Module S30

Création d’objets et mises à jour

25.09

Table des matières

Sur ce document

Formation	Module S30
Titre	Création d'objets et mises à jour
Révision	25.09
PDF	https://dali.bo/s30_pdf
EPUB	https://dali.bo/s30_epub
HTML	https://dali.bo/s30_html
Slides	https://dali.bo/s30_slides
TP	https://dali.bo/s30_tp
TP (solutions)	https://dali.bo/s30_solutions

Vous trouverez en ligne les différentes versions complètes de ce document.

Chers lectrices & lecteurs,

Nos formations PostgreSQL sont issues de nombreuses années d'études, d'expérience de terrain et de passion pour les logiciels libres. Pour Dalibo, l'utilisation de PostgreSQL n'est pas une marque d'opportunisme commercial, mais l'expression d'un engagement de longue date. Le choix de l'Open Source est aussi le choix de l'implication dans la communauté du logiciel.

Au-delà du contenu technique en lui-même, notre intention est de transmettre les valeurs qui animent et unissent les développeurs de PostgreSQL depuis toujours : partage, ouverture, transparence, créativité, dynamisme… Le but premier de nos formations est de vous aider à mieux exploiter toute la puissance de PostgreSQL mais nous espérons également qu'elles vous inciteront à devenir un membre actif de la communauté en partageant à votre tour le savoir-faire que vous aurez acquis avec nous.

Nous mettons un point d'honneur à maintenir nos manuels à jour, avec des informations précises et des exemples détaillés. Toutefois malgré nos efforts et nos multiples relectures, il est probable que ce document contienne des oublis, des coquilles, des imprécisions ou des erreurs. Si vous constatez un souci, n'hésitez pas à le signaler via l'adresse formation@dalibo.com[1] !

À propos de DALIBO

DALIBO est le spécialiste français de PostgreSQL. Nous proposons du support, de la formation et du conseil depuis 2005.

Retrouvez toutes nos formations sur https://dalibo.com/formations

[1] mailto:formation@dalibo.com

Remerciements

Ce manuel de formation est une aventure collective qui se transmet au sein de notre société depuis des années. Nous remercions chaleureusement ici toutes les personnes qui ont contribué directement ou indirectement à cet ouvrage, notamment :

Alexandre Anriot, Jean-Paul Argudo, Carole Arnaud, Alexandre Baron, David Bidoc, Sharon Bonan, Franck Boudehen, Arnaud Bruniquel, Pierrick Chovelon, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Ronan Dunklau, Vik Fearing, Stefan Fercot, Dimitri Fontaine, Pierre Giraud, Nicolas Gollet, Nizar Hamadi, Florent Jardin, Virginie Jourdan, Luc Lamarle, Denis Laxalde, Guillaume Lelarge, Alain Lesage, Benoit Lobréau, Jean-Louis Louër, Thibaut Madelaine, Cédric Martin, Adrien Nayrat, Alexandre Pereira, Flavie Perette, Robin Portigliatti, Thomas Reiss, Maël Rimbault, Jehan-Guillaume de Rorthais, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Arnaud de Vathaire, Cédric Villemain, Thibaud Walkowiak, Frédéric Yhuel.

Forme de ce manuel

Les versions PDF, EPUB ou HTML de ce document sont structurées autour des slides de nos formations. Le texte suivant chaque slide contient le cours et de nombreux détails qui ne peuvent être données à l'oral.

Licence Creative Commons CC-BY-NC-SA

Cette licence interdit la réutilisation pour l'apprentissage d'une IA. Elle couvre les diapositives, les manuels eux-mêmes et les travaux pratiques.

Cette formation peut également contenir quelques images et schémas dont la redistribution est soumise à des licences différentes qui sont alors précisées.

Marques déposées

PostgreSQL® Postgres® et le logo Slonik sont des marques déposées[3] par PostgreSQL Community Association of Canada.

Versions de PostgreSQL couvertes

Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 13 à 17.

Sur les versions précédentes susceptibles d'être encore rencontrées en production, seuls quelques points très importants sont évoqués, en plus éventuellement de quelques éléments historiques.

Sauf précision contraire, le système d'exploitation utilisé est Linux.

[3] https://www.postgresql.org/about/policies/trademarks/

1/ Création d’objet et mises à jour

1.1 INTRODUCTION

— DDL, gérer les objets
— DML, écrire des données
— Gérer les transactions

Le module précédent nous a permis de voir comment lire des données à partir de requêtes SQL. Ce module a pour but de présenter la création et la gestion des objets dans la base de données (par exemple les tables), ainsi que l'ajout, la suppression et la modification de données.

Une dernière partie sera consacrée aux transactions.

1.1.1 Menu

— DDL (`Data Definition Language`)
— DML (`Data Manipulation Language`)
— TCL (`Transaction Control Language`)

1.1.2 Objectifs

— Savoir créer, modifier et supprimer des objets
— Savoir utiliser les contraintes d'intégrité
— Savoir mettre à jour les données
— Savoir utiliser les transactions

1.2 DDL

- DDL
  - `Data Definition Language`
  - langage de définition de données
- Permet de définir des objets dans la base de données

Les ordres DDL (acronyme de *Data Definition Language*) permettent de définir des objets dans la base de données et notamment la structure de base du standard SQL : les tables.

1.2.1 Objets d'une base de données

- Objets définis par la norme SQL :
  - schémas
  - séquences
  - tables
  - contraintes
  - domaines
  - vues
  - fonctions
  - triggers

La norme SQL définit un certain nombre d'objets standards qu'il est possible de créer en utilisant les ordres DDL. D'autres types d'objets existent bien entendu, comme les domaines. Les ordres DDL permettent également de créer des index, bien qu'ils ne soient pas définis dans la norme SQL.

La seule structure de données possible dans une base de données relationnelle est la table.

1.2.2 Créer des objets

- Ordre `CREATE`
- Syntaxe spécifique au type d'objet
- Exemple :

```
CREATE SCHEMA s1;
```

La création d'objet passe généralement par l'ordre `CREATE`. La syntaxe dépend fortement du type d'objet. Voici trois exemples :

```
CREATE SCHEMA s1;
CREATE TABLE t1 (c1 integer, c2 text);
CREATE SEQUENCE s1 INCREMENT BY 5 START 10;
```

Pour créer un objet, il faut être propriétaire du schéma ou de la base auquel appartiendra l'objet ou avoir le droit `CREATE` sur le schéma ou la base.

1.2.3 Modifier des objets

- Ordre `ALTER`
- Syntaxe spécifique pour modifier la définition d'un objet
- Exemple :
  - renommage

```
ALTER type_objet ancien_nom RENAME TO nouveau_nom ;
```

  - changement de propriétaire

```
ALTER type_objet nom_objet OWNER TO proprietaire ;
```

  - changement de schéma

```
ALTER type_objet nom_objet SET SCHEMA nom_schema ;
```

Modifier un objet veut dire modifier ses propriétés. On utilise dans ce cas l'ordre `ALTER`. Il faut être propriétaire de l'objet pour pouvoir le faire.

Deux propriétés sont communes à tous les objets : le nom de l'objet et son propriétaire. Deux autres sont fréquentes et dépendent du type de l'objet : le schéma et le tablespace. Les autres propriétés dépendent directement du type de l'objet.

1.2.4 Supprimer des objets

- Ordre `DROP`
- Exemples :
  - supprimer un objet :

```
DROP type_objet nom_objet ;
```

  - supprimer un objet et ses dépendances :

```
DROP type_objet nom_objet CASCADE ;
```

Seul un propriétaire peut supprimer un objet. Il utilise pour cela l'ordre `DROP`. Pour les objets ayant des dépendances, l'option `CASCADE` permet de tout supprimer d'un coup. C'est très pratique, et c'est en même temps très dangereux : il faut donc utiliser cette option à bon escient.

Si un objet dépendant de l'objet à supprimer a lui aussi une dépendance, sa dépendance sera également supprimée. Ainsi de suite jusqu'à la dernière dépendance.

1.2.5 Schéma

- Identique à un espace de nommage
- Permet d'organiser les tables de façon logique
- Possibilité d'avoir des objets de même nom dans des schémas différents
- Pas d'imbrication (contrairement à des répertoires par exemple)
- Schéma `public`
  - existe par défaut dans une base PostgreSQL
  - ouvert en écriture par défaut ! (< v15)
  - parfois supprimé pour la sécurité

La notion de schéma dans PostgreSQL est à rapprocher de la notion d'espace de nommage (ou *namespace*) de certains langages de programmation. Le catalogue système qui contient la définition des schémas dans PostgreSQL s'appelle d'ailleurs `pg_namespace`.

Les schémas sont utilisés pour répartir les objets de façon purement logique, suivant un schéma interne à l'organisation. Ils servent aussi à faciliter la gestion des droits (il suffit de révoquer le droit d'utilisation d'un schéma à un utilisateur pour que les objets contenus dans ce schéma ne soient plus accessibles à cet utilisateur).

Un schéma `public` est créé par défaut dans toute nouvelle base de données.

Jusque PostgreSQL 14, tout le monde a le droit d'y créer des objets. Cela posait des problèmes de sécurité, donc ce droit était souvent révoqué, ou le schéma `public` supprimé.
À partir de PostgreSQL 15, le droit d'écriture dans `public` doit être donné explicitement.

Il existe des schémas système masqués, par exemple pour des tables temporaires ou les tables système (schéma `pg_catalog`).

1.2.6 Gestion d'un schéma

- `CREATE SCHEMA nom_schéma`
- `ALTER SCHEMA nom_schéma`
  - renommage
  - changement de propriétaire
- `DROP SCHEMA [ IF EXISTS ] nom_schéma [ CASCADE ]`

L'ordre `CREATE SCHEMA` permet de créer un schéma. Il suffit de lui spécifier le nom du schéma. `CREATE SCHEMA` offre d'autres possibilités qui sont rarement utilisées.

L'ordre `ALTER SCHEMA nom_schema RENAME TO nouveau_nom_schema` permet de renommer un schéma. L'ordre `ALTER SCHEMA nom_schema OWNER TO proprietaire` permet de changer le propriétaire d'un schéma.

Enfin, l'ordre `DROP SCHEMA` permet de supprimer un schéma si il est vide. La clause `IF EXISTS` permet d'éviter la levée d'une erreur si le schéma n'existe pas (très utile dans les scripts SQL). La clause `CASCADE` permet de supprimer le schéma ainsi que tous les objets qui sont positionnés dans le schéma.

Exemples

Création d'un schéma `reference` :

```
CREATE SCHEMA reference;
```

Une table peut être créée dans ce schéma :

```
CREATE TABLE reference.communes (
  commune      text,
  codepostal   char(5),
  departement  text,
  codeinsee    integer
);
```

La suppression directe du schéma ne fonctionne pas car il porte encore la table `communes` :

```
DROP SCHEMA reference;
ERROR:  cannot drop schema reference because other objects depend on it
DETAIL:  table reference.communes depends on schema reference
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

L'option `CASCADE` permet de supprimer le schéma et ses objets dépendants :

```
DROP SCHEMA reference CASCADE;
NOTICE:  drop cascades to table reference.communes
```

1.2.7 Accès aux objets

- Nommage explicite
  - `nom_schema`.`nom_objet`
- Chemin de recherche de schéma
  - paramètre `search_path`
  - `SET search_path = schema1,schema2,public;`
  - par défaut : `$user, public`

Le paramètre `search_path` permet de définir un chemin de recherche pour pouvoir retrouver les tables dont le nom n'est pas qualifié par le nom de son schéma. PostgreSQL procèdera de la même façon que le système avec la variable `$PATH` : il recherche la table dans le premier schéma listé. S'il trouve une table portant ce nom dans le schéma, il préfixe le nom de table avec celui du schéma. S'il ne trouve pas de table de ce nom dans le schéma, il effectue la même opération sur le prochain schéma de la liste du `search_path`. S'il n'a trouvé aucune table de ce nom dans les schémas listés par `search_path`, PostgreSQL lève une erreur.

Comme beaucoup d'autres paramètres, le `search_path` peut être positionné à différents endroits. Par défaut, il est assigné à `$user, public`, c'est-à-dire que le premier schéma de recherche portera le nom de l'utilisateur courant, et le second schéma de recherche est `public`.

Il est possible de vérifier la configuration de la variable `search_path` à l'aide de la commande `SHOW` :

```
SHOW search_path;

  search_path
----------------
 "$user",public
```

Pour obtenir une configuration particulière, la variable `search_path` peut être positionnée dans le fichier `postgresql.conf` :

```
search_path = '"$user",public'
```

Cette variable peut aussi être positionnée au niveau d'un utilisateur. Chaque fois que l'utilisateur se connectera, il prendra le `search_path` de sa configuration spécifique :

```
ALTER ROLE nom_role SET search_path = "$user", public;
```

Cela peut aussi se faire au niveau d'une base de données. Chaque fois qu'un utilisateur se connectera à la base, il prendra le `search_path` de cette base, sauf si l'utilisateur a déjà une configuration spécifique :

```
ALTER DATABASE nom_base SET search_path = "$user", public;
```

La variable `search_path` peut également être positionnée pour un utilisateur particulier, dans une base particulière :

```
ALTER ROLE nom_role IN DATABASE nom_base SET search_path = "$user", public;
```

Enfin, la variable `search_path` peut être modifiée dynamiquement dans la session avec la commande `SET` :

```
SET search_path = "$user", public;
```

1.2.8 Séquences

- Séquence
  - génère une séquence de nombres
- Paramètres
  - valeur minimale `MINVALUE`
  - valeur maximale `MAXVALUE`
  - valeur de départ `START`
  - incrément `INCREMENT`
  - cache `CACHE`
  - cycle autorisé `CYCLE`

Les séquences sont des objets standards qui permettent de générer des séries de valeur. Elles sont utilisées notamment pour générer un numéro unique pour un identifiant ou, plus rarement, pour disposer d'un compteur informatif, mis à jour au besoin.

Le cache de la séquence a pour effet de générer un certain nombre de valeurs en mémoire afin de les mettre à disposition de la session qui a utilisé la séquence. Même si les valeurs pré-calculées ne sont pas consommées dans la session, elles seront consommées au niveau de la séquence. Cela peut avoir pour effet de créer des trous dans les séquences d'identifiants et de consommer très rapidement les numéros de séquence possibles. Le cache de séquence n'a pas besoin d'être ajusté sur des applications réalisant de petites transactions. Il permet en revanche d'améliorer les performances sur des applications qui utilisent massivement des numéros de séquences, notamment pour réaliser des insertions massives.

1.2.9 Création d'une séquence

```
CREATE SEQUENCE nom [ INCREMENT incrément ]
    [ MINVALUE valeurmin | NO MINVALUE ]
    [ MAXVALUE valeurmax | NO MAXVALUE ]
    [ START [ WITH ] début ]
    [ CACHE cache ]
    [ [ NO ] CYCLE ]
    [ OWNED BY { nom_table.nom_colonne | NONE } ]
```

La syntaxe complète est donnée dans le slide.

Le mot clé `TEMPORARY` ou `TEMP` permet de définir si la séquence est temporaire. Si tel est le cas, elle sera détruite à la déconnexion de l'utilisateur.

Le mot clé `INCREMENT` définit l'incrément de la séquence, `MINVALUE`, la valeur minimale de la séquence et `MAXVALUE`, la valeur maximale. `START` détermine la valeur de départ initiale de la séquence, c'est-à-dire juste après sa création. La clause `CACHE` détermine le cache de séquence. `CYCLE` permet d'indiquer au SGBD que la séquence peut reprendre son compte à `MINVALUE` lorsqu'elle aura atteint `MAXVALUE`. La clause `NO CYCLE` indique que le rebouclage de la séquence est interdit, PostgreSQL lèvera alors une erreur lorsque la séquence aura atteint son `MAXVALUE`. Enfin, la clause `OWNED BY` détermine l'appartenance d'une séquence à une colonne d'une table. Ainsi, si la colonne est supprimée, la séquence sera implicitement supprimée.

Exemple de séquence avec rebouclage :

```
CREATE SEQUENCE testseq INCREMENT BY 1 MINVALUE 3 MAXVALUE 5 CYCLE START WITH 4;

SELECT nextval('testseq');

 nextval
---------
       4

SELECT nextval('testseq');

 nextval
---------
       5

SELECT nextval('testseq');

 nextval
---------
       3
```

1.2.10 Modification d'une séquence

```
ALTER SEQUENCE nom [ INCREMENT increment ]
    [ MINVALUE valeurmin | NO MINVALUE ]
    [ MAXVALUE valeurmax | NO MAXVALUE ]
    [ START [ WITH ] début ]
    [ RESTART [ [ WITH ] nouveau_début ] ]
    [ CACHE cache ] [ [ NO ] CYCLE ]
    [ OWNED BY { nom_table.nom_colonne | NONE } ]
```

— Il est aussi possible de modifier
  — le propriétaire
  — le schéma

Les propriétés de la séquence peuvent être modifiées avec l'ordre `ALTER SEQUENCE`.

La séquence peut être affectée à un nouveau propriétaire :

```
ALTER SEQUENCE [ IF EXISTS ] nom OWNER TO nouveau_propriétaire
```

Elle peut être renommée :

```
ALTER SEQUENCE [ IF EXISTS ] nom RENAME TO nouveau_nom
```

Enfin, elle peut être positionnée dans un nouveau schéma :

```
ALTER SEQUENCE [ IF EXISTS ] nom SET SCHEMA nouveau_schema
```

1.2.11 Suppression d'une séquence

```
DROP SEQUENCE nom [, …]
```

Voici la syntaxe complète de `DROP SEQUENCE` :

```
DROP SEQUENCE [ IF EXISTS ] nom [, …] [ CASCADE | RESTRICT ]
```

Le mot clé `CASCADE` permet de supprimer la séquence ainsi que tous les objets dépendants (par exemple la valeur par défaut d'une colonne).

1.2.12 Séquences, utilisation

- Obtenir la valeur suivante
  - `nextval('nom_sequence')`
- Obtenir la valeur courante
  - `currval('nom_sequence')`
  - mais `nextval()` doit être appelé avant dans la même session
- Ne pas utiliser pour une numérotation « sans trou » !

La fonction `nextval()` permet d'obtenir le numéro de séquence suivant. Son comportement n'est pas transactionnel. Une fois qu'un numéro est consommé, il n'est pas possible de revenir dessus, malgré un `ROLLBACK` de la transaction. La séquence est le seul objet à avoir un comportement de ce type. C'est cependant nécessaire, notamment pour des raisons de performance.

La fonction `currval()` permet d'obtenir le numéro de séquence courant, mais son usage nécessite d'avoir utilisé `nextval()` dans la même session.

Il est possible d'interroger une séquence avec une requête `SELECT`. Cela permet d'obtenir des informations sur la séquence, dont la dernière valeur utilisée dans la colonne `last_value`. Cet usage n'est pas recommandé en production et doit plutôt être utilisé à titre informatif.

Exemples

Utilisation d'une séquence simple :

```
CREATE SEQUENCE testseq
INCREMENT BY 1 MINVALUE 10 MAXVALUE 20 START WITH 15 CACHE 1;

SELECT currval('testseq');

ERROR:  currval of sequence "testseq" is not yet defined in this session

SELECT * FROM testseq ;

- [ RECORD 1 ]-+--------
sequence_name  | testseq
last_value     | 15
start_value    | 15
increment_by   | 1
max_value      | 20
min_value      | 10
cache_value    | 5
log_cnt        | 0
is_cycled      | f
is_called      | f

SELECT nextval('testseq');

 nextval
---------
      15
```

```
SELECT currval('testseq');
 currval
---------
      15
SELECT nextval('testseq');
 nextval
---------
      16
ALTER SEQUENCE testseq RESTART WITH 5;
ERROR:  RESTART value (5) cannot be less than MINVALUE (10)
DROP SEQUENCE testseq;
```

Utilisation d'une séquence simple avec cache :

```
CREATE SEQUENCE testseq INCREMENT BY 1 CACHE 10;
SELECT nextval('testseq');
 nextval
---------
       1
```

Déconnexion et reconnexion de l'utilisateur :

```
SELECT nextval('testseq');
 nextval
---------
      11
```

Suppression en cascade d'une séquence :

```
CREATE TABLE t2 (id serial);
\d t2
                          Table "s2.t2"
 Column |  Type   |                     Modifiers
--------+---------+---------------------------------------------------
 id     | integer | not null default nextval('t2_id_seq'::regclass)
DROP SEQUENCE t2_id_seq;
ERROR:  cannot drop sequence t2_id_seq because other objects depend on it
DETAIL:  default for table t2 column id depends on sequence t2_id_seq
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
DROP SEQUENCE t2_id_seq CASCADE;
NOTICE:  drop cascades to default for table t2 column id
\d t2
        Table "s2.t2"
 Column |  Type   | Modifiers
--------+---------+-----------
 id     | integer | not null
```

Une séquence n'est pas le bon outil s'il vous faut générer des suites de nombres « sans trou », par exemple des numéros de factures. Une séquence est conçue d'abord pour livrer des numéros uniques, et certaines valeurs peuvent être « perdues ».
Une suite unique réclame des techniques plus complexes nécessitant un verrouillage plus lourd.

1.2.13 Type SERIAL

- Type `serial` / `bigserial` / `smallserial`
  - séquence générée automatiquement
  - valeur par défaut `nextval(…)`
- Préférer un entier avec `IDENTITY`

Certaines bases de données offrent des colonnes auto-incrémentées (`autoincrement` de MySQL ou `identity` de SQL Server).

PostgreSQL possède `identity` à partir de PostgreSQL 10. Il était déjà possible d'utiliser `serial`, un équivalent qui s'appuie sur les séquences et la possibilité d'appliquer une valeur par défaut à une colonne.

Par exemple, si l'on crée la table suivante :

```
CREATE TABLE exemple_serial (
  id SERIAL PRIMARY KEY,
  valeur INTEGER NOT NULL
);
```

On s'aperçoit que la table a été créée telle que demandé, mais qu'une séquence a aussi été créée. Elle porte un nom dérivé de la table associé à la colonne correspondant au type `serial`, terminé par `seq` :

```
\d
                List of relations
 Schema |          Name           |   Type   | Owner
--------+-------------------------+----------+--------
 public | exemple_serial          | table    | thomas
 public | exemple_serial_id_seq | sequence | thomas
```

En examinant plus précisément la définition de la table, on s'aperçoit que la colonne `id` porte une valeur par défaut qui correspond à l'appel de la fonction `nextval()` sur la séquence qui a été créée implicitement :

```
\d exemple_serial
                        Table "public.exemple_serial"
 Column |  Type   |                          Modifiers
--------+---------+-------------------------------------------------------------
```

```
 id     | integer | not null default nextval('exemple_serial_id_seq'::regclass)
 valeur | integer | not null
Indexes:
    "exemple_serial_pkey" PRIMARY KEY, btree (id)
```

`smallserial` et `bigserial` sont des variantes de `serial` s'appuyant sur des types d'entiers plus courts ou plus longs.

1.2.14 Domaines

— Permet d'associer
  — un type standard
  — et une contrainte (optionnelle)

Un domaine est un type standard (numérique, texte…) auquel ont été ajoutées des contraintes particulières.

Les domaines sont utiles pour ne pas définir les mêmes contraintes sur plusieurs colonnes. La maintenance en est ainsi facilitée.

Les domaines sont utiles pour ramener la définition de contraintes communes à plusieurs colonnes sur un seul objet. La maintenance en est ainsi facilitée.

L'ordre `CREATE DOMAIN` permet de créer un domaine, `ALTER DOMAIN` permet de modifier sa définition, et enfin, `DROP DOMAIN` permet de supprimer un domaine.

Exemples : gestion d'un domaine `salaire` :

Commençons par le domaine et une table d'exemple :

```
CREATE DOMAIN salaire AS integer CHECK (VALUE > 0);
CREATE TABLE employes (id serial, nom text, paye salaire);
```

```
\d employes
                              Table « public.employes »
 Colonne |  Type   |  NULL-able |              Par défaut
---------+---------+------------+-------------------------------------
 id      | integer |  not null  | nextval('employes_id_seq'::regclass)
 nom     | text    |            |
 paye    | salaire |            |
```

Insérons des données dans la table :

```
INSERT INTO employes (nom, paye) VALUES ('Albert', 1500);
INSERT INTO employes (nom, paye) VALUES ('Alphonse', 0);
```

```
ERROR:  value for domain salaire violates check constraint "salaire_check"
```

L'erreur ci-dessus est logique vu qu'on ne peut avoir qu'un entier strictement positif.

```
INSERT INTO employes (nom, paye) VALUES ('Alphonse', 1000);
INSERT INTO employes (nom, paye) VALUES ('Bertrand', NULL);
```

Tous les employés doivent avoir un salaire. Il faut donc modifier la contrainte pour s'assurer qu'aucune valeur `NULL` (vide) ne soit saisie

```
ALTER DOMAIN salaire SET NOT NULL;

ERROR:  column "paye" of table "employes" contains null values
```

En effet, une ligne avec `NULL` est déjà présente, il faut la corriger pour pouvoir ajouter la contrainte.

```
UPDATE employes SET paye=1500 WHERE nom='Bertrand';

ALTER DOMAIN salaire SET NOT NULL;
INSERT INTO employes (nom, paye) VALUES ('Delphine', NULL);

ERROR:  domain salaire does not allow null values
```

La contrainte est donc bien vérifiée, et la ligne avec `NULL` rejetée.

Supprimons maintenant la contrainte :

```
DROP DOMAIN salaire;

ERROR:  cannot drop type salaire because other objects depend on it
DETAIL:  table employes column paye depends on type salaire
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

Il n'est pas possible de supprimer le domaine car il est référencé dans une table. Il faut donc utiliser l'option `CASCADE` pour détruire aussi les objets qui dépendent du domaine.

Soyez très prudent en supprimant des objets avec `CASCADE` !

```
DROP DOMAIN salaire CASCADE;

NOTICE:  drop cascades to table employes column paye
DROP DOMAIN
```

Le domaine a été supprimée ainsi que toutes les colonnes de ce type :

```
\d employes
                              Table « public.employes »
 Colonne |  Type   | NULL-able |              Par défaut
---------+---------+-----------+---------------------------------------
 id      | integer | not null  | nextval('employes_id_seq'::regclass)
 nom     | text    |           |
```

Exemples : création et utilisation d'un domaine `code_postal_us` :

```
CREATE DOMAIN code_postal_us AS TEXT
CHECK(
   VALUE ~ '^\d{5}$'
OR VALUE ~ '^\d{5}-\d{4}$'
);

CREATE TABLE courrier_us (
```

```
  id_adresse SERIAL PRIMARY KEY,
  rue1 TEXT NOT NULL,
  rue2 TEXT,
  rue3 TEXT,
  ville TEXT NOT NULL,
  code_postal code_postal_us NOT NULL
);

INSERT INTO courrier_us (rue1,ville,code_postal)
VALUES ('51 Franklin Street', 'Boston, MA', '02110-1335' );

INSERT INTO courrier_us (rue1,ville,code_postal)
VALUES ('10 rue d''Uzès','Paris','F-75002') ;

ERREUR:  la valeur pour le domaine code_postal_us viole la contrainte de
vérification « code_postal_us_check »
```

1.2.15 Tables

- Équivalent ensembliste d'une relation
- Composé principalement de
  - colonnes ordonnées
  - contraintes

La table est l'élément de base d'une base de données. Elle est composée de colonnes (à sa création) et est remplie avec des enregistrements (lignes de la table). Sa définition peut aussi faire intervenir des contraintes, qui sont au niveau table ou colonne.

1.2.16 Création d'une table

- Définition de son nom
- Définition de ses colonnes
  - nom, type, contraintes éventuelles
- Clauses de stockage
- `CREATE TABLE`

Pour créer une table, il faut donner son nom et la liste des colonnes. Une colonne est définie par son nom et son type, mais aussi des contraintes optionnelles.

Des options sont possibles pour les tables, comme les clauses de stockage. Dans ce cas, on sort du contexte logique pour se placer au niveau physique.

1.2.17 CREATE TABLE

```
CREATE TABLE nom_table (
  definition_colonnes
  definition_contraintes
) clause_stockage;
```

La création d'une table passe par l'ordre `CREATE TABLE`. La définition des colonnes et des contraintes sont entre parenthèse après le nom de la table.

1.2.18 Définition des colonnes

```
nom_colonne type [ COLLATE collation ] [ contrainte ]
[, …]
```

Les colonnes sont indiquées l'une après l'autre, en les séparant par des virgules.

Deux informations sont obligatoires pour chaque colonne : le nom et le type de la colonne. Dans le cas d'une colonne contenant du texte, il est possible de fournir le collationnement de la colonne. Quelle que soit la colonne, il est ensuite possible d'ajouter des contraintes.

1.2.19 Valeur par défaut

- `DEFAULT`
  - affectation implicite
- Utiliser directement par les types sériés

La clause `DEFAULT` permet d'affecter une valeur par défaut lorsqu'une colonne n'est pas référencée dans l'ordre d'insertion ou si une mise à jour réinitialise la valeur de la colonne à sa valeur par défaut.

Les types sériés définissent une valeur par défaut sur les colonnes de ce type. Cette valeur est le retour de la fonction `nextval()` sur la séquence affectée automatiquement à cette colonne.

Exemples

Assignation d'une valeur par défaut :

```
CREATE TABLE valdefaut (
  id integer,
  i integer DEFAULT 0,
  j integer DEFAULT 0
);

INSERT INTO valdefaut (id, i) VALUES (1, 10);

SELECT * FROM valdefaut ;
 id | i  | j
----+----+---
  1 | 10 | 0
(1 row)
```

1.2.20 Copie de la définition d'une table

- Création d'une table à partir d'une autre table
  - `CREATE TABLE … (LIKE table clause_inclusion)`
- Avec les valeurs par défaut des colonnes :
  - `INCLUDING DEFAULTS`
- Avec ses autres contraintes :
  - `INCLUDING CONSTRAINTS`
- Avec ses index :
  - `INCLUDING INDEXES`

L'ordre `CREATE TABLE` permet également de créer une table à partir de la définition d'une table déjà existante en utilisant la clause `LIKE` en lieu et place de la définition habituelle des colonnes. Par défaut, seule la définition des colonnes avec leur typage est repris.

Les clauses `INCLUDING` permettent de récupérer d'autres éléments de la définition de la table, comme les valeurs par défaut (`INCLUDING DEFAULTS`), les contraintes d'intégrité (`INCLUDING CONSTRAINTS`), les index (`INCLUDING INDEXES`), les clauses de stockage (`INCLUDING STORAGE`) ainsi que les commentaires (`INCLUDING COMMENTS`). Si l'ensemble de ces éléments sont repris, il est possible de résumer la clause `INCLUDING` à `INCLUDING ALL`.

La clause `CREATE TABLE` suivante permet de créer une table `archive_evenements_2010` à partir de la définition de la table `evenements` :

```
CREATE TABLE archive_evenements_2010
 (LIKE evenements
  INCLUDING DEFAULTS
  INCLUDING CONSTRAINTS
  INCLUDING INDEXES
  INCLUDING STORAGE
  INCLUDING COMMENTS
);
```

Elle est équivalente à :

```
CREATE TABLE archive_evenements_2010
 (LIKE evenements
  INCLUDING ALL
);
```

1.2.21 Modification d'une table

- `ALTER TABLE`
- Définition de la table
  - renommage de la table
  - ajout/modification/suppression d'une colonne
  - déplacement dans un schéma différent
  - changement du propriétaire
- Définition des colonnes
  - renommage d'une colonne
  - changement de type d'une colonne
- Définition des contraintes
  - ajout/suppression d'une contrainte

Pour modifier la définition d'une table (et non pas son contenu), il convient d'utiliser l'ordre `ALTER TABLE`. Il permet de traiter la définition de la table (nom, propriétaire, schéma, liste des colonnes), la définition des colonnes (ajout, modification de nom et de type, suppression… mais pas de changement au niveau de leur ordre), et la définition des contraintes (ajout et suppression).

1.2.22 Conséquences des modifications d'une table

- contention avec les verrous
- vérification des données
- performance avec une possible réécriture de la table

Suivant l'opération réalisée, les verrous posés ne seront pas les mêmes, même si le verrou par défaut sera un verrou exclusif. Par exemple, renommer une table nécessite un verrou exclusif mais changer la taille de l'échantillon statistiques bloque uniquement certaines opérations de maintenance (comme `VACUUM` et `ANALYZE`) et certaines opérations DDL. L'utilisation de la commande `ALTER TABLE` sur un serveur en production doit donc souvent passer par une opération de maintenance planifiée.

Certaines opérations nécessitent de vérifier que les données satisfassent les nouvelles contraintes. C'est évident lors de l'ajout d'une clé primaire ou d'une contrainte `NOT NULL`, par exemple. Or, relire

une grosse table peut donc être très coûteux ! Dans certains cas, PostgreSQL sait éviter un contrôle inutile, par exemple lors d'un passage de `varchar(10)` à `varchar(20)`.

Certaines opérations nécessitent une réécriture de la table. Par exemple, convertir une colonne de type `varchar(5)` vers le type `int` impose une réécriture de la table car il n'y a pas de compatibilité binaire entre les deux types.

Il convient donc d'être très prudent lors de l'utilisation de la commande `ALTER TABLE`. Elle peut poser des problèmes de performances, à cause de verrous posés par d'autres commandes, de verrous qu'elle réclame, de la vérification des données, voire de la réécriture de la table.

1.2.23 Suppression d'une table

— Supprimer une table :

```
DROP TABLE nom_table;
```

— Supprimer une table et tous les objets dépendants :

```
DROP TABLE nom_table CASCADE;
```

L'ordre `DROP TABLE` permet de supprimer une table. L'ordre `DROP TABLE … CASCADE` permet de supprimer une table ainsi que tous ses objets dépendants. Il peut s'agir de séquences rattachées à une colonne d'une table, à des colonnes référençant la table à supprimer, etc.

1.2.24 Contraintes d'intégrité

— ACID
  — **C**ohérence
  — une transaction amène la base d'un état stable à un autre
— Assurent la cohérence des données
  — unicité des enregistrements
  — intégrité référentielle
  — vérification des valeurs
  — identité des enregistrements
  — règles sémantiques

Les données dans les différentes tables ne sont pas indépendantes mais obéissent à des règles sémantiques mises en place au moment de la conception du modèle de données. Les contraintes d'intégrité ont pour principal objectif de garantir la cohérence des données entre elles, et donc de veiller à ce

qu'elles respectent ces règles sémantiques. Si une insertion, une mise à jour ou une suppression viole ces règles, l'opération est purement et simplement annulée.

1.2.25 Clé primaire d'une table

- Identifie une ligne de manière unique
- Une seule clé primaire par table
- Une ou plusieurs colonnes
- À choisir parmi les clés candidates
  - clé naturelle ou artificielle (technique, invisible)
  - les autres clés possibles peuvent être `UNIQUE`
- Index automatique

Une « clé primaire » permet d'identifier une ligne de façon unique.

Il n'existe qu'une seule clé primaire par table.

Une clé primaire exige que toutes les valeurs de la ou des colonnes qui composent cette clé soient uniques et non nulles. La clé peut être composée d'une seule colonne ou de plusieurs colonnes, selon le besoin.

La clé primaire est déterminée au moment de la conception du modèle de données. Cette clé peut être « naturelle » et visible de l'utilisateur, ou purement technique et non visible. Le débat entre les deux modélisations a longtemps fait rage.

Clé primaire naturelle :

Par exemple, une table des factures peut avoir comme clé primaire « naturelle » le numéro de la facture. C'est le plus intuitif.

De nombreuses applications utilisent des « clés naturelles », par exemple un numéro de commande ou de Sécurité Sociale. Ces clés peuvent même être composées de plusieurs champs.

Cela peut poser des soucis techniques. Par exemple, il y a des contraintes légales sur l'incrémentation des numéros de facture qui obligent à le générer tard dans le processus. Enregistrer une facture incomplète sans son numéro devient compliqué. Un numéro provisoire est possible, mais le changer implique de modifier aussi toutes les tables qui y font référence.

De manière générale, on évitera toujours de modifier une clé primaire, car elle sert d'identifiant vers la ligne dans d'autres tables, et une modification serait à répercuter dans toutes ces tables !

Liée à notre table des factures, une table des lignes de facture aurait alors comme clé primaire composée le numéro de la facture et le numéro de la ligne. Cela fonctionne, mais une clé composée, d'ailleurs

souvent de type texte, est moins performante pour les jointures qu'un champ monocolonne numérique.

Clé primaire technique :

> Les bonnes pratiques conseillent plutôt d'identifier chaque ligne de chaque table par une clé technique **monocolonne** et **numérique**, de valeur arbitraire et sans aucune signification métier.

Cette *surrogate keys* est donc une clé primaire destinée à résoudre les soucis techniques des clés primaires naturelles.

L'utilisateur final de l'application ne la verra pas.

Cette clé technique doit être générée avec une séquence, ou un UUID (quasi-aléatoire). Ce qui compte est l'unicité.

La clé technique n'a pas à être modifiée puisque sa valeur n'a pas de sens fonctionnel.

Les jointures se font alors sur cet unique champ numérique, de manière efficace.

Les clés « naturelles » restent présentes sous forme d'un champ d'une table, avec une contrainte d'unicité. Si cette clé naturelle doit être modifiée pour une raison ou une autre, il suffit de changer la valeur dans le champ de la table, et il n'y a aucune modification à faire dans les tables possédant une clé étrangère vers cette table.

Dans notre exemple, la table des factures porterait une clé primaire numérique `facture_id` arbitraire, et un champ `numero_facture`, unique, qui, lui, peut être modifié ou être temporairement vide. La table des lignes de facture porterait une clé primaire technique `facture_ligne_id` sans lien, et une clé étrangère (non composée) reprenant `facture_id`, et un champ unique `numero_ligne`.

Exemple 2 :

Pour une assurance ou un garagiste, une table des véhicules ne peut avoir pour clé primaire la plaque d'immatriculation : elle peut changer, elle peut être inconnue, provisoire, absente, voire fausse.

Un identifiant technique est largement préférable.

Exemple 3 :

Une table des clients ne peut pas porter de clé primaire naturelle liée aux nom, prénom, date de naissance… à cause des nombreuses homonymies, et des changements et corrections d'état-civil possibles.

Un code client est plus envisageable, mais il pourrait changer aussi (migration de logiciel, fusion de bases clients…). Le code client peut devenir une clé unique dans une table des clients portant une clé purement technique.

Exemple 4 :

Une table de personnes identifiées par le numéro de Sécurité Sociale ne peut utiliser ce dernier comme clé naturelle : valeur parfois absente ou inconnue, changement possible, voire doublons (!), sans parler de la confidentialité.

Un code de personne propre à l'application est plus pertinent.

Là encore, on créera plutôt une clé technique, et le code de personne et le code de Sécurité Sociale sont juste des champs uniques.

Exemple 5 :

Une table des commandes clients porte une référence vers une table des adresses. Cette table des adresses doit impérativement porter une clé technique : il n'y a guère de sens à créer un « code adresse », et surtout les clients changent régulièrement d'adresse.

Un changement d'adresse d'un client consiste à créer une nouvelle ligne dans la table des adresses et à modifier l'identifiant d'adresse dans la table des clients. Les anciennes commandes et factures doivent en effet conserver l'identifiant des anciennes adresses.

Exemple 6 :

La table des adresses possède un champ renvoyant à une table des pays.

Un code pays ISO est ce qui se rapproche le plus d'une clé naturelle acceptable comme clé technique : courte (`FR` , `DE`), normalisée et sans risque de changement (le code reste identique et n'est pas recyclé).

Index :

La création d'une clé primaire crée implicitement un index sur le champ, pour des raisons de performance.

1.2.26 Déclaration d'une clé primaire

Construction :

```
[CONSTRAINT nom_contrainte]
PRIMARY KEY ( nom_colonne [, … ] )
```

- Séquence
- `serial`
- `GENERATED ALWAYS BY IDENTITY`
- UUID

Exemple avec clé technique manuelle :

```
CREATE TABLE region
  (
    id       int   PRIMARY KEY,
    libelle  text  NOT NULL  UNIQUE
  );

INSERT INTO region VALUES (1, 'Alsace');
INSERT INTO region VALUES (2, 'Île-de-France');
```

La clé primaire est forcément `NOT NULL` : ceci va être rejeté :

```
INSERT INTO region VALUES (NULL, 'Corse');

ERROR:  null value in column "id" of relation "region" violates not-null constraint
DÉTAIL : Failing row contains (null, Corse).

INSERT INTO region VALUES (1, 'Corse');

ERROR:  duplicate key value violates unique constraint "region_pkey"
DÉTAIL : Key (id)=(1) already exists.

TABLE region ;

 id |    libelle
----+---------------
  1 | Alsace
  2 | Île de France
```

Exemple avec séquence :

La séquence génère des identifiants que l'on peut utiliser :

```
CREATE SEQUENCE departement_seq AS int ;

CREATE TABLE departement
  (
    id       int   PRIMARY KEY,
    libelle  text  NOT NULL UNIQUE,
    code     varchar(3) NOT NULL UNIQUE,
    region_id int NOT NULL REFERENCES region (id)
  );

INSERT INTO departement
SELECT nextval ('departement_seq'), 'Bas-Rhin', '67', 1 ;

INSERT INTO departement
VALUES ( nextval ('departement_seq'),  'Haut-Rhin', '68', 1 ),
       ( nextval ('departement_seq'),  'Paris',     '75', 2 ) ;
```

Noter que cette dernière erreur va « consommer » un numéro de séquence, ce qui en fait n'a pas d'importance pour une clé technique.

```
INSERT INTO departement
SELECT nextval ('departement_seq'), 'Yvelines', '78', null ;

ERROR:  null value in column "region_id" of relation "departement" violates not-null
 ↪  constraint
DETAIL : Failing row contains (4, Yvelines, 78, null).

WITH nouveaudept AS (
  INSERT INTO departement
  SELECT nextval ('departement_seq'), 'Yvelines', '78', 2
  RETURNING *)
SELECT * FROM nouveaudept ;

 id | libelle  | code | region_id
----+----------+------+-----------
   | Yvelines | 78   |         2
```

```
TABLE departement ;
```

```
 id |  libelle   | code | region_id
----+------------+------+-----------
  1 | Bas-Rhin   | 67   |         1
  2 | Haut-Rhin  | 68   |         1
  3 | Paris      | 75   |         2
  5 | Yvelines   | 78   |         2
```

Exemples avec serial :

Noter que `serial` est considéré comme un type (il existe aussi `bigserial`). Il y a création d'une séquence en arrière-plan, utilisée de manière transparente.

```
CREATE TABLE ville
  (
    id       serial   PRIMARY KEY,
    libelle  text     NOT NULL,
    departement_id int NOT NULL REFERENCES departement
  );

INSERT INTO ville (libelle, departement_id) VALUES ('Strasbourg',1);
INSERT INTO ville (libelle, departement_id) VALUES ('Paris',3);
```

Interférer avec l'autoincrémentation est une très mauvaise idée :

```
INSERT INTO ville VALUES (3, 'Mulhouse', 2);
INSERT INTO ville (libelle, departement_id) VALUES ('Sélestat',1);

ERROR:  duplicate key value violates unique constraint "ville_pkey"
DETAIL : Key (id)=(3) already exists.
```

Exemple avec IDENTITY :

```
CREATE TABLE ville2
  (
    id       int   GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    libelle  text  NOT NULL,
    departement_id int NOT NULL REFERENCES departement
  );
```

Le maniement est identique au `serial`. Il faut préférer l'identité pour des raisons de respect de standard SQL et à cause de quelques spécificités dans la gestion automatique des séquences sous-jacentes.

Exemple avec une clé composée :

Nous verrons plus loin un exemple avec une clé primaire composée, qui doit se déclarer alors après les colonnes :

```
CREATE TABLE stock
  (
    vin_id        int    not null,
    contenant_id  int    not null,
    annee         int4   not null,
    nombre        int4   not null,
```

```
    PRIMARY KEY (vin_id,contenant_id,annee),

    FOREIGN KEY(vin_id) REFERENCES vin(id) ON DELETE CASCADE,
    FOREIGN KEY(contenant_id) REFERENCES contenant(id) ON DELETE CASCADE
  );
```

1.2.27 Contrainte d'unicité

- Garantit l'unicité des valeurs d'une ou plusieurs colonnes
- Clause `UNIQUE`
- Contrainte `UNIQUE` != index `UNIQUE`

Une contrainte d'unicité permet de garantir que les valeurs de la ou des colonnes sur lesquelles porte la contrainte sont uniques.

Une contrainte d'unicité peut être créée simplement en créant un index `UNIQUE` approprié. Ceci est fortement déconseillé du fait que la contrainte ne sera pas référencée comme telle dans le schéma de la base de données. Il sera donc très facile de ne pas la remarquer au moment d'une reprise du schéma pour une évolution majeure de l'application. Une colonne possédant juste l'index `UNIQUE` peut malgré tout être référencée par une clé étrangère.

Les contraintes d'unicité créent implicitement et obligatoirement un index pour implémenter cette unicité.

1.2.28 Déclaration d'une contrainte d'unicité

Construction :

```
[ CONSTRAINT nom_contrainte]
{ UNIQUE { NULLS NOT DISTINCT } ( nom_colonne [, … ] )
```

Voici un exemple complet.

Sans contrainte d'unicité, on peut insérer plusieurs fois la même valeur, sans erreur :

```
CREATE TABLE utilisateurs(id integer);
INSERT INTO utilisateurs VALUES (10);
INSERT INTO utilisateurs VALUES (10);
```

Ce n'est plus le cas en déclarant une contrainte d'unicité :

```
TRUNCATE utilisateurs;
ALTER TABLE utilisateurs ADD UNIQUE(id);
INSERT INTO utilisateurs (id) VALUES (10);
INSERT INTO utilisateurs (id) VALUES (11);
INSERT INTO utilisateurs (id) VALUES (11);
```

```
ERROR:  duplicate key value violates unique constraint "utilisateurs_id_key"
DETAIL:  Key (id)=(11) already exists.
```

Le cas de `NULL` est un peu particulier. Par défaut, plusieurs valeurs `NULL` peuvent figurer dans un index car elles ne sont pas considérées comme égales, mais de valeur inconnue (*unknown*). Par exemple, une table de personnes physiques peut contenir un champ `numero_secu` qui ne doit pas contenir de doublon, mais n'est pas forcément rempli, donc contient de nombreuses valeurs nulles.

Ici, on peut insérer plusieurs valeurs NULL :

```
INSERT INTO utilisateurs (id) VALUES (NULL);
INSERT INTO utilisateurs (id) VALUES (NULL);
INSERT INTO utilisateurs (id) VALUES (NULL);
```

Ce comportement est modifiable en version 15, et une seule valeur `NULL` sera tolérée. Lors de la création de la contrainte, il faut préciser ce nouveau comportement :

```
TRUNCATE utilisateurs;
ALTER TABLE utilisateurs DROP CONSTRAINT utilisateurs_id_key;
ALTER TABLE utilisateurs ADD UNIQUE NULLS NOT DISTINCT(id);
INSERT INTO utilisateurs (id) VALUES (10);
INSERT INTO utilisateurs (id) VALUES (10);

ERROR:  duplicate key value violates unique constraint "utilisateurs_id_key"
DETAIL:  Key (id)=(10) already exists.

INSERT INTO utilisateurs (id) VALUES (11);
INSERT INTO utilisateurs (id) VALUES (NULL);
INSERT INTO utilisateurs (id) VALUES (NULL);

ERROR:  duplicate key value violates unique constraint "utilisateurs_id_key"
DETAIL:  Key (id)=(null) already exists.
```

1.2.29 Intégrité référentielle

- **Contrainte d'intégrité référentielle**
  - ou **Clé étrangère**
- Référence une **clé primaire** ou un groupe de colonnes `UNIQUE` et `NOT NULL`
- Garantit l'intégrité des données
- `FOREIGN KEY`

Une clé étrangère sur une table fait référence à une clé primaire ou une contrainte d'unicité d'une autre table. La clé étrangère garantit que les valeurs des colonnes de cette clé existent également dans la table portant la clé primaire ou la contrainte d'unicité. On parle de **contrainte référentielle d'intégrité** : la contrainte interdit les valeurs qui n'existent pas dans la table référencée. Toute insertion ou modification qui viole cette règle est rejetée.

C'est ce mécanisme qui permet de garantir qu'une commande sera liée à un client existant dans la table des clients, ou que le pays dans une adresse existe bien dans la table des pays. Et à l'inverse, on

ne pourra supprimer un pays ou un client dans les tables des commandes ou adresses qui possèdent une contrainte d'intégrité dessus.

À titre d'exemple nous allons utiliser la base `cave`. La base **cave** (dump de 2,6 Mo, pour 71 Mo sur le disque au final) peut être téléchargée et restaurée ainsi :

```
curl -kL https://dali.bo/tp_cave -o cave.dump
psql -c "CREATE ROLE caviste LOGIN PASSWORD 'caviste'"
psql -c "CREATE DATABASE cave OWNER caviste"
pg_restore -d cave cave.dump
# NB : une erreur sur un schéma 'public' existant est normale
```

Le schéma suivant montre les différentes tables de la base :

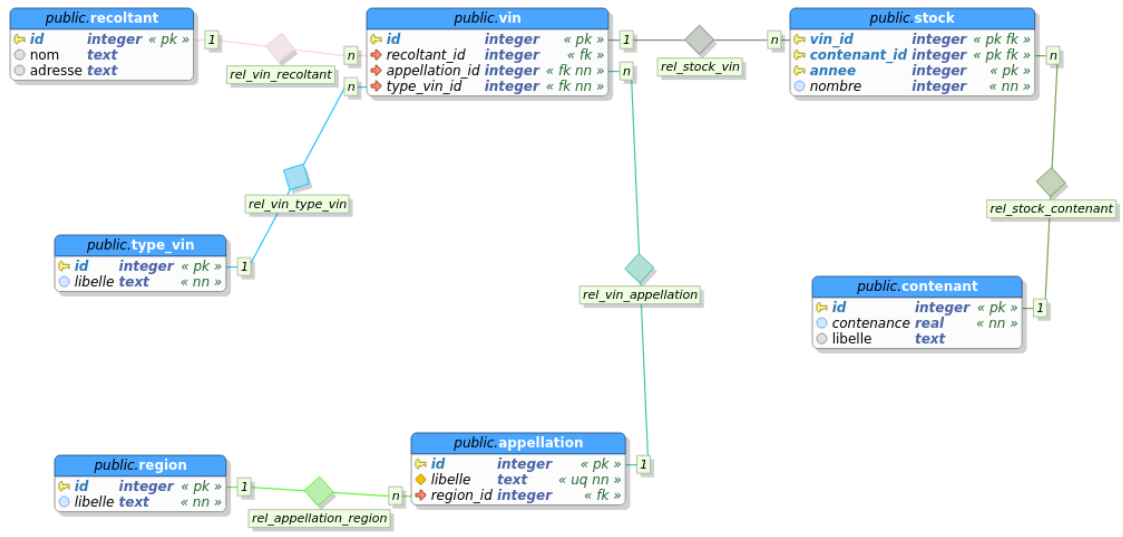


FIGURE 1/ .1 – Schéma base cave

Ainsi, la base cave définit une table `region` et une table `appellation`. Une appellation d'origine est liée au terroir, et par extension à son origine géographique. La table `appellation` est donc liée par une clé étrangère à la table `region` : la colonne `region_id` de la table `appellation` référence la colonne `id` de la table `region`.

Cette contrainte permet d'empêcher les utilisateurs d'entrer dans la table `appellation` des identifiants de région (`region_id`) qui n'existent pas dans la table `region`.

1.2.30 Déclaration d'une clé étrangère

```
[ CONSTRAINT nom_contrainte ] FOREIGN KEY ( nom_colonne [, …] )
    REFERENCES table_reference [ (colonne_reference [, … ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
    [ ON DELETE action ] [ ON UPDATE action ] }
```

Exemples :

Définition de la table `stock` :

```
CREATE TABLE stock
  (
    vin_id        int    not null,
    contenant_id  int    not null,
    annee         int4   not null,
    nombre        int4   not null,

    PRIMARY KEY(vin_id,contenant_id,annee),

    FOREIGN KEY(vin_id) REFERENCES vin(id) ON DELETE CASCADE,
    FOREIGN KEY(contenant_id) REFERENCES contenant(id) ON DELETE CASCADE
  );
```

Cette table possède une clé primaire composée : il ne peut exister qu'une seule ligne pour un trio vin, contenant, année précis.

Création d'une table « mère » et d'une table « fille ». La table fille possède une clé étrangère qui référence la table mère :

```
CREATE TABLE mere (id integer, t text);

CREATE TABLE fille (id integer, mere_id integer, t text);

ALTER TABLE mere ADD CONSTRAINT pk_mere PRIMARY KEY (id);

ALTER TABLE fille
  ADD CONSTRAINT fk_mere_fille
      FOREIGN KEY (mere_id)
      REFERENCES mere (id)
         MATCH FULL
         ON UPDATE NO ACTION
         ON DELETE CASCADE;
```

Ces trois dernières options sont optionnelles, et les valeurs par défaut sont différentes (voir plus bas).

```
INSERT INTO mere (id, t) VALUES (1, 'val1'), (2, 'val2');
```

L'ajout de données dans la table fille qui font bien référence à la table mere est accepté :

```
INSERT INTO fille (id, mere_id, t) VALUES (1, 1, 'val1');
INSERT INTO fille (id, mere_id, t) VALUES (2, 2, 'val2');
```

L'ajout de données dans la table fille qui ne font *pas* référence à la table mère est refusé :

```
INSERT INTO fille (id, mere_id, t) VALUES (3, 3, 'val3');

ERROR:  insert or update on table "fille" violates foreign key constraint
        "fk_mere_fille"
DETAIL:  Key (mere_id)=(3) is not present in table "mere".

SELECT * FROM fille;

 id | mere_id |  t
----+---------+------
  1 |       1 | val1
  2 |       2 | val2
```

Mettre à jour la référence dans la table mère ne fonctionnera pas car la contrainte a été définie pour refuser les mises à jour (`ON UPDATE NO ACTION`) :

```
UPDATE mere SET id=3 WHERE id=2;

ERROR:  update or delete on table "mere" violates foreign key constraint
        "fk_mere_fille" on table "fille"
DETAIL:  Key (id)=(2) is still referenced from table "fille".
```

Par contre, la suppression d'une ligne de la table mère référencée dans la table fille va propager la suppression jusqu'à la table fille (`ON DELETE CASCADE`) :

```
DELETE FROM mere WHERE id=2;

DELETE 1

SELECT * FROM fille;

 id | mere_id |  t
----+---------+------
  1 |       1 | val1

SELECT * FROM mere;

 id |  t
----+------
  1 | val1
```

1.2.31 Vérification simple ou complète

- Vérification complète ou partielle d'une clé étrangère
- `MATCH`
  - `MATCH FULL` (complète)
  - `MATCH SIMPLE` (partielle)

La directive `MATCH` permet d'indiquer si la contrainte doit être entièrement vérifiée (`MATCH FULL`) ou si la clé étrangère autorise des valeurs `NULL` (`MATCH SIMPLE`). `MATCH SIMPLE` est la valeur par défaut.

Avec `MATCH FULL`, toutes les valeurs des colonnes qui composent la clé étrangère de la table référençant doivent avoir une correspondance dans la table référencée.

Avec `MATCH SIMPLE`, les valeurs des colonnes qui composent la clé étrangère de la table référençant peuvent comporter des valeurs `NULL`. Dans le cas des clés étrangères multicolonnes, toutes les colonnes peuvent ne pas être renseignées. Dans le cas des clés étrangères sur une seule colonne, la contrainte autorise les valeurs `NULL`.

Exemples

Les exemples reprennent les tables `mere` et `fille` créées plus haut.

```
INSERT INTO fille VALUES (4, NULL, 'test');

SELECT * FROM fille;

 id | mere_id |  t
----+---------+------
  1 |       1 | val1
  2 |       2 | val2
  4 |         | test
```

1.2.32 Clé primaire et colonne identité

- Identité d'un enregistrement
- `GENERATED … AS IDENTITY`
  - `ALWAYS`
  - `BY DEFAULT`
- Préférer à `serial`
- Ne dispense pas d'ajouter la contrainte de clé primaire !

La syntaxe `GENERATED … AS IDENTITY` permet d'avoir une colonne dont la valeur est incrémentée automatiquement, soit en permanence (clause `ALWAYS`), soit quand aucune valeur n'est saisie (clause `BY DEFAULT`). `GENERATED … AS IDENTITY` est apparue dans PostgreSQL après le pseudo-type `serial`, d'utilisation similaire, mais il en corrige certains défauts.

Pour les raisons qui suivent, préférez l'utilisation de `GENERATED … AS IDENTITY` à celle de `serial`, lequel peut se rencontrer encore dans de nombreuses applications.

Tout d'abord, `GENERATED … AS IDENTITY` fait partie du standard SQL, alors que le pseudo-type `serial` est propre à PostgreSQL.

`serial` n'est en effet qu'un raccourci pour créer à la fois une séquence, un champ entier et un `DEFAULT nextval()` pour alimenter ce champ, et cela peut poser quelques soucis. Notamment, le DDL saisi par l'utilisateur diffère de celui stocké en base ou sorti par `pg_dump`, ce qui n'est pas idéal. Ou encore, l'ordre de copie de structure de tables `CREATE TABLE … LIKE (INCLUDING ALL)` copie le `serial` sans en changer la séquence ni en créer une autre : on a alors une séquence partagée par deux tables ! Il n'est pas non plus possible d'ajouter ou de supprimer un pseudo-type `serial` avec l'instruction `ALTER TABLE`. La suppression de la contrainte `DEFAULT` d'un type `serial` ne supprime pas la séquence associée. Tout ceci fait que la définition d'une colonne d'identité est préférable à l'utilisation du pseudo-type `serial`.

Il reste obligatoire de définir une clé primaire ou unique si l'on tient à l'unicité des valeurs, car même une clause `GENERATED ALWAYS AS IDENTITY` peut être contournée avec une mise à jour portant la mention `OVERRIDING SYSTEM VALUE`.

Exemple :

```
CREATE TABLE personnes (id int GENERATED ALWAYS AS IDENTITY, nom TEXT);
INSERT INTO personnes (nom) VALUES ('Dupont') ;
INSERT INTO personnes (nom) VALUES ('Durand') ;

SELECT * FROM personnes ;

 id |  nom
----+--------
  1 | Dupont
  2 | Durand

INSERT INTO personnes (id,nom) VALUES (3,'Martin') ;

ERROR:  cannot insert into column "id"
DÉTAIL : Column "id" is an identity column defined as GENERATED ALWAYS.
ASTUCE : Use OVERRIDING SYSTEM VALUE to override.

INSERT INTO personnes (id,nom) OVERRIDING SYSTEM VALUE VALUES (3,'Martin') ;
INSERT INTO personnes (id,nom) OVERRIDING SYSTEM VALUE VALUES (3,'Dupond') ;

SELECT * FROM personnes ;

 id |  nom
----+--------
  1 | Dupont
  2 | Durand
  3 | Martin
  3 | Dupond
```

1.2.33 Mise à jour de la clé primaire

- Que faire en cas de mise à jour d'une clé primaire ?
  - les clés étrangères seront fausses
- `ON UPDATE`
- `ON DELETE`
- Définition d'une action au niveau de la clé étrangère
  - interdiction
  - propagation de la mise à jour
  - NULL
  - valeur par défaut

Si des valeurs d'une clé primaire sont mises à jour ou supprimées, cela peut entraîner des incohérences dans la base de données si des valeurs de clés étrangères font référence aux valeurs de la clé primaire touchées par le changement.

Afin de pouvoir gérer cela, la norme SQL prévoit plusieurs comportements possibles. La clause `ON UPDATE` permet de définir comment le SGBD va réagir si la clé primaire référencée est mise à jour. La clause `ON DELETE` fait de même pour les suppressions.

Les actions possibles sont :

- `NO ACTION` (ou `RESTRICT`), qui produit une erreur si une ligne référence encore le ou les lignes touchées par le changement ;
- `CASCADE`, pour laquelle la mise à jour ou la suppression est propagée aux valeurs référençant le ou les lignes touchées par le changement ;
- `SET NULL`, la valeur de la colonne devient `NULL` ;
- `SET DEFAULT`, pour lequel la valeur de la colonne prend la valeur par défaut de la colonne.

Le comportement par défaut est `NO ACTION`, ce qui est habituellement recommandé pour éviter les suppressions en chaîne mal maîtrisées.

Exemples

Les exemples reprennent les tables `mere` et `fille` créées plus haut.

Tentative d'insertion d'une ligne dont la valeur de `mere_id` n'existe pas dans la table `mere` :

```
INSERT INTO fille (id, mere_id, t) VALUES (1, 3, 'val3');
```

```
ERROR:  insert or update on table "fille" violates foreign key constraint
        "fk_mere_fille"
DETAIL:  Key (mere_id)=(3) is not present in table "mere".
```

Mise à jour d'une ligne de la table `mere` pour modifier son `id`. La clé étrangère est déclarée `ON UPDATE NO ACTION`, donc la mise à jour devrait être interdite :

```
UPDATE mere SET id = 3 WHERE id = 1;

ERROR:  update or delete on table "mere" violates foreign key constraint
        "fk_mere_fille" on table "fille"
DETAIL:  Key (id)=(1) is still referenced from table "fille".
```

Suppression d'une ligne de la table `mere`. La clé étrangère sur `fille` est déclarée `ON DELETE CASCADE`, la suppression sera donc propagée aux tables qui référencent la table `mere` :

```
DELETE FROM mere WHERE id = 1;

SELECT * FROM fille ;

 id | mere_id |  t
----+---------+------
  2 |       2 | val2
```

1.2.34 Vérifications

- Présence d'une valeur
  - `NOT NULL`
  - NB : obligatoire dans une clé primaire
- Vérification de la valeur d'une colonne
  - `CHECK`

La clause `NOT NULL` permet de s'assurer que la valeur de la colonne portant cette contrainte est renseignée. Dit autrement, elle doit obligatoirement être renseignée. Par défaut, une colonne peut avoir une valeur `NULL`, donc n'est pas obligatoirement renseignée. Rappelons qu'une clé primaire ne peut pas être à `NULL`.

La clause `CHECK` spécifie une expression de résultat booléen que les nouvelles lignes ou celles mises à jour doivent satisfaire pour qu'une opération d'insertion ou de mise à jour réussisse. Les expressions de résultat `TRUE` ou `NULL` réussissent. Si une des lignes de l'opération d'insertion ou de mise à jour produit un résultat `FALSE`, une exception est levée et la base de données n'est pas modifiée. Une contrainte de vérification sur une colonne ne fait référence qu'à la valeur de la colonne tandis qu'une contrainte sur la table fait référence à plusieurs colonnes. Une fonction définie par l'utilisateur peut être utilisée à condition de respecter ces mêmes règles.

Actuellement, les expressions `CHECK` ne peuvent ni contenir des sous-requêtes ni faire référence à des variables autres que les colonnes de la ligne courante. C'est techniquement réalisable, mais non supporté.

```
CREATE TABLE produits (
    no_produit integer,
    nom text,
    prix numeric CHECK (prix > 0),
    prix_promotion numeric,
```

```
    CONSTRAINT promo_valide CHECK (prix_promotion > 0 AND prix > prix_promotion)
);
```

Cet exemple est inspiré de la documentation officielle, page Contraintes de vérification[1].

1.2.35 Vérifications différés

- Vérifications après chaque ordre SQL
  - problèmes de cohérence
- Différer les vérifications de contraintes
  - clause `DEFERRABLE`, `NOT DEFERRABLE`
  - `INITIALLY DEFERED`, `INITIALLY IMMEDIATE`

Par défaut, toutes les contraintes d'intégrité sont vérifiées lors de l'exécution de chaque ordre SQL de modification, y compris dans une transaction. Cela peut poser des problèmes de cohérences de données : insérer dans une table fille alors qu'on n'a pas encore inséré les données dans la table mère, la clé étrangère de la table fille va rejeter l'insertion et annuler la transaction.

Le moment où les contraintes sont vérifiées est modifiable dynamiquement par l'ordre `SET CONSTRAINTS` :

```
SET CONSTRAINTS { ALL | nom [, …] } { DEFERRED | IMMEDIATE }
```

mais ce n'est utilisable que pour les contraintes déclarées comme déferrables.

Voici quelques exemples :

- avec la définition précédente des tables `mere` et `fille` :

```
BEGIN;
UPDATE mere SET id=3 where id=1;

ERROR:  update or delete on table "mere" violates foreign key constraint
        "fk_mere_fille" on table "fille"
DETAIL:  Key (id)=(1) is still referenced from table "fille".
```

- cette erreur survient aussi dans le cas où on demande que la vérification des contraintes soit différée pour cette transaction :

```
BEGIN;
SET CONSTRAINTS ALL DEFERRED;
UPDATE mere SET id=3 WHERE id=1;

ERROR:  update or delete on table "mere" violates foreign key constraint
        "fk_mere_fille" on table "fille"
DETAIL:  Key (id)=(1) is still referenced from table "fille".
```

- il faut que la contrainte soit déclarée comme étant différable :

[1] https://docs.postgresql.fr/current/ddl-constraints.html#DDL-CONSTRAINTS-CHECK-CONSTRAINTS

```
CREATE TABLE mere (id integer, t text);
CREATE TABLE fille (id integer, mere_id integer, t text);
ALTER TABLE mere ADD CONSTRAINT pk_mere PRIMARY KEY (id);
ALTER TABLE fille
  ADD CONSTRAINT fk_mere_fille
      FOREIGN KEY (mere_id)
      REFERENCES mere (id)
         MATCH FULL
         ON UPDATE NO ACTION
         ON DELETE CASCADE
         DEFERRABLE;
INSERT INTO mere (id, t) VALUES (1, 'val1'), (2, 'val2');
INSERT INTO fille (id, mere_id, t) VALUES (1, 1, 'val1');
INSERT INTO fille (id, mere_id, t) VALUES (2, 2, 'val2');

BEGIN;
SET CONSTRAINTS all deferred;
UPDATE mere SET id=3 WHERE id=1;

SELECT * FROM mere;

 id |  t
----+------
  2 | val2
  3 | val1

SELECT * FROM fille;

 id | mere_id |  t
----+---------+------
  1 |       1 | val1
  2 |       2 | val2

UPDATE fille SET mere_id=3 WHERE mere_id=1;

COMMIT;
```

1.2.36 Vérifications plus complexes

- Un trigger
  - si une contrainte porte sur plusieurs tables
  - si sa vérification nécessite une sous-requête
- Préférer les contraintes déclaratives

Les contraintes d'intégrités du SGBD ne permettent pas d'exprimer une contrainte qui porte sur plusieurs tables ou simplement si sa vérification nécessite une sous-requête. Dans ce cas là, il est nécessaire d'écrire un trigger spécifique qui sera déclenché après chaque modification pour valider la contrainte.

Il ne faut toutefois pas systématiser l'utilisation de triggers pour valider des contraintes d'intégrité. Cela aurait un fort impact sur les performances et sur la maintenabilité de la base de données. Il vaut mieux privilégier les contraintes déclaratives et les colonnes générées, et n'utiliser les triggers qu'en dernier recours.

1.2.37 Colonnes par défaut et générées

```
CREATE TABLE paquet (
  code       text          PRIMARY KEY,
  reception  timestamptz   DEFAULT now(),
  livraison  timestamptz   DEFAULT now() + interval '3d',
  largeur    int,    longueur int,   profondeur int,
  volume     int
      GENERATED ALWAYS AS ( largeur * longueur * profondeur )
      STORED    CHECK (volume > 0.0)
  ) ;
```

- `DEFAULT` : expressions très simples, modifiables
- `GENERATED`
  - fonctions « immutables », ne dépendant **que** de la ligne
  - (avant v17) difficilement modifiables
  - peuvent porter des contraintes

Les champs `DEFAULT` sont très utilisés, mais PostgreSQL supporte les colonnes générées (ou calculées).

1.2.37.1 DEFAULT ...

Les champs avec une clause `DEFAULT` sont remplis automatiquement à la création de la ligne quand une valeur n'est pas fournie dans l'ordre `INSERT` (celui-ci peut ne renseigner que certains champs).

Seules sont autorisées avec `DEFAULT` des expressions simples, sans variable, ni utilisation de champs de la ligne, ni sous-requête. Sont acceptées des constantes, certains calculs ou fonctions simples, comme `now()`, ou un appel à `nextval ('nom_séquence')`.

Ajouter une clause `DEFAULT` sur un champ existant calcule les valeurs pour toutes les lignes pré-existantes de la table, ce qui peut entraîner la réécriture de la table ! Une optimisation évite de tout réécrire si les anciennes lignes se retrouvent toutes avec la même valeur constante.

La valeur par défaut peut être écrasée, par déclaration explicite du champ lors de l'`INSERT`, ou plus tard avec `UPDATE`.

Changer l'expression d'une clause `DEFAULT` est possible :

```
ALTER TABLE paquet
ALTER COLUMN livraison SET DEFAULT now()+interval '4d';
```

mais cela n'a pas d'impact sur les lignes existantes, juste les nouvelles.

1.2.37.2 GENERATED ALWAYS AS (...) STORED

La syntaxe est :

```
nomchamp  <type> GENERATED ALWAYS AS ( <expression> ) STORED ;
```

Les colonnes générées sont recalculées à chaque fois que les champs sur lesquels elles sont basées changent, donc aussi lors d'un `UPDATE`. Ces champs calculés sont impérativement marqués `ALWAYS`, c'est-à-dire obligatoires et non modifiables, et `STORED`, c'est-à-dire stockés sur le disque (et non re-calculés à la volée comme dans une vue). Ils ne doivent pas se baser sur d'autres champs calculés.

Un intérêt est que les champs calculés peuvent porter des contraintes, par exemple la clause `CHECK` ci-dessous, mais encore des clés étrangères ou unique.

Exemple :

```
CREATE TABLE paquet (
  code       text          PRIMARY KEY,
  reception  timestamptz   DEFAULT now(),
  livraison  timestamptz   DEFAULT now() + interval '3d',
  largeur    int,    longueur int,   profondeur int,
  volume     int
      GENERATED ALWAYS AS ( largeur * longueur * profondeur )
      STORED    CHECK (volume > 0.0)
  ) ;

INSERT INTO paquet (code, largeur, longueur, profondeur)
VALUES ('ZZ1', 3, 5, 10) ;

\x on

TABLE paquet ;

-[ RECORD 1 ]----------------------------
code       | ZZ1
reception  | 2024-04-19 18:02:41.021444+02
livraison  | 2024-04-22 18:02:41.021444+02
largeur    | 3
longueur   | 5
profondeur | 10
volume     | 150

-- Les champs DEFAULT sont modifiables
-- Changer la largeur va modifier le volume
```

```
UPDATE paquet
SET largeur=4,
    livraison = '2024-07-14'::timestamptz,
    reception = '2024-04-20'::timestamptz
WHERE code='ZZ1' ;

TABLE paquet ;

-[ RECORD 1 ]----------------------
code       | ZZ1
reception  | 2024-04-20 00:00:00+02
livraison  | 2024-07-14 00:00:00+02
largeur    | 4
longueur   | 5
profondeur | 10
volume     | 200

-- Le volume ne peut être modifié
UPDATE paquet
SET volume = 250
WHERE code = 'ZZ1' ;

ERROR:  column "volume" can only be updated to DEFAULT
DETAIL : Column "volume" is a generated column.
```

Expression immutable :

Avec `GENERATED`, l'expression du calcul doit être « immutable », c'est-à-dire ne dépendre **que** des autres champs de la même ligne, n'utiliser que des fonctions elles-mêmes immutables, et rien d'autre. Il n'est donc pas possible d'utiliser des fonctions comme `now()`, ni des fonctions de conversion de date dépendant du fuseau horaire, ou du paramètre de formatage de la session en cours (toutes choses autorisées avec `DEFAULT`), ni des appels à d'autres lignes ou tables…

La colonne calculée peut être convertie en colonne « normale » :

```
ALTER TABLE paquet ALTER COLUMN volume DROP EXPRESSION ;
```

Modifier l'expression n'est pas possible avant PostgreSQL 17, sauf à supprimer la colonne générée et en créer une nouvelle. Il faut alors recalculer toutes les lignes et réécrire toute la table, ce qui peut être très lourd.

À partir de PostgreSQL 17, l'expression est modifiable avec cette syntaxe :

```
ALTER TABLE paquet ALTER COLUMN volume
SET EXPRESSION AS  ( largeur * longueur * profondeur + 1 ) ;
```

Attention, la table est totalement bloquée le temps de la réécriture (verrou `AccessExclusiveLock`).

Utilisation d'une fonction :

Il est possible de créer sa propre fonction pour l'expression, qui doit aussi être immutable :

```
CREATE OR REPLACE FUNCTION volume (l int, h int, p int)
RETURNS int
AS $$
 SELECT l * h * p ;
$$
```

```
LANGUAGE sql
-- cette fonction dépend uniquement des données de la ligne donc :
PARALLEL SAFE
IMMUTABLE ;

-- Changement à partir de PostgreSQL v17
ALTER TABLE paquet ALTER COLUMN volume
SET EXPRESSION AS  ( volume (largeur, longueur, profondeur) );

-- Changement avant PostgreSQL 16
ALTER TABLE paquet DROP COLUMN volume ;
ALTER TABLE paquet ADD COLUMN volume int
  GENERATED ALWAYS AS ( volume (largeur, longueur, profondeur) )
  STORED;

TABLE paquet ;

-[ RECORD 1 ]----------------------
code       | ZZ1
reception  | 2024-04-20 00:00:00+02
livraison  | 2024-07-14 00:00:00+02
largeur    | 4
longueur   | 5
profondeur | 10
volume     | 200
```

Attention : modifier la fonction ne réécrit pas spontanément la table, il faut forcer la réécriture avec par exemple :

```
UPDATE paquet SET longueur = longueur ;
```

et ceci dans la même transaction que la modification de fonction. On pourrait imaginer de négliger cet `UPDATE` pour garder les valeurs déjà présentes qui suivraient d'anciennes règles… mais ce serait une erreur. En effet, les valeurs calculées ne figurent pas dans une sauvegarde logique, et en cas de restauration, tous les champs sont recalculés avec la dernière formule !

On préférera donc gérer l'expression dans la définition de la table dans les cas simples.

Un autre piège : il faut résister à la tentation de déclarer une fonction comme immutable sans la certitude qu'elle l'est bien (penser aux paramètres de session, aux fuseaux horaires…), sous peine d'incohérences dans les données.

Cas d'usage :

Les colonnes générées économisent la création de triggers, ou de vues de « présentation ». Elles facilitent la dénormalisation de données calculées dans une même table tout en garantissant l'intégrité.

Un cas d'usage courant est la dénormalisation d'attributs JSON pour les manipuler comme des champs de table classiques :

```
ALTER TABLE personnes
ADD COLUMN  lastname text
GENERATED ALWAYS AS ((datas->>'lastName')) STORED ;
```

L'accès au champ est notablement plus rapide que l'analyse systématique du champ JSON.

Par contre, les colonnes `GENERATED` ne sont **pas** un bon moyen pour créer des champs portant la dernière mise à jour. Certes, PostgreSQL ne vous empêchera pas de déclarer une fonction (abusivement) immutable utilisant `now()` ou une variante. Mais ces informations seront perdues en cas de restauration logique. Dans ce cas, les triggers restent une option plus complexe mais plus propre.

1.3 DML : MISE À JOUR DES DONNÉES

- `SELECT` peut lire les données d'une table ou plusieurs tables
  - mais ne peut pas les mettre à jour (sauf fonction)
- Ajout de données dans une table
  - `INSERT`
- Modification des données d'une table
  - `UPDATE`
- L'un ou l'autre
  - `MERGE`
- Suppression des données d'une table
  - `DELETE`

L'ordre `SELECT` permet de lire une ou plusieurs tables, sans modification. Il faut toutefois savoir que `SELECT` peut aussi servir à appeler des fonctions, et que celles-ci sont susceptibles de faire à peu près n'importe quoi dans la base.

Les mises à jours utilisent des ordres distincts. L'ordre `INSERT` permet d'ajouter ou insérer des données dans une table. L'ordre `UPDATE` permet de modifier des lignes déjà existantes.

Depuis PostgreSQL 15, il existe un ordre `MERGE` (conforme au standard SQL) permettant par exemple de mettre à jour une ligne dont la clé existe, ou de l'insérer au besoin. L'utilisation peut être complexe[2].

Enfin, l'ordre `DELETE` permet de supprimer des lignes.

Ces derniers ordres ne peuvent modifier qu'une seule table à la fois. Si on souhaite par exemple insérer des données dans deux tables, il est nécessaire de réaliser deux ordres `INSERT` distincts.

Noter qu'il est possible de regrouper plusieurs `SELECT` / `DELETE` / `INSERT` /etc. dans une même requête, grâce aux *Common Table Expressions*[3], que nous n'aborderons pas ici.

[2] https://docs.postgresql.fr/current/sql-merge.html
[3] https://docs.postgresql.fr/current/queries-with.html

1.3.1 Ajout de données : INSERT

— Ajoute des lignes à partir des données de la requête
— Ajoute des lignes à partir d'une requête `SELECT`
— Syntaxe :

```
INSERT INTO nom_table [ ( nom_colonne [, …] ) ]
    { liste_valeurs | requete }
```

— Précisez toujours les champs à renseigner

L'ordre `INSERT` insère de nouvelles lignes dans une table. Il permet d'insérer une ou plusieurs lignes spécifiées par les expressions de valeur, ou zéro ou plusieurs lignes provenant d'une requête.

La liste des noms des colonnes est optionnelle. L'ordre des noms des colonnes dans la liste n'a pas d'importance particulière, il suffit de nommer les colonnes mises à jour.

Si la liste n'est pas spécifiée, alors PostgreSQL utilisera implicitement la liste de toutes les colonnes de la table dans l'ordre de leur déclaration, ou les `N` premiers noms de colonnes si seules `N` valeurs de colonnes sont fournies dans la clause `VALUES` ou dans la requête.

Cette dernière pratique est à éviter car elle rend votre code sensible aux modifications de structures : ajout de champs, modification de l'ordre des champs… Cela arrive plus facilement qu'on ne le croit.
Prenez l'habitude de nommer les champs à renseigner.

Chaque colonne absente de la liste, implicite ou explicite, se voit attribuer sa valeur par défaut, s'il y en a une ou `NULL` dans le cas contraire. Les expressions de colonnes qui ne correspondent pas au type de données déclarées sont transtypées automatiquement dans la mesure du possible. Si vous avez défini des triggers avant insertion, ils se déclencheront aussi.

Dans l'exemple suivant, seules deux colonnes sur cinq sont renseignées à l'insertion :

```
CREATE TABLE demoins (
    id    int GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    zero  int DEFAULT 0,
    x     float,
    y     float GENERATED ALWAYS AS (x/2.0) STORED,
    z     timestamp with time zone) ;

INSERT INTO demoins (x, z)
SELECT 3.14159, '2025-07-01' ;

TABLE demoins ;
```

```
 id | zero |    x    |    y     |           z
----+------+---------+----------+------------------------
  1 |    0 | 3.14159 | 1.570795 | 2025-07-01 00:00:00+02
```

1.3.2 INSERT avec liste d'expressions

```
INSERT INTO nom_table [ ( nom_colonne [, …] ) ]
VALUES ( { expression | DEFAULT } [, …] ) [, …]
```

La clause `VALUES` permet de définir une liste d'expressions qui va constituer la ligne à insérer dans la base de données. Les éléments de cette liste d'expression sont séparés par une virgule. Cette liste d'expression est composée de constantes ou d'appels à des fonctions retournant une valeur, pour obtenir par exemple la date courante ou la prochaine valeur d'une séquence. Les valeurs fournies par la clause `VALUES` ou par la requête sont associées à la liste explicite ou implicite des colonnes de gauche à droite.

Exemples

Insertion d'une ligne dans la table `stock` :

```
INSERT INTO stock (vin_id, contenant_id, annee, nombre)
VALUES (12, 1, 1935, 1);
```

Insertion d'une ligne dans la table `vin` :

```
INSERT INTO vin (id, recoltant_id, appellation_id, type_vin_id)
VALUES (nextval('vin_id_seq'), 3, 6, 1);
```

1.3.3 INSERT à partir d'un SELECT

```
INSERT INTO nom_table [ ( nom_colonne [, …] ) ]
    …requête…
```

L'ordre `INSERT` peut aussi prendre une requête SQL en entrée. Dans ce cas, `INSERT` va insérer autant de lignes dans la table d'arrivée qu'il y a de lignes retournées par la requête `SELECT`. L'ordre des colonnes retournées par `SELECT` doit correspondre exactement à l'ordre précisé dans l'ordre `INSERT`. Leur type de données doit également correspondre.

Exemple :

Insertion dans une table `stock2` à partir d'une requête `SELECT` sur la table `stock1` :

```
INSERT INTO stock2 (vin_id, contenant_id, annee, nombre)
SELECT vin_id, contenant_id, annee, nombre FROM stock;
```

1.3.4 Mise à jour de données : UPDATE

- Met à jour une ou plusieurs colonnes d'une même ligne
  - à partir des valeurs de la requête
  - à partir des anciennes valeurs
  - à partir d'une requête `SELECT`
  - à partir de valeurs d'une autre table

L'ordre de mise à jour de lignes s'appelle `UPDATE`.

1.3.5 Construction d'UPDATE

```
UPDATE nom_table
    SET
    {
     nom_colonne = { expression | DEFAULT }
    |
     ( nom_colonne [, …] ) = ( { expression | DEFAULT } [, …] )
    } [, …]
    [ FROM liste_from ]
    [ WHERE condition | WHERE CURRENT OF nom_curseur ]
```

L'ordre `UPDATE` permet de mettre à jour les lignes d'une table.

L'ordre `UPDATE` ne met à jour que les lignes qui satisfont les conditions de la clause `WHERE`. La clause `SET` permet de définir les colonnes à mettre à jour. Le nom des colonnes mises à jour doivent faire partie de la table mise à jour.

Les valeurs mises à jour peuvent faire référence aux valeurs avant mise à jour de la colonne, dans ce cas on utilise la forme `nom_colonne = expression`. La partie de gauche référence la colonne à mettre à jour, la partie de droite est une expression qui permet de déterminer la valeur à appliquer à la colonne. La valeur à appliquer peut bien entendu être une référence à une ou plusieurs colonnes et elles peuvent être dérivées par une opération arithmétique.

La clause `FROM` ne fait pas partie de la norme SQL mais certains SGBDR la supportent, dont PostgreSQL. Elle permet de réaliser facilement la mise à jour d'une table à partir des valeurs d'une ou plusieurs tables annexes. La norme SQL permet aussi de réaliser des mises à jour en utilisant une sous-requête, sans clause `FROM`.

Exemples :

Mise à jour du prix d'un livre particulier :

```
UPDATE livres SET prix = 10
WHERE isbn = '978-3-8365-3872-5';
```

Augmentation de 5 % du prix des livres :

```
UPDATE livres
SET prix = prix * 1.05;
```

Mise à jour d'une table `employees` à partir des données d'une table `bonus_plan` :

```
UPDATE employees e
   SET commission_rate = bp.commission_rate
  FROM bonus_plan bp
    ON (e.bonus_plan = bp.planid)
```

La même requête avec une sous-requête, conforme à la norme SQL :

```
UPDATE employees
   SET commission_rate = (SELECT commission_rate
                            FROM bonus_plan bp
                           WHERE bp.planid = employees.bonus_plan);
```

Lorsque plusieurs colonnes doivent être mises à jour à partir d'une jointure, il est possible d'utiliser ces deux écritures :

```
UPDATE employees e
   SET commission_rate = bp.commission_rate,
       commission_rate2 = bp.commission_rate2
  FROM bonus_plan bp
    ON (e.bonus_plan = bp.planid);
```

et :

```
UPDATE employees e
   SET (commission_rate, commission_rate2) = (
      SELECT bp.commission_rate, bp.commission_rate2
      FROM bonus_plan bp ON (e.bonus_plan = bp.planid)
   );
```

1.3.6 Suppression de données : DELETE

— Supprime les lignes répondant au prédicat
— Syntaxe :

```
DELETE FROM nom_table [ [ AS ] alias ]
    [ WHERE condition
```

L'ordre `DELETE` supprime l'ensemble des lignes qui répondent au prédicat de la clause `WHERE`.

```
DELETE FROM nom_table [ [ AS ] alias ]
    [ WHERE condition | WHERE CURRENT OF nom_curseur ]
```

Exemples :

Suppression d'un livre épuisé du catalogue :

```
DELETE FROM livres
WHERE isbn = '978-0-8707-0635-6';
```

Suppression de tous les livres :

```
DELETE FROM livres ;
```

N'oubliez jamais de préciser la clause `WHERE` !

Pour des raisons techniques, il faut savoir qu'il est beaucoup plus rapide de vider une table avec l'ordre `TRUNCATE TABLE livres ;` qu'avec un `DELETE` sans clause `WHERE`.

1.3.7 Clause RETURNING

- Spécifique à PostgreSQL
- Permet de retourner les lignes complètes ou partielles résultants de `INSERT`, `UPDATE` ou `DELETE`
- Syntaxe :

```
requete_sql RETURNING ( * | expression )
```

La clause `RETURNING` est une syntaxe propre à PostgreSQL et très pratique. Elle permet de retourner les lignes insérées, mises à jour ou supprimées par un ordre DML de modification. Il est également possible de dériver une valeur retournée.

L'emploi de la clause `RETURNING` peut nécessiter des droits complémentaires sur les objets de la base.

Exemple :

Mise à jour du nombre de bouteilles en stock :

```
SELECT annee, nombre FROM stock
WHERE vin_id = 7 AND contenant_id = 1 AND annee = 1967;
```

```
 annee | nombre
-------+--------
  1967 |     17
```

```
UPDATE stock SET nombre = nombre - 1
WHERE vin_id = 7 AND contenant_id = 1 AND annee = 1967
RETURNING nombre;
```

```
 nombre
--------
     16
```

1.4 TRANSACTIONS

- ACID
  - **A**tomicité
  - un traitement se fait en entier…
  - …ou pas du tout
- TCL pour *Transaction Control Language*
  - ouvrir une transaction : `BEGIN`
  - valide une transaction : `COMMIT`
  - annule une transaction : `ROLLBACK`
  - points de sauvegarde : `SAVEPOINT`

Les transactions sont une partie essentielle du langage SQL. Elles permettent de rendre atomiques un certain nombre de requêtes. Le résultat de toutes les requêtes regroupées dans une transaction est validé ou pas, mais on ne peut pas enregistrer d'état intermédiaire.

Un client qui ouvre une transaction et déroule des ordres, est isolé des autres sessions, dans le sens où les autres sessions *ne voient pas* ses modifications avant la validation par `COMMIT`.

Pour les autres sessions, même appartenant au même utilisateur, les données sont soit telles qu'elles étaient *avant* la transaction, soit telles qu'elles sont devenues *après* validation.

L'utilisateur qui fait les modifications peut voir les éventuels états intermédiaires d'une transaction entre deux ordres (il peut faire des `SELECT` sur ce qu'il vient de modifier), mais uniquement depuis la session de la transaction.

Il est impossible de voir les étapes intermédiaires d'un ordre SQL, car une session exécute séquentiellement les ordres, et attend la fin de chaque ordre avant de poursuivre.

Par défaut, une transaction en cours voit les modifications validées dans d'autres sessions parallèles. Ce « niveau d'isolation » peut se changer pour que la transaction ne voit pas les modifications d'autres transaction, comme si la base était figée pendant sa durée.

Le langage SQL définit qu'une transaction peut être ouverte avec `BEGIN`, et à la fin être validée ou annulée. Ce sont respectivement les ordres `COMMIT` et `ROLLBACK`. Il est aussi possible de faire des points de reprise ou de sauvegarde dans une transaction. Ils se font en utilisant l'ordre `SAVEPOINT`.

1.4.1 Auto-commit et transactions

- Par défaut, PostgreSQL fonctionne en autocommit
  - à moins d'ouvrir explicitement une transaction
- Ouvrir une transaction
  - `BEGIN ;`

Une transaction débute toujours par un `BEGIN ;`. (`BEGIN TRANSACTION` et `START TRANSACTION` sont des synonymes, mais se rencontrent très peu.)

PostgreSQL fonctionne en autocommit. Autrement dit, sans `BEGIN`, une requête est considérée comme une transaction complète et n'a donc pas besoin de `COMMIT`. Celui-ci est implicite.

Pour l'utilisateur, cela est en fait géré par l'outil client, qui ajoute des `BEGIN` invisibles si on le configure avec autocommit à `off`. La transaction ne finira qu'avec un `COMMIT`, un `ROLLBACK` ou la déconnexion (le `ROLLBACK` est alors implicite).

1.4.2 Validation ou annulation d'une transaction

- Valider une transaction
  - `COMMIT ;`
- Annuler une transaction
  - `ROLLBACK ;`
- Sans validation, une transaction est forcément annulée
- Quasi-instantané sous PostgreSQL

Une transaction est toujours terminée par un `COMMIT ;` quand on veut que les modifications soient définitivement enregistrées. `END ;` existe aussi mais est rarissime et moins clair.

`ROLLBACK` quitte la transaction en annulant *toutes* les modifications qui y ont été faites, donc en revenant à l'état précédant la transaction. D'autres transactions ont pu faire d'autres actions pendant ce temps, elles ne seront pas annulées.

Ce retour à l'état initial doit se comprendre d'un point de vue logique et pas physique. Les modifications que vous avez pu effectuer dans une transaction terminée par un `ROLLBACK` ont quand même une incidence sur le stockage de vos données. Il peut y avoir eu de nombreux changements qui sont éventuellement à nettoyer plus tard par PostgreSQL, et peuvent avoir un impact sur les performances.
N'abusez donc pas des `ROLLBACK` si vous pouvez l'éviter.

Contrairement à certains produits concurrents, PostgreSQL exécute presque instantanément les ordres `COMMIT` et `ROLLBACK`.
En simplifiant, PostgreSQL se limite à noter que la transaction est valide ou pas, et que dans le futur les modifications concernées dans les fichiers de données doivent être prises en compte ou ignorées.

Si une session se termine, quelle que soit la raison, la transaction en cours sans `COMMIT` et sans `ROLLBACK` est considérée comme annulée.

Exemple :

Avant de retirer une bouteille du stock, on vérifie tout d'abord qu'il reste suffisamment de bouteilles en stock :

```
BEGIN TRANSACTION;

SELECT annee, nombre FROM stock WHERE vin_id = 7 AND contenant_id = 1
AND annee = 1967;

 annee | nombre
-------+--------
  1967 |     17

UPDATE stock SET nombre = nombre - 1
WHERE vin_id = 7 AND contenant_id = 1 AND annee = 1967 RETURNING nombre;

 nombre
--------
     16

COMMIT;
```

1.4.3 Programmation

— Certains langages implémentent des méthodes de gestion des transactions
  — PHP, Java, etc.
— Utiliser ces méthodes prioritairement

La plupart des langages permettent de gérer les transactions à l'aide de méthodes ou fonctions particulières. Il est recommandé de les utiliser.

En Java, ouvrir une transaction revient à désactiver l'autocommit :

```
String url =
    "jdbc:postgresql://localhost/test?user=fred&password=secret&ssl=true";
Connection conn = DriverManager.getConnection(url);
conn.setAutoCommit(false);
```

La transaction est confirmée (`COMMIT`) avec la méthode suivante :

```
conn.commit();
```

À l'inverse, elle est annulée (`ROLLBACK`) avec la méthode suivante :

```
conn.rollback();
```

1.4.4 Points de sauvegarde

- Certains traitements dans une transaction peuvent être annulés
  - mais la transaction est atomique
- Définir un point de sauvegarde
  - `SAVEPOINT nom_savepoint ;`
- Valider le traitement depuis le dernier point de sauvegarde
  - `RELEASE SAVEPOINT nom_savepoint ;`
- Annuler le traitement depuis le dernier point de sauvegarde
  - `ROLLBACK TO SAVEPOINT nom_savepoint ;`
  - et on continue dans la même transaction

Au sein d'une transaction, les points de sauvegarde (ordre `SAVEPOINT nom ;`) permettent d'encadrer un traitement sur lequel on peut vouloir revenir sans quitter la transaction (par exemple, selon le retour d'un test), ou au cas où un ordre tombe en erreur.

Revenir à l'état du point de sauvegarde se fait avec `ROLLBACK TO SAVEPOINT nom ;` pour annuler uniquement la partie du traitement voulue, ou pour revenir à l'état d'avant l'erreur, sans annuler le début de la transaction. On peut ensuite continuer.

Les *savepoints* n'existent qu'au sein de la transaction ; on ne peut bien sûr revenir sur une validation par `COMMIT` qu'une fois celui-ci terminé.

L'ordre `RELEASE SAVEPOINT nom ;` permet de « libérer » (oublier) un *savepoint* précédent ainsi que ceux posés *après*. C'est surtout utile pour libérer quelques ressources dans des transactions complexes.

Les points de sauvegarde sont des éléments nommés, il convient donc de leur affecter un nom particulier. Leur nom doit être unique dans la transaction courante.

N'abusez pas des `SAVEPOINT` : ils complexifient le code, et trop de `ROLLBACK` partiels peuvent entraîner des soucis de performance. Les `SAVEPOINT` sont rarement nécessaires quand on crée des transactions vraiment atomiques en « tout ou rien ». Réservez-les aux transactions vraiment complexes.

Exemple :

Transaction avec un point de sauvegarde et la gestion de l'erreur :

```
BEGIN;

INSERT INTO mere (id, val_mere) VALUES (10, 'essai');

SAVEPOINT insert_fille;

INSERT INTO fille (id_fille, id_mere, val_fille) VALUES (1, 10, 'essai 2');

ERROR:  duplicate key value violates unique constraint "fille_pkey"
DETAIL:  Key (id_fille)=(1) already exists.

ROLLBACK TO SAVEPOINT insert_fille;

COMMIT;

SELECT * FROM mere;

 id | val_mere
----+----------
  1 | mere 1
  2 | mere 2
 10 | essai
```

1.5 CONCLUSION

- SQL : toujours un traitement d'ensembles d'enregistrements
  - c'est le côté relationnel
- Pour les définitions d'objets
  - `CREATE`, `ALTER`, `DROP`
- Pour les données
  - `INSERT`, `UPDATE`, `DELETE`

Le standard SQL permet de traiter des ensembles d'enregistrements, que ce soit en lecture, en insertion, en modification et en suppression. Les ensembles d'enregistrements sont généralement des tables qui, comme tous les autres objets, sont créées (`CREATE`), modifiées (`ALTER`) et/ou supprimées (`DROP`).

1.5.1 Questions

```
BEGIN ;

WITH q AS (
  SELECT * FROM questions
  )
INSERT INTO reponses
SELECT * FROM q
WHERE texte_reponse IS NOT NULL ;

COMMIT ;
```

1.6 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/s30_solutions.

Cet exercice utilise la base **tpc**. La base **tpc** (dump de 31 Mo, pour 267 Mo sur le disque au final) et ses utilisateurs peuvent être installés comme suit :

```
curl -kL https://dali.bo/tp_tpc -o /tmp/tpc.dump
curl -kL https://dali.bo/tp_tpc_roles -o /tmp/tpc_roles.sql
# Exécuter le script de création des rôles
psql < /tmp/tpc_roles.sql
# Création de la base
createdb --owner tpc_owner tpc
# L'erreur sur un schéma 'public' existant est normale
pg_restore -d tpc /tmp/tpc.dump
```

Les mots de passe sont dans le script `/tmp/tpc_roles.sql`. Pour vous connecter :

```
$ psql -U tpc_admin -h localhost -d tpc
```

Pour cet exercice, les modifications de schéma doivent être effectuées par un rôle ayant suffisamment de droits pour modifier son schéma. Le rôle **tpc_admin** a les droits suffisants.

> Ajouter une colonne `email` de type `text` à la table `contacts`. Cette colonne va permettre de stocker l'adresse e-mail des clients et des fournisseurs. Ajouter également un commentaire décrivant cette colonne dans le catalogue de PostgreSQL (utiliser la commande `COMMENT`).

> Mettre à jour la table des contacts pour indiquer l'adresse e-mail de *Client6657* qui est `client6657@dalibo.com`.

> Ajouter une contrainte d'intégrité qui valide que la valeur de la colonne `email` créée est bien formée (vérifier que la chaîne de caractère contient au moins le caractère `@`).

> Valider la contrainte dans une transaction de test.

> Déterminer quels sont les contacts qui disposent d'une adresse e-mail et affichez leur nom ainsi que le code de leur pays.

> La génération des numéros de commande est actuellement réalisée à l'aide de la séquence `commandes_commande_id_seq`. Cette méthode ne permet pas de garantir que tous les numéros de commande se suivent. Proposer une solution pour sérialiser la génération des numéros de commande. Autrement dit, proposer une méthode pour obtenir un numéro de commande sans avoir de « trou » dans la séquence en cas d'échec d'une transaction.

> Noter le nombre de lignes de la table `pieces`. Dans une transaction, majorer de 5% le prix des pièces de moins de 1500 € et minorer de 5 % le prix des pièces dont le prix actuel est égal ou supérieur à 1500 €. Vérifier que le nombre de lignes mises à jour au total correspond au nombre total de lignes de la table `pieces`.

> Dans une même transaction, créer un nouveau client en incluant l'ajout de l'ensemble des informations requises pour pouvoir le contacter. Un nouveau client a un solde égal à 0.

1.7 TRAVAUX PRATIQUES (SOLUTIONS)

Cet exercice utilise la base **tpc**. La base **tpc** (dump de 31 Mo, pour 267 Mo sur le disque au final) et ses utilisateurs peuvent être installés comme suit :

```
curl -kL https://dali.bo/tp_tpc -o /tmp/tpc.dump
curl -kL https://dali.bo/tp_tpc_roles -o /tmp/tpc_roles.sql
# Exécuter le script de création des rôles
psql < /tmp/tpc_roles.sql
# Création de la base
createdb --owner tpc_owner tpc
# L'erreur sur un schéma 'public' existant est normale
pg_restore -d tpc /tmp/tpc.dump
```

Les mots de passe sont dans le script `/tmp/tpc_roles.sql`. Pour vous connecter :

```
$ psql -U tpc_admin -h localhost -d tpc
```

> Ajouter une colonne `email` de type `text` à la table `contacts`. Cette colonne va permettre de stocker l'adresse e-mail des clients et des fournisseurs. Ajouter également un commentaire décrivant cette colonne dans le catalogue de PostgreSQL (utiliser la commande `COMMENT`).

```
-- Ajouter une colonne email de type text
ALTER TABLE contacts
ADD COLUMN email text;

-- Ajouter un commentaire
COMMENT ON COLUMN contacts.email IS 'Adresse e-mail du contact';
```

> Mettre à jour la table des contacts pour indiquer l'adresse e-mail de *Client6657* qui est `client6657@dalibo.com`.

```
UPDATE contacts
   SET email = 'client6657@dalibo.com'
WHERE nom = 'Client6657';
```

Vérifier les résultats :

```
SELECT *
FROM contacts
WHERE nom = 'Client6657';
```

> Ajouter une contrainte d'intégrité qui valide que la valeur de la colonne `email` créée est bien formée (vérifier que la chaîne de caractère contient au moins le caractère `@`).

```
ALTER TABLE contacts
ADD CONSTRAINT chk_contacts_email_valid
   CHECK (email LIKE '%@%');
```

Cette expression régulière est simplifiée et simpliste pour les besoins de l'exercice. Des expressions régulières plus complexes permettent de valider réellement une adresse e-mail.

Voici un exemple un tout petit peu plus évolué en utilisant une expression rationnelle simple, ici pour vérifier que la chaîne précédent le caractère `@` contient au moins un caractère, et que la chaîne le suivant est une chaîne de caractères contenant un point :

```
ALTER TABLE contacts
ADD CONSTRAINT chk_contacts_email_valid
   CHECK (email ~ '.+@.+\..+');
```

> Valider la contrainte dans une transaction de test.

Démarrer la transaction :

```
BEGIN ;
```

Tenter de mettre à jour la table `contacts` avec une adresse e-mail ne répondant pas à la contrainte :

```
UPDATE contacts
   SET email = 'test';
```

L'ordre `UPDATE` retourne l'erreur suivante, indiquant que l'expression régulière est fonctionnelle :

```
ERROR:  new row for relation "contacts" violates check constraint
        "chk_contacts_email_valid"
DETAIL: Failing row contains
        (300001, Client1737, nkD, SA, 20-999-929-1440, test).
```

La transaction doit être ensuite annulée :

```
ROLLBACK ;
```

> Déterminer quels sont les contacts qui disposent d'une adresse e-mail et affichez leur nom ainsi que le code de leur pays.

```
SELECT nom, code_pays
FROM contacts
WHERE email IS NOT NULL;
```

> La génération des numéros de commande est actuellement réalisée à l'aide de la séquence `commandes_commande_id_seq`. Cette méthode ne permet pas de garantir que tous les numéros de commande se suivent. Proposer une solution pour sérialiser la génération des numéros de commande. Autrement dit, proposer une méthode pour obtenir un numéro de commande sans avoir de « trou » dans la séquence en cas d'échec d'une transaction.

La solution la plus simple pour imposer la sérialisation des numéros de commandes est d'utiliser une table de séquences. Une ligne de cette table correspondra au compteur des numéros de commande.

```
-- création de la table qui va contenir la séquence :
CREATE TABLE numeros_sequences (
   nom text NOT NULL PRIMARY KEY,
   sequence integer NOT NULL
```

```
);

-- initialisation de la séquence :
INSERT INTO numeros_sequences (nom, sequence)
SELECT 'sequence_numero_commande', max(numero_commande)
FROM commandes;
```

L'obtention d'un nouveau numéro de commande sera réalisé dans la transaction de création de la commande de la façon suivante :

```
BEGIN ;

UPDATE numeros_sequences
   SET sequence = sequence + 1
WHERE nom = 'numero_commande'
RETURNING sequence;

/* insertion d'une nouvelle commande en utilisant le numéro de commande
   retourné par la commande précédente :
   INSERT INTO commandes (numero_commande, ...)
   VALUES (<la nouvelle valeur de la séquence>, ...) ;
*/

COMMIT ;
```

L'ordre `UPDATE` pose un verrou exclusif sur la ligne mise à jour. Tant que la mise à jour n'aura pas été validée ou annulée par `COMMIT` ou `ROLLBACK`, le verrou posé va bloquer toutes les autres transactions qui tenteraient de mettre à jour cette ligne. De cette façon, toutes les transactions seront sérialisées.

Concernant la génération des numéros de séquence, si la transaction est annulée, alors le compteur `sequence` retrouvera sa valeur précédente et la transaction suivante obtiendra le même numéro de séquence. Si la transaction est validée, alors le compteur `sequence` est incrémenté. La transaction suivante verra alors cette nouvelle valeur et non plus l'ancienne. Cette méthode garantit qu'il n'y ait pas de rupture de séquence.

Il va de soi que les transactions de création de commandes doivent être extrêmement courtes. Si une telle transaction est bloquée, toutes les transactions suivantes seront également bloquées, paralysant ainsi tous les utilisateurs de l'application.

> Noter le nombre de lignes de la table `pieces`. Dans une transaction, majorer de 5% le prix des pièces de moins de 1500 € et minorer de 5 % le prix des pièces dont le prix actuel est égal ou supérieur à 1500 €. Vérifier que le nombre de lignes mises à jour au total correspond au nombre total de lignes de la table `pieces`.

```
BEGIN ;

SELECT count(*)
FROM pieces;

UPDATE pieces
   SET prix = prix * 1.05
WHERE prix < 1500;
```

```
UPDATE pieces
   SET prix = prix * 0.95
WHERE prix >= 1500;
```

Au total, la transaction a mis à jour 214200 (99922+114278) lignes, soit 14200 lignes de trop mises à jour.

Annuler la mise à jour :

```
ROLLBACK ;
```

Explication : Le premier `UPDATE` a majoré de 5 % les pièces dont le prix est inférieur à 1500 €. Or, tous les prix supérieurs à 1428,58 € passent la barre des 1500 € après le premier `UPDATE`. Le second `UPDATE` minore les pièces dont le prix est égal ou supérieur à 1500 €, ce qui inclue une partie des prix majorés par le précédent `UPDATE`. Certaines lignes ont donc subies *deux* modifications au lieu d'une. L'instruction `CASE` du langage SQL, qui sera abordée dans le prochain module, propose une solution à ce genre de problématique :

```
UPDATE pieces
   SET prix = (
      CASE
         WHEN prix < 1500 THEN prix * 1.05
         WHEN prix >= 1500 THEN prix * 0.95
      END
   );
```

> Dans une même transaction, créer un nouveau client en incluant l'ajout de l'ensemble des informations requises pour pouvoir le contacter. Un nouveau client a un solde égal à 0.

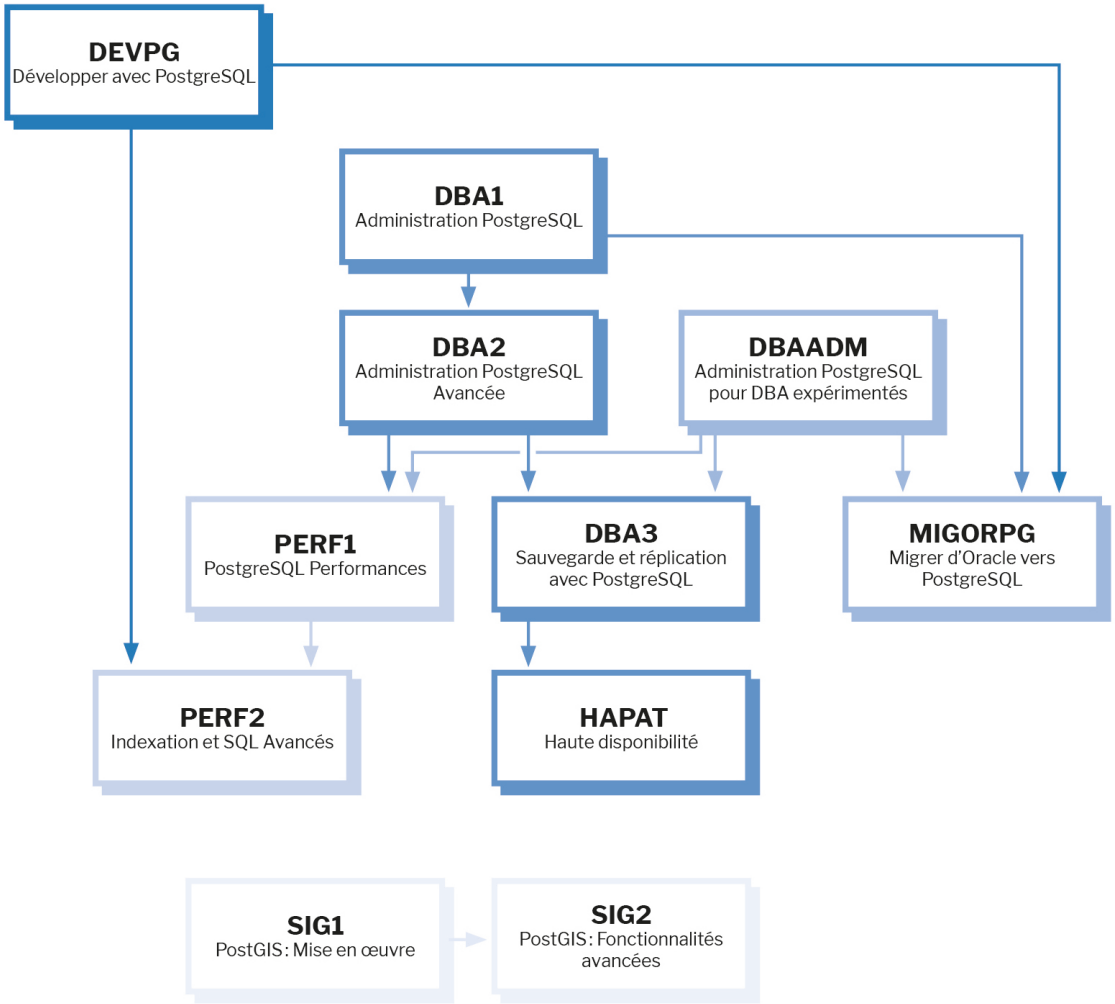
```
-- démarrer la transaction
BEGIN ;

-- créer le contact et récupérer le contact_id généré
INSERT INTO contacts (nom, adresse, telephone, code_pays)
   VALUES ('M. Xyz', '3, Rue du Champignon, 96000 Champiville',
           '+33554325432', 'FR')
RETURNING contact_id;

-- réaliser l'insertion en utilisant le numéro de contact récupéré précédemment
INSERT INTO clients (solde, segment_marche, contact_id, commentaire)
   -- par exemple ici avec le numéro 350002
   VALUES (0, 'AUTOMOBILE', 350002, 'Client très important');

-- valider la transaction
COMMIT ;
```

Les formations Dalibo

Retrouvez nos formations et le calendrier sur https://dali.bo/formation

Pour toute information ou question, n'hésitez pas à nous écrire sur contact@dalibo.com.

Cursus des formations



Retrouvez nos formations dans leur dernière version :

— DBA1 : Administration PostgreSQL
https://dali.bo/dba1
— DBA2 : Administration PostgreSQL avancé
https://dali.bo/dba2
— DBA3 : Sauvegarde et réplication avec PostgreSQL
https://dali.bo/dba3
— DEV1 : Introduction à SQL
https://dali.bo/dev1
— DEVPG : Développer avec PostgreSQL
https://dali.bo/devpg
— PERF1 : PostgreSQL Performances
https://dali.bo/perf1
— PERF2 : Indexation et SQL avancés
https://dali.bo/perf2
— MIGORPG : Migrer d'Oracle à PostgreSQL
https://dali.bo/migorpg
— HAPAT : Haute disponibilité avec PostgreSQL
https://dali.bo/hapat

Les livres blancs

— Migrer d'Oracle à PostgreSQL
https://dali.bo/dlb01
— Industrialiser PostgreSQL
https://dali.bo/dlb02
— Bonnes pratiques de modélisation avec PostgreSQL
https://dali.bo/dlb04
— Bonnes pratiques de développement avec PostgreSQL
https://dali.bo/dlb05

Téléchargement gratuit

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open source ou sous licence Creative Commons.