Contrôle d'accès dans les bases de données : conception et vérification

Antonin DUREY, avec Pierre BOURHIS et Sophie TISON

Février 2017

1 Introduction

Nous produisons de plus en plus de données, à gérer dans différents endroits. Celles-ci sont sont généralement stockées dans des bases de données. On cherche à limiter le droit d'accès selon les droits d'utilisateurs pour des questions de sécurité. Pour cela, on sépare les données visibles des données invisibles. Les données visibles ont pour but d'être accessibles et manipulables contrairement aux données invisibles qui sont des données propriétaires. Les données invisibles sont stockées dans des tables et les données visibles sont stockées dans des vues. Les vues sont définies par des formules logiques portant sur les tables et permettent leur construction à partir de données invisibles contenues dans les tables. Le but est de protéger les données invisibles.

Ce type de construction pose néanmoins problème. Selon la construction des vues, des contraintes de la base de données et des clefs étrangères, il peut être possible de déduire des informations ou des données qui auraient dû rester cachées. Par exemple, sur Facebook, à partir de photos et de mentions "like" sur ces photos, il est possible de déduire des liens d'amitié que le propriétaire de la photo a voulu cacher. Ce type de faille est appelé le trou de sécurité.

La vision *Security by design* est une approche qui consiste à développer un logiciel, programme ou outil de manière à ce qu'il soit sécurisé. Cela implique de faire des choix à la conception en faveur de la sécurité au détriment d'autres facteurs comme la rapidité ou la polyvalence.

Dans notre cadre, on veut s'assurer que les données invisibles sont bien cachées. Pour éviter de telles fuites de données, dans une vision de *Security by design*, on veut vérifier que, à la conception de schéma, il n'y aura jamais de fuites. Ce problème de vérification de fuites a été formalisé sur le schéma récemment pour une large classe de définition de vues et de schémas de base de données [2]. Ces problèmes sont les problèmes de $\exists PQI$ *et* PQI. Toutefois, leurs résultats montrent que si ce problème est solvable, il est en général très coûteux. Même pour les classes les plus simples à résoudre, le problème est NP-hard, PSPACE-hard.

Je me suis intéressé au problème de l'implémentation d'un tel algorithme de vérification de trous de sécurité dans des vues de base de données, celui-ci se basant sur la structure des tables et des vues. Le but de ce projet est donc de faire un prototype pour un cadre restreint mais pratique pour montrer la faisabilité de ces algorithmes.

Pour évaluer notre algorithme, nous avons tout d'abord voulu le valider sur un grand nombre de cas. Nous avons ainsi créé un générateur de schéma et de contraintes. Par la suite, nous avons également voulu le tester sur un cas concret. C'est ainsi que nous avons étudié le schéma du laboratoire CRIStAL.

La partie 2 présente le contexte du projet, les termes techniques et la bibliographie déjà présente sur le sujet. Les parties 3 et 4 présentent deux implémentations d'algorithmes pour 2 cas. Dans chacune des parties sont présentés les résultats obtenus et leur analyse. Enfin, la partie 5 présente les perspectives futures pour ces travaux, de l'optimisation des implémentations au cas le plus général possible.

2 Contexte

2.1 Définition

Une **base de données** est un outil qui permet le stockage et la récupération d'un gros volume de données. Les données sont stockées dans des relations. Une relation est un élément d'une base de données où il est possible de stocker des données. Une relation peut être visible ou invisible. Elle possède un ensemble d'attributs dont les valeurs possibles sont comprises dans un domaine donné. Ce domaine est un ensemble infini de valeurs possibles, comme le domaine des nombres entiers, le domaine des chaînes de caractères. Dans le cadre de ce travail, on considère que tous les attributs prennent leur valeur dans un ensemble particulier appelé DOM.

Un **tuple** associe à chaque attribut d'une relation une valeur de son domaine.

L'ensemble des relations se nomme le **schéma**, et est noté S. Ainsi $S = S_h \biguplus S_v$ où S_h représente un ensemble de relations invisibles - les tables - et S_v représente un ensemble de relations visibles - les vues.

Une **instance de schéma** S, noté I, associe à chaque relation visible ou invisible un ensemble de tuples.

Un atome est un élément de la forme $A(x_1, ..., x_k)$ où A est une relation et x_1,...,x_k des variables non contenues dans DOM. Une conjonction d'atomes est un élément de la forme $atome_1 \wedge atome_2$.

Une **contrainte** est une règle qui doit vérifiée par les instances.
Une **dépendance de génération de tuples** (en anglais : tuple-generated dependancy - TGD) est un certain type de contrainte d'une base de données. C'est une phrase du premier ordre de la forme $\forall \overrightarrow{x}, \overrightarrow{y}\ P(\overrightarrow{x}, \overrightarrow{y}) \rightarrow \exists \overrightarrow{z}\ Q(\overrightarrow{x}, \overrightarrow{z})$. P représente le corps de la contrainte et est une conjonction d'atomes, et Q la tête qui est aussi une conjonction d'atomes. La frontière est l'ensemble des attributs présents dans le corps et la tête de la requête, ici $\overrightarrow{x}$.
Une **dépendance d'inclusion** (ou clef étrangère) est une TGD où P et Q sont des requêtes d'atomes dont toutes les variables ne sont répétées qu'une fois dans le corps et la tête. Une dépendance d'inclusion est un cas particulier de dépendance linéaire.

Une **requête** est une question posée à la base de données. La réponse donnée peut être sous différentes formes : vrai/faux, un nombre, une liste de tuples, etc. Dans notre cas, la réponse sera toujours vrai/faux. Une **requête conjonctive** est une conjonction de requêtes. Un **secret** est une requête conjonctive dont on cherche à prouver qu'elle est vraie, alors que cette information aurait dû rester cachée.

Dans une base de données, une **vue** représente un accès à des données visibles. Les vues peuvent être construites à partir d'une requête. Une CQvue est une vue construire à partir d'une requête conjonctive. Une **vue atomique** est une vue formée à partir d'une requête conjonctive contenant un seul élément. Si on donne une instance sur S_h et S_v, on peut définir la correction des vues par les contraintes suivantes :

$$R(\overrightarrow{x}) = \exists \overrightarrow{z}\ tq\ Q(\overrightarrow{x}, \overrightarrow{z})$$

$$\forall \overrightarrow{x}\ tq\ R(\overrightarrow{x}) \rightarrow \exists \overrightarrow{z}\ tq\ Q(\overrightarrow{x}, \overrightarrow{z})$$

$$\forall \overrightarrow{x}, \overrightarrow{y}\ tq\ Q(\overrightarrow{x}, \overrightarrow{y}) \rightarrow R(\overrightarrow{x})$$

Un **homomorphisme** est une association élément → élément de deux domaines qui permet par la suite de faire des transformation sur une instance. Formellement, c'est une association $f : A \to B$ qui définit une opération x d'arité k définit sur A et B si et seulement si $f(x_A(a_1, ..., a_k)) = x_B(f(a_1), ..., f(a_k))$.

2.2 Problèmes et approches associées

Le problème que nous cherchons à résoudre est le suivant : suivant un ensemble de vues, est ce qu'il existe une base de données dans laquelle un attaquant peut découvrir le secret, en sachant qu'il connaît les définitions des vues et des tables de la base de données ? Ce problème est lié à d'autres problèmes qui ont principalement été étudiés théoriquement. Pour plus d'information, se référer à [2]. Le problème se formalise en se demandant s'il existe une instance visible tel que le secret peut décider la certitude de la requête par rapport à une instance visible.

D'après la définition 2 [2], le problème de PQI (Positive Query Implication) est le suivant :
Pour Q une contrainte sur un schéma S, $\mathscr{C}$ un ensemble de contrainte sur S et V une instance sur un schéma visible $S_v \subseteq S$, $PQI(Q, \mathscr{C}, S, V)$ est vrai si pour chaque instance F qui satisfait $\mathscr{C}$ et si $V = Visible(F)$, alors $Q(F)$ est vraie.

Le problème général est le problème de $\exists PQI$, définit par la définition 3 [2] comme ceci :
Pour Q une contrainte sur un schéma S et $\mathscr{C}$ un ensemble de contrainte sur S, $\exists PQI(Q, \mathscr{C}, S)$ est vrai s'il existe une instance V sur un schéma visible $S_v \subseteq S$ tel que $PQI(Q, \mathscr{C}, S, V)$ est vrai.

Dans le cas général, c'est à dire avec C comme ensemble de TGDs et Q une union de requêtes conjonctives booléennes, le problème est indécidable (Théorème 15 [2]). Cette indécidabilité fait qu'il est difficile de résoudre ce problème. Néanmoins, nous pouvons appliquer plusieurs restrictions à ce problème.

La première restriction est de réduire les vues à des vues atomiques et les TGDs en des dépendances d'inclusion sans constante. Ainsi, dans le cadre spécifié par le théorème 18 et le corollaire 20 [2], le problème $\exists PQI(Q, \mathscr{C}, S)$, le problème est décidable et est calculable en PSpace-complet.

Ces propriétés sont beaucoup plus intéressantes, et bien que la complexité soit toujours élevée, il nous est désormais possible d'avoir un algorithme.

2.3 Approche

D'après le théorème 10 [2], on peut effectuer une réduction du problème comme ceci :

$$\exists PQI(Q, \mathscr{C}, S) \leftrightarrow PQI(Q, \mathscr{C}, S, V_{\{a\}})$$

$V_{\{a\}}$ est définie comme une instance fixée de la partie visible du schéma dont le domaine contient une seule valeur *a*. Ainsi, chaque relation visible est représentée dans l'instance par un tuple de la forme {a, ..., a}. Ainsi, au lieu de vérifier si une instance existe pour satisfaire $\mathscr{C}$ et Q, il nous suffit désormais de vérifier si l'instance $V_{\{a\}}$ satisfait C et Q.

De plus, la proposition 12 [2] nous permet d'effectuer une nouvelle transformation du problème :

$$PQI(Q, \mathscr{C}, S, V_{\{a\}}) \leftrightarrow \forall\ K\ in\ Chase_{vis}(\mathscr{C}, S, V)\ , K\ satisfies\ Q$$

Enfin, le lemme 13 [2] nous donne le problème du chase suivant :
K in $Chase_{vis}(\mathscr{C}, S, V)$ correspond homomorphiquement à $chase_{vis}(S, \mathscr{C}, S, V_{\{a\}})$ tel que $h(K) \subseteq chase_{vis}(S, \mathscr{C}, S, V_{\{a\}})$ pour un homomorphisme h.

Le chase est un algorithme de test et de vérification de dépendances d'implication dans un schéma. Il est définit comme ceci :

$$Chase_{vis}(\mathscr{C}, S, V)$$

C représente l'ensemble de contraintes devant être satisfaites, S représente le schéma et V représente une partie visible de S. Le chase retourne une nouvelle instance calculée à partir de V et des contraintes $\mathscr{C}$ portant sur S.

L'algorithme du chase consiste à ajouter des faits à l'instance de départ pour satisfaire les contraintes passées en paramètre du chase. Par la suite, l'unification consiste à inférer les règles décrites par les contraintes pour modifier des valeurs, toujours dans le but de satisfaire les contraintes. L'unification ne crée donc pas de faits, mais en modifie certains.

Exemple : Soit le schéma S ayant pour relations visibles R et S d'arité 2, et pour relations invisibles T et U d'arité 2. Les faits contenus dans l'instance I_a de départ sont donc les suivants :

$$\{R(a,a),\ S(a,a)\}$$

Les contraintes $C_{v,h}$ allant des relations visibles vers invisibles données en parallèle du schéma sont les suivantes :

$$R(x,y) \rightarrow T(y,z)$$

$$R(x,x), S(x,y) \rightarrow U(x,z)$$

Enfin, les contraintes $C_{h,v}$ données sont les suivantes :

$$U(x,y) \rightarrow R(x,z)$$

$$T(x,y) \rightarrow R(x,z), S(y,z)$$

Dans le passage de la boucle du chase, la première contrainte $R(x,y) \rightarrow T(y,z)$ n'est pas satisfaite. L'algorithme va donc créer l'homomorphisme avec lequel la contrainte pourra être satisfaite. Pour rappel, un homomorphisme correspond à un ensemble {variable → valeur}. Ici, l'homomorphisme est le suivant :

$$\{x \rightarrow a,\ y \rightarrow a\}$$

La partie de la contrainte à satisfaire est T(y, z). Avec l'homomorphisme précédent, on sait que y vaut a. En revanche, z n'a pas de valeur attribuée, et on va donc créer une nouvelle valeur que l'on va affecter à z. La nouvelle valeur sera ici b. Ainsi, après ces calculs, l'instance I contient donc les faits suivants :

$$\{R(a,a),\ S(a,a),\ T(a,b)\}$$

De la même manière, la seconde contrainte présentée ci-dessus n'est pas satisfaite. L'algorithme va donc rajouter un fait à l'instance I, qui contient désormais :

$$\{R(a,a),\ S(a,a),\ T(a,b),\ U(a,c)\}$$

Toutes les contraintes $C_{v,h}$ sont désormais satisfaites. L'algorithme va donc entamer l'étape de l'unification. On voit que la contrainte $T(x,y) \rightarrow R(x,z), S(y,z)$ n'est pas satisfaite. Pour rappel, l'unification ne crée pas de nouvelles valeurs, mais modifie les données invisibles pour les rendre cohérentes avec les contraintes et les données visibles. On trouve d'abord l'homomorphisme avec lequel la partie droite de la contrainte est satisfaite :

$$\{x \rightarrow a,\ y \rightarrow a,\ z \rightarrow a\}$$

Avec cette homomorphisme, on va créer un ensemble de couples ancienne valeur $\rightarrow$ nouvelle valeur à remplacer. Ici, on obtient l'ensemble suivant :

$$\{a \rightarrow a,\ b \rightarrow a\}$$

Enfin, on remplace les anciennes valeurs des faits de l'instance par les nouvelles valeurs. En remplaçant b par a, on obtient :

$$\{R(a,a),\ S(a,a),\ T(a,a),\ U(a,c)\}$$

L'ensemble des contraintes $C_{h,v}$ est satisfait, et l'algorithme s'arrête donc ici.

Il existe plusieurs types de chase [3], mais il n'existe actuellement pas d'outils pour manipuler le type de chase sur lequel nous nous concentrerons, c'est-à-dire le $chase_{vis}$.

2.4 Difficulté du problème

D'après le théorème 18 [2], la complexité du problème est PSpace-hard. Lorsque l'arité est fixée, elle est NP-dur.

Preuve : On réduit notre problème $\exists PQI$ à un problème d'inclusion de requêtes $Q_1 \subseteq Q_2$ pour un schéma d'arité 2. La valeur de l'arité a peu d'importance mais elle doit être fixée.

Soient Q_1 et Q_2 deux requêtes conjonctives booléennes et S un schéma d'arité 2. On définit alors le schéma invisible S_h égal à l'union de S et d'une table $HOKQ_1(x)$, et le schéma visible S_v égal à une vue $VOKQ_1(x)$

On définit alors la CQvue comme l'équivalence entre $VOKQ_1(x)$ et $HOKQ_1(x) \wedge Q_1$.

D'après le théorème 10 [2], le problème général

$$\exists PQI(S_h, S_v, \mathscr{C}, Q_2)$$

peut être réduit en ajoutant une instance du schéma où tous les tuples visibles sont initialisés avec une valeur par défaut, ici *a*. On obtient ainsi :

$$PQI(S_h, S_v, \mathscr{C}, Q_2, I_{\{a\}})$$

On peut par la suite réduire ce problème au problème du chase et de son unification d'après la proposition 12 [2]. De ce fait, on obtient :

$$Chase + unificiation(\mathscr{C}, I_{\{a\}}) \models Q_2$$

Après avoir effectué le calcul du chase et son unification, on obtient :

$$\{HOKQ_1(a), VOKQ_1(a)\} \wedge I_{Q_1} \models Q_2$$

I_{Q_1} est l'instance créée par le chase et l'unification. Comme toutes les valeurs de I_{Q_1} sont différentes de a, I_{Q_1} est isomorphique au corps de Q_1.

Dans la formule précédente $VOKQ_1(x) \leftrightarrow KQ_1(x) \wedge Q_1$, Q_1 ne partage pas de variables avec $HOKQ_1$ et $VOKQ_1$. Lors de l'unification, Q_1 ne sera donc pas modifiée. Ainsi, toutes les valeurs de I_{Q_1} sont différentes de la valeur par défaut, notée *a*. Ces relations n'appartiennent pas à l'ensemble des noms d'atomes de S. On peut donc enlever le membre $\{HOKQ_1(a), VOKQ_1(a)\}$:

$$I_{Q_1} \models Q_2$$

D'après [1], on obtient alors :

$$\exists h \; tq \; Q2 \xrightarrow{h} Q_1$$

D'après le théorème 6.2.3 [1], ce problème est finalement équivalent au problème d'inclusion de requêtes, dont la complexité est NP-dur :

$$Q_1 \subseteq Q_2$$

3 Vues formées à partir de requêtes conjonctives

Dans un premier temps, nous nous limiterons au cadre où les contraintes sont des CQvues. Comme montré précédemment, ce problème est décidable et le chase termine. La complexité est alors en NP-hard. Cela nous permet de définir un premier problème plus abordable et qui peut potentiellement donner de bons résultats.

3.1 Algorithmique

La première partie des travaux réalisés a porté sur l'implémentation d'un algorithme de vérification de secret où les vues sont des CQvues. Nous rappelons que l'algorithme $\exists PQI$ se termine dans le cas où l'arité est fixée, et qu'il est en NP-hard pour le cas avec les CQVues. L'algorithme est présenté ci-dessous :

Algorithm 1: Algorithme du chase et unification dans le cadre des CQvues

```
I ← I_a

for c in C_{v,h} do
    if I not satisfy c then
        I ← I + createFaitForMatching(I, c)
    end
end

added ← true
while added do
    added ← false
    association ← findConstraintHomomorphismAndFaitNotSatisfied(C_{v,h})

    if assocation not null then
        homomorphism ← createHomomorphism(association)
        modifyInstanceWithHomomorphism(I, homomorphism)
        added ← true
    end
end
```

La première partie de l'algorithme représente l'implémentation du chase. Pour chaque contrainte non satisfaite, l'algorithme va créer l'ensemble des faits qui vont satisfaire cette contrainte, et les ajouter à l'instance I.

La seconde partie représente l'unification. Cette partie ne crée pas de faits, mais les modifie en fonction des contraintes non satisfaites et des homomorphismes obtenus.

3.2 Expérimentation

Pour évaluer notre solution, nous avons mis en place un générateur de schémas et de contraintes. Il nous permet de construire automatiquement et de façon aléatoire un schéma comprenant des relations visibles et invisibles, et des contraintes pour former les CQvues. Le générateur prend en paramètre une série de valeurs pour spécifier plusieurs éléments :

- nbRelations : Le nombre de relations totale du schéma - visibles et invisibles.
- relationArity : L'arité des relations. Elle sera la même pour toutes les relations.
- nbConstraints : Le nombre de contraintes.
- contraintSize : Le nombre d'atomes dans le corps de la requête.
- visibleRelation : La proportion de relations visibles. Par calcul, on obtient la proportion de relations invisibles en soustrayant ce nombre à 1.

Ces paramètres nous offrent une grande modulation sur nos tests, ce qui nous a permis de tester de nombreux cas très facilement. Les résultats obtenus sont présentés ci-dessous. Pour chaque ensemble de valeurs de paramètres, nous avons lancé au moins 100 générations. Cela nous permet d'être confiant dans les valeurs obtenues dans le cas où certaines données créées seraient du bruit.

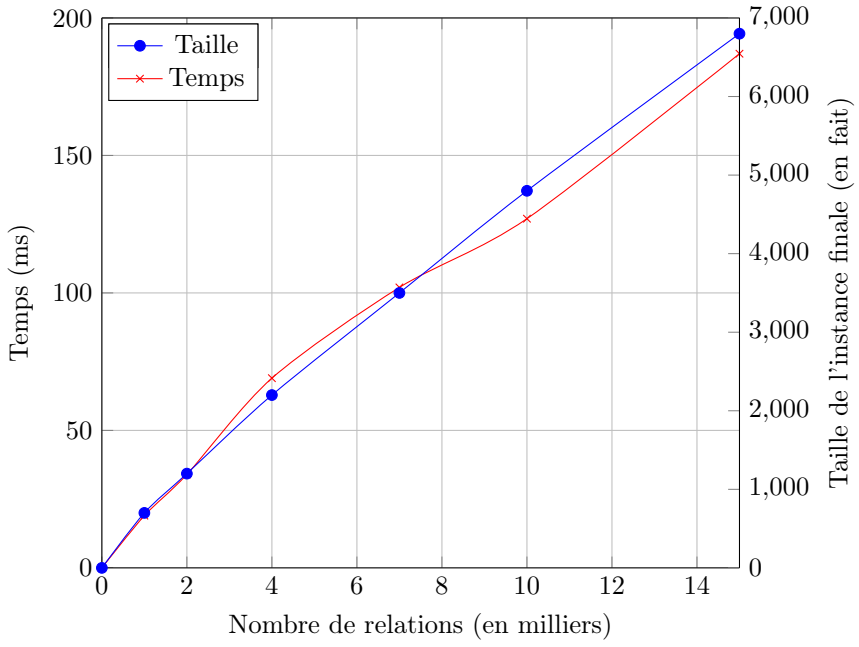

FIGURE 1 – Courbes représentant la taille finale de l'instance et le temps d'exécution en fonction du nombre de relations.

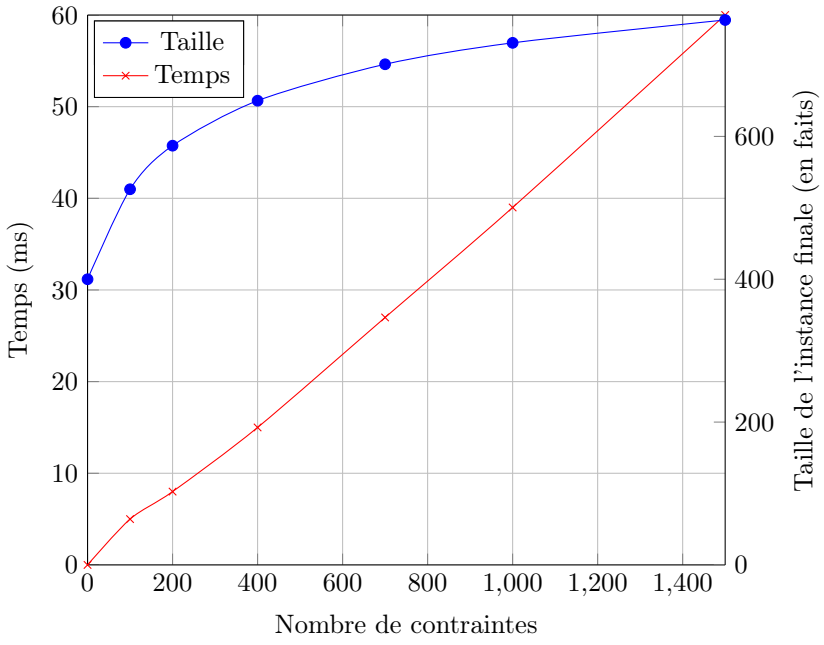

FIGURE 2 – Courbes représentant la taille finale de l'instance et le temps d'exécution en fonction du nombre de contraintes.

Paramètre	Taille de l'instance	Ecart-type	Temps	Ecart-type
Nombre de relations	Linéaire	de 16 pour 1 000 relations à 66 pour 15 000 relations	Linéaire	Entre 4 de 13
Arité des relations	Constante	Entre 16 et 18	Constant	Entre 5 et 6
Nombre de contraintes	Logarithmique	Entre 14 et 17	Linéaire	Entre 2 de 9
Taille de contrainte	Peudo-constant	Entre 15 et 19	Logarithmique	Entre 5 de 7
Proportion de relations visibles	Pseudo-affine	de 20 pour 0.1 à 9 pour 0.9	Pseudo-affine	Entre 5 et 7

FIGURE 3 – Tableau représentant les différentes tailles d'instances, temps d'exécution et écart-types selon le paramètre variant.

Sur la première figure, on analyse le temps mis pour que l'algorithme du chase se fasse, et le nombre de faits de l'instance finale en fonction de nombre de relations en entrée. Les autres paramètres ont été fixés comme ceci :

- relationArity : 4
- nbConstraints : 500
- constraintSize : 4
- visibleRelations 0.4

On constate que les 2 courbes sont presques linéaires.

Sur la deuxième figure, on analyse les mêmes éléments, mais cette fois-ci en fonction du nombre de contraintes. Les autres paramètres ont été fixés comme ceci :

- nbRelations : 1 000
- relationArity : 4
- constraintSize : 4
- visibleRelations 0.4

Lorsque l'on fait varier l'arité des relations, l'instance finale n'est, sans surprise, pas plus grande. Par contre, on constate que le temps d'exécution ne varie pas non plus. Pour 1 000 relations dont 40% de relations visibles, et 500 contraintes de taille 4, on obtient toujours une instance finale d'environ 670 éléments calculé en 19 ms.

La variation de la taille des contraintes influe peu sur la taille de l'instance finale, mais le temps d'exécution de l'algorithme augmente de manière logarithmique.

Enfin, lorsque le paramètre est la proportion de relations visibles, les fonctions représentant la taille de l'instance finale tout comme le temps mis à l'exécution sont pseudo-affines : elles sont affines hormis au point 0 où les 2 valeurs sont à 0. Cela s'explique facilement puisque, comme 0% des relations sont visibles, il n'y a pas d'instance initiale et donc pas de faits calculés.

La tableau présentée en haut de page récapitule les données obtenues en faisant varier les différents paramètres.

Pour tous les paramètres, on remarquera que l'écart-type très faible, que ce soit au niveau du temps d'exécution ou de la taille de l'instance. Cela nous conforte encore davantage dans nos données.

On remarquera également que les temps donnés sont très bons - quelques millisecondes - même pour des tailles de problèmes très grandes.

4 Contraintes de clef et vues atomiques

4.1 Problème

Le premier problème présenté ne prend pas en compte un élément très important des bases de données : les clefs étrangères. Comme expliqué précédemment, un clef étrangère est une contrainte qui garantie l'intégrité référencielle entre des données provenant de tables. L'étape suivante du projet avait donc pour but d'introduire ces clefs étrangères dans l'algorithme traitant des CQvues.

La manière de procédé est présenté ici, mais a déjà été démontré ailleurs (page 32 [2]). Pour cette variante du chase, il est plus simple de représenter $chase_{vis}(S, \mathscr{C}, S, V_{\{a\}})$ sous forme de graphe orienté où les noeuds représentent les faits du chase et les arcs décrivent les étapes qui créent de nouveaux faits à partir des faits existants et des clefs étrangères. Ainsi, les faits contenus dans l'instance de départ n'auront pas d'arcs entrants puisque ce sont les noeuds qui prendront place à la racine. De plus, on se réduit ici au TGDs [linéaires] et les faits ne sont donc générés qu'à partir d'un autre fait.

Par la suite, on ne construira pas le graphe complet, mais on se contentera de retenir la distance entre le fait généré et la racine. Les faits de $V_{\{a\}}$ ont donc une distance égale à 0.

Le problème principal lié aux clefs étrangères est qu'il est possible d'avoir des cycles de dépendance. Par exemple, avec trois relations invisibles R, S et T d'arité 2 et les clefs suivantes, certaines données de R dépendent de S, qui dépendent elles-même de T qui à leur tour dépendent de R. Cela crée un cycle de dépendance entre les relations :

$$\{R(a,b) \rightarrow S(a,a),\ S(a,b) \rightarrow T(a,c),\ T(a,a) \rightarrow R(b,a)\}$$

Calcul de limite : Ce cycle implique de devoir créer un nombre potentiellement infini de faits à partir d'autres faits. Pour cela, il est possible de couper l'arbre pour avoir un préfixe tel que si la requête est vraie, alors elle est vraie dans le préfixe. Nous avons par la suite cherché à trouver la hauteur maximale de l'arbre à partir de laquelle les faits générés seraient déjà présents dans le préfixe. D'après le lemme 5 [3], nous obtenons cette formule :

$$hauteur = |C| \,.\, |\epsilon| \,.\, (W\ +\ 1)^W$$

ϵ représente le nombre de contraintes de clefs, W représente la taille de la frontière et C représente le nombre de relations invisibles du schéma. La taille de l'arbre ainsi généré est $2^{hauteur}$. Ainsi, pour 50 relations invisibles, une frontière de taille 2 et 20 clefs étrangères, l'arbre contiendrait 2^{9000} noeuds.

Schéma Cristal : Pour simplifier cette formule, nous avons voulu travailler sur un schéma réel pour voir quelles étaient les bonnes pratiques et quels seraient les facteurs simplifiant ou amplifiant la complexité de la formule. Ainsi, nous avons obtenu le schéma de création des tables et de vues pour le laboratoire Cristal [1]. Ce schéma concerne la gestion des équipes, des sujets traitées par celles-ci, des chercheurs et doctorants, etc.

En analysant ce schéma, nous avons tout d'abord trouvé que la frontière était toujours égale à 1. En effet, la manière avec laquelle le schéma était construit impliquait qu'une relation contenait toujours une clef primaire, et que c'était cette clef primaire qui était reprise en tant que clef étrangère dans les autres tables.

La suite de notre analyse a porté sur le nombre de tables et de vues du schéma. Il possède 93 tables et 63 vues. Bien que ces éléments ne nous servent pas directement pour simplifier la hauteur de l'arbre, ils nous seront utiles au moment de faire des tests.

1. https ://www.cristal.univ-lille.fr/

La deuxième amélioration de la formule de calcul de la hauteur a porté sur le calcul du plus grand cycle de contraintes. En obtenant cette valeur, il est possible de réduire la limite en remplaçant ϵ par la taille du plus grand cycle. Avec ces deux améliorations, la formule finale est donc :

$$hauteur \ = \ taille_{biggest\ cycle} \ . \ \epsilon \ . \ 2$$

Enfin, nous avons cherché à calculer la taille du plus grand cycle contenu dans le schéma Cristal, puisque la taille du plus grand cycle est déterminante pour la hauteur de l'arbre. Nous avons ainsi trouvé 2 cycles, un cycle de taille 1 et un cycle de taille 2. C'est un élément très important qui nous permettra de pouvoir faire des estimations plausibles, et de pouvoir tester par la suite le générateur sur des cas réels.

Ainsi, avec un plus grand cycle de taille 3 et ϵ inchangée à 20, l'arbre ne contiendrait plus au maximum que 2^{120} noeuds.

4.2 Algorithmique

L'algorithme ainsi implémenté est présenté ci-dessous :

Algorithm 2: Algorithme du chase et unification dans le cadre des clefs étrangères avec des vues atomiques

```
I ← I_a

for c in C_{v,h} do
    if I not satisfy c then
        I ← I + createFaitForMatching(I, c)
    end
end

added ← true
while added do
    added ← false
    association ← findKeyConstraintHomomorphismAndFaitNotSatisfied(C_{h,h})

    if assocation not null then
        I ← I + createFaitForMatching(association)
        added ← true
    end
end

added ← true

while added do
    added ← false
    association ← findConstraintHomomorphismAndFaitNotSatisfied(C_{v,h})

    if assocation not null then
        homomorphism ← createHomomorphism(association)
        modifyInstanceWithHomomorphism(I, homomorphism)
        added ← true
    end
end
```

Par rapport à l'algorithme du chase précédemment présenté, la première boucle while a été rajoutée. C'est la boucle qui permet d'ajouter un fait lorsqu'une contrainte de clef n'est pas satisfaite. L'unification est faite ensuite.

4.3 Exemple

Soit le schéma S ayant pour relations visibles R et S d'arité 2, et pour relations invisibles T et U d'arité 2. Les faits contenus dans l'instance I_a de départ sont donc les suivants :

$$\{R(a,a)_0,\ S(a,a)_0\}$$

Le nombre à la fin du fait indique à quel hauteur de l'arbre il a été calculé.

Les contraintes $C_{v,h}$ allant des relations visibles vers invisibles données en parallèle du schéma sont les suivantes :

$$R(x,y) \rightarrow T(y,z)$$

$$S(x, y) \rightarrow U(x, z)$$

De même, les contraintes $C_{h,v}$ données sont les suivantes :

$$U(x, y) \rightarrow R(x, z)$$
$$T(x, y) \rightarrow S(y, z)$$

Enfin, les contraintes de clefs étrangères sont les suivantes :

$$U(y, x) \rightarrow T(x, y)$$
$$T(x, y) \rightarrow U(x, z)$$

Les premiers faits générés sont ceux créés par les contraintes de vues, comme pour l'algorithme du chase pour les CQvues. Ici, on obtient l'instance intermédiaire suivante :

$$\{R(a, a)_0,\ S(a, a)_0,\ T(a, b)_0,\ U(a, c)_0\}$$

La seconde étape consiste donc à prendre une clef étrangère non satisfaite, et de créer le fait pour qu'elle le soit. Prenons par exemple la contrainte $U(x, x) \rightarrow T(z, x)$. Elle n'est pas satisfaite. On rajoute donc le fait $T(d, c)$ et on obtient l'instance :

$$\{R(a, a)_0,\ S(a, a)_0,\ T(a, b)_0,\ U(a, c)_0, T(d, c)_1\}$$

Désormais, la contrainte $T(x, y) \rightarrow U(x, z)$ n'est plus satisfaite. On rajoute donc à l'instance le fait pour la satisfaire :

$$\{R(a, a)_0,\ S(a, a)_0,\ T(a, b)_0,\ U(a, c)_0, T(d, c)_1, U(d, e)_2\}$$

A nouveau, la contrainte $U(x, x) \rightarrow T(z, x)$ n'est plus satisfaite. Ici, l'exemple est potentiellement infini. Cela est dû aux clefs étrangères qui forment un cycle de dépendance entre les 2 relations T et U. Supposons que le secret à vérifier est de taille 1, la hauteur de l'arbre à partir de laquelle on peut s'arrêter de calculer de nouveaux faits est de 2. Dans cette exemple, on ne peut donc plus ajouter de nouveaux faits à l'instance. La partie sur la vérificaiton des clefs étrangères s'arrête donc ici.

De la même manière que pour les CQVues, on unifie désormais les faits selon les contraintes $C_{h,v}$. L'instance finale ainsi calculée est donc :

$$\{R(a, a)_0,\ S(a, a)_0,\ T(a, a)_0,\ U(a, c)_0, T(d, a)_1, U(a, e)_2\}$$

L'ensemble des contraintes $C_{h,v}$ est satisfait, et l'algorithme s'arrête donc ici.

4.4 Expérimentation

Nous avons mené des tests de performance avec notre générateur précédemment présenté. Etant donné la complexité du problème et le nombre de faits potentiellement générés, nous avons voulu nous limiter à un cas réel d'utilisation. Ainsi, nous avons repris les données extraites du schéma Cristal pour générer des schémas proches de la réalité. Les paramètres utilisés sont donc les suivants :

- nbRelations : 200
- relationArity : 8
- nbConstraints : 70
- contraintSize : 1
- visibleRelation : 0.4

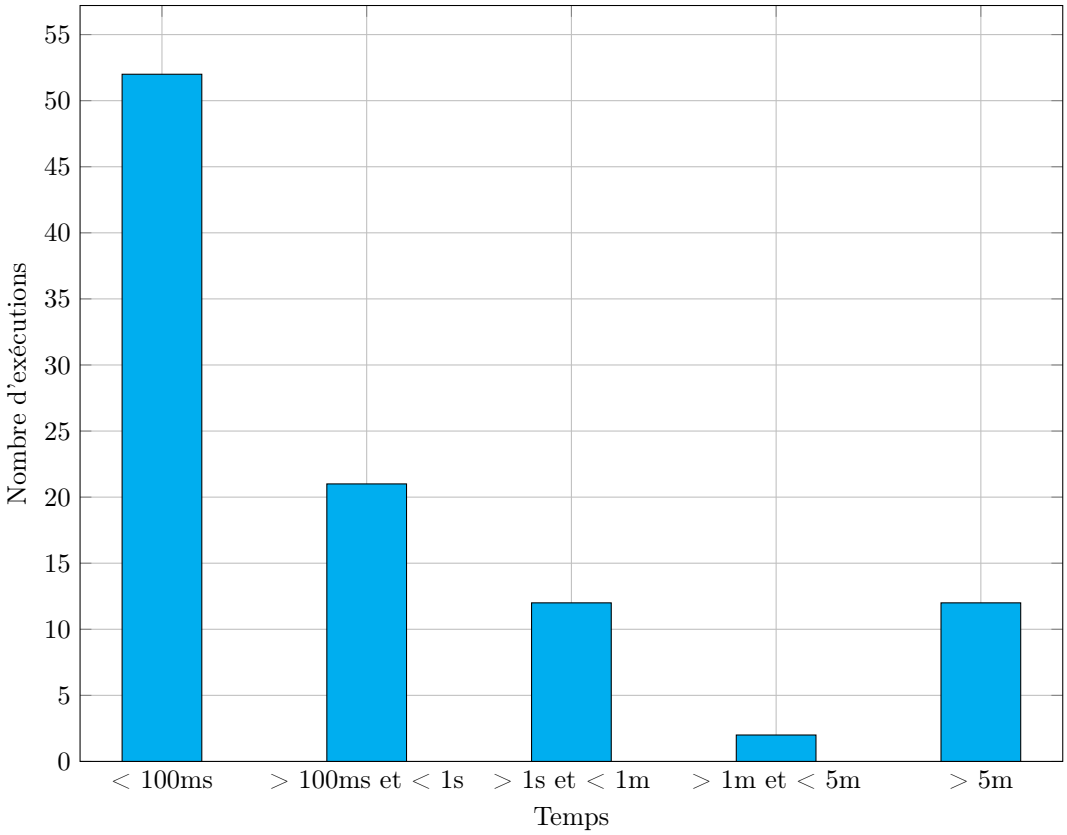

FIGURE 4 – Histogramme comptant le nombre d'exécutions en fonction du temps d'exécution de l'algorithme.

A ces paramètres déjà existants, nous avons ajouté de nouveaux paramètres répondant à l'ajout des clefs étrangères.

- smallestCycleSize : la taille du plus petit cycle de dépendance entre relations. Il est donc possible d'avoir des cycles plus grand.
- secretSize : la taille du secret.

Dans un premier temps, nous avons testé en modifiant plusieurs fois ces deux derniers paramètres pour voir quel était le temps d'exécution moyen. On obtient alors l'histogramme ci-dessus :

L'histogramme ci-dessous nous montre que les résultats sont en moyenne très bons. Dans 75% des cas, l'algorithme termine en moins d'une seconde. Dans le cas où l'exécution dépasse 5 minutes, nous avons stoppé l'exécution et ne savons donc pas combien de temps cela a pris. Néanmoins, pour un cas donné (plus grand cycle de taille 4), nous avons laissé l'algorithme fonctionner jusqu'au bout sans l'arrêter du tout : celui-ci a mis **plus de 3h30**.

Par la suite, nous avons trié ces données selon la taille du plus grand cycle. Nous obtenons la courbe présentée page suivante :

La courbe qui représente le temps d'exécution en fonction de la taille du plus grand cycle est très clairement exponentielle avec des temps en moyenne très faibles pour des cycles de petites taille - quelques millisecondes pour des cycles de taille 0, 1 ou 2 - et des temps en moyenne importants pour des cycles de taille 6 ou 7.

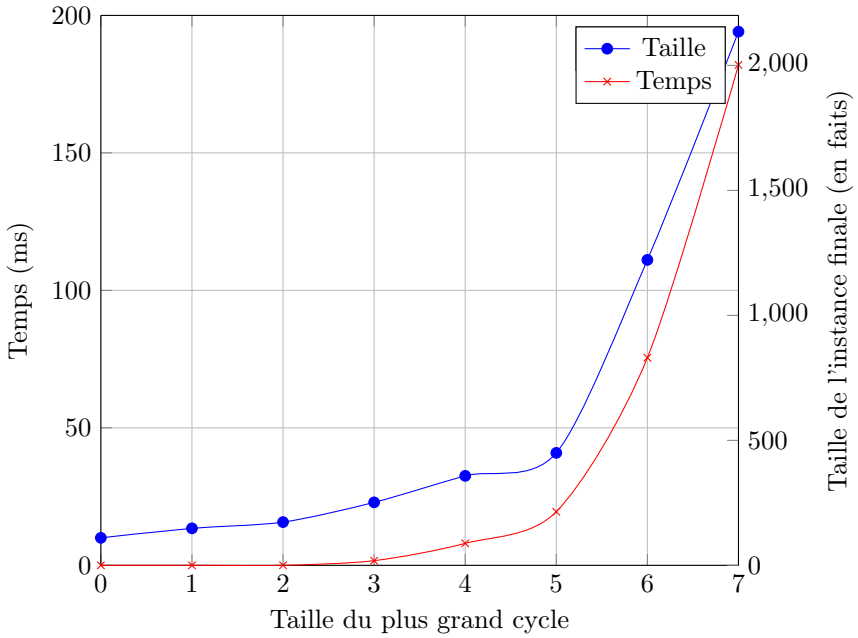

FIGURE 5 – Courbes représentant la taille de l'instance finale et le temps d'exécution en fonction de la taille du plus grand cycle.

5 Discussion et conclusion

Pour ce projet, nous nous sommes intéressés au problème de fuites de données dans une base de données. Ces problèmes, formalisés récemment sous le nom de $\exists PQI$ et PQI ont une complexité très élevée ou sont indécidables. Nous avons réduit nos problèmes selon des restrictions qui englobent des cas très utilisés en pratique. Grâce à ces réductions, nous avons été capables de montrer la faisabilité de ces problèmes en une complexité moindre et avons ainsi pu produire 2 algorithmes pour évaluer ce problème : le premier avec des CQvues, le second avec des clefs étrangères.

Par la suite, nous avons mené une évaluation de nos algorithmes avec une grande quantité de schémas et de contraintes grâce à un générateur. Les données récoltées dans le cas du problèmes des CQVues montrent une très bonne faisabilité même avec des paramètres ayant des valeurs élevées. Néanmoins, le problème des clefs étrangères est, comme attendu, nettement moins performant. Par manque de temps, nous n'avons pas pu mener l'analyse de ce second problème jusqu'au bout. Il aurait pu être intéressant d'affiner l'analyse pour voir quels autres paramètres pourrait avoir un impact sur la performance de l'algorithme.

Nous avons déjà effectué plusieurs optimisations, comme la réduction de la hauteur de l'arbre des faits dans le cadre des clefs étrangères ou l'étude d'un cas pratique, mais il est encore possible d'améliorer l'algorithme.

La première amélioration serait d'unifier les faits générés avec les clefs étrangères juste après leur création. Cela ralentirait certes la boucle de création des faits, mais cela permettrait de ne pas générer un fait qui, unifié, serait déjà existant dans l'instance. Pour des grosses instances, le gain de temps serait sûrement significatif.

La seconde amélioration possible serait de passer en paramètre du problème les relations contenues dans le secret que le pirate cherche à connaître. En connaissant cette information, il serait possible de ne pas

calculer certains faits dont la relation n'appartient pas au secret ou ne génère pas d'autres faits dont la relation appartient au secret.

Une fois ces améliorations faites, nous pourrons envisager de fusionner l'algorithme sur les CQVues avec l'algorithme sur les clefs étrangères. Dans les dernières semaines des travaux, nous avons vu que théoriquement, cela était décidable et donc faisable. Néanmoins, il n'est actuellement pas possible d'extraire un préfixe d'un chase où les contraintes sont des CQvues et des clefs étrangères. Dans un premier temps, il faudrait donc réduire le problème à un problème sans cycle. Cela nous donnerait un algorithme pour traiter les cas généraux qui sont actuellement utilisés pour construire des schémas de bases de données.

Références

[1] Serge Abiteboul, Richard Hull et Victor Vianu. *Foudations of Databases*. Addison-Wesley Publishing Company, 1995.

[2] Michael Benedikt, Pierre Bourhis, Balder ten Cate et Gabriele Puppis. « Querying Visible and Invisible Tables in the Presence of Integrity Constraints ». In : *Logic in Computer Science* (sept. 2015).

[3] D. S. Johnson et A. Klug. *Testing containment of conjunctive queries under functional and inclusion dependencies*. 1984.