Où et comment sont stockées mes données avec PostGreSQL ?

Comme tous les SGBD relationnel, PostGreSQL stocke les données des tables et des index dans des fichiers organisés sous forme de pages. Que les données soient dans le cache (donc en mémoire vive), ou bien sur le disque, ces pages sont structurées d'une manière bien particulière afin d'exploiter au mieux les lectures et écritures physiques et logiques ainsi que la gestion des versions de ligne.
L'ensemble des pages figurent dans des fichiers.
Cet article a pour but de vous présenter comment et où PostGreSQL stocke les données et compare sa façon de faire aux autres SGBDR que sont Oracle et MS SQL Server.
Mais contrairement à ses grands frères, PostGreSQL ne dispose pas d'une gestion des espaces de stockage... Voyons ce que cela induit notamment sur le plan des performances et de la volumétrie.

Par Frédéric Brouard, Expert SGBDR et SQL - sqlpro@club-internet.fr
auteur de : « SQL », collection Synthex - Pearson Education, 4e édition Paris 2012
co auteur de « UML pour les bases de données » - Eyrolles Paris 2012
Enseignant CNAM PACA, ISEN TOULON, CESI/EXIA Aix en Provence et Lyon

1 - Quelques notions fort utiles

Rappelons tout d'abord quelques notions.
Une **table** est un ensemble de lignes et comme tout ensemble, les lignes d'une table n'ont aucun ordre particulier. On parle alors de « tas » ou *heap* en anglais.
Un **index** est une copie de certaines colonnes (au moins une) d'une table organisée dans une structure particulière permettant d'accélérer certaines recherches. Dans ce cas, les données d'un index sont ordonnées suivant l'algorithme de recherche spécifique au type d'index (arbre équilibré, hachage, matrice de bit...).
Toutes les données, que ce soit d'un index ou d'une table, sont stockées dans un ensemble de **pages** dont la structure est identique. La taille des pages de données de PostGreSQL est de 8 Ko, comme pour MS SQL Server, alors que pour Oracle, cette taille est variable, mais uniforme pour une même

base et peut être spécifiée à la création de la base afin d'optimiser le stockage en fonction des OS et du sous-système disque.
Ces pages figurent dans les **fichiers** et dans postGreSQL, chaque table est représentée par au moins un fichier, voire plusieurs s'il y a des index.
Les fichiers des tables sont rangés dans un ou plusieurs **espaces de stockage** qui est un pointeur logique vers un répertoire du sous-système disque. Les espaces de stockage sont appelés *tablespace*, tout comme dans Oracle, bien que cette dénomination soit impropre puisque dans ces espaces de stockage figurent aussi bien des tables que des index. Notons que SQL Server, tout comme Sybase ASE parle de *filegroup*, c'est à dire groupe de fichier pour désigner un espace de stockage.
PostGreSQL appelle « **relation** » (abrégé « rel ») tout objet de type, table, vue, index, type composé... Ce qui à nouveau est impropre, car une relation au sens mathématique du terme (algèbre relationnelle) est l'objet conteneur de données doté d'une clef et dont tous les n-uplets sont valués (contrairement à une table qui accepte les NULLs).
Dans le même genre, PostgreSQL parle de **namespace** pour désigner ce que SQL appelle un schéma et on trouve tantôt l'un tantôt l'autre dans la documentation comme dans les informations de méta données (par exemple dans la table système pg_class on parle de relnamespace, mais dans les vue d'information de schéma on parle de TABLE_SCHEMA...).
Un « **tuple** » (en français *n-uplet*), est pour PostGreSQL, une ligne d'une table possédant au plus une valeur pour chaque colonne de la table, et là encore le terme est impropre car il se réfère en principe à la relation au sens mathématique et non à la table pour laquelle on doit parler de ligne.
Enfin un **oid** est un Object IDentifier, c'est à dire un entier qui identifie de manière unique tout élément (base, table, vue, index, contrainte...) dans une instance de postGreSQL. À noter, dans la plupart des tables système de postGreSQL, figure une colonne oid, la plupart du temps non décrite.

2 - Les espaces de stockage

Dans postGreSQL, les espaces de stockage sont communs à l'instance (une installation de machine que postGreSQL appelle « cluster »). On peut en obtenir la liste et le chemin de chacun d'eux, à l'aide de la requête suivante (voir listing 1) :

```
SELECT DISTINCT T.oid AS table_space_oid,
       spcname AS table_space_name,
       CASE spcname
          WHEN 'pg_default'
             THEN setting || '/base/'
          WHEN 'pg_global'
             THEN setting || '/global/'
          ELSE spclocation
       END AS location
FROM   pg_tablespace AS T
       CROSS JOIN pg_settings AS S
WHERE  S.name = 'data_directory' ;
```

Listing 1 - requête pour obtenir la liste des espaces de stockage

Ce qui, chez moi, donne (voir tableau 1) :

table_space_oid	table_space_name	location
1664	pg_global	C:/Program Files/PostgreSQL/9.1/data/global/
1663	pg_default	C:/Program Files/PostgreSQL/9.1/data/base/
25314	ts_mes_index	C:/PG_databases/mesIndex
25310	ts_mes_data	C:/PG_databases/MaBase

Tableau 1 - liste des espaces de stockage de l'instance PostGreSQL

"pg_default" pointe vers répertoire vers lequel chaque nouvelle base créée un sous répertoire dont le nom est l'oid de la base. "pg_global" est un tablespace contenant les objets communs à toutes les

bases et réservé aux objets systèmes, telle que la table des bases de données (pg_database) ou la table des espaces de stockage (pg_tablespace).
Notez que dans la version 9.2, vous disposez de la fonction pg_tablespace_location(oid) afin de renvoyer l'emplacement des fichiers de l'objet spécifié par son oid.

Il est possible de créer ses propres espaces de stockage à l'aide de la commande :

```
CREATE TABLESPACE nom_tablespace
[ OWNER nom_utilisateur ]
LOCATION 'répertoire' ;
```

La création d'un tablespace PostGreSQL ne permet de définir que l'emplacement et le propriétaire de l'espace de stockage. Dans Oracle ou SQL Server, chaque espace de stockage peut être doté de fichiers ou seront stockées toutes les tables et index (un même fichier stockant plusieurs objets) et l'on peut spécifier si l'espace est en mode read/write ou read only.

L'exemple suivant montre la création d'un nouveau tablespace et l'affectation d'une table dans ce nouveau tablespace lors de la création de la table :

```
CREATE TABLESPACE TS_mes_data LOCATION 'C:\PG_databases\MaBase\';

CREATE TABLE T_ma_table (ma_colonne INT) TABLESPACE TS_mes_data;
```

Vous noterez que PostGreSQL, créé dans le répertoire spécifié un sous répertoire nommé PG_ ???_ !!!!!!!!! ou ??? est le numéro de version de PostGreSQL, par exemple 9.1 et !!!!!!!!! le numéro de version du catalogue (ensemble des tables système) de votre instance PostGreSQL, et cette dernière information peut être retrouvée par la commande pg_controldata en ligne de commande.

De la même manière, on peut créer un index en spécifiant un tablespace particulier :

```
CREATE TABLESPACE TS_mes_index LOCATION 'D:\PG_databases\mesIndex\';

CREATE INDEX X_ma_table_ma_colonne
   ON T_ma_table (ma_colonne) TABLESPACE TS_mes_index;
```

Notez que par sécurité il n'est pas possible de créer une table dans l'espace de stockage pg_global, et c'est bien normal car il est réservé aux évolutions futures des versions de PostgreSQL qui peuvent y rajouter à tout moment de nouveaux objets.

L'espace de stockage pg_default contenant à défaut toutes les tables de toutes les bases, l'ensemble des tables de chaque base est cloisonné des autres par un sous répertoire dont le nom est l'OID de la base. La requête suivante (voir listing 2) donne le chemin final de pg_default pour toutes les bases :

```
SELECT D.datname AS database_name,
       T.oid AS table_space_oid,
       spcname AS table_space_name,
       setting || '/base/' || CAST(D.oid AS VARCHAR(16)) AS location
FROM   pg_tablespace AS T
       CROSS JOIN pg_settings AS S
       CROSS JOIN pg_database AS D
WHERE  S.name = 'data_directory'
AND    spcname = 'pg_default'
ORDER BY 1, 2;
```

Listing 2 - Requête donnant le chemin final des espaces de stockage par défaut pour toutes les bases
Ce qui, chez moi, donne (voir tableau 2) :

database_ name	table_ space_oid	table_ space_name	location
DB_TEST	1663	pg_default	C:/Program Files/PostgreSQL/9.1/data/base/16594
postgres	1663	pg_default	C:/Program Files/PostgreSQL/9.1/data/base/11913
template0	1663	pg_default	C:/Program Files/PostgreSQL/9.1/data/base/11905
template1	1663	pg_default	C:/Program Files/PostgreSQL/9.1/data/base/1
toto	1663	pg_default	C:/Program Files/PostgreSQL/9.1/data/base/25317

Tableau 2 - liste des espaces de stockage par défaut pour toutes les bases

Voici maintenant une requête donnant l'ensemble des espaces de stockage et des chemins pour la base courante (voir Listing 3).

```
SELECT DISTINCT T.oid AS table_space_oid,
       spcname AS table_space_name,
       CASE spcname
          WHEN 'pg_default'
             THEN setting || '/base/' || CAST(D.oid AS VARCHAR(16))
          WHEN 'pg_global'
             THEN setting || '/global/'
          ELSE spclocation
       END AS location
FROM   pg_tablespace AS T
       CROSS JOIN pg_settings AS S
       CROSS JOIN pg_database AS D
WHERE  S.name = 'data_directory'
  AND  spcname <> 'pg_global'
  AND  D.datname = current_database();
```

Listing 3 - Requête donnant l'ensemble des espaces de stockage et des chemins pour la base courante

Pour toutes ces requêtes, vous aurez noté que les chemins sont donnés avec une barre oblique et non une barre oblique inverse. Dans le cas où vous seriez sous Windows, vous pouvez appliquer la fonction REPLACE comme ceci : REPLACE(location, '/', '\').

Enfin, les tables systèmes pg_tables et pg_indexes permettent d'obtenir des informations au niveau logique sur les tables et les index, en particulier, le schéma SQL, le propriétaire, mais aussi l'espace de stockage.

CONSEIL...

Séparer les tables et les index en différents tablespaces est une bonne chose dès que la base atteint une certaine volumétrie, car cela permet de ventiler les opérations physiques d'entrée sortie (IO pour *Input Output*), à condition que chaque tablespace figure sur un disque physique différent. Néanmoins, PostGreSQL, à la différence de ses grands frères que sont Oracle ou SQL Server, ne sait pas effectuer des opérations parallélisées pour une même requête. Dès lors l'impact sur les performances de cette technique ne peut concerner que des requêtes exécutées concurremment. Pour pallier ce manque, il est possible de jouer sur la structure physique des disques en les agrégeant notamment par le biais du RAID 0 ou plus précisément par un niveau de RAID combinant stripping et redondance, chose que l'on trouve fréquemment sur les SAN.

NOTA : nous avons vu quelques tables systèmes (catalogue) c'est pg_tablespace qui renseigne sur les espaces de stockage, pg_settings qui stocke les principaux paramètres de l'instance PostGreSQL et pg_database qui contient la liste des bases de données.

3 - Les fichiers

Nous avons dit qu'une table était constituée de pages de 8 Ko contenues dans un fichier. Mais il existe différents fichiers pour une même table. Le fichier principal d'une table est nommé par l'oid de la table et figure dans le tablespace par défaut de la base ou bien dans un tablespace spécifique. Les fichiers contenant les données de PostGreSQL sont limités à 1 Go. Si votre table ou votre index dépasse cette taille, alors PostGreSQL en créé plusieurs et les chaîne.
Mais attention, lors des opérations TRUNCATE, REINDEX, CLUSTER et pour certaines options d'ALTER TABLE, PostGreSQL modifie le nom du fichier (filenode) tout en préservant l'oid. Il est alors nécessaire de retrouver le véritable fildenode à l'aide de la fonction pg_relation_filenode().
Pour les tables temporaires, le nom du fichier est différent. Il commence par un numéro dont la valeur est l'identifiant du processus qui a créé la table (un genre de n° de session) suivi d'un blanc souligné et de l'oid.

Pour savoir où sont stockés chacun des objets de données de votre base, vous pouvez utiliser la requête présentée dans le Listing 4.

```
-- situation des fichiers des tables (hors objets PG)
SELECT  C.oid,
        CASE C.relkind
          WHEN 'r' THEN 'TABLE'
          WHEN 'i' THEN 'INDEX'
          WHEN 't' THEN 'TOAST'
       END AS object_type, N.nspname AS object_schema,
       C.relname AS object_name,
       CASE
          WHEN COALESCE(T.spclocation, '') = ''
             THEN current_setting('data_directory') || '/'
                  || pg_relation_filepath(C.oid)
          ELSE replace(pg_relation_filepath(C.oid),
                       'pg_tblspc/' || CAST(T.oid AS VARCHAR(32)),
                      T.spclocation)
       END AS object_file_location
FROM   pg_class AS C
       CROSS JOIN pg_settings AS S
       LEFT OUTER JOIN pg_tablespace AS T
             ON C.reltablespace = T.oid
       INNER JOIN pg_namespace AS N
             ON C.relnamespace = N.oid
WHERE S.name = 'data_directory'
  AND C.relkind IN ('r', 'i', 't')
  AND N.nspname NOT IN ('pg_catalog', 'information_schema')
  AND C.oid NOT IN
      (SELECT C1.relTOASTrelid
       FROM   pg_class AS C1
              INNER JOIN pg_namespace AS N
                    ON C1.relnamespace = N.oid
       WHERE  N.nspname IN ('pg_catalog',
                            'information_schema')
       UNION  ALL
       SELECT C1.relTOASTidxid
       FROM   pg_class AS C1
              INNER JOIN pg_class AS C0
                    ON C1.oid = C0.relTOASTrelid
              INNER JOIN pg_namespace  AS N
                    ON C0.relnamespace = N.oid
       WHERE  N.nspname IN ('pg_catalog',
                            'information_schema'))
ORDER BY 1
```

Listing 4 - Requête donnant l'emplacement de stockage de chaque des objets de données de la base courante

Quelques petites explications sur cette requête : les objets contenant des données sont, des tables (relkind = 'r'), des index (relkind = 'i') et des extensions de tables (relkind = 't') appelées TOAST (The Oversized-Attribute Storage Technique) destinés à stocker les LOB (Large OBjects) or des lignes relationnelles de la table, le tout figurant dans une table système de nom pg_class.
Les objets d'une base figurant dans un schéma, nous avons fait une jointure avec la table système pg_namespace qui liste les schémas SQL.
La fonction pg_relation_filepath donne un chemin relatif avec un lien symbolique (ou jonction sous Windows) spécifique à PostGreSQL et déterminé par PGDATA/pg_tblspc, pour les tables situées dans l'espace de stockage par défaut (pg_default). Pour retrouver le chemin réel, il faut se livrer à une gymnastique peu évidente entre les différentes chaines de caractères.
Pour éviter de lister les objets systèmes internes contenus dans chaque base de données, nous nous sommes interdits de scruter les schémas SQL pg_catalog et information_schema. Cependant, tout ce qui est TOAST, venant d'une table système, comme d'une table de production figure dans le schéma TOAST ce qui rend difficile de savoir quels sont les objets TOASTs afférent à des tables ou des index non systèmes, c'est pourquoi la dernière restriction de la clause WHERE porte sur un NOT IN qui encapsule deux requêtes concaténées par une union, afin de retrouver les oid des tables et index TOAST afférent aux objets systèmes.
Voici un extrait de ce que donne cette requête sur ma base (voir tableau 3).

oid	object_ type	object_ schema	object_name	object_file_location
25311	TABLE	public	t_ma_table	C:/PG_databases/MaBase/PG_9.1_201105231/16594/25311
25319	INDEX	public	x_ma_table_ma_colonne	C:/PG_databases/mesIndex/PG_9.1_201105231/16594/25319
25320	TABLE	public	ttttttt	C:/Program Files/PostgreSQL/9.1/data/base/16594/25320
25323	TOAST	pg_TOAST	pg_TOAST_25320	C:/Program Files/PostgreSQL/9.1/data/base/16594/25323
25325	INDEX	pg_TOAST	pg_TOAST_25320_index	C:/Program Files/PostgreSQL/9.1/data/base/16594/25325
25326	INDEX	public	ttttttt_pkey	C:/Program Files/PostgreSQL/9.1/data/base/16594/25326

Tableau 3 - emplacement de stockage de chaque des objets de données d'une courante (extrait)

Voici maintenant différentes requêtes enchainées par un UNION ALL permettant d'obtenir la taille des bases, des espaces de stockage, des objets de la base courante (voir Listing 5).

```
SELECT '0-DATABASE' AS object_type,
       oid, datname AS object_name,
       pg_database_size(oid)/1024 AS size_KO
FROM   pg_database
UNION  ALL
SELECT '1-TABLE_SPACE' AS object_type,
       oid, spcname,
       pg_tablespace_size(oid)/1024
FROM pg_tablespace
UNION ALL
SELECT CASE relkind
          WHEN 'r' THEN '2-TABLE'
          WHEN 't' THEN '4-TOAST'
          WHEN 'i' THEN '3-INDEX'
          WHEN 'S' THEN '5-SEQUENCE'
       END,
       C.oid,
       current_database() || '.'
       || N.nspname || '.' || relname,
       pg_relation_size(C.oid)/1024
FROM   pg_class AS C
       INNER JOIN pg_namespace AS N
            ON C.relnamespace = N.oid
WHERE  relkind IN ('r', 't', 'i', 'S')
UNION ALL
SELECT '6-DATA',
       C.oid,
       current_database() || '.'
```

```
       || N.nspname || '.' || relname,
       pg_total_relation_size(C.oid)/1024
FROM   pg_class AS C
       INNER JOIN pg_namespace AS N
            ON C.relnamespace = N.oid
WHERE  relkind = 'r'
ORDER BY 1, 3
```

Listing 5 - requêtes obtenant la taille des bases, des espaces de stockage et des objets de la base courante

Les fonctions pg_database_size, pg_tablespace_size, pg_relation_size et pg_total_relation_size permettre de connaitre la taille en octet et peuvent prendre en paramètre aussi bien l'oid de l'objet que son nom (complet, c'est à dire avec le schéma SQL lorsque c'est un objet de la base : table, TOAST, index, séquence). La fonction complémentaire, pg_size_pretty convertit un nombre d'octets dans un format littéral plus lisible (KB , MB, GO, TB...).
La fonction pg_total_relation_size cumule les données de la table, les index et les données externes dans des tables TOAST.

En sus des fichiers contenant les données, il existe des fichiers de nom <oid>_vm (pour VacuuM) qui permettent de savoir pour une table donnée quelles sont les pages ayant des lignes mortes afin que le processus VACUUM puisse y accéder directement.
Les fichiers de nom <oid>_fsm (pour Free Space Map) permet de savoir pour une table ou un index BTree, quelles sont les pages ayant des emplacements de ligne libre.
Les tables non journalisées (en général des tables temporaires) et les index desdites tables disposent d'un fichier d'initialisation de nom <oid>_init, afin de permettre la reconstruction de la table en cas de crash du serveur.
Enfin, dans chaque base figure un fichier de nom pg_filenode.map qui fournit la cartographie transitoire des filenode en cas d'évolution suite aux commandes TRUNCATE, REINDEX, CLUSTER, ALTER TABLE... mais aussi pour certaines tables systèmes du catalogue qui à l'évidence ne peuvent pas figurer dans les données de la table système, comme la table pg_class qui, justement, contient les données des filnode de tous les autres objets.

Afin de limiter l'utilisation de la mémoire, PostGreSQL ouvre et ferme dynamiquement les fichiers afférents aux tables ou aux index en fonction des opérations physiques à réaliser, ceci afin de ne pas avoir trop de descripteurs de fichiers ouverts simultanément. Et comme dans tous les bons SGBDR, seul le journal est écrit de manière synchrone (à moins de désactiver cette synchronisation au risque de rendre la base corrompue en cas de crash) et les données de temps à autre par l'intermédiaire d'un processus d'arrière-plan le BGWRITER qui profite des moments d'inactivité pour effectuer les écritures physique, ou bien à l'aide de la commande CHECKPOINT, et bien entendu à condition que les pages sales (celles ayant évoluées en mémoire et pas encore écrites sur le disque) aient été définitivement validées par la transaction.
Mais ces opérations d'ouverture et de fermeture des fichiers laissent les fichiers à la merci de n'importe quel utilisateur mal intentionné, comme à un utilisateur maladroit qui pourrait, s'il possède les droits d'accès, supprimer intentionnellement ou non certains fichiers de certaines base.
Dans ce cas, la base serait signalée comme corrompue et les données définitivement perdues seraient susceptibles d'entacher d'autres objets du fait du chainage de l'intégrité référentielle comme de la référence de tels objets au sein des vues et des routines par exemple.
Voici ce qui se passerait dans un tel scénario... Créons d'abord quelques tables et insérons quelques données (voir listing 6).

```
CREATE TABLE T_DELETE1
(DEL_ID     SERIAL NOT NULL PRIMARY KEY,
 DEL_DATA1   VARCHAR(200));
INSERT INTO T_DELETE1 (DEL_DATA1)
```

```
VALUES ('aaaa'), ('bbbb'), ('cccc');

 CREATE TABLE T_DELETE2
(DEL_ID     INT NOT NULL PRIMARY KEY
            REFERENCES T_DELETE1 (DEL_ID),
 DEL_DATA2  VARCHAR(200))

INSERT INTO T_DELETE2 (DEL_ID, DEL_DATA2)
VALUES (1, 'AAAA'), (2, 'BBBB');
```

Listing 6 - Création d'objets tests

Récupérons la référence de fichier en s'aidant de la requête suivante (voir listing 7).

```
-- récupération de l'oid de la base et de la table
-- en vue de supprimer le fichier
SELECT C.oid AS table_oid, D.oid AS database_oid
 FROM   pg_class AS C
       CROSS JOIN pg_database AS D
WHERE  relname = 't_delete1'
  AND  datname = current_database()
  -- supprimez le fichier dans l'espace par défault de la base,
  -- par exemple :
  -- ...\PostgreSQL\9.1\data\base\<database_oid>\<table_oid>
```

Listing 7 - récupération des oid de la base et de la table à supprimer

Des opérations de manipulation de tables sont-elles encore possibles ? (voir listing 8)

```
-- lecture des données
SELECT * FROM T_DELETE1;
-- le système accepte

 -- insertion d'une ligne dans la table fille
INSERT INTO T_DELETE2 (DEL_ID, DEL_DATA2)
VALUES (3, 'CCCC');
-- l'insertion s'effectue sans broncher
```

Listing 8 - Requêtes de test après suppression physique d'un fichier de table

Nous pouvons lire la table et même insérer une ligne dans la table fille qui vérifie la référence... C'est normal, toutes nos tables sont dans le cache. Mais en forçant les écritures physiques à l'aide de la commande CHECKPOINT, nous obtenons le message d'erreur suivant :

> *ERREUR: échec de la demande de point de vérification*
> *HINT: Consultez les messages récents du serveur dans les journaux applicatifs pour plus de détails.*
> *********** Erreur **********
> *État SQL :XX000*

Dans le fichier de suivi des opérations, situé dans l'arborescence de PostGreSQL (...PostgreSQL\9.1\data\pg_loge\...) on trouve dans le fichier le plus récent, les informations suivantes, répétées :

> *2012-04-21 17:35:50 CEST INSTRUCTION : CHECKPOINT*
> *2012-04-21 17:35:59 CEST ERREUR: n'a pas pu ouvrir le fichier « base/16594/25367 » : No such file or directory*
> *2012-04-21 17:35:59 CEST CONTEXTE : écriture du bloc 0 de la relation base/16594/25367*
> *2012-04-21 17:35:59 CEST ATTENTION: n'a pas pu écrire le bloc 0 de base/16594/25367*
> *2012-04-21 17:35:59 CEST DÉTAIL: Échecs multiples --- l'erreur d'écriture pourrait être permanent.*

Il convient de faire donc très attention à restreindre au maximum les accès aux répertoires contenant les données des bases PostGreSQL afin que le malhonnête ou l'étourdi n'endommage pas votre base de données...

CRITIQUE :

PostGreSQL ne comporte pas de routines de bas niveau pour la gestion des fichiers et confie cela à l'OS, contrairement à ce qui se passe pour Oracle ou MS SQL Server, pour lesquels il est, par exemple, possible de dimensionner largement les fichiers constitutifs des espaces de stockage afin d'éviter des opérations de croissance ou les placer en lecture seule.
De la même manière il n'est pas possible de choisir l'emplacement des fichiers sur le disque alors qu'Oracle ou SQL Server auditent le disque pour trouver l'emplacement optimal de stockage des fichiers (et ce d'une seul tenant afin d'éviter la fragmentation physique), généralement les bords externes des plateaux du disque (cylindres extérieurs).

4 - Taille effective des données

Vous pouvez aussi utiliser la fonction pg_column_size pour obtenir la taille d'une donnée dans une colonne. Voici un exemple d'utilisation qui va nous faire comprendre quel est le coût du stockage des données dans postGreSQL pour les chaînes de caractères (voir Listing 9).

```
-- test de la fonction taille
CREATE  TABLE T_TEST_TAILLE
(TTT_ID            SERIAL NOT NULL PRIMARY KEY,
 TTT_DATAFIX       CHAR(16),
 TTT_DATAVAR       VARCHAR(16),
 TTT_DATAVARBIG    VARCHAR(6000));

-- insertion de quelques lignes
INSERT INTO T_TEST_TAILLE (TTT_DATAFIX, TTT_DATAVAR, TTT_DATAVARBIG)
           SELECT     NULL, NULL, NULL
UNION ALL  SELECT '', '', ''
UNION ALL  SELECT '12345678', '12345678', REPEAT('*', 3000)
UNION ALL  SELECT '1234567890123456', '1234567890123456', REPEAT('*', 6000)

-- insertion d'une très longue chaîne constituée de caractères aléatoires
WITH RECURSIVE
S AS (SELECT generate_series AS I FROM generate_series(1, 6000)),
T AS (SELECT I, CHR(20 + CAST(CEILING(RANDOM()*236) AS SMALLINT)) AS C
      FROM   S),
R AS (SELECT CAST(C AS VARCHAR(6000)) AS CC, C, I
      FROM   T
      WHERE  I = 1
      UNION ALL
      SELECT CAST(CC || T.C AS VARCHAR(6000)),T.C,  T.I
      FROM   T
             INNER JOIN R
                   ON T.I = R.I + 1)
INSERT INTO T_TEST_TAILLE (TTT_DATAVARBIG)
SELECT CC
FROM   R
WHERE  I = 6000;

-- requête visionnant les données et leurs longueurs
SELECT ttt_id, pg_column_size(TTT_DATAFIX)    AS fix_col_size,
               char_length(TTT_DATAFIX)       AS fix_chr_len,
               octet_length(TTT_DATAFIX)      AS fix_oct_len,
               pg_column_size(TTT_DATAVAR)    AS var_col_size,
               char_length(TTT_DATAVAR)       AS var_chr_len,
```

```
               octet_length(TTT_DATAFIX)      AS var_oct_len,
               pg_column_size(TTT_DATAVARBIG) AS big_col_size,
               char_length(TTT_DATAVARBIG)    AS big_chr_len,
               octet_length(TTT_DATAVARBIG)   AS big_oct_len
FROM   T_TEST_TAILLE;
```

Listing 9 - Script SQL pour le test des différentes mesures de taille des colonnes d'une table

Cette dernière requête donne les résultats que voici (voir tableau 4) :

	CHAR(16)			VARCHAR(16)			VARCHAR(6000)		
ttt_id	**col_ size**	**chr_ len**	**oct_ len**	**col_ size**	**chr_ len**	**oct_ len**	**col_ size**	**chr_ len**	**oct_ len**
1									
2	17	0	16	1	0	16	1	0	0
3	17	8	16	9	8	16	44	3000	3000
4	17	16	16	17	16	16	78	6000	6000
5							9249	6000	9249

Tableau 4 - Resultat des tests des différentes mesures de taille des colonnes d'une table.

Une colonne sans données n'est pas stockée (ttt_id = 1).
Une colonne de type CHAR(*n*) coute 1 octet de plus que la longueur *n* spécifiée.
Une colonne de type VARCHAR(n) coute 1 octet de plus que la longueur de la donnée stockée.
À partir d'une certaine taille, que ce soit du CHAR ou du VARCHAR, postGreSQL utilise un algorithme de compression. Mais cette compression peut s'avérer néfaste lorsque la donnée à stocker est fortement aléatoire (ttt_id = 5, ici, la compression accroît la taille de la donnée de plus de 50% !).
Cela dit, il est rare d'avoir des informations à stocker aussi aléatoire que celle de notre test. En pratique la compression est plutôt un bon point, à condition d'éviter de véhiculer trop tôt dans les requêtes de telles colonnes !

5 - Pages

Les pages d'une base de données constituent l'unité minimale d'entrée/sortie (IO pour Input / Output) pour gérer les lectures et écritures tant en mémoire cache que sur le disque, la représentation des données sous forme de pages étant la même du côté physique (disque) que logique (RAM). Comme les SGBDR de types Client/Server travaillent exclusivement en mémoire, lire une seule donnée non encore présente dans le cache de données, revient à monter en RAM la page figurant sur le disque et qui contient la donnée demandée.

Nous avons dit que les pages de PostGreSQL sont de taille fixe et font 8 Ko, tout comme celles de SQL Server. La structure interne des pages est similaire pour tous les SGBDR. Une page n'est affectée qu'à un seul objet (une table ou un index). La page commence par un entête contenant des données techniques, le reste étant affecté aux données. Mais comme les lignes peuvent être de tailles différentes compte tenu des types VARCHAR, du NULL et de la compression, alors il est nécessaire de savoir à quel endroit commence chaque ligne. Pour cela figure un tableau des pointeurs (offset) d'emplacement (slot) de ligne (voir figure 1). Dans PostGreSQL, ce tableau est situé en haut de la page et les lignes sont construites à partir du bas, comme dans oracle. Dans SQL Server c'est l'inverse : le tableau figure en bas et les lignes en haut. Enfin, seules les pages d'index ont en supplément un espace en pied de page de taille spécifique au type d'index et contenant des données spécifiques aux index, permettant notamment de chainer les pages.

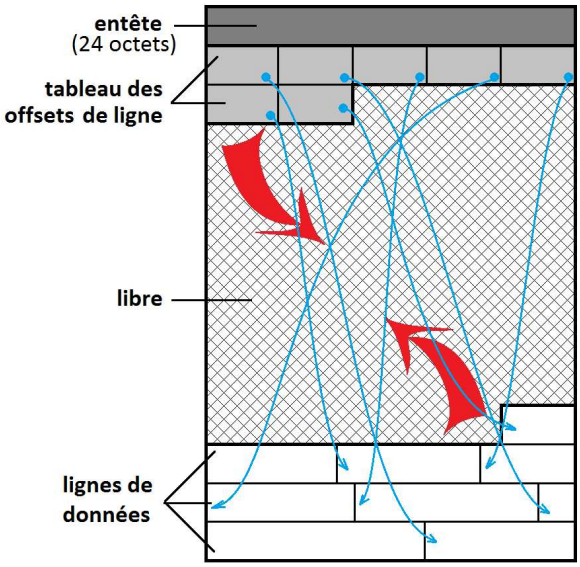


Figure 1. Structure des pages PostGreSQL

L'entête comporte les informations suivantes :

Attribut	Taille o	Description
pd_lsn	8	N° de Log Segment Number (LSN) du journal de transaction indiquant la dernière transaction relative aux données de la page
pd_tli	2	Complément d'information de la dernière marque transactionnelle relative à la page et précisant le temps
pd_flags	2	Bits d'état d'informations internes
pd_lower	2	Pointeur indiquant le début de l'espace libre
pd_upper	2	Pointeur indiquant la fin de l'espace libre
pd_special	2	Pointeur indiquant le début du pied de page de l'espace réservé aux données spécifique des index
pd_pagesize_version	2	Taille de la page et indicateur de version de PostGreSQL
pd_prune_xid	4	Identifiant de la plus vieille transaction (XMAX) ayant effectuée un delete non encore validé.

Tableau 5 - Structure des attributs d'entête de page

6 - Lignes

Rien de tel pour comprendre comment est structurée une ligne dans une page qu'un simple test. Créons une table et insérons une ligne (voir listing 10).

```
CREATE TABLE T_TEST_LIGNE
(TSL_DATA0      CHAR(8),
 TSL_DATA1      CHAR(8),
 TSL_DATA2      VARCHAR(8),
 TSL_DATA3      CHAR(8),
 TSL_DATA4      VARCHAR(8),
 TSL_DATA5      CHAR(8),
 TSL_DATA6      VARCHAR(8)) ;

INSERT INTO T_TEST_LIGNE
VALUES ('ZZZZZZZZ', NULL, NULL, 'AAAA', 'aaaa', 'BBBB', 'bbbb');
```

Listing 10 - Test d'insertion de données variable et fixe

Voyons maintenant le binaire de la page, en particulier la partie basse contenant les données de cette ligne (voir figure 2).

```
00001FC0   C9 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00   É . . . . . . . . . . . . . . . .
00001FD0   01 00 07 00 03 08 18 79 13 5A 5A 5A 5A 5A 5A 5A   . . . . . y . Z Z Z Z Z Z Z
00001FE0   5A 13 41 41 41 41 20 20 20 20 0B 61 61 61 61 13   Z . A A A A       . a a a a .
00001FF0   42 42 42 42 20 20 20 20 0B 62 62 62 62 00 00 00   B B B B       . b b b b . . .
```

Figure 2 - Binaire du slot de ligne inséré

On constate que les données sont strictement stockées dans le même ordre que les positions ordinales des colonnes lors de la création de la table. On constate aussi que les données CHAR(n) - dans notre exemple en majuscule - sont stockées sur N caractères avec une complétion à blanc, tandis que les données de taille variable (VARCHAR - dans notre exemple en minuscule) sont stockées sur la taille réelle des données stockées. Notons aussi qu'entre chaque donnée figure un octet de séparation (valeur hexadécimale 13). Par comparaison, SQL Server stocke toutes les données des colonnes de taille fixe en premier (sans séparateur) puis toutes les données de taille variable en second, avec deux octets d'entête de chaque valeur pour en spécifier la longueur stockée. Il en résulte des lectures plus rapide : on sait à l'avance à quel offset figure chaque colonne de taille fixe dans la ligne, et pour les colonnes de taille variable, il n'est pas nécessaire de lire tous les octets, mais de faire des sauts en tenant compte de la longueur stockée de chaque données avant celle concernée par la recherche.
Les entêtes de ligne comportent 32 octets suivis par une série d'octets constituant la matrice de nullabilité, si de telles colonnes figurent dans la table (à raison d'un bit par colonne NULL).
À l'exception de la matrice de nullabilité située en dernier, la constitution des entêtes des lignes est la suivante (voir tableau 6).

attribut	taille o	description
t_xmin	4	Identifiant de la transaction d'insertion
t_xmax	4	Identifiant de la transaction de suppression
t_cid	4	Identifiant de la commande d'insertion et de suppression (surcharge avec t_xvac)
t_xvac	4	Identifiant de la transaction pour l'opération VACUUM déplaçant une version de ligne
t_ctid	6	TID en cours pour cette version de ligne ou pour une version plus récente
t_infomask2	2	nombre d'attributs suivi de quelques bits d'état
t_infomask	2	différents bits d'options (flag bits)
t_hoff	1	décalage vers les données utilisateur

Tableau 6 - Structure des attributs de l'entête de ligne

Si l'on compare finement la structure des lignes de PostGreSQL par rapport à ses grands frères, on constate trois différences :

- les entêtes de pages sont plus fournis dans Oracle (100 octets) et SQL Server (96 octets) que dans PostGreSQL (24 + pied) ;
- dans SQL Server, les valeurs des colonnes de taille fixe sont stockées en premier dans la ligne et celles de taille variable en dernier dans la ligne, afin d'optimiser la lecture des données de taille fixes ;
- les entêtes de lignes sont plus petits dans Oracle (3 octets) et dans SQL Server (4 octets).

Notons que le LOBs (BLOB, CLOB ou NCLOB) dans PostGreSQL sont systématiquement compressés. Ceci est généralement une bonne chose, car de telles données peuvent être grandes et pleines de vide. Cependant cette technique est à double tranchant : pour des données vidéo, image ou son déjà compressés par des méthodes spécifiques la compression PostGreSQL n'y ajoute rien en termes de volumétrie, mais fait perdre du temps en process.
PostGreSQL ne propose pas non plus de technique comme le DataLink (norme SQL) pour stocker des fichiers électroniques dans le file system, mais sous la responsabilité exclusive du SGBDR, comme le fait par exemple MS SQL Server avec le FilStream et son extensions plus généralisées la « FileTable »
Enfin les fonctions de manipulation des LOBs sont spécifiques et ne répondent pas à de simples requêtes SQL basiques.

Une notion importante manque à postGreSQL, celle d'extension ou *extent* en anglais. Une extension est un bloc contigu de pages qui constitue l'unité minimale de lecture sur le disque. En effet, le déplacement d'une tête de lecture pour aller lire des données est relativement très lent comparée à la vitesse des opérations de lecture en RAM. De plus une tête de lecture de disque magnétique étant mécanique elle est sujette à l'inertie inhérente à tous les systèmes physiques lors des déplacements. D'où l'idée de profiter de cette inertie pour lire plus de page que nécessaire. Par exemple pour MS SQL Server, les extensions sont constituées de bloc de 8 pages de 8 kilo octets situées de manière contigües sur le disque, soit 64 Ko. Lorsque SQL Server veut lire des données situées sur le disque, il monte en mémoire l'extension, soit les 8 pages contigües, ce qui lui permet de faire des lectures anticipées, supposant que les pages autour de la page demandée ont de grandes chances d'être rapidement utilisées, ce qui est particulièrement vrai lorsque la table est organisée en CLUSTER sur un auto incrément ou une clef d'horodatation.

Enfin, du fait de l'absence de gestion des fichiers à bas niveau, il n'existe pas de routine propre à PostGreSQL pour vérifier et réparer une table corrompue physiquement (suite à une erreur d'écriture par exemple) ou logiquement (une données hors type, une contrainte non respectée...). Ce genre de commande existe bien sous Oracle comme sous MS SQL Server (DBCC CHECK...) ce qui permet de réparer à chaud les erreurs, mais ce genre de réparation n'est pas toujours garantie sans pertes de données... Il n'y a pas de miracle ! Mais il est possible d'utiliser des techniques avancées de restauration permettent de récupérer les pages corrompues et seulement celle-là à partie de sauvegardes complètes et application successives de journaux de transaction.

7 – Versions des lignes (MVCC)

MultiVersion Concurrency Control (littéralement contrôle de concurrence de multi-version) est un système de gestion des versions des lignes de tables de la base. Ce système, maintenant implémenté sous diverses formes dans les SGBDR, permet de prendre un cliché des données à manipuler au démarrage de la transaction (lectures dans le passée) et travailler sur une version des lignes au lieu de scruter des données susceptibles d'évoluer au cours du traitement et donc de conduire à des blocages. À l'origine, c'est Oracle qui le premier a implémenté ce système à partir de la version 3.
Notons d'ailleurs que PostGreSQL ne fournit aucune alternative à ce mécanisme alors que la norme SQL permet de descendre au niveau READ UNCOMMITTED afin d'éviter tout blocage, ce que supporte SQL Server par exemple.
Mais voyons voir comment cela est implémenté dans PostGreSQL. En fait, chaque version de ligne est conservée en page tant qu'il existe une transaction encore active sur la ligne visée. Lorsque la transaction est achevée, les lignes anciennes perdurent dans la page et seront « nettoyées » par le processus actif en tâche de fond AUTOVACUUM, si vous l'avez activé (ce qui est fortement conseillé).
Dans Oracle, les versions des lignes en cours de transaction sont stockées dans un journal particulier appelé UNDO. Dans MS SQL Server, c'est dans la base tempdb (celle gérant les objets temporaires) que sont stockées de manière transitoire les versions de ligne. Dans les deux cas, les pages des données des tables ne sont pas « enflées » par de multiples versions anciennes d'une même donnée...
Bref, la gestion du MVCC fait grossir artificiellement les pages de données, surtout si l'activité transactionnelle est forte, et plus une base est volumineuse, moins les performances sont bonnes, même si cette baisse de performance est marginale du fait de la présence des index. Enfin, l'absence d'une routine de nettoyage peut avoir des conséquences fâcheuses, même si les slots de lignes sont généralement récupérées pour de nouvelles entrées, et la récupération physique de la place dans les

pages nécessite d'autres opérations, comme la reconstruction des tables ou des index, via une commande VACUUM FULL ou CLUSTER.

8 - Les journaux de transaction

Comme toute base de données transactionnelle, PostGreSQL utilise un journal de transaction pour assurer l'ACIDité (Atomicité, Cohérence, Isolation Durabilité) des transactions. Le journal est stocké dans le répertoire PG_XLOG dans l'arborescence d'installation (par exemple C:\Program Files\PostgreSQL\9.1\data). Le journal de transaction est en fait constitué de plusieurs fichiers si nécessaire, d'une taille de 16 Mo, qui sont chaînés les uns aux autres. Le premier fichier a pour nom 000000010000000000000000 dont les huit premiers caractères correspondent au TimeLineID (un marqueur temporel) et les suivants à un numéro de séquence. La création des nouveaux fichiers se fera au fur et à mesure lorsque le fichier précédent sera plein.

Parce que les journaux de transaction sont souvent un point de contention dans les SGBDR, il est important de placer les journaux de PostGreSQL sur un disque physiquement indépendant de celui des données ou du système et si possible sur un agrégat RAID permet de ventiler les IO sur différents axes, surtout pour l'écriture. On préférera donc du RAID 0, mais par sécurité on utilisera du RAID 0+1 ou du RAID 10 avec 4, 6 ou 8 disques. On évitera le RAID 5 particulièrement lent en écriture. Il est même préférable lorsque c'est possible et dans un budget donné, de diminuer le volume de chaque disque en augmentant leur nombre. À noter, l'emploi des SSD (Solid State Device) n'est pas encore recommandé pour les SGBDR, tout à la foi pour des raisons techniques (l'écriture y est moins rapide qu'un bon RAID 10), comme pour des raisons économiques (pour un même budget, un bon SAN dédié sera plus rapide et plus pérenne).

Pour placer les journaux, il est possible d'activer le paramètre -X dans initdb lors de l'installation d'une instance PostGreSQL, avec le nouveau chemin en paramètre.
Pour une instance déjà créée, il faut utiliser le lot de commande en ligne suivant (Linux) :

```
pg_ctl stop
mv $PGDATA/pg_xlog <nouveau chemin>
ln -s <nouveau chemin>/pg_xlog
pg_ctl start
```

Qui arrête l'instance, déplace les fichiers (commande linux mv), redéfini le lien symbolique (commande linux ln -s) et redémarre l'instance.

Dans une instance PostGreSQL il n'existe qu'un seul journal de transaction pour toutes les bases de données.

9 – Les tables systèmes

En matière de stockage PostGreSQL est loin d'égaler ce que sont capables de faire Oracle ou MS SQL Server. Même si les choses progressent petit à petit. Or, les impétrants de PostGreSQL revendiquent une certaine paternité à Oracle. On le ressent assez bien d'ailleurs dans la forte complexité des tables et vues systèmes :

- les noms de colonnes changent au gré des tables, ce qui est peu pratique pour effectuer des jointures (exemple *reltablespace* du côté *pg_class* et *oid* du côté *pg_tablespace*)
- certaines informations varient en fonction d'autres paramètres pas toujours situés dans la table scrutée ou la fonction utilisée (exemple la fonction *pg_relation_filepath*)

- certaines colonnes, notamment les fameux *oid*, ne sont pas visibles dans la liste publiée des colonnes de la table, mais dans quelles tables se cachent-ils ? Mystère... Même la documentation officielle en ligne ne le dit pas (exemple il y a bien une colonne cachée *oid* dans *pg_class*, mais pas dans *pg_aggregate*)
- certains acronymes sont donnés à deux concepts différents. Exemple dans la documentation officielle de PostgreSQL on parle de TID pour TimeLineID pour la fonction *pg_resetxlog*, tandis que dans la rubrique « Acronymes » on décrit le TID, comme étant le « Tuple IDentifier »...
- certaines informations ne sont pas disponibles par requête. Par exemple l'obtention du numéro de version du catalogue système n'est disponible que par une commande en ligne

Tout ceci n'est pas très pratique et rend peu intuitif la mise au point des requêtes d'interrogation de métadonnées, nécessaires à l'exploitation de PostGreSQL. Est-ce une des raisons qui le rend plus couteux en administration que MS SQL Server ?

10 - Comparaison du stockage Oracle et SQL Server / PostGreSQL

Nous voilà finalement au pied du mur : comparons ce que fait PostGreSQL par rapport aux poids lourds que sont Oracle ou MS SQL Server. N'oublions pas que PostGreSQL se pose souvent en challenger « gratuit » d'oracle dans beaucoup de publications…

Un partitionnement inepte !

J'ai eu l'occasion de faire un article comparant le système de partitionnement proposé par PostGreSQL à celui de MS SQL Server… Le moins que l'on puisse dire c'est que le partitionnement avec PostGreSQL c'est juste un concept ! En pratique il est impossible de le mettre en œuvre. Rien donc de comparable à ce que font Oracle ou SQL Server, et mieux vaut l'ignorer. Les auteurs de PostGreSQL en sont bien conscients car le projet d'un vrai système de partitionnement est à nouveau dans les tuyaux…
À lire donc : « Partitionner une table. Comparaison PostGreSQL / MS SQL Server »
http://blog.developpez.com/sqlpro/p10378/langage-sql-norme/partitionner_une_table_comparaison_postg

MVCC occupe de la place inutile… Il faut nettoyer et cela coûte !

Le système MVCC occupe de la place pour stocker les différentes versions des lignes, mêmes lorsqu'elles ne sont plus inutilisées. Il en résulte des tables et index fortement fragmentés qu'il convient de reconstruire. Il faut donc prévoir de vider régulièrement les espaces morts, ce que le processus AUTOVACUUM permet de faire en tâche de fond. Problème : cette commande pompe des ressources en permanence et en cas de charge importante peut bloquer certaines requêtes en production ! On peut aussi reconstruire les tables par la commande CLUSTER ou encore en utilisant un VACUUM FULL. Problème : ces deux méthodes nécessite de verrouiller exclusivement les tables qui s'en trouvent donc inaccessible !
En comparaison Oracle utilise un espace à part pour stocker ses versions de ligne (les Undo Segments), tout comme SQL Server (la base tempdb). Le résultat est que les bases PostGreSQL sont naturellement plus grosses et les données plus fragmentées.

La défragmentation en question : où comment bloquer les données…

Oracle, comme SQL Server ont besoin de défragmenter les index. Tout comme PostGreSQL. La différence est que du fait du versionnement des lignes par le MVCC les données des tables et index PG sont beaucoup plus fragmentées que ne le sont celle d'Oracle ou SQL Server. En effet, dans ces deux derniers SGBDR les versions de lignes sont stockées à part et auto-nettoyées par des processus indépendant des lectures et écritures des données de production.
Or PostGreSQL ne propose aucune méthode de défragmentation des index ou des tables « online ».
En effet, les commandes CLUSTER ou REINDEX de PG, posent un verrou exclusif sur les objets à, défragmenter, ce qui n'est pas le cas des méthodes de reconstruction d'index ou de table que proposent Oracle ou SQL Server avec l'option « online » qui laisse possible les accès à l'objet tant en lecture qu'à l'écriture.

Des opérations de croissance pénalisantes…

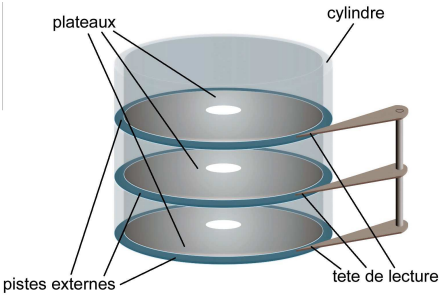
La gestion des fichiers implique des opérations d'extension régulière et PostGreSQL ne permet pas d'allouer une taille spécifique aux fichiers que ce soit pour une table, un index, comme pour les journaux de transaction. Cela permettrait d'éviter les opérations de croissance et la fragmentation physique des fichiers. Les opérations de croissance sont toujours très coûteuses et ont plus de chances de survenir au moment où l'activité transactionnelle est forte, c'est à dire au moment où il faudrait minimiser la durée de verrouillage. Par comparaison, Oracle comme SQL Server permettent de pré-tailler les espaces de stockage, comme de dimensionner les journaux de transactions ce qui ôte toute problématique liée à la fragmentation physique. Pour les fichiers des tables et des index de PostGreSQL, la fragmentation physique peut être combattue avec un outil de défragmentation externe. Mais cela suppose d'arrêter totalement le service des données !

Un placement des fichiers indéterminé…

PostGreSQL ne permet pas de faire des opérations d'allocation de fichier à bas niveau, contrairement à Oracle ou SQL Server. Les fichiers sont donc créés ou le système trouve un peu de place et se trouvent donc fortement morcelés lors des opérations de croissance (voir ci avant).
En comparaison, Oracle comme SQL Server permettent de pré réserver les espaces de stockage et dans ce cas, les routines internes du SGBDR permettent de choisir le meilleur emplacement sur le disque afin de minimiser le trajet de la tête de lecture (cylindres externes des plateaux du disque).
Ceci permet d'obtenir des temps d'accès en lecture et écriture physique bien supérieurs à ce que l'on obtient d'un placement aléatoire des fichiers. Dans ce cas, la vitesse de rotation des disques devient le critère le plus important car le déplacement linéaire des têtes de lecture est minimisé.

Placement des fichiers de bases de données sur un disque

Grace à la gestion fine des espaces de stockage, les bases de données comme Oracle ou MS SQL Server savent placer les fichiers de données aux meilleurs emplacements des disques physiques. Ce meilleur emplacement est constitué par les cylindres externes de l'ensemble des plateaux du disque, comme l'indique la figure ci-dessous :



De ce fait, la tête de lecture bouge assez peu et le temps d'accès est plus lié à la vitesse de rotation du disque. Le fichier étant ventilé au même endroit des différents plateaux, il est possible de simuler un parallélisme d'accès à tous les plateaux. Enfin ce placement, associé à une cartographie intime des pages sur le disque, permet d'effectuer des écritures regroupées par contiguïté physique.
Il en ressort une sensible diminution des temps de réponse principalement à l'écriture, à condition cependant d'éviter des agrégats de disque en RAID 5 ou 6 et des LUN taillées à cheval sur les disques physique.
Image : © Claude Leroy D.R.

Des écritures et des lectures aléatoires…

L'absence de gestion des espaces de stockage en grands fichiers comme le fait Oracle ou MS SQL Server et notamment l'absence de la notion d'extension combinée à l'absence de routines de bas niveau pour la gestion des lectures et écritures physiques, empêche certaines optimisations comme la lecture anticipée (une lecture physique profite de la latence des têtes de lectures du disque pour remonter plusieurs pages en cache d'un même bloc ou extension) ou plus encore l'algorithme d'écriture optimale des pages (détermination du meilleur parcours de la tête de lecture lors des écritures des pages sales par regroupement).

Bloqué par l'absence de gestion des espaces de stockage…

Comment faire si vous avez placé une table sur un disque trop petit et que votre base a grossit considérablement ? Le seul moyen avec PostGreSQL sera d'arrêter le service des données, créer un tablespace sur un autre disque, déplacer les fichiers de la table (mais lesquels ?) sur cet espace de stockage et redéfinir dans les tables système le lien entre cette table et le tablespace… Rien d'intuitif et toute la base est bloquée !
En comparaison Oracle comme SQL Server permettent de migrer à chaud des tables comme des index du fait de la gestion des fichiers et des espaces de stockage directement par le SGBDR, y compris lorsque la table est partitionnée (création, fusion et échange de partitions à chaud).

Pas de données « read only »

Une des autres conséquences de l'absence de gestion des espaces de stockage est qu'il n'est pas possible de placer certaines tables dans un état « read only ». Par exemple pour des données sans évolution (calendrier, données d'historiques, bilans clos… par exemple) ou à évolution très lente (référentiels de données…), il est possible de créer des espaces de stockage particulier et de les placer en lecture seule après les avoir alimentés. Ceci induit deux avantages subséquents : ne plus poser aucun verrou sur les données lues et permettre une indexation notablement accrue n'ayant plus aucune incidence sur la production.

Trop de fichiers pénalisent le système…

Nous savons tous que la manipulation de trop nombreux fichiers à un seul et même endroit du file system pénalise ce dernier. Or la multiplication des fichiers liées à une même table (les données d'un cotés et autant de fichier supplémentaires que d'index de l'autre), multiplié par le nombre de tables, arrive parfois à écroulés les performances de l'OS.
Cet inconvénient est inconnu des SGBDR que sont Oracle ou SQL Server du fait que les fichiers des espaces de stockage concentre les données de toutes les tables ou de tous les index, ou comme on le souhaite en créant autant de fichier et d'espace de stockage que l'o veut.

Récupérer des tables corrompues ?

Du fait que PostGreSQL confie la gestion des fichiers et du stockage des données des tables à l'OS, il n'existe presque aucun moyen de récupérer ou réparer des tables endommagés (pour des index c'est moins grave, il suffit de les reconstruire). Seule méthode proposée : la restauration… C'est-à-dire une probable perte irréversible de données. Notez qu'il n'existe pas non plus d'outil pour vérifier la consistance du stockage des données de PG...

Oracle comme SQL Server proposent des méthodes de réparation, comme de vérification (physique et logique) des données qui permettent de trouver rapidement les problèmes et les corriger, même à chaud. Par exemple le DBCC CHECKTABLE de SQL Server permet de réparer les lignes des tables en récupérant les informations des index à condition que ces informations figurent dans un moins un index. De même, la réparation de pages endommagées est possible en retrouvant les pages non corrompues dans une sauvegarde et en appliquant tous les journaux de transactions intermédiaires.
D'où l'importance de vérifier régulièrement la consistance physique (checksum du contenu des pages) comme logique (typage des données, contraintes appliquées) des données figurant dans les pages… Tâche apparemment impossible à faire sous PG !

Une gestion des LOBs minimaliste…

L'idée de compresser les données des LOBs est bonne à condition que l'on y stocke que des données littérales. Elle devient mauvaise pour des fichiers contenant des images, des sons ou de la vidéo. Il faut donc s'en méfier. Or pour des données provenant de fichiers électronique PostGreSQL ne propose rien et certains vont même jusqu'à conseiller de les stocker à part dans le file system à la manière de ce que l'on faisait dans dbase ou Access ! Un peu léger lorsque l'on a besoin de synchroniser données relationnelles et fichiers…
En comparaison SQL Server permet de stocker les fichiers électroniques par le biais des FileStream et FileTable (une implémentation proche du DATALINK de la norme SQL), c'est-à-dire en les laissant à titre de fichiers, mais sous la responsabilité exclusive du SDGBR, transactionnement compris et en assurant la consistance de la sauvegarde à chaud.
D'ailleurs, à la page consacré au type DATALINK de la documentation de PostGreSQL il est même écrit que le concept de DATALINK « *is probably even impossible to do properly on plain unix* »… Bizarre d'écrire cela alors qu'IBM DB2 le propose, justement sur des machines UNIX !

Le journal de transaction … toujours un point de contention !

Comme dans tous les SGBDR, les journaux de transactions sont un point de contention. Or PostGreSQL n'utilise qu'un seul journal pour toutes ses bases (morcelé en plusieurs fichiers), contrairement à Oracle ou MS SQL Server qui activent un journal par base avec, en particulier pour SQL Server, un journal de transaction supplémentaire pour les objets temporaires parce qu'ils sont produits dans une base à part (tempdb).
Or le journal de transaction constitue toujours le point le plus critique de tous SGBDR. Sa non multiplicité à travers différentes bases peut avoir des effets pervers… pas exemple rendre lente une base peu sollicité par le fait qu'une autre base fait un import massif de données !
En sus la corruption du journal entrainera la corruption de toutes les bases… pas de quoi être rassuré !

Une sécurité passoire !

Pour terminer, la sécurité des fichiers est faible en comparaison de ce que font Oracle ou MS SQL Server. Nous avons vu qu'il était possible de modifier ou supprimer un fichier d'une base de données sans que PostGreSQL ne s'en aperçoive de prime abord. Ceci est impossible avec SQL Server sous Windows qui maintiennent des verrous d'accès exclusif sur l'ensemble des fichiers des bases, (sauf le cas où le DBA mettrait sciemment une base hors ligne).
De plus PostGreSQL n'offre aucun moyen de crypter le stockage des données des bases contrairement à Oracle ou SQL Server ou les fichiers (pas les données en mémoire) peuvent être cryptés de manière « transparente » (TDE ou Transparent Data Encryption) c'est-à-dire à la volée lors

des écritures et lectures physiques, et ce y compris pour le journal de transaction et sans qu'il en résulte un ralentissement considérables des opérations de la base. Une telle sécurité garantit le vol des fichiers, comme celui des sauvegardes qui se trouvent naturellement cryptées du fait du cryptage des fichiers de la base.
C'est aujourd'hui un point de passage presque obligé pour ceux qui veulent travailler dans le domaine de la santé (hôpitaux, mutuelles…) ou celui de la finance.

Faut-il miser sur PostGreSQL ?

Le professionnel que je suis déconseille l'utilisation de postGreSQL sur des bases fortement transactionnelles (site web marchand par exemple), comme sur des domaines sensibles sur le plan de la sécurité ou de la confidentialité, telles que la santé ou les finances.

En revanche PostGreSQL reste un bon choix pour des bases de données relationnelles de petite à moyenne taille (quelques dizaines à quelques centaines de Go) lorsque l'on en connait les limites et que l'on met en œuvre les moyens de se prémunir des quelques inconvénients du système.

RÉFÉRENCES

WEB

THE DESIGN OF THE POSTGRES STORAGE SYSTEM (Michael Stonebraker)
http://db.cs.berkeley.edu/papers/ERL-M87-06.pdf

PostgreSQL Row Storage Fundamentals (Jeremiah Peschka) :
http://facility9.com/2011/03/postgresql-row-storage-fundamentals/

Mvcc Unmasked (Bruce Momjian)
http://momjian.us/main/writings/pgsql/mvcc.pdf

PostgreSQL et ses journaux de transactions (Guillaume Lelarge)
http://www.dalibo.org/glmf108_postgresql_et_ses_journaux_de_transactions
http://www.unixgarden.com/index.php/gnu-linux-magazine/postgresql-et-ses-journaux-de-transactions

Livres

The PostGreSQL 9.0 Reference manual - Volume 3: Server Administration Guide
Network Theory LTD - 2010

PostGreSQL (Korry Douglas, Susan Douglas)
Developer's Library / Sams Publishing - 2006

PostGreSQL 9 Administration Cookbook (Simon Riggs, Hannu Krosing)
PACKT Publishing - 2010

Base de données PostGreSQL - Gestion des performances (Gregory Smith)
Pearson Education - 2011

Les techniques utilisées par la concurrence :

Comment fonctionne le versionnement des lignes d'Oracle et de SQL Server (par Ivan Lebedev) MS SQL Server vs Oracle: Row Versioning
http://addref.blogspot.fr/2011/09/ms-sql-server-vs-oracle-row-versioning.html

FileStream et FileTable dans SQL Server (par Arshad Ali) FileStream and FileTable in SQL Server 2012 :
http://www.databasejournal.com/features/mssql/filestream-and-filetable-in-sql-server-2012.html

Le cryptage transparent des données (Wikipedia) Transparent Data Encryption :
http://en.wikipedia.org/wiki/Transparent_Data_Encryption