Module S6

Types de base

25.03

Table des matières

Sur ce document . . . 1
Chers lectrices & lecteurs, . . . 1
À propos de DALIBO . . . 1
Remerciements . . . 2
Forme de ce manuel . . . 2
Licence Creative Commons CC-BY-NC-SA . . . 2
Marques déposées . . . 3
Versions de PostgreSQL couvertes . . . 3

1/ Types de base **5**
1.0.1 Préambule . . . 5
1.0.2 Menu . . . 5
1.0.3 Objectifs . . . 6
1.1 Les types de données . . . 7
1.1.1 Qu'est-ce qu'un type ? . . . 7
1.1.2 Impact sur les performances . . . 7
1.1.3 Impacts sur l'intégrité . . . 7
1.1.4 Impacts fonctionnels . . . 8
1.2 Types numériques . . . 9
1.2.1 Types numériques : entiers . . . 9
1.2.2 Types numériques : flottants . . . 10
1.2.3 Types numériques : numeric . . . 10
1.2.4 Opérations sur les numériques . . . 11
1.2.5 Choix d'un type numérique . . . 12
1.3 Types temporels . . . 14
1.3.1 Types temporels : date . . . 14
1.3.2 Types temporels : time . . . 15
1.3.3 Types temporels : timestamp . . . 15
1.3.4 Types temporels : timestamp with time zone . . . 16
1.3.5 Types temporels : interval . . . 17
1.3.6 Choix d'un type temporel . . . 17
1.4 Types chaînes . . . 19
1.4.1 Types chaînes : caractères . . . 19
1.4.2 Types chaînes : binaires . . . 19
1.4.3 Quel type choisir ? . . . 20
1.4.4 Collation . . . 20
1.4.5 Collation & sources . . . 23
1.5 Types avancés . . . 26
1.5.1 Types faiblement structurés . . . 26
1.5.2 `json` . . . 26

1.5.3 `xml` . . . 27
1.6 Types intervalle de valeurs . . . 28
1.6.1 `range` . . . 28
1.6.2 Manipulation . . . 29
1.6.3 Contraintes d'exclusion . . . 30
1.7 Types géométriques . . . 33
1.8 Types utilisateurs . . . 34
1.8.1 Types composites . . . 34
1.8.2 Type énumération . . . 35

Les formations Dalibo **37**
Cursus des formations . . . 37
Les livres blancs . . . 38
Téléchargement gratuit . . . 38

Sur ce document

Formation	Module S6
Titre	Types de base
Révision	25.03
PDF	https://dali.bo/s6_pdf
EPUB	https://dali.bo/s6_epub
HTML	https://dali.bo/s6_html
Slides	https://dali.bo/s6_slides

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, Florent Jardin, Virginie Jourdan, Luc Lamarle, Denis Laxalde, Guillaume Lelarge, Alain Lesage, Benoit Lobréau, Jean-Louis Louër, Thibaut Madelaine, 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 formation est sous licence **CC-BY-NC-SA**[2]. Vous êtes libre de la redistribuer et/ou modifier aux conditions suivantes :

- Paternité
- Pas d'utilisation commerciale
- Partage des conditions initiales à l'identique

Vous n'avez pas le droit d'utiliser cette création à des fins commerciales.

Si vous modifiez, transformez ou adaptez cette création, vous n'avez le droit de distribuer la création qui en résulte que sous un contrat identique à celui-ci.

Vous devez citer le nom de l'auteur original de la manière indiquée par l'auteur de l'œuvre ou le titulaire des droits qui vous confère cette autorisation (mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'œuvre). À chaque réutilisation ou distribution de cette création, vous devez faire apparaître clairement au public les conditions contractuelles de sa mise à disposition. La meilleure manière de les indiquer est un lien vers cette page web. Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits sur cette œuvre. Rien dans ce contrat ne diminue ou ne restreint le droit moral de l'auteur ou des auteurs.

Le texte complet de la licence est disponible sur http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

[2] http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

Cela inclut 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/ Types de base

- PostgreSQL offre un système de typage complet
  - types standards
  - types avancés propres à PostgreSQL

1.0.1 Préambule

- SQL possède un typage fort
  - le type utilisé décrit la donnée manipulée
  - garantit l'intégrité des données
  - primordial au niveau fonctionnel
  - garantit les performances

1.0.2 Menu

- Qu'est-ce qu'un type ?
- Les types SQL standards
  - numériques
  - temporels
  - textuels et binaires
- Les types avancés de PostgreSQL

1.0.3 Objectifs

- Comprendre le système de typage de PostgreSQL
- Savoir choisir le type adapté à une donnée
- Être capable d'utiliser les types avancés à bon escient

1.1 LES TYPES DE DONNÉES

- Qu'est-ce qu'un type ?
- Représentation physique
- Impacts sur l'intégrité
- Impacts fonctionnels

1.1.1 Qu'est-ce qu'un type ?

- Un type définit :
  - les valeurs que peut prendre une donnée
  - les opérateurs applicables à cette donnée

1.1.2 Impact sur les performances

- Choisir le bon type pour :
  - optimiser les performances
  - optimiser le stockage

1.1.3 Impacts sur l'intégrité

- Le bon type de données garantit l'intégrité des données :
  - la bonne représentation
  - le bon intervalle de valeur

Le choix du type employé pour stocker une donnée est primordial pour garantir l'intégrité des données.

Par exemple, sur une base de données mal conçue, il peut arriver que les dates soient stockées sous la forme d'une chaîne de caractère. Ainsi, une date malformée ou invalide pourra être enregistrée dans la base de données, passant outre les mécanismes de contrôle d'intégrité de la base de données. Si une date est stockée dans une colonne de type `date`, alors ces problèmes ne se posent pas :

```
postgres=# CREATE TABLE test_date (dt date);
CREATE TABLE

postgres=# INSERT INTO test_date VALUES ('2015-0717');
ERROR:  invalid input syntax for type date: "2015-0717"
LINE 1: INSERT INTO test_date VALUES ('2015-0717');
                                      ^
postgres=# INSERT INTO test_date VALUES ('2015-02-30');
ERROR:  date/time field value out of range: "2015-02-30"
LINE 1: INSERT INTO test_date VALUES ('2015-02-30');

postgres=# INSERT INTO test_date VALUES ('2015-07-17');
INSERT 0 1
```

1.1.4 Impacts fonctionnels

- Un type de données offre des opérateurs spécifiques :
  - comparaison
  - manipulation
- Exemple: une date est-elle comprise entre deux dates données ?

1.2 TYPES NUMÉRIQUES

- Entiers
- Flottants
- Précision fixée

1.2.1 Types numériques : entiers

- 3 types entiers :
  - `smallint` : 2 octets
  - `integer` : 4 octets
  - `bigint` : 8 octets
- Valeur exacte
- Signé
- Utilisation :
  - véritable entier
  - clé technique

1.2.2 Types numériques : flottants

- 2 types flottants :
  - `real` / `float4`
  - `double precision` / `float8`
- Données numériques « floues »
  - valeurs non exactes
- Utilisation :
  - stockage des données issues de capteurs

1.2.3 Types numériques : numeric

- 1 type
  - `numeric(.., ..)`
- Type exact
  - mais calcul lent
- Précision choisie : totale, partie décimale
- Utilisation :
  - données financières
  - calculs exacts
- Déconseillé pour :
  - clés primaires
  - données non exactes (ex : résultats de capteurs)

1.2.4 Opérations sur les numériques

- Indexable : `>`, `>=`, `=`, `<=`, `<`
- `+`, `-`, `/`, `*`, modulo (`%`), puissance (`^`)
- Pour les entiers :
  - `AND`, `OR`, `XOR` (`&`, `|`, `#`)
  - décalage de bits (*shifting*): `>>`, `<<`
- Attention aux conversions (*casts*) / promotions !

Tout les types numériques sont indexables avec des indexes standards btree, permettant la recherche avec les opérateurs d'égalité / inégalité. Pour les entiers, il est possible de réaliser des opérations bit-à-bit :

```
SELECT 2 | 4;

 ?column?
----------
        6
(1 ligne)

SELECT 7 & 3;

 ?column?
----------
        3
(1 ligne)
```

Il faut toutefois être vigilant face aux opérations de cast implicites et de promotions des types numériques. Par exemple, les deux requêtes suivantes ramèneront le même résultat, mais l'une sera capable d'utiliser un éventuel index sur `id`, l'autre non :

```
EXPLAIN SELECT * FROM t1 WHERE id = 10::int4;

                                QUERY PLAN
-------------------------------------------------------------------------
 Bitmap Heap Scan on t1  (cost=4.67..52.52 rows=50 width=4)
   Recheck Cond: (id = 10)
   ->  Bitmap Index Scan on t1_id_idx  (cost=0.00..4.66 rows=50 width=0)
         Index Cond: (id = 10)
(4 lignes)

EXPLAIN SELECT * FROM t1 WHERE id = 10::numeric;

                     QUERY PLAN
-----------------------------------------------------
 Seq Scan on t1  (cost=0.00..195.00 rows=50 width=4)
   Filter: ((id)::numeric = 10::numeric)
(2 lignes)
```

Cela peut paraître contre-intuitif, mais la conversion est réalisée dans ce sens pour ne pas perdre d'information. Par exemple, si la valeur numérique cherchée n'est pas un entier. Il faut donc faire spécialement attention aux types utilisés côté applicatif. Avec un ORM tel qu'Hibernate, il peut être tentant de faire correspondre un `BigInt` à un `numeric` côté SQL, ce qui engendrera des casts implicites, et potentiellement des indexes non utilisés.

1.2.5 Choix d'un type numérique

- `integer` ou `biginteger` :
  - identifiants (clés primaires et autre)
  - nombres entiers
- `numeric` :
  - valeurs décimales exactes
  - performance non critique
- `float`, `real` :
  - valeurs flottantes, non exactes
  - performance demandée : `SUM()`, `AVG()`, etc.

Pour les identifiants, il est préférable d'utiliser des entiers ou grands entiers. En effet, il n'est pas nécessaire de s'encombrer du bagage technique et de la pénalité en performance dû à l'utilisation de `numeric`. Contrairement à d'autres SGBD, PostgreSQL ne transforme pas un `numeric` sans partie décimale en entier, et celui-ci souffre donc des performances inhérentes au type `numeric`.

De même, lorsque les valeurs sont entières, il faut utiliser le type adéquat.

Pour les nombres décimaux, lorsque la performance n'est pas critique, préférer le type `numeric` : il est beaucoup plus simple de raisonner sur ceux-ci et leur précision que de garder à l'esprit les subtilités du standard IEEE 754 définissant les opérations sur les flottants. Dans le cas de données décimales nécessitant une précision exacte, il est impératif d'utiliser le type `numeric`.

Les nombres flottants (`float` et `real`) ne devraient être utilisés que lorsque les implications en terme de perte de précision sont intégrées, et que la performance d'un type `numeric` devient gênante. En pratique, cela est généralement le cas lors d'opérations d'agrégations.

Pour bien montrer les subtilités des types `float`, et les risques auquels ils nous exposent, considérons l'exemple suivant, en créant une table contenant 25000 fois la valeur `0.4`, stockée soit en `float` soit en `numeric` :

```
CREATE TABLE t_float AS (
    SELECT 0.04::float AS cf,
           0.04::numeric AS cn
    FROM generate_series(1, 25000)
);

SELECT sum(cn), sum(cf) FROM t_float ;

   sum   |       sum
---------+-----------------
 1000.00 | 999.99999999967
(1 ligne)
```

Si l'on considère la performance de ces opérations, on remarque des temps d'exécution bien différents :

```
SELECT sum(cn) FROM t_float ;

   sum
---------
 1000.00
(1 ligne)

Temps : 10,611 ms

SELECT sum(cf) FROM t_float ;

       sum
-----------------
 999.99999999967
(1 ligne)

Temps : 6,434 ms
```

Pour aller (beaucoup) plus loin, le document suivant détaille le comportement des flottants selon le standard IEEE[1].

[1] https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

1.3 TYPES TEMPORELS

- Date
- Date & heure
  - …avec ou sans fuseau

1.3.1 Types temporels : date

- `date`
  - représente une date, sans heure
  - affichage format ISO : `YYYY-MM-DD`
- Utilisation :
  - stockage d'une date lorsque la composante heure n'est pas utilisée
- Cas déconseillés :
  - stockage d'une date lorsque la composante heure est utilisée

```
SELECT now()::date ;
```

```
    now
------------
 2019-11-13
```

1.3.2 Types temporels : time

- `time`
  - représente une heure sans date
  - affichage format ISO `HH24:MI:SS`
- Peu de cas d'utilisation
- À éviter :
  - stockage d'une date et de la composante heure dans deux colonnes

```
SELECT now()::time ;

        now
-----------------
 15:19:39.947677
```

1.3.3 Types temporels : timestamp

- `timestamp` (`without time zone` !)
  - représente une date et une heure
  - fuseau horaire non précisé
- Utilisation :
  - stockage d'une date et d'une heure

```
SELECT now()::timestamp ;

             now
----------------------------
 2019-11-13 15:20:54.222233
```

Le nom réel est `timestamp without time zone`. Comme on va le voir, il faut lui préférer le type `timestamptz`.

1.3.4 Types temporels : timestamp with time zone

- `timestamp with time zone` = `timestamptz`
  - représente une date et une heure
  - fuseau horaire inclus
  - affichage : `2019-11-13 15:33:00.824096+01`
- Utilisation :
  - stockage d'une date et d'une heure, cadre mondial
  - à préférer à `timestamp without time zone`

Ces deux exemples ont été exécutés à quelques secondes d'intervalle sur des instances en France (heure d'hiver) et au Brésil :

```
SHOW timezone;

   TimeZone
--------------
 Europe/Paris

SELECT  now() ;

              now
-------------------------------
 2019-11-13 15:32:09.615455+01

SHOW timezone;

  TimeZone
-------------
 Brazil/West
(1 ligne)

SELECT now() ;

              now
-------------------------------
 2019-11-13 10:32:39.536972-04

SET timezone to 'Europe/Paris' ;

SELECT now() ;

              now
-------------------------------
 2019-11-13 15:33:00.824096+01
```

On préférera presque tout le temps le type `timestamptz` à `timestamp` (sans fuseau horaire), ne serait-ce qu'à cause des heures d'été et d'hiver. Les deux types occupent 8 octets.

1.3.5 Types temporels : interval

- `interval`
  - représente une durée
- Utilisation :
  - exprimer une durée
  - dans une requête, pour modifier une date/heure existante

1.3.6 Choix d'un type temporel

- Préférer les types avec *timezone*
  - toujours plus simple à gérer au début qu'à la fin
- Considérer les types `range` pour tout couple « début / fin »
- Utiliser `interval` / `generate_series`

De manière générale, il est beaucoup plus simple de gérer des dates avec timezone côté base. En effet, dans le cas où une seule timezone est gérée, les clients ne verront pas la différence. Si en revanche les besoins évoluent, il sera beaucoup plus simple de gérer les différentes timezones à ce moment là.

Les points suivants concernent plus de la modélisation que des types de données à proprement parler, mais il est important de considérer les types `range` dès lors que l'on souhaite stocker un couple « date de début / date de fin ». Nous aurons l'occasion de revenir sur ces types dans la suite de ce module.

Enfin, une problématique assez commune consiste à vouloir effectuer des jointures contre une table de dates de références. Une (mauvaise) solution à ce problème consiste à stocker ces dates dans une table. Il est beaucoup plus avantageux en terme de maintenance de ne pas stocker ces dates, mais de les générer à la volée. Par exemple, pour générer tous les jours de janvier 2015 :

```
postgres=# SELECT * FROM generate_series(
    '2015-01-01',
    '2015-01-31',
    '1 day'::interval
);
    generate_series
------------------------
 2015-01-01 00:00:00+01
```

```
2015-01-02 00:00:00+01
2015-01-03 00:00:00+01
2015-01-04 00:00:00+01
2015-01-05 00:00:00+01
2015-01-06 00:00:00+01
2015-01-07 00:00:00+01
2015-01-08 00:00:00+01
2015-01-09 00:00:00+01
2015-01-10 00:00:00+01
2015-01-11 00:00:00+01
2015-01-12 00:00:00+01
2015-01-13 00:00:00+01
2015-01-14 00:00:00+01
2015-01-15 00:00:00+01
2015-01-16 00:00:00+01
2015-01-17 00:00:00+01
2015-01-18 00:00:00+01
2015-01-19 00:00:00+01
2015-01-20 00:00:00+01
2015-01-21 00:00:00+01
2015-01-22 00:00:00+01
2015-01-23 00:00:00+01
2015-01-24 00:00:00+01
2015-01-25 00:00:00+01
2015-01-26 00:00:00+01
2015-01-27 00:00:00+01
2015-01-28 00:00:00+01
2015-01-29 00:00:00+01
2015-01-30 00:00:00+01
2015-01-31 00:00:00+01
```

1.4 TYPES CHAÎNES

- Texte à longueur variable
- Binaires

En général on choisira une chaîne de longueur variable. Nous ne parlerons pas ici du type `char` (à taille fixe), d'utilisation très restreinte.

1.4.1 Types chaînes : caractères

- `varchar(_n_)`, `text`
- Représentent une chaîne de caractères
- Valident l'encodage
- Valident la longueur maximale de la chaîne (contrainte !)
- Utilisation :
  - stocker des chaînes de caractères non binaires

1.4.2 Types chaînes : binaires

- `bytea`
- Stockage de données binaires
  - encodage en hexadécimal ou séquence d'échappement
- Utilisation :
  - stockage de courtes données binaires
- Cas déconseillés :
  - stockage de fichiers binaires

Le type `bytea` permet de stocker des données binaires dans une base de données PostgreSQL.

1.4.3 Quel type choisir ?

- `varchar` (sans limite) ou `text` (non standard)
- Implémenter la limite avec une contrainte
  - plus simple à modifier

```
CREATE TABLE t1 (c1 varchar CHECK (length(c1) < 10))
```

En règle générale, il est recommandé d'utiliser un champ de type `varchar` tout court, et de vérifier la longueur au niveau d'une contrainte. En effet, il sera plus simple de modifier celle-ci par la suite, en modifiant uniquement la contrainte. De plus, la contrainte permet plus de possibilités, comme par exemple d'imposer une longueur minimale.

1.4.4 Collation

- L'ordre de tri dépend des langues & de conventions variables
- Collation par colonne / index / requête
- `SELECT * FROM mots ORDER BY t COLLATE "C" ;`
- `CREATE TABLE messages ( id int, fr TEXT COLLATE "fr_FR.utf8", de TEXT COLLATE "de_DE.utf8" );`

L'ordre de tri des chaînes de caractère (« collation ») peut varier suivant le contenu d'une colonne. Rien que parmi les langues européennes, il existe des spécificités propres à chacune, et même à différents pays pour une même langue. Si l'ordre des lettres est une convention courante, il existe de nombreuses variations propres à chacune (comme é, à, æ, ö, ß, å, ñ…), avec des règles de tri propres. Certaines lettres peuvent être assimilées à une combinaison d'autres lettres. De plus, la place relative des majuscules, celles des chiffres, ou des caractères non alphanumérique est une pure affaire de convention.

La collation dépend de l'encodage (la manière de stocker les caractères), de nos jours généralement `UTF8`[2] (standard Unicode). PostgreSQL utilise par défaut UTF8 et il est chaudement conseillé de ne pas changer cela.

[2] https://fr.wikipedia.org/wiki/UTF-8

La collation par défaut dans une base est définie à sa création, et est visible avec `\l` (ci-dessous pour une installation en français). Le type de caractères est généralement identique.

```
\l
```

```
                    Liste des bases de données
Nom        | Propriétaire | Encodage | Collationnement | Type caract. |...
-----------+--------------+----------+-----------------+--------------+
 pgbench   | pgbench      | UTF8     | fr_FR.UTF-8     | fr_FR.UTF-8  |
 postgres  | postgres     | UTF8     | fr_FR.UTF-8     | fr_FR.UTF-8  |
 template0 | postgres     | UTF8     | fr_FR.UTF-8     | fr_FR.UTF-8  | ...
 template1 | postgres     | UTF8     | fr_FR.UTF-8     | fr_FR.UTF-8  | ...
```

Parmi les collations que l'on peut rencontrer, il y a par exemple `en_US.UTF-8` (la collation par défaut de beaucoup d'installations), ou `C`, basée sur les caractères ASCII et les valeurs des octets. De vieilles installations peuvent encore contenir `fr_FR.iso885915@euro`.

Si le tri par défaut ne convient pas, on peut le changer à la volée dans la requête SQL, au besoin après avoir créé la collation.

Exemple avec du français :

```
CREATE TABLE mots (t text) ;

INSERT INTO mots
VALUES ('A'),('a'),('aa'),('z'),('ä'),('å'),('Å'),('aa'),('æ'),('ae'),('af'), ('ß'),
 ↪  ('ss') ;

SELECT * FROM mots ORDER BY t ; -- sous-entendu, ordre par défaut en français ici
```

```
 t
 a
 A
 å
 Å
 ä
 aa
 aa
 ae
 æ
 af
 ss
 ß
 z
```

Noter que les caractères « æ » et « ß » sont correctement assimilés à « ae » et « ss ». (Ce serait aussi le cas avec `en_US.utf8` ou `de_DE.utf8`).

Avec la collation `C`, l'ordre est plus basique, soit celui des codes UTF-8 :

```
SELECT * FROM mots ORDER BY t COLLATE "C" ;
```

```
 t
 A
 a
```

```
aa
aa
ae
af
ss
z
Å
ß
ä
å
æ
```

Un intérêt de la collation `C` est qu'elle est plus simple et se repose sur la glibc du système, ce qui lui permet d'être souvent plus rapide qu'une des collations ci-dessus. Il suffit donc parfois de remplacer `ORDER BY champ_texte` par `ORDER BY champ_text COLLATE "C"`, à condition bien sûr que l'ordre ASCII convienne.

Il est possible d'indiquer dans la définition de chaque colonne quelle doit être sa collation par défaut :

Pour du danois :

```
-- La collation doit exister sur le système d'exploitation
CREATE COLLATION IF NOT EXISTS "da_DK" (locale='da_DK.utf8');

ALTER TABLE mots ALTER COLUMN t TYPE text COLLATE "da_DK" ;

SELECT * FROM mots ORDER BY t ; -- ordre danois
```

```
t
A
a
ae
af
ss
ß
z
æ
ä
Å
å
aa
```

Dans cette langue, les majuscules viennent traditionnellement avant les minuscules, et « å» et « aa » viennent après le « z ».

Avec une collation précisée dans la requête, un index peut ne pas être utilisable. En effet, par défaut, il est trié sur disque dans l'ordre de la collation de la colonne. Un index peut cependant se voir affecter une collation différente de celle de la colonne, par exemple pour un affichage ou une interrogation dans plusieurs langues :

```
CREATE INDEX ON mots (t);                    -- collation par défaut de la colonne
CREATE INDEX ON mots (t COLLATE "de_DE.utf8"); -- tri allemand
```

La collation n'est pas qu'une question d'affichage. Le tri joue aussi dans la sélection quand il y a des inégalités, et le français et le danois revoient ici des résultats différents :

```
SELECT * FROM mots WHERE t > 'z' COLLATE "fr_FR";

 t
(0 ligne)

SELECT * FROM mots WHERE t > 'z' COLLATE "da_DK";

 t
 aa
 ä
 å
 Å
 aa
 æ
(6 lignes)
```

1.4.5 Collation & sources

Source des collations :

- le système : installations séparées nécessaires, différences entre OS
- (>= v 10) : librairie externe ICU

```
CREATE COLLATION danois (provider = icu, locale = 'da-x-icu') ;
```

Des collations comme `en_US.UTF-8` ou `fr_FR.UTF-8` sont dépendantes des locales installées sur la machine. Cela implique qu'elles peuvent subtilement différer entre deux systèmes, même entre deux versions d'un même système d'exploitation ! De plus, la locale voulue n'est pas forcément présente, et son mode d'installation dépend du système d'exploitation et de sa distribution…

Pour éliminer ces problèmes tout en améliorant la flexibilité, PostgreSQL 10 a introduit les collations ICU, c'est-à-dire standardisées et versionnées dans une librairie séparée. En pratique, les paquets des distributions l'installent automatiquement avec PostgreSQL. Les collations linguistiques sont donc immédiatement disponibles via ICU :

```
CREATE COLLATION danois (provider = icu, locale = 'da-x-icu') ;
```

La librairie ICU fournit d'autres collations plus spécifiques liées à un contexte, par exemple l'ordre d'un annuaire ou l'ordre suivant la casse. Par exemple, cette collation très pratique tient compte de la valeur des chiffres (« tri naturel ») :

```
CREATE COLLATION nombres (provider = icu, locale = 'fr-u-kn-kr-latn-digit');

SELECT * FROM
  (VALUES ('1 sou'),('01 sou'),('02 sous'),('2 sous'),
  ('10 sous'),('0100 sous') ) AS n(n)
ORDER BY n COLLATE nombres ;
```

```
     n
-----------
 01 sou
 1 sou
 02 sous
 2 sous
 10 sous
 0100 sous
```

Alors que, par défaut, « 02 » précéderait « 1 » :

```
SELECT * FROM
  (VALUES ('1 sou'),('01 sou'),('02 sous'),('2 sous'),
  ('10 sous'),('0100 sous') ) AS n(n)
ORDER BY n ;  -- tri avec la locale par défaut
```

```
     n
-----------
 0100 sous
 01 sou
 02 sous
 10 sous
 1 sou
 2 sous
```

Pour d'autres exemples et les détails, voir ce billet de Peter Eisentraut[3] et la documentation officielle[4].

Pour voir les collations disponibles, consulter `pg_collation` :

```
  SELECT collname, collcollate, collprovider, collversion
  FROM pg_collation WHERE collname LIKE 'fr%' ;
```

```
  collname   | collcollate | collprovider | collversion
-------------+-------------+--------------+-------------
 fr-BE-x-icu | fr-BE       | i            | 153.80
 fr-BF-x-icu | fr-BF       | i            | 153.80
 fr-CA-x-icu | fr-CA       | i            | 153.80.32.1
 fr-x-icu    | fr          | i            | 153.80
 ...
 fr_FR       | fr_FR.utf8  | c            | ¤
 fr_FR.utf8  | fr_FR.utf8  | c            | ¤
 fr_LU       | fr_LU.utf8  | c            | ¤
 fr_LU.utf8  | fr_LU.utf8  | c            | ¤
(57 lignes)
```

Les collations installées dans la base sont visibles avec `\dO` sous `psql` :

```
\dO
```

```
                Liste des collationnements
 Schéma |     Nom      |    Collationnement    | … | Fournisseur | …
--------+--------------+-----------------------+---+-------------+----
 public | belge        | fr-BE-x-icu           | … | icu         | …
```

[3] https://blog.2ndquadrant.com/icu-support-postgresql-10/
[4] https://docs.postgresql.fr/current/collation.html

```
public | chiffres_fin | fr-u-kn-kr-latn-digit | … | icu         | …
public | da_DK        | da_DK.utf8            | … | libc        | …
public | danois       | da-x-icu              | … | icu         | …
public | de_DE        | de_DE.utf8            | … | libc        | …
public | de_phonebook | de-u-co-phonebk       | … | icu         | …
public | es_ES        | es_ES.utf8            | … | libc        | …
public | espagnol     | es-x-icu              | … | icu         | …
public | fr_FR        | fr_FR.utf8            | … | libc        | …
public | français     | fr-FR-x-icu           | … | icu         | …
```

1.5 TYPES AVANCÉS

- PostgreSQL propose des types plus avancés
- De nombreuses extensions !
  - faiblement structurés (`JSON` …)
  - intervalle
  - géométriques
  - tableaux

1.5.1 Types faiblement structurés

- PostgreSQL propose plusieurs types faiblement structurés :
  - `hstore` (clé/valeur historique)
  - `json`
  - `xml`

1.5.2 `json`

- `json`
  - stockage sous forme d'une chaîne de caractère
  - valide un document JSON sans modification
- `jsonb` (PG > 9.4)
  - stockage binaire optimisé
  - beaucoup plus de fonctions (dont jsonpath en v12)
  - à préférer

1.5.3 `xml`

- `xml`
  - stocke un document XML
  - valide sa structure
- Quelques opérateurs disponibles

1.6 TYPES INTERVALLE DE VALEURS

- Représentation d'intervalle
  - utilisable avec plusieurs types : entiers, dates, timestamps, etc.
  - contrainte d'exclusion

1.6.1 `range`

- représente un intervalle de valeurs continues
  - entre deux bornes
  - incluses ou non
- plusieurs types natifs
  - `int4range`, `int8range`, `numrange`
  - `daterange`, `tsrange`, `tstzrange`

Les intervalles de valeurs (`range`) représentent un ensemble de valeurs continues comprises entre deux bornes. Ces dernières sont entourées par des crochets `[` et `]` lorsqu'elles sont incluses, et par des parenthèses `(` et `)` lorsqu'elles sont exclues. L'absence de borne est admise et correspond à l'infini.

- `[0,10]` : toutes les valeurs comprises entre 0 à 10 ;
- `(100,200]` : toutes les valeurs comprises entre 100 et 200, 100 exclu ;
- `[2021-01-01,)` : toutes les dates supérieures au 1er janvier 2021 inclus ;
- `empty` : aucune valeur ou intervalle vide.

Le type abstrait `anyrange` se décline en `int4range` (`int`), `int8range` (`bigint`), `numrange` (`numeric`), `daterange` (`date`), `tsrange` (`timestamp without timezone`), `tstzrange` (`timestamp with timezone`).

1.6.2 Manipulation

- opérateurs spécifiques `*`, `&&`, `<@` ou `@>`
- indexation avec GiST ou SP-GiST
- types personnalisés

Les opérateurs d'inclusion `<@` et `@>` déterminent si une valeur ou un autre intervalle sont contenus dans l'intervalle de gauche ou de droite.

```
SELECT produit, date_validite FROM produits
 WHERE date_validite @> '2020-01-01'::date;
```

```
               produit                |      date_validite
--------------------------------------+-------------------------
 a0fd7a5a-6deb-4454-b7a7-9cd38eef53a4 | [2012-07-12,)
 79eb3a63-eb76-43b9-b1d6-f9f82dd77460 | [2019-07-31,2021-04-01)
 e4edaac4-33f1-426d-b2b0-4ea3b1c6caec | (,2020-01-02)
```

L'opérateur de chevauchement `&&` détermine si deux intervalles du même type disposent d'au moins une valeur commune.

```
SELECT produit, date_validite FROM produits
 WHERE date_validite && '[2021-01-01,2021-12-31]'::daterange
```

```
               produit                |      date_validite
--------------------------------------+-------------------------
 8791d13f-bdfe-46f8-afc6-8be33acdbfc7 | [2012-07-12,)
 000a72d5-a90f-4030-aa15-f0a05e54b701 | [2019-07-31,2021-04-01)
```

L'opérateur d'intersection `*` reconstruit l'intervalle des valeurs continues et communes entre deux intervalles.

```
SELECT '[2021-01-01,2021-12-31]'::daterange
     * '[2019-07-31,2021-04-01)'::daterange AS intersection;
```

```
      intersection
-------------------------
 [2021-01-01,2021-04-01)
```

Pour garantir des temps de réponse acceptables sur les recherches avancées avec les opérateurs ci-dessus, il est nécessaire d'utiliser les index GiST ou SP-GiST. La syntaxe est la suivante :

```
CREATE INDEX ON produits USING gist (date_validite);
```

Enfin, il est possible de créer ses propres types `range` personnalisés à l'aide d'une fonction de différence. L'exemple ci-dessous permet de manipuler l'intervalle de données pour le type `time`. La fonction `time_subtype_diff()` est tirée de la documentation RANGETYPES-DEFINING[5].

[5] https://docs.postgresql.fr/current/rangetypes.html#RANGETYPES-DEFINING

```
-- fonction utilitaire pour le type personnalisé "timerange"
CREATE FUNCTION time_subtype_diff(x time, y time)
RETURNS float8 AS
  'SELECT EXTRACT(EPOCH FROM (x - y))'
LANGUAGE sql STRICT IMMUTABLE;

-- définition du type "timerange", basé sur le type "time"
CREATE TYPE timerange AS RANGE (
  subtype = time,
  subtype_diff = time_subtype_diff
);

-- Exemple
SELECT '[11:10, 23:00]'::timerange;

      timerange
---------------------
 [11:10:00,23:00:00]
```

1.6.3 Contraintes d'exclusion

- Utilisation :
  - éviter le chevauchement de deux intervalles (`range`)
- Performance :
  - s'appuie sur un index

```
CREATE TABLE vendeurs (
  nickname varchar NOT NULL,
  plage_horaire timerange NOT NULL,
  EXCLUDE USING GIST (plage_horaire WITH &&)
);
```

Une contrainte d'exclusion s'apparente à une contrainte d'unicité, mais pour des intervalles de valeurs. Le principe consiste à identifier les chevauchements entre deux lignes pour prévenir l'insertion d'un doublon sur un intervalle commun.

Dans l'exemple suivant, nous utilisons le type personnalisé `timerange`, présenté ci-dessus. La table `vendeurs` reprend les agents de vente d'un magasin et leurs plages horaires de travail, valables pour tous les jours ouvrés de la semaine.

```
CREATE TABLE vendeurs (
  nickname varchar NOT NULL,
  plage_horaire timerange NOT NULL,
  EXCLUDE USING GIST (plage_horaire WITH &&)
);
```

```
INSERT INTO vendeurs (nickname, plage_horaire)
VALUES
  ('john', '[09:00:00,11:00:00)'::timerange),
  ('bobby', '[11:00:00,14:00:00)'::timerange),
  ('jessy', '[14:00:00,17:00:00)'::timerange),
  ('thomas', '[17:00:00,20:00:00]'::timerange);
```

Un index GiST est créé automatiquement pour la colonne `plage_horaire`.

```
\x on
\di+

List of relations
-[ RECORD 1 ]-+---------------------------
Schema        | public
Name          | vendeurs_plage_horaire_excl
Type          | index
Owner         | postgres
Table         | vendeurs
Persistence   | permanent
Access method | gist
Size          | 8192 bytes
Description   |
```

L'ajout d'un nouveau vendeur pour une plage déjà couverte par l'un de ces collègues est impossible, avec une violation de contrainte d'exclusion, gérée par l'opérateur de chevauchement `&&`.

```
INSERT INTO vendeurs (nickname, plage_horaire)
VALUES ('georges', '[10:00:00,12:00:00)'::timerange);

ERROR:  conflicting key value violates exclusion constraint
        "vendeurs_plage_horaire_excl"
DETAIL:  Key (plage_horaire)=([10:00:00,12:00:00)) conflicts
         with existing key (plage_horaire)=([09:00:00,11:00:00)).
```

Il est aussi possible de mixer les contraintes d'unicité et d'exclusion grâce à l'extension `btree_gist`. Dans l'exemple précédent, nous imaginons qu'un nouveau magasin ouvre et recrute de nouveaux vendeurs. La contrainte d'exclusion doit évoluer pour prendre en compte une nouvelle colonne, `magasin_id`.

```
CREATE EXTENSION btree_gist;
ALTER TABLE vendeurs
  DROP CONSTRAINT IF EXISTS vendeurs_plage_horaire_excl,
  ADD COLUMN magasin_id int NOT NULL DEFAULT 1,
  ADD EXCLUDE USING GIST (magasin_id WITH =, plage_horaire WITH &&);

INSERT INTO vendeurs (magasin_id, nickname, plage_horaire)
VALUES (2, 'georges', '[10:00:00,12:00:00)'::timerange);
```

En cas de recrutement pour une plage horaire déjà couverte par le nouveau magasin, la contrainte d'exclusion lèvera toujours une erreur, comme attendu.

```
INSERT INTO vendeurs (magasin_id, nickname, plage_horaire)
VALUES (2, 'laura', '[09:00:00,11:00:00)'::timerange);
```

```
ERROR:  conflicting key value violates exclusion constraint
        "vendeurs_magasin_id_plage_horaire_excl"
DETAIL:  Key (magasin_id, plage_horaire)=(2, [09:00:00,11:00:00)) conflicts
         with existing key (magasin_id, plage_horaire)=(2, [10:00:00,12:00:00)).
```

1.7 TYPES GÉOMÉTRIQUES

- Plusieurs types natifs 2D :
  - point, ligne, segment, polygone, cercle
- Utilisation :
  - stockage de géométries simples, sans référentiel de projection
- Pour la géographie :
  - extension PostGIS

1.8 TYPES UTILISATEURS

- Plusieurs types définissables par l'utilisateur
  - types composites
  - domaines
  - enums

1.8.1 Types composites

- Regroupe plusieurs attributs
  - la création d'une table implique la création d'un type composite associé
- Utilisation :
  - déclarer un tableau de données composites
  - en PL/pgSQL, déclarer une variable de type enregistrement

Les types composites sont assez difficiles à utiliser, car ils nécessitent d'adapter la syntaxe spécifiquement au type composite. S'il ne s'agit que de regrouper quelques attributs ensemble, autant les lister simplement dans la déclaration de la table.

En revanche, il peut être intéressant pour stocker un tableau de données composites dans une table.

1.8.2 Type énumération

- Ensemble fini de valeurs possibles
  - uniquement des chaînes de caractères
  - 63 caractères maximum
- Équivalent des énumérations des autres langages
- Utilisation :
  - listes courtes figées (statuts…)
  - évite des jointures

Référence :

- Type énumération[6]

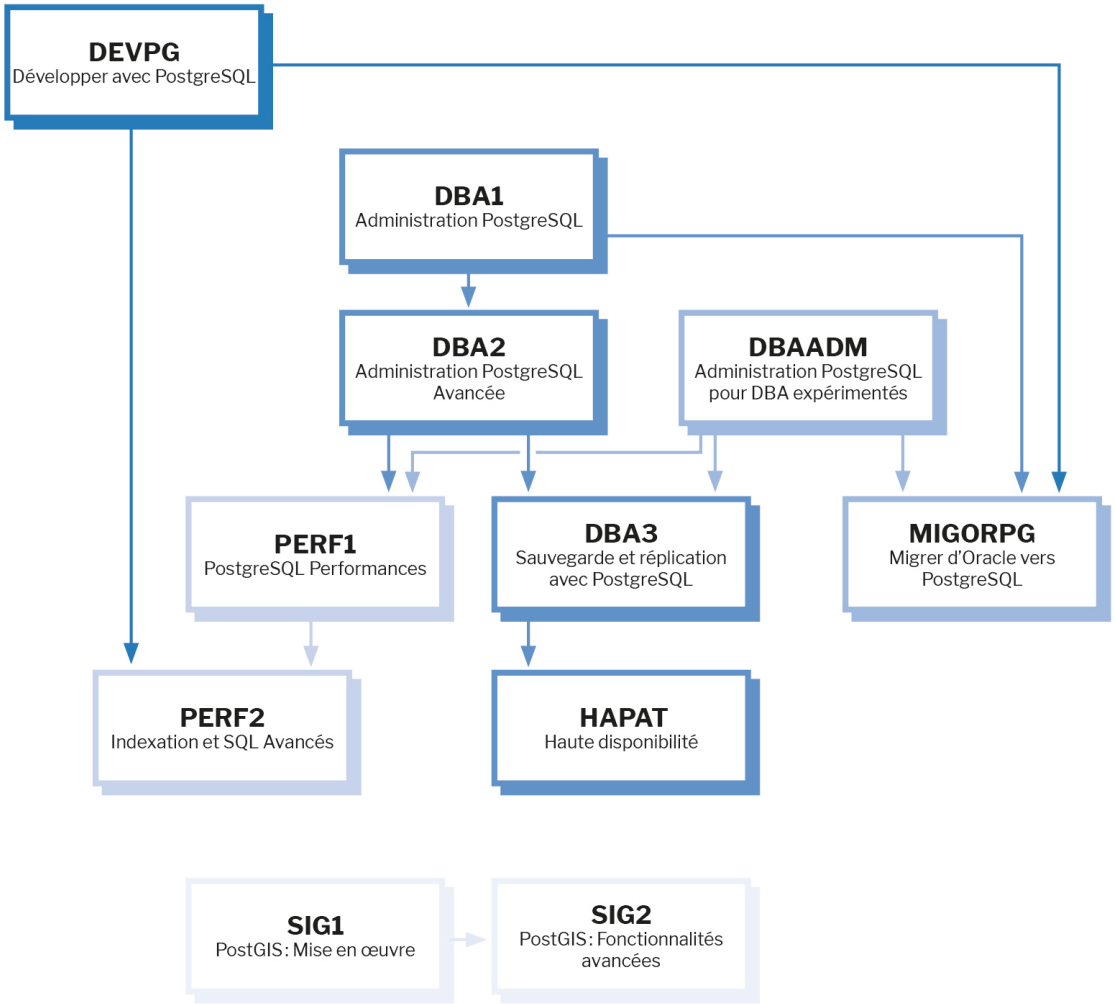
[6] https://docs.postgresql.fr/current/datatype-enum.html

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
- 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.