UMONS
Université de Mons

Faculté
des Sciences

Mémoire

Data Repairing

Project made by : Maxime Van Herzeele
Academic Year : 2017-2018
Dissertation director : Jef Wijsen
Section : 2^{nd} Master Bloc in ComputerSciences

Faculté des Sciences • University of Mons • Place du Parc 20 • B-7000 Mons

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

De nombreuses institutions et entreprises collectent, stockent et utilisent de nombreuses informations. Ces données peuvent être *erronées* ce qui peut induire en erreur n'importe quelle personne voulant utiliser la base de données. Afin d'éviter ce problème, les données devraient respecter les contraintes d'intégrités. Ces contraintes sont des règles devant être respectées par les données, et n'importe quelle information qui ne les respectent pas est considérées comme étant erronée. Malheureusement, ces contraintes peuvent être imprécises et par conséquent elle peuvent échouée dans la différenciation entre les bonnes données et les données erronées. Pour cette raison, certaines données sont identifiées comme étant des violations de ces contraintes (données erronées) malgré qu'elles ne le devraient pas et d'un autre côté, certaines données ne sont pas identifiées comme étant des violations alors qu'elles le devraient. Ces erreurs à la fois sur les données et sur les contraintes, sont un problème pour quiconque souhaite utiliser la base de donnée.

Par exemple, durant mon stage en entreprise, j'ai pu travailler sur un projet associé de près à une base de données ayant ce problème semblable. Cela a eu un énorme impact sur une partie de mon projet. Le projet de réparation de ces données est prévu pour le courant de l'année 2018.

Le terme *Data repairing* ou réparation de données signifie réparer les données mais aussi réparer les contraintes d'intégrité. Il serait naïf de penser que l'on puisse supprimer des données erronées comme on le souhaite. La perte d'information serait important parce que une telle pratique demanderait d'effacer une ligne complète de la table et ce malgré qu'il n'y ait qu'une seule erreur dans la ligne. En outre, les contraintes d'intégrité peuvent aussi ne pas être correcte ce qui veut dire que l'on pourrait supprimer une ligne ne contenant que des données correctes. Pour cette raison, nous avons besoin de techniques afin de réparer à la fois les données et les contraintes et ce sans perdre trop d'information tout en évitant d'échouer dans la détection d'erreurs dans les données.

Dans cette thèse de mémoire, nous allons analyser le *modèle de réparation θ-tolérant* comme il a été introduit dans un papier scientifique[4]. Dans un premier temps nous allons introduire le concept de *denial constraint*, une forme de contraintes d'intégrité qui va nous aider à définir et comprendre le concept du modèle de réparation θ-tolérant. Nous allons également introduire quelques bases de données que nous utiliserons pour illustrer les différentes notions que nous allons aborder. Ensuite, nous allons présenter une implémentation du modèle θ-tolérant. Et enfin, nous terminerons par une analyse des performances de l'implémentation du modèle.

Chapitre 2

Les contraintes d'intégrité

Dans ce chapitre, nous allons rappeler quelques notions bien connues mais nous allons également introduire de nouveaux concept. Dans un premier temps nous allons introduire quelques bases de données que nous utiliserons en tant qu'exemple pour expliquer et illustrer de nombreuses propriétes et définitions. Ces bases de données suivent le modèle relationnel qui a été introduit par E.F. Codd [2]. Ensuite nous allons travailler sur les contraintes d'intégrités et nous allons introduire un nouveau type de contrainte appelé *denial constraint*. Nous allons expliquer plusieurs caractéristiques et propriétés de ces contraintes et expliquer pourquoi nous n'utilisons pas une forme plus conventionnel de contrainte, comme par exemple les dépendances fonctionnelles.

2.1 Base de données

Dans cette section nous allons présenter des bases de données que nous allons utiliser comme exemple dans cette thèse de mémoire. Nous utiliser ces bases de données pour illustrer le modèle de réparation de données θ-tolérant ainsi que d'autres notions que nous définirons.

La première base de données est tirée de l'article principal utilisés dans la bibliographie de cette thèse [4].

	Nom	Anniversaire	Num_Tel	Année	Revenu	Taxe
t1	Ayres	8-8-1984	322-573	2007	21k	0
t2	Ayres	5-1-1960	***-389	2007	22k	0
t3	Ayres	5-1-1960	564-389	2007	22k	0
t4	Stanley	13-8-1987	868-701	2007	23k	3k
t5	Stanley	31-7-1983	***-198	2007	24k	0
t6	Stanley	31-7-1983	930-198	2008	24k	0
t7	Dustin	2-12-1985	179-924	2008	25k	0
t8	Dustin	5-9-1980	***-870	2008	100k	21k
t9	Dustin	5-9-1980	824-870	2009	100k	21k
t10	Dustin	9-4-1984	387-215	2009	150k	40k

TABLE 2.1 – Base de données de l'article principal [4]

.

La seconde base de données que nous allons utiliser est inspiré d'une expérience personnelle. Lors d'un stage en entreprise, j'ai pu travailler sur un projet lié à une base de donnée contenant des données erronées. Ces données ne pouvant pas être utilisé en dehors de l'entreprise, nous utiliserons une base de données reprenant l'idée générale. C'est une table appelée 'Personne' contenant différentes informations basiques sur des personnes en Belgique[1] .

- **NISS :** Le numéro national de la personne. Un numéro national est unique. En règle général, un NISS est formé de la manière suivante : [1]
  - Il commence avec la date de naissance de la personne dans un format YY-MM-DD. Des exceptions existe pour les étranger (c'est à dire des personne n'ayant pas la nationalité Belge) mais nous n'allons pas considérer ces cas. En effet ces cas peuvent être difficile à comprendre et ne sont aucunement intéressant pour la suite.
  - Le nombre composé du septième, huitième et neuvième chiffres est pair pour les hommes et impair pour les femmes
  - Le nombre composé des deux derniers chiffres est le resulat de is $n \mod 97$ avec n le nombre formé des 9 premiers chiffres
- **Nom :** Nom de famille de la personne.
- **Prénom :** Prénom de la personne.
- **Nai_Date :** Date de naissance de la personne dans le format DD-MM-YYYY.
- **Dec_Date :** Date de décès de la personne dans le format DD-MM-YYYY.
- **Etat_Civil :** État civil courant de la personne, celui ci doit être parmi les suivants : (célibataire, décédés, marié, divorcé, décédé, veuf)

[1] Les données sont fictives

— **Ville** : La ville où la personne vit.

— **Code_Post** : Le code postal de la ville.

— **Salaire** : Le salaire perçu par la personne en une année.

— **Taxe** : Le montant de taxe payé par la personne en une année.

— **Enfant** : Le nombre d'enfant que la personne a à charge.

	Niss	Nom	Prénom	Nai_Date	Dec_Date	Etat_Civil	Ville	Code_Post	Salaire	Taxe	Enfant
t1	14050250845	Dupont	Jean	14-05-1902	18-05-1962	décédé	Ath	7822	25k	4k	2
t2	08042910402	Brel	Jacques	08-04-1929	09-10-1978	décédé	Schaerbeek	1030	100k	8k	1
t3	45060710204	Merckx	Eddy	07-06-1945	null	décédé	Schaerbeek	1030	125k	9k	2

TABLE 2.2 – La table Personne

.

2.2 Contraintes sur les bases de données

Les bases de données devraient n'accepter que des valeurs qui respectent certaines normes en relation avec la base de données. Ce serait un problème si on pouvait ajouter n'importe quelle valeur à chaque colonne d'une base de données. Pour eviter ce problème nous avons recours à des règles sur les bases de données. Ces règles sont appelées *contraintes d'intégrité* et fonctionnent de la manière suivante : Si un tuple t respecte toutes les conditions alors les données sont acceptables. Sinon t n'est pas correcte et au moins une des valeurs du tuple est erronée.

Le modèle relationnel des bases de données introduit la notion de *dépendance fonctionnelle* :

Definition 1. Une **dépendance fonctionnelle (DF)** est une expression $X \rightarrow Y$ avec $X, Y \subseteq sort(R)$ et où $sort(R) = \{A_1, A_2, ..., A_n\}$

En d'autre mots, la contrainte $X \rightarrow Y$ signifie que pour une valeur spécifique de X, il n'y a au plus une valeur possible pour Y. Si la DF est respectée sur la relation R, nous pouvons dire que R satisfait la DF. Prenons quelques exemple sur la table 2.2 :

1. *Un NISS identifie une personne* : En d'autre mot, pour une valeur spécifique du NISS, il n'y a qu'une seule valeur possible pour tout le reste de la table. Cela peut se décrire par la DF suivante : $NISS \rightarrow Nom, Prnom, Nai\_Date, Dec\_Date, Etat\_Civil, Ville, Code\_Post, Sa$
2. *Deux personnes avec le même code postal vivent dans la même ville.* : Pour une valeur spécifique de $Code\_Post$ dans notre table il n'y a qu'une valeur possible de $Ville$. Par exemple si la valeur de $Code\_Post$ d'une personne est '7822', la seule valeur possible pour l'attribut $Ville$ est 'Ath'. La dépendance fonctionnelle dans ce cas est $Code\_Post \rightarrow Ville$.

Si pour chaque tuple de la relation R, la DF τ est respectée, nous disons que la relation R *satisfait* τ. Cela ce note $R \models \tau$. Évidement, certaines bases de données ne contiennent pas qu'une seule contrainte mais plusieurs. Il est important qu'elle soient toute respectée. Définissons cela comme ceci :

Definition 2. Soit un ensemble Σ de DF sur la relation R. On dit que la relation R satisfait Σ noté $R \models \Sigma$ si pour chaque DF $\tau \in \Sigma$, on a $R \models \tau$

Malheureusement les dépendances fonctionnelles sont limitées en terme de puissance. En effet, il existe de nombreuses contraintes que nous ne pouvons pas exprimer avec une DF. Par exemple, si nous souhaitons exprimer le fait que *'Une personne ne peut être née avant sa propre mort'*, nous avons besoin de comparer la Nai_Date et la Dec_Date de la personne et de s'assurer que la date de décès ne soit antérieure à la date de naissance. Les dépendances fonctionnelles ne permettent pas d'utiliser des opérateurs de comparaison, il est donc nécessaire d'exprimer les contraintes d'une autre façon. Pour ce faire nous allons introduire un nouveau type de contrainte qui répondra bien à nos besoins : les *denial constraints*.

2.3 Les Denials constraints

Dans cette section nous allons définir ce qu'est une denial constraint. Nous allons aussi expliquer son utilisation dans les bases de données et nous allons également lister et expliquer plusieurs propriétés que peuvent avoir ces contraintes. Commençons d'abord par définir la denial constraint

Definition 3. Considérons un schéma de relation R avec un ensemble S fini d'*attribut*. Une *denial constraint (DC)* sur l'ensemble S est une fonction partielle qui associe l'ensemble S vers le powerset OP de $\{<, =, >\}$. Nous utiliserons la lettre grecque φ pour représenter une DC

Par définition le powerset d'un ensemble S noté $\mathcal{P}(S)$ est l'ensemble de tous les sous-ensemble de S. Cela inclut l'ensemble S lui même mais aussi l'ensemble vide $\emptyset$. Par exemple le powerset de OP est $\mathcal{P}(OP)$ = $\{\emptyset, \{<\}, \{=\}, \{>\}, \{<, =\}, \{=, >\}, \{<, >\}, \{<, =, >\}\}$. Il existe différentes abréviations pour les éléments de OP, ceux-ci étant répertorié dans la table 2.4. Nous avons eu besoin d'introduire 2 nouveaux opérateur $\top$ et $\bot$, chacun étant l'abréviation pour l'ensemble $\{<, =, >\}$ et $\emptyset$ respectivement. Nous les définissons comme tel : $\forall a, b \in$ **dom**,nous avons $d_1 \bot d_2$ est toujours faux et $d_1 \top d_2$ est toujours vrai. Nous utiliserons la lettre grecque ϕ pour représenter un opérateur.

Pour chaque opérateur dans OP nous pouvons définir son inverse, sa réciproque et son implication. Les valeurs de l'inverse, la réciproque et l'implication de chaque élément de OP se trouve également à la table 2.4.

Definition 4. Soit ϕ un élément de OP

L'inverse de ϕ noté $\overline{\phi}$ est égal à $\{<, =, >\} \setminus \emptyset$

La réciproque de ϕ noté $\hat{\phi}$ s'obtient en inter-changeant < et > dans ϕ

L'implication de ϕ noté $Imp(\phi)$ est un ensemble d'élément de OP tel que pour n'importe quelle valeur a et b, si $a\phi_2 b$ **implique**[2] **toujours** $a\phi_1 b$ alors $\phi_2 \in Imp(\phi_1)$.

Notons que $\forall \phi_1, \phi_2$, si $\phi_2 \in Imp(\phi_1)$ alors ϕ est un sous ensemble de ϕ_2. Par exemple $\neq \in Imp(>)$ et $\{>\} \in \{<, >\}$.

Une DC est un fonction partielle de l'ensemble S des attributs vers OP. Reprenons nos exemples sur les DF et essayons de les exprimer en terme de DC. Pour notre table 2.2, nous avons l'ensemble $S = \{NISS, Nom, Prnom, Nai\_Date, Dec\_Date, Etat\_Civil, Ville, Code\_Post, Salaire$

— *Un NISS identifie une personne.* : $\{(NISS, =), (Nom, \neq)(Prnom \neq), (Nai\_Date \neq), (Dec\_Date \neq), (Etat\_Civil \neq), (Ville \neq), (Code\_Post \neq), (Salaire \neq), (Taxe \neq), (Enfant \neq)\}$

We can express previous constraints in first-order logic :

1. *A NISS identify a person.* : $\forall s \forall t : (R(S) \wedge R(t) \rightarrow s(Niss) = t(NISS))$

[2] Tout tuple qui satisfait $a\phi_2 b$ satisfait $a\phi_1 b$

Élément	Abréviation	inverse	réciproque	implication
$\emptyset$	$\bot$	$\top$	$\bot$	$\{\bot\}$
$\{<\}$	$<$	$\geq$	$>$	$\{<, \leq, \neq, \top\}$
$\{=\}$	$=$	$\neq$	$=$	$\{=, \leq, \geq, \top\}$
$\{>\}$	$>$	$\leq$	$<$	$\{>, \geq, \neq, \top\}$
$\{<, =\}$	$\leq$	$>$	$\geq$	$\{\leq, \top\}$
$\{<, >\}$	$\neq$	$=$	$\neq$	$\{\geq, \top\}$
$\{>, =\}$	$\geq$	$<$	$\leq$	$\{\neq, \top\}$
$\{<, =, >\}$	$\top$	$\bot$	$\top$	$\{\top\}$

TABLE 2.3 – Element de OP, le powerset of $\{<, =, >\}$

2. *Two persons with the same post code live in the same district.* : $\forall s \forall t(R(s) \wedge R(t) \wedge s(Post\_code) = t(Post\_code) \rightarrow s(District) = t(District))$
3. *If someone died the march 18th 1962 , his civil status should be equal to decease.* : $\forall t(R(t) \rightarrow \neg(t.Decease\_date =$'18-05-1962'$\wedge t.Civil\_state = deceased)$.

So it's possible to express more constraints with first-order logic than with FD. For this reason, it would be interesting to define another kind of constraints based on the first-order logic. We define a denial constraint as :

Definition 5. Consider a relation scheme R with S a finite set of *attribute*. A **denial constraint (DC)** over S is a partial mapping from S to the powerset OP of $\{<, =, >\}$. We will use the Greek letter φ to denote a DC.

By definition a powerset of a set S denoted $\mathcal{P}(S)$ is the set of all subset of S. It includes S itself and also the empty set. It exists different syntactic shorthands for elements of OP as you can see in table 2.4. We had to introduce 2 new operators $\top$ and $\bot$ defined as follows : For every $d_1, d_2 \in \textbf{dom}$, $d_1 \bot d_2$ is false and $d_1 \top d_2$ is true. We will use the greek letter ϕ to denote operator.

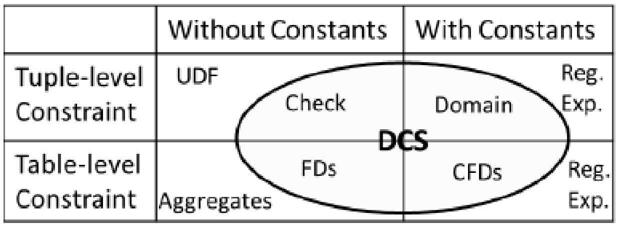
Example 1. Let $S = \{Name, Birtday, CP, Year, Income, Tax\}$. We can have the following denial constraints over S : $\varphi_1 = \{(Name, =), (Birthday, =), (CP, \neq)\}$ and $\varphi_2 : \{(Income, >), (Tax, \neq)\}$

Definition 6. Let $(\textbf{dom}, \leq)$ be a totally ordered domain containing at least two distinct elements. A *tuple over S* is a total mapping from S to $\textbf{dom}$. A *relation over S* is a finite set of tuples over S. Let φ be a denial constraint over S. We say that a relation I over S satisfies φ, denoted $I \models \varphi$, if there exist no two tuples $s, t \in I$ such that for every attribute A in the domain of φ, we have $s(A)\ \theta\ t(A)$ where $\theta = \varphi(A)$. We say that φ is a *valid* DC if $I \models \varphi$.

Example 2. $\{(\mathsf{Income}, >), (\mathsf{Tax}, \leq)\}$ is satisfied by a relation I if there exist no two tuples $s, t \in I$ such that $s(\mathsf{Income}) > t(\mathsf{Income})$ and $s(\mathsf{Tax}) \leq t(\mathsf{Tax})$. In other words, $\{(\mathsf{Income}, >), (\mathsf{Tax}, \leq)\}$ is satisfied by I if for all $s, t \in I$, if $s(\mathsf{Income}) > t(\mathsf{Income})$, then $s(\mathsf{Tax}) > t(\mathsf{Tax})$.

Element	Abbreviation
$\emptyset$	$\bot$
$\{<\}$	$<$
$\{=\}$	$=$
$\{>\}$	$>$
$\{<,=\}$	$\leq$
$\{<,>\}$	$\neq$
$\{>,=\}$	$\geq$
$\{<,=,>\}$	$\top$

TABLE 2.4 – Element of OP, powerset of $\{<,=,>\}$



	Without Constants	With Constants
Tuple-level Constraint	UDF Check	Domain Reg. Exp.
Table-level Constraint	FDs Aggregates	CFDs Reg. Exp.

FIGURE 2.1 – A denial constraint(DC) can express many type of others constraints

If we want to translate φ_1 into first-order logic it will be :

$$\varphi : \forall s,t : R(t) \wedge R(s) \rightarrow (s.Name = t(Name) \wedge s(CP) = t(CP) \wedge \\ s(Birthday) = t(Birthday))$$

In the main article [4] they defined the DC as an expression of the the form $\forall s,t \in R \neg(P_1 \wedge P_2 \wedge ... \wedge P_m)$ where P_i is in the from $s(A)\phi t(A)$ or $s(A)\phi c$ or $t(a)\phi c$ and ϕ in one of the element of OP. This expression is satisfied if if all the predicates are not true at the same time. In others words, this constraints is satisfied if at least one of the P_i is false.

A DC can be oversimplified which means a correct data could consider as a violation. Let's take an example on the table 2.1. If we take the following constraint :

$$\varphi : s,t \in R, \{(Name,=) \wedge (CP,\neq)\}$$

This constraint means that any person with the same *Name* should get the same cellphone number (*CP*), which is incorrect because two different person can get the same name and of course different cellphone number. For example t_1 and t_2 don't respect this denial constraint. In this case (t_1,t_2) are violation of the DC φ. But if we look closely, we can guess it's two different person with the same name. Indeed, they got two different Birthday so their age are different. If we want to improve the accuracy of this constraint, we need to check the *Birthday* attribute :

$$\varphi : \forall s, t : R(t) \wedge R(s) \rightarrow (s.Name = t(Name) \wedge s(CP) = t(CP) \wedge \\ s(Birthday) = t(Birthday))$$

In the opposite of oversimplified, a DC can be overrefined which means a dirty data could be considered as a clean data. An example could be :

$$\varphi : \forall s, t : R(t) \wedge R(s) \rightarrow (s.Name = t(Name) \wedge s(CP) = t(CP) \wedge \\ s(Birthday) = t(Birthday) \wedge s(Year) = t(Year)$$

In this case the Year information is not useful. We don't recognize t_5 and t_8 as a violation with this constraint. The year information represents the year we store the information and of course we can store the information of the same person in two different years.

2.3.1 Some definitions and properties

In this section we will present some definitions and properties on the denial constraint. They will help us in following chapters.

2.3.1.1 Trivial DC

A DC can be useless and always true. Such DCs should not be present because they never identify any violation. In that case we say that the denial constraint is *trivial*. We'll define a trivial DC as :

Definition 7. We say a denial constraint φ is **trivial** if for any instance I of R we have $I \models \varphi$

It's quite easy to discover trivial denial constraint with the following property [3] :

Property 1. $\forall P_i, P_j$ if $\overline{P_i} \in Imp(P_j)$, then $\neg(P_i \wedge P_j)$ is a trivial DC.

In [4], they said for $Imp(\phi)$:

Definition 8. For any two values a and b, if $a\phi_2 b$ always implies[3] $a\phi_1 b$, it means $\phi_2 \in Imp(\phi_1)$.

With this definition, we see that if P_i is true, P_j is false and vice versa.

For example if we have $x = y$ as predicate and we add $x < y$, both the predicates can't be true at the same time. Table 2.5 show for each ϕ the correspondent $Imp(\phi)$.

[3] any tuples who satisfy $a\phi_2 b$ satisfy $a\phi_1 b$

ϕ	=	$\neq$	>	<	$\leq$	$\geq$	$\top$	$\bot$
$\overline{\phi}$	$\neq$	=	$\leq$	$\geq$	<	>	$\bot$	$\top$
$Imp(\phi)$	= $\geq$,$\leq$	$\neq$	>,$\geq$,$\neq$	<,$\leq$,$\neq$	$\geq$	$\leq$	$\top$	$\bot$

TABLE 2.5 – Table of operator, their opposite and their implication

.

If we say for data the table 2.1 :

$$\varphi : t_\alpha, t_\beta \in R \neg(t_\alpha.Tax = t_\beta.Tax \wedge t_\alpha.Tax < t_\beta.Tax)$$

It's a trivial by property 1 It's quite obvious it's impossible that two persons have the same tax rate and one's tax rate is greater than the other. For the rest of this study we won't consider trivial DCs

2.3.1.2 Augmentation

Further in this report, we'll see addition and deletion operations in order to perform data repairing on these constraints. But adding predicates to valid DCs is useless because of the augmentation property [3] :

Property 2. If $\varphi = \neg(P_1 \wedge P_2 \wedge ... \wedge P_n)$ is a valid DC, then $\varphi' = \neg(P_1 \wedge P_2 \wedge ... \wedge P_n \wedge Q)$ is also a valide DC

This property is quite trivial. Remember that φ is a valid DC over I means $I \models \varphi$ so $\forall t \in I$ we have $\neg(P_1 \wedge P_2 \wedge ... \wedge P_n)$ true. Then $\neg(P_1 \wedge P_2 \wedge ... \wedge P_n \wedge Q)$ is true too.

2.3.1.3 Transitivity

In [3] they defined the transitivity of DCs as :

Property 3. If $\varphi = \neg(P_1 \wedge P_2 \wedge ... \wedge P_n \wedge Q_1)$ and $\varphi' = \neg(R_1 \wedge R_2 \wedge ... \wedge R_n \wedge Q_2)$ are both valid DCs and $Q_2 \in Imp(\overline{Q_1})$,
then $\varphi'' = \neg(P_1 \wedge ... \wedge P_n \wedge R_1 \wedge ... \wedge R_n)$ is also a valid DC.

In other words if two **valid** DCs, each with one predicate that can't be false in the same time, then merging those DCs and removing the two predicates will produce a **valid** DC.

It's possible to prove that :

Démonstration.

φ is a valid DC : $\neg(P_1 \wedge P_2 \wedge ... \wedge P_n \wedge Q_1)$ is true.
φ' is a valid DC : $\neg(R_1 \wedge R_2 \wedge ... \wedge R_n \wedge Q_2)$ is true.
$Q_2 \in Imp(\overline{Q_1})$: $Q_1 \oplus Q_2$ is true
then $\neg(P_1 \wedge ... \wedge P_n) \vee \neg(R_1 \wedge ... \wedge R_n \wedge Q_2) \equiv \varphi''$ is true[4] $\square$

[4] $\neg p \vee \neg Q \equiv \neg(p \wedge q)$

2.3.1.4 Refinement

In [4] they define the refinement of a DC as :

Definition 9. φ_2 is a **refinement** of φ_1, denoted by $\varphi_1 \preceq \varphi_2$, if for each $P_i \in pred(\varphi)$ there exists Q_j such that P_i is implied by Q_j.

Example 3. Let $\varphi : \neg(s.Tax \leq t.Tax \wedge s.Income > 25k)$ and $\varphi' : \neg(s.Tax < t.Tax \wedge s.Income > 25k \wedge s.Year = t.Year)$ we have $\varphi \preceq \varphi'$ because $s.Tax \leq t.Tax$ implies $s.Tax < t.Tax$ and $s.Income > 25k$ implies $s.Income > 25k$

As we can see, if we insert an additional predicates in a DC φ , the variant φ' is a refinement of φ

Definition 10. Σ_2 is a **refinement** of Σ_1, denoted by $\Sigma_1 \preceq \Sigma_2$, if for each $\varphi_2 \in \Sigma_2$, there exists a $\varphi_2 \in \Sigma_1$ such that $\varphi_1 \preceq \varphi_2$

As we can see, if you want to change less data, you should refine your DCs with insertion or substitution. For example if our DC is $t_\alpha.Tax \leq t_\beta.Tax$ and we change it to $t_\alpha.Tax < t_\beta.Tax$ you'll change less data.

Chapitre 3

Data Repairing

Errors are frequent in database and these anomalies can make applications unreliable. Some methods detect them but don't repair the detected anomalies. But if one simply filters the dirty data you've detected, applications could still be unreliable. [5] Instead of only detecting errors and filter them, it's better to repair the dirty data.

Definition 11. We define $cell(\varphi)$ as :

$$cell(\varphi) = \{t.A|P : t.A\phi c \in pred(\varphi)\} \cup \{t.A, s.A|P : t.A\phi s.A \in pred(\varphi)\}$$

So $cell(\varphi)$ are all the t.A involved in φ. We can also define $cell(\Sigma)$ as $\cup_{\varphi \in \Sigma}\ cell(\varphi)$. If t.A is not in $cell(\Sigma)$, it cannot be a violation of a constraint and therefore don't need to be repair.

The goal of data repairing is to find a modification I' for an instance I of R, in which all of the violations in the constraints set Σ are eliminated. In other words, we want $I' \models \Sigma$ (I' satisfy Σ). Data repairing process follows the minimum change principle : the data repair I' have to minimize the data repair cost define as [4] :

Definition 12. If I' is a repair for I instance of R by modifying attribute values without any deletion or insertion tuples, the data repair cost is :

$$\Delta(I, I') = \sum_{t \in I, A \in attr(R)} w(t.A).dist(I(t.A), I'(t.A)$$

where :

— $dist(I(t.A), I'(t.A))$ is the distance between two values on $t.A$ in I and I'.

— $w(t.A)$ is the weight of cell $t.A$.

We can see that the cost can be the number of values in $cell(\varphi)$ we changed if we put :

$$dist(I(t.A), I'(t.A)) = \begin{cases} 1\ if I(t.A) \neq I'(t.A)\ (the\ value\ changed) \\ 0\ otherwhise\ (no\ changes\ were\ made) \end{cases}$$

We can put the distance for numerical values on the difference of the two values. For string values we can use the edit distance.

The weight $w(t.A)$ can show the trust of the original value in cell which is subjective or simply be a constant if we don't have a lot of knowledge about the data. For example in the table 2.2, we can expect a child attribute value to be more accurate than a Salary or Tax attribute value.

It is important to notice it is possible we are not able to find any repair I' that can eliminates all the violations. Sometimes there is no value in $dom(A)$ that can fit the constraint. In that case, we use a $freshvariable$ outside the $dom(A)$ in order to extend the domain. A fresh variable is a value that does not satisfy any predicate (all predicates are false), we are sure that we can satisfy the DC. (it's satisfy if at least one of the predicates is false). Each time we are not able to find a value in the $dom(A)$ to repair a cell, we put a new fresh variable fv

Let us take an example on the table 2.1. Let's say our Denial Constraint is the following one :

$$\varphi : t_\alpha, t_\beta \in R, \neg(t_\alpha.Income > t_\beta.Income \wedge t_\alpha.Tax \leq t_\beta.Tax)$$

In other words, we supposed that if someone get a higher income than another person then he should paid an higher tax every year. We have $\langle t_2, t_1 \rangle \not\models \varphi$ because t_1.Income < $t2$.Income and $t2$.Tax $\leq$ $t1$.Tax. Same problem with $\langle t_3, t_1 \rangle$, $\langle t_5, t_1 \rangle$, ect... one can find all the violation in the figure 3.1 A repair I' could be the following one :

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_18	t_9	t_10
Tax	0	fv_1	fv_2	3k	fv_3	fv_4	fv_5	21k	21k	40k

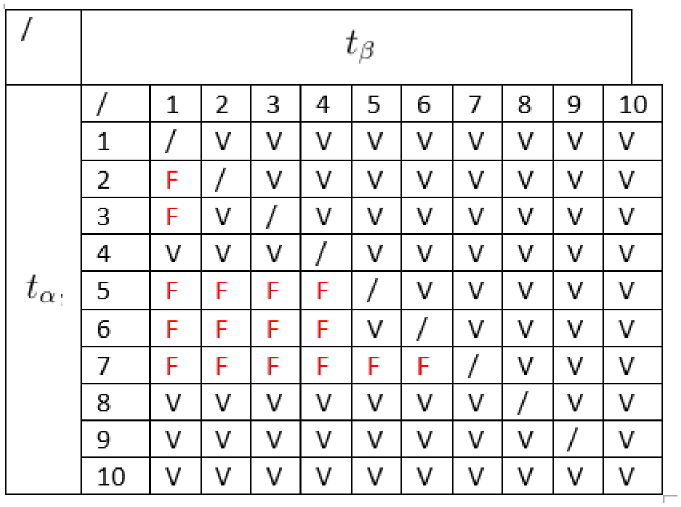
TABLE 3.1 – Example of repair I' with Tax for φ

.

The reason we put fv_1 as Tax value for t_2 is because we knew the following things :

1. $I(t_1.Tax) = 0$ so $I'(t_2.Tax) > 0$ because $I(t_1.Income) < I(t_2.Income)$
2. $I(t_3.Tax) = 3$ so $I'(t_2.Tax) < 3$ because $I(t_2.Income) < I(t_3.Income)$
3. $dom(Tax) = \{0, 3k, 21k, 40k\}$

Because we had no values in the $dom(Tax)$ that would respect 1 and 2, we need to use a fresh variable fv_1 out of the $dom(Tax)$. The same logic can be used to understand why we had to use fv_2 to fv_5. We could put random value instead but it could be a problem if we insert correct value in the future (These correct values could be see as dirty one). We only know that 0< fv_1,fv_2 <3k in order to respect 1 and 2. In the same idea, we have 3k < fv_3,fv_4,fv_5 < 21k.



/	t_β										
t_α	/	1	2	3	4	5	6	7	8	9	10
	1	/	V	V	V	V	V	V	V	V	V
	2	F	/	V	V	V	V	V	V	V	V
	3	F	V	/	V	V	V	V	V	V	V
	4	V	V	V	/	V	V	V	V	V	V
	5	F	F	F	F	/	V	V	V	V	V
	6	F	F	F	F	V	/	V	V	V	V
	7	F	F	F	F	F	F	/	V	V	V
	8	V	V	V	V	V	V	V	/	V	V
	9	V	V	V	V	V	V	V	V	/	V
	10	V	V	V	V	V	V	V	V	V	V

FIGURE 3.1 – All the violation for φ

We can compute the repair cost for Tax in this table. Let's say that :

$$\forall a, b \in dom(A)\ with\ a \neq b. \begin{cases} dist(a,a) = 0 \\ dist(a,b) = 1 \\ dist(a,fv) = 1.5 \end{cases}$$

When we don't change anything, the distance is obviously equal to zero. $dist(a, fv)$ have to be higher than $dist(a, b)$ otherwise the cost for a non-domain value will be lower than a domain value and we want to avoid fresh variable as much as possible [1]. If we had $dist(a, fv)$ lower than $dist(a, b)$, any repair that uses fresh variable outside the domain would be better and of course it's not a good behavior. In our example, with the value said just before, we can compute a $\Delta(I, I')$ = 7.5

3.1 Integrity constraints variations

We saw earlier that a constraint can be overrefined failing to detect some error or in the opposite a constraint can be oversimplified leading to consider some good data as an error. Because constraints can be inaccurate we need to modify them. We'll consider two types of constraint variance : predicate deletion and in the opposite predicate insertion.

When we perform a predicate insertion, some tuples no longer violate the DC. With this variation we can repair a oversimplified constraint but we need to be careful otherwise the DC can be useless. We need to avoid insertion which can lead to a trivial DC or

[1] it's always better to work with values in dom(A) when it's possible

insertion of predicates with obvious constants(like $t_\alpha.Salary$<0 in the table 2.2).

An example of trivial DC is a DC φ with a predicate $P_i : x\phi_i y$ and we had another predicate $P_j : x\phi_j y$ in the DC with $\overline{\phi_j} \in Imp(\phi_i)$.

For overrefined DCs, we need to remove some predicates but we need to be careful. If too many predicates are withdraw, we can get an new oversimplified DCs. The more the predicates are deleted, the higher the data repair cost will be as stated in the Lemma1 in [4]. In the other hand the more the predicates are added, the lower the data repair will be. So if you add more predicates than you remove, there will be less data to change. In the opposite, if you remove more predicates than you add, there will be more data to change. The *data repair cost function* take this effect into consideration. It count positively predicate insertion and negatively predicate addition. For Σ' a variant of Σ in which all $\varphi' \in \Sigma'$ are obtained by insertion or deletion of predicates for corresponding $\varphi \in \Sigma$, in [4] they define the constraint variation cost :

Definition 13. For a variant Σ' of Σ, the contraint variation cost is defined as

$$\Theta(\Sigma, \Sigma') = \sum_{\varphi in \Sigma} edit(\varphi, \varphi')$$

where φ' is a variant of φ and $edit(\varphi, \varphi')$ is the corresponding cost.

the $edit(\varphi, \varphi')$ indicates the cost of changing φ to φ' is defined as :

Definition 14.

$$edit(\varphi, \varphi') = \sum_{P \in \varphi \setminus \varphi'} c(P) + \lambda \sum_{P \in \varphi' \setminus \varphi} c(P)$$

where $c(P)$ denote the weighted cost of predicate P and λ is the weight of a deletion compare to an addition and -1<λ<0.

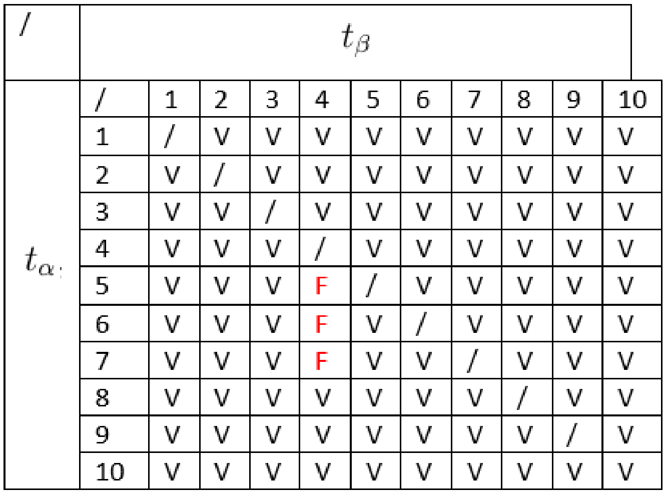
We don't have to use λ =-1 otherwise a deletion followed by an addition would have a cost equal to 0(and it's a bad idea for predicate substitution). For example if we have :

$$\varphi : t_\alpha, t_\beta \in R, \neg(t_\alpha.Income > t_\beta.Income \wedge t_\alpha.Tax \leq t_\beta.Tax)$$

This DC express the fact that if I get a higher income than someone else, i should pay a higher tax rate. We'll change this DC by deleting $t_\alpha.Tax \leq t_\beta.Tax$ and add $t_\alpha.Tax < t_\beta.Tax$. It can express the fact that someone with a very low Income could get a Tax equals to 0.

$$\varphi' : t_\alpha, t_\beta \in R, \neg(t_\alpha.Income > t_\beta.Income \wedge t_\alpha.Tax < t_\beta.Tax)$$

The constraint variation with λ = $\frac{1}{2}$ and $c(P)$ = 0 is : $edit(\varphi, \varphi') = c(t_\alpha.Tax < t_\beta.Tax) + \frac{1}{2}c(\alpha.Tax \leq t_\beta.Tax)$ = $\frac{1}{2}$.



/	t_β										
$t_{\alpha:}$	/	1	2	3	4	5	6	7	8	9	10
	1	/	V	V	V	V	V	V	V	V	V
	2	V	/	V	V	V	V	V	V	V	V
	3	V	V	/	V	V	V	V	V	V	V
	4	V	V	V	/	V	V	V	V	V	V
	5	V	V	V	F	/	V	V	V	V	V
	6	V	V	V	F	V	/	V	V	V	V
	7	V	V	V	F	V	V	/	V	V	V
	8	V	V	V	V	V	V	V	/	V	V
	9	V	V	V	V	V	V	V	V	/	V
	10	V	V	V	V	V	V	V	V	V	V

FIGURE 3.2 – All the violation for φ'

With this new constraint, we got less violations. All the violations can be found the figure 3.2. The modifications we made in table 3.1 are :

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_18	t_9	t_10
Tax	0	0	0	0	0	0	0	21k	21k	40k

TABLE 3.2 – Example of repair with Tax

.

Indeed, a repair for a column with one correction is better than the previous repair with 5 fresh variables (see table 3.1) as a correction. On our new DC, only the $(t_4.Tax)$=3k have to be changed with the value 0. Further in this thesis, we'll see how we can reach these repairs.

When we do constraint modification, we should not consider the case where we delete an entire constraint. We should not remove all the predicates because a DC like $\neg()$ doesn't mean anything. We can't even say if it's always true or always false.

3.1.1 Maximal Constraint Variants

Every constraint variant Σ' with cost $\Theta(\Sigma, \Sigma')$ shouldn't be considered. They're some variation that we're sure they are worst than any another? To perform a prunning of constraints variant that doesn't generate minimum data repair, we'll use the definition of refinement we explained in the previous chapter at the section 2.3.1.4.

Definition 15. [4] We say that a variant φ' of a constraint φ with $\varphi \preceq \varphi'$ is **maximal**, if there does not exist another φ'' such that $\varphi' \preceq \varphi''$ and $edit(\varphi, \varphi'') = edit(\varphi, \varphi')$

Property 4. [4]
For any inserted predicate $P : x\phi y \in pred(\varphi') \setminus pred(\varphi)$, if $\phi \in \{\leq, \geq, \neq\}$, then φ' is not maximal.

This property comes from the $Imp(\varphi)$ definition. If $Varphi_1 \in Imp(\varphi_2)$ then $a\varphi_1 b$ implies $a\varphi_2 b$. $\leq\geq$ and $\neq$ are the 3 operators who implies operator that are no themselves (see table 2.5). With this property we see that it's useless to insert every predicate that you're able to construct. We only have to insert predicates with operators >,<,= when considering variants of φ Let's illustrate that with an example. We'll take two denial constraint on table 2.1 for this.

$$\varphi_1 : \forall t_\alpha, t_\beta \in R, \neg(t_\alpha.Name = t_\beta.Name \wedge t_\alpha.Income = t_\beta.Income \wedge t_\alpha.CP \neq t_\beta.CP)$$

$$\varphi_2 : \forall t_\alpha, t_\beta \in R, \neg(t_\alpha.Name = t_\beta.Name \wedge t_\alpha.Income \leq t_\beta.Income \wedge t_\alpha.CP \neq t_\beta.CP)$$

We know that $\leq \in Imp(=)$ (see table 2.5) for Income so we have $\varphi_2 \preceq \varphi_1$. By the last property we know that φ_2 is not maximal because it contains $\leq$ operator. In this scenario the data repair cost is 7 for φ_2 and φ will get a data repair cost equal to 3.

	Name	Cellphone Number	Income
t1	Ayres	564-389	22k
t2	Ayres	564-389	22k
t3	Ayres	564-389	22k
t4	Stanley	930-198	24k
t5	Stanley	930-198	24k
t6	Stanley	930-198	24k
t7	Dustin	824-870	100k
t8	Dustin	824-870	100k
t9	Dustin	824-870	100k
t10	Dustin	824-870	100k

TABLE 3.3 – Correction with φ_2 : 7 changes needed only for the collumn CP.

.

	Name	Cellphone Number	Income
t1	Ayres	322-573	21k
t2	Ayres	564-389	22k
t3	Ayres	564-389	22k
t4	Stanley	868-701	23k
t5	Stanley	930-198	24k
t6	Stanley	930-198	24k
t7	Dustin	179-924	25k
t8	Dustin	824-870	100k
t9	Dustin	824-870	100k
t10	Dustin	387-215	150k

TABLE 3.4 – Correction with φ_1 : only 3 changes are needed.

.

We see that the refinement got a better data repair cost. It's not a coincidence because the following lemma exist : [4]

Lemma 1. Given two constraints variants Σ_1, Σ_2 of Σ such that Σ_2 is also a refinement of Σ_1, have $\Sigma \preceq \Sigma_1 \preceq \Sigma_2$, is always has $\Delta(I, I_1) \geq \Delta(I, I_2)$, where I_1 and I_2 are the minimum data repairs with regards to Σ_1 and Σ_2, respectively.

As a consequence of this Lemma, any non-maximal set of denial constraint Σ can be removed from the possibilities of good repair

3.1.2 Pruning our candidates

In this subsection we'll focus on removing the constraint variant Σ' that can't generate the minimum data repair. We already have seen that Σ with a non maximal constraint φ can be remove. To go further, we will consider two bounds of possible minimum data repairs cost for the instance I : the lower bound noted as $\delta_l(\Sigma', I)$ and the upper bound noted $\delta_u(\Sigma, I)$. We consider the following property :

Property 5. For two constraints variants Σ_1 and Σ_2 for the instance I of R, if $\delta_u(\Sigma_1, I) < \delta_l(\Sigma_2, I)$ then Σ_2 can be discarded.

It means the worst bound(upper) of repair for Σ_1 is still better than the best bound(lower) of repair for Σ_2, then Σ_2 is useless and can be ignored.

3.1.2.1 Conflict Graph

Now, we'll introduce the conflict graph which can represent the violations in an instance I of R. In the first place, we need to find all the violations and we could get the data repair cost bound from them. We define the violation set as : [4]

Definition 16. The violation set noted as $viol(I, \varphi) = \{\langle t_i, t_j, ...\rangle | \langle t_i, t_j, ...\rangle \not\models \varphi \text{ with } t_i, t_j, ... \in I\}$ is a set of tuple lists that violate φ. The violation set of Σ is $viol(I, \Sigma) = \cup_{\varphi \in \Sigma} viol(I, \varphi)$.

With the conflict hypergraph G we can represent the violations in I. For each violation tuples $\langle t_i, t_j, ...\rangle \in viol(I, \varphi)$ there are an edge for $cell(t_i, t_j, ..., \varphi)$ in G. A good repairing I' consists in correcting the data base to be able to remove all the edges on the graph. The hypergraph of I' should be empty.

Let's take an example on the table 2.1 with a denial constraint we already used :

$$\varphi' : \forall t_\alpha, t_\beta \in R, \neg(t_\alpha.Income > t_\beta.Income \wedge t_\alpha.Tax < t_\beta.Tax)$$

For this relation the violation set is (see Figure 3.2) :

$$viol(I, \varphi') = \{\langle t_5, t_4\rangle, \langle t_6, t_4\rangle, \langle t_7, t_4\rangle\}$$

On our hypergraph, $\langle t_5, t_4\rangle \in viol(I, \varphi')$ consist of $cell(t_5, t_4; \varphi')$ which is equal to $\{t_5.income, t_4.Incom$ We want to remove a vertex, i.e eliminate a conflict. Let's first introduce some definition and a Lemma : [4]

Definition 17. We denote $\min_{a \in dom(A)} dist(I(t.A), a)$ the weight of a vertex t.A, i.e, the minimum cost should be paid to repair t.A.

Definition 18. $\mathbb{V}(G)$ is the **minimum weighted vertex cover** of the hypergraph G corresponding to Σ with weight

$$||\mathbb{V} * (\mathbb{G})|| = \sum_{t.A \in \mathbb{V}(G)} min_{a \in dom(A)} dist(I(t.A), a)$$

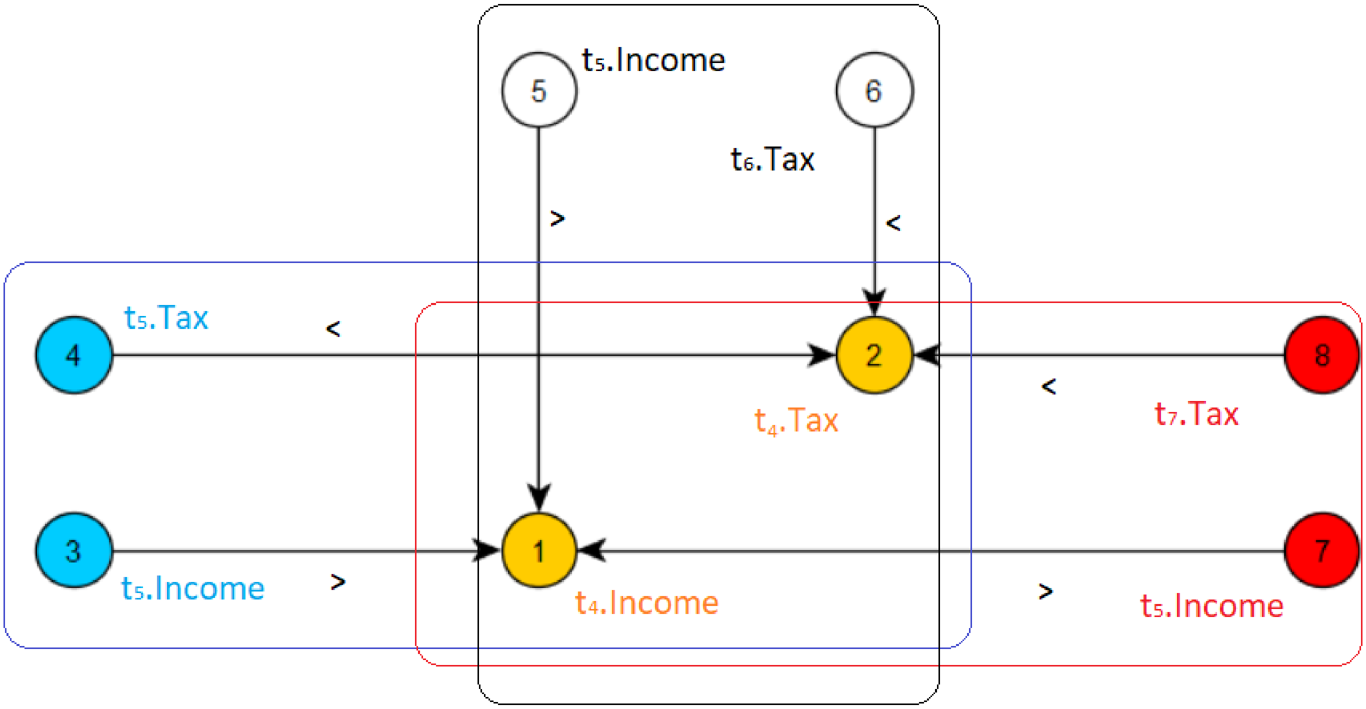


FIGURE 3.3 – Conflict hypergraph for φ

Lemma 2. For any valid repair I' of I, i.e, $I' \models \Sigma$, we have $\Delta(I, I') \leq ||V * (G)||$.

In [4] they define the upper and lower bound of repair cost in this way :

$$\delta_l(\Sigma, I) = \frac{||\mathbb{V}(G)||}{Deg((\Sigma)}$$

$$\delta_u(\Sigma, I) = \sum_{t.A \in \mathbb{V}(G)} dist(I(t.A), fv)$$

If we come back to our example and suppose that we have :

$$\forall a, b \in dom(A) \; with \; a \neq b. \begin{cases} dist(a, a) = 0 \\ dist(a, b) = 1 \\ dist(a, fv) = 1.1 \end{cases}$$

So if each vertex has a weight of 1 (=$dist(a, b)$) and if we put $\mathbb{V}(G)$ = $\{t_4.Tax\}$ we get $||\mathbb{V}(G)||$ = 1. We also have $Deg(\varpi')$ = 4, so with formula we have we know upper and lower bound : $\delta_l(\Sigma, I)$=$\frac{||\mathbb{V}(G)||}{Deg((\Sigma)}$ = $\frac{1}{4}$ = 0.25 and $\delta_u(\Sigma, I) = \sum_{t.A \in \mathbb{V}(G)} dist(I(t.A), fv)$ = $dist(a, fv)$=1.1 .

3.2 θ-tolerant model

In this section we'll talk about the θ-tolerant repair model which is the main models we want to study. θ is a threshold on the variation on the set of constraint Σ, so we don't want a constraint variation cost greater than $\theta : \Theta(\Sigma, \Sigma') \leq \theta$. It helps to avoid any kind of over-refinement and so some undetected dirty data. To avoid the over-simplification and identify correct data as dirty data, the repairing should pursue the minimum change principle. We need to find a repair I' of the original instance I and minimize the repair cost $\Delta(I, I')$.

Finding the best (minimum) θ-tolerant repair is difficult, it's a NP-hard problem. An NP-hard problem is a class of decision problems which are at least as hard as the hardest problems in NP [2]. What we have to remind is it's not possible to resolve it in a polynomial time. Even is the first approach we could made is to get all the constraints variant Σ' and then compute $\Theta(\Sigma, \Sigma')$, look if it's lower than θ and then find the minimum data repair I'. But this way is oviously high in complexity

We saw that we could replace some value by a fresh variable fv. It's better to not turn all of them in a fresh variable mainly because fv is not $dom(A)$ and also a repair like this will never return the optimal repair because we want to minimize the repair cost.

[2] problems whose a solution can be verified as good one in a polynomial time

Now, consider $\mathbb{D} = \Sigma'_1 x \Sigma'_2 x ... \Sigma'_{|\Sigma|}$ where each $\Sigma'_i \in \mathbb{D}$ is a variant of Σ obtained by previous variations. Consider those variations bounded by θ so we have $\Theta(\Sigma, \Sigma') \leq \theta$. The algorithm 1 return the best instance I_{min} for our set of constraint variation Σ. The algorithm is simple. For each Σ_i , if the lower bound is lower than the previous upper bound (remember the property 5). we update the value of δ_{min} if a better repaired instance comes from I_i.

Algorithme 1 : θ-TolerantRepair$(\mathbb{D}, \Sigma, I)$

```
Input   : Instance $I$, a constraint set $\Sigma$, a set $\mathbb{D}$of constraint variants with
          variation bound by $\theta$
Output : A minimum data repair $I_{min}$
$\delta_{min}$ = $\delta_u(\Sigma, I)$
for each constraint variant $\Sigma_i \in \mathbb{D}$ do
    if $\delta_l(\Sigma_i, I) \leq \delta_{min}$ then
        $I_i$ = DATAREPAIR($\Sigma_i, I, \mathbb{V}(G_i), \delta_{min}$
        if $\Delta(I, I_i) \leq \delta_m in$ then
            $\delta_{min}$ = $\Delta(I, I_i)$
            $I_{min}$ = $I_i$
return $I_{min}$
```

To get an example , imagine we have for a $\theta = \frac{1}{2}$, a set of constraint variation $\mathbb{D}$ = $\{\Sigma_1, \Sigma_2\}$ with the first set of constraints $\Sigma_1 = \{\varphi'\}$ and the second set of constraints Σ_2 = $\{\varphi''\}$ with :

$$\varphi' : \forall t_\alpha, t_\beta \in R, \neg(t_\alpha.Income > t_\beta.Income \wedge t_\alpha.Tax < t_\beta.Tax)$$

$$\varphi'' : \forall t_\alpha, t_\beta \in R, \neg(t_\alpha.Income > t_\beta.Income \wedge t_\alpha.Tax = t_\beta.Tax)$$

We already have done the conflict hypergraph for φ in figure 3.3 we also know that $\delta(\Sigma_1, I) = 1.1$

For $\Sigma 2$ we obtain the hypergraph in figure 3.4 (and the violations in figure 3.5) with $\mathbb{V}(G_2)$ = $\{t_2.Tax, t_3.Tax, t_5.Tax, t_6.Tax, t_7.Tax\}$. We have $Deg(\Sigma_2)$ = $Deg(\varphi')$ [3], so we have $\delta_l(\Sigma_2, I)$= $\frac{6}{2}$ = 1,5. Remember that $\delta_u(\Sigma_1, I)$ =1.1, so we have $\delta_u(\Sigma_1, I) < \delta_l(\Sigma_2, I)$ which means we can ignore $\Sigma 2$ and don't call the DATAREPAIR function on it. We will talk about the DATAREPAIR function later.

Let's talk about the complexity. If we say that l is the maximum number involved in a constraint of Σ then we can say that the construction of G_i for each $\Sigma_i \in \mathbb{D}$ cost $O(|I|^l)$. The data repairing algorithm get a complexity in time of $O(|I|^l)$ and the algorithm 1 runs in $O(|I|^l|\mathbb{D}|)$ time. Usualy a denial constraint get 2 tuples [4].

[3] same reasoning : 4 cells involved.

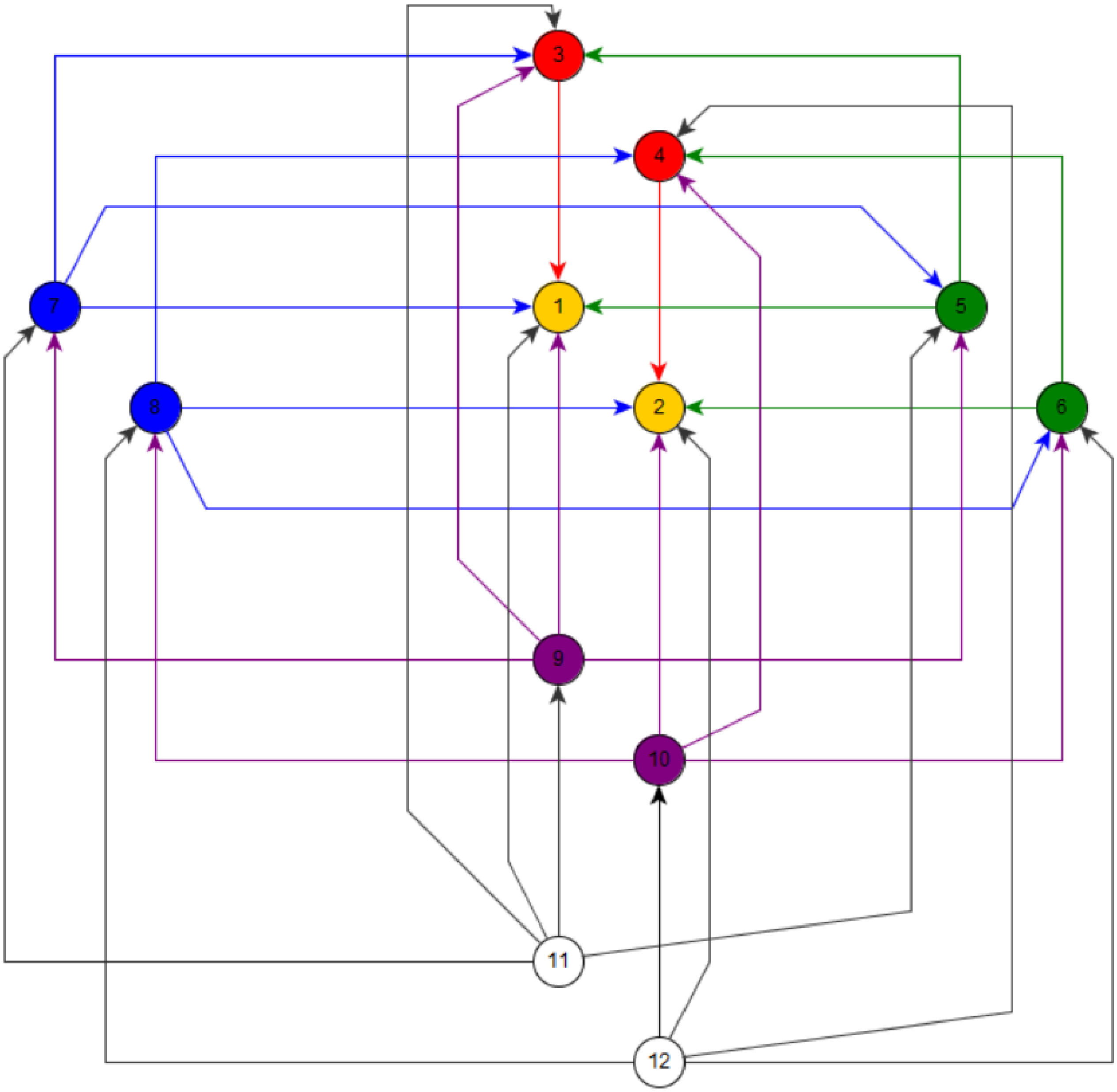
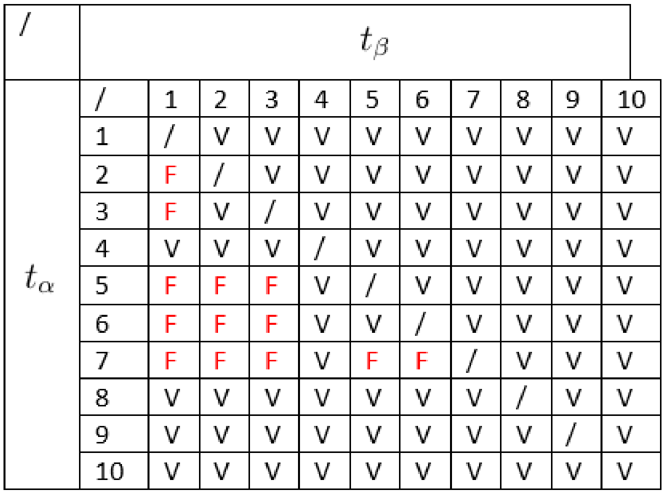


FIGURE 3.4 – Conflict hypergraph for Σ_2 with :
odd number for Tax, even number for Income.
t_1 in yellow, t_2 in red, t_3 in green,
t_4 in blue, t_5 in purple and t_6 in white.



/	t_β										
t_α	/	1	2	3	4	5	6	7	8	9	10
	1	/	V	V	V	V	V	V	V	V	V
	2	F	/	V	V	V	V	V	V	V	V
	3	F	V	/	V	V	V	V	V	V	V
	4	V	V	V	/	V	V	V	V	V	V
	5	F	F	F	V	/	V	V	V	V	V
	6	F	F	F	V	V	/	V	V	V	V
	7	F	F	F	V	F	F	/	V	V	V
	8	V	V	V	V	V	V	V	/	V	V
	9	V	V	V	V	V	V	V	V	/	V
	10	V	V	V	V	V	V	V	V	V	V

FIGURE 3.5 – All the violations for φ''

3.3 Minimum Data Repair and Violation Free

After using the θ-tolerant model, we know which constraint set Σ' derived from Σ we have to use to perform a repairing. But we haven't see how to repair yet. In this section we'll focus on the minimum data repair I' based on the Σ'. To make this we'll use the violation free principle to be sure we don't create any violation after correcting one data. For example, if we put $t5.Tax$ to 22, we solved the violation $\langle t_5, t_4\rangle$ we had with φ' [4]but we introduce a new violation $\langle t_8, t_5\rangle$.

Remember we already said that finding a minimum repairing is NP-hard problem. For this reason we need to make some approximation in order to repair. For the following explanation and definition we'll note $\mathbb{C}$ the selected cells $\mathbb{V}(G)$

3.3.1 Suspect identification

Definition 19. [4] The suspect set $susp(\mathbb{C}, \varphi)$ of a φ is a set of tuple lists $\langle t_i, t_j, ... : \varphi\rangle$ satisfying all the predicates in φ which do not involve cells in $\mathbb{C}$.

and they satisfy the suspect condition :

$$\begin{aligned} sc(t_\alpha, t_\beta, ... : \varphi) = \{&I(v_1)\phi c | P : v_1\phi c \in pred(\varphi), v_1 \in \{C\}\} \cup \\ &\{I(v_1)\phi v_2 | P : v_1 \phi v_2 \in pred(\varphi), v_1, v_2 \in \{C\}\} \end{aligned}$$

Any violation is of course in the suspect list, which lead to the trivial lemma :

[4] remember the $\varphi : \forall t_\alpha, t_\beta \in R, \neg(t_\alpha.Income > t_\beta.Income \wedge t_\alpha.Tax < t_\beta.Tax)$ we used many times

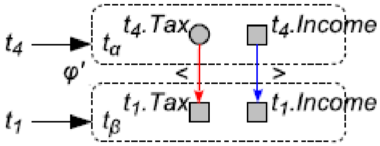


FIGURE 3.6 – Suspect condition represented by blue arrows and repair context represented by red arrows (with inverse operator).

Lemma 3. For any $\mathbb{C}$, it always has $viol(I, \varphi) \subseteq susp(\mathbb{C}, \varphi)$

And by this way, if we catch all the suspect, we also get all the violation.

To explain it, let's return on the example φ' related with the hypergraph at figure 3.3. We will change only $t_4.Tax$ as we made in the table 3.1. So we have $\mathbb{C}$ = $\{t_4.Tax\}$ and $susp(\mathbb{C}, \varphi') = \{\langle t_4, t_1\rangle, \langle t_4, t_2\rangle, \langle t_4, t_3\rangle, \langle t_5, t_4\rangle, \langle t_6, t_4\rangle, \langle t_7, t_4\rangle, \langle t_8, t_4\rangle, \langle t_9, t_4\rangle, \langle t_{10}, t_4\rangle\}$ If we get $\langle t_4, t_1\rangle$, we have $t_4.Income > t_1.Income$ but there is a chance that any change on $t_4.Tax$ leads to a new violation($I'(t_4.Tax) < I(t_1.Tax)$. This is the reason why it's on the suspect list. In the figure 3.6 we have a graph in which every cells not in $\mathbb{C}$ are represented by circles and cell in $\mathbb{C}$ are represented by squares. Red arrows are respected predicates and blue arrows are respected predicates.

3.3.2 Repair context over suspects

We can now define a repair context. The repair contest of a suspect tuple is something that makes sure the suspects will not satisfy the predicates declared on $\mathbb{C}$. The reason why we need it is because a denial constraint needs at least one false predicate for every rows of the database. The repair context takes the inverse operator of predicates in $\mathbb{C}$. In [4], the repair context $rc(t_i, t_j, ... : \varphi)$ of a suspect $\langle t_i, t_j, ...\rangle$is defined as :

Definition 20.

$$\begin{aligned} rc(t_\alpha, t_\beta, ... : \varphi) = & \{I'(v_1)\overline{\phi}c | P : v_i\phi c \in pred(\varphi), v_1 \in \mathbb{C}\} \cup \\ & \{I'(v_1)\overline{\phi}I'(v_2) | P : v_i\phi v_2 \in pred(\varphi), v_1, v_2 \in \mathbb{C}\} \cup \\ & \{I'(v_1)\overline{\phi}'(v_2) | P : v_i\phi v_2 \in pred(\varphi), v_1, \in \mathbb{C}, v2 \notin \mathbb{C}\} \cup \\ & \{I(v_1)\overline{\phi}I'(v_2) | P : v_i\phi v_2 \in pred(\varphi), v_2 \in \mathbb{C}, v_1 \notin \mathbb{C}\}. \end{aligned}$$

Property 6. Any assignment that satisfies all the repair contexts forms a valid repair I' without introducing any new violations, i.e, $I' \preceq \Sigma$

If we continue with our previous example with φ', we have $rc(t_4, t_1 : \varphi')$ = $\{I'(t_4.Tax \geq I(t_1.Tax)\}$, $\geq$ is the inverse operator of $<$ and we only consider predicates of φ' with cells from $\mathbb{C}$ which are red arrows on the figure 3.6. We also have $rc(t_5, t_4 : \varphi')$ = $\{I'(t_5.Tax \geq I(t_4.Tax)\}$. With both of these repair constraint we have $0 = t_1.Tax \leq t_4.Tax \leq t_5.Tax = 0$ which lead to only one possible value : $t_4.Tax = 0$. Remember that previously we put a fresh variable fv instead of 0 (see the table 3.1 in the previous chapter).

We want a repair cost as small as possible, which leads to to minimize the repair cost under repair cost constraint. We have to solve the following problem :

$$\min \sum_{t_i.A \in \mathbb{C}} dst(I(t_i.A), I'(t_i.A))$$

$$under\ the\ constraint : rc(t_i, t_j, ... : \varphi)\ with\ \langle t_i, t_j, ...\rangle \in susp(\mathbb{C}, \varphi), \varphi \in \Sigma$$

But it could be possible we can't assign any value (in our domain) that can fit all the repair context. In these case, we can't put any value except a fresh variable. If we decide to assign a fresh variable fv to a cell can remove every repair context with this cell. The reason is that fv are defined as a way they don't satisfy any kind of predicates which include predicates in repair context. If we are not able to solve our problems i.e we can't found value in dom(A) for a repaired instance I', we'll assign a fresh variable until the problems is solvable. It's better to start by cells with the largest number of appearance in predicates in the repair context. We can say $I'(t.A) = fv$ if $rc(t.A, \Sigma)$ which represent all the repair context declared between a constant or between $t.A$ and $v_i \notin \mathbb{C}$.

If we come back to one of our first example : $\varphi : t_\alpha, t_\beta \in R, \neg(t_\alpha.Income > t_\beta.Income \wedge t_\alpha.Tax \leq t_\beta.Tax)$ For t_2 we have $rc(t_2.Tax, \{\varphi\}) = \{I'(t_2.Tax) > I(t_1.Tax), I(t_4.Tax) > I'(t_2.Tax), I(t_8.Tax) > I'(t_2.Tax), I(t_9.Tax) > I'(t_2.Tax), I(t_{10}.Tax) > I'(t_2.Tax)\}$ In the same way we did for φ', here we have $0 = I(t_1.Tax) < I'(t_2.Tax) < I(t_4.Tax) = 3k$ and there is no value who respect it in dom(A) = $\{0, 3k, 21k, 40\}$. The only possible solution in this case is to assign a fresh variable fv to $t2.Tax$. When we do this, we can remove every red arrows related from the hypergraph. But we didn't solve all of the repair contexts.

3.4 Other repairing

TODO

3.4.1 Holistic data repair

3.4.2 ...

Chapitre 4

Implementation and comparison with others models

TODO

Chapitre 5

Conclusion

TODO

Bibliographie

[1] Description des données du registre national et du registre bcss. https://www.ksz-bcss.fgov.be/sites/default/files/assets/services_et_support/cbss_manual_fr.pdf. accessed : 2018-02-15.

[2] ics relational database model. http://databasemanagement.wikia.com/wiki/Relational_Database_Model. Accessed : 2018-02-13.

[3] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. Discovering denial constraint. Technical report, University of Waterloo and QCRI, 2013.

[4] Shaoxu Song, Han Zhu, and Jianmin Wang. Constraint-variance tolerant data repairing. Technical report, Tsinghua National Laboratory of Information Science and Technology, 2016.

[5] Aoqian Zhang, Shaoxu Song, Jianmin Wang, and Philip S. Yu. Time series data cleaning : From anomaly detection to anomaly repairing. Technical report, Tsinghua National Laboratory of Information Science and Technology, 2017.