Anonymisation de graphes RDF

Introduction

Dans le contexte actuel du Big Data, l'accès aux données devient de plus en plus facile chaque jour. En effet, les plateformes à partir desquelles il est possible de récupérer des données sont maintenant très nombreuses, une partie d'entre-elles étant notamment mises en place par les gouvernements eux-mêmes. Néanmoins, ce phénomène de démocratisation pose aujourd'hui de nouveaux problèmes dans le sens où certaines informations publiées peuvent parfois être sensibles et mener à des violations de l'intimité.

C'est pour répondre à ce genre de problématique que des acteurs ont commencé à se mobiliser afin d'apporter des méthodes d'anonymisation des données. Leur objectif est clair: masquer/supprimer les informations sensibles sans que cela ne soit fait au dépend de l'utilité des données. Ainsi, après une opération d'anonymisation, il doit toujours être possible d'exploiter le jeu de données afin d'en tirer de la connaissance, le but final consiste à trouver un compromis entre ces deux objectifs.

Query-based Anonymisation

Principes

La solution présentée dans l'article de recherche s'articule autour de deux notions: les *Privacy policies* (politiques d'intimité) et les *Utility Policies* (politiques d'utilité).

Une Privacy policy correspond à un ensemble de requêtes SPARQL sur un graphe RDF pour lesquelles on ne souhaiterait obtenir aucun résultat lorsqu'on les exécute. Elles représentent par conséquent les données sensibles que l'on voudrait anonymiser.

De l'autre côté, une Utility Policy correspond à un ensemble de requêtes pour lesquelles les réponses ne doivent pas changer après que le graphe ait été anonymisé. Concrètement, on souhaite que l'exécution d'une requête dans la politique donne le même résultat sur le graphe d'origine que sur celui obtenu après l'application d'opérations d'anonymisation. Ces politiques garantissent que l'anonymisation ne fait pas disparaître des résultats importants que l'on voudrait exploiter lors de traitements ultérieurs.

On retrouve ici la dualité entre le fait de vouloir masquer les données et la volonté de pouvoir continuer à les exploiter.

L'algorithme proposé prend en entrée ces deux politiques et cherche à produire un ensemble d'opérations d'anonymisation les satisfaisant toutes les deux en même temps. Il faut bien comprendre que l'algorithme n'effectue aucune modification sur le graphe en lui-même et se contente de renvoyer un ensemble d'opérations possibles. Il parcourt

l'intégralité des triplets contenus dans les privacy policies et vérifie leur validité, autrement dit, qu'il n'apparaissent dans aucune des utility policies.

Dans le cas où un triplet n'est pas valide, on considère qu'il existe un conflit entre l'intimité et l'utilité et le triplet est ignoré sinon, trois opérations d'anonymisation sont à envisager:

- ➢ Dans tous les cas, l'algorithme propose la suppression du triplet en question
- ➢ Si le sujet apparaît comme le sujet ou l'objet d'un autre triplet de la privacy policy, on propose de le remplacer par un noeud vide afin de briser le lien entre les deux triplets. La même opération est réalisée si le sujet apparaît parmi les variables projetées (dans la clause SELECT).
- ➢ Respectivement si l'objet du triplet est une IRI et apparaît comme le sujet ou l'objet d'un autre triplet ou qu'il fait partie des variables projetées, on propose de le remplacer par un noeud vide.

Notre implantation

Pour notre implémentation, nous avons eu recours à la librairie Apache Jena afin de pouvoir manipuler et visiter des requêtes SPARQL. Nous nous sommes également reposés sur le code publié sur GitHub par les auteurs de l'article pour voir de quelle manière certaines parties de leur algorithme avaient été implantées et comment nous pourrions l'adapter en Scala.

La classe *Anonymization*, située dans le package du même nom, contient notre adaptation des algorithmes 1 et 2 décrits dans le papier de recherche ainsi que les fonctions utilitaires *check_subject* et *check_object*.

Le package *utils* contient deux classes utilitaires. La première, *QueryUtils*, définit les méthodes permettant de parcourir une requête SPARQL avec Jena et d'en extraire les triplets. La seconde *Constants* contient la représentation sous forme de chaînes de caractères des requêtes SPARQL qui constitueront nos politiques et sur lesquelles nous lancerons notre algorithme. Ces requêtes ont été reprises directement depuis le projet GitHub des auteurs et sont celles utilisées dans les exemples de l'article.

```
# Privacy query P1
SELECT ?ad
WHERE {
  ?u  a    tcl:User.
  ?u  vcard:hasAddress  ?ad.
}
```

```
# Privacy query P2
SELECT ?u ?lat ?long
WHERE {
  ?c  a    tcl:Journey.
  ?c  tcl:user       ?u.
  ?c  geo:latitude   ?lat.
  ?c  geo:longitude  ?long.
}
```

```
# Utility query U1
SELECT ?u ?age
WHERE {
  ?u  a  tcl:User.
  ?u  foaf:age  ?age.
}
```

```
# Utility query U2
SELECT ?c ?lat ?long
WHERE {
  ?c  a    tcl:Journey.
  ?c  geo:latitude   ?lat.
  ?c  geo:longitude  ?long.
}
```

Avantages de l'approche

L'approche proposée dans l'article est particulièrement intéressante car elle analyse de manière statique des requêtes SPARQL, ainsi, la complexité de l'algorithme dépend uniquement de la taille des politiques définies par l'utilisateur et ne grandit pas avec la taille du jeu de données. On évite par conséquent le problème de devoir gérer des graphes de très grande taille quand on sait que certains d'entre eux peuvent compter jusqu'à plusieurs milliards de noeuds et relations.

Limites de l'approche

De manière générale, l'approche ne prend pas du tout en compte les informations implicites pouvant être inférées dans le contexte d'un graphe RDF. On peut citer deux problèmes en particulier:

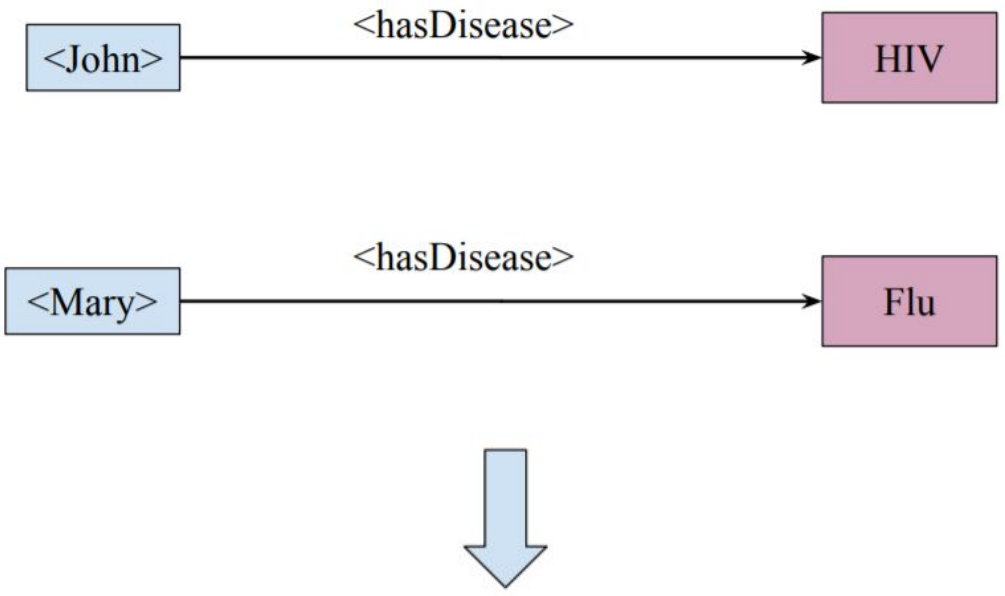
Si notre Utility Policy contient des résultats inférés par la Privacy Policy alors la première politique sera simplement ignorée. En effet, aucun conflit ne sera détecté (car les triplets sont différents) mais la suppression du triplet dans la Privacy Policy entraînera pourtant la suppression du triplet inféré dans la Utility Policy.

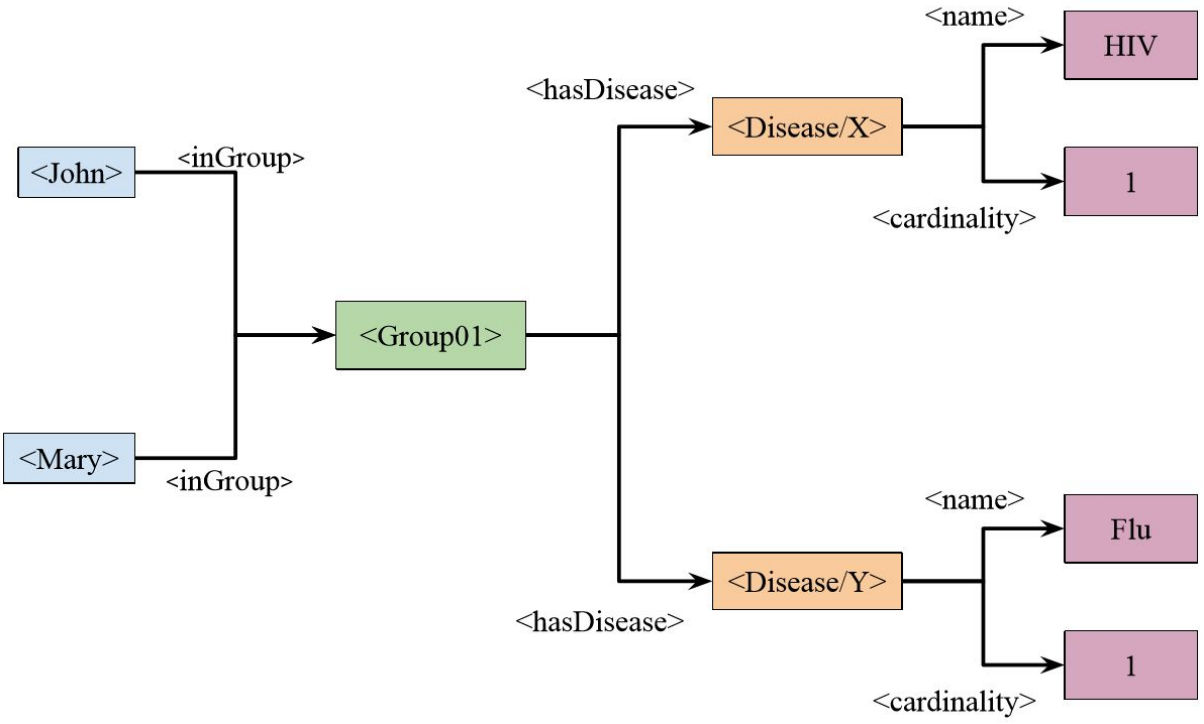
Si la Privacy Policy contient un triplet qui est le résultat d'une inférence alors aucune des opérations proposées par l'algorithme ne permettront d'anonymiser ce triplet. Étant donné que le triplet n'existe pas physiquement dans le graphe, nos requêtes UPDATE s'effectueront sur des données qui ne sont pas présentes.

Anatomisation

Principes

Dans le contexte des graphes RDF, le processus d'anatomisation consiste à briser les liens entre des ressources contenant des quasi-identifiants et des ressources correspondant à des attributs sensibles. Concrètement, cela va se traduire par l'introduction de noeuds "proxy" entre ces différentes ressources qui vont faire office de groupes et rassembler plusieurs attributs sensibles.





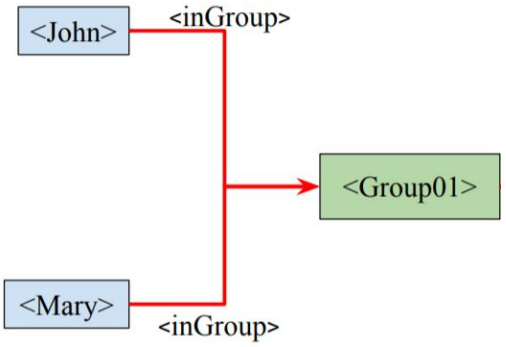
Dans le schéma ci-dessus, on prend l'exemple d'un jeu de données où des individus sont reliés à leur maladie, on introduit un nouveau noeud *Group01* qui va permettre de casser la relation sensible. À ce groupe, on relie d'autres noeuds intermédiaires *DiseaseX* et *DiseaseY* de manière à pouvoir lier un attribut sensible à son nombre d'occurrences.

En fin de compte, on sait toujours qu'une personne (ou *x* personne, tout dépend de la valeur de la cardinalité) dans le groupe souffre du Sida mais il est impossible de pouvoir l'identifier précisément.

Notre implantation de l'approche

Notre algorithme prend en entrée une chaîne de caractères correspondant à la relation sensible que l'on souhaite anonymiser et retourne la liste des opérations à effectuer pour modifier le graphe de manière à avoir nos fameux groupes (on ne fait pas de modifications réelles sur le graphe d'origine).

Le principe est simple, on commence par supprimer tous les triplets de la forme *(?s predicate ?o)* pour casser le lien et on crée un nouveau triplet (*?s inGroup GroupX*).

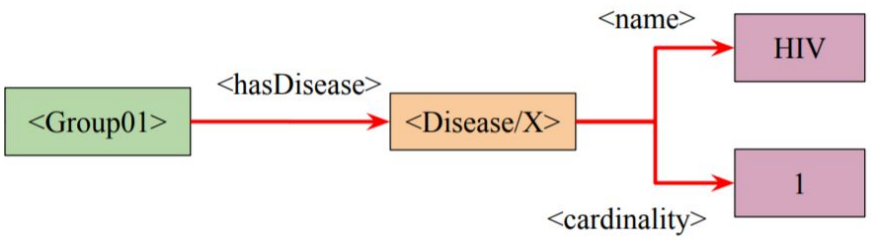


Pour conserver l'information sur la cardinalité de chaque attribut, on effectue au préalable une requête d'agrégation (on imagine ici un prédicat *hasDisease*):

```
PREFIX disease: <http://examples.ontotext.com/disease#>
SELECT ?sensitiveAttribute (COUNT(?sensitiveAttribute) as ?count)
WHERE {
    ?s ?p ?sensitiveAttribute .
    FILTER(?p = disease:hasDisease)
} GROUP BY ?sensitiveAttribute
```

	sensitiveAttribute	count
1	data:Cancer	"2"^^xsd:integer
2	data:Flu	"1"^^xsd:integer

On itère ensuite sur les résultats de cette requête afin de créer les liens entre le groupe et les attributs:



Dans notre implantation actuelle, nous ne somme capables de traiter qu'une seule relation sensible mais il serait intéressant et plutôt facile, de généraliser ce processus en prenant en entrée une liste de prédicats.

Avantages de l'approche

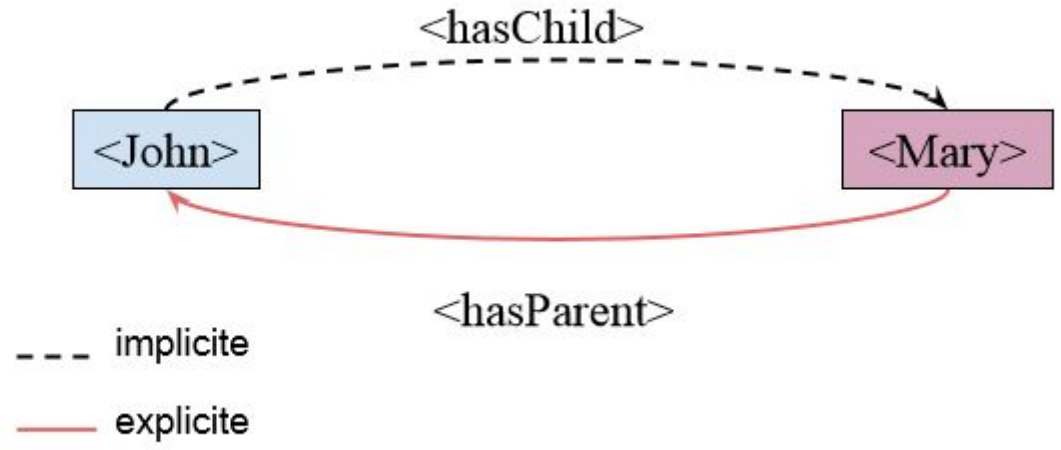
L'avantage majeur que l'on peut attribuer à l'anatomisation est le fait que cette approche conserve la connectivité du graphe, on ne supprime ou ne modifie aucune relation entre les noeuds (mis à part les relations que l'on souhaite briser bien entendu).

Par exemple, K-RDF-Neighbourhood (une des seules méthodes d'anonymisation proposées actuellement) se basent sur l'approche k-anonymity et anonymise le graphe de manière à former des clusters de K noeuds dont les voisinages sont indistinguables les uns des autres. Pour se faire, l'algorithme doit supprimer des relations, ce qui conduit forcément à une perte d'information plus importante et par conséquent à une perte d'utilité des données.

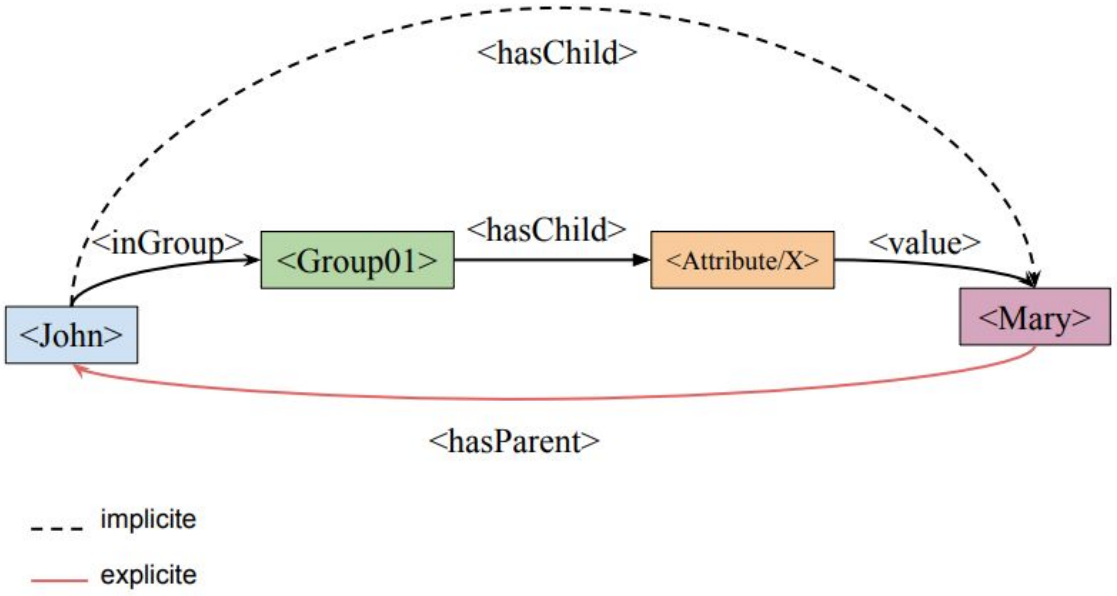
De notre côté, en laissant les relations entre les noeuds inchangés, on limite dans une certaine mesure ces pertes.

Limites identifiées

Comme pour l'anonymisation basée sur les requêtes, nous ne prenons pas en compte les problèmes liés aux inférences: on va prendre l'exemple d'une relation de filiation *hasChild* et sa relation inverse *hasParent* et imaginons que l'on souhaite anonymiser la première relation. On a le jeu de données suivant:



En effectuant notre algorithme, on obtiendrait le graphe anonymisé suivant:



Dans la situation présentée, le problème survient d'une relation inverse mais le même cas de figure pourrait se produire avec une relation transitive.

Ainsi, dans le cas où la relation sensible est inférée, il faudrait mettre en place un mécanisme capable de déterminer la relation (ou l'ensemble de relations) permettant d'obtenir ce résultat et de la supprimer. Tant que cette relation persiste, le lien implicite sera conservé et l'anonymisation restera incomplète.

Un autre problème que l'on pourrait relever est que notre approche est un peu trop générale dans le sens où on ne crée qu'un seul groupe pour chaque relation sensible.

Si l'on reprend l'exemple des maladies, on se retrouverait donc avec un noeud proxy qui pourrait regrouper toute sorte de pathologies qui n'ont rien à voir entre elles: une grippe pourrait être mise au même niveau que le Sida, ce qui n'est pas forcément pertinent tant au niveau de la nature de ces maladies que de leur degré de dangerosité.

Plus simplement, en étant trop général, on perd finalement de l'information et la capacité d'effectuer des analyses plus fines.

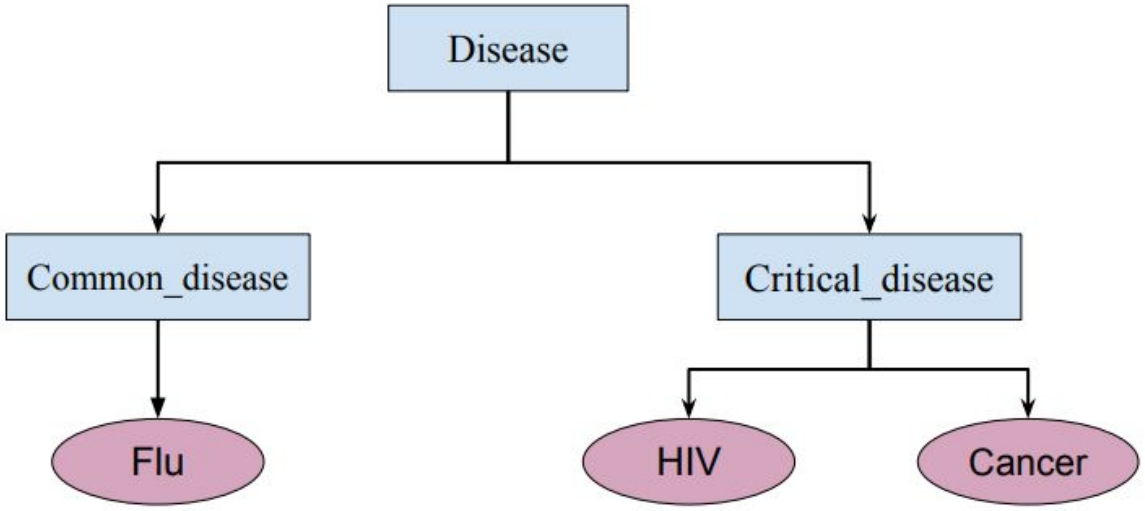
Enfin, et il s'agit d'un problème inhérent à l'anatomisation, puisque nous brisons le lien entre les quasi-identifiants et les attributs sensibles, il devient compliqué de réaliser des analyses en liant ces deux notions.

Imaginons qu'un des quasi-identifiants dans notre graphe RDF des maladies soit l'âge du patient, il n'est pas possible par exemple de chercher le nombre de personnes ayant eu la grippe en fonction de leur âge.

Améliorations possibles

Pour répondre au problème de généralisation, on pourrait penser à un moyen plus intelligent de regrouper les attributs sensibles et créer plusieurs clusters pour une seule relation. L'idée serait de piloter le clustering en se basant sur la hiérarchie des concepts dans la TBox, les attributs proches dans cette arborescence seraient regroupés en priorité.

Prenons une hiérarchie de concepts très simple:



On peut maintenant regrouper les maladies critiques ensemble dans un même groupe et ce groupe sera lui-même une instance de *Critical_disease*. Avec cette approche, on atténue donc la perte d'utilité des données en évitant de tout écraser dans un même ensemble, il est beaucoup plus simple pour un utilisateur de pouvoir raisonner sur des maladies similaires.

Pour regrouper deux attributs, il nous faut un moyen de trouver le concept commun le plus spécifique entre ces derniers: cela correspond au problème du *least common subsumer*. Nous avons réussi à mettre au point une requête SPARQL dans ce but:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
SELECT ?super
WHERE {
    concept1 +  rdfs:subClassOf ?super .
    concept2 +  rdfs:subClassOf ?super2 .
    FILTER NOT EXISTS {
        ?moreSpecificClass rdfs:subClassOf ?super .
        concept1 + " rdfs:subClassOf ?moreSpecificClass .
        concept2 + " rdfs:subClassOf ?moreSpecificClass .
    }
    FILTER(?super != owl:Thing)" +
};
```

L'idée aurait ensuite été de réaliser un clustering hiérarchique qui est un algorithme itératif de machine learning: le principe consiste, pour chaque étape, à trouver les deux clusters les plus proches et à les regrouper.

En adaptant cette méthode à nos besoins, on aurait itérer sur chacun des attributs sensibles (Flu, HIV, Cancer, etc…), trouver l'attribut (ou le cluster) le plus proche et effectuer la fusion. Le processus s'achève lorsque tous les attributs ont été traités

Certaines difficultés doivent néanmoins être prises en compte:

- ➢ Le calcul de la similarité entre deux concepts: certains articles de recherche traitent de ce problème et calcule ce qu'on peut appeler la *similarité taxonomique* notamment en comparant les ancêtres communs des deux concepts.
- ➢ Comment faire si un attribut ne partage aucun concept commun avec les autres ? On pourrait penser à créer des concepts artificiels dans ces cas de figure.
- ➢ Il est possible de se retrouver avec des clusters trop peu divers (seulement deux ou trois attributs sensibles différents), c'est un problème que l'on peut également retrouver avec l'approche k-anonymity. Une solution possible serait de construire nos clusters puis, une fois tous les attributs traités, de vérifier la présence de cluster invalide et de les fusionner avec un autre cluster (toujours en se basant sur un calcul de similarité). L'algorithme s'arrêterait au moment où tous les clusters seraient valides.