CONCEPTION DES BASES DE DONNEES

Chapitre 1 : Rappels du modèle relationnel

1. *Le modèle relationnel*

1.1 Présentation

Le modèle relationnel représente la base de données comme un ensemble de tables, sans préjuger de la façon dont les informations sont stockées dans la machine. Les tables constituent donc la structure logique du modèle relationnel. Au niveau physique, le système est libre d'utiliser n'importe quelle technique de stockage (fichiers séquentiels, indexage, adressage dispersé, séries de pointeurs, compression…) dès lors qu'il est possible de relier ces structures à des tables au niveau logique. Les tables ne représentent donc qu'une abstraction de l'enregistrement physique des données en mémoire. De façon informelle, le modèle relationnel peut être défini de la manière suivante :

- Les données sont organisées sous forme de tables à deux dimensions, encore appelées relations, dont les lignes sont appelées **n-uplet** ou **tuple** en anglais ;
- Les données sont manipulées par des opérateurs de l'algèbre relationnelle ;
- L'état cohérent de la base est défini par un ensemble de contraintes d'intégrité.

1.2 Eléments constitutifs du modèle

Définition : Un **attribut** est un identifiant (un nom) décrivant une information stockée dans une base. Exemples d'attribut : *l'âge* d'une personne, son *nom*, le *numéro de sécurité sociale*.

Définition : Le **domaine** d'un attribut est l'ensemble, fini ou infini, de ses valeurs possibles.

Exemple : l'attribut *numéro de sécurité sociale* a pour domaine l'ensemble des combinaisons de quinze chiffres. L'attribut *nom* a pour domaine l'ensemble des combinaisons de lettres (une combinaison comme cette dernière est généralement appelée chaîne de caractères ou, plus simplement, chaîne).

Définition : Une **relation** est un sous-ensemble du produit cartésien de n domaines d'attributs (n>0). Une relation est représentée sous la forme d'une table à deux dimensions dans laquelle les n attributs correspondent aux titres des n colonnes.

Définition : Un **schéma de relation** précise le nom de la relation ainsi que la liste des attributs avec leurs domaines.

Exemple de relation avec son schéma :

PERSONNE		
NSécu	**Nom**	**Prénom**
354338532958234	KODJO	Afi
354338532958235	HODABALO	Abire
354338532958236	Kombate	Sambiani
354338532958237	OURO	Safiatou

On note cette relation de schéma de la façon suivante :
PERSONNE (NSécu : Entier, Nom : Chaîne, Prénom : Chaîne)

Définition : Le **degré** d'une relation est son nombre d'attributs.

Définition : Une **occurrence,** ou **n-uplets** ou **tuples**, est un élément de l'ensemble figuré par une relation. Autrement dit, une occurrence est une ligne de la table qui représente la relation.

Définition : La **cardinalité** d'une relation est son nombre d'occurrences.

Définition : Une **clé candidate** d'une relation est un ensemble minimal des attributs de la relation dont les valeurs identifient à coup sûr une occurrence.

La valeur d'une clé candidate est donc distincte pour tous les tuples de la relation. La notion de clé candidate est essentielle dans le modèle relationnel.

Règle : toute relation a au moins une clé candidate et peut en avoir plusieurs.
Ainsi, il ne peut jamais y avoir deux tuples identiques au sein d'une relation. Les clés candidates d'une relation n'ont pas forcément le même nombre d'attributs. Une clé candidate peut être formée d'un attribut arbitraire qui n'a d'autre objectif que de servir de clé.

Définition : La **clé primaire** d'une relation est une de ses clés candidates. Pour signaler la clé primaire, ses attributs sont généralement soulignés.

Définition : Une **clé étrangère** dans une relation est formée d'un ou plusieurs attributs qui constituent une clé candidate dans une autre relation. Attention, une clé étrangère != clé candidate dans une relation.

Définition : Un **schéma relationnel** est constitué par l'ensemble des schémas de relation avec mention des clés étrangères.

Définition : Une **base de données relationnelle** est constituée par l'ensemble des nuplets des différentes relations du schéma relationnel.

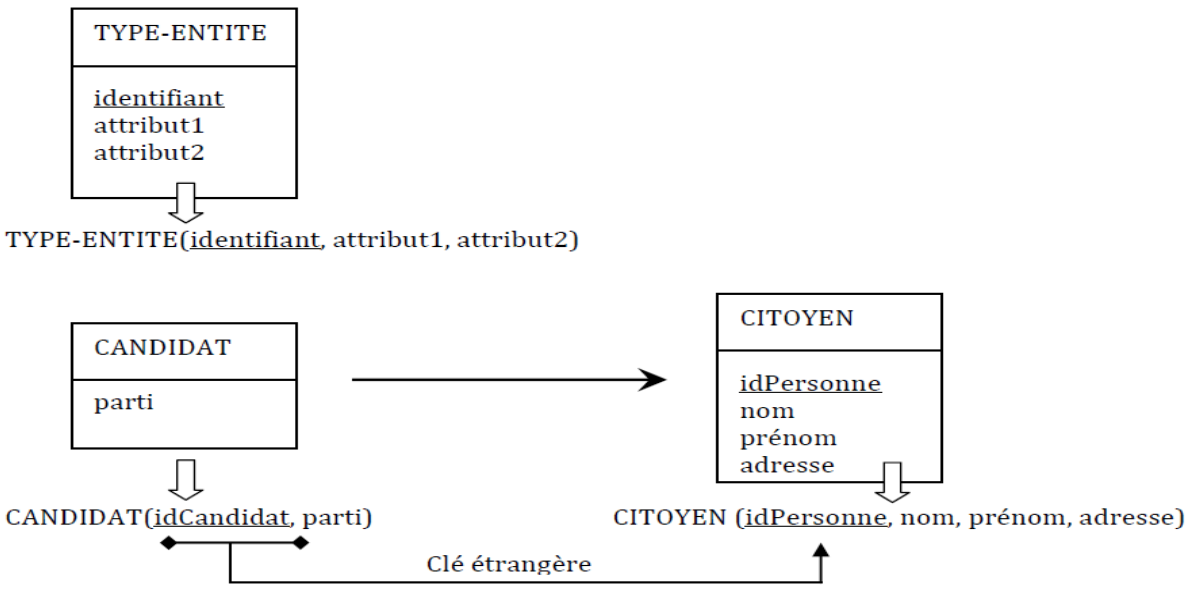
2. *Passage du modèle E/A au modèle relationnel*

2.1 Règles générales

Pour traduire un modèle entités-associations vers un modèle relationnel, il faut appliquer les règles suivantes :

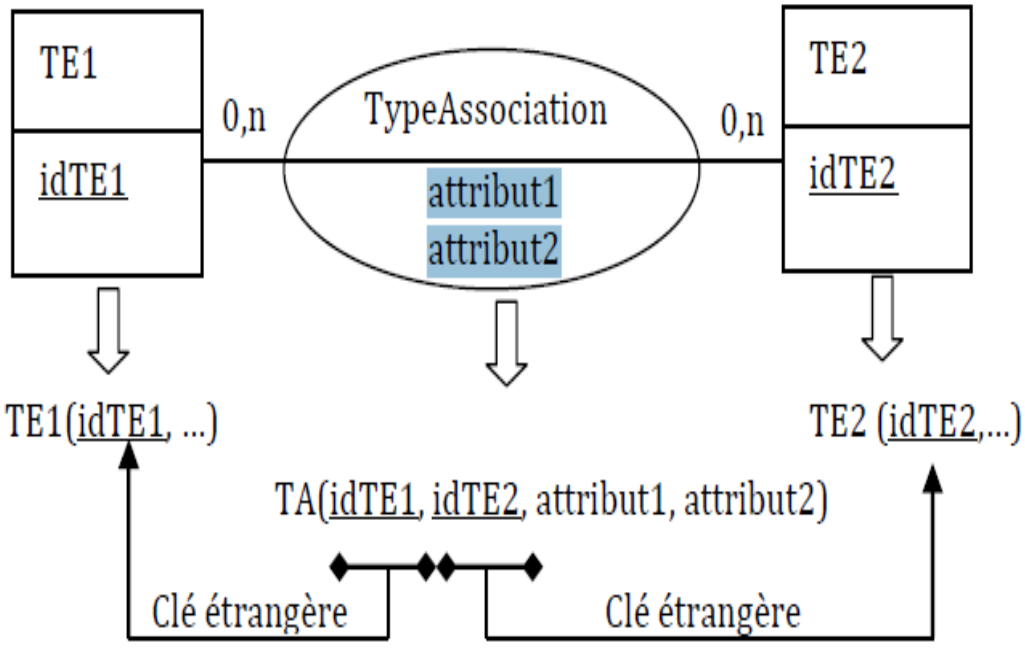
a. La normalisation devrait toujours être effectuée avant le passage au modèle relationnel. Dans les faits, elle est parfois faite a posteriori ce qui impose une surcharge de travail importante et produit un schéma relationnel non conforme au modèle entités –associations.

b. Chaque type-entité donne naissance à un schéma de relation. Chaque attribut de ce type-entité devient un attribut du schéma de relation. L'identifiant est conservé en tant que clé du schéma de relation. Il faut faire attention aux éventuels type entités spécifiques qui traduisent l'apparition d'au moins une clé étrangère. Cette étape est illustrée dans cette figure :



c. Chaque type-association maillé (chacune des pattes à pour cardinalité maximale n) donne naissance à un schéma de relation. Chaque attribut de ce

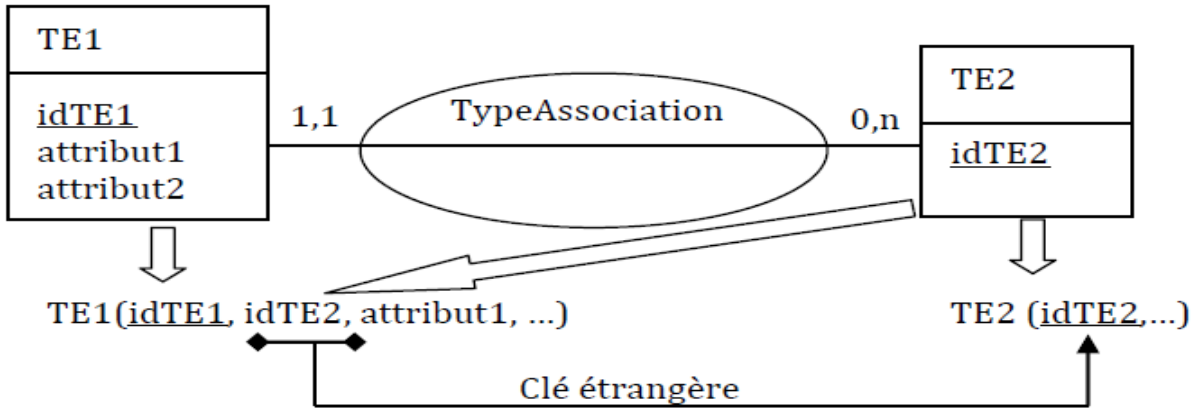
type association devient un attribut du schéma de relation. L'identifiant est formé par l'ensemble des identifiants des types-entités qui interviennent dans le type association. Chacun de ces identifiants devient une clé étrangère faisant référence au schéma de relation correspondant au type-entité dont l'identifiant provient cette étape est illustrée ci-dessous.



Chaque type-association maillé donne naissance à un schéma de relation dont la clé primaire est composée de clés étrangères.

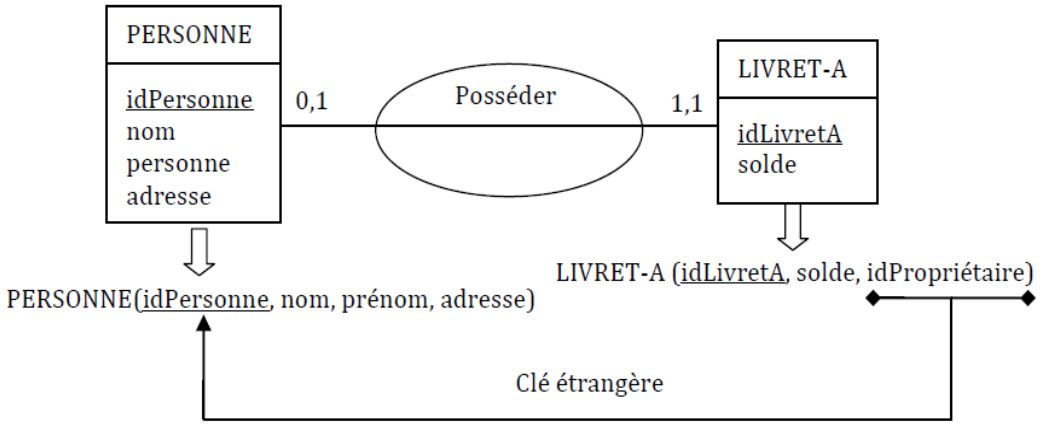
d. Un type-association dont une patte a une cardinalité maximale égale à 1 (il ne doit donc pas posséder d'attribut) ne devient pas un schéma de relation. Il décrit en effet une dépendance fonctionnelle. Le schéma de relation correspondant au type-entité dont la patte vers le type-association a une cardinalité maximale valant 1, se voit ajouter comme attribut (et donc comme clé étrangère) l'identifiant de l'autre type-entité. Attention, si la patte correspond à un lien identifiant, l'attribut ajouté doit être incorporé à la clé du schéma de relation.

Ainsi, un type-association fonctionnel ne devient pas un schéma de relation mais se traduit simplement par une clé étrangère.



2.2. Cas particulier d'un type-association 1 vers 1

Soit le type-association, Posséder, suivant :



Ce type-association a toutes ses cardinalités maximales à 1. L'application des règles de passage du modèle entités-associations au modèle relationnel que nous avons énoncées précédemment nous donnerait :

- PERSONNE (idPersonne, nom, prénom, adresse, idLivretA) où idLivretA est une clé étrangère qui fait référence au schéma de relation LIVRET-A.
- LIVRET-A(idLivretA, solde, idPersonne) où idPersonne est une clé étrangère qui fait référence au schéma de relation PERSONNE.

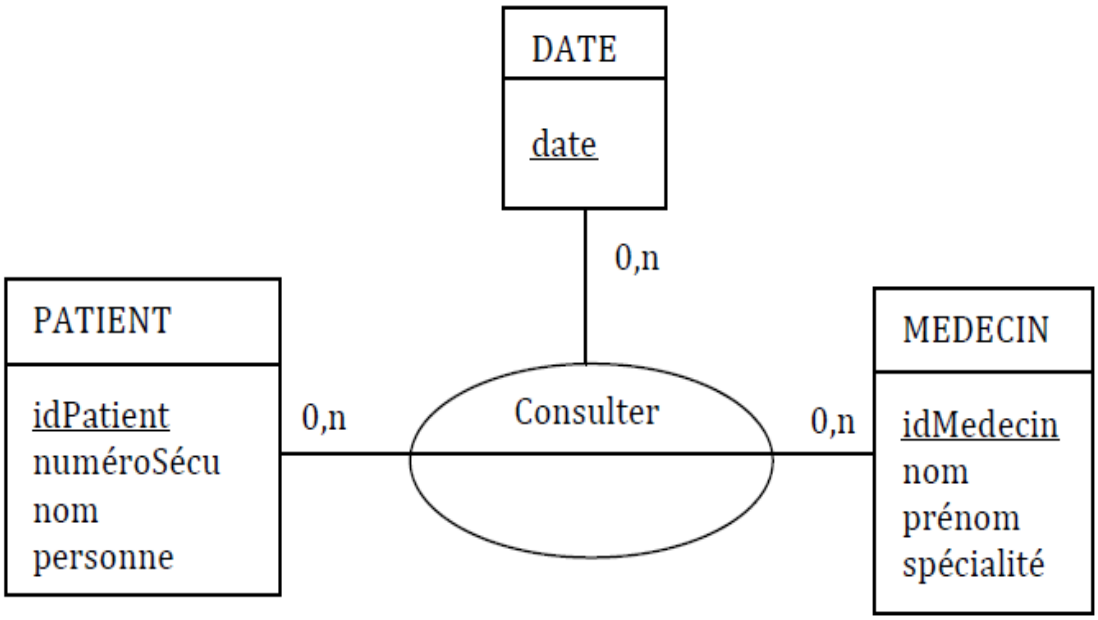
Le type-association Posséder étant du type 1 vers 1, il est entièrement matérialisé dans le schéma de relation LIVRET-A par la clé étrangère idPersonne. Il est donc inutile de la matérialiser à nouveau dans le schéma de relation PERSONNE (ou inversement). Il faut donc choisir de supprimer idLivretA de PERSONNE ou idPersonne de LIVRET-A. La cardinalité 0,1 nous indique le bon choix : une personne n'a pas forcément de livret A.

Le schéma relationnel adéquat devient donc ;

- PERSONNE (idPersonne, nom, prénom, adresse)
- LIVRET-A (idLivretA, solde, idPropriétaire) où idPropriétaire , nouveau nom de idPersonne, est une clé étrangère qui fait référence au schéma de relation PERSONNE.

2.3. Cas particulier d'un type-entité sans attribut autre que sa clé

Lorsqu'un type-entité ne possède pas d'attributs en dehors de sa clé, il ne faut pas nécessairement en faire un schéma de relation.



Dans cet exemple, le type-entité DATE ne doit pas se traduire par un schéma de relation, car ce schéma ne véhiculerait pas d'information. Le schéma relationnel adéquat correspond au modèle entités-associations de cet exemple est :

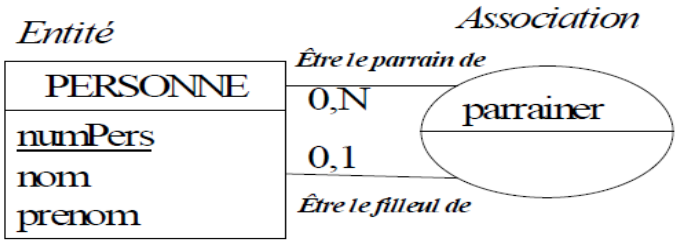
- PATIENT (idPatient, numéroSécu, nom, prénom)
- MEDECIN (idMédecin, nom, prénom, spécialité)
- CONSULTER (idPatient, idMédecin, date) où idPatient et idMédecin sont des clés étrangères qui font respectivement référence aux schémas de relation PATIENT et MEDECIN.

Par contre, si le type-entité sans attribut autre que sa clé correspond à un type énuméré (Comme GENRE par exemple), il faut le matérialiser par un schéma de relation.

2.4 Cas particulier d'une association réflexive

Une association réflexive se traduit selon l'application des associations binaires (1,1-0,n ou 1,1-1,n) ou N-naires (0,n-1,n ou 1,n-1,n ou 0,n-0,n), mais nécessite la

modification des noms de colonnes afin d'éviter les noms identiques (de plus, cela aide à la compréhension des liens).



OU BIEN

Tables relationnelles

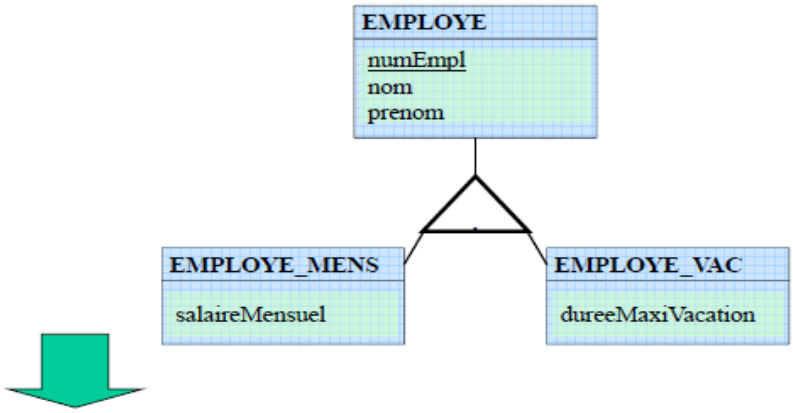
PERSONNE (numPers, nom, prenom)

PARRAINER (#numPers_filleul, #numPers_parrain)

PERSONNE (numPers, nom, prenom, #numPers_parrain)

2.5 Exemple Complet

Dans le cas de la spécialisation, 2 solutions sont possibles : les relations issues des entités spécialisées récupèrent l'identifiant et les propriétés de l'entité mère, ou bien elles deviennent des tables indépendantes et récupèrent l'identifiant de l'entité mère comme clef primaire.



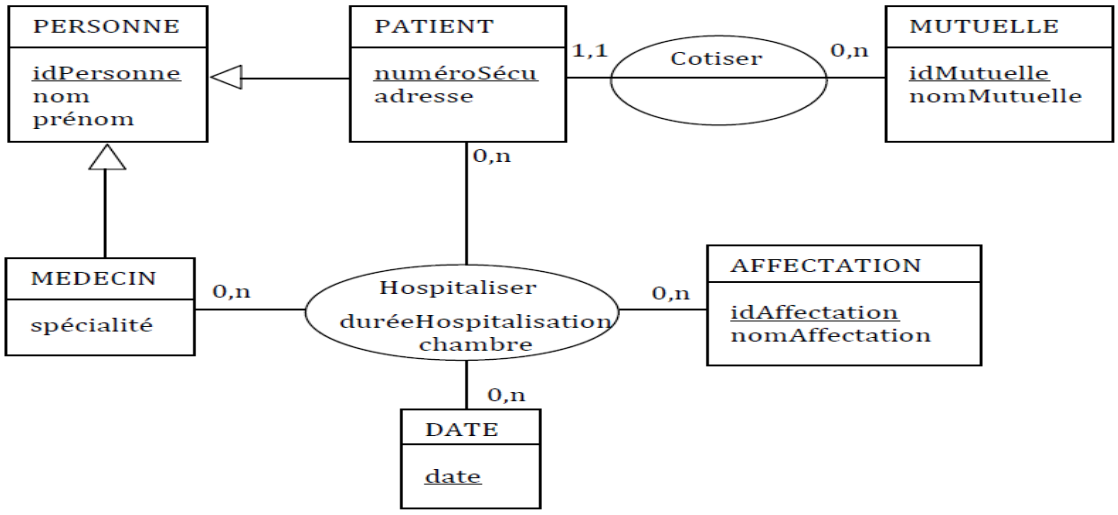
Tables relationnelles

EMPLOYE (numEmpl, nom, prenom)
EMPLOYE_MENS (#numEmpl, salaireMensuel)
EMPLOYE_VAC (#numEmpl, dureeMaxiVacation)

Tables relationnelles

EMPLOYE_MENS (numEmpl, nom, prenom ,salaireMensuel)
EMPLOYE_VAC (numEmpl, nom, prenom, dureeMaxiVacation)

2.6 Exemple Complet



Le schéma relationnel déduit de ce modèle entités-associations est :

- PERSONNE (idPersonne, nom, prénom)
- MEDECIN (idMedecin, spécialité) où idMedecin est une clé étrangère qui fait référence au schéma de relation PERSONNE

- PATIENT (idPatient, numéroSécu, adresse, mutuelle) où idPAtient et mutuelle sont des clés étrangères qui font respectivement référence aux schémas de Relation PERSONNE et MUTUELLE
- MUTUELLE (idMutuelle, nomMutuelle)
- AFFECTATION (idAffectation, nomAffectation)
- HOSPITALISER (idPatient, idAffectation, idMedecin, dateEntrée, chambre, durée-Hospitalisation) où idPatient, idAffectation et idMédecin sont des clés étrangères qui font respectivement référence aux schémas de relation PATIENT, AFFECTATION et MEDECIN.

Conclusion

Vous voilà mieux armé pour venir à bout des modèles relationnels. Cependant; il reste bien d'autres choses que nous n'avons pas vues, que vous devrez aller faire des recherches pour élargir vos connaissances.

Chapitre 2 : Normalisation

1.1 Introduction

L'objectif de la normalisation est de construire un schéma de base de données cohérent et possédant certaines propriétés vérifiées par la satisfaction de **formes normales**.

Pour une application spécifique, il est en effet possible de proposer plusieurs schémas. Les questions qui se posent alors sont les suivantes :

- qu'est-ce qu'un bon schéma ?
- quel schéma choisir ?

Un mauvais schéma défini lors de la phase de conception peut conduire à un certain nombre d'anomalies pendant la phase d'exploitation de la base :

- des **redondances** d'information,
- des **anomalies** lors des opérations de **mise à jour** (insertions, suppressions, modifications).

Exemple :

Soit le schéma de relation FOURNISSEUR (Nom_Fournisseur, Adresse, Produit, Prix).

Une relation (table) correspondant à ce schéma pourra éventuellement contenir plusieurs produits pour un même fournisseur. Dans ce cas, il faudra faire face à un certain nombre de problèmes :

- l'adresse du fournisseur sera dupliquée dans chaque n-uplet (redondance),
- si on souhaite modifier l'adresse d'un fournisseur, il faudra rechercher et mettre à jour tous les n-uplets correspondant à ce fournisseur,
- si on insère un nouveau produit pour un fournisseur déjà référencé, il faudra vérifier que l'adresse est identique,

- si on veut supprimer un fournisseur, il faudra retrouver et supprimer tous les n-uplets correspondant à ce fournisseur (pour différents produits) dans la table.

Ces anomalies n'apparaitront pas si on décompose le schéma initial de base de données.

Par contre, la décomposition d'un schéma relationnel au cours de la normalisation risque d'entrainer une dégradation des performances, du fait des opérations de

1.2 Dépendances fonctionnelles

Définition : un attribut (ou un groupe d'attributs) B est dit "fonctionnellement dépendant" d'un attribut (ou d'un groupe d'attributs) A si :

Pour chaque n-uplet de A on a un unique uplet de B

On dit alors que **A "détermine" B**, et on note **A -> B**.

Exemple :

Soit le schéma de relation PERSONNE (No_SS, Nom, Adresse, Age, Profession).

Les dépendances fonctionnelles qui s'appliquent sur ce schéma de relation sont les suivantes :

- No_SS -> Nom,
- No_SS -> Adresse,
- No_SS -> Age,
- No_SS -> Profession.

On pourra aussi écrire :

No_SS -> Nom Adresse Age Profession.

L'attribut No_SS détermine tous les attributs du schéma de relation. Il s'agit d'une **propriété de la clé** d'une schéma de relation.

Exercice :

Soit la relation suivante **r** de schéma **R** (A, B, C, D, E).

A	B	C	D	E
a1	b1	c1	d1	e1
a1	b2	c2	d2	e1
a2	b1	c3	d3	e1
a2	b1	c4	d3	e1
a3	b2	c5	d1	e1

Question : quelles sont les dépendances fonctionnelles satisfaites par R ?

Réponse : les dépendances fonctionnelles satisfaites par R sont les suivantes :

A -> E ; B -> E ; C -> ABDE ; D -> E ; AB -> D ; AD -> B ; BD -> A.

1.3. Axiomes d'Armstrong et couverture minimale

A partir d'un ensemble F de dépendances fonctionnelles entre les attributs d'un schéma de relation R, on peut en déduire d'autres à partir des trois propriétés suivantes :

1. transitivité : si X -> Y, et Y -> Z, alors X -> Z,
2. augmentation : si X -> Y, alors XZ -> Y pour tout groupe Z d'attributs appartenant au schéma de relation,
3. réflexivité : si X contient Y, alors X -> Y.

A partir de ces trois axiomes de base, on peut déduire d'autres règles :

1. union : si X -> Y et Y -> Z, alors X -> YZ,

2. pseudo-transitivité : si X -> Y et WY -> Z, alors WX -> Z,
3. décomposition : si X -> Y et Z contenu dans Y, alors X -> Z.

La **fermeture d'un ensemble d'attributs** X, notée **$(X)^+$**, représente l'ensemble des attributs de R qui peuvent être déduits de X à partir d'une famille de dépendances fonctionnelles en appliquant les axiomes d'Armstrong. Ainsi, Y sera inclus dans $(X)^+$ ssi X -> Y.

1.4 DF élémentaire

***Définition** : Dépendance fonctionnelle élémentaire*

Soit G un groupe d'attributs et A un attribut, une DF G→A est élémentaire si A n'est pas incluse dans G et s'il n'existe pas d'attribut A' de G qui détermine A.

Exemple : DF élémentaires

AB→C est élémentaire si ni A, ni B pris individuellement ne déterminent C.
Nom, DateNaissance, LieuNaissance→Prénom est élémentaire.

- RéférenceProduit → Désignation
- NuméroCommande, RéférenceProduit → Quantité

Exemple : DF non élémentaires

AB→A n'est pas élémentaire car A est incluse dans AB.
AB→CB n'est pas élémentaire car CB n'est pas un attribut, mais un groupe d'attributs.

- NuméroCommande, RéférenceProduit → Désignation
- La troisième dépendance fonctionnelle n'est élémentaire car il existe à l'intérieur d'elle: RéférenceProduit qui détermine la Désignation (RéférenceProduit → Désignation)

Remarque

On peut toujours réécrire un ensemble de DF en un ensemble de DFE, en supprimant les DF triviales obtenues par réflexivité et en décomposant les DF à partie droite non atomique en plusieurs DFE.

Réécriture de DF en DFE

On peut réécrire les DF non élémentaires de l'exemple précédent en les décomposant DFE :
AB→A n'est pas considérée car c'est une DF triviale obtenu par réflexivité.

AB→CB est décomposée en AB→C et AB→B, et AB→B n'est plus considérée car triviale.
N°SS→Nom, Prénom est décomposée en N°SS→Nom et N°SS→Prénom.

1.5. Dépendances fonctionnelles élémentaire directe:

On dit que la dépendance fonctionnelle A→ B est directe s'il n'existe aucun attribut C tel que l'on puisse avoir A → C et C→ B. En d'autres termes, cela signifie que la dépendance entre A et B ne peut être obtenue par transivité.

- Exemple:
- NumClasse → NumElève
- NumElève → NomElève
- NumClasse → NomElève

La troisième dépendance fonctionnelle n'est pas directe car nous pourrions écrire:

Car la dépendance fonctionnelle NumClasse → NomElève peut être déduite par transitivité

- NumClasse → NumElève et NumElève → NomElève donc NumClasse → NomElève

1.6 Notion de fermeture transitive des DFE

Définition : Fermeture transitive

On appelle fermeture transitive F+ d'un ensemble F de DFE, l'ensemble de toutes les DFE qui peuvent être composées par transitivité à partir des DFE de F.

Exemple

Soit l'ensemble F = {A→B, B→C, B→D, A→E}.
La fermeture transitive de F est F+ = { A→B, B→C, B→D, A→E, A→C, A→D }

1.7 Notion de couverture minimale des DFE

Définition : Couverture minimale

La couverture minimale d'un ensemble de DFE est un sous-ensemble minimum des DFE permettant de générer toutes les autres DFE.

Remarque

Tout ensemble de DFE (et donc tout ensemble de DF) admet au moins une couverture minimale (et en pratique souvent plusieurs).

Exemple

L'ensemble F = {A→B, A→C, B→C, C→B} admet les deux couvertures minimales : CM1 = {A→C, B→C, C→B} et CM2 = {A→B, B→C, C→B}

1.8 Notion de graphe des DFE

On peut représenter un ensemble de DFE par un graphe orienté (ou plus précisément un réseau de Pétri), tel que les nœuds sont les attributs et les arcs les DFE (avec un seul attribut en destination de chaque arc et éventuellement plusieurs en source).

Exemple : Relation Voiture

Soit la relation Voiture(NVH, Marque, Type, Puis, Couleur) avec l'ensemble des DF F= {NVH→Type, Type→Marque, Type→Puis, NVH→Couleur}. On peut représenter F par le graphe ci-dessous :

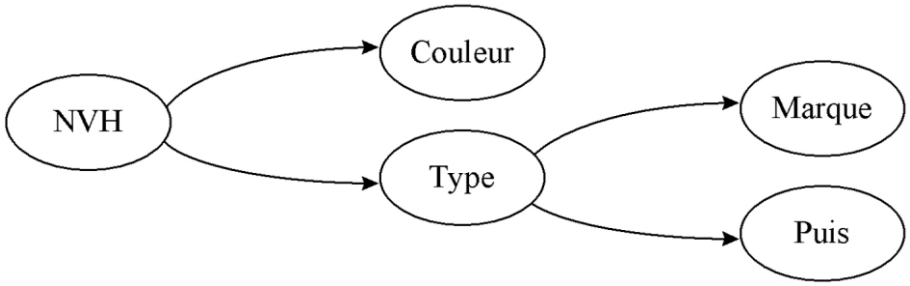


Image 1 Graphe des DFE de la relation Voiture

Exemple : Relation CodePostal

Soit la relation CodePostal(Code,Ville, Rue) avec l'ensemble desDF F={Code→Ville, (Ville,Rue)→Code}. On peut représenter F par le graphe ci-dessous :

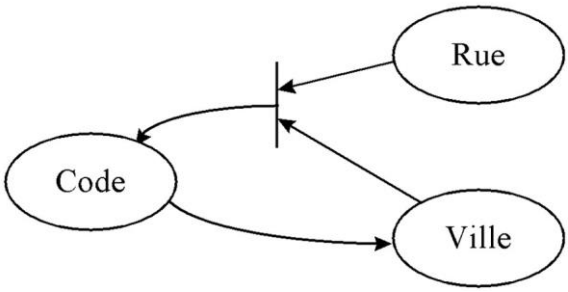


Image 2 Graphe des DFE de la relation CodePostal

1.9. Clé d'une relation

Définition1 : Une clé minimale, c'est une clé à laquelle on ne peut enlever aucun attribut pour garder les propriétés de la relation.

Définition 2 : Clé

Soient une relation R(A1,A2,...,An) et K un sous-ensemble de A1,A2,... ,An.

K est une clé de R si et seulement si :

1. K→A1, A2,..., An
2. et il n'existe pas X inclus dans K tel que X→A1,A2,...,An.

Déterminer les clés minimales :

1- Une relation a toujours au moins une clé : c'est l'ensemble de ses attributs (dans une relation les n-uplets sont tous des clés). On part de cette clé.
2- Tout attribut qui n'apparaît dans aucune dépendance fonctionnelle appartient à toutes les clés.
3- Un attribut qui n'apparaît qu'en partie droite de dépendances fonctionnelles n'apparaît dans aucune clé minimale.
4- Un attribut qui n'apparaît qu'en partie gauche de la dépendance fonctionnelle appartient à toutes les clés minimales.
5- Les attributs qui apparaissent à gauche de certaines dépendances et à droite d'autres peuvent apparaître ou non dans les clés minimales.

Ex : code postal
CP(Ville, Rue, ZIP, D)
Ville, Rue $\rightarrow$ Zip, Zip $\rightarrow$ Ville
Clé : Ville, Rue, Zip, D
Clé minimale : Ville, Rue, D ou Rue, Zip, D

Fondamental

Une clé est donc un ensemble minimum d'attributs d'une relation qui détermine tous les autres.

***Remarque**: Clés candidates et clé primaire*

Comporte plusieurs clés, chacune est dite clé candidate et l'on en choisit une en particulier pour être la clé primaire.

***Attention** : Les clés candidates sont des clés !*

Toutes les clés candidates sont des clés, pas seulement la clé primaire.

***Remarque**: Les clés candidates se déterminent mutuellement*

Toute clé candidate détermine les autres clés candidates, puisque qu'une clé détermine tous les attributs de la relation.

***Complément**: Relation "toute clé"*

Étant donné qu'une relation dispose forcément d'une clé, si une relation R n'admet aucune clé K sous ensemble des attributs A1..An de R, alors c'est que K=A1..An (la clé est composée de tous les attributs de R).

On parle de relation "toute clé".

Exercice :

Considérons le schéma de la relation suivante :

r (A, B, C, D, E)

Cette relation est définie en intention par les tuples suivants :

a1	b2	c2	d3	e2
a1	b2	c2	d1	e4
a2	b3	c2	d1	e5
a2	b4	c5	d1	e5

Après avoir énoncé les DF, déterminer, parmi les groupes d'attributs suivants, lesquels sont des clés ?

A, B ,C, D ,
E, {B,E},
{A,B,C,D,E}

1.10 Les formes normales

Le but essentiel de la normalisation est d'éviter les anomalies transactionnelles pouvant découler d'une mauvaise modélisation des données et ainsi éviter un certain nombre de problèmes potentiels tels que les anomalies de lecture, les anomalies d'écriture, la redondance des données et la contre-performance.

La normalisation des modèles de données permet de vérifier la robustesse de leur conception pour améliorer la modélisation (et donc obtenir une meilleure représentation) et faciliter la mémorisation des données en évitant la redondance et les problèmes sous-jacents de mise à jour ou de cohérence. La normalisation s'applique à toutes les entités et aux relations porteuses de propriétés.

1.10.1 Avantages et inconvénients

En pratique, la première et la deuxième forme normale sont nécessaires pour avoir un modèle relationnel juste. Les formes normales supplémentaires ont leurs avantages et leurs inconvénients.

Les avantages sont :

- De limiter les redondances de données (multiples écritures) ;
- Donc de limiter l'espace disque nécessaire ;
- De limiter les incohérences de données qui pourraient les rendre inutilisables (multiples écritures) ;
- D'éviter les processus de mise à jour (réécritures).

Les inconvénients sont :

- Des temps d'accès potentiellement plus longs si les requêtes sont trop complexes (lectures plus lentes), cela dit, avec des index, notamment sur les clefs primaires, on peut généralement obtenir des résultats satisfaisants ;
- Une plus grande fragilité des données étant donné la non-redondance (lecture impossible) ;
- La difficulté de réorganiser la structure (dans le cas où l'on ne sait pas à l'avance le lien entre les entités, ce qui est à éviter si possible).

Pour des petites bases de données, se limiter à la troisième forme normale est généralement une des meilleures solutions d'un point de vue architecture de base de données, mais pour des bases de données plus importantes, cela n'est pas toujours le cas. Il s'agit de faire l'équilibre entre deux options :

- La génération dynamique des données via les jointures entre tables ;
- L'utilisation statique de données correctement mises à jour.

En d'autres mots, il ressort de ces avantages et inconvénients qu'un arbitrage devra être effectué sur le niveau de normalisation selon que les tables de la base de données sont plus sollicitées en lecture ou plus en écriture. Si une table (base de données) est plus intensivement écrite que lue, il sera préférable de normaliser le plus possible. À contrario, si une table (base de données) est plus intensivement lue qu'écrite, il pourra être judicieux d'être moins strict sur le respect de la normalisation pour permettre d'améliorer les performances d'accès aux données.

La troisième forme normale est généralement reconnue comme étant la plus importante à respecter.

1.10.2. Principe de la décomposition

L'objectif de la décomposition est de "casser" une relation en relations plus petites afin d'en éliminer les redondances et sans perdre d'information.

Définition : Décomposition

La décomposition d'un schéma de relation R(A1,A2,...,An) est le processus de remplacement de ce schéma par une collection de schémas R1,R2,...,Rn telle qu'il est possible de reconstruire R par des opérations relationnelles de jointure sur R1,R2,...,Rn.

Définition : Décomposition préservant les DF

Une décomposition d'une relation R en relations R1,R2,...Rn préserve les DF si la fermeture transitive F+ des DF de R est la même que celle de l'union des fermetures transitives des DF de R1,R2,...,Rn.

***Exemple** : Décomposition préservant les DF d'une relation Voiture*

Soit la relation Voiture (Numéro,Marque,Type,Puissance,Couleur) avec la fermeture transitive suivante :

Numéro→Marque
Numéro→Type
Numéro→Puissance
Numéro→Couleur
Type→Marque
Type→Puissance

On peut décomposer Voiture en préservant les DF en deux relations :

R1(Numéro,Type,Couleur)
R2(Type,Puissance,Marque).

1.10.3. Première forme normale

Définition : 1NF

Une relation est en 1NF si elle possède au moins une clé et si tous ses attributs sont atomiques.

Définition : Attribut atomique

Un attribut est atomique s'il ne contient qu'une seule valeur pour un tuple donné, et donc s'il ne regroupe pas un ensemble de plusieurs valeurs.

***Exemple1** : Avoir plusieurs métiers*

Soit la relation Personne instanciée par deux tuples:

Personne(#Nom, Profession)

1 (Dupont, Géomètre)
2 (Durand, (Ingénieur,Professeur))

La relation n'est pas en 1NF, car l'attribut Profession peut contenir plusieurs valeurs.

Pour que la relation soit en 1NF, on pourrait par exemple ajouter Profession à la clé et faire apparaître deux tuples pour Durand, on obtiendrait :

Personne (#Nom, #Profession)

1 (Dupont, Géomètre)

2 (Durand, Ingénieur)
3 (Durand, Professeur)

Une autre solution aurait été d'ajouter un attribut ProfessionSecondaire. On obtiendrait ainsi :

Personne (#Nom, Profession, ProfessionSecondaire)

1 (Dupont, Géomètre, Null)
2 (Durand, Ingénieur, Professeur)

Exemple2 : PRODUIT_FOURNISSEUR (Produit, Fournisseur)

Produit	Fournisseur
téléviseur	VIDEO SA, HITEK LTD

Dans ce cas les valeurs du fournisseur sont multivaluées et ne sont pas atomiques. Pour que cette relation soit en première forme normale, il faut décomposer les attributs de la colonne fournisseur comme suit :

Solution:

Produit	Fournisseur
téléviseur	VIDEO SA
téléviseur	HITEK LTD

Remarque: Relativité de la notion d'atomicité

L'atomicité d'un attribut est souvent relative : on peut décider qu'un attribut contenant une date n'est pas atomique (et que le jour, le mois et l'année constituent chacun une valeur), ou bien que l'attribut est de domaine date et donc qu'il est atomique.

1.10.4. Deuxième forme normale

Introduction

La deuxième forme normale permet d'éliminer les dépendances entre des parties de clé et des attributs n'appartenant pas à une clé.

Définition : 2NF

Une relation est en 2NF si elle est en 1NF et si tout attribut qui n'est pas dans une clé ne dépend pas d'une partie seulement d'une clé. C'est à dire encore que toutes les DF issues d'une clé sont élémentaires.

Exemple1 *: Echelle de salaire*

Soit la relation Personne :

Personne(#Nom, #Profession, Salaire)

Soit les DF suivantes sur cette relation :

Nom, Profession → Salaire
Profession→Salaire

On note alors que Salaire dépend d'une partie de la clé Profession et non de la totalité de la clé et donc que le schéma n'est pas en 2NF.

Pour avoir un schéma relationnel en 2NF, il faut alors décomposer Personne en deux relations :

Personne(#Nom, #Profession)
Profession(#Profession, Salaire)

On remarque que ce schéma est en 2NF (puisque Salaire dépend maintenant fonctionnellement d'une clé et non plus d'une partie de clé).

On remarque aussi que la décomposition a préservé les DF, puisque nous avons à présent :

Profession→Salaire (DF de la relation Profession)
Nom,Profession→Profession (par Réflexivité)

Nom,Profession→Salaire (par Transitivité)

Exemple2:

PRODUIT_FOURNISSEUR(Produit, Fournisseur , Adresse fournisseur)

Produit	Fournisseur	Adresse fournisseur
téléviseur	VIDEO SA	13 rue du cherche-midi
écran plat	VIDEO SA	13 rue du cherche-midi
téléviseur	HITEK LTD	25 Bond Street

Admettons que la clé de cette table soit une clé composite (produit - fournisseur). Dans le cas d'un changement d'adresse d'un fournisseur, il faudra faire preuve de beaucoup d'attention pour n'oublier aucun endroit où l'adresse est mentionnée. En effet, on constate que le champ adresse ne dépend que d'une partie de la clé : le champ fournisseur, ce qui induit la possibilité d'une redondance au sein de la table. Il convient donc de scinder la table en deux:

Solution en seconde forme normale :

PRODUIT (Produit, Fournisseur),

Produit	Fournisseur
téléviseur	VIDEO SA
téléviseur	HITEK LTD
écran plat	VIDEO SA

FOURNISSEUR(Fournisseur , Adresse fournisseur)

Fournisseur	Adresse fournisseur
VIDEO SA	13 rue du cherche-midi
HITEK LTD	25 Bond Street

1.10.5. Troisième forme normale

Introduction

La troisième forme normale permet d'éliminer les dépendances entre les attributs n'appartenant pas à une clé.

Définition : 3NF

Une relation est en 3NF si elle est en 2NF et si tout attribut n'appartenant pas à une clé ne dépend pas d'un autre attribut n'appartenant pas à une clé.

Exemple1 : Échelle de salaire et de prime

Soit la relation Profession :

Profession (#Profession, Salaire, Prime)

Soit les DF suivantes sur cette relation :

Profession→Salaire
Profession→Prime
Salaire→Prime

Cette relation n'est pas en 3NF car Salaire, qui n'est pas une clé, détermine Prime.

Pour avoir un schéma relationnel en 3NF, il faut décomposer Profession :

1 Profession(#Profession, Salaire=>Salaire)
2 Salaire(#Salaire, Prime)

Ce schéma est en 3NF, car Prime est maintenant déterminé par une clé.

On remarque que cette décomposition préserve les DF, car par transitivité, Profession détermine Salaire qui détermine Prime, et donc Profession détermine toujours Prime.

Exemple 2 :

*exemple: FOURNISSEUR(*Fournisseur,Adressefournisseur,Ville,Pays)

Fournisseur	Adresse fournisseur	Ville	Pays
VIDEO SA	13 rue du cherche-midi	PARIS	FRANCE
HITEK LTD	25 Bond Street	LONDON	ENGLAND

Le pays de l'adresse n'est pas dépendant de la clé de la table, à savoir le nom du fournisseur, mais est fonction de la ville de l'adresse. De nouveau, il est préférable de scinder la table en deux:

Solution normalisée :

*FOURNISSEUR (*Fournisseur, Adresse fournisseur, Ville)

Fournisseur	Adresse fournisseur	Ville
VIDEO SA	13 rue du cherche-midi	PARIS
HITEK LTD	25 Bond Street	LONDON

VILLE (Ville , Pays)

Ville	Pays
PARIS	FRANCE
LONDON	ENGLAND

De cette manière, une modification de l'orthographe pour un pays (par exemple : ENGLAND en GREAT BRITAIN) ne donnera lieu qu'à une seule modification.

Fondamental :

Il est souhaitable que les relations logiques soient en 3NF. En effet, il existe toujours une décomposition sans perte d'information et préservant les DF d'un schéma en 3NF. Si les formes normales suivantes (BCNF, 4NF et 5NF) assurent un niveau de redondance encore plus faible, la décomposition permettant de les atteindre ne préserve plus les DF.

Remarque*: Limite de la 3NF*

Une relation en 3NF permet des dépendances entre des attributs n'appartenant pas à une clé vers des parties de clé.

1.10.6. Forme normale de Boyce-Codd

Introduction

La forme normale de Boyce-Codd permet d'éliminer les dépendances entre les attributs n'appartenant pas à une clé vers les parties de clé.

Définition : BCNF

Une relation est en BCNF si elle est en 3NF et si tout attribut qui n'appartient pas à une clé n'est pas source d'une DF vers une partie d'une clé. C'est à dire que les seules DFE existantes sont celles dans lesquelles une clé détermine un attribut.

Exemple: Employés

Soit la relation Personne :

Personne(#N°SS, #Pays, Nom, Région)

Soit les DF suivantes sur cette relation:

N°SS,Pays→Nom
N°SS,Pays→Région
Région→Pays

Il existe une DFE qui n'est pas issue d'une clé et qui détermine un attribut appartenant à une clé. Cette relation est en 3NF, mais pas en BCNF (car en BCNF toutes les DFE sont issues d'une clé).

Pour avoir un schéma relationnel en BCNF, il faut décomposer Personne :

Personne(#N°SS, #Region=>Region, Nom)

Region(#Region, Pays)

Remarquons que les DF n'ont pas été préservées par la décomposition puisque
N°SS et Pays ne déterminent plus Région.

Remarque: Simplicité

La BCNF est la forme normale la plus facile à appréhender intuitivement et formellement, puisque les seules DFE existantes sont de la forme K→A où K est une clé.

Attention : Non préservation des DF

Une décomposition en BCNF ne préserve pas toujours les DF.

Chapitre 3 : Le Langage de Définition de Données avec les Contraintes d'Intégrités

SQL ne se résume pas aux requêtes d'interrogation d'une base. Ce langage permet aussi de :

- Créer des tables
- De modifier la structure de tables existantes
- De modifier le contenu des tables
- D'octroyer et de retirer des droits et privilèges

Le LDD est le langage de définition de données (DDL en anglais).

Il permet de créer des tables par l'instruction CREATE TABLE et de modifier la STRUCTURE (et non le contenu) des TABLES avec ALTER TABLE ou encore de supprimer une table avec DROP TABLE.

Le LMD est le langage de manipulation de données. Parfois on inclut le LID (langage d'interrogation de données) dans cette partie. Il permet de modifier le CONTENU d'une table en

- Ajoutant de nouvelles lignes INSERT INTO nom_table
- Modifiant une ou plusieurs lignes UPDATE nom_table SET
- Supprimant des lignes DELETE FROM nom_table

1. Introduction au langage de définition de données

1.1. Création d'une table : CREATE TABLE

1.1.1 Syntaxe

Syntaxe simple :

CREATE TABLE nom_table

(nom_Colonne 1 type_Colonne 1,

Nom_Colonne 2 type_Colonne 2,

……

Nom_colonne n type_Colonne n) ;

1.1.2 Exemple :

CREATE TABLE employe

(

empno number(8) not null,

nom varchar2(20),

fonction varchar2(10), sal

number(8,2),

code_service char(3),

date_embauche date) ;

1.1.3 Les contraintes liées aux colonnes et tables

NOT NULL

Indique qu'on est obligé de mettre une valeur dans la colonne

UNIQUE

Indique que les valeurs dans la colonne doivent être toutes différentes (comme pour les clés, sauf que ce n'est pas une clé)

DEFAULT

Permet d'indiquer une valeur par défaut à la création d'une nouvelle ligne

Ex :

```
CREATE TABLE compte
(
num_cpte NUMBER(11) NOT NULL,
type_cpt VARCHAR2(5) DEFAULT "chèque",
solde NUMBER(10, 2) DEFAULT 0,
…
);
```

et les contraintes (clé primaire, clé étrangère, …) voir le paragraphe 3 (approfondissements)

1.2 La modification d'une table : ALTER TABLE

Ajouter une ou plusieurs colonnes

```
ALTER TABLE nom_table
ADD (nom_colonne type_colonne);
```

On peut ajouter plusieurs colonnes en les séparant par des virgules dans les parenthèses du ADD Exemples:

```
ALTER TABL E service
ADD (budget number(6) ) ;
```

Modifier le type ou tout autre propriété d'une colonne

ALTER TABLE nom_table

MODIFY (nom_colonne nouveau_type_et/ou_nouvelle_propriété);

Exemples:

ALTER TABLE employe

MODIFY (sal number(10,2), fonction varchar2(30)) ;

ALTER TABLE emp

MODIFY(job null) ;

1.3 Renommer et supprimer une table

Supprimer une table

DROP TABLE nom_table;

Ex : DROP table employe ;

Renommer une table

RENAME TABLE nom_table TO nouveau_nom_table;

Ex : RENAME employe to salarie ;

2. Les contraintes d'intégrité

Une contrainte d'intégrité est une règle qui permet d'assurer la validité (cohérence) des données stockées dans une base.

Le SGBD contrôle les contraintes d'intégrité à chaque modification dans les tables (saisies, modification ou suppression).

Les différentes contraintes que l'on peut définir sont :

- non nullité (obligation) : NOT NULL

La colonne ne peut pas contenir de valeurs NULL.

- Unicité : UNIQUE

Toutes les valeurs de la (des) colonnes doivent être différentes ou NULL

- Clé primaire : PRIMARY KEY

Chaque ligne de la table doit avoir une valeur différente pour la ou les colonnes qui font partie de la clé primaire. Les valeurs NULL sont rejetées. primary key = unique + not null

- Valeur par défaut : DEFAULT

Permet de spécifier une valeur par défaut à la colonne (dans le cas où aucune valeur n'est explicitement donnée)

- Intégrité de domaine : CHECK

Permet de spécifier les valeurs acceptables pour une colonne.

- Intégrité référentielle (clé étrangère) : FOREIGN KEY

Cette colonne fait référence à une colonne clé d'une autre table.

Nous allons voir comment une contrainte d'intégrité peut se définir au moment de la création d'une table (nous verrons plus tard d'autres techniques).

Les contraintes peuvent soit concerner uniquement une seule colonne, soit concerner une table toute entière. Dans les deux cas, la syntaxe d'expression est un peu différente.

Les contraintes de colonne s'écrivent tout simplement après la colonne à laquelle elles se rapportent. Attention, on ne met pas de virgule entre la définition de la colonne et celle de la contrainte.

2.1 Contrainte d'obligation : NOT NULL

Cette contrainte permet de rendre obligatoire la saisie d'une colonne.

Exemple :

```
CREATE TABLE PERSONNE
(
NomPers     Varchar2(30) NOT NULL,
PrenomPers  Varchar2(30)
) ;
INSERT INTO PERSONNE VALUES ('Dupont', 'Jean');
INSERT INTO PERSONNE VALUES ('Martin', NULL);
                                                    autorisé
~~INSERT INTO PERSONNE VALUES (NULL, 'toto');~~      refusé
~~INSERT INTO PERSONNE(PrénomPers) VALUES ('titi');~~
_____refusé
```

Car il faut obligatoirement que le nom ait une valeur.

2.2 Contrainte d'unicité : UNIQUE

Cette contrainte permet de contrôler que chaque ligne a une valeur unique pour la colonne ou l'ensemble de colonne unique (pas de doublons)

Sur une seule colonne

```
CREATE TABLE PERSONNE (…
 Téléphone CHAR(10) UNIQUE
);
```

Sur plusieurs colonnes :

```
CREATE TABLE PERSONNE
( NomPers  Varchar2(30)  NOT  NULL,
  PrénomPers Varchar2(30),
  UNIQUE (NomPers, PrénomPers)
```

);

Une telle contrainte indique qu'on ne pourra pas avoir deux personnes ayant le même nom et le même prénom.

2.3 Contrainte de clé primaire

La contrainte de clé primaire permet d'indiquer que la ou les colonnes sont uniques et ne peuvent pas avoir de valeur nulle. On peut dire que :

PRIMARY KEY = UNIQUE + NOT NULL

Si la table possède une clé composée d'une seule colonne, alors il est possible de l'indiquer juste après la colonne clé.

```
CREATE TABLE
PERSONNE (idPers
PRIMARY KEY,
... );
```

Si la table a une clé compose de plusieurs colonnes, alors il est impossible d'indiquer PRIMARY KEY au niveau des colonnes.

```
CREATE TABLE P ERSONNE
(NomPers Varchar2(30) PRIMARY KEY,
 PrénomPers Varchar2(30) PRIMARY KEY,
...);
```

On doit déclarer la clé à part, après la déclaration des colonnes.

```
CREATE TABLE PERSONNE
(NomPers
varchar2(30),
PrénomPers
Varchar2(30),
PRIMARY KEY (NomPers,
```

PrénomPers));

Autre exemple :

```
CREATE TABLE ligne_commande
(no_commande NUMBER(5),
code_article VARCHAR2(10),
quantite NOMBER(3),
PRIMARY KEY (no_commande,code_article)
);
```

2.4 Valeur par défaut : DEFAULT

```
CREATE TABLE PERSONNE
(idPers PRIMARY KEY,
NomPers Varchar2(30) NOT NULL,
PrénomPers Varchar2(30) DEFAULT 'prénom inconnu',
DateNaiss Date DEFAULT CURRENT DATE
);
```

Ainsi, si on insère la personne toto de la manière suivante :

INSERT INTO PERSONNE (idPers, NomPers) VALUES (100, 'toto');

Alors les valeurs de ses champs seront :

(100, 'toto', 'prénom inconnu', '17/03/04') ;

2.5 Contrainte de domaine : CHECK

La définition des types de données pour chaque colonne définit déjà un domaine pour les valeurs. On peut encore restreindre les domaines grâce à la clause CHECK.

Attention, cette contrainte est très "coûteuse" en temps.

Syntaxe : après le mot clé CHECK, on indique entre parenthèses

le critère à vérifier.

Exemple:

```
CREATE TABLE
PERSONNE
( idpers   Number  PRIMARY
KEY,  NomPers  Varchar2(30)
NOT NULL,
PrenomPers      Varchar2(30)      DEFAULT
'prénom inconnu', age Number CHECK (age
>= 0 AND age < 130),
  etat_civil  Varchar2(20)  CHECK (etat_civil  IN ('marié(e)', célibataire', 'veuf(ve)',
  'divorcé(e)'),
 );
```

2.6 Exemple récapitulatif

```
CREATE TABLE
PERSONNE
( idpers  Number PRIMARY KEY,
nom Varchar2(30) NOT NULL,
prenom Varchar2(30) DEFAULT 'prénom
inconnu', age Number CHECK (age >= 0
AND age < 130),
etat_civil  Varchar2(20)  CHECK  (etat_civil  IN  ('marié(e)',  célibataire',  'veuf(ve)',
'divorcé(e)')
 );
```

Il est possible le de donner un nom à chaque contrainte (sinon le système en donne un par défaut). Il suffit de faire précéder la contrainte par le

mot-clé CONSTRAINT suivi de son nom

```
CREATE TABLE PERSONNE
( idpers  Number CONSTRAINT clé_primaire PRIMARY KEY,
nom Varchar2(30) CONSTRAINT nom_existant NOT NULL,
prenom Varchar2(30) CONSTRAINT prénom_par_défaut DEFAULT 'prénom
inconnu',
age Number CONSTRAINT verify_age CHECK (age >= 0  AND
age < 130), etat_civil Varchar2(20) CONSTRAINT
domaine_état_civil
        CHECK (etat_civil IN ('marié(e)', célibataire', 'veuf(ve)', 'divorcé(e)')
 );
```

2.7 Intégrité référentielle : REFERENCES

L'intégrité référentielle permet de vérifier la cohérence des liens entre tables, lors de l'ajout, de la suppression et de la modification de lignes. Elle est utilisée lorsqu'on a une clé étrangère. Elle permet d'assurer que toute valeur de clé étrangère correspond bien à une valeur de clé primaire de la table liée.

Voyons tout d'abord la syntaxe avec un exemple:

Soit le modèle relationnel suivant :

CLIENT (id_client, Nom_client, …)

FACTURE (code_fact, #id_client, …) Voilà comment on le traduit en SQL :

```
CREATE TABLE CLIENT
(id_client Number PRIMARY KEY,
 Nom_client Varchar2(30) NOT NULL,
 … );
```

```
CREATE TABLE FACTURE
(code_fact Number PRIMARY
KEY,
id_client Number REFERENCES CLIENT(id_client)
...);
```

On indique le mot clé REFERENCES après la clé étrangère, puis le nom de la table qui est en relation suivi entre parenthèse de la clé primaire de cette table référencée.

L'application de l'intégrité référentielle implique plusieurs choses :

- on ne peut pas ajouter une facture avec un id_client qui n'existe pas dans la table client (sinon un message d'erreur apparaît). Il faut d'abord créer le client avant de créer sa facture.
- on ne peut pas supprimer un client s'il existe des factures qui lui correspondent. Il faut d'abord supprimer toutes les factures qui le concernent avant de pouvoir le supprimer
- on ne peut pas modifier l'id_client dans la table client s'il existe des factures qui le concernent.
- On peut modifier l'id_client de facture seulement si on choisit un id_client qui existe dans la table client.

Cela revient à vérifier que toutes les factures correspondent toujours à un client qui existe.

2.7.1 Options des contraintes d'intégrité référentielles

Effacer en cascade ON DELETE CASCADE

```
CREATE TABLE FACTURE
(code_fact Number
PRIMARY KEY,
id_client Number REFERENCES CLIENT(id_client) ON DELETE CASCADE
```

…);

Si on ajoute cette option à la contrainte d'intégrité référentielle, alors si on supprime une ligne de client, toutes les factures de ce client seront supprimées.

Attention : cette contrainte s'applique à la table facture ('enfant') mais elle est vérifiée lorsqu'on modifie la table client ('parent')

Modifier en cascade

Dans la norme SQL 2, il existe aussi l'option ON UPDATE CASCADE qui permet de répercuter sur toutes les lignes 'enfants' les modifications de valeur de la clé 'parent'.

2.7.2 Références à des colonnes multiples

Il arrive qu'une clé primaire, composée de plusieurs colonnes, soit elle-même clé étrangère d'une autre table. On indique alors cette clé étrangère composée après la définition des colonnes, grâce au mot clé FOREIGN KEY suivi des colonnes clés étrangères, puis de REFERENCES, le nom de la table référencée ainsi que le nom des colonnes référencées entre parenthèses.

Exemple :

CHAMBRE(id_hotel, num_ch, tarif)

RESERVATION(id_resa , #id_hotel, #num_ch, date)

```
CREATE TABLE
RESERVATION (id_resa
Number PRIMARY KEY,
id_hotel Number,
 num_ch Number,
 date Date NOT NULL,
 FOREIGN KEY (id_hotel, num_ch) REFERENCES CHAMBRE (id_hotel, num_ch)
```

);

2.7.3 Exemple de relation où la clé primaire est composée de plusieurs clés étrangères

```
CREATE TABLE LIGNE_COMMANDE
(id_commande Number REFERENCES
COMMANDE(id_commande), ref_produit Varchar2(10)
REFERENCES PRODUIT(ref_prod), quantité Number
DEFAULT 1,
PRIMARY KEY (id_commande, ref_produit)
);
```

Remarque : dans la table produit, la référence est codée par ref_prod et dans la table COMMANDE, c'est ref_produit

2.8 L'importance de l'intégrité référentielle

La gestion de l'intégrité référentielle est un critère primordial pour distinguer les SGBD relationnels et les autres. Un SGBD dépourvu de la gestion de l'intégrité référentielle (tel que MySQL jusqu'à la version
3.23) ne peut pas être qualifié de relationnel.

Lorsqu'on utilise de tels SGBD, il faut alors coder (programmer) les différentes vérifications à effectuer lors de la saisie, la suppression et la modification de données, ce qui est très contraignant et de plus peu sûr.

Une technique alternative aux contraintes d'intégrité, mais prise en charge par le SGBD est celle des triggers. Cette technique sera vue dans les prochains chapitres.

2.9 Ajout ou modification d'une contrainte

Pour ajouter une contrainte à une table existante, il suffit d'utiliser la commande ALTER TABLE

Syntaxe générale :

ALTER TABLE nom_de_la_table

ADD type_contrainte (colonne(s));

Exemple1 : on ajoute la contrainte d'intégrité référentielle dans la table COMMANDE. ALTER TABLE COMMANDE ADD FOREIGN KEY (id_client) REFERENCES CLIENT(id_client); Exemple 2 : on ajoute la contrainte de vérification de l'âge d'une personne

ALTER TABLE PERSONNE ADD CHECK (age <= 130);

Sur la table EMP(EmpNo, Enom, Job, DirNo, Sal, Prime), on peut imposer que ;

– les salaires doivent être positif, les primes doivent être supérieures ou égales à zéro.

ALTER TABLE EMP ADD CONSTRAINT sal_prime_pos CHECK (Sal > 0 AND Prime >= 0);

– le salaire d'un employé doit être supérieur que son prime, s'il en a:

ALTER TABLE EMP ADD CONSTRAINT sal_sup_prime CHECK (Sal > Prime);

2.10 Suppression de Contraintes

Lorsqu'une contrainte n'est plus valide pour l'application, on peut la supprimer:

Alter table DEPT drop unique(dname, loc);

Alter table EMP drop primary key, drop constraint dept_fkey;

Drop table EMP cascade constraints;

La dernière commande supprime la table et toutes les contraintes associés

directes ou indirectes.

10.11. Désactivation et activation de contraintes

ALTER TABLE <table> DISABLE CONSTRAINT <nom_contrainte>

ALTER TABLE <table> ENABLE CONSTRAINT <nom_contrainte>

Chapitre 4 : Architecture Client /Serveur

1. Introduction

C'est un concept de la technologie de l'information mais surtout, une autre manière de gérer l'information répartie au sein de l'organisation. C'est un concept sur lequel, les ordinateurs, reliés en réseau, travaillent mutuellement. Un ordinateur, le serveur exécute une tâche à la demande d'un autre ordinateur, le client, pendant que les deux ordinateurs gardent leur autonomie.

En principe, le client/serveur n'est autre chose qu'une répartition des tâches appliquées aux ordinateurs, ceci pour créer un environnement flexible, favorable à la croissance de l'organisation, à la demande de système de communication et à l'intégration. Le client/serveur découle de deux innovations fabuleuses : le micro-ordinateur avec ses capacités inouïes d'autonomie, et de l'interface graphique (le Gui : graphic user interface) dont la prolifération a chamboulé fondamentalement l'informatique. Le client/serveur est à la conjonction de trois mondes : celui de grandes bases de données (SGBD), des réseaux, et des interfaces utilisateurs graphiques.

2. Définitions du concept client /serveur

Plusieurs définitions parmi lesquelles :

Définition 1 : Une architecture client/serveur peut être définie comme une architecture logicielle ouverte (fidèle aux standards) qui fournit des services distants (base de données, messagerie, impression, etc...) à des clients interconnectés via un réseau, de manière transparente pour l'hétérogénéité des ressources informatiques mises en jeu (ordinateur, réseau, et logiciels de base)

Définition 2 : Le client /serveur est une architecture logicielle répartie, dans laquelle les systèmes sont divisés en procédures autonomes, permettant aux clients de solliciter des services qui sont exécutés par des serveurs. Dans cet environnement, l'utilisateur est plus flexible et peut manipuler les données dont il a besoin sur un système local.

Définition 3 : Le client/serveur est un concept dans lequel les tâches sont réparties entre clients et serveurs, permettant ainsi aux tâches d'être exécutées par des machines adaptées à une situation donnée.

- Les principaux composants du client / serveur

Le client/serveur se compose essentiellement de trois composants, à savoir : le client, le serveur, et le middleware (/)

Le client : est un système autonome (machine /logiciel) qui demande des services (requêtes) au serveur. Le client gère l'interface graphique des applications permettant à l'utilisateur de valider l'entrée des données, et peut même assurer une partie du traitement. Le client offre une ergonomie conviviale et dispose d'outils standards : tableurs, traitements de textes, ensembles d'outils statistiques, frontaux de base de données, le faisant ainsi passer du concept de terminal passif au concept de poste de travail.

Le serveur : est chargé de répondre aux requêtes formulées par les utilisateurs à partir de leur poste client : Le serveur héberge un S.G.B.D qui, dans la plupart des cas, est de type relationnel. Le client, dans ce cas, génère une requête SQL qui est ensuite envoyée, sous forme de message, au serveur de base de données. Celui-ci, à son tour, utilise ses capacités et processus, pour satisfaire la requête.

Le middleware (/) : Représente la barre oblique entre le client et le serveur. Il s'agit en fait, d'une couche logicielle située entre l'application et le réseau qui permet d'établir et de maintenir le dialogue entre l'application cliente et le serveur. Il est constitué d'une interface de programmation d'application appelée API, et d'un protocole réseau. L'A.P.I est une interface de programmation située aussi bien du côté de l'application client que du côté serveur.

Les principaux services attendus de la couche middleware sont :

- Le service de présentation de l'interface utilisateur

- Le service d'interrogation et de gestion de données
- Le service de communication sur le réseau dans l'échange des données
- Les services de conception et de programmation

Lorsque l'application cliente envoie un message, c'est cette couche logicielle qui le capte. Le middleware se charge alors de transmettre le message au serveur à travers le réseau en opérant les conversions de protocoles nécessaires, c'est lui également qui retournera le résultat à l'application cliente.

Ainsi, chaque éditeur de SGBD fournit un moyen de construire des applications client/serveur compatibles avec leur SGBD (souvent sous forme d'un API). On dit qu'il s'agit d'API propriétaires, parce que chaque API permet uniquement d'accéder à la base pour laquelle elle a été développée par l'éditeur du SGBD.

Exemple : OCI d'oracle, DB-lib de sybase, SQL/services de rdb

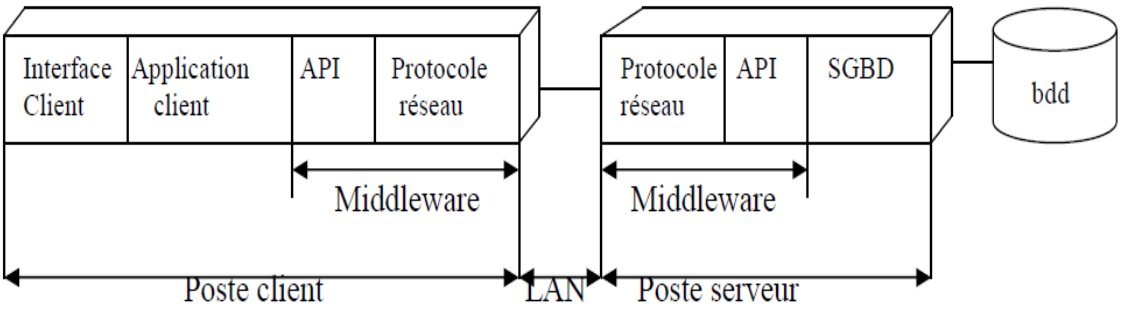


Fig 1 : Communication entre poste client et serveur de base de données

- Les architectures logicielles client/serveur

Pour définir simplement un modèle client /serveur le cabinet Gartner Group décompose une application en trois (3) niveaux conceptuels distincts :

- La présentation qui concerne tout ce qui a trait à l'interface graphique utilisateur

- La logique applicative qui désigne l'ensemble des traitements effectués par l'application

- La gestion des données concerne essentiellement l'accès aux données et le maintien de leur intégrité

A ces trois niveaux, il peut être associé trois niveaux d'outils de développement d'applications client/serveur, conceptuellement hétérogènes :

- La boite à outils graphiques, pour le niveau de *présentation*

- L'AGL ou le langage de développement (L3G, L4G) qui peut être fourni par l'éditeur de S.G.B.D, pour le niveau *application*

- S.Q.L qui permet de définir, manipuler et contrôler une base de données relationnelle, pour le niveau *gestion des données*

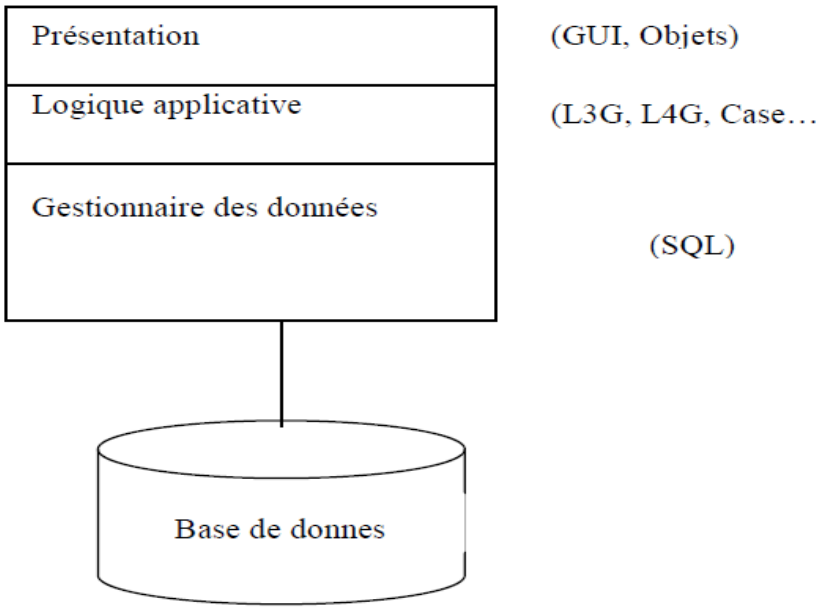


Fig2: Architecture fonctionnelle d'une application client/serveur et les outils de développement correspondants.

Dès lors, il est possible de construire différents types d'architectures client/serveur. On distingue trois types d'architectures client/serveur :

- Le un–tiers

- Le deux–tiers

- Le trois–tiers

Remarque : le tiers est le niveau qui sépare le client du serveur.

Il équivaut, en quelque sorte, au nombre de machines parcourues de la ‘’machine cliente’’ à la ‘’machine serveur’’

a) architecture ‘’un-tiers’’

Cette solution préconise la cohabitation des processus clients et serveurs sur la même machine physique, et leur communication est gérée par le middleware. Le client constitue l'interface avec l'utilisateur, et le serveur exécute le travail demandé Le un–tiers est surtout utilisé dans le cas des prototypes de démonstration ou lorsque les ressources matérielles sont limitées.

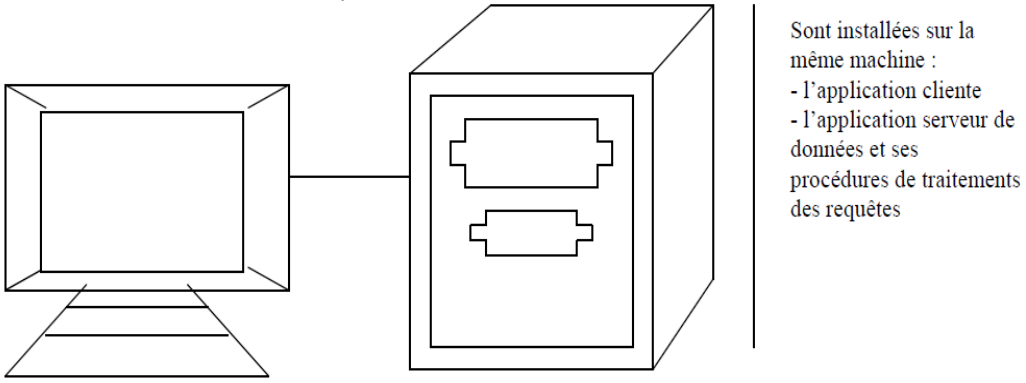


Fig. 3: Schémas architecture client/ serveur ‘’un-tiers’’

b) Architecture ‘’deux-tiers’’

Dans le ‘’deux-tiers’’, le client parle directement au serveur sans un poste intermédiaire, et attend la réponse à sa requête en restant connecté. C'est une architecture adaptée à un département (de moins de cinquante personnes) d'une administration.

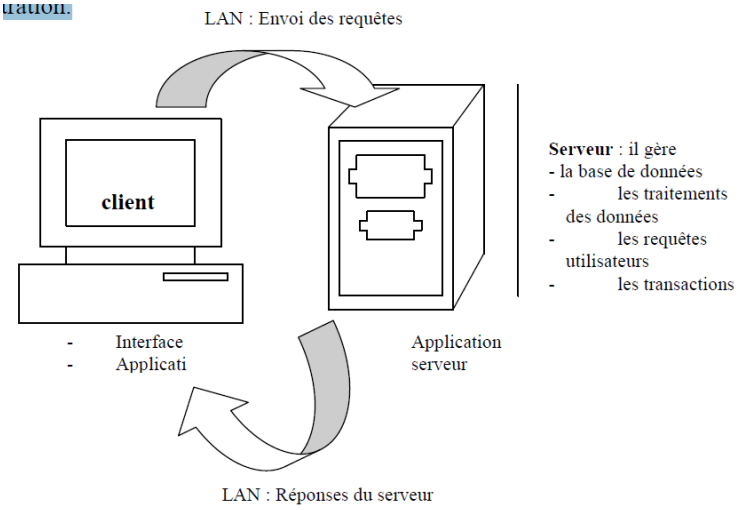


Fig. 4: Schémas architecture client/ serveur ''deux-tiers''

c) Architecture ''trois-tiers''

Préconise l'ajout d'un serveur intermédiaire appelé ''serveur de clients''. Il se charge de répartir une requête cliente auprès de différents serveurs, en collecter les résultats, et acheminer une seule réponse au client. Particulièrement adaptée à la logique répartie, cette architecture offre aux clients la possibilité de lancer leurs requêtes et de continuer de travailler sans être obligé d'attendre la réponse du serveur de clients. Ce dernier catalogue les résultats des diverses requêtes en attendant la demande de lecture par les clients concernés.

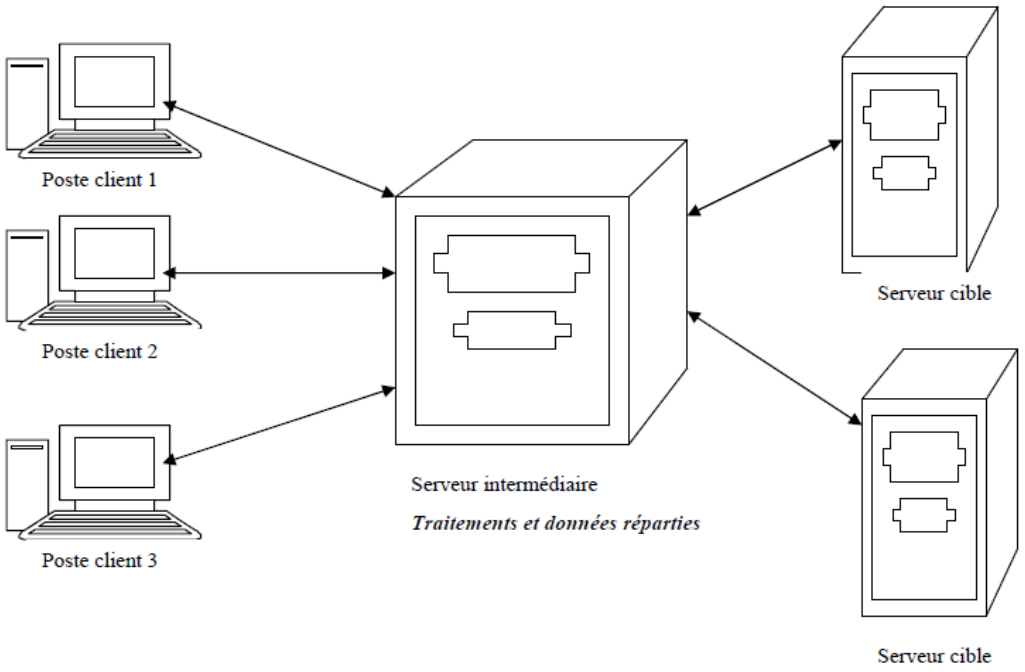


Fig 5 : Schéma Architecture client/serveur ''trois-tiers''

Chapitre 5 : Procédures, fonctions stockées et déclencheurs

1. Procédures et fonctions stockées¶

Les procédures stockées constituent une alternative à l'écriture de programmes avec un langage de programmation généraliste. Commençons par étudier plus en détail les avantages et inconvénients respectifs des deux solutions avant d'entrer dans les détails techniques.
Une procédure stockée s'exécute au sein du SGBD, ce qui évite les échanges réseaux qui sont nécessaires quand les mêmes fonctionnalités sont implantées dans un programme externe communiquant en mode client/serveur avec la base de données.
Le recours à une procédure stockée permet de regrouper du côté serveur l'ensemble des requêtes SQL et le traitement des données récupérées. La procédure est compilée une fois par le SGBD, au moment de sa création, ce qui permet de l'exécuter rapidement au moment de l'appel. De plus les échanges réseaux ne sont plus nécessaires puisque la logique de l'application est étroitement intégrée aux requêtes SQL. Le rôle du programme externe se limite alors à se connecter au serveur et à demander l'exécution de la procédure, en lui passant au besoin les paramètres nécessaires. Il existe malheureusement une contrepartie à tous ces avantages : chaque éditeur de SGBD propose sa propre extension procédurale pour créer des procédures stockées, ce qui rend ces procédures incompatibles d'un système à un autre. Cela peut être dissuasif si on souhaite produire un logiciel qui fonctionne avec tous les SGBD relationnels.
La description qui suit se base sur le langage PL/SQL d'Oracle ("PL" signifie *Procedural Language*) qui est sans doute le plus riche du genre.

2. Structures de contrôle

Le IF et le CASE fonctionnent de la même façon que dans les autres langages impératifs :

```
IF /*condition1*/ THEN
/ *instructions1*/
ELSE
/ *instructions2*/
END IF;
```

Voire

```
IF /*condition1*/
THEN
/ *instructions1*/
ELSIF /*condition2*/
/ *instructions2*/
ELSE
/ *instructions3*/
ENDIF
;
```

Les conditions sont les mêmes qu'en SQL. Le switch du langage C s'implémente en PL/SQL de la façon suivante :

```
CASE *variable*/
WHEN / *valeur1*/
THEN
/ *instructions1*/
WHEN / *valeur2*/
THEN
/ *instructions2*/
. . .
WHEN / *valeurn*/
THEN
/ *instructionsn*/
ELSE
/ *instructions par défaut*/
END CASE;
```

3. Les instructions repetitive

LOOP ... END LOOP : permet d'implémenter les boucles

```
LOOP
/*instruction*/
END LOOP ;
```

L'instruction EXIT WHEN permet de quitter une boucle.

```
LOOP
/*instruction*/
EXIT WHEN / * c o n d i t i o n */ ;
END LOOP ;
```

La boucle FOR existe aussi en PL/SQL :

```
FOR / *variables*/ IN / *inf*/ . . /*sup*/ LOOP
/*instruction*/
END LOOP ;
```

Ainsi que la boucle WHILE :

```
WHILE /*conditions*/ LOOP
/*instructions*/
END LOOP ;
```

4. Procédures

Syntaxe : On définit une procédure de la sorte
CREATE OR REPLACE PROCEDURE nom (/* paramètres */)
IS
/ declaration des variables locales /
BEGIN
/ instructions /
END ;
Les paramètres sont une simple liste de couples nom type. Par exemple, la procédure suivante a che un compte à rebours.
CREATE OR REPLACE PROCEDURE compteARebours (n NUMBER) IS
BEGIN
IF n >= 0 THEN DBMS_OUTPUT.PUT_LINE(n); compteARebours(n 1);
END IF;
END;
Invocation En PL/SQL, une procédure s'invoque tout simplement avec son nom. Mais sous SQL+, on doit utiliser le mot-clé CALL. Par exemple, on invoque le comptearebours sous SQL+ avec la commande CALL compteARebours(20) ou execute compteARebours(20) ;
Passage de paramètres, Oracle permet le passage de paramètres par

référence. Il existe trois types de passage de paramètres :
– IN : passage par valeur
– OUT : aucune valeur passée, sert de valeur de retour
– IN OUT : passage de paramètre par référence Par défaut, le passage de paramètre se fait de type IN .

```
CREATE OR REPLACE PROCEDURE incr (val IN OUT NUMBER) IS
BEGIN
val := val + 1;
END;
```

5. Fonctions

Syntaxe

On crée une nouvelle fonction de la façon suivante :

```
CREATE OR REPLACE FUNCTION nom_Function (/ parametres /)
RETURN type
IS
/ declaration des variables locales /
BEGIN
/ instructions /
END ;
```

L'instruction RETURN sert à retourner une valeur. Par exemple,

```
CREATE OR REPLACE FUNCTION module (a NUMBER , b NUMBER) RETURN
NUMBER IS BEGIN
IF a < b THEN RETURN a;
ELSE
RETURN module(a  b, b);
END IF;
END;
```

Invocation Tout comme les procédures, l'invocation des fonctions ne pose aucun problème en PL/SQL, par contre, sous SQL+, c'est quelque peu particulier. On passe par une pseudo-table nommée DUAL de la façon suivante : SELECT module (21, 12) FROM DUAL;

6. Triggers

6.1. Principe

Principe Un trigger est une procédure stockée qui se lance automatiquement lorsqu'un ´évènement se produit. Par ´évènement, on entend dans ce cours toute modification des données se trouvant dans les tables. On s'en sert pour contrôler ou appliquer des contraintes qu'il est impossible de formuler de façon déclarative. Classification Type d'évènement Lors de la création d'un trigger, il convient de préciser que les type d''évènements qui le déclenche. Nous réaliserons dans ce cours des triggers pour les ´évènements suivants : –INSERT – DELETE – UPDATE Moment de l''exécution On précise aussi si le trigger doit ˆêtre ´exécuté avant (BEFORE) ou après (AFTER) l´évènement. Evènements non atomiques Lors que l'on fait un DELETE ..., il y a une seule instruction, mais plusieurs lignes sont éjectées. Le trigger doit-il être exécuté pour chaque ligne affectée (FOR EACH ROW), ou seulement une fois pour toute l'instruction (STATEMENT)?
– un FOR EACH ROW TRIGGER est exécuté à chaque fois qu'une ligne est éjectée.
– un STATEMENT TRIGGER est exécutée à chaque fois qu'une instruction est lancée.

6.2. Création Syntaxe,

On déclare un trigger avec l'instruction suivante :

```
CREATE OR REPLACE TRIGGER nomtrigger
[BEFORE | AFTER]
[INSERT | DELETE | UPDATE] ON nomtable
[FOR EACH ROW | ]
DECLARE /*declarations */
BEGIN
/* instructions */
END ;
```

Par exemple:

```
SQL> CREATE OR REPLACE TRIGGER

pasDeDeleteDansClient

BEFORE DELETE ON CLIENT
BEGIN
```

```
RAISE_APPLICATION_ERROR(-20555, 'Va te faire ... ' );
END;
SQL> SELECT COUNT(*) FROM CLIENT ;
COUNT(*)
---------------
21
SQL> DELETE FROM CLIENT ;
DELETE FROM CLIENT * ERREUR `a la ligne 1 : ORA-20555: Va te faire ...
ORA-06512: à "SCOTT.PASDEDELETEDANSCLIENT" ,
ligne 2 ORA-04088: erreur lors d'exécution du déclencheur
'SCOTT.PASDEDELETEDANSCLIENT'
SQL> SELECT COUNT(*) FROM CLIENT ;
COUNT(*)
---------------
21
```

L'instruction RAISE APPLICATION ERROR (code, message) lève une exception sans nom portant un code et un message d'erreur message.

Vous remarquez que comme l'erreur a été levée avant la suppression, les données sont toujours présentes dans la table CLIENT. Le trigger a contrôlé une règle, et comme elle n''était pas respectée, il a lancé une erreur.

Combinaisons d´évènements : Il est possible, en séparant les types d'évènements par le mot-clé OR, de définir un trigger déclenché par plusieurs ´évènements. Les variables booléennes INSERTING, UPDATING et DELETING permettent d'identifier l'évènement qui a déclenché le trigger.

```
CREATE OR REPLACE TRIGGER afficheEvenement
BEFORE INSERT ORUPDATEOR DELETE ON CLIENT
FOR EACH ROW
BEGIN
IF INSERTING THEN DBMS_OUTPUT . PUT_LINE( 'Insertion dans CLIENT');
ELSIF UPDATING THEN DBMS_OUTPUT . PUT_LINE( 'Mise à jour dans CLIENT');
ELSE DBMS_OUTPUT . PUT_LINE( 'Suppression dans CLIENT' );
END IF;
END;
```

6.3. Accès aux lignes en cours de modification.

Dans les FOR EACH ROW triggers, il est possible avant la modification de chaque ligne, de lire l'ancienne ligne et la nouvelle ligne par l'intermédiaire des deux variables structurées **: old** et **: new**. Par exemple le trigger suivant empêche de diminuer un salaire :

```
CREATE OR REPLACE TRIGGER pasDeBaisseDeSalaire
BEFORE UPDATE ON EMP
FOR EACH ROW
BEGIN
        IF (: old.sal > :new.sal) THEN RAISE_APPLICATION_ERROR(-20567, 'Pas de
baisse de salaire ! ');
        END IF;
END;

SQL> CREATE OR REPLACE TRIGGER beforeStatement
BEFORE UPDATEON CLIENT
DECLARE
        NB NUMBER;
BEGIN
        SELECT COUNT(*) INTO NB FROM CLIENT;
END;

SQL> CREATE OR REPLACE TRIGGER afterStatement
AFTER UPDATE ON CLIENT
DECLARE
        NB NUMBER;
BEGIN
        SELECT COUNT(*) INTO NB FROM CLIENT;
END;
```

6.4. Suppression d'un déclencheur

```
SQL> DROPTRIGGER beforeForEachRow ;
```

Par exemple, si l'on souhaite empêcher un client d'avoir plus de 10 comptes en banque, une solution est de placer dans la table client une colonne contenant le nombre de comptes.

```
ALTER TABLE CLIENT ADD nbComptes number;
UPDATE CLIENT SET nbComptes = 0;
```

Une fois cette table créée, il convient de s'assurer que les données de la colonne nbComptes contiennent toujours les bonnes valeurs. On le fait avec plusieurs sous-programmes :

```
CREATE OR REPLACE TRIGGER metAJourNbComptes
AFTER INSERT OR UPDATE OR DELETE ON COMPTECLIENT
BEGIN UPDATE CLIENT SET nbComptes = ( SELECT COUNT(*) FROM
```

```
COMPTECLIENT CC WHERE CC.numCli = numCli );
END;

CREATE OR REPLACE TRIGGER verifieNbComptes
BEFORE INSERT ON COMPTECLIENT
FOR EACH ROW
DECLARE
        nbComptes NUMBER;
BEGIN
        SELECT nbComptes INTO nbComptes FROM CLIENT WHERE numCli = new.numcli;
        IF (nbComptes >= 10) THEN
                RAISE_APPLICATION_ERROR(-20556, 'Ce client a deja trop de comptes ');
        END IF;
END;
```