Module V1

Partitionnement sous PostgreSQL

24.04

Table des matières

Sur ce document

Formation	Module V1
Titre	Partitionnement sous PostgreSQL
Révision	24.04
PDF	https://dali.bo/v1_pdf
EPUB	https://dali.bo/v1_epub
HTML	https://dali.bo/v1_html
Slides	https://dali.bo/v1_slides
TP	https://dali.bo/v1_tp
TP (solutions)	https://dali.bo/v1_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 :

Jean-Paul Argudo, Alexandre Anriot, Carole Arnaud, Alexandre Baron, David Bidoc, Sharon Bonan, Franck Boudehen, Arnaud Bruniquel, Pierrick Chovelon, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Jehan-Guillaume de Rorthais, Ronan Dunklau, Vik Fearing, Stefan Fercot, Pierre Giraud, Nicolas Gollet, Dimitri Fontaine, 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, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, 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

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

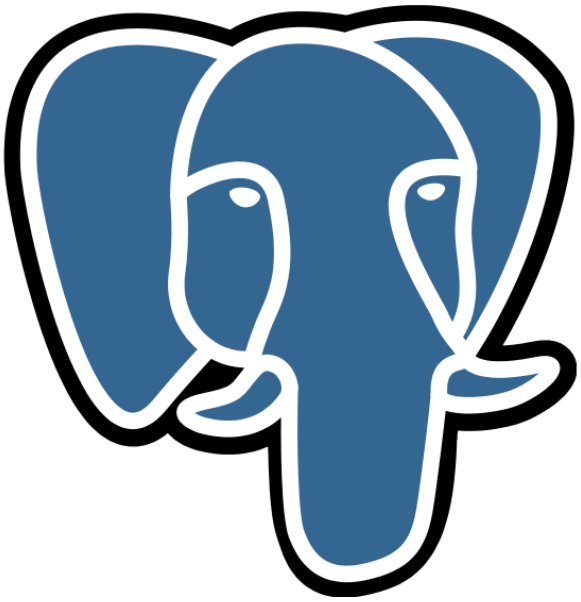
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 12 à 16.

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/ Partitionnement sous PostgreSQL



- Ses principes et intérêts
- Historique
- Les différents types

1.1 PRINCIPE & INTÉRÊTS DU PARTITIONNEMENT

- Faciliter la maintenance de gros volumes
  - `VACUUM` (`FULL`), réindexation, déplacements, sauvegarde logique…
- Performances
  - parcours complet sur de plus petites tables
  - statistiques par partition plus précises
  - purge par partitions entières
  - `pg_dump` parallélisable
  - tablespaces différents (données froides/chaudes)
- Attention à la maintenance sur le code

Maintenir de très grosses tables peut devenir fastidieux, voire impossible : `VACUUM FULL` trop long, espace disque insuffisant, autovacuum pas assez réactif, réindexation interminable… Il est aussi aberrant de conserver beaucoup de données d'archives dans des tables lourdement sollicitées pour les données récentes.

Le partitionnement consiste à séparer une même table en plusieurs sous-tables (partitions) manipulables en tant que tables à part entière.

Maintenance

La maintenance s'effectue sur les partitions plutôt que sur l'ensemble complet des données. En particulier, un `VACUUM FULL` ou une réindexation peuvent s'effectuer partition par partition, ce qui permet de limiter les interruptions en production, et lisser la charge. `pg_dump` ne sait pas paralléliser la sauvegarde d'une table volumineuse et non partitionnée, mais parallélise celle de différentes partitions d'une même table.

C'est aussi un moyen de déplacer une partie des données dans un autre *tablespace* pour des raisons de place, ou pour déporter les parties les moins utilisées de la table vers des disques plus lents et moins chers.

Parcours complet de partitions

Certaines requêtes (notamment décisionnelles) ramènent tant de lignes, ou ont des critères si complexes, qu'un parcours complet de la table est souvent privilégié par l'optimiseur.

Un partitionnement, souvent par date, permet de ne parcourir qu'une ou quelques partitions au lieu de l'ensemble des données. C'est le rôle de l'optimiseur de choisir la partition (*partition pruning*), par exemple celle de l'année en cours, ou des mois sélectionnés.

Suppression des partitions

La suppression de données parmi un gros volume peut poser des problèmes d'accès concurrents ou de performance, par exemple dans le cas de purges.

En configurant judicieusement les partitions, on peut résoudre cette problématique en supprimant une partition (`DROP TABLE nompartition ;`), ou en la *détachant* (`ALTER TABLE table_partitionnee DETACH PART`) pour l'archiver (et la réattacher au besoin) ou la supprimer ultérieurement.

D'autres optimisations sont décrites dans ce billet de blog d'Adrien Nayrat[1] : statistiques plus précises au niveau d'une partition, réduction plus simple de la fragmentation des index, jointure par rapprochement des partitions…

La principale difficulté d'un système de partitionnement consiste à partitionner avec un impact minimal sur la maintenance du code par rapport à une table classique.

[1] https://blog.anayrat.info/2021/09/01/cas-dusages-du-partitionnement-natif-dans-postgresql/

1.2 PARTITIONNEMENT APPLICATIF

- …ou la réinvention de la roue
- Gestion au niveau applicatif, table par table
- Complexité pour le développeur
- Intégrité des données ?

L'application peut gérer le partitionnement elle-même, par exemple en créant des tables numérotées par mois, année… Le moteur de PostgreSQL ne voit que des tables classiques et ne peut assurer l'intégrité entre ces données.

C'est au développeur de réinventer la roue : choix de la table, gestion des index… La lecture des données qui concerne plusieurs tables peut devenir délicate.

Ce modèle extrêmement fréquent est bien sûr à éviter.

1.3 MÉTHODES DE PARTITIONNEMENT INTÉGRÉES À POSTGRESQL

- Partitionnement par héritage (historique)
- Partitionnement déclaratif (>=v10, préférer v13+)

Un partitionnement entièrement géré par le moteur, n'existe réellement que depuis la version 10 de PostgreSQL. Il a été grandement amélioré en versions 11 et 12, en fonctionnalités comme en performances.

Jusqu'à PostgreSQL 9.6 n'existaient que le partitionnement dit par héritage, nettement moins flexible, et bien sûr le partitionnement géré intégralement par l'applicatif.

1.4 PARTITIONNEMENT PAR HÉRITAGE

- Historique ou pour cas très spécifique
- Syntaxe :

```
CREATE TABLE primates (debout boolean) INHERITS (mammiferes) ;
```

- Table mère :
  - définie normalement, peut contenir des lignes
- Tables filles :
  - héritent des propriétés de la table mère
  - …mais pas des contraintes, index et droits
  - colonnes supplémentaires possibles
- Insertion applicative ou par trigger (lent !)

Principe du partitionnement par héritage :

PostgreSQL permet de créer des tables qui héritent les unes des autres. L'héritage d'une table mère transmet les propriétés suivantes à la table fille :

- les colonnes, avec le type et les valeurs par défaut ;
- les contraintes `CHECK`.

Les tables filles peuvent ajouter leurs propres colonnes. Par exemple :

```
CREATE TABLE animaux (nom text PRIMARY KEY);
INSERT INTO animaux VALUES ('Éponge');
INSERT INTO animaux VALUES ('Ver de terre');

CREATE TABLE cephalopodes (nb_tentacules integer) INHERITS (animaux);
INSERT INTO cephalopodes VALUES ('Poulpe', 8);

CREATE TABLE vertebres (nb_membres integer) INHERITS (animaux);

CREATE TABLE tetrapodes () INHERITS (vertebres);
ALTER TABLE ONLY tetrapodes ALTER COLUMN nb_membres SET DEFAULT 4 ;

CREATE TABLE poissons (eau_douce boolean) INHERITS (tetrapodes);
INSERT INTO poissons (nom, eau_douce) VALUES ('Requin', false);
INSERT INTO poissons (nom, nb_membres, eau_douce) VALUES ('Anguille', 0, false);
```

La table `poissons` possède les champs des tables dont elle hérite :

```
\d+ poissons
                    Table "public.poissons"
```

```
   Column    |  Type   | Collation | Nullable | Default | …
-------------+---------+-----------+----------+---------+---
 nom         | text    |           | not null |         | …
 nb_membres | integer |           |          | 4       | …
 eau_douce   | boolean |           |          |         | …
Inherits: tetrapodes
Access method: heap
```

On peut créer toute une hiérarchie avec des branches parallèles, chacune avec ses colonnes propres :

```
CREATE TABLE reptiles (venimeux boolean) INHERITS (tetrapodes);
INSERT INTO reptiles VALUES ('Crocodile', 4, false);
INSERT INTO reptiles VALUES ('Cobra', 0, true);

CREATE TABLE mammiferes () INHERITS (tetrapodes);

CREATE TABLE cetartiodactyles (
    cornes boolean,
    bosse boolean
) INHERITS (mammiferes);
INSERT INTO cetartiodactyles VALUES ('Girafe', 4, true, false);
INSERT INTO cetartiodactyles VALUES ('Chameau', 4, false, true);

CREATE TABLE primates (debout boolean) INHERITS (mammiferes);
INSERT INTO primates (nom, debout) VALUES ('Chimpanzé', false);
INSERT INTO primates (nom, debout) VALUES ('Homme', true);

\d+ primates

                    Table "public.primates"
   Column    |  Type   | Collation | Nullable | Default | …
-------------+---------+-----------+----------+---------+---
 nom         | text    |           | not null |         | …
 nb_membres | integer |           |          | 4       | …
 debout      | boolean |           |          |         | …
Inherits: mammiferes
Access method: heap
```

On remarquera que la clé primaire manque. En effet, l'héritage ne transmet pas :

- les contraintes d'unicité et référentielles ;
- les index ;
- les droits.

Chaque table possède ses propres lignes :

```
SELECT * FROM poissons ;

   nom    | nb_membres | eau_douce
----------+------------+-----------
 Requin   |          4 | f
 Anguille |          0 | f
```

Par défaut une table affiche aussi le contenu de ses tables filles et les colonnes communes :

```
SELECT * FROM animaux ORDER BY 1 ;
```

```
   nom
--------------
 Anguille
 Chameau
 Chimpanzé
 Cobra
 Crocodile
 Éponge
 Girafe
 Homme
 Poulpe
 Requin
 Ver de terre
```

```
SELECT * FROM tetrapodes ORDER BY 1 ;
```

```
    nom     | nb_membres
------------+------------
 Anguille   |          0
 Chameau    |          4
 Chimpanzé  |          4
 Cobra      |          0
 Crocodile  |          4
 Girafe     |          4
 Homme      |          4
 Requin     |          4
```

```
EXPLAIN SELECT * FROM tetrapodes ORDER BY 1 ;
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Sort  (cost=420.67..433.12 rows=4982 width=36)
   Sort Key: tetrapodes.nom
   ->  Append  (cost=0.00..114.71 rows=4982 width=36)
         ->  Seq Scan on tetrapodes  (cost=0.00..0.00 rows=1 width=36)
         ->  Seq Scan on poissons  (cost=0.00..22.50 rows=1250 width=36)
         ->  Seq Scan on reptiles  (cost=0.00..22.50 rows=1250 width=36)
         ->  Seq Scan on mammiferes  (cost=0.00..0.00 rows=1 width=36)
         ->  Seq Scan on cetartiodactyles  (cost=0.00..22.30 rows=1230 width=36)
         ->  Seq Scan on primates  (cost=0.00..22.50 rows=1250 width=36)
```

Pour ne consulter que le contenu de la table sans ses filles :

```
SELECT * FROM ONLY animaux ;
```

```
     nom
--------------
 Éponge
 Ver de terre
```

Limites et problèmes :

En conséquence, on a bien affaire à des tables indépendantes. Rien n'empêche d'avoir des doublons entre la table mère et la table fille. Cela empêche aussi bien sûr la mise en place de clé étrangère,

puisqu'une clé étrangère s'appuie sur une contrainte d'unicité de la table référencée. Lors d'une insertion, voire d'une mise à jour, le choix de la table cible se fait par l'application ou un trigger sur la table mère.

Il faut être vigilant à bien recréer les contraintes et index manquants ainsi qu'à attribuer les droits sur les objets de manière adéquate. L'une des erreurs les plus fréquentes est d'oublier de créer les contraintes, index et droits qui n'ont pas été transmis.

Ce type de partitionnement est un héritage des débuts de PostgreSQL, à l'époque de la mode des « bases de donnée objet ». Dans la pratique, dans les versions antérieures à la version 10, l'héritage était utilisé pour mettre en place le partitionnement. La maintenance des index, des contraintes et la nécessité d'un trigger pour aiguiller les insertions vers la bonne table fille, ne facilitaient pas la maintenance. Les performances en écritures étaient bien en-deçà des tables classiques ou du nouveau partitionnement déclaratif.

Table partitionnée en détournant le partitionnement par héritage :

```
CREATE TABLE t3 (c1 integer, c2 text);
CREATE TABLE t3_1 (CHECK (c1 BETWEEN       0 AND  999999)) INHERITS (t3);
CREATE TABLE t3_2 (CHECK (c1 BETWEEN 1000000 AND 1999999)) INHERITS (t3);
CREATE TABLE t3_3 (CHECK (c1 BETWEEN 2000000 AND 2999999)) INHERITS (t3);
CREATE TABLE t3_4 (CHECK (c1 BETWEEN 3000000 AND 3999999)) INHERITS (t3);
CREATE TABLE t3_5 (CHECK (c1 BETWEEN 4000000 AND 4999999)) INHERITS (t3);
CREATE TABLE t3_6 (CHECK (c1 BETWEEN 5000000 AND 5999999)) INHERITS (t3);
CREATE TABLE t3_7 (CHECK (c1 BETWEEN 6000000 AND 6999999)) INHERITS (t3);
CREATE TABLE t3_8 (CHECK (c1 BETWEEN 7000000 AND 7999999)) INHERITS (t3);
CREATE TABLE t3_9 (CHECK (c1 BETWEEN 8000000 AND 8999999)) INHERITS (t3);
CREATE TABLE t3_0 (CHECK (c1 BETWEEN 9000000 AND 9999999)) INHERITS (t3);

-- Fonction et trigger de répartition pour les insertions :

CREATE OR REPLACE FUNCTION insert_into() RETURNS TRIGGER
LANGUAGE plpgsql
AS $FUNC$
BEGIN
  IF NEW.c1    BETWEEN       0 AND  999999 THEN
    INSERT INTO t3_1 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 1000000 AND 1999999 THEN
    INSERT INTO t3_2 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 2000000 AND 2999999 THEN
    INSERT INTO t3_3 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 3000000 AND 3999999 THEN
    INSERT INTO t3_4 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 4000000 AND 4999999 THEN
    INSERT INTO t3_5 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 5000000 AND 5999999 THEN
    INSERT INTO t3_6 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 6000000 AND 6999999 THEN
    INSERT INTO t3_7 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 7000000 AND 7999999 THEN
    INSERT INTO t3_8 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 8000000 AND 8999999 THEN
    INSERT INTO t3_9 VALUES (NEW.*);
  ELSIF NEW.c1 BETWEEN 9000000 AND 9999999 THEN
    INSERT INTO t3_0 VALUES (NEW.*);
```

```
  END IF;
  RETURN NULL;
END;
$FUNC$;

CREATE TRIGGER tr_insert_t3
BEFORE INSERT ON t3 FOR EACH ROW
EXECUTE PROCEDURE insert_into();
```

Noter qu'il reste encore à gérer les mises à jour de lignes… À cause de ce trigger, le temps d'insertion peut être allègrement multiplié par huit ou dix par rapport à une insertion dans une table normale ou dans une table avec le partitionnement déclaratif moderne.

La même table en partitionnement déclaratif par liste est :

```
CREATE TABLE t2 (c1 integer, c2 text) PARTITION BY RANGE (c1);
CREATE TABLE t2_1 PARTITION OF t2 FOR VALUES FROM (      0) TO ( 1000000);
CREATE TABLE t2_2 PARTITION OF t2 FOR VALUES FROM (1000000) TO ( 2000000);
CREATE TABLE t2_3 PARTITION OF t2 FOR VALUES FROM (2000000) TO ( 3000000);
CREATE TABLE t2_4 PARTITION OF t2 FOR VALUES FROM (3000000) TO ( 4000000);
CREATE TABLE t2_5 PARTITION OF t2 FOR VALUES FROM (4000000) TO ( 5000000);
CREATE TABLE t2_6 PARTITION OF t2 FOR VALUES FROM (5000000) TO ( 6000000);
CREATE TABLE t2_7 PARTITION OF t2 FOR VALUES FROM (6000000) TO ( 7000000);
CREATE TABLE t2_8 PARTITION OF t2 FOR VALUES FROM (7000000) TO ( 8000000);
CREATE TABLE t2_9 PARTITION OF t2 FOR VALUES FROM (8000000) TO ( 9000000);
CREATE TABLE t2_0 PARTITION OF t2 FOR VALUES FROM (9000000) TO (10000000);
```

Si le partitionnement par héritage fonctionne toujours sur les versions récentes de PostgreSQL, il est déconseillé pour les nouveaux développements.

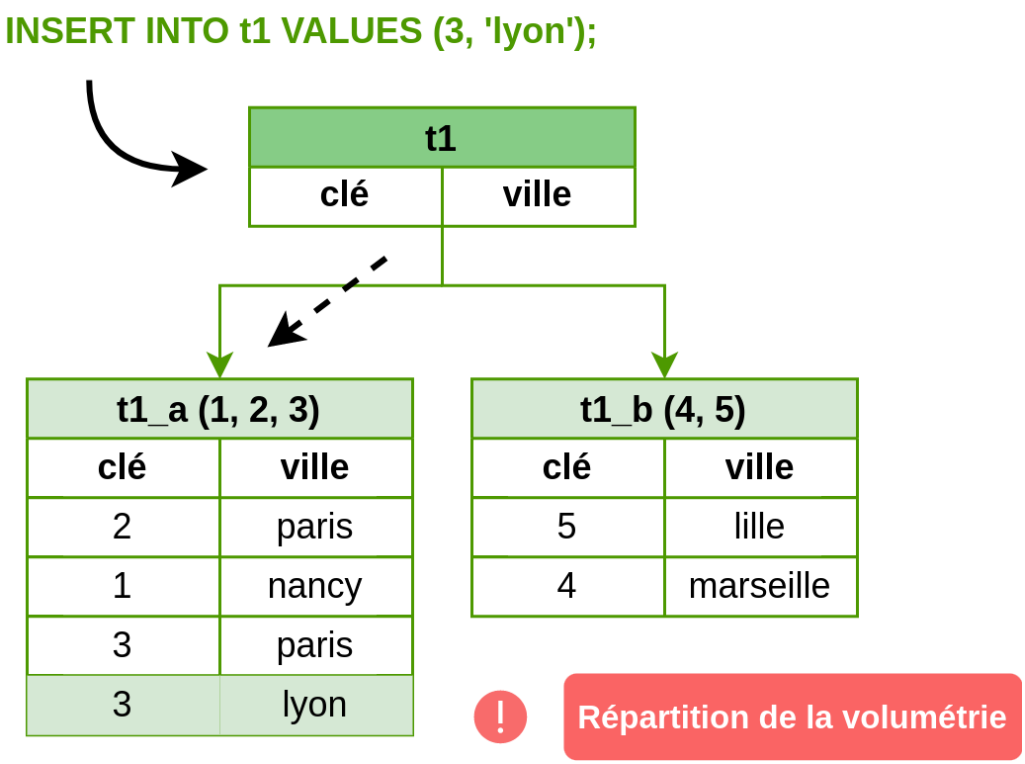
1.5 PARTITIONNEMENT DÉCLARATIF

- Préférer version 13+
- Mise en place et administration simplifiées (intégrées au moteur)
- Gestion automatique des lectures et écritures (et rapide)
- Partitions
  - attacher/détacher une partition
  - contrainte implicite de partitionnement
  - expression possible pour la clé de partitionnement
  - sous-partitions possibles
  - partition par défaut

Le partitionnement déclaratif est le système à privilégier de nos jours. Apparu en version 10, il est à présent mûr. Son but est de permettre une mise en place et une administration simples des tables partitionnées. Des clauses spécialisées ont été ajoutées aux ordres SQL, comme `CREATE TABLE` et `ALTER TABLE`, pour attacher (`ATTACH PARTITION`) et détacher des partitions (`DETACH PARTITION`).

Au niveau de la simplification de la mise en place par rapport à l'ancien partitionnement par héritage, on peut noter qu'il n'est pas nécessaire de créer une fonction *trigger* ni d'ajouter des *triggers* pour gérer les insertions et mises à jour. Le routage est géré de façon automatique en fonction de la définition des partitions, au besoin vers une partition par défaut, et sans pénalité notable en performances. Contrairement au partitionnement par héritage, la table partitionnée ne contient pas elle-même de ligne, ce n'est qu'une coquille vide.

1.5.1 Partitionnement par liste



1.5.2 Partitionnement par liste : implémentation

- Liste de valeurs par partition
  - statut, client, pays, année ou mois…
- Clé de partitionnement forcément mono-colonne
- Syntaxe :

```
CREATE TABLE t1(c1 integer, c2 text) PARTITION BY LIST (c1) ;

CREATE TABLE t1_a PARTITION OF t1 FOR VALUES IN (1, 2, 3);
CREATE TABLE t1_b PARTITION OF t1 FOR VALUES IN (4, 5);
…
```

Il est possible de partitionner une table par valeurs. Ce type de partitionnement fonctionne

uniquement avec une clé de partitionnement mono-colonne (on verra qu'il est possible de sous-partitionner). Il faut que le nombre de valeurs soit assez faible pour être listé explicitement. Le partitionnement par liste est adapté par exemple au partitionnement par :

- type d'un objet ;
- client final (si peu de clients) ;
- géographie : pays, entité commerciale, entrepôt… ;
- statut d'un objet, pour isoler les données froides dans leur partition ;
- année ou mois (ne pas oublier de créer régulièrement de nouvelles partitions) pour faciliter les purges.

Voici un exemple de création d'une table partitionnée par liste et de ses partitions :

```
CREATE TABLE t1(c1 integer, c2 text) PARTITION BY LIST (c1);

CREATE TABLE t1_a PARTITION OF t1 FOR VALUES IN (1, 2, 3);
CREATE TABLE t1_b PARTITION OF t1 FOR VALUES IN (4, 5);
```

Les noms des partitions sont à définir par l'utilisateur, il n'y a pas d'automatisme ni de convention particulière.

Lors de l'insertion, les données sont correctement redirigées vers leur partition, comme le montre cette requête :

```
INSERT INTO t1 VALUES (1);
INSERT INTO t1 VALUES (2);
INSERT INTO t1 VALUES (5);
SELECT tableoid::regclass, * FROM t1;
```

```
 tableoid | c1 | c2
----------+----+----
 t1_a     |  1 |
 t1_a     |  2 |
 t1_b     |  5 |
```

Il est aussi possible d'interroger directement une partition (ici `t1_a` et non `t1`) :

```
SELECT * FROM t1_a ;
```

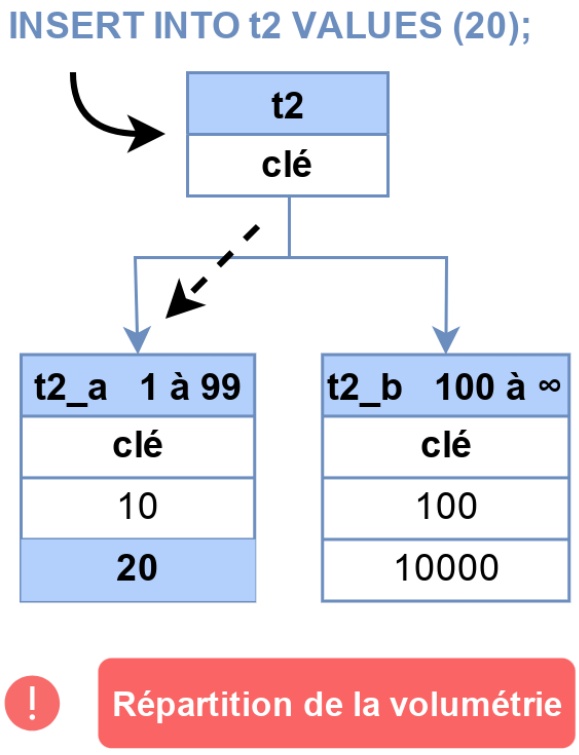
```
 c1 | c2
----+----
  1 |
  2 |
```

Si aucune partition correspondant à la clé insérée n'est trouvée et qu'aucune partition par défaut n'est déclarée, une erreur se produit.

```
INSERT INTO t1 VALUES (0);

ERROR:  no PARTITION OF relation "t1" found for row
DETAIL:  Partition key of the failing row contains (c1) = (0).

INSERT INTO t1 VALUES (6);

ERROR:  no PARTITION OF relation "t1" found for row
DETAIL:  Partition key of the failing row contains (c1) = (6).
```

Si la clé de partitionnement d'une ligne est modifiée par un `UPDATE`, la ligne change automatiquement de partition (sauf en version 10, où ce n'est pas implémenté, et l'on obtient une erreur).

1.5.3 Partitionnement par intervalle



1.5.4 Partitionnement par intervalle : implémentation

- Clé de partitionnement mono- ou multi-colonnes
  - dates, id…
- Bornes :
  - supérieure exclue
  - `MINVALUE` / `MAXVALUE` pour infinis
- Syntaxe :

```
CREATE TABLE t2(c1 integer, c2 text) PARTITION BY RANGE (c1);

CREATE TABLE t2_1 PARTITION OF t2 FOR VALUES FROM (1) TO (100);
CREATE TABLE t2_2 PARTITION OF t2 FOR VALUES FROM (100) TO (MAXVALUE);
…
```

Le partitionnement par intervalle est très courant quand il y a de nombreuses valeurs différentes de la clé de partitionnement, ou qu'elle doit être multicolonne, par exemple :

- selon une tranche de dates, notamment pour faciliter des purges ;
- selon des plages d'identifiants d'objets : client final, produit…

Voici un exemple de création de la table partitionnée et de deux partitions :

```
CREATE TABLE t2(c1 integer, c2 text) PARTITION BY RANGE (c1);
CREATE TABLE t2_1 PARTITION OF t2 FOR VALUES FROM (1) to (100);
CREATE TABLE t2_2 PARTITION OF t2 FOR VALUES FROM (100) TO (MAXVALUE);
```

Le `MAXVALUE` indique la valeur maximale du type de données : `t2_2` acceptera donc tous les entiers supérieurs ou égaux à 100.

Noter que les bornes supérieures des partitions sont **exclues** ! La valeur 100 ira donc dans la seconde partition.

Lors de l'insertion, les données sont redirigées vers leur partition, s'il y en a une :

```
INSERT INTO t2 VALUES (0);

ERROR:  no PARTITION OF relation "t2" found for row
DETAIL:  Partition key of the failing row contains (c1) = (0).

INSERT INTO t2 VALUES (10, 'dix');
```

```
INSERT INTO t2 VALUES (100, 'cent');
INSERT INTO t2 VALUES (10000, 'dix mille');
SELECT * FROM t2 ;
```

```
  c1   |    c2
-------+-----------
    10 | dix
   100 | cent
 10000 | dix mille
(3 lignes)
```

```
SELECT * FROM t2_2 ;
```

```
  c1   |    c2
-------+-----------
   100 | cent
 10000 | dix mille
(2 lignes)
```

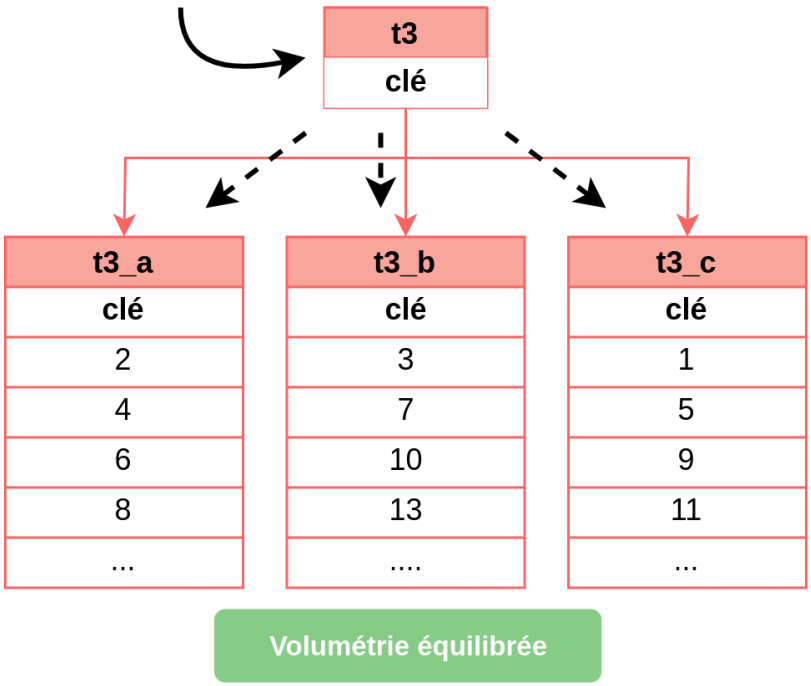
La colonne système `tableoid` permet de connaître la partition d'où provient une ligne :

```
SELECT ctid, tableoid::regclass, * FROM t2 ;
```

```
 ctid  | tableoid |  c1   |    c2
-------+----------+-------+-----------
 (0,1) | t2_1     |    10 | dix
 (0,1) | t2_2     |   100 | cent
 (0,2) | t2_2     | 10000 | dix mille
```

1.5.5 Partitionnement par hachage

INSERT INTO t3 SELECT generate_series(1, 100) ;



1.5.6 Partitionnement par hachage : principe

Pour une répartition uniforme des données :

- Hachage de valeurs par partition
  - indiquer un modulo et un reste
- Clé de partitionnement mono- ou multi-colonnes
- Syntaxe :

```
CREATE TABLE t3(c1 integer, c2 text) PARTITION BY HASH (c1);

CREATE TABLE t3_a PARTITION OF t3 FOR VALUES WITH (modulus 3,remainder 0);
CREATE TABLE t3_b PARTITION OF t3 FOR VALUES WITH (modulus 3,remainder 1);
CREATE TABLE t3_c PARTITION OF t3 FOR VALUES WITH (modulus 3,remainder 2);
```

Si la clé de partitionnement n'est pas évidente et que le besoin est surtout de répartir la volumétrie

en partitions de tailles équivalentes, le partitionnement par hachage est adapté. Voici comment partitionner par hachage une table en trois partitions :

```
CREATE TABLE t3(c1 integer, c2 text) PARTITION BY HASH (c1);
CREATE TABLE t3_1 PARTITION OF t3 FOR VALUES WITH (modulus 3, remainder 0);
CREATE TABLE t3_2 PARTITION OF t3 FOR VALUES WITH (modulus 3, remainder 1);
CREATE TABLE t3_3 PARTITION OF t3 FOR VALUES WITH (modulus 3, remainder 2);
```

Une grosse insertion de données répartira les données de manière équitable entre les différentes partitions :

```
INSERT INTO t3 SELECT generate_series(1, 1000000);

SELECT relname, count(*) FROM t3
JOIN pg_class ON t3.tableoid=pg_class.oid
GROUP BY 1;

 relname | count
---------+--------
 t3_1    | 333263
 t3_2    | 333497
 t3_3    | 333240
```

1.5.7 Clé de partitionnement multi-colonnes

- Clé sur plusieurs colonnes :
  - si partitionnement par intervalle ou hash (pas par liste)
  - et si 1er champ toujours présent
- Syntaxe :

```
CREATE TABLE t1(c1 integer, c2 text, c3 date)
PARTITION BY RANGE (c1, c3) ;

CREATE TABLE t1_a PARTITION OF t1
FOR VALUES FROM (1,'2017-08-10') TO (100, '2017-08-11') ;
…
```

Avec le partitionnement par intervalle, il est possible de créer les partitions en utilisant plusieurs colonnes. On profitera de l'exemple ci-dessous pour montrer l'utilisation conjointe de tablespaces différents. Commençons par créer les tablespaces :

```
CREATE TABLESPACE ts0 LOCATION '/tablespaces/ts0';
CREATE TABLESPACE ts1 LOCATION '/tablespaces/ts1';
CREATE TABLESPACE ts2 LOCATION '/tablespaces/ts2';
CREATE TABLESPACE ts3 LOCATION '/tablespaces/ts3';
```

Créons maintenant la table partitionnée et deux partitions :

```
CREATE TABLE t2(c1 integer, c2 text, c3 date not null)
       PARTITION BY RANGE (c1, c3);
CREATE TABLE t2_1 PARTITION OF t2
       FOR VALUES FROM (1,'2017-08-10') TO (100, '2017-08-11')
       TABLESPACE ts1;
CREATE TABLE t2_2 PARTITION OF t2
       FOR VALUES FROM (100,'2017-08-11') TO (200, '2017-08-12')
       TABLESPACE ts2;
```

La borne supérieure étant exclue, la valeur `(100, '2017-08-11')` fera donc partie de la seconde partition. Si les valeurs sont bien comprises dans les bornes, tout va bien :

```
INSERT INTO t2 VALUES (1, 'test', '2017-08-10');
INSERT INTO t2 VALUES (150, 'test2', '2017-08-11');
```

Mais si la valeur pour `c1` est trop petite :

```
INSERT INTO t2 VALUES (0, 'test', '2017-08-10');
```

```
ERROR:  no partition of relation "t2" found for row
DÉTAIL : Partition key of the failing row contains (c1, c3) = (0, 2017-08-10).
```

De même, si la valeur pour `c3` (colonne de type date) est antérieure :

```
INSERT INTO t2 VALUES (1, 'test', '2017-08-09');
```

```
ERROR:  no partition of relation "t2" found for row
DÉTAIL : Partition key of the failing row contains (c1, c3) = (1, 2017-08-09).
```

Les valeurs spéciales `MINVALUE` et `MAXVALUE` permettent de ne pas indiquer de valeur de seuil limite. Les partitions `t2_0` et `t2_3` pourront par exemple être déclarées comme suit et permettront d'insérer les lignes qui étaient ci-dessus en erreur.

```
CREATE TABLE t2_0 PARTITION OF t2
       FOR VALUES FROM (MINVALUE, MINVALUE) TO (1,'2017-08-10')
       TABLESPACE ts0;

CREATE TABLE t2_3 PARTITION OF t2
       FOR VALUES FROM (200,'2017-08-12') TO (MAXVALUE, MAXVALUE)
       TABLESPACE ts3;
```

Enfin, on peut consulter la table `pg_class` afin de vérifier la présence des différentes partitions :

```
ANALYZE t2;
SELECT relname,relispartition,relkind,reltuples
FROM pg_class WHERE relname LIKE 't2%';
```

```
 relname | relispartition | relkind | reltuples
---------+----------------+---------+-----------
 t2      | f              | p       |         0
 t2_0    | t              | r       |         2
 t2_1    | t              | r       |         1
 t2_2    | t              | r       |         1
 t2_3    | t              | r       |         0
```

Performances :

Si le premier champ de la clé de partitionnement n'est pas fourni, il semble que l'optimiseur ne sache pas cibler correctement les partitions. Il balaiera toutes les partitions. Ce peut être gênant si ce premier champ n'est pas systématiquement présent dans les requêtes.

Le sous-partitionnement est une alternative à étudier, également plus souple.

Il faut faire attention à ce que le nombre de combinaisons possibles ne mène pas à trop de partitions.

1.5.8 Sous-partitionnement

S'il y a deux chemins d'accès privilégiés :

```
CREATE TABLE objets (id int, statut int, annee int, t text)
 PARTITION BY LIST (statut) ;

CREATE TABLE objets_123
 PARTITION OF objets FOR VALUES IN (1, 2, 3)
 PARTITION BY LIST (annee) ;

CREATE TABLE objets_123_2023
 PARTITION OF objets_123 FOR VALUES IN (2023) ;
CREATE TABLE objets_123_2024
 PARTITION OF objets_123 FOR VALUES IN (2024) ;

CREATE TABLE objets_45
 PARTITION OF objets FOR VALUES IN (4,5) ;
```

- Plus souple que le partitionnement multicolonne

Principe :

Les partitions sont des tables à part entière, qui peuvent donc être elles-mêmes partitionnées. Ce peut être utile si les requêtes alternent entre deux schémas d'accès.

Exemple :

L'exemple ci-dessus crée deux partitions selon `statut` (`objets_123` et `objets_45`). La première partition est elle-même sous-partitionnée par année (`objets_123_2023` et `objets_123_2024`). Cela permet par exemple, de faciliter la purge des données ou d'accélérer le temps de traitement si l'on requête sur une année entière. Il n'a pas été jugé nécessaire de sous-partitionner la seconde partition `objets_45` (par exemple parce qu'elle est petite).

```
\dt objets*
                  Liste des relations
 Schéma |       Nom       |        Type        | Propriétaire
--------+-----------------+--------------------+--------------
```

```
 public | objets          | table partitionnée | postgres
 public | objets_123      | table partitionnée | postgres
 public | objets_123_2023 | table              | postgres
 public | objets_123_2024 | table              | postgres
 public | objets_45       | table              | postgres
```

Il n'est pas obligatoire de sous-partitionner avec la même technique (liste, intervalle, hachage…) que le partitionnement de plus haut niveau.

Il n'y a pas besoin de fournir la première clé de partitionnement pour que les sous-partitions soient directement accessibles :

```
EXPLAIN (COSTS OFF) SELECT * FROM objets
WHERE annee = 2023 ;

                QUERY PLAN
-------------------------------------------
 Append
   ->  Seq Scan on objets_123_2023 objets_1
         Filter: (annee = 2023)
   ->  Seq Scan on objets_45 objets_2
         Filter: (annee = 2023)
```

Fournir uniquement la première clé de partitionnement entraînera le parcours de toutes les sous-partitions concernées :

```
EXPLAIN (COSTS OFF) SELECT * FROM objets
WHERE statut = 3 ;

                QUERY PLAN
-------------------------------------------
 Append
   ->  Seq Scan on objets_123_2023 objets_1
         Filter: (statut = 3)
   ->  Seq Scan on objets_123_2024 objets_2
         Filter: (statut = 3)
```

Bien sûr, l'idéal est de fournir les deux clés de partitionnement pour n'accéder qu'à une partition :

```
EXPLAIN (COSTS OFF) SELECT * FROM objets
WHERE statut = 3 AND annee = 2024 ;

                QUERY PLAN
-------------------------------------------
 Seq Scan on objets_123_2024 objets
   Filter: ((statut = 3) AND (annee = 2024))
```

Cette fonctionnalité peut être ponctuellement utile, mais il ne faut pas en abuser en raison de la complexité supplémentaire. Toutes les clés de partitionnement devront se retrouver dans les clés primaires techniques des tables. Le nombre de partitions peut devenir très important.

Comparaison avec le partitionnement multicolonnes :

Le partitionnement multicolonne est conceptuellement plus simple. Pour un même besoin, le nombre de partitions est identique. Mais le sous-partitionnement est plus souple :

- mélange possible des types de partitionnement (par exemple un partitionnement par liste sous-partitionné par intervalle de dates) ;
- type de sous-partitionnement différent possible selon les partitions.

1.5.9 Partition par défaut

- Pour le partitionnement par liste ou par intervalle
- Toutes les données n'allant pas dans les partitions définies iront dans la partition par défaut

```
CREATE TABLE t2_autres PARTITION OF t2 DEFAULT ;
```

- La conserver petite

Ajouter une partition par défaut permet de ne plus avoir d'erreur au cas où une partition n'est pas définie. Par exemple :

```
CREATE TABLE t1(c1 integer, c2 text) PARTITION BY LIST (c1);
CREATE TABLE t1_a PARTITION OF t1 FOR VALUES IN (1, 2, 3);
CREATE TABLE t1_b PARTITION OF t1 FOR VALUES IN (4, 5);

INSERT INTO t1 VALUES (0);

ERROR:  no PARTITION OF relation "t1" found for row
DETAIL:  Partition key of the failing row contains (c1) = (0).

INSERT INTO t1 VALUES (6);

ERROR:  no PARTITION OF relation "t1" found for row
DETAIL:  Partition key of the failing row contains (c1) = (6).

-- partition par défaut
CREATE TABLE t1_defaut PARTITION OF t1 DEFAULT ;
-- on réessaie l'insertion
INSERT INTO t1 VALUES (0);
INSERT INTO t1 VALUES (6);

SELECT tableoid::regclass, * FROM t1;

 tableoid  | c1 | c2
-----------+----+----
 t1_a      |  1 |
 t1_a      |  2 |
 t1_b      |  5 |
 t1_defaut |  0 |
 t1_defaut |  6 |
```

Comme la partition par défaut risque d'être parcourue intégralement à chaque ajout d'une nouvelle partition, il vaut mieux la garder de petite taille.

Un partitionnement par hachage ne peut posséder de table par défaut puisque les données sont forcément aiguillées vers une partition ou une autre.

1.5.10 Attacher une partition

```
ALTER TABLE … ATTACH PARTITION … FOR VALUES … ;
```

- La table doit préexister
- Vérification du respect de la contrainte par les données existantes
  - parcours complet de la table
  - potentiellement lent !
  - …sauf si contrainte `CHECK` identique déjà ajoutée
- Si la partition par défaut a des données qui iraient dans cette partition :
  - erreur à l'ajout de la nouvelle partition
  - détacher la partition par défaut
  - ajouter la nouvelle partition
  - déplacer les données de l'ancienne partition par défaut
  - ré-attacher la partition par défaut

Ajouter une table comme partition d'une table partitionnée est possible mais cela nécessite de vérifier que la contrainte de partitionnement est valide pour toute la table attachée, et que la partition par défaut ne contient pas de données qui devraient figurer dans cette nouvelle partition. Cela résulte en un parcours complet de la table attachée, et de la partition par défaut si elle existe, ce qui sera d'autant plus lent qu'elles sont volumineuses.

Ce peut être très coûteux en disque, mais le plus gros problème est la durée du verrou sur la table partitionnée, pendant toute cette opération. Il est donc conseillé d'ajouter une contrainte `CHECK` adéquate **avant** l'`ATTACH` : la durée du verrou sera raccourcie d'autant.

Si des lignes pour cette nouvelle partition figurent déjà dans la partition par défaut, des opérations supplémentaires sont à réaliser pour les déplacer. Ce n'est pas automatique.

Exemple :

```
\set ECHO all
\set timing off

DROP TABLE IF EXISTS t1 ;
```

```
-- Une table partitionnée avec partition par défaut
CREATE TABLE t1 (c1 integer, filler char (10)) PARTITION BY LIST (c1);
CREATE TABLE t1_123 PARTITION OF t1 FOR VALUES IN (1, 2, 3);
CREATE TABLE t1_45 PARTITION OF t1 FOR VALUES IN (4, 5);
CREATE TABLE t1_default  PARTITION OF t1 DEFAULT ;

-- Données d'origine
INSERT INTO t1 SELECT 1+mod(i,5) FROM generate_series (1,5000000) i;
-- Les données sont bien dans les partitions t1_123 et t1_45
SELECT tableoid::regclass, c1, count(*) FROM t1
GROUP BY 1,2 ORDER BY c1 ;

-- Création d'une table pour les valeurs 6 à attacher
CREATE TABLE t1_6 (LIKE t1 INCLUDING ALL) ;
INSERT INTO t1_6 SELECT 6 FROM generate_series (1,1000000);

\dt+ t1*

\timing on
-- on attache la table : elle est scannée, ce qui est long
ALTER TABLE t1 ATTACH PARTITION t1_6 FOR VALUES IN (6) ;
-- noter la nouvelle contrainte sur la table
\d+ t1_6
-- on la détache, la contrainte a disparu
ALTER TABLE t1 DETACH PARTITION t1_6 ;
\d+ t1_6

-- on remet manuellement la même contrainte que ci-dessus
-- (ce qui reste long mais ne pose pas de verrou sur t1)
ALTER TABLE t1_6 ADD CONSTRAINT t1_6_ck CHECK(c1 IS NOT NULL AND c1 = 6) ;
\d+ t1_6
-- l'ATTACH est cette fois presque instantané
ALTER TABLE t1 ATTACH PARTITION t1_6 FOR VALUES IN (6) ;

\timing off

-- On insère par erreur des valeurs 7 sans avoir fait la partition
-- (et sans avoir le droit de les enlever de t1 ensuite)
INSERT INTO t1 SELECT 7 FROM generate_series (1,100);
-- Créer la partition échoue avec "constraint for default partition "t1_default"
 ↪  would be violated""
CREATE TABLE t1_7 PARTITION OF t1 FOR VALUES IN (7);

-- Pour corriger cela, au sein d'une transaction,
-- on transfére les données de la partition par défaut
-- vers une nouvelle table qui est ensuite attachée
CREATE TABLE t1_7 (LIKE t1 INCLUDING ALL) ;
ALTER TABLE t1_7 ADD CONSTRAINT t1_7_ck CHECK(c1 IS NOT NULL AND c1 = 7) ;
BEGIN ;
    INSERT INTO t1_7 SELECT * FROM t1_default WHERE c1=7 ;
    DELETE FROM t1_default WHERE c1=7 ;
    ALTER TABLE t1 ATTACH PARTITION t1_7 FOR VALUES IN (7) ;
COMMIT ;
```

1.5.11 Détacher une partition

```
ALTER TABLE … DETACH PARTITION …
```

- Simple et rapide
- Mais nécessite un verrou exclusif
  - option `CONCURRENTLY` (v14+)

Détacher une partition est beaucoup plus rapide qu'en attacher une. En effet, il n'est pas nécessaire de procéder à des vérifications sur les données des partitions. La partition détachée devient alors une table tout à fait classique. Elle conserve les index, contraintes, etc. dont elle a pu hériter de la table partitionnée originale.

Cependant, il reste nécessaire d'acquérir un verrou exclusif sur la table partitionnée, ce qui peut prendre du temps si des transactions sont en cours d'exécution. L'option `CONCURRENTLY` (à partir de PostgreSQL 14) mitige le problème malgré quelques restrictions[2], notamment : pas d'utilisation dans une transaction, incompatibilité avec la présence d'une partition par défaut, et nécessité d'une commande `FINALIZE` si l'ordre a échoué ou été interrompu.

1.5.12 Supprimer une partition

```
DROP TABLE nom_partition ;
```

Une partition étant une table, supprimer la table revient à supprimer la partition, et bien sûr les données qu'elle contient. Il n'y a pas besoin de la détacher explicitement auparavant. L'opération est simple et rapide, mais demande un verrou exclusif.

Il est fréquent de partitionner par date pour profiter de cette facilité dans la purge des vieilles données, et réduire énormément sa durée mais aussi les écritures de journaux.

[2] https://docs.postgresql.fr/current/sql-altertable.html#SQL-ALTERTABLE-DETACH-PARTITION

1.5.13 Fonctions de gestion et vues système

- Sous psql : `\dP`
- `pg_partition_tree ('logs')` : liste entière des partitions
- `pg_partition_root ('logs_2019')` : racine d'une partition
- `pg_partition_ancestors ('logs_201901')` : parents d'une partition

Voici le jeu de tests pour l'exemple qui suivra. Il illustre également l'utilisation de sous-partitions (ici sur la même clé, mais cela n'a rien d'obligatoire).

```
-- Table partitionnée
CREATE TABLE logs (dreception timestamptz, contenu text) PARTITION BY
 ↪  RANGE(dreception);
-- Partition 2018, elle-même partitionnée
CREATE TABLE logs_2018 PARTITION OF logs FOR VALUES FROM ('2018-01-01') TO
 ↪  ('2019-01-01')
                       PARTITION BY range(dreception);
-- Sous-partitions 2018
CREATE TABLE logs_201801 PARTITION OF logs_2018 FOR VALUES FROM ('2018-01-01') TO
 ↪  ('2018-02-01');
CREATE TABLE logs_201802 PARTITION OF logs_2018 FOR VALUES FROM ('2018-02-01') TO
 ↪  ('2018-03-01');
…
-- Idem en 2019
CREATE TABLE logs_2019 PARTITION OF logs FOR VALUES FROM ('2019-01-01') TO
 ↪  ('2020-01-01')
                       PARTITION BY range(dreception);
CREATE TABLE logs_201901 PARTITION OF logs_2019 FOR VALUES FROM ('2019-01-01') TO
 ↪  ('2019-02-01');
…
```

Et voici le test des différentes fonctions :

```
SELECT pg_partition_root('logs_2019');

 pg_partition_root
-------------------
 logs

SELECT pg_partition_root('logs_201901');

 pg_partition_root
-------------------
 logs

SELECT pg_partition_ancestors('logs_2018');

 pg_partition_ancestors
------------------------
```

```
 logs_2018
 logs

SELECT pg_partition_ancestors('logs_201901');

 pg_partition_ancestors
------------------------
 logs_201901
 logs_2019
 logs

SELECT * FROM pg_partition_tree('logs');

    relid    | parentrelid | isleaf | level
-------------+-------------+--------+-------
 logs        |             | f      |     0
 logs_2018   | logs        | f      |     1
 logs_2019   | logs        | f      |     1
 logs_201801 | logs_2018   | t      |     2
 logs_201802 | logs_2018   | t      |     2
 logs_201901 | logs_2019   | t      |     2
```

Noter les propriétés de « feuille » (*leaf*) et le niveau de profondeur dans le partitionnement.

Sous psql, `\d` affichera toutes les tables, partitions comprises, ce qui peut vite encombrer l'affichage. `\dP` affiche uniquement les tables et index partitionnés :

```
=# \dP
               Liste des relations partitionnées
 Schéma |   Nom    | Propriétaire |        Type        |  Table
--------+----------+--------------+--------------------+----------
 public | logs     | postgres     | table partitionnée |
 public | t2       | postgres     | index partitionné  | bigtable
```

La table système `pg_partitioned_table`[3] permet des requêtes plus complexes. Le champ `pg_class.relpartbound`[4] contient les définitions des clés de partitionnement.

Pour masquer les partitions dans certains outils, il peut être intéressant de déclarer les partitions dans un schéma différent de la table principale.
Dans un cadre « multitenant » avec de nombreux schémas, et des partitions de même noms chacune dans son schéma, positionner `search_path` permet de sélectionner implicitement la partition, facilitant la vie au développeur ou permettant de « mentir » à l'application.

[3] https://www.postgresql.org/docs/current/catalog-pg-partitioned-table.html
[4] https://www.postgresql.org/docs/current/catalog-pg-class.html

1.5.14 Clé primaire et clé de partitionnement

La clé primaire doit contenir toutes les colonnes de la clé de partitionnement.

- Idem pour une contrainte unique
- Pas un problème si on partitionne selon la clé
- Plus gênant dans d'autres cas (date, statut…)

Le partitionnement impose une contrainte importante sur la modélisation : la clé de partitionnement doit impérativement faire partie de la clé primaire (ainsi que des contraintes et index uniques). En effet, PostgreSQL ne maintient pas d'index global couvrant toutes les partitions. Il ne peut donc garantir l'unicité d'un champ qu'au sein de chaque partition.

Dans beaucoup de cas cela ne posera pas de problème, notamment si on partitionne justement sur tout ou partie de cette clé primaire. Dans d'autres cas, c'est plus gênant. Si la vraie clé primaire est un identifiant géré par la base à l'insertion (`serial` ou `IDENTITY`), le risque reste limité. Mais avec des identifiants générés côté applicatif, il y a un risque d'introduire un doublon. Dans le cas où les valeurs de la clé de partitionnement ne sont pas une simple constante (par exemple des dates au lieu d'une seule année), le problème peut être mitigé en ajoutant une contrainte unique directement sur chaque partition, garantissant l'unicité de la clé primaire réelle au moins au sein de la partition.

Une solution générale est de créer une autre table non partitionnée avec la clé primaire réelle, et une contrainte vers cette table depuis la table partitionnée. Conceptuellement, cela est équivalent à ne pas partitionner une grosse table mais à en « sortir » les données dans une sous-table partitionnée portant une contrainte.

Exemple :

```
-- On voudrait partitionner ainsi mais le moteur refuse
CREATE TABLE factures_p
                  (id bigint PRIMARY KEY,
                   d timestamptz,
                   id_client int NOT NULL,
                   montant_c int NOT NULL DEFAULT 0)
PARTITION BY RANGE (d);

ERROR:  unique constraint on partitioned table must include all partitioning columns
DÉTAIL : PRIMARY KEY constraint on table "factures_p" lacks column "d" which is part
 ↪  of the partition key.

-- On se rabat sur une clé primaire incluant la date
CREATE TABLE factures_p
                  (id bigint NOT NULL,
                   d timestamptz NOT NULL,
                   id_client int,
                   montant_c int,
                   PRIMARY KEY (id, d)
                   )
PARTITION BY RANGE (d);
```

```
CREATE TABLE factures_p_202310 PARTITION OF factures_p
  FOR VALUES FROM ('2023-10-01') TO ('2023-11-01');
CREATE TABLE factures_p_202311 PARTITION OF factures_p
  FOR VALUES FROM ('2023-11-01') TO ('2023-12-01');
ALTER TABLE factures_p_202310 ADD CONSTRAINT factures_p_202310_uq UNIQUE (id);
-- Ces contraintes sécurisent les clés primaire au niveau partition
ALTER TABLE factures_p_202311 ADD CONSTRAINT factures_p_202311_uq UNIQUE (id);
-- Ajout de quelques lignes de 1 à 5 sur les deux partitions
INSERT INTO factures_p (id, d, id_client)
SELECT i, '2023-10-26'::timestamptz+i*interval '2 days', 42 FROM generate_series
 ↪  (1,5) i;

BEGIN ;
-- Ce doublon est accepté car les deux valeurs 3 ne sont pas dans la même partition
INSERT INTO factures_p (id, d, id_client)
SELECT 3, '2023-11-01'::timestamptz-interval '1s', 42 ;
-- Vérification que 3 est en double
SELECT tableoid::regclass, id, d FROM factures_p ORDER BY id ;
ROLLBACK ;

-- Cette table permet de garantir l'unicité dans toutes les partitions
CREATE TABLE factures_ref (id bigint NOT NULL PRIMARY KEY,
                      d timestamptz NOT NULL,
                      UNIQUE (id,d) -- nécessaire pour la contrainte
                      )  ;

INSERT INTO factures_ref SELECT id,d FROM factures_p ;

-- Contrainte depuis la table partitionnée
ALTER TABLE factures_p ADD CONSTRAINT factures_p_id_fk
FOREIGN KEY (id, d)  REFERENCES factures_ref (id,d);

-- Par la suite, il faut insérer chaque nouvelle valeur de `id`
-- dans les deux tables
-- Ce doublon est à présent correctement rejeté :
WITH ins AS (
    INSERT INTO factures_p (id, d, id_client)
    SELECT 3, '2023-11-01'::timestamptz-interval '1s', 42
    RETURNING id,d )
INSERT INTO factures_ref
SELECT id, d FROM ins ;
```

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

1.5.15 Indexation

- Propagation automatique
- Index supplémentaires par partition possibles
- Clés étrangères entre tables partitionnées

Les index sont propagés de la table mère aux partitions : tout index créé sur la table partitionnée sera automatiquement créé sur les partitions existantes. Toute nouvelle partition disposera des index de la table partitionnée. La suppression d'un index se fait sur la table partitionnée et concernera toutes les partitions. Il n'est pas possible de supprimer un tel index d'une seule partition.

Gérer des index manuellement sur certaines partitions est possible. Par exemple, on peut n'avoir besoin de certains index que sur les partitions de données récentes, et ne pas les créer sur des partitions de données d'archives.

Une clé primaire ou unique peut exister sur une table partitionnée (mais elle devra contenir toutes les colonnes de la clé de partitionnement) ; ainsi qu'une clé étrangère d'une table partitionnée *vers* une table normale.

Depuis PostgreSQL 12, il est possible de créer une clé étrangère *vers* une table partitionnée de la même manière qu'entre deux tables normales. Par exemple, si les tables `ventes` et `lignes_ventes` sont toutes deux partitionnées :

```
ALTER TABLE lignes_ventes
ADD CONSTRAINT lignes_ventes_ventes_fk
FOREIGN KEY (vente_id) REFERENCES ventes (vente_id) ;
```

Noter que les versions 10 et 11 possèdent des limites sur ces fonctionnalités, que l'on peut souvent contourner en créant index et contraintes manuellement sur chaque partition.

1.5.16 Planification & performances

- Mettre la clé dans la requête autant que possible
- ou : cibler les partitions directement
- Temps de planification
  - nombre de tables, d'index, leurs statistisques…
  - max ~ 100 partitions
- À activer ?
  - `enable_partitionwise_aggregate`
  - `enable_partitionwise_join`

Cibler la partition par la clé :

Si la clé de partitionnement n'est pas fournie, l'exécution concernera toutes les partitions, qu'elles soient accédées par *Seq Scan* ou *Index Scan*.

Autant que possible, le développeur doit fournir la clé de partitionnement dans chaque requête, et le plan ciblera directement la bonne partition.

Comme avec les index, il faut vérifier que la clé est bien claire pour PostgreSQL. Si ce n'est pas le cas toutes les partitions seront lues :

```
EXPLAIN SELECT * FROM pgbench_accounts WHERE aid + 0 = 123 LIMIT 1 ;
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Limit  (cost=0.00..6.29 rows=1 width=97)
   ->  Append  (cost=0.00..314250.00 rows=50000 width=97)
         ->  Seq Scan on pgbench_accounts_1  (cost=0.00..3140.00 rows=500 width=97)
               Filter: ((aid + 0) = 123)
         ->  Seq Scan on pgbench_accounts_2  (cost=0.00..3140.00 rows=500 width=97)
               Filter: ((aid + 0) = 123)
        …
        …
               Filter: ((aid + 0) = 123)
         ->  Seq Scan on pgbench_accounts_99  (cost=0.00..3140.00 rows=500 width=97)
               Filter: ((aid + 0) = 123)
         ->  Seq Scan on pgbench_accounts_100  (cost=0.00..3140.00 rows=500 width=97)
               Filter: ((aid + 0) = 123)
```

alors que si PostgreSQL reconnaît la clé de partitionnement :

```
EXPLAIN SELECT * FROM pgbench_accounts WHERE aid = 123 LIMIT 1 ;

                                    QUERY PLAN
-------------------------------------------------------------------------------
 Limit  (cost=0.29..8.31 rows=1 width=97)
   ->  Index Scan using pgbench_accounts_1_pkey on pgbench_accounts_1
↪  pgbench_accounts  (cost=0.29..8.31 rows=1 width=97)
         Index Cond: (aid = 123)
```

Partition pruning :

Dans le cas où la clé de partitionnement dépend du résultat d'un calcul, d'une sous-requête ou d'une jointure, PostgreSQL prévoit un plan concernant toutes les partitions, mais élaguera à l'exécution les appels aux partitions non concernées. Ci-dessous, seule `pgbench_accounts_8` est interrogé (et ce peut être une autre partition si l'on répète la requête) :

```
EXPLAIN (ANALYZE,COSTS OFF)
SELECT * FROM pgbench_accounts WHERE aid = (SELECT (random()*1000000)::int )  ;

                                    QUERY PLAN
-------------------------------------------------------------------------------
 Append (actual time=23.083..23.101 rows=1 loops=1)
   InitPlan 1 (returns $0)
     ->  Result (actual time=0.001..0.002 rows=1 loops=1)
   ->  Index Scan using pgbench_accounts_1_pkey on pgbench_accounts_1 (never
↪  executed)
         Index Cond: (aid = $0)
   ->  Index Scan using pgbench_accounts_2_pkey on pgbench_accounts_2 (never
↪  executed)
         Index Cond: (aid = $0)
   ->  Index Scan using pgbench_accounts_3_pkey on pgbench_accounts_3 (never
↪  executed)
         Index Cond: (aid = $0)
   ->  Index Scan using pgbench_accounts_4_pkey on pgbench_accounts_4 (never
↪  executed)
         Index Cond: (aid = $0)
   ->  Index Scan using pgbench_accounts_5_pkey on pgbench_accounts_5 (never
↪  executed)
         Index Cond: (aid = $0)
   ->  Index Scan using pgbench_accounts_6_pkey on pgbench_accounts_6 (never
↪  executed)
         Index Cond: (aid = $0)
   ->  Index Scan using pgbench_accounts_7_pkey on pgbench_accounts_7 (never
↪  executed)
         Index Cond: (aid = $0)
   ->  Index Scan using pgbench_accounts_8_pkey on pgbench_accounts_8 (actual
↪  time=23.077..23.080 rows=1 loops=1)
         Index Cond: (aid = $0)
   ->  Index Scan using pgbench_accounts_9_pkey on pgbench_accounts_9 (never
↪  executed)
         Index Cond: (aid = $0)
   …
   …
   ->  Index Scan using pgbench_accounts_100_pkey on pgbench_accounts_100 (never
↪  executed)
         Index Cond: (aid = $0)
```

```
 Planning Time: 1.118 ms
 Execution Time: 23.341 ms
```

Temps de planification :

Une limitation sérieuse du partitionnement tient au temps de planification qui augmente très vite avec le nombre de partitions, même petites. En effet, chaque partition ajoute ses statistiques et souvent plusieurs index aux tables système. Par exemple, dans le cas le plus défavorable d'une session qui démarre :

```
-- Base pgbench de taille 100, non partitionnée
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)
SELECT * FROM pgbench_accounts WHERE aid = 123 LIMIT 1 ;

                                  QUERY PLAN
------------------------------------------------------------------------------
 Limit (actual time=0.021..0.022 rows=1 loops=1)
   Buffers: shared hit=4
   ->  Index Scan using pgbench_accounts_pkey on pgbench_accounts (actual
↪  time=0.021..0.021 rows=1 loops=1)
         Index Cond: (aid = 123)
         Buffers: shared hit=4
 Planning:
   Buffers: shared hit=70
 Planning Time: 0.358 ms
 Execution Time: 0.063 ms

-- Base pgbench de taille 100, partitionnée en 100 partitions
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)
SELECT * FROM pgbench_accounts WHERE aid = 123 LIMIT 1 ;

                                  QUERY PLAN
------------------------------------------------------------------------------
 Limit (actual time=0.015..0.016 rows=1 loops=1)
   Buffers: shared hit=3
   ->  Index Scan using pgbench_accounts_1_pkey on pgbench_accounts_1
↪  pgbench_accounts (actual time=0.015..0.015 rows=1 loops=1)
         Index Cond: (aid = 123)
         Buffers: shared hit=3
 Planning:
   Buffers: shared hit=423
 Planning Time: 1.030 ms
 Execution Time: 0.061 ms
```

La section `Planning` indique le nombre de blocs qu'une session qui démarre doit mettre en cache, liés notamment aux tables systèmes et statistiques (ce phénomène est encore une raison d'éviter des sessions trop courtes). Dans cet exemple, sur la base partitionnée, il y a presque six fois plus de ces blocs, et on triple le temps de planification, qui reste raisonnable.

En général, on considère qu'il ne faut pas dépasser 100 partitions si l'on ne veut pas pénaliser les transactions courtes. Les dernières versions de PostgreSQL sont cependant meilleures sur ce point.

Ce problème de planification est moins gênant pour les requêtes longues (analytiques).

Pour contourner cette limite, il est possible d'utiliser directement les partitions, s'il est facile pour le développeur (ou le générateur de code…) de trouver leur nom, en plus de toujours fournir la clé. Interroger directement une partition est en effet aussi rapide à planifier qu'interroger une table monolithique :

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)
SELECT * FROM pgbench_accounts_1 WHERE aid = 123 LIMIT 1 ;
```

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Limit (actual time=0.006..0.007 rows=1 loops=1)
   Buffers: shared hit=3
   ->  Index Scan using pgbench_accounts_1_pkey on pgbench_accounts_1 (actual
↪  time=0.006..0.006 rows=1 loops=1)
         Index Cond: (aid = 123)
         Buffers: shared hit=3
 Planning Time: 0.046 ms
 Execution Time: 0.016 ms
```

Utiliser directement les partitions est particulièrement économe si leur nombre est grand, mais on perd alors le côté « transparent » du partitionnement, et on augmente la complexité du code applicatif.

Paramètres « partitionwise » :

Dans des cas plus complexes, notamment en cas de jointure entre tables partitionnées, le temps de planification peut exploser. Par exemple, pour la requête suivante où la table partitionnée est jointe à elle-même, le plan sur la table non partitionnée, cache de session chaud, renvoie :

```
EXPLAIN (BUFFERS, COSTS OFF, SUMMARY ON) SELECT *
FROM pgbench_accounts a INNER JOIN pgbench_accounts b USING (aid)
WHERE a.bid = 55 ;
```

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Gather
   Workers Planned: 4
   ->  Nested Loop
         ->  Parallel Seq Scan on pgbench_accounts a
               Filter: (bid = 55)
         ->  Index Scan using pgbench_accounts_pkey on pgbench_accounts b
               Index Cond: (aid = a.aid)
 Planning:
   Buffers: shared hit=16
 Planning Time: 0.168 ms
```

Avec cent partitions, le temps de planification est ici multiplié par 50 :

```
 Gather
   Workers Planned: 4
   ->  Parallel Hash Join
         Hash Cond: (b.aid = a.aid)
         ->  Parallel Append
               ->  Parallel Seq Scan on pgbench_accounts_1 b_1
```

```
               ->  Parallel Seq Scan on pgbench_accounts_2 b_2
               …
               ->  Parallel Seq Scan on pgbench_accounts_99 b_99
               ->  Parallel Seq Scan on pgbench_accounts_100 b_100
         ->  Parallel Hash
               ->  Parallel Append
                     ->  Parallel Seq Scan on pgbench_accounts_1 a_1
                           Filter: (bid = 55)
                     ->  Parallel Seq Scan on pgbench_accounts_2 a_2
                           Filter: (bid = 55)
                     …
                     ->  Parallel Seq Scan on pgbench_accounts_99 a_99
                           Filter: (bid = 55)
                     ->  Parallel Seq Scan on pgbench_accounts_100 a_100
                           Filter: (bid = 55)
 Planning Time: 5.513 ms
```

Ce plan est perfectible : il récupère tout `pgbench_accounts` et le joint à toute la table. Il serait plus intelligent de travailler partition par partition puisque la clé de jointure est celle de partitionnement. Pour que PostgreSQL cherche à faire ce genre de chose, un paramètre doit être activé :

```
SET enable_partitionwise_join TO on ;
```

Les jointures se font alors entre partitions :

```
Gather
   Workers Planned: 4
   ->  Parallel Append
         ->  Parallel Hash Join
               Hash Cond: (a_55.aid = b_55.aid)
               ->  Parallel Seq Scan on pgbench_accounts_55 a_55
                     Filter: (bid = 55)
               ->  Parallel Hash
                     ->  Parallel Seq Scan on pgbench_accounts_55 b_55
…
         ->  Nested Loop
               ->  Parallel Seq Scan on pgbench_accounts_100 a_100
                     Filter: (bid = 55)
              ->  Index Scan using pgbench_accounts_100_pkey on pgbench_accounts_100
 ↪  b_100
                     Index Cond: (aid = a_100.aid)
 Planning:
   Buffers: shared hit=1200
 Planning Time: 12.449 ms
```

Le temps d'exécution passe de 1,2 à 0,2 s, ce qui justifie les quelques millisecondes perdues en plus en planification.

Un autre paramètre est à activer si des agrégations sur plusieurs partitions sont à faire :

```
SET enable_partitionwise_aggregate TO on ;
```

`enable_partitionwise_aggregate` et `enable_partitionwise_join` sont désactivés par défaut à cause de leur coût en planification sur les petites requêtes, mais les activer est souvent rentable. Avec `SET`, cela peut se décider requête par requête.

1.5.17 Opérations de maintenance

- Changement de tablespace
- autovacuum/analyze
  - sur les partitions comme sur toute table
- `VACUUM`, `VACUUM FULL`, `ANALYZE`
  - sur table mère : redescendent sur les partitions
- `REINDEX`
  - avant v14 : uniquement par partition
- `ANALYZE`
  - prévoir aussi sur la table mère (manuellement…)

Les opérations de maintenance profitent grandement du fait de pouvoir scinder les opérations en autant d'étapes qu'il y a de partitions. Des données « froides » peuvent être déplacées dans un autre tablespace sur des disques moins chers, partition par partition, ce qui est impossible avec une table monolithique :

```
ALTER TABLE pgbench_accounts_8 SET TABLESPACE hdd ;
```

L'autovacuum et l'autoanalyze fonctionnent normalement et indépendamment sur chaque partition, comme sur les tables classiques. Ainsi ils peuvent se déclencher plus souvent sur les partitions actives. Par rapport à une grosse table monolithique, il y a moins souvent besoin de régler l'autovacuum.

Les ordres `ANALYZE` et `VACUUM` peuvent être effectués sur une partition, mais aussi sur la table partitionnée, auquel cas l'ordre redescendra en cascade sur les partitions (l'option `VERBOSE` permet de le vérifier). Les statistiques seront calculées par partition, donc plus précises.

Reconstruire une table partitionnée avec `VACUUM FULL` se fera généralement partition par partition. Le partitionnement permet ainsi de résoudre les cas où le verrou sur une table monolithique serait trop long, ou l'espace disque total serait insuffisant.

Noter cependant ces spécificités sur les tables partitionnées :

REINDEX :

À partir de PostgreSQL 14, un `REINDEX` sur la table partitionnée réindexe toutes les partitions automatiquement. Dans les versions précédentes, il faut réindexer partition par partition.

ANALYZE :

L'autovacuum ne crée pas spontanément de statistiques sur les données pour la table partitionnée dans son ensemble, mais uniquement partition par partition. Pour obtenir des statistiques sur toute la table partitionnée, il faut exécuter manuellement :

```
ANALYZE table_partitionnée ;
```

1.5.18 Sauvegardes

Sauvegarde physique : peu de différence
Avec `pg_dump` :

- `--jobs` : efficace
- `--load-via-partition-root`
- exclusion de partitions (v16+) :
  - `--table-and-children`
  - `--exclude-table-and-children`
  - `--exclude-table-data-and-children`

Grâce au partitionnement, un export par `pg_dump --jobs` devient efficace puisque plusieurs partitions peuvent être sauvegardées en parallèle.

La parallélisation peut être aussi un peu meilleure avec un outil de sauvegarde physique (comme pgBackRest ou Barman), qui parallélise les copies de fichiers, mais les grosses tables non partitionnées étaient de toute façon déjà découpées en fichier de 1 Go.

`pg_dump` a des options pour gérer l'export des tables partitionnées :

`--load-via-partition-root` permet de générer des ordres `COPY` ciblant la table mère et non la partition. Ce peut être pratique pour restaurer les données dans une base où la table est partitionnée séparément.

À partir de PostgreSQL 16, n'exporter qu'une table partitionnée se fait avec `--table-and-children` (et non `--table`/`-t` qui ne concernerait que la table mère). Exclure des tables partitionnées se fait avec `--exclude-table-and-children` (et non `--exclude-table`/`-T`). Pour exclure uniquement les données d'une table partitionnée en gardant sa structure, on utilisera `--exclude-table-data-and-children`. Ces trois options acceptent un motif (par exemple : `pgbench_accounts_*`) et peuvent être répétées dans la commande.

1.5.19 Limitations du partitionnement déclaratif et versions

- Pas de création automatique des partitions
- Planification : 100 partitions max conseillé
- Pas d'héritage multiple, schéma fixe
- Partitions distantes : sans propagation d'index
- PostgreSQL >= 13 conseillée !
  - limitations sur versions précédentes
  - contournement : travailler par partition

Une table partitionnée ne peut être convertie en table classique, ni vice-versa. (Par contre, une table classique peut être attachée comme partition, ou une partition détachée).

Les partitions ont forcément le même schéma de données que leur partition mère.

Leur création n'est pas automatisée : il faut les créer par avance manuellement ou par script planifié, et éventuellement prévoir une partition par défaut pour les cas qui ont pu être oubliés.

Les clés de partition ne doivent pas se recouvrir. Les contraintes ne peuvent s'exercer qu'au sein d'une même partition : les clés d'unicité doivent donc inclure toute la clé de partitionnement, les contraintes d'exclusion ne peuvent vérifier toutes les partitions.

Il n'y a pas de notion d'héritage multiple.

Éviter d'avoir trop de partitions, pour limiter les risques de dérapage du temps de planification. Si possible, cibler les requêtes directement sur les partitions qui les intéressent.

L'ordre `CLUSTER`, pour réécrire une table dans l'ordre d'un index donné, ne fonctionne pour les tables partitionnées qu'à partir de PostgreSQL 15. Il peut toutefois être exécuté manuellement table par table.

Un `TRUNCATE` d'une table distante n'est pas possible avant PostgreSQL 14.

Il est possible d'attacher comme partitions des tables distantes, généralement déclarées avec `postgres_fdw` ; cependant la propagation d'index ne fonctionnera pas sur ces tables. Il faudra les créer manuellement sur les instances distantes. (Restriction supplémentaire en version 10 : les partitions distantes ne sont accessibles qu'en lecture, si accédées *via* la table mère.)

Les partitions par défaut n'existent pas en version 10.

Les limitations sur les index et clés primaires et étrangères avant la version 12 ont été évoquées plus haut.

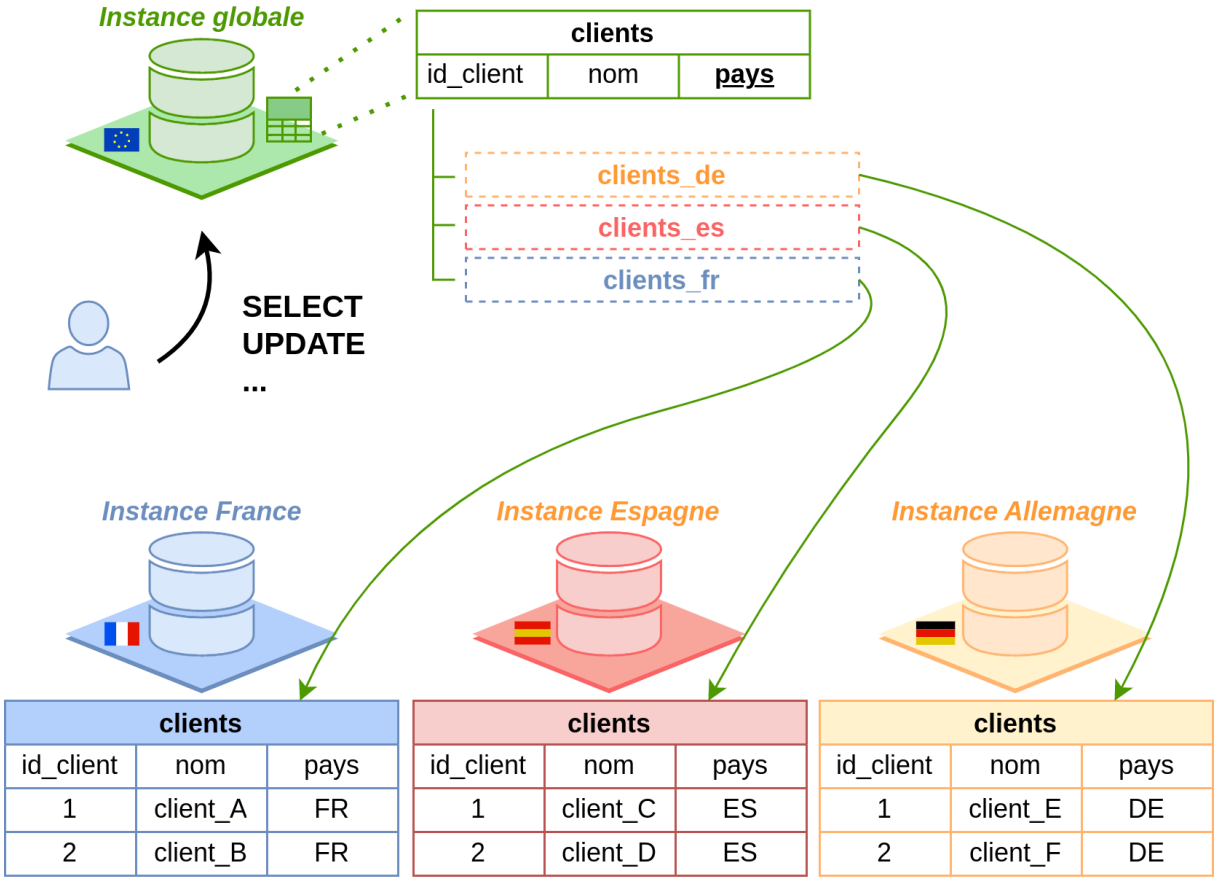
Les triggers de lignes ne se propagent pas en version 10. En v11, on peut créer des triggers `AFTER UPDATE … FOR EACH ROW`, mais les `BEFORE UPDATE … FOR EACH ROW` ne peuvent toujours pas être créés sur la table mère. Il reste là encore la possibilité de les créer partition par partition au

besoin. À partir de la version 13, les triggers `BEFORE UPDATE … FOR EACH ROW` sont possibles, mais il ne permettent pas de modifier la partition de destination.

Enfin, la version 10 ne permet pas de faire une mise à jour (`UPDATE`) d'une ligne où la clé de partitionnement est modifiée de telle façon que la ligne doit changer de partition. Il faut faire un `DELETE` et un `INSERT` à la place.

On constate que des limitations évoquées plus haut dépendent des versions de PostgreSQL. Si le partitionnement vous intéresse, il est conseillé d'utiliser une version la plus récente possible, au moins PostgreSQL 13.

1.6 TABLES DISTANTES & SHARDING



- Tables distantes comme partitions : sharding
- (v14+) Interrogation simultanée asynchrone

Il est possible d'attacher comme partitions des tables distantes (situées sur d'autres serveurs), généralement déclarées avec le *Foreign Data Wrapper* `postgres_fdw`.

NB : Dans le reste de ce chapitre, nous nommerons **table étrangère** (*foreign table* dans la documentation officielle) l'objet qui sert d'interface pour accéder, depuis l'instance locale, à la **table distante** (*remote table*), qui contient réellement les données.

Par exemple, si trois instances en France, Allemagne et Espagne possèdent chacune des tables `clients` et `commandes` ne contenant que les données de leur pays, on peut créer une autre instance utilisant des tables étrangères pour accéder aux trois tables, chaque table étrangère étant une partition d'une table partitionnée de cette instance européene.

Pour les instances nationales, cette instance européenne n'est qu'un client comme un autre, qui en-

voie des requêtes, et ouvre parfois des curseurs (fonctionnement normal de `postgres_fdw`). Si le pays est précisé dans une requête, la bonne partition est ciblée, et l'instance européenne n'interroge qu'une seule instance nationale.

La maquette suivante donne une idée du fonctionnement :

```
-- Maquette rapide sous psql de sharding
-- avec trois bases demosharding_fr , _de, _es
-- et une base globale pour le requêtage

\set timing off
\set ECHO all
\set ON_ERROR_STOP 1

\connect postgres postgres serveur1
DROP DATABASE IF EXISTS demosharding_fr ;
CREATE DATABASE demosharding_fr ;
ALTER DATABASE demosharding_fr SET log_min_duration_statement TO 0 ;
\connect postgres postgres serveur2
DROP DATABASE IF EXISTS demosharding_de ;
CREATE DATABASE demosharding_de ;
ALTER DATABASE demosharding_de SET log_min_duration_statement TO 0 ;
\connect postgres postgres serveur3
DROP DATABASE IF EXISTS demosharding_es ;
CREATE DATABASE demosharding_es ;
ALTER DATABASE demosharding_es SET log_min_duration_statement TO 0 ;

\connect postgres postgres serveur4
DROP DATABASE IF EXISTS demosharding_global ;
CREATE DATABASE demosharding_global ;
ALTER DATABASE demosharding_global SET log_min_duration_statement TO 0 ;

-- Tables identiques sur chaque serveur
\connect demosharding_fr postgres serveur1

CREATE TABLE clients    (id_client  int GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
                         nom         text,
                         pays        char (2) DEFAULT 'FR' CHECK (pays = 'FR')
                         ) ;
CREATE TABLE commandes (id_commande  bigint GENERATED ALWAYS AS IDENTITY  PRIMARY
 ↪  KEY,
                         pays         char (2) DEFAULT 'FR'  CHECK (pays = 'FR'),
                         id_client    int REFERENCES clients ,
                         montant      float
                         );

\connect demosharding_de postgres serveur2

CREATE TABLE clients    (id_client  int GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
                         nom         text,
                         pays        char (2) DEFAULT 'DE'  CHECK (pays = 'DE')
                         ) ;
CREATE TABLE commandes (id_commande  bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
                         pays         char (2) DEFAULT 'DE'  CHECK (pays = 'DE') ,
                         id_client    int  REFERENCES clients,
```

```
                        montant      float
                        );

\connect demosharding_es postgres serveur3

CREATE TABLE clients    (id_client  int GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
                        nom         text,
                        pays        char (2) DEFAULT 'ES'  CHECK (pays = 'ES')
                        ) ;
CREATE TABLE commandes (id_commande  bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
                        pays         char (2) DEFAULT 'ES'  CHECK (pays = 'ES'),
                        id_client    int REFERENCES clients ,
                        montant      float
                        );

-- Tables partitionnées globales
\connect demosharding_global postgres serveur4

CREATE TABLE clients (id_client int, nom text, pays char(2))
PARTITION BY LIST (pays);
CREATE TABLE commandes (id_commande bigint, pays char(2),
                        id_client int, montant float)
PARTITION BY LIST (pays);

-- Serveurs distants (adapter les chaines de connexion)
-- NB : l'option async_capable n existe pas avant PostgreSQL 14
CREATE EXTENSION postgres_fdw ;

CREATE SERVER dist_fr
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (host 'localhost', dbname 'demosharding_fr', port '16001',
async_capable 'on', fetch_size '10000') ;

CREATE SERVER dist_de
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (host 'localhost', dbname 'demosharding_de', port '16001',
async_capable 'on', fetch_size '10000') ;

CREATE SERVER dist_es
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (host 'localhost', dbname 'demosharding_es', port '16001',
async_capable 'on', fetch_size '10000') ;

CREATE USER MAPPING IF NOT EXISTS FOR current_user SERVER dist_fr ;
CREATE USER MAPPING IF NOT EXISTS FOR current_user SERVER dist_de ;
CREATE USER MAPPING IF NOT EXISTS FOR current_user SERVER dist_es ;

-- Les partitions distantes
CREATE FOREIGN TABLE clients_fr PARTITION OF clients FOR VALUES IN ('FR')
SERVER dist_fr OPTIONS (table_name 'clients') ;
CREATE FOREIGN TABLE clients_de PARTITION OF clients FOR VALUES IN ('DE')
SERVER dist_de OPTIONS (table_name 'clients') ;
CREATE FOREIGN TABLE clients_es PARTITION OF clients FOR VALUES IN ('ES')
SERVER dist_es OPTIONS (table_name 'clients') ;

CREATE FOREIGN TABLE commandes_fr PARTITION OF commandes FOR VALUES IN ('FR')
```

```
SERVER dist_fr OPTIONS (table_name 'commandes') ;
CREATE FOREIGN TABLE commandes_de PARTITION OF commandes FOR VALUES IN ('DE')
SERVER dist_de OPTIONS (table_name 'commandes') ;
CREATE FOREIGN TABLE commandes_es PARTITION OF commandes FOR VALUES IN ('ES')
SERVER dist_es OPTIONS (table_name 'commandes') ;

-- Alimentations des pays (séparément)

\connect demosharding_fr postgres serveur1

WITH ins_clients AS (
    INSERT INTO clients (nom)
    SELECT md5 (random()::text) FROM generate_series (1,10) i
    WHERE random()<0.8
RETURNING id_client
),
ins_commandes AS (
    INSERT INTO commandes (id_client, montant)
    SELECT c.id_client, random()*j::float
    FROM ins_clients c CROSS JOIN generate_series (1,100000) j
    WHERE random()<0.8
RETURNING *
)
SELECT count(*) FROM  ins_commandes ;

\connect demosharding_de postgres serveur2

\g

\connect demosharding_es postgres serveur3

\g

\connect demosharding_global postgres serveur4

-- Les ANALYZE redescendent sur les partitions
ANALYZE (VERBOSE) clients, commandes ;

-- Pour un plan optimal
SET enable_partitionwise_join TO on ;
SET enable_partitionwise_aggregate TO on ;

-- Requête globale : top 8 des clients
-- Plan disponible sur https://explain.dalibo.com/plan/27f964651518a65g

SELECT pays,
       nom,
       count(DISTINCT id_commande) AS nb_commandes,
       avg(montant) AS montant_avg_commande,
       sum(montant) AS montant_sum
FROM
    commandes INNER JOIN clients USING (pays, id_client)
GROUP BY 1,2
ORDER BY montant_sum DESC
LIMIT 8 ;
```

Une nouveauté de PostgreSQL 14 est ici particulièrement intéressante : l'option `async_capable` du serveur étranger (éventuellement de la table) peut être passée à `on` (le défaut est `off`). Les nœuds *Foreign Scan* typiques des accès distants sont alors remplacés par des nœuds *Async Foreign Scan* (asynchrones), et le serveur principal interroge alors simultanément les trois serveurs qui lui renvoient les données. Dans cet extrait des traces, les ordres `FETCH` sont entremêlés :

```
…user=postgres,db=demosharding_de,app=postgres_fdw,client=::1
LOG:  duration: 0.384 ms  execute <unnamed>: DECLARE c1 CURSOR FOR
        SELECT id_commande, pays, id_client, montant FROM public.commandes
…user=postgres,db=demosharding_es,app=postgres_fdw,client=::1
LOG:  duration: 0.314 ms  execute <unnamed>: DECLARE c2 CURSOR FOR
        SELECT id_commande, pays, id_client, montant FROM public.commandes
…user=postgres,db=demosharding_fr,app=postgres_fdw,client=::1
LOG:  duration: 0.374 ms  execute <unnamed>: DECLARE c3 CURSOR FOR
        SELECT id_commande, pays, id_client, montant FROM public.commandes
…user=postgres,db=demosharding_de,app=postgres_fdw,client=::1
LOG:  duration: 6.081 ms  statement: FETCH 10000 FROM c1
…user=postgres,db=demosharding_es,app=postgres_fdw,client=::1
LOG:  duration: 5.878 ms  statement: FETCH 10000 FROM c2
…user=postgres,db=demosharding_fr,app=postgres_fdw,client=::1
LOG:  duration: 6.263 ms  statement: FETCH 10000 FROM c3
…user=postgres,db=demosharding_de,app=postgres_fdw,client=::1
LOG:  duration: 2.418 ms  statement: FETCH 10000 FROM c1
…user=postgres,db=demosharding_de,app=postgres_fdw,client=::1
LOG:  duration: 2.397 ms  statement: FETCH 10000 FROM c1
…user=postgres,db=demosharding_es,app=postgres_fdw,client=::1
LOG:  duration: 2.423 ms  statement: FETCH 10000 FROM c2
…user=postgres,db=demosharding_fr,app=postgres_fdw,client=::1
LOG:  duration: 4.381 ms  statement: FETCH 10000 FROM c3
```

Dans cette configuration, activer les paramètres `enable_partitionwise_join` et `enable_partitionwise_aggregat` est particulièrement important. Dans l'idéal, comme dans le plan suivant (voir la version complète[5]), ces paramètres permettent que les agrégations et les jointures soient « poussées » au niveau du nœud (et calculées directement sur les serveurs distants) :

```
->  Append  (cost=6105.41..110039.12 rows=21 width=60) (actual time=330.325..594.923
 ↪  rows=21 loops=1)
          ->  Async Foreign Scan  (cost=6105.41..28384.99 rows=6 width=60) (actual
 ↪  time=2.284..2.286 rows=6 loops=1)
                Output: commandes.pays, clients.nom, (count(DISTINCT
 ↪  commandes.id_commande)), (avg(commandes.montant)), (sum(commandes.montant))
                Relations: Aggregate on ((public.commandes_de commandes) INNER JOIN
 ↪  (public.clients_de clients))
                Remote SQL: SELECT r4.pays, r7.nom, count(DISTINCT r4.id_commande),
 ↪  avg(r4.montant), sum(r4.montant) FROM (public.commandes r4 INNER JOIN
 ↪  public.clients r7 ON (((r4.pays = r7.pays)) AND ((r4.id_client =
 ↪  r7.id_client)))) GROUP BY 1, 2
          ->  Async Foreign Scan  (cost=6098.84..28353.37 rows=6 width=60) (actual
 ↪  time=2.077..2.078 rows=6 loops=1)
```

[5] https://explain.dalibo.com/plan/27f964651518a65g#raw

```
                Output: commandes_1.pays, clients_1.nom, (count(DISTINCT
↪ commandes_1.id_commande)), (avg(commandes_1.montant)),
↪ (sum(commandes_1.montant))
                Relations: Aggregate on ((public.commandes_es commandes_1) INNER
↪ JOIN (public.clients_es clients_1))
              Remote SQL: SELECT r5.pays, r8.nom, count(DISTINCT r5.id_commande),
↪ avg(r5.montant), sum(r5.montant) FROM (public.commandes r5 INNER JOIN
↪ public.clients r8 ON (((r5.pays = r8.pays)) AND ((r5.id_client =
↪ r8.id_client)))) GROUP BY 1, 2
        ->  Async Foreign Scan  (cost=9102.09..53300.65 rows=9 width=60) (actual
↪ time=2.189..2.190 rows=9 loops=1)
                Output: commandes_2.pays, clients_2.nom, (count(DISTINCT
↪ commandes_2.id_commande)), (avg(commandes_2.montant)),
↪ (sum(commandes_2.montant))
                Relations: Aggregate on ((public.commandes_fr commandes_2) INNER
↪ JOIN (public.clients_fr clients_2))
              Remote SQL: SELECT r6.pays, r9.nom, count(DISTINCT r6.id_commande),
↪ avg(r6.montant), sum(r6.montant) FROM (public.commandes r6 INNER JOIN
↪ public.clients r9 ON (((r6.pays = r9.pays)) AND ((r6.id_client =
↪ r9.id_client)))) GROUP BY 1, 2
```

Quand on utilise les tables étrangères, il est conseillée d'utiliser `EXPLAIN (VERBOSE)`, pour afficher les requêtes envoyées aux serveurs distants et vérifier que le minimum de volumétrie transite sur le réseau.

Cet exemple est une version un peu primitive de *sharding*, à réserver aux cas où les données sont clairement séparées. L'administration d'une configuration « multimaître » peut devenir compliquée : cohérence des différents schémas et des contraintes sur chaque instance, copie des tables de référence communes, risques de recouvrement des clés primaires entre bases, gestion des indisponibilités, sauvegardes cohérentes…

Noter que l'utilisation de partitions distantes rend impossible notamment la gestion automatique des index, il faut retourner à une manipulation table par table.

1.7 EXTENSIONS & OUTILS

- Extension pg_partman[6]
  - automatisation
- Extensions dédiées à un domaine :
  - timescaledb
  - citus

L'extension pg_partman[7], de Crunchy Data, est un complément aux systèmes de partitionnement de PostgreSQL. Elle est apparue d'abord pour automatiser le partitionnement par héritage. Elle peut être utile avec le partitionnement déclaratif, pour simplifier la maintenance d'un partitionnement sur une échelle temporelle ou de valeurs (par *range*).

PostgresPro proposait un outil nommé pg_pathman[8], à présent déprécié en faveur du partitionnement déclaratif intégré à PostgreSQL.

timescaledb est une extension spécialisée dans les séries temporelles. Basée sur le partitionnement par héritage, elle vaut surtout pour sa technique de compression et ses utilitaires. La version communautaire sur Github[9] ne comprend pas tout ce qu'offre la version commerciale.

citus[10] est une autre extension commerciale. Le principe est de partitionner agressivement les tables sur plusieurs instances, et d'utiliser simultanément les processeurs, disques de toutes ces instances (*sharding*). Citus gère la distribution des requêtes, mais pas la maintenance des instances PostgreSQL supplémentaires. L'éditeur Citusdata a été racheté par Microsoft, qui le propose à présent dans Azure. En 2022, l'entièreté du code est passée sous licence libre[11]. Le gain de performance peut être impressionnant, mais attention : certaines requêtes se prêtent très mal au *sharding*.

[7] https://github.com/pgpartman/pg_partman
[8] https://github.com/postgrespro/pg_pathman
[9] https://github.com/timescale/timescaledb
[10] https://github.com/citusdata/citus
[11] https://www.citusdata.com/blog/2022/06/17/citus-11-goes-fully-open-source/

1.8 CONCLUSION

Le partitionnement déclaratif a de gros avantages pour le DBA

- …mais les développeurs doivent savoir l'utiliser
- Préférer une version récente de PostgreSQL

Le partitionnement par héritage n'a plus d'utilité pour la plupart des applications.

Le partitionnement déclaratif apparu en version 10 est mûr dans les dernières versions. Il introduit une complexité supplémentaire, que les développeurs doivent maîtriser, mais peut rendre de grands services quand la volumétrie augmente.

1.9 QUIZ

https://dali.bo/v1_quiz

1.10 TRAVAUX PRATIQUES

1.10.1 Partitionnement

But : Mettre en place le partitionnement déclaratif

Nous travaillons sur 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
```

Nous allons partitionner la table `stock` sur l'année.

Pour nous simplifier la vie, nous allons limiter le nombre d'années dans `stock` (cela nous évitera la création de 50 partitions) :

```
-- Création de lignes en 2001-2005
INSERT INTO stock SELECT vin_id, contenant_id, 2001 + annee % 5, sum(nombre)
FROM stock GROUP BY vin_id, contenant_id, 2001 + annee % 5;
-- purge des lignes prédédentes
DELETE FROM stock WHERE annee<2001;
```

Nous n'avons maintenant que des bouteilles des années 2001 à 2005.

- Renommer `stock` en `stock_old`.
- Créer une table partitionnée `stock` vide, sans index pour le moment.

- Créer les partitions de `stock`, avec la contrainte d'année : `stock_2001` à `stock_2005`.

- Insérer tous les enregistrements venant de l'ancienne table `stock`.

- Passer les statistiques pour être sûr des plans à partir de maintenant (nous avons modifié beaucoup d'objets).

- Vérifier la présence d'enregistrements dans `stock_2001` (syntaxe `SELECT ONLY`).
- Vérifier qu'il n'y en a aucun dans `stock`.

- Vérifier qu'une requête sur `stock` sur 2002 ne parcourt qu'une seule partition.

- Remettre en place les index présents dans la table `stock` originale.
- Il se peut que d'autres index ne servent à rien (ils ne seront dans ce cas pas présents dans la correction).

- Quel est le plan pour la récupération du stock des bouteilles du `vin_id` 1725, année 2003 ?

- Essayer de changer l'année de ce même enregistrement de `stock` (la même que la précédente). Pourquoi cela échoue-t-il ?

- Supprimer les enregistrements de 2004 pour `vin_id` = 1725.
- Retenter la mise à jour.

- Pour vider complètement le stock de 2001, supprimer la partition `stock_2001`.

- Tenter d'ajouter au stock une centaine de bouteilles de 2006.
- Pourquoi cela échoue-t-il ?

- Créer une partition par défaut pour recevoir de tels enregistrements.
- Retenter l'ajout.

- Tenter de créer la partition pour l'année 2006. Pourquoi cela échoue-t-il ?

- Pour créer la partition sur 2006, au sein d'une seule transaction :
- détacher la partition par défaut ;
- y déplacer les enregistrements mentionnés ;
- ré-attacher la partition par défaut.

1.10.2 Partitionner pendant l'activité

But : Mettre en place le partitionnement déclaratif sur une base en cours d'activité

1.10.2.1 Préparation

> Créer une base **pgbench** vierge, de taille 10 ou plus.

NB : Pour le TP, la base sera d'échelle 10 (environ 168 Mo). Des échelles 100 ou 1000 seraient plus réalistes.

> Dans une fenêtre en arrière-plan, laisser tourner un processus `pgbench` avec une activité la plus soutenue possible. Il ne doit pas tomber en erreur pendant que les tables vont être partitionnées ! Certaines opérations vont poser des verrous, le but va être de les réduire au maximum.

> Pour éviter un « empilement des verrous » et ne pas bloquer trop longtemps les opérations, faire en sorte que la transaction échoue si l'obtention d'un verrou dure plus de 10 s.

1.10.2.2 Partitionnement par *hash*

> Pour partitionner la table `pgbench_accounts` par *hash* sur la colonne `aid` sans que le traitement pgbench tombe en erreur, préparer un script avec, dans une transaction :
>
> - la création d'une table partitionnée par hash en 3 partitions au moins ;
> - le transfert des données depuis `pgbench_accounts` ;
> - la substitution de la table partitionnée à la table originale.
>
> Tester et exécuter.

> Supprimer l'ancienne table `pgbench_accounts_old`.

1.10.2.3 Partitionnement par valeur

`pgbench` doit continuer ses opérations en tâche de fond.

> La table `pgbench_history` se remplit avec le temps. Elle doit être partitionnée par date (champ `mtime`). Pour le TP, on fera 2 partitions d'une minute, et une partition par défaut. La table actuelle doit devenir une partition de la nouvelle table partitionnée.
>
> - Écrire un script qui, dans une seule transaction, fait tout cela et substitue la table partitionnée à la table d'origine.
>
> NB : Pour éviter de coder des dates en dur, il est possible, avec `psql`, d'utiliser une variable :
>
> ```
> SELECT ( now()+ interval '60s') AS date_frontiere \gset
> SELECT :'date_frontiere'::timestamptz ;
> ```

Exécuter le script, attendre que les données s'insèrent dans les nouvelles partitions.

1.10.2.4 Purge

- Continuer de laisser tourner `pgbench` en arrière-plan.
- Détacher et détruire la partition avec les données les plus anciennes.

1.10.2.5 Contraintes entre tables partitionnées

- Ajouter une clé étrangère entre `pgbench_accounts` et `pgbench_history`. Voir les contraintes créées.

Si vous n'avez pas déjà eu un problème à cause du `statement_timeout`, dropper la contrainte et recommencer avec une valeur plus basse. Comment contourner ?

1.10.2.6 Index global

On veut créer un index sur `pgbench_history (aid)`.

Pour ne pas gêner les écritures, il faudra le faire de manière concurrente. Créer l'index de manière concurrente sur chaque partition, puis sur la table partitionnée.

1.11 TRAVAUX PRATIQUES (SOLUTIONS)

1.11.1 Partitionnement

But : Mettre en place le partitionnement déclaratif

Pour nous simplifier la vie, nous allons limiter le nombre d'années dans `stock` (cela nous évitera la création de 50 partitions).

```
INSERT INTO stock
SELECT vin_id, contenant_id, 2001 + annee % 5, sum(nombre)
FROM stock
GROUP BY vin_id, contenant_id, 2001 + annee % 5 ;

DELETE FROM stock WHERE annee<2001 ;
```

Nous n'avons maintenant que des bouteilles des années 2001 à 2005.

- Renommer `stock` en `stock_old`.
- Créer une table partitionnée `stock` vide, sans index pour le moment.

```
ALTER TABLE stock RENAME TO stock_old;
CREATE TABLE stock(LIKE stock_old) PARTITION BY LIST (annee);
```

- Créer les partitions de `stock`, avec la contrainte d'année : `stock_2001` à `stock_2005`.

```
CREATE TABLE stock_2001 PARTITION of stock FOR VALUES IN (2001) ;
CREATE TABLE stock_2002 PARTITION of stock FOR VALUES IN (2002) ;
CREATE TABLE stock_2003 PARTITION of stock FOR VALUES IN (2003) ;
CREATE TABLE stock_2004 PARTITION of stock FOR VALUES IN (2004) ;
CREATE TABLE stock_2005 PARTITION of stock FOR VALUES IN (2005) ;
```

- Insérer tous les enregistrements venant de l'ancienne table `stock`.

```
INSERT INTO stock SELECT * FROM stock_old;
```

- Passer les statistiques pour être sûr des plans à partir de maintenant (nous avons modifié beaucoup d'objets).

```
ANALYZE;
```

- Vérifier la présence d'enregistrements dans `stock_2001` (syntaxe `SELECT ONLY`).
- Vérifier qu'il n'y en a aucun dans `stock`.

```
SELECT count(*) FROM stock_2001;
SELECT count(*) FROM ONLY stock;
```

- Vérifier qu'une requête sur `stock` sur 2002 ne parcourt qu'une seule partition.

```
EXPLAIN ANALYZE SELECT * FROM stock WHERE annee=2002;
```

```
                                QUERY PLAN
--------------------------------------------------------------------------
 Append  (cost=0.00..417.36 rows=18192 width=16) (...)
   ->  Seq Scan on stock_2002  (cost=0.00..326.40 rows=18192 width=16) (...)
         Filter: (annee = 2002)
 Planning Time: 0.912 ms
 Execution Time: 21.518 ms
```

- Remettre en place les index présents dans la table `stock` originale.
- Il se peut que d'autres index ne servent à rien (ils ne seront dans ce cas pas présents dans la correction).

```
CREATE UNIQUE INDEX ON stock (vin_id,contenant_id,annee);
```

Les autres index ne servent à rien sur les partitions : `idx_stock_annee` est évidemment inutile, mais `idx_stock_vin_annee` aussi, puisqu'il est inclus dans l'index unique que nous venons de créer.

- Quel est le plan pour la récupération du stock des bouteilles du `vin_id` 1725, année 2003 ?

```
EXPLAIN ANALYZE SELECT * FROM stock WHERE vin_id=1725 AND annee=2003 ;
```

```
 Append  (cost=0.29..4.36 rows=3 width=16) (...)
   ->  Index Scan using stock_2003_vin_id_contenant_id_annee_idx on stock_2003 (...)
         Index Cond: ((vin_id = 1725) AND (annee = 2003))
 Planning Time: 1.634 ms
 Execution Time: 0.166 ms
```

- Essayer de changer l'année de ce même enregistrement de `stock` (la même que la précédente). Pourquoi cela échoue-t-il ?

```
UPDATE stock SET annee=2004 WHERE annee=2003 and vin_id=1725 ;
```

```
ERROR:  duplicate key value violates unique constraint
↪  "stock_2004_vin_id_contenant_id_annee_idx"
DETAIL:  Key (vin_id, contenant_id, annee)=(1725, 1, 2004) already exists.
```

C'est une violation de contrainte unique, qui est une erreur normale : nous avons déjà un enregistrement de stock pour ce vin pour l'année 2004.

- Supprimer les enregistrements de 2004 pour `vin_id` = 1725.
- Retenter la mise à jour.

```
DELETE FROM stock WHERE annee=2004 and vin_id=1725;

UPDATE stock SET annee=2004 WHERE annee=2003 and vin_id=1725 ;
```

- Pour vider complètement le stock de 2001, supprimer la partition `stock_2001`.

```
DROP TABLE stock_2001 ;
```

- Tenter d'ajouter au stock une centaine de bouteilles de 2006.
- Pourquoi cela échoue-t-il ?

```
INSERT INTO stock (vin_id, contenant_id, annee, nombre) VALUES (1, 1, 2006, 100) ;

ERROR:  no partition of relation "stock" found for row
DETAIL:  Partition key of the failing row contains (annee) = (2006).
```

Il n'existe pas de partition définie pour l'année 2006, cela échoue donc.

- Créer une partition par défaut pour recevoir de tels enregistrements.
- Retenter l'ajout.

```
CREATE TABLE stock_default PARTITION OF stock DEFAULT ;

INSERT INTO stock (vin_id, contenant_id, annee, nombre) VALUES (1, 1, 2006, 100) ;
```

- Tenter de créer la partition pour l'année 2006. Pourquoi cela échoue-t-il ?

```
CREATE TABLE stock_2006 PARTITION of stock FOR VALUES IN (2006) ;

ERROR:  updated partition constraint for default partition "stock_default"
        would be violated by some row
```

Cela échoue car des enregistrements présents dans la partition par défaut répondent à cette nouvelle contrainte de partitionnement.

- Pour créer la partition sur 2006, au sein d'une seule transaction :
- détacher la partition par défaut ;
- y déplacer les enregistrements mentionnés ;
- ré-attacher la partition par défaut.

```
BEGIN ;

ALTER TABLE stock DETACH PARTITION stock_default;

CREATE TABLE stock_2006 PARTITION of stock FOR VALUES IN (2006) ;

INSERT INTO stock SELECT * FROM stock_default WHERE annee = 2006 ;

DELETE FROM stock_default WHERE annee = 2006 ;

ALTER TABLE stock ATTACH PARTITION stock_default DEFAULT ;

COMMIT ;
```

1.11.2 Partitionner pendant l'activité

But : Mettre en place le partitionnement déclaratif sur une base en cours d'activité

1.11.2.1 Préparation

Créer une base **pgbench** vierge, de taille 10 ou plus.

```
$ createdb pgbench
$ /usr/pgsql-14/bin/pgbench -i -s 10 pgbench
```

Dans une fenêtre en arrière-plan, laisser tourner un processus `pgbench` avec une activité la plus soutenue possible. Il ne doit pas tomber en erreur pendant que les tables vont être partitionnées ! Certaines opérations vont poser des verrous, le but va être de les réduire au maximum.

```
$ /usr/pgsql-14/bin/pgbench -n -T3600 -c20 -j2 --debug pgbench
```

L'activité est à ajuster en fonction de la puissance de la machine. Laisser l'affichage défiler dans une fenêtre pour bien voir les blocages.

Pour éviter un « empilement des verrous » et ne pas bloquer trop longtemps les opérations, faire en sorte que la transaction échoue si l'obtention d'un verrou dure plus de 10 s.

Un verrou en attente peut bloquer les opérations d'autres transactions venant après. On peut annuler l'opération à partir d'un certain seuil pour éviter ce phénomène :

```
pgbench=# SET lock_timeout TO '10s' ;
```

Cela ne concerne cependant pas les opérations une fois que les verrous sont acquis. On peut garantir qu'un ordre donné ne durera pas plus d'une certaine durée :

```
SET statement_timeout TO '10s' ;
```

En fonction de la rapidité de la machine et des données à déplacer, cette interruption peut être tolérable ou non.

1.11.2.2 Partitionnement par *hash*

Pour partitionner la table `pgbench_accounts` par *hash* sur la colonne `aid` sans que le traitement pgbench tombe en erreur, préparer un script avec, dans une transaction :

- la création d'une table partitionnée par hash en 3 partitions au moins ;
- le transfert des données depuis `pgbench_accounts` ;
- la substitution de la table partitionnée à la table originale.

> Tester et exécuter.

Le champ `aid` n'a pas de signification, un partitionnement par *hash* est adéquat.

Le script peut être le suivant :

```
\timing on
\set ON_ERROR_STOP 1

SET lock_timeout TO '10s' ;
SET statement_timeout TO '10s' ;

BEGIN ;

-- Nouvelle table partitionnée

CREATE TABLE pgbench_accounts_part (LIKE pgbench_accounts INCLUDING ALL)
PARTITION BY HASH (aid) ;

CREATE TABLE pgbench_accounts_1 PARTITION OF pgbench_accounts_part
FOR VALUES WITH (MODULUS 3, REMAINDER 0 ) ;

CREATE TABLE pgbench_accounts_2 PARTITION OF pgbench_accounts_part
FOR VALUES WITH (MODULUS 3, REMAINDER 1 ) ;

CREATE TABLE pgbench_accounts_3 PARTITION OF pgbench_accounts_part
FOR VALUES WITH (MODULUS 3, REMAINDER 2 ) ;

-- Transfert des données

-- Bloquer les accès à la table le temps du transfert
-- (sinon risque de perte de données !)
LOCK TABLE pgbench_accounts ;

-- Copie des données
INSERT INTO pgbench_accounts_part
SELECT * FROM pgbench_accounts ;

-- Substitution par renommage
ALTER TABLE pgbench_accounts RENAME TO pgbench_accounts_old ;
ALTER TABLE pgbench_accounts_part RENAME TO pgbench_accounts ;

-- Contrôle

\d+

-- On ne validera qu'après contrôle
-- (pendant ce temps les sessions concurrentes restent bloquées !)

COMMIT ;
```

À la moindre erreur, la transaction tombe en erreur. Il faudra demander manuellement `ROLLBACK`.

Si la durée fixée par `statement_timeout` est dépassée, on aura cette erreur :

```
ERROR:  canceling statement due to statement timeout
Time: 10115.506 ms (00:10.116)
```

Surtout, le traitement pgbench reprend en arrière-plan. On peut alors relancer le script corrigé plus tard.

Si tout se passe bien, un `\d+` renvoie ceci :

```
                          Liste des relations
 Schéma |         Nom          |        Type        | Propriétaire | Taille  | …
--------+----------------------+--------------------+--------------+---------+--
 public | pgbench_accounts     | table partitionnée | postgres     | 0 bytes |
 public | pgbench_accounts_1   | table              | postgres     | 43 MB   |
 public | pgbench_accounts_2   | table              | postgres     | 43 MB   |
 public | pgbench_accounts_3   | table              | postgres     | 43 MB   |
 public | pgbench_accounts_old | table              | postgres     | 130 MB  |
 public | pgbench_branches     | table              | postgres     | 136 kB  |
 public | pgbench_history      | table              | postgres     | 5168 kB |
 public | pgbench_tellers      | table              | postgres     | 216 kB  |
```

On peut vérifier rapidement que les valeurs de `aid` sont bien réparties entre les 3 partitions :

```
SELECT aid FROM pgbench_accounts_1 LIMIT 3 ;
```

```
 aid
-----
   2
   6
   8
```

```
SELECT aid FROM pgbench_accounts_2 LIMIT 3 ;
```

```
 aid
-----
   3
   7
  10
```

```
SELECT aid FROM pgbench_accounts_3 LIMIT 3 ;
```

```
 aid
-----
   1
   9
  11
```

Après la validation du script, on voit apparaître les lignes dans les nouvelles partitions :

```
SELECT relname, n_live_tup
FROM pg_stat_user_tables
WHERE relname LIKE 'pgbench_accounts%' ;
```

```
       relname        | n_live_tup
----------------------+------------
 pgbench_accounts_old |    1000002
 pgbench_accounts_1   |     333263
 pgbench_accounts_2   |     333497
 pgbench_accounts_3   |     333240
```

> Supprimer l'ancienne table `pgbench_accounts_old`.

```
DROP TABLE pgbench_accounts_old ;
```

1.11.2.3 Partitionnement par valeur

`pgbench` doit continuer ses opérations en tâche de fond.

> La table `pgbench_history` se remplit avec le temps. Elle doit être partitionnée par date (champ `mtime`). Pour le TP, on fera 2 partitions d'une minute, et une partition par défaut. La table actuelle doit devenir une partition de la nouvelle table partitionnée.
>
> - Écrire un script qui, dans une seule transaction, fait tout cela et substitue la table partitionnée à la table d'origine.
>
> NB : Pour éviter de coder des dates en dur, il est possible, avec `psql`, d'utiliser une variable :
>
> ```
> SELECT ( now()+ interval '60s') AS date_frontiere \gset
> SELECT :'date_frontiere'::timestamptz ;
> ```

La « date frontière » doit être dans le futur (proche). En effet, `pgbench` va modifier les tables en permanence, on ne sait pas exactement à quel moment la transition aura lieu (et de toute façon on ne maîtrise pas les valeurs de `mtime`) : il continuera donc à écrire dans l'ancienne table, devenue partition, pendant encore quelques secondes.

Cette date est arbitrairement à 1 minute dans le futur, pour dérouler le script manuellement :

```
SELECT ( now()+ interval '60s') AS date_frontiere \gset
```

Et on peut réutiliser cette variable ainsi ;

```
SELECT :'date_frontiere'::timestamptz ;
```

Le script peut être celui-ci :

```
\timing on
\set ON_ERROR_STOP 1

SET lock_timeout TO '10s' ;
SET statement_timeout TO '10s' ;

SELECT ( now()+ interval '60s') AS date_frontiere \gset
SELECT :'date_frontiere'::timestamptz ;

BEGIN ;

-- Nouvelle table partitionnée
CREATE TABLE pgbench_history_part (LIKE pgbench_history INCLUDING ALL)
PARTITION BY RANGE (mtime) ;
```

```
-- Des partitions pour les prochaines minutes

CREATE TABLE pgbench_history_1
PARTITION OF pgbench_history_part
FOR VALUES FROM (:'date_frontiere'::timestamptz  )
             TO (:'date_frontiere'::timestamptz + interval '1min' ) ;

CREATE TABLE pgbench_history_2
PARTITION OF pgbench_history_part
FOR VALUES FROM (:'date_frontiere'::timestamptz + interval '1min' )
             TO (:'date_frontiere'::timestamptz + interval '2min' ) ;

-- Au cas où le service perdure au-delà des partitions prévues,
-- on débordera dans cette table
CREATE TABLE pgbench_history_default
PARTITION OF pgbench_history_part DEFAULT ;

-- Jusqu'ici pgbench continue de tourner en arrière plan

-- La table devient une simple partition
-- Ce renommage pose un verrou, les sessions pgbench sont bloquées
ALTER TABLE pgbench_history RENAME TO pgbench_history_orig ;

ALTER TABLE pgbench_history_part
ATTACH PARTITION pgbench_history_orig
FOR VALUES FROM (MINVALUE) TO (:'date_frontiere'::timestamptz) ;

-- Contrôle
\dP

-- Substitution de la table partitionnée à celle d'origine.
ALTER TABLE pgbench_history_part RENAME TO pgbench_history ;

-- Contrôle
\d+ pgbench_history


COMMIT ;
```

> Exécuter le script, attendre que les données s'insèrent dans les nouvelles partitions.

Pour surveiller le contenu des tables jusqu'à la transition :

```
SELECT relname, n_live_tup, now()
FROM pg_stat_user_tables
WHERE relname LIKE 'pgbench_history%' ;

\watch 3
```

Un `\d+` doit renvoyer ceci :

```
                                 Liste des relations
 Schéma |           Nom           |        Type        | Propriétaire | Taille  | …
--------+-------------------------+--------------------+--------------+---------+--
 public | pgbench_accounts        | table partitionnée | postgres     | 0 bytes |
```

```
 public | pgbench_accounts_1      | table              | postgres     | 44 MB   |
 public | pgbench_accounts_2      | table              | postgres     | 44 MB   |
 public | pgbench_accounts_3      | table              | postgres     | 44 MB   |
 public | pgbench_branches        | table              | postgres     | 136 kB  |
 public | pgbench_history         | table partitionnée | postgres     | 0 bytes |
 public | pgbench_history_1       | table              | postgres     | 672 kB  |
 public | pgbench_history_2       | table              | postgres     | 0 bytes |
 public | pgbench_history_default | table              | postgres     | 0 bytes |
 public | pgbench_history_orig    | table              | postgres     | 8736 kB |
 public | pgbench_tellers         | table              | postgres     | 216 kB  |
```

1.11.2.4 Purge

- Continuer de laisser tourner `pgbench` en arrière-plan.
- Détacher et détruire la partition avec les données les plus anciennes.

```
ALTER TABLE pgbench_history
DETACH PARTITION pgbench_history_orig ;

-- On pourrait faire le DROP directement

DROP TABLE pgbench_history_orig ;
```

1.11.2.5 Contraintes entre tables partitionnées

- Ajouter une clé étrangère entre `pgbench_accounts` et `pgbench_history`. Voir les contraintes créées.

NB : les clés étrangères entre tables partitionnées ne sont pas disponibles avant PostgreSQL 12.

```
SET lock_timeout TO '3s' ;
SET statement_timeout TO '10s' ;

CREATE INDEX ON pgbench_history (aid) ;

ALTER TABLE pgbench_history
ADD CONSTRAINT pgbench_history_aid_fkey FOREIGN KEY (aid) REFERENCES
 ↪  pgbench_accounts ;
```

On voit que chaque partition porte un index comme la table mère. La contrainte est portée par chaque partition.

```
pgbench=# \d+ pgbench_history
  Table partitionnée « public.pgbench_history »
 …
Clé de partition : RANGE (mtime)
Index :
    "pgbench_history_aid_idx" btree (aid)
Contraintes de clés étrangères :
    "pgbench_history_aid_fkey" FOREIGN KEY (aid) REFERENCES pgbench_accounts(aid)
Partitions: pgbench_history_1 FOR VALUES FROM ('2020-02-14 17:41:08.298445')
```

```
                              TO ('2020-02-14 17:42:08.298445'),
            pgbench_history_2 FOR VALUES FROM ('2020-02-14 17:42:08.298445')
                              TO ('2020-02-14 17:43:08.298445'),
            pgbench_history_default DEFAULT

pgbench=# \d+ pgbench_history_1
   Table « public.pgbench_history_1 »
…
Partition de : pgbench_history FOR VALUES FROM ('2020-02-14 17:41:08.298445')
                                          TO ('2020-02-14 17:42:08.298445')
Contrainte de partition : ((mtime IS NOT NULL)
    AND(mtime >= '2020-02-14 17:41:08.298445'::timestamp without time zone)
    AND (mtime < '2020-02-14 17:42:08.298445'::timestamp without time zone))
Index :
    "pgbench_history_1_aid_idx" btree (aid)
Contraintes de clés étrangères :
    TABLE "pgbench_history" CONSTRAINT "pgbench_history_aid_fkey"
                            FOREIGN KEY (aid) REFERENCES pgbench_accounts(aid)
Méthode d'accès : heap
```

> Si vous n'avez pas déjà eu un problème à cause du `statement_timeout`, dropper la contrainte et recommencer avec une valeur plus basse. Comment contourner ?

Le `statement_timeout` peut être un problème :

```
SET
pgbench=# ALTER TABLE pgbench_history
          ADD CONSTRAINT pgbench_history_aid_fkey FOREIGN KEY (aid)
          REFERENCES pgbench_accounts ;
ERROR:  canceling statement due to statement timeout
```

On peut créer les contraintes séparément sur les tables. Cela permet de ne poser un verrou sur la partition active (sans doute `pgbench_history_default`) que pendant le strict minimum de temps (les autres partitions de `pgbench_history` ne sont pas utilisées).

```
SET statement_timeout to '1s' ;
ALTER TABLE pgbench_history_1 ADD CONSTRAINT pgbench_history_aid_fkey
 FOREIGN KEY (aid) REFERENCES pgbench_accounts ;
ALTER TABLE pgbench_history_2 ADD CONSTRAINT pgbench_history_aid_fkey
 FOREIGN KEY (aid) REFERENCES pgbench_accounts ;
ALTER TABLE pgbench_history_default ADD CONSTRAINT pgbench_history_aid_fkey
 FOREIGN KEY (aid) REFERENCES pgbench_accounts ;
```

La contrainte au niveau global sera alors posée presque instantanément :

```
ALTER TABLE pgbench_history ADD CONSTRAINT pgbench_history_aid_fkey
 FOREIGN KEY (aid) REFERENCES pgbench_accounts ;
```

1.11.2.6 Index global

> On veut créer un index sur `pgbench_history (aid)`.
>
> Pour ne pas gêner les écritures, il faudra le faire de manière concurrente. Créer l'index de manière concurrente sur chaque partition, puis sur la table partitionnée.

Construire un index de manière concurrente (clause `CONCURRENTLY`) permet de ne pas bloquer la table en écriture pendant la création de l'index, qui peut être très longue. Mais il n'est pas possible de le faire sur la table partitionnée :

```
CREATE INDEX CONCURRENTLY ON pgbench_history (aid) ;

ERROR:  cannot create index on partitioned table "pgbench_history" concurrently
```

Mais on peut créer l'index sur chaque partition séparément :

```
CREATE INDEX CONCURRENTLY ON pgbench_history_1 (aid) ;
CREATE INDEX CONCURRENTLY ON pgbench_history_2 (aid) ;
CREATE INDEX CONCURRENTLY ON pgbench_history_default (aid) ;
```

S'il y a beaucoup de partitions, on peut générer dynamiquement ces ordres :

```
SELECT 'CREATE INDEX CONCURRENTLY ON ' ||
       c.oid::regclass::text || ' (aid) ; '
FROM pg_class c
WHERE relname like 'pgbench_history%' AND relispartition \gexec
```

Comme lors de toute création concurrente, il faut vérifier que les index sont bien valides : la requête suivante ne doit rien retourner.

```
SELECT indexrelid::regclass FROM pg_index WHERE NOT  indisvalid ;
```

Enfin on crée l'index au niveau de la table partitionnée : il réutilise les index existants et sera donc créé presque instantanément :

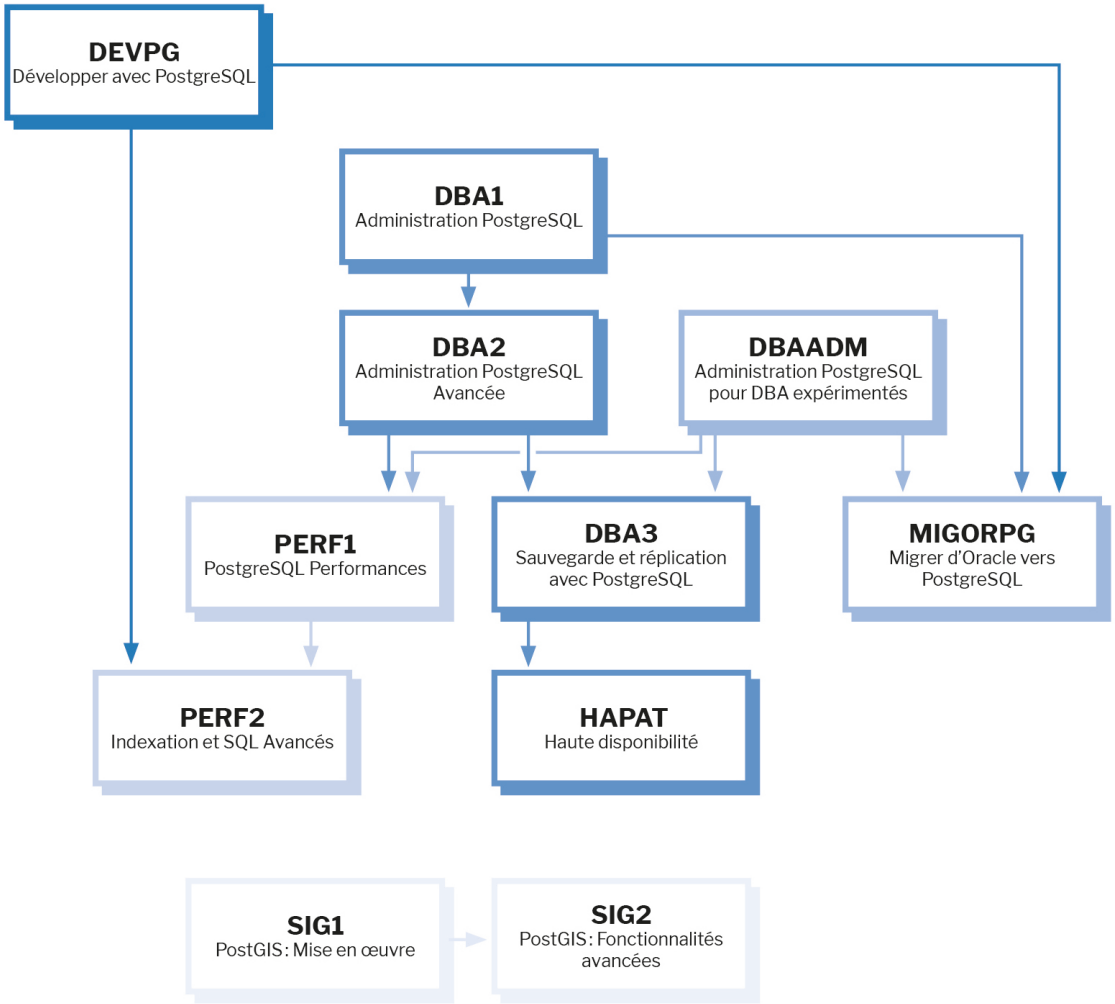
```
CREATE INDEX ON pgbench_history(aid) ;

pgbench=# \d+ pgbench_history
..
Partition key: RANGE (mtime)
Indexes:
    "pgbench_history_aid_idx" btree (aid)
…
```

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.