Nettoyage et Optimisation

Valentin DEHAINAULT ~ Amina KACIMI ~ Minh-Hoang DANG

Préambule:

A la fin de la phase précédente, nous avons réussi à nettoyer et normaliser plus de 35000 enregistrements en 1min 35s. Bien que nous avons utilisé les techniques d'optimisation pour obtenir ce résultat, nous cherchons à réduire encore ce temps, c'est à dire augmenter la performance en optimisant les requêtes utilisés. Nous allons présenter tous les optimisations utilisées précédemment et celles qu'on va ajouter.

Nettoyage:

Lors du précédent semestre on a procédé au nettoyage de manière à séparer les lignes combinées, modifier le contenu de la colonne Discriminant pour insérer d'autres données plus cohérentes, reformatter les dates, harmoniser les textes et les rendres plus uniformes et standardisés. Cela dans le but d'avoir des données cohérentes, lisibles, atomiques nous permettant de procéder à notre traitement. Nous sommes contents de l'état actuel car il nous permet déjà à faire des requêtes de manière pertinente.

Schéma:

Nous avons procédé étape par étape pour arriver au schéma ci-dessous, qui est en troisième forme normale. En quatrième forme normale, on aura beaucoup plus de table, c'est à dire plus de jointure plus tard. Pour cette raison nous avons décidé de rester avec ce schéma. Nous avons décidé d'ajouter les coordonnées WGS84 (avec longitude et latitude) pour utiliser dans notre tableau de bord.

Optimisation:

Modification des configurations serveur sous Postgres:

Nous avons changé les paramètres du système selon la configuration de la machine (Source: https://www.postgresql.org/docs/9.6/runtime-config-resource.html):

- `effective_cache_size`: le planificateur la quantité de mémoire pour la mise en cache. Une quantité qui est à 1/2 du RAM suffit.

- `work_mem`: permet de choisir quantité de mémoire que nous pouvons allouer pour un tri ou une opération de jointure associée. Une requête complexe unique peut donc utiliser plusieurs fois cette quantité de mémoire.

```
SET effective_cache_size TO '9GB';
SET maintenance_work_mem TO '768MB';
SET default_statistics_target TO '100';
SET random_page_cost TO '4';
SET effective_io_concurrency TO '2';
SET work_mem TO '157286kB';
SET max_parallel_workers_per_gather TO '2';
SET max_parallel_workers TO '4';
```

Nous n'avons pas oublié de remettre ces paramètres à leurs valeurs par défaut à la fin.

Insertion des données:

Pour les transferts de données, notamment pendant la normalisation, nous nous sommes servis des techniques d'optimisation pour écrire les données de manière plus efficace.

- Désactiver les contraints sur la table d'arrivée et les restituer à la fin.
- Supprimer les indexes avant l'importation.
- Insérer plusieurs lots de données au lieu d'insérer une ligne de données à la fois. Des lots sont préparés avec les ARRAY ou TYPE COMPOSITE. On peut donc insérer une seule fois à la fin.

Exemple de type composite:

```
1  CREATE TYPE CoordLambert93 AS (
2     CoordX numeric,
3     CoordY numeric
4  );
5
6  CREATE TYPE insert_array AS (
7     field varchar[]
8  );
```

Example d'insertion en lots:

```
-------------------------------------------------------------------------------
-- Insérer dans la bonne table les données séparées
-------------------------------------------------------------------------------
INSERT INTO Correction (ReferenceCindoc, Serie, Article, Discriminant, Ville,
                Sujet, DescDet, Date, NoteBP, Idx_Per, FicNum, Idx_Ico, NbrCli,
                TailleCli, N_V, C_G, Remarques, Coordonnees)
VALUES(
    unnest(
        array_expand(
            ReferenceCindocVals.field,
            maxLength, ReferenceCindocVals.field[1])::int[]
        ),
    NEW.serie,
    cast(NEW.article as int),
    unnest(traitement_discriminant(
              array_expand(
                  DiscriminantVals.field, maxLength,
```

```
                DiscriminantVals.field[1]
            )
        )
    ),
    unnest(VilleVals.field),
    unnest(pretty(array_expand(SujetVals.field, maxLength, SujetVals.field[1]))),
    NEW.DescDet,
    unnest(pretty(array_expand(DateVals.field, maxLength, DateVals.field[1]))),
    unnest(pretty(array_expand(NoteBPVals.field, maxLength, NoteBPVals.field[1]))),
    unnest(pretty(array_expand(Idx_PerVals.field, maxLength, Idx_PerVals.field[1]))),
    unnest(pretty(array_expand(FicNumVals.field, maxLength, FicNumVals.field[1]))),
    unnest(pretty(array_expand(Idx_IcoVals.field, maxLength, Idx_IcoVals.field[1]))),
    unnest(pretty(array_expand(NbrCliVals.field, maxLength, NbrCliVals.field[1]))),
    unnest(pretty(array_expand(TailleCliVals.field, maxLength, TailleCliVals.field[1]))),
    unnest(pretty(array_expand(N_VVals.field, maxLength, N_VVals.field[1]))),
    unnest(pretty(array_expand(C_GVals.field, maxLength, C_GVals.field[1]))),
    NEW.Remarques,
    unnest(array_expand(CoordVals, maxLength, CoordVals[1])::CoordLambert93[])
);
```

Procédure traitement():

La fonction traitement() repose principalement sur les expressions régulières pour extraire et nettoyer les informations. Nous pouvons accélérer ces opérations avec les indexes trigrammes. Ces index se basent sur la recherche approximative des chaînes de caractères qui correspondent à un motif approximatif plutôt qu'à une correspondance exacte.

Voici un extrait du code dans cette fonction:

```
ReferenceCindocVals.field := string_to_array(NEW.ReferenceCindoc, '|');
        DiscriminantVals.field := string_to_array(NEW.Discriminant, '|');
        NEW.Ville := regexp_replace(NEW.Ville, '--', '-');
        VilleVals.field := pretty(string_to_array(regexp_replace(NEW.Ville,
'\s', '', 'g'), ','));
        SujetVals.field := string_to_array(NEW.sujet, ',');
        DateVals.field := traitement_date(NEW.Date);
        NoteBPVals.field := split_string(NEW.NoteBP, '|', '/');
        FicNumVals.field := split_string(
            regexp_replace(NEW.FicNum, '(.*)\.(.*)', '\1'), '|', '/');
        Idx_PerVals.field := traitement_oeuvre(NEW.Idx_per);
        Idx_IcoVals.field := split_string(lower(NEW.Idx_ico), '|', '/');
        Idx_IcoVals.field := ARRAY(SELECT unnest(string_to_array(a, ','))
FROM unnest(Idx_IcoVals.field) a);
```

```
        NbrCliVals.field := split_string(NEW.NbrCli, '|', '/');
        NbrCliVals.field := ARRAY(SELECT unnest(string_to_array(d, ','))
FROM unnest(NbrCliVals.field) d);
        TailleCliVals.field := traitement_cliche(NEW.TailleCli);
        N_VVals.field := traitement_n_v(string_to_array(NEW.n_v, ','));
        C_GVals.field := traitement_c_g(string_to_array(NEW.c_g, ','));
        CoordVals := traitement_coord(VilleVals.field);
```

Nous allons indexer seulement les colonnes concernées, dès l'importation des données dans la table CorrectionTemp.

```
1   DROP INDEX IF EXISTS index_trgm_correction_temp;
2   CREATE INDEX index_trgm_correction_temp ON CorrectionTemp USING gin(
3       ReferenceCindoc gin_trgm_ops,
4       Discriminant gin_trgm_ops,
5       Ville gin_trgm_ops,
6       Sujet gin_trgm_ops,
7       Date gin_trgm_ops,
8       FicNum gin_trgm_ops,
9       Idx_per gin_trgm_ops,
10      Idx_ico gin_trgm_ops,
11      NbrCli gin_trgm_ops,
12      TailleCli gin_trgm_ops,
13      n_v gin_trgm_ops,
14      c_g gin_trgm_ops
15  );
```

Les coordonnées Lambert93:

```
1   DROP FUNCTION IF EXISTS traitement_coord;
2   CREATE OR REPLACE FUNCTION traitement_coord( villeArr anyarray)
3   RETURNS CoordLambert93[] AS $$
4       DECLARE
5           res CoordLambert93 ARRAY;
6           row CoordLambert93;
7           i int;
8           CoordX numeric;
9           CoordY numeric;
10          CodePostal varchar;
11      BEGIN
12          FOR i IN 1..coalesce(array_length(villeArr, 1), 0) LOOP
```

```
            SELECT INTO CodePostal viCodePostal
            FROM VilleTemp
            WHERE viVille ILIKE villeArr[i];

            SELECT viLambertX, viLambertY INTO CoordX, CoordY
            FROM VilleTemp
            WHERE viCodePostal = CodePostal;

            row.CoordX := CoordX;
            row.CoordY := CoordY;

            res = array_append(res, row);
        END LOOP;
        RETURN res;
    END;
$$ LANGUAGE plpgsql;
```

Nous avons importé une table de données externe qui contient toutes les informations concernant les villes. Nous cherchons à optimiser la performance des requêtes SELECT surlignés ci-dessus. On veut trouver rapidement le code postal à partir du nom de la ville (regexp sur le nom), les coordonnées lambert à partir du code postal. Comme on ne regarde qu'une minorité des données, on n'est sûr que PostgreSQL va préférer un index scan qu'un seq scan, qui présente plus d'avantage.

```
CREATE INDEX idx_ville_nom_cp ON VilleTemp(viVille, viCodePostal);
CREATE INDEX idx_ville_cp_coords ON VilleTemp(viCodePostal, viLambertX, viLambertY);
CREATE INDEX idx_ville_patterns ON VilleTemp USING gin(viVille gin_trgm_ops);
```

Requêtes:

La photo dans laquelle figure une personne. (fonction)

```
DROP FUNCTION IF EXISTS chercher_photo_par_personne;
CREATE OR REPLACE FUNCTION chercher_photo_par_personne(str varchar)
RETURNS TABLE(Article integer, Discriminant integer, NomOeuvre varchar) AS $$
   BEGIN
       RETURN QUERY(
           SELECT DISTINCT PhotoArticle, D.Discriminant, I.NomOeuvre FROM
           Document D JOIN IndexPersonne I on D.idOeuvre=I.idOeuvre
           WHERE I.NomOeuvre LIKE '%' || str || '%'
```

```
9         );
10     END;
11 $$ language plpgsql;
```

Dans cette requête, aucune optimisation n'a été effectuée car toutes les tentatives n'ont pas conclu en un gain de temps ou de coût. En revanche, nous avons remarqué que cette requête contenait des lignes redondantes dû à l'absence de `Discriminant` dans la sélection de la requête. Ce point a donc été modifié. Nous avons également ajouté un `DISTINCT` car nous nous sommes aperçus que la jointure des tables `IndexPersonne` et Document faisaient apparaître des lignes qui n'avaient pas lieu d'être sélectionnées.

Les photos qui présentent un sujet. (fonction)

```
1  DROP FUNCTION IF EXISTS chercher_photo_par_sujet;
2  CREATE OR REPLACE FUNCTION chercher_photo_par_sujet(str varchar)
3  RETURNS TABLE(PhotoArticle integer, Discriminant integer, NomOeuvre
4  varchar) AS $$
5     BEGIN
6         RETURN QUERY(
7             SELECT DISTINCT D.PhotoArticle, D.Discriminant, S.DescSujet FROM
8             Document D JOIN Sujet S on D.idSujet = S.idSujet
9             WHERE S.DescSujet ILIKE '%' || str || '%'
10        );
11    END;
12 $$ language plpgsql;
```

Dans cette requête, aucune optimisation n'a été effectuée car toutes les tentatives n'ont pas conclu en un gain de temps ou de coût. En revanche Nous avons remarqué que cette requête contenait des lignes redondantes dû à l'absence de Discriminant dans la sélection de la requête. Ce point a donc été modifié. Nous avons également ajouté un `DISTINCT` car nous nous sommes aperçus que la jointure des tables Sujet et Document faisaient apparaître des lignes qui n'avaient pas lieu d'être sélectionnées.

Les photos qui ont un fichier numérique (view)

```
1  DROP INDEX IF EXISTS index_ficnum;
2  CREATE INDEX index_ficnum ON Document(FicNum);
3  CREATE OR REPLACE VIEW view_photo_numeric AS
4  SELECT DISTINCT(Article), Remarques, NbrCli, DescDet, idSerie
5  FROM Photo P JOIN Document D ON P.Article = D.PhotoArticle
```

```
6   WHERE d.FicNum IS NOT NULL;
```

Pour cette vue, l'optimisation réside dans la création d'un index sur l'attribut `FicNum` de la table `Document`. En effet, il existe environ 3300 fichiers numériques pour 34000 articles différents. Comme le pourcentage des articles ayant un fichier numérique est d'environ 9% (une partie minime des données), PostgreSQL utilisera un index si il en a un à disposition. Ainsi on passe d'un coût total de 4583.24..4834.88 à 3822.47..4074.11, ce qui n'est pas négligeable !

Autres

Pour toutes les fonctions ou vues créés qui n'ont pas été mentionnées, aucune modification n'a été apportée car

- aucun index ne permettait de gagner du temps ou de baisser le coût de cette dernière
- l'ensemble des opérateurs utilisés le sont à bon escient, on ne peut pas les remplacer par d'autres afin de gagner du temps
- les fonctions ou vues non citées ne comportaient pas d'erreur notables.
- les fonctions ou vues non citées sont des dérivées d'autres fonctions ou vues pour lesquelles on sélectionne simplement les résultats qu'on attend.
- La création d'un index afin d'optimiser la fonction ou la vue en question devait se faire sur une table ne contenant qu'une valeur. L'ajout de cet index n'est alors nécessaire que si on prévoit d'agrandir la base. (exemple : fonction `rechercher_serie_photo` avec la table série qui ne contient qu'un tuple)

Conclusion:

Le nettoyage de données est une opération coûteuse: les opérations prennent 5 - 35 minutes. Pour réduire le temps de développement, nous avons cherché à optimiser chaque requête écrit. Finalement, on a obtenu un temps de traitement considérablement réduit.