- Première Partie -

TABLE DES MATIÈRES

LES CONCEPTS DU MODELE RELATIONNEL

I LE RELATIONNEL

I.1. Naissance du Modèle Relationnel

Historique : précédé par modèle hiérarchique et modèle en réseau (CODASYL).
Inventé en 1970 par **CODD** à partir d'1 théorie mathématique simple (théorie des ensembles)
=> concepts rigoureux et rationnels pour la gestion des données

I.2. Concepts de base

I.2.1. DOMAINE

Définition : 1 domaine est 1 ensemble de valeurs.
exemples :

DOMAINES définis en extension :

NOM=(Jean, Paul, Alice, Michel, Anne, Remi, Sophie)

VILLE=(Paris, Grenoble, Lyon, Londres, Rome)

DOMAINES définis en intention :

ENFANT=(X ∈ NOM / age(X) <= 10)

I.2.2. PRODUIT CARTESIEN

Définition : Soient n domaines D1, D2, ... , Dn
produit cartésien = ensemble des "n_uplets" < Vl, V2, ..., Vn > tels que Vi ∈ Di.

Notation : Dl x D2 x ... Dn

exemple :

NOM x VILLE =

Jean Paris
Jean Grenoble
Jean Lyon
Jean Londres
Jean Rome
Paul Paris

Paul Grenoble
Paul Lyon
Paul Londres
Paul Rome
Alice Paris
...

(7 x 5 = 35 n-uplets)

I.2.3. RELATION

Définition : relation = sous-ensemble du produit cartésien d'1 liste de domaines.

relation = tableau à 2 dimensions (TABLE) :

* colonne du tableau identifiée par CONSTITUANT (issu d'1 domaine auquel on a donné 1 certain sens)

DOMAINE + ROLE JOUE = CONSTITUANT (ou ATTRIBUT)

* En dessous des noms des CONSTITUANTS, chaque ligne $\supset$ n_uplet de la relation.

Relation définie en **EXTENSION** par le tableau

Hypothèses

* Dans 1 relation, jamais 2 n-uplets identiques

* Liste exhaustive des n-Uplets $\in$ relation: tout ce qui n'est pas indiqué dans l'ensemble des n-uplets considéré comme faux.

exemple 1

Domaines : NOM et VILLE
Relation à créer : «EST NE A».
1er constituant : NOMP (NOM de Personne)
2eme : VILLEN (VILLE de Naissance)

EST NE A <-- nom de la relation

NOMP	VILLEN	
		<-- CONSTITUANTS de la relation
Jean	Londres	
Paul	Paris	
Alice	Lyon	
Michel	Grenoble	<-- n_uplets
Anne	Paris	
Remi	Lyon	
Sophie	Lyon	

Remarque

Ordre des colonnes indifférent (le constituant suffit à identifier une colonne)

exemple 2

table précédente + colonne VILLEH (VILLE Habitée actuellement par la personne)
=> relation IDENTITE :

IDENTITE

NOMP	VILLEN	VILLEH
Jean	Londres	Paris
Paul	Paris	Paris
Alice	Lyon	Grenoble
Michel	Grenoble	Grenoble
Anne	Paris	Lyon
Remi	Lyon	Paris
Sophie	Lyon	Lyon

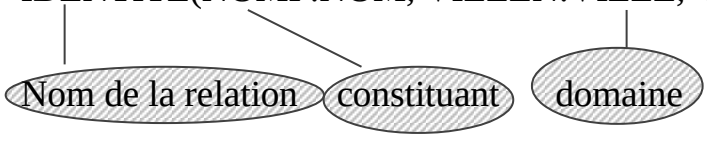
Signification de cette relation :

REGLE

Le n.uplet < n vl v2> ∈ IDENTITE si la personne de nom n est née à v1 et habite v2

Description de IDENTITE :

IDENTITE(NOMP:NOM, VILLEN:VILLE, VILLEH:VILLE)



I.2.4. SCHEMA D'UNE RELATION

tableau + n_uplets => relation définie en **EXTENTION**

schéma => définition d'1 relation en **INTENTION** (n_uplets omis)

Schéma d'1 relation :

* Nom de la relation (R)
* Domaines (D1 ... Dn)
* Constituants ou Attributs (Xi:Di)
* Clé(s) (soulignées)
+ règle (prédicat) indiquant si OUI ou NON 1 n_uplet (a1, a2, ...,an) ∈ R

Notation : R(Xl:Dl, .. Xn:Dn)

Simplification d'écriture : R(Xl, ... Xn).

Comportement des données :

n_uplets dynamiques (créations, modifications, destructions)

mais schéma de la relation fixe

II ALGEBRE RELATIONNELLE

Interrogation des données par opérateurs ensemblistes :

opérateurs / Relations ----> Nouvelles relations

5 opérateurs de base :

1) Produit cartésien (*)
2) Projection ([...])
3) Sélection (:condition logique)
4) Union ($\cup$)
5) Différence (-)

Opérateurs dérivés :

6) Intersection ($\cap$)
7) Division ($\div$)
8) Complément ($\neg$)
9) Jointure externe ( )

exemple :

PRODUIT(NPRO, NOMP, QTS, COULEUR)
produit identifié par son N° NPRO, nom NOMP, quantité QTS en stock et COULEUR

PRODUIT

NPRO	NOMP	QTS	COULEUR
Pl	raquette	200	rouge
P2	ballon	150	bleu
P3	ski	500	noir
P4	planche	70	bleu
P5	voile	50	vert

VENTE(NVEN, NOMC, NPRV, QTV, DATE)

vente identifiée par numéro NVEN, nom du client NOMC, N° du produit vendu NPRV, quantité QTV du produit vendu, et date DATE de la vente.

VENTE

NVEN	NOMC	NPRV	QTV	DATE
1	Dupont	Pl	2	01-04-87
2	Dupont	P3	2	01-10-87
3	Toto	P3	1	10-01-87
4	Toto	P4	1	15-05-87
5	Toto	PS	1	15-05-87
6	Toto	Pl	1	25-09-87

II.1. Produit cartésien

rapproche des tables => apparition de nouvelles informations si colonnes communes.

Notation : R1 * R2

ex : PRODUIT * VENTE

NPRO	NOMP	QTS	COULEUR	NVEN	NOMC	NPRV	QTV	DATE
Pl	raquette	200	rouge	1	Dupont	Pl	2	01-04-87
Pl	raquette	200	rouge	2	Dupont	P3	2	01-10-87
Pl	raquette	200	rouge	3	Toto	P3	1	10-01-87
Pl	raquette	200	rouge	4	Toto	P4	1	15-05-87
Pl	raquette	200	rouge	5	Toto	P5	1	15-05-87
Pl	raquette	200	rouge	6	Toto	Pl	1	25-09-87
P2	ballon	150	bleu	1	Dupont	Pl	2	01-04-87
P2	ballon	150	bleu	2	Dupont	P3	2	01-10-87
P2	ballon	150	bleu	3	Toto	P3	1	10-01-87
P2	ballon	150	bleu	4	Toto	P4	1	15-05-87
P2	ballon	150	bleu	5	Tota	P5	1	15-05-87
P2	ballon	150	bleu	6	Toto	Pl	1	25-09-87
P3	ski	500	noir	1	Dupont	Pl	2	01-04-87
...								

* explosion de la quantité de n_uplets

* table résultante inintéressante .....

sauf pour les n_uplets où 2 colonnes $\supset$ une donnée commune.

exemple :

Pl raquette 200 rouge 1 Dupont Pl 2 01-04-87

--> le produit Pl acheté par le client Dupont en quantité 2 pour la vente 1 est en fait une raquette rouge !

THETA PRODUIT (jointure) :

produit cartésien ---> THETA PRODUIT
intérêt : filtrer n_uplets résultants par 1 contrainte portant sur 1 ou plusieurs constituants communs à R1 et R2

Notation : Rl(X θ Y) * R2

ex : produit cartésien précédent avec condition de juxtaposition NPRO = NPRV

PRODUIT(NPRO = NPRV) * VENTE

NPRO	NOMP	QTS	COULEUR	NVEN	NOMC	NPRV	QTV	DATE
Pl	raquette	200	rouge	1	Dupont	Pl	2	01-04-87
Pl	raquette	200	rouge	6	Toto	Pl	1	25-09-87
P3	ski	500	noir	2	Dupont	P3	2	01-10-87
P3	ski	500	noir	3	Toto	P3	1	10-01-87
P4	planche	70	bleu	4	Toto	P4	1	15-05-87
P5	voile	50	vert	5	Toto	P5	1	15-05-87

=> TETA PRODUIT (**JOINTURE)** de PRODUIT et VENTES sous condition NPRO = NPRV

Généralisation :

Opérateurs de jointure entre 2 relations

* Egalité : Rl(XR1 = XR2) * R2
* Différence : Rl(XRl != XR2) * R2
* Supérieur strictement : Rl(XRl > XR2) * R2
* Supérieur ou égal : Rl(XRl >= XR2) * R2
* Inférieur strictement : Rl(XRl < XR2) * R2
* Inférieur ou égal : Rl(XR1 <= XR2) * R2

Manière d'effectuer 1 THETA PRODUIT entre 2 relations

1) Faire le produit cartésien des 2 relations
2) Ne conserver que les n-uplets répondant à la condition de sélection

II.2. Projection

intérêt : spécifier les colonnes d'1 table à visualiser et leur ordre d'apparition

Notation : R1 [X1, X2, ...Xn]

exemple 1 : projection du THETA PRODUIT précédent sur NPRO, NOMP, NOMC, QTV et DATE

(PRODUIT(NPRO = NPRV) * VENTE)[NPRO, NOMP, NOMC, QTV, DATE]

NPRO	NOMP	NOMC	QTV	DATE
Pl	raquette	Dupont	2	01-04-87
Pl	raquette	Toto	1	25-09-87
P3	ski	Dupont	2	01-10-87
P3	ski	Toto	1	10-01-87
P4	planche	Toto	1	15-05-87
P5	voile	Toto	1	15-05-87

exemple 2 : VENTE[NOMC]

NOMC
Dupont
Toto

II.3. Sélection

intérêt : filtrer n-uplets au moyen d'1 condition portant sur 1 ou plusieurs constituants.

Notation : R1 : *condition*

exemple1 : produits de couleur bleu

PRODUIT : COULEUR='bleu'

NPRO	NOMP	QTS	COULEUR
P2	ballon	150	bleu
P4	planche	70	bleu

exemple2 : ventes réalisées avec client Toto et antérieures au 20-09-87

VENTE : NOMC='Toto' et DATE < 20-09-87

NVEN	NOMC	NPRV	QTV	DATE
3	Toto	P3	1	10-01-87
4	Toto	P4	1	15-05-87
5	Toto	P5	1	15-05-87

II.4. Union

réalisable si les 2 relations ont au moins 1 colonne en commun.
table résultante = n_uplets de R + n_uplets de S non déjà cités.

exemple :

PRODUIT[NPRO] VENTE[NPRV] PRODUIT[NPRO] U VENTE[NPRV]

NPRO	NPRV	NPRO
Pl	P1	P1
P2	P3	P2
P3	P4	P3
P4	P5	P4
P5		P5

remarque : ici l'union n'apporte rien car tout produit vendu $\in$ stocks :

VENTE[NPRV] $\subset$ PRODUIT[NPRO] .

II.5. Différence

réalisable si les 2 relations ont au moins 1 colonne en commun.

table résultante $\supset$ les n-uplets de R sauf ceux $\in$ S

exemple : produits en stock mais non encore vendus:

PRODUIT[NPRO] - VENTE[NPRV]

NPRO

P2

III. CONCEPTION DE SCHEMAS RELATIONNELS

III.1. Modèle ENTITE-RELATION

représentation des entités et des associations --> modèle relationnel.

Modélisation du monde réel => apparitions d'entités (objets, personnes, associations entre ces objets.

exemple :

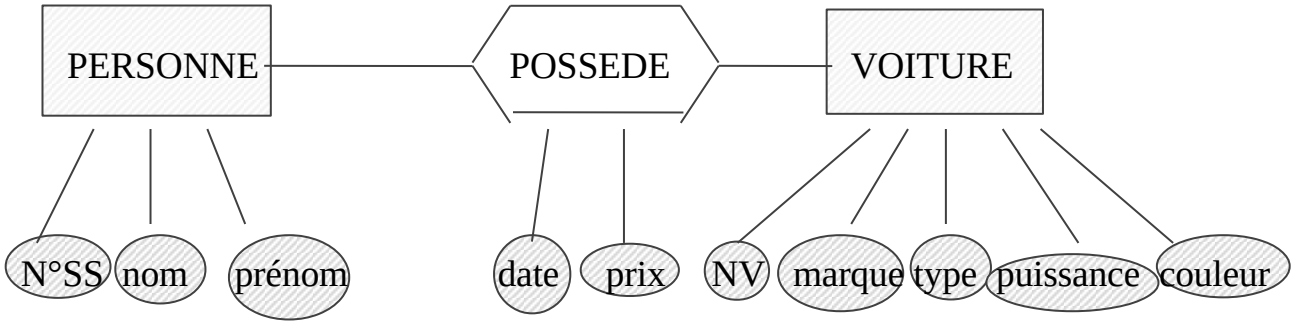
on peut caractériser...

* personne : Numéro de Sécurité Sociale (NSS), nom, prénom.
identifiée de manière unique par son NSS.

* voiture : Numéro minéralogique (NV), marque, type, puissance, couleur.
identifiée de manière unique par son NV.

* association d'1 personne et d'1 voiture : symbolisée par la relation «POSSEDE» depuis la date d'achat et d'1 certain prix.

Modélisation du monde réel par schéma ENTITE-ASSOCIATION :



III.2. Représentation dans le modèle relationnel

entité ≡ relation dont le schéma est le nom de 1'entité + liste des attributs :

ex : PERSONNE(NSS, NOM, PRENOM)

VOITURE(NV,MARQUE, TYPE, PUISSANCE, COULEUR)

association ≡ relation dont le schéma est le nom de 1 'association + liste des identifiants des entités participantes + attributs de l'association

ex : POSSEDE(NSS, NV, DATE, PRIX)

III.3. Problèmes soulevés par une mauvaise perception du réel

Hypothèse :au lieu des 3 relations précédentes, 1 relation «PROPRIETAIRE» :

PROPRIÉTAIRE

NV	MARQUE	TYPE	PUISS.	COUL	NSS	NOM	PRÉNOM	DATE	PRIX
672RH 75	RENAULT	R12TS	6	ROUGE	100	MARTIN	Jacques	10.02.75	10 000
800AB64	PEUGEOT	504	9	VERTE	100	MARTIN	Jacques	11.06.80	50 000
686HK75	CITROEN	2CV	2	BLEUE	200	DUPOND	Pierre	20.04.76	5 000
720CD 60	CITROEN	AMI8	5	BLEUE	200	DUPOND	Pierre	20.08.80	20 000
400XY75	RENAULT	R18B	9	VERTE	300	FANTOMAS	Yves	11.09.81	25 000

Anomalies :

* données redondantes : MARTIN Jacques et DUPOND Pierre apparaissent 2 fois.

  1 personne apparaît autant de fois qu'elle possède de voitures.

  => gaspillage de 1 'espace mémoire

  => risques d'incohérence : si modification de Pierre par Jean, risque d'oublis

* valeurs nulles non autorisées :

  or voitures sans propriétaire ou personnes ne possédant pas de voiture !.

Autre exemple

R(produit, client, adresse, qte)

REGLE = < p c a q > ∈ R si

le client **c** habitant à l'adresse **a** a commandé la quantité **q** du produit **p**

R

PRODUIT	CLIENT	ADRESSE	QTE
lotion	Martin	Paris	10
laque	Martin	Paris	250
crème	Martin	Paris	20
lotion	Jones	Londres	30
crème	Dupont	Lyon	10

Hypothèse: nom du client et adresse unique

Problèmes ? :

1) Modification de l'adresse : si Martin déménage de Paris à Lyon => 3 modifs.
oubli => 2 adresses pour Martin !!

2) Insertion d'1 nouvelle commande <crème Jones Lille 45 >
2 adresses pour Jones !!

3) Insertion d'1 nouveau client n'ayant pas de commande en cours :
< ? Durand Nice ? > !!
Traitement des valeurs nulles PRODUIT et QTE ?

4) Suppression de la commande <crème Dupont Lyon 10>
perte du client Dupont et de son adresse !!

Conclusion :

analyse maladroite des entités et associations

=> relations porteuses d'incohérences + lourdeurs dans la saisie

Bonnes Méthodes de conception : (Merise, Yourdon, SADT, ..)

III.4. Les dépendances fonctionnelles

III.4.1. But - Définition

introduites par CODD en 1970

but : déterminer la décomposition juste d'1 relation.

comment ? : étude des D.F. d'1 relation puis mise en «forme normale»

Définition : soit R(X,Y,Z) où X, Y, Z ensembles de constituants.

On dit que :

X détermine Y

Y dépend fonctionnellement de X

et on le note :

X --> y

si la connaissance d'1 valeur de X détermine AU PLUS 1 valeur de Y.

exemple : VOITURE(NV,MARQUE, TYPE, PUISSANCE, COULEUR)

D.F. :

NV --> COULEUR
TYPE --> MARQUE
TYPE --> PUISSANCE
TYPE, MARQUE --> PUISSANCE

mais :

TYPE -/-> COULEUR

MARQUE -/-> TYPE

III.4.2. Propriétés des DF

(axiomes d'Amstrong)

1) **REFLEXIVITE** : si X $\supset$ Y alors X --> Y

Tout ensemble de constituant détermine lui-même ou 1 partie de lui-même

2) **AUGMENTATION** : si X --> Y alors X, Z --> Y, Z

Si X détermine Y alors les 2 ensembles d'attributs peuvent être enrichis par 1 même 3ème

3) **TRANSITIVITE** : si X --> Y et Y --> Z alors X --> Z

exemple :

NV --> TYPE et TYPE --> PUISSANCE donc NV --> PUISSANCE

Autres règles déduites :

4) **PSEUDO-TRANSITIVITE** : si X --> Y et Y, W --> Z alors X, W --> Z

5) **UNION** : si X --> Y et X --> Z alors X --> Y, Z

6) **DECOMPOSITION** : si X --> Y et Y contient Z alors X --> Z

Dépendance fonctionnelle élémentaire (**DFE**):

Définition : La DF X --> Y est une DFE
si

$\forall$ X' $\subset$ X, X' -/-> Y

exemple :

TYPE, MARQUE --> PUISSANCE non DFE (car TYPE --> PUISSANCE)

CLE d'1 relation :

Définition : X est clé de R(X, Y, Z) si X --> Y, Z DFE

exemple :

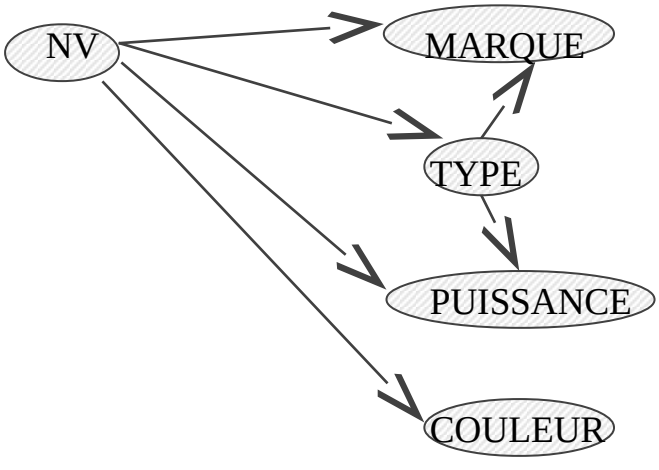
NV --> MARQUE, TYPE, PUISSANCE, COULEUR DFE

=> NV = clé de la relation VOITURE

Notation : VOITURE(NV,MARQUE,TYPE, PUISSANCE, COULEUR)

remarque : relation à plusieurs clés possible

III.4.3. Graphe des D.F.



IV.4.4. Problèmes associés aux DF

1) Fermeture d'1 ensemble de D.F. (= retrouver toutes les DF par application des propriétés)

2) Couverture minimale (∃? 1 ensemble minimal de DF engendrant toutes les DF)

III.5. Formes normales

III.5.1. But - Définition

but de la normalisation : éviter les incohérences.

Point de départ : mise en évidence des DF entre constituants

<u>lere forme normale (1 FN) :</u>

tout constituant contient 1 valeur atomique.

But : améliorer la lisibilité de la table et permettre 1 meilleure performance en machine par réduction de la taille des enregistrements

<u>exemple :</u>

PERSONNE(NOM, AGE, ENFANTS) NON 1 FN

ENFANTS(PRENOM, DATENAISS) 1 FN

		ENFANTS	
NOM	AGE	PRENOM	DATENAISS
Jules	50	Jim	1970
		Joe	1965
		Alice	1963
Jean	45	Joe	1969

NON 1 FN : 1 domaine peut être 1 relation

décomposition en 1 FN :

PERSONNE1(NOM, AGE, PRENOM, DATENAISS)

NOM	AGE	PRENOM	DATENAISS
Jules	50	Jim	1970
Jules	50	Zoe	1965
Jules	50	Alice	1963
Jean	45	Joe	1969

2ème forme normale (2 FN)

Définition :

1 FN + Tout constituant ∉ la clé ne doit pas dépendre d'1 sous-ensemble de la clé

exemple 1 :

R(PRODUIT, CLIENT, ADRESSE, QTE) NON 2 FN

car :
PRODUIT, CLIENT --> ADRESSE, QTE et CLIENT --> ADRESSE

décomposition en 2 FN :

R1(CLIENT, ADRESSE) 2 FN
R2(PRODUIT, CLIENT, QTE) 2 FN

Inconvénients évités par le passage en 2 FN :

***** impossible d'entrer les valeurs < client adresse > tant que le client n'avait pas achète 1 pièce.

***** Si erreur sur produit acheté par client, annuler l'enregistrement => perte des informations du client (nom et adresse).

***** Si modifier nom ou adresse du client : le faire pour toutes les occurrences du client dans la relation (100 produits commandés par client => modifier 100 fois adresse ou nom du client)

Autre formulation : relation en 2 FN si tous les constituants non clés dépendent PLEINEMENT des clés.

exemple 2 :

VOITURE(NV, MARQUE, TYPE, PUISSANCE, COULEUR) 2 FN

car :
NV --> MARQUE, TYPE, PUISSANCE, COULEUR clé simple

3ème forme normale (3FN)

2 FN + Tout constituant ∉ une clé ne dépend pas d'1 constituant non clé

exemple

VOITURE(NV, MARQUE, TYPE, PUISSANCE, COULEUR) NON 3 FN

car :

NV --> MARQUE, TYPE, PUISSANCE, COULEUR

mais :

TYPE --> PUISSANCE, MARQUE

décomposition en 3 FN :

VOITURE1(NV, TYPE, COULEUR) 3 FN

VOITURE2(TYPE, PUISSANCE, MARQUE) 3 FN

Inconvénients évités par le passage en 3 FN :

***** impossible de créer 1 type de voiture avec puissance et marque
tant que voiture ∉ 1 client

***** si 1 type de voiture possédée par 1 seul client :

si client ne possède plus cette voiture et enregistrement annulé =>

perte des valeurs des attributs liées à cette voiture (type, marque et puissance).

***** si puissance fiscale d'1 type de véhicule modifiée :

répercuter la modif. sur toutes les voitures existantes de ce type

(100 voitures de ce type => répéter modif. 100 fois !).

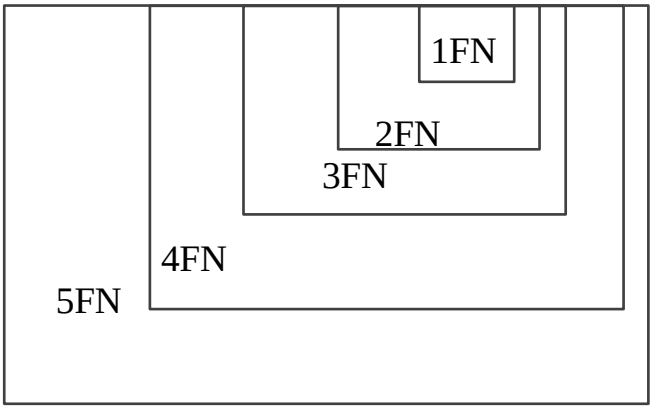
REMARQUES :

* Mise en 3 FN des relations suffisante pour éliminer redondances et anomalies de MAJ.

* existence 4 FN et 5 FN

* Structure des FN :

passage à la FN > si niveaux < valides :



PRESENTATION DE POSTGRESQL

1. HISTORIQUE

1970 : article "A Relational Model of Data for Large Data Banks" (Codd)

1977 : création d'Oracle Corporation

1977 : création d'Ingres (RDBMS)
Université de Californie à Berkeley

1986 : création de Postgres (ORDBMS)
Université de Californie à Berkeley
acheté par Illustra et commercialisé par "Informix"

1991 : version 7 d'Oracle

1994 : Postgres1995
Postgres + capacités récentes d'SQL,
par Jolly CHEN et Andrew YU

1996 : Open Source SQL Database (janvier)
4 développeurs initiaux + milliers de développeurs sur Internet(Mailing list)

1996 : PostgreSQL(décembre)
Sortie du produit

1997 : Distribution Répandue de PostgreSQL
Red Hat

2003 : version 10 d'Oracle
BD orientée objet, BD internet, norme SQL99, interfaces de prog.,
langage de procédures, portable sur diverses plateformes, transactionnel

2013 : version 9.3 de PostgreSQL
presque idem Oracle et gratuit !!!

2. PRINCIPALES CARACTERISTIQUES de PostgreSQL

- SGBD relationnel
- Gestion intégrée de l'ensemble de données d'1 entreprise accessibles aux utilisateurs et applications
- Sécurité, cohérence et intégrité
- Portabilité sur grande variété de plates-formes matérielles et systèmes d'exploitation (architecture ouverte)
- Outils utilisables dans toutes les étapes d'1 projet d'informatisation

3. POSTGRESQL : UN SGBD RELATIONNEL

PostgreSQL possède les fonctionnalités classiques d'un SGBD relationnel :

3.1. LA DEFINITION ET LA MANIPULATION DE DONNEES

LDD : Langage de Définition de Données

LMD : Langage de Manipulation de Données (op. rel. : X, σ , $\cup$, $\cap$, -)
interactif, ou dans 1 pgme d'application (L3G ou L4G).

PLIpgSQL : extension procédurale du SQL (pour traitements complexes).

Fonctions stockées **et** ***trigger*** : pour traitements répétitifs
(appelées par pgmes ut. ou par système, après réalisation d'1 événement)

3.2. LA COHERENCE DES DONNEES

*D*éfinition, validation et annulation de transactions par PostgreSQL.

Transaction = ensemble d'opérations de MAJ de la BD constituant 1 unité logique de traitements

Sous-transaction : évite d'annuler la totalité de la transaction

3.3. LA CONFIDENTIALITE DES DONNEES

privilèges : attribués à 1 ut. pour effectuer 1 opération sur 1 objet qui $\notin$ ut.

Droit éventuel d'attribuer ce même privilège par l'ut. à d'autres ut.

ex : privilèges de connexion à la base, créations et manipulations des objets (tables, vues, ...).

Privilège particulier : administration de la BD (créer des uts, administrer la BD, sauvegarde, restauration, gérer l'espace physique, ...).

groupes : possèdent un ensemble de privilèges en commun

vues : restreignent l'accès aux données en donnant 1 visibilité partielle à certains uts

ex :pour permettre à 1 ut de n'accéder qu'à qques colonnes ou lignes d'1 table, on définit 1 vue sur cette table et on n'autorise à cet ut. que l'accès à travers cette vue.

3.4. L'INTEGRITE DES DONNEES

Conception d'1 BD = définition de la structure des données (attributs, tables) et des *contraintes d'intégrités* par le LDD.

ex : intégrité de valeur, intégrité référentielle, intégrité de domaine, etc..

3.5. LA SAUVEGARDE ET LA RESTAURATION DES DONNEES

Techniques de **reprise à chaud et à froid** pour remettre la BD dans 1 état cohérent suite à 1 panne matérielle ou logicielle.

Journaux de reprise tenus à jour par PostgreSQL en cas de panne.

3.6. LA GESTION DES ACCES CONCURRENTS

Accès simultané aux mêmes données à plusieurs uts par techniques de verrouillage.

Verrouillage : accès interdit à 1 partie des données pendant son utilisation par 1 unité de traitement.

Données protégées contre des opérations de M.A.J. incorrectes ou affectant les structures

Détection d'interblocages et déblocage :

Si 2 uts se trouvent dans 1 état où chacun attend la libération d'1 partie de données en cours d'utilisation par l'autre,

=> avortement de l'1 des 2 travaux, selon certaines règles.

4. PostgreSQL : UN SGBD A ARCHITECTURE OUVERTE

4.1. LA PORTABILITÉ DANS PostgreSQL

PostgreSQL écrit en langage C disponible sur :

plusieurs plates-formes matérielles (Sparc, PC, PS, Macintosh,...)

plusieurs systèmes d'exploitation (Unix, Linux, HP/UX, Mac OS, SunOSn Windows).

=> Portage d'1 application et des données PostgreSQL d'1 machine vers 1 autre sans modifs majeures.

=> Environnement de développement peut être ≠ de celui de l'exploitation.

4.2. LA COMPATIBILITE AUX NORMES

PostgreSQL membre des organismes internationaux de normalisation (ANSI, ISO, AFNOR, SQL Access Group, X/OPEN, etc)..

SQL PostgreSQL compatible aux normes SQL86, SQL89, SQL92 et à celle d'autres SGBD (SQL/DS, DB d'IBM..)

=> permet d'utiliser avec 1 minimum de modifs des applications développées autour d'autres SGBD.

Une application développée avec 1 outil utilisant l'interface ODBC (Open Database Connectivity) de Microsoft peut accéder aux données PostgreSQL sans modifs.

5. PostgreSQL : UNE SOLUTION COMPLETE ET INTEGREE

Outils de développement d'applications :

Interface	Langage	Type	Avantages
LIBPQ	C	Compilé	Interface native
LIBPGEASY	C	Compilé	C simplifié
ECPG	C	Compilé	ANSI encastré SQL C
LIBPQ++	C++	Compilé	Orienté objet C
ODCB	ODBC	Compilé	Connectivité application
PERL	Perl	Interprété	Traitement texte
PGTCLSH	Tcl/TK	Interprété	Interface graphique
PHP	HTML	Interprété	Dynamique page Web
PYTHON	Python	Interprété	Orienté objet
JDBC	Java	Les deux	Portabilité

Outils de Bureautique :

PgAccess : interface graphique pour néophytes ressemblant à Access

Outils d'administration et d'aide à l'exploitation :

Permettent à l'administrateur et à certains types d'utilisateurs de :

- maintenir la cohérence des données.
- assurer les opérations de sauvegarde et de restauration des données.
- effectuer le chargement des données à partir de fichiers ext. à PostgreSQL.

6. CARACTERISTIQUES SUPPLEMENTAIRES

6.1. FONCTIONNELLES

6.1.1. SUPPORT COMPLET DES CONTRAINTES D'INTEGRITES

PostgreSQL permet de définir les contraintes d'intégrité :

- Caractère obligatoire/facultatif
- Unicité des lignes
- Clé primaire
- Intégrité référentielle (clé étrangère)
- Contrainte de valeurs (intervalle ou liste de valeurs).

6.1.2. SUPPORT DES FONCTIONS STOCKEES

Nouveaux objets définis et stockés dans la BD puis partagés par les uts. comme les objets classiques (tables, vues, ...) :

fonctions stockées *:* ensemble de commandes PL/pgSQL pour

- minimiser les transferts réseau dans le cas d'1 architecture client/serveur
- améliorer les performances (compilation des commandes SQL ou PL/pgSQL en 1 seule fois)

6.1.3. SUPPORT DES TRIGGERS

trigger (ou déclencheur) : procédure stockée dans la base et associée à un événement pouvant intervenir sur 1 table.

S'exécute quand 1 commande SQL spécifiée de MAJ (insertion, suppression ou modification) affecte la table associée au déclencheur.

6.1.4. SUPPORT DES DONNEES NON STRUCTUREES

Les LOB (Large Objects) : images, vidéos, sons, textes, etc..

6.1.5. UTILISATION // DE PLUSIEURS FICHIERS DE REPRISE

Mêmes informations écrites simultanément sur les fichiers de reprise.

=> En cas de perte de l'1 de ces fichiers, utilisation d'autres fichiers.

6.1.6. TYPES DE PRIVILEGES

Privilèges de création, suppression ou modification de tables, de procédures ou d'ut.
Deux catégories de privilèges *:*

privilèges globaux : droit de création, modification et suppression de BDs
droit de création, modification et suppression d'utilisateurs

privilèges objet : droit de création, modification et suppression d'objets
droit de création, modification et suppression de données

6.1.7. EXTENSION DU SUPPORT DE LANGUE NATIONALE

Spécification du format de la date, du symbole monétaire, du séparateur des milliers (espace ou point) et le caractère décimal (',' ou '.')

6.1.8. ARCHITECTURE À SERVEUR PARALLELE

Démarrer des instances en // (fichiers de données, de reprise et de contrôle partagés entre ≠ instances).

6.2 PERFORMANCES

6.2.1 Architecture basée sur la notion de serveur partagé (multi-threaded)

Partage des processus serveurs entre ≠ processus uts.
Un nombre optimal de processus serveur répond efficacement aux demandes de processus uts.

6.2.2 Optimiseur

Optimisation basée sur :

- syntaxe des requêtes
- statistiques du comportement des tables .

6.2.3 Méthode d'accès basée sur le hashage :

Clusters hashés (hash clusters) : accès aux données stockées dans des clusters.

=> accès + rapide qu'avec 1 index. Pour tables stables (peu de MAJ)

6.2.4 Répartition de l'espace en cas de suppression des lignes d'une table

Commande (TRUNCATE) pour la suppression rapide de toutes les lignes d'une table

6.3 ADMINISTRATION

6.3.1. Simplification de la gestion des privilèges à l'aide des groupes :

groupe = ensemble d'utilisateurs ayant les mêmes privilèges.

6.3.2. Gestion des segments d'annulation (rollback segments)

Adaptation de leur taille, activation (online) ou désactivation (offline).
Spécification d'1 segment d'annulation pour 1 transaction donnée.

ARCHITECTURE FONCTIONNELLE DE PostgreSQL

1. COMPOSANTS PostgreSQL

- **couches de base :**

  noyau : fcts de base d'1 SGBD
  dictionnaire de données : gérer l'ensemble des objets
  couche SQL : accès aux données
  couche PL/SQL : extension procédurale du langage SQL.

- **outils de développement d'applications :**

  développement d'applications construites autour du SGBD.

- **outils d'administration**

2. COUCHES DE BASE

2.1. LE NOYAU

Communication avec la BD.

Fonctions :

- ***Intégrité et cohérence des données, confidentialité des données, sauvegarde & restauration des données, gestion des accès concurrents***

- ***Optimisation de l'exécution des requêtes :***

Requête soumise au noyau analysée, optimisée (optimiseur), exécutée.

- ***Gestion des accélérateurs :***

index : accès direct aux lignes d'1 table.

cluster : regroupe les lignes de 2 tables ayant la même valeur sur 1 clé commune (tables à colonnes communes et/ou souvent accédées ensemble).

hash cluster : accès + rapide que index (pour tables ± stables).

- ***Stockage physique des données :***

*R*eprésentation & stockage des données ds fichiers, géré par noyau
=> indépendance vis-à-vis du SE.

2.2. LE DICTIONNAIRE DE DONNEES

Métabase décrit d'1 façon dynamique la BD :

- objets de la base (tables, colonnes, vues, index, synonymes, clusters, séquences,..)
- uts accédant à PostgreSQL avec leurs privilèges et droits sur les ≠ objets
- infos relatives à l'activité de la BD (connexions, ressources utilisées, verrouillages)

2.3. LA COUCHE SQL

Interface entre noyau et outils PostgreSQL pour :

- interpréter les commandes SQL,
- vérifier leur syntaxique et sémantique,
- les décomposer en opérations élémentaires,
- les soumettre au noyau pour exécution.

=> résultat transmis à l'application ou l'outil ayant soumis la commande.

2 catégories de commandes SQL :

- langage de définition de données (**LDD**) : création, modification, suppression des structures de données (tables, vues, index, ...).
- langage de manipulation de données (**LMD**) : consultation, insertion, modification, suppression des données.

2.4. LA COUCHE PL/pgSQL

Extension procédurale du langage SQL.

Utiliser possibilités des L3G et L4G :

- structures de contrôle (traitements conditionnels et itérations)
- utilisation de variables
- traitements d'erreurs.

Unités de traitement de PL/pgSQL = blocs

Blocs utilisables à partir de tous les outils PostgreSQL.

3. OUTILS DE DEVELOPPEMENT D'APPLICATIONS

3.1. psql

Interface interactive permettant :

- utilisation interactive de SQL et PL/pgSQL (lancées à partir de psql).
- paramétrage de l'environnement de travail : longueur d'1 ligne, nbre de lignes/page,
- formatage des résultats : pour afficher en HTML, en LaTeX, définir un titre, etc ...
- mémorisation des commandes SQL, PL/pgSQL et psql dans des fichiers de commandes

3.2. INTERFACES DE PROGRAMMATION

Accès aux données PostgreSQL depuis un langage de prog. (C, C++, java, HTML, ...).

=> Commandes SQL insérées dans pgme écrit en langage hôte traduites par précompilo.

Exemple d'Interfaces de programmation : LIBPQ, LIBPGEASY, ECPG, LIBPQ++, ODCB, PERL, PGTCLSH, PHP, PYTHON , JDBC

4. OUTILS DE BUREAUTIQUE

4.1. pgAccess (pour uts néophytes)

- Accès aux fonctionnalités de PostgreSQL à travers 1 interface graphique conviviale :

  utilisation interactive de la BD et transfert des données vers d'autres logiciels (tableur ou traitement de texte).

- Création, modif. ou suppression de :

  tables, requêtes, vues, séquences, fonctions, états, formulaires, utilisateurs, schémas, bases de données

- Insertion, modif. ou consultation des données sans connaissance du SQL.

- Construction d'applications simples à base de fenêtres, avec menus éventuels pour intégrer les composantes d'1 application.

- Mode query-by-example (QBE) utilisé (requête SQL correspondante générée automatiquement)

- Aide en ligne (guide l'ut pour opérations).

5. OUTILS D'ADMINISTRATION

5.1. Administration

Outils pour l'adm. de BD PostgreSQL : **pg_ctl, initdb, createdb, dropdb, vacuumdb**, ...
Pour :

- démarrage et arrêt d'1 instance (pg_ctl),
- chargement et déchargement d'1 groupe de BD (initdb),
- creation et suppression d'1 BD (createdb, dropdb),
- pilotage en temps réel du fonctionnement de PostgreSQL (pg_ctl),
- sauvegarde et restauration des données et des journaux (pg_ctl),
- maintenance physique (système) et analytique (performances) d'1 BD (vacuumdb)

5.2. Transfert de données entre une table et un de fichier

Outils de base pour le transfert :

copy : pour alimenter une table d'1 BD PostgreSQL par des données provenant d'un fichier externe à PostgreSQL de type texte ou binaire.

5.3. Sauvegarde/restauration des données

Outils pour effectuer sauvegarde et restauration totale ou partielle d'1 BD PostgreSQL :

pg_dump : créer 1 copie d'1 partie ou de la totalité des objets d'1 BD (tables, ut., droits, index,...), compressés ou non.

pg_restore : intégrer ds 1 BD des objets exportés de la même base ou d'1 autre BD PostgreSQL.

LES OBJETS MANIPULES DANS PostgreSQL

Stockés dans le dictionnaire de données .

1. LES OBJETS CREES

database : base de données

function : unités de traitements composées de commandes SQL et/ou PL/SQL, stockées dans le dictionnaire de données sous forme compilée.

groupe : ensemble d'utilisateurs ayant les mêmes privilèges attribués à ce groupe.

index : structure contenant l'@ physique de chaque ligne d'1 table.
=> Accès direct à l'info.

language : language choisi pour écrire les fonctions de la BD courante

operator : opérateur créé dans la BD

séquence *:* générateur de séquences de nombres uniques (pour les identifiants)

trigger (déclencheur) : traitements définis et déclenchés lorsqu'un événement se réalise (logique événementielle). Défini par :

- table sur laquelle il s'applique,
- événement qui le déclenche (m.a.j., insertion ou suppression),
- traitement à effectuer (défini à l'aide de procédures),
- moment où ce trmt sera exécuté (< ou > événement déclencheur).

table : structure de données maintenant les données d'1 BD Représente 1 entité du monde réel ou 1 relation entre 2 entités. Composée de :

- *colonne* = 1 caractéristique de l'entité (ex : N° SS client ou son nom).
- *ligne* = 1 occurrence de l'entité ou de la relation.

type : nouveau type dans la BD

user : utilisateur de la BD

vue : représentation logique issue de la combinaison de la définition d'1 ou de plusieurs tables ou vue. Utilisée pour :

- assurer la sécurité de données,
- masquer la complexité des données,
- réduire la complexité de la syntaxe des requêtes,
- représenter les données pour 1 autre perspective.

2. LES CONTRAINTES D'INTEGRITE

Clé primaire :

- Composée d'1 ou de plusieurs colonnes de la table et utilisée pour identifier chaque ligne d'une manière unique.
- Ne doit pas contenir de colonnes à valeurs nulles.

Clé unique :

- Possède les mêmes propriétés qu'1 **clé** primaire sauf qu'1 colonne définie comme clé unique doit avoir 1 valeur distinct pour chaque ligne de la table.
- Peut contenir des valeurs nulles.

Clé étrangère :

- Représente 1 relation entre les tables.
- Composée d'1 ou de plusieurs colonnes dans 1 table dite fille dont le valeurs dépendent d'1 clé primaire ou unique d'1 table appelée parent.

Intégrité référentielle *:*

- Les relations représentée par les clés primaires et étrangères sont maintenues.
- Assure la consistance de données.

3. CLASSIFICATION DES COMMANDES SQL

3 familles de commandes dans SQL :

Commandes de définition de données (*langage de définition de données* ou LDD) :

Décrire les objets modélisant l'univers de discours en créant des schémas et les objets qui les composent (tables, synonymes, séquences, etc.).

Commandes de contrôle de données :

Maintenir confidentialité et intégrité des données contre usages malveillants.

Commandes de manipulation de données (*langage de manipulation de données* ou LMD):

Extraire et mettre à jour les données d'1 base par des opérateurs relationnels et ensemblistes.

BASE DE DONNEES EXEMPLE : Gestair

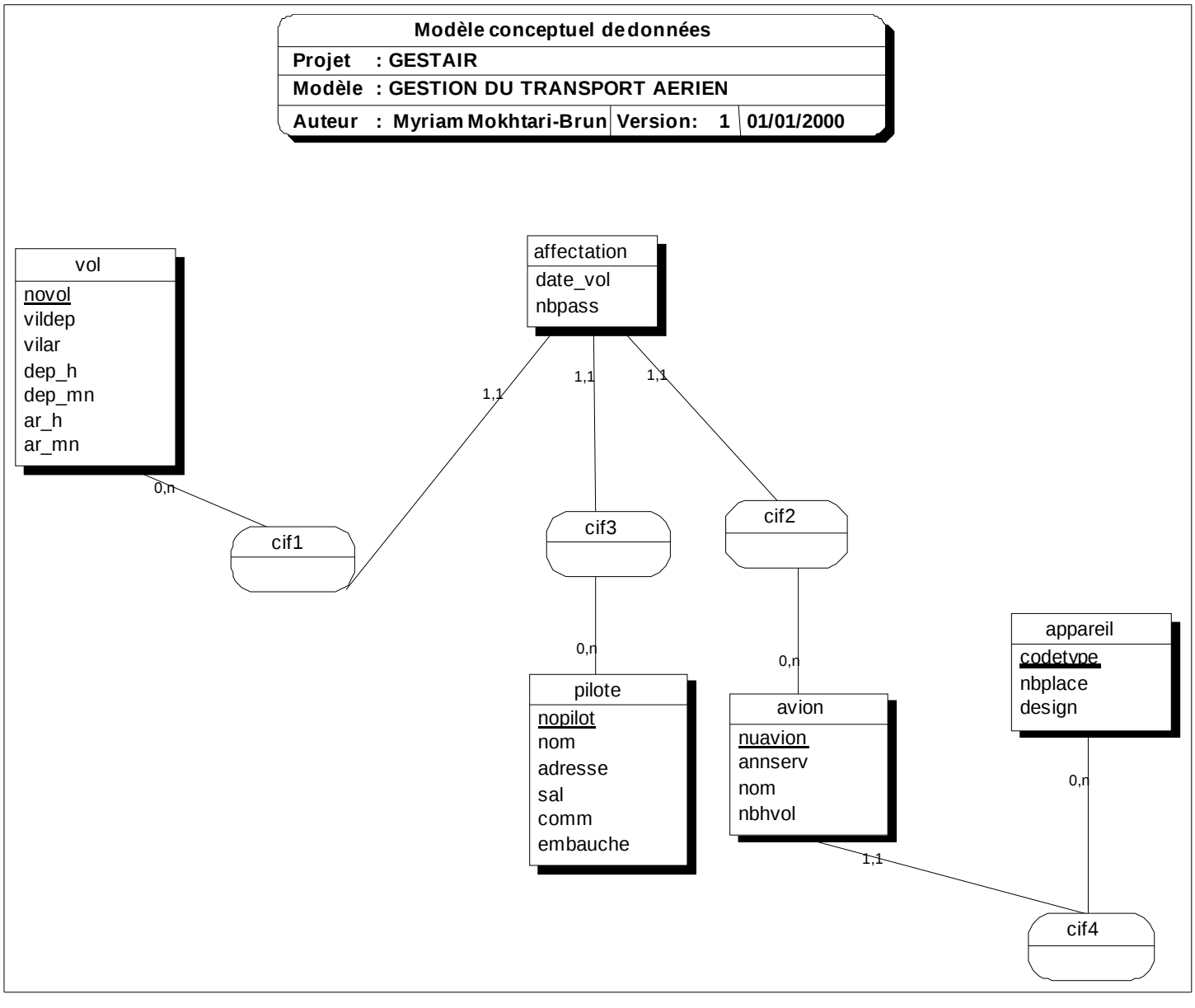
Gestair : BD contient infos pour la gestion (très simplifiée) du transport aérien.

- 1 vol pour un trajet donné effectué selon plusieurs dates, ou jamais desservi.
- Vol effectué => affectation d'1 pilote et 1 avion.

- 1 pilote ou 1 avion peut ne jamais avoir été affecté à 1 vol ; comme ils peuvent l'avoir été pour plusieurs.

- 1 avion doit appartenir à une catégorie d'appareils répertoriée.
- Il peut y avoir des appareils enregistrés pour lesquels il n'existe aucun avion.

1. MODELE CONCEPTUEL DE DONNEES (MCD)



2. DESCRIPTION DES TABLES

VOL

novol	CHAR(6)	n° identification d'un vol
vildep	VARCHAR(30)	ville de départ
vilar	VARCHAR(30)	ville d'arrivée
dep_h	NUMBER(2)	heure de départ (heure)
dep_mn	NUMBER(2)	heure de départ (minute)
ar_h	NUMBER(2)	heure d'arrivée (heure)
ar_mn	NUMBER(2)	heure d'arrivée (minute)

Clé primaire : novol

APPAREIL

codetype	CHAR(3)	code d'1 famille d'avions
nbplace	NUMBER(3)	nbre de places
design	VARCHAR(50)	nom de la famille d'avions

Clé primaire : codetype

AVION

nuavion	CHAR(4)	n°immatriculation d'1 avion
type	CHAR(3)	code d'1 famille d'avions
annserv	NUMBER(4)	année de mise en service
nom	VARCHAR(50)	nom avion non obligatoire
nbhvol	NUMBER(8)	nbre d'heures vol depuis sa mise en service

Clé primaire : nuavion
Clé étrangère : type, référence appareil.codetype

PILOTE

nopilot	CHAR(4)	n° matricule du pilote
nom	VARCHAR(35)	nom du pilote
adresse	VARCHAR(30)	adresse (ville) du pilote
sal	NUMBER(8, 2)	salaire mensuel
comm	NUMBER(8, 2)	commission éventuelle
embauche	DATE	date d'embauche

Clé primaire : nopilot

AFFECTATION

vol	CHAR(6)	n° identification d'un vol
date_vol	DATE	date vol (jj.mm.aa)
nbpass	NUMBER(3)	nbre effectif de passagers
pilote	CHAR(4)	n° du pilote conduisant l'avion pour le vol
avion	CHAR(4)	n° immatriculation de l'avion affecté au vol

Clé primaire : vol + date_vol
Clé étrangère : vol, référence vol.novol
Clé étrangère : pilote, référence pilote.nopilot
Clé étrangère : avion, référence avion.nuavion

3. CONTENU DES TABLES

gestair=# SELECT * FROM vol;

NOVOL	VILDEP	VILAR	DEP_H	DEP_MN	AR_H	AR_MN
AF8810	PARIS	DJERBA	9	0	11	45
AF8809	DJERBA	PARIS	12	45	15	40
IW201	LYON	FORT DE FRANCE	9	45	15	25
IW655	LA HAVANNE	PARIS	19	55	12	35
IW433	PARIS	ST-MARTIN	17	0	8	20
IW924	SIDNEY	COLOMBO	17	25	22	30
IT319	BORDEAUX	NICE	10	35	11	45
AF3218	MARSEILLE	FRANCFORT	16	45	19	10
AF3530	LYON	LONDRES	8	0	8	40
AF3538	LYON	LONDRES	18	35	19	15
AF3570	MARSEILLE	LONDRES	9	35	10	20

gestair=# SELECT * FROM appareil;

CODETYPE	NBPLACE	DESIGN
74E	150	BOEING 747-400 COMBI
AB3	180	AIRBUS A300
741	100	BOEING 747-100
SSC	80	CONCORDE
734	450	BOEING 737-400

```
gestair=# SELECT * FROM avion;
NUAVION TYPE    ANNSERV NOM                  NBHVOL
------- ---- ---------- -------------------- ----------
8832    734        1988 VILLE DE PARIS            16000
8567    734        1988 VILLE DE REIMS             8000
8467    734        1995 LE SUD                      600
7693    741        1988 PACIFIQUE                 34000
8556    AB3        1989                           12000
8432    AB3        1991 MALTE                     10600
8118    74E        1992                           11800

gestair=# SELECT * FROM pilote;
NOPILOT NOM          ADRESSE    SAL    COMM   EMBAUCHE
------- ---------- --------   ------ -----  ---------
1333    FEDOI       NICE       24000   1780  01-MAR-92
6589    DUVAL       PARIS      18600   5580  12-MAR-92
7100    MARTIN      LYON       15600  16000  01-APR-93
3452    ANDRE       REIMS      22670         12-DEC-92
3421    BERGER      REIMS      18700         28-DEC-92
6548    BARRE       LYON       22680   8600  01-DEC-92
1243    COLLET      PARIS      19000      0  01-FEB-90
5643    DELORME     PARIS      21850   9850  01-FEB-92
6723    MARTIN      ORSAY      23150         15-MAY-92
8843    GAUCHER     CACHAN     17600         20-NOV-92
3465    PIC         TOURS      18650         15-JUL-93

gestair=# SELECT * FROM affectation;
VOL    DATE_VOL      NBPASS PILOTE AVION
------ --------- ---------- ------ -----
IW201  01-MAR-94        310 6723   8567
IW201  02-MAR-94        265 6723   8832
AF3218 12-JUN-94         83 6723   7693
AF3530 12-NOV-94        178 6723   8432
AF3530 13-NOV-94        156 6723   8432
AF3538 21-DEC-94        110 6723   8118
IW201  03-MAR-94        356 1333   8567
IW201  12-MAR-94        211 6589   8467
AF8810 02-MAR-94        160 7100   8556
IT319  02-MAR-94        105 3452   8432
IW433  22-MAR-94        178 3421   8556
IW655  23-MAR-94        118 6548   8118
IW655  20-DEC-94        402 1243   8467
IW655  18-JAN-94        398 5643   8467
IW924  30-APR-94        412 8843   8832
IW201  01-MAY-94        156 6548   8432
AF8810 02-MAY-94         88 6723   7693
AF3218 01-SEP-94         98 8843   7693
AF3570 12-SEP-94         56 1243   7693
```

ENVIRONNEMENT DE TRAVAIL (psql)

1. DEMARRER ET QUITTER psql

- **Se connecter :**

$> psql -h nom_serveur -U nom_utilisateur nom_BDD
exemple : psql -h postgres -U mbrun gestair

gestair=#

- **Quitter :**

gestair=# \q

2. ENTRER ET EXECUTER DES COMMANDES

- **gestair=#**SELECT nopilot FROM pilote ;

Exécute la commande SQL

```
NOPILOT
------------
8843
3465
…
```

- **gestair=#** *-- ligne*

Indique que la ligne est un commentaire

Exemple :

gestair=# -- Requete qui affiche le numero des pilotes
gestair=#SELECT nopilot
gestair-# FROM
gestair-# pilote;

2.1. Commandes de formatage

- **gestair=#\pset title** [texte]

Place un titre en haut de chaque résultat de requête, ou le supprime si texte pas précisé.

- **gestair=#\pset fieldsep** [car]

Précise le séparateur de colonnes de chaque résultat de requête si les tuples ne sont pas alignés (par défaut '|').

- **gestair=#\pset format** [unaligned | HTML | LATEX]

Précise le format de sortie de chaque résultat de requête (par défaut aligned), ou le supprime si rien n'est précisé.

Exemple :

gestair=#\pset title 'Liste des pilotes'
gestair=#\pset fieldsep '*'
gestair=#\pset format unaligned
gestair=#select nom, sal, embauche FROM pilote WHERE nopilot = '3465' or nopilot='8843';

```
Liste des pilotes
nom*sal*embauche
PIC*8650*15-JUL-93
GAUCHER*17600*20-NOV-92
```

- **gestair=#\pset recordsep** [car]

Précise le séparateur de lignes de chaque résultat de requête si les tuples ne sont pas alignés (par défaut '\n').

Exemple :

gestair=#\pset recordsep '!'
gestair=#select nom, sal, embauche FROM pilote WHERE nopilot = '3465' or nopilot='8843';

```
Liste des pilotes
nom*sal*embauche!PIC*8650*15-JUL-93!GAUCHER*17600*20-NOV-92
```

- **gestair=# \pset tuples_only**

Bascule entre l'affichage du nom des colonnes et pas d'affichage.

Exemple :

gestair=# \pset tuples_only
gestair=#select nom, sal, embauche FROM pilote WHERE nopilot = '3465' or nopilot='8843';

```
PIC*8650*15-JUL-93!GAUCHER*17600*20-NOV-92
```

- **gestair=#\pset border** [*valeur*]

Précise l'épaisseur du contour des tables en mode HTML.

- **gestair=#\pset expanded**

Bascule entre les formats classique et étendu (1ère colonne : nom des colonnes, 2ème colonne : veleurs). Pratique si lignes trop grandes.

- **gestair=#\pset null [***valeur***]**

Remplace l'affichage des valeurs NULL par *valeur*.

- **gestair=#\pset tableattr [***'attr=valeur attr=valeur .....'***]**

Définit un attribut HTML qui sera placé dans la balise <table> si mode HTML (ex: 'width=100%cellspacing=10').

- **gestair=#\pset pager**

Active ou désactive l'utilisation d'une paginateur pour l'affichage des résultats.

2.2. Commandes d'affichage d'informations sur la BD et ses objets

- **gestair=#\d** *[table]*

Affiche la définition de la table spécifiée (colonnes, types, indexes, contraintes, ...), ou de toutes les tables.

Exemple :

gestair=#\d *pilote*

```
 Colonne                        Type         Modifications
 ------------------------------ ------------------------
 NOPILOT                        CHAR(4)      NOT NULL
 NOM                            VARYING(35)  NOT NULL
 ADRESSE                        VARYING(30)  NOT NULL
 SAL                            NUMERIC(8,2)NOT NULL
 COMM                           NUMERIC(8,2)
 EMBAUCHE                       DATE         NOT NULL

Index : pk_nopilot primary key btree (nopilot)
Check constraints : .......
.........
```

- **gestair=#\da** *[nom_agrégat]*

Affiche la définition de l'agrégat spécifié ou de tous les agrégats.

Exemple :

gestair=#\da *avg* *(moyenne)*

```
Liste des fonctions d'agrégation
Schema       | Nom    |   Types de données |   Description
-----------------------------------------------------------
pg_catalog   | avg    |   bigint           |
pg_catalog   | avg    |   double precision |
pg_catalog   | avg    |   integer          |
...
```

- **gestair=#\dd** *[nom]*

Affiche la définition de l'objet spécifié (table, fonction, opérateur, index, ...) ou de tous les objets.

- **gestair=#\df** *[nom_fonction]*

Affiche la définition de la fonction spécifiée ou de toutes les fonctions.

- **gestair=#\d[istvSlopT]** *[nom]*

affiche la définition de :
l'index si **i**
la séquence si **s**
la table si **t**
la vue si **v**
la table système si **S**
Le grand objet si **l**
l'opérateur si **o**
la permission d'accès si **p**
Le type de donnéesa si **T**

si *nom* est spécifié, sinon de tous.

2.3. Commandes d'information sur psql et PostgreSQL

- **gestair=# \?**

Donne l'aide sur les commandes \.

- **gestair=# \h** [*commande*]

Donne l'aide sur la commande SQL *commande*

Exemple :

gestair=# \h select

2.4. Commandes d'entrées/sorties

- **gestair=# *p***

Liste la commande SQL stockée dans le buffer SQL

Exemple :

gestair=# \p
```
SELECT nopilot FROM pilote
```

- **gestair=#\g**

Exécute la commande SQL stockée dans le buffer SQL

Exemple :

gestair=# \g
```
NOPILOT
------------
8843
3465
```
…

- **gestair=#\i** nomfichier.sql

Exécute le fichier de commande *nom_fichier.sql.*

Exemple :

Si le fichier *requete1.sql* contient '*SELECT nopilot FROM pilote*',

gestair=#\i r*equete1.sql*

affichera les n° de pilotes.

- **gestair=#\w** *nom_fichier.sql*

Sauvegarde le contenu du buffer dans un fichier *nom_fichier.sql*

Exemple :

gestair=#\w toto.sql

Sauvegarde la requête précédente dans le fichier *toto.sql*

- **gestair=#\echo** [*texte*]

Affiche le message *texte*

Exemple :

gestair=#\echo Requete client
```
Requete client
```

- **gestair=# \o** [*nom_fichier[.ext]]*

Redirige les sorties futures (c-à-d les données obtenues après l'exécution de cette commande) vers le fichier *nom_fichier*. Sans paramètre, la sortie est redirigées vers l'écran.

2.5. Commandes sytème

- **gestair=# \!** *commande_unix*

Exécute la commande unix sans quitter psql.

Exemple pour lister le répertoire courant :

gestair=#\ ! ls

- **gestair=#\edit**[*nom_fichier[.ext]]*

Edite le fichier de commande *nom_fichier.ext.* Pour éditer le contenu du buffer, omettre le nom. L'éditeur par défaut est *vi*.

2.6. Substitution des variables

- **gestair=#\set** [*variable* [*valeur*]]

Définit une variable utilisateur et lui associe une valeur; ou liste les valeurs des variables définies si rien n'est précisé. Si *valeur* omis, la variable est remise à vide.

Exemple :

gestair=#\set numpilot 1345

Si une requête contient le paramètre numpilot, la valeur définie est utilisée et non demandée.

Exemple :

gestair=#SELECT nom FROM pilote WHERE nopilot = ':numpilot';
Requête exécutée avec la valeur définie de *numpilot*, sinon affichage du message suivant pour toute occurrence de &numpilot dans la requête:

- **gestair=#\unset** *variable …*

Supprime la définition d'une ou de plusieurs variables.

Exemple :

gestair=#\unset nopilot

La variable est vide.

2.7. Invite de psql

- **gestair=#\set PROMPT1** | **PROMPT2** | **PROMPT3** *chaîne*

Modifie l'invite de psql où :

PROMPT1 : invite normale.
PROMPT2 : invite à chaque nouvelle ligne d'une instruction ou requête non terminée.
PROMPT3 : invite à la saisie de données pendant la commande COPY.

Exemple :

gestair=#\set PROMPT2 'psql:%/%R(%n)# '

```
psql:gestair-(myriam)#
```

gestair=#\set PROMPT2 '\n[%`date`]\n%n:%/=# '

```
[Mer sep 6 16:09:17 CEST 2004]
myriam:gestair=#
```

Voir le manuel de références pour la signification des caractères de substitution (%/, %n, R, %`commande`, ...).

3. OPTIONS GENERALES DE LA COMMANDE PSQL

Syntaxe complète : psql [*options*] [*base_de_donnees*] [*nom_utilisateur*]

où :

base_de_donnees : bd à laquelle *nom_utilisateur* veut se connecter

nom_utilisateur : compte de l'utilisateur sous lequel on veut se connecter.

Si les 2 omis, connexion à une bd et sous un compte de nom= utilisateur système.

options : *-lettre* ou -- *commande*

-a, --echo-all
Donne un écho des lignes saisies. Equiv. à *set ECHO* dans psql.

-A, --no-align
Format de sortie non aligné. Par défaut, aligné.

-e, --echo-queries
Donne un écho des requêtes saisies.

-E, --echo-hidden
N'affiche pas les requêtes saisies. Equiv. à *set ECHO HIDDEN* dans psql.

-f *nom_fichier*, --file *nom_fichier*
psql lit et exécute les instructions SQL dans le fichier *nom_fichier,* puis se termine.

-F *séparateur*, --field-separator *séparateur*
Le délimiteur de champs (colonnes) des résultats de requêtes sera *séparateur*.

-H, --html
Affiche les résultats au format HTML.

-l, --list
Affiche la liste des BD auxquelles l'on peut se connecter.

-o, *nom_fichier*, --output *nom_fichier*
Redirige la sortie dans le fichier *nom_fichier.*

-P *nom=valeur*, --pset *nom=valeur*
Précise les options de formatage des sorties (voir commande *pset format*).

-q , --quiet
Mode silencieux. Aucun texte d'information affiché.

-R *séparateur*, --record-separator *séparateur*
Le délimiteur de lignes des résultats de requêtes sera *séparateur*.

-s, --single-step
Mode pas à pas. Confirmer ou annuler chaque exécution d'une commande SQL.

-S, --single-ligne
Mode ligne par ligne. Retour à la ligne = exécution d'une commande SQL.

-t, --tuples-only
Nom des colonnes et nombre de lignes renvoyées d'une requête pas affichés.

-T *attribut_table*, --table-attr *attribut_table*
Définit un attribut HTML qui sera placé dans la balise <table> si mode HTML (ex: width=100%). Si plusieurs attributs, les placer entre 2 apostrophes (ex : -T 'width=90% cellspacing=10'). Voir *pset tableattr*.

-U *nom_utilisateur*, --username *nom_utilisateur*
Connexion sous le compte utilisateur *nom_utilisateur*.

-W, --password
Demande un mot de passe pour se connecter.

-?, --help
Affiche une aide rapide sur les paramètres de la commande psql.

Langage de Manipulation de Données (LMD)

1 Types de données

Principaux types :

1.1 Type Caractère - char

chaîne de caractères de longueur fixe.
char(longueur) où 1 ≤ longueur ≤ 2000 (par défaut 1)

1.2 Type Caractère – varchar

chaîne de caractères de longueur variable.

varchar(longueur) où 1 ≤ longueur ≤ 4000 (par défaut 1)

1.3 Type booléen – boolean

true (vrai) ou false (faux)

1.4 Type entier court – smallint, int2

entiers signés sur 2 octets.

1.5 Type entier – integer, int4

entiers signés sur 4 octets.

1.6 Type Long – bigint, int8

entiers signés sur 8 octets.

1.7 Type réel – real, float4

nombres à virgule flottante sur 4 octets.

1.8 Type réel précis – double precision, float8

nombres à virgule flottante sur 8 octets.

1.9 Type Numérique – numeric

nombres numériques exactes, de $1.0*10^{-130}$ à $9*10^{125}$

numeric [(précision [,échelle])]

où précision : nbre chiffres significatifs, de 1 à 38 (38 par défaut)
échelle : nbre chiffres après la virgule, de -84 à +127.

Exemples :

salaire numeric(8,2) : 8 chiffres significatifs dont 2 après la virgule
salaire numeric(8,-2) : résultat arrondi à la centaine

1.10 Type Date et heure – date, time, timestamp, interval

date : Date du calendrier (jour, mois, année) sur 4 octets.

time : Heure (heure, minute, seconde, micro-secondes) dans une journée sur 4 octets.

timestamp : Date et Heure sur 8 octets.

interval : Délai quelconque sur 12 octets. Syntaxe : qté1 unité1 [, qté2 unité2 ...][ago]

où qté : entier pour
unité : *minute, hour, day, week, month, year, decade, century, millenium*
ago : si omis, intervalle positif sinon intervalle négatif.

1.11 Type identifcateur – oid

identificateur d'objet dans la base PostgreSQL. Jusqu'à 4 Go. Utile pour le traitement d'objets volumineux (LOB) comme les fichiers au format texte ou image.

2 Constantes

2.1 Constante numérique

nombre contenant 1 signe, point décimal et / ou exposant puissance de 10.

Exemples :

-10 (entière) 2.5 (décimale) 1.2 E-23 (flottante)

2.2 Constante alphabétique

chaîne de caractères entre apostrophes, où majuscule ≠ minuscule

Exemples :

'MARTIN' 'Titre de l''Application'

2.3 Constante date

chaîne de caractères entre apostrophes au format :

Format général	Exemple
ISO	2004-06-25 12:24:00-00
SQL	06/25/2004 12:24:00.00 PDT
Postgres	Mon 25 Jun 12:24:00 2004 PDT

Pour préciser le format :

Gestair# SET DATESTYLE TO SQL;

Pour choisir aussi l'ordre entre le jour et le mois :

Gestair# SET DATESTYLE TO SQL, EUROPEAN; -- 25/06/2004

Gestair# SET DATESTYLE TO SQL, US; -- 06/25/2004

2.4 Constante time

chaîne de caractères entre apostrophes.

Exemples :
01:24 01:24 AM 01:24:PM 13:24 01:24:11.112

2.5 Constante timestamp

constante **date** + **' '** + constante **time**

2.6 Constante interval

chaîne de caractères entre apostrophes.

Exemples:

SELECT date '1980-06-25' + interval '21 years 8 days' as 'Date resultat';

=>
Date resultat
2001-07-03 00:00:00

SELECT date '1980-06-25' + interval '21 years 8 days ago' as 'Date resultat';
=>
Date resultat
1959-06-17 00:00:00

3 Opérateurs de base

opérateurs du modèle relationnel :

- Projection
- Sélection
- Jointure
- Division

opérateurs de la théorie des ensembles :

- Union
- Intersection
- Différence
- Produit cartésien

3.1 Opérateur de Projection

3.1.1 Définition

but : afficher le contenu d'une table en spécifiant les attributs souhaités.

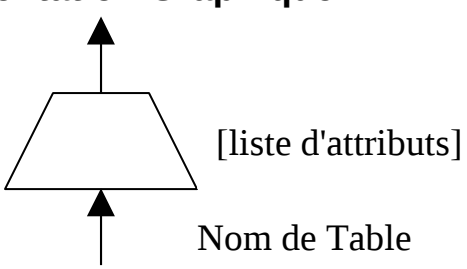
Rem : cardinalité de la relation conservée.

Exemple : AVION(numéro, annserv, nom, nbhvol, type) de la base Gestair.

Projection sur numéro et nom **=>**

Numéro	**Nom**
8832	Ville de Paris
8467	Le Sud
7693	Pacifique
8432	Malte
8567	Ville de Reims

3.1.2 Représentation Graphique



3.1.3 Traduction dans le langage SQL

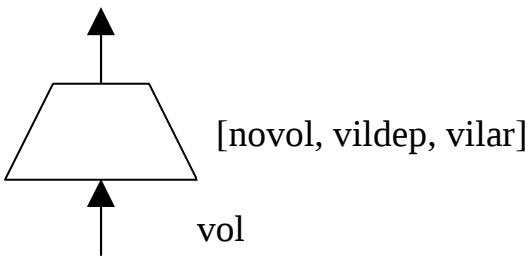
SELECT [**DISTINCT**] liste_attributs **FROM** nom_de_table;

Où :

- Attribut= nom_table.nom_colonne ou nom_colonne ou expression
- Liste_attribut = attribut, attribut, … ou *****
- DISTINCT élimine les lignes identiques

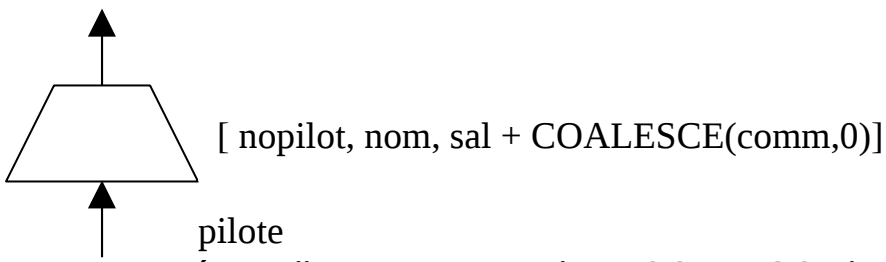
Exemples :

- Liste des vols enregistrés dans la BD, avec affichage du n° de vol, de la ville de départ et de la ville d'arrivée.



SELECT novol as "Numéro de vol", vildep as "Ville de départ", vilar as "Ville d'arrivée" FROM vol;

- Liste des pilotes et de leur revenu (= salaire + commission)



SELECT nopilot as "Numéro Pilote", nom, sal + COALESCE(comm,0) as "Revenu" FROM Pilote;
rem : COALESCE retourne la commission si non NULL, sinon retourne 0.

3.1.4 Tri du résultat d'une requête

ORDER BY nom_col1 | n°_col1 | expression [**DESC**] [**,** nom_col2 | n°_col2 | expression [**DESC**]

Tri selon 1ère col., puis pour 1 même valeur de 1ère col. selon la 2ème col., …

DESC : tri décroissant (par défaut croissant)

Exemple :

```
SELECT nopilot as "Numéro Pilote", nom, sal + NVL(comm,0) as "Revenu"
FROM Pilote
ORDER BY 3 DESC, nom;
```

=> affichage des pilotes par revenu décroissant et pour un même revenu affichage selon l'ordre alphabétique des noms.

3.1.5 LIMIT

limite le nombre de lignes résultat de l'exécution d'une requête.

Exemple :

```
SELECT nopilot as "Numéro Pilote", nom
FROM Pilote
LIMIT 10 ;
```

=> affichage des 10 premiers pilotes.

3.1.6 Expression CASE :

```
CASE WHEN condition1 THEN resultat1
       WHEN condition2 THEN resultat1
       [...]
       [ELSE resultat_defaut]
END [AS alias]
```

```
Ex : SELECT nopilot,
            CASE WHEN sal > 10000 THEN 'plus de 10 000'
                 WHEN sal > 5000 THEN 'compris entre 5000 et 1000'
                 ELSE 'moins de 5000'
            END AS 'Message'
     FROM pilote;
```

3.2 Opérateur de Sélection

3.2.1 Définition

but : afficher les lignes d'une table vérifiant une condition.

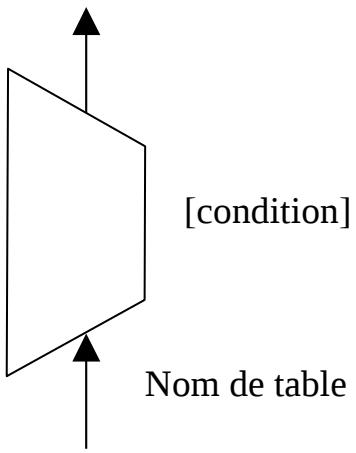
Rem : degré de la relation conservée.

Exemple : AVION(numéro, annserv, nom, nbhvol, type) de la base Gestair.

Sélection des avions ayant effectué plus de 10000 heures de vol **=>**

Numéro	**Annserv**	**Nom**	**Nbhvol**	**Type**
8832	1988	Ville de Paris	16000	734
7693	1988	Pacifique	34000	741
8432	1991	Malte	10600	AB3

3.2.2 Représentation Graphique



3.2.3 Traduction dans le langage SQL

SELECT *
FROM nom_de_table
WHERE prédicat ;

où prédicat = condition de sélection (restriction)

3.2.3.1. Prédicat simple

- <expr1> **opérateur** <expr2>
  où opérateur ∈ { **=**, **!=** , **<>** , **>** , **>=** , **<** , **<=** }

- <expr1> **BETWEEN** <expr2> **AND** <expr3>
- <expr1> **IN** (expr2, expr3, …exprn)
- <expr1> **LIKE** <chaîne>
  où chaîne ⊃ caractères génériques de substitution (_, %)
- <expr1> **IS [NOT] NULL**
  vrai (Faux) si l'expression a la valeur NULL (non définie)

3.2.3.2 Prédicat composés

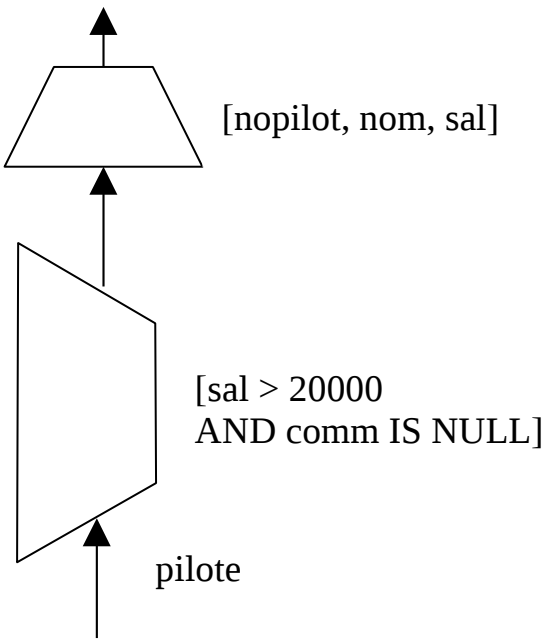
plusieurs prédicats simples reliés par opérateurs logiques **AND**, **OR**, ou **NOT**.

NOT inverse le sens du prédicat

AND prioritaire / OR

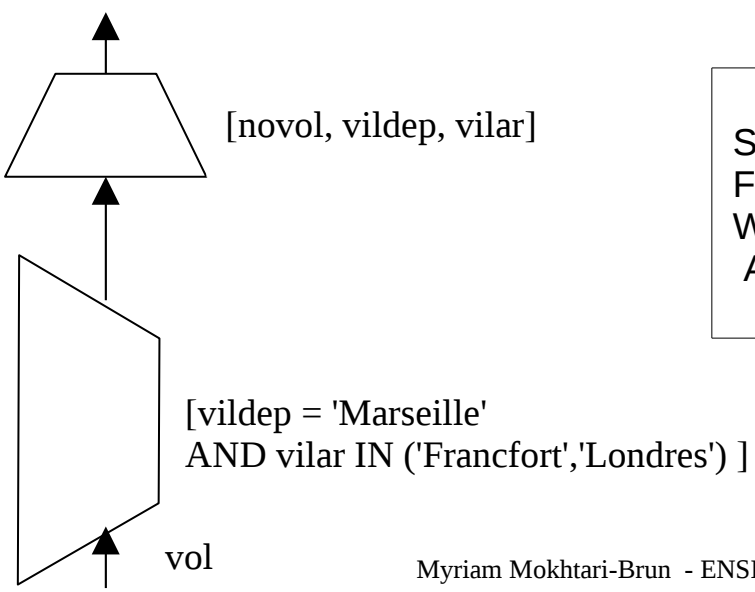
Exemples :

- Liste alphabétique des pilotes de salaire > 20000 et sans commission




```
SELECT nopilot, nom, sal
FROM pilote
WHERE sal > 20000
 AND comm IS NULL
ORDER BY 2 ;
```

- Liste des vols qui relient Marseille à Francfort ou à Londres




```
SELECT novol, vildep, vilar
FROM vol
WHERE vildep ='Marseille'
 AND vilar IN ('Francfort','Londres') ;
```

3.3 Opérateur Produit Cartésien

3.3.1 Définition

Table T crée à partir de 2 tables T1 et T2 où :
chaque ligne de la 1ère table est concaténée par chaque ligne de la 2ème.
Degré de T = degré de T1 + degré de T2
Cardinalité de T = cardinalité de T1 * cardinalité de T2.

Exemple :

AVION (numéro, annserv, nom, nbhvol, type)

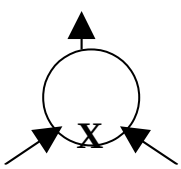
APPAREIL (code, nbplace, design)

Produit cartésien de AVION par APPAREIL

=>T (numéro, annserv, nom, nbhvol, type, code, nbplace, design)

composée de tous les avions par tous les appareils.

3.3.2 Représentation Graphique

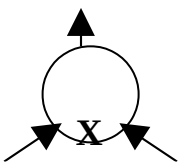


Nom de table Nom de table

3.3.3 Traduction dans le langage SQL

SELECT *
FROM Nom_table1, Nom_table2 ;

Exemple :



avion appareil

3.4 Opérateur de Jointure

But : mettre en relation 2 ou plusieurs tables

Jointure = produit cartésien + sélection (pivot de jointure)

3.4.1 Equijointure

3.4.1.1 Définition

égalité entre clé primaire de T1 et clé étrangère de T2

Exemple :

AVION (**numéro**, annserv, nom, nbhvol, type)

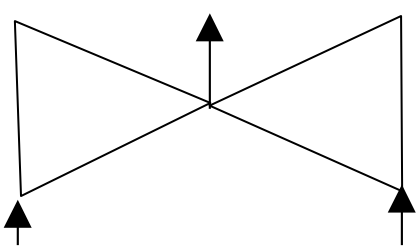
APPAREIL (**code**, nbplace, design)

Jointure par le pivot : code = type

=>T (numéro, annserv, nom, nbhvol, type, code, nbplace, design)

composée des avions de chaque appareil

3.4.1.2 Représentation Graphique



[pivot]

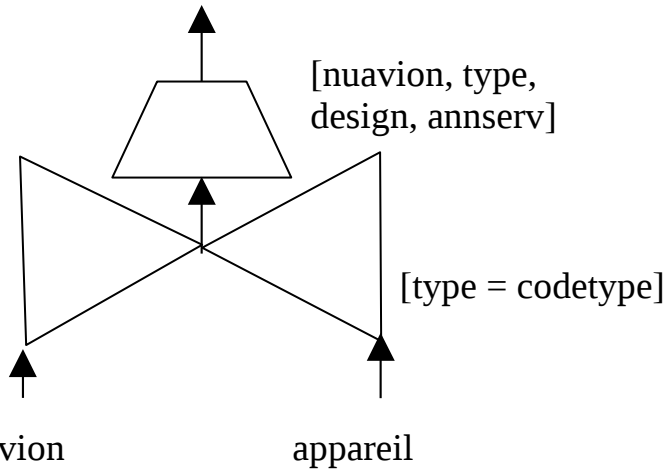
Nom de table Nom de table

3.4.1.3 Traduction dans le langage SQL

```
SELECT *
FROM Nom_table1, Nom_table2
WHERE [table1.]colonne =  [table2.].colonne;
```

Exemple :

Liste des avions avec n° d'avion, type, désignation et année de mise en service :



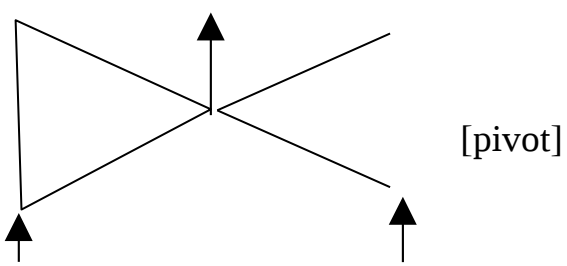

```
SELECT nuavion, type, design, annserv
FROM avion, appareil
WHERE codetype = type ;
```

3.4.2 Jointure externe

3.4.2.1 Définition

T1 : table dominante T2 : table subordonnée (éléments manquants)
jointure / lignes de T1 affichées même si condition jointure non réalisée.

3.4.2.2 Représentation Graphique



où pivot = [Nom de table.]colonne = [Nom de table.]colonne **(+)**

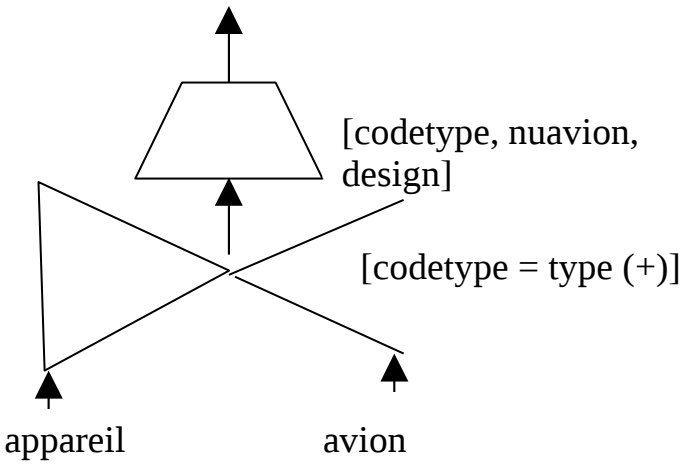
3.4.2.3 Traduction dans le langage SQL

FROM table1 LEFT JOIN table2 ON ([table1.]colonne = [table2.].colonne)

Exemple :

Liste de tous les types d'avions avec les n° d'avions correspondants.

Possibilité : appareils enregistrés sans avion !




```
SELECT codetype, nuavion, design
FROM appareil as a1 LEFT JOIN
avion as a2 ON a1.codetype = a2.type;
```

3.4.3 Thétajointure

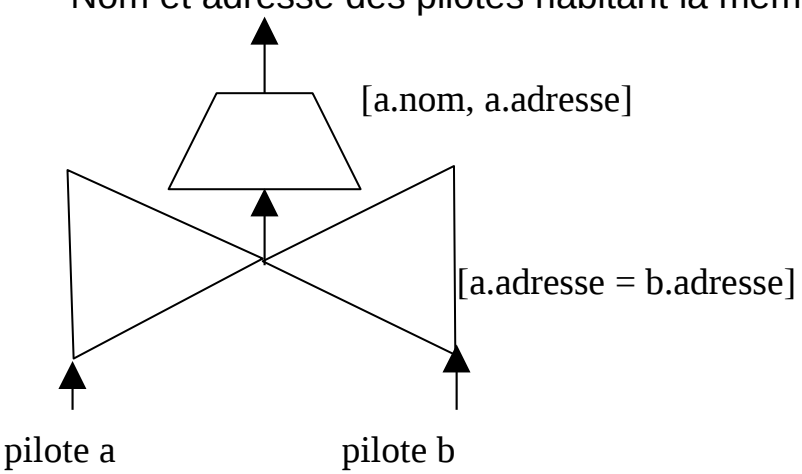
jointure où opérateur du pivot ∈ {<, <=, >, >=, !=, <>}

3.4.4 Autojointure

jointure d'une table à elle-même.
=> utiliser des alias

Exemple :

Nom et adresse des pilotes habitant la même ville (tri par ordre alphabétique).



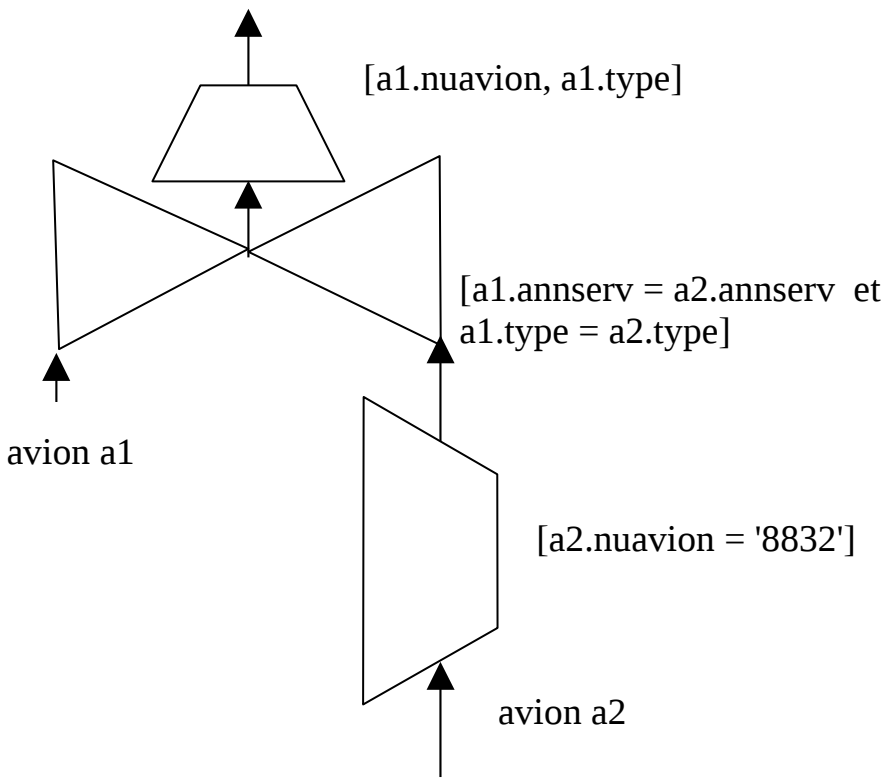

```
SELECT a.nom, a.adresse
FROM pilote as a, pilote as b
WHERE a.adresse = b.adresse
ORDER BY 1,2 ;
```

3.4.5 Jointure et sélection simultanées

SELECT ….
FROM Nom_table1, Nom_table2
WHERE *pivot*
 AND *conditions de sélection* ;

Exemple :

Type et n° des avions du même type que l'avion 8832 et mis en service la même année




```
SELECT   a1.nuavion, a1.type
FROM     avion as a1, avion as a2
WHERE    a1.annserv = a2.annserv
 AND             a1.type = a2.type
 AND             a2.nuavion = '8832' ;
```

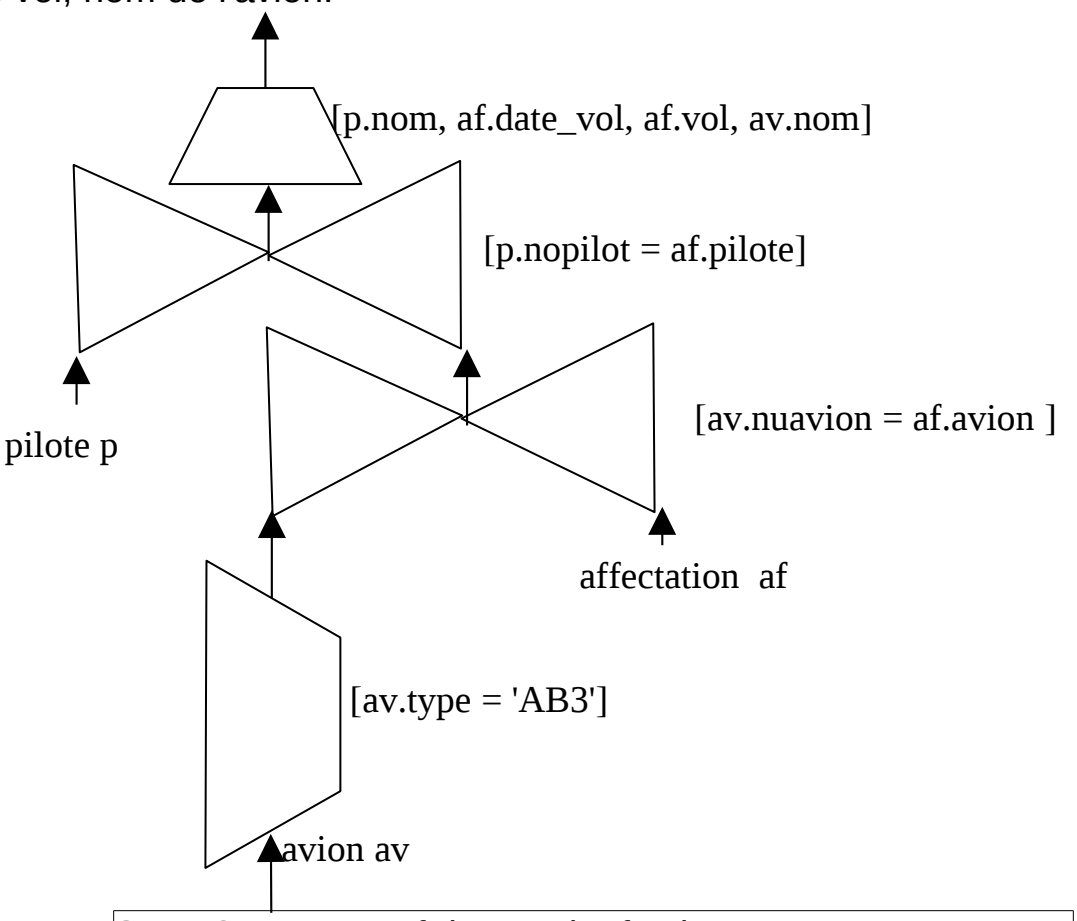
3.4.6 Jointures multiples

jointure de plus de 2 tables

SELECT …
FROM Nom_table1, Nom_table2, …, Nom_tablen
WHERE pivot1 AND pivot2 …..
AND *conditions de selection* ;

Exemple :

Pilotes conduisant des avions de code type AB3. Lister le nom, date de vol, n° de vol, nom de l'avion.



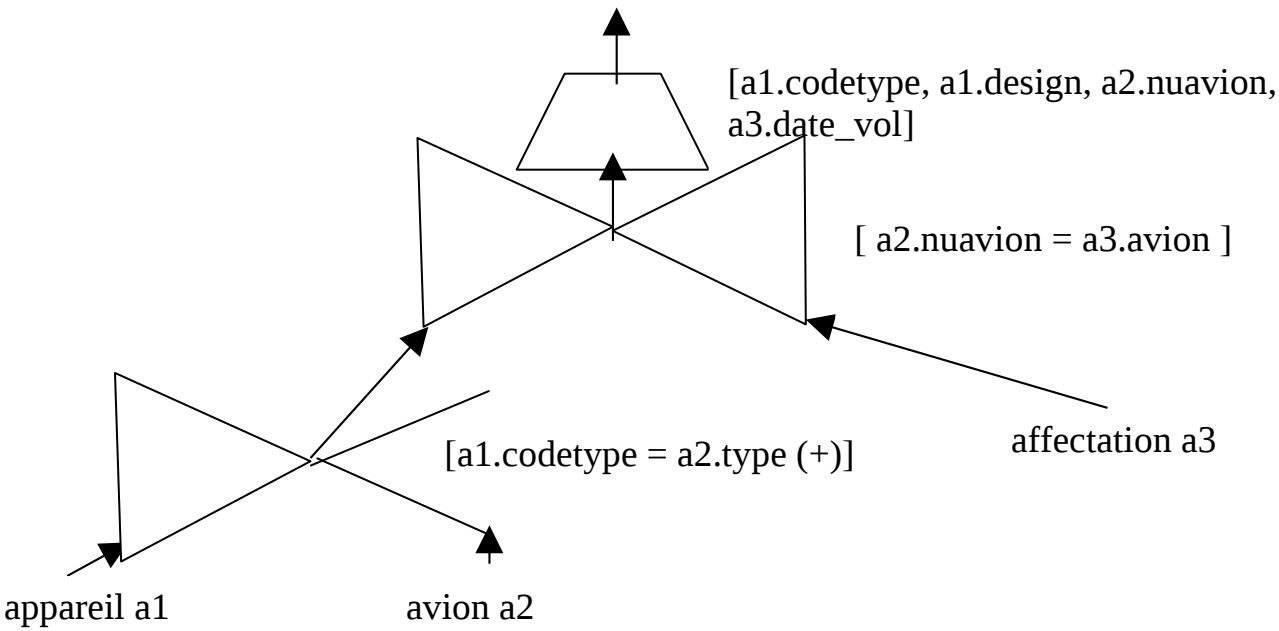

```
SELECT p.nom, af.date_vol, af.vol, av.nom
 FROM avion as av, affectation as af, pilote as p
 WHERE av.nuavion = af.avion
  AND p.nopilot =  af.pilote
  AND av.type = 'AB3' ;
```

3.4.7 Jointures multiples dont une jointure externe

Exemple :

Liste de tous les types d'avions avec les n° d'avions correspondants ayant déjà été affectés.

Possibilité : appareils enregistrés sans avion !




```
SELECT codetype, nuavion, design
FROM appareil as a1 LEFT JOIN avion as a2 ON a1.codetype = a2.type
INNER JOIN affectation as a3 ON a2.nuavion= a3.avion ;
```

3.5 Opérateurs Ensemblistes

3.5.1 Union

3.5.1.1 Définition

ensemble des lignes de 2 tables

conditions : même nombre d'attributs
même type pour attributs de même rang

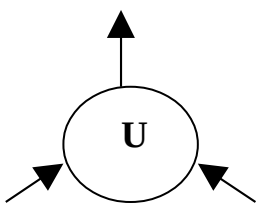
Exemple :

ANC_APPAREIL(code, nbplace, design)

NOUV_APPAREIL(ncode, nnbplace, ndesign)

union **=>**T(code, nbplace, design)

3.5.1.2 Représentation Graphique



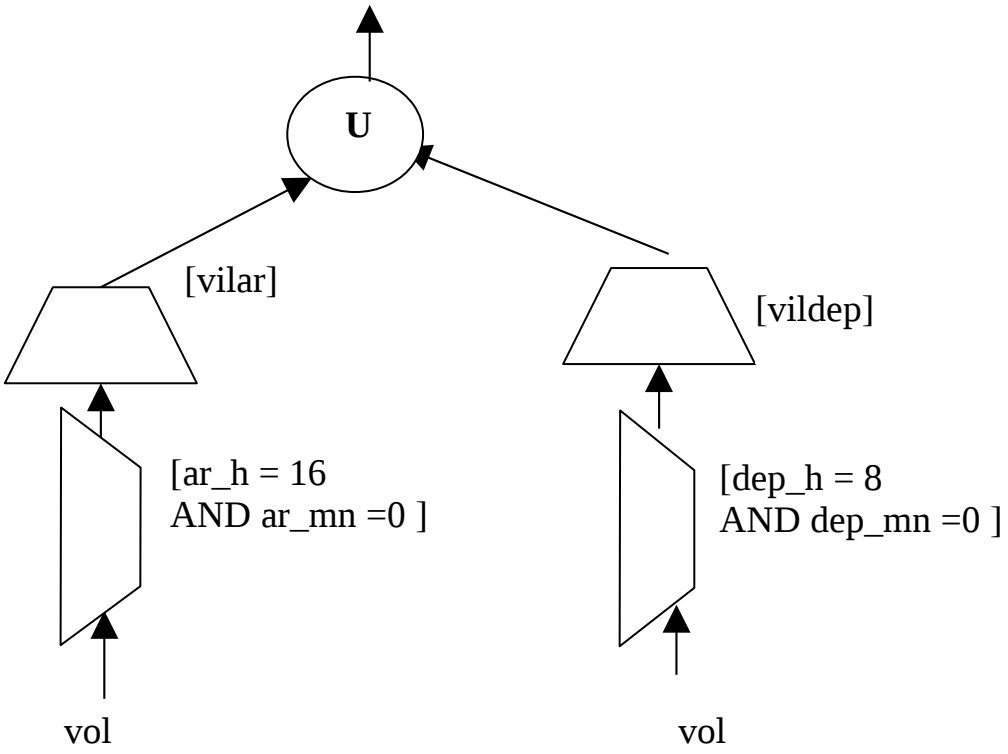
Nom de table Nom de table

3.5.1.3 Traduction dans le langage SQL

SELECT …
FROM …
WHERE …
UNION
SELECT …
FROM …
WHERE …
ORDER BY …..

Exemple :

Liste des villes d'arrivées de vol à 16h ou des villes de départ de vol à 8h.




```
SELECT vildep
FROM vol
WHERE  dep_h =16 AND dep_mn =0
UNION
SELECT vilar
FROM vol
WHERE  ar_h =8 AND ar_mn =0
ORDER BY 1;
```

3.5.2 Intersection

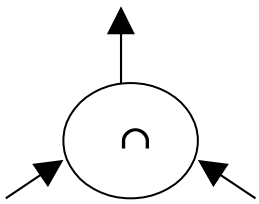
3.5.2.1 Définition

lignes appartenant simultanément aux 2 tables

conditions : même nombre d'attributs

même type pour attributs de même rang

3.5.2.2 Représentation Graphique



Nom de table Nom de table

3.5.2.3 Traduction dans le langage SQL

SELECT …
FROM …
WHERE …
INTERSECT
SELECT …
FROM …
WHERE …
ORDER BY …..

3.5.3 Différence

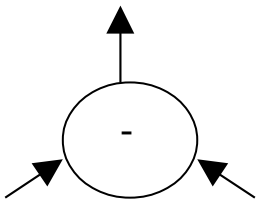
3.5.3.1 Définition

lignes appartenant à la 1ère table et pas la 2ème.

conditions : même nombre d'attributs

même type pour attributs de même rang

3.5.3.2 Représentation Graphique



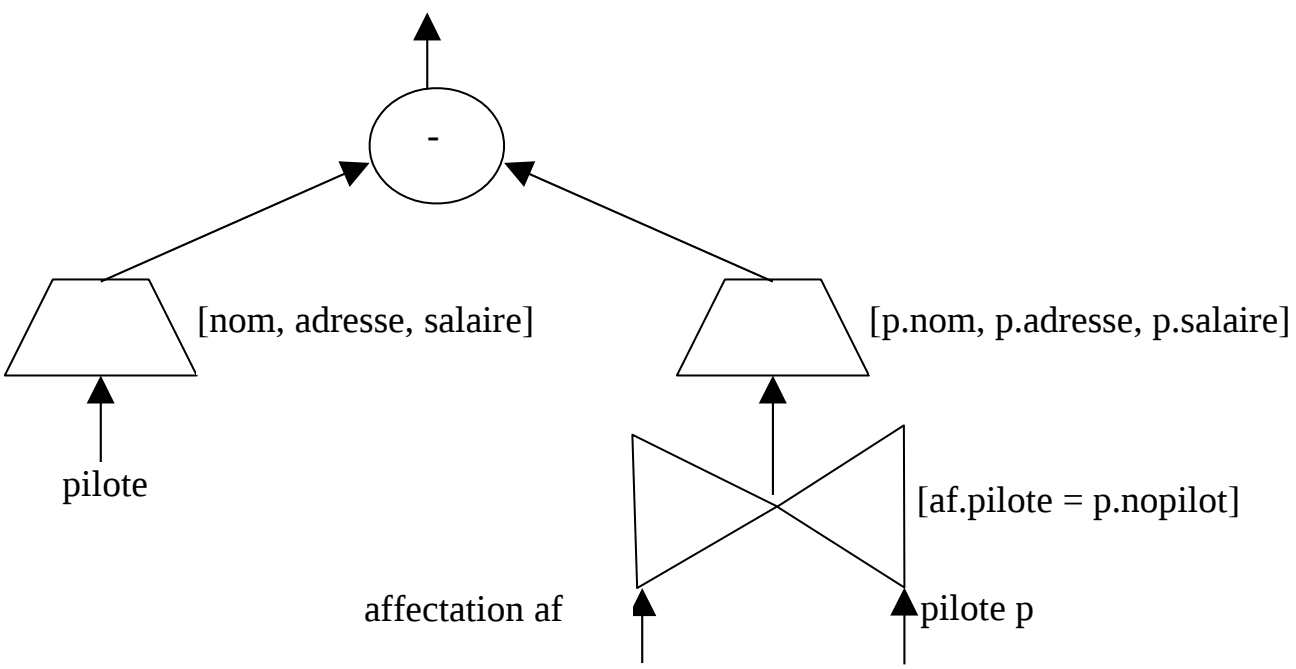
Nom de table Nom de table

3.5.3.3 Traduction dans le langage SQL

SELECT …
FROM …
WHERE …
EXCEPT
SELECT …
FROM …
WHERE …
ORDER BY …..

Exemple :

Liste des pilote n'ayant pas été affectés à un vol (nom, adresse, salaire)

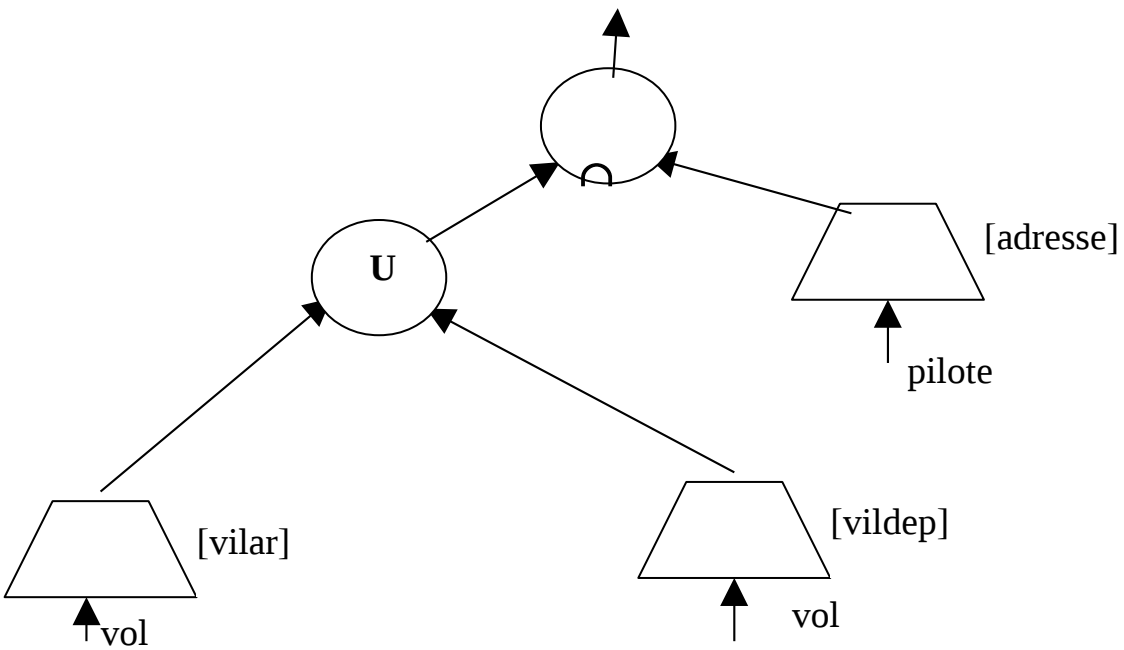


3.5.4. Combinaison de plusieurs opérateurs ensemblistes

- UNION, EXCEPT, MINUS, projection, sélection, jointure combinés
- Evaluation des ordres SELECT de gauche à droite
- Modification de l'ordre d'évaluation par des parenthèses.

Exemple :

Liste des noms des villes qui sont résidences d'un pilote




```
SELECT vilar FROM vol
 UNION
 SELECT vildep FROM vol
 EXCEPT
 SELECT adresse FROM pilote ;
```

3.6 Opérateur de Division

non implanté dans SQL de PostgreSQL.

4 Sous-requêtes

4.1 Principe

```
SELECT …
FROM …
WHERE attribut opérateur (  SELECT …
                            FROM …
                            WHERE …
                         )
```

Requête imbriquée

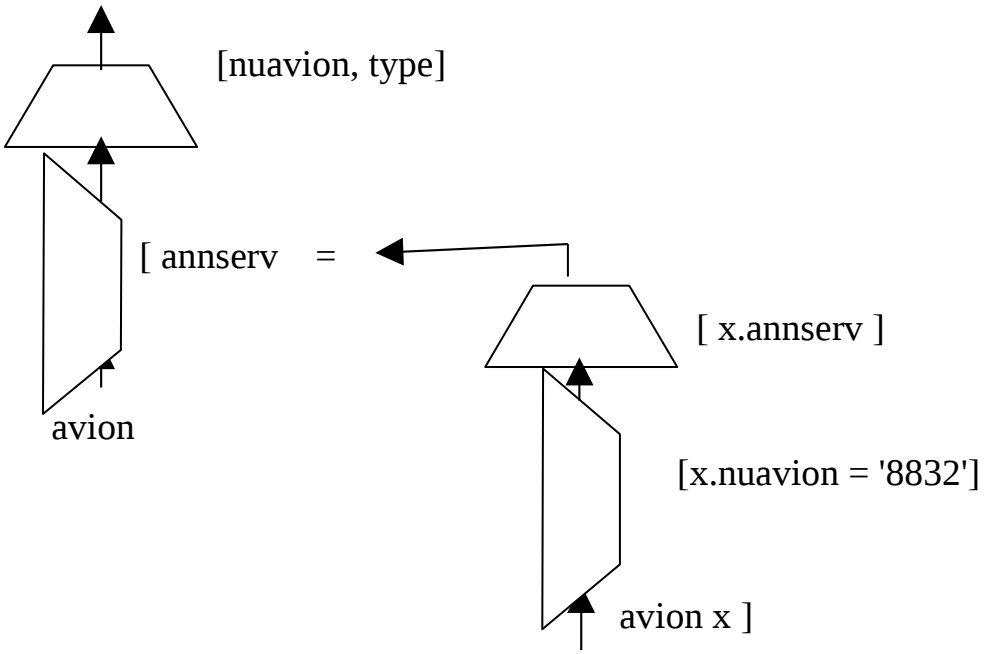
- plusieurs imbrications possibles (nombre de niveaux illimité)
- résultat de la sous requête de niveau n = valeur de référence dans la condition de la requête de niveau n -1 (principale)

4.2 Sous-requête indépendante, renvoyant 1 seule ligne

- valeur de référence de la condition de sélection unique
- évaluation de la sous requête avant la requête principale.

Exemple :

Type et n° des avions mis en service la même année que l'avion 8832



```
SELECT   nuavion, type
FROM     avion
WHERE    annserv  = ( SELECT x.annserv
                      FROM avion as x
                      WHERE x.nuavion = '8832'
                     );
```

4.3 Sous requête indépendante, pouvant renvoyer plusieurs lignes

- plusieurs valeurs de référence pour la condition de sélection
- Opérateur IN, opérateur (=, !=, <>, <, >, <=, >=) suivi de ALL ou ANY
- évaluation de la sous requête avant la requête principale.

IN :

Condition vraie si attribut ∈ {valeurs renvoyées par sous requête }.

ANY :

Comparaison vraie si vraie pour au moins 1 des valeurs ∈ {valeurs renvoyées par sous requête }.

ALL :

Comparaison vraie si vraie pour chacune des valeurs ∈ {valeurs renvoyées par sous requête }.

Remarque :

attribut **IN** (SELECT ... FROM …) équivalent à *attribut* **= ANY** (SELECT … FROM …)

attribut **NOT IN** (SELECT … FROM …) équivalent à *attribut* **!= ALL** (SELECT … FROM …)

Exemple :

Pilotes conduisant des avions de code type AB3. Lister le nom, date de vol, n° de vol.

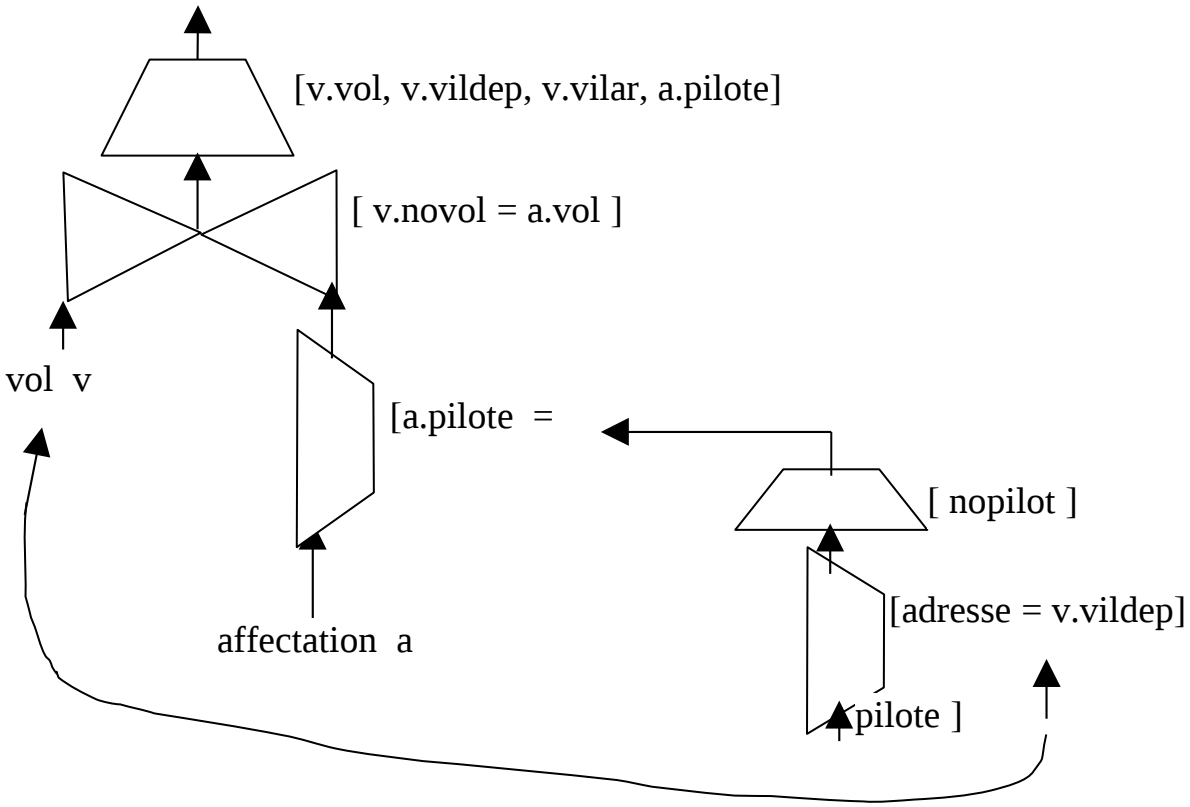
```
SELECT p.nom, a.date_vol, a.vol
 FROM   affectation as a, pilote as p
 WHERE a.avion IN (   SELECT nuavion FROM   avion
                                WHERE type = 'AB3' )
  AND p.nopilote = a.pilote;
```

4.4 Sous requête synchronisée avec la requête principale

- Sous requête faisant référence à une (ou des) colonne(s) de la (ou des) table(s) de la requête principale.
- Sous requête évaluée pour chaque ligne de la requête principale.

Exemple :

Liste des vols ayant un pilote qui habite la ville de départ du vol. Editer le numéro de vol, la ville de départ, la ville d'arrivée et le n° du pilote.



```
SELECT v.vol, v.vildep, v.vilar, a.pilote
FROM vol as v, affectation as a
WHERE v.novol = a.vol
 AND a.pilote IN (SELECT nopilot
                  FROM pilote
                  WHERE adresse = v.vildep
                 );
```

4.5 Cas particulier : opérateur EXISTS

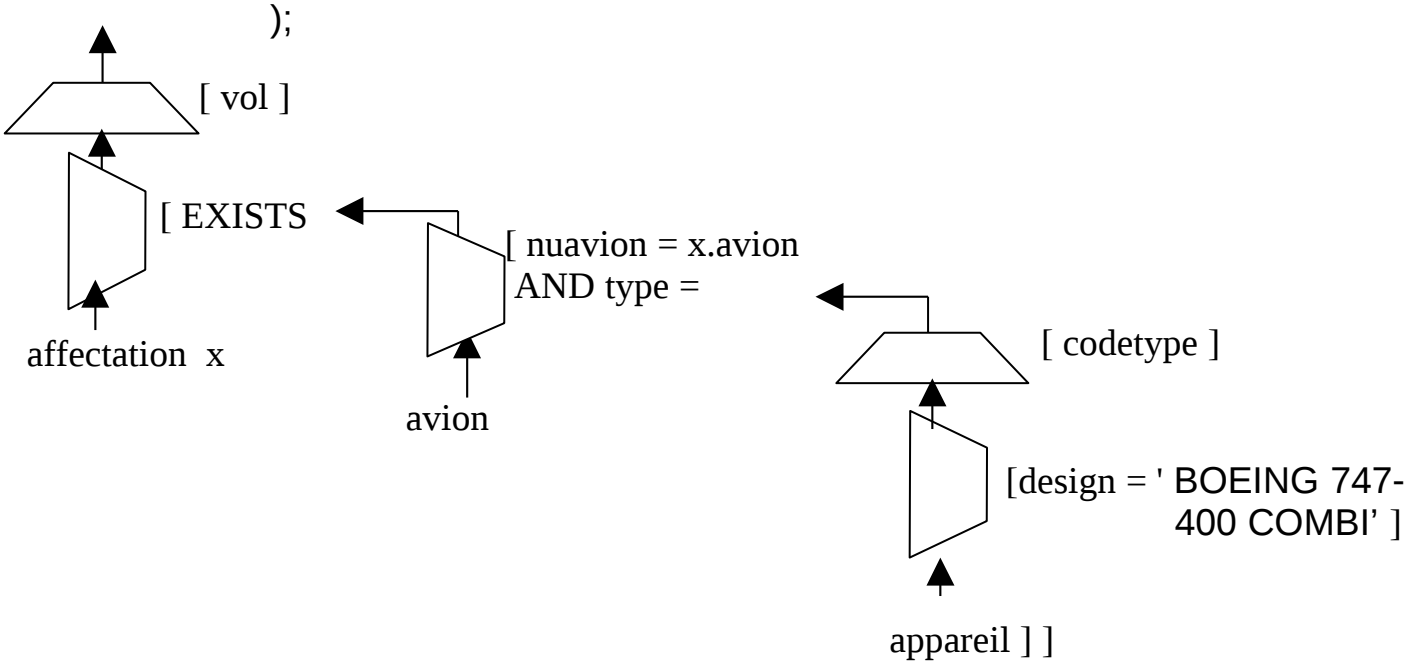
construit un prédicat évalué à vrai si la sous-requête renvoie au moins 1 ligne.

….WHERE EXISTS (SELECT ….)

Exemple :

Liste des n° de vols ayant utilisé au moins une fois un Boeing 747-400 COMBI.

```
SELECT vol
FROM affectation x
WHERE EXISTS (SELECT 'a'
                FROM avion
                WHERE nuavion = x.avion
               AND type = ( SELECT codetype
                            FROM appareil
                            WHERE design ='BOEING 747-400 COMBI'
                          )
             );
```

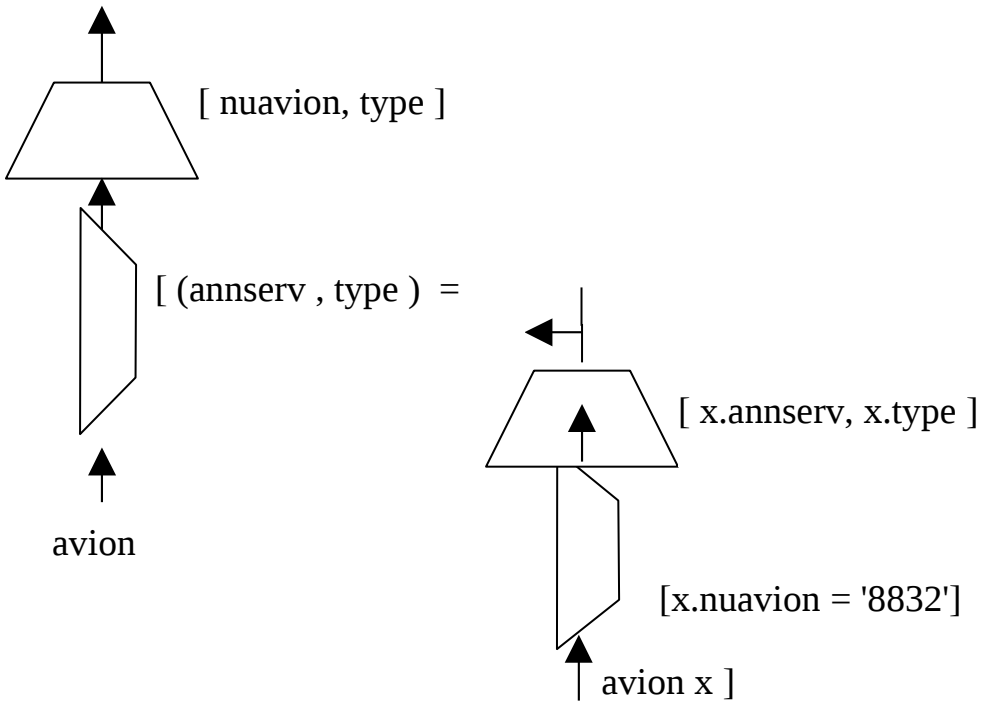


4.6 Sous requête avec plusieurs colonnes

```
SELECT …
FROM …
WHERE (colonne1, colonne2, …) = (SELECT colonne1, colonne2, …
                                 FROM …
                                 )
```

Exemple :

Type et n° des avions du même type que l'avion 8832 et mis en service la même année.




```
SELECT   nuavion, type
FROM     avion
WHERE      (annserv, type)  =  ( SELECT x.annserv, x.type
                                 FROM avion as x
                                 WHERE x.nuavion = '8832'
                                 );
```

5 Expressions et Fonctions

5.1 Définition, utilisations

expression = combinaison d'attributs, constantes, opérateurs, fonctions
dans : Projection, Sélection, Tri, Clause HAVING

Conversions implicites si combinaison de type d'expressions :

dates + chaînes de caractères **->** date
Nombres + chaînes de caractères **->** nombre

5.2 Expression arithmétique

attributs, constantes, fonctions arithmétiques, opérateurs, parenthèses

5.2.1 Opérateurs

(+, -, *, /) ***** et **/** prioritaires par rapport à **+** et **-**

5.2.2 Fonctions

POW (n, m) : n^m

ROUND (n, [,d]) : n arrondi à 10^d (par défaut d=0)
ROUND(n, 2) : conserve 2 décimales ROUND(n, -2) : arrondit à la centaine
ex : ROUND(2450,-2) = 2500 ROUND(2449,-2) = 2400

TRUNC (n, [,d]) : tronque n à 10^d (par défaut d=0)

CEIL (n) : entier directement >= n

FLOOR (n) : entier directement <= n (partie entière)

ABS (n) : valeur absolue de n

MOD (n, m) : le reste de la division de n par m

SIGN (n) : vaut 1 si n > 0 , 0 si n = 0, -1 si n < 0

SQRT (n) : $\sqrt{n}$ (si n < 0, résultat = NULL)

5.3 Expression sur chaînes de caractères

5.3.1 Opérateurs de concaténation

Chaîne1 **||** chaîne2 **->** chaîne1chaîne2

5.3.2 Fonctions

LENGTH (chaîne) : longueur de la chaîne

SUBSTR (chaîne, pos [,long]) : extraction d'une sous-chaîne de longueur *long* à partir de la position *pos***.**

STRPOS (chaîne, sous-chaîne) : -> position de *sous-chaîne* dans la *chaîne* (si **->** 0 : *sous-chaîne* $\notin$) .

UPPER (chaîne) : minuscules **->** majuscules

LOWER (chaîne) : majuscules **->** minuscules

INITCAP (chaîne) : met en majuscule la 1ère lettre de chaque mot $\in$ *chaîne*.

LPAD (chaîne, long, [,car]) : complète à gauche (ou tronque) *chaîne* à la longueur *long* par le caractère *car* (= ' ' par défaut).

RPAD (chaîne, long, [,car]) : idem à LPAD mais à droite**.**

LTRIM (chaîne, caractères) : supprime à gauche les caractères = *caractères*

RTRIM (chaîne, caractères) : idem à LTRIM mais à droite.

REPLACE (chaîne, chaîne_source, chaîne_remplacement) : remplace dans *chaîne* les *chaîne_source* par la *chaîne_remplacement*.

5.4 Expression sur DATES

5.4.1 Opérateurs

date + / - nombre **->** date + / - nombre de jours
date + / - interval **->** date + / - l'intervalle
date2 - date1 **->** nombre de jours entre date2 et date1
(réel si date1 et/ou date2 ⊃ heures)

Exemples :

Oper.	***Exemple***	***Résultat***
+	*date '2001-09-28' + integer '7'*	*date '2001-10-05'*
+	*date '2001-09-28' + interval '1 hour'*	*timestamp '2001-09-28 01:00'*
+	*date '2001-09-28' + time '03:00'*	*timestamp '2001-09-28 03:00'*
+	*time '03:00' + date '2001-09-28'*	*timestamp '2001-09-28 03:00'*
+	*interval '1 day' + interval '1 hour'*	*interval '1 day 01:00'*
+	*timestamp '2001-09-28 01:00' + interval '23 hours'*	*timestamp '2001-09-29 00:00'*
+	*time '01:00' + interval '3 hours'*	*time '04:00'*
+	*interval '3 hours' + time '01:00'*	*time '04:00'*
-	*- interval '23 hours'*	*interval '-23:00'*
-	*date '2001-10-01' - date '2001-09-28'*	*integer '3'*
-	*date '2001-10-01' - integer '7'*	*date '2001-09-24'*
-	*date '2001-09-28' - interval '1 hour'*	*timestamp '2001-09-27 23:00'*
-	*time '05:00' - time '03:00'*	*interval '02:00'*
-	*time '05:00' - interval '2 hours'*	*time '03:00'*
-	*timestamp '2001-09-28 23:00' - interval '23 hours'*	*timestamp '2001-09-28 00:00'*
-	*interval '1 day' - interval '1 hour'*	*interval '23:00'*
-	*interval '2 hours' - time '05:00'*	*time '03:00'*
-	*timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00'*	*interval '1 day 15:00'*
*	*double precision '3.5' * interval '1 hour'*	*interval '03:30'*
*	*interval '1 hour' * double precision '3.5'*	*interval '03:30'*
/	*interval '1 hour' / double precision '1.5'*	*interval '00:40'*

où les types : timestamp = Date et heure time =Heure interval = délai quelconque

5.4.2 Fonctions

AGE (date2, date1) : -> interval (intervalle) entre date2 et date1

DATE_TRUNC ([,précision], date) : -> *date* tronquée à la *précision* spécifiée.

Précision :

microseconds, milliseconds, second
minute, hour, day, month, year
decade, century, millennium

Ex : DATE_TRUNC('year',date '2004-08-24') : **->** 2004-01-01.

Ex : DATE_TRUNC('decade',date '2004-08-24') : **->** 2000-01-01.

DATE_PART ([,précision], date) ou DATE_PART ([,précision], interval) :
-> unité correspondant à la *précision* spécifiée.

Ex : DATE_PART('year',date '2004-08-24'): **->** 2004.
Ex : DATE_PART('minute', interval '3 days 4 hours 12 minutes') **->** 12.

CURRENT_DATE : date et heure courantes du système d'exploitation hôte**.**

5.5 Fonctions de conversion

5.5.1 Formatage d'un nombre : TO_CHAR()

TO_CHAR (nombre, masque) : -> chaîne

où masque :

- **caractères de substitution :**

  9 chiffre (zéro non significatif non représenté)

  0 chiffre (zéro non significatif représenté)

  V indique la puissance n de 10 du nombre. Ex : 999V99 **->** xxx * 10^2

- **caractères d'insertion :**

  . un point décimal apparaîtra à cet endroit

  , une virgule apparaîtra à cet endroit

  $ un $ précédera le 1^{er} chiffre significatif

- **autres :**

  EEEE le nombre est représenté avec un exposant

  MI le signe négatif sera à droite

  PR une valeur négative sera entre < >

Exemple : TO_CHAR(sal,'9990,V00')

5.5.2 Conversion d'une chaîne de caractères en nombre : TO_NUMBER()

TO_NUMBER (chaîne_de_caractères_numériques, masque) : -> nombre

5.5.3 Conversion d'une date en chaîne de caractères : TO_CHAR()

TO_CHAR (date, masque) : -> chaîne

où masque indique la partie de la date à afficher, et est une combinaison de :

SCC : siècle avec signe **CC :** siècle sans signe

SY, YYY : année (avec signe ou virgule) **Y,YYY :** année (avec virgule)

YYYY : année **YYY :** 3 chiffres **YY :** 2 chiffres **Y :** dernier chiffre

Q : n° trimestre dans l'année

WW : n° semaine dans l'année **W :** n° semaine dans le mois

MM : n° du mois

DDD : n° jour ds l'année **DD :** n° jour ds le mois **D :** n° jour ds la semaine

HH ou HH12 : heure (sur 12) **HH24 :** heure (sur 24)

MI : minutes

SS : secondes

J : jour du calendrier Julien

YEAR : année en lettre

MONTH : nom du mois **MON :** nom du mois sur 3 lettres

DAY : nom du jour **DY :** nom du jour sur 3 lettres

AM ou PM : indication AM ou PM

BC ou AD : indication BC (avant J. Christ) ou AD (après J. Christ)
Ex : TO_CHAR(CURRENT_DATE,'DD.MON.YY') **->** 23.MAR.98

5.5.4 Conversion d'une chaîne de caractères en date : TO_DATE()

TO_DATE (chaîne, masque) : -> date

où masque : idem à masque de TO_CHAR()

Ex : TO_DATE(datej,'DD.MON.YY') **->** 23.MAR.98

5.6 Autres fonctions

ASCII (caractère) :
-> code ASCII du caractère.

CHR (n) :
-> caractère dont le code ASCII = n

COALESCE (expr1, expr2, expr3, ...) :
-> 1ère expression non NULL, sinon NULL

6 Groupement des données

6.1 Principe

Ex : SELECT type, nuavion, nbhvol FROM avion ;
-> nombre d'heures de vol par avion

Possibilité de :
- calculer le nombre d'heures de vol par type d'avion
- sélectionner les types pour lesquels il existe plus de 5 avions ….

6.2 Définition d'un groupe

Ensemble de lignes, résultat d'une requête, ayant 1 valeur commune dans 1 ou plusieurs colonnes (facteur de groupage).

SELECT …FROM … WHERE …
GROUP BY *facteur de groupage*

6.3 Fonctions d'agrégat

-> calculs sur les données d'1 colonne pour les lignes d'1 même groupe.

AVG ([DISTINCT] expression) : moyenne

SUM ([DISTINCT] expression) : somme

MIN ([DISTINCT] expression) : plus petite valeur

MAX ([DISTINCT] expression) : plus grande valeur

COUNT ([DISTINCT] expression) : nombre de valeurs

VARIANCE ([DISTINCT] expression) : variance des valeurs

STDDEV ([DISTINCT] expression) : $\sqrt{}$ variance

DISTINCT : seule prise en compte des valeurs distinctes de l'expression

Remarque : seule prise en compte des valeurs non NULL de l'expression

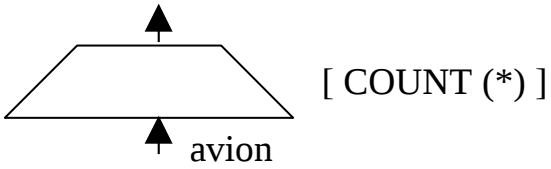
Cas particulier : COUNT(*)**->** nbre de lignes satisfaisant la condition WHERE

6.4 Calcul sur un seul groupe

Fonction de groupe sans de GROUP BY

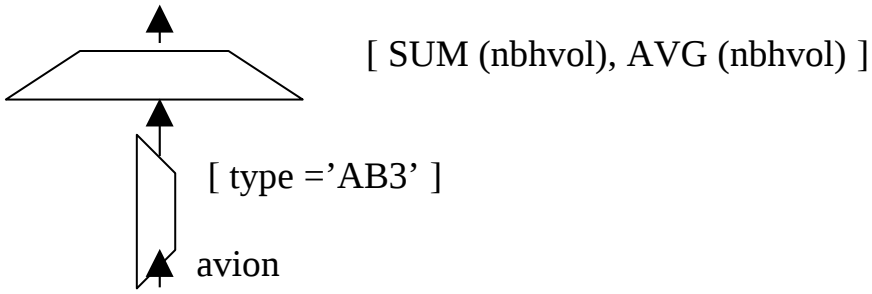
Exemples :

- Nombre total d'avions dans Gestair



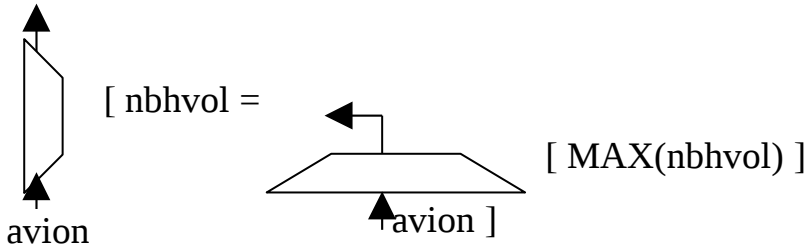
SELECT COUNT(*) FROM avion ;

- Nombre d'heures de vol cumulées pour les avions de type AB3, et moyenne



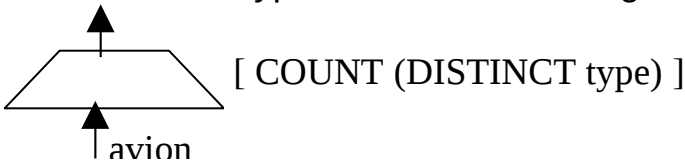
SELECT SUM(nbhvol), AVG (nbhvol) FROM avion WHERE type ='AB3' ;

- Avion ayant le plus grand nombre d'heures de vol



SELECT * FROM avion
WHERE nbhvol = (SELECT MAX(nbhvol) FROM avion) ;

- Nombre de type d'avions distincts gérés dans Gestair



SELECT COUNT(DISTINCT type)
FROM avion ;

6.5 Calcul sur plusieurs groupes

6.5.1 Généralités

Lignes de résultats d'un ordre SELECT subdivisées en plusieurs groupes

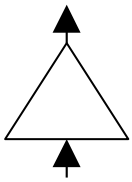
Groupe = lignes ayant 1 ou plusieurs caractéristiques communes

Nb groupes = nb valeurs distinctes des caractéristiques descriptives

Langage SQL :

SELECT …
FROM …
WHERE …
GROUP BY *expression1*, *expression2*, …
ORDER BY ….

Forme Graphique :



[expression1, expression2, …]

expression1, *expression2*, … : facteur de groupage

Remarque : 1 ligne résultat pour chaque groupe

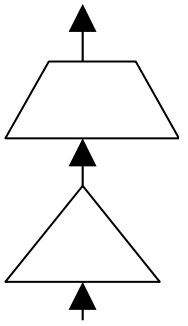
6.5.2 Cohérence du résultat

Attributs projetés :

- Fonctions de groupe
- Expressions figurant dans la clause GROUP BY

Exemples :

- Nombre total et moyenne d'heures de vol par type d'avion



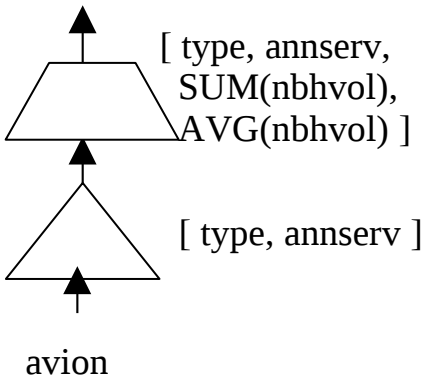
[SUM(nbhvol) , AVG(nbhvol), type]

[type]

avion

```
SELECT SUM(nbhvol), AVG(nbhvol), type
FROM avion
GROUP BY type ;
```

- Nombre total et moyenne d'heures de vol par type d'avion et par année de mise en service, tri par type et année




```
SELECT type, annserv, SUM(nbhvol), AVG(nbhvol)
FROM avion
GROUP BY type, annserv
ORDER BY 1,2 ;
```

6.5.3 Sélection de groupe : clause HAVING

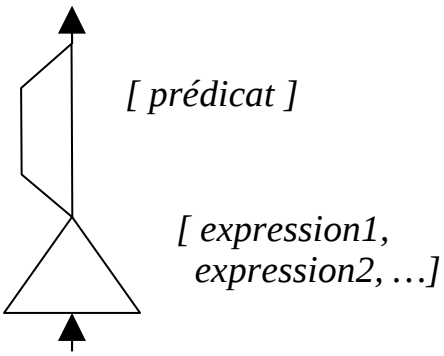
WHERE : Condition de sélection appliquée à des lignes

HAVING : Condition de sélection appliquée à des groupes

Langage SQL :

SELECT …
FROM …
WHERE prédicat
GROUP BY *expression1*, *expression2*, …
HAVING prédicat
ORDER BY ….

Forme Graphique :



Prédicat du HAVING :

- Mêmes règles de syntaxe que pour le prédicat du WHERE
- Conditions sur fonctions de groupe ou expression du GROUP BY

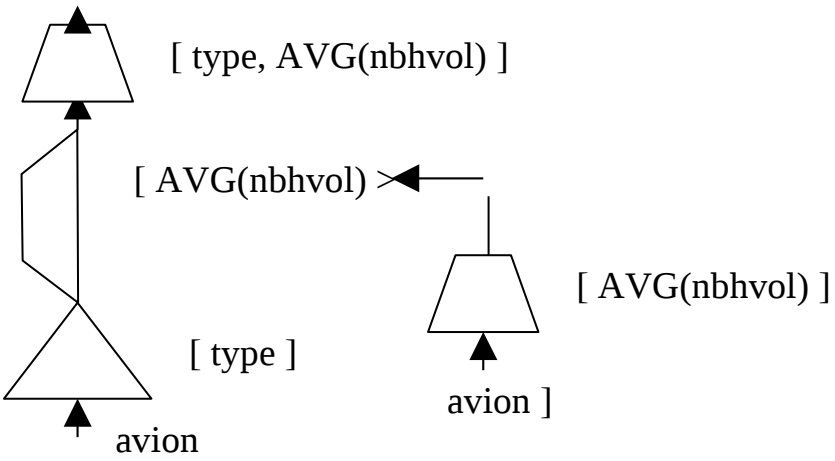
Exécution :

- sélection des lignes par WHERE
- constitution des groupes à partir des lignes sélectionnées par GROUP BY
- évaluation des fonctions de groupe sur les groupes

Exemples :

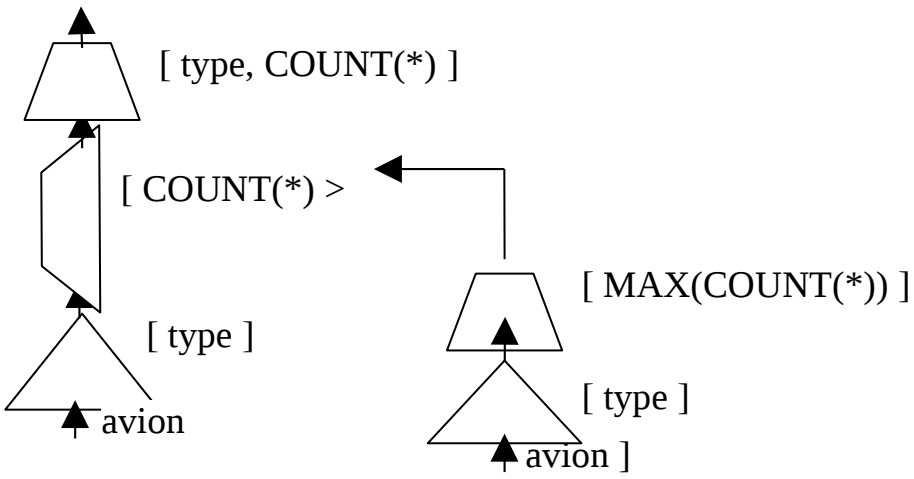
- moyennes des heures de vol par type d'avion pour les avions ayant un nbre d'heures moyen > la moyenne du nbre d'heures tout type confondu.

```
SELECT type, AVG(nbhvol)  "Moyenne heures de vol"
FROM avion
GROUP BY type
HAVING AVG(nbhvol)   >  ( SELECT AVG(nbhvol) FROM avion );
```



- type d'avion qui comporte le plus d'avions (type, nombre d'avions) ?

```
SELECT type, COUNT(*)  "Nombre d'avions"
FROM avion
GROUP BY type
HAVING COUNT(*)   =    (   SELECT MAX(COUNT(*))
                           FROM avion GROUP BY type ) ;
```



Fonction de groupe à 2 niveaux **interdit** avec PostgreSQL => utiliser 1 vue

7 Modification des données - UPDATE

Modification des valeurs d'1 ou plusieurs colonnes, dans 1 ou plusieurs lignes d'une table

UPDATE table
SET colonne1 **=** { expression1 | (SELECT …) }
[,colonne2 **=** { expression2 | (SELECT …) } …]
[**WHERE** prédicat] ;

remarque :

SELECT ne doit ramener qu'1 seule ligne mais peut ramener plusieurs colonnes

Exemple :

Augmenter de 10% le salaire mensuel des pilotes n'ayant pas de commission

```
UPDATE pilote
SET sal = sal * 1.1
WHERE comm IS NULL;
```

8 Insertion de lignes – INSERT

8.1 Insertion d'une ligne

Insertion d'une ligne dans une table en spécifiant les valeurs à insérer

INSERT INTO table [(colonne1, colonne2 ...)]
VALUES (valeur1, valeur2, …) **;**

- Par défaut, liste de colonnes = ensemble ordonné des colonnes de la table
- Valeurs nulles pour colonnes non spécifiées
- Correspondance positionnelle entre noms de colonnes cités et valeurs
- Valeurs possibles : Constante, NULL, Résultat de expression

Exemple :

```
INSERT INTO affectation
VALUES ('AF3218', TO_DATE('17/02/94' ,'DD/MM/YY'), '1243', '8432') ;
```

8.2 Insertion de plusieurs lignes

Lignes à insérer résultats de sous requête

INSERT INTO table [(colonne1, colonne2, …)]
SELECT ….

Remarque : pas de ORDER BY ni de CONNECT BY dans la sous requête.

8.3 Création et insertion simultanées

CREATE TABLE table
AS SELECT …

Les colonnes de la table créée héritent du nom et du type des attributs projetés

Exemple :

Création de la table RECAPAVION qui donne pour chaque type d'avion le total et la moyenne des heures de vol effectuées :

```
CREATE TABLE recapavion
AS SELECT type, SUM(nbhvol) total, AVG(nbhvol) moyenne
FROM avion
GROUP BY type ;
```

9 Suppression de lignes – DELETE

Suppression des lignes satisfaisant une condition :

DELETE FROM table
[**WHERE** prédicat] **;**

Exemple :

Supprimer les affectations au départ de Marseille à partir du 01/05/92

```
DELETE FROM affectation
WHERE date > TO_DATE('01-MAY-92','DD-MON-YY')
AND vol IN   (     SELECT novol
                   FROM vol
                   WHERE vildep = 'MARSEILLE'
              ) ;
```

10 Suppression de lignes – TRUNCATE

Suppression de toutes les lignes d'une table :

TRUNCATE TABLE table**;**

Commande plus rapide en exécution que la commande DELETE.
Ne peut être exécutée dans un bloc transactionnel chaîné (voir plus loin).

11 Gestion des transactions

11.1 Notion de transaction

= ensemble minimal de modifications de la base

L'utilisateur travaille sur 1 copie privée des tables => modifications locales

Modifications effectives pour l'ensemble des utilisateurs si validation

Possibilité de revenir à tout moment à l'état précédent la m-a-j

11.2 Positionnement d'un début de transaction

Par exécution de la commande **BEGIN TRANSACTION;**

11.3 Validation d'une transaction

Par exécution de la commande **COMMIT**

ou

lors de la sortie définitive de SQL par **EXIT.**
=> Toutes les modifications réalisées par la transaction sont appliquées à la BD.

11.4 Annulation d'une transaction

Par exécution de la commande **ROLLBACK**

=> Toutes les modifications depuis le début de la transaction sont défaites.
à condition de ne pas avoir fait un COMMIT après ce point.

Fin anormale d'1 tâche => exécution automatique de ROLLBACK pour transactions non terminées.

LE LANGAGE DE DEFINITION DE DONNEES (LDD)

Les séquences

1 Généralité

Générateur de valeurs séquentielles (N° de séquence) uniques pour :

. Générer des valeurs de clé primaire
. Coordonner les valeurs de clé dans plusieurs lignes ou tables.

2 Mise en œuvre du générateur

2.1 Création d'une définition de séquence

CREATE SEQUENCE [schéma.]nom_séquence
[INCREMENT valeur] [START valeur]
[MAXVALUE valeur] [MINVALUE valeur]
[CYCLE] [CACHE valeur]

Où:

nom_séquence	nom du N° de séquence enregistré ds le dico de données.
INCREMENT	pas d'incrémentation du N° de séquence (>0 ou <0).
START	valeur de départ (par défaut = MINVALUE pour séq. Asc. ou MAXVALUE pour séq. Des.).
CYCLE	si N° de séquence atteint valeur MAXVALUE (resp. MINVALUE), repart à MINVALUE (resp. MAXVALUE). Par défaut pas de cycle;
MAXVALUE	limites haute (par défaut 10E27-1) si sens ascendant
MINVALUE	limite basse (par défaut 1) si sens descendant
CYCLE	reprise après MAXVALUE (resp MINVALUE) .
CACHE	demande 1 pré-génération de N° de séquence / pas d'attente lors d'1 demande de valeur (par défaut 20 valeurs stockées en mémoire).

Remarque :

Privilège nécessaire pour création d'1 définition de séquence.

Transmission de droit possible sur 1 N° de séquence (comme table).

2.2 Utilisation

N° de séquence appelé par SELECT, INSERT ou UPDATE (pseudo-colonne) :

NEXTVAL('nom_séquence').

valeur suivante générée

CURRVAL('nom_Séquence').

donne valeur courante du N° de séquence (si déjà N° généré par appel à NEXTVAL).

SETVAL('nom_Séquence', valeur).

Positionne la valeur courante du N° de séquence.

Remarque :

Même nom_Séquence utilisable simultanément par plusieurs uts. (privilégiés).
=>Or N° de séquence générés uniques
=> valeurs 'vues' par chaque ut. avec trous dans la séquence

Exemple

Création :

```
CREATE SEQUENCE espilote
START WITH 1000
INCREMENT BY 10;
```

Utilisation :

```
SELECT NEXTVAL('espilote');
 -> affiche 10

INSERT INTO pilote (nopilot,nom, adresse)
    VALUES (CURRVAL('espilote'),'DUPOND','NICE');
 -> insert  '10       DUPOND   NICE'
```

2.3 Suppression

=> supprimer 1 génération de N° de séquence :

DROP SEQUENCE [schéma.]nom_séquence

Les contraintes d'intégrité

1 Généralités

but : assurer le maintien de la cohérence des données de la base.

Etat :

Activées : vérifiées par noyau lors de l'exécution d'instructions d'accès aux données (par SQL, outils associés au SGBD).

Désactivées : présentes à titre descriptif dans le dico. de données.

2 Expression des contraintes

5 types de contraintes :

- Caractère obligatoire ou facultatif;
- Unicité des lignes;
- Clé primaire;
- Intégrité référentielle ou clé étrangère;
- Contrainte de valeurs.

Exprimées au niveau :

- table (contraintes globales, concernent 1 ensemble de colonnes)
- colonne (locale, concerne la colonne)

Définies lors :

- création d'1 table (CREATE TABLE),
- modification de structure d'1 table (ALTER TABLE).

Nommées :

- nom implicite donné par SGBD
- nom explicite donné par la clause CONSTRAINT

2.1 Contraintes de table

Les options suivantes définissent les contraintes au niveau table :

- UNIQUE (liste de colonnes)
- PRIMARY KEY (liste de colonnes)
- FOREIGN KEY (liste de colonnes) REFERENCES nom_table(liste de colonnes) [ON DELETE CASCADE]
- CHECK condition

2.2 Contraintes de colonne

Les options suivantes définissent les contraintes au niveau colonne :

- NULL | NOT NULL
- UNIQUE
- PRIMARY KEY
- REFERENCES nom_table (colonne) [ON DELETE CASCADE]
- CHECK condition

3 Etude des différentes contraintes

3.1 Caractère obligatoire / facultatif

NULL autorise la colonne à ne pas avoir de valeur pour certaines lignes.

NOT NULL la colonne doit posséder 1 valeur pour toutes les lignes de la table.

Ex :

```
        nom  VARCHAR(25)                                  NOT NULL
ou      nom  VARCHAR(25)        CONSTRAINT nn_nom        NOT NULL
```

3.2 Unicité

spécifier les clés candidates.
2 tuples de la table ne peuvent pas avoir la même valeur de clé unique.

ex : cdintrv CHAR(5) NOT NULL CONSTRAINT unq_cdintrv **UNIQUE**
ex : codsup CHAR(5) CONSTRAINT unq_codsup **UNIQUE**

Une clé unique peut prendre des valeurs nulles.

Clé composée : spécifiée à part, après ou avant la déf. des attributs :

```
ex : CREATE TABLE monde
     (
     ville   VARCHAR(15),
     pays  VARCHAR(15)
     ......
     CONSTRAINT unq_ville_pays UNIQUE(ville,pays)
     );
```

La contrainte *unq_ville_pays* assure que la même combinaison des valeurs de ville et pays n'apparaît pas plus d'1 fois dans la table.

Index automatiquement créé avec comme nom le nom de la contrainte (PostgreSQL).

3.3 Clé primaire

permet d'identifier chaque ligne de manière unique.

Clé primaire à valeur déterminée (non nulle) et unique pour la table.

rem : clé primaire **=>** clé unique (l'inverse n'est pas vrai)

Clé primaire non composée : spécifier **PRIMARY KEY** ds la définition de l'attribut

ex : cdprs CHAR(5) CONSTRAINT pk_cdprs **PRIMARY KEY**

Clé composée (segmentée) : spécifiée à part, après ou avant déf. des attributs

```
ex : CREATE TABLE order
     (
      att1  NUMERIC,
      att2  NUMERIC,
     ....,
     CONSTRAINT pk_order PRIMARY KEY (att1,att2)
     );
```

Index automatiquement créé avec comme nom le nom de la contrainte (PostgreSQL).

3.4 Clé étrangère, intégrité référentielle

3.4.1 Définition

clé étrangère ou clé externe dans 1 table T1 = toute colonne (ou combinaison de colonnes) qui apparaît comme colonne clé primaire ou clé unique dans une autre table T2, appelée table primaire.

3.4.2 Syntaxe

La clause :

FOREIGN KEY (colonne [,colonne] ....

définit une clé étrangère sur la table T1

La clause :

REFERENCES [schéma.]nom_table [(colonne [,colonne] ....)]

[ON DELETE CASCADE]

définit la clé primaire référencée en tant que clé étrangère dans la contrainte d'intégrité référentielle. Avec :

nom_table : table de base.

colonne : clé primaire (PRIMARY KEY) ou colonnes(s) décrite(s) avec l'option UNIQUE, dans T2.

ON DELETE CASCADE : supprimer automatiquement valeurs d'1 colonne de T2 référencée par FOREIGN KEY et lignes correspondantes ds T1.

ex :
```
CREATE TABLE emp
(
 empno     NUMERIC(4),
 ename     VARCHAR(10),
 job       VARCHAR(9),
...
deptno     NUMERIC(2) CONSTRAINT fk_deptno REFERENCES dept(deptno)
);
```

La contrainte *FK_DEPTNO* assure que tous les employés de la table *EMP* travaillent dans un département de la table *DEPT*.

Cependant, certains employés peuvent avoir un N° de département nul.
Clé étrangère segmentée : spécifiée à part après ou avant déf. des attributs

ex : CREATE TABLE order2
(
att1 NUMERIC,
att2 NUMERIC NOT NULL,
....,
CONSTRAINT fk_od2 FOREIGN KEY(att1,att2)REFERENCES personnel(att1, att2)
);

Remarques :

- Colonnes clé étrangère et clé primaire : même type de données.
- Une colonne clé étrangère peut ne pas avoir de valeurs.
- Une clé étrangère peut faire référence à la clé primaire de la même table.
- Pour définir une contrainte REFERENCE sur T2, le créateur de T1 doit avoir le privilège de créer des références sur cette table.

3.4.3 Mise *en œuvre*

- Exécution de INSERT ou UPDATE possible ds T1, que si la valeur de la clé étrangère existe dans la clé primaire T2
- Impossible de supprimer des lignes par DELETE dans T2 tant qu'il ∃ des lignes dans T1 ayant comme valeur de clé étrangère 1 valeur = valeur de la clé primaire de T1 à supprimer.

ON DELETE CASCADE : suppression d'1 occurrence de la table primaire
=> suppression occurrences liées de la table secondaire.

3.5 Contraintes de valeurs

CHECK : spécifie une contrainte qui doit être vérifiée à tout moment par les tuples de la table.

ex : cdtpi CHAR(2) NOT NULL
CONSTRAINT check_cdtpi CHECK (cdtpi IN ('01,'02','03'))

ex : salaire NUMERIC(7,2) CHECK (salaire > 6000)

ex : tpsp NUMERIC(4) NOT NULL
CONSTRAINT check_tpsp CHECK (tpsp BETWEEN 5 AND 480)

ex : nom VARCHAR(9)
CONSTRAINT check_nom CHECK (nom = UPPER(nom))

(rem : *check_nom* assure que les noms sont en majuscule).

Contrairement aux autres types de contraintes, CHECK appliquée à 1 colonne de la table peut imposer des règles sur les autres colonnes.

ex : CREATE TABLE emp
(
empno NUMERIC(4),
ename VARCHAR(10),
job VARCHAR(9),
sal NUMERIC(7,2)
comm NUMERIC(7,2),
deptno NUMERIC(2),
CONSTRAINT check_sal_comm CHECK (**sal + comm** <= 10000)
);

3.6 Exemples de description de table avec contraintes

Exemple 1 :

```
CREATE TABLE ligne_Commande
(
numcom    NUMERIC  NOT NULL,
nuligne   NUMERIC  NOT NULL CHECK (nuligne > 0),
nuprod    CHAR(8)          NOT NULL,
qte_cmd   NUMERIC(2)       NOT NULL CHECK (qte_cmd >0),
qte_livrée  NUMERIC (2),
CONSTRAINT ct_lgcmd_1 PRIMARY KEY (numcom, nuligne),
CONSTRAINT ct_lgcmd_2 FOREIGN KEY (numcom)
      REFERENCES martin.commande (numcom),
CONSTRAINT ct_lgcmd_3 FOREIGN KEY (nuprod)
      REFERENCES martin.produit (nuprod),
CONSTRAINT ct_lgcmd_4 CHECK (qte_cmd >= qte_livrée)
) ;
```

****

Exemple 2 :

```
CREATE TABLE order_detail
    (CONSTRAINT pk_od PRIMARY KEY (order_id, part_no),
    order_id    NUMERIC
                CONSTRAINT fk_oid REFERENCES order(order_id),
    part_no     NUMERIC
                CONSTRAINT fk_pno REFERENCES part(part_no),
    quantity    NUMERIC
                CONSTRAINT nn_qty NOT NULL
                CONSTRAINT check_qty_low CHECK (quantity > 0),
    cost        NUMERIC
                CONSTRAINT check_cost CHECK (cost > 0)
    );
```

Exemple 3 :

```
CREATE TABLE emp
(empno      NUMERIC
            CONSTRAINT pk_emp PRIMARY KEY,
ename       VARCHAR(10)
            CONSTRAINT nn_ename NOT NULL
            CONSTRAINT upper_name CHECK(ename = UPPER(ename)),
job         VARCHAR(9),
chef        NUMERIC
            CONSTRAINT fk_chef REFERENCES emp(empno),
hiredate    DATE                DEFAULT CURRENT_DATE,
sal         NUMERIC(10,2)
            CONSTRAINT check_sal CHECK (sal > 4000),
comm        NUMERIC(9,0)        DEFAULT NULL,
deptno      NUMERIC(2)
            CONSTRAINT nn_deptno NOT NULL
            CONSTRAINT fk_deptno REFERENCES dept(deptno));
```

4 Gestion des contraintes

4.1 Description

effectuée lors de la création ou de la modification d'une table.

pour une contrainte de colonne :

[CONSTRAINT nom_Contrainte]
{
[NOT] NULL | UNIQUE | PRIMARY KEY

| REFERENCES [schéma.]table[(colonne)] [ON DELETE CASCADE]
[DEFERRABLE | NOT DEFERRABLE] [INITIALLY DEFERRED | INITIALLY IMMEDIATE]
| CHECK (condition)
}

pour une contrainte de table :

[CONSTRAINT nom_Contrainte]
{
[UNIQUE | PRIMARY KEY] (colonne [,colonne] ...)
| FOREIGN KEY (colonne [,colonne] ...)
REFERENCES [schéma.]table[(colonne [,colonne] ...)] [ON DELETE CASCADE]
[DEFERRABLE | NOT DEFERRABLE] [INITIALLY DEFERRED | INITIALLY IMMEDIATE]
| CHECK (condition)
}

avec :

DEFERRABLE Permet de reporter l'activation de la contrainte à la fin d'une transaction, plutôt que de l'activer à la fin de chaque requête action (update, delete,insert). Dans ce cas, ajouter INITIALLY DEFERRED.

NOT *DEFERRABLE* Par défaut. Requêtes actions validées au fur et à mesure. Inutile d'ajouter alors *INITIALLY IMMEDIATE (par défaut).*

4.2 Suppression d'une contrainte

ALTER TABLE [schéma.]table

DROP définition_contrainte [CASCADE]

avec restrictions :

suppression impossible d’1 clé primaire ou unique utilisée dans 1 contrainte d'intégrité référentielle.

-> supprimer à la fois la clé référencée et la clé étrangère en spécifiant la clé référencée avec l'option CASCADE.

4.3 Ajout d'une contrainte

ALTER TABLE [schéma.]table

ADD définition_contrainte;

4.4 Activer/désactiver une ou plusieurs contraintes

SET CONSTRAINTS { ALL | nom_contraint [, ...] }

{ DEFERRED | IMMEDIATE }

avec restrictions :

DEFERRED interdit si la contrainte a été déclarée **NOT DEFERRABLE** (par défaut).

Script SQL de la base de données Gestair

```
-- ========================================
--  Nom de la base      : Gestair
--  Nom de SGBD         : PostgreSQL version 9.x
--  Date de creation : 01/01/2014
--  Auteur          : Myriam Mokhtari-Brun
-- ========================================

– REM Pour avoir les informations de l'utilisateur en cours et de la date

SELECT    USER as utilisateur,
          TO_CHAR(CURRENT_DATE,'DAY DD-MONTH-YY HH24:MI') as "date";



CREATE TABLE vol
      (
      novol       CHAR(6)                    PRIMARY KEY,
      vildep      VARCHAR(30)   NOT NULL,
      vilar       VARCHAR(30)   NOT NULL,
      dep_h       NUMERIC(2)    NOT NULL CHECK(dep_h BETWEEN 0 AND 23),
      dep_mn      NUMERIC(2)    NOT NULL CHECK(dep_mn BETWEEN 0 AND 59),
      ar_h        NUMERIC(2)    NOT NULL CHECK(ar_h BETWEEN 0 AND 23),
      ar_mn       NUMERIC(2)    NOT NULL CHECK(ar_mn BETWEEN 0 AND 59)
      );


CREATE TABLE appareil
      (
      codetype    CHAR(3)                    PRIMARY KEY,
      nbplace     NUMERIC(3)    NOT NULL,
      design      VARCHAR(50)   NOT NULL
      );


CREATE TABLE avion
      (
      nuavion     CHAR(4)                    PRIMARY KEY,
      type        CHAR(3)       NOT NULL REFERENCES appareil(codetype),
      annserv     NUMERIC(4)    NOT NULL,
      nom         VARCHAR(50),
      nbhvol      NUMERIC(8)    NOT NULL
      );
```

```
CREATE TABLE pilote
        (
        nopilot     CHAR(4)                 PRIMARY KEY,
        nom         VARCHAR(35)   NOT NULL,
        adresse     VARCHAR(30)   NOT NULL,
        sal         NUMERIC(8,2)  NOT NULL,
        comm        NUMERIC(8,2),
        embauche    DATE          NOT NULL
        );

CREATE TABLE affectation
        (
        vol         CHAR(6)       NOT NULL REFERENCES vol(novol),
        date_vol    DATE          NOT NULL,
        nbpass      NUMERIC(3)    NOT NULL,
        pilote      CHAR(4)       NOT NULL REFERENCES pilote(nopilot),
        avion       CHAR(4)       NOT NULL REFERENCES avion(nuavion),
        PRIMARY KEY (vol,date_vol)
        );
```

LES VUES

1 Concept de vue

Vue = table virtuelle pour vision logique des données (schéma externe)

Stockage de la description de la vue sous forme d'une requête.

Pas de données associée.

Intérêts :

- Réponse aux besoins de confidentialité
- Facilité pour les utilisateurs dans la manipulation de données (requêtes complexes)
- Sauvegarde des requêtes dans le dictionnaire de données

2 Création d'une vue

CREATE [OR REPLACE]
VIEW nom_vue [(colonne1, colonne2, …)]
AS SELECT …

Exemple 1:

Restriction de la table *pilote* aux pilotes habitant Paris :

CREATE VIEW v_pilote AS SELECT * FROM pilote WHERE adresse = 'Paris' ;

3 Utilisation des vues

3.1 Interrogation à travers une vue

Vue référencée par SELECT à la place d'une table de base.

Exemple 1 :

SELECT * FROM v_pilote ;

Exemple 2 :

Solution à la question : type d'avion qui comporte le plus d'avions (type, nombre d'avions)? de la page p94 .

Fonction de groupe à 2 niveaux interdite.
=> nécessité de créer d'abord une vue calculant le nombre d'avions par type :

CREATE VIEW **nb_avions** AS SELECT COUNT(*) as **nb**FROM avion GROUP BY type ;

Puis faire référence au max :

SELECT type, COUNT(*) "Nombre d'avions"
FROM avion
GROUP BY type
HAVING COUNT(*) = (SELECT MAX(**nb**) FROM **nb_avions**) ;

3.2 Mise à jour à travers une vue

Modifications de données par INSERT, DELETE et UPDATE à travers une vue impossibles.

3.3 Transmission de droits

Donner des droits à d'autres utilisateurs sur seulement un sous-ensemble des colonnes de sa table
=> créer une vue et ne donner des droits que sur la vue

Exemple :

Donner le droit d'accès aux autres utilisateurs sur seulement le nom et la date d'embauche des pilotes habitant PARIS, à travers une vue.

CREATE VIEW r_pilote
AS SELECT nom, embauche FROM pilote WHERE adresse = 'PARIS' ;

GRANT SELECT ON r_pilote TO PUBLIC ;

4 Suppression d'une vue

DROP VIEW nom_vue [CASCADE];

CASCADE : supprimer la vue et supprimer celles qui en dépendent.

Les index

1 Généralités

index = objet optionnel associé à 1 table ou vue ou cluster

utilisé :

- Comme accélérateur dans l'exécution des requêtes :

  ```
  SELECT * FROM pilote
  WHERE nom ='MARTIN';
  ```

  Balayer toute la table retrouver la ou les lignes / nom = 'MARTIN'.

  => temps de réponse prohibitifs pour grosses tables (+ de 100 lignes).

  Solution : créer index pour améliorer performances requêtes de recherche.

- Comme traduction d'une clé primaire :

  => index automatiquement crée lorsque PRIMARY KEY ou UNIQUE définie à la création d'une table.

Plusieurs index associés à 1 même table.

Index créé à tout moment sur 1 table.

index créé => m-à-j automatique à chaque modification de la table

=> temps d'insertion et temps de modification des lignes augmenté.

index sur 1 ou plusieurs colonnes (*index composé).*

2 Structure d'un Index

index = structure pour retrouver 1 ligne dans 1 table à partir de la valeur d'1 colonne ou d'1 ensemble de colonnes.

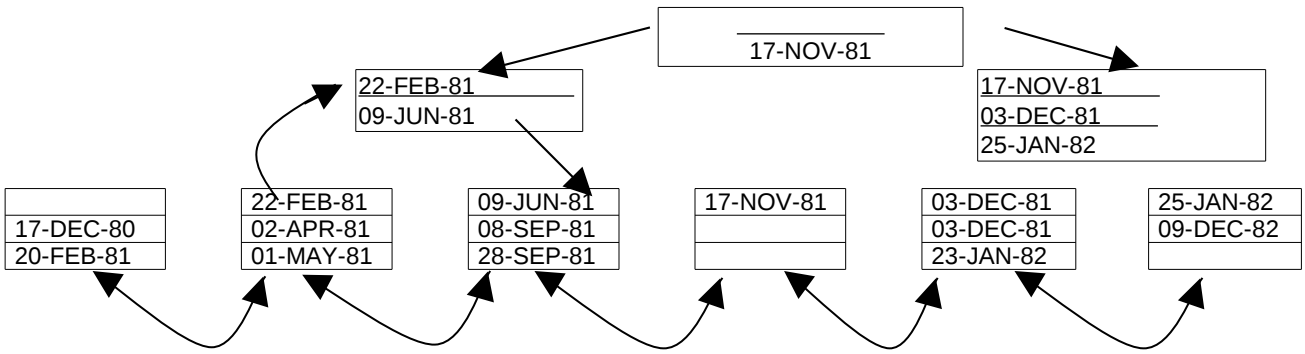
Index stocké sous forme d'arbre équilibré (B*-arbre) :

bloc racine
bloc intermédiaire qui pointent sur les
blocs feuilles (⊃ données index et valeurs des ROWID correspondant aux lignes associées).

Blocs feuilles = même profondeur

=> même tps de recherche ∀ valeur de l'index.

Ex. de structure d'index :



3 Gestion des index

3.1 Création

CREATE *[UNIQUE]* INDEX [schéma.]nom_d'index

ON nom_de_table (nom_colonne [ASC | DESC], [nom_colonne [ASC | DESC]], ...)

Où:

UNIQUE : 2 lignes n'ont pas même valeur pour l'index

ex1 : Créer 1 index sur nom client ds table *clients*. *(utile si on souhaite parcourir la table par ordre alphabétique des noms de client)*.

CREATE INDEX *i_client_nom* ON *client*(*nom*);

i_client_nom : nom de l'index.

ex2 : Créer index sur les valeurs décroissantes de la date de commande et pour chaque date
sur l'ordre croissant des numéros de *clients*.

CREATE INDEX *i_cmd_date_num* ON *commandes*(*datecom* DESC, *numcli* ASC);

rem : Par défaut, index crée selon ordre croissant.

ex3 : La notion de clé primaire, implantée par la clause PRIMARY KEY lors de la création de la table

CREATE TABLE *client*
(
cdcli CHAR(5) PRIMARY KEY,
...
);

peut aussi être implantée par la création d'un index avec l'option UNIQUE

CREATE TABLE *client*
(
cdcli CHAR(5),
...
);
CREATE UNIQUE INDEX *i_cdcli* ON *client(cdcli);*

3.2 Suppression

DROP INDEX [schéma.]nom_d'index [CASCADE [RESTRICT]

Où:
CASCADE : objets dépendants de l'index supprimés.
RESTRICT : index pas supprimé si des objets en dépendent. (par défaut).

=> Espace libéré ré-affecté au tablespace où index créé.

4 Optimisation des ordres SQL par utilisation des index

4.1 Effets de l'indexation

Adjonction d'1 index à 1 table :

- ralentit m-à-j (insertion, suppression)
- accélère beaucoup recherche des lignes.

- Recherche des lignes ayant une valeur d'index > , = ou < à une valeur donnée.

Exemple:

Bénéfice de index sur nopilot pour :

```
SELECT * FROM pilote WHERE nopilot = 7899
SELECT * FROM pilote WHERE nopilot >= 4000
SELECT * FROM pilote WHERE nopilot BETWEEN 3000 AND 5000
```

- Utilisation d'1 index bénéfique même si critère de recherche constitué seulement du début de clé

Exemple :

Bénéfice de index sur 'nom' pour :

```
SELECT * FROM pilote WHERE nom LIKE 'M%'
```

4.2 Valeurs 'NULL'

non stockées dans l'index pour minimiser volume.

=> index inutile pour retrouver valeurs nulles, si critère de recherche :
IS NULL ou IS NOT NULL.

Exemple :

Pour sélectionner les pilotes ayant 1 commission il vaut mieux utiliser :

```
SELECT * FROM pilote WHERE comm  >= 0
```

que :

```
SELECT * FROM pilote WHERE comm IS NOT NULL
```

avec index sur *comm*.

4.3 Conversion

Index non utilisé lors de l'évaluation d'1 requête si la (les) colonne(s) correspondante(s) sont dans expression, fct ou conversion implicite.

Exemples :

SELECT * FROM pilote WHERE sal * 12 > 1 0000 ;

=> pas d'effet de index sur *sal* car *sal* dans 1 expression.

écrire plutôt : SELECT * FROM pilote WHERE sal > 1 0000 / 12 ;

SELECT * FROM pilote WHERE TO_CHAR(date_embauche,'DD/MM/YY') ='01/01/93';

=> pas d'effet de index sur *date_embauche* car *date_embauche* dans 1 fonction.

écrire plutôt :
SELECT * FROM pilote WHERE date_embauche =
TO_DATE('01/01/93','DD/MM/YY');

=> bénéfice de l'index sur date_embauche.

4.4 Choix de l'indexation

Créer 1 index sur colonnes :

- définies comme clé primaire (INDEX UNIQUE);
- utilisées comme critère de jointure;
- servant souvent de critère de sélection (WHERE).
- intervenant dans un tri (ORDER BY)

Ne pas créer d'index sur colonnes :

- de tables de moins de 200-300 lignes
- $\supset$ peu de valeurs distinctes (index peu efficace);
- souvent modifiées;
- toujours utilisées par l'intermédiaire d'1 expression dans 1 clause WHERE.
- $\in$ fonction de groupe (SUM, AVG, …) sauf MIN et MAX
- intervenant sur un GROUP BY.

Les clusters

1 Généralités

CLUSTER = organisation physique ds 1 même bloc de disque, des lignes d'1 table selon 1 ou plusieurs colonnes (clé primaire et référence).

But : Accélérer opérations de jointure.

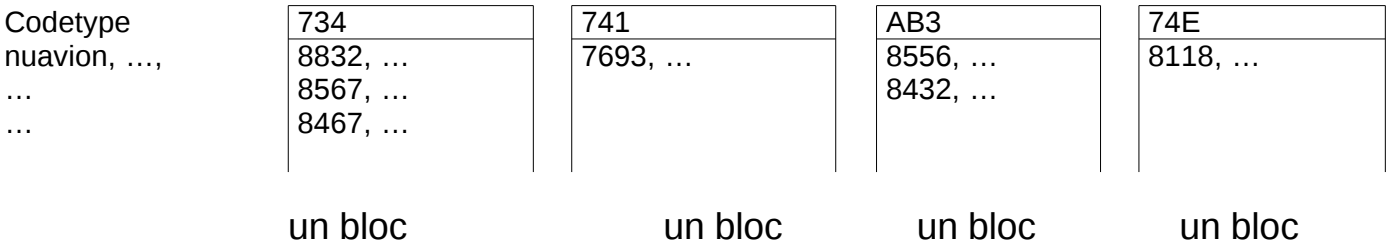
2 Clé de cluster

- constituée d'1 ou plusieurs colonnes de la table mise en cluster.
- valeur de la clé de cluster stockée 1 fois par bloc de données.
- m-a-j des valeurs des colonnes composant 1 clé de cluster

=> nécessité de relancer le clustering.

Exemple de cluster :

Cluster (indexé) avec table AVION et clé de cluster *codetype :*



3 Types de cluster

3.1 Cluster à index

- 1 index de cluster doit être créé sur la clé de cluster avant toute opération effectuée sur les tables du cluster.
- Il contient 1 entrée / valeur de clé de cluster.
- Index de cluster stockés dans segments index.

3.2 Cluster à hashage

- lignes des tables stockées et accédées par fct de hash.
- Fonction de hashacge appliquée à clé de cluster 1 valeur hash clé.
- clé de cluster définie sur 1colonne numérique avec valeurs uniformes

=> clé hash = valeur de colonne (fct hash inutile).

4 Gestion des clusters

4.1 Mise en cluster d'une table

4.1.1 1ère étape : Création de l'index

voir chapitre sur les indexes.

4.1.2 2ème étape : Utilisation de l'index

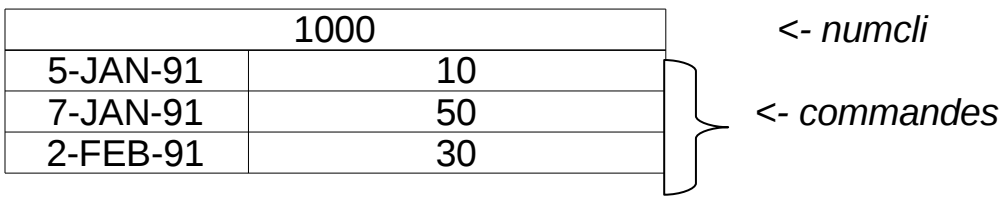
CLUSTER nom_index ON nom_table

Exemple :

```
CREATE TABLE commande
(
datecom    Date,
qtecom     NUMBER,
numcli     CHAR(4)
) ;
```

CREATE INDEX ind_co **ON** commande(numcli) ;

Représentation dans un bloc PostgreSQL :



Soit la requête suivante :

```
SELECT nomcli, datecom, qtecom
FROM client, commande
WHERE client.numcli = commande.numcli
 AND client.numcli = 1000;
```

=>

- Accès à l'index du cluster qui fournit le N° du bloc
- Lecture séquentielle du bloc pour avoir commandes du client N°1000.

4.2 Reorganisation du cluster

Mises à jour effectuées sur valeurs de la colonne du cluster => relancer le clustering :

CLUSTER nom_table

=> réorganisation de la table selon le même index.

4.3 Reorganisation de tous les clusters de la BD

Mises à jour effectuées sur valeurs de la colonne du cluster => relancer le clustering :

CLUSTER

=> réorganisation de toutes les tables selon leur même index.

Contrôle des accès

1 Objectifs

Répartir et sélectionner *droits* des uts de BD pour assurer protection des données de la BD.

Plusieurs niveaux :

- Gestion des accès à la BD;
- Gestion des accès aux données de la BD;
- Limitation des ressources accessibles aux uts;
- Attribution d'accès par défaut.

Protection décentralisée des objets de la base :

- tout objet a 1 ut. «créateur» possédant tous les droits (consultation, modif. et suppression) sur cet objet.
- aucun droit sur cet objet de la part des autres ut. non privilégiés

2 Principes

2.1 Utilisateur

Accès à BD par uttilisateur (compte PostgreSQL) défini par :

- nom d'utilisateur
- mot de passe (optionnel selon la configuration)
- ensemble de privilèges;

Avant l'installation de PostgreSQL, création d'1 super-utilisateur qui gèrera PostgreSQL :

$ su - -c "useradd postgres"

=> le super_utilisateur postgres est créé par le root sous UNIX.

Connexion avec ce compte => pleins pouvoirs de création et modification des BDs et utilisateurs.

2.2 Privilèges

2.2.1.Privilège au niveau global :

Privilèges attribués par un super-utilisateur à un utilisateur :

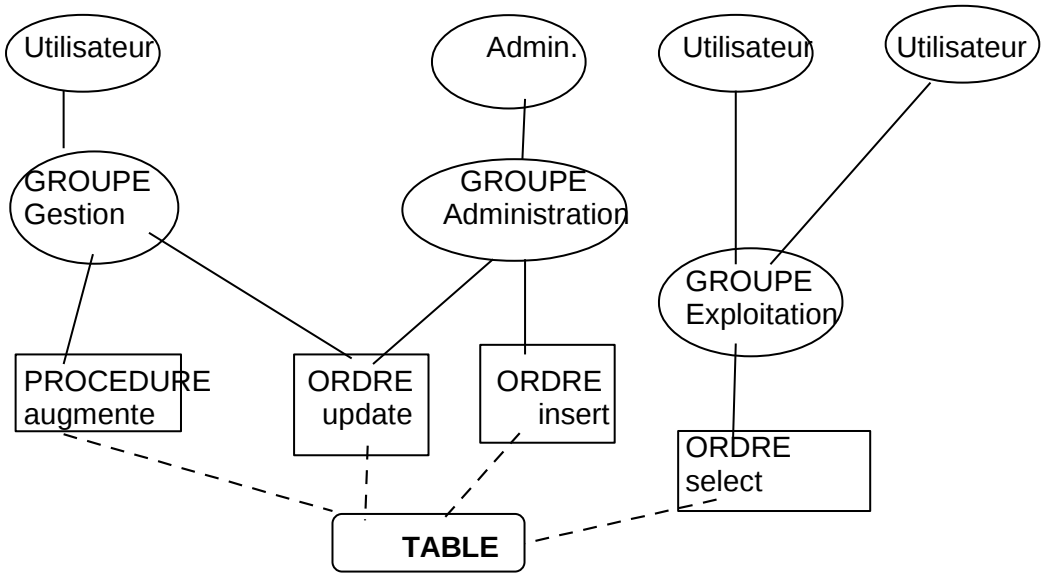
- autorisation de créer et de détruire des bases de données
- statut de super-utilisateur ou non

2.2.2 Privilège au niveau objet

= autorisation donnée par créateur d'1 objet particulier à d'autres uts.

2.3 Groupe

= regroupement d'utilisateurs ayant les mêmes privilèges attribués à ce groupe.



- identifié par nom de groupe
- attribué de privilèges système (globaux) et / ou privilèges objet.
- composé de 1 ou plusieurs utilisateurs

2.4 Gestion du serveur

Permet contrôler l'activité des bases de données en limitant les ressources accessibles :

- Nb max de blocs lus.
- Temps CPU.
- Nb max. connexions.

- Durée sans occupation autorisée.
- Nb tampons disques en mémoire partagée alloués pour le serveur.
- Répertoire des données des bases de données.
- Etc.

Si atteinte de limite de ressources lors de l'utilisation d'un BD :

alors selon la configuration du serveur :

- opération en cours arrêtée,

  ou
- transaction annulée,

  ou
- code d'erreur renvoyé

3 Gestion des utilisateurs

3.1 Introduction

Introduction d'1 nouvel ut. en 2 étapes :

1. Création de l'ut. par : CREATE USER
2. Allocation des privilèges par : GRANT et ALTER USER.

Modifier propriétés d'1 ut. par exécution :
- GRANT ou ALTER USER
- REVOKE.

Utilisateur particulier : PUBLIC créé à l'initialisation de la BD, représente tous les utilisateurs.

3.2 Création d'1 ut.

CREATE USER *nom_utilisateur*

[

SYSID uid

| [ENCRYPTED | UNENCRYPTED] PASSWORD '*password*'

| CREATEDB | NOCREATEDB

| CREATEUSER | NOCREATEUSER

| IN GROUP nom_*groupe* [, ...]

| VALID UNTIL '*expiration*'

]

où

nom_utilisateur

Nom du nouvel utilisateur.

uid

ID utilisateur explicite du nouvel utilisateur (si omis, fourni par PostgreSQL).

ENCRYPTED | UNENCRYPTED
mot de passe (si fourni) crypté ou non (par défaut).

password
mot de passe du nouvel utilisateur. Doit être fourni si BD configurée pour exiger une authentification par mot de passe. Sinon, inutile.

CREATEDB | NOCREATEDB
droit de créer des BDs (par défaut NOCREATEDB).

CREATEUSER | NOCREATEUSER
droit de créer des utilisateurs (par défaut NOCREATEUSER). => droit super-utilisateur !!!

nom_groupe
groupe auquel appartiendra l'utilisateur.

expiration
date d'expiration du mot de passe si donné.

3.3 Modification d'un ut.

ALTER USER *nom_utilisateur* [

| [ENCRYPTED | UNENCRYPTED] PASSWORD '*password*'

| CREATEDB | NOCREATEDB

| CREATEUSER | NOCREATEUSER

| VALID UNTIL '*expiration*'

]

Seul le mot de passe peut être modifié par un utilisateur ordinaire.

Tout peut être modifié par le super-utilisateur.

3.4 Suppression d'un ut.

DROP USER *nom_utilisateur* [CASCADE]

ut. non supprimé si propriétaire d'objet.

4 Gestion des groupes

4.1 Création d'un groupe

CREATE GROUP *nom_groupe* [
SYSID uid
| [USER |nom_utilisateur] [, ...]
]

où

nom_groupe

Nom du nouveau groupe.

uid

ID utilisateur explicite du nouveau groupe (si omis, fourni par PostgreSQL).

USER *nom_utilisateur*

utilisateur ou (liste des utilisateurs) à inclure dans le groupe. Doivent être déjà créés.

=> Utiliser GRANT pour affecter des privilèges ou des rôles à ce rôle.

4.2 Modification d'un groupe

ALTER GROUP *nom_groupe ADD USER nom_utilisateur [, ...]*

ou

ALTER GROUP *nom_groupe DROP USER nom_utilisateur [, ...]*

Permet au super-utilisateur d'ajouter ou de supprimer un utilisateur dans *nom_groupe*.

L'utilisateur ajouté doit exister. L'utilisateur supprimé du groupe n'est pas supprimé du système.

4.3 Suppression d'un groupe

DROP GROUP *nom_groupe*

Permet au super-utilisateur de supprimer le groupe *nom_groupe*.

Les utilisateurs appartenant à ce groupe ne sont pas supprimés du système.

5 Gestion des privilèges

5.1 Attribution de privilèges

Un ut. créateur d'un objet a tous les droits sur celui-ci

Les autres ut.s (sauf un super-utilisateur) n'ont aucun droit sur cet objet.

Le créateur peut donner des droits à :

- quelques ut.s,
- tous les ut.s par 1 seule commande PUBLIC.

Attribution d'un privilège :

GRANT { { SELECT | INSERT | UPDATE | DELETE | REFERENCES | TRIGGER }
[,...] | ALL [PRIVILEGES] }
ON [TABLE] nom_table [, ...]
TO { *nom_utilisateur* | GROUP *nom_groupe* | PUBLIC } [, ...]

GRANT { { CREATE | TEMPORARY | TEMP } [,...] | ALL [PRIVILEGES] }
ON DATABASE *nom_base* [, ...]
TO { *nom_utilisateur* | GROUP *nom_groupe* | PUBLIC } [, ...]

GRANT EXECUTE
ON FUNCTION *nom_fonction* ([type, ...]) [, ...]
TO { *nom_utilisateur* | GROUP *nom_groupe* | PUBLIC } [, ...]

GRANT USAGE
ON LANGUAGE *nom_language* [, ...]
TO { *nom_utilisateur* | GROUP *nom_groupe* | PUBLIC } [, ...]

GRANT { { CREATE | USAGE } [,...] | ALL [PRIVILEGES] }
ON SCHEMA *nom_schéma* [, ...]
TO { *nom_utilisateur* | GROUP *nom_groupe* | PUBLIC } [, ...]

Privilèges attribuables sur une table :

SELECT	Lecture de lignes
INSERT	Insertion lignes
UPDATE [(col, …)]	Modif. de lignes (éventuellement limité à certaines col.)
DELETE	Suppression de lignes
REFERENCES[(col, …)]	Référence à contraintes définies sur objet
TRIGGER	Créer un trigger sur la table
ALL	Tous les droits

Privilège attribuable sur une fonction :

EXECUTE	Exécuter fonction

Privilège attribuable sur un langage :

USAGE	Utiliser le langage pour créer une fction PL/pgSQL.

Privilèges attribuables sur un schéma :

CREATE	Créer des objets dans le schéma.
USAGE	Utiliser les objets du schéma (selon les droit mis sur chacun)
ALL	Tous les droits

Privilèges attribuables sur une BD :

CREATE	Créer des schémas dans la bd.
TEMP	Créer des tables temporaires dans la bd.
ALL	Tous les droits

Plusieurs droits accordés à plusieurs uts par 1 seul GRANT, ex :

GRANT SELECT, UPDATE ON EMP TO LEROI, DUVAL

Remarque:

Pour consulter la table EMP, DUVAL ou LEROI doit utiliser le nom hiérarchique complet de la table sous la forme *nom_schéma.nom_table* :

SELECT * FROM MARTIN.EMP

5.2 Suppression de privilèges

REVOKE { { SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES | TRIGGER }
[,...] | ALL [PRIVILEGES] }
ON [TABLE] *nom_table* [, ...]
FROM { *nom_utilisateur* | GROUP *nom_groupe* | PUBLIC } [, ...]

REVOKE { { CREATE | TEMPORARY | TEMP } [,...] | ALL [PRIVILEGES] }
ON DATABASE *nom_base*[, ...]
FROM { *nom_utilisateur* | GROUP *nom_groupe* | PUBLIC } [, ...]

REVOKE EXECUTE
ON FUNCTION *nom_fonction* ([type, ...]) [, ...]
FROM { *nom_utilisateur* | GROUP *nom_groupe* | PUBLIC } [, ...]

REVOKE USAGE
ON LANGUAGE *nom_langage* [, ...]
FROM { *nom_utilisateur* | GROUP *nom_groupe* | PUBLIC } [, ...]

REVOKE { { CREATE | USAGE } [,...] | ALL [PRIVILEGES] }
ON SCHEMA *nom_schéma* [, ...]
FROM { *nom_utilisateur* | GROUP *nom_groupe* | PUBLIC } [, ...]

6 Schéma

- Schéma ={ tables, vues, séquences, index, fonctions, droits}.
- Schéma ≈Espace de nom.
- Un ut. peut créer plusieurs schémas dans la BD s'il en a le droit.

  => Intérêt : avoir des objets de même nom mais organisés dans des schémas différents.
- Possibilité de Créer tables, vues et droits par 1 seule opération lors de la création du schéma.
- Si les objets n'appartiennent pas à un schéma spécifique :

  => accès par l'ut. à tous ses objets en spécifiant leur nom *nom_objet*.
- Si les objets appartiennent à un schéma *nom_schéma* :

  => accès par l'ut. à tous ses objets en spécifiant *nom_schéma*.*nom_objet*.

CREATE SCHEMA *nom_schéma* [AUTHORIZATION *nom_utilisateur*] [*schema_element* [...]]CREATE SCHEMA AUTHORIZATION *nom_utilisateur* [*schema_element* [...]]

où
schema_element
CREATE TABLE, CREATE VIEW ... , ou GRANT ....

AUTHORIZATION AY **AYTHOPIZATION**THOPIZATION
si omis, le schéma appartiendra à l'utilisateur de la commande. Sinon, le schéma appartiendra à *nom_utilisateur* (si le créateur en a le droit).

Exemples :

CREATE SCHEMA myschema; => schéma créé sous l'utilisateur en cours

CREATE SCHEMA AUTHORIZATION joe; => schéma créé sous l'utilisateur joe

```
CREATE SCHEMA titi
   CREATE TABLE pilote (nopilot  char(4) PRIMARY KEY, nom varchar(35), ....)
   CREATE VIEW v_pilote AS
      SELECT * FROM pilote WHERE comm IS NOT NULL;
```

Remarque : si 1 commande échoue, création de l'ensemble annulé.

1 Gestion des tables

1.1 Création d'une table

CREATE [TEMPORARY] TABLE [schéma.]*nom_table*
(*colonne type* [DEFAULT *expression*][*contrainte_de_colonne*]
[, colonne ...]
[c*ontrainte_de_table*])
[AS *requête*]

où :

contrainte_de_colonne
voir chapitre sur les contraintes.

contrainte_de_table
voir chapitre sur les contraintes.

TEMPORARY
Crée une table temporaire, détruite à la fin de la session. Toutes les constructions liées à cette table (index, contraintes, ...) seront également détruites.

1.2 Modification de la description d'une table

1.2.1 Ajout d'une colonne

ALTER TABLE *nom_table*
ADD *colonne type* [DEFAULT *expression*][*contrainte_de_colonne*]

valeur initiale des colonnes créées pour chaque ligne de la table : NULL.

1.2.2 Suppression d'une colonne

ALTER TABLE *nom_table*
DROP [COLUMN] *colonne* [RESTRICT | CASCADE]

où
CASCADE
supprimer toutes contraintes d'intégrité référentielles y faisant référence.

1.2.3 Modification de la description d'une colonne

ALTER TABLE *nom_table*
ALTER [COLUMN] *colonne* {SET DEFAULT *expression* | DROP DEFAULT] }

Ajoût ou suppression de la valeur par défaut.

ALTER TABLE *nom_table*
ALTER [COLUMN] *colonne* {SET | DROP } NOT NULL

Rendre une colonne obligatoire (à condition qu'il n'y ait pas déjà des valeurs nulles).

1.2.4 Modification du nom d'une colonne

ALTER TABLE *nom_table*
RENAME [COLUMN] *colonne* TO *nouv_nom*

1.2.5 Modification des contraintes d'intégrité

Voir chapitre sur les contraintes pour l'ajoût et la suppression d'une contrainte.

1.2.6 Modification du nom de la table

ALTER TABLE *nom_table*
RENAME TO *nouv_nom*

1.2.7 Modification du propiétaire de la table

ALTER TABLE *nom_table*
OWNER TO *nouv_owner*

1.3 Suppression d'une table

DROP TABLE *nom_table* [RESTRICT | CASCADE]

=> index et droits ("grant") sur table supprimés.

=> vues, triggers non supprimées mais invalides.

CASCADE : supprimer toutes contraintes d'intégrité référentielles à la clé primaire de la table.

2 Gestion d'une BD

2.1 Création d'une base

Installation de PostgeSQL => base par défaut *template1* modifiable ou création nouvelle BD :

CREATE DATABASE *nom_base*
[[WITH] [OWNER [=] *dbowner*] [LOCATION [=] '*dbpath*']
[TEMPLATE [=] *template*] [ENCODING [=] *encoding*]]

où

nom_base
nom de la nouvelle base.

dbowner
propriétaire de cette base. Si omis, le créateur est le proriétaire.

db_path
répertoire où stocker les données de la BD. Si omis, répertoire indiqué par le fichier de
config.

template
nom de la BD que l'on clône pour créer cette base. Par défaut, *template1*.

Exemple :

Création de la base de nom "gestair" sous UNIX avec fichier de données précisé :

```
mkdir private_db
initlocation ~/private_db

CREATE DATABASE gestair
WITH LOCATION  '/home/myriam/private_db';
```

2.2 Modification d'une base

ALTER DATABASE *nom_base* SET *paramètre* { TO | = } { *valeur* | DEFAULT }

ALTER DATABASE *nom_base* RESET *paramètre*

Modifie la valeur de session par défaut (initialisée la 1ère fois dans postgresql.conf).

ALTER DATABASE *nom_base* RENAME TO n*ouveau_nom*

Modifie le nom de la base.

2.3 Suppression d'une base

ALTER DATABASE *nom_base*

Dictionnaire de données

1 Dictionnaire de données

- contient : descriptions objets gérés par le SGBD.

- constitué de : tables gérées par noyau du SGBD en réponse à :
  CREATE objet ALTER objet DROP objet GRANT… REVOKE …

- créé lors de : initialisation de PostgresSQL.

- accessible par :
  - SGBD en Lecture / Ecriture.
  - uts en Lecture seule par intermédiaire de vues.

2 Principaux catalogues système

Interroger le catalogue en 2 étapes :

gestaire=#\d nom_catalogue
-> nom_col1, nom_col2 , ... nom des colonnes interessantes du catalogue

gestaire=#SELECT nom_col1, nom_col2, ...FROM nom_catalogue WHERE ... ;
-> infos sur le catalogue.

Nom du catalogue	Contenu
pg_aggregate	fonctions d'aggrégation
pg_am	méthodes d'accès aux index
pg_amop	opérateurs des méthodes d'accès
pg_amproc	procédures de support des méthodes d'accès
pg_attrdef	valeurs par défaut des colonnes
pg_attribute	colonnes des tables (<< attributs >>)
pg_cast	conversions de types de données (cast)
pg_class	tables, indexes, séquences (<< relations >>)
pg_constraint	contraintes de vérification, contraintes unique, contraintes de clés primaires, contraintes de clés étrangères
pg_conversion	informations de conversions d'encodage
pg_database	bases de données de l'installation PostgreSQL
pg_depend	dépendances entre objets de la base de données

Nom du catalogue	Contenu
pg_description	descriptions ou commentaires des objets de base de données
pg_group	groupes d'utilisateurs de la base de données
pg_index	informations supplémentaires des index
pg_inherits	hiérarchie d'héritage de tables
pg_language	langages pour écrire des fonctions
pg_largeobject	gros objets
pg_listener	support de notification asynchrone
pg_namespace	schémas
pg_opclass	classes d'opérateurs de méthodes d'accès aux index
pg_operator	opérateurs
pg_proc	fonctions et procédures
pg_rewrite	règles de réécriture de requêtes
pg_shadow	utilisateurs de la base de données
pg_statistic	statistiques de l'optimiseur de requêtes
pg_trigger	déclencheurs
pg_type	types de données

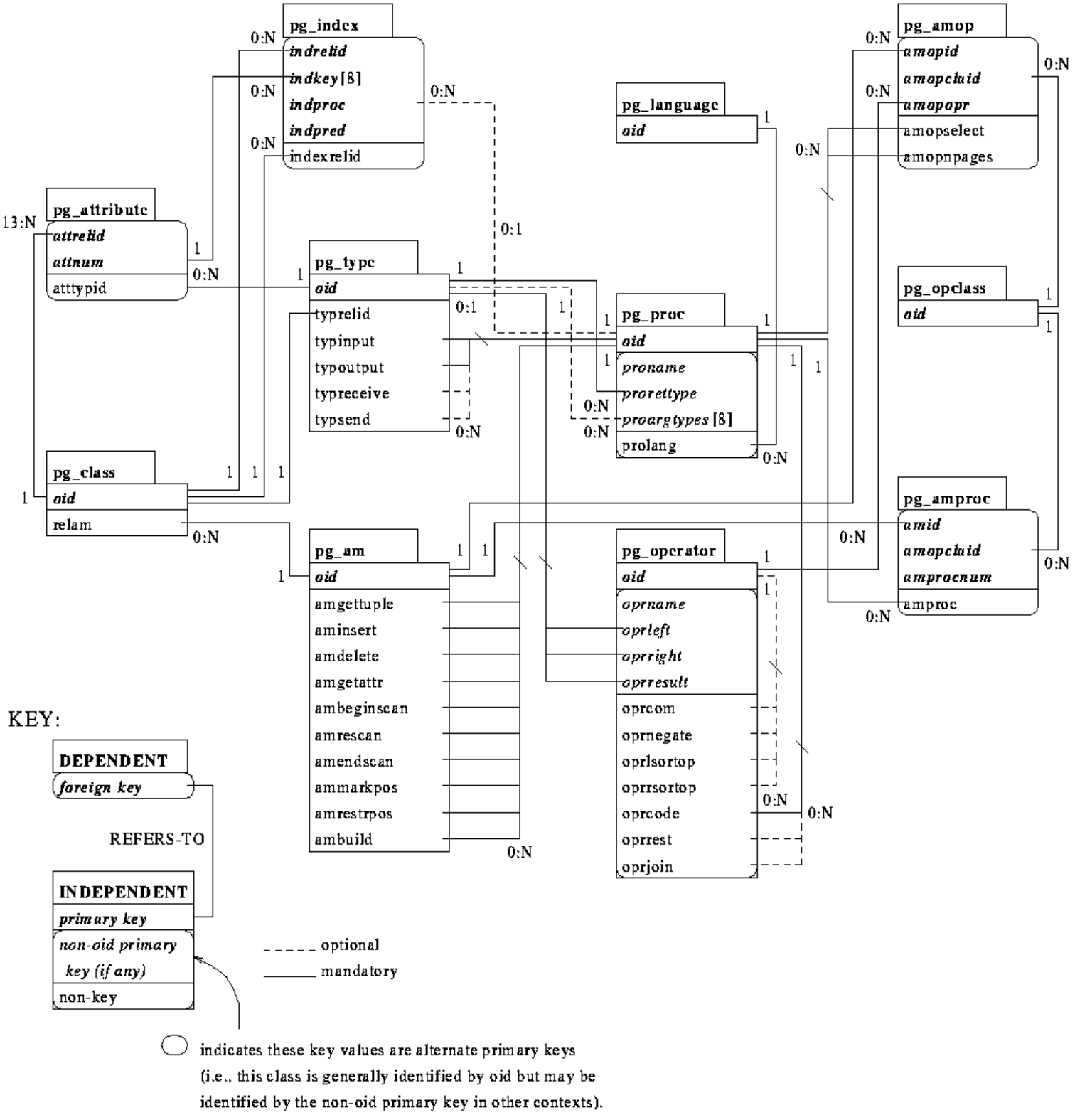


Figure 3. The major POSTGRES system catalogs.

Sauvegarde, Restauration, Chargement

1 Sauvegarde par l'outil pg_dump sous UNIX

but : copier la BD dans un endroit sûr (bande ou disque).

Sauvegarde complète ou partielle de la base exploitable par **pg_restore** :

- sauvegarde de qques tables, d'objets d'uts choisis, ou de toute la base
- sauvegarde incrémentale de base entière :

  sauvegarder que ce qui a été modifié depuis la dernière sauvegarde
- 1 seul fichier d'export contenant toutes les données de la base
- Réorganisation de la BD par le DBA (Database User administrateur).

Un ut. quelconque peut exporter :

- quelques unes de ses tables
- tous ses objets (tables, vues, séquences, clusters, index, contraintes d'intégrités, …)
- tous les objets qui lui sont accessibles (sur lesquels il doit posséder le droit SELECT).

Un DBA peut exporter :

- la base entière
- seulement les modif. faites à la base lors du dernier export
- ses propres objets
- les objets d'autres uts.

Syntaxe de la commande en ligne **pg_dump** :

pg_dump [OPTION]... [nom_base]

Options générales :

-f *nom*, --file=*nom* fichier de sortie
-F c|t|p, --format=c |t|p format du fichier de sortie : archivé et compressé (extension .tar.gz), archivé extension .tar) ou texte (par défaut).
-v, --verbose affiche un rapport détaillé des opérations
-Z 0-9, --compress=0-9 niveau de compression pour format compressé
--help affiche cette aide

Options contrôlant le contenu du fichier de sortie :

-a, --data-only sauvegarde des données, pas du schéma
-b, --blobs inclut les grands objets
-c, --clean insère d'abord le DROP avant chaque création d'objet
-C, --create insère d'abord le CREATE DATABASE
-d, --inserts insère des INSERT au lieu des COPY (insertion lente mais + sure)
-D, --column-inserts comme -d, avec en + la liste des colonnes avant VALUES
-o, --oids inclut les OIDs
-O, --no-owner n'inclut pas le propriétaire des objets (propriétaire des objets restaurés = utilisateur de la commande pg_restore)
-s, --schema-only sauvegarde du schéma, pas des données
-t *nom*, --table=*nom* seule sauvegarde de la table (par défaut toutes)
-x, --no-privileges ne sauvegarde pas les privilèges (grant/revoke)

Options de connexion :

-h *nom*, --host=*nom* nom hôte su serveur (si BD sur autre serveur)
-p PORT, --port=PORT n° de port du serveur (par défaut n° configuré à l'installation)
-U *nom*, --username=*nom* Connexion sous le nom utilisateur précisé
-W, --password demande un mot de passe

Si *nom_base* omis, la variable d'environnement PGDATABASE est utilisée.

Exemples :

$ pg_dump -f gestair_sauvegarde.sql gestair

$ pg_dump -F c -f gestair_sauvegarde.sql.tar.gz gestair

2 Sauvegarde par l'outil pg_dumplo sous UNIX

but : sauvegarde de données binaires (les LOBs, c-à-d les colonnes de type OID).

Syntaxe de la commande en ligne **pg_dumplo** :

pg_dumplo [OPTION]

Options :

-h --help	**Affiche cette aide**
-u *nom*, --username=*nom*	**Connexion sous le nom utilisateur précisé**
-p *password,* --password=*password*	**Password pour la connexion au serveur**
-d *nom_base,* --db=*nom_base*	**Nom de la base**
-t *nom_hote,* --host=*nom_hote*	**Nom de l'hôte**
-o *port,* --port=*port*	**N° de port du serveur (par défaut: 5432)**
-s *dir,* --space=*dir*	**Répertoire avec arboresence pour l'export/import**
-i --import	**Importation de LOBs dans la BD**
-e --export	**Exportation de LOBs (par défaut)**
-l *able.attr ...*,	**Inclure les noms de colonnes dans l'exportation**
-a --all	**Importation de tous les LOBs dans la BD (par défaut)**
-r --remove	**si -i, supprime d'abord les anciens LOBs de la BD**
-q --quiet	**Mode silencieux**
-w --show	**pas de dump, mais montre tous les LOBs de la BD.**

Si *nom_base* omis, la variable d'environnement PGDATABASE est utilisée.

Exemples :

$ pg_dumplo -d gestair -s /ma_sauvegarde/dir -l ob_appareil.plan ob_appareil.photo

=> ne sauvegarde que les colonnes *plan* et *photo* de la table ob_appareil de la bd *gestair* sous le répertoire */ma_sauvegarde/dir*.

$ pg_dumplo -i -d gestair -s /ma_sauvegarde/dir

=> importation dans *gestair* des lobs se trouvant sous le répertoire */ma_sauvegarde/dir*.

$ pg_dumplo -w -d gestair -s

=> affichage des lobs de la bd *gestair* .

3 Sauvegarde par l'outil pg_dumpall sous UNIX

but : sauvegarde de toutes les bases de données.

Syntaxe de la commande en ligne **pg_dumpall** :

pg_dumpall [OPTION]

Options :

-v, --verbose	**affiche un rapport détaillé des opérations**
--help	**affiche cette aide**
-c, --clean	**insère d'abord le DROP avant chaque création des bases et users**
-d, --inserts	**insère des INSERT au lieu des COPY (insertion lente mais + sure)**
-D, --column-inserts	**comme -d, avec en + la liste des colonnes avant VALUES**
-o, --oids	**inclut les OIDs**
-g, --globals-only	**seule sauvegarde des objets globaux (par ex : utilisateurs, groupes)**
-h *nom*, --host=*nom*	**nom hôte su serveur (si BDs sur autre serveur)**
-p PORT, --port=PORT	**n° de port du serveur (par défaut n° configuré à l'installation)**
-U *nom*, --username=*nom*	**Connexion sous le nom utilisateur précisé**
-W, --password	**demande un mot de passe**

Exemple :

$ pg_dumpall -c > toutes_les_bds.sql

=> sauvegarde de toutes les Bds dans le fichier toutes_les_bds.sql avec insertion des commandes DROP avant chaqy=ue création des Bds et users.

4 Restauration par l'outil pg_restore sous UNIX

but : remettre ds état cohérent BD suite à un incident.

pg_restore permet de restaurer à partir d'archives (autres qu'au format *texte***)** créées par pg_dump :

- des objets BD endommagés accidentellement

  ex : table supprimée par mégarde

  => la ré-introduire à partir d'1 sauvegarde de fichier export
  (restauration partielle de la base)

- une base entière jusqu'à date du dernier export de la bd (import incrémental).

Syntaxe de la commande en ligne **pg_restore** :

pg_restore [OPTION]... [FILE]

Option générales :

-d *nom*, --dbname=*nom* nom de la BD à restaurer. Si -C (création de BD), *nom* = template1.
-f *nom*, --file=*nom* fichier de sortie.
-F c | t , --format=c |t format du fichier d'entrée : archivé et compressé (extension .tar.gz), archivé (extension .tar).
-v, --verbose affiche un rapport détaillé des opérations.
--help affiche cette aide.

Options contrôlant le contenu du fichier de sortie :

-a, --data-only restauration des données, pas du schéma.
-c, --clean insère d'abord le DROP avant chaque création d'objet.
-C, --create insère d'abord le CREATE DATABASE.
-I *nom*, --index=*nom* seule restauration de l'index indiqué (par défaut tous).
-l, --list seule restauration de la table des matières (format TOC) des objets de la BD. Fichier résultat réutilisé par restore avec option -L.
-L *nom*, --use-list=*nom* utilise le fichier *nom* (format TOC) pour déterminer les objets à restaurer.
-N, --orig-order restauration selon l'ordre de sauvegarde originale (utilisée avec -L)
-o, --oids inclut les OIDs.
-O, --no-owner n'inclut pas le propriétaire des objets à restaurer.

-P *nom(args)*, --function=*nom(args)* seule restauration de la fonction indiquée (par défaut toutes).
-R, --no-reconnect ignore toutes les instructions \connect dans le fichier.
-s, --schema-only restauration du schéma, pas des données.
-S *nom*, --superuser=*nom* specifie the nom du super-utilisateur.
-t *nom*, --table=*nom* seule restauration de la table indiquée (par défaut toutes).
-T *nom*, --trigger=*nom* seule restauration de trigger indiqué (par défaut tous).
-x, --no-privileges ne restaure pas les privilèges (grant/revoke).
-X disable-triggers, --disable-triggers
désactive les triggers durant la restauration des données.

Options de connexion :

-h *nom*, --host=*nom* nom hôte su serveur (si BD sur autre serveur)
-p PORT, --port=PORT n° de port du serveur (par défaut n° configuré à l'installation)
-U *nom*, --username=*nom* Connexion sous le nom utilisateur précisé
-W, --password demande un mot de passe

Exemples :

$ pg_restore -v -C -O   -d template1 gestair.sql.tar

=> résultat :

Connecting to database for restore
Creating DATABASE gestair
Coonnecting to new DB 'gestair' as postgres
Connecting to gestair as postgres
Creating TABLE pilote
....

5 Chargement/sauvegarde d'une table par l'outil COPY sous SQL

pg_restore: chargement de données provenant d'1 BD PostgreSQL

COPY : chargement de données provenant de fichiers non PostgreSQL.

Fichiers de données au format :

- table binaire de PostgreSQL
- texte ASCII standard.

Enregistrements des fichiers de données textes au format :

- fixe (enregistrements même longueur et même structure)
- variable (champs délimités par un séparateur (, ou ; ou tabulation ....).

Copie entre un fichier et une table :

- chargement d'une table à partir d'un fichier (FROM)
- sauvegarde d'une table dans un fichier (TO).

5.1Syntaxe de la commande COPY sous SQL

```
COPY nom_table [ ( colonne [, ...] ) ]
    FROM { 'nom_fichier' | STDIN }
    [ [ WITH ]
          [ BINARY ]
          [ OIDS ]
          [ DELIMITER [ AS ] 'délimiteur' ]
          [ NULL [ AS ] 'chaîne' ] ]
```

ou

```
COPY nom_table [ ( colonne [, ...] ) ]
    TO { 'nom_fichier' | STDOUT }
    [ [ WITH ]
          [ BINARY ]
          [ OIDS ]
          [ DELIMITER [ AS ] 'délimiteur' ]
          [ NULL [ AS ] 'chaîne' ] ]
```

où :

nom_table
nom de la table.

colonne
Liste des colonnes à copier. Par défaut, toutes.

nom_fichier
Nom du fichier d'entrée ou de sortie.

STDIN
Les données proviennent de l'entrée standard (clavier).

STDOUT
Les données sont destinées à la sortie standard (écran).

BINARY
Fichier binaire. Les options DELIMITER ou NULL sont interdites.

OIDS
Copier aussi l'OID de chaque ligne.

délimiteur
Caractère délimiteur qui sépare les colonnes de chaque enregistrement (ligne). Par défaut '\t'.

chaîne
Chaîne représentant la valeur NULL. Par défaut '\N'.

Remarques :

COPY vérifie les contraintes et les triggers qui ont pu être placés.

COPY s'arrête dès qu'il rencontre une erreur.

Le chargement est exécuté en 1 seule transaction (meilleures performances qu'une succession de INSERT).

5.2 Exemples de programme de chargement

Soit la table à charger *appareil*

codetype	CHAR(3)	code d'1 famille d'avions
nbplace	NUMERIC(3)	nbre de places
design	VARCHAR(50)	nom de la famille d'avions

Soit le fichier de données *appareil.txt* se trouvant sous /tmp :

```
74E         150   BOEING 747-400 COMBI
AB3         180   AIRBUS A300
741         100   BOEING 747-100
SSC         80    CONCORDE
734         450   BOEING 737-400
```

Chargement des données dans la table *appareil* :

```
gestair=#copy appareil FROM '/tmp/appareil.txt' USING DELIMITERS '\t' ;
```

Exportation des données de la table *appareil* vers le fichier /tmp/appareilbis.txt avec séparateur ';' :

```
gestair=#copy appareil TO '/tmp/appareilbis.txt' USING DELIMITERS ';' ;
```

=> contenu du fichier appareilbis.txt :

```
74E;150;BOEING 747-400 COMBI
AB3;180;AIRBUS;A300
741;100;BOEING;747-100
SSC;80;CONCORDE
734;450;BOEING 737-400
```

- Deuxième Partie -

TABLE DES MATIERES

Thème **Page**

Thème **Page**

Le langage PL/pgSQL

1 Introduction

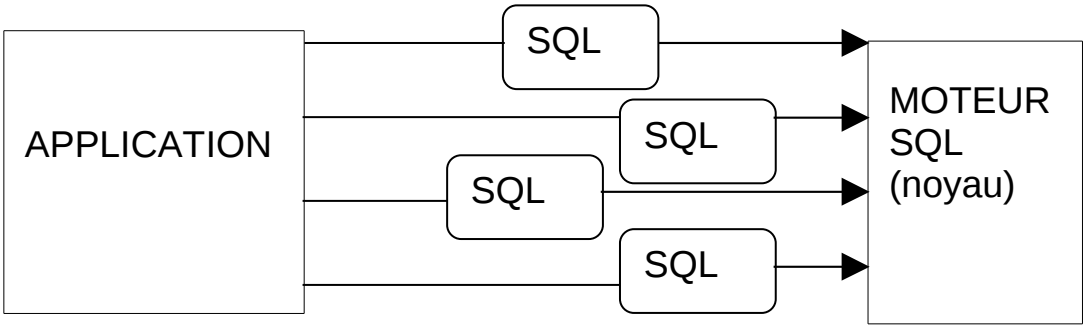
Langage PL/pgSQL (Procedural Language / PostgresSQL) ≡ extension du SQL pour :

- utilisation d'un sous-ensemble du langage SQL
- mise en œuvre de structures procédurales
- gestion des erreurs
- optimisation de l'exécution des requêtes.

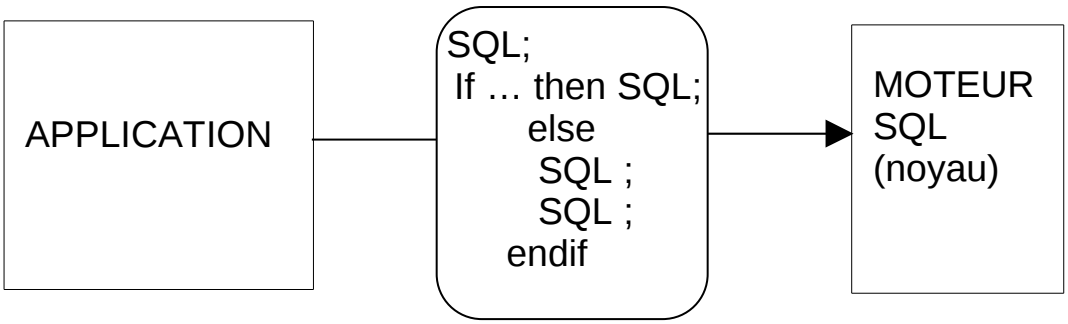
2 Bloc PL/pgSQL

2.1 Environnements SQL et PL/pgSQL

Environnement SQL : ordres du langage transmis au moteur SQL et exécutés les uns à la suite des autres :



Environnement PL/pgSQL : ordres SQL et PL/pgSQL regroupés en BLOCS (1 seul transfert vers moteur PL/pgSQL) :



2.2 Structure d'un bloc

DECLARE

Structures, variables locales au bloc, constantes, curseurs.

[section facultative]

BEGIN

Instructions PL/pgSQL et SQL.
Possibilité de blocs imbriqués.

[section obligatoire]

END;

2.3 Instruction

- instructions d'affectation,
- instructions SQL :
  - CLOSE
  - OPEN
  - FETCH
  - INSERT
  - DELETE
  - UPDATE
  - SELECT ... INTO
  - DROP
  - CREATE
  - ALTER
  - TO_CHAR, TO_DATE, UPPER, SUBSTR, ROUND, …
- instructions de contrôle (tests, boucles, séquences),
- instructions de gestion de curseurs,
- instructions de gestion des erreurs.

3 Gestion des données

Variables définies dans DECLARE pour stocker résultats de requêtes

3.1 Types de variable

3.1.1 Types scalaires

- Variable de type SQL :

  nom_variable nom_type

  nom_type ≡ CHAR, NUMERIC, DATE, VARCHAR, ...

- Variable par référence à 1 colonne d'1 table ou à 1 variable :

  nom_variable nom_table.nom_colonne%type.
  nom_variable nom_variable%type.

  Ex : v_nom pilote.nom%type;
  x NUMERIC(10,3);
  y x%type;

- Autres types scalaires définis

  nom_variable nom_type

  nom_type ≡ BOOLEAN, DECIMAL, FLOAT,
  INTEGER, REAL, SMALLINT, OID, …

3.1.2 Types composés

Enregistrement

2 façons pour déclarer une variable de ce type :

- par référence à une structure de table :

  nom_variable nom_table%ROWTYPE

- par référence au type RECORD (type non prédéfini ressemblant à ROWTYPE) :

  nom_variable RECORD;

  La variable *nom_variable* aura pour structure celle de la valeur qui lui sera assignée.

3.1.3 Variable et constante

- attribuer une valeur initiale à une variable au moment de sa déclaration :

  nom_variable type := valeur

- fixer une valeur constante :

  nom_variable CONSTANT type:= valeur

3.1.4 Variable et clause NOT NULL

- Indiquer qu'une variable ne peut pas prendre la valeur NULL :

  nom_variable NOT NULL type;

  L'affectation d'une valeur NULL à *nom_variable* provoquera une erreur d'exécution.

3.2 Visibilité d'une variable

bloc où elle est définie
blocs imbriqués dans le bloc de définition
(sauf si renommée dans un bloc interne)

3.3 Conversion de types

- explicites par utilisation de TO_DATE, TO_CHAR, TO_NUMBER …
- implicites par conversion automatique (voir langages C, C++, ...).

4 Affectation d'une valeur à une variable

- opérateur :=
- ordre FETCH (voir paragraphe sur curseur)
- option INTO de l'ordre SELECT

4.1 Opérateur d'affectation

4.1.1 Variable de type simple

nom_variable := valeur

Exemple :

```
DECLARE
 PI    NUMERIC (8,6);
BEGIN
 PI:= 3.14159;
END ;
```

4.1.2 Variable de type composé

Référencer 1 variable de type enregistrement :

nom_variable.nom_champ

Exemple - enregistrement

```
employe            pilote%ROWTYPE;
  …
employe.no_pilot :='DUPUY';    employe.sal:= 12345.00;
…
```

4.2 Valeur résultat d'une requête SELECT

SELECT liste d'expressions INTO liste de variables FROM ….

utilisable si 1 seul tuple retourné (sinon utiliser la notion de CURSEUR).

Exemple1 :

```
DECLARE
        u_nom    pilote.nom%type;
         u_sal      pilote.sal%type;

BEGIN
        SELECT     nom, sal    INTO u_nom, u_sal     FROM pilote
        WHERE nopilot ='7937';
END;
```

Exemple2 :

```
DECLARE
        employe            pilote%ROWTYPE;

BEGIN
        SELECT     *       INTO employe FROM pilote
        WHERE nopilot ='7937';
END;
```

5 Instructions de contrôle

5.1 Structure alternative: IF

5.1.1 1ère forme :

IF condition THEN instructions;
END IF;

Instructions exécutées si la condition est VRAIE.

Exemple:

```
IF v_no_pilot <> 0 THEN
   UPDATE pilote SET sal = v_sal WHERE nopilot= v_nopilot;
END IF;
```

5.1.2 2ème forme :

IF condition THEN instructions; ELSE instructions;
END IF;

Instructions suivant THEN exécutées si condition VRAI;
celles suivant ELSE exécutées si condition FAUSSE.

5.1.3 3ème forme

IF condition THEN instructions;
ELSIF condition THEN instructions; ELSE instructions;
END IF;

Permet d'imbriquer 2 structures alternatives.

Exemple:

```
IF number = 0 THEN
   result := "zero";
ELSIF number > 0 THEN
   result := "positif";
ELSIF number < 0 THEN
   result := "negatif";
ELSE
   -- la seule possibilité est que le nombre soit null
   result := "NULL";
END IF;
```

5.2 Structures répétitives

5.2.1 LOOP

Répète indéfiniment 1 séquence d'instructions.
Sortie de la boucle possible par exécution de EXIT.

```
LOOP
      instructions

END LOOP;
```

Exemple:

```
LOOP
  ……. ;
  IF condition THEN EXIT;
  END IF;
  ……. ;
END LOOP;
```

ou:

```
LOOP
  ..... ;
  EXIT WHEN condition;
  ..... ;
END LOOP;
```

5.2.2 FOR

```
FOR variable_indice IN [REVERSE] valeur_début .. valeur_fin
LOOP
      instructions;
END LOOP;
```

où

variable_indice : locale à la boucle non déclarée dans DECLARE
valeur_début et valeur_fin : variables locales précédemment déclarées et initialisées, ou constantes

Pas égal à 1

Incrémentation <0 ou >0 selon utilisation ou non de REVERSE.

5.2.3 WHILE

Répète les instructions tant que la condition a la valeur VRAI

```
WHILE condition
LOOP
        instructions;
END LOOP;
```

5.3 Condition

expression de condition simple ou composée
mêmes règles d'évaluation que pour C ou C++.

6 Curseur

6.1 Définition et utilisation

2 types de curseurs :

- Implicite (créé automatiquement lors de l'exécution de l'ordre SQL)

  ➔ ordres SELECT sous plsql

  ➔ ordres SELECT donnant **1** ligne résultat

  ➔ ordres UPDATE, INSERT et DELETE.

- Explicite (décrit et géré au niveau de la procédure)

  ➔ ordre SELECT donnant plusieurs lignes résultats

6.2 Gestion d'un curseur explicite (4 étapes)

- déclaration du curseur (section DECLARE)
- ouverture du curseur (section instructions)
- traitement des lignes (section instructions)
- fermeture du curseur (section instructions)

6.2.1 Déclaration du curseur

définir la requête SELECT et l'associer à un curseur.

nom curseur CURSOR IS requête

Curseur paramétré :

nom curseur CURSOR (nom_paramètre_formel type [:= valeur par défaut] [,…])
IS requête

Paramètres utilisés dans condition de sélection, expression, ou comme critère de la clause ORDER BY.

Exemples

```
DECLARE
  C1 CURSOR IS SELECT nom FROM pilote WHERE sal > 1 000;

  C2 CURSOR (psal numeric(7,2), pcom numeric(7,2)) IS
  SELECT ename FROM pilote WHERE sal > psal AND comm > pcom;
```

6.2.2 Ouverture du curseur

alloue un espace mémoire pour le curseur

OPEN nom_curseur

ou:

OPEN nom_Curseur (paramètres effectifs)

Chaque paramètre effectif associé à 1 paramètre formel occupant la même position dans la liste.

Exemples :

```
OPEN CI;
OPEN C2(12000,2500);
```

6.2.3 Fermeture du curseur

libère la place mémoire.

CLOSE nom_Curseur

6.2.4 Traitement des lignes

Lignes obtenues par exécution de la requête SQL distribuées 1 à 1 par exécution de FETCH inclus dans structure répétitive.

FETCH nom_curseur INTO liste_variables.

Exemple 1:

```
DECLARE
    C3 cursor IS SELECT nom, sal FROM pilote;
    v_nom       pilote.nom%type;
    v_sal       pilote.sal%type;

BEGIN
    OPEN C3;

    LOOP

     FETCH C3 INTO v_nom, v_sal;
     EXIT WHEN NOT FOUND;
     traitement;

    END LOOP;

    CLOSE C3;
END;
```

Forme syntaxique condensée

```
DECLARE
      nom_curseur CURSOR IS requête;

BEGIN
 FOR nom_enregistrement IN nom_curseur [(paramètres effectifs)]
 LOOP
      traitement;
 END LOOP;
END;
```

équivalente à :

```
DECLARE

        nom_curseur             CURSOR IS requête;
        nom_enregistrement      nom_curseur%ROWTYPE;

BEGIN

 OPEN nom_curseur;
 LOOP
        FETCH nom_curseur INTO nom_enregistrement;
        EXIT WHEN NOT FOUND;
        traitement;
 END LOOP;
 CLOSE nom_curseur;

END ;
```

Exemple 1:

```
DECLARE

        C4 CURSOR IS SELECT nom, sal from pilote;
        v_nom           pilote.nom%type;
        v_sal           pilote.sal%type;
        rec_C4          c4%type ;

BEGIN

 FOR rec_C4 IN C4
 LOOP
        v_nom:= rec_C4.nom;     /* visibilité de rec_C4 interne à la boucle */
        v_sal:= rec_C4.sal;
 END LOOP;

END;
```

équivalente à :

```
DECLARE

        v_nom           pilote.nom%type;
        v_sal           pilote.sal%type;
        rec_C4          record;

BEGIN

 FOR rec_C4 IN SELECT nom, sal from pilote
 LOOP
        v_nom:= rec_C4.nom;     /* visibilité de rec_C4 interne à la boucle */
        v_sal:= rec_C4.sal;
 END LOOP;

END;
```

Exemple 2 :

```
DECLARE

      i             pilote.%rowtype;

BEGIN

  FOR i   IN SELECT nopilot, sal, comm FROM pilote
              WHERE  embauche >TO_DATE('01-01-1993','dd-mm-yyyy')
  LOOP

      IF i.comm IS NULL
      THEN      DELETE FROM pilote WHERE  nopilot=i.nopilot;
      ELSIF i.comm > i.sal
            THEN      UPDATE   pilote
                      SET sal = i.sal + i.comm,  comm = 0
                      WHERE nopilot=i.nopilot;
      END IF

  END LOOP;
END;
```

7 Gestion des erreurs

7.1 Utilité

Affecter un traitement approprié aux erreurs lors de l'exécution d'1 bloc PL/pgSQL.

7.2 Syntaxe

Utiliser l'instruction RAISE pour rapporter des messages et lever des erreurs :

RAISE *niveau* '*format*' [, *variable* [, ...]];

où :

'*format*' [, *variable* [, ...]] :

Message de l'erreur contenant éventuellement des % remplacés par la valeur des *variables*.
Ecrire *%%* pour signifier un caractère *%*.
Les variables doivent être de simples variables, non des expression.

niveau :

={DEBUG, LOG, INFO, NOTICE, WARNING et EXCEPTION} .
EXCEPTION lève une erreur et interrompt la transaction courante.
Les autres niveaux ne font que générer des messages aux différents niveaux de priorité.

Exemple1 :

RAISE NOTICE ''Appel de f_cree_pilote(%)'', v_nopilot;

=> La valeur de *v_nopilot* remplacera le *%* dans le message affiché.

Exemple2 :

RAISE EXCEPTION 'nopilot --> % inexistant', v_nopilot;

=> la transaction est interrompue et le message d'erreur est affiché.

Exemple3 :

```
DECLARE

v_libelle            appareil.design%type;


BEGIN

 SELECT design INTO v_ libelle
  FROM appareil
  WHERE codetype = 'AB3';
  IF NOT FOUND
         THEN RAISE NOTICE 'AB3 :  type inconnu' ;
  END IF;

  END;
```

Exemple 4 :

```
CREATE TABLE erreur (lib1 VARCHAR(15), lib2 VARCHAR(50));

DECLARE
  y_nom              pilote.ename%type;
  v_sal              pilote.sal%type;
  v_comm             pilote.comm%type;

BEGIN

SELECT nom, sal,comm INTO v_nom, v_sal, v_comm
FROM pilote WHERE nopilot = '1254' ;

IF NOT FOUND THEN RAISE EXCEPTION  'pilote inconnu';
END IF;

IF v_sal < v_comm THEN RAISE NOTICE 'COMM > SAL';
ELSE RAISE NOTICE 'OK';
END IF;

END;
```

Fonctions stockées

1 Généralités

PostgreSQL n'utilise le langage PL/pgSQL qu'à travers des appels de fonctions stockées.
Fonction stockée = programme en PL/pgSQL effectuant un ensemble de traitements fréquemment utilisés.

Fonction appelée :

- En mode interactif par psql sous SQL ;
- Dans programme hôte utilisant des ordres SQL imbriqués (Préprocesseur C ECPG);
- Dans autres fonctions stockées;
- Dans déclencheurs.

2 Développement d'une fonction stockée

2.1 Création

2.1.1 Création d'une fonction

CREATE [OR REPLACE] FUNCTION nom_fonction
[(type [, type] …)] RETURNS type_retour
[IS | AS] '
bloc_PL/SQL
' LANGUAGE 'plpgsql';

avec :

nom_fonction	nom attribué à la fonction.
type	type de donnée du paramètre (définis en PL/pgSQL)
type_retour	type de la valeur retournée par la fonction.
bloc_PL/SQL	corps de la procédure ⊃ RETURN (variable_résultat) Les côtes « ' » dans le bloc doivent être doublées.

2.1.2 Fonction retournant le type void

Une fonction dont *type_retour* est *void* est une procédure.
Même si la fonction ne retourne rien, RETURN doit exister.

2.1.3 Alias de Paramètres de Fonctions

nom ALIAS FOR $n;

Paramètres passés aux fonctions nommés par les identifiants *$1*, *$2*, etc.

Possibilité de déclarer des alias pour les noms de paramètres pour améliorer la lisibilité.

Exemple :

```
CREATE FUNCTION sales_tax(real) RETURNS real AS '
BEGIN
    RETURN $1 * 0.06;
END;
' LANGUAGE 'plpgsql';
```

ou :

```
CREATE FUNCTION sales_tax(real) RETURNS real AS '
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
' LANGUAGE 'plpgsql';
```

2.1.4 Exemples

- Création d'1 nouveau pilote

```
CREATE OR REPLACE FUNCTION nv_pilote
  (pilote.nopilot%type, pilote.nom%type, pilote.adresse%type,
   pilote.sal%type, pilote.comm%type)  RETURNS void
   is '
    BEGIN
            INSERT INTO pilote VALUES ($1t, $2, $3, $4, $5);
            RETURN;
    END ;
    ' LANGUAGE 'plpgsql';
```

- Suppression d'1 pilote à partir de son n°

```
CREATE OR REPLACE FUNCTION del_pilote (pilote.nopilot%type)
RETURNS void
is '
DECLARE
  x_nopilot ALIAS FOR $1;
BEGIN
        DELETE FROM pilote WHERE nopilot = x_nopilot;
        RETURN;
END ;
'LANGUAGE 'plpgsql';
```

- Calcul du nbre moyen d'heures de vol des avions dont le code type est transmis en paramètre

```
CREATE OR REPLACE FUNCTION moy_h_vol
(appareil.codetype%type)
RETURNS NUMERIC
is '
DECLARE
  x_codetype ALIAS FOR $1;
  nbhvol_avg NUMERIC (8,2) := 0;  /*  transmettre la valeur résultat */

BEGIN
        SELECT AVG(nbhvol) INTO nbhvol_avg
        FROM avion
        WHERE type = x_codetype ;

        RETURN nbhvol_avg ;

END ;
'LANGUAGE 'plpgsql';
```

2.2 Compilation

- éditer fonction
- exécuter fichier sous psql pour le compiler
- erreur de syntaxe => Message : 'ERROR: parser: parse error at or near ... '

  => *STATUT INVALIDE*

. Appel d'1 fonction => consultation du statut par SGBD :

Si Statut INVALIDE alors :

- re-compilation par SGBD
- statut = VALID si aucune erreur
- exécution.

2.3 Suppression d'une fonction

DROP FUNCTION nom_fonction.

3 Appel aux fonctions

3.1 A partir d'un bloc PL/pgSQL

Si nom_fonction est une procédure :

PERFORM nom_fonction [(liste paramètres effectifs)]

Si nom_fonction est une fonction :

variable locale := nom_fonction [(liste paramètres effectifs)]
ou
SELECT nom_fonction [(liste paramètres effectifs)] INTO ... FROM ...
ou
SELECT ... INTO ... FROM ... WHERE ... nom_fonction [(liste paramètres effectifs)] ...
Exemples :

- Appel à la procédure *nv_pilote* pour créer 1 nouveau pilote

```
CREATE OR REPLACE FUNCTION appel_nv_pilote()
RETURNS varchar
is '
BEGIN
  PERFORM  nv_pilote(''1234'',''Brun'', ''Caen'',2400, 50);
  RETURN 'Terminé';
END;
'LANGUAGE 'plpgsql';
```

- Appel à la fonction *moy_h_vol* pour avions de famille AB3

```
CREATE OR REPLACE FUNCTION appel_moy_h_vol()
RETURNS varchar
is '
DECLARE
res NUMERIC;
BEGIN
   res:= moy_h_vol(''AB3'') ;
RAISE NOTICE ''Le nombre moyen d''heures de vol du type AB3 est %'',
res;
   RETURN 'Terminé';
END;
'LANGUAGE 'plpgsql';
```

3.2 A partir de psql

Si nom_fonction est une procédure :

Appel direct impossible. Appeler une fonction qui appelle cette procédure.

Si nom_fonction est une fonction :

SELECT nom_fonction [(liste paramètres effectifs)]
ou
SELECT nom_fonction [(liste paramètres effectifs)] FROM ... WHERE ...
ou
SELECT ... FROM ... WHERE ... nom_fonction [(liste paramètres effectifs)] ...

Exemples :

- Appel à procédure *nv_pilote* pour créer 1 nouveau pilote

SELECT **appel**_nv_pilote();

- Appel à fonction *moy_h_vol* pour avions de famille AB3

SELECT moy_h_vol('AB3');

3.3 A partir d'un programme hôte

Dans le langage du pré-processeur C ECPG, précéder référence au nom de fct par **EXEC SQL**.

Exemple :

Suppression d'1 pilote par une fonction hôte en C++ appelant la procédure *del_pilote*.

```
#include <string>

 EXEC SQL BEGIN DECLARE SECTION;
 string numero;
 EXEC SQL END DECLARE SECTION;

 int main()
 {
 EXEC SQL CONNECT TO gestair;
 cout << "\nEntrez le numéro d'employé : " ;
 cin >>  numero;
 EXEC SQL del_pilote( :numero);
 }
```

3.4 Appel à partir d'un autre schéma

préfixer nom fonction par nom schéma propriétaire.

Exemple : Appel de procédure *moy_h_vol* créée dans schéma MARTIN à partir de psql

```
SELECT martin.moy_h_vol('AB3');
```

4 Gestion des erreurs

Exemple 1:

Si pilote supprimé, vérifier que le pilote n'est affecté à aucun vol :

```
CREATE OR REPLACE  FUNCTION del_pilote
  (pilote.nopilot%type) RETURNS void
is '
```

```
DECLARE

 x_nopilot  ALIAS FOR $1;
 filler CHAR(1);

BEGIN

 SELECT "x" INTO filler FROM affectation WHERE pilote = x_nopilot;

 IF FOUND
 THEN RAISE NOTICE "Le pilote de numéro % est déjà affecté", x_nopilot ;
 ELSE DELETE FROM pilote WHERE nopilot = x_nopilot;
 END IF;

 RETURN;

END ;
 'LANGUAGE 'plpgsql';
```

Exemple 2 :

Vérifier si la commission est inférieure au salaire pour un n° de pilote donné :

```
CREATE OR REPLACE FUNCTION verif_comm
(pilote.nopilot%type) RETURNS void
is '

DECLARE

 x_nopilot  ALIAS FOR $1;
dif     pilote.sal%type := 0;

BEGIN
 SELECT sal - COALESCE(comm,0) INTO dif
 FROM pilote
 WHERE nopilot = x_nopilot;

 IF dif < 0
 THEN RAISE EXCEPTION "commission >  salaire" ;
 END IF;

 RETURN;
END ;
 'LANGUAGE 'plpgsql';
```

Les déclencheurs

1 Généralités

déclencheur ou trigger = *traitement* déclenché par un *événement.*

But : implémenter règles de gestion complexes

compléter règles d'intégrité référentielles.

Exemple: vérifier lors de chaque affectation d'1 avion à 1 vol que celui-ci n'est pas déjà utilisé par 1 autre affectation pendant la durée du vol.

2 Caractéristiques d'un déclencheur

- Traitement exprimé sous forme d'une procédure PL/pgSQL
- Associé à 1 et 1 seule table
- opérationnel jusqu'à la suppression de la table à laquelle il est lié
- Exécuté 1 fois (trigger par ordre) ou pour chaque ligne de la table (trigger ligne)
- Exécution
  - conditionnée par l'arrivée d'1 événement,
  - complétée éventuellement par 1 condition.

- Traitement $\supset$ ordres SQL agissant sur tables auxquelles sont associés des déclencheurs

déclenchement d'1 déclencheur => déclenchement d'autres déclencheurs en cascade.

3 Description d'un déclencheur

Déclencheur défini par :

- *Le séquencement*

  Exécution du traitement associé

  avant prise en compte de l'événement (**BEFORE**)
  *ou après (**AFTER**)*

- *Le ou les événements*

  Commandes SQL déclenchant 1 trigger lié à la table :

  INSERT, UPDATE, DELETE

- La table sur laquelle il agit

- Le *type*

  Traitement exécuté

  1 seule fois,
  ou pour chaque ligne concernée par l'événement (**FOR EACH ROW)**

- L' appel de la procédure trigger associée (déjà créée)

- *Des restrictions complémentaires éventuelles lors de la définition de la procédure*

  Si déclencheur activé par plusieurs événements :

  tester la variable spéciale TG_OP pour exécuter une séquence particulière du traitement en fct du type d'événement déclencheur.

  Exemple dans la définition de la procédure :

```
...
IF TG_OP="INSERT"  THEN … END IF;
IF TG_OP="DELETE" THEN … END IF.
IF TG_OP="UPDATE"        THEN … END IF;
...
END;
```

4 Variables spéciales accessibles dans la procédure trigger associée

NEW

Type de données RECORD; variable contenant la nouvelle ligne de base de données pour les opérations INSERT/UPDATE dans les déclencheurs de niveau ligne. Cette variable est null dans un trigger de niveau instruction (ordre).

OLD

Type de données RECORD; variable contenant l'ancienne ligne de base de données pour les opérations UPDATE/DELETE dans les triggers de niveau ligne. Cette variable est null dans les triggers de niveau instruction (ordre).

TG_NAME

Type de données name; variable qui contient le nom du trigger réellement lancé.

TG_WHEN

Type de données text; une chaîne, soit *BEFORE* soit *AFTER* selon la définition du déclencheur.

TG_LEVEL

Type de données text; une chaîne, soit *ROW* soit *STATEMENT* selon la définition du déclencheur.

TG_OP

Type de données text; une chaîne, *INSERT*, *UPDATE*, ou *DELETE* indiquant pour quelle opération le déclencheur a été lancé.

TG_RELID

Type de données oid; l'ID de l'objet de la table qui a causé l'invocation du trigger.

TG_RELNAME

Type de données name; le nom de la table qui a causé l'invocation du trigger.

5 Déclencheur par ordre

Exécuté 1 seule fois pour l'ensemble des lignes manipulées par l'événement.

5.1 Création

```
CREATE TRIGGER [schéma.]nom_déclencheur
séquence
événement [OR événement]
ON nom_table
EXECUTE PROCEDURE nom_procedure();
```

avec

séquence	BEFORE ou AFTER
événement	INSERT ou UPDATE ou DELETE
nom_table	nom de la table à laquelle le déclencheur est lié
nom_procedure	procédure trigger à exécuter déjà créée.

La procédure écrite en PL/pgSQL doit être de la forme :

```
CREATE FUNCTION nom_procedure() RETURNS TRIGGER AS '
 DECLARE
   ...
 BEGIN
    ...;
   RETURN NEW;
END;
' LANGUAGE 'plpgsql';
```

5.2 Utilisation

Exemple: Vérifier que le nbre moyen d'heures de vol des avions reste <= 200000.

```
DROP FUNCTION verif_nbhvol() CASCADE;   /* supprime aussi les triggers qui y font référence */

CREATE FUNCTION verif_nbhvol() RETURNS TRIGGER AS '
 BEGIN
 DECLARE
        v_avg_nbhvol     NUMERIC;
```

```
      BEGIN
        SELECT AVG(nbhvol) INTO v_avg_nbhvol FROM avion;
        IF v_avg_nbhvol > 200000
        THEN RAISE EXCEPTION ''Nbre moyen d'heures de vol : % trop élevé' ',
v_avg_nbhvol;
        END IF;
      END;
        RETURN NEW;
 END;
' LANGUAGE 'plpgsql';

CREATE TRIGGER trig_verif_nbhvol
AFTER UPDATE OR INSERT ON avion
EXECUTE PROCEDURE  verif_nbhvol();
```

Remarque : Le déclencheur ordre n'est pas encore implémenté sous PostgreSQL.

6 Déclencheur ligne

exécuté pour chacune des lignes manipulées par l'exécution de l'événement.

6.1 Création

CREATE TRIGGER [schéma.]nom_déclencheur
séquence
événement [OR événement]
FOR EACH ROW
ON nom_table
EXECUTE PROCEDURE nom_procedure();

6.2 Utilisation

déclencheur ligne avec option BEFORE :

effectuer traitements d'initialisation avant exécution des modifs sur table

Exemple: Comptabiliser dans une variable *nbmodif*, nbre de lignes de la table pilote manipulées par chaque accès en m-à-j.

```
CREATE FUNCTION audit_pilote() RETURNS TRIGGER AS '
 BEGIN
   nbmodif := nbmodif + 1;
   RETURN NEW;
 END;
' LANGUAGE 'plpgsql';

CREATE TRIGGER trig_audit_pilote
BEFORE INSERT OR UPDATE OR DELETE ON pilote
FOR EACH ROW
EXECUTE PROCEDURE  audit_pilote();
```

déclencheur ligne avec option AFTER :

propager modifs ou gérer historiques.

=> Souvent utilisé avec référence anciennes et/ou nouvelles valeurs colonnes

Référence à OLD, NEW ds corps du traitement :

valeur d'1 colonne avant modification **OLD**

valeur d'1 colonne après modification **NEW**

Selon ordre SQL en cause, valeur prise en compte :

	OLD	**NEW**
INSERT	NULL	valeur créée
DELETE	valeur avant suppression	NULL
UPDATE	valeur avant modification	valeur après modification

Exemple :

Pour chaque modif de la table PILOTE, garder ds la table valeur_pilote, un historique des lignes manipulées et le nom du trigger déclenché.

```
CREATE FUNCTION audit2_pilote() RETURNS TRIGGER AS '
BEGIN
   INSERT INTO valeur_pilote VALUES (current_date, TG_NAME, OLD.nopilot, OLD.nom,
  OLD.adresse,OLD.sal, OLD.comm);
 RETURN NEW;
END;
' LANGUAGE 'plpgsql';

CREATE TRIGGER trig_audit2_pilote
AFTER DELETE OR UPDATE
ON pilote
FOR EACH ROW
EXECUTE PROCEDURE audit2_pilote() ;
```

Rem : valeur_pilote est une table déjà créée devant contenir l'historique.

7 Gestion des déclencheurs

7.1 Création, exécution

- Exécution de CREATE TRIGGER -> stockage code source du déclencheur ds BD.

Vue PG_TRIGGER -> infos sur déclencheurs de BD.

- Création déclencheur permis si privilège sur table concernée :
  - GRANT CREATE TRIGGER ....

- Appel du déclencheur -> corps re-compilé, code exécutable non stocké ds BD.

7.2 Déclencheurs en cascade

- Exécution déclencheur ⊃ INSERT, DELETE, UPDATE

  => exécution autre déclencheur associé à table modifiée par actions

- Si déclencheur de type ligne :
  - aucun ordre ne doit accéder 1 table déjà utilisée en mode modification par 1 autre utilisateur.
  - aucun ordre ne doit modifier la valeur d'1 colonne déclarée avec PRIMARY KEY, UNIQUE KEY ou FOREIGN KEY.

7.3 Modification d'un déclencheur

ALTER TRIGGER *ancien_nom* ON *nom_table* RENAME TO *nouveau_nom*

7.4 Suppression d'un déclencheur

DROP TRIGGER *nom_déclencheur* [CASCADE]

CASCADE
Suppression aussi de la fonction trigger associée

SQL dynamique sous PL/pgSQL

1 But

- Exécuter des requêtes construites lors de l'exécution (inconnue à la compil.)
- Référencer un objet de la BD inexistant à la compilation
- Créer dynamiquement des blocs PL/pgSQL

2 Execute

Vérifie la syntaxe et exécute dynamiquement un ordre SQL.

EXECUTE chaîne_dynamique ;

avec

chaîne_dynamique : ordre SQL quelconque sauf SELECT

Exemples : Différentes possibilités de EXECUTE IMMEDIATE

```
DECLARE

  requete VARCHAR(200);
  vnocli NUMERIC :=8;
  vca NUMERIC(8,2) :=10000.54;
  vnom VARCHAR(10):='toto';
  nom_table VARCHAR(10) :='pilote';
  condition VARCHAR(50) :='sal > 10000';

BEGIN

  -- ordre LDD statique, EXECUTE inutile

 EXECUTE  'CREATE TABLE clients (nocli numeric(6), nom varchar(10), ca
numeric(8,2))';

-- ordre LDD dynamique

EXECUTE 'DROP TABLE '  || nom_table;
-- ordre LMD dynamique

requete:='INSERT INTO clients values(' || quote_literal(vnocli )|| ',' ||
quote_literal(vnom) || ',' || quote_literal(vca) || ')';
```

```
EXECUTE requete ;


-- ordre LMD dynamique

  EXECUTE 'DELETE FROM clients WHERE nocli=' || quote_literal(vnocli) ;


-- ordre LMD dynamique

EXECUTE  ' DELETE FROM ' || nom_table || ' WHERE ' || condition;


-- ordre LDD dynamique

requete:='CREATE OR REPLACE function ajout_client(clients.nom%type, clients.ca
%type) AS ''

BEGIN
  INSERT into clients (nocli, nom, ca) VALUES (nextval(''seq''), $1,$2 );
END;
'' LANGUAGE plpgsql';

EXECUTE requete;


-- ordre LDD dynamique

requete:='CREATE OR REPLACE function supp_client(clients.nocli%type) AS ''
  BEGIN
    execute ''DELETE FROM clients WHERE nocli='' ||  quote_literal($1);
  END;
''LANGUAGE plpgsql';

EXECUTE requete ;
```

3 OPEN FOR, FETCH et CLOSE

Pour traiter les requêtes dynamiques ramenant plusieurs 1 ou plusieurs lignes.

- **OPEN FOR (Ouvrir un curseur)**

  OPEN variable_curseur FOR EXECUTE requête_dynamique

- **FETCH (Récupérer une ligne)**

  FETCH variable_curseur INTO {variable, … | enregistrement}

- **CLOSE (Fermer un curseur)**

  CLOSE variable_curseur

Exemple: Sélection du nom et du salaire du pilote n° vnopilot

```
requete:='SELECT nom, sal FROM pilote  WHERE nopilot= ' || quote_literal (vnopilot);
open cur for EXECUTE requete
loop
   raise info 'nom=% salaire=%',cur.nom, cur.sal;
end loop;
```

Exemple: Sélection des n° de pilotes parisiens

```
DECLARE
  type CurTyp is ref cursor;
  cur CurTyp;  -- variable curseur
  vno pilote.nopilot%type;
  vadr pilote.adresse%type:='PARIS';
BEGIN
  open cur for
        EXECUTE 'Select nopilot from pilote where adresse=' || quote_literal (vadr)
  loop
    fetch cur into vno; --ramener la ligne suivante
    exit when NOT FOUND; --quitter la boucle si plus de lignes
    --traiter les informations
  end loop;
  close cur;
END;
```

Interface de programmation ECPG (SQL intégré au C)

1 Introduction

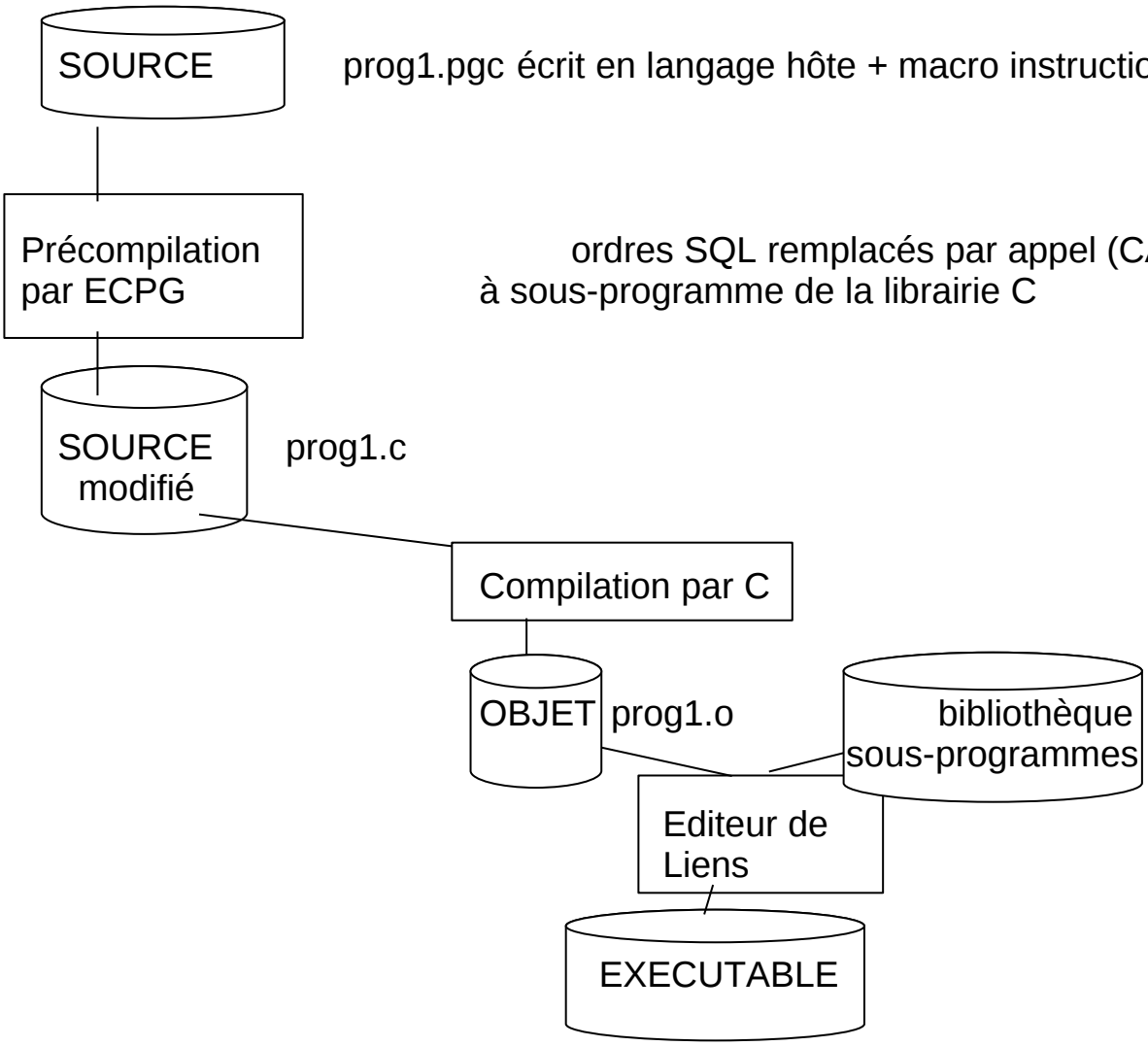
langage SQL : non procédural – Pas de :

- structure de contrôle
- gestion de variables

Solution : incorporer ordres SQL ds programmes langages procéduraux
(C, C++, Perl, PHP, Java, Python, Tcl/Tk, …)

Le pré-processeur ECPG (Embedded SQL C Preprocessor) permet d'écrire du SQL intégré au langage C.

Processus de développement :



2 Principes et définitions

Ordres SQL intégré (Embedded SQL)

= ordres SQL ds programme hôte écrit en langage ECPG :

Ordres interactifs	*Ordres non interactifs*
ALTER SESSION	CLOSE
ALTER	CONNECT
COMMENT	DESCRIBE
COMMIT	EXECUTE
CREATE	PREPARE
DELETE	FETCH
DROP	OPEN
GRANT	
INSERT	
RENAME	
REVOKE	
ROLLBACK	
SELECT	
TRUNCATE	
UPDATE	

Instruction de déclaration

- objets ;
- zones de communication et variables SQL pour transfert de données entre BD et ECPG.

Syntaxe d'un ordre SQL inclus

Délimité par *EXEC SQL* et par ;

Exemples :

```
EXEC SQL CREATE TABLE T(n integer, m char(10));
EXEC SQL CREATE UNIQUE INDEX idx1 ON T(n);
EXEC SQL INSERT INTO T values(999, 'toto');
EXEC SQL COMMIT;
```

Ordre SQL statique

Complètement défini dans le programme source hôte (comme précédemment).

Ordre SQL dynamique

Non connu au moment de la rédaction du programme source ou qui diffère partiellement d'1 exécution à 1 autre.

Variable hôte

Variables (de type scalaire ou tableau) déclarées ds programme hôte pour transfert données entre ordres SQL intégrés et programme hôte.

Utilisées comme :

- variables paramètre des ordres SQL
- zones de réception des données renvoyées par 1 interrogation (clause INTO de SELECT ou de FETCH).

Exemple :

EXEC SQL INSERT INTO T values(:x, :y);

Variable indicatrice

Associée éventuellement à variable hôte *x* pour compléter l'info véhiculée par *x*

Gestion des erreurs

Exécution d'1 ordre SQL intégré -> succès ou échec.

2 mécanismes de gestion :

- *Une zone de communication SQLCA*
  - incluse ds programme hôte
  - variables renseignées à la fin de l'exécution de chaque ordre SQL intégré
  - comptes-rendus et codes d'erreur fournis par variables.

- *Une instruction WHENEVER*

  - pour définir des actions à effectuer si certaines conditions prédéfinies se produisent lors de l'exécution d'1 ordre SQL intégré.

3 Variable hôte

3.1 Type simple

3.1.1 Déclaration

délimitée par :

EXEC SQL BEGIN DECLARE SECTION
et
EXEC SQL END DECLARE SECTION

Exemple :

```
EXEC SQL BEGIN DECLARE SECTION ;

int         pilote_numero;
char        pilote_nom [15];
int         pilote_salaire;
EXEC SQL END DECLARE SECTION;
```

Cas particulier : Type VARCHAR

- variables de communication pour chaînes caractères longueur variable :

  VARCHAR nom_variable [longueur]

- déclaration équivalente à (langage C) :

```
struct{
unsigned short int len;
unsigned char arr[longueur];
} nom_variable;
```

3.1.2 Utilisation

Variable hôte :

Exemples :

```
EXEC SQL INSERT INTO pilote VALUES (:n, :m, :s);
EXEC SQL SELECT nopilot, nom, sal
INTO :pilote_numéro, :pilote_nom, :pilote_salaire
FROM pilote;
```

Variable indicatrice :

Dans clause INTO de SELECT précédée de ":" immédiatement après variable hôte recevant valeur renvoyée par ordre SELECT.

Exemple :

Augmenter le salaire de 5% pour les pilotes qui n'ont pas de commission.

```
EXEC SQL BEGIN DECLARE SECTION;
int     pilote_numéro;
char  pilote_nom[15];
float  pilote_salaire;
float  pilote_commission;
smallint ind_commission;
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT nom, sal, comm
INTO :pilote_nom, :pilote_salaire, :pilote_commission, :ind_commission
FROM pilote
WHERE nopilot =:pilote_numero;

If (ind_commission == -1) {
        pilote_salaire = pilote_salaire * 1.05 ;
        ...
}
```

3.2 Type tableau

3.2.1 But

- Traiter requête *SELECT … INTO* ... renvoyant plusieurs lignes comme requête renvoyant 1 ligne.
- Distribuer plusieurs lignes à chaque exécution d'1 ordre *FETCH.*

3.2.2 Déclaration

```
EXEC SQL BEGIN DECLARE SECTION;

int     numéro [50];
float   montant [50];

EXEC SQL END DECLARE SECTION;
```

3.2.3 Utilisation

Avec SELECT *(ordre FETCH évité)*

```
EXEC SELECT nopilot, sal INTO  :numéro, :montant FROM pilote
WHERE     sal > 10000;
```

=> ensemble des pilotes dont le salaire est > 10000.

Avec INSERT *(insérer plusieurs lignes par 1 seul ordre)*

```
EXEC SQL INSERT INTO tablex VALUES (:numéro, :montant);
```

=> crée 50 lignes dans table TABLEX à partir des tableaux *numéro* et *montant*.

Avec UPDATE

Possible si tableau ds clause WHERE.

```
EXEC SQL UPDATE pilote SET sal=:montant WHERE nopilot =:numéro;
```

=> modifie la colonne *sal* par le tableau *montant* pour pilotes du tableau *numéro*.

Remarque

limiter nombre de lignes à utiliser ds tableau

-> ajouter avant ordre SQL ***FOR :variable hôte***

Exemple :

```
n = 5;
EXEC SQL FOR :n
UPDATE pilote SET sal =:montant
WHERE nopilot =:numéro;
```

=> limiter la m-à-j aux 5 premières lignes du tableau *montant*.

4 Connexion

1er ordre SQL intégré exécuté :

EXEC SQL CONNECT [TO :nom_base] [USER :nom_utilisateur
[IDENTIFIED BY :mot_passe]];

5 Déconnexion

EXEC SQL DISCONNECT [nom_base];

6 Accès à la base

6.1 Requête ne ramenant qu'1 seule ligne

Variables hôtes recevant valeurs colonnes spécifiées ds INTO … de SELECT.

Exemple:

```
EXEC SQL SELECT nom, sal INTO :pilote_nom, :pilote_sal  FROM pilote;
```

6.2 Requête pouvant ramener plusieurs lignes

6.2.1 utilisation de variables hôtes de type tableau

Voir paragraphe 3.2.

*6.2.2 utilisation d'1 structure CURSEUR (*PL/pgSQL)

- Déclaration

EXEC SQL DECLARE nom_curseur CURSOR FOR ordre_SQL;

- Ouverture curseur

EXEC SQL OPEN nom_curseur ;

- Fermeture curseur

EXEC SQL CLOSE nom_curseur ;

- Distribution d'1 ligne

EXEC SQL FETCH nom_curseur INTO liste variables hôtes ;

Plusieurs lignes :

```
for( ; ;)
{
 EXEC SQL WHENEVER NOT FOUND DO.....;
 EXEC SQL FETCH C1 INTO :pilote_nom, :pilote_sal;
...
 }
```

7 Gestion des transactions

Remarque : par défaut, le mode “autocommit” est OFF sous ECPG.

- Validation

EXEC SQL COMMIT ;

- Annulation

EXEC SQL ROLLBACK ;

à condition de ne pas avoir fait de *commit* avant, ou que le mode pas défaut ne soit pas autocommit ON.

- Positionnement du commit par défaut

EXEC SQL SET AUTOCOMMIT TO {ON | OFF} ;

8 Contrôle des transferts

8.1 Variables indicatrices

associée à :

- *Requête:* comportement requête (détection anomalies) indiquée par valeur :

 0 valeur renvoyée non nulle stockée ds variable hôte associée.

-1 aucune valeur renvoyée, variable hôte non modifiée.

> 0 valeur renvoyée tronquée stockée ds variable hôte

Exemple :

(voir paragraphe 3.1.2)

- *Opération de m-à-j* : attribuer valeur NULL à 1 colonne par exécution de INSERT ou UPDATE (variable indicatrice = -1).

Exemple :

```
icomm = -1;
EXEC SQL INSERT INTO pilote
VALUES(:no, :nom, :salaire, :commission:icomm);
```

=> ne pas enregistrer de valeur pour colonne *comm*.

Remarque : interdit d'écrire

```
EXEC SQL SELECT nom, sal INTO :pilote_nom, :pilore_sal
            FROM pilote WHERE comm= :commission:ind_comm ;
```

permis d'écrire

```
EXEC SQL SELECT nom, sal INTO :pilote_nom, :pilore_sal
            FROM pilote WHERE comm IS NULL;
```

8.2 Zone de communication

- Incorporée en début de programme hôte par :

EXEC SQL INCLUDE SQLCA

- Fin d'exécution de ordre SQL

=> zone SQLCA contient les comptes-rendus et codes d'erreur éventuels :

- **SQLCODE**

donne status de l'ordre SQL :

0 : exécution normale >0 : avertissement (warning) <0 : erreur fatale

- **SQLERRM**

2 champs, donne infos complémentaires si erreur ou d'avertissement :

sqlerrml largeur du texte.
sqlerrmc texte du message d'erreur.

- **SQLERRD**

tableau de 6 éléments, dont :

sqlerrd[2] nb lignes traitées par ordre SQL si SQLCODE=0.
sqlerrd[4] position erreur ds texte de ordre SQL.

- **SQLWARN**

tableau de 8 éléments :

sqlwarn [0] positionné si 1 des autres éléments est utilisé.

sqlwarn [1]	positionné si valeur d'1 colonne tronquée lors transfert ds zone de réception.
sqlwarn [3]	positionné si nb expressions ramenées par SELECT ou FETCH ≠ nb champs de réception de clause INTO.
sqlwarn [4]	positionné si UPDATE ou DELETE sans WHERE.
sqlwarn [5]	positionné si EXEC SQL CREATE FUNCTION échoué.

8.3 Traitements des erreurs

- Utilisation infos de zone SQLCA ;

Exemple :

```
....
EXEC SQL CONNECT TO gestair USER ...;
if (sqlca.sqlcode)
{
        printf("Erreur de connexion à la BD\n");
        printf("%i %s \n", sqlca.sqlcode, sqlca.sqlerrm.sqlerrmc);
        exit(0);
}
```

- Utilisation *WHENEVER* pour déclarer actions effectuées en fct d'événements:

EXEC SQL WHENEVER <événement> <action>

où événement :

SQLERROR présence d'1 erreur (sqlerror <0)

SQLWARNING présence d'1 anomalie indiquée ds zones sqlwarn[1] à sqlwarn[7]

NOT FOUND détection de fin de distribution de lignes pour FETCH.

où action :

STOP arrêt exécution programme, transaction en cours annulée

BREAK arrêt exécution des instructions dans les boucles et *switch*

CONTINUE effet de WHENEVER neutralisé

GO TO *label* branchement à étiquette spécifiée

DO nom_fonction appel de la fonction spécifée

SQLPRINT affichage d'un message d'erreur (prédéfini par ECPG) sur stderr.

Exemple :

```
EXEC SQL WHENEVER SQLERROR DO sqlerro();
….
void sqlerro( )
 {
        EXEC SQL WHENEVER SQLERROR CONTINUE;
 }
```

9 Fichiers d'inclusion

EXEC SQL INCLUDE nom_fichier

si nom_fichier contient des instructions SQL intégrées au C.

#include nom_fichier

si nom_fichier ne contient pas d'instructions SQL intégrées au C.

10 Ordres SQL dynamique

10.1 Principe

- Exécuter ordres SQL définis au moment de l'exécution du programme;
- Utile si :
  - Texte ordre SQL non connu lors de la rédaction du programme
  - Nombre variable de variables hôtes
  - Type des données variable.
- Plusieurs cas :
  - Exécution dynamique ordre SQL autre que SELECT, sans variable hôte de paramétrage;
  - Exécution dynamique ordre SQL autre que SELECT, avec variable hôte de paramétrage;
  - Ordre SELECT dont le nombre et type des colonnes transférées connus à l'avance;
  - Ordre SQL entièrement dynamique.

10.2 Ordre SQL autre que SELECT, sans variable hôte de paramétrage

Ordre SQL contenu ds variable hôte ou transmis sous chaîne de caractères :

EXEC SQL EXECUTE IMMEDIATE [:variable_hôte l 'chaîne']

Exemple:

```
EXEC SQL EXECUTE IMMEDIATE 'DELETE FROM pilote';
ou :

EXEC SQL BEGIN DECLARE SECTION;
  VARCHAR          v_requête[70];
EXEC SQL END DECLARE SECTION;
…
strcpy(v_requête.arr,"DELETE FROM pilote");
v_requête.len= strlen(v_requête.arr);

EXEC SQL EXECUTE IMMEDIATE :v_requête;
```

10.3 Ordre SQL autre que SELECT, avec variable hôte de paramétrage

Type et nombre de variables hôtes connus à l'avance. Traitement en 2 temps :

10.3.1 Préparation

Attribuer un *nom d'ordre* à l'ordre SQL défini par FROM et le compiler :

EXEC SQL PREPARE nom_ordre FROM [:variable_hôte l 'chaîne']

10.3.2 Exécution

Exécuter ordre SQL en utilisant les valeurs des variables hôtes paramètres :

EXECUTE nom_ordre USING liste de valeurs

Exemple:

```
EXEC SQL BEGIN DECLARE SECTION;
  VARCHAR        v_chaine[150], ville[351];
   int  montant;
EXEC SQL END DECLARE SECTION;

strcpy(v_chaine.arr,"DELETE FROM pilote WHERE adresse=:v1 AND sal<:v2");
v_chaine.len= strlen(v_chaine.arr);

EXEC SQL PREPARE delpilote FROM :v_chaîne;

ville ='LYON';   montant = 12000;

EXEC SQL EXECUTE delpilote USING :ville, :montant;
```

10.4 Ordre SELECT avec ou sans variable hôte de paramétrage

- attributs projetés et paramètres de condition de sélection connus à l'avance.
- variables correspondantes définies ds section DECLARE.

10.4.1 Préparation

EXEC SQL PREPARE nom_ordre FROM {:variable_hôte l 'chaine'}

Exemple :

```
EXEC SQL BEGIN DECLARE SECTION;

  VARCHAR select_stmt[150], nom[25];
   int            mt_salaire;

EXEC SQL END DECLARE SECTION;

strcpy(select_stmt.arr,"SELECT nom FROM pilote WHERE sal <:v1");
select_stmt.len = strlen(select_stmt.arr);

EXEC SQL PREPARE sql_req FROM :select_stmt;
```

10.4.2 Déclaration d'un curseur associé

EXEC SQL DECLARE nom_curseur CURSOR FOR nom_ordre

Exemple : EXEC SQL PREPARE pilote_cr CURSOR FOR sql_req;

10.4.3 Ouverture du curseur

EXEC SQL OPEN nom_curseur [USING liste variables_paramètres]

Exemple: EXEC SQL OPEN pilote_cr USING :mt_salaire

10.4.4 Distribution des lignes ramenées par l'exécution d'une requête

EXEC SQL FETCH nom_curseur INTO liste variables_sélection

Exemple: EXEC SQL FETCH pilote_cr INTO :nom;

10.4.5 Fermeture du curseur

EXEC SQL CLOSE nom_curseur

Exemple: EXEC SQL CLOSE pilote_cr;

10.5 Ordre SQL entièrement dynamique

- Attributs à projeter et/ou variables paramètres inconnus au moment de la rédaction du programme.
- Problème à résoudre : définition à l'exécution des
  - éléments à projeter
  - variables paramètres.
- Infos contenus dans la **SQLDA (SQL Description Area)** = zone de communication dynamique.
- PostgresSQL permet l'exécution d'ordre SQL entièrement dynamique. Voir la documentation (très peu fournie).

Interface de programmation LIBPQ (C natif)

1. Introduction

PostgreSQL : écrit en langage C.

libpq : interface de programmation C, base pour d'autres interfaces comme libpq++ (C++), libpgtch (Tcl), Perl ou ECPG.

2. Fonctions de Connexion

2.1. PQconnectdb

PGconn ***PQconnectdb**(const char *conninfo);

Ouvre une connexion au serveur de BD. Les paramètres se trouvent dans *conninfo sous la forme* "[*paramètre= valeur]* [*paramètre= valeur]* ...."

où *paramètre* :

host

Nom de l'hôte.

hostaddr

Adresse IP de l'hôte. (ex :*172.28.40.9).*

port

N° de port.

dbname

Nom de la BD. Par défault, *dbname* est égale au nom *user*.

user

Nom de l'utilisateur.

password

Mot de passe si le serveur le demande.

connect_timeout

Durée maximal d'attente de connexion (en secondes).

Si *conninfo* est vide, les paramètres par défaut sont utilisés.

2.2. PQsetdbLogin

PGconn ***PQsetdbLogin**(const char *pghost,
const char *pgport,
const char *pgoptions,
const char *pgtty,
const char *dbName,
const char *login,
const char *pwd);

Etablit une nouvelle connexion au serveur de BD. Pareil à Pqconnectdb. Mettre NULL ou "" pour les paramètres par défaut.

2.3. PQfinish

void **PQfinish**(PGconn *conn);

Ferme la connexion au serveur. Libère la mémoire utilisée par l'objet PGconn. A appeler même si la connexion n'a pu être établie.

2.4. PQreset

void **PQreset**(PGconn *conn);

Reinitialise la connexion au serveur (fermeture puis ouverture). Utile si connexion perdue.

3. Fonctions sur le statut d'une connexion

Fonctions retournant les valeurs des paramètres de la connexion établie :

3.1. PQdb

char ***PQdb**(const PGconn *conn);

Retoune le nom de la BD connectée.

3.2. PQuser

char ***PQuser**(const PGconn *conn);

Retourne le nom de l'utilisateur connnecté.

3.3. PQpass

char ***PQpass**(const PGconn *conn);

Retourne le mot de passe de l'utilisateur connnecté.

3.4. PQhost

char ***PQhost**(const PGconn *conn);

Retourne le nom du serveur hôte de la connexion.

3.5. PQport

char ***PQport**(const PGconn *conn);

Retourne le n° de port de la connexion.

3.6. PQoptions

char ***PQoptions**(const PGconn *conn);

Retourne les options de la ligne de commande (*conninfo*) passée en paramètre lors de l'appel de **Pqconnectdb.**

Fonctions retournant le statut de la connexion :

3.7. PQstatus

ConnStatusType **PQstatus**(const PGconn *conn);

Retourne :

CONNECTION_OK : si la connexion a été établie.

CONNECTION_BAD : sinon.

3.8. PQparameter Status

const char ***PQparameterStatus**(const PGconn *conn, const char *paramName);

Retourne le statut des paramètres *is_superuser*, *session_authorization*, et *DateStyle,* positionnés par le serveur lors de la connexion.

3.9. PQerrorMessage

char ***PQerrorMessage**(const PGconn* conn);

Retourne le message d'erreur généré par la dernière opération sous la connexion.

4. Fonctions d'exécution

Une fois la connexion établie, appeler les fonctions suivantes pour l'exécution des requêtes et commandes.

4.1. Fonctions principales

PQexec

Soumets une ou plusieurs commandes SQL au serveur et attends le résultat.

PGresult ***PQexec**(PGconn *conn, const char *command);

Retourne NULL si problème mémoire ou problème Serveur (utiliser PQerrorMessage pour obtenir des détails). Sinon, retourne un pointeur sur le résultat.

Si plusieurs commandes (séparées par ';' dans la chaîne), seul le résultat de la dernière est retourné.

PQexecParams

Comme **PQexec** mais admet 1 seule commande paramétrée.

PGresult ***PQexecParams**(PGconn *conn,
const char *command,
int nParams,
const Oid *paramTypes,
const char * const *paramValues,
const int *paramLengths, const int *paramFormats,
int resultFormat);

Les paramètres sont référencés par $1, $2 dans la commande.

command : chaîne constante contenant la commande.

nParams : nombre de paramètres

paramTypes[] : type de chacun des paramètres (si NULL, type à déduire)

paramValues[] : valeur de chacun des paramètres

paramLengths[] : longueur de la valeur de chacun des paramètres au format binaire (0 pour paramètres texte). Si NULL, pas de paramètres au format binaire)

paramFormats[] : format de la valeur de chacun des paramètres (0 pour format texte, 1 sinon). Si NULL, pas de paramètres au format binaire).

resultFormat : 0 pour obtenir résultats format texte, 1 sinon.

Exemple :

```
const char * valeurs[2];

valeurs[0]="1234";  valeurs[1]= "1000";

res = PQexecParams(conn,"SELECT nom  FROM pilote WHERE nopilot = $1
AND sal=$2",
                   2,          /* 2 paramètres */
                   NULL,       /* type à déduire*/
                   valeurs,
                   NULL,       /* longueurs inutiles car paramètres tous type texte */
                   NULL,       /* formats inutiles car paramètres tous type texte  */
                   1);         /* résultat binaire */
```

PQexecPrepared

Comme **PQexecParams** mais la chaîne *command* est une variable.

PGresult ***PQexecPrepared**(PGconn *conn,
const char *stmtName, int nParams,
const char * const *paramValues, const int *paramLengths,
const int *paramFormats, int resultFormat);

La commande *stmtName* est analysée et préparée une seule fois.
Performances améliorées si exécution répétée de cette commande.

PQresultStatus

Retourne le statut du résulat d'une commande.

ExecStatusType **PQresultStatus**(const PGresult *res);

Valeur retournée :

PGRES_EMPTY_QUERY

La requête était vide !!.

PGRES_COMMAND_OK

Exécution OK. (pour commandes autres que SELECT et SHOW).

PGRES_TUPLES_OK

Exécution OK. (pour commandes SELECT ou SHOW).

PGRES_NONFATAL_ERROR

Une erreur non fatale (info, notice ou warning) a été générée.

PGRES_FATAL_ERROR

Une erreur fatale a été générée.

PQresStatus

Convertit le type énuméré retourné par *PQresultStatus* en une chaîne.

char ***PQresStatus**(ExecStatusType status);

PQresultErrorMessage

Retourne le message d'erreur éventuelle associé à la commande.

char ***PQresultErrorMessage**(const PGresult *res);

PQresultErrorField

Retourne un message précis d'erreur éventuelle associé à la commande

char ***PQresultErrorField**(const PGresult *res, int fieldcode);

où *fieldcode* : PG_DIAG_SEVERITY

-> Niveau (*ERROR*, *FATAL*, *PANIC, WARNING*, *NOTICE*, *DEBUG*, *INFO, LOG).*

PG_DIAG_SQLSTATE

Etat (code) de l'erreur.

PG_DIAG_MESSAGE_PRIMARY

Message résumé de l'erreur (1 ligne).

PG_DIAG_MESSAGE_DETAIL

Message détaillé de l'erreur (plusieurs lignes).

PG_DIAG_MESSAGE_HINT

Suggestions pour la résolution de l'erreur.

PG_DIAG_STATEMENT_POSITION

Position de l'erreur dans la commande.

PG_DIAG_CONTEXT

Contexte de l'erreur dans la commande (ex : nom de la fct. PL/pgSQL).

PQclear

Libère la mémoire allouée par *PGresult*.

void **PQclear**(PQresult *res);

Appelée si plus besoin de res.

4.2. Fonctions sur le résultat d'une commande SELECT

Ces fonctions sont utilisées pour extraire des informations à partir d'un objet de type PGresult obtenu après exécution d'une commande de type SELECT.

PQntuples

Retourne le nombre de lignes.

int **PQntuples**(const PGresult *res);

PQnfields

Retourne le nombre de colonnes dans chaque ligne résultat.

int **PQnfields**(const PGresult *res);

PQfname

Retourne le nom de la colonne dont le n° est donné (commence à 0).

char ***PQfname**(const PGresult *res, int column_number);

NULL est retourné si le n° est hors domaine.

PQfnumber

Retourne le n° de colonne dont le nom est donné.

int **PQfnumber**(const PGresult *res, const char *column_name);

-1 est retourné si aucun nom de colonne ne corresponds.

Exemple :

select 1 as FOO, 2 as "BAR";

résultats :

PQfname(res, 0) foo

PQfname(res, 1) BAR

PQfnumber(res, "FOO") 0

PQfnumber(res, "foo") 0

PQfnumber(res, "BAR") -1

PQfnumber(res, "\"BAR\"") 1

PQftable

Retourne l'OID de la table à partir de laquelle une colonne a été récupérée.

Oid **PQftable**(const PGresult *res, int column_number);

InvalidOid est retourné si le n° de colonne est hors domaine.

PQftablecol

Retourne le rang de l'attribut dans la table correspondant à une colonne récupérée.

int **PQftablecol**(const PGresult *res, int column_number);

Zero est retourné si le n° de colonne est hors domaine, ou si la colonne ne fait pas référence à une colonne d'une table (ex : expression).

PQfformat

Retourne le format de la colonne dont le n° est donné.

int **PQfformat**(const PGresult *res, int column_number);

Zero est retourné si format texte, 1 est retourné si format binaire.

PQftype

Retourne l'OID du type de la colonne dont le n° est donné.

Oid **PQftype**(const PGresult *res, int column_number);

PQfsize

Retourne la taille (en octets) de la colonne dont le n° est donné.

int **PQfsize**(const PGresult *res, int column_number);

PQbinaryTuples

Retourne 1 si le PGresult contient des données binaire, 0 si données textes.

int **PQbinaryTuples**(const PGresult *res);

PQgetvalue

Retourne la valeur d'une colonne donnée pour une ligne donnée.

char* **PQgetvalue**(const PGresult *res, int row_number, int column_number);

Le n° de ligne commence à 0.

PQgetisnull

Teste si la valeur d'une colonne donnée pour une ligne donnée est NULLE.

int **PQgetisnull**(const PGresult *res, int row_number, int column_number);

Retourne 1 si oui, 0 sinon.

PQgetlength

Retourne la taille réelle (en octets) de la valeur d'une colonne donnée pour une ligne donnée.

int **PQgetlength**(const PGresult *res, int row_number, int column_number);

Equivalent à *strlen*().

PQprint

Affiche toutes les lignes avec les noms des colonnes en option.

```
void PQprint(FILE* fout,      /* buffer de sortie */
             const PGresult *res,
             const PQprintOpt *po);

typedef struct {
    pqbool  header;      /* affiche les en-têtes et le nombre de lignes */
    pqbool  align;       /* champs alignés */
    pqbool  standard;    /* format  standard */
    pqbool  html3;       /* format HTML */
    pqbool  expanded;    /* format étendu */
    pqbool  pager;       /* pagination */
    char    *fieldSep;   /* caractère séparateur de champs */
    char    *tableOpt;   /* attributs de TABLE si format HTML */
    char    *caption;    /* caption de TABLE si format HTML */
    char    **fieldName; /* alias des colonnes */
} PQprintOpt;
```

4.3. Fonctions sur le résultat d'une commande autre que SELECT

Ces fonctions sont utilisées pour extraire des informations à partir d'un objet de type PGresult obtenu après exécution d'une commande autre que SELECT.

PQcmdStatus

Retourne le statut de la commande SQL qui a généré le PGresult.

char * **PQcmdStatus**(PGresult *res);

Généralement, retourne le nom de la commande.

PQcmdTuples

Retourne le nombre de lignes affectées (supprimées, insérées, ...) par la commande.

char * **PQcmdTuples**(PGresult *res);

PQoidValue

Retourne l'OID de la ligne insérée (si commande INSERT). Retourne *InvalidOid* si pas INSERT ou si plusieurs insertions.

Oid **PQoidValue**(const PGresult *res);

4.4. Fonction de protection de caractères spéciaux dans une chaîne

PQescapeString

Protège les caractères spéciaux (comme ', \) des constantes chaînes utilisées dans une commande SQL.

size_t **PQescapeString** (char *to, const char *from, size_t length);

from pointe sur le 1er caractère à protéger de la chaîne. *length* est le nombre maximal de caractères à considérer. *to* est la chaîne résultat. La longueur de la chaîne *to* est retournée. Ressemble à *strncpy*.

4.5. Fonction de protection de caractères spéciaux dans une chaîne binaire

PQescapeBytea

Idem. **PQescapeString** mais avec chaîne binaire.

unsigned char ***PQescapeBytea**(const unsigned char *from, size_t from_length,
size_t *to_length);

from_length est le nombre d'octets à considérer. *to_length* est la chaîne résultat. Qui est en plus retournée.

PQunescapeBytea

Inverse de **PQescapeBytea.**

unsigned char ***PQescapeBytea**(const unsigned char *from,
size_t *to_length);

PQfreemem

rend l'espace alloué par **PQescapeBytea et PqunescapeBytea.**

void **PQfreemem**(void * ptr);

5. Fonctions de contrôle

Ces fonctions sont utilisées pour débugger.

5.1. PqsetErrorVerbosity

Détermine le degré de détails des messages retournés par *PQerrorMessage* et *PQresultErrorMessage*.

```
typedef enum {
    PQERRORS_TERSE,
    PQERRORS_DEFAULT,
    PQERRORS_VERBOSE
} PGVerbosity;
```

PGVerbosity **PQsetErrorVerbosity**(PGconn *conn, PGVerbosity verbosity);

mode *TERSE* : Message résumé de l'erreur (1 ligne).

mode *DEFAULT* : Message détaillé de l'erreur + position et contexte de l'erreur (plusieurs lignes).

mode *VERBOSE* : Message détaillé de l'erreur + toute autre information existante.

5.2. Pqtrace

Active la trace pour le debugger dans un fichier résulat.

void **PQtrace**(PGconn *conn, FILE *stream);

5.3. PQuntrace

Désactive la trace mise en place par *Pqtrace*.

void **PQuntrace**(PGconn *conn);

6. Variables d'environnement

Variables de connexion utilisées par *PQconnectdb*, *PQsetdbLogin* et *Pqsetdb :*

- PGHOST : nom du serveur BD.
- PGHOSTADDR : adresse IP address du serveur BD.
- PGPORT : n° port TCP.
- PGDATABASE : nom de la BD.
- PGUSER : nom de l'utilisateur de la BD.
- PGOPTIONS : options du serveur.
- PGCONNECT_TIMEOUT : délai d'attente (en secondes) pour la connexion.

Variables de session (voir aussi ALTER USER et ALTER DATABASE pour choisir ces valeur par utilisateur et/ou par BD) :

- PGDATESTYLE : représentation des dates et heures (équivalent à *SET datestyle TO ...*.)
- PGTZ : zone pour le temps (équivalent à *SET timezone TO ...*.)

7. Compilation

Etapes **:**

- Inclure le fichier libpq-fe.h :

  #include <libpq-fe.h>

- Indiquer au compilateur le répertoire où se trouvent les fichiers PostgreSQL inclus :

  cc -c -I/usr/local/pgsql/include testprog.c

  Si un makefile est utilisé, ajouter :

  CPPFLAGS += -I/usr/local/pgsql/include

  Pour savoir où se trouvent les fichiers d'inclusion, utiliser *pg_config* :

  $ pg_config --includedir
  /usr/local/include

- A l'édition de liens, ajouter l'option *-lpq* pour importer la librairie *libpq* et l'option *-Ldirectory* pour indiquer où elle se trouve. Pour une portabilité maximale, mettre l'option *-L* avant l'option *-lpq.* Par example:

  cc -o testprog testprog1.o testprog2.o -L/usr/local/pgsql/lib -lpq

  Pour savoir où se trouvent les librairies, utiliser *pg_config* :

  $ pg_config --libdir
  /usr/local/pgsql/lib

8. Programmes exemples

Programme Exemple 1

```
/*testlibpq.c */
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"

static void exit_nicely(PGconn *conn)
{
        PQfinish(conn);
        exit(1);
}

int main(int argc, char **argv)
{
        const char *conninfo;
        PGconn     *conn;
        PGresult   *res;
        int              nFields;
        int                   i, j;

        if (argc > 1)
                conninfo = argv[1];
        else
                conninfo = "dbname = template1";

        /* Etablit une connection à la BD */
        conn = PQconnectdb(conninfo);

        /* Vérifie la connexion */
        if (PQstatus(conn) != CONNECTION_OK)
        {
                fprintf(stderr, "Connexion à la BD '%s' échouée.\n", PQdb(conn));
                fprintf(stderr, "%s", PQerrorMessage(conn));
                exit_nicely(conn);
        }


        /*  Utilisation d'un curseur, donc BEGIN ... END obligatoire.
           L'exo peut être fait avec PQexec() de "select * from pg_database" ! */
```

```
/* Commencer un bloc transactionnel */

res = PQexec(conn, "BEGIN");

if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
   fprintf(stderr, "Commande BEGIN échouée : %s", PQerrorMessage(conn));
   PQclear(res);
   exit_nicely(conn);
}


/* Libération de la mémoire allouée pour res */

PQclear(res);


/* Récupérer les lignes du cataloque pg_database */

res = PQexec(conn, "DECLARE c CURSOR FOR select * from pg_database");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
   fprintf(stderr, "DECLARE CURSOR échoué: %s", PQerrorMessage(conn));
   PQclear(res);
   exit_nicely(conn);
}

PQclear(res);

res = PQexec(conn, "FETCH ALL in c");
if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
        fprintf(stderr, "FETCH ALL échoué : %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
}

/* Afficher d'abord les noms des colonnes */

nFields = PQnfields(res);
for (i = 0; i < nFields; i++) printf("%-15s", PQfname(res, i));
printf("\n\n");
```

```
    /* Afficher les lignes */

    for (i = 0; i < PQntuples(res); i++)
    {
        for (j = 0; j < nFields; j++) printf("%-15s", PQgetvalue(res, i, j));
        printf("\n");
    }

    PQclear(res);


    /* Fermer le curseur, sans vérification ... */

    res = PQexec(conn, "CLOSE c");
    PQclear(res);


    /* Fin de la transaction */

    res = PQexec(conn, "END");
    PQclear(res);


    /* Fermer la connexion à la base */

    PQfinish(conn);

    return 0;
}
```

Programme Exemple 2

```
/* testlibpq2.c
   Faire d'abord sous SQL :
   CREATE TABLE test1 (i int4, t text, b bytea);

   INSERT INTO test1 values (1, 'joe''s place', '\\000\\001\\002\\003\\004');
   INSERT INTO test1 values (2, 'ho there', '\\004\\003\\002\\001\\000');

   But du programme : obtenir en sortie :

   tuple 0: obtenu
   i = (4 octets) 1
   t = (11 octets) 'joe's place'
   b = (5 octets) \000\001\002\003\004
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include "libpq-fe.h"

/* Pour ntohl/htonl */
#include <netinet/in.h>
#include <arpa/inet.h>

static void exit_nicely(PGconn *conn)
{
        PQfinish(conn);
        exit(1);
}

int main(int argc, char **argv)
{
        const char *conninfo;
        PGconn     *conn;
        PGresult   *res;
        const char *valeurs[1];
        int                     i, j;
        int                     i_fnum, t_fnum, b_fnum;
        if (argc > 1)
                conninfo = argv[1];
        else
                conninfo = "dbname = template1";
```

```
/* Etablit une connection à la BD */
 conn = PQconnectdb(conninfo);

 /* Vérifie la connexion */
 if (PQstatus(conn) != CONNECTION_OK)
 {
        fprintf(stderr, "Connexion à la BD '%s' échouée.\n", PQdb(conn));
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
 }


 /*  Utilisation de PqexecParams() avec entrée texte et résultats binaires
  */


 /* Valeur du paramètre */
 valeurs[0] = "joe's place";

 res = PQexecParams(conn,"SELECT * FROM test1 WHERE t = $1",
                 1,            /* 1 paramètre */
                 NULL,         /* type à déduire*/
                 valeurs,
                 NULL,         /* longueurs inutiles car paramètres tous type texte */
                 NULL,         /* formats inutiles car paramètres tous type texte  */
                 1);           /* résultat binaire */

 if (PQresultStatus(res) != PGRES_TUPLES_OK)
 {
        fprintf(stderr, "SELECT échoué: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
 }

 /* PQfnumber retourne la place dans le résultat de l'attribut indiqué  */
 i_fnum = PQfnumber(res, "i");
 t_fnum = PQfnumber(res, "t");
 b_fnum = PQfnumber(res, "b");

 for (i = 0; i < PQntuples(res); i++)
 {
        char       *iptr, *tptr, *bptr;
        int         blen, ival;

        /* Récupère les valeurs  (on suppose quelles ne sont pas NULL) */
        iptr = PQgetvalue(res, i, i_fnum);
```

```
            tptr = PQgetvalue(res, i, t_fnum);
            bptr= PQgetvalue(res, i, b_fnum);

            /* Les conversion qui suivent sont sans importance */

            /* Convertir la représentation binaire de INT4.*/
            ival = ntohl(*((uint32_t *) iptr));

            /*La représentation binaire d'un TEXT est du texte.*/

            /* La représentation binaire d'un BYTEA est une suite d'octets,
            pouvant inclure des nulls, donc afficher caractère par caractère */

            blen = PQgetlength(res, i, b_fnum);

            printf("tuple %d: obtenu \n", i);
            printf(" i = (%d octets) %d\n", PQgetlength(res, i, i_fnum), ival);
            printf(" t = (%d octets) '%s'\n", PQgetlength(res, i, t_fnum), tptr);
            printf(" b = (%d octets) ", blen);
            for (j = 0; j < blen; j++) printf("\\%03o", bptr[j]);
            printf("\n\n");
        }

        PQclear(res);

        /* Ferme la connexion */
        PQfinish(conn);

        return 0;

}
```

GESTION DES GRANDS OBJETS (LOB)

But

- Travailler avec des objets de grande dimension (Large OBject) :
  Binary LOB, Character LOB, uNicode.
- Pour importer des fichiers (text, son, images) entiers dans une BD.
- Gestion des fichiers stockés dans la BD facilitée par les outils du SGBD :
  Ex : dans la BD *gestair*, rechercher le fichier descriptif des appareils dont le nombre de place *nbplace* est supérieur à 300.

1. Fonctions côté serveur BD

- Utilisables sous SQL.
- Utilisables par le propriétaire de la BD (nécessité donc d'avoir le droit de *create database*).
- Utilisent le type oid (Object Identifier).

Liste des fonctions implémentées sous PostgreSQL 7.3.4 :

Type du résultat	Nom	Type des paramètres
integer	lo_close	integer
oid	lo_creat	integer
integer	lo_export	oid, text
oid	lo_import	text
integer	lo_open	oid, integer
integer	lo_tell	integer
integer	lo_unlink	oid

Exemple :

Soit la création de la table ob_appareil :

codetype	CHAR(3)
nbplace	NUMERIC(3)
design	VARCHAR(30)
description	OID
photo	OID
plan	OID

où

description : description détaillée de l'appareil
photo : photo de l'appareil

plan : plan de l'appareil.

1.1. Insertion des Gros Objets (LOB)

Utiliser la fonction l**o_import**(*nom_fichier_physique*).

Exemple :

Insertion d'un nouvel appareil :

```
INSERT INTO ob_appareil VALUES ('A34', 261, 'A340-200',
            lo_import('/tmp/A34_desc.txt'),
            lo_import('/tmp/A34_photo.jpg') , lo_import('/tmp/A34_plan.gif'));
```

=> génération d'un OID unique et chargement pour chacun des fichiers.

Exemple :

SELECT * FROM ob_appareil donne :

codetype	nbplace	design	description	photo	plan
A34	261	A340-200	51745	51746	51747

1.2. Exportation des Gros Objets (LOB)

Utiliser la fonction **lo_export**(*attribut_de_type_OID*,*nom_fichier_physique*).

Exemple :

```
SELECT    lo_export(description,'/tmp/A34_desc_exp.txt'),
          lo_export(photo, '/tmp/A34_photo_exp.jpg') ,
          lo_export(plan, '/tmp/A34_plan_exp.gif')
FROM ob_appareil
WHERE codetype='A34';
```

1.3. Suppression des Gros Objets (LOB)

Utiliser la fonction **lo_unlink**(*attribut_de_type OID*) puis l'instruction **DELETE**.
Exemple :

```
SELECT lo_unlink(description), lo_unlink(photo) , lo_unlink(plan)
FROM ob_appareil
WHERE codetype='A34';
```

=> les fichiers sont supprimés du disque.

Nécessité de supprimer également les identificateurs d'objets dans la BD :

```
DELETE FROM ob_appareil WHERE codetype='A34';
```

Attention : respecter l'ordre des étapes.

1.4. Création, ouverture, fermeture, position courante

Les fonctions implémentées **lo_creat, lo_open, lo_close** et **lo_tell** sont inutiles sous le SQL de PostgreSQL 7.3.4 car les fonctions de manipulations lo_read, lo_write et lo_seek ne sont pas implémentées.

2. Fonctions côté client

- Utilisables à partir d'un langage hôte (comme le langage C avec la librairie *libpq*).
- Utilisables par tous les utilisateurs BD.
- Utilisent le type Oid (Object Identifier).
- Utilisent une connexion à la BD (de type PGconn *) préalablement établie.

2.1. Création des Gros Objets (LOB)

Oid **lo_creat**(PGconn *conn, int mode);

où

mode
INV_READ (lecture), INV_WRITE(écriture), NV_READ|INV_WRITE (lec./écr.).

L'OID de l'objet créé (vide) est retourné.

Exemple :

inv_oid = lo_creat(INV_READ|INV_WRITE);

2.2. Importation des Gros Objets (LOB)

Oid **lo_import**(PGconn *conn, const char *filename);

où

filename
nom du fichier système (avec chemin) à importer.

L'OID de l'objet importé est retourné.

2.3. Exportation des Gros Objets (LOB)

int **lo_export**(PGconn *conn, Oid lobjId, const char *filename);

où
lobjId
OID du LOB (existant) à ouvrir.

filename
nom du fichier système (avec chemin) contenant le LOB exporté.

Si l'opération échoue, un nombre négatif est retourné.

2.4. Ouverture des Gros Objets (LOB)

int **lo_open**(PGconn *conn, Oid lobjId, int mode);

où
lobjId
OID du LOB (existant) à ouvrir.
mode
INV_READ (lecture), INV_WRITE(écriture), NV_READ|INV_WRITE (lec./écr.).

Retourne le descripteur de l'objet utilisé pour lo_read, lo_write, lo_lseek, lo_tell, et lo_close. Si l'opération échoue, un nombre négatif est retourné.

2.5. Ecriture dans des Gros Objets (LOB)

int **lo_write**(PGconn *conn, int fd, const char *buf, size_t len);

où
fd
descripteur de l'objet retourné précédemment par *lo_open*.
buf
chaîne d'octets à écrire dans l'objet.
len
nombre d'octets à écrire dans l'objet.

Retourne le nombre effectif d'octets écrits. Si l'opération échoue, un nombre négatif est retourné.

2.6. Lecture à partir de Gros Objets (LOB)

int **lo_read**(PGconn *conn, int fd, char *buf, size_t len);

où

fd : descripteur de l'objet retourné précédemment par *lo_open*.

buf : chaîne d'octets à lire dans l'objet.

len : nombre d'octets à lire dans l'objet.

Retourne le nombre effectif d'octets lus. Si l'opération échoue, un nombre négatif est retourné.

2.7. Positionnement dans de Gros Objets (LOB)

int **lo_lseek**(PGconn *conn, int fd, int offset, int whence);

où

fd

descripteur de l'objet retourné précédemment par *lo_open*.

offset

déplacement relatif par rapport à *whence* en nombre d'octets.

whence

SEEK_SET (début), SEEK_CUR (position courante), et SEEK_END (fin).

2.8. Retour de la position courante dans de Gros Objets (LOB)

int **lo_tell**(PGconn *conn, int fd);

Retourne le position courante dans le LOB (nombres d'octets par rapport au début). Si l'opération échoue, un nombre négatif est retourné.

2.9. Fermeture de Gros Objets (LOB)

int **lo_close**(PGconn *conn, int fd);

où

fd : descripteur de l'objet retourné précédemment par *lo_open*.

Si l'opération échoue, un nombre négatif est retourné.

2.10. Suppression de Gros Objets (LOB) de la BD

int **lo_unlink**(PGconn *conn, Oid lobjId);

où

lobjId : descripteur de l'objet retourné précédemment par *lo_open*.

Si l'opération échoue, un nombre négatif est retourné.

Le fichier système associé n'est pas supprimé.

2.11. Programme exemple

Remarque : ce programme travail avec des LOBs sous la BD mais pas dans une table particulière. Rien n'empêche de le faire ici. Ainsi, par exemple au lieu d'obtenir un OID par importation à partir d'un fichier système, comme : *lobjOid = lo_import(conn, in_filename)*, l'on peut obtenir un OID par une recherche dans une table le contenant :

req=query(conn, "SELECT description FROM ob_appareil where codetype='AB3'");

lobjOid = PQgetvalue(res,1,1);

/* Programme exemple.c */

```
#include <stdio.h>
#include "libpq-fe.h"
#include "libpq/libpq-fs.h"
#define BUFSIZE         1024

/* importe le fichier unix "filename" dans le LOB "lobjOid" de la BD*/
Oid importFile(PGconn *conn, char *filename)
{
   Oid      lobjId;
   int      obj_fd;
   char     buf[BUFSIZE];
   int      nbytes, tmp, fd;

   /*ouvre le fichier à lire */
   fd = open(filename, O_RDONLY, 0666);
   if (fd < 0) fprintf(stderr, "ouverture du fichier unix %s impossible\n", filename);

   /* créé un LOB vide en lecture | écriture */
   lobjId = lo_creat(conn, INV_READ | INV_WRITE);
   if (lobjId == 0) fprintf(stderr, "création du LOB impossible\n");

   lobj_fd = lo_open(conn, lobjId, INV_WRITE);

   /* lit le fichier unix et écrit dans le LOB par tranches de 1024 octets */
   while ((nbytes = read(fd, buf, BUFSIZE)) > 0)
   {
      tmp = lo_write(conn, lobj_fd, buf, nbytes);
      if (tmp < nbytes) fprintf(stderr, "erreur lors d'écriture dans le LOB\n");
   }

   (void) close(fd);  /* fermer le fichier */
   (void) lo_close(conn, lobj_fd); /* fermer le LOB */
   return lobjId;
}
```

```
/* Exporte le LOB "lobjOid" vers le fichier unix  "filename" */

void exportFile(PGconn *conn, Oid lobjId, char *filename)
{
   int          lobj_fd;
   char       buf[BUFSIZE];
   int          nbytes,tmp;
   int         fd;

   /*ouvre le LOB en lecture*/

   lobj_fd = lo_open(conn, lobjId, INV_READ);
   if (lobj_fd < 0)
        fprintf(stderr, "ouverture du LOB %d impossible\n",lobjId);

   /* ouvre le fichier unix en écriture */

   fd = open(filename, O_CREAT | O_WRONLY, 0666);
   if (fd < 0)
      fprintf(stderr, "ouverture du fichier %d impossible\n ", filename);

   /* lit le LOB et écrit dans le fichier unix par tranches de 1024 octets */

   while ((nbytes = lo_read(conn, lobj_fd, buf, BUFSIZE)) > 0)
   {
      tmp = write(fd, buf, nbytes);
      if (tmp < nbytes)
          fprintf(stderr, "erreur lors d'écriture dans le fichier %s\n", filename);
   }

   (void) lo_close(conn, lobj_fd);

   (void) close(fd);

   return;
}
```

/ lecture de *len* octets à partir de la position *start* dans le lob *lobjid ,* puis affichage */

```
void pickout(PGconn *conn, Oid lobjId, int start, int len)
{
   int        lobj_fd;
   char       *buf;
   int        nbytes;
   int        nread;

   lobj_fd = lo_open(conn, lobjId, INV_READ);

   if (lobj_fd < 0)
       fprintf(stderr, "ouverture du LOB %d impossible\n", lobjId);

   /* déplacement de start octets à partir du début du LOB */
   lo_lseek(conn, lobj_fd, start, SEEK_SET);

   buf = malloc(len + 1);

   nread = 0;
   /* Lit le LOB et remplit la zone mémoire buf
   while (len - nread > 0)
   {
      nbytes = lo_read(conn, lobj_fd, buf, len - nread);
      buf[nbytes] = ' ';
      fprintf(stderr, ">>> %s", buf);         /* affiche au fur et à mesure buf */
      nread += nbytes;
   }

   free(buf);
   fprintf(stderr, "\n");
   lo_close(conn, lobj_fd);

}
```

/* écriture de *len* octets à partir de la position *start* dans le lob *lobjid*/

```
void overwrite(PGconn *conn, Oid lobjId, int start, int len)
{
    int         lobj_fd;
    char        *buf;
    int         nbytes, nwritten, i;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0) fprintf(stderr, "ouverture du LOB %d impossible\n",lobjId);

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);
        /* on veut écrire XXX..... dans le LOB
    for (i = 0; i < len; i++) buf[i] = 'X';
    buf[i] = ' ';

    nwritten = 0;
    while (len - nwritten > 0)
    {
        nbytes = lo_write(conn, lobj_fd, buf + nwritten, len - nwritten);
        nwritten += nbytes;
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}
```

/* Terminer la connexion et quitter le programme */

```
void exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}
```

```
int main(int argc, char **argv)
{
   char       *in_filename, *out_filename, *database;
   Oid         lobjOid;
   PGconn     *conn;
   PGresult         *res;
   if (argc != 4)
   {
      fprintf(stderr, "Usage: %s nom_bd   fichier_entree    fichier_sortie", argv[0]);
      exit(1);
   }

   database = argv[1]; in_filename = argv[2]; out_filename = argv[3];

   /*Etablir la connexion */
   conn = PQsetdb(NULL, NULL, NULL, NULL, database);

   /* Vérifier si la connexion a bien été établie*/
   if (PQstatus(conn) == CONNECTION_BAD)
   {
      fprintf(stderr, "Connexion à la base '%s' échouée.\n", database);
      fprintf(stderr, "%s", PQerrorMessage(conn));
      exit_nicely(conn);
   }

   res = PQexec(conn, "begin"); PQclear(res);

   printf("Importation du fichier %s\n", in_filename);
/*  lobjOid = importFile(conn, in_filename);   solution plus lourde mais explicative */
   lobjOid = lo_import(conn, in_filename);

   printf("sous le LOB %d.\n", lobjOid);

   printf("lecture et affichage des octets n° 1000 à 2000 du LOB\n");
   pickout(conn, lobjOid, 1000, 1000);

   printf("écriture de XXXX... sur les octets de n° 1000 à 2000 du LOB\n");
   overwrite(conn, lobjOid, 1000, 1000);

   printf("Exportation du LOB vers le fichier %s\n", out_filename);
/*  exportFile(conn, lobjOid, out_filename); solution plus lourde mais explicative*/
   lo_export(conn, lobjOid, out_filename);

   res = PQexec(conn, "end");
   PQclear(res); PQfinish(conn);
   exit(0);
}
```