Sémantique et extension du langage C2QL pour la composition de techniques protégeant la confidentialité dans le nuage

Santiago Bautista

Juillet 2017

Résumé

Des applications de tout genre manipulent des données personnelles et utilisent le cloud pour s'exécuter ou s'héberger. Différentes techniques existent pour protéger la confidentialité de ces données. Pendant ce stage nous avons étudié la sémantique et prouvé les propriétés algébriques d'un langage, nommé C2QL, permettant d'optimiser efficacement la composition de plusieurs de ces techniques, comme la fragmentation et le chiffrement.

Mots clés : privacy, cloud, semantics, proof of correctness, algebraic laws, fragmentation, C2QL language

Table des matières

1 Introduction

De plus en plus de logiciels sont développés pour être exécutés dans le cloud, et ses logiciels, de quelque sorte qu'ils soient (messagerie, gestion d'agenda personnel, commande de pizza, reconnaissance vocale, . . .) traitent des données personnelles, qu'ils doivent donc protéger.

Une des propriétés qui doit être garantie dans la protection des données personnelles est la confidentialité. Différentes techniques existent pour protéger la confidentialité des données, comme par exemple le chiffrement et la fragmentation.

Dans sa thèse de 2016 [1], Ronan Cherrueau montre que chacune de ces techniques a des avantages et des inconvénients, mais qu'*en composant les différentes techniques ensemble* on peut profiter de tous les avantages de ces techniques en éliminant la plupart des inconvénients.

Il développe donc un langage, nommé C2QL (pour *Cryptographic Compositions for Query Language*), qui permet de décrire la composition de techniques de protection des données. Grâce à C2QL, on peut non seulement raisonner sur cette composition en vu d'en prouver la correction, mais aussi passer d'un programme ne tenant pas compte de la protection à un programme composant efficacement les techniques de sécurité, en utilisant la méthode décrite ci-après et détaillée à la section 2.3 .

Le langage se présente comme un ensemble de fonctions que l'on peut composer entre elles. Parmi ces fonctions, il y a les fonctions classiques pour faire des requêtes dans des bases de données, telles que la projection et la sélection, tout comme des fonctions décrivant la protection des données, comme le chiffrement ou la fragmentation.

Un des intérêts de ce langage est que, pour décider comment protéger les données des utilisateurs, le développeur peut suivre un processus simple en trois étapes. D'abord, le développeur écrit les requêtes *en ne tenant compte* ni du fait que le programme s'exécute dans le nuage, ni des mécanismes pour le protéger. Puis, il compose sa requête avec les fonctions de protection nécessaires (qui dépendent du problème en particulier, des contraintes de confidentialité spécifiques) pour avoir une requête sécurisée. Finalement, le développeur utilise des lois de commutation entre les différentes fonctions pour optimiser sa requête sécurisée.

Par conséquent, disposer de lois qui indiquent à quelles conditions les différentes fonctions du langage commutent est très important.

Dans sa thèse, R. Cherrueau donne la plupart de ces lois. Cependant, il n'a pas eu le temps de les démontrer, ni de contempler tous les cas de figure.

C'est pourquoi, pendant ce stage, j'ai complété l'ensemble de lois fournies (section 3.2) dans la thèse de Ronan et j'ai formalisé la sémantique des différentes fonctions (section 3.1) pour ensuite démontrer la correction de ces lois (section 3.3).

Une description du contexte dans lequel s'inscrit ce stage, et du travail qui avait déjà été fait, est donnée à la section 2, et une discussion sur les aspects qui n'ont pas été traités est donnée à la section 4.

2 Contexte

2.1 De l'importance de composer les techniques de protection

De plus en plus d'applications, en particulier les applications web et les applications pour téléphone portable, cherchent à utiliser le nuage, soit pour stocker du code ou des données, soit pour faire des calculs, voir les deux à la fois.

En effet, le cloud peut offrir des services (que ce soit de l'infrastructure, des plateformes ou du logiciel) disposant d'une forte disponibilité et faciles à redimensionner.

Autrement dit, *pouvoir utiliser le cloud* est devenu, dans certains cas, un enjeu de la conception logicielle, à cause des avantages de disponibilité et redimensionnement que cela offre.

Mais ce n'est pas le seul enjeu de la conception logicielle.

Vu que la plupart de ces applications manipulent des données personnelles, garantir la *confidentialité* des données est également un enjeu de ces applications-là ; tout comme le sont les *performances* pour garantir une meilleure expérience à l'utilisateur de l'application.

C'est à ces trois enjeux-là de la conception logicielle (l'utilisation du cloud, la protection de la confidentialité et la recherche des performances) que s'intéresse R. Cherrueau dans sa thèse.

Il s'intéresse également à trois techniques particulières de protection de la confidentialité utilisées dans le développement logiciel et regarde comment elles interagissent avec les trois enjeux cités plus haut. Ces trois techniques, qu'on va décrire brièvement, sont le *chiffrement*, la *fragmentation verticale* et l'*exécution* de l'application *chez l'utilisateur*.

Le chiffrement Lorsqu'il est bien utilisé, le chiffrement permet de garantir la confidentialité des données de l'utilisateur. De plus, dans certains cas, des calculs peuvent être faits sur les données chiffrées. On appelle chiffrement homomorphe un chiffrement avec lequel on peut effectuer des calculs avec les données chiffrées. Les chiffrements homomorphes totaux, comme celui de Gentry [2] sont pour l'instant trop contraignants pour pouvoir être utilisés dans la plupart des applications, mais les chiffrements homomorphes partiels, c'est à dire les chiffrements avec lesquels on peut effectuer *certaines* opérations sur les données chiffrées, peuvent se révéler très utiles. C'est le cas des chiffrements déterministes (dont les chiffrements symétriques) qui sont des chiffrements homomorphes partiels, permettant le test d'égalité.

Dans tous les cas, le chiffrement implique un surcoût en terme de calculs, donc diminue les performances. Dans certains cas, il améliore la confidentialité et permet l'utilisation du nuage.

L'exécution côté client Si le programme était exécuté entièrement par la machine de l'utilisateur, cela serait à la fois bon pour la confidentialité (car les données ne seraient pas du tout exposées aux risques liés à l'utilisation du nuage et du réseau) et pour les performances, puisque, à moins de traiter une trop grande quantité de données dans un calcul hautement parallélisable, les performances du cloud sont moins bonnes que celles des machines des utilisateurs. Par contre, l'exécution côté client ne permet pas de profiter des avantages du cloud.

La fragmentation verticale consiste à séparer les différentes données que manipule le programme entre deux clouds n'ayant aucun rapport entre eux (à deux endroits géographiques

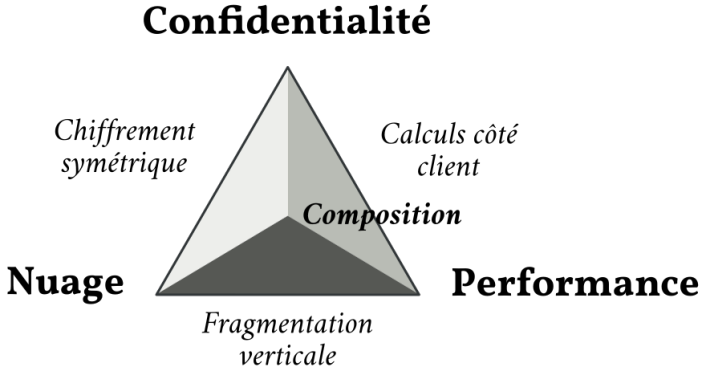

FIGURE 1 – Enjeux et techniques dans le cloud-computing

(image provenant de la thèse de Ronan Cherrueau [1])

différents, gérés par des entités différentes, etc. . .). Ceci permet de protéger celles des données personnelles qui sont constituées d'une *association* de deux données. Par exemple, dans une application stockant un ensemble de rendez-vous, l'association $(\text{date}, \text{lieu})$ doit être protégée, car une personne malveillante ayant accès à ces deux informations là à la fois pourrait suivre l'utilisateur de l'application.

L'idée de fragmentation verticale, la façon de l'utiliser, et même le fait de décrire certaines contraintes de confidentialité sous la forme d'un couple d'attributs sont empruntés à un article de Oury et Swierstra [3].

La fragmentation verticale contribue à protéger la confidentialité, mais d'une façon parfois moins forte que celles du chiffrement ou de l'exécution côté client. Par contre, chacun des fragments de données qu'elle génère peuvent être opérés séparément, en introduisant ainsi, lorsque le programme le permet, une dose de parallélisation supplémentaire qui peut améliorer les performances et permettre de tirer encore plus d'avantages du cloud.

La figure 1 résume comment ces trois techniques là interagissent avec les trois enjeux cités plus haut.

Dans sa thèse, Ronan Cherrueau montre qu'en composant ces différentes techniques on peut à la fois profiter des avantages du nuage, protéger la confidentialité et améliorer les performances d'un programme.

Pour pouvoir décrire comment s'effectue une telle composition, pour pouvoir vérifier la correction d'une telle composition et pour pouvoir raisonner dessus, Cherrueau a introduit un langage : C2QL.

2.2 Un langage pour décrire la composition : C2QL

Le langage C2QL donne une façon d'exprimer comment les techniques mentionnées ci-dessus se composent avec les fonctions classiques de l'algèbre relationnelle.

Les données manipulées sont donc représentées sous forme de tables, ou relations, c'est à dire un ensemble de lignes contenant des valeurs pour chacun(e) des différent(e)s attributs ou

colonnes considéré(e)s.

Dans l'exemple de l'application stockant des rendez-vous, cette table contiendrait autant de lignes que de rendez-vous stockés dans l'application et (par exemple) trois colonnes ou attributs : nom de l'utilisateur ayant stocké le rendez-vous, date du rendez-vous et lieu du rendez-vous.

Les opérateurs empruntés à l'algèbre relationnelle présents dans ce langage sont

- La projection, notée π qui consiste à ne considérer que certains des attributs de la table.
- La sélection, notée σ qui consiste, pour une table donnée, à ne considérer que les lignes satisfiant un certain prédicat.
- La jonction naturelle, notée $\bowtie$, qui combine les informations de deux tables en fonction des attributs qu'elles ont en commun.
- L'aggrégation et la réduction (notées respectivement group et fold) permettant, respectivement, de regrouper les lignes qui partagent les mêmes valeurs pour un certain ensemble d'attributs et d'effectuer des opérations sur les groupes de lignes ainsi obtenus.

Les opérateurs relatifs à la protection des données présents dans le langage sont

- La fragmentation verticale, notée frag, qui sépare une table en deux tables : la première contenant certains des attributs (colonnes) de la table d'origine, et la deuxième contenant le reste des attributs de la table d'origine.
- La défragmentation verticale, notée defrag, qui effectue l'opération inverse : à partir de deux tables n'ayant pas d'attributs en commun, reconstruit une seule table. Un attribut spécial, *id*, servant à identifier chaque ligne et créer lors de la fragmentation d'une table, rend la défragmentation possible.
- Le chiffrement, noté crypt qui chiffre un attribut dans une table.
- Le déchiffrement, noté decrypt qui déchiffre un attribut dans une table.

« Où est passée le calcul côté client ? » est peut-être une question que vous vous posez peut-être en ce moment. La raison pour laquelle il n'y a aucun opérateur dans C2QL qui permette d'exprimer le rapatriement des données sur la machine de l'utilisateur est que cette gestion est faite implicitement, pour garantir que les requêtes C2QL protègent toujours la confidentialité, par conception. En effet, dans le langage C2QL on suppose que les données à protéger sont toujours soit un attribut, dans lequel cas il peut être protégé par chiffrement, soit une association de deux attributs, dans lequel cas elle peut être protégée par fragmentation. Par conséquent, dans une requête, le premier déchiffrement ou la première défragmentation (ou l'absence de chiffrement ou de fragmentation dans des cas où ils auraient été nécessaires) brisent la protection des données et requièrent que les données soient rapatriées chez l'utilisateur pour cette opération et toutes les opérations postérieures.

Ainsi, on peut, en regardant une expression C2QL et en connaissant les contraintes de confidentialité à respecter, déduire à quel moment les données doivent être rapatriées chez l'utilisateur ; sans que cette opération aie besoin d'apparaître explicitement dans l'expression.

Rajouter un exemple serait probablement utile

TODO Expliquer les contraintes de sécurité avant l'aspect implicite du c.c.

Ce langage a été défini comme un Langage de Domaine Spécifique Embarqué dans Idris, qui est un langage à types dépendants. Le système de typage d'Idris et les types donnés aux

opérateurs permettent de vérifier, lorsqu'on écrit une requête en C2QL que celle-ci aura un sens au moment de l'exécution.

2.3 Utiliser C2QL : commuter les opérateurs

Dans sa thèse, Cherrueau expose un méthode pour se servir facilement et efficacement de C2QL, en trois étapes.

Première étape : écrire la version locale de l'application. Le développeur peut commencer par écrire les requêtes de son application sans tenir compte de l'utilisation du cloud ni de la protection des données. C'est ce qu'on appelle ici une version « locale » du programme ; c'est le programme tel qu'il pourrait s'exécuter dans la machine de l'utilisateur, localement. Cette version de l'application est beaucoup plus facile à écrire qu'une version utilisant efficacement les techniques de protection.

Deuxième étape : ajouter les techniques de protection nécessaires. Le langage C2QL suppose que les contraintes de confidentialité sont de deux types : soit il s'agit d'une association entre deux attributs que l'on veut protéger, dans lequel cas on choisit de mettre les deux attributs de l'association dans des fragments distincts grâce à la fragmentation verticale, soit il s'agit de la valeur d'un attribut que l'on veut protéger, dans lequel cas soit on chiffre cette valeur, soit on la décompose sur plusieurs fragments avec la technique exposée par Aggarwal (**TODO rajouter la référence**). Dans les deux cas on peut passer des versions locales des requêtes à des versions protégées en faisant apparaître à droite de la requête les fonctions de protections composées avec leurs fonctions réciproques.

Troisième étape : faire commuter les différentes fonctions. En faisant commuter les différents opérateurs qui apparaissent dans les requêtes, on peut aboutir à une requête optimisée, profitant des avantages du cloud et de bonnes performances. Pour cela, on a besoin de savoir à quelles conditions les différents opérateurs de C2QL peuvent commuter.

C'est sur cet ensemble de lois qui indiquent à quelle condition deux fonctions de C2QL peuvent commuter que je me suis intéressé pendant mon stage.

3 Contribution

Dans mon travail avec l'ensemble de lois de C2QL, j'ai commencé par poser une sémantique formelle pour les différentes fonctions du langage C2QL (section 3.1),puis j'ai complété cet ensemble de lois, en m'intéressé à toutes les combinaisons possibles de lois que l'on pouvait vouloir commuter dans le langage (section 3.2), ensuite j'ai démontré la correction sémantique de ces lois-là (section 3.3) et en ce moment je travaille sur l'automatisation de l'optimisation des requêtes, en réfléchissant à comment déterminer, parmi les commutations possibles, lesquelles il faut choisir pour optimiser une requête(section 3.4).

3.1 Établir des définitions

Les définitions des différents opérateurs présentes dans la thèse de 2016 sont données en français, ce qui facilite leur compréhension, mais rend impossible une preuve mathématique de la correction des lois les concernant.

J'ai donc commencé par poser des définitions formelles des fonctions du langage C2QL.

Pour former ces définitions, j'ai du faire des choix à plusieurs reprises. À chaque fois, j'ai utilisé les deux mêmes critères pour guider mes choix : d'une part il faut que les définitions que je choisi permettent au langage d'être le plus expressif possible, d'autre part il faut que les définitions choisies facilitent une démonstration rigoureuse de la correction des lois de commutation.

Pour un document contenant l'ensemble des définition, se référer à l'annexe A.

Dans ce rapport je vais me centrer principalement sur celles des définitions où il y a eu des choix à faire (en exposant les motivations qui ont permit d'effectuer un choix plutôt qu'un autre) et celles des définitions où il y a des différences par rapport aux définitions qui étaient suggérées par les notations de la thèse, en expliquant les motivations de ces différences.

Des tuples ou des fonctions ?

Dans son livre de 1982, Ullman parle de deux façons de définir les relations (tables) de l'algèbre relationnelle : soit comme un sous-ensemble d'un produit de domaines (donc comme un ensemble de tuples, où chaque tuple représente une ligne de la table et chaque coordonnée de chaque tuple correspond à la valeur pour un attribut donné), soit comme un ensemble de fonctions définies sur l'ensemble des attributs (chaque fonction correspond donc à une ligne de la table, et la valeur d'un attribut pour une ligne donnée est son image par cette fonction).

Ullman décide d'utiliser la première définition pour le reste de son livre, et ne détaille pas plus la deuxième définition. C'est pourtant cette deuxième définition que j'ai décidé de prendre dans ce cas ci, puisque que ce soit pour le chiffrement comme pour la fragmentation, on fait ici référence aux différentes colonnes d'une table par leur nom (toutes les colonnes ont un nom, et tous les noms des colonnes sont différents) et il est donc plus facile de raisonner, par exemple, sur la fragmentation, avec cette définition-là : en effet, la fragmentation se définit alors en thermes de simples restrictions sur certains ensembles.

Ainsi, la définition d'une relation que j'ai retenue a été la suivante (où $\mathcal{V}$ est un ensemble appelé ensemble des valeurs) :

Définition. *On appelle* relation *de schéma relationnel* Δ *tout ensemble de fonctions de* $\Delta \cup \{id\}$ *dans* $\mathcal{V}$.

Expressivité ou prudence ?

Lorsqu'on définit un langage et sa sémantique, il faut entre autres décider quelles expressions appartiennent au langage (ou « ont du sens ») et lesquelles n'y appartiennent pas.

Face à ce choix, deux attitudes sont possibles. Soit on favorise l'expressivité du langage en donnant du sens au plus d'expressions possibles, soit on est « prudent », restrictif, en ne donnant du sens qu'à un ensemble restreint d'expressions.

L'avantage de l'approche restrictive est qu'elle évite des erreurs de programmation où le développeur se retrouve à écrire quelque chose dont le sens ne correspond pas à celui qu'il désirait.

Dans l'implémentation de C2QL en tant que langage embarqué dans Idris que R. Cherrueau a développé en 2016, c'est l'approche « prudente » qui est privilégiée.

Cependant, je me suis aperçu que, du fait que l'implémentation déjà existante privilégie un langage restrictif, les définitions que je pose peuvent définir un langage plus expressif sans que cela enlève les avantages du langage restrictif.

En effet, si jamais une loi de commutation est correcte avec des définitions moins restrictives, le sens de la requête reste le même avant et après commutation et donc la loi de commutation est également sémantiquement correcte pour les définitions plus restrictives.

Des sélections portant sur plus d'un attribut à la fois.

Dans certaines des lois présentes dans la thèse, la notation suggère que les prédicats utilisés lors des filtrages ne portent que sur un seul attribut à la fois.

Ainsi, par exemple, dans la loi (7) à la page 30 de la thèse concernant la commutation entre la jointure naturelle et la sélection, on lit

$$\sigma_{p\alpha \wedge q\beta} \circ \bowtie \equiv \bowtie \circ (\sigma_{p\alpha}, \sigma_{q\beta}) \qquad \text{si } \alpha \in \Delta \text{ et } \beta \in \Delta'$$

où Δ et Δ' représentent respectivement le schéma relationnel du premier argument et le schéma relationnel du deuxième argument.

Cette notation suggère que les prédicats p et q portent chacun sur un seul attribut (respectivement α et β).

Le problème est qu'il y a certains prédicats qui ne peuvent pas ce décomposer en des prédicats portant chacun sur un seul attribut.

Par exemple, si on dispose d'une base données concernant des ornithorynques, qu'un des attributs de la base de données indique la date où l'ornithorynque a vu un kangourou pour la première fois, et qu'un autre des attributs indique la date où l'ornithorynque a pondu des oeufs pour la première fois, et que l'on ne veut s'intéresser qu'aux ornithorynques ayant vu des kangourous avant de pondre des œufs pour la première fois, alors le prédicat ("date de première vision d'un kangourou" < "date de première ponte des oeufs") ne peux pas être décomposé en deux prédicats qui porteraient l'un sur la date de la première vision d'un kangourou et l'autre sur la date de la première ponte d'œufs.

Ainsi, dans ma définition de ce qu'est une sélection et ce qu'est un prédicat, j'ai supposé que la valeur de vérité d'un prédicat dépendait de plusieurs attributs à la fois, et pour refléter cela j'utilise la notion de *domaine d'un prédicat*.

Définition. *On appelle* domaine *d'un prédicat p le plus petit ensemble D tel que :*

$$\forall (l, l') \in \mathrm{L}^2, (l|_D = l'|_D \Rightarrow p(l) = p(l'))$$

et on le note $\mathrm{dom}(p)$.

(Où L *est l'ensemble de toutes les lignes possibles.)*

Avec cette définition là, la loi de commutation entre la jonction et la sélection devient

$$\begin{aligned} \sigma_p \circ \bowtie &= \bowtie \circ (\sigma_p, \mathrm{id}) && \text{si } \mathrm{dom}(p) \subset \delta_1 \\ \sigma_p \circ \bowtie &= \bowtie \circ (\mathrm{id}, \sigma_p) && \text{si } \mathrm{dom}(p) \subset \delta_2 \end{aligned}$$

(où δ_1 et δ_2 sont les schémas relationnels de, respectivement, le premier et le deuxième argument).

Compter, ou agréger et réduire ?

Le développeur peut vouloir agir sur plusieurs lignes à la fois.

Le choix qui a été fait dans la thèse, c'est de traiter l'exemple de la fonction `count`, qui permet de regrouper, *en les comptant*, les lignes ayant les mêmes valeurs pour un certain ensemble d'attributs.

Je me suis posé la questions de savoir si les opportunités d'optimisation exposées dans la thèse par rapport à la fonction `count` et à sa façon de commuter avec les autres fonctions lui étaient propres, ou si elles étaient généralisables à toutes les fonctions d'agrégation. Plutôt que de m'intéresser à la fonction `count`, je me suis donc intéressé aux fonctions `group` et `fold`, qui permettent de regrouper, *en appliquant une fonction quelconque aux autres attributs*, les lignes ayant les mêmes valeurs pour un certain ensemble d'attributs.

En faisant cela, j'ai remarqué que les optimisations proposées pour la fonction `count` ne lui sont pas propres et se généralisent bien aux fonctions `group` et `fold`.

Remarque Pour pouvoir vraiment exprimer la fonction `count` à l'aide des fonctions `group` et `fold` il faudrait également introduire des fonctions permettant d'ajouter ou de renommer des colonnes aux/des relations manipulées. Or, comme le fait remarquer Cherrueau dans sa thèse en citant Ullman, « The nonquery aspects of a query langage are often straighforward ». L'ajout et le renommage d'attributs n'en sont pas l'exception : leur intéraction avec les autres fonctions du langage ne présente *a priori* aucune difficulté particulière, et donc ces deux opérations-là n'ont pas été traitées.

Défragmentation et jointure, un air de famille trompeur

Dans un premier temps, nous avons voulu voir la défragmentation comme un cas particulier de jointure naturelle, où l'attribut en commun entre les deux relations serait l'identifiant des lignes.

Cependant, il s'est avéré que ce n'était pas possible de traiter la défragmentation comme un cas particulier de jointure, car, du fait que la jointure peut produire plusieurs lignes là où avant il n'y en avait qu'une seule, la jointure ce doit de changer les identifiants des lignes pour des identifiants frais, ce qui est incompatible avec le comportement qu'a la défragmentation vis-à-vis des identifiants des lignes.

Ainsi, il a été nécessaire de créer des définitions séparées pour ces deux notions.

	proj	sel	frag	defrag	crypt	decrypt	join	group	fold
proj	M	M	-	M	-	C	M	A	A
sel		C	-	C	-	C	M	A	A
frag			-	C	C	C	-	-	-
defrag				-	A	C	A	C	A
crypt					A	C	-	-	-
decrypt						A	A	M	A
join							A	A	A
group								A	A
fold									A

FIGURE 2 – Bilan des lois conservées (C), modifiées (M) et ajoutées (A)

3.2 Compléter les propriétés

Pendant ce stage, j'ai également cherché à compléter l'ensemble des lois de commutation.

J'ai choisi comme critère de complétude le fait que tous les couples de fonctions du langage (tous les cas de figure possibles où l'on pourrait vouloir commuter deux fonctions) soient envisagés.

Le tableau de la figure 2 montre quelles lois étaient déjà présentes dans la thèse de 2016 ou dans l'article de 2015 de R. Cherrueau et ont été conservées (marquées avec un *C*), quelles lois nous avons modifiées (marquées avec un *M*), quelles lois nous avons rajoutées (marquées avec un *A*) et quelles lois n'ont pas à être considérées dans l'approche actuelle de C2QL (marquées avec un tiret « - »).

Pour un document contenant l'ensemble des lois après ce stage, se référer à l'annexe B.

3.3 Prouver les propriétés

Pour pouvoir utiliser les lois de commutation pour créer des programmes C2QL optimaux, il est nécessaire de s'assurer que ces lois sont correctes, c'est à dire que le résultat obtenu avant et après la commutation des opérateurs est bien le même.

Pour démontrer la correction des lois de commutation, deux approches sont possibles : soit nous essayons de les démontrer à la main, soit nous essayons de les démontrer de façon automatique avec un *prouveur* ou un *assistant de preuve* tel que `Coq`.

Nous avons opté pour la première approche, principalement pour des raisons de temps. En effet, vu qu'il s'agit d'un stage de L3 nous n'avons aucune expérience dans l'utilisation des prouveurs ou des assistants de preuve, et donc un temps non négligeable aurait été nécessaire à la prise en main de tels outils. Or, l'appropriation du sujet et du langage C2QL avaient déjà pris un temps certain, donc nous n'étions pas sûrs d'avoir le temps nécessaire. Ceci est d'autant plus vrai que nous n'avions aucun moyen de nous assurer que les méthodes de preuves automatiques (ou semi-automatiques) seraient adaptés au problème en question.

La structure des preuves est, à quelques exceptions près, toujours la même. En effet, il s'agit, de montrer que pour deux fonctions f et g du langage C2QL, les fonctions $f \circ g$ et $g \circ f$ sont, selon le cas, égales ou équivalentes.

Pour cela la méthode suivie est de considérer une relation (ou un tuple de relations) r quelconque et de montrer que $(f \circ g)(r)$ est égal (ou équivalent) à $(g \circ f)(r)$.

Mis à part certains cas où la fonction frag intervient, $(f \circ g)(r)$ et $(g \circ f)(r)$ sont tous les

deux des ensembles, et donc pour prouver leur égalité (ou leur équivalence) soit on prouve qu'ils sont l'un inclus dans l'autre et l'autre inclus dans l'un, soit on prouve que l'un est inclus dans l'autre et qu'ils ont tous deux même cardinal.

Égalité ou équivalence La plupart du temps, c'est bien d'une égalité qu'il s'agit entre les deux résultats en question. Cependant, on va ci-dessous expliquer pourquoi lorsque l'une des fonctions qu'on cherche à faire commuter est la fonction de jonction, on ne peux pas parler d'égalité proprement dite et il faut parler d'équivalence.

Cela vient de la gestion des identifiants des lignes. Pour pouvoir défragmenter des relations issues d'une fragmentation, on utilise des identifiants pour les lignes : au sein de chacune des relations manipulées, chaque ligne possède un identifiant qui (dans la relation) lui est unique. Ainsi, lors de la fragmentation, les deux morceaux d'une ligne gardent le même identifiant, ce qui permet de les faire correspondre lors de la défragmentation. Comme au sein de chaque relation chaque identifiant est unique, cette correspondance ce fait sans ambiguïté ; c'est bien le comportement désiré.

Cependant, lors d'une jointure, une ligne d'une des tables jointes peut donner lieu à plusieurs lignes dans le résultat, et il faut bien que ces lignes-là aient un identifiant (dans le cas où l'on voudrait les fragmenter par la suite). Or cet identifiant ne peut pas correspondre à celui de l'une des lignes qui lui a donné naissance, parce qu'il ne serait alors pas unique. Il ne peut pas non plus correspondre à un couple avec les deux identifiants des deux lignes lui ayant donné naissance, car une autre jonction (avec un ensemble différent d'attributs en communs, par exemple après une projection) pourrait donner lieu, ailleurs dans le système, à des lignes ayant le même identifiant que la ligne crée par la jonction, sans que ces deux lignes aient un rapport entre elles.

Ainsi, les effets propres à la jonctions nécessitent que lors d'une jonction, les lignes du résultat aient des identifiants frais, qui ne soient présents nulle part ailleurs dans le système.

Une conséquence directe de cela, est que les lois de commutation ne peuvent pas assurer l'égalité stricte des résultats lorsqu'une jonction intervient, car la jonction crée un identifiant frais pour chaque ligne résultat, et que deux identifiants frais (ceux de chaque résultat) n'ont aucune raison d'être égaux entre eux.

Dans ces cas-là il s'agit donc de prouver une égalité *aux identifiants des lignes près*. C'est ça qu'on appelle *équivalence*.

3.4 Optimiser automatiquement les requêtes

Une partie importante dans le développement des applications C2QL est le passage d'une requête sous sa forme locale (sans tenir compte du fait qu'elle s'exécute dans le cloud, ni tenir compte des mécanismes de protection utilisés) à une version optimisée.

En effet, tout l'intérêt du langage C2QL est d'obtenir un programme tirant avantage de la composition pour être particulièrement performant (au sens des trois critères définis dans la section 2) tout en facilitant au maximum le développement pour le développeur (en lui permettant de n'écrire que la version locale des requêtes).

C'est l'utilisation des lois de commutation qui permet de passer de la version locale à la version optimisée, mais cette utilisation se fait pour le moment à la main : à chaque étape, le

développeur doit regarder, dans les lois applicables, laquelle il veut appliquer.

Pour faciliter la tache au programmeur, on peut vouloir automatiser ce processus d'optimisation, ce qui pose plusieurs questions.

Choix des métriques Les lois, peuvent avoir des effets sur les trois objectifs/critères suivants :

— Utiliser le cloud le plus possible
— Utiliser le moins de temps possible
— Faire transiter le moins de données possibles à travers le réseau

Ainsi donc, si jamais on doit choisir entre deux lois de commutation différentes applicables à une requête donnée et mutuellement exclusives ; pour faire un choix objectif il faut disposer d'une manière de mesurer l'effet produit par chacune de ces lois par rapport à ces critères-ci.

Qui plus est, si les lois parmi lesquelles il faut choisir agissent sur des critères différents, il peut être nécessaire de pouvoir comparer l'importance des différents critères entre eux. Autrement dit, si l'on dispose d'une métrique pour chaque critère, il faut avoir une façon d'en déduire une métrique globale. Ceci peut nécessiter de s'intéresser aux nécessités et visées spécifiques de l'application en question.

Choix de la méthode d'exploration Les différentes suites de lois appliquées pour optimiser une requête se présentent comme un arbre de transformations possibles, qu'il faut donc explorer pour trouver la forme la plus optimale pour la requête.

Il faut donc choisir une méthode d'exploration pour explorer l'arbre en question.

Une première approche, naïve En ce moment, nous sommes en train de développer une toute première approche, complètement naïve d'un optimiseur automatique de requêtes C2QL. La méthode d'exploration choisie est l'exploration exhaustive. La métrique choisie consiste en, pour chaque lois et chaque critère, affecter un score de 1, 0 ou -1 selon le fait que la loi favorise, soit neutre, ou défavorise le critère en question ; et on affecte la même importance, le même poids, aux métriques des trois critères, par une simple somme, pour obtenir la métrique globale.

4 Travail futur

Comme nous l'avons vu dans la section 3, le but de ce stage était surtout de gagner en assurance et en « complétude » sur un aspect précis du langage C2QL : l'ensemble de lois de commutation qui permettent de passer d'optimiser une requête C2QL.

Ce but-là a été réussi, mais il reste, en ce qui concerne le langage C2QL, des ajouts et approfondissements intéressants qui peuvent être faits (tels que l'ajout d'un compilateur vers une application concrète ou la prise en compte d'autres critères de sécurité et d'autres mécanismes de sécurité), et, en ce qui concerne le travail de ce stage, l'automatisation de l'optimisation de requêtes, qui n'a été que effleurée, pourrait être traitée de façon plus approfondie.

Un compilateur concret de C2QL

Pour l'instant, C2QL dispose d'une implémentation en tant que langage embarqué dans Idris ; qui permet de vérifier certaines propriétés de « bonne formulation » des requêtes qu'on écrit en C2QL. Par exemple, lorsque, dans une requête, on déchiffre un attribut, l'implémentation actuelle vérifie que l'attribut en question était bien chiffré. Ce genre de vérifications permet d'éviter un certain nombre d'erreurs de programmation.

Pour information, Idris est un langage avec des types dépendants, ce qui veut dire qu'en Idris, le type des variables peut dépendre d'une ou plusieurs valeurs. Ainsi, on peut préciser comme type « Liste avec 3 éléments » et ce type là *dépend* de la valeur 3. Mais les valeurs dont dépend un type ne doivent pas forcément être constantes. Le type d'une variable peut en effet dépendre de la valeur d'une autre variable, ou même du résultat de l'application d'un certain type de fonction, les fonctions *totales*, à une variable. Pour faire simple, une fonction est dite *totale* en Idris, si Idris est capable de vérifier statiquement que cette fonction termine. Par exemple, le calcul de la longueur d'une liste peut être implémenté de façon totale, et donc « un couple de listes de même longueur » peut très bien être un type en Idris.

Grâce à l'utilisation de types dépendants en Idris, en typant les requêtes avec les schémas relationnels des relations utilisées, l'implémentation actuelle de C2QL peut vérifier de façon fiable des propriétés qui seraient autrement plus difficiles à obtenir.

Cependant, même si l'implémentation actuelle permet d'exprimer les requêtes à effectuer et d'en vérifier des propriétés ; elle n'offre aucun moyen d'exécuter concrètement ces requêtes sur des vraies données. Il a été envisagé par l'auteur de la thèse de réaliser un compilateur qui transforme un programme C2QL (i.e. un ensemble de requêtes C2QL) en des programmes écrits par exemple en JavaScript pour être exécutés l'un par la machine du client, les autres par les différents clouds concernés. Un travail futur pourrait consister en l'élaboration d'un tel compilateur de C2QL vers une application « concrète », exécutable sur le cloud.

De la même façon, l'auteur de la thèse avait envisagé de créer un « compilateur » de C2QL vers ProVérif pour pouvoir, avec ProVérif, vérifier que le programme C2QL écrit respecte bien les contraintes de confidentialité qui avaient été fixées.

Il existe actuellement une transformation de C2QL vers le pi-calcul ; il resterait donc à créer une transformation du pi-calcul vers ProVérif.

Un optimiseur réaliste des requêtes

Comme mentionné dans la section 3.4, l'optimiseur automatique de requêtes que nous sommes en train de développer suit une approche naïve (tantôt dans sa façon de mesurer l'effet de l'application d'une commutation, comme dans l'exploration des suites de commutations possibles) et a pour seule ambition d'être une première approche.

Un travail futur pourrait être de créer un optimiseur de requêtes plus réaliste ; ce qui nécessiterait une réflexion plus approfondie, en particulier en ce qui concerne les métriques permettant d'estimer l'effet d'une commutation donnée.

D'autres critères et mécanismes de sécurité

C2QL est un langage qui permet d'exprimer la composition de différents mécanismes de sécurité, comme on l'a vu dans la section 2.2.

Les trois mécanismes de sécurité actuellement pris en compte dans C2QL (le chiffrement, la fragmentation verticale et le fait de réaliser les calcul sur la machine du client) sont tous les trois des mécanismes permettant de protéger la *confidentialité* des données.

Or, la confidentialité n'est pas le seul critère de sécurité que l'on peut vouloir garantir. L'intégrité des données ou la disponibilité des données, par exemple, peuvent également être des critères de sécurité importants ; et pour les satisfaire, il est nécessaire de prendre en compte d'autres mécanismes de protection.

Par exemple, le tatouage des données (qui consiste à altérer légèrement les données, à en ajouter une « marque ») peut être utilisé pour protéger l'intégrité des données. En effet, si la marque en question constitue une signature électronique, alors toute attaque voulant porter atteinte à l'intégrité des données (en remplaçant les vraies données par des fausses données) serait repérable par une vérification de la marque associée à ces données là.

Ajouter d'autres mécanismes de sécurité (tels que le tatouage) pour prendre en compte d'autres critères de sécurité (tels que l'intégrité) serait une extension possible du langage C2QL.

5 Conclusion

Comme vu à la section 2, l'objectif de ce stage était de compléter et de démontrer un ensemble de lois décrivant dans quelles situations les fonctions qui constituent le langage C2QL commutent entre elles. Comme vu à la section 3, pour atteindre ces deux objectifs, il a fallu d'abord poser un cadre sémantique plus précis (*i.e.* donner des définitions plus formelles) aux fonctions du langage C2QL.

Ces trois choses (poser un cadre formel plus précis, compléter l'ensemble des propriétés et démontrer l'ensemble des propriétés en question) ont été réussies pendant le stage. Ces trois choses ayant été faites, nous avons commencé à développer une première approche, naïve, d'un optimiseur automatique des requêtes C2QL.

Comme exposé à la section 4, il reste beaucoup de choses qui pourraient être faites pour améliorer ou compléter le langage C2QL ; comme par exemple intégrer d'autres critères et mécanismes de sécurité, automatiser le processus d'optimisation des requêtes d'une façon moins naïve que celle proposée dans ce stage, ou encore créer un compilateur qui transforme les programmes C2QL en applications concrètes, capable de manipuler des données.

Bibliographie

References

[1] Ronan Cherrueau. “Un langage de composition des techniques de sécurité pour préserver la vie privée dans le nuage”. PhD thesis. École des mines de Nantes - 4 rue Alfred Kaster - Nantes - France: Université Nantes Angers le Mans, Nov. 2016.

[2] Craig Gentry and Shai Halevi. “Implementing Gentry’s Fully-Homomorphic Encryption Scheme”. In: *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*. 2011, pp. 129–148. DOI: 10.1007/978-3-642-20465-4_9. URL: http://dx.doi.org/10.1007/978-3-642-20465-4_9.

[3] Nicolas Oury and Wouter Swierstra. “The power of Pi”. In: *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*. 2008, pp. 39–50. DOI: 10.1145/1411204.1411213. URL: http://doi.acm.org/10.1145/1411204.1411213.

Annexe A : Définitions sémantiques

Le but de ce document est de donner une définition formelle des fonctions dont est composé le langage C2QL.

Préambule

Définition 1. *On appelle* nom d'attribut *toute chaîne de caractères.*

Ici, pour simplifier, on appelle chaîne de caractères tout mot sur l'alphabet

$$\Sigma = \{a, \ldots, z\} \cup \{A, \ldots, Z\} \cup \{0, \ldots, 9\}$$

Vu que le nom d'attribut « id » joue un rôle particulier, on appelle, par opposition, nom d'attribut régulier *tout nom d'attribut autre que « id ».*

Définition 2. *On appelle* schéma relationnel *tout ensemble de noms d'attributs réguliers.*

Définitions générales

Définition 3. *On appellera* valeur *tout élément d'un certain ensemble $\mathcal{V}$, que l'on suppose non-vide, infini, dénombrable, et stable par formation de n-uplets (i.e. $\forall k \in \mathbf{N}, \mathcal{V}^k \subset \mathcal{V}$).*

Définition 4. *On appelle* relation *de schéma relationnel Δ tout ensemble de fonctions de $\Delta \cup \{id\}$ dans $\mathcal{V}$.*

Chacun des éléments de la relation (chacune de ces fonctions) est appelé(e) ligne.

Pour chaque ligne l de la relation et chaque α de Δ, $l(\alpha)$ est appelé attribut *de* nom *α pour la ligne l.*

L'image de id est appelée identifiant de la ligne, *et elle est, au sein de chaque relation, unique pour chaque ligne.*

Définition 5. *On appelle* S *l'ensemble des schémas relationnels possibles. Autrement dit, on pose* $\mathrm{S} = \mathcal{P}(\Sigma^* \setminus \{id\})$.

On appelle R *l'ensemble des relations possibles,*

et on introduit la fonction sch *de* R *dans* S *qui à une relation associe son schéma relationnel.*

Projections et sélections

Définition 6. *Pour tout ensemble δ de noms d'attributs réguliers, on appelle* projection sur les attributs *δ la fonction suivante :*

$$\begin{array}{rccl} \pi_\delta : & \mathrm{R} & \to & \mathrm{R} \\ & r & \mapsto & \{l|_{(\delta \cap \mathrm{sch}(r)) \cup \{id\}} / l \in r\} \end{array}$$

Définition 7. *On appelle* L *l'ensemble de toutes les lignes possibles.*

On appelle prédicat *toute fonction de* L *dans* $\{true, false\}$.

On appelle domaine *d'un prédicat p le plus petit ensemble D tel que :*

$$\forall (l, l') \in \mathrm{L}^2, (l|_D = l'|_D \Rightarrow p(l) = p(l'))$$

et on le note $\mathrm{dom}(p)$.

Définition 8. *On appelle* sélection de prédicat p*, pour tout prédicat p, la fonction :*

$$\begin{array}{rccc} \sigma_p : & \mathrm{R} & \to & \mathrm{R} \\ & r & \mapsto & r \cap p^{-1}(\{true\}) \end{array}$$

Jointure naturelle

Définition 9. *Si r et r' sont deux relations et que l est un élément de r. On appelle* correspondants de l dans r' *l'ensemble des lignes l' telles que*

$$\forall \alpha \in \mathrm{sch}(r) \cap \mathrm{sch}(r'), l'(\alpha) = l(\alpha)$$

On note $\mathrm{cor}_{r,r'}(l)$ *l'ensemble de ces lignes-là.*

Définition 10. *Si l et l' sont deux lignes correspondantes, on appelle* concaténation de l et de l'*, notée $l.l'$ la fonction de* $\mathrm{sch}(l) \cup \mathrm{sch}(l') \cup \{id\}$ *définie par :*

$$\begin{cases} l.l'(\alpha) = l(\alpha) & \text{si } \alpha \in \mathrm{sch}(r) \setminus \mathrm{sch}(r') \\ l.l'(\alpha) = l'(\alpha) & \text{si } \alpha \in \mathrm{sch}(r') \setminus \mathrm{sch}(r) \\ l.l'(\alpha) = l(\alpha) = l'(\alpha) & \text{si } \alpha \in \mathrm{sch}(r) \cap \mathrm{sch}(r') \\ l.l'(id) = \gamma & \gamma \text{ étant un identifiant frais} \end{cases}$$

où on appelle identifiant frais *une valeur qui ne soit l'identifiant d'aucune autre ligne dans le système.*

Définition 11. *Pour r et r' deux relations, on appelle* jointure naturelle *de r et r' la relation*

$$r \bowtie r' = \{l.l' / l \in r, l' \in \mathrm{cor}_{r,r'}(l)\}$$

On utilisera aussi la notation préfixe. En effet, on vient de définir la fonction

$$\begin{array}{rccc} \bowtie : & \mathrm{R}^2 & \to & \mathrm{R} \\ & (r, r') & \mapsto & r \bowtie r' \end{array}$$

Fragmentation et défragmentation

La défragmentation est presque un cas particulier de jointure naturelle, où l'identifiant serait considéré comme un attribut en commun pour les deux tables et il serait le seul.

Définition 12. *Deux relations r et r' sont dites* unifiables *si :*

$$\mathrm{sch}(r) \cap \mathrm{sch}(r') = \emptyset$$

On remarquera que deux relations unifiables non vides sont également joignables.

On note Ru *l'ensemble des paires de relations unifiables, qui est un sous-ensemble de* R^2.

Définition 13. *Pour tout ensemble de noms d'attributs réguliers δ on appelle* fragmentation de fragment gauche *δ l'application suivante :*

$$\begin{array}{rccl} \text{frag}_\delta & \text{R} & \to & \text{Ru} \\ & r & \mapsto & (\{l|_{(\text{sch}(r)\cap\delta)\cup\{id\}}/l \in r\}, \{l_{(\text{sch}(r)\setminus\delta)\cup\{id\}}/l \in r\}) \end{array}$$

Définition 14. *On dit que deux lignes l et l' sont* unifiables *si elles partagent le même identifiant et que les relations correspondantes sont unifiables.*

On définit alors leur unification $\text{Unif}(l, l')$ *comme la fonction définie sur* $\text{sch}(l) \cup \text{sch}(l') \cup \{id\}$ *par*

$$\begin{cases} \text{Unif}(l,l')(\alpha) = l(\alpha) & si\ \alpha \in \text{sch}(l) \\ \text{Unif}(l,l')(\alpha) = l'(\alpha) & si\ \alpha \in \text{sch}(l') \\ \text{Unif}(l,l')(id) = l(id) = l'(id) & \end{cases}$$

Définition 15. *On appelle* défragmentation *la fonction de* Ru *à valeur dans* R *définie par :*

$$\begin{array}{rccl} \text{defrag}: & \text{Ru} & \to & \text{R} \\ & (r, r') & \mapsto & \{\text{Unif}(l,l')/l(id) = l'(id), l \in r, l' \in r'\} \end{array}$$

Chiffrement et déchiffrement

Vu que pour l'instant on s'intéresse uniquement aux contenus des tables pour démontrer la correction sémantique des lois de composition, on ne parlera pas pour l'instant des éventuelles clefs de chiffrement et déchiffrement.

Définition 16. *On appelle* chiffrement *tout couple $\mathtt{c}$ de fonctions* (Enc, Dec) *de $\mathcal{V}$ dans $\mathcal{V}$ vérifiant* $\text{Dec} \circ \text{Enc} = \text{id}$.

Pour toute valeur v de $\mathcal{V}$ on note $\mathtt{c}(v) = \text{Enc}(v)$ *et* $\mathtt{c}^{-1}(v) = \text{Dec}(v)$

Définition 17. *Pour une ligne l définie sur Δ, pour α un attribut, et pour $\mathtt{c}$ un chiffrement, on appelle* version de l chiffrée pour α avec le chiffrement $\mathtt{c}$ *la ligne notée $\mathtt{c}(l)_\alpha$ définie par :*

$$\begin{cases} \forall \beta \in \Delta \setminus \{\alpha\} & \mathtt{c}(l)_\alpha(\beta) = l(\beta) \\ & \mathtt{c}(l)_\alpha(\alpha) = \mathtt{c}(l(\alpha)) \quad si\ \alpha \in \Delta \end{cases}$$

De même, on définit la version de l déchiffrée pour α avec le chiffrement $\mathtt{c}$, *notée $\mathtt{c}^{-1}(l)_\alpha$, par :*

$$\begin{cases} \forall \beta \in \Delta \setminus \{\alpha\} & \mathtt{c}^{-1}(l)_\alpha(\beta) = l(\beta) \\ & \mathtt{c}^{-1}(l)_\alpha(\alpha) = \mathtt{c}^{-1}(l(\alpha)) \quad si\ \alpha \in \Delta \end{cases}$$

Définition 18. *Pour α un nom d'attribut et $\mathtt{c}$ un chiffrement, on appelle* fonction de chiffrement de α par $\mathtt{c}$ *la fonction*

$$\begin{array}{rccl} \text{crypt}_{\alpha,\mathtt{c}}: & \text{R} & \to & \text{R} \\ & r & \mapsto & \{\mathtt{c}(l)_\alpha/l \in r\} \end{array}$$

De même, on appelle fonction de déchiffrement de α par $\mathtt{c}$ *la fonction*

$$\begin{array}{rccl} \text{decrypt}_{\alpha,\mathtt{c}}: & \text{R} & \to & \text{R} \\ & r & \mapsto & \{\mathtt{c}^{-1}(l)_\alpha/l \in r\} \end{array}$$

Définition 19. *On dit d'un prédicat p et un chiffrement c sont compatibles pour l'attribut α s'il existe un autre prédicat $\mathtt{c}_\alpha \Rightarrow p$ ne dépendant que de p, $\mathtt{c}$ et du nom d'attribut α tel que*

$$\forall l \in L, p(l) = (\mathtt{c}_\alpha \Rightarrow p)(\mathtt{c}(l)_\alpha)$$

Agrégation

Définition 20. *Pour δ un ensemble de noms d'attributs réguliers, on appelle* nom de groupe pour δ *toute application n définie de δ à valeurs dans $\mathcal{V}$.*

On remarque que tout nom de groupe est une ligne.

δ est appelé domaine *du nom de groupe n, et noté* $\mathrm{dom}(n)$.

De plus, pour r une relation, on définit l'ensemble des noms de groupe de r pour δ *:*

$$r_\delta = \{l|_\delta / l \in r\}$$

Définition 21. *Pour r une relation et n un groupe, on appelle* groupe de r pour le nom n *l'ensemble des éléments de r coïncidant avec n sur* $\mathrm{sch}(r) \cap \mathrm{dom}(n)$. *On le note r_n.*

Autrement dit :

$$r_n = \{l \in r / l|_{\mathrm{sch}(r)\cap\mathrm{dom}(n)} = n|_{\mathrm{sch}(r)\cap\mathrm{dom}(n)}\}$$

De plus, on appelle identifiants du groupe r_n *l'ensemble des identifiants des lignes du groupe. On note* $\mathrm{IDs}(r_n)$ *cet ensemble.*

Autrement dit :

$$\mathrm{IDs}(r_n) = \{l(id) / l \in r_n\}$$

Définition 22. *On dira qu'une application f est* plus petite *qu'une application g si f est une restriction de g.*

On dira qu'un nom de groupe n_0 est minimal *pour une relation r donnée si c'est une plus petite application n pour laquelle le groupe de r pour n vaut r_{n_0}.*

Définition 23. *Pour r une relation, n un nom de groupe, et α un attribut de* $(\mathrm{sch}(r) \setminus \mathrm{dom}(r)) \cup \{id\}$, *on appelle* valeurs du groupe r_n pour l'attribut α *la fonction*

$$\begin{array}{rccc} r_n(\alpha): & \mathrm{IDs}(r_n) & \to & \mathcal{V} \\ & l(id) & \mapsto & l(\alpha) \end{array}$$

Remarque : Souvent, on supposera que l'ensemble des identifiants possible est totalement ordonné et on s'en servira pour considérer des fonctions définies sur un ensemble d'identifiants (par exemple les $r_n(\alpha)$ définis ci-dessus) comme des listes.

On définit la longueur de telles listes comme le cardinal de leur ensemble de départ. Par exemple, la longueur de $r_n(\alpha)$ est $|r_n(\alpha)| = |\mathrm{IDs}(r_n)|$

Définition 24. *Pour r une relation, et n un nom de groupe, on appelle* ligne de groupe de r pour n *la ligne notée* $\mathrm{lg}_{r,n}$ *définie sur* $\mathrm{sch}(r) \cup \{id\}$ *par :*

$$\begin{cases} \mathrm{lg}_{r,n}(\alpha) = n(\alpha) & \text{si } \alpha \in \mathrm{sch}(r) \cap \mathrm{dom}(n) \\ \mathrm{lg}_{r,n}(\alpha) = r_n(\alpha) & \text{si } \alpha \in (\mathrm{sch}(r) \setminus \mathrm{dom}(n)) \\ \mathrm{lg}_{r,n}(id) = \gamma & \text{où } \gamma \text{ est un identifiant frais} \end{cases}$$

Définition 25. *Pour δ un ensemble de noms d'attributs, on appelle* fonction d'agrégation pour les attributs δ *la fonction suivante :*

$$\begin{array}{rccc} \mathrm{group}_\delta: & \mathrm{R} & \to & \mathrm{R} \\ & r & \mapsto & \{\mathrm{lg}_{r,n} / n \in r_\delta\} \end{array}$$

Réduction

La plupart du temps, les agrégations sont faites pour pouvoir faire une réduction ensuite.

On suppose que les identifiants des lignes peuvent être totalement ordonnés et donc que les fonctions définies sur des ensembles d'identifiants peuvent être vues comme des listes.

Pour toute liste l on notera $\mathrm{hd}(l)$ le premier élément de la liste, et $\mathrm{tl}(l)$ le reste de la liste.

Dans les définitions qui suivent, f est une fonction de $\mathcal{V}^2$ dans $\mathcal{V}$ et z est un élément de $\mathcal{V}$.

Définition 26. *On appelle* réduction d'une liste t par la fonction f avec l'élément neutre z *la valeur* $\mathrm{red}_{f,z}(t)$ *définie par induction sur la liste par :*

$$\begin{cases} \mathrm{red}_{f,z}(\emptyset) = z \\ \mathrm{red}_{f,z}(t) = \mathrm{red}_{f,f(z,hd(t))}(\mathrm{tl}(t)) \end{cases}$$

Si une valeur v de $\mathcal{V}$ n'est pas une liste, on la considère alors comme une liste à un seul élément et on pose donc $\mathrm{red}_{f,z}(v) = f(z, v)$.

Définition 27. *Pour l une ligne définie sur δ, et α un nom d'attribut régulier, on appelle* réduction de l'attribut α dans la ligne l par la fonction f avec l'élément neutre z *la ligne* $\mathrm{red}_{\alpha,f,z,l}$ *définie sur δ par :*

$$\begin{cases} \mathrm{red}_{\alpha,f,z,l}(\alpha) = \mathrm{red}_{f,z}(l(\alpha)) & si\ \alpha \in \delta \\ \mathrm{red}_{\alpha,f,z,l}(\beta) = l(\beta) & si\ \beta \neq \alpha \end{cases}$$

Définition 28. *On appelle* fonction de réduction de l'attribut α par la fonction f avec l'élément neutre z *la fonction suivante :*

$$\begin{cases} \mathrm{fold}_{\alpha,f,z} : & \mathrm{R} & \to & \mathrm{R} \\ & r & \mapsto & \{\mathrm{red}_{\alpha,f,z,l}\ /\ l \in r\} \end{cases}$$

Opérations ensemblistes : union, différence, fragmentation horizontale

Définition 29. *On dit que deux tables r et r' sont* défragmentables horizontalement *si elles ont même schéma relationnel e leurs ensembles d'identifiants sont disjoints. Autrement dit,*

$$\begin{cases} \mathrm{sch}(r) = \mathrm{sch}(r') \\ \{l(id)/l \in r\} \cap \{l(id)/l \in r'\} = \emptyset \end{cases}$$

On appelle union *ou* défragmentation horizontale *de deux tables r et r' défragmentables horizontalement la table $r \cup r'$, aussi notée* $\mathrm{hdefrag}(r, r')$.

Définition 30. *On appelle* différence ensembliste *de deux tables r et r' ayant le même schéma relationnel la table $r \setminus r'$.*

Définition 31. *Pour p un prédicat, on appelle* fragmentation horizontale de critère p *la fonction*

$$\begin{array}{lllll} \mathrm{hfrag} : & \mathrm{R} \to & \mathrm{R}^2 & & \\ & r & \mapsto & (\{l \in r/p(l)\}, \{l \in r, \neg p(l)\}) \end{array}$$

On remarquera que les deux tables du résultat sont horizontalement défragmentables.

Annexe B : Ensemble des lois pour le langage C2QL

Pour tout chiffrement `c`, on appellera `c'` le chiffrement qui agit sur une liste en appliquant `c` à chacun des éléments de la liste.

Lois de projection

Projection et projection

$$\pi_{\delta_1} \circ \cdots \circ \pi_{\delta_n} = \pi_{\delta_1 \cap \cdots \cap \delta_n}$$

Projection et sélection

$$\pi_\delta \circ \sigma_p = \sigma_p \circ \pi_\delta \qquad \text{si } \mathrm{dom}(p) \subset \delta$$

Projection et défragmentation (verticale)

En appelant δ_1 le schéma relationnel du premier argument et δ_2 le schéma relationnel du deuxième argument, on a :

$$\pi_\delta \circ \mathrm{defrag} = \mathrm{defrag} \circ (\pi_\delta, \pi_\delta) \qquad \text{si } \delta_1 \cap \delta_2 = \emptyset$$

Projection et déchiffrement d'un attribut projeté ou non

$$\pi_\delta \circ \mathrm{decrypt}_{\alpha,\mathtt{c}} \equiv \mathrm{decrypt}_{\alpha,\mathtt{c}} \circ \pi_\delta$$

Projection et déchiffrement d'un attribut non projeté

$$\pi_\delta \circ \mathrm{decrypt}_{\alpha,\mathtt{c}} \equiv \pi_\delta \qquad \text{si } \alpha \notin \delta$$

Projection et jointure

En appelant δ_1 le schéma relationnel du premier argument et δ_2 le schéma relationnel du deuxième argument, on a :

$$\pi_\delta \circ \bowtie = \bowtie \circ (\pi_\delta, \pi_\delta) \qquad \text{si } \delta_1 \cap \delta_2 \subset \delta$$

Projection et agrégation

$$\text{group}_\delta \circ \pi_{\delta'} \equiv \pi_{\delta'} \circ \text{group}_\delta \qquad \text{si } \delta \subset \delta'$$

Projection et réduction d'un attribut projeté ou non

$$\text{fold}_{\alpha,f,z} \circ \pi_\delta \equiv \pi_\delta \circ \text{fold}_{\alpha,f,z}$$

Projection et réduction d'un attribut non projeté

$$\text{fold}_{\alpha,f,z} \circ \pi_\delta \equiv \pi_\delta \qquad \text{si } \alpha \notin \delta \cup \{id\}$$

Lois de sélection

Sélection et sélection

$$\sigma_{p_1} \circ \cdots \circ \sigma_{p_n} \equiv \sigma_{p_1 \wedge \cdots \wedge p_n}$$

Sélection et défragmentation

En appelant δ_1 le schéma relationnel du premier argument, et δ_2 le schéma relationnel du deuxième arguemnt,

$$\sigma_p \circ \text{defrag} \equiv \text{defrag} \circ (\sigma_p, \text{id}) \qquad \text{si } dom(p) \subset \delta_1$$
$$\sigma_p \circ \text{defrag} \equiv \text{defrag} \circ (\text{id}, \sigma_p) \qquad \text{si } dom(p) \subset \delta_2$$

Sélection et déchiffrement non sélectif

$$\sigma_p \circ \text{decrypt}_{\alpha,\mathsf{c}} = \text{decrypt}_{\alpha,\mathsf{c}} \circ \sigma_p \qquad \text{si } \alpha \notin \text{dom}(p)$$

Sélection et déchiffrement d'un attribut sélectif

$$\sigma_p \circ \text{decrypt}_{\alpha,\mathsf{c}} = \text{decrypt}_{\alpha,\mathsf{c}} \circ \sigma_{\mathsf{c} \Rightarrow p} \qquad \text{si } p \text{ est compatible avec } \mathsf{c}$$

Sélection et jointure

Soit δ_1 le schéma relationnel du premier argument et δ_2 le schéma relationnel du deuxième argument.

$$\sigma_p \circ \bowtie = \bowtie \circ (\sigma_p, \mathrm{id}) \qquad \text{si } \mathrm{dom}(p) \subset \delta_1$$
$$\sigma_p \circ \bowtie = \bowtie \circ (\mathrm{id}, \sigma_p) \qquad \text{si } \mathrm{dom}(p) \subset \delta_2$$

Sélection et agrégation

$$\mathrm{group}_\delta \circ \sigma_p \equiv \sigma_p \circ \mathrm{group}_\delta \qquad \text{si } \mathrm{dom}(p) \subset \delta$$

Sélection et réduction

$$\sigma_p \circ \mathrm{fold}_{\alpha,f,z} = \mathrm{fold}_{\alpha,f,z} \circ \sigma_p \qquad \text{si } \alpha \notin \mathrm{dom}(p)$$

Lois de fragmentation

Fragmentation et défragmentation

$$\mathrm{defrag} \circ \mathrm{frag}_\delta = \mathrm{id}$$

Fragmentation et chiffrement

$$\mathrm{frag}_\delta \circ \mathrm{crypt}_{\alpha,\mathtt{c}} \equiv (\mathrm{crypt}_{\alpha,\mathtt{c}}, \mathrm{id}) \circ \mathrm{frag}_\delta \qquad \text{si } \alpha \in \delta$$
$$\mathrm{frag}_\delta \circ \mathrm{crypt}_{\alpha,\mathtt{c}} \equiv (\mathrm{id}, \mathrm{crypt}_{\alpha,\mathtt{c}}) \circ \mathrm{frag}_\delta \qquad \text{si } \alpha \notin \delta$$

Fragmentation et déchiffrement

$$\mathrm{frag}_\delta \circ \mathrm{decrypt}_{\alpha,\mathtt{c}} \equiv (\mathrm{decrypt}_{\alpha,\mathtt{c}}, \mathrm{id}) \circ \mathrm{frag}_\delta \qquad \text{si } \alpha \in \delta$$
$$\mathrm{frag}_\delta \circ \mathrm{decrypt}_{\alpha,\mathtt{c}} \equiv (\mathrm{id}, \mathrm{decrypt}_{\alpha,\mathtt{c}}) \circ \mathrm{frag}_\delta \qquad \text{si } \alpha \notin \delta$$

Lois de défragmentation

Défragmentation et chiffrement

Soit δ_1 le schéma relationnel du premier argument et δ_2 le schéma relationnel du deuxième argument.

$$\begin{aligned} \text{defrag} \circ (\text{crypt}_{\alpha,\mathtt{c}}, \text{id}) &\equiv \text{crypt}_{\alpha,\mathtt{c}} \circ \text{defrag} && \text{si } \alpha \in \delta_1 \\ \text{defrag} \circ (\text{id}, \text{crypt}_{\alpha,\mathtt{c}}) &\equiv \text{crypt}_{\alpha,\mathtt{c}} \circ \text{defrag} && \text{si } \alpha \in \delta_2 \end{aligned}$$

Défragmentation et déchiffrement

Soit δ_1 le schéma relationnel du premier argument et δ_2 le schéma relationnel du deuxième argument.

$$\begin{aligned} \text{decrypt}_{\alpha,\mathtt{c}} \circ \text{defrag} &\equiv \text{defrag} \circ (\text{decrypt}_{\alpha,\mathtt{c}}, \text{id}) && \text{si } \alpha \in \delta_1 \\ \text{decrypt}_{\alpha,\mathtt{c}} \circ \text{defrag} &\equiv \text{defrag} \circ (\text{id}, \text{decrypt}_{\alpha,\mathtt{c}}) && \text{si } \alpha \in \delta_2 \end{aligned}$$

Défragmentation et jointure

On appelle, $\delta_1, \delta_2, \delta_3, \ldots$ les schémas relationnels respectifs du premier, deuxième et troisième argument.

$$\begin{aligned} \bowtie \circ (\text{defrag}, \text{id}) &\equiv \text{defrag} \circ (\text{id}, \bowtie) && \text{si } \delta_1 \cap (\delta_2 \cup \delta_3) = \emptyset \\ \bowtie \circ (\text{id}, \text{defrag}) &\equiv \text{defrag} \circ (\bowtie, \text{id}) && \text{si } \delta_3 \cap (\delta_1 \cup \delta_2) = \emptyset \end{aligned}$$

Défragmentation et agrégation

Soit δ_1 le schéma relationnel du premier argument et δ_2 le schéma relationnel du deuxième argument.

$$\begin{aligned} \text{group}_\delta \circ \text{defrag} &\equiv \text{defrag} \circ (\text{send} \circ \text{group}_\delta, \text{receiveAndGroup}) && \text{Si } \delta \subset \delta_1 \\ \text{group}_\delta \circ \text{defrag} &\equiv \text{defrag} \circ (\text{receiveAndGroup}, \text{send} \circ \text{group}_\delta) && \text{Si } \delta \subset \delta_2 \end{aligned}$$

Défragmentation et réduction

Soit δ_1 le schéma relationnel du premier argument et δ_2 le schéma relationnel du deuxième argument.

$$\begin{aligned} \text{fold}_{\alpha,f,z} \circ \text{defrag} &= \text{defrag} \circ (\text{fold}_{\alpha,f,z}, \text{id}) && \text{si } \alpha \in \delta_1 \\ \text{fold}_{\alpha,f,z} \circ \text{defrag} &= \text{defrag} \circ (\text{id}, \text{fold}_{\alpha,f,z}) && \text{si } \alpha \in \delta_2 \end{aligned}$$

Lois de chiffrement

Chiffrement et chiffrement

$$\text{crypt}_{\alpha,\mathtt{c}} \circ \text{crypt}_{\beta,\mathtt{s}} \equiv \text{crypt}_{\beta,\mathtt{s}} \circ \text{crypt}_{\alpha,\mathtt{c}} \qquad \text{si } \alpha \neq \beta$$

Chiffrement et déchiffrement

$$\text{id} \equiv \text{decrypt}_{\alpha,\mathtt{c}} \circ \text{crypt}_{\alpha,\mathtt{c}}$$

Lois de déchiffrement

Déchiffrement et déchiffrement

$$\text{decrypt}_{\alpha,\mathtt{c}} \circ \text{decrypt}_{\beta,\mathtt{s}} \equiv \text{decrypt}_{\beta,\mathtt{s}} \circ \text{decrypt}_{\alpha,\mathtt{c}} \qquad \text{si } \alpha \neq \beta$$

Déchiffrement et jointure

Soit δ_1 le schéma relationnel du premier argument et δ_2 le schéma relationnel du deuxième argument.

En appelant (P) la propriété « Soit $\mathtt{c}$ est injectif, soit $\alpha \notin \delta_1 \cap \delta_2$ »,

$$\begin{aligned} \text{decrypt}_{\alpha,\mathtt{c}} \circ \bowtie &\equiv \bowtie \circ (\text{decrypt}_{\alpha,\mathtt{c}}, \text{id}) && \text{si } \alpha \in \delta_1 \text{ et } (P) \\ \text{decrypt}_{\alpha,\mathtt{c}} \circ \bowtie &\equiv \bowtie \circ (\text{id}, \text{decrypt}_{\alpha,\mathtt{c}}) && \text{si } \alpha \in \delta_2 \text{ et } (P) \end{aligned}$$

Déchiffrement et agrégation

$$\begin{aligned} \text{group}_{\delta} \circ \text{decrypt}_{\alpha,\mathtt{c}} &\equiv \text{decrypt}_{\alpha,\mathtt{c}}, \circ \text{group}_{\delta} && \text{Si } \alpha \notin \delta \\ \text{group}_{\delta} \circ \text{decrypt}_{\alpha,\mathtt{c}} &\equiv \text{decrypt}_{\alpha,\mathtt{c}} \circ \text{group}_{\delta} && \text{Si } \alpha \in \delta \text{ et } \mathtt{c} \text{ est compatible avec l'égalité} \end{aligned}$$

Déchiffrement et réduction

Soit δ_1 le schéma relationnel du premier argument et δ_2 le schéma relationnel du deuxième argument.

$$\begin{aligned} \text{fold}_{\alpha,f,z} \circ \text{decrypt}_{\beta,\mathtt{c}} &= \text{decrypt}_{\beta,\mathtt{c}} \circ \text{fold}_{\alpha,f,z} && \text{si } \alpha \neq \beta \\ \text{fold}_{\alpha,f,z} \circ \text{decrypt}_{\alpha,\mathtt{c}} &= \text{decrypt}_{\alpha,\mathtt{c}} \circ \text{fold}_{\alpha,\mathtt{c}\Rightarrow f,\mathtt{c}\Rightarrow z} && \text{si } \mathtt{c} \text{ est compatible avec } f \end{aligned}$$

Lois de jointure

Jointure et jointure

$$\bowtie \circ(\bowtie, \mathrm{id}) \equiv \bowtie \circ(\mathrm{id}, \bowtie)$$

Jointure et agrégation

Soit δ_1 le schéma relationnel du premier argument et δ_2 le schéma relationnel du deuxième argument.

$$\mathrm{group}_\delta \circ \bowtie \equiv \bowtie \circ(\mathrm{group}_\delta, \mathrm{group}_\delta) \qquad \text{si } \delta = \delta_1 \cap \delta_2$$

Jointure et réduction

Soit δ_1 le schéma relationnel du premier argument et δ_2 le schéma relationnel du deuxième argument.

$$\begin{aligned}
\mathrm{fold}_{\alpha,f,z} \circ \bowtie &= \bowtie \circ(\mathrm{fold}_{\alpha,f,z}, \mathrm{id}) && \text{si } \alpha \in \delta_1 \setminus \delta_2 \\
\mathrm{fold}_{\alpha,f,z} \circ \bowtie &= \bowtie \circ(\mathrm{id}, \mathrm{fold}_{\alpha,f,z}) && \text{si } \alpha \in \delta_2 \setminus \delta_1 \\
\mathrm{fold}_{\alpha,f,z} \circ \bowtie &= \bowtie \circ(\mathrm{fold}_{\alpha,f,z}, \mathrm{fold}_{\alpha,f,z}) && \text{si } \mathrm{red}_{\alpha,f,z,\bullet} \text{ est injective}
\end{aligned}$$

Lois d'agrégation

Agrégation et agrégation

group ne commute pas avec lui-même

Agrégation et réduction

$$\mathrm{fold}_{\alpha,f,z} \circ \mathrm{group}_\delta = \mathrm{group}_\delta \circ \mathrm{fold}_{\alpha,f,z} \qquad \text{si } \mathrm{red}_{\alpha,f,z,\bullet} \text{ est injective et } \alpha \in \delta$$

Lois de réduction

Réduction et réduction

$$\mathrm{fold}_{\alpha,f,z} \circ \mathrm{fold}_{\beta,g,z'} = \mathrm{fold}_{\beta,g,z'} \circ \mathrm{fold}_{\alpha,f,z} \qquad \text{si } \alpha \neq \beta$$