arXiv:2301.04003v2 [cs.DB] 2 Dec 2023

无连接的变更传播

Qichen Wang 香港浸会大学
qcwang@hkbu.edu.hk

Xiao Hu
滑铁卢大学
xiaohu@uwaterloo.ca

Binyang Dai, Ke Yi
香港科技大学
{bdaiab,yike}@ust.hk

摘要

我们重新审视了经典的更新场景下查询求值的变更传播框架。标准框架以查询计划为输入，并将中间视图实体化；这会在空间和时间上产生高昂的多项式成本，而罪魁祸首正是连接算子。本文提出了一种无需连接的新变更传播框架，从而自然地避免了这种多项式膨胀。与此同时，我们证明，新框架仍支持对增量和完整查询结果进行常数延迟枚举，与标准框架相同。此外，我们定量分析了其更新成本，不仅重现了该问题近期的许多理论结果，还提供了一种有效的查询计划优化方法。该新框架也易于集成到现有的流式数据库系统中。实验结果表明，我们使用 Flink DataStream API 实现的系统原型，在空间、时间和延迟方面均显著优于其他系统。

1 引言

我们研究更新下的查询求值问题，也称为增量视图维护。给定一个查询 Q、一个数据库 D以及一系列更新，其中每次更新都是插入或删除一个元组，目标是持续维护查询结果 $Q(D)$。更准确地说，有两种模式可以向用户（终端用户或上层应用程序）返回已更新的 $Q(D)$：完整枚举 和增量枚举。前者是拉取式的，也就是说，系统会在用户请求时被动地返回 $Q(D)$；而在后一种情况下，我们会在每次更新 t后向用户推送增量 $\Delta Q(D, t)$，也就是由插入/删除 t导致的 $Q(D)$ 变化。这两种模式适用于不同的场景。如果 $Q(D)$ 很大，就不能过于频繁地进行完整枚举，而且两次请求之间可能会错过一些短暂事件。增量枚举能够以低延迟提供实时响应，但要求用户能够及时“消费”这些增量。它可以被视为一个流入流出算子：输入是基表更新流，输出是查询结果的更新流（即增量流）。如果用户希望始终获得完整且准确的 $Q(D)$，就必须维护 $Q(D)$，并在收到增量时据此更新。如果可以接受近似结果，则可以改用一些效率更高的流式算法。

变更传播。变更传播 [12, 27, 34] 是数据库系统中广泛使用的一种框架，用于解决这一问题。它可以实例化为任意查询计划；查询计划是一棵树，叶节点是基关系，每个内部节点都是一个关系运算符。在每个内部节点处，它维护与该节点子树相对应的子查询结果，这通常称为物化视图。图 1(a) 展示了一个特定的

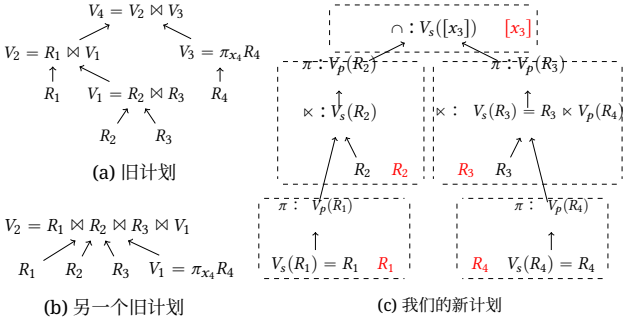


图 1：对于 $Q = \pi_{x_1, x_2, x_3, x_4} R_1(x_1, x_2) \bowtie R_2(x_2, x_3) \bowtie R_3(x_3, x_4) \bowtie R_4(x_4, x_5)$，1(a) 和 1(b) 是标准变更传播框架下的两个计划，而 1(c) 是我们的新计划。

基准测试 [30]中的 4跳查询的查询计划

$$Q := \pi_{x_1,x_2,x_3,x_4} R_1(x_1, x_2) \bowtie R_2(x_2, x_3) \bowtie R_3(x_3, x_4) \bowtie R_4(x_4, x_5).$$

在标准变更传播框架下，我们维护四个物化视图 V_1，V_2，V_3，$V_4 = Q$（如果只需要进行增量枚举，则无需维护 V_4）。当一个元组 t在某个关系中被插入或删除时，例如 R_1，变更会沿着叶到根路径传播至根节点。具体而言，首先计算

$$\Delta V_2 = \Delta R_1 \bowtie V_1 = t \bowtie V_1 \qquad \Delta Q = \Delta V_4 = ，然后计$$

$\Delta V_2 \bowtie V_3$ 算。请注意，借助物化视图，可以避免在更新期间重新计算部分子查询。

然而，代价在于空间：在最坏情况下，V_1 和 V_2 的大小都可能达到二次级别 [4]。为避免空间膨胀，可以采用另一种查询计划，例如图 1(b) 所示的计划。该查询计划没有任何物化视图（除了 $V_1 = \pi_{x_4} R_4$，其大小至多为线性），但每次更新 R_4时都必须计算多路连接，例如 $R_1 \bowtie R_2 \bowtie R_3 \bowtie t$，这可能需要二次时间。更糟的是，对于涉及更多关系的查询，这种二次膨胀会进一步加剧 [4]。

先前工作设计了先进技术，以应对这种空间或时间膨胀。动态 Yannakakis 算法 [20–22]具有线性空间和线性更新时间，同时支持自由连通查询的常数延迟枚举[1]；对于q-层次查询，更新时间还能进一步降低至 $O(1)$ 摊销[2]。与此同时，Berkholz 等人 [8] 为 q-层次化情形设计了另一种算法，具有相同的空间和时间保证。然而，这些算法尚未集成到任何成熟的数据库或数据仓库产品中，原因可能是其技术复杂，而且使用了现有数据库系统中并不常见的非标准运算。

[1]引言中的所有技术术语均在第 3 节中正式定义。[2]本文中的所有更新时间界均为摊销界。

无连接的变更传播。本文的主要贡献是在标准变更传播框架下实现上述结果（并针对某些查询类别和/或更新序列加以改进）。我们的观察是，唯一可能导致超线性膨胀的关系运算符是连接。因此，如果查询计划不包含连接，那么空间和更新时间都至多为线性。为避免连接，我们的高层策略是将查询计划中的每个连接替换为一个半连接（或交集）加一个投影。不过，并非每个查询计划都适用于这种替换策略。因此，本文的关键技术贡献是为每个自由连通合取查询构造这样一种查询计划。例如，图 1(c) 展示了前述查询的一种无连接查询计划，其细节将在第4节中阐述。

由于我们的查询计划没有连接，线性空间和线性更新时间可直接得出。不过，仍有两个技术挑战：(1) 如何支持常数延迟枚举；(2) 如何实现优于线性的更新时间。在传统查询计划中，根节点对应查询结果 $Q(D)$，因此问题 (1) 很简单。由于我们的查询计划不含连接，计划中没有任何节点对应 $Q(D)$。相反，我们的查询计划可以看作是一个多项式规模 $Q(D)$ 的紧凑线性规模表示。借鉴静态情形 [5], 中的思路，我们展示了如何通过恰当地遍历这一紧凑表示，以常数延迟枚举 $Q(D)$ 。另一方面，支持对增量 $\Delta Q(D,\ t)$进行常数延迟枚举，与静态情形截然不同，因此我们需要利用查询计划若干重要性质的新技术。

为解决问题 (2)，Wang 和 Yi [36] 提出了更新序列的包围度 λ 这一概念，用以刻画更新序列的难度。最坏情况下，包围度是线性的；但在许多常见情形下，它通常是常数，例如任何先进先出（FIFO）更新序列。他们还为外键无环查询设计了一种算法，其更新成本为$O(\lambda)$ 。这类查询相对容易处理，因为其结果大小至多为线性，因此不会受到非键连接（例如自由连通查询）所导致的多项式膨胀问题的影响。事实上，我们证明（参见定理 6.2），存在一个简单的自由连通查询，即使在先进先出更新序列下，也不可能实现 $O(|D|^{1/2-\varepsilon})$ 更新时间。这意味着，先前定义的 λ 无法用于自由连通查询。尽管如此，我们表明，只需对定义作简单的放宽，λ 仍是衡量更新复杂度的恰当指标；特别是，我们证明，对于每个自由连通查询，在新定义下，采用我们的查询计划进行变更传播可实现 $O(\lambda)$ 更新时间。为了进一步说明新定义 λ的实用性，我们证明，对于某些查询（例如 q-层次查询）和/或更新序列（例如先进先出或仅插入），λ 确实是一个较小的常数。对于一般查询，λ 也为变更传播中何种查询计划才算良好提供了指导。

我们的成果。 具体而言，本文取得了以下成果：

（1）我们展示了如何为任意自由连通合取查询构造一个不含连接的变更传播查询计划，使该查询计划所需的空间为线性，更新时间为 $O(\lambda)$，其中更新序列的包围度 λ 采用了适当定义的概念。

（2）我们展示了如何在我们的查询计划中，以常数延迟枚举完整查询结果以及每个增量。（3）我们证明，对于某些类别的合取查询（例如 q-层次查询）和/或特殊的更新序列（例如先进先出或仅插入），λ 是一个常数。这些结果不仅重现了先前关于 q-层次查询的已知结果 [8, 20]，还将其扩展到涵盖实践中常见的许多其他情形。（4）我们展示了我们的框架如何处理各种扩展，例如选择、聚合和非自由连通查询。（5）我们通过在 Flink 之上实现这一新框架，并将其与最先进的视图维护方法和流式 SQL 系统进行比较，展示了其实用性。

2 相关工作

我们的新变更传播框架受到多个研究方向的启发。在静态情形下，经典的 Yannakakis 算法 [37] 对于每个自由连通查询的运行时间为 $O(|D| + |Q(D)|)$ 。它分为两个阶段。第一阶段通过一系列半连接，在 $O(|D|)$ 时间内移除所有悬挂元组；第二阶段执行两两连接，在 $O(|Q(D)|)$ 时间内计算 $Q(D)$ 。动态 Yannakakis 算法 [20] 将该算法扩展到了动态情形，但它偏离了变更传播框架，因此难以集成到现有数据库系统中。我们的算法也可以视为 Yannakakis 算法的动态版本，但它严格遵循标准变更传播框架，同时实现了更好的运行时间。对于自由连通查询，动态 Yannakakis 算法的更新成本为 $O(|D|)$，而我们的算法实现了 $O(\lambda)$ 的更新时间，其中 λ 是更新序列的包围度 。对于所有更新序列，我们都有 $\lambda \leqslant |D|$ ，而在真实世界的更新序列中，前者通常小得多。此外，动态 Yannakakis 仅对 q- 层次查询实现 $O(1)$ 的更新时间；而对于非 q- 层次查询，如果更新序列具有某些特殊性质（例如先进先出或仅插入，正式定义见第 6.1 节），我们的算法也能实现 $O(1)$ 的更新时间。在某些非 q-层次查询上，动态 Yannakakis 与我们的算法之间的差距可高达 $\Theta(|D|)$ （见示例 6.12）。

Bagan 等人 [5] 指出，在静态情形下，可以改进 Yannakakis 算法的第二阶段，以支持常数延迟枚举。我们将他们的思路加以改造，使其能够在动态情形下支持我们的查询计划进行枚举。然而，由于静态情形下不存在增量这一概念，我们需要一些新思路来支持具有常数延迟的增量枚举，这在很大程度上依赖于我们的查询计划的一些优良特性。

Kara 等人 [26] 表明，对于层次化（但非 q-层次化）查询，可以通过增加枚举延迟来换取更快的更新时间。我们没有考虑这种权衡，因为我们认为常数延迟很重要，而且对于大多数查询和更新序列，我们的更新成本λ 已经足够低。此外，他们提出的权衡仅适用于完全枚举，不适用于增量枚举。尽管如此，对于 λ 较高的情形，探索这种权衡将是一个有趣的方向。

在标准变更传播框架中，对基关系的一次更新可能会导致中间

视图发生许多变化。为解决这一问题，研究者提出了高阶增量视图维护（HIVM）[3]。它将视图的变更作为另一个查询（增量查询），并递归地维护该增量查询。在实践中，对于许多复杂查询，HIVM 的表现优于增量视图维护（IVM），还可以扩展用于加速若干机器学习任务 [32, 33],，但其更新时间没有理论保证。此外，HIVM 仍使用超线性空间。

该问题也与流连接有关。具体而言，现金寄存器流对应于仅插入更新序列，而转闸流则是包含任意插入和删除的更新序列。滑动窗口流模型是先进先出更新序列的一种特殊情况。大多数流处理系统（如 Flink [9] 和 Trill [11] ）在多路流连接中采用标准变化传播，我们将在第 8 节对此进行比较。一些专用系统针对双路流连接 [13, 16, 24, 29, 35], 进行了设计，但无法扩展到多路连接。

3 预备知识

3.1 问题定义

合取查询。我们关注以下形式的合取查询（CQ）：

$$Q := \pi_{\mathbf{y}} \left(R_1(e_1) \bowtie R_2(e_2) \bowtie \cdots \bowtie R_n(e_n) \right), \tag{1}$$

其中，每个 R_i 都是一个关系，具有一组属性/变量 $e_i, i = 1, \ldots, n$。每个元组 $t \in R_i$ 为 e_i中的每个属性赋一个值。对于任意 $x \in e_i$，$t[x] = \pi_x t$表示 t在属性 x上的值。类似地，对于属性子集 $e \subseteq e_i$，$t[e] = \pi_e t$ 表示由 t 在属性 e上的值组成的元组。

令 $\mathbf{V} = e_1 \cup \cdots \cup e_n$为查询中所有属性的集合。我们将 $\mathbf{y} \subseteq \mathbf{V}$称为输出属性，而 $\bar{\mathbf{y}} = \mathbf{V} - \mathbf{y}$则称为非输出属性，也称为存在变量。如果$\mathbf{y} = \mathbf{V}$，则这样的查询称为全连接查询；否则，称为连接投影查询。为简单起见，我们假设每个 R_i in Q 都不相同，即查询不存在自连接。尽管如此，自连接也很容易处理：假设一个关系 R 在查询中出现两次（属性重命名不同）。那么我们将它们视为 R的两个相同副本，并且对于针对 R的任何更新，都将该更新应用到 R的两个副本上。

给定一个数据库 D，我们将 Q在 D上的查询结果记为 $Q(D)$。我们用 $Q(D \ltimes t)$ 表示依赖于给定元组 t的查询结果，并将 $Q(D \ltimes t)$ 称为由 t见证的查询结果。这种见证查询将在本文中频繁使用。给定一个形式为(1)的查询 Q和一个元组 $t \in R_i$，显然

$$Q(D \ltimes t) = \pi_{\mathbf{y}} \left(R_1 \bowtie \cdots \bowtie R_{i-1} \bowtie \{t\} \bowtie R_{i+1} \bowtie \cdots \bowtie R_n \right).$$

注意，对于全连接合取查询，我们有 $Q(D \ltimes t) = Q(D + t) \ltimes t$；对于连接-投影查询，由于对$\mathbf{y}$进行了投影，$Q(D \ltimes t)$ 中可能不会出现 t本身。在分析算法的开销时，我们采用数据复杂度的概念，即将查询 Q的大小视为常数，而将 $|D|$ 作为渐近参数。

半连接。 半连接 $R_i(x_i) \ltimes R_j(x_j)$ 定义为

$$R_i(x_i) \ltimes R_j(x_j) = \{t | t \in \pi_{x_i} R_i \bowtie R_j\}.$$

更新与增量。 数据库 D 的更新 是指在 D的某个关系 R_i中插入或删除元组 t。本文采用集合语义。我们将 $D + t$表示为在插入 t之后的数据库，将 $D - t$表示为删除 t之后的数据库。具体而言，这意味着，如果 R_i已包含 t，那么将 t插入 R_i不会改变 R_i；如果 R_i不包含 t，从 R_i中删除 t也不会产生任何影响。我们忽略这些无效更新。

对 Q 进行更新的增量 定义为

$$\Delta Q(D, t) = Q(D + t) - Q(D)$$

在插入 t 的情况下，以及

$$\Delta Q(D, t) = Q(D) - Q(D - t)$$

在删除的情况下。对于全连接查询，$\Delta Q(D,\ t) = Q(D \ltimes t)$.对于连接-投影查询，$\Delta Q(D,\ t) \subseteq Q(D \ltimes t)$。特别地，即使 $Q(D \ltimes t) \neq \emptyset$，也可能出现 $\Delta Q(D,\ t) = \emptyset$。

对于完全枚举和增量枚举，我们都要求满足常数延迟 [5]，也就是说，从枚举过程开始到 $Q(D)$ 中的第一个元组（或 $\Delta Q(D,\ t)$）、任意一对相邻元组之间，以及最后一个元组到枚举过程终止之间的时间，都应有常数上界。

3.2 合取查询分类

无环合取查询。 无环查询有多种等价定义 [7, 15],，此处我们采用一种基于广义连接树 [20]的定义。广义关系 R_e定义在属性 $e \subseteq \mathbf{V}$ 的一个子集上，并且区别于原始关系。

定义 *3.1*（无环查询）。 *</style}*合取查询V，E，y) 是无环的，当且仅当存在一棵树 T，其中每个节点都对应一个不同的输入关系或广义关系，并满足以下性质：

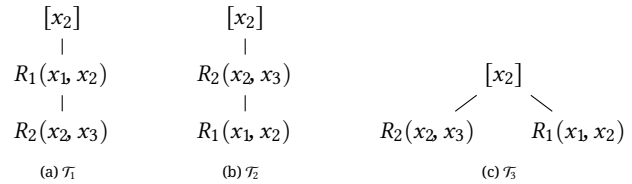
- **（覆盖性质）**E 中的每个输入关系都对应 T 中一个不同的节点；此外，T 的每个叶节点都对应 E 中的一个输入关系；
- **（连通性质）**对于每个属性 $x \in \mathbf{V}$，T 中所有包含 x的节点构成 T 的一个连通分量；

T 称为 Q 的广义连接树。如果 T 中的所有节点都对应 E 中的关系，则称 T 为传统连接树。示例如图2(c)所示。在 T 中，我们用 r 表示根节点，用 $\mathbf{T}_e$ 表示以节点 e 为根的子树，用 $\mathbf{C}_e$ 表示节点 e 的子节点集合，并用 p_e 表示节点 e 的父节点。如果 e 是叶节点，则 $\mathbf{C}_e = \emptyset$；对于根节点 r，$p_r = \emptyset$。令 $\mathbf{key}(e) = e \cap p_e$ 表示节点 e 和 p_e 之间的连接键。

自由连通合取查询。 合取查询 $Q = (\mathbf{V},\ \mathbf{E},\ \mathbf{y})$ 是自由连通的，当且仅当 Q 和 $(\mathbf{V}\ \{e_1,\ \ldots\ldots,\ e_n,\ \mathbf{y}\},\ \mathbf{y})$ 都是无环合取查询。根据定义，任何自由连通查询都必须是无环的，而无环的全连接查询必须是自由连通的。在本文的论述中，我们需要一个基于自由连通树的等价定义（该等价性在附录 A 中得到证明）：

定义 *3.2*（自由连通合取查询）。一个合取查询 $Q = (\mathbf{V},\ \mathbf{E},\ \mathbf{y})$ 是自由连通的，当且仅当存在一棵树 T，其中每个节点对应一个不同的输入关系或广义关系，并满足以下性质：

- **（覆盖性质）**E 中的每个输入关系都对应 T 中一个不同的节点；此外，T 的每个叶节点都对应 E 中的一个输入关系；
- **（连通性质）**对于每个属性 x，T 中所有包含 x 的节点构成 T 的一个连通分量；



(a) $\mathcal{T}_1$ (b) $\mathcal{T}_2$ (c) $\mathcal{T}_3$

图2： $Q_1 = R_1(x_1, x_2) \bowtie R_2(x_2, x_3)$**的三个自由连通连接树。在2(c)中，节点** $[x_2]$ **是一个只有一个属性** x_2**的广义关系。**$\mathcal{T}_1$，$\mathcal{T}_2$ **的高度为**2**，而**$\mathcal{T}_3$ **的高度为**1。

• **（守卫性质）**如果节点 e 对应一个广义关系，那么$e \subseteq e'$ 对于 e的每个子节点 e' 都成立；• **（上方性质）**任何对应广义关系的节点都位于任何对应输入关系的节点上方；• **（连通性质）**存在 T 的一个连通子树 $\mathrm{E}_{\text{连}}$，满足：(i) $\mathrm{E}_{\text{连}}$ 包含 T 的根节点；(ii) 对于任意节点 $e \in \mathrm{E}_{\text{连}}$，键$(e) \subseteq \mathbf{y}$；(iii) $\mathbf{y} \subseteq \bigcup_{e \in \mathcal{E}_{\text{con}}} e$。

T称为Q的自由连通连接树，E_{con} 称为连通子树。T的高度定义为任意叶到根路径上关系的最大数量，不计广义关系。例如，对于 $Q_1' := \pi_{x_2} R_1(x_1, x_2) \bowtie R_2(x_2, x_3)$，图2中的三个自由连通连接树都是有效的自由连通连接树，其连通子树为 $\{[x_2]\}$。

Q-层次化合取查询 [8]。合取查询 Q 是q-层次化的，当且仅当：(1) 对于每一对属性 x_1, x_2，$\mathrm{E}_{x_1} \subseteq \mathrm{E}_{x_2}$ 或 $\mathrm{E}_{x_2} \subseteq \mathrm{E}_{x_1}$ 或 $\mathrm{E}_{x_1} \cap \mathrm{E}_{x_2} = \varnothing$；并且 (2) 对于每一对属性 x_1, x_2，如果 $x_1 \in \mathbf{y}$ 且 $\mathrm{E}_{x_1} \subsetneq \mathrm{E}_{x_2}$，那么 $x_2 \in \mathbf{y}$，其中 E_x 表示包含属性 x的关系集合。有趣的是，合取查询是 q-层次化的，当且仅当存在一棵高度为 1 的自由连通连接树（见附录 A）。

首先，一个全连接查询可以在线性时间内求值（以输入大小和输出大小计），当且仅当它是无环的；对于连接-投影合取查询，这一完整类别扩展到自由连通查询。此外，自由连通合取查询和 q-层次化合取查询在查询枚举中发挥了重要作用。[5] 证明，在静态场景中，对于合取查询，经过线性时间预处理后能够以常数延迟进行枚举，当且仅当它是自由连通的。Berkholz 等人 [8] 证明，在动态场景中，对于合取查询，若使用的数据结构能够在常数时间内更新，则能够以常数延迟进行枚举，当且仅当该查询是 q-层次化的。

4 无连接的变更传播

4.1 一种新的查询计划

给定一个自由连通查询 Q，我们的新查询计划由 Q 的一棵自由连通连接树 T 引导。我们以图 1 中的查询为例说明这一构造，其中连接树以红色标出（请注意，连接树并非唯一）。遵循这棵连接树的常规查询计划会计算一系列连接 $(R_1 \bowtie R_2) \bowtie (R_3 \bowtie \pi_{x_4} R_4)$。在我们的新查询计划中，我们将每个连接替换为一个半连接，后接一个投影。更准确地说，我们为每个节点 $e \in$T 维护两个视图 $V_s(R_e)$ 和 $V_p(R_e)$，分别是半连接视图和投影视图，并按以下方式递归定义。

算法 1： PlanGeneration$(Q,\ T)$

输入： 一个自由连通连接树 T，用于查询 Q；
输出： 一个新的查询计划 T，用于 Q；
对于 后序遍历中的节点 e T**执行**
  将 e替换为 $V_s(R_e)$，位于 T中；
  如果 e 不是 T 的根节点**则**
    在 $V_s(R_e)$与 e的父节点之间添加 $V_p(R_e)$；
返回 T；

每个非根节点 $e \in$ T 都有一个投影视图

$$V_p(R_e) := \pi_{\mathsf{key}(e)} V_s(R_e). \tag{2}$$

请注意，根节点没有投影视图。

为了定义半连接视图 $V_s(R_e)$，我们区分三种情况 .
(i) 如果 e 是叶节点，R_e 是输入关系，且 $V_s(R_e) := R(e)$。
(ii) 如果 e 是内部节点，且 R_e 是输入关系，那么

$$V_s(R_e) := R_e \ltimes V_p(R_{e_1}) \ltimes \cdots \ltimes V_p(R_{e_k}), \tag{3}$$

其中 $\mathrm{C}_e = \{e_1,\ e_2, \ldots,\ e_k\}$ 是 e 的子节点 .
(iii) 如果 e 是对应于广义关系 R_e 的内部节点，由于所有 $V_p(R_{e_i})$ 的属性键 $(e_i) = e_i \cap e = e$ 对每个 $i \in [k]$ 都相同（根据定义 3.1 中的守卫性质），(3) 可简化为交集：

$$V_s(R_e) := V_p(R_{e_1}) \cap \cdots \cap V_p(R_{e_k}). \tag{4}$$

我们的查询计划只需使用上述公式将这些视图连接起来。算法 1 以自由连通连接树为输入，并在我们的框架下输出一种新的查询计划。图 1(c) 展示了示例查询的新查询计划。注意，R_2 和 R_4 属于情况 (ii)，而根节点 $[x_3]$ 属于情况 (iii)。

由于投影和半连接（包括作为特例的交集）都不会扩大输入关系，因此以下结论显而易见：

引理 4.1。 查询计划中的所有视图大小均为 $O(|D|)$。

例 *4.2*。 图 3(a)展示了为图 1中的查询构建的初始索引。对于 R_1和 R_4，半连接视图和投影视图都定义为其自身。$V_s(R_2)$ 包含 R_2中能够与 $V_p(R_1)$连接的元组，其中包括 $(2,\ 2)$ 和 $(2,\ 4)$。$V_s(R_3)$ 的定义类似，其中包含来自 R_3的 4 个元组。对于广义节点 $[x_3]$，我们定义广义关系 $R([x_3]) = V_p(R_2) \cup V_p(R_3)$。只有元组 (4) 属于 $R([x_3])$，因为 $R([x_3])$ 中的其他所有元组都无法与 $V_p(R_2)$ 和 $V_p(R_3)$连接：它们的计数器必须为 2。

4.2 变更传播

使用我们新的查询计划进行变更传播，可以采用标准的（实际上，对某些运算符而言甚至更简单的）传播公式 [12]。为求完整起见，我们将在下文简要介绍这些公式；理解第 5 节中的算法也需要这些内容。

S-更新 当 $V_s(R_e)$ 中某个 e发生更新时，我们使用 S-更新，根据公式 (2) 更新 $V_p(R_e)$。通过推导计数 [12], 这一在投影中传播变更的标准技术，可以在 $O(1)$ 时间内完成。具体来说，我们为每个元组 $t' \in V_p(R_e)$ 关联一个计数器 count$[t']$，用于存储满足 $t[\mathsf{key}(e)] = t'$ 的元组 $t \in V_s(R_e)$ 的数量。详细过程如下，

算法 2： S-更新$(e,\ t)$

输入： 更新 t来自 $V_s(R_e)$;
输出： 已更新 $V_p(R_e)$;
$t' \leftarrow t$[键(e)];
如果 t是对 $V_s(R_e)$的插入**则**
  如果 $t' \in V_p(R_e)$**则** 计数$[t'] \leftarrow$计数$[t'] + 1$;
  else
    $V_p(R_e) \leftarrow V_p(R_e) \cup \{t'\}$, 计数$[t'] \leftarrow 1$,
    P-更新$(p_e,\ t')$;
否则
  如果 计数$[t'] = 1$**则**
    $V_p(R_e) \leftarrow V_p(R_e) - \{t'\}$, P-更新$(p_e,\ t')$;
  否则 计数$[t'] \leftarrow$计数$[t'] - 1$;

该过程需要区分插入和删除，详见算法 2。请注意，为使算法在 $O(1)$时间内运行，我们需要在 $V_p(R_e)$上建立哈希索引。

P-更新 设 e_i 是 e的子节点。当某个$V_p(R_{e_i})$发生更新时，如果 e 是输入关系，我们就使用 P-更新，根据公式 (3) 更新 $V_s(R_e)$；如果 e 是广义关系，则根据公式 (4) 更新 $V_s(R_e)$。我们先考虑前一种情况；后一种情况类似。

半连接 [18] 的标准变更传播公式会将其改写为一个连接后接一个投影，例如 $R_e \ltimes R_{e_i}$: $= \pi_e(R_e \bowtie R_{e_i})$。这就违背了避免连接的初衷。然而请注意，在我们的查询计划中，R_{e_i} 已在半连接之前投影到键 $(e_i) = e_i \cap e \subseteq e$ 上，因此我们可以用一种非常简单且高效的方式，将整个多路半连接 (3) 作为一个算子来维护；这也可以视为推导计数的一种"水平"形式。更准确地说，我们为 R_e 中的每个元组 t' 维护一个计数器 count$[t']$，记录满足 t'[key $(e_i)] \in V_p(R_{e_i})$的子节点 $e_i \in$Ce 的数量。当且仅当 count$[t'] =$ |C$_e$| 时，元组 t' 才会出现在 $V_s(R_e)$ 中。算法如算法 3 所示，显而易见。此外，我们还需要在 R_e 上建立一个哈希索引（该索引需支持将 $e \cap e_i$ 作为每个 $e_i \in$Ce 的键），以便在 $O(1)$ 时间内完成每次计数器变更。不过，与 S-更新不同，P-更新可能需要超过常数时间，因为多个元组的计数器可能会发生变化。事实上，在我们的查询计划中，变更传播期间更新时间增加的唯一情况就在这里。

R-更新。最后一种情况是输入关系 R_e 发生更新，此时我们也需要根据公式 (3) 更新 $V_s(R_e)$。我们将其称为 R-更新。算法 4 给出了详细过程，它以直接明了的方式维护 R_e 中的计数器，然后维护 $V_s(R_e)$中的计数器。显然，R-更新需要 $O(1)$ 时间（同时使用 $V_p(R_{e_i})$上的哈希索引）。

示例 *4.3*。 图 3(b)展示了将 $(1,\ 1)$插入 R_1后的索引。这个新元组首先触发向 $V_p(R_1)$插入，进而递增 $V_s(R_2)$ 中三个元组的计数器，这些元组随后被带入 $V_s(R_2)$。从这里开始，传播分成三条路径。元组 $(1,\ 2) \in R_2$递增了 $(1) \in V_p(R_2)$ 的计数器，但这条传播路径到此为止。元组$(1,\ 1) \in V_s(R_2)$首先向 $V_p(R_2)$插入一个新元组 (1)，随后递增根节点中元组 (1) 的计数器，使其达到

算法 3： P-更新$(e,\ t)$

输入： 对某个 $e_i \in$Ce，将 t 从 $V_p(R_{e_i})$ 更新;
输出： 已更新 $V_s(R_e)$;
如果 t 是对 $V_p(R_{e_i})$ 的插入，**则**
  对于每个 $t' \in R_e$满足 $t'[\text{键}(e_i)] = t$，**执行**
    计数$[t'] \leftarrow$计数$[t'] + 1$;
    如果 计数$[t'] =$ |C$_e$| **则**
      $V_s(R_e) \leftarrow V_s(R_e) \cup \{t'\}$, S-更新$(e,\ t')$;
否则
  对于每个 $t' \in R_e$使用 $t'[\text{键}(e_i)] = t$**执行**
    计数$[t'] \leftarrow$计数$[t'] - 1$;
    如果计数$[t'] = |C_e| - 1$ **那么**
      $V_s(R_e) \leftarrow V_s(R_e) - \{t'\}$, S-更新$(e,\ t')$;

算法 4： R-更新$(e,\ t)$

输入： 来自输入关系 R_e 的更新 t;
输出： 已更新的 $V_s(R_e)$;
如果 t 是对 R的插入 e **then**
  计数$[t] \leftarrow 0$;
  对于每个 $e_i \in Ce$ **执行**
    如果 $t[\text{键}(e_i)] \in V_p(e_i)$**则**
      计数$[t] \leftarrow$计数$[t] + 1$;
  如果 计数$[t] =$ |C$_e$| **则**
    $V_s(R_e) \leftarrow V_s(R_e) \cup \{t\}$, S-更新$(e,\ t)$;
否则
  如果 计数$[t] =$ |C$_e$| **则**
    $V_s(R_e) \leftarrow V_s(R_e) - \{t\}$, S-更新$(e,\ t)$;

$V_s([x_3])$。元组 $(1,\ 4) \in R_2$递增了 $(4) \in V_p(R_2)$ 的计数器，传播随之停止。

图 3(c) 展示了删除 $(1,\ 1)$ from R_4 后的索引。此次删除首先将元组 $(1) \in V_p(R_4)$的计数器减一，并将其从 $V_p(R_4)$中移除；随后又将 $(1,\ 1) \in V_s(R_3)$的计数器减一，并将其从 $V_s(R_3)$ 中移除。最后，$(1) \in V_p(R_3)$ 的计数器从 2 减至 1，传播到此为止。

引理 4.4。 查询计划中的所有投影视图和半连接视图都可以在 $O(|D|)$ 时间内更新。

5 枚举

5.1 完全结果枚举

我们首先考虑如何从查询计划中对 $Q(D)$ 执行常数延迟枚举。我们需要以下引理:

Lemma 5.1. *For any node e,* $V_s(R_e) = \pi_e(\bowtie_{e' \in \mathcal{T}_e} R_{e'})$.

引理 5.1 的证明。我们通过对自由连通连接树 T 的高度进行归纳来证明。首先，对于任意叶节点 e，结论成立，因为 $V_s(R_e) = R_e$。接下来考虑任意内部节点 e。令 $C_e = \{e_1, e_2, \cdots, e_k\}$为节点 e 的子节点集合。根据假设，我们假设此引理对每个 $e_i \in C_e$ 都成立，即

$$V_s(R_{e_i}) = \pi_{e_i}(\bowtie_{e' \in \mathcal{T}_{e_i}} R_{e'})$$

算法 5： FullEnum($\mathcal{T}$ e, t)

输入： 一个带有连通子树 Econ 的自由连通连接树 T，一个节点 $e \in \mathrm{T}$ 和一个键 $t \in \pi_{\mathrm{key}(e)} R_e$；

输出： T 上可与 t连接的查询结果；

1 **如果** $e \notin \mathcal{E}con$ **则**
2 产出 $\langle\rangle$
3 **否则**
4 设 $\mathrm{C}_e = \{e_1, \ e_2, \ \cdots, \ e_k\}$;
5 **对于每个** $t_0 \in \pi_{\mathrm{y}\cap e} V_s(R_e)$ 满足条件 $\pi_{key(e)} t_0 = t$ **执行**
6 **对于每个** $t_1 \in \textsc{FullEnum}(\mathcal{T}_{e_1}, e_1, t_0[key(e_1)])$ **执行**
7 **对于每个** $t_2 \in \textsc{FullEnum}(\mathcal{T}_{e_2}, e_2, t_0[key(e_2)]$ **执行**
8 $\cdots$
9 **对于每个** $t_k \in$ FullEnum($\mathrm{T}_{e_k}, \ e_k, \ t_0[\text{键}(e_k)]$) **执行**
10 输出 $t_0 \bowtie t_1 \bowtie t_2 \bowtie \cdots \bowtie t_k$;

那么，我们可以将 $V_s(R_e)$ 重写如下：

$$\begin{aligned}
&\Leftrightarrow R_e \ltimes V_p(R_{e_1}) \ltimes \cdots \ltimes V_p(R_{e_k}) \\
&\Leftrightarrow R_e \bowtie V_p(R_{e_1}) \bowtie \cdots \bowtie V_p(R_{e_k}) \\
&\Leftrightarrow R_e \bowtie \left(\pi_{\mathrm{key}(e_1)} V_s(R_{e_1})\right) \bowtie \cdots \bowtie \left(\pi_{\mathrm{key}(e_k)} V_s(R_{e_k})\right) \\
&\Leftrightarrow R_e \bowtie \left(\pi_{\mathrm{key}(e_1)} (\bowtie_{e' \in \mathcal{T}_{e_1}} R_{e'})\right) \bowtie \cdots \bowtie \left(\pi_{\mathrm{key}(e_k)} (\bowtie_{e' \in \mathcal{T}_{e_n}} R_{e'})\right) \\
&\Leftrightarrow \pi_e (R \bowtie_{e' \in \mathcal{T}_e} R_{e'}))
\end{aligned}$$

其中，第一个等式由半连接视图的定义得出，第二个等式由键 $(e_i) = e_i \cap e \subseteq e$这一事实得出，第三个等式由投影视图的定义得出，第四个等式由假设得出，最后一个等式则由以下事实得出：$\mathrm{T}_e = \{e\} \cup \mathrm{T}_{e_1} \cup \mathrm{T}_{e_2} \cup \cdots \cup \mathrm{T}_{e_k}$，并且键$(e_i)$ 恰好是 R_e 与 T_{e_i}中的任意关系所共有的连接属性集合。□

通俗地说，节点 e 的半连接视图实质上是将以 e为根的子树中各关系的连接结果投影到属性 e上。由此立即可得

Corollary 5.2. $V_s(R_r) = \pi_r Q(D)$.

这意味着，根节点 r 处的半连接视图（请回想一下，r没有投影视图）恰好包含所有投影到 r上的查询结果。利用见证查询的概念，可以得到以下关于完整枚举的有用事实，其中 $\biguplus$ 表示不交并：

Lemma 5.3. $Q(D) = \biguplus_{t \in V_s(R_r)} Q(D \ltimes t)$.

引理 5.1、推论 5.2 和引理 5.3 使我们能够使用与 Bagan 等人基本相同的算法 [5]，以实现对 $Q(D)$的常数延迟枚举；见算法 5。该算法以节点 $e \in \mathrm{T}$ 和键 $t \in \pi_{\text{键}(e)} R_e$为输入，并生成 T_e 上可与 t连接的查询结果。要枚举 $Q(D)$，我们只需调用 FullEnum(T r, $\cdot$)。

引理 5.4。*算法 5 以 $Q(D)$ 为 $O(1)$ 延迟进行枚举。*

证明。我们通过对 T 的高度进行归纳法来证明。若 $r \notin \mathrm{Econ}$，算法 5 就会停止；这种情况只会在 $\mathrm{y} = \varnothing$ 时发生。为便于

算法 6： DeltaEnum($\mathcal{T}$ t)

输入： 一棵自由连通连接树 T；一个已更新元组 t_o。

输出： 由 t引起的增量结果。

1 令 $e_0, \ e_1, \ \cdots, \ e_k = r$ 为 t 的传播路径上的节点；
2 **对每个** 见证元组 t'，取自 t **执行**
3 令 e_i为满足 $t' \in \pi_{\mathrm{y}\cap e_i} \Delta V_s(R_{e_i}, \ t)$的节点；
4 $S \leftarrow t' \bowtie V_l(R_{e_{i+1}}) \bowtie \cdots \bowtie V_l(R_{e_k})$;
5 **对每个** $q \in S$ **执行**
6 $S_i \leftarrow$FullEnum($\mathrm{T}_{e_i}, \ e_i, \ q[\text{键}(e_i)]$);
7 $S_j \leftarrow$FullEnum($\mathrm{T}_{e_j} - \mathrm{T}_{e_{j-1}}, \ e_j, \ q[\text{键}(e_j)]$), $j \in [i+1, \ k]$;
8 产出 $S_i \bowtie S_{i+1} \bowtie \cdots \bowtie S_k$;

表述，假设从 T 中移除任意满足 $e$with $e \notin$ Econ 的节点。我们首先确立一个基础情形，其中 T 只包含一个节点。算法会在 $O(|\pi_{\mathrm{y}\cap e} V_s(R_e)|)$ 时间内返回 $\pi_{\mathrm{y}\cap e} V_s(R_e)$，因为 $\pi_{\mathrm{y}\cap e} V_s(R_e)$ 中的所有元组都能在 $O(1)$ 延迟内枚举出来。因此，这个基础情形可以在 $O(1)$ 延迟内处理。

一般而言，我们假设所有子节点 e_i 都满足e （第 6–9 行）：只要 e_i属于连通子树，使 $\mathrm{key}(e_i) \subseteq e \cap \mathrm{y}$，算法 5 就能枚举出子树 T_{e_i} 中属性 $\mathrm{key}(e_i)$ 上与值 $\pi_{\mathrm{key}(e_i)} t$ 一致的所有连接结果，其中 $\mathrm{key}(e_i) \subseteq \mathrm{y}$。设 t_i是从 T_{e_i}返回的一个连接结果。根据连通子树的性质，第 10 行会返回有效的连接结果。在 e的所有子树上枚举连接结果的所有组合，只需花费 $O(1)$ 时间。□

5.2 增量枚举

在标准查询计划中，增量枚举很直接，因为根节点对应于 $Q(D)$，所以传播到根节点的所有变化恰好就是 $\Delta Q(D, \ t)$。然而，在我们的新查询计划中，情况变得棘手，因为没有节点对应于 $Q(D)$；若要实现 $Q(D)$ 的线性规模表示，这种情况是必然的。在我们的查询计划中，不能只检查根节点，因为并非每个变化都会传播到根节点，而且许多传播会中途停止，这实际上正是我们的查询计划不仅空间高效，而且时间高效的主要原因。回顾一下，完整枚举算法依赖于引理 5.3。那么关键问题是，我们能否为增量 $\Delta Q(D, \ t)$找到引理 5.3 的类似结论？换言之，我们能否为 t确定一组见证元组 t'，使得增量 $\Delta Q(D, \ t)$ 是 $Q(D \ltimes t')$的不交并？幸运的是，答案是肯定的，但这一结论并不像引理 5.3 那么简单。

我们先考虑插入情形。当我们将 t 插入某个R_e时，传播会沿着从 e到 r的路径进行，可能先应用 R-更新，然后依次应用 S-更新、P-更新、S-更新、P-更新，.....。请回想，S-更新和 R-更新都只会向上传播一项更改（见算法 2 第 8、12 行和算法 4），但 P-更新可能会向上传播多项更改（见算法 2 第 6、12 行）。因此，可能存在多条从 t开始的传播路径。更准确地说，我们将从 e到 r 路径上的节点记为 $e_0 = e$、e_1、e_2、$\cdots$、$e_k = r$。每条传播路径都会向路径上的每个视图插入一个元组，我们将这条路径上插入的元组记为 (t、t_0^s、t_0^p、t_1^s、t_1^p、$\cdots$、)，其中对于 $i \in \{0$、1、2、$\cdots$、$k\}$，$t_i^s \in V_s(R_i)$和 $t_i^p \in V_p(R_i)$。

现在，我们根据传播路径的终止元组区分三种情形：(1) t；(2) 对于某个 $j \in \{0、1、2、\cdots、k\}$，t_j^p；(3) 对于某个 $i \in \{0、1、2、\cdots、k\}$，t_i^s。

情况（1）发生在第一次更新是 R-更新，且不再传播任何进一步的变化时。这意味着在算法 4 中，存在某个子节点 e'，使得 $e\ t[\text{key}(e')] \notin V_p(R_{e'})$，即 t 无法与 T_e 连接。在这种情况下，t 不会对 $Q(D)$ 产生任何变化，因此可以忽略。

情况（2）发生在 P-更新$(e_j,\ t_j^p)$不再传播任何进一步的变化时。将其代入算法 3，这意味着不存在可以与 t_j^p 连接的元组 $t' \in R_{p_{e_j}}$；或者，如果存在这样的元组，但由于其计数器小于 $|\mathrm{C}_{p_{e_j}}|$，它无法与某个子节点 e' 的 p_{e_j} 在 T_e' 上的任何查询结果连接（如果 $R_{p_{e_j}}$ 是广义关系，则为相交）。在这两种情况下，这条传播路径都不会对 $Q(D)$ 造成任何变化，因此也可以忽略。

情况（3）发生在 S-更新$(e_i,\ t_i^s)$ 不再传播任何进一步的变化时。将其代入算法 2，这意味着我们要么已经到达根节点，要么存在某个其他元组 $V_p(R_i)$ $t' \neq t_i^s$ $t_i^s[\ \ (e_i)] = t'[\ \ (e_i)]$，使得 and key key。请注意，$R_i$ 可以是输入关系，也可以是广义关系。这是唯一一种对 $Q(D)$ 的更改可能发生的情况。我们稍后将对此情况给出更详细的刻画。

实时视图。为了支持常数延迟增量枚举，我们为每个节点 $e \in$ Econ 维护一个实时视图：

$$V_l(R_e) := \pi_{\mathbf{y} \cap e} Q(D),$$

这些是投影到 e 上的“实时”元组（即出现在查询结果中的元组）。请注意，$V_l(R_e) \subseteq \pi_{\mathbf{y} \cap e} V_s(R_e)$，这意味着对于 $e \subseteq \mathbf{y}$，只需在 $V_s(R_e)$ 中额外添加一位，指示相应的元组是否位于 $V_l(R_e)$ 中，即可实现。

对于根节点 r，无需单独维护 $V_l(R_r)$，因为根据推论 5.2，$V_l(R_r) = V_s(R_r)$。叶节点的实时视图也无需维护，因为增量枚举不会用到它们。其他实时视图可以根据以下观察来维护：

引理 5.5. 对于任意非根节点 e，满足 $e \in \mathcal{E}_{con}$，以及任意元组 $t \in \pi_{\mathbf{y} \cap e} V_s(R_e)$，$t \in V_l(R_e)$ 当且仅当 $t \ltimes V_l(R_{p_e}) \neq \emptyset$。

根据引理 5.5，维护 $V_l(R_e)$ 可以搭增量枚举的便车：枚举出结果 $t' \in \Delta Q(D,\ t)$ 后，我们以自顶向下的方式更新实时视图。对于每个满足 $e \cap \mathbf{y} \neq \emptyset$ 的非根 e，如果此次更新是插入，则总是将 $t'[e]$ 添加到 $V_l(R_e)$；如果此次更新是删除，则当 $t'[e]$ 无法与 $V_l(R_{p_e})$ 连接时，从 $V_l(R_e)$ 中删除 $t'[e]$。借助 $V_l(R_{p_e})$ 上的哈希索引，这一步可以在 $O(1)$ 时间内完成（该哈希索引在物理上与 $e \subseteq \mathbf{y}$ 上的 $V_s(R_{p_e})$ 哈希索引相同）。这只会使增量枚举的延迟增加一个常数。

见证元组。现在，我们可以更精确地刻画情况（3）中的终止元组，确定哪些元组实际上会导致 $Q(D)$ 发生变化；我们称之为见证元组：

定义 *5.6*（见证元组）。假设将 t 插入 D 或从 D 中删除。若元组 t' 是 t 的见证，则

$$t' \in \Delta V_s(R_r, t), \text{ or} \tag{5}$$

$$t' \in \pi_{\mathbf{y} \cap e} \Delta V_s(R_e, t) \ltimes V_l(R_{p_e}) \tag{6}$$

对于某个非根节点 e，使得 $e \in$ Econ。

这里，$\Delta V_s(R_e,\ t)$ 表示由于 t 而要插入 $V_s(R_e)$ 或从中删除的元组，$V_l(R_{p_e})$ 是更新前的当前视图。下面介绍定义 5.6 背后的一些直观解释。首先，(5) 是推论 5.2 在增量枚举中的对应形式，并且可以保证这样的 t' 会产生对 $Q(D)$ 的更改。(6) 是增量枚举特有的，用于处理前面提到的情况：传播在中途停止，却仍然导致 $Q(D)$ 发生更改。请注意，在这种情况下，t' 的属性为 $e \cap \mathbf{y}$。于是，由 (6) 可知 $t' \in \pi_{\mathbf{y} \cap e} \Delta V_s(R_e,\ t)$，且 $t'[\text{key}(e)] \in \pi_{\text{key}(e)} V_l(R_{p_e})$。由于 $t'[\text{key}(e)] \in \pi_{\text{key}(e)} V_l(R_{p_e})$，它必定具有 $t'[\text{key}(e)] \in V_p(R_e)$，即 $t'[\text{key}(e)] \notin \Delta V_p(R_e,\ t)$，这意味着在情况（3）下，传播会在节点 e 处停止。此外，每个见证元组 t' 都应当 (i) 对由 t 引起的 T_e 上的增量有所贡献，以及 (ii) 与 $\mathrm{T} - \mathrm{T}_e$ 中其余关系的元组连接。对于 (i)，只需要求 $t' \in \Delta\left(\pi_{\mathbf{y} \cap e} V_s(R_e)\right) = \pi_{\mathbf{y} \cap e} \Delta V_s(R_e,\ t)$，因为 $\Delta\left(\pi_{e \cap \mathbf{y}}\left(\bowtie_{e' \in \mathrm{T}_e} R_{e'}\right)\right) = \Delta\left(\pi_{\mathbf{y} \cap e} V_s(R_e)\right)$。对于 (ii)，只需要求 $t' \ltimes V_l(R_{p_e}) \neq \emptyset$；这正是我们最初引入 $V_l(R_e)$ 的原因。

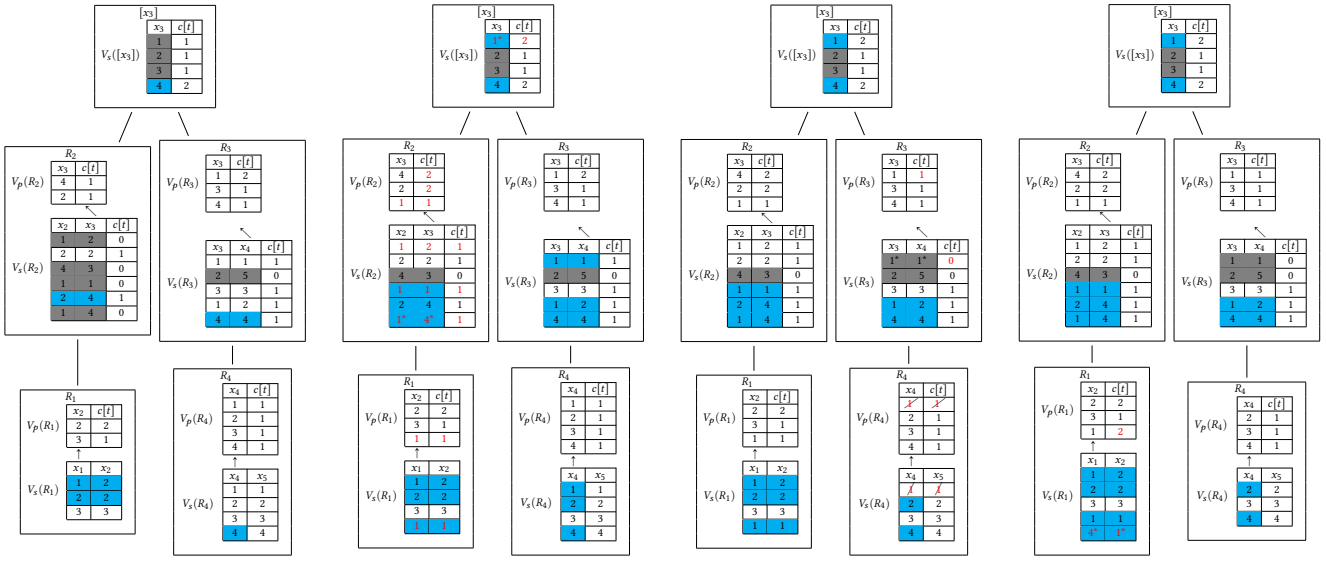
Lemma 5.7. $\Delta Q(D, t) = \biguplus_{t' : a\ witness\ of\ t} Q(D \ltimes t')$.

现在，我们可以在引理 5.7 中陈述引理 5.3 关于增量枚举的对应结论。与引理 5.3 不同，引理 5.7 的证明并不简单，具体细节见附录 B。

该算法。要使用引理 5.7 执行增量枚举，我们还需要解决两个问题：（1）如何找到所有见证元组 t'；（2）如何以常数延迟枚举 $Q(D \ltimes t')$。

为了找到所有见证元组，我们考虑定义 5.6 中的两种情况：(5) 在更新 $V_s(R_r)$ 后可以轻松计算；对于 (6)，只需额外检查 $V_l(R_{p_e})$，利用 $V_l(R_{p_e})$ 上的哈希索引即可在 $O(1)$ 时间内完成。这些步骤只会使更新成本增加一个常数因子。

接下来描述如何为每个见证 t' 枚举 $Q(D \ltimes t')$。与之前一样，令 $e_0, e_1, \ldots, e_k = r$ 为传播路径上的节点，并假设给定了某个 的见证元组 $t' \in$ π i y $\cap_{e_i} \Delta V_s(R_{e_i}, t)$。我们首先枚举由 t' 参与的查询结果，以及从 e_{i+1} 到根节点 r 路径上的关系，将其记为 S。这可以通过将 t 与这些节点关联的实时视图进行连接来完成。对于每个这样的结果 $q \in S$，我们枚举由 q 参与的查询结果。具体做法是将整个自由连通连接树划分为互不相交的子树 $\mathrm{T}_{e_i}, \mathrm{T}_{e_{i+1}} - \mathrm{T}_{e_i}, \cdots, \mathrm{T}_{e_k} - \mathrm{T}_{e_{k-1}}$，并分别对每棵子树调用 FullEnum。最后，我们将这些子树连接起来。详细过程见算法 6。请注意，按当前写法，该算法无法实现常数延迟枚举。不过，这很容易修正。首先，第 4 行中的连接可以从 t' 开始，使用 FullEnum（或其变体）以常数延迟进行枚举。然后，我们交错执行这两个枚举过程：每枚举出一个 $q \in S$，就立即调用第 6–8 行。最后，可以将第 6–8 行改写为嵌套循环，从而以常数延迟枚举连接 $S_i \bowtie \cdots \bowtie S_k$。实际上，这个连接更像是一个笛卡尔积（公共属性必须取相同的值，与 q 中的值相同），总共会产生 $\Pi_{j=i}^{k} |S_j|$ 个结果。



(a) 初始化。 (b) 将 $(1,1)$ 插入 R_1 后。 (c) 从 R_4 删除 $(1,1)$ 后。 (d) 将 $(4,1)$ 插入 R_1 后。

图3：使用图 1(c) 中的计划，对图 1 中的查询进行运行实例演示。白色元组属于 $V_s(R)$，灰色元组属于 $R \backslash V_s(R)$，青色元组属于 $V_l(R)$（叶节点的实时视图并非必需，但为清晰起见仍予以展示），带星形符号的是见证元组。每一步的变更均以红色标出。

Example 5.8. 我n figure 3(a), th这里是 two 查询结果lts $(1,2,4,4)$ 以及 $(2、2、4、4)$。在图 3(b) 中，当传播停止时，我们有 • 元组 $(12) \in R_2$ 不是见证，因为它无法与 $V_l([x_3])$ 中的任何元组连接，因此不会产生增量；• 元组 $(1) \in [x_3]$ (是见证，会触发增量枚举。对于根节点中的见证，DeltaEnum 会直接退化为 FullEnum(T$_r$ r (1))，其输出为 $\{(1、1、1、1)(1、1、1、2)\}$。• 元组 $(14) \in R_2$ 是见证，会触发增量枚举。DeltaEnum 找到 $S = (1$ $4) \bowtie V_l([x_3]) = \{(14)\}$。对于 $(14) \in S$，它会调用 FullEnum(T $_r$ $-$ T^{R_2} r (4))，返回 $\{(44)\}$；并调用 FullEnum(T^{R_2} R_2 (14))，返回 $\{(1、1、4)\}$。将它们连接后得到增量 $\{(1、1、4、4)\}$。最后，每当枚举出一个新结果，我们都会更新实时视图。在图 3(c) 中，元组 $(11) \in \Delta V_s(R_3)$ 是见证。DeltaEnum 首先找到 $S = \{(11)\}$。对于 $(11) \in S$，它会调用 FullEnum(T$_r$ $-$T^{R_3} r (1))，返回 $\{(1、1、1)\}$；并调用 FullEnum(T^{R_3} R_3 (11))，返回 $\{(1、1、4)\}$（删除时的增量枚举会在删除元组之前执行，以便找到增量）。将它们连接后得到增量 $\{(1、1、1、1)\}$。最后，我们用该增量更新实时视图。

引理 5.9。 算法 *6* 以常数延迟枚举 $\Delta Q(D、t)$。

我们现在已经闭合了这一循环：在枚举 $\Delta Q(D、t)$ 时，我们按照前文所述更新实时视图，而这些视图是枚举下一个增量所必需的。

6 更新成本分析

我们已经证明，完全查询结果和增量的枚举延迟都是常数；对于第 4.1 节中所述由任意自由连通连接树定义的查询计划，这一结论同样成立。另一方面，不同查询计划的更新成本各不相同，最坏情况下可能达到 $O(|D|)$。这是由 P-更新造成的，它可能会触发对其父节点中每个元组执行 S-更新。不过，这种最坏情况只会出现在刻意构造的更新序列中，实际更新成本可能低得多。刻画更新成本对于构造良好的查询计划至关重要，因为对于给定的自由连通查询，可能存在许多自由连通连接树。正如我们将看到的，连接树的高度是一个重要参数；这正是我们让框架适用于任意自由连通连接树的原因，因为自由连通连接树的高度可能低于任何标准连接树的高度。例如，图2中的查询有一棵高度为 1 的自由连通连接树，而两棵标准连接树的高度均为 2；图 1 中的查询有一棵高度为 2 的自由连通连接树，而任何标准连接树的高度至少为 3。

6.1 包围度

更新序列与生命周期。 给定一个更新序列 S_D，元组 t 的生命周期是一个区间 $I(t) = [t^+、t^-]$，其中 t^+ 表示将 t 插入 D 的时间戳，而 t^- 表示从 D 删除 t 的时间戳。我们将 $t^+ = -\infty$ 设为，以表示 t 存在于初始 D 中；$t^- = +\infty$ 表示在该更新序列之后，t 仍然存在于 D 中。注意，如果一个元组被反复插入和删除，它将被视为多个元组，这些元组具有相同的值，但生命周期彼此不相交。

尽管我们的算法能够处理任意更新序列，但如果更新序列具备某些良好性质，其性能会更好。具体而言，以下两类限制性更新序列在实践中具有重要意义：

• **先进先出（FIFO）**。如果对于任意两个元组 $t_1, t_2 \in S_D$，$t_1^+ < t_2^+$ 蕴含 $t_1^- < t_2^-$，则更新序列 S_D 为先进先出。先进先出序列在实践中十分常见，例如流数据上的滑动窗口或滚动窗口模型。

• **仅插入或仅删除**。如果对于任意元组 $t \in S_D$ $t^- = -\infty$，S_D 是仅插入的（相对于仅删除），则称更新序列 S_D 是仅插入的（相对于仅删除）。这两种情况是对称的，因此本文仅讨论仅插入的情况。

包围度 这一概念最初由 [36] 引入，用于针对具体实例刻画更新序列的困难性，下面我们将简要回顾这一概念。

定义 *6.1*（包围度）。给定一个更新序列 S_D，元组 $t \in S_D$ 的包围度为

$$\lambda(t) := \max_{\substack{\mathcal{J} \subseteq S_D \\ \forall t_1 \in \mathcal{J}, I(t_1) \subset I(t) \\ \forall t_2, t_3 \in \mathcal{J}, I(t_2) \cap I(t_3) = \varnothing}} |\mathcal{J}|, \tag{7}$$

即，S_D 中包含于 $I(t)$ 的互不相交生命周期的最大数量。然后，更新序列的包围度是所有元组包围度的平均值（但至少为 1），即

$$\lambda(S_D) := \max\left(\frac{\sum_{t \in S_D} \lambda(t)}{|S_D|}, 1\right). \tag{8}$$

我们通常省略 S_D，直接将更新序列的包围度写作 $\lambda := \lambda(S_D)$。

接下来，他们给出了一种算法，可以在 $O(\lambda)$ 时间内更新任意外键无环查询，适用于任意 S_D，同时支持 $O(1)$延迟枚举。这很有吸引力，因为尽管在最坏情况下，λ 可能与$O(|S_D|)$ 一样大，但对于许多常见的更新序列（包括先进先出、先进后出（FILO）以及仅插入／仅删除序列），它通常是一个较小的常数。只有当许多生命周期较长的元组与许多生命周期较短的元组连接时，才会出现最坏情况；而这在实践中并不常见（也就是说，查询发生了许多幅度很大但持续时间很短的变化）。

然而，他们的分析关键依赖于外键无环查询的一个良好性质：其结果大小至多为线性，而非键连接则不具备这一性质。事实上，我们将在下文证明，对于自由连通查询，$O(\lambda)$ 更新时间是无法实现的；这一结论可由我们将在下文证明的否定性结果推出：

定理 6.2 $Q = R_1(x_1) \bowtie R_2(x_1\ x_2) \bowtie .. R_3(x_2\ x_3) \bowtie R_4(x_3\ x_4) \bowtie R_5(x_4)$ </style} 考虑 *FIFO* 更新序列上的查询，，。如果存在一种算法，对 Q 的更新时间为 $O(|D|^{1/2-\epsilon})$，同时支持对任意常数 $\epsilon > 0$ 以 $O(|D|^{1-\epsilon})$ 延迟枚举完整结果，那么 *OuMv* 猜想[3] 将不成立。

请注意，该定理区分了（至少一种）自由连通查询与外键无环查询的难度；对于后者，FIFO 序列可以实现 $O(1)$ 更新时间 [36]。

[3]OuMv 猜想 [19]认为，对于任意常数 $\epsilon > 0$，以下问题无法在 $O(n^{3-\epsilon})$ 时间内解决：给定一个 $n \times n$ 矩阵 M和一个由 n维向量 u_1、v_1、u_2、v_2、$\cdots$、u_n、v_n组成的序列，对布尔半环上的每个 i 计算 $u_i M v_i$。算法必须在揭示 u_{i+1}、v_{i+1}之前返回 $u_i M v_i$。

6.2 连接树特定包络度

尽管上述否定结果令人失望，但希望并未完全破灭。首先，定理 6.2 只适用于某个特定的自由连通查询；其他查询仍可能在 $O(1)$ 时间内得到更新。其次，[36] 中的包络性定义只考虑时间维度，而忽略了结构维度，也就是说，没有考虑每次更新作用于哪个关系。这些观察促使我们提出一个更精细的包络性定义，该定义还取决于连接树（即更新作用于哪些节点）。正如我们将看到的，像定理 6.2 中那样困难的查询，在同时考虑结构维度和时间维度的信息后，仍然可以高效求解。

定义 *6.3*（输入关系中元组的有效生命周期）。给定一个自由连通查询 Q、Q的自由连通连接树 T、数据库 D以及更新序列 S_D，输入元组 $t_1 \in R_e$的生命周期为 $I(t_1) = [t_1^+, t_1^-]$ 时，其两个有效生命周期为

$$\widehat{I}(t_1) = \left[t_1^+, \min\left(t_1^-, \min_{t_2 \in R_{e'}: e' \in \mathcal{T}_e - \{e\}, t_2^- > t_1^+} t_2^-\right)\right];$$

$$\check{I}(t_1) = \left[\max\left(t_1^+, \max_{t_2 \in R_{e'}: e' \in \mathcal{T}_e - \{e\}, t_2^+ < t_1^-} t_2^+\right), t_1^-\right].$$

其中，最小值和最大值分别取遍所有位于子树 T$_e$中的任意输入关系 $R_{e'}$ 的元组 t_2。

通俗地说，$\widehat{I}(t_1)$ 是通过将 $I(t_1)$ 的结束时间向后移至 e的任意后代中的输入元组首次被删除的时间而得到的；而要得到 $\check{I}(t_1)$，我们将其开始时间移至 e的任意后代中的输入元组最后一次插入的时间。

现在可以定义元组的连接树特定包络度：

定义 *6.4*。给定一个自由连通查询 Q、Q的自由连通连接树 T、数据库 D以及更新序列 S_D，对于节点$e \in$ T和元组 $t \in R_e$，其包围度为

$$\lambda_{\mathcal{T}}(t) = \max_{\substack{\forall t' \in \mathcal{J}, \exists e' \in \mathcal{T}_e - \{e\}, t' \in R_{e'} \\ \forall t_1 \in \mathcal{J}, \dot{I}(t_1) \subseteq I(t) \\ \forall t_2, t_3 \in \mathcal{J}, \dot{I}(t_2) \cap \dot{I}(t_3) = \varnothing}} |\mathcal{J}|, \tag{9}$$

其中每个 $\dot{I}$ 要么是 $\widehat{I}$，要么是 $\check{I}$，也就是说，是 e的后代中包含在 t的生命周期内的元组的互不相交有效生命周期的最大数量。此时，更新序列的包围度仍为平均值：

$$\lambda_{\mathcal{T}}(S_D) := \max\left(\frac{\sum_{t \in S_D} \lambda_{\mathcal{T}}(t)}{|S_D|}, 1\right).$$

我们常用 $\lambda_{\mathcal{T}} := \lambda_{\mathcal{T}}(S_D)$ 表示更新序列相对于 T 的包围度。

例*6.5*。考虑图2中的 $Q := R_1(x_1,\ x_2) \bowtie R_2(x_2,\ x_3)$，其中包含 T$_1$、T$_2$、T$_3$。对于图 4 中的更新序列，$\lambda_{\mathcal{T}_1} = \lambda_{\mathcal{T}_2} = n$ 且 $\lambda_{\mathcal{T}_3} = 1$。事实上，对于任意更新序列，$\lambda_{\mathcal{T}_3} = 1$。

本文的主要解析结果是以下定理，其证明相当复杂，详见附录C：

定理 6.6。对于任意自由连通查询 Q，由任意给定的自由连通连接树$\mathcal{T}$所诱导的第*4*节中的查询计划，其更新成本为 Q *is*$O(\lambda_{\mathcal{T}})$，适用于包围度为 $\lambda_{\mathcal{T}}$的任意更新序列。

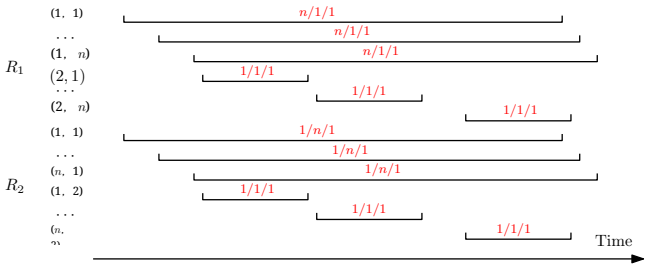


图4：例6.5中的更新序列。每个区间表示一个元组的生命周期，每个区间上方的三个数字分别表示图2中$\mathcal{T}_1$、$\mathcal{T}_2$ 和$\mathcal{T}_3$ 的包围度。

这一结果还有一个匹配的下界作为补充，至少适用于一个特定查询：$Q = \pi_{x_1}(R_1(x_1, x_2) \bowtie R_2(x_2))$；该查询有一棵如图2(a)所示的连接树（也可以在顶部添加一个广义关系 $[x_1]$，但这不会改变包围度）。因此，对于这个查询，$\lambda_{\mathcal{T}}$ 实际上并不依赖于 T。

定理 6.7。 *[36]* 假设存在一种算法，用于查询$Q = \pi_{x_1}(R_1(x_1, x_2) \bowtie R_2(x_2))$，其更新时间为 $O(\lambda^{1-\epsilon})$，同时支持以$O(\lambda^{1-\epsilon})$延迟枚举完整结果（对任意常数$\epsilon > 0$均成立），那么 *OMv* 猜想[4]不成立。

6.3 封闭性的影响

我们将介绍包围度（针对连接树）和定理 6.6 的一些影响，展示更新序列的难度与查询复杂度之间有趣的权衡。

任意更新序列。 对于任意更新序列，先前工作 [8, 20] 已展示如何实现 $O(1)$ 更新时间，同时支持以 $O(1)$延迟枚举任意 q-层次查询的结果。事实证明，这可以轻易地由定理 6.6、以下关于 q-层次查询的结构性质，以及一个简单事实推出：若 T 的高度为 1，则 $\lambda_{\mathcal{T}} = 1$：

引理 6.8。 每个 *q*-层次合取查询都有一棵高度为 *1* 的自由连通连接树。

对于任意更新序列，q-层次查询恰好是能够实现 $O(1)$ 更新时间的查询类别[8]。因此，对于不属于此类别的查询，我们必须限制更新序列，才能实现 $O(1)$ 更新时间。我们考虑以下两类更新序列。

先进先出序列。 研究表明，对于先进先出序列上的外键无环连接，更新时间为 $O(1)$ [36], ；但对于非键连接，目前尚无已知结果（不依赖先进先出序列的 q-层次查询除外）。我们给出在这一方向上的第一个扩展：

引理 6.9。 对于任意自由连通查询 Q，若其自由连通连接树 $\mathcal{T}$ 的高度至多为2，则 $\lambda_{\mathcal{T}} = 1$ 对于任意先进先出序列均成立。

请注意，高度限制为 2 已是所能期望的最佳结果，因为定理 6.2 中的查询具有高度为 3 的连接树，而该定理表明，它无法在 $O(1)$ 时间内针对先进先出序列进行更新。尽管高度为 2 的限制缩小了查询的类别，但这已经涵盖了一些相当复杂的查询，例如图 1 中的查询；更多示例见第 8 节。

仅插入序列。 进一步限制更新序列后，我们就能在 $O(1)$ 时间内处理更多查询。为简洁起见，以下结果只考虑仅插入序列，但同样的结果也适用于仅删除序列或 FILO 序列。

引理 6.10。 对于任意自由连通合取查询 Q和自由连通连接树 $\mathcal{T}$，$\lambda_{\mathcal{T}} = 1$ 适用于任意仅插入更新序列。

结合定理 6.6 和引理 6.10，下面的定理显而易见。

定理 6.11。 对于自由连通查询 Q，存在一个索引，在任意仅插入更新序列下均可在 $O(1)$ 摊销时间内更新，同时支持 $O(1)$-延迟枚举。

注意，引理 6.10 将静态结果 [5] 作为一个特例纳入其中。给定一个静态数据库 D，我们只需将 D 中的每个元组插入查询计划即可。根据引理 6.10，这会在 $O(|D|)$ 时间内构建一个数据结构，支持对$Q(D)$进行 $O(1)$-延迟枚举。此外，[5] 的二分结果指出，只有对于自由连通查询，才可能实现 $O(|D|)$时间预处理和 $O(1)$-延迟枚举。因此，即使是在仅插入序列下，引理 6.10 也无法扩展到自由连通查询之外。

示例 *6.12*。 考虑图 1 中查询的一个仅插入更新序列：(1) 最初将元组 $(i, j) \in [n] \times [n]$ 插入到 R_2、R_3 和 R_4 中；(2) 随后将元组 $(i, j) \in [n] \times [n]$ 插入到 R_1 中。标准变更传播或 HIVM 需要物化 $\Delta(R_1 \bowtie R_2 \bowtie R_3)$，因此会产生 $O(n^3)$ 的开销；动态 Yannakakis 算法 [20] 需要扫描 $(i', j') \in R_2$ 中的所有元组，以处理 $i' = j$；当 (i, j) 被插入到 R_1时，因此会产生 $\Theta(n)$ 的开销；而我们的框架只会产生 $O(1)$ 的开销。

查询计划优化。 如果给定的查询和/或更新序列不属于上述三种能够保证 $O(1)$ 更新时间的情形，那么我们的包围性分析仍能为选择合适的 T 提供有效的启发式方法，而这又决定了查询计划。首先，显然应始终优先选择高度较小的 T。此外，定义 6.4 表明，我们应将更新较多的节点放在 T 中较高的位置，因为某个节点中的元组可能会增加其祖先节点中元组的包围度。因此，在我们的实现中，我们构造所有连接树，并选取使 $\sum_{e \in \mathcal{T}} d(e)N(e)$ 最小的连接树，其中 $d(e)$ 是 e 的深度

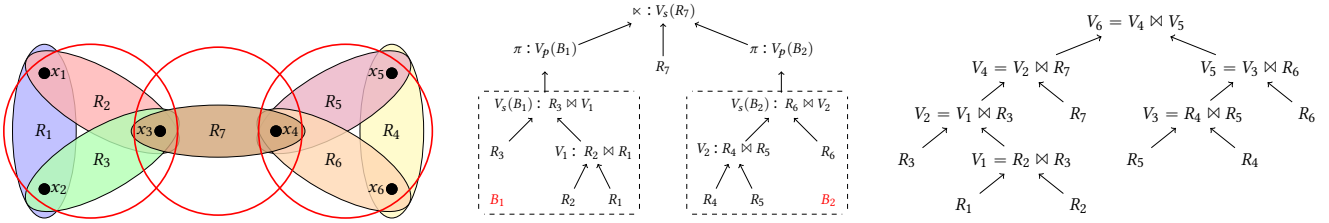
T 中（不计广义关系及其自身），而 $N(e)$ 是对 e的更新次数。如果无法获得 $N(e)$，我们可以通过观察（并缓冲）前几次更新来估算它。

7 对一般查询的扩展

7.1 一般合取查询

无环但非自由连通的查询。 考虑这样的查询 $\pi_{x_1, x_3}R_1(x_1, x_2) \bowtie R_2(x_2, x_3)$。我们只需添加 x_2 作为输出属性，将其转化为自由连通查询，然后在枚举过程中对 x_1，x_3 进行投影。请注意，枚举结果可能包含重复项。因此，如果显式声明了 DISTINCT 关键字，就必须删除重复项，这会使延迟超过常数；但由于下界 [5]，这是不可避免的。

[4]OMv 猜想与 OuMv 猜想类似，区别在于算法需要针对每个 v_i计算 Mv_i。



(a) 超图和广义超树分解　(b) 新的变更传播下的查询计划　(c) 标准变更传播下的查询计划

图 5：5(a) 是“哑铃”查询 $Q = R_1(x_1, x_2) \bowtie R_2(x_1, x_3) \bowtie R_3(x_2, x_3) \bowtie R_4(x_5, x_6) \bowtie R_5(x_4, x_5) \bowtie R_6(x_4, x_6) \bowtie R_7(x_3, x_4)$ 的超图，广义超树分解以红色圆圈标出。5(c)、5(b) 分别是标准变更传播框架和新的变更传播框架下的查询计划。在 5(c) 中，B_1, B_2 被视为两个基本关系，并在其上构造投影和半连接视图。

循环查询。 在我们的框架中，也可以通过采用广义超树分解（GHD）[17]轻松处理循环查询。更具体地说，通过将若干关系组合成一个包，可以将任意合取查询转化为自由连通查询。例如，图 5(a) 展示了“哑铃”查询的一种 GHD，其中包含 3 个包。

我们可以在每个包内使用标准变更传播，并在各个包之间应用我们的框架。由此得到图 5(b) 中的查询计划，其空间复杂度为 $O(N^2)$，更新时间为 $O(N)$，同时支持常数延迟枚举。另一方面，标准变更传播框架会采用类似图 5(c) 的查询计划，其空间复杂度为 $O(N^3)$，更新时间为……当然，以上都是最坏情况界；对于现实输入，开销会更低，但如第 8 节所示，我们的新查询计划仍比旧计划快一个数量级。

如果希望进一步改进理论界，可以用更好的算法替换用于维护每个包内查询结果的算法。例如，Kara 等人 [25] 提出了一种用于维护三角形连接的算法。替换为新算法可以将空间用量从 $O(N^2)$ 降低到 $O(N\sqrt{N})$。另一方面，尽管算法 [25] 可以将每个包的更新成本降至 $O(\sqrt{N})$，“哑铃”查询仍面临 $O(N)$ 的更新成本。这确实无法避免，因为单次元组更新最多可能改变某个包中已物化的 $O(N)$ 个查询结果，而这些变化还会进一步传播到整个框架。因此，这种基于广义超树分解的变更传播框架的更新成本，不仅取决于每个包内部的更新，也取决于包与包之间的更新。

除三角形连接外，目前已知的结果并不多。这仍是一个正在积极研究的问题；将这里的任何改进纳入我们的框架，也会改进一般合取查询。

定理 7.1. 给定一个具有宽度为[5]w的自由连通广义超树分解的合取查询 Q，存在一个大小为 $O(N^w)$ 的索引，可在 $\Omega(N^w)$ 时间内更新，同时支持 $O(1)$延迟枚举。

证明。维护任意一个关系包需要 $\Omega(N^w)$ 时间，并且需要 $O(N^w)$ 空间来存储包中的所有查询结果。

[5]宽度的定义取决于用于维护每个包内查询结果的算法。若采用标准变更传播框架，宽度定义为所有包的宽度最大值；一个包的宽度是为该包导出的相应子查询的最优整数边覆盖数。

维护完每个关系包后，可以使用第4节提出的算法来维护各个包之间的关系，维护所需时间为 $O(|D|)$。注意，当前数据库大小 $|D|$受最大包大小限制，该大小为 N^w，因此总维护时间为 $\Omega(N^w)$，空间开销为 $O(N^w)$。□

以下引理可由上述定理轻易推出。

引理 7.2. 对于“哑铃”查询，存在一个大小为 $O(N^{1.5})$的索引，每次元组更新可在 $O(N)$ 时间内完成，同时支持 $O(1)$延迟枚举。

7.2 选择、并集与集合差

第 4.1 节中的查询计划适用于包含连接和投影的合取查询，但也可以轻松地为其添加其他运算符。

- 如果在输入关系 R_e 上有选择操作 σ_ϕ，其中 ϕ 是关于 e的谓词，那么对于包含元组 $t \in R_e$的更新，我们只需检查 $\phi(t)$ 是否为真；如果不是，就丢弃这次更新。这只会给更新成本增加 $O(1)$ 的时间。
- 对于合取查询的并集 $Q = Q_1 \cup \cdots \cup Q_k$，我们将分别维护每个 Q_i。利用 [10]中的技术，可以支持延迟为 $O(1)$ 的完整枚举。请注意，[10] 假设每个 $Q_i(D)$ 上的数据结构都能在 $O(1)$时间内检查给定的 t是否属于 $t \in Q(D_i)$，而我们的查询计划确实支持这一点。对于增量枚举，我们可以使用相同的技术来枚举 $\Delta Q_1(D) \cup \cdots \cup \Delta Q_k(D)$。然而，这与 $\Delta(Q_1 \cup \cdots \cup Q_k)$并不相同，我们需要检查例如某个新结果 $t \in \Delta(Q_1)$ 是否已存在于 $Q_2(D)$中。因此，尽管 [10] 中的技术对于增量枚举仍然正确，但延迟无法保证为常数。如何支持 UCQ 的 $O(1)$延迟增量枚举，仍是一个有趣的开放问题。
- 对于类似 $Q = Q_1 - Q_2$ 这样的查询，我们可以像上面那样分别维护 Q_1 和 Q_2。对于枚举，我们枚举每个 $t \in Q_1(D)$，并检查是否满足 $t \in Q_2(D)$，尽管这无法保证常数延迟。事实上，即使在静态情形下，在线性时间预处理后，是否能以 $O(1)$延迟枚举 $Q_1(D) - Q_2(D)$ 仍是一个开放问题，其中 Q_1 和 Q_2 都自由连通。

7.3 聚合

标准关系代数可以扩展以支持聚合，我们采用以下形式化框架 [1, 23]。设 $(S, \oplus, \otimes)$ 为一个交换环[6]。每个元组 $t \in R_e$ 都有一个注释 $w(t) \in S$。对于形式为 (1) 的全合取查询 Q，任意连接结果 $t \in Q(D)$ 的注释定义为 $w(t) := \bigotimes_{e \in Q} w(t[e])$。对于非完全查询 $\pi_{\mathbf{y}} Q$ 后，投影变为分组 $\mathbf{y}$，并且每个结果 $t \in \pi_{\mathbf{y}} Q(D)$ 的注释（即每个组的聚合结果）为 $w(t) := \bigoplus_{t' \in Q(D) \,:\, \pi_{\mathbf{y}} t' = t} w(t')$。

我们的新变更传播框架可以轻松扩展以支持聚合。对于任意关系 e，令 $w(t)$ 表示元组 $t \in R_e$ 的注释，$w_s(t)$ 表示 $t \in V_s(R_e)$ 的注释，$w_p(t)$ 表示 $t \in V_p(R_e)$ 的注释。根据半连接视图 V_s 和投影视图 V_p 的定义（见第4节），$t \in V_s(R_e)$ 的注释定义为

$$w_s(t) := \left(\bigotimes_{e' \in C_e : e' \notin \mathcal{E}_{\text{con}}} w_p(\pi_{\text{key}(e')} t) \right) \otimes w(t). \tag{10}$$

$t \in V_p(R_e)$ 的注释定义为

$$w_p(t) := \bigoplus_{t' \in V_s(R_e) : \pi_{\text{key}(e)} t' = t} w_s(t'). \tag{11}$$

为简化枚举过程，我们为每个节点 $e \in \text{Econ}$ 定义一个额外的视图 $V_a(R_e) = \pi_{e \cap \mathbf{y}} V_s(R_e)$，其中任意元组 $t \in V_a(R_e)$ 的注释定义为

$$w_a(t) := \bigoplus_{t' \in V_s(R_e) : \pi_{e \cap \mathbf{y}} t' = t} w_s(t'). \tag{12}$$

此外，如果 $\mathbf{y} = \varnothing$，我们会在根节点 r 之上维护一个特殊注释 $w_a(r)$：

$$w_a(r) := \bigoplus_{t \in V_s(R_r)} w_s(t). \tag{13}$$

更新。我们将这些注释连同对应的计数器一起存储在查询计划中。然后，应根据上述公式修改变更传播和枚举过程。更准确地说，每当元组的计数器更新时，我们也会根据 (10)、(11)、(12) 和 (13) 更新其注释。

枚举。枚举分为两种情况。如果 $\mathbf{y} \neq \varnothing$，我们会为每个查询结果 $t \in Q$ 计算注释 $w(t)$，如下所示：

$$w(t) := \bigotimes_{e \in \mathcal{E}_{\text{con}}} w_a(\pi_{e \cap \mathbf{y}} t). \tag{14}$$

在 $O(1)$ 时间内完成，只要为每个节点 $e \in \text{Econ}$ 妥善维护 $V_a(\cdot)$。如果 $\mathbf{y} = \varnothing$，我们只需返回 $w_a(r)$。

引理 7.3。 如果 $\mathbf{y} = \varnothing$，$w_a(r)$ 就是最终聚合。

引理 7.4。 如果 $\mathbf{y} \neq \varnothing$，(14) 就是查询结果 $t \in Q$ 的注释。

下面，我们通过引理 7.3 和引理 7.4 证明其正确性。给定一个自由连通连接树 T，我们将节点 e 的非连通子树定义为 $\mathbf{X}_e$，它是 T 中以 e 为根的最大子树，且满足 $(\mathbf{X}_e - \{e\}) \cap \text{Econ} = \varnothing$。如果 $e \notin \text{Econ}$，则 $\mathbf{C}_e \subseteq \mathbf{X}_e$。显然，对于任意一对不同的节点 $e, e' \in \text{Econ}$，$\mathbf{X}_e$ 和 $\mathbf{X}_{e'}$ 必定是节点不相交的。我们进一步将 $\mathbf{Q}_f(e, t)$ 定义为连接结果 $\bowtie_{e' \in \mathcal{X}_e} e'$ 的集合，使得对于任意 $t_i \in \mathbf{Q}_f(e, t)$, $\pi_e t_i = t$。

[6]在静态情形下，$(S, \oplus, \otimes)$ 只需是一个半环，但我们需要加法逆元来支持删除。

引理 7.5。 给定一个自由连通查询 Q，其自由连通连接树为 $\mathcal{T}$，以及任意节点 $e \notin \mathcal{E}_{con}$，则对于任意元组 $t \in V_s(R_e)$

$$w_s(t) := \oplus_{t_i \in Q_f(e,t)} w(t_i).$$

引理 7.5 的证明。我们可以对 $\mathbf{X}_e$ 的大小使用归纳法来证明。当 $\mathbf{X}_e$ 只包含 e 时，结论显然成立，因为只有一个连接结果 t，且带有 $w_s(t) = w(t)$。一般情况下，我们假设该结论对 e 的每个子节点都成立。那么，

$$\begin{aligned} w_s(t) &= w(t) \otimes \left(\bigotimes_{e' \in C_e} \left(w_p(\pi_{\text{key}(e')} t) \right) \right) && \text{by (10)} \\ &= w(t) \otimes \left(\bigotimes_{e' \in C_e} \left(\bigoplus_{t_j \in Q_f(e', \pi_{\text{key}(e')} t)} w_s(t_j) \right) \right) && \text{by (11),} \end{aligned}$$

$\mathbf{C}_e$ 中任意一对子节点之间的连接都可以退化为笛卡尔积，因为对于任意 $e_1, e_2 \in \mathbf{C}_e$，以及任意 $t_1 \in \pi_{e_1} \mathbf{Q}_f(e, t)$ 和 $t_2 \in \pi_{e_2} \mathbf{Q}_f(e, t)$，要么 $e_1 \cap e_2 = \varnothing$，要么 $e_1 \cap e_2 \subseteq e$ 和 $\pi_{e_1 \cap e_2} t_1 = \pi_{e_1 \cap e_2} t_2 = \pi_{e_1 \cap e_2} t$。无论哪种情况，

$$Q_f(e, t) = t \bowtie (\times_{e' \in C_e} Q_f(e', \pi_{\text{key}(e')} t)).$$

我们进一步将 $t_i \in \mathbf{Q}_f(e, t)$ 分解为 $t_i := \bowtie_{e' \in \mathcal{X}_e} (\pi_{e'} t_i)$，那么

$$\oplus_{t_i \in Q_f(e,t)} w(t_i) = \oplus_{t_i \in Q_f(e,t)} \left(\otimes_{e' \in \mathcal{X}_e} w(\pi_{e'} t_i) \right)$$

结合 $w_s(t') := \oplus_{t_i' \in Q_f(e,\ t)} w(t_i')$ 对于每个 $e' \in \mathbf{C}_e$，我们可以进一步将 $w_s(t)$ 改写为

$$\begin{aligned} &= w(t) \otimes \left(\bigotimes_{e' \in C_e} \left(\bigoplus_{t_j \in Q_f(e', \pi_{\text{key}(e')} t)} w_s(t_j) \right) \right) \\ &= w(t) \otimes \left(\bigotimes_{e' \in C_e} \left(\bigoplus_{t_j \in Q_f(e', \pi_{\text{key}(e')} t)} \left(\bigotimes_{e'' \in \mathcal{X}_{e'}} w(\pi_{e''} t_j) \right) \right) \right) \\ &= w(t) \otimes \left(\bigoplus_{t_i \in Q_f(e,t)} \left(\bigotimes_{e' \in \mathcal{X}_e - \{e\}} w(\pi_{e'} t_i) \right) \right) \\ &= \left(\bigoplus_{t_i \in Q_f(e,t)} \left(w(t) \times \bigotimes_{e' \in \mathcal{X}_e - \{e\}} w(\pi_{e'} t_i) \right) \right) = \bigoplus_{t_i \in Q_f(e,t)} w(t_i) \end{aligned}$$

至此，证明完成。 □

因此，如果 $\mathbf{y} = \varnothing$，$\mathbf{X}_r = \mathbf{T}$。引理 7.3 的正确性由引理 7.5 在 $e = r$ 上的正确性推出。

引理 7.6。 给定一个自由连通查询 Q 及其自由连通连接树 $\mathcal{T}$，以及任意节点 $e \in \mathcal{E}_{con}$，则对于任意 $t \in V_s(R_e)$

$$w_s(t) := \oplus_{t_i \in Q_f(e,t)} w(t_i).$$

引理 7.6 的证明。设 $\{e_1, e_2, \cdots, e_k\} = \mathbf{C}_e \cap \mathbf{X}_e$。根据引理 7.5，我们知道，对于每个元组 $t_i \in V_s(R_{e_i})$，

$$w_s(t_i) = \oplus_{t' \in Q_f(e,t_i)} w(t').$$

给定任意 $t \in V_s(R_e)$，我们可以将 $w_s(t)$ 改写如下：

$$
\begin{aligned}
&= w(t) \otimes \left(\bigotimes_{e' \in C_e \cap \mathcal{X}_e} \left(w_p(\pi_{\text{key}(e')} t \right) \right) \\
&= w(t) \otimes \left(\bigotimes_{e' \in C_e \cap \mathcal{X}_e} \left(\bigoplus_{t_j \in Q_f(e', \pi_{\text{key}(e')} t)} w_s(t_j) \right) \right) \\
&= w(t) \otimes \left(\bigotimes_{e' \in C_e \cap \mathcal{X}_e} \left(\bigoplus_{t_j \in Q_f(e', \pi_{\text{key}(e')} t)} \left(\bigotimes_{e'' \in \mathcal{X}_{e'}} w(\pi_{e''} t_j) \right) \right) \right) \\
&= w(t) \otimes \left(\bigoplus_{t_i \in Q_f(e,t)} \left(\bigotimes_{e' \in \mathcal{X}_e - \{e\}} w(\pi_{e'} t_i) \right) \right) \\
&= \left(\bigoplus_{t_i \in Q_f(e,t)} \left(w(t) \times \bigotimes_{e' \in \mathcal{X}_e - \{e\}} w(\pi_{e'} t_i) \right) \right) = \bigoplus_{t_i \in Q_f(e,t)} w(t_i)
\end{aligned}
$$

□

引理 7.4 的证明。任取查询结果 $t \in Q$ 和全连接结果 $t_1, t_2, \cdots, t_\ell$，使得对于任意 $i \in [\ell]$，都有 $\pi_{\mathbf{y}} t_i = t$。根据注释的定义，

$$w(t) := w(t_1) \oplus w(t_2) \oplus \cdots \oplus w(t_\ell). \tag{15}$$

我们先考虑 $\mathbf{T} = \mathbf{E}_{\text{con}}$ 的情况。在这种情况下，对于任意 $e \in \mathbf{E}_{\text{con}}$，对于每个元组 $t \in V_s(R_e)$，都有 $w_s(t) = w(t)$。这样，我们可以将 (15) 重写为

$$w(t) := \bigotimes_{e \in \mathcal{E}} \left(\bigoplus_{i \in [\ell]} w(\pi_e t_i) \right), \tag{16}$$

并且对于任意 $i \in [\ell]$,

$$\bigoplus_{i \in [\ell]} w(\pi_e t_i) = w_a(\pi_{\mathbf{y} \cap e} t)$$

根据定义。替换所有项后，我们可以得到方程 (14)。

另一方面，如果 $\mathbf{E}_{\text{con}} \subsetneq \mathbf{T}$，我们注意到 $\mathbf{T} = \uplus_{e \in \mathcal{E}_{\text{con}}} \mathbf{X}_e$，其中 $\oplus$ 表示不交并。任取 Q 的一个查询结果 t。这样，我们有

$$
\begin{aligned}
w(t) &= \oplus_{i \in [\ell]} w(t_i) \\
&= \oplus_{i \in [\ell]} \left(\otimes_{e \in \mathcal{E}_{\text{con}}} \left(\otimes_{e' \in \mathcal{X}_e} w(\pi_{e'} t_i) \right) \right) \\
&= \otimes_{e \in \mathcal{E}_{\text{con}}} \left(\oplus_{t_j \in Q_f(e, \pi_{\text{key}(e)} t)} w(t_j) \right) \\
&= \otimes_{e \in \mathcal{E}_{\text{con}}} \left(\oplus_{t_k \in V_s(R_e) : \pi_{e \cap y} t_k = \pi_{e \cap y} t} w_s(t_k) \right) \\
&= \otimes_{e \in \mathcal{E}_{\text{con}}} \left(w_a(\pi_{e \cap y} t) \right)
\end{aligned}
$$

至此，整个证明完成。 □

8 实验

8.1 设置

原型实现。 我们实现了这些算法，并在 Flink DataStream API 之上构建了一个名为 CROWN（**C**hange p**RO**pagation **W**ithout joi**N**s）的系统原型。我们的所有算法均实现为 DataStream 函数，这些函数以更新流作为输入。更新流中的每个元组

	CROWN	Flink	DBToaster	DBToaster Spark	Trill
分布式	✓	✓		✓	
Full 枚举	✓	✓	✓	✓	
增量枚举	✓				✓
更新	任意	FIFO	任意	批处理	任意
内部	This 论文	标准变更传播	HIVM	HIVM	标准变更 传播

表 1：不同查询处理引擎的比较。

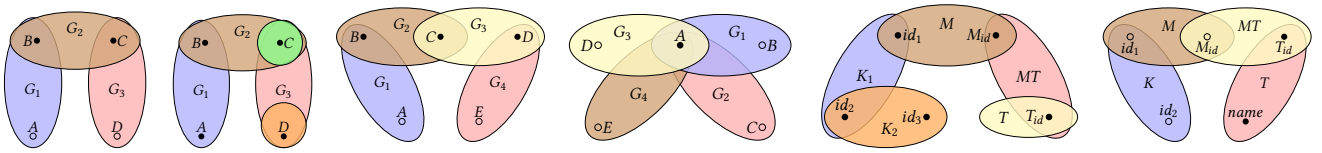
都带有一个标志，用于指示该更新是插入还是删除，同时还包含已更新关系的名称。处理完一次更新后，DataStream 函数会输出由该更新触发的增量。用户可根据需要调用完全查询结果的枚举。在 Flink 上实现该原型，使我们能够继承 Flink 的所有优势，例如容错能力，以及使用多种数据源和接收端的能力。为了以负载均衡的方式分发元组，我们借鉴了大规模并行算法中的类似思路，例如 HyperCube [2, 6, 36]。

我们在集中式和分布式环境中评估了我们的算法。集中式版本在单台机器上以单线程运行，并禁用了 Flink 的某些功能，例如容错、序列化和调度。这是为了与不支持这些功能的其他集中式系统（DBToaster 和 Trill）进行公平比较。分布式版本则启用了所有这些功能。它运行在两台机器上，每台机器均配备两颗主频为 2.1GHz、各有 48 个核心的英特尔至强处理器，以及 416 GB 内存。机器运行 Linux，并安装了 Scala 2.11.12、dotnet 5.0.403、Flink 1.13.5 和 Spark 2.2.3。每个查询都在每个引擎上评估 10 次，并报告平均运行时间。我们为每次运行设置了 4 小时的时间限制。

对比的查询处理引擎。我们将 CROWN 与以下系统进行比较：(1) DBToaster [3], , 这是支持集中式环境下对任意更新流执行多路连接的最佳 HIVM 引擎；(2) DBToaster Spark [31], , 它支持在分布式/并行环境中通过批量更新实现增量视图维护；(3) Trill [11], , 这是一个基于标准变更传播框架、针对流数据进行连续查询求值的系统；以及 (4) 原生 Flink SQL 流数据引擎。

表 1 总结了这些系统的各种特性。请注意，只有 CROWN 同时支持完整枚举和增量枚举。Flink 可以支持仅插入更新流或窗口数据流，但不支持任意更新流。我们对每项实验各运行两次：一次进行增量枚举，另一次进行完整枚举。进行完整枚举时，我们在处理完更新序列中每 10% 的内容后请求完整查询结果。由于 Trill 不支持完整枚举，我们让 Trill 在完整枚举时报告整个增量数据流。

查询与更新。 我们在两类查询上评估所有系统。第一类包含 Nguyen 等人提出的基准测试中的图模式查询，该基准测试使用 SNAP 数据集（斯坦福网络分析项目） [30], [28]。此类基准测试用于评估各系统在静态数据上执行连接查询的性能，而



(a) 3跳查询 (b) 2 梳状查询 (c) 4跳查询，SNB Q2 (d) 星形查询 (e) SNB Q3 (f) SNB Q4

图6：查询的关系超 **图。实心点表示连接-投影和聚合查询的输出属性** **es.**

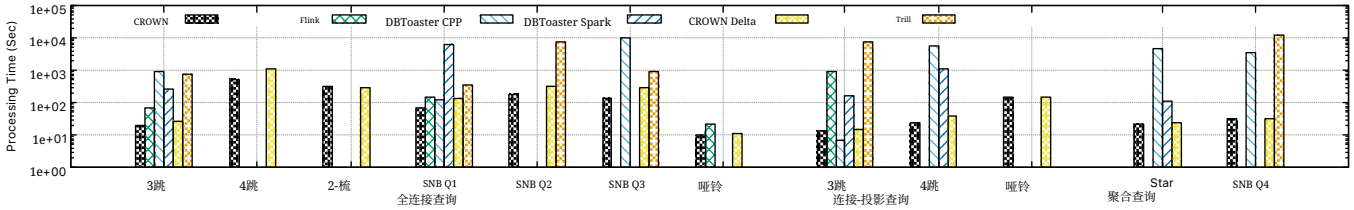
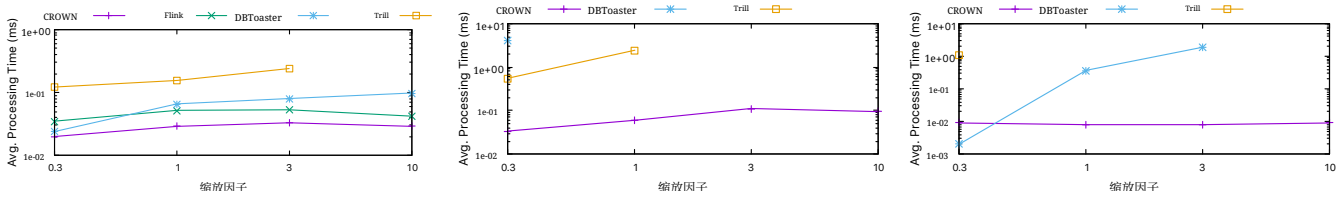


图7：CROWN、Flink、DBToaster 和 Trill 的处理时间



(a) SNB Q1 (b) SNB Q2 (c) SNB Q4

图8：平均处理时间与缩放因子的关系

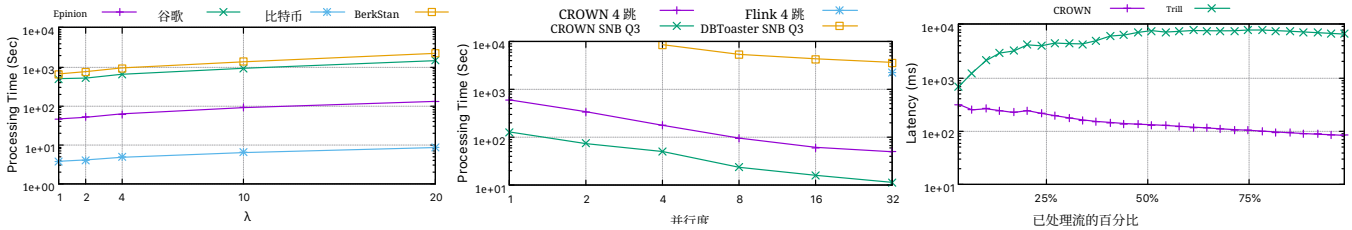


图9：运行时间与包围度 λ的关系。 **图10：运行时间与并行度 p。** **图 11：平均延迟。**

我们对其进行修改，以适用于动态场景。我们测试了基准测试中的所有无环查询，例如跳数（路径）查询、星形查询和梳形查询。我们还测试了哑铃查询，它是棒棒糖查询的一种变体。详细的查询定义见附录D，下面给出了一个3跳查询示例，其中我们使用一个过滤器来控制输出大小。

选择 G1.src 作为 A，G2.src 作为 B,G3.src 作为 C，G3.dst 作为 D从 G G1，G G2，G G3其中 G1.dst = G2.src 并且 G2.dst = G3.src并且 过滤器 OVER (G3.dst)

第二类包括针对 LDBC社交网络基准测试（LDBC-SNB）[14], 的更复杂的分析查询，这些查询会在持续更新的情况下访问图中给定节点的邻域。以下是一个示例：在满足筛选条件的同时，查找与特定标签ID相关联的不同消息数量：

SELECT t_name, t_tagid, COUNT(DISTINCT m_messageid) FROM 标签, 消息, message_tag, 认识 WHERE m_messageid = mt_messageid AND mt_tagid = t_tagid AND m_creatorid = k_person2id AND m_c_replyof IS NULL AND 过滤器 OVER (k_person1id) 分组 t_name, t_tag_ids

图6展示了所有查询的连接超图。除了 2-梳、SNB Q3 和 Q4 之外，它们都具有高度为2的自由连通连接树。星形查询（图6(d)）具有高度为 1 的自由连通连接树，

因此它是 q-层次化的。4跳查询（图6(c)）和 SNB Q4 查询（图6(f)）具有相同的超图结构，但输出属性不同；4跳查询具有高度为2的自由连通连接树，而 SNB Q4 查询则没有。

我们使用参数 w创建先进先出流。对于图查询，我们为图中的每条边 e分配一个不同的整数 t_e，其中e的生命周期为 $[t_e, t_e + w]$。对于 LDBC-SNB 查询，基准测试中的每个元组 t都已有插入时间戳 t^+，我们将其删除时间设为插入后 w 天，即 $t^- = t^+ + w$。请注意，图查询的滑动窗口是基于计数的，也就是说，窗口始终包含相同数量的元组。另一方面，LDBC-SNB 查询的窗口是基于时间的，因此窗口中的元组数量会随时间波动。

8.2 实验结果

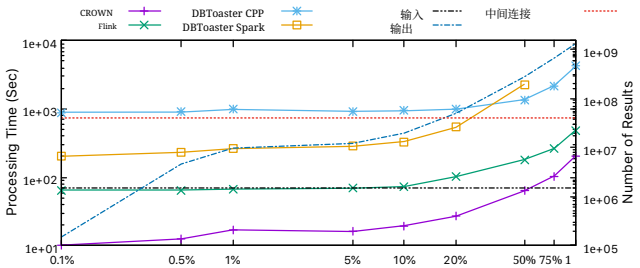
运行时间。图 7 展示了在集中式环境中，针对中等规模图 *Epinions* 上的每个图查询以及每个 SNB 查询进行评估的总运行时间。该图包含约 500K 条边和 76K 个顶点，以及 3.7B 条 3 跳路径和 378B 条 4跳路径。另一方面，我们对所有 SNB 查询均使用默认缩放因子 1。在此缩放因子下，原始数据的总大小为$1.5GB$，最大的关系包含 15 个属性。我们设置筛选条件，使所有查询仅保留指定端点的 10%。图中缺失的柱状条表示相应系统未能在 4 小时限制内完成，或因错误而中止（主要是内存溢出错误和垃圾回收超时）。只有 CROWN 能够成功完成所有查询。Trill 只能处理少数图查询。一种可能的解释是，图查询往往会生成大量增量。另一方面，Flink 在评估 SNB Q2、Q3 和 Q4 时内存耗尽。对于系统能够完成的那些查询，我们发现，与 Flink 相比，CROWN 的加速比为 2x 至 67x；与 DBToaster 相比为 1.8x 至 234x；与 Trill 相比为 2.7x 至 523x。此外，在处理连接-投影查询时，CROWN 所需时间远少于处理相应的全连接查询，而 Flink 所需时间则更长。此外，CROWN 在完全枚举和增量枚举方面均表现出色，不同的输出模式不会影响 CROWN 的整体性能。

包围度。为测试包围度的影响，我们针对不同的 λ，在 SNAP 数据集中的不同图上创建多个更新序列。我们禁用输出，以观察更新开销会随不同的 λ 而变化。实验结果如图9所示。从结果可以看出，CROWN 的维护开销几乎随着 λ 的增加而线性增长。

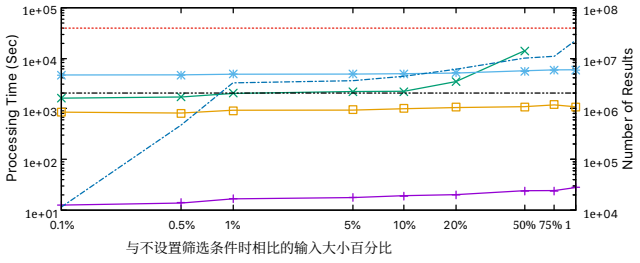
分布式处理。为了在分布式环境下将 CROWN 与 DBToaster Spark 和 Flink 进行比较，我们搭建了一个包含 32 个任务槽的小型集群，并测试了 4跳和 SNB Q3查询；DBToaster 和 Flink 在集中式环境下无法完成这些查询。图 10 展示了结果；缺失的数据点或曲线表示系统无法在时限内完成。尽管我们采用 HyperCube算法来分发所有元组，CROWN 仍能在 $p < 16$ 下实现线性加速比，其中 p是工作节点数。可用工作节点越多，边际收益越小。这符合预期，因为：(1) HyperCube 算法意味着增加工作节点时，加速比会变为次线性；(2) 处理时间已经很短，导致系统开销占据主导地位整个运行时间。对于所有已完成的数据点，CROWN 可提供 45 倍至 324 倍的加速比。

由于 Flink 和 DBToaster 无法使用 128GB 内存完成所有实验，因此我们将这两个系统的内存用量增加到 500GB，但它们仍只能完成一小部分实验。另一方面，CROWN 仅用 128GB 内存就能完成所有实验。如果进一步将 CROWN 的内存用量限制为 16GB（即每个工作节点 500MB），CROWN 仍能良好运行，性能几乎没有变化。

延迟。最后，我们测试了增量枚举的延迟，即从收到更新到输出相应增量所需的时间。图11展示了结果。CROWN 的平均延迟不到 90 毫秒，而 Trill 的平均延迟超过 6 秒。此外，CROWN 的平均延迟保持稳定，而 Trill 的平均延迟则持续增长，导致其无法长时间处理数据流。



(a) 3跳查询



(b) 4跳查询

图12：运行时间与选择率的关系

可扩展性。为了测试不同平台的可扩展性，我们调整 SNB 基准测试的缩放因子，并比较不同平台的平均更新开销。实验结果如图8所示。结果表明，CROWN 的平均处理时间在不同数据规模下保持稳定。相比之下，数据规模会影响其他平台的平均处理时间，这说明 CROWN 的可扩展性优于其他平台。

选择率。图 12(a) 展示了连接条件选择率变化时的运行时间。对于标准变化传播和 HIVM，维护开销不仅取决于输入大小和输出大小，还取决于中间视图的大小。对于 3跳查询 $G_1(A, B) \bowtie G_2(B, C) \bowtie G_3(C, D)$，针对 $G_1 = G_2 = G$和$G_3 =$ 过滤器(G)，即使 G_3 为空，维护开销也会受视图 $G_1 \bowtie G_2$ 的大小限制。与此同时，

CROWN 的维护开销只取决于输入大小和输出大小。为了更好地展示这一特性，我们调整了 3跳查询中的筛选条件，只改变 $|G_3|$，而不是 $|G_1 \bowtie G_2|$。此处省略了 Trill，因为除第一个数据点外，它在所有数据点上的运行时间都超过了 4 小时。当 $|G_3| \geqslant 0.5\%|G|$ 时，输出大小超过输入大小；而当 $|G_3| \geqslant 20\%|G|$ 时，输出大小超过中间连接大小 $|G_1 \bowtie G_2|$。从结果可以看出，CROWN 的运行时间几乎随 $|G|+|Q|$ 线性增长，这符合预期，因为更新序列具有 $\lambda = 1$。另一方面，DBToaster 和 Flink 的运行时间与 $|G_1 \bowtie G_2|+|Q|$ 成正比，因此当 $|G_3| \leqslant 20\%|G|$ 时性能较差。在图12(b)中评估带投影的 4跳查询时，可以观察到更大的差距：即使没有任何筛选条件，中间连接大小也超过查询结果的大小。Flink 和 DBToaster 执行 4跳查询的运行时间超过执行 3跳查询的运行时间，即使输出大小很小。与此同时，CROWN 的运行时间要短得多，而且只取决于输入大小和输出大小。

参考文献

[1] Mahmoud Abo Khamis、Hung Q Ngo 和 Atri Rudra。2016。FAQ：常见问题。在第*35*届 *ACM SIGMOD-SIGACT-SIGAI* 数据库系统原理研讨会论文集。13–28。[2] Foto N. Afrati 和 Jeffrey D. Ullman。2011。在 Map-Reduce 环境中优化多路连接。*IEEE*知识与数据工程汇刊 23, 9 2011 1282 1298 ( )，–。[3] Yanif Ahmad、Oliver Kennedy、Christoph Koch 和 Milos Nikolic。2012。DBToaster：动态频繁更新视图的高阶增量处理。*VLDB* 基金会论文集 5, 10 (2012), 968–979。[4] Albert Atserias、Martin Grohe 和 Dániel Marx。2013。关系连接的规模界限和查询计划。*SIAM J. Comput.* 42, 4 (2013), 1737–1767。[5] Guillaume Bagan、Arnaud Durand 和 Etienne Grandjean。2007。关于无环合取查询与常数延迟枚举。在计算机科学逻辑。Springer Berlin Heidelberg，德国海德堡柏林，208–222。[6] Paul Beame、Paraschos Koutris 和 Dan Suciu。2017。并行查询处理的通信步骤。*J. ACM* 64, 6，文章 40（2017年10月），58页。https://doi.org/ 10 1145 3125644 。/ C. Beeri、R. Fagin、D. Maier 和 M. Yannakakis。1983。无环数据库模式的可取性。*JACM* 30, 3 (1983), 479–513。[8] Christoph Berkholz、Jens Keppeler 和 Nicole Schweikardt。2017。更新条件下回答合取查询。在第*36*届 *ACM SIGMOD-SIGACT-SIGAI* 数据库系统原理研讨会论文集（美国伊利诺伊州芝加哥）*(PODS '17)*。计算机协会，美国纽约州纽约，303–318。https://doi.org/10.1145/3034786.3034789[9] Paris Carbone、Asterios Katsifodimos、Stephan Ewen、Volker Markl、Seif Haridi 和 Kostas Tzoumas。2015。Apache Flink：单引擎中的流处理与批处理。*IEEE*数据工程通报 38, 4 (2015), 28–38。[10] Nofar Carmeli 和 Markus Kröll。2019。合取查询并集的枚举复杂度。在第*38*届 *ACM SIGMOD-SIGACT-SIGAI* 数据库系统原理研讨会论文集。ACM，134–148。[11] Badrish Chandramouli、Jonathan Goldstein、Mike Barnett、Robert DeLine、Danyel Fisher、John C Platt、James F Terwilliger 和 John Wernsing。2014。Trill：适用于多种分析的高性能增量查询处理器。*VLDB* 基金会论文集 8, 4 (2014), 401–412。[12] Rada Chirkova 和 Jun Yang。2012。物化视图。*Foundations and Trends® in Databases* 4, 4 (2012), 295–405。[13] Mohammed Elseidy、Abdallah Elguindy、Aleksandar Vitorovic 和 Christoph Koch。2014。可扩展自适应在线连接。*Proc. VLDB Endow.* 7, 6 (2月 2014 441 452 10 14778 2732279 2732281)，–。https://doi.org/。/。Orri Erling、Alex Averbuch、Josep Larriba-Pey、Hassan Chafi、Andrey Gubichev、Arnau Prat、Minh-Duc Pham 和 Peter Boncz。2015。LDBC 社交网络基准测试：交互式工作负载。在*2015* 年 *ACM SIGMOD* 数据管理国际会议论文集（澳大利亚维多利亚州墨尔本）*(SIGMOD '15)*。计算机协会，美国纽约州纽约，619 630 10 1145 2723372 2742786 –。https://doi.org/。/。R. Fagin。无。超图与关系数据库模式的无环度。*JACM* 30, 3 (1983), 514–550。[16] Buğra Gedik、Rajesh R Bordawekar 和 Philip S Yu。2009。CellJoin：面向 Cell 处理器的并行流连接算子。VLDB 期刊 18，2（2009），501–519。[17] Georg Gottlob、Nicola Leone 和 Francesco Scarcello。2002。超树分解与易处理查询。计算机与系统科学杂志 64，3（2002），579–627。[18] Timothy Griffin 和 Bharat Kumar。1998。半连接和外连接查询的代数变更传播。SIGMOD 记录 27，3（1998 年 9 月），22–27。https: //doi.org/10.1145/290593.290597[19] Monika Henzinger、Sebastian Krinninger、Danupon Nanongkai 和 Thatchaphol Saranurak。2015。通过在线矩阵向量乘法猜想统一并强化动态问题的困难性。载于第四十七届 ACM 理论计算研讨会论文集（美国俄勒冈州波特兰）（STOC '15）。计算机协会，美国纽约州纽约， 21 30 10 1145 2746539 2746609 –。https://doi.org/ . / . Muhammad Idris、Martin Ugarte 和 Stijn Vansummeren。2017。《动态 Yannakakis 算法：更新下紧凑高效的查询处理》。载于 2017 年 ACM 数据管理国际会议论文集（美国伊利诺伊州芝加哥）（SIGMOD '17）。计算机协会，美国纽约州纽约， 1259 1274 10 1145 3035918 3064027 –。https://doi.org/ . / . Muhammad Idris、Martín Ugarte、Stijn Vansummeren、Hannes Voigt 和 Wolfgang Lehner。2019。动态变化数据集的高效查询处理。ACM SIGMOD 记录 48，1（2019），33–40。[22] Muhammad Idris、Martín Ugarte、Stijn Vansummeren、Hannes Voigt 和 Wolfgang Lehner。2020。通用动态 Yannakakis：更新下带有 θ 连接的合取查询。VLDB 期刊 29，2（2020），619–653。[23] Manas R. Joglekar、Rohan Puttagunta 和 Christopher Ré。2016。AJAR：带注释关系上的聚合与连接。载于第35届 ACM SIGMOD-SIGACT-SIGAI 数据库系统原理研讨会论文集（美国加利福尼亚州旧金山）（PODS '16）。计算机协会，美国纽约州纽约， 91 106 10 1145 2902251 2902293 –。https://doi.org/ . / . Jaewoo Kang 、 Jeffrey F Naughton 和 Stratis D Viglas。2003。评估无界流上的窗口连接。载于第19届国际数据工程会议论文集（目录号 03CH37405）。IEEE，341–352。[25] Ahmet Kara、Hung Q. Ngo、Milos Nikolic、Dan Olteanu 和 Haozhe Zhang。2020。更新下维护三角形查询。ACM 数据库系统汇刊 45，3，文章 11 2020 46 10 1145 3396375 (aug)，页数。https://doi.org/ . / Ahmet Kara、Milos Nikolic、Dan Olteanu 和 Haozhe Zhang。2020。层次化查询静态与动态求值的权衡。载于第39届 ACM SIGMOD-SIGACT-SIGAI 数据库系统原理研讨会论文集。375–392。[27] Ki Yong Lee、Jin Hyun Son 和 Myoung Ho Kim。2001。数据仓库中的高效增量视图维护。载于第十届信息与知识管理国际会议论文集（美国佐治亚州亚特兰大）（CIKM '01）。计算机协会，美国纽约州纽约，349–356。https://doi.org/ 10 1145 502585 502644 . / . Jure Leskovec 和 Andrej Krevl。SNAP 数据集：斯坦福大型网络数据集集合。2014。http://snap.stanford.edu/data。[29] Qian Lin、Beng Chin Ooi、Zhengkui Wang 和 Cui Yu。2015。可扩展的分布式流连接处理。载于 2015 年 ACM SIGMOD 数据管理国际会议论文集。811–825。[30] Dung Nguyen、Molham Aref、Martin Bravenboer、George Kollias、Hung Q Ngo、Christopher Ré 和 Atri Rudra。2015。图模式连接处理：老方法的新技巧。载于 GRADES'15 论文集。1–8。[31] Milos Nikolic、Mohammad Dashti 和 Christoph Koch。2016。如何赢得吃热狗比赛：采用批量更新的分布式增量视图维护。载于 ACM SIGMOD 数据管理国际会议论文集。ACM，511–526。[32] Milos Nikolic 和 Dan Olteanu。2018。采用三重锁因式分解的增量视图维护。载于 ACM SIGMOD 数据管理国际会议论文集。ACM，365–380。[33] Milos Nikolic、Haozhe Zhang、Ahmet Kara 和 Dan Olteanu。2020。F-IVM：基于快速演变的关系数据进行学习。载于 2020 年 ACM SIGMOD 数据管理国际会议论文集。2773–2776。[34] Kenneth A. Ross、Divesh Srivastava 和 S. Sudarshan。1996。物化视图维护与完整性约束检查：以空间换时间。载于 1996 年 ACM SIGMOD 数据管理国际会议论文集（加拿大魁北克省蒙特利尔）（SIGMOD '96）。计算机协会，美国纽约州纽约， 447 458 10 1145 233269 233361 –。https://doi.org/ . / . Pratanu Roy、Jens Teubner 和 Rainer Gemulla。2014。低延迟握手连接。VLDB 基金会论文集 7，9（2014），709–720。[36] Qichen Wang 和 Ke Yi。2020。更新下维护无环外键连接。载于 2020 年 ACM SIGMOD 数据管理国际会议论文集。1225–1239。[37] Mihalis Yannakakis。1981。无环数据库模式算法。载于超大型数据库国际会议论文集。82–94。

第3节缺失的证明

引理 A.1。 给定一个无环合取查询 $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$，当且仅当它具有第 *3* 节中定义的自由连通连接树 $\mathcal{T}$ 时，它才是自由连通的。

证明。若方向成立。假设存在一个如第 3 节所定义的自由连通连接树 T。我们只需添加一个节点 r，其中恰好包含y中的所有输出属性，并将其设为 T 的根，然后执行以下过程：

- 我们以自底向上的方式访问根节点。对于每个满足$e \in \mathrm{T}_{\mathrm{con}}$的节点 e，我们删除 e 与其父节点之间的边，然后将 e 连同以 e 为根的当前子树，一起移到 r下作为子节点。令 T′ 为所得的树。

显然，T ′中的每个节点都对应于 $\mathrm{E} \cup \{\mathbf{y}\}$的一个关系或广义关系。此外，根据 T 的定义，对于每个节点 $e \in \mathrm{T}_{\mathrm{con}}$及其父节点 p_e，都有$e \cap p_e \subseteq \mathbf{y}$。这样，y中输出属性的连通条件得以保留，因为$\mathbf{y} \subseteq r$。根据 [20], 中无环合取查询的定义，存在一个自由连通连接树，其对应于 $(\mathrm{V}, \mathrm{E} \cup \{\mathbf{y}\}, \mathbf{y})$；因此，$(\mathrm{V}, \mathrm{E} \cup \{\mathbf{y}\}, \mathbf{y})$ 也是无环的。根据 [5], 中自由连通合取查询的定义，Q 是自由连通的。

仅当方向。假设给定一个自由连通合取查询 $\mathrm{Q} = (\mathrm{V}, \mathrm{E}, \mathbf{y})$。接下来，我们将展示如何构造第 3 节中定义的自由连通连接树。令 $\mathrm{E}_{\mathbf{y}} = \{e \cap \mathbf{y} : e \in \mathrm{E}\}$。我们先给出以下辅助引理：

引理 A.2（[5], 引理 21）。对于任意自由连通合取查询$Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$，存在一棵传统连接树$\mathcal{T}'$，其对应于 $(\mathcal{V}, \mathcal{E} \cup \mathcal{E}_{\mathbf{y}}, \mathbf{y})$，以及一个关系子集$\mathcal{E}_{con} \subseteq \mathcal{E} \cup \mathcal{E}_{\mathbf{y}}$，使得$\mathcal{E}_{con}$对应的节点构成$\mathcal{T}'$, 的一棵连通子树；也就是说，$\mathcal{E}_{con}$ 包含$\mathcal{T}'$, 的根$\mathbf{y} = \bigcup_{e \in \mathcal{E}_{con}} e$，且该子树是连通的。

令 T ′为这样一棵传统连接树，对应于 $(\mathrm{V}, \mathrm{E} \cup \mathrm{E}_{\mathbf{y}}, \mathbf{y})$，并以节点 r为根。注意，$r \subseteq \mathbf{y}$。我们将通过以下步骤把 T′变换为 Q 的自由连通连接树：

步骤 1：移除所有满足 $e \cap \mathbf{y} = \varnothing$ 的节点 e。我们先给出一个观察。对于任意一对 e、$e' \in \mathrm{E} \cup \mathrm{E}\mathbf{y}$，若 $e \cap \mathbf{y} = \varnothing$ 且 $e' \cap \mathbf{y} \neq \varnothing$，则在 T ′中，$e$ 不可能是 e' 的祖先节点。反设不然，$e' \notin \mathrm{Econ}$，因为 $e \notin \mathrm{Econ}$。令 $x \in e' \cap \mathbf{y}$ 为 e' 中的一个输出属性。由 Econ 的性质可知，必定存在某个节点 $e'' \in \mathrm{Econ}$ 满足 $x \in e''$。此外，由于 $e \notin \mathrm{Econ}$ 且 Econ 构成一棵连通子树，因此 $e'' \notin \mathrm{T}_e$。于是，根据 x 的连通性质，位于 e'' 与 e'' 之间路径上的所有节点（包括 e）都必须包含 x。这与 $x \notin e$ 矛盾，因为 $e \cap \mathbf{y} = \varnothing$。

这样，我们按如下方式从 T′ 中移除所有关系 $e \cap \mathbf{y} = \varnothing$。对于任意满足 $e \cap \mathbf{y} = \varnothing$ 且 $p_e \cap \mathbf{y} \neq \varnothing$ 的关系 $e \in \mathrm{E}$，我们将以 e 为根的子树整体移除。根据上述观察，位于以 e 为根的子树中的所有关系都不包含任何输出属性。最后，如果 $p_e - \mathbf{y} \neq \varnothing$，我们会将这棵子树作为关系 p_e 的子节点放回；如果 $p_e - \mathbf{y} = \varnothing$，则将其作为任意关系的子节点放回。

步骤 2：移除所有节点 $e' \in \mathrm{E}\mathbf{y}$ **满足** $e' \subsetneq e$ **对于某个**$e \in \mathrm{Econ}$。如果存在一对节点 $e \in \mathrm{E}\mathbf{y}e' \in \mathrm{Econ}$ 满足$e \subsetneq e'$，我们可以移除 e，并将其每个子节点（如果 e' 是 e 的子节点，则不包括 e'）作为 e' 的新子节点添加。它可以是娱乐主管很容易验证，连通性质得以保留。约简后的 Econ 仍是已更新 T ′的有效连通子树。此步骤之后，我们假设不存在一对节点 $e \in \mathrm{E}\mathbf{y}e' \in \mathrm{Econ}$ 满足 $e \subsetneq e'$。

引理 A.3. 对于任意节点 $e \notin \mathcal{E}_{con}$ 及其最低祖先节点 $e' \in \mathcal{E}_{con}$, $e \cap \mathbf{y} \subseteq e'$.

证明。反设存在一个输出属性 $x \in e \cap \mathbf{y} - e'$。由连通性质可知，$\mathrm{T}' - \mathrm{T}_{e'}$中没有其他节点包含 x。令 e''为位于从 e' 到 e的路径上的 e'的子节点。此外，$\mathrm{T}'_{e''}$ 中除属于 Econ的节点外，没有其他节点，因为 e' 是 Econ中 e 的较低祖先。因此，Econ 中没有节点包含 x，这与 Econ的性质矛盾。 □

引理 A.4. 步骤 *2* 之后，$\mathcal{E}_{\mathbf{y}} \subseteq \mathcal{E}_{con}$.

证明。反设 $e' \in \mathrm{E}_{\mathbf{y}} - \mathrm{Econ}$。令 $e \in \mathrm{Econ}$ 为 e'的最低祖先节点。由引理 A.3可知，$e' \subseteq e$，因此 e' 将在步骤 2 中被移除。 □

步骤 3：为 $\mathrm{E}_{\mathbf{y}}$ 中的每个关系添加一个守卫。考虑任意节点 $e \in \mathrm{E} - \mathrm{Econ}$ 及其父节点 p_e，其中 $p_e \subseteq \mathbf{y}$ 且 $p_e \neq e \cap \mathbf{y}$。我们移除以 e 为根节点的子树，并将其作为$e' \in \mathrm{Econ}$ 的子节点（若存在这样的节点），其中 $e' = e \cap \mathbf{y}$。接下来证明连通性质得以保持。由于变换前 T′ 的连通性质，$e' \cap p_{e'} = e' \cap p_{e'} \cap \mathbf{y} \subseteq e$，因此此步骤保持了连通性质。

引理 A.5。 步骤 *3* 之后，对于任意节点 $e' \in \mathcal{E}_{\mathbf{y}}$，它都有一个子节点 $e \in \mathcal{E}$，使得 $e' = e \cap \mathbf{y}$。

证明。考虑任意节点 $e' \in \mathrm{E}\mathbf{y}$。注意到 $e' \in \mathrm{Econ}$，这由引理 A.4 得出。反设任意满足 $e' = e \cap \mathbf{y}$ 的节点 $e \in \mathrm{E}$ 都不是 e' 的子节点。

如果 e' 是 e 的祖先节点，令 e'' 为从 e 到 e' 的路径上经过的 e' 的子节点。必定有 $e' = e' \cap e \subseteq e''$。如果 $e'' \in \mathrm{E}$连通，$e' \subseteq e''$，这与步骤 2 矛盾。如果 $e'' \notin \mathrm{E}$连通，那么根据引理 A.3，$e' = e' \cap e = e' \cap e'' = e'' \cap \mathbf{y}$ 意味着 e'' 是 e' 的这样一个子节点，这与步骤 2 矛盾。

否则，$e' = e' \cap e \subseteq p_{e'}$。由于 $e \in \mathrm{E}$连通，$p_{e'} \in \mathrm{E}$连通。综合起来，$e' \subseteq p_{e'}$，这与步骤 2 矛盾。□

步骤 4：用广义关系替换节点。最后，我们开始以自顶向下方式遍历 $\mathrm{E}_{\mathbf{y}}$ 中的节点。考虑任意节点 e'。注意，$e' \subseteq \mathbf{y}$。根据引理 A.5，它必定有某个子节点 $e \in \mathrm{E}$，其满足 $e \cap \mathbf{y} = e'$，此条件在步骤 3 之后成立。如果对于 e' 的每个子节点 e，$e' \subseteq e$ 都成立，并且如果 $p_{e'}$ 存在，则 $p_{e'}$ 是一个广义关系，我们就将 e' 替换为广义关系 $[e']$。否则，我们用 e 替换 e'。如果 $e \notin \mathrm{E}$连通，我们就将其加入 E连通。对于 e' 的任意子节点 $e'' \in \mathrm{E}$连通，它现在是 e 的子节点，我们有 $e'' \cap e' = e'' \cap e' \cap \mathbf{y} = e'' \cap e \subseteq \mathbf{y}$。不难验证，连通性质得以保持，并且每个广义关系都具备守卫性质。此外，如果存在广义关系的父节点，该父节点也是广义关系。因此，上方性质得以保持。

经过这四个步骤后，得到的树就是一棵满足所有性质的自由连通连接树。□

B 第 5 节缺失的证明

引理 5.5 的证明。我们先证明“仅当”方向。对于任意 $t \in V_l(R_e)$，都存在一个 $t' \in Q(D)$，使得 $\pi_e t' = t$。与此同时，这表明 $t'' = \pi_{p_e} t'$ 必须满足 $t'' \in V_l(R_{p_e})$，因为 $t'' \in \pi_{p_e} Q(D)$。因此，t'' 可以与 t 连接，这表明 $t \bowtie V_l(R_{p_e}) \neq \emptyset$。

对于“若”方向。设 $t' \in V_l(R_{p_e})$ 是一个可以与 t 连接的元组。我们将连接树 T 划分为两个子树 $(\mathrm{T}_e、\ \mathrm{T} \setminus \mathrm{T}_e)$，并相应地将输出属性 y 划分为两个集合 $(\mathrm{y}_e、\ \mathrm{y} \setminus \mathrm{y}_e)$。由于 $t' \in V_l(R_{p_e})$，$\pi_{\mathrm{y} \setminus \mathrm{y}_e} Q(D \ltimes t') \neq \emptyset$，且我们令 t'' 为 $\pi_{\mathrm{y} \setminus \mathrm{y}_e} Q(D \ltimes t')$ 中的一个元组。另一方面，由于 $t \in V_s(R_e)$，在 $\pi_{\mathrm{y}_e}(\bowtie_{e' \in \mathrm{T}_e} R_{e'})$ 中也存在一个元组 t_e。t_e 可以与 t'' 连接，因为 t 可以与 t' 连接，且 $\pi_e t_e = t$，$\pi_{p_e} t'' = t'$。因此，$t_e \bowtie t'' \in Q(D)$ 表明 $t \in V_l(R_e)$。□

引理 5.7 的证明。不失一般性，假设 t 被插入。t 被删除的情形可用相同的论证证明。

方向 $\supseteq$。我们证明，对于 t 的每个见证元组 t'，$Q(D \ltimes t')$ 中的每个结果也会出现在 $\Delta Q(D,\ t)$ 中。不失一般性，考虑某个见证元组 t' 对应的查询结果 $q \in Q(D \ltimes t')$，其中对于 $e \subseteq \mathrm{y}$，要么 $t' \in R_e$，要么对于 $e \cap \mathrm{y} - p_e \neq \emptyset$，$t' \in \pi_{\mathrm{y}} R_e$。

首先，$Q(D \ltimes t') \subseteq Q(D + t)$ 因为在插入 t 之后，我们有 $t' \in \Delta\left(\pi_{\mathrm{y}} V_s(R_e)\right)$ 和 $\pi_{\text{键}(e)} t' \in V_p(R_e)$。因此，所有由 t' 见证的结果都会在插入 t 之后出现在 $Q(D + t)$ 中，即 $q \in Q(D + t)$。接下来我们证明 $q \notin Q(D)$。现在回到插入 t 之前的时间戳。根据见证元组的定义，当时有 $t' \notin \pi_{\mathrm{y}} V_s(R_e)$。我们再区分两种情形。

- 情况 1：$e \subseteq \mathrm{y}, q \notin Q(D)$ 因为 $\pi_e q = t'$ 但是在 t 插入之前 $t' \notin \pi_e Q(D)$。这进一步表明 $q \notin Q$。
- 情况 2：$e - \mathrm{y} \neq \emptyset$ 和 $e \cap \mathrm{y} - p_e \neq \emptyset, t' \notin \pi_{\mathrm{y}} V_s(R_e)$ 在 t' 插入之前。这样，$t' \notin \pi_{\mathrm{y}} Q(D)$，因此 $q \notin Q$。

结合上述分析，我们有 $q \in Q(D + t)$ $q \notin Q(D)$ $q \in \Delta Q(D\ t)$ $\uplus_{t'}$ $_t Q(D \ltimes t') \subseteq \Delta Q(D \ltimes t)$，并且即 。因此，是的一个见证。

方向 $\subseteq$。接下来我们证明，$\Delta Q(D, t)$ 中的每个结果都属于 $Q(D+t) \ltimes t'$，其中存在 t 的某个见证元组 t'。考虑一个任意查询结果 $q \in Q(D + t) - Q(D)$。

只需证明至少存在一个节点 $e \in \mathrm{T}$，使得元组 $t' = \pi_e q$ if $e \subseteq \mathrm{y}$，或者元组 $t' \in \pi_{\mathrm{y}} R_e$ with $t' = \pi_{e \cap \mathrm{y}} q$ if $e \cap \mathrm{y} - p_e \neq \emptyset$，必定是一个见证。一个重要的观察是，$t'$ 现在属于 $\Delta\pi_{\mathrm{y}} V_s(R_e)$；否则，$q \in Q(D)$，这将导致矛盾。现在考虑满足 $t_1 = \pi_{e_1 \cap \mathrm{y}} q$ and $t_1 \in \Delta\left(\pi_{\mathrm{y}} V_s(R_{e_1})\right)$ 的最高节点 e_1。如果 e_1 是根节点，那么根据定义 5.6，t_1 必定是 t 的见证。否则，e_1 不是根节点。考虑 $t_2 = \pi_{p_{e_1} \cap \mathrm{y}} q$。由于 $t_2 \notin \Delta V_s(R_{e_2})$，在插入 t 之前，t_2 必须位于 $V_s(R_{e_2})$ 和 $t_2 \in \pi_{e_2} Q(D)$ 中，这表明 $\pi_{\mathrm{key}(e_1)} t_1 = \pi_{\mathrm{key}(e_1)} t_2 \in V_p(R_{e_1})$，且 $\pi_{\mathrm{key}(e_1)} t_1 \in \pi_{\mathrm{key}(e_1)} Q(D)$。因此，根据定义，$t_1$ 是 t 的见证。

关键性质： $\Delta Q(D,\ t_1) \cap \Delta Q(D,\ t_2) = \emptyset$ **对任意一对见证元组** t_1，t_2 **均成立**。还需证明 $\bigcup_{t'} {}_t \Delta Q(D \ltimes t')$ 中不存在重复的查询结果。反证，假设存在一个查询结果 q，它至少有两个见证元组。不失一般性，令 t_1，t_2 是 q 中两个不同的见证元组，其中 $t_1 \in R_{e_1}$ 对于某个 $e_1 \subseteq \mathrm{y}$ 或 $e_1 \cap \mathrm{y} - p_{e_1} \neq \emptyset$，以及某个 $e_2 \subseteq \mathrm{y}$ 或 $e_2 \cap \mathrm{y} - p_{e_2} \neq \emptyset$。首先，$e_1 \neq e_2$，因为 q 在每个关系中最多包含一个元组。注意，$t \in R_e$ 的插入只能改变 e 的祖先节点中的元组状态。不失一般性，令 e_1 为 e_2 的祖先节点。令 e_3 为 e_2 的父节点（可能出现 $e_1 = e_3$ 的情况）。令 $t_3 = \pi_{e_3 \cap \mathrm{y}} q$。根据见证元组的定义，$t_3 \in \pi_{\mathrm{y}} V_s(R_{e_3})$ 在插入 t 之前。根据 $t_3 \notin \Delta\pi_{\mathrm{y}} V_s(R_{e_3})$，$t_1 \in \pi_{\mathrm{y}} V_s(R_{e_1})$ 在插入之前，这与 t_1 是见证元组这一事实矛盾。这样，$\Delta Q(D,\ t)$ 中的每个结果都对应一个见证元组，因此，不同见证元组对应的扩展查询结果之间不存在重复项。□

引理 5.9 的证明。我们首先证明算法 6 的正确性。考虑任意一个见证元组 $t' \in R_{e_1}$。依次将位于从 e_1 到 r 的路径上的节点记为 $e_1,\ e_2,\ \cdots,\ e_k(r)$。我们可以先将 $Q(D \ltimes t')$ 展开如下：

$$t' \bowtie \left(\bowtie_{i=1}^{k} V_r(e_i)\right) \bowtie Q_{\mathcal{T}_{e_1}} \bowtie \left(\bowtie_{i=2}^{k} Q_{\mathcal{T}_{e_i} - \mathcal{T}_{e_{i-1}}}\right) \tag{17}$$

其中 $Q_{\mathcal{T}}$ 表示定义在 T 中关系上的查询。根据连接算子和自由连通连接树的性质，我们可以进一步将 (17) =改写为：

$$\bigcup_{S \in t' \bowtie \left(\bowtie_{i=1}^{k} V_r(e_i)\right)} S \bowtie Q_{\mathcal{T}_{e_1}} \bowtie \left(\bowtie_{i=2}^{k} Q_{\mathcal{T}_{e_i} - \mathcal{T}_{e_{i-1}}}\right)$$

$$= \bigcup_{S \in t' \bowtie \left(\bowtie_{i=1}^{k} V_r(e_i)\right)} S \times \left(Q_{\mathcal{T}_{e_1}} \ltimes \{S\}\right) \times \left(\bowtie_{i=2}^{k} \left(Q_{\mathcal{T}_{e_i} - \mathcal{T}_{e_{i-1}}} \ltimes \{S\}\right)\right)$$

这恰好与算法 6 的步骤一致。结合引理 5.7，$\Delta Q(D,\ t)$ 的所有结果均可枚举且不重复。

接下来分析时间复杂度。由于所有见证元组都可以存储在支持常数延迟枚举的数据结构（例如链表）中，因此每个 t'（第 1 行）都可以在 $O(1)$ 延迟内检索到。于是，只需证明对于每个 t'，$Q(D \ltimes t')$ 都可以在 $O(1)$ 延迟内枚举。注意，借助我们的哈希索引，子查询 $t' \bowtie \left(\bowtie_{i=1}^{k} V_r(e_i)\right)$（第 4 行）可以在 $O(1)$ 延迟内完成。对于剩余的子查询 $Q_{\mathcal{T}_{e_1}} \ltimes \{S\}$ 或 $Q_{\mathcal{T}_{e_i} - \mathcal{T}_{e_{i-1}}} \ltimes \{S\}$，我们调用过程 FullEnum（第 6–8 行），并且根据引理 5.4，所有查询结果都可以在 $O(1)$ 延迟内枚举。将这些子查询以笛卡尔积的形式组合，可在 $O(1)$ 延迟内得到查询结果，从而完成整个证明。□

C 本节缺少材料 6

定理 6.2 的证明。给定一个 OuMv 实例，我们分别用 R_3 和 (v_i, u_i) 对矩阵进行编码，并用 R_2 和 R_4 对向量进行编码。我们构造如下更新序列 S，用于 Q：(1) 添加一个元组

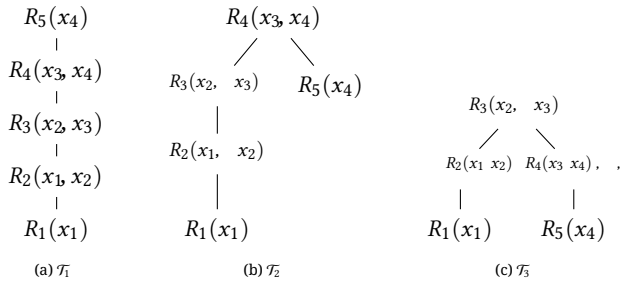


图13： $Q = R_1(x_1) \bowtie R_2(x_1、\ x_2) \bowtie R_3(x_2、\ x_3) \bowtie R_4(x_3、\ x_3) \bowtie R_5(x_4)$**的连接树**。

$t=(i,j)$，其生命周期为 $I(t)=[-k,k]$，对每一对 $(i,j)\in[n]\times[n]$，若 $M_{ij}\neq 0$；(2) 将一个生命周期为 $I(t)=[i-2k,i]$ 的元组 $t=(i)$ 添加到 R_1 和 R_5 中；(3) 对每一对向量 (v_i,u_i)，若 $v_{ij}\neq 0$，则将元组$t=(i,v_{ij})$（生命周期为 $I(t)=[i,i+2k]$）添加到 R_2 中；若 $u_{ij}\neq 0$，则将生命周期为 $I(t)=[i,i+2k]$ 的元组 $t=(u_{ij},i)$ 添加到 R_4 中；(4) 若枚举出查询结果，则对 $v_i^T M u_i$ 输出 true，否则输出 false；(5) 对下一对 (v_{i+1},u_{i+1}) 重复步骤 (3)-(4)，直到处理完全部 n 对向量。S 中的每个元组的生命周期都与 $2k$ 相同，因此它是一个先进先出序列。

注意，在 Q 的任意自由连通连接树 T 中，总存在一棵子树，其中 $R_1-R_2-R_3$ 或 $R_5-R_4-R_3$ 是一条叶到根路径。不失一般性，假设 $R_1-R_2-R_3$ 是一条叶到根路径。首先，对于每个元组 $t\in R_1$，$\lambda(t)=1$ 是 R_1 的叶节点。对于 $t=(i\ v_{ij})\in R_2$ $\tilde{I}(t)=[i\ i]$ $I(t)=[i\ i+2k]$，我们观察到，因为 $I(t')=[i-2k\ i]$ 且 $t'\in R_1$ 对某个元组成立。但在这种情况下，$\lambda(t)=1$ 仍然成立，因为不存在元组 $t'\in R_1$ 满足 $\tilde{I}(t')\subseteq[i,\ i]$。然而，对于每个元组 $t\in R_3$，$\lambda(t)=n$，都存在一个元组 $t'\in R_2$，使得对于每个 $i\in[n]$，$\tilde{I}(t')=[i,\ i]$。因此，S 在每棵自由连通连接树上的包围度为 $\lambda=\frac{n^2\cdot n+n^2\cdot 1}{n^2}=n$。

这个模拟的正确性显而易见。因此，如果存在一种数据结构，能够在 $O(\lambda^{1-\epsilon})$ 时间内完成更新，同时支持对任何先进先出序列在 Q 上进行 $O(\lambda^{2-\epsilon})$延迟枚举，那么就可以在 $O(n^2\cdot\lambda^{1-\epsilon}+n\cdot\lambda^{2-\epsilon})=$时间 $O(n^{3-\epsilon})$ 内解决 OuMv问题。注意，上述构造要求数据库的大小至少为 $n^2=\lambda^2$，因此 $\lambda\leqslant\sqrt{|D|}$。□

定理 6.6 的证明。接下来我们讨论索引的更新成本。如第 6 节开头所述，整个序列的总更新成本在渐近意义上由 P- 更新的更新成本主导，而后者进一步受所有计数器 count$[t]$ 能够发生变化的次数所限制。下面的引理将这一数量与更新序列的包围度联系起来。

引理 C.1. 对于任意元组 t，*count*$[t]$ *changes* $O(\lambda(t))$ *times*。

证明。元组 $t\in R_e$ 的状态变化分为以下两种情况之一：(1) 元组 t 被插入或删除；(2) 为 $e'\in$ Te 插入或删除某个元组 $t'\in R_{e'}$，且此更新传播至 t。注意，元组 t 在其生命周期内最多插入和删除各一次，因此其数量有界于 $O(1)$，相应开销计入 R-更新。接下来，我们将重点讨论第二种情况。

我们先考虑 e 在 T 中有一个子节点的情况。在这种情况下，count$[t]$ 的值会在 0 和 1 之间变化。注意，如果一次插入将 t 从 $R_e/V_s(R_e)$ 变为 $V_s(R_e)$，后续插入不会改变 t的状态，除非发生删除。考虑一组按顺序排列的 k 个不相交区间 $\tilde{I}_1$、$\tilde{I}_2$、$\cdots$、$\tilde{I}_k$，使得对于每个 $j\in[k]$，$\tilde{I}_j\in$ Ie、$\tilde{I}_j\subseteq I(t)$，并且不存在额外的区间 I满足 $\tilde{I}\subsetneq\tilde{I}_j$或 $\tilde{I}'\subseteq[\tilde{I}_j^+$、$\tilde{I}_{j+1}^-]$，其中任意 $j\in\{1,2,\cdots,k\}$均如此。这些 k 个区间中的每一个至多能改变 t的状态两次，因此它们总共至多能改变 t的状态 $O(k)$ 次。t的有效生命周期恰好刻画了这一数量。

接下来，我们考虑 e 有两个子节点 $e_1,e_2\in$T 的情况。类似地，考虑一组按顺序排列的 k 不相交区间 $\tilde{I}_1,\tilde{I}_2,\cdots,\tilde{I}_k$，使得 $I_j\in$ Ie，对于每个 $j\in[k]$ 均成立，并且不存在额外的区间 $\tilde{I}$，使得 $\tilde{I}\subsetneq\tilde{I}_j$ 或 $\tilde{I}\subseteq[\tilde{I}_j^+,\tilde{I}_{j+1}^-]$ 对任意 $j\in\{1,2,\cdots,k\}$ 均成立。我们可以得出以下两个观察结果：

(1) 对于任意 $\tilde{I}_j$，count$[t]$ 在 $\tilde{I}$ 内最多变化 2 次。
(2) 对于任意两个相邻区间 $\tilde{I}_j,\tilde{I}_{j+1}$，count$[t]$ 在它们的间隙中最多可以变化 4 次。

总之，我们可以得出结论：当存在两个子节点时，count$[t]$ 最多可以变化若干次。接 $6\cdot k=O(k)$ 下来，我们分别详细讨论 (1) 和 (2)。

对于 (1)，不失一般性，假设 $\tilde{I}_j\in$ Ie_1。根据有效生命周期的定义，在 $\tilde{I}_j$ 内，Te_1 的任何节点上都不可能发生插入或删除。不过，Te_2 的某个节点上仍可能在 I_j 内发生更新，进而改变 count$[t]$。我们再区分两种情况。如果由于来自 Te_2 的一次插入，count$[t]$ 从 1 变为 2，那么根据不存在任何 I 使得 $\tilde{I}\subsetneq\tilde{I}_j$ 这一事实，Te_2 的任何节点上都不能在 I_j 内发生删除。因此，在这种情况下，count$[t]$ 在 $\tilde{I}_j$ 内最多变化一次。否则，来自 Te_2 的一次删除会使 count$[t]$ 从 2 变为 1。随后我们转入第一种情况，count$[t]$ 最多还能变化一次。总之，count$[t]$ 最多可以变化两次。

对于（2），显然，在 I_j 的右端点和 I_{j+1}的左端点处，count$[t]$ 会因一个有效生命周期的删除和插入而各变化一次。同时，它们之间的间隙中不存在其他有效生命周期，因此，对于任意 $e_i\in\{e_1,\ e_2\}$，在 Te_i 中的一次插入之后，间隙内不会发生删除。这意味着，count$[t]$ 在它们之间的间隙中最多变化四次（即 $2\to1\to0\to1\to2$）。

最后，我们考虑一般情况：e在 T 中有多个子节点。在这种情况下，count$[t]$ 的值会在 0、1、$\cdots$、j之间变化，其中 j是 e的子节点数。通过扩展前面的两个观察，我们得出结论：count$[t]$ 最多变化 $3j\cdot k$ 次，其中 j可视为常数。对于 k的所有可能取值，我们观察到

$$k\leqslant\max_{\substack{\mathcal{J}\subseteq\tilde{I}_e\\ \forall I(t_1)\in\mathcal{J},I(t_1)\subseteq I(t)\\ \forall I(t_2),I(t_3)\in\mathcal{J},I(t_2)\cap I(t_3)=\varnothing}}1+|\mathcal{J}|=\lambda(t),$$

因此，count$[t]$ 最多会变化 $O(\lambda(t))$ 次。□

算法 3 的时间成本由 for 循环（第 2 行或第 7 行）的迭代次数决定。容易看出，在每次迭代中，某个元组 t 的 count$[t]$ 都会发生变化，因此运行时间可以由所有元组 t 的 count$[t]$ 变化总次数来界定。现在考虑一个包围度为 λ 的更新序列 S。由引理 C.1 可知，总更新成本为 $O\left(\sum_{t\in I}\lambda(t)\right)$，即摊销为 $O\left(\frac{\sum_{t\in_I}\lambda(t)}{|I|}\right)=O(\lambda)$。

综合以上结果，我们完成了定理 6.6 的证明。□

引理 6.8 的证明。在 [20], 中已证明，合取查询是 q-层次化的，当且仅当存在一棵“简单”的广义连接树，使得所有原始关系都是 T 的叶节点，并且每个对应于广义关系的内部节点 e'，对于它的每个子节点 e 都必须满足 $e'\subseteq e$。我们注意到，这样一棵简单的广义连接树本质上就是第 3 节中定义的高度为 1 的自由连通连接树。□

引理 6.9 的证明。给定一个高度为 2 的自由连通连接树 T，并考虑任意元组 $t\in R_e$ e $\tilde{I}(t)=$。如果是叶节点，，且 $[t^+\ t^-]$ $\lambda(t)=1$ e $\tilde{I}(t)\subseteq[t^+\ t^-]$ 如果是内部节点，，。

但这里，由于该连接树是高度为 2 的自由连通连接树，因此每个e的子节点必定是叶节点，所以对于$e' \in \mathrm{T}e$，每个元组 $t_1 \in R_{e'}$ 都满足 $\tilde{I}(t_1) = [t_1^+, t_1^-]$，并且不存在元组 t_2 使得 $t^+ < t_2^+$ 且 $t_2^- < t^-$。由于每个元组 t 都满足 $\lambda_{\mathcal{T}}(t) = 1$，根据定义，$\lambda_{\mathcal{T}}(S) = 1$。□

引理 6.10 的证明。由于每个元组 t, $\tilde{I}(t) = [t^+ + \infty]$ t $\lambda(t) = 1$ □, 都没有删除。因此，对于每个，根据定义，$\mathcal{T}$。

D SQL 查询

3 跳全连接查询。

选择G1.src 作为A，G2.src 作为B,G3.src 作为C，G3.dst 作为D来自G G1，G G2，G G3其中G1.dst = G2.src 并且G2.dst = G3.src并且对 (G3.dst) 执行过滤器

4 跳全连接查询。

选择G1.src 作为A，G2.src 作为B,G3.src 作为C，G3.dst 作为D,G4.dst 作为E来自G G1，G G2，G G3，G G4其中G1.dst = G2.src 并且G2.dst = G3.src 并且G3.dst = G4.src 并且基于 (G4.dst) 的过滤器

3 跳连接投影查询。

选择 G2.src 作为 B，G3.src 作为 C从 G G1，G G2，G G3其中 G1.dst = G2.src 且 G2.dst = G3.src

4 跳连接投影查询。

选择 G2.src 作为 B，G3.src 作为 C，G3.dst 作为 D从 G G1，G G2，G G3，G G4其中 G1.dst = G2.src 且 G2.dst = G3.src 且 G3.dst = G4.src 且 对 (G4.dst) 应用过滤器

2 梳状查询。

选择 G1.src 作为 A，G2.src 作为 B，G3.src 作为 C，G3.dst 作为 D从 G G1，G G2，G G3，V1，V2其中 G1.dst = G2.src 且 G2.dst = G3.src 且 V1.v = G1.src且 V2.v = G3.dst

星形查询。

选择 G1.src，计数(G1.dst，G2.dst，G3.dst，G4.dst)来自 G G1，G G2，G G3，G G4其中 G1.src = G2.src 并且 G1.src = G3.src 并且 G1.src = G4.src分组G1.src;

哑铃形完全连接查询。

选择 *从 G G1，G G2，G G3，G G4，G G5，G G6，G G6其中 G1.dst = G2.src并且 G2.dst = G3.src并且 G3.dst = G1.src并且 G5.dst = G6.src 并且 G6.dst = G7.src并且 G7.dst = G5.src 并且 G4.src = G3.dst并且 G4.dst = G5.src

哑铃形连接投影查询。

选择 G4.src，G4.dst来自 G G1，G G2，G G3，G G4，G G5，G G6，G G6其中 G1.dst = G2.src并且 G2.dst = G3.src 并且 G3.dst = G1.src 并且 G5.dst = G6.src并且 G6.dst = G7.src并且 G7.dst = G5.src并且 G4.src = G3.dst并且 G4.dst = G5.src

SNB 查询 1。

选择 p_人物编号，p_名字，p_姓氏，m_消息编号，k_人物 1 编号从 人物、消息、认识关系其中 p_人物编号 = m_创建者编号 且 k_人物 2 编号 = p_人物编号;

SNB 查询 2。

选择 k1.k_人物 1 编号，k1.k_人物 2 编号，k2.k_人物 2 编号，t_标签编号，m_消息编号 从 标签、消息、消息_标签、认识关系 1 k1、认识关系 2 k2 其中 m_消息编号 = mt_消息编号 且 mt_标签编号 = t_标签编号 且 k1.k_人物 2 编号 = k2.k_人物 1 编号 且 m_创建者编号 = k2.k_人物 2 编号 且 m_c_回复消息编号 为空 且 过滤器 在 (k1.k_人物 1 编号) 上

SNB 查询 3。

选择 k1.k_人物 1 编号，k1.k_人物 2 编号，k2.k_人物 2 编号，t_标签编号，m_消息编号 从 标签、消息、消息_标签、认识关系 1 k1、认识关系 2 k2 其中 m_消息编号 = mt_消息编号 且 mt_标签编号 = t_标签编号 且 k1.k_人物 2 编号 = k2.k_人物 1 编号 且 k2.k_人物 2 编号 <> k1.k_人物 1 编号 且 m_创建者编号 = k2.k_人物 2 编号 且 m_c_回复消息编号 为空 且 过滤器 在 (k1.k_ 人物 1 编号) 上

SNB 查询 4。

SELECT t_name, t_tagid, count(distinct m_messageid) FROM 标签, 消息, message_tag, 认识 WHERE m_messageid = mt_messageid AND mt_tagid = t_tagid AND m_creatorid = k_person2id AND m_c_replyof is NULL AND 过滤器 OVER (k_person1id) 分组 BY t_name, t_tagid