2024-2025

IUT Clermont Auvergne
Département Informatique, Aubière
2ème année

SQL dans un langage de programmation

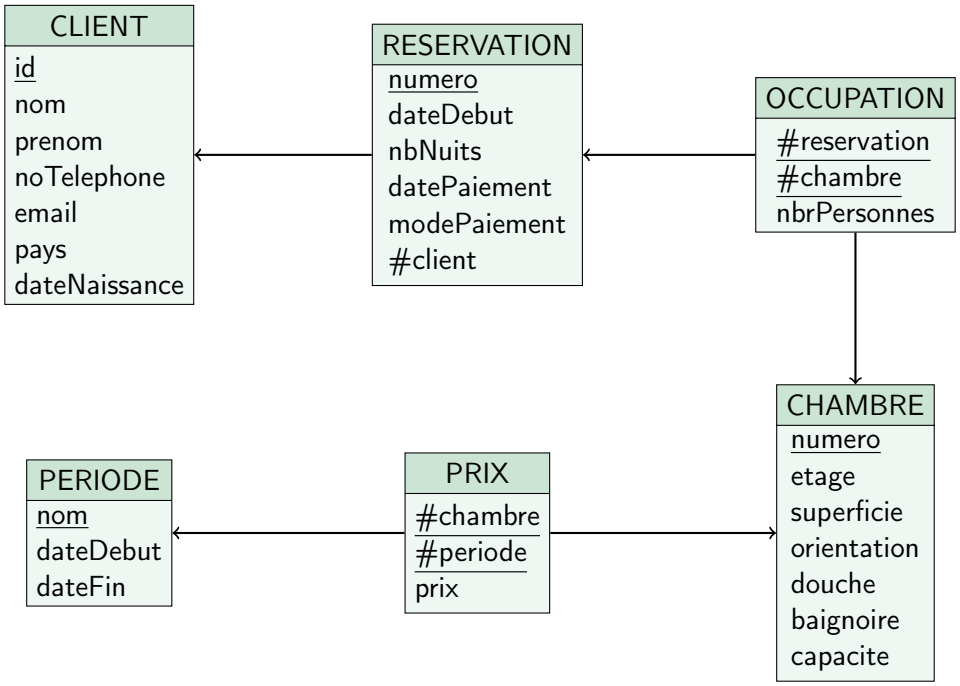
Polycopié de cours

Anaïs Durand, Franck Glaziou
Julie Rossignol, Xavier Valette

Plan du cours

MLD utilisé

Sauf mention contraire, le MLD suivant est celui qui sera utilisé dans tous les exemples et exercices de ce cours. Il correspond à un extrait de la base de données d'un hôtel. On a donc des informations sur les chambres (taille, équipement, ...), leurs prix en fonction des périodes de l'année ainsi que sur les clients et leurs réservations.



1. SQL

1.1. Types de données

Les principaux types de données d'une base PostgreSQL sont les suivants.

Types de données	Description
`char(n)`	Chaîne de caractères de longueur **fixe** `n`. Si la taille de la chaîne est inférieure à `n`, du *padding* est utilisé pour la compléter avec des espaces.
`varchar(n)`	Chaîne de caractères de longueur **variable**, au plus `n`. La chaîne de caractères entrée n'est donc pas complétée avec du padding.
`text`	Chaîne de caractères de longueur illimitée.
`numeric(p,s)`	Nombre entier ou décimal. `p` correspond au nombre total de chiffres et `s` correspond au nombre de chiffres après la virgule. Il est possible d'utiliser `numeric` sans `s` ni `p`. `numeric(p)` correspond à un entier sur `p` chiffres.
`integer`	Nombre entier entre -2147483648 to +2147483647
`date`	Date
`time`	Heure
`timestamp`	Date et heure
`interval`	Durée
`boolean`	Booléen `true` ou `false`

1.2. Gestion des tables

La **table** est l'objet principal permettant de stocker des données. Pour pouvoir définir une nouvelle table dans une base de données, il faut avoir le droit d'en créer.

▶ **Création d'une table.** La commande `CREATE TABLE` permet de créer une table.

Les tables d'un utilisateur doivent avoir des noms différents, mais des tables d'utilisateurs différents peuvent porter le même nom. Les colonnes d'une table doivent avoir des noms différents, mais deux tables peuvent avoir une colonne de même nom.

EXEMPLE1.1

```
CREATE TABLE Chambre(
    numero      char(3) PRIMARY KEY,
    etage       numeric(1),
    superficie  numeric(2),
    orientation char(1),
    douche      boolean,
    baignoire   boolean,
    capacite    numeric(1)
);
```

▶ **Suppression d'une table.** La commande **DROP TABLE** permet de supprimer une table. L'option **CASCADE** (facultative) permet de supprimer les contraintes référentielles dans les tables filles faisant référence à la table mère supprimée.

EXEMPLE1.2

```
DROP TABLE chambre [CASCADE];
```

▶ **Modification de la structure d'une table.** La commande **ALTER TABLE** permet de modifier la structure d'une table (ses colonnes, ses contraintes ...).

EXEMPLE1.3

```
--Colonne ajoutée
ALTER TABLE Chambre ADD accesPMR boolean;
--Contrainte ajoutée
ALTER TABLE Chambre ADD CONSTRAINT ck_chambre_etage CHECK(etage >= 0);
--Contrainte supprimée
ALTER TABLE Chambre DROP CONSTRAINT ck_chambre_etage;
```

1.3. Contraintes non référentielles

Les **contraintes non référentielles** permettent de préciser les données d'un attribut. Les contraintes peuvent être définies lors de la création de la table ou peuvent être ajoutées/supprimées lors d'une modification ultérieure.

▶ **Syntaxe :**

```
CREATE TABLE <nom_table> (
    <nom_colonne 1> <type> [contrainte de colonne],
    <nom_colonne 2> <type> [contrainte de colonne],
    ...,
    [contrainte de table],
    ...
);
```

▶ **Contraintes possibles :**

— **PRIMARY KEY** : Définit la **clé primaire** d'une table. Il ne peut y avoir d'une clé primaire par table, mais celle-ci peut être composée de plusieurs colonnes (dans ce cas, la notation sous forme de contrainte de table est obligatoire). La valeur d'une clé primaire doit toujours être définie (non **NULL**) et unique.

— **DEFAULT** : Indique la valeur par défaut à mettre dans la colonne en cas d'insertion où la valeur de colonne concernée n'est pas précisée.

— **NULL**/**NOT NULL** : Autorise/interdit la valeur **NULL** pour la colonne.

— **UNIQUE** : Contraint l'unicité de la valeur.

— **CHECK** : Effectue une vérification sur la valeur de la/des colonne(s), par exemple : `col < 10`, `col IN (1,2,3,4)`, `colA = 1 AND colB = 2`, ...

⚠ Si la contrainte **CHECK** porte sur plusieurs colonnes (par exemple, `colA < colB`), celle-ci doit être définie sous forme de contrainte de table.

Exemple1.4

```
CREATE TABLE Chambre(
    numero      char(3) PRIMARY KEY,
    etage       numeric(1) NOT NULL CHECK(etage >= 0),
    superficie  numeric(2) CHECK(superficie > 0),
    orientation char(1) CHECK(orientation IN ('N', 'S', 'O', 'E')),
    douche      boolean,
    baignoire   boolean,
    capacite    numeric(1) CHECK(capacite > 0)
);
```

Il est préférable de nommer la contrainte. En effet, lorsqu'une requête SQL engendre la violation d'une contrainte un message d'erreur apparaît en indiquant la contrainte concernée. Si le nom est clairement défini, l'erreur est plus facilement compréhensible.

Exemple1.5

```
CREATE TABLE Periode(
    nom       char(10) CONSTRAINT periode_pk PRIMARY KEY,
    dateDebut date CONSTRAINT periode_dateDebut NOT NULL,
    dateFin   date CONSTRAINT periode_dateFin NOT NULL,
    CONSTRAINT periode_ck_dates CHECK(dateDebut < dateFin)
);

INSERT INTO periode VALUES ('AUTOMNE22', '2022-09-22'::date,
                                '2022-02-22'::date);

--ERREUR:  la nouvelle ligne de la relation ``periode'' viole la
   contrainte de
--                 vérification ``periode_ck_dates''
--DÉTAIL : La ligne en échec contient (AUTOMNE22 , 2022-09-22,
   2022-02-22)
```

Il est possible d'ajouter, supprimer ou modifier une contrainte après la création de la table en utilisant la commande **ALTER TABLE**.

Exemple1.6

```
CREATE TABLE MaTable (
    nombre numeric PRIMARY KEY
);

ALTER TABLE MaTable ADD CONSTRAINT ck_nombre CHECK (nombre < 10);
ALTER TABLE MaTable DROP CONSTRAINT ck_nombre;
```

1.4. Contraintes référentielles

Une **contrainte référentielle** permet de matérialiser, au niveau de la base de données, les clés étrangères définies dans le MLD. Il faut que la table mère ait une clé primaire définie et que la table fille fasse référence à cette clé primaire. Les valeurs dans la colonne de la table fille doivent alors forcément exister dans la table mère.

▶ **Syntaxe :**

```
[CONSTRAINT <nom_contrainte> [FOREIGN KEY (col1, ...)]
    REFERENCES <nom_table>(<nom_colonne>) [ON DELETE CASCADE]
```

— **FOREIGN KEY** permet de définir la (ou les) colonne(s) de la table fille qui font partie de la clé étrangère. Cette clause est obligatoire si la contrainte est définie sous forme de contrainte de table.

— **REFERENCES** définit la contrainte référentielle par rapport à une clé unique ou primaire. Il n'est pas nécessaire de préciser le nom de la (ou les) colonne(s) référencées dans la table mère s'il s'agit de sa clé primaire.

— **ON DELETE CASCADE** est une option permettant de conserver l'intégrité référentielle en supprimant automatiquement les lignes de la table fille lorsque la ligne de la table mère à laquelle elles font références est supprimée par un **DELETE**.

EXEMPLE1.7

```
CREATE TABLE Reservation(
    numero       char(8) PRIMARY KEY,
    dateDebut    date    NOT NULL,
    nbNuits      numeric CHECK (nbNuits IS NOT NULL AND nbNuits > 0),
    datePaiement date,
    modePaiement char(2) CHECK (modePaiement IN ('CB', 'LI', 'CH')),
    client       char(8) REFERENCES Client(id)
);

CREATE TABLE Prix(
    chambre char(3) REFERENCES Chambre,
    periode char(10),
    prix    numeric CHECK(prix > 0),
    PRIMARY KEY(chambre, periode),
    FOREIGN KEY (periode) REFERENCES Periode
);
```

Exercice 1

1.5. Fonctions mono-lignes

Les fonctions mono-lignes agissent sur chaque ligne indépendamment. Ces fonctions peuvent s'imbriquer. Elles peuvent être utilisées dans une contrainte, dans un **SELECT**, ...

▶ **Manipulation des chaînes de caractères.**

Nom	Définition
`c1 || c2`	Concaténation des chaînes `c1` et `c2` *Exemple :* `SELECT 'dra'||'gon';` ⟶ `'dragon'`
`char_length(c)`	Longueur de `c` *Exemple :* `SELECT char_length('dragon');` ⟶ `6`
`lower(c)`	Minuscules *Exemple :* `SELECT lower('DRAGON');` ⟶ `'dragon'`
`upper(c)`	Majuscules *Exemple :* `SELECT upper('Dragon');` ⟶ `'DRAGON'`

Nom	Définition
`initcap(c)`	Première lettre de chaque mot en majuscule *Exemple :* `SELECT initcap('dragon noir');` ⟶ `'Dragon Noir'`
`ltrim(c1,c2)`	Suppression des caractères de `c1` de gauche à droite appartenant à l'ensemble `c2`, tant qu'un caractère de `c1` est dans `c2` *Exemple :* `SELECT ltrim('_ _dragon_', '_ ');` ⟶ `'dragon_'`
`rtrim(c1,c2)`	Suppression des caractères de `c1` de droite à gauche appartenant à l'ensemble `c2`, tant qu'un caractère de `c1` est dans `c2` *Exemple :* `SELECT rtrim('_dragon_ _', '_ ');` ⟶ `'_dragon'`
`substr(c, i)`	Sous-chaîne de `c` à partir du `i`ème caractère *Exemple :* `SELECT substr('dragon', 3);` ⟶ `'agon'`
`substr(c, i, n)`	Sous-chaîne de `c` de longueur `n` à partir du `i`ème caractère *Exemple :* `SELECT substr('dragon', 2, 3);` ⟶ `'rag'`

▶ Expressions régulières : `chaine [NOT] LIKE 'format'`

Le mot-clé `LIKE` permet de comparer la valeur d'une chaîne de caractères (une colonne, une expression, *etc.*) à un format. On peut utiliser des caractères génériques :

- `'%'` correspond à plusieurs caractères (0, 1, ou plus)
- `'_'` correspond à exactement un caractère

`'\%'` correspond au caractère % et n'est pas un caractère générique.

Exemple1.8

Tous les clients dont le numéro de téléphone commence par 06.

```
SELECT *
FROM Client
WHERE noTelephone LIKE '06%';
```

▶ Manipulation des valeurs numériques.

Fonctions	
Nom	**Définition**
`abs(n)`	Valeur absolue de `n` *Exemple :* `SELECT abs(-2.73);` ⟶ `2.73`
`ceil(n)`	Valeur entière supérieure de `n` *Exemple :* `SELECT ceil(2.73);` ⟶ `3`
`floor(n)`	Valeur entière inférieure de `n` *Exemple :* `SELECT floor(2.73);` ⟶ `2`
`mod(n1, n2)`	Reste de la division entière `n1/n2` *Exemple :* `SELECT mod(9,4);` ⟶ `1`
`round(n, p)`	Arrondi de `n` à `p` chiffres en partie décimale *Exemple :* `SELECT round(2.76,1);` ⟶ `2.8`
`trunc(n, p)`	Tronque de `n` à `p` chiffres en partie décimale *Exemple :* `SELECT trunc(2.76,1);` ⟶ `2.7`
`sign(n)`	Signe de `n` (`-1` si négatif, `0` si nul, `1` si positif) *Exemple :* `SELECT sign(-2.76);` ⟶ `-1`

Conversion	
Nom	**Définition**
`to_char(n,f)`	Converti le nombre `n` en chaîne de caractères selon le format `f` *Exemple :* `SELECT to_char(2.73,'09.999');` ⟶ `'02.730'`
`to_number(c,f)`	Converti la chaîne de caractères `c` en nombre selon le format `f` *Exemple :* `SELECT to_number('02.730','09.999');` ⟶ `2.73`

Principaux caractères utilisables dans le format : `9` pour un chiffre à afficher, sauf si non significatif ; `0` pour un chiffre toujours affiché, même si non significatif.

▶ **Manipulation des dates et heures.**

Opérations	
Nom	**Définition**
`d + nbj`	Ajoute `nbj` jours à la date `d` *Exemple :* `SELECT '2022-12-10'::date+3;` ⟶ `'2022-12-13'`
`d - nbj`	Retire `nbj` jours à la date `d` *Exemple :* `SELECT '2022-11-10'::date-3;` ⟶ `'2022-11-07'`
`d1 - d2`	Nombre de jours entre les dates `d1` et `d2` *Exemple :* `SELECT '2022-11-10'::date - '2022-11-06'::date;` ⟶ `4`
`d + t`	Timestamp correspondant à la date `d` et l'heure `t` *Exemple :* `SELECT '2022-11-10'::date + '10:39:51.2'::time;` ⟶ `'2022-11-10 10:39:51.2'`

Fonctions	
Nom	**Définition**
`CURRENT_DATE`	Date courante *Exemple :* `SELECT CURRENT_DATE;` ⟶ `'2022-07-21'`
`CURRENT_TIME`	Heure courante *Exemple :* `SELECT CURRENT_TIME;` ⟶ `'10:39:51.662522-05'`
`CURRENT_TIMESTAMP`	Date et heure courante *Exemple :* `SELECT CURRENT_TIMESTAMP;` ⟶ `'2022-07-21 10:39:51.662522-05'`
`date_trunc(p, t)`	Tronque le timestamp `t` à la précision `p` (parmi `'year'`, `'month'`, `'day'`, `'hour'`, `'minute'`, `'second'` ...) *Exemple :* `SELECT date_trunc('year', '2022-11-20'::date);` ⟶ `2022-01-01 00:00:00+01`
`date_part(p, t)`	Extrait la précision `p` du timestamp `t` *Exemple :* `SELECT date_part('year', '2021-11-20'::date);` ⟶ `2021`

Conversion	
Nom	**Définition**
`make_date(y,m,d)`	Crée une date *Exemple :* `SELECT make_date(2022,7,12);` ⟶ `'2022-07-12'`
`make_time(h,mi,s)`	Crée une heure *Exemple :* `SELECT make_time(17,34,20.5);` ⟶ `'17:34:20.5'`
`make_timestamp(y, m,d,h,mi,s)`	Crée un timestamp *Exemple :* `SELECT make_timestamp(2022,7,12,17,34,20.5);` ⟶ `'2022-07-12 17:34:20.5'`
`to_char(d,f)`	Converti la date ou le timestamp `d` en chaîne de caractères selon le format `f` *Exemple :* `SELECT to_char(CURRENT_DATE,'DD MON YYYY');` ⟶ `'20 JUL 2022'`

Conversion	
Nom	**Définition**
`to_date(c,f)`	Converti la chaîne de caractères `c` en date selon le format `f` *Exemple :* `SELECT to_date('20 JUL 2022','DD MON YYYY');` ⟶ `'2022-07-20'`
`to_timestamp(c,f)`	Converti la chaîne `c` en timestamp selon le format `f` *Exemple :* `SELECT to_timestamp('20/07/21 13:42','DD/MM/YY HH24:MI');` ⟶ `'2022-07-20 13:42:00.0'`

Principaux caractères utilisables dans le format :

- `HH` : heure (01-12)
- `HH24` : heure (01-24)
- `MI` : minute
- `SS` : seconde
- `YYYY` : année sur 4 chiffres
- `YY` : année sur 2 chiffres
- `MM` : numéro du mois
- `MON` : nom du mois sur 3 lettres
- `MONTH` : nom du mois
- `DD` : jour du mois
- `DAY` : nom du jour

Exercice 2

1.6. Manipulation de lignes

▶ **Insertion.** La commande `INSERT INTO` permet d'ajouter des lignes dans une table. Il n'est pas nécessaire de préciser les colonnes concernées si les valeurs à insérer renseignent toutes les colonnes de la table et si les valeurs sont données dans l'ordre de définition des colonnes lors de la création de la table.

Exemple1.9

```
INSERT INTO Chambre(numero, superficie, capacite) VALUES ('101', 12, 2);
INSERT INTO Chambre VALUES ('102', 1, 15, 'N', false, true, 2);
```

▶ **Mise à jour.** La commande `UPDATE` permet de modifier les valeurs des lignes d'une table. La clause `WHERE` est facultative.

Exemple1.10

```
UPDATE Chambre
SET capacite = 3
WHERE superficie > 15 AND superficie < 22;
```

▶ **Suppression.** La commande `DELETE` permet de supprimer des lignes d'une table. La clause `WHERE` est facultative.

Exemple1.11

```
DELETE FROM Chambre
WHERE etage = 5;
```

1.7. Consultation

La commande **SELECT** permet de consulter le contenu des tables.

▶ **Syntaxe :**

```
SELECT [ALL | DISTINCT] liste_de_sélection
[FROM liste_d_objets
  [WHERE condition]
  [GROUP BY critère
    [HAVING condition]]
  [ORDER BY {expression | position} [ASC | DESC] [, ...]]];
```

— **SELECT** permet de définir la liste de sélection. Cette liste peut contenir des colonnes, des expressions, des sous-requêtes délivrant une seule ligne ... L'option **DISTINCT** permet d'éliminer les lignes en doublon dans le résultat.
— **FROM** permet de définir la liste des objets qui vont être consultées. Ces objets peuvent être des tables, des vues, le résultat d'une fonction PL/pgSQL ...
— **WHERE** permet de filtrer le résultat selon une condition.
— **ORDER BY** permet d'ordonner le résultat selon un ou plusieurs critères.
— **GROUP BY** permet d'appliquer une fonction d'agrégation (voir Section 1.7.3) sur des sous-ensembles de lignes regroupés selon un critère.
— **HAVING** permet de filtrer le résultat selon une condition portant sur une fonction d'agrégation.

1.7.1. Jointures

▶ **Produit cartésien.** Le produit cartésien de plusieurs tables $T_1, T_2, \ldots, T_k$ permet d'obtenir toutes les combinaisons possibles entre une ligne de T_1, une ligne de T_2, ..., et une ligne de T_k. Le produit cartésien peut être écrit avec une virgule (**,**) entre les noms des tables ou en utilisant le mot-clé **CROSS JOIN**.

Exemple1.12

Les deux requêtes ci-dessous sont équivalentes :

```
SELECT *
FROM Periode, Prix;

SELECT *
FROM Periode CROSS JOIN Prix;
```

Si la table **Periode** contient six lignes et la table **Prix** en contient cinquante, $6 \times 50 = 300$ lignes sont retournées par ces requêtes.

▶ **Jointure interne.** Le produit cartésien associe toute combinaison de lignes possibles entre les tables indiquées dans le **FROM**, même s'il n'y a aucune cohérence entre ces lignes. En ajoutant des conditions de jointure, on peut filtrer le résultat du produit cartésien pour ne garder que lignes cohérentes.

Exemple1.13

Liste des chambres réservées pour ce soir avec le nombre de personnes les occupants.

```
SELECT c.numero, c.etage, o.nbrPersonnes
FROM Chambre c, Occupation o, Reservation r
WHERE c.numero = o.chambre /* conditions de jointure */
```

```
            AND o.reservation = r.numero
            AND r.dateDebut <= CURRENT_DATE /* conditions locales */
            AND r.dateDebut+r.nbNuits > CURRENT_DATE
ORDER BY c.etage, c.numero;
```

Cette notation, bien qu'encore utilisée, est désormais dépréciée. La notation `JOIN ... ON`, plus explicite, est préférée. `JOIN` est un raccourci pour `INNER JOIN`, autrement dit jointure interne en français.

Exemple1.14

Requête équivalente à celle de l'exemple 1.13.

```
SELECT c.numero, c.etage, o.nbrPersonnes
FROM Chambre c JOIN Occupation o ON c.numero = o.chambre
               JOIN Reservation r ON o.reservation = r.numero
WHERE r.dateDebut <= CURRENT_DATE
           AND r.dateDebut+r.nbNuits > CURRENT_DATE
ORDER BY c.etage, c.numero;
```

▶ **Jointure externe.** Lors d'une jointure interne entre une table T_1 et une table T_2, toute ligne de T_1 qui ne peut pas être associée à une ligne de T_2 (car aucune d'entre elles ne satisfait la condition de jointure) n'apparaît pas dans le résultat. De même, toute ligne de T_2 qui ne peut pas être associée à une ligne de T_1 n'apparaît pas non plus dans le résultat.

Si on ne souhaite pas se retrouver dans la situation précédente où des lignes sont "perdues" lors de la jointure, il faut utiliser une jointure externe ou `OUTER JOIN`. Il existe trois types de jointures externes : `LEFT OUTER JOIN`, `RIGHT OUTER JOIN` et `FULL OUTER JOIN`. Le mot-clé `OUTER` est facultatif.

- Après une jointure interne, le `LEFT JOIN` consiste à ajouter au résultat toutes les lignes de la première table (à gauche) n'ayant pas été associées à une ligne de la seconde table, en mettant des valeurs `NULL` dans toutes les colonnes de la seconde table.
- À l'inverse, après une jointure interne, le `RIGHT JOIN` consiste à ajouter au résultat toutes les lignes de la seconde table (à droite) n'ayant pas été associées à une ligne de la première table, en mettant des valeurs `NULL` dans toutes les colonnes de la première table.
- `FULL JOIN` est la combinaison de `LEFT JOIN` et `RIGHT JOIN`.

Exemple1.15

Réservations et nombre de personnes occupant une chambre, pour toutes les chambres de l'hôtel. Les chambres pour lesquelles aucune réservation n'a été enregistrée apparaissent également avec des valeurs `NULL` dans les colonnes `reservation` et `nbrPersonnes`.

```
SELECT c.numero, c.etage, o.reservation, o.nbrPersonnes
FROM Chambre c LEFT JOIN Occupation o ON c.numero = o.chambre;
```

▶ **Jointure naturelle.** Une jointure naturelle ou `NATURAL JOIN` est une jointure pour laquelle la condition de jointure est automatiquement réalisées sur les colonnes de même nom dans les deux tables. Une seule instance de chaque colonne est conservée dans le résultat. Tous les types précédents de jointures peuvent être réalisés avec une jointure naturelle : `NATURAL INNER JOIN`, `NATURAL LEFT OUTER JOIN`, `NATURAL RIGHT OUTER JOIN` et `NATURAL FULL OUTER JOIN`.

⚠ Si plusieurs colonnes portent le même nom dans les deux tables, elles seront toutes utilisées dans la condition de jointure implicite.

Exemple1.16

Numéro et prix des chambres réservées lors de la réservation numéro `'R12092345'`. Les deux requêtes ci-dessous sont équivalentes.

```
SELECT chambre, prix
FROM Reservation r JOIN Occupation o ON r.numero = o.reservation
                   NATURAL JOIN Prix p
WHERE r.numero = 'R12092345';

SELECT o.chambre, p.prix
FROM Reservation r JOIN Occupation o ON r.numero = o.reservation
                   JOIN Prix p ON o.chambre = p.chambre
WHERE r.numero = 'R12092345';
```

Notez qu'il est possible d'indiquer seulement `chambre` dans la première version, sans préciser la table, puisqu'une seule instance de la colonne est conservée par le `NATURAL JOIN`.

Exercice 3

1.7.2. Sous-requêtes

▶ **Sous-requête dans la clause `WHERE`.** Les sous-requêtes dans la clause `WHERE` permettent de filtrer le résultat en fonction du résultat d'une autre requête.

— Sous-requête délivrant une seule ligne :

```
WHERE col|expr comparateur (SELECT ... FROM ...)
```

— Sous-requête délivrant plusieurs lignes :

- Valeur correspondant à au moins une/aucune ligne de la sous-requête :
  ```
  WHERE col|expr [NOT] IN (SELECT ... FROM ...)
  ```
- Comparaison évaluée à vrai avec toutes les lignes de la sous-requête :
  ```
  WHERE col|expr comparateur ALL (SELECT ... FROM ...)
  ```
- Comparaison évaluée à vrai avec au moins une ligne de la sous-requête :
  ```
  WHERE col|expr comparateur ANY|SOME (SELECT ... FROM ...)
  ```

— Tester si la sous-requête délivre des lignes :

```
WHERE [NOT] EXISTS (SELECT ... FROM ...)
```

Exemple1.17

Numéro et étage des plus grandes chambres de l'hôtel.

```
SELECT numero, etage
FROM Chambre
WHERE superficie >= ALL (SELECT superficie
                         FROM Chambre);
```

Les sous-requêtes peuvent être utilisées dans la clause `WHERE` d'une commande `SELECT` mais aussi des commandes `UPDATE` et `DELETE`. Ces types de sous-requêtes peuvent également être utilisés dans la clause `HAVING`.

Exercice 4

▶ **Sous-requête dans la clause `FROM`.** Il est possible de faire une sous-requête dans le `FROM`. Dans ce cas, c'est le résultat de la sous-requête qui sera filtré par la requête principale.

EXEMPLE1.18

Tailles minimum et maximum des chambres de l'hôtel exposées au Sud et équipées de baignoire.

```
SELECT min(c.superficie), max(c.superficie)
FROM (SELECT superficie
      FROM Chambre
      WHERE exposition = 'S' AND baignoire = true) c;
```

⚠ La table intermédiaire obtenue par la sous-requête dans le `FROM` doit obligatoirement être nommée avec un alias.

▶ **Sous-requête dans un `INSERT`.** Il est possible d'utiliser une sous-requête pour générer les valeurs à insérer dans une table sur une requête de type `INSERT`.

EXEMPLE1.19

Ajout de toutes les chambres du 3ème étage à la réservation `'R0000123'`.

```
INSERT INTO Reservation
SELECT 'R000123', numero, capacite
FROM Chambre;
```

▶ **Sous-requête dans un `CREATE TABLE`.** Il est également possible de créer une table à partir du résultat d'une requête.

EXEMPLE1.20

```
CREATE TABLE OccupationAujourdhui AS (
    SELECT c.nom, c.noTelephone, r.chambre
    FROM Client c JOIN Reservation r ON c.id = r.client
                  JOIN Occupation o ON r.numero = o.reservation
    WHERE r.dateDebut <= CURRENT_DATE
              AND r.dateDebut + r.nbNuits >= CURRENT_DATE);
```

1.7.3. Agrégation

L'**agrégation** permet d'effectuer des calculs sur plusieurs lignes. Les fonctions d'agrégation ne peuvent pas être utilisées dans un `WHERE`.

▶ **Fonctions d'agrégation.**

Nom	Définition
`count(*)`	Compte le nombre de lignes, `0` si aucune ligne.
`count(col | exp)`	Compte le nombre de lignes dont la valeur de la colonne `col` ou de l'expression `exp` ne vaut pas `NULL`.
`count(DISTINCT col | exp)`	Compte le nombre de valeurs distinctes (non `NULL`) de la colonne `col` ou de l'expression `exp`.

Nom	Définition
`sum(col | exp)`	Somme des lignes.
`avg(col | exp)`	Valeur moyenne des lignes.
`max(col | exp)`	Valeur maximum des lignes.
`min(col | exp)`	Valeur minimum des lignes.
`stddev(col | exp)`	Écart-type des lignes.
`variance(col | exp)`	Variance des lignes.

⚠ `COUNT` tient compte des valeurs `NULL`. Les autres fonctions les ignorent.

Exemple1.21

— Nombre de chambres au 2ème étage.

```
SELECT count(*)
FROM Chambre
WHERE etage = 2;
```

— Clients ayant effectué au moins 2 réservations distinctes.

```
SELECT c.*
FROM Client c JOIN Reservation r ON c.id = r.client
GROUP BY c.*
HAVING count(*) >= 2;
```

Exercice 5

1.8. Vues

Une **vue** correspond à une requête de consultation stockée en base de données. Aucune donnée n'est stockée dans une vue. Lors de l'exécution de la requête, le SGBD remplace la vue par le code SQL correspondant. Une des utilités est de ne pas réécrire les requêtes souvent utilisées. Une vue s'interroge comme une table.

Exemple1.22

Vue correspondant au prix minimum et maximum des chambres en fonction de leur capacité.

```
CREATE VIEW Tarification AS
   SELECT c.capacite, min(p.prix) prixMin, max(p.prix) prixMax
   FROM Prix p JOIN Chambre c ON p.chambre = c.numero
   GROUP BY c.capacite;

SELECT prixMin, prixMax
FROM Tarification
WHERE capacite = 2;

DROP VIEW Tarification;
```

Comme la vue ne stocke aucune donnée, plusieurs interrogations successives tiendront compte des modifications des tables manipulées par la vue qui ont pu se produire entre temps.

▶ **Vue matérialisée.** Une **vue matérialisée** est une vue mais dont les données sont stockées.

Exemple1.23

```
CREATE MATERIALIZED VIEW Tarification AS
   SELECT c.capacite, min(p.prix) prixMin, max(p.prix) prixMax
   FROM Prix p JOIN Chambre c ON p.chambre = c.numero
   GROUP BY c.capacite;

DROP MATERIALIZED VIEW Tarification;
```

Les données sont stockées lors de la création de la vue matérialisée. Contrairement à une vue, plusieurs interrogations successives **ne** tiendront **pas** compte des modifications des tables manipulées par la vue qui ont pu se produire entre temps, sauf si la vue matérialisée est explicitement rafraîchie.

Pour rafraîchir manuellement une vue matérialisée, il faut utiliser la commande suivante :

```
REFRESH MATERIALIZED VIEW Tarification;
```

Les vues matérialisées sont utilisées lorsque l'obtention des données nécessite un temps de traitement important et que les données sont souvent accédées mais peu souvent modifiées.

1.9. Outils supplémentaires

▶ **CASE.** `CASE` permet d'évaluer une expression et de modifier la valeur retournée en fonction de la valeur de l'expression.

Exemple1.24

```
SELECT numero, etage,
    CASE
    WHEN superficie < 15 THEN 'économique'
    WHEN superficie >= 15 AND superficie < 25 THEN 'standard'
    WHEN superficie >= 25 AND superficie < 35 THEN 'supérieure'
    ELSE 'premium' END
FROM Chambre;
```

▶ **COALESCE.** `COALESCE` est une fonction qui prend autant d'arguments que l'on souhaite. Elle retourne le premier d'entre eux (de gauche à droite) qui ne vaut pas `NULL`.

⚠ Les types de tous les arguments doivent être compatibles.

Exemple1.25

```
SELECT numero, datePaiement, COALESCE(modePaiement, 'inconnu')
FROM Reservation
WHERE datePaiement IS NOT NULL;
```

▶ **LIMIT.** Lors d'un `SELECT`, il est possible de limiter le nombre de lignes retournées en rajoutant une clause `LIMIT`.

Exemple1.26

25 premiers clients par ordre alphabétique.

```
SELECT *
```

```
FROM Client
ORDER BY nom, prenom
LIMIT 25;
```

2. Optimisation des requêtes

Dans toute base de données, la rapidité d'accès aux données est un élément à prendre en compte. Les accès disque sont les éléments les plus pénalisant ainsi plus ces accès sont limités, plus le temps de réponse sera court. Nous supposerons ici que le MLD est correctement construit. La qualité des modèles de conception sera étudiée lors du prochain semestre.

2.1. Est-ce que la requête est bien écrite ?

Avant d'optimiser une requête, il est important de comprendre exactement le résultat attendu. Fréquemment, le développeur part d'une requête simple et la fait évoluer. Au fur et à mesure de ces évolutions, le développeur arrive à obtenir le résultat qu'il souhaite mais la requête n'est pas forcément propre et optimale.

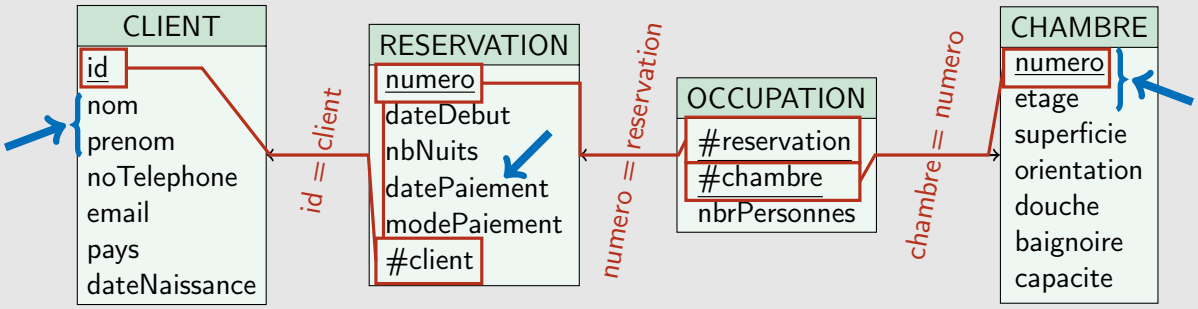
Il faut donc, entre autres, vérifier que :

- les tables (ou vues, ...) renseignées dans la clause `FROM` sont toutes nécessaires.
- des conditions de jointures n'ont pas été oubliées.
- si nécessaire, les sous-requêtes sont bien jointes à la requête principale.
- les conditions présentes dans la clause `WHERE` sont les plus restrictives possibles.

▶ **Jointures.** Lorsqu'une requête nécessite de faire des jointures entre plusieurs tables, il est important de se référer au MLD pour déterminer les tables et conditions de jointures nécessaires afin de ne pas oublier de conditions de jointures, ou d'utiliser des tables non nécessaires.

Exemple2.1

Clients (nom, prénom) et chambres (numéro, étage) concernés par une réservation qui n'a pas encore été réglée.



Il faut donc une jointure entre quatre tables : `Client`, `Reservation`, `Occupation` et `Chambre` et trois conditions de jointures :

1. `id = client` entre `Client` et `Reservation`,
2. `numero = reservation` entre `Reservation` et `Occupation`, et
3. `chambre = numero` entre `Occupation` et `Chambre`.

```sql
SELECT c.nom, c.prenom, ch.numero, ch.etage
FROM Client c JOIN Reservation r ON c.id = r.client
              JOIN Occupation o ON r.numero = o.reservation
              JOIN Chambre ch ON o.chambre = ch.numero
WHERE r.datePaiement IS NULL;
```

EXEMPLE2.2

Identifiants des clients ayant réservé la chambre `'204'`.

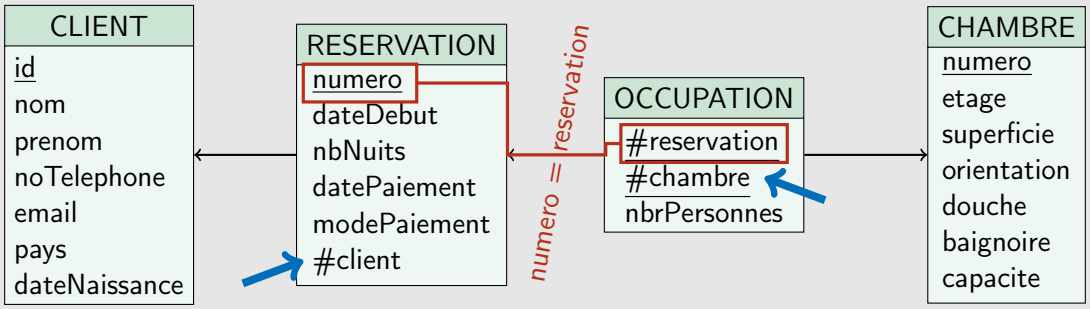
```sql
SELECT DISTINCT c.id
FROM Client c JOIN Reservation r ON c.id = r.client
              JOIN Occupation o ON r.numero = o.reservation
              JOIN Chambre ch ON o.chambre = ch.numero
WHERE ch.numero = '204';
```

Supposons que les tables `Client`, `Reservation`, `Occupation` et `Chambre` contiennent respectivement 1000, 2000, 2500 et 50 lignes.
Pour calculer cette requête, le SGBD peut réaliser les actions suivantes :

1. Calcul de la jointure entre la table `Client` et la table `Reservation` : Il faut d'abord calculer le produit cartésien entre ces deux tables, autrement dit, de générer $1000 \times 2000 = 2000000$ lignes.

   Ces lignes sont ensuite filtrées selon la condition de jointure. Etant donné qu'il n'y a qu'un seul client associé à une réservation, seules 2000 lignes sont conservées.
2. Calcul de la jointure entre les 2000 lignes obtenues à l'étape précédente et la table `Occupation` : De même, le produit cartésien est calculé produisant $2000 \times 2500 = 5000000$ lignes. Les lignes générées sont filtrées par la condition de jointure. À nouveau, une occupation ne concerne qu'une seule réservation, 2500 lignes sont conservées.
3. Calcul de la jointure entre les 2500 lignes générées à l'étape précédente et la table `Chambre` . Le produit cartésien génère $2500 \times 50 = 125\,000$ lignes. Une occupation ne concernant qu'une seule chambre, la condition de jointure filtre les lignes et il en reste 2500.
4. La condition du `WHERE` filtre les 2500 lignes générées à l'étape précédente.

Les étapes de calcul des jointures sont particulièrement coûteuses, générant temporairement plusieurs millions de lignes. Cependant deux tables ne sont pas nécessaires :



La requête suivante permet ainsi d'obtenir le même résultat beaucoup plus efficacement :

```sql
SELECT DISTINCT r.client
FROM Reservation r JOIN Occupation o ON r.numero = o.reservation
WHERE o.chambre = '204';
```

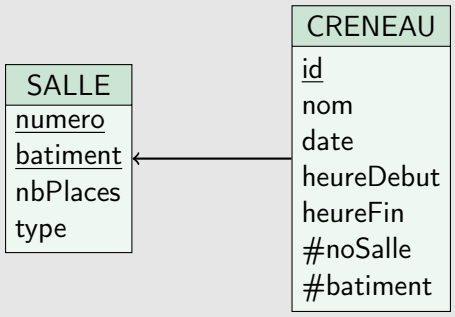
Avec cette requête, seules les étapes 2 et 4 de la requête précédente sont réalisées.

Exercice 6

▶ **Conditions de jointures.** Lorsqu'on écrit des requêtes complexes, des conditions de jointures peuvent être oubliées. En plus d'obtenir un résultat différent de celui attendu, la taille du résultat peut exploser. En effet, une jointure entre deux tables sans condition de jointure (ou avec une condition de jointure incomplète) mène à calculer un produit cartésien.

EXEMPLE2.3

Considérons le MLD suivant :



Nous voulons connaître les salles occupées ce jour. La requête suivante est utilisée :

```
SELECT DISTINCT s.numero, s.batiment
FROM Salle s JOIN Creneau c ON s.numero = c.noSalle
WHERE c.date = CURRENT_DATE;
```

La condition de jointure utilisée ne porte que sur le numéro de la salle. Or une salle est identifiée par son numéro ET son bâtiment. S'il y a par exemple une salle 101 dans le bâtiment A et dans le bâtiment B, et que seule la salle 101 du bâtiment A est occupée aujourd'hui, les deux seront tout de même retournée par la requête ci-dessus.

Pour éviter ces problèmes, il est conseillé de :

— Utiliser la syntaxe `JOIN ... ON ...` dans le `FROM` plutôt que la syntaxe (dépréciée) avec des virgules entre les tables dans le `FROM` et les conditions de jointures placées dans le `WHERE`.

— Dissocier les conditions de jointures (entre deux tables) et les conditions de filtrage (locales à une table) en placant les premières dans les `ON` correspondant du `FROM` et les secondes dans le `WHERE`.

— Privilégier, lorsque cela est pertinent, des clés primaires simples (formées d'un seul attribut) plutôt que des clés primaires complexes (formées de plusieurs attributs). Cela peut par exemple être un id incrémenté automatiquement (de type `serial` ou en utilisant des séquences personnalisées).

Exercice 7

▶ **Restriction des données.** Lorsque c'est possible, il faut limiter les quantités de données manipulées en filtrant les colonnes (via une liste de colonnes sélectionnées dans le `SELECT` plutôt que *) et/ou les lignes (via des conditions dans le `WHERE` ou le `HAVING`) retournées par la requête ou par des sous-requêtes.

Exercice 8

▶ **Utilisation de sous-requêtes `EXISTS`.** Dans certaines situations, utiliser une sous-requête `EXISTS` permet d'accélérer le traitement d'une requête. En effet, dès qu'une ligne peut être retournée par la sous-requête, le calcul du `EXISTS` est interrompu et il retourne vrai.

Cette technique est particulièrement utile lorsqu'on manipule une grande quantité de données avec une sous-requête de type `IN` ou avec une sous-requête corrélée.

Exemple2.4

Pour trouver les clients ayant une réservation commençant aujourd'hui, il peut être plus rapide d'utiliser la seconde requête ci-dessous plutôt que la première.

```
SELECT *
FROM Client
WHERE id IN (SELECT client
             FROM Reservation
             WHERE dateDebut = CURRENT_DATE);

SELECT *
FROM Client c
WHERE EXISTS (SELECT 1
              FROM Reservation r
              WHERE c.id = r.client AND dateDebut = CURRENT_DATE);
```

Exercice 9

2.2. Index

Un **index** est une structure de données permettant d'accéder plus rapidement aux données recherchées. Un index contient l'ensemble des valeurs contenues dans la colonne indexée. Ces valeurs sont ordonnées pour permettre une recherche plus rapide. Pour chaque valeur, l'index contient des pointeurs vers les véritables lignes de la table. Un index peut concerner plusieurs colonnes. Dans ce cas, l'ordre des colonnes lors de la création de l'index est important.

L'utilisation d'un index facilite le filtrage des lignes lors d'un `SELECT` mais peut également servir lors d'un `UPDATE` ou `DELETE` avec des conditions. Il peut également accélérer les requêtes avec jointures si l'index est défini sur une partie de la condition de jointure. Enfin, il peut également servir pour la vérification de contraintes comme les contraintes d'unicité par exemple.

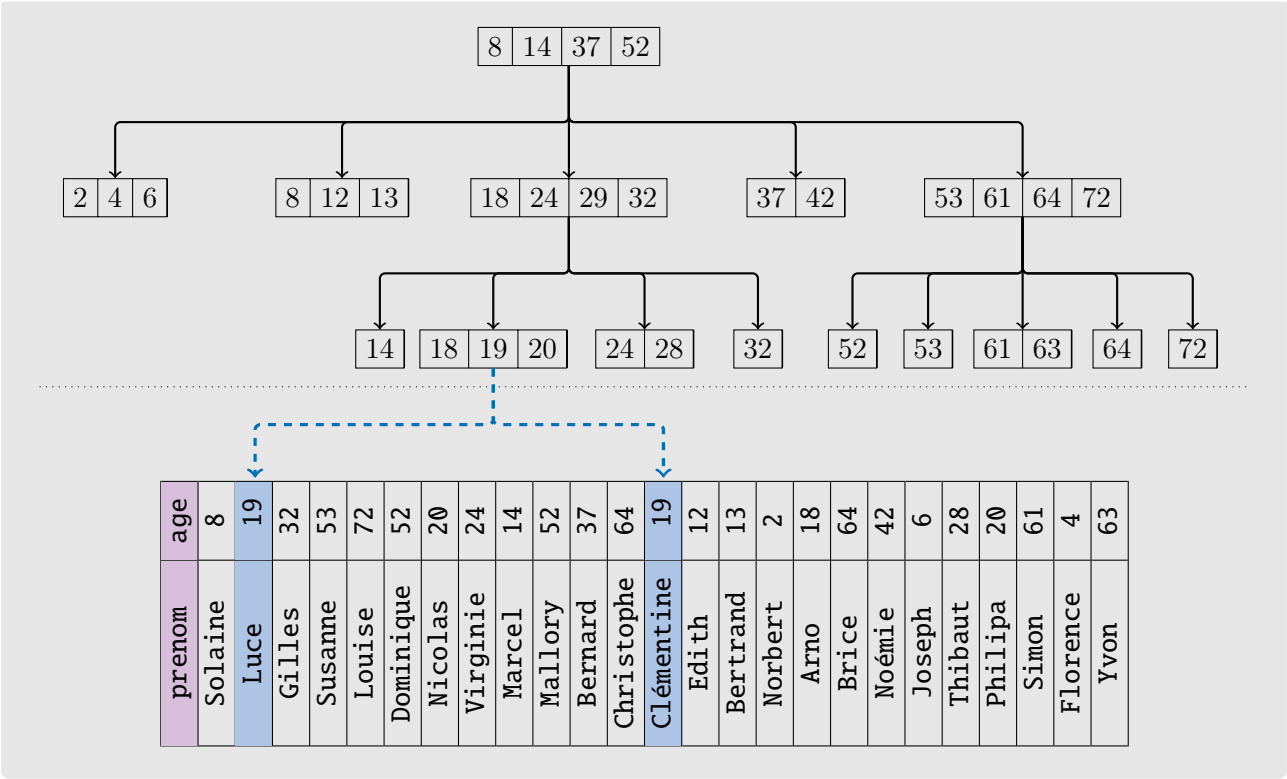
⚠ Maintenir un index est coûteux. Il doit être mis à jour à chaque modification de la table. Il faut donc bien choisir les colonnes à indexer et il est préférable de supprimer les indexes qui sont peu ou pas utilisés.

Notez que, les index ne servant qu'à optimiser l'accès aux données, ils ne font pas partie de la norme SQL. Il peut donc y avoir des différences importantes dans la façon dont ils sont gérés et dans la syntaxe de création d'un index d'un système de gestion de base de données à l'autre.

▶ **Index B-tree.** Ce type d'index est le plus courant. Un index **B-tree** est organisé sous forme d'arbre. Les terminaisons s'appellent des feuilles. Les éléments intermédiaires, reliés aux feuilles par les branches sont les nœuds.

Exemple2.5

Le schéma ci-dessous est un exemple de B-tree.
Chaque nœud contient des valeurs seuils et des pointeurs. Pour le premier nœud, il faut lire : Pour les valeurs inférieures à 8, suivre le pointeur vers le premier nœud fils. Pour les valeurs de 8 (inclue) à 14 (exclue), suivre le pointeur vers le second nœud fils, ...
Pour trouver une ligne dont la valeur est 19, il faut donc partir du haut de l'arbre, puis aller dans le 4ème nœud et enfin le 8ème nœud si l'on numérote les nœuds de gauche à droite et de haut en bas.



▶ **Utilisation des index.**

— Création d'un index : `CREATE INDEX idx ON ma_table(col);`

— Création d'un index sur plusieurs colonnes (⚠ L'ordre des colonnes est primordial.) :

```
CREATE INDEX idx ON ma_table(col1, col2, ...);
```

— Création d'un index sur une expression : `CREATE INDEX idx ON ma_table(exp(col1, ...));`

(⚠ Il faut généralement entourer l'expression de parenthèses.)

— Suppression d'un index : `DROP INDEX idx;`

▶ **Autres types d'index.** Par défaut, l'index créé par `CREATE INDEX` est un index B-tree qui a de nombreux avantages, notamment en terme de performances moyennes et de concurrence d'accès. Mais pour certains types de requêtes, il est parfois préférable d'utiliser d'autres d'autres types d'index proposés par PostgreSQL : index Hash, index GiST, index BRIN, index GIN ...

Un index `UNIQUE` permet de provoquer une erreur lors d'une insertion de doublons :

```
CREATE UNIQUE INDEX idx ON ma_table(col);
```

On préférera cependant utiliser des contraintes d'unicité (qui de toute façon, entraîneront la création d'un index).

2.2.1. Quelles colonnes indexer ?

▶ **Clés primaires.** PostgreSQL crée automatiquement des index sur les colonnes concernées par les contraintes `UNIQUE` ou `PRIMARY KEY`. En effet, l'index est un moyen rapide de vérifier l'unicité d'une valeur.

▶ **Clés étrangères.** Indexer les colonnes faisant partie d'une clé étrangère est utile pour les jointures. En effet, la colonne référencée est forcément indexée (puisqu'une clé étrangère fait références à des colonnes ayant une contrainte `UNIQUE` ou `PRIMARY KEY`) ce qui permet donc de la parcourir de façon

triée. En indexant également la colonne qui fait la référence, on peut également parcourir de façon triée cette colonne et donc permettre de calculer la jointure beaucoup plus rapidement.

De même, lors de modifications de la table mère, une indexation des colonnes faisant partie de la clé étrangère référençant cette table permet de vérifier plus rapidement si la modification doit être empêchée (sous peine de violer la contrainte de clé étrangère) ou propagée (en cas d'option `CASCADE`).

▶ **Clause `WHERE`.** Ajouter un index sur une colonne concernée par une clause `WHERE` peut grandement améliorer les performances d'une requête.

EXEMPLE2.6

Un index sur la colonne `telephone` de `Client` peut être utile pour cette requête :

```
SELECT *
FROM Client
WHERE telephone = '0600112233';
```

▶ **Clause `SELECT`.** Certaines requêtes peuvent être optimisées en posant un index sur une colonne de la clause `SELECT` car l'index peut éviter d'aller chercher les blocs de données et n'utiliser alors que les blocs d'index.

EXEMPLE2.7

Un index sur `(email, id)` de `Client` permet d'exécuter la requête suivante en utilisant que l'index sans lire le contenu de la table elle-même.

```
SELECT id
FROM Client
WHERE email = 'john.doe@gmail.com';
```

2.2.2. *Quels types de colonnes utiliser ?*

Il est généralement préférable d'utiliser des colonnes dont le type nécessite peu de mémoire comme le type `numeric` pour les index et en particulier pour les clés primaires. En effet, l'index se révèle le plus efficace lorsqu'il peut entièrement être chargé dans la mémoire vive. Utiliser des types nécessitant peu de mémoire, comme un `numeric`, permet donc de maximiser les chances de charger cet index en mémoire, contrairement à l'utilisation de chaînes de caractères par exemple.

2.3. Plan d'exécution

Lorsqu'une requête SQL est soumise, PostgreSQL définit la méthode à utiliser pour obtenir les données demandées. Il se base sur différentes informations (nombre de lignes, taille de ligne, ...) et définit un **plan d'exécution**. Plus le coût d'un plan d'exécution est faible, plus l'exécution de la requête sera rapide.

Dans `psql`, il est possible de voir le plan d'exécution d'une requête en préfixant une commande par `EXPLAIN`. Par exemple, `EXPLAIN SELECT * FROM Client;`

Dans le plan d'exécution, il y a les informations suivantes (liste non exhaustive) :

Ligne du plan	Explication
`Seq Scan`	Lecture de toutes les lignes de la table.
`Index Scan`	Parcours de la table en utilisant un index.

Ligne du plan	Explication
`Hash Join`	Pour les deux tables concernées par la jointure, des valeurs de hash sont calculées à partir des clés et les hachés sont comparés. `Hash Cond` = Condition utilisée pour comparer les hachés lors de la jointure.
`Hash Aggregate`	Pour le calcul d'une fonction d'agrégation avec **GROUP BY**, des valeurs de hash sont calculées sur les colonnes servant de critère de regroupement pour faciliter le tri des lignes et leur regroupement avant l'agrégation. `Group Key` = Critère de regroupement des lignes.
`Sort`	Tri des lignes. `Sort Key` = Critère de tri des lignes.

En rajoutant l'option **ANALYZE** après **EXPLAIN**, la requête est réellement exécutée et non pas seulement planifiée. Cela permet d'avoir des informations supplémentaires comme le temps d'exécution et l'espace mémoire utilisé.

Exemple2.8

```
EXPLAIN
SELECT DISTINCT o.chambre
FROM Client c JOIN Reservation r ON c.id = r.client
              JOIN Occupation o ON o.reservation = r.numero;
```

Donne le plan d'exécution suivant :

```
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 HashAggregate  (cost=66.54..68.54 rows=200 width=16)
   Group Key: o.chambre
   ->  Hash Join  (cost=38.00..63.94 rows=1040 width=16)
         Hash Cond: (r.client = c.id)
         ->  Hash Join  (cost=24.18..47.33 rows=1040 width=40)
               Hash Cond: (o.reservation = r.numero)
               ->  Seq Scan on occupation o  (cost=0.00..20.40 rows=1040 width=40)
               ->  Hash  (cost=16.30..16.30 rows=630 width=48)
                     ->  Seq Scan on reservation r  (cost=0.00..16.30 rows=630 width=48)
         ->  Hash  (cost=11.70..11.70 rows=170 width=24)
               ->  Seq Scan on client c  (cost=0.00..11.70 rows=170 width=24)
(11 lignes)
```

En rajoutant **ANALYZE**, on obtient plus d'informations :

```
EXPLAIN ANALYZE
SELECT DISTINCT o.chambre
FROM Client c JOIN Reservation r ON c.id = r.client
              JOIN Occupation o ON o.reservation = r.numero;
```

```
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
HashAggregate  (cost=66.54..68.54 rows=200 width=16) (actual time=0.186..0.193 rows=7
                                                                                  loops=1)
   Group Key: o.chambre
   ->  Hash Join  (cost=38.00..63.94 rows=1040 width=16) (actual time=0.152..0.176
                                                                  rows=15 loops=1)
         Hash Cond: (r.client = c.id)
         ->  Hash Join  (cost=24.18..47.33 rows=1040 width=40) (actual time=0.084..0.100
                                                                  rows=15 loops=1)
               Hash Cond: (o.reservation = r.numero)
               ->  Seq Scan on occupation o  (cost=0.00..20.40 rows=1040 width=40)
                                             (actual time=0.011..0.015 rows=15 loops=1)
               ->  Hash  (cost=16.30..16.30 rows=630 width=48) (actual time=0.047..0.048
```

```
                                                                    rows=12 loops=1)
                    Buckets: 1024  Batches: 1  Memory Usage: 9kB
                    ->  Seq Scan on reservation r  (cost=0.00..16.30 rows=630 width=48)
                                                   (actual time=0.009..0.014 rows=12 loops=1)
        ->  Hash  (cost=11.70..11.70 rows=170 width=24) (actual time=0.044..0.045
                                                                    rows=10 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 9kB
              ->  Seq Scan on client c  (cost=0.00..11.70 rows=170 width=24) (actual
                                                     time=0.031..0.036 rows=10 loops=1)
 Planning Time: 0.540 ms
 Execution Time: 0.374 ms
(15 lignes)
```

2.3.1. *Gains et pertes.*

Les index sont des structures de données organisées. Lors d'une modification sur les données de la table concernées par cet index, l'index est modifié afin de prendre en compte cette modification.

Exemple2.9

```
UPDATE ma_table SET id = id + 100 WHERE id < 20;
```

S'il y a un index sur la colonne `id` de `ma_table`, pour chaque ligne de `ma_table` qui est modifiée, une entrée de l'index est supprimée et une autre ajoutée. Ces modifications de l'index peuvent pénaliser les performances de la base.

▶ **Avantages.** Les performances en lecture (lors d'un `SELECT`) sont améliorées ainsi que celles d'un `DELETE` ou `UPDATE` (s'il y a une clause `WHERE`) pour récupérer les lignes à modifier ou supprimer.

▶ **Inconvénients.** Il peut y avoir des pertes de performances dans certains cas lors de modifications de la base, notamment lorsqu'il y a un grand nombre de modifications amenant à modifier l'index.

Exemple2.10

Soit une table contenant 1 million de lignes.

— Si on supprime une seule ligne identifiée par l'index : **gain !**
  Le gain de performance apporté par l'index pour trouver la ligne à supprimer est beaucoup plus important que le coût de la modification de l'index pour cette seule ligne.

— Si on supprime 70% des lignes : **perte !**
  Le nombre de modifications nécessaires dans l'index devient plus coûteux que le gain de performance que l'on a avec l'index pour trouver les lignes à supprimer.

2.3.2. *Statistiques*

Pour concevoir le plan d'exécution, PostgreSQL se base sur des statistiques concernant le coût d'utilisation des index, le nombre de lignes dans les tables, ... Si PostgreSQL n'a pas d'informations sur des objets, ou si les statistiques sont obsolètes, il peut arriver que le plan d'exécution ne soit pas optimal. Mettre à jour les statistiques peut aider à obtenir de meilleurs plans d'exécution.

La commande `ANALYZE` permet de récolter des statistiques sur les tables listées.

```
ANALYZE table1, table2 ...;
```

Exemple2.11

Soit la requête suivante :

```
SELECT nom
FROM Athlete
WHERE char_length(nom) > 20;
```

Le plan d'exécution est :

```
                                    QUERY PLAN
-------------------------------------------------------------------------------------
 Seq Scan on athlete  (cost=0.00..275.70 rows=3882 width=8) (actual time=0.200..3.952
                                                                      rows=17 loops=1)
   Filter: (char_length((nom)::text) > 20)
   Rows Removed by Filter: 11630
 Planning Time: 0.157 ms
 Execution Time: 3.990 ms (5 lignes)
```

Nous rajoutons un index sur le calcul nécessaire à la condition utilisée dans le **WHERE** :

```
CREATE INDEX idx ON Athlete(char_length(nom));
```

Le plan d'exécution est désormais :

```
                                    QUERY PLAN
-------------------------------------------------------------------------------------
 Bitmap Heap Scan on athlete  (cost=78.37..237.60 rows=3882 width=8) (actual
                                                    time=0.146..0.214 rows=17 loops=1)
   Recheck Cond: (char_length((nom)::text) > 20)
   Heap Blocks: exact=14
   ->  Bitmap Index Scan on idx  (cost=0.00..77.40 rows=3882 width=0) (actual
                                                    time=0.115..0.116 rows=17 loops=1)
         Index Cond: (char_length((nom)::text) > 20)
 Planning Time: 0.207 ms
 Execution Time: 0.278 ms
(7 lignes)
```

Exemple2.12

Soit la requête suivante :

```
SELECT a1.pays, a1.nom, a1.prenom
FROM Athlete a1
WHERE dateNaiss <= (SELECT min(a2.dateNaiss)
                    FROM Athlete a2
                    WHERE a1.pays = a2.pays)
ORDER BY a1.pays;
```

Le plan d'exécution est le suivant :

```
                                    QUERY PLAN
-------------------------------------------------------------------------------------
 Sort  (cost=2874258.78..2874268.49 rows=3882 width=19) (actual time=15118.520..15118.533
                                                                    rows=206 loops=1)
   Sort Key: a1.pays
   Sort Method: quicksort  Memory: 40kB
   ->  Seq Scan on athlete a1  (cost=0.00..2874027.37 rows=3882 width=19) (actual
                                                  time=94.933..15118.060 rows=206 loops=1)
         Filter: (datenaiss <= (SubPlan 1))
         Rows Removed by Filter: 11441
         SubPlan 1
           ->  Aggregate  (cost=246.73..246.74 rows=1 width=4) (actual time=1.297..1.297
                                                              rows=1 loops=11647)
                 ->  Seq Scan on athlete a2  (cost=0.00..246.59 rows=57 width=4) (actual
```

```
                                                    time=0.023..1.280 rows=258 loops=11647)
                      Filter: (a1.pays = pays)
                      Rows Removed by Filter: 11389
 Planning Time: 0.352 ms
 Execution Time: 15118.609 ms
 (13 lignes)
```

Cette requête est longue (15 secondes). Nous rajoutons un index sur la colonne `dateNaiss`.

```
CREATE INDEX idx1 ON Athlete(dateNaiss);
```

Le nouveau plan d'exécution est le suivant :

```
                                  QUERY PLAN
-------------------------------------------------------------------------------------------
 Sort  (cost=164003.99..164013.70 rows=3882 width=19) (actual time=1123.474..1123.486
                                                                        rows=206 loops=1)
   Sort Key: a1.pays
   Sort Method: quicksort  Memory: 40kB
   ->  Seq Scan on athlete a1  (cost=0.00..163772.57 rows=3882 width=19) (actual
                                                    time=26.517..1123.244 rows=206 loops=1)
         Filter: (datenaiss <= (SubPlan 2))
         Rows Removed by Filter: 11441
         SubPlan 2
           ->  Result  (cost=14.03..14.04 rows=1 width=4) (actual time=0.096..0.096 rows=1
                                                                        loops=11647)
                 InitPlan 1 (returns $1)
                   ->  Limit  (cost=0.29..14.03 rows=1 width=4) (actual time=0.096..0.096
                                                                rows=1 loops=11647)
                         ->  Index Scan using idx1 on athlete a2  (cost=0.29..770.02
                                                  rows=56 width=4) (actual time=0.095..0.095
                                                  rows=1 loops=11647)
                               Index Cond: (datenaiss IS NOT NULL)
                               Filter: (a1.pays = pays)
                               Rows Removed by Filter: 252
 Planning Time: 0.565 ms
 Execution Time: 1123.559 ms
(16 lignes)
```

C'est beaucoup mieux (environ 1 seconde) ! On peut encore améliorer les performances en ajoutant un index sur la colonne `pays`.

```
CREATE INDEX idx2 ON Athlete(pays);
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------------------
Index Scan using idx2 on athlete a1  (cost=0.29..164269.97 rows=3882 width=19) (actual
                                                    time=9.823..1121.820 rows=206 loops=1)
   Filter: (datenaiss <= (SubPlan 2))
   Rows Removed by Filter: 11441
   SubPlan 2
     ->  Result  (cost=14.03..14.04 rows=1 width=4) (actual time=0.096..0.096 rows=1
                                                                        loops=11647)
           InitPlan 1 (returns $1)
             ->  Limit  (cost=0.29..14.03 rows=1 width=4) (actual time=0.095..0.095
                                                                rows=1 loops=11647)
                   ->  Index Scan using idx1 on athlete a2  (cost=0.29..770.02 rows=56
                             width=4) (actual time=0.095..0.095 rows=1 loops=11647)
                         Index Cond: (datenaiss IS NOT NULL)
                         Filter: (a1.pays = pays)
                         Rows Removed by Filter: 252
 Planning Time: 0.553 ms
```

```
Execution Time: 1121.880 ms
(13 lignes)
```

Exercice 10

3. Concurrence d'accès

3.1. Transaction

Une **transaction** est une unité logique de traitements regroupant un ensemble d'opérations élémentaires. Une transaction effectue un ensemble d'ordres SQL, par exemple `SELECT` puis `UPDATE` puis `DELETE`. Ce n'est qu'à la fin de ces opérations que l'utilisateur décide de **valider** son travail ou de l'**annuler**. Les mises à jour ne sont effectives qu'au moment de la validation, sinon il y a annulation des modifications.

- **Début de transaction :** Une nouvelle transaction commence dès l'exécution de la première commande agissant sur des données ou prévoyant d'agir sur les données.
- **Fin de transaction :** Il y a deux moyens pour terminer une transaction :
  1). **Validation** d'une transaction par :
  - Exécution explicite de la commande "`COMMIT;`"
  - Exécution implicite de la commande "`COMMIT;`", par exemple si le paramètre `AUTOCOMMIT` est mis à `ON` (ce qui est le cas par défaut dans `psql`).
  - Fin normale d'un programme ou d'une session `psql`, `COMMIT` implicite.

  2). **Annulation** d'une transaction par :
  - Exécution explicite de la commande "`ROLLBACK;`".
  - Fin anormale d'un programme ou d'une session `psql`, donc `ROLLBACK` implicite.

3.2. ACID

En 1983, Harder et Reuter ont défini un modèle de transactions pour les actions effectuées sur une base de données. Ce modèle leur permet d'introduire 4 concepts fondamentaux pour garantir la cohérence et l'intégrité des données.

▶ **Atomicité :** Gestion des modifications de données sous formes de transactions. S'il y a par exemple, un problème technique, l'ensemble de la transaction est annulée. Une transaction se fait donc complètement ou ne se fait pas du tout.

▶ **Cohérence :** Une transaction fait passer la base d'un état cohérent à un autre état cohérent. Différents mécanismes permettent de garantir le retour dans un état cohérent en cas de panne du SGBD. Ce principe s'applique aussi au niveau des lectures.

▶ **Isolation :** Toute transaction doit s'exécuter comme si elle était seule. Il ne peut y avoir aucune dépendance entre les transactions.

▶ **Durabilité :** Lorsque la base atteint un état cohérent, cet état est enregistré et pérenne, même en cas de panne matériel.

3.3. Lecture cohérente et ROLLBACK

Lorsqu'une commande modifie les données, la nouvelle valeur vient écraser l'ancienne. Mais lorsque la transaction n'est pas terminée, PostgreSQL ne peut pas savoir si la transaction va être validée ou s'il elle va être annulée. Il faut donc que la base de données conserve la valeur avant modification pour pouvoir revenir en arrière en cas de ROLLBACK. Le mécanisme utilisé par PostgreSQL est un système multiversions (MVCC pour *Multiversion Concurrency Control*) qui utilise des labels.

▶ **xmin et xmax.** PostgreSQL conserve les différentes versions d'une ligne. Chaque ligne est labelisée avec deux valeurs `xmin` et `xmax`.

1. Lorsqu'une transaction crée une ligne (avec un INSERT), `xmin` prend comme valeur l'id de cette transaction.
2. Lorsqu'une transaction supprime une ligne (avec un DELETE), `xmax` prend comme valeur l'id de cette transaction indiquant que la ligne a été supprimée, mais elle ne l'est pas vraiment.
3. Lorsqu'une transaction modifie une ligne (avec un UPDATE), une version de la ligne est supprimée (comme dans le point 2) et une autre version de la ligne est créée (comme dans le point 1).

En plus de `xmin` et `xmax`, des booléens permettent d'indiquer s'il y a eu un commit ou un rollback validant/invalidant ces lignes. Si la transaction est finalement annulée, les modifications sont labelisée comme étant annulée et les requêtes suivantes n'en tiendront pas compte.

Les valeurs `xmin` et `xmax` définissent ainsi les transactions qui peuvent voir chaque version : seules les transactions dont l'id est compris entre `xmin` et `xmax` peuvent les voir.

(Nous supposons ici que chaque nouvelle transaction a un id supérieur aux transactions précédentes et donc que les ids grossissent indéfiniment. PostgreSQL possède un mécanisme permettant de gérer les ids des transactions avec un espace mémoire fini, mais nous ne rentrerons pas dans le détail de ce mécanisme dans ce cours.)

⚠ Comme les anciennes versions des lignes sont sauvegardées et ne sont pas supprimées, l'espace mémoire occupé par une base de données PostgreSQL grossit à chaque modification des données. Cependant, PostgreSQL dispose d'un mécanisme appelé **autovacuum** qui permet automatiquement de nettoyer les vieilles versions de lignes qui ne sont plus nécessaires.

3.3.1. *Lecture cohérente*

Lorsqu'une transaction est en cours, les autres sessions ne doivent pas voir les modifications effectuées par la transaction tant que cette dernière n'est pas validée.

Exemple3.1

Tps	Session S1	Session S2
T1	`UPDATE Client SET noTelephone='0601020304' WHERE id='C123';` Modification de la 12045$^{\text{ième}}$ ligne de la table `Client`. Deux versions de la ligne sont conservées : avant et après modification.	

Tps	Session S1	Session S2
T2		`SELECT * FROM Client WHERE id='C123';` La valeur lue est celle de la version avant modification. La version après modification est ignorée, car elle n'a pas encore été validée
T3	Fin de la transaction	

Notez que les temps T1 et T2 peuvent être inversés et le fonctionnement reste le même. Les données sont cohérentes à l'instant T1. Que le `SELECT` soit fait avant ou après l'`UPDATE`, toutes les données retournées à S2 sont celles qui étaient présentes à l'instant T1.

C'est ce principe qui permet d'effectuer une **lecture cohérente**.

3.3.2. Exemple de transaction

Soit la table suivante :

```
CREATE TABLE Info(
    id  numeric PRIMARY KEY,
    msg varchar(50)
);
```

Tps	Session S1	Session S2
T1	`SELECT count(*) FROM Info;` 0 ligne	
T2		`INSERT INTO Info VALUES (1,'blabla');` 1 ligne créée
T3	`SELECT count(*) FROM Info;` 0 ligne **Les modifications effectuées par S2 ne sont pas visibles car S2 n'a pas terminé sa transaction.**	
T4		`COMMIT;`
T5	`SELECT count(*) FROM Info;` 1 ligne **Les données sont visibles !**	

3.4. Gestion des accès concurrents

Dans une architecture multi-utilisateurs, la modification simultanée des mêmes données étant impossible, le système devra assurer :

— Un mécanisme de concurrence d'accès aux données.
— Un mécanisme de lecture cohérente.

PostgreSQL assure cela grâce à son système de multiversions et grâce à l'utilisation de verrous. Il n'existe pas de concurrence d'accès aux données en lecture (avec un `SELECT` simple). Un utilisateur qui lit une donnée n'interfère pas avec une transaction.

▶ **Catégories de commandes.** Il existe 3 catégories de commandes dans une base de données.

— **DDL : Data Definition Language** (en français, LDD : Langage de Définition de Données). Cette catégorie regroupe toutes les commandes permettant de **gérer la structure** des données. Ces commandes sont `CREATE`, `ALTER`, `DROP`, `GRANT`, `REVOKE`, ... (plus simplement, toutes les commandes non DML).

Exemple3.2

```
ALTER TABLE Client ADD carteFidelite char(8);
```

— **DML : Data Manipulation Language** (en français : Language de Manipulation de Données). Cette catégorie regroupe toutes les commandes permettant de **manipuler les données**. Ces commandes sont `INSERT`, `UPDATE` et `DELETE`. La commande n'affecte pas la structure des données dans la base, mais uniquement le contenu.

Exemple3.3

```
UPDATE Reservation
SET datePaiement = CURRENT_DATE,
    modePaiement = 'CB'
WHERE numero = 'R00123';
```

— `SELECT` : Cette commande est à part car elle ne modifie en rien la structure ou les données mais ne fait que de la consultation.

Exemple3.4

```
SELECT nom, prenom
FROM Client
WHERE pays != 'France';
```

▶ **Verrous.** Le but d'un verrou est d'empêcher la modification d'une donnée par plusieurs sessions en même temps. Le verrou est donc posé par la première session qui le demande et les autres sessions demandant la modification de la donnée déjà verrouillée, attendent. Une fois que la session détenant le verrou termine sa transaction (par `COMMIT` ou `ROLLBACK`), elle relâche tous ses verrous. La session suivante, qui était bloquée par le verrou, pose alors son verrou et peut travailler. Il existe deux niveaux de verrouillage :

— Au niveau de la ligne (DML lock)
— Au niveau de la table (DDL lock)

La possibilité de lever un verrou est attribuée en mode *FIFO* (*First In First Out*). Soit une session S1 qui, lors d'une transaction pose un verrou. Soit plusieurs sessions bloquées par le verrou ci-dessus, arrivées dans l'ordre A, B, C, D, ... Lorsqu'un verrou est libéré par S1, c'est la première session qui a été bloquée qui pose le nouveau verrou (donc la session A). Lorsque A libérera le verrou, ce sera au tour de B et ainsi de suite.

▶ **Verrou de ligne.** Chaque ligne peut être verrouillée individuellement. Le verrou empêche donc la modification d'une ou plusieurs lignes mais permet la modification des autres lignes de la table. Il est aussi appelé verrou de manipulation de données ou DML lock.

Lorsqu'un tel verrou est positionné sur une ligne, la base de données va :

1. Poser un verrou de manipulation de données sur la ligne, pour empêcher les données d'être modifiées.

2. Poser un verrou de définition de données sur la table pour empêcher les modifications de structure de la table.

▶ **Verrou de table.** Un verrou de table est un verrou empêchant la modification de la structure (`ALTER`), ainsi que la suppression de la table. Ce verrou n'agit pas au niveau des lignes, ce qui signifie que les commandes `INSERT`, `UPDATE`, `DELETE`, ... sont toujours possibles. Ce type de verrou est aussi appelé verrou de définition de données ou DDL lock.

▶ **Légende.** Dans les tableaux ci-après, les actions de verrouillage, de déverrouillage et d'attente d'un verrou sont représentées ainsi :

— 

: verrou de ligne, `xx` indique l'identifiant de la ligne concernée

— 

: libération du verrou de ligne concernant la ligne `xx`

— 

: indique que la session est bloquée par un verrou de ligne concernant la ligne `xx`

— 
: verrou de table

— 
: libération du verrou de table

— 
: indique que la session est bloquée par un verrou de table

Exemple3.5

Soit la table suivante :

```
CREATE TABLE Info(
    id  numeric PRIMARY KEY,
    msg varchar(50)
);
```

Tps	Session S1	Session S2	Session S3

T1	`INSERT INTO Info VALUES (10,'val1');`
	1 ligne créée
	`INSERT INTO Info VALUES (11,'val2');`
	1 ligne créée
	`INSERT INTO Info VALUES (12,'val3');`
	1 ligne créée
	`COMMIT;`
	Validation effectuée		

T2		`UPDATE Info SET msg='new1' WHERE id=10;` 1 ligne modifiée 	

Tps	Session S1	Session S2	Session S3
T3		10	`UPDATE Info SET msg='new2' WHERE id=11;` 1 ligne modifiée 11
T4		10	`UPDATE Info SET msg='new11' WHERE id=10;` 11 10
T5	`ALTER TABLE Info ADD quand DATE;`	10	11 10
T6		`COMMIT;` 10	11 10
T7			1 ligne modifiée 11 10
T8			`ROLLBACK;` Annulation effectuée 11 10
T9	Table modifiée		
T10	`COMMIT;`		

3.5. Interblocage

L'**interblocage** (ou **deadlock**) de transactions arrive quand des sessions s'interbloquent les unes avec les autres. Dans ce cas, PostgreSQL met fin à une des deux sessions pour permettre à l'autre de continuer.

Exemple3.6

Soit la table suivante :

```
CREATE TABLE Info(
    id  numeric PRIMARY KEY,
    msg varchar(50)
);
```

```
INSERT INTO Info VALUES (10, 'val1');
INSERT INTO Info VALUES (11, 'val2');

COMMIT;
```

Tps	Session S1	Session S2

T1	`UPDATE Info`
	`SET msg='new1'`
	`WHERE id=10;`
	1 ligne modifiée
	10 🔒	

T2		`UPDATE Info`
		`SET msg='new2'`
		`WHERE id=11;`
		1 ligne modifiée
		11 🔒

T3	`UPDATE Info`
	`SET msg='new3'`
	`WHERE id=11;`
	10 🔒 11 🚫	

T4		`UPDATE Info`
		`SET msg='new4'`
		`WHERE id=10;`
		11 🔒 10 🚫

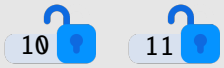
T5	10 🔒 11 🚫	`ERREUR: interblocage (deadlock) dé tecté`
		`DÉTAIL : Le processus 122628 attend ShareLock sur transaction 1518 ; bloqué par le processus 121876.`
		11 🔓

T6	1 ligne modifiée 10 🔒 11 🔒	
T7	10 🔒 11 🔒	`ROLLBACK;`

T8	10 🔒 11 🔒	`SELECT * FROM Info;`
		`id | msg`
		`--+--`
		`10 | val1`
		`11 | val2`
		`(2 lignes)`

Tps	Session S1	Session S2

T9	`SELECT * FROM Info;`
	`id | msg`
	`--+--`
	`10 | new1`
	`11 | new3`
	`(2 lignes)`
	 	

T10	`COMMIT;` 	

Notez qu'après une erreur dans une transaction, comme c'est le cas ici dans la transaction 2 à cause du deadlock, PostgreSQL empêche l'exécution de toute autre commande jusqu'à la fin de la transaction :

```
ERREUR:  la transaction est annulée, les commandes sont ignorées jusqu à la
fin du bloc de la transaction
```

Exercice 11 **Exercice 12**

4. PL/pgSQL

Dans ce chapitre, nous revenons sur les notions de PL/pgSQL vues en première année et nous introduisons des notions avancées.

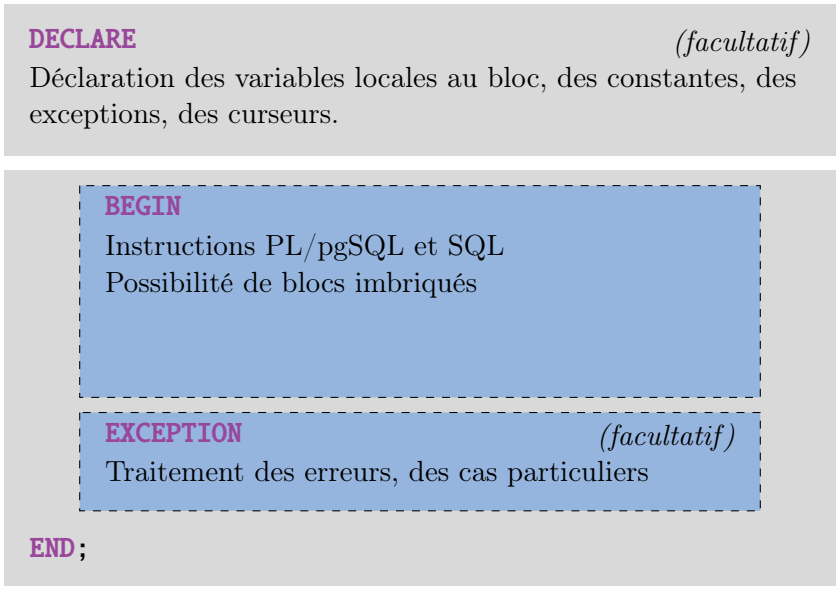
PL/pgSQL est un langage procédural utilisé dans le cadre d'une base de données PostgreSQL. Il permet de combiner des requêtes SQL et des instructions procédurales (boucles, conditions ...) dans le but de créer des traitements complexes destinés à être stockés sur le serveur de base de données. Les principaux avantages d'un langage procédural intégré à la base de données sont les suivants :

— Peu importe le langage de programmation utilisé pour coder une application, le code PL/pgSQL est le même.
— Le PL/pgSQL peut être appelé depuis n'importe quel client de la base de données.
— Il peut être déclenché directement par des mécanismes internes (triggers, ...)
— Il est plus performant qu'un traitement externe car il peut être optimisé par le SGBD.
— Il intègre nativement le SQL.

L'inconvénient évident est qu'il est lié au moteur de base de données. Donc une application dont les données sont stockée dans une base PostgreSQL avec du PL/pgSQL ne peut pas fonctionner sur Oracle ou MySQL par exemple sans redévelopper la partie procédurale interne à la base de données.

PL/pgSQL permet de créer des fonctions personnalisées, de gérer les exceptions, mais aussi de créer des déclencheurs qui réagissent à des actions faites sur la base de données (voir Chapitre 5).

4.1. Bloc PL/pgSQL



En SQL, les commandes sont transmises les unes après les autres au moteur SQL qui les traitent

séparément. Au contraire, les commandes PL/pgSQL sont transmises et interprétées par **bloc**. Un bloc PL/pgSQL est composé de trois parties :

— Les parties `DECLARE` et `EXCEPTION` sont facultatives.
— Chaque instruction au sein d'un bloc (quelle que soit la partie) est terminée par un `;`
— Il est possible d'ajouter des commentaires : `-- commentaire sur une ligne`

Un bloc PL/pgSQL peut être utilisé dans une fonction (voir Section 4.5) ou comme un bloc anonyme. Dans ce cas, il doit être entouré par `DO $$ ... $$;`

```
DO $$
    DECLARE
    ...
    BEGIN
    ...
    EXCEPTION
    ...
    END;
$$;
```

4.2. Déclaration de variables (`DECLARE`)

Pour utiliser une variable locale dans un bloc, elle doit être déclarée dans la partie `DECLARE`.

```
<nom_variable> [CONSTANT] <type> [NOT NULL] [{DEFAULT | :=} <valeur>]
```

Le type de la variable doit être définit lors de la déclaration de la variable. Il peut être n'importe quel type de données de base. Il peut aussi être définit en faisant référence à une colonne ou une table.

▶ **Type d'une colonne.** Il est possible de définir un type de variable directement en faisant référence à une donnée déjà existante dans la base avec la commande `%TYPE`. Si le type de données change dans la table, il n'est pas nécessaire de modifier le code PL/pgSQL y faisant référence.

Exemple4.1

```
var_id Client.id%TYPE;
```

▶ **Type d'une ligne.** De la même manière, il est possible de récupérer la définition complète d'une table et de l'affecter à une variable PL/pgSQL qui contiendra un tuple de la table. Pour cela, il faut utiliser `%ROWTYPE`.

Exemple4.2

```
var_client Cient%ROWTYPE;
```

▶ **Affectations.** Il est possible d'affecter une valeur aux variables lors de leur déclaration.

Exemple4.3

```
var1 date := CURRENT_DATE;
var2 CONSTANT numeric(3,1) := 49.3;
num  numeric(1) DEFAULT 5;
cond boolean NOT NULL := false;
```

Le mot-clé **CONSTANT** permet d'empêcher la modification de la valeur de la variable après son initialisation.

Une variable déclarée **NOT NULL** ne peut être affectée à la valeur **NULL** (que ce soit à son initialisation ou plus tard dans le traitement). Elle doit donc forcément être initialisée lors de sa déclaration.

4.3. Traitements (BEGIN)

4.3.1. *SELECT ... INTO ...*

Lorsqu'un **SELECT** retourne une seule ligne, il est possible de récupérer le résultat dans une variable grâce au mot clé **INTO**.

```
SELECT col1, col2 INTO var1, var2
FROM table
[WHERE ...];
```

⚠ Si le **SELECT** retourne plusieurs lignes, la première d'entre elles sera mise dans la variable. De même, si aucune ligne n'est retournée, la variable prendra la valeur **NULL**. Si l'on ne veut pas ce comportement il faut ajouter l'option **STRICT** :

```
SELECT col1, col2 INTO STRICT var1, var2
FROM table
[WHERE ...];
```

Dans ce cas, une erreur **no_data_found** est levée si la requête ne retourne aucune ligne et une erreur **too_many_rows** est levée si la requête retourne plus d'une ligne.

Traiter un **SELECT** dont le résultat peut avoir plusieurs lignes nécessite un curseur (voir Section 4.6).

EXEMPLE4.4

```
DO $$
DECLARE
    n Client.nom%TYPE;
    p Client.prenom%TYPE;
BEGIN
    SELECT nom, prenom INTO n, p
    FROM Client
    WHERE id = 'C0042';

    RAISE NOTICE 'Le client C042 est % %', upper(n), initcap(p);
END;
$$;
```

4.3.2. *Conditions IF ... THEN ... END IF;*

Il est possible d'utiliser des conditions comme dans d'autres langages de programmation. Plusieurs syntaxes sont possibles :

- **IF** <condition> **THEN** <traitement> **END IF;**
- **IF** <cond> **THEN** <trait1> **ELSE** <trait2> **END IF;**
- **IF** <cond1> **THEN** <trait1> **ELSIF** <cond2> **THEN** <trait2> **ELSE** <trait3> **END IF;**

Les opérateurs utilisés dans les conditions en PL/pgSQL sont les mêmes qu'en SQL : =, !=, <, >, <=, >=, **AND**, **OR**, **NOT**, ...

Exemple4.5

```
DO $$
DECLARE
   n Chambre.numero%TYPE := '102';
   e Chambre.etage%TYPE;
   s Chambre.superficie%TYPE;
BEGIN
   SELECT etage, superficie INTO e, s
   FROM Chambre
   WHERE numero = n;

   IF s <= 25 THEN
      RAISE NOTICE 'Chambre % (étage %)', n, e;
   ELSE
      RAISE NOTICE 'Suite % (étage %)', n, e;
   END IF;
END;
$$;
```

4.3.3. Boucles *WHILE*

Le corps des boucles **WHILE** est exécuté tant que la condition est évaluée à vrai. La condition (une combinaison d'expressions booléennes utilisant les opérateurs =, !=, **AND**, **OR**, ...) est évaluée avant chaque exécution du corps de la boucle.

```
WHILE <condition> LOOP
    <corps de la boucle>
END LOOP;
```

Exemple4.6

Calcul de factorielle de 10.

```
DO $$
DECLARE
   n   numeric := 10;
   i   numeric := 1;
   res numeric := 1;
BEGIN
   WHILE i <= n LOOP
      res := i * res;
      i := i+1;
   END LOOP;
   RAISE NOTICE 'Factorielle de % = %', n, res;
END;
$$;
```

4.3.4. Boucles *LOOP*

Une boucle **LOOP** est une boucle sans condition, dont le corps est exécuté indéfiniment jusqu'à l'exécution d'un **EXIT** ou **RETURN**.

```
LOOP
    <corps de la boucle>
    EXIT WHEN <condition>;
END LOOP;
```

Exemple4.7

```
DO $$
DECLARE
   i   numeric := 1;
BEGIN
   LOOP
       i := i+1;
       EXIT WHEN i = 50;
   END LOOP;
   RAISE NOTICE 'i = %', i;
END;
$$;
```

4.3.5. *Boucles FOR ... IN*

Une boucle **FOR ... IN** permet d'itérer un compteur tant que sa valeur reste dans les bornes **mini** et **maxi**. Il n'est pas nécessaire de déclarer la variable de boucle dans la partie **DECLARE**.

```
FOR <variable de boucle> IN [REVERSE] <mini> .. <maxi> [BY <pas>] LOOP
    <corps de la boucle>
END LOOP;
```

Exemple4.8

```
DO $$
BEGIN
   FOR i IN 1 .. 10 LOOP
       -- i prend les valeurs 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
       RAISE NOTICE 'i = %', i;
   END LOOP;

   FOR i IN REVERSE 10 .. 1 LOOP
       -- i prend les valeurs 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
       RAISE NOTICE 'i = %', i;
   END LOOP;

   FOR i IN 1 .. 10 BY 2 LOOP
       -- i prend les valeurs 1, 3, 5, 7, 9
       RAISE NOTICE 'i = %', i;
   END LOOP;
END;
$$;
```

4.4. Gestion des erreurs (EXCEPTION)

Lorsqu'une instruction se passe mal en PL/pgSQL, une **exception** est levée. Il est possible d' "attraper" ces exceptions et de définir le comportement adapté permettant de gérer l'erreur dans la partie **EXCEPTION**. Bien que facultative, cette partie est essentielle pour une bonne gestion des erreurs.

4.4.1. *Traiter les exceptions*

Dès qu'une erreur PostgreSQL se produit, l'exécution passe automatiquement à la partie **EXCEPTION** pour réaliser le traitement approprié à l'exception en question.

```
EXCEPTION
WHEN nom_exception1 THEN traitement1;
[WHEN nom_exception2 THEN traitement2;]
...
[WHEN OTHERS THEN traitement;]
```

Exemple4.9

```
DO $$
DECLARE
   n1 numeric := 1;
   n2 numeric := 2;
   n3 numeric := 3;
   res numeric;
BEGIN
   res := n3/(n2 - 2*n1);
   RAISE NOTICE 'res = %', res;
EXCEPTION
   WHEN division_by_zero THEN
      RAISE NOTICE 'Division par zero';
   END;
$$;
```

4.4.2. *Exceptions PostgreSQL*

De nombreuses exceptions PostgreSQL sont définies avec un code d'erreur **SQLSTATE** et le nom correspondant. Quelques exceptions PostgreSQL parmi les plus courantes sont décrites ci-dessous :

SQLSTATE	Nom	Détails
22012	division_by_zero	Division par zéro
23503	foreign_key_violation	Contrainte REFERENCES violée
23505	unique_violation	Contrainte UNIQUE violée
23514	check_violation	Contrainte CHECK violée
24000	invalid_cursor_definition	Problème de curseur
P0002	no_data_found	Pas de lignes retournées
P0003	too_many_rows	Plusieurs lignes retournées alors qu'on en attendait qu'une

Il est possible de récupérer le code et le message de l'erreur avec les variables **SQLSTATE** et **SQLERRM**, respectivement.

⚠ Ces variables ne sont disponibles que dans la partie **EXCEPTION**.

Exemple4.10

```
DO $$
DECLARE
    i Client.id%TYPE;
BEGIN
   SELECT id INTO STRICT i
   FROM Client
   WHERE pays = 'France';

   RAISE NOTICE 'id = %', i;
EXCEPTION
```

```
    WHEN OTHERS THEN
        RAISE NOTICE '% - %', SQLSTATE, SQLERRM;
        -- affiche : P0003 - la requête a renvoyé plus d'une ligne
END;
$$;
```

Pour avoir encore plus de détails sur l'exception et le contexte dans lequel elle a été levée, il est possible d'utiliser la commande **GET STACKED DIAGNOSTICS** et de récupérer les champs de diagnostic qui nous intéresse dans des variables.

Quelques champs de diagnostic utiles sont listés ci-dessous :

Nom	Description
RETURNED_SQLSTATE	le code erreur **SQLSTATE** de l'exception
MESSAGE_TEXT	le message d'erreur **SQLERRM** de l'exception
PG_EXCEPTION_CONTEXT	le contexte (la pile d'exécution) au moment de la levée de l'exception
PG_EXCEPTION_DETAIL	des détails sur l'exception, s'il y en a
PG_EXCEPTION_HINT	des indications/astuces pouvant aider à résoudre le problème, s'il y en a

Exemple4.11

```
DO $$
DECLARE
    i Client.id%TYPE;
    var1 text;
    var2 text;
    var3 text;
BEGIN
   SELECT id INTO STRICT i
   FROM Client
   WHERE pays = 'France';

   RAISE NOTICE 'id = %', i;
EXCEPTION
   WHEN OTHERS THEN
       GET STACKED DIAGNOSTICS var1 = MESSAGE_TEXT,
                               var2 = PG_EXCEPTION_CONTEXT,
                               var3 = PG_EXCEPTION_HINT;
       RAISE NOTICE '%', var1;
       RAISE NOTICE '%', var2;
       RAISE NOTICE '%', var3;
END;
$$;

--NOTICE:  la requête a renvoyé plus d'une ligne
--NOTICE:  fonction PL/pgSQL inline_code_block, ligne 8 à instruction SQL
--NOTICE:  Assurez-vous que la requête ne renvoie qu'une seule ligne ou
   utilisez LIMIT 1.
```

4.4.3. *Signaler une erreur*

La commande **RAISE** permet d'afficher un message ou de lever une exception.

```
RAISE [<niveau>] <chaîne formatée> [USING <option> = <expression> [, ... ]];
```

Le niveau indique la sévérité de l' "erreur". Les niveaux possibles sont : **DEBUG**, **LOG**, **INFO**, **NOTICE**, **WARNING** et **EXCEPTION**. **EXCEPTION** est le niveau par défaut et permet de lever une exception. Les autres niveaux permettent simplement de générer des messages dont la priorité dépend du niveau. Par défaut, les messages dont le niveau est au moins **INFO** sont affichés dans le terminal.

Il est possible de rajouter des informations supplémentaires avec **USING**. Par exemple :

- **DETAIL** : définit les détails du message d'erreur
- **HINT** : définit l'indication/l'astuce permettant de résoudre le problème
- **ERRCODE** : définit le code d'erreur de l'exception. Peut prendre comme valeur soit un **SQLSTATE**, soit le nom d'une exception PostgreSQL, par exemple `22012` ou **division_by_zero** pour une division par zéro. Par défaut, il s'agit de **errcode_raise_exception** (`P0001`).
- **MESSAGE** : définit le message d'erreur. Ne peut être utilisé qu'avec la seconde syntaxe décrite ci-après (sans la chaîne de caractères avant le **USING**).

Exemple4.12

```
DO $$
DECLARE
    n Client.nom%TYPE;
    i Client.id%TYPE := 'C0141';
BEGIN
    SELECT nom INTO n
    FROM Client
    WHERE id = i;

    IF n IS NULL THEN
        RAISE EXCEPTION 'ID client % inconnu', i
        USING HINT = 'Vérifier l''id du client';
    END IF;

    RAISE NOTICE 'Nom du client % : %', i, n;
END;
$$;

--ERREUR:  ID client C0141 inconnu
--ASTUCE : Vérifier l'id du client
--CONTEXTE : fonction PL/pgSQL inline_code_block, ligne 11 à RAISE
```

Une autre syntaxe permet de lever des exceptions PostgreSQL :

```
RAISE <nom_exception> [USING <option> = <expression> [, ... ]];
```

Exemple4.13

```
DO $$
DECLARE
    n Client.nom%TYPE;
    i Client.id%TYPE := 'C0141';
BEGIN
    SELECT nom INTO n
    FROM Client
    WHERE id = i;

    IF n IS NULL THEN
        RAISE no_data_found USING MESSAGE = 'ID client inconnu';
```

```
    END IF;

    RAISE NOTICE 'Nom du client % : %', i, n;
END;
$$;
```

4.5. Fonctions

L'utilité des blocs anonymes `DO $$ ... $$;` est assez limitée, à part pour regrouper des instructions. Dans beaucoup de cas, on veut pouvoir paramétrer le bloc PL/pgSQL pour le rendre réutilisable. Pour pouvoir faire cela, il faut définir des **fonctions**.

```
CREATE FUNCTION ma_fonction(param1, param2, ...) RETURNS type_retour AS $$
   corps de la fonction
$$ LANGUAGE plpgsql;
```

Le corps d'une fonction est un bloc PL/pgSQL : `[DECLARE] ... BEGIN ... [EXCEPTION] ... END;`

Pour supprimer une fonction, il faut utiliser : `DROP FUNCTION ma_fonction(param1, param2, ...);` Si le nom de la fonction est unique (pas de surcharge), il est possible de supprimer une fonction en indiquant uniquement son nom (sans les paramètres) : `DROP FUNCTION ma_fonction;`

▶ **Paramètres.** Une fonction peut prendre des paramètres. Pour faire référence à ces paramètres dans le corps de la fonction, il est possible d'utiliser le nom du paramètre ou son numéro, par exemple `$1`.

▶ **Fonction retournant un type "simple".** Comme dans d'autres langages de programmation, une fonction qui ne retourne aucune valeur (ou une procédure) est définie comme retournant le type `void`.

Une fonction peut aussi retourner un type simple, comme par exemple `numeric`, `char`, ... La valeur à retourner sera défini par l'instruction `RETURN`.

Exemple4.14

Fonction augmentant le prix d'une chambre donnée selon le pourcentage indiqué en paramètre et retournant le nombre de tarifs mis à jour.

```
CREATE FUNCTION maj_prix(c Chambre.numero%TYPE, pourc numeric) RETURNS
   integer AS $$
DECLARE
    i integer;
BEGIN
    SELECT count(*) INTO i
    FROM Prix
    WHERE chambre = $1;

    UPDATE Prix
    SET prix = prix+prix*pourc/100
    WHERE chambre = c;

    RETURN i;
END;
$$ LANGUAGE plpgsql;
```

```
SELECT maj_prix('101', 10);
```

▶ **Fonction retournant une table.** Une fonction peut également retourner un ensemble de valeurs.

Exemple4.15

Fonction calculant les coordonnées des clients qui arrivent à l'hôtel aujourd'hui.

```
CREATE FUNCTION arrivees()
RETURNS TABLE(nom Client.nom%TYPE, prenom Client.prenom%TYPE, tel Client.
    noTelephone%TYPE) AS $$
BEGIN
    RETURN QUERY SELECT DISTINCT c.nom, c.prenom, c.noTelephone
    FROM Client c JOIN Reservation r ON c.id = r.client
    WHERE r.dateDebut = CURRENT_DATE
    ORDER BY 1, 2;
END;
$$ LANGUAGE plpgsql;
```

▶ **Utilisation d'une fonction.** Une fonction PL/pgSQL peut être utilisée comme les fonctions SQL de base, dans un `SELECT` ou un `WHERE` par exemple.

Une fonction retournant une table peut également être utilisée comme source dans une clause `FROM`.

Exercice 13 **Exercice 14** **Exercice 15**

4.6. Curseurs

Lorsqu'une requête `SELECT` peut retourner plusieurs lignes, il n'est pas possible de mettre le résultat dans une variable "classique". Il faut utiliser un **curseur explicite** qui va permettre d'accéder aux lignes du résultat.

4.6.1. Parcours "manuel" avec curseur

Il est possible de faire un parcours manuel avec un curseur. L'algorithme à utiliser est similaire à celui permettant la lecture séquentielle dans un fichier.

```
DECLARE
    --declaration du curseur
    nom_du_curseur cursor FOR SELECT ... FROM ...;
BEGIN
    OPEN nom_du_curseur; --ouverture du curseur
    --lecture de la prochaine entrée du curseur
    FETCH nom_du_curseur INTO var1, var2, var3 ...;
    WHILE FOUND LOOP --boucle tant qu'il y a des entrées
        -- traitement de la ligne lue
        FETCH nom_du_curseur INTO var1, var2, var3 ...;
    END LOOP;   ...
    CLOSE nom_du_curseur; --fermeture du curseur
END;
```

4.6.2. Parcours avec boucle `FOR` sur curseur

En reprenant le principe de la boucle `FOR ... IN`, la boucle `FOR` sur curseur facilite la vie du développeur. En effet, il n'est pas nécessaire de déclarer le curseur, de l'ouvrir, le fermer, gérer les `FETCH` et la

fin du parcours.

```
FOR <nom_tuple> IN {<nom_curseur> | (<select>)} LOOP
    <corps_de_la_boucle>
END LOOP;
```

Exemple4.16

Fonction calculant le total de la facture d'une réservation (le prix d'une nuit étant celui de la chambre le premier jour de la réservation).
Avec un curseur :

```
CREATE FUNCTION total(Reservation.numero%TYPE) RETURNS numeric AS $$
DECLARE
    res  numeric := 0;
    d    Reservation.dateDebut%TYPE;
    nb   Reservation.nbNuits%TYPE;
    p    Prix.prix%TYPE;
    curs cursor FOR SELECT chambre
                    FROM Occupation
                    WHERE reservation = $1;
BEGIN
    SELECT dateDebut, nbNuits INTO d, nb
    FROM Reservation
    WHERE numero = $1;

    FOR chbr IN curs LOOP
        SELECT p.prix INTO p
        FROM Prix p JOIN Periode pe ON pe.nom = p.periode
        WHERE pe.dateDebut <= d AND pe.dateFin >= d
            AND p.chambre = chbr.chambre;
        res := res + p;
    END LOOP;

    RETURN res*nb;
END;
$$ LANGUAGE plpgsql;
```

Sans curseur prédéfini :

```
CREATE FUNCTION total_v2(Reservation.numero%TYPE) RETURNS numeric AS $$
DECLARE
    chbr record;
    res  numeric := 0;
    d    Reservation.dateDebut%TYPE;
    nb   Reservation.nbNuits%TYPE;
    p    Prix.prix%TYPE;
BEGIN
    SELECT dateDebut, nbNuits INTO d, nb
    FROM Reservation
    WHERE numero = $1;

    FOR chbr IN (SELECT chambre
                 FROM Occupation
                 WHERE reservation = $1) LOOP
        SELECT p.prix INTO p
        FROM Prix p JOIN Periode pe ON pe.nom = p.periode
        WHERE pe.dateDebut <= d AND pe.dateFin >= d
            AND p.chambre = chbr.chambre;
        res := res + p;
```

```
    END LOOP;

    RETURN res*nb;
END;
$$ LANGUAGE plpgsql;
```

Notez que dans le cas où le curseur n'est pas prédéfini, il faut que la variable qui sert à stocker les tuples lors du parcours du curseur doit être définit dans la partie **DECLARE** avec un type **record**.

Exercice 16

4.6.3. Modifications avec curseur

Lorsque l'on parcours une table avec un curseur et **FETCH**, il est possible de mettre à jour ou supprimer la ligne courant en utilisant **WHERE CURRENT OF** `nom_du_curseur`. Le curseur doit dans ce cas être déclaré **FOR UPDATE** (pour éviter des problèmes liés à des accès concurrents).

EXEMPLE4.17

Des travaux sont réalisés au 2ème étage pour modifier la superficie des chambres.

```
DO $$
DECLARE
    curs cursor FOR SELECT superficie
                    FROM Chambre
                    WHERE etage = 2
                    FOR UPDATE;
BEGIN
    FOR r IN curs LOOP
        IF r.superficie < 14 THEN
            UPDATE Chambre
            SET superficie = superficie + 2;
            WHERE CURRENT OF curs;
        ELSIF r.superficie > 18
            UPDATE Chambre
            SET superficie = superficie - 2;
            WHERE CURRENT OF curs;
        END IF;
    END LOOP;
END;
$$;
```

5. Déclencheurs/Triggers

Les **déclencheurs** ou **triggers** permettent d'exécuter du code PL/pgSQL lorsqu'un événement arrive. La majorité des triggers concernent des actions sur une table. Il est par exemple possible de définir le déclenchement d'un code PL/pgSQL à chaque insertion d'une nouvelle donnée dans une table. Un trigger s'exécute dans le cadre d'une transaction.

5.1. Création d'un trigger

Pour créer un trigger, il faut définir quand ce trigger va se déclencher et qu'est-ce qui doit être exécuté en cas déclenchement.

▶ **Syntaxe simplifiée.**

```
CREATE TRIGGER <nom> BEFORE  INSERT   ON <table>
                    AFTER   UPDATE
                            DELETE
[FOR EACH ROW]
[WHEN ( <condition> )]
EXECUTE FUNCTION <nom_fonction>();
```

— **BEFORE** ou **AFTER** : La fonction trigger peut être exécutée avant (**BEFORE**) qu'une opération ait lieu sur une ligne (avant la vérifications des contraintes d'intégrité et la réalisation de l'**INSERT**/**UPDATE**/**DELETE**) ou après (**AFTER**) qu'une opération ait eu lieu (après la vérifications des contraintes d'intégrité et la réalisation de l'**INSERT**/**UPDATE**/**DELETE**).

— **INSERT**, **UPDATE** ou **DELETE** : Action pour laquelle la fonction trigger est déclenchée. Il est possible de mettre plusieurs actions séparées par **OR**.

— **ON** : nom de la table concernée par le trigger

— **FOR EACH ROW** : Permet d'exécuter la fonction trigger pour chaque ligne concernée par l'action (par exemple, pour chaque ligne insérée). On parlera alors de trigger **niveau ligne**. Sans cette option, la fonction trigger est appelée une seule fois, quel que soit le nombre de lignes concernées et on parlera de trigger **niveau instruction**.

— **WHEN** : permet de restreindre l'exécution du trigger au cas respectant la condition indiquée après. Cette condition peut porter sur les nouvelles et/ou les anciennes valeurs des lignes.

— **EXECUTE FUNCTION** : indique la fonction trigger à exécuter en cas de déclenchement.

⚠ La fonction trigger doit être définit **avant** le trigger.

Exercice 17

5.2. Fonction trigger

Même s'il est possible de définir des fonctions trigger dans plusieurs langages, nous nous focaliserons dans ce cours sur les fonctions trigger PL/pgSQL.

⚠ Une fonction trigger ne peut pas avoir de paramètres.

▶ **Variables spéciales.** Quand une fonction PL/pgSQL est appelée en tant que trigger, des variables spéciales sont automatiquement déclarées.

— `NEW` : Variable contenant la valeur de la nouvelle ligne après un `INSERT` ou `UPDATE` pour un trigger niveau ligne. Cette variable vaut `NULL` pour les `DELETE` et les triggers niveau instruction.
— `OLD` : Variable contenant la valeur de l'ancienne ligne après un `DELETE` ou `UPDATE` pour un trigger niveau ligne. Cette variable vaut `NULL` pour les `INSERT` et les triggers niveau instruction.
— `TG_OP` : Chaîne de caractères indiquant l'opération ayant déclenché le trigger : `INSERT`, `UPDATE` ou `DELETE`.

▶ **Valeur de retour.** Une fonction appelée en tant que trigger doit avoir une valeur de retour de type `trigger`. Cette fonction doit retourner soit `NULL`, soit une ligne de la table ayant déclenché le trigger. Plus précisément :

— Une fonction trigger `BEFORE` niveau ligne peut retourner `NULL` pour signaler que le reste de l'opération sur cette ligne doit être annulée, autrement dit que l'`INSERT`/`UPDATE`/`DELETE` ne doit pas avoir lieu pour cette ligne. Si une valeur non nulle est retournée, l'opération doit s'effectuer avec cette valeur là.

  Autrement dit, si une valeur différente de `NEW` est retournée, cela va impacter les valeurs qui vont effectivement être ajoutées ou modifiées sur un `INSERT` ou `UPDATE`. Il est possible de modifier directement la valeur

  Si on veux que le trigger n'altère pas les opérations d'insertion ou de mise à jour, il faut retourner `NEW`. Notez que pour une opération `DELETE`, la valeur retournée n'a pas d'effet, mais par convention, on retourne généralement `OLD`.
— La valeur retournée par une fonction trigger `AFTER` ou niveau transaction est toujours ignorée. On retourne donc généralement `NULL`.

Exercice 18 Exercice 19

5.3. Gestion des triggers

▶ **Suppression d'un trigger.** Il est possible de supprimer un trigger grâce à la commande :

```
DROP TRIGGER <nom_du_trigger> ON <nom_de_la_table>;
```

Notez qu'il n'est pas possible de supprimer une fonction trigger tant qu'elle utilisée par des triggers. Cependant, il est possible de rajouter l'option `CASCADE` lors de la suppression de la fonction pour entraîner la suppression automatique des triggers faisant appel à cette fonction.

▶ **Désactivation d'un trigger.** Il est possible de désactiver temporairement un trigger. Le trigger existe toujours mais n'est pas déclenché.

```
-- désactive le trigger nom_trigger
ALTER TABLE <nom_table> DISABLE TRIGGER <nom_trigger>;
-- désactive tous les triggers définit par un utilisateur sur la table
   nom_table
```

```
ALTER TABLE <nom_table> DISABLE TRIGGER USER;
-- désactive tous les triggers sur la table nom_table
ALTER TABLE <nom_table> DISABLE TRIGGER ALL;
```

⚠ Désactiver **tous** les triggers d'une table implique la désactivation de trigger systèmes. Il faut disposer des droits superuser pour faire cela.

▶ **Réactivation d'un trigger.** Il est possible de réactiver un ou des triggers désactivés.

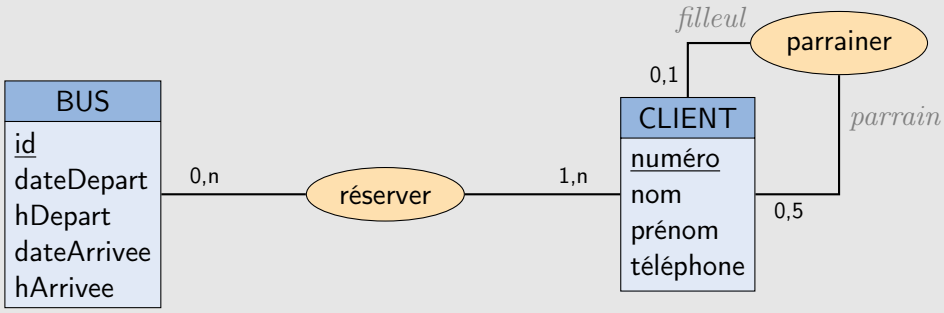
```
-- réactive le trigger nom_trigger
ALTER TABLE <nom_table> ENABLE TRIGGER <nom_trigger>;
-- réactive tous les triggers définit par un utilisateur sur la table
   nom_table
ALTER TABLE <nom_table> ENABLE TRIGGER USER;
-- réactive tous les triggers sur la table nom_table
ALTER TABLE <nom_table> ENABLE TRIGGER ALL;
```

5.4. Trigger pour garantir des cardinalités

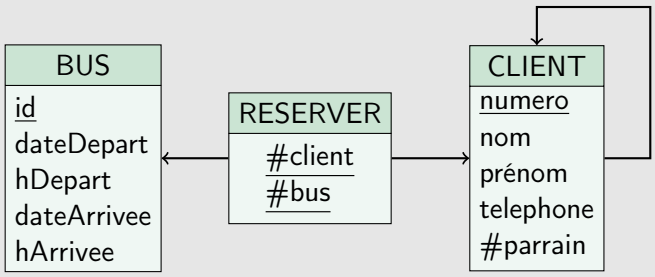
Lors du passage du MCD au MLD, puis lors de l'implémentation de la base de données en SQL, certaines informations portées par les cardinalités sont perdues, car elles ne peuvent pas être intégrées à la base de données de façon "simple".

Exemple5.1

Soit le MCD suivant extrait de la base de données gérant une ligne de bus :



Ce MCD est traduit en MLD ainsi :



La traduction de l'association réserver en relation permet d'exprimer les cardinalités maximum à n des deux côtés de cette association. En effet, un bus donné peut apparaître plusieurs fois dans la relation RESERVER, de même qu'un client donné. La cardinalité minimum à 0 côté BUS n'étant pas contraignante, elle exprimée *de facto*. Par contre, rien ne permet de garantir la cardinalité minimum à 1 côté CLIENT. Il faudrait vérifier que tout client apparaît bien dans RESERVER (autrement dit l'inverse de la propriété apportée par la clé étrangère client dans RESERVER).
Pour l'association parrainer, à nouveau, les cardinalités minimum à 0 sont exprimées *de facto*. La cardinalité maximum à 1 côté filleul est exprimée par la clé étrangère faisant référence à la relation `CLIENT` dans `CLIENT` puisqu'elle ne peut prendre qu'une seule valeur. Un seul parrain (au

maximum, en ne mettant pas la contrainte **NOT NULL**) est donc associé à un client. La cardinalité maximum à 5 côté parrain est elle perdue. La traduction permet à un client de faire plusieurs parrainage. Cependant, la clé étrangère (et n'importe quelle contrainte que l'on pourrait rajouter) ne permet pas de garantir que le client n'a pas plus de 5 filleuls.

Il est cependant possible d'utiliser un trigger pour vérifier ces cardinalités. Dans l'exemple précédent, on pourrait créer un trigger se déclenchant lors de l'insertion dans la table **CLIENT**. Ce trigger vérifierait qu'il y a bien une ligne dans la table **RESERVER** concernant ce client, et annulerait l'insertion si ce n'est pas le cas. Il se pose alors un problème d'ordre des opérations :

a). Si le client est créé en premier, le trigger se déclenche. La réservation n'ayant pas encore été enregistré, le trigger annule l'insertion du client.

b). À l'inverse, si la réservation est créée en premier, une erreur est levée par la clé étrangère référençant le client, puisque celui-ci n'a pas encore été créé. L'insertion de la réservation est annulée.

Le même problème se pose lors de la suppression d'une réservation s'il s'agit de la seule d'un client.

Pour résoudre ce problème, il faut retarder la vérification de la clé étrangère. Pour faire cela, il faut utiliser l'option **DEFERRABLE INITIALLY DEFERRED** :

— **DEFERRABLE** signifie que la vérification de la contrainte peut être repoussé à la fin de la transaction.

— **INITIALLY DEFERRED** signifie que par défaut, la vérification de la contrainte est repoussée à la fin de la transaction.

Il faut donc les deux options. (Ce comportement peut être changé *a posteriori* avec la commande **SET CONSTRAINT**.)

EXEMPLE5.2

Pour vérifier la cardinalité minimum à 1 côté CLIENT de l'association réserver dans l'exemple 5.1, il faut donc repousser la vérification de la clé étrangère dans la table **RESERVER** :

```
CREATE TABLE Reserver(
       client numeric REFERENCES Client DEFERRABLE INITIALLY DEFERRED,
       bus numeric REFERENCES Bus,
       PRIMARY KEY(client, bus)
);
```

Il faut également mettre en place un trigger sur l'insertion dans la table **CLIENT** :

```
CREATE OR REPLACE FUNCTION tg_client_insert() RETURNS trigger AS $$
DECLARE
  nb numeric;
BEGIN
  SELECT count(*) INTO nb
  FROM Reserver
  WHERE client = NEW.no;

  IF nb = 0 THEN
    RAISE no_data_found USING MESSAGE = 'Insertion impossible car le
                    client ' || NEW.no || 'n''a pas de réservation.';
  END IF;

  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER client_insert BEFORE INSERT ON Client
FOR EACH ROW
```

```
EXECUTE FUNCTION tg_client_insert();
```

Il suffit ensuite, dans un même transaction, d'insérer d'abord une ligne dans la table RESERVER avant de créer le client dans la table CLIENT.
La même chose peut être faite lors de la suppression d'une réservation pour automatiquement supprimer le client associé s'il s'agissait de la seule réservation le concernant :

```
CREATE OR REPLACE FUNCTION tg_reserver_delete() RETURNS trigger AS $$
DECLARE
  nb numeric;
BEGIN
  SELECT count(*) INTO nb
  FROM Reserver
  WHERE client = OLD.client;

  IF nb = 0 THEN
    RAISE NOTICE 'Suppression du client % car il n''a plus de
                                          réservation.', OLD.client;
    DELETE FROM Client
    WHERE no = OLD.client;
  END IF;

  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER reserver_delete AFTER DELETE ON Reserver
FOR EACH ROW
EXECUTE FUNCTION tg_reserver_delete();
```

Il est également possible d'utiliser des triggers pour garantir des cardinalités maximum avec une valeur précise, comme par exemple une cardinalité maximum à 5.

EXEMPLE5.3

Pour vérifier la cardinalité maximum à 5 côté parrain de l'association parrainer dans l'exemple 5.1, il faut mettre en place un trigger qui interdit l'insertion d'un client si son parrain a déjà 5 filleuls.

```
CREATE OR REPLACE FUNCTION tg_parrain_insert() RETURNS trigger AS $$
DECLARE
  nb numeric;
BEGIN
  SELECT count(*) INTO nb
  FROM Client
  WHERE parrain = NEW.parrain;

  IF nb = 5 THEN
    RAISE too_many_rows USING MESSAGE = 'Insertion impossible car le
   client ' || NEW.parrain || ' a déjà parrainé 5 clients.';
  END IF;

  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER parrain_insert BEFORE INSERT ON Client
```

```
FOR EACH ROW
WHEN (NEW.parrain IS NOT NULL)
EXECUTE FUNCTION tg_parrain_insert();
```

5.5. Séquence

Dans les systèmes de gestion de bases de données telles que MariaDB ou MySQL, il est possible de créer une table avec une colonne de type "*autoincrement*". Ainsi, à l'insertion d'une nouvelle ligne dans la table, la valeur qui est insérée dans cette colonne est une incrémentation de la valeur précédente. Cette colonne sert alors de clé primaire. Dans une base de données PostgreSQL, le type équivalent est le type **serial**. Il existe cependant un mécanisme plus complet, la **séquence**, qui permet d'avoir le même fonctionnement mais en personnalisant les valeurs.

Une séquence est un compteur qui s'incrémente dès qu'il est consulté. Pour utiliser une séquence pour gérer les valeurs d'identifiants, il faut créer une séquence, puis, lors de l'insertion d'une ligne dans la table, il faut lire la valeur suivante de la séquence pour l'affecter à la colonne en question. Pour que cela soit fait automatiquement, les séquences sont souvent utilisées via des triggers, voir Exercice 21.

En réalité, derrière les mécanismes des colonnes **serial** (ou **GENERATED AS IDENTITY**), se cache une séquence implicite.

▶ **Syntaxe :**

```
CREATE SEQUENCE <nom_sequence>
[INCREMENT BY <increment>]
[MINVALUE <min>]
[MAXVALUE <max>]
[START WITH <val_debut>]
[CYCLE | NO CYCLE]
[OWNED BY <table.col>];
```

— **INCREMENT BY** : détermine l'incrémentation, la valeur de la séquence augmente par exemple de 1, 10, -1, ou -5.
— **MINVALUE** : détermine la valeur minimale de la séquence (optionnel).
— **MAXVALUE** : détermine valeur maximale de la séquence (optionnel).
— **START WITH** : détermine la valeur de départ de la séquence.
— **CYCLE**/**NO CYCLE** : indique si la séquence est cyclique ou non, autrement dit si, lorsqu'une borne est atteinte (**MINVALUE** ou **MAXVALUE**), la séquence repart depuis l'autre borne (respectivement **MAXVALUE** et **MINVALUE**). Par défaut, la séquence est non cyclique.
— **OWNED BY** permet d'associer la séquence à la colonne d'une table. Dans ce cas, lorsque la colonne (ou la table entière) est supprimée, la séquence sera automatiquement supprimée.

EXEMPLE5.4

```
CREATE SEQUENCE ma_sequence
START WITH 100 INCREMENT BY 1 NO CYCLE;
```

▶ **Utilisation de la séquence :**

— Valeur courante : `SELECT currval('ma_sequence');`

— Valeur suivante (et incrémentation de la valeur courante) : `SELECT nextval('ma_sequence');`

Si la séquence n'est pas cyclique (`NO CYCLE`) et une borne est atteinte (`MAXVALUE` si la séquence est croissante, `MINVALUE` sinon), un appel à `nextval` retourne une erreur.

— Suppression de la séquence : `DROP SEQUENCE ma_sequence;`

⚠ Les apostrophes sont nécessaires autour du nom de la séquence pour les fonctions `currval` et `nextval`.

Exercice 20 **Exercice 21**

Mémo

▶ **Connexion à la base.** `psql -h londres -d <database> -U <login> -W`

— `<login>` = `login_ENT`, par exemple `anaduran`
— `<database>` = `dblogin`, par exemple `dbanaduran`
— mot de passe = `achanger`

▶ **Changement de mot de passe.** Après s'être connecté à PostgreSQL :

```
ALTER USER login WITH PASSWORD 'mon nouveau mot de passe';
```

⚠ Ne pas oublier les `' '` (qui ne font pas partie du mot de passe) et ne pas oublier le `;` à la fin !

Si vous avez oublié votre mot de passe, vous pouvez le réinitialiser en allant sur : `londres.uca.local`

▶ **Commandes utiles.**

— `\?` : aide
— `\q` : quitter `psql`
— `\d` : liste des tables
— `\d Table` : description de la table `d`
— `\! clear` : efface l'écran
— `\set AUTOCOMMIT {ON|OFF}` : active/désactive le mode `AUTOCOMMIT`

▶ **Prompt `psql`.** Le prompt de `psql` donne des indications sur ce que l'on est en train de faire :

— `psql=>` : prompt de base
— `psql->` : au milieu d'une requête
— `psql(>` : au milieu d'une requête, attend la fermeture d'une parenthèse
— `psql$>` : au milieu d'un bloc PL/pgSQL
— `psql=*>` : transaction en cours, en attente de commit ou rollback
— `psql=!>` : transaction en cours où il y a eu une erreur, plus aucune requête ne sera exécuté jusqu'à un rollback

Pour annuler la saisie d'une commande en cours : `Ctrl+C`

▶ **Aide en ligne :** `https://s2i.iut.uca.fr/page/documentation/sgbd/`

Pour installer PostgreSQL sur votre ordinateur personnel :
`https://wiki.postgresql.org/wiki/Detailed_installation_guides`