冗余度驱动的 Top-k 函数依赖发现

Xiaolong Wan、Xixian Han

arXiv:2601.10130v1 [cs.DB] 15 Jan 2026

摘要——函数依赖（FD）是关系数据库中的基本约束，可用于许多数据管理任务。大多数函数依赖发现算法会找出所有有效依赖，但这会带来两个问题。首先，计算成本高得难以承受：计算复杂度随元组数量呈二次增长，随属性数量呈指数增长，因此在大规模、高维数据上进行发现的速度很慢。其次，结果集可能非常庞大，难以识别有用的依赖。我们提出 SDP（选择性发现与剪枝），用于发现按冗余计数排名的 Top-k 个函数依赖。冗余计数衡量函数依赖所解释的重复信息量，并与存储开销和更新异常直接相关。SDP 利用冗余上界剪枝搜索空间。我们证明该上界具有单调性：添加属性会细化分区，从而降低上界。一旦上界低于 Top-k 阈值，便可跳过整个分支。我们通过三项优化改进 SDP：按分区基数对属性排序；利用分区基数矩阵中的成对统计信息收紧上界；以及使用全局调度器优先探索有希望的分支。在 40 多个数据集上的实验表明，与穷举方法相比，SDP 速度快得多且内存占用更少。

索引词——函数依赖、前k 个发现、数据冗余、剪枝策略。

一、引言

FUNCTIONAL 数依赖（函数依赖）是关系数据库 [1]中的基本约束。函数依赖 $X \to A$ 表示， X 的值决定 A的值；对于任意两个元组，如果它们在 X 上一致，那么它们在A上也必须一致，其中 X 是左侧（LHS）， A 是右侧（RHS）。函数依赖在数据管理中有许多实际应用，包括模式规范化 [2], 、数据集成 [3], 、数据清洗 [4], 和完整性维护 [5]。

鉴于其理论和实践意义，研究人员开发了许多算法来发现所有有效函数依赖 [6]–[9]。但即便付出了这些努力，函数依赖发现仍难以扩展，其计算复杂度相对于元组数量呈二次增长，相对于属性数量呈指数增长 [10]。因此，在大规模和高维数据上进行函数依赖发现的计算成本高得难以承受 [11]。更重要的是，即使计算可行，结果集本身的规模也往往会成为瓶颈。有效函数依赖的数量可能随着属性数量呈指数增长。例如，在包含 1,000 个元组和 109 个属性的 Flight 数据集上，发现了超过 980,000 个函数依赖 [8]。

作者来自中国哈尔滨工业大学计算机科学与技术学院。（电子邮箱：wxl@hit.edu.cn、hanxx@hit.edu.cn）稿件于202X年XX月XX日收到；于202X年XX月XX日修订。

（a）关系实例 r

ID	航班号	出发地	Dest
t_1	UA57	SFO	JFK
t_2	UA57	SFO	JFK
t_3	UA57	SFO	JFK
t_4	AA20	LAX	ORD
t_5	AA20	LAX	ORD

（b）冗余分析

函数依赖 1：标识符 →出发地

按左部（标识符）分组：

$\{t_1\}, \{t_2\}, \{t_3\}, \{t_4\}, \{t_5\}$

冗余度：0（无重复值）

函数依赖 2：航班号 → 出发地

按左部（航班号）分组：

$\{t_1, t_2, t_3\} \xrightarrow{yields}$ **2**冗余

$\{t_4, t_5\} \xrightarrow{yields}$ **1**冗余

总冗余度： $2 + 1 = \mathbf{3}$

图 1. 函数依赖冗余度的一个示例

如此庞大的输出会让用户难以应对，也使他们难以找出对下游任务真正有价值的函数依赖。

这促使我们不再局限于穷举式函数依赖发现，而是转向 top-k 范式，按照依赖关系的实际影响对其进行排名。数据冗余是一项自然的评判标准。对于函数依赖$X \to A$，冗余计数衡量的是按 X对元组进行分组后， A 中有多少重复值是由该依赖关系所蕴含的，进而反映该依赖关系能够解释多少重复信息 [12],[13]。与仅将函数依赖视为完整性约束不同，冗余计数将函数依赖与存储开销和更新异常等具体后果联系起来 [14]。因此，优先考虑冗余计数较大的函数依赖，可以让发现过程聚焦于更可能有助于规范化和数据清洗的依赖关系。

示例 1。图 1 展示了一个简单实例 r，其中 ID→ Origin 和 FlightNo → Origin 均成立。第一个依赖关系由唯一标识符决定。由于 ID 是键，ID →Origin 的冗余计数为 0，除了键本身已经蕴含的信息外，它几乎没有提供更多信息。相比之下，FlightNo → Origin 反映了数据中反复出现的规律，即具有相同航班号的元组也具有相同的出发地。因此，在这个示例中，其冗余计数为 3。这个例子说明了冗余度为何有助于筛选：它能突出那些反映重复结构、且可能对规范化和数据清洗具有意义的依赖关系。

为解决这些问题，我们专注于前-k 函数依赖发现。我们不再寻找所有有效函数依赖，而是仅返回具有最高冗余计数的k 依赖项。现有研究主要关注如何高效执行穷举式发现，或将冗余度视为相关性标准 [13]的概念性排序指标。它们并未解释如何利用冗余度实际引导发现过程。尽管更广泛的数据剖析领域（例如发现有意义的键 [15] 或增强实体的规则 [16], [17], ）日益转向选择性发现，但一种将冗余度直接整合到函数依赖发现过程中的方法

仍然缺失。

填补这一空白并非只是增加一个排序步骤。如果我们先枚举所有有效函数依赖，再对它们排序，就仍需承担穷举发现的全部开销，而这正是我们试图避免的。关键是在发现过程中使用冗余度指标，尽早剪除格中潜力较低的区域。这并不简单，因为冗余度计数是依赖数据的统计量。它取决于元组在 X 下如何分组，因此无法仅凭模式推断。在实践中，要可靠地计算该值，就必须检查分区并进行昂贵的验证，这使得在枚举候选项时难以将冗余度转化为低成本的剪枝测试。

为应对这一挑战，我们提出 SDP 算法（选择性发现与剪枝），采用基于超图遍历的枚举框架。SDP 并非枚举每一个有效函数依赖，而是优先搜索高冗余度候选项，并尽早剪除前景不佳的区域。其核心观察很简单：当我们通过添加更多属性来扩展左部时，元组分区会变得更细，即等价类的数量不会减少，因此函数依赖可能达到的冗余度不会增加。这为集合枚举格中任意子树的函数依赖候选提供了一个单调上界。SDP 将此上界与当前的 top-k 阈值 τ 结合起来执行剪枝操作。一旦某个节点的上界低于 τ，其所有后代节点都不可能进入 top-k 结果，因此可以跳过整个分支。由此，SDP 无需验证大量潜力较低的候选项，同时仍能返回 top-k 函数依赖。

我们通过三项优化进一步加速 SDP。首先，按照分区基数对属性排序，使搜索尽早找到高冗余候选项，并快速提高阈值 τ。其次，利用存储在分区基数矩阵（PCM）中的成对划分统计量，使估计的上界更加紧致；这些统计量能捕捉单属性计数无法反映的交互作用。第三，我们不再逐一探索右部树，而是使用全局调度器交错处理这些树，并始终推进当前最有希望的分支，从而在实践中加快 τ 的收敛。

总之，我们的贡献如下。

- 问题。我们研究基于冗余计数的 top-k 函数依赖发现，将相关性度量转化为具体的检索任务。
- 算法。我们提出 SDP，这是一种精确的 top-k 算法，它利用单调上界，在基于超图横截集的枚举框架内剪枝搜索空间。
- 优化。我们引入三项实用改进，以减少探索的节点数量和验证次数。
- 评估。在 40 多个真实数据集上的实验表明，SDP 的速度显著快于穷举基线，且内存效率更高。

本文其余部分安排如下。第二节介绍必要的预备知识，第三节综述相关工作。第四节介绍基线算法。第五节和第六节分别介绍核心 SDP算法及其三种优化策略。第七节给出全面的实验评估。最后，第八节总结全文。

II. 预备知识

本节将形式化定义函数依赖和冗余计数的概念，随后定义本文所要解决的 top-k 发现问题。

设 $R = \{A_1, A_2, \ldots, A_m\}$ 为由一组属性构成的关系模式。r 是 R 上的一个关系实例，由有限个 n 元组组成，其中每个元组都是一个有序值列表，对应于 R中的属性。对于任意元组 $t \in r$ 和属性 $A \in R$，$t[A]$ 表示 A 在 t中的值。同样，对于属性子集 $X \subseteq R$，$t[X]$ 表示将 t 投影到 X上所得的结果。

A. 函数依赖与冗余度

函数依赖（FD）刻画了属性集之间的约束。

定义 1（函数依赖）。给定属性集$X \subseteq R$ 和属性 $A \in R$，若且唯若对于所有元组对 t_1，$t_2 \in r$，$t_1[X] = t_2[X]$ 蕴含 $t_1[A] = t_2[A]$，则函数依赖（记为 $X \to A$）在关系 r 中成立。

其中，X 称为左部（LHS），A 称为右部（RHS）。若 $A \in X$，则函数依赖 $X \to A$ 是平凡的。若不存在真子集 $Y \subset X$ 满足 $Y \to A$，则该函数依赖是最小的。在本文中，我们关注非平凡且最小的函数依赖。

为了量化函数依赖的效用，我们采用等价类的概念。给定 r 和 $X \subseteq R$，元组 t 关于 X的等价类（记为 $[t]_X$）定义为 $[t]_X = \{u \in r \mid u[X] = t[X]\}$。所有此类等价类构成了 r 按 X划分所得的分区，记为 $\pi_X = \{[t]_X \mid t \in r\}$。我们将冗余计数作为 top-$k$ 函数依赖发现的效用目标，并根据左侧属性诱导的分区高效计算该计数。

定义 2（冗余计数）。对于关系 r 中有效的函数依赖 $X \to A$，冗余计数记为 $red(X \to A)$，定义为所有等价类中冗余出现次数的总和：

$$red(X \to A) = \sum_{[t]_X \in \pi_X} (|[t]_X| - 1) = |r| - |\pi_X|.$$

直观而言，在大小为 s的等价类中，由于左侧属性值相同，函数依赖要求右侧属性值也必须相同。因此，$s - 1$ 值实际上是冗余重复项。冗余计数越高，意味着该函数依赖约束了数据结构一致性的更大部分，因此可自然地作为相关性的替代指标。请注意，在函数依赖成立的前提下，$red(X \to A)$ 仅由左部的分区 π_X 和元组总数 $|r|$ 决定。

B. 问题陈述

经典的函数依赖发现侧重于穷举所有最小且非平凡的函数依赖。然而，这通常

会产生大量实用性各异的结果。为此，我们提出了 top-k 发现问题。

定义 3（Top-k 函数依赖发现）。给定一个关系实例r 和一个正整数 k，设 $\mathcal{F}$ 为 r中所有有效的极小且非平凡的函数依赖的集合，top-k 函数依赖发现问题旨在确定一个大小至多为 k的子集 $\mathcal{F}_k \subseteq \mathcal{F}$，使得 $\forall f \in \mathcal{F}_k$且 $\forall f' \in \mathcal{F} \setminus \mathcal{F}_k$: $red(f) \geq red(f')$。

我们严格考虑冗余度为正的函数依赖。冗余度为零的依赖（例如，对所有类都有$|[t]_X| = 1$ 的键依赖）不在考虑范围内，因为它们不具备数据约简潜力。这种表述使我们能够将计算资源集中于发现影响最大的知识。

III. 相关工作

A. 函数依赖发现

经典函数依赖发现的主要目标，是识别在给定关系实例上成立的全部极小且非平凡的函数依赖。尽管现有算法在候选项生成和验证机制方面各不相同，但它们都遵循穷举发现这一范式。这些算法可分为三类：面向属性、面向元组和混合方法。

属性导向方法。大量研究通过枚举属性格中的候选项，并尽可能进行剪枝来发现函数依赖。TANE [18] 是标准的逐层方法，它逐层遍历属性格，使用剥离分区检查候选项，并通过推理规则删除许多候选项。FUN [19] 和 FD Mine [20]保留相同的枚举方案，但引入了额外的剪枝条件。DFD [21] 则按右侧属性拆分搜索，针对每个右侧属性探索一个格，并采用随机深度优先遍历。还有一些变体依赖哈希 [22],：左侧属性值相同的元组会被放入同一个桶中，以加快函数依赖检查。这些方法在中等规模的模式上表现良好，但当 $|R|$ 较大时，格节点的数量会成为主要负担，使枚举和验证都变得十分昂贵。

元组导向方法。元组导向方法采用了不同的路径。它们根据元组对之间观察到的一致与差异来推导函数依赖。早期系统（如 FDEP [23] 和 Dep-Miner [24] ）依赖大量成对比较；随着 n 增大，这种做法很快会变得昂贵。FastFDs [25]以启发式深度优先方式搜索差异集，从而降低检查成本，避免遍历所有元组对。近来，FSC [9] 通过预先计算哪些元组对可以进行比较来减少比较次数，并直接从这些元组对的差异集中推导函数依赖。实践中，元组导向技术在宽模式上通常颇具优势，但在长表上竞争力较弱，因为元组对的数量会成为主要负担。

混合方法。混合方法的出发点是，基于属性和基于元组的技术各有互补优势。HyFD [7] 是一个典型代表。它首先对元组对进行采样，以生成有希望的候选项，然后切换到属性导向阶段对其进行验证。算法可根据验证效率，在两个阶段之间来回切换。DHyFD [26] 在 HyFD 的基础上加入了 FD 树变体和动态数据管理。最近，F DHITS [6] 将函数依赖发现转化为击中集枚举搜索，并结合并行探索与单遍验证，因此在现代场景中表现出色。

摘要。现有的函数依赖发现算法无论采用何种策略，都试图找出所有有效函数依赖。这会产生庞大的结果集，让用户应接不暇，也难以识别有用的函数依赖。

B. 规则发现中的 Top-k 范式

Top-k 函数依赖发现通过评分函数 [27]–[29] 对依赖项进行评分，并仅返回 k 个最有用的依赖项，从而解决这一问题。它不会用所有有效函数依赖让用户应接不暇，而是找出最相关的依赖项，且无需设置阈值。

规则与项集挖掘。关联规则的 Top-k 发现已得到充分研究，所用指标包括支持度和置信度 [30]–[32]。类似地，高效用项集挖掘依据模式的经济价值而非出现频率对其进行排序 [33]。这些技术专为事务数据设计。本文将 Top-k 发现应用于函数依赖；函数依赖是关系数据库中的完整性约束。

实体增强规则。Top-k 发现也已应用于实体增强规则（REE） [16], [17]。REE 使用主动学习，纳入用户对规则是否有趣且多样的反馈，而非仅依赖客观度量指标。这项工作表明，排序方法也适用于类似函数依赖的规则。

函数依赖。Wei 和 Link [13] 提出用冗余计数来衡量函数依赖的实际效用。该度量指标反映了函数依赖所解释的重复信息量，这与存储开销和更新异常直接相关。尽管他们提出按冗余度对函数依赖进行排序，却没有说明如何利用这一度量指标来指导发现。现有方法并未将冗余度纳入搜索过程，以获取排名前 k 的结果。

IV. 基线算法

据我们所知，目前尚无专门基于冗余计数进行前 k 个函数依赖发现的算法。因此，我们首先建立一种基线方法，称为 FDR（全量发现与排序）。

A. FDR 算法

FDR 采用两阶段后处理策略。它首先发现所有有效函数依赖，然后按冗余计数对其进行排序，以找出排名前 k 的结果。

1）阶段 1（穷举式函数依赖发现）：第一阶段从关系 r 中发现所有有效函数依赖，记为 $\mathcal{F}$。为建立强有力的基线，我们采用最先进的函数依赖发现算法 FDHITS [6], 。

下面我们将详细介绍 FDHITS 的内部机制。理解这些机制很重要，因为 SDP（第五节）基于这些机制实现高效的 top-k 发现。

FDHITS 的执行过程。FDHITS 扩展了 HPIVALID [34] 的基于超图的框架。它将函数依赖的无效性表述为差异集上的击中集问题。为避免计算所有成对差异所需的二次成本 $O(n^2)$，它采用基于采样的近似方法，随后进行精确验证。关键步骤如下。

- 差异集采样。给定关系 r 和模式 R，两个元组 t_1, t_2 的差异集为 $\Delta(t_1, t_2) = \{A \in R \mid t_1[A] \neq t_2[A]\}$。FDHITS 通过对每个属性的等价类分区执行加权采样，避免生成所有元组对。它会生成一个暂定超图 $\mathcal{H}' = (R, \mathcal{D}')$，其中 $\mathcal{D}'$ 包含从采样元组对中得到的差异集。

- 诱导子超图。对于特定的右侧属性$A \in R$，算法会构造一个诱导子超图 $\mathcal{H}'_A = (R, \mathcal{D}'_A)$，其中 $\mathcal{D}'_A = \{E \setminus \{A\} \mid A \in E, E \in \mathcal{D}'\}$。该结构编码了决定 A的函数依赖所需的约束。
- 最小击集枚举。候选左部 $X \subseteq R \setminus \{A\}$ 对于样本而言是有效的，当且仅当它命中 $\mathcal{H}'_A$ 中的每条边（即 $X \cap E \neq \emptyset, \forall E \in \mathcal{D}'_A$）。FDHITS 使用 MMCS 树搜索算法 [35]枚举最小命中集。请注意，该枚举是穷尽式的，并且在冗余度方面与顺序无关。
- 验证与细化。根据命中集 $X \to A$导出的每个候选函数依赖，都会针对完整数据集 r 进行验证，验证时使用关于 X 的等价类划分。如果验证无效，则会将新的差异集添加到 $\mathcal{H}'$，并细化搜索。

终止后，阶段 1 输出所有有效函数依赖的精确集合 $\mathcal{F}$。

2）阶段 2（后处理与排序）：在第二阶段，FDR 遍历 $\mathcal{F}$ 以计算冗余计数。关键在于，在阶段 1 的验证步骤中，FDHITS 会为每个候选左部 X构建等价类分区 π_X。我们利用这一点，立即计算冗余计数：

$$red(X \to A) = \sum_{[t]_X \in \pi_X} (|[t]_X| - 1).$$

最后，按照冗余计数的降序对所有函数依赖进行排序，并返回排名前k 的函数依赖。

B. 基线分析

由于能够保证正确性，FDR 基线提供了一个重要的参考点。由于阶段 1 是穷举式的，FDR 可以证明会返回精确的排名前k 结果。

局限性。FDR 的主要问题在于将发现与排序分开。阶段 1 可能会发现许多冗余度很低的函数依赖（例如近键依赖），而这些依赖随后会在阶段 2 被丢弃。这种两阶段方法在大型数据集上开销很大。为解决这一问题，SDP 将冗余度纳入搜索过程，以便及早剪除没有希望的分支。

V. SDP算法

SDP（选择性发现与剪枝）通过将基于冗余度的剪枝纳入搜索过程，避免穷举搜索。它基于 FDHITS框架 [6]，构建，该框架使用超图上的最小击集枚举。其核心思想是利用冗余上界。一旦某个候选项的上界低于 top-k 阈值，我们便跳过整个分支，不进行验证。

A. 上界估计策略

为了有效剪枝，我们需要一个能够快速计算的冗余度上界。我们利用分区大小的单调性来获得该上界。回顾一下，对于函数依赖 $X \to A$，冗余计数为 $red(X \to A) = n - |\pi_X|$。由于 n 为常数，最大化冗余度等价于最小化分区大小 $|\pi_X|$。

定理V.1. 对于任意属性子集 $X \subseteq R$ 和任意属性 $B \in X$，分区基数满足：

$$|\pi_B| \leq |\pi_X|.$$

证明。 $\forall t_1, t_2 \in r$，如果它们在 π_X 中属于同一个等价类（即 $t_1[X] = t_2[X]$），则由于 $B \in X$，它们必然蕴含 $t_1[B] = t_2[B]$。因此， π_X 中的每个等价类都是 π_B中某个等价类的子集。这意味着 π_X 是 π_B的细化，从而有 $|\pi_B| \leq |\pi_X|$。□

这种单调性自然可以推广到属性子集。

定理V.2. 对任意 $Y \subseteq X \subseteq R$，均有 $|\pi_Y| \trianglelefteq \pi_X|$。

证明. $\forall t_1, t_2 \in r$，若 $t_1[X] = t_2[X]$，则对于任意子集 Y，$t_1[Y] = t_2[Y]$。因此， π_X 细化了 π_Y和 $|\pi_Y| \leq |\pi_X|$。 □

此外，对于有效的函数依赖，右侧会对分区大小施加约束。

定理V.3. 对于有效的函数依赖 $X \to A$，均有 $|\pi_A| \trianglelefteq \pi_X|$。

证明. 根据函数依赖的定义， $t_1[X] = t_2[X]$ 蕴含$t_1[A] = t_2[A]$。因此，同一左部等价类中的元组映射到相同的右侧值，这意味着 π_X 细化了 π_A。 □

结合这些定理，我们推导出上界。由于 $|\pi_X| \geq \max_{B \in X} |\pi_B|$且 $|\pi_X| \geq |\pi_A|$（如果有效）， $|\pi_X|$ 的可能最小大小受这些下界的约束。因此，有效 $X \to A$的最大可能冗余计数（记为 $\overline{red}(X \to A)$）为：

$$\overline{red}(X \to A) = \min\left(n - \max_{B \in X} |\pi_B|, \quad n - |\pi_A|\right)$$

计算效率。我们在初始采样阶段为每个属性 $B \in R$ 预先计算 $|\pi_B|$。因此，计算任意候选项 X 的上界，只需遍历 X 中的属性，耗时为 $O(|X|)$。这比验证快得多，因此剪枝检查非常迅速。

B. SDP 执行过程

SDP 遵循差异集采样和超图构造的一般流程。然而，与基线不同，SDP 纳入了动态剪枝。为此，SDP 维护：

- 一个大小为 k 的最小堆 MH，用于存储目前已发现的前 k 个函数依赖。
- 动态剪枝阈值 τ，定义为 MH 中的最小冗余计数（如果 $|MH| < k$，则为 0）。

对于特定的 $A \in R$，SDP 会枚举诱导子超图 $\mathcal{H}'_A$ 上的最小命中集，并旨在识别右侧属性为 A 的有效函数依赖。与 FDR 不同，SDP 使用以下剪枝原则来缩小搜索空间。

剪枝原则。基于推导出的上界，我们制定了以下剪枝规则：

对于任意候选左部 X 和右侧 A，如果 $\overline{red}$（$X \to A$）$< \tau$，那么无论是 $X \to A$ 还是它的任何超集（特化项）可以是 top-k 结果。

定理 V.4. $\forall X \subseteq R$ 和 $B \in R \setminus (X \cup \{A\})$、$\overline{red}(X \cup \{B\} \to A) \leq \overline{red}(X \to A)$。

证明。令 $M(X) = \max_{B \in X} |\pi_B|$。由于 $M(X \cup \{B\}) = \max(M(X), |\pi_B|) \geq M(X)$，因此有 $n - M(X \cup \{B\}) \leq n - M(X)$。项 $n - |\pi_A|$ 独立于 X。对于任意实数 $u \leq v$ 及任意常数 c，有 $\min(u, c) \leq \min(v, c)$。取 $u = n - M(X \cup \{B\})$、$v = n - M(X)$ 和 $c = n - |\pi_A|$，可得 $red(X \cup \{B\} \to A) = \min(n - M(X \cup \{B\}), n - |\pi_A|) \leq \min(n - M(X), n - |\pi_A|) = \overline{red}(X \to A)$。 □

定理 V.4 证明了 $red(X \to A)$ 上界的单调性以及剪枝原则的正确性。

融合剪枝的树搜索。我们使用一种改编自 MMCS [35] 的集合枚举树搜索来执行最小击集枚举。树中的每个节点 o 都维护以下信息。

- $o.atts$：候选项当前的左侧属性。
- $o.cand$：用于扩展候选项的潜在属性。
- $o.uncov$：$\mathcal{H}'_A$ 中尚未被 $o.atts$ 覆盖的超边，用于识别最小击集。

至关重要的是，我们将剪枝操作整合到树搜索的两个阶段中（见算法1）：

1) 扩展前剪枝（第24行）。在通过添加 B ($B \in o.cand$) 来扩展 o 、生成子节点之前，我们会检查扩展当前左部是否有希望。如果 $\overline{red}(o.atts \cup \{B\} \to A) < \tau$，则不会将该子节点压入栈中。

2）验证前剪枝（第 6 行）。找到一个极小击中集后，我们会在调用开销较大的分区验证之前再次检查其上界。如果 $\overline{red}$（$o.atts \to A$）$< \tau$、$o.atts \to A$ 及其特化形式即使有效，也不可能成为前 k 个结果。因此，无需验证 $o.atts \to A$，并且可以将以 o 为根的子树整体剪枝。否则，如果验证确认 $o.atts \to A$ 有效，SDP 会更新 MH 并更新 τ（第 9 行至第 13 行）。

算法1 单个右侧属性的 SDP

输入：超图 $\mathcal{H}'_A$、目标 k、全局堆 MH。
输出：更新后的堆 MH。

```
1: 初始化栈 Q，并将根节点 rt 放入其中。
2: τ ← (MH.size() < k)?0 : MH.min().redundancy
3: 当 Q 不为空时，执行
4:     o ← Q.pop();    X ← o.atts
5:     如果 o.uncov = ∅ 和 o 是最小的，则
         // 情况 1：找到候选项，先检查界限。
6:       如果red(X → A) ≥ τ 则
7:           valid ← VALIDATE(X → A, r) {耗时的检查}
8:           如果 valid则
9:               rc ← n − |π_X|
10:              如果 rc > τ 则
11:                  MH.pop() 且 MH.add(X → A)
12:                  τ ← MH.min().redundancy
13:              结束如果
14:          else
15:              使用验证失败产生的新边更新 H'_A 。
16:              使用新边重新评估 o.uncov 。
17:              跳转到第 21 {行，现在按情况 2 处理}
18:          结束 if
19:      结束 if
20:  else
         // 情况 2：扩展节点
21:      e_m ← arg min_{e∈o.uncov} |e ∩ o.cand|
22:      对每个 B ∈ e_m ∩ o.cand 执行
23:          cld ← 生成子节点(o, B)
24:          如果 red(cld.atts → A) ≥ τ 则
25:              Q.push(cld) {剪枝分支}
26:          结束 if
27:      结束 for
28:  结束 if
29: 结束 while
30: 返回 MH
```

算法 2 SDP算法（总体）

要求：关系 r、整数 k。确保：前 k 个函数依赖。1：计算所有 $A \in R$ 的分区 π_A。2：$\mathcal{H}' \leftarrow$←差异集采样(r) 3：$MH \leftarrow$←新建最小堆(k) 4：对于每个属性 $A \in R$ 5：$\mathcal{H}'_A \leftarrow$ ←诱导超图($\mathcal{H}'$, A) 6：$MH \leftarrow$←单个右侧属性的 SDP($\mathcal{H}'_A$, k, MH) 7：结束循环 8：返回 MH

算法 2 概述了 SDP 的全局过程。通过在不同右侧属性的迭代过程中维护一个全局堆 MH，阈值 τ 会单调上升。这意味着，较早为某个属性发现的函数依赖有助于剪枝后续属性的搜索空间，从而使算法随着运行不断加快。正如定理 V.5 所证明的，当 SDP 终止时，MH 中的函数依赖是前 k 个结果 $\mathcal{F}_k$。

定理 V.5。SDP算法识别出的集合 $\mathcal{F}_k$ 满足前 k 个发现要求：$\forall f \in \mathcal{F}_k$ 和 $\forall f' \in \mathcal{F} \setminus \mathcal{F}_k$，$red$（$f$）$\geq red$（$f'$）。

证明。根据定理 V.4，随着左部中添加属性，上界 $\overline{red}$ 单调不增。这意味着，对于以当前候选项 $X \to A$为根的子树中的任意函数依赖 f'，其真实冗余计数受 $\overline{red}(X \to A)$ 的界定。如果满足剪枝条件 $\overline{red}(X \to A) < \tau$，则可推出 $red(f') < \tau$，而被剪枝分支中的任何 f'都不可能属于全局 top-k 集合。因此，不会遗漏任何潜在的 top-k 结果，从而确保

$\mathcal{F}_k$的全局最优性。 □

VI. 算法优化策略

基本的 SDP算法能够正确运行，但速度可能较慢。其速度取决于两个因素：阈值 τ增长得有多快，以及上界有多紧。我们通过三项改进来优化这两个因素。

加速 τ。我们通过优先搜索格中冗余度较高的部分，使 τ 更快增长。这需要使用属性排序（第六节-A）和全局引导（第六节-C）。

更紧的界限。我们利用分区基数矩阵中的成对属性信息，计算更紧的上界 $\overline{red}^*$ （第六节-B）。

A. 启发式属性访问顺序

优先搜索高冗余候选项能提升剪枝效果。随机顺序无法使 τ 快速增长。我们采用全局排序 σ，优先搜索格中冗余度较高的部分。

启发式方法。我们按分区大小的升序对所有属性进行排序。令 σ为 $\{1,\dots,m\}$的一个排列，使得:

$$|\pi_{A_{\sigma(1)}}| \le |\pi_{A_{\sigma(2)}}| \le \cdots \le |\pi_{A_{\sigma(m)}}|$$

我们通过两种方式使用这一排序，以更快地发现高冗余函数依赖。由于 $\overline{red}(X \to A) = \min(n - \max_{B\in X}|\pi_B|, n - |\pi_A|)$，我们希望使两项都保持较大：为第二项选择较小的 $|\pi_A|$，并添加较小的 $|\pi_B|$，使第一项保持较高。

- 右侧选择（外层循环，算法 2 的第 4 行）。我们按顺序遍历右侧属性 A σ。通过先选择较小的 A $|\pi_A|$，项($n-|\pi_A|$)会达到最大值。由于有效函数依赖必须满足 $|\pi_X| \ge |\pi_A|$，分区较小的属性更有可能成为高冗余函数依赖的右侧。较小的 $|\pi_A|$ 使这一约束更容易满足，从而更有可能产生高冗余函数依赖。
- 左侧扩展（内层循环，算法1第22行）。在 MMCS 树中，我们通过添加属性 B 来扩展候选左部 X，顺序也为 σ。添加 $|\pi_B|$ 较小的属性可将 $\max_{B\in X}|\pi_B|$的增幅降至最低，从而保持较高的上界。这使搜索倾向于选择等价类较少的左部组合，进而产生较高的冗余度。

优先搜索上界较高的候选项，会使 τ 增长得更快，从而为后续搜索剪去更多分支。请注意，这种排序纯粹是启发式的，不会影响 SDP 的正确性。

B. 使用属性对进行更紧密剪枝

单属性上界可能不够紧。例如，属性 B 和 C 的分区可能各自都很小，但它们的组合 BC 却可能构成一个键（分区很大）。为了捕捉这类交互作用，我们纳入属性对的分区基数。

成对上界。利用性质 $|\pi_{BC}| \ge \max(|\pi_B|, |\pi_C|)$，我们细化候选函数依赖 $X \to A$ 的上界:

$$\overline{red}^*(X \to A) = n - \max\left(|\pi_A|, \max_{B\in X}|\pi_B|, \max_{BC\subseteq X}|\pi_{BC}|\right).$$

显然，$\overline{red}^* \le \overline{red}$，可提供更强的剪枝能力。

分区基数矩阵（PCM）。计算所有 $\binom{m}{2}$ 两两分区的开销过高（$O(m^2n)$）。我们通过分区基数矩阵（PCM）解决这一问题，该矩阵采用混合策略。

1）选择性预计算。我们仅按照启发式顺序，为前 d 个属性预计算两两基数 σ。设置 $d = \lceil\sqrt{m}\rceil$ 可确保预计算成本与对所有属性进行一次遍历的成本大致相当，从而满足预算约束。2）机会式更新。在搜索过程中，每当验证一个具有 2 个属性左部的候选函数依赖（例如 $BC \to A$）时，$|\pi_{BC}|$ 也会作为副产品被计算出来。我们将该值缓存在 PCM 中。这使 PCM 能够随着算法的推进不断学习并变得更加稠密，从而以零额外成本改善对后续分支的剪枝效果。

增量式界限计算。对 $\overline{red}^*$ 进行朴素评估需要 $O(|X|^2)$。通过利用集合枚举树结构，我们将其优化为 $O(|X|)$。设子节点 o_{child} （属性集 Y）通过添加属性 C（即 $Y = X\cup\{C\}$）由父节点 o_{parent} （属性集 X）生成。我们维护变量 $M^*(Y)$，用于跟踪在 Y 内观察到的最大分区大小（同时考虑单个属性和可用的属性对）。该变量按增量方式更新:

$$M^*(Y) = \max\left(M^*(X),\ |\pi_C|,\ \max_{B\in X}\text{PCM}(B,C)\right),$$

其中，PCM(B,C) 在 PCM 中有值时返回 $|\pi_{BC}|$，否则返回 0。这只需遍历 X 中现有的属性，从而确保检查的复杂度保持为线性 $O(|X|)$。

相应的上界为 $\overline{red}^*(X \to A) = n - \max(M^*(X), |\pi_A|)$，且 $\overline{red}^*(X \to A) \le \overline{red}(X \to A)$。

定理 VI.1. 对于任意两个属性集 X 和 Y，其中 $X \subseteq Y$ A $Y \to A$ $\le$，以及一个固定的右侧，成立: $\overline{red}^*(\ )(\ )$。

证明。回顾一下，$M^*(X)$ 表示 X 中单个属性及可用属性对的最大划分基数。由于 $X \subseteq Y$，计算 $M^*(X)$ 时考虑的组件集合是计算 $M^*(Y)$ 时所考虑组件集合的子集，因此 $M^*(X) \le M^*(Y)$。于是，$\max(M^*(X), |\pi_A|) \le \max(M^*(Y), |\pi_A|)$。从 n 中减去后，不等式方向反转，从而得到所述结论。 □

定理 VI.1 保证了剪枝策略的安全性。具体而言，如果某个节点因 $\overline{red}^* < \tau$ 而被剪枝，则可以跳过其后代节点，因为它们的冗余度上界不可能超过 $\overline{red}^*$。由于 $\overline{red}^* \le \overline{red}$ 始终成立，SDP 在继承基础框架正确性的同时，实现了更强的剪枝能力。

接下来，我们使用 $\overline{red}^*$ 作为 SDP 中的默认上界，以最大限度地提高剪枝效率。

C. 通过全局最佳优先搜索实现加速收敛

逐个处理右侧属性可能会在没有 top-k 函数依赖的右侧属性上浪费时间，而其他右侧属性可能有更好的候选项。为解决这一问题，我们采用全局最佳优先搜索策略。

我们不再按顺序处理 m 集合枚举树，而是使用全局优先队列对其进行调度。我们维护 m 个独立的栈 Q_1，……，Q_m，其中 Q_i 跟踪右侧 A_i的深度优先遍历。全局优先队列 $\mathcal{P}$管理 Q_1，……，Q_m中的搜索前沿（即顶部节点）。 $\mathcal{P}$ 中的每个元素都是一个元组（$score$，i），其中 i 是栈索引， $score$ 是 Q_i中顶部节点的 $\overline{red}^*$ 。

执行流程。调度器的工作方式如下。

1) 从 $\mathcal{P}$中提取得分最高的元组。设该元组为（$\overline{red}^*$，i），表示所有右侧中的最佳活动分支。2) 从 Q_i 中弹出并处理顶部节点（验证候选函数依赖，或将其子节点推入 Q_i 以扩展该节点）。3) 如果 Q_i 不为空，则为新的顶部节点计算 $\overline{red}^*$ ，并将更新后的元组重新插入 $\mathcal{P}$。

这确保算法始终针对所有右侧属性中的最佳候选项进行处理。高冗余度的函数依赖会更早被发现，从而使 τ 增长得更快，并在所有树中剪掉更多分支。

定理 VI.2。给定一个上界 $\overline{red}^*$ 和一个动态更新的阈值 τ，任何遍历所有满足 $\overline{red}^*$（$X \to A$）$\geq \tau$ 的候选节点 X 的遍历策略，都能保证返回正确的 top-k 函数依赖集合。

证明。设 $\mathcal{F}_k$ 为真正的 top-k 函数依赖集合，τ_{final} 为 $\mathcal{F}_k$ 中第 k 高的冗余计数。由于 τ 在整个过程中单调不减，因此 $\forall f \in \mathcal{F}_k$ 的分数在任意时间步都满足 red（f）$\geq \tau_{final} \geq \tau_{current}$。根据安全上界的定义，$\overline{red}^*$（$f$）$\geq red$（$f$）$\geq \tau_{current}$。因此，无论遍历顺序如何，$f$ 都不会被剪枝。由于该策略保证访问所有未剪枝的节点，每个 $f \in \mathcal{F}_k$ 最终都会得到验证并纳入结果。 □

值得注意的是，全局最佳优先搜索会因维护 m 搜索树而占用更多内存，但其加速效果值得付出这一代价。由于 τ 增长得更快，SDP 可以更早剪除更多分支，从而缩小整体搜索空间。正如定理 VI.2所证明的，全局调度器只会改变遍历顺序，不会影响正确性。最终的 top-k 集合与顺序方法得到的相同。

VII. 实验评估

本节在不同维度和规模的真实数据集上评估 SDP。实验将 SDP 与基线 FDR 进行比较，并开展消融研究，以衡量各项优化的贡献。本节首先介绍实验设置，然后展示并分析实验结果。

A. 实验设置

我们使用 Java（jdk-21 windows-x64）实现这些算法。实验在一台 DELL OptiPlex 7010MT 工作站上运行，其配置为 Intel(R) Core(TM) i9-13900CPU @ 2.00GHz（24 核）、64 GB 主内存、4 TB 硬盘，以及 Windows 11（64 位）操作系统。

真实世界数据集来自 UCI机器学习知识库[1] 和 Kaggle[2]。为确保比较公平，我们通过 -Xmx 参数为每种算法分配 56 GB 内存（总内存为 64 GB）。

NULL语义。在真实世界数据集中，空值被广泛用于表示缺失或不可用的条目。对空值的解释会影响发现结果。在我们的实验中，我们采用标准NULL-EQ语义，将所有空值视为彼此相等，但与任何非空值不同 [34], [36]。

我们使用五项指标评估算法性能。

运行时间。我们记录每个数据集从开始发现到完成的总实际运行时间。

峰值内存消耗。我们使用 JVM 的 MemoryMXBean接口测量 SDP 和 FDR 的峰值内存消耗。

搜索剪枝率 (PR_S)。此指标衡量集合枚举树中的剪枝情况。设 #Nodes(SDP) 为 SDP 访问的节点数，# TotalNodes(FDR) 为穷举基线访问的节点数，且 PR $_S = 1 - \#$Nodes(SDP) $/\#$TotalNodes(FDR)。此指标反映 SDP 避免搜索的搜索空间比例。

候选剪枝率 (PR_C)。此指标衡量候选验证阶段的剪枝情况。设 #AllCandidates(FDR) 为 FDR 考虑的函数依赖候选数，# VerifiedCandidates(SDP)为 SDP 验证的候选项数，且 PR $_C = 1 - \#$VerifiedCandidates(SDP) $/\#$AllCandidates(FDR) 。此指标表明 SDP 在验证前过滤无关候选项的效果。

结果效用（LIFT_k）。LIFT_k 通过前-k 个函数依赖的平均冗余度与其余函数依赖的平均冗余度之比（$\bar{s}_{\text{top}}/\bar{s}_{\text{rest}}$）来衡量前 k 个函数依赖的质量，其中 $\bar{s}_{\text{top}} = \frac{1}{k}\sum_{f\in\mathcal{F}_k} red(f)$，且 $\bar{s}_{\text{rest}} = \frac{1}{|\mathcal{F}\setminus\mathcal{F}_k|}\sum_{f\in\mathcal{F}\setminus\mathcal{F}_k} red(f)$。$\text{LIFT}_k > 1$表示前-$k$个函数依赖的平均冗余度远高于其余函数依赖。

B. 总体性能比较

表 I 总结了 SDP 和 FDR 在公共知识库中 40 多个真实世界数据集上的实验结果。这些数据集包含 8 至 109 个属性，元组数量从数百个到数千万个不等，覆盖了广泛的规模范围。

每种算法均在相同的 56 GB 内存限制下运行。如果内存消耗超过此限制，则将该情况标记为 OM（内存溢出错误）。对于所有数据集和算法，我们均使用 k =10，这是前-k 发现中一种典型的选择。不同 k 取值的敏感性将在第七节-E 中讨论。

在高维和大规模数据集上，SDP 比 FDR 快得多，实现了 10-1000× 的加速比，同时占用

[1]http://archive.ics.uci.edu/ [2] https://www.kaggle.com/

表 I 真实数据集上的实验结果。

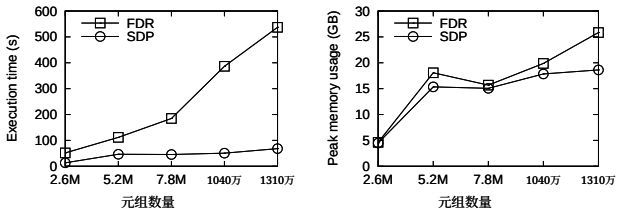
数据集	$\|r\|$	$\|R\|$	#FDs	函数依赖数量$_{>0}$	FDR		SDP		加速比	PR_S	PR_C	$LIFT_k$
					时间[s]	内存[G]	时间[秒]	内存[G]				
广告表	268255	24	3019	2996	56.054	15.8	8.575	3.5	6.537	0.831	0.872	7.511
成人	32561	15	78	78	0.314	0.093	0.284	0.093	1.106	0.0196	0.0672	18.897
Airbnb 开放数据	102599	26	20149	12064	76.96	16.615	38.257	8.062	2.012	0.835	0.850	10.299
航空公司数据集	98619	15	864	164	1.623	2.344	1.44	2.313	1.127	0.594	0.773	199.646
航空公司航班	300153	12	43	31	1.282	0.584	0.995	0.474	1.288	0.266	0.451	3.616
aggnycyellowtaxi	921373	18	2261	2189	140.529	5.375	118.771	8.563	1.183	0.348	0.414	18.009
atom	160000	31	7273	5135	43.173	5.5	0.978	0.519	44.144	0.968	0.990	11.401
Avila	10430	11	336	79	0.156	0.093	0.146	0.093	1.069	0.676	0.817	118.439
银行交易	1048567	9	59	26	4.695	1.125	3.701	1.438	1.269	0.339	0.443	361.391
blerssi	5191	15	14	1	0.095	0.005	0.046	0.0052	2.065	0.086	0.6	∞
碳纳米管	10721	8	72	24	0.115	0.0313	0.078	0.0625	1.474	0.407	0.542	14.944
census	199523	42	-	-	–	OM	21.213	6.362	–	–	–	–
conflongdemo jsi	164860	8	51	5	0.268	0.158	0.188	0.156	1.426	0.323	0.522	∞
covtype	581012	55	–	–	–	OM	53.27	24.104	–	–	–	–
信用卡 2023	568630	31	814	784	3.129	3.967	2.771	4.031	1.129	0.901	0.985	1.039
ditag 特征	3960124	13	58	38	17.534	8.5	12.56	7.767	1.396	0.4	0.658	16.559
eshopclothing	165474	14	22	14	0.582	0.178	0.448	0.151	1.299	0.243	0.386	5.124
fd-reduced-30	250000	30	89571	23	49.26	5.094	20.459	4.937	2.407	0.948	0.958	7027.556
flight	1000	109	982631	734583	195.179	2.5	0.166	0.093	1175.8	0.9999	0.99999	21.167
ftda	13094522	11	57	47	536.898	25.843	68.204	18.625	7.872	0.137	0.341	172.586
性别差距	4746	21	5520	3162	0.866	1.156	0.447	0.25	1.937	0.771	0.809	202.858
全球健康统计	1000000	22	91658	3480	886.744	5.875	864.045	10.656	1.026	0.701	0.839	128.968
grocery sales	6758125	9	17	3	9.788	7.906	9.795	8.134	0.999	0.363	0.5	∞
har70plus	2259597	8	9	2	6.013	1.581	4.372	1.596	1.375	0.25	0.478	∞
harth	5603043	8	58	18	12.26	8.005	12.808	9.531	0.957	0.196	0.396	47043.7
健康追踪	365000	12	96	2	0.945	0.649	0.607	0.546	1.556	0.64	0.779	∞
homec	503911	32	–	–	–	OM	140.398	18.046	–	–	–	–
马	300	28	128726	128565	23.494	1.375	5.608	0.718	4.189	0.735	0.744	13.087
人力资源跨国公司数据	2000000	13	251	32	20.382	18.781	20.179	14.125	1.01	0.495	0.668	3208927
HT传感器	928991	12	949	467	23.613	8.531	28.378	8.718	0.832	0.459	0.598	2.174
互联网防火墙	65532	12	88	88	0.754	0.156	0.251	0.0732	3.004	0.508	0.695	2.578
肺癌	890000	17	2120	984	102.881	5.187	77.09	8.781	1.334	0.410	0.531	15.869
MetroPT3	1516948	17	345	313	69.645	20.687	58.403	12.299	1.192	0.203	0.318	2.767
采矿过程	737453	24	38125	38125	567.004	11.031	458.145	11.594	1.237	0.652	0.705	217.339
电影数据集	999999	17	4073	91	31.219	12.719	23.299	13.718	1.339	0.638	0.792	333969
OpenFDA药物	1000	39	7789	7502	0.94	0.312	0.084	0.0312	11.191	0.991	0.998	4.534
ReactionNetwork	65554	29	42960	42954	137.882	44.888	1.669	0.764	82.614	0.983	0.991	1.347
Kaggle拼车	693071	57	–	–	–	OM	129.544	36.171	–	–	–	–
sg bioentry	184292	9	19	4	0.407	0.175	0.278	0.181	1.464	0.667	0.75	∞
超导性	21263	82	–	–	–	OM	0.952	0.2897	–	–	–	–
tnweather 1.8m	1888422	14	103	40	11.433	10.781	9.268	6.531	1.233	0.289	0.5026	428306
wecsdataset	287999	49	61207	61207	85.397	42.94	27.261	9.963	3.133	0.899	0.959	1.0004
YouTube 推荐	1000000	14	328	124	16.286	6.593	15.414	8.593	1.056	0.186	0.473	59941
Zomato	51717	17	618	532	1.621	2.3125	1.322	2.343	1.226	0.374	0.491	17.204

更少的内存。例如，在 flight 和 ReactionNetwork 上，SDP 的加速比分别达到 1175× 和 82×。这是因为 τ 增长迅速。SDP 的全局调度器优先发现高冗余函数依赖，使 τ 增长速度快于 FDR 中的顺序搜索。借助单调上界，SDP 随后会尽早剪枝大量分支。搜索剪枝率（PR_S）表明，在 flight 和 OpenFDA药物等数据集上，该比率超过 0.99，这意味着 SDP 访问的节点数不到 FDR 所需节点数的 1%。

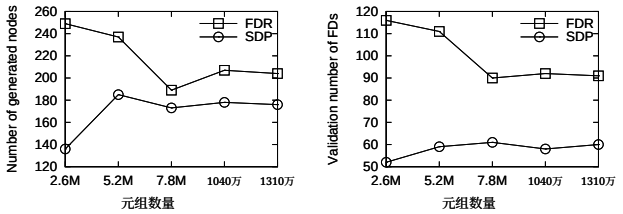
FDR 在五个数据集上因内存耗尽而无法运行：census、covtype、homec、Kaggle拼车和超导性。这是因为 FDR 会为所有候选项物化分区，而这在高维数据上会占用过多内存。SDP 通过提前剪枝处理了所有这些数据集：它会在物化分区之前跳过候选项。在宽模式数据集上，候选剪枝率（PR_C）通常超过 0.95，这表明 SDP 跳过了大多数低冗余度候选项。这使内存使用量保持在较低水平；内存使用量取决于发现的高冗余函数依赖数量，而不是模式总大小。

$LIFT_k$ 也表明冗余度是一种有效的度量指标。在 conflongdemo jsi 和 har70plus 等一些数据集上，$LIFT_k$ 达到无穷大，这意味着只有排名前 k 的函数依赖具有非平凡的冗余度，而其余函数依赖的冗余度接近于零。这说明冗余度能够有效地区分重要的函数依赖与低冗余度函数依赖。通过聚焦于冗余度，SDP 能够高效地找出最有用的函数依赖。

低复杂度场景下的性能开销。SDP 在高维数据集上实现了显著的加速比，但在一些低维数据集（例如 ht sensor、harth、grocery sales）上，性能与 FDR 相当或略慢。例如，在 ht sensor（$|R| = 12$）上，SDP 比 FDR 慢约 4.7 秒。这是因为 SDP 的优化（例如预计算 PCM 并维护全局优先队列）会带来额外开销。当格较小时，穷举搜索只需几秒就能完成，这些开销便会变得明显。不过，这点额外开销是可以接受的。SDP 能避免 FDR 在 flight 和 superconductivity 等复杂数据集上遭遇的失败。

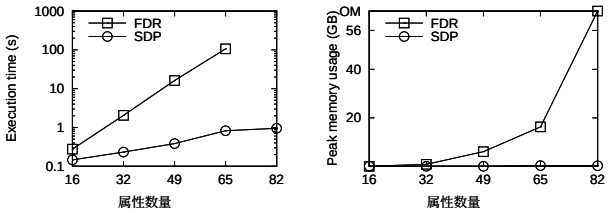


(a) 执行时间　(b) 峰值内存使用量

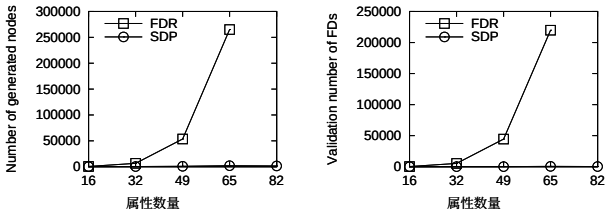


（c）生成节点数量　（d）函数依赖验证数量

图2. 元组数量可扩展性分析示意图（ftda数据集）。



(a) 执行时间　(b) 峰值内存使用量



(c) 生成节点数量　(d) 函数依赖验证数量

图3. 属性数量可扩展性分析示意图（超导数据集）。

C. 元组数量可扩展性分析

我们评估了随着元组数量增加，SDP 的可扩展性。我们使用最大的 ftda 数据集，该数据集包含超过 1300 万个元组和 11 个属性。我们按原始大小的 20%、40%、60%、80% 和 100% 创建 ftda 的前缀子集。这展示了在保持数据特征相似的同时，数据增长对性能的影响。使用 $k = 10$，我们在每个子集上运行 SDP 和 FDR，并测量它们的运行时间。

图 2 展示了随着元组数从 260 万增加到 1310 万，SDP 在 ftda 上的可扩展性。我们从三个方面分析结果。

运行时间效率。图 2(a) 显示，FDR 的运行时间随数据量线性增长，而 SDP 的增长速度低于线性，在 1310 万个元组时达到 7.87× 的加速比。这是因为我们的优化能够有效协同工作。属性排序和全局搜索使 τ 快速增长，而 PCM 则收紧了上界。这些优化共同降低了分区细化的成本。尽管随着数据量增长，验证单个候选项的成本会增加，但 SDP 减少了需要验证的候选项数量，从而使总运行时间保持较低。

FDR 的节点数为何会减少。图 2(c) 和 (d) 展示了一种有趣的规律：随着元组数从 260 万增加到 780 万，FDR 的节点数逐渐减少，之后趋于稳定。这是由数据分布收敛造成的。在较小的样本（260 万）中，伪依赖常常看起来有效，迫使 FDR 在格中进行更深层的搜索。当数据量增加到 780 万时，更多反例出现，使无效函数依赖能在搜索树中更早被剪枝。相比之下，SDP 保持稳定，只需验证 50–60 个候选项。这表明，无论数据量大小，SDP 的上界都能发挥良好作用。与对反例敏感的函数依赖有效性不同，随着数据量增长，前 k 个高冗余函数依赖保持不变，因此 SDP 能够维持一致的剪枝效果。

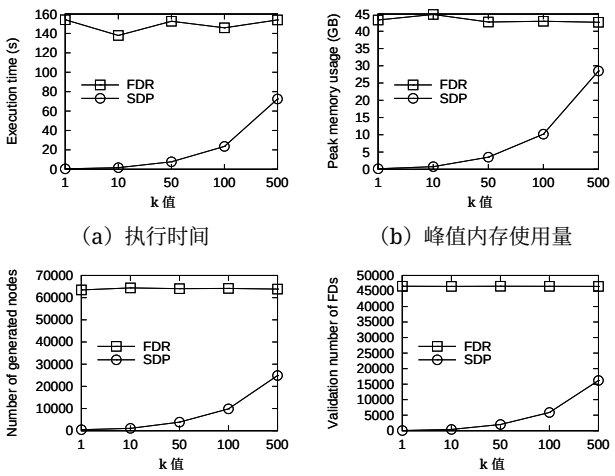
内存使用量。图 2(b) 显示，SDP 比 FDR 使用更少的内存，在 1310 万个元组时节省了 7 GB。SDP 的内存使用量取决于两个因素。首先，全局搜索需要维护 {v1} 个并发树的执行状态（栈和优先队列），因此会产生一定开销。其次，积极剪枝避免了大量中间分区的物化。结果表明，剪枝节省的内存远远超过其带来的开销，因此 SDP 避免了影响 FDR 的内存溢出故障。

D. 按属性数量进行可扩展性分析

我们评估 SDP 随着属性数量增加的扩展能力。这项工作颇具挑战，因为搜索空间会随属性数量呈指数级增长。我们使用超导数据集（含 82 个属性），这是我们属性数量最多的数据集之一。我们分别选取 20%、40%、60%、80% 和 100% 的属性创建子集。使用 $k = 10$，我们在每个子集上运行 SDP 和 FDR。图3展示了各算法随属性数量增加的扩展情况。

FDR 为何失效。图3(a)显示，当属性数量从 16 增加到 82 时，FDR 的运行时间增长速度远快于 SDP。FDR 的运行时间呈指数级增长，并在属性数量达到 82 时耗尽内存。这是因为搜索空间呈指数级增长（$2^{|R|}$）。每增加一个属性，搜索空间就会扩大一倍，导致节点数量从 419 个（16 个属性时）增至 265 万个（65 个属性时）。相比之下，SDP 的运行时间增长缓慢得多，这是因为其优化措施降低了组合爆炸的影响。属性排序和全局搜索使 τ 快速增长，而 PCM 则收紧上界。综合而言，这些方法使 SDP 只需搜索高冗余度候选项，从而避免低冗余组合呈指数级增长。

内存使用量。图 3(b) 展示了 FDR 为何在 82 个属性时失效。FDR 的内存使用量从 16 个属性时的 0.09 GB 激增至 65 个属性时的 16.3 GB，随后在 82 个属性时超过 56 GB 的限制。这是因为 FDR 会为所有候选项物化分区以验证函数依赖。相比之下，SDP 使用的内存少得多，始终低于 0.38 GB。这是因为 SDP 的剪枝判定大多数候选项无法进入 top-k 结果，因此 SDP 从不分配



（a）执行时间　（b）峰值内存使用量

(c) 生成节点数量　(d) 函数依赖验证数量

图4. k 对结果的影响示意图（ReactionNetwork 数据集）。

内存给它们。即使属性很多，这也能避免内存故障。

SDP 为何验证更少的候选项。图3(c)和(d)展示了 SDP 速度更快的原因。随着属性数量从 16 增至 65，FDR 验证的函数依赖数量从 329 个增至 219,855 个。相比之下，SDP 仅验证 31 至 246 个函数依赖。即使有 82 个属性，SDP 也只生成 1,518 个节点，并且仅验证 246 个候选项。这表明，尽管函数依赖的总数呈指数级增长，SDP 需要验证的候选项数量却增长缓慢。PCM 帮助 SDP 实现了这一点。随着维度增加，PCM 中的成对信息帮助 SDP 更早地剪枝候选项。这表明 SDP 能够找到排名前 k 的高冗余函数依赖，同时跳过数十万个低冗余函数依赖。

E. 对 k 的参数敏感性

我们研究 SDP 的性能如何随 k（要检索的排名靠前的函数依赖数量）变化。我们使用 ReactionNetwork 数据集，其中包含 42,954 个冗余度非零的函数依赖。这使我们能够测试 SDP 在不同 k 值下的表现。我们运行 SDP，设置 $k \in \{1, 10, 50, 100, 500\}$，取值范围从冗余度最高的单个函数依赖到 500 个函数依赖。我们测量每个 k 的运行时间。图4显示，随着 k 变化，SDP 和 FDR 的表现有所不同。

k 对 FDR 不敏感。图 4 显示，FDR 的性能几乎保持平稳，对 k 不敏感。FDR 的运行时间维持在约 145–155 秒，内存使用量保持在 42.6 GB 至 44.9 GB 之间。这是因为 FDR 采用两阶段方法。它首先发现所有有效函数依赖，然后选择前 k 个结果。因此，FDR 的开销只取决于数据集，而不取决于 k。这些小幅波动来自系统随机性和超图采样，而不是 k。

为什么 SDP 对 k 敏感。相比之下，SDP 的性能会随着 k 的增加而提升。当 k 从 1 增加到 500 时，其运行时间从 0.43s 增长到 72.45s，生成节点数量从 442 增长到 24,000。这是因为 k 会影响 τ。阈值 τ 是迄今为止找到的第 k 个最佳函数依赖的冗余计数。当 k 较小时（例如 $k = 1$），τ 会迅速达到较高值，因此剪枝非常激进。SDP 几乎剪掉了所有候选项，仅验证了 53 个。然而，随着 k 增加到 500，τ 会降低。较低的 τ 意味着更多候选项通过上界检查，因此 SDP 必须探索更多候选项，从而产生更多节点并进行更多验证。

内存使用量。图 4(b) 显示，SDP 的内存消耗随着 k 增加而增加。内存使用量从 0.14 GB（$k = 1$）上升到 28.49 GB（$k = 500$）。这是因为当 k 较小时，τ 较高，因此 SDP 会在创建分区之前剪枝掉大多数候选项（只有 1% 通过）。当 k 较大时，τ 较低，因此更多候选项通过上界检查，SDP 必须实例化更多分区。然而，即使在 k =500 时，SDP（28.5 GB）的内存使用量仍远低于 FDR（44 GB），这表明即使 k 较大，SDP 的剪枝仍然有效。

表 II SDP 中优化组件的消融研究

数据集	变体 时间[秒]	内存[G]	#训练	#VAN	#函数依赖 g
atom	SDP 核心	15.836	3.1	2746	1408	1260
	SDP+O	4.991	3.1	534	247	192
	SDP+P	4.912	1.6	462	218	181
	SDP（完整）	0.978	0.5	379	90	50
census	SDP 核心	-	OM	-	-	-
	SDP+O	315.055	30.9	25089	17879	17655
	SDP+P	355.054	29.8	25447	17885	17655
	SDP（完整）	21.213	6.4	2548	1135	1066
ReactionNetwork	SDP 核心	81.792	32.4	27829	17736	15586
	SDP+O	8.239	4.6	3892	2600	2293
	SDP+P	8.514	4.5	3791	2558	2263
	SDP（完整）	2.849	1.3	1688	724	523

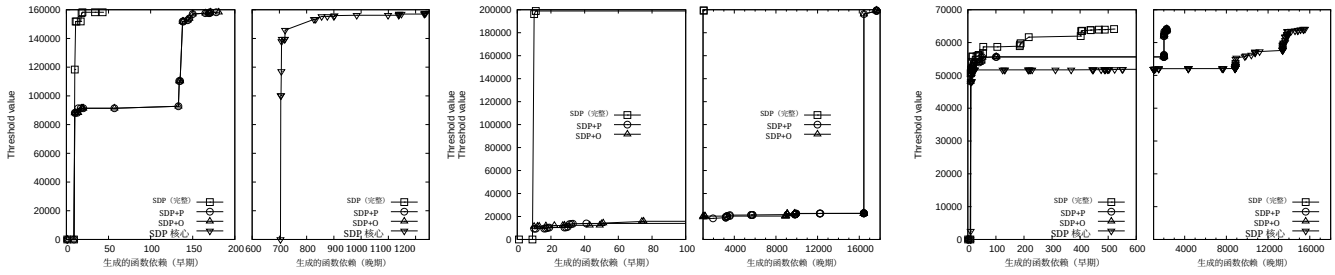
F. 优化策略的消融研究

我们进行了详细的消融研究，以评估三种优化策略各自的效果。为单独考察每个组件的影响，我们比较了算法的全部四种变体的性能。

- SDP 核心：SDP 的基础版本，包含统一剪枝规则，但按顺序处理右侧属性，且不包含任何高级优化。
- SDP+O：此版本在 SDP 核心的基础上加入了第一种优化，即启发式排序策略。
- SDP+P：此版本在 SDP+O 的基础上进一步加入了第二种优化，即利用 PCM 中的成对属性信息进行更精确的上界估计。
- SDP（完整）：SDP 的完整版本，用第三种优化——全局引导搜索策略——取代静态的右侧属性处理顺序。

我们在三个数据集（atom、census 和 ReactionNetwork）上进行了实验，$k = 10$。表 II 展示了实验结果。我们测量了运行时间（Time[s]）、峰值内存（Mem[G]）、生成的节点数（#TRN）、执行的验证次数（#VAN），以及找到前 k 个结果之前生成的函数依赖数量（#FDg）。

1) 启发式排序的影响：SDP 核心和 SDP+O 表明，属性排序至关重要。表 II 显示，SDP 核心在 census 上因内存溢出而失败，在 ReactionNetwork 上则耗时 81.792 秒。启发式排序（SDP+O）解决了内存问题，并将运行时间缩短至 8.239 秒（加速比为 10× ）。



(a) 原子数据集　　(b) 人口普查数据集　　（c）ReactionNetwork 数据集

图 5。不同优化策略的阈值 值示意图
。

这是因为启发式排序有助于更快地提高 τ。SDP 核心按随机顺序处理属性，因此可能先探索低冗余分支，导致 τ 增长缓慢。较低的 τ 无法有效剪枝。相比之下，SDP+O 会优先处理基数较低的属性。这有助于快速找到高冗余函数依赖，从而使 τ 增长更快。图 5(c) 证实了这一点。SDP+O 的 τ 值比 SDP 核心上升得早得多，因此 SDP+O 能更积极地剪枝，避免探索低冗余分支。

2）PCM 引导剪枝的影响：我们比较 SDP+O 和 SDP+P，以展示 PCM 的影响。在某些情况下，运行时间的提升幅度较小（例如 atom：从 4.991 秒缩短至 4.912 秒），但内存使用量显著下降。在 atom 上，峰值内存下降了 50%，从 3.1 GB 降至 1.6 GB。

这是因为 PCM 提供了更紧的上界。启发式排序会提高 τ，但算法仍需要紧的上界才能有效剪枝。SDP+O 使用宽松的单属性上界。SDP+P 使用 PCM，根据成对属性信息计算更紧的上界。因此，SDP+P 能更早（在更小的 $|X|$ 处）检测到无效分支，并在未生成其分区的情况下剪去候选项。尽管计算 PCM 会带来少量额外开销（这解释了 census 上运行时间的小幅增加），但节省的内存表明 PCM 是有效的。

3）全局搜索策略的影响：SDP（完整）加入了全局搜索，并实现了最大的加速比。在 census 上，运行时间从 355.054 秒（SDP+P）降至 21.213 秒，节点数从 25,000 减少到 2,548。在 atom 上，运行时间从 4.9 秒降至 0.978 秒。

这是因为全局搜索会在所有右侧属性之间共享 τ。在顺序方法中，各搜索树彼此隔离。在最后一棵树中发现的高 τ，无法帮助已经完成搜索的前面各树。SDP（完整）使用全局优先队列来调度所有 m 搜索树。任何一棵树找到高冗余函数依赖后，τ 就会立即更新，并剪去其他所有树。图 5(a) 和 (b) 显示，SDP（完整）的 τ 值几乎立即达到最大值，因此 SDP 能够及早剪去其他属性的搜索分支。这显著减少了节点生成。

G. 案例研究

我们选取了五个具有代表性的数据集进行定性分析（表 III）。这些数据集来自不同领域（例如，空间、医学、科学），且规模各异（从 10^3 到 10^6个元组）。冗余度有助于识别各个领域中最重要的函数依赖。

受篇幅限制，表 III 展示了每个数据集中 1-3 个有代表性的排名靠前的 k 函数依赖。这些示例涵盖了不同类型的依赖（例如，对称等价、层次结构、业务逻辑）。

1) 电影数据集。排名第一的函数依赖（发行日期 $\rightarrow$ 发行年份）体现了时间层次结构。该函数依赖的冗余计数为 972,242，LIFT $_k$ 725 为 750，表明在几乎整个数据集中，发行日期都能决定发行年份。这说明对模式进行规范化有助于节省存储空间。

2) tnweather 1.8m。相互依赖（降水$\leftrightarrow$雨）的冗余计数均为 1,888,120，几乎等于数据集大小（1,888,422）。这表明这些列代表相同的信息（属性同义词）。因此，可以合并或删除其中一列以减少冗余。

3) YouTube 推荐。观看时长、持续时间和百分比之间排名靠前的函数依赖（LIFT$_k$: 156,585）体现了日志记录系统中的数学关系。这些依赖可用于验证数据完整性。任何违反情况都表明计算逻辑或数据摄取存在错误。

4) 人力资源跨国公司数据。该数据集在我们的研究中 LIFT$_k$ (>9.2 百万) 最高。函数依赖（例如，职位名称 $\rightarrow$ 部门）体现了组织政策：职位名称决定部门。这些函数依赖可用于基于角色的访问控制和合规审计。

5) 性别差距。尽管该数据集较小（4,746 个元组），函数依赖揭示了调查指标如何映射到性别。类别指标（C API、C 手册）到性别的映射具有 4,739 的冗余度；适用于 99.8% 的元组。这表明人口统计类别定义明确。7 个违反该映射的元组（0.2%）有助于识别人口统计字段中的潜在错误。

VIII. 结论

本文探讨函数依赖（FD）发现中的信息过载问题。我们不再寻找所有函数依赖，而是提出 SDP（选择性发现与剪枝），以找出排名前-k 的函数依赖。我们使用冗余计数作为度量指标，对函数依赖进行排名。我们的关键贡献是提出一个紧致的单调上界，使精确发现成为可能，同时剪除大部分搜索空间。我们还提出了三种优化方法：启发式属性排序、PCM 引导的剪枝，以及协调多个搜索过程的全局调度

表 III 不同领域中具有代表性的前 k 个函数依赖及其语义意义

数据集	$\|r\|$	Rank	前 k 个函数依赖	#red	提升度$_k$	语义意义
电影数据集	999999	1	发行日期 →发行年份	972242	725750	自然时间层次
tnweather 1.8m	1888422	1 2	降水 →雨 雨 →降水	1888120 1888120	627303	属性同义等价
YouTube 推荐	1000000	1 2	{观看时长、观看百分比} →视频时长 {视频时长、观看时长} →观看百分比	509623 509623	156585	计算数学不变量
人力资源跨国公司数据	2000000	1 2	职位名称 →部门 入职日期 →状态	1999971 1994521	9209523	组织业务政策
性别差距	4746	1 2 3	{C API，C 手册} →性别 权重IJ →E NEds NIJ → E NEds	4739 4730 4730	168	调查元数据索引逻辑

。在 40 多个数据集上的实验表明，在高维和大规模数据上，SDP 可实现 10-1000× 的加速比，同时保持排名前-k 结果的精确性。案例研究表明，排名前-k 的函数依赖揭示了有助于数据管理的有意义模式。

致谢

本研究部分受到国家自然科学基金（项目编号：62402135、U21A20513）、山东省泰山学者计划（项目编号：tsqn202211091）以及山东省自然科学基金（项目编号：ZR2023QF059）的支持。

参考文献

[1] A. Silberschatz、H. F. Korth 和 S. Sudarshan，《数据库系统概念》第七版。McGraw-Hill Book Company，2020 年。[2] H. K¨ohler 和 S. Link，“SQL模式设计：基础、范式与规范化”，《信息系统》，第 76 卷，第 88–113 页，2018 年。[3] M. Masmoudi、S. B. A. B. Lamine、M. Karray、B. Archim`ede 和 H. B. Zghal，“语义数据集成与查询：综述与挑战”，《ACM计算评论》，第 56 卷，第 8 期，第 209:1–209:35 页，2024 年。[4] T. Boeckling 和 A. Bronselaer，“使用 swipe 清理数据”，《ACM数据与信息质量期刊》，第 17 卷，第 1 期，第 1–29 页，2025 年。[5] Z. Zhang 和 S. Link，“混合覆盖：使用最小键和函数依赖优化更新与查询”，《VLDB期刊》，第 34 卷，第 3 期，第 29 页，2025 年。[6] T. Bleifuß、T. Papenbrock、T. Bl¨asius、M. Schirneck 和 F. Naumann，“通过击中集枚举发现函数依赖”，《ACM管理数据会议录》，第 2 卷，第 1 期，第 43:1–43:24 页，2024 年。[7] T. Papenbrock 和 F. Naumann，“一种函数依赖发现的混合方法”，载于《2016年国际数据管理会议录》，SIGMOD 2016会议。ACM，2016 年，第 821–833 页。[8] T. Papenbrock、J. Ehrlich、J. Marten、T. Neubert、J. Rudolph、M. Sch¨onberg、J. Zwiener 和 F. Naumann，“函数依赖发现：七种算法的实验评估”，《VLDB会议录》，第 8 卷，第 10 期，第 1082–1093 页，2015 年。[9] X. Wan、X. Han、J. Wang 和 J. Li，“海量数据上的高效函数依赖发现”，《IEEE知识与数据工程汇刊》，第 36 卷，第 1 期，第 107–121 页，2024 年。[10] J. Liu、J. Li、C. Liu 和 Y. Chen，“从数据中发现依赖关系——综述”，《IEEE知识与数据工程汇刊》，第 24 卷，第 264 页，2012 年，第 2 期，第 251 页。[11] T. Bl¨asius、T. Friedrich 和 M. Schirneck，“关系数据库中依赖检测与发现的复杂性”，《理论计算机科学》，第 900 卷，第 79–96 页，2022 年。[12] M. W. Vincent，“关系数据库设计中第四范式的语义基础”，《Acta Informatica》，第 36 卷，第 3 期，第 173–213 页，1999 年。[13] Z. Wei 和 S. Link，“迈向有意义函数依赖的高效发现”，《信息系统》，第 116 卷，第 102224 页，2023 年。[14] M. Arenas 和 L. Libkin，“关系数据和XML数据范式的信息论方法”，《ACM期刊》，第 52 卷，第 283 页，2005 年，第 2 期，第 246 页。

[15] H. Koehler 和 S. Link，“以高精确率和高召回率挖掘有意义的键和外键”，《VLDB会议录》，第 18 卷，第 12 期，第 5363–5366 页，2025 年。[16] W. Fan、Z. Han、Y. Wang 和 M. Xie，“使用主观标准和客观标准发现前 k 条规则”，《ACM管理数据会议录》，第 1 卷，第 1 期，第 70:1–70:29 页，2023 年。[17] W. Fan、Z. Han、M. Xie 和 G. Zhang，“发现前 k 条相关且多样化的规则”，《ACM管理数据会议录》，第 2 卷，第 4 期，第 195:1–195:28 页，2024 年。[18] Y. Huhtala、J. K¨arkk¨ainen、P. Porkka 和 H. Toivonen，“TANE：一种发现函数依赖和近似依赖的高效算法”，《计算机杂志》，第 42 卷，第 2 期，第 100–111 页，1999 年。[19] N. Novelli 和 R. Cicchetti，“函数依赖和嵌入依赖推断：数据挖掘视角”，《信息系统》，第 26 卷，第 7 期，第 477–506 页，2001 年。[20] H. Yao 和 H. J. Hamilton，“从数据中挖掘函数依赖”，《数据挖掘与知识发现》，第 16 卷，第 2 期，第 197–219 页，2008 年。[21] Z. Abedjan、P. Schulze 和 F. Naumann，“DFD：高效的函数依赖发现 ”，载于 CIKM。ACM，2014 年，第 949–958 页。[22] J. Liu、F. Ye、J. Li 和 J. Wang，“关于从数据中发现函数依赖”，《数据与知识工程》，第 86 卷，第 146–159 页，2013 年。[23] P. A. Flach 和 I. Savnik，“数据库依赖发现：一种机器学习方法”，《AI通讯》，第 12 卷，第 3 期，第 139–160 页，1999 年。[24] S. Lopes、J. Petit 和 L. Lakhal，“函数依赖和Armstrong关系的高效发现”，载于 EDBT，第 1777 卷。Springer，2000 年，第 350–364 页。[25] C. M. Wyss、C. Giannella 和 E. L. Robertson，“Fastfds：一种用于从关系实例中挖掘函数依赖的启发式深度优先算法——扩展摘要”，载于 DaWaK，第 2114 卷。Springer，2001 年，第 101–110 页。[26] Z. Wei 和 S. Link，“函数依赖的发现与排名”，载于 ICDE。IEEE，2019 年，第 1526–1537 页。[27] J. Chomicki，“偏好查询的逻辑基础”，《IEEE数据工程公报》，第 34 卷，第 2 期，第 3–10 页，2011 年。[28] X. Han、J. Li 和 H. Gao，“海量数据上的高效前 k 项检索”，《IEEE知识与数据工程汇刊》，第 27 卷，第 10 期，第 2687–2699 页，2015 年。[29] ——，“海量数据上的高效前 k 项支配计算”，《IEEE知识与数据工程汇刊》，第 29 卷，第 6 期，第 1199–1211 页，2017 年。[30] X. Wan 和 X. Han，“海量数据上的高效前 k 项频繁项集挖掘”，《数据科学与工程》，第 9 卷，第 2 期，第 177–203 页，2024 年。[31] X. Liu、X. Niu 和 P. Fournier-Viger，“使用规则生成性质剪枝进行快速前 k 项关联规则挖掘”，《应用智能》，第 51 卷，第 4 期，第 2077–2093 页，2021 年。[32] F. Petitjean、T. Li、N. Tatti 和 G. I. Webb，“Skopus：基于杠杆率挖掘前 k 项序列模式”，《数据挖掘与知识发现》，第 30 卷，第 5 期，第 1086–1111 页，2016 年。[33] R. Kumar 和 K. Singh，“前 k 项高效用项集挖掘：当前状态与未来方向”，《知识工程评论》，第 39 卷，2024 年。[34] J. Birnick、T. Bl¨asius、T. Friedrich、F. Naumann、T. Papenbrock 和 M. Schirneck，“利用部分信息进行唯一列组合发现的击中集枚举”，《VLDB会议录》，第 13 卷，第 11 期，第 2270–2283 页，2020 年。[35] K. Murakami 和 T. Uno，“大规模超图对偶化的高效算法”，《离散应用数学》，第 170 卷，第 83–94 页，2014 年。[36] L. Berti-´Equille、H. Harmouch、F. Naumann、N. Novelli 和 S. Thiru-muruganathan，“从包含缺失值的关系数据中发现真实函数依赖”，《VLDB会议录》，第 11 卷，第 8 期，第 880–892 页，2018 年。