动态编译中列式数组的控制流复制

Sebastian Kloibhofer[a]、LukasMakor[a]、David Leopoldseder[b]、Daniele Bonetta[c]、Lukas Stadler[b]和 Hanspeter Mössenböck[a]
a 奥地利林茨约翰内斯·开普勒大学系统软件研究所 b 奥地利 Oracle Labs c 荷兰 Oracle Labs

摘要 列式数据库是加速联机分析处理（OLAP）查询的一种成熟方式。如今，数据处理（例如存储、可视化和分析）通常在编程语言层面进行，因此，在常见的语言运行时中采用列式数据结构也十分理想。

虽然已有框架、库和 API 可在编程语言中启用列式数据存储，但将其集成到应用程序中通常需要开发人员介入。在先前的工作中，研究人员为 GraalVM JavaScript 运行时实现了一种将数组自动转换为列式数组的方法。然而，这种方法在较小的工作负载以及更复杂的嵌套数据结构上都存在性能问题。我们发现，优化列式数组访问的关键在于识别查询并针对它们应用特定的优化。

本文介绍了 GraalVM 编译器中的新型编译器优化，可用于优化针对列式数组的查询。在 JIT 编译时，我们会识别访问潜在列式数组的循环，并对其进行复制，以专门优化对列式数组的访问。此外，我们还介绍了一种新方法，通过执行多级存储转换，从由复杂对象构成的数组创建列式数组。我们通过针对 JavaScript 日期对象的实现来展示这种方法。

我们的工作表明，即使对于小型工作负载，将数组自动转换为列式存储也是可行的，而且更复杂的对象数组也能从多级转换中受益。此外，我们展示了如何通过复制来优化处理不同状态数组的方法。我们在微基准测试和成熟的数据分析工作负载（TPC-H）上评估了这项工作，结果表明，它的性能显著优于先前的工作，在特定查询上加速比最高可达 10 倍。查询也能从多级转换中受益，加速比约为 2 倍。此外，我们还表明，对于不适合进行存储转换的工作负载，我们不会造成显著开销。

我们认为，自动创建的列式数组可以帮助数据密集型应用的开发者，作为使用专用 API 操作手动创建的列式数组的另一种方式。通过自动检测并优化针对可能为列式数组的数据的查询，我们可以提升数据处理性能，并进一步推动其在常见编程语言（尤其是动态编程语言）中的应用。

ACM CCS 2012

- **软件及其工程** → **动态编译器**；**运行环境**；解释器；
- **信息系统** → 列式存储；

关键词 列式存储、编译时复制、动态语言、动态编译、程序优化

《编程的艺术、科学与工程》

提交于 2023年1月1日**发表于**
2023年2月15日

DOI 10.22152/programming-journal.org/2023/7/9

© S. Kloibhofer, L. Makor, D. Leopoldseder, D. Bonetta, L. Stadler, H. Mössenböck 本作品采用“CC BY 4.0”许可协议。在《编程的艺术、科学与工程》第7卷第3期（2023年）中，第9篇文章；共39页。

1 引言

数据库管理系统早已证明，优化数据布局对于提升大型复杂数据[1, 2, 3, 11, 32, 47, 49,79, 83]的查询性能至关重要。列式数据库 [4, 13, 48, 78] 已成为读密集型查询执行的事实标准，例如联机分析处理（OLAP）场景，其中的查询面向大型、多维且通常只读的数据 [21]。在列式数据库中，表的记录会被拆分，以便将每一列的所有值存储在连续的内存区域中。这提升了针对少量列的查询性能，因为可以在紧凑的循环中高效处理各条记录的列值，从而提高缓存和内存效率。

如今，数据密集型操作通常在编程语言层面执行，运行时和编译器承担着与数据库查询优化器类似的职责 [7, 14,35, 38, 54, 57, 63, 75]。不过，尽管编程语言提供了用于列式存储的特定 API [7, 56, 75]，，但这使得为不同场景选择合适内存布局的责任转移到了应用程序开发者身上。

基于这些观察，Makoret al. [55]提出了一种方法，可在 GraalVM JavaScript 运行时中将对象数组自动转换为列式存储 [66]。他们会在运行时检测合适的数组，并在数组足够大且被频繁访问时，将底层存储转换为列式布局。GraalVM 编译器中的自定义编译器优化 [29, 53, 77] 使他们能够在即时编译（JIT 编译）后获得性能提升。

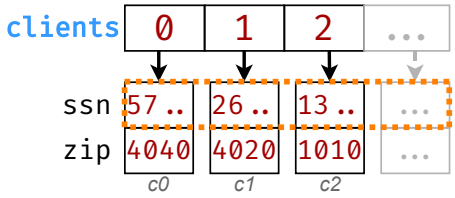
请看图 1a 中的函数 shippingCosts ，它接收一个客户端数组，用于计算每个客户端适用的运费。图 1b 展示了客户端的内存布局，其中各个对象分散在堆中。在上述方法中，数组会在执行期间的某个时刻转换为列式存储，从而形成图 1d 所示的内存布局。随后，函数会在编译过程中得到优化。图 1c 展示了编译后函数的高级表示。此时，运行时检查（即所谓的保护条件）会确保该数组确实为列式数组；如果不是，编译后的代码就会中止执行（参见第 2.1 节）。这样便可进一步优化，消除不再必要的内存访问，并将操作移出循环。

这种方法的一个问题是，编译器优化依赖于这样的假设：优化时可以专注于单一数组状态，例如，优化后的访问总是针对列式数组。因此，一个方法要么针对列式数组编译，要么针对非列式数组编译。如果调用已编译的方法时，数组处于非预期状态，就会触发去优化 [41]（参见示例中的保护条件），也就是说，执行会退回到解释器，且编译后的代码会被丢弃。然而，由于需要重新编译，这一过程会带来额外开销，对规模较小、执行频率较低的工作负载影响尤为明显。此外，这种行为在即时编译环境中并不可取，因为理想情况下，热点方法的编译应在一段时间后趋于稳定。

此外，使用包含引用类型属性的对象数组的工作负载性能不佳，因为访问这些属性占用了大部分运行时开销，而且无法进行充分优化。当查询中使用日期和字符串时，这个问题尤为明显。

(a)用于计算数组运费的函数
根据客户端的位置。

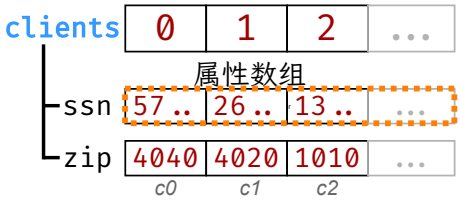
```
函数shippingCosts(clients) {
  常量成本 = Array(clients.length).fill(0);
  对于(令 i = 0; i < 客户端.length; i++)
    cost[i] = 客户端[i].邮政编码 == 4040? 15 : 20;
  返回 cost;
}
```



(b)数组的内存表示
转换之前。

(c)变换后的函数高级表示
对客户端进行列式存储转换。

```
函数shippingCosts(clients) {
  常量 成本 = 数组(clients.length).fill(0);
  保护条件 <是否为列式>(clients);
  常量邮政编码 = clients.__属性数组["邮政编码"];
  对于(令 i = 0; i < clients.length; i++)
    cost[i] = 邮政编码[i] == 4040? 15 : 20;
  返回 cost;
}
```



(d)变换后的数组的内存表示
。

图 1 客户端数组的转换。

因此，我们提出了一种新方法，可在 JIT 编译时检测可优化的查询，随后通过复制代码来实现对列式数组循环的优化。此外，我们还展示了一种新方法，可通过多级存储转换来处理列式数组中的嵌套数据。

总体而言，本研究的贡献如下：1. 针对 [55] 中提出的方法，提出新颖的编译器优化，自动复制用于访问列式数组的控制流，以减少重新编译的工作并加速较小的工作负载（第 4 节）。2. 改进存储转换方法，以提升访问包含具有日期属性的对象的对象数组数据结构时的性能；此方法还可扩展至其他复杂嵌套类型（第 3 节）。3. 基于微基准测试和 TPC-H 查询 [23] 对这些改进进行评估，结果表明，较小工作负载的性能最高可提升 3 倍，较大工作负载的性能最高可提升 10 倍（第 5 节）。

此外，第 2 节介绍了编译器以及本研究所依据方法的必要背景。第 6 节阐述了我们方法的局限性。第 7 节讨论相关工作。

2 背景

GraalVM [65] 是一款高性能 Java 虚拟机。Truffle 语言实现框架 [39, 85, 88]——GraalVM 中用于实现

抽象语法树（AST）解释器的组件——支持执行其他编程语言。我们将所提出的方法集成到了基于 Truffle、符合标准的 JavaScript 运行时 GraalVM JavaScript [66] 和 GraalVM 编译器中。

2.1 GraalVM 编译器

GraalVM 编译器 [29, 53, 77] 是一种自托管的高性能动态即时编译器，利用运行时剖析信息优化程序。该编译器基于运行时观测应用推测优化，对方法作出假设（例如关于类型和值的假设），并启用后续优化 [30,31]。如果某个假设失败，运行时会使用去优化。去优化允许运行时中止已编译方法的执行，丢弃编译后的代码，并在解释器中恢复执行 [41]。

GraalIR 在 GraalVM 编译器中，方法首先被变换为一种称为 Graal IR 的中间表示 [29]。Graal IR 是一种结构化的有向图，由表示方法控制流和数据流的节点组成。该中间表示采用静态单赋值形式 [19, 25, 59, 60, 84], ，其中 ϕ 节点表示依赖于当前路径的值（例如， ϕ 节点捕获一个赋值，其值取决于分支；循环 ϕ 表示在循环中修改的值）。为了在编译期间遍历中间表示，Graal IR 节点通常会被安排在基本块 [20, 70]中。基本块是由最少数量节点组成的无分支集合，并与其相应的前驱节点和后继节点连接。

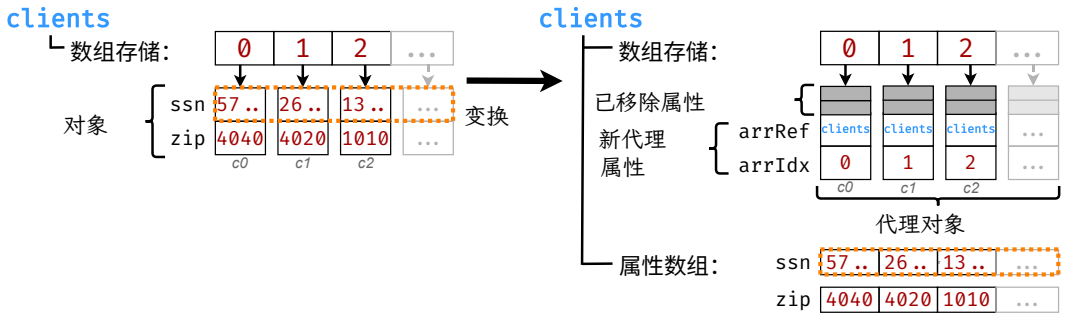
所谓的支配关系 [6, 22, 25, 37, 51, 74]在一个方法的基本块上施加偏序：如果控制流中通往B_1 的每条路径都先经过B_0 ，那么基本块B_0 支配另一个基本块B_1 。利用每个基本块的直接支配块（即离它最近的支配块），这些基本块构成支配树，其中根块代表方法入口。因此，编译器阶段经常利用这一结构遍历方法的节点。

编译期间，中间表示会在各种编译器阶段中被修改。这些阶段包括部分逃逸分析 [77], 、内联 [73], 和 SIMD 向量化 [15, 24, 43, 44]。最后，编译器根据中间表示生成高度优化的机器码。

2.2 Truffle

Truffle [39, 85, 88] 是一种语言实现框架，可通过 AST 解释执行实现宿主语言集成。运行时，宿主语言的抽象语法树会根据解释执行过程中积累的剖析信息进行部分求值和优化 [42, 50, 87, 89] ，随后使用 GraalVM 编译器进行编译。Truffle 还提供了一种标准化的对象存储模型 [86]，可供所有语言实现共用。

我们的方法基于高性能 JavaScript Truffle 实现 GraalVM JavaScript [66]。GraalVM JavaScript 符合 ECMAScript [33] ，并具有 Node.js 后端 [69]。GraalVM JavaScript 使用 Truffle 对象存储模型 [86]来支持 JavaScript 对象和数组语义 [33]。

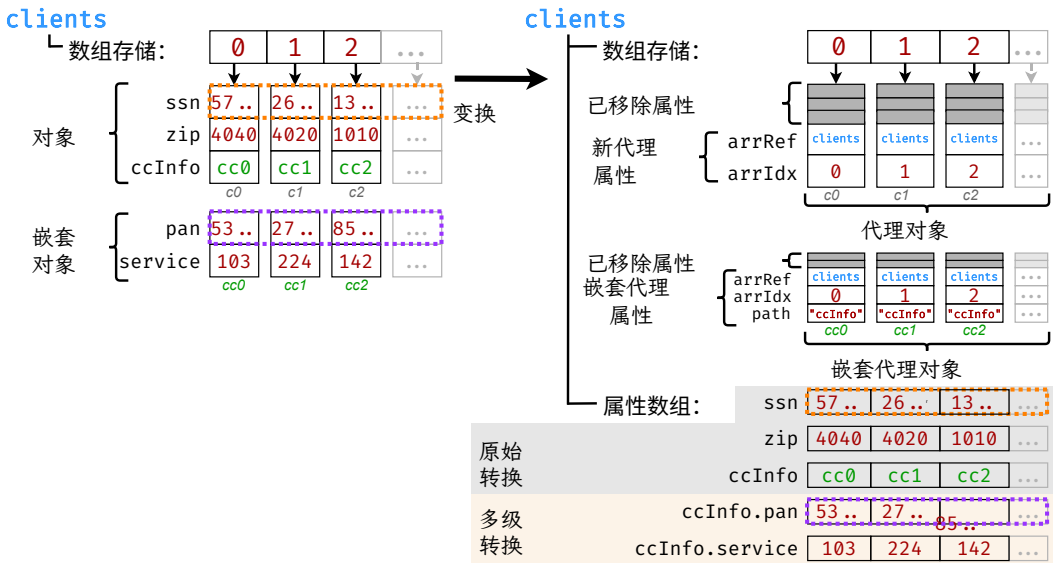


■ **图 2** 将对象数组变换为列式数组（基于 [55]）。

2.3 GraalVM JavaScript 中的列式数组

Makor 等人 [55]表明，通过将统一对象数组转换为列式数组，并相应地优化对所得列式存储的访问，可以显著加速分析型工作负载中的数据密集型查询。这种表示方式通常能提高遍历数组时的缓存利用率，并且还允许在编译时优化循环中的数组访问，如 [55]所示。要执行此类转换，需要在运行时跟踪对象数组的信息。当检测到一个符合条件且频繁使用、仅包含同一类型对象的数组时（参见左侧的客户端 图 2），便会启动向列式数组的转换。首先，通过验证步骤确保该数组确实只包含合适的对象（所有对象类型相同、其属性可访问等）。否则，将中止转换步骤。如果验证成功，则为原始数组中对象的每个属性创建一个新数组。然后，将对象中相应属性的所有值复制到新数组中。因此，这些数组称为属性数组。如图 2 右下方所示，原始数组随后会引用这些属性数组。

原数组中的每个对象都会转换为一个代理对象。如 [55],所述，名称代理 采用自相关工作 [56, 71]，不要将其与 ECMAScript 代理对象 [33]混淆。之所以需要进行代理转换，是因为数组转换后，每个对象在需要访问自身数据时，都必须访问列式存储。要访问构成列式存储的属性数组，代理对象需要访问原数组，以加载正确的属性数组；同时还需要知道数组元素的索引，才能在正确的位置访问属性数组。因此，如图 2 右上角所示，每个代理对象都包含原数组引用（数组引用）和对象在数组中的索引（数组索引）。代理对象的转换仅通过更改内部类型描述符来体现，以反映对象的新角色，因此不会影响对象标识和类型检查，并且能够保留 JavaScript 对象语义。如果对列式数组执行不受支持的操作，则会进行恢复，即将数组转换回非列式状态。这类不受支持的操作包括插入不兼容的元素、引入空洞，以及添加或删除数组元素的属性，或为其赋予与相应属性数组不兼容的新值（例如，将字符串赋给数值属性）。



■ **图 3** 将对象数组多级转换为列式数组。

这种转换仅适用于大型、通常用于读取的数组，但会带来一些初始开销。不过，自定义编译器优化可以通过移除冗余检查、根据数组状态编译方法，以及将循环不变代码移出热点循环，逐步摊销这部分开销。由此，在微基准测试中可实现最高 9 倍的加速比。

3 多级存储转换

Makor 等人 [55] 的结果表明，在执行处理非对象类型属性（例如布尔值 或数字）的查询时，性能有显著提升。然而，结果也表明，在处理引用类型的属性（例如日期、字符串或用户对象类型）时，由于实际数据存储在这些对象的属性中，加速效果会减弱，甚至导致性能下降。这是因为从引用类型对象创建的属性数组（例如图 3 中所示的ccInfo 属性的属性数组）仅包含对实际对象的引用，而不包含最终会被访问的数据（例如ccInfo对象的pan 属性）。通常，对对象进行的许多操作都需要访问其一个或多个属性，因此在循环中使用时，会访问堆中分散的内存位置。然而，存储转换正是要避免这种情况。

为解决这一问题，我们提出了一种在运行时自动进行多级存储转换的方法，以改进 Makor 等人提出的存储转换 [55]。我们对该转换进行了调整，使其不仅涵盖待转换数组中对象的属性（例如图 3 中的 ssn、邮政编码 和 ccInfo），还涵盖嵌套对象的属性（例如嵌套ccInfo 对象的 pan 和 服务属性）。因此，我们不仅将数组元素的属性放置在连续内存区域中，还将嵌套对象的属性置于其中。

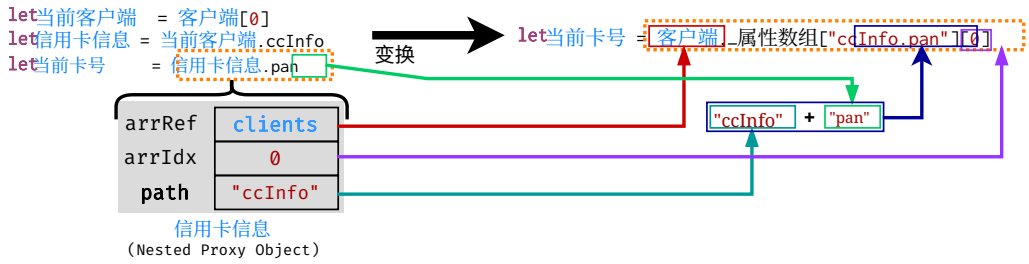


图4 通过嵌套代理对象访问经转换为列式存储后存储在属性数组中的数值属性。

图 3 展示了应用多级存储转换的示例。该示例与图 2 中的示例类似，不同之处在于每个客户端都额外包含一个字段ccInfo，该字段引用一个对象，其中以主账号（pan）和服务代码（服务）的形式存储与该客户端关联的信用卡信息。在使用原始（即单级）方法转换客户端数组时，会为ssn、邮政编码和ccInfo 创建 3 个属性数组，其中ccInfo 属性数组包含指向信用卡信息对象的引用，如图 3 右下方灰色框所示。然而，在执行多级存储转换时，除了这些属性数组外，还会为嵌套ccInfo对象的属性创建属性数组，即分别为ccInfo 对象的pan 属性和ccInfo 对象的服务属性各创建一个属性数组，如图 3 右下角的橙色框所示。将嵌套属性的数据复制到相应的属性数组后，会从嵌套对象中移除这些属性。对嵌套对象属性的访问（例如图 3 中对信用卡信息对象属性的访问）需要访问这些新的属性数组。因此，嵌套对象也需要转换为代理对象，这与 [55] 中将客户端对象转换为代理对象的方式类似。

由于数组中存储的对象可能包含多个嵌套对象，而这些对象可能具有同名属性，因此我们不能仅根据属性名称来存储嵌套属性数组，而需要根据从数组元素到该属性的路径来存储。在图 3 所示的示例中，这意味着，为客户端对象的 ccInfo 属性所引用的嵌套对象的 pan 属性创建的属性数组，是使用路径“ccInfo.pan”存储的。因此，要访问正确的属性数组，就必须知道这条路径。访问属性时，通常只能获取该属性的名称，如图 4 左侧所示：当访问从第一个数组元素的 ccInfo 属性加载的 creditInfo 变量的 pan 属性时，能够获取的只有名称 pan 。遗憾的是，正如前文所述，这些信息不足以访问正确的属性数组。因此，为了解决这个问题，我们不仅需要为嵌套代理对象存储 arrRef 和 arrIdx 属性，还需要存储从数组元素到嵌套对象的属性路径。图 3 右侧中央部分展示了一个示例：与客户端代理对象（即第一级代理对象）相比，嵌套代理对象上还存储了一个名为 path 的额外代理属性，其值为字符串"ccInfo" 。

与原始存储转换类似，加载属性数组的过程独立于嵌套代理对象，即不依赖于图 3 中具体的 客户端 和 ccInfo对象；通常，在处理整个数组时，可以将其移出循环（参见第 4 节中的编译器优化）。因此，循环内部只剩下对正确属性数组的索引访问，从而确保后续迭代访问空间位置相近的内存位置，进而提高缓存利用率。

在附录A中，我们介绍了这种方法如何处理特殊情况，例如嵌套代理对象属于不同数组的情况。

案例研究：JavaScript 日期对象的多级存储转换 我们按照第 3 节所述实现了多级存储转换，但仅将其用于 Date 对象。由于 JavaScript 日期对象在内部以时间戳表示，我们调整了转换步骤，以便对嵌套日期对象执行多级存储转换，从而为这些嵌套日期对象的时间戳创建属性数组。我们关注日期对象，因为它们常用于联机分析处理查询。应用该转换的启发式方法与 [55] 中相同。

将多级存储转换应用于 Date对象后，我们在基于处理Date对象的微基准测试中实现了最高 2 倍的加速比。更详细的结果见第 5 节。

4 列式数组的控制流复制

正如 Makor 等人所示， [55], 列式数组带来的性能提升在很大程度上依赖于对相应访问操作的优化，尤其是在循环中。

编译涉及列式数组的代码时，我们的目标是生成针对特定数组状态（列式或非列式）优化数组循环的代码。我们不希望生成这样的代码：进入循环后才区分列式数组和非列式数组（例如，访问代理对象的属性时），因为这会在每次循环迭代中增加额外的检查和内存访问，并阻止将操作移出循环。遗憾的是，如果不加干预，典型编译器生成的正是这种代码。

Makor 等人 [55] 依靠 守卫来解决这一问题。守卫是嵌入抽象语法树中的运行时检查。随后，GraalVM 编译器会将守卫编译为一项检查；如果守卫失败，该检查就会触发去优化。因此，编译时可以省略守卫失败后的代码。他们在访问列式数组的方法中加入守卫，以断言数组处于特定状态（即列式或非列式）。同样，他们确保数组的任何转换或恢复操作都会触发去优化。这样，编译器就能针对这种特定数组状态优化此类方法（及其循环）。

GraalVM 中的守卫通常以这样的方式使用：反复发生保护失败会促使编译器省略去优化，转而编译一个更通用的方法版本，其中包含这两种情况（即保护条件失败和成功的情况）。虽然这通常会使编译更加复杂，甚至可能导致生成的代码优化程度降低（因为后续依赖于保护条件的指令将不再能够被优化或移除），

但对于其他情况下会频繁重新编译的方法，这能提升其性能，因为它可以避免过度去优化，并使方法趋于稳定；也就是说，如果不再存在保护条件，或保护条件不再被违反，重新编译就会减少并最终消失。

为确保能够快速访问列式数组，[55]中的方法必须阻止这种稳定化；否则，如果没有保护条件，就又必须逐次数组访问（包括在循环中）来区分列式数组和非列式数组。尽管他们的评估结果表明，（大型）列式数组能够从这种方法中获益，因为运行时成本被由此获得的加速比所抵消，但较小的数组或调用频率较低的方法，往往会因转换开销和频繁重新编译而变慢。

我们提出了对这种方法的一项改进。编译方法时，我们通常会区分两条路径：一条是经常使用、因而经过大量优化的路径——快速路径；另一条是使用较少、包含开销更高操作的路径——慢速路径。例如，对于 JavaScript 中两个值的加法，我们可以作如下区分：`a + b`。通过检查*a* 和*b*的类型，我们可以区分快速路径和慢速路径：快速路径会将两个数值 相加；慢速路径则会对两个任意对象执行开销更高的字符串拼接。

我们可以将同样的原则应用于可能接收列式数组的方法编译：快速路径假定数组为列式数组，因此可以进行激进优化。慢速路径中，数组最初不是列式数组，但可能会变换为列式变体。因此，该版本必须在每次访问时检查数组状态，所以无法进行显著优化。通常会使用守卫来验证是否可以采用快速路径，这样只有当程序确实需要慢速路径时，才会编译该路径。在列式数组的上下文中，[55], 选择了这种做法：只编译一条路径（例如列式路径），以便进行优化。不过，我们希望改进这样一点：同时接收列式数组和非列式数组的方法，必须在每次状态变化后重新编译。由于我们希望保留两条路径，同时仍能针对快速路径进行明确优化，因此不能使用守卫。

为解决这些问题，我们提出了一种新方法，目标如下：*i*）在保留对非列式数组访问的同时，对列式数组访问应用编译器优化；*ii*）确保编译状态稳定，不受由数组状态导致的去优化影响，即使对于既可使用列式数组又可使用非列式数组调用的方法也是如此。因此，我们决定使用代码复制来实现这些目标。

代码复制在编译器中经常用于启用后续优化 [17, 53, 61, 62]。由于这意味着需要在通过后续优化实现的性能提升与代码大小增加之间进行权衡，因此通常会采用经过精细调整的启发式方法，来影响是否复制代码的决策 [58]。

图 5 展示了复制如何在保留未优化代码版本以防止反优化的同时，实现对列式数组访问的后续优化。（**a**）展示了对对象数组（客户端）的访问。Makor 等人提出的存储转换方法扩展了语言实现，从而支持对列式数组的访问。在我们的新方法中，我们希望保留对数组的所有访问变体，因此不对数组访问使用保护条件。这样便得到

动态编译器中列式数组的控制流复制

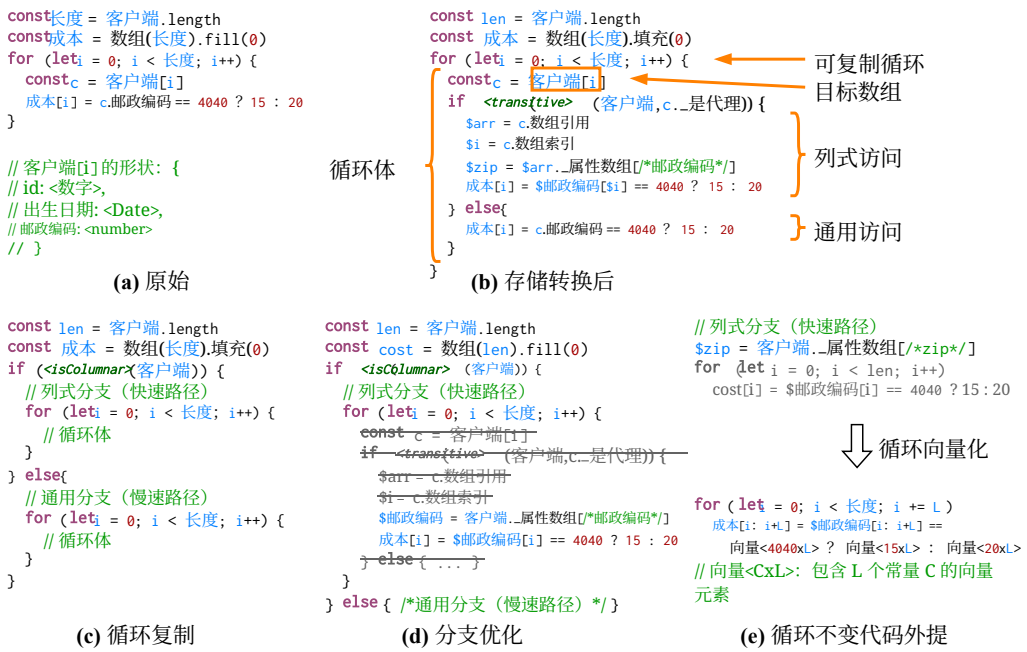


图 5 列式访问的（编译时）复制与优化（高层表示）

各种访问变体，因此我们不对数组访问使用守卫。其结果如（**b**）所示。部分（**c-e**）展示了这一新方法的步骤。首先，它展示了整个循环的复制（**c**），并在此之前进行检查，以确保生成的循环中有一个只处理列式变体的客户端。这种区分支持对“列式分支”进行进一步优化，例如移除冗余检查（**d**），最终优化循环本身（**e**）。以下各节将重点介绍图 5 中的各项优化和准备步骤。请注意，除第 4.1 节外，接下来的各节所介绍的都是相较于 [55]的新想法。

4.1 列式数组标记与内建函数

我们的方法需要语言实现与编译器紧密协作，才能提升性能。运行时需要向编译器传递某些运行时信息和性能分析数据，以帮助检测和优化对列式数组的访问。在编译时，我们需要识别包含适当列式访问的可复制循环。因此，我们在语言实现层使用了多种内建函数。在解释执行期间，内建函数仅表示对内置函数的调用。编译器会检测这些函数，并将其替换为内联的中间表示或汇编代码。在我们的方法中，我们还使用内建函数来标记某些访问和操作，例如将数组转换为列式存储。因此，我们也将这类内建函数称为标记。表 1介绍了我们在方法中使用的一些自定义内建函数。

例如，在图 5 中，我们使用运行时检查 <是否为列式> 来检查数组的状态。在编译时，我们会检测此内建函数，因此可以推断出

表 1 用于向编译器传递信息的自定义内建函数和标记

内建函数	描述
<是否为列式>(arr)	检查数组是否为列式。在编译时，这 check 用于识别已知数组 为列式数组的分支。
<传递性>(arr, cond)	对列式数组始终为 true、对非列式数组始终为 false 的数组检查 。如果已知数组状态，则可以替换条件 cond 如果已知数组状态，则可以替换。
<变换>(arr)	启动对给定数组的存储转换（参见第 2.3 节）。返回值指示转换是否成功。 tion 成功了。
<恢复>(arr)	将给定的列式数组还原为其原始形式。

在相应的分支中，数组已知为列式。类似地，我们使用 <传递性> 来标记在编译时可针对已知数组状态进行优化的检查。例如，当从一个已知为列式的数组中加载对象时，我们可以传递性地得知该对象是代理对象。因此，可以移除这类检查。为防止运行时错误，编译器通常会在图 5 **(b)**中用于访问属性的膨胀代码中添加大量额外的边界检查、类型检查和空值检查。因此，我们还使用特殊的内建函数进行非托管访问，以避免冗余检查，例如，对属性数组进行的边界检查（当数组本身的边界检查已提供保护时）。为简洁起见，图中省略了这些检查。

4.2 循环的复制、选择与优化

在我们的方法中，我们会复制表示查询的整个循环，这些查询会访问（可能是）列式数组。整体原则如图 6 所示的 Graal IR 中所示。图的左侧展示了一个简单方法的控制流（红色箭头），其中包含一个循环 l，该循环会访问数组 α（蓝色箭头表示数据流依赖）；该数组可能处于列式状态，也可能处于非列式状态。A、B 和 C 表示其他任意节点，它们分别出现在循环之前、内部和之后。循环开始处的 条件判断 表示循环条件，用于决定循环是继续执行还是结束。

如图 6 右侧所示，我们会复制包含对（可能是）列式数组的访问的目标循环，并通过 <是否为列式>检查来区分这两个副本，确保只有当数组为列式时才进入左侧循环（l'）。随后，可以针对列式访问优化该循环。由于数组可能会在原始循环（l）内部变换为列式存储，因此该循环必须处理列式访问和非列式访问。因此，我们将其视为慢速路径。循环结束后，控制流会再次合并。

GraalVM 编译器已提供复制部分中间表示的手段 [67]。基于此，我们的新技术必须 *i)* 正确识别一个合适的循环，该循环

动态编译器中列式数组的控制流复制

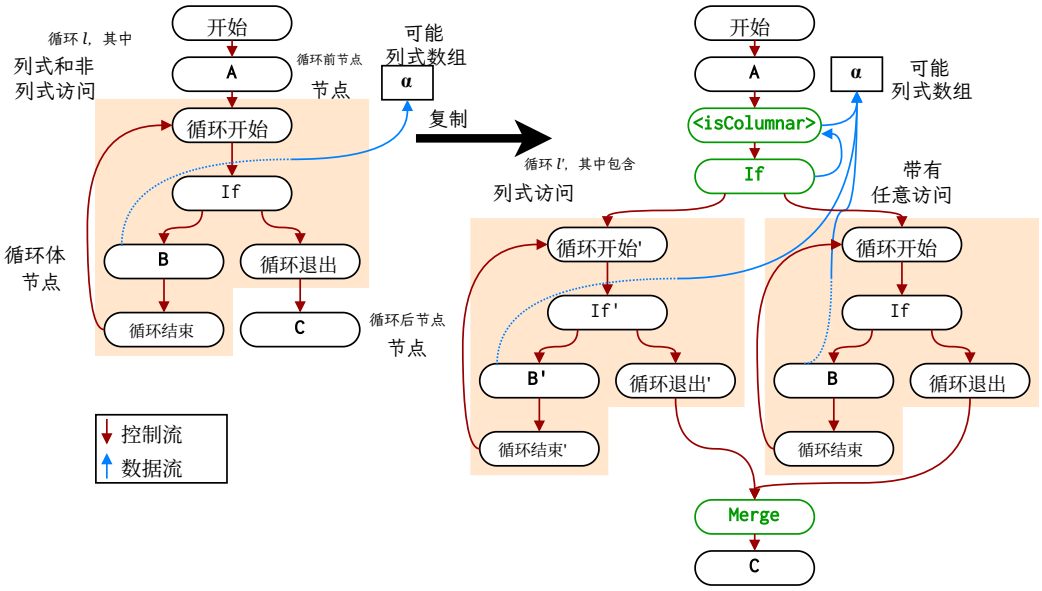


图 6 列式数组访问的控制流复制

包含列式数组访问，*ii*) 在目标循环之前插入数组状态检查，*iii*) 将复制后的循环插入中间表示，并 *iv*) 根据确定的数组状态分别优化两个循环。

以下各节将详细介绍这些具体步骤。

4.2.1 循环分析与选择

由于复制会显著增加代码大小，我们在复制之前增加了一个分析步骤，用于针对目标数组选择要复制的循环。此选择过程还必须确保我们能够安全地优化循环。在语言实现中，我们确保任何不受列式格式支持的数组更改都会触发恢复。我们还使用第 4.1 节中的内建函数来标记数组访问、转换和恢复。这样，我们便能在编译时识别它们，并为每个循环和数组累积度量指标。总之，我们的复制方法对循环有以下要求：

1. 循环必须包含对可能为列式的数组的读取访问。2. 循环必须是在目标数组处于列式状态时执行的。3. 循环不得包含数组恢复或可能导致恢复的操作（例如元素插入、删除）。不过，循环可以包含将数组从非列式格式转换为列式格式的操作。4. 循环不得包含可能导致数组逃逸（并可能改变其状态）的方法调用或其他节点。虽然这看起来可能有所限制，但 GraalVM 部分求值器会内联大多数抽象语法树操作和嵌套调用，因此它们很少会出现在循环体中 [42, 50, 87, 89]。

4.2.2 循环复制

在复制之前，我们会插入一个自定义的 <是否为列式> 检查，如图 5 **(c)** 中的示例和图 6 中的中间表示所示。结合这一检查，以及我们已知列式数组在循环内不会恢复到原始状态这一事实（参见第 4.2.1 节），可以确保在针对列式访问进行优化的分支中，我们能够安全地假定数组严格处于列式状态。因此，我们将此分支及其对应的循环称为列式分支。另一个分支及其循环则称为通用分支，因为其中既包含列式访问，也包含非列式访问。

GraalVM 编译器已经支持对循环节点进行复制。此过程还会在循环之后合并控制流，并为依赖于所使用循环的值引入 ϕ 节点。因此，合并后的中间表示节点不会受到复制的影响。最后，我们将复制后的循环附加到 <是否为列式>检查对应的分支上。

4.2.3 复制后的循环的优化

我们方法的主要目标是优化列式分支。如前所述，我们知道，在这一版本的循环中，数组始终是列式的。因此，复制后，我们可以移除冗余指令，例如数组状态检查（由 <是否为列式> 内建函数标记），以及对列式数组传递性成立的其他检查（<传递性>），例如检查从数组中加载的对象是否确实是代理（参见图 5 （**d**））。由于通用分支不局限于单一数组状态，因此在此阶段无法对其进行显著优化。不过，在第4.3节中，我们将介绍一种改进方法，使该分支也能够在一定程度上得到优化。

循环不变代码外提 循环不变代码外提 [5] 是 GraalVM 编译器已采用的一种编译器优化，旨在将循环体中的指令移至循环之前，前提是这些指令不受循环执行的影响。因此，它可以避免冗余的内存访问和计算；这些操作可以在循环开始前执行一次，而不必在每次循环迭代中重复执行。通过上述优化，我们可以使访问列式数组的大部分开销成为循环不变项。因此，大多数操作都可以移出列式循环，通常只剩下属性数组访问（即访问实际数据，如图 5（**e**）所示）。

启用循环向量化 由于以索引访问取代字段访问是列式数组固有的特性，因此这类数据结构的访问非常适合循环向量化 [15]。循环向量化是指利用硬件支持的 SIMD 指令 [43, 44] 来优化循环的过程。通过合并各次循环迭代，循环体中包含算术运算或条件判断的某些模式可以改写为并行执行多个操作（例如，使用一条 256 位向量指令，即可一次性修改、加载或写入八个连续存储的 32 位整数值）。

由于 GraalVM 编译器已经支持循环向量化 [24], ，我们的挑战在于确保涉及列式数组的循环具有适当的形式，使其

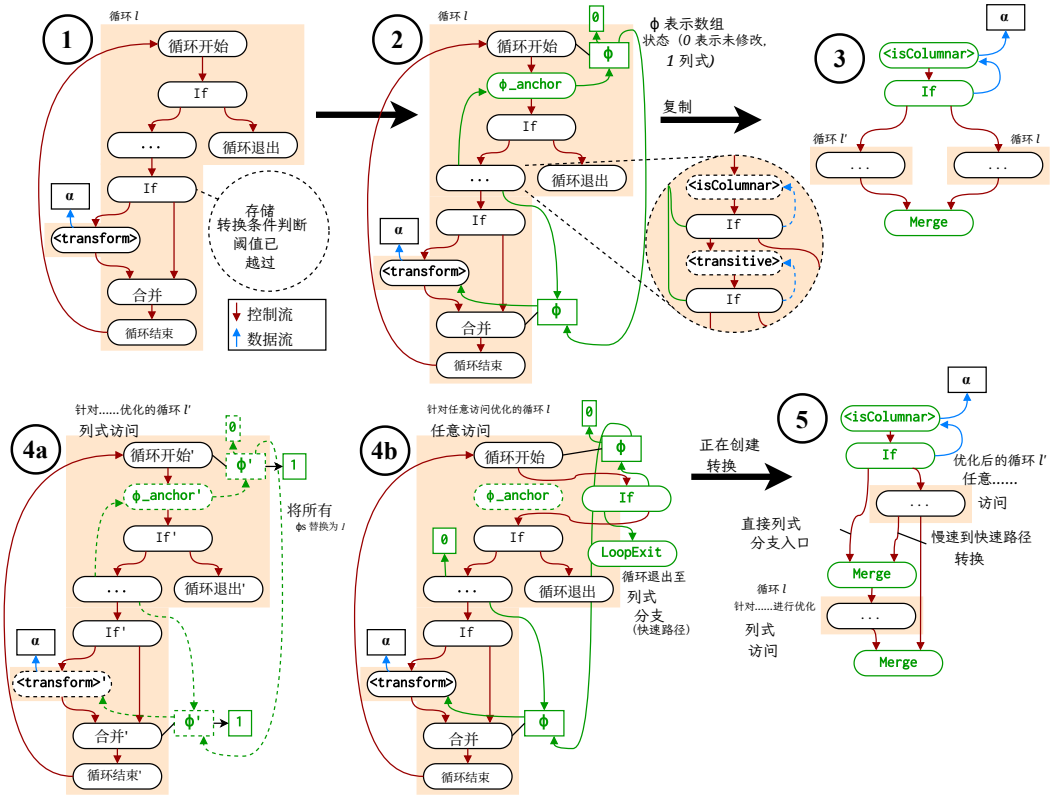


图 7 慢速路径到快速路径的转换与通用分支优化

随后能够被向量化器识别。如前所述，我们自定义的编译器阶段会优化循环体，使其通常只剩下属性数组访问。因此，对于图 5 (e) 中的查询，可以将属性数组上的某些操作替换为 SIMD 指令，从而进一步提升性能（参见第5.1节）。

4.3 改进：慢速路径到快速路径的转换

迄今为止，我们的复制方法存在一个问题：如果优化方法主要处理仅在该方法自身的循环中进行变换的数组，那么它就无法带来任何收益。考虑在一个仅包含单个循环的方法中进行复制，该循环与图 6 右侧的循环类似：如果调用该方法时传入的数组尚未处于列式形态，就会执行通用分支。在循环中，如果访问次数越过阈值，数组可能会被转换为列式布局；然而，执行仍会继续进入通用循环。由于优化仅限于列式分支，即使数组完成转换，它也无法从复制中获益，因为整个执行过程始终都在通用分支中。

基于这一观察，我们改进了复制方法，使其能够在数组经过变换后，*a*) 转换到列式（快速）分支

同时在通用分支中执行，并对通用分支本身进行一些有限但有效的优化，*b*)。

图 7 展示了这种新复制方法的主要步骤：

1. 第 4.2.1 节中的分析步骤根据收集到的数组指标和循环的形状，选择并优先处理合适的循环。
2. 我们添加一个新的循环 ϕ 节点。它的二进制值表示循环中每个位置上目标数组的状态（0 表示非列式，1 表示列式）。我们将这个新 ϕ 节点连接到控制流中的一个自定义节点，并称之为 ϕ 锚点。该节点唯一的作用是标记循环的起点，并捕获循环迭代开始时的 ϕ值。此过程的后续步骤会将其移除。然后，我们按照支配者顺序（参见第 2.1 节）遍历循环的节点，同时持续跟踪数组已知的最后状态。

   a) 我们将数组上的自定义检查内建函数（例如 <检查>、 <传递性>；参见第 4.1 节）替换为已知的最新状态。b) 遇到该数组的转换调用（<变换>）时，我们将此函数的返回值用作新的数组状态（即，如果转换成功，返回值为 1，否则为0）。c) 在合并点，如果一条或多条路径中发生了转换，我们会创建额外的 ϕ 节点来合并输入状态。d) 在每个循环末尾，最新状态都会传播到初始循环 ϕ。

3. 我们复制该循环，并插入预先检查和分支条件。
4. 随后分别优化生成的分支：

   a) 在列式分支循环l'中，所有表示数组状态的 ϕ 节点都会被替换为常量 1。之后，内置优化阶段会优化现已成为常量的条件、移除冗余分支等。b) 在通用分支循环l中，我们将 ϕ 锚点替换为新的条件判断条件，用于检查此处 ϕ 节点的值；如果数组已变为列式，则退出循环。否则（ ϕ 值为0），继续执行通用循环。在这种情况下，所有对 ϕ 锚点的使用都会被替换为常量0，从而表示如果循环继续，数组就不可能是列式的。随后便可优化常量条件。循环中剩余的 ϕ 合并节点表示数组状态仍不确定的情况。

5. 经过单独优化后，从通用循环中新引入的循环退出点会与列式分支中循环头之前的节点合并，从而实现从通用分支到列式分支的转换。此外，还必须修补列式循环中现有的循环 ϕ 节点，使其在通用分支此前已执行的情况下能够复用其中累积的 ϕ 值（例如，如果使用了计数器，列式循环就应复用通用分支中的最后一个计数器值，而不是将其重置为初始值）。

该算法对包含多个列式格式数组的循环采用类似的处理方式。在这种情况下，我们会跟踪所有这些数组的状态，并且只有在所有数组都为列式格式时，才允许从慢速路径切换到快速路径。

5 评估

与 Makor 等人 [55], 的工作相比，我们希望通过限制重新编译开销，尤其提升迭代次数较少的小型工作负载的性能。我们的多级存储转换方法应能提升带有 日期属性的对象数组的性能。我们还希望不会给不适用于该方法的工作负载带来显著开销。

由于我们的方法集成到了 GraalVM JavaScript 运行时中，因此还必须确保符合标准。我们的方法通过了 V8 和 ECMAScript 测试套件 [34], , 两者合计包含超过 51000 个测试用例。在这些测试运行期间，我们大幅降低了转换阈值，以强制对较小的数组执行存储转换。总的来说，我们尝试对数组进行约 450000 次转换，其中约 1000 次成功。不过，这些数组中约有 750 个后来不得不恢复，因为执行过程中遇到了不受支持的操作。

遗憾的是，JavaScript 中缺乏针对大型数组的数据密集型基准测试，而传统的 JavaScript 基准测试套件（例如 Octane [16], 、JetStream [36]）通常不包含我们的方法所针对的工作负载。因此，我们复用了 [55] 中的微基准测试[1] 和 TPC-H [23] 移植版本，并对其进行了扩展。

在附录 D 中，我们评估了第 4.3 节引入的慢速路径到快速路径的转换。在附录 E 中，我们评估了转换本身的开销。在附录 F 中，我们还表明，对于不包含适用数组的工作负载，我们的方法不会造成显著开销。本文提出的方法基于较新版本的 GraalVM，因此我们必须移植 Makor 等人 [55] 的方法，才能进行有效比较。因此，基准测试得分可能与他们的评估结果略有不同。在所有基准测试中，我们将数组的最小大小设为 5000，并将数组符合转换条件所需的最小读取次数设为 2500。与 [55] 中的原始版本相比，这些阈值更低，因为复制方法能够更快地改善较小数组的性能。

除非另有说明，基准测试均在一台双插槽 Intel(R) Xeon(R) CPU E5-2690 @ 8×2.90GHz（32 个逻辑核心、192G 可用内存）上执行。我们的方法基于 GraalVM EE 22.3 版本。为确保充分预热，我们在测量前多次重复执行每项基准测试。

5.1 微基准测试

图 8 展示了我们的方法在一组微基准测试中的性能，并与 [55]以及未进行存储转换的 GraalVM JavaScript 进行了比较。每项基准测试都会在一个大型对象数组上执行特定查询。图中结果的差异体现在查询应用于同一数组的次数（100、1000）以及生成数组的大小（10k、1M）。附录 B 给出了其他配置下的结果，并更详细地描述了各项微基准测试。结果凸显了复制的关键优势：与 [55], 相比，我们的方法

[1] 代码：https://github.com/lmakor-jku/data-intensive-js-benchmarks （访问于 2022-12-30 ）.

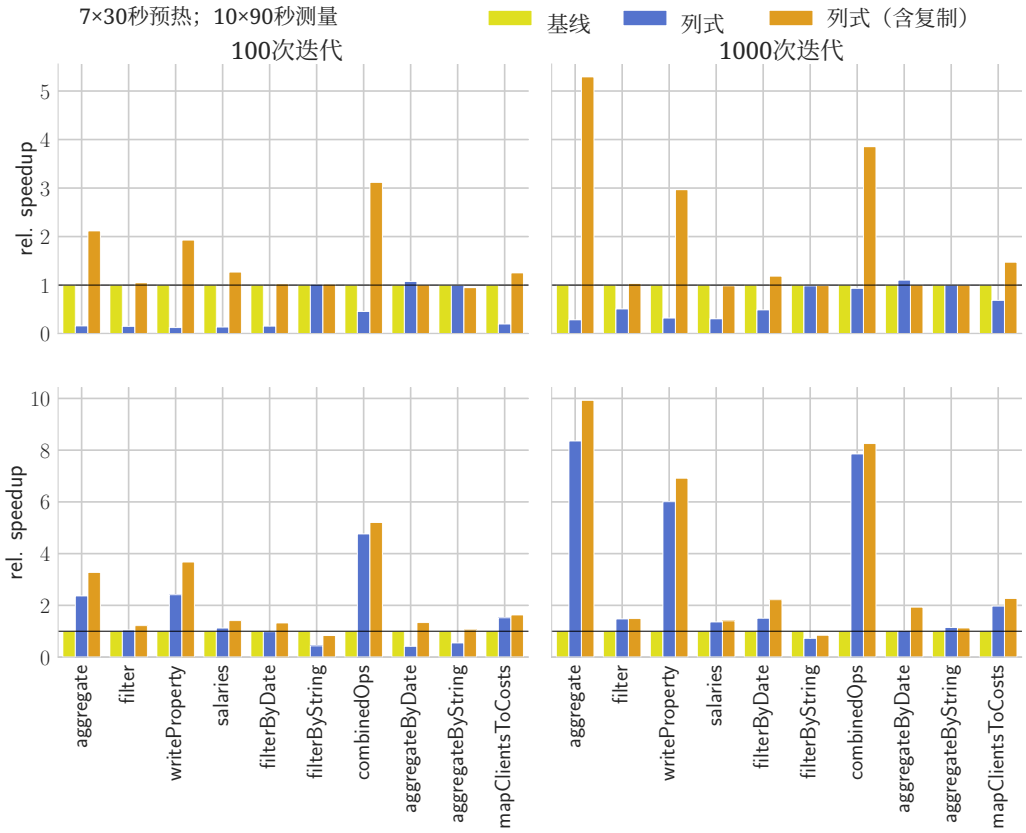
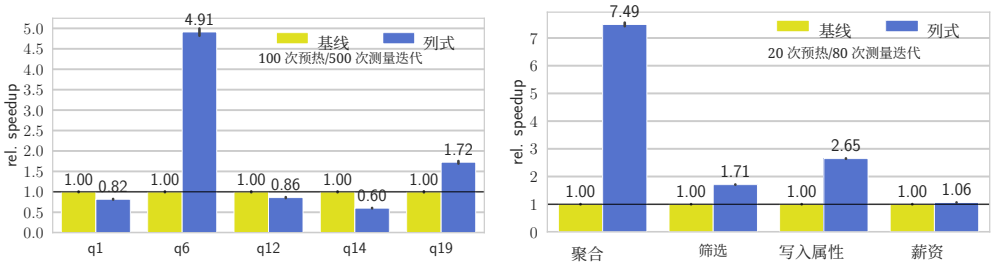


图8 与未进行存储转换的［55］和 GraalVM JavaScript 相比，我们的方法的微基准测试吞吐量 （越高越好）

如今即使在所展示的最小配置下，也已能在某些查询仅迭代少数几次时实现加速（执行 100 次迭代时，聚合 和 写入属性 约为 2 倍）。筛选/聚合 的日期/字符串变体包含涉及嵌套对象的工作负载。结果表明，我们针对日期进行的多级存储转换，如何使涉及日期的查询实现加速。对于更大的数组和更多的迭代次数，聚合和 写入属性 分别实现了约 10 倍和 7 倍的加速，而 [55] 中介绍的方法分别为 8 倍和 6 倍。*mapClientsToCosts*的列式变体采用了 SIMD 向量化，加速最高可达 2 倍（参见第4.2.3节）。

5.2 TPC-H

TPC-H 决策支持基准测试 [23]是一套常用于衡量数据库性能的基准测试。它包含针对大型数据集的多种查询，似乎也适合用于评估我们的方法。遗憾的是，这套基准测试仅提供 SQL 格式（ [56]也指出了这一问题），因此我们选择使用与 [55] 相同的定制JavaScript移植版本。这些移植版本包含 TPC-H 22 个查询中的 5 个：*q1*、*q6*、*q12*、*q14* 和 *q19*。附录C详细说明了选择这些特定查询的原因；这一选择基于各个查询的属性，旨在展示我们的方法在使用不同查询操作和特性时的性能。



(a) TPC-H **(b)** Lodash

■ **图9** 采用我们的方法相对于未进行存储转换的基线，在不同基准测试中的查询性能（越高越好）

■ **表2** 性能 缓存未命中评估

指标	基线计数	列式计数	减少量
缓存未命中	2 575 530 996 544.6	50 406 360 352.9	51.1x
L1 数据缓存加载未命中	3 142 811 326 847.5	137 072 211 496.1	22.9x
末级缓存加载未命中	520 871 505 214.4	530 809 018.5	981.3倍

图 9a 展示了此次评估的结果。与在标准 GraalVM JavaScript 上的执行相比，查询*q1*、*q12*和*q14* 在我们的方法上表现不佳。这是因为它们更多地使用了分组和字符串函数，而这些并未受到我们的优化影响。此外，它们经常包含无法被 GraalVM 的部分求值器完美内联的操作，因此会产生方法调用，阻止我们的复制机制将该查询纳入考虑范围。然而，查询*q6* 和*q19*的性能有所提升，其中*q6* 的性能提升尤为显著（接近 5 倍）。

5.3 Lodash

对于图9b中的结果，我们在一系列基于微基准测试的 Lodash [26]查询上执行了我们的方法（参见第5.1节），并将其性能与标准 GraalVM JavaScript 的执行性能进行了比较。如图所示，我们的方法显著提升了大多数查询的速度，而薪资 仅有轻微改善。

5.4 缓存行为

我们使用*perf stat*评估了我们的方法与未进行存储转换的 GraalVM JavaScript 相比的缓存行为。我们执行了具有 5M 数组大小和 4000 次迭代的聚合 微基准测试，以确保编译器线程不会影响计数器。列式布局的主要目标之一是借助属性数组实现更好的数据局部性。因此，我们预计，使用列式布局数组的应用程序比使用普通数组的应用程序产生更少的缓存未命中。表2列出了此次评估中不同*perf* 事件的结果。列式变体的缓存未命中显著更少。受硬件限制，

此次评估是在单插槽 Intel(R) Core(TM) i7-1065G7 CPU @ 1.30 GHz 上进行的，该处理器具有 8 个逻辑核心、32 G 可用内存，以及 4×192 KiB L1d、 4×128KiB L1i、 4×2 MiB L2 和 1×8MiB L3 缓存。

6 局限性

应用循环复制会带来编译时和代码大小方面的成本。因此，查询复制只应应用于能够通过复制进行优化的查询。同样，应用多级存储转换也需要谨慎考虑，因为它会产生更多的转换开销。

查询分析 我们的循环分析会选择那些包含列式数组访问、且不会将数组恢复为原始格式的循环。更复杂的分析（例如识别查询类型，如更新、映射或过滤查询）可以支持更有针对性的优化。作出这一决定时，必须考虑到，未经优化的列式存储访问明显慢于常见的属性访问。

通用多级存储转换 目前，我们仅针对日期 对象实现了多级存储转换。因此，最多只需再应用一个转换层级。若要对任意类型进行多级存储转换，就意味着可能存在任意层级的嵌套对象，并且需要对其进行处理。此外，通用多级转换还必须考虑对象层次结构可能存在循环。对此，一种解决方案是限制要转换的层级数。

7 相关工作

我们将与本研究相关的工作分为三个不同类别：a）列式和以内存布局为中心的数据结构研究，b）查询优化技术，c）相关的编译器优化。

7.1 列式数据结构

虽然有多种方法将列式数据结构集成到常见的语言运行时中，但其中大多数都针对现有的列式数据进行操作，或为此类数据结构提供 API。由于采用静态数据布局，无需进行运行时转换，因此开发者可以专门将这些方法用于高性能应用。另一个优势是，由于数据布局在运行时不会发生剧烈变化，编译器通常可以自动优化对这类集合的访问。与我们的方法相比，这些方法的缺点是需要开发者介入才能使用列式存储。据我们所知，目前还没有其他研究提出像我们这样自动进行运行时存储转换。

数据分析与可视化库（例如 *Apache Arrow* [7] 和 *RAPIDS* [75]）也支持 JavaScript 和其他动态语言中的列式数据结构。数据框 支持在 Python [80]和 Julia [46]中使用表格布局。Julia 还支持*StructArray* 类型，为结构体数组 [45]提供了一种抽象。

Mattis 等人 [56]为 Python 实现了列式 API。与我们的方法类似，他们将列式数组的元素表示为代理对象。PyPy 运行时的即时编译器可以利用列式数据布局来提升性能，而无需修改编译器。他们通过在访问元素时生成新的代理，来限制代理的生命周期。这样，即时编译器便能在某些情况下完全避免代理分配。

Pivarski 等人 [71] 在抽象语法树层面执行代码转换，以便快速访问已有列式布局的 数据集。这样便可以通过传统的面向对象 API 访问此类数据，同时避免将原本以列式形式存储的数据物化为对象。我们有着共同目标：将优化数据访问的职责转交给运行时和编译器。不过，他们处理的数据本身已经是列式形式，而且并不致力于自动转换非列式数据。

Noth [64] 提出了拆解式 Java 对象，其中字段值存储在并行数组中，以缩减内存占用，并通过提高缓存利用率来提升性能。拆解式对象与我们的代理对象相似，重要区别在于：代理始终属于某个（列式）数组，而拆解式对象则是独立的。因此，拆解式对象的字段值存储在全局并行数组中，而不是各个数组专属的属性数组中。与我们的方法不同，他们采用 Java 源到源转换，由用户标注会生成拆解式对象的类。这样，他们无需修改 JVM，便能在拆解式对象上支持大多数 Java 语义。我们的方法需要修改语言实现和编译器，但能够自动转换数组。

De Wael 等人 [28] 为即时数据结构实现了一个 API。这类数据结构由开发者通过自定义领域专用语言定义，并包含转换规则，以便在运行时更改底层数据布局。虽然这意味着开发者必须了解特定场景下的最佳数据布局，但他们还提出了一种利用机器学习的方法，根据执行时间自动选择最佳数据布局。

7.2 查询优化

Zhang 等人 [90] 为用户代码中由所谓的编译指示标记的循环生成高效的查询执行计划。他们使用 Truffle 框架从循环生成抽象语法树，随后应用不同的查询计划。通过在线和离线分析评估这些计划，以选出性能最佳的计划。由于我们在编译期间应用查询优化，因此不打算增加额外的评估步骤，以免进一步影响编译时间。在我们的方法中，用户无需标记或预先选择要优化的查询；只要使用列式数组，这些查询就会自动得到优化。

Shaikhha 等人 [76]采取了不同的方向：他们接收一个针对列式存储的查询，随后在一种

高级语言中执行类似编译器的优化。这包括多种策略，用于物化列式存储中与查询相对应的行。他们提出的许多查询优化已由 GraalVM 编译器隐式执行。在物化方面，我们的目标似乎相似：他们试图避免不必要的行物化，而我们则试图消除冗余的数组元素访问。Chen 等人 [18] 提出了类似的方法，他们对从数据库查询生成的中间表示执行了多种编译器优化。Grulich 等人 [40] 也对中间表示进行优化，但借助 Truffle 框架，支持查询与用户定义函数混合的多语言执行。由于这些方法的输入都是专用查询，因此可以根据查询类型应用更具针对性的优化。相反，它们并不旨在改变底层数据存储——而这正是我们方法中的关键一步。

7.3 编译器优化

在编译器中使用复制通常是在代码大小与后续优化可能带来的预期收益之间权衡。尽管这一概念早在 20 多年前就已成功应用 [61, 62], ，但近来也有一些与我们的方法存在一定关联的研究。

Zhao 等人 [91] 使用复制来并行化 JVM 中的循环。他们检测适用循环的方法与我们的循环分析相似，但目标和筛选标准不同：他们检查循环的形态，即退出条件和参数形态；而我们则利用语言实现中引入的内建函数来寻找目标。同样，他们会在复制后的循环前添加预检查，以便在其中一个循环中进行边界检查消除等优化。由于他们考虑的是数组的静态属性（即其长度），因此无需担心中间状态变化。我们会在循环前检查数组状态以进行区分，但该状态可能在循环中发生变化。因此，我们引入了从慢速路径到快速路径的转换（参见第4.3节）。

Leopoldseder 等人 [53] 在 GraalVM 编译器中使用模拟 ，以预测控制流合流后将代码复制到各个分支所带来的收益和成本。到目前为止，我们的复制决策纯粹基于前述循环分析步骤。不过，我们认为，使用模拟还能进一步改进我们的方法，从而评估复制及后续优化的影响。

循环优化在 PyPy [8, 9, 72],V8 [82], HotSpot C2 服务端编译器 [68], 和 GraalVM 编译器 [24, 52]等现代编译器中十分常见。循环剥离 [10]和循环展开 [27]等技术利用复制来复现循环体。它们通过展开单次循环迭代来简化循环，或促成并行化等后续优化。

在我们的方法中，复制可确保生成两个不同的循环，每个变体负责处理处于特定状态的数组。与传统的循环优化不同，这一过程会通过专门查询语言实现中的内建函数，考虑列式数组是否以及如何在循环中被修改或访问。由于我们的列式数组仅在运行时生成，无法在静态阶段确定（也就是说，编译器无法自动推断某个数组是列式数组），因此必须采用这一要求来缩小可进行复制的候选范围。

我们的方法中的优化同样由内建函数驱动，因为它们能揭示编译器原本无法利用的优化潜力，例如在已知数组为列式数组的情况下移除检查，并将属性访问与数组元素访问解耦。随后，循环不变代码外提 [5] 会将许多剩余操作移出循环。理想情况下，列式循环变体随后可与针对静态列式数组的循环编译结果相媲美，而另一个变体则包含对数组（无论是列式还是非列式）的通用访问。

元追踪 Python 即时编译器 PyPy [81] 在保护失败反复发生时会生成所谓的桥接代码 [8]。这意味着需要从发生失败的保护条件处编译追踪，而该追踪最终可能跳回包含该保护条件的已编译追踪。使用列式数组时，如果数组的状态频繁变化（从而导致保护失败），我们就可能创建一个具有新状态的桥接代码。根据所采用的优化和桥接代码的长度，最终生成的已编译追踪可能类似于我们复制方法中的分支。不过，由于 GraalVM 编译器是基于方法的，（元）追踪即时编译器技术并不适用。

8 结论

本文介绍了一种方法，用于检测表示对潜在列式数组执行类查询操作的循环，并在先进的即时编译器中通过复制来优化此类循环。具体而言，复制后的循环会分别进行优化：通用循环（慢速路径）处理对列式和非列式存储格式的访问，而复制后的循环仅包含对列式存储的访问（快速路径）。由于在我们的方法中，数组可能会在查询执行期间转换为列式数组，我们还实现了从慢速路径到快速路径的转换，使这类数组也能从列式访问优化中受益。为了加速处理引用类型属性的查询，我们引入了一种多级存储转换，不仅以列式形式存储对象的属性，还将整个对象层次结构存储在属性数组中。我们以日期 属性为例实现了这项技术，作为一项案例研究。

对微基准测试以及广泛使用的数据库查询移植版本的评估表明，我们的方法确实能够提升针对列式数组的查询性能，但在使用“分组”操作时仍存在问题。对于适用的查询，我们的方法还显著改善了缓存行为。此外，我们的结果表明，多级存储转换也能提升使用诸如日期 比较等操作的查询。总的来说，我们的研究或许能启发研究员探索在编程语言中运用列式数据结构的不同方式。

致谢 作者 Sebastian Kloibhofer 和 Lukas Makor 对本文的贡献相同。本研究项目部分由 Oracle Labs 资助。我们感谢 Oracle Labs 虚拟机研究组的全体成员。我们也感谢林茨约翰·开普勒大学系统软件研究所的所有研究员对我们工作的支持并提出宝贵意见。

A 多级存储转换的特殊情况

为了保持程序语义，第 3 节介绍的多级存储转换必须能够处理若干不同的特殊情况。这些情况都源于这样一个事实：经过转换的嵌套对象属性可能引用对象，而这些对象也可能在其他地方被引用。

一个数组中包含一个嵌套对象的对象这是默认情况：一个嵌套对象（例如日期）与某个对象相关联，而该对象是数组的一部分。转换后，嵌套代理对象必须记住引用它的属性名称（例如“lastChanged”），以便获取属性数组的标识符、它所属对象的索引以及变换后的数组本身。随后，便可通过解引用该数组，并在给定索引位置访问具有指定名称的属性数组，从而访问嵌套对象的属性值。

一个对象的多个属性引用同一个嵌套对象同一个嵌套对象也可能被一个对象通过多个不同的属性引用（例如，lastChanged 和deletedAt 属性引用同一个日期对象）。在这种情况下，转换会将该对象的值添加到两个不同的属性数组中，每个属性对应一个。转换后，我们必须确保对其中一个属性数组的任何写入也会更新另一个数组（例如，对lastChanged.timestamp 属性数组的更新也必须应用于deletedAt.timestamp属性数组）。

同一对象在多个（列式/非列式）数组中包含嵌套对象 如果某个带有嵌套对象属性的对象出现在多个数组中，那么对与嵌套代理对象关联的属性数组进行的写入，也必须复制到所有其他列式数组中。对于非列式数组，我们已经通过使用前文提到的数组、索引和属性名称三元组，确保对嵌套对象属性的访问会被正确地重新指向某个属性数组。

同一对象在同一数组中多次包含嵌套对象 类似地，一个带有嵌套对象属性的对象可能会在同一数组中的不同索引位置出现多次。在这种情况下，对与嵌套代理对象关联的属性数组进行的写入，必须反映到每一个索引位置上——例如，在索引i 处写入lastChanged.timestamp 属性时，如果数组在i和j两个位置包含同一对象，那么索引 j 处的值也必须改变。

多个对象在同一数组中多次引用同一嵌套对象 最后，同一数组中的多个不同对象仍可能引用同一个嵌套对象。这是前述情况的更一般形式；同样，对该嵌套对象的属性数组中某个位置的写入，可能也必须应用到其他位置。

Makor 等人 [55] 已经指出，将数组转换为列式格式可能会导致问题，例如对象同时属于多个数组的情况。他们

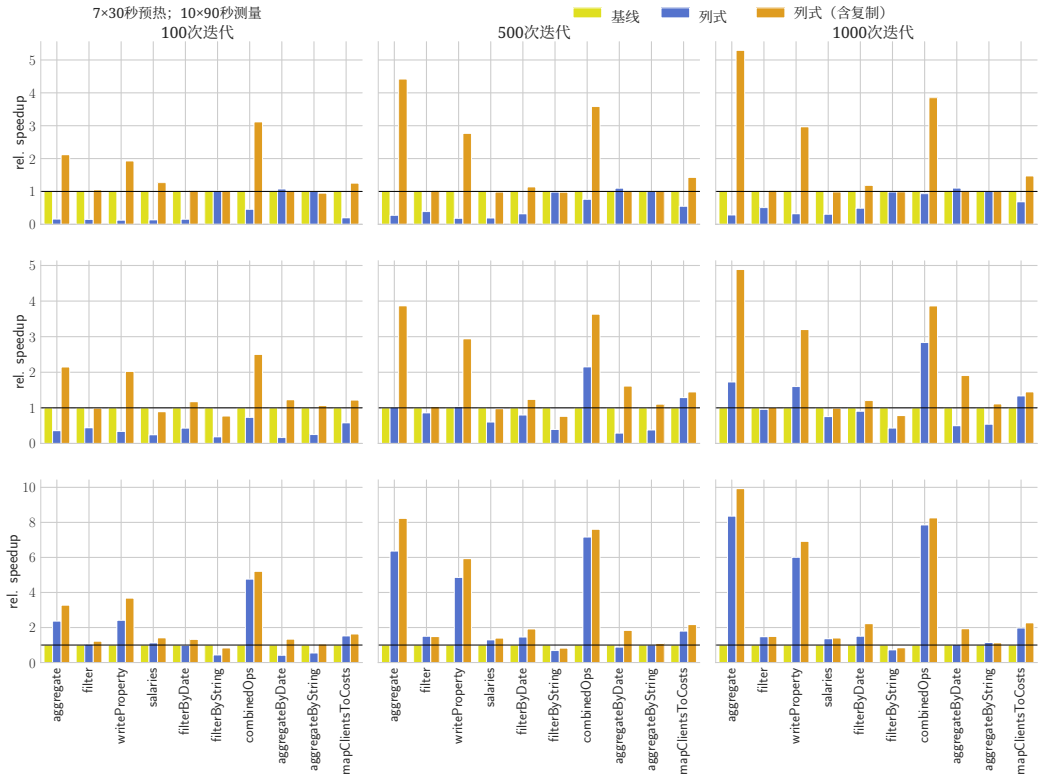


图10 我们的方法与 [55] 以及未进行存储转换的 GraalVM JavaScript 的微基准测试吞吐量对比 （越高越好）

通过为每个转换后的代理对象关联一组数组-索引对，解决了这些问题，其中每一对都标识一个位置，程序可能会从该位置访问相应的代理（或其属性）。我们调整了这一解决方案，以处理上述情况。然而，要访问属性数组，嵌套代理对象需要三项信息：存储属性数组的关联列式数组、引用作为属性数组标识符的嵌套对象的属性名称，以及该对象在列式数组中的索引；该索引还标识嵌套对象属性值在属性数组中的位置。因此，每个嵌套代理对象都必须管理一组三元组。对嵌套代理对象的属性进行写访问时，会遍历这组数据，对每个关联的属性数组执行写入。因此，只要一个嵌套对象被多次引用，写入操作就会变得昂贵。

B 微基准测试详情

图10展示了我们在 9 种不同配置下进行微基准测试的结果。如第5.1节所述，复制可以在迭代次数较少的较小数组上带来加速比，同时也能提升大多数基准测试的整体性能。

■ **表 3** 我们的微基准测试描述

查询	说明
聚合	对单个属性的值求和。
筛选	根据单个属性筛选数组。
写入属性	计算并更新单个 基于另一个属性的值。
薪资	计算并更新……的奖金和工资 根据员工获得的薪资，同时 累加所有新薪资。
filterByDate	基于日期比较进行筛选。
filterByString	根据字符串比较进行筛选。
组合操作	连续执行聚合、筛选 和 写入属性 操作。
按日期聚合	对单个属性的值求和后 根据日期比较进行筛选。
通过字符串相等进行聚合	在……之后对单个属性的值求和 根据字符串比较进行筛选。
mapClientsToCosts	根据客户邮政编码计算运费 代码。可以采用列式格式进行向量化。

我们目前用于评估的微基准测试集合包含 10 个不同的查询，旨在展示大型数组上的不同操作。所有微基准测试都采用清单 1 中所示的结构：生成一个给定大小的数组，并用统一对象填充。请注意，不同的基准测试可能使用具有不同属性的对象（例如，filterByString 使用具有字符串属性的对象，而聚合 仅使用具有基本类型属性的对象）。随后，在该数组上重复执行给定的查询 （例如，聚合、写入属性）。我们确保编译器无法丢弃中间结果或数组本身。表 3 对各个查询进行了更详细的描述。

■ **清单 1** 我们的微基准测试基本模板

```
1 函数benchmark(arraySize, queryApplications) {2 // 创建一
个
cr
5
Q
```

■ **表 4** 部分 TPC-H 查询的属性

查询	分组 BY	排序 BY	AGG	子查询	字符串 OP	DATE OP
q1	x	x	x			x
q2		x	x*	x	x	
q3	x	x	x		x	x
q4	x	x	x	x		x
q5	x	x	x		x	x
q6			x			x
q7	x	x	x	x	x*	x*
q8	x	x	x	x	x	x*
q9	x	x	x	x	x*	x
q10	x	x	x		x	x
q11	x**	x	x	x	x	
q12	x	x	x		x	x
q13	x	x	x	x	x*	
q14			x		x	x
*q15****		x	x*	x		x*
q16	x	x	x	x	x	
q17			x	x	x	
q18	x	x	x	x	x	x
q19			x		x	
q20		x	x*	x	x	x*
q21	x	x	x	x	x	x
q22	x	x	x	x		x

* 在子查询或视图中具有属性 ** 包含 HAVING 子句 *** 创建视图 日期/字符串操作，包括部分提取（提取（year. .. ））和排序

C TPC-H 查询选择

如第 5.2 节所述，我们在一部分 TPC-H 查询的 JavaScript 移植版本上评估了我们的方法。我们有意选择这些查询，以便在具有不同属性的一组查询上评估我们的方法。这些移植版本只使用传统的 JavaScript 结构，不使用外部库。对于涉及分组或字符串操作的复杂查询，我们的方法表现不佳，因为这类查询通常会产生复杂的循环和嵌套方法调用，从而阻碍复制。

表 4 对所有TPC-H 查询进行了高层次分析（请注意，各个查询变体之间可能存在细微差异），说明某些查询是否使用了特定的查询特性（Boncz 等人 [12]也对 TPC-H 查询集进行了更详细的分析，并同样强调了分组和聚合的广泛使用）。此外，该表还突出显示了那些被

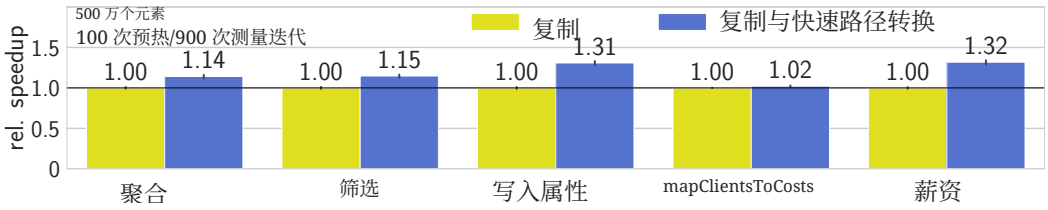


图 11 相对于未进行存储转换的 GraalVM JavaScript，慢速路径到快速路径的转换的结果 （越高越好）

选用于评估的查询（红色表示我们的方法性能较差，绿色表示该方法提升了性能）。*q6* 和*q19* 属于复杂度最低的查询之列，既不包含分组或排序函数，也不使用子查询或（复杂的）字符串操作——与*q1* 和*q12* 形成对比，后两者涵盖了多种不同的查询特性。虽然*q14* 看起来也相对简单，但它包含使用LIKE 运算符的字符串操作，因此需要比*q19* 更复杂的字符串比较，后者只检查是否完全匹配。这一差异十分显著，因为*q14* 中的比较会在编译时生成复杂得多的中间表示，其中还包含部分求值器无法内联的方法调用。由于其他所有 TPC-H 查询也具有类似属性（大多数使用分组、排序和子查询），我们预计这些查询也会得到同样较差的结果。

D 慢速路径到快速路径的转换

为评估我们在第4.3节提出的改进，我们使用大小为 5M 的数组运行了部分微基准测试。由于只有当方法处理在该方法内经过变换的数组时，转换带来的性能提升才会显现，因此在每项基准测试中，我们都会初始化这样一个数组，然后执行一次查询方法（例如，聚合、筛选）一次。整个过程重复了 1000 次，前 100次迭代用于预热。我们只测量查询方法的执行时间，否则数组初始化会使结果产生偏差。图 11 显示，大多数查询确实有所改进。

E 存储转换开销

尤其是在较小的工作负载下，我们的方法所产生的转换成本会显著影响查询性能。为了测量这项开销，我们在包含 1M 个对象的数组上，以不同数量的查询应用（1、100、1000）运行了部分微基准测试。因此，对于每种基准测试配置，数组都只转换为列式数组一次。随后，我们测量了转换时间以及查询总执行时间。表5评估了部分微基准测试中的转换成本。结果表明，对于简单查询（例如 聚合），转换成本占总执行时间的很大一部分

■ **表5** 转换开销

查询	迭代次数 (#)	总计 (ms)	查询 (ms)	转换 (ms)	(%)
聚合	1	1300	645	655	50.4
	100	1653	991	662	40.0
	1000	2202	1521	681	30.9
筛选	1	1717	1056	661	38.5
	100	4630	3959	671	14.5
	1000	22958	22296	662	2.9
写入属性	1	1488	907	581	39.0
	100	1944	1362	582	29.9
	1000	3060	2517	543	17.7
薪资	1	2061	1554	507	24.6
	100	3454	2925	529	15.3
	1000	11240	10716	524	4.7

不过，随着查询应用数量增加，这部分成本会逐渐摊薄。

F Octane 基准测试详情

图 12 展示了我们的方法在不包含适用数组的工作负载上的表现。为此，我们选择了 Octane [16] 基准测试套件。虽然该套件已正式停用，但仍经常用于编译器评估。结果表明，对于常见工作负载，我们的方法在性能、内存使用量和已安装代码大小方面不会造成显著开销。唯一的例外是*splay*基准测试：我们的方法会使代码大小增加约 45%，编译时增加约 35 %，但不会影响性能。

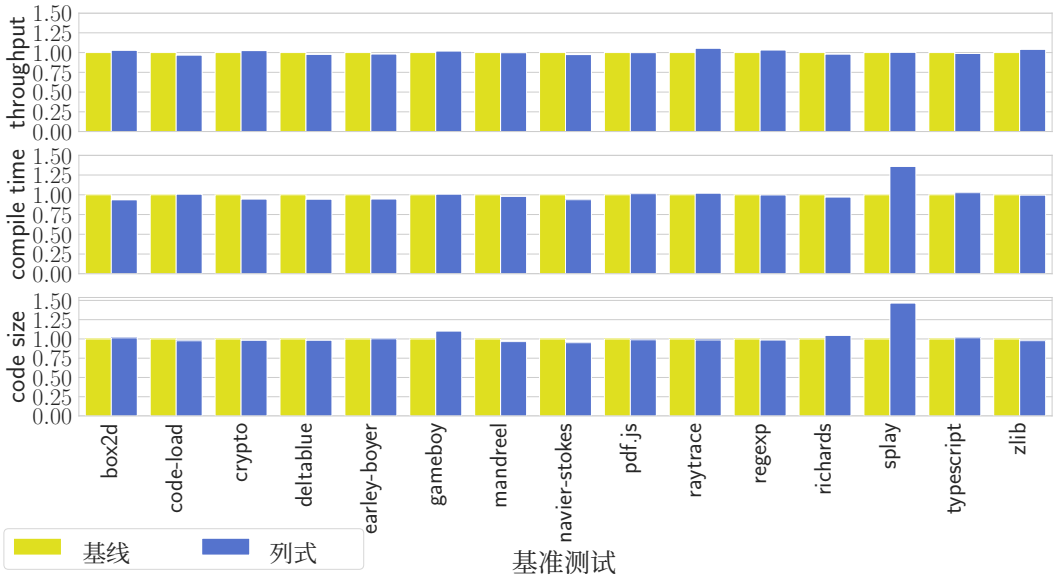


图 12 Octane 基准测试性能：吞吐量（越高越好）、编译时（越低越好）、代码大小（越小越好）

参考文献

[1] Daniel Abadi、Peter Boncz、Stavros Harizopoulos 和 Samuel Madden。“现代列式数据库系统的设计与实现”。载于：*Foundations and Trends® in Databases*（2013年），第 197–280 页。ISSN：1931-7883。DOI：10.1561/1900000024。[2] Daniel Abadi、Samuel Madden 和 Miguel Ferreira。“在列式数据库系统中集成压缩与执行”。载于：*2006*年 *ACM SIGMOD* 数据管理国际会议论文集。SIGMOD ’06。美国纽约州纽约：ACM，2006年 27日，第 671–682 页。ISBN：978-1-59593- 434-5。DOI：10.1145/1142473.1142548。[3] Daniel Abadi、Daniel Myers、David DeWitt 和 Samuel Madden。“列式数据库管理系统中的物化策略”。载于：*2007 IEEE* 第 *23* 届数据工程国际会议。2007 IEEE 第 23 届数据工程国际会议。土耳其伊斯坦布尔：IEEE，2007年，第 466–475 页。DOI：10.1109/ICDE.2007.367892。[4] Daniel J. Abadi、Peter A. Boncz 和 Stavros Harizopoulos。“列式数据库系统”。载于：*Proc. VLDBEndow.*（2009年），第 1664–1665 页。ISSN：2150-8097。DOI：10.14778/1687553.1687625。[5] Alfred V. Aho、Ravi Sethi 和 Jeffrey D. Ullman。编译器：原理、技术与工具。马萨诸塞州雷丁：Addison-Wesley 出版公司，1986年。ISBN：978-0-201-10088-4。网址：https://suif.stanford.edu/dragonbook/（访问于 2022-12-30）。[6] Frances E. Allen。“控制流分析”。载于：*SIGPLAN Not.*（1，1970年），第 1– 19 页。ISSN：0362-1340。DOI：10.1145/390013.808479。

[7] Apache 软件基金会。*Apache Arrow*。Apache 软件基金会，6，2022年。网址：https://github.com/apache/arrow （访问于2022-07-07）。[8] Hakan Ardo。循环不变代码外提。PyPy。11，2011年。网址：
h
t
t
p
s://www.pypy.org/posts/2011/01/loop-invariant-code-motion-1998392217676829154.html（访问于2022-12-09）。[9] Håkan Ardö、Carl Friedrich Bolz 和 Maciej FijaBkowski。“PyPy 跟踪式即时编译器中的循环感知优化”。载于：*SIGPLAN Not.*（22，2012年），第63–72页。ISSN：0362-1340。DOI：10.1145/2480360.2384586。[10] David F. Bacon、Susan L. Graham 和 Oliver J. Sharp。“面向高性能计算的编译器转换”。载于：*ACM Comput. Surv.*（1，1994年），第345–420页。ISSN：0360-0300。DOI：10.1145/197405.197406。[11] Haoqiong Bian、Ying Yan、Wenbo Tao、Liang Jeff Chen、Yueguo Chen、Xiaoyong Du 和 Thomas Moscibroda。“基于列排序和复制的宽表布局优化”。载于：*2017*年 *ACM* 数据管理国际会议论文集。SIGMOD ’17。美国纽约州纽约：计算机协会，9，2017年，第299–314页。ISBN：978-1-4503-4197-4。DOI：10.1145/3035918.3035930。[12] Peter Boncz、Thomas Neumann 和 Orri Erling。“TPC-H 分析：一项颇具影响力的基准测试中的隐藏信息与经验教训”。载于：性能表征与基准测试。Raghunath Nambiar 和 Meikel Poess 编。《计算机科学讲义》。查姆：施普林格国际出版公司，2014年，第61–76页。ISBN：978-3-319-04936-6。DOI：10.1007/978-3-319-04936-6_5。[13] Peter A. Boncz、Marcin Zukowski 和 Niels Nes。“MonetDB/X100：超流水线查询执行”。载于：第二届创新数据系统研究双年会议（*CIDR 2005*），美国加利福尼亚州阿西洛马，*2005*年*1*月*4*日至*7*日，在线论文集。美国加利福尼亚州阿西洛马：www.cidrdb.org，2005年，第225–237页。网址：http://cidrdb.org/cidr2005/papers/P19.pdf （访问于2022-05-11）。[14] Mike Bostock。*D3.js* - 数据驱动文档。29，2022年。网址：https://d3js.org/（访问于2022年6月29日）。[15] David Callahan、Jack Dongarra 和 David Levine。“向量化编译器：测试套件与结果”。载于：*1988*年 *ACM/IEEE* 超级计算会议论文集。Supercomputing ’88。美国华盛顿特区：IEEE 计算机学会出版社，1，1988年，第98–105页。ISBN：978-0-8186-0882-7。DOI：10.1109/SUPERC.1988.44642。[16] Stefano Cazzulani。*Octane*：面向现代 *Web* 的 *JavaScript* 基准测试套件。Chromium 博客。2012年。网址：https://blog.chromium.org/2012/08/octane-javascript-benchmark-suite-for.html（访问于2022-09-22）。[17] Pohua P. Chang、Scott A. Mahlke 和 Wen-Mei W. Hwu。“利用配置文件信息辅助经典代码优化”。载于：软件：实践与经验（1991年），第1301–1321页。ISSN：1097-024X。DOI：10.1002/spe.4380211204。

[18] Hanfeng Chen、Joseph Vinish D'silva、Hongji Chen、Bettina Kemme 和 Laurie Hendren。"HorseIR：将数组编程语言与数据库查询处理融为一体"。载于：第*14*届 *ACM SIGPLAN* 动态语言国际研讨会论文集。DLS 2018。美国纽约州纽约：计算机协会，24，2018年，第37–49页。ISBN：978-1-4503- 6030-2。DOI：10.1145/3276945.3276951。[19] Fred Chow、Sun Chan、Robert Kennedy、Shin-Ming Liu、Raymond Lo 和 Peng Tu。"一种基于 SSA 形式的部分冗余消除新算法"。载于：*SIGPLAN Not.* （1，1997年），第273–286页。ISSN：0362-1340。DOI：10.1145/258916.258940。[20] Cliff Click 和 Michael Paleczny。"一种简单的基于图的中间表示"。载于：*SIGPLANNot.* （1，1995年），第35–49页。ISSN：0362-1340。DOI：10.1145/202530.202534。[21] E. F. Codd、S. B. Codd 和 C. T. Salley。"为用户分析师提供 OLAP：一项信息技术任务"。载于：E.F. Codd Associates，1993年。网址：https://www.semanticscholar.org / paper / Providing - OLAP - to - User - Analysts %3A - An - IT - Mandate - Salley - Codd/a0bd1491a54a4de428c5eef9b836ef6ee2915fe7#related-papers （访问于2022年12月19日）。[22] Keith Cooper、Timothy Harvey 和 Ken Kennedy。"一种简单快速的支配算法"。载于：莱斯大学计算机科学技术报告 *06-33870* （1，2006年）。网址：https://www.cs.rice.edu/~keith/EMBED/dom.pdf （访问于2022年8月30日）。[23] 事务处理性能委员会。*TPC-H* 基准测试 - 标准规范。美国加利福尼亚州旧金山：事务处理性能委员会（TPC），2021年，第138页。网址：https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.0.pdf （访问于2022年4月19日）。[24] Andrew Craik。*GraalVM* 中的增强型自动向量化。graalvm。16，2021年。网址：
h
t
t
p
s://medium.com/graalvm/enhanced-automated-vectorization-in-graalvm-76cd99925b6d （访问于2022年8月30日）。[25] Ron Cytron、Jeanne Ferrante、Barry K. Rosen、Mark N. Wegman 和 F. Kenneth Zadeck。"高效计算静态单赋值形式与控制依赖图"。载于：*ACM Trans.Program.Lang. Syst.* （1，1991年），第451–490页。ISSN：0164-0925。DOI：10.1145/115372.115320。[26] John-David Dalton。*Lodash*。29，2022年。网址：https://github.com/lodash/lodash（访问于2022年6月29日）。[27] Jack W. Davidson 和 Sanjay Jinturkar。"可重定向优化编译器中的激进循环展开"。载于：编译器构造。Tibor Gyimóthy 编。《计算机科学讲义》。柏林、海德堡：施普林格，1996年，第59–73页。ISBN：978-3-540-49939-8。DOI：10.1007/3-540-61053-7_53。

[28] Mattias De Wael、Stefan Marr、Joeri De Koster、Jennifer B. Sartor 和 Wolf- gang De Meuter。《即时数据结构》。载于：*2015 ACM* 国际动态语言研讨会：编程与软件的新思想、新范式和新思考（*Onward!*）Onward! 2015。美国纽约州纽约：ACM，21，2015年，第61–75页。isbn：978-1-4503-3688-8。doi：10.1145/2814228.2814231。[29] Gilles Duboscq、Lukas Stadler、Thomas Wuerthinger、Doug Simon、Christian Wimmer 和 Hanspeter Mössenböck。《Graal IR：一种可扩展的声明式中间表示》。载于：亚太编程语言与编译器研讨会论文集。第二届亚太编程语言与编译器研讨会。中国深圳，23，2013年，第9页。网址：https://ssw.jku.at/General/Staff/GD/APPLC-2013-paper_12.pdf （访问于2022-12-28）。[30] Gilles Duboscq、Thomas Würthinger 和 Hanspeter Mössenböck。《无悔推测：减少 Graal 编译器中的去优化元数据》。载于：*2014* 年 *Java* 平台编程原则与实践国际会议：虚拟机、编程语言与工具论文集。PPPJ '14。波兰克拉科夫：ACM，23，2014年，第187–193页。isbn：978-1-4503-2926-2。doi：10.1145/2647508.2647521。[31] Gilles Marie Duboscq。《将推测优化与副作用的灵活调度相结合》。林茨，2016年4月，2016年。网址：https://resolver.obvsg.at/urn:nbn:at:at-ubl:1-9708 （访问于2022-09-29）。[32] Amit Dwivedi、C. Lamba 和 Shweta Shukla。《列式数据库与行式数据库的性能分析》。载于：《国际计算机应用杂志》（1，2012年），第31–34页。doi：10.5120/7841-1050。[33] ECMA International。*ECMA-262，第12版，2021年6月*。美国加利福尼亚州旧金山：ECMA（欧洲通信系统标准化协会），2020年，第879页。网址：https://262.ecma-international.org/13.0/（访问于2022-12-29）。[34] ECMA International。*Tc39/Test262*。Ecma TC39，26，2022年。网址：https://github.com/tc39/test262 （访问于2022-04-27）。[35] Kareem El Gebaly 和 Jimmy Lin。*Afterburner*：浏览器内分析的理由。12，2016年。doi：10.48550/arXiv.1605.04035。arXiv：1605.04035 [cs].[36] Filip Pizlo。*JetStream2.1*。2014年。网址：https://browserbench.org/JetStream/（访问于2022-12-14）。[37] Loukas Georgiadis、Renato F. Werneck、Robert E. Tarjan、Spyridon Triantafyllis 和 David I. August。《实践中的支配节点查找》。载于：算法——*ESA 2004*。Susanne Albers 和 Tomasz Radzik 编。《计算机科学讲义》。柏林、海德堡：施普林格，2004年，第677–688页。isbn：978-3-540- 30140-0。doi：10.1007/978-3-540-30140-0_60.[38] Google。*Lovefield*。16，2021年。网址：https://github.com/google/lovefield （访问于2022-09-29）。

[39] Matthias Grimmer、Chris Seaton、Roland Schatz、Thomas Würthinger 和 Hanspeter Mössenböck。《多语言运行时中的高性能跨语言互操作性》。载于：第*11*届动态语言研讨会论文集。DLS 2015。美国纽约州纽约：计算机协会，21，2015年，第78–90页。isbn：978-1-4503-3690-1。doi：10.1145/2816707.2816714。[40] Philipp Marian Grulich、Steffen Zeuch 和 Volker Markl。《Babelfish：多语言查询的高效执行》。载于：*Proc. VLDB Endow.*（2021年），第196–210页。issn：2150-8097。doi：10.14778/3489496.3489501。[41] Urs Hölzle、Craig Chambers 和 David Ungar。《使用动态去优化调试优化代码》。载于：*ACM SIGPLAN 1992* 编程语言设计与实现会议论文集。PLDI '92。美国纽约州纽约：计算机协会，1，1992年，第32– 43页。isbn：978-0-89791-475-8。doi：10.1145/143095.143114。[42] Christian Humer、Christian Wimmer、Christian Wirth、Andreas Wöß 和 Tho- mas Würthinger。《用于构建自优化 AST 解释器的领域特定语言》。载于：*2014* 年生成式编程：概念与经验国际会议论文集。GPCE 2014。美国纽约州纽约：计算机协会，15，2014年，第123–132页。isbn：978-1-4503-3161-6。doi：10.1145/2658761.2658776。[43] 英特尔。使用英特尔进行向量化指南® *C++* 编译器。2010年。网址：
h
t
t
p
s://
s
o
f
t
w
a
re.intel.co
m/content/dam/develop/external/us/en/documents/31848-compilerautovectorizationguide.pdf （访问于2022-09-29）。[44] 英特尔。英特尔® *64* 和 *IA-32* 架构软件开发人员手册，第*1*卷：基本架构，第*5*章——指令集摘要。手册。2022年，第111– 148页。网址：https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-1-manual.html（访问于2022-12-30）。[45] JuliaArrays 协作者。*StructArrays*。11，2022年。网址：https://github.com/JuliaArrays/StructArrays.jl（访问于2022-12-13）。[46] JuliaData 协作者。*DataFrames.Jl*。13，2022年。网址：https://github.com/JuliaData/DataFrames.jl（访问于2022-12-13）。[47] Michael S. Kester、Manos Athanassoulis 和 Stratos Idreos。《主存优化数据系统中的访问路径选择：应该扫描还是探测？》。载于：*2017 ACM* 数据管理国际会议论文集。SIGMOD '17。美国纽约州纽约：计算机协会，9，2017年，第715–730页。isbn：978-1-4503-4197-4。doi：10.1145/3035918.3064049。[48] Tirthankar Lahiri、Shasank Chavan、Maria Colgan、Dinesh Das、Amit Ganesh、Mike Gleeson、Sanket Hase、Allison Holloway、Jesse Kamp、Teck-Hua Lee、Juan Loaiza、Neil Macnaughton、Vineet Marwah、Niloy Mukherjee、Atrayee Mullick、Sujatha Muthulingam、Vivekanandhan Raja、Marty Roth、Ekrem Soylemez 和 Mohamed Zait。《Oracle 内存数据库：双格式内存数据库》。载于：*2015 IEEE* 第*31*届国际数据工程会议。2015

IEEE 第31届国际数据工程会议。2015年，第1253–1258页。doi:10.1109/ICDE.2015.7113373.[49] Harald Lang、Tobias Mühlbauer、Florian Funke、Peter A. Boncz、Thomas Neu- mann 和 Alfons Kemper。“数据块：在压缩存储上结合向量化与编译，实现混合联机事务处理和联机分析处理”。载于：*2016*年国际数据管理会议论文集。SIGMOD '16。美国纽约州纽约：计算机协会，2016年14日，第311–326页。isbn：978-1-4503-3531-7。doi:10.1145/2882903.2882925.[50] Florian Latifi、David Leopoldseder、Christian Wimmer 和 Hanspeter Mössen- böck。“CompGen：在多语言虚拟机中生成快速即时编译器”。载于：第*17*届*ACM SIGPLAN*动态语言国际研讨会论文集。DLS 2021。美国纽约州纽约：计算机协会，2021年19日，第35–47页。isbn：978-1-4503-9105-4。doi：10.1145/3486602.3486930.[51] Thomas Lengauer 和 Robert Endre Tarjan。“一种在流图中查找支配节点的快速算法”。载于：*ACM*程序设计语言与系统汇刊(1，1979年)，第121–141页。issn：0164-0925。doi：10.1145/357062.357071.[52] David Leopoldseder、Roland Schatz、Lukas Stadler、Manuel Rigger、Thomas Würthinger 和 Hanspeter Mössenböck。“对非计数循环进行快速路径循环展开，以支持后续编译器优化”。载于：第*15*届托管语言与运行时国际会议论文集。Man- Lang '18。美国纽约州纽约：计算机协会，2018年12日，第1–13页。isbn：978-1-4503-6424-9。doi：10.1145/3237009.3237013.[53] David Leopoldseder、Lukas Stadler、Thomas Würthinger、Josef Eisl、Doug Si- mon 和 Hanspeter Mössenböck。“基于支配关系的复制模拟（DBDS）：通过代码复制支持编译器优化”。载于：*2018*年代码生成与优化国际研讨会论文集 - *CGO 2018*。2018年国际研讨会。奥地利维也纳：ACM出版社，2018年，第126–137页。isbn：978-1-4503-5617-6。doi：10.1145/3168811.[54] Jimmy Lin 和 Kareem El Gebaly。“大数据的未来是……JavaScript？”载于：*IEEE*《互联网计算》(2016年)，第82–88页。issn：1941-0131。doi：10.1109/MIC.2016.109.[55] Lukas Makor、Sebastian Kloibhofer、David Leopoldseder、Daniele Bonetta、Lukas Stadler 和 Hanspeter Mössenböck。“运行时将数组自动转换为列式存储”。载于：第*19*届托管编程语言与运行时国际会议论文集。MPLR 2022。比利时布鲁塞尔：计算机协会，2022年14日。isbn：978-1-4503-9696-7。doi:10.1145/3546918.3546919.[56] Toni Mattis、Johannes Henning、Patrick Rein、Robert Hirschfeld 和 Malte Appeltauer。“列式对象：提升分析型应用的性能”。载于：*2015*年*ACM*编程与软件新思想、新范式和反思国际研讨会（*Onward!*）SPLASH '15：系统、编程、编程语言与应用：软件

Humanity。美国宾夕法尼亚州匹兹堡：ACM，2015年21日，第197–210页。isbn：978-1- 4503-3688-8。doi:10.1145/2814228.2814230.[57] Joe Minichino。*LokiJS*。2022年28日。网址：https://github.com/techfort/LokiJS（访问于2022年6月29日）。[58] Raphael Mosaner、David Leopoldseder、Lukas Stadler 和 Hanspeter Mössen- böck。“利用机器学习预测动态编译器中重复启发式方法对代码大小的影响”。载于：第*18*届*ACM SIGPLAN*托管编程语言与运行时国际会议论文集。MPLR 2021。美国纽约州纽约：计算机协会，2021年29日，第127–135页。isbn：978-1-4503-8675-3。doi：10.1145/3475738.3480943.[59] Hanspeter Mössenböck。为*Java Hotspot*添加静态单赋值形式和图着色寄存器分配器*TM* 客户端编译器。奥地利林茨：约翰内斯·开普勒大学实用计算机科学研究所，2000年，第34页。网址：https://ssw.jku.at/Research/Reports/Report15.PDF（访问于2022-12-30）。[60] Hanspeter Mössenböck 和 Michael Pfeiffer。“SSA形式和寄存器约束背景下的线性扫描寄存器分配”。载于：编译器构造。R. Nigel Horspool 编。《计算机科学讲义》。柏林，海德堡：施普林格，2002年，第229–246页。isbn：978-3-540-45937-8。doi:10.1007/3-540-45937-5_17.[61] Frank Mueller 和 David B. Whalley。“通过代码复制避免条件分支”。载于：*ACM SIGPLAN 1995*年编程语言设计与实现会议论文集。PLDI ’95。美国纽约州纽约：计算机协会，1995年1日，第56–66页。isbn：978-0-89791-697-4。doi：10.1145/207110.207116.[62] Frank Mueller 和 David B. Whalley。“通过代码复制避免无条件跳转”。载于：*ACM SIGPLAN 1992*年编程语言设计与实现会议论文集。PLDI ’92。美国纽约州纽约：计算机协会，1992年1日，第322–330页。isbn：978-0- 89791-475-8。doi:10.1145/143095.143144.[63] Rostislav Netek、Jan Brus 和 Ondrej Tomecka。“标记聚类和热力图可视化技术的性能测试：JavaScript制图库的比较研究”。载于：*ISPRS*国际地理信息期刊(2019年8期)，第348页。issn：2220-9964。doi：10.3390/ijgi8080348.[64] Michael E. Noth。“面向性能的Java对象展开”。博士论文。美国：华盛顿大学，2003年。网址：https:// digital. lib.washington.edu /researchworks/handle/1773/6889 （访问于2022-12-09）。[65] 甲骨文。*GraalVM*。2021年9日。网址：https://www.graalvm.org/ （访问于2022年9月29日）。[66] 甲骨文。*GraalVMJavaScript*。2022年5日。网址：https://github.com/oracle/graaljs（访问于2022-12-07）。

[67] 甲骨文。*GraalVM* 循环复制。甲骨文，29，2022年。网址：https://github.com/oracle/graal/blob/2849fa6c1c087153479b5bb306841073de498614/compiler/src/org.graalvm.compiler.nodes/src/org/graalvm/compiler/nodes/loop/LoopFragment.java （访问于2022年8月30日）。[68] 甲骨文。*HotSpot Server VM* 编译器（*C2*）中的循环优化 - *HotSpot Server VM* 编译器（*C2*）中的循环优化 - *OpenJDK Wiki*。OpenJDK Wiki。2014 年。网址：https://wiki.openjdk.org/pages/viewpage.action?pageId=20415918（访问于2022年12月13日）。[69] 甲骨文。*Node.js* 运行时。25，2022年。网址：https://www.graalvm.org/22.0/reference-manual/js/NodeJS/（访问于2022年4月25日）。[70] Michael Paleczny、Christopher Vick 和 Cliff Click。“Java HotSpotTM 服务器编译器”。载于：第1届 Java 虚拟机研究与技术研讨会论文集，2001年4月23日至24日，美国加利福尼亚州蒙特雷。美国加利福尼亚州蒙特雷：USENIX，1，2001年，第13页。网址：https://www.usenix.org/conference/jvm-01/java-hotspot%E2%84%A2-server-compiler（访问于2022年9月7日）。[71] Jim Pivarski、Peter Elmer、Brian Bockelman 和 Zhe Zhang。“通过代码转换快速访问列式层次嵌套数据”。载于：*2017 IEEE* 国际大数据会议（大数据）。2017 IEEE 国际大数据会议（大数据）。美国马萨诸塞州波士顿：IEEE，2017年，第253–262页。doi：10.1109/BigData.2017.8257933。[72] Richard Plangger 和 Andreas Krall。“PyPy 追踪即时编译器中的向量化”。载于：第*19*届嵌入式系统软件与编译器国际研讨会论文集。SCOPES ’16：第19届嵌入式系统软件与编译器国际研讨会。德国圣戈阿尔：ACM，23，2016年，第67–76页。isbn：978-1-4503-4320-6。doi：10.1145/2906363.2906384。[73] Aleksandar Prokopec、Gilles Duboscq、David Leopoldseder 和 Thomas Würthinger。“面向优化的即时编译器增量内联替换算法”。载于：*2019 IEEE/ACM*代码生成与优化国际研讨会论文集。2019 IEEE/ACM代码生成与优化国际研讨会（CGO）。CGO 2019。美国华盛顿哥伦比亚特区：IEEE Press，16，2019年，第164–179页。isbn：978-1-72811-436-1。doi：10.1109/CGO.2019.8661171。[74] Ganesan Ramalingam。“论循环、支配节点和支配边界”。载于：*ACM* 程序设计语言与系统汇刊（1，2002年），第455–490页。issn：0164-0925。doi：10.1145/570886.570887。[75] RAPIDS 开发团队。*RAPIDS*：端到端 *GPU* 数据科学库集合。手册。2018年。网址：https://rapids.ai（访问于2022年12月28日）。[76] Amir Shaikhha、Yannis Klonatos 和 Christoph Koch。“使用高级语言构建高效查询引擎”。16，2016年。arXiv：1612.05566 [cs]。网址：http://arxiv.org/abs/1612.05566 （访问于2022年9月29日）。

[77] Lukas Stadler、Thomas Würthinger 和 Hanspeter Mössenböck。“Java 的部分逃逸分析与标量替换”。载于：年度 *IEEE/ACM*代码生成与优化国际研讨会论文集。CGO ’14。美国佛罗里达州奥兰多：ACM，15，2014年，第165–174页。isbn：978-1-4503-2670-4。doi：10.1145/2581122.2544157。[78] Mike Stonebraker、Daniel Abadi、Adam Batkin、Xuedong Chen、Mitch Cherniack、Miguel Ferreira、Edmond Lau、Amerson Lin、Sam Madden、Elizabeth O’Neil、Pat O’Neil、Alex Rasin、Nga Tran 和 Stan Zdonik。“C-Store：列式数据库管理系统”。载于：第*31*届超大型数据库国际会议论文集。VLDB ’05。挪威特隆赫姆：VLDB基金会，30，2005年，第553–564页。isbn：978-1-59593-154-2。doi：10.1145/3226595.3226638。[79] Sivaprasad Sudhir、Michael Cafarella 和 Samuel Madden。“内存数据库系统的复制布局”。载于：*VLDB*基金会论文集（1，2021年），第984–997页。issn：2150-8097。doi：10.14778/3503585.3503606。[80] pandas开发团队。*Pandas-Dev/Pandas*：*Pandas*。Zenodo，22，2022年。doi：10.5281/zenodo.7344967。[81] PyPy团队。*PyPy*。PyPy。28，2019年。网址：https://www.pypy.org/（访问于2022年12月20日）。[82] V8编译器团队。“TurboFan编译器概览”。网址：https://docs.google.com/presentation/d/1H1lLsbclvzyOF3IUR05ZUaZcqDx07_-8f4yJoxdMooU（访问于2022年12月13日）。[83] Hannes Voigt、Alfred Hanisch 和 Wolfgang Lehner。“Flexs——物理数据布局的逻辑模型”。载于：数据库与信息系统新趋势*II*。Nick Bassiliades、Mirjana Ivanovic、Margita Kon-Popovska、Yannis Manolopoulos、Themis Palpanas、Goce Trajcevski 和 Athena Vakali 编。智能系统与计算进展。查姆：Springer国际出版公司，2015年，第85–95页。isbn：978-3-319-10518-5。doi：10.1007/978-3-319-10518-5_7.[84] Christian Wimmer 和 Michael Franz。“SSA形式上的线性扫描寄存器分配”。载于：第*8*届*IEEE/ACM*年度代码生成与优化国际研讨会论文集。CGO ’10。美国纽约州纽约：计算机协会，24，2010年，第170–179页。isbn：978-1-60558-635-9。doi：10.1145/1772954.1772979。[85] Christian Wimmer 和 Thomas Würthinger。“Truffle：自优化运行时系统”。载于：第*3*届年度系统、编程与应用大会：面向 *Humanity* 的软件论文集。SPLASH ’12。美国亚利桑那州图森：ACM，19，2012年，第13–14页。isbn：978-1-4503-1563-0。doi：10.1145/2384716.2384723。[86] Andreas Wöß、Christian Wirth、Daniele Bonetta、Chris Seaton、Christian Humer 和 Hanspeter Mössenböck。“面向 Truffle 语言实现框架的对象存储模型”。载于：*2014*年 *Java* 平台虚拟机、编程语言与工具的编程原理与实践国际会议论文集 - *PPPJ ’14*。2014年国际会议。

波兰克拉科夫：ACM出版社，2014年，第133–144页。ISBN：978-1-4503-2926-2。DOI：10.1145/2647508.2647517.[87] Thomas Würthinger、Christian Wimmer、Christian Humer、Andreas Wöß、Lukas Stadler、Chris Seaton、Gilles Duboscq、Doug Simon 和 Matthias Grimmer。《面向高性能动态语言运行时的实用部分求值》。载于：第*38*届*ACM SIGPLAN*编程语言设计与实现会议论文集。PLDI 2017。美国纽约州纽约：计算机协会，2017年14日，第662–676页。ISBN：978-1-4503-4988-8。DOI：10.1145/3062341.3062381.[88] Thomas Würthinger、Christian Wimmer、Andreas Wöß、Lukas Stadler、Gilles Duboscq、Christian Humer、Gregor Richards、Doug Simon 和 Mario Wolczko。《一台虚拟机统御一切》。载于：*2013*年*ACM*新思想、新范式与编程及软件反思国际研讨会论文集。Onward! 2013。美国印第安纳州印第安纳波利斯：ACM，2013年29日，第187–204页。ISBN：978-1-4503-2472-4。DOI：10.1145/2509578.2509581.[89] Thomas Würthinger、Andreas Wöß、Lukas Stadler、Gilles Duboscq、Doug Simon 和 Christian Wimmer。《自优化 AST 解释器》。载于：第*8*届动态语言研讨会论文集。DLS '12。美国纽约州纽约：计算机协会，2012年22日，第73–82页。ISBN：978-1-4503-1564-7。DOI：10.1145/2384577.2384587.[90] Wangda Zhang、Junyoung Kim、Kenneth A. Ross、Eric Sedlar 和 Lukas Stadler。《面向数据密集型分析的自适应代码生成》。载于：*Proc. VLDBEndow.*（2021年），第929–942页。ISSN：2150-8097。DOI：10.14778/3447689.3447697.[91] Jisheng Zhao、I. Rogers、C. Kirkham 和 I. Watson。《面向 Jikes RVM 的循环并行化》。载于：第六届并行与分布式计算应用及技术国际会议（*PDCAT'05*）。第六届并行与分布式计算应用及技术国际会议（PDCAT'05）。2005年，第35–39页。DOI：10.1109/PDCAT.2005.164。

关于作者

Sebastian Kloibhofer 是奥地利林茨约翰内斯·开普勒大学的一名博士生。请通过sebastian.kloibhofer@jku.at联系他。

Lukas Makor是奥地利林茨约翰内斯·开普勒大学的博士生。联系邮箱：lukas.makor@jku.at。

David Leopoldseder 是 Oracle Labs 的研究员。联系邮箱：david.leopoldseder@oracle.com

Daniele Bonetta 是 Oracle Labs 的研究员。联系邮箱：daniele.bonetta@oracle.com

Lukas Stadler是 Oracle Labs 的研究员。联系邮箱：lukas.stadler@oracle.com

Hanspeter Mössenböck 是奥地利林茨约翰内斯·开普勒大学的教授。请通过以下邮箱联系他：hanspeter.moessenboeck@ jku.at