Qualité du schéma relationnel

Lorsqu'une application modifie les données d'une base de données relationnelle, il faut s'assurer que le schéma relationnel de cette base de données est "sain", c'est-à-dire qu'il ne contient pas de redondance ou d'incohérence. En effet un schéma qui contiendrait de la redondance entrainerait des anomalies de modification, d'insertion et de suppression qui rendraient l'application ingérable.

Par exemple, si on a le schéma relationnel suivant, qui n'est pas normalisé

Fournisseurs (idFournisseur, nomFournisseur, codeVille, nombreHabitantsVille)

F1	Zeblouse	MTP	350 000
F2	Zétofrais	NIM	3 000
F3	Bricot	MTP	350 000
F4	Gator	MTP	350 000

Ici, le schéma contient de la redondance. On doit indiquer plusieurs fois qu'il y a 350 000 habitants à Montpellier. A cause de cela, il pourrait même y avoir de l'incohérence. On pourrait avoir un fournisseur qui habite à Montpellier où il y a 400 000 habitants. Si ensuite on doit faire une requête pour savoir combien il y a d'habitants à Montpellier on risque d'avoir plusieurs résultats.

Mais le plus grave est qu'à cause de cette redondance nous allons être confrontés à trois types d'anomalies lorsque nous allons vouloir mettre à jour les données qui se trouvent dans la table :

- Anomalies de modification : si un recensement nous indique qu'il y a maintenant 400 000 habitants à Montpellier, il faut modifier plusieurs lignes (si on ne modifie pas toutes les lignes concernées on va encore avoir de l'incohérence).
- Anomalies de suppression : si le fournisseur F2 vient à disparaitre, on va supprimer sa ligne dans la table. Mais comme par chance il s'agit du seul fournisseur nîmois, en le supprimant on va aussi perdre le nombre d'habitants de la ville de Nîmes.
- Anomalies d'insertion : si on nous indique qu'il y a 70 000 habitants à Béziers, mais qu'on n'a pas encore de fournisseur de cette ville (on en aura probablement demain), il n'est pas possible d'enregistrer cette information.

Pour ne pas avoir toutes ces anomalies de mises à jour, il faut travailler sur un schéma "sain", sans redondance : un schéma normalisé. Pour cela on décompose notre table Fournisseurs en deux tables distinctes :

Fournisseurs (idFournisseur, nomFournisseur, codeVille#)
Villes (codeVille, nombreHabitantsVille)

Ici, il n'y a plus de redondance. Et il n'y a donc plus d'anomalies de mise à jour. Si on modifie la population de Montpellier on ne change qu'une seule ligne. Si on supprime le fournisseur F2, on garde le nombre d'habitants de Nîmes, et on peut rajouter la ville de Béziers (et son nombre d'habitants) même si on n'a pas encore de fournisseur à Béziers.

La contrepartie ce cette normalisation est qu'on a maintenant plusieurs tables et que les requêtes vont devoir comporter des jointures. Ces requêtes seront donc plus complexes à écrire (mais si vous êtes des Ninjas SQL cela ne devrait pas vous déranger) et plus longues à s'exécuter (mais si vous êtes des Ninjas SQL vous saurez où placer vos index).

Dans ce cours nous allons voir comment détecter rapidement un schéma qui n'est pas normalisé (même s'il s'agit d'un schéma complexe) puis comment le normaliser pour éradiquer toute redondance grâce à des décompositions successives. Nous verrons aussi comment faire pour s'assurer que ces décompositions ne perdent pas des informations (des données ou des dépendances fonctionnelles). Enfin nous terminerons par voir dans quels cas on peut se permettre d'avoir un schéma non normalisé.

Toute la théorie de la normalisation s'appuie sur le concept de Dépendance Fonctionnelle (qui a beaucoup de similitudes avec ce que vous avez vu en mathématiques dans les graphes) que nous allons commencer par définir.

1 Dépendances Fonctionnelles

La Dépendance Fonctionnelle (ou DF) est le concept principal sur lequel est basé la normalisation que nous détaillerons lors du prochain paragraphe. Après avoir vu les différents types de DF, nous énumérerons les axiomes d'Armstrong qui permettent de manipuler ces DF. Ensuite nous analyserons comment il faut s'y prendre pour calculer la fermeture transitive d'un ensemble de DF. Et nous verrons enfin que les DF peuvent être utiles pour trouver la clé d'une relation ou bien pour placer des propriétés dans un modèle e/a.

1.1 Les différents types de Dépendances Fonctionnelles

1.1.1 Définition de la dépendance fonctionnelle (DF)

Soit A et B qui sont des attributs ou bien des ensembles d'attributs, on dit qu'il y a une dépendance fonctionnelle de A vers B (ou bien que B dépend fonctionnellement de A) que l'on note A→B si et seulement si pour une valeur de A correspond **au plus** une valeur de B (0 ou 1). On dit que A est la source de la dépendance fonctionnelle et B en est le but.

Par exemple, `numEtudiant→nomEtudiant` est une DF car pour un numéro d'étudiant donné on a au plus qu'un seul nom d'étudiant (puisque deux étudiants ne peuvent pas avoir le même numéro). Par contre la réciproque n'est pas vraie. `nomEtudiant→numEtudiant` n'est pas une DF car si plusieurs étudiants ont le même nom, pour un nom d'étudiant donné il peut y avoir plusieurs numéros d'étudiants.

Il est à noter que la validité d'une dépendance fonctionnelle **dépend du contexte**.

Par exemple la DF `numEtudiant→prénomEtudiant` sera vraie dans une application où on ne stockera qu'un seul prénom par étudiant, mais elle sera fausse dans une autre application où on voudra répertorier tous les prénoms des étudiants.

Comme nous le verrons plus tard, lorsque plusieurs dépendances fonctionnelles ont une même source, il est possible de les regrouper en une seule expression où le but est composé de plusieurs attributs (union). Et inversement, une dépendance fonctionnelle qui a un **but** composé de **plusieurs attributs peut éventuellement être décomposée** en plusieurs dépendances fonctionnelles distinctes (décomposition) avec la même source.

Par exemple, si on a `A→B` et `A→C`, il est possible de regrouper ces deux DF en écrivant `A→BC`. Et inversement, la DF `A→BC` peut être décomposée en `A→B` et `A→C`.

Lorsque c'est **la source** de la dépendance fonctionnelle qui est composée **de plusieurs attributs**, la dépendance fonctionnelle **n'est pas décomposable**. Cela signifie que le but de la dépendance fonctionnelle dépend de tous les attributs sources de la dépendance fonctionnelle.

Par exemple si on a `numEtudiant, numRessource → moyenne`, cela signifie que pour un numéro d'étudiant et un numéro de ressource on a au plus une moyenne. La moyenne dépend des deux attributs sources. Il n'est pas possible de décomposer cette DF car un numéro étudiant seul ne donne pas la moyenne de l'étudiant à la ressource ; et un numéro ressource seul ne donne pas non plus la moyenne de l'étudiant à la ressource.

Pour qu'une dépendance fonctionnelle soit juste, **il faut qu'elle soit toujours vraie**. On ne peut pas dire qu'une DF est correcte si elle ne se vérifie que dans 99% des cas.

Par exemple la DF `nomProf, adresseProf, telProf, numRessource → numProf` n'est pas acceptable si on peut avoir deux enseignants (avec des numéros différents) qui ont le même nom, prénom, téléphone et qui enseignent la même ressource. En effet même si cela est rare, cela peut tout de même se produire s'il s'agit d'enseignants Ninja.

1.1.2 Dépendance fonctionnelle élémentaire (DFE)

A→B est une dépendance fonctionnelle élémentaire si B n'est pas en dépendance fonctionnelle avec une partie de A (et si A→B est une dépendance fonctionnelle).

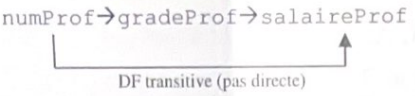
Par exemple, la DF `numEtudiant, nomEtudiant → prénomEtudiant` n'est pas élémentaire (on dit qu'il s'agit d'une DF partielle) car `numetudiant → prénomEtudiant`

Il est à noter que toutes les dépendances fonctionnelles correctes qui ont une **source composée d'un seul attribut sont forcément élémentaires**.

1.1.3 Dépendance fonctionnelle directe (DFD)

A→B est une dépendance fonctionnelle directe si elle ne peut pas être retrouvée par transitivité (et si A→B est une dépendance fonctionnelle).

Par exemple, si on a les DF `numProf→gradeProf` et `gradeProf→salaireProf`, alors la DF `numProf→salaireProf` n'est pas directe (on dit qu'elle est transitive)



1.2 Les axiomes d'Armstrong

Armstrong a proposé six axiomes mathématiques (que l'on ne démontrera pas) qui permettent de manipuler les dépendances fonctionnelles.

1.2.1 Réflexivité :

`∀ X, X→X`

`∀ YCX, X→Y`

Par exemple, `numProf→numProf` ou bien numProf, nomProf → numProf

1.2.2 Augmentation :

`X→Y ⇒ XZ→YZ`

Par exemple, si on a `numPof→nomProf`, alors on peut en déduire qu'en rajoutant adresse prof des deux côtés on a `numProf, adresseProf → nomProf, adresseProf`

1.2.3 Transitivité :

`X→Y et Y→Z ⇒ X → Z`

Par exemple, si on a `numEtudiant→classe` et `classe→numProfPrincipal` on peut en déduire que `numEtudiant→numProfPrincipal`

1.2.4 Union :

`X→Y ; X→Z ⇒ X→YZ`

Par exemple, si on a `numProf→nomProf` et `numProf→adresseProf` on peut en déduire que `numProf → nomProf, adresseProf`

1.2.5 Décomposition :

`X→YZ ⇒ X→Y et X→Z`

Par exemple, si on a `numProf → nomProf, adresseProf`, on peut décomposer cela en écrivant `numProf→nomProf` et `numProf→adresseProf`

1.2.6 Pseudo transitivité :

`X→Y ; WY→Z ⇒ XW→Z`

Par exemple, si on a `numEtudiant→groupeTD` et `groupeTD, numRessource → numProf` alors on peut en déduire que `numEtudiant, numRessource → numProf`

1.3 Fermeture transitive d'un ensemble de DF

Soit F un ensemble de dépendances fonctionnelles, on appelle fermeture transitive de F (notée F^+), l'ensemble des DF élémentaires que l'on obtient après avoir enrichi F avec des DF obtenues par transitivité.

En d'autre termes, on rajoute à F toutes les DF qu'il est possible de trouver par transitivité ou pseudo-transitivité, puis à la fin on retire toutes les DF qui ne sont pas élémentaires dans F^+.

Par exemple, si on a F = {`A→B ; AB→C`}

Grâce à la transitivité on peut rajouter à F la DF `A→C` (en effet, on a `A→B` donc par augmentation `AA→AB` donc `A→AB` ; mais comme on a aussi `AB→C` par transitivité on en déduit que `A→C`). On peut donc écrire F^+ = {`A→B ; AB→C ; A→C`}

Comme il n'y a plus de DF à rajouter par transitivité, il ne reste plus qu'à retirer les DF qui ne sont pas élémentaires (attention il ne faut pas retirer les DF qui sont transitives sinon on retournerait sur F).

Maintenant que l'on a dans F^+ `A→C`, la DF `AB→C` n'est plus élémentaire. On peut donc la retirer de F^+. Donc F^+ = {`A→B ; A→C`}

On dit que deux ensembles de DF F1 et F2 sont équivalents s'ils ont la même fermeture transitive. C'est-à-dire que F1 ≡ F2 si et seulement si $F1^+ = F2^+$.

1.4 Trouver la clé d'une relation grâce aux DF

Une clé est un ensemble **minimal** d'attributs qui permet de retrouver grâce à des DF (réflexives, directes ou transitives) la totalité des attributs de la relation.
C'est à dire que soit la relation R(A1, A2, …, An), avec X^+ la liste des attributs qui dépendent de X par transitivité, X est une clé de R si :

- X^+ = A1, A2, …, An
- $\nexists$ Y⊂X tel que Y^+ = A1, A2, …, An (c'est à dire que X est minimal)

Par exemple, avec la relation R(A,B,C,D) et l'ensemble de DF, F = {A→B ; B→C ; C→D}
On essaye de voir si B peut être une clé. On calcule donc B^+. En partant de B on a B par réflexivité, C grâce à la DF directe B→C et D par transitivité car si on a C on a aussi D car C→D. Donc B^+= BCD. Mais comme il manque A, alors B n'est pas une clé.
Par contre si on prend A^+ = ABCD on a tous les attributs de la relation R. Donc A est une clé (elle est forcément minimale car elle est composée d'un seul attribut).
Par contre AB n'est pas une clé. Car bien que AB^+ = ABCD, le couple AB n'est pas minimal car A seul est clé.

Il est à noter que **minimal ne veut pas dire le plus court possible**. Cela veut dire que tous les attributs de la clé sont indispensables.

Par exemple, si on a la relation R(W, X, Y, Z) et qu'on a trouvé la clé W^+ = WXYZ, on peut éventuellement avoir une autre clé XY^+ = WXYZ même si elle est plus longue que la clé W, à condition qu'elle soit minimale c'est-à-dire que X seul ne soit pas clé et que Y seul ne soit pas clé.

1.5 Utiliser les DF pour modéliser un modèle e/a ou un diagramme de classes

Les dépendances fonctionnelles peuvent aussi être utilisées pour placer les différentes propriétés dans un modèle e/a ou bien dans un diagramme de classes d'analyse.

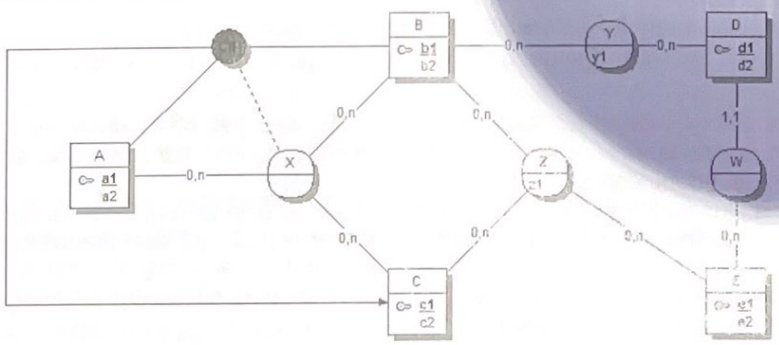
Ainsi, sur un modèle e/a qui doit forcément être normalisé (s'il est bien construit), une propriété doit se trouver à l'intérieur d'une entité que si elle est en DFED avec l'identifiant de l'entité. Et une propriété doit se trouver à l'intérieur d'une association (une association donc porteuse) que si elle est en DFED avec tous les identifiants des entités qui relient l'association.

Les dépendances fonctionnelles peuvent aussi permettre de trouver les CIF sur une association ternaire ou n-aire. En effet, une CIF est juste que s'il y a une DFED entre l'ensemble des identifiants des entités sources de la CIF et l'identifiant de l'entité cible de la CIF.

Il est aussi possible d'utiliser les dépendances fonctionnelles pour détecter les associations ternaires ou n-aires potentiellement fausses. En effet, pour qu'une association n-aire soit justifiée, il ne faut pas qu'il y ait de DF entre les identifiants de deux entités reliées par l'association (du moins de DF qui concernent le sujet de l'association). Sinon, l'association n-aire est décomposable.

Cette CIF implique que sur la ternaire X on doit avoir la DF suivante : a1,b1 → c1.

Si y1 est dans l'association Y cela implique que b1,d1 → y1.
Si ce n'est pas le cas, et qu'un couple b1,d1 peut avoir plusieurs y1, il faut alors remplacer l'association Y par une entité coordinatrice.



Si a2 est dans l'entité A, cela signifie qu'on a la DF a1→a2

L'association W signifie qu'on doit avoir la DF d1→e1

La ternaire X implique que dans le contexte de l'association X, on ne doit pas avoir les DF suivantes : a1→b1 ; a1→c1 ; b1→a1 ; b1→c1 ; c1→a1 ; c1→b1.

La ternaire Z implique que dans le contexte de l'association Z, on ne doit pas avoir les DF suivantes : b1→c1 ; b1→e1 ; c1→b1 ; c1→e1 ; e1→b1 ; e1→c1.
De plus, comme la ternaire est porteuse on doit avoir la DF b1,c1,e1 → z1