Bases de données avancées

Projet

Modalités

Le projet est à réaliser en binômes.

Le rendu sera sans doute pendant la session de Soutenances (entre le 19 et 22 mai) ; les modalités définitives vous seront communiquées ultérieurement.

Pour toute question ou doute à propos du sujet, nous vous encourageons fortement à utiliser le forum dédié sur Moodle pour que tout le monde puisse profiter des réponses.

Vos projets doivent être hébergés sur le gitlab de l'UFR.
Tous les chargés de TP (leur noms figurent sur le moodle du cours) doivent être invités en tant que `Reporter`.
Le fichier `README` ou `README.md` doit contenir les noms et prénoms ainsi que le groupe de TP des auteurs du projet, pour chaque membre de binôme.

1 Le but du projet

Nous avons vu en cours une procédure appelée la *chase* permettant de tester si une décomposition est sans perte d'information (poly *Normalisation* transparents 58 à 70). Cet algorithme a été proposé pour la première fois en 1979, à la fois par Aho et al. 1979 d'une part et Maier et al. 1979 d'une autre. Depuis, la *chase* a trouvé des applications dans de nombreux domaines (théoriques et appliqués) des bases de données. La diversité de ces applications a donné lieu à maintes variations et optimisations, comme la *oblivious chase*, la *oblivious Skolem chase*, ou encore la *core chase*.

Ce projet est composé de deux parties.
Dans la première partie, vous devez implémenter une généralisation de l'algorithme de *chase* vu en cours, expliqué dans la section 3.

Dans la deuxième partie, vous devez implémenter une ou deux extensions au choix parmi celles proposées dans la section 4. Cette partie n'est pas optionnelle ! Si vous n'implémentez aucune extension, vous n'aurez très probablement pas la moyenne.

Vous pouvez choisir votre langage d'implémentation selon vos préférences parmi la liste suivante :

- Java
- C++
- OCaml
- Go
- Python

Si vous voulez utiliser un autre langage, vous pouvez faire une demande justifiée à votre chargé de TD/TP pour avoir (ou non !) l'autorisation.

Pour représenter et manipuler votre base de données, vous pouvez soit implémenter vos propres structures de données dans votre langage choisi, soit utiliser postgres, auquel cas votre programme devra se connecter à la base à l'aide d'un *driver* (une liste est disponible ici.) Vous pouvez trouver une courte introduction aux *drivers* dans le chapitre 5 du livre *Database Systems* Concepts par Silberschatz, Korth and Sudarshan, 6$^{\text{ème}}$ edition, McGraw-Hill. Vous aurez aussi sûrement envie d'étendre postgres avec un nouveau type pour gérer les valeurs inconnues. Vous pouvez trouver toutes les informations nécessaires pour ça dans les sections *CREATE TYPE* et 8.16. Composite Types de la documentation postgres.

En général, il n'est pas possible de déterminer par avance si une exécution de la *chase* va terminer ou non. De plus, l'algorithme étant non-déterministe, certaines exécutions peuvent mener à des comportements infinis, tandis que d'autres terminent pour les mêmes arguments. Pensez donc à prévoir un mécanisme d'arrêt qui pourrait être, par exemple, manuel, une limite de temps, une limite de nombre d'applications de contraintes, ou autre.

Les algorithmes des sections 3 et 4 sont définis informellement. Si besoin, vous pouvez trouver les définitions formelles en annexe. Ces définitions sont là pour vous aider en cas de doute ou confusion, il n'est pas nécessaire de les comprendre pour pouvoir implémenter les algorithmes.

2 Un mot sur les contraintes

En cours nous avons vu un seul type de contrainte : les *dépendances fonctionnelles* (DF).

Définition 2.1 Dépendance Fonctionnelle – Cours *Normalisation*, p. 12

Soit $R(U)$ un schéma de relation avec U son ensemble d'attributs.
Une *dépendance fonctionnelle* (DF) est une expression de la forme $X \to Y$, avec $X, Y \subseteq U$. Une instance J de $R(U)$ satisfait $X \to Y$ si pour toute paire de tuples $t, u \in J$, $t[X] = u[X] \Rightarrow t[Y] = u[Y]$. En d'autres mots, si t et u sont en accord sur X, alors ils doivent aussi être en accord sur Y.

L'algorithme de *standard chase* est défini pour une classe plus générale que les DF – les *dépendances engendrant des égalités* ou EGD (en anglais *Equality Generating Dependency*), qui, comme leur nom l'indique, expriment des contraintes *d'égalité* entre éléments de la base de données. Contrairement aux DF, celles-ci peuvent exprimer des contraintes sur des éléments de relations différentes.

Pour rappel, une *conjonction* est une suite de termes séparés par le connecteur binaire *et*, noté $\wedge$.

Définition 2.2 Dépendance engendrant des égalités – Abiteboul 2000

Une *dépendance engendrant des égalités* est une formule de la logique du premier ordre qui a la forme suivante :

$$\forall \vec{x}\ \forall \vec{y}\ \phi(\vec{x}, \vec{y}) \to \exists \vec{z}\ \psi(\vec{x}, \vec{z})$$

où $\vec{x}, \vec{y}$ et $\vec{z}$ sont deux à deux disjoints, $\phi(\vec{x}, \vec{y})$ est une conjonction d'atomes relationnels et d'égalités et $\psi(\vec{x}, \vec{z})$ est une conjonction d'atomes d'égalité seulement. Un atome relationnel est de la forme $R(\vec{w})$ où chaque w_k est une variable ou une constante. Un atome d'égalité est de la forme $w_i = w_j$ où chaque w_k est une variable ou une constante.

On dit que $\phi(\vec{x}, \vec{y})$ est le *corps* de la dépendance, tandis que $\psi(\vec{x}, \vec{z})$ s'appelle sa *tête*.

Vous avez sûrement remarqué que ϕ et ψ prennent deux arguments et que les variables sont réparties en trois domaines $\vec{x}, \vec{y}$ et $\vec{z}$. Cette distinction existe pour pouvoir facilement différencier les variables en commun entre ϕ et ψ.

Exemple 2.3

La dépendance fonctionnelle *IdEtu* $\rightarrow$ *NomEtu* peut aussi s'écrire comme

$$\forall \texttt{IdEtu}, \texttt{NomEtu}_1, \texttt{NomEtu}_2$$
$$(\texttt{Etu}(\texttt{IdEtu}, \texttt{NomEtu}_1) \wedge \texttt{Etu}(\texttt{IdEtu}, \texttt{NomEtu}_2)) \rightarrow \texttt{NomEtu}_1 = \texttt{NomEtu}_2$$

Ici $\vec{x} = \{\texttt{NomEtu}_1, \texttt{NomEtu}_2\}$, $\vec{y} = \{\texttt{IdEtu}\}$, $\vec{z} = \emptyset$,
$\phi(\vec{x}, \vec{y}) = (\texttt{Etu}(\texttt{IdEtu}, \texttt{NomEtu}_1) \wedge \texttt{Etu}(\texttt{IdEtu}, \texttt{NomEtu}_2))$,
et $\psi(\vec{x}, \vec{z}) = \texttt{NomEtu}_1 = \texttt{NomEtu}_2$.

En plus des contraintes d'égalités, nous allons nous intéresser aux contraintes d'existence de tuples appelées les *dépendances engendrant des tuples* ou TGD (en anglais *Tuple Generating Dependency*). À la différence des EGD, une TGD ne contient que des *atomes relationnels* qui expriment l'existence d'un (ou de) tuple(s) dans la base de données.

Définition 2.4 Dépendance engendrant des tuples – Abiteboul 2000

Une *dépendance engendrant des tuples* est une formule de la logique du premier ordre qui a la forme suivante :

$$\forall \vec{x}\ \forall \vec{y}\ \phi(\vec{x}, \vec{y}) \rightarrow \exists \vec{z}\ \psi(\vec{x}, \vec{z})$$

où où $\vec{x}, \vec{y}$ et $\vec{z}$ sont deux à deux disjoints, $\phi(\vec{x}, \vec{y})$ et $\psi(\vec{x}, \vec{z})$ sont des conjonctions d'atomes relationnels, c'est à dire de la forme $R(\vec{w})$ où chaque w_k est une variable ou une constante.

Comme pour les EGD, $\phi(\vec{x}, \vec{y})$ est le *corps* de la contrainte, $\psi(\vec{x}, \vec{z})$ est sa *tête*.

Exemple 2.5

La TGD suivante

$$\forall \texttt{IdLect}, \texttt{IdLivre}$$
$$\texttt{Emprunt}(\texttt{IdLect}, \texttt{IdLivre}) \rightarrow \exists \texttt{Nom}, \texttt{Auteur}\ \ \texttt{Livres}(\texttt{IdLivre}, \texttt{Titre}, \texttt{Auteur})$$

dit qu'un lecteur ne peut pas emprunter un livre qui n'existe pas.
Plus précisément, si il y a un tuple de la forme (`IdLect`, `IdLivre`) dans notre relation `Emprunt` alors il doit aussi y avoir un tuple de la forme (`IdLivre`, `Titre`, `Auteur`) dans notre relation `Livres`. Ici, $\vec{x} = \{\texttt{IdLect}, \texttt{IdLivre}\}$, $\vec{y} = \emptyset$, $\vec{z} = \{\texttt{Titre}, \texttt{Auteur}\}$,
$\phi(\vec{x}, \vec{y}) = \texttt{Emprunt}(\texttt{IdLect}, \texttt{IdLivre})$ et
$\psi(\vec{x}, \vec{z}) = \texttt{Livres}(\texttt{IdLivre}, \texttt{Titre}, \texttt{Auteur})$.

Remarque 2.6

Les domaines ($\vec{x}, \vec{y}$ et $\vec{z}$) sont souvent laissés implicites. La contrainte de l'exemple 2.3 peut donc s'écrire comme

$$(\texttt{Etu}(\texttt{IdEtu}, \texttt{NomEtu}_1) \wedge \texttt{Etu}(\texttt{IdEtu}, \texttt{NomEtu}_2)) \rightarrow \texttt{NomEtu}_1 = \texttt{NomEtu}_2$$

et celle de l'exemple 2.5 comme

$$\texttt{Emprunt}(\texttt{IdLect}, \texttt{IdLivre}) \rightarrow \texttt{Livres}(\texttt{IdLivre}, \texttt{Titre}, \texttt{Auteur})$$

3 La standard chase

La *standard chase* diffère de l'algorithme vu en cours par deux aspects. D'une part, le but de l'algorithme n'est plus de vérifier l'intégrité des informations, mais de *réparer* une base de données dans laquelle des informations sont potentiellement manquantes ou erronées. D'une autre part, il faut maintenant déterminer le type de la dépendance (EGD ou TGD) non-satisfaite et appliquer la règle appropriée.

Algorithme 3.1 Standard Chase

ENTRÉE : Une base de données D et un ensemble de contraintes Σ

ALGORITHME :

```
tant que il existe une contrainte e ∈ Σ et
          un ensemble de tuples T ∈ D t.q.
          (T satisfait le corps de e
          et T ne satisfait pas la tete de e
          et e n'a pas encore ete appliquee a T)
    si e est une TGD
        ajouter un nouveau tuple u a D
                t.q. D ∪ u satisfait e
    si e est une EGD
        egaliser* les tuples de T en accord avec e
si D satisfait Σ, retourner vrai
sinon retourner faux
```

*C'est à dire, pour chaque paire de variables w_i, w_j telles que $w_i = w_j$ apparaît dans la contrainte, remplacer l'élément du tuple correspondant à w_i par celui correspondant à w_j (ou l'inverse). Si l'un des deux éléments est une constante, on la choisit toujours comme "remplaçant", sinon le choix est libre.

Remarques 3.2

Pour les TGD, si le nouveau tuple doit contenir des éléments qui ne sont pas dans le corps, il faut générer une "valeur inconnue" (aussi appelé un *NULL*) différentes de toutes les autres "valeurs inconnues". Attention, ces *NULL* sont un peu différents des *NULL* classiques de SQL qui ne peuvent pas être égalisés.

Pour les EGD, il ne faut pas oublier d'égaliser toutes les variables, pas seulement celles qui apparaissent dans les tuples manipulés. Par exemple, quand deux NULL ont été précédemment égalisés, si l'un d'entre eux est égalisé avec une autre valeur alors les deux le sont.

Exemple 3.3

Soit une base de données D avec trois relations

$R[A, B]$, $P[C, D]$ et $Q[E, F, G]$

et trois contraintes

$c_1 = R(x_1, x_2) \rightarrow Q(x_1, x_2, z_1)$ (si R contient un tuple (a, b) alors il doit y avoir un tuple dans Q qui commence par a, b)

$c_2 = Q(y_1, x_1, y_2) \rightarrow P(x_1, z_1)$ (si Q contient un tuple (a, b, c) alors il doit y avoir un tuple dans P dont le premier élément est b)

$c_3 = R(x_1, y_1) \wedge P(y_2, x_2) \wedge y_1 = y_2 \rightarrow x_1 = x_2$ (si le deuxième élément d'un tuple de R et égal au premier élément d'un tuple de P, alors le premier élément du tuple de R doit être égal au deuxième élément du tuple de P)

Soit I une instance de D qui contient un seul tuple :

$R(a, b)$

l'exécution de la *standard chase* sur I sous les contraintes $\{c_1, c_2, c_3\}$ se déroule comme suit.

Comme les relations P et Q sont vides, les contraintes c_2 et c_3 sont trivialement satisfaite et ne peuvent donc pas être appliquées.

Le corps de c_1 est satisfait par le tuple (a, b) de R donc, comme sa tête n'est pas satisfaite, c_1 peut être appliquée. c_1 est une TGD donc son application consiste à rajouter un nouveau tuple de la forme précisée dans la tête, c'est-à-dire $Q(x_1, x_2, z_1)$. Ici, $x_1 = a$, $x_2 = b$ et z_1 est une nouvelle valeur qu'on va noter $\perp_1$. On ajoute donc le tuple $(a, b, \perp_1)$ à Q.

Notre instance contient les tuples suivants :

$R(a, b)$ $Q(a, b, \perp_1)$

La contrainte c_3 ne peut toujours pas être appliquée et c_1 l'a déjà été pour le seul tuple possible. Cependant, l'ajout d'un tuple à Q à rendu l'application de c_2 possible. c_2 est aussi une TGD donc on ajoute un nouveau tuple de la forme (x_1, z_1) à P où $x_1 = b$ et z_1 est une nouvelle valeur qu'on va noter $\perp_2$. On ajoute donc $(b, \perp_2)$ à P.

Notre instance contient les tuples suivants :

$R(a, b)$ $P(b, \perp_2)$ $Q(a, b, \perp_1)$

On peut maintenant appliquer c_3, qui est une EGD donc son application consiste à égaliser les éléments précisés par la tête. Ici, $x_1 = a, y_1 = b, x_2 = \perp_2$ et $y_2 = b$. On égalise x_1 avec x_2, c'est à dire en fait a avec $\perp_2$. Comme on privilégie les constantes aux variables, on va remplacer chaque élément $\perp_2$ de I par a.

Notre instance contient les tuples suivants :

$R(a, b)$ $P(b, a)$ $Q(a, b, \perp_1)$

Toutes les contraintes étant satisfaites, la *standard chase* termine et la base de données est réparée.

4 Extensions

La *standard chase*, comme définie dans la section 3, a mené à de nombreuses variantes. Certaines sont d'intérêt plutôt théorique, comme la *oblivious chase*, qui répond à la question "que se passe-t-il si on applique les contraintes même si elles sont déjà satisfaites ?". D'autres sont d'intérêt plutôt pratique, comme la *core chase* dont le but est de déterminiser l'algorithme.

Les extensions proposées ci-dessous sont annotées par des $\star$ (allant de $\star$ à $\star\star\star$). Le but de cette classification est de donner une idée de la difficulté d'implémentation *après avoir implémenté la standard chase*.

4.1 Oblivious chase $\star$

La *Oblivious chase* (qui pourrait s'appeler en français la *chase indifférente*), introduite par Marnette 2009, est une modification de la *standard chase* qui ne vérifie pas que la tête d'une contrainte n'est pas satisfaite pour l'appliquer. C'est à dire qu'une contrainte peut s'appliquer alors qu'il y a déjà un tuple dans la base de donnée qui la satisfait.

Attention, cette variante n'est définie que pour les TGD !

Exemple 4.1 Greco et al. 2012, exemple 4.12

Soit la contrainte $c = E(x, z) \rightarrow \exists y E(x, y)$ et la base de données $D = \{E(a, b)\}$.
Le corps de c est satisfait dans D par le tuple $E(a, b)$ donc on peut l'appliquer. On ajoute donc un tuple $(a, \perp_1)$ dans la relation E.
Le corps de c est toujours satisfait dans D, maintenant par le nouveau tuple $E(a, \perp_1)$, donc on peut encore l'appliquer. On ajoute un tuple $(a, \perp_2)$ dans la relation E.
Le corps de c est maintenant satisfait par $E(a, \perp_2)$, ainsi que tous les prochains tuples générés, *ad nauseam*.

Remarque 4.2

Cette variante à tendance à produire beaucoup de résultats infinis. Pensez donc à prévoir un mécanisme d'arrêt qui pourrait être, par exemple, manuel, une limite de temps, une limite de nombre d'applications de contraintes, ou autre.

4.2 Oblivious Skolem chase ⋆⋆

La *Oblivious Skolem chase*, aussi dûe à Marnette 2009, est une modification de la *oblivious chase* qui a pour but de limiter certains comportements infinis. Celle-ci génère des NULL non pas de façon arbitraire mais en fonction de $\vec{x}$ et de la contrainte. Ainsi, si une contrainte est satisfaite par plusieurs tuples qui partagent le même domaine $\vec{x}$, et qu'elle à déjà été appliquée une fois pour l'un d'entre eux, l'appliquer une fois de plus va générer un tuple déjà existant et la base de données sera inchangée.
Attention, cette variante n'est définie que pour les TGD !

Exemple 4.3 Greco et al. 2012, exemple 4.13

Soient c et D comme dans l'exemple 4.1.
La première application de c génère le tuple $(a, f(a))$, c'est à dire $(a, \perp_1)$ comme pour la oblivious chase.
La deuxième application de c, pour le tuple $(a, \perp_1)$, génère le tuple $(a, f(a))$, c'est à

dire $(a, \perp_1)$ étant donné que f est une fonction et retourne donc toujours le même résultat pour la même entrée.
L'algorithme peut maintenant s'arrêter comme il n'y a plus de tuple auquel c n'a pas été appliquée.

4.3 Oblivious et et oblivious Skolem pour les EGD ⋆⋆

Les trois algorithmes ci-dessus ne sont définis que pour les TGD. Une façon de les étendre aux EGD, serait d'ajouter une relation temporaire de la forme $E[A, B]$ pour représenter l'égalité des variables deux à deux afin de pouvoir réécrire chaque EGD comme une TGD, puis, à la fin de l'exécution, égaliser deux à deux toutes les variables (x_i, x_j) qui apparaissent comme tuple de la table E.

Exemple 4.4

Étant donné une relation $E[A, B]$, la contrainte de l'exemple 2.3 peut se réécrire comme

$$(\texttt{Etu}(\texttt{IdEtu}, \texttt{NomEtu}_1) \wedge \texttt{Etu}(\texttt{IdEtu}, \texttt{NomEtu}_2)) \rightarrow E(\texttt{NomEtu}_1, \texttt{NomEtu}_2)$$

Soit deux tuples $t_1 = \texttt{Etu}(1, \texttt{Alice})$ et $t_2 = \texttt{Etu}(1, \perp_1)$ d'une instance I.
Appliquer la contrainte ci-dessus à $\{t_1, t_2\}$, revient à ajouter le tuple $(\perp_1, \texttt{Alice})$ dans la table E.
Une fois la chase terminée, on remplace chaque $\perp_1$ dans I par `Alice` puisque la table E contient le tuple $(\perp_1, \texttt{Alice})$.

4.4 Core chase ⋆⋆⋆

Dans le but de pallier au non-déterminisme de la *standard chase*, Deutsch et al. 2008 ont introduit la *core chase* qui, à chaque tour de boucle, évalue l'union des résultats de toutes les applications possibles de contraintes, puis calcule le *noyau* (*core*) de la base de données ainsi obtenue. De cette façon, pour une même base de données et un même

ensemble de contraintes, toutes les exécutions de la *core chase* seront identiques. Intuitivement, le *core* d'une base de données est son plus petit sous-ensemble qui vérifie encore les contraintes. Un algorithme possible pour le calculer est le suivant :

Algorithme 4.5 Trouver le *core* – Arenas et al. 2014

Entrée : Une base de données D et un ensemble de contraintes Σ

Algorithme :

```
pour chaque tuple t de D
        retirer temporairement t de D
        si D satisfait toujours Σ
            retirer t de D de facon permanente
        sinon
            remettre t dans D
retourner D
```

Exemple 4.6 Arenas et al. 2014, exemple 6.12

Soit une base de données D avec deux relations

$R[A, B]$ et $S[C, D]$

et une seule contrainte $c = R(x_1, y_1) \rightarrow S(x_1, z_1)$

Soit I une instance de D qui contient les tuples suivants :

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

L'exécution de la *core chase* sur I sous la contrainte c_1 se déroule comme suit.

Le corps de la contrainte c est vérifié par les deux tuples de R. On va donc évaluer les deux résultat possibles *indépendamment* et ajouter leur union à I.

L'application de c au tuple $R(a, b)$ génère un nouveau tuple $S(a, \perp_1)$.

L'application de c au tuple $R(a, c)$ génère un nouveau tuple $S(a, \perp_2)$.

Notre instance I contient maintenant les tuples suivants :

$R\{(a, b), (a, c)\}$ $S\{(a, \perp_1), (a, \perp_2)\}$

On calcule maintenant le *noyau* de I. Si on retire le tuple $S(a, \perp_1)$, la contrainte c est toujours satisfaite. On peut donc le retirer de façon permanente. Si on retire maintenant le tuple $S(a, \perp_2)$, on retourne à notre instance initiale dans laquelle c

n'est pas satisfaite. On doit donc garder le tuple $S(a, \perp_2)$.
Le noyau de I contient donc les tuples suivants :
$R\{(a, b), (a, c)\}$ $S\{(a, \perp_2)\}$
On pourrait bien sûr commencer par retirer $S(a, \perp_2)$, dans quel cas le tuple gardé aurait été $S(a, \perp_1)$. Ce choix produit deux résultat équivalents et n'a pas d'incidence sur la suite de l'exécution.
Toutes les contraintes étant satisfaites, la *core chase* termine et la base de données est réparée.

4.5 Benchmark sur de vraies données ⋆⋆⋆

Si vous voulez tester votre algorithme sur de vraies données, beaucoup d'organisme proposent des jeux de données libres, comme par exemple l'institut national de la statistique et des études économiques, le gouvernement Américain ou encore Kaggle, le service de partage de données de Google. Attention, ces données ne sont généralement pas fournies avec des contraintes.
Une autre possibilité serait d'utiliser les données de test d'une autre implémentation de la *chase*, par exemple celles du projet BART, dont les résultat des tests de performances sont disponibles ici (au cas où vous auriez envie de comparer).
Dans tous les cas, avant de pouvoir executer la *chase*, il va falloir introduire des *erreurs* dans les données, c'est à dire supprimer des tuples ou des éléments de tuples.

5 Soutenance

La soutenance se déroulera par binôme, mais la notation et les questions pourront être individualisées. Vous devrez chacun maîtriser l'ensemble de ce qui est présenté, quelle que soit la façon dont vous vous êtes répartis le travail.
Vous pourrez être amenés à modifier une partie de ce que vous avez fait pour que nous puissions apprécier votre réactivité.
Nous devrions passer un minimum de temps au clavier, pouvoir enchaîner rapidement les questions, retrouver des lignes de codes très rapidement, etc.

Cela n'est possible que si vous avez pensé à tous les cas de figure, conservé et ordonné vos propres tests.

Soyez prêts : ayez de quoi faire une démonstration complète, avec des scripts préparés à l'avance qui montrent l'exécution de chacune des variantes implémentées.

Références

Abiteboul, S. (2000). *Fondements des bases de donnees*. Vuibert informatique.

Aho, A. V., Beeri, C., & Ullman, J. D. (1979). The Theory of Joins in Relational Databases. *ACM Trans. Database Syst.*, *4*(3), 297-314. https://doi.org/10.1145/320083.320091

Arenas, M., Barceló, P., Libkin, L., & Murlak, F. (2014). *Foundations of Data Exchange*. Cambridge University Press. http://www.cambridge.org/9781107016163

Cali, A., Gottlob, G., & Kifer, M. (2013). Taming the Infinite Chase : Query Answering under Expressive Relational Constraints. *J. Artif. Intell. Res.*, *48*, 115-174. https://doi.org/10.1613/jair.3873

Deutsch, A., Nash, A., & Remmel, J. B. (2008). The chase revisited. In M. Lenzerini & D. Lembo (Éd.), *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2008, June 9-11, 2008, Vancouver, BC, Canada* (p. 149-158). ACM. https://doi.org/10.1145/1376916.1376938

Fagin, R., Kolaitis, P. G., Miller, R. J., & Popa, L. (2005). Data exchange : semantics and query answering. *Theor. Comput. Sci.*, *336*(1), 89-124. https://doi.org/10.1016/j.tcs.2004.10.033

Greco, S., Molinaro, C., & Spezzano, F. (2012). *Incomplete Data and Data Dependencies in Relational Databases*. Morgan ; Claypool Publishers.

Maier, D., Mendelzon, A. O., & Sagiv, Y. (1979). Testing Implications of Data Dependencies. *ACM Trans. Database Syst.*, *4*(4), 455-469. https://doi.org/10.1145/320107.320115

Marnette, B. (2009). Generalized schema-mappings : from termination to tractability. In J. Paredaens & J. Su (Éd.), *Proceedings of the Twenty-Eigth ACM SIGMOD-*

SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2009, June 19 - July 1, 2009, Providence, Rhode Island, USA (p. 13-22). ACM. https://doi.org/10.1145/1559795.1559799

A Notions communes

Avant de pouvoir présenter les algorithmes formellement, nous allons avoir besoin de définir quelques notions de base.

Un *homomorphisme* est une fonction d'un ensemble K vers un ensemble J qui *préserve* les constantes et les structures.

Définition A.1 Homomorphisme

Soient K et J deux instances d'un même schéma. On appelle $\texttt{Consts}(K)$ (resp. $\texttt{Consts}(J)$) l'ensemble des constantes qui apparaissent dans K (resp. J) et $\texttt{NULL}(K)$ (resp. $\texttt{NULL}(J)$) l'ensemble des NULL qui apparaissent dans K (resp. J). Un *homomorphisme* $h : K \to J$ est une fonction qui va de $\texttt{Consts}(K) \cup \texttt{NULL}(K)$ dans $\texttt{Consts}(J) \cup \texttt{NULL}(J)$ telle que :

(1.) Pour chaque constante c de $\texttt{Consts}(K)$, $h(c) = c$

(2.) Pour chaque tuple $R(k_1, k_2, \ldots, k_n)$ de K, $h(R) = (h(k_1), h(k_2), \ldots, h(k_3))$ est un tuple de J.

On dit que K est *homomorphe* à J S'il existe un homomorphisme de K dans J et que K et J sont *équivalents par homomorphisme* si K est homomorphe à J et J est homomorphe à K.

Dans la définition précédente, la première condition dit simplement qu'une constante ne peut pas changer de valeur. La deuxième condition porte sur les *structures* (ou *relations* dans le contexte des bases de données) et dit, d'une part, que chaque instance de relation (c'est à dire, chaque tuple) de K doit aussi exister dans J et, d'autre part, que l'image d'un tuple est constituée des images de ses éléments. En d'autres mots, si un élément de K apparaît dans plusieurs tuples, son image dans J sera la même pour tous ces tuples.

Cette notion se généralise naturellement aux contraintes, qui peuvent aussi être vues

comme des ensembles de relations dont les éléments sont des constantes et des variables.

Exemple A.2 Greco et al. 2012, exemple 4.2

Soient $D_1 = \{R(a,b), R(a,\bot)\}$, $D_2 = \{R(a,b)\}$ et $D_3 = \{R(a,b), R(a,d)\}$ des bases de données sur le schéma $R\,[A,B]$ oú a, b, d sont des constantes et $\bot$ est un NULL.

- D_1 et D_2 sont équivalents par homomorphisme. En effet, D_1 est homomorphe à D_2 par l'homomorphisme $h_1 = \{a \leftarrow a, b \leftarrow b, \bot \leftarrow b\}$ qui envoie les constantes sur elles mêmes et le NULL sur b (remarquez que ici les deux tuples de D_1 sont fusionnés dans D_2) et D_2 est homomorphe à D_1 par l'homomorphisme $h_2 = \{a \leftarrow a, b \leftarrow b\}$.
- D_1 est homomorphe à D_3 par l'homomorphisme $h_3 = \{a \leftarrow a, b \leftarrow b, \bot \leftarrow d\}$ mais D_3 n'est pas homomorphe à D_1 car on ne peut pas préserver la constante d qui n'existe pas dans D_1. Ils ne sont donc pas équivalents par homomorphisme.
- Soit $c = R(x,y) \rightarrow R(x,z)$ une contrainte. Celle-ci est satisfaite dans D_1 car il existe un homomorphisme pour le corps $h_{corps} = \{x \leftarrow a, y \leftarrow b\}$ et un pour la tête $h_{tete} = \{x \leftarrow a, z \leftarrow \bot\}$. De même pour D_2 et D_3.

On dit que h' est une *extension* de h, si h' ajoute des nouveaux éléments à h tout en conservant les images définies par h pour les constantes.

Définition A.3 Extension d'un homomorphisme

Soient $h : X \rightarrow Y$ et $h' : X' \rightarrow Y'$ deux homomorphismes. On dit que h' est une *extension* de h si

(1.) $X \subset X'$ et $Y \subseteq Y'$

(2.) pour tout $x \in \texttt{Consts}(X)$, $h(x) = h'(x)$

Exemple A.4

Dans l'exemple A.2, $h_1 = \{a \leftarrow a, b \leftarrow b, \bot \leftarrow b\}$ et $h_3 = \{a \leftarrow a, b \leftarrow b, \bot \leftarrow d\}$ sont des extensions de $h_2 = \{a \leftarrow a, b \leftarrow b\}$.

B Standard Chase

Dans sa définition formelle, la *standard chase* est une suite d'applications d'*étapes*, où chaque étape correspond à une application de contrainte.

Définition B.1 Étape de la *standard chase* – Greco et al. 2012

Soit K une instance de base de données et r une contrainte. Une *étape* de la *standard chase* est une application de r à K définie comme suit.

(1.) Si $r = \phi(\vec{x}, \vec{y}) \rightarrow \exists\psi(\vec{x}, \vec{z})$ est une *TGD* telle qu'il existe un homomorphisme h de $\phi(\vec{x}, \vec{y})$ dans K et il n'existe pas d'extension h' de h allant de $\phi(\vec{x}, \vec{y}) \cup \psi(\vec{x}, \vec{z})$ dans K, on dit que r peut être appliquée à K par h.
Soit h' une extension de h qui assigne une variable NULL fraîche à chaque élément de $\vec{z}$ et T l'ensemble des images des atomes de ψ sous h'.
On dit que $K' = K \cup T$ est le résultat de l'application de r à K par h, noté $K \xrightarrow{r,h} K'$.

(2.) Si $r = \phi(\vec{x}) \rightarrow x_1 = x_2$ est une EDG telle qu'il existe une homomorphisme h de $\phi(\vec{x})$ dans K dans lequel $h(x_1) \neq h(x_2)$, on dit que r peut être appliquée à K par h.
Si $h(x_1)$ et $h(x_2)$ sont toutes les deux des constantes, alors le résultat de l'application de r à K par h est un échec, noté $K \xrightarrow{r,h} \perp$.
Sinon, soit K' l'extension de K où $h(x_1) = h(x_2)$. Si l'un de $h(x_1)$, $h(x_2)$ est une constante, alors l'autre est remplacé *partout* par la constante. Sinon, les deux NULL prennent la valeur de l'un des deux (choisi arbitrairement).
On dit que K' est le résultat de l'application de r à K par h, noté $K \xrightarrow{r,h} K'$.

Les conditions d'existence d'homomorphisme sont une formalisation de la notion qu'il existe un tuple qui satisfait le corps (ou la tête) d'une contrainte. L'extension de K précise les tuples ou données à ajouter.

Nous avons maintenant toutes les définitions nécessaires pour pouvoir définir la *standard chase*.

Définition B.2 Standard chase – Fagin et al. 2005

Étant donné une instance K et un ensemble de TGD et EGD Σ

- Une sequence de *chase* de K avec Σ est une séquence (finie ou infinie) d'étapes $K_i \xrightarrow{r_i,h_i} K_{i+1}$ où $i = 0, 1, \ldots$, $K_0 = K$ et $r_i \in \Sigma$.
- Une exécution finie de la *chase* de K avec Σ est une séquence de *chase* finie $K_i \xrightarrow{r_i,h_i} K_{i+1},\ 0 \leqslant i < m$ telle que soit $K_m = \bot$, dans quel cas on dit que la *chase échoue*, soit il n'existe pas de contrainte $r_m \in \Sigma$ et d'homomorphisme h_m tels que r_m peut être appliquée à K par h, dans quel cas on dit que la *chase réussit*.

C Oblivious Chase

La *oblivious chase*, définie uniquement pour les TGD, relâche les conditions sous lesquelles une contrainte peut être appliquée. Plus précisément, la non-existence d'une extension pour $\phi(\vec{x}, \vec{y}) \cup \psi(\vec{x}, \vec{z})$ dans K est supprimée.

Définition C.1 Étape de la *oblivious chase* – Cali et al. 2013

Soit K une instance de base de données et $r = \phi(\vec{x}, \vec{y}) \to \exists\psi(\vec{x}, \vec{z})$ une TGD. Une *étape* de la *standard chase* est une application de r à K définie comme suit.
S'il existe un homomorphisme h de $\phi(\vec{x}, \vec{y})$ dans K, on dit que r peut être appliquée à K par h.
Soit h' une extension de h qui assigne une variable NULL fraîche à chaque élément de $\vec{z}$ et T l'ensemble des images des atomes de ψ sous h'.
On dit que $K' = K \cup T$ est le résultat de l'application de r à K par h, noté $K \xrightarrow{r,h} K'$.

La définition de l'exécution est inchangée.

D Oblivious Skolem Chase

La *oblivious skolem chase*, définie uniquement pour les TGD, modifie les valeurs générées par une application de contrainte. En effet, quand la *standard chase* tire une variable

NULL *fraîche* (c'est à dire différentes de toutes les autres) pour chaque $z_i \in \vec{z}$, la *oblivious skolem chase* calcule la variable NULL *en fonction de* $\vec{x}$*,* z_i *et la contrainte appliquée*. Ainsi, si une TGD est appliquée plusieurs fois sur des tuples en accord sur $\vec{x}$, la *oblivious skolem chase* va produire exactement les mêmes nouveaux tuples.

Définition D.1 Étape de la *oblivious skolem chase* – Greco et al. 2012

Soit K une instance de base de données et $r = \phi(\vec{x}, \vec{y}) \rightarrow \exists\psi(\vec{x}, \vec{z})$ une TGD. Une *étape* de la *standard chase* est une application de r à K définie comme suit.
S'il existe un homomorphisme h de $\phi(\vec{x}, \vec{y})$ dans K, on dit que r peut être appliquée à K par h.
Soit h' une extension de h qui assigne une variable NULL à chaque élément de $\vec{z}$ telle que $h'(z_i) = f(r, z_i, \vec{x})$ et T l'ensemble des images des atomes de ψ sous h'. On dit que $K' = K \cup T$ est le résultat de l'application de r à K par h, noté $K \xrightarrow{r,h} K'$.

La définition de l'exécution est inchangée.

E Core Chase

Attention ! Cette variante est conceptuellement plus difficile que les précédentes ! L'algorithme donné pour calculer le noyau dans la section 4.4 devrait être suffisant pour faire l'implémentation, cependant, si vous rencontrez des problèmes (ou que vous avez très envie de comprendre), n'hésitez pas à poser des questions.

Intuitivement, le *noyau* (en anglais *core*) d'une base de données est la plus petite *solution universelle*, c'est à dire le plus petit ensemble de tuples qui satisfait toutes les contraintes. Pour définir cette notion formellement, nous avons d'abord besoin de définir les notions de *sous-instance* et *sous-instance propre*.

Définition E.1 Sous-instances et sous-instances propres – Arenas et al. 2014

Soient I et I' deux instances d'un même schéma.
On dit que I' est une *sous-instance* de I, si pour pour toute relation V, l'ensemble des tuples de V dans I', noté $V^{I'}$, est un sous-ensemble de l'ensemble des tuples de V dans I, noté V^I, c'est à dire $V^{I'} \subseteq V^I$.
Si I' est une sous-instance de I, et il existe au moins une relation V telle que $V^{I'} \subset V^I$, alors I' est une *sous-instance propre* de I.

En d'autres mots, I' est une sous-instance de I si tous les tuples de I' sont aussi dans I et I' est une sous-instance propre si I contient tous les tuples de I' ainsi qu'au moins un tuple qui n'est pas dans I'.

Définition E.2 Noyau – Arenas et al. 2014

Soient I et I' deux instances d'un même schéma telles que I' est une sous-instance de I.
On dit que I' est un *noyau* de I s'il existe un homomorphisme de I dans I' mais il n'existe pas d'homomorphisme de I dans une sous-instance propre de I'.

C'est à dire que I' est un noyau de I si c'est le plus petit ensemble pour lequel il existe un homomorphisme.

Une étape de la *core chase* se fait en deux temps. D'abord on applique toutes les contraintes possibles en *parallèle*, puis on remplace l'instance ainsi obtenue par son noyau.

Définition E.3 Étape parallèle de la *core chase* – Deutsch et al. 2008

Soit K une instance de base de données et Σ un ensemble de TGD. J est le résultat de l'application parallèle de Σ à K, noté $K \xrightarrow{\Sigma} J$ si

1. K ne satisfait pas Σ
2. $J = \bigcup \{J \,|\, c \in \Sigma,\ t \in \texttt{dom}(K),\ K \xrightarrow{c,t} J\}$, c'est à dire l'union de toutes les instances obtenues en appliquant chaque contrainte (c) à chaque tuple possible (t)

Définition E.4 Étape de la *core chase* – Deutsch et al. 2008

Soit K une instance de base de données et Σ un ensemble de TGD. J est le résultat d'une *étape de la core chase*, noté $K \xrightarrow{\Sigma\downarrow}$ si $A \xrightarrow{\Sigma} K'$ et $K = core(K')$.

La définition de l'exécution est inchangée.