Partitionnement et transaction autonomes avec PostgreSQL

Quelles solutions ?

Gilles Darold

Auteur : Gilles Darold

- Société : **Dalibo**
- Date : Juin 2017
- Développeur / Mainteneur :
  - Ora2Pg : http://www.ora2pg.com/
  - PgCluu : http://pgcluu.darold.net/
  - PgBadger : http://dalibo.github.io/pgbadger/
  - PgFormatter : http://sqlformat.darold.net/
  - Et autres : http://www.darold.net/

Quand partitionner ?

- A partir de quel volume partitionner une table dans PostgreSQL?
  - Ma table fait 80 Go, dois-je partitionner ?

- A partir de combien de millions de lignes partitionner une table PostgreSQL?
  - J'ai plus de 250 Millions de lignes dans une table, dois-je partitionner ?

Améliorer les performances

- Le partitionnement n'est pas une question de nombre de ligne ou de volume.
- C'est une réponse à des problèmes de performances
  - Soit en terme de lecture / écriture
  - Soit en terme de maintenance (reindex, vacuum, sauvegarde)
  - Les deux
- Bien sur il intervient principalement sur de gros volume, mais pas que.

Quand ne pas partitionner ?

- Lorsque le problème de performance peut être résolu avec :
  - Des index partiels utilisant les filtres des requêtes
    - CREATE INDEX ON users(login) WHERE date_part('year', create_date) = 2016;
  - Des index couvrants pour favoriser le parcours d'index seul
    - CREATE INDEX ON users(login, create_date) WHERE date_part('year', create_date) = 2016 ;
  - Des index BRIN ou des index Bloom
    - index plus petits indexant des ensembles
    - nécessitent une vérification pour trouver les enregistrements dans les ensembles
- Et bien sur que les temps de maintenance des index sont supportables.

Bénéfices du partitionnement

- Dépasser les limites de l'architecture physique
- Réduction de la taille des index
  - index locaux à chaque partition
  - les index tiennent en mémoire
- Réduction de la taille des tables
  - Segmentation sur des critères utiles aux requêtes.
  - Accès séquentiels privilégié sur des volumes bien plus faible
  - Les partitions tiennent en mémoire

Bénéfices du partitionnement

- Chargement de données plus rapide
  - Mise à jour d'index peu volumineux.
  - Dispersion des partitions sur des disques ou serveurs différents
- Purge des données par DROP TABLE plutôt que des DELETE lents.
- Taches de maintenance plus rapides
  - Reindex : les index sont locaux à chaque partition
  - Sauvegarde : des données statiques peuvent être omises lors de la sauvegarde après archivage

Et dans quel but ?

- Quels sont les objectifs du partitionnement ?
  - Archivage de données.
  - Purge de données.
  - Performances.
  - Facilité d'administration.
  - Accélération des backups.
  - Stockage sur différents types de disques.
  - Cas particulier de la fragmentation.
- Plusieurs objectifs sont possibles, définir les priorités.

MVCC et partitionnement

- Le partitionnement peut être utilisé pour répondre à un problème particulier de fragmentation lié au fonctionnement interne de PostgreSQL.
- Peu de lignes ou de volume à l'instant T mais énormément de volume en E/S d'une table.
  - Table de type file d'attente : INSERT + DELETE en très forte charge
  - La fragmentation augmente constamment même après tuning de l'autovacuum.
- Partitionnement à la journée ou l'heure pour pouvoir utiliser des TRUNCATE à échéance de l'utilisation d'une partition.

Prérequis au partitionnement

- Déterminer la clé de partitionnement !
  - Colonne(s) toujours utilisée(s) dans les prédicats des requêtes.
- Déterminer la stratégie de partitionnement
  - RANGE : La valeur de la clé est comprise entre telle et telle valeur.
  - LIST : la valeur de la clé est dans la liste fournie
- La clé du partitionnement par liste est limité à une seule colonne
- Il y a d'autres types de partitionnement : HASH et REFERENCE
  - ils ne sont pas supportés par PostgreSQL pour l'instant.

Optimisation de la volumétrie

- L'alignement ! Sur de grands volumes de lignes, les attributs doivent impérativement être alignés.
- CREATE TABLE fat_table (col1 integer, col2 bigint, col3 boolean, col4 integer, col5 text) ;

- CREATE TABLE thin_table (col1 integer, col4 integer, col2 bigint, col5 text, col3 boolean) ;

- Gain : ici 7 octets sur 32, plus de 20 % d'espace disque préservé !

Optimisation de la volumétrie

- Même chose pour les index BTREE, l'alignement est sur 8 octets.
- CREATE INDEX ON thin_table (col1) ;

- CREATE INDEX ON thin_table (col1, col4) ;

- Pas de gain d'espace disque, mais plus d'information stockée pour le même espace.

Optimisation de la volumétrie

- Utiliser la compression réalisée par TOAST !
- Plusieurs attributs ou lignes d'entités peuvent être regroupés au sein d'un seul document JSON ou agrégées dans un tableau.
- Le stockage TOAST compresse automatiquement les données
  - Gain d'espace disque => x5 à x7, voir plus en fonction de l'agrégation des données,
  - ex : une table de 150 GB passe à 30 GB
- Autres gains :
  - La table tiens en RAM
  - Les VACUUM/ANALYSE se font beaucoup plus rapidement
- A limiter aux données non-relationnelles, pas de contrainte d'unicité, ni de FK.
- Nécessite une adaptation de l'applicatif.

Partitionnement par héritage

- Avantages :
  - Partitions par défaut supportées
  - Division des données dans des tables locales et/ou distantes
  - Écriture dans les partitions distantes
  - Sous partitions possibles
  - Modification sur la clé de partitionnement par triggers
- Inconvénients :
  - Lenteurs en raison des triggers
  - Administration compliquée même si des extensions la simplifie comme :
    - pg_partman https://github.com/keithf4/pg_partman
    - range_partition https://github.com/moat/range_partitioning
  - Impossible d'ajouter/supprimer une partition sans modifier le trigger.
  - Les index doivent être créés manuellement sur chaque partition

Partitionnement par héritage

```
CREATE TABLE list_part (
        deptno bigint,
        deptname varchar(20),
        quarterly_sales double precision,
        state varchar(2)
) ;
CREATE TABLE list_part_q1_northwest (
        CHECK (STATE IN ('OR', 'WA')
) ) INHERITS (LIST_PART);
CREATE TABLE list_part_q1_southwest (
        CHECK (STATE IN ('AZ', 'CA', 'NM')
) ) INHERITS (LIST_PART);
...
```

Partitionnement par héritage

```
CREATE OR REPLACE FUNCTION LIST_PART_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
	IF ( NEW.STATE IN ('OR', 'WA') ) THEN INSERT INTO LIST_PART_Q1_NORTHWEST VALUES (NEW.*);
	ELSIF ( NEW.STATE IN ('AZ', 'CA', 'NM') ) THEN INSERT INTO LIST_PART_Q1_SOUTHWEST VALUES (NEW.*);
	ELSIF ( NEW.STATE IN ('NY', 'VT', 'NJ') ) THEN INSERT INTO LIST_PART_Q1_NORTHEAST VALUES (NEW.*);
	ELSIF ( NEW.STATE IN ('FL', 'GA') ) THEN INSERT INTO LIST_PART_Q1_SOUTHEAST VALUES (NEW.*);
	ELSIF ( NEW.STATE IN ('MN', 'WI') ) THEN INSERT INTO LIST_PART_Q1_NORTHCENT VALUES (NEW.*);
	ELSIF ( NEW.STATE IN ('OK', 'TX') ) THEN INSERT INTO LIST_PART_Q1_SOUTHCENT VALUES (NEW.*);
	ELSE
		RAISE EXCEPTION 'Value out of range in partition. Fix the LIST_PART_insert_trigger() function!';
	END IF;
	RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

Partitionnement déclaratif

- Nouvelle fonctionnalité de la version 10.0
- Utilise la même infrastructure que le partitionnement par héritage mais :
  - La table mère ne peut avoir de données
  - La table mère ne peut avoir d'index, donc pas de PK ni UK
  - Chaque partition a une contrainte implicite de partitionnement
  - Les partitions ne peuvent avoir de colonnes additionnelles
  - L'héritage multiple n'est pas permis
- Une colonne clé du partitionnement peut être une expression.
- Les données insérées sont automatiquement routées dans la bonne partition.

Partitionnement déclaratif

- Avantages :
  - Rapide, pas de triggers SQL, les accès sont routés directement vers les partitions.
  - Administration simplifiée et intégrée au SGBD
  - Permet une division des données dans des tables locales et/ou distantes
  - Sous partitions supportées
  - DROP partition_name supprime la partition et ses données (ACCES EXCLUSIVE LOCK sur la table parente).
  - Possibilité d'attacher / détacher (ATTACH/DETACH) des partitions.
- Inconvénients :
  - Pas de partitions par défaut
  - Ne permet pas l'écriture dans les partitions distantes
  - Pas de modification sur la clé de partitionnement
  - Les index doivent être créer manuellement sur chaque partition

Partitionnement déclaratif

```
CREATE TABLE list_part (
        deptno bigint,
        deptname varchar(20),
        quarterly_sales double precision,
        state varchar(2)
) PARTITION BY LIST (state) ;

CREATE TABLE list_part_northwest PARTITION OF list_part
    FOR VALUES IN ('OR', 'WA');

CREATE TABLE list_part_southwest PARTITION OF list_part
    FOR VALUES IN ('AZ', 'CA', 'NM');
...
```

Partitionnement déclaratif

- Suppression d'une partition :
  - ALTER TABLE list_part DETACH PARTITION list_part_southwest ;
- Ajout ou réintégration d'une partition :
  - ALTER TABLE list_part ATTACH PARTITION list_part_southwest FOR VALUES IN ('AZ', 'CA', 'NM');
- Les partitions peuvent contenir des données.
- Partitions attachées inaccessibles le temps d'un seq_scan de la table pour vérifier que la clé de partitionnement est respectée
  - Solution créer la contrainte check sur la table avant et la supprimer après l'attachement.
- Supporté par pg_partman >= 3.0.1

Changement du catalogue

- Table pg_class
  - table partitionnée : relkind = 'p'
  - partition : relkind = 'r'
- Deux nouvelles colonnes :
  - relispartition : à vrai ('t') pour les deux
  - relpartbound : représentation interne des bornes du partitionnement

Changement du catalogue

- Nouvelle table du catalogue : pg_partitioned_tables
  - partrelid => OID de la table dans pg_class
  - partstrat => stratégie de partitionnement: l = list, r = range
  - partnatts => nombre de colonnes dans la clé de partitionnement
  - partattrs => indices des colonnes dans la table, 0 indique une expression
  - partclass => classe d'opérateur utilisée par chaque colonne de la clé
  - partcollation => OID de la collation utilisée par chaque colonne de la clé
  - partexprs => représentation interne des expressions utilisées dans la clé de partitionnement (quand partattrs=0)

Performances de partitionnement

- Test COPY de 1 millions d'enregistrements dans une table
  - Table non partitionnée: 15 672,801 ms
  - Partition par héritage: 54 168,627 ms
  - Partitionnement déclaratif: 17 458,848 ms
- Temps d'insertion du partitionnement déclaratif identique à une table normale
- Le partitionnement par héritage retourne 0 lignes insérées à cause des triggers
- Le partitionnement déclaratif retourne le nombre de lignes insérées.

  COPY 1000000

  Time: 16694,061 ms (00:16,694)
- Comme pour le partitionnement par héritage :
  - SET constraint_exclusion = partition;

Limitations du partitionnement

- Pas de clé primaire, de contrainte unique ou exclusive globale
- Donc pas de FK pointant vers ou depuis une table partitionnée
- Donc la clause ON CONFLICT ne peut pas être utilisée avec les tables partitionnées
- Pas de triggers au niveau ligne sur une table partitionnée
- Le partitionnement de type RANGE de texte ou de type HASH est sensible au changement d'architecture.
  - Les tris pour les ranges et les fonctions de hashing ne renvoient pas forcement le même résultats suivant l'architecture
  - Erreurs à la restauration des dumps, les valeurs ne correspondent pas à la partition
  - Option –dump-partition-data-with-parent à pg_dump ?

pg_pathman

- Extension qui propose une gestion optimisée du partitionnement
- Les informations sur les partitions sont stockées en shared memory pour être utilisées par le planificateur
- En fonction des prédicats d'une requête, recherche des partitions concernées et définition du plan
- Nombreuses fonctions d'administration
- Support des partitions par RANGE et HASH
- Création automatique des partitions par RANGE
- COPY est capable d'insérer directement dans les partitions
- Supporte les FDW (partitions distantes)
- A venir : partitionnement par LIST, sous partitionnement
- https://github.com/postgrespro/pg_pathman/

Migration partition Oracle

- Ora2Pg migre les partitions de type RANGE et LIST
  - sous forme d'héritage avec trigger
  - sous forme déclarative avec PostgreSQL v10.0
- Les sous partitions sont gérées dans les deux modes
- Les partitions par défaut sont gérées en mode héritage
- Différences sur la forme déclarative :
  - FOR VALUES IN ({ bound_literal | NULL }
  - FOR VALUES FROM ({ bound_literal }) TO ({ bound_literal })
- Qui n'acceptent pas une expression comme l'utilisation d'une fonction
- Elles doivent être repositionnées manuellement dans la clause PARTITION BY

Wanted / Missing

- Partitionnement RANGE avec un INTERVAL avec création automatique de nouvelles partitions.
- Partitionnement par défaut (en cours).
- Partitionnement de type HASH (en cours).
- Partitionnement de type Référence.
- Update sur la clé de partitionnement (en cours).

Transaction autonomes

- Pour réaliser une transaction autonome il faut passer par une connexion.
- Les FDW n'aident pas, ils sont transactionnels, un échec de la transaction annule toute modification réalisée sur une table distante.
- L'extension dblink permet de réaliser cette connexion.
  - https://www.postgresql.org/docs/devel/static/dblink.html
- L'extension pg_background permet la même chose mais au travers d'un background worker.
  - https://github.com/vibhorkum/pg_background
- BackgroundSession Api en C pour exécuter une requête dans un background worker. [WIP] dblink pourrait profiter de cette API au lieu de passer par la libpq. Patch en standby pour le moment.
  - https://postgr.es/m/9dbc706d-a5b4-83b8-9545-056ac3ec3815@2ndquadrant.com

Extension dblink

```
CREATE OR REPLACE FUNCTION log_action (
        username text, event_date timestamp, msg text
)
RETURNS VOID AS $body$
DECLARE
       v_conn_str  text := 'port=5432 dbname=testdb host=localhost user=pguser password=pgpass';
       v_query     text;
BEGIN
        v_query := 'SELECT true FROM log_action_atx ( ' || quote_nullable(username) ||
                                ',' || quote_nullable(event_date) || ',' || quote_nullable(msg) || ' )';
        PERFORM * FROM dblink(v_conn_str, v_query) AS p (ret boolean);
END;
$body$
LANGUAGE plpgsql SECURITY DEFINER;
```

Extension pg_background

```
CREATE OR REPLACE FUNCTION log_action (
        username text, event_date timestamp, msg text
) RETURNS VOID AS $body$
DECLARE
        v_query     text;
BEGIN
        v_query := 'SELECT true FROM log_action_atx ( ' || quote_nullable(username) ||
                                ',' || quote_nullable(event_date) || ',' || quote_nullable(msg) || ' )';
        PERFORM * FROM pg_background_result(pg_background_launch(v_query))
                        AS p (ret boolean);
END;
$body$
LANGUAGE plpgsql SECURITY DEFINER;
```

BackgroundSession API

```
#include "tcop/bgsession.h"

void
myfunc()
{
    BackgroundSession *session;
    BackgroundSessionResult *result;

    session = BackgroundSessionStart();

    result = BackgroundSessionExecute(session, "SELECT … FROM … WHERE ...");
    elog(INFO, "returned %d rows", list_length(result->tuples));

    BackgroundSessionEnd(session);
}
```

Merci pour votre attention !

Des questions ?