产函数，直至传递到叶子节点，此过程会将查询计划树切分成若干流水线. 其次，叶子节点会逐层向上调用父节点的消费函数直至当前流水线的根节点，此过程则会为每一个流水线生成其对应的代码. 最后，将这些代码片段组合后就形成了整个查询的完整执行代码. 由于该方式是以流水线为单位进行代码生成，因而产出的执行代码更为紧凑，即一个元组上能连续执行的操作均被封装在 1 个代码块中，执行时也更为高效. 该模型属于模板技术的变体，针对的是以数据为中心的代码生成. 除 Hyper 外，Spark SQL, LingoDB[30], Grizzly[36] 等系统也采用这种模型来构建自己的代码生成器. 上述生产/消费模型将代码生成工作分散到生产和消费这 2 类函数中，在开发新算子时，需要对算子计算逻辑进行拆解，然后将各部分用这 2 类函数来实现，不利于代码生成器的拓展. 因而 Tahboub 等人[8] 基于 Scala 的特性和编译器拓展，采用第一二村映射（the first Futamura projection）的思想实现了一个类型安全、可读性高、架构清晰的查询编译器. Tahboub 等人[8] 首先构建了一个以数据为中心的解释器，并通过回调函数来实现算子间的通信，随后通过特化该解释器便能够得到代码生成器. 这种方式将代码生成工作集中在一个函数中，因此在增加新算子时，只需要实现 1 个函数即可，而且该方式也使得整个查询编译器能够同火山模型一样具有清晰的结构.

代码模板技术和生产/消费模型是构建代码生成器的常用框架，面向的是整个执行计划，并不是仅针对表达式、算子或是用户自定义函数中的某一种. 不同系统可以组合其中二者来使用，比如算子的代码生成通过生产/消费模型实现，内部表达式的代码生成则可以通过代码模板来实现. 用户自定义函数通常是由高级语言编写的，一般可以直接或间接（如通过通用编译器编译出中间代码后）地与其余中间代码进行拼接. 虽然自定义函数并没有按照上述 2 种模板方式来实现，但是从整个查询的角度来看，自定义函数的代码生成过程也属于基于模板或生产/消费模型的代码生成中的一个环节.

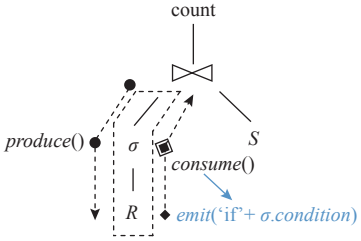


Fig. 8 produce/consume code generation framework

图 8 produce/consume 代码生成框架

2.3 总结与分析

诸如 Impala, HIQUE, Hyper 等系统关于如何构建一个高效且通用的代码生成器展开了深入研究，从最初仅针对简单表达式直到随后面向多个算子生成紧凑的执行代码，一步步实现了更高的系统吞吐. 在保证代码生成器的通用性方面，类似解释执行中迭代器结构的生产/消费模型也在逐步成为构建生成器时的参考标准. 值得注意的是，给系统实现代码生成技术时，是采用表达式等简单对象还是算子上的代码生成取决于性能收益和系统入侵程度的取舍. 元组变形和表达式计算过程较为简单，因此对这 2 类对象进行代码生成，实现上较为容易，对于整个系统的入侵以及改动也相对较小. 关系算子是数据管理系统中的核心对象，对其进行代码生成相当于重新构建基于查询编译的执行引擎，且算子级别的代码生成工作通常也实现了数据格式与表达式的代码生成，因而开发复杂度较高. 因此对于一些基于解释执行的数据管理系统来说，可通过实现元组变形和表达式计算的代码生成来加速查询执行. 此外，现有工作主要考虑的是一棵计划树上的代码生成，在多查询优化[37] 问题下，系统同时要对多条查询进行优化，查询编译能否为该问题带来新的优化机会值得探究.

3 中间表示

中间表示是整个查询编译器的核心模块，其作为代码生成器输出的中间代码的载体，需要准确高效地表达出查询内容；作为运行时的输入，要能够被快速地编译成机器码. 中间表示的设计需要从 2 个方面入手，包括中间表示的描述语言和中间表示的层级.

一般情况下代码生成器不会直接生成可读性较低的低层指令（汇编、机器指令），而是生成某种中间表示作为与低层指令之间的桥梁. 如图 9 所示，该中间表示可通过低级语言、高级语言等多种语言来描

述,因此首先要确定中间表示的描述语言.在确定好合适的语言后,若要提高中间表示的表达能力,需要考虑是否采用多层架构,但该架构会增大编译延迟,因而应当根据系统需求进行选择.本节首先介绍了不同语言的特点以及基于该语言构建中间表示的研究工作,随后深入分析了多级中间表示的相关工作.

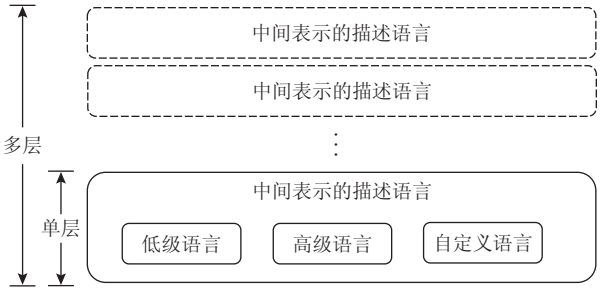


Fig. 9 Description languages and architecture of IR

图 9 中间表示的描述语言和架构

3.1 中间表示的描述语言

在编译器领域中,已有的语言通常在设计之初就已经综合考虑了延迟与表达能力,因此现有系统会结合自身上下文直接选择已有的语言作为中间表示.现有工作采用的语言主要包括 3 类:高级语言、低级语言以及自定义语言.其中,高级语言包括 C++, CUDA 等相对机器码而言抽象级别较高的通用程序语言;低级语言指抽象级别较低的通用表示或指令,如 LLVM IR、汇编等.其中低级语言的编译延迟较低,但表达能力不佳,不利于代码生成.因此,部分工作通过构建自定义语言来提高表达能力,同时进一步降低了编译延迟.以查询中的加法表达式为例,图 10 展示了 3 种不同语言下的中间表示.

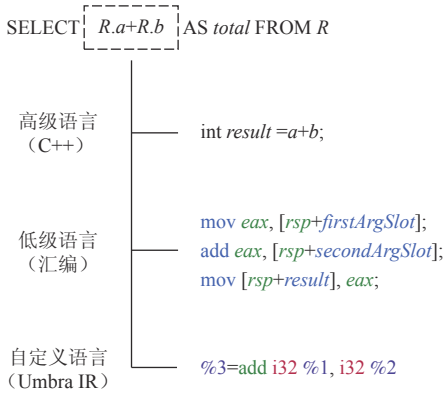


Fig. 10 Illustration of different IRs

图 10 不同中间表示示意图

3.1.1 高级语言

高级语言可读性更高,并且有社区不断维护与优化,因此大部分系统选择高级语言作为中间表示.不同系统根据自身所处的上下文,会选择不同的高级语言,包括:1)原生语言如 C++, CUDA 等;2)托管语言如 Java, C#等;3)混合语言.

1)原生语言

原生语言也可称为本地语言,指代码执行前需要被编译成本地机器码,执行速度快但无法跨平台,包括 C/C++等. C/C++凭借自身的高效性和可移植性等特点被不少系统选用.诸如 HIQUE, Hekaton, Amazon Redshift[38] 等均生成 SQL 对应的 C/C++代码,然后调用 GCC 或 Clang 编译成可执行文件. Hyper 最初的设计也是直接生成 C++代码,但考虑到 C++ 的编译器将 C++编译成机器码时速度不佳、无法完全控制生成的代码、溢出标志不可用等因素,最终放弃了完全使用 C++[6],转而使用 LLVM IR 作为目标代码.

2)托管语言

托管语言在执行前不需要被编译成机器码,而是直接交由虚拟机解释执行,执行速度相较于原生语言较慢,但支持跨平台.查询编译器在为中间表示选择语言时,除了考虑性能因素外,还需要兼顾数据管理系统本身所处的上下文. Spark SQL 底层执行引擎基于 Scala 实现, Scala 本身是交由 JVM 托管的,因此为查询生成 Java 或 JVM 字节码可以更高效地调用系统其他模块.值得注意的是, JVM 字节码的语义与 Java 的语法结构等有较为直接的对应关系,抽象级别较高,因此也归为高级语言的范畴.同时 JVM 提供的反射机制、即时编译等也有利于查询代码的高效调用和执行,因此内存数据库系统原型 JAMDB[39] 也基于 Java 来构建.

3)混合语言

部分工作会选择生成原生与托管语言混合的执行代码. LINQ[40] 查询将 SQL 语法集成到 C#语言中,使得用户可以在 C#语言中直接查询和操作各种数据. Nagel 等人[41] 通过代码生成的方式以提高 LINQ 查询处理的效率,生成的代码同时包含了 C#和 C.其中 C#用于对数据进行“分层”, C 则用来加速计算.之所以不将所有逻辑都交由 C 处理主要有 2 个原因:一是因为 C 可能无法访问部分 C#数据;二是对于一些简单查询如选择-投影而言,将数据交由 C 处理所需做的转换代价超过了用 C 计算带来的收益.

3.1.2 低级语言

相对高级语言而言,低级语言更加靠近机器语言、可读性较低、增加了代码生成器的开发复杂度等.但直接生成低级语言表示的执行代码,可以避免高级语言方式下存在的编译器前端开销,这 2 种语

言在具体编译流程中的区别如图 11 所示. 低级语言具体包括通用中间表示以及汇编.

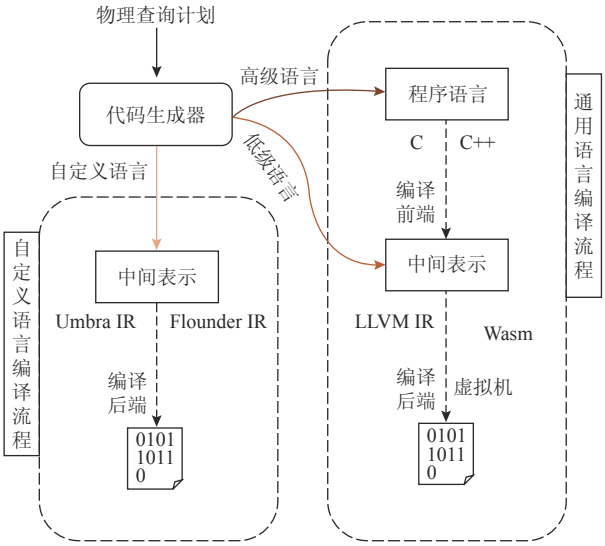


Fig. 11 Compilation process of three types of languages

图 11 3 种语言的编译流程

通用中间表示指通用编译器的中间表示, 如 LLVM IR, WebAssembly[42](简称 Wasm)等. LLVM IR 类似于汇编语言, 由一些简单指令组成. 为查询生成对应的 LLVM IR, 可以利用 LLVM 优化器对 IR 进行优化并降级至机器码. LLVM 框架也提供了一些能够提高开发效率的模块, 如辅助生成 IR 的 IRBuilder[43]等. 相较于 C++等高级语言, 更低级的 LLVM IR 能提供更快的机器码生成速度. 以 TPC-H 查询为例, LLVM IR 代码的机器码生成速度比 C++代码的机器码生成速度快了 2 个数量级左右[6]. 诸如 Impala, Single-Store[14], TUPLEWARE 等系统也都选用 LLVM IR 作为自己查询编译器的中间表示. Wasm 是一种基于堆栈虚拟机的二进制指令格式, 诞生于 Web 领域, 凭借紧凑、独立于硬件、执行时接近本地执行速度等特点, 受到了 Web 外其他领域的关注. Haffner 等人[44]为查询计划生成 Wasm 格式的中间代码, 并将其交由 V8 引擎[45]执行.

查询编译思想的出现最早可以追溯至 System R, System R 直接为 SQL 生成对应的汇编代码[5]. 由于生成汇编码步骤复杂, 且缺乏先进的编译器技术, 所以 System R 整体性能不佳. 此外, 因为生成的是汇编码, 所以需要为不同的操作系统或硬件平台实现不同的查询编译器, 这削减了编译器的移植性和可维护性. 因此, IBM 研发 DB2 时便放弃了该技术.

3.1.3 自定义语言

低级语言虽然避免了编译前端的开销, 但表达能力不佳. 具体而言, 通过低级语言来描述数据库的特定语义操作时更为复杂, 当中间表示由多种语言组成时, 难以进行多语言的协同优化. 此外, 低级语言被编译成机器码的过程中仍存在延迟. 因此, 一些工作通过设计并实现自定义语言来解决上述问题. 从抽象层次角度来看, 自定义语言更加靠近低级语言如 LLVM IR. 具体可以分为基于独立编译栈的自定义语言和基于开源编译器框架的自定义语言.

部分工作通过独立地实现一套编译器框架, 包括中间表示和编译后端等, 从而提高了中间表示的表达能力. Weld 通过统一的中间表示 Weld IR, 实现了多系统间的深度优化, 避免了冗余的数据传输等. Umbra[16]针对数据库领域设计并实现了中间表示 Umbra IR, 包括特定的数据结构以及指令等. Umbra IR 在设计过程中能够实现高度的灵活性, 在提高表达能力的同时也降低了编译延迟. 此外, SQLite[46]虽然生成的是也自己定义的字节码, 但这些字节码本质上还是交给由 C 编写的虚拟数据库引擎(virtual database engine, VDBE)去解释执行.

另一些工作通过利用编译器领域的技术框架来构建统一的中间表示, 以此实现协同优化. LingoDB 借助多层中间表示(multi-level intermediate representation, MLIR)框架[47], 利用 MLIR 提供的方言实现了自己的中间表示, 可支持数据处理和线性回归等模型的协同优化.

表 1 从编译速度、执行速度以及复杂度对比了不同中间表示下的代表性语言, 其中编译速度指生成机器码的速度, 复杂度综合考量了表达能力、实现成本等. 表达能力越低, 实现成本越高, 则该语言的复杂度越高. 值得注意的是, 以下对比针对 x86 架构, 且原生语言 C 以及 LLVM IR 均开启编译优化.

Table 1 Features Comparison of Different IRs

表 1 不同中间表示的特点对比

类别	中间表示	编译速度	执行速度	复杂度
高级语言	C	慢	快	较低
	Java		慢	低
低级语言	LLVM IR	较慢	快	较高
自定义语言	Umbra IR	快	较快	高

编译速度层面, LLVM IR 等中间表示避免了 C 等原生语言存在的编译前端开销, 因而机器码生成得更快. 自定义语言由于构建了特定编译栈, 具有更简洁的编译流程, 因而编译速度最快. Java 的编译通常是指编译成字节码, 故不考虑 Java 的编译速度.

执行速度层面, Java 等托管语言交由虚拟机解释执行, 即使开启 Java 的即时编译优化, 执行速度也不及原生语言和 LLVM IR. C 和 LLVM IR 所处的抽象级别不同, 但均采用 LLVM 框架编译与优化, 生成的机器码质量差别不大, 因而都具有高效的执行速度. 自定义语言由于精简了编译流程, 减少了通用编译器中的优化环节, 因而执行速度不及原生语言 C 和 LLVM IR.

复杂度层面, 中间表示的表达能力越强越有利于代码生成, 且高级语言 C 的机器码生成完全交由通用编译器完成, 因此通常具有较低的复杂度. LLVM IR 等表达能力更低, 大多中间表示以静态单一赋值(static single assignment, SSA)形式给出, 并且与编译器框架的交互更复杂[48], 这些都增加了代码生成的复杂性. 自定义语言虽然表达能力强, 但需要设计并实现特定的中间表示以及针对不同架构的编译后端等, 需要对编译器结构、机器架构等有深入了解, 因而复杂度最高.

3.2 中间表示的层级

初期以 Hyper 为代表的查询编译器主要采用单层架构, 表达能力不佳且难以维护. 部分工作提出在可接受的编译延迟范围内, 通过多级架构的中间表示来解决上述问题.

3.2.1 单层架构

基于单层架构所构建的查询编译器直接将查询计划翻译成某种中间表示如 C++, 该中间表示随后经 GCC 等编译器编译后生成可执行文件. 在此架构下, 流水线划分、算法实现、数据结构定义(如哈希表、列表)等逻辑模块全部封装在一层中实现. 这种耦合方式违背了软件设计的原则, 导致系统可维护性、可拓展性以及灵活性等受限; 而且多种模块之间可能会产生组合爆炸的问题, 导致代码库代码爆炸[49].

3.2.2 多层架构

Klonatos 等人[7]指出基于单层架构所构建的查询编译器难以实现且错失了优化机会, 存在灵活性差、难以维护等缺陷. 因此 Shaikhha 等人[49]提出使用多级中间表示架构来构建查询编译器, 通过将原先单层中间表示处理内容解耦到多级中间表示上, 以此解决上述问题. 采用多级架构的 DBLAB/LB 按照类似图 12 的模式构建, IR_1 至 IR_N 的抽象级别由高到低, 各 *IR* 负责不同的转换工作, 如哈希表特化、列表特化等. DBLAB/LB 首先将查询计划翻译为声明式的高级 IR_1, 然后将此 IR_1 逐级降级到更低级的 IR_2, 直至翻译成最底层的 IR_N. 这种多层抽象与逐级优化的模式提高了查询编译器的优化能力, 降低了整体的复杂性, 也影响了随后其他数据管理系统查询编译器的开发, 如 LingoDB, Umbra 等. 多级中间表示中通常需要实现自定义的高层中间表示, 该中间表示的抽象层次一般高于 C/C++等高级语言, 如 Excalibur[50]中的 VOILA[51]、Hekaton 中的 MAT 等. 这类语言更贴近 SQL 层, 因而便于进行高层次的优化. 代码生成时, 这类语言作为和 C/C++之间的中间层, 也增加了代码生成的灵活性.

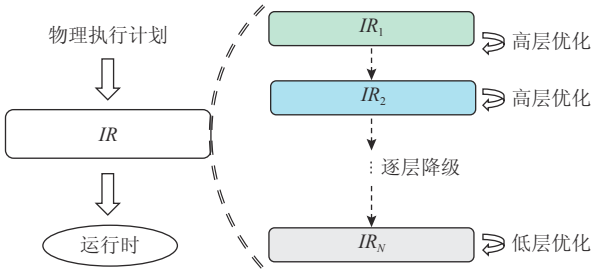


Fig. 12 Architecture of multiple layer compiler

图 12 多层编译器架构

虽然多层架构避免了单层架构的局限性, 但要求研发人员必须花费精力去实现多种中间表示和对应的降级逻辑, 且多层降级过程也增大了代码生成延迟. Jungmair 等人[30]也指出开发采用多级架构编译器往往需要投入大量人力去实现一些重复的模块, 如类型转换、与低级编译器框架的交互等. 这一方面造成了资源的浪费; 另一方面, 不同系统的查询编译器之间的隔离性导致很难将其中一个的技术在另一个编译器中应用或验证. 因此, Jungmair 等人[30]提出基于 MLIR 框架来构建查询编译器. 开源的 MLIR 提供了统一的中间表示框架, 通过方言[52]可以在同一中间表示框架上构建出不同抽象层次的中间表示(如高级的执行图、低级的汇编指令等), 这与查询编译器的多层架构非常匹配. 此外, MLIR 提供的跨领域优化能力也有利于深层次地优化嵌有机器学习负载的 SQL 查询. 基于 MLIR, LingoDB 还将查询优化融合到代码生成阶段, 为查询优化提供了新的可能性.

Tahboub 等人[8]则指出基于多级中间表示架构构建查询编译器增加了不必要的复杂性, 于是通过重构生产/消费模型实现了架构清晰的查询编译器. 这种方式虽然降低了查询编译器整体的复杂性, 但由于需要较长的代码生成时间, 也增加了查询延迟.

3.3 总结与分析

如 3.1~3.2 节所述, 不同系统在构建自己的中间表示时, 应当根据自身所处的上下文来选择合适的语言, 不过结合已有研究工作和工业界系统来看,

LLVM IR 通常更受欢迎. 此外, 从最开始的单层架构到如今的多级架构, 围绕着如何设计一个合理且高效的中间表示架构的探讨不断推动着查询编译器的发展. 这些工作影响了后续查询编译器的设计, Kersten 等人[16] 在研发 Umbra 时便借鉴了多层架构思想以及 Tahboub 等人[8] 所提出的新颖方案. 不过至今为止, 关于如何构建查询编译器仍然没有一个统一的方案. 然而, 现有工作主要关注如何降低编译延迟或提高表达能力, 但同时也应当注意相应技术是否会给技术落地带来难度, 因为较低的系统复杂度、实现成本等在工业系统中通常比高性能更为重要. 因此, 如何针对工业界需求去设计更合理的中间表示实现方式应当受到更多关注.

4 机器码生成与运行

如 3.1 节所述, 不少系统都会采用执行效率高的原生语言或 LLVM IR 作为自身的中间表示, 因此如图 13 所示, 在得到这类语言所表示的中间代码后, 系统需要借助通用编译器将其编译成机器码并交由运行时运行才能得到最终的查询结果. 在机器码生成环节存在的关键问题是, 针对运行时间短的查询, 如何降低机器码生成开销; 在机器码运行环节存在的关键问题是, 如何进一步提高机器码运行时的性能. 本节将针对上述 2 个问题对相关工作展开介绍.

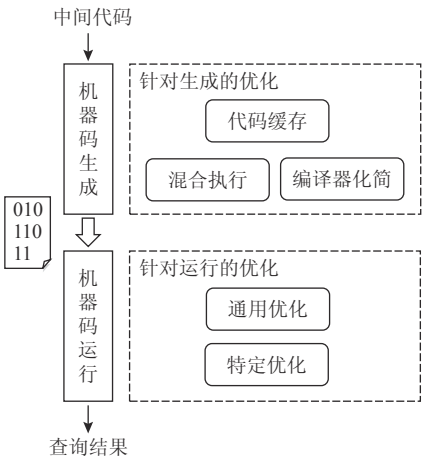


Fig. 13 Optimization for machine code generation and execution

图 13 机器码生成时与运行时优化

4.1 针对机器码生成的优化

通过代码生成器得到的 C/C++等所表示的中间代码并不能直接运行, 首先需要被 GCC, Clang 等通用编译器编译成可执行文件. 这段编译时间通常为毫秒级至秒级, 以 Hyper 执行 TPC-H1 为例, 将中间代码通过编译器编译成可由机器执行的机器码耗时高达 49 ms. 这段开销对于运行时间较长的查询而言可以忽略不计, 但对于运行时间短但生成的代码却复杂的查询而言, 编译所需时间甚至会远超查询执行时间[53], 导致机器码生成这一过程成为性能瓶颈. 在极端情况下, 编译执行的总体性能甚至还不如传统火山模型. 不同于普通程序的 C/C++或 LLVM IR 代码, 生成的中间代码表达的是查询的执行逻辑, 因而有 2 个特性: 一是可能与其他查询的中间代码相同或具有相似的部分; 二是具有 SQL 特有的语义. 现有工作分别利用这 2 个特性对上述问题进行了优化, 包括代码缓存、混合执行和编译器化简.

1)代码缓存. Redshift 构建了编译服务, 当需要编译的代码过多时, 编译服务会将编译任务卸载到远程的计算资源上来加速编译过程. 对于相同或相似的查询, 可以进一步重用已经编译好的部分, 编译服务便通过缓存和复用执行过的查询代码来避免重复编译, 进而降低编译开销[38].

2)混合执行. Kohn 等人[53] 通过混合解释执行与编译执行来降低 Hyper 中的编译开销, 并尽可能地最大化吞吐. 解释执行指的是将中间代码 LLVM IR 通过解释器来解释执行, 比如 LLVM 内置的解释器可以直接解释 LLVM IR 模块, 从而避免编译带来的延迟. 但由于内置解释器的执行性能太低, Kohn 等人[53] 开发了字节码虚拟机, 系统将 LLVM IR 翻译成高效的字节码后, 通过虚拟机对该字节码解释执行, 以此来提升解释执行性能. 然而对于一个通用编程语言实现上述虚拟机等技术并非易事, 不过由于中间代码表达的是 SQL 的执行逻辑, 代码中涉及的主要是 SQL 语义以内的指令, 因此实现相应技术时只需考虑此类指令便可简化技术实现. 优化后的解释执行虽然避免了编译延迟, 但其性能仍然不及编译后的机器码性能, 无法满足系统的高吞吐需求, 因此 Kohn 等人[53] 提出了混合执行. 具体的, Hyper 接收到查询后, 首先生成查询对应的中间代码即 LLVM IR, 随后将中间代码转换为字节码并交给虚拟机解释执行. 在解释执行过程中, 系统会根据运行时的监控信息来判断是否需要切换到编译执行. 如果采用编译执行带来的性能收益高于引入的编译开销, 便自动进行切换. 因为 2 种执行方式执行的是完全相同的代码, 因而很容易做到无缝切换. Menon 等人[54] 在自己的内存数据库中也采用了类似的混合执行策略, 自适应地切换执行方式以兼顾查询处理的低延迟与高

吞吐.

3)编译器化简. 通过精简通用编译器的冗余编译操作来提高编译速度. Kersten 等人[16]、Funke 等人[55]认为通用编译器包含复杂的代码分析等过程，这些对于编译结构简单的 SQL 代码来说是不必要的. 因此，他们针对关系查询设计并实现了定制的编译栈，其包括中间表示和编译后端等，避免了原先编译过程中一些不必要的操作. 举例来说，Umbra 通过编译后端 Flying Start 直接将 Umbra IR 翻译成 x86_64 机器码，在 1～2 ms 内便能够完成 TPC-H 查询的编译. 随后 Umbra 面向 AArch64 架构也构建了低延迟编译后端 FireARM[48]. ReSQL[55] 也通过实现 Flounder IR 达到了低延迟的查询编译.

4.2 针对机器码运行的优化

编译得到的可执行文件的性能相较于原先采用“一次一元组”解释执行，虽然有了显著提升，但如果运行时的数据分布与查询优化器估计的不一致，会导致执行效率降低. 此外，代码生成器生成的代码大多面向的是 CPU，如果在 GPU 上运行相同的代码逻辑，可能会出现 Warp 分歧现象，降低执行效率. 针对上述问题，现有工作在代码运行过程中，分别利用自适应等通用优化技术和特定优化技术予以解决.

1)通用优化. Menon 等人[54]将自适应查询处理应用到了查询编译中，在执行时动态调整谓词顺序、连接顺序等来加速代码执行，并且通过分解和结构化过滤操作等方法来避免重新编译查询. Haffner 等人[44]指出 V8 引擎自身就支持上述混合执行和自适应优化等，因此基于 V8 实现了查询引擎 mutable，并将上述执行技术全部委托给 V8 处理. 具体地，mutable 首先为查询计划生成基于 Wasm 的中间代码，随后将该代码交由 V8 的 Liftoff 进行快速编译与执行. V8 在执过程中会逐渐切换到 TurboFan 产生的优化代码上来执行. 其中，Liftoff 提供了类似 Umbra 中 Flying Start 的功能，TurboFan 则类似 LLVM 这一优化编译器.

2)特定优化. 以数据为中心的代码生成会将多个算子执行逻辑封装在 1 个函数中，使得生成的代码紧凑且高效，但是这种方式也会导致函数有时会包含相对复杂的分支判断等逻辑. 在 GPU 中执行这类函数时，可能会加剧 Warp 分歧现象，DogQC[22] 通过下推式并行和填充机制优化了该问题.

4.3 总结与分析

如何降低查询编译所引人的编译开销，Kersten 等人[16]、Kohn 等人[53]给出了精简通用编译器、混合解释执行与编译执行等解决思路. 此外，Menon 等人[54]的研究工作则针对运行时存在的问题进行了优化，进一步提高了系统吞吐. 机器码的生成与运行和底层硬件息息相关，随着底层硬件的产业变革，FireARM 等工作开始研究 ARM 架构之上的机器码生成. 相同的变革不仅改变着数据管理系统领域，同样影响着深度学习领域的技术演进，如支持不同体系架构的深度学习编译器 TVM[56]. 因此研究者在探索新硬件或 ARM 架构之上的查询编译器时，应当积极借鉴或复用其他领域的已有技术，一方面能够加速查询编译领域的发展，另一方面也有利于未来不同领域系统间的融合.

5 基于查询编译的数据管理系统对比

查询编译给查询处理带来的性能提升催生了一众基于查询编译的数据管理系统，包括工业系统 Spark SQL, SingleStore 以及学术领域中的 LegoBase, Umbra 等. 不同系统在代码生成器框架、中间表示的语言以及所采取的优化技术的选择上各有不同，因而在性能和功能等维度上有显著差异. 在表 2 中，本文结合近期工作[48]，从延迟、吞吐、表达能力等指标上综合比对了相关系统. 其中选取的系统主要采用以数据为中心的代码生成(参见 2.1.2 节)，因为该策略性能提升最为显著，被主流基于查询编译的系统采用.

1)延迟与吞吐. 延迟与吞吐均是系统的性能指标. 延迟指对查询进行代码生成和机器码生成所需的编译时间，对于交易场景而言是主要的评价指标，延迟越低表明系统可以越快地响应前端请求. 吞吐指系统每秒所能处理的元组数量，吞吐大小与生成的代码质量密切相关. 对于计算密集型负载而言，吞吐越大表明系统能够完成作业的速度越快. 在基于查询编译的数据管理系统中，延迟与吞吐有时会相互制约：比如减少编译器优化步骤可以降低编译延迟，然而这会导致产生的代码质量变低，从而使得吞吐减小. 但系统可以通过采用第 4 节所述的优化手段来提高自身吞吐.

2)表达能力. 表达能力用来评估在系统的中间表示上能否高效便捷地表达算法细节，如算子语义、内存交互等. 良好的表达能力对于生成高质量查询代码至关重要，同时也有利于降低代码生成的复杂性.

3)移植能力. 移植能力用来评估将基于查询编译的系统移植到不同硬件或架构之上时，对硬件或

Table 2 Comparison of Different Metrics for Data Management Systems Based on Query Compilation

表 2 基于查询编译的数据管理系统的不同指标对比

中间表示架构	系统	年份	中间表示	延迟	吞吐	表达能力	移植能力	是否开源
单层	HyPer[6]	2011	LLVM IR	++	++	−−−	−	×
	ReSQL[55]	2020	Flounder IR	++	+	−−−	++	×
	mutable[44]	2023	Wasm	+++	+	−−−	−	×
多层	Hekaton[35]	2013	MAT，PIT，C	−−−	++	+	−−	×
	DBLAB/LB[49]	2016	Scala DSLs	−−−	++	++	−−	×
	LegoBase[7]	2018	Scala，C	−−−	++	+	−−	×
	Excalibur[50]	2021	VOILA	−−	++	++	+	√
	Umbra(低延迟)[16]	2021	Umbra IR	+++	++	+++	+	×
	Umbra(高吞吐)[16]	2021	Umbra IR	−	+++	−	−	×
	LingoDB[30]	2022	MLIR	−−−	+	++	−	√

注：“+”越多表明该系统在该指标上表现越优，“−”越多表明该系统在该指标上更为薄弱.

架构特有属性如内存模型、特有指令等的应对能力. GPU, FPGA 等新硬件以及 ARM 架构的兴起给数据管理系统带来了新的变革，而它们通常具有不同特性，因此良好的移植能力对系统而言也是必不可少的.

4)是否开源. 不同系统所采用的查询编译执行模型在诸多模块上都有着相同之处，比如代码生成器框架、与运行时的交互模块等. 系统开源能够减少重复开发上述模块所消耗的人力、物力成本，也有利于各类创新想法的验证与融合，更有助于查询编译的创新与发展.

6 未来研究趋势

在传统关系型数据管理系统中，各类优化工作使得查询编译技术愈来愈成熟与完善. 但与此同时，分析型负载变得越来越复杂多样，新型编译技术层出不穷，新硬件和云计算也逐渐演变成了数据管理系统中的基础组件. 这些变革给查询编译带来新机遇的同时也提出了新的挑战. 本节将从 1.2 节的研究挑战出发，结合查询编译领域当前的前沿工作，从代码生成、中间表示和机器码生成与运行 3 个层面来展望该技术未来可能的发展趋势，具体可细分为 4 个方向：

1)应对复杂分析负载的高吞吐与低延迟需求. 在代码生成层面，需要考虑上层新型负载对构建代码生成器提出的新需求. 如今的数据分析任务变得越来越复杂，仅仅通过原生的 SQL 查询已经无法满足这类负载. 以数据探索分析为例，一个分析工作往往需要多个库或多个系统来协同合作. 包括从数据管理系统中检索出所需数据，随后交由数据清洗、特征提取等库进行处理，最后输入至机器学习模型进行训练或推理. 上述多个库或系统互为黑盒，这导致整个分析任务中存在不必要的数据传输与冗余计算，进而造成整个分析查询吞吐低或延迟高. 近些年来一些研究工作聚焦于通过查询编译来解决上述问题，通过将不同系统的执行计划映射到统一的中间表示上以实现跨系统的协同优化[15,30,57]，并通过代码生成产生高效的执行代码. 而跨系统的分析任务多种多样，仍存在不少优化空间，需要进一步地深入研究. 此外，通过查询编译技术来加速其他引擎也是一个新生方向，如流计算[36]、图计算引擎等.

2)融合新型编译技术提升查询编译的综合能力. 如 3.3 节所述，如何构建查询编译器或中间表示仍没有一个统一的方案，因此在中间表示层面可以通过借鉴或利用编译器领域的新技术来进一步探索. 查询编译与编译器领域密切相关，不少核心思想或技术都来源于该领域，如多级中间表示、LLVM IR 等. 编译器领域中相关技术不断推陈出新，其中一些技术依然可以被查询编译所采用. 诸如通过编译器基础设施 MLIR 来构建统一的多层中间表示，利用新型字节码格式 Wasm 来加速机器码生成等. 新型编译技术层出不穷，如何合理且有效地整合这些新型编译技术值得进一步挖掘. 此外，查询编译的特性增加了系统开发和性能分析难度[58]，这也使得相关厂商在开发系统时，不得不放弃使用查询编译. 如何针对查询编译构建高效的代码调试等工具，从而提高开发效率[59]并完善功能特性也是一个前沿研究方向.

3)面向新硬件和云计算的高效代码生成. 在机器码生成与运行层面，应当关注如何根据不同硬件特性来设计高效的代码生成策略，充分考虑异构

计算下的诸如存储层级和数据传输等，从而进一步提高机器码运行时的性能. GPU 等硬件加速器为数据处理带来了巨大的潜力，越来越多的工作面向新硬件采用代码生成技术进一步提高了异构计算性能[22-23,60]. 然而，这类新硬件与 CPU 相比从编程语言到底层架构都存在不小的差异. 此外，无服务器计算的出现也给查询编译带来了新的可能，例如通过无服务器计算提升机器码生成速度[61]等. 随着数据管理系统不断云化，如何面向云场景设计更合理的查询编译策略也值得深入研究.

4)针对不同体系架构的查询编译范式. 在机器码生成与运行层面，还应当考虑如何面向 RISC 架构扩展现有的查询编译执行模型. 随着摩尔定律逐步放缓，硬件市场的格局也已经发生了变化. 基于复杂指令集的 x86 已不再是主导市场的核心架构，不少厂商逐步投入到采用精简指令集架构的 ARM 或 RISC-V. 与此同时，近期一些工作也指出基于 ARM 的数据管理系统性能会超过基于 x86 的，这可能结束 x86 架构在数据管理系统中的主导地位[48,62]. 因此，一些前沿工作开始设计支持不同架构和硬件的领域特定中间表示[50-51]，从而使得基于查询编译的系统在新架构下也能兼具较低的编译延迟和高吞吐. 此外，智能手机等终端设备的持续普及以及嵌入式数据库的发展也会加速上述变革，如何适应体系架构的转变并设计新型查询编译范式是未来值得关注的发展方向.

7 总 结

查询编译为查询生成紧凑高效的定制化代码，能够显著提升查询处理性能. 越来越多的系统通过查询编译技术提升查询执行性能，有的是即时编译简单表达式，有的则是针对算子或整个查询进行编译. 在加速查询执行方面，查询编译器正在逐渐成为数据管理系统不可或缺的组件. 本文首先概述了查询编译的基本概念，随后从代码生成、中间表示、机器码生成与运行 3 个角度对查询编译现有工作进行了总结分析. 如今，新型硬件加速器和编译器技术等给查询编译带来机遇的同时也引入了新的挑战，本文最后总结了该领域的前沿研究方向，同时展望了查询编译未来可能的发展方向. 总之，在提升查询编译性能的同时降低其技术复杂度仍然任重道远.

作者贡献声明：潘青峰完成了论文撰写；徐辰提出指导意见并修改论文.

参 考 文 献

[1] Kersten T, Leis V, Kemper A, et al. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask[J]. Proceedings of the VLDB Endowment, 2018, 11(13): 2209−2222

[2] Boncz P A, Zukowski M, Nes N. MonetDB/X100: Hyper-pipelining query execution[C/OL]//Proc of the 2nd Conf on Innovative Data Systems Research (CIDR). 2005 [2023-11-28].https://www.cidrdb.org/cidr2005/papers/P19.pdf

[3] Raposa R, Roscigno D, Bragin T, et al. Distinctive features of ClickHouse [EB/OL]. [2023-06-13].https://clickhouse.com/docs/en/about-us/distinctive-features#vector-engine

[4] Garcia-Molina H, Ullman J D, Widom J. Database System Implementation Second Edition[M]. Upper Saddle River: Prentice Hall, 2010: 759−760

[5] Chamberlin D D, Astrahan M M, Blasgen M W, et al. A history and evaluation of System R[J]. Communications of the ACM, 1981, 24(10): 632−646

[6] Neumann T. Efficiently compiling efficient query plans for modern hardware[J]. Proceedings of the VLDB Endowment, 2011, 4(9): 539−550

[7] Klonatos Y, Koch C, Rompf T, et al. Building efficient query engines in a high-level language[J]. Proceedings of the VLDB Endowment, 2014, 7(10): 853−864

[8] Tahboub R Y, Essertel G M, Rompf T. How to architect a query compiler, revisited[C]//Proc of the 2018 ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2018: 307−322

[9] PostgreSQL Community. What is JIT compilation [EB/OL]. [2023-08-17].https://www.postgresql.org/docs/current/jit-reason.html

[10] OceanBase Community. ob_enable_jit [EB/OL]. [2023-08-17].https://en.oceanbase.com/docs/common-oceanbase-database-10000000000931187

[11] openGauss Community. Other optimizer options [EB/OL]. [2023-08-17].https://docs.opengauss.org/en/docs/1.0.0/docs/Developerguide/other-optimizer-options.html

[12] Armbrust M, Xin R S, Lian Cheng, et al. Spark SQL: Relational data processing in Spark[C]//Proc of the 2015 ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2015: 1383−1394

[13] Agarwal S, Liu D, Xin R. Apache Spark as a compiler: Joining a billion rows per second on a laptop[EB/OL]. [2023-06-13].https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html

[14] Prout A, Wang S P, Victor J, et al. Cloud-native transactions and analytics in SingleStore[C]//Proc of the 2022 ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2022: 2340−2352

[15] Palkar S, Thomas J, Narayanan D, et al. Evaluating end-to-end optimization for data analytics applications in Weld[J]. Proceedings of the VLDB Endowment, 2018, 11(9): 1002−1015

[16] Kersten T, Leis V, Neumann T. Tidy tuples and flying start: Fast

compilation and fast execution of relational queries in Umbra[J]. The VLDB Journal, 2021, 30(5): 883−905

[17] Neumann T. Evolution of a compiling query engine[J]. Proceedings of the VLDB Endowment, 2021, 14(12): 3207−3210

[18] Viglas S D. Just-in-time compilation for SQL query processing [C]//Proc of the 30th IEEE Int Conf on Data Engineering (ICDE). Piscataway, NJ: IEEE, 2014: 1298−1301

[19] Melnik D, Buchatskiy R, Zhuykov R, et al. JIT-compiling SQL queries in PostgreSQL using LLVM [EB/OL]. [2023-06-13].https://www.pgcon.org/2017/schedule/events/1092.en.html

[20] Kornacker M, Behm A, Bittorf V, et al. Impala: A modern, open-source SQL engine for Hadoop[C/OL]//Proc of the 7th Conf on Innovative Data Systems Research (CIDR). 2015[2023-11-28].https://www.cidrdb.org/cidr2015/Papers/CIDR15_Paper28.pdf

[21] Krikellas K, Viglas S D, Cintra M. Generating code for holistic query evaluation[C]//Proc of the 26th IEEE Int Conf on Data Engineering (ICDE). Piscataway, NJ: IEEE, 2010: 613−624

[22] Funke H, Teubner J. Data-parallel query processing on non-uniform data[J]. Proceedings of the VLDB Endowment, 2020, 13(6): 884−897

[23] Yang Zhenhua, Pan Qingfeng, Xu Chen. Fine-grained tuple transfer for pipelined query execution on CPU-GPU coprocessor[C]//Proc of the 28th Int Conf on Database Systems for Advanced Applications (DASFAA). Berlin: Springer, 2023: 19−34

[24] Foufoulas Y, Simitsis A, Stamatogiannakis L, et al. YeSQL: "You extend SQL" with rich and highly performant user-defined functions in relational databases[J]. Proceedings of the VLDB Endowment, 2022, 15(10): 2270−2283

[25] Kläbe S, DeSantis R, Hagedorn S, et al. Accelerating Python UDFs in vectorized query execution[C/OL]//Proc of the 12th Conf on Innovative Data Systems Research (CIDR). 2022 [2023-09-19].https://www.cidrdb.org/cidr2022/papers/p33-klaebe.pdf

[26] Hirn D, Grust T. PL/SQL without the PL[C]//Proc of the 2020 ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2020: 2677−2680

[27] Ramachandra K, Park K, Emani K V, et al. Froid: Optimization of imperative programs in a relational database[J]. Proceedings of the VLDB Endowment, 2017, 11(4): 432−444

[28] Chen Hanfeng, D'silva J V, Chen Hongji, et al. HorseIR: Bringing array programming languages together with database query processing[C]//Proc of the 14th ACM SIGPLAN Int Symp on Dynamic Languages. New York: ACM, 2018: 37−49

[29] Sichert M, Neumann T. User-defined operators: Efficiently integrating custom algorithms into modern databases[J]. Proceedings of the VLDB Endowment, 2022, 15(5): 1119−1131

[30] Jungmair M, Kohn A, Giceva J. Designing an open framework for query optimization and compilation[J]. Proceedings of the VLDB Endowment, 2022, 15(11): 2389−2401

[31] Chen Hanfeng, D'silva J V, Hendren L, et al. HorsePower: Accelerating database queries for advanced data analytics[C]//Proc of the 24th Int Conf on Extending Database Technology (EDBT). Berlin: Springer, 2021: 361−366

[32] Crotty A, Galakatos A, Dursun K, et al. An architecture for compiling UDF-centric workflows[J]. Proceedings of the VLDB Endowment, 2015, 8(12): 1466−1477

[33] Zhang Wangda, Kim J, Ross K A, et al. Adaptive code generation for data-intensive analytics[J]. Proceedings of the VLDB Endowment, 2021, 14(6): 929−942

[34] Răducanu B, Boncz P, Zukowski M. Micro adaptivity in vectorwise[C]//Proc of the 2013 ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2013: 1231−1242

[35] Freedman C, Ismert E, Larson P Å. Compilation in the Microsoft SQL Server Hekaton engine[J]. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 2014, 37(1): 22−30

[36] Grulich P M, Sebastian B, Zeuch S, et al. Grizzly: Efficient stream processing through adaptive query compilation[C]//Proc of the 2020 ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2020: 2487−2503

[37] Sellis T K. Multiple-query optimization[J]. ACM Transactions on Database Systems, 1988, 13(1): 23−52

[38] Armenatzoglou N, Basu S, Bhanoori N, et al. Amazon Redshift re-invented[C]//Proc of the 2022 ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2022: 2205−2217

[39] Rao J, Pirahesh H, Mohan C, et al. Compiled query execution engine using JVM[C]//Proc of the 22nd IEEE Int Conf on Data Engineering (ICDE). Piscataway, NJ: IEEE, 2006: 23−34

[40] Meijer E, Beckman B, Bierman G. LINQ: Reconciling object, relations and XML in the .NET framework[C]//Proc of the 2006 ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2006: 706−706

[41] Nagel F, Bierman G, Viglas S D. Code generation for efficient query processing in managed runtimes[J]. Proceedings of the VLDB Endowment, 2014, 7(12): 1095−1106

[42] Haas A, Rossberg A, Schuff D L, et al. Bringing the web up to speed with WebAssembly[C]//Proc of the 38th ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2017: 185−200

[43] LLVM Community. Class template reference [EB/OL]. [2023-06-13]. http://llvm.org/docs/doxygen/html/classllvm_1_1IRBuilder.html

[44] Haffner I, Dittrich J. A simplified architecture for fast, adaptive compilation and execution of SQL queries[C]//Proc of the 26th Int Conf on Extending Database Technology (EDBT). Berlin: Springer, 2023: 1−13

[45] Bynens M, Steiner T, Sharma U, et al. What is V8 [EB/OL]. [2023-06-13].https://v8.dev

[46] SQLite Consortium. The SQLite bytecode engine [EB/OL]. [2023-06-13].https://www.sqlite.org/opcode.html

[47] Lattner C, Amini M, Bondhugula U, et al. MLIR: Scaling compiler infrastructure for domain specific computation[C]//Proc of the 19th Int Symp on Code Generation and Optimization (CGO). New York: ACM, 2021: 2−14

[48] Gruber F, Bandle M, Engelke A, et al. Bringing compiling databases to RISC architectures[J]. Proceedings of the VLDB Endowment, 2023, 16(6): 1222−1234

[49] Shaikhha A, Klonatos Y, Parreaux L, et al. How to architect a query compiler[C]//Proc of the 2016 ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2016: 1907−1922

[50] Gubner T, Boncz P. Excalibur: A virtual machine for adaptive fine-grained JIT-compiled query execution based on VOILA[J]. Proceedings of the VLDB Endowment, 2022, 16(4): 829−841

[51] Gubner T, Boncz P. Charting the design space of query execution using VOILA[J]. Proceedings of the VLDB Endowment, 2021, 14(6): 1067−1079

[52] MLIR Community. MLIR language reference: Dialects [EB/OL]. [2023-12-15].https://mlir.llvm.org/docs/LangRef/#dialects

[53] Kohn A, Leis V, Neumann T. Adaptive execution of compiled queries[C]//Proc of the 34th IEEE Int Conf on Data Engineering (ICDE). Piscataway, NJ: IEEE, 2018: 197−208

[54] Menon P, Ngom A, Ma Lin, et al. Permutable compiled queries: Dynamically adapting compiled queries without recompiling[J]. Proceedings of the VLDB Endowment, 2020, 14(2): 101−113

[55] Funke H, Mühlig J, Teubner J. Low-latency query compilation[J]. The VLDB Journal, 2022, 31(6): 1171−1184

[56] Chen Tianqi, Moreau T, Jiang Ziheng, et al. TVM: An automated end-to-end optimizing compiler for deep learning[C]//Proc of the 13th USENIX Symp on Operating Systems Design and Implementation (OSDI). Berkeley, CA: USENIX Association, 2018: 578−594

[57] Damme P, Birkenbach M, Bitsakos C, et al. DAPHNE: An open and extensible system infrastructure for integrated data analysis pipelines[C/OL]//Proc of the 12th Conf on Innovative Data Systems Research(CIDR).2022[2023-09-19].https://www.cidrdb.org/cidr2022/papers/p4-damme.pdf

[58] Behm A, Palkar S, Agarwal U, et al. Photon: A fast query engine for lakehouse systems[C]//Proc of the 2022 ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2022: 2326−2339

[59] Kersten T, Neumann T. On another level: How to debug compiling query engines[C/OL]//Proc of the 8th Int Workshop on Testing Database Systems. New York: ACM, 2020[2023-11-28].https://dl.acm.org/doi/pdf/10.1145/3395032.3395321

[60] Pirk H, Moll O, Zaharia M, et al. Voodoo-a vector algebra for portable database performance on modern hardware[J]. Proceedings of the VLDB Endowment, 2016, 9(14): 1707−1718

[61] Chinta K, Li Quan, Kleider M, et al. Fast and predictable performance with serverless compilation using Amazon Redshift [EB/OL]. [2023-06-13].https://aws.amazon.com/cn/blogs/big-data/fast-and-predictable-performance-with-serverless-compilation-using-amazon-redshift/

[62] Gubner T, Boncz P A. Highlighting the performance diversity of analytical queries using VOILA[C]//Proc of the 12th Int Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS). New York: ACM, 2021: 47−54



Pan Qingfeng, born in 1998. PhD candidate. His main research interest includes large-scale data processing systems.

潘青峰, 1998 年生. 博士研究生. 主要研究方向为大规模数据处理系统.



Xu Chen, born in 1988. PhD, professor, PhD supervisor. Member of CCF. His main research interests include large-scale data processing systems, database systems, and intelligent computing systems.

徐　辰, 1988 年生. 博士，教授，博士生导师. CCF 会员. 主要研究方向为大规模数据处理系统、数据库系统、智能计算系统.