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	NumTel	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* :

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

En d'autre mots, la contrainte $X \to Y$ 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 : $NISS \to Nom, Prnom, Nai\_Date, Dec\_Date, Etat\_Civil, Ville, Code\_Post, Sa$
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 $Code\_Post \to Ville$.

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 :

Définition 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*.

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 $\{<, =, >\}$

2.3 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

Définition 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

Définition 4. Soit $(\mathbf{dom}, \leq)$ un domaine totalement ordonné contenant au moins deux éléments distincts. Un *tuple sur* S est une fonction totale de S à **dom**. Une *relation sur S* est un ensemble fini de tuples sur S.

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 2.3. 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 ou θ pour représenter un opérateur.

Expliquons maintenant la sémantique qui se cache derrière la denial constraint.

Définition 5. On dit qu'une relation I sur S *satisfait* la DC φ, noté $I \models \varphi$ si il **n'existe pas** deux tuples $s, t \in I$ tel que pour chaque attribut A dans le domaine de φ, nous avons $s(A)\theta t(A)$ avec $\theta = \varphi(A)$

Prenons un exemple sur la table 2.1, nous avons $S = \{Nom, Anniversaire, NumTel, Anne, Revenu, T$ Une DC pour S est $\varphi = \{(Nom, =), (Anniversaire, =), (NumTel, \neq), (Annee, \top), (Revenu, \top), (Taxe, \top)\}$ Celle-ci est satisfaite par la relation I si il n'existe pas deux tuples $s, t \in I$ tel que $s(Nom) = t(Nom) \wedge s(Anniversaire) = t(Anniversaire) \wedge s(NumTel) \neq t(NumTel) \wedge s(Annee) \top y(Annee) \wedge s(Revenu) \top t(Revenu) \wedge s(Taxe) \top t(Taxe)$.

Soit φ une DC sur S . Nous appellerons *prédicat* P de φ l'expression de la forme (A, θ) avec $A \in S$ que l'on appelle attribut du prédicat et $\theta = \varphi(A)$ que l'on appelle opérateur du prédicat. Soit $pred(\varphi)$ l'ensemble des prédicats de la DC φ. Soit I une relation sur S. Dès lors on peut dire que φ est satisfaite si au moins un des prédicats est faux. Si un prédicat P a pour opérateur $\top$ alors P sera toujours vrai pour tout $t, s \in I$. Dès lors à l'avenir, nous ne noterons plus les prédicats ayant top pour opéraeur par facilité syntaxique. L'exemple précédent s'écrira désormais$\varphi = \{(Nom, =), (Anniversaire, =)$. Si un prédicat a pour opérateur $\perp$, il sera toujours faux. Dès lors $I \not\models \varphi$. La DC $\varphi = \{(A_1, \top), (A_2, \top), ...(A_n, \top)\} \equiv \{\}$ n'est satisfaite par aucune relation excepté par une relation vide.

Si nous prenons I comme étant la table 2.1, nous avons $I \not\models \varphi$. En effet prenons $s = t_2$ et $t = t_3$ nous avons bien $t_2(Nom) = t_3(Nom) \wedge t_2(Anniversaire) = t_2(Anniversaire) \wedge y_2(NumTel) \neq t_3(NumTel)$. On dit que $\langle t_2, t_3 \rangle$ *viole* la contrainte φ

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 2.3.

Définition 6. 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 peut être *sur-simplifiée* ce qui veut dire qu'une donnée correcte peut être considérée comme une violation. Prenons un exemple sur la table 2.1 avec la denial constraint suivante :

$$\varphi_2 = (Nom, =)(NumTel, \neq)$$

Cette contrainte veut dire que si une personne possède le même nom qu'une autre, alors elle ne peut pas avoir un numéro de téléphone différent. Ceci est biensur incorrect, en effet deux personnes différentes ne peuvent avoir le même numero de téléphone. Le

2. Tout tuple qui satisfait $a\phi_2 b$ satisfait $a\phi_1 b$

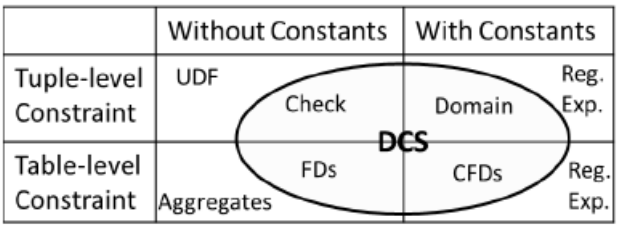

FIGURE 2.1 – A denial constraint(DC) can express many type of others constraints

nom seul ne suffit pas à identifier si deux personne sont identiques. Prenons par exemples t_1 et t_2, ils ne satisfont pas φ_2. Si l'on regarde de plus prêt, on peut facilement comprendre qu'il s'agit de deux personnes différentes. Ces deux personnes n'ont pas le même age i.e elles ont une date d'$Anniversaire$ différent. Si nous souhaitons améliorer la précision de la contrainte et éviter que $\langle t_1, t_2 \rangle$ soit considérer comme une violation, nous avons besoins de regarder l'attribut $Anniversaire$. Une meilleure DC serait :

$$\varphi_2' = (Nom, =), (Anniversaire, =), (NumTel, \neq)$$

Une DC peut être également *sur-raffiné* ce qui entraine qu'une donnée erronée peut être considérée comme correcte par la DC. Prenons un exemple sur la table 2.1 avec la denial constraint suivante :

$$\varphi_2' = (Nom, =), (Anniversaire, =), (NumTel, \neq), (Anne, =)$$

Dans ce cas, l'information $Anne$ n'est pas utile pour distingué deux personne différente. Dans la table, l'attribut année correspond à l'année où les autres attributs ont été encodés. Un même personne peut être encodé deux fois à deux années différentes. Avec cette DC on ne reconnait pas $\langle t_5, t_8 \rangle 0$ comme étant une violation.

2.3.1 Quelques défintion et propriétés

Dans cette sous-section, nous allons définir quelques notions et propriétés sur les DC qui ne serviront dans les chapitres qui suivront.

2.3.1.1 Satisfiabilité

Définition 7. Soit φ DC sur S. On dit que φ est *satisfiable* si elle peut être satisfaite par une relation non vide sur S, i.e si $\exists I$ over S avec I non vide tel que $I \models \varphi$, alors φ est *satisfiable*.

Si φ n'est pas satisfiable, nous dirons qu'il est *insatisfiable*

Il est intéressant de savoir à l'avance si une une denial constraint est satisfiable ou pas. Le lemme

Lemme 1. Soit φ une denial constraint sur S, alors φ est satisfiable si et seulement si il existe un prédicat $P_i \in pred(\varphi)$ de la forme (A_i, θ_i) tel que θ_i ne contient pas $=$, i.e $\theta_i \notin \{\{=\}, \{<, =\}, \{=, >\}, \{<, =, >\}, \}$

Démonstration.

$\boxed{\Longrightarrow}$ Supposons que pour tout $P \in pred(\varphi)$, θ contient $=$. Alors pour chaque tuple s sur S, pour chaque $P_i \in pred(\varphi)$ on a $s(A_i)\theta_i s(A_i)$. Il s'ensuit que toute relation non vide ne satisfait pas φ

$\boxed{\Longleftarrow}$ Supposons $B \in S$ tel que B ne contient pas $=$. Alors pour chaque tuple s sur S, nous avons que $s(B)\ \theta s(B)$ avec $\theta = \varphi(B)$ faux. Il s'ensuit que n'importe quelle relation avec exactement un tuple satisfait φ.

□

2.3.1.2 Implication logique

Définition 8. Soit φ_1, φ_2 deux DC sur S. On dit que φ_1 *implique (logiquement)* φ_2, que l'on note $\varphi_1 \models \varphi_2$, si pour chaque relation I sur S, si $I \models \varphi_2$ alors on a $I \models \varphi_1$. On dira aussi que φ_2 est *plus faible* que φ_1 ou bien que φ_1 est *plus fort* que φ_2

Exemple 1. Soit $S = \{A, B\}$. Soit $\varphi_1 = \{(A, \leq), (B, \neq)\}$ et $\varphi_2 = \{(A, <), (B, >)\}$. Alors φ_1 implique φ_2. En effet, soit I une relation qui satisfait φ_1. Alors pour tout tuples $s, t \in I$, on a $s(A) > t(A)$ ou bien $s(B) = t(B)$ (ou éventuellement les deux en même temps). Il s'ensuit que pour tout tuples $s, t \in I$, on a $s(A) \geq t(A)$ ou bien $s(B) \leq t(B)$ (ou les deux en même temps). Dès lors, I ne contient pas deux tuples s, t tel que $s(A) < t(A)$ et $s(B) > t(B)$. On a donc I qui satisfait φ_2. D'un autre côté, φ_2 n'implique pas φ_1. En effet, considérons la relation I suivante.

I	A	B
	1	2
	1	3

Dès lors, nous avons $I \models \varphi_2$, mais $I \not\models \varphi_1$.

Lemme 2. Soit φ_1 et φ_2 deux denial constraints sur S. Si $\varphi_2(A) \subseteq \varphi_1(A)\ \forall A \in S$, alors $\varphi_1 \models \varphi_2$.

Démonstration. Supposons que $\varphi_2(A) \subseteq \varphi_1(A)$ pour tout $A \in S$. Soit I une relation sur S tel que $I \models \varphi_1$. Nous avons besoin de démontrer que $I \models \varphi_2$. Soit $s, t \in I$. Puisque $I \models \varphi_1$, nous pouvons supposer l'existence d'un $A \in S$ tel que $s(A)\ \theta\ t(A)$ est faux, avec $\theta = \varphi(A)$. Puisque $\varphi_2(A) \subseteq \varphi_1(A)$, alors nous aurons $s(A)\ \theta'\ t(A)$ est faux, avec $\theta' = \varphi_2(A)$. □

Exemple 2. $\{(A, =)\} \models \{(A, \leq)\}$. car $\{=\} \subseteq \{<, =\} \equiv \leq$

2.3.1.3 Trivialité

Une DC peut être inutile et toujours vraie. De telles DC ne devraint pas être présentes dans la base de données puisqu'ils ne détecteront jamais aucune violation. Dans ce cas on dira que le DC est *triviale*.

Définition 9. Une DC φ est dite *triviale* si $\forall I$ sur S, on a $I \models \varphi$

Lemme 3. Soit φ une denial constraint sur S. alors φ est triviale si et seulement si $\varphi(A) = \perp$ pour un prédicat $P = (A, \theta) \in pred(\varphi)$.

Démonstration.
$\boxed{\Longrightarrow}$ Supposons que pour chaque prédicat $P = (A, \theta) \in pred(\varphi)$. Soit s, t deux tuples tel que pour chaque $P = (B, \theta)$, nous avons $s(B)\ \theta\ t(B)$ vrai. Puisque $\theta \neq \perp$ et que **dom** contient au moins deux éléments, s, t peuvent être construit. Dès lors $\{s, t\}$ ne satisfont pas φ puisque φ n'est pas triviale.
$\boxed{\Longleftarrow}$ Supposons qu'il existe un prédicat $P = (A, \theta) \in pred(\varphi)$ tel que $\theta = \perp$. Puisque $s(A) \perp t(A)$ est faux pour tout tuples s, t sur S, aucune relation ne peut contenu deux tuples s, t tel que $s(A) \perp t(A)$ est vrai. □

2.3.1.4 Augmentation

Dans les chapitres suivants, nous verrons la modification de DC afin d'améliorer des denials contraints pour qu'elles puissent détecter les données erronées de manière correcte. Pour ce faire nous aurons besoins de modifié des prédicats et donc de changer les opérateurs de ceux ci. Mais modifié un opérateur $\top$ est inutile par la propriétés suivante : 5

Propriété 1. Soit une DC φ sur S et une relation I tel que $I \models \varphi$. Si il existe un prédicat $P_i = (A_i, \theta_i) \in pred(\varphi)$ tel que $\theta_i = \top$ alors la DC φ' tel que $\varphi' = \varphi$ à l'exception du prédicat $P'_i = (A_i, \theta_i)$ et $\theta_i \neq \top$, on a $I \models \varphi'$

Cette propriété est triviale. Souvenons nous que φ est un DC tel que $I \models \varphi$ donc $\forall t \in I$ on a φ vrai. Imaginons que φ contient le prédicat de la forme $(A, \top)$ Prenons φ' une DC qui est la variante de φ tel que $\varphi = \varphi'$ à l'exception du prédicat $(A, \top)$ qui devient (A, θ) avec $\theta \neq \top$. Puisque φ était satisfaite par I, il y avait déjà un autre prédicat de φ qui était faut $\forall t \in I$. Donc φ' est satisfaite pour qu'importe la valeur de (A, θ).

2.3.1.5 Transitivity

Propriété 2. Soit φ et φ' deux DC sur S et I une relation sur S. Si $\varphi\{P_1, P_2, ..., P_{i-1}, P_i\}$ satisfaite par I et $\varphi' = \{P'_i, P_{i+1}, ..., P_n\}$ satisfaite par I, avec P_j prédicat de la forme (A_j, θ_j) et $\theta'_i \in Imp(\overline{\theta_i})$, alors $\varphi'' = \{P_1, P_2, ..., P_{i-1}, P_{i+1}, ..., P_n\}$ est également satisfaite par I

En d'autre mots, il est possible de fusionner deux denial constraint satisfaite par I si ces deux contraintes possède chacune un prédicat et ceux-ci ne peuvent pas être faux en même temps. Alors la fusion de ces deux contraintes sans les deux prédicats est toujours satisfaite.

2.3.1.6 Raffinement

Dans l'article [4] ils définissent le raffinement d'une denial constraint comme étant :

Définition 10. φ_2 est un **raffinement** de φ_1, noté $\varphi_1 \preceq \varphi_2$, Si pour chaque prédicat $(A, \theta_A) \in pred(\varphi_1)$ on a un prédicat $(A, \theta'_A) \in pred(\varphi_2)$ tel que θ'_A implique θ_A ($\theta'_A \in Imp(\theta_A)$)

Exemple 3. Soit $\varphi_1 = \{(Taxe, \leq), (Revenu, <)\}$ et $\varphi_2 = \{(Taxe, <), (Revenu, <), (Anne, =)\}$ nous avons $\varphi_2 \preceq \varphi_1$ car $\varphi_1(Taxe) \in Imp(\varphi_2(Taxe))$ et $\varphi_1(Revenu) \in Imp(\varphi_2(Revenu))$ et $\varphi_1(Anne) \in Imp(\varphi_2(Anne))$

Notons que φ est raffinement de lui même et que remplacer l'opérateur $\top$ d'un prédicat par n'importe quel autre

Définition 11. Σ_2 est **raffinement** de Σ_1, noté $\Sigma_1 \preceq \Sigma_2$, si pour chaque $\varphi_2 \in \Sigma_2$, Il existe un $\varphi_2 \in \Sigma_1$ tel que $\varphi_1 \preceq \varphi_2$

Si nous voulons changer moins de données, nous pouvons raffiner nos DC. Dans notre exemple précédent, la nouvelle denial constraint est plus faible que la précédente, diminuant le nombre de tuples détecté comme étant une violation.

Chapitre 3

Data Repairing

Les erreurs sont fréquentes dans les bases de données et ces anomalies nuisent à la fiabilité de certaines applications les utilisant. Il existe des méthodes dont le but est de détecter ces erreurs mais ces méthodes ne réparent pas les erreurs. A la place, les applications pourront filtrer les données et ignorer les erreurs détectées mais les applications peuvent toujours être non fiable [5]. Au lieu de simplement détecter les erreurs et les filtrer, il est préférable de réparer les données erronées.

Dans le chapitre précédent nous avons vu comment détecter des erreurs au moyen de denial constraints. Nous avons aussi discuter brièvement de la sur-simplification ou du sur-raffinement de ces DC. Nous allons maintenant aborder la réparation des données mais aussi la réparation des DC.

Le but d'une réparation de donnée est de trouver une instance I' qui est une modification d'une instance I de S pour laquelle il y a au moins un tuple t contenant une donnée erronée.

Définition 12. Soit Σ un ensemble de DC sur S. Soit une relation I sur S. On dit que I *satisfait* Σ noté $I \models \Sigma$ si pour chaque DC φ avec $\varphi \in \Sigma$, nous avons $I \models \varphi$

Définition 13. Soit Σ un ensemble de DC sur S. Soit une relation I sur S. Une *réparation de* I est une fonction f de domaine I qui attribue à chaque tuple t de I un nouveau tuple $f(t)$ tel que $f(t) \models \Sigma$. L'ensemble d'arrivée de f est noté $I' = f(I)$. On a donc $I' \models \Sigma$

Donc lorsque l'on parle de réparation de donnée, on cherche à trouver I' tel que $I' \models \Sigma$ c'est à dire une nouvelle instance où toutes les violations dans le set de contrainte Σ sont éliminées. Nous allons considérer que lors d'une réparation, nous ne supprimons pas de tuples mais nous pouvons que le modifier. Nous n'ajoutons pas de tuples à la relation non plus. Cela peut conduire à avoir deux tuples avec les mêmes valeurs une fois la réparation effectuée.

Il est important de noté que si on a un tuple $t \in I$ tel que $I \models \Sigma$ alors il n'est pas nécessaire de le modifié et donc nous avons la propriété suivante :

Propriété 3. Soit Σ un ensemble de DC sur S. Soit une relation I sur S et $f(I) = I'$ une réparation de I. Si on a un tuple $t \in I$ tel que $t \models \Sigma$ alors nous avons $f(t) = t$. Sinon $f(t) \neq t$.

La réparation de donnée suit le principe du changement minimum : La nouvelle instance I' doit minimiser le coût de réparation de donnée défini comme étant :

Définition 14. Soit I une relation sur S et $f(I)$ une réparation de I, alors le coût de réparation est le suivant :

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

où :

— $dist(t.A, f(t).A)$ est la distance entre la valeur $t.A$ et sa réparation $f(t.A)$.

— $w(t.A)$ est un poid sur l'attribut A.

Dans le coût nous avons un poid $w(t.A)$ pour un attribut $A \in S$ et un tuple $t \in I$. Ce poid correspond à la confiance que l'on a en la valeur de t pour l'attribut A. On peut grâce à cette valeur influencer la réparation de données pour privilégier la réparation d'une valeur plutôt qu'une autre. Pour pouvoir assigner une valeur à $w(t.A)$, il faut avoir une bonne connaissance du contexte de la base de donnée d'origine.Il est courant d'avoir la même valeur pour $w(t.A)$ et $w(s.A)$ avec $s, t \in I$ car en général on a connaissance de la confiance pour un attribut en particulier par chaque valeur du tuple. Par exemple pour la table 2.2, nous pouvons supposer qu'une valeur pour l'attribut *Enfant* est plus susceptible d'être précise que la valeur pour l'attribut *Salaire* ou *Taxe*. Lorsqu'on manque de connaissance sur la base de donnée, on fixe le même poid à chaque attribut.

Le coût de réparation peut être le nombre de valeur de I que l'on a changé si nous décidons que :

$$dist(t.A, f(t).A) = \begin{cases} 1 \; Sit.A \neq f(t).A \; (La \; valeur \; a \; chang) \\ 0 \; Sinon \; (aucun \; changement \; n'est \; fait) \end{cases}$$

Nous pouvons aussi décider que la distance est égale à la différence entre les deux valeurs dans le cas d'un attribut numérique. Pour un attribut de type chaine de caractère nous pouvons utiliser la distance d'édition [1]

.

1. Le nombre minimum d'opération nécessaire pour transformer la chaine de caractère initiale en la chaine de caractère cible

Pour réparer une donnée, nous devons remplacer sa valeur par une autre plus adéquate. Mais quel valeur choisir ? Dans l'article de référence principal [4], si la valeur $t.A$ est erronée, il essaye de trouver une valeur pour $f(t).A$ qui soit dans le $\mathbf{dom}(A)$ et qui respectent les contraintes. Si ce n'est pas possible ils attribuent une *variable fraiche* fv à $t.A$.

Définition 15. Une *variable fraiche* fv est une valeur qui ne satisfait aucun prédicat c'est à dire : soit φ une DC sur S et I une relation sur S. Une variable fraiche fv pour $A \in S$ est une valeur tel que $fv \notin \mathbf{dom}(A)$ le prédicat $(A, \theta_A) \in pred(\varphi)$ est toujours faux quelque soit l'opérateur θ_A donc pour tout tuple t avec $t \in I$, si $t.A = fv$ alors $t \models \varphi$.

Nous allons procéder de manière un peu différente par rapport à l'article de référence [4]. Nous allons automatiquement attribué une nouvelle valeur fv pour chaque donnée erronée. Par la suite, nous allons essayer de trouver une valeur adéquate pour remplacer la valeur fraiche fv si possible. Une des motivations derrière ce changement vient du fait qu'après une réparation, certaines valeur du domaine n'y figure plus (ces valeurs étaient erronées). De plus il n'est pas pertinent de croire que si une valeur du domaine peut réparé une valeur erronée, alors elle est forcément la bonne valeur. Une valeur en dehors du domaine peut tout aussi bien convenir.

Notons toutefois que la variable fraiche $fv_{t.A}$ et la valeur fraiche $fv_{s.A}$ peuvent être différente ou identique et ce pour tout tuple $s, t \in I$. Par défaut nous considérons que toutes les variables fraiches sont différentes.

Exemple 4. Prenons un exemple avec comme relation I la table 2.1. Supposons que notre denial constraint est la suivante :

$$\varphi = \{(Revenu, >), (Taxe, \leq)\}$$

En d'autre terme, on suppose par cette contrainte que si une personne perçoit un revenu annuel plus élevé qu'une seconde personne, alors cette première personne doit payer un montant plus élevé de taxe par an. Nous avons $(t_2, t_1) \not\models \varphi$ parce que $t_2.Revenu > t_1.Revenu$ et $t_2.Taxe \leq t_1.Taxe$. Nous avons également $(t_3, t_1) \not\models \varphi, (t_5, t_1) \not\models \varphi, ...$ Toutes les violations de φ peuvent être trouvé à la figure 3.1. Une réparation I' de I pourrait être le suivant :

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

TABLE 3.1 – Exemple de réparation I' pour l'attribut Tax for φ.

Nous avons mit fv_1 comme variable fraiche pour t_2. Bien que ce soit une variable fraiche, nous savons plusieurs chose à propos d'elle. En effet nous savons les choses suivantes :

/	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 – Toutes les violations pour φ

1. $I(t_1.Tax) = 0$ donc $I'(t_2.Tax) > 0$ car $I(t_1.Income) < I(t_2.Income)$
2. $I(t_4.Tax) = 3$ donc $I'(t_2.Tax) < 3$ car $I(t_2.Income) < I(t_4.Income)$

Nous avons assigner une variable fraiche fv_1 comme valeur pour $t_2.Tax$ mais nous savons quand même que $0 < fv_1 < 3k$ i.e $fv_1 \in]0, 3k[$ grâce à 1 et 2. Nous pouvons utiliser la même logique pour connaitre les valeurs possible pour pouvoir dire que $fv_2 = fv_1$,$3k < fv_3 < 21k$,$fv_4 = fv_3$ et $fv_3 < fv_5 < 21k$. Maintenant que nous avons identifié les valeurs erronées et identifié un ensemble de valeur pour remplacer les variables fraiches, il reste à savoir quel valeur finale on peut prendre. Pour cela nous avons plusieurs solution possible.

— Prendre une valeur de $\mathbf{dom}(A)$. C'est la solution envisagée dans le principal article de référence que nous utilisons. Il ne sera pas toujours possible de prendre une valeur dans le domaine, par exemple il n'y a pas de valeur dans le $dom(Taxe) = \{0, 3k, 21k, 40k\}$ qui puisse satisfaire la condition fv_1. Cette pratique n'est pas la plus logique puisque même si nous avions une valeur dans $\mathbf{dom}(Taxe)$ qui puisse satisfaire fv_1, il y a plusieurs valeurs en dehors de $\mathbf{dom}(Taxe)$ qui sont tout aussi correcte.

— Prendre une valeur aléatoire mais respectant les conditions sur la variable fraiche. C'est une très mauvaise idée puisque nous avons une chance de s'éloigner de la vraie valeur. Cela peut impacter énormément l'ajout de tuple dans la relation après que la réparation soit effectuée. Ces nouveaux tuples malgré qui soient correcte pourraient être perçu comme contenant des donnée erronées.

— Garder les variables fraiches dans la base de données tout en conservant les informations que l'on connait à propos de celles-ci. Si nous n'avons qu'une seule valeur possible alors nous privilégions cette valeur à fv. Cette solution est celle qui respectera

le mieux l'intégrité des données. Le seul problème qu'apporte cette solution est que de nombreuses applications ne pourront plus fonctionner avec de telles variables. Les informations que l'on connait sur elles peuvent changer au fur et à mesure que la relation se remplit.

Nous pouvons calculer coût de réparation pour la relation de la table 2.1 en considérant les distances suivantes :

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

Lorsqu'on ne change pas la valeur, la distance est bien évidement égale à zéro. $dist(a, fv)$ soit être supérieure à $dist(a, b)$ pour privilégier les valeurs aux variables fraiches. $dist(fv, fv)$ représente le fait qu'on avait déjà une variable fraiche venant d'une réparation antérieure, et qu'on garde une variable fraiche. $dist(fv, b)$ représente le changement d'une variable fraiche venant d'une réparation antérieure et on change la change par une valeur fixe. Cela peut arriver lorsque l'on possède de plus amples informations sur la valeur fraiche, grâce à de nouveaux tuples dans la relation. Nous avons besoin que $dist(fv, fv) > dist(fv, b)$ pour favoriser la correction par une vraie valeur quand c'est possible. Dans l'exemple précédent, avec les valeurs susmentionné, nous pouvons calculer un coût de réparation $\Delta(I, I')$ = 7,5.

3.1 Variation sur les denial constraints

Nous avons vu précédemment qu'une denial constraint peut être sur-raffiné, échouant donc dans la détection d'erreurs. Une DC peut aussi être sur-simplifiée ce qui conduit à considérer des données correctes comme étant erronées. Parce que ces contraintes peuvent être imprécise et inexacte, nous avons besoin de les modifier. En modifiant ces contraintes nous pouvons obtenir une meilleure réparation

Exemple 5. Par exemple prenons la DC suivante :

$$\varphi = \{(Revenu, >), (Taxe, \leq)\}$$

La denial constraint φ exprime le fait que si je reçoit un revenu plus élevé qu'une autre personne, alors je dois payer un montant de taxe strictement plus élevé. Nous allons modifier φ pour obtenir une nouvelle denial constraint φ' en changeant le prédicat $(Taxe, \leq)$ en $(Taxe, <)$. Dès lors maintenant φ' exprime le fait que si je reçoit un revenu plus élevé qu'une autre personne, alors je dois payer un montant de taxe plus élevé ou équivalent.

/	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	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'

Cela permet d'exempté de taxe plusieurs personnes ayant un Revenu faible, mais différent les uns des autres.

$$\varphi' = \{(Revenu, >), (Taxe, <)\}$$

Avec cette nouvelle contrainte, nous avons moins de violations détectée. Toutes les violations peuvent être trouvées à la figure 3.2. Les modifications que nous avons faites dans la table 3.1 sont :

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_18	t_9	t_10
Taxe	0	0	0	fv_1	0	0	0	21k	21k	40k

TABLE 3.2 – Example of repair with Tax

.

Contrairement à la réparation de φ qui proposait une correction avec 5 variables fraiches (voir table 3.1), nous avons ici qu'une seule variable fraiche. De plus nous savons certaines choses à propos de cette variable fraiche :

1. $t_1.Tax = 0$ donc $f(t_4).Tax \geq 0$ car $t_1.Income < t_4.Income$
2. $t_5.Tax = 0$ donc $f(t_4).Tax \leq 0$ car $t_4.Income < It_5.Income$

Nous avons donc $0 \leq fv_1 \geq 0$, dès lors la seule possibilité que nous avons est que $fv_1 = 0$. Nous avons donc un coût de réparation de $\Delta(I, I') = 1$ en reprenant les valeurs de distances de l'exemple. **??**.

Nous voyons donc que la modification de la contrainte conduit à un coût de réparation plus faible. Mais modifié une contrainte doit aussi avoir un coût. En effet , si on ne

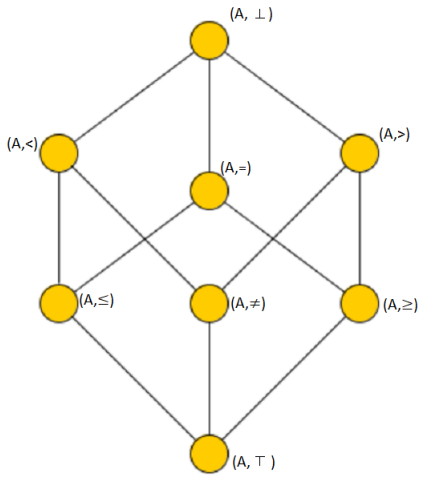

FIGURE 3.3 – Treillis pour un prédicat d'attribut A

donne pas de coût à la modification d'une contrainte, il suffirait de sur-raffiné toutes nos contraintes pour détecter moins de violation et donc faire moins de modification. Nous allons aussi considérer que l'on ne rajoute pas DC ni n'en retire de Σ pour obtenir Σ'.

Lorsque nous modifions une DC, il faut faire attention à ne pas la rendre triviale. Nous avons vu dans le chapitre précédent que ce genre de contrainte étaient inutiles car elles ne considéraient aucun tuple comme étant une violation. Considérons le treillis tel qu'illustrer à la figure 3.3 pour un attribut A de S. Chaque nœud du treillis représente un prédicat différent pour A, donc A associé à un opérateur, un des éléments du powerset de OP. Descendre dans une branche du treillis consiste à ajouter un élément de OP à l'opérateur. Par exemple passer de $(A, <)$ à $(A, \leq)$ consiste à ajouter l'opérateur $=$ à $<$. Lorsque l'on monte dans le treillis, on retire un élément de OP à l'opérateur. Par exemple, passer de $(A, \geq)$ à $(A, =)$ consiste à retirer l'opérateur $>$ à $\geq$.

Nous devons modifier nos DC de tel manière à ce que une contrainte sur-raffinée ou sur-simplifiée ne le soit plus. Pour les DC sur-raffinée, nous avons besoin de supprimer certains prédicats, i.e pour certains attributs A de S, nous devons passer d'un $\varphi(A) \neq \top$ à $\varphi(A) = \top$. Mais nous devons faire attention à ne pas retire trop de prédicats, sinon elle risque de devenir sur-simplifiée. Si la DC est sur-simplifiée, nous avons besoin d'ajouter des prédicats, i.e pour certains attribut A de S, nous devons passer d'un $\varphi(A) = \top$ à un $\varphi(A) \neq \top$. Plus nous changeons de prédicats, plus le *coût de variation des contraintes* sera important. Le *coût de variation des contraintes* se définit comme tel :

Définition 16. Soit Σ un ensemble de DC sur un ensemble S. Pour une variante Σ' de Σ, la fonction de calcul du *coût de variations des contraintes* de Σ est :

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

avec φ' est une variante de φ et $edit(\varphi, \varphi')$ est le coût pour changer φ en φ'.

La fonction $edit(\varphi, \varphi')$ qui indique le coût pour changer φ en φ' est défini comme étant :

Définition 17.

$$edit(\varphi, \varphi') = \sum_{A \in S} path(\varphi(A), \varphi'(A))$$

avec $path(\varphi(A), \varphi'(A)$ le coût du chemin emprunté dans le treillis.

Si nous considérons notre treillis comme un graphe, il reste maintenant à trouver le poids de chacun des arcs de ce graphe.

3.1.1 Limitation de nos candidats

Dans notre treillis nous avons 8 nœuds. Pour une DC sur un ensemble S de taille 1, nous avons 8 variantes de DC différentes. Pour un S de taille n nous avons 8^n variations. Ce nombre est très important et envisager toutes les variations de contraintes serait coûteux en terme de complexité. Nous avons besoin de limiter le nombre de variations à considérer. Pour ce faire, nous allons utiliser des propriétés sur les DC pour limiter le nombre de contraintes à considérer.

Nous savons déjà par le chapitre précédent que les contraintes triviales sont inutiles. En effet, une contrainte triviale sera toujours vraie et donc considérera tous les tuples comme étant corrects. Le Lemme 3 nous permet de rejeter toutes les DC dont au moins un des prédicats est de la forme $(A, \perp)$.

Nous avons également besoin que la denial constraint soit satisfiable. Si une DC n'est pas satisfiable alors elle possède uniquement des prédicats avec l'opérateur $\top$ ou $=$. Nous pouvons donc rejeter toutes les DC ne possédant aucun opérateur parmi $\{\leq, \geq, \neq, <, >\}$

3.1.2 Denial constraints maximales

Nous avons besoin également que les DC soit *maximale*. Dans [4] ils définissent une DC maximale comme étant :

Définition 18. La variante φ' d'une DC φ est dite *maximale* si $\varphi \preceq \varphi'$ [2] est qu'il n'existe pas de DC φ'' tel que $\varphi' \preceq \varphi'$

2. voir définition du raffinement section 2.3.1.6

Propriété 4. [4] Soit φ une DC sur S. Soit φ' une variante de φ sur S. Pour chaque attribut A de S tel que $\varphi(A) = \top$, si $\varphi'(A) \in \{\neq, \leq, \geq\}$ alors φ' n'est pas maximal.

Cette propriété vient de la définition de $Imp(\varphi)$. Soit φ_1, φ_2 deux DC sur S Soit une relation I sur S. Pour deux tuples $s, t \in I$ et un attribut A de S, si $Varphi_1 \in Imp(\varphi_2)$ alors $s.A\varphi_1(A)t.A$ implique $s.A\varphi_2(A)t.A$. $\leq\geq$ et $\neq$ sont les 3 opérateurs qui impliquent d'autres opérateurs que eux même comme on peut le voir à la table 2.3(c'est également le cas pour l'opérateur $\top$, mais celui-ci est exclus de la propriété).

Grâce à cette propriété, nous savons maintenant qu'il est inutile de considérer toutes les insertion [3] de prédicats possible. Nous avons seulement besoin d'insérer des prédicats avec les opérateur $\{<, =, >\}$ lorsque que l'on considère les variantes de φ. Illustrons cela avec un exemple.

Exemple 6. Nous allons prendre deux DC sur la relation de la table 2.1 :

$$\varphi_1 : \{(Nom, =), (Revenu, =), (Numtel, \neq)\}$$

$$\varphi_2 : \{(Nom, =), (Revenu, \leq), (Numtel, \neq)\}$$

Nous savons que $\leq \in Imp(=)$ (voir table 2.3) donc nous avons $\varphi_2 \preceq \varphi_1$. Grâce à la propriété 4, φ_2 n'est pas une denial constraint maximale parce qu'elle contient l'opérateur $\leq$. Dans ce scénario, nous pouvons calculer un coût de réparation pour l'attribut $NumTel$ de 7 pour la DC φ_2 et la DC φ aura un coût de réparation de $Numtel$ de valeur 3.

	Nom	NumTel	Revenu
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 avec φ_2 : 7 changement nécessaire dans la colonne $NumTel$

.

	Nom	NumTel	Revenu
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 avec φ_1 : seulement 3 changements sont nécessaires.

.

3. Par insertion de prédicats, comprenons ici que nous passons de $\varphi(A) = \top$ à $\varphi(A) \neq \top$

Nous voyons que la DC raffinée a un coût de réparation plus faible. Ce n'est pas une coincidence puisque le Lemme suivant existe : [4]

Lemme 4. Soit Σ un ensemble de DC sur S et I une relation sur S. Soit 2 variantes de Σ noté Σ_1, Σ_2 et Σ_2 est un raffinement de Σ. Si $\Sigma \preceq \Sigma_1$ et $\Sigma_1 \preceq \Sigma_2$ alors $\Delta(I, I_1) \geq \Delta(I, I_2)$ avec I_1 la relation résultante de la réparation avec Σ_1 et I_2 la relation résultante de la réparation avec Σ_2.

Comme conséquence à ce Lemme, toutes les ensemble de denial constraint Σ qui ne sont pas maximal ne donneront pas la meilleure réparation..

3.1.3 Limitation par bornes

Nous avons vu précédemment que pour une seule denial constraint, nous avons $8^{|S|}$ variantes possibles. Nous avons donné des pistes pour limité le nombre de candidats. Pour continuer dans cette lancée, nous allons consi 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 :

Propriété 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.

3.1.3.1 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]

Définition 19. The violation set noted as $viol(I, \varphi) = \{\langle t_i, t_j, ...\rangle | \langle t_i, t_j, ...\rangle \not\models \varphi$ 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 :

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

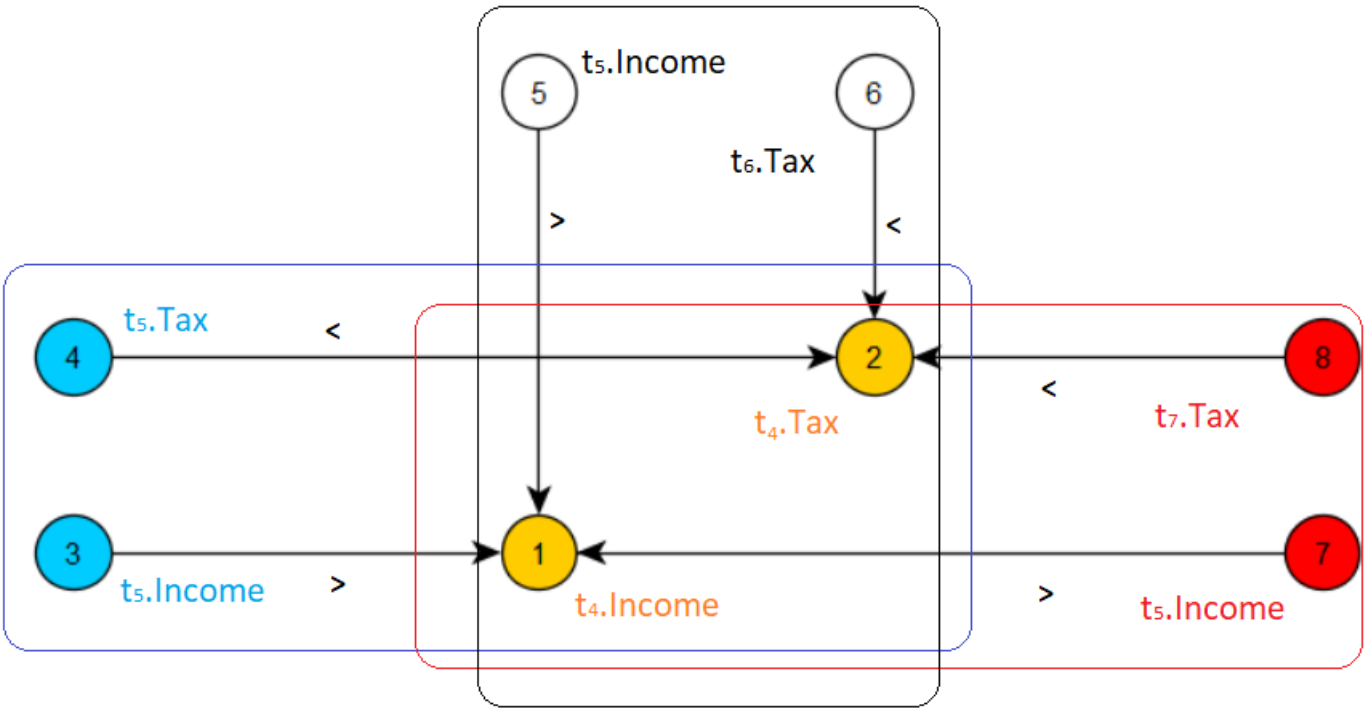

FIGURE 3.4 – Conflict hypergraph for φ

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]

Définition 20. 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.

Définition 21. $\mathbb{V}(G)$ is the **minimum weighted vertex cover** of the hypergraph G corresponding to Σ with weight

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

Lemme 5. 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 θ : $\Theta(\Sigma, \Sigma') \leq \theta$. 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 [4]. 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.

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 Σ, a set $\mathbb{D}$of constraint variants with variation bound by θ
Output : A minimum data repair I_{min}

```
δ_min = δ_u(Σ, I)
for each constraint variant Σ_i ∈ 𝔻 do
    if δ_l(Σ_i, I) ≤ δ_min then
        I_i = DATAREPAIR(Σ_i, I, 𝕍(G_i), δ_min
        if Δ(I, I_i) ≤ δ_min then
            δ_min = Δ(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 $\Sigma_2 = \{\varphi''\}$ with :

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

4. problems whose a solution can be verified as good one in a polynomial time

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

We already have done the conflict hypergraph for φ in figure 3.4 we also know that $\delta(\Sigma_1, I) = 1.1$

For $\Sigma 2$ we obtain the hypergraph in figure 3.5 (and the violations in figure 3.6) with $\mathbb{V}(G_2) = \{t_2.Tax, t_3.Tax, t_5.Tax, t_6.Tax, t_7.Tax\}$. We have $Deg(\Sigma_2) = Deg(\varphi')$ [5], 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].

3.3 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' [6]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

Définition 22. [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 :

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

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

5. same reasoning : 4 cells involved.
6. 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

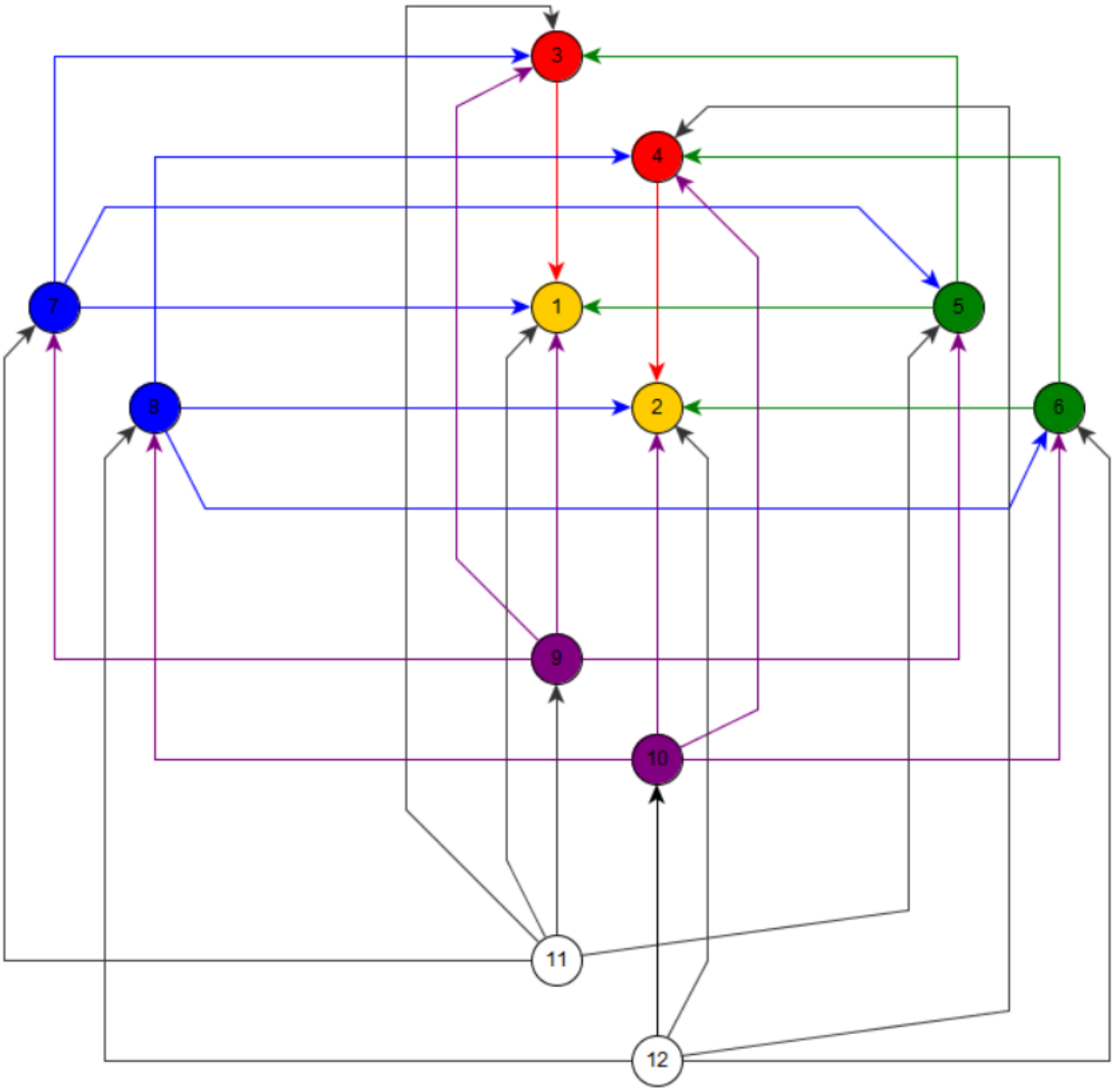

FIGURE 3.5 – Conflict hypergraph for Σ_2 with :
odd number for Tax, even number for Income.
t_1 in yellow, t_2 in red, t_3 in green,
t_4 in blue, t_5 in purple and t_6 in white.

/	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 3.6 – All the violations for φ''

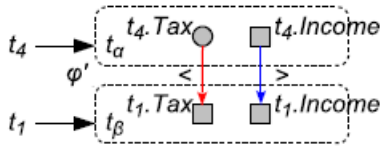

FIGURE 3.7 – Suspect condition represented by blue arrows and repair context represented by red arrows (with inverse operator).

Lemme 6. 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 3.4. 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 3.7 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 :

Définition 23.

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

Propriété 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 3.7. 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 Différence par rapport à l'article de base

Dans ce chapitre, nous parlerons des différences entre l'article de référence sur le modèle θ-tolérant et ce mémoire. Certaines notions et définitions ont été revues, et le traitement théorique a été amélioré. Certaines erreurs ont été corrigées notamment des erreurs concernant les variations de contraintes et coût de réparation de la base de donnée.

Pour la rédaction de ce mémoire, nous nous sommes inspiré de deux articles. Le premier est un article sur le modèle θ-tolérant que nous aborderons au prochain chapitre, le second est un article qui porte sur les denial constraints. Ils utilisent une approche différente et une définition différente de la notre. Dans [4, 3] ils définissent une DC comme étant :

Définition 24. Considérons un schéma de relation R avec comme attributs $att(R)$. Soit l'espace de prédicat $\mathbb{P}$ qui est un ensemble de prédicat P de la forme $v_1 \phi v_2$ ou $v_1 \phi c$ avec $v_1, v_2 \in t_x.A,\ x \in \{\alpha, \beta\}$, $t_\alpha, t_\beta \in R$, $A \in attr(R)$, c une constante et $\phi \in \{=, <, >, \leq, \geq, \neq\}$ est un opérateur. Une *denial constraint* (DC)

$$\varphi : t_\alpha, t_\beta, ... \in R, \neq (P_1 \wedge P_2 \wedge ... \wedge P_m)$$

signifie que pour tout tuples t_α, t_β dans R, tous les prédicats $P_i \in pred(\varphi)$, i = 1,....,m, ne devraient pas être tous vrai en même temps.

Une DC peut donc être vue comme une conjonction de prédicats et l'un de ses prédicats doit être faux afin que la DC soit vraie, i.e si pour deux tuples chaque prédicats est vrai, alors il y a au moins une donnée erronée dans l'un des deux tuples.

La définition bien qu'étant différente d'un point syntaxique, elle représente toutes les deux la même chose. Cette définition-ci permet en plus de comparé un attribut à une constante. Bien qu'il puisse être intéressant d'avoir des denial contraints qui fixe un salaire minimum par exemple $t.Revenu > 10k$ ou exprime le fait qu'un revenu ou une taxe ne peut être négative $t.Revenu > 0 \wedge t.Taxe > 0$ cela apporte plusieurs problèmes.

Dans notre définition, nous avons un nombre de prédicats maximum qui est la norme de S. Ici nous ne sommes pas limité et ce à cause des constantes. Et cela peut avoir des problèmes lorsque l'on tente de changer une DC lors de la réparation. Et malgré que cela puisse être un problème, ce n'est jamais abordé dans [4].

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.