EAIFD：一种快速且可扩展的增量函数依赖发现算法

Yajuan Xu、Xixian Han、Xiaolong Wan

arXiv:2601.16025v1 [cs.DB] 22 Jan 2026

摘要——**函数依赖（FD）是关系数据库中的基本完整性约束，但在增量更新下发现函数依赖仍然面临挑战。静态算法需要完全重新执行，因此效率低下；而增量式算法则受到严重的性能和内存瓶颈制约。为应对这些挑战，本文提出了一种用于增量函数依赖发现的新算法 EAIFD。EAIFD 维护差异集的部分超图，并将增量函数依赖发现问题重新表述为超图上的最小击中集枚举，从而避免完全重新运行。EAIFD 引入了两项关键创新。首先，设计了一种多属性哈希表（MHT），用于存储有效函数依赖的高频键值映射，并证明其内存消耗与数据集大小无关。其次，开发了两步验证策略，以高效验证枚举出的候选项：该策略利用 MHT 有效缩小验证空间，随后有选择地加载数据块，对剩余候选项进行批量验证，从而有效避免重复的输入/输出操作。在真实世界数据集上的实验结果展现了 EAIFD 的显著优势。与现有算法相比，EAIFD 的运行时间最高可提升一个数量级，同时将内存使用量减少两个数量级以上，因此成为一种高效且可扩展的增量函数依赖发现方案。**

索引词—**数据库、算法、函数依赖、增量数据**

一、引言

FUNCTIONAL 数依赖（FD）是关系数据库中的基本完整性约束 [1], ，广泛应用于许多领域，例如模式设计与规范化 [2], [3], 、查询优化 [4],、数据清洗 [5]和数据集成 [6]。形式上，对于模式为R的关系实例r，当且仅当对于任意两个元组 r，若它们在 X 上的取值相同，则它们在 A上的取值也相同，函数依赖（FD） $X \to A$ 才是有效的；其中， $X \subseteq R$ 和 $A \in R$分别称为左部（LHS）和右部（RHS）。若 $A \notin X$，则函数依赖 $X \to A$ 是非平凡的；若不存在真子集 $X' \subset X$ 使得 $X' \to A$ 在 r中成立，则它是极小的。函数依赖发现旨在找出完整的F极小非平凡函数依赖集合。

示例 I.1。 表 *1* 展示了图 *1* 中的学生表。*SName* → *SAge* 是无效的，因为元组对（s_2和 s_6）的姓名相同，但年龄不同。{*SNO,SName*} → *SMajor* 是有效但非极小的函数依赖。而 $\mathcal{F}$ = {*SNO* → *SName*、*SNO* → *SAge*、*SNO* → *SGrade*、*SNO* →*SMajor*、*SNO* → *SDorm*、*SMajor* → *SDorm*}。

Yajuan Xu、Xixian Han 和 Xiaolong Wan 就职于中国哈尔滨工业大学计算机科学与技术学院。（电子邮件：xuyajuan@stu.hit.edu.cn、hanxx@hit.edu.cn、wxl@hit.edu.cn）稿件收到日期：XX XX, XXXX；修订日期：XX XX, XXXX。

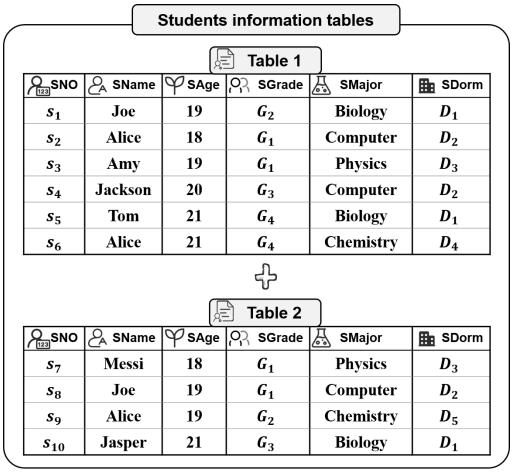

图 1：两个学生信息表的示意图。表 1 包含 6 个学生信息元组，表 2 包含 4 个学生信息元组，表 2 中的元组将以增量方式添加到表 1 中。

现实数据集通常会随着元组的添加而持续增长。例如，现代报纸数据库[1] 每年新增至少 100 个报纸名称。Ncvoter 数据集[2] 会随着北卡罗来纳州各县在每次选举后确认选民历史记录而进行增量式更新。数据库的增量式更新往往会导致 F 发生变化。如图 1 所示，当将表 2 中的新元组添加到表 1 时，先前有效的函数依赖 SMajor → SDorm 会因元组对（s_6， s_9）而变为无效。{SGrade, SMajor} → SDorm 成为新的最小有效函数依赖。

传统函数依赖发现算法 [7], [8], [9], [10], [11], [12], [13], [14], [15], [16] 是为静态数据集设计的，每当有数据插入时都必须重新运行，因此不适用于数据持续增长的场景。因此，研究人员开始探索增量数据库上的函数依赖发现。尽管增量算法 DynFD [17] 和 DHSFD [18], 的表现优于静态函数依赖发现算法，但仍面临以下挑战。具体而言，**(1) 预处理时间长**。 DynFD 和 DHSFD 都需要耗费大量成本进行预处理，其时间复杂度分别为 $O(m \cdot 2^m)$ 和 $O(n^2)$，这在现实场景中并不实用；**(2)宽模式上的可扩展性差**。随着 $|R|$ 增加，候选搜索空间呈指数级增长，导致验证和维护成本上升；**(3)内存消耗过高**。它们需要维护许多辅助数据结构，这些结构会随着数据量迅速增长，导致内存使用量居高不下。

[1]https://www.nlcpress.com/DigitalPublishingView.aspx?DPId=10.[2] https://www.ncsbe.gov/results-data/voter-history-data.

为应对这些挑战，本文提出了 EAIFD，这是一种高效的增量函数依赖发现算法，具有快速初始化、面向高维数据集的可扩展性以及低内存消耗等特点。EAIFD 通过三项关键贡献系统地应对上述挑战。**首先**，EAIFD 通过基于采样的初始化解决预处理时间长这一挑战。它仅在采样子集上计算差异集并构建部分超图，从而显著降低$O(n^2)$时间复杂度，并大幅缩短预处理时间。EAIFD 通过枚举部分超图上的击中集来生成候选函数依赖。**其次**，EAIFD 优化验证阶段，以降低候选项指数级增长所带来的高昂成本。具体而言，它采用迭代分组哈希验证（IGHV），并结合多属性哈希表（MHT），该哈希表存储左部值与右部值之间的映射。数据更新后，IGHV 将更新后的数据与预先构建的 MHT 进行比较，并将验证范围限制在相关数据块内，同时批量验证候选项。有效函数依赖会被加入 F，而无效函数依赖则用于更新超图，并迭代生成新的候选项。**第**三，为控制内存使用量，EAIFD 采用高频映射项保留策略，仅保留出现频率高于指定频率阈值的映射项，从而确保 MHT 保持轻量，并在内存消耗与性能效率之间取得平衡。我们在现实数据集上进行了全面实验。实验结果验证了 EAIFD 比增量函数依赖发现算法更高效。值得注意的是，与现有算法相比，EAIFD 的运行时间最高可提升一个数量级，同时将内存使用量降低两个数量级以上。

本文其余部分安排如下：第二节综述相关工作，第三节陈述问题并介绍预备知识，第四节介绍 EAIFD 算法，第五节给出全面的实验评估，第六节总结全文。

II. 相关工作

本节综述函数依赖发现方面的相关工作，并介绍与本文相关的两条研究方向：(i) 面向静态数据集的函数依赖发现算法；(ii) 支持增量数据集更新的函数依赖发现算法。

A. 静态数据的函数依赖发现

研究人员针对静态关系数据库提出了许多函数依赖发现算法 [7], [8], [19], [20], [12], [9], [10],[11], ，这些算法包含两个关键阶段：(i) 候选函数依赖生成和 (ii) 候选函数依赖验证 [13]。现有算法从不同方面提升函数依赖发现的效率，大致可分为以下三类。

基于属性的算法 [7], [21], [9], [12], [15], [8]在属性组合的幂集格中枚举候选函数依赖，并采用逐层或深度优先策略遍历格。它们通常使用位置列表索引（PLI）来验证候选函数依赖，并通过许多剪枝规则减少搜索空间。它们在元组数量较多的数据集上表现良好，但由于采用由候选项驱动的搜索策略，在属性数量较多的数据集上扩展性较差。

基于元组的算法 [11], [14], [19], [20] 基于对所有元组对进行值比较。这些算法通过构造属性的相同集（属性值相同）和差异集（属性值不同）来发现函数依赖。这些算法在属性数量增加时仍具有良好的扩展性。然而，对所有元组对进行比较往往会导致难以承受的 $O(n^2)$ 复杂度，从而严重降低其在长数据集上的性能。

混合算法 [10], [16]结合了基于属性的算法和基于元组的算法各自的优势。通过对抽样数据采用基于行的策略，并对完整数据集采用基于列的策略，它们有效避免了大量无效的候选函数依赖验证和元组比较。实验结果表明，这种混合设计的表现优于基于属性的算法和基于元组的算法。

B. 增量数据的函数依赖发现

传统的静态函数依赖发现算法必须在每次数据更新后重新执行整个发现过程。近年来，不同研究者开始致力于开发适用于增量数据场景的函数依赖发现算法。这些方法旨在数据更新后以较低的计算成本发现完整的极小非平凡函数依赖集，而无需重新执行整个算法。DynFD [17] 和 DHSFD [18] 被提出用于增量函数依赖发现，分别从属性和元组的角度进行增量函数依赖发现。

基于属性的算法： DynFD 是首个能够为增量更新的数据维护完整极小非平凡函数依赖集的算法。DynFD 不会频繁地重新计算所有函数依赖，而是检测数据中的变化，并基于先前的结果和一批更新操作，通过增量更新函数依赖集来推断这些变化对函数依赖的影响。它维护一个由极小函数依赖组成的正覆盖，以及一个由极大非函数依赖组成的负覆盖。增量数据更新后，DynFD 会验证最一般的函数依赖是否仍然成立。如果某个函数依赖无效，就会将其从正覆盖中移除，加入负覆盖，并通过向其左部添加新属性进行特化。随后，仅使用包含新插入元组的 PLI 簇验证特化后的候选项，从而提高效率。如果特化后的函数依赖成立，则将其加入正覆盖。当某轮验证中无效函数依赖的比例超过 10% 时，DynFD 会切换到违例驱动策略，将新元组与相关现有记录进行比较，以定位无效函数依赖。对于每个无效函数依赖，都会相应更新正覆盖和负覆盖，以保持一致性。

基于元组的算法： DHSFD 将增量函数依赖发现问题转化为超图上的动态击中集枚举问题，这些超图由整个数据集的差异集构建而成。DHSFD 维护包括 PLI 和带权重的差异集在内的辅助结构。增量数据更新后，利用 PLI 更新差异集及其权重。这些更新被视为超图中的边添加

，其中每条边代表一个差异集。随后，DHSFD 在更新后的超图上计算极小击中集。它利用关键边验证极小性，并采用一种称为 WalkDown 的深度优先策略扩展候选函数依赖。由此得到的极小击中集构成新发现的最小函数依赖的左部。通过仅比较包含增量元组的簇中的元组，并避免重建搜索树，DHSFD 比静态算法效率更高。

讨论。本文聚焦于增量数据的函数依赖发现问题，这类数据会通过添加元组持续更新。根据当前的数据场景，增量函数依赖发现面临以下挑战。

预处理时间过长。在预处理阶段，DynFD 从正覆盖推导负覆盖，而 DHSFD 则需要比较所有元组对，以计算数据集的差异集。对于包含 n 个元组的数据集，这一步的时间复杂度为 $O(n^2)$。当数据集或 |F| 较大时，这两种方法都会严重影响启动时间。

算法性能低下。由于搜索空间呈指数级增长，以及维护正覆盖和负覆盖所需的内存使用量增加，DynFD 在高维数据集上的效率较低。当失效率超过阈值时，DynFD 会切换到违例驱动验证，此时遍历包含新元组的 PLI簇的成本会变得极高。DHSFD 更新其差异集和超图的成本仍会显著增加。

辅助数据结构的内存空间消耗很高。这两种算法通常依赖消耗大量内存的辅助数据结构。DynFD维护正覆盖、负覆盖、位置列表索引（PLI）和字典编码。同样，DHSFD需要维护PLI、差异集和超图。随着新数据不断追加，这些结构的规模不断增大，从而导致内存使用量显著增加。

受这些局限性的推动，我们的目标是设计一种增量函数依赖发现算法，使其能够支持高维数据集、避免昂贵的预处理，并以极小的内存开销运行。

III. 问题定义

设r 为模式R上的关系实例。我们将增量数据记为 Δr。经过增量更新后，更新后的数据集记为 $r \cup \Delta r$。 $\forall t \in r$，我们分别用 $t[X]$ 或 $t[A]$ 表示 t 在属性集 $X \subseteq R$ 或单个属性 $A \in R$上的投影。

定义 III.1。（函数依赖）。函数依赖 $X \to A$，其中 $X \subseteq R$ 和 $A \in R$，对于 R的实例 r 是有效的，当且仅当 $\forall t_1$、$t_2 \in r : t_1[X] = t_2[X] \Rightarrow t_1[A] = t_2[A]$。如果不存在 $Y \subset X$、$Y \to A$ 和$A \notin X$，则 $X \to A$ 是极小且非平凡的。

给定函数依赖 $X \to A$ 和 $Y \to A$，其中 $X \subset Y$， $X \to A$ 是 $Y \to A$ 的**泛化**，而 $Y \to A$ 是 $X \to A$ 的**特化**。函数依赖发现关注极小且非平凡函数依赖的完整集合 F，因为所有其他有效函数依赖都可以使用 Armstrong 公理 [22]从它们推导出来。

定义 III.2. (违反依赖的元组对**)**。如果候选函数依赖 $X \to A$ 无效，那么必定存在一对元组 t_1、$t_2 \in r$，其中 $t_1[X] = t_2[X]$ 但 $t_1[A] \neq t_2[A]$。$(t_1$、$t_2)$ 称为候选函数依赖 $X \to A$的违反依赖的元组对。

本文遵循基于元组的算法 [14]、[23], [18]，将函数依赖发现转换为超图中的击中集枚举。下面介绍必要的定义。

定义 III.3。（差异集）。对于 t_1、$t_2 \in r$，它们的差异集为 $D(t_1$、$t_2) = \{A \in R \mid t_1[A] \neq t_2[A]\}$，表示两个元组具有不同值的属性集合。

r 的差异集记为 $D_r = \{D(t_1$、$t_2) \mid t_1$、$t_2 \in r$、$D(t_1$、$t_2) \neq \varnothing\}$。对于属性 $A \in R$， r 模 A 的差异集记为 $D_r^A = \{D \setminus \{A\} \mid D \in D_r \wedge A \in D\}$。如果不存在 $D' \in D_r$ 和 $D' \subset D$，则差异集 $D \in D_r$ 是极小的。$\overline{D_r} = \{D \in D_r \mid D' \in D_r \wedge D' \subseteq D \Rightarrow D' = D\}$，其中$\overline{D_r}$由 D_r的所有极小差异集组成。 r的超图记为 $\mathrm{H}_r = (R$、$D_r)$，其中 R 是顶点集，D_r 是超边集。属性A 的子超图记为 $\mathrm{H}_A = (R$、$D_r^A)$。最小超图 $\mathrm{H}_r = (R$、$D_r)$仅包含H_r的最小超边。

定义 III.4.（击中集）。给定一个子超图 $\mathcal{H}_A = (R, D_r^A)$，$X$ 是 $\mathcal{H}_A$ 的击中集，当且仅当它与 D_r^A 中的每条超边都相交。如果 X 的任何真子集都不是击中集，则 X 是极小击中集。 $HS(\cdot)$ 表示一个超图的极小击中集集合。

性质 III.1. 如果函数依赖 $X \to A\,(A \notin X)$ 在 r 上成立，则 X 必须是 $\mathcal{H}_A$ 的击中集。此外， $X \to A$ 是极小且非平凡的函数依赖，当且仅当 X 是 $\mathcal{H}_A$ 的极小击中集。

性质 III.2. $\mathcal{H}_A$ 的任意击中集也是 $\mathcal{H}_A$ 的击中集。$\mathcal{H}_A$ 的完整极小击中集集合等于 $\mathcal{H}_A$ 的完整极小击中集集合。

枚举击中集时，只需考虑 H_r 。事实上，H_r 通常远小于 H_r。在以下各节中，超图和子超图均指其极小形式。

问题陈述。给定一个模式为R的关系实例 r ，增量式函数依赖发现旨在每次数据更新后，无需重新执行算法，即可找出在数据集上成立的极小且非平凡函数依赖的完整集合$\mathcal{F}$ 。

IV. EAIFD 算法

本节介绍 EAIFD，这是一种面向增量数据库的新型函数依赖发现算法。EAIFD 能够高效地发现数据集上极小且非平凡函数依赖的完整集合 F，并在每次数据增量后快速更新 F。与现有增量方法相比，EAIFD 可快速初始化，能够高效扩展至高维数据集，并降低内存消耗。EAIFD 的整体框架如图 2 所示。

EAIFD 的核心思想是：通过对根据差异集构建的超图进行击中集枚举来生成候选函数依赖，并对候选函数依赖进行验证。

EAIFD

由两个核心组件组成：针对初始数据集的一次性发现过程，以及针对所有后续增量数据的高效增量更新机制。一次性发现过程利用采样实现快速初始化，并构建关键的多属性哈希表（MHT），以加速后续更新。增量更新机制智能地利用 MHT 进行快速验证剪枝，并在数据块上采用 IGHV，以高效验证剩余候选项。两个组件的详细信息分别在第 IV-A 节和第 IV-B 节中介绍。

A. 初始数据集上的一次性发现过程 r

预处理。输入数据集 r 首先按每个属性升序排序并存储在磁盘上，同时记录每个不同值的索引范围，以便在后续步骤中快速加载数据块。然后，EAIFD 以抽样率 $\varepsilon = 0.3$对r 进行均匀无放回随机抽样，从包含 n条记录的数据集中所有记录对中选取 $((\binom{n}{2}))^{\varepsilon}$ 个记录对。这种方法兼顾了数据覆盖率与效率，并已证明在函数依赖发现中具有很高的有效性 [11], [24]。我们将采样数据记为 s。

一次性发现过程（如算法1所示）生成在r 上成立的全部极小非平凡函数依赖集合 F，并通过两个核心步骤构建 MHT：候选函数依赖生成（第 IV-A1 节）和验证（第 IV-A2 节）。

1）使用采样数据生成候选函数依赖 s：EAIFD 将每个属性 $A \in R$ 依次作为右部来生成候选函数依赖（第 2–28 行）。对于每个 A，它根据采样数据 s 构建部分子超图 $\mathrm{H}_A = (R, D_s^A)$（第 2 行），其中 $D_s^A = D(t_i, t_j) \setminus A \mid t_i, t_j \in s \wedge A \in D(t_i, t_j)$中的超边表示在 A上取值不同的元组对所对应的差异集，并移除 A 本身。D_s^A捕获涉及 A 的差异，并定义以 A 为右部的候选左部搜索空间。随后，EAIFD 对每个 H_A 应用 MMCS 算法，枚举以 A 为右部的极小左部候选项（第 6 行），从而形成候选函数依赖集合 C_{FDs}。这里，MMCS [25] 是一种基于超边、用于从超图中枚举极小击中集的分支算法。它通过构建搜索树并结合极小性检查来高效探索搜索空间，从而避免不必要的分支。由于这些候选函数依赖是根据采样数据集 s上的部分子超图推导而来，因此需要进行验证，以确保它们在 r上有效。

2）使用 *IGHV* 验证候选函数依赖：我们提出一种内存使用量较低的高效验证方法，称为迭代分组哈希验证（IGHV），用于检查基于r 上的部分子超图推导出的候选函数依赖是否有效。图 3 展示了其主要流程。IGHV 既适用于静态数据，也适用于增量更新，两者仅有少许差异。

为了理解 IGHV 的新颖之处和优势，首先回顾标准的基于哈希的验证方法的原理和局限性会有所帮助。令 B 表示完整的哈希桶集合，B_X 表示候选函数依赖 $X \to A$ 对应的哈希桶集合。 $\forall t \in r$，该方法将哈希函数 f 应用于 $t[X]$ 以计算哈希值 h，并将 t 分配到桶 B_X^h中。同一桶 B_X^h 中的任意两个元组 t_i，t_j都满足 $t_i[X] = t_j[X]$。如果 $t_i[A] \neq t_j[A]$，则会出现违反依赖的

算法 1： EAIFD 初始函数依赖发现

输入： 样本数据 s 和频率阈值 θ
输出： 完整的有效函数依赖集合 F、多属性哈希表 MHT 和子超图 H

1 将 F、 Δ_{diff}、B 初始化为空集
2 为每个属性构建包含子超图的 H
3 **对** $A \in R$ **执行**
4 **重复**
5 使用 Δ_{diff} 更新 H，并清除 Δ_{diff}
6 $\mathrm{C}_{FDs} \leftarrow$MMCS($\mathrm{H}_A$)
7 G $\leftarrow$组(C_{FDs})
8 **对** $g \in$G **中的每个元素执行**
9 选择 g的所有左部中的一个公共属性 B
10 **对** 按 B **排序的** r_i 块 r 中的每个元素执行
11 **对每个**元组 $t \in r_i$ **执行**
12 **对** $X_i \to A \in g$ **中的每个元素执行**
13 $h \leftarrow f(\{t[X_i]\})$
14 $\mathrm{B}_{X_i}^h \leftarrow \mathrm{B}_{X_i}^h \cup \{t\}$
15 **对每个** $\mathcal{B}_{X_i} \in$B **执行**
16 **对每个** $\mathcal{B}_{X_i}^h \in \mathcal{B}_{X_i}$ **执行 如果** $\exists t_1, t_2 \in$
17 $\mathcal{B}_{X_i}^h$ 与 $t_1[A] \neq t_2[A]$ **then**
18 计算.....的差异集 (t_1, t_2) 并添加到 Δ_{diff}
19 $g \leftarrow g \setminus \{X_i \to A\}$
20 **中断**
21 **如果** $X_i \to A$ 有效 **则**
22 **对于每个** $\mathcal{B}_{X_i}^h \in \mathcal{B}_{X_i}$ **do**
23 **如果** 中的元组数量 $\mathcal{B}_{X_i}^h \geq \theta \times |r|$ **则**
24 将 $(h, t[A])$ 添加到 $MHT(X \to A)$
25 清空 B_{X_i}
26 F $\leftarrow$ F $\cup g$
27 **直到** Δ_{diff} 为空
28 **返回**$\mathcal{F}$ MHT, $\mathcal{H}$

元组对（t_i，t_j）会被检测到，并且 $X \to A$ 会被判定为无效。为了验证 $X \to A$，该方法会遍历整个数据集以构建 B_X。这种方法在小型数据集上效果良好。然而，随着候选函数依赖数量增加或数据集规模扩大，哈希桶的内存消耗会变得过高，并可能导致内存溢出。

IGHV 显著提高了验证效率，同时有效控制内存使用量。算法1描述了 IGHV 验证通过 MMCS 树搜索（第 7–26 行）获得的候选函数依赖 C_{FDs} 的过程。IGHV 首先对 C_{FDs}进行分组，使每组中的函数依赖至少共享一个公共左部属性（第 7 行）。如果两个候选函数依赖 $X' \to A$ 和$X'' \to A$ 满足 $X' \cap X'' \neq \varnothing$，则可以将它们放在同一组 g中。对于每个组 $g = \{X_1 \to A, X_2 \to A, \ldots\ldots, X_k \to A\}$，其中 $X_1 \cap X_2 \cap \cdots \cap X_k \neq \varnothing$，选择一个值分布均匀的公共属性 $B \in X_1 \cap X_2 \cap \cdots \cap X_k$作为公共排序属性（第 9 行）。这可以确保数据块大小均衡，从而避免因数据块过大而导致内存消耗过高，以及因小数据块过多而造成频繁的加载开销。在预处理期间，数据按 B排序，并可划分为数据块 $r = r_1 \cup r_2 \cup \cdots \cup r_m$，

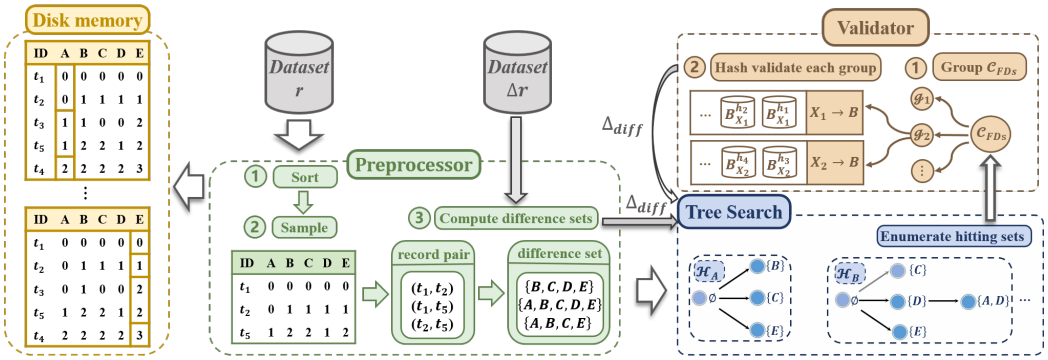

图 2：EAIFD 及其组件概览。

其中，数据块 r_i 中的所有元组在 B上具有相同的值。验证期间，数据集按 B的排序顺序加载。每个数据块 r_i 按以下方式处理（第 10–25 行）：(i) 对于每个元组 $t \in r_i$，IGHV 计算当前组 g中所有候选函数依赖的左部的哈希值 $f(t[X_1])$ $f(t[X_2])$、……、 $f(t[X_k])$，并将它们插入相应的哈希桶集合 B_{X_1}、B_{X_2}、……、B_{X_k}中（第 11–14 行）。(ii) 对于每个候选函数依赖 $X \to A \in g$，检查哈希桶集合 B_X 中的每个桶，以验证桶内所有元组在属性 A 上的值是否相同（第 15–20 行）。如果一个哈希桶中包含在 A上取值不同的元组，则识别出一个违反依赖的元组对（t_p，t_q），其中 $t_p[X] = t_q[X]$但 $t_p[A] \neq t_q[A]$，这表明 $X \to A$ 不成立。这些违反依赖的元组对的差异集会被添加到新的超边集 Δ_{diff} 中，以便后续更新超图。为避免冗余验证，随后移除无效的候选函数依赖（第 17–19 行）。

验证完数据块 r_i 后，IGHV 会清除相应的哈希桶（第 25 行），然后继续验证下一个块。处理完所有数据块后，该组的验证结束，并将该组中的有效函数依赖添加到 F 中（第 26 行）。在验证下一组之前，使用新收集的差异集 Δ_{diff} 更新 H 中与这些差异集相对应的部分子超图（第 5 行）。随后，MMCS 继续生成新的候选函数依赖，并对其进行相同的验证，直至不再产生新的差异集。最后，对其余属性子超图应用相同的过程。

3）构建多属性哈希表：在一次性发现过程中，我们引入多属性哈希表（MHT），用于保存有效函数依赖的左部到右部的值映射，并与 IGHV 集成，以便在增量场景中剪枝搜索空间并加速候选函数依赖验证。

MHT 旨在进行函数依赖验证和冲突检测，其依据的原则是：对于有效函数依赖 $X \to A$，所有在 X 上相同的元组，在 A上也必须相同。对于 r中的每个元组 t，通过对左部投影 $t[X]$应用哈希函数 f，生成哈希键，再将其与右部值 $t[A]$配对，从而形成映射项 $(f(t[X])\ t[A])$。这种方法可以减少内存使用量并加快查找速度。为确保正确性，哈希函数 f 可确保哈希键针对 X唯一标识一个元组等价类；对于任意元组 t_i，$t_j \in r$ 如果 $t_i[X] = t_j[X]$，那么 $f(t_i[X]) = f(t_j[X])$；并且如果 $t_i[X] \neq t_j[X]$，那么$f(t_i[X]) \neq f(t_j[X])$。通过为所有元组计算映射项，即可构建完整的 MHT，清晰地体现有效函数依赖中从左部到右部的值映射。如果 $X \to A$成立，每个哈希键 $f(t_i[X])$都必须对应一个唯一值 $t[A]$。 MHT 支持快速冲突检测：如果一个哈希键映射到不同的值，就表明存在违反依赖的元组对t_i，t_j 满足 $t_i[X] = t_j[X]$ 但 $t_i[A] \neq t_j[A]$。在验证阶段， MHT 支持 $O(k)$时间复杂度，使其在函数依赖验证中尤其高效。

对于包含大量候选函数依赖的大型数据集， MHT 中的映射项数量可能会迅速增长，导致内存使用量过高。因此，EAIFD采用高频映射项保留策略，仅缓存频率超过频率阈值 θ 的映射项。对于任意有效函数依赖 $X \to A$，IGHV 会统计 X 与 A 之间的映射频率（第 21-24 行）。如果哈希映射项（h，a）满足频率条件：

$$\frac{|t \in r_j | r_j \in r \ f\ t[X] = h\ t[A] = a|}{|r|} \geq \theta,$$

则会将其缓存到 $MHT(X \to A)$ 中。频繁映射表示左部与右部之间经常观察到的值关联。

阈值 θ 的选择必须在较高阈值与较低阈值之间取得平衡：较高阈值会生成稀疏的 MHT，剪枝效果较弱；较低阈值则会导致 MHT 过大，并造成内存使用量过高。本文根据以下三点考虑，将 $\theta = 80$ 设为 %。首先，采用高频阈值仅保留最重要元素是业界公认的做法 [26]。其次，出现频率超过 80% 的值映射能够有效捕捉现实数据集中最具代表性的属性值组合。第三，后文给出的实验结果进一步证实了这一选择的有效性。该策略利用高频映射项高效剪枝验证空间，同时避免 MHT变得过大，从而在性能效率与内存使用量之间实现平衡。

处理完所有子超图后，一次性发现过程结束。该过程为 r 中的每个有效函数依赖 $X \to A$ 推导出 F 和多属性哈希表 MHT，其中仅包含高频值映射项，以及包含每个属性对应子超图的超图集合 H。

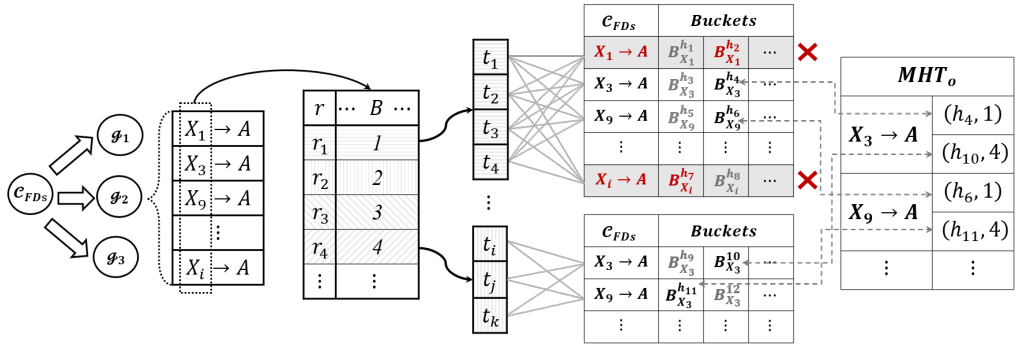

图 3：IGHV 的流程。分组后，选择一些 C_{FDs} in g_2，并展示其在数据块 r_1 和 r_4上的核心验证步骤

B. 关于 $r \cup \Delta r$ 的增量更新机制

当初始数据集 r 使用增量数据 Δr 更新时，会触发增量更新机制，以高效地更新数据集 $r \cup \Delta r$ 上的 F 和 MHT。与一次性发现过程类似，该机制同样采用迭代过程：由 MMCS 生成候选函数依赖，并通过两步验证策略进行验证。该机制会迭代更新超图，直至不再产生新的候选项，从而完成 F 的更新。增量更新机制的完整流程如算法2所示。

1) 使用增量数据生成候选函数依赖 Δr：增量更新后，将增量数据 Δr中的元组两两比较，以生成新的差异集 Δ_{diff}（第 1 行）。由于 $|\Delta r|$ 的规模通常较小，这一操作的时间开销仍可接受。基于 Δ_{diff}，更新相应的子超图（第 4 行），并在更新后的超图上执行 MMCS 树搜索，以生成候选函数依赖，也记作 C_{FDs}（第 5 行）。MMCS 从在一次性发现过程中获得的初始数据集 r 所构建超图的极小击中集对应的树节点开始搜索，从而避免完全重新开始。MMCS 首先将Δ_{diff} 作为新的超边来更新超图，并识别出无法覆盖这些新超边的受影响初始极小击中集。随后，对于每个受影响的初始极小击中集，MMCS 从未覆盖的属性中选取候选属性，并对其进行扩展以生成新的候选函数依赖。

然而，通过 MMCS 得到的候选函数依赖 C_{FDs} 只能保证在 r 和 Δr 上局部成立，在 $r \cup \Delta r$ 上则不一定成立。这是因为，超图仅根据一次性发现过程中由 r 生成的差异集，以及增量数据 Δr 内部成对比较生成的差异集构建，并未纳入 r 与 Δr 之间元组对所产生的差异集。MMCS 得到的候选函数依赖 $X \to A$ 分别在 r 和 Δr 上有效，但如果元组 $t_i \in r$ 和 $t_j \in \Delta r$ 满足 $t_i[X] = t_j[X]$ 而不满足 $t_i[A] \neq t_j[A]$，则该依赖在 $r \cup \Delta r$ 上可能无效。因此，必须进一步验证 C_{FDs} 中的每个候选函数依赖，以确认其在 $r \cup \Delta r$ 上是否有效。

候选函数依赖 C_{FDs} 必须在 r 上成立，原因有二。首先，一次性发现过程 会迭代扩展部分超图，直到其极小击中集与根据 r的所有极小差异集构建的全局超图的极小击中集一致，从而确保 F 对所有

算法2： EAIFD 增量函数依赖发现

输入： Δr, $\mathcal{F}$, MHT, $\mathcal{H}$
输出： $\mathcal{F}$, MHT
1: 计算 Δr 的差异集并将其添加到 Δ_{diff}
2: **对于** $A \in R$ **执行**
3: **当** $\Delta_{diff} \neq \varnothing$ **时执行**
4: 通过 Δ_{diff} 更新 H，然后清除 Δ_{diff}
5: $C_{FDs} \leftarrow$MMCS(H_A)
6: $C_1 \leftarrow C_{FDs} \cap$F
7: $C_2 \leftarrow C_{FDs} \setminus C_1$
// 验证 C 中的候选函数依赖$_1$
8: res1 ← VCF(C_1,"C_1_MODE")
// 验证 C 中的候选函数依赖$_2$
9: res2 ← VCF(C_2,"C_2_MODE")
10: 根据 res1 和 res2 更新 F，Δ_{diff}, MHT
11: **返回**$\mathcal{F}$, MHT

在 r上的最小非平凡函数依赖都是完备的。研究表明，在这一迭代扩展过程中，部分超图的每个极小击中集要么是全局超图的极小击中集，要么可以扩展为其中一个 [17]。当扩展不再产生新的超边时，部分超图与全局超图中的极小击中集集合相同，这表明 r 上的所有最小非平凡函数依赖都已在 F 中捕获。其次，在增量更新期间，超图会通过来自 Δr的差异集 Δ_{diff} 进行动态更新。除非 Δ_{diff} 引入新的极小超边，否则原有的极小击中集保持不变。已证明 [18] 更新后超图中的每个新极小击中集都必须包含一个先前的极小击中集，这意味着它是先前函数依赖的特化。这确保了从更新后的超图生成的候选函数依赖仅排除 F 中因Δr而无效的候选项，而所有其余候选项和新增候选项在 r上仍然成立。

为提高验证效率，增量更新机 制将 C_{FDs} 分为两类：(i) 在 r 上成立的最小函数依赖，记为 $C_1 = C_{FDs} \cap$F；(ii) 在 r 上也成立的非最小函数依赖，记为 $C_2 = C_{FDs} \setminus C_1$，，可将其视为 C_1 的特化项。根据 MHT 是否可用，分别验证这两类函数依赖（第 8–9 行）。C_1 和 C_2 的不同验证机制详见 IV-B2 节和 IV-B3 节，并在算法 3 中概述。

2) 在 C_1 中使用 *IGHV* 和 MHT验证候选函数依赖：由于 Δr 通常远小于 r，大多数更新只会影响少量函数依赖的有效性。大多数元组仍然

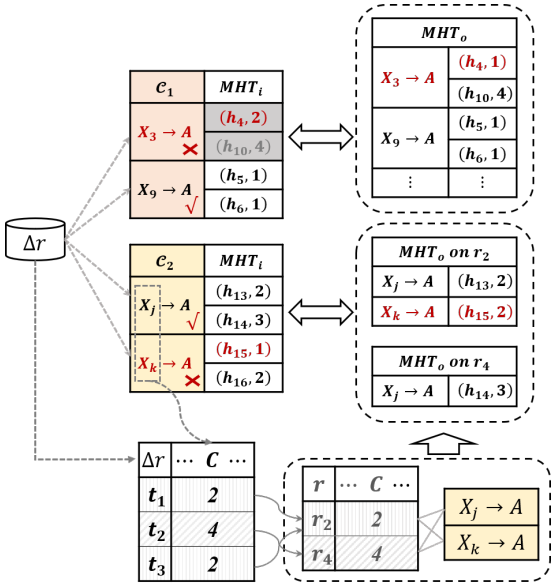

图 4：验证两种类型 C_{FDs} 的过程：C_1 和 C_2。为便于说明，Δr 仅包含 3 个元组，并且每种类型各验证两个候选函数依赖。

支持原有函数依赖，并且大多数函数依赖在更新后仍然有效。因此，C_1 占候选函数依赖总数的很大比例。因此，提高 C_1 的验证效率已成为优化整体效率的关键。

为了高效验证 $r \cup \Delta r$ 上 C_1 中的候选项，EAIFD 采用两步验证策略。第一步，MHT 基于验证，将基于 Δr 构建的局部哈希表（记为 MHT_Δ）与在 r 上预先计算的 MHT 进行比较，以快速识别冲突并高效缩减验证空间。任何无法在基于 *MHT* 的验证 步骤中判定的候选项，都会进入第二步，即表扫描验证，该步骤从r 加载相关数据块，并使用 IGHV 进行验证。图 4 展示了 EAIFD 如何处理 C_1 中的候选函数依赖。

步骤 1：MHT 基于验证。对于候选函数依赖 $X \to A$，EAIFD 首先在 Δr 上构建一个局部多属性映射哈希表，记为 $MHT_\Delta(X \to A)$（第 3 行）。由于$|\Delta r|$ 较小，$MHT_\Delta(X \to A)$ 的内存使用量是可控的。构建完 MHT_Δ 后，MHT 基于验证 会将 $MHT_\Delta(X \to A)$ 中的映射项与在 r 上构建的 $MHT(X \to A)$ 进行比较（第 5 行）。在比较过程中，候选函数依赖 $X \to A$ 的验证结果必定属于以下三种情况之一。

- 有效。如果 $MHT_\Delta(X \to A) \subseteq MHT(X \to A)$，则 MHT_Δ 中的所有映射都已存在于 MHT 中，这意味着 Δr 不会引入新的左部到右部的值映射。$X \to A$ 在 $r \cup \Delta r$ 上自然成立，并被更新为 F，同时从 C_1 中移除，无需进一步验证。
- 无效。如果 $MHT_\Delta(X \to A)$ 中的项 $(f(t_j[X])\ t_j[A])$ 与 $MHT(X \to A)$ 中的项 $(f(t_i[X])\ t_i[A])$ 满足 $f(t_i[X]) = f(t_j[X])$，但不满足 $t_i[A] \neq t_j[A]$，那么元组 $t_j \in \Delta r$ 和 $t_i \in r$ 构成违反依赖的元组对。该元组对共享相同的哈希键，但右侧值不同值，因此 $X \to A$ 在 $r \cup \Delta r$ 上无效，并从 C_1中移除。对 (t_i, t_j) 的比较会产生一个新的差异集，并将其添加到 Δ_{diff}，然后用于在下一次迭代之前更新 H 中相关的子超图。
- 不确定。在比较 $MHT_\Delta(X \to A)$ 和 $MHT(X \to A)$ 时，$MHT_\Delta(X \to A)$ 中同时存在于 MHT 的任何映射项 (h, a) 都会被移除，以减少冗余。如果比较后 $MHT_\Delta(X \to A) \neq \varnothing$，则表明 $MHT_\Delta(X \to A)$ 包含未出现在 $MHT(X \to A)$ 中的映射项。这些项出现的原因有两个：(i) 它们存在于初始数据集 r 中，但其出现频率未达到频率阈值 θ，因此未被记录在 MHT 中；或 (ii) 它们表示由 Δr 引入的新左部到右部的值映射。在这种情况下，$X \to A$ 的有效性仍不确定，并会触发表扫描验证。

步骤 2：表扫描验证。在基于 *MHT* 的验证之后，只有被归类为不确定 的候选函数依赖才需要进一步进行表扫描验证，该验证会对初始数据集中选择性加载的数据块使用 IGHV。

具体而言，表扫描验证会根据 Δr 中的左部属性值，从 r 中选择性加载相应的数据块。我们只需检测来自 Δr 的一个元组与来自 r 的一个元组之间是否存在潜在冲突，以检查候选函数依赖。当 r 中的元组包含与 Δr 中元组相同的左部属性值、从而生成相同的哈希键但具有不同的右侧值时，可能会发生冲突。由于r 的数据块中的元组若不与 Δr 中的任何元组共享排序属性值，就无法与 Δr 中的元组生成相同的哈希键，因此无法从这些元组中检测到冲突。扫描此类数据块只会产生冗余的输入/输出和计算开销。由于$|\Delta r|$ 通常较小，需要加载的数据块数量也受到限制，因此该策略在确保正确性的同时，能够显著提高验证效率。

表扫描验证 首先为每个候选函数依赖在 Δr 上构建局部 MHT_Δ（第 3 行）。随后，使用与一次性发现过程 中 IGHV 相同的分组策略对候选函数依赖进行分组（第 7 行）。分组后，为每个组选择一个排序属性 B（第 9 行）。验证该组中的候选函数依赖时，EAIFD 仅加载来自 r、按 B 排序且其 B 值也存在于 Δr中的数据块。形式上，要加载的数据块集合定义为：$r' = \{r_i \mid t_i \in r_i \wedge t_i[B] \in \{t_j[B] \mid t_j \in \Delta r, f(t_j[X]) \in K_\Delta\}\}$，其中$K_\Delta$ = KeySet($MHT_\Delta(X \to A)$)（第 10 行）。随后，该方法在 r' 中的数据块上验证这一组候选项，而不是扫描整个 r，并依次对每个数据块执行批量验证。

获得相关数据块集合 r' 后，EAIFD 依次加载每个数据块，并对当前块进行批量验证。对于组中的每个候选函数依赖 $X \to A$，它首先根据当前数据块中的元组构建一个本地临时 $MHT(X \to A)$，用于记录左部哈希键值到右部值的映射；然后比较本地临时 $MHT(X \to A)$ 和 $MHT_\Delta(X \to A)$（第 13 行）。只有当 r'中所有块之间都未发现冲突时，$X \to A$ 才有效。反之，如果存在满足 $f(t_i[X]) = f(t_j[X])$ 但 $t_i[A] \neq t_j[A]$ 的 $t_j \in \Delta r$ 和 $t_i \in r$，则该 $X \to A$

算法 3： 验证候选函数依赖（VCF）

输入： 候选集 C 和验证模式 $mode$
输出： $Valid_{FDs}$，Δ_{diff}
// 当 $mode$ 等于"C_1_MODE"时：C =C_1
// 当 $mode$ 等于"C_2_MODE"时：C =C_2
1 $Valid_{FDs}$、Δ_{diff} 初始化为空集
2 **对于**每个 $X \to A \in C$ **执行**
3 在 Δr 上构建 $MHT_{\Delta}(X \to A)$
4 **如果** $mode$ 等于"C_1_MODE"**则**
5 检查 $MHT_{\Delta}(X \to A)$ 和 $MHT(X \to A)$，并收集结果
6 根据结果更新 $Valid_{FDs}$、C、Δ_{diff}
7 G ←组(C)
8 **对**每个 $g \in \mathcal{G}$ **执行**
9 选择 g的所有左部共有的属性 B
10 **对**每个块 $r_i \in r'$ **执行**
11 **对**每个 $X \to A \in g$ **执行**
12 在 r_i上构建 $MHT(X \to A)$
13 检查 $MHT_{\Delta}(X \to A)$ 和 $MHT(X \to A)$ 并收集结果
14 通过比较结果更新 $Valid_{FDs}$, Δ_{diff}
15 **返回** $Valid_{FDs}$，Δ_{diff}

会被判定为无效，并将 (t_i, t_j) 的差异集添加到 Δ_{diff}，以便在下一次迭代中更新超图。每次块验证后，频率超过 θ 的本地临时 MHT 中的映射项会被记录并更新到全局完整 MHT 中，然后清空本地临时 MHT，以减少内存使用量。处理完 r' 中的所有块后，将记录有效函数依赖和新的差异集（第 14 行）。该过程会对所有组重复执行（第 8–14 行）。处理完所有组后，验证过程通过返回最终的 $Valid_{FDs}$ 和 Δ_{diff} 结束（第 15 行）。

因此，通过对 MHT_{Δ} 和 MHT 进行基于 MHT的比较，然后使用 IGHV 对已加载的数据块进行选择性表扫描验证，EAIFD 能够高效验证 C_1，并识别仍然有效的最小函数依赖。

3) 使用 *IGHV* 验证C_2 中的候选函数依赖：对于 C_2，中在 r 上成立但并非极小的候选函数依赖，MHT 中不存在对应的映射项，因为它仅存储极小且非平凡函数依赖的值映射。EAIFD 使用与 IV-B2节中针对 C_1，中不确定候选函数依赖所述的表扫描验证相同的流程来验证这些依赖，并从 r 加载包含 Δr 中属性值的数据块。该方法既确保正确性，又避免了冗余计算。此过程如图 4 所示。

验证 C_1 和 C_2, 后，EAIFD 会将 $r \cup \Delta r$ 上成立的函数依赖更新到 F 中，并从违反依赖的元组对中返回一个差异集 Δ_{diff}。这会更新 H 中相关的超图，以供下一次迭代使用。

C. 算法分析

为确立 EAIFD 算法在理论上的可靠性和效率，本部分首先给出正确性证明，随后进行详细的复杂度分析。前者保证算法的准确性，后者则量化其性能优势。

1）正确性证明：EAIFD 在一次性发现过程和增量更新机制中都涉及候选生成和验证，其整体正确性取决于两个步骤。对于候选生成，其正确性由以下三个方面保证：(i) 完备性、(ii) 极小性和 (iii) 非平凡性。

完备性。对于每个右部属性 A，一次性发现过程 都会构建一个子超图 H_A，并通过从违反依赖的元组对中推导出的新超边对其进行迭代更新，直到无法生成新的超边。正如引理 IV.1、推论 IV.1 和 IV.2 所证明的，当不再生成新的超边时，H_A 与由 r 中所有极小差异集构建的全局超图具有相同的极小击中集。因此，H_A 能够保证候选函数依赖生成的完备性。MMCS 会枚举 H_A 的所有极小击中集，每个极小击中集都根据定义 III.4 对应一个有效的极小函数依赖 $X \to A$。

增量更新机制 的完备性 依赖于与一次性发现过程相同的迭代式超图细化原理。该机制以 H_A 为起点（一次性发现过程的完备性 已在该超图上得到证明），并通过添加两类新的差异集对其进行迭代更新：一类来自 Δr 内的成对比较，另一类由 Δr 与 r 之间违反依赖的元组对生成。一旦不再出现新的超边，引理 IV.1 和推论 IV.1–IV.2 即可确保 MMCS 枚举出的极小击中集与从 $r \cup \Delta r$ 上的所有极小差异集推导出的极小击中集相同，从而保证增量更新机制中候选函数依赖生成的完备性。

引理 IV.1. 对于任意两个超图 $\mathcal{H}$ 和 $\mathcal{H}'$，$\mathcal{H} \lessdot \mathcal{H}'$ 当且仅当 $HS(\mathcal{H}) \supseteq HS(\mathcal{H}')$。 *[24]*

推论 IV.1. 部分超图 $\mathcal{P}_A$ 的极小击中集 X 是全局最小函数依赖（即 $X \in HS(\mathcal{D}_A)$）当且仅当函数依赖 $X \to A$ 在该数据集上成立。 *[24]*

推论 IV.2. 当算法迭代未产生新的差异集（即新的超边）时，当前超图 $\mathcal{P}_A$ 的极小击中集 $HS(\mathcal{P}_A)$ 构成全部最小函数依赖，即 $HS(\mathcal{P}_A) = HS(\mathcal{D}_A)$。 *[24]*

极小性。如性质 III.1所述，MMCS可保证候选函数依赖在一次性发现过程和增量更新机制中的极小性，因为它只枚举极小击中集。H_A 上的每个击中集 X 都直接对应一个最小函数依赖 $X \to A$。

非平凡性。为确保非平凡性，一次性发现过程 使用由在 A 上取值不同的元组对生成的差异集构造子超图 H_A，同时排除A。一次性发现过程 和增量更新机制 使用由在 A 上取值不同的元组对生成的新差异集更新 H_A，同时排除 A。随后，MMCS 在 H_A 上枚举击中集。根据定义 III.3 和 III.4，由于 $A \notin X$ 和 X 决定 A，因此函数依赖 $X \to A$ 是非平凡的。

确保 EAIFD 正确性的第二个关键步骤是候选验证；两个阶段中的验证策略都能准确判定候选项是否有效，并找出超图细化所需的所有违反依赖的元组对。

一次性发现过程 使用 IGHV 验证候选函数依赖。IGHV 首先将候选项分组，并为每组选择一个公共排序属性。然后，它依次加载按该属性排序的数据块，并在每个块内执行验证。在哈希函数 f 下，左部值相同的元组共享相同的哈希值，而左部值不同的元组具有不同的哈希值，从而确保排序后哈希值相同的元组位于同一块中。因此，验证只需检查当前块内的元组对是否存在违反情况，这将比较限制在局部范围内，大幅降低成本，同时确保正确性。如果某个块中包含违反的元组对，则该函数依赖被判定为无效；IGHV 会生成差异集以更新超图。如果所有块中均未发现冲突，则该函数依赖被视为有效。

增量更新机制 通过两步验证策略确保验证的正确性。候选函数依赖被划分为 C_1 和 C_2，并通过不同的方法进行验证。C_1 中的候选项通过基于 *MHT* 的验证，比较 MHT_Δ 中的项与 MHT中的项，从而实现快速验证。由于 MHT存储了来自r 的候选函数依赖中的高频哈希左部值到右部值映射项，而 MHT_Δ 存储了来自 Δr的所有哈希左部值到右部值映射项，因此二者之间的比较可以准确检测出 Δr 与 r之间违反的元组对。C_1 中无法通过基于 *MHT* 的验证验证的其余候选函数依赖，以及 C_2，，则使用表扫描验证进行验证，该方法通过选择性加载数据块来执行 IGHV。在对候选函数依赖进行分组并选择公共排序属性后，该方法不会加载 r的所有数据块。相反，它只选择性地加载那些包含元组的块，这些元组在公共排序属性上的值与 Δr 中的元组相同。只有这些块中的元组可能具有与增量数据相同的左部哈希值，因此需要验证其右侧属性，以检测违反的元组对。由于其他块中的元组具有不同的排序值，因而具有不同的左部哈希值，所以可以安全地跳过这些块。该方法在大幅降低输入/输出和计算成本的同时，保证了正确性。

因此，基于已证明的候选生成和验证步骤均正确，可证明 EAIFD 算法是正确的。

2) 复杂度分析： 本部分全面分析 EAIFD 的时间复杂度，涵盖一次性发现过程和增量更新机制，最后分析空间复杂度。我们考虑一个包含 r 个元组和 n 个属性的数据集 m，以及包含 n_Δ 个元组的增量数据 Δr 。

时间复杂度。 我们首先讨论一次性发现过程。其预处理步骤是按每个属性对 r 排序，以便后续验证，耗时为 $O(m \times n\log\ n)$。接着，对样本数据 s 中的元组进行两两比较，以计算差异集，复杂度为 $O(|s|^2)$。这避免了 DHSFD 需要进行所有元组两两比较所带来的 $O(m \times n^2)$ 复杂度，并使 EAIFD 能够快速初始化。

在预处理之后，一次性发现过程 会针对每个右侧属性 A 迭代执行候选生成和验证。在每次迭代中，候选生成依赖 MMCS 算法。正如 [25], 所证明的，MMCS 在子超图 $\mathrm{H}_A = (R, D_r^A)$ 上执行单次搜索步骤的时间复杂度为 $O(|\mathrm{H}_A|)$，其中 $|\mathrm{H}_A| = \sum_{D \in D_r^A} |D|$ 表示超边的总大小。发现一个极小击中集通常需要执行多个 MMCS 搜索步骤，其中大多数搜索步骤仅用于状态更新或剪枝。在一次迭代中生成k 个极小击中集的时间复杂度为$O(k \times |\mathrm{H}_A| \times \alpha)$，其中 α 是发现一个极小击中集所需搜索步骤数的平均值。

在同一轮迭代中，候选验证使用 IGHV 验证由 MMCS 生成的 k 个候选函数依赖。最坏情况下，每个候选函数依赖都需要在所有数据块中进行验证，复杂度为 $O(k \times n)$；而一旦发生冲突，就会立即剪枝。设 b 为每个候选函数依赖平均需要验证的数据块数，$|V|$ 为组 r 中公共排序属性的不同取值数。若数据块的近似大小为 $\frac{n}{|V|}$，则每个候选函数依赖的验证成本为 $O(b \times \frac{n}{|V|})$，因此验证 k 个候选项的平均复杂度为 $O(k \times b \times \frac{n}{|V|})$。

结合候选生成与验证，单次迭代的时间复杂度为 $O(k \times (|\mathrm{H}_A| \times \alpha + b \times \frac{n}{|V|}))$。将一次迭代的开销乘以迭代次数，即可得到一次性发现过程 的总开销，而迭代次数取决于底层数据。

接下来，我们分析增量更新机制的时间复杂度。该机制首先通过对 Δr 中的元组进行成对比较，计算新的差异集，时间复杂度为 $O(n_\Delta^2)$。然后，它使用 MMCS 迭代生成候选函数依赖并对其进行验证。由于前文已讨论 MMCS 的复杂度，此处我们着重关注验证。候选函数依赖分为 C_1 和 C_2，，并采用两步验证策略。随后，我们将形式化地刻画针对 k 个候选项的两个验证步骤的时间复杂度。

对于 C_1，中的候选函数依赖，基于 *MHT* 的验证 会为 Δr 中的每个 $X \to A$构建一个本地 MHT_Δ，开销为 $O(k \times n_\Delta)$。将 MHT_Δ与 MHT进行比较，需要遍历 MHT_Δ 个项并查询 MHT；由于需要进行哈希表查找，每个候选函数依赖的耗时为 $O(1)$，因此对 k 个候选项的总耗时为 $O(k)$。验证 C_1 中 k 个候选项的总体复杂度为 $O(k \times (n_\Delta + 1))$，效率很高，因为 n_Δ 通常很小。

表扫描验证用于处理 C_1 中的不确定候选项以及 C_2 中的所有候选函数依赖。假设候选项总数为 k ，首先针对 Δr 上的每个函数依赖构建一个局部 MHT_Δ ，其开销为 $O(k \times n_\Delta)$。候选项按公共左侧属性分组，并对每组依次进行验证。与会加载所有数据块的一次性发现过程不同，表扫描验证会选择性地仅加载包含 Δr 中属性值的数据块。令 V_Δ 和 V 分别表示 Δr 和 r 中公共排序属性的不同值。加载的数据块大约包含 $n \times \frac{|V_\Delta|}{|V|}$ 个元组。一旦候选项失效，便立即将其剪枝。令 b' 为每个候选项平均验证的数据块数。包含 $|\overset{k}{\mathcal{G}}|$ 个候选项的每组总开销为 $O(|\overset{k}{\mathcal{G}}| \times b' \times \frac{n}{|V|})$，因此总复杂度为 $O(k \times (\overline{n_\Delta} + b' \times \frac{\overline{n}}{|V|}))$。与在一次性发现过程中使用 IGHV 验证 k 个候选项的 $O(k \times b \times \frac{n}{|V|})$复杂度相比，表扫描验证

效率更高，因为 $n_\Delta \ll n$ 和 $b' \ll b$。

空间复杂度。EAIFD 使用 MHT 作为核心辅助结构，以加速候选函数依赖验证，同时控制内存使用量。若不进行筛选，每个函数依赖最多可存储 n 个哈希条目，最坏情况下的空间复杂度为 $O(|\mathbb{F}| \times n)$。为减少内存使用量，EAIFD 采用高频映射项保留策略，阈值为 $\theta \in (0, 1)$，仅缓存满足 $\frac{count_{(h)}}{n} \geq \theta$ 的哈希键 h。设 p 表示每个函数依赖的缓存项数量；由于 $p \times (\theta \times n) \leq n$，因此有 $p \leq \frac{1}{\theta}$，从而每个函数依赖的上界为 $O(\frac{1}{\theta})$。因此，整体空间复杂度为 $O(|\mathcal{F}_\theta|)$，与数据集大小无关n，且可扩展至大型数据集，使 EAIFD 即使面对大规模数据集也能将内存使用量控制在可控范围内。

总而言之，EAIFD 将预处理降至 $O(m \times n \log n)$，避免了成对比较所需的 $O(m \times n^2)$ 开销，从而实现快速初始化。其优化后的验证在一次性发现过程和增量更新机制中均实现了近线性时间复杂度。结合高频映射项保留策略，空间复杂度上界为 $O(\frac{|\mathcal{F}|}{\theta})$，使 EAIFD 能够在增量更新的数据集上保持正确性并高效扩展。

V. 性能评估

本节评估 EAIFD 在一台配备 Intel(R) Core(TM) i7-13700 CPU（16 核，3.00 GHz）、32 GB 内存并运行 64 位 Windows 11 的 DELL OptiPlex Tower Plus 7010 台式电脑上的性能。所有实验均使用 Java (JDK 8) 实现。

数据集。 我们在 20 个不同的真实数据集上评估 EAIFD，以考察其实际性能。表 I 总结了这些数据集的特性，它们在元组数量、属性数量和数据分布方面各不相同，从而能够进行全面的性能分析。所有真实数据集均可公开获取[3]。

NULL语义。 在关系数据库中，NULL 表示缺失或不适用的值，这类值在真实世界的数据集中很常见。不同的 NULL 处理策略会影响相关函数依赖的有效性。我们采用 NULL 等于 NULL 的语义，将所有缺失值视为相等，这是函数依赖发现中常见的默认设置 [27], [16]。

对比算法。我们选取静态函数依赖发现算法 FD-HITS [11] 以及两种增量函数依赖发现算法 DynFD [17] 和 DHSFD [18], 进行比较[4]。FDHITS 的表现优于其他静态算法 [11]。DynFD（一种基于属性的增量式算法）维护正覆盖和负覆盖，通过 PLI 生成并验证候选函数依赖；而 DHSFD（一种基于元组的增量式算法）则将函数依赖发现表述为一个针对差异集超图的击中集枚举问题，并采用动态枚举策略。将这三种算法与 EAIFD 进行比较，可以全面评估 EAIFD 的增量处理效果。

参数。 EAIFD 的性能受两个参数影响：抽样率 ε 和映射频率阈值 θ。给定一个包含 n条记录的数据集，记录对的总数为 $p = \binom{n}{2}$。EAIFD 在不重复抽样的情况下，均匀抽取约 p^ε 个记录对（$0 < \varepsilon < 1$），以构建初始部分超图。既有研究表明，将 $\varepsilon = 0.3$ 设置为该值能够在覆盖率与效率之间实现良好平衡 [24], [11]。映射频率阈值 θ决定存储在 MHT中的项目数量，从而控制剪枝强度和内存使用量。实验表明， $\theta = 80\%$能够实现有效平衡。因此，我们采用$\varepsilon = 0.3$ 和 $\theta = 80\%$作为默认设置。

其他设置。对于每个数据集，我们随机选择 80% 的元组作为 r ，并按照 DHSFD [18]中的方法，利用剩余的 20% 生成 Δr 。除非特别说明，所有时间均以秒为单位；TL 表示运行时间超过 5 小时的限制。每项结果均为三次执行的平均值，且每次执行前都会重启机器。

实验结构。我们开展了全面的实验，以评估 EAIFD 的效率、可扩展性和独特特性。在现实数据集上进行的实验 1 展示了 EAIFD 的性能优势。实验 2、3 和 4 分别分析了其在元组数量、属性数量和增量数据大小方面的可扩展性。实验 5 将 EAIFD 与当前最先进的静态算法在增量更新方面进行了比较。实验 6、7 和 8 分别展示了 EAIFD 的关键优势：快速初始化、采用高频映射项保留策略的低内存 MHT ，以及 θ 在大范围内的效果。

实验 1：总体性能比较。实验1在20个真实世界数据集上，将EAIFD与现有增量函数依赖发现算法进行比较，其中 $\frac{|\Delta r|}{|r|}$ 范围为10%至30%， $\varepsilon = 0.3$和 $\theta = 80\%$。如表I所示，在所有插入比例下，EAIFD的速度始终快于DynFD和DHSFD。

与DynFD相比，EAIFD在若干数据集（例如Plista）上的加速比最高可达两个数量级。在Flights、Census、Fd-Reduced-30和CAB等较大的数据集上，EAIFD的优势尤为显著，因为DynFD始终超出时间限制（TL）。这证实了DynFD因采用逐层候选生成和基于PLI的验证而存在可扩展性局限。相比之下，EAIFD将候选生成转化为超图上的击中集枚举问题，并采用两步验证策略，从而实现了更高的可扩展性。

尽管 DHSFD 和 EAIFD 都能够处理大规模数据集，但在多个数据集（例如 Nursery）上，EAIFD 的运行速度几乎比 DHSFD 快一个数量级。这一显著的加速效果验证了 EAIFD 中两步验证策略的有效性。基于 *MHT* 的验证 利用高效的 $O(1)$ 哈希查找，快速处理大多数候选项（C_1）。随后，表扫描验证 有选择地加载数据块，并批量验证剩余的函数依赖。该策略大幅提升了整体效率。

实验 2：元组数量的可扩展性 $|r|$。实验 2 通过改变 $|r|$，评估 EAIFD、DynFD 和 DHSFD 在 CAB 和 Pitches 数据集上的行可扩展性，如图 5(a) 和图 5(d) 所示。我们保持 $\frac{|\Delta r|}{|r|} = 20\%$ 不变，并固定 $|R|$ （CAB 为 54，Pitches 为 40）。具体而言， $|r|$ 的取值范围为

[3]数据集可从 https://hpi.de/naumann/projects/repeatability/data-profiling/fds.html 和 https://www.kaggle.com/datasets 获取。[4]所有代码均来自 https://hpi.de/naumann/projects/repeatability/ data-profiling/fds.html 和 https://github.com/RangerShaw/DHSFD。

表 I：在现实数据集上的实验结果。EAIFD 与两种增量算法 DynFD 和 DHSFD 的运行时间（秒）。最快的运行时间以粗体标出。TL 表示算法超出时间限制。

数据集属性				增量式（$\lvert\Delta r\rvert = 10\%\lvert r\rvert$）			增量式（$\lvert\Delta r\rvert = 20\%\lvert r\rvert$）			增量式（$\lvert\Delta r\rvert = 30\%\lvert r\rvert$）		
数据集	$\lvert r\rvert$	$\lvert R\rvert$	$\lvert\mathcal{F}\rvert$	DynFD	DHSFD	EAIFD	DynFD	DHSFD	EAIFD	DynFD	DHSFD	EAIFD
桥梁	108	13	142	0.2	0.0013	**0.001**	0.23	0.004	**0.003**	0.28	0.008	**0.005**
平衡	625	5	1	0.13	0.03	**0.006**	0.15	0.06	**0.007**	0.21	0.071	**0.009**
鲍鱼	4177	9	137	0.1	0.03	**0.008**	0.21	0.07	**0.011**	0.31	0.094	**0.024**
Iris	147	5	4	0.166	0.03	**0.009**	0.21	0.037	**0.009**	0.2	0.049	**0.012**
Nursery	13000	9	1	0.8	0.17	**0.015**	1.1	0.21	**0.018**	1.6	0.235	**0.027**
NCV	1000	19	758	0.3	0.04	**0.02**	0.5	0.09	**0.05**	0.54	0.11	**0.07**
肝炎	155	20	8250	1.2	0.25	**0.07**	2.5	0.3	**0.09**	2.8	0.384	**0.121**
Claim	20000	11	12	0.42	1.8	**0.27**	0.47	2.5	**0.45**	0.7	3.5	**0.507**
信件	20000	17	61	14	1.5	**0.53**	22.1	1.9	**0.77**	28	2.6	**1.21**
Plista	1001	63	173409	143.4	4.6	**0.5**	147.5	7.5	**1.3**	145.1	12.8	**3.2**
马	300	29	128726	30.1	2.1	**0.9**	33	4.1	**1.8**	34.5	5.5	**3.906**
生物条目	184292	9	19	69	5.6	**2.6**	83	14.8	**8.4**	103	34.1	**17.9**
Tax	1000000	15	263	346	52	**29**	387	93	**57**	494	176	**87**
双标签_特征	3960124	13	58	TL	358	**44**	TL	1439	**287**	TL	5204	**743**
Fd-Reduced-30	250000	30	89571	TL	144	**76**	TL	367	**174**	TL	583	**297**
航班	250000	31	117367	TL	190	**83**	TL	410	**198**	TL	630	**410**
Pitches	250000	40	608928	TL	310	**132**	TL	600	**420**	TL	920	**780**
CAB	67300	54	3353531	TL	330	**197**	TL	700	**580**	TL	1350	**990**
Census	196000	42	41861	TL	341	**206**	TL	622	**461**	TL	1077	**845**
Lineitem	6000000	16	3984	TL	1308	**968**	TL	3314	**2767**	TL	7123	**6594**

CAB 为 10,000 至 14,000（$\lvert\Delta r\rvert$ 从 2,000 至 2,800），Pitches 为 10,000 至 50,000（$\lvert\Delta r\rvert$ 从 2,000 至 10,000）。运行时间超过 5 小时的实验不予列出。

由于基于PLI的验证成本会随着 $\lvert r\rvert$迅速增长，DynFD在两个数据集上均无法在时限内完成，导致其行可扩展性较差。相比之下，DHSFD和EAIFD的性能随 $\lvert r\rvert$ 增加而平稳变化，表现稳定。值得注意的是，EAIFD始终优于DHSFD。随着 $\lvert r\rvert$ 增加，更新PLI、计算新的差异集以及优化超图所需的成本不断上升，限制了DHSFD的行可扩展性。EAIFD通过利用 Δr中元组对导出的差异集更新超图，避免了涉及r的冗余全对比较。随后，两步验证不会扫描初始数据集，而是利用 MHT 并加载相关数据块进行验证，从而确保较高的行可扩展性，并在 $\lvert r\rvert$ 增加时保持稳定的性能。

实验 3：属性数量的可扩展性 $\lvert R\rvert$。实验 3 通过改变 $\lvert R\rvert$，评估 EAIFD、DynFD 和 DHSFD 在 CAB 和 Pitches 数据集上的列可扩展性，同时固定 $\lvert r\rvert$和 $\lvert\Delta r\rvert$ （CAB 为 10,000 和 2,000；Pitches 为 50,000 和 10,000）。图 5(b) 和图 5(e) 分别展示了 CAB 的 $\lvert R\rvert$ 从 34–54 变化以及 Pitches 的 $\lvert R\rvert$ 从 20–40 变化时的结果。

DynFD 在两个数据集上的列可扩展性都较差。在 CAB 上，随着 $\lvert R\rvert$ (713s-8，831s) 增加，其运行时间急剧上升，并且在 $\lvert R\rvert = 54$时限内无法完成。在 Pitches 上，DynFD 在所有测试的属性数量下（$\lvert R\rvert$ 从 20 到 40）均未能在时限内完成。这种严重的性能下降源于其依赖基于PLI的验证：属性数量增加会导致候选数量和集合相交操作都迅速增长。

如图 5(b) 和 (e) 所示，基于元组的两种算法 DHSFD 和 EAIFD 均表现出显著优于基于列的 DynFD 的列可扩展性，并且随着 $\lvert R\rvert$ 增加，其运行时间仅呈中等幅度增长。EAIFD 始终优于 DHSFD，表现出更低的运行时间和增长趋势。随着涉及的属性增多，更新 PLI、差异集和超图的复杂性不断增加，DHSFD 的性能仍会受到影响。相比之下，EAIFD 通过两项关键设计实现了更优的列可扩展性。一方面，EAIFD 将函数依赖生成建模为部分超图上的击中集枚举，并有选择地生成新的候选项，从而避免候选项数量过度增长。另一方面，其两步验证策略以 MHT 结构取代开销高昂的基于 PLI 的集合交集操作，以实现快速验证；随后，在相关数据块上批量验证剩余的候选函数依赖，从而确保较高的列可扩展性。

实验 4：增量规模下的可扩展性 $\lvert\Delta r\rvert$。实验 4 评估了 EAIFD、DynFD 和 DHSFD 在 CAB（$\lvert r\rvert = 10$，000, $\lvert R\rvert = 54$, $\lvert\Delta r\rvert = 2$，000–10，000, $\frac{\lvert\Delta r\rvert}{r} = 20$ -100 $\lvert r\rvert = 50\,000$ $\lvert R\rvert = 40$ $\lvert\Delta r\rvert =$ —— 10 000 22 500 -45 5 % %) 和 Pitches（, , ,, –, , $\frac{\lvert\Delta r\rvert}{r} = 20\%$ %）上的增量可扩展性，如图 (c) 和图 5(f) 所示。

在两个数据集上，DynFD 都未能在时限内完成，这证实了其可扩展性较差，因为它在 $r \cup \Delta r$ 上进行的基于PLI的验证成本高得难以承受。随着增量比率 $\frac{\lvert\Delta r\rvert}{r}$ 增加，DHSFD 和 EAIFD 的运行时间均显著增长，这是因为差异集计算、超图细化和验证操作的成本更高。然而，EAIFD 始终优于 DHSFD，且运行时间增长率更低。EAIFD 的关键优势直接来自两个方面。首先，EAIFD 无需枚举所有跨集合对，并在 Δr 内更新差异集，而 DHSFD 则计算 Δr 与 r 之间的差异集。其次，两步验证策略能够高效地进行候选验证。即使 $\frac{\lvert\Delta r\rvert}{r}$ 增加，EAIFD 中基于 MHT 的验证也能通过快速冲突检测，高效处理大多数 C_1 候选项。随后，表扫描验证仅批量加载剩余候选项所需的数据块，从而减少不必要的输入/输出成本。

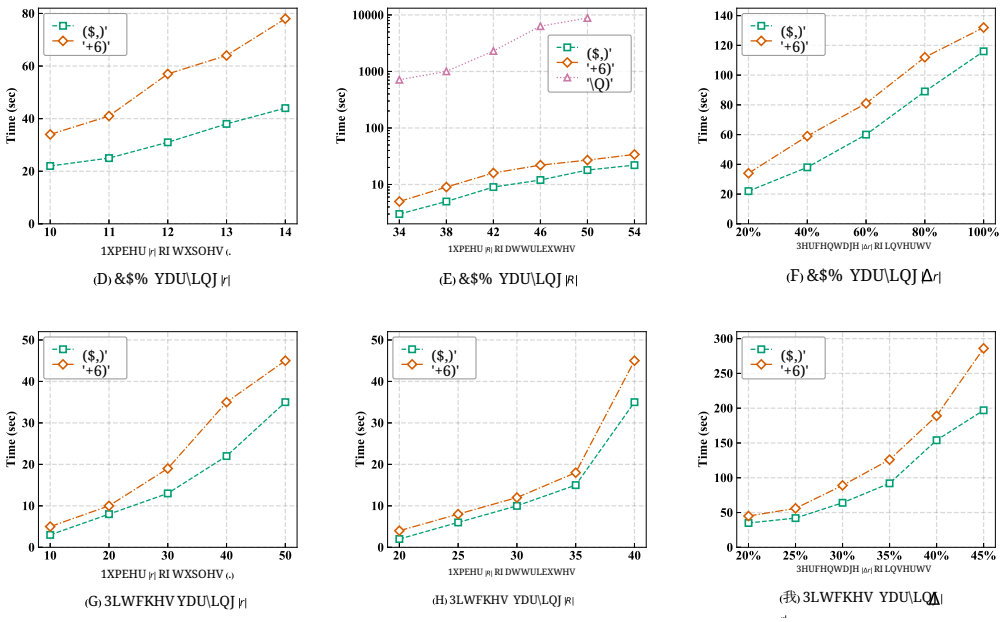

图 5：EAIFD 与 DynFD、DHSFD 在 CAB 和 Pitches 上的时间性能对比。子图 (a、b、c) 分别对应 CAB 中不同的元组数量 $|r|$、属性数量 $|R|$ 和增量数据大小 $|\Delta r|$；子图 (d 、 e 、 f) 则展示 Pitches 上的相同情况。

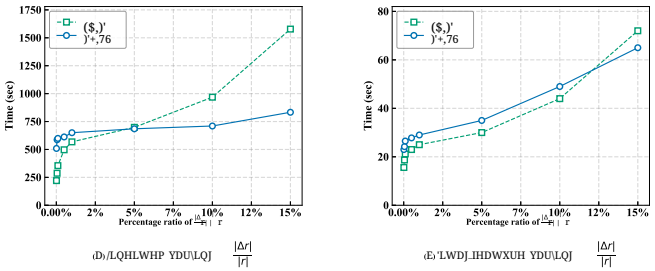

图 6：EAIFD 与 FDHITS 在不同增量比例 $\frac{|\Delta r|}{r}$ 下的时间性能对比。子图 (a) 和 (b) 分别对应 Lineitem 和 Ditag_feature。

实验5：EAIFD 与静态函数依赖发现算法的对比。实验 5 评估了 EAIFD 和 FD-HITS（一种先进的静态算法）在 Lineitem（$|r|$ =6，000，000, $|R|$ = 16）和 Ditag_feature（$|r|$ = 3，960，124, $|R|$ =13）上的性能。对于 FDHITS，选择*sep*变体，而非*joint* 变体，原因有二。首先，原论文 [11] 表明，*sep* 和*joint* 变体的性能极具竞争力，而前者在许多真实世界数据集上优于后者。其次，*joint* 变体的性能对采样数据特征非常敏感，而*seq* 变体则能提供更稳定、可预测的性能。我们固定 $|r|$ 和 $|R|$，并在很大的范围内改变增量比率 $\frac{|\Delta r|}{r}$（0.01% 0.05% 0.1% 0.5%，1%，5%，10%，15%），以确定 EAIFD 与 FDHITS 之间的性能平衡点。

图6(a)和(b)中的结果显示了两种截然不同的趋势。FDHITS 的运行时间较长，但随着增量比例变化，其增长幅度较小，因为它的运行时间取决于数据集总大小（$|r|+|\Delta r|$）。相比之下，EAIFD 在增量较小的情况下效率优势显著（Lineitem：$\frac{|\Delta r|}{r} \leq 5\%$；Ditag_feature：$\frac{|\Delta r|}{r} \leq 10\%$），运行时间大幅缩短——时间。随着增量比率上升，EAIFD 的运行时间迅速增长，最终超过 FDHITS。这是因为 EAIFD 的运行时间主要取决于增量数据大小（$|\Delta r|$）。尽管如此，该实验证实了 EAIFD 的实际价值：在典型应用中，增量更新 Δr 通常只占不断增长的大型数据集 r 的一小部分。因此，EAIFD 始终处于性能平衡点的有利一侧，更适用于真实场景。

表II：不同数据集上三种算法的预处理。

数据集属性				预处理时间		
数据集	$\|r\|$	$\|R\|$	$\|\mathcal{F}\|$	DynFD	DHSFD	EAIFD
Iris	147	5	4	0.06秒	0.03秒	**0.017秒**
桥梁	108	13	142	0.15小时	1.1s	**0.77秒**
肝炎	155	20	8250	0.45小时	1.6s	**0.88秒**
马	300	29	128726	5.1h	2.5s	**0.63秒**
平衡	625	5	1	**0.02秒**	0.62秒	0.03秒
NCV	1000	19	758	67.4秒	4.6s	**0.7s**
Plista	1001	63	173409	5.5h	6.7s	**2.8s**
Claim	20000	11	12	128s	123s	**62s**
信件	20000	17	61	164s	145s	**92s**
CAB	67300	54	3353531	6.5h	1.34小时	**0.77小时**
生物条目	184292	9	19	0.25小时	0.42小时	**0.12小时**
航班	250000	31	117367	6.6h	3.45小时	**0.34小时**
Pitches	250000	40	608928	7.9h	5.6h	**0.56 小时**
Ditag_特征	3960124	13	58	7.2h	7.4h	**0.67 小时**

实验 6：算法的预处理时间。实验 6 评估了 EAIFD、DynFD 和 DHSFD 在真实世界数据集上的一次性预处理开销。三种算法的启动机制有显著差异。DynFD 根据初始数据集计算正覆盖和负覆盖。DHSFD 比较所有元组对以构建超图。相比之下，EAIFD 首先按属性对数据集排序，然后仅对抽样数据进行成对比较。

表II中的实验结果表明，EAIFD 的预处理时间最短，在大型数据集上通常快一个数量级以上

。DynFD 在包含大量函数依赖的大型数据集上速度较慢，而 DHSFD 处理包含大量元组的长数据集时可能需要数小时，导致其在超大型数据集上的启动几乎不切实际。EAIFD 无需构建 PLI 或正覆盖和负覆盖等辅助结构，而是对每个属性进行排序（$O(|R| \times |r| \log |r|)$），并仅在小型采样数据集上进行成对比较（$O(|s|^2)$）。有限的样本量使预处理保持高效，从而在实践中能够快速初始化。

实验 7：辅助结构的内存使用量。实验 7 评估了 PLI 和 MHT（ θ = 80% 的 r）的内存使用量。由于 PLI 是常见的验证结构，因此可作为重要基准。为分析影响内存使用量的因素，我们记录 $|r|$、 $|R|$、 |F|以及数据集的内存大小。结果汇总于表 III。

实验结果表明，随着 $|r|$ 和 $|R|$的增加，PLI 的内存使用量会快速增长，这是由于需要维护完整的属性分区索引。相比之下， MHT 采用高频映射项保留策略，仅缓存频率超过阈值 θ 的有效函数依赖的关键值映射项。因此，每个函数依赖最多存储 $O(\frac{1}{\theta})$ 个映射项，使总体空间复杂度为 $O(\frac{|\mathcal{F}|}{\theta})$，且与数据集大小无关。因此，在 Clai—m 和生物条目等函数依赖较少的数据集上， MHT 所消耗的内存显著少于 PLI；即使对于大规模数据集，其内存使用量也仍然可控。

在复杂数据集或函数依赖较多的情况下， MHT的内存使用量可能超过 PLI，这也与其 $O(|\mathcal{F}_\theta|)$空间复杂度完全一致。然而，这正是我们的时间-内存权衡设计：EAIFD 以适度的内存开销换取显著的性能提升。

表 III：不同数据集上的辅助结构。

数据集属性				辅助结构		
数据集	$\|r\|$	$\|R\|$	|F|	Data	PLIs	MHT（θ = 80%）
Iris	147	5	4	5KB	1.7KB	**0.04KB**
桥梁	108	13	142	6KB	3KB	**1.4KB**
肝炎	155	20	8250	6.1千字节	**3.6千字节**	8.4KB
马	300	29	128726	18KB	**6.3KB**	1.2MB
平衡	625	5	1	7KB	2.3KB	**0**
NCV	1000	19	758	151KB	76KB	**10.3KB**
Plista	1001	63	173409	496KB	**312KB**	1.7MB
Claim	20000	11	12	2.6MB	1.5MB	**5KB**
信件	20000	17	61	696KB	221KB	**15KB**
CAB	67300	54	335353111	1.4MB	**8.9MB**	98MB
生物条目	184292	9	19	24MB	16.7MB	**6KB**
航班	250000	31	117367	19.7KB	**11KB**	1.1MB
Pitches	250000	40	608928	50MB	27.3MB	**19MB**
Ditag_特征	3960124	13	58	348MB	202MB	**14KB**

实验 8：频率阈值的影响 θ。实验8评估了在不同映射频率阈值下，EAIFD 保留 MHT中高频项的可扩展性。实验使用四个数据集：Plista、Pitches、CAB 和 Flight，增量比率为 $\frac{|\Delta r|}{|r|}$ = 10%。

表IV展示了当频率阈值 θ 从 70% 增至 90% 时， MHT 在 r 上的性能和内存消耗。当 θ = 70% 时，EAIFD 的效率最高，但内存使用量也最大，因为初始 MHT 中保留了更多高频映射项。这使得很大一部分验证能够

表IV：四个数据集上， MHT 在不同映射频率 θ 下的运行时间和内存使用量。

数据集	运行时间（秒）					MHT 的内存（MB）				
	70%	75%	80%	85%	90%	70%	75%	80%	85%	90%
Plista	0.09	0.32	0.50	2.91	4.51	1.90	1.75	1.70	1.43	1.27
航班	42	68	83	174	406	3.27	1.93	1.10	0.82	0.56
Pitches	72	95	132	357	630	54.7	31.6	19.0	13.6	10.8
CAB	78	144	197	522	927	272.6	134.2	98.7	77.3	45.5

通过哈希查找和 MHT_Δ 高效完成，而无需重新验证初始数据块。这体现了空间消耗与时间性能之间的权衡。随着 θ 增大，满足阈值的映射项减少， MHT 的内存使用量也随之降低。由于通过比较 MHT中的项能够验证的候选项减少，因此需要加载并验证更多数据块，从而增加运行时间。

结果清楚地表明， θ 是影响时间–内存权衡的高度敏感参数。综合考虑MHT内存使用量和运行时间，结果证实 θ =80%是一个平衡良好的参数选择。总体而言，高频映射项保留策略使 EAIFD 能够兼顾存储效率和计算性能。

实验分析总结。实验结果表明，EAIFD 在增量函数依赖发现方面表现优异。通过将函数依赖发现重新表述为部分超图击中集枚举，并采用 MHT 保留历史计算，再结合高效的验证方法，EAIFD 的表现优于现有的静态算法和增量算法。面对不同规模的元组、属性和增量数据时，EAIFD 均能保持稳定的性能，同时缩短预处理时间。此外， MHT的高频映射项保留策略兼顾了内存和计算效率，而核心频率阈值 θ 在广泛范围内均表现良好。总体而言，EAIFD 为增量函数依赖发现提供了一种高效、可扩展且实用的解决方案。

VI. 结论

本文提出了 EAIFD，这是一种用于在关系数据库中应对持续增量更新的高效函数依赖发现算法。EAIFD 克服了静态算法重复执行所带来的冗余开销，以及现有增量方法的性能瓶颈。它引入了两个核心组件：（1）将增量函数依赖发现建模为部分超图上的击中集枚举；（2）采用高效的两步增量验证策略。这种设计在实现高性能的同时降低了内存消耗。

大量实验表明，EAIFD始终优于现有的增量函数依赖发现方法（在 Plista 等数据集上，速度最高可提升一个数量级），并且在增量数据比例较小时，其性能显著优于静态算法。其效率优势源于核心操作主要取决于增量数据大小 $|\Delta r|$，而非完整数据集大小 $|r|$。这一特性进一步提升了长期应用中的效率和可扩展性，此类应用中的 $\frac{|\Delta r|}{r}$

会随时间推移而减少。在内存方面，EAIFD借助采用高频映射项保留策略的MHT，实现了不受$|r|$影响的低内存消耗。此外，EAIFD还可实现快速初始化，其预处理复杂度为$O(|R| \times |r| \log |r|)$。

尽管 EAIFD 在增量函数依赖发现方面取得了显著进展，未来仍有若干研究方向，例如支持删除和修改以实现全动态函数依赖发现，并探索并行或分布式实现，从而提升其在海量数据集上的可扩展性。

参考文献

[1] A. Silberschatz、H. F. Korth 和 S. Sudarshan，《数据库系统概念》第七版。麦格劳-希尔图书公司，2020。[2] T. Papenbrock 和 F. Naumann，“数据驱动的模式规范化”，载于《数据库技术拓展国际会议》第 20 届会议论文集，2017 年，第 342–353 页。[3] Z. Wei 和 S. Link，“嵌入式函数依赖与数据完备性导向的数据库设计”，《ACM 数据库系统汇刊》，第 46 卷，第 2 期，第 7:1–7:46 页，2021。[4] J. Kossmann、T. Papenbrock 和 F. Naumann，“用于查询优化的数据依赖：综述”，《VLDB 杂志》，第 31 卷，第 1 期，第 1–22 页，2022。[5] P. Bohannon、W. Fan、F. Geerts、X. Jia 和 A. Kementsietsidis，“用于数据清洗的条件函数依赖”，载于《国际数据工程会议论文集》，2007 年，第 746–755 页。[6] A. Doan、A. Y. Halevy 和 Z. G. Ives，《数据集成原理》。Morgan Kaufmann，2012。[7] Y. Huhtala、J. Kärkkäinen、P. Porkka 和 H. Toivonen，“TANE：一种用于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。[8] N. Novelli 和 R. Cicchetti，“函数依赖与嵌入式依赖推断：数据挖掘视角”，《信息系统》，第 26 卷，第 7 期，第 477–506 页，2001。[9] Z. Abedjan、P. Schulze 和 F. Naumann，“DFD：高效的函数依赖发现”，载于《ACM 信息与知识管理国际会议论文集》，2014 年，第 949–958 页。[10] T. Papenbrock 和 F. Naumann，“一种函数依赖发现的混合方法”，载于《2016 年 ACM 数据管理国际会议论文集》，2016 年，第 821–833 页。[11] T. Bleifuß、T. Papenbrock、T. Bläsius、M. Schirneck 和 F. Naumann，“通过击中集枚举发现函数依赖”，《ACM 管理数据会议论文集》，第 2 卷，第 1 期，第 43:1–43:24 页，2024。[12] J. Liu、F. Ye、J. Li 和 J. Wang，“从数据中发现函数依赖”，《Data Knowl. Eng.》，第 86 卷，第 146–159 页，2013。[13] 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。[14] C. M. Wyss、C. Giannella 和 E. L. Robertson，“Fastfds：一种启发式驱动的深度优先算法，用于从关系实例中挖掘函数依赖——扩展摘要”，载于《国际数据仓库与知识发现会议论文集》，2001 年，第 101–110 页。[15] 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。[16] Z. Wei 和 S. Link，“函数依赖的发现与排序”，载于《国际数据工程会议论文集》，2019 年，第 1526–1537 页。[17] P. Schirmer、T. Papenbrock、S. Kruse、F. Naumann、D. Hempfing、T. Mayer 和 D. Neuschäfer-Rube，“Dynfd：动态数据集中的函数依赖发现”，载于《国际数据库技术拓展会议论文集》，2019 年，第 253–264 页。[18] R. Xiao、Y. Yuan、Z. Tan、S. Ma 和 W. Wang，“通过动态击中集枚举进行动态函数依赖发现”，载于《国际数据工程会议论文集》，2022 年，第 286–298 页。[19] P. A. Flach 和 I. Savnik，“数据库依赖发现：一种机器学习方法”，《AI Commun.》，第 12 卷，第 3 期，第 139–160 页，1999。[20] S. Lopes、J. Petit 和 L. Lakhal，“函数依赖和阿姆斯特朗关系的高效发现”，载于《国际数据库技术拓展会议论文集》，《计算机科学讲义丛书》，第 1777 卷，2000 年，第 350–364 页。[21] T. Papenbrock、J. Ehrlich、J. Marten、T. Neubert、J. Rudolph、M. Schönberg、J. Zwiener 和 F. Naumann，“函数依赖发现：七种算法的实验评估”，《Proc. VLDB Endow.》，第 8 卷，第 10 期，第 1082–1093 页，2015。[22] H. Garcia-Molina、J. D. Ullman 和 J. Widom，《数据库系统：完整教程》（第 2 版）。培生教育，2009。

[23] H. Mannila 和 K. Räihä，“从关系中推断函数依赖的算法”，*Data Knowl. Eng.*，第 12 卷，99，1994 1，83第 – 页。[24] J. Birnick、T. Bläsius、T. Friedrich、F. Naumann、T. Papenbrock 和 M. Schirneck，“利用部分信息进行击中集枚举以发现唯一列组合”，*Proc. VLDB Endow.*，第 13 卷，第 11 期，第 2270–2283 页，2020。[25] K. Murakami 和 T. Uno，“对偶化大规模超图的高效算法”，*Discret. Appl. Math.*，第 170 卷，第 83–94 页，2014。[26] R. Agrawal、T. Imielinski 和 A. N. Swami，“在大型数据库中挖掘项目集之间的关联规则”，载于*Proc. 1993 ACM Int. Conf. Manag. Data*，1993 年，第 207–216 页。[27] L. Berti-Équille、H. Harmouch、F. Naumann、N. Novelli 和 S. Thiru-muruganathan，“从含缺失值的关系数据中发现真实函数依赖”，*Proc. VLDB Endow.*，第 11 卷，第 8 期，第 880–892 页，2018。