学校代码： 10246

硕 士 学 位 论 文

（学术学位）

针对动态增量数据集的逐点次序依赖发现

Pointwise Order Dependency Discovery on Dynamic Data

编　　号： 17210240012

专　　业： 计算机软件与理论

院　　系： 计算机科学技术学院

完 成 日 期： 2020 年 3 月 24 日

目录

第一章 引 言 **1**
第 1 节 背景介绍 1
第 2 节 问题简介 2
第 3 节 本文主要贡献与结构 5

第二章 相关工作 **7**
第 1 节 依赖的层次结构 7
第 2 节 相关约束说明 9
2.1 字典序次序依赖 9
2.2 基于集合的规范次序依赖 11
2.3 否定约束 12
第 3 节 约束的发现算法 13
第 4 节 不等式连接 14
第 5 节 本章小结 16

第三章 PODs 定义及全量、增量算法说明 **17**
第 1 节 定义与符号 17
第 2 节 逐点次序依赖的全量发现算法 20
第 3 节 逐点次序依赖的增量发现 21
第 4 节 本章小结 22

第四章 索引的建立与使用 **23**
第 1 节 使用索引目的说明 23
第 2 节 索引结构 25
2.1 数据预处理以及索引定义 25
2.2 最优索引的构建 27
第 3 节 最优索引的证明 30
第 4 节 索引发现矛盾 32
第 5 节 索引的共用说明 34
第 6 节 本章小结 36

第五章 选择、更新索引与增量发现算法 **37**
第 1 节 选择索引 37

第 2 节 跳表的查询与插入 40
2.1 常见的数据结构对比 40
2.2 跳表结构 41
第 3 节 索引更新 43
3.1 索引的更新说明 43
第 4 节 POD 的扩展 46
第 5 节 本章小结 50

第六章 实验分析 **51**
第 1 节 实验配置 51
第 2 节 实验结果 53
第 3 节 本章小结 58

第七章 总结与展望 **59**

摘要

逐点次序依赖 (PODs) 是一种描述数据集中元组不同属性值之间应满足的次序关系。POD 的全量发现是指在给定数据集 D 上识别有效和最小 PODs 集合 Σ 的过程。实际中数据集 D 可能很大并且会动态变化。新增加的数据会带来数据与原有约束之间的矛盾。使用全量的发现算法Hydra$*$ 重新计算新数据集 $D+\triangle D$ 上有效 PODs 集合会有很多冗余计算。本文提出一个 PODs 增量发现算法IncPOD 计算约束集合 Σ 的变化集 $\triangle\Sigma$，使得集合 Σ 上加入变化集 $\triangle\Sigma$ 的结果 $(\Sigma \oplus \triangle\Sigma)$ 是数据集 $D+\triangle D$ 的有效和最小 PODs 集合。增量发现算法流程为：先根据数据集 $D+\triangle D$ 与原始约束集 Σ 收集违背元组对集合 T；再使用违背元组对集合 T 计算约束变化集合 $\triangle\Sigma$。因为逐点次序依赖是描述属性值上的次序关系，所以如何收集违背元组对集合 T 是等同于一个不等式连接查询的难点问题。此外，因为修复不成立 PODs 时除了在左侧属性集 LHS 增加属性外，还需要更新 LHS 和右侧属性集 RHS 的运算符，所以 $\triangle\Sigma$ 的计算流程复杂。因此，逐点次序依赖的增量发现问题具有很高的技术难度。

本文研究了 POD 的增量发现问题，主要贡献如下：

(1) 给出了一套以约束集合 Σ 、数据集 D 以及增量数据集 $\triangle D$ 为输入的索引收集违背元组对集合 T 的方法。其中包含：建立数据集 D 中两个带运算符属性上的最优索引算法OptIndex; 依据索引收集违背元组对算法Fetch; 依据约束集合 Σ 和数据集 D 选择索引算法ChooseIndex 以及根据插入数据集 $\triangle D$ 更新索引的方法。文中证明了算法OptIndex 结果的最优性质、算法Fetch 的正确性、算法Fetch 利用索引查找违背元组对集合 T 的复杂度为 $log(|D|)$ 加上违背元组对集合 T 大小的收集时间。

(2) 给出了基于约束集合 Σ 和数据集 $D+\triangle D$ 计算约束变化集 $\triangle\Sigma$ 的增量发现算法IncPOD。算法IncPOD 首先通过索引计算出违背元组对集合 T；然后通过集合 T 发现约束集合 Σ 中不成立的 PODs ；最后通过算法ExtendPOD 扩展不再有效的 PODs 。文中证明了IncPOD 算法结果 $\Sigma \oplus \triangle\Sigma$ 为数据集 $D+\triangle D$ 上有效和最小的 POD 。

(3) 在真实和人工数据集上做大量实验说明索引收集违背元组对的效率、评估索引方法的合理性、增量发现算法IncPOD 的效率。实验结果表明本文提出的索引收集违背元组对的效率明显优于对比算法IEJoin 、评估索引打分函数合理、增量发现算法IncPOD 明显优于全量算法Hydra$*$ 。

关键字: 算法，数据修复，次序约束，数据约束

中图分类号 TP391

Abstract

Pointwise order dependencies (PODs) are dependencies that specify ordering semantics on attributes of tuples. POD discovery refers to the process of identifying the valid and minimal PODs set Σ on a given data set D. In practice D is typically large and dynamic. The inserted data may incurr new volations of Σ. Batch algorithm may take much time and space to recalculate the set of valid and minimal POD on $D + \triangle D$. This urges the need for an incremental algorithm that computes changes $\triangle\Sigma$ to Σ such that $\Sigma \oplus \triangle\Sigma$ is the set of valid and minimal PODs on D with a set $\triangle D$ of tuple insertion updates, instead of computing all PODs from scratch. Incremental algorithm is to identify the set T of violate tuple pairs first, then canculate $\triangle\Sigma$ with T. Finding T is a challenge since PODs contain inequality join. Also, a completely new refinement strategy for PODs is requires for computing $\triangle\Sigma$, since we can not only add more attributes to the LHS, but also refine operators for both LHS and RHS attributes.

In this research, we study the incremental POD discovery problem.

(1) At first, we propose a novel indexing technique for inputs Σ and D. We give algorithms to build and choose indexes for Σ and D, and to update indexes in response to $\triangle D$. We show that POD violations *w.r.t.* Σ incurred by $\triangle D$ can be efficiently identified by leveraging the proposed indexes, with a cost dependent on $log(|D|)$ and of violations number.

(2) Then we present an effective algorithm for computing $\triangle\Sigma$, based on Σ and identified violations caused by $\triangle D$. The PODs in Σ that become invalid on $D + \triangle D$ are efficiently detected with the proposed indexes, and further new valid PODs on $D + \triangle D$ are identified by refining those invalid PODs in Σ on $D + \triangle D$.

(3) Finally, using both real-life and synthetic datasets, we experimentally show that our approach outperforms the batch approach that computes from scratch, up to orders of magnitude.

Keywords: Algorithms, data repair, order dependency, Data dependency
CLC number TP391

第一章　引 言

第 1 节　背景介绍

互联网的广泛使用以及长时间的数据积累，使得生活中的数据量越来越大。相应地，数据质量带来的问题也越来越多。目前数据质量方面的研究主要分为数据一致性、数据准确性、数据冗余、信息完整和数据时效性 [39]。数据一致性问题研究如何使用数据准确地表达出现实中的信息。与现实中信息矛盾的数据称为不一致数据。为了能快速发现或者修复不一致数据，研究人员提出使用数据约束来描述数据集需要满足的条件。人工设计约束项往往费时并且容易受人为因素影响，所以自动发现数据集中有效约束的技术获得了广泛的关注。

数据约束的发现问题主要研究如何在数据集上发现有效的数据约束。目前约束发现研究主要是在唯一列组合 [21]、函数依赖 [24,25]、条件函数依赖 [5]、字典序次序依赖 [15,16]、否定约束 [18] 等约束的研究。数据约束的全量发现算法是指在静态数据集上发现完备的有效约束集合算法。实际中，很多数据集会随着时间而变化。所以在不同时间段，数据集满足的数据约束也可能会不同。

动态数据上约束发现问题研究如何在动态数据集中发现有效的数据约束。可以通过约束的全量发现算法在整个数据集上重新计算成立的约束集来解决这样的问题。使用全量发现算法解决问题的缺点在于即使改动小部分数据集也需要花费大量的时间重新计算。所以有必要研究动态数据集上的约束的增量发现算法解决该问题。约束的增量发现算法是以数据集 D、增量数据集 $\triangle D$、在数据集 D 上成立的约束集合 Σ 作为输入，发现数据集 $D+\triangle D$ 上有效约束的算法。增量发现算法流程为：先在数据集 $D+\triangle D$ 上发现数据与已有的约束集合 Σ 间的矛盾；再通过矛盾修复已有的约束集合。已有的增量发现算法研究主要有唯一列组合 [10]、函数依赖 [11]、字典序次序依赖 [32] 等约束。

相比于函数依赖只能描述属性值上相等关系之间的约束以及字典序次序依赖是定义在属性列上的次序关系，逐点次序依赖 (PODs) 表示的约束意义更为泛化。PODs 是定义在带运算符属性集合上描述不同元组属性值次序间的关系。在数据集中，定义次序有实际意义的属性很普遍，比如时间、工资和流水号等。在这些不同属性间可能有次序之间的相关性，比如订单处理时间越靠后，流水号越大；同一个员工工资与入职时间长短成正相关。PODs 有着良好的理论基础 [7]，但是目前没有动态数据集上逐点次序依赖的发现问题的相关研究。因此，本文研究了动态数据集上逐点次序依赖的发现方法：首先通过一种特殊索引查找数据集与约束间矛盾；然后通过扩展 PODs 修复矛盾。

第 2 节　问题简介

数据约束在数据科学领域有非常重要的作用，一般应用于模式设计 [1]，数据清洗 [2,3] ，一致性检测 [4] 以及数据修复 [40] 等相关问题。已有的研究中定义了多种数据约束来描述数据应满足的不同条件，比如函数依赖 (FDs)、条件函数依赖 (CFDs) [5] 、字典序次序依赖 (LODs) [9]、逐点次序依赖 (PODs) [7] 以及否定约束 (DCs) [6] 等等。其中逐点次序约束 (PODs) 是一种描述元组属性值之间次序关系的约束语言。大部分属性上判断属性值是否相等是有实际含义的，比如身份证号、姓名以及邮政编码等等。常见属性中按照属性值排序是有实际含义的，比如工资、日期以及时间等等。PODs 不仅具有良好的理论基础，还能表达属性值上相等关系、次序关系，因此具有广泛的实际应用价值。

首先通过一个示例来说明 PODs 的表达形式与实际含义。

表 1.1: 税率数据: D_1

TID	FName	LName	Date	SSN	NUM	ST	ZIP	SAL	RATE	TXA
t_0	Ali	Sam	20140410	719975883	1448	CO	80612	32000	1.24	3000
t_1	Eser	Dup	20140224	303975883	1401	CO	80612	50000	1.42	1500
t_2	Hen	Han	20140413	701178073	1486	ND	58671	30000	1.21	3400
t_3	Rene	Beke	20130403	801350874	1386	UT	84308	55000	2.04	1300
t_4	Ali	Sam	20140329	719975883	1427	CO	80612	7500	1.21	0
t_5	Hen	Han	20130404	701178073	1386	ND	58671	6500	0.24	900
t_6	Per	Mot	20150324	970122634	1547	CO	80209	95000	2.85	1100

表 1.2: 增量数据集: $\triangle D_1$

TID	FName	LName	Date	SSN	NUM	ST	ZIP	SAL	RATE	TXA
t_7	Yui	Uckun	20130322	435849162	1368	UT	84308	40000	1.42	3300
t_8	Hen	Han	20140218	701178073	1478	ND	58671	33000	2.04	1400
t_9	Rene	Beke	20150213	801350874	1533	UT	84308	31000	1.23	3200

示例 1-1: 考虑表格1.1中的税率数据 (关系实例 D_1)。每个元组表示一个具有以下属性的缴税记录：元组 ID(TID)，名字 (FName)，姓氏 (LName)，纳税日期 (Date)，社会保险号 (SSN)，流水号 (NUM)，州 (ST)，邮政编码 (ZIP)，工资 (SAL)，税率 (RATE) 和免税额 (TXA)。可以验证以下约束在 D_1 上成立：

σ_1: 邮政编码相同的人生活在相同的州

σ_2: 同一个人 (SSN) 的姓名相同

σ_3: 工资较低，免税额较高的记录税率较低

σ_4: 同一个人 (SSN) 的税号随纳税日期而增加

约束 σ_1 和 σ_2 是两个函数依赖：邮政编码 ZIP(社会保险号 SSN) 的相等 (=) 可以推导出州名 (姓名 FName,LName) 的相等 (=) 。然而，约束 σ_3 和 σ_4 的形式化需要更多不等运算符，以表达出元组在属性上值的其他排序语义。示例中的约束都可以表示为 PODs (第三章形式化定义 PODs)。具体表示如下：

σ_1: $\{\mathsf{ZIP}^{=}\} \hookrightarrow \{\mathsf{ST}^{=}\}$

σ_2: $\{\mathsf{SSN}^{=}\} \hookrightarrow \{\mathsf{FName}^{=},\mathsf{LName}^{=}\}$

σ_3: $\{\mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{<}\}$

σ_4: $\{\mathsf{SSN}^{=}, \mathsf{Date}^{<}\} \hookrightarrow \{\mathsf{NUM}^{<}\}$ □

为了充分利用 PODs 在数据集 D 上进行数据一致性检测和数据修复等工作。总希望能够提前得知在数据集 D 中成立的有效 PODs 集合 Σ。然而通过人工手动设计 PODs 依赖项通常太昂贵。人工设计不仅需要大量资深的领域专家参与，而且会非常耗时和麻烦。而且手动设计的依赖容易受到人为惯性思维错误的影响、发现不了全部成立的 PODs 以及不能够及时地随数据集的变化而快速地更新依赖集合。所以有必要研究如何自动发现数据集中成立的约束集合 Σ。相对于人工设计约束集而言，自动发现数据集中成立的约束集合不仅能够降低数据库维护的成本价格，还能够随数据集的变化自动更新约束集合。

实际中的数据集通常会随时间变化而更新。银行的流水数据、税收数据、股票交易额数据、医院病人每日病情数据等等都会随着时间变化进行更改。动态数据集是在原始数据集 D 上更新一个增量数据集 $\triangle D$。增量数据集 $\triangle D$ 是在关系数据库中的增加、删除的元组数据集。因为删除数据集中的元组不会引发约束和数据集之间的新矛盾。所以在原数据集上成立的约束仍然是删除元组后的新数据集上成立的约束。另外在一些特定数据集上可能禁止删除操作，例如数据仓库和区块链、个人银行消费记录、餐馆的排队数据等等。但是在绝大部分数据集上面，都是支持插入新的数据操作。所以在本文中，只探讨在元组插入的情况下，如何计算数据集 $D + \triangle D$ 上所有成立的有效的 PODs 约束集合。

一个最平凡的方法是利用约束发现的全量算法重新在数据集 $D + \triangle D$ 计算出所有的 PODs 。因为全量算法会重新计算原始数据 D 与原始数据 D 之间的比较，所以这样的方法总是计算昂贵的。而且因为不同约束的全量发现算法复杂度总是和整体数据集大小有关。所以即使增加少量数据，约束发现的全量算法也需要很长的时间去重新发现数据集 $D + \triangle D$ 上的约束集合。

直观来讲，当数据集 $\triangle D$ 的元组数目远小于数据集 D 的元组数目时，一个更好的发现约束方法是计算原始数据集上成立的约束集合 Σ 的变化集合 $\triangle\Sigma$。这个变化集合使得 $\Sigma' = \Sigma \oplus \triangle\Sigma$ 是数据集 $D + \triangle D$ 上成立的 PODs 集合。此处使用记号 "$\oplus$" 是因为在数据集 $D + \triangle D$ 上集合 Σ 中的某些 POD 不再成立，所以需要从集合 Σ 中移除；同时要将一些新扩展的成立 PODs 增加到 Σ。即在集合 Σ 中不仅有增加新成立 PODs 的操作，还有移除不成立的 PODs 操作，所以不能直接使用集合的并集符号 $\cup$ 来连接 Σ 和 $\triangle\Sigma$。

全量约束发现方法是只通过数据集 D 来发现依赖关系或者约束关系的方法。与全量约束方法不同，通过增量数据集 $\triangle D$ 和已有的约束集合 Σ 来发现约束的变化集合 $\triangle\Sigma$ 的方法称为增量依赖发现 [10,11]。在元组插入时，增量 POD 的发现比其他约束的增量约束发现更为复杂。如下示例所示：

示例 1-2: 考虑在数据集 D_1 上增加数据 $\triangle D_1$ (表格1.2)。可以验证例子1-1中的约束集 $\Sigma = \{\sigma_1, \sigma_2, \sigma_3, \sigma_4\}$ 是在数据集 D_1 上的有效 POD 集合。可以发现在数据集 $D_1 + \triangle D_1$ 上，约束 σ_1、σ_2 和 σ_4 仍然是有效的 POD 。因为元组对 t_1, t_7 和元组对 t_3, t_8 违背了 σ_3，所以 σ_3 不再是有效的 POD 。

以元组对 t_1,t_7 为例，元组 t_1 在属性SAL 上的值 50000 大于元组 t_7 在属性SAL 上的值 40000。元组 t_1 在属性TXA 上的值 3000 小于元组 t_7 在属性TXA 上的值 3300。根据约束 σ_3：工资越低，免税额越高的记录税率越低。可以推出元组 t_1 在属性Rate 上的值应该大于元组 t_7 在属性Rate 上的值。然而实际数据集中，元组 t_1 与元组 t_7 在属性Rate 上的值均为 1.42。

元组对 t_1,t_7 和 t_3,t_8 为数据集与约束 σ_3 的矛盾，也称为违背元组对。为了解决违背元组对 t_1,t_7 和 t_3,t_8 与约束间的矛盾，需要修复不成立的约束 σ_3。因为约束 σ_3 在数据集 $D_1 + \triangle D_1$ 上不成立，所以需要将约束 σ_3 从集合 Σ 中移除。其次，更重要的是发现在数据集 $D_1 + \triangle D_1$ 上新有效 PODs ，并将新有效 PODs 增加到约束集合 Σ。总体而言，需要将数据集 $D_1 + \triangle D_1$ 上不成立的 PODs 依据违背元组对扩展成新的有效 PODs 。

以不成立的约束 σ_3 为例，数据集 $D_1 + \triangle D_1$ 上以下两个新 PODs 成立：

σ_3'：$\{\mathsf{ST}^{=}, \mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{<}\}$

σ_3''：$\{\mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{\leq}\}$。

其中 σ_3' 是在约束 σ_3 的左侧属性集 (LHS) 上面增加属性，使得 LHS 的条件被加强，从而减少满足 LHS 条件的元组对。σ_3'' 是在 σ_3 的右侧属性集 (RHS) 上将运算符'$<$' 修改为'$\leq$'，从而增加满足右侧属性集的条件的元组对。

可以看到在数据集 D_1 上，约束 σ_3' 和 σ_3'' 都是有效的约束。很显然，在数据集 $D_1 + \triangle D_1$ 上成立的任何 PODs 都在数据集 D_1 上成立。但是发现约束算法一般只发现最小的 PODs ，所以约束 σ_3' 和 σ_3'' 都不在集合 Σ 中。约束 σ_3 在数据集 D_1 上成立并且逻辑蕴含了 σ_3' (或者 σ_3'')，所以 σ_3' (或者 σ_3'') 都不是数据集 D_1 上的最小 POD 。约束 σ_3 逻辑蕴含 σ_3' (或者 σ_3'') 也就是任何满足约束 σ_3 的数据集一定满足 σ_3' (或者 σ_3'')。当 σ_3 在数据集 $D_1 + \triangle D_1$ 上不再成立时，σ_3' 和 σ_3'' 变成了新的有效最小 PODs ，需要加入到集合 Σ 中。 □

根据上面的示例可知，增量 PODs 的发现难点在于：

1、查找矛盾元组对分为增量数据与原始数据之间的矛盾元组对以及增量数据与增量数据的矛盾元组。

2、增量 PODs 的扩展需要保证 PODs 的最小性以及最后结果集的完备性。和其他约束的扩展只需要变化属性不同，扩展 PODs 还需要进行运算符的变化。

第 1 点中查找数据集与 PODs 的矛盾大部分情况是查找非等于符号的违背元组对。普通索引和方法无法满足查找非等于符号的元组对需求。实际上，查找非等于符号的矛盾元组对等同于不等连接查询问题。在数据集上进行不等连接的查询本身就已经是一个非常值得探讨的问题，在第二章第四节中会更加详细探讨不等连接查询问题与已有的解决方法。

第 2 点中约束 PODs 修复算法相较于其他约束修复算法更为复杂。已有的约束修复算法扩展不成立的约束只需要增加约束 LHS 中的属性，比如函数依赖等。因为 PODs 表达形式以及最小性的特殊定义，扩展过程中属性上的运算符号也需要进行变动。因此需要更加细致地分情况处理 PODs 的扩展。

为了解决以上两个难点问题，本文分别做了:

1、收集违背元组对是增量 POD 发现的关键步骤。本文提出一种建立在两个带运算符属性上的索引。可以利用这样的索引在 $log(|D|)$ 的时间内发现由增量数据集 $\triangle D$ 中元组引发的矛盾所在位置，其中 $|D|$ 为数据集 D 的元组数目。并可以在与矛盾集合大小时间内收集增量数据集带来的矛盾。同时在第五章中说明索引表示为跳表结构。跳表结构可以保证收集矛盾元组对的同时更新索引。索引查找矛盾与更新可以同步进行的机制保证了增量数据自身间的矛盾元组对可以通过索引一并查找出来，而不需要进行多余的操作。在第四章会详细说明索引的效率。第六章的实验中验证了本文的索引在增量数据集上面的不等连接查询效率比对比算法高很多。

2、第五章给出了基于 Σ 和矛盾元组对生成 $\triangle\Sigma$ 扩展 PODs 的算法ExtendPOD 并且证明了该扩展算法结果的正确性。算法ExtendPOD 讨论了如何在不同矛盾的情况下扩展 PODs 。

第 3 节 本文主要贡献与结构

本文主要研究了增量 POD 的发现算法，基本流程为：在已知数据集 D 上的约束集合 Σ，根据新增数据集 $\triangle D$，利用索引快速地发现与收集所有数据上的违背元组对；然后根据违背元组对以及已有的约束集合 Σ 修复矛盾。下面说明本文的主要贡献与文章结构：

(1) 第一章主要介绍了文章研究背景，并对本文问题做了简略介绍并通过实例说明问题的难点以及本文的解决方法。

(2) 第二章主要说明目前国内外对逐点次序依赖增量发现的相关工作以及对不等连接的相关研究工作。逐点次序依赖增量发现的相关工作主要包含依赖间的层次结构说明、已有的不同约束全量和增量发现算法的简介、相关约束与逐点次序依赖的包含关系和转换方法等。不等连接的相关研究工作主要介绍了不等连接问题的定义和算法IEJoin 。

(3) 第三章详细说明了本文使用的符号以及给出了逐点次序依赖的定义、依

赖间的层次关系、最小逐点次序依赖、违背元组对等定义。并依据否定约束与逐点次序依赖的关系给出了全量发现算法Hydra∗ 。

(4) 第四章详细说明了本文使用的索引结构，包含了索引的建立、最优索引的证明、索引收集矛盾的算法以及索引的共用。

(5) 第五章说明如何通过已有约束集合 Σ 选择索引、索引的表示、索引的更新以及根据发现的矛盾元组扩展不成立的逐点次序依赖。

(6) 第六章在现实生活和人工的数据集上进行了大量的实验。实验包括验证索引的有效性和效率、增量 PODs 发现方法IncPOD 和第三章给出的全量方法Hydra∗ 的对比、索引打分函数的性能。

(7) 第七章对全文进行总结并对在本文基础上可以开展的研究做了展望。

第二章　相关工作

约束发现是数据科学领域重要的研究之一，目前定义了多种数据约束来表示实际中的数据应满足的条件，比如近似函数依赖，条件函数依赖，分布式函数依赖，次序依赖，匹配原则，图上的函数依赖等等 [12,13]。本节将展示与研究问题相近的工作：常见的依赖间的层次结构、已有的次序依赖和否定约束 DCs 的发现、增量上的约束发现以及不等符号连接查询的技术。

第 1 节　依赖的层次结构

目前常见的依赖关系之间有着密切关系，下面介绍一些常见约束间的层次关系以及约束的全量发现、增量发现算法。

键依赖 (UCC)、函数依赖 (FD) 是目前最基础，研究最广泛的依赖，之后的依赖基本上都是这两种依赖的泛化。键依赖是形如 $\{C_1, C_2, \cdots\}$ 的属性组合，其中这样的属性组合为数据集上的候选键或者主键。函数依赖是将键依赖泛化为属性集 X 上的值可以决定属性集 Y 上的值，表达形式为 $X \to Y$。

其余文献中定义了一种不同的次序依赖，称为字典序次序依赖 (LOD) [9,14]。与 PODs 是定义在属性集上的次序关系不同，LODs 是定义在属性列表上。一个字典序次序依赖是定义在属性列表 X 和属性列表 Y 的依赖，表达形式为 $X \mapsto Y$。下一节使用具体实例证明 PODs 严格泛化了 LODs [9]。这意味着任何一个 LOD 都可以表示为若干 PODs 的集合。任何满足该 LOD 的实例，也应该同时满足集合中的所有 PODs 。因为存在 PODs 无法使用 LODs 进行表示，所以这样的泛化是严格的。基于集合的规范次序依赖 (SODs) 是基于 LODs 提出的一种次序依赖 [15,16]。这样的依赖关系是 LODs 的一个泛化。同样的，PODs 也严格地泛化了基于集合的规范次序依赖。本章第二节具体说明三种不同次序依赖的表达形式、包含关系以及转换方法。

否定约束 (DCs) [17,18] 是一种一阶逻辑形式，使用否定谓词之间的连接构成的逻辑表示的数据约束。表达形式为 $\neg(P_1 \wedge P_2 \cdots \wedge P_m)$。每一个谓词 P_i 是由两个属性值和运算符组成或者由一个属性值，常量和运算符组成。但是本文使用的 DCs 不涉及带有常量的否定约束的情况，所以本文的 DC 的谓词 P_i 是由两个属性值和一个运算符组成。DCs 泛化了 PODs 及上述的所有依赖，在示例2-1中以及本章第二节可以看到每一个 POD 以及上述的依赖都可以转换为一个 DC 。在限定了 DCs 的谓词条件以及组成 DCs 的条件后，DCs 可以和 PODs 进行对等变换。具体的限定条件在本章第二节以及第三章中具体说明。

图2.1 是上述约束间的层次关系图。

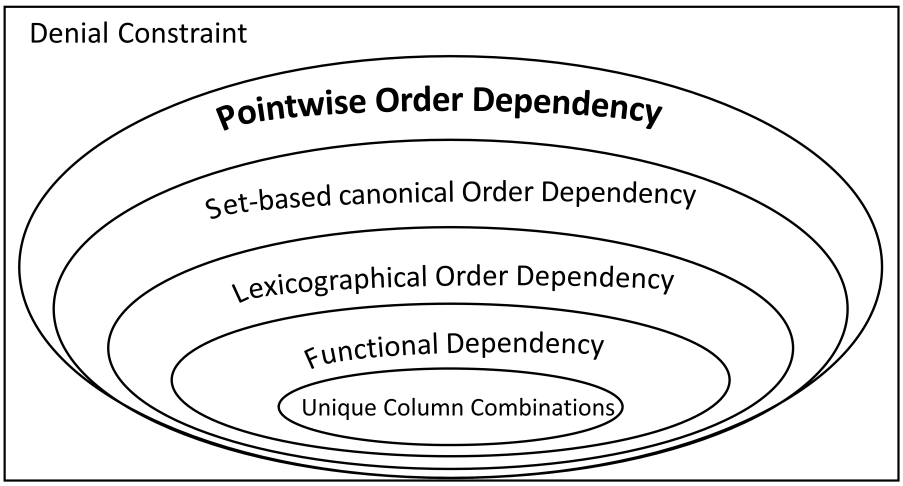


图 2.1: 不同约束间的层次关系

下面通过示例来说明图2.1中不同约束的表达形式。

表 2.1: 职工信息表 r

TID	EmpId	ManId	Day	Salary	City	ZIP
t_0	M101	M101	20130524	10000	BeiJing	100000
t_1	E4001	M101	20140628	8176	BeiJing	100000
t_2	E4001	M101	20140830	9100	BeiJing	100000
t_3	M102	M102	20130403	9000	ShangHai	200000
t_4	E9020	M102	20140329	8000	ShangHai	200000
t_5	E9020	M102	20140712	8500	ShangHai	200000

示例 2-1: 以表格2.1中的职工信息表 r 为例，探讨不同的数据约束的表示形式。可以验证以下约束成立：

σ_5：同一个时间同一个员工号只出现一次。

σ_6：同一个员工只有一个经理。

σ_7：同一个员工的工资随时间是递增的。

其中 σ_5 是键依赖 (唯一列组合),即图2.1中的 $Unique\,Column\,Combinations$，表达形式为 $\{EmpId,Day\}$。表示含义为 $\forall t_1, t_2 \in r, (t_1 \neq t_2) \Rightarrow (t_1[EmpId, Day] \neq t_2[EmpId, Day])$，即在数据集 r 中的任何两个不同元组 t_1, t_2，员工号 $EmpId$ 和日期 Day 不会同时相同。

σ_6 为函数依赖,即图2.1中的 $Functional\ Dependency$,其表达形式为 $EmpId \rightarrow ManId$。表示含义为 $\forall t_1, t_2 \in r$，如果有 $t_1[EmpId] = t_2[EmpId]$ 成立，那么一定有 $t_1[ManId] = t_2[ManId]$ 成立。

σ_7 需要比等于"=" 更多的运算符来表示。可以使用逐点次序依赖表示为 $\{\mathsf{EmpId}^{=},\mathsf{Day}^{>}\} \hookrightarrow \{\mathsf{Salary}^{>}\}$。表示含义为 $\forall t_1, t_2 \in r$，如果有 $t_1[EmpId] = t_2[EmpId]$ 以及 $t_1[Day] > t_2[Day]$ 成立，那么一定有 $t_1[Salary] > t_2[Salary]$。

上面三个约束均可转为否定约束的形式，即图2.1中的 *Denial Constaint*：

σ_5：$\forall t_1, t_2 \in r, \neg(t_1[EmpId] = t_2[EmpId] \wedge t_1[Day] = t_2[Day])$。

σ_6：$\forall t_1, t_2 \in r, \neg(t_1[EmpId] = t_2[EmpId] \wedge t_1[ManId] \neq t_2[ManId])$。

σ_7：$\forall t_1, t_2 \in r, \neg(t_1[EmpId] = t_2[EmpId] \wedge t_1[Day] > t_2[Day] \wedge t_1[Salary] < t_2[Salary])$。 □

第 2 节 相关约束说明

目前在属性值上的次序依赖关系研究主要分为：字典序次序依赖、基于集合的规范次序依赖、逐点次序依赖。本节介绍字典次序依赖，基于集合的规范次序依赖的定义以及与逐点次序依赖的包含关系，转换方法。因为否定约束也可以表示属性值上的次序关系，所以本节同样说明否定约束的相关内容。

2.1 字典序次序依赖

字典序次序依赖和逐点次序依赖都可以表达属性值之间的次序关系。字典序次序依赖与右侧属性列的顺序有关系；逐点次序依赖与右侧属性的顺序没有关系。接下来说明字典序次序依赖的定义、逐点次序依赖和字典序次序依赖之间的联系和互相转换转换方法。

字典序次序依赖定义 对于属性集 R 上的两个属性列 X 和 Y，$X \mapsto Y$ 表示一个字典序次序依赖，其含义为属性列 X 上的次序关系决定了属性列 Y 上的次序。如果 $X \mapsto Y$ 与 $Y \mapsto X$ 同时成立，则记为 $X \leftrightarrow Y$ [19,20]。

字典序次序依赖与函数依赖有着非常强的包含关系，一个字典序次序依赖 $X \mapsto Y$ 成立那么函数依赖 $X \rightarrow Y$ 也成立。函数依赖有属性值等于的决定关系: 属性 X 决定 Y。字典序次序依赖将其泛化为在整个数据集上根据属性列 X 的次序排序所有元组得到的结果在属性列 Y 上也有序。

表 2.2: 基本工资信息表 r_1

TID	Year	Month	Day	Date	Salary
t_0	2013	5	24	20130524	7000
t_1	2014	5	30	20140530	8176
t_2	2015	6	28	20150628	8500
t_3	2015	12	28	20151228	9500

示例 2-2: 表格2.2的数据集 r_1 描述了某公司员工的基本工资信息。有如下两个字典序次序依赖成立：

工资发放时间与工资有字典序次序依赖 $[Date] \mapsto [Salary]$。实际含义为某公司员工的基本工资会随着时间而提高。可以转换为逐点次序依赖的表示：$\{\mathsf{Date}^{=}\} \hookrightarrow \{\mathsf{Salary}^{=}\}$ 以及 $\{\mathsf{Date}^{<}\} \hookrightarrow \{\mathsf{Salary}^{<}\}$。

同样的，因为日期是先按照年份、月份排序，最后再按照天数进行排序，所以字典序次序依赖 $[Year, Month, Day] \mapsto [Date]$ 成立。 □

字典序次序依赖与逐点次序依赖的转换 一个左侧有 m 个属性，右侧有 n 个属性的字典序次序依赖可以转换为 $m * n + 1$ 个逐点次序依赖组成的集合。字典序次序依赖可以根据下面的转换方法转为多个逐点次序依赖组成的集合。

示例 2-3: 对于字典序次序依赖：$[A_1, A_2, A_3] \mapsto [B_0, B_1]$ 可以转换为如下等价的 7 条逐点次序依赖组成的集合 $\{\Phi\}$:

ϕ_1:$\{\mathsf{A}_1^{<}\} \hookrightarrow \{\mathsf{B}_0^{\leq}\}$

ϕ_2:$\{\mathsf{A}_1^{<}, \mathsf{B}_0^{=}\} \hookrightarrow \{\mathsf{B}_1^{<}\}$

ϕ_3:$\{\mathsf{A}_1^{=}, \mathsf{A}_2^{<}\} \hookrightarrow \{\mathsf{B}_0^{\leq}\}$

ϕ_4:$\{\mathsf{A}_1^{=}, \mathsf{A}_2^{<}, \mathsf{B}_0^{=}\} \hookrightarrow \{\mathsf{B}_1^{\leq}\}$

ϕ_5:$\{\mathsf{A}_1^{=}, \mathsf{A}_2^{=}, \mathsf{A}_3^{<}\} \hookrightarrow \{\mathsf{B}_0^{\leq}\}$

ϕ_6:$\{\mathsf{A}_1^{=}, \mathsf{A}_2^{=}, \mathsf{A}_3^{<}, \mathsf{B}_0^{=}\} \hookrightarrow \{\mathsf{B}_1^{\leq}\}$

ϕ_7:$\{\mathsf{A}_1^{=}, \mathsf{A}_2^{=}, \mathsf{A}_3^{=}\} \hookrightarrow \{\mathsf{B}_0^{=}, \mathsf{B}_1^{=}\}$

如果元组对违背了这 7 条逐点次序依赖中的若干依赖等价于元组对违背了字典序次序依赖：$[A_1, A_2, A_3] \mapsto [B_0, B_1]$。可使用类似本例的方法将任意字典序次序依赖转换成逐点次序依赖，已有具体转换算法 [9]。 □

逐点次序依赖严格包含字典序次序依赖 对于任意一个字典序次序依赖 $X \mapsto Y$，都可以通过转换方法进行转换为多个逐点次序依赖。在表格2.3的人工数据集上逐点次序依赖 $\{\mathsf{A}^{>}, \mathsf{B}^{>} \hookrightarrow \mathsf{C}^{\geq}\}$ 成立 [9]。表格中数据是按照属性 A 进行排序；当属性 A 上值相同时，再按照属性 B 排序。但是这样的数据集在属性 C 上并不是有序的。因此无法找到包含属性 A, B, C 的字典序次序依赖。从而说明逐点次序依赖是严格包含字典序次序依赖。

表 2.3: 说明数据

TID	A	B	C
t_0	0	0	0
t_1	0	1	1
t_2	2	2	2
t_3	3	2	3
t_4	4	4	4
t_5	5	5	4

TID	A	B	C
t_6	6	6	6
t_7	7	6	6
t_8	8	8	8
t_9	8	9	8
t_{10}	10	10	10
t_{11}	10	10	11

TID	A	B	C
t_{12}	12	12	13
t_{13}	12	13	12
t_{14}	14	14	15
t_{15}	15	14	14
t_{16}	16	17	16
t_{17}	17	16	16

2.2 基于集合的规范次序依赖

基于集合的规范次序依赖在字典序次序依赖基础上进行泛化，是严格包含字典序次序依赖 [15, 16]。下面说明基于集合的规范次序依赖定义，基于集合的规范次序依赖与字典序次序依赖、逐点次序依赖的转换和包含关系。

记号说明 对于属性集 R 上的两个属性列 X 和 Y，如果在数据集 D 上满足字典序次序依赖 $XY \leftrightarrow YX$，则记为 $X \sim Y$。属性 A 在 $\mathcal{X}$ 的每个等价类上都为常数指：在属性集 $\mathcal{X}$ 上的一个属性 A，对于 X 的任意排列 X' 均有 $X' \mapsto X'A$ 成立，记为 $\mathcal{X} : [\,] \mapsto A$。如果对于属性 A, B 和 X 的任意排列 X' 均有 $X'A \sim X'B$ 成立，则称 A 和 B 关于 $\mathcal{X}$ 是次序相容的，记为 $\mathcal{X} : A \sim B$。

基于集合的规范次序依赖定义 称这两种形式的次序依赖 $\mathcal{X} : [\,] \mapsto A$ 和 $\mathcal{X} : A \sim B$ 为基于集合的规范次序依赖 [15]。

将数据集 D 先根据属性列 X 上的次序排序，在属性列 X 上相等时再按照属性列 Y 上的次序排序得到的结果记为 D'。将数据集 D 先根据属性列 Y 上的次序排序，在属性列 Y 上相等时再按照属性列 X 上的次序排序得到的结果记为 D''。$X \sim Y$ 实际含义为 $D' = D''$。

示例 2-4: 以表格2.2数据集 r_1 为例。将数据集 r_1 上先按照年份 $Year$ 排序，在年份相同的情况下按照月份 $Month$ 排序的结果记为 r_1'。将数据集 r_1 先按照 $Month$ 排序，在 $Month$ 相同的情况下按照年份 $Year$ 排序的结果记为 r_1''。实际中，同一年同一个月的工资只发放一次，所以有 $r_1' = r_1''$。在 r_1 上成立基于集合的规范次序依赖 $Year \sim Month$。

然而，在数据集 r_1 上只按照月份 $Month$ 排序后的结果在属性 $Year$ 上是不能保证次序的，即字典序次序依赖 $Month \mapsto Year$ 不成立。同样的，在数据集 r_1 上只按照属性 $Year$ 排序后的结果在月份 $Month$ 上也是不能保证次序的，即字典序次序依赖 $Year \mapsto Month$ 也不成立。 □

与字典序次序依赖、逐点次序依赖的转换 字典序次序依赖可以通过如示例2-5转为若干个基于集合的规范次序依赖 [15]。

示例 2-5: 一个字典序次序依赖 $[AB] \mapsto [CD]$ 可以转为：$\{A, B\} : [\,] \mapsto C$；$\{A, B\} : [\,] \mapsto D$；$\{\} : A \sim C$；$\{A\} : B \sim C$；$\{C\} : A \sim D$；$\{A, C\} : B \sim D$。□

因为基于集合的规范次序依赖是在字典序次序依赖上的定义上进行扩展的，所以仿照字典次序依赖与逐点次序依赖的转换方法即可完成基于集合的规范次序依赖到逐点次序依赖的转换。

逐点次序依赖严格包含基于集合的规范次序依赖 在表格2.3中，$AB \sim C$，$AC \sim B$，$BC \sim A$ 等包含属性 A, B, C 的基于集合的规范次序依赖均不成立。因此逐点次序依赖严格包含基于集合的规范次序依赖。

2.3　否定约束

否定约束定义 一个否定约束 $\varphi : \neg(P_1 \wedge P_2 \wedge \cdots \wedge P_m)$ 是通过连接多个谓词的逻辑表达式定义的数据约束。一个谓词为形如 $P = (t_\alpha[A]\ op\ t_\beta[B])$ 或者 $P = (t_\alpha[A]\ op\ c)$，其中 t_α, t_β 为数据集中的元组，可以为同一元组。A, B 为数据集中的属性，可以为同一属性。c 为常数。op 是运算符，可以选取 6 种不同的运算符，$op \in \{<, >, \geq, \leq, =, \neq\}$ [6]。如果一个否定约束中的谓词全为 $P = (t_\alpha[A]\ op\ t_\beta[B])$ 形式，则称为 VDC，否则称为 CDC。

否定约束指数据集中元组或元组对不能使在约束中的所有谓词同时为真，否则称该元组或元组对为否定约束的矛盾。特别的，函数依赖是谓词运算符只包含等于和不等于的否定约束。

示例 2-6: 以表格1.1中数据集 D_1 为例，存在以下否定约束：

$c_1 : \forall t_\alpha \in D_1, \neg(t_\alpha[ZIP] = \text{"80612"} \wedge t_\alpha[ST] \neq \text{"}CO\text{"})$

以表格2.1中数据集 r 为例，存在以下否定约束：

$c_2 : \forall t_\alpha, t_\beta \in r, \neg(t_\alpha[ManId] = t_\beta[ManId] \wedge t_\alpha[City] \neq t_\alpha[City])$　□

逐点次序依赖到否定约束的转换 因为逐点次序依赖是不同元组的属性值上的次序约束，否定约束也可以表示属性值间的次序关系。所以逐点次序依赖可以转换为 $\varphi = \neg(P_1 \wedge P_2 \wedge \cdots \wedge P_m)$，其中每一个谓词 P 只包含一个属性。

示例 2-7: 如图2.1所示，否定约束是包含多种不同的约束。下面展示之前例子中提到的约束转为否定约束形式：

对于示例1-1中的约束可以转换为相应的否定约束：

$\varphi_1 : \neg(t_\alpha[ZIP] = t_\beta[ZIP] \wedge t_\alpha[ST] \neq t_\beta[ST])$

$\varphi_3 : \neg(t_\alpha[TXA] > t_\beta[TXA] \wedge t_\alpha[SAL] < t_\beta[SAL] \wedge t_\alpha[RATE] \geq t_\beta[RATE])$

$\varphi_4 : \neg(t_\alpha[SSN] = t_\beta[SSN] \wedge t_\alpha[Date] < t_\beta[Date] \wedge t_\alpha[NUM] \geq t_\beta[NUM])$

因为约束 σ_2 的 RHS 有两个属性，所以需要转换成两个否定约束：

$\varphi_2 : \neg(t_\alpha[SSN] = t_\beta[SSN] \wedge t_\alpha[FName] \neq t_\beta[FName])$

$\varphi_2' : \neg(t_\alpha[SSN] = t_\beta[SSN] \wedge t_\alpha[LName] \neq t_\beta[LName])$　□

否定约束严格包含逐点次序依赖 否定约束具有非常强表达显示约束能力。即使限制了谓词不能进行跨属性比较。因为否定约束中谓词的运算符有 6 种，逐点次序依赖中带运算符属性只有 5 种不同的运算符。

限定条件的否定约束与逐点次序依赖的等价性 通过上述转换说明可知，任何一个逐点次序依赖 $\{\mathsf{A}_1^{\mathsf{op}_1}, \mathsf{A}_2^{\mathsf{op}_2} \cdots \mathsf{A}_\mathsf{n}^{\mathsf{op}_\mathsf{n}} \hookrightarrow \mathsf{B}^{\mathsf{op}'}\}$ 等价于否定约束 $\neg(t_\alpha[A_1]\ op_1\ t_\beta[A_1] \wedge t_\alpha[A_2]\ op_2\ t_\beta[A_2]\ \cdots \wedge t_\alpha[A_n]\ op_n\ t_\beta[A_n] \wedge t_\alpha[B]\ \overline{op'}\ t_\beta[B])$。其中 $\overline{op'}$ 为运算符 op' 的反 (对应关系见第三章表格3.1)。因为逐点次序依赖运算符只包含 $\{<, >, \geq, \leq, =\}$ 五种，所以逐点次序依赖等价于限制了谓词不能进行跨属性比较并且至多包含一个谓词运算符为不等于的否定约束。

第 3 节 约束的发现算法

键值约束 在数据集中发现所有唯一 (和非唯一) 列组合是任何数据分析工作的核心。唯一的列组合类似于关系数据集的候选键，数据集上的两个不同元组在唯一列组合中至少有一个属性值不同。非唯一列组合是指存在数据集上的两个不同元组在非唯一列组合中属性值相同。查找唯一 (非唯一) 约束的全量算法算法：基于列算法 $GORDIAN$ [21]，$DUCC$ [22],HCA [23]。然而，这些方法都不适用于动态数据集上的应用，例如事务数据库、社交网络和科学应用。在这些情况下，数据分析技术应该能够在元组插入或删除之后有效地发现新的唯一和非唯一 (并验证旧的唯一) 列组合，而无需重新分析整个数据集。$SWAN$ 是一种有效发现动态数据集上唯一和非唯一约束的方法 [10]。

函数依赖 如果函数依赖 $\varphi : X \to A$ 是成立的函数依赖，并且去掉 X 中的任意属性后均为不成立的函数依赖，则称 φ 是最小的函数依赖。如果 A 是 X 的一个子集，则称为平凡的函数依赖。目前发现最小非平凡的函数依赖全量方法主要有：基于模式驱动的算法 $TANE$ 的 [24]，基于实例驱动的算法 $FASTFD$ [25]。$TANE$ 的算法复杂度与关系模式的大小，即属性个数密切相关；$FASTFD$ 的算法复杂度与数据集大小关系更密切。基于模式驱动的函数依赖发现算法还有：$FDMine$ [26],FUN [27],DFD [28]。基于实例驱动的算法还有：$DepMiner$ [28]。算法 $HyFD$ 是将两种基于不同驱动的算法结合使用 [30]。同样的，这些算法也无法直接使用在需要进行元组插入或者删除的数据集上。$Dynfd$ [11] 为一种发现函数依赖的增量算法。

字典序次序依赖与基于集合的规范次序依赖 字典序次序依赖与基于集合的规范次序依赖的全量发现算法研究主要在 [15, 16, 19, 20]。其中 [19, 20] 主要研究字典序次序依赖的全量发现算法，[15, 16] 主要研究基于集合的规范次序依赖的全量发现算法。算法 $FastOD$ 是发现完全成立和近似成立的字典序次序依赖的算法 [31]。这些算法也没有考虑在动态数据集上的情况，目前增量字典序次序依赖的增量算法研究较少，主要讨论的也是数据进行插入时的情况 [32]。

否定约束 全量 DC 发现算法已经有研究成果 [18,33,34]。算法 $FASTDC$ 是将算法 $FASTFD$ 扩展到否定约束的算法 [18]。$HYDRA$ 是先在抽样数据集上算出成立的否定约束集合 Ψ，然后再通过查找整个数据集上与 Ψ 中约束矛盾的元组对集合与抽样数据集组成 $Evidence\ Set$。最后在整个 $Evidence\ Set$ 上扩展 Ψ。算法 $DCFinder$ 主要是将 $FASTDC$ 的算法扩展为查找近似的否定约束 [34]。本文主要是发现成立的 PODs 。在发现完全成立的 DCs 时，其实验结果表明，$HYDRA$ 运行时间比 $FASTDC$ 会快一到两个数量级。本文在 $HYDRA$ 的基础上修改得到全量 POD 的算法Hydra$*$ ，作为增量 POD 发现算法IncPOD(第五章) 的对比算法。

逐点次序依赖 目前没有文章对 POD 的全量发现算法进行研究。因为 PODs 是 LODs 和 SODs 的严格泛化，所以相关方法都不能直接应用到 PODs 的全量发现。在第三章中可以看到，PODs 的全量发现复杂度为属性个数的指数倍，和基于集合的规范次序依赖 SODs 的全量发现算法复杂度一致 [15, 16]。相对而言，LODs 全量发现的复杂度更为麻烦，是属性个数的排列数的指数倍 [19, 20]。

因为实际生活中的需要数据集进行变动，在动态数据集上发现约束受到越来越多的关注。目前已有用于唯一列组合 UCC，函数依赖 FD，以及近年的 LOD 的增量发现算法。给定数据集 D 和数据集上发现的约束集合 Σ，以及更新集合 $\triangle D$，增量发现问题主要有两个子问题组成。第一个子问题是快速地识别出由 $\triangle D$ 引起的 Σ 的矛盾。第二个子问题是将约束集合 Σ 演化为 $\Sigma \oplus \triangle\Sigma$，即在更新后的数据集上成立的有效约束。当然，这两个子问题的解决方案因约束类型的不同而有显著差异。增量的 POD 发现是更复杂的，因为 PODs 包含 FDs (UCCs) 并且是 LODs 的泛化。正如第一章以及第四章中讨论的例子，检测增量数据集引起的一个 POD 的矛盾就是一个值得深入研究的问题。本文引入了一种新的索引结构收集矛盾。此外，计算 $\triangle\Sigma$ 时不仅仅需要向属性集 LHS 增加更多的属性，还会优化 LHS 和 RHS 属性上的运算符。

第 4 节　不等式连接

在关系数据库中，一般的查询都是进行相等的条件筛选或者是进行单属性上的排序。在两个属性上进行不相等的连接查询的技术研究比较少。不等式连接是指值之间的连接关系并非等于条件，多用于实际数据查找中。在数据库系统中有非常多连接优化索引，包含 $B/B+$ 树、位图等多种索引。但是对不等式连接的优化很少，所以包含不等式连接的查询通常非常慢。优化此类查询的常用方法包含排序合并连接 [35] 和基于区间的索引 [36–38]。排序合并连接是基于连接属性对数据进行排序并将其合并来减少搜索空间。然而对于不等式连接条件的查询仍然是平方复杂度。基于区间的索引通过位图区间索引降低查询搜索空间。然而位图索引需要较大的内存空间和较长的索引构建时间。

不等式连接技术IEJoin 在上面的方法基础进行扩展，它利用了每个属性上的排序数组和一个位数组来编码两个属性的连接 [41]。

表 2.4: 时间花费表 *west*

TID	t_id	time	cost	cores
t_0	404	100	6	4
t_1	498	140	11	2
t_2	676	80	10	1
t_3	742	90	5	4

示例 2-8: 在数据集2.4上进行如下的 SQL 查询语句：

Q_1 ： SELECT $s_1.t\_id,\ s_2.t\_id$
FROM $west\ s_1,\ west\ s_2$
WHERE $s_1.time\ >\ s_2.time$

这样的 SQL 查询语句返回的是整个数据集 $west$ 上两个不同元组 s_1, s_2 在属性 t_id 上的值，其中元组 s_1, s_2 满足 s_1 的时间比 s_2 的大，这样的查询语句最普通的解决方法是直接进行 $O(n^2)$ 的元组比较。也可以先按照属性 $time$ 进行排序，那么后面的元组在属性 $time$ 上的值就是大于或者等于前面元组在属性 $time$ 上的值。如果在排序的时候记住不同取值的个数或者使用等价类等方法可以将查询时间降至 $O(n * log(n))$。按照属性 $time$ 上的值进行排序后的结果为 $t_2(80), t_3(90), t_0(100), t_1(140)$，因此 SQL 查询语句 Q_1 的返回结果为 $(t_2, t_3), (t_2, t_0), (t_2, t_1), (t_3, t_0), (t_3, t_1), (t_0, t_1)$。这样的查询是基于单个谓词的查询，可以看到这样的查询的结果集是依赖于数据集在属性 $time$ 上不同值的多少。一般情况下，运算符 $>, \geq$ 或者 $<, \leq$ 基本上选择性比较低。这与第四章第一节描述一致。考虑两个谓词组成的查询，如 SQL 查询语句 Q_2：

Q_2 ： SELECT $s_1.t\_id,\ s_2.t\_id$
FROM $west\ s_1,\ west\ s_2$
WHERE $s_1.time\ >\ s_2.time$ AND $s_1.cost\ <\ s_2.cost$

这样的 SQL 查询语句返回的是整个数据集 $west$ 上两个不同元组 s_1, s_2 在属性 t_id 上的值，其中元组元组 s_1, s_2 满足 s_1 的时间比 s_2 的大，并且 s_1 的花费比 s_2 小。这样的查询语句普通的解决方法是进行 $O(n^2)$ 的比较。

IEJoin 做法是先将数据集按照属性 $time$，$cost$ 上进行排序有：

按照属性 $time$ 的排序结果：$L_1 = [t_2(80), t_3(90), t_0(100), t_1(140)]$。

按照属性 $cost$ 的排序结果：$L_2 = [t_3(5), t_0(6), t_2(10), t_1(11)]$。

在排序的同时可以算出 L_2 与 L_1 的置换数组 $P = [2, 3, 1, 4]$。L_2 的第一个为 t_3，同时 t_3 在 L_1 中为第二个数，所以置换数组 P 第一个数字为 2。初始化一个空白数组 $B = [0, 0, 0, 0]$，表示按照顺序 L_1 排序的元组是否访问。依次按照 P 数组的值将 B 中对应位置修改为 1，表示已经访问该元组。第一轮有 $B = [0, 1, 0, 0]$，第二轮有 $B = [0, 1, 1, 0]$。在每一轮更新 B 的同时会向后扫描，如果在对应位置后面的值为 1 时，说明两个元组满足查询条件。第三轮有 $B = [1, 1, 1, 0]$，可以得到矛盾元组对 $(t_3, t_2), (t_0, t_2)$。最后一轮有 $B = [1, 1, 1, 1]$。最后结果为 $(t_3, t_2), (t_0, t_2)$。 □

IEJoin 的做法可以应用到动态数据集中，可是仍然需要将增量数据集与原始数据集按照不同属性一起排序。这样的做法不利于动态数据上查找和更新矛盾元组结果集。第六章中的实验结果表明，本文构建的索引在动态数据集上进行查找矛盾的时间比IEJoin 的运行时间快，并且原始数据集和增量数据集的大小对本文索引查询时间的影响明显低于IEJoin 。

第 5 节　本章小结

本章主要介绍了不同约束间的层次关系、有关的次序依赖以及否定约束、不等式连接。本章第一节首先通过文字和层次关系图说明不同约束间的泛化关系，通过实例来展示不同约束的表达形式以及实际含义。然后在第二节介绍了字典序次序依赖、基于集合的规范次序依赖以及否定约束的定义、通过实例说明了其表达形式与含义。并且介绍了约束间的转换方法以及通过实例说明了逐点次序依赖严格泛化字典序次序依赖，文字说明了否定约束严格泛化逐点次序依赖。第三节说明不同约束已有的全量或者增量发现算法研究成果以及说明逐点次序依赖增量发现算法的两个子问题。第四节说明逐点次序依赖增量发现算法第一个子问题不等式连接的相关研究，使用实例说明算法IEJoin 的流程。

第三章　PODs 定义及全量、增量算法说明

本章首先给出关于 PODs 的基本符号；然后给出了最小 PODs 的定义以及说明全量 POD 发现算法的基本流程和时间复杂度；最后给出增量 POD 发现的问题描述以及相关符号说明。

第 1 节　定义与符号

基础记号 $R(A,B,\dots)$ 表示有着属性 A, B, $\dots$ 的关系模式。使用符号 D 表示一个特定的实例或关系表。符号 t, s 表示数据集中的元组，$t[A]$ 表示元组 t 在属性 A 上的取值。符号 X 表示若干属性组成的属性集，$t[X]$ 表示元组 t 在属性集 X 上的取值。每一个元组 t 都有着唯一标识 (id)，记为 t_{id}。

带运算符属性 (marked attribute) 对于 R 中的一个属性 A，属性 A 上的带运算符属性格式为 A^{op}，其中运算符 $op \in \{<, \leq, >, \geq, =\}$。对两个元组 t 和 s，如果满足 $t[A]$ op $s[A]$，则写为$A^{op}(t, s)$ 。符号 $\mathcal{X}$ 表示带运算符属性的集合，并且对应的 X 表示带运算符属性的集合 $\mathcal{X}$ 中的属性集合。

示例 3-1: 在表格1.1中，因为元组 t_1 在属性SAL 上的值为 50000，元组 t_0 在属性上的值为 32000，所以 $t_1[SAL] > t_0[SAL]$。元组对 (t_1, t_0) 满足 $SAL^{>}$，写为$\mathsf{SAL}^{>}(t_1, t_0)$ 。同样的，因为元组 t_2 和元组 t_5 在属性 ST 上均为 ND。所以有 $t_2[ST] = t_5[ST]$。元组对 (t_2, t_5) 满足 $ST^{=}$，写为$\mathsf{ST}^{=}(t_2, t_5)$ 。 □

逐点次序依赖 (PODs) 一个 POD σ 是形如 $\mathcal{X} \hookrightarrow \mathcal{Y}$ ，其中$\mathcal{X}$ 和 $\mathcal{Y}$ 是两个带运算符属性的集合。带运算符集合 $\mathcal{X}$ 为 POD σ 的左侧属性集合，记为 LHS ；带运算符集合 $\mathcal{Y}$ 为 POD σ 的右侧属性集合，记为 RHS 。

逐点次序依赖 POD σ ： $\mathcal{X} \hookrightarrow \mathcal{Y}$ 表示含义为：当元组对 (t, s) 满足所有 $A^{op}(t, s)$ 时，那么元组对 (t, s) 满足所有的 $B^{op}(t, s)$，其中 $A^{op} \in \mathcal{X}$ ，$B^{op} \in \mathcal{Y}$ 。当 σ 在数据集 D 上成立时，称 σ 是数据集 D 上的有效 POD 。

POD 的 RHS 多属性说明 一个 POD σ：$\mathcal{X} \hookrightarrow \mathcal{Y}$ 可根据 RHS 属性集 $\mathcal{Y}$ 等价拆分成多个 RHS 为单个带运算符的属性的 PODs ： $\mathcal{X} \hookrightarrow B^{op}$，其中 $B^{op} \in \mathcal{Y}$ 。

示例 3-2: POD σ_2：$\{\mathsf{SSN}^{=}\} \hookrightarrow \{\mathsf{FName}^{=}, \mathsf{LName}^{=}\}$ 可以等价分解成两个 POD ：

POD σ_2'：$\{\mathsf{SSN}^{=}\} \hookrightarrow \{\mathsf{FName}^{=}\}$

POD σ_2''：$\{\mathsf{SSN}^{=}\} \hookrightarrow \{\mathsf{LName}^{=}\}$

这样等价拆分 POD 的 RHS 的方法成立，为了说明简便本文之后讨论 PODs 的 RHS 为单个带运算符属性。 □

平凡的 PODs 如果 POD σ 在所有实例上面均成立，则称 σ 为是平凡的 POD 。比如带运算符属性集合$\mathcal{X}$ 是包含于$\mathcal{Y}$ 等情况。本文只考虑非平凡的 PODs $\mathcal{X} \hookrightarrow \mathcal{Y}$ 。因为后面只讨论 RHS 为单个带运算符属性，那么非平凡 PODs 即为每一个属性 A (不考虑符号) 最多只在$\mathcal{X} \cup \mathcal{Y}$ 出现一次。

示例 3-3: 可以验证在例1-1，例1-2中出现的 PODs 都是数据集 D_1 上非平凡的有效的 PODs 。而且 $\sigma_1, \sigma_2, \sigma_4$ 仍是数据集 $D_1 + \triangle D_1$ 上的有效 PODs 。

比如对数据集 $D_1 + \triangle D_1$ 任意的两个元组可以验证：如果$\mathsf{SSN}^{=}(s,t)$ 和 $\mathsf{Date}^{<}(s,t)$ 成立，那么$\mathsf{NUM}^{<}(s,t)$ 成立。即在数据集 $D_1 + \triangle D_1$ 上约束 $\sigma_4 = \{\mathsf{SSN}^{=}, \mathsf{Date}^{<}\} \hookrightarrow \{\mathsf{NUM}^{<}\}$ 是有效的 POD 。 □

等价 POD 定义 因为 POD 可以转为对应的否定约束，如果通过转化方法两个 POD 可以转为相同的否定约束，则称这两个 POD 等价。

示例 3-4: 例1-1中的 POD σ_3: $\{\mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{<}\}$ 可转换为否定约束 φ: $\forall t_1, t_2 \in D_1, \neg(t_1[TXA] > t_2[TXA] \wedge t_1[SAL] < t_2[SAL] \wedge t_1[RATE] \geq t_2[RATE])$。与 POD σ_3 等价的 POD 有：

σ':$\{\mathsf{TXA}^{>}, \mathsf{RATE}^{\geq}\} \hookrightarrow \{\mathsf{SAL}^{\geq}\}$

σ'':$\{\mathsf{SAL}^{<}, \mathsf{RATE}^{\geq}\} \hookrightarrow \{\mathsf{SAL}^{\leq}\}$

为了避免重复发现等价的 PODs 以及减少 POD 冗余的判断。在本文算法中，会先按照实例中属性顺序给属性编号。发现和扩展 PODs 时选取 RHS 为 POD 中序号最小的属性，这样确保每一个 POD 有唯一的表达。因此可以通过判断 $\mathcal{X}, \mathcal{X}'$ 以及 $\mathcal{Y}, \mathcal{Y}'$ 是否相等来判断 POD $\sigma: \mathcal{X} \hookrightarrow \mathcal{Y}$, σ':$\mathcal{X}' \hookrightarrow \mathcal{Y}'$ 是否等价。 □

表 3.1: 运算符的反转，蕴含，对称

op	$=$	$<$	$>$	$\leq$	$\geq$
$\overline{op}$	$<, >$	$\geq$	$\leq$	$>$	$<$
$im(op)$	$=, \leq, \geq$	$<, \leq$	$>, \geq$	$\leq$	$\geq$
$sym(op)$	$=$	$>$	$<$	$\geq$	$\leq$

运算符的反转，对称与蕴含关系 $\overline{op}$,$sym(op)$, $im(op)$ 分别为一个运算符 op 的反转，对称，蕴含运算符 (表格3.1)。可以看到：(1) 对于任意的两个元组 t, s 要么 $A^{op}(t,s)$ 成立，或者 $A^{\overline{op}}(t,s)$ 成立。(2) 如果两个元组 t, s 满足 $A^{op}(t,s)$，那么 $A^{sym(op)}(s,t)$。(3) 如果两个元组 t, s 满足 $A^{op}(t,s)$，那么 $A^{im(op)}(t,s)$。

示例 3-5: 运算符 $>$ 与 $\leq$ 互为反转。即在数据集 D 上，任意两个不同元组 t, s 要么满足 $\mathsf{SAL}^{>}(t,s)$，要么满足 $\mathsf{SAL}^{\leq}(t,s)$，不会同时满足，也不会同时不满足。

运算符 $>$ 与 $<$ 互为对称。即在数据集 D 上，任意两个不同元组 t, s 如果满足 $\mathsf{SAL}^{>}(t,s)$，那么一定满足 $\mathsf{SAL}^{<}(s,t)$。

运算符 $>$ 是蕴含了 $\geq$。即在数据集 D 上，任意两个不同元组 t, s 如果满足 $\mathsf{SAL}^{>}(t,s)$，那么一定满足 $\mathsf{SAL}^{\geq}(t,s)$。 □

带运算符属性的包含关系 两个带运算符属性的集合 $\mathcal{X}$ 和 $\mathcal{X}'$，$\forall A^{op'} \in \mathcal{X}'$，如果有 $A^{op} \in \mathcal{X}$，那么 $op' \in im(op)$，则称$\mathcal{X}$ 包含 $\mathcal{X}'$。写作 $\mathcal{X}' \subseteq \mathcal{X}$。

示例 3-6: $\{\mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \subseteq \{\mathsf{ST}^{=}, \mathsf{TXA}^{>}, \mathsf{SAL}^{<}\}$ 以及 $\{\mathsf{RATE}^{\leq}\} \subseteq \{\mathsf{RATE}^{<}\}$。直观来看，$\mathcal{X}' \subseteq \mathcal{X}$ 表示$\mathcal{X}$ 比 $\mathcal{X}'$ 的条件更“严格”，即 POD:$\mathcal{X} \hookrightarrow \mathcal{X}'$ 在所有的数据集 D 上是有效的 POD，即为一个平凡的 POD。 □

POD 的逻辑蕴含 在任意的数据集上 D 如果满足 POD σ' 那么一定满足 POD σ，则称 σ 被 σ' 逻辑蕴含。

最小 PODs 当一个 POD σ 没有被其他有效的 POD σ' 逻辑蕴含，则称 σ 是最小的 POD。下面基于带运算符集合的包含关系，给出最小 PODs 的规范化定义。如果不存在成立的 POD $\sigma' = \mathcal{X}' \hookrightarrow \mathcal{Y}'$，其中 $\mathcal{X}' \subseteq \mathcal{X}$ 以及 $\mathcal{Y} \subseteq \mathcal{Y}'$，那么称 POD $\sigma = \mathcal{X} \hookrightarrow \mathcal{Y}$ 是最小的 POD。

示例 3-7: 在任何满足 $\{\mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{<}\}$ 的数据集上，$\{\mathsf{ST}^{=}, \mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{<}\}$ 和 $\{\mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{\leq}\}$ 都是有效的 PODs。如果 $\{\mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{<}\}$ 是有效的 POD，那么 $\{\mathsf{ST}^{=}, \mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{<}\}$ 和 $\{\mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{\leq}\}$ 都不是最小的 POD。 □

矛盾或者违背元组对 元组对 (t, s) 满足了 σ 中 LHS 条件，但是不满足 σ 中 RHS 条件时称元组对 (t, s) 为约束 POD σ 的矛盾或者违背元组对。

示例 3-8: 以数据集1.1中的元组对 t_1, t_7 和示例1-1中的 σ_3：$\{\mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{<}\}$ 为例。元组 t_1 在属性TXA 上的值 3000 小于元组 t_7 在属性TXA 上的值 3300，即 (t_7, t_1) 满足$\mathsf{TXA}^{>}$。元组 t_1 在属性SAL 上的值 50000 大于元组 t_7 在属性SAL 上的值 40000，即 (t_7, t_1) 满足$\mathsf{SAL}^{<}$。元组对 (t_7, t_1) 满足 σ_3 的 LHS 的条件。然而元组 t_1 与元组 t_7 在属性Rate 上的值相等为 1.42，即元组对 (t_7, t_1) 不满足$\mathsf{RATE}^{<}$。即元组对 (t_7, t_1) 不满足 σ_3 的 RHS 的条件。因此元组对 (t_1, t_7) 为 σ_3 的违背元组对。 □

其他说明 可以显然地看出以下结论：

(1) 函数依赖 (FDs) 是一种特殊形式的 PODs。一个函数依赖 FD $\mathcal{X} \rightarrow \mathcal{Y}$ 是只包含运算符为 ” $=$ ” 的 POD。

(2) POD $\mathcal{X} \hookrightarrow \mathcal{Y}$ 也可以写为一系列的 PODs $\mathcal{X} \hookrightarrow B_i^{op_i}$，其中 $B_i^{op_i} \in \mathcal{Y}$。显然，$\mathcal{X} \hookrightarrow \mathcal{Y}$ 是有效的当且仅当每一个 POD$\mathcal{X} \hookrightarrow B_i^{op_i}$ 都是有效的。即上述的 POD 的 RHS 可以拆分的说明。

(3) 每一个 POD σ 有一个对称的 POD σ_{sym}，是通过将每一个带运算符属性的符号反转得到，比如将 “$>$” 反转为 “$<$”。比如 $\{\mathsf{A}_1^{=}, \mathsf{A}_2^{>}, \mathsf{A}_3^{\leq}\} \hookrightarrow \{\mathsf{B}^{>}\}$ 和 $\{\mathsf{A}_1^{=}, \mathsf{A}_2^{<}, \mathsf{A}_3^{\geq}\} \hookrightarrow \{\mathsf{B}^{<}\}$ 互为对称 POD。可以看到，σ 是有效的 POD 当且仅当 σ_{sym} 是有效的。为了避免冗余的 PODs，只考虑右侧属性 RHS 的符号为 $<, \leq, =$ 的 PODs。例如 $\{\mathsf{A}_1^{=}, \mathsf{A}_2^{\leq}, \mathsf{A}_3^{\geq}\} \hookrightarrow \{\mathsf{B}^{<}\}$。显然，这样的改动并不会丢失信息。

第 2 节　逐点次序依赖的全量发现算法

全量发现复杂度 在一个数据集 D 上有效的 PODs 数目是非常大且有相互蕴含关系。所以比起发现所有的 PODs，发现最小的 PODs 更为必要。给定一个关系模式为 R 的数据集 D，全量的 POD 发现算法是指发现一个 POD 的完备集合 Σ。完备集合 Σ 是所有数据集 D 上最小有效非平凡的 POD 。POD 发现复杂度是指 POD 的整个可能的搜索空间大小。

命题 1: POD 发现复杂度是 $O(3 \times |R| \times 6^{(|R|-1)})$，其中 $|R|$ 是属性个数。　□

证明: 对任意的一个 POD, (1) 在右侧的带符号属性 RHS ，一个属性 $A \in R$ 是有 3 个不同符号的选择 (因为运算符的对称性，本文发现 POD 的 RHS 均为 "$<, \leq, =$")，(2) 在左侧的带符号属性集合 LHS ，每一个剩余的属性 $R \setminus A$ 要么 (a) 不出现在该 POD 中，或者 (b) 出现在该 POD 中。如果出现在 POD 则有 5 种不同的运算符 "$>, \geq, <, \leq, =$" 选择。　□

限定条件 DC 与 POD 的等价性 可以看到一个 POD $\{{A_1}^{op_1}, \cdots, {A_m}^{op_m}\} \hookrightarrow B^{op'}$ 可以转换为 DC $\neg(t_\alpha[A_1]\ op_1\ t_\beta[A_1] \wedge \cdots \wedge t_\alpha[A_m]\ op_m\ t_\beta[A_m] \wedge t_\alpha[B]\ \overline{op'}\ t_\beta[B])$。根据违背元组对的定义可知如果一个元组对违背了 POD ，那么也一定违背了转换后的 DC ，反之亦然。转换后的 DC 没有跨属性比较；每一个属性只出现一次；至多有一个属性上符号为 $\neq$。同样的，对一个有这样限定条件的 DC ：(1) 若存在一个属性运算符为 $\neq$，则将这个属性作为 POD 的 RHS 。(2) 若不存在一个属性运算符为 $\neq$，则选取属性序号最小的属性作为 POD 的 RHS 。通过转换，限定条件的 DC 可以一对一转换为 POD 。

其余说明 和约束 LOD ，SOD 和 DC 的发现复杂度比较说明：

(1) POD 的发现复杂度是属性个数 $|R|$ 的指数倍，比 LOD 发现的属性阶乘复杂度 $O(|R|!)$ 要小。相较于 POD 的发现复杂度而言，即使忽略 LHS 和 RHS 有相同的属性，LOD 发现有更大的发现复杂度。注意到，在忽略 LHS 和 RHS 有相同的属性的设定下 LODs 不再泛化 FDs [9]。

(2) 基于集合的约束 SODs 是泛化的 LODs 。SODs 的发现复杂度也是属性个数 $|R|$ 的指数倍，这是因为一些不同的 LODs 可能映射为相同的 SODs 。

(3) PODs 是 SODs 的进一步泛化，发现复杂度是属性个数 $|R|$ 的指数倍 [15]。PODs 的泛化是很有必要。在实际中，有部分次序依赖不能被 LODs 和 SODs 表示。比如在数据集2.3上，$\{\mathsf{A}^{>}, \mathsf{B}^{>}\} \hookrightarrow \{\mathsf{C}^{>}\}$ 成立, 但是数据集上这样的 POD 不能被 LODs 或者 SODs 等价的表示。

(4) 对任意一个 DC ，一个谓词要么出现在 DC 中，要么不出现在 DC 中。因此 DC 的发现复杂度为 $2^{|P|}$，其中 $|P|$ 为谓词空间的大小 [18]。本文增量发现算法的对比全量发现算法Hydra* 流程为：通过更改后的 $Hydra$ 发现完备限定条件的 DC 集；再等价转换为 POD 集合。

第 3 节 逐点次序依赖的增量发现

增量发现 对于给定的数据集 D 、模式 R、在数据集 D 上的有效最小 PODs 集合 Σ 以及一个增量数据集 $\triangle D$。增量 POD 的发现是指发现一个约束变化集合 $\triangle\Sigma$ 更新到已有约束集合 Σ 使得 $\Sigma \oplus \triangle\Sigma$ 是数据集 $D + \triangle D$ 的 PODs 。特别的，$\triangle\Sigma = \triangle\Sigma^+ \cup \triangle\Sigma^-$，其中 (1) $\triangle\Sigma^+ \cap \Sigma = \emptyset$: $\triangle\Sigma^+$ 包含了数据集 $D + \triangle D$ 中新的最小有效 PODs $D + \triangle D$; (2) $\triangle\Sigma^- \subseteq \Sigma$: $\triangle\Sigma^-$ 包含了 Σ 在数据集 $D + \triangle D$ 中不成立的 PODs 。需要从 Σ 中移除不成立的 PODs 。也就是说，$\Sigma \oplus \triangle\Sigma$ 是等同于 $(\Sigma \cup \triangle\Sigma^+) \setminus \triangle\Sigma^-$。

增量发现的主要难点在于计算出正确的 $\triangle\Sigma$。一般解决方法为先发现约束和数据集的矛盾集合；计算出不成立的 POD 集合 $\triangle\Sigma^-$；在违背元组对集合和 $\triangle\Sigma^-$ 的基础上扩展出新的成立 POD 集合 $\triangle\Sigma^+$。这样的解决方法主要由两个子问题组成。第一个问题是如何快速地得到 $\triangle D$ 引起的数据集与约束集 Σ 的矛盾集合，即收集违背元组对。第二个问题是怎样正确地扩展出新成立的 POD ，即在更新后的数据集上成立的有效约束。

收集违背元组对 检测增量数据集引起的 POD 的矛盾本身就是一个值得深入研究的问题。检测矛盾等同于动态数据集上不等式连接查询问题。本文介绍了一种建立在两个带运算符属性$\mathsf{A}^{\circ p_1}$ 和$\mathsf{B}^{\circ p_2}$ 上的索引结构$\mathsf{Index}(\mathsf{A}^{\circ p_1}, \mathsf{B}^{\circ p_2})$ 。可以利用这样的索引快速地发现和收集违背元组对集合 T。违背元组对集合 T 是类似于示例2-8中两个不等谓词查询 Q_2 的结果集。直觉上而言，三个或者更多的不等谓词查询的结果集会更小。而且矛盾集合越小则后面的修复时间越少。但是用三个或者更多的不等连接查询去覆盖原始约束集合 Σ 会导致索引变多。索引变多就会使得存储空间变大。在第五章中会详细说明如何选择索引覆盖 Σ。

扩展 POD 正确计算约束变化集合 $\triangle\Sigma$ 也是一个复杂问题。扩展不成立的键约束是直接添加属性；扩展不成立的函数依赖是直接添加属性到 FD 的 LHS ；扩展不成立的 DC 是直接添加谓词。这样的处理方法是来源于 UCC ，FD 以及 DC 的最小性定义。因为 POD 的最小性不只在属性上定义，还和属性的运算符有关。如第一章的示例1-2所示，扩展不成立 POD 时不仅仅要向属性集 LHS 增加更多的带运算符属性，还会优化 LHS 和 RHS 属性上的运算符。第五章会详细说明 POD 的扩展流程。

其他说明 增量数据集引发的矛盾不仅有增量数据集 $\triangle D$ 和原始数据集 D 之间的矛盾，还有增量数据集 $\triangle D$ 自身的矛盾。本文给出的索引更新机制确保能在索引收集矛盾的流程中一并收集增量数据集自身的矛盾。在其他约束修复算法中，也有相应的索引更新流程。比如 FD 修复中会更新等价类等等。本文的索引更新可以和发现违背元组对一起进行，在第五章中会详细描述本文索引的表示方法以及本文索引的更新流程。

第 4 节　本章小结

本章首先给出了约束相关的符号以及定义，包含平凡 POD 、最小 POD 、矛盾元组对等。通过大量的例子辅助说明符号的实际意义。随后说明了全量发现算法的复杂度、限定条件 DC 与 POD 的等价关系、对比全量算法Hydra∗ 的正确性。最后给出增量发现问题的定义和本文对两个子问题的解决方法。

第四章　索引的建立与使用

本文提出了一种基于两个带符号属性 A^{op_1}, B^{op_2} 的索引$Index(A^{op_1}, B^{op_2})$ 。基于该索引的结构可以快速地找到增量数据引发的矛盾。本章介绍了索引建立的目的、索引的结构、给出了如何建立最优索引的算法、算法建立最优索引的证明以及描述了利用索引查找矛盾的方法。

第 1 节　使用索引目的说明

原始数据集中没有有效 POD 约束的违背元组对。所以在动态数据集上面修复数据约束只需要发现原始数据和增量数据的矛盾、增量数据自身间的矛盾。相关的数据约束修复研究往往会选择建立索引的方法来保留一些原始数据的性质。这样就可以通过索引快速地发现原始数据和增量数据之间的矛盾。比如在增量的函数依赖修复中，会根据函数依赖左边的属性建立等价类或者划分。插入增量数据时可以快速地通过等价类找到相关的元组。但是以往的索引目的大多是为了查找值相等的相关元组，不适用查找 POD 矛盾。

一个 POD 包含了不等的运算符，所以需要建立索引来查找属性值上满足相关次序的元组对。不等运算符的引入，即“$<$，$\leq$，$>$，$\geq$”，使元组插入上的违反检测明显比查询相等的情况复杂，如下示例所示：

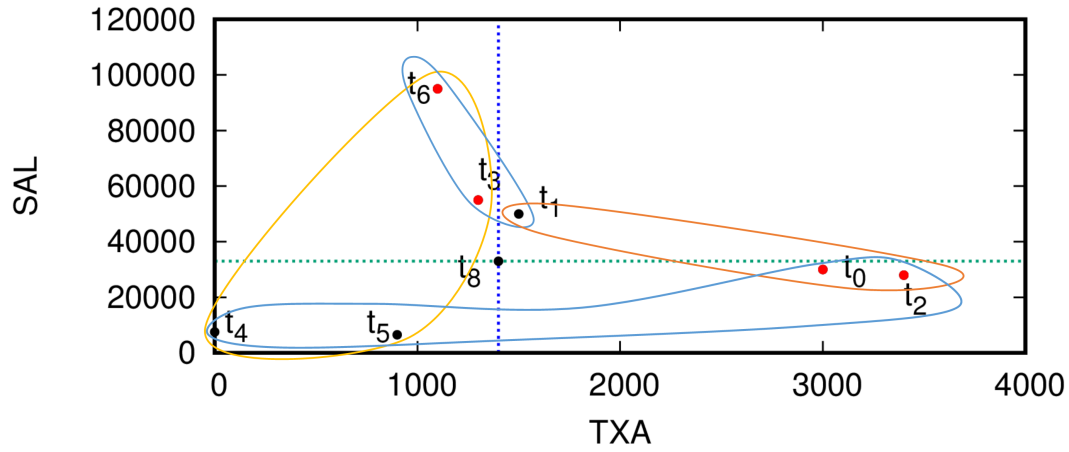


图 4.1: 与元组 t_8 可能违背的元组

示例 4-1: 以表格1.1中数据集 D_1 和表格1.2数据集 $\triangle D_1$ 中元组 t_8 为例。发现数据集 D_1 上有效的约束 $\sigma_3 = \{\mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{<}\}$ 的矛盾。查找与元组 t_8 可能构成矛盾的元组流程如下：

(1) 按照数据集 D_1 中元组的 TXA, SAL 属性值与元组 t_8 的 TXA, SAL 属性值的大小关系计算出 4 个不同集合 (图4.1中使用不同颜色表示)：

属性TXA 上的值比 t_8 小的：$\{t|\mathsf{TXA}^{<}(t, t_8)\ \} = \{t_3, t_4, t_5, t_6\}$，

属性TXA 上的值比 t_8 大的：$\{t|\mathsf{TXA}^{>}(t, t_8)\ \} = \{t_0, t_1, t_2\}$，

属性 SAL 上的值比 t_8 小的：$\{t|\mathsf{SAL}^{<}(t, t_8)\ \} = \{t_0, t_2, t_4, t_5\}$，

属性 SAL 上的值比 t_8 大的：$\{t|\mathsf{SAL}^{>}(t, t_8)\ \} = \{t_1, t_3, t_6\}$。

(2) 然后通过这 4 个不同的集合来计算可能的矛盾集合 $\{t|\mathsf{TXA}^{<}(t, t_8)\ \} \cap \{t|\mathsf{SAL}^{>}(t, t_8)\ \} = \{t_3, t_6\}$，$\{t|\mathsf{TXA}^{>}(t, t_8)\ \} \cap \{t|\mathsf{SAL}^{<}(t, t_8)\ \} = \{t_0, t_2\}$。结果集如图4.1中的红色点所示。直观地说，这两个集合中的元组和 t_8 构成的元组对满足了约束 σ_3 矛盾的部分条件，但不是全部条件。

(3) 最后需要进一步检查元组对是否满足约束 σ_3 矛盾的其他条件。比如，当数据集满足$\mathsf{RATE}^{\geq}(t_8, t_3)$ 时，因为 t_3 在集合 $\{t|\mathsf{TXA}^{>}(t_8, t)\ \} \cap \{t|\mathsf{SAL}^{<}(t_8, t)\ \}$ 中，所以 (t_3, t_8) 是约束 σ_3 的一个矛盾。

数据集 D_1 中的任一元组 s 要么$\mathsf{TXA}^{>}(s, t_8)$ 成立；要么 $\mathsf{TXA}^{>}(t_8, s)$ 成立。所以需要访问数据集 D_1 中所有的元组才能计算出步骤 (1) 中的四个集合。这说明根据运算符 "=" 上建立的索引对不等式查询的问题是没有太大帮助，并不能减少访问元组的数目。 □

为了弥补单个带运算符属性的选择性不足，本文提出了一种构建在多个带运算符属性上的索引。这里有一个属性数量的权衡：将更多的属性进行组合，索引自然会有更好的选择性。索引只适用于具有所有带运算符属性的 PODs 。多属性组合的索引无法被较多 PODs 共用。第五章中会利用索引集合覆盖约束集合 Σ 中的约束。如果使用无法被较多 PODs 共用的索引，会增加索引集合的大小和索引的秩 (本章第二节定义)。本文使用构建在两个带运算符属性上的索引。本章介绍两个带符号属性建立最优索引的方法OptIndex 、索引收集矛盾算法Fetch、索引共用时收集矛盾的方法、第三节中详细证明了算法OptIndex 的正确性、第五节说明了收集矛盾方法的正确性。第六章通过相关实验说明了索引查询矛盾的有效性以及选择索引方法的合理性。

收集 POD 的矛盾集合等同于在数据集上进行不等式连接查询。根据第二章第四节中的不等式连接介绍，算法IEJoin 可以应用到动态数据集不等式连接的查询 [41]。然而每一次插入增量数据，IEJoin 都需要进行增量和原始数据的共同排序。所以在需要多次更新数据集或者更新数据集远小于原始数据集的情况下，IEJoin 会有着非常冗余的排序计算。本文索引不需要对每一次插入增量数据都进行重新建立索引，所以本文索引在多次更新数据集或者小部分更新数据时也有很好的性能。第六章的实验表明在收集增量数据集引发的矛盾元组对时本文的索引比对比算法IEJoin 更快。而且数据集大小、增量数据集大小的变化对利用索引收集矛盾的运行时间影响比对算法IEJoin 收集矛盾的运行时间影响小。

第 2 节　索引结构

给定数据集 D 和两个带符号的属性 A^{op_1}、B^{op_2}，其中 op_1, op_2 为四种不等符号，即 $op_1, op_2 \in \{>, \geq, <, \leq\}$。本节介绍一种由数据集上元组组成的索引 $Index(A^{op_1}, B^{op_2})$。这样的索引将原始数据集无序的数据转为多个互相不相交的有序数据结构 Sorted。

2.1　数据预处理以及索引定义

在进行索引结构的介绍前，需要先介绍原始数据集 D 的数据压缩预处理。数据集的预处理使得每个元组在索引 $Index(A^{op_1}, B^{op_2})$ 上有唯一的节点以及索引的每一个节点值都不同。

数据预处理 数据集 D 中存在不同元组在属性 A 和属性 B 上的取值相等。为了使索引上节点值都不同，在建立索引 $Index(A^{op_1}, B^{op_2})$ 前需要将属性 A, B 上取值相同的元组压缩为一个节点。

为了在后续利用矛盾扩展约束时没有信息缺失，需要将属性 A, B 上取值相同的元组标识符记录到集合 Equ_{AB}。将数据集 D 中元组转为一个能够快速建立元组值与元组集合的相互查找关系的映射 map。这个 map 包含了多个键值对 $< key, value >$，其中 key 为一个元组的标识符 t_{id}；$value$ 为和元组 t_{id} 在属性 A, B 取值相同的元组标识符集合 $Equ_{AB}^{t_{id}}$。map 中每一个 key 都是不同的值，所以每一个元组只在一个 Equ 集合之中。索引的节点为 map 中的键值 t_{id} 以及元组 t_{id} 在属性 A, B 上的值。这样的处理方式保证了数据集 D 上的任意元组在索引上有着唯一的节点与之对应。后续收集矛盾元组过程中，如果节点上的元组需要加入到矛盾集合，那么将 map 中以该节点元组的 id 为 key 的 $value$ 加入到矛盾集合。在数据集 D 上进行数据压缩预处理保证了每一个 Sorted 上的不同节点在属性 A, B 上的取值不同。

示例 4-2: 以表格1.1中数据集 D_1 为例，选取属性 $FName$ 以及 $LName$ 进行数据压缩可得映射 map_0。其中 map_0 有 5 个不同的 key 值：t_0,t_1,t_2,t_3,t_6，对应的 $value$ 值为 $\{t_0, t_4\}$,$\{t_1\}$,$\{t_2, t_5\}$,$\{t_3\}$,$\{t_6\}$。

以表格2.1中数据集 r 为例，选取属性 $City$ 以及 ZIP 进行数据压缩可得映射 map_1，其中 map_1 有两个不同的 key 值：t_0 以及 t_3，对应的 $value$ 值为 $\{t_0, t_1, t_2\}, \{t_3, t_4, t_5\}$。 □

索引定义 对于两个带符号的属性 A^{op_1}、B^{op_2}，索引 $\mathsf{Index}(A^{op_1}, B^{op_2}) = \{\mathsf{Sorted}_1, \ldots, \mathsf{Sorted}_k\}$ 满足以下条件：

(1). 对每一个元组 t，存在且只存在一个 Sorted 使得 t 在 Sorted 上。

(2). 在每一个 Sorted 上的不同元组 t', t 满足：如果元组 t' 在元组 t 之后，那么有 $A^{op_1}(t', t)$ 和 $B^{op_2}(t', t)$ 成立。

索引Index(A^{op_1}, B^{op_2}) 将数据集 D 预处理后的所有节点分割成多个排序的数据结构 Sorted。Index$(A^{op_1}, B^{op_2}) = \{\text{Sorted}_1, \ldots, \text{Sorted}_k\}$，称 k 为索引的秩。直觉上来讲，在一个索引上发现矛盾的时间是与索引的秩成正相关性。所以将索引的秩作为一个评价索引好坏的指标。

索引中的每一个 Sorted$_i$ $(i \in [1, k])$ 是由元组标识符 t_{id} 组成的数据结构，可以看做是一个复杂版本的数组。本文使用数组中的记号表示索引 $Index$ 中的单个排序的数据结构 Sorted 中数据。Sorted[0] 表示 Sorted 上的第一个节点，Sorted.$size$ 表示 Sorted 上的节点数目。

可以明显地看到 $Index$ 中包含数据集 D 中的所有元组。$Index$ 节点数目与映射 map 中包含的键值对数目相同。实际上，$Index$ 旨在将数据集 D 中元组按照属性 A,B 的取值分割成不相交的 Sorted，在每一个 Sorted 上分别维护了节点在属性 A 和属性 B 的次序性。这样的分割成多个不相交的 Sorted 想法来源于：无法在全局上有一个比较好的性质，所以只在部分数据上维护 A,B 的次序性。

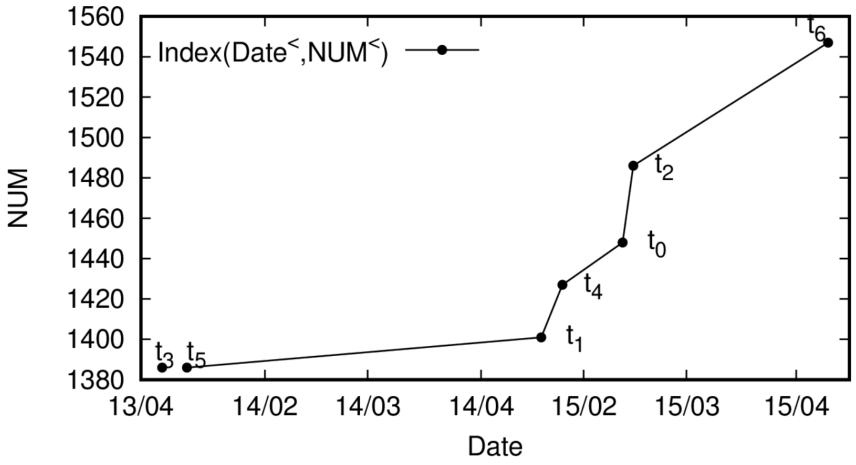


图 4.2: 索引Index(Date$^<$, NUM$^<$) : $Index_0$

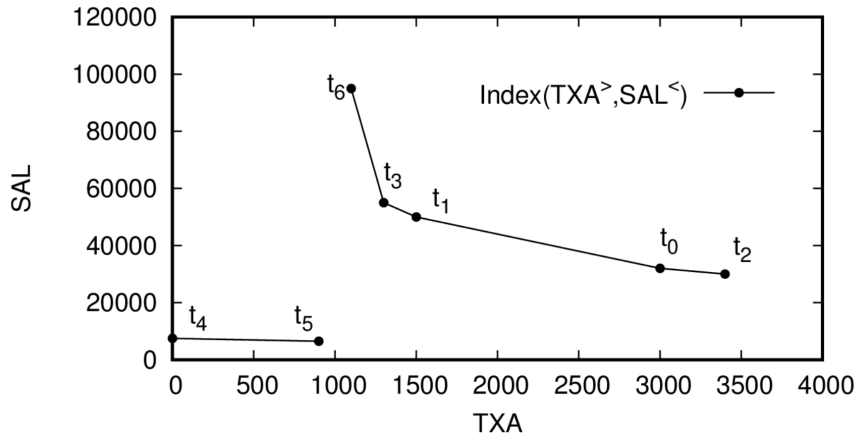


图 4.3: 索引Index(TXA$^>$, SAL$^<$)

示例 4-3: 以表格1.1中数据集 D_1，带运算符属性Date$^<$ 、NUM$^<$ 为例。图4.2中的每一条线或者单个点代表索引中的一个 Sorted。

索引 $Index_0 = \{\text{Sorted}_1, \text{Sorted}_2\}$，其中 $\text{Sorted}_1 = \{t_3\}$，$\text{Sorted}_2$=$\{t_6\ ,\ t_2\ ,\ t_0\ ,\ t_4\ ,\ t_1\ ,\ t_5\ \}$。因而索引 $Index_0$ 的秩为 2。使用数组的记号来表示索引中的每一个 Sorted 的元组。例如 $\text{Sorted}_1[0]$ 表示 Sorted_1 的第一个元组 t_3。可以验证或者从图中看到每一个 Sorted 满足：后面的元组比前面的元组在属性Date 上的值更小，属性NUM 上的值更小。比如在 Sorted_2 中元组 t_2 在元组 t_6 之后，满足 $\text{Date}^<(t_2, t_6), \text{NUM}^<(t_2, t_6)$。

同样，在数据集 D_1 上建立带运算符属性 TXA$^>$ 和 SAL$^<$ 的索引。可以建立如图4.3所示的索引Index(TXA$^>$, SAL$^<$) 。可以验证这两个索引和文章后面部分出现的索引都满足索引的两个条件。 □

2.2　最优索引的构建

给定数据集 D 以及两个带运算符属性 A^{op_1}、B^{op_2}，满足两个条件的索引结构不一定唯一。对于带符号属性 Date$^<$、NUM$^<$，除了索引 $Index_0 = \{[t_6, t_2, t_0, t_4, t_1, t_5], [t_3]\}$ (图4.2)，还有另一个满足条件的索引 $Index_1 = \{[t_6, t_3], [t_2, t_0, t_5], [t_4, t_1]\}$ (图4.4)。$Index_0$ 的秩为 2，$Index_1$ 的秩为 3。因为 $Index_0$ 的秩比 $Index_1$ 的秩更小，会倾向于使用索引 $Index_0$ 来收集矛盾元组对，而不是索引 $Index_1$。下面说明如何构建两个带运算符属性上的最优索引。

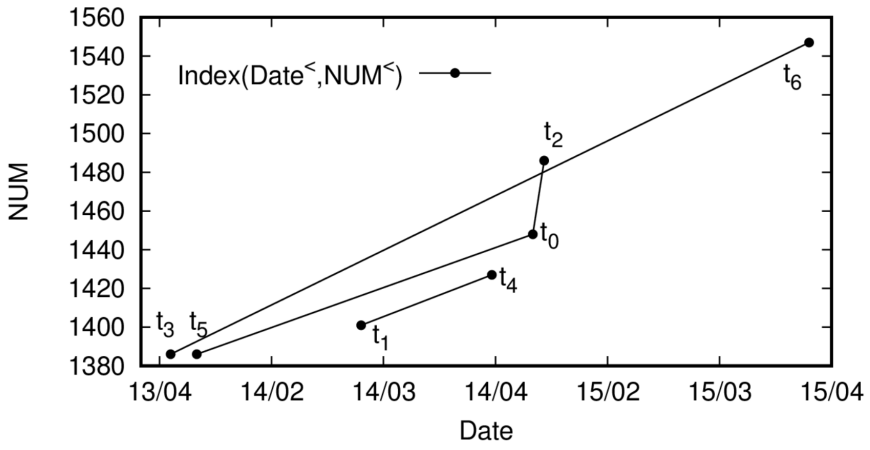


图 4.4: 索引 $Index(Date^<, NUM^<) : Index_1$

最优索引 如果一个索引$\text{Index}_{\text{opt}}(A^{op_1}, B^{op_2})$ 是所有满足 A^{op_1}, B^{op_2} 两个条件的索引中秩最小的索引，则称$\text{Index}_{\text{opt}}(A^{op_1}, B^{op_2})$ 为最优索引。

对于不同的带符号属性对，总是希望能够在最优索引上面进行查找和收集矛盾。算法OptIndex 描述了如何建立数据集上最优索引。这个算法是不依赖于原始数据和带运算符属性对。本章第三节证明了建立最优索引算法OptIndex 的正确性。不影响文章理解时，下文使用Index(A^{op_1}, B^{op_2}) 表示最优索引。

Algorithm 1: OptIndex

input : *relation* D *and marked attributes* A^{op_1}, B^{op_2}

output: *an optimal index for* A^{op_1} *and* B^{op_2} *on* D

```
1  Sort tuples in D on A according to op1, and then on B according to op2
     for breaking ties;
2  Index ← { [ D[0] ] };
3  for each tuple s ∈ D \ D[0] do
4      pos ← NULL; min ← −1;
5      for each Sorted_i ∈ Index do
6          t ← the last element of Sorted_i;
7          if B^{op2}(s, t) then
8              if min = −1 OR min > |(s_B − t_B)| then
9                  min ← |(s_B − t_B)|;
10                 pos ← Sorted_i;
11     if pos! = NULL then append s to the end of pos;
12     else add [s] into Index ;
```

算法流程说明 算法OptIndex 是在数据集 D 上建立两个带运算符属性 A^{op_1}、B^{op_2} 上有最小秩的最优索引，其中 $op_1, op_2 \in \{<, \leq, >, \geq\}$。流程如下：

(1) 将数据集 D 按照元组在属性 A, B 的值以及运算符 op_1, op_2 进行排序：如果 $op_1 \in \{<, \leq\}$ 则按照元组在属性 A 的值降序排列；如果 $op_1 \in \{>, \geq\}$ 则按照元组在属性 A 的值升序排列。如果元组在属性 A 上的值相等时，那么按照元组在属性 B 的值进行排序：如果 $op_2 \in \{<, \leq\}$ 则按照元组在属性 B 的值降序排列；如果 $op_2 \in \{>, \geq\}$ 则按照元组在属性 B 的值升序排列 (第 1 行)。根据数据预处理说明，属性 A, B 值相同的元组会映射成一个节点。所以这样的排序结果唯一，将排序后得到的新数据集仍记为 D。

(2) 首先将步骤 (1) 得到的数据集 D 第一个元组加入初始化索引。初始化索引只包含一个排序数据结构$\mathsf{Sorted}_1 = \{D[0]\}$(第 2 行)。然后逐一将数据集 D 剩余的元组加入到索引之中 (第 3-12 行)。具体操作如下：对当前待处理元组 s，如果在索引的 Sorted 集合中存在若干个 Sorted_i 满足$B^{op_2}(s, t)$ 是成立的，其中元组 t 是 Sorted_i 的最后一个元组。因为 D 是在属性 A 上按照运算符 op_1 排序，所以$A^{op_1}(s, t)$ 成立。那么将元组 s 加入到Sorted_i 之后索引仍然满足索引的两个条件。最后OptIndex 会选择在属性 B 上的变化最小的 Sorted(第 5-10 行)，将元组 s 加入到这个 Sorted 之后 (第 11 行)。如果在当前的索引中找不到Sorted_i 使得元组 s 加到Sorted_i 之后还满足索引两个条件，那么新建一个$\mathsf{Sorted}_j = \{[s]\}$，并将这个$\mathsf{Sorted}_j$ 加入到索引之中 (第 12 行)。

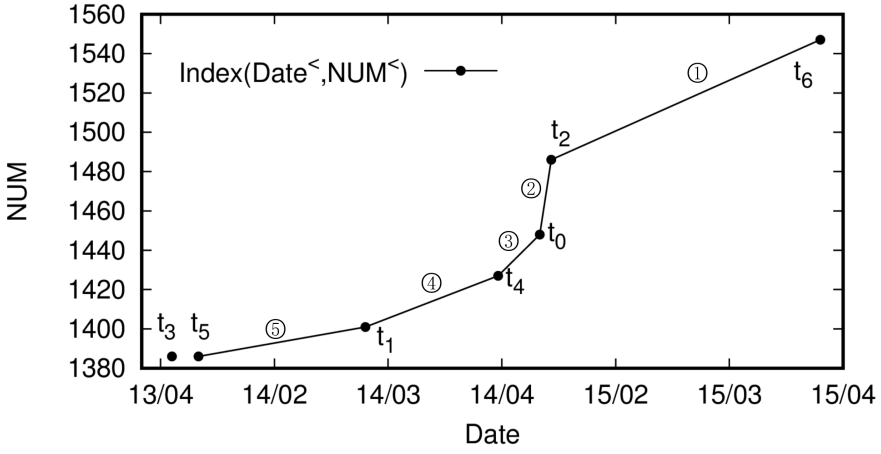


图 4.5: OptIndex 构建Date$^<$ ，NUM$^<$ 上的索引 $Index_0$

示例 4-4: 以算法OptIndex 在数据集 D_1 上建立带运算符属性对 Date$^<$, NUM$^<$ 上的索引 $Index_0$ 为例，说明算法OptIndex 的流程：

首先按照属性Date 的降序进行排序元组，结果为 $[t_6, t_2, t_0, t_4, t_1, t_5, t_3]$。使用数据集中的第一个元组 t_6 初始化索引。初始的索引结构只包含一个排序数据结构Sorted$_1$ =$\{[t_6]\}$。逐一处理数据集 D_1 中剩余的元组，过程如图4.5所示，图中圈圈中的数字表示下面的处理顺序。

(1) 元组 t_2 在属性NUM 上的值比元组 t_6 在属性NUM 上的值小，即元组 t_2, t_6 满足 NUM$^<(t_2, t_6)$。所以元组 t_2 可以加到Sorted$_1$ 之后。处理完元组 t_2 之后得到Sorted$_1$ =$\{[t_6, t_2]\}$。

(2) 元组 t_0 在属性NUM 上的值比元组 t_2 在属性NUM 上的值小，即元组 t_0, t_2 满足 NUM$^<(t_0, t_2)$。所以元组 t_0 可以加到Sorted$_1$ 之后。处理完元组 t_0 之后得到Sorted$_1$ =$\{[t_6, t_2, t_0]\}$。

(3-5) 同样的道理，元组 t_4, t_1, t_5 可以依次加到Sorted$_1$ 之后。处理完元组 t_4, t_1, t_5 后得到Sorted$_1$ =$\{[t_6, t_2, t_0, t_4, t_1, t_5]\}$。

元组 t_3 和元组 t_5 在属性NUM 上有着相同的值，所以元组 t_3 无法加到Sorted$_1$ 之后。需要新建一个Sorted$_2$ = $\{t_3\}$，并将Sorted$_2$ 加到索引 $Index_0$。

最后，最优索引为 $Index_0 = \{[t_6, t_2, t_0, t_4, t_1, t_5], [t_3]\}$。 □

其他说明 简而言之，算法OptIndex 旨在最小化属性 A 、属性 B 上值变化的同时保证 Sorted 上的元组在属性 A 上保持 op_1 的顺序、属性 B 上保持 op_2 的顺序。算法第 1 行对数据集 D 按照属性 A, B 以及运算符 op_1, op_2 的排序方法使得在属性 A 上保持 op_1 的顺序；算法第 3 行依次处理排序后的数据集 D 使得属性 A 上的变化最小；算法第 7 行判断是否可以将新元组加到原有的 Sorted 之后使得在属性 B 上保持 op_2 的顺序；算法第 8-10 行选取在属性 B 上的变化最小的 Sorted 使得属性 B 上的变化最小。本章第三节通过证明命题 2 和命题 3 来证明了算法OptIndex 的正确性。

第 3 节　最优索引的证明

本章节主要证明：算法OptIndex 得到的索引是全部索引中秩最小的索引 (最优索引)。首先证明如下命题 2：

命题 2: 任何一个已有的 A^{op_1}, B^{op_2} 索引 I 都可以在不增加秩的前提下通过有限步的更改后转换为算法OptIndex 的结果 I_{opt}。 □

证明: 可以在有限步内通过如下更改方式将索引 I 转换为 I_{opt}：

(1) 将原始数据集按照算法OptIndex 流程说明 (1) 中描述排序：按照属性 A 和运算符 op_1 的进行排序；当属性 A 上的值相同的时候，按照属性 B 和运算符 op_2 的进行排序。将排序后的元组顺序记为 μ。

(2) 依照最优索引 I_{opt}，按照顺序 μ 逐一处理索引 I 上的元组 s。处理元组 s 过程保证：1、I 仍然满足索引定义中两个条件；2、索引 I 的秩并不会增加；3、要么索引 I 和索引 I_{opt} 同时在元组 s 前没有元组，要么索引 I 和索引 I_{opt} 中元组 s 在同一个元组 t 之后。将每次处理完元组 s 后的索引仍然记为 I。

对元组 s 的处理方法：如果元组 s 满足特殊条件之一，那么直接处理元组 s 之后的元组。特殊条件：(a) 元组 s 是 I_{opt} 某个 Sorted 的第一个元组；(b) 元组 s 在索引 I 和 I_{opt} 当中是处于同一个元组 t 之后。

下面讨论元组 s 在索引 I_{opt} 上是处于元组 t 之后；在变动的索引 I 上并不处于元组 t 之后。为了满足第 3 点，需要将元组 s 更改到元组 t 之后。(在这一步更改中，将元组 s 之后的元组一并和 s 进行移动)

记元组 s 之前的元组为 o，元组 t 之后的元组为 p。那么按照 o, p 是否存在划分为如下 4 种不同情况：(图中每一种颜色表示索引 I 中的一个 Sorted)

(a) 元组 o, p 均不存在的情况。即在当前索引 I 上面：元组 s 之前没有元组；元组 t 之后也没有元组。在这样的情况下，可以直接将元组 s 及 s 之后的元组直接加到元组 t 之后作为一个新的 Sorted。如图4.6所示，将两个 Sorted 合并为一个 Sorted 的更改不会增加索引 I 的秩，反而会减少索引 I 的秩。

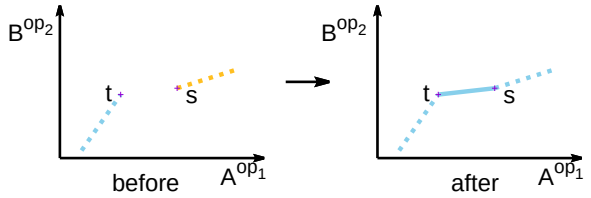


图 4.6: (a): 元组 o, p 均不存在

(b) 当 p 存在，o 不存在的情况。如图4.7所示：断开元组 t、p 的连接；将元组 s 和 s 之后的元组都加到元组 t 之后作为一个新的 Sorted；将元组 p 和 p 之后的元组单独构造成一个 Sorted。很明显这样的操作也不会增加索引的秩。

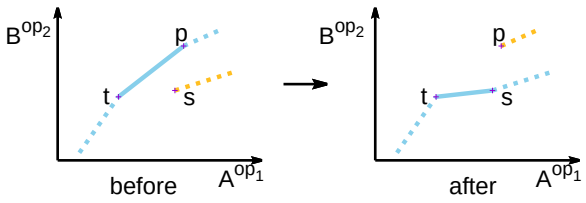


图 4.7: (b):p 存在，o 不存在

(c) 当 o 存在，p 不存在的情况。如图4.8所示：断开元组 o、s 的连接；将元组 s 以及 s 以后的元组都加到 t 之后作为一个新的 Sorted；将元组 o 和 o 之前的元组单独构造成一个 Sorted。这样的操作也不会增加索引的秩。

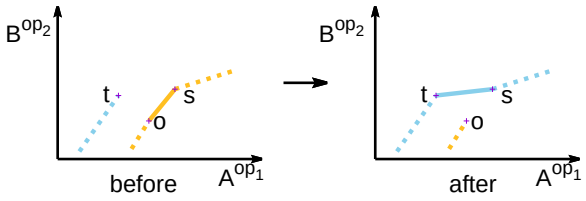


图 4.8: (c): 当 o 存在，p 不存在

(d) 当 o, p 均存在的情况。如图4.9所示：将 s 和 o, 以及 t 和 p 的连接都断开；将元组 s 以及 s 之后的元组都加到 t 之后作为一个新的 Sorted；元组 p 以及 p 之后的元组都加到元组 o 之后作为一个新的 Sorted。这样的操作也不会增加索引的秩。

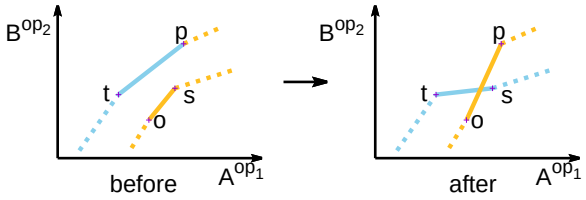


图 4.9: (d):o, p 均存在

显然，按照情况 (a)(b)(c) 更改后的 I 满足索引的定义。对于情况 (d)：
(1) 索引 I 中元组 s 在元组 o 之后，根据索引定义知 $A^{op_1}(s, o)$。
(2) 元组 p 在元组 s 之前进行更改会断掉元组 t、p 之间的连接。这和假设矛盾。所以顺序 μ 上元组 s 在元组 p 之后，元组 p, s 满足 $A^{op_1}(p, s)$。
(3) 算法Fetch 中元组 t 是属性 B 上变化最小的元组,所以元组 t, o 满足 $B^{op_2}(t, o)$。
(4) 索引 I 中元组 p 在元组 t 之后，根据索引定义知 $B^{op_2}(p, t)$。

根据 (1)(2) 及运算符的传递性可知，元组 o, p 满足 $A^{op_1}(p, o)$；根据 (3)(4) 及运算符的传递性可知，元组 o, p 满足 $B^{op_2}(p, o)$。因此可以将元组 p 以及 p 之后的元组都加到元组 o 之后作为一个新的 Sorted。

按照顺序 μ 逐一通过上述的方法处理完所有元组后，索引 I 就转换为算法OptIndex 得到的结果索引 I_{opt}。因此命题2成立。 □

命题 3: 对于两个带运算符属性 A^{op_1} 和 B^{op_2} $(op_1, op_2 \in \{<, \leq, >, \geq\})$，算法OptIndex 得到的索引是所有 A^{op_1}, B^{op_2} 索引中秩最小的索引。 □

命题2说明了任何满足定义的索引，其秩都是大于等于算法OptIndex 得到的结果索引 I_{opt} 的秩。故命题3成立。

在命题2的证明过程中，将 A 和 B 互换后按照OptIndex 的方法仍然可以得到最优的索引。即OptIndex 的最优性和属性 A,B 的顺序无关。

算法复杂度分析 OptIndex 算法的复杂度为 $O(|D| \cdot log(|D|))$。算法中的第 1 行和第 3-12 行都是有各自的复杂度。为了发现合适的Sorted_i 来插入当前考虑的元组，索引需要检查索引中Sorted_i 的最后一个元组。可以维护一个 Sorted 最后一个元组的值排序的结构，然后在排序结构上面进行查找合适的Sorted_i ，复杂度为 $O(log(k))$。在实验中可以发现 $k << |D|$，k 为索引的秩。

第 4 节　索引发现矛盾

本节介绍如何利用最优索引收集违背元组对。为了更清晰说明算法收集违背元组对流程，首先进行一些符号说明：

违背元组的候选集合 对一个元组 s，将满足 $A^{op_1}(s,t)$, $B^{op_2}(s,t)$ 的元组 t 组成的集合记为 $T^s_{A^{op_1},B^{op_2}}$ 。相应地，将满足 $A^{op_1}(t,s)$, $B^{op_2}(t,s)$ 的元组 t 组成的集合记为 $\overline{T}^s{}_{A^{op_1},B^{op_2}}$ 。即$T^s_{A^{op_1},B^{op_2}} = \{t|A^{op_1}(s,t),\ B^{op_2}(s,t)\}$; $\overline{T}^s{}_{A^{op_1},B^{op_2}} = \{t|A^{op_1}(t,s),\ B^{op_2}(t,s)\}$。需要注意，违背元组候选集合中的元组与元组 s 不一定组成违背元组对。为了说明简便，简称为违背元组集合。

Algorithm 2: Fetch

input : *Tuple* s, $\mathsf{Index}(A^{op_1}, B^{op_2})$
output: $T^s_{A^{op_1},B^{op_2}}$, $\overline{T}^s{}_{A^{op_1},B^{op_2}}$

1 $T^s_{A^{op_1},B^{op_2}} \leftarrow \emptyset$; $\overline{T}^s{}_{A^{op_1},B^{op_2}} \leftarrow \emptyset$;
2 **for** *each* $\mathsf{Sorted}_i \in \mathsf{Index}(A^{op_1}, B^{op_2})$ **do**
3 　$p \leftarrow find(s, A^{op_1}, \mathsf{Sorted}_i)$;
4 　$q \leftarrow find(s, B^{op_2}, \mathsf{Sorted}_i)$;
5 　$right \leftarrow max(p,q)+1$; $left \leftarrow min(p,q)$;
6 　**while** $right < \mathsf{Sorted}_i.size$ **do**
7 　　add $\mathsf{Sorted}_i[right]$ into $\overline{T}^s{}_{A^{op_1},B^{op_2}}$;
8 　　$right \leftarrow right + 1$;
9 　**while** $left \geq 0$ **do**
10 　　add $\mathsf{Sorted}_i[left]$ into $T^s_{A^{op_1},B^{op_2}}$;
11 　　$left \leftarrow left - 1$;

算法流程说明 算法Fetch 利用索引 Index(A^{op_1}, B^{op_2}) 收集增量元组 s 的违背元组集合。在索引 Index(A^{op_1}, B^{op_2}) 的每一个 Sorted$_i$ 上做如下操作：

(1) 查找出位置 p (第 3 行)。具体而言为：(a) 当 A^{op_1}(Sorted$_i[0], s$) 时，位置 $p = -1$；或者 (b) 当 $A^{\overline{op_1}}$ (Sorted$_i$[Sorted$_i.size-1$], s) 时，位置 p = Sorted$_i.size-1$；或者 (c) 位置 p 满足 $A^{\overline{op_1}}$(Sorted$_i[p]$, s) 和 A^{op_1}(Sorted$_i[p+1]$,s)。对应的，将 A^{op_1} 替换为 B^{op_2} 则可查找出位置 q (第 4 行)；

(2) 由位置 $right = max(p, q) + 1$ 开始，逐步向右遍历 (增加下标)，将元组 Sorted$_i[right]$ 加入集合$\overline{T}^{s}{}_{A^{op_1},B^{op_2}}$ (第 5-8 行)；

(3) 由位置 $left = min(p, q)$ 开始，逐步向左遍历 (减少下标)，将元组 Sorted$_i[left]$ 加入集合 $T^{s}_{A^{op_1},B^{op_2}}$ (第 9-11 行)。

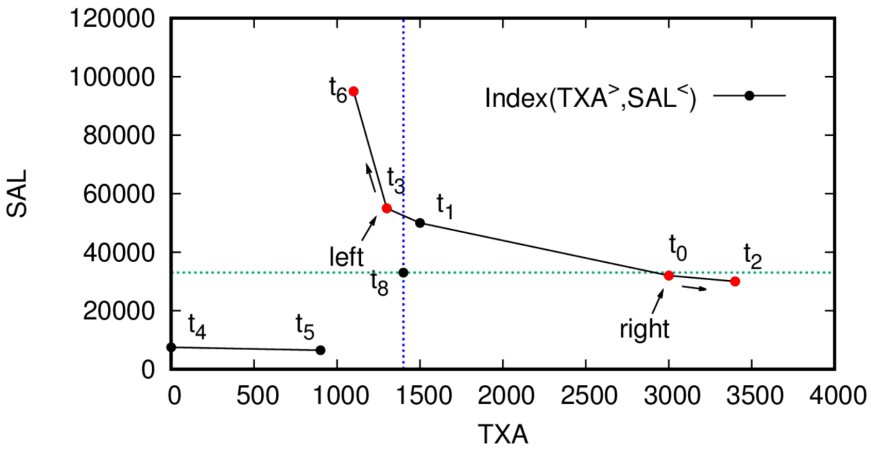


图 4.10: 算法Fetch 在索引Index(TXA$^{>}$, SAL$^{<}$) 的结果

示例 4-5: 以数据集 D_1 上的索引 Index(TXA$^{>}$, SAL$^{<}$) = $\{[t_6, t_3, t_1, t_0, t_2], [t_4, t_5]\}$ 和元组 $t_8 \in \triangle D_1$ 为例。图4.10展示算法Fetch 逐步收集元组至集合 $T^{t_8}_{\text{TXA}>\text{SAL}<}$ 和 $\overline{T}^{t_8}{}_{\text{TXA}>\text{SAL}<}$ 的流程：

(1)Sorted$_1$：位置 p=1 满足TXA$^{\leq}$(Sorted$_1[p], t_8$) 和TXA$^{>}$(Sorted$_1[p+1], t_8$) ；位置 q=2 满足SAL$^{\geq}$(Sorted$_1[q], t_8$) 和 SAL$^{<}$(Sorted$_1[q+1], t_8$) 。

位置 $right = max(p, q)$+1，将 Sorted$_1[right] = t_0$ 加入集合 $\overline{T}^{t_8}{}_{\text{TXA}>\text{SAL}<}$。接着将位置 $right$ 向右移动得到 $right = 4$，将 Sorted$_1[4] = t_2$ 加入集合 $\overline{T}^{t_8}{}_{\text{TXA}>\text{SAL}<}$。

位置 $left = min(p, q)$，将元组 Sorted$_1[left] = t_3$ 加入集合 $T^{t_8}_{\text{TXA}>\text{SAL}<}$。将位置 $left$ 向左移动得到 $left = 0$，将 Sorted$_1[0] = t_6$ 加入集合 $T^{t_8}_{\text{TXA}>\text{SAL}<}$。

(2) Sorted$_2$：位置 $p = 1$ 满足TXA$^{\leq}$(Sorted$_2$ [Sorted$_2$.size-1], t_8) 成立。位置 $q = -1$ 满足SAL$^{<}$(Sorted$_2[0], t_8$) 。

位置 $right$ = Sorted$_2.size$ 以及 $left = -1$，所以没有元组需要加入到集合$T^{t_8}_{\text{TXA}>\text{SAL}<}$ 或者 $\overline{T}^{t_8}{}_{\text{TXA}>\text{SAL}<}$。

综上，$T^{t_8}_{\text{TXA}>\text{SAL}<}$=$\{t_3, t_6\}$，$\overline{T}^{t_8}{}_{\text{TXA}>\text{SAL}<}$=$\{t_0, t_2\}$。 □

算法正确性说明 算法 Fetch 的正确性可以直接由索引的定义推导出：

增量数据集 $\triangle D$ 中的一个元组 s 和数据集 D 上的索引 Index(A^{op_1}, B^{op_2}) 中的 Sorted$_i$ 上的元组 t：(1) 如果 $A^{op_1}(t,s)$ (或者 $B^{op_2}(t,s)$)，那么 Sorted$_i$ 上在元组 t 之后的任意元组 t' 满足 $A^{op_1}(t',s)$ (或者 $B^{op_2}(t',s)$)；以及 (2) 如果 $A^{\overline{op_1}}(t,s)$ (或者 $B^{\overline{op_2}}(t,s)$)，那么 Sorted$_i$ 上在元组 t 之前的任意元组 t' 满足 $A^{\overline{op_1}}(t',s)$ (或者 $B^{\overline{op_2}}(t',s)$)。所以在 $left$ 和 $right$ 之间的元组均与元组 s 没有矛盾。因此Fetch 得到的结果是正确的违背元组集合。

Fetch 时间复杂度说明 对于给定索引Index$(A^{op_1}, B^{op_2}) = \{$Sorted$_1, \ldots,$ Sorted$_k\}$，算法Fetch 需要花费 $\sum_{i\in[1,k]} O(log(n_i)) \leq k \cdot O(log(|D|))$ 在所有的 Sorted$_i$ 上来识别位置 p、q，其中 n_i 是 Sorted$_i$ 中的节点个数以及 $|D| = \sum_{i\in[1,k]} n_i$ 是数据集 D 上的所有元组数目。在每一个Sorted$_i$ 上，需要花费和$T^s_{A^{op_1},B^{op_2}}$ (或者 $\overline{T}^s{}_{A^{op_1},B^{op_2}}$) 线性的时间来收集集合$T^s_{A^{op_1},B^{op_2}}$ (或者 $\overline{T}^s{}_{A^{op_1},B^{op_2}}$) 中的元素。这样必要的收集时间与矛盾集合大小线性相关。

特殊说明 数据预处理将数据集 D 按照属性 A、B 上的值进行映射，将属性 A、B 上值相同的元组映射为索引的一个节点。Equ^t_{AB} 是和元组 t 在属性 A 和 B 上值相等的元组集合。如果将索引上的元组 t 加入集合$T^s_{A^{op_1},B^{op_2}}$ (或者 $\overline{T}^s{}_{A^{op_1},B^{op_2}}$)，那么需要将 Equ^t_{AB} 中的所有元组加入集合 $T^s_{A^{op_1},B^{op_2}}$ (或者 $\overline{T}^s{}_{A^{op_1},B^{op_2}}$)。

索引检测矛盾的时间是与 $log(|D|)$ 和矛盾数目相关而不是 $|D|$。可以看到索引的秩 k 是影响时间的重要因素，所以倾向于选择秩比较小的索引。索引数目也是影响时间的重要因素，本章第五节研究不同索引之间的共用；第五章研究如何选取带运算符属性对覆盖约束集合。第六章在不同数据集上对索引秩、索引数目进行相关实验。

第 5 节　索引的共用说明

带运算符属性对上建立的索引除了收集运算符相同的矛盾，还可以收集其他矛盾。本节展示索引Index(A^{op_1}, B^{op_2}) 的共用。

索引的共用 有两种不同情况能够共用索引：

(1) 利用索引Index$(A^{\geq}, B^{op_2})$ 可以收集满足 $A^{>}$ 和 B^{op_2} 的违背元组集合，只需要在算法Fetch 利用索引收集矛盾集合 $T^s_{A^{>},B^{op_2}}$ (或者 $\overline{T}^s{}_{A^{>},B^{op_2}}$) 时忽略掉满足 $A^{=}(t,s)$ 的元组 t。同理，Index$(A^{>}, B^{op_2})$ 也可以收集满足 $A^{\geq}$ 和 B^{op_2} 的违背元组集合，只是需要在利用索引收集矛盾集合 $T^s_{A^{\geq},B^{op_2}}$ (或者 $\overline{T}^s{}_{A^{\geq},B^{op_2}}$) 时增加满足 $A^{=}(t,s)$ 的元组 t。属性 A, B 互相交换位置并不会影响索引的性质，所以同理可以得到 Index$(A^{op_1}, B^{\geq})$ 可以收集 $T^s_{A^{op_1},B^{>}}$ (或者 $\overline{T}^s{}_{A^{op_1},B^{>}}$)，Index$(A^{op_1}, B^{>})$ 可以收集$T^s_{A^{op_1},B^{\geq}}$ (或者 $\overline{T}^s{}_{A^{op_1},B^{\geq}}$)。同样的，将 $\geq$，$>$ 和 $\leq$，$<$ 互换可以得到的小于等于和小于之间的索引共用。

(2) 通过简单的修改算法 Fetch，就可通过索引 Index($A^<, B^<$) (或者 Index($A^<, B^>$)) 来计算$T^s_{A^<B^>}$ 和 $\overline{T}^s{}_{A^<B^>}$ (或者 $T^s_{A^<B^<}$ 和 $\overline{T}^s{}_{A^<B^<}$)。这样的更改并不会影响 Fetch 的复杂度。不失一般性，下面用算法 3 和例子说明利用索引 Index($A^<, B^<$) 计算$T^s_{A^<B^>}$ 和 $\overline{T}^s{}_{A^<B^>}$ 的方法。

Algorithm 3: 反转的Fetch

input : *Tuple* s, Index($A^<, B^<$)

output: $T^s_{A^<,B^>}$, $\overline{T}^s{}_{A^<,B^>}$

1 $T^s_{A^<,B^>} \leftarrow \emptyset$; $\overline{T}^s{}_{A^<,B^>} \leftarrow \emptyset$;
2 **for** *each* Sorted$_i \in$ Index($A^<, B^<$) **do**
3 　$p \leftarrow find(s, A^<, \text{Sorted}_i)$;
4 　$q \leftarrow find(s, B^<, \text{Sorted}_i)$;
5 　$right \leftarrow max(p, q)$; $left \leftarrow min(p, q) + 1$;
6 　**while** $right \geq left$ **AND** $left <$Sorted$_i.size$ **do**
7 　　add Sorted$_i[left]$ into $\overline{T}^s{}_{A^<,B^>}$ or $T^s_{A^<,B^>}$;
8 　　$left \leftarrow left + 1$;

算法流程说明 反转的Fetch 在 Index($A^<, B^<$) 的每一个 Sorted$_i$ 做如下操作：

(1) 首先查找出位置 p(第 3 行)，q(第 4 行)。具体而言为：(a) 当 $A^<$(Sorted$_i[0], s$) 时，位置 $p = -1$；或者 (b) 当 $A^{\geq}$ (Sorted$_i$[Sorted$_i.size - 1$], s) 时，位置 $p =$ Sorted$_i.size-1$；或者 (c) 位置 p 满足 $A^{\geq}$(Sorted$_i[p]$, s) 和 $A^<$(Sorted$_i[p+1]$,s)。对应的，将 $A^<$ 替换为 $B^<$ 则可查找出位置 q；

(2) 收集矛盾元组对集合与算法Fetch 不同：在Sorted$_i$ 上，由位置 $left = min(p, q)+1$ 开始，逐步向右 (增加下标) 遍历，将元组 Sorted$_i[left]$ 加入集合$\overline{T}^s{}_{A^{op_1}B^{op_2}}$ 或者$T^s_{A^{op_1},B^{op_2}}$ 直到 $right = max(p, q)$ 位置 (第 6-8 行)。

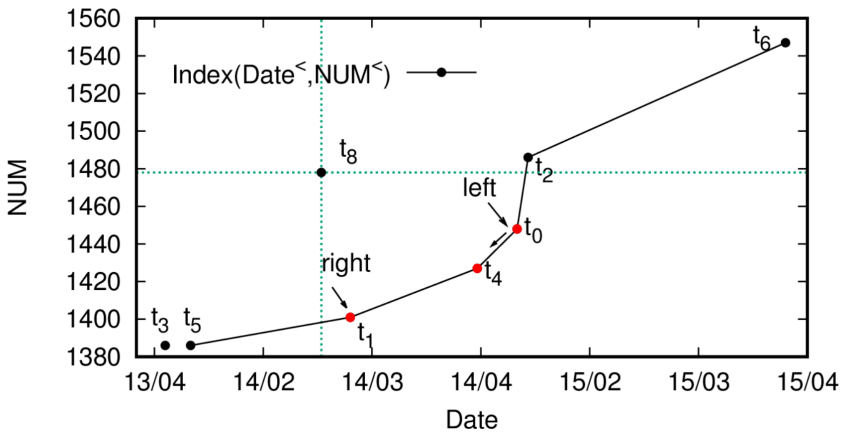


图 4.11: 使用反转Fetch 收集矛盾

示例 4-6: 以数据集 D_1 上索引 Index(Date$^<$, NUM$^<$) $= \{[t_6,\ t_2, t_0, t_4, t_1, t_5], [t_3]\}$ 和元组 $t_8 \in \triangle D$ 为例。图4.11展示反转的Fetch 收集元组至集合$T^{t_8}_{\text{Date}<\text{NUM}>}$ 的流程：

(1)Sorted$_1$ ：位置 p=4 满足 Date$^<$(Sorted$_1[p+1], t_8$) 和 Date$^\geq$(Sorted$_1[p], t_8$)；位置 q=1 满足 NUM$^<$ (Sorted$_1[q+1], t_8$) 和 NUM$^\geq$(Sorted$_1[q], t_8$)。

和原始的算法Fetch 流程不同：位置 $left$ =2 以及位置 $right$=4；然后由位置 $left$ 遍历到 $right$ 进行收集元组到集合$T^{t_8}_{\text{Date}<,\text{NUM}>}$ 或者 $\overline{T}^{t_8}{}_{\text{Date}<,\text{NUM}>}$。所有从 $left$ 到 $right$ 的节点实际上都是属于同一矛盾集合，图中表现为红色点都在元组 t_8 的右下方。因为 Date$^<$(t_8, t_0) 以及 NUM$^>$(t_8, t_0)，所以需要将元组 t_0、t_4 以及 t_1 加入到集合$T^{t_8}_{\text{Date}<\text{NUM}>}$。

(2)Sorted$_2$ ：没有发现新元组需要加入到集合$T^{t_8}_{\text{Date}<,\text{NUM}>}$ 或者 $\overline{T}^{t_8}{}_{\text{Date}<,\text{NUM}>}$。

综上，$T^{t_8}_{\text{Date}<,\text{NUM}>} = \{t_0, t_1, t_4\}$ 以及 $\overline{T}^{t_8}{}_{\text{Date}<,\text{NUM}>}=\emptyset$。 □

算法正确性与复杂度 反转的Fetch 正确性可以通过和Fetch 类似地利用索引定义证明。反转的Fetch 复杂度与Fetch 一致。

第 6 节　本章小结

本章首先对使用索引结构的目的进行说明，介绍了索引结构Index(A$^{\text{op}_1}$, B$^{\text{op}_2}$)的定义。第二节给出了建立给定输入：原始数据集 D 以及带运算符属性 A$^{\text{op}_1}$, B$^{\text{op}_2}$ 建立最优索引的算法OptIndex 。第二节最后说明了算法OptIndex 的复杂度。第三节给出了算法正确性的证明。第四节用伪代码和示例说明利用索引发现矛盾的算法Fetch 流程，以及理论分析了Fetch 的复杂度和说明算法Fetch 的正确性。第五节详细介绍了索引共用情况和反转Fetch 的流程。

第五章　选择、更新索引与增量发现算法

本章说明选取索引覆盖约束集合的算法ChooseIndex 、更新索引机制以及增量算法的整体流程IncPOD。索引覆盖约束集合的算法ChooseIndex 是不依赖与增量数据集 $\triangle D$，可以看做整体增量算法的预处理过程。

第 1 节　选择索引

对集合 Σ 中的每一个 POD 都单独建立索引会造成非常多的时间浪费和重复计算。所以希望找到一个能够高效地发现所有矛盾元组对的索引集合。本节说明如何基于原始数据集 D 以及约束集合 Σ 选取索引。

识别出增量数据集 $\triangle D$ 引起约束集合 Σ 所有矛盾：元组 s, t：(s,t) 或者 (t,s) 违背了约束集合 Σ 中的某些 PODs 。显然元组 s,t 至少有一个元组是在增量数据集 $\triangle D$。本章第三节说明通过索引更新机制可以在索引收集违背元组集合时一并收集两个元组 s, t 均在 $\triangle D$ 的矛盾。现在假设 $s \in \triangle D$ ，$t \in D$；元组 s, t 与约束 $\sigma = \{A_1^{op_1}, \ldots, A_m^{op_m}\} \hookrightarrow B^{op'}$ 矛盾，那么一定成立 (a) $A_i^{op_i}(s,t)$ 以及 $B^{\overline{op'}}(s,t)$，其中 $i \in [1,m]$ 或者 (b) $A_i^{op_i}(t,s)$ 以及 $B^{\overline{op'}}(t,s)$，其中 i∈$[1,m]$。

等值索引 如果在约束 $\{A_1^{op_1}, \ldots, A_m^{op_m}\} \hookrightarrow B^{op'}$ 中某个运算符 op_i (i ∈[1,m]) 为等于 "="，可以构建属性 A_i 上的等值索引。等值索引就是一个简单地依赖于属性 A_i 上的值进行排序的索引，记为Index($A_i^{=}$) 。利用等值索引 Index($A_i^{=}$) 来发现满足$A_i^{=}(t,s)$ 的元组 t，然后检查元组 t 是否满足约束 σ 的其他条件。

假设 op_i, op_j∈$\{<, \leq, >, \geq\}$ $(i,j \in [1,m])$。如果元组对 s,t 与约束 σ 违背，那么 $t \in T^s_{A_i^{op_i}, A_j^{op_j}} \cup \overline{T}^s_{A_i^{op_i}, A_j^{op_j}}$。在索引Index($A_i^{op_i}, A_j^{op_j}$) 上使用算法Fetch(第四章第四节) 来计算集合 $T^s_{A_i^{op_i}, A_j^{op_j}}$ 和 $\overline{T}^s_{A_i^{op_i}, A_j^{op_j}}$，然后对两个违背元组集合中的元组 t 进一步检查是否满足约束 σ 的其余条件。

利用索引覆盖 Σ 中的 POD 通过上述描述，可知 (1) 对约束 $\sigma = \{A_1^{op_1}, \ldots, A_m^{op_m}\} \hookrightarrow B^{op'}$，可以使用三种不同的索引进行覆盖 (a) 如果某 $op_i, i \in [1, \cdots, m]$ 为等于时，可使用等值索引 Index($A_i^{=}$) ，或者 (b) Index($A_i^{op_i}, A_j^{op_j}$) ，或者 (c) Index($A_i^{op_i}, B^{\overline{op'}}$) ；(2) 可以使用索引 Index($A^{\geq}, A'^{>}$) 或者 Index($A^{>}, A'^{<}$) 来覆盖有 $A^{>}A'^{>}$ 带运算符属性的 PODs 。第四章中的索引共用和算法反转的Fetch说明索引Index($A^{\geq}, A'^{>}$) 可以收集有 $A^{>}A'^{>}$ 带运算符属性的 PODs 矛盾元组对；(3) 注意到如果 op' 为 "="，用索引Index($A_i^{op_i}, B^{\geq}$) 或者Index($A_i^{op_i}, B^{\leq}$) 时需要查找四个集合$T^s_{A_i^{op_i}, B^{>}}$ 、$\overline{T}^s_{A_i^{op_i}, B^{>}}$、$T^s_{A_i^{op_i}, B^{<}}$ 和 $\overline{T}^s_{A_i^{op_i}, B^{<}}$ 才能够找出矛盾元组。

如果对约束集合 Σ 的每一个约束 $\sigma \in \Sigma$，索引集合 $Ind(\Sigma)$ 中至少有一个索引能收集约束 σ 的违背元组对集合，称这样的情况为：Σ 被索引集合 $Ind(\Sigma)$ 覆盖。发现增量数据 $\triangle D$ 中一个元组 s 与不同 PODs 的矛盾元组，可以只访问一次覆盖这些 PODs 索引。

等值索引的打分 直觉上，如果在属性 A 上面不同取值的数目越多，在属性 A 上的运算符 "=" 则会有更好的选择性。所以使用如下打分函数来评价等值索引$\mathsf{Index}(A^{=})$，更倾向于选择更小分数的索引。

$$score(\mathsf{Index}(A^{=})) = 1 - \frac{\text{属性 A 上不同值的数目}}{|D|}$$

非等值索引的打分 对于非等值索引的打分相比于等值索引的打分会更加复杂。第四章中提到索引的秩和索引的查询时间有关。直接计算数据集 D 中所有带运算符属性对或者约束集合 Σ 中所有带运算符属性对构成的可能索引会非常麻烦和耗时。使用如下打分函数来评价非等值索引 $\mathsf{Index}(A^{op_1}, B^{op_2})$，更倾向于选择分数较小的非等值索引。

$$score(A^{op_1}, B^{op_2}) = \frac{1-|r(A,B)|}{coverage(A^{op_1}, B^{op_2})}$$

其中 $coverage(A^{op_1}, B^{op_2})$ 是索引 $\mathsf{Index}(A^{op_1}, B^{op_2})$ 能覆盖 Σ 中的 PODs 数目，$|r(A,B)|$ 是属性 A 和属性 B 相关系数的绝对值。$r(A,B)$ 通过数据集 D 上所有元组 $t \in D$ 在属性 A,B 上的值 $t[A], t[B]$ 计算，方式如下所示：

$$r(A,B) = \frac{\sum(t[A]*t[B]) - \sum t[A] \sum t[B]}{\sqrt{\sum t[A]^2 - (\sum t[A])^2}\sqrt{\sum t[B]^2 - (\sum t[B])^2}}$$

非等值索引打分分数越小：建立在相关系数较高的带运算符属性对；能够覆盖掉更多约束集合 Σ 中的 PODs 。此外，如果属性 A, B 是正相关，即属性 A, B 相关系数 $r(A,B) > 0$ 时，索引 $\mathsf{Index}(A^{>}, B^{>})$ 和 $\mathsf{Index}(A^{\geq}, B^{>})$ 更倾向于有着更小的秩。相反的，在属性对 A, B 是负相关，即属性 A, B 相关系数 $r(A,B) < 0$ 时，索引 $\mathsf{Index}(A^{>}, B^{<})$ 和 $\mathsf{Index}(A^{\geq}, B^{<})$ 更倾向于有更小的秩。

示例 5-1: 表格1.1中的数据集 D_1，因为相关系数 $r(\mathsf{Date}, \mathsf{NUM})$=0.88，所以更倾向于建立索引 $\mathsf{Index}(\mathsf{Date}^{<}, \mathsf{NUM}^{<})$ 。索引$\mathsf{Index}(\mathsf{Date}^{<}, \mathsf{NUM}^{<})$ 能够覆盖 σ_4，并且比索引$\mathsf{Index}(\mathsf{Date}^{<}, \mathsf{NUM}^{>})$ 有着更小的秩。 □

选择索引算法流程 算法 ChooseIndex 基于原始数据集 D 和已有约束集合 Σ 选择覆盖约束集合 Σ 中所有有效约束的索引集合 $Ind(\Sigma)$，流程如下：

首先选择等值索引：如果等值索引$\mathsf{Index}(A^{=})$ 能够覆盖约束集合 Σ 中的 PODs 并且分数是低于一个自定义的阈值 l，则加入索引集合$Ind(\Sigma)$；并将等值索引$\mathsf{Index}(A^{=})$ 覆盖的约束从 Σ 中移除 (第 1-4 行)。本文第六章的实验中，设置阈值 $l = 0.6$。为了计算方便，打分分数在抽样数据集上计算。

Algorithm 4: ChooseIndex

input : *a set* Σ *of PODs*

output: *a set* $Ind(\Sigma)$ *of indexes for covering PODs in* Σ

1 **foreach** $\sigma = \mathcal{X} \hookrightarrow \mathcal{B}^{op'} \in \Sigma$ **do**
2 **if** *there exists* $A^{=} \in \mathcal{X}$ *and* $score(\mathsf{Index}(A^{=})) < l$ **then**
3 add $\mathsf{Index}(A^{=})$ into $Ind(\Sigma)$;
4 remove from Σ all PODs covered by $\mathsf{Index}(A^{=})$;
5 $Candidates \leftarrow \{\}$;
6 **foreach** $\sigma = \mathcal{X} \hookrightarrow \mathcal{B}^{op'} \in \Sigma$ **do**
7 **for** ${A_i}^{op_i}, {A_j}^{op_j} \in \mathcal{X} \cup \{B^{\overline{op'}}\}$ $(op_i, op_j \in \{<, \leq, >, \geq\})$ **do**
8 add $({A_i}^{op_i}, {A_j}^{op_j})$ into $Candidates$, in ascending order of $score({A_i}^{op_i}, {A_j}^{op_j})$;
9 **for** *each* $({A_i}^{op_i}, {A_j}^{op_j})$ *in* $Candidates$ **do**
10 **if** Σ *is empty* **then** Break;
11 **foreach** $\sigma = \mathcal{X} \hookrightarrow \mathcal{B}^{op'} \in \Sigma$ *such that* ${A_i}^{op_i}, {A_j}^{op_j}$ *cover* σ **do** remove σ from Σ;
12 $op_j \leftarrow$ reverse op_j according to $r(A_i, A_j)$ and parameter α;
13 add $\mathsf{Index}(A_i^{op_i}, A_j^{op_j})$ into $Ind(\Sigma)$;
14 **for** *each* $\sigma = \mathcal{X} \hookrightarrow \mathcal{B}^{op'} \in \Sigma$ **do**
15 choose $A_i^{=} \in \mathcal{X}$ with the smallest $score(\mathsf{Index}(A_i^{=}))$ among all $A^{=} \in \mathcal{X}$, and add $\mathsf{Index}(A_i^{=})$ into $Ind(\Sigma)$;

然后选择非等值索引：将约束集 Σ 中存在的所有带运算符属性对加入预选集合 $Candidates$；并且将预选集合中的带运算符属性对按照索引的分数升序排序 (第 6-8 行)。逐一计算预选集合 $Candidates$ 中的带运算符属性对：如果能够覆盖新的 PODs ，那么建立该带运算符属性对上的索引 (第 9-13 行)。为了得到秩更小的索引，根据参数 $\alpha > 0$，相关系数 $r(A_i, A_j)$ 调整索引$\mathsf{Index}(A_i^{op_i}, A_j^{op_j})$上的运算符 op_j(第 12 行)。第六章实验中设置了一个参数 $\alpha > 0$，相关系数 $r(A_i, A_j) > \alpha$ 时，建立索引 $\mathsf{Index}(A_i^{<}, A_j^{<})$ 或者$\mathsf{Index}(A_i^{>}, A_j^{>})$ 。

最后，如果仍然存在没有被覆盖的 PODs，那么可以很明显地看出这些 PODs 一定包含 $A^{=}$，并且 $score(\mathsf{Index}(A^{=})) \geq l$。则启发式地选择有最小分数的等值索引 $A_i^{=}$ 来覆盖剩余的 PODs (第 14-15 行)。

算法复杂度 在最坏的情况下，算法ChooseIndex 需要 $O(|R|^2 \cdot |\Sigma|)$ 覆盖约束集合 Σ 中所有的 PODs 。其中 $|R|$ 是属性个数，$|\Sigma|$ 是约束集合 Σ 中的所有 PODs 数目。计算出索引 $\mathsf{Index}(A^{=})$ 和 $\mathsf{Index}(A_i^{op_1}, A_j^{op_2})$ 分数的时间上限分别为 $O(|R| \cdot |D|log(|D|))$ 和 $O(|R|^2 \cdot |D|)$。两个复杂度均与数据集大小有关。因此，本文使用数据集 D 中抽样数据估计索引分数。

第 2 节　跳表的查询与插入

根据上述描述可知，索引需要进行大量的查找以及插入操作，但是不需要进行删除操作。下面介绍实际操作中常见的数据结构，并根据索引需求进行比较说明本文选取数据结构跳表表示索引的原因。

2.1　常见的数据结构对比

数组 数组是固定大小存储相同类型元素的顺序集合。在早先的数组使用中，数组建立后无法进行扩展。随着编译器的更新，可变长数组也成了可选功能。但是因为可变数组在使用中会出现栈溢出等安全问题，所以可变长数组并没得到大范围使用。数组的数据结构非常不便于大量插入操作。数组的优点是能够快速地进行查找。但是进行插入时，需要进行大量的数据移动。

链表 链表是一种通过指针连接节点的数据结构。在链表中，无法进行节点跨越查找，所以链表的查找复杂度非常高。因为数据是单个节点进行存储，所以只需要少量的时间即可插入节点。

二叉树 二叉树是将数据按照树的结构进行存储的数据结构。在一般情况和最坏情况下时间复杂度相差较大。时间复杂度与二叉树的结构密切相关。然而和哈希表一样，也很难在不同数据集上保持比较好的插入时间复杂度。

AVL 树 平衡二叉树 (AVL) 是二叉树的一种优化数据结构，在一般情况和最坏情况下查找时间复杂度均良好。但是平衡二叉树需要在插入时需要进行节点的旋转操作，所以插入时间复杂度会高。

红黑树 红黑树是一种自平衡二叉查找树。其查找时间与插入时间复杂度均与跳表相同。红黑树本身是树的结构，没有叶子节点间的连接，这就导致了遍历收集矛盾元组复杂度远大于跳表收集矛盾元组的复杂度。

B 树/B+ 树 B 树是一个节点可以多于 2 个子节点的查找树，能够实现快速地查找操作和写操作。但是遍历数据和其他查找树一样需要中序遍历。B+ 树是 B 树的扩展，其所有叶子结点构成一个有序链表。但是 B+ 树的插入操作会涉及到多个节点的链接变化或者节点数据更改。

跳表 SkipList 跳表是链表与数组结合的一种轻量级数据结构，和 AVL 树、B 树/B+ 树、红黑树查找复杂度一致为 $O(log(N))$，其中 N 为数据集大小。这里需要说明的是，度为 M 的 B 树/B+ 树的查找复杂度在 $O(log_{M-1}(N))$ 到 $O(log_{M/2}(N))$ 之间，根据定义可知，都是与 $O(log(N))$ 相等。在遍历数据的时候，跳表的复杂度和 B+ 树一致，只需要遍历最底层的链表即可。跳表的插入复杂度为 $O(log(N))$，而这样的复杂度是来源于查找需要插入的位置和更新节点所需。在下一节会详细说明在索引上进行矛盾收集的同时进行节点的插入。这样的性能是其他数据结构无法保证的。

2.2　跳表结构

跳表结构对后续索引的插入与更新有着非常大地帮助，下面详细说明跳表的结构以及查询和插入的时间复杂度。

在一个如图5.1所示的有首节点 $HEAD$，尾结点 $TAIL$ 的单链表上查找数据时，需要从头到尾的遍历单链表。

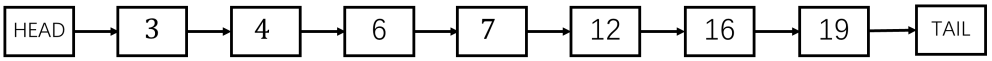


图 5.1: 原始单链表

跳表是将单链表上的部分数据组成一个查找单链表的预先查询步骤，旨在跳过链表中的一些数据。如图5.2中所示结构：最底层 (第一层) 为原始单链表，第二层为单链表中的部分数据，第三层为第二层的部分数据。每一个非底层节点均有一个向下的指针，指向在层数低的链表中同数据值对应的位置。

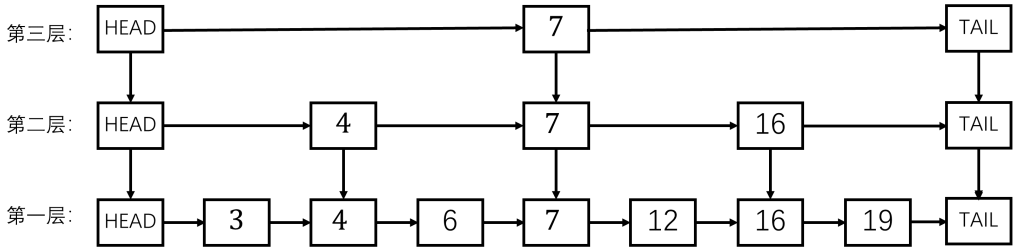


图 5.2: 对应的跳表结构

跳表的查询流程 跳表结构上进行搜索总是从跳表的最上层开始逐步往下层进行查找。在跳表层数为 i+1 的节点相当于给层数为 i 的节点做辅助结构，能够在查找的过程中帮助跳过一些层数为 i 的节点。

示例 5-2: 在图5.2上查找节点值为 7 的数据时，可以直接在第三层中找到数据值为 7 之后，向下找到底层链表中的位置。

查找数据值为 19 时，首先从第三层进行查找，发现头节点后的节点值为 7 比 19 小；向下到第二层进行查找，发现节点值为 7 后的节点值为 16 比 19 小；从第一层值为 16 的节点开始向后遍历，找到值为 19 的节点。　□

跳表的建立流程 建立跳表等同于在空跳表上逐一插入单链表上的所有节点。通过上述跳表的结构和查询流程说明可知跳表结构的性能和每一层与下一层节点数目之比密切相关。设置参数 $limit$ 来限制跳表中第 i 层与 $i+1$ 层节点数目之比，一般选取 $limit = 0.5$。如果直接将增加数据插到最底层，当插入的数据过多时，无法保证跳表查询的良好性质。所以在增加节点的时候，层数较高的层也需要进行改动。因此跳表插入新节点的流程中，也会依据参数 $limit$ 来判断插入节点的层数。这样就能保证跳表插入节点后仍然保持良好的性质。

跳表的插入流程 跳表插入数据的机制是插入数据后仍然保持良好性质的关键。本文采用随机数 $rand$ 来判断插入节点需要变动的层数 [42]：假设在总共层数为 h 的跳表中插入节点 $node$：选择和已有跳表结构一致的参数 $limit$；首先将节点 $node$ 插入到最底层 (第一层)；当节点插入到第 i 层时，随机选取 $[0,1]$ 之间的随机数 $rand$，如果随机数 $rand < limit$，那么将节点插入第 $i+1$ 层；如果随机数 $rand \geq limit$ 时，则结束插入节点流程。每一层都独立地选取新 $[0,1]$ 之间的随机数来判断是否需要继续向上一层插入节点。

示例 5-3: 在图5.2的跳表中插入数据值为 23 的节点。通过查询操作知道插入位置为节点值为 19 之后，即底层单链表的最后。选取 $limit = 0.5$，首先将 23 插入到第一层后；假设在第一层中选取随机数 $rand = 0.36$，则需要将 23 插入第二层；假设在第二层选取随机数 $rand = 0.42$，则需要继续将 23 插入跳表第三层；假设第三层选取随机数 $rand = 0.12$，则需要新建新层：有着头节点 $HEAD$，尾节点 $TAIL$ 以及节点 23 的新层。最后插入结果如图5.3所示。 □

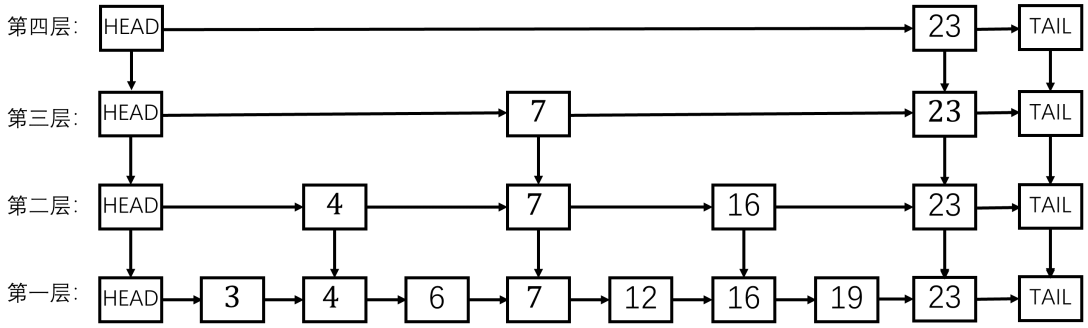


图 5.3: 更新后的跳表结构

特别说明 1、原始跳表总层数为 h，那么插入新节点的流程中有可能将节点插入到 $h+1$。甚至当随机数满足条件时，可能会一直将新节点插入到新的层。这样会造成很多资源浪费，并且有损跳表的查找和插入性质。所以在插入一个节点时，最多只允许加入到第 $h+1$ 层。

2、参数 $limit$ 的选取决定索引的结构与性质。建立跳表等同于在空的跳表中逐个插入数据，所以 $limit$ 应该与建立跳表时设置的参数一致。

3、跳表一般是双向链表。文中前面章节提到的索引共用以及章节中索引收集矛盾的算法的需要，索引均是双向链表。为了便于理解，在本节的示例图片中均采用单链表表示。

跳表复杂度说明 如上描述，跳表的性能与参数 $limit$ 密切相关，其查询、插入复杂度为 $O(log_{\frac{1}{limit}}(n))$。如 $limit = 0.5$，则跳表查询、插入复杂度为 $O(log_2(n))$。这样的查询、插入复杂度等同于平衡二叉树查询、插入复杂度。如果 $limit = \frac{1}{3}$，则跳表查询、插入复杂度为 $O(log_3(n))$。这样的查询、插入复杂度等同于度为 3 的 $B+$ 树查询、插入复杂度。

第 3 节　索引更新

本文使用一种轻量级的数据结构跳表来实现索引。每一个Sorted_i 被表示为一个特殊的跳表。图5.4 展示了例子4-4中的索引 $\mathsf{Index}(\mathsf{Date}^{<}, \mathsf{NUM}^{<})$ 中的Sorted_1。因为数据集 D 上的预处理，所以索引的最低层链表有着如图5.5所示的对应关系，每一个节点对应一个 Equ 集合。

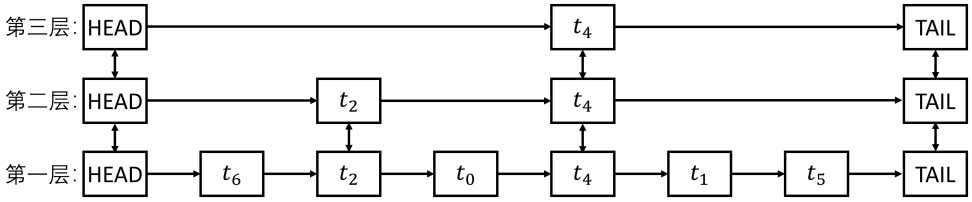


图 5.4: 索引的结构

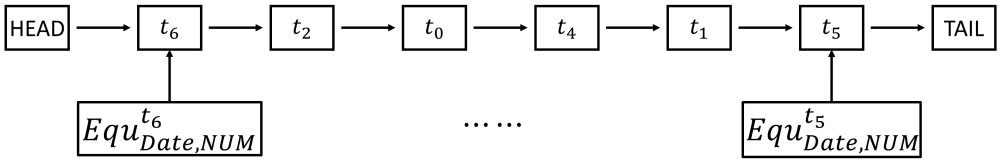


图 5.5: 索引最底层对应关系

3.1　索引的更新说明

第四章表明可以通过索引快速地进行发现和收集元组 $t{\in}D$ 和 $s{\in}\triangle D$ 上引起的与 POD 的矛盾。那么还需要处理两个元组 s,t 均在增量数据集 $\triangle D$ 上引发的矛盾。为了同样能够通过索引来发现这样的矛盾，需要在索引发现元组 $t{\in}D$ 和 $s{\in}\triangle D$ 上引起的与 POD 的矛盾的矛盾同时更新索引的节点。

索引$\mathsf{Index}(A^{op_1}, B^{op_2})$ 上插入元组 t 后需要满足索引两个条件才可以继续使用索引发现和收集矛盾。索引的共用并不会对索引的结构有影响，所以在判断元组 t 是否能加入到索引原有的 Sorted 条件与普通索引一致。

索引更新的判断条件 插入元组 t 时，只需要满足下列条件其一即能够将元组 t 加入到索引$\mathsf{Index}(A^{op_1}, B^{op_2})$ 上原有的Sorted_i：

1、元组 t 与该Sorted_i 上的某个元组 t' 在属性 A, B 上的取值相等。

2、在该Sorted_i 上满足三个条件之一：(a)、$\mathsf{A}^{\mathsf{op}_1}(t, \mathsf{Sorted}_i[\mathsf{Sorted}_i.size()-1])$，$\mathsf{B}^{\mathsf{op}_2}(t, \mathsf{Sorted}_i[\mathsf{Sorted}_i.size()-1])$；(b)、$\mathsf{A}^{\mathsf{op}_1}(\mathsf{Sorted}_i[0], t)$，$\mathsf{B}^{\mathsf{op}_2}(\mathsf{Sorted}_i[0], t)$；(c)、能找到位置 $pos \in [0, \mathsf{Sorted}_i.size()-2]$ 满足 $\mathsf{A}^{\mathsf{op}_1}(t, \mathsf{Sorted}_i[pos])$，$\mathsf{B}^{\mathsf{op}_2}(t, \mathsf{Sorted}_i[pos])$，$\mathsf{A}^{\mathsf{op}_1}(\mathsf{Sorted}_i[pos+1], t)$，$\mathsf{B}^{\mathsf{op}_2}(\mathsf{Sorted}_i[pos+1], t)$。

第一种情况说明：新加入的元组 t 与某个Sorted$_i$ 上的元组 t' 在属性 A, B 上相等。相等的情况等同于元组 t 与原始数据集 D 和在元组 t 之前的增量数据上某个元组 t' 在属性集 A, B 上相同。增量数据与原始数据集上在属性集 A, B 上有相同数据值的可能性非常大。尤其是在形如姓名、工号、工资等有特殊实际含义的属性上，增量数据中出现的数据值一般在原始数据集中出现过。

第二种情况说明：(a) 如果元组 t 满足 $A^{op_1}(\text{Sorted}_i[0], t)$，$B^{op_2}(\text{Sorted}_i[0], t)$，则元组 t 可以插入到Sorted$_i$ [0] 之前，即在Sorted$_i$ 最前插入元组 t，选取 $pos = -1$；(b) 如果元组 t 满足 $A^{op_1}(t, \text{Sorted}_i[\text{Sorted}_i.size-1])$，$B^{op_2}(t, \text{Sorted}_i[\text{Sorted}_i.size-1])$，则元组 t 可以插入到Sorted$_i$ [Sorted$_i$.size-1] 之后，即在Sorted$_i$ 最后插入元组 t，选取 $pos = \text{Sorted}_i.size - 1$；(c) 如果查找到满足 $A^{op_1}(t, \text{Sorted}_i[pos])$，$B^{op_2}(t, \text{Sorted}_i[pos])$，$A^{op_1}(\text{Sorted}_i[pos+1], t)$，$B^{op_2}(\text{Sorted}_i[pos+1], t)$ 的位置，才能保证元组 t 能够插入到Sorted$_i$ $[pos]$ 之后。根据算法Fetch 和反转Fetch 的说明，位置 p 满足 $A^{\overline{op_1}}(\text{Sorted}_i[p], t)$ ，$A^{op_1}(\text{Sorted}_i[p+1], t)$ 根据符号的反的定义，则在 $A^{\neq}(\text{Sorted}_i[p], t)$ 时其等同于满足 $A^{op_1}(t,\text{Sorted}_i[p])$ 和 $A^{op_1}(\text{Sorted}_i[p+1],t)$。即这里的 p 和 pos 是相等的。同样的，根据位置 q 的定义，q 和 pos 也是相等的。因此得出：在 $A^{\neq}(\text{Sorted}_i[p], t)$，$B^{\neq}(\text{Sorted}_i[q], t)$ 时，如果算法Fetch 发现在Sorted$_i$ 上位置 $p = q$，则表明元组 t 能够插入到该Sorted$_i$ 。

补充说明 1、当算法Fetch 发现的位置 $p = q$ 时，说明整个Sorted$_i$ 上的元组与元组 t 都构成矛盾。从宏观来讲，一个元组 t 很难与整个数据集 D 上的元组矛盾。但是当数据被分为一个个 Sorted 后，存在某一个Sorted$_i$ 上的元组与元组 t 是全部矛盾的概率提高了。一个Sorted$_i$ 上的节点数目或者元组数目是小于原始数据集大小，所以即使元组 t 能够插入到若干个Sorted$_i$ 中也不会影响到整体索引收集矛盾元组对集合的效率。

2、在插入元组 t 时，可以在算法Fetch 收集元组矛盾的同时进行索引的更新：如果发现满足条件 1 的Sorted$_i$ ，则直接将元组 t 加入对应的 Equ 集合中；在满足条件 2 的Sorted$_i$ 上计算其在属性 B 上的变化量 $dis(\text{Sorted}_i)$，即特殊情况 (1)：$dis(\text{Sorted}_i) = |t[B] - \text{Sorted}_i[0][B]|$，特殊情况 (2):$dis(\text{Sorted}_i) = |t[B] - \text{Sorted}_i[\text{Sorted}_i.size - 1][B]|$。一般情况：$dis(\text{Sorted}_i) = |t[B] - \text{Sorted}_i[pos][B]|$。如果有多个 Sorted 时，则启发式地选取在属性 B 上的变化量最小的 Sorted。如果没有，则单独建立一个 Sorted=$\{t\}$ 加入索引中。这样的插入方式保证了插入新元组 t 之后的索引中每个 Sorted 仍然满足条件 2(见索引定义)。

3、在算法Fetch 发现的位置 p, q 满足 $A^{=}(\text{Sorted}_i[p], t)$ 或者 $B^{=}(\text{Sorted}_i[q], t)$ 时，根据 Sorted 的定义可知只需要多检查位置 $p-1, q-1, p+1, q+1$ 处是否能够插入元组 t 即可。

4、因为普通的跳表中数据只需要满足数据的大小顺序即可，所以在普通的跳表中插入数据时，只需要查找出数据的节点前后是否能够满足顺序。但是 Sorted 上是两个属性上的值，所以需要判断两个属性上是否均满足顺序关系。

在索引共用下的更改 1、在反转索引中，因为索引的收集与索引的结构并无太大关系，其条件以及做法和普通索引一致。只是在位置 $p = q$ 时表示的含义相反，反转索引中在Sorted_i 位置 $p = q$ 时表明元组 t 与Sorted_i 上的元组没有矛盾。

2、在使用符号为 $\geq$ 或者 $\leq$ 的索引来查找 $>$ 或者 $<$ 的时候，因为位置 p 满足 $A^{op_1}(t,\mathsf{Sorted}_i[p])$ 和 $A^{op_1}(\mathsf{Sorted}_i[p{+}1],t)$，所以只需要多考虑 $p-1$，$q-1$，$p+1$，$q+1$ 处的元组即可。

索引更新流程 每一个索引上面都需要根据增量数据 $\triangle D$ 进行更新。在每一个索引 I 上，(1) 通过和建立索引 I 时相同的排序方法来对 $\triangle D$ 进行排序 (算法OptIndex 的第一行)；(2) 对每一个增量数据集的元组 $s{\in}\triangle D$，通过索引 I 发现与 s 有关的矛盾元组然后依据 s 更新索引 I。通过这样的方法，可以在更新索引的过程中，将增量数据集中的元组插入索引中，就可以发现 $s, s'{\in}\triangle D$ 的矛盾。接下来说明插入元组 s 可以和查找索引上面和 s 的矛盾一起进行 (算法 Fetch)。更新索引的时间并不会特别多。

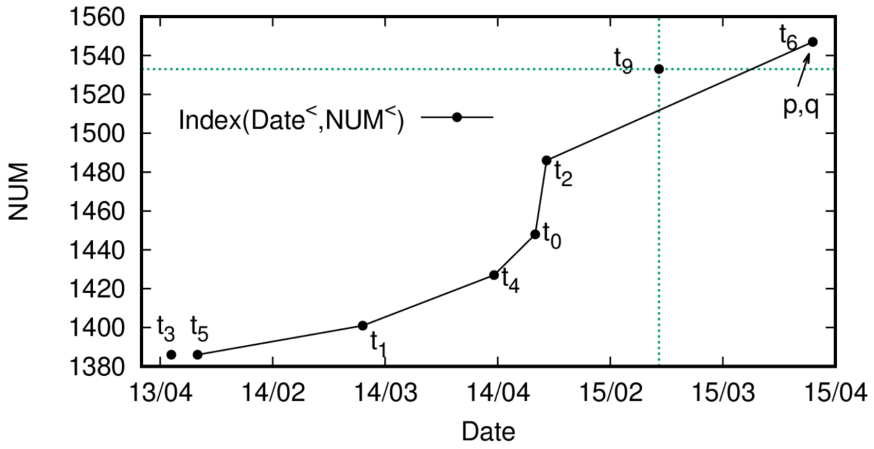


图 5.6: 元组 t_9 以及 D_1

示例 5-4: 图5.4展示了$\mathsf{Index}(\mathsf{Date}^<, \mathsf{NUM}^<)$ 的 Sorted_1 的跳表结构。下面说明单独插入元组 t_9 时，使用算法反转Fetch 来检测可能的矛盾同时进行更新索引。图5.6 展示了将元组 t_9 和索引节点的位置关系。考虑Sorted_1 ：

确定位置 p：满足 $Date^<(\mathsf{Sorted}_1[p+1], t_9)$ 和 $Date^{\geq}(\mathsf{Sorted}_1[p], t_9)$。首先从 Sorted_1 中的最高层第三层开始查询，发现 $Date^<(t_4, t_9)$，需要在下一层继续查找；在第二层中，发现有 $Date^<(t_2, t_9)$，需要继续向下走到第一层；根据 $Date^<(t_2, t_9)$ 以及 $Date^{\geq}(t_6, t_9)$，得到位置 $p = 0$。同样的，可以确定位置 $q = 0$ 使得 $NUM^<(\mathsf{Sorted}_1[q+1], t_9)$ 并且 $NUM^{\geq}(\mathsf{Sorted}_1[q], t_9)$。因为 $p = q$ 所以元组 t_9 能被插入到 $\mathsf{Sorted}_1 = [t_6, t_2, t_0, t_4, t_1, t_5]$ 中。可以看到在这个跳表中没有矛盾发现，这样的结果与之前的反转索引说明一致。同理可以发现，元组 t_9 可以插入到 $\mathsf{Sorted}_2 = [t_3]$ 最前面。因为 $dis(\mathsf{Sorted}_1) = |t_9[NUM] - t_6[NUM]| = 14$，$dis(\mathsf{Sorted}_2) = |t_9[NUM] - t_0[NUM]| = 85$。所以会将元组 t_9 插到Sorted_1 中。

更新索引：更新 Sorted 等同于跳表的插入。当使用一个元组 s 去更新跳表时，元组 s 总是插入到最低层。元组 s 也会依照一个随机参数插入到更高层 (本章第二节)，如图元组 t_9 会插入到第二层。图5.7 展示了将元组 t_9 插入到 Sorted_1 之后的跳表结构。 □

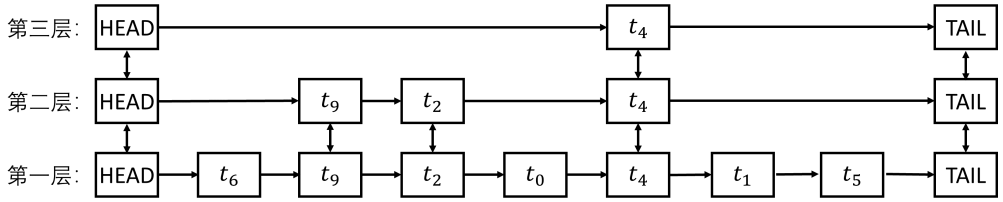


图 5.7: 更新索引后的跳表

特殊说明 1、一个元组 $s \in \triangle D$ 能够插入到 Sorted 中的条件是能够发现位置 $p = q$ 。如果存在多个这样的 Sorted，启发式地选择改变最小的一个；如果没有这样的 Sorted 存在，那么需要新建立一个 Sorted 加到索引中。新建立 Sorted 的操作就会增加索引的秩。第六章实验分析了索引更新前后秩的变化。实验结果表明在大部分数据集上，索引更新对索引秩的影响并不会很大。

2、在下一节会说明如何通过矛盾元组对修复不成立的 PODs ，可以验证在数据集 D 和约束集合 Σ 上建立的索引仍然能够覆盖掉 $\Sigma \oplus \triangle\Sigma$ 的约束。因此更新后的索引仍然可以继续使用在数据集 $D + \triangle D$ 与约束集 $\Sigma \oplus \triangle\Sigma$ 的动态的 POD 发现。对连续变化的数据集不需要重新构建索引。如果对于多次小规模的数据集更新时，不需要每次更新都重新计算索引。这样的性质说明了本文算法比对比算法IEJoin 实用性更高。

第 4 节　POD 的扩展

增量的 POD 发现整体流程：首先通过选择索引算法ChooseIndex 选择需要构建的索引集合$Ind(\Sigma)$ ；再通过算法OptIndex 构建最优索引；然后通过算法Fetch 发现增量数据 $\triangle D$ 带来的矛盾；最后通过算法ExtendPOD 利用矛盾的元组对和原始的约束集合 Σ 计算出 $\triangle\Sigma$。

因为选择索引和构建索引为预处理步骤，所以不纳入整体算法之中。增量发现 PODs 算法 IncPOD 输入为数据集 D 成立的约束集合 Σ、数据集 D、增量数据集 $\triangle D$、属性集合 R，以及建立在数据集上的索引$Ind(\Sigma)$ 。算法IncPOD 目的是发现一个 Σ' ：数据集 $D + \triangle D$ 上所有最小有效的 PODs 集合。增量约束发现方法一般是计算出变化的约束集合 $\triangle\Sigma$，使得 $\Sigma' = \Sigma \oplus \triangle\Sigma$。变化的约束集合 $\triangle\Sigma = \triangle\Sigma^+ \cup \triangle\Sigma^-$，其中 $\triangle\Sigma^+$ 包含了所有需要增加到 Σ 的新 PODs 集合，而 $\triangle\Sigma^-$ 为所有需要被移除的不成立 PODs 。

Algorithm 5: IncPOD

Input: a complete set Σ of minimal and valid PODs on D, a set $\triangle D$ of tuple insertions, attribute set R, and the set $Ind(\Sigma)$ of indexes for covering PODs in Σ

Output: a complete set Σ' of minimal valid PODs on $D + \triangle D$. $\Sigma' = \Sigma \oplus \triangle\Sigma$, $\triangle\Sigma = \triangle\Sigma^+ \cup \triangle\Sigma^-$

```
1  find the set T of violating tuple pairs w.r.t. Σ on D + △D, by
     leveraging Ind(Σ);
2  Σ' ← Σ;
3  for each tuple pair(t, s) ∈ T do
4  |   Σ_temp ← Σ';
5  |   for each σ ∈ Σ_temp do
6  |   |   if (t, s) violate σ then
7  |   |   |   Ψ ← ExtendPOD(σ, (t, s), R);
8  |   |   |   Σ' ← Σ'\σ ∪ Ψ;
9  Σ' ← Minimize(Σ', Σ'\Σ);
10 △Σ+ ← Σ' \ Σ; △Σ− ← Σ \ Σ';
```

算法流程 首先通过索引集合 $Ind(\Sigma)$ 来收集可能违背元组对集合。通过算法Fetch发现索引上可能违背元组对集合 T，即该集合中的元组对是数据集 $D + \triangle D$ 中可能违背原始约束集 Σ 的若干个约束。(第 1 行)。这里包含了通过索引结构发现收集违背元组对的时间 (第四章第四节) 以及插入增量数据集 $\triangle D$ 带来的索引更新时间 (本章第三节)。Σ' 首先会初始化为 Σ (第 2 行)，接着对每一个违背元组对集合中的元组对 (t, s)，计算所有约束 $\sigma \in \Sigma'$ (第 3-8 行)。如果违背元组对 (t, s) 违背了 σ，通过算法 ExtendPOD 计算出集合 Ψ：为了修复元组对 (t, s) 带来的矛盾，扩展约束 σ 生成的约束集合 (第 7 行)。Σ' 更新为去掉不成立的约束 σ 并且加入新的 POD 集合 Ψ (第 8 行)。即 σ 被加入集合 $\triangle\Sigma^-$，同时 Ψ 加入 $\triangle\Sigma^+$。当元组对 (t, s) 遍历完所有的约束后，IncPOD 接着用下一个元组对来更新 Σ'。在之后例子可以看到，在处理一个元组对 (t, s) 时生成的新 PODs 可能会在后续的处理中再次被更改。但是这样的修改不会引起元组对 (t, s) 的新矛盾。即一旦元组对 (t, s) 被解决后，Σ' 再也不会有来自元组对 (t, s) 的矛盾。

Minimize 是为了移除集合 $\Sigma' \setminus \Sigma$ 中非最小的 PODs 集合 (第 9 行)，即移除修复算法生成的新非最小 PODs 。要注意到在原始约束集合 Σ 的在数据集 $D + \triangle D$ 上仍然是最小的 POD 。检查约束 $\sigma = \mathcal{X} \hookrightarrow B^{op'}$ 的最小性 (第三章)。即考虑在约束集合 Σ 中是否存在 PODs $\mathcal{X}' \hookrightarrow B^{op''}$，其中 $|\mathcal{X}'| \leq |\mathcal{X}|$ 以及 $op' \in im(op'')$。Σ' 是数据集 $D + \triangle D$ 上最小有效完备的 PODs 集合，以及 $\triangle\Sigma^+$ 和 $\triangle\Sigma^-$ 分别是集合 Σ 与 Σ' 的差集，Σ' 与 Σ 的差集 (第 10 行)。

Algorithm 6: ExtendPOD

Input: a violating tuple pair(t, s) *w.r.t.* $\sigma = \mathcal{X} \hookrightarrow B^{op'}$, attribute set R

Output: the set Ψ of PODs

1 $\Psi \leftarrow \{\}$;
2 **for** *each* $x^{op} \in \mathcal{X}$ **do**
3 **if** $op \in \{\leq, \geq\}$ **then**
4 $\mathcal{X}' \leftarrow$ replace $x^{\leq}$ (resp. $x^{\geq}$) by $x^{<}$ or $x^{=}$ (resp. $x^{>}$ or $x^{=}$) in $\mathcal{X}$, if the violation incurred by t, s is resolved;
5 $\Psi \leftarrow \Psi \cup \{\mathcal{X}' \hookrightarrow B^{op'}\}$;
6 **if** $op' \in \{=, <\}$ **then**
7 $op'' \leftarrow$ replace $op' =$ (or $<$) by $\leq$, if the violation incurred by t, s is resolved;
8 $\Psi \leftarrow \Psi \cup \{\mathcal{X} \hookrightarrow B^{op''}\}$;
9 **for** *attribute* $A \in R \backslash (X \cup \{B\})$ **do**
10 **for** *each* A^{op} *such that* $A^{\overline{op}}(t, s)$ **do**
11 $\Psi \leftarrow \Psi \cup \{\mathcal{X} \cup \{A^{op}\} \hookrightarrow B^{op'}\}$

算法 ExtendPOD 是算法IncPOD 的核心步骤，主要是将在约束集 $\triangle\Sigma^{-}$ 中不成立的 PODs 扩展成为成立的 PODs 。ExtendPOD 生成新 PODs 集合 Ψ 是通过扩展所有的可能性。具体而言，ExtendPOD 会做以下扩展 (1) 处理 LHS 的属性：如果运算符为 $\{\leq, \geq\}$，则增强运算符使得约束集合成立 (第 2-5 行)；(2) 泛化 RHS 的条件：当右侧属性的运算符为 $\{=, <\}$，则变运算符为 $\leq$(第 6-8 行)；或者 (3) 在约束的 LHS 增加更多的带符号属性 (第 9-11 行)。可以很明显地看出约束集 σ 逻辑蕴含任何 $\sigma' \in \Psi$。因此当使用 Σ' 中的 Ψ 中的约束替换掉 σ，是不会引发新的矛盾 (算法IncPOD 的第 8 行)。

示例 5-5: 比如对 $\{\text{TXA}^{>}, \text{SAL}^{<}\} \hookrightarrow \{\text{RATE}^{<}\}$,违背元组对 (t_8, t_3): $TXA^{>}(t_8, t_3)$, $SAL^{<}(t_8, t_3)$ 以及 $RATE^{=}(t_8, t_3)$。ExtendPOD 可以增加左侧的属性 (算法 9-11 行)：扩展为 $\{\text{ST}^{=}, \text{TXA}^{>}, \text{SAL}^{<}\} \hookrightarrow \{\text{RATE}^{<}\}$ 或者 $\{\text{NUM}^{<}, \text{TXA}^{>}, \text{SAL}^{<}\} \hookrightarrow \{\text{RATE}^{<}\}$，又或者将其 RHS 的符号更改 (算法 6-8 行) 扩展为：$\{\text{TXA}^{>}, \text{SAL}^{<}\} \hookrightarrow \{\text{RATE}^{\leq}\}$。

考虑 $\{\text{NUM}^{<}, \text{TXA}^{>}, \text{SAL}^{<}\} \hookrightarrow \{\text{RATE}^{<}\}$ 和违背元组对 (t_7, t_1)，其中 $NUM^{<}(t_7, t_1)$，$TXA^{>}(t_7, t_1)$，$SAL^{<}(t_7, t_1)$，$RATE^{\geq}(t_7, t_1)$。即 (t_7, t_1) 仍然违背了 $\{\text{NUM}^{<}, \text{TXA}^{>}, \text{SAL}^{<}\} \hookrightarrow \{\text{RATE}^{<}\}$。为了解决这样的矛盾，ExtendPOD 可能会继续增加新的属性，扩展为 $\{\text{ST}^{=}, \text{NUM}^{<}, \text{TXA}^{>}, \text{SAL}^{<}\} \hookrightarrow \{\text{RATE}^{<}\}$。然而因为 $\{\text{ST}^{=}, \text{TXA}^{>}, \text{SAL}^{<}\} \hookrightarrow \{\text{RATE}^{<}\}$ 是有效的 POD 。那么 $\{\text{ST}^{=}, \text{NUM}^{<}, \text{TXA}^{>}, \text{SAL}^{<}\} \hookrightarrow \{\text{RATE}^{<}\}$ 不是最小的 POD ，所以在Minimize 步骤过程会被移除 (算法IncPOD 的第 9 行)。 □

命题 4: IncPOD 能够发现数据集 $D + \triangle D$ 中最小有效 PODs 完备集合。 □

证明: (1) 有效性。如果元组对 (t, s) 和约束集合 σ 中的若干约束有矛盾的时候，那么算法 ExtendPOD 会基于约束集 Σ 扩展出新 PODs 集合 Ψ。ExtendPOD 不会引发集合 T 中新的矛盾，即在数据集 $D + \triangle D$ 中发现的与 Σ 违背的元组对。特别是，可以看到任何元组对 (t', s') 违背了集合 Ψ 中的若干 PODs，那么 (t', s') 也会违背集合 σ，所以也应该在集合 T 之中。注意到在这样的情况下，(t', s') 不可能在处理元组对 (t, s) 前被算法 IncPOD 处理，要不然，约束 σ 在处理 (t', s') 的时候会从 Σ' 中移除。所以当元组对集合 T 中的违背元组对被处理完以后，所有在约束集合 Σ' 中的 PODs 都是在数据集 $D + \triangle D$ 上有效的。

(2) 最小性。结果集中约束的最小性是由在算法IncPOD 中的第 9 行 Minimize 的处理来保证。Minimize 会去除 $\triangle\Sigma$ 中不是最小的约束。

(3) 完备性。假设一个 POD σ 在数据集 $D + \triangle D$ 上是最小有效的，但是并不在约束集合 Σ'。那么 σ 在数据集 $D + \triangle D$ 上就不是最小 POD 。(a) 如果 σ 在数据集 D 是最小的 POD，那么 σ 是在集合 Σ 中，因为 σ 在数据集 D 上为有效并且最小的 POD 。集合 Σ 的定义为数据集 D 上的最小有效 PODs 的完备集合。这样就和约束 σ 不在 Σ'=$\Sigma \oplus \triangle\Sigma$ 的假设矛盾。因为约束 σ 在数据集 $D + \triangle D$ 中有效，所以 σ 很显然不在 Σ^- 中。(b) 如果 σ 在数据集 D 上不是最小的 POD，那么一定存在一个 σ' 在数据集 D 上有效并且逻辑蕴含了 σ。很显然如果 σ' 仍然在数据集 $D + \triangle D$ 上是成立，那么 σ 不会是最小的。现在根据假设可知，σ' 在数据集 $D + \triangle D$ 上不成立。ExtendPOD 在数据集 $D + \triangle D$ 上基于 σ' 扩展 PODs 时会考虑到所有可能的扩展方法。如果 σ 并不在 Σ' 集合中。那么一定存在有效约束集合 Σ' 中的 POD 逻辑蕴含了 σ，因此约束 σ 在数据集 $D + \triangle D$ 上不是最小的 POD 。 □

复杂度分析 实验结果表明算法IncPOD 比全量的 POD 发现算法明显有效，因为是算法IncPOD 在数据集 $D + \triangle D$ 上基于集合 Σ 发现不成立的 PODs 在扩展得出新 PODs，可以减少原始数据之间的比较。复杂度主要有：

(1) 约束集合 Σ 在数据集 $D + \triangle D$ 的违背元组对集合 T 可以通过索引集合$Ind(\Sigma)$ 快速发现。发现矛盾时间取决于增量数据集 $|\triangle D|$ 的遍历时间。每一个新插入的元组需要 $log(|D|)$ 查找矛盾时间与矛盾元组数目的收集时间。

(2) 每一个矛盾元组对集合 T 上的元组只会被处理一次，所以计算 $\triangle\Sigma$ 的时间是线性于矛盾元组对集合的大小 $|T|$。

(3) 扩展 PODs 每次会尽可能增加不同属性或者改变 LHS 、RHS 的运算符，所以ExtendPOD 扩展单个 POD 的复杂度与属性个数 $|R|$ 成正比。

第 5 节　本章小结

本章第一节主要讲了选择索引的算法ChooseIndex：利用不同索引去覆盖原始约束集 Σ。第二节详细说明选取跳表结构表示的优点，并对跳表结构做了详细地查询和插入的流程说明以及复杂度说明。第三节使用示例说明了索引结构上的更新流程。第四节介绍了整个 PODs 的扩展算法IncPOD。

第六章　实验分析

第 1 节　实验配置

机器参数 实验中使用了一台配置为 14 核 Intel Xeon CPU, 64GB 的内存的 PC 服务器。实验中为了更准确地观测结果，实验数据都是五次实验取的平均值。

数据集 本文使用了不同的真实数据集和人工数据集做实验。具体如下：

(1) 真实数据集：SPS 数据集是不同股票在一段时间内的最高最低价格等信息数据集 (http://pages. swcp.com/stocks/)。FLI 是美国的飞机的编号起始地目的地等信息数据集 (www.transtats.bts.gov)。LETTER 是 26 个字母的不同特征数据集 (https://archive.ics.uci.edu/ml/datasets)。NCV 是南美洲的投票结果数据集 (https://ncsbe.gov)。STR 是一个关于蛋白质、核酸和复杂组合体的三维形状的数据集 (http://www.rcsb.org)。

(2) 人工数据集 FDR15 和FDR30 是两个人工数据集。(http://metanome.de)。

表格6.1详细说明了各个数据集的属性个数 (|R|) 和数据集大小 (|D|)。

表 6.1: 数据集及实验结果

Data	$\|R\|$	$\|D\|$	$\|\triangle D\|$	IncPOD	Hydra∗	Finder∗	$\|Ind(\Sigma)\|$	$\|\Sigma\|$	$\|\triangle\Sigma^-\|$	$\|\triangle\Sigma^+\|$
SPS	7	90K	27K	**0.9s**	8.57s	104.5s	5	20	4	6
FLI	17	300K	60K	**67s**	461s	1927s	25	1,481	48	84
STR	5	450K	125K	**9s**	109s	2376.1s	11	21	0	0
LETTER	12	15K	4.5K	**175s**	1538s	4912s	23	8,172	661	58
NCV	18	300K	100K	**76s**	7601s	3188s	13	1,134	17	19
FDR15	15	200K	50K	**14s**	65s	1521s	13	185	3	37
FDR30	30	200K	50K	**48s**	265s	1791s	31	1,152	6	695

表 6.2: 索引空间与时间

Data	$\|R\|$	$\|D\|$	数据集内存大小	索引内存大小	索引个数	索引构造时间
SPS	7	90K	28MB	42MB	5	1.44s
FLI	17	300K	198MB	501MB	25	61.18s
STR	5	450K	124MB	650MB	11	20.55s
LETTER	12	15K	2MB	8MB	23	2.68s
NCV	18	300K	355MB	635MB	13	17.87s
FDR15	15	200K	167MB	263MB	13	9.636s
FDR30	30	200K	345MB	580MB	31	22.80s

算法实现 所有的算法都是用 java 实现的。具体实现的算法如下：

(1) 第四章第二节的构造最优索引算法OptIndex 以及第五章第一节的选择原始约束中的索引集合算法ChooseIndex 都是算法IncPOD 的一个预处理。

(2) IncPOD 是增量 POD 的发现算法：在 (1) 之后，利用索引集合收集矛盾元组对集合 T；收集矛盾元组对同时更新索引；以及通过矛盾元组对扩展已有的 PODs 。其中算法Fetch 通过索引发现增量数据造成的矛盾元组对 (第四章第四节)；索引在发现的同时进行更新的流程 (第五章第三节)；最后算法ExtendPOD 通过约束集合 Σ 和矛盾元组对集合 T 进行修复约束 (第五章第四节)。

(3) Hydra$*$ 是 POD 发现的全量算法，是在数据集上直接发现 POD 。正如第三章第二节中所提，本文是通过 Hydra 的思想更改成Hydra$*$ 。Hydra 是发现 DC 的一个优良算法。每一个 POD 都可以转为相应的 DC 形式。例如这样的 POD $\{\mathsf{A}^{>},\mathsf{B}^{>}\}\hookrightarrow\{\mathsf{C}^{>}\}$ 和这样的 DC: $\forall t,s,\neg(t_A > s_A \wedge t_B > s_B \wedge t_C \leqslant s_C)$ 等价。因为Hydra 算法已有源码，所以本文的对比算法Hydra$*$ 是直接在源码上进行相应的更改所得。(Hydra 源码：www.metanome.de)。Finder$*$ 也是 POD 的全量发现算法，是通过DCFinder 的源码进行更改。DCFinder 是发现近似和普通 DC 的算法，所以也可以进行修改后发现 POD 。

(4) IEJoin 在数据集变动的时候快速发现不等式连接查询的算法。在数据集 D 上面检测一个 PODs 的矛盾也可以转换为查找数据集 D 与自身在不等的条件下的结果。比如查找 POD 的矛盾：$\{\mathsf{A}^{>},\mathsf{B}^{>}\}\hookrightarrow\{\mathsf{C}^{>}\}$ 等同于 SQL 语句:

SELECT $r.id$, $s.id$

FROM D r, D s

WHERE $r.A > s.A$ AND $r.B > s.B$ AND $r.C \leq s.C$

本文中的索引查找出的矛盾相当于两个不等式谓词的连接，比如使用索引Index($\mathsf{A}^{>},\mathsf{B}^{>}$) 发现矛盾等同于：

WHERE $r.A > s.A$ AND $r.B > s.B$

参数设置 从三个参数来考察算法和对比算法的性能。(1) $|D|$: 数据集元组数目; (2) $|\triangle D|$: 增量元组数目；(3) $|R|$: 属性个数。这三个参数的变动方法：采取简单的随机不放回抽样变动原始数据集和增量元组数目；变换数据属性个数时，先不放回地随机抽样数据集的属性，再将数据在被选中的属性上的值作为一个新的数据集。为了减少随机抽样带来的影响，每组实验均是做了 5 次后取平均值。

度量指标 首先通过全量发现算法Hydra$*$ 计算了数据集 D 中包含的 PODs 集合 Σ，然后通过原始数据集 D 以及原有约束集合 Σ 选择并建立索引。通过算法IncPOD 在建立的索引集合发现矛盾然后扩展 POD 集合，使得扩展出的集合在数据集 $D + \triangle D$ 均成立。IncPOD 的正确性是通过和全量算法Hydra$*$ 在数据集 $D + \triangle D$ 上得到的结果相等得知。IncPOD 的算法时间是包含了索引访问时间、索引更新时间、收集矛盾结果集的时间、扩展相应的 PODs 即计算 $\triangle\Sigma$ 的时间。Hydra$*$ 的运行时间是发现数据集 $D + \triangle D$ 中的所有 PODs 时间。

第 2 节 实验结果

实验 1：IncPOD 与 Hydra∗ 、Finder∗ 的比较 表格6.1 展示了不同数据集上面 IncPOD 和 Hydra∗ 的运行时间 (时间单位：秒)。可以很清晰地看到算法 IncPOD 在所有的数据集上的运行时间都是明显地比算法 Hydra∗ 快，最大的差别高达两个数量级。表格6.1 也展示了在集合 Σ，$\triangle\Sigma^-$ 和 $\triangle\Sigma^+$ 中 PODs 的数目，分别通过 $|\Sigma|$，$|\triangle\Sigma^-|$ 以及 $|\triangle\Sigma^+|$ 来表示。可以观察到即使在 $\triangle\Sigma = \triangle\Sigma^+ \cup \triangle\Sigma^-$ 的数目比较多的时候, 比如在数据集 LETTER 和 FDR30 上, IncPOD 仍然比算法 Hydra∗ 、Finder∗ 的运行时间明显快很多。

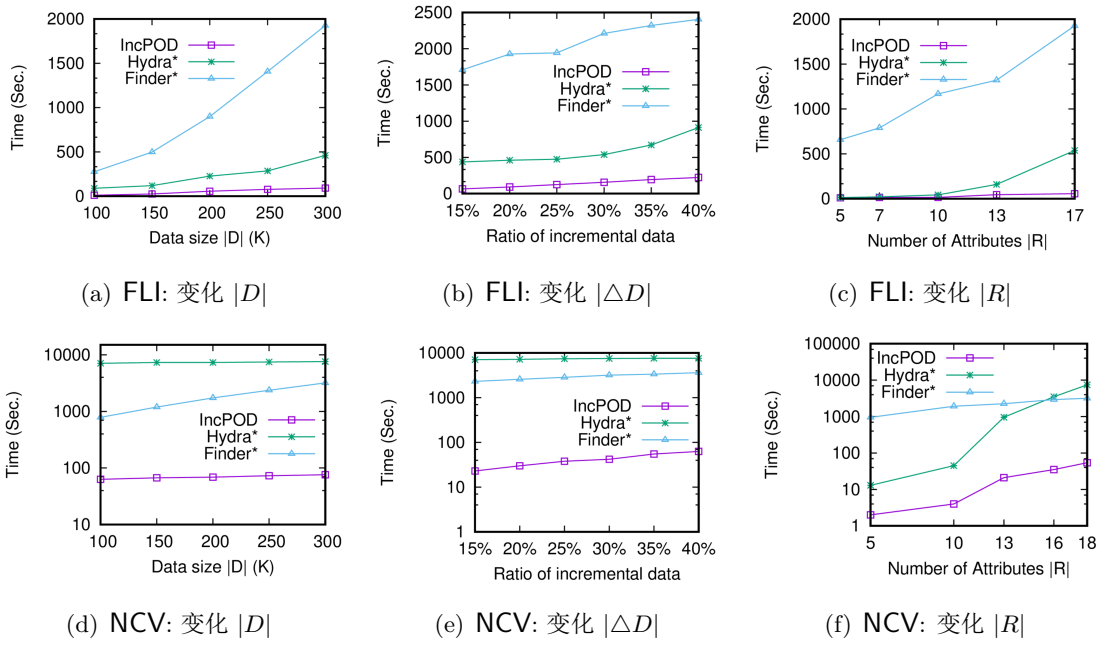
利用在每一组实验中首先设置缺省值；然后变动一个参数，固定其余参数；观察三个算法的运行时间变化来观察算法IncPOD 和算法Hydra∗ 、Finder∗ 与数据集 $|D|$ 的大小、增量数据集大小 $|\triangle D|$、数据集的属性个数 $|R|$ 的关系。

(1) 在数据集FLI 上，设置 $|D| = 300K$，$\frac{|\triangle D|}{|D|} = 20\%$，$|R| = 17$ 为缺省值。

变化数据集大小 $|D|$ 实验结果图6.1(a) 是将数据集大小 $|D|$ 从 100K 变换到 300K (对应的 $|\triangle D|$ 由 20K 变化到 60K) 三个算法的时间变化。算法Hydra∗ 在数据集的大小变化时表现良好，这样的结果和 [33] 的结论是相吻合的，算法Finder∗ 对数据集大小的变化有着明显的增加。但是IncPOD 的运行时间对数据集的大小变化更加不敏感。在数据集大小 $|D|$= 300K 时，IncPOD 仅仅需要 67 秒，Hydra∗ 需要 461 秒，Finder∗ 需要 1927 秒。

变化增量数据集大小 $|\triangle D|$ 实验结果图6.1(b) 展示了将增量数据集大小 $|\triangle D|$ 从 45K 变化到 120K (即增量数据与原始数据大小的比例 $|\triangle D|$ to $|D|$ 从 15% 增加到 40%) 三个算法的结果。可以看到IncPOD 很明显的比Hydra∗ 、Finder∗ 的运行时间少，而且对增量数据大小的变化也不敏感。当增量数据大小增加时，总的运行时间只是从 65 秒增加到 223 秒。

变化属性个数 $|R|$ 实验结果图6.1(c) 是将数据集的属性个数从 5 变化到 17 的三个算法的运行时间对比。可以看到算法Hydra∗ 、Finder∗ 随着属性个数的增加，运行时间指数级地增长。属性个数的变化对算法Hydra∗ 、Finder∗ 的影响远大于IncPOD。根据第三章描述可知，这是因为属性大小变化时，能够组成的谓词空间大小线性增长。Hydra∗ 、Finder∗ 的理论时间是谓词空间的指数倍。比较之下，IncPOD 在属性个数变化时的时间增长就非常平稳。当属性个数从 5 增长到 17 时，Hydra∗ 的运行时间增长了 40 多倍，而IncPOD 的运行时间只增长了 5 倍。可以发现 IncPOD 的运行时间取决于覆盖所有 PODs 的索引个数即$Ind(\Sigma)$ 。表格6.1中展示了索引个数的结果，使用 $|Ind(\Sigma)|$ 表示索引集合中索引个数。从表格中可以看到大多数情况下索引个数 $|Ind(\Sigma)|$ 是远小于原始有效约束个数 $|\Sigma|$。这也是算法IncPOD 在变化属性个数 R 时比算法 Hydra∗ 表现好的原因。在后续的实验 3 中，会进一步对数据集中索引个数 $|Ind(\Sigma)|$、变化数据集大小 $|D|$ 引起索引个数的变化、索引秩以及索引更新后秩的变化进行探讨。



(a) FLI: 变化 $|D|$　(b) FLI: 变化 $|\triangle D|$　(c) FLI: 变化 $|R|$

(d) NCV: 变化 $|D|$　(e) NCV: 变化 $|\triangle D|$　(f) NCV: 变化 $|R|$

图 6.1: IncPOD 与 Hydra∗ 、Finder∗ 的时间

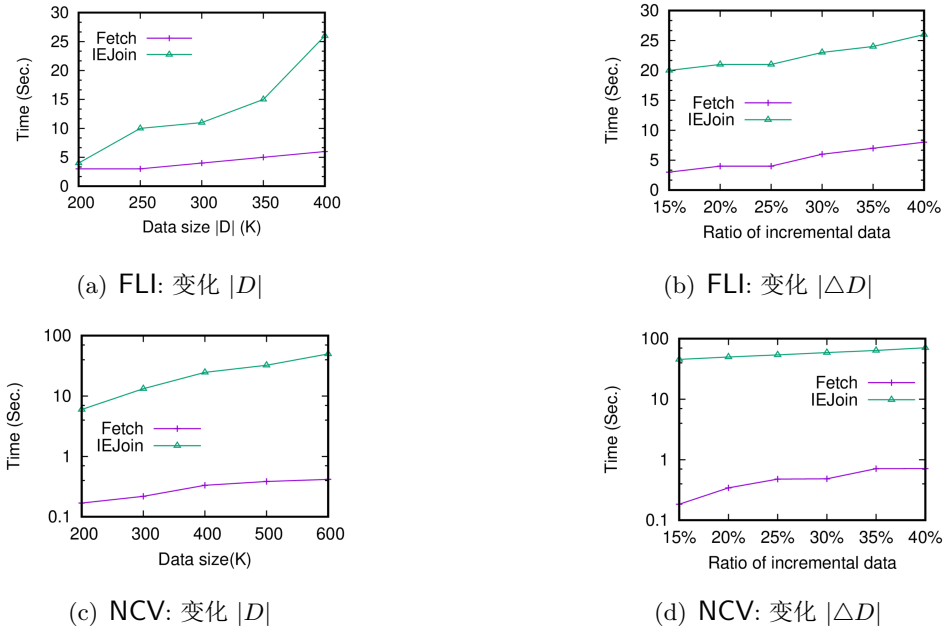
(2) 同样的，选取数据集NCV 进行实验，设置数据集大小 $|D| = 300\text{K}$, 增量数据大小 $|\triangle D| = 100K$ 以及属性个数 $|R| = 18$ 为缺省值。

变化数据集大小 $|D|$ 这里与数据集FLI 实验不同在于上面的实验中是保持了增量数据与原始数据的比例不变，而这个实验探讨增量数据大小不变。图6.1(d)展示了将数据集大小 $|D|$ 从 100K 增长到 300K 的运行时间结果。可以观察到和数据集FLI 相同，算法IncPOD 比全量算法都快数量级倍。算法Finder∗ 在数据集NCV 上的运行速度比Hydra∗ 更快，而且明显地对数据集大小敏感。

变化增量数据集大小 $|\triangle D|$ 图6.1(e) 展示了变化增量数据大小，即将增量数据与原始数据大小的比例 $\frac{|\triangle D|}{|D|}$ 从 15% 增加到 40% 。可以观察到增量数据集对三个算法的运行时间影响都不如原始数据集和属性变化带来的影响大。

变化属性个数 $|R|$ 图6.1(f) 展示了将属性个数 $|R|$ 从 5 增长到 18 的运行时间对比。这个实验中使用了对数坐标，因为算法 IncPOD 的运行时间基本上比全量算法 Hydra∗ 的快两个数量级。可以发现在数据集NCV 的大部分属性上，元组的取不同值的个数更多。这就导致算法 Hydra∗ 、Finder∗ 在数据集NCV 上处理不相等的符号时所花的时间远大于数据集 FLI 。对应的，这样的变化对增量算法没有太大的影响，所以在整体时间上增量算法IncPOD 的速度会远小于全量算法Hydra∗ 。相比较之下，算法Hydra∗ 对属性个数的敏感度明显比Finder∗ 更大。在发现 DC 的实验中也体现了相同的特性：DCFinder 与数据集 D 的大小密切相关，Hydra 与属性个数更密切 [33, 34]。

实验 2: Fetch 与 IEJoin 的比较 为了比较索引带来的效率提升，选取对比算法IEJoin 在不同数据上进行比较。每一次都随机选择一组属性对 A, B 和符号 op_1, op_2，然后比较Fetch、IEJoin 收集集合$T_{A^{op_1}B^{op_2}}$ 和 $\overline{T}_{A^{op_1}B^{op_2}}$ 的时间。



(a) FLI: 变化 $|D|$　(b) FLI: 变化 $|\triangle D|$

(c) NCV: 变化 $|D|$　(d) NCV: 变化 $|\triangle D|$

图 6.2: Fetch 与 IEJoin 时间

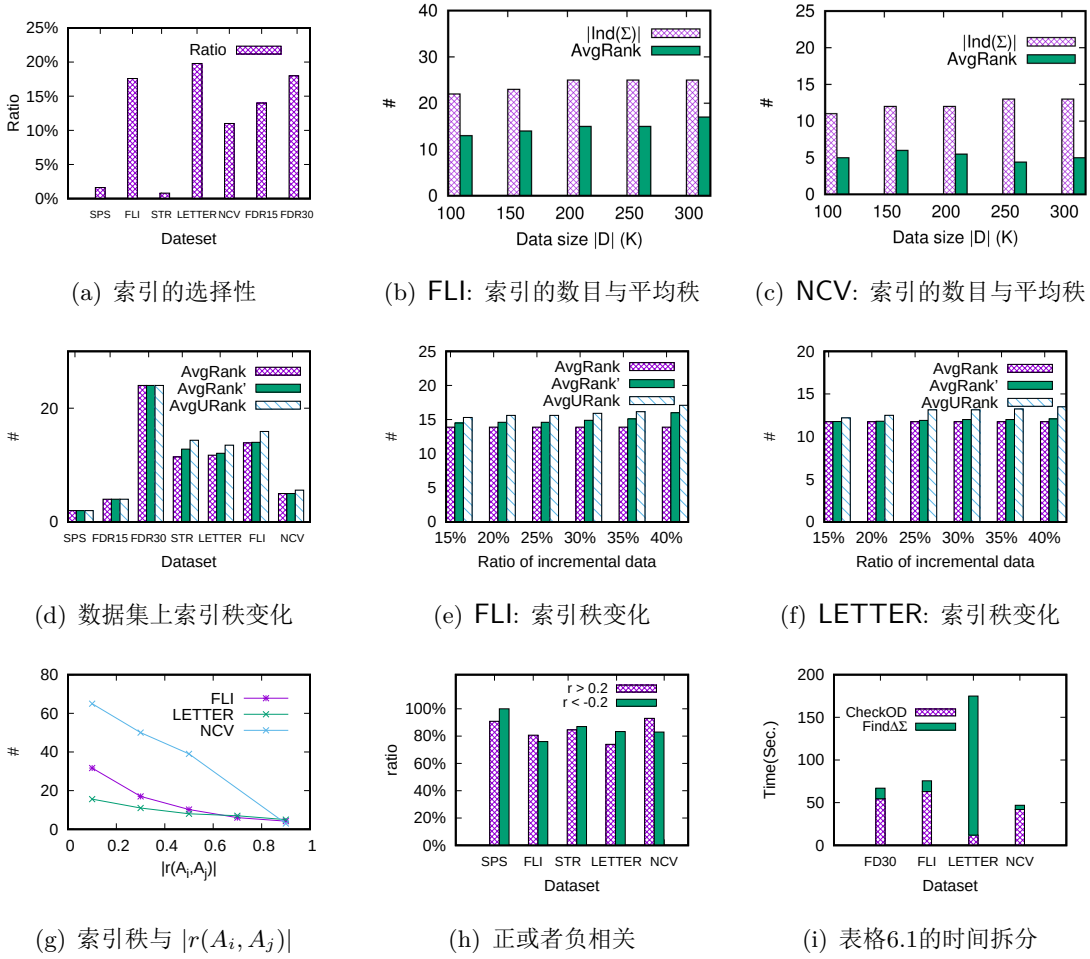
(1) 在数据集 FLI 上，设置 $|D| = 300K$, $\frac{|\triangle D|}{|D|} = 20\%$ 为缺省值。

改变数据集大小 $|D|$ 图6.2(a) 是将数据集大小 $|D|$ 从 200K 增加到 400K 两个算法的时间变化。算法Fetch 在数据集变化时，时间的增长速度比 IEJoin 小。算法Fetch 的运行时间从 3.3 秒增长到 6.4 秒，而IEJoin 的运行时间从 4 秒增长到了 27 秒。算法Fetch 的增长主要来源于收集矛盾结果集的变大。

改变增量数据大小 $|\triangle D|$ 图6.2(b)是将增量比例 $\frac{|\triangle D|}{|D|}$ 从 15% 增加到 40% 的算法运行时间结果。Fetch 与 $|\triangle D|$ 的增加没有太明显的上升：运行时间由 3.4 秒增加到 8 秒。从结果来看，Fetch 的运行时间基本上和 $|\triangle D|$ 成正比。IEJoin 需要将原始数据集和增量数据一起排序，所以是与 $|D + \triangle D|$ 相关。

(2) 在数据集NCV 上，设置 $|D| = 600$K 以及 $\frac{|\triangle D|}{|D|} = 20\%$ 为缺省值。图6.2(c)为将数据集从 200K 增长至 600K 的运行时间结果。对应的图6.2(d)为将增量比例 $\frac{|\triangle D|}{|D|}$ 由 15% 增加到 40% 的结果。值得注意的是，在数据集NCV 的矛盾结果集大小远小于 FLI 。这就导致了两个算在数据集NCV 总时间比FLI 小。收集结果集是必备时间，而算法 Fetch 比 IEJoin 在其他步骤上花费的时间要少很多。由实验结果可知 Fetch 比 IEJoin 对数据集大小 $|D|$ 的变化时，运行时间增长更慢。当数据集大小 $|D|$ 从 200K 增加到 600K，算法Fetch 的时间由 0.16 秒增加到 0.4 秒。而此时算法IEJoin 由 6 秒增长至 50 秒。即使在增量比例达到 40% 时，即 $\frac{|\triangle D|}{|D|} = 40\%$，算法Fetch 的运行时间仍然比IEJoin 快一个数量级。

实验 3：索引性能分析 本实验通过不同数据测试了索引的选择性以及效率。



(a) 索引的选择性　(b) FLI: 索引的数目与平均秩　(c) NCV: 索引的数目与平均秩

(d) 数据集上索引秩变化　(e) FLI: 索引秩变化　(f) LETTER: 索引秩变化

(g) 索引秩与 $|r(A_i, A_j)|$　(h) 正或者负相关　(i) 表格6.1的时间拆分

图 6.3: 索引的效率，索引数目和索引秩以及索引打分函数

(1) 索引可以快速地发现增量数据集 $w.r.t.$ $\triangle D$ 带来的矛盾元组对集。表格6.2描述了各个数据集上索引所占内存大小和索引构建时间。图6.3(a)将表格6.1中的所有数据集发现的矛盾结果集大小和 $|D|$ 做一个比例。可以看到在所有的数据上面比例都是小于 20%，这就表明了索引至少减少了和 80% 原始数据的比较。在大多数情况下，两个带符号属性的组合就已经能有很好的选择性了。

(2) 实验 1 可以看到算法IncPOD 的运行时间是与索引数目 $|Ind(\Sigma)|$ 有关。$AvgRank$ 表示集合 $Ind(\Sigma)$ 中所有索引秩的平均值，不考虑等值索引Index($A_i^=$) 。表格6.1 给出了数据集上的索引数目 $|Ind(\Sigma)|$。可以发现往往只需要少量的索引就可以覆盖 Σ 中的所有 PODs 。选取数据集FLI 为例，图6.3(b)展示了将属性个数固定为 $|R| = 17$，数据集大小 $|D|$ 从 100K 增长到 300K。覆盖全部约束的索引数目 $|Ind(\Sigma)|$ 只从 22 增加到 25。而索引的平均秩$AvgRank$ 也在范围 [13, 17]。图6.3(c)是选取数据集NCV ，固定属性数目 $|R| = 18$ 改变数据集大小 $|D|$，从 100K 增长 300K。索引个数 $|Ind(\Sigma)|$ 在 [11, 13] 以及平均秩 $AvgRank$ 在区间 [4.4, 6]。观察到 $|Ind(\Sigma)|$、$AvgRank$ 与数据集大小 $|D|$ 没太大关系。

(3) $AvgURank$ 为更新索引之后，在数据集 $D+\triangle D$ 上的平均索引的秩。作为对比，在数据集 $D+\triangle D$ 上通过算法OptIndex 构建最优索引，并记最优索引的秩平均数为$AvgRank'$ 。图6.3(d) 展示了表格6.1中所有数据的索引秩的变化情况。可以看到 $AvgURank$ 总是小幅度增长，控制在 $AvgRank$ 的 115%。而且，$AvgURank$ 和 $AvgRank'$ 的差别也是非常小，$AvgRank'$ 最多比 $AvgURank$ 高出 12%。也可以看到索引秩的变化与增量数据大小 $\triangle D$ 关系不大。通过改变增量数据的比例来测试索引秩与增量数据的关系：在数据集FLI 上，固定 $|R|$=17 和数据集大小 $|D|$=300K，图6.3(e)是变化增量数据集的比例 $\frac{|\triangle D|}{|D|}$ 从 15% 增长到 40% 的结果。$AvgURank$ 缓慢地从 15.5 增长到 17.1，一直是维持在$AvgRank'$ 的 [105%, 107%]。图6.3(f) 是在数据集LETTER 上固定属性个数 $|R|$=12 和数据集大小 $|D|$=15K 的实验结果。随着 $\frac{|\triangle D|}{|D|}$ 从 15% 增长到 40%，$AvgURank$ 从 11.75 增长到了 13.5。$AvgURank$ 一直是保持在$AvgRank'$ 的 [104%,112%]，即使是 $\frac{|\triangle D|}{|D|}$ 高达 40%。可以看到索引更新带来的索引秩增长也是非常小的。

实验 4：打分函数和参数 α 以下实验探讨了索引的打分函数和算法 ChooseIndex 中参数 α 的实际效果。

(1) 在真实数据集FLI 、LETTER 和 NCV: 随机抽样出 100 个属性对；然后在这些属性对上面建立索引；分别计算每一个属性对 A_i, A_j 的相关系数的绝对值，即 $|r(A_i, A_j)|$；基于相关系数，将相关系数的绝对值分为 5 个区间 [0, 0.2], ..., [0.8, 1]；然后计算出相关系数的绝对值在区间上的属性对建立的索引的秩平均值。图6.3(g)是索引秩与相关系数的绝对值实验结果图。可以很清晰地可以看到相关系数绝对值 $|r(A_i, A_j)|$ 越大则拥有越小的秩。

(2) 设置参数一个 α>0，根据相关系数与参数的大小来确定索引符号。如果相关系数 $r(A_i, A_j)>\alpha$，建立索引 Index($A_i^<, A_j^<$) (或者Index($A_i^<, A_j^>$)) 而不是Index($A_i^<, A_j^>$) (或者 Index($A_i^<, A_j^<$))。如果相关系数 $r(A_i, A_j) < -\alpha$，建立索引Index($A_i^<, A_j^>$) (或者 Index($A_i^<, A_j^<$)) 而不是Index($A_i^<, A_j^<$) (或者Index($A_i^<, A_j^>$))。特别地，在每个数据集上可能的属性对 A_i, A_j 上计算满足如下性质的属性对数目：(1) 相关系数 $r(A_i, A_j)>\alpha$，索引Index($A_i^<, A_j^<$) 的秩不大于索引 Index($A_i^<, A_j^>$) 的秩；(2) 相反地，相关系数 $r(A_i, A_j)<\alpha$，索引Index($A_i^<, A_j^<$) 的秩不小于索引 Index($A_i^<, A_j^>$) 的秩。计算出这样属性对与所有索引属性对的比例。图6.3(h) 是数据集上设置了参数 α=0.2 的结果。实验结果表示当参数 α=0.2，根据 $r(A_i, A_j)$ 的符号来选择索引的符号可以在平均 85% (最低 78%) 来降低索引的秩。

实验 5：时间拆分 将算法IncPOD 的时间拆分成 (1) 发现可能违背的元组对和更新索引时间，记为CheckOD (2) 计算 $\triangle\Sigma$，记为Find $\triangle\Sigma$ 。图6.3(i) 展示了表格6.1的时间拆分图。省略时间非常小的数据集。CheckOD 在数据集上面占用了特别多的时间。在数据集LETTER 上，可以明显看到Find $\triangle\Sigma$ 占用了绝大部分时间。如表格6.1所示，数据集LETTER 有非常大的 $|\Sigma|$ 和 $|\triangle\Sigma^-|$，需要更多的时间去计算新成立的 PODs 以及检查新扩展的 PODs 的最小性。

实验结果分析 通过以上的实验可以得知：
(1) 增量的 POD 发现算法是非常高效的。除了数据集NCV，当增量数据比例 $\frac{|\triangle D|}{|D|}$ 为 30%，算法 IncPOD 比全量的 PODs 发现算法Hydra$*$ 运行时间平均快 10 倍。在数据集NCV 上面算法IncPOD 比算法Hydra$*$ 快两个数量级。在数据集FLI，当属性个数 $|R|$ 增长近 3 倍时，算法Hydra$*$ 的运行时间与算法IncPOD 的运行时间的比值分别从 1.17 倍增长至 9.6 倍在数据集FLI 。在数据集NCV 上，这样的比值从 6.5 倍增长至 100 倍。
(2) 索引结构是高效的。在数据集FLI 和NCV 上，算法Fetch 通过索引结构收集矛盾元组对比算法IEJoin 分别快 4 倍和 75 倍。
(3) $|Ind(\Sigma)|$ 和 $AvgRank$ 对数据集大小 $|D|$ 都不是很敏感。当 $|D|$ 变为原来的 3 倍时，$|Ind(\Sigma)|$ 和 $AvgRank$ 增长幅度分别在 $[13\%, 18\%]$ 和 $[20\%, 29\%]$ 以内。索引的更新对索引秩的改变是非常小的。测试数据集中，当增量数据比例 $\frac{|\triangle D|}{|D|}$ 从 15% 增长至 40%，在 $D + \triangle D$ 中的 $AvgURank$ 比数据集 D 上的$AvgRank$ 增加的比例在 [4%,18%]。当增量数据集的比例 $\frac{|\triangle D|}{|D|}$ 从 15% 增长到 40% 时，在数据集 $D + \triangle D$ 中最优索引的平均秩与数据集 D 上的最优索引平均秩增长比例在 $[4\%, 10\%]$。索引更新后的平均秩比最优索引的平均秩增长比例在 $[5\%, 9\%]$。
(5) 属性对的相关系数是一个选择索引属性对和符号很好的评测指标。

第 3 节　本章小结

本章通过实验证明了第四章中索引的性能和效率分析。对第五章索引选择的实际效果做了实验。并将索引的效率与对比算法IEJoin 进行比较以及增量算法与全量发现算法的比较。并对索引的更新带来的影响做了相关实验。

第七章　总结与展望

前两章通过实例说明 PODs 的实际意义并对相关约束做了介绍。第三章给出了 PODs 最小性、蕴含等相关符号定义以及 PODs 增量发现的问题。第三章还说明了全量 PODs 发现算法的复杂度。在第四章、第五章提出了一套索引算法来发现 PODs 在动态数据集上的矛盾以及修复 POD 的算法。第六章使用了真实和人工数据集说明了索引效率。以下几个方面可作为基于本文工作未来可继续开展的研究方向：

1、不同索引查找矛盾是互相独立的，所以可以并行处理不同索引发现矛盾。这样的处理方式可以将索引查找矛盾的时间从所有索引查找矛盾时间之和更改为所有索引中查找矛盾最长时间。因此可减少整体算法时间。

2、否定约束 DC 是 POD 的严格泛化，目前没有增量 DC 的发现的研究。因为 DC 查找矛盾也等同于多个谓词连接查询，可以将本文索引算法、IEJoin 的算法联合使用快速收集数据集与 DC 的矛盾。所以将OptIndex 、IEJoin 的算法联合使用扩展为增量 DC 的发现算法。

3、本文只考虑了增加数据的情况，并没有考虑删除数据时发现最小 POD 的情形。所以可以扩展到删除数据情形下的数据约束。

参考文献

[1] S. Abiteboul, R. Hull, V. Vianu. Foundations of databases. Addison-Wesley[M], 1995.

[2] Fan W, Geerts F. Foundations of data quality management[J]. Synthesis Lectures on Data Management, 2012, 4(5): 1-217.

[3] Raman, Vijayshankar. Foundations and Trends in Databases[M]. Now Publishers Inc. 2007.

[4] Bertossi L . Database Repairing and Consistent Query Answering[M]. Morgan & Claypool Publishers, 2011.

[5] Fan W , Geerts F , Jia X , et al. Conditional functional dependencies for capturing data inconsistencies[J]. ACM Transactions on Database Systems, 2008, 33(2):1-48.

[6] Xu Chu, I.F. Ilyas, P. Papotti. Holistic data cleaning: Putting violations into context[C]// International Conference on Data Engineering (ICDE), 2013.

[7] Seymour, Ginsburg, and, et al. Order dependency in the relational model[J]. Computer Compacts, 1983.

[8] Ginsburg S , Hull R . Sort sets in the relational model[C]// the 2nd ACM SIGACT-SIGMOD symposium. ACM, 1983.

[9] Szlichta J , Godfrey P , Gryz J , et al. Expressiveness and complexity of order dependencies[J]. Proceedings of the VLDB Endowment, 2013, 6(14):1858-1869.

[10] Abedjan Z , Quiane-Ruiz J A , Naumann F . Detecting unique column combinations on dynamic data[J]. International Conference on Data Engineering (ICDE), 2014:1036-1047.

[11] Abedjan, Ziawasch and Schulze, Patrick and Naumann, Felix. DFD: Efficient Functional Dependency Discovery[M].Association for Computing Machinery, 2014.

[12] Wenfei Fan, Floris Geerts, Laks V. S. Lakshmanan et al. Discovering Conditional Functional Dependencies[C]// Proceedings of the 25th International Conference on Data Engineering (ICDE), 2009.

[13] Golab L , Karloff H , Korn F , et al. On generating near-optimal tableaux for conditional functional dependencies[J]. Proceedings of the VLDB Endowment, 2008, 1(1):376-390.

[14] J. Szlichta, P. Godfrey, J. Gryz. Fundamentals of order dependencies[C]// Proceedings of the Vldb Endowment, 5(11):1220-1231,2012.

[15] Szlichta, Jaroslaw, Godfrey, Parke, Golab, Lukasz. Effective and complete discovery of order dependencies via set-based axiomatization[J]. Proceedings of the Vldb Endowment, 10(7):721-732.

[16] Jaroslaw Szlichta, Parke Godfrey, Lukasz Golab, Mehdi Kargar, and Divesh Srivastava. Effective and complete discovery of bidirectional order dependencies via set-based axioms[C]// Proceedings of the Vldb Endowment, 27(4):573–591, 2018.

[17] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. Holistic data cleaning: Putting violations into context[C]// International Conference on Data Engineering (ICDE), pages 458–469, 2013.

[18] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. Discovering denial constraints[C]// Proceedings of the PVLDB, 6(13):1498–1509, 2013.

[19] Philipp Langer and Felix Naumann. Efficient order dependency detection[J]. VLDB J., 25(2):223–241, 2016.

[20] Cristian Consonni, Paolo Sottovia, Alberto Montresor, and Yannis Velegrakis. Discovering order dependencies through order compatibility[C]// Proceedings of the International Conference on Extending Database Technology(EDBT), pages 409–420, 2019.

[21] Yannis Sismanis, Paul Brown, Peter J. Haas, and Berthold Reinwald. Gordian: Efficient and scalable discovery of composite keys[C]// Proceedings of International Conference on Very Large Data Bases(VLDB), 2006.

[22] Arvid Heise, Jorge-Arnulfo Quiané-Ruiz, Ziawasch Abedjan, Anja Jentzsch, and Felix Naumann. Scalable discovery of unique column combinations[C]// Proceedings of the Vldb Endowment, 2013.

[23] Ziawasch Abedjan and Felix Naumann. Advancing the discovery of unique column combinations[C]// Proceedings of the 20th ACM Conference on Information and Knowledge Management In CIKM, 2011.

[24] Yka Huhtala, Juha Karkkainen, Pasi Porkka, and Hannu Toivonen. TANE: an efficient algorithm for discovering functional and approximate dependencies[J]. The Computer Journal., 42(2):100–111, 1999.

[25] WYSS, Catharine, GIANNELLA, et al. FastFDs: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances: Extended abstract[C]// International Conference on Data Warehousing & Knowledge Discovery. Springer-Verlag, 2001.

[26] Yao H , Hamilton H J , Butz C J . FD_Mine: Discovering Functional Dependencies in a Database Using Equivalences[C]// Proceedings of the IEEE International Conference on Data Mining (ICDM), 2002.

[27] Noel Novelli and Rosine Cicchetti. Fun: An efficient algorithm for mining functional and embedded dependencies[C]// Proceedings of the International Conference on Database Theory(ICDT), 2001.

[28] Abedjan Z , Schulze P , Naumann F . DFD: Efficient Functional Dependency Discovery[C]// Proceedings of International Conference on Conference on Information and Knowledge Management(CIKM), 2014.

[29] Stephane Lopes, Jean Marc Petit, and Lotfi Lakhal. Efficient discovery of functional dependencies and armstrong relations[C]// In International Conference on Extending Database Technology(EDBT), 2000.

[30] Thorsten Papenbrock and Felix Naumann. A hybrid approach to functional dependency discovery[C]// In Proceedings of the 2016 International Conference on Management of Data, 2016.

[31] Alexandar Mihaylov, Parke Godfrey, Lukasz Golab, Mehdi Kargar, and Jaroslaw Szlichta. Fastod: Bringing order to data[C]// International Conference on Data Engineering (ICDE), 2018.

[32] Lin Zhu, Xu Sun, Zijing Tan, Kejia Yang, Weidong Yang, Xiangdong Zhou, and Yingjie Tian. Incremental discovery of order dependencies on tuple insertions[C]// In DASFAA, pages 157–174, 2019.

[33] Tobias Bleifuβ, Sebastian Kruse, and Felix Naumann. Efficient denial constraint discovery with hydra[C]// Proceedings of PVLDB, 11(3):311 – 323, 2017.

[34] Eduardo H. M. Pena, Eduardo Cunha de Almeida, and Felix Naumann. Discovery of approximate (and exact) denial constraints[C]// Proceedings of PVLDB, 2019.

[35] David Dewitt, Jeffrey Naughton, and Donovan Schneider. An evaluation of non-equijoin algorithms[C]// Proceedings of International Conference on Very Large Data Bases(VLDB), 2001.

[36] Jost Enderle, Matthias Hampel, and Thomas Seidl. Joining interval data in relational databases[C]// Proceedings of the ACM SIGMOD International Conference on Management of Data, 2004.

[37] Chee Yong Chan and Yannis E. Ioannidis. An efficient bitmap encoding scheme for selection queries[C]// Proceedings of the ACM SIGMOD Record, 28(2), 2000.

[38] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized Search Trees for Database Systems[M]. 1995.

[39] Fan W, Geerts F. Foundations of data quality management[J]. Synthesis Lectures on Data Management, 2012, 4(5): 1-217.

[40] Dallachiesa M, Ebaid A, Eldawy A, et al. NADEEF: a commodity data cleaning system[C]//Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. ACM, 2013: 541-552.

[41] Zuhair Khayyat, William Lucia, Meghna Singh, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiane-Ruiz, Nan Tang, and Panos Kalnis. Lightning fast and space efficient inequality joins[C]// Proceedings of the Vldb Endowment, 8(13):2074–2085, 2015.

[42] William Pugh. Skip lists: A probabilistic alternative to balanced trees[M]. Commun. ACM, 33(6):668–676, 1990.

攻读硕士学位期间发表论文情况和研究成果

1. 2018 5th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/2018 4th IEEE International Conference on Edge Computing and Scalable Cloud EdgeCom(第二作者)
2. 计算机应用与软件（第一作者）

复旦大学
学位论文独创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。论文中除特别标注的内容外，不包含任何其他个人或机构已经发表或撰写过的研究成果。对本研究做出重要贡献的个人和集体，均已在论文中作了明确的声明并表示了谢意。本声明的法律结果由本人承担。

作者签名：_____ 日期：_____

复旦大学
学位论文使用授权声明

本人完全了解复旦大学有关收藏和利用博士、硕士学位论文的规定，即：学校有权收藏、使用并向国家有关部门或机构送交论文的印刷本和电子版本；允许论文被查阅和借阅；学校可以公布论文的全部或部分内容，可以采用影印、缩印或其它复制手段保存论文。涉密学位论文在解密后遵守此规定。

作者签名：_____ 导师签名：_____ 日期：_____