Workshop 9.6

Nouveautés de PostgreSQL 9.6

Dalibo & Contributors

http://dalibo.github.io/workshops

Nouveautés de PostgreSQL 9.6

Workshop 9.6

TITRE : Nouveautés de PostgreSQL 9.6
SOUS-TITRE : Workshop 9.6

Table des Matières

NOUVEAUTÉS DE POSTGRESQL 9.6

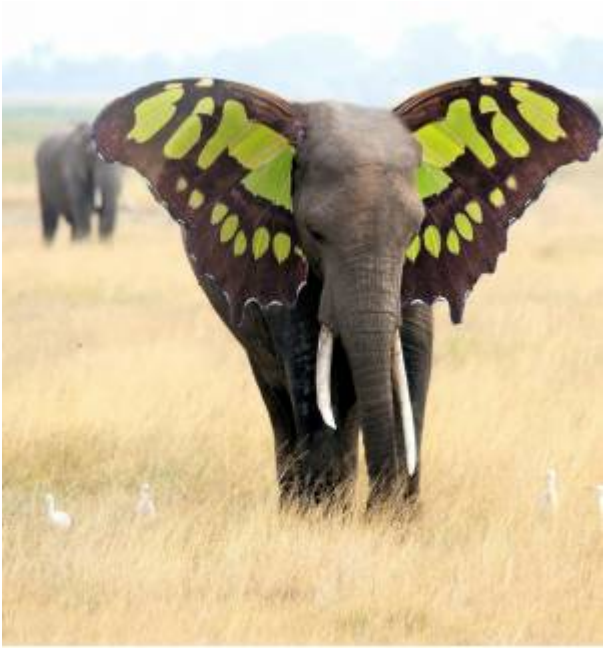

Figure 1: PostgreSQL

Photographie obtenue sur https://pixabay.com/en/elephant-ears-butterfly-1057465/

CC0 Public Domain, Free for commercial use, No attribution required

---

INTRODUCTION

- Développement commencé en juin 2015
- Version Beta1 sortie le 12 mai 2016
- Sortie le 29 septembre 2016

Comme toutes les versions majeures de PostgreSQL, le développement se fait en un peu plus d'un an et est suivi de plusieurs mois de tests approfondis. Le développement de la version 9.6 a donc commencé en juin 2015. Cette version est toujours en développement.

Aucune version alpha n'a été proposée, les développeurs ont choisi de travailler sur des versions bêta. La version finale est sortie le 29 septembre 2016.

Au niveau de l'organisation, une Release Team a été montée depuis mai 2016 pour suivre la bonne gestion des open items (problèmes à résoudre avant la sortie de la version finale) et pour s'assurer de la sortie en temps et en heure de la version finale qui devrait être disponible fin septembre.

Afin de pouvoir facilement identifier les nouvelles fonctionnalités que proposent cette mouture par rapport à une version précédente, vous pouvez vous rendre sur le site http://www.postgresql.org/about/featurematrix/ : ceci vous permettra d'avoir une vue cumulative de ces fonctionnalités.

AU MENU

- Performances
  - Administration
  - SQL
  - Monitoring
  - Régressions / changements
  - Et une petite ouverture vers le futur

PERFORMANCES

- Parallélisme :
  - parcours séquentiel
  - jointure
  - agrégation
- Index bloom
- Améliorations globales
  - checkpoint

La grosse amélioration au niveau des performances est l'ajout du parallélisme. Cependant, les développeurs de PostgreSQL n'en sont pas restés et proposent des évolutions qui auront des conséquences très positives pour les performances.

PARALLÉLISME

- Avant la 9.6
  - une requête = un processus
  - pas de multithreading
- À partir de la 9.6
  - toujours pas de multithreading
  - toujours un processus pour une session
  - mais il peut être aidé par d'autres processus appelés **workers**

PostgreSQL fonctionne en mode multi-processus avec mémoire partagée. Il a été décidé que ce dernier ne fonctionnerait pas sur le principe des threads (en raison de la complexité que cela pouvait engendrer au développement, voir Why don't you use threads?[1]).

Dans le cas classique donc, une session qui exécute une requête va utiliser un cœur de processeur pour effectuer son calcul.

Il n'est plus rare aujourd'hui de voir des processeurs avec plus de dix cœurs et des serveurs avec plus de deux emplacements pour processeurs. Ainsi les serveurs disposent de plusieurs dizaines de cœurs disponibles. Dans le cas d' une utilisation dite "transactionnelle" (ou OLTP) de l'instance PostgreSQL, avec beaucoup de sessions exécutant de nombreuses petites requêtes, ces derniers sont utilisés à bon escient.

Mais dans le cas d'une utilisation dite "décisionnelle" (ou OLAP), avec peu de sessions mais exécutant des requêtes complexes, on peut se retrouver avec une partie de la puissance de calcul de la machine qui n'est pas utilisée au vu de cette limitation.

Une manière d'optimiser l'utilisation des ressources, et éventuellement d'améliorer les performances, serait de pouvoir utiliser deux processus ou plus pour partager la charge de calcul engendrée par cette opération.

Longtemps attendue, cette fonctionnalité, le parallélisme, fait partie de la version 9.6.

---

[1] https://wiki.postgresql.org/wiki/Developer_FAQ#Why_don.27t_you_use_threads.2C_raw_devices.2C_async-I.2FO.2C_.3Cinsert_your_favorite_wizz-bang_feature_here.3E.3F

PARALLÉLISME : OPÉRATIONS

- Parcours séquentiel
- Jointure
- Agrégation
- Uniquement les requêtes en lecture !

La version 9.6 intègre la parallélisation pour différentes opérations : les parcours séquentiels, les jointures et les agrégats. Mais attention, cela ne concerne que les requêtes en lecture : pas les `INSERT`/`UPDATE`/`DELETE`, pas les CTE en écriture, pas les opérations de maintenance (`CREATE INDEX`, `VACUUM`, `ANALYZE`).

Les prochains chapitres vont détailler les différentes opérations actuellement parallélisables.

PARALLÉLISME : PARAMÈTRES 1/2

- `max_worker_processes` : Nombre max de background worker
- `max_parallel_workers_per_gather` : Nombre max de worker par gather
- `min_parallel_relation_size` : Taille min de la relation pour déclencher le parallélisme
  - Possibilité de forcer le nombre de worker possible pour une table :
  - `SET (parallel_workers = x)`

Dans un premier temps, il faut définir les paramètres suivants :

- `max_worker_processes` : nombre maximum de processus supplémentaires pouvant être lancés par l'instance. Par défaut à 8. Ce paramètre est global à tous les background worker. Par exemple, si vous avez deux background worker déjà en place (extension ...), seulement 6 workers pourront être lancés pour la parallélisation.
- `max_parallel_workers_per_gather` : nombre maximum de workers qu'un processus pourra utiliser. Par défaut à 2 pendant la beta, à 0 depuis la RC1. **Autrement dit, le parallélisme est désactivé par défaut!**. C'est une limite par **gather**, si une requête utilise plusieurs gathers elle pourra dépasser ce seuil.
- `min_parallel_relation_size` : taille minimale de la relation pour déclencher le parallélisme. Par défaut à 8 Mo.
- Il est aussi possible de forcer le nombre de workers possible pour une table avec l'instruction `SET (parallel_workers = x)` à la création de la table `CREATE TABLE`

`xxx SET (parallel_workers = x)` ou en modifiant les propriétés d'une table déjà présente : `ALTER TABLE xxx SET (parallel_workers = x)`

La modification de ces paramètres nécessite un rechargement de la configuration sauf pour `max_worker_processes` qui nécessite un redémarrage.

PARALLÉLISME : PARAMÈTRES 2/2

- `parallel_setup_cost` : Coût de la mise en place du parallélisme
- `parallel_tuple_cost` : Coût pour passer une ligne d'un worker au gather.

La mise en place du parallélisme représente un coût : Il faut mettre en place une mémoire partagée, lancer des processus... Ce coût est pris en compte par le planificateur à l'aide du paramètre `parallel_setup_cost`. Par ailleurs, le transfert d' enregistrement entre un worker et un autre processus a également un coût représenté par le paramètre `parallel_tuple_cost`.

Ainsi une lecture complète d'une grosse table peut être moins couteuse sans parallélisation du fait que le nombre de lignes retournées par les workers est très important. En revanche, en filtrant les résultats, le nombre de lignes retournées peut être moins important et le planificateur peut être amené à choisir un plan comprenant la parallélisation.

PARCOURS SÉQUENTIEL PARALLÈLE : EXPLICATIONS 1

- Traitement des blocs par processus
- Compteur de blocs
  - Parallel Heap Scan Current Block
- Fonctionne en mode partiel
  - impossible à utiliser avec des données à trier

Le processus de la session connectée va devoir diviser l'activité qu'il doit traiter par le biais du planificateur. Il réalise cette opération en créant des processus supplémentaires, appelés workers, qui pourront chacun utiliser un cœur.

En mémoire partagée est créé un compteur : PHSCB pour parallel heap scan current block, commun au processus initial ainsi qu'aux workers supplémentaires créés par ce dernier. Ce compteur contient le numéro du bloc de données à lire. Chaque worker va aller poser un verrou (Spin Lock) sur ce compteur pour le lire et l' incrémenter, puis relâche le verrou

et va travailler sur les données qu'il s' est affecté par ce biais. L'autre worker procède de la même manière et ainsi de suite.

Le processus initial peut lui aussi travailler sur les données s'il n'est pas constamment occupé à traiter les données renvoyées par les workers. C'est de cette manière que les différents processus se partagent la tâche.

Les fonctions supportées par le parallélisme doivent donc pouvoir fonctionner en mode partiel, c'est pourquoi le parallélisme n'est pour l'instant supporté qu'en lecture, et ne fonctionne pas pour des requêtes dont les données ont besoin d'être triées.

PARCOURS SÉQUENTIEL PARALLÈLE : EXEMPLE 1

- Exemple d'une table `test_parallel_little`
  - une seule colonne
  - non indexée
  - contenant 500 000 lignes

On insère 500 000 lignes dans une table de test, ce qui représente environ 17 Mo.

EXEMPLE 1 : JEU DE TESTS

```
CREATE TABLE test_parallel_little (id int);

INSERT INTO test_parallel_little
  SELECT generate_series(0,500000);

SELECT n.nspname as "Schema",
       c.relname as "Name",
       pg_size_pretty(pg_table_size(c.oid)) as "Size"
FROM pg_class c
LEFT JOIN pg_namespace n ON n.oid=c.relnamespace
WHERE c.relname = 'test_parallel_little';


 Schema |     Name      |  Size
--------+---------------+--------
 public | test_parallel | 17 MB
```

Il n'existe pas d'index sur cette colonne et la taille est supérieure au seuil de 8Mo défini au niveau du paramètre `min_parallel_relation_size`. En faisant une extraction sur cette table, on se trouve dans les conditions nécessaires pour le déclenchement d'un parcours séquentiel parallélisé.

EXEMPLE 1 : PLAN

```
EXPLAIN (ANALYZE, VERBOSE, BUFFERS)
SELECT * FROM test_parallel_little WHERE id = 12;

 Gather  (cost=1000.00..6889.58 rows=1 width=4)
         (actual time=30.300..30.385 rows=1 loops=1)
   Output: id
   Workers Planned: 1
   Workers Launched: 1
   Buffers: shared hit=2264
   ->  Parallel Seq Scan on public.test_parallel_little
       (cost=0.00..5889.48 rows=1 width=4)
       (actual time=18.568..27.957 rows=0 loops=2)
         Output: id
         Filter: (test_parallel_little.id = 124862)
         Rows Removed by Filter: 250000
         Buffers: shared hit=2213
         Worker 0: actual time=7.054..25.832 rows=1 loops=1
           Buffers: shared hit=978
 Planning time: 0.141 ms
 Execution time: 31.862 ms
(14 lignes)
```

En regardant le plan, on constate que l'on a utilisé le parallélisme mais avec un worker seulement, la charge a été divisée entre le Gather (processus principal) et le worker 0.

PARCOURS SÉQUENTIEL PARALLÈLE : EXPLICATIONS 2

- Taille de la table = T
- `min_parallel_relation_size` = S
- Si T < S, pas de worker
- Si T > S, un worker
- Si T > Sx3, un worker supplémentaire (2)
- Si T > Sx3x3, deux workers supplémentaires (3)
- Si T > Sx3^3, trois workers supplémentaires (4)

Par exemple, dans notre cas :

- à partir de 24 Mo on passe à 2 workers
- à partir de 72 Mo on passe à 3 workers
- à partir de 216 Mo on passe à 4 workers
- à partir de 648 Mo on passe à 5 workers

et ainsi de suite dans la limite du nombre de workers qu'il est possible de créer.

PARCOURS SÉQUENTIEL PARALLÈLE : EXEMPLE 2

- Exemple d'une table test_parallel
  - une seule colonne
  - non indéxée
  - contenant 5 000 000 lignes

Testons avec une table plus volumineuse.

EXEMPLE 2 : JEU DE TESTS

```
CREATE TABLE test_parallel_little (id int);

INSERT INTO test_parallel_little
  SELECT generate_series(0,5000000);

SELECT n.nspname as "Schema",
       c.relname as "Name",
       pg_size_pretty(pg_table_size(c.oid)) as "Size"
FROM pg_class c
```

```
LEFT JOIN pg_namespace n ON n.oid=c.relnamespace
WHERE c.relname = 'test_parallel';

 Schema |     Name      |  Size
--------+---------------+--------
 public | test_parallel | 173 MB
```

On insère 5 millions de lignes dans une table de test, ce qui représente environ 173 Mo.

EXEMPLE 2 : PLAN

```
EXPLAIN (ANALYZE, VERBOSE, BUFFERS)
SELECT * FROM test_parallel WHERE id=12;
 Gather  [...]
   Output: id
   Workers Planned: 3
   Workers Launched: 3
   Buffers: shared hit=5465 read=16812
   ->  Parallel Seq Scan on public.test_parallel  [...]
         Output: id
         Filter: (test_parallel.id = 124862)
         Rows Removed by Filter: 1250000
         Buffers: shared hit=5312 read=16812
         Worker 0: actual time=3.766..139.729 rows=1...
           Buffers: shared hit=1748 read=5466
         Worker 1: actual time=138.532..138.532 rows=0...
           Buffers: shared hit=812 read=2669
         Worker 2: actual time=138.508..138.508 rows=0...
           Buffers: shared hit=816 read=2683
 Planning time: 0.120 ms
 Execution time: 151.700 ms
(18 lignes)
```

Détaillons un peu ce plan d'exécution :

```
Workers Planned: 3
Workers Launched: 3
```

Cette partie signifie que le planificateur a choisi que trois workers devraient être utilisés pour traiter cette opération (Workers Planned), et qu'en effet, trois workers ont été utilisés pour la réaliser (Workers Launched).

Si par exemple, la limite des workers définie dans `max_worker_processes` est atteinte, on pourra se retrouver avec l'information suivante :

```
   Workers Planned: 3
   Workers Launched: 1
```

Concernant la répartition de la charge, dans le cas du parcours séquentiel parallélisé, c'est assez simple, chacun des trois workers supplémentaires prend à sa charge une partie des données à lire.

```
        Worker 0: actual time=3.766..139.729 rows=1 loops=1
          Buffers: shared hit=1748 read=5466
        Worker 1: actual time=138.532..138.532 rows=0 loops=1
          Buffers: shared hit=812 read=2669
        Worker 2: actual time=138.508..138.508 rows=0 loops=1
          Buffers: shared hit=816 read=2683
```

Dans notre exemple, le premier processus à lu 7214 blocs (1748+5466), le second en a lu 3481 (812+2669) et le dernier 3499 (816+2683).

C'est dans la partie des données traitées par le premier worker que la ligne concernée a été trouvée.

Le processus principal lit lui aussi des données :

```
        Buffers: shared hit=5312 read=16812
```

Et la partie supérieure (Gather) donne le détail de l'assemblage des informations fournies par les trois workers :

```
 Gather  (cost=1000.00..43285.39 rows=1 width=4) (actual time=150.116..150.190
  rows=1 loops=1)
   Output: id
   Workers Planned: 3
   Workers Launched: 3
   Buffers: shared hit=5465 read=16812
```

Si on regarde du côté des processus postgres :

```
UID        PID  CMD
postgres   341  /usr/pgsql-9.6/bin/postmaster -D /var/lib/pgsql/9.6/data
postgres   343   \_ postgres: logger process
postgres   345   \_ postgres: checkpointer process
postgres   346   \_ postgres: writer process
postgres   347   \_ postgres: wal writer process
postgres   348   \_ postgres: autovacuum launcher process
postgres   349   \_ postgres: archiver process
postgres   350   \_ postgres: stats collector process
```

```
postgres   455   \_ postgres: postgres postgres [local] SELECT
postgres   550   \_ postgres: bgworker: parallel worker for PID 455
postgres   551   \_ postgres: bgworker: parallel worker for PID 455
```

On voit bien que trois processus `worker` ont été créés et sont relatifs à la session de PID 455, qui est la session dans laquelle on a lancé notre EXPLAIN. On voit aussi que ce ne sont pas des fils directs du processus en communication avec le client, mais des fils du processus `postmaster`.

PARALLÉLISME SUR UNE JOINTURE

```
EXPLAIN ANALYZE SELECT * FROM big b1 JOIN big b2 USING (id) WHERE b1.id < 400000;
 Gather  [...]
   Workers Planned: 2
   Workers Launched: 2
   ->  Hash Join  [...]
         Hash Cond: (b1.id = b2.id)
         Worker 0: [...]
         Worker 1: [...]
         ->  Parallel Seq Scan on b1  [...]
               Filter: (b1.id < 400000)
               Rows Removed by Filter: 16903334
               Worker 0: [...]
               Worker 1: [...]
         ->  Hash  [...]
               Buckets: 4194304  Batches: 1  Memory Usage: 141753kB
               Worker 0: [...]
               Worker 1: [...]
               ->  Seq Scan on public.t3  [...]
                     Worker 0: [...]
                     Worker 1: [...]
```

Dans le cas d'une jointure classique, on a un premier parcours d'une table, puis le calcul d'une table de hachage, puis le parcours de la seconde table tenant compte du calcul précédent.

Dans le cas d'une jointure parallélisée, les parcours séquentiels sont parallélisés, comme dans le cas d'un parcours classique et c'est le nœud Gather qui s'occupe de réaliser le calcul du hachage (première étape) puis la jointure en tant que telle (deuxième étape).

Il est aussi possible de constater une vraie parallélisation de la réalisation de la jointure, où les opérations de création de la table de hachage et le premier parcours de la table sont faits en parallèle. Néanmoins, chaque worker créant sa table de hachage, on en crée

donc autant qu'il y a de workers utilisés pour la parallélisation. Le parcours suivant est lui aussi fait en parallèle et les informations sont finalement agrégées par le nœud gather.

PARALLÉLISME SUR UNE AGRÉGATION

```
EXPLAIN (ANALYZE)
SELECT count(*), min(C1), max(C1) FROM t1;

                          QUERY PLAN
-----------------------------------------------------------------------
 Finalize Aggregate
 (actual time=1766.820..1766.820 rows=1 loops=1)
   ->  Gather
       (actual time=1766.767..1766.799 rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate
             (actual time=1765.236..1765.236 rows=1 loops=3)
               ->  Parallel Seq Scan on t1
                   (actual time=0.021..862.430 rows=6666667
                    loops=3)
 Planning time: 0.072 ms
 Execution time: 1769.164 ms
(8 rows)
```

Le principe est globalement le même.

Pour les fonctions type min et max, chacun des workers va récupérer la valeur minimale ou maximale des données qu'il traite, et ensuite le nœud Gather va se charger de comparer les valeurs intermédiaires pour renvoyer la valeur minimale ou maximale pour l'ensemble des données.

Pour une fonction de comptage, de même, chacun des workers compte le nombre de lignes qu'il traite et le nœud Gather fait la somme de l'ensemble.

Pour la fonction average, chacun des nœuds renvoie la somme des valeurs traitées et le nombre de valeurs traitées. Le nœud Dather se charge de

calculer réellement la moyenne.

Les valeurs intermédiaires sont calculées par les nœuds `Partial Aggregate`, et la valeur finale est renvoyée par le nœud `Finalize Aggregate`.

PARALLÉLISATION : LIMITATIONS

- Les fonction doivent être *Parallel Safe*
- Il faut indiquer au planificateur qu'une fonction peut être utilisée :
  - Dans un gather et worker : `PARALLEL SAFE`
  - Dans un gather uniquement `RESTRICTED`
  - Ne peut pas être utilisée pour la parallélisation : `PARALLEL UNSAFE`
- Pas de parallélisation pour :
  - Requête effectuant des écritures ou posant des verrous
  - Une requête qui pourrait être suspendue pendant son exécution. Exemple `CURSOR`
  - Une sous requête exécutée dans une requête déjà parallélisée.
  - Niveau d'isolation *serializable*

INDEX BLOOM

Infrastructure pour de nouvelles méthodes d'accès sous forme d'extensions.

- Premier exemple : les index **bloom**
  - extension
  - basés sur la probabilité de présence, avec perte
  - plus petits mais impliquent un accès à la table
  - permettent d'avoir un index pour des **recherches multicritères** si les btree sont impraticables ou trop gros
  - égalité seulement pour entiers et chaînes

Cette version apporte l'infrastructure pour ajouter de nouvelles méthodes d'accès sous forme d'extension. Il sera ainsi possible d'ajouter un nouveau type d'index sans dépendre de la version majeure (à partir de la 9.6 bien entendu). Actuellement seul le support des index bloom a été ajouté. Il y a de fortes chances pour que nouveaux types d'index soient proposés sous forme d'extension.

DIVERS - CHECKPOINT

- Avant
  - écriture aléatoire car blocs lus dans l'ordre du cache
- Après
  - parcours intégral des blocs pour trier et mutualiser les écritures
- `checkpoint_flush_after` : Flush les données sur disques tous les X octets
  - Permet d'éviter le tuning des options `dirty_background_*` du kernel

L'opération checkpoint a été améliorée dans cette version.

L'opération checkpoint permet d'écrire réellement les données se trouvant encore dans le cache disque de PostgreSQL et de valider définitivement que ces dernières ne pourront pas être perdues (elles ont été écrites précédemment dans les WAL).

Auparavant, le checkpoint synchronisait ces données dans l'ordre dans lequel il les trouvait en cache vers les fichiers. Désormais, il réalise un parcours intégral des données, ce qui lui permet de synchroniser les données en fonction de leur destination : par tablespace, puis par relation, puis par numéro de fork, puis par numéro de bloc.

DIVERS - TRI

- Gain sur les opérations de tris
  - `replacement_sort_tuples`
  - Meilleure utilisation du cache des CPU

Lorsque le nombre d'enregistrements à trier est de moins de `replacement_sort_tuples`, le moteur utilise un autre algorithme de tri plus performant qui exploite mieux le cache du CPU.

ADMINISTRATION

- Gestion des slots de réplication pour `pg_basebackup`
- Sauvegardes à chaud concurrentes
- Réplication synchrone sur plusieurs nœuds
- Amélioration des Foreign Data Wrapper
- Amélioration du freeze
  - et nouvelle extension : `pg_visibility`
- Divers
  - `pg_config`, `pg_control*`

SLOTS DE RÉPLICATION POUR PG_BASEBACKUP

- Nouvelle option
  - `-S slotname` ou `--slot=slotname`
  - s'utilise obligatoirement avec `-X stream`
- Plus besoin d'archivage pour créer un serveur secondaire
  - sans aucun risque

Voici la commande complète pour créer un secondaire à distance :

```
$ /usr/pgsql-9.6/bin/pg_basebackup  -d 'host=127.0.0.1 port=5432
user=replication password=repli application_name=standby2' \
-D /pg_data/9.6/instance2/ -R -S standby2 -X stream
```

Cette fonctionnalité permet à `pg_basebackup` de connaître le slot de réplication à utiliser et donc de le définir automatiquement dans le fichier `recovery.conf` grâce à l'option `-R`.

Dans le cas d'une instance secondaire qui utilise les slots de réplication, cela permet aussi d'assurer que le serveur primaire va conserver tous les WAL nécessaires au serveur esclave tant que la restauration n'est pas terminée. Ainsi, on peut facilement démarrer le serveur esclave sans mettre en place d' archivage des journaux.

SAUVEGARDES À CHAUD CONCURRENTES

- Déjà possible avec `pg_basebackup`...
  - mais pas en manuel avec `pg_start_backup()`
- Nouvelle API
- `pg_start_backup()` et `pg_stop_backup()`
  - paramètre supplémentaire pour utilisation concurrente
  - par défaut à false
- Contraintes lorsqu'on utilise l'option concurrente
  - maintien de la session
  - plus de `backup_label`
  - => Privilégier des outils supportant cette fonctionnalité.

Une nouvelle API existe désormais pour réaliser des sauvegardes physiques

concurrentes. En effet, avant la version 9.6, il était impossible de réaliser plusieurs sauvegardes physiques simultanées avec la méthode s'appuyant sur les commandes `pg_start_backup()` et `pg_stop_backup()`.

Une nouvelle API est donc disponible depuis la 9.6 qui permet de lancer plusieurs sauvegardes physiques en parallèle.

Néanmoins, les contraintes concernant la réalisation de cette sauvegarde sont bien plus importantes. En effet, la session qui exécute la commande `pg_start_backup()` doit être la même que celle qui exécute `pg_stop_backup()`. Si la connexion venait à être interrompue, alors la sauvegarde doit être considérée comme invalide.

Autre point important, lors d'une sauvegarde non concurrente, le fichier `backup.label` est créé dans le répertoire de l'instance sauvegardée. Dans le cas des sauvegardes concurrentes, ce fichier serait écrasé.

Pour pallier à ce problème, c'est la commande `pg_stop_backup()` qui renvoie les informations qui se trouvaient avant dans le fichier `backup_label` et elle se charge dans le même temps de créer un fichier `000000010000000100000000XX.000000XX.backup` contenant les informations de fin de sauvegarde. De ce fait, si vous voulez implémenter cette nouvelle méthode, il vous faudra récupérer et conserver vous-même les informations renvoyées par la commande de fin de sauvegarde.

La sauvegarde PITR devient donc plus complexe, il est donc recommandé d' utiliser pg_basebackup ou des outils supportant ces fonctionnalités (pitrery, pg_backrest ...).

RÉPLICATION SYNCHRONE, L'EXISTANT

- 9.0, réplication asynchrone
- 9.1, réplication synchrone
  - plusieurs esclaves potentiellement synchrones
  - mais un seul synchrone à un instant t

La version de PostgreSQL 9.0 a apporté une vraie réplication en disposant d'un envoi plus fluide des informations grâce à la streaming replication, mais aussi avec la possibilité d'accéder en lecture seule au serveur répliqué.

La réplication synchrone arrive dans PostgreSQL avec la version 9.1. Plusieurs serveurs secondaires pouvaient être indiqués comme potentiellement

synchrones, mais un seul était réellement synchrone à un instant t. Autrement dit, le serveur primaire attendait la confirmation par le serveur secondaire de l'inscription des données sur son disque pour renvoyer le succès de l'opération à l'utilisateur.

En cas d'indisponibilité du serveur secondaire synchrone, le serveur suivant dans la liste des serveurs synchrones potentiels configurés avec le paramètre `synchronous_standby_names` prenait la main comme serveur synchrone.

Si aucun serveur potentiellement synchrone n'était disponible, la validation des écritures sur le serveur primaire était mise en attente.

SYNCHRONOUS_COMMIT

- Positionné à ON par défaut
- Soit le client, soit le maître
- Les modifications sont écrites dans les WAL avant d'envoyer la confirmation
- Nouvelle valeur :
  - `remote_apply`

La valeur du paramètre `synchronous_commit` permet de définir les conditions nécessaires pour pouvoir renvoyer à l'utilisateur que sa transaction a été validée.

Ce paramètre, dans le cadre d'un serveur synchrone répliqué, est utilisé pour valider au serveur primaire qu'une transaction a été validée sur le serveur secondaire.

Il peut prendre plusieurs valeurs :

- `off` : La transaction est directement validée, mais elle pourra être écrite plus tard dans les WAL. Cela correspond au maximum à 3 fois la valeur de `wal_writer_delay` (200ms par défaut). Ce paramétrage peut causer la perte de certaines transactions si le serveur se crashe et que les données n'ont pas encore été écrites dans les fichiers WAL.
- `local` : Permet de fonctionner, pour les prochaines requêtes de la session, en mode de réplication asynchrone. Le commit est validé lorsque les données ont été écrites et synchronisées sur le disque de l'instance primaire. En revanche, l' instance primaire ne s'assure pas que le secondaire a reçu la transaction.
- `remote_write` : Permet d'avoir de la réplication synchrone en mémoire. Cela veut dire que la donnée n'est pas écrite sur disque au niveau de l'esclave mais il l'a en mémoire. Les modifications sont écrites sur disque via le système d' exploitation, mais sans avoir demandé le vidage du cache système sur disque. Les informations sont donc dans la mémoire système. Cela permet de gagner beaucoup en performance, avec une fenêtre de perte de données bien moins importante que le mode asynchrone, mais toujours présente. Si c'est l'instance primaire PostgreSQL qui se crashe, les informations ne sont pas perdues. Par contre, il est possible de perdre des données si l'instance secondaire crashe (en cas de bascule).
- `on` (par défaut) : Le commit est validé lorsque les données ont été écrites et synchronisées sur le disque de l'instance primaire et secondaire. C'est la réplication synchrone. En revanche, l'instance primaire ne s'assure pas que le secondaire a **rejoué** la transaction.

La version 9.6 apporte une nouvelle valeur pour ce paramètre : `remote_apply`.

- `remote_apply`. Les modifications doivent être enregistrées dans les WAL et rejouées avant que la confirmation ne soit envoyée. Cette méthode est la seule garantissant qu'une transaction validée sur le maître sera visible sur le secondaire. Cependant, elle rajoute une latence supplémentaire.

Tableau récapitulatif :

Table 1: Valeur paramètre `synchronous_commit`

Ecriture	WAL local (synchronisé sur disque)	Mémoire distant (non-synchronisé sur disque)	WAL distant (synchronisé sur disque)	Rejeu distant (enregistrement visible)
`off`				
`remote_write`	X	X		

Table 1: Valeur paramètre `synchronous_commit`

`local`	X		
`on`	X	X	
`remote_apply`	X	X	X

RÉPLICATION SYNCHRONE SUR PLUSIEURS NŒUDS

- Plusieurs serveurs synchrones
- Configuration
  - synchronous_standby_names = 'N (liste serveurs)'
  - N = nombre serveurs synchrones
- Contraintes
  - latence induite par la validation des transactions sur plusieurs esclaves
  - requêtes en attente sur le primaire s'il y a moins de N serveurs synchrones disponibles

La 9.6 apporte la possibilité de réaliser de la réplication synchrone avec plusieurs esclaves.

En effet, la déclaration d'un esclave synchrone se fait en définissant le paramètre `synchronous_standby_names`. Ce dernier possède une nouvelle syntaxe telle que : `'N (<Slave_n°1>, <Slave_n°2>, ...)'`.

Cette méthode, bien utilisée, permet d'avoir plusieurs serveurs répliqués avec réellement les mêmes informations que le serveur maître.

Néanmoins, les contraintes amenées par cette méthode ne sont pas légères :

- La latence est induite par l'attente de la validation des transactions répliquées sur l'ensemble des serveurs répliqués définis.
- Si 2 esclaves synchrones ont été définis, les requêtes se bloquent dès l' absence d'un des deux esclaves.
- Lorsque les transactions sont suspendues sur le maître dans ce cadre, l' instance maître n'a aucune information à ce sujet.

Voici un exemple de ce qu'il se passe quand le nombre de serveurs synchrones n' est pas suffisant.

Cette écriture va être bloquée :

```
INSERT INTO text_exemple
  VALUES ('la recherche fonctionne souvent par hasard deuxième');
```

Cela ne se voit pas dans la vue `pg_stat_activity` :

```
SELECT * FROM pg_stat_activity;
-[ RECORD 1 ]----+----------------------------------
datid            | 13322
datname          | postgres
pid              | 2075
usesysid         | 10
usename          | postgres
application_name | psql
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2016-08-18 08:18:17.999042+02
xact_start       | 2016-08-18 08:18:42.488234+02
query_start      | 2016-08-18 08:18:42.488234+02
state_change     | 2016-08-18 08:18:42.488241+02
wait_event_type  |
wait_event       |
state            | active
backend_xid      |
backend_xmin     | 1914
query            | select * from pg_stat_activity;
```

Cependant, on peut tout de même retrouver cette information en consultant les processus côté système :

```
postgres 22395  0.0  7.0 484864 283600 ?       Ss   août17   0:55
    \_ postgres: postgres textes [local] INSERT waiting for 1/53410200
```

FOREIGN DATA WRAPPER

- Norme SQL/MED
- Interroger des données externes.
- Données externes présentées comme des tables
- Attention aux performances !

Il existe dans PostgreSQL la possibilité de travailler sur des bases distantes grâce aux connecteurs appelés `Foreign Data Wrapper`.

Le FDW pour PostgreSQL, appelé `postgres_fdw`, est disponible sous la forme d'une extension à installer sur la base où l'on veut créer des tables distantes.

Le principe est le suivant. En s'appuyant sur l'extension `postgres_fdw`, il est possible de créer une table dans la base courante, pointant sur une table de structure identique mais se trouvant dans une autre instance PostgreSQL.

```
CREATE FOREIGN TABLE table1_distante (
  id serial,
  bla text)
SERVER serveurdistant
OPTIONS (table_name 'table1');
```

Lorsqu'on fait une extraction sur cette table, un FDW est capable de ne récupérer que les lignes nécessaires, la clause `WHERE` pouvant être envoyée au serveur secondaire. D'autres optimisations ont été apportées au fil des versions mais certaines manquent toujours à l'appel.

FDW : JEU DE TESTS

```
CREATE TABLE t1 (id integer);
INSERT INTO t1 SELECT generate_series(1, 10000000);
CREATE INDEX ON t1(id);

CREATE EXTENSION postgres_fdw;

CREATE SERVER ailleurs FOREIGN DATA WRAPPER postgres_fdw;

CREATE USER MAPPING FOR postgres SERVER ailleurs;

CREATE FOREIGN TABLE ft1(id integer)
  SERVER ailleurs
  OPTIONS (table_name 't1');
```

FDW : NOUVEAUTÉS

- SORT pushdown
  - exécution du tri sur le serveur distant
- JOIN pushdown
  - exécution de la jointure sur le serveur distant

La version 9.6 apporte désormais la possibilité de réaliser les opérations de jointure et de tri à distance, et de ne récupérer que la partie nécessaire pour la sélection demandée

dans l'ordre requis.

Dans certains cas, cela va apporter un gain de performance notoire.

FDW : TRI EN 9.5

```
EXPLAIN (ANALYZE, VERBOSE, COSTS off)
SELECT * FROM ft1 ORDER BY id ;

                          QUERY PLAN
---------------------------------------------------------------
 Sort (actual time=9452.057..10129.182 rows=10000000 loops=1)
   Output: id
   Sort Key: ft1.id
   Sort Method: external sort  Disk: 136856kB
   ->  Foreign Scan on public.ft1
       (actual time=1.064..5981.536 rows=10000000 loops=1)
         Output: id
         Remote SQL: SELECT id FROM public.t1
 Planning time: 0.131 ms
 Execution time: 10425.730 ms
(9 rows)
```

FDW : TRI EN 9.6

```
EXPLAIN (ANALYZE, VERBOSE, COSTS off)
SELECT * FROM ft1 ORDER BY id;

                       QUERY PLAN
-------------------------------------------------
 Foreign Scan on public.ft1
 (actual time=2.092..7438.416 rows=10000000 loops=1)
   Output: id
   Remote SQL: SELECT id FROM public.t1
               ORDER BY id ASC NULLS LAST
 Planning time: 0.168 ms
 Execution time: 7748.122 ms
(5 rows)
```

On constate bien dans le second plan que la clause `ORDER BY` a été faite a distance ce qui n'était pas le cas dans la version précédente.

FDW : JOINTURE EN 9.5

```
EXPLAIN (VERBOSE, ANALYZE)
SELECT * FROM table1_distante
JOIN table2_distante ON table1_distante.id = table2_distante.id
WHERE table2_distante.id = 1;

                        QUERY PLAN
-----------------------------------------------------------
 Nested Loop  [...]
   Output: table1_distante.id, table1_distante.bla,
           table2_distante.id, table2_distante.bli
   ->  Foreign Scan on public.table1_distante  [...]
         Output: table1_distante.id, table1_distante.bla
         Remote SQL: SELECT id, bla FROM public.table1
                     WHERE ((id = 1))
   ->  Foreign Scan on public.table2_distante  [...]
         Output: table2_distante.id, table2_distante.bli
         Remote SQL: SELECT id, bli FROM public.table2
                     WHERE ((id = 1))
 Planning time: 0.217 ms
 Execution time: 65.616 ms
```

En version 9.5, on peut constater sur ce plan d'exécution que les sélections complètes des tables sont récupérées depuis la base distante (partie `Remote SQL`), puis la sélection demandée est réalisée sur place (partie `Nested Loop`)

FDW : JOINTURE EN 9.6

```
EXPLAIN (VERBOSE, ANALYZE)
SELECT * FROM table1_distante
JOIN table2_distante
  ON table1_distante.id=table2_distante.id
WHERE table2_distante.id = 1;

                        QUERY PLAN
-----------------------------------------------------------
 Foreign Scan  [...]
   Output: table1_distante.id, table1_distante.bla,
           table2_distante.id, table2_distante.bli
```

```
  Relations: (public.table1_distante)
             INNER JOIN (public.table2_distante)
  Remote SQL: SELECT r1.id, r1.bla, r2.id, r2.bli
              FROM (public.table1 r1
              INNER JOIN public.table2 r2
                 ON (((r2.id = 1)) AND ((r1.id = 1))))
Planning time: 0.170 ms
Execution time: 2.299 ms
```

De la même manière, toute la requête est réalisée sur le serveur distant en 9.6.

FREEZE

- Amélioration de la fonction de gel des données
- pg_visibility(oid)
  - blkno
  - all_visible
  - all_frozen
  - pd_all_visible

La Visibility Map est un tableau qui, avant la version 9.6, permettait de savoir, pour un bloc de données donné, si toutes les lignes qu'il contenait étaient des lignes "visibles", c'est-à-dire des lignes non supprimées ou non obsolètes suite à la mise à jour.

Elle a récupéré dans la version 9.6 une nouvelle information permettant de

savoir, de la même manière, pour un bloc de données, si ce dernier contenait des lignes gelées.

Ces informations sont importantes dans le cas de l'exécution d'un `VACUUM FREEZE` pour que ce dernier ne traite pas des blocs qui n'en ont pas besoin. Diminuer le nombre de lectures et de traitements permet de rendre l'opération plus rapide.

Nous intéresser à cette amélioration nous permet de nous pencher aussi sur l'extension `pg_visibility`, arrivée elle aussi avec la version 9.6 de PostgreSQL.

- `blkno` : numéro de bloc
- `all_visible` : booléen, true si toutes les lignes du bloc sont visibles
- `all_frozen` : booléen, true si toutes les lignes du bloc sont gelées
- `pd_all_visible` : identique à all_visible mais stocké dans l'entête du bloc lui-même.

Il peut exister une différence entre all_visible et pd_all_visible. Il faut dans ce cas faire confiance à la valeur se trouvant dans all_visible. En effet, en mode recovery, pd_all_visible peut parfois rester défini tandis que la visibility map est purgée. Ils peuvent aussi être différents si `pg_visibility()` examine la visibility map et qu'un changement survienne avant que la fonction ait examiné le bloc de données correspondant.

PG_CONFIG, PG_CONTROL

- Outil pg_config
  - informations de compilation
- Outil pg_controldata
  - informations du fichier pg_control
- Informations uniquement disponibles sur la ligne de commande avant la 9.6
- Maintenant disponibles aussi via l'interface SQL
  - vue pg_config
  - fonctions pg_control_system(), pg_control_init(), pg_control_checkpoint() et pg_control_recovery()

L'outil `pg_config` permet de récupérer un ensemble d'informations sur la configuration de la compilation :

```
$ pg_config
BINDIR = /opt/postgresql/96/bin
DOCDIR = /opt/postgresql/96/share/doc
HTMLDIR = /opt/postgresql/96/share/doc
INCLUDEDIR = /opt/postgresql/96/include
PKGINCLUDEDIR = /opt/postgresql/96/include
INCLUDEDIR-SERVER = /opt/postgresql/96/include/server
LIBDIR = /opt/postgresql/96/lib
PKGLIBDIR = /opt/postgresql/96/lib
LOCALEDIR = /opt/postgresql/96/share/locale
MANDIR = /opt/postgresql/96/share/man
SHAREDIR = /opt/postgresql/96/share
SYSCONFDIR = /opt/postgresql/96/etc
PGXS = /opt/postgresql/96/lib/pgxs/src/makefiles/pgxs.mk
CONFIGURE = '--prefix=/opt/postgresql/96' '--enable-nls' \
 '--with-openssl' '--with-libxml' '--enable-thread-safety' \
 '--enable-debug' '--with-wx-version=3.1'
```

```
CC = gcc
CPPFLAGS = -DFRONTEND -D_GNU_SOURCE -I/usr/include/libxml2
CFLAGS = -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-statement \
 -Wendif-labels -Wmissing-format-attribute -Wformat-security -fno-strict-aliasing \
 -fwrapv -fexcess-precision=standard -g -O2
CFLAGS_SL = -fpic
LDFLAGS = -L../../src/common -Wl,--as-needed \
 -Wl,-rpath,'/opt/postgresql/96/lib',--enable-new-dtags
LDFLAGS_EX =
LDFLAGS_SL =
LIBS = -lpgcommon -lpgport -lxml2 -lssl -lcrypto -lz -lreadline -lrt -lcrypt -ldl -lm
VERSION = PostgreSQL 9.6rc1
```

Ces informations sont maintenant disponibles via l'interface SQL :

```
postgres=# SELECT * FROM pg_config;
-[ RECORD 1 ]--------------------------
name    | BINDIR
setting | /opt/postgresql/96/bin
-[ RECORD 2 ]--------------------------
name    | DOCDIR
setting | /opt/postgresql/96/share/doc
[...]
```

L'outil pg_controldata est maintenant doublé par un ensemble de fonctions SQL permettant de récupérer des informations contenues dans le fichier `global/pg_control` :

```
postgres=# select * from pg_control_checkpoint();
-[ RECORD 1 ]--------+-------------------------
checkpoint_location  | 4/6985EE10
prior_location       | 4/6985ED30
redo_location        | 4/6985EDD8
redo_wal_file        | 000000010000000400000069
timeline_id          | 1
prev_timeline_id     | 1
full_page_writes     | t
next_xid             | 0:3447
next_oid             | 91758
next_multixact_id    | 1
next_multi_offset    | 0
oldest_xid           | 1797
```

```
oldest_xid_dbid       | 1
oldest_active_xid     | 3447
oldest_multi_xid      | 1
oldest_multi_dbid     | 1
oldest_commit_ts_xid  | 0
newest_commit_ts_xid  | 0
checkpoint_time       | 2016-09-02 11:47:56+02
```

Voici le détail sur les fonctions en question :

- `pg_control_system()` renvoie des informations sur l'état courant du fichier de contrôle.
- `pg_control_init()` renvoie la configuration de l'instance (taille d'un bloc d'un fichier de données ou d'un journal de transactions, le nombre de blocs par segment dans un fichier de données, le nombre d'octet par segment dans un journal de transactions, le nombre maximum de colonnes pour un index, ...).
- `pg_control_checkpoint()` renvoie des informations sur l'état du checkpoint courant (lorsque l' instance réalise un checkpoint, la position de celui-ci est sauvegardée dans le fichier `pg_control`).
- `pg_control_recovery()` renvoie des informations sur l'état de la récupération en cours, n'est utilisable que sur une instance dans cet état : vide si le serveur est ouvert en lecture/écriture.

SQL

- Recherche plein-texte pour les phrases
- Option CASCADE pour les extensions
- Pivotage des résultats
- Autres nouveautés
  - fonction mathématiques
  - `jsonb_insert()`

RECHERCHE PLEIN-TEXTE POUR LES PHRASES

- FTS disponible dans PostgreSQL depuis la version 8.3
- Opérateurs disponibles jusqu'à présent
  - ET logique : &
  - OU logique : |
  - NON logique : !
- Nouvel opérateur :
  - Succession : <->
  - Position dans la phrase : (où N est un entier représentant le nombre de mot séparant les deux motifs)

Avant que la recherche plein texte soit disponible dans le moteur (en 8.3), la seule manière de rechercher des expressions dans des tables se faisait avec la commande LIKE ou une expression rationnelle.

Celle-ci est très longue, limitée par le motif employé et il est très rarement possible d'utiliser l'indexation (un index Btree est inutilisable pour un `LIKE '%mot%'`).

La recherche plein texte s'appuie sur la racine native de l'expression donnée. Celle-ci est dépendante de la langue utilisée :

```
SHOW default_text_search_config;

 default_text_search_config
----------------------------
 pg_catalog.french
```

```
SELECT to_tsvector('empêcherais');

 to_tsvector
-------------
 'empêch':1
```

Combien de textes possèdent la racine de *empêcherais*

```
EXPLAIN ANALYZE SELECT count(*)
FROM textes WHERE to_tsvector('french',contenu) @@ to_tsquery('empêcherais');
                              QUERY PLAN
-----------------------------------------------------------------------------------
 Aggregate  (cost=11146387.36..11146387.37 rows=1 width=8)
   (actual time=282384.083..282384.083 rows=1 loops=1)
   ->  Seq Scan on textes  (cost=0.00..11146125.45 rows=104764 width=0)
     (actual time=1.256..282372.941 rows=32141 loops=1)
         Filter: (to_tsvector('french'::regconfig, contenu) @@
           to_tsquery('empêcherais'::text))
```

```
        Rows Removed by Filter: 20917562
 Planning time: 192.846 ms
 Execution time: 282384.109 ms
(6 rows)
```

L'idéal pour optimiser les performances est de créer un index fonctionnel avec la fonction `to_tsvector()` :

On peut utiliser un index GIN :

```
CREATE INDEX index_contenu_vectorise
  ON text_exemple
  USING gin ( to_tsvector('french',contenu) );
```

Cette fois la requête est bien plus rapide :

```
EXPLAIN ANALYZE SELECT count(*)
FROM textes
WHERE to_tsvector('french',contenu) @@ to_tsquery('empêcherais');
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate  (cost=293383.13..293383.14 rows=1 width=8)
   (actual time=159.360..159.361 rows=1 loops=1)
   ->  Bitmap Heap Scan on textes  (cost=984.05..293121.26 rows=104749 width=0)
     (actual time=9.216..156.727 rows=32141 loops=1)
         Recheck Cond: (to_tsvector('french'::regconfig, contenu) @@
           to_tsquery('empêcherais'::text))
         Heap Blocks: exact=29635
         ->  Bitmap Index Scan on index_contenu_vectorise  (cost=0.00..957.86
             rows=104749 width=0) (actual time=4.979..4.979 rows=32141 loops=1)
               Index Cond: (to_tsvector('french'::regconfig, contenu) @@
                 to_tsquery('empêcherais'::text))
 Planning time: 18.162 ms
 Execution time: 159.397 ms
(8 rows)
```

Note, voici ce que donnait la requête sans index mais avec parallélisation :

```
EXPLAIN ANALYZE SELECT count(*)
FROM textes
WHERE to_tsvector('french',contenu) @@ to_tsquery('empêcherais');
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Finalize Aggregate  (cost=2198596.85..2198596.86 rows=1 width=8) (actual
   time=65043.325..65043.325 rows=1 loops=1)
   ->  Gather  (cost=2198596.23..2198596.84 rows=6 width=8) (actual
        time=65042.856..65043.315 rows=7 loops=1)
         Workers Planned: 6
         Workers Launched: 6
```

```
        ->  Partial Aggregate  (cost=2197596.23..2197596.24 rows=1 width=8)
           (actual time=65037.435..65037.435 rows=1 loops=7)
              ->  Parallel Seq Scan on textes  (cost=0.00..2197552.58 rows=17461
                 width=0) (actual time=14.707..65035.315 rows=4592 loops=7)
                    Filter: (to_tsvector('french'::regconfig, contenu) @@
                     to_tsquery('empêcherais'::text))
                    Rows Removed by Filter: 2988223
 Planning time: 0.108 ms
 Execution time: 65046.598 ms
```

Un index fonctionnel est donc bien plus performant.

```
textes=# SELECT count(*) FROM textes WHERE contenu LIKE '%empêcherais%';
 count
-------
    31
(1 row)

textes=# SELECT count(*)
FROM textes
WHERE to_tsvector('french',contenu) @@ to_tsquery('empêcherais');
 count
-------
 32141
(1 row)
```

On constate bien par cet exemple que la recherche par motif est très différente de la recherche plein texte. En effet, lorsqu'on utilise le motif, on ne va pas pouvoir récupérer de déclinaison du mot, si ce mot n'est pas tout à fait bien orthographié, de la même manière, il ne matchera pas.

Concernant la nouvelle fonctionnalité :

Avant, lorsqu'on désirait rechercher une phrase, on était obligé de faire d' abord la sélection des mots avec FTS (ce qui permettait d'utiliser l'index de manière à améliorer la rapidité de l'exécution) puis de filtrer avec un LIKE pour obtenir exactement la combinaison de mots désirée :

```
textes=# SELECT *
FROM textes
WHERE to_tsvector('french',contenu) @@ to_tsquery('monter&à&cheval')
  AND contenu LIKE '%monter à cheval%';
```

Maintenant, on peut utiliser notre nouvel opérateur ("monter à cheval" et variations à un mot de distance)

```
textes=# SELECT contenu
FROM textes
```

```
WHERE  to_tsvector('french',contenu) @@ to_tsquery('monter<->à<->cheval');
```

Il se décline de manière à ne pas avoir forcément des mots se succédant, par exemple :

```
textes=# SELECT contenu
FROM textes
WHERE  to_tsvector('french',contenu) @@ to_tsquery('monter<3>cheval');
```

Le chiffre 3 dans <3> indique la distance entre *monter* et *cheval*.

AJOUT DE L'OPTION CASCADE POUR LES EXTENSIONS

- Avant la version 9.6

```
postgres=# CREATE EXTENSION ltree_plpythonu;
ERREUR:  l'extension « ltree » requise n'est pas installée
```

- Après la version 9.6

```
postgres=# CREATE EXTENSION ltree_plpythonu CASCADE;
NOTICE:  installing required extension "ltree"
NOTICE:  installing required extension "plpythonu"
CREATE EXTENSION
```

On obtient désormais la possibilité d'utiliser l'option CASCADE lorsque l'on crée une extension. En effet, avant la version 9.6, il fallait gérer les dépendances de façon manuelle, en installant chacune des extensions supplémentaires nécessaires.

PIVOTAGE DES RÉSULTATS

- `\crosstabview [ colV [ colH [ colD [ sortcolH ] ] ] ]`
- Exécute le contenu du tampon de requête
  - la requête doit renvoyer au moins 3 colonnes
- Affiche le résultat dans une grille croisée
  - colV, en-tête vertical
  - colH, en-tête horizontal
  - colD, contenu à afficher dans le tableau
  - sortColH, colonne de tri pour l'en-tête horizontal

Disons que nous voulons récupérer le nombre de bouteilles en stock par type de vin (blanc, rosé, rouge) par année, pour les années entre 1950 et 1959 :

```
cave=# SELECT t.libelle, s.annee, sum(s.nombre)
FROM stock s
  JOIN vin v ON v.id=s.vin_id
  JOIN type_vin t ON t.id=v.type_vin_id
WHERE s.annee BETWEEN 1950 AND 1959
GROUP BY t.libelle, s.annee
ORDER BY s.annee;

 libelle | annee |  sum
---------+-------+-------
 blanc   |  1950 | 69166
 rose    |  1950 | 70311
 rouge   |  1950 | 69836
 blanc   |  1951 | 67325
 rose    |  1951 | 67616
 rouge   |  1951 | 66708
 blanc   |  1952 | 67501
 rose    |  1952 | 67481
 rouge   |  1952 | 66116
 blanc   |  1953 | 67890
 rose    |  1953 | 67902
 rouge   |  1953 | 67045
 blanc   |  1954 | 67471
 rose    |  1954 | 67759
 rouge   |  1954 | 67299
 blanc   |  1955 | 67306
 rose    |  1955 | 68015
 rouge   |  1955 | 66854
 blanc   |  1956 | 67281
 rose    |  1956 | 67458
 rouge   |  1956 | 66990
 blanc   |  1957 | 67323
 rose    |  1957 | 67374
 rouge   |  1957 | 66409
 blanc   |  1958 | 67469
 rose    |  1958 | 67738
 rouge   |  1958 | 66782
 blanc   |  1959 | 67765
 rose    |  1959 | 67903
 rouge   |  1959 | 67560
(30 rows)
```

Et maintenant nous souhaitons afficher ces informations avec en abscisse le libellé du type de vin et en ordonnée les années :

```
cave=# \crosstabview
 libelle | 1950  | 1951  | 1952  | 1953  | 1954  | ... | 1959
```

```
---------+-------+-------+-------+-------+-------+-----+-------
 blanc   | 69166 | 67325 | 67501 | 67890 | 67471 | ... | 67765
 rose    | 70311 | 67616 | 67481 | 67902 | 67759 | ... | 67903
 rouge   | 69836 | 66708 | 66116 | 67045 | 67299 | ... | 67560
(3 rows)
```

Et si nous souhaitons l' inverse (les années en abscisse et les types de vin en ordonnée), c'est tout aussi simple :

```
cave=# \crosstabview annee libelle
 annee | blanc | rose  | rouge
-------+-------+-------+-------
  1950 | 69166 | 70311 | 69836
  1951 | 67325 | 67616 | 66708
  1952 | 67501 | 67481 | 66116
  1953 | 67890 | 67902 | 67045
  1954 | 67471 | 67759 | 67299
  1955 | 67306 | 68015 | 66854
  1956 | 67281 | 67458 | 66990
  1957 | 67323 | 67374 | 66409
  1958 | 67469 | 67738 | 66782
  1959 | 67765 | 67903 | 67560
(10 rows)
```

AUTRES NOUVEAUTÉS

- Amélioration des performances globales pour plusieurs fonctions mathématiques
  - ln() / log() / exp() / pow()
- Ajout des fonctions
  - sind() / cosd() / tand()

Les performances globales de différentes fonctions mathématiques telles que logarithme, exponentielle, puissance, ... ont été améliorées.

Les fonctions `sind()`, `cosd()` et `tand()` ont été créées, permettant de calculer sinus, cosinus, tangente avec une valeur en degrés plutôt qu'avec une valeur en radian, ce qui permet d'obtenir les résultats exacts, plutôt qu'inexacte après une conversion. Par exemple, la tangente de 90° retourne la valeur infinie.

```
textes=# SELECT cos(3.14159);
        cos
```

```
--------------------
 -0.999999999996479
(1 ligne)

textes=# SELECT cosd(180);
 cosd
------
   -1
(1 ligne)

textes=# SELECT tan(1.5708);
        tan
-------------------
 -272241.808409276
(1 ligne)


textes=# SELECT tand(90);
   tand
----------
 Infinity
(1 ligne)
```

MONITORING

- Nouvelles colonnes dans `pg_stat_activity`
- Nouvelle fonction `pg_blocking_pid()`
- Nouvelle vue `pg_stat_progress_vacuum`

VUE PG_STAT_ACTIVITY, PG_BLOCKING_PIDS()

- Avant la 9.6
  - identification complexe d'une requête bloquante
  - et de la raison du blocage
- Ajout de la fonction `pg_blocking_pids()`
- modification de la colonne waiting en deux colonnes
  - wait_event
  - wait_event_type

Il était parfois compliqué d'identifier d'où provenait un blocage lorsqu' une requête ne parvenait pas à se terminer.

Dans un cas où deux 2 sessions ont chacune une transaction ouverte, l'une insérant des données dans une table et l'autre tentant de supprimer cette table, tant que la transaction d'insertion n'est pas terminée, la session qui cherche à supprimer la table attends.

Jusqu'à la version 9.5, on disposait dans la vue `pg_stat_activity` des informations suivantes :

```
postgres=# SELECT pid,state,waiting,query FROM pg_stat_activity;

pid     | 3279
state   | active
waiting | t
query   | drop table test_blocage ;


pid     | 3281
state   | idle in transaction
waiting | f
query   | INSERT INTO test_blocage (champ) (SELECT 'ligne '||i
        | FROM generate_series(1, 10000) i);
```

On constate bien que la session de pid 3279 est en attente : `waiting = t` (true) . Mais on n'a pas plus d'informations, notamment pourquoi celle-ci est bloquée, par qui etc ...

La colonne `waiting` a été séparée en deux colonnes, `wait_event_type` et `wait_event`, ce qui nous apporte plus d'informations.

```
postgres=# SELECT pid,wait_event_type,wait_event,state,query
FROM pg_stat_activity;

pid             | 2800
wait_event_type | Lock
wait_event      | relation
state           | active
query           | drop table test_blocage ;

pid             | 3085
wait_event_type |
wait_event      |
state           | idle in transaction
query           | INSERT INTO test_blocage (champ) (SELECT 'ligne '||i
                | FROM generate_series(1, 10000) i);
```

FONCTION PG_BLOCKING_PIDS()

- Avant la 9.6
  - requête complexe pour trouver la session bloquante
  - voire collection de requêtes
- Depuis la 9.6
  - fonction `pg_blocking_pids()`

Avant pour obtenir la requête bloquante, il fallait écrire la requête suivante :

```
SELECT
    waiting.locktype           AS waiting_locktype,
    waiting.relation::regclass AS waiting_table,
    waiting_stm.query  AS waiting_query,
    waiting.mode               AS waiting_mode,
    waiting.pid                AS waiting_pid,
    other.locktype             AS other_locktype,
    other.relation::regclass   AS other_table,
    other_stm.query    AS other_query,
    other.mode                 AS other_mode,
    other.pid                  AS other_pid,
    other.granted              AS other_granted
FROM
    pg_catalog.pg_locks AS waiting
JOIN
    pg_catalog.pg_stat_activity AS waiting_stm
    ON (
        waiting_stm.pid = waiting.pid
    )
JOIN
    pg_catalog.pg_locks AS other
    ON (
        (
            waiting."database" = other."database"
        AND waiting.relation  = other.relation
        )
        OR waiting.transactionid = other.transactionid
    )
JOIN
    pg_catalog.pg_stat_activity AS other_stm
    ON (
        other_stm.pid = other.pid
    )
WHERE NOT waiting.granted
AND waiting.pid <> other.pid;
```

```
waiting_locktype | relation
waiting_table    | test_blocage
waiting_query    | drop table test_blocage ;
waiting_mode     | AccessExclusiveLock
waiting_pid      | 3279
other_locktype   | relation
other_table      | test_blocage
other_query      | INSERT INTO test_blocage (champ) (SELECT 'ligne '||i
                 |  FROM generate_series(1, 10000) i);
other_mode       | RowExclusiveLock
other_pid        | 3281
other_granted    | t
```

Désormais il suffit de passer le pid de la requête bloquée :

```
postgres=# SELECT pg_blocking_pids(2800);
-[ RECORD 1 ]----+-------
pg_blocking_pids | {3085}
```

Dans notre cas, notre requête `drop table test_blocage;` est bloquée par la session de pid `3085`

PG_STAT_PROGRESS_VACUUM

- Intégration d'une API pour le suivi de progression d'une commande
- Utilisation de cette API pour VACUUM
- Vue pg_stat_progress_vacuum
  - nombreuses informations sur la progression d'une opération VACUUM

Les différentes phases dans lesquelles peut se trouver un `VACUUM` :

- `initializing` : `VACUUM` se prépare à lire les données non indexées.
- `scanning heap` : `VACUUM` parcourt les données non indexées. La colonne `heap_blks_scanned` peut donner une indication sur l'avancement de cette étape.
- `vacuuming indexes` : `VACUUM` procède au vacuum des index.
- `vacuuming heap` : `VACUUM` procède au vacuum des données non indexées.
- `cleaning up indexes` : `VACUUM` procède au nettoyage des index.
- `truncating heap` : `VACUUM` agrège les blocs vides des données non indexées de manière à ce qu' ils puissent être réutilisés.
- `performing final cleanup` : `VACUUM` réalise le nettoyage final, ce qui inclut le traitement de la FSM, la mise à jour des statistiques dans le catalogue système `pg_class` et le report de ces informations vers le collecteur de statistiques.

Et voici le contenu de la vue `pg_stat_progress_vacuum` :

```
View "pg_catalog.pg_stat_progress_vacuum"
       Column        |  Type   | Modifiers
---------------------+---------+-----------
 pid                 | integer |
 datid               | oid     |
 datname             | name    |
 relid               | oid     |
 phase               | text    |
 heap_blks_total     | bigint  |
 heap_blks_scanned   | bigint  |
 heap_blks_vacuumed  | bigint  |
 index_vacuum_count  | bigint  |
 max_dead_tuples     | bigint  |
 num_dead_tuples     | bigint  |
```

RÉGRESSIONS / CHANGEMENTS

- Il n'est plus possible de créer des rôles commençant par `pg_*`
- L'option -c de psql (exécution de commande) n'active plus –no-psqlrc
- L'option -t de `pg_restore` ne correspond plus seulement aux tables
  - mais à tous types de relations
- Niveaux `archive` et `hot_standby` n'existent plus pour `wal_level`.
  - Remplacés par un niveau `replica` <=> `hot_standby`

Les rôles commençant par `pg_*` sont réservés aux rôles systèmes de gestion de droits (comme `pg_signal_backend`).

Les niveaux `archive` et `hot_standby` n'existent plus pour `wal_level`. Ils sont remplacés par le niveau `replica`. Afin de conserver une compatibilité, les niveaux `archive` et `hot_standby` sont remplacé en interne par `replica`.

FUTUR

- Version 10, en septembre/octobre 2017
- Beta 1 depuis le 18 mai
- Fonctionnalités majeures
  - réplication logique
  - partitionnement natif de table (suite et liste)
  - amélioration du partitionnement (parcours d'index, de bitmap, jointure par tri)
  - quorum pour la réplication synchrone

La roadmap par société : https://wiki.postgresql.org/wiki/PostgreSQL10_Roadmap La documentation : https://www.postgresql.org/docs/10/static/index.html

---

QUESTIONS

ATELIER

INSTALLATION

Les machines de la salle de formation sont en CentOS 6. L'utilisateur dalibo peut utiliser sudo pour les opérations système.

Le site postgresql.org propose son propre dépôt RPM, nous allons donc l'utiliser.

```
rpm -ivh https://download.postgresql.org/pub/repos/yum/9.6/redhat/rhel-
6-x86_64/pgdg-centos96-9.6-3.noarch.rpm

Retrieving https://download.postgresql.org/pub/repos/yum/9.6/redhat/rhel-
6-x86_64/pgdg-centos96-9.6-3.noarch.rpm
Preparing...                ########################################### [100%]
   1:pgdg-centos96          ########################################### [100%]
```

```
yum install postgresql96 postgresql96-server postgresql96-contrib
```

Les paquets suivants seront installés :

```
(1/5): libxslt-1.1.26-2.el6_3.1.x86_64.rpm
(2/5): postgresql96-9.6.0-1PGDG.rhel6.x86_64.rpm
(3/5): postgresql96-contrib-9.6.0-1PGDG.rhel6.x86_64.rpm
(4/5): postgresql96-libs-9.6.0-1PGDG.rhel6.x86_64.rpm
(5/5): postgresql96-server-9.6.0-1PGDG.rhel6.x86_64.rpm
```

Commençons par créer une instance PostgreSQL 9.6 :

```
sudo /etc/init.d/postgresql-9.6 initdb
Initialisation de la base de données :                     [  OK  ]
```

Et par la lancer (ce n'est pas automatique sur RedHat/CentOS) :

```
sudo /etc/init.d/postgresql-9.6 start
```

Les fichiers de la base de données seront dans `/var/lib/pgsql/9.6/data`, y compris `postgresql.conf` et `pg_hba.conf`.

Pour se connecter sans modifier `pg_hba.conf` :

```
sudo -iu postgres psql
```

PARALLÉLISATION

Créer les tables suivantes :

```
CREATE TABLE t1 AS SELECT * FROM generate_series(1,51110000) id;
CREATE TABLE t2 AS SELECT * FROM generate_series(1,30000000) id;
CREATE TABLE t3 AS SELECT * FROM generate_series(1,3100000) id;
```

Modifier le paramètre `max_parallel_workers_per_gather` afin de permettre la parallélisation.

- Calculer la taille de chaque objet :

```
b1=# SELECT pg_relation_size('t1')/1024/8 AS t1_blocks_nb;
 t1_blocks_nb
--------------
   226151
(1 ligne)

b1=# SELECT pg_relation_size('t2')/1024/8 AS t2_blocks_nb;
 t2_blocks_nb
--------------
   132744
(1 row)

b1=# SELECT pg_relation_size('t3')/1024/8 AS t3_blocks_nb;
```

```
 t3_blocks_nb
--------------
    13717
(1 row)
```

- Chercher le plan d'un SELECT sur toute la table :

```
b1=# EXPLAIN (ANALYZE,BUFFERS,VERBOSE) SELECT * FROM t1;
                                QUERY PLAN
---------------------------------------------------------------------------
 Seq Scan on public.t1  (cost=0.00..737251.16 rows=51110016 width=4)
   (actual time=0.025..2823.214 rows=51110000 loops=1)
   Output: id
   Buffers: shared hit=769 read=225382
 Planning time: 0.027 ms
 Execution time: 4477.773 ms
(5 rows)
```

=> Pas de Parallélisation : le coût pour remonter les lignes au gather est trop important.

- Si l'on teste avec un filtre :

```
b1=# EXPLAIN (ANALYZE,BUFFERS,VERBOSE) SELECT * FROM t1 WHERE id = 4;
                                QUERY PLAN
---------------------------------------------------------------------------
 Gather  (cost=1000.00..386869.90 rows=1 width=4)
   (actual time=0.170..1096.948 rows=1 loops=1)
   Output: id
   Workers Planned: 4
   Workers Launched: 4
   Buffers: shared hit=1202 read=225153
   ->  Parallel Seq Scan on public.t1  (cost=0.00..385869.80 rows=1 width=4)
         (actual time=852.222..1071.570 rows=0 loops=5)
         Output: id
         Filter: (t1.id = 4)
         Rows Removed by Filter: 10222000
         Buffers: shared hit=998 read=225153
         Worker 0: actual time=1065.204..1065.204 rows=0 loops=1
           Buffers: shared hit=149 read=46784
         Worker 1: actual time=1065.214..1065.214 rows=0 loops=1
           Buffers: shared hit=157 read=35837
         Worker 2: actual time=1065.336..1065.336 rows=0 loops=1
           Buffers: shared hit=162 read=35762
         Worker 3: actual time=1065.331..1065.331 rows=0 loops=1
           Buffers: shared hit=361 read=62164
 Planning time: 0.040 ms
 Execution time: 1108.502 ms
```

Moins de lignes à retourner : Parallélisation (cf le paramètre `parallel_tuple_cost` qui

indique le coût par ligne)

Lire les compteurs loops et row.

Compter les blocs

- Test avec une jointure :

```
b1=# EXPLAIN (ANALYZE,BUFFERS,VERBOSE)
     SELECT * FROM t1 JOIN t2 ON t1.id=t2.id;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join  (cost=924932.72..4523070.84 rows=30000032 width=8)
   (actual time=5315.295..41891.417 rows=30000000 loops=1)
   Output: t1.id, t2.id
   Hash Cond: (t1.id = t2.id)
   Buffers: shared hit=1636 read=357262, temp read=238699 written=237677
   ->  Seq Scan on public.t1  (cost=0.00..737251.16 rows=51110016 width=4)
         (actual time=0.033..3210.287 rows=51110000 loops=1)
         Output: t1.id
         Buffers: shared hit=1158 read=224993
   ->  Hash  (cost=432744.32..432744.32 rows=30000032 width=4)
         (actual time=5311.760..5311.760 rows=30000000 loops=1)
         Output: t2.id
         Buckets: 131072  Batches: 512  Memory Usage: 3083kB
         Buffers: shared hit=475 read=132269, temp written=87470
         ->  Seq Scan on public.t2  (cost=0.00..432744.32 rows=30000032 width=4)
               (actual time=0.019..1849.495 rows=30000000 loops=1)
               Output: t2.id
               Buffers: shared hit=475 read=132269
 Planning time: 0.199 ms
 Execution time: 42907.328 ms
(16 rows)
```

Là encore trop de lignes retournées, et pas de Parallélisation

- Jointure avec présence d'un filtre :

```
b1=# EXPLAIN (ANALYZE,BUFFERS,VERBOSE)
     SELECT * FROM t1 JOIN t2 ON t1.id=t2.id WHERE t2.id = 100000;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop  (cost=2000.00..803393.83 rows=1 width=8)
   (actual time=1546.641..2173.989 rows=1 loops=1)
   Output: t1.id, t2.id
   Buffers: shared hit=2298 read=356813
   ->  Gather  (cost=1000.00..493349.10 rows=1 width=4)
         (actual time=1542.343..1542.345 rows=1 loops=1)
         Output: t1.id
```

```
         Workers Planned: 2
         Workers Launched: 2
         Buffers: shared hit=1294 read=224965
         ->  Parallel Seq Scan on public.t1  (cost=0.00..492349.00 rows=1 width=4)
               (actual time=1027.876..1539.231 rows=0 loops=3)
               Output: t1.id
               Filter: (t1.id = 100000)
               Rows Removed by Filter: 17036666
               Buffers: shared hit=1186 read=224965
               Worker 0: actual time=1536.289..1536.289 rows=0 loops=1
                 Buffers: shared hit=263 read=52931
               Worker 1: actual time=5.143..1539.207 rows=1 loops=1
                 Buffers: shared hit=628 read=119079
   ->  Gather  (cost=1000.00..310044.72 rows=1 width=4)
         (actual time=4.296..631.640 rows=1 loops=1)
         Output: t2.id
         Workers Planned: 2
         Workers Launched: 2
         Buffers: shared hit=1004 read=131848
         ->  Parallel Seq Scan on public.t2  (cost=0.00..309044.62 rows=1 width=4)
               (actual time=420.601..629.706 rows=0 loops=3)
               Output: t2.id
               Filter: (t2.id = 100000)
               Rows Removed by Filter: 10000000
               Buffers: shared hit=896 read=131848
               Worker 0: actual time=627.925..627.925 rows=0 loops=1
                 Buffers: shared hit=304 read=43543
               Worker 1: actual time=629.771..629.771 rows=0 loops=1
                 Buffers: shared hit=312 read=44036
 Planning time: 0.063 ms
 Execution time: 2175.036 ms
(33 rows)
```

=> On obtient une jointure entre deux parallel seq scan. Ce n'est pas une jointure parallélisée ! La parallélisation n'a joué que sur les parcours des deux tables, toutes les deux filtrées par la condition.

- Jointure avec un filtre retournant de nombreuses lignes.

```
EXPLAIN (ANALYZE,BUFFERS,VERBOSE)
SELECT * FROM t1 JOIN t3 ON t1.id=t3.id WHERE t1.id < 400000;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Gather  (cost=84467.00..581042.97 rows=28735 width=8)
   (actual time=734.866..1889.276 rows=399999 loops=1)
   Output: t1.id, t3.id
   Workers Planned: 2
```

```
   Workers Launched: 2
   Buffers: shared hit=37823 read=229759
   ->  Hash Join  (cost=83467.00..577169.47 rows=28735 width=8)
         (actual time=744.604..1804.314 rows=133333 loops=3)
         Output: t1.id, t3.id
         Hash Cond: (t1.id = t3.id)
         Buffers: shared hit=37703 read=229759
         Worker 0: actual time=747.513..1823.809 rows=174698 loops=1
           Buffers: shared hit=13202 read=75647
         Worker 1: actual time=751.694..1826.538 rows=166089 loops=1
           Buffers: shared hit=12929 read=75868
         ->  Parallel Seq Scan on public.t1  (cost=0.00..492349.00 rows=197398
               width=4) (actual time=0.041..1014.365 rows=133333 loops=3)
               Output: t1.id
               Filter: (t1.id < 400000)
               Rows Removed by Filter: 16903334
               Buffers: shared hit=673 read=225478
               Worker 0: actual time=0.047..1018.647 rows=174698 loops=1
                 Buffers: shared hit=217 read=74835
               Worker 1: actual time=0.046..1018.083 rows=166089 loops=1
                 Buffers: shared hit=217 read=74783
         ->  Hash  (cost=44717.00..44717.00 rows=3100000 width=4)
               (actual time=729.633..729.633 rows=3100000 loops=3)
               Output: t3.id
               Buckets: 4194304  Batches: 1  Memory Usage: 141753kB
               Buffers: shared hit=36870 read=4281
               Worker 0: actual time=733.596..733.596 rows=3100000 loops=1
                 Buffers: shared hit=12905 read=812
               Worker 1: actual time=737.104..737.104 rows=3100000 loops=1
                 Buffers: shared hit=12632 read=1085
               ->  Seq Scan on public.t3  (cost=0.00..44717.00 rows=3100000
                     width=4) (actual time=0.013..183.375 rows=3100000 loops=3)
                     Output: t3.id
                     Buffers: shared hit=36870 read=4281
                     Worker 0: actual time=0.014..185.594 rows=3100000 loops=1
                       Buffers: shared hit=12905 read=812
                     Worker 1: actual time=0.019..187.250 rows=3100000 loops=1
                       Buffers: shared hit=12632 read=1085
 Planning time: 0.156 ms
 Execution time: 1911.047 ms
(39 rows)
```

Là on a une jointure (*hash join*) parallélisé.

Note : *Rows Removed by Filter: 16903334* => à multiplier pour le nombre de loops => 50710002

Dans le hash de t3 : La table t3 est lue 3 fois!

`Buffers: shared hit=36870 read=4281 => 41151 = t3*3 = 13717 *3`

Chaque worker a lu la table une fois et le gather l'a lu une fois également.

Pour vérifier les nombres de lignes ou blocs lus , consulter les tables système `pg_stat_user_tables` ou `pg_stat_io_user_tables`, que vous pouvez réinitialiser entre deux essais avec la fonction `pg_stat_reset`.

- Les multiples parcours de t3 ci-dessus n'étaient possibles que parce que t3 tenait en mémoire. Exécuter la requête précédente en modifiant `work_mem` (mémoire de travail disponible au processus dédié, retournée avec `show work_mem`, modifiable avec `set work_mem = '12MB'`, `1 GB` ou `96MB`)

INDEX BLOOM

- Nous allons comparer un index btree classique et un bloom sur seulement deux attributs :

```
-- pour garantir une reproductibilité des ordres random() ci-dessous
SELECT setseed(1);

CREATE TABLE tab800 AS
  SELECT generate_series AS id,
    (random()*1000)::int4 AS bla,          -- integer
    md5(random()::text) AS bli,            -- varchar
    (random()*1000) AS blu                 -- double precision
  FROM generate_series (1, 10*1000*1000); --800 Mo et 10 millions de lignes

\d tab800

CREATE INDEX idx_bt_tab800 ON tab800 USING btree (bla, bli varchar_pattern_ops);

ANALYZE tab800;
```

La ligne qui va nous servir d'exemple à rechercher vaudra :

`bla=50` et `bli = 'e00b425b7ff60f42bd5fa61e043a46d6'`.

- Le plus efficace dans ce cas précis restera le btree (`Index Seq Scan`) :

```
EXPLAIN ANALYZE
SELECT * FROM tab800 WHERE bla = 50 AND bli = 'e00b425b7ff60f42bd5fa61e043a46d6';
```

- Si le critère ne porte que sur la deuxième colonne, l'index devient inutilisable et on se retrouve avec un Seq Scan sur la table même :

```
EXPLAIN (ANALYZE ,VERBOSE, BUFFERS)
SELECT * FROM tab800 WHERE bli = 'e00b425b7ff60f42bd5fa61e043a46d6';
```

- Ajouter l'index bloom :

```
CREATE EXTENSION bloom;

CREATE INDEX idx_bloom_tab800 ON tab800 USING bloom (bla, bli text_ops);

ANALYZE tab800 ;
```

- Comparer les tailles de la table et des index

```
SELECT
  relname,
  pg_size_pretty( pg_relation_size(relname::text) ) AS taille,
  relpages AS nb_blocs
FROM pg_class
WHERE relname LIKE '%tab800%';
```

- Mêmes recherches avec l'index bloom seul :

```
DROP INDEX idx_bt_tab800 ;

EXPLAIN (ANALYZE ,VERBOSE, BUFFERS)
SELECT *
FROM tab800
WHERE bla = 50 AND bli = 'e00b425b7ff60f42bd5fa61e043a46d6';
```

Le plan obtenu est le suivant. L'index bloom est moins efficace que le btree car il doit être parcouru entièrement. Noter les mentions sur le besoin de retourner dans la table filtrer les lignes : 7 sont alors filtrées alors que l'index les avait trouvées.

```
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on public.tab800  (cost=178436.00..178440.02 rows=1 width=49)
   (actual time=84.572..84.581 rows=1 loops=1)
   Output: id, bla, bli, blu
   Recheck Cond: ((tab800.bla = 50) AND
     (tab800.bli = 'e00b425b7ff60f42bd5fa61e043a46d6'::text))
   Rows Removed by Index Recheck: 7
   Heap Blocks: exact=8
   Buffers: shared hit=19616
   ->  Bitmap Index Scan on idx_tab800_bloom  (cost=0.00..178436.00 rows=1 width=0)
         (actual time=84.554..84.554 rows=8 loops=1)
         Index Cond: ((tab800.bla = 50) AND
           (tab800.bli = 'e00b425b7ff60f42bd5fa61e043a46d6'::text))
```

```
         Buffers: shared hit=19608
 Planning time: 0.291 ms
 Execution time: 84.678 ms
(11 lignes)
```

- Par contre si l'on n'a que la deuxième colonne, l'index bloom reste utilisable :

```
EXPLAIN (ANALYZE ,VERBOSE, BUFFERS)
SELECT * FROM tab800 WHERE bli = 'e00b425b7ff60f42bd5fa61e043a46d6';
```

```
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on public.tab800  (cost=153436.00..153440.01 rows=1 width=49)
   (actual time=68.305..83.765 rows=1 loops=1)
   Output: id, bla, bli, blu
   Recheck Cond: (tab800.bli = 'e00b425b7ff60f42bd5fa61e043a46d6'::text)
   Rows Removed by Index Recheck: 16528
   Heap Blocks: exact=15286
   Buffers: shared hit=34894
   ->  Bitmap Index Scan on idx_tab800_bloom  (cost=0.00..153436.00 rows=1 width=0)
         (actual time=62.100..62.100 rows=16529 loops=1)
         Index Cond: (tab800.bli = 'e00b425b7ff60f42bd5fa61e043a46d6'::text)
         Buffers: shared hit=19608
 Planning time: 0.130 ms
 Execution time: 83.868 ms
(11 lignes)
```

- Enfin, une des limitations de l'index est son incapacité à gérer les inégalités : le code suivant générera un `Seq Scan`.

```
EXPLAIN (ANALYZE ,VERBOSE, BUFFERS)
SELECT *
FROM tab800
WHERE bla BETWEEN 50 AND 51 AND bli LIKE 'e00b%';
```

FDW

- Dans une première instance : instance1 (celle par défaut sur votre poste)
  - Installer l'extension `postgres_fdw`
  - Créer un utilisateur bobby avec un mot de passe
- Dans une seconde instance : instance2 (créée sur le même poste sur un autre port, ou sur une autre machine)
  - Créer un utilisateur bobby avec un mot de passe
  - Créer 2 tables test1 et test2
  - Donner les droits à bobby sur ces deux tables
- Dans la première instance
  - Créer un serveur distant
  - Créer deux tables distantes atteignant les tables test1 et test2
  - Donner les droits à bobby sur ces deux tables
  - Afficher le plan d'exécution d'une requête réalisant un tri sur une table distante.
- Sur la première instance (sur laquelle on écrira les requêtes) :

Création de l'extension :

```
postgres=# CREATE EXTENSION postgres_fdw;
```

- Créer la seconde instance si une autre n'est pas disponible ailleurs. Ce qui suit suppose une instance 9.6 installée sur la même machine, port 5433.
- Sur les deux instances, création de l'utilisateur :

```
postgres=# CREATE USER bobby WITH PASSWORD 'bobby';
```

- Sur la seconde instance on crée la base "distante" et les tables :

```
postgres=# CREATE DATABASE distante;
\c distante
distante=# CREATE TABLE table1 (id serial, bla text);
distante=# CREATE TABLE table2 (id serial, bli text);
```

Accorder les droits à l'utilisateur bobby :

```
distante=# GRANT SELECT,INSERT ON table1 TO bobby;
distante=# GRANT SELECT,INSERT ON table2 TO bobby;
```

Toujours sur l'instance2, modifier pg_hba.conf pour que l'on puisse se connecter à l'utilisateur bobby depuis l'instance1:

```
host bobby           bobby           127.0.0.1/32         md5
```

Recharger la configuration de l'instance2 : `SELECT pg_reload_conf();`. Tester la connexion avec `psql -p 5433 -h localhost distante bobby`

- Sur l'instance1, ajouter la ligne suivante en tête de pg_hba.conf pour pouvoir vous connecter en tant que bobby sur votre poste :

```
local   postgres            bobby           127.0.0.1/32         md5
```

- Sur l'instance1, créer le serveur distant en précisant hôte, nom de base de données et port de la base distante :

```
postgres=# CREATE SERVER serveurdistant
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (host '127.0.0.1', dbname 'distante', port '5433');
```

Le nom de l'utilisateur est géré séparément, on crée une liaison entre le bobby de l'instance 1 et celui de l'instance distante :

```
postgres=# CREATE USER MAPPING FOR bobby
SERVER serveurdistant OPTIONS (user 'bobby' ,  password 'bobby');
```

Sur l'instance1, créer les deux tables distantes :

```
postgres=# CREATE FOREIGN TABLE table1_distante (id serial, bla text)
SERVER serveurdistant OPTIONS (table_name 'table1');

postgres=# CREATE FOREIGN TABLE table2_distante (id serial, bli text)
SERVER serveurdistant OPTIONS (table_name 'table2');
```

Attention ! * Spécifier le nom de la table accédée n'est pas optionnel. * Faire attention à bien déclarer la table distante avec la même définition.

On ne peut profiter d'une table distante pour renommer ses colonnes. Pour définir de nombreuses tables, voir `IMPORT FOREIGN SCHEMA`.

On peut consulter la liste des tables distantes avec `\det+`.

Accorder les droits à bobby sur les tables distantes :

```
postgres=# GRANT SELECT,INSERT ON table1_distante TO bobby;
postgres=# GRANT SELECT,INSERT ON table2_distante TO bobby;
```

- Enfin on teste depuis l'instance1 :

Se connecter avec l'utilisateur bobby et afficher le plan d'exécution d'une sélection triée par l'identifiant :

```
\c postgres bobby
postgres=> INSERT INTO table1_distante SELECT generate_series (1, 1000);
postgres=> INSERT INTO table2_distante SELECT generate_series (2, 1000);

postgres=> EXPLAIN (ANALYZE,VERBOSE)
SELECT * FROM table1_distante ORDER BY id;
                                QUERY PLAN
```

```
------------------------------------------------------------------------------
 Sort  (cost=222.03..225.44 rows=1365 width=36)
   (actual time=0.214..0.217 rows=20 loops=1)
   Output: id, bla
   Sort Key: table1_distante.id
   Sort Method: quicksort  Memory: 26kB
   ->  Foreign Scan on public.table1_distante (cost=100.00..150.95
         rows=1365 width=36) (actual time=0.199..0.202 rows=20 loops=1)
        Output: id, bla
        Remote SQL: SELECT id, bla FROM public.table1
 Planning time: 0.056 ms
 Execution time: 0.418 ms
(9 lignes)


postgres=> EXPLAIN (VERBOSE, ANALYZE)
SELECT * FROM table1_distante
JOIN table2_distante
  ON table1_distante.id=table2_distante.id
WHERE table2_distante.id = 1  ;

                                QUERY PLAN
------------------------------------------------------------------------------
 Foreign Scan  (cost=100.00..155.35 rows=49 width=72)
   (actual time=188.179..188.179 rows=0 loops=1)
   Output: table1_distante.id, table1_distante.bla, table2_distante.id,
     table2_distante.bli
   Relations: (public.table1_distante) INNER JOIN (public.table2_distante)
   Remote SQL: SELECT r1.id, r1.bla, r2.id, r2.bli FROM (public.table1 r1
     INNER JOIN public.table2 r2 ON (((r2.id = 1)) AND ((r1.id = 1))))
 Planning time: 0.371 ms
 Execution time: 188.982 ms
(6 lignes)
```

- Tenter les mêmes exemples avec des volumétries plus importantes, chercher la cause des débits assez bas.

SAUVEGARDES

- Faire une sauvegarde classique et vérifier le contenu des fichiers backup_label puis XXXX.XXXX.backup
- Faire deux sauvegardes concurrentes et vérifier ces mêmes fichiers

Préalable : dans `postgresql.conf`, l'archivage doit être actif:`wal_level = replica`, `archive_mode = on`, `archive_command = 'cp %p /ARCHIVAGE_LOGS/%f'` (ou juste

pour tester : `archive_command = '/bin/true'`).

- Petit rappel de la méthode non concurrente :

```
postgres=# select pg_start_backup('save95', true);
 pg_start_backup
-----------------
 0/7000028
(1 ligne)
```

La fonction `pg_start_backup()`, en autre, crée un fichier `backup_label` contenant les informations suivantes :

```
-bash-4.2$ cat backup_label

START WAL LOCATION: 0/7000028 (file 000000010000000000000007)
CHECKPOINT LOCATION: 0/7000060
BACKUP METHOD: pg_start_backup
BACKUP FROM: master
START TIME: 2016-08-12 17:50:47 CEST
LABEL: save95
```

On copie nos fichiers et on termine la sauvegarde :

```
postgres=# SELECT pg_stop_backup();
NOTICE:  pg_stop_backup terminé, tous les journaux de transactions requis
  ont été archivés
 pg_stop_backup
----------------
 0/7000130
```

La fonction `pg_stop_backup()` a ajouté des informations dans le fichier `backup_label`, qu'elle a renommé et déplacé dans le répertoire d'archivage.

```
-bash-4.2$ cat 000000010000000000000007.00000028.backup

START WAL LOCATION: 0/7000028 (file 000000010000000000000007)
STOP WAL LOCATION: 0/7000130 (file 000000010000000000000007)
CHECKPOINT LOCATION: 0/7000060
BACKUP METHOD: pg_start_backup
BACKUP FROM: master
START TIME: 2016-08-12 17:50:47 CEST
LABEL: save95
STOP TIME: 2016-08-12 17:51:36 CEST
```

- Maintenant intéressons nous à la réalisation de deux sauvegardes concurrentes, depuis deux sessions différentes, la première en étant connecté sur la base piloup, la suivante sur la base postgres :

```
piloup=# SELECT * FROM pg_start_backup('save96', true, false) ;
 pg_start_backup
-----------------
 1/17000028
(1 ligne)
```

On lance la copie des fichiers de notre instance :

```
cp -R /pg_data/9.6/instance1/* /pg_backup/snapshot/96/save96/
```

On lance la seconde sauvegarde :

```
postgres=# SELECT * FROM pg_start_backup('save96-bis', true, false);
 pg_start_backup
-----------------
 1/18000060
(1 ligne)
```

On lance la copie des fichiers :

```
cp -R /pg_data/9.6/instance1/* /pg_backup/snapshot/96/save96-bis/
```

On termine la première sauvegarde :

```
piloup=# SELECT * FROM pg_stop_backup(false);

NOTICE:  pg_stop_backup terminé, tous les journaux de transactions requis ont été
  archivés
    lsn     |                          labelfile                           | ...
------------+--------------------------------------------------------------+----
 1/1D0002F0 | START WAL LOCATION: 1/1C000060 (file 00000001000000010000001C)+|
            | CHECKPOINT LOCATION: 1/1C000098                              +|
            | BACKUP METHOD: streamed                                      +|
            | BACKUP FROM: master                                          +|
            | START TIME: 2016-08-12 18:00:38 CEST                         +|
            | LABEL: save96                                                +|
            |                                                               |
```

Le fichier indiquant la fin du backup apparaît parmi les fichiers de transaction dans pg_xlog :

```
-bash-4.2$ cat 00000001000000010000001C.00000060.backup
START WAL LOCATION: 1/1C000060 (file 00000001000000010000001C)
STOP WAL LOCATION: 1/1D0002F0 (file 00000001000000010000001D)
```

```
CHECKPOINT LOCATION: 1/1C000098
BACKUP METHOD: streamed
BACKUP FROM: master
START TIME: 2016-08-12 18:00:38 CEST
LABEL: save96
STOP TIME: 2016-08-12 18:13:12 CEST
```

On termine ensuite la seconde sauvegarde :

```
postgres=#  SELECT * FROM pg_stop_backup(false);
NOTICE:  pg_stop_backup terminé, tous les journaux de transactions requis ont été
archivés
    lsn     |                          labelfile                           | ...
------------+--------------------------------------------------------------+----
 1/1E000088 | START WAL LOCATION: 1/1D000028 (file 000000010000000010000001D)+|
            | CHECKPOINT LOCATION: 1/1D000060                              +|
            | BACKUP METHOD: streamed                                      +|
            | BACKUP FROM: master                                          +|
            | START TIME: 2016-08-12 18:01:57 CEST                         +|
            | LABEL: save96-bis                                            +|
            |                                                               |
```

Un autre fichier apparaît dans `pg_xlog` :

```
$ cat 000000010000000010000001D.00000028.backup
START WAL LOCATION: 1/1D000028 (file 000000010000000010000001D)
STOP WAL LOCATION: 1/1E000088 (file 000000010000000010000001E)
CHECKPOINT LOCATION: 1/1D000060
BACKUP METHOD: streamed
BACKUP FROM: master
START TIME: 2016-08-12 18:01:57 CEST
LABEL: save96-bis
STOP TIME: 2016-08-12 18:14:58 CEST
```

VISIBILITY MAP

- On crée une nouvelle table avec 451 lignes :

```
CREATE TABLE test_visibility AS SELECT generate_series(0,450) AS id;
```

On regarde dans quel état est la visibility map :

```
CREATE EXTENSION pg_visibility;

SELECT * FROM pg_visibility('test_visibility');
 blkno | all_visible | all_frozen | pd_all_visible
```

```
-------+-------------+------------+----------------
     0 | f           | f          | f
     1 | f           | f          | f
```

Les deux blocs que composent la table test_visibility sont à false, c'est normal puisque l'opération de vacuum n'a jamais été exécutée sur cette table.

On lance donc une opération de vacuum :

```
VACUUM VERBOSE test_visibility ;

INFO:  exécution du VACUUM sur « public.test_visibility »
INFO:  « test_visibility » : 0 versions de ligne supprimables, 451 non
supprimables parmi 2 pages sur 2
DÉTAIL : 0 versions de lignes mortes ne peuvent pas encore être supprimées.
Il y avait 0 pointeur d'éléments inutilisés.
Ignore 0 page à cause des verrous de blocs.
0 page est entièrement vide.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
```

Vacuum voit bien nos 451 lignes, et met donc la visibility_map a jour. Lorsqu'on la consulte, on voit bien que toutes les lignes sont visibles

```
SELECT * FROM pg_visibility('test_visibility');
 blkno | all_visible | all_frozen | pd_all_visible
-------+-------------+------------+----------------
     0 | t           | f          | t
     1 | t           | f          | t
(2 lignes)
```

On va maintenant réaliser un update sur les 50 dernières lignes. En pratique, ces 50 lignes vont être taguées comme obsolètes et 50 nouvelles lignes vont être créées à la suite.

```
UPDATE test_visibility SET id = 3 where id > 400;
```

Lorsqu'on regarde de nouveau la visibility map, on constate que le premier bloc est resté inchangé, qu'un nouveau bloc a été créé, et que lui et un nouveau bloc sont passés à false.

```
SELECT * FROM pg_visibility('test_visibility');

 blkno | all_visible | all_frozen | pd_all_visible
-------+-------------+------------+----------------
     0 | t           | f          | t
     1 | f           | f          | f
     2 | f           | f          | f
```

On exécute de nouveau un vacuum verbose :

```
VACUUM VERBOSE test_visibility ;

INFO:  exécution du VACUUM sur « public.test_visibility »
INFO:  « test_visibility » : 50 versions de ligne supprimées parmi 1 pages
INFO:  « test_visibility » : 50 versions de ligne supprimables, 451 non
supprimables parmi 3 pages sur 3
DÉTAIL : 0 versions de lignes mortes ne peuvent pas encore être supprimées.
Il y avait 0 pointeur d'éléments inutilisés.
Ignore 0 page à cause des verrous de blocs.
0 page est entièrement vide.
CPU 0.00s/0.00u sec elapsed 0.00 sec.

SELECT * FROM pg_visibility('test_visibility');
 blkno | all_visible | all_frozen | pd_all_visible
-------+-------------+------------+----------------
     0 | t           | f          | t
     1 | t           | f          | t
     2 | t           | f          | t
```

Toutes les lignes contenues dans les trois blocs sont visibles. A priori, il reste de quoi insérer 50 lignes dans le deuxième bloc.

Si on relance une opération de vacuum alors que toutes les lignes sont visibles :

```
vacuum VERBOSE test_visibility ;
INFO:  exécution du VACUUM sur « public.test_visibility »
INFO:  « test_visibility » : 0 versions de ligne supprimables, 451 non
supprimables parmi 3 pages sur 3
DÉTAIL : 0 versions de lignes mortes ne peuvent pas encore être supprimées.
Il y avait 49 pointeurs d'éléments inutilisés.
Ignore 0 page à cause des verrous de blocs.
0 page est entièrement vide.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
```

« Il y avait 49 pointeurs d'éléments inutilisés » : effectivement, il reste bien de l'espace dans le deuxième bloc.

```
SELECT count (*) FROM test_visibility ;
 count
-------
   451
```

si on compte 226 lignes pour 1 bloc, on devrait pouvoir ajouter 220 lignes et ne pas excéder 3 blocs (672 lignes)

```
INSERT INTO test_visibility (SELECT generate_series(1,220));

SELECT * FROM pg_visibility('test_visibility');
```

```
blkno | all_visible | all_frozen | pd_all_visible
-------+-------------+------------+----------------
     0 | t           | f          | t
     1 | f           | f          | f
     2 | f           | f          | f
```

Les deux derniers blocs ont été modifiés, ce qui signifie que les espaces du deuxième bloc ont été réutilisés et que le reste des lignes a été ajouté à la suite.

```
VACUUM VERBOSE test_visibility ;

INFO:  exécution du VACUUM sur « public.test_visibility »
INFO:  « test_visibility » : 0 versions de ligne supprimables, 671 non
supprimables parmi 3 pages sur 3
DÉTAIL : 0 versions de lignes mortes ne peuvent pas encore être supprimées.
Il y avait 0 pointeur d'éléments inutilisés.
Ignore 0 page à cause des verrous de blocs.
0 page est entièrement vide.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
```

On ajoute 10 lignes :

```
INSERT INTO test_visibility (SELECT generate_series(1,10));

SELECT count(*) FROM test_visibility;
 count
-------
   681
(1 ligne)
```

On a bien créé un nouveau bloc et modifié le dernier, les deux premiers blocs sont eux restés identiques :

```
SELECT * FROM pg_visibility('test_visibility');

 blkno | all_visible | all_frozen | pd_all_visible
-------+-------------+------------+----------------
     0 | t           | f          | t
     1 | t           | f          | t
     2 | f           | f          | f
     3 | f           | f          | f
```

```
VACUUM VERBOSE test_visibility ;
INFO:  exécution du VACUUM sur « public.test_visibility »
INFO:  « test_visibility » : 0 versions de ligne supprimables, 681 non
supprimables parmi 4 pages sur 4
DÉTAIL : 0 versions de lignes mortes ne peuvent pas encore être supprimées.
```

```
Il y avait 0 pointeur d'éléments inutilisés.
Ignore 0 page à cause des verrous de blocs.
0 page est entièrement vide.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
```

« Il y avait 0 pointeur d'éléments inutilisés » nous indique bien que les espaces inutilisés ont été remplis.

- Regardons maintenant comment se comporte le VACUUM FREEZE :

On lance un VACUUM FREEZE sur notre table test_visibility :

```
SELECT * FROM pg_visibility('test_visibility');

 blkno | all_visible | all_frozen | pd_all_visible
-------+-------------+------------+----------------
     0 | t           | t          | t
     1 | t           | t          | t
     2 | t           | t          | t
     3 | t           | t          | t
```

En consultant la vue on constate que toutes les lignes, dans tous les blocs sont passées en gelées.

Effectivement, tous les `xmin` des lignes ont été passés à la même valeur :

```
SELECT xmin, xmax, * FROM test_visibility LIMIT 10;
 xmin | xmax | id
------+------+----
 1882 |    0 |  0
 1882 |    0 |  1
 1882 |    0 |  2
 1882 |    0 |  3
 1882 |    0 |  4
 1882 |    0 |  5
 1882 |    0 |  6
 1882 |    0 |  7
 1882 |    0 |  8
 1882 |    0 |  9
```

si maintenant on fait une mise à jour de presque toute la table :

```
UPDATE test_visibility SET id = 3 WHERE id < 400;
```

Tous les `xmin` des lignes mises à jour sont modifiés :

```
SELECT xmin, xmax, * FROM test_visibility LIMIT 10;
 xmin | xmax | id
------+------+-----
 1882 |    0 | 400
 1890 |    0 |   3
 1890 |    0 |   3
 1890 |    0 |   3
 1890 |    0 |   3
 1890 |    0 |   3
 1890 |    0 |   3
 1890 |    0 |   3
 1890 |    0 |   3
 1890 |    0 |   3
```

Vue l'ampleur des changements, un autovacuum a été déclenché au moment de cette opération :

```
SELECT * FROM pg_stat_user_tables WHERE relname = 'test_visibility';

relname             | test_visibility
last_autovacuum     | 2016-08-12 10:57:01.125126+02
vacuum_count        | 4
autovacuum_count    | 1
```

Ce qui nous donne le résultat suivant lorsque l'on vérifie pg_visibility :

```
SELECT * FROM pg_visibility('test_visibility');

 blkno | all_visible | all_frozen | pd_all_visible
-------+-------------+------------+----------------
     0 | t           | t          | t
     1 | t           | t          | t
     2 | t           | t          | t
     3 | t           | f          | t
     4 | t           | f          | t
     5 | t           | f          | t
     6 | t           | f          | t
```

Les lignes modifiées des blocs 0,1,2 et 3 peuvent désormais être réécrites, et qu'elles sont restées gelées.

Si maintenant on réalise une insertion d'une dizaine de lignes :

```
INSERT INTO test_visibility (SELECT generate_series(1,10));

SELECT * FROM pg_visibility('test_visibility');
```

```
 blkno | all_visible | all_frozen | pd_all_visible
-------+-------------+------------+----------------
     0 | f           | f          | f
     1 | t           | t          | t
     2 | t           | t          | t
     3 | t           | f          | t
     4 | t           | f          | t
     5 | t           | f          | t
     6 | t           | f          | t
```

Ces lignes ont été écrites dans le bloc 0 et entraînent le retour à false des attributs all_visible et all_frozen.

FONCTIONS SQL

Afficher le sinus pour 30° puis pour 0,523599 rd

```
SELECT sind(30);
 sind
------
  0.5

SELECT sin(0.523599);
        sin
-------------------
 0.500000194337561
```

RÉPLICATION SYNCHRONE

- Installer deux nouvelles instances en 9.6.
- Mettre en place la réplication.
- Mettre en place la réplication synchrone.
- arrêter le slave et lancer une commande sur le maître.
- redémarrer le slave.
- Installation des 2 instances supplémentaires avec `initdb` (RedHat, CentOS…) ou `pg_createcluster` (Debian), port d'écoute `5433` pour l'instance 2 et `5434` pour l'instance 3.
  - Création de l'utilisateur de réplication sur l'instance primaire :

```
createuser -p 5432 replication --replication -P
```

- Modification de `pg_hba.conf` pour lui donner accès :

```
local   all             all                                     trust
host    all             all             127.0.0.1/32            trust

local   replication     replication                             trust
host    replication     replication        127.0.0.1/32         trust
```

- Modification de postgresql.conf :

Pour la réplication, dans les trois `postgresql.conf` :

```
wal_level = logical             # minimal, replica, or logical
max_wal_senders = 5             # max number of walsender processes
max_replication_slots = 5       # max number of replication slots
hot_standby = on                # "on" allows queries during recovery
```

- Création des slots de réplication sur le maître :

```
SELECT pg_create_physical_replication_slot('standby2');
SELECT pg_create_physical_replication_slot('standby3');
```

- Pour la réplication synchrone dans `postgresql.conf` :

```
synchronous_standby_names = '2 (standby2, standby3)'
```

- Restauration de l'instance primaire sur les deux autres serveurs : d'abord on vide les répertoires cibles, puis on lance la copie, et on profite aussi

```
pg_basebackup -D /pg_data/9.6/instance2/ -p 5433 -U replication \
--write-recovery-conf --slot=standby2 --xlog-method=stream --progress --verbose
pg_basebackup -D /pg_data/9.6/instance3/ -p 5434 -U replication \
--write-recovery-conf --slot=standby3 --xlog-method=stream --progress --verbose
```

- Modification des fichiers recovery.conf : il faut notamment préciser `application_name` pour que le serveur primaire reconnaisse les secondaires.

Instance 2 :

```
standby_mode=on
primary_conninfo = 'host=127.0.0.1 port=5432 user=replication
  password=repli application_name=standby2'
primary_slot_name='standby2'
```

Instance 3 :

```
standby_mode = on
primary_conninfo = 'host=127.0.0.1 port=5432 user=replication
  password=repli application_name=standby3'
primary_slot_name='standby3'
```

- Les instances doivent démarrer et répercuter les modifications aux données de l'instance maître.
- Si l'une des deux instances secondaires est arrêtée, aucune modification de données n'est possible sur l'instance maître, sauf à modifier le paramètre ainsi : `synchronous_standby_names = '1 (standby2, standby3)'`
- Testez les différentes de performance entre les différentes valeurs de `synchronous_commit`, par exemple avec pgbench :
  - Initialisation dans une base dédiée nommée pgbench :
    `pgbench -i -s 100 --foreign-keys -p 5432 pgbench`
  - Tests (à répéter plusieurs fois à cause du cache) avec:
    `pgbench --client=3 --jobs=4 --transactions=1000 -p 5432 pgbench`