一刀切并不适用于所有情况：物理表示对 Datalog 求值的重要性

Nick Rassau约翰内斯·古腾堡大学德国美因茨
rassau@uni-mainz.de

Felix Schuhknecht约翰内斯·古腾堡大学德国美因茨
schuhknecht@uni-mainz.de

arXiv:2602.05651v1 [cs.DB] 5 Feb 2026

摘要—**Datalog 是一种日益流行的递归查询语言，其设计本质上具有声明式特征，这意味着程序必须由引擎转换为实际的物理执行计划。在生成该计划时，一个核心决策是如何对其中涉及的所有关系进行物理表示；在这方面，现有的 Datalog 引擎出人意料地受到限制，往往采用一刀切解决方案。其原因在于，Datalog 程序的典型执行计划不仅会针对物理表示执行单一类型的操作，而是会混合执行多种操作，例如插入操作、查找操作和包含性检查。此外，每种操作类型的重要性在很大程度上取决于工作负载特征；这些特征既包括熟悉的属性，例如各个关系的大小、重数和元数，也包括非常具体的 Datalog 属性，例如关系多次出现时规则的“交织”，尤其是查询的递归性——它可能在求值过程中动态生成新元组。这表明，需要采用多种物理表示，各自发挥所长、弥补所短，以满足不同工作负载场景的特定需求。为对此进行评估，我们深入开展了一项实验研究，考察了可能适用的物理表示与工作负载特征的七个维度之间的相互作用；这些特征维度在实际 Datalog 程序中各不相同。研究揭示了哪些属性确实重要。基于这些发现，我们设计了一种自动选择机制，利用一组决策树为给定工作负载确定合适的物理表示。**

一、引言

如今，Datalog 已应用于广泛的领域，从程序分析 [1], [2] 到网络监控 [3]、分布式计算 [4], [5],和分布式存储 [6]。Datalog 如此受欢迎的原因在于它具有声明式特性：查询只描述要计算什么，而不描述如何计算。基于此，Datalog 引擎会生成一个物理计划，并在数据集上执行该计划。在制定计划时，一个核心决策是如何以物理方式表示所有相关关系。这种物理表示包含多个方面。首先是访问类型，它决定元组如何存储和定位。例如，可以将所有元组存储在扁平元组存储中，再通过扫描这些存储来检索符合条件的条目。也可以结合使用元组存储和非聚簇索引，以便在需要时定位并从存储中检索完整元组。另一种方案是使用覆盖索引，在索引内部直接物化所有元组，从而完全不需要单独的元组存储。此外，还需要选择数据结构，这对于索引尤为重要。此处可以采用各种有序和无序索引结构，例如 B 树或哈希表，它们在性能和空间特性方面可能存在显著差异。

尽管有如此多的选项，Datalog 引擎通常仍采用相当静态的 [7] 或相当有限的选择 [8]–[11]。例如，最先进的 Datalog 引擎 Souffl´e [1], [12], [13] 将每个关系表示为覆盖 B 树。虽然这降低了实现工作量并简化了代码生成，但（a）尚不清楚这一决策在所有情况下是否都是最佳选择（或至少是良好选择）。此外，（b）这一设计决策也带来了副作用：如果某个关系参与了多个使用不同连接键的连接，就必须创建多个 B 树，其中冗余地存储着元组，并采用不同的排序顺序，从而大幅增加内存占用。为解决这一问题，Souffl´e 对宽关系改用非聚簇 B 树，并将元组单独存储一次。尽管这一启发式方法能减少宽关系的内存占用，但它当然会造成大量随机访问；如果连接选择率较低，查询性能可能会因此大幅下降。

A. 一刀切还是量身定制？

尽管如此，Datalog 引擎在这方面如此简单，也有其合理之处。首先，Datalog 程序的典型执行计划不仅会对物理表示执行单一类型的操作，而是会混合执行多种操作，例如发现新元组时进行插入操作、连接处理时进行查找操作，以及为确保集合语义而进行包含性检查。因此，不能只选择在某一特定操作上表现良好的物理表示，而必须考量其在一整组操作中的性能。其次，各类操作的重要性在很大程度上取决于工作负载的特征。这些特征既包括大小、重数和各个关系的元数等常见属性，也包括非常具体的 Datalog 属性，例如关系出现在多条规则中时规则之间的“交织”，尤其是查询的递归性；递归性可能会在求值过程中生成新元组。例如，对于在求值过程中会识别出大量新元组的工作负载，递归关系需要采用能够高效处理单个追加操作的物理表示。

这些观察表明，一刀切解决方案可能无法满足 Datalog 求值的需求。相反，可能需要从物理表示目录中进行选择，目录中的物理表示各有优缺点，具体取决于所面对的工作负载特征。

为评估这一点，我们深入研究了不同的潜在适用物理表示之间的相互作用，以及实际 Datalog 程序中各异的工作负载特征。这揭示了物理表示和工作负载的哪些属性真正重要。基于这些见解，我们构建了一组决策树，可为未来集成物理表示目录的系统中的物理查询规划器奠定基础。我们的贡献如下：

(1)为了研究不同物理表示与工作负载特征之间的关系，我们开发了一款高度灵活的 Datalog 引擎，名为 PlayLog（https://gitlab.rlp.net/rassau/playlog）。它采用受 Souffl´e 启发的执行模型，同时从设计之初就支持多种物理表示，因此具有很高的灵活性。这种灵活性不会带来任何性能损失，因为 PlayLog 会执行查询编译，生成的结果始终是一个专用的 C++ 程序，其外观和运行方式都与手写执行计划相同。

(2) 在 PlayLog 中，我们集成了以下物理表示，并将其视为由两个部分组合而成：（i）关系元组的访问类型。我们支持创建独立的覆盖索引、基于元组存储的非聚簇键索引、基于元组存储的非聚簇指针索引，以及通过全扫描进行搜索的独立元组存储。（ii）用于表示索引的数据结构。我们支持朴素的有序数组、B+-树、基数树和哈希表。元组存储采用扁平行存储表示。这些结构此前已用于 Datalog 场景，例如 Souffl´e、RecStep [8], BigDatalog [14] 和 Myria [15]。

(3) 我们深入研究了物理表示与工作负载特征之间的相互作用。对于工作负载特征，我们考察了 Datalog 程序中经常变化的七个维度：**(i)** 关系大小，**(ii)** 关系重数，**(iii)** 关系宽度，**(iv)** 键宽度，**(v)** 初始化与查询开销的比例，**(vi)** 规则之间及规则内部按重复出现的关系衡量的交织程度，以及 **(vii)** 按生成的事实数量、唯一事实数量和新事实数量衡量的规则递归性。我们在可控的合成工作负载和四种真实世界工作负载上研究了这些方面。

(4) 我们提出了一种选择机制，它依据能够捕捉工作负载相关属性的签名，使用一组决策树为给定的 Datalog 程序实例化物理表示。为了评估该选择机制的有效性，我们将其应用于四种真实世界工作负载，并证明它能够识别出近乎理想的配置，其表现优于一组基线系统。

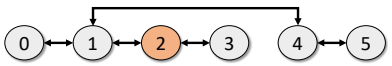
请注意，鉴于该主题的复杂性很高，本研究聚焦于单节点主存评估。分析和优化分布式 Datalog 引擎超出了本文的范围，但这是值得探讨的课题，我们计划在未来的工作中加以研究。

II. 背景与动机

我们首先讨论 Datalog 程序的求值方式。在此基础上，我们将探讨为何物理表示的选择对 Datalog 求值而言既具挑战性又十分重要。

A. Datalog 简述

为了了解 Datalog，我们计算双向图中的可达性。起始节点和边构成所谓的输入事实，并被物化为相应关系中的元组，例如 边 ={(0,1),(1,2),(1,4),(2,3),(4,5)}和 可达 ={(2)}，类似于下图：



为了计算从节点 (2) 出发的可达性，我们编写一个由两条规则 组成的 Datalog 查询，这些规则在逻辑上逐条求值：

可达(y) :- 可达(x), 边(x,y)
可达(y) :- 可达(x), 边(y,x)

让我们看看第一条规则的求值：规则的右侧（即其规则体）通过引入共享变量 x，在第一个属性上将关系 可达 与关系 边 进行连接。通过引入第二个变量 y，我们捕获与来自可达的元组完成连接的每个元组中 边 的第二个属性的值。由于这正是我们在可达性计算中所关注的内容，我们将所得的值 y 作为可能的新元组插入规则左侧（即其 规则头）的可达 关系中。由于我们的规则头关系也是规则体的一部分，这会产生递归求值，其中该规则被反复求值，直到不再生成新的事实。请注意，边 也被称为外延数据库（EDB），因为它包含初始事实并保持静态，而 可达 被称为内涵数据库（IDB），因为它可能会接收新的事实。

B. 程序求值

求值 Datalog 程序时，我们区分非递归规则和递归规则。求值形式为 D(i,l) :- A(i,j), B(j,k), C(k,l) 的非递归规则，需执行以下步骤：首先，初始化所有关系（创建物理表示，并视情况填入数据）。其次，根据连接键，将规则体中的关系按顺序进行连接，从而可能产生事实；第三，如果这些事实是新的，就将它们插入规则头关系。求值递归规则（例如 A(i,l):- A(i,j), B(j,k), C(k,l) ）则涉及更多步骤。其中，事实上的标准技术 [16] 是半朴素自底向上求值，例如 Souffl´e、BigDatalog、SociaLite [17], 或 RecStep [8]均采用了这一技术。其核心思想是将递归规则转换为一系列非递归求值，直到不再产生新事实为止。为降低开销，序列中的每次迭代都只基于上一次迭代发现的事实进行求值。算法 1 以伪代码展示了程序的求值过程，其中连接按一系列（索引）嵌套循环连接执行，正如 Souffl´e 所采用的方式：对于 IDB A，还会额外创建一个名为 Adelta 的关系。在任何时候，

该关系都只包含上一次迭代中新生成的事实，不包括A中已存在的潜在重复项。为启动求值过程，Adelta 中会填入A的所有初始元组。随后，主循环开始第一次迭代，并对规则体进行求值。然而，找到的每个事实都不会直接插入A，而是插入第三个关系Anew中，该关系会在每次迭代中重新创建。生成本次迭代的所有事实后，必须构建下一次迭代所用的Adelta 关系，并将Anew中的所有元组放入其中。进入下一次迭代前，将Adelta 合并到A中。如果 Adelta 仍为空，则返回A作为最终结果。

算法 1： 半朴素自底向上求值。颜色标示不同的性能关键操作。

```
   // A(i,l) :- A(i,j),          B(j,k),      C(k,l)
1  A.create(on 键 0 1),A.批量加载(输入A)
2  B.create(基于键 0)，B.批量加载(输入B)
3  C.create(基于键 0)，C.批量加载(输入C)
4  A_delta.create(基于键 0 1)，A_delta.批量加载(输入A)
5  当!A_delta.empty()时执行
6      A_new.create(基于键 0 1)
7      对于 A 条目 : A_delta 执行
8          B 条目 =B.探测((A 条目[1]))
9          对于 B 条目：B 条目集合 执行
10             C 条目 =C.探测((B 条目[1]))
11             对于 C 条目：C 条目集合 执行
12                 事实已找到 =(A 条目[0],C 条目[1])
13                 如果!A.contains(事实已找到)则
14                     如果！A_new.包含(找到的事实)那么
15                         A_new.append(找到的事实)
16     A_new.已完成 append()
17     对于 A_new 条目 : A_new 执行
18         A.append(A_new 条目)
19     A.已完成 append()
20     A_delta.从(A_new)移出
21 返回A
```

C. 性能关键操作无处不在！

如算法 1 所示，物理表示上的性能关键操作（我们用不同颜色突出显示了不同的操作）无处不在：

最初，必须从头创建A、Adelta、Anew、B和C，其中除Anew 之外的所有关系随后都将批量加载可能数量很大的输入元组。由于这些输入元组通常会在运行过程中动态到达 [1], [18], [19], 这就成为实际程序求值的一部分。因此，如果构建过程是程序求值的主要部分，那么在这方面非常高效的物理表示（例如无需构建辅助结构，或能在批量加载期间利用访问局部性）可能是更好的选择，即使它在后续使用中的性能可能不及竞争方案。在连接序列中，我们必须区分两部分：进入每个连接对右侧物理表示的探测部分，以及随后对探测结果进行的迭代；后者是继续连接序列所必需的。根据元组数量、匹配的连接伙伴数量以及每次匹配中的重复项数量，各部分的开销可能存在很大差异，因此应选择针对主导操作进行优化的物理表示。不同部分之间的开销可能存在很大差异，因此应选择针对主导操作进行优化的物理表示。识别出一个已找到的事实后，必须通过分别对A 和Anew执行包含性检查来完成去重，这要求物理表示能够高效地测试一个元组是否处于包含关系中。确认某个事实确实是新的之后，必须将其插入 Anew 关系。因此，如果找到大量新事实，支持快速单个追加操作的物理表示很可能比缺乏灵活性的表示更适合处理这种情况。最后，必须将Anew 的内容复制到基关系中，而当前的Anew 关系则成为下一个Adelta 关系。前者是批量加载的另一种形式，不过这次操作发生在一个已经填充数据的物理表示中，并且可能以分散的方式对其进行修改。如果Anew 的物理表示与Adelta的相匹配，后者就可以作为无需成本的“重新命名”来完成，以下内容中我们将作此假设。除了求值期间各项操作的平衡方式之外，模式宽度、键宽度以及共享物理表示的潜力等属性，也很可能会影响物理表示的选择。在第 IV 节中，我们将深入研究这些关系。

III. 物理表示

作为物理表示，我们考虑四种访问类型和五种数据结构，共计 13 种组合。

访问类型。访问类型决定元组的存储位置、索引方式（如果有）以及检索方式。第一种访问类型是覆盖索引（**CI**），它将所有元组直接物化在底层数据结构中，因此无需单独的 store 来存放这些元组。**CI** 的缺点在于，如果相应关系参与了多个连接，且这些连接使用不同的连接键，其中一个连接键不是另一个的前缀，就会产生问题。这会导致创建多个**CI**，从而冗余存储该关系的元组。为避免这种情况，对于具有许多不同连接键的宽关系，可以改用非聚集指针索引（**UPI**）。在这种索引中，所有元组都存储在单独的行存储中，底层索引结构仅引用这些元组。该索引结构不会显式物化键并建立映射，而只存储元组地址（因此得名指针型），并在探测时对其解引用以重建键。虽然这会导致昂贵的随机访问，但采用这种设计的原因是，宽复合键在 Datalog 程序中相当常见 [1], [20], [21]。此外，我们也支持非聚集键索引（**UKI**），它会显式物化键，从而避免在探测时随机访问元组行存储，对于较窄的键而言可能是不错的选择。第四种访问类型是全扫描（**FS**），它通过扫描并筛选符合条件的条目来响应探测。

数据结构。作为数据结构，我们支持以下选项：B+-树（**BP**），它将所有元组存储在叶层，并支持高效的范围扫描。我们采用 TLX 库 [22] 中广受欢迎的实现，并使用其默认配置。此外，在求值中，我们还支持以 seq 库 [23]中的 `seq::radix_tree`形式实现的基数树（**RX**）。

	CI	UKI	UPI	FS
SA(++)	T[]	<K,T*>[]	T*[]	-
BP	tlx::btree_多重集合<T>	tlx::btree_多重映射<K,T*>	tlx::btree_多重集合<T*>	-
HT	google 密集哈希映射<K,std::vector<T>>	google 密集哈希映射<K,std::vector<T*>>	google 密集哈希映射<T*,std::vector<T*>>	-
RX	seq::基数映射<T,std::vector<T>>	seq::基数映射<K,std::vector<T*>>	seq::基数映射<T*,std::vector<T*>>	-
RS	-	-	-	T[][]

表 I：所有支持的组合 访问类型和数据结构及其内部表示的选项。`T`是类型关系元组的.....，其中 其中`K` 是键的类型。对于所有**UKI** 和**UPI** 组合，`T*` 指向一个单独的**行存储** .

此变体实现了一种可变基数基数树（VART），它会根据数据集的稀疏程度自适应地采用不同的节点大小，与 ART [24] 或 JudyArray [25]类似。为了纳入一种内存占用极小的有序数据结构，我们考虑使用有序连续元组数组（**SA**）。为高效处理插入操作（这对于稠密数组而言颇具挑战），我们采用了两项优化。首先，我们使用系统调用`mremap()`，按 2MB内存块（即大页的大小）的粒度按需扩展数组。这避免了复制物理内存，并且仅在必要时才重新定位虚拟内存。其次，在执行`append()` 操作时，我们不会在每次插入元组后都进行排序。相反，我们先将元组存储在数组末尾的无序区域中，待`append()` 操作序列结束后，再对它们进行排序。请注意，如果`contains()`检查与`append()` 操作交错进行，这可能会降低检查速度，因为如果无法在有序区域中找到元组，就必须扫描无序区域才能响应请求。为解决这一问题，我们提供了另一种方案：在`append()` 操作期间关闭`contains()` 语义。因此，数据结构中可能会出现重复项；当求值表明追加序列已结束时，再将其移除。这样可以消除交错执行`contains()` 和`append()` 操作所带来的二次复杂度，但需要通过`std::unique`额外进行一次追加后遍历。下文将这两种变体分别记为**SA** （正常的`contains()`行为）和**SA ++** （在`append()` 序列期间停用`contains()` ），两者在探测操作下的行为相同。由于规则求值中的一项核心操作是执行等值连接，我们还将哈希表（**HT**）作为高效点查找的自然候选结构纳入考虑。我们使用 Google 的稀疏哈希库 [26]，中的`google::dense_hash_map()`，它在内部采用开放寻址和线性探测，并将默认最大负载因子设为 0.5。哈希函数采用 SMHasher套件 [27]中的 64 位 MurmurHash 变体。由于缺乏有序性，**HT**的`probe()` 操作不支持范围查找，因此单一物理表示无法用于多个以前缀连接键为条件的连接。由于**HT** 和**RX** 的底层结构都不支持重复键，我们使用元组向量作为值，以便仍能处理这种情况。此外，我们还纳入了一种扁平数据结构——行存储（**行存储**）。由于某些访问类型可能会引用此行存储中的元组，我们不会 像**SA**那样，将其组织为通过`mremap()`调整大小的稠密数组，因为这可能会移动虚拟内存区域并破坏引用。相反，我们将**RS** 组织为一个可扩展列表，由连续的内存块组成，每块大小为 2MB。

请注意，默认情况下，我们的所有数据结构都支持存储重复项，并依靠执行过程通过适当的 `contains()` 检查，在所有关系上强制实施集合语义。这使我们能够灵活地测试构建递归关系的不同策略，我们将在第 IV-F 节中讨论这一点。此外，为了精确测量**BP**、**RX**和**HT**的内存占用，我们为内部索引配备了一个可跟踪所有分配的自定义分配器。表 I 展示了访问类型和数据结构的全部 13 种有意义的组合。

IV. 物理表示与工作负载特征的相互作用

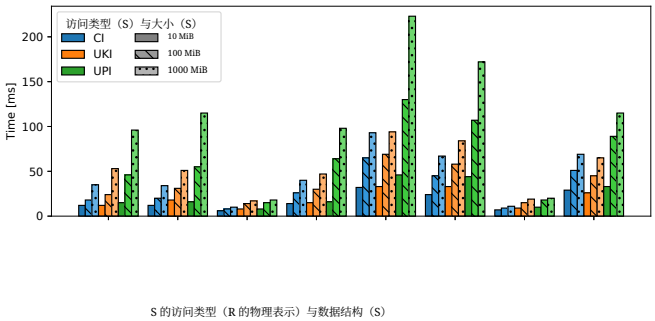
接下来，我们将分析物理表示与七种不同工作负载特征之间的相互作用。为应对 Datalog 求值的复杂性，我们将 Datalog 中最重要的方面拆解为独立实验。随后在第 V 节中，我们会将这些实验结果应用于一组决策树，选择机制将在这些决策树上运行。

我们在一台最高运行频率为 5.2GHz 的 Intel i9-12900K 上进行所有后续实验。该 CPU 配备 30MB 共享 L3 缓存，而单个性能核心可使用 1.25MB L2 缓存和 32KB L1 缓存。这台机器配备 128GB DDR4 主内存，运行 64 位 Arch Linux，内核版本为 6.13.4。透明大页已启用，可用于内存分配。我们使用版本 14.2.1 的 g++，并以 O3 优化级别编译 PlayLog。所有报告的运行时间均为三次运行的平均值。此外，以下求值中使用的所有关系，其所有属性均存储 8B 无符号整数。请注意，Soufflé 的核心引擎也仅支持定长数据类型；字符串等变长数据类型仍可通过单独的字典将值编码为整数或从整数解码来处理。

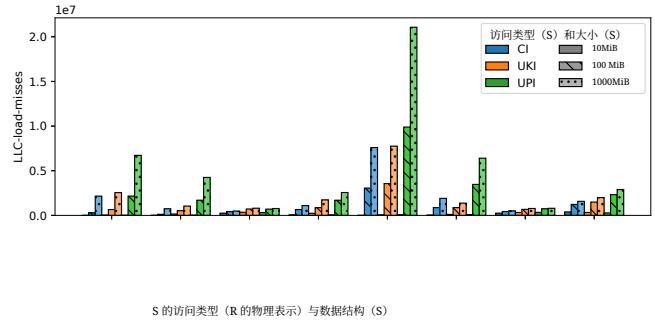
A. 规则体求值的扩展行为

为初步了解不同物理表示的特点，我们首先分析随着所含元组数量（也就是探测的元组数量）变化时，它们的扩展行为。暂且假设关系已经批量加载完毕，并且只关注规则体求值：我们对形式为 `R(x,y),S(x,z)` 的规则体进行求值，从连接结果的 `y` 和 `z` 属性计算校验和。我们将外关系 `R` 固定为 1.25MiB，并将被探测的内关系 $\overline{S}$ 从 10MiB 增至 100MiB，再增至 1,000MiB（原始元组的内存占用）；两个关系中填充的都是经过密集选择、但顺序打乱的元组。在改变 `S`的大小时，我们确保 `R` 中的每个元组在 `S` 中恰好找到两个连接伙伴，从而保持连接结果的数量不变，也就使实际连接开销保持不变

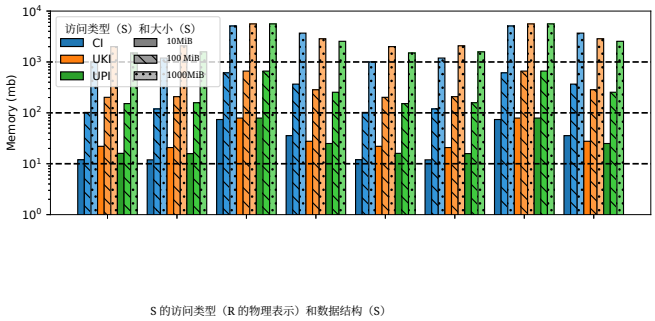
固定。这使我们能够评估不同物理表示如何处理探测端的扩展。



(a) 主体评估时间。



(b) 主体评估的 L3 缓存未命中数（LLC-load-misses 计数器）。



(c) 内存占用。

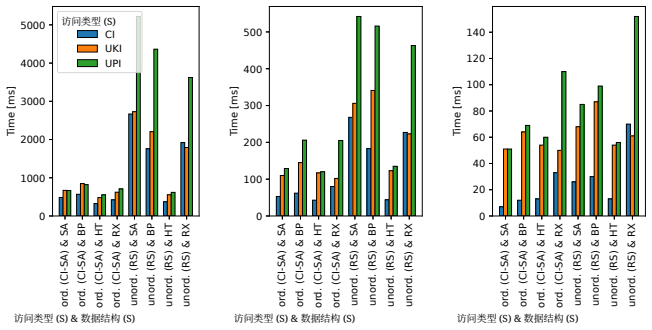
图 1：不同物理表示的扩展行为。

图 1a 展示了不同设置下规则体求值的总耗时。对于内关系 S，我们展示所有有意义的数据结构与访问类型组合；而对于外关系 R，为保证图表清晰易读，只展示两种代表性组合：**CI-SA**，它会对 S进行有序探测；以及 **RS**，可视为生成无序探测的对应方案。从结果中可以看出，所测试的物理表示在性能上存在显著差异。首先看访问类型，可以发现 **CI**通常优于 **UPI**，而 **UKI**则居于两者之间。这一排序合乎情理，因为 **CI**完全避免了对单独 store 的随机访问，而 **UKI**则会对底层 **RS**进行随机访问以获取元组。**UPI**在随机访问下的表现更差，因为它在索引遍历期间还必须访问底层的 **RS** ，也就是执行键比较。性能差异而且，S 越大，这种差异就越明显：当大小为 10MiB 时，**覆盖索引** 和 **UPI** 的性能差异不到 2×；而当大小为 1,000 MiB 时，差异最高可达 8×。这表明，在表示较大的关系时，应谨慎使用**UKI** ，尤其是**UPI** 。产生这种影响的原因是：对于S 而言，30MB 的 L3 缓存和用于大页的 32 个槽位的 L1 TLB（可覆盖 64MB）很快就会显得太小，无法避免随机访问期间代价高昂的未命中。图 1b 证实了这一点。该图绘制了使用perf对主体求值进行性能分析运行时得到的LLC-load-misses计数器，我们可以清楚地看到它与运行时间之间的相关性。dTLB-misses 计数器也有相同的情况，但受空间约束所限，这里不予展示。

将数据结构纳入讨论后，我们可以得出几个有趣的观察结果：首先，**BP**并非性能最佳的选项。对于有序探测，它的表现最差，甚至不如**SA**；后者利用了访问模式较高的时间局部性，同时结构开销更低，因此能更长时间地驻留于缓存。**RX** 的表现也略好，因为它无需比较，并且能非常有效地为稠密元组建立索引。不出所料，**HT** 的表现显著优于所有其他变体；与**UKI**和**UPI** 结合使用时，它也格外稳健，因为每次探测所需的内存访问更少，因此导致的缓存和 TLB 未命中也更少。这是个好消息，因为使用 **UKI** 或**UPI** 通常并非自由选择，而是受空间约束所迫。在这种情况下，令人欣慰的是，与**HT**结合使用仍能获得良好的性能。不利的一面是，正如展示内存占用的图 1c 所示，**HT** 在所有数据结构中占用空间最大，因此再次限制了它在此情境下的可用性（至少在像本文这样使用最大负载因子 0.5 时如此）。在无序探测下，所有基于比较的数据结构的表现都会变差，此时**BP**相较于**SA**成为表现第二差的选项。不出所料，**HT**基本不受访问模式影响，因此在这种情况下是最合理的选择。总体而言，这些初步发现表明，若仅依赖单一数据结构（例如**BP**）作为适用于所有情况的解决方案，就会忽视许多有趣的机会。此外，访问类型会显著影响有序结构的性能，因此必须与数据结构相匹配。

B. 规则体求值：探测与探测结果迭代

从端到端的角度研究规则体求值中的连接操作后，让我们聚焦考察探测和探测结果迭代这两个部分的影响。例如，对于 **CI-BP**，前一部分是为确定符合条件条目的下界而进行的树遍历，后一部分则是扫描叶节点，直到找到上界。为了调整各阶段所需的工作量，我们现在将内关系 S 的大小固定为 100 MiB，并改变其中重复项的数量——重复项越多，探测阶段所需的工作量越少，探测结果迭代所需的工作量则越多。我们测试三种情况：每个键分别只出现一次、10 次和 100 次。在外关系 R中，每个键始终只出现一次，因此这三组实验中的连接伙伴数量固定不变。



(a)每个键 1×。(b)每个键 10×。(c)每个键 100×。图2：在改变S中重复项数量的同时，将连接拆分为探测和探测结果迭代。

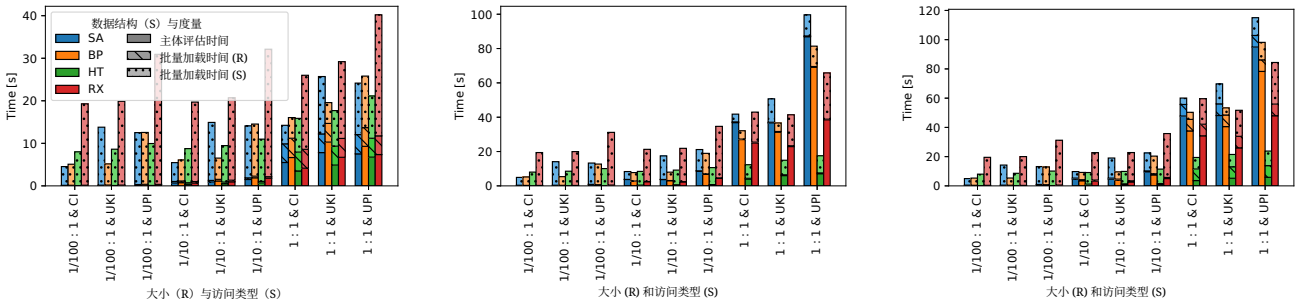
图 2 展示了三种配置下的主体评估时间。可以看出，重复项的数量不仅会影响运行时间（重复项越多，时间显著缩短），还会影响各物理表示之间的相对性能，这表明不同表示应对这种变化的效果各不相同。随着重复项增加，有序探测的优势会减弱，因为更多工作量转移到了探测结果迭代上。因此，在重复项很多时，引擎可以选择对 **SA**、**BP** 或 **RX** 这类有序数据结构进行无序探测，而不必强制使用稳健但构建成本更高、空间效率更低的 **HT**。从数据结构的角度看，四种变体之间的关系会随重复项数量而变化。**SA**、**HT** 和 **RX** 会将所有重复项连续存储，因此受益于重复项增加带来的更有效预取；相比之下，**BP** 扫描的是单个叶节点。从访问类型的角度看，**CI**、**UKI** 和 **UPI** 之间的相对差异，会随着探测和探测结果迭代的工作量比例而显著变化。当重复项较少、重点在探测阶段时，**CI** 和**UKI** 的性能非常接近，而**UPI** 则明显较差（对于无序探测）。当重复项较多时，**CI** 显著优于**UKI**和**UPI**，后两者此时的表现则非常接近。这表明，在前一种情况下，**UKI** 是性能上不逊于**CI**的替代方案；而在后一种情况下则显然不是，因为元组查找期间会出现大量缓存未命中和 TLB 未命中。如果空间允许，应使用 **CI** 。*C.* 物理表示的批量加载与主体求值

到目前为止，我们只关注了规则体求值。然而，在 Datalog 的世界中，EDB 往往是动态到达的，因此，物理表示的批量加载也成为整个程序求值的一部分。根据工作负载特征，批量加载或规则体求值可能会占据主导地位——这里的问题是，物理表示的选择是否应取决于这种大小关系。为 выяс明这一点，我们从前几小节的设置出发，并进行如下修改：不再改变内关系 S 的大小，而是将 S 固定为 1,000MiB，并将外关系 R 的大小从 10MiB（1 100:1 关系 R 与S之间的大小关系）经过 100MiB（1 10:1）增至 1,000MiB（1:1）。这样，在改变与之相关的规则体求值开销的同时，我们固定了正在探测的关系的批量加载开销。由此，我们可以调整两个阶段中哪一个占据主导地位。

图3展示了R的三种不同物理表示的结果，其中我们测试**覆盖索引-SA** ，将其作为在批量加载开销方面最轻量的、用于生成有序探测的方法，并将**行存储** 和**覆盖索引-哈希表** 分别作为生成无序探测的轻量级和重量级代表。我们将总时间拆分为R和S的批量加载时间以及主体评估时间。我们首先关注R 与S之间大小关系为1:1的情况。即便在这种主体求值开销较高的情况下，批量加载时间仍占总时间的很大一部分，这表明批量加载通常不能被忽略，而必须在查询开销与初始化开销大致平衡时加以考虑。更进一步，我们可以看到，对于有序探测（**覆盖索引-SA**），无论S采用何种物理表示，批量加载时间都主导了总时间；而对于无序探测（**行存储**和**覆盖索引-哈希表**），在1:1关系下尚未出现这种情况。这表明在决定具体物理表示时，还必须将探针顺序考虑在内。如果我们考察S的批量加载时间，可以看到它在不同物理表示之间差异很大：就访问类型而言，**UPI** 与有序数据结构结合时的初始化时间显著高于**覆盖索引**和**UKI**，后两者所需时间大致相同。这是因为在排序以及插入到正确位置的过程中，需要频繁地对底层行存储进行随机访问；而对于**哈希表**，在哈希计算期间只需访问一次store。聚焦于数据结构，**SA**由于只进行排序，初始化时间自然最短，其次是**BP**、**哈希表**和**RX**。**哈希表** 虽然不涉及排序步骤，却出人意料地昂贵；不过，它在批量加载期间会受到随机访问的影响。当同时考虑1 100:1和1 10:1的关系时，我们可以观察到主体求值与批量加载时间之间的反向关系，这表明确实应根据哪一方占主导来做出不同的决策。我们在开销更高的无序探测（**行存储**和**覆盖索引-哈希表**）中尤其能看到这一点：当关系为1 100:1时，由于初始化时间较低，**覆盖索引-SA** 是最佳选择。相反，当关系为1:1时，由于其主体评估时间较高，它成为最差的选项。还需注意，不对R 使用索引而仅使用**行存储** ，可以因构建更快而显著提升求值速度。*D.* 模式宽度和键宽度

此前研究的规则体 R(x,y), S(x,z) 在两个各有两个属性的关系之间，对一个属性进行连接。由于模式和键宽度也是变化范围很大的参数，我们现在分别调整它们，并研究它们对物理表示的影响。

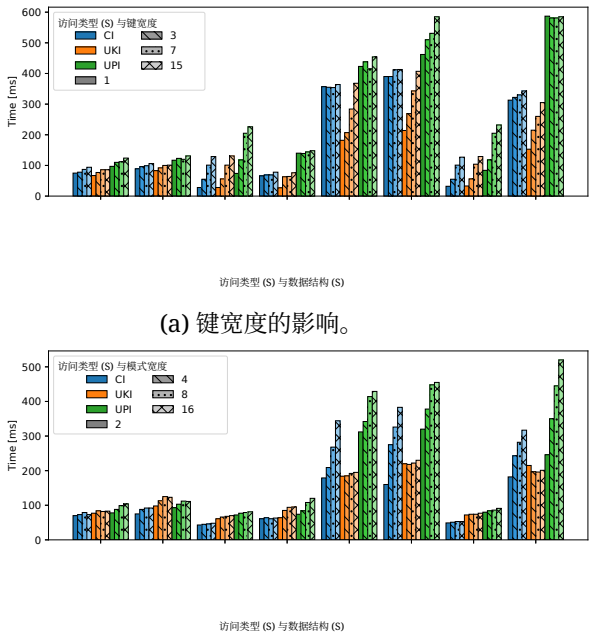
我们首先考察键宽度：将R 和S 的模式都扩展到 16 个属性，然后在 1、3、7 和 15 之间改变连接键的宽度。我们分别为R 和S 填充 100 万个元组（每个关系 122MiB ）。对于每种配置，我们



(a) 物理表示 (R)：**覆盖索引-SA**。 (b) 物理表示 (R)：**行存储**。 (c) 物理表示 (R)：**覆盖索引-哈希表**。

图 3：批量加载与主体评估。

相应地在S 上构建索引，即使用相应宽度的（复合）键。图 4a 展示了主体评估时间，仍分别采用有序探测和无序探测。可以看到，对于所有变体，运行时间都会随着键宽度的增加而增加，但增幅各不相同。**SA**和**BP** 与**CI** 和**UPI**结合时基本不受键宽度影响，因为它们无需在内部显式物化键（而是存储完整元组，仅需 124MiB），只会受到额外比较的影响。与**UKI**结合时情况有所不同：在无序探测下，性能与键宽度相关。不过，当键宽度显著小于模式宽度时，这种相关性也可能带来优势：对于**SA** 和**BP**，键宽度为 1 时的**UKI** 速度约为**CI**的两倍，表明非聚簇数据结构也可能具有性能优势。对于**RX**，无论访问类型如何，键越宽，基数树中的遍历路径就越长，因此其对键宽度的敏感性很高。**HT** 与**CI**和**UKI**结合时也同样如此，因为必须为哈希表的每个槽位显式物化键。这使**UKI**的大小从键宽度为 1 时的 140MiB（小型索引）增加到键宽度为 15 时的 248MiB。**HT** 需要更多空间：对于所有访问类型，键宽度为 1 时需要 190MiB；而对于**CI**和**UKI**，键宽度为 15 时则需要 415MiB。正如预期，在使用**UPI**时，键宽度不会影响内存占用。在图 4b 中，我们将内关系S 的模式宽度从 2 个属性调整为 16 个属性，同时将R 的模式固定为 2 个属性，并在第一个属性上执行连接。可以观察到，对于有序探测，所有变体基本不受模式宽度增加的影响。对于无序探测，情况则不同：**UKI** 保持非常稳定（因为它只处理键），而**CI**和**UPI** 变体在与有序数据结构**SA**、**BP**和**RX**结合时，性能会下降，因为它们需要在显著更大的内存区域上进行随机访问，从而导致更多缓存和 TLB 未命中。我们再次观察到**UKI**的优势：由于它独立于模式宽度，随着模式宽度增加，主体评估时间仍保持稳定，显著



(a) 键宽度的影响。

(b) 模式宽度的影响。

图4：键宽度和模式宽度的影响。

优于**覆盖索引**。对于被访问的内存，我们更仔细地考察**SA** ，并比较模式宽度为2和16时被访问内存大小的差异：**覆盖索引** (16→124 MiB)，**UKI** (16→16MiB)和**UPI** (24→132MiB)。因为在一次遍历过程中只需访问这16MiB，**UKI**无论模式宽度如何都能保持其性能。

E. 物理表示的共享

接下来，我们分析单个关系在程序中出现多次的情况。这种情况可能发生在单条规则的规则体内，也可能跨越多条规则。在这种情况下，如果一次连接参与的连接键是另一次连接参与的连接键的前缀，引擎既可以在较长的连接键上创建单一物理表示并在两种情况下使用它，也可以创建针对各个连接专门化的多个表示。乍一看，前者似乎更优，因为它的批量加载开销更小、内存占用也更小，但它也有两个潜在缺点：首先，即使一个连接键是顺序的前缀，也并不意味着两者以相同顺序进行探测。因此，在最坏情况下，除一次连接外，所有连接都会经历无序探测。其次，无序数据结构，例如

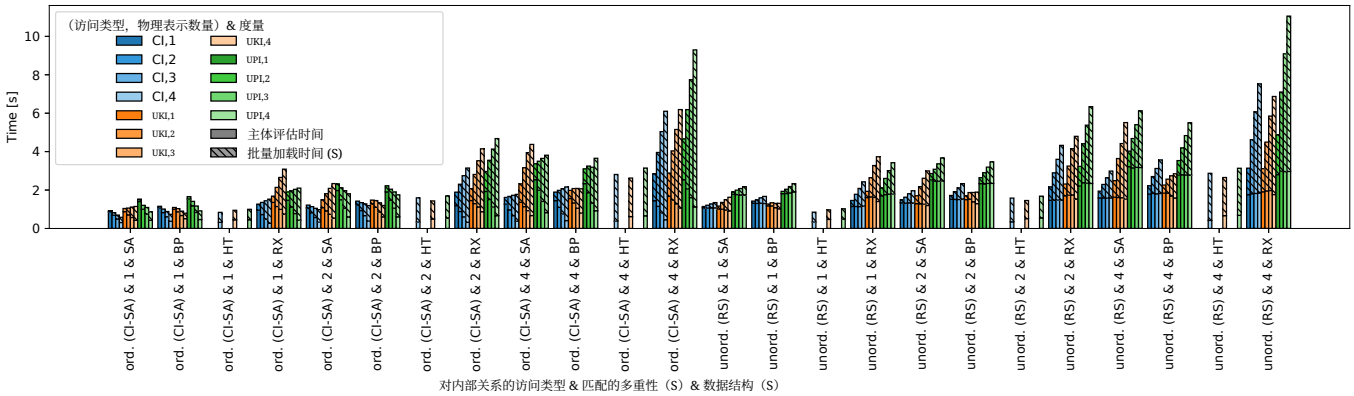


图 5：包含四条规则的规则交织，其中关系 S 出现在每条规则中，并使用 1、2、3 或 4 种物理表示。

HT完全不能用于在索引键前缀上操作的连接，因为它们不支持范围查找。为了引发并分析这一权衡，我们构造了如下程序，它由四条规则（即规则体）组成，这些规则都共享内关系S：

(1) R_1(A), S(A,S_2,S_3,S_4,S_5,S_6,S_7,S_8) (2) R_2(B,A), S(A,B,S_3,S_4,S_5,S_6,S_7,S_8) (3) R_3(C,A,B), S(A,B,C,S_4,S_5, S_6,S_7,S_8) (4) R_4(D,B,C,A), S(A,B,C,D,S_5,S_6,S_7,S_8)

每个规则体（i）都有其专属的外关系R_i ，它与S的前 i 个属性连接。请注意，为了引发前文提到的排序效应，我们改变了R一侧复合连接键的顺序。在实验中，

	物理表示的数量			
	1	2	3	4
$\mathbf{R_1}$	p_1(0 1 2 3)	p_1(0 1 2 3)	p_1(0 1 2 3)	p_4(0)
$\mathbf{R_2}$	p_1(0 1 2 3)	p_1(0 1 2 3)	p_3(1 0)	p_3(1 0)
$\mathbf{R_3}$	p_1(0 1 2 3)	p_2(2 0 1)	p_2(2 0 1)	p_2(2 0 1)
$\mathbf{R_4}$	p_1(0 1 2 3)	p_1(0 1 2 3)	p_1(0 1 2 3)	p_1(3 1 2 0)

表 II：每个物理表示的索引键 p_j已创建。绿色条目表示对 $\mathbf{R_i}$ 的探测将按照与该物理表示相同的顺序进行。

我们现在将为S 创建的物理表示数量从一个增加到四个。对于每个创建并共享的物理表示，我们选择其索引键，使其参与的某个连接中的探测有序。表 II 列出了所有创建的物理表示 $p_j(k_j)$ 及其索引键 k_j，其中 $k_j = 1\ 0$ 表示索引键由属性 1 后接属性 0 组成。绿色条目表示有序探测；随着专用表示数量的增加，有序探测也随之增加。图 5 展示了批量加载时间和主体评估时间：所有关系R_i 均包含 1M 个元组，同时改变物理表示的数量、 S 的物理表示类型，以及此前讨论的各个R_i 关系所导致的探针顺序。为研究创建更多物理表示所带来的额外批量加载时间，我们还将S 中每个键的重复项数量（重数）从 1 改为 2，再改为 4，因此 S分别有 1M、2M 和 4M 个元组。首先可以看到，对于有序探测，当重数为 1 和 2 时（除RX外的所有情况），增加独占物理表示能够缩短总体求值时间，尽管初始化开销有所增加；这充分利用了匹配的索引键。同时我们也看到，重数为 4 时，额外的批量加载开销可能抵消主体求值方面的优势。因此，对于有序探测，是否创建独占结构，必须考虑此前讨论的主体评估与批量加载开销之间的关系，以及内存占用增加可能带来的限制。对于无序探测而言，为有序结构**SA**、**BP**和**RX**创建独占物理表示无论如何都不划算，因为探测端不会利用可用的有序性。不过，可以为每个独占索引引入**HT**来解决这一问题；我们也评估了这种方案，但受篇幅限制，未在此展示。采用这一策略时，重数为 1 的所有情况下，增加独占**HT** 表示的数量都会缩短求值时间；重数为 2 时，**UPI** 也有此效果。对于重数为 4 的配置，**CI,1-SA** 的内存占用最低，为 246MiB；随着数量增加，这一数值线性上升至**CI,4-SA**的 984MiB，几乎与**CI,4-HT** 配置所需的内存相同。使用**UPI** 的四个索引在所有数据结构中总共仅占用 375 MB，**RX**除外（612 MB），这凸显了**UPI**通过共享索引节省内存的优势。

F. 递归关系的构建

现在，我们将讨论扩展到递归求值所特有的方面：关系通过不断加入新的事实而逐步构建。为此，我们聚焦于半朴素求值的一次迭代，研究如何识别出已找到的事实是否为新的且唯一的，并在满足条件时将其追加到递归关系中。为排除已分析过的主体求值的影响，我们在一组预先生成且具有特定特征的元组上测试递归关系的构建；这些特征代表实际主体求值所能找到的元组。识别并追加新的唯一元组有多种策略，各策略执行操作的顺序各不相同，因此我们评估了四种不同方案：

策略 **S1** （算法 2）与之前介绍的由 Souffl´e 实现的变体相似。作为替代方案，我们评估 **S2**

算法 2：S1.

```
对于每个元组 t执行
    如果 !R.contains(t) 则
        如果 !R_new.包含(t)
          then
            R_new.append(t)
R_new.已完成 append()
R.append(R_new)
R.finished append()
```

算法3: S2。

```
遍历元组 t执行
    如果 !R_new.contains(t) 则
        如果！ R.contains(t)
          then
            R_new.append(t)
R_new.finished append()
R.append(R_new)
R.finished append()
```

算法4：S3。

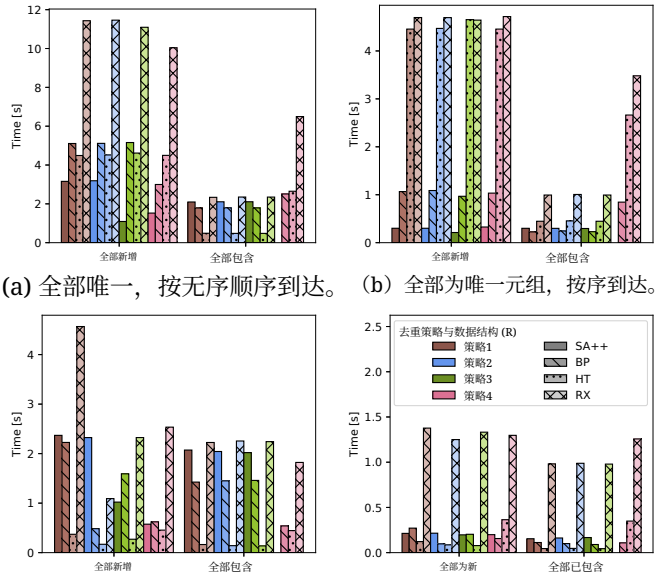
```
对于每个 元组 t执行
    如果 !R.contains(t) 则
        R.append(t)
        R_new.append(t)
R.finished append()
R_new.已完成 append()
```

算法 5：S4。

```
对于每个元组 t，执行
    如果!R_new.contains(t) 则
        R_new.append(t)
R_new.finished append()
对于每个元组 t∈R，执行
    如果R_new.contains(t) 则
        R_new.remove(t)
R.append(R_new)
R.finished a_ ppend
```

()图6：构建递归关系的四种策略。第一种变体 **S1** 已在算法 1 中展示。

（算法3），其中两个`contains()`包含性检查的顺序颠倒了。如果生成了许多重复项，这可能会带来积极影响，因为对于每个重复组，**S2** 只需对基关系执行一次包含性检查。在策略 **S3**（算法 4）中，我们不再将新元组的识别与将它们合并到基关系中的过程分开，而是尽可能早地将每个新元组合并到基关系中。这样可以避免对新关系进行单独遍历；不过，如果找到的元组到达时没有顺序，向基关系执行的`append()`操作也会无序。策略 **S4**（算法 5）省去了对基关系的包含性检查。相反，在构建新关系后，它会遍历基关系，并从新关系中移除所有已包含的条目。之后，再将新关系合并到基关系中。就物理表示而言，为了让分析易于理解，我们坚持采用**CI** 作为访问类型，并改变更为相关的数据结构；所选的物理表示用于`R`、`Rdelta`和`Rnew`。我们



(a) 全部唯一，按无序顺序到达。（b）全部为唯一元组，按序到达。

（c）全部为 100×，无序到达。（d）全部为 100×，按序到达。

图7：不同策略下递归关系的构建。建立一个具有两个属性的递归关系，并在其基关系`R` 中填入 6.5M 个唯一元组（总计 100 MiB）。随后，我们生成数量相同的已发现元组，并测试以下两种情况：这些元组要么全部是基关系中的新元组，要么已经全部包含在其中。此外，

我们改变已发现元组中重复项的数量。作为第三个维度，我们区分已发现元组是相对于`R`、`Rdelta`和`Rnew`所维护的顺序无序到达，还是按序到达。

图7展示了结果。总体来看，我们已经可以观察到两个有趣的现象：首先，在个别情况下，策略会显著影响性能，而且不同策略在不同情形下各有优势，因此接下来应考虑多种策略。然而，我们可以直接排除**S4**，因为在任何配置下，它的表现都不及其他策略。这简化了分析，因为我们不再需要考虑`remove()`的性能。其次，工作负载特征与策略的组合会影响哪种物理表示性能最佳。让我们详细看看这一点，先从所有找到的元组都唯一的情况说起。图7a展示了找到的元组以无序方式到达时的情况。由于 **S1** 和**S2** 对每个元组都分别在`R` 和`Rnew` 上执行无序的`contains()` 操作，然后分别向`Rnew` 执行无序的`append()`操作，以及向`R`执行有序的`append()` 操作，因此它们的性能相同。**S3** 需要执行更少的`contains()` 操作，但代价是只能执行无序的`append()`操作。**HT**利用了这一点，而**SA++**更是如此：在`append_finished()`发生时，它会将批次中追加的所有元组排序，显然胜过其他方案。与最差的选择相比，这几乎带来了 11×的性能提升。如果所有找到的元组都已包含在`R`中，情况则截然不同，因为此时所有策略都只需执行无序的`contains()` 操作。尽管 **S1** 和**S3** 需要执行的此类操作更少，但所有策略的表现都非常相近，其中**HT**胜过其余物理表示，因为它处理无序`contains()` 操作的能力最强。如果元组按顺序到达，如图7b所示，那么所有有序物理表示的速度都会显著更快。它们都能很好地处理有序的`contains()`操作，而**SA++** 凭借更出色的`append()` 性能，明显优于**BP**。**HT** 和**RX** 在这种情况下远非理想选择。如果按顺序到达的所有元组都已包含在内，那么与相应的无序情况相比，所有方法的性能都会进一步提升。然而，先前表现最佳的**HT** 此时自然成了糟糕的选择，而**BP**胜过其余有序物理表示，展现了其出色的`contains()`性能。当找到的元组中存在重复项时，情况又会发生显著变化，因为重复项的数量不仅会影响各种操作之间的权衡

还会影响contains() 和append() 操作的数量，也会影响**SA++**的append_finished() 操作所需的工作量。图7c展示了所有找到的元组都是新元组时的情况。对于**S3**和**S2** 而言，在处理重复项时，它们自然是最佳策略；此时**SA++** 不再是最佳选择，而是最差选择，因为其重复项消除操作代价高昂。因此，在这种情况下，**HT** 的速度比**SA++** 快约 5× 。相反，如果找到的元组已包含在内，重复项基本不会产生影响。最后，让我们看看图7d中有序情况下存在重复项的情形。有趣的是，面对新元组时，**HT** 仍胜过有序物理表示。总体而言，这些发现表明，必须谨慎选择递归关系的物理表示。

V. 自动物理表示选择

在第 IV 节中，我们仔细分析了各种工作负载特征对物理表示选择的影响。现在，我们将把获得的洞见落实到一种选择机制中，该机制能够为给定工作负载识别合适的物理表示。具体而言，对于某个工作负载，我们首先构建一种称为工作负载签名的表示，以精简的形式捕获该工作负载的所有相关特征。利用这一签名（它捕获了所执行操作的数量以及这些操作对齐的潜力），我们为每种物理表示计算一个合适的索引键，以确定操作的实际对齐方式。随后，我们使用三棵决策树，为整个工作负载选择数据结构、访问类型以及共享程度。

为了展示这一选择机制的有效性，接下来我们将其应用于程序分析和图分析领域的四个真实世界工作负载，以确定有效配置。表 III 概述了这些工作负载，并列出了相关状态。对于我们的贯穿示例 Andersen 分析，其情况如下：

points_to(y,x):-address_of(y,x) points_to(y,x):-assign(y,z),points_to(z,x) points_to(y,w):-load(y,x),points_to(x,z),points_to(z,w) points_to(y,w):-store(y,x),points_to(y,z),points_to(x,w)

Datalog 程序	数据集	顶点	边
程序分析			
Andersen 分析 [8], [28]	50000 [8]	-	-
图分析			
可达性 [8], [14], [29]	livej. [30], [31]	4,847k	68,993k
同代关系 [8], [14]	G10K-0.001 [32]	10k	100k
传递闭包 [8], [14]	G10K-0.01 [32]	10k	1,000k

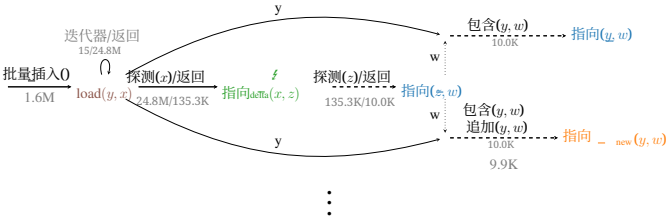
表 III：真实世界工作负载。

A. 工作负载特征

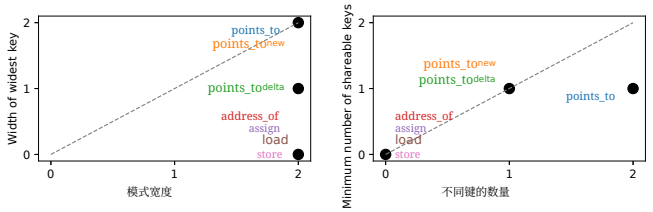
以 Andersen 分析为例，来看一下工作负载签名是什么样的。

对齐与操作数量（图 8a）。在所有实验中，我们观察到，性能受到两个属性的显著影响：（1）操作是否利用对齐；（2）操作的整体构成。关于（1），我们按如下方式刻画这种“对齐”：首先，对于程序的每次主体求值，从对最外层关系进行迭代开始，到构建规则头关系结束，我们创建一个对齐图。该对齐图记录了嵌套循环的每一层中，当前关系与下一层关系之间是否可能对齐。这实际上沿着嵌套循环创建了对齐路径。请注意，只有当 EDB 包含对齐属性时，才可能出现涉及多条边的对齐路径。关于（2），我们为先前创建的对齐图添加实际执行的操作构成，例如批量_load()、contains()、append()和probe() 操作，从而确定每种操作对整体求值的重要性。为了以易于理解的方式捕捉工作负载的这一特征，我们对求值代码进行插桩，并执行一次性能分析运行。

图 8a 展示了 Andersen 分析生成的八张图之一。在规则points_to(y,w) :- load(y,x), points_to(x,z),points_to(z,w)的对齐图中，load 关系在探测期间可能与$\texttt{points_to}_{\texttt{delta}}$ 关系完全对齐（实线），也可能在points_to 和$\texttt{points_to}_{\texttt{new}}$ 关系的contains()/append()操作期间部分对齐（虚线）。因此，为load选择何种物理表示，会对其他关系的物理表示选择产生重大影响。请注意，尽管从$\texttt{points_to}_{\texttt{delta}}$ 继续探测到points_to，但这种访问永远无法对齐（点线），因为$\texttt{points_to}_{\texttt{delta}}$的两个属性之间不存在固有的对齐关系。如果数据集中存在这种对齐关系，load 关系的影响会更大。查看执行的操作数量可以发现，针对$\texttt{points_to}_{\texttt{delta}}$的probe() 操作明显多于其余操作，表明它对物理表示的选择至关重要。



(a) 对齐情况和操作数量（针对一条规则展示）。



(b) 模式宽度与键宽度。　(c) 不同的键与可共享键。

Fig. 8: Sigandersen 分析的性质 ysis.

模式宽度和键宽度与不同键及可共享键数量的关系（图 8b 和图 8c）。在实验中，我们发现，物理表示的选择还会受到关系和键的宽度、这些关系和键在程序中的总出现次数，以及共享物理表示的潜力的显著影响。为捕捉这些方面，签名的第三和第四部分分别为每个

关系关联以下属性：模式宽度和最宽出现键的宽度，以及不同键的数量和可共享键的最小数量。*B.* 选择机制

现在让我们看看我们的选择机制如何利用签名来得出高效的选择。

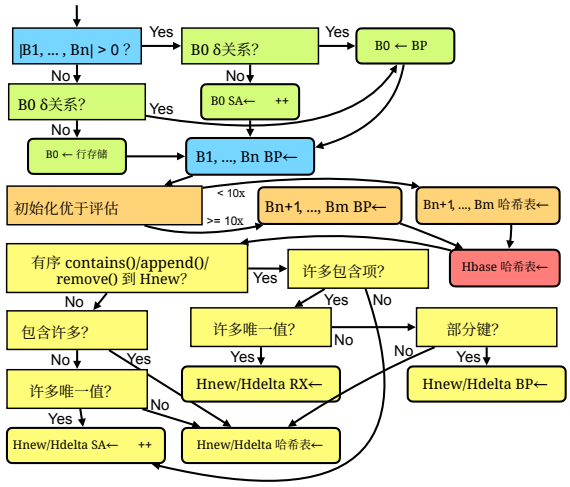
索引键。正如我们已观察到对齐访问的重要性，必须首先计算索引键，因为它们决定了对齐访问可能发生的位置（如果稍后选择有序数据结构）。如果存在多条不同的对齐路径，如我们的示例图中所示，我们希望选择最有可能从对齐访问中获益最多的路径。为了识别这条路径以及通向它的索引键选择，我们从规则体中最左侧的关系（在我们的例子中为`load(y,x)`）开始，分别在索引键仅设置为第一个属性以及仅设置为第二个属性时沿对齐路径前进，同时计算每条路径上执行的操作总数。然后，我们选择执行操作最多的那条路径的索引键。在我们的例子中，选择了第二个属性上的索引键，因为它会带来 24.8M 次有序的 `probe()` 操作，而第一个属性上的索引键仅导致对 `points_to` 和 $\texttt{points_to}_{\texttt{new}}$ 的 29K 次（部分）有序访问。随后所有后续关系的索引键都会被设置，以便最昂贵路径上的顺序尽可能长时间地保持完整。总体而言，对于 Andersen 分析，这会产生索引键 `address_of(0_1)`、`assign(1)`、`load(1)`、`store(0)`、$\texttt{points_to}_{\texttt{delta/new}}$`(0_1)`。请注意，此时 `points_to(0)` 和`points_to(0_1)` 都会被考虑并传递到下一阶段，因为它们允许共享。

决策树。计算出所有索引键后，我们的选择机制接下来必须确定物理表示形式，以及在多个出现位置上的共享程度。为此，它会在规则级别上逐步使用三棵决策树，将先前确定的最重要发现转化为实际决策，以平衡性能与空间效率。

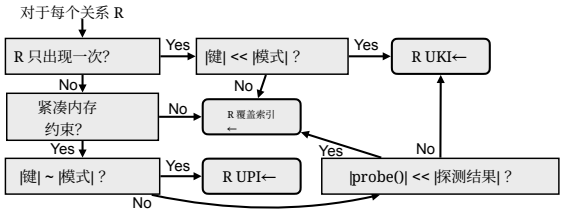
为了选择数据结构，我们首先将要实例化的规则拆分为不同部分。具体而言，在规则体中，我们区分最外层关系（B_0）、可利用对齐的关系（B_1 到 B_n），以及无法利用与 B_0对齐的关系（B_{n+1} 到 B_m）。在规则头中，我们区分

H_{base} 和 H_{new}。接着，图9a所示的决策树根据一系列标准，逐步实例化规则的各个部分。这些标准旨在：(a) 创建对齐访问；(b) 考虑初始化与求值的比率；(c) 考虑生成的元组的特征。请注意，针对 H_{new}和 H_{delta} 的冲突决策都会被纳入下一棵决策树；如果可行且值得推荐，可能会因此创建两种表示，或共享一种表示。对于 Andersen 分析，应用该决策树后，会为 B_0 EDB 选择 **SA**，以启动对齐访问路径：`address_of`、`assign`、`load`和`store` ；为 `points_to` 选择 **HT** ，以高效处理基关系中的 `contains()` 操作；并

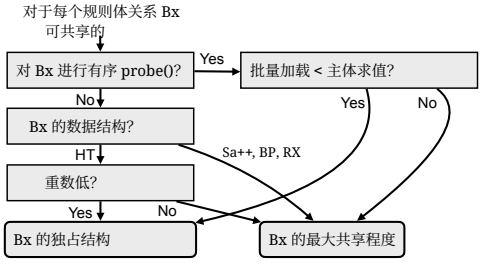
为高效处理有序探测而选择**BP** 以用于$\texttt{points_to}_{\texttt{delta}}$。选定数据结构后，接下来确定访问类型。在这里，除非索引键明显窄于模式，或存在严格的空间约束，否则我们明确优先选择**覆盖索引**。对于 Andersen 分析，我们不假设存在这样的空间约束。因此，所有物理表示均选择**覆盖索引** 。最终的决策树会针对每个可以共享的物理表示，决定是否应该共享。同样，我们会考虑关系中是否存在有序探测、初始化开销与求值之间的比率，也会考虑所选数据结构，以及在进行探测时的重数（若有相关信息）。因此，我们要么为该关系创建独占物理表示，要么尽可能多地共享。对于 Andersen 分析，该决策树仅应用于`points_to`，并决定创建两个独占物理表示：`points_to(0)` 和 `points_to(0_1)`（两者均为**覆盖索引-哈希表**）。



(a) 选择数据结构。



(b) 选择访问类型。



(c) 选择共享物理表示。

图 9：决策树。 B_0 是第一个主体关系， B_1...B_n利用对齐，B_{n+1}...B_m 无法利用对齐。

C. 实验结果和基线比较

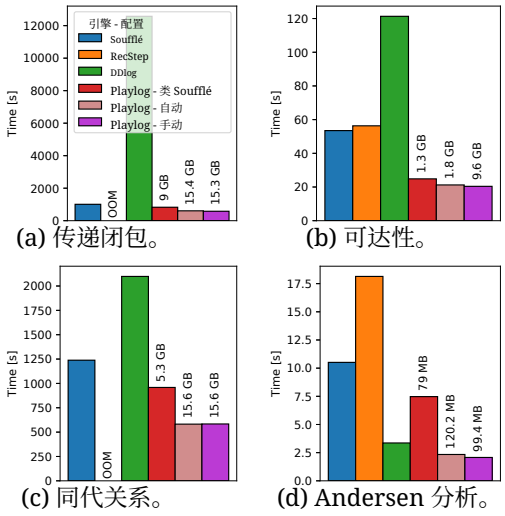
为评估选择机制的质量，我们进行了以下实验：对于四种真实世界工作负载中的每一种，我们都应用选择机制生成一种配置，称为 PlayLog-Auto。随后，我们将此配置与引擎中的 Soufflé 类配置进行比较；该配置对所有物理表示均仅使用**覆盖索引-BP**，并采用 Soufflé 的索引键，称为 PlayLog-Soufflé 类配置；我们还将其与最佳手工定制配置进行比较，称为 PlayLog-Hand。此外，为了更全面地衡量 PlayLog 的结果，我们还纳入了另外三个可比较的单节点内存引擎（即 Soufflé、RecStep 和 DDLog）对这些工作负载进行求值所需的时间。图 10 展示了全部四种工作负载的结果。我们可以看到，尽管这些引擎看似简单，实际性能却存在巨大差异。先来看 PlayLog 的结果。可以看到，在全部四种工作负载中，Soufflé 类配置的表现最差，而我们的手工定制配置性能最佳，加速比从 1.2 倍（可达性）到 3.5 倍（Andersen 分析）不等。这表明，对于真实世界工作负载，选择合适的物理表示确实很重要。此外，我们还可以看到，对于所有工作负载，我们的选择机制都能找出接近手工定制配置的配置，速度最多仅慢.1 倍。基线系统的表现始终不及 PlayLog 的自动选择；其中，RecStep 在同代关系和传递闭包工作负载下，因物化了大量中间连接结果而耗尽内存。除此之外，在半朴素求值的一次迭代中，RecStep 不会对新关系或基关系执行包含性检查，导致产生大量重复项，并需要耗费大量资源进行去重和集合差运算。DDLog 专为增量计算而设计，并维护时间索引（按逻辑时间索引），其中包含关系的先前版本。差分数据流（DDLog 的后端）执行的求值，并不等同于 Soufflé 和 PlayLog 使用的左深连接树。这可能导致递归规则求值更加频繁；同代关系和传递闭包中就出现了这种情况。但对于由许多规则定义的关系 points，也可能提升性能

– –

在 Andersen 分析中。 –

VI. 相关工作

[16] 对当前 Datalog 领域进行了出色的概述。关于物理表示，其他研究也考察了不同选项的影响：在 [10], [11],中，作者在 Soufflé 的背景下比较了并发 BTree 与并发基数树。在 [8], 中，作者提出了建立在 QuickStep 关系系统之上的 RecStep [33],，主要使用哈希表，但也实现了位矩阵，以更好地处理稠密图数据集。相对较早的系统 BDDBDDB [7] 使用了特殊的二元决策图。BigDatalog 基于 Apache Spark [34] 及专用 RDD，而 SociaLite 引入了尾嵌套表，这些表与基数树非常相似，并且在空间使用方面十分高效，适用于社交



(a) 传递闭包。 (b) 可达性。

(c) 同代关系。 (d) Andersen 分析。

图 10：生成配置的比较。

图。不同系统在这一问题上采取了截然不同的路径，这一事实表明我们的研究针对的是一个重要问题，尤其是因为这些表示从未在统一环境下进行过评估。关于 Datalog 中的查询优化，相关工作还研究了其他子问题：在 [35], 中，有研究提出了一种技术，通过共享物理表示来自动选择所需的最少索引数量，从而可能降低内存占用。然而，如第 IV-E 小节所示，这可能会阻碍相关优势的发挥。在 [36],中，作者为 Soufflé 提出了一种连接排序优化器。为此，作者扩展了 Soufflé，加入了一个离线性能分析遍历：该遍历按照程序的默认顺序运行程序，并收集一个必须额外提供的代表性数据集上的统计数据。基于这些统计数据，系统会确定最优连接顺序，同时将不同递归性在各次迭代之间的差异纳入考虑。我们的研究成果与其中介绍的技术正交。

VII. 结论

在本研究中，我们仔细分析了 Datalog 在选择不同物理表示方面的特殊需求。在高度灵活的 Datalog 引擎 PlayLog 中，我们首先研究了各种工作负载特征对各类物理表示的影响。基于研究发现，我们推导出决策树，并设计了一种自动选择机制，能够为真实世界工作负载识别高性能配置，其表现接近于经过人工优化的配置和功能完备的 Datalog 系统。

致谢。本研究由德国研究联合会（DFG）资助，项目编号 508316729。

AI 生成内容致谢

GitHub Copilot 在开发过程中协助完成了部分样板代码生成。

参考文献

[1] B. Scholz 等，“关于使用 Datalog 进行快速大规模程序分析”，《ACM 编译器构造 2016》系列，2016年。 [在线]。可访问：https://doi.org/10.1145/2892208.2892226

[2] T. Szab´o、S. Erdweg 和 M. Voelter，“IncA：一种用于定义增量程序分析的领域特定语言”，《IEEE/ACM ASE ’16 国际会议》系列，2016年。 [在线]。可访问：https://doi.org/10. 1145/2970276.2970298[3] S. Abiteboul 等，“异步离散事件系统的诊断：Datalog 来助力！”《ACM SIGMOD PODS ’05》系列，2005年。 [在线]。可访问：https://doi.org/
10 1145 1065167 1065214 。/。P. Alvaro 等，“
[4]
Dedalus：时间与空间中的 Datalog”，载于《第一届 Datalog Reloaded 国际会议论文集》，《Datalog’
10,
2010 [ ] 10 1007 978 3 642 24206 9 ¯ 》系列。在线。可访问：https://doi.org/。/——，“Boom 分析：探索面向数据的声明式云编程”，《ACM EuroSys ’10》系列，2010年。 [在线]。可访问：https://doi.org/
10 1145 1755913 1755937 。/。Datomic，“Datomic”，访问日期：2024年5月23日。 [在线]。可访问：https://www.datomic.com/ [7] M. S.
[6]
2024,
Lam、J. Whaley、V. B. Livshits、M. C. Martin、D. Avots、M. Carbin 和 C. Unkel，“将上下文敏感程序分析作为数据库查询”，载于《第二十四届 ACM SIGMOD-SIGACT-SIGART 数据库系统原理研讨会论文集》，《PODS ’05》系列。美国纽约州纽约：计算机协会，2005年6月，第 1–12 页。 [在线]。可访问：https://dl.acm.org/doi/ 10 1145 1065167 1065169 。/。
[8]
Z. Fan 等，“扩展内存 Datalog 处理：观察与技术”，《VLDB 会议录》，第 12 卷第 6 期，第 695–708 页，2019年。[在线]。可访问：http://www.vldb.org/pvldb/vol12/p695-fan.pdf [9] H. Jordan、P. Suboti´c、D. Zhao 和 B. Scholz，“针对 Datalog 专门化并行数据结构”，第 34 卷第 2 期，2022年。 [在线]。可访问：https://doi.org/10.1002/cpe.5643[10] ——，“Brie：一种用于并发 Datalog 的专用字典树”，载于《第十届多核与众核编程模型及应用国际研讨会论文集》，《PMAM’19》系列。美国纽约州纽约：计算机协会，2019年，第
31 40 [ ] 10 1145 3303084 3309490 – 页。在线。可访问：https://doi.org/。/。——，“一种用于并发 Datalog 求值的专用 B 树”，载于《第二十四届并行编程原理与实践研讨会论文集》，《PPoPP ’19》系列。美国纽约州纽约：计算机协会，2019年，第 327–339 页。 [在线]。可访问：https://doi.org/ 10 1145 3293883 3295719 。/。Y.
[11]
[12]
Liu、S. Mechtaev、P. Subotic 和 A. Roychoudhury，“由 Datalog 定义的静态分析引导的程序修复”，载于《第31届 ACM 欧洲软件工程会议与软件工程基础研讨会联合会议论文集》，《ESEC/FSE 2023》系列。美国纽约州纽约：计算机协会，2023年，第 1216–1228 页。 [在线]。可访问：https://doi.org/
10 1145 3611643 3616363 。/。A. Sahebolamri、T. Gilray 和
[13]
K. Micinski，“ 通过宏实现无缝演绎推理 ”，载于《第 31 届 ACM SIGPLAN 编译器构造国际会议论文集》，《CC 2022》系列。美国纽约州纽约：计算机协会，2022年，第
77 88 [ ] 10 1145 3497776 3517779 – 页。在线。可访问：https://doi.org/。/。A. Shkapsky、M. Yang、M. Interlandi、H.
[14]
Chiu、T. Condie 和 C. Zaniolo，“使用 Spark 上的 Datalog 查询进行大数据分析”，载于《2016年国际数据管理会议论文集》，《SIGMOD ’16》系列。美国纽约州纽约：计算机协会，2016年6月，第 1135–1149 页。 [在线]。可访问：https://dl.acm.org/doi/ 10 1145 2882903 2915229 。/。
[15]
J. Wang、M. Balazinska 和 D. Halperin，“无共享引擎中的异步容错递归 Datalog 求值”，《VLDB 会议录》，第 8 卷第 12 期，第 1542–1553 页，2015年8月。 [在线]。可访问：https://doi.org/
10 14778 2824032 2824052 。/。B. Ketsman 和 P.
[16]
Koutris，“现代 Datalog 引擎”，《数据库基础与趋势》，第 12 卷第 1 期，第 1–68 页，2022年。 [在线]。可访问：https://doi.org/
10 1561 1900000073 。/ M. Kifer 和 E. L. Lozinskii，“通过静
[17]
态过滤实现演绎数据库中的编译时查询优化”，《ACM 数据库系统汇刊》，第 15 卷第 3 期，第 385–426 页，1990年9月。 [在线]。可访问：https://doi.org/10.1145/88636.87121

[19] C. Zhang、L. Wang 和 M. Rigger，《发现 Datalog 引擎中的跨规则优化错误》，《ACM 程序语言》，第 8 卷，OOPSLA1 期，2024年4月。 [在线]。获取地址：https://doi.org/10.1145/ 3649815[20] J. M. Hellerstein，《声明式的命令式：分布式逻辑中的经验与推测》，SIGMOD 纪录，第 39 卷，第 1 期，第 5–19 页，2010年9月。 [在线]。获取地址：https://doi.org/10.1145/1860702. 1860704[21] L. Bellomarini、E. Sallinger 和 G. Gottlob，《Vadalog 系统：用于知识图谱的基于 Datalog 的推理》，《VLDB 会议录》，第 11 卷，第 9 期，第 975–987 页，2018年5月。 [在线]。获取地址：https://doi.org/
10 14778 3213880 3213888 . / . ( ) tlx - C 数据结构、算法及其
[22] 2025
++
他辅助工具集合。 [在线]。获取地址：https://github.com/tlx/tlx [23] (2025) Seq 库。 [在线]。获取地址：https://github.com/Thermadiag/ seq [24] V. Leis、A. Kemper 和 T. Neumann，《自适应基数树：面向主存数据库的 ART 索引》，载于《第 29 届 IEEE 国际数据工程会议（ICDE 2013）》，澳大利亚布里斯班，2013年4月8日至12日，C. S. Jensen、C. M. Jermaine 和 X. Zhou 编。IEEE 计算机学会，2013年，第 38–49 页。 [在线]。获取地址：https://doi.org/ 10 1109 2013 6544812 . /ICDE. . HP. ( ) [
[25] 2025
Judy IV 商店手册。在线。获取地址：https: // judy.sourceforge.net/application/shop interm.pdf [26] (2025) Sparsehash 库。 [在线]。获取地址：https://github.com/ sparsehash/sparsehash [27] (2025) SMHasher Murmur 实现。 [在线]。获取地址：https://github.com/aappleby/smhasher/ [28] L. O. Andersen，《C 程序设计语言的程序分析与特化》，博士论文，哥本哈根大学，丹麦哥本哈根，1994年，网址：https://www.cs.cornell.edu/courses/cs711/ 2005fa/papers/andersen-thesis94.pdf。[29] J. Gu、Y. H. Watanabe、W. A. Mazza、A. Shkapsky、M. Yang、L. Ding 和 C. Zaniolo，《RaSQL：在 Spark 上使用递归聚合 SQL 提升大数据分析的能力与性能》，载于《2019年国际数据管理会议论文集》，系列 SIGMOD ’19。美国纽约：计算机协会，2019年6月，第 467–484 页。 [在线]。获取地址：https://dl.acm.org/doi/
10 1145 3299869 3324959 . / . ( ) Livejournal 数据集。
[30] 2025
[ ]
在线。获取地址：http://snap.stanford.edu/ data/com-LiveJournal.html[31] J. Yang 和 J. Leskovec，《基于真实标注定义和评估网络社群》，载于《ACM SIGKDD 数据语义挖掘研讨会论文集》，系列 MDS ’12。美国纽约：计算机协会，2012年。 [在线]。获取地址：https://doi.org/
10 1145 2350190 2350193 . / . ( ) GTgraph 生成器。在线。获取
[32] 2025
[ ]
地址：https://www.cse.psu. edu/~kxm85/software/GTgraph/ [33] J. M. Patel、H. Deshmukh、J. Zhu、N. Potti、Z. Zhang、M. Spehlmann、H. Memisoglu 和 S. Saurabh，《Quickstep：一种基于纵向扩展方法的数据平台》，《VLDB 会议录》，第 11 卷，第 6 期，第 663–676 页，2018年2月。[34] M. Zaharia、R. S. Xin、P. Wendell、T. Das、M. Armbrust、A. Dave、X. Meng、J. Rosen、S. Venkataraman、M. J. Franklin、A. Ghodsi、J. Gonzalez、S. Shenker 和 I. Stoica，《Apache Spark：用于大数据处理的统一引擎》，ACM 通讯，第 59 卷，第 11 期，第 56–65 页，10月。
2016 [ ] 10 1145 2934664 。在线。获取
[35]
地址：https://doi.org/ 。/ P. Subotic 等，《大规模 Datalog 计算的自动索引选择》，《VLDB 会议录》，第 12 卷，第 2 期，2018年。 [在线]。获取地址：https://doi.org/ 10 14778 3282495 3282500 . / . S.
[36]
Arch 等，《为 Souffl´e 构建连接优化器》，系列 LOPSTR 2022，A. Villanueva 编，第 13474 卷，2022年。 [在线]。获取地址：https://doi.org/10.1007/978-3-031-16767-6 5

–