# **数据质量研究报告**

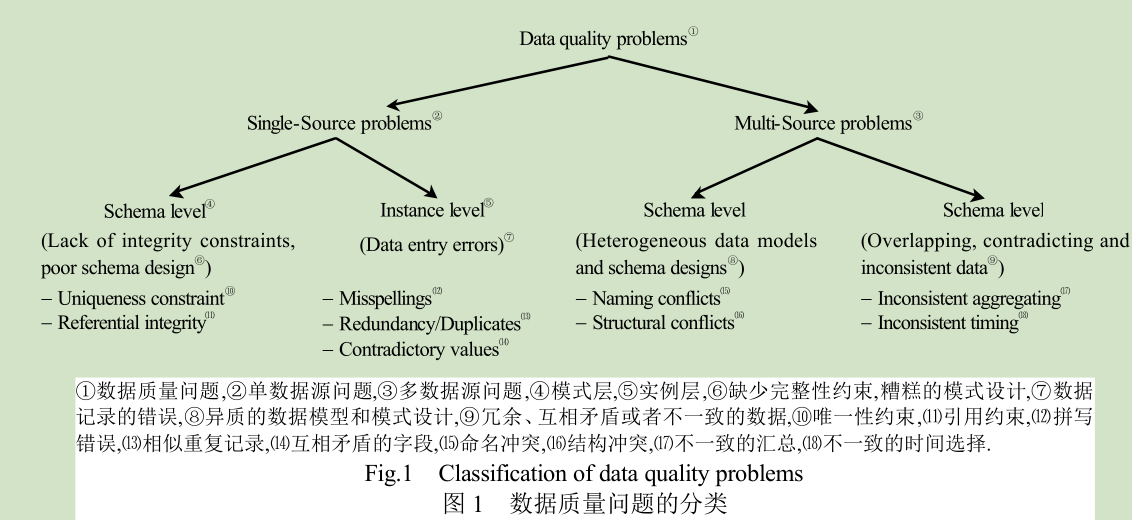
# 一、简介

随着信息处理技术的不断发展，各行各业已建立了很多计算机信息系统，积累了大量的数据。为了使数据能够有效地支持组织的日常运作和决策，要求数据可靠无误，能够准确地反映现实世界的状况。数据是信息的基础，好的数据质量是各种数据分析如OLAP、数据挖掘等有效应用的基本条件。人们常常抱怨 “数据丰富 ,信息贫乏 ”，究其原因 ，一是缺乏有效的数据分析技术，二是数据质量不高，如数据输入错误、不同来源数据引起的不同表示方法，数据间的不一致等，导致现有的数据中存在这样或那样的脏数据。它们主要表现为 : 拼写问题、打印错误、不合法值、空值、不一致值、简写、同一实体的多种表示 ( 重复 )、不遵循引用完整性等。

数据清洗的目的是检测数据中存在的错误和不一致，剔除或者改正它们，以提高数据的质量。

根据处理的是单数据源还是多数据源以及问题出在模式层还是实例层，数据质量问题分为 4 类(如图 1 所示)：单数据源模式层问题、单数据源实例层问题、多数据源模式层问题和多数据源实例层问题。图 1表示了这种分类，并且分别列出了每一类中典型的数据质量问题。

单数据源情形中出现的问题在多数据源的情况下会变得更加严重。图 1 对多数据源没有列出在单数据源情形中就已经出现的问题。模式层次上的问题也会体现在实例层次上。糟糕的模式设计、缺少完整性约束的定义以及多个数据源之间异质的数据模型、命名和结构冲突等，都属于该类问题。可以通过改进模式设计、模式转化和模式集成来解决模式层次上的问题。实例层次上的问题在模式层次上不可见，一些可能的情况有数据拼写错误、无效的数据值、重复记录等。下面我们主要考虑实例层次上的问题。



本文主要针对数据一致性问题进行讨论，讨论了各种算法和框架用于提高数据清洗的质量。

# 专题介绍

## 1.Discovering Conditional Functional Dependencies

### I.简介

条件函数依赖（CFD）相比于标准函数依赖（FD）在对于非一致性数据的检测及修复过程中都有更高的效率。在利用CFD进行检测之前，我们首先要得到它。然而，通过专家来设计CFD往往是不现实的，基于这种需求我们需要一种能够从数据中自动发现CFD的技术。

### II.概念及基础介绍

函数依赖

函数依赖规则，即functional dependency，简称为FD。若，为关系中的两个属性集合，当中的任意两个元组属性值相同时，他们的属性值也相同，称为函数决定或函数依赖于函数，记为。我们用表示箭头左侧的所有属性集合，即，同理用表示。例如，对于某个关系表上的某个函数依赖，表示中任意两个元组，，当， 时，总有。

条件函数依赖

条件函数依赖规则，即conditional functional dependency，简称为CFD。我们将关系上的CFD定义为，为一个标准的FD， 是一个包含和中属性的表格，我们称之为模式表。对于每一个属性以及每一个模式表中的元组，那么是一个的属性范围内固定的属性值“a”或者一个未命名的变量“-”。例如，一个条件函数依赖，模式表如表 1-1 所示，表示R中任意两元组，，当，时，总有；当，时，总有。

表 1-1 一个条件函数依赖的模式表

|  |  |  |
| --- | --- | --- |
| A | B | C |
| 1 | — | — |
| — | 3 | — |

我们可以将传统的FD视作CFD的一个特例，只不过这个特别的CFD中的模式表中只有一个仅含“-”的元组。

### 问题描述

给定一关系表R，我们的目的是找到定义在R上的CFD。为了避免冗余的CFD，我们需要找到CFD集合的正则覆盖。

最小CFD：

CFD中，若A∈X，则该CFD无价值，我们仅考虑有价值的CFDs。

一个常量CFD左简化定义为：中，若任意Y包含于X，r不满足。

一个变量CFD左简化定义为：中，（1）对于X中任意真子集Y，r不满足；（2）r不满足，tp<<tp’。

最小CFD务必要做到非平凡，左简化，故最小CFD均是非冗余的。

频繁CFD：

CFD的支集sup(,r)被定义为满足t[X]<=tp[X] and t[A]<=tp[A]的元组集合。若sup(,r)>=k，则称为k-频繁。在关系r上关于k的正则覆盖CFD集为最小k-频繁CFD集。

问题定义：

给出一个关系表r以及支集阈值k，发现CFD的问题等价于找到r上关于k的CFD正则覆盖集。

### III.主要算法

1. CFDMiner（constant CFD discovery）

**找到形式为的k-频繁最小正则覆盖CFD集合。**

首先介绍几个概念：

开项目集和闭项目集：

一个项目集定义为（X，tp），X为R中的属性集合子集，tp为X的常量模式表。（X，tp）的支集写成supp（X，tp，r），为和tp匹配的元组集合。

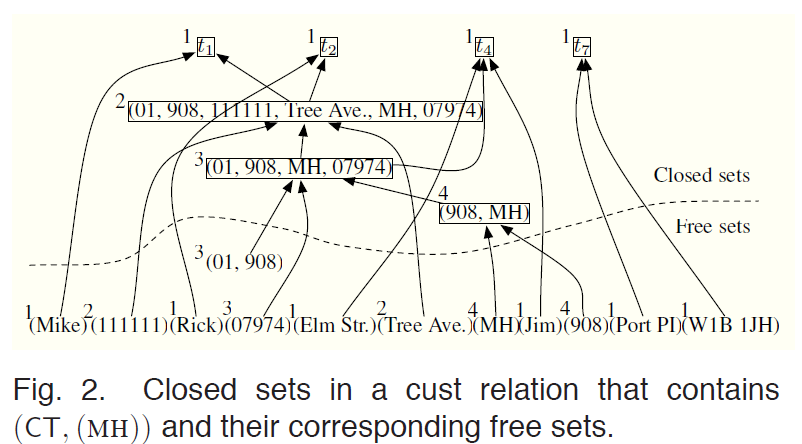
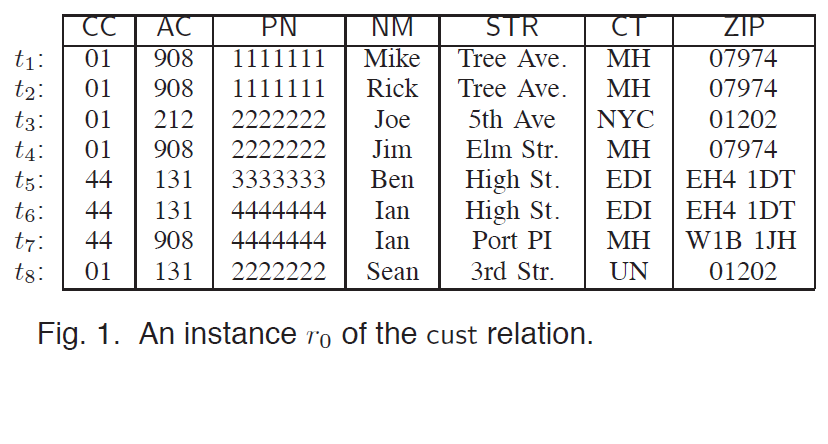
（X，tp）≤（Y，sp）：若Y为X的子集且sp=tp[Y]，成为（Y，sp）比（X，tp）更一般。

（X，tp）<（Y，sp）：若Y为X的真子集且sp=tp[Y]，成为（Y，sp）严格一般于（X，tp）。显然，若（X，tp）≤（Y，sp）则supp(X,tp,r)为supp(Y,sp,r)的子集。

闭项目集：若不存在项目集（Y，sp），使（Y，sp）≤（X，tp）且supp(X,tp,r) = supp(Y,sp,r)则将（X，tp）称为闭集。显然，一个闭集在不减小支持度（support）的情况下不能扩展。

开项目集：若不存在项目集（Y，sp），使（X，tp）≤（Y，sp）且supp(X,tp,r) = supp(Y,sp,r)则将（X，tp）称为开集。显然，一个开集在不增大支持度（support）的情况下不能扩展。

（X，tp）被称为k-频繁，如果| supp(X,tp,r)|>=k，(k>=1)。



**例**：表2展示了表1关系中包含（CT，（MH））的闭集，同样展示了与之对应的开集。为了简化，仅展示了项目集支持度的大小而不展示属性的名称。（[CC,AC,CT,ZIP], (01,908,MH,07974)）是一个支持度为3的闭集；（[zip],(079740)）（[CC,AC], (01,908)）是开集，支持度同样都为3。

K-频繁开闭项目集和k频繁左简化常量CFD的关系：

**命题1**：对于一个关系r和任意k频繁左简化常量CFD，r满足当且仅当：

（1）（X，tp）是k-频繁开集，且不包含（A，a）

（2）clo（X，tp）≤（A，a）

（3）(X,tp)中不包含一个更小的开集（Y，sp）使开集（X，tp）≤（Y，sp），Y为X真子集且clo（Y，tp）≤（A，a）

**例**：（[CC,AA]->CT，（01,908||MH））是一个3频繁常量CFD。实际是从模式表闭集（[CC,AC,CT,ZIP]，(01,908,MH,07974)）中获得的，其中开集（[CC,AC],(01,908)）被作为常量CFD的左侧，不包含（CT,MH）满足条件（1）;clo([CC,AC],(01,908))= （[CC,AC,CT,ZIP]，(01,908,MH,07974)）满足条件（2）；然而，其中包含了一个更小的开集（AC，（908）），这个更小开集的闭集（[AC,CT]，（908，MH））包含了（CT，（MH）），故不满足条件（3），因此不是左简化的。

类似的可以判断（AC->CT，(908||MH)）为4频繁左简化常量CFD。

**CFDMiner算法流程**：

1. 对于每一个k-频繁闭集（X，tp）向哈希表H中加入它的开集（GCGROWTH算法为每个k-频繁闭集返回对应的开集）
2. 对于每一个闭集（X，tp），为其开集（Y，sp）关联集合RHS（Y，sp）=（X\Y，tp[X\Y]）。在此过程中，建立一个按集合大小顺序排列的k-频繁开集的列表L。
3. 对于L中的每一个开集（Y，sp）：
   1. 每一个Y中的真子集Y’，（Y’，sp[Y’]）属于L，用RHS（Y，sp）∩RHS（Y’，sp[Y’]）代替RHS（Y，sp）。
   2. 在所有的子集（Y，sp）都检测完毕后，对于所有的（A，a）属于RHS（Y，sp）CFDMiner将输出k-频繁常量CFD（Y->A,(sp||a)）。
4. CTANE

一个层次化用于发现最小k-频繁CFD的算法。（不限variable and constant）

首先考虑用CTANE挖掘1-频繁最小CFD，之后修改CTANE算法来挖掘k-频繁CFD。

CTANE从独立集合（A，）开始（A∈attr(R),∈dom（A）∪{\_}），而后扩展到更大的属性集合以及数据表中。查看CFDs（X\{A}->A,(sp[X\{A}]||sp[A])）(A∈X)。这样可以确保只有非平凡CFD呗考虑到了。此外，CTANE算法为每个（X，sp）保存一个集合C+（X，sp）来判断CFDs（X\{A}->A,(sp[X\{A}]||sp[A])）是否是最小的。

**剪枝策略**：对于每一个（X，sp），有一个集合C+（X，sp）由(A, cA) ∈ attr(R)×{dom(A)∪{\_}}组成,它满足以下条件：

1.若A∈X，则cA=sp[A];

2.对于所有B∈X，r不满足(X \ {A,B} –>B, (sp[X \ {A,B}] || sp[B]))

3.对于所有B∈X\{A}，r不满足(X \ {A} -> A, (spB|| cA)),spB[C]=sp[C],C≠B，spB[B]=’\_’。

第一个条件避免产生不一致的CFD；第二个条件用于确认CFD左侧不能简化；第三个条件用于确认模式表中的元组是最一般化的。这里一般化的含义和CFDMiner中的意义类似，（Y，sp）称为较（X，tp）更一般化，如果Y是X的子集，同时tp[Y]<<sp。<<表示tp[Y]=sp或tp[Y]=’\_’。

**引理2**：X为attr(R)的子集，sp为X的模式表，A∈C，现假设r满足 = (X \ {A} –> A, (sp[X \{A}] || sp[A]))。那么为最小当且仅当对于所有B∈X，(A, sp[A]) ∈ C+(X \ {B}, sp[X \ {B}])。

**CTANE：**

L**l**是L中（X，sp）大小为l的元素集合即|X|=**l**。现假设L**l**是按顺序排列的，如果X=Y，tp<<sp,那么，（X，sp）在（Y，tp）之前出现。初始化L1={(A, \_) | A∈attr(R)}∪{(A, a1) | a1 ∈πA(r), A∈attr(R)},C+()=L1且**l** =1。之后进行如下步骤直到L**l**非空：

1. 通过L**l**的LHS计算最小CFD的RHS，

即，对于所有的（X，sp）∈L**l**计算C+(X ,sp}=∩B∈XC+(X \ {B}, sp[X \ {B}])

1. 对于每一个（X，sp）∈L**l**，查找有效的CFDs，

即，对于所有A∈X，（A，cA）∈C+（X，sp），进行如下步骤：

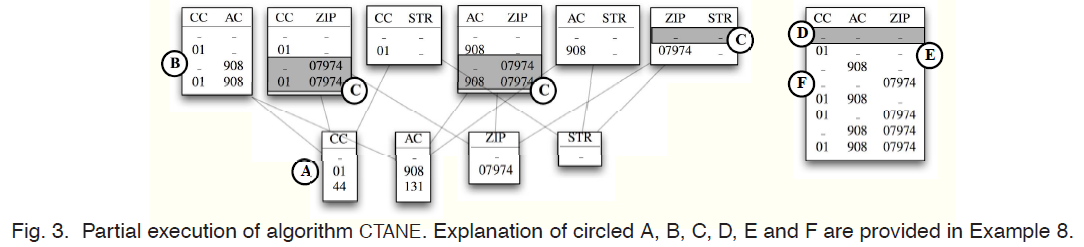
1. = (X \ {A} –> A, (sp[X \ {A}] || cA))，r是否满足
2. 若r是否满足，输出
3. 若r是否满足，那么对于所有（X，up）∈L**l**，up[A]=cA且up[X\{A}]<<sp[X\{A}]，更新C+（X，up），去掉所有B∈attr(R)\X中的(A,cA)和(B,cB)
4. 对L**l**剪枝，去掉L**l**中使C+（X，sp）为空集的（X，sp）集合。
5. 生成L**l+1**，
   1. 初始化L**l+1**为空集
   2. 对于一对不重复的（X，sp）（Y，tp）∈L**l**，且在前**l-1**个属性上匹配：
      1. Z=X∪Y，且up=(sp,tp[Yn]);Yn代表Y中最后一个属性；
      2. 如果投影πZ(r)中存在一个元组满足up，那么用（Z，up）继续进行
      3. 若对于所有A∈Z，(Z \ {A}, up[Z \ {A}])∈L**l**,那么将（Z，up）加到L**l+1**中。
   3. **L= l**+1**；**

此后，扩展算法使适用于k-频繁CFD的查找。

首先，做如下观察。=（X->A,(tp,cA)）为定义在r上的一个CFD，用（Xc，tpc）来表示由（X，tp）的常量部分组成的项集。那么是k频繁的当且仅当supp（Xc，tpc，r）>=k,X不为空集，且|r|>=k。这就是说为了得到其恰当的k，我们需限定元组（X，sp）∈L**l**使其对应的（Xc，tpc）是一个k频繁项集。这个目的可以通过如下两点做到：（1）从L1 = {(A,\_) | A ∈attr(R)} ∪ {(A, a1) | supp(A, a1, r) >=k, A ∈ attr(R)};（2）替换第4步的b(ii)为判断supp（Zc，upc，r）>=k

例：对于图1中的关系表，仅对属性CC，AC，ZIP，STR执行CTANE的算法，限定k>=3,

表3中展示了L的两个层次，第三个层次相当于[CC,AC,ZIP],对于CTANE检查的每一个元素（X,sp）,列出可能的模式同时列出属性集合列表，用‘\_’的数量来排列他们。



（A）初始化L1为单独的一对出现至少k次的属性/属性值，每一个属性出现一个变量‘\_’。在这一步中所有的C+（A，cA）都包含（A，cA）。由于r不满足左侧为空的CFD，故在第二步中C+集合不更新，类似的在第三步中也没有从L1中去掉。

（B）在第4步中，将属性组对并创建了一致模式。注意到对于（CC，AC），由于k=3，故常量44未出现。

（C）图中灰色的部分，第2步找到有效的CFDs：

（ZIP->CC,(07974||\_)）,(ZIP->CC,(07974||01)),(ZIP->AC, (07974||01)), (ZIP->AC, (07974||908)),(STR->ZIP,(\_||\_))

在第二步中C+（[CC,ZIP],(\_,07974)）和C+（[AC,ZIP],(\_,07974)）被更新，去掉（CC，\_）和（AC，\_）

（D） 在第4步现在建立三属性的模式表，这里仅展示（CC，AC，ZIP），第二步中得到（[CC,AC]->ZIP,(\_,\_||\_)）

（E）CTANE在第二步c中更新C+集合，（ZIP，\_）被从C+中移除，这确保了后序产生的CFD仅包含最一般化的LHS模式。

（F） 最后，在步骤1中，模式表的C+集合（\_,\_,07974）。然而，回想C+（[CC,ZIP],(\_,07974)）和C+（[AC,ZIP],(\_,07974)）被更新过了，故（CC，\_）和（AC，\_）都不会出现在C+集合（\_,\_,07974）中，这意味着找到最小CFD只能通过检测（[AC,CC]->ZIP,(\_,\_||07974)）。为得到最小CFD，C+集合减少了可能的RHS。

1. FastCFD

与CTANE广度优先不同的是，FastCFD是深度优先方法，由FastFD扩展而来。

FastCFD首先将找正则覆盖的问题分解为找到由特定RHS属性的CFD组成的CFD正则覆盖。对于每一个attr（R）中的A，FastCFD试图寻找形式为=（Y->A,tp）,Y为attr（R）\{A}的子集，最小，且sup()>=k，用Cover（A，r，k）来表示这个集合。显然，通过∪A∈attr（R）Cover（A，r，k）可以得到所有k-频繁最小CFD。FastCFD将问题的技术挑战变成了在A∈attr（R），r,k>=0时，Cover（A，r，k）的计算。

下面介绍几个概念：

差别集：t1,t2∈r,D(t1,t2;r)={B∈attr（R）|t1[B]≠t2[B]}，即，t1，t2中属性值不同的属性，此外定义D（r）={ D(t1,t2;r)| t1,t2∈r }为r上的差别集。DA（r）={Y\{A}|Y∈D（r），A∈Y}表示属性集合Y\{A},存在r上的元组不满足Y中所有的属性值。若对所有Y’∈ DA（r），Y’为Y的子集=>Y’=Y,则Y∈DA（r）别称为最小。

P（attr(R)）表示attr（R）的幂集。令Z为attr(R)的子集，X为P（attr(R)）的子集。我们称Z覆盖X，当且仅当，对于每个Y∈X，Y∩Z≠∅。若Z中不存在真子集Z’覆盖X，则Z为最小覆盖。

**引理4：**

* + 1. 任意CFD,r|=,sup(,r)>=k,当且仅当，|rtp|>=k,DAm(rtp)=∅，且πA（rtp）=（a）。
    2. 任意CFD,r|=,sup(,r)>=k,当且仅当，|rtp|>=k,X覆盖DAm(rtp)。

引理4提供了一个检测CFD是否包含差别集的方法。

引理5：是一个包含变量的CFD，r|=,sup(,r)>=k。若为最小，那么tp中的常量模式表（Xc，tpc）为k频繁开集。

**FastCFD：**

由两部分组成**FindCover**和**FineMin**。给定一个r,k>0，对所有A∈attr(R),FastCFD调用FineCover（A，r，k）,最终结果返回∪Cover（A，r，k）。

**FineCover**：

由命题1和引理5表明将k-频繁开集看做CFD的常量模式表就可以了。因此，首先获得k频繁开集Frk（r），其次，为Frk（r）中每个（X，tp）保存一个DAm(rtp)，即，从rtp中获得的最小差别集，最后，递归的调用FindMin来找到DAm(rtp)的最小覆盖Y，同时检测（a）（b1）(b2)条件。

**FineMin**：

通过深度优先遍历所有attr(R)\{A}的子集找寻最小覆盖。

用<attr表示attr(R)上的排序关系。例如，attr（R）={A，B，C，D}和A<attrB<attrC<attrD，那么从空集开始，attr(R)\{A}的子集按照下面顺序生成：{B}, {B,C}, {B,C,D}, {B,D}, {C}, {C,D}，{D}

输入：A∈attr(R),(X,tp)∈Frk（r），Y为attr(R)\{A}的子集，DAm(rtp)[Y]，<attr

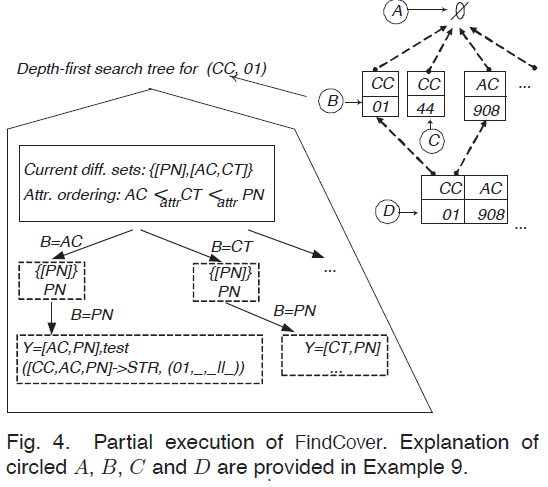
输出：最小CFDs=（[X,Y]->A,(tp,\_,…,||ta)）, ta为常量或‘\_’.

1. 若DAm(rtp)[Y]中存在空集，返回空集。
2. 若Y包含attr(R)\{A}中的最后属性，但是DAm(rtp)[Y]≠∅，返回空集。
3. 若DAm(rtp)[Y]= ∅，那么Y为DAm(rtp)的一个覆盖，相对于条件（a）和条件（b1-b2）有两种情况：
   1. 若DAm(rtp)[Y]= ∅,根据引理4，存在一个常量ta，r |= (X-> A,(tp||ta)).X中是否存大小为|X|-1的真子集X’, r |= (X’-> A,(tp[X’]||ta)),若不存在，输出CFD(X-> A,(tp||ta))。
   2. 若DAm(rtp)[Y]≠∅，根据引理4，r|=（[X,Y]->A,(tp,\_,…,||\_)）.为确定最小性，需要验证如下两条：
      1. Y中不存在大小为|Y|-1的真子集Y’覆盖DAm(rtp[X])
      2. X中不存在大小为|X|-1的真子集X’，Y∪（X\X’）覆盖DAm(rtp[X‘])

若i，ii条件满足，输出（[X,Y]->A,(tp,\_,…,||\_)）。

1. 对每一个在Y后面出现的属性B：
   1. 令Y’=Y∪{B}，DAm(rtp)[Y’]表示DAm(rtp)[Y]中B未覆盖的差集
   2. 调用FineMin（A，（X，tp），Y’, DAm(rtp)[Y’],<attr）.

**例9：**通过一个例子来执行FineCover算法，FineCover（attr(R)\STR,STR,cust,2）,仅包含属性CC，AC，PN，CT，ZIP，STR。假设k=2，<attr按照字母顺序排列属性。如下图4所示，图四的右上角展示了Frk（r）中k频繁开集。和空集最接近的开集子集被箭头标出；左下角画了一个（CC,01）的FineMin执行树。



1. FineCover将开集传递给FindMin。假设第一个开集是空集。这种情况下，FindMin和FastFD一样，返回所有最小FD（Y->STR,(\_,…,\_||\_)）.
2. 考虑开集（CC，01），rCC=01={t1，t2,t3,t4,t8},最小差别集DSTRm(rCC=01)= {[PN],[AC,CT]}。

因此，FineMin通过第4步的递归，找到了DSTRm(rCC=01)的一个覆盖，图4中展示了{AC，CT，PN}子集的枚举树。DSTRm(rCC=01)的覆盖Y为[AC,PN],[CY,PN]。考虑覆盖集[AC,PN]合它的最小CFD =（[CC,AC,PN]->STR,(01,\_,\_||\_)），在第3步，b中，尽管算法证实对于rCC=01是最小的（3.b（i））；还需要检查[CC,AC,PN]是否覆盖DSTRm(r∅)（3.b（ii）），空集是（CC，01）的唯一最接近子集；这种情况下，验证了[CC,AC,PN]覆盖DSTRm(r∅)，表明r|=（[CC,AC,PN]->STR,(\_,\_,\_||\_)）。因此，不是最小CFD。

1. 类似的，考虑（CC，44），rCC=44={t5，t6,t7}，最小差别集DSTRm(rCC=44)={[AC,CT,ZIP]}。

对于覆盖AC和它对应的最小CFD =（[CC,AC]->STR,(44,\_||\_)）。

3.b（i）中证实对于rCC=44是最小的；在3.b（ii）中需要证实[CC,AC]是否覆盖DSTRm(r∅)，同样空集是（CC，44）的唯一最接近子集。D（t2，t4）={PN，STR}，[PN]∈DSTRm(r)，这表明[CC,AC]不是DSTRm(r)的覆盖。因此是最小CFD。

1. 考虑（X，tp）=（[CC,AC],[01,908]）,rtp={t1,t2,t4},最小CFD =([CC, AC, PN] -> STR, (01, 908,\_||\_ ))。

3.b（i）中证实对于rtp为最小的；还需要检查（[CC,AC],[01,908]）的所有最接近子集，例如（CC，01）和（AC，908）来判断的最小性。假设FindMin先检查（CC，01），发现[AC,PN]是DSTRm(rCC=01)的一个覆盖，故不是最小CFD。

### IV.结论

本文共提出三种CFD发现算法：

1. CFDMiner：用于挖掘最小常量CFD。
2. CTANE：用层次化的方法发现一般最小CFD。
3. FastCFD：基于深度优先搜索的方法发现一般最小CFD。

这些方法适用于不同的情况，当仅需要常量CFD时，显然用CFDMiner就可以了，不必生成最小一般化的CFD。当数据集合很大的情况下，可以选择FastCFD。当需要k频繁CFD且k很大时，可以应用CTANE。

## 2.Conditional Functional Dependencies for Data Cleaning

### I.简介

关于CFD的简介详见上篇论文（Discovering Conditional Functional Dependencies）

### II.CFD一致性及推理系统

2.1 CFD的一致性

与FD不同的是，CFD需要考虑到一致性，不同的CFD之间可能相互矛盾。例如：（R：A->B,T1），T1有两个模式表元组（\_, b）和（\_, c）组成，那么没有元组能够同时满足这个CFD，其本身存在矛盾。

CFD一致性问题：判断关系表R中是否存在非空元组集合满足CFD集合。

定理1：CFD一致性问题是一个NP-完全性问题。

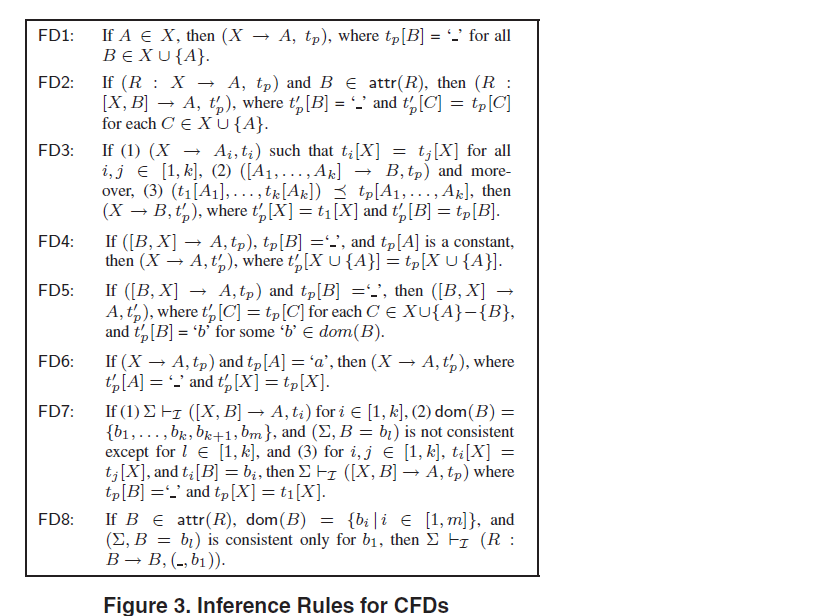
定理2：给定一个关系表R和一组CFD集合∑，∑的一致性能在O(|∑|2)时间内判断出来，前提是关系表预定义过或者∑中没有属性有有限范围。

2.2 CFD推理系统

CFD含义问题：给定一个关系表R上定义的CFD集合∑和一个单独的CFD，判断定义∑后是否需要定义，表示成∑ |= ，即判断是否对所有R中的所有I，若I |=∑，则I |= 。

两个集合∑1和∑2等价：表示为∑1∑2，若对任意I |=∑当且仅当I |= 。

如下图3中提供了一系列推理规则I，给定规则集合∑∪{}，用∑|-I 来表示通过推理规则I在∑中可证实的。



例：考虑∑由两个规则=（A->B,(\_,b)）和=（B->C,(\_,c)）组成，同时含有一个单独的规则=（A->C,(a,\_)）均定义在同一个关系表上，那么∑|-I 将被证明如下：



定理3：推理系统对于CFD蕴含问题是健全且完善的。若∑|=，则∑|-I 。

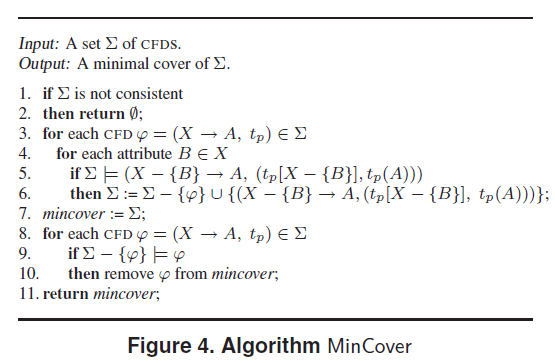
2.3 计算CFD的最小覆盖集

最小覆盖集∑mc满足如下条件：

* + 1. 对于∑mc中的每一个CFD形式为（R：X->A，tp）
    2. ∑mc∑
    3. ∑mc中不存在能够代表它的子集
    4. 对于∑mc中每一个CFD（R：X->A，tp），不存在（R：X’->A，tp[X’∪A]）使X为X’的子集。

MinCover算法：

首先检测∑是否一致（1-2行），若一致，则去掉CFD集合∑中冗余的属性（3-6行）。我们用（tp[X-{B}],tp(A)）来表示模式表tp’，对任意C∈X-{B}，tp’[A]=tp[A]，tp’[C]=tp[C]。接下来从∑去掉冗余的CFD（8-10行）。

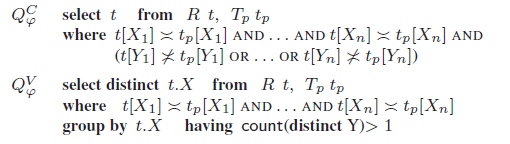


### III.基于CFD的非一致性数据检测

3.1问题描述：给定关系表R和一组规则∑，找到所有违反规则的元组。

这里用SQL语句查询的方法做检测，首先考虑对单一CFD的检测。

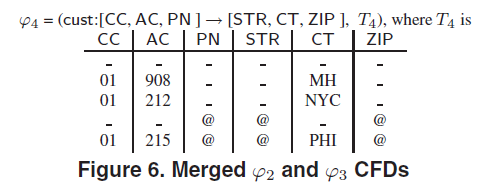
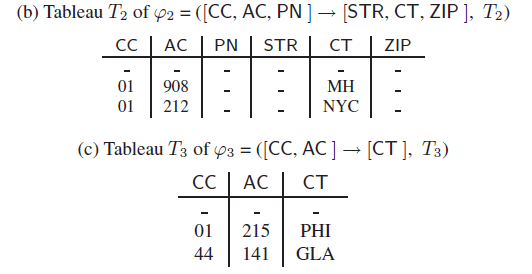
例如对（X->Y，Tp）的检测



对单一CFD的检测1可以通过上述两个SQL查询得到，对于多个CFD构成的集合∑的检测从效率考虑，首先将集合中的CFD重组合并。

**合并CFD：**

考虑一个∑中含有两个CFD=(X->Y, T)和=(X’->Y’, T’)。对于生成合并模式表T∑有两个挑战。第一个挑战是T和T’不能求并，即X≠X’或Y≠Y’。通过Z = (X∪Y )−(X’∪Y’)扩展模式表T。对于Z中的每一个属性A和T中每一个元组tc，令tc[A]=‘@’，用来表示一个无所谓的值。这样扩展后的模式表是可求并的。T∑为他们的并。图6中展示中和的合并。



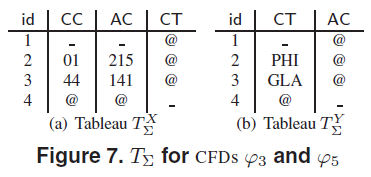
由于‘@’的加入，需要重新形式化表示CFD的满足性。tc[X,Y]为包含‘@’的模式表，用Xtcfree和Ytcfree来表示不含‘@’的X，Y的子集。

I |= <=

t1[Xtcfree]= t2[Xtcfree] tc[Xtcfree] => t1[Ytcfree]= t2[Ytcfree] tc[Ytcfree].



第二个挑战是，如何将合并后的模式表翻译成SQL语句。属性A可出现在一个规则左侧的同时出现在另一个规则的右侧。也就是说我们需要区分这两部分的元组集合，并且分开来进行翻译，将模式表成两部分的结果如图7所示。

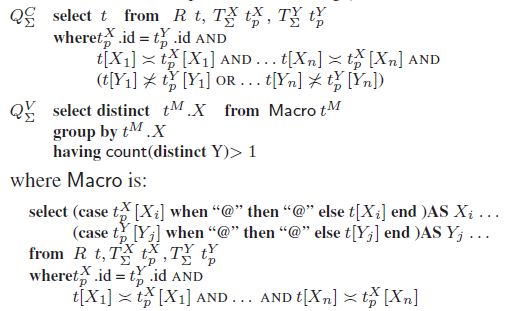


生成SQL查询语句：

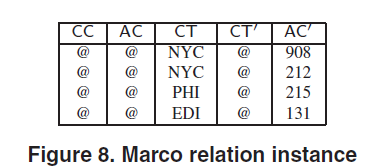
由于‘@’的出现，在进行group by操作时，之前的做法是对全部X都得属性进行group by，然而现在不能这样操作，例如图7中就只需要对前两个属性进行group by，因为第三个属性值均为‘@’和‘-’。

为了使查询大小由规则中嵌入的FD的数量来限定而和模式表的大小无关，需要进行如下操作：

方法的核心是应用SQL中的case从句，下面考虑合并T∑X和T∑Y，下面两个查询可以用来检测违反规则的元组：



Macro考虑到模式表中的每一个元组，并将其作为整个关系表的一个面具。若关系表中一个属性为‘@’则该属性所有值将被‘@’代替。图8中展示了连接图7中T∑X和T∑Y的第四个元组的结果，查询隐瞒了CC和AC的属性值，仅仅对不含‘@’的属性进行group by，这里仅仅对CT进行group by。这样就得到了一对SQL查询语句。仅需要两次数据库扫描。



### IV.结论

本文提出CFD的概念，建立了一个健全完整的CFD推理系统，并将CFD应用到数据清洗中，CFD相比于FD更有利于检测数据问题，此外，提出了一种基于SQL查询的非一致性数据检测算法。

## 3.Descriptive and Prescriptive Data Cleaning

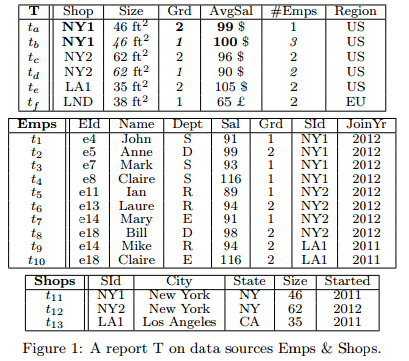
### I.简介

数据清洗技术通常情况下依赖于使用某种规则来定位违反规则的元组，然后使用修复技术修复元组。这些规则都是有某种业务逻辑的，某些情况下只能在多个数据集生成的目标报告上定义。这就造成了违反规则的元组时间和空间上不一致，我们无法再数据源上找出冲突的元组。而且，修复技术要随着数据源改变不断运行。最终，如果在修复的时间和金钱花费上可以承受，定位和分析错误的源头对将来冲突的避免几乎没有帮助。这篇文章提出了一个系统解决了时间和空间分离的问题。系统在输出上定义规则并给出了错误元组可能的解释。这样，我们可以在源数据上描述错误，并在目标数据上改正错误。

### II.系统提出的动机

数据清洗的一般思路是在目标数据库上应用一个系列数据质量规则，找出违反这些规则的数据，并修复。在修复环节，我们要在原数据库的基础上生成一个新的数据库，新的数据库满足数据质量规则的要求。但实际上，数据规则通常是定义在最后的结果中的，在源数据中我们无法定位错误。这就导致错误在时间和空间上分离，之后的数据处理也要使用错误的数据。

例1：考虑下图的数据报告T，HR部门在报告中定位了两处冲突。第一个冲突是(ta和tb)，这是违反了同一家店内管理人员(GRD=2)的薪水应该比普通员工(GRD=1)的薪水高。第二个冲突是(tb和td)，它违反了大的商店不应该有少数普通员工。



为了解释这些错误，我们用谓词来总结冲突。在这个例子中，由于有问题的元组都有属性Region = US，我们就使用[*T.Region* = *US*]。注意这个解释总结了所有冲突的元组，但并不足以解释错误的元组。在某些情况下，更改一个属性就能够使数据库变得一致。例如，一种修复算法将tb.Grd定位为可能的错误。通过更改tb.Grd，两个冲突都被解决了。限制错误元组可以得到更精确的描述。在这个例子中，[*T.Region* = *US∧ T.Shop* = *NY*1]是一个更精确的描述，如果我们假设tb.Grd是错的。

解释数据错误有两方面：一是定位可能错误的元组；而是找出这些元组精确的描述，可以让分析者或者用户看懂。

由于结果报告中的冲突都是来自于源数据集合的错误，我们想知道错误是如何传递的。考虑下面这个例子。

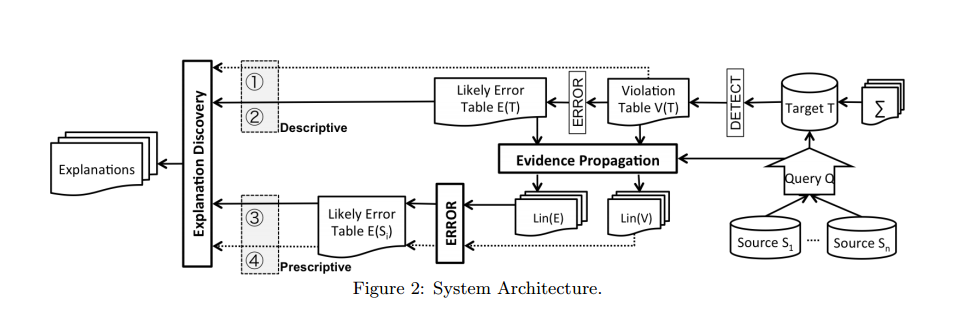
例2 我们假设之前的报告T是一个很多商店数据的并操作。关注原关系Emps和Shops上的查询操作。

SELECT SId as Shop, Size, Grd, AVG(Sal) as  
AvgSal, COUNT(EId) as #Emps,‘US’ as Region  
FROM US.Emps JOIN US.Shops ON SId  
GROUP BY SId, Size, Grd

我们想要追踪那些导致结果冲突的元组。元组ta-td在T中是冲突的，它们的传播路径是{t1-t8}和{t11-t12}。将这些元组去掉，可以解决冲突。两个可能的解释是Emps表格中的[*Emps.JoinY r* = 2012]和Shops表格中的[*Shops.State* = *NY*]。

像之前提到过的，tb是被修复算法定位的错误。它的传播路径是{t 1 , t 3, t 4}和{t 11}。通过关注这些元组，我们可以找出更精确的错误描述，例如[*Emps.Dept* = *S*]。更进一步，通过分析tb的传播路径，t4是最有可能的错误源头。可以移除t4，，这样整体的薪水降低了，解决了冲突问题。因此最准确的描述应该是[*Emps.EId* = *e*8]。这个例子告诉我们计算错误可能性能够找出最准确的错误描述。

### III.系统的结构



本文提出了一个准确描述错误的数据清洗系统(简写为DBRx)。DBRs在输出上定义一系列质量规则，并计算错误的描述。给定一个情景(源数据*Si，查询Q*)和一系列质量规则Σ，DBRx计算冲突表格VT。挖掘VT来发现描述性信息[*T.Region* = *US*]。冲突的传播表格让我们计算更精确的描述[*Emps.JoinY r* = 2012]和[*Shops.State* = *NY*]。找到错误之后，使用修复算法修改目标，这就得出了一个更加精确的描述[*T.Region* = *US ∧ T.Shop* = *NY*1]。

建立这个系统有几项挑战：1. 冲突元组的传播信息包含了大量候选。例如，聚集查询会把很多元组聚集在一起，只有其中的一小部分包含错误。简单的将数据划分为干净或是脏效率很低下。在不同的条件下，元组对错误的贡献也不同。2. 我们需要一个装置记录元组在不同规则下的冲突。对于目标而言，有几种修复算法可供选择。但是对于源数据，我们需要一种放宽语义要求的修复算法。3. 在定位可能错误的元组后，我们要给出用户能看得懂的解释。

### IV.系统各个模块中的关键技术

A.Data Quality Rules数据质量规则

有两种定义规则的形式：一种是否定限制DC。举个例子：

考虑一系列有限内置操作符{>,<,=,*, ≤, ≥*}，一个DC的一般形式是*ϕ* : *∀tα, tβ, tγ, . . . ∈ R,* q(*P*1 *∧ . . . ∧ Pm*)。

另一种定义方式就是使用声明式语言(SQL)和过程式代码(JAVA)程序。

B.目标冲突检测

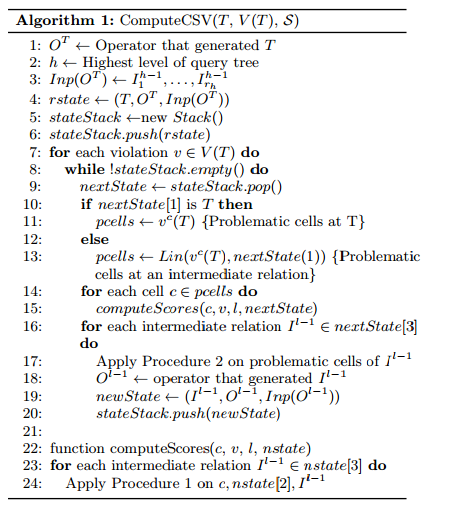
给定一个规则集合Σ，我们需要确保任意属于Σ的规则r都有一个检测函数，生成一个冲突元组的集合。冲突表有自己的模式(vid, r, tid, att, val),vid是冲突的id，r是违反的规则，tid是元组原来的id，att是相关属性名称，val是属性值。

对于DC类规则来说，冲突很容易被检测出来。因为只要不满足条件测试的，都是冲突元组。

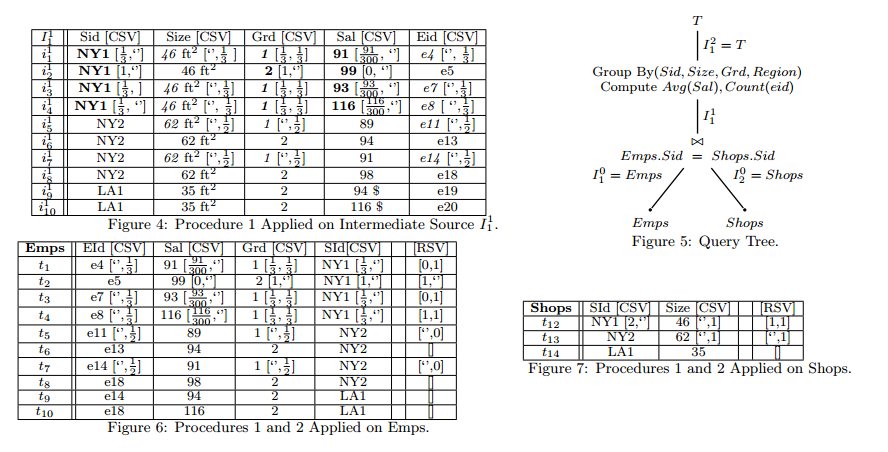
C.错误传播

记录错误传播的路径有两方面内容。首先是要在源头和目标上记录元组的传播路径，我们使用Cui提出的反向查询转化技术来解决这个问题；第二个任务是从源头开始错误是如何传播的。

我们是通过计算CSV(contribution scores vector)来衡量每个元组对错误的贡献。

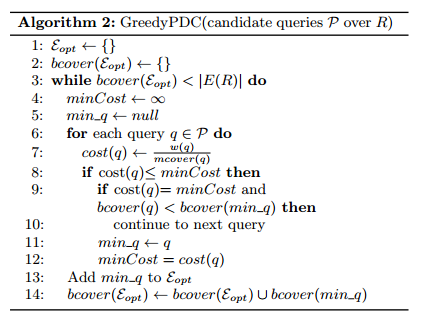


计算RSV(Removal scores vector)来衡量移除元组对冲突的改善有多少，如果移除一个元组之后，冲突消失了，那么这个元组就是重要的。



D.错误描述

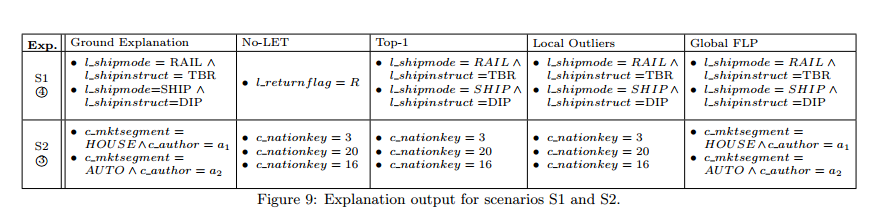
最后我们用一种贪心算法来描述错误，每次都找出能够解决最多冲突的谓词构成描述。



### V.系统的应用

Scenario S1. *cQ*10 : q(*tα.revenue > δ*1).  
Scenario S2. *c0Q*10 : q(*tα.name* = *tβ.name∧tα.c phone*[1*,* 2] *6*= *tβ.c phone*[1*,* 2]).

Scenario S3. *cQ*3 : q(*tα.revenue > δ*2),*c0Q*3 : q(*tα.o orderdate* = *tβ.o orderdate∧tα.o shippriority 6*= *tβ.o shippriority*)



## 4.Scorpion: Explaining Away Outliers in Aggregate Queries

### I.简介

数据库用户通常使用聚集操作将大量的数据转换成少量低维度的数据，并可视化结果。数据可视化之后就会暴露出某些离群点(outlier)，这些离群点有时代表数据错误，而有时代表输入数据集的一些令人惊奇的特点。而现有的数据库和可视化系统并没有提供将离群点反向推回输入数据的功能。Scorpion，是一个能够将聚集操作结果中用户指定的离群点作为输入，使用一些谓词描述这些离群点性质的系统。特别地，这些谓词能够帮助我们在以后输入数据时，避免离群点的出现。为了找出这些谓词，系统定义了谓词的“影响力”指标，并设计算法找出具有最大影响力的谓词。

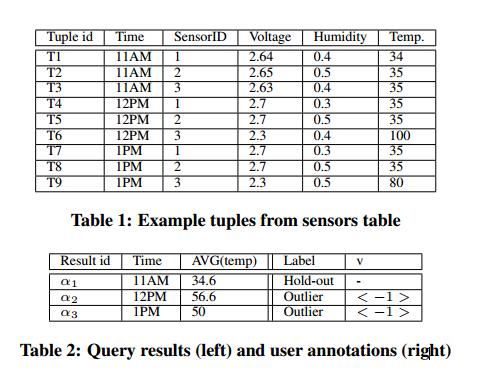
### II.系统提出的动机

为了说明Scorpion可以帮助用户了解输入数据的何种性质影响了聚集结果，我们考虑以下几个场景。

传感器数据：考虑一个数据分析师正在分析一个传感器数据集，如图1所示。下面的查询语句想计算每个小时内的平均温度，图2列出了聚集操作的查询结果。

SELECT avg(temp), time

FROM sensors GROUP BY time



分析师认为12点和1点的平均温度出乎意料的高，她想知道是为什么。在这种情况下，她想问以下几个问题。

(1)哪个传感器造成了这种不正常现象？

(2)哪个传感器的值是造成这种现象的主要原因？

(3)为什么这些传感器会有这么高的温度？

(4)这个现象昨天没有发生，今天为什么发生了？

这些问题都是对传感器的某些性质提问。例如1和2包含了影响程度，4是与其他正常情况对比。

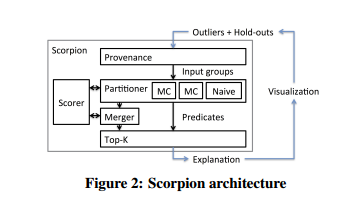
医疗成本分析：我们希望可以节约医院的各项开支成本。在对大量的肿瘤患者的调查中发现，花费在前15%的患者占据了半数以上的开支。令人惊讶的是，那些不太严重的病人并没有比中等花费病人更好或是更坏。后来他们发现有一小部分医生喜欢开很多不必要的处方，而这并没有导致医院的收入增加。医院很想知道花费最多的是哪部分，这样可以有针对性的减少成本。

以上这些场景的共同点是：我们不仅想知道结果，也想知道是什么样的原因导致了这些结果。现有的系统在追踪源头的时候会考虑很多候选，但大部分的候选是没有意义的。Scorpion可以将结果的影响集合大大缩小。

### III.系统的结构

Scorpion是一个端到端的数据分析工具。用户可以选择数据库和聚集函数，最后的结果会可视化为图表或表格形式。用户能够选择任意的结果，将它们标记为离群点或是保持点，选择用于建立谓词模型的属性，并回送回Scorpion系统。

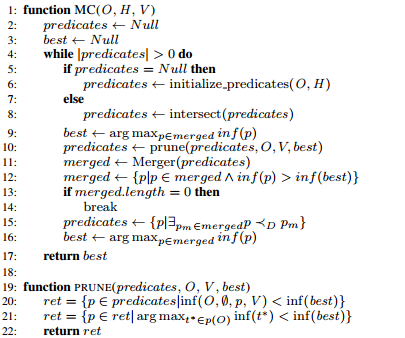
首先，Scorpion使用Provenance部件计算标记结果的来源并返回输入集合。输入集合和原始数据一起被送到Partitioner，根据聚集函数选择相应的划分算法。最后生成一个谓词排序列表，每个谓词都有一个预测影响力的分数。然后将列表送入Merger，贪心地将相似的谓词合并以扩大影响力分数。Partitioner和Merger都把候选为此送到Scorer，它计算预先定义好的影响力分数。最后，排名最高的谓词会被返回可视化结果，展示给用户。



### IV.系统各个模块中的关键技术

A.划分

对于没有良好性质的聚集函数来说，最理想的划分就是穷尽所有可能的谓词。这是因为给定元组的影响力与其他元组有关，所以简单的贪心算法不行。NAIVE算法首先定义所有唯一单属性从句，从句使用一个属性替代” in (…)”这样的形式。连续属性的从句需要先分割成固定大小的集合，并穷举所有集合的组合形式。这个算法效率很低因为单属性从句个数是以指数增长的。所有可能连词的空间是属性个数的指数函数。这两个问题的组合使得搜索范围过大，即使是小数据集合也不现实。因此用户可以设定最大从句数来减小搜索范围。



B.合并

Merger输入根据影响力分数排序的列表，计算两个谓词之间相应属性的距离。使用贪心算法不断的扩展已有谓词并排序，直到影响力分数不再改变。

C.聚集性质

上面两个算法的搜索空间都过于巨大，为了能在可控的之间内算出结果，这里我们给出几个性质，剪枝候选谓词。

(1)增量移除

所谓增量移除就是更新后的结果可以通过移除的子集s直接计算，而不用考虑全部输入集合D。例如SUM函数，SUM可以具有增量移除性质，因为SUM(D-s) = SUM(D) – SUM(s)，SUM(D)可以被缓存起来。事实上，一个聚集操作的影响是可以增量移除的当且仅当增量函数自己是可增量移除的。

(2)独立性假设

Scorpion的独立性假设是元组对聚集结果的影响是独立的，不受其他元组影响。因此两个谓词组合一定比一个谓词的影响力大。

(3)反单调性影响

反单调性类似于频繁项集的思想，如果元组s违反了这个性质，那么它的所有自己都不满足这个性质。在这里，如果p是影响小的，那么p中任意谓词影响都不会超过p。这样可以大大减少候选元组数量。

## 5.Efficient Approximation Algorithms for Repairing Inconsistent Database

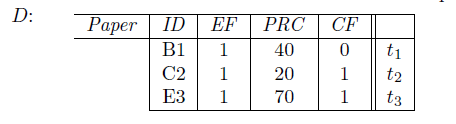
### I.背景和领域介绍

在修复非一致数据库时一般是修改数据的值。在这篇文章中，我们主要是关注在计算复杂度的分析方面。我们用一种有效的近似算法来修复数据库，并且提出了一种执行时间为O(nlogn)的算法。在本文中，实验表明，即是是对大型数据库，我们的算法也同样有效。

在处理非一致性数据库时，有两种方法：

1. 数据清理
2. 一致性查询问答（CQA）：它保持源数据不改变，在查询时解决非一致性，识别一致性数据。

Example



约束条件：只有当PRC>= 50,且CF=1时，EF才为1；

所以在数据库D中，元组t1和t2不满足约束条件。有两种方法进行修复：

1. 删除最小数量的元组来满足约束条件，即要删除t1和t2，这样只有t3留在数据库D中。
2. 修改最少的的属性值来满足约束条件。例如，我们可以将t2中的EF值从1改为0，这样就可以减少丢失的信息。

由于在找从关于IC的属性更新修复到已知输入事件的距离是MAXSNP-hard问题。通过将这个问题转换为最小权值覆盖问题，这个修复问题可以通过例如greedy或者layer的近似算法来计算。

但是greedy算法的执行时间是O(n^3)，这对于真正的数据库来说是不够高效的。在这篇文章中，提出来一种变形greedy算法，在假设某元组可以参与到多种在恒定范围内的非一致的地方，其执行时间可以降为O(n^2 logn)。通过实验得出，修改后的greedy算法要比greedy算法快。但greedy算法要比layer和修改后的layer算法都要快。

### II.算法的说明与步骤

Cardinality Repairs

在有不一致的数据时，可删除该数据。但基础原子查询的查询回复的复杂性是。并且在语义条件下获得数据库修复是计算复杂度。所以需要一种近似算法。在每个关系中增加一个属性，它是数据库中唯一可以被修改的属性。在元组中，表示其在数据库中，反之，时，表示是删除的元组。所以数据库中的不一致数据可以将的值从1改为0，就相当于删除了该数据。

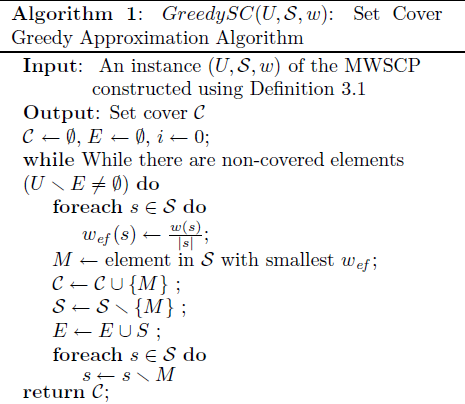


**Attribute-Update Repairs**

**算法背景：**把问题转换为MWSCP，Minimum Weighted Set Cover Optimization Problem，即为最小权值覆盖优化问题。给定集合U，S是U的子集，权值集合w（Si），覆盖集C是S的子集。U中的每个元素都至少属于C中的一个，其中所有权值的和最小即可。

Algorithms 1：GreedySC（U,S,w）

**伪代码：**



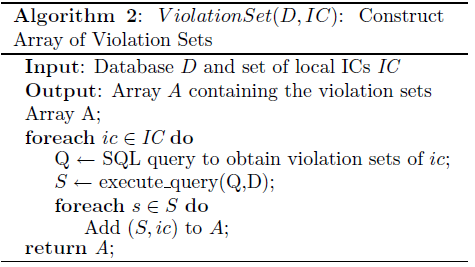
**算法说明和解释：**输入分别是U,S,w，只要还有没被覆盖的元素，就进行循环，选取S中最小的Wef，最终输出C是最优覆盖集。

**算法结论：**当数据库的大小是n时，该算法的执行时间是O（n^3），并且当Deg（D，IC）为常数时，则为O（n^2）。即使当不一致的度是已界定时，其执行时间可控制在平方时间内。但对大型数据库来说代价还是太高，所以我们要修改原greedy算法来获得更高效的方法。

Algorithms 2：

**算法背景：**在原greedy 算法中，在找没有被覆盖元素的最小权值是代价消耗最大的一步；**算法基本思想：**我们将S存储在优先队列P中，在优先队列中每个元素都包含一个非一致元组t，其权值，和指向U中元素的指针，是一个violation sets，他们会存储在数组中。主要是获得包含violation sets的数组A。通过重写每个完整性约束作为SQL视图，即是如果被满足则为空。

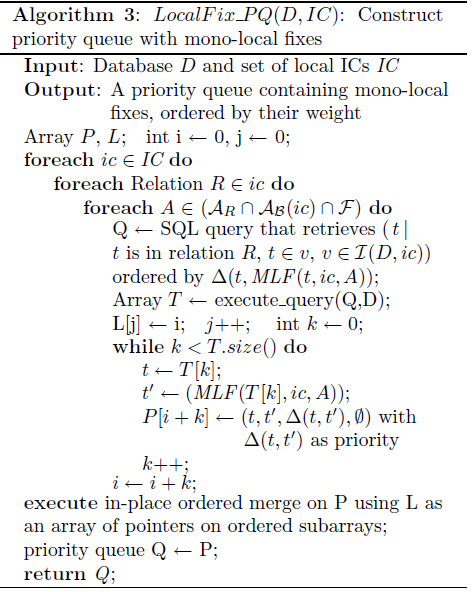
**伪代码：**



Algorithms 3：

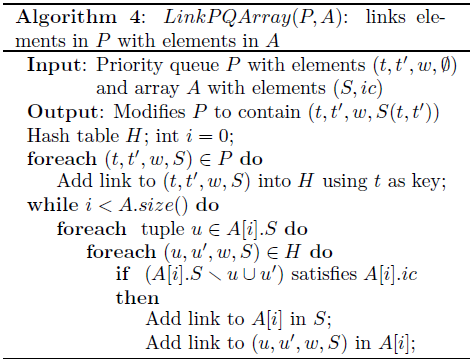
**算法基本思想：**通过计算每个非一致元组t的mono-local fix，t’=MLF（t, ic, A）。并将其对应的权值按顺序存在数组中。这个有序数组可看作是堆优先队列实现的数组。但在此阶段，P中的s（t,t’）是空；

**伪代码**：



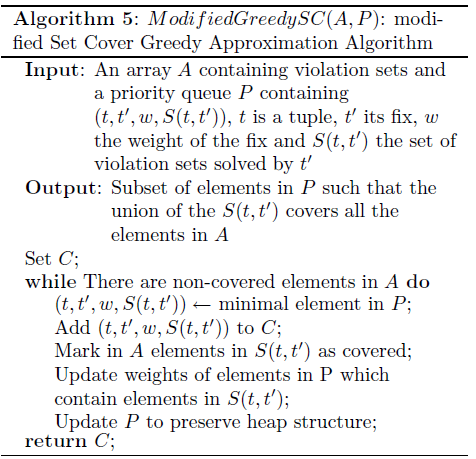
Algorithms 4：

**算法基本思想**：将在数组A存储的violation sets加入到每个s(t,t’)中，这个算法运用了包含P中元素链接的辅助哈希表来实现。



Algorithms 5：ModifiedGreedySC

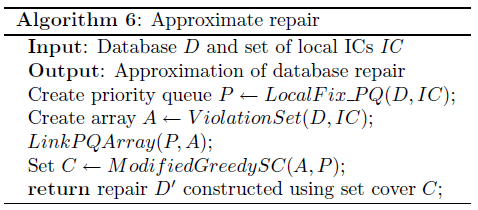
**算法基本思想：**在每次迭代中都将优先队列中最小元素加入到覆盖集中，则每次优先队列的权重也相应更新，由于A中包含了链接P中的指针，所以每次更新的效率都很高效。其次，我们还要存储堆结构，这通过自上而下堆来实现更新元素。



Algorithms 6：

**算法基本思想：**完整的修复算法包括MWSC问题的构建和modified-greedy算法的执行。

**伪代码：**

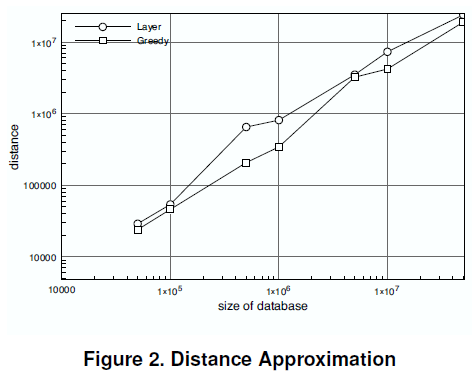


算法结论：其执行时间为O(n^2logn)，若Deg（D,IC）为常数，则为O(nlogn);

### III.实验

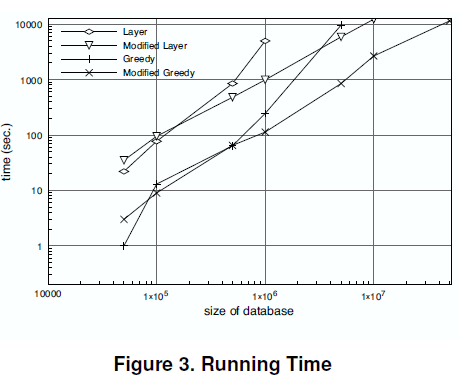
随机产生3个不同大小的数据库，其中30%的元组是不一致的。

1. 比较greedy和layer算法，看哪个能给出一个更好的近似数据库修复方法。如图2；



理论上，我们预期layer要比greedy算法好，但实验结果表明，greedy能产生更好的近似最优MWSCP权重。

1. 比较MWSCP近似算法的执行时间，如图3；



采用优先队列来存储MWSCP对，能提高性能。

* 对大型数据库来说，修改后的greedy算法是最有效的；
* 修改后的greedy算法可有效的用于计算相对于一组本地数据库完整性约束的修复，甚至可用于大型数据库；
* 修改后greedy算法运行速度快，相比于layer算法，能给出更好的近似结果；

### IV.结论

这篇文章提出了一种优化算法来计算在关于局部线性否定约束的非一致数据库中进行属性更新修复。

这个算法首先将修复问题转化成MWSCP，并用一种有效的修复贪心算法来解决MWSCP，我们实验得出这个算法也可以适用于更大型数据库；

由于greedy和layer算法分别和对数，常数因子接近，所以我们预期从layer算法中会得到更好的估计值。然而，实验结果表明贪心算法结果更好，且修改后的贪心算法比修改前执行时间更短，比layer更快。

运用修改后的greedy算法可以将元组删除修复法转化成属性更新问题。这样，数据库就不用主键约束，也不用要求在线性否定ICs上的局部性设定。

本文的结果同样在经过小的修改后运用到其他语义修复上。例如，将设定为不同的值，这样的值也会随之改变，可以确定从表中删除元组的优先级，表示先从表R中删除元组。



## 6.Interaction Between Record Matching and Data Repairing

### I.背景和领域介绍

匹配（Matching）：主要将元组和现实中的实物对应起来；

修复（Repairing）：用约束条件来修复数据库中的错误；

现阶段，两者处于相对独立的过程。所以在这篇文章中，我们研究了一个新的问题，即记录匹配和数据修复之间的交互作用。我们发现修复可以有效地帮助我们找到匹配的实物；同理，匹配也能有效的帮助修复。

为了得到这种交互作用，我们提出了一种统一的，能无缝结合修复，匹配操作的框架。在完整性约束，匹配规则和主数据的基础上来清理数据库。

再通过匹配和修复清理数据问题上，我们给出了详细的分析，包括约束规则共同作用的静态分析，其复杂性，终止性和确定性数据清理分析；但这些问题是NP，和coNP完全，PSPACE完全的难问题。

但我们才用了同时通过匹配和修复清理数据的高效算法。该算法分别通过置信分析和熵分析得到确定性修复和可靠性修复，这比通过启发式产生的可能性修复要精确多了。通过实验证明采用现实数据，我们的方法比作为独立国产记录匹配和数据修复的精确度要高得多。

新问题：有数据库D，主数据Dm，CFD，匹配规则г；找到Dr使得

1. 满足各项CFD条件，且为一致性；
2. Dr中没有能和Dm匹配的元组；
3. Dr与源数据D相比差别是最小的；

所以，我们提出了一个统一的框架来清理数据：将CFDs和MDs都作为清理规则，能告诉我们如何修复错误；利用a.用户设定的数据置信值；b.数据自己产生的熵来计算数据确定性；c.主数据，三者来确保其准确性；

### II.算法说明及步骤

由于数据清理问题是NP-hard，所以我们提出来三种算法构成的方法：

1. 依据置信值分析和主数据确定精确的确定性修复；
2. 当置信值很低或没有时，采取根据信息熵来计算可靠性修复；
3. 采用启发式算法在脏数据中找到一致性修复；

以此来提高精确度；

我们采用DBLP和US Health & Human Service的数据；实验证明我们的方法比独立采用匹配和修复方法要好得多，且确定性修复和可靠性修复要比启发式方法的修复要精确的多。尽管清理数据库是高复杂性的，但我们的算法也能很好的运行。

数据清理规则：

CFD：Conditional Functional Dependencies 有条件函数依赖；

CFDφ是定义在R上的配对（X—>Y, tp）。当

1. X🡪Y是R上的一个标准函数依赖，相当于FD是嵌入在φ中。
2. Tp是规则模式，X和Y的属性，对于每个XUY中的属性A，tp[A]要么是常数，要么是未命名的变量。

MD：Matching Dependencies 相似谓词的集合г

分别有定义在R和Rm上的positive MD和Negative MD；

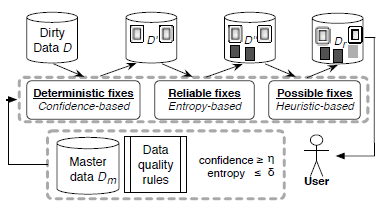
三级数据清理方法（A tri-level Data Cleaning Solution）

* UniClean:

输入：D，Dm，清理规则θ，阈值：ηδ分别从熵和置信值得到；

UniClean是通过这些清理规则，将匹配和修复整合在一起来产生修复；用三级精度值区分这些修复，强调其准确度

如图所示的框架概览，通过三个算法一个接一个的执行来得到修复；



1. 通过置信值得到置信修复：只有当t中某属性的置信度高于阈值η时，才根据置信度用清理规则更新t[A];
2. 通过熵得到可靠性修复：当某些属性置信度很低或没有置信度时，则采用熵来修复；只有当t的某一属性的熵低于阈值δ，则采用清理规则г来更新t[A];
3. 并不是所有的错误都能在前两步修复，所以采用启发式方法产生修复，即possible fix；

数据质量规则的两个主要问题

1. 一致性问题：在给定Dm和θ=∑∪г，确定是否有非空D，使D|=∑，且（D，Dm）|=г。直观的来看，是判断在θ中的规则本身是否是脏数据。（NP-complete）；
2. 蕴含问题（Implication Problem）：给定Dm，∑，和CFD/MD ζ，确定是否有∑|=ζ。帮助我们找到并删除在∑中多余的规则，以此提高性能（coNP-complete）

基于规则的数据清理的两个问题

1. 终止问题（Termination Problem），看整个过程是否终止，到达一个不动点，没有任何清理规则能再运用；
2. 决定性问题（Deterministic Problem），看是否所有终止问题都在相同点不动点结束；

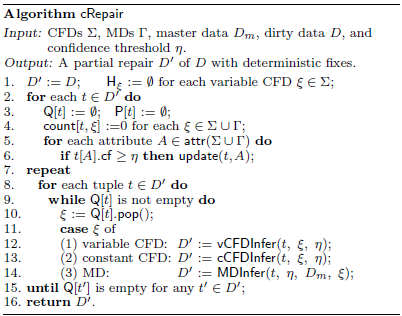
看整个过程的输出是否独立于规则应用的顺序，都是PSPACE-Complete，所以不能再实际中找到有效。

由数据置信值得到的确定性修复：(Deterministic Fixes with Data Confidence)

确定性修复由三种方法产生：a. 从MD；b. 从固定的CFD；c. 从变量CFD

* **基于置信值的数据清理算法cRepair**

**伪代码：**



**算法解释：**

输入：CFD∑，MDг，主数据Dm，脏数据D，置信阈值η

Line1—6：初始化

Line7—15：递归计算确定性修复，分别从变量CFD，常量CFD和MD中得到vCFDinfer，cCFDinfer，MDinfer。直到没有更多的确定性修复能找到；

Line16：返回较干净的数据D’；

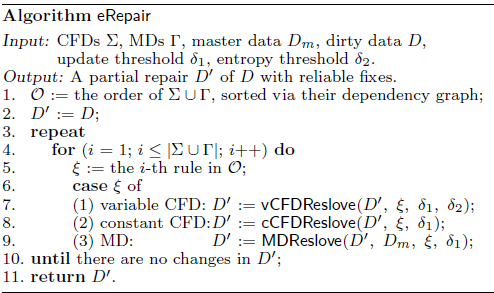
**算法结论：**复杂性：O（|D||Dm|size(∑∪г)）;由于采用MD找确定性修复时，可采用后缀树，则可优化为O（|D|size(∑∪г)）;

* **基于信息熵的可靠性修复（Reliable Fixes with Infromation Entropy）**

**算法基本思想：**离散随机变量χ的熵为：；熵值能描述χ的确信度。越小，预测越准确。



**伪代码：**基于熵的数据清理算法eRepair，如下所示：



**算法解释：**

输入：CEDs∑，MDsг，主数据Dm，脏数据D，δ1更新频率阈值，δ2熵的阈值；

找到D的可靠性修复，且在此过程中deterministic修复不发生变化；

Line1：根据依赖关系图，将∑∪г中得到的清理规则进行排序，有较大影响力的条件则优先应用，依次类推；

Line3—9：依次将排好序的规则解决D中的冲突，根据规则的类型，选取方案，得到vDFDResolve，cCFDResolve，MDResolve。

Line10：直到没有更多的规则可用或所有数据的值都超过δ次改变，则停止。

Line11：返回D’，是一个干净的带有可靠性修复的数据库；

### III.实验

**Exp1：匹配能帮助修复**

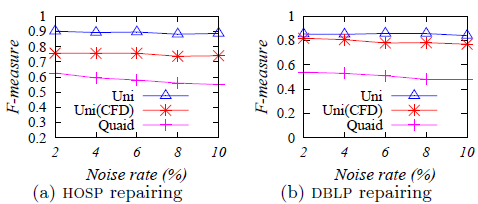
分别将Uni，Uni（CFD），quaid得到的F-measure值进行对比，Noi%从2%~10%；

结论：A. Uni比Uni（CFD）和quaid都要好，所以匹配能够帮助修复；

B. 随着noi%的上升，F-measure下降，但对Uni 的影响较小；

C．即使只采用CFD，Uni（CFD）比quaid好，因为quaid只产生possible fixes，而Uni（CFD）可靠性和确定性修复都能找到；

如下图所示：



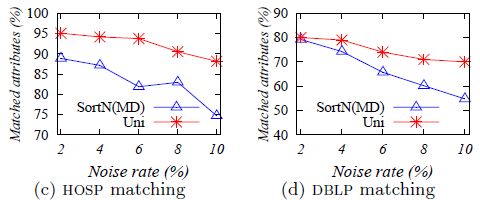
**Exp2：修复能帮助匹配**

对比Uni和采用MD的SortN（MD）的匹配属性所占的比例；

结论：A. Uni比SorN（MD）的效果好，所以修复能帮助到匹配；

B．随着noi%的升高，F-measure的值下降，但Uni影响较小；和Exp1中的结论一样；

如下图所示：



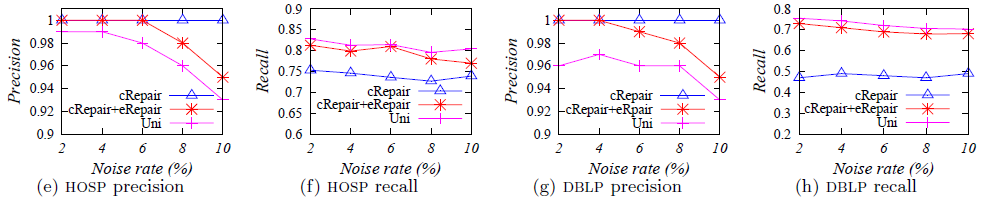
**Exp3：确定性和可靠性修复的精确度**

对比cRepair产生的确定性修复，cRepair+eRepair产生的确定性和可靠性修复，Uni产生的所有修复；

结论：

1. 确定性修复的精确度最高，但回溯率很低；因为cRepair只对asserted attribute进行修复，对noi%不敏感；
2. 由Uni得到的修复精确度最低，回溯率最高，且精确度对noi%十分敏感；因为Uniclean的最后一步是采用启发式方法，会产生possible fix；
3. cRepair+eRepair得到的精确度和回溯率在两者中间，精确度同样对noi%敏感；

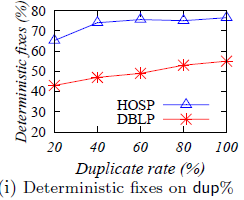
如下图所示：



**Exp4：dup%和asr%对确定性修复的影响**

看在cRepair中确定性修复占有的比例；我们将asr%=40%，dup%从20~100%变化；

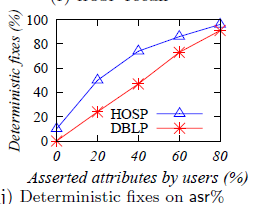
如图：



结论：当dup%升高时，找确定性修复越多；

若将dup%=40%，asr%从0%~80%变化；

如图：

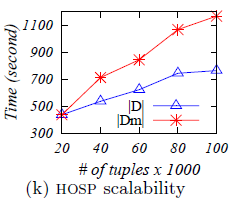


结论：cRepair中找到的确定性修复主要由asr%控制，所以清理规则只对asserted attribute使用；

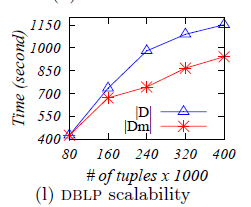
**Exp5：可扩展性**

|D|大小的脏数据和|Dm|大小的主数据，noi%=6%，dup%=40%；

如图所示：



确定|Dm|的大小为60K，|D|的大小从20K~100K



确定|D|的大小为60K，|Dm|的大小从20K~100K

结论：Uni的扩展性比quaid好，说明Uni中的索引结构和优化方法十分有效；

实验结论：

1. 通过将匹配和修复结合在一起来清理数据比分别单独使用配合和修复的方法性能提高了30%，15%；
2. 确定性修复和可靠性修复的精确度较高，等noi%<4%时，精确度可达100%；且当noi%上升时，精确度降低的很缓慢；
3. 通过UniClean得到的备选修复景区度可达96%；
4. 我们的数据清理方法可扩展性也很强，比只用CFD的quaid快将近50倍；

### IV.结论

提出了将匹配和修复两个过程结合起来的方法。提出了一个基于CFD和MD得到的清理规则的统一的框架，使得匹配和修复过程结合起来。并且，根据确信值和熵值提出了确定性修复和可靠性修复。我们通过实验证明这个方法能够有效的提高数据质量，且它们的可扩展性也很强。

## 7.Improving Data Quality:Consistency and Accuracy

### I.算法提出的背景和动机

1. 算法提出的背景

当前文献的主要研究方向主要集中如下：

1) 许多文献已经提出了大量的约束规则，利用SQL检索违背条件函数依赖的元组等，但是许多文献没有提出修复方法。

2) 基于-约束的数据清晰技术主要关注两个话题：其一，修复是查找到与原始数据库一致的，具有最小不同的数据库；其二，一致性查询解决方案是在原始数据库中查找到满足查询的所有元素对。

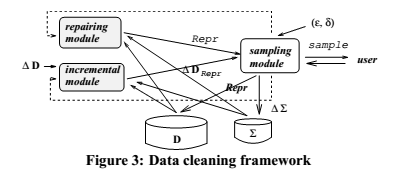
3) 某个文献中针对违背函数依赖时，给出了修复算法。本文将其扩展到违背条件函数依赖的情况。

4) 一些文献将修改值作为一种修复手段。

5) 一些文献将合并不一致项作为主要的修复方法。

1. 算法提出的动机

参考以上文献，该算法包含如下三个模块：



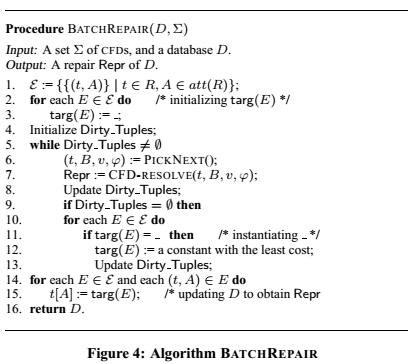
* 1. 针对数据清洗模块(repairing module)，提出了BATCHREPAIR算法，其动机为：当某些元组违背了某种函数依赖，对这些元组进行修复后，修复后的新元组可能又违背了另一个函数依赖，当再次修复这些元组后，可能造成违背了上一个函数依赖，即很可能导致一个无限循环。因此，本文提出了BATCHREPAIR算法以解决该问题。
  2. 针对增量模块(incremental module)，提出了INCREPAIR算法，其动机为：为了解决增量修复问题。
  3. 样例模块(sampling module)为将准确率上界和置信度(ϵ,δ)作为输入，该模块产生的输出样例与用户进行交互，用户进行反馈，直到准确率满足在置信度的上界之内时为止。

### II.算法的基本思想

1. BATCHREPAIR算法的基本思想：当某些元组违背了同一条件函数依赖时，我们通过归并-更新操作来修复元组。当前的所有目标取值包括’\_’,’a’,null。其中，’\_’表示当前取值未知待定，’a’表示当前值为一个特定的常量，null表示当前值取任何值均会违背某一个条件函数依赖，因此当前值无法判定。且任意一个待修复位置处的取值仅可从’\_’变为’a’，或从’a’变为null。BATCHREPAIR算法的目标是对所有违背条件函数依赖的位置处赋予’\_’,’a’,null之一以清洗数据。
2. INCREPAIR算法的基本思想：它将(只包含插入操作，由于删除操作必定不会导致任何不一致)，干净的数据库D,一系列条件函数依赖集作为输入，从而找到一个具有最小代价的修复来完成对数据库的清洗操作。INCREPAIR算法的核心是TUPLERESOLVE程序，它旨在解决局部修复问题。通过不断地选择最好的k个属性集，直到所有属性集被覆盖完毕。

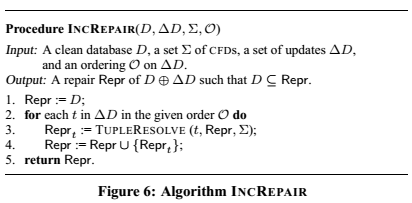
### III.算法的伪代码+解释

1. BATCHREPAIR算法伪代码



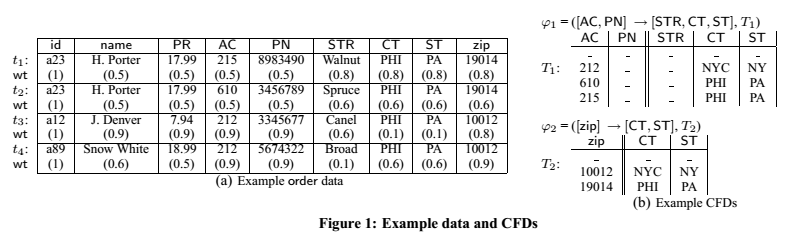
首先，对等价类和脏数组集进行初始化工作(第1-4行)。只要当前存在脏数组(第5行)，就利基于贪心思想的PICKNEXT算法来寻找出会产生最小修复代价的元组作为下一个修复元组(第6行)。通过同时调用CFD-RESOLVE函数分析待修复元组如何违背了条件函数依赖，生成一个更新等价类的集合(第7行)。对脏元组集进行更新(第8行)。如果当前脏元组集为空，则首先令每个等价类的目标值为’\_’，随后将每个目标值更改为一个具有最小代价的常量’a’(第10-12行)。同时更新脏元组集(第13行)。经过所有的迭代后，对等价类的每个目标值进行更新，得到最终的修复元组(第14-15行)。

1. INCREPAIR算法伪代码



首先，它用干净数据库D来对修复集Repr进行初始化(第1行)。随后根据给定的顺序O(第2行)来调用TUPLERESOLVE程序完成对中每个元组t的清洗(第3行)，并将关于t的局部修复更新添加到Repr中(第4行)。一旦中的所有元组均被处理完毕，则返回最终的修复结果(第5行)。其中，TUPLERESOLVE本质是个NP-完全问题，此时利用了贪心思想，每次“贪心”地寻找关于元组t的k个最好的属性集进行修复。

### IV.例子解释算法



1. BATCHREPAIR算法示例

现假定添加一个新的元组。首先，违背了条件函数依赖。我们通过令来解决这一问题。此时。但是当前的违背了条件函数依赖。若我们令来解决违背条件函数依赖的问题。但是这再次导致违背条件函数依赖。因此，为了解决这一无限循环问题，引入了BATCHREPAIRE算法。

现假定我们需要解决违背了条件函数依赖这一问题。此时关于CT的目标值为PHI,关于ST的目标值为PA，因此我们无法再将CT,ST上的值更改为其他常量值。此时我们可以更改的左侧项的目标值。即令。

1. INCREPAIR算法示例

INCREPAIR算法的核心是TUPLERESOLVE程序。现假定为了解决插入所产生的违背条件函数依赖的问题。

首先假定k=2。现设TUPLEREPAIR已经固定了所有的属性(除了CT和ST)。因为违背条件函数依赖和，仅仅通过更改两个非固定属性CT和ST无法解决该问题，因此令k=3，并且令C={CT,ST,zip}。此时可以通过令来解决这个问题。

### V.算法的结论

1. 提出了一个基于条件函数依赖的改进数据质量的框架。
2. 证明了寻找最优修复的问题和增量地寻找最优修复的问题均是NP-完全的。
3. 为了改进数据的准确性，我们提出了一个统计方法以确保寻找到能够在一个高的置信度的一个预定准确率之上的一个修复方案。
4. 该方法既考虑到了一致性，又考虑到了准确性，是第一个对关于条件依赖的数据清洗方案。

## 8.On Approximating Optimum Repairs for Functional Dependency Violations

### I.算法提出的背景和动机

1. 算法提出的背景

随着数据集成系统的发展，大型数据库违背了潜在约束集，进而导致数据的不一致现象时有发生。为了处理不一致数据，许多研究人员通过寻找满足约束限制同时尽可能地与原始数据接近的最小修复来进行数据清洗工作。当前主要研究趋势分为如下三类：

1. 通过寻找所有可能的最小修复并取它们的交集作为修复结果。
2. 通过进行插入、删除、更改操作来完成数据修复，同时以具有最小代价的修复(即选择具有与原始数据库最小距离的修复操作)完成数据清洗工作。
3. 通过生成一个主核(即可得出一致性的查询结果的修复的集成代表)，来得出满足一致性的结果。
4. 算法提出的动机

参考上述文献，本文采用编辑距离手段来寻找出最小代价修复，我们提出了V-repair方法，它可以看作是一个特殊的条件表示例，当不存在局部条件时，能够应用于每个不同的值对来表示不完整的信息。这与以往研究中只能针对固定的值的方法有所区别。其主要贡献如下：

1. 相比于以往研究，我们的距离测量方法更加简单。
2. 当实践中只能对一小部分值进行修改操作时，它更适用，且它对以往的启发式算法作了有效补充。

FINDVREPAIR算法提出的动机：理论证明寻找一个多项式时间的近似最优修复算法是一个Max-SNP-难问题。因此，我们利用顶点覆盖问题原型得出了一个能够在多项式时间完成修复的算法，且该V-repair结果仅仅是关于最优修复编辑距离的一个常量因子。故该算法能够在多项式时间内找到一个较优的修复结果，降低了时间复杂性，提高了效率。

### II.算法的基本思想

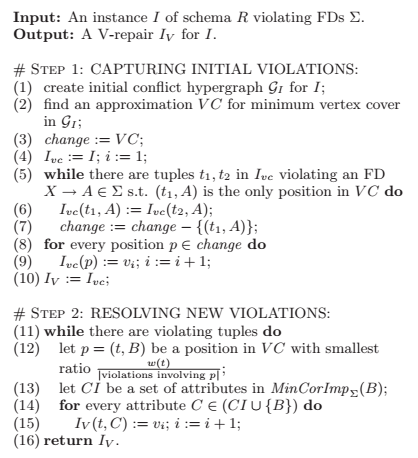
V-repair为一个新的数据修复模型——通过常量-到-常量的修改，或者为常量-到-当前属性中可取的一个变量的修改来解决违背函数依赖问题。我们定义了一个检测一个数据库和修复后的数据库之间的距离检测方案：仅仅依赖于值修改的数量和修改的元组的权重(以往研究中元组的权重是指数据拥有者对该元组的置信度)。而我们的检测方案中仅仅考量了值修改的数目，因此将所有修改的元组看作是同等重要的，即权重相同，可以忽略不计。故该方法更加简洁。

FINDREPAIR算法的基本思想如下：

1. 首先，我们对输入的不一致的数据库实例I，初始化构造出一个违背约束的超图G1。
2. 随后我们利用图论中的一个算法，得到一个关于G1的一个近似的最小顶点覆盖VC。
   1. (VC中的位置即为初始时刻需要改变的值的集合)。
3. 对于VC中的每个位置，如果通过对当前位置的值取该属性处可取的另外一个值能够满足函数依赖，那么就将当前位置的值更改为另一个可取的值；若均无法满足函数依赖，则我们令当前这个值取另外一个新值。
4. 得到的V-实例可能还包含违背函数依赖的情况，因此我们可以进一步调用RESOLVE NEW VIOLATIONS来进一步更改某些位置的值以保证不会有新的违背函数依赖的情况发生。

### III.算法的伪代码+解释

1. FINDVREPAIR算法伪代码



1. FINDVREPAIR算法解释

STEP1解释：

首先，我们对输入的不一致的数据库实例I，初始化构造出一个违背约束的超图G1(第1行)。随后我们利用图论中的一个算法，得到一个关于G1的一个近似的最小顶点覆盖VC(第2行)。其中VC中的位置即为初始时刻需要改变的值的集合(第3行)。对于VC中的每个位置，如果通过对当前位置的值取该属性处可取的另外一个值能够满足函数依赖，那么就将当前位置的值更改为另一个可取的值；若均无法满足函数依赖，则我们令当前这个值取另外一个新值(第4-9行)。得到的V-实例可能还包含违背函数依赖的情况，因此我们在在最后一步中通过调用STEP2:RESOLVE NEW VIOLATIONS以某种方式来更改某些位置的值以保证不会有新的违背函数依赖的情况发生(第10行)。

STEP2解释：

当引入新的违背函数依赖的情况时(第11行)，根据贪心算法，我们选择一个会产生最小代价的位置(即具有最小的)来进行更改(第12-15行)，因此这样能够覆盖所有的违背约束的情况通过挑选出VC中的一个集合中的总权重最多为VC中所有位置权重和的一半的情况。最后返回(第16行)。

### IV.例子解释算法

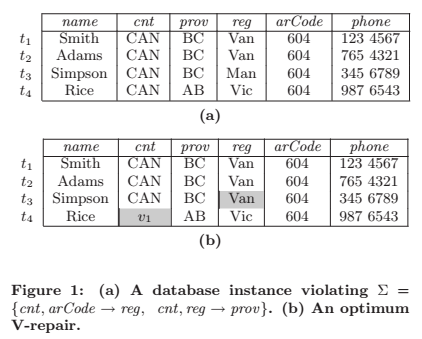


Figure1(a),Figure1(b)分别表示一个数据库实例，和一个优化的V-repair方案(属性分别为name,coutry(cnt),province/state(prov),region(reg),area code(arCode),phone)。该数据库实例违背了函数依赖。其中，Figure1(b)的V-repair反映了两个必要的值修改。其中一个修改时将reg中的’Man’更改为正确的值’Van’，另一个修改是将值’CAN’更改为(即一个可行的方案是将属性cnt的该位置处的值修改为另外一个值)。其中的语义是取一个当前数据库实例中的该属性可取的值之外的任意值(此时的意思是可以取除’CAN’之外的其他任意可取值)。

### V.算法的结论

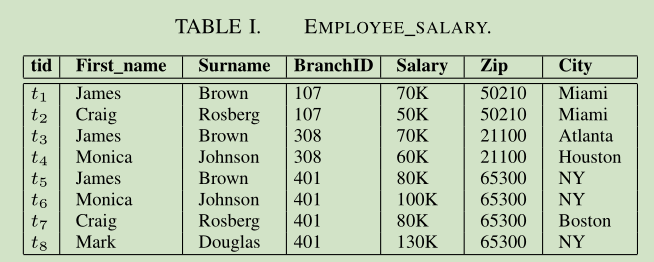
1. 该算法主要针对违背函数依赖的修复进行研究，并提出了一个根据值修改的数目和被更改的元组的权重相关的修复模型。
2. 该算法虽然比以往的修复算法更简洁。
3. 我们提出了一个多项式时间算法，当函数依赖的集合是固定时，它能够产生一个关于最优修复的常数因子的近似修复算法，且该算法也能扩展到解决违背条件函数依赖问题。
4. 未来可以通过结合用户的背景知识与偏好等，来设计出更好的算法；用于解决条件函数依赖的算法还能做更广泛地扩充。

## 9.Continuous Data Cleaning

### I. 动机

对于现代商业应用程序来说，其数据和函数依赖都在不断的变化。在这种动态环境中，即使数据或函数依赖发生了变化，我们也很难发现。因此，这就很容易造成数据的不一致性问题。

为了解决这种动态环境中的数据不一致性问题，我们提出了一个数据清洗框架。在该框架中可以进行三种类型的修复：数据修复、函数依赖修复、数据和函数依赖一起修复。下面举两个例子，来进行具体说明。



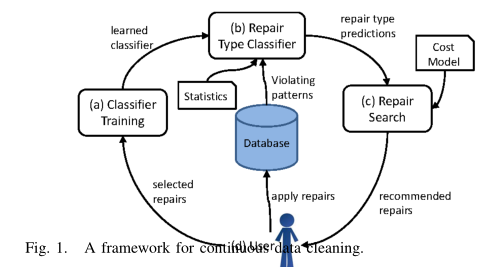
对于表I和函数依赖 ，我们从表I中可以看出[t1,t3,t5]、[t2,t7]、[t4,t6]均违反了函数依赖F1，此时我们发现只要在约束条件的左边加上一个BranchID，就能保证数据的一致性。这时，我们采用的就是函数依赖修复。



然后我们考虑表I和函数依赖 ，我们从表I中可以看出[t3,t4]、[t5,t6,t7,t8]均违反了函数依赖F2。对于[t3,t4]，我们发现其属性Zip的值都是21100，但属性City的值却不同，此时只需把其中一个属性值改为和另一个相同即可（如把t3.City改为Houston，此时t3和t4的City值都是Houston）。同样的对于[t5,t6,t7,t8]，其Zip的值都是65300，只有t7属性City的值和其余3个不同，此时可把t7.City改为NY。此时，数据一致性就得到了恢复，我们采用的方法就是数据修复。



### II.系统结构



如上图所示，我们的框架由四个阶段（a、b、c、d）组成。在第一阶段（a），使用被用户验证过的修复我们训练出了一个概率分类器,这些修复结合了用户和应用程序的偏好。在第二阶段（b），我们使用一个修复类型分类器来对输入的违反模式进行修复类型的预测，本文中提出了7种修复类型，分别是：1NOT REPAIRED、2REPAIRED COMPLETELY BY FD REPAIR(S) 、3REPAIRED COMPLETELY BY DATA REPAIR(S)、4REPAIRED COMPLETELY BY DATA AND FD REPAIR(S)、5REPAIRED PARTIALLY BY FD REPAIR(S) 、6REPAIRED PARTIALLY BY DATA REPAIR(S)、7REPAIRED PARTIALLY BY DATA AND FD REPAIR(S)。为了进行修复类型预测，我们计算了一组（22个）关于数据和约束的统计量，这些统计量能较好的描述违反模式的性质。通过计算这组统计量，我们的分类器会产生修复类型（数据修复、函数依赖修复、数据和函数依赖混合修复）预测。对于预测出来的修复类型，我们的分类器再计算其概率。然后我们就得到了一个最终的修复类型（上面7种修复中的一种），分类器把最终的修复类型传递给一个修复搜索算法，我们就进入了第三阶段（c）。在该阶段，利用预测出来的修复类型，修复搜索算法可以缩小修复搜索空间，修复搜索算法会根据一个代价模型来决定使用一个最好的修复来解决不一致性问题。修复搜索算法会给出一个数据修复和（或）函数依赖修复的集合给用户，这样进入第四阶段（d）。在这个阶段中，用户将决定哪一个修复被应用。被应用修复的数据会再次被分类器（b）作为训练集进行训练，重复整个过程。

### III.关键技术

A.修复统计量

在以往的工作中，我们发现统计量可以提高数据清洗的质量。在动态环境中，数据和函数依赖都在不断变化，我们可以使用统计量来捕获这种变化。因此，我们提出了一组修复统计量来提高预测的准确性。

对于关系（M个属性），我们定义表示该关系的一个元组集合，表示元组的数量，为该关系的一个函数依赖集。

数据修复可以看作是模式替换，记作。

函数依赖修复可以看作是函数依赖替换，记作。

对于函数依赖，模式，其中表示违反函数依赖的元组集合。

我们使用来表示违反函数依赖F的元组数，表示包含模式p的元组数。

**关于函数依赖F的统计量**

1. 

**违反函数依赖F的模式p统计量**

1. 
2. 

**遍历所有的函数依赖获得关于模式p的统计量**

1. 对于一个函数依赖，我们定义，利用这个函数，我们计算函数依赖集中所有描述性统计量：









这里std是标准偏差函数。

**遍历所有可能的约束修复获得关于模式p的统计量**

5）函数依赖的函数依赖修复集合为，对于任意，我们定义如下函数。

我们注意到，0意味着函数依赖修复修复了所有包含模式p的元组。我们定义如下统计量来概括这个函数的功能：







我们没有给出max形式的统计量，因为这种函数依赖修复了很少（或零个）违反F的模式，对分类器几乎不能提供有用的信息。

**遍历函数依赖F的所有数据修复获得关于模式p的统计量**

6）对于函数依赖F，我们定义，基于频率的熵统计量如下：









7）对于函数依赖F，我们定义，基于频率的熵统计量如下：









8）对于函数依赖F，我们定义，基于频率的熵统计量如下：









B.逻辑回归

在逻辑回归模型下，一个给定类别的概率被如下函数给出：



这里是一个和具有相同维度的权重向量（每一个统计量都对应一个权重），是偏差项。

我们定义，表示模式p的类别。那么目标类别的概率为：



通过最小化交叉熵，我们可以让尽可能的匹配：



其中。

为了解决各个类别权重的不平衡问题，我们重新定义交叉熵项的权重：

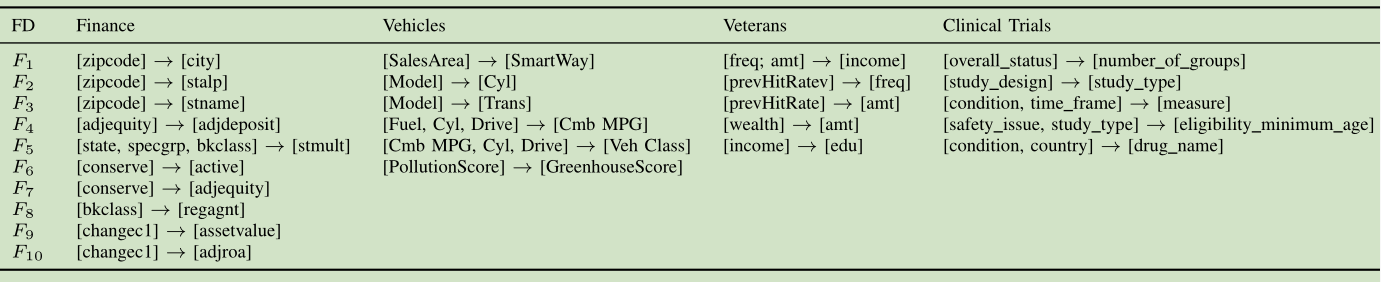


这里，这样就保证每一个类别的权重都相等。

我们使用随机梯度下降的方法来最小化。

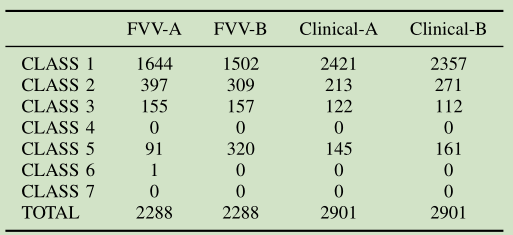
### IV. 应用

我们使用两个真实的数据集合进行实验。第一个数据集（FVV）包含财政、车辆和退伍军人数据集。财政数据集包含27602个元组（25个属性，10个函数依赖）。车辆数据集包含2294个元组（18个属性，6个函数依赖），退伍军人数据集包含95412个元组（9个属性，5个函数依赖）。第二个数据集包含来自加拿大、德国、意大利和日本四个国家的临床试验数据集。我们只使用了这个数据集的一个子集，这个子集包含34818个元组（29个属性，5个函数依赖）。关于每个数据集的函数依赖见下表：



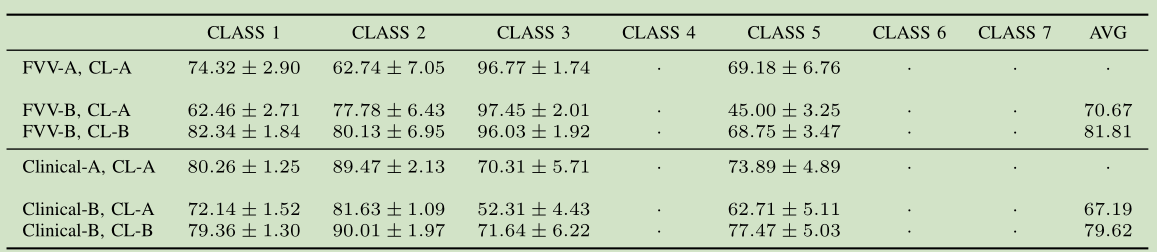
为了研究这个框架的有效性，在每一个数据集上我们训练并评估两个分类器模型。当我们的框架一开始启动时，我们不可能利用被用户验证过的修复来训练分类器。为了解决冷启动问题，我们利用Chiang和Miller提出的修复算法的原始版本产生初始的修复集合（集合A）。使用集合A作为最基本的事实，我们训练出分类器CL-A，并把该分类器的预测结果传递给分类修复搜索算法，这个算法将产生新的修复集合，这些修复被提高给用户进行验证，并产生被用户验证过的修复集合B，我们利用集合B重新训练出分类器CL-B，我们利用集合B的子集来分别评估分类器CL-A和CL-B。

对于包含财政、车辆和退伍军人的数据集FVV和临床试验数据集Clinal，通过实验我们得出不同修复类别（1~7）在这两个数据集的不同版本（A和B）的分布。数据集FVV和Clinal的版本有FVV-A、FVV-B、Clinal-A、Clinal-B。具体分布如下：



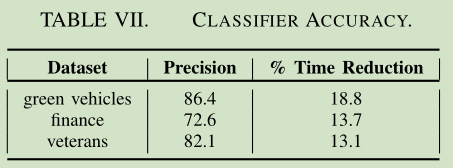
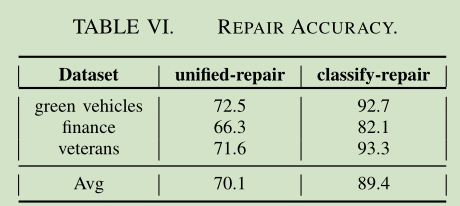
从数据集我们可以看出，对于数据集FVV，无论是FVV-A还是FVV-B，模式的总数是相等的，都是2288。对于每一个类别，除了类别4、6、7基本没有之外，其他的类别除了类别5呈明显增加趋势，其他类别基本都呈下降趋势。这就说明与修复搜索算法初始化的数据修复集FVV-A相比，用户更倾向于用函数依赖修复来修复数据集。

我们利用集合B重新训练出分类器CL-B，我们利用集合B的子集来分别评估分类器CL-A和CL-B的准确性，实验结果如下：



我们可以看出在两个数据集上分类器CL-B比分类器CL-A的准确性要提高10%左右。

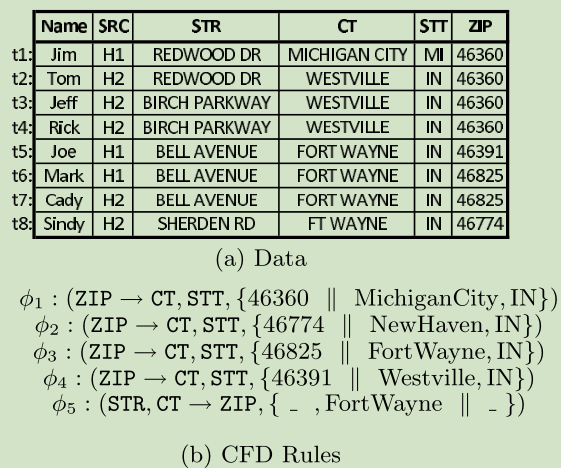
接下来，我们还统计了分类修复算法（classify-repair）和统一修复算法（unified-repair）的准确性。对于分类修复算法，我们还进一步分析了修复准确性提高的同时，运行时间方面改善效果。具体结果如下：



从上表可以看出，分类修复算法（classify-repair）不仅在准确性上比统一修复算法（unified-repair）提高了20%左右，在时间效率方面也提高了13%~19%。

## 10.Guided Data Repair

### I. 动机

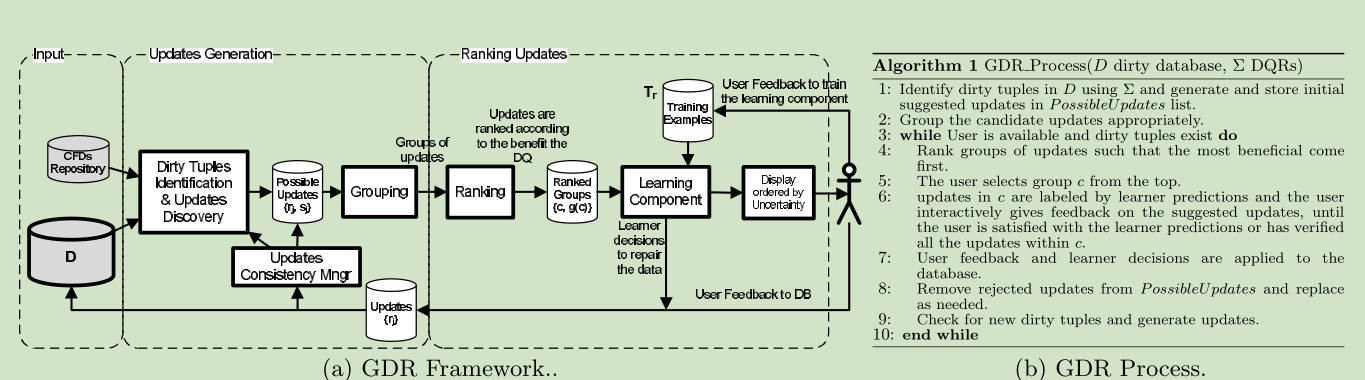


在这篇论文中，我们提出了一个有指导的数据修复框架。在数据修复过程中，我们把用户的反馈和现有的自动修复技术相结合，来提高数据质量。每次产生的自动更新都会提交给用户，根据用户的反馈结果（确认、拒绝、重新训练），我们来决定是否把这个更新应用到数据库中进行数据修复。

考虑上表关系Customer(Name,SRC,STR,CT,STT,ZIP)，该表表示个人地址信息，各个属性依次表示姓名、数据来源、街道（Street->STR）、城市（City->CT）、国家（State->STT）和邮编（zip code->ZIP），我们使用条件函数依赖（CFD）作为数据质量规则，条件函数依赖（CFD）由两部分构成：标准函数依赖（形式为：X→Y）和表模式。根据（b）给出的条件函数规则，我们可以验证（a）中的数据全部违反了规则。为了让用户更有效的参与到修复过程中，修复算法产生的更新最好以分组的形式提交给用户来验证。

在上表中，我们假设清洗算法产生了两组更新。第一组，替换元组t2、t3、t4的属性CT为Michigan City。第二组，替换元组t5和t8的属性ZIP为46825。让我们进一步假设，用户能够获得用户反馈正确的值，也就是说，用户能够确认替换属性CT为Michigan City对于元组t2和t3是正确的，但对于元组t4是不正确的，替换ZIP为46825对于元组t5是正确的，但对于t8是不正确的。在这种情况下，因为第一组有更多的正确更新，所以第一组能以更快的速度收敛到清洁的数据库，第二组将不会有这么快的速度。

### II.系统结构



上图（a）给出了数据清洗的框架，图（b）给出了数据清洗的过程。GDR引导用户把他们的注意力集中在对系统提供的更新给出反馈，这些反馈能快速的提高数据质量，与此同时，用户也引导系统自动识别和应用这些更新到数据上。当数据库中有脏数据且用户能够给出反馈时，这个不断的反馈过程就会一直运行，图（b）中的3-10步给出了描述。

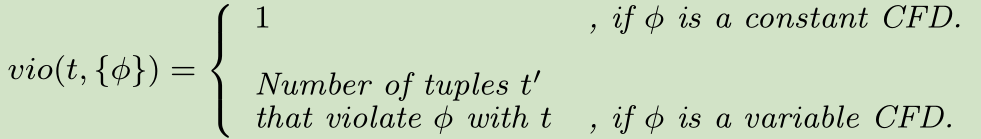
步骤1中，违反规则的脏数据都被识别出来，并且一个修复算法将生成候选更新。在步骤2种，为了便于用户对这些更新进行批量审查，我们以某种方式对这些更新进行分组。在步骤3-10中，一开始就对更新组进行排名，这样我们可以以最快的速度使数据库达到清洁状态。然后用户取出更新组中的第一组，用交互式主动学习的方式对这组更新给出反馈。在步骤7中，来自用户和学习组件的决策都被应用到数据库。在步骤8种，通过替换被拒绝的更新和由于更新被应用到数据库产生的新的脏数据的更新，候选更新列表会被修改。获得用户反馈后，违反的元组会被一致性管理器重新计算，并且会产生新的更新。我们的假设是，如果每一个元组的属性都被用户验证了，那么最终的数据是规则一致的。这就保证了我们始终朝着最终一致的数据库前进，并且这个过程最终会停止。

### III.关键技术

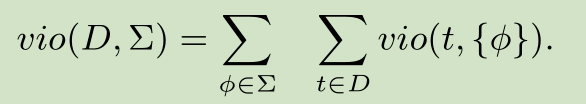
A.基于VOI的排名

我们利用决策论中的概念-信息价值（value of information-VOI）来对产生的更新组进行排序。VOI是对未知信息价值进行量化的一种方式。VOI的核心是损失函数，这个函数量化了当前数据库的数据质量相比期望数据库的距离。

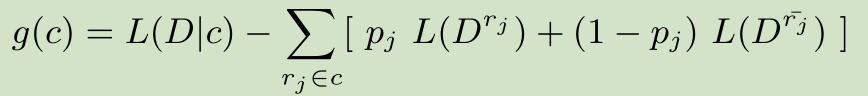
给定数据库D和一个条件函数依赖，我们定义元组t违反了，表示为，具体定义如下：



因此，给定条件函数依赖集合，数据库D的总违反数是：

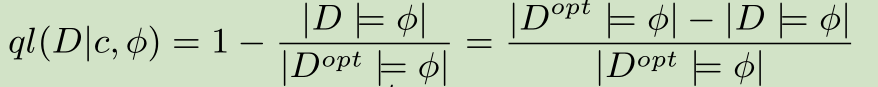


给定一个数据库D和一组更新，对于更新，用户可能将更新至数据库D或不更新，我们分别用和表示这两个数据库。我们假设用户以概率接受，那么期望的数据质量损失为：。如果我们进一步假设所有的组内更新都是独立的，那么对于更新通过用户反馈所获得的更新收益为：

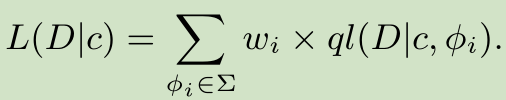


这里是在给定更新的前提下的数据质量损失。

数据质量损失是和满足条件函数依赖集合的元组数成反比的。为了计算数据质量损失，我们首先应该计算违反每一个条件函数依赖的质量损失，这里。我们假设是清洁的数据库，那么*ql*可以表达如下：



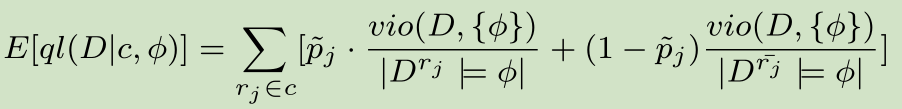
这里，和表示数据库D和中满足规则的元组数，因此，可以按如下公式计算：



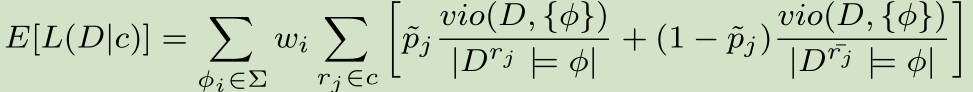
这里是用户定义的对于规则的权重。在我们的实验中，我们定义权重，其中表示落入规则上下文的元组数。直观的解释就是越多的元组落入规则，那么满足规则就越重要。

为了使用上面的公式，我们面临着两个挑战：（1）因为我们不知道更新的正确性，所以我们不知道概率。（2）我们不知道期望的清洁数据库。

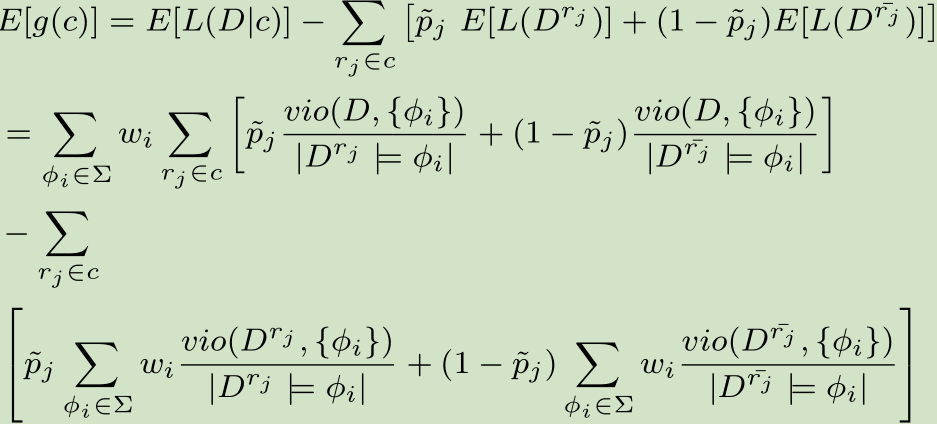
我们假设每一个更新都是独立的，然后我们用下面的公式估计*ql*：



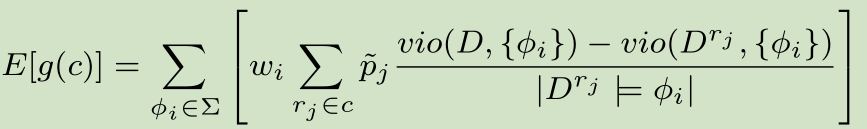
这里，我们使用来估计，那么对于数据D的损失，我们可以用下式估计：



最后，我们可以获得对的估计：



整理后，得：



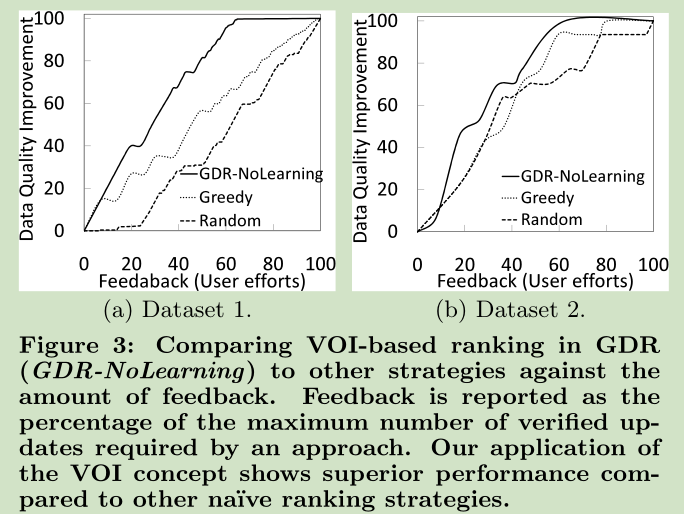
B.主动学习排序

对于更新，我们使用主动学习的方法对进行排序。对于训练集数据我们训练出了k个分类器，其中每一个分类器都是一个随机森林，我们利用k个分类器对每一个更新进行投票，以此来确定每一个更新的不确定性，我们利用熵来计算，具体例子如下：

我们假设和是两个候选的更新，这两个更新更改元组t2和t3属性CT的值为Michigan City，我们建立的模型含有5个随机森林，对于，我们获得预测是{confirm，confirm，confirm，reject，retain}，对于，我们预测的是{confirm, reject, reject, reject, reject}。这样我们对的预测是confirm，不确定性得分为0.86（），对于，最终的预测是reject，不确定性得分为0.45。因为的得分更高，会在之前提交给用户。

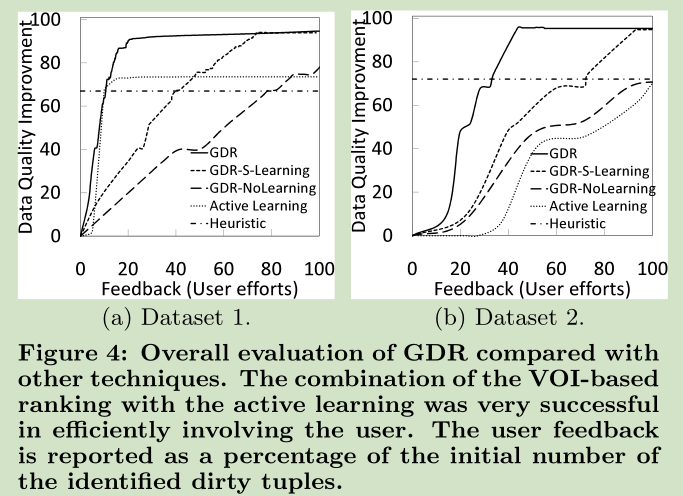
### IV. 应用

在这一部分，我们的目标是评估基于VOI排名机制的GDR有效性。在试验中，我们使用如下技术进行比较：GDR-NoLearning(没有学习组件的GDR框架)、贪心法和随机化的方法。具体对比结果如下：



以上结果清晰的表明了GDR排名组件的重要性和有效性。GDR-NoLearning适合修复非常重要的数据，因为每一个更新在应用之前都会被验证。

下面我们评估使用学习组件来减少用户参与的GDR的性能，在实验中，我们评估了如下方法：GDR、GDR-S-Learning、Active-Learning、 GDR-NoLearning、Automatic-Heuristic，具体结果如下：



实验结果表明了，学习组件对于实现快速收敛到一个较好数据质量的重要性，结果支持了我们最初的假设，脏数据值和正确值之间是有相关性的。基于VOI的排名和主动学习方法的结合优于主动学习机制。

## 11.Descriptive and Prescriptive Data Cleaning

### I.背景

数据清洗技术通常依赖于一些质量规则，来识别违规元组，然后通过使用一些修复算法来解决违规元组。通常情况下，这些与业务逻辑相关的规则，只能通过在多个数据源转换生成一些目标的报告来定义。这样就产生这样一种情况，在报告中检测出的违规是由错误的实际源在时间和空间上解耦得到的，另外在报告上应用这种修复，需要在数据源发生变化时重复更新。最后，即使修复报告是可能的并且价格合理的，对于确定和分析错误的实际来源以供将来防止在目标的违规也是帮助不大的。在本文中，我们提出了一个系统来解决这种解耦。这个系统用质量规则定义了转换器的输出，并且计算出在输出上看到的错误的解释。它不仅执行在目标水平来描述这些错误，还在源水平上规定措施来解决这些错误。我们提出了可扩展的技术来检测，传播，并解释错误。

### II.介绍

常见的解决脏数据的方法是在目标数据库上应用一组数据质量规则或约束，从检测，并到最终修复错误的数据。修复步骤一般是清洗脏数据通过产生一组在D上方的更新产生一个满足规则的新的数据库D0。

为了解释这些错误，我们采用了总结在数据库谓词方面的错误的方法。对于有的属性涉及违规的所有元组，但也包括并不一定是错误的元组。在许多情况下，只更新一个错误的元组（元组的集合），就足以使数据库成为一致的状态。例如，修复算法将确定在报告中tb.Grd作为一个错误的可能。因此，通过更新tb.Grd，两个违规将被删除，限制错误的元组可以引导我们得到一个更精确的解释。如果我们相信tb.Grd是错误的属性，在这个例子中[T:Region = US^T:Shop = NY 1] 是比较特殊的。解释错误数据的进程是双重的，识别一组可能的错误元组，并且找到简洁的描述汇总这些错误并能够由用户或其他分析层利用。

我们将重点介绍错误被确定在不同空间和延后的阶段，而不是错误刚刚出现时。由于在报告中检测的违规实际上是由在早期错误悄悄引起的。从源头，传播这些错误从更高的水平变换到更深层的源头可以帮助识别错误的来源并在规定操作予以纠正。我们要追溯促成所述问题的元组在目标中。通过从所有源中移除这些元组，违规将被移除。正如我们前面提到的，tb是由修复算法确定了错误的元组。基于以上我们设计了一个支持描述和说明的数据清洗系统。DBRx用质量规则定义转换器的输出并且计算出错误的解释。鉴于转换方案（来源SI，1<I<N，和查询Q）和一组质量规则，DBRx计算出一个包含不满足规则的元组的违规表。但是建立这个DBRx系统还有以下一些挑战。

### III.错误判定和检测

设S={S1，S2，….Sn}是n个源关系的集合，对于每个源关系Si有di个属性，R是由S产生的目标模式。转换器是由S产生的一个特殊实例T包含R的t个属性所构成的一组查询集合。Detect(T)确定在T中的属性不满足规则，并且把它们存储在违规表VT中。Error (V (T))返回对于VT最可能的错误属性，并且把它们存储在错误表ET中。尽管Detect(T)含有明确的语义，在Error中需要一些说明。我们的方法可以应用任何有效的修复算法，我们的目标是用简洁的解释描述有问题的数据。不幸的是，所有的错误必须被覆盖，并且不能覆盖没有错误的数据，最坏的情况下，准确的结论是不存在的，在其他情况中错误的个数等同于查询的个数，导致解释难以被理解。为了增加灵活性，我们放弃对于结论精确性的严格要求，允许它覆盖一些正确的元组。对于查询我们引入了权重的概念，依靠正确和错误元组的个数。这个权重有两个成员，它支持覆盖大量错误的查询来减少查询的个数获得全覆盖，它也支持覆盖了一些正确元组的查询。我们可以这个问题映射到著名的加权集合覆盖问题，这被证明是一个NP完全问题。

给出一组规则的集合，VID代表违反ID，R是规则，tid是元组，ID，ATT是元组的属性名称和值，value TID：在元组中的att，我们表示目标的视图为违规的表。检测可以很容易地获得。A DC指出在同一时间所有谓词不能是真的，否则，我们设定为一个违规。对于P中的规则，当应用于数据时，输出通过代码发射，可以用来提取所需要的输出检测。正如我们在引言中提到的，确定实际的错误的能力可以提高该系统的性能。我们可以在数据修复上依靠数据文献让它作为一种工具来识别数据库中的错误。如果一个元组需要改变使情况一致，那么该源组被认为是一个错误。修复是指在检测的过程中纠正发现的违规，已经提出了几种算法的修复不一致的数据，主要是基于声明的数据质量规则，给出一个特定的违反规则，该函数输出一个更新到数据库，满足确定违规的相应检测。通过启发式算法来计算修复确定元组的最小数目的变化来获得符合规则的例子。

### IV.证据传播

证据传播模块包括两个任务：第一个任务是追溯从目标到源产生的违规，我们通过提出Cui et al实施了逆查询技术。第二个任务是确定怎样通过证据传播违规。对于在违规表中的

每一个元组，我们表示在t中的元组，并且问题元组参与其中。从某些源的元组，这些元组依次被计算。为了解决违规行为，我们认为用删除操作源。然而，如上面所讨论的，我们不想确定最小的有问题的元组。相反的，我们将采取切实可行的方法。给出一个违规v，对于v，我们想要衡量每个有问题的源单元格的值。事实上，并非所有的问题元组对于v都有贡献。我们从敏感性分析中获得我们的定义的贡献和清除分数。我们要通过熟人计算一个模型的灵敏度。在一般情况下，给定的一个的功能，该影响是由输出改变了多少给定输入变量之一的变化所定义。在我们的情况下，该模型是运营商的SQL查询，取一组源元组作为有问题的元组中输入和输出。

相反，贡献得分，去除分数在元组上直接计算。如果违规可以通过除去一个源元组被淘汰独立于其它的元组，那么这样的元组是重要的。这种启发式规则让我们确定，可以通过去除解决它违规元组最小的子集。相反，计算所有子集，检查每个源元组允许快速计算。我们用一个简单的自底向上的算法来计算RSVs。它开始在违规的来源元组。对于每个源关系S和每个问题的元组属于S，它消除s和在从S中的路径上的中间关系至T的查询树。如果违规被删除，我们分配分数1 to s1，否则为0。

### V.错误发现

鉴于目标的违法行为，我们用我们的评分方法以确定在源中最可能的错误。由于目标要正确地从非错误元组中分离出潜在错误的元组，我们使用了最有可能的误差，预期具有较高的分数的直觉。对于每个违规，元组的得分的TOP-K分析可以发现潜在的错误。然而，K表不能适用于所有的情况。我们提出了两种方法来解决这个问题。第一种，我们设计了一个异常值函数分别进行高而得分较低的元组的每个冲突。第二种，我们展示从选址减少问题并应用多项式时间日志正逼近算法来计算可能的误差源。

在一些情况下（如与聚集体的查询），基于我们的评分模型，源在违规的谱系元组组成的一个子集的元组具有高的分数，剩下的，同时具有低的分数。精确测量的元组之间的距离，我们定义它如下。两元组具有较高的分数预计将有一个较小的比一高之间的距离在它们之间的距离评分元组和一个低分之一。我们的目标是获得高和低分元组之间的最佳的分离。我们引入了查找全局最可能的错误的元组的另一种方法。该算法识别元组的全局（累计）贡献（所有违法行为）是显着高于他人，这一全球性的信息会导致更高的精度相比于基于距离误差分离，而是以一个较低的召回如果多于一个元组是参与违反。因为它是更容易从解释中发现比从误差和清洁元组的混合，发现更少的错误。

### VI.总结

给定一个源的视图和一组质量规则来识别违规行为，我们引进了不仅在在目标，还在有问题的数据源的水平的解释。为了使这些解释对于用户来说更容易理解，我们制尽量减少它们的大小，同时保证覆违规的问题。工作的主要包括：（ⅰ）违规在目标级别可以表示为来源问题的证据，（ii）总结这些证据导致的问题的有意义的解释。我们计划通过考虑多层次的变化。

## 12.Sampling the Repairs of Functional Dependency Violations under Hard Constraints

### I.背景

函数依赖（FDS）的违规函数在实践中常见的，经常出现在整合数据的上下文中或是web数据的提取。解决这些违规被称为是具有挑战性，对于各种原因，其中之一是可能的“修复”的指数。以前的工作已经解决了这个问题，或者通过产生一个相对于一些度量是（几乎）最佳的单一修复。或者通过计算一致的答案来选择没有明确生成修复查询的类。在本文中，我们设计了并不限定于查找单个修复或到特定类的查询的新的数据的清洗方法，即，从可能的修复空间采样。我们设计了一些促进方案，从FD修复的空间采样是可取。提出了一类新的有用的修复，并且提出了随机从这个空间采样的算法。我们还给出了如何限制基于该定义输入关系的不可变可信子集的用户自定义的硬约束生成修理的空间。

### II.介绍

函数依赖（FDS）可以被看作是编码数据的语义完整性约束，函数依赖的违规指出了从预期的语义的偏差。可能通过数据质量问题引起的。在实践中，函数依赖倾向于打破从Web集成异类数据或解压后的数据。即使是在传统的DBMS，未知函数依赖可能隐藏在一个复杂的演变模式，数据库管理员可以选择因各种原因不执行一些函数依赖。经常有一个非常大的数量的方式来修改一个表，以便它满足所有需要的函数依赖。一种方法是，删除违规元组（理想情况下，删除了尽可能少的这样的元组），使得剩余满足所有函数依赖。在本文中，我们提出了一种新的方法来修复违规函数依赖，从“适用于”维修空间抽样。我们的技术是互补的，对于现有的数据质量和清洁工具。正如我们展示的，它是在各种实际情况下是有用的，在本文，虽然我们专注于修复函数依赖，我们认为拟议的抽样方法可以为复杂的维修空间其他完整性约束的有效工具。

令R是一组函数依赖的被定义的关系，R的属性记为ATTRS(R)，DOM(A)表示属性的域。R的实例I是一组元组，其中每一个所属的域。我们记DomI (A)为属性值的集合。我们假设，在I中的每一个元组是与标识符t保持不变，即使一些相关联属性一直在变化。一个不一致实例I的修复是另一个满足所有规则的实例I0。我们将只考虑由一组包含I的一组修改集合的修理。我们假设每个属性A∈ ATTRS（R）。

### III.抽取可能的修复

单独从修复空间中抽样，由于违规函数依赖之间的相互作用是具有挑战的。例如，解决了一个违反都可能导致解决其他违规行为的副作用，并可能创造新的违规行为。因此，我们给出了一个算法，从各个基数，设定最低的维修，加上一些设置最小维修空间。需要注意的是，虽然现有的方法可以找到单个近似最优修复可被修改，以生成多个随机修理。对于生成的维修空间，他们不给任何保证。

需要注意的是对于任何两个元组t1，t2，违反了一个函数依赖X → A。满足X但不满足A。足以产生一个修复：我们可以修改T1[A]使其等于T2[A]（反之亦然），或者在t1或t2修改属性B∈X，以至于t1[B]≠t2[B]。概括这一观察，如果一个元组的集合不违反函数依赖。这组集合C∪{C}的一致性，对于任何元组C，总是可以通过修改C来执行（如有必要）。我们的算法是基于这一观察;它维护一组元组的清洗是通过在每次迭代中插入新随机选择的元组，对插入的元组进行有必要的修改。我们首先定义一组清洗单元，实例如下。在实例ID一组元组C是干净的，如果有至少一种修复I1∈Repairs(I)就像∀C∈C; I1(C) = I(C)。

也就是说，一个实例I中的一组元组是干净的，如果在I中元组的值可以保持不变，在获得I的一个修复中。请注意，这是不足以证明C中元组单独的不违反任何函数依赖。例如，考虑，这表明的一个实例中的元组的集合，假设我们需要确定是否显示元组是干净的。虽然显示的元组不违反任何函数依赖。没有修复可能含有其它单元的值与那些单元的当前值无关。这是因为t1[C]=t2[C](by A→C)，和t2[B]=t3[B]表示t2[C]=t3[C](By B→C)但是t1[C]≠t3[C] 为了系统地确定一组元组是否是干净的，我们需要跟踪等价类。我们分别用E等价关系（即一组等价类）。我们表示由EC（E; CI）的等价E 2 E到该元组Ci的所属。我们的意思是在E中取代合并两个等价类，他们由一个新的等价类别等于他们的结合。算法1描述了如何建立一个等价关系E的一组单元C的一个实例I。

### IV.总结

在本文中，我们提出了基于约束的数据库修复新技术，我们在其中产生可能的随机样本修复。我们描述了一个实现该技术的函数依赖背景，给定一个新的修复空间，它结合了两个著名的现有空间的特点。我们也扩大了我们的采样算法以允许指定的一组单元的用户定义硬约束，在修复过程中必须保持不变。我们的实验研究表明，分割输入的实例为可以修复的块独立地导致数量级性能提升。

# 三、结论

函数依赖和条件函数依赖通过对数据一致性的形式化建模, 定义了应用相关的正确数据的模式, 可以检测数据元组粒度的不一致性。它也体现了特定应用数据间的内在语义关联, 通过应用相关的语义约束检测实例层次上的不一致数据。一致性检测和求解极小覆盖等技术有助于提高大规模数据清洗的效率, 为自动实现数据不一致性检测、减少清洗过程的人工干预提供了基础。

数据清洗问题的重要性是不言而喻的。从市场上如此多的相关产品，可以明白这一点。有些人认为数据清洗是一个需要大量劳动力的过程，而且往往过于依赖特定应用领域。其实不然，在数据清洗系统的灵活框架上仍然有很多东西值得研究。在多语言环境中，如何准确地识别多语言文本的相似重复记录，也有很多工作可以做。当前 Web 数据量迅速增长，对Web 搜索引擎返回的结果进行清洗也是一个有价值的问题。随着 XML 数据处理标准的日见成熟，如何定义XML 文档数据的质量标准以及针对它们的清洗过程和针对关系数据的清洗过程的区别，都是值得研究的。有文献提出了XML键的概念，完全有理由相信它们可以促进针对 XML 数据的清洗，正如关系表的键在数据集成中扮演了特殊的角色一样。