

BASE DE DONNÉES

Support de cours de base de données de la filière MMIC de l'IUT de Mesures Physiques d'Orsay
2008-2009

Table des matières

1 -Intérêts.....	2
1.1 -Organisation et cohérence des informations.....	2
1.2 -Traitements des données.....	2
1.3 -Utilisations.....	3
2 -MERISE.....	4
2.1 -Modèle conceptuel de données-MCD.....	4
2.2 -Modèle logique de données-MLD.....	17
2.3 -Modèle physique de données-MPD.....	22
2.4 -Conclusion.....	23
2.5 -Références.....	23
3 -Le langage SQL	24
3.1 -Concept :.....	25
3.2 -Fonctionnalités avancées.....	34
3.3 -Syntaxe SQL	37
4 -PgAdmin III.....	42
4.1 -Introduction.....	42
4.2 -Création de table.....	43
4.3 -Créer une clé étrangère.....	44
4.4 -Insérer les données dans les tables.....	44
4.5 -Requêtes.....	45
5 -Annexe.....	47
5.1 -Commandes SQL	47
5.2 -Applications client de PostgreSQL	50
5.3 -Applications relatives au serveur PostgreSQL	50

1 - Intérêts

Dans de nombreuses entreprises, associations, sites, vous pouvez trouver des bases de données. Pourquoi un tel engouement?

1.1 - Organisation et cohérence des informations

1.1.1 - Éviter la répétition d'une même information

Dans un inventaire, plusieurs articles proviennent du même fournisseur; une base de données permet d'éviter de répéter le nom du fournisseur pour chaque article. De plus, pour chaque achat d'un même client, inutile de répéter toutes les informations liées au client, seul un identifiant est nécessaire. De même dans une gestion de bibliothèque, plusieurs livres ont le même auteur, une base de données permet d'éviter de répéter toutes les informations sur l'auteur pour chaque livre.

1.1.2 - Éviter les problèmes de graphie

Une base de données permet d'homogénéité l'orthographe des noms : traits d'union ou pas, majuscules ou pas, accent ou pas. Par exemple, est-ce que St NAZAIRE est la même ville que Saint Nazaire ou que Saint-Nazaire? Est-ce que Jean-Jacques ROUSSEAU est bien le même que Jean Jacques ROUSSEAU ou JJ ROUSSEAU?

1.2 - Traitements des données

Une base de données permet des tris, des regroupements et des classements de toutes sortes. Dans une bibliothèque, nous pouvons par exemple souhaiter connaître :

- tous les livres empruntés le mois dernier,
- tous les livres empruntés le mois dernier, par des femmes,
- tous les livres empruntés le mois dernier, par les hommes de plus de 50 ans ayant plus de cinq ans d'adhésion à la bibliothèque,
- tous les livres empruntés le mois dernier par les femmes de moins de 25 ans ayant plus de deux ans d'adhésion à la bibliothèque et n'ayant jamais rendu de livre en retard,
- etc.

Pour répondre à ces questions nous utilisons des **requêtes**. Parmi toutes les informations stockées, une requête permet de récupérer celles qui correspondent à certains critères.

1.3 - Utilisations

Pour créer une base de données, nous utilisons couramment la méthode **Merise**. C'est une méthode d'analyse, de conception et de gestion de projet.

Les Systèmes de Gestion de Base de Données (SGBD) sont apparus dans les années 60. Ils permettent de créer des bases de données, de mettre à jour leurs données, d'y rechercher des informations particulières et de les visualiser.

Il existe de nombreux SGBD : PostgreSQL, Oracle, Sybase, Access, MySQL,...

Les SGBD utilisent généralement le langage SQL.

Les bases de données sont des outils couramment utilisés :

- gestion de réservation de billets, d'achat sur internet, ...
- gestion des résultats d'expériences, ...
- gestion d'emploi du temps, gestion du personnel, ...
- gestion d'une médiathèque, d'une vidéothèque, ...
- gestion de vos DVD, CD, ...
- ...

2 - MERISE

Cette section regroupe les notions de base concernant le modèle entité-association, le schéma relationnel et la traduction de l'un vers l'autre.

Quand nous construisons directement les tables d'une base de données dans un logiciel de gestion des bases de données (PostgreSQL, Oracle, SQL Server, DB2, base, Access, MySQL, ...), nous sommes exposés à deux types de problème :

- nous ne savons pas toujours dans quelle table placer certaines colonnes (par exemple, l'adresse de livraison se met dans la table des clients ou dans la table des commandes ?) ;
- nous avons du mal à prévoir les tables de jonction intermédiaires (par exemple, la table des interprétations qui est indispensable entre les tables des films et la table des acteurs).

Il est donc nécessaire de recourir à une étape préliminaire de conception.

Les techniques présentées ici font partie de la méthodologie Merise (Méthode d'Étude et de Réalisation Informatique pour les Systèmes d'Entreprise) élaborée en France en 1978 (cf. [4]) qui permet notamment de concevoir un système d'information d'une façon standardisée et méthodique.

Le but de ce support de cours est d'introduire le schéma entité-associations (Section 2.1), le schéma relationnel (Sections 2.2 et 2.3) et d'expliquer la traduction entre les deux (Section 2.2.3).

2.1 - Modèle conceptuel de données-MCD

Avant de réfléchir au schéma relationnel d'une application, il est bon de modéliser la problématique à traiter d'un point de vue conceptuel et indépendamment du logiciel utilisé. C'est le but de cette partie.

2.1.1 - Dictionnaire des données

Il est utile, voir indispensable, d'effectuer un inventaire brut des données qui seront stockées et/ou calculées par la base de données. Il faut donc différencier deux types de données, les données stockées et les données qui peuvent être déduites ou calculés à partir des données stockées. Ce document sera présenté sous forme de tableau de 3 colonnes, avec dans la première colonne le nom de la donnée, dans la seconde si cette donnée est calculée ou stockée et dans la troisième un commentaire descriptif de la donnée.

2.1.2 - Schéma entité-association

Une **entité** est une population d'individus n'ayant que des caractéristiques comparables. Par exemple, dans une entreprise qui achète et vend des produits, nous avons l'entité *client*, l'entité *article* et l'entité *fournisseurs* (voir illustration 1).

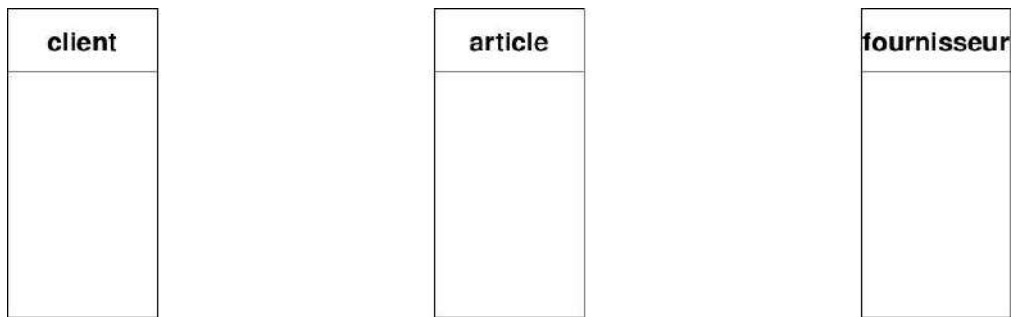


Illustration 1: Exemple d'entités

Une **association** est un lien entre plusieurs entités. Dans notre exemple illustration 2, l'association *acheter* fait le lien entre les entités articles et clients, tandis que l'association *livrer* fait le lien entre les entités articles et fournisseurs. Nous avons souvent intérêt à distinguer deux verbes pour décrire l'association, l'un dans un sens l'autre dans l'autre sens. . .

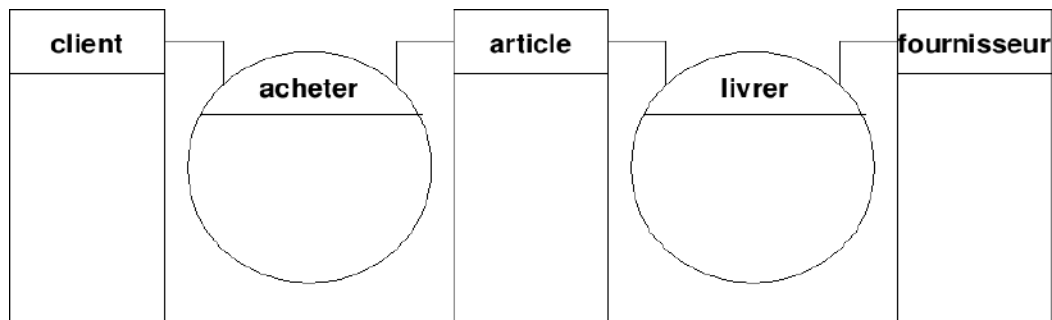


Illustration 2: Exemple d'associations

Un **attribut** est une propriété d'une entité ou d'une association (voir illustration 3). Toujours dans notre exemple, le *prix unitaire* est un attribut de l'entité article, le *nom du client* est un attribut de l'entité client. La *quantité commandée* est un attribut de l'association acheter, la *date de livraison* est un attribut de l'association livrer.

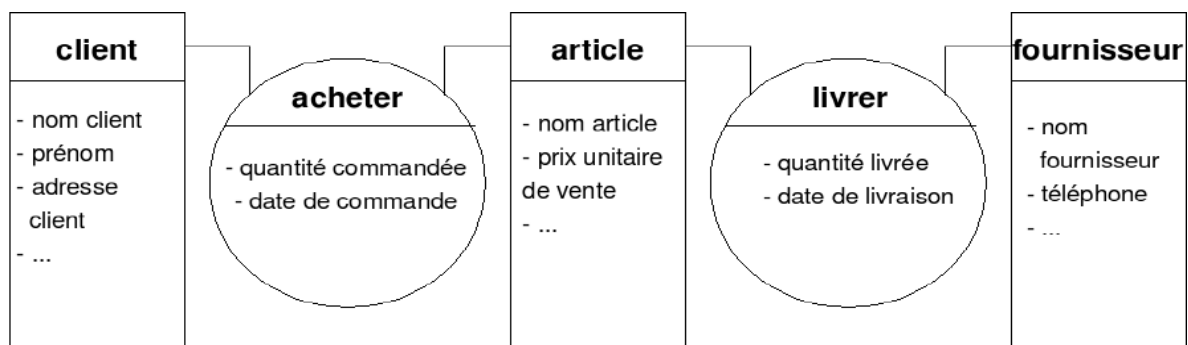


Illustration 3: Exemples d'attributs

Chaque individu d'une entité doit être identifiable de manière unique. C'est pourquoi les entités doivent posséder un attribut sans doublon : **l'identifiant**. Le *numéro de client* est l'identifiant de l'entité client (nous soulignons cet attribut sur le schéma, voir l'illustration 4).

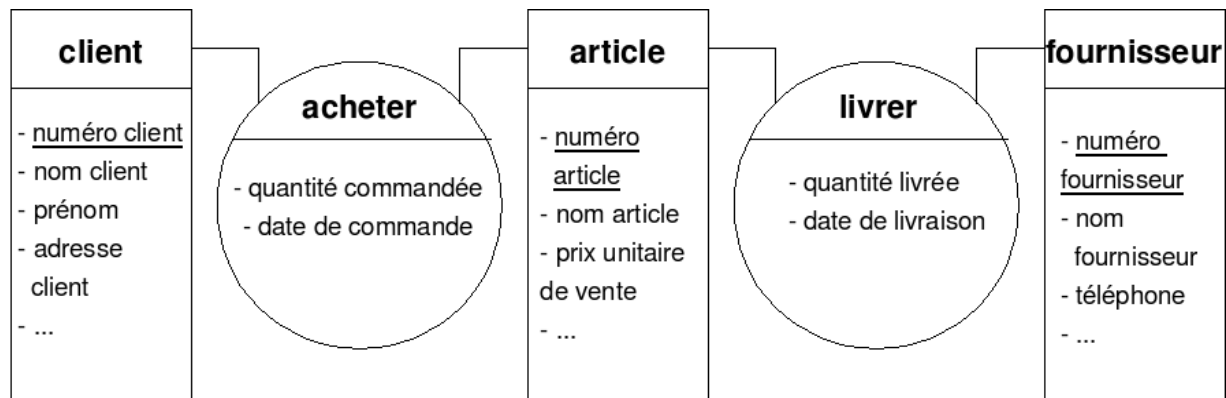


Illustration 4: Exemples d'identifiants

Remarques :

- un attribut ne doit pas figurer dans deux entités ou associations différentes (donc il faut spécialiser l'attribut *nom* en *nom du client*, *nom du produit* et *nom du fournisseur*) ;
- une entité possède au moins un attribut (son identifiant) ;
- une association peut ne pas posséder d'attribut.

La **cardinalité** d'un lien entre une entité et une association est le minimum et le maximum de fois qu'un individu de l'entité peut être concerné par l'association. Un client a au moins acheté un article et peut acheter n articles (n étant indéterminisé), tandis qu'un article peut avoir été acheté entre 0 et n fois (même si ce n'est pas le même n) et n'est livré que par 1 fournisseur. Nous obtenons alors le **schéma entité-association complet** de l'illustration 5.

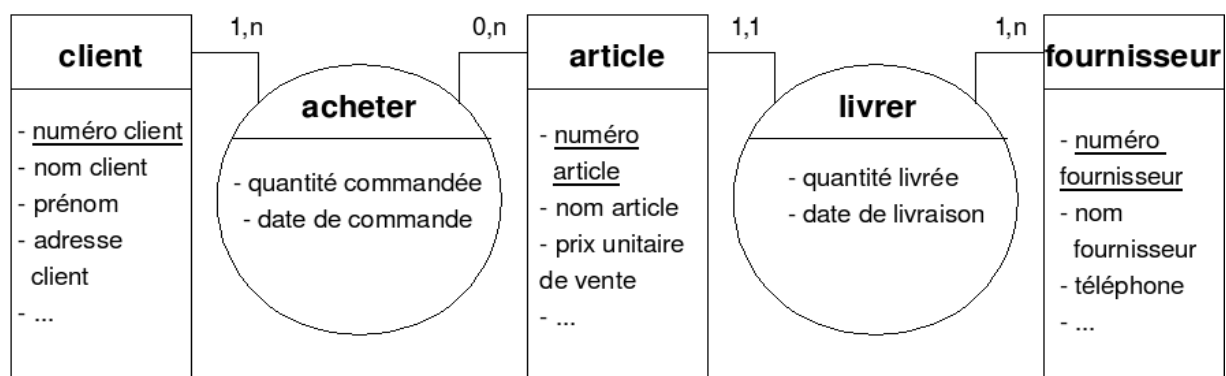


Illustration 5: Exemple de cardinalités

Remarque : Si une cardinalité est connue et vaut 2, 3, etc. on considère qu'elle est indéterminisé (n), de telle sorte que nous ne puissions avoir que des cardinalités 0, 1 ou n.

2.1.3 - Cas particuliers

Exemple d'**association binaire réflexive** : dans l'illustration 6, un employé est dirigé par un employé (sauf le directeur général) et un employé peut diriger plusieurs employés. Pour distinguer les deux liaisons, nous utilisons la notion de rôle avec l'ajout d'étiquettes (ici *supérieur*, *personnel*).

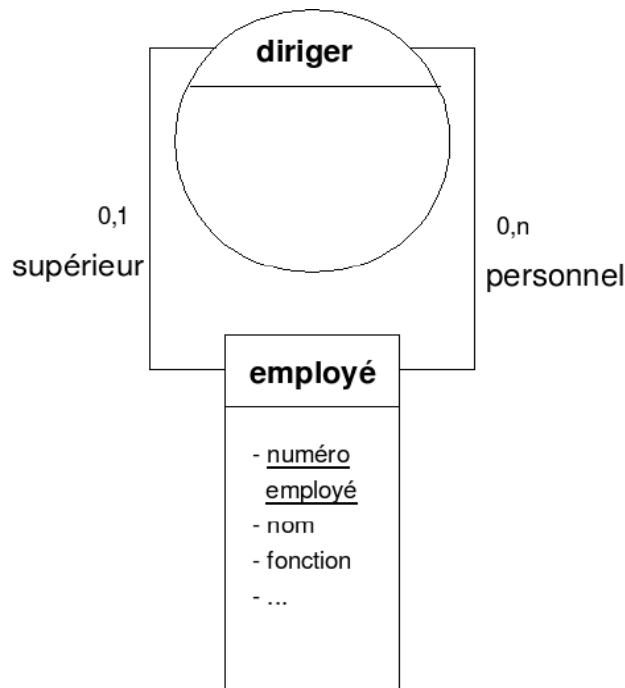


Illustration 6: Exemple d'association réflexive

Exemple d'association entre trois entités (**association ternaire**) : dans ce cas, un avion, un pilote ou un aéroport peuvent être utilisés 0 ou plusieurs fois par l'ensemble des vols (d'où les cardinalités) ; voir les illustrations 7 et 8.

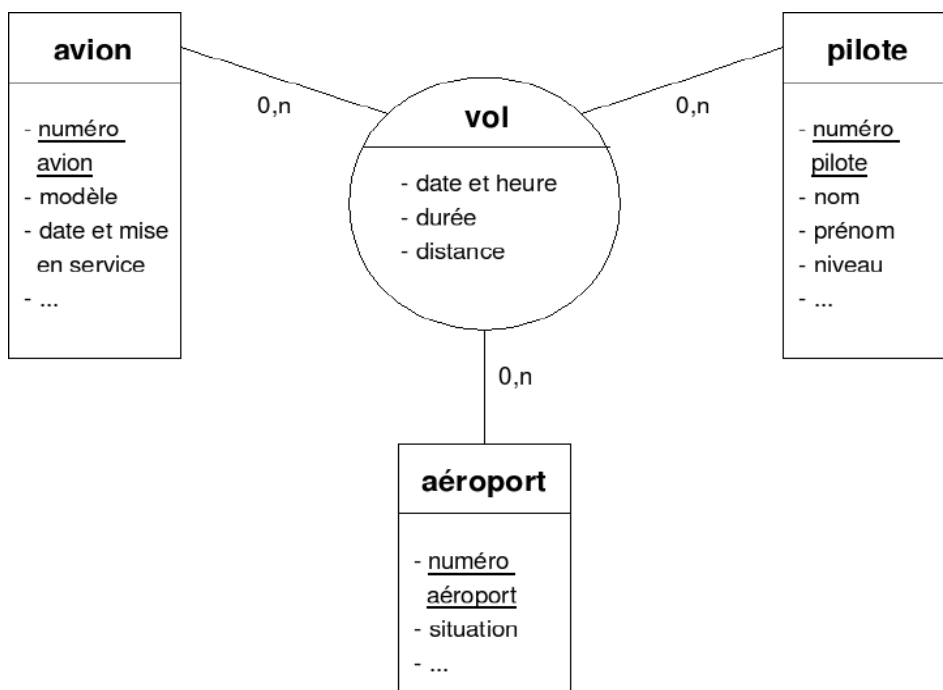


Illustration 7: Exemple d'association ternaire

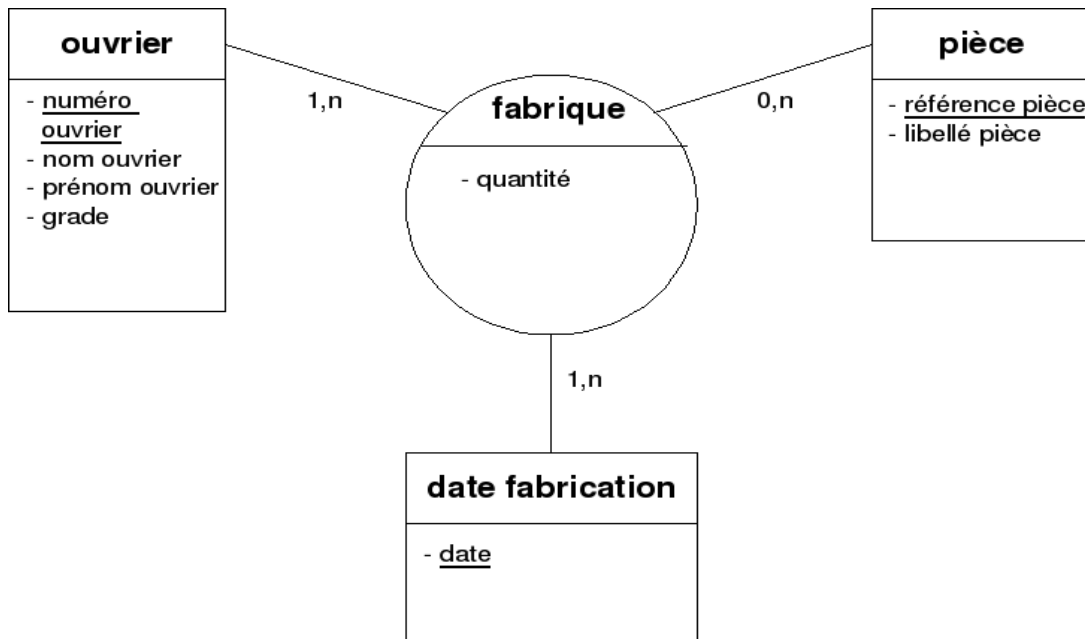


Illustration 8: Deuxième exemple d'association ternaire. Pour une pièce donnée, nous avons de 0 à n couples (ouvrier, date fabrication) différents.

Exemple de **plusieurs associations entre deux mêmes entités** : dans ce cas, un être humain peut être propriétaire, locataire et/ou chargé de l'entretien. Nous supposons qu'un être humain ne loue qu'un logement au maximum, qu'un logement n'est occupé que par une personne au maximum et qu'un logement est entretenu par une et une seule personne ; voir l'illustration 9.

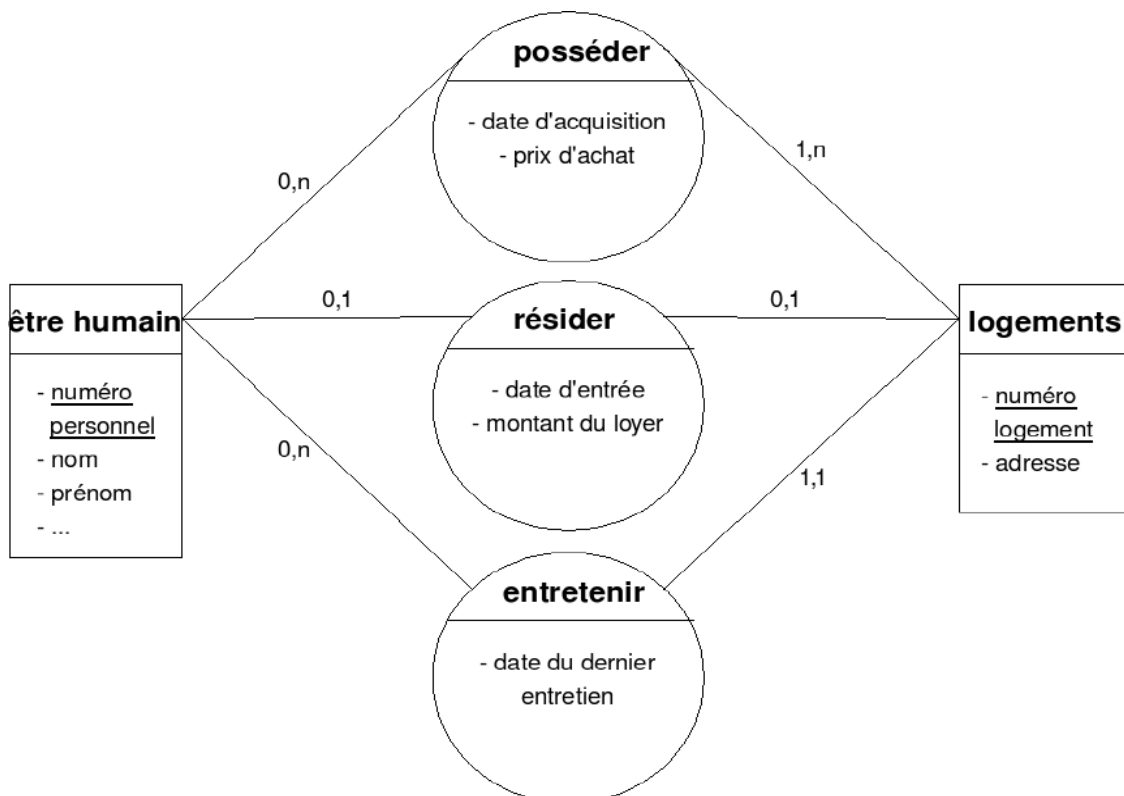


Illustration 9: Exemple d'associations plurielles

2.1.4 - Règles de normalisation

Un bon schéma entité-associations doit répondre à différentes règles.

Normalisation des entités

Toutes les entités qui sont remplaçables par une association doivent être remplacées (voir l'illustration 10).

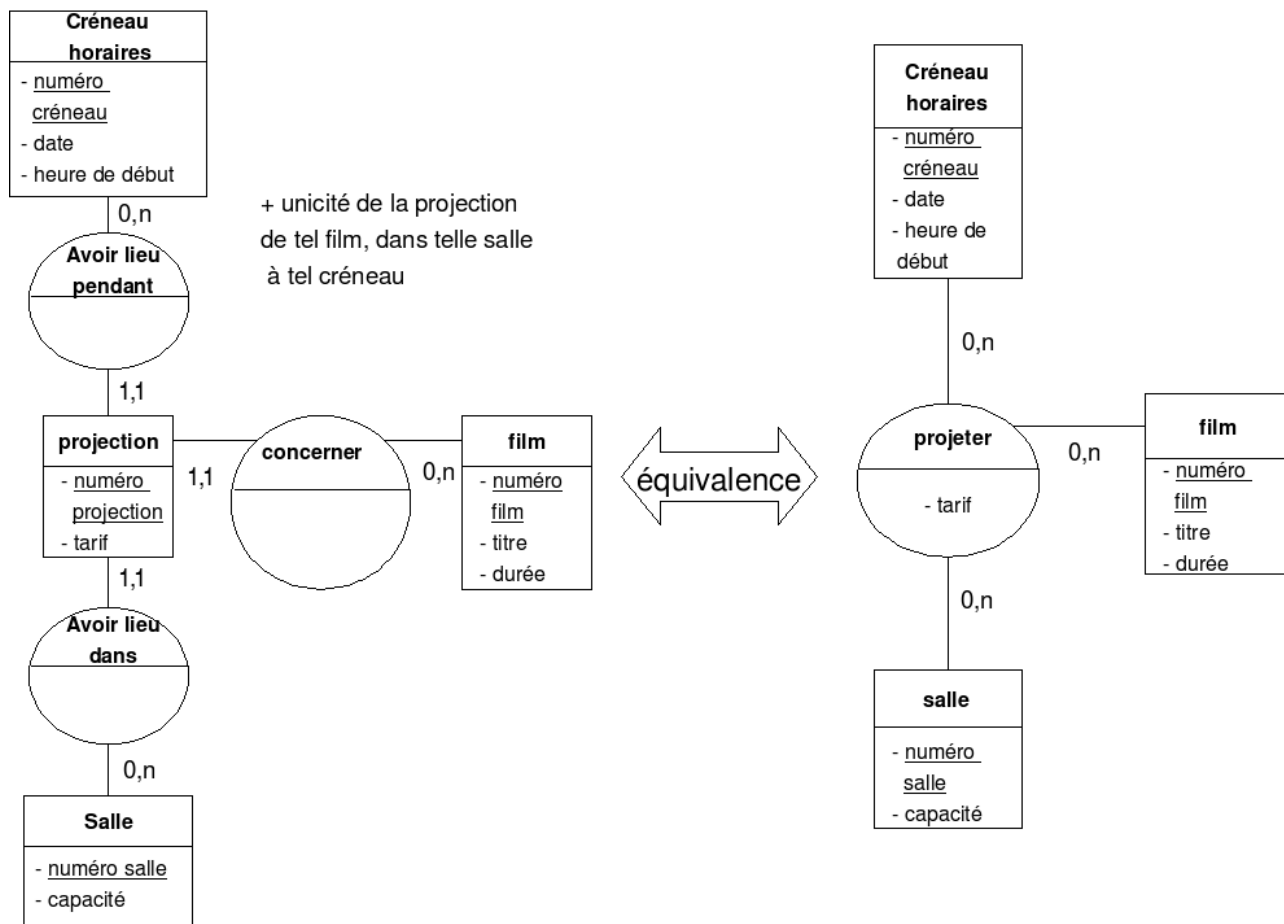


Illustration 10: Entité remplaçable par une association ternaire

Normalisation des noms

Le nom d'une entités d'une association ou d'un attribut doit être unique.

Remarque : lorsqu'il reste plusieurs fois le même nom, c'est parfois symptomatique d'une modélisation qui n'est pas terminée (voir l'illustration 11) ou le signe d'une redondance (voir l'illustration 12).

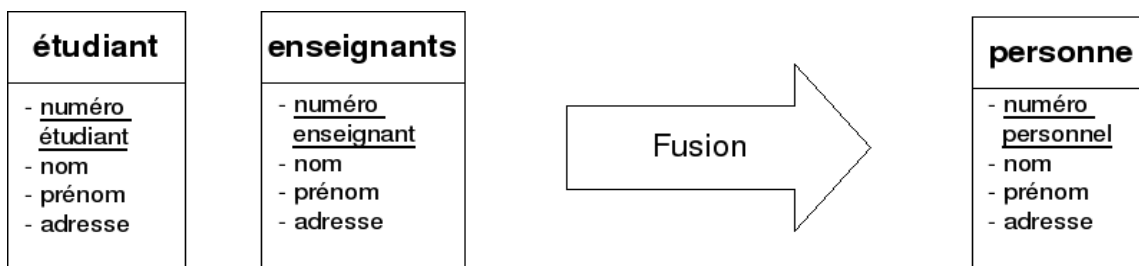


Illustration 11: Deux entités homogènes peuvent être fusionnées

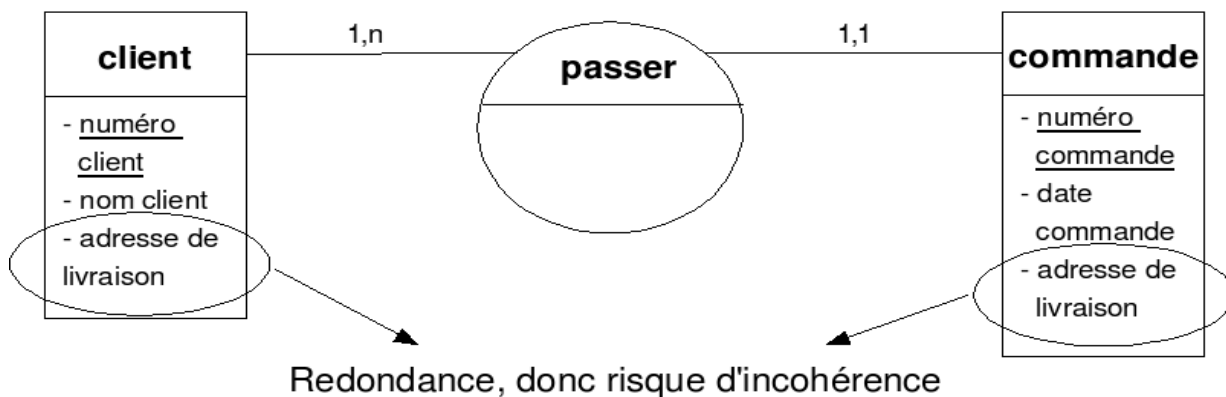


Illustration 12: Si deux attributs contiennent les mêmes informations, alors la redondance induit non seulement un gaspillage d'espace mais également un grand risque d'incohérence : ici, les adresses risquent de ne pas être les mêmes et dans ces conditions où faut-il livrer ?

Normalisation des identifiants

Chaque entité doit posséder un identifiant.

Conseils :

- éviter les identifiants composés de plusieurs attributs (comme par exemple un identifiant formé par les attributs nom et prénom), car d'une part c'est mauvais pour les performances et d'autre part, l'unicité supposée par une telle démarche finit tôt ou tard par être démentie ;
- préférer un identifiant court pour rendre la recherche la plus rapide possible (éviter notamment les chaînes de caractères comme un numéro de plaque d'immatriculation, un numéro de sécurité sociale ou un code postal) ;
- éviter également les identifiants susceptibles de changer au cours du temps (comme les plaques d'immatriculation ou les numéros de sécurité sociale provisoires).

Conclusion : l'identifiant sur un schéma entité-associations (et donc la future clé primaire dans le schéma relationnel) doit être un entier, de préférence incrémenté automatiquement.

Normalisation des attributs

Il faut remplacer les attributs en plusieurs exemplaires en une association supplémentaire de cardinalités maximales n et il ne faut pas ajouter d'attribut calculable à partir d'autres attributs.

En effet, d'une part, les attributs en plusieurs exemplaires posent des problèmes d'évolutivité du modèle (voir l'illustration 13 à gauche, comment faire si un employé a deux adresses secondaires ?) et d'autre part, les attributs calculables induisent un risque d'incohérence entre les valeurs des attributs de base et celles des attributs calculés (voir l'illustration 14).

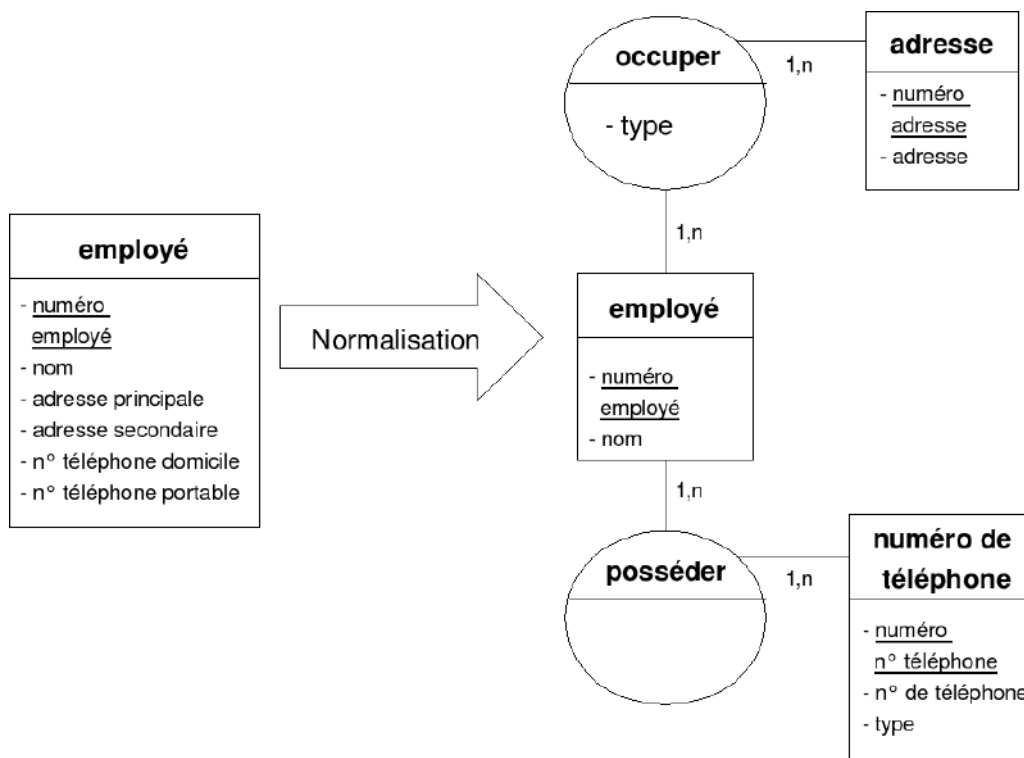


Illustration 13: Attributs en plusieurs exemplaires remplacés par une association supplémentaire

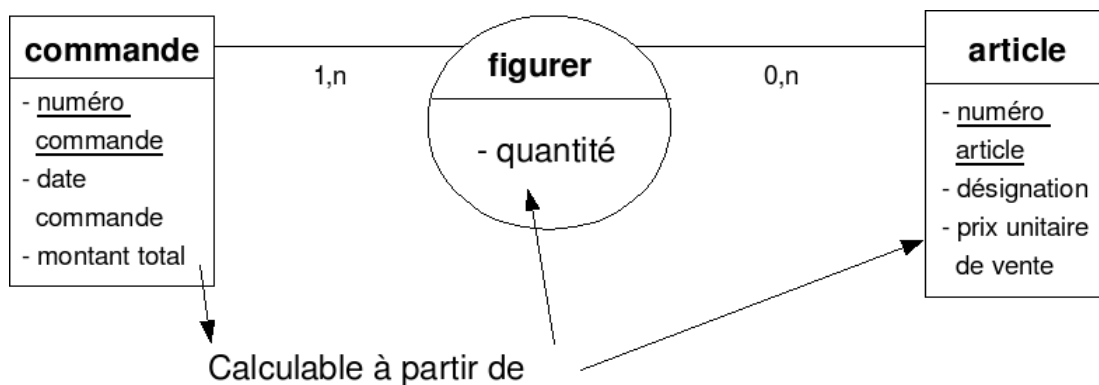


Illustration 14: Attribut calculable qu'il faut retirer du schéma

D'autres attributs calculables classiques sont à éviter, comme l'âge (qui est calculable à partir de la date de naissance) ou encore le département (calculable à partir d'une sous-chaîne du code postal).

Normalisation des attributs des associations

Les attributs d'une association doivent dépendre directement des identifiants de toutes les entités en association.

Par exemple, sur l'illustration 5 la quantité commandée dépend à la fois du numéro de client et du numéro d'article, par contre la date de commande non. En effet la date dépend du client et non d'un des articles commandés. Il faut donc faire une entité commandes à part, idem pour les livraisons (voir l'illustration 15).

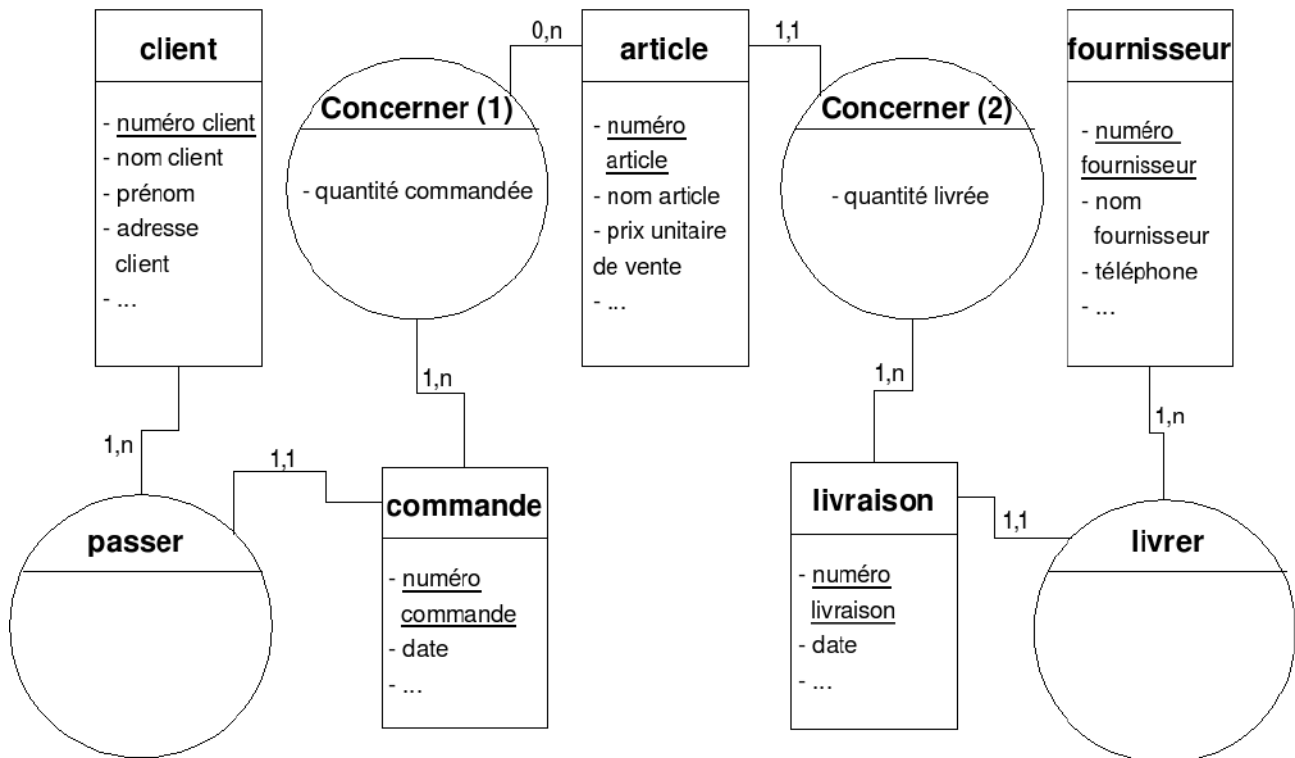


Illustration 15: Normalisation des attributs des associations

L'inconvénient de cette règle de normalisation est qu'elle est difficile à appliquer pour les associations qui ne possèdent pas d'attribut. Pour vérifier malgré tout qu'une association sans attribut est bien normalisée, nous pouvons donner temporairement à cette association un attribut imaginaire (mais pertinent) qui permet de vérifier la règle.

Par exemple, entre les entités livres et auteurs de l'illustration 20, l'association écrire ne possède pas d'attribut. Imaginons que nous ajoutions un attribut pourcentage qui contient le pourcentage du livre écrit par chaque auteur (du même livre). Comme cet attribut pourcentage dépend à la fois du numéro de livre et du numéro d'auteur, l'association écrire est bien normalisée.

Autre conséquence de la normalisation des attributs des associations : une entité avec une cardinalité de 1,1 ou 0,1 aspire les attributs de l'association (voir l'illustration 16).

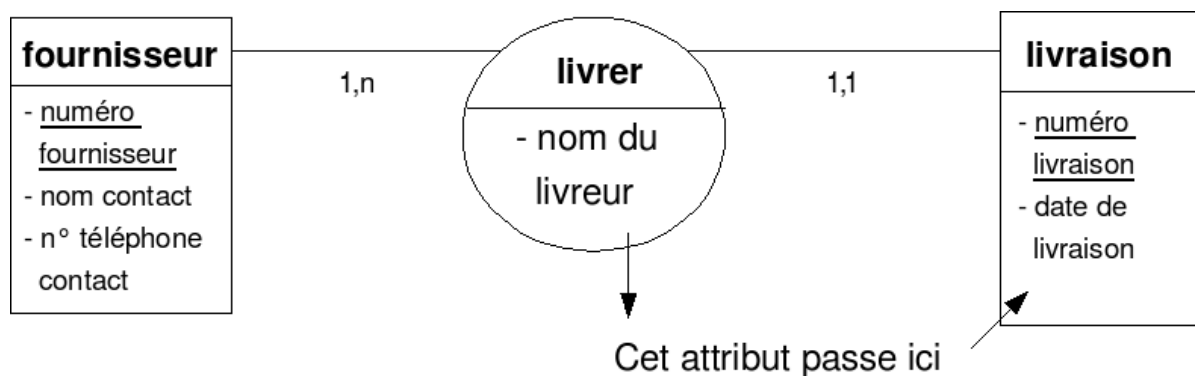


Illustration 16: Cardinalité 1,1 et attributs d'une association

Normalisation des associations

Il faut éliminer les associations **fantômes** (voir l'illustration 17), **redondantes** (voir l'illustration 18) ou en **plusieurs exemplaires** (voir l'illustration 19).

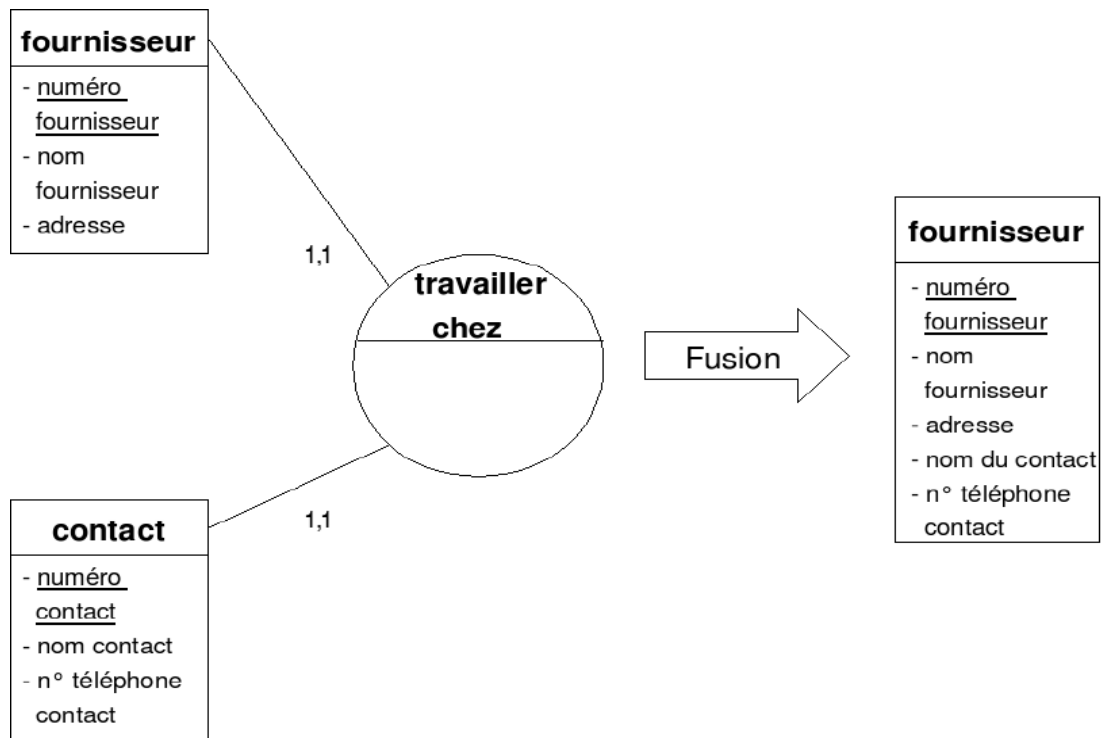


Illustration 17: Les cardinalités sont toutes 1,1 donc c'est une association fantôme

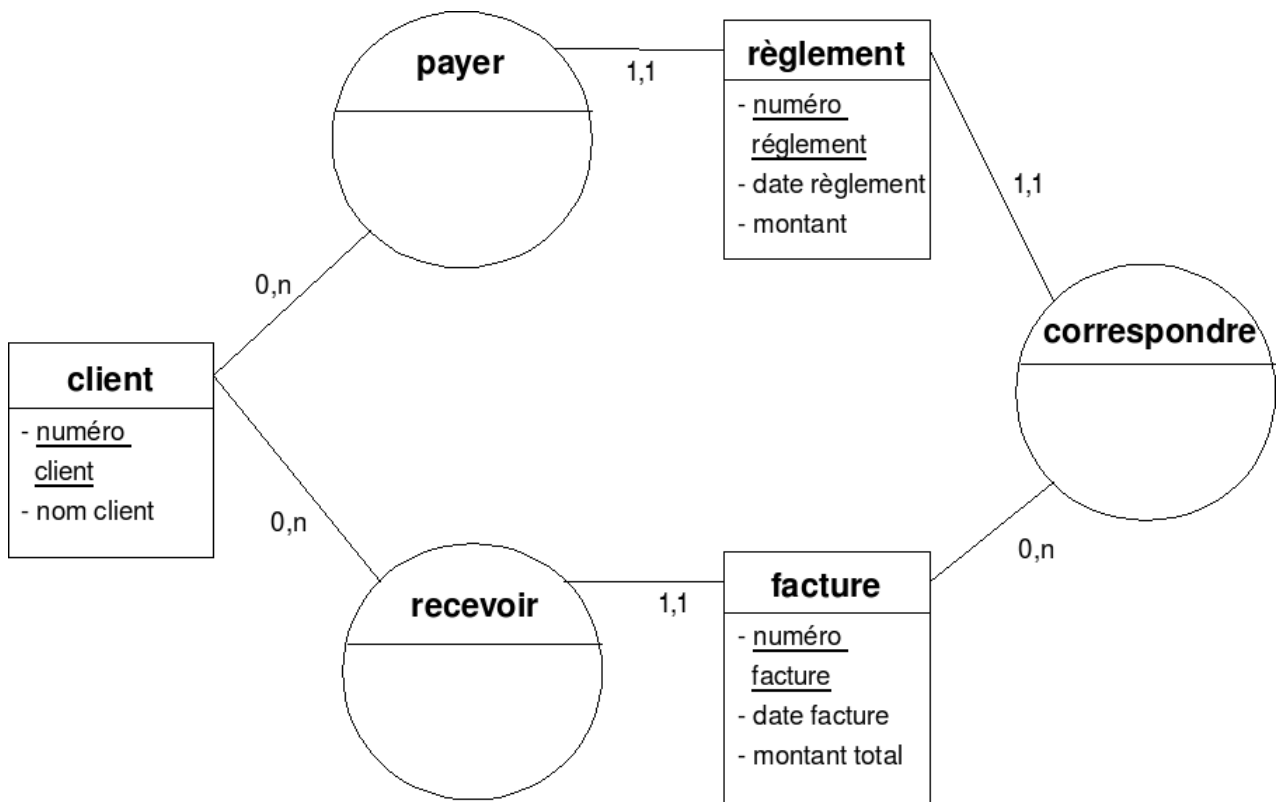


Illustration 18: Si un client ne peut pas régler la facture d'un autre client, alors l'association payer est inutile et doit être supprimée (dans le cas contraire, l'association payer doit être maintenue)

En ce qui concerne les associations redondantes, cela signifie que s'il existe deux chemins pour se rendre d'une entité à une autre, alors ils doivent avoir deux significations ou deux durées de vie différente. Sinon, il faut supprimer le chemin le plus court, car il est déductible à partir de l'autre chemin. Dans notre exemple de l'illustration 18, si nous supprimons l'association payer, nous pouvons retrouver le client qui a payer le règlement en passant par la facture qui correspond.

Remarque : une autre solution pour le problème de l'illustration 18 consiste à retirer l'entité règlement et d'ajouter une association régler avec les mêmes attributs sauf l'identifiant) entre les entités client et facture.

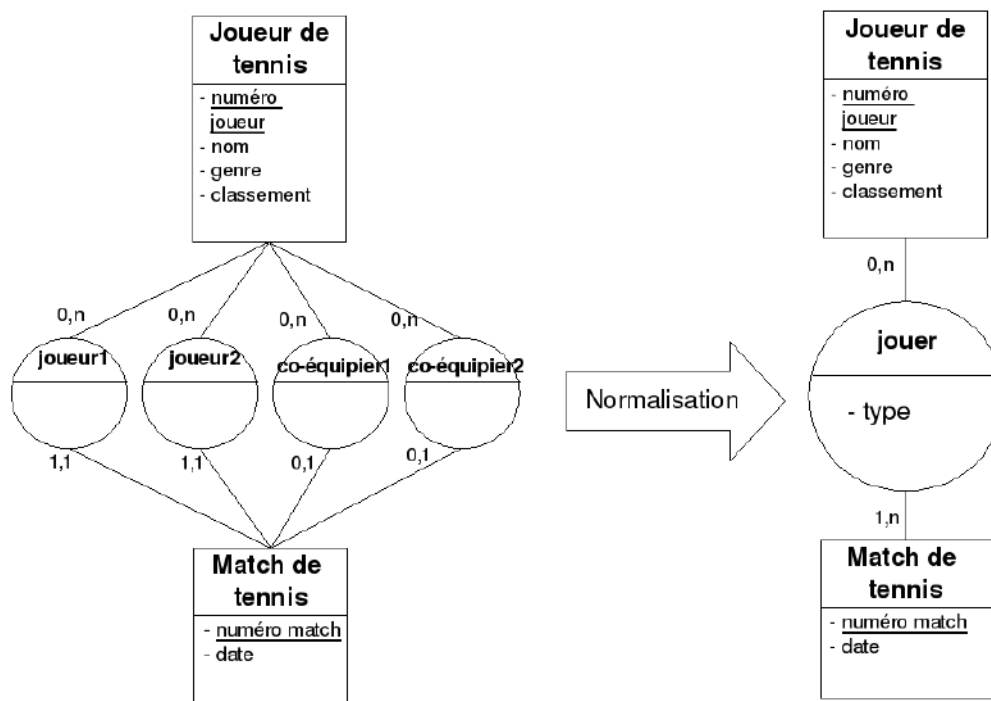


Illustration 19: Une association suffit pour remplacer les 4 associations

Normalisation des cardinalités

Une cardinalité minimale est toujours 0 ou 1 (et pas 2, 3 ou n) et une cardinalité maximale est toujours 1 ou n (et pas 2, 3...).

Cela signifie que si une cardinalité maximale est connue et vaut 2, 3 ou plus (un nombre limité d'emprunts dans une bibliothèque par exemple), alors nous considérons quand même qu'elle est indéterminisé, et vaut n. Cela se justifie par le fait que même si nous connaissons n au moment de la conception, il se peut que cette valeur évolue au cours du temps. Il vaut donc mieux considérer n comme une inconnue dès le départ.

Cela signifie également que nous ne modélisons pas les cardinalités minimales qui valent plus de 1 car ce genre de valeur est aussi mené à évoluer. Par ailleurs, avec une cardinalité maximale de 0, l'association n'aurait aucune signification.

A ces 7 règles de normalisation, il convient d'ajouter les 3 premières formes normales traditionnellement énoncées pour les schémas relationnels, mais qui trouvent toute aussi bien leur place en ce qui concerne les schémas entité-associations.

Première forme normale

A un instant donné, dans une entité, pour un individu, un attribut ne peut prendre qu'une valeur et non pas un ensemble ou une liste de valeurs.

Si un attribut prend plusieurs valeurs, alors ces valeurs doivent faire l'objet d'une entité supplémentaire, en association avec la première (voir l'illustration 20).

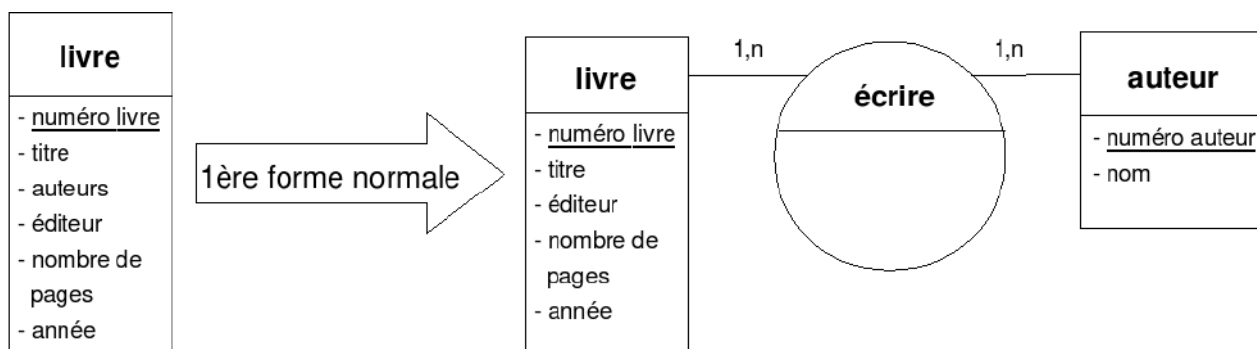


Illustration 20: Application de la première forme normale : il peut y avoir plusieurs auteurs pour un livre donné.

Deuxième forme normale

L'identifiant peut être composé de plusieurs attributs mais les autres attributs de l'entité doivent dépendre de l'identifiant en entier (et non pas d'une partie de cet identifiant).

Cette deuxième forme normale peut être oubliée si nous suivons le conseil de n'utiliser que des identifiants non composés et de type entier. En vérité elle a été vidée de sa substance par la règle de normalisation des attributs des associations.

Considérons malgré tout le contre-exemple suivant : dans une entité clients dont l'identifiant est composé des attributs nom et prénom, la date de fête d'un client ne dépend pas de son identifiant en entier mais seulement de prénom. Elle ne doit pas figurer dans l'entité clients, il faut donc faire une entité calendrier à part, en association avec l'entité clients.

Troisième forme normale de Boyce-Codd

Tous les attributs d'une entité doivent dépendre directement de son identifiant et d'aucun autre attribut.

Si ce n'est pas le cas, il faut placer l'attribut pathologique dans une entité séparée, mais en association avec la première.

Par exemple, l'entité avion (l'illustration 21 à gauche) dont les valeurs sont données dans le tableau 1, n'est pas en troisième forme normale de Boyce-Codd, car la capacité et le constructeur d'un avion ne dépendant pas du numéro d'avion mais de son modèle. La solution normalisée est donné par l'illustration 21 à droite.

Numéro de l'avion	Constructeur	Modèle	Capacité	Propriétaire
1	Airbus	A380	180	Air France
2	Boeing	B747	314	British Airways
3	Airbus	A380	180	KLM

Tableau 1: Il y a redondance (et donc risque d'incohérence) dans les colonnes constructeur et modèle

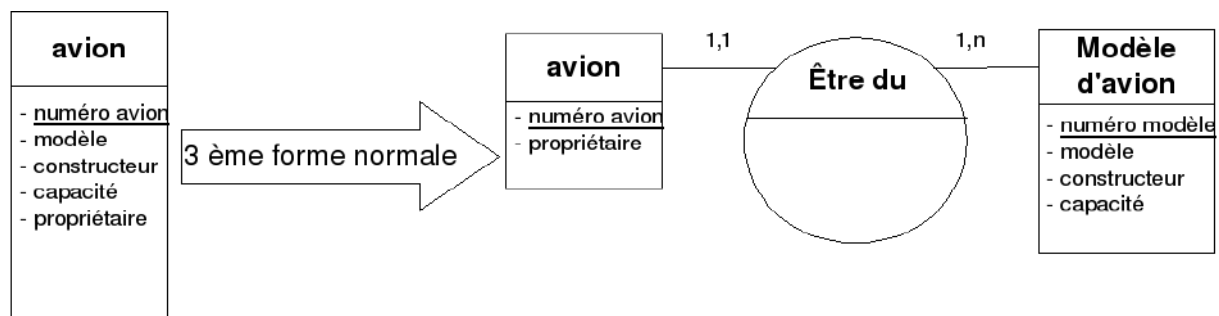


Illustration 21: Application de la troisième forme normal

2.1.5 - Méthodologie

Face à un problème procéder ainsi :

1. identifier les entités en présence ;
2. lister leurs attributs ;
3. ajouter les identifiants (numéro arbitraire et auto-incrémenté) ;
4. établir les associations binaires entre les entités ;
5. lister leurs attributs ;
6. calculer les cardinalités ;
7. vérifier les règles de normalisation et en particulier, la normalisation des entités (c'est à ce stade qu'apparaissent les associations non binaires), des associations et de leurs attributs ainsi que la troisième forme normale de Boyce-Cood ;
8. effectuer les corrections nécessaires.

Il faut garder également à l'esprit que le modèle doit être exhaustif (c'est-à-dire contenir toutes les informations nécessaires) et éviter toute redondance (le prix unitaire d'un article n'a pas besoin de figurer dans l'entité commandes) qui constitue une perte d'espace, une démultiplication du travail de maintenance et un risque d'incohérence.

Il va de soi que cette méthodologie ne doit pas être suivie pas-à-pas une bonne fois pour toute. Au contraire, il faut itérer plusieurs fois les étapes successives, pour espérer converger vers une modélisation pertinente de la situation.

2.2 - Modèle logique de données-MLD

Maintenant que le MCD est établi, nous pouvons le traduire en différents systèmes logiques, comme les systèmes par fichiers ou les bases de données.

2.2.1 - Les systèmes logiques

Avant l'apparition des systèmes de gestion de base de données (SGBD ou DBMS pour Data Base Management System), les données étaient stockées dans des fichiers binaires et gérés par des programmes exécutables (Basic, Cobol ou Dbase par exemple). La maintenance des programmes (en cas de modification de la structure des données par exemple) était très problématique.

Sont alors apparus les SGBD hiérarchiques dans lesquels les données sont organisées en arbre (IMSDL1 d'IBM par exemple), puis les SGBD réseaux dans lesquels les données sont organisées selon un graphe plus général (IDS2 de Bull par exemple).

Aujourd'hui, ils sont largement remplacés par les SGBD relationnels (SGBDR) avec lesquels l'information peut être obtenue par une requête formulée dans un langage quasiment naturel. Ce sont les SGBD les plus répandus (PostgreSQL, Oracle, DB2 et base par exemple).

Nous nous contentons ici du modèle logique de données relationnel (MLDR). Plus récemment, sont apparus des SGBD orientés objets qui offrent à la fois un langage de requête et une navigation hypertexte.

Un modèle logique de données orienté objet permet par exemple de concevoir des classes Java s'appuyant sur une base de données relationnelle et permettant le développement d'une application web. Le langage Python (logiciel libre) possède une bibliothèque de plus en plus complète de SGBDRO.

2.2.2 - Schéma relationnel

Concentrons-nous sur le MLDR. Lorsque des données ont la même structure (comme par exemple, les bordereaux de livraison), nous pouvons les organiser en table dans laquelle les colonnes décrivent les attributs en commun et les lignes contiennent les valeurs de ces attributs pour chaque n-uplet.

Les n-uplets d'une table doivent être uniques, cela signifie qu'un attribut (au moins) doit servir de clé primaire. La clé primaire d'un n-uplet ne doit pas changer au cours du temps, alors que les autres attributs le peuvent.

Par ailleurs, il se peut qu'un attribut *Attribut1* d'une table ne doive contenir que des valeurs prises par un attribut *Attribut2* d'une autre table (par exemple, le numéro du client sur une commande doit correspondre à un vrai numéro de client). *Attribut2* doit être sans doublons (bien souvent il s'agit d'une clé primaire) et nous disons que *Attribut1* est une clé étrangère.

Par convention, nous soulignons les clés primaires et nous faisons précéder les clés étrangères d'un dièse # dans la description des attributs d'une table :

```
clients(numéro client, nom client, prénom, adresse, ...)  
commandes(numéro commande, date, #numéro client, ...)
```

Remarques :

- une même table peut avoir plusieurs clés étrangères mais une seule clé primaire ;
- une clé étrangère peut aussi être primaire ;
- une clé étrangère peut être composée (c'est le cas si la clé primaire en liaison est composée).

La section suivante contient des exemples. Nous pouvons représenter les tables d'une base de données relationnelle par un schéma relationnel dans lequel les tables sont appelées relations et les liens entre les clés étrangères et leur clé primaire est symbolisé par un connecteur (voir l'illustration 22).

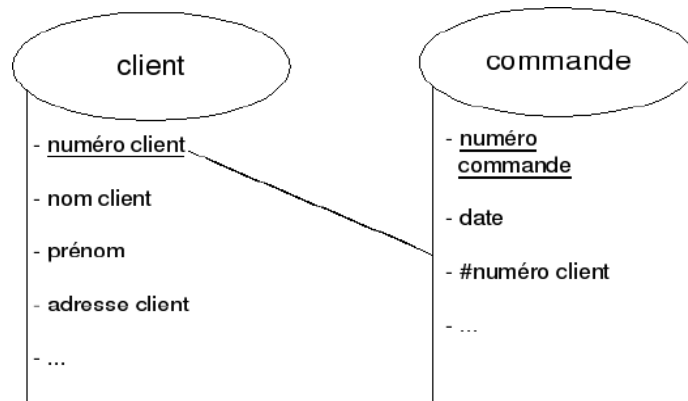


Illustration 22: Exemple d'un schéma relationnel

2.2.3 - Traduction

Pour traduire un MCD en un MLDR, il suffit d'appliquer cinq règles.

Notations : Nous disons qu'une association entre deux entités (éventuellement réflexive) est de type :

- 1 : 1 si les deux cardinalités sont 0,1 ou 1,1 (illustration 23) ;
- 1 : n si une des deux cardinalités est 0,n ou 1,n, l'autre cardinalité est 0,1 ou 1,1 (illustration 16) ;
- n : m (plusieurs à plusieurs) si les deux cardinalités sont 0,n ou 1,n (illustration 14).

	0,1	1,1	0,n	1,n
0,1	1:1	1:1	1:n	1:n
1,1	1:1	1:1	1:n	1:n
0,n	1:n	1:n	n:m	n:m
1,n	1:n	1:n	n:m	n:m

Tableau 2: Récapitulatif des types d'associations

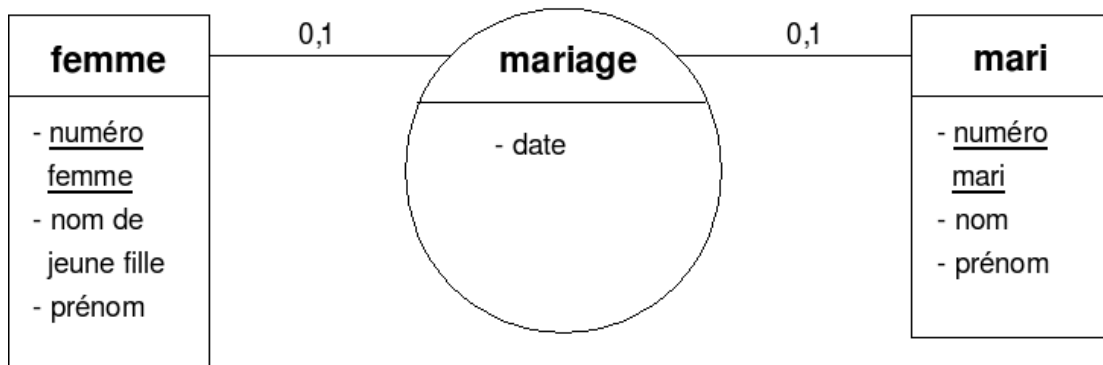


Illustration 23: Exemple d'une association de type 1:1

Règle 1

Toute entité devient une table dans laquelle les attributs sont ceux de l'entité. L'identifiant de l'entité constitue alors la clé primaire de la table. Par exemple, l'entité *article* de l'illustration 15 devient la table :

```
articles(numéro article, nom article, prix unitaire de vente, ...)
```

Règle 2

Une association binaire de type 1 : n disparaît, au profit d'une clé étrangère dans la table côté 0,1 ou 1,1 qui référence la clé primaire de l'autre table. Cette clé étrangère ne peut pas recevoir la valeur vide si la cardinalité est 1,1.

Par exemple, l'association *livrer* de l'illustration 15 est traduite par :

```
fournisseur(numéro fournisseur, nom contact, numéro téléphone contact)
livraison(numéro livraison, date de livraison, nom livreur,
#numéro fournisseur(nonvide))
```

Il ne devrait pas y avoir d'attribut dans une association de type 1 : n, mais s'il en reste, alors ils glissent vers la table côté 1.

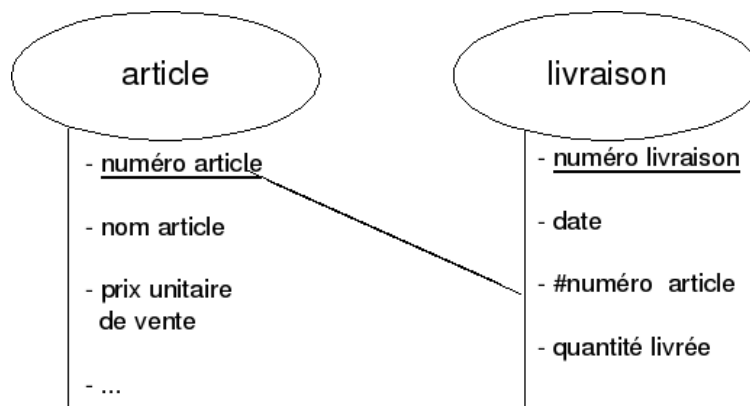


Illustration 24: Traduction d'une association de type 1:n

Règle 3

Une association binaire de type $n : m$ devient une table supplémentaire (parfois appelée table de jonction, table de jointure ou table d'association) dont la clé primaire est composée de deux clés étrangères (qui référencent les deux clés primaires des deux tables en association). Les attributs de l'association deviennent des attributs de cette nouvelle table.

Par exemple, l'association *concerner(1)* de l'illustration 15 est traduite par la table supplémentaire *ligne de commande* (illustration 25) :

ligne de commande(#numéro commande, #numéro article, quantité commandée)

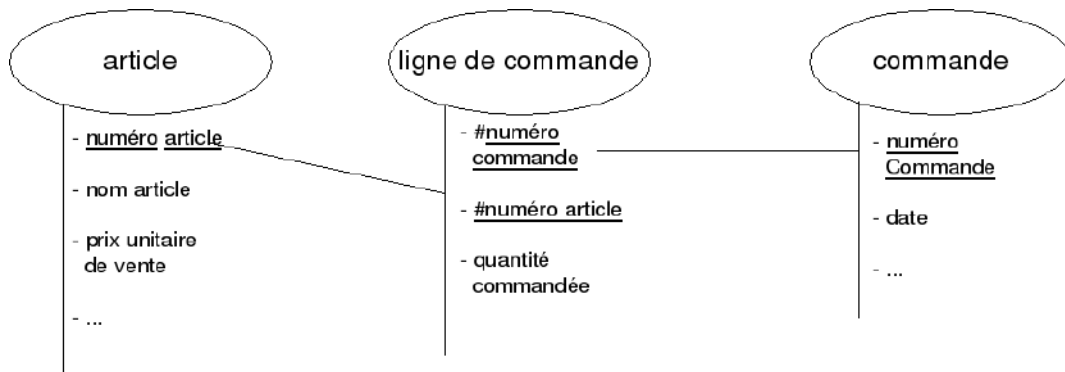


Illustration 25: Traduction d'une association de type $n:m$

Règle 4

Une association binaire de type $1 : 1$ disparaît, au profit d'une clé étrangère dans la table côté $1,1$ qui référence la clé primaire de l'autre table. Cette clé étrangère ne peut pas recevoir la valeur vide. Si les deux côtés sont de cardinalité $0,1$ alors la clé étrangère peut être placée indifféremment dans l'une des deux tables.

Par exemple, l'association *diriger* de l'illustration 26 est traduite par :

service(numéro service, nom service, #numéro employé (non vide, unique))
employé(numéro employé, nom)

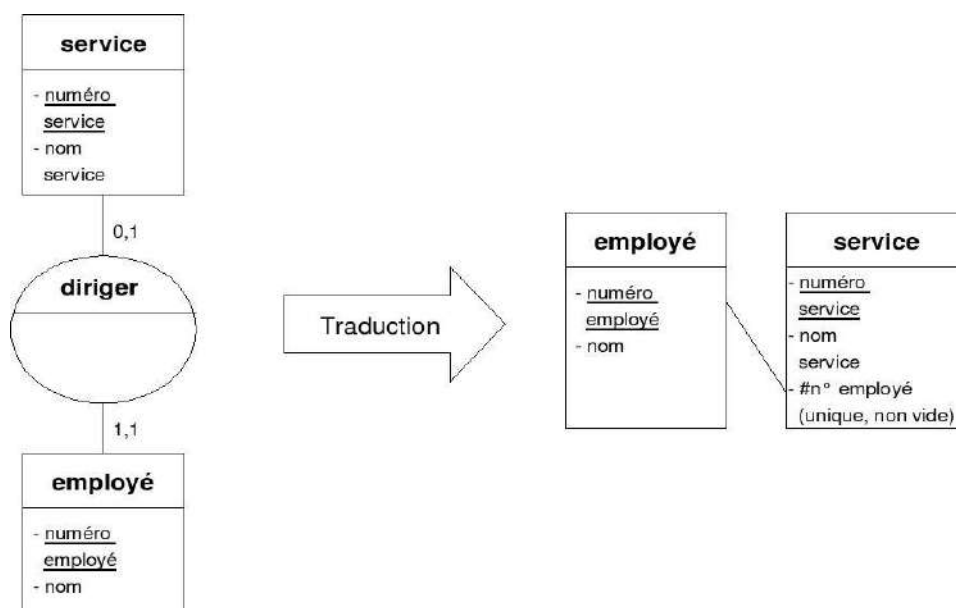


Illustration 26: Traduction d'une association de type $1:1$

Règle 5

Pour une association réflexive, nous avons deux cas :

- **premier cas** : pour une association de type 1 : n, la clé primaire de l'entité se dédouble et devient une clé étrangère dans la nouvelle table. Exactement comme si l'entité se dédoublait et était reliée par une relation binaire 1 : n. Par exemple, l'association *diriger* de l'illustration 6 est traduite par :

```
employe(numéro employé, nom, fonction, #numéro supérieur)
```

- **deuxième cas** : pour une association de type n : m, de même, tout se passe exactement comme si l'entité se dédoublait et était reliée par une relation binaire n : m. Il y a donc création d'une nouvelle table. Par exemple, l'association *parenté* de l'illustration 27 est traduite par :

```
personne(numéro personne, nom personne)  
parenté(#numéro parent, #numéro enfant)
```

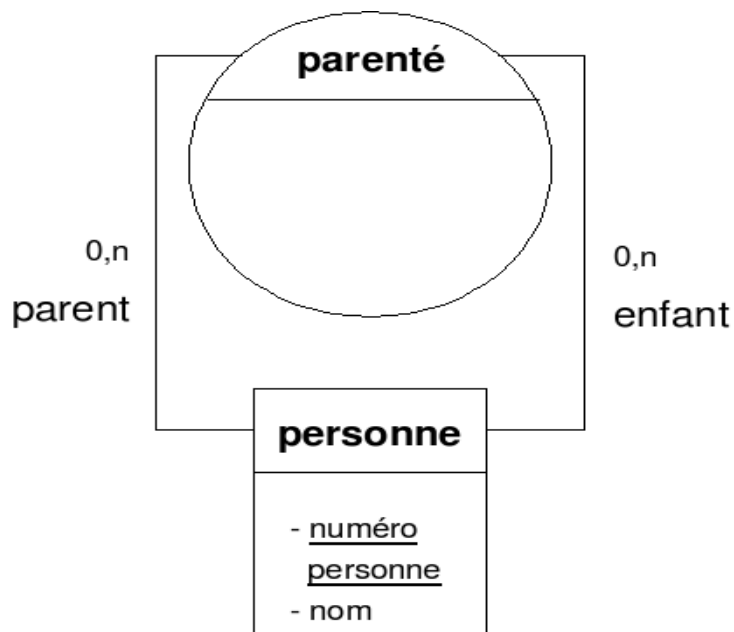


Illustration 27: Association binaire de type n : m.
Chaque personne a de 0 ou n descendants directs
(enfants) et a aussi de 0 à n ascendants (parents)

Règle 6

Une association non binaire est traduite par une table supplémentaire dont la clé primaire est composée d'autant de clés étrangères que d'entités en association. Les attributs de l'association deviennent des attributs de cette nouvelle table.

Par exemple, l'association *vol* de l'illustration 7 devient la table (illustration 28) :

```
vol(#numéro avion, #numéro pilote, #numéro_aéroport, date_et_heure, durée, distance)
```

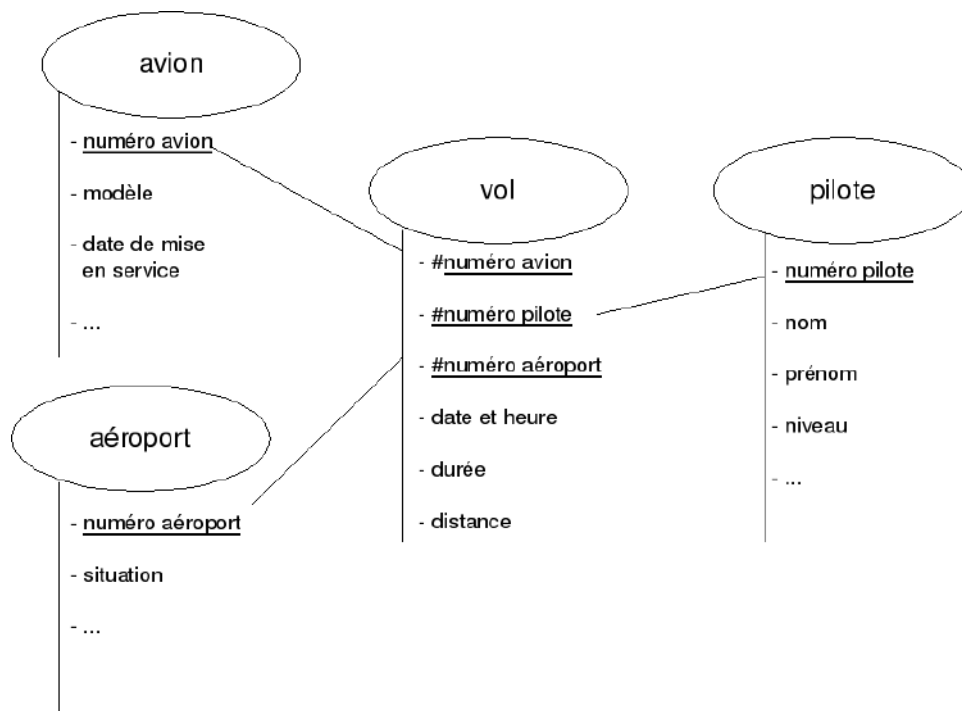


Illustration 28: Traduction d'une association ternaire

2.3 - Modèle physique de données-MPD

Bien que certains outils considèrent que le MPD et le MLD représentent la même chose, c'est faux. Le MPD est une implémentation particulière du MLD pour un matériel, un environnement et un logiciel donné.

Notamment, le MPD s'intéresse au stockage des données à travers le type et la taille (en octets ou en bits) des attributs du MCD. Cela permet de prévoir la place nécessaire à chaque table dans le cas d'un SGBDR.

Le MPD tient compte des limites matérielles et logicielles afin d'optimiser l'espace consommé et d'optimiser le temps de calcul (qui représentent deux optimisations contradictoires). Dans le cas d'un SGBDR, le MPD définit les index et peut être amené à accepter certaines redondances d'information afin d'accélérer les requêtes.

Par exemple, la table *commande* de l'illustration 25 peut être supprimées et ses colonnes, notamment date, sont ajoutées à la table *ligne de commande*. Nous renonçons donc à la troisième forme normale puisque la date est répété autant de fois qu'il y a de lignes dans la commande, mais nous évitons une jointure coûteuse en temps de calcul lors des requêtes.

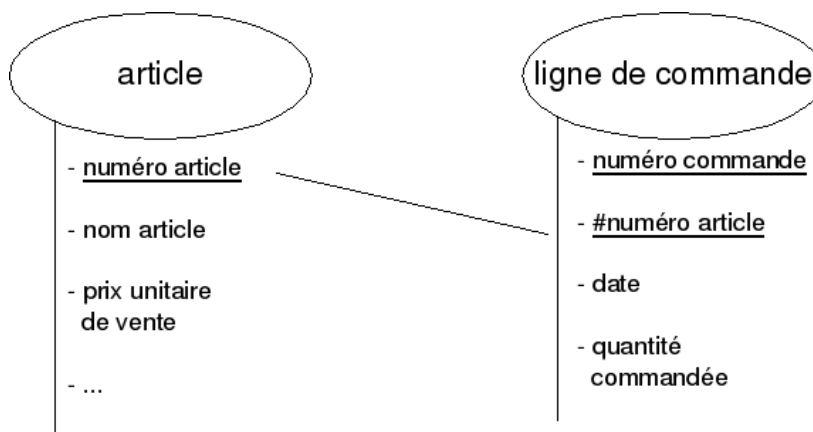


Illustration 29: Sacrifice de la troisième forme normale

Une application pratique de la séparation entre le MLDR et le MPD est le portage d'une base de données d'un SGBD à un autre. Par exemple, nous pouvons traduire un MPD base en un MLDR puis traduire ce MLDR en un MPD Oracle.

2.4 - Conclusion

Les Intérêts de la décomposition MCD/MLD/MPD sont que :

- le MCD permet d'éviter certaines erreurs de conception (en contradiction avec les règles de normalisation) ;
- le MCD permet de voir quelles associations de type $n : m$, non binaires ou réflexives sont en présence ;
- le MLD peut être obtenu automatiquement par des outils de génie logiciel ;
- le MCD peut être traduit en différents MLD cohérents (notamment nous pouvons traduire un MCD en un MLDR puis en une base de données tandis qu'en parallèle le MCD est traduit en un ensemble de classes Java (MPD orienté objet) afin de développer une application web sur cette base).

2.5 - Références

0 Gabay, J. Apprendre et pratiquer Merise. Masson, 1989. Ce livre très synthétique permet de s'exercer sur la méthode.

1 Matheron, J.-P. Comprendre Merise. Eyrolles, 1994. Cet ouvrage très accessible permet vraiment de comprendre la méthode.

2 Nanci, D., Espinasse, B., Cohen, B. et Heckenroth, H. Ingénierie des systèmes d'information avec Merise. Sybex, 1992. Cet ouvrage complet détaille la méthode dans son ensemble.

3 Panet, G., Letouche, R. et Tardieu, H. Merise/2 : Modèles et techniques Merise avancés. Edition d'organisation, 1994. Ce livre décrit la version 2 de la méthode.

4 Tardieu, H., Rochfeld, A. et Coletti, R. La méthode Merise. Principes et outils. Edition d'organisation, 1986. Il s'agit-là du livre de référence par les auteurs de la méthode.

3 - Le langage SQL

SQL signifie **Structured Query Language** c'est-à-dire « langage d'interrogation structuré ». C'est un langage de gestion des bases de données relationnelles. Il a été conçu par IBM dans les années 70. Il est devenu le langage standard des **Systèmes de Gestion de Bases de Données (SGBD) Relationnelles (SGBDR)**. Il a pour fonction l'aide à la définition, la manipulation et le contrôle des données d'une base de données relationnelle. Une base de données relationnelle n'est rien d'autre qu'une base de données perçue par l'utilisateur comme un ensemble de tables. Une table est un ensemble non ordonné de lignes.

SQL est à la fois :

- un langage de définition des données : ordres CREATE, ALTER, DROP;
- un langage de manipulation des données : ordres UPDATE, INSERT, DELETE;
- un langage d'interrogation : ordre SELECT.

Le tableau 3 contient un exemple de base de données contenant des fournisseurs et des pièces détachées. Les tables F,P et FP de cette figure représentent respectivement les fournisseurs, les pièces détachées et les expéditions de pièces détachées faites par les fournisseurs.

F			
F_NO	F_NOM	STATUT	VILLE
F1	0	20	0
F2	0	10	0
F3	0	30	0
F4	0	20	0
F5	0	30	0

FP		
F_NO	P_NO	QTE
F1	0	300
F1	0	200
F1	0	400
F1	0	200
F1	0	100
F1	0	100
F2	0	300
F2	0	400
F3	0	200
F4	0	200
F4	0	300
F4	0	400

P				
P_NO	P_NOM	COULEUR	POIDS	VILLE
P1	0	0	12	0
P2	0	0	17	0
P3	0	0	17	0
P4	0	0	14	0
P5	0	0	12	0
P6	0	0	19	0

Tableau 3: Echantillons de données de la base fournisseurs-pièces détachées

Le langage SQL est utilisé par les principaux SGBDR : PostgreSQL, MySQL, fireBird, DB2, Oracle, Sybase, etc.

Chacun de ces SGBDR a cependant sa propre variante du langage. Ce support de cours présente les principales commande disponible sous PostgreSQL.

3.1 - Concept :

PostgreSQL est un *système de gestion de bases de données relationnelles* (SGBDR).

Chaque table est un ensemble de *lignes*. Chaque ligne d'une table donnée a le même ensemble de *colonnes* et chaque colonne est d'un *type* de données particulier. Tandis que les colonnes ont un ordre fixé dans chaque ligne, il est important de savoir que SQL ne garantit, d'aucune façon, l'ordre des lignes à l'intérieur de la table (bien qu'elles puissent être explicitement triées pour l'affichage).

3.1.1 - Créer une nouvelle table :

Vous pouvez créer une nouvelle table en spécifiant le nom de la table, suivi de tous les noms des colonnes et de leur type :

```
CREATE TABLE temps (  
  ville          varchar(80),  
  t_basse       int,      -- température basse  
  t_haute       int,      -- température haute  
  prcp          real,     -- précipitation  
  date          date  
);
```

Vous pouvez saisir cela dans **psql** avec les sauts de lignes. **psql** reconnaîtra que la commande n'est pas terminée jusqu'à arriver à un point-virgule.

Les espaces blancs (c'est-à-dire les espaces, les tabulations et les retours à la ligne) peuvent être librement utilisés dans les commandes SQL. Cela signifie que vous pouvez saisir la commande ci-dessus alignée différemment ou même sur une seule ligne. Deux tirets (« -- ») introduisent des commentaires. Ce qui les suit est ignoré jusqu'à la fin de la ligne. SQL est insensible à la casse pour les mots-clé et les identifiants excepté quand les identifiants sont entre double guillemets pour préserver leur casse (non fait ci-dessus).

varchar(80) spécifie un type de données pouvant contenir une chaîne de caractères arbitraires de 80 caractères au maximum. int est le type entier normal. real est un type pour les nombres décimaux en simple précision. date devrait s'expliquer de lui-même (oui, la colonne de type date est aussi nommée date ; cela peut être commode ou porter à confusion, à vous de choisir).

PostgreSQL™ prend en charge les types SQL standards int, smallint, real, double precision, char(N), varchar(N), date, time, timestamp et interval ainsi que d'autres types d'utilité générale et un riche ensemble de types géométriques. PostgreSQL™ peut être personnalisé avec un nombre arbitraire de types de données définis par l'utilisateur. En conséquence, les noms des types ne sont pas des mots-clé syntaxiques sauf lorsqu'il est requis de supporter des cas particuliers dans la norme SQL.

Le second exemple stockera des villes et leur emplacement géographique associé :

```
CREATE TABLE villes (  
  nom            varchar(80),  
  emplacement    point  
);
```

Le type point est un exemple d'un type de données spécifique à PostgreSQL™.

Pour finir, vous devez savoir que si vous n'avez plus besoin d'une table ou que vous voulez la recréer différemment, vous pouvez la supprimer en utilisant la commande suivante :

```
DROP TABLE nom_table;
```

3.1.2 - Remplir une table avec des lignes

L'instruction **INSERT** est utilisée pour remplir une table avec des lignes :

```
INSERT INTO temps VALUES ('San Francisco', 46, 50, 0.25, '1994-11-27');
```

Notez que tous les types utilisent des formats d'entrées plutôt évident. Les constantes qui ne sont pas des valeurs numériques simples doivent être habituellement entourées par des guillemets simples (') comme dans l'exemple. Le type date est en réalité tout à fait flexible dans ce qu'il accepte mais, pour ce tutoriel, nous collerons au format non ambigu montré ici.

Le type point demande une paire de coordonnées en entrée comme cela est montré ici :

```
INSERT INTO villes VALUES ('San Francisco', '(-194.0, 53.0)');
```

La syntaxe utilisée jusqu'à maintenant nécessite de se rappeler l'ordre des colonnes. Une syntaxe alternative vous autorise à lister les colonnes explicitement :

```
INSERT INTO temps (ville, t_basse, t_haute, prcp, date)
VALUES ('San Francisco', 43, 57, 0.0, '1994-11-29');
```

Vous pouvez lister les colonnes dans un ordre différent si vous le souhaitez ou même omettre certaines colonnes ; par exemple, si la précipitation est inconnue :

```
INSERT INTO temps (date, ville, t_haute, t_basse)
VALUES ('1994-11-29', 'Hayward', 54, 37);
```

De nombreux développeurs considèrent que le listage explicite des colonnes est un meilleur style que de compter sur l'ordre implicite.

S'il vous plaît, exécutez toutes les commandes vues ci-dessus de façon à avoir des données sur lesquelles travailler dans les prochaines sections.

Vous auriez pu aussi utiliser **COPY** pour charger de grandes quantités de données depuis des fichiers texte. C'est habituellement plus rapide car la commande **COPY** est optimisée pour cet emploi mais elle est moins flexible que **INSERT**. Par exemple :

```
COPY temps FROM '/home/user/temps.txt';
```

où le nom du fichier source doit être disponible sur la machine serveur et non pas sur le client puisque le serveur lit le fichier directement.

3.1.3 - Interroger une table :

Pour retrouver les données d'une table, elle est *interrogée*. Une instruction SQL **SELECT** est utilisée pour faire cela. L'instruction est divisée en liste de sélection (la partie qui liste les colonnes à retourner), une liste de tables (la partie qui liste les tables à partir desquelles les données seront retrouvées) et une qualification optionnelle (la partie qui spécifie les restrictions). Par exemple, pour retrouver toutes les lignes de la table temps, saisissez :

```
SELECT * FROM temps;
```

Ici, * est un raccourci pour « toutes les colonnes ». Donc, le même résultat pourrait être obtenu avec :

```
SELECT ville, t_basse, t_haute, prcp, date FROM temps;
```

Le résultat devrait être ceci :

ville	t_basse	t_haute	prcp	date
San Francisco	46	50	0.25	1994-11-27
San Francisco	43	57	0	1994-11-29
Hayward	37	54		1994-11-29

(3 rows)

Alors que SELECT * est utile pour des requêtes rapides, c'est généralement considéré comme un mauvais style dans un code en production car l'ajout d'une colonne dans la table changerait les résultats.

Vous pouvez écrire des expressions, pas seulement des références à de simples colonnes, dans la liste de sélection. Par exemple, vous pouvez faire :

```
SELECT ville, (t_haute+t_basse)/2 AS temp_moy, date FROM temps;
```

Cela devrait donné :

ville	temp_moy	date
San Francisco	48	1994-11-27
San Francisco	50	1994-11-29
Hayward	45	1994-11-29

(3 rows)

Notez comment la clause AS est utilisée pour renommer la sortie d'une colonne (cette clause AS est optionnelle).

Une requête peut être « qualifiée » en ajoutant une clause WHERE qui spécifie quelles lignes sont souhaitées. La clause WHERE contient une expression booléenne et seules les lignes pour lesquelles l'expression booléenne est vraie sont renvoyées. Les opérateurs booléens habituels (AND, OR et NOT) sont autorisés dans la qualification. Par exemple, ce qui suit recherche le temps à San Francisco les jours pluvieux :

```
SELECT * FROM temps
WHERE ville = 'San Francisco' AND prcp > 0.0;
```

Résultat :

ville	t_basse	t_haute	prcp	date
San Francisco	46	50	0.25	1994-11-27

(1 row)

Vous pouvez demander que les résultats d'une requêtes soient renvoyés dans un ordre trié :

```
SELECT * FROM temps
ORDER BY ville;
```

ville	t_basse	t_haute	prcp	date
Hayward	37	54		1994-11-29
San Francisco	43	57	0	1994-11-29
San Francisco	46	50	0.25	1994-11-27

Dans cet exemple, l'ordre de tri n'est pas spécifié complètement, donc vous pouvez obtenir les lignes San Francisco dans n'importe quel ordre. Mais, vous auriez toujours obtenu les résultats affichés ci-dessus si vous aviez fait :

```
SELECT * FROM temps
ORDER BY ville, t_basse;
```

Vous pouvez demander que les lignes dupliquées soient supprimées du résultat d'une requête :

```
SELECT DISTINCT ville
FROM temps;
```

```
ville
-----
Hayward
San Francisco
(2 rows)
```

De nouveau, l'ordre des lignes résultats pourrait varier. Vous pouvez vous assurer des résultats cohérents en utilisant DISTINCT et ORDER BY ensemble : [\[3\]](#)

```
SELECT DISTINCT ville
FROM temps
ORDER BY ville;
```

3.1.4 - Jointures entre les tables :

Jusqu'ici, nos requêtes avaient seulement consulté une table à la fois. Les requêtes peuvent accéder à plusieurs tables en même temps (ou accéder à la même table de façon à ce que plusieurs lignes de la table soient traitées en même temps). Une requête qui consulte plusieurs lignes de la même ou de différentes tables en même temps est appelée requête de *jointure*. Comme exemple, supposez que vous souhaitez lister toutes les entrées de la table temps avec la colonne des noms de toutes les lignes de la table des villes et que vous choisissiez les paires de lignes où ces valeurs correspondent.

Ceci sera accompli avec la requête suivante :

```
SELECT *
FROM temps, villes
WHERE ville = nom;
```

```
ville      | t_basse | t_haute | prcp | date       | nom           | emplacement
-----+-----+-----+-----+-----+-----+-----
San Francisco | 46      | 50      | 0.25 | 1994-11-27 | San Francisco | (-194,53)
San Francisco | 43      | 57      | 0     | 1994-11-29 | San Francisco | (-194,53)
(2 rows)
```

Remarquez deux choses à propos du résultat :

- Il n'y a pas de lignes pour la ville de Hayward dans le résultat. C'est parce qu'il n'y a aucune entrée correspondante dans la table villes pour Hayward, donc la jointure ignore les lignes n'ayant pas de correspondance avec la table temps. Nous verrons rapidement comment cela peut être résolu.
- Il y a deux colonnes contenant le nom des villes. C'est correct car les listes de colonnes des tables temps et villes sont concaténées. En pratique, ceci est indésirable, vous voudrez probablement lister les colonnes explicitement plutôt que d'utiliser « * » :

```
SELECT ville, t_basse, t_haute, prcp, date, emplacement
FROM temps, villes
WHERE ville = nom;
```

Puisque toutes les colonnes ont un nom différent, l'analyseur a automatiquement trouvé à quelle table elles appartiennent. Si des noms de colonnes sont communs entre les deux tables, vous aurez besoin de *qualifier* les noms des colonnes pour préciser celle dont vous parlez. Par exemple :

```
SELECT temps.ville, temps.t_basse, temps.t_haute, temps.prcp, temps.date, villes.emplacement
FROM temps, villes
WHERE villes.nom = temps.ville;
```

La qualification des noms de colonnes dans une requête de jointure est fréquemment considérée comme une bonne pratique. Cela évite l'échec de la requête si un nom de colonne dupliqué est ajouté plus tard dans une des tables.

Les requêtes de jointure vues jusqu'ici peuvent aussi être écrites dans un format alternatif :

```
SELECT *
FROM temps INNER JOIN villes ON (temps.ville = villes.nom);
```

Cette syntaxe n'est pas aussi couramment utilisée que les précédentes mais nous la montrons ici pour vous aider à comprendre les sujets suivants.

Maintenant, nous allons essayer de comprendre comment nous pouvons avoir les entrées de Hayward. Nous voulons que la requête parcourt la table temps et que, pour chaque ligne, elle trouve la (ou les) ligne(s) de villes correspondante(s). Si aucune ligne correspondante n'est trouvée, nous voulons que les valeurs des colonnes de la table villes soient remplacées par des « valeurs vides ». Ce genre de requêtes est appelé *jointure externe* (outer join). (Les jointures que nous avons vues jusqu'ici sont des jointures internes -- inner joins). La commande ressemble à cela :

```
SELECT *
FROM temps LEFT OUTER JOIN villes ON (temps.ville = villes.nom);
```

ville	t_basse	t_haute	prcp	date	nom	emplacement
Hayward	37	54		1994-11-29		
San Francisco	46	50	0.25	1994-11-27	San Francisco	(-194,53)
San Francisco	43	57	0	1994-11-29	San Francisco	(-194,53)

(3 rows)

Cette requête est appelée une *jointure externe à gauche* (left outer join) parce que la table mentionnée à la gauche de l'opérateur de jointure aura au moins une fois ses lignes dans le résultat tandis que la table sur la droite aura seulement les lignes qui correspondent à des lignes de la table de gauche. Lors de l'affichage d'une ligne de la table de gauche pour laquelle il n'y a pas de correspondance dans la table de droite, des valeurs vides (appelées NULL) sont mises pour les colonnes de la table de droite.

Nous pouvons également joindre une table avec elle-même. Ceci est appelé une *jointure réflexive*. Comme exemple, supposons que nous voulons trouver toutes les entrées de temps qui sont dans un intervalle de température d'autres entrées de temps. Nous avons donc besoin de comparer les colonnes t_basse et t_haute de chaque ligne de temps aux colonnes t_basse et t_haute de toutes les autres lignes de temps. Nous pouvons faire cela avec la requête suivante :

```
SELECT T1.ville, T1.t_basse AS bas, T1.t_haute AS haut,
       T2.ville, T2.t_basse AS bas, T2.t_haute AS haut
FROM temps T1, temps T2
WHERE T1.t_basse < T2.t_basse
AND T1.t_haute > T2.t_haute;
```

ville	bas	haut	ville	bas	haut
San Francisco	43	57	San Francisco	46	50
Hayward	37	54	San Francisco	46	50

(2 rows)

Ici, nous avons renommé la table temps en T1 et en T2 pour être capable de distinguer le côté gauche et droit de la jointure. Vous pouvez aussi utiliser ce genre d'alias dans d'autres requêtes pour économiser de la frappe, c'est-à-dire :

```
SELECT *
FROM temps t, villes v
WHERE t.ville = v.nom;
```

Vous rencontrerez ce genre d'abréviation assez fréquemment.

3.1.5 - Les opérations ensemblistes

Les opérations ensemblistes en SQL, sont celles définies dans l'algèbre relationnelle. Elles sont réalisées grâce aux opérateurs :

- **UNION**
- **INTERSECT** (ne fait pas partie de la norme SQL et n'est donc pas implémenté dans tous les SGBD)
- **EXCEPT** (ne fait pas partie de la norme SQL et n'est donc pas implémenté dans tous les SGBD)

Ces opérateurs s'utilisent entre deux clauses SELECT.

L'opérateur UNION

Cet opérateur permet d'effectuer une union des lignes sélectionnées par deux clauses SELECT (les deux tables sur lesquelles on travaille devant avoir le même schéma).

```
SELECT ---- FROM ---- WHERE -----
UNION
SELECT ---- FROM ---- WHERE -----
```

Par défaut les doublons sont automatiquement éliminés. Pour conserver les doublons, il est possible d'utiliser une clause UNION ALL.

L'opérateur INTERSECT

Cet opérateur permet d'effectuer une intersection des tuples sélectionnés par deux clauses SELECT (les deux tables sur lesquelles on travaille devant avoir le même schéma).

```
SELECT ---- FROM ---- WHERE -----  
INTERSECT  
SELECT ---- FROM ---- WHERE -----
```

L'opérateur INTERSECT n'étant pas implémenté dans tous les SGBD, il est possible de le remplacer par des commandes usuelles :

```
SELECT a,b FROM table1  
WHERE EXISTS (SELECT c,d FROM table2  
              WHERE a=c AND b=d )
```

L'opérateur EXCEPT

Cet opérateur permet d'effectuer une différence entre les lignes sélectionnés par deux clauses SELECT, c'est-à-dire sélectionner les lignes de la première table n'appartenant pas à la seconde (les deux tables devant avoir le même schéma).

```
SELECT a,b FROM table1 WHERE -----  
EXCEPT  
SELECT c,d FROM table2 WHERE -----
```

L'opérateur EXCEPT n'étant pas implémenté dans tous les SGBD, il est possible de le remplacer par des commandes usuelles :

```
SELECT a,b FROM table1  
WHERE NOT EXISTS ( SELECT c,d FROM table2  
                   WHERE a=c AND b=d )
```

3.1.6 - Fonctions d'agrégat

Une fonction d'agrégat calcule un seul résultat à partir de plusieurs lignes en entrée. Par exemple, il y a des agrégats pour calculer le nombre (count), la somme (sum), la moyenne (avg), le maximum (max) et le minimum (min) d'un ensemble de lignes.

Comme exemple, nous pouvons trouver la température la plus haute parmi les températures basses avec :

```
SELECT max(t_basse) FROM temps;  
  
max  
-----  
46  
(1 row)
```

Si nous voulons connaître dans quelle ville (ou villes) ces lectures se sont produites, nous pouvons essayer :

```
SELECT ville FROM temps WHERE t_basse = max(t_basse);  
FAUX
```

mais cela ne marchera pas puisque l'agrégat max ne peut pas être utilisé dans une clause WHERE (cette restriction existe parce que la clause WHERE détermine les lignes qui seront traitées par l'agrégat ; donc les lignes doivent être évaluées avant que les fonctions d'agrégat calculent).

Cependant, comme cela est souvent le cas, la requête peut être répétée pour arriver au résultat attendu, ici en utilisant une *sous-requête* :

```
SELECT ville FROM temps
WHERE t_basse = (SELECT max(t_basse) FROM temps);

ville
-----
San Francisco
(1 row)
```

Ceci est correct car la sous-requête est un calcul indépendant qui traite son propre agrégat séparément à partir de ce qui se passe dans la requête externe.

Les agrégats sont également très utiles s'ils sont combinés avec les clauses GROUP BY. Par exemple, nous pouvons obtenir température la plus haute parmi les températures basses observées dans chaque ville avec :

```
SELECT ville, max(t_basse)
FROM temps
GROUP BY ville;

ville      | max
-----+-----
Hayward    | 37
San Francisco | 46
(2 rows)
```

ce qui nous donne une ligne par ville dans le résultat. Chaque résultat d'agrégat est calculé avec les lignes de la table correspondant à la ville. Nous pouvons filtrer ces lignes groupées en utilisant HAVING :

```
SELECT ville, max(t_basse)
FROM temps
GROUP BY ville
HAVING max(t_basse) < 40;

ville      | max
-----+-----
Hayward    | 37
(1 row)
```

ce qui nous donne le même résultat uniquement pour les villes qui ont toutes leurs valeurs de t_basse en-dessous de 40. Pour finir, si nous nous préoccupons seulement des villes dont le nom commence par « S », nous pouvons faire :

```
SELECT ville, max(t_basse)
FROM temps
WHERE ville LIKE 'S%'
GROUP BY ville
HAVING max(t_basse) < 40;
```


Il est important de comprendre l'interaction entre les agrégats et les clauses SQL WHERE et HAVING. La différence fondamentale entre WHERE et HAVING est que WHERE sélectionne les lignes en entrée avant que les groupes et les agrégats ne soient traités (donc, cette clause contrôle les lignes qui se retrouvent dans le calcul de l'agrégat) tandis que HAVING sélectionne les lignes groupées après que les groupes et les agrégats aient été traités. Donc, la clause WHERE ne doit pas contenir des fonctions d'agrégat ; cela n'a aucun sens d'essayer d'utiliser un agrégat pour déterminer quelles lignes seront en entrée des agrégats. D'un autre côté, la clause HAVING contient toujours des fonctions d'agrégat (pour être précis, vous êtes autorisés à écrire une clause HAVING qui n'utilise pas d'agrégats mais c'est rarement utilisé. La même condition pourra être utilisée plus efficacement par un WHERE).

Dans l'exemple précédent, nous pouvons appliquer la restriction sur le nom de la ville dans WHERE puisque cela n'a besoin d'aucun agrégat. C'est plus efficace que d'ajouter la restriction dans HAVING parce que nous évitons le groupement et les calculs d'agrégat pour toutes les lignes qui ont échoué lors du contrôle fait par WHERE.

3.1.7 - Mises à jour

Vous pouvez mettre à jour une ligne existante en utilisant la commande **UPDATE**. Supposez que vous découvrez que les températures sont toutes excédentes de 2 degrés après le 28 novembre. Vous pouvez corriger les données de la façon suivante :

```
UPDATE temps
SET t_haute = t_haute - 2, t_basse = t_basse - 2
WHERE date > '1994-11-28';
```

Regardez le nouvel état des données :

```
SELECT * FROM temps;
```

ville	t_basse	t_haute	prcp	date
San Francisco	46	50	0.25	1994-11-27
San Francisco	41	55	0	1994-11-29
Hayward	35	52		1994-11-29

(3 rows)

3.1.8 - Suppressions

Les lignes peuvent être supprimées de la table avec la commande **DELETE**. Supposez que vous n'êtes plus intéressé par le temps de Hayward. Vous pouvez faire ce qui suit pour supprimer ses lignes de la table :

```
DELETE FROM temps WHERE ville = 'Hayward';
```

Toutes les entrées de temps pour Hayward sont supprimées.

```
SELECT * FROM temps;
```

ville	t_basse	t_haute	prcp	date
San Francisco	46	50	0.25	1994-11-27
San Francisco	41	55	0	1994-11-29

(2 rows)

Faire très attention aux instructions de la forme

```
DELETE FROM nom_table;
```

Sans une qualification, **DELETE** supprimera *toutes* les lignes de la table donnée, la laissant vide. Le système le fera sans demander de confirmation !

3.2 - Fonctionnalités avancées

3.2.1 - Introduction

Le chapitre précédant couvre les bases de l'utilisation de SQL pour le stockage et l'accès aux données avec PostgreSQL™. Il est temps d'aborder quelques fonctionnalités avancées du SQL qui simplifient la gestion et empêchent la perte ou la corruption des données.

Le langage SQL pour les modifier ou les améliorer. Il est donc préférable d'avoir lu ce chapitre. Quelques exemples de ce chapitre sont également disponibles dans advanced.sql. Ce fichier contient, de plus, quelques données à charger pour utiliser l'exemple.

3.2.2 - Vues

Si la liste des enregistrements du temps et des villes est d'un intérêt particulier pour l'application considérée mais qu'il devient contraignant de saisir la requête à chaque utilisation, il est possible de créer une *vue* avec la requête. De ce fait, la requête est nommée et il peut y être fait référence de la même façon qu'il est fait référence à une table :

```
CREATE VIEW ma_vue AS
  SELECT ville, t_basse, t_haute, prcp, date, emplacement
  FROM temps, villes
  WHERE ville = 'San Francisco';

SELECT * FROM ma_vue;
```

Une vue est une table virtuelle, c'est-à-dire une table qui n'a pas d'existence propre mais qui apparaît réelle à l'utilisateur. La commande **SELECT** définissant une vue n'est pas exécutée au moment de la création de la vue mais est simplement mémorisée par le système. Mais l'utilisateur a maintenant l'impression qu'il existe réellement dans la base de données une table portant le nom `ma_vue`.

L'utilisation des vues est un aspect clé d'une bonne conception des bases de données SQL. Les vues permettent d'encapsuler les détails de la structure des tables. Celle-ci peut alors changer avec l'évolution de l'application, tandis que l'interface reste constante.

Les vues peuvent être utilisées dans quasiment toutes les situations où une vraie table est utilisable. Il n'est, de plus, pas inhabituel de construire des vues reposant sur d'autres vues.

3.2.3 - Clé primaire

La commande **CREATE TABLE** comprend une option **PRIMARY KEY** permettant de créer une clé primaire. Voici un exemple où la clé primaire est multi-colonne:

```
CREATE TABLE (  
    col1    int,  
    col2    int,  
    col3    int,  
    CONSTRAINT ma_cle PRIMARY KEY (col1, col2)  
);
```

Si une clé primaire est uni-colonne, nous pouvons créer une clé primaire ainsi :

```
CREATE TABLE Pays (  
    P_nom      varchar(40) PRIMARY KEY,  
    Langue_Off varchar(20),  
    HAB        int,);
```

3.2.4 - Clés étrangères

Il s'agit maintenant de s'assurer que personne n'insère de ligne dans la table temps qui ne corresponde à une entrée dans la table villes. On appelle cela maintenir l'*intégrité référentielle* des données. Dans les systèmes de bases de données simplistes, lorsqu'au moins c'est possible, cela est parfois obtenu par la vérification préalable de l'existence d'un enregistrement correspondant dans la table villes, puis par l'insertion, ou l'interdiction, du nouvel enregistrement dans temps. Puisque cette approche, peu pratique, présente un certain nombre d'inconvénients, PostgreSQL™ peut se charger du maintien de l'*intégrité référentielle*.

La nouvelle déclaration des tables ressemble alors à ceci :

```
CREATE TABLE villes (  
    ville      varchar(80) primary key,  
    emplacement point  
);  
  
CREATE TABLE temps (  
    ville      varchar(80) references villes,  
    t_haute int,  
    t_basse int,  
    prcp      real,  
    date      date  
);
```

Lors d'une tentative d'insertion d'enregistrement non valide :

```
INSERT INTO temps VALUES ('Berkeley', 45, 53, 0.0, '1994-11-28');  
  
ERROR: insert or update on table "temps" violates foreign key constraint "temps_ville_fkey"  
DETAIL: Key (ville)=(a) is not present in table "villes".
```

Le comportement des clés étrangères peut être adapté très finement à une application particulière.

3.2.5 - Héritage

L'héritage est un concept issu des bases de données orientées objet. Il ouvre de nouvelles possibilités intéressantes en conception de bases de données.

Soient deux tables : une table villes et une table capitales. Les capitales étant également des villes, il est intéressant d'avoir la possibilité d'afficher implicitement les capitales lorsque les villes sont listées. On pourrait écrire ceci :

```

CREATE TABLE capitales (
  nom      text,
  population real,
  altitude int, -- (en pied)
  etat     char(2) );

CREATE TABLE non_capitales (
  nom      text,
  population real,
  altitude int -- (en pied)
);

CREATE VIEW villes AS
SELECT nom, population, altitude FROM capitales
UNION
SELECT nom, population, altitude FROM non_capitales;

```

Cela fonctionne bien pour les requêtes, mais la mise à jour d'une même donnée sur plusieurs lignes devient vite un horrible casse-tête.

Une meilleure solution :

```

CREATE TABLE villes (
  nom      text,
  population real,
  altitude int -- (en pied)
);

CREATE TABLE capitales (
  etat char(2)
) INHERITS (villes);

```

Dans ce cas, une ligne de capitales *hérite* de toutes les colonnes (nom, population et altitude) de son *parent*, villes. Le type de la colonne nom est text, un type natif de PostgreSQL™ pour les chaînes de caractères à longueur variable. Les capitales d'état ont une colonne supplémentaire, etat, qui affiche l'état dont elles sont la capitale. Sous PostgreSQL™, une table peut hériter de zéro à plusieurs autres tables.

La requête qui suit fournit un exemple de récupération des noms de toutes les villes, en incluant les capitales des états, situées à une altitude de plus de 500 pieds :

```

SELECT nom, altitude
FROM villes
WHERE altitude > 500;

```

ce qui renvoie :

nom	altitude
Las Vegas	2174
Mariposa	1953
Madison	845

(3 rows)

À l'inverse, la requête qui suit récupère toutes les villes qui ne sont pas des capitales et qui sont situées à une altitude d'au moins 500 pieds :

```

SELECT nom, altitude
FROM ONLY villes
WHERE altitude > 500;

```

nom	altitude
Las Vegas	2174
Mariposa	1953

(2 rows)

Ici, ONLY avant villes indique que la requête ne doit être exécutée que sur la table villes, et non pas sur les tables en dessous de villes dans la hiérarchie des héritages. La plupart des commandes déjà évoquées -- **SELECT**, **UPDATE** et **DELETE** -- supportent cette notation (ONLY).

Note

Bien que l'héritage soit fréquemment utile, il n'a pas été intégré avec les contraintes d'unicité et les clés étrangères, ce qui limite son utilité.

3.3 - Syntaxe SQL

Une entrée SQL consiste en une séquence de *commandes*. Une commande est composée d'une séquence de *jetons*, terminés par un point-virgule (« ; »). La fin du flux en entrée termine aussi une commande. Les jetons valides dépendent de la syntaxe particulière de la commande.

Un jeton peut être un *mot clé*, un *identifieur*, un *identifieur entre guillemets*, un *littéral* (ou une constante) ou un symbole de caractère spécial. Les jetons sont normalement séparés par des espaces blancs (espace, tabulation, nouvelle ligne) mais n'ont pas besoin de l'être s'il n'y a pas d'ambiguïté (ce qui est seulement le cas si un caractère spécial est adjacent à des jetons d'autres types).

De plus, des *commentaires* peuvent se trouver dans l'entrée SQL. Ce ne sont pas des jetons, ils sont réellement équivalents à un espace blanc.

Par exemple, ce qui suit est (syntaxiquement) valide pour une entrée SQL :

```
SELECT * FROM MA_TABLE;
UPDATE MA_TABLE SET A = 5;
INSERT INTO MA_TABLE VALUES (3, 'salut ici');
```

C'est une séquence de trois commandes, une par ligne (bien que cela ne soit pas requis ; plusieurs commandes peuvent se trouver sur une même ligne et une commande peut se répartir sur plusieurs lignes).

3.3.1 - Identifieurs et mots clés

Les jetons tels que SELECT, UPDATE ou VALUES dans l'exemple ci-dessus sont des exemples de *mots clés*, c'est-à-dire des mots qui ont une signification dans le langage SQL. Les jetons MA_TABLE et A sont des exemples d'*identifieurs*. Ils identifient des noms de tables, colonnes ou d'autres objets de la base de données suivant la commande qui a été utilisée. Du coup, ils sont quelques fois simplement nommés des « noms ». Une liste complète des mots clé est disponible dans l'Annexe.

Les identifieurs et les mots clés SQL doivent commencer avec une lettre ou un tiret bas (_). Les caractères suivants dans un identifieur ou dans un mot clé peuvent être des lettres, des tirets-bas, des chiffres.

L'identifieur et les noms de mots clés sont insensibles à la casse. Du coup :

```
UPDATE MA_TABLE SET A = 5;
```

peut aussi s'écrire de cette façon :

```
uPDaTE ma_Table SeT a = 5;
```

Une convention couramment utilisée revient à écrire les mots clés en majuscule et les noms en minuscule, c'est-à-dire :

```
UPDATE ma_table SET a = 5;
```

Voici un deuxième type d'identifieur : l'*identifieur délimité* ou l'*identifieur entre guillemets*. Il est formé en englobant une séquence arbitraire de caractères entre des guillemets doubles ("). Un identifieur délimité est toujours un identifieur, jamais un mot clé. Donc, "select" pourrait être utilisé pour faire référence à une colonne ou à une table nommée « select », alors qu'un select sans guillemets sera pris pour un mot clé et du coup, pourrait provoquer une erreur d'analyse lorsqu'il est utilisé alors qu'un nom de table ou de colonne est attendu. L'exemple peut être écrit avec des identifieurs entre guillemets comme ceci :

```
UPDATE "ma_table" SET "a" = 5;
```

Mettre un identifieur entre guillemets le rend sensible à la casse alors que les noms sans guillemets sont toujours convertis en minuscules. Par exemple, les identifieurs FOO, foo et "foo" sont considérés identiques par PostgreSQL™ mais "Foo" et "FOO" sont différents des trois autres et entre eux.

3.3.2 - Constantes

Il existe trois *types implicites de constantes* dans PostgreSQL™ : les chaînes, les chaînes de bits et les nombres.

Constantes de chaînes

Une constante de type chaîne en SQL est une séquence arbitraire de caractères entourée par des guillemets simples ('), c'est-à-dire 'Ceci est une chaîne'. Pour inclure un guillemet simple dans une chaîne constante, saisissez deux guillemets simples adjacents, par exemple 'Le cheval d"Anne'. Notez que ce n'est *pas* au guillemet double (").

Deux constantes de type chaîne séparées par un espace blanc *avec au moins une nouvelle ligne* sont concaténées et traitées réellement comme si la chaîne avait été écrite dans une constante. Par exemple :

```
SELECT 'foo'  
'bar';
```

est équivalent à :

```
SELECT 'foobar';
```

mais :

```
SELECT 'foo' 'bar';
```

n'a pas une syntaxe valide.

PostgreSQL™ accepte aussi les constantes de chaîne d'« échappement » qui sont une extension au standard SQL. Une constante de type chaîne d'échappement est indiquée en écrivant la lettre E (en majuscule ou minuscule) juste avant le guillemet d'ouverture, par exemple E'foo'. (Pour continuer une constante de ce type sur plusieurs lignes, écrire E seulement avant le premier guillemet d'ouverture). \b est un antislash, \f est un saut de page \n est un retour à la ligne, \t est une tabulation. Tout autre caractère suivi d'un antislash est pris littéralement. Du coup, pour inclure un caractère antislash, écrivez deux antislashes (\\). De plus, un guillemet simple peut être inclus dans une chaîne d'échappement en écrivant \', en plus de la façon normale ".

Le caractère de code zéro ne peut être placé dans une constante de type chaîne.

Constantes numériques

Les constantes numériques sont acceptées dans ces formes générales :

```
chiffres
chiffres.[chiffres][e[+-]chiffres]
[chiffres].chiffres[e[+-]chiffres]
chiffrese[+-]chiffres
```

où chiffres est un ou plusieurs chiffres décimaux (de 0 à 9). Au moins un chiffre doit être avant ou après le point décimal, s'il est utilisé. Au moins un chiffre doit suivre l'indicateur d'exponentiel (e), s'il est présent. Il peut ne pas y avoir d'espaces ou d'autres caractères imbriqués dans la constante. Notez que tout signe plus ou moins en avant n'est pas forcément considéré comme faisant part de la constante ; il est un opérateur appliqué à la constante.

Voici quelques exemples de constantes numériques valides :

```
42
3.5
4.
.001
5e2
1.925e-3
```

3.3.3 - Valeur NULL

La valeur nulle est une valeur particulière qu'on peut trouver dans une colonne d'une table. Nulle signifie ici que l'information est manquante. C'est ce qui se produira, par exemple, si la valeur est inconnue ou si elle n'est pas du type requis. La manière dont les valeurs nulles sont représentées dépend de l'implantation. Le système doit toutefois pouvoir distinguer les valeurs nulles des valeurs « connues », c'est-à-dire non nulles. Voici comment les valeurs nulles sont traitées dans les opérations arithmétique et scalaires:

- Soient A et B des objets de type numérique. Si A est nul, alors les expressions +A et -A prendront valeur nulle. Si l'un des objets A ou B est nul, ou si les deux sont nuls, alors les expressions suivantes sont nulles : A+B, A-B, A*B et A/B.
- Soient A et B des objets comparables entre eux. Si au moins un des deux est nul alors, à l'intérieur d'une clause WHERE ou HAVING, chacune des expressions A=B, A<>B, A<B, A>B, A<=B et A>=B prend la valeur logique inconnue, c'est-à-dire ni vraie ni fausse. Voici comment les tables de vérité définissent la valeur logique inconnue (V=vrai, F=faux, ? = inconnu):

ET	V	?	F
V	V	?	F
?	?	?	F
F	F	F	F

OU	V	?	F
V	V	V	V
?	V	?	?
F	V	?	F

NON	
V	F
?	?
F	V

Deux opérateurs particuliers du langage SQL, IS NULL et IS NOT NULL, permettent de vérifier si une valeur est nulle.

- Toutefois, deux valeurs nulles sont considérées comme égales, ou encore comme des copies réciproques, lorsqu'on veut éliminer les valeurs redondantes (DISINCT), et lorsqu'on désire faire des regroupements (GROUP BY) ou des tris (ORDER BY).

3.3.4 - Opérateurs

Un nom d'opérateur est une séquence provenant de la liste suivante :

+ - * / < > = ~ ! @ # % ^ & | ` ?

Néanmoins, il existe quelques restrictions sur les noms d'opérateurs :

- -- et /* ne peuvent pas apparaître quelque part dans un nom d'opérateur car ils seront pris pour le début d'un commentaire.
- Un nom d'opérateur à plusieurs caractères ne peut pas finir avec + ou -, sauf si le nom contient aussi un de ces caractères :

~ ! @ # % ^ & | ` ?

Par exemple, @- est un nom d'opérateur autorisé mais *- ne l'est pas. Cette restriction permet à PostgreSQL™ d'analyser des requêtes compatibles avec SQL sans requérir des espaces entre les jetons.

3.3.5 - Commentaires

Un commentaire est une séquence arbitraire de caractères commençant avec deux tirets et s'étendant jusqu'à la fin de la ligne, par exemple :

```
-- Ceci est un commentaire standard en SQL
```

Autrement, les blocs de commentaires style C peuvent être utilisés :

```
/* commentaires multilignes
 * et imbriqués: /* bloc de commentaire imbriqué */
 */
```

où le commentaire commence avec /* et s'étend jusqu'à l'occurrence de */. Ces blocs de commentaires s'imbriquent, comme spécifié dans le standard SQL mais pas comme dans le langage C. De ce fait, vous pouvez commenter des blocs importants de code pouvant contenir des blocs de commentaires déjà existants.

3.3.6 - Précédence lexicale

Le tableau 4 affiche la précédence et l'associativité des opérateurs dans PostgreSQL™.

Opérateur/Élément	Associativité	Description
.	gauche	séparateur de noms de table et de colonne
::	gauche	conversion de type, style PostgreSQL™
[]	gauche	sélection d'un élément d'un tableau
-	droite	négation unaire
^	gauche	exponentiel
* / %	gauche	multiplication, division, modulo
+ -	gauche	addition, soustraction
IS		IS TRUE, IS FALSE, IS UNKNOWN, IS NULL
ISNULL		test pour NULL
NOTNULL		test pour non NULL
(autres)	gauche	tout autre opérateur natif et défini par l'utilisateur
IN		appartenance à un ensemble
BETWEEN		compris entre
OVERLAPS		surcharge un intervalle de temps
LIKE ILIKE SIMILAR		correspondance de modèles de chaînes
< >		inférieur, supérieur à
=	droite	égalité, affectation
NOT	droite	négation logique
AND	gauche	conjonction logique
OR	gauche	disjonction logique

Tableau 4: Précédence des opérateurs (en ordre décroissant)

4 - PgAdmin III

L'utilisateur **Administrateur** est le seul qui a le droit de modifier l'installation : vous n'avez pas le droit de l'utiliser. Les autres utilisateurs sont munis de mots de passe transparents : le mot de passe est exactement le nom de l'utilisateur, en respectant majuscules et minuscules.

Chaque utilisateur dispose d'un bureau adapté aux tâches qu'il est censé accomplir et le répertoire \ mes documents dépend de l'utilisateur connecté. Vous devez vous connecter comme utilisateur bdX (X étant le numéro indiqué sur le bord de votre écran).

Nous utiliserons le langage PostgreSQL via l'interface pgAdmin III. PostgreSQL est un logiciel libre d'administration de la base de données. pgAdmin III est conçu pour répondre à la plupart des besoins, depuis l'écriture de simples requêtes SQL jusqu'au développement de bases de données complexes. L'interface graphique supporte les fonctionnalités de PostgreSQL les plus récentes et fait de l'administration un jeu d'enfant. Il est disponible pour plusieurs systèmes d'exploitation, y compris MS Windows, GNU/Linux et FreeBSD. Le site est http://www.pgadmin.org/?locale=fr_FR

4.1 - Introduction

L'objectif est l'utilisation d'une base de données pour gérer une liste de films avec l'année de sortie ou de réalisation ainsi que le genre du film un peu comme IMDB

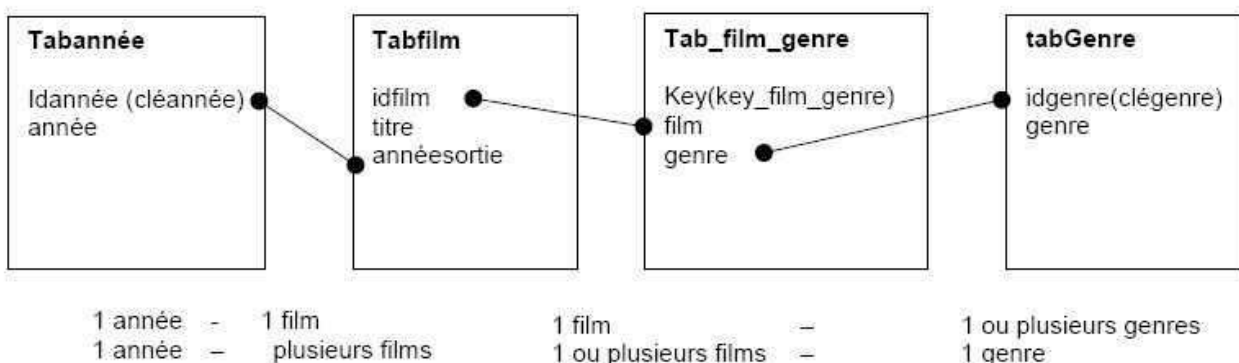
Le nom donné à cette base sera AMDB.

Hypothèses

Un film a un seul titre, une seule année de sortie et peut avoir un ou plusieurs genres. Trois tableaux seront utilisés : un comportant le nom des films, un autre les années et un troisième les genres.

Le tableau année pourra servir à d'autres relations avec d'autres tableaux dans le cas d'une extension de la base. Il faudra créer un 4ème tableau pour les relations multiples entre les films et les genres (Tab_film_genre).

Relations entre les tables



Pour ajouter à votre serveur la base de donnée AMDB, faites un clic droit sur le nom de votre serveur dans le navigateur objet.

Une fenêtre s'ouvre : indiquez le nom de votre nouvelle base de données puis validez.

4.2 - Création de table

Dans AMDB/Schémas/public/ faites un clique droit sur Table pour créer votre 1ère table.

Puis :

- Ajoutez d'une table tabgenre (indiquez juste son nom)
- Sélectionnez la table dans le navigateur d'objet
- Ajoutez des colonnes :
 - **idgenre** : sélectionnez le type de données sérial pour l'incrémement automatique, valider la commande par ok
 - **genre** : le type de données est caractère non nul, avec un maximum de 40 caractères
- Créez une clé primaire : dans la partie contrainte sélectionnée « Ajouter une clé primaire... »
 - **clégenre** : sélectionnez comme colonne idgenre correspondant à clé primaire.
- Visualisez votre première table : clique droit sur votre table « Afficher les données/Visualiser les 100 premières lignes ». Il n'y a rien , seules les colonnes apparaissent.

Notez que lors de la création de la table, vous pouvez directement ajouter les colonnes et les contraintes.

- Le panneau sql affiche les commandes, pour la colonne idgenre nous observons :

```
-- Column: idgenre
-- ALTER TABLE tabgenre DROP COLUMN idgenre;

ALTER TABLE tabgenre ADD COLUMN idgenre integer;
ALTER TABLE tabgenre ALTER COLUMN idgenre SET STORAGE PLAIN;
ALTER TABLE tabgenre ALTER COLUMN idgenre SET NOT NULL;
ALTER TABLE tabgenre ALTER COLUMN idgenre SET DEFAULT nextval('tabgenre_idgenre_seq'::regclass);
```

Créez la table tabannée qui a :

- 2 colonnes :
 - **idannée** de type serial,
 - **année** de type entier,
- et une contrainte **cléannée** clé primaire sur le colonne idannée;

et enfin la table tabfilm qui a :

- 3 colonnes :
 - **idfilm** : type serial,
 - **titre** : type 40 caractères,
 - **annéesortie** : type entier (clé étrangère, pointe sur la tabannée).
- une contrainte **cléfilm** de clé primaire sur la colonne idfilm

4.3 - Créer une clé étrangère

4.3.1 - Relation 1-n entre deux tables

Entre film et année : nous avons une seule année de sortie pour un film et plusieurs films pour une année. C'est pourquoi, nous avons créer une colonne du type **annéesortie** dans **tabfilm**.

La colonne **annéesortie** de la table **tabfilm** va pointer sur la **cléannée** de **tabannée** type de relation **n – 1**. La clé étrangère est créée dans la table ou il y a « plusieurs ». Pour cela créer une contrainte clé étrangère dont le nom sera **keyannée** et la référence la table **tabannée**.

Notons que les colonnes **Annéesortie** et **cléannée** sont de même type (entier).

4.3.2 - Relation n-n entre deux tables

Entre film et genre : nous avons plusieurs genres pour un film et plusieurs films pour un genre. C'est une relation n-n nous avons donc besoin d'une table intermédiaire.

Pour cela :

Créez une table intermédiaire ou de jointure **tab_film_genre** avec :

- 3 colonnes :
 - key : de type serial
 - film : de type entier
 - genre : de type entier
- 3 contraintes :
 - une clé primaire sur la colonne key
 - une clé étrangère ayant pour nom **titre**, référence la table **tabfilm** et dont la colonne locale est **film** référence par la clé **idfilm** de **tabfilm**.
 - une clé étrangère ayant pour nom **genre**, référence la table **tabgenre** et dont la colonne locale est **genre** référence par la clé **idgenre** de **tabgenre**.

4.4 - Insérer les données dans les tables

Pour insérer des données dans une table nous pouvons utiliser la fonction script : cliquer sur l'icône SQL.

Pour insérer de nouvelle donnée, nous écrivons dans la nouvelle fenêtre des requêtes SQL. On indique le nom du tableau et ensuite les valeurs dans l'ordre des colonnes

```
idgenre = 1 et genre = comédie INSERT INTO tabgenre VALUES ( 1,'comédie');
```

ou uniquement la valeur de genre, idgenre sera automatiquement incrémenté :

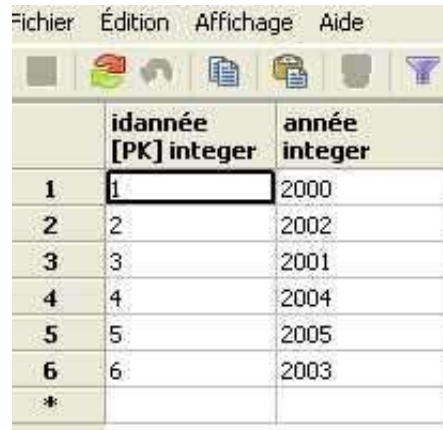
```
INSERT INTO tabgenre(genre) VALUES ( 'drame');
```

En visualisant la table **tabgenre**, vous pouvez observer vos deux nouvelles lignes.

Ajouter les genres suivants : policier, horreur, aventure, histoire et fict

ion'.

Pour la table tabannée, obtenez la table suivante :



The screenshot shows a database application window with a menu bar (Fichier, Édition, Affichage, Aide) and a toolbar. Below the toolbar is a table with two columns: 'idannée' and 'année'. The 'idannée' column is marked as a primary key [PK] and has the data type 'integer'. The 'année' column has the data type 'integer'. The table contains six rows of data, with the first row highlighted. The first row has '1' in the 'idannée' column and '2000' in the 'année' column. The second row has '2' and '2002', the third has '3' and '2001', the fourth has '4' and '2004', the fifth has '5' and '2005', and the sixth has '6' and '2003'. A row with an asterisk '*' is at the bottom.

	idannée [PK] integer	année integer
1	1	2000
2	2	2002
3	3	2001
4	4	2004
5	5	2005
6	6	2003
*		

Pour la table **tab_film** ajoutez les lignes suivantes :

```
INSERT INTO tabfilm(titre,annéesortie) VALUES ('Forge', 3);  
INSERT INTO tabfilm(titre,annéesortie) VALUES ('Star', 4);  
INSERT INTO tabfilm(titre,annéesortie) VALUES ('histoire', 2);  
INSERT INTO tabfilm(titre,annéesortie) VALUES ('Soleil', 1);
```

Observez que la valeur de l'**annéesortie** est une clé de la table **tabannée**.

Pour la table **tab_film_genre** ajoutez les lignes suivantes :

```
INSERT INTO tab_film_genre(film,genre) VALUES ( 4,2);  
INSERT INTO tab_film_genre(film,genre) VALUES ( 1,1);  
INSERT INTO tab_film_genre(film,genre) VALUES ( 1,4);  
INSERT INTO tab_film_genre(film,genre) VALUES ( 3,2);  
INSERT INTO tab_film_genre(film,genre) VALUES ( 2,3);  
INSERT INTO tab_film_genre(film,genre) VALUES ( 2,6);  
INSERT INTO tab_film_genre(film,genre) VALUES ( 1,1);
```

4.5 - Requêtes

4.5.1 - Afficher le contenu d'une seule table

Affichez via des requêtes SQL, les informations suivantes :

- la table tabannée
- la colonne année de la table tabannée
- les année supérieur à 2001 de la table tabannée
- la table année trier en ordre chronologique
- la colonne genre de la table tabgenre en ordre alphabétique
- le nombre de film de la table tabfilm

4.5.2 - Afficher le contenu de deux tables liées par une clé étrangère

Affichez via des requêtes SQL, les informations suivantes :

- l'ensemble des films avec leur année de sortie
- l'ensemble des films avec leur année de sortie, trié sur l'année.
- L'ensemble des films avec leur année de sortie, sortis en 2002.
- l'ensemble des films avec leur année de sortie, sortie depuis 2001.

4.5.3 - Afficher le contenu de deux tables liées par une table intermédiaire de type n-n

Affichez via des requêtes SQL, les informations suivantes :

- l'ensemble des films et de leurs genres, triés en ordre alphabétique des films;
- l'ensemble des films et de leurs genre, triés en ordre alphabétique des genres;
- l'ensemble des films dont le genre est drame;
- le nombre de film dont le genre est drame;

5 - Annexe

5.1 - Commandes SQL

ABORT — Interrompre la transaction en cours

ALTER AGGREGATE — Modifier la définition d'une fonction d'agrégat

ALTER CONVERSION — Modifier la définition d'une conversion

ALTER DATABASE — Modifier une base de données

ALTER DOMAIN — Modifier la définition d'un domaine

ALTER FUNCTION — Modifier la définition d'une fonction

ALTER GROUP — Modifier le nom d'un rôle ou la liste de ses membres

ALTER INDEX — Modifier la définition d'un index

ALTER LANGUAGE — Modifier la définition d'un langage procédural

ALTER OPERATOR — Modifier la définition d'un opérateur

ALTER OPERATOR CLASS — Modifier la définition d'une classe d'opérateur

ALTER OPERATOR FAMILY — Modifier la définition d'une famille d'opérateur

ALTER ROLE — Modifier un rôle de base de données

ALTER SCHEMA — Modifier la définition d'un schéma

ALTER SEQUENCE — Modifier la définition d'un générateur de séquence

ALTER TABLE — Modifier la définition d'une table

ALTER TABLESPACE — Modifier la définition d'un tablespace

ALTER TEXT SEARCH CONFIGURATION — modifier la définition d'une configuration de recherche plein texte

ALTER TEXT SEARCH DICTIONARY — modifier la définition d'un dictionnaire de recherche plein texte

ALTER TEXT SEARCH PARSER — modifier la définition d'un analyseur de recherche plein texte

ALTER TEXT SEARCH TEMPLATE — modifier la définition d'un modèle de recherche plein texte

ALTER TRIGGER — Modifier la définition d'un déclencheur

ALTER TYPE — Modifier la définition d'un type

ALTER USER — Modifier un rôle de la base de données

ALTER VIEW — modifier la définition d'une vue

ANALYZE — Collecter les statistiques d'une base de données

BEGIN — Débuter un bloc de transaction

CHECKPOINT — Forcer un point de vérification dans le journal des transactions

CLOSE — Fermer un curseur

CLUSTER — Réorganiser une table en fonction d'un index

COMMENT — Définir ou modifier le commentaire associé à un objet

COMMIT — Valider la transaction en cours

COMMIT PREPARED — Valider une transaction préalablement préparée en vue d'une validation en deux phases

COPY — Copier des données depuis/vers un fichier vers/ depuis une table

CREATE AGGREGATE — Définir une nouvelle fonction d'agrégat

CREATE CAST — Définir un transtypage

CREATE CONSTRAINT TRIGGER — Définir un nouveau déclencheur sur contrainte

CREATE CONVERSION — Définir une nouvelle conversion d'encodage

CREATE DATABASE — Créer une nouvelle base de données

CREATE DOMAIN — Définir un nouveau domaine

CREATE FUNCTION — Définir une nouvelle fonction
CREATE GROUP — Définir un nouveau rôle de base de données
CREATE INDEX — Définir un nouvel index
CREATE LANGUAGE — Définir un nouveau langage procédural
CREATE OPERATOR — Définir un nouvel opérateur
CREATE OPERATOR CLASS — Définir une nouvelle classe d'opérateur
CREATE OPERATOR FAMILY — définir une nouvelle famille d'opérateur
CREATE ROLE — Définir un nouveau rôle de base de données
CREATE RULE — Définir une nouvelle règle de réécriture
CREATE SCHEMA — Définir un nouveau schéma
CREATE SEQUENCE — Définir un nouveau générateur de séquence
CREATE TABLE — Définir une nouvelle table
CREATE TABLE AS — Définir une nouvelle table à partir des résultats d'une requête
CREATE TABLESPACE — Définir un nouvel tablespace
CREATE TEXT SEARCH CONFIGURATION — définir une nouvelle configuration de recherche plein texte
CREATE TEXT SEARCH DICTIONARY — définir un dictionnaire de recherche plein texte
CREATE TEXT SEARCH PARSER — définir un nouvel analyseur de recherche plein texte
CREATE TEXT SEARCH TEMPLATE — définir un nouveau modèle de recherche plein texte
CREATE TRIGGER — Définir un nouveau déclencheur
CREATE TYPE — Définir un nouveau type de données
CREATE USER — Définir un nouveau rôle de base de données
CREATE VIEW — Définir une vue
DEALLOCATE — Désaffecter (libérer) une instruction préparée
DECLARE — Définir un curseur
DELETE — Supprimer des lignes d'une table
DISCARD — Annuler l'état de la session
DROP AGGREGATE — Supprimer une fonction d'agrégat
DROP CAST — Supprimer un transtypage
DROP CONVERSION — Supprimer une conversion
DROP DATABASE — Supprimer une base de données
DROP DOMAIN — Supprimer un domaine
DROP FUNCTION — Supprimer une fonction
DROP GROUP — Supprimer un rôle de base de données
DROP INDEX — Supprimer un index
DROP LANGUAGE — Supprimer un langage procédural
DROP OPERATOR — Supprimer un opérateur
DROP OPERATOR CLASS — Supprimer une classe d'opérateur
DROP OPERATOR FAMILY — Supprimer une famille d'opérateur
DROP OWNED — Supprimer les objets de la base possédés par un rôle
DROP ROLE — Supprimer un rôle de base de données
DROP RULE — Supprimer une règle de réécriture
DROP SCHEMA — Supprimer un schéma
DROP SEQUENCE — Supprimer une séquence
DROP TABLE — Supprimer une table
DROP TABLESPACE — Supprimer un tablespace
DROP TEXT SEARCH CONFIGURATION — Supprimer une configuration de recherche plein texte
DROP TEXT SEARCH DICTIONARY — Supprimer un dictionnaire de recherche plein texte
DROP TEXT SEARCH PARSER — Supprimer un analyseur de recherche plein texte
DROP TEXT SEARCH TEMPLATE — Supprimer un modèle de recherche plein texte
DROP TRIGGER — Supprimer un déclencheur

DROP TYPE — Supprimer un type de données
DROP USER — Supprimer un rôle de base de données
DROP VIEW — Supprimer une vue
END — Valider la transaction en cours
EXECUTE — Exécuter une instruction préparée
EXPLAIN — Afficher le plan d'exécution d'une instruction
FETCH — Récupérer les lignes d'une requête à l'aide d'un curseur
GRANT — Définir les droits d'accès
INSERT — Insérer de nouvelles lignes dans une table
LISTEN — Attendre une notification
LOAD — Charger ou décharger une bibliothèque partagée
LOCK — verrouiller une table
MOVE — positionner un curseur
NOTIFY — engendrer une notification
PREPARE — prépare une instruction pour exécution
PREPARE TRANSACTION — prépare la transaction en cours pour une validation en deux phases
REASSIGN OWNED — Modifier le propriétaire de tous les objets de la base appartenant à un rôle spécifique
REINDEX — reconstruit les index
RELEASE SAVEPOINT — détruit un point de sauvegarde précédemment défini
RESET — réinitialise un paramètre d'exécution à sa valeur par défaut
REVOKE — supprime les droits d'accès
ROLLBACK — annule la transaction en cours
ROLLBACK PREPARED — annule une transaction précédemment préparée en vue d'une validation en deux phases
ROLLBACK TO SAVEPOINT — annule les instructions jusqu'au point de sauvegarde
SAVEPOINT — définit un nouveau point de sauvegarde à l'intérieur de la transaction en cours
SELECT — récupère des lignes d'une table ou d'une vue
SELECT INTO — définit une nouvelle table à partir des résultats d'une requête
SET — change un paramètre d'exécution
SET CONSTRAINTS — initialise le mode de vérification de contrainte de la transaction en cours
SET ROLE — initialise l'identifiant utilisateur courant de la session en cours
SET SESSION AUTHORIZATION — Initialise l'identifiant de session de l'utilisateur et l'identifiant de l'utilisateur actuel de la session en cours
SET TRANSACTION — initialise les caractéristiques de la transaction actuelle
SHOW — affiche la valeur d'un paramètre d'exécution
START TRANSACTION — débute un bloc de transaction
TRUNCATE — vide une table ou un ensemble de tables
UNLISTEN — arrête l'écoute d'une notification
UPDATE — mettre à jour les lignes d'une table
VACUUM — récupère l'espace inutilisé et, optionnellement, analyse une base
VALUES — calcule un ensemble de lignes

5.2 - Applications client de PostgreSQL

clusterdb — Grouper une base de données PostgreSQL™
createdb — Créer une nouvelle base de données PostgreSQL™
createlang — Définir un langage procédural sous PostgreSQL™
createuser — Définir un nouveau compte utilisateur PostgreSQL™
dropdb — Supprimer une base de données PostgreSQL™
droplang — Supprimer un langage procédural
dropuser — Supprimer un compte utilisateur PostgreSQL™
ecpg — Préprocesseur C pour le SQL embarqué
pg_config — récupérer des informations sur la version installée de PostgreSQL™
pg_dump — sauvegarder une base de données PostgreSQL™ dans un script ou tout autre fichier d'archive
pg_dumpall — extraire une grappe de bases de données PostgreSQL™ dans un fichier de script
pg_restore — restaure une base de données PostgreSQL™ à partir d'un fichier d'archive créé par **pg_dump**
psql — terminal interactif PostgreSQL™
reindexdb — reindexe une base de données PostgreSQL™
vacuumdb — récupère l'espace inutilisé et, optionnellement, analyse une base de données PostgreSQL™

5.3 - Applications relatives au serveur PostgreSQL

initdb — Créer un nouveau « cluster »
ipcclean — Supprimer la mémoire partagée et les sémaphores d'un serveur PostgreSQL™ victime d'un arrêt brutal
pg_controldata — afficher les informations de contrôle d'un groupe de bases de données PostgreSQL™
pg_ctl — démarrer, arrêter ou redémarrer le serveur PostgreSQL™
pg_resetxlog — réinitialiser les WAL et les autres informations de contrôle d'une grappe de bases de données PostgreSQL™
postgres — Serveur de bases de données PostgreSQL™
postmaster — Serveur de bases de données PostgreSQL™