Bases de Données

Sébastien Jeudy

(Version V3)

Bases de Données

Objectif et sommaire : *acquérir les connaissances essentielles de conception, création, manipulation et programmation de bases de données relationnelles*

I. Introduction aux bases de données

II. Modèle Conceptuel de Données (MCD)

III. Modèle Logique de Données (MLD)

IV. Modèle Physique de Données (MPD)

V. Serveur PostgreSQL

VI. SQL avec PostgreSQL

VII. Introduction à PL/pgSQL

VIII. Le langage PL/SQL

IX. Eléments avancés de PL/SQL et SQL

I. Introduction aux bases de données

1) Définition et historique
2) Caractéristiques
3) Méthodologies de conception et modélisations

II. Modèle Conceptuel de Données (MCD)

1) Dictionnaire des données
2) Schéma entités-associations
3) Associations particulières
4) Règles de normalisation
5) Méthodologie de base

III. Modèle Logique de Données (MLD)

1) Généralités
2) Tables, colonnes et lignes
3) Clés primaires et clés étrangères
4) Schéma relationnel
5) Traduction d'un MCD en un MLDR

IV. Modèle Physique de Données (MPD)

1) Généralités
2) Optimisations

V. Serveur PostgreSQL

1) Présentation
2) Schémas
3) Outils clients

VI. SQL avec PostgreSQL

1) La norme SQL. Positionnement de PostgreSQL. Création d'une base de données

2) Types de données PostgreSQL. Contraintes d'intégrité sur les tables

3) Tables. Séquences

4) Ajout, modification et suppression des données

5) Interrogation du schéma d'une base. Sélection, restriction, tri, jointure

6) Extractions complexes

7) Fonctions intégrées. Vues

8) Transactions

VII. Introduction à PL/pgSQL

1) Principes des procédures et fonctions
2) Quand utiliser le langage PL/SQL ?
3) Création, modification et suppression de fonctions

VIII. Le langage PL/SQL

1) Principes essentiels du langage. Structure d'un script complet. Les types de données

2) Expression et exécution automatiques de requêtes simples

3) Passage de paramètres

4) Les tests et les boucles

5) Manipulation avancée des curseurs

IX. Eléments avancés de PL/SQL et SQL

1) La gestion des erreurs par les exceptions
2) Les déclencheurs (triggers)
3) Sécurisation des fonctions
4) Les index. L'optimiseur. EXPLAIN

I. Introduction aux bases de données

I. Introduction aux bases de données

1) Définition et historique
2) Caractéristiques
3) Méthodologies de conception et modélisations

1) Définition et historique

Définition :

Une base de données (« BD » ou « BDD ») est un **lot d'informations stocké dans un dispositif informatique**

Les technologies existantes permettent d'organiser et de structurer une base de données de manière à pouvoir **manipuler facilement le contenu** et **stocker efficacement de très grandes quantités d'informations**

Le logiciel qui manipule les bases de données est appelé **système de gestion de bases de données (SGBD)**, généralement hébergé sur un **serveur de bases de données**

Historique :

1960 - 1965 : apparition des mémoires auxiliaires magnétiques (disques), développement de la théorie des fichiers

1965 - 1970 : premières bases de données à structure hiérarchique (IMS) et à structure réseau, recommandations du CODASYL

1970 - 1980 : commercialisation des bases de données du type précédent, développement de la théorie des bases de données relationnelles (Codd)

1980 - 1983 : implantation des premières bases de données relationnelles (même sur petits systèmes)

1983 - 1992 : nombreux développements sur les bases de données : théoriques (objets) ou pratiques (réseaux, bases réparties,...) voire conceptuels (infocentres, data warehouse,...)

Depuis les années 1990 : généralisation de l'utilisation des bases de données – principalement relationnelles – sur tout système (facilitée par la globalisation des communications réseau et les capacités de stockage)

Complément :

Fin des années 2000 : Big Data (données massives) & NoSQL

2) Caractéristiques

Exhaustivité des informations :

Les informations contenues dans une base de données doivent **être suffisamment complètes** pour que les applications prévues puissent fonctionner

Exemple : dans une entreprise commerciale, l'édition d'une facture suppose que la base de données contienne des informations sur

- le client (nom, adresse, conditions de vente,...)
- la commande (date, numéro, articles demandés, références, quantités,...)
- les articles (référence, quantité en stock, prix unitaire, taux de TVA,...)
- la livraison (date, articles livrés, articles restant à livrer,...)

Non-redondance des informations :

Dans la mesure du possible, **la même information ne doit pas figurer plusieurs fois** dans une base de données

Cela conduirait :

- à gaspiller de la place de stockage (espace disque)
- à effectuer des mises à jour complexes (ou d'en oublier...)

Toutefois, il n'est **pas toujours facile de ne pas dupliquer certaines informations** (c'est même parfois indispensable). Il conviendra donc de juger si une redondance est indispensable ou non

Partage des informations :

Les informations contenues dans une base de données doivent **être accessibles à plusieurs utilisateurs simultanés**

Mais problèmes :

- la sécurité des données (à partager ou non)
- les accès concurrents à une même information

Standard d'organisation :

Cet aspect est lié à la **centralisation qui se manifeste concrètement par l'autorité de l'administrateur de la base de données** (en anglais DBA : DataBase Administrator)

Celui-ci assure, entre autres tâches, la **standardisation des règles qui régissent la structure et le fonctionnement** de la base de données (notamment les opérations de maintenance)

Sécurité des informations :

Le DBA, et lui-seul, autorise l'accès aux informations, concrètement :

- l'**accès à une partie bien définie** de la base de données
- la **possibilité d'y effectuer des opérations bien définies**

Ces dispositions :

- **garantissent** la confidentialité des informations
- **limitent** le détournement, la modification frauduleuse ou accidentelle, ou encore la destruction des informations

Le problème de la sécurité n'est pas un problème simple (comme la lecture des journaux le démontre régulièrement)

Intégrité des informations :

Les informations contenues dans une base de données doivent **être exactes** si l'on veut utiliser celle-ci de manière efficace

Dans la pratique, **l'exactitude absolue (zéro défaut) n'existe pas**, on peut tout au plus améliorer la qualité d'intégrité

Dans ce but, le DBA définira des **systèmes de contrôle des informations (contraintes d'intégrité) après toute opération de mise à jour** (insertion, modification, suppression)

Ces systèmes, quoique performants, ne peuvent malheureusement éviter toutes les inexactitudes

Indépendance des informations :

Les méthodes de stockage et d'accès des données sont normalement indépendantes du schéma conceptuel. **Une réorganisation du schéma doit être transparente pour l'utilisateur**

Mais :

- **les bases de données actuelles les plus courantes ne possèdent pas complètement cette propriété** (qu'il faut donc considérer ici comme un objectif à atteindre plutôt qu'une réalité établie)

- les bases de données « orientées objets » manipulent à la fois des données et des traitements

3) Méthodologies de conception et modélisations

En définissant directement les tables d'une base de données dans un SGBD (Oracle, DB2, PostgreSQL, MySQL, SQL Server,...), nous sommes exposés à **deux types de problèmes :**

- nous ne savons pas toujours dans quelle table placer certaines colonnes

- nous avons du mal à prévoir les tables de jonctions intermédiaires

Il est donc **nécessaire de recourir à une étape préliminaire de conception et de modélisation**

La méthodologie Merise (Méthode d'Étude et de Réalisation Informatique pour les Systèmes d'Entreprises), élaborée en France dans les années 1970, **permet de concevoir un système d'information d'une façon standardisée et méthodique**

→ *essentiellement française mais tout en s'insérant dans le cadre d'une réflexion internationale autour du modèle relationnel d'Edgar Frank Codd*

Concernant la conception de bases de données proprement dite, celle-ci se base sur le schéma entités-associations via le **« Modèle Conceptuel de Données (MCD) »**, puis le schéma relationnel via le **« Modèle Logique de Données (MLD) »** et le **« Modèle Physique de Données (MPD) »**

Merise **propose également la modélisation des traitements et la gestion des projets** (mais non abordés ici)

II. Modèle Conceptuel de Données (MCD)

II. Modèle Conceptuel de Données (MCD)

1) Dictionnaire des données
2) Schéma entités-associations
3) Associations particulières
4) Règles de normalisation
5) Méthodologie de base

1) Dictionnaire des données

Le dictionnaire des données est le **résultat de la phase de collecte des données**. C'est la **première phase d'informatisation** d'un système d'information, également appelée recueil d'informations

Pendant la phase de conception, toutes les données recueillies sont donc répertoriées dans un même dictionnaire (exhaustif)

→ *outil important car référence de toutes les études effectuées ensuite*

Pour chaque donnée, ce dictionnaire doit contenir les informations suivantes (listées dans un tableau) :

nom fonctionnel, nom "technique" correspondant, type, longueur, contraintes / calculs (règles de gestion), éventuellement commentaires

Complément :

Exemple (partiel) :

Nom fonctionnel	Nom technique	Type	Longueur	Contraintes / Calculs
Nom personne	nom_pers	texte	20	non nul
Prenom personne	prenom_pers	texte	20	non nul
Date de naissance	date_nais	date	8	≤ date du jour
Age	age	entier	3	date du jour - date de naissance

2) Schéma entités-associations

Avant de réfléchir au schéma relationnel d'une base de données, il est bon de **modéliser la problématique à traiter d'un point de vue conceptuel et indépendamment du SGBD utilisé**

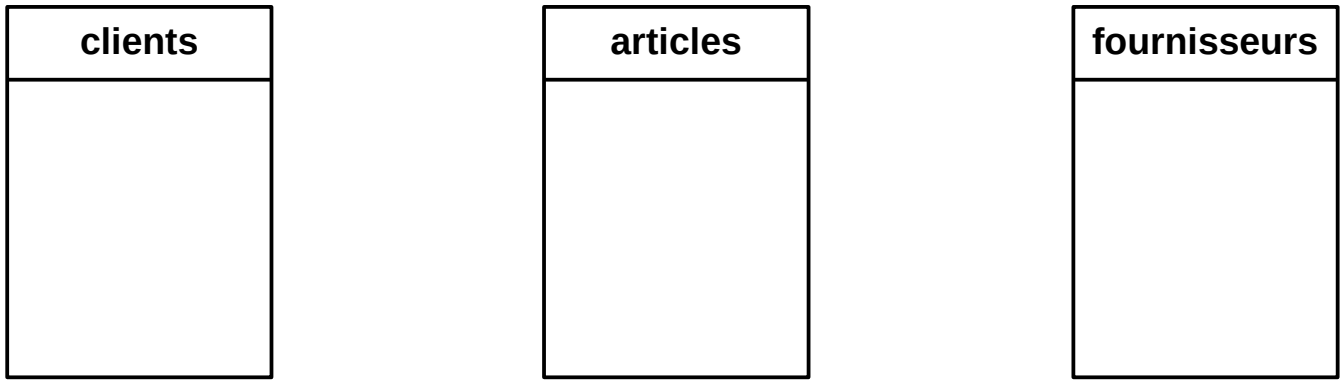
Avec la méthode Merise, **un MCD décrit le fonctionnement d'un système d'information d'un point de vue fonctionnel**, sans aucune considération technique, c'est-à-dire uniquement conceptuel

Cette modélisation conceptuelle conduit à l'élaboration d'un type de schéma très répandu, le **schéma entités-associations**

Une entité est une population d'individus homogènes

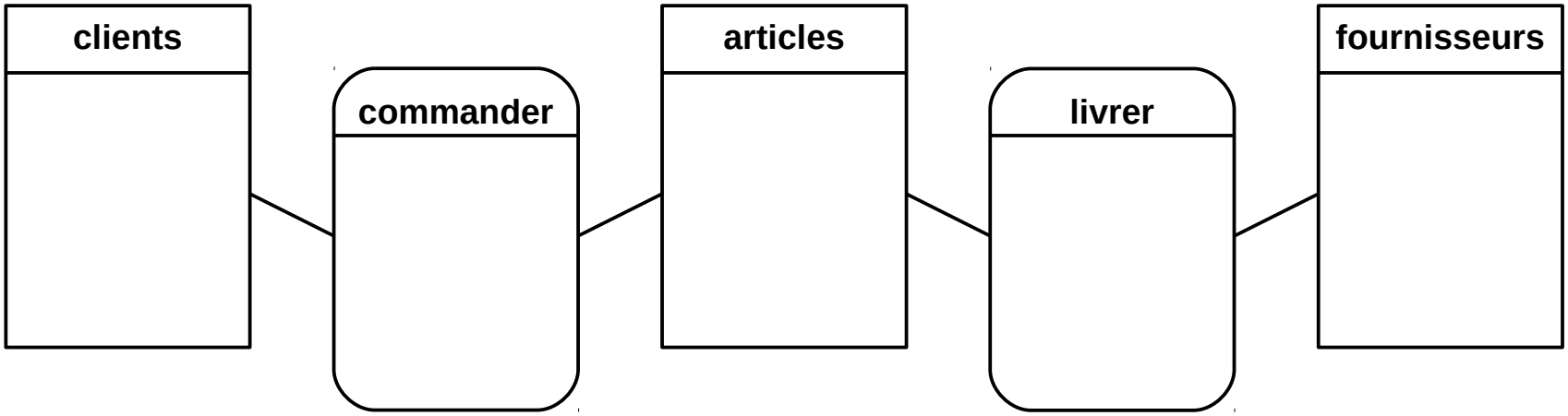
Par exemple, les produits ou **les articles vendus par une entreprise peuvent être regroupés dans une même entité « articles »**, car d'un article à l'autre, les informations ne changent pas de nature (à chaque fois, il s'agit de la désignation, du prix unitaire, etc)

Par contre, **les articles et les clients ne peuvent pas être regroupés** : leurs informations ne sont pas homogènes (un article ne possède pas d'adresse et un client ne possède pas de prix unitaire). Il faut donc leur réserver deux entités distinctes, l'entité « articles » et l'entité « clients » :



Une association (ou relation) est une liaison qui a une signification précise entre plusieurs entités

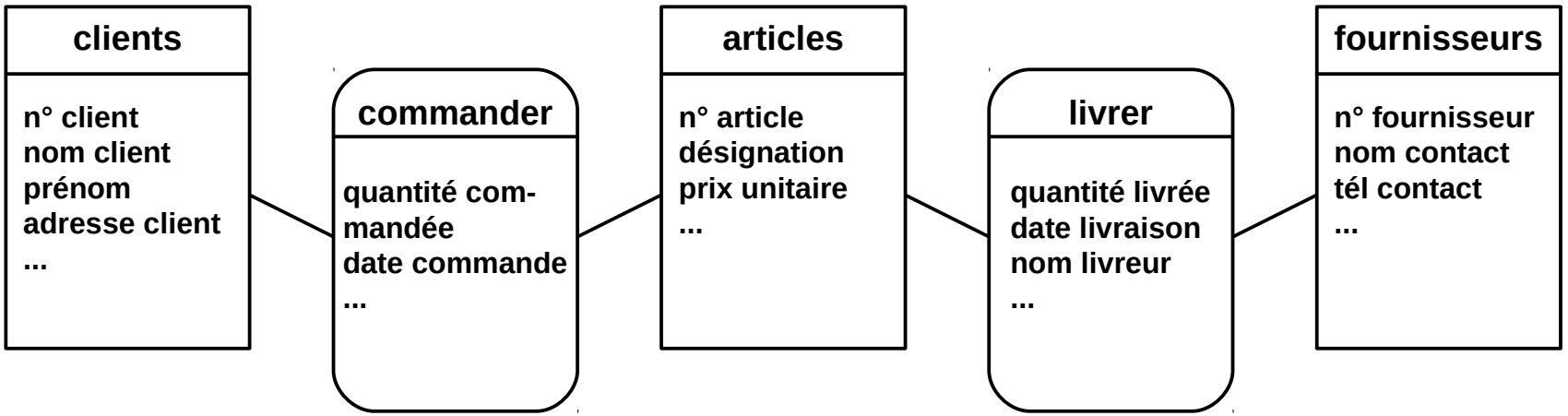
Dans notre exemple, **l'association « commander » est une liaison évidente entre les entités « articles » et « clients »**, tandis que l'association « livrer » établit le lien sémantique entre les entités « articles » et « fournisseurs » :



→ *les entités « clients » et « fournisseurs » ne sont pas liées directement, mais indirectement via l'entité « articles » (ce qui est naturel)*

Un attribut est une propriété d'une entité ou d'une association

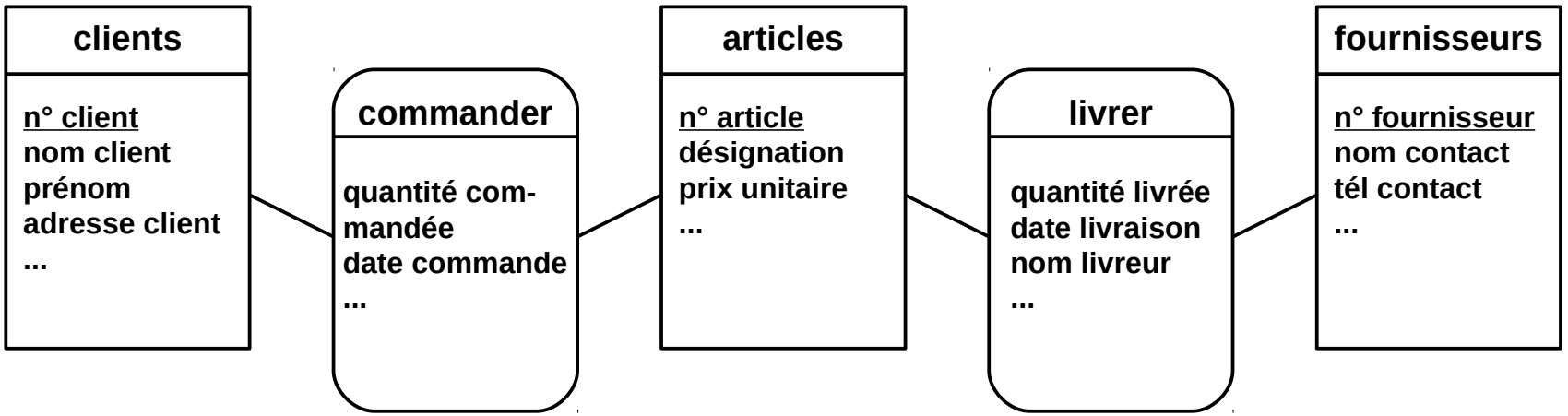
Dans notre exemple, **le prix unitaire est un attribut de l'entité « articles »**, le nom de client est un attribut de l'entité « clients », **la quantité commandée est un attribut de l'association « commander »** et la date de livraison est un attribut de l'association « livrer » :



→ pour une cohérence du modèle, une entité et ses attributs ne doivent traiter que d'un sujet (pas d'informations fournisseurs dans les articles)

Chaque individu (occurrence) d'une entité doit être identifiable de manière unique. C'est pourquoi, **toute entité doit posséder un attribut sans doublon** (c'est-à-dire ne prenant pas deux fois la même valeur)

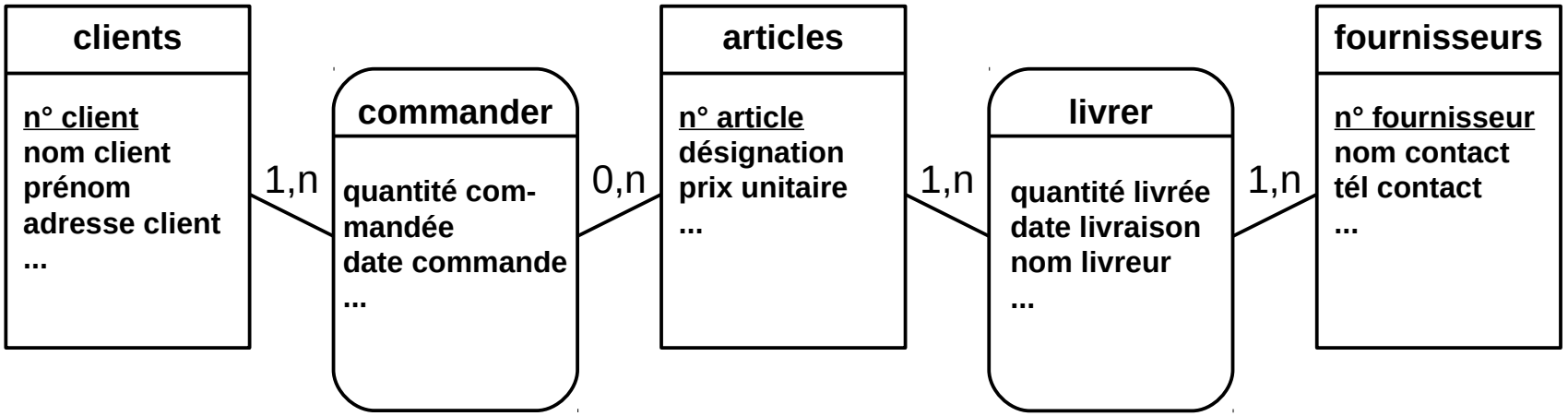
Il s'agit de l'**identifiant souligné sur le schéma**. Dans notre exemple, le numéro de client est un identifiant classique pour l'entité « clients » :



→ *une entité possède au moins un attribut (son identifiant), au contraire une association peut être dépourvue d'attribut et n'a pas d'identifiant*

La cardinalité d'un lien entre une entité et une association précise le minimum et le maximum de fois qu'un individu (occurrence) de l'entité peut être concerné par l'association

Exemple : un client a au moins commandé 1 article et peut commander n articles (n étant indéterminé), tandis qu'un article peut avoir été commandé entre 0 et n fois (pas forcément le même n). On obtient alors :



Une cardinalité minimale de 1 se justifie quand les individus de l'entité ont besoin de l'association pour exister (ici un client n'existe pas avant une commande). Dans tous les autres cas, la cardinalité minimale vaut 0 (c'est le cas pour une liste préétablie d'articles)

La discussion autour d'une cardinalité minimale de 0 ou 1 n'est vraiment intéressante que lorsque la cardinalité maximale vaut 1 (lors de la traduction du MCD en MLD, on ne peut pas faire la différence entre une cardinalité minimale de 0 ou 1 quand la cardinalité maximale vaut n)

Dans notre exemple, un article peut être commandé par plusieurs clients : tous les articles ne sont pas numérotés individuellement, mais portent un numéro d'article "collectif" et ce ne sont pas à proprement parler les mêmes articles à chaque fois (c'est un choix de modélisation, chaque article aurait pu être numéroté vraiment individuellement)

Complément :

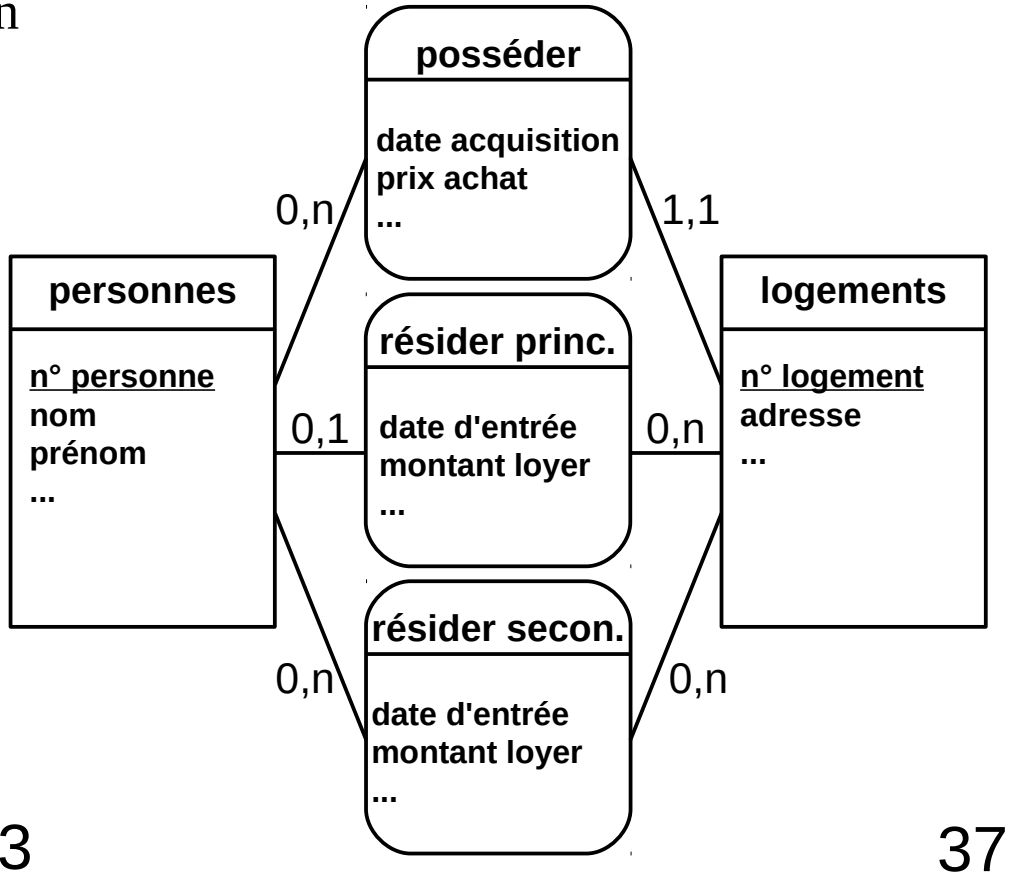
La seule difficulté est de se poser les questions cardinalités dans le bon sens, pour l'association « commander » :

- côté « clients », la question est "un client peut commander combien d'articles ?" et la réponse est "entre 1 et plusieurs"

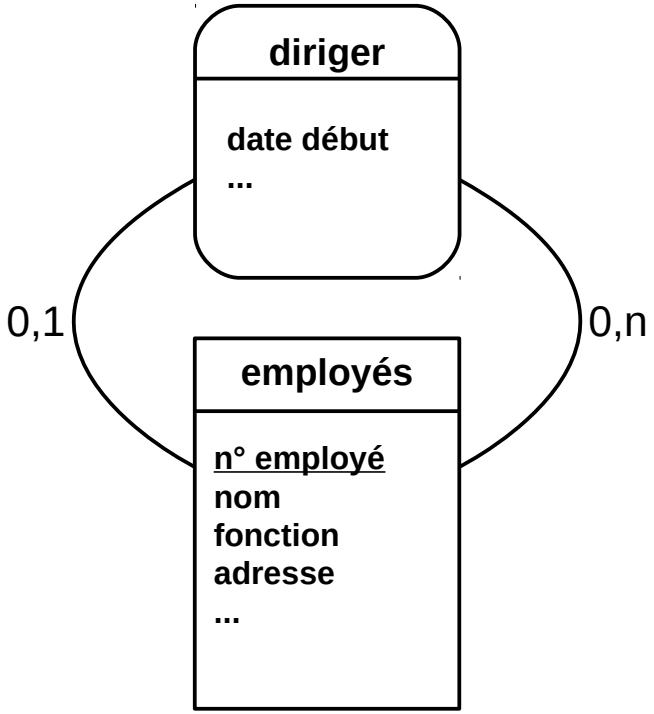
- côté « articles », la question est "un article peut être commandé par combien de clients ?" et cette fois-ci la réponse est "entre 0 et plusieurs"

3) Associations particulières

Associations plurielles : deux mêmes entités peuvent être plusieurs fois en association



Associations réflexives : une association peut être reliée plusieurs fois à la même entité



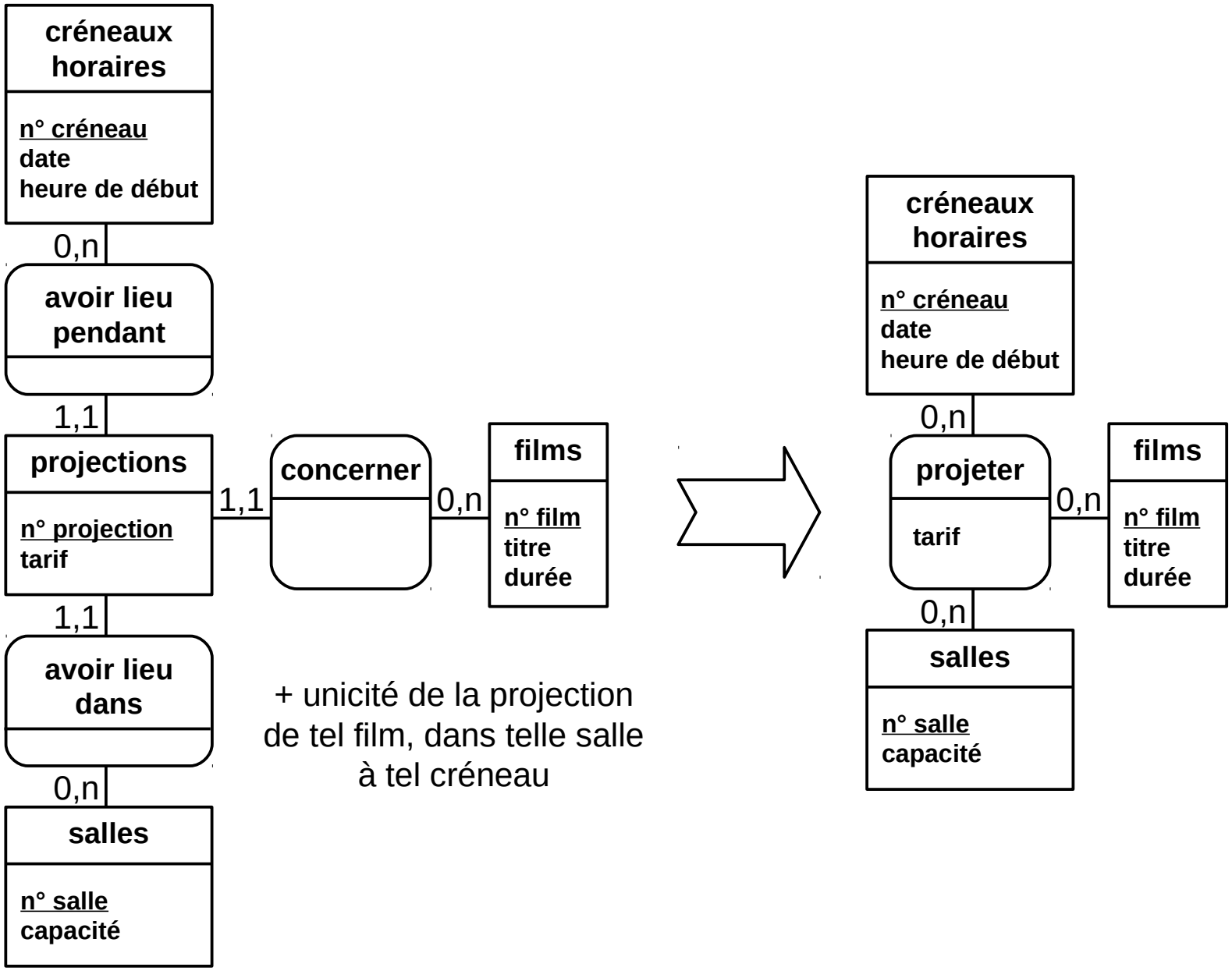
Dans cet exemple, tout employé est dirigé par un autre employé (sauf le directeur général) et un employé peut diriger plusieurs autres employés (ou aucun)

Associations non binaires :

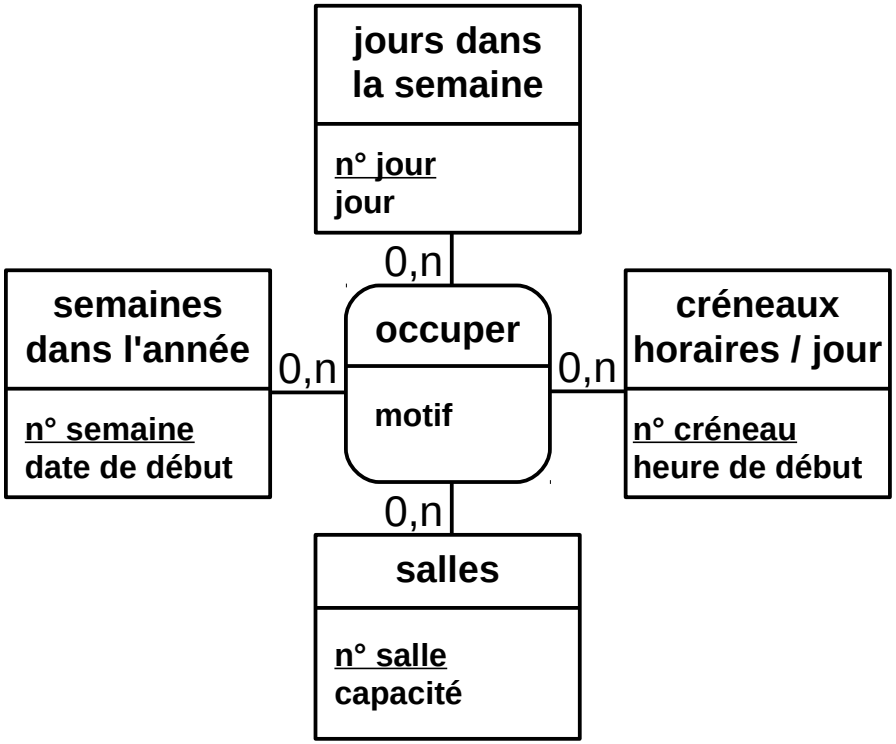
Lorsqu'autour d'une entité, toutes les associations ont pour cardinalités maximales 1 au centre et n à l'extérieur, cette entité peut être remplacée par une association reliée à toutes les entités voisines avec des cardinalités identiques 0,n

La deuxième condition qu'il faut impérativement satisfaire est la règle de normalisation des attributs des associations (voir plus loin) : cette règle conduit parfois à l'apparition d'associations qui établissent un lien sémantique entre 3 entités ou plus

Dans l'exemple suivant d'un cinéma, l'entité « projections » est uniquement entourée d'associations dont les cardinalités maximales sont 1 côté « projections » et n de l'autre côté. De plus, la donnée d'un créneau, d'un film et d'une salle suffit à déterminer une projection unique. On peut donc la remplacer par une association « projeter » directement reliée aux 3 entités « créneaux horaires », « films » et « salles », **on parle alors d'association ternaire**


créneaux horaires
n° créneau
date
heure de début
0,n
avoir lieu pendant
1,1
projections
n° projection
tarif
1,1
concerner
0,n
films
n° film
titre
durée
1,1
avoir lieu dans
0,n
salles
n° salle
capacité
+ unicité de la projection de tel film, dans telle salle à tel créneau
créneaux horaires
n° créneau
date
heure de début
0,n
projeter
tarif
0,n
films
n° film
titre
durée
0,n
salles
n° salle
capacité

Une association peut être liée à plus de 3 entités :



Le conseil pour être certain de la légitimité de cette association quaternaire (ou 4-aire) est de vérifier les cardinalités sur un schéma intermédiaire faisant apparaître, à la place, une entité « occupations » avec 4 associations binaires (comme précédemment)

4) Règles de normalisation

Un bon schéma entités-associations doit répondre à **plusieurs règles de normalisation que le concepteur doit respecter**

Normalisation des entités : toutes les entités qui sont remplaçables par une association doivent être remplacées (comme précédemment)

Normalisation des noms : le nom d'une entité, d'une association ou d'un attribut doit être unique

Conseils :

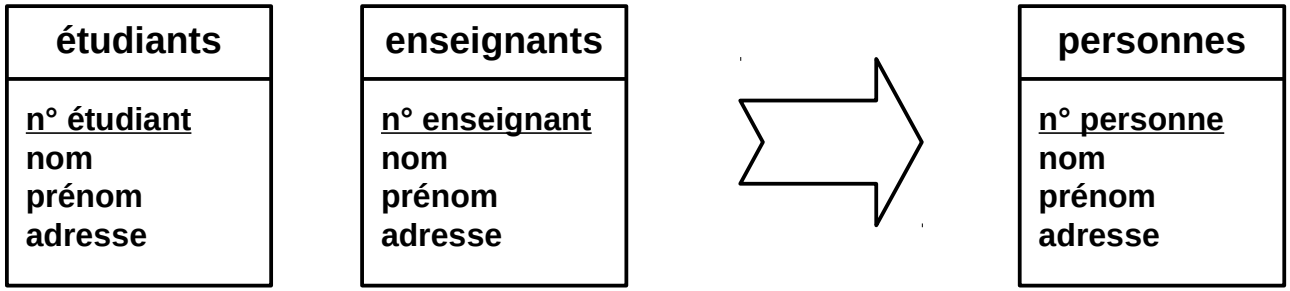
Pour les entités, utiliser un nom commun au pluriel (exemple : clients)

Pour les associations, utiliser un verbe à l'infinitif (exemples : effectuer, concerner) éventuellement à la forme passive (exemple : être commandé) et accompagné d'un adverbe (exemples : avoir lieu dans, pendant, à)

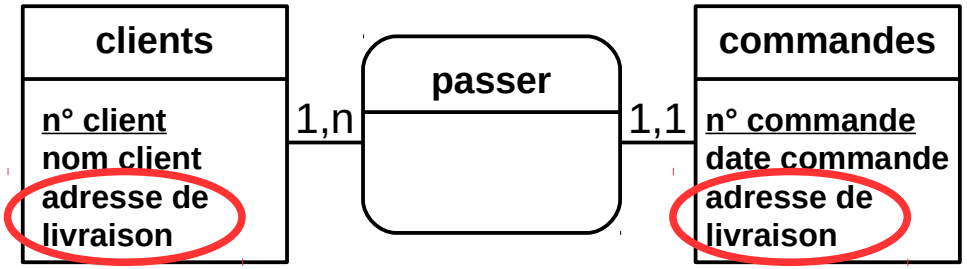
Pour les attributs, utiliser un nom commun singulier (exemples : nom, numéro, libellé, description) éventuellement accompagné du nom de l'entité ou de l'association dans laquelle il se trouve (exemples : nom de client, numéro d'article)

Lorsqu'il reste plusieurs fois le même nom, c'est souvent symptomatique d'**une modélisation non aboutie ou le signe d'une redondance**

Deux entités homogènes peuvent être fusionnées :



Redondance, donc risque d'incohérence :



→ si deux attributs contiennent les mêmes informations, alors la redondance induit un gaspillage d'espace et un grand risque d'incohérence (si les adresses ne sont pas les mêmes, où livrer ?)

Normalisation des identifiants : chaque entité doit posséder un identifiant

Conseils :

Eviter les identifiants composés de plusieurs attributs (comme un identifiant formé par les attributs nom et prénom), car ceux-ci aggravent les performances et une telle unicité finit tôt ou tard par être démentie

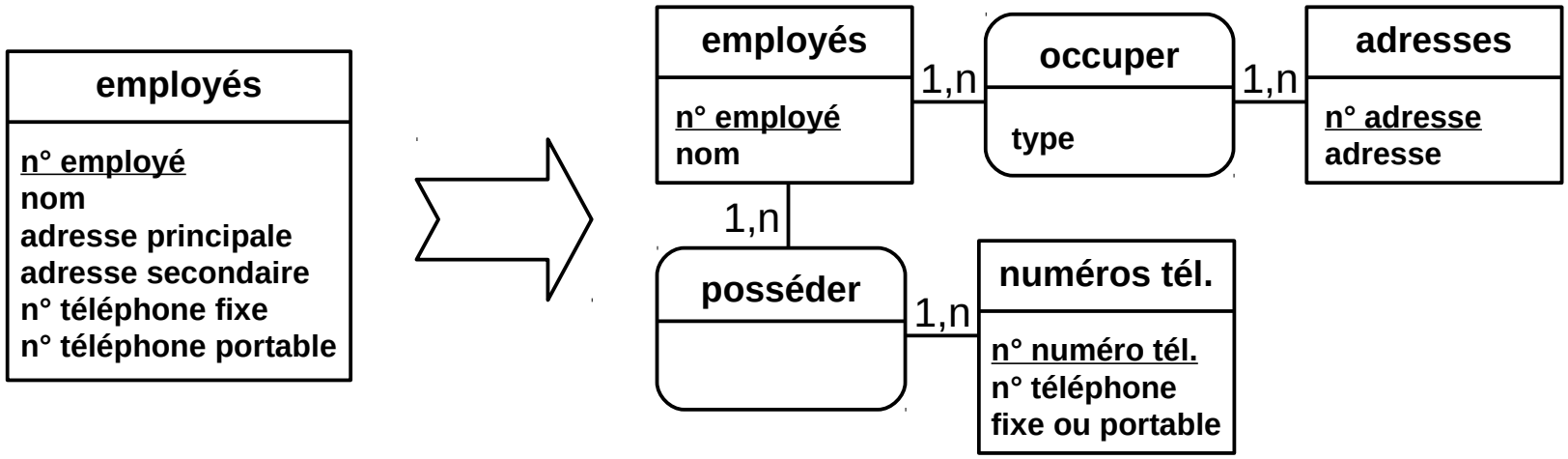
Préférer un identifiant court pour rendre la recherche la plus rapide possible (éviter les chaînes de caractères comme un numéro de plaque d'immatriculation, un numéro de sécurité sociale ou un code postal)

Eviter également les identifiants susceptibles de changer (comme les plaques d'immatriculation ou les numéros de sécurité sociale provisoires)

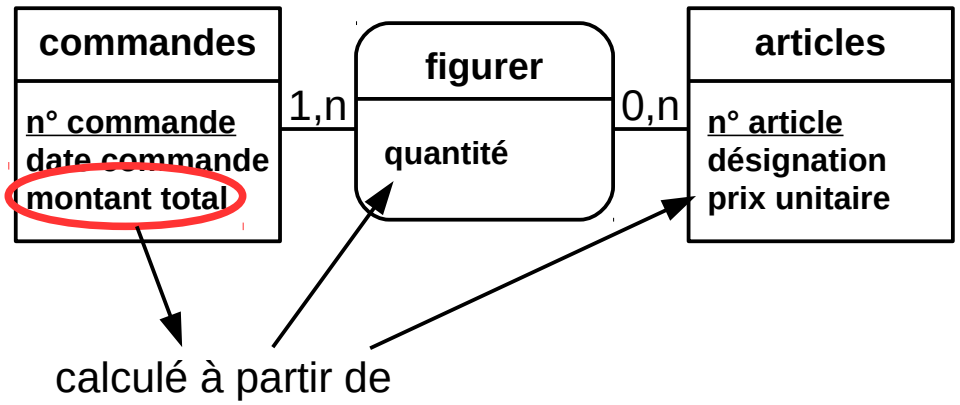
Conclusion : l'identifiant sur un schéma entités-associations (et donc la future clé primaire dans le schéma relationnel MLD / MPD) doit être **un entier, de préférence incrémenté automatiquement**

Normalisation des attributs : remplacer les attributs en plusieurs exemplaires par une association supplémentaire de cardinalités maximales n, et ne pas ajouter d'attributs calculables à partir d'autres attributs

Les attributs en plusieurs exemplaires posent des problèmes d'évolutivité du modèle (comment faire si un employé a plusieurs adresses secondaires ou plusieurs numéros de téléphone ?) :



Les attributs calculables induisent un risque d'incohérence entre les valeurs des attributs de base et celles des attributs calculés :

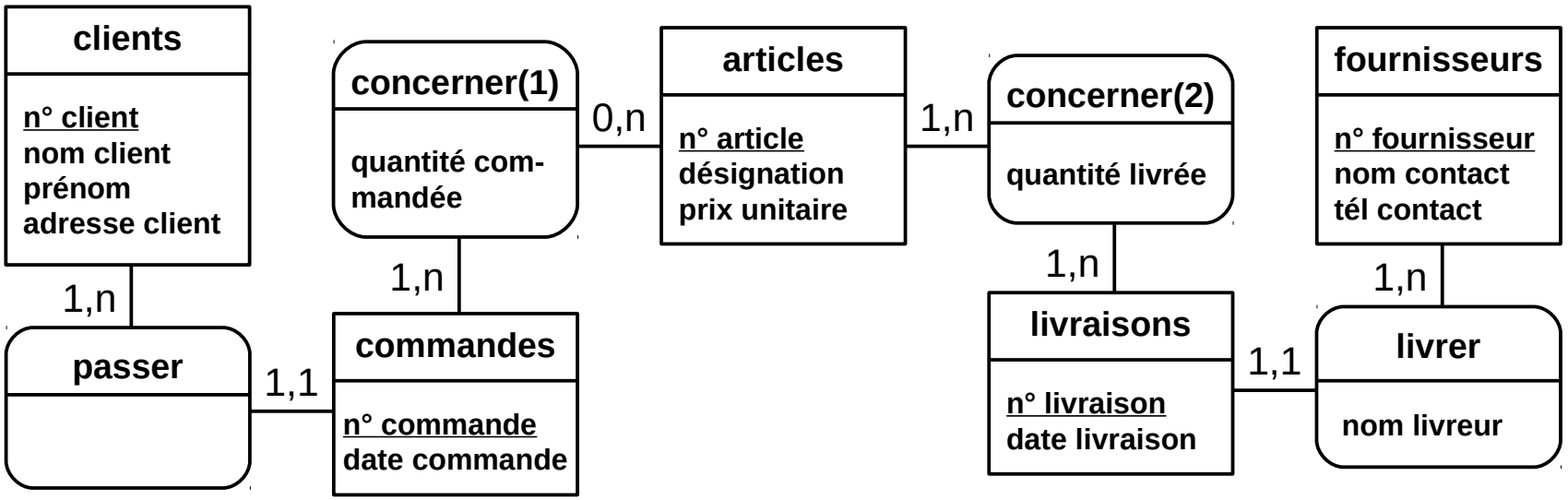


Donc **à retirer du schéma** (présent et calculé côté applicatif)

→ *d'autres attributs calculables classiques sont à éviter : l'âge (calculable à partir de la date de naissance) ou encore le département (calculable à partir d'une extraction du code postal)*

Normalisation des attributs des associations : les attributs d'une association doivent dépendre directement des identifiants de toutes les entités en association

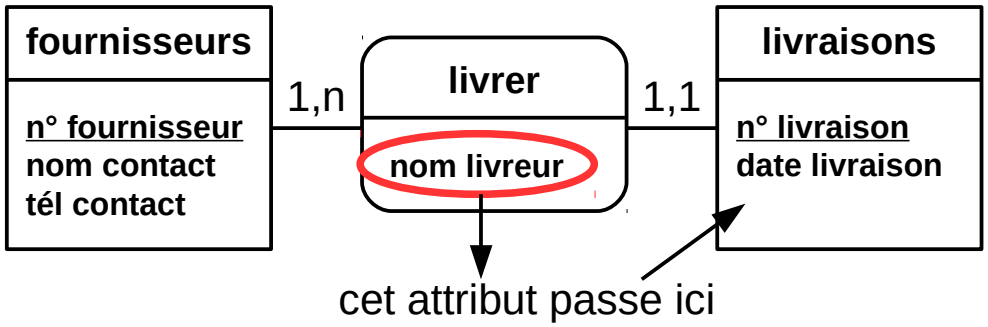
Dans notre exemple de départ, la quantité commandée dépend bien à la fois du numéro de client et du numéro d'article, mais pas la date de commande. Il faut donc **faire une entité « commandes » à part, idem pour les livraisons :**



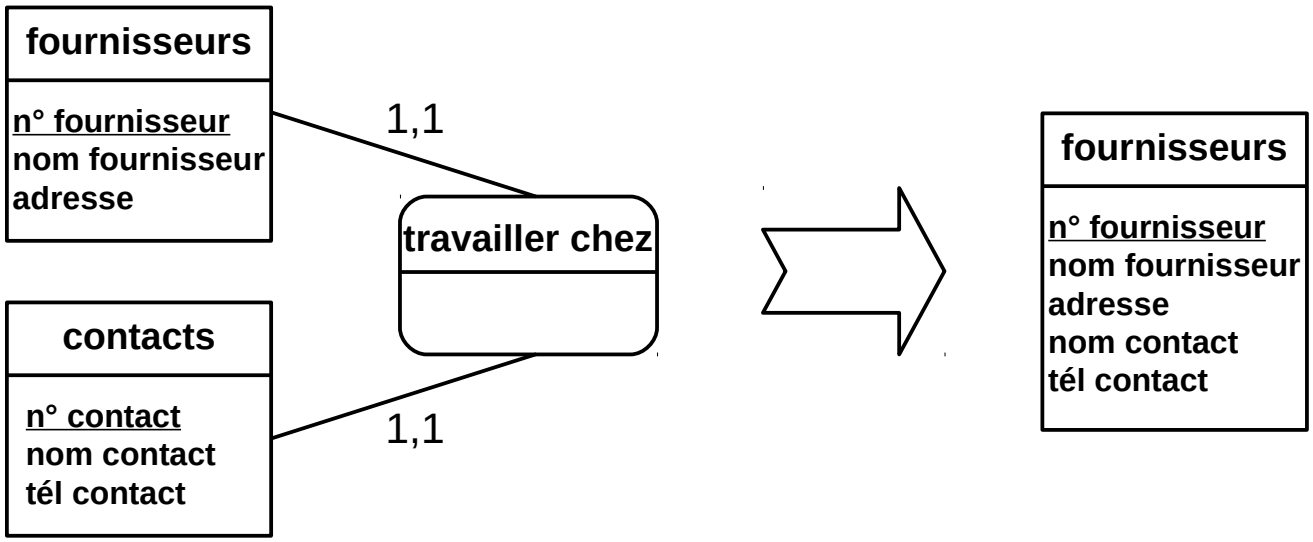
L'inconvénient de cette règle est qu'elle est **difficile à appliquer pour les associations ne possédant pas d'attribut**. Pour vérifier qu'une association sans attribut est bien normalisée, on peut donner à cette association un attribut "imaginaire" permettant de vérifier la règle

Exemple : entre une entité « livres » et une entité « auteurs », une association « écrire » ne possède pas d'attribut. Imaginons que nous ajoutions un attribut « pourcentage » contenant le pourcentage du livre écrit par chaque auteur. Cet attribut dépendant à la fois du numéro de livre et du numéro d'auteur, l'association « écrire » est bien normalisée

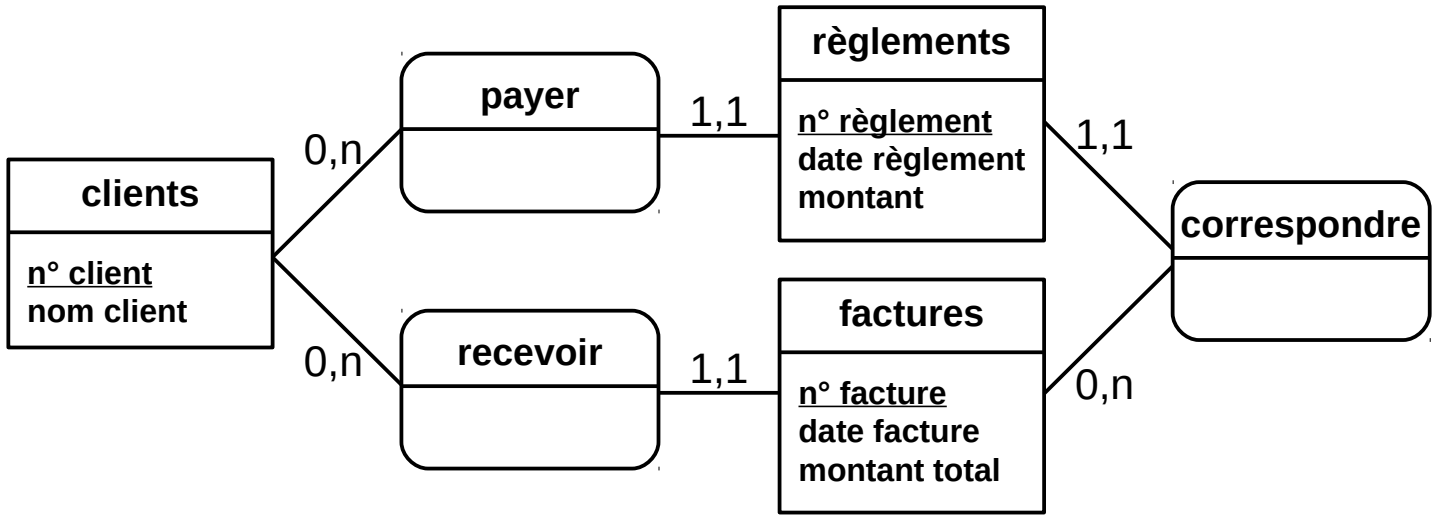
Autre conséquence de la normalisation des attributs des associations : une entité avec une cardinalité de 1,1 ou 0,1 aspire les attributs de l'association



Normalisation des associations : il faut éliminer les associations fantômes, redondantes ou en plusieurs exemplaires



→ *les deux cardinalités sont à 1,1 c'est donc une association fantôme*



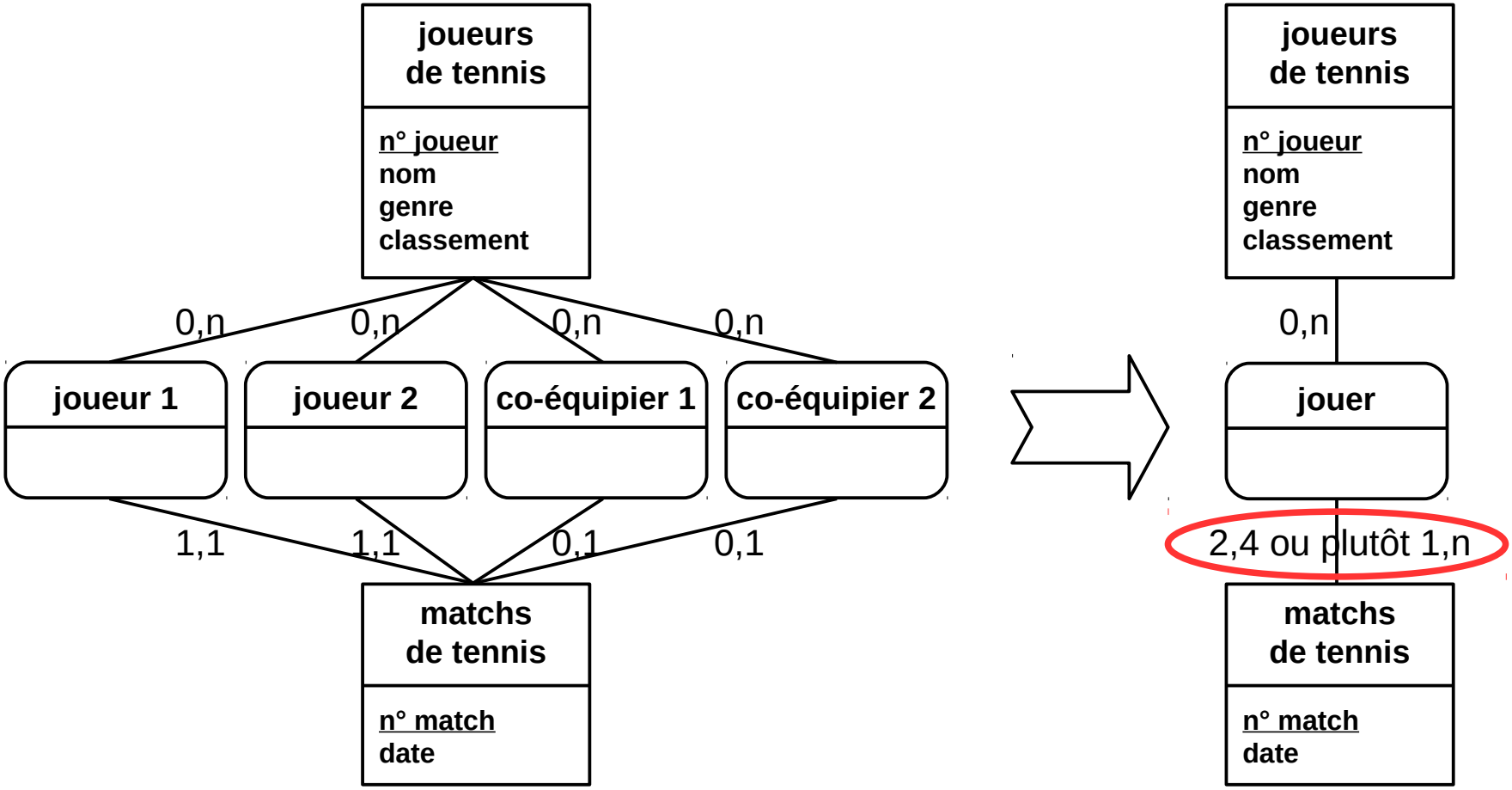
→ *si un client ne peut pas régler la facture d'un autre client, alors l'association « payer » est inutile et doit être supprimée (dans le cas contraire, l'association « payer » doit être maintenue)*

Complément :

En ce qui concerne les associations redondantes, s'il existe deux chemins pour se rendre d'une entité à une autre, ceux-ci doivent avoir deux significations ou deux durées de vie différentes. Sinon, **il faut retirer le chemin le plus court** car déductible à partir de l'autre chemin

Dans notre exemple, si l'on supprime l'association « payer », on peut retrouver le client qui a payé le règlement en passant par la facture qui correspond

Remarque : une autre solution consiste à retirer l'entité « règlements » et d'ajouter une association « régler » avec les mêmes attributs (sauf l'identifiant) entre les entités « clients » et « factures »



→ *une seule association suffit pour remplacer les 4 associations « joueur / co-équipier » (à gauche) en tant que « jouer » (à droite)*

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,...)

Si une cardinalité maximale est connue et vaut 2, 3 ou plus (comme sur le schéma précédent, ou pour un nombre limité d'emprunts dans une bibliothèque), **il faut quand-même considérer qu'elle est indéterminée et vaut n** : même si nous connaissons n au moment de la conception, il se peut que cette valeur évolue au cours du temps (donc la considérer comme inconnue)

On ne modélise pas non plus les cardinalités minimales qui valent plus de 1 car ce genre de valeurs est aussi amené à évoluer

Par ailleurs, **avec une cardinalité maximale de 0, l'association n'aurait aucune signification**

5) Méthodologie de base

Face à une situation bien définie (énoncé précis, collection de formulaires ou d'états du système d'information souhaité,...), **procéder ainsi :**

- définir le **dictionnaire des données** → *le plus exhaustif possible*
- identifier les **entités** en présence
- lister leurs **attributs** (d'après le dictionnaire des données)
- ajouter les **identifiants** (numéros arbitraires et auto-incrémentés)
- établir les **associations** binaires (ou non binaires) entre les entités
- lister leurs **attributs** (d'après le dictionnaire des données)
- calculer les **cardinalités**
- vérifier les **règles de normalisation**
- effectuer les **corrections** nécessaires

III. Modèle Logique de Données (MLD)

III. Modèle Logique de Données (MLD)

1) Généralités
2) Tables, colonnes et lignes
3) Clés primaires et clés étrangères
4) Schéma relationnel
5) Traduction d'un MCD en un MLDR

1) Généralités

Suite à l'étape de modélisation du MCD, la méthode Merise fournit un **procédé permettant d'aboutir à la structure finale de la base de données : le MLD** qui décrit comment sont organisées les données

→ *ou MLDR pour « Relationnel » et bases de données relationnelles*

Le passage au MLD détermine la liste des colonnes de chacune des tables du système d'information décrit par le MCD

C'est une étape intermédiaire semi technique : les éléments du MCD sont devenus des tables et des colonnes, mais aucun choix technique (SGBD ou SGBDR) n'est encore fait

2) Tables, colonnes et lignes

Lorsque des données ont la même structure (comme les renseignements relatifs à des clients), **on peut les organiser en table** dans laquelle les **colonnes** décrivent les champs en commun et les **lignes** contiennent les valeurs de ces champs pour chaque enregistrement :

n° client	nom	prénom	adresse
1	Dupont	Michel	127, rue...
2	Durand	Jean	314, boulevard...
3	Dubois	Claire	51, avenue...
4	Dupuis	Marie	2, impasse...
...	...	...	...

→ *contenu de la table « clients », avec en première ligne les intitulés des colonnes*

3) Clés primaires et clés étrangères

Clé primaire :

Les lignes d'une table doivent être uniques, une colonne (au moins) doit donc servir à les identifier. Il s'agit de la **clé primaire de la table**

L'absence de valeur dans une clé primaire ne doit pas être autorisée :

Autrement dit, la valeur vide (NULL) est interdite dans une colonne qui sert de clé primaire. Ce qui n'est pas forcément le cas des autres colonnes, dont certaines peuvent ne pas être renseignées à toutes les lignes

De plus, **la valeur de la clé primaire** d'une ligne **ne doit pas changer** au cours du temps (en principe)

Clé étrangère :

Il se peut qu'**une colonne « Colonne1 » d'une table ne doive contenir que des valeurs prises par la colonne « Colonne2 » d'une autre table** (exemple : le numéro du client sur une commande doit correspondre à un vrai numéro de client)

'2' doit être sans doublons (souvent il s'agit d'une clé primaire). On dit alors que **'1' est clé étrangère** et qu'elle **référence '2'**

Par convention, on **souligne les clés primaires** et on fait **précéder les clés étrangères d'un dièse '#'** dans la description des colonnes d'une table :

clients(n° client, nom client, prénom, adresse client)

commandes(n° commande, date commande, #n° client (non vide))

Précisions :

Une même table peut **avoir plusieurs clés étrangères mais une seule clé primaire** (éventuellement composée de plusieurs colonnes)

Une colonne **clé étrangère peut aussi être primaire** (d'une même table)

Une clé étrangère peut être composée (c'est le cas si la clé primaire référencée est composée)

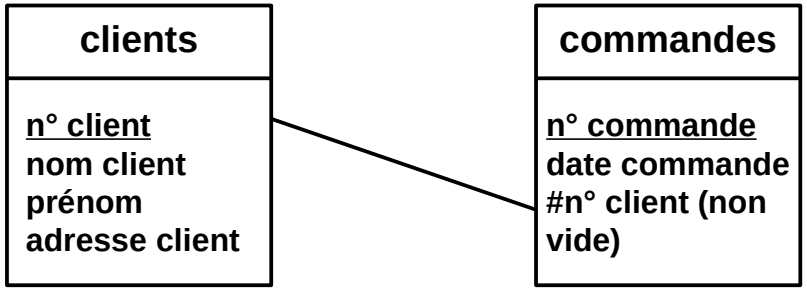
Implicitement, chaque colonne qui compose **une clé primaire ne peut pas recevoir la valeur vide** (NULL interdit)

Si une colonne clé étrangère ne doit pas recevoir la valeur vide (souvent le cas), il faut le préciser dans la description des colonnes

→ *les SGBDR vérifient au coup par coup que chaque clé étrangère ne prend pas de valeurs différentes de celles déjà prises par la ou les colonne(s) qu'elle référence. Ce mécanisme, qui agit à chaque insertion, modification ou suppression de lignes dans les tables, garantit ce que l'on appelle « l'intégrité référentielle des données »*

4) Schéma relationnel

On peut 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 symbolisés par un connecteur** (en général un trait) :



→ *le connecteur peut être également représenté par une flèche (pointe côté clé primaire), ou encore avec un '1' côté clé primaire et un '?' côté clé étrangère*

5) Traduction d'un MCD en un MLDR

Pour traduire un MCD en un MLDR, **il suffit d'appliquer 5 règles**

Notations : on dit qu'une association binaire (entre deux entités ou réflexive) est de type :

1 : 1 (un à un) si aucune des deux cardinalités maximales n'est n

1 : n (un à plusieurs) si une des deux cardinalités maximales est n

n : m (plusieurs à plusieurs) si les deux cardinalités maximales sont n

En fait, **un schéma relationnel ne peut faire la différence entre 0,n et 1,n**. Par contre, il peut la faire entre 0,1 et 1,1 (règles 2 et 4)

Règle 1 :

Toute entité devient une table dans laquelle **les attributs deviennent les colonnes**

L'identifiant de l'entité constitue alors la clé primaire de la table

Dans notre exemple de départ, l'entité « articles » devient donc la table suivante :

articles(n° article, désignation, prix unitaire)

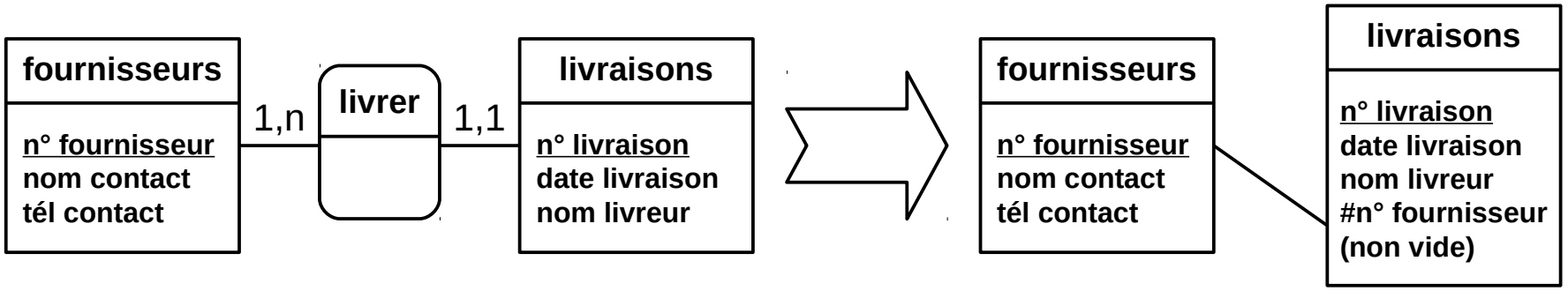
Règle 2 :

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

Dans notre exemple de départ, l'association « livrer » est donc traduite par :

fournisseurs(n° fournisseur, nom contact, tél contact)

livraisons(n° livraison, date livraison, nom livreur, #n° fournisseur (non vide))



Complément :

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

Règle 3 :

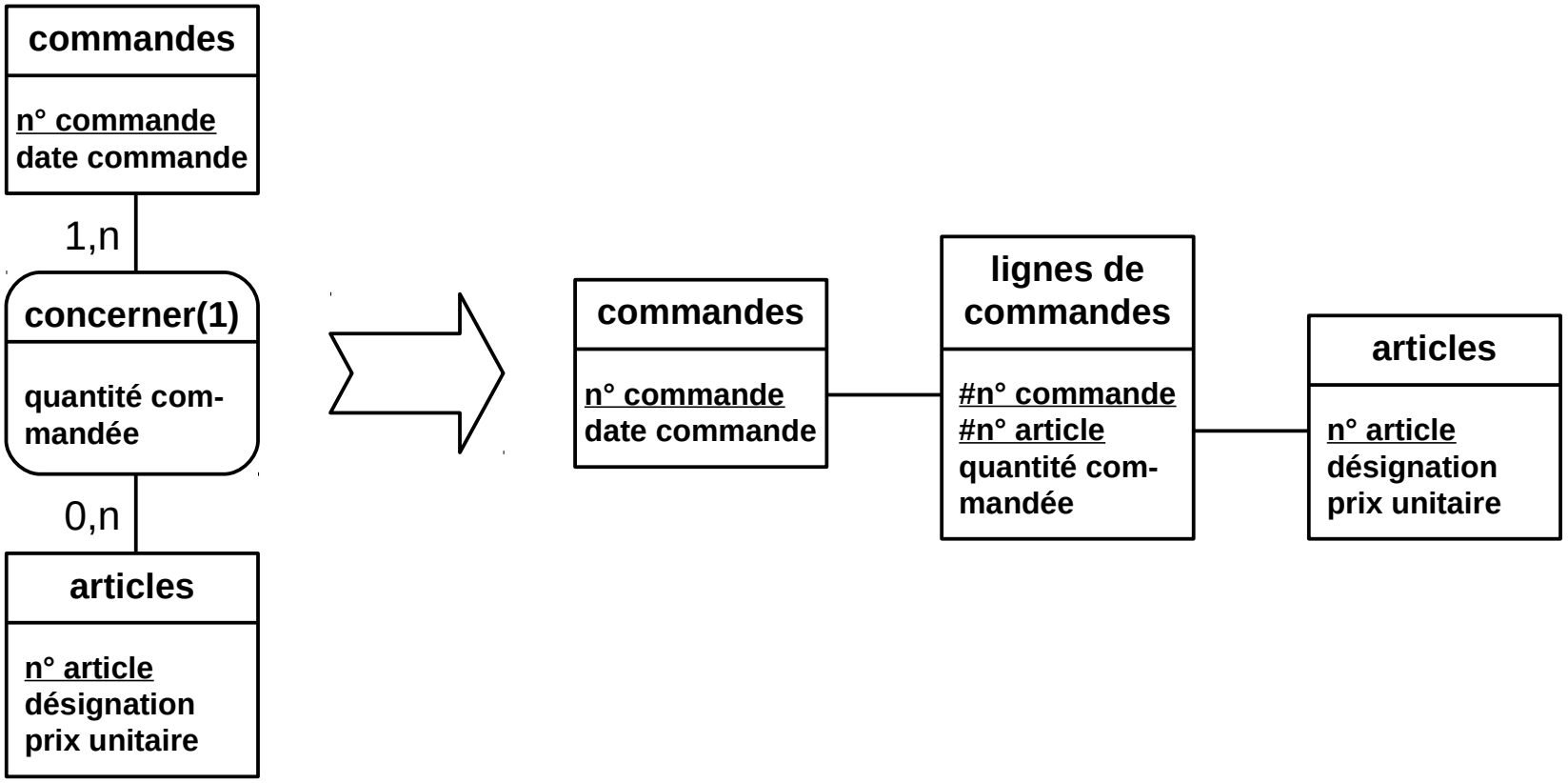
Une association binaire de type « n : m » devient une table supplémentaire (parfois appelée table de jonction, table de jointure ou table d'association), **dont la clé primaire est composée de deux clés étrangères** (qui référencent les deux clés primaires des deux tables en association)

Les attributs de l'association deviennent des colonnes de cette nouvelle table

Dans notre exemple de départ, l'association « concerner(1) » est donc traduite par la table supplémentaire « lignes de commandes » suivante :

lignes de commandes(#n° commande, #n° article, quantité commandée)

Traduction de cette association de type « n : m » :



Règle 4 :

Une association binaire de type « 1 : 1 » est traduite comme une association binaire de type « 1 : n », sauf que la clé étrangère se voit imposer une contrainte d'unicité en plus d'une éventuelle contrainte de non vacuité

→ *cette contrainte d'unicité impose à la colonne correspondante de ne prendre que des valeurs distinctes*

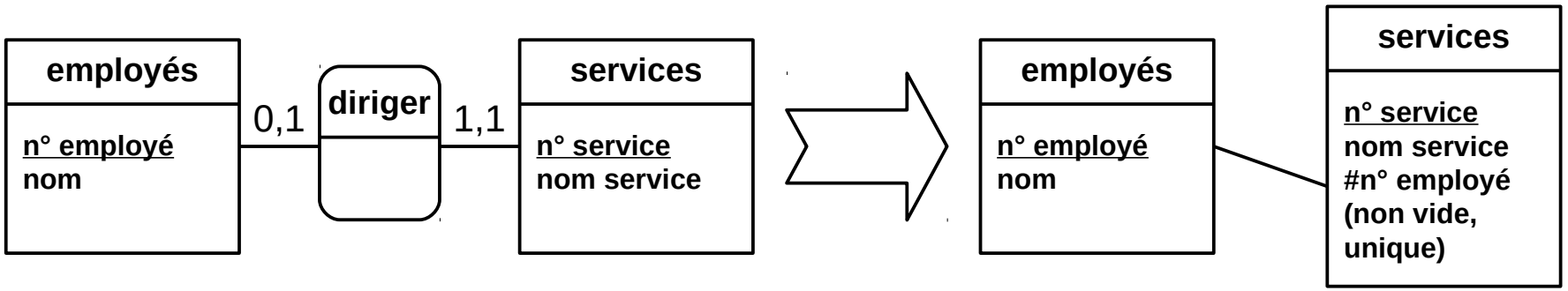
Si les associations fantômes ont été éliminées, il devrait y avoir au moins un côté de cardinalité 0,1. C'est alors dans la table du côté opposé que doit aller la clé étrangère

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 » ci-dessous est donc traduite par :

employés(n° employé, nom)

services(n° service, nom service, #n° employé (non vide, unique))



Règle 5 :

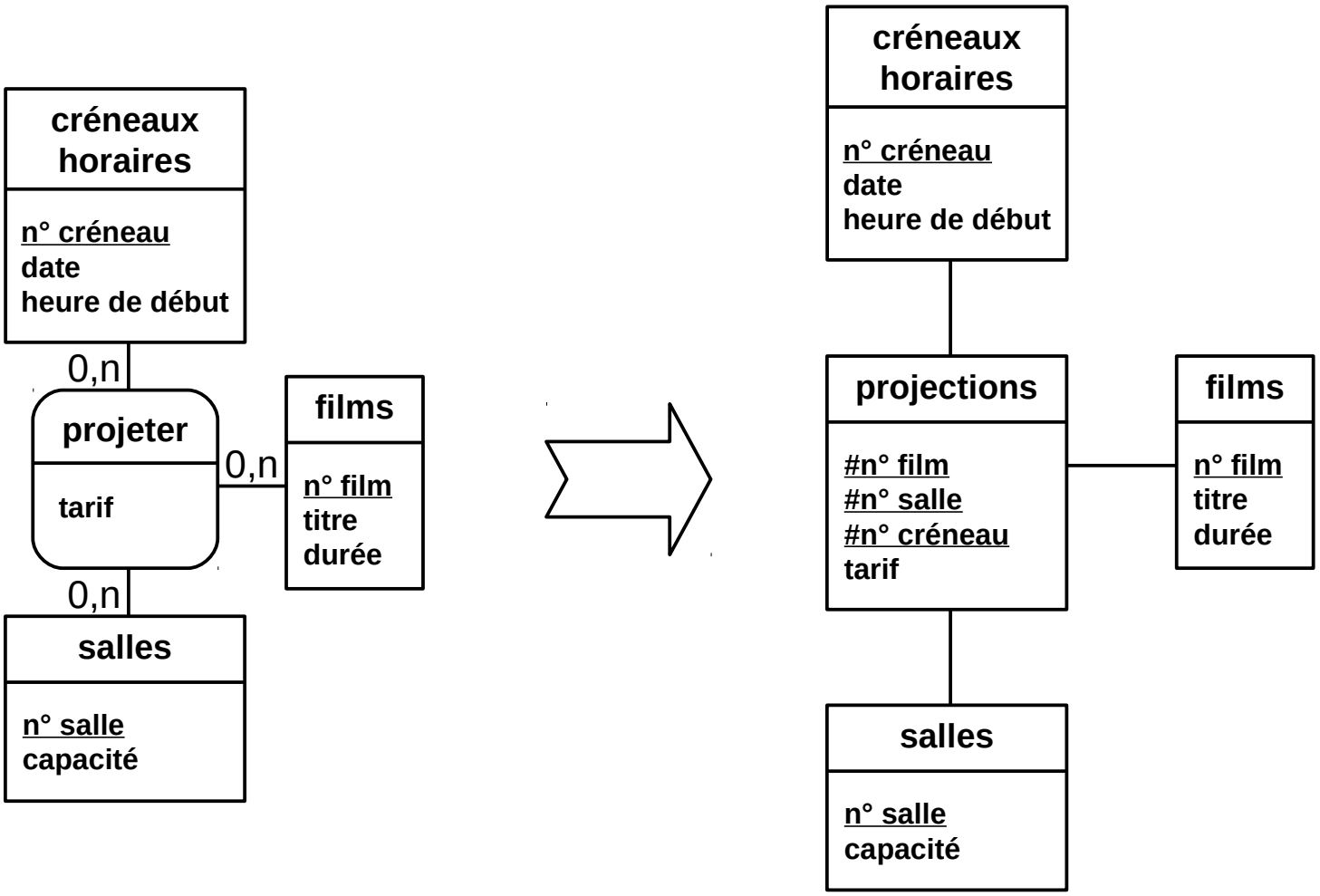
Une association non binaire est traduite par une table supplémentaire, dont la clé primaire est composée d'autant de clés étrangères que d'entités en association

Les attributs de l'association deviennent des colonnes de cette nouvelle table

Par exemple, l'association « projeter » ci-après devient donc la table suivante :

projections(#n° film, #n° salle, #n° créneau, tarif)

Traduction de cette association ternaire :



IV. Modèle Physique de Données (MPD)

IV. Modèle Physique de Données (MPD)

1) Généralités
2) Optimisations

1) Généralités

La dernière étape de la méthode Merise est la **conversion du MLD en MPD**. C'est l'implémentation particulière du MLD **par un certain SGBD**

Il s'agit principalement de **décrire le type (chaîne de caractères, numérique,...) de chacune des colonnes** des tables

Ces types de données peuvent varier selon les différents systèmes de gestion de bases de données relationnelles (SGBDR) → *choix technique*

→ *le code SQL créant la base de données est ensuite une déduction directe de ce MPD (en fonction du SGBDR choisi et de sa version)*

2) Optimisations

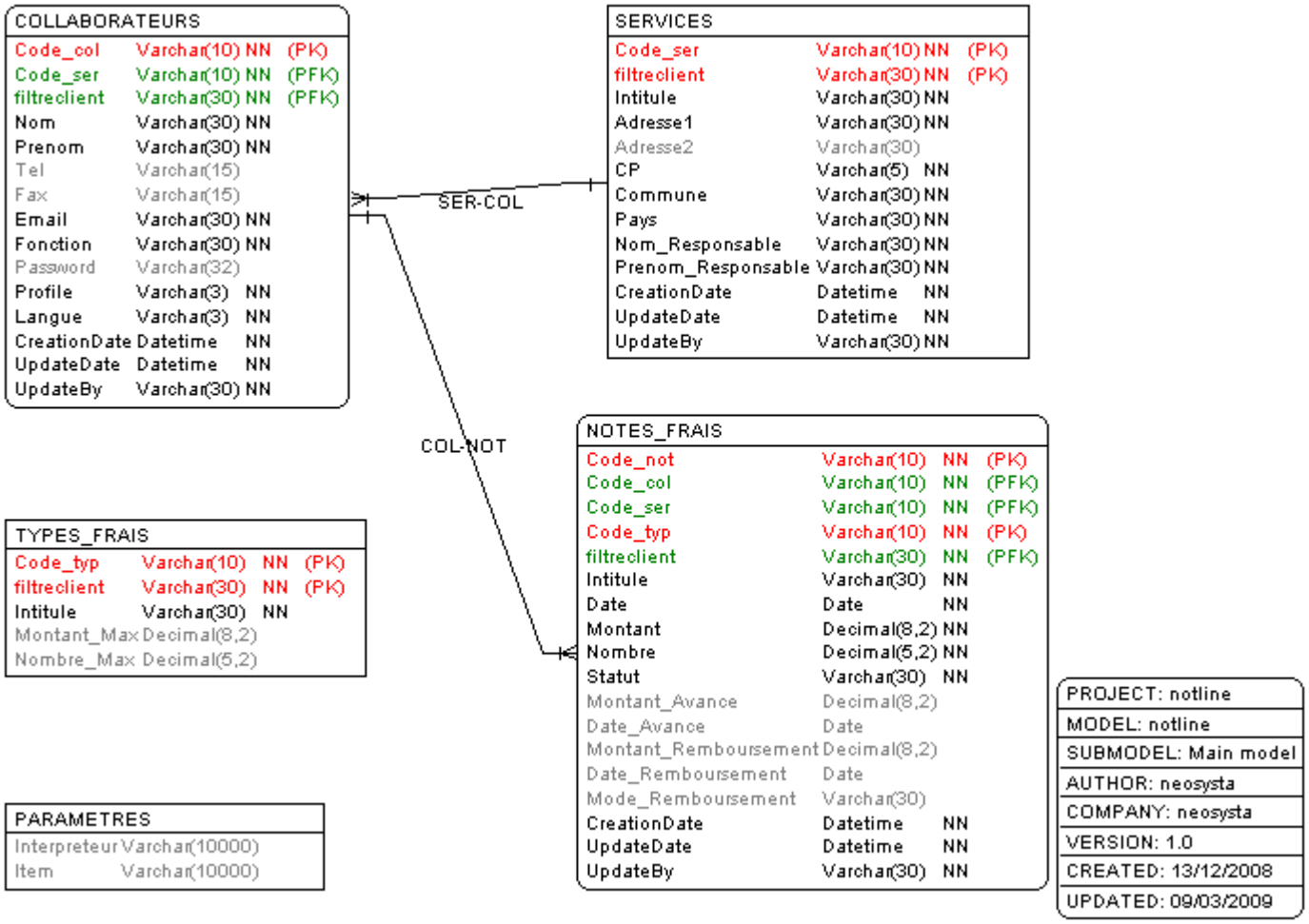
L'optimisation des performances en temps de calcul se fait toujours au détriment de l'espace mémoire consommé. Dans le pire des cas, réduire les temps de réponse consiste à dé-normaliser volontairement le système d'information (avec risques d'incohérence et problèmes de gestion). Cette optimisation peut passer par :

L'ajout d'index aux tables (au minimum sur les clés primaires et étrangères) : consomment de la mémoire mais la base reste normalisée

L'ajout de colonnes calculées ou de certaines redondances pour éviter des jointures coûteuses (mais base dé-normalisée) : veiller à la cohérence des colonnes avec des déclencheurs (triggers) ou côté application cliente

La suppression des contraintes d'unicité, de non vacuité ou encore de clé étrangère : intégrité des données à assurer par le code client

Complément : MPD de la base de données étudiée (« notline »)



V. Serveur PostgreSQL

V. Serveur PostgreSQL

1) Présentation
2) Schémas
3) Outils clients

1) Présentation

Généralités :

Système de gestion de bases de données relationnelles et objets (SGBDRO) basé sur le modèle Client-Serveur

Première version : 1995 (issu d'Ingres redéveloppé de zéro en 1985)

Développeur : Michael Stonebraker (USA)

Systèmes d'Exploitation : Unix / Linux, Mac OS X, BSD, Windows,...

Licence : outil libre selon les termes d'une licence de type BSD

Complément :

http://www.postgresql.org

http://www.postgresql.fr

(http://www.postgresqlfr.org)

Nombreuses fonctionnalités modernes :

Standard SQL avec **requêtes complexes**

Clés étrangères

Triggers

Vues modifiables

Intégrité transactionnelle avec **contrôle des versions concurrentes** (MVCC, « MultiVersion Concurrency Control »)

Possibilité de nouveaux **types de données**, d'**opérateurs** et de **fonctions d'agrégat**

Ajout de nouvelles **méthodes d'indexation**

Création de **fonctions / procédures stockées** (via PL/pgSQL)

Intégration d'autres **langages procéduraux** (PL/Tcl, PL/Perl, PL/Python)

Interfaces clientes pour de nombreux langages (JDBC, ODBC, .NET, PHP, C/C++, Tcl, Perl, Python,...)

Complément :

Tout en étant Open Source, PostgreSQL est largement reconnu pour son comportement stable **proche d'Oracle (fonctionnalités comprises)**

Egalement très utilisé par les sites Web et **proposé par la majorité des hébergeurs**

Downloads (portail général officiel, par plateforme) :

http://www.postgresql.org/download/

Documentations (par version) :

http://www.postgresql.org/docs/

http://docs.postgresql.fr

Portails développeurs (diverses ressources) :

http://www.postgresql.org/developer/

http://www.postgresql.fr/devel:accueil

Entraides communautaires et supports commerciaux :

http://www.postgresql.org/support/

http://wiki.postgresql.fr/support:start

2) Schémas

Avec PostgreSQL, toute connexion cliente au serveur ne peut accéder qu'aux données d'une seule base (celle indiquée dans la requête de connexion)

Une base de données contient un ou plusieurs schéma(s) nommé(s) qui, eux, contiennent des tables (et aussi d'autres objets nommés : types de données, fonctions,...)

Dans la même base de données, **le même nom d'objet peut être utilisé dans différents schémas sans conflit** (« schema1 » et « schema2 » peuvent tous les deux contenir une table nommée « ma_table »)

Un utilisateur peut accéder aux objets de n'importe quel schéma de la base de données à laquelle il est connecté, sous réserve qu'il en ait le droit

→ *utiles à l'organisation (fonctionnelle) de la base de données et à la gestion de ses droits (par schéma, objet et utilisateur)*

Complément :

Notation pointée : → *orientée objet*

schema.table

base.schema.table → *uniquement la base de connexion (à ce jour)*

Schéma « public » : les tables (et autres objets) créées sans qu'un nom de schéma ne soit indiqué sont **placées par défaut dans un schéma nommé « public » (contenu dans toute nouvelle base créée)**

Dans ce cas : « CREATE TABLE ma_table (...); » équivaut à

« CREATE TABLE public.ma_table (...); »

→ *de même à l'utilisation des objets (schéma « public » par défaut)*

3) Outils clients

Accès aux principaux outils et commandes PostgreSQL :

Sur Windows :

Normalement accessibles depuis le chemin **"C:\Program Files\ PostgreSQL\x.x\bin"**, à ajouter – si nécessaire – à la variable d'environnement « PATH » pour éviter de devoir le préciser à chaque utilisation (ou via la commande « cd »)

Sur Linux :

Normalement accessibles depuis le chemin **"/usr/bin"**, déjà ajouté à la variable d'environnement « PATH » pour éviter de devoir le préciser à chaque utilisation (commande « cd » inutile)

Se connecter au serveur PostgreSQL (local) avec son outil client de saisie de commandes (en utilisant l'administrateur « postgres ») :

Sur Linux :

su postgres

psql (ou « psql -U postgres »)

Sur Windows : psql -U postgres

Puis afficher l'aide de « psql » : help

Afficher les informations de connexion et utilisateur : \conninfo

Afficher les bases de données existantes : \l

Quitter « psql » : \q

L'outil client ligne de commandes « psql » :

Outil client par défaut de saisie de commandes (dédiées et SQL) sur serveur PostgreSQL

Pour l'exécuter dans une Console Windows ou un Terminal Shell Unix / Linux :

« psql » éventuellement suivi d'options '-' et de paramètres

Pour avoir une aide sur sa syntaxe en ligne de commande :

Windows / Unix / Linux : **psql --help | more**

Unix / Linux (plus complète) : **man psql**

Quelques options / paramètres utiles de « psql » :

-U, --username=nom : permet de préciser l'utilisateur utilisé

-W, --password : demande de saisir le mot de passe de l'utilisateur

→ *mode par défaut*

-h, --host=nomserveur : permet de se connecter à un serveur PostgreSQL distant (hôte) nommé « nomserveur » (ou par son adresse IP)

-p, --port=numéro : permet de choisir le port du serveur PostgreSQL (5432 par défaut)

Complément :

« psql », sans précision d'utilisateur de connexion « -U », utilise par défaut l'utilisateur de la session système (si autorisé)

-d, --dbname=nombase : indique quelle base de données utiliser (par défaut celle du nom de l'utilisateur)

-c, --command="commande" : exécute la commande (une seule, dédiée ou SQL) et quitte immédiatement « psql »

-f, --file=fichier : exécute les commandes (dédiées et SQL) du fichier et quitte immédiatement « psql »

Exemples :

psql -U postgres -h localhost -p 5432 -d postgres

psql -U postgres -d postgres -c "\l"

psql -U postgres -d postgres -c "select 'hello' as hello;"

psql -U postgres -d postgres -f test.sql

Complément :

Exemple pour le fichier « test.sql » :

```
\conninfo
\l
\du
select 123;
select 'hello' as hello;
```

Autre syntaxe : psql -U postgres -d postgres < test.sql

Commande « help » de « psql » :

You are using psql, the command-line interface to PostgreSQL.

Type: \copyright for distribution terms

\h for help with SQL commands

\? for help with psql commands

\g or terminate with semicolon to execute query → « ; »

\q to quit

Complément :

Voir aussi **« \h » + mot clé SQL**

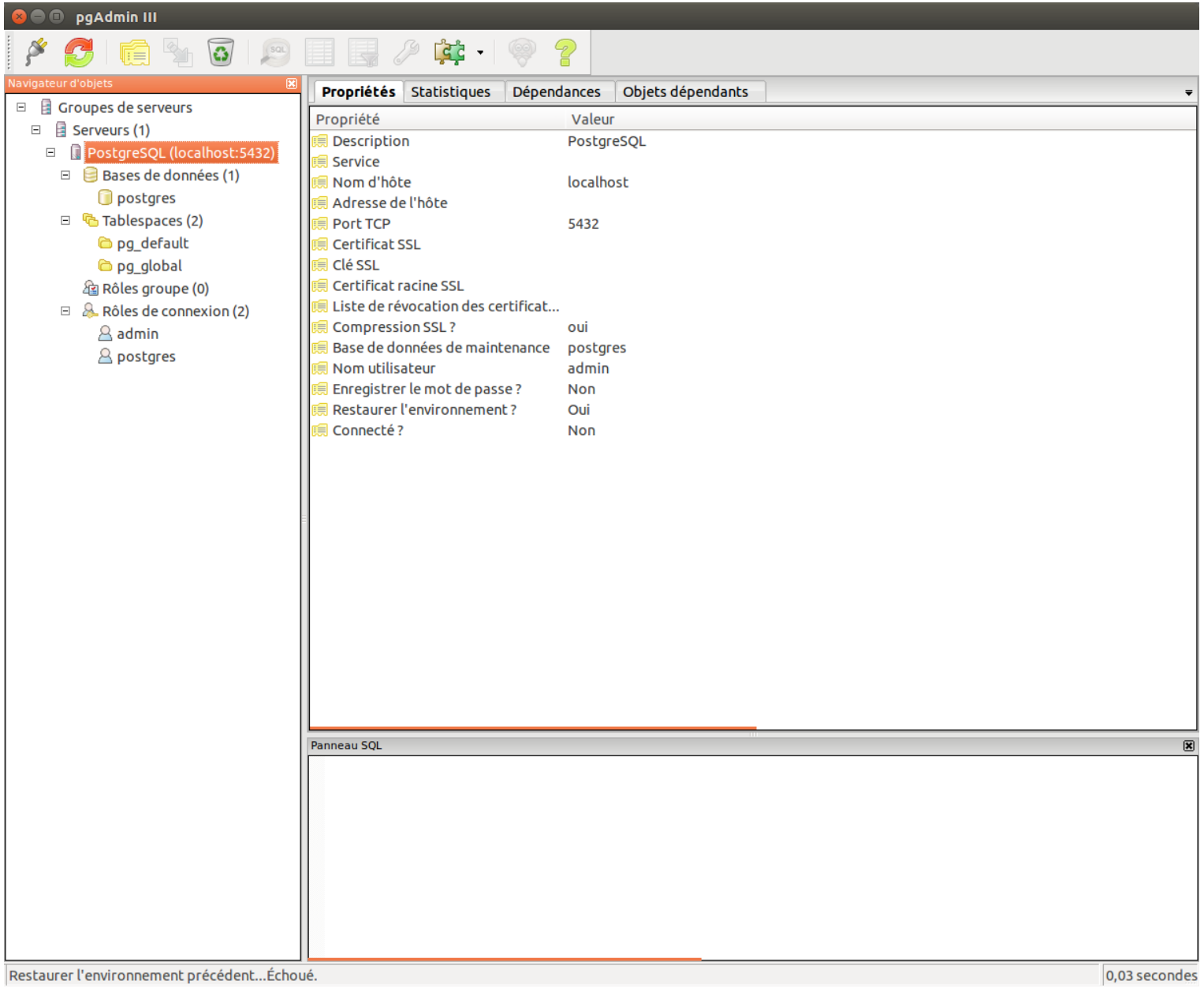
Avec PostgreSQL, **une instruction SQL doit toujours se terminer par « ; »** pour être exécutée (ou « \g »)

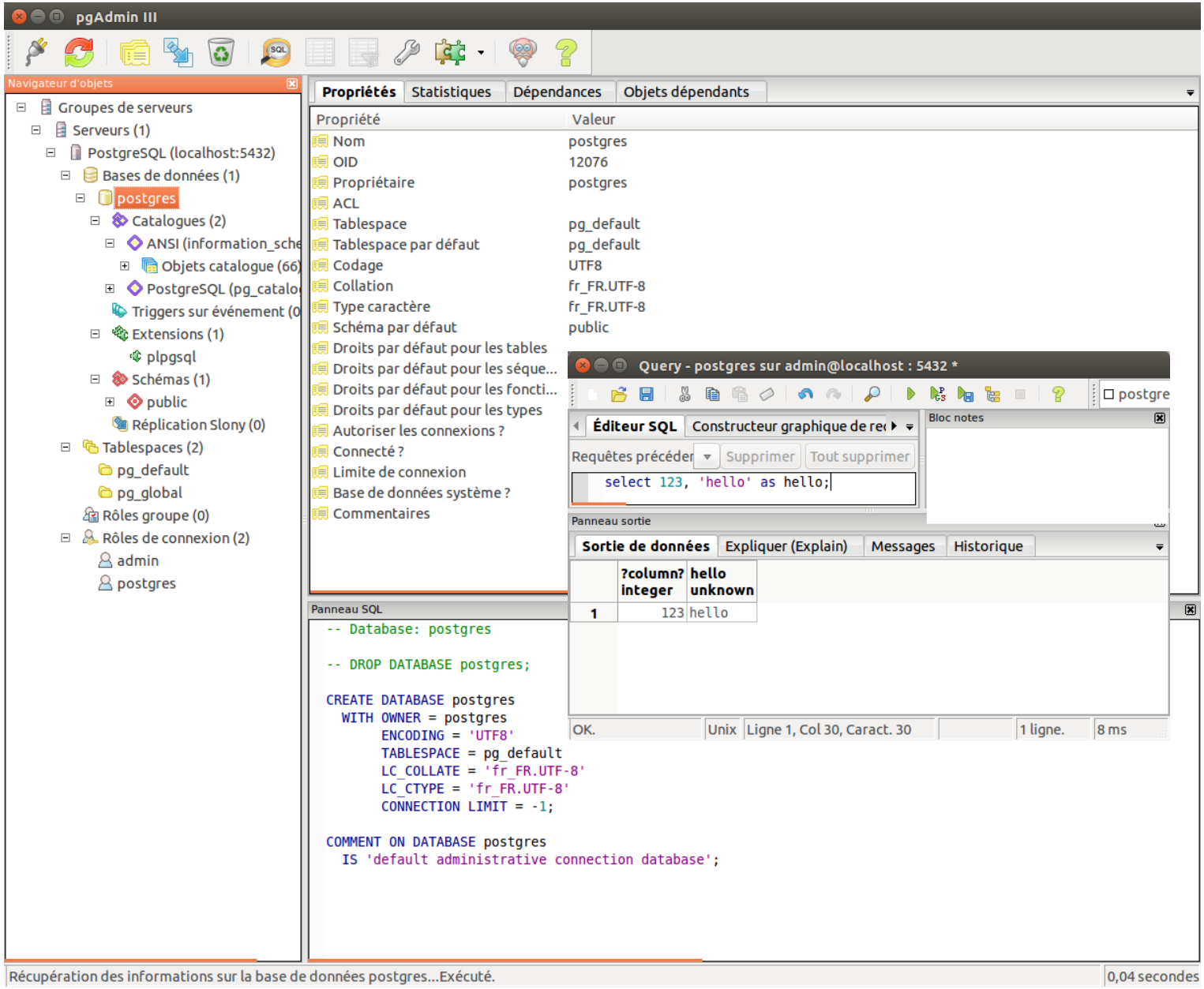
L'outil graphique « pgAdmin » :

Outil graphique d'utilisation et d'administration PostgreSQL

http://www.pgadmin.org → *Open Source et tout OS*

Installation sur Windows : fourni avec PostgreSQL
Installation sur Linux CentOS : non fourni dans les paquets de base
→ *installation spécifique*
Installation sur Linux Ubuntu :
apt-cache search pgadmin
sudo apt-get install pgadmin3


pgAdmin III
Navigateur d'objets
Groupes de serveurs
Serveurs (1)
PostgreSQL (localhost:5432)
Bases de données (1)
postgres
Tablespaces (2)
pg_default
pg_global
Rôles groupe (0)
Rôles de connexion (2)
admin
postgres
Propriétés
Statistiques
Dépendances
Objets dépendants
Propriété
Valeur
Description
PostgreSQL
Service
Nom d'hôte
localhost
Adresse de l'hôte
Port TCP
5432
Certificat SSL
Clé SSL
Certificat racine SSL
Liste de révocation des certificat...
Compression SSL ?
oui
Base de données de maintenance
postgres
Nom utilisateur
admin
Enregistrer le mot de passe ?
Non
Restaurer l'environnement ?
Oui
Connecté ?
Non
Panneau SQL
Restaurer l'environnement précédent...Échoué.
0,03 secondes


pgAdmin III
Navigateur d'objets
Groupes de serveurs
Serveurs (1)
PostgreSQL (localhost:5432)
Bases de données (1)
postgres
Catalogues (2)
Tablespaces (2)
pg_default
pg_global
Rôles groupe (0)
Rôles de connexion (2)
admin
postgres
Propriétés
Statistiques
Dépendances
Objets dépendants
Propriété
Valeur
Nom
postgres
OID
12076
Propriétaire
postgres
ACL
Tablespace
pg_default
Tablespace par défaut
pg_default
Codage
UTF8
Collation
fr_FR.UTF-8
Type caractère
fr_FR.UTF-8
Schéma par défaut
public
Query - postgres sur admin@localhost : 5432 *
Éditeur SQL
Bloc notes
select 123, 'hello' as hello;
Panneau sortie
Sortie de données
Expliquer (Explain)
Messages
Historique
?column? integer
hello unknown
1
123
hello
Panneau SQL
-- Database: postgres
-- DROP DATABASE postgres;
CREATE DATABASE postgres
WITH OWNER = postgres
ENCODING = 'UTF8'
TABLESPACE = pg_default
LC_COLLATE = 'fr_FR.UTF-8'
LC_CTYPE = 'fr_FR.UTF-8'
CONNECTION LIMIT = -1;
COMMENT ON DATABASE postgres
IS 'default administrative connection database';
OK.
Unix
Ligne 1, Col 30, Caract. 30
1 ligne.
8 ms
Récupération des informations sur la base de données postgres...Exécuté.
0,04 secondes

Complément :

Autre outil (application Web) : **phpPgAdmin**

VI. SQL avec PostgreSQL

VI. SQL avec PostgreSQL

1) La norme SQL. Positionnement de PostgreSQL. Création d'une base de données

2) Types de données PostgreSQL. Contraintes d'intégrité sur les tables

3) Tables. Séquences

4) Ajout, modification et suppression des données

5) Interrogation du schéma d'une base. Sélection, restriction, tri, jointure

6) Extractions complexes

7) Fonctions intégrées. Vues

8) Transactions

1) La norme SQL. Positionnement de PostgreSQL. Création d'une base de données

La norme SQL et positionnement de PostgreSQL :

SQL a été adopté comme **recommandation par l'institut de normalisation américaine ANSI en 1986, puis comme norme internationale par l'ISO en 1987** sous le nom de « ISO/CEI 9075 - Technologies de l'information - Langages de base de données - SQL »

Depuis, la norme internationale SQL est passée par **un certain nombre de révisions** (la dernière datant de 2016)

PostgreSQL 12 suit la **norme ANSI SQL:2016** (ISO/CEI 9075:2016)

Syntaxe de création d'une base de données (voir « \h CREATE DATABASE ») :

CREATE DATABASE name
[[WITH] [OWNER [=] user_name]
[TEMPLATE [=] template] → *base de données modèle*
[ENCODING [=] encoding]
[LC_COLLATE [=] lc_collate]
[LC_CTYPE [=] lc_ctype]
[TABLESPACE [=] tablespace_name]
[CONNECTION LIMIT [=] connlimit]];

Exemple (modèle proposé) : CREATE DATABASE notline;
Voir les bases de données existantes : \l+

Modifier une base de données (exemple) :
ALTER DATABASE notline RENAME TO autrebase;
ALTER DATABASE autrebase RENAME TO notline;

Se connecter à une autre base de données : \c notline
Voir la base de données utilisée :
« \conninfo » ou SELECT CURRENT_DATABASE();

Supprimer une base de données : DROP DATABASE notline;

Complément :

Les mots clés SQL et les fonctions intégrées ne sont **pas sensibles à la casse**, ni le nom des objets (tables, colonnes,...) **sauf s'il est défini entre guillemets (" ")**

(les commandes internes du client « psql » sont sensibles à la casse)

Schémas :

Rappel : toute connexion cliente au serveur ne peut accéder qu'aux données d'une seule base (celle indiquée dans la requête de connexion)

Une base de données contient un ou plusieurs schéma(s) nommé(s) qui, eux, contiennent des tables (et autres objets)

Schéma « public » : les tables (et autres objets) créées sans qu'un nom de schéma ne soit indiqué sont **placées par défaut dans le schéma nommé « public » (contenu dans toute nouvelle base créée)**
→ *si le schéma par défaut n'a pas été changé (sinon ce dernier)*

Dans ce cas : « CREATE TABLE ma_table (...); » équivaut à
« CREATE TABLE public.ma_table (...); »

2) Types de données PostgreSQL. Contraintes d'intégrité sur les tables

Nom	Alias	Description
bigint	int8	Entier signé sur 8 octets
bigserial	serial8	Entier sur 8 octets à incrémentation automatique
bit [(n)]		Suite de bits de longueur fixe
bit varying [(n)]	varbit	Suite de bits de longueur variable
boolean	bool	Booléen (Vrai / Faux)
box		Boîte rectangulaire dans le plan
bytea		Donnée binaire (« tableau d'octets »)
character [(n)]	char [(n)]	Chaîne de caractères de longueur fixe
character varying [(n)]	varchar [(n)]	Chaîne de caractères de longueur variable

Nom	Alias	Description
cidr		Adresse réseau IPv4 ou IPv6
circle		Cercle dans le plan
date		Date du calendrier (année, mois, jour)
double precision	float8	Nombre à virgule flottante de double précision (sur 8 octets)
inet		Adresse d'ordinateur IPv4 ou IPv6
integer	int, int4	Entier signé sur 4 octets
interval [champs] [(p)]		Intervalle de temps
json		Données texte JSON
jsonb		Données binaires JSON, décomposées
line		Droite (infinie) dans le plan
lseg		Segment de droite dans le plan
macaddr		Adresse MAC (pour Media Access Control)
money		Montant monétaire

Nom	Alias	Description
numeric [(p, s)]	decimal [(p, s)]	Nombre exact dont la précision peut être précisée
path		Chemin géométrique dans le plan
pg_lsn		Séquence numérique de journal (Log Sequence Number)
point		Point géométrique dans le plan
polygon		Chemin géométrique fermé dans le plan
real	float4	Nombre à virgule flottante de simple précision (sur 4 octets)
smallint	int2	Entier signé sur 2 octets
smallserial	serial2	Entier sur 2 octets à incrémentation automatique
serial	serial4	Entier sur 4 octets à incrémentation automatique
text		Chaîne de caractères de longueur variable

Nom	Alias	Description
time [(p)] [without time zone]		Heure du jour (pas du fuseau horaire)
time [(p)] with time zone	timetz	Heure du jour, avec fuseau horaire
timestamp [(p)] [without time zone]		Date et heure (pas du fuseau horaire)
timestamp [(p)] with time zone	timestamptz	Date et heure, avec fuseau horaire
tsquery		Requête pour la recherche plein texte
tsvector		Document pour la recherche plein texte
txid_snapshot		Image de l'identifiant de transaction au niveau utilisateur
uuid		Identifiant unique universel
xml		Données XML

Complément :

Voir les types de données existants : \dTS+

Les types de données suivants sont conformes à la norme SQL :

bigint, bit, bit varying, boolean, char, character varying, character, varchar, date, double precision, integer, interval, numeric, decimal, real, smallint, time (avec et sans fuseau horaire), timestamp (avec et sans fuseau horaire), xml

Exemple de création d'un nouveau type de données (voir « \h CREATE TYPE ») :

CREATE TYPE jour AS ENUM ('lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi', 'dimanche');

→ *ordre des valeurs important pour les tris et les comparaisons*

Vérifier : \dT+

Possibilité de créer des types complexes mais aussi de nouveaux opérateurs (voir « \h CREATE OPERATOR »)

Valeurs littérales :

Chaînes (respectent la casse) :

'Un texte avec apostrophe '' et guillemet " '

→ *"Une colonne" pour les noms de colonnes (ou de tables) avec espaces*

Nombres : 0 , 1254 , +153 , -256 , 132.45 , 12.00 , -21032.6309e+10

Dates et heures (entre « ' ' ») : 1999-01-08 , January 8, 1999 , 1/8/1999 , 1/18/1999 , 01/02/03 , 1999-Jan-08 , Jan-08-1999 , 08-Jan-1999 , 99-Jan-08 , 08-Jan-99 , Jan-08-99 , 19990108 , 990108 , 1999.008 , J2451187 , January 8, 99 BC , 04:05:06.789 , 04:05:06 , 04:05 , 040506 , 04:05 AM , 04:05 PM , 04:05:06.789-8 , 04:05:06-08:00 , 04:05-08:00 , 040506-08 , 04:05:06 PST , 2003-04-12 04:05:06 America/New_York

Booléens : TRUE , 't' , 'true' , 'y' , 'yes' , 'on' , '1' , FALSE , 'f' , 'false' , 'n' , 'no' , 'off' , '0'

Commentaires : -- , /* */

Contraintes d'intégrité sur les tables :

Les contraintes d'intégrité sont **utilisées pour définir des règles sur les valeurs pouvant être stockées** dans les colonnes et pour limiter le type de données pouvant être inséré dans la table

Leur rôle est donc de **faire respecter l'intégrité de la base de données**

Elles peuvent être **classées en deux types : au niveau colonne et au niveau table**

Les contraintes au niveau colonne peuvent s'appliquer à une seule colonne alors que les contraintes au niveau table sont appliquées à l'ensemble de la table

Elles sont généralement **déclarées au moment de la création de la table**

Contraintes principales avec PostgreSQL :

NOT NULL : permet de spécifier qu'une colonne ne peut pas contenir la valeur NULL (remarque : la contrainte NULL existe aussi)

CHECK : contrôle les valeurs dans la colonne associée, en déterminant si la valeur est valide ou non à partir d'une expression logique

DEFAULT : dans une table PostgreSQL, chaque colonne doit contenir une valeur (y compris un NULL). Lors de l'insertion des données dans la table, si aucune valeur n'est fournie à la colonne, celle-ci prend la valeur définie par défaut

UNIQUE : ne permet pas d'insérer de valeur en double dans une colonne, maintient ainsi l'unicité de la colonne dans la table (plusieurs colonnes UNIQUE peuvent être définies dans la même table)

PRIMARY KEY (clé primaire) : contrainte d'unicité permettant d'identifier de manière unique un enregistrement dans la table (peut être composée d'une ou plusieurs colonnes). Un index implicite est associé à cette clé primaire pour accéder plus rapidement aux données de la table

FOREIGN KEY (clé étrangère) : contrainte garantissant l'intégrité référentielle entre deux tables (liaison). Identifie une ou plusieurs colonnes d'une table comme référençant une ou plusieurs colonnes d'une autre table (référencée), garantissant ainsi l'existence des valeurs (les colonnes de la table référencée doivent faire partie d'une clé primaire ou d'une contrainte d'unicité)

Complément :

Syntaxe générale de création d'une table avec contraintes :

CREATE TABLE [table name]
([column name] [data type]([size]) [column constraint] ...,
[table constraint] ([column name], ...) ...);

3) Tables. Séquences

Syntaxe de création d'une table (voir « \h CREATE TABLE ») :

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } |
UNLOGGED ] TABLE [ IF NOT EXISTS ] table_name ( [
  { column_name data_type [ COLLATE collation ] [ column_constraint
[ ... ] ]
    | table_constraint
    | LIKE source_table [ like_option ... ] }
    [, ... ]
] );
```

Complément :

La base de données concernée doit être sélectionnée (par « \c ») et attention au schéma sélectionné

CREATE TEMPORARY TABLE : existante uniquement le temps de la session utilisateur et de la connexion à la base de données

Exemple (modèle proposé) :

```
Create table TYPES_FRAIS (
Code_typ Varchar(10) NOT NULL,
filtreclient Varchar(30) NOT NULL,
Intitule Varchar(30) NOT NULL,
Montant_Max Decimal(8,2),
Nombre_Max Decimal(5,2),
 Primary Key (Code_typ,filtreclient));
```

Voir la définition d'une table : \dt+ TYPES_FRAIS

Voir la description d'une table : \d TYPES_FRAIS

Voir les tables : \dt+

→ *uniquement pour le schéma en cours*

Complément :

Syntaxe à saisir ligne par ligne dans « psql », ou sur une seule ligne, ou encore à l'aide d'un éditeur de texte puis **en chargeant son fichier dans « psql » (script SQL) :**

\i table-typesfrais.sql

Insérer des données dans une table (exemple) :

INSERT INTO TYPES_FRAIS (Code_typ,filtreclient,Intitule,Montant_Max,Nombre_Max) VALUES ('C001','neosysta','Communication téléphone',null,null);

Visualiser tout le contenu d'une table :

SELECT * FROM TYPES_FRAIS;

Copier une table sans ses données (avec ses clés) :

CREATE TABLE COPIE_FRAIS (LIKE TYPES_FRAIS INCLUDING ALL);

Ou copier une table avec ses données (sans ses clés) :

CREATE TABLE COPIE_FRAIS AS SELECT * FROM TYPES_FRAIS;

Vider une table (ses données) : TRUNCATE TYPES_FRAIS;

Supprimer une table : DROP TABLE COPIE_FRAIS;

Modifier une table (exemples) : voir « \h ALTER TABLE »

Retirer/Ajouter NOT NULL à une colonne :

ALTER TABLE TYPES_FRAIS ALTER COLUMN Intitule DROP NOT NULL;

ALTER TABLE TYPES_FRAIS ALTER COLUMN Intitule SET NOT NULL;

Ajouter une colonne :

ALTER TABLE TYPES_FRAIS ADD COLUMN Active Char(1);

Renommer une colonne :

ALTER TABLE TYPES_FRAIS RENAME COLUMN Active TO Actif;

Modifier le type de données d'une colonne :

ALTER TABLE TYPES_FRAIS ALTER COLUMN Actif TYPE Varchar(3);

Supprimer une colonne :

ALTER TABLE TYPES_FRAIS DROP COLUMN Actif;

Supprimer la clé primaire (une seule par table) :

ALTER TABLE TYPES_FRAIS DROP CONSTRAINT types_frais_pkey; → *nom de la clé primaire (voir « \d ... »)*

Ajouter une clé primaire (une seule par table) :

ALTER TABLE TYPES_FRAIS ADD PRIMARY KEY (Code_typ,filtreclient);

Ajouter/Supprimer un index sur une colonne de table (vérifier « \d ... » et « \di ») : → *attention aux nommages car noms en commun...*

CREATE INDEX ind_Intitule ON TYPES_FRAIS(Intitule);

DROP INDEX ind_Intitule;

Renommer une table :

ALTER TABLE TYPES_FRAIS RENAME TO AUTRES_FRAIS;

ALTER TABLE AUTRES_FRAIS RENAME TO TYPES_FRAIS;

Créer les tables suivantes (modèle proposé) :

SERVICES :

```
       Column        |            Type             | Modifiers
---------------------+-----------------------------+-----------
 code_ser            | character varying(10)       | not null
 filtreclient        | character varying(30)       | not null
 intitule            | character varying(30)       | not null
 adresse1            | character varying(30)       | not null
 adresse2            | character varying(30)       |
 cp                  | character varying(5)        | not null
 commune             | character varying(30)       | not null
 pays                | character varying(30)       | not null
 nom_responsable     | character varying(30)       | not null
 prenom_responsable | character varying(30)       | not null
 creationdate        | timestamp without time zone | not null
 updatedate          | timestamp without time zone | not null
 updateby            | character varying(30)       | not null
Indexes:
    "services_pkey" PRIMARY KEY, btree (code_ser, filtreclient)
```

COLLABORATEURS :

```
    Column    |            Type             | Modifiers
--------------+-----------------------------+-----------
 code_col     | character varying(10)       | not null
 code_ser     | character varying(10)       | not null
 filtreclient | character varying(30)       | not null
 nom          | character varying(30)       | not null
 prenom       | character varying(30)       | not null
 tel          | character varying(15)       |
 fax          | character varying(15)       |
 email        | character varying(30)       | not null
 fonction     | character varying(30)       | not null
 password     | character varying(32)       |
 profile      | character varying(3)        | not null
 langue       | character varying(3)        | not null
 creationdate | timestamp without time zone | not null
 updatedate   | timestamp without time zone | not null
 updateby     | character varying(30)       | not null
Indexes:
    "collaborateurs_pkey" PRIMARY KEY, btree (code_col, code_ser,
filtreclient)
```

NOTES_FRAIS :

```
        Column         |            Type             | Modifiers
-----------------------+-----------------------------+-----------
 code_not              | character varying(10)       | not null
 code_col              | character varying(10)       | not null
 code_ser              | character varying(10)       | not null
 code_typ              | character varying(10)       | not null
 filtreclient          | character varying(30)       | not null
 intitule              | character varying(30)       | not null
 date                  | date                        | not null
 montant               | numeric(8,2)                | not null
 nombre                | numeric(5,2)                | not null
 statut                | character varying(30)       | not null
 montant_avance        | numeric(8,2)                |
 date_avance           | date                        |
 montant_remboursement | numeric(8,2)                |
 date_remboursement    | date                        |
 mode_remboursement    | character varying(30)       |
 creationdate          | timestamp without time zone | not null
 updatedate            | timestamp without time zone | not null
 updateby              | character varying(30)       | not null
Indexes:
    "notes_frais_pkey" PRIMARY KEY, btree (code_not, code_col, code_ser, code_typ,
filtreclient)
```

Ajouter les clés étrangères suivantes (modèle proposé) :

Alter table COLLABORATEURS add Foreign Key (Code_ser,filtreclient) references SERVICES (Code_ser,filtreclient) on delete restrict on update restrict;

Alter table NOTES_FRAIS add Foreign Key (Code_col,Code_ser,filtreclient) references COLLABORATEURS (Code_col,Code_ser,filtreclient) on delete restrict on update restrict;

Vérifier :

\d SERVICES

\d COLLABORATEURS

\d NOTES_FRAIS

Complément :

Supprimer une clé étrangère (exemple) :

ALTER TABLE NOTES_FRAIS DROP CONSTRAINT notes_frais_code_col_fkey;

Clauses possibles sur une clé étrangère :

[**ON DELETE** {CASCADE | SET NULL | SET DEFAULT | NO ACTION | RESTRICT}]

[**ON UPDATE** {CASCADE | SET NULL | SET DEFAULT | NO ACTION | RESTRICT}]

Exemple de création d'une séquence de nombres entiers (voir « \h CREATE SEQUENCE ») :

CREATE SEQUENCE tablename_colname_seq;

CREATE TABLE tablename (
colname integer DEFAULT nextval('tablename_colname_seq'));

Vérifier : « \ds », « \d nbpair » et « SELECT * FROM nbpair; »

Les fonctions « currval », « nextval » et « setval » sont ensuite utilisées pour agir sur la séquence

Avec PostgreSQL, **il existe aussi le type « SERIAL »** (similaire à « AUTO_INCREMENT » de MySQL) :

CREATE TABLE tablename (colname SERIAL);

4) Ajout, modification et suppression des données

Insérer des données dans toutes les tables (prérequis) :

INSERT INTO TYPES_FRAIS
(Code_typ,filtreclient,Intitule,Montant_Max,Nombre_Max) VALUES
('F001','neosysta','Forfait transport',null,null);

INSERT INTO TYPES_FRAIS
(Code_typ,filtreclient,Intitule,Montant_Max,Nombre_Max) VALUES
('H001','neosysta','Hôtel',null,null);

...

Complément :

Autres syntaxes d'insertion (voir « \h INSERT ») :

INSERT INTO TYPES_FRAIS VALUES
('T008','neosysta','Parking',null,null); → *avec valeurs concordantes*

INSERT INTO table SELECT * FROM autre_table; → *recopie*

```
INSERT INTO SERVICES
(Code_ser,filtreclient,Intitule,Adresse1,Adresse2,CP,Commune,Pays,No
m_Responsable,Prenom_Responsable,CreationDate,UpdateDate,Update
By) VALUES
('S000','neosysta','Direction','neosysta','','75000','PARIS','FRANCE','MA
RTIN','Sébastien','2013-12-16 15:23:04','2014-01-02 19:09:17','smartin');

INSERT INTO SERVICES
(Code_ser,filtreclient,Intitule,Adresse1,Adresse2,CP,Commune,Pays,No
m_Responsable,Prenom_Responsable,CreationDate,UpdateDate,Update
By) VALUES
('S001','neosysta','Informatique','neosysta','','75000','PARIS','FRANCE','
MARTIN','Sébastien','2013-12-19 22:21:42','2013-12-19
22:27:19','smartin');
...
```

```
INSERT INTO COLLABORATEURS
(Code_col,Code_ser,filtreclient,Nom,Prenom,Tel,Fax,Email,Fonction,Password,Profile,Langue,CreationDate,UpdateDate,UpdateBy) VALUES
('smartin','S000','neosysta','MARTIN','Sébastien','01 00 00 00 00','01 00 00 00 00
01','sebastien.martin@fauxmail.fr','Directeur','','ADM','ENG','2013-12-16 15:26:08','2014-01-05 14:39:25','smartin');

INSERT INTO COLLABORATEURS
(Code_col,Code_ser,filtreclient,Nom,Prenom,Tel,Fax,Email,Fonction,Password,Profile,Langue,CreationDate,UpdateDate,UpdateBy) VALUES
('omuller','S001','neosysta','MULLER','Olivier','','','olivier.muller@fauxmail.fr','Développeur','','COL','FRA','2013-12-30 11:31:17','2014-01-02 19:08:40','smartin');
...
```

```
INSERT INTO NOTES_FRAIS
(Code_not,Code_col,Code_ser,Code_typ,filtreclient,Intitule,Date,Montant
,Nombre,Statut,Montant_Avance,Date_Avance,Montant_Remboursement
,Date_Remboursement,Mode_Remboursement,CreationDate,UpdateDate,
UpdateBy) VALUES ('N0001','smartin','S000','R002','neosysta','Repas
client','2013-12-30',50.00,4.00,'TRT',50.00,'2013-12-22',150.00,'2013-12-
31','CHQ','2013-12-30 17:18:17','2014-01-04 13:14:05','smartin');

INSERT INTO NOTES_FRAIS
(Code_not,Code_col,Code_ser,Code_typ,filtreclient,Intitule,Date,Montant
,Nombre,Statut,Montant_Avance,Date_Avance,Montant_Remboursement
,Date_Remboursement,Mode_Remboursement,CreationDate,UpdateDate,
UpdateBy) VALUES ('N0002','smartin','S000','F001','neosysta','Transport
Paris','2013-12-31',50.00,1.00,'VAL',null,null,null,null,null,'2013-12-31
09:45:29','2014-01-04 13:09:55','smartin');
...
```

Syntaxe de mise à jour d'enregistrements (voir « \h UPDATE ») :

```
UPDATE [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    SET { column_name = { expression | DEFAULT } |
          ( column_name [, ...] ) = ( { expression | DEFAULT } [, ...] ) }
[, ...]
    [ FROM from_list ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ];
```

Exemples de requêtes courantes :

```
UPDATE SERVICES SET Adresse2 = 'BP 111';

UPDATE TYPES_FRAIS
SET Montant_Max = 100, Nombre_Max = 1
WHERE Code_typ LIKE 'C%';
```

Attention : UPDATE sans WHERE modifie tous les enregistrements

Complément :

Les requêtes imbriquées sont possibles (mais impossible de mettre à jour une table qui se lit déjà en sous-requête)

Syntaxe de suppression d'enregistrements (voir « \h DELETE ») :

```
DELETE FROM [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    [ USING using_list ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ];
```

Exemple :

```
DELETE FROM TYPES_FRAIS WHERE Code_typ = 'T008';
```

Attention : DELETE sans WHERE supprime tous les enregistrements

Complément :

Vider une table (ses données) : TRUNCATE [TABLE] table_name;

5) Interrogation du schéma d'une base. Sélection, restriction, tri, jointure

Syntaxe de sélection d'enregistrements (voir « \h SELECT ») :

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    [ * | expression [ [ AS ] output_name ] [, ...] ]
    [ FROM from_item [, ...] ] [ WHERE condition ]
    [ GROUP BY expression [, ...] ] [ HAVING condition [, ...] ]
    [ WINDOW window_name AS ( window_definition ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS
{ FIRST | LAST } ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ];
```

Complément :

Egalement pour afficher le résultat de fonctions (sans table) :

```
SELECT CURRENT_DATABASE();
SELECT CURRENT_DATE;
SELECT USER;
```

Et des valeurs ou le résultat de calculs :

```
SELECT 123;
SELECT 'Hello';
SELECT 3 + 2;
```

Quelques exemples (requêtes courantes) :

SELECT * FROM TYPES_FRAIS;

TABLE TYPES_FRAIS;

SELECT * FROM TYPES_FRAIS LIMIT 10 OFFSET 5;

SELECT DISTINCT Montant_Max FROM TYPES_FRAIS;

SELECT Prenom, Nom, Code_ser FROM COLLABORATEURS WHERE Code_ser = 'S000';

SELECT count(*) FROM TYPES_FRAIS WHERE Intitule LIKE 'Forfait%';

Complément :

Lister toutes les colonnes prend moins de temps de traitement que « * »

Le joker « % » et « IN » sont gourmands en ressources

Conseil : écrire la syntaxe d'une requête complexe étape par étape

Quelques exemples (suite) :

```
SELECT Code_typ, Intitule, Montant_Max FROM TYPES_FRAIS
WHERE Intitule LIKE 'Repas%' AND Montant_Max IS NOT NULL;
```

```
SELECT * FROM TYPES_FRAIS WHERE Code_typ IN
('F001','H001','T001');
```

```
SELECT * FROM TYPES_FRAIS WHERE Code_typ BETWEEN
'C002' AND 'R001';
```

```
SELECT Code_ser as Service, concat(Nom,' ',Prenom) as Contact
FROM COLLABORATEURS ORDER BY Contact DESC;
```

Sélection avec jointure simple entre 2 tables (2 syntaxes) :

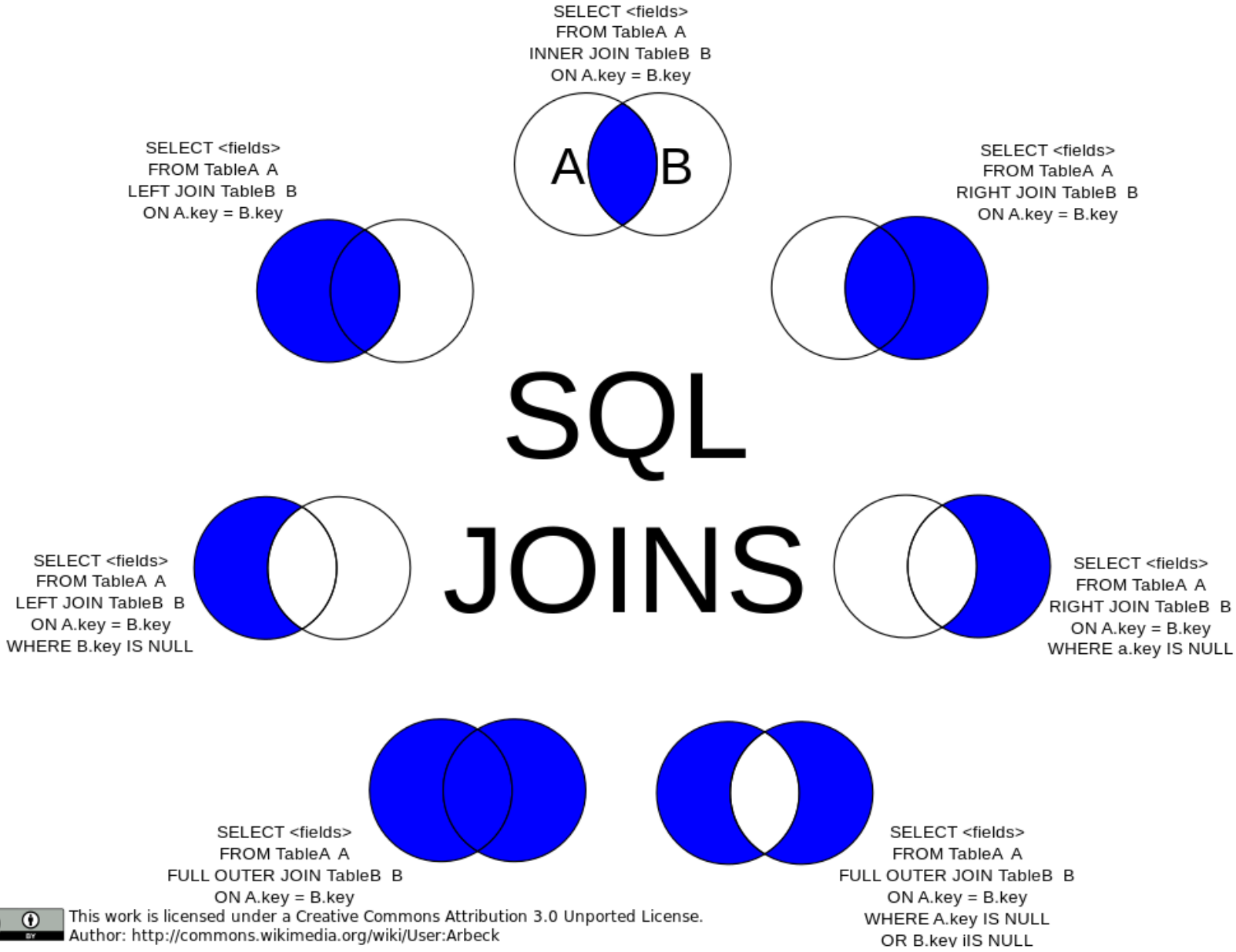
→ *par clés étrangères / primaires (en général)*

```
SELECT C.Prenom, C.Nom, C.Fonction, S.Intitule as Service
FROM COLLABORATEURS C, SERVICES S
WHERE C.Code_ser = S.Code_ser
AND C.filtreclient = S.filtreclient;
```

```
SELECT C.Prenom, C.Nom, C.Fonction, S.Intitule as Service
FROM COLLABORATEURS C JOIN SERVICES S
ON C.Code_ser = S.Code_ser
AND C.filtreclient = S.filtreclient;
```

Complément :

Il est possible de joindre plus de 2 tables (jointures multiples)



Sélections avec INNER / LEFT / RIGHT JOIN entre 2 tables :

```
SELECT C.Prenom, C.Nom, C.Fonction, S.Intitule as Service
FROM COLLABORATEURS C INNER JOIN SERVICES S
ON C.Code_ser = S.Code_ser AND C.filtreclient = S.filtreclient;
```

```
SELECT C.Prenom, C.Nom, C.Fonction, S.Intitule as Service
FROM COLLABORATEURS C LEFT JOIN SERVICES S
ON C.Code_ser = S.Code_ser AND C.filtreclient = S.filtreclient;
```

```
SELECT C.Prenom, C.Nom, C.Fonction, S.Intitule as Service
FROM COLLABORATEURS C RIGHT JOIN SERVICES S
ON C.Code_ser = S.Code_ser AND C.filtreclient = S.filtreclient;
```

6) Extractions complexes

Exemples de regroupements (utiles pour les fonctions d'agrégation COUNT, MIN, MAX, SUM, AVG,...) :

SELECT substr(Code_typ,1,1) as Type, Intitule FROM TYPES_FRAIS WHERE Intitule LIKE 'Forfait%' OR Intitule LIKE 'Repas%';

SELECT count(*) as Total, substr(Code_typ,1,1) as Type FROM TYPES_FRAIS WHERE Intitule LIKE 'Forfait%' OR Intitule LIKE 'Repas%'; → *problème de regroupement / agrégation*

SELECT count(*) as Total, substr(Code_typ,1,1) as Type FROM TYPES_FRAIS WHERE Intitule LIKE 'Forfait%' OR Intitule LIKE 'Repas%' GROUP BY Type;

Complément :

SELECT count(*) as Total, substr(Code_typ,1,1) as Type FROM TYPES_FRAIS WHERE Intitule LIKE 'Forfait%' OR Intitule LIKE 'Repas%' GROUP BY Type HAVING count(*) > 2;

Précision : « HAVING Total > 2 » ne fonctionne pas...

Union de sélections (même nombre de colonnes et de mêmes types) :

```
SELECT Nom_Responsable as Nom, Prenom_Responsable as Prenom
FROM SERVICES
UNION ALL
SELECT Nom, Prenom FROM COLLABORATEURS;
```

```
SELECT Nom_Responsable as Nom, Prenom_Responsable as Prenom
FROM SERVICES
UNION
SELECT Nom, Prenom FROM COLLABORATEURS;
```

Complément :

L'UNION simple ne retourne pas les enregistrements en double (équivalent à UNION DISTINCT)

Exemples de requêtes imbriquées :

```
SELECT * FROM NOTES_FRAIS WHERE Montant IN (
SELECT Montant_Max FROM TYPES_FRAIS);

SELECT * FROM NOTES_FRAIS WHERE Montant >= ALL (
SELECT Montant_Max FROM TYPES_FRAIS WHERE Montant_Max
IS NOT NULL);

SELECT * FROM NOTES_FRAIS WHERE Montant >= ANY (
SELECT Montant_Max FROM TYPES_FRAIS WHERE Montant_Max
IS NOT NULL);
```

Complément :

Les requêtes imbriquées sont gourmandes en ressources (privilégier les vues)

7) Fonctions intégrées. Vues

Fonctions de comparaison :

LIKE, NOT LIKE, SIMILAR TO, NOT SIMILAR TO, BETWEEN AND, NOT BETWEEN AND, IS NULL, IS NOT NULL, IS DISTINCT FROM, IS NOT DISTINCT FROM, IS TRUE, IS NOT TRUE, IS FALSE, IS NOT FALSE, IS UNKNOWN, IS NOT UNKNOWN,...

Opérateurs de comparaison : <, >, <=, >=, =, <>, !=

Opérateurs logiques : AND, OR, NOT

Complément :

Listes non exhaustives, voir aussi : → *souvent spécifiques PostgreSQL*

http://www.postgresql.org/docs/current/static/functions.html
http://docs.postgresql.fr/x.x/functions.html

\df * → *liste des fonctions*
\df fonction → *pour une fonction*
\do * → *liste des opérateurs*
\do opérateur → *pour un opérateur*

Fonctions de contrôle :

CASE/WHEN/THEN/ELSE/END, COALESCE, NULLIF, GREATEST, LEAST,...

Fonctions chaînes :

bit_length, char_length, character_length, lower, octet_length, overlay, position, substring, trim, upper, ascii, btrim, chr, concat, concat_ws, convert, convert_from, convert_to, decode, encode, format, initcap, left, length, lpad, ltrim, md5, pg_client_encoding, quote_ident, quote_literal, quote_nullable, regexp_matches, regexp_replace, regexp_split_to_array, regexp_split_to_table, repeat, replace, reverse, right, rpad, rtrim, split_part, strpos, substr, to_ascii, to_hex, translate,...

Fonctions mathématiques :

abs, cbrt, ceil, ceiling, degrees, div, exp, floor, ln, log, mod, pi, power, radians, round, sign, sqrt, trunc, width_bucket, random, setseed, acos, asin, atan, atan2, cos, cot, sin, tan,...

Opérateurs mathématiques : +, -, *, /, %, ^, |/, ||/, !, !!, @

Opérateurs binaires : &, |, #, ~, <<, >>

Fonctions dates et heures : age, clock_timestamp, current_date, current_time, current_timestamp, date_part, date_trunc, extract, isfinite, justify_days, justify_hours, justify_interval, localtime, localtimestamp, make_date, make_interval, make_time, make_timestamp, make_timestamptz, now, statement_timestamp, timeofday, transaction_timestamp,...

Fonctions de formatage des types de données :

to_char, to_date, to_number, to_timestamp,...

Fonctions d'informations système :

current_catalog, current_database, current_query, current_schema, current_schemas, current_user, inet_client_addr, inet_client_port, inet_server_addr, inet_server_port, pg_backend_pid, pg_conf_load_time, pg_is_other_temp_schema, pg_listening_channels, pg_my_temp_schema, pg_postmaster_start_time, pg_trigger_depth, session_user, user, version,...

Complément :

La plupart des fonctions peuvent être testées avec « SELECT »

Gestion des vues :

Les vues permettent de **définir une représentation sous forme de table résultant d'une requête SELECT complexe** (avec éventuellement liaison de plusieurs tables ou même d'autres vues)

Utiliser des vues **optimise les traitements, facilite et renforce la sécurité** de la base de données (tables restreintes pour certains utilisateurs)

Création d'une vue (exemple) : voir « \h CREATE VIEW »

```
CREATE OR REPLACE VIEW colparser
(Prenom, Nom, Fonction, Service) AS
SELECT C.Prenom, C.Nom, C.Fonction, S.Intitule
FROM COLLABORATEURS C INNER JOIN SERVICES S
ON C.Code_ser = S.Code_ser AND C.filtreclient = S.filtreclient;
```

Voir la définition d'une vue : \d+ colparser
Voir les vues : \dv+

Utilisation d'une vue (exemple) :

SELECT * FROM colparser ORDER BY Nom;

Modification de la définition d'une vue (voir « \h ALTER VIEW ») :

ALTER VIEW [IF EXISTS] view_name ...;

Suppression d'une vue (voir « \h DROP VIEW ») :

DROP VIEW colparser;

Complément :

Certaines vues peuvent être mises à jour par INSERT, UPDATE et DELETE (dépend de leur structure)

8) Transactions

Les transactions permettent de garantir qu'une suite de commandes SQL, introduite par « START TRANSACTION ou BEGIN » et terminée par « COMMIT » ou « ROLLBACK », **soit exécutée intégralement soit pas du tout** (même en cas de coupure serveur)

Si la suite d'opérations ne déclenche pas d'erreurs, l'ensemble peut être **validé (commit)** sinon **annulé (rollback)**

PostgreSQL dispose d'un gestionnaire de transactions très évolué (système d'isolation **MVCC, « MultiVersion Concurrency Control »**), avec validation (commit), annulation (rollback) et restauration (après crash)

Les transactions de PostgreSQL **respectent les propriétés ACID** : Atomicity (atomicité), Consistency (cohérence), Isolation (isolation), Durability (durabilité)

Atomicité : assure qu'une transaction se fait au complet ou pas du tout

Cohérence : assure que chaque transaction amène le système d'un état valide à un autre état valide (sans erreurs, contraintes d'intégrité respectées, rollbacks en cascade et déclencheurs garantis,…)

Isolation : assure que l'exécution simultanée de plusieurs transactions produit le même état qu'une exécution en série (sans interférences)

Durabilité : assure que lorsqu'une transaction a été confirmée, celle-ci demeure enregistrée même à la suite d'une panne ou d'un problème

Utilisation des transactions :

Par défaut (utilisation normale), toute requête est automatiquement validée à la fin de son traitement (mode "autocommit")

Pour démarrer une transaction :

START TRANSACTION;

ou

BEGIN;

Une fois la transaction terminée, on la valide par :

COMMIT;

ou on l'annule par :

ROLLBACK;

Complément :

Attention :

La validation par « COMMIT » ou l'annulation par « ROLLBACK » réactivent le mode "autocommit"

En cours de transaction, **on peut définir des points de sauvegarde intermédiaires :**

SAVEPOINT point_sauvegarde;

Puis annuler uniquement jusqu'à ce point :

ROLLBACK TO SAVEPOINT point_sauvegarde;

La validation par « COMMIT » ou l'annulation par « ROLLBACK » suppriment les points de sauvegarde

L'annulation par « ROLLBACK TO SAVEPOINT » supprime uniquement les points de sauvegarde suivants

Complément :

Attention :

Un « ROLLBACK TO SAVEPOINT » à un point de sauvegarde inexistant – de même que toute autre erreur de traitement – plante la transaction (ROLLBACK complet)

Exemple classique :

SELECT * FROM TYPES_FRAIS;

START TRANSACTION; → *ou « BEGIN; »*

INSERT INTO TYPES_FRAIS VALUES
('T008','neosysta','Parking',null,null);

SELECT * FROM TYPES_FRAIS;

SAVEPOINT insert1;

```
INSERT INTO TYPES_FRAIS VALUES
('T009','neosysta','Péage',null,null);

SELECT * FROM TYPES_FRAIS;

SAVEPOINT insert2;

ROLLBACK TO SAVEPOINT insert1;

SELECT * FROM TYPES_FRAIS;

ROLLBACK;

SELECT * FROM TYPES_FRAIS;
```

Complément :

« COMMIT » aurait validé la transaction (avec ses dernières modifications effectuées)

Voir également ce que donnent les traitements depuis une autre session (tant que la transaction n'est pas validée ou annulée)

Conseil : préférer les transactions courtes

VII. Introduction à PL/pgSQL

VII. Introduction à PL/pgSQL

1) Principes des procédures et fonctions
2) Quand utiliser le langage PL/SQL ?
3) Création, modification et suppression de fonctions

1) Principes des procédures et fonctions

PL/pgSQL est un langage de programmation procédurale utilisé pour créer des fonctions standards et des triggers

→ *procédures stockées sur le serveur de bases de données (mais pas de procédures au sens propre ni de « CREATE PROCEDURE »)*

A partir de la version 9.0 de PostgreSQL, **PL/pgSQL est installé par défaut** (mais reste néanmoins un module chargeable, parmi d'autres, que les administrateurs craignant pour la sécurité peuvent retirer / remplacer)

Il permet d'utiliser des variables (types de données PostgreSQL ou créés), les ordres SQL classiques, des calculs, des structures de contrôle, les fonctions intégrées de PostgreSQL, d'autres fonctions créées,...

Complément :

Les fonctions créées s'utilisent ensuite comme les fonctions intégrées de PostgreSQL

Extensions au SQL :

Délimiteur : les fonctions contenant des requêtes se terminant par ';', on doit changer temporairement le délimiteur du « CREATE FUNCTION » en indiquant « AS $$ » avant le code source, puis « $$ » après celui-ci

Retourner un résultat (si nécessaire) : → *void si aucun*

Mots clés « RETURNS » (déclaration) et « RETURN » (renvoi)

Déclarer des variables (locales) : → *« DECLARE » avant « BEGIN »*

nom [CONSTANT] type [COLLATE nom_collationnement]

[NOT NULL] [{DEFAULT | := | =} expression];

Affecter des variables (calculs possibles) : variable := valeur;

Récupérer une valeur de requête : → *si ligne : table%ROWTYPE*

SELECT expression INTO variable FROM ...;

Utiliser des structures de contrôle (tests et itérations) :

IF, CASE, LOOP, WHILE, FOR, FOREACH,… → *CONTINUE / EXIT*

2) Quand utiliser le langage PL/SQL ?

Avantages des procédures stockées / fonctions :

Réduisent le trafic du réseau (traitements côté serveur)

Peuvent être appelées plusieurs fois depuis n'importe quel langage (PHP, Java,…)

Préservent une cohérence de la base de données (modifiables sans toucher aux programmes qui les appellent)

Permettent aux utilisateurs qui n'ont pas accès à une table de récupérer ses données ou de les modifier dans certaines circonstances

3) Création, modification et suppression de fonctions

Syntaxe de création d'une fonction (voir « \h CREATE FUNCTION ») :

```
CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
    [ RETURNS rettype
      | RETURNS TABLE ( column_name column_type [, ...] ) ]
```

→ *accepte la surcharge des fonctions : noms identiques de fonctions avec arguments différents (attention : liste / type des arguments à bien préciser lors des utilisations / modifications / suppressions)*

```
  { LANGUAGE lang_name
    | WINDOW
    | IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT |
STRICT
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ]
SECURITY DEFINER
    | COST execution_cost
    | ROWS result_rows
    | SET configuration_parameter { TO value | = value | FROM
CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
  } ...
    [ WITH ( attribute [, ...] ) ];
```

Modification de la définition d'une fonction (voir « \h ALTER FUNCTION ») : → *uniquement ses caractéristiques*

ALTER FUNCTION name ([[argmode] [argname] argtype [, ...]])

action [...] [RESTRICT] | RENAME TO new_name | OWNER TO new_owner | SET SCHEMA new_schema;

action : CALLED ON | RETURNS NULL ON NULL INPUT | STRICT

IMMUTABLE | STABLE | VOLATILE | [NOT] LEAKPROOF

[EXTERNAL] SECURITY INVOKER | SECURITY DEFINER

COST execution_cost | ROWS result_rows

SET configuration_parameter { TO | = } { value | DEFAULT }

SET configuration_parameter FROM CURRENT

RESET configuration_parameter | RESET ALL

Suppression d'une fonction (voir « \h DROP FUNCTION ») :

DROP FUNCTION [IF EXISTS] name (...) [CASCADE | RESTRICT];

Complément :

En cas de suppression « DROP » et de recréation « CREATE » d'une fonction, la nouvelle fonction n'est pas la même entité que l'ancienne :

il faut supprimer les vues, déclencheurs, règles, etc. qui référencent l'ancienne fonction puis les recréer sur la nouvelle...

« CREATE OR REPLACE FUNCTION » permet de modifier la définition d'une fonction sans casser les objets qui s'y réfèrent

De plus, « ALTER FUNCTION » peut être utilisé pour modifier la plupart des propriétés supplémentaires d'une fonction existante

→ *de même pour d'autres objets PostgreSQL (attention)*

VIII. Le langage PL/SQL

VIII. Le langage PL/SQL

1) Principes essentiels du langage. Structure d'un script complet. Les types de données

2) Expression et exécution automatiques de requêtes simples

3) Passage de paramètres

4) Les tests et les boucles

5) Manipulation avancée des curseurs

1) Principes essentiels du langage. Structure d'un script complet. Les types de données

```
CREATE OR REPLACE FUNCTION compteur(pmax int)
RETURNS SETOF int AS $$
DECLARE
  n int := 1;
BEGIN
  WHILE n <= pmax LOOP
    RETURN NEXT n;
    n := n + 1;
  END LOOP;
END;
$$ LANGUAGE plpgsql;
```

Complément :

« RETURNS SETOF » et « RETURN NEXT » car plusieurs valeurs retournées

Exemple d'exécution : SELECT compteur(5);

Autre exemple :

```
CREATE OR REPLACE FUNCTION combine(chaine1
VARCHAR(10), chaine2 VARCHAR(30))
RETURNS VARCHAR(50) AS $$
DECLARE
   chaine VARCHAR(50);
BEGIN
   chaine := CONCAT(chaine1,' : ',chaine2);
   RETURN chaine;
END;
$$ LANGUAGE plpgsql;
```

Complément :

Exemples d'utilisations :

SELECT combine('123','texte') as Résultat;

SELECT combine(Code_typ,Intitule) as Frais FROM TYPES_FRAIS;

Types de données (variables, paramètres) utilisables en PL/SQL :

Les mêmes que pour les colonnes de tables (voir plus haut)

Voir le code source d'une fonction :

\sf compteur
\sf combine

Voir les fonctions :

\df
\df+ → *avec codes sources*

2) Expression et exécution automatiques de requêtes simples

```
CREATE OR REPLACE FUNCTION ligne()
RETURNS TYPES_FRAIS AS $$
DECLARE
  ligne TYPES_FRAIS%ROWTYPE;
BEGIN
  SELECT * INTO ligne FROM TYPES_FRAIS
  WHERE Code_typ = 'T001';
  ligne.Code_typ := 'T010';
  ligne.Intitule := 'VTC';
  INSERT INTO TYPES_FRAIS VALUES (ligne.*);
  RETURN ligne;
END;
$$ LANGUAGE plpgsql;
```

Complément :

Exécution : SELECT ligne();

Résultat : table TYPES_FRAIS;

3) Passage de paramètres

Paramètres en entrée (IN), en sortie (OUT) ou les deux (INOUT) :

→ *IN par défaut si non précisé*

```
CREATE FUNCTION somme_produit(x int, y int, OUT somme int,
OUT produit int) AS $$
BEGIN
  somme := x + y;
  produit := x * y;
END;
$$ LANGUAGE plpgsql;
SELECT somme_produit(3,7);
```

DROP FUNCTION somme_produit(int,int); → *attention aux paramètres*

4) Les tests et les boucles

IF-THEN :

```
IF boolean-expression THEN
   statements
END IF;
```

IF-THEN-ELSE :

```
IF boolean-expression THEN
   statements
ELSE
   statements
END IF;
```

IF-THEN-ELSIF :

```
IF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
    ...]]
[ ELSE
    statements ]
END IF;
```

CASE :

```
CASE search-expression
    WHEN expression [, expression [ ... ]] THEN
      statements
  [ WHEN expression [, expression [ ... ]] THEN
      statements
    ... ]
  [ ELSE
      statements ]
END CASE;
```

LOOP :

```
LOOP
   statements
END LOOP;
```

WHILE :

```
WHILE boolean-expression LOOP
   statements
END LOOP;
```

FOR :

FOR name IN [REVERSE] expression .. expression [BY expression]
LOOP
    statements
END LOOP;

FOREACH :

FOREACH target [SLICE number] IN ARRAY expression LOOP
    statements
END LOOP;

5) Manipulation avancée des curseurs

Les curseurs **permettent de traiter différemment chaque ligne d'un SELECT**, avec possibilité de la mettre à jour (UPDATE ou DELETE et « WHERE CURRENT OF curseur; »)

Ils doivent être déclarés (**DECLARE**) puis ouverts (**OPEN**) avant le début de la boucle qui traite chaque enregistrement (**FETCH**), et fermés (**CLOSE**) une fois utilisés **(voir « \h ... »)**

On peut aussi repositionner librement un curseur sans récupérer de données (**MOVE**)

Plusieurs curseurs peuvent être définis dans la même fonction (avec des noms distincts)

Complément :

Attention : les curseurs peuvent ralentir considérablement les traitements

```
CREATE OR REPLACE FUNCTION concatcol()
RETURNS TEXT AS $$
DECLARE
  resultat VARCHAR(30) DEFAULT '';
  total TEXT DEFAULT '';
  c1 CURSOR FOR SELECT Intitule FROM TYPES_FRAIS;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO resultat;
    EXIT WHEN NOT FOUND;
    total := concat(total,resultat,' ');
  END LOOP;
  CLOSE c1;
  RETURN total;
END;
$$ LANGUAGE plpgsql;
```

Complément :

Concatène toutes les valeurs d'une colonne sur une seule ligne

Exécution : SELECT concatcol();

IX. Eléments avancés de PL/SQL et SQL

IX. Eléments avancés de PL/SQL et SQL

1) La gestion des erreurs par les exceptions
2) Les déclencheurs (triggers)
3) Sécurisation des fonctions
4) Les index. L'optimiseur. EXPLAIN

1) La gestion des erreurs par les exceptions

PostgreSQL dispose d'erreurs types avec codes erreurs (SQLSTATE)

Celles-ci peuvent être contrôlées dans les fonctions :

```
CREATE OR REPLACE FUNCTION testerreur()
RETURNS void AS $$
BEGIN
  DELETE FROM COLLABORATEURS WHERE Code_col = 'smartin';
  EXCEPTION → à définir à la fin du bloc de code (juste avant END; )
  WHEN SQLSTATE '23000' THEN → autant de WHEN que nécessaire
    RAISE NOTICE 'Violation de contrainte d''intégrité'; → ou autre traitement
END;
$$ LANGUAGE plpgsql;
```

Complément :

Tester : SELECT testerreur();

→ *dès qu'une erreur se produit, le traitement du bloc de code s'arrête : l'exception (si présente) se déclenche et traite le cas d'erreur correspondant (si prévu)*

Liste des codes erreurs PostgreSQL :

http://www.postgresql.org/docs/current/static/errcodes-appendix.html

http://docs.postgresql.fr/x.x/errcodes-appendix.html

2) Les déclencheurs (triggers)

Un trigger est associé à une fonction (procédure stockée) qui est **exécutée lorsqu'un événement particulier survient sur une table ou une vue** (INSERT, UPDATE, DELETE ou TRUNCATE)

On peut préciser qu'il **s'active avant (BEFORE), après (AFTER) ou à la place (INSTEAD OF)** de la commande qui le déclenche, et utiliser dans la fonction exécutée les mots clés :

« OLD.colonne » pour faire référence à une colonne d'une ligne avant modification ou suppression

« NEW.colonne » pour faire référence à une colonne d'une ligne après modification ou insertion

Complément :

Utiles pour des vérifications de valeurs avant mise à jour (règles de gestion complexes), ou pour des calculs de macrodonnées (audits, rapports, statistiques,...) après modification des données d'une table

Syntaxe de création d'un trigger (voir « \h CREATE TRIGGER ») :

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER |
INSTEAD OF } { event [ OR ... ] }
    ON table_name
    [ FROM referenced_table_name ]
    [ NOT DEFERRABLE | [ DEFERRABLE ] { INITIALLY
IMMEDIATE | INITIALLY DEFERRED } ]
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE PROCEDURE function_name ( arguments );

event :
    INSERT | UPDATE | DELETE | TRUNCATE
    [ OF column_name [, ... ] ]
```

Complément :

Plusieurs événements peuvent être précisés en les séparant par OR

Le nom d'un trigger ne peut pas être qualifié d'un nom de schéma, le déclencheur héritant du schéma de sa table (et de son propriétaire/owner)

Des triggers peuvent avoir le même nom entre différentes tables, mais un nom distinct dans la même table → *définition par table*

Si plusieurs triggers du même genre sont définis pour le même événement, ils sont déclenchés suivant l'ordre alphabétique de leur nom

Exemple de fonction exécutée par un trigger :

```
CREATE OR REPLACE FUNCTION montantmax() → définition arguments inutile
RETURNS TRIGGER AS $$ → fonction trigger (utilisable par plusieurs triggers)
DECLARE
  mntmax numeric(8,2);
BEGIN
  SELECT t.Montant_Max INTO mntmax FROM TYPES_FRAIS t
  WHERE t.Code_typ = NEW.Code_typ AND t.filtreclient = NEW.filtreclient;
  IF NEW.Montant > mntmax THEN
    NEW.Montant := mntmax;
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trig_montantmax BEFORE INSERT ON NOTES_FRAIS
FOR EACH ROW EXECUTE PROCEDURE montantmax();
```

Complément :

Exemple d'utilisation (80.00 > 50.00) :

INSERT INTO NOTES_FRAIS (Code_not,Code_col,Code_ser,Code_typ,filtreclient,Intitule,Date,Montant,Nombre,Statut,Montant_Avance,Date_Avance,Montant_Remboursement,Date_Remboursement,Mode_Remboursement,CreationDate,UpdateDate,UpdateBy) VALUES ('N0003','smartin','S000','R002','neosysta','Repas client','2013-12-30',**80.00**,4.00,'TRT',50.00,'2013-12-22',150.00,'2013-12-31','CHQ','2013-12-30 17:18:17','2014-01-04 13:14:05','smartin');

SELECT * FROM NOTES_FRAIS;

SELECT * FROM TYPES_FRAIS;

Voir le code source d'une fonction trigger : \dft+ montantmax

Voir les fonctions triggers :

\dft

\dft+ → *avec codes sources*

Voir les triggers d'une table : \d NOTES_FRAIS

Modification de la définition d'un trigger (voir « \h ALTER TRIGGER ») :

ALTER TRIGGER name ON table_name RENAME TO new_name;

Suppression d'un trigger (voir « \h DROP TRIGGER ») :

DROP TRIGGER [IF EXISTS] name ON table_name [CASCADE | RESTRICT];

3) Sécurisation des fonctions

Rappel : \h CREATE FUNCTION

Avec : SECURITY INVOKER | SECURITY DEFINER

SECURITY INVOKER spécifie que la fonction est exécutée avec les **droits de l'utilisateur qui l'appelle** → *propriété par défaut*

SECURITY DEFINER spécifie que la fonction est exécutée avec les **droits de l'utilisateur qui l'a créée** → *owner (attention)*

Vérifier : \df+

Complément :

admin : CREATE ROLE usertest LOGIN PASSWORD 'password';

usertest : SELECT concatcol(); → *« \dp » et « \df+ » (invoker)*

admin : GRANT SELECT ON TYPES_FRAIS TO usertest; → *« \dp »*

usertest : SELECT concatcol();

admin : REVOKE EXECUTE ON FUNCTION concatcol() FROM usertest;

usertest : SELECT concatcol();

admin : REVOKE EXECUTE ON FUNCTION concatcol() FROM PUBLIC; → *attention*

usertest : SELECT concatcol();

admin : GRANT EXECUTE ON FUNCTION concatcol() TO usertest;

usertest : SELECT concatcol();

admin : REVOKE SELECT ON TYPES_FRAIS FROM usertest; → *« \dp »*

usertest : SELECT concatcol();

admin : ALTER FUNCTION concatcol() SECURITY DEFINER; → *« \df+ » (definer)*

usertest : SELECT concatcol(); → *avec admin (owner)*

4) Les index. L'optimiseur. EXPLAIN

Les index sont des structures qui **accélèrent la recherche, le tri et le regroupement** des données → *souvent associés à une clé*

Des index **peuvent être ajoutés à des colonnes d'une table existante avec la commande « CREATE INDEX »** (« CREATE TABLE » ne crée des index que sur les contraintes UNIQUE et PRIMARY KEY, automatiquement et par défaut)

Un index multi-colonnes peut être aussi créé sur plusieurs colonnes

→ *l'optimiseur de requêtes de PostgreSQL utilise ensuite ces index lors des traitements*

Les index sont **formés par concaténation des valeurs des colonnes spécifiées**

« CREATE INDEX » ne peut pas être utilisé pour créer une clé primaire (PRIMARY KEY)

Syntaxe générale de création d'un index (sur des colonnes de table) :

CREATE [UNIQUE] INDEX [index_name]

ON table_name [USING method] (column_name, ...);

→ *PostgreSQL possède plusieurs méthodes d'indexation (via des algorithmes évolués et complexes, comme par défaut « btree »)*

EXPLAIN (voir « \h EXPLAIN ») :

Fournit un **diagnostic sur le traitement d'une requête** (SELECT, INSERT, UPDATE, DELETE,...) → *plan d'exécution*

→ *utile pour l'analyse et l'optimisation des requêtes et des index*

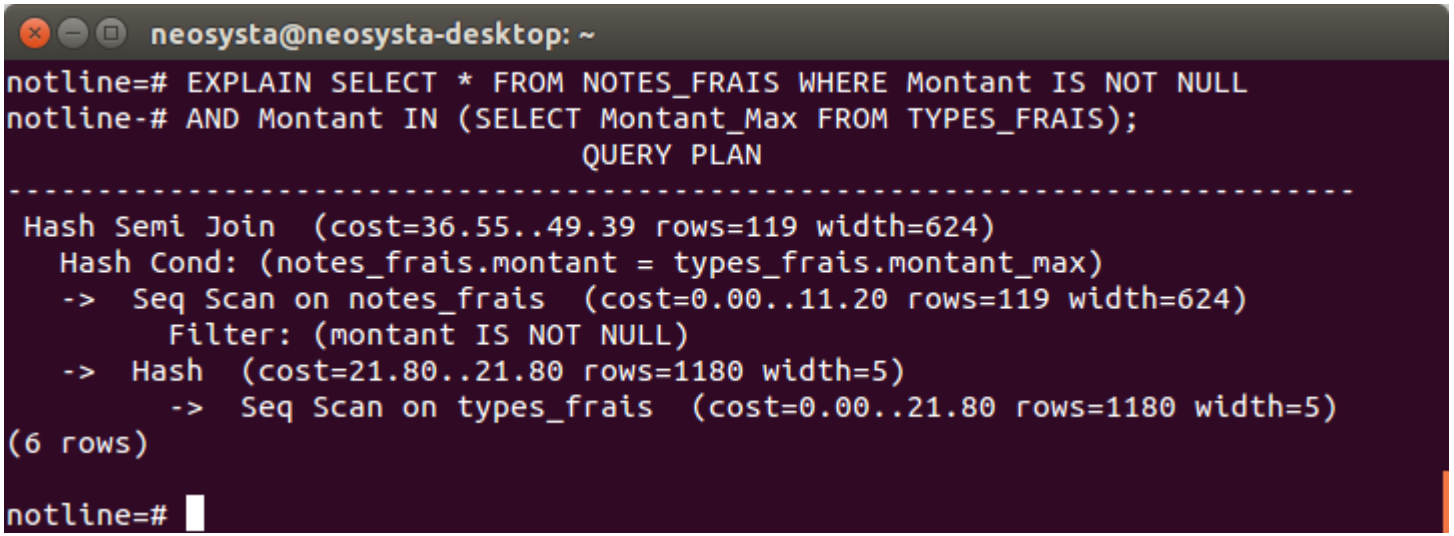
(EXPLAIN sans option ne réalise pas les mises à jour)

Exemple :

```
EXPLAIN SELECT * FROM NOTES_FRAIS WHERE Montant IS NOT NULL
AND Montant IN (SELECT Montant_Max FROM TYPES_FRAIS);
```

Remarque : EXPLAIN n'est pas défini dans le standard SQL

Complément :



La partie la plus importante de l'affichage concerne les coûts estimés d'exécution de la requête (mesurés en une unité de coût arbitraire)

Exemple d'optimisation avec index :

```
CREATE TABLE test (id int);
CREATE FUNCTION test() RETURNS void AS $$ BEGIN FOR n IN 1..50000 LOOP INSERT INTO test VALUES (trunc(random()*50000)); END LOOP; END; $$ LANGUAGE plpgsql;
SELECT test();
SELECT * FROM test ORDER BY id;
EXPLAIN SELECT * FROM test ORDER BY id;
CREATE INDEX ind_test ON test(id); → « \di »
EXPLAIN SELECT * FROM test ORDER BY id;
DROP INDEX ind_test; → « \di »
EXPLAIN SELECT * FROM test ORDER BY id;
DROP FUNCTION test();
DROP TABLE test;
```