线性存在规则的半无知追赶终止性：实验研究

Marco Calautti 米兰大学
marco.calautti@unimi.it

Mostafa Milani 西安大略大学
mostafa.milani@uwo.ca

Andreas Pieris 爱丁堡大学 & 塞浦路斯大学
apieris@inf.ed.ac.uk

arXiv:2303.12851v1 [cs.DB] 22 Mar 2023

摘要

追赶过程是数据库中的一种基础算法工具，使我们能够利用约束（例如存在规则）进行推理，并有着广泛的应用。它以一个数据库和一组约束作为输入，并根据这些约束迭代地补全数据库。然而，一个关键挑战在于，该过程可能无法终止，因此需要解决这样一个问题：给定一个数据库和一组约束，判断它是否会终止。在本研究中，我们关注适用于实际实现的半无知版本追赶，以及线性存在规则这一具有多种应用的核心约束类别。在这一背景下，已有大量成熟的理论研究，提出了追赶终止的语法刻画、用于判断追赶是否终止的算法、精确的复杂度结果，以及追赶结果大小的最坏情况最优界（当结果有限时）。我们的主要目标是通过实验评估现有的追赶终止算法，以了解哪些输入参数会影响其性能，明确它们能否用于实际应用，并揭示其性能局限。

PVLDB 引用格式：
Marco Calautti、Mostafa Milani 和 Andreas Pieris。《线性存在规则的半无知追赶终止性：实验研究》。PVLDB，14(1): XXX-XXX，2020。doi:XX.XX/XXX.XX

PVLDB 研究成果可用性：
源代码、数据和/或其他成果已发布于 URL TO YOUR ARTIFACTS。

1 引言

〈style id='1'〉追赶过程 〈/style〉（简称追赶）是一种基础的算法工具，已成功应用于多个数据库问题，例如检查约束的逻辑蕴涵 [7, 19], 约束下的查询包含 [3], 计算数据交换解 [15], 以及本体查询回答 [11], 等。追赶过程以数据库 D和一组约束 Σ作为输入；在本文中，这些约束是〈style id='9'〉存在规则 〈/style〉（又称〈style id='11'〉元组生成依赖 〈/style〉（TGDs）），形式为 $\forall\bar{x}\forall\bar{y}(\phi(\bar{x}$〈style id='16'〉,〈/style〉$\bar{y}) \rightarrow \exists\bar{z}\,\psi(\bar{x}$〈style id='21'〉,〈/style〉$\bar{z}))$，其中$\phi$（体）和 ψ（头）是关系原子的合取；它生成实例 D_Σ，该实例是 D 的〈style id='27'〉泛模型 〈/style〉，并且 Σ，即一个能够同态嵌入到 D 和 Σ的其他所有模型中的模型。在某种意义上，D_Σ 充当了 D 和 Σ的所有模型的代表。这就是追赶过程在数据库领域无处不在的原因，正如 [13]中所讨论的。事实上，许多数据库问题只需给出一个泛模型即可解决。

1.1 追赶简介

粗略来说，追赶会按照 Σ 的元组生成依赖，向数据库 D 中添加新元组（可能涉及空值，这些空值充当存在量化变量的见证），并持续进行，直到 Σ 的所有元组生成依赖都得到满足。原则上，有三种不同的方式可以将这一简单思路形式化，它们会导出不同版本的追赶过程：

无视追赶。第一种方式导出无视追赶，具体如下：对于迄今构造的实例 I 中每一对由项组成的元组 $(\bar{t},\bar{u})$，如果 $\phi(\bar{t},\bar{u}) \subseteq I$，且元组生成依赖 $\sigma = \forall\bar{x}\forall\bar{y}(\phi(\bar{x},\bar{y}) \rightarrow \exists\bar{z}\,\psi(\bar{x},\bar{z}))$ 尚未在先前的追赶步骤中因同一对 $(\bar{t},\bar{u})$ 而应用过，则应用该依赖，并向 I 添加原子集 $\psi(\bar{t},\bar{v})$；其中，$\bar{v}$ 是一个由新项组成的元组，这些项均未出现在 I 中。

半无知追赶。第二种方法是对无视追赶的细化，它产生了半无知追赶，具体如下：对于迄今构造的实例 I 中每一对由项组成的元组 $(\bar{t},\bar{u})$，若 $\phi(\bar{t},\bar{u}) \subseteq I$，且此前的追赶步骤中尚未因元组对 $(\bar{t},\bar{u}\prime)$ 而应用过 σ，则应用元组生成依赖 $\sigma = \forall\bar{x}\forall\bar{y}(\phi(\bar{x},\bar{y}) \rightarrow \exists\bar{z}\,\psi(\bar{x},\bar{z}))$；其中 $\bar{u}$ 和 $\bar{u}\prime$可能不同，并将原子集 $\psi(\bar{t},\bar{v})$ 添加到 I，其中 $\bar{v}$ 是一个由不属于 I 的新项组成的元组。换言之，上述形式的元组生成依赖 σ 只会因变量 $\bar{x}$ 的某个特定见证 $\bar{t}$ 而应用一次。

受限追赶。第三种方法会导向受限（又称标准）追赶，具体如下：对于迄今构造出的实例 I 中每一对项元组 $(\bar{t},\bar{u})$，若 $(,)$，则应用元组生成依赖 $\forall\bar{x}\forall\bar{y}\ \phi\ \bar{x}\ \bar{y} \rightarrow \exists\bar{z}\psi\ \bar{x}\ \bar{z}$ $\phi\ \bar{t}\ \bar{u} \subseteq I$ $\bar{u}$ $(\,(\,,)\,(\,,))$；并且不存在这样的项元组 $\prime$，其来自 I，满足 $\psi(\bar{t},\bar{u}\prime) \subseteq I$，也就是说，该元组生成依赖尚未满足；然后向 I 添加原子集 $\psi(\bar{t},\bar{v})$，其中 $\bar{v}$ 是由 I 中不存在的新项组成的元组。

总而言之，（半）无关版本与受限版本的追赶过程之间，关键区别在于：前者在体满足时就会应用元组生成依赖，而后者只有在体满足但头不满足时才会应用元组生成依赖。

1.2 受限追赶与（半）无知追赶。

不难验证，一般而言，受限追赶构造的实例比（半）无知追赶构造的实例更小。事实上，很容易设计出这样一个例子：根据受限追赶，所有

本作品采用知识共享署名-非商业性使用-禁止演绎 4.0 国际许可协议授权。请访问 https://creativecommons.org/licenses/by-nc-nd/4.0/ 查看该许可协议副本。对于超出该许可协议范围的任何使用，请发送电子邮件至 info@vldb.org 获取许可。版权归所有者/作者所有。出版权已授予 VLDB基金会。VLDB基金会论文集，第 14 卷，第 1 期，ISSN 2150-8097。doi:XX.XX/XXX.XX

元组生成依赖都不应被应用，而（半）无知追赶却构造出一个无限实例。下面是摘自 [10] 的一个例子：

例*1.1*。 考虑数据库 $D = \{R(a,\ a)\}$ 和元组生成依赖

$$\forall x \forall y (R(x, y) \rightarrow \exists z\, R(z, x)).$$

受限追赶会检测到该数据库已经满足此元组生成依赖，而（半）无知追赶则会构造出实例

$$\{R(a, a), R(\bot_1, a), R(\bot_2, \bot_1), R(\bot_3, \bot_2), \ldots\},$$

其中 $\bot_1$, $\bot_2$, $\bot_3$, ...是（带标签的）空值。 ■

很容易构造这样的例子：半无知追赶不应用任何元组生成依赖，而无视追赶却会构造出无限实例。人们普遍认为，尽管无视追赶是一种非常有用的理论工具，但由于它会推导出大量冗余信息，进而导致实例规模非常大且往往是无限的，因此没有实际应用。至于追赶的其他变体，受限追赶显然优于半无知追赶，因为它通常构造的实例更小。当然，这种优势并非毫无代价：在每一步中，受限追赶都必须检查是否无法满足当前元组生成依赖的头部，而这在实践中可能代价高昂。最近有人指出，对于基于 RAM 的实现，受限追赶是合适的方案，因为生成较小实例所带来的收益足以抵消额外的检查工作，即检查元组生成依赖是否已经得到满足；例如参见 [8, 17]。然而，正如 [8], 所讨论的，基于关系数据库管理系统的受限追赶实现颇具挑战性，而半无知追赶则可以高效实现。因此，半无知追赶和受限追赶这两种版本，都是实际实现中值得采用的工具。

1.3 线性元组生成依赖与追赶终止

确实存在高效的半无知追赶和受限追赶实现，使我们能够通过采用基于物化的方法来解决数据库中的核心问题 [8, 17, 21, 24]。然而，要使这种方法在实践中可行，我们需要保证追赶能够终止，但事实并非总是如此。这一事实推动了关于追赶终止问题的大量研究：给定一个数据库 D 和一个由元组生成依赖组成的集合 Σ，检查在 D 上应用 Σ 后，半无知追赶或受限追赶是否终止。众所周知，一般来说，这是一个不可判定问题。 [13] 已证明受限追赶的情形不可判定；一年后， [20] 指出，同一证明也表明半无知追赶同样不可判定。然而， [13], 给出的不可判定性证明构造了一个复杂的元组生成依赖集合，超出了现有那些具有特定句法属性的良性元组生成依赖类别。这一观察引出了一个显而易见的问题：当我们聚焦于良性的元组生成依赖类别时，追赶终止问题是否可以通过算法求解?

线性元组生成依赖是 [11] 提出的一类性质良好的元组生成依赖，因其简单性以及在表达能力与复杂度之间取得了良好的平衡而备受关注。如果一个元组生成依赖的体中只有一个原子，而头可以是任意原子的合取，那么它就是线性的。如果体中的每个变量只出现一次，而头中的变量可以重复出现，则称这种元组生成依赖为简单线性，且不受任何限制。尽管乍看之下，（简单）线性元组生成依赖似乎表达能力很弱，但事实证明，它们的表达能力足以表示数据库完整性约束以及本体公理。特别是，我们知道，构成一类核心约束的参照完整性约束（又称包含依赖） [1], 可以轻松地表示为简单线性元组生成依赖。此外，重要的本体语言 DL-Lite$_R$ [12] 基于描述逻辑，也是 OWL 2 QL 的逻辑基础。OWL 2 QL 是 W3C委员会的Web本体语言（OWL）标准中一种广受欢迎的本体语言配置文件。DL-Lite$_R$ [12] 可以轻松嵌入简单线性元组生成依赖这一类中。

近年来，关于（简单）线性受限生成依赖下的追赶终止问题，已有大量研究。对于追赶的半无知版本，理论研究已相当成熟，涵盖了基于适当无环性概念刻画追赶何时终止的句法条件、检查追赶终止性的算法、精确的复杂度结果，以及对追赶实例大小的最坏情况下最优界（只要该实例有限） [9]。另一方面，对于受限追赶版本，我们目前只有一项可判定性结果：在假设线性受限生成依赖的头只包含一个原子的情况下，有一种运行时间为双指数时间的算法 [18]。研究进展上的这种显著差异，应归因于受限追赶下的追赶终止问题要困难得多。

1.4 主要目标

既然我们已经对（简单）线性受限生成依赖下的半无知追赶终止问题有了完整的理论认识，下一步就是通过实验评估所提出的算法，旨在了解哪些输入参数会影响其性能，厘清这些算法能否应用于实际场景，并揭示其性能局限。这正是本文的主要目标。请注意，我们不考虑受限追赶，因为在缺乏对相关问题充分理论认识的情况下，这么做还为时过早；后者是当前一项持续进行的研究活动的主题。

根据追赶终止领域的文献，我们可以借鉴两类算法，用于处理存在（简单）线性受限生成依赖时的半无视追逐终止问题，即基于物化的和基于无环性的 [9], 算法，具体如下：

基于物化的方法。基于物化的算法利用了追赶实例大小存在最坏情况下的最优界这一事实（只要追赶实例是有限的）。具体而言，给定一个数据库D 和一组 Σ （简单）线性受限生成依赖，我们有一个整数$k_{D,\ \Sigma}$使得，使用 Σ 对 D 进行追赶，当且仅当追赶实例的大小（即其中原子的数量）至多为$k_{D,\ \Sigma}$时，追赶才会终止。这立即引出一种概念上简单的追赶终止算法：只需使用 Σ 对 D 进行半无知追赶，并对生成的原子数量进行计数；如果计数超过$k_{D,\ \Sigma}$，则判定追赶不会终止；否则，判定其会终止。

基于无环性。另一方面，基于无环性的算法利用适当的无环性概念，对追赶何时终止进行句法刻画。具体而言，给定数据库 D 和一组 Σ （简单）线性受限生成依赖，我们知道，当且仅当 Σ 的依赖图不包含“环”时，D 在 Σ 下的追赶才会终止（依赖图是表示一组受限生成依赖的一种标准

方法，定义见第3节）。所谓“坏”环，是指它见证了这样一个事实：在 D 与 Σ 的追赶过程中，最终会进入循环追赶推导并导致不终止。这又带来了一种概念上简单的追赶终止算法：构造 Σ的依赖图；如果其中存在“坏”环，就判定追赶不会终止；否则，追赶会终止。

探索性分析表明，对于追赶终止性检查而言，基于物化的算法成本实在过高。这是因为从 [9] 出发进行追赶所得结果的最坏情况上界非常大，因此，通常算法必须先构造极其庞大的实例，才能安全地识别出追赶不会终止。另一方面，我们观察到，基于无环性的算法效率尚可，且仍有很大的优化和改进空间。因此，为了实现我们的主要目标，我们将重点放在基于无环性的算法上。

1.5 主要成果与挑战

我们的实验分析表明，对于简单线性元组生成依赖，影响基于无环性的算法运行时间的主要参数是输入的元组生成依赖集的大小，而输入数据库的大小并不起关键作用。值得注意的是，即使面对规模较大的简单线性元组生成依赖集（最多包含100K 条元组生成依赖），该算法也运行得非常快（只需数秒）。至于更有趣的线性元组生成依赖情形，我们的分析显示，基于无环性的算法由两个性质不同的组件构成。具体而言，其中一个组件依赖于数据库，其性能仅受数据库大小影响；另一个组件独立于数据库，其运行时间主要受 TGD 集的大小影响。值得注意的是，该算法的总体运行时间相当合理，这有力地表明，对于线性元组生成依赖而言，快速检查半无视追赶是否终止并非不切实际的目标。需要指出的是，该算法的端到端总运行时间大部分耗费在依赖于数据库的组件上，这表明我们今后的工作应着重改进这一组件。

技术挑战。 为了实现上述基于无环性的算法所取得的成果，我们必须克服若干技术挑战，这些挑战也带来了具有独立研究价值的结果：

- 若只是直接实现相关算法，就无法得出上述有见地的结论，因为这会导致性能不佳；第 4 节将对此进行讨论。因此，我们必须重新审视并改进 [9] 中的理论算法，以获得适用于高效实现的算法。改进后算法的底层实现细节见第 5 节。

- 为了对相关算法进行压力测试，我们必须合成生成数据库和元组生成依赖集合。然而，正如第6节所述，现有的数据和元组生成依赖生成器并不适用于我们的目的，因为它们无法调节某些对于评估我们的追赶终止算法至关重要的参数。为此，我们开发了自己的数据和元组生成依赖生成器，并用它们精心生成了用于实验分析的数据库和元组生成依赖。

实验基础设施和源代码可在 *https://github.com/mostafamilani/chase-termination* 获取。

2 预备知识

我们考虑三个互不相交的可数无限集C、N和V，它们分别由常量、(有标签的)空值和变量构成。我们将常量、空值和变量统称为项。对于整数 $n > 0$，我们用 $[n]$ 表示集合 $\{1, \ldots, n\}$。

关系数据库。模式 S 是一个有限的关系符号（或谓词）集，每个符号都关联一个元数。我们用 R/n 表示 R 的元数为 $n > 0$；也可以用 $\mathrm{ar}(R)$ 表示整数 n。S 的一个（谓词）位置是一个有序对 (R, i)，其中 $R/n \in$ 属于 S，且 $i \in [n]$，它本质上标识了 R 的第 i 个参数。我们用 pos(S) 表示 S 的位置集，即集合 $\{(R, i) \mid \mid R/n \in$ 属于 S 且 $i \in [n]\}$。S 上的一个原子是形如 $R(\bar{t})$ 的表达式，其中 $R/n \in$ 属于 S，且 $\bar{t}$ 是一个由项构成的 n 元组。事实是参数仅由常量构成的原子。对于 $\bar{t} = (t_1, \ldots, t_n)$ 中的变量 x，令 $\mathrm{pos}(R(\bar{t}), x) = \{(R, i) \mid \mid t_i = x\}$。我们用 $\mathrm{var}(R(\bar{t}))$ 表示 $\bar{t}$ 中变量的集合。记法 $\mathrm{pos}(\cdot, x)$ 和 $\mathrm{var}(\cdot)$ 可扩展到原子集。S 上的实例是由 S 上的原子构成、包含常量和空值的（可能无限的）集合。S 上的数据库是由 S 上的事实构成的有限集合。实例 I 的活动域记作 $\mathrm{dom}(I)$，是指在 I 中出现的项（常量和空值）的集合。对于单元素实例 $\{\alpha\}$，我们简单地用 $\mathrm{dom}(\alpha)$ 代替 $\mathrm{dom}(\{\alpha\})$。

替换与同态。将一组项 T映射到一组项 T'的替换是一个函数 $h\colon T \to T'$。此后，我们将替换 h 视为映射集合 $\{t \mapsto h(t) \mid t \in T\}$。将 h限制到 T的子集 S上，记作 $h_{|S}$，即替换 $\{t \mapsto h(t) \mid t \in S\}$。从原子集 A到原子集 B的同态，是一个从 A 中的项集映射到 B 中的项集的替换 h，满足 h 在 C 上为恒等映射；并且若 $R(t_1, \ldots, t_n) \in A$，则 $h(R(t_1, \ldots, t_n)) = R(h(t_1) \ldots, h(t_n)) \in B$。

元组生成依赖。元组生成依赖（TGD） σ是一个（不含常量的）句子 $\forall \bar{x} \forall \bar{y} (\phi(\bar{x}, \bar{y}) \to \exists \bar{z}\, \psi(\bar{x}, \bar{z}))$，其中 $\bar{x}$、$\bar{y}$和 $\bar{z}$是 V 中的变量元组，而 $\phi(\bar{x}, \bar{y})$ 和 $\psi(\bar{x}, \bar{z})$ 是非空的原子合取式，分别只包含来自 $\bar{x} \cup \bar{y}$和 $\bar{x} \cup \bar{z}$的变量。请注意，按照记号滥用的惯例，我们可以将变量元组视为变量集。我们将 σ写作 $\phi(\bar{x}, \bar{y}) \to \exists \bar{z}\, \psi(\bar{x}, \bar{z})$，并用逗号代替 $\wedge$来连接原子。我们将 $\phi(\bar{x}, \bar{y})$ 和 $\psi(\bar{x}, \bar{z})$分别称为 σ的体和头，记作 $\mathrm{body}(\sigma)$ 和 $\mathrm{head}(\sigma)$。元组生成依赖 σ 的前沿，记作 $\mathrm{fr}(\sigma)$，是变量集 $\bar{x}$，即同时出现在 σ的体和头中的变量。一组元组生成依赖 Σ的模式，记作 $\mathrm{sch}(\Sigma)$，是出现在 Σ 中的谓词集。若对上述元组生成依赖 σ，每当存在一个从 $\phi(\bar{x}, \bar{y})$ 到 I 的同态 h 时，就存在 $h' \supseteq h_{|\bar{x}}$，且它是从 $\psi(\bar{x}, \bar{z})$ 到 I 的同态，则称实例 I满足该依赖，记作 $I \models \sigma$；我们可以将原子合取式视为原子集。若实例 I满足元组生成依赖集 Σ，则记作 $I \models \Sigma$，即对每个 $\sigma \in \Sigma$，都有 $I \models \sigma$。

线性。 如果元组生成依赖只有一个体原子，则称其为线性的，由所有线性元组生成依赖的有限集构成的相应类记作 L。进一步地，如果变量在体原子中出现不超过一次，则称线性元组生成依赖为简单的；

相应的类记作 SL。显然，SL $\subsetneq$ L。

3 半忽略追赶过程

半忽略追赶（或简称追赶）以数据库D和一组元组生成依赖 Σ 为输入，并构造一个包含 D且满足 Σ的实例。在此背景下，一个核心概念是触发器。

定义 *3.1.* 给定一组元组生成依赖 Σ 和一个实例 I，对于 I上的 Σ，一个触发器是一个有序对（σ，h），其中 $\sigma \in \Sigma$，且 h是从体(σ)到 I的同态。有序对（σ，h）的结果记作 result(σ，h)，是集合 μ(头(σ))，其中 $\mu : \mathrm{var}(\text{头}(\sigma)) \to \mathbf{C} \cup \mathbf{N}$ 的定义如下：

$$\mu(x) = \begin{cases} h(x) & \text{if } x \in \mathrm{fr}(\sigma) \\ \perp^{x}_{\sigma, h_{|\mathrm{fr}(\sigma)}} & \text{otherwise} \end{cases}$$

其中$\perp^{x}_{\sigma, h_{|\mathrm{fr}(\sigma)}} \in \mathbf{N}$ 令 $T(\Sigma, I)$为 I上 Σ 的触发器集合。 ■

请注意，在 result(σ，h) 的定义中，头(σ)中的每个存在量化变量 x 都由 μ 映射到N 中的一个空值，其名称由触发器（σ，h）和变量 x 本身唯一确定。这意味着，给定一个触发器（σ，h），我们可以明确地构造原子集 result(σ，h)。追赶的核心思想是从数据库 D出发，对迄今构造出的实例反复应用给定元组生成依赖集合 Σ 中的触发器，直至无法继续。更准确地说，给定一个数据库 D 和一组元组生成依赖 Σ，令

$$\mathrm{chase}^0(D, \Sigma) = D,$$

并且对于每个 $i > 0$，令

$$\mathrm{chase}^i(D, \Sigma) = \mathrm{chase}^{i-1}(D, \Sigma) \cup \bigcup_{(\sigma,h) \in S} \mathrm{result}(\sigma, h),$$

其中 $S = T(\Sigma,$ 追赶$^{i-1}(D, \Sigma))$。最后，我们将 D关于 Σ 的追赶结果定义为以下（可能是无限的）实例

$$\mathrm{chase}(D, \Sigma) = \bigcup_{i \geq 0} \mathrm{chase}^i(D, \Sigma).$$

追赶终止。即使在非常简单的设置下，追赶结果也可能是无限的：容易看出，对于 $D = \{R(a, b)\}$ 和 $\Sigma = \{R(x, y) \to \exists z\, R(y, z)\}$，追赶($D$，$\Sigma$) 是无限的。这引出了以下问题，其参数为一类元组生成依赖 C，例如 SL（简单线性元组生成依赖类）和 L（线性元组生成依赖类）：

> 输入：一个数据库 D 和一组属于 C 的元组生成依赖 Σ。
> 问题：实例追赶(D，Σ) 是否有限?

这一问题近期已在 [9] 中针对简单线性类和线性元组生成依赖进行了研究。有趣的是，对于这两类，追赶结果的有限性都已通过利用非一致弱无环性的概念，以句法方式刻画。我们接下来回顾这一无环性概念，然后介绍 [9], 中建立的刻画；这些刻画进而导出了用于检验追赶结果是否有限的简单算法。请注意，为了表述清晰，在本文余下部分中，我们假设元组生成依赖具有非空前沿，即假设至少存在一个变量，同时出现在元组生成依赖 σ的体(σ)和头(σ)中。这一假设不失一般性，因为给定一个数据库 D 和一组 Σ 元组生成依赖，我们可以通过稍微修改 Σ，轻松构造出一组具有非空前沿的元组生成依赖 Σ'，使得当且仅当追赶(D, Σ)有限时，追赶(D, Σ')也有限。

非一致弱无环性。弱无环性在 [15] 中被提出，作为数据交换的主要形式化框架，它保证对于每个输入数据库，追赶结果都是有限的。非一致弱无环性是 [9] 中提出的弱无环性的数据库依赖变体。下面给出形式化定义。首先回顾一组元组生成依赖 Σ 的依赖图这一概念，该图定义为有向多重图 $\mathrm{dg}(\Sigma) = (N, E)$，其中 $N = \mathrm{pos}(\mathrm{sch}(\Sigma))$，且 E仅包含以下边。对于每个头部为 $\mathrm{head}(\sigma) = \{\alpha_1, \ldots, \alpha_k\}$ 的元组生成依赖 $\sigma \in \Sigma$，对于每个 $x \in \mathrm{fr}(\sigma)$，以及对于每个位置 $\pi \in \mathrm{pos}(\mathrm{body}(\sigma)\ x)$：

- 对于每个 $i \in [k]$，以及每个 $\pi' \in \mathrm{pos}(\alpha_i, x)$，都存在一条普通边（$\pi, \pi'$） $\in E$。
- 对于每个存在量化变量 z（属于 σ、$i \in [k]$ 和 $\pi' \in \mathrm{pos}(\alpha_i, z)$），都存在一条特殊边（$\pi, \pi'$） $\in E$。

我们还需要定义一个谓词何时可从另一个谓词到达。给定谓词 R, $P \in \mathrm{sch}(\Sigma)$，如果 $R = P$，或者在 $\mathrm{dg}(\Sigma)$ 中存在一条路径，从形如（R, i）的位置到形如（P, j）的位置，则 P可从 R（关于 Σ）到达。给定一个数据库D，若 $\mathrm{dg}(\Sigma)$ 中存在一条（不一定简单且可能含环的）路径 C，并且存在一个原子 $R(\bar{t}) \in D$ 以及 C中的一个形如（P, i）的节点，使得 P可从 R到达，则称该路径为 D-支持的。现在我们可以回顾（非一致的）弱无环性了。

定义 *3.2.* 考虑一个数据库 D 和一组元组生成依赖 Σ。如果 $\mathrm{dg}(\Sigma)$ 中不存在带有特殊边的 D支持环，我们就称 Σ 关于 D-弱无环，或称 D-弱无环。如果 $\mathrm{dg}(\Sigma)$ 中不存在带有特殊边的环，我们就称 Σ 弱无环。

■

刻画追赶的有限性。 不难证明，只要一组元组生成依赖 Σ（不一定是线性的）是 D-弱无环的，那么实例追赶 chase(D, Σ) 就是有限的。换言之， Σ 的 D-弱无环性是 chase(D, Σ) 有限的充分条件。更有意思的是，假设 Σ 是一组简单线性元组生成依赖，那么 Σ 的 D-弱无环性也是 chase(D, Σ) 有限的必要条件。这就导出了 [9]中确立的以下刻画：

定理 3.3. 考虑一个数据库 D 和一组 $\Sigma \in$ 简单线性类元组生成依赖。则追赶追赶(D, Σ) 有限，当且仅当 Σ 是 D-弱无环的。

对于线性元组生成依赖，事实证明，非一致弱无环性不足以刻画追赶实例的有限性。下面是 [9] 给出的一个例子，用来说明这一事实：

例 *3.4.* 考虑数据库 $D = \{R(a, b)\}$以及由（非简单的）线性元组生成依赖组成的单元素集合 Σ

$$R(x, x) \to \exists z\, R(z, x).$$

很容易看出， Σ 在 D上没有触发器。这意味着追赶(D, Σ) $= D$ 是有限的，而 Σ 并非 D-弱无环的。 ■

为了获得类似于定理 3.3 的刻画， [9] 的作者使用简化 技术将线性元组生成依赖转换为简单线性元组生成依赖，同时保持追赶实例的有限性。下面我们回顾这一技术。设 $\bar{t} = (t_1, \ldots\ldots, t_n)$ 是由项组成的元组（这些项不一定互不相同）。我们

用 unique($\bar{t}$) 表示从 $\bar{t}$ 得到的元组，其中只保留 $\bar{t}$ 中每个项的首次出现。例如，若 $\bar{t} = (x, y, x, z, y)$，则 unique($\bar{t}$) $= (x, y, z)$。对于每个 $i \in [n]$，$\bar{t}$ 中 t_i 的标识符（记作 $\mathrm{id}_{\bar{t}}(t_i)$）是一个整数，用于标识 unique($\bar{t}$) 中 t_i 出现的位置。我们用 id($\bar{t}$) 表示元组 $(\mathrm{id}_{\bar{t}}(t_1) \ldots, \mathrm{id}_{\bar{t}}(t_n))$。例如，若 $\bar{t} = (x, y, x, z, y)$，则 id($\bar{t}$) $= (1, 2, 1, 3, 2)$。对于原子 $\alpha = R(\bar{t})$，α 的简化（记作 simple(α)）是原子 $R\mathrm{id}\,(\bar{t})$ (unique($\bar{t}$))；而 α 的形状（记作 shape(α)）是谓词 $R\mathrm{id}\,(\bar{t})$。我们可以自然地谈论原子集的简化和形状。对于变量元组 $\bar{x} = (x_1, \ldots, x_n)$，$\bar{x}$ 的一个特化是一个从 $\bar{x}$ 到 $\bar{x}$ 的函数 f，满足 $f(x_1) = x_1$，并且对于每个 $i \in \{2, \ldots, n\}$，$f(x_i) \in \{f(x_1) \ldots, f(x_{i-1})\ x_i\}$。我们用 $f(\bar{x})$ 表示 $(f(x_1) \ldots, f(x_n))$。现在我们可以回顾如何将一组线性元组生成依赖转换为一组简单线性元组生成依赖。

定义 *3.5.* 考虑如下形式的线性元组生成依赖 σ

$$R(\bar{x}) \rightarrow \exists \bar{z}\, \psi(\bar{y}, \bar{z}),$$

其中 $\bar{y} \subseteq \bar{x}$，且 f 是 $\bar{x}$的一个特化。由 f 诱导的 σ简化是简单线性 TGD

$$\mathrm{simple}(R(f(\bar{x}))) \rightarrow \exists \bar{z}\, \mathrm{simple}(\psi(f(\bar{y}), \bar{z})).$$

我们用 simple(σ) 表示由某个 $\bar{x}$的特化所诱导的 σ 的所有简化构成的集合。对于由元组生成依赖组成的集合 $\Sigma \in \mathrm{L}$，Σ 的简化定义为集合

$$\mathrm{simple}(\Sigma) = \bigcup_{\sigma \in \Sigma} \mathrm{simple}(\sigma)$$

该集合仅由简单线性元组生成依赖构成。 ■

现在我们可以回顾线性元组生成依赖的追赶实例有限性的刻画，该刻画在 [9], 中确立。它与简单线性元组生成依赖的刻画类似，关键区别在于，我们首先需要简化数据库和线性元组生成依赖集合:

定理 3.6. 考虑数据库 D和由元组生成依赖组成的集合 $\Sigma \in \mathrm{L}$。那么，追赶(D, Σ) 有限，当且仅当 simple(Σ) 是simple(D)-弱无环的。

显然，定理 3.3 和定理 3.6 给出了用于检查追赶实例是否有限的简单算法。具体而言，给定数据库 D 和一组简单线性元组生成依赖 Σ，我们只需检查 Σ 是否为 D-弱无环；若是，算法返回 true，否则返回 false。对于线性元组生成依赖集合 Σ，情况也一样，不同之处在于算法需要先简化 D 和 Σ，然后再执行无环性检查。我们的目标是对上述算法进行实验评估，以了解哪些输入参数会影响其性能，明确它们能否应用于实际场景，并揭示其性能局限。当然，若朴素地实现上述算法，尤其是针对需要执行代价高昂的简化操作的线性元组生成依赖，性能会很差，因此对实现我们的目标帮助不大。因此，我们需要设法将上述理论算法转化为可高效实现的实用算法。下一节将讨论这一主题。

4 实用终止算法

我们首先介绍算法 IsChaseFinite[SL]，它以数据库 D 和一组简单线性元组生成依赖 Σ 为输入，并检查 Σ 是否为 D-弱无环；这等价于判断实例追赶 chase(D, Σ) 是否有限。请注意，在依赖图中朴素地搜索“坏”环代价过高，因为我们可能需要遍历指数多个环。因此，IsChaseFinite[SL]依靠一种改进的机制，来搜索带有特殊边的强连通分量。接下来，我们给出一个类似的算法，称为 IsChaseFinite[L]，用于线性元组生成依赖。该算法主要对给定的数据库 D 和线性元组生成依赖集合 Σ 进行简化，然后检查 simple(Σ) 是否为 simple(D)-弱无环；这等价于判断 chase(D, Σ) 是否有限。不过请注意，IsChaseFinite[L] 依赖于一种改进的简化方法：利用给定的数据库 D对 Σ 进行动态简化，而不是像定义 3.5 中那样在不考虑任何数据库的情况下进行静态简化。动态简化的目标是仅保留 simple(Σ) 中那些确实需要用于检查追赶实例是否有限的元组生成依赖。请注意，本节将从高层次介绍上述算法，不深入讨论实现细节；后者将在第 5 节讨论。

4.1 简单线性元组生成依赖

在介绍 IsChaseFinite[SL], 之前，我们需要先引入几个辅助概念。在有向图 G 中，强连通分量（SCC）是一个极大子图，其中任意一对节点之间都存在一条（有向）路径。依赖图中的特殊强连通分量是至少包含一条特殊边的强连通分量。现在我们可以讨论 IsChaseFinite[SL],，其流程如算法 1 所示。首先，它为输入集 Σ 中的简单线性元组生成依赖构建依赖图 G（第 1 行）。随后，它将 G 的特殊强连通分量收集到集合 S 中（第 2 行）；显然，这些分量可能形成违反非一致弱无环性条件的“坏”环。当然，要发生这种情况，特殊强连通分量中的某些节点（即谓词位置）必须受到给定数据库 D 的支持，定义见第 2 节。为检查这一点，该算法首先从 G 的每个特殊强连通分量 C 中各选取一个节点 v_C，并将其收集到集合 P 中（第 3 行）；如何选择 v_C 并不重要。然后，它检查 D 是否支持 P 中的任何节点（第 4 行）。如果是，则 G 中存在一个由 D 支持且包含特殊边的环，因此算法返回 false；否则返回 true。BuildDepGraph、FindSpecialSCC 和 Supports 的实现细节分别见第 5.1、5.2 和 5.3 节。根据定理 3.3，IsChaseFinite[SL] 的正确性如下:

引理 4.1。 考虑一个数据库 D和一组 $\Sigma \in$ 简单线性类 元组生成依赖。此时，IsChaseFinite[简单线性类](D, Σ) = 为真 当且仅当追赶(D, Σ) 有限。

4.2 线性元组生成依赖

尽管算法 IsChaseFinite[SL] 结合简化技术（见定义 3.5），可以立即得出一种用于检查线性元组生成依赖的追赶实例是否有限的简单算法，但对简化技术进行朴素实现会导致性能不佳。事实上，我们针对现实世界中的线性元组生成依赖数据集进行了探索性实验，发现朴素实现无法扩展，因为算法很快就会

算法 1： IsChaseFinite[SL]

输入： 一个数据库 D 和一组 $\Sigma \in$ 简单线性类元组生成依赖

输出： 若追赶(D, Σ)有限，则为 true；否则为 false

$G \leftarrow$BuildDepGraph(Σ);

$S \leftarrow$FindSpecialSCC(G);

$P \leftarrow \bigcup_{C \in S}\{v_C\}$;

如果 Supports(D, P, G) **则返回** false;

返回true

因处理大型元组生成依赖集而耗尽内存。这是因为在不考虑底层数据库的情况下，对线性元组生成依赖集 Σ 进行静态简化，会产生一个规模呈指数级增长的简单线性元组生成依赖集；特别是，集合 simple(Σ) 的大小相对于 sch(Σ) 中谓词的最大元数呈指数级增长。因此，由于 simple(Σ) 的依赖图规模极大，超出主存容量，算法 IsChaseFinite[SL] 变得不切实际。

动态简化。我们通过考虑底层数据库来细化简化的概念，由此引出动态简化技术。具体而言，给定数据库D 和一组线性元组生成依赖 Σ，目标是定义一个集合 simple$_D$(Σ)，它是 simple(Σ) 的子集，并具有两个关键性质：

(1) 当且仅当实例追赶(simple(D),simple(Σ)) 是有限的时，实例追赶(simple(D), simple$_D$(Σ)) 也是有限的；这实质上表明，动态简化技术保留了追赶的有限性。

(2) 通常，集合 simple$_D$(Σ) 比通过静态简化 Σ得到的集合 simple(Σ) 小几个数量级。

下面的引理 4.3 证明了性质 (1)。性质 (2) 无法从数学上证明，因为在某些情况下，静态简化和动态简化会生成相同的线性元组生成依赖集合。然而，我们通过实验验证，对于现有数据库以及文献中的元组生成依赖集合（实际上，即第 9 节中使用的那些），动态简化后的元组生成依赖集合的大小平均比相应的静态简化后的集合小 5 倍。绝对差异有所不同；在最理想的情况下，动态简化后的集合最多可小 1000 倍。

动态简化的核心思想是利用给定数据库中出现的原子的形状来指导简化。更准确地说，给定数据库 D 和一组 Σ 线性元组生成依赖，我们首先收集能够通过 Σ中的元组生成依赖从 shape(D) 推导出的形状；我们将这个集合记为 Σ(shape(D))。然后，simple$_D$(Σ)从集合 simple(Σ) 中仅保留那些体原子的谓词属于 Σ(shape(D))的简单线性元组生成依赖，因为只有这些元组生成依赖才能在构造实例追赶(simple(D),simple(Σ))时应用。只要输入数据库是 D，simple(Σ)中的所有其他元组生成依赖都是多余的，因为它们在构造追赶(simple(D),simple(Σ))时永远不会被应用。我们接下来将这一思想形式化。为此，我们需要引入一些辅助概念。

对于模式 S，令 shape(S) 表示所有提及 S 中某个谓词的形状的集合，也就是说，这个有限形状集合

$$\text{shape(S)} = \left\{R_{\text{id}(\bar{t})} \mid R \in \text{S and } \bar{t} \in \left(\mathbf{C}^{\text{ar}(R)} \cup \mathbf{V}^{\text{ar}(R)}\right)\right\}.$$

对于形状集合 $S \subseteq$shape(S)，由 S 所诱导的数据库记为 DB[S]，其定义为数据库 $\{R(\text{id}(\bar{t})) \mid R_{\text{id}(\bar{t})} \in S\}$。例如，假设 $S = \{R(1,2)\ P(1,1,2)\}$，则 DB[S] $= \{R(1, 2)\ P(1, 1, 2)\}$。现在考虑线性元组生成依赖 $\sigma = R(x_1, \ldots, x_n) \rightarrow \exists \bar{z}\, \psi(\bar{y}, \bar{z})$，并令 h 为从 $\{R(x_1, \ldots, x_n)\}$ 到 $\{R(i_1, \ldots, i_n)\} \subseteq$DB[shape($\{R\}$)] 的一个同态。元组 $(x_1, \ldots, x_n)$ 的 h-特化是 $(x_1, \ldots, x_n)$ 的（去重）特化 f，满足对于任意 $i, j \in [n]$，$f(x_i) = f(x_j)$ 当且仅当 $h(x_i) = h(x_j)$。例如，假设 h 是从 $\{R(x, y, x, z)\}$ 到 $\{R(1, 1, 1, 2)\}$ 的同态，则 (x, y, x, z) 的 h-特化是函数 f，满足 $f(x) = x$，

$f\ y = x \qquad f\ z = z$ ( ) ，且 ( ) 。现在我们可以继续形式化动态简化。

考虑一组 Σ线性元组生成依赖和一组形状 $S \subseteq$shape(Σ)；为简洁起见，我们将 shape(sch(Σ)) 记为 shape(Σ)。形状 $R_{\text{id}(\bar{t})} \in$shape(Σ) 是 S 和 Σ 的直接后果，如果：

(1) $R_{\text{id}(\bar{t})} \in S$，或者 (2) 存在一个元组生成依赖 $R(\bar{x}) \rightarrow \exists \bar{z}\, \psi(\bar{y}, \bar{z})$，属于 Σ，并且存在一个同态 h，从 $\{R(\bar{x})\}$ 到 数据库[S]，使得 $R_{\text{id}(\bar{t})}$出现在由 h-特化诱导的 σ 的简化的头部中。

简单来说，第 (2) 项告诉我们，在 simple(Σ) 中存在一个形如 $R_{\text{id}(\bar{t}')}(\bar{x}) \rightarrow \exists \bar{z} \ldots, R_{\text{id}(\bar{t})}(\bar{y})$…的元组生成依赖，其中 $\bar{t}' \in \Gamma_\Sigma(\bar{x})$。其直接后果算子是一个函数：$2^{\text{形状}(\Sigma)} \rightarrow 2^{\text{形状}(\Sigma)}$（和通常一样，$2^X$ 表示集合的幂集X），满足

$\Gamma_\Sigma(S) = \left\{R_{\text{标识符}(\bar{t})} \mid R_{\text{标识符}(\bar{t})} \text{是由 } S \text{和 } \Sigma\right\}$.直接推出的

通过反复应用上述算子，我们可以计算出使用 Σ的元组生成依赖从 S 能够推导出的形状。形式化地，

$$\Gamma_\Sigma^0(S) = S \quad \text{and} \quad \Gamma_\Sigma^i(S) = \Gamma_\Sigma(\Gamma_\Sigma^{i-1}(S)) \ \text{ for } \ i > 0$$

最后，我们令

$$\Sigma(S) = \bigcup_{i \geq 0} \Gamma_\Sigma^i(S).$$

乍看之下，构造 Σ(S)需要无限次迭代。然而，由于 $\Sigma(S) \subseteq$ 形状(Σ)，最坏情况下，经过 |形状(Σ)| 次迭代后即可得到Σ(S)。实际上，很容易验证 $\Sigma(S) = \Gamma_\Sigma^{|\text{简单的}(\Sigma|)}(S)$。因此，由于形状(Σ)是有限的，我们可以得出结论：经过有限步后即可得到 Σ(S)。现在，我们已经具备了正式定义动态简化所需的全部要素。

定义 4.2。考虑一个数据库 D和一个线性元组生成依赖集 Σ。[1]动态简化 Σ 相对于 D （或 D-简化Σ），记作 simple$_D$(Σ)，定义为集合

$$\{\text{simple}(R(f(\bar{x}))) \rightarrow \exists \bar{z}\, \text{simple}(\psi(f(\bar{y}), \bar{z})) \mid R(\bar{x}) \rightarrow \exists \bar{z}\, \psi(\bar{y}, \bar{z}) \in \Sigma \text{ and } f \text{ is the } h\text{-specialization of } \bar{x}$$

对于某个从 $\{R(\bar{x})\}$ 到 数据库(Σ(形状(D)))$\}$ 的同态 h

仅由简单线性元组生成依赖组成。 ∎

不难验证，Σ 的 D-简化本质上收集了 simple(Σ) 中所有这类元组生成依赖：它们的

[1]不失一般性，我们假设 D 的原子只提及 sch(Σ) 中的谓词，因此，形状(D) ⊆ 形状(Σ)。事实上，D 中谓词不属于 sch(Σ) 的原子，不会以任何方式影响实例 追赶(D,Σ) 的大小。

体原子的谓词属于 $\Sigma(\text{shape}(D))$。现在我们继续证明，动态简化确实保留了追赶的有限性。

引理 4.3。 考虑一个数据库 D 和一组 $\Sigma \in \mathsf{L}$ 元组生成依赖。以下两者等价：

(1) 追赶(简单的(D)，简单的(Σ)) 是有限的。

(2) 追赶(简单的(D)，简单的$_D(\Sigma)$) 是有限的。

证明。根据定义，简单的$_D(\Sigma) \subseteq$ 简单的(Σ)，因此显然 (1)蕴含(2)。有趣的方向是(2)蕴含(1)。我们将证明所讨论蕴含的逆否命题，即如果实例追赶(简单的(D)，简单的(Σ))是无限的，那么实例追赶(简单的(D)，简单的$_D(\Sigma)$)也无限。为此，我们首先证明一个辅助技术引理：

引理 4.4。 考虑依赖图中的一条路径$(R_1, i_1) \ldots, (R_n, i_n)$，该图属于简单的$(\Sigma)$，且简单的$(D)$中存在一个形如$R_1(\bar{c})$ 的原子。则$\{R_1, \ldots, R_n\} \subseteq \Sigma($形状$(D))$。

证明。我们按路径的长度 n 进行归纳。

基础情形。 显然，$\text{shape}(D) \subseteq \Sigma(\text{shape}(D))$. 由于 $\text{shape}(D)$ 与 $\text{simple}(D)$ 中原子所用的谓词集合一致，并且根据假设， R_1 被 $\text{simple}(D)$ 中的一个原子使用，因此我们得到 $R_1 \in \Sigma(\text{shape}(D))$，证毕。

归纳步骤。 考虑 $\text{simple}(\Sigma)$ 的依赖图中的一条路径 $(R_1, i_1) \ldots, (R_n, i_n)$，其中 R_1 是 $\text{simple}(\Sigma_D$ R_1 R_{n-1} $\subseteq)$ 中某个原子所用的谓词。根据归纳假设，$\{,\ldots,\}$ (shape( ))。接下来只需证明 (shape( ))。假设 σ 是依赖图中边 $((R_{n-1}, i_{n-1}), (R_n, i_n))$ 的见证元组生成依赖，其中该依赖图属于 $\text{simple}(\Sigma)$。存在 $i \geq 0$，使得 $R_{n-1} \in \Gamma^i_\Sigma(\text{shape}(D))$. 由于 R_{n-1} 是 σ 的体原子的谓词， $R_n \in \Gamma^{i+1}_\Sigma(\text{shape}(D))$. 因此， $R_n \in \Sigma(\text{shape}(D))$.

□

现在我们可以证明所需的蕴含关系。根据假设，追赶(simple(D), simple(Σ))是无限的，因此由定理 3.3 可得，在 $\text{simple}(\Sigma)$ 的依赖图中存在一个带有特殊边的 $\text{simple}(D)$-支持环。令

$$(R_1, i_1), \ldots, (R_n, i_n),$$

with $(R_1, \ i_1) = (R_n, \ i_n)$ 为这样的环。由于该环是 $\text{simple}(D)$-支持的，因此环中存在一个原子 $P(\bar{c}) \in D$and 一个节点 $(R_j, \ i_j)$，使得 $P \rightsquigarrow_\Sigma R_j$。由于 Σ 的元组生成依赖具有非空前沿，在 $\text{simple}(\Sigma)$ 的依赖图中存在一条路径

$$(P_1, k_1), \ldots, (P_m, k_m)$$

with $P_1 = P$and $(P_m, \ k_m) = (R_j, \ i_j)$。总而言之，在 $\text{simple}(\Sigma)$ 的依赖图中，存在一条如下形式的路径

$$(P_1, k_1), \ldots, (P_{m-1}, k_{m-1}),$$
$$(R_j, i_j), \ldots, (R_{n-1}, i_{n-1}), (R_1, i_1), \ldots, (R_{j-1}, i_{j-1}), (R_j, i_j).$$

假设该路径由元组生成依赖 $\sigma_1, \ldots, \sigma_{n+m-2}$ 见证。根据引理 4.4，$\{P_1, \ldots, P_{m-1}, R_1, \ldots, R_{n-1}\} \subseteq \Sigma(\text{shape}(D))$。根据动态简化的定义（见定义 4.2），我们得到元组生成依赖 $\sigma_1, \ldots, \sigma_{n+m-2}$ 属于 $\text{simple}_D(\Sigma)$。因此，上述由 $\text{simple}(D)$ 支持且含有特殊边的环也出现在 $\text{simple}_D(\Sigma)$ 的依赖图中。于是，根据定理 3.3，$\text{chase}(\text{simple}(D), \text{simple}_D(\Sigma))$ 是无限的，结论得证。 □

算法 2： DynSimplification

输入： 一个数据库 D 和一组元组生成依赖 $\Sigma \in \mathsf{L}$

输出： D的简化： Σ

```
S ← FindShapes(D);
Σ_s ← ∅;
ΔS ← S;
当 ΔS ≠ ∅ 时，执行
    Σ_辅助 ← Applicable(ΔS, Σ);
    S_辅助 ← {R_标识符(t̄) ∈ 形状(Σ) | 存在一个元组生成依赖 σ ∈
      Σ_辅助 使得 R_标识符(t̄) 出现在头(σ)};
    Σ_s ← Σ_s ∪ Σ_辅助;
    ΔS ← S_辅助 \ S;
    S ← S ∪ ΔS;
返回 Σ_s;
```

动态简化的另一项关键性质有助于我们进一步提升线性元组生成依赖终止性算法的性能：$\text{simple}(D)$-弱无环性与 $\text{simple}_D(\Sigma)$ 的弱无环性一致。这是因为，按照构造，$\text{simple}_D(\Sigma)$ 中元组生成依赖所含的所有谓词都可以从 $\text{simple}(D)$ 中出现的某个谓词到达，因此，$\text{simple}_D(\Sigma)$ 的依赖图中每个含有特殊边的环都显然由 $\text{simple}(D)$ 支持。上述性质立即推出下面的引理 4.5：根据定理 3.6，检查 $\text{chase}(\text{simple}(D), \text{simple}_D(\Sigma))$ 是否有限，等价于检查 $\text{simple}(D)$-弱无环性是否适用于 $\text{simple}_D(\Sigma)$；而这归结为检查 $\text{simple}_D(\Sigma)$ 是否弱无环，无须显式检查是否由 $\text{simple}(D)$ 支持，从而避免了简化 D 这一开销巨大的任务。

引理 4.5. 考虑一个数据库 D 和一组元组生成依赖 $\Sigma \in \mathsf{L}$。以下条件等价：

(1) 追赶(简单的(D)，简单的$_D(\Sigma)$) 是有限的。*(2)* 简单的$_D(\Sigma)$ 是弱无环的。

动态简化算法。 动态简化确实能够利用给定的数据库，从通过静态简化一组线性元组生成依赖而得到的集合中筛除多余的简单线性元组生成依赖。然而，我们仍需提供一种具体算法，以对一组线性元组生成依赖进行动态简化，并使其适用于高效实现。为此，我们接下来介绍算法 DynSimplification，如算法 2 所示。该算法以数据库 D 和一组线性元组生成依赖 Σ 为输入，构造 Σ的 D-简化结果。算法首先查找 D中各原子的形状，即计算集合 $\text{shape}(D)$（第 1 行）。接着，它初始化简化后的元组生成依赖集合 Σ_s（第 2 行）以及新形状集合 ΔS（第 3 行）。随后，算法迭代生成简化后的元组生成依赖，并收集添加到 ΔS中的新形状，持续执行直至达到不动点，即 $\Delta S = \emptyset$（第 4 行）。具体而言，在每次迭代中，算法计算非多余的简化后元组生成依赖，也就是说，这些依赖可在构造 $\text{chase}(\text{simple}(D), \text{simple}(\Sigma))$时应用，并被添加到 Σ_s 中（第 5 行和第 7 行）。这一过程通过 Applicable 过程完成，该过程以.....为输入

算法 3： IsChaseFinite[L]

输入： 一个数据库 D 和一组元组生成依赖 $\Sigma \in \mathrm{L}$

输出： 若追赶(D, Σ)有限则为 true，否则为 false

1 $\Sigma_s \leftarrow$ DynSimplification(D, Σ);
2 $G \leftarrow$ BuildDepGraph(Σ_s);
3 **若** FindSpecialSCC(G) $\neq \emptyset$ **则返回** `false`;
4 **返回**`true`

形状集合 $\hat{}S$ 和线性元组生成依赖集合 $\hat{}\Sigma$，并返回集合

$$\{\text{simple}(R(f(\bar{x}))) \rightarrow \exists \bar{z}\, \text{simple}(\psi(f(\bar{y}), \bar{z})) \mid R(\bar{x}) \rightarrow \exists \bar{z}\, \psi(\bar{y}, \bar{z}) \in \hat{\Sigma} \text{ and } f \text{ is the } h\text{-specialization of } \bar{x} \text{ for some homomorphism } h \text{ from } \{R(\bar{x})\} \text{ to } DB[\hat{S}]\}.$$

本质上，Applicable 过程会计算 simple($\hat{}\Sigma$) 中的元组生成依赖集，其中体的谓词属于 $\hat{}S$。该算法还会收集新生成的形状，即 Applicable(ΔS, Σ) 中元组生成依赖的头中出现的谓词，并将它们添加到 ΔS（第 6 行和第 8 行）。请注意，在每次迭代中，算法都会将元组生成依赖应用于 ΔS，而非 S；只有第一次迭代例外，此时 $S = \Delta S$。这样做可行，是因为第一次迭代之后，S 上不会再有新的适用元组生成依赖：元组生成依赖是线性的（只有一个体原子），而所有适用于 S 的元组生成依赖都已在第一次迭代中应用。FindShapes 和 Applicable 的实现细节分别在第 5.1 节和第 5.2 节讨论。DynSimplification 的正确性由构造保证：

引理 4.6。 考虑一个数据库 D 和一个元组生成依赖集 $\Sigma \in \mathrm{L}$。有 DynSimplification(D, Σ) = 简单的$_D(\Sigma)$。

终止性算法。 有了 DynSimplification，现在可以轻松设计出算法 IsChaseFinite[L]，（见算法 3），用于检查线性元组生成依赖情况下追赶的有限性。DynSimplification 的正确性、定理 3.6、引理 4.3 和引理 4.5 保证了算法 IsChaseFinite[L], 的正确性，由此可得以下引理：

引理 4.7。 给定一个数据库 D 和一组 $\Sigma \in$ 线性 元组生成依赖，IsChaseFinite[线性](D, Σ) = 为真 当且仅当 追赶(D, Σ) 是有限的。

5 实现细节

我们接下来讨论第 4 节中介绍的用于检查追赶实例有限性的算法的实现细节。具体而言，我们将讨论有助于提升算法性能、但第 4 节的描述中未提及的实现选择。在第 5.1、5.2 和 5.3 节中，我们分别讨论算法 1 中使用的过程 BuildDepGraph、FindSpecialSCC 和 Supports。算法 3 中使用的 DynSimplification 的细节将在第 5.4 节中讨论。

5.1 构建依赖关系图

过程 BuildDepGraph 接受一个元组生成依赖集 Σ 作为输入，并以邻接表的形式返回 Σ 的依赖图。回想一下，有向图的邻接表是一个列表的列表。每个列表对应图中的一个节点 v，其中的成员表示此类列表中的成员表示 v 的出边。为了实现这种邻接表，我们将依赖图存储为一个由节点对象组成的列表。每个节点对象表示图中的一个节点，并包含一个边对象 列表，用于表示图的边。对于每个边对象，我们还会存储一个二进制值，用于指明相应的边是否为特殊边。虽然存储依赖图只需使用单向链表，但出于性能考虑，我们将依赖图的邻接表实现为双向链表。通过采用双向链表，每个节点对象除了包含边对象列表外，还包含一个反向边对象列表，用于表示方向相反的边。这些反向边对象使得我们能够沿着与边相反的方向遍历图，在检查特殊强连通分量中位置的支持情况时很有帮助，正如我们在第 5.3 节中所述。现在，给定一个简单线性元组生成依赖集 Σ，BuildDepGraph 会遍历所有元组生成依赖，并根据这些依赖的规定，为新访问的位置创建新元素并将其连接起来，从而构建 Σ 的依赖图。为了加快依赖图的构建过程，该过程还使用一种索引结构，将谓词位置映射到邻接表中对应的元素。解析新的元组生成依赖时，该索引可快速访问元素（节点），以便添加新的连接（边）。借助索引结构，该过程能够以与输入的元组生成依赖集大小呈线性关系的时间构建依赖图。

5.2 查找特殊强连通分量

为了实现 FindSpecialSCC，我们改进了用于在有向图中查找强连通分量的知名 *Tarjan* 算法 [23],，其基本原理将在下文简要回顾。文献中还介绍了其他查找强连通分量的算法（例如，参见 [14, 22]），其中一些算法比 Tarjan 算法更简单，例如 Kosaraju 算法 [22]。尽管如此，我们仍以 Tarjan 算法为基础，因为它在实践中效率更高。

Tarjan 算法。 给定一个有向图，Tarjan 算法会构造一个生成森林，即由一组树组成的集合，其中包含图中的所有节点，且每棵树都对应一个强连通分量。为了找到这些树，算法会从图中的任意节点开始进行深度优先搜索，并通过遍历每棵树中的节点，再转向下一棵树来构造这些树。在搜索树中的节点时，算法会遍历两类边：(i) 指向新节点的边；(ii) 指向已访问节点的边。第一类边称为树边，因为它们会将新节点添加到树中。第二类边又分为三个子类：(ii.a) 从树中的祖先指向其后代之一的边，算法会忽略这类边；(ii.b) 从后代指向其祖先的边，称为回边；(ii.c) 在同一棵树中从一棵子树指向另一棵子树的边，称为交叉边。Tarjan 算法按照节点在搜索过程中被访问的顺序为节点编号（即编号较小的节点先被访问），并将节点压入栈中。这个栈称为强连通分量栈，与用于实现深度优先搜索的栈不同。在深度优先搜索中，当一个已访问节点的所有子节点都访问完毕后，回溯时该节点会从搜索栈中移除。然而，只要该节点存在一条通往栈中其下方节点的路径，它就会继续保留在强连通分量栈中。这使得强连通分量栈能够跟踪当前树中的节点。

算法利用已分配的编号，在访问完一个节点的所有子节点后判断该节点是否为根节点。判断方法是检查访问后代节点时所经过的边是否为回边或交叉边。找到根节点后，算法会从强连通分量栈的栈顶弹出节点，并将它们加入一个强连通分量，直到弹出当前树的根节点。算法会持续执行，直至图中的所有节点都被访问，并返回得到的各棵树（即图的强连通分量）。

FindSpecialSCC 过程。这是用于在依赖图中查找特殊强连通分量的 Tarjan 算法的简单扩展。之所以需要这样的扩展，是因为我们需要一种机制来检查 Tarjan 算法得到的强连通分量是否确实是特殊强连通分量。具体做法是：每当算法遍历一条特殊边时，就将一个哑元标记压入栈中（该栈用于存储当前分量中已访问的节点）。请注意，即使特殊边被忽略，算法也会压入这个哑元标记，正如我们在 Tarjan 算法中所解释的那样。找到当前强连通分量的根节点并弹出节点以创建该强连通分量后，如果弹出的节点之间存在哑元标记，算法就会将该强连通分量标记为特殊强连通分量。算法只存储并返回特殊强连通分量。

5.3 位置支持检查

Supports 过程包含两个步骤：(1) 查询数据库以查找外延谓词的位置；(2) 从特殊强连通分量中的位置开始，按逆序遍历依赖图，直到到达第一步中计算出的位置。步骤 (1) 通过一条 SQL 查询实现，该查询返回非空关系的列表，然后我们用它来创建外延谓词的位置集合，记为 P_D。该 SQL 查询利用存储数据库的数据库管理系统的目录实现，因此能够更快地查找关系名称集合，而无需访问实际数据。对于步骤 (2)，我们从给定的位置集合 P（即特殊强连通分量中的位置集合）开始，并使用依赖图邻接表中的反向链接按逆序遍历图，如第 5.1 节所述。如果第二步中的图遍历从外延谓词到达 P_D 中的一个节点（即一个位置），该过程就返回 true；否则返回 false。

5.4 动态简化

DynSmplification 过程以数据库 D 和线性元组生成依赖 Σ 集合作为输入，并返回简单$_D(\Sigma)$，即简单的线性元组生成依赖集合。该过程是一个迭代过程，使用两个子过程：FindShapes 用于计算 D中原子的形状 S集合，Applicable 则利用 S的形状计算简化后的元组生成依赖。我们将讨论这两个子过程的实现细节。

FindShapes 过程 FindShapes。 该过程有两种实现方式，即内存实现和数据库内实现，下面将分别讨论。在第 8 节和第 9 节中，我们通过实验比较 FindShapes 的两种实现方式的性能，并讨论哪种方式在何种情况下表现更优。

内存实现

对于内存实现，我们对每个关系 R 执行一次 SQL 查询，从 D 中加载 R 的所有元组到主存中。然后，我们通过以下方式构建每个关系 R 中原子的形状集合：遍历其元组 $\bar{c}$，并生成每个原子 $R(\bar{c})$ 的形状。对于无法全部加载到主存中的关系，我们将其拆分为更小的关系并分别处理。

数据库内实现

数据库内实现不会加载关系，而是运行 SQL 查询来查找每个关系中原子的形状。具体而言，我们将每种可能的形状转换为布尔查询；如果该形状存在于数据库中，查询就会返回 true。谓词 R 的形状对应的查询一般形式如下：

```
SELECT CASE WHENEXISTS(SELECT* FROMRWHERE相
等_条件AND 不等_条件)THEN 1 ELSE 0END
```

其中，相等条件和不等条件是根据给定形状进行的一致性检查。例如，形状 $R_{[1,1,2]}$ 转换为以下 SQL 查询 Q:

```
SELECT CASE WHEN EXISTS
      (SELECT * FROM R WHERE a1=a2 AND a2!=a3)
          THEN 1 ELSE 0 END
```

其中，我们假设谓词 R 包含属性 a1、a2 和 a3。当然，对于元数较高的谓词，每种形状都运行一个 SQL 查询会产生许多查询。不过，根据数据库 D的不同，其中许多查询可能并无必要，因为它们找不到任何形状。为了避免运行其中一些查询，我们借鉴 Apriori 算法的思路来发现关联规则 [2]。我们首先运行查询，检查更一般形状（例如，$R(1,1,2)$）是否存在，然后再检查更具体的形状（例如，$R(1,1,1)$）。对于每种形状，我们运行一对查询。第一个查询是上文所述通用查询的宽松版本，不包含不等条件。例如，对于 $R(1,1,2)$，第一个查询 Q' 是上文 Q所述的、去掉带下划线条件后的查询。只有当 Q' 的计算结果为 true 时，我们才继续运行 Q。此外，如果 Q' 的计算结果为 false，我们就不会运行针对更具体形状（例如，$R(1,1,1)$）的查询。这样，只需对更一般的形状运行单个查询，就能避免执行许多针对具体形状的查询。

过程Applicable。 回顾一下，Applicable 的输入是形状集合 S 和线性元组生成依赖集合 Σ，输出是 simple(Σ) 中体的谓词属于 S 的元组生成依赖集合。如第 4 节所述，该过程通过遍历所有形如 $R(\bar{x}) \rightarrow \exists \bar{z}\, \psi(\bar{y}, \bar{z})$ 的元组生成依赖 $\sigma \in \Sigma$ 以及从 $\{R(\bar{x})\}$ 到 数据库$[S]$ 的同态 h来实现，并收集由 $\bar{x}$ 的 h-特化所诱导的 σ 的简化结果。对于大量形状和大量元组生成依赖应用上述迭代过程，代价可能非常高。因此，为了提高该过程的性能，实现采用了一种索引结构，以便快速访问元组生成依赖。该索引结构将每个谓词 $R \in \text{sch}(\Sigma)$ 映射到 Σ 中体原子使用 R 的元组生成依赖集合。这样，每次迭代中，该过程都可以遍历当前形状，并快速访问索引中相关的元组生成依赖。不过，检查相关元组生成依赖是否适用仍然代价高昂。这项任务需要检查从当前形状与该元组生成依赖得到的所有简化元组生成依赖中体原子的形状。为此，我们生成并存储每个元组生成依赖的体原子的形状。此外，

我们还存储一个字符串数组，其中包含模式最大元数以内的所有可能元组标识符，以便快速查找简化元组生成依赖中体原子的形状。

6 实验基础设施

我们的目标是对第 5 节中介绍的终止性算法的行为进行实验评估，以明确它们能否应用于实际场景；如果不能，则揭示其局限性。为此，我们将使用合成数据和元组生成依赖集开展大量实验。因此，我们需要一种方法来生成适用于此类实验评估的数据库和元组生成依赖集。接下来，我们将讨论用于生成数据（第6节.1）和元组生成依赖集（第6节.2）的工具。需要说明的是，在本文余下部分中，每当我们说从某个元素空间中随机选取一个元素时，实际指的是均匀随机地选取该元素。

6.1 数据生成器

有一些可免费使用的数据生成器，例如 TPC-H[2] 和 DataFiller[3]。然而，现有工具都不适合我们的用途。为了有效评估上文介绍的动态简化过程（它是线性元组生成依赖终止性算法的关键分量），我们需要确保生成的数据库包含多种形状。这正是现有数据生成器的局限所在，因为它们不允许我们控制所生成原子的形状。因此，我们不得不自行实现一个数据生成器，以克服上述局限。

我们的数据生成器具有一些调优参数，可用于确定生成的数据库 D的关键属性： D中的谓词数量、这些谓词的最小和最大元数、数据库论域的大小（即 dom(D) 中的值的数量），以及 D中每个关系的元组数量。具体而言，生成器以一个包含调优参数整数值的元组（preds、min、max、dsize、rsize）为输入，并构造一个数据库 D，使得 $\mathbf{S} = \{R \mid R(\bar{c}) \in D\}$， $|\mathbf{S}| = \text{preds}$，S 中谓词的元数介于 min 和 max 之间， $|\text{dom}(D)| = \text{dsize}$，并且对于每个 $R \in \mathbf{S}$， $|\{\bar{c} \mid R(\bar{c}) \in D\}| = \text{rsize}$。为此，它首先生成一个由 preds 个不同谓词组成的集合 S，并从范围 [min, max] 中为每个谓词随机选择一个元数。然后，它通过向 S 中的每个谓词添加 rsize 个元组来生成数据库，这些元组由 dsize 个不同的常量值构成。为了确保生成的数据库包含不同的形状，每个元组都通过随机选择一种形状，并从大小为 dsize 的数据库论域中随机选取值来填充各个位置而生成，且不重复选取；也就是说，一种形状决定了同一个值在一个元组中重复出现的次数。

6.2 TGD生成器

至于数据生成器，现有的 TGD生成器（例如，参见 [5**?**]）并不适合我们的用途，因为它们无法控制生成的元组生成依赖的体中原子的形状，而这对于生成适用于我们实验的元组生成依赖集合至关重要。因此，我们不得不自行实现一个支持这一关键功能的 TGD生成器。

[2]https://www.tpc.org/tpch/ [3] https://github.com/memsql/datafiller

我们的 TGD生成器具有一些调优参数，可用于确定所生成的元组生成依赖集合 Σ 的关键属性：模式的大小（即 sch(Σ) 的大小）、sch(Σ) 中谓词的最小和最大元数、 Σ中的元组生成依赖数量，以及其所属的类别 Σ （即 Σ 是简单线性的元组生成依赖集合还是线性的元组生成依赖集合）。具体而言，TGD生成器以谓词集合 S 和一组调优参数值（ssize、min、max、tsize、tclass）作为输入，并构造一个元组生成依赖集合 Σ，使得 $\text{sch}(\Sigma) \subseteq \mathbf{S}$， $|\text{sch}(\Sigma)| = \text{ssize}$，sch($\Sigma$) 中的谓词元数介于 min 和 max 之间， $|\Sigma| = \text{tsize}$，且 Σ 属于 tclass 类别。为此，它首先从 S 中选取一个子集 $\mathbf{S}'$，使得 $|\mathbf{S}'| = \text{ssize}$，且其中的谓词元数介于 min 和 max 之间，然后使用 $\mathbf{S}'$ 中的所有谓词生成所需的简单线性或线性元组生成依赖集合；具体生成过程如下所述。

需要强调的是，在我们的实验中，我们只考虑单头元组生成依赖，即只有一个头原子的元组生成依赖，尽管我们的终止算法也适用于多头元组生成依赖，即头部包含多个原子的元组生成依赖。这是因为，与元组生成依赖的数量相比，出现在元组生成依赖头部中的原子数量通常可以忽略不计，而且元组生成依赖的头部包含多个原子并不会影响数据库中出现的形状数量。正如我们将在第7节和第8节中看到的，元组生成依赖的数量和数据库形状数量是影响算法运行时间的主要参数，因此，即便只考虑单头元组生成依赖，我们的实验评估仍能得出确凿的结果。因此，为了清晰起见，下面介绍的TGD生成器专用于生成单头元组生成依赖。

简单线性元组生成依赖

要生成简单线性 TGD，生成器会从$\mathbf{S}$，中随机选择两个谓词，分别用于构造体原子和头原子。随机选择时允许重复，因此同一个谓词可以同时出现在 TGD 的体和头中。由于我们要生成简单线性 TGD，因此用不同的变量填充体原子中的各个位置。对于头原子，则用存在量化变量，或已在体原子中使用的全称量化变量来填充各个位置。具体而言，对于头原子的每个位置 π，生成器以 10% 的概率将其分类为存在位置。在这种情况下， π会填入一个未在体原子中出现过的新变量；否则，则填入一个从体原子中随机选取的变量。

线性元组生成依赖

生成线性元组生成依赖的方式与生成简单线性元组生成依赖相同，但有一个关键区别：随机选择体原子和头原子的谓词后，生成器会随机选择体原子的形状，然后按照与生成简单线性元组生成依赖相似的步骤，填充体原子和头原子中的各个位置。请注意，体变量的选择由所选形状决定，这使得生成的线性元组生成依赖的体原子中可以重复出现变量。

现在，我们可以开始进行实验评估了。请注意，实验使用的服务器配备英特尔酷睿 i5 3.00GHz 中央处理器和 16GB 随机存取存储器；实验中的所有数据库均存储在 PostgreSQL 11.5 实例中，相关终止算法则使用 Java SE 11 实现。

7 简单线性元组生成依赖的评估

我们首先对算法 1 中所示的 IsChaseFinite[SL]，进行实验评估。为了进行更细致的分析，我们将把它的端到端运行时间（记为 t-总计）分解为以下三个时间参数：

- t-解析：从输入文件解析元组生成依赖所需的时间，
- t-图：构建输入的元组生成依赖集的依赖图 G 所需的时间，以及
- t-计算：在图 G中查找特殊强连通分量所需的时间。

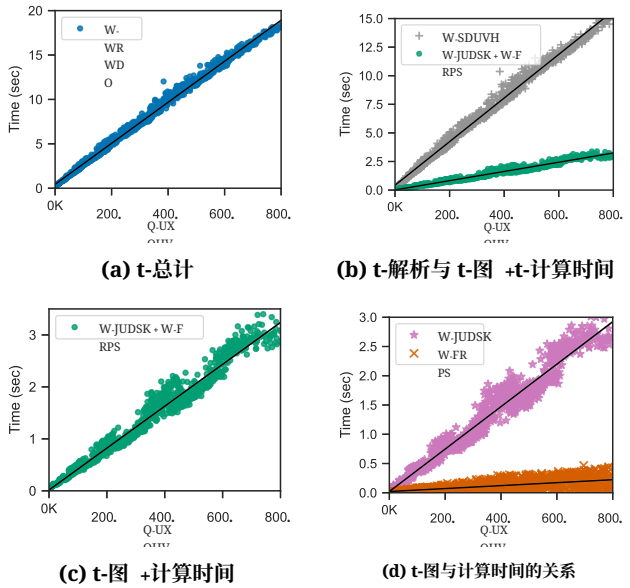
在本节余下部分中，我们将说明如何生成用于实验评估的简单线性元组生成依赖集，随后展示实验结果并讨论主要结论。不过，首先我们先作几点澄清。

备注 1. 在我们的分析中，我们忽略 Supports 过程所需的时间。如第 5.3 节所述，该过程包含两个步骤：(1) 找出输入数据库中出现的谓词；(2) 从特殊强连通分量中的位置出发，按逆序遍历依赖图，以到达第一步中计算出的关系的位置。步骤 (1) 通过对存储输入数据库的数据库管理系统的目录运行快速查询来完成；由于它不会影响算法的其余部分，因此可以放心忽略。至于步骤 (2)，与查找特殊强连通分量所需的时间相比，遍历依赖图所需的时间可以忽略不计，而前者本身只需毫秒级时间。总而言之，Supports 过程所需的时间与算法其余部分相比微不足道，因此我们可以直接忽略它，而不影响对 IsChaseFinite[SL] 的分析。因此，在实验中，我们假设简单线性元组生成依赖集使用的所有谓词都出现在数据库中，因而特殊强连通分量中的所有位置都自然得到支持。这也简化了实验，因为我们可以使用一个非常简单的数据库来进行实验，该数据库可由简单线性元组生成依赖集 Σ 推导得出，记为 D_Σ，无需使用前文讨论的数据生成器。事实上，对于每个谓词 $R \in \text{sch}(\Sigma)$，D_Σ 都有一个原子 $R(c_1, \ldots\ldots, c_n)$，其中 $c_1, \ldots\ldots, c_n$ 是互不相同的常量。

备注 2. 解析时间（t-解析）显然取决于解析器的性能，对于评估 IsChaseFinite[SL] 的性能并非十分关键。不过，为了准确统计算法的端到端运行时间，我们将其纳入分析，并将其与另外两个时间参数进行比较。

7.1 生成简单的线性元组生成依赖

现在讨论实验中使用的简单线性元组生成依赖集合是如何生成的。为了系统地生成具有代表性的元组生成依赖集合类别，同时不偏向两个关键参数中的任何一个，即底层模式的大小和元组生成依赖的数量，我们考虑了三个谓词配置，每个配置由提及 [5,200], [200,400], 和 [400,600] 个元数介于 1 至 5 的谓词的元组生成依赖集合构成；此外，我们还考虑了三个元组生成依赖配置，每个配置由包含 [1,333K], [333K、666K], 和 [666K、1M] 个元组生成依赖的集合构成。请注意，我们将谓词的元数设定在 1 至 5 之间，这与我们在现实场景中的观察一致，因为元数通常较小。将这些谓词配置与元组生成依赖配置组合起来，可得到九种组合配置，每种配置由句法属性相似的元组生成依赖集合构成。对于



(a) t-总计　(b) t-解析与 t-图 +t-计算时间

(c) t-图 +计算时间　(d) t-图与计算时间的关系

图1：IsChaseFinite IsChaseFinite[简单线性类]**的运行时间。**

例如，由谓词配置[200,400]和元组生成依赖配置 [333K、666K] 得到的组合配置，包含满足以下条件的元组生成依赖集合：Σ 使得 $200 \le \text{sch}(\Sigma) \le 400$，$\text{sch}(\Sigma)$ 中的每个谓词的元数都介于 1 至 5 之间，并且 $333\text{K} \le |\Sigma| \le 666\text{K}$。实验中，我们为九种组合配置各生成了 100 个元组生成依赖集合，共计 900 个简单线性元组生成依赖集合。具体步骤如下。我们首先生成 1000 个谓词来构建底层模式S，并从范围 [1,5] 中随机选取它们的元数。然后，对于由谓词配置 $[x, y]$ 和元组生成依赖配置 $[z, w]$导出的组合配置，我们以模式S 及调优参数值元组（$ssize$，1，5，$tsize$，SL）为输入，反复运行 TGD生成器，从而生成 100 个简单线性元组生成依赖集合。其中$ssize$和$tsize$分别从值范围 $[x, y]$和 $[z, w]$中随机选取。

7.2 实验评估

对上述组合配置中的 900 个元组生成依赖集，逐一运行了算法 IsChaseFinite[简单线性类]。请回想一下，输入数据库由输入的元组生成依赖集诱导而来，即对于元组生成依赖集 Σ，数据库为 D_Σ。图1中的散点图展示了 IsChaseFinite[简单线性类](D_Σ, Σ) 对组合配置中的每个元组生成依赖集 Σ 的运行时间。具体而言，图中的每个点对应 900 个元组生成依赖集之一。图1a展示了不同大小（n-规则）的元组生成依赖集的总运行时间（t-总计）。图1b将 t-总计细分为解析元组生成依赖集所需的时间（t-解析），以及构建其依赖图并查找特殊强连通分量所需的时间（t-图 + 计算时间）。图1c放大显示了 t-图 +计算时间，它们分别展示在图1d中。

从上述散点图可以明显看出，随着 n-规则数量增加，时间参数 t-解析和 t-图呈线性增长，而 t-计算时间增长得非常缓慢。值得一提的是，我们没有观察到时间参数

t-解析和 t-图与底层模式中的谓词数量之间存在这种线性关系（事实上，我们没有观察到任何相关性）。t-解析与 n-规则之间的线性关系，是因为解析每个元组生成依赖都只需常量时间：谓词的元数处于有限范围 [1,5], 内，并且每个元组生成依赖的体和头中都只有一个原子。请注意，允许多头也不会改变这一点，因为正如第6节所述，头部原子的数量与元组生成依赖的数量相比微不足道。t-图呈线性增长（如图 1d 所示），是因为该算法会遍历元组生成依赖，处理每个元组生成依赖并通过添加新节点和边来更新图，所花费的时间都是常量。同样，由于谓词的元数处于 [1,5], 内，并且每个元组生成依赖的体和头中都只有一个原子，由某个元组生成依赖在依赖图中添加的节点和边数量处于一个较小的固定范围内，因此更新图中与每个元组生成依赖相关的部分所需的时间是常量。t-计算时间增长得非常缓慢，是因为查找特殊强连通分量完全取决于依赖图，而依赖图通常远小于 TGD 集；与此同时，Tarjan 算法效率很高，其运行时间与底层图的规模呈线性关系。

7.3 主要结论

简单线性元组生成依赖的实验结果表明，影响 IsChaseFinite[SL] 运行时间的主要参数是元组生成依赖的数量（n-规则）。我们还观察到，即使面对极大规模的元组生成依赖集合，该算法的运行速度也非常快。事实上，端到端运行时间的大部分都花在解析（t-解析）和构建依赖图（t-图）上；与 t-解析和 t-图相比，查找特殊强连通分量所需的时间（t-comp）微不足道。具体而言，t-解析耗时远超 t-图，实际上占据了算法端到端总运行时间的大部分。这体现了 IsChaseFinite[SL] 的有效性：对于大规模元组生成依赖集合，实际检查追赶实例是否有限所需的时间，远少于从输入文件读取并解析元组生成依赖所需的时间。

8 线性元组生成依赖的评估

现在，我们来评估算法 3 中描绘的 IsChaseFinite[L], 。与 IsChaseFinite[SL], 不同，后者的输入数据库并未发挥任何关键作用；而现在，我们有一个严重依赖数据库的分量，即计算数据库形状的过程，它属于动态简化。换言之，IsChaseFinite[L], 包含依赖于数据库的组件，即查找数据库形状；以及数据库无关组件，即利用数据库形状简化给定的线性元组生成依赖集、构建简化后的 TGD 集的依赖图，并找出该图中的特殊强连通分量。我们认为，这两个分量——下文分别称为数据库相关和数据库无关分量——应当分别评估，因为它们的运行时间受不同参数影响。显然，数据库相关分量只受数据库影响，而 TGD 集不起作用。另一方面，尽管数据库无关分量受 TGD 集影响这一点很清楚，但它是否受输入数据库影响则不那么容易判断，因为它处理的是动态简化后的 TGD 集。有意思的是，我们将在下文通过实验确认，事实确实如此。因此，

为了对算法 IsChaseFinite[L], 进行更精细的分析，我们将考虑以下四个时间参数：

- t-shapes：查找数据库形状所需的时间，
- t-parse：从输入文件解析元组生成依赖所需的时间，
- t-graph：构建简化后输入的元组生成依赖集的依赖图所需的时间 G（包括使用数据库形状进行简化所需的时间），以及
- t-comp：在图中查找特殊强连通分量所需的时间 G。

显然，t-shapes 指的是数据库相关分量的运行时间，而 t-parse + t-graph + t-comp（我们将其记为 t-total）指的是数据库无关组件的端到端运行时间。本节余下部分将说明我们如何生成实验评估中使用的数据库和线性元组生成依赖集，确认数据库无关组件不受输入数据库的影响，随后给出 IsChaseFinite[L] 两个组件的实验结果，并讨论其中的主要结论。请注意，正如第 7 节的备注 2 所述，尽管解析时间对于评估数据库无关组件的性能并不重要，但为了准确计算端到端运行时间，我们仍将其纳入考量，并将其与 t-graph 和 t-comp 进行比较。

8.1 生成数据库和线性元组生成依赖

我们的目标是生成一系列形如（D, Σ）的配对，其中D 是数据库，Σ 是线性元组生成依赖集合。这些配对将作为实验评估期间 IsChaseFinite[L] 的输入。为此，我们首先使用数据生成器构建了一个非常庞大的数据库，并将其命名为 $D\star$。具体而言，我们向数据生成器提供了输入（1000，1，5，500K，500K），由此得到 $D\star$。它包含 1000 个元数在 1 到 5 之间的谓词，每个谓词有 500K 个元组，因此整个数据库共有 500M 个元组。我们还在 $D\star$ 上设计了视图，以便按需定义每个谓词分别包含 1K、50K、100K、250K 和 500K 个元组的虚拟数据库；相应数据库的元组总数分别为 1M、50M、100M、250M 和 500M。这些视图实际上通过简单的 SQL 查询实现，查询仅保留每个谓词的前 1K、50K、100K、250K 和 500K 个元组。请注意，$D\star$ 中的元组按字典序排序，这意味着 $D\star$ 的一个关系中的不同形状分布均匀。这进而确保了通过视图定义的虚拟数据库包含多种形状，这对我们的研究至关重要。准备好数据库 $D\star$ 及其数据库视图后，我们按照下文所述生成了所需的一系列配对。

对于第 7 节中用于生成简单线性 TGD 的九种组合配置，我们生成了 5 组线性元组生成依赖，共计 45 组。具体而言，对于由谓词配置 $[x, y]$ 和 TGD 配置 $[z, w]$ 诱导的组合配置，我们通过反复运行 TGD 生成器生成了 5 组线性元组生成依赖，输入的模式为 $\{R \mid R(\bar{c}) \in D\star\}$，即 $D\star$ 中出现的 1000 个谓词，以及调优参数的取值元组（$ssize$，1，5，$tsize$，L），其中 $ssize$和 $tsize$ 分别从取值范围 $[x, y]$ 和 $[z, w]$ 中随机选取。令 $\Sigma\star$ 为收集所生成的 45 组线性元组生成依赖的类别。然后，对于每一组线性元组生成依赖 $\Sigma \in \Sigma\star$，我们利用上文讨论的数据库视图，获得了五个大小各异的虚拟数据库（每个谓词分别包含 1K、50K、100K、250K 和 500K 个元组

），分别记为 D^1_Σ、D^{50}_Σ、D^{100}_Σ、D^{250}_Σ 和 D^{500}_Σ，从而得到五对。综上，我们生成了类别

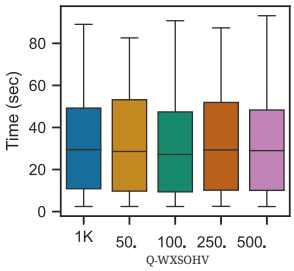
$$\left\{(D^s_\Sigma, \Sigma) \mid \Sigma \in \Sigma^\star \text{ and } s \in \{1, 50, 100, 250, 500\}\right\}$$

由225对组成，这些数据将作为 IsChaseFinite[L] 的输入。

8.2 实验评估

在深入评估算法 IsChaseFinite[L]，的两个分量之前，我们先确认数据库无关组件确实不受输入数据库的影响，这也证明了我们分别评估这两个分量的决定是合理的，因为它们的运行时间受不同参数的影响。

分开考察这两个分量。 下图展示了对所有生成的数据对取平均后，构建使用 D^s_Σ 的形状来查找特殊强连通分量的 Σ 动态简化版本的依赖关系图所需的时间；每对数据由一个大小为 $s \in \{1, 50, 100, 250, 500\}$ 的数据库 D^s_Σ 和一组 Σ 线性元组生成依赖组成：



有趣的是，这证实了数据库大小不会影响构建依赖图和查找特殊强连通分量所需的时间；因此，正如上文所述，它也不会影响数据库无关组件的端到端运行时间。这可以用以下事实来解释：随着数据库大小增加，数据库中的形状数量增长非常缓慢；图2对此进行了说明。具体而言，图2中的条形图展示了大小为 s 的所有数据库 D^s_Σ 中形状数量的平均值，每个图对应一种谓词配置 $[x, y]$，即 Σ 属于谓词配置 $[x, y]$。显然，随着数据库大小增加，形状数量也会增加；这在一定程度上是意料之中的，因为数据库大小增加会带来更多元组，而这些元组可能会产生更多去重后的形状。然而，值得注意的是，这种增长非常缓慢，这应归因于许多元组可能会产生相同的形状。此外，一个新形状只会产生少量简化后的元组生成依赖，而且不会显著影响构建和处理依赖图所需的时间。

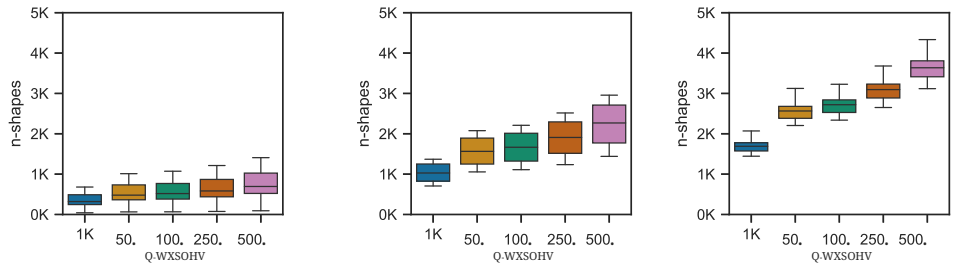
最后需要指出的是，通过比较这三幅条形图可以清楚地看出，谓词数量（反映在谓词配置中）会影响形状数量。这并不令人意外，因为谓词越多，形状也就越多。这意味着，底层模式中的谓词数量是影响形状数量的一个参数，这也解释了为什么我们在上文的分析中必须分别考察这三种谓词配置。

对数据库依赖分量的评估。我们针对每个数据库 D^s_Σ 运行过程 FindShapes，其中 $\Sigma \in \Sigma_\star$，且 $s \in \{1, 50, 100, 250, 500\}$；总计执行 225 次。请注意，过程 FindShapes 有两种实现方式，分别为内存中实现和数据库内实现（见第 5.4 节）。图3和图4中的条形图分别展示了针对内存中实现和数据库内实现，在所有特定规模的数据库 D^s_Σ s 上查找形状的平均运行时间；每个图对应一种特定的谓词配置。请注意，对于这两种实现，我们观察到了相似的趋势：数据库内实现优于内存中实现。

从图4中的条形图可以明显看出，随着数据库大小增加，查找形状所需的时间也会增加。这并不令人意外，因为如上所述，数据库大小增加时，形状数量也会增加。不过请注意，查找形状所需的时间增长速度远快于形状数量本身，这应归因于查找形状时实际上需要扫描整个数据库。最后需要指出的是，通过比较这三幅条形图可以看出，谓词数量（反映在底层谓词配置中）也会影响查找形状所需的时间，这解释了为什么我们必须分别分析每种谓词配置。

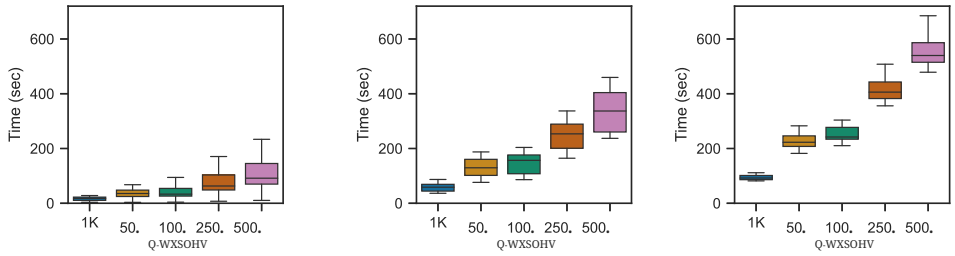
数据库无关组件的评估。 图5中的散点图展示了算法 IsChaseFinite[L] 在以输入 (D^s_Σ, Σ) 执行时，数据库无关组件的运行时间；每个集合 $\Sigma \in \Sigma_\star$ 均属于谓词配置 [400,600]，且 $s \in \{1, 50, 100, 250, 500\}$。具体而言，图中的每个点都对应一对 (D^s_Σ, Σ)。需要强调的是，与简单线性元组生成依赖对应的图1不同，我们聚焦于特定的谓词配置，因为否则无法观察到运行时间随元组生成依赖数量变化的趋势。换言之，这些图中表面上呈现的线性趋势，仅适用于来自同一谓词配置的元组生成依赖集合。这是因为底层模式中的谓词数量会影响形状数量，进而影响动态简化过程和依赖图的规模，因此也会影响时间参数 t-graph 和 t-comp。这解释了为什么我们必须分别分析每种谓词配置。为便于阅读，谓词配置 [5,200] 和 [200,400] 对应的类似图表在附录中给出并讨论。图5a展示了不同规模（n-规则）的元组生成依赖集合的总运行时间（t-total）。图5b将 t-total 分解为解析元组生成依赖所需的时间（t-parse），以及构建其依赖图并寻找特殊强连通分量所需的时间（t-graph + t-comp）；图5c则分别展示 t-graph 和 t-comp。图5d放大显示了 t-comp。

从上述散点图可以明显看出，随着 n-规则 数量增加，时间参数 t-解析 和 t-图 呈线性增长，而 t-计算时间 的增长则非常缓慢。这与我们在图 1 中对简单线性元组生成依赖的观察基本一致，关键区别在于：与构建依赖图并查找特殊强连通分量所需的时间（t-图 + t-计算时间）相比，现在解析元组生成依赖（t-解析）所需的时间要少得多。不过，不应忘记，对于线性元组生成依赖，我们需要聚焦于单一谓词配置，才能观察到这些线性趋势；否则，如果同时考虑所有谓词配置，就看不到任何趋势。还要注意，与简单线性元组生成依赖的情况相比，绝对运行时间有所增加。



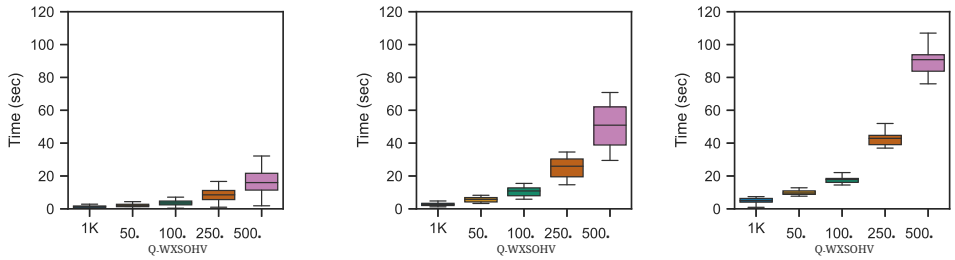
(a) [5,200] **(b) [200,400]** **(c) [400,600]**

图2：形状数量。



(a) [5,200] **(b) [200,400]** **(c) [400,600]**

图3： FindShapes **（内存实现的运行时间）。**



(a) [5,200] **(b) [200,400]** **(c) [400,600]**

图4： FindShapes 的运行时间

实现）（数
据库内）。

8.3 要点总结

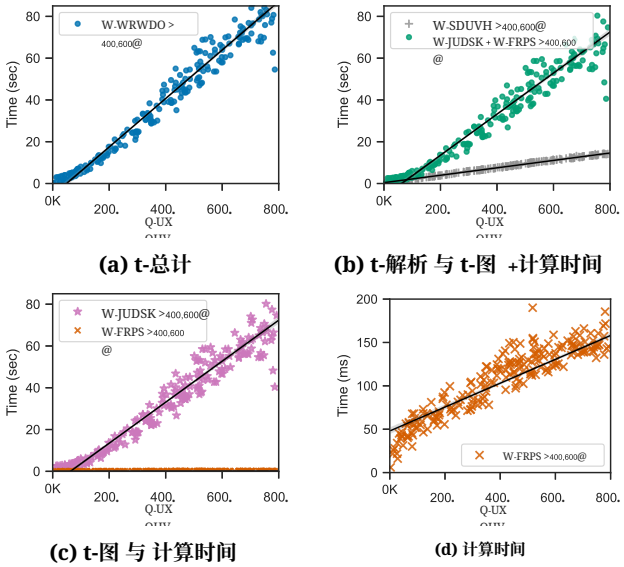
主要结论是，算法 IsChaseFinite[L] 由两个性质不同的组件构成，因为输入中不同的参数会影响它们的运行时间。一方面是数据库相关组件，负责查找数据库形状，其运行时间只受数据库大小影响。另一方面是数据库无关组件：它利用数据库形状简化给定的线性元组生成依赖集，构建简化后 TGD 集的依赖图，并在该图中查找特殊强连通分量；其运行时间主要受元组生成依赖数量（n-规则）影响。话虽如此，我们还观察到，谓词数量也会影响数据库无关组件的运行时间，因为它会影响形状数量，进而影响动态简化过程和依赖图的大小。最后，我们观察到，IsChaseFinite[L]的总运行时间相当合理，这有力地表明，对于线性元组生成依赖，快速检查追赶实例的有限性并非不切实际的目标。请注意，大多数算法的端到端总运行时间都花在寻找数据库形状上，这表明我们未来应致力于改进数据库相关组件。

9 结果验证

在第 7 节和第 8 节中，我们使用合成生成的数据库和元组生成依赖集合，对算法 IsChaseFinite[SL] 和 IsChaseFinite[L] 进行了实验评估。在最后一节中，我们使用文献中已有的数据库和元组生成依赖集合，旨在验证基于合成场景进行的压力测试分析所得的主要结论。

9.1 采用的场景

我们考察了三类数据库和元组生成依赖集，下面将简要介绍。这些内容也汇总于表 1，其中报告了一些统计数据，涉及：(i) 底层模式，即谓词数量（n-pred）和谓词元数（arity）；(ii) 数据库的规模，即原子数（n-atoms）和形状数（n-shapes）；以及 (iii) 元组生成依赖数量（n-rules）。



(a) t-总计 **(b) t-解析 与 t-图 +计算时间**

(c) t-图 与 计算时间 **(d) 计算时间**

图5：数据库无关组件的运行时间。

Deep. 这一类别包含同时满足弱无环条件的简单线性元组生成依赖集。它旨在测试以下场景：追赶应用次数较多、源实例规模较大，以及在数据交换设置中包含大量源到目标元组生成依赖和目标元组生成依赖。

LUBM。 这是一个广受欢迎的基准测试，包含一个使用核心描述逻辑（DL）EL 建模的本体，称为 Univ-Bench，以及一个名为 UBA 的数据生成器，用于生成 Univ-Bench 词汇表上的合成数据 [16]。我们使用 LUBM 类别中的四个成员，即 LUBM-1、LUBM-10、LUBM-100 和 LUBM-1K。需要说明的是，Univ-Bench 中的 DL 公理可以轻易转换为头部包含一个原子且原子中变量不重复的元组生成依赖。然而，并非所有公理都能导出线性元组生成依赖，因此我们只保留那些可以转换为线性元组生成依赖（同时也是简单线性的公理）。

iBench。 这是一个用于生成依赖（例如元组生成依赖）的框架，具有可控制广泛属性的调优参数 [5]。在我们的实验中，我们使用 iBench 生成的以下几组简单线性元组生成依赖：（i）STB-128，源自较早的 STBenchmark [4]，，是该类别中规模较小的场景；（ii）ONT-256，这是一个源实例规模大数倍的场景。对于这两组元组生成依赖，我们使用了 [8] 中的数据库；这些数据库由 [6] 中的数据生成器生成，每个源关系包含 1000 个元组。

下面我们讨论算法 IsChaseFinite[L] 在线性元组生成依赖上的实验评估。对于算法 IsChaseFinite[SL] 在简单线性元组生成依赖上的表现，除了它在上述所有场景中都能在几毫秒内运行这一事实外，没什么特别值得讨论的。这证实了对于简单线性元组生成依赖，可以非常高效地检查追赶实例是否有限。

类别	Name	谓词数	元数	原子数	形状数	n-规则
Deep	Deep-100	1299	4	1000	1000	4241
	Deep-200					4541
	Deep-300					4841
LUBM	LUBM-1	104	[1,2]	99547	30	137
	LUBM-10			1272575		
	LUBM-100			13405381		
	LUBM-1K			133573854		
iBench	STB-128	287	[1,10]	1109037	129	231
	ONT-256	662	[1,11]	2146490	245	785

表 1：Deep、LUMB 和 iBench 系列。

9.2 实验评估

我们在上述所有场景中运行算法 IsChaseFinite[L]，实验结果汇总于表 2。请注意，t-total 指检查追赶有限性的端到端运行时间，即 t-total = t-parse +t-graph + t-comp + t-shapes。我们用方框标出在采用内存中或数据库内实现来查找数据库形状时所获得的最佳端到端运行时间。

在所有场景中，解析时间（t-parse）都微不足道，因为影响解析时间的主要参数——规则数（n-rules）最多为 4000。构建依赖图的时间（t-graph）以及查找特殊强连通分量的时间（t-comp）也可以忽略不计。这是因为这些场景中的形状数量（n-shapes）有限，因此，利用这些形状进行的动态简化不会生成大量简化后的元组生成依赖，进而形成的依赖图也相当小。从表 2 可以明显看出，在所有场景中，最耗时的任务是查找数据库形状；这与我们在第 7 节中的观察一致。

我们分别报告了 FindShape 过程的内存实现和数据库内实现查找形状的时间（t-shapes）。对于 Deep 类别，由于底层数据库中有许多单例关系（即仅包含一个元组的关系），因此内存实现速度更快，加载元组和查找形状都能高效完成。另一方面，数据库内实现耗时更长，因为它针对每个关系运行一个查询，因而会产生大量查询。对于 LUBM 和 iBench 类别，由于谓词数量较少，数据库内实现只需运行少量查询即可找到形状，因此速度更快。请注意，通常情况下，数据库内实现的运行时间会随关系中的元组数量增加；LUBM 场景清楚地体现了这一点。因此，查找 LUBM-1K 的形状耗时较长。不过，这仍远低于使用内存实现查找数据库形状所需的时间，因为后者需要加载大量元组。

9.3 讨论

可以认为，本节进行的实验评估证实了第8节中对算法 IsChaseFinite[L] 的分析所得出的主要结论。具体而言，我们观察到，耗时最多的任务确实是寻找数据库形状，

				数据库内		内存中	
Name	t-解析	t-图	计算时间	t-shapes	t-总计	t-shapes	t-总计
Deep-100	214	90	10		6,957	447	**763**
Deep-200	265	116	9	6,641	7,033	447	**839**
Deep-300	234	100	11		6,986	500	**846**
LUBM-1	84	10	1	221	**318**	2,724	2,820
LUBM-10	46	10	1	830	**889**	10,943	11,002
LUBM-100	45	11	1	6,396	**6,454**	70,131	70,189
LUBM-1K	43	231	80	65,578	**65,932**	854,015	854,369
STB-128	78	18	7	4,991	**5,096**	7,379	7,484
ONT-256	179	35	8	11,726	**11,949**	15,761	15,984

表 2：IsChaseFinite[L] 的运行时间（毫秒）。

而数据库无关组件所耗费的时间可以忽略不计。此外，我们发现，在实践中，检查追赶实例的有限性可以相当高效地完成。具体来说，对于包含数千个元组生成依赖的集合（如 Deep 场景），以及包含数百万个事实的数据库（如 LUBM-10），耗时不到一秒。对于谓词数量较多的模式（如 STB-128 和 ONT-256），以及包含大量原子的数据库（如 LUBM-100），耗时不到 10 秒。最后，对于包含数亿个原子的超大型数据库（如 LUBM-1K），耗时约一分钟。考虑到输入的实际规模，这一耗时是合理的。

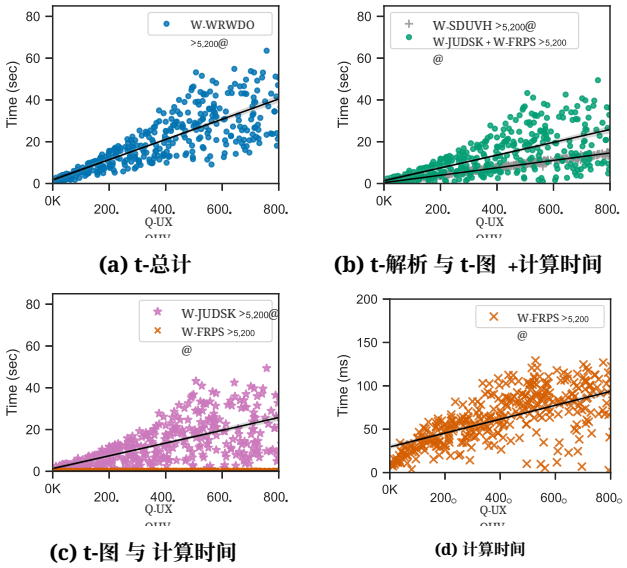
本节实验评估的另一个有趣结论是，在内存实现和数据库内实现之间，FindShape 的实现方式并无明确的优劣之分。具体而言，当数据库中每个关系的元组数量较少时，内存实现更受青睐；而当底层模式包含的谓词较少且元数较小时，数据库内实现的性能更好。对于谓词众多且每个谓词在输入数据库中都包含大量元组的模式，两种实现都需要耗费大量时间，此时离线计算数据库形状或许更为可取。

10 结论与未来工作

我们的工作首次系统地尝试对用于在存在（简单）线性受限生成依赖时判定半无知追赶是否终止的算法进行实验评估。我们的分析表明，对于简单线性元组生成依赖，即使数据库和元组生成依赖集合非常庞大，我们也能高效地检查追赶是否终止。对于线性元组生成依赖，算法的整体运行时间相当合理，但仍有改进空间。有趣的是，我们的分析显示，针对线性元组生成依赖的算法由两个独立的分量构成：数据库相关组件和数据库无关组件。算法的这种模块化特性使我们能够分别研究和改进这两个分量。具体而言，我们观察到，开销较大的分量是数据库相关组件，因此今后可以着重提升该组件的性能。尽管我们的分析依赖于用于查找形状的过程的数据库内实现和内存实现，但我们也可以根据底层应用采用其他技术，而不影响数据库无关组件。一个值得探索的方向是将形状物化并在数据库中增量更新形状，从而提升数据库相关组件的性能。

参考文献

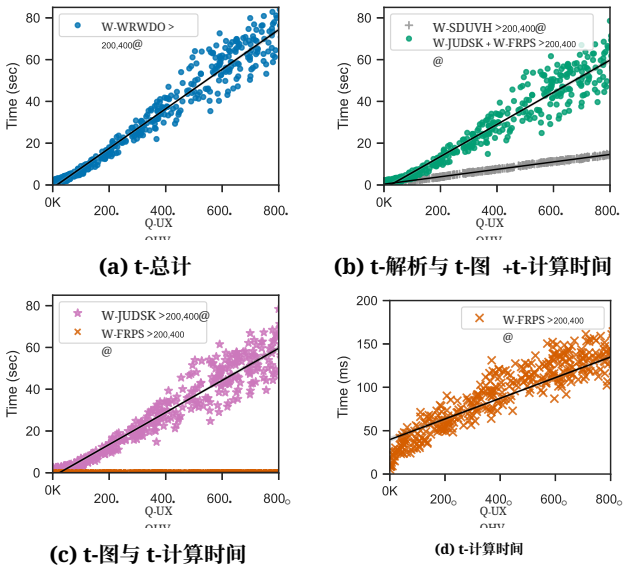
[1] Serge Abiteboul、Richard Hull 和 Victor Vianu。1995。《数据库基础》。Addison-Wesley。[2] Rakesh Agrawal、Ramakrishnan Srikant 等。1994。用于挖掘关联规则的快速算法。载于《第 20 届国际超大型数据库会议论文集》，VLDB，第 1215 卷。Citeseer，487–499。[3] Alfred V. Aho、Yehoshua Sagiv 和 Jeffrey D. Ullman。1979。关系表达式类的高效优化。《ACM 数据库系统汇刊》4，4（1979），435–454。[4] Bogdan Alexe、Wang Chiew Tan 和 Yannis Velegrakis。2008。STBenchmark：面向映射系统的基准测试。载于《VLDB基金会论文集》1，1（2008），230–244。[5] Patricia C Arocena、Boris Glavic、Radu Ciucanu 和 Renée J Miller。2015。iBench 集成元数据生成器。《VLDB基金会论文集》9，3（2015），108–119。[6] Denilson Barbosa、Alberto Mendelzon、John Keenleyside 和 Kelly Lyons。2002。ToXgene：一种基于模板的 XML 数据生成器。载于《2002 年 ACM SIGMOD 数据管理国际会议论文集》，616–616。[7] Catriel Beeri 和 Moshe Y. Vardi。1984。数据依赖的证明过程。《ACM 杂志》31，4（1984），718–741。[8] Michael Benedikt、George Konstantinidis、Giansalvatore Mecca、Boris Motik、Paolo Papotti、Donatello Santoro 和 Efthymia Tsamoura。2017。追赶基准测试。载于 PODS。37–52。[9] Marco Calautti、Georg Gottlob 和 Andreas Pieris。2022。非一致终止的追赶：规模与复杂性。载于 PODS。369–378。[10] Marco Calautti 和 Andreas Pieris。2021。半无知追赶终止性：黏性情形。《理论计算机系统》65，1（2021），84–121。[11] Andrea Calì、Georg Gottlob 和 Thomas Lukasiewicz。2012。一个用于本体上可处理查询应答的通用 Datalog 框架。《网络语义学杂志》14（2012），57–83。[12] Diego Calvanese、Giuseppe De Giacomo、Domenico Lembo、Maurizio Lenzerini 和 Riccardo Rosati。2007。描述逻辑中的可处理推理与高效查询应答：DL-Lite 家族。《自动推理杂志》39，3（2007），385–429。[13] Alin Deutsch、Alan Nash 和 Jeff B. Remmel。2008。追赶的重新审视。载于 PODS。149–158。[14] Edsger W Dijkstra。1982。在有向图中寻找最大的强连通分量。载于《计算领域精选著作：个人视角》。Springer，22–30。[15] Ronald Fagin、Phokion G. Kolaitis、Renée J. Miller 和 Lucian Popa。2005。数据交换：语义与查询应答。《理论计算机科学》336，1（2005），89–124。[16] Yuanbo Guo、Zhengxiang Pan 和 Jeff Heflin。2005。LUBM：OWL 知识库系统的基准测试。《网络语义学杂志》3，2-3（2005），158–182。[17] Markus Krötzsch、Maximilian Marx 和 Sebastian Rudolph。2019。终止追赶的威力（特邀报告）。载于 ICDT。3:1–3:17。[18] Michel Leclère、Marie-Laure、Michaël Thomazo 和 Federico Ulliana。2019。一种判定线性存在规则追赶终止性的统一方法。载于 ICDT。18:1–18:19。[19] David Maier、Alberto O. Mendelzon 和 Yehoshua Sagiv。1979。检验数据依赖蕴涵关系。《ACM 数据库系统汇刊》4，4（1979），455–469。[20] Bruno Marnette。2009。广义模式映射：从终止性到可处理性。载于 PODS。13–22。[21] Yavor Nenov、Robert Piro、Boris Motik、Ian Horrocks、Zhe Wu 和 Jay Banerjee。2015。RDFox：高可扩展性的 RDF 存储。载于 ISWC。3–20。[22] Micha Sharir。1981。一种强连通性算法及其在数据流分析中的应用。《计算机与数学及其应用》7，1（1981），67–72。[23] Robert Tarjan。1972。深度优先搜索与线性图算法。《SIAM Journal on Computing》1，2（1972），146–160。[24] Jacopo Urbani、Markus Krötzsch、Ceriel J. H. Jacobs、Irina Dragoste 和 David Carral。2018。使用 VLog 高效构造 Horn 逻辑模型——系统描述。载于 IJCAR。680–688。



(a) t-总计 (b) t-解析 与 t-图 +计算时间

(c) t-图 与 计算时间 (d) 计算时间

图 6：谓词配置 [5,200] 的数据库无关分量的运行时间。



(a) t-总计 (b) t-解析与 t-图 +t-计算时间

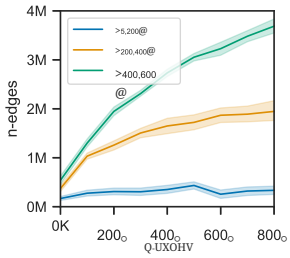
(c) t-图与 t-计算时间 (d) t-计算时间

图7：谓词配置 [200,400] 下数据库无关组件的运行时间。

数据库无关组件

请注意，出于可读性考虑，在第8节中，我们省略了展示 chase 终止算法 IsChaseFinite[L] 的数据库无关组件运行时间的图，这些图对应较小的谓词配置。谓词配置 [5,200] 的图见图 6，而配置 [200,400] 的图见图 7。与谓词配置 [400,600]，类似，我们可以观察到，只要我们

n-规则数量增加，而 t-计算时间则增长非常缓慢。不过，从图中可以明显看出，当我们转向更小的谓词配置和更大的 TGD 集时，上述线性趋势就不那么明显了。这一现象可以解释如下。随着模式中出现的谓词数量减少，大量元组生成依赖可能只会在底层依赖关系中产生数量有限的边。这是因为许多元组生成依赖会导向相同的边，而这些边在图中当然只会计入一次。以下图表证实了这一点



该图展示了我们实验评估中所用线性元组生成依赖集的依赖图平均边数。对于较小的谓词配置，随着元组生成依赖数量增加，图的规模增幅小于较大谓词配置的增幅。具体而言，对于配置 [5,200]，随着元组生成依赖数量增加，依赖图的边数几乎保持不变。