Base de données de la rédaction d'un journal

Théo Delalande-Delarbre Thomas Minier
Benjamin Sientzoff

22 mars 2015

Université de Nantes - Licence 3 Informatique
X6I0050 Base de données 2

Table des matières

Introduction

Dans le cadre du cours *Base de données 2*, il nous a été demandé de réaliser, par groupe de trois, une base de données réaliste et réalisable en entreprise. Nous avons choisi de modéliser la rédaction d'un journal. Ensuite tout au long du module, nous devons ajouter dans notre base de données des applications des concepts vu en cours.

Dans une première partie, nous détaillerons l'organisation de notre *trinôme* pour la réalisation de ce projet. Ensuite, nous présenterons la conception de notre base de données en partant d'une *Big table*. Nous déterminerons les différentes dépendances fonctionnelles, puis nous appliquerons deux algorithmes de normalisation sur notre table : l'algorithme de Bernstein et l'algorithme de décomposition, tous deux vus en cours.

Nous présenterons également les différentes vues que nous avons mises en place pour visualiser notre base. Nous présenterons également les différents *triggers* créés ainsi que les fonctions stockées pour interagir avec la base de données.
Enfin, nous discuterons des différents atouts et modifications possibles de notre base de données.

semaine	Tâche	Responsable
4	Big table	groupe complet
5	Rédaction rapport	groupe complet
5	Algo de décomposition	Théo DD
5	Algo de Bernstein	Thomas M
5	Planning	Benjamin S
6	Rendu rapport partiel	groupe complet
6	Implémentation SQL des tables	groupe complet
6	Insertion données dans BD	groupe complet
7	Définissions des requêtes stockées	Théo DD
8	Procédures PL/SQL	Thomas M, Benjamin S
8	Mise à jour du planning	Benjamin S
9	Correction du rapport	Thomas M
10	Correction rôle pour droit d'accès	Benjamin S
11	Procédures PL/SQL	Théo DD, Thomas M
12	Démonstration	groupe complet
13	Rendu rapport final	groupe complet

FIGURE 1 – Répartition des tâches dans le groupe

1 Base du projet

1.1 Répartition des tâches

Le projet étant réalisé en *trinôme*, il est alors indispensable de répartir au mieux les tâches. Le tableau à la figure 1 présente la répartition des tâches dans le groupe ainsi que les dates prévisionnelles de leur réalisation.

1.2 Définition de la *big table*

Notre base de données va représenter la gestion des numéros d'un journal. Chaque **édition** a un numéro, une Une, un prix, une date de parution, un type, un rédacteur en chef. Chaque **personne** a un numéro d'identification, un nom, un prénom et un numéro de téléphone. Chaque personne a un métier (photographe, rédacteur, etc.) ce qui définit un salaire de base. Un **contenu** a un titre, un type et un auteur. Dans la base de données on stockera l'emplacement des contenus sous forme d'URL. Un **article** a une date de rédaction, un titre et un résumé. Un **article** est composé d'un ensemble de **contenus** et une **édition** est composée d'un ensemble d'articles.
Le schéma à la figure 3 présente toutes les dépendances fonctionnelles de notre base de données.

2 Conception de la base de données

2.1 Pourquoi décomposer la table ?

Il y a au début une unique table contenant une vingtaine d'attributs. Pour illustrer la redondance et les dépendances fonctionnelles, cet exemple de *tuples*

idArt	titreArt	idJrnal	typeJrnal	idCont	nomCont	nomPers	nomMetier
1	Les pieuvres	1	Hebdo	1	Photo	Bertrand	Photographe
1	Les pieuvres	1	Hebdo	2	Texte	Nadine	Redacteur
2	Manger du sel	1	Hebdo	3	Schéma	Yves	Infographie
2	Manger du sel	1	Hebdo	4	Texte	Nadine	Redacteur
3	Critique film	1	Hebdo	5	Texte	Manon	Critique
4	Des poissons	3	HS	6	Texte	Nadine	Redacteur
2	Manger du sel	3	HS	1	Schéma	Yves	Infographie
2	Manger du sel	3	HS	2	Texte	Nadine	Redacteur

FIGURE 2 – Exemple de *tuples* dans la *big table* (vue partielle)

ne sera centré que sur quelques attributs pour des raisons de lisibilité. La figure 2.1 présente un aperçu de notre *big table*.

On remarque qu'il a une répétition des données relatives au journal ou au contenu : un contenu a toujours les mêmes ID, nom et auteur associé. Les journaux d'un même ID ont leur type en commun, de même pour les identifiants des articles et leurs titres. Enfin, les personnes ont toujours le même métier. On peut avoir une perte d'information du fait de cette configuration car si l'on supprime par exemple tous les contenus créés par une personne, on perd les données qui lui sont associées, à savoir son nom, son prénom, son numéro, etc.

De ce fait, en prenant en compte tous les attributs nous avons dégagé un ensemble de dépendances fonctionnelles.

Dépendances fonctionnelles Les dépendances fonctionnelles de la figure 3 peuvent être retranscrites de la manière suivante.

(1) idJournal → prix, nbTirage, idUne, idType, redacChef, dateParu
(2) idType → nomType
(3) idUne → idJournal
(4) idArticle → dateRedac, titreArt, resumeArt
(5) idContenu → URL, titreCont, typeCont
(6) idPers → nomPers, prenom, numTel, idMetier
(7) idMetier → salaireBase, nomMetier
(8) idArticle, idPers, dateParu → idJournal

Clés De ce schéma, on peut en déduire trois clés $\{idJournal, idArticle, idContenu\}$, $\{idUne, idArticle, idContenu\}$ et $\{dateParu, idContenu, idArticle\}$. Ces ensembles d'attributs sont des clés, car ils déterminent l'ensemble des attributs.

2.2 Algorithme de Bernstein

Une fois nos dépendances fonctionnelles établies, nous commençons par appliquer l'algorithme de Bernstein pour décomposer notre grande table. Tout d'abord, nous déterminons sa couverture minimale. Pour cela, on utilise la clé {idJournal, idArticle, idContenu}.

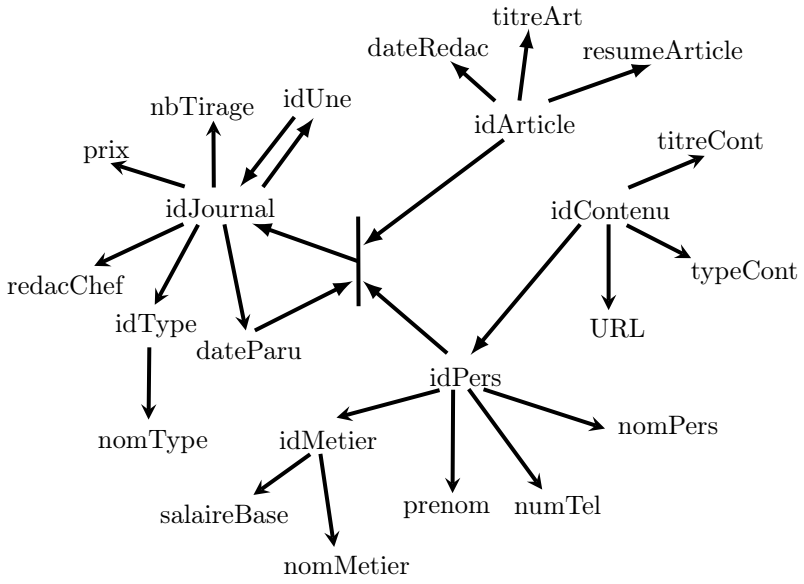

FIGURE 3 – Schéma des dépendances fonctionnelles de la base de données

Les dépendances (1), (4), (5), (6) et (7) ne sont pas élémentaires, nous avons donc dû les décomposer. Toutes les relations n'ont qu'un seul attributs à gauche, il n'est donc pas nécessaire de calculer la fermeture de ces attributs. Mais il est cependant nécessaire de le faire pour la dépendance (8) qui a plus d'un attribut en partie gauche.

Calcul de la fermeture des attributs en partie gauche de la dépendance fonctionnelle (8) :

- $idArticle+ = \{dateRedac, titreArt, resumeArticle\}$
- $idPers+ = \{nomPers, numTel, prenom, idMetier, nomMetier, salaireBase\}$
- $dateParu+ = \{\}$

Comme aucun élément des trois fermetures n'apparaît en partie gauche de la dépendance (8), cette dernière n'a pas besoin d'être modifiée.

Couverture minimale

(1.1) idJournal → prix

(1.2) idJournal → nbTirage

(1.3) idJournal → idUne

(1.4) idJournal → idType

(1.5) idJournal → redacChef

(1.6) idJournal → dateParu

(2) idType → nomType

(3) idUne → idJournal

(4.1) idArticle → dateRedac

(4.2) idArticle → titreArt

(4.3) idArticle → resumeArt

(5.1) idContenu → URL

(5.2) idContenu → titreCont

(5.3) idContenu → typeCont

(6.1) idPers → nomPers

(6.2) idPers → prénom

(6.3) idPers → numTel

(6.4) idPers → idMetier

(7.1) idMetier → salaireBase

(7.2) idMetier → nomMetier

(8) idArticle, idPers, dateParu → idJournal

Nous regroupons maintenant les dépendances ayant même partie gauche comme ci-dessous.

(1) $DF_1 = \{(1.1), (1.2), (1.3), (1.4), (1.5), (1.6)\}$

(2) $DF_2 = \{(2)\}$

(3) $DF_3 = \{(3)\}$

(4) $DF_4 = \{(4.1), (4.2), (4.3)\}$

(5) $DF_5 = \{(5.1), (5.2), (5.3)\}$

(6) $DF_6 = \{(6.1), (6.2), (6.3), (6.4)\}$

(7) $DF_7 = \{(7.1), (7.2)\}$

(8) $DF_8 = \{(8)\}$

On construit à présent les schémas $< R_i(U_i), DF_i >$ pour chaque DF_i, où U_i est l'ensemble des attributs apparaissant dans DF_i.

(1) $< R_1(idJournal, nbTirage, idUne, idType, redacChef, dateParu), DF_1 >$

(2) $< R_2(idType, nomType), DF_2 >$

(3) $< R_3(idUne, idJournal), DF_3 >$

(4) $< R_4(idArticle, dateRedac, titreArt, resumeArt), DF_4 >$

(5) $< R_5(idContenu, URL, titreCont, typeCont), DF_5 >$

(6) $< R_6(idPers, nomPers, prenom, numTel, idMetier), DF_6 >$

(7) $< R_7(idMetier, salaireBase, nomMetier), DF_7 >$

(8) $< R_8(idArticle, idPers, dateParu, idJournal), DF_8 >$

Pour terminer, comme la clé entière n'étant pas présente dans le schéma, on l'ajoute le schéma suivant $< R_9(idJournal, idArticle, idContenu), \{\} >$.

Après avoir appliqué de l'algorithme de Bernstein sur notre table de départ, nous obtenons neuf relations. Ce schéma est sans perte d'informations, car la dernière relation permet de relier toutes les tables entre elles grâce à la clé, sans perte de dépendances fonctionnelles, car nous n'avons perdu aucune dépendance fonctionnelle par rapport au modèle de base et est en troisième forme normale, car l'algorithme nous le garantit.

Mais qu'en est-il pour l'algorithme de décomposition ? Donne-t il le même résultat ?

2.3 Algorithme de décomposition

On considère la relation de départ
$R(idJournal, idContenu, idArticle, titreCont, typeCont, URL, dateRedac,$
$titreArt, resumeArticle, idPers, nomPers, numTel, prenom, idMetier,$
$salaireBase, nomMetier, dateParu, idUne, nbTirage, prix,$
$redacChef, idType, nomType)$ avec les dépendances fonctionnelles énumérées au début de ce rapport, $DF = \{(1), (2), (3), ..., (8)\}$.

On décompose la relation avec l'algorithme de décomposition vu en cours. Plusieurs résultats sont possibles. Dans cette exécution nous avons arbitrairement choisi l'ordre nous permettant de conserver les sous-ensembles les plus proches de la réalité de notre table. Les tables obtenues sont celles-ci :

(Étape 1) $R_1(idMetier, salaireBase, nomMetier), DF_1 = \{(7)\}$
$R_2(idJournal, idContenu, idArticle, titreCont, typeCont, URL,$
$dateRedac, titreArt, resumeArticle, idPers, nomPers, numTel,$
$prenom, idMetier, dateParu, idUne, nbTirage, prix, redacChef,$
$idType, nomType), DF_2 = \{(1), (2), (3), (4), (5), (6), (8)\}$

(Étape 2) $R_{21}(idType, nomType), DF_{21} = \{(2)\}$
$R_{22}(idJournal, idContenu, idArticle, titreCont, typeCont,$
$URL, dateRedac, titreArt, resumeArticle, idPers, nomPers,$
$numTel, prenom, idMetier, dateParu, idUne, nbTirage,$
$prix, redacChef, idType), DF_{22} = \{(1), (3), (4), (5), (6), (8)\}$

(Étape 3) $R_{221}(idPers, nomPers, prenom, numTel, idMetier), DF_{221} = \{(6)\}$
$R_{222}(idJournal, idContenu, idArticle, titreCont, typeCont, URL,$
$dateRedac, titreArt, resumeArticle, idPers, dateParu, idUne,$
$nbTirage, prix, redacChef, idType), DF_{222} = \{(1), (3), (4), (5), (8)\}$

(Étape 4) $R_{2221}(idContenu, URL, idPers, titreCont, typeCont), DF_{2221} = \{(5)\}$
$R_{2222}(idJournal, idContenu, idArticle, dateRedac, titreArt,$
$resumeArticle, dateParu, idUne, nbTirage, prix, redacChef, idType),$
$DF_{2222} = \{(1), (3), (4), (8)\}$

(Étape 5) $R_{22221}(idJournal, idUne, nbTirage, prix, redacChef, idType, dateParu),$
$DF_{22221} = \{(1)\}$
$R_{22222}(idJournal, idContenu, idArticle, dateRedac, titreArt,$
$resumeArticle), DF_{22222} = \{(3), (4), (8)\}$

(Étape 6) $R_{222221}(idArticle, dateRedac, titreArt, resumeArticle), DF_{222221} = \{(4)\}$
$R_{222222}(idArticle, idContenu, idJournal), DF_{222222} = \{()\}$

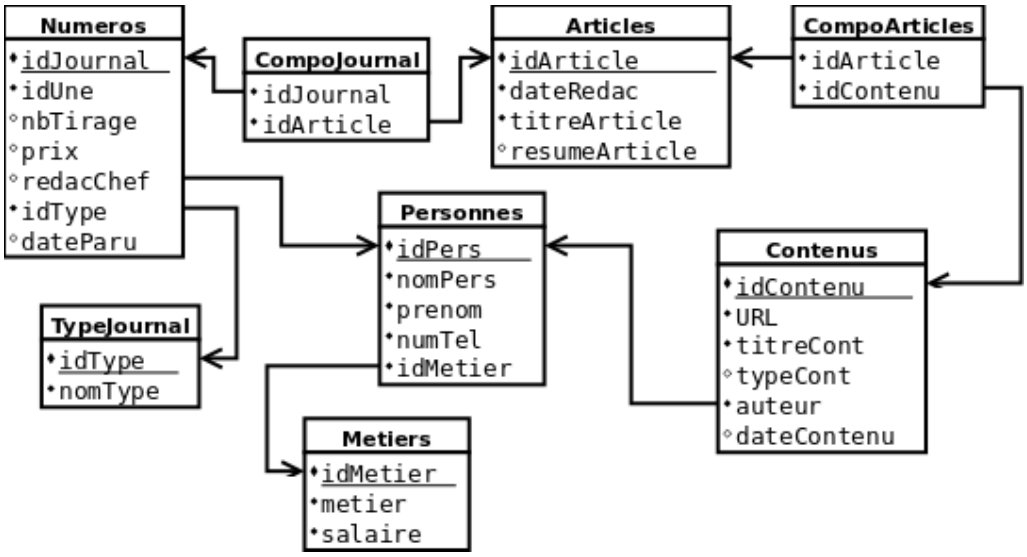

FIGURE 4 – Schéma des tables

On récupère ensuite les feuilles de l'arbre pour obtenir les tables à créer : $R_2, R_{22}, R_{222}, R_{2222}, R_{22222}, R_{222221} et R_{222222}$

Choix du schéma Nous venons de voir deux décompositions différentes de notre table de départ. Il faut à présent choisir l'une des deux pour l'exploitation de nos données dans un système de gestion de base de données.

Nous choisissons le résultat de l'algorithme de décomposition car le résultat est sans perte d'information et en troisième forme normale. Il s'approche en outre plus de la réalité de la façon dont on souhaite représenter les informations.

Nous avons néanmoins remarqué que cette décomposition posait problème au niveau de la dernière table regroupant les attributs clé : dans notre cas, un numéro est composé d'articles lui-même composé de contenus. Dans le cas ou un article se retrouvait dans plus d'un numéro, on aurait eu une répétition de la composition des articles dans cette table, nous avons donc préféré la diviser en deux : d'une part une table d'association $(idArticle, idContenu)$ et d'autre part une autre table $(idJournal, idArticle)$.

On obtient donc le schéma de base de données représenté sur la figure 4. Le fait de devoir effectuer cet ajustement indique peut-être une erreur dans la façon dont nous avons relevé nos dépendances fonctionnelles, une mauvaise exécution de l'algorithme de composition, ou une forme normale de niveau insuffisant.

3 Procédures stockées, vues et *triggers*

3.1 Procédures stockées

Ajout d'article On souhaite simplifier la possibilité d'ajouter un article pour les utilisateurs : la procédure ne demande qu'un titre et un résumé et remplit avec la date actuelle et l'identifiant issu de la séquence `seqArticle`.

Ajout de contenu d'article La plupart du temps, un contenu ajouté à la base de données est utilisé dans un article, on a donc créé une procédure qui permet d'ajouter un contenu à un article. Cette procédure a un double effet : d'une part ajouter le nouveau contenu à la table et d'autre part associer ce contenu à l'article dans la table d'association. En outre, un contenu requiert l'identifiant de son créateur : au lieu d'entrer celui-ci, la procédure demande le nom et le prénom de ce dernier puis associe l'identifiant correspondant en faisant une jointure avec la table `personnes`.

Ajout de personnes On souhaite aussi simplifier l'insertion d'une personne dans la base de données. Pour ce faire, nous créons une procédure `insert_personnes`, qui prend en paramètre les différents attributs de la table `Personnes` (id, nom, prenom, etc). La procédure commence par vérifier que l'id_metier passé en paramètre existe bien dans la table `Metiers`. Si ce n'est pas le cas, une exception est alors levée. Sinon, le nouvel enregistrement est inséré dans la table `Personnes`.

Cette procédure est utile pour insérer des nouvelles personnes rapidement et facilement dans la base, mais elle sera aussi utilisée par certains *triggers*. Cela permettra de séparer la logique liée à l'insertion de la logique des *triggers*.

3.2 Vues de la base de données

Pour faciliter la visualisation des données dans notre base, nous avons choisit de mettre en place quelques vues. Elles nous permettent d'avoir accès rapidement à certaines informations sans avoir à utiliser des requêtes complexes.

Vues des métiers La table `Metiers` contient une entrée pour chaque métier existant et est liée par une clé étrangère à la table `personnes` et associe ainsi une personne à son métier. Il pourrait être intéressant d'avoir accès rapidement à toutes personnes ayant le même métier.

Pour ce faire, nous créons une vue pour chaque métier recensé dans la table `metiers`. Chacune d'entre elle contiendra une requête sur la table `Personnes` qui sélectionnera les personnes ayant un id_metier correspondant à l'identifiant du métier que représente la vue. Par exemple, la vue `Illustrateurs` regroupe les personnes ayant un id_metier égal à 6, car c'est l'identifiant associé au métier d'illustrateur.

Toutes ces vues sont modifiables. En effet, elles ne sélectionnent que des attributs non nuls et elles n'opèrent que sur une seule table qui n'est pas utilisé dans une sous-requête.

Personnes et métiers Dans le but de faciliter la recherche des métiers de personnes, on a créé une vue associant les deux. Ainsi, on a une vue `personnes_metiers` qui est une projection des personnes et de leurs métiers. De cette manière, on dispose d'une vue associant les personnes à leur métier. Cette vue permet de faciliter certaines requêtes dans lesquelles on souhaite trouver cette correspondance.
La vue est modifiable. En effet, s'agissant d'une simple projection de deux tables, l'insertion de données dans cette dernière est répercutée sur les tables d'origines.

3.3 Traitements automatiques

Interactions avec les vues Étant donné que nous avons créé une vue par métiers existants, il peut être intéressant pour l'utilisateur d'insérer directement une personne dans la vue correspondant à son métier, ce qui équivaut à insérer la personne dans la table `Personnes` mais avec l'id_metier correspondant au métier que la vue représente. Pour permettre une telle insertion, nous créons, pour chaque vue liée à un métier, un *trigger* de type INSTEAD OF. Ce dernier se contente d'appeler la procédure `insert_personnes` en lui passant en paramètre les données de la requête ainsi que l'id_metier correspondant au métier de la vue.

Contraintes d'intégrité Nous avons aussi décidé de créer des *triggers* pour vérifier des contraintes d'intégrité lors des insertions et mises à jours dans certaines tables. Nous vérifions donc que, lors de l'insertion ou à la mise à jour d'un article dans la table `Articles` que la date de l'article ne dépasse pas la date d'aujourd'hui. De plus, lors de l'insertion ou la mise à jour d'un numéro dans la table `Numéros`, nous mettons à jour la date de parution.

Néanmoins, ces *triggers* ne fonctionnent pas car ils causent un problème de table mutante. Une façon de résoudre ce problème est d'utiliser des contraintes à la place des *triggers*. Ces contraintes étant assez simples, l'utilisation de *triggers* pour les vérifier est peu naturelle, les contraintes seraient donc à privilégier.

4 Critique de la base données

4.1 Droits d'accès à la base de données

Parce que nous travaillons à trois sur la base de données, il a fallu que les autres membres de l'équipe puissent voir et avoir accès à ce qu'a fait un autre membre. Pour cela, on a joué avec les droits d'accès de la base de données Oracle SQL. Pour simplifier la gestion des droits, on a créé toutes les tables sur un même compte, ce dernier attribuera ensuite des rôles précis aux personnes.

Typiquement pour notre projet, Nous avons deux rôles. Le rôle `resp` et `reader`. Un responsable (`resp`) a tous les droits sur les tables du projet alors qu'un lecteur (`reader`) n'a que le droit de consulter les tables. Le code PL/SQL ci-dessous donne un exemple d'accord de droits sur la table `articles` des droits de sélection, mise à jour, insertion et suppression des données pour le rôle `resp` au utilisateurs L3_3 et L3_13. L'utilisateur L3_4 n'a que le droit de consulter la table `articles`.

```
CREATE ROLE resp;
CREATE ROLE reader;

GRANT ALL ON articles TO resp;

GRANT SELECT ON articles TO reader;

GRANT resp TO L3_3, L3_13;
GRANT reader TO L3_4;
```

On procède de même pour chaque table, à savoir `contenus`, `metiers`, `numeros`, `personnes` et `typeJournal` ainsi que pour les différentes vues du projet.

Pour améliorer l'utilisation des rôles et se rapprocher encore plus de la réalité, on pourrait imaginer une vue `personnes` qui serait une sélection de la table du même nom sans certain attribut. Comme par exemple, les mots de passe des personnes (en supposant qu'elles en ont un). Ainsi en attribuant uniquement un rôle `reader` sur cette vue, et en attribuant aucun droit au rôle `reader` sur la table `personnes`, les utilisateurs qui ont le rôle `reader` ne peuvent pas voir, d'aucune manière que ce soit, les mots de passe des personnes.

4.2 Optimisation

L'une des optimisations la plus commune sur une base de données est d'ajouter des indexes sur les attributs des tables sur lesquelles on fait le plus requêtes. En effet, l'indexation de données permet au SGBD d'organiser et donc d'accéder de façon plus efficace à des données indexées.
Dans notre cas, la quasi-totalité des requêtes s'effectuent sur les clés des tables. Les clés étant indexées par défaut dans Oracle, nous n'avons pas eu besoin de demander explicitement au SGBD d'indexer nos tables. Donc, nous n'avons pas créé d'index.

4.3 Atouts

Sécurité La base de données est sécurisée. En effet, grâce aux rôles et aux divers droits qui leurs sont attribués, une personnes sensée pouvoir lire uniquement les données, ne pourra jamais les modifier. De même, en cachant certaines tables ou attributs de ses dernières, les informations qu'on ne souhaiterais pas rendre publique ne le sont pas.
En outre, certains *triggers* et contraintes (en particuliers les clés étrangères) assurent la cohérence des données stockées en base.

Optimisée Les requêtes travaillant principalement (99% du temps) sur des données indexées, l'interrogation de la base de données est rapide. De plus, les procédures stockées et les *triggers* nous permettent d'automatiser les traitements répétitifs. Quant aux vues, elles permettent, pour certaines, de réaliser des jointures naturelles fréquemment faites.

La schéma de la base se décomposant en plusieurs tables, ainsi, l'ajout d'attributs peut s'y faire aisément. Et il en est de même si on souhaite y ajouter une table supplémentaire.

4.4 Faiblesses et améliorations

Telle qu'elle est livrée, notre base de données permet d'ajouter du contenu à des journaux, des utilisateurs ainsi que de consulter son contenu. Ce qui remplit le contrat de base. Mais elle pourrait faire bien plus et à moindre frais. C'est ce qu'il est expliqué ci-dessous.

Connexion des utilisateurs Les utilisateurs de notre base de données possèdent un identifiant unique. Cet identifiant sert de clé primaire à la table des utilisateurs. Si on leur ajoute en plus un mot de passe, on pourrait ainsi permettre aux utilisateurs, de se connecter au serveur et ainsi de poster eux-même divers contenus. Cet attribut mot de passe serait bien-sûr caché dans la base de données grâce aux divers droits et rôles attribués aux utilisateurs.

Gestion des articles et des numéros Les procédures permettent d'ajouter de nouveaux éléments de façon simplifiée. Les modifications et suppressions restent néanmoins fastidieuses. On peut imaginer des procédures et déclencheurs supplémentaires permettant une gestion simplifiée de ces actions.
Par exemple, la suppression d'un article, contenu ou numéro référencé dans une table d'association est impossible : on aurait pu ajouter un *trigger* qui y supprime les lignes contenant l'élément que l'on cherche à supprimer. Les journaux ont pour référence une personne, le rédacteur chef : on peut imaginer que dans le cas où l'on supprime cette personne, cet attribut soit effacé ou modifié.

Cohérence Pour assurer la cohérence des données dans la base, nous avons ajouter des *triggers* au fur et à mesure que le projet avançait. Or, certains de ces *triggers* peuvent très bien être remplacés par des contraintes sur les tables. Un tel changement optimiserait encore plus la base de données mais nous ne l'avons pas fait car nous nous sommes focaliser sur d'autres points de la base de données.

Conclusion

Comme le montre ce rapport, nous avons décomposé notre schéma de base en plusieurs relations qui sont devenues des tables. Après avoir créé ces tables dans notre base de données, nous avons ajouté au projet des procédures, déclencheurs et vues pour pouvoir rendre la base de données plus intéressante et exploitable. Dans une certaine mesure, nous avons sécurisé l'utilisation de la base de données en créant des rôles spécifiques à divers utilisateurs.

Ce projet en groupe nous a permis de travailler à plusieurs et d'échanger nos points de vues sur l'utilisation et la gestion d'une base de données. De plus, c'était également un très bon exercice

Table des figures