Table des matières

Nouvelle gestion des journaux applicatifs sous PostgreSQL 8.3

Cet article, écrit par Guillaume Lelarge, a été publié dans le magazine GNU/Linux Magazine France, numéro 105 (Mai 2008). Il est disponible maintenant sous licence Creative Commons. Un supplément n'a pas eu le temps d'arriver avant impression. Il est disponible sur dalibo.org.

PostgreSQL comporte un grand nombre de paramètres permettant une configuration avancée de la journalisation applicative. Cela permet d'avoir des informations en plus ou moins grande quantité. Plus vous disposez d'informations, meilleure est votre idée de la situation, que vous ayez un problème à régler, ou que vous ayez tout simplement à surveiller le travail du serveur. La version 8.0 a amélioré l'administration des journaux applicatifs en ajoutant des options de rotation des fichiers lorsque les fichiers de la journalisation applicative sont gérés par PostgreSQL.

Auparavant, si un administrateur de base de données utilisait la sortie des erreurs pour la journalisation, il devait installer et configurer un outil supplémentaire comme le programme rotatelogs d'Apache. Ce n'est plus le cas. De plus, la configuration se faisant au sein de PostgreSQL, il est possible de modifier la configuration de la journalisation sans avoir à redémarrer PostgreSQL. Enfin, le serveur de bases de données peut aussi surveiller le processus de rotation des journaux et le relancer en cas de problème. Malheureusement, depuis cette version, peu de nouveautés sont apparues dans ce domaine. Il faut ajouter que l'utilisation de ces fichiers est restée assez complexe. Il est nécessaire d'ouvrir les fichiers dans un éditeur de texte, et de faire des recherches plus ou moins avancées suivant les capacités de l'éditeur : difficile de faire un filtre sur le contenu (en dehors de la commande Unix grep), difficile de faire un tri, etc. La version 8.3 améliore la situation en proposant de stocker les journaux applicatifs au format CSV. Ce format est souvent utilisé par les tableurs pour échanger des données de type tableau. Il est aussi utilisable par PostgreSQL pour importer et exporter le contenu d'une table. Cet article a pour but de faire un point complet sur la journalisation applicative dans PostgreSQL et de mettre en avant les nouveautés de la version 8.3.

Configuration du moyen de stockage

Après avoir installé un serveur PostgreSQL, il est important, entre autres configurations, de bien réfléchir à la façon dont on va stocker les journaux applicatifs. Il existe trois grands choix :

- soit l'administrateur décide que PostgreSQL se charge lui-même de la gestion des fichiers constituant les journaux ;
- soit l'administrateur préfère déléguer cette tâche à un autre outil, plus spécialisé, comme syslog (ce qui est plus dans la philosophie Unix, et qui sera certainement privilégié au cas où cet administrateur gère plusieurs serveurs, physiques et/ou logiciels, car toutes les traces pourront être regroupées et gérées au sein d'un même logiciel) ;
- soit l'administrateur veut gérer lui-même les fichiers.

Commençons par le premier cas. Le paramètre log_destination indique la destination des traces. Il faut le configurer à stderr pour que les traces soient envoyées sur la sortie standard des erreurs. L'activation de logging_collector (redirect_stderr dans les versions antérieures à la 8.3) indique à PostgreSQL qu'il va devoir gérer le stockage des fichiers et leur rotation. Pour cela, l'administrateur doit indiquer le répertoire de stockage grâce au paramètre log_directory, ainsi que le nom du fichier avec log_filename. Ce dernier peut utiliser tous les caractères joker de la fonction C strftime() pour que le nom du fichier dépende du contexte comme la date et l'heure de création du fichier. Quant à la rotation, elle dépend de deux paramètres : l'âge du fichier (avec log_rotation_age) et sa taille (avec log_rotation_size). Ainsi, un fichier peut être créé tous les jours en s'assurant qu'il ne dépasse pas la taille de 100 Mo. Cet exemple correspond à la configuration suivante :

```
log_destination = 'stderr'
logging_collector = on
log_directory = 'pg_log'
log_filename = 'postgresql-%Y-%m-%d.log'
log_rotation_age = 1d
log_rotation_size = 100MB
```

Si, suite à une rotation, le nouveau nom du fichier correspond à un fichier déjà existant, PostgreSQL pourra gérer la situation de deux façons : soit en écrasant le fichier existant, soit en ajoutant les messages à la suite du fichier existant. Cela se configure avec le paramètre log_truncate_on_rotation. Toujours dans ce cadre-là, la version 8.3 ajoute une possibilité de destination qu'on abordera plus en détail plus tard, à savoir csvlog. Là encore, PostgreSQL se charge de stocker les messages dans des fichiers qui, cette fois, seront au format CSV.

Prenons maintenant le deuxième cas, à savoir l'outil externe. Si vous êtes sous Unix, vous allez utiliser un outil comme syslog. Tout à fait logiquement, vous allez paramétrer log_destination en utilisant la valeur syslog. Les paramètres syslog_facility et syslog_ident permettent de préciser respectivement le type de programme et son identification. Voici un exemple de cette configuration :

```
log_destination = 'syslog'
logging_collector = off
syslog_facility = 'local0'
syslog_ident = 'postgres'
```

Dans ce cas, la rotation des journaux applicatifs ne dépend plus des paramètres PostgreSQL mais d'un outil externe comme logrotate.

Note : Au cas où vous seriez sous Windows, la valeur syslog n'est pas disponible. Par contre, vous pouvez envoyer les traces au journal des événements de Windows en initialisant log_destination à eventlog.

Le dernier cas, où l'administrateur veut gérer cette partie lui-même, la configuration demande de configurer log_destination à stderr sans activer logging_collector. Ainsi les messages seront envoyés sur la sortie standard des erreurs. À la charge de l'administrateur d'envoyer les messages sur un fichier, un script ou tout autre chose.

Configuration de la quantité d'informations à tracer

Quand les développeurs choisissent de publier une information via les journaux applicatifs, ils indiquent le message mais aussi son niveau d'importance. Un administrateur peut choisir de ne stocker les messages que s'ils sont d'un certain niveau. Pour cela, il faut utiliser le paramètre log_min_messages. Ce dernier peut prendre une valeur parmi celles-ci :

- DEBUG5, DEBUG4, DEBUG3, DEBUG2, DEBUG1, messages de débogage utiles principalement aux développeurs (mais aussi aux utilisateurs qui veulent en apprendre plus sur le fonctionnement du système… par exemple DEBUG2 est le niveau permettant, entre autres, de connaître les tables qui seront traitées par un VACUUM et/ou un ANALYZE suite à l'activation du démon autovacuum) ;
- INFO, informations mineures ;
- NOTICE ;
- WARNING, messages d'avertissements (donc sans conséquence dans l'immédiat) ;
- ERROR, principalement des requêtes en erreur ;
- LOG, informations importantes ;
- FATAL, le processus ne peut plus continuer (généralement lorsqu'une connexion échoue, par exemple un mot de passe erroné) ;
- PANIC, le système n'est plus disponible car une erreur empêche son utilisation sans intervention de l'administrateur.

L'administrateur peut demander à ce que chaque message enregistré soit peu détaillé, normalement détaillé ou beaucoup détaillé. Le paramètre log_error_verbosity aura les valeurs respectives suivantes : terse, default, verbose.

Si une instruction SQL a causé l'envoi d'un message (souvent une erreur de syntaxe), il est souvent intéressant de savoir laquelle. Le paramètre

log_min_error_statement indique le niveau minimum qui déclenchera l'enregistrement de la requête. Les niveaux sont un peu différents de ceux de log_min_messages : DEBUG5, DEBUG4, DEBUG3, DEBUG2, DEBUG1, INFO, NOTICE, WARNING, ERROR, LOG, FATAL et PANIC. Dans le même style, il est intéressant de connaître les requêtes SQL longues à exécuter. Le paramètre log_min_duration_statement précise la durée minimale d'exécution (en milli-secondes) avant de tracer la requête et sa durée d'exécution.

Note : Récupérer les requêtes SQL en erreur a deux intérêts. Lors de la phase de développement ou de tests d'une application, il est plus facile de prévenir les développeurs d'un soucis, tout en leur fournissant la requête erronée. Après une mise en production, le problème de l'injection SQL peut survenir. Il vous sera facile de détecter une tentative de ce genre si vous tracez les requêtes en erreur.

silent_mode est un paramètre un peu à part. Une fois activé, le serveur est silencieux sur la sortie des erreurs. Il n'est donc intéressant de l'activer que quand syslog ou eventlog sont utilisés.

Informations spécifiques à tracer

En dehors des messages d'utilisation standard, il est possible de demander des informations supplémentaires pour surveiller l'activité de certains processus comme bgwriter ou autovacuum. L'administrateur dispose donc des paramètres suivants :

Paramètre	Version	Commentaires
log_checkpoints	8.3	Trace l'activité des checkpoints
log_connections	< 8.0	Trace chaque connexion (réussie ou échouée)
log_disconnections	8.0	Trace chaque déconnexion (et précise la durée de la connexion)
log_statement	< 8.0	Trace chaque instruction exécutée qui satisfait ce niveau : none pour aucune, ddl pour les modifications de structure de la base, mod pour les modifications (structure et données), all pour toutes
log_duration	< 8.0	Trace la durée de chaque instruction qui satisfait le niveau log_statement
log_hostname	< 8.0	Remplace l'adresse IP par le nom d'hôte dans la trace (attention au problème de lenteur dû à la requête DNS)
log_lock_waits	8.3	Trace toute session qui attend plus de deadlock_timeout millisecondes pour obtenir un verrou
log_temp_files	8.3	Trace la suppression de fichiers temporaires de taille supérieure à ce nombre

L'un des principaux apports de la version 8.3 pour la journalisation concerne le suivi d'activités spécifiques et principalement liées aux performances : les checkpoints, les verrous, les fichiers temporaires.

Ajouter un préfixe aux traces

Chaque message envoyé ne contient que le message : pas d'horodatage, pas d'informations sur la session concernée. syslog ajoute automatiquement son propre préfixe contenant date, heure, PID, nom d'hôte, etc. Si vous utilisez la sortie stderr, il vous est conseillé de modifier le paramètre log_line_prefix pour au moins ajouter la date et l'heure, informations capitales pour trier les messages. Cela étant dit, le préfixe étant complètement personnalisable, un ensemble de caractères joker permettent à l'administrateur d'indiquer bien plus d'informations comme, par exemple, l'utilisateur connecté, la base de connexion, l'adresse IP et le port du client, etc. Attention, si vous utilisez la sortie csvlog, cette dernière est déjà strictement définie. Personnaliser log_line_prefix aura plus d'inconvénients que d'avantages.

Astuce ! n'oubliez pas d'ajouter un espace à la fin du préfixe pour bien séparer préfixe et message, PostgreSQL ne le fait pas pour vous.

Langue des traces

PostgreSQL est capable de s'exprimer en français. Les traces peuvent donc être enregistrées en français. C'est intéressant quand on débute car il est plus facile d'appréhender ce SGBD avec sa langue native. Malheureusement, cela complique aussi les choses pour plusieurs raisons. La première est qu'une recherche sur Google a plus de chances de rapporter des informations si le message recherché est en anglais. La seconde, à peu près du même ordre, est que vous devrez donner le message en anglais si vous cherchez de l'aide sur une des listes de discussion du projet PostgreSQL. Les utilisateurs avancés et les développeurs ne feront pas l'effort de traduire les messages laissés en français. Enfin, la dernière raison, la plus gênante, les anciennes traductions laissaient à désirer : parfois erronées, parfois vagues. La version 8.3.0 est la première à disposer de traductions entièrement revues. Les prochaines versions mineures des branches 8.2 à 7.4 auront aussi ces nouvelles traductions.

PostgreSQL utilise le paramètre lc_messages pour savoir dans quelle langue les messages du serveur doivent être traduits. Pour basculer à la langue originale, l'anglais, il suffit de configurer cette variable à 'C'. Voici un exemple de configuration en direct :

```
tests=# SHOW lc_messages;
 lc_messages
-------------
 fr_FR.UTF-8
(1 ligne)

tests=# ALTER;
ERREUR:  erreur de syntaxe sur ou près de « ; »
LIGNE 1 : ALTER;
          ^
tests=# SET lc_messages TO 'C';
SET
tests=# ALTER;
ERROR:  syntax error at or near ";"
LIGNE 1 : ALTER;
          ^
```

La grosse nouveauté de la version 8.3 : la journalisation au format CSV

PostgreSQL nous permet de sélectionner l'outil qui va gérer les fichiers de la journalisation, il nous permet aussi de préciser le niveau des informations désirées ainsi que d'avoir des informations spécifiques sur certains événements. Le dernier point restant est une utilisation simple des journaux, permettant un tri et un filtre rapide. Une solution est apportée avec la version 8.3.

Un format commun pour l'échange d'informations sous forme de tableau est le CSV. Ce format sépare chaque colonne par une virgule. L'importation dans un tableur, quel qu'il soit, en est simplifié. PostgreSQL propose donc ce format pour les journaux applicatifs. Pour l'activer, il suffit d'initialiser log_destination à csvlog. PostgreSQL fonctionne de la même façon que si log_destination avait été configuré à stderr, c'est-à-dire qu'il stocke les fichiers dans le répertoire pointé par log_directory, et qu'il les nomme suivant le modèle indiqué par log_filename (à l'exception de l'extension qui sera remplacé par .csv s'il y avait une extension, dans le cas contraire .csv est tout simplement ajouté).

Voici le contenu du répertoire pointé par log_directory :

```
guillaume@laptop:/opt/postgresql-8.3/data/pg_log$ ll
total 44
-rw------- 1 postgres postgres     0 2008-02-27 08:49 postgresql-2008-02-27.log
-rw------- 1 postgres postgres 13209 2008-02-27 09:02 postgresql-2008-02-27.csv
-rw------- 1 postgres postgres     0 2008-02-28 00:00 postgresql-2008-02-28.log
-rw------- 1 postgres postgres  9499 2008-02-28 00:00 postgresql-2008-02-28.csv
```

Dans cet exemple, deux journaux sont créés tous les jours, un avec une extension .log, l'autre avec l'extension .csv. En fait, PostgreSQL crée bien un journal .log mais il y a peu de chances d'y trouver des informations (d'où la taille nulle). PostgreSQL le crée au tout début pour récupérer des messages qui ne seraient pas correctement envoyés via la fonction elog() sur la sortie standard des erreurs. L'exemple le plus facile à comprendre concerne le script d'archivage des journaux de transaction. Ce dernier peut envoyer des messages sur la sortie des erreurs. Comme ils ne peuvent pas être envoyées à la fonction de traitement interne des messages, ils sont envoyés directement sur la sortie standard des erreurs, et finissent donc dans le fichier .log du jour.

Ce qui suit montre le contenu d'un journal applicatif au format CSV.

```
utilisateur=guillaume base=postgres hôte=[local]",,,,,,,,
2008-02-27 10:12:45.175 CET,,,6731,"",47c5298d.1a4b,1,"/opt/postgresql-8.3/bin/postgres",2008-02-27 10:12:45 CET,,0,LOG,00000,"connexion reçue : hôte=[local]",,,,,,,,
2008-02-27 10:12:45.176 CET,"guillaume","tests",6731,"[local]",47c5298d.1a4b,2,"authentication",2008-02-27 10:12:45 CET,,0,LOG,00000,"connexion autorisée : utilisateur=guillaume, base de données=tests",,,,,,,,
2008-02-27 10:12:52.599 CET,"guillaume","tests",6731,"[local]",47c5298d.1a4b,3,"idle",2008-02-27 10:12:45 CET,1/4,0,LOG,00000,"instruction : show shared_buffers;",,,,,,,,
2008-02-27 10:13:04.401 CET,"guillaume","tests",6731,"[local]",47c5298d.1a4b,4,"idle",2008-02-27 10:12:45 CET,1/5,0,LOG,00000,"instruction : show lc_messages;",,,,,,,,
2008-02-27 10:13:15.511 CET,"guillaume","tests",6731,"[local]",47c5298d.1a4b,5,"idle",2008-02-27 10:12:45 CET,1/6,0,LOG,00000,"instruction : set lc_messages to 'C';",,,,,,,,
2008-02-27 10:13:16.764 CET,"guillaume","tests",6731,"[local]",47c5298d.1a4b,6,"idle",2008-02-27 10:12:45 CET,1/7,0,LOG,00000,"statement: show lc_messages;",,,,,,,,
2008-02-27 10:13:26.845 CET,"guillaume","tests",6731,"[local]",47c5298d.1a4b,7,"idle",2008-02-27 10:12:45 CET,,0,LOG,00000,"disconnection: session time: 0:00:41.669 user=guillaume database=tests host=[local]",,,,,,,,
```

Le constat est simple : le tableau contient déjà de nombreuses colonnes, et est peu lisible ainsi, encore moins que le journal habituel. Il est donc nécessaire de passer par un outil pour mieux le déchiffrer. Le détail des colonnes est disponible dans le tableau ci-dessous.

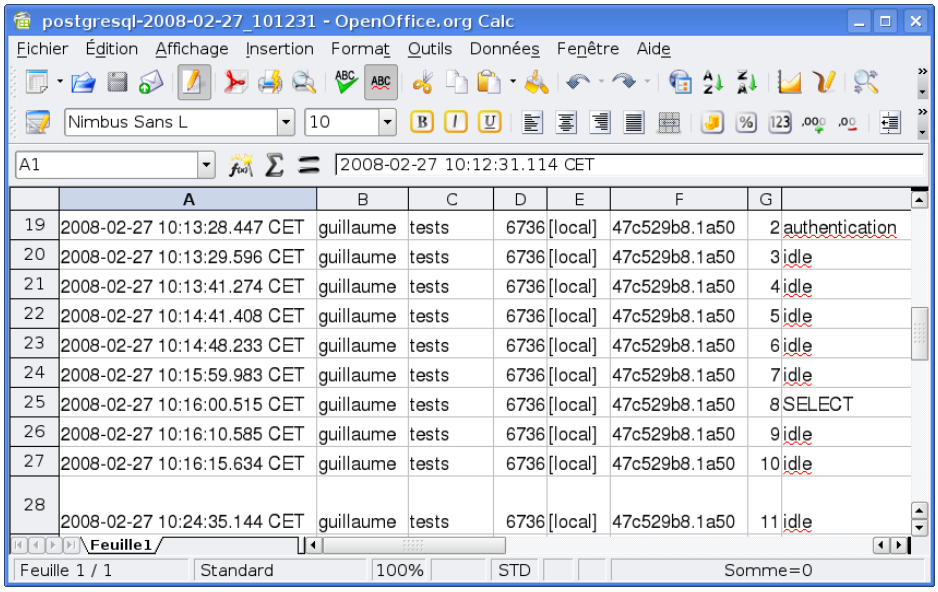
Colonne	Type	
1	timestamp(3)	Horodatage de la trace
2	text	Nom de l'utilisateur

3	text	Nom de la base
4	integer	PID
5	text	Adresse IP (ou alias) de connexion
6	text	Identifiant de session
7	bigint	Numéro de ligne de la session (dans le cas où le message est sur plusieurs lignes, par exemple une requête)
8	text	Tag de la commande
9	timestamp with timezone	Horodatage du début de la session
10	text	Identifiant de la transaction virtuelle
11	bigint	Identifiant de transaction
12	text	Sévérité (niveau) de l'erreur
13	text	Code d'état SQL
14	text	Message
15	text	Détail
16	text	Conseil
17	text	Requête interne
18	integer	Position sur la requête interne
19	text	Contexte
20	text	Requête
21	integer	Position sur la requête
22	text	Emplacement

Intégration du journal dans un tableur

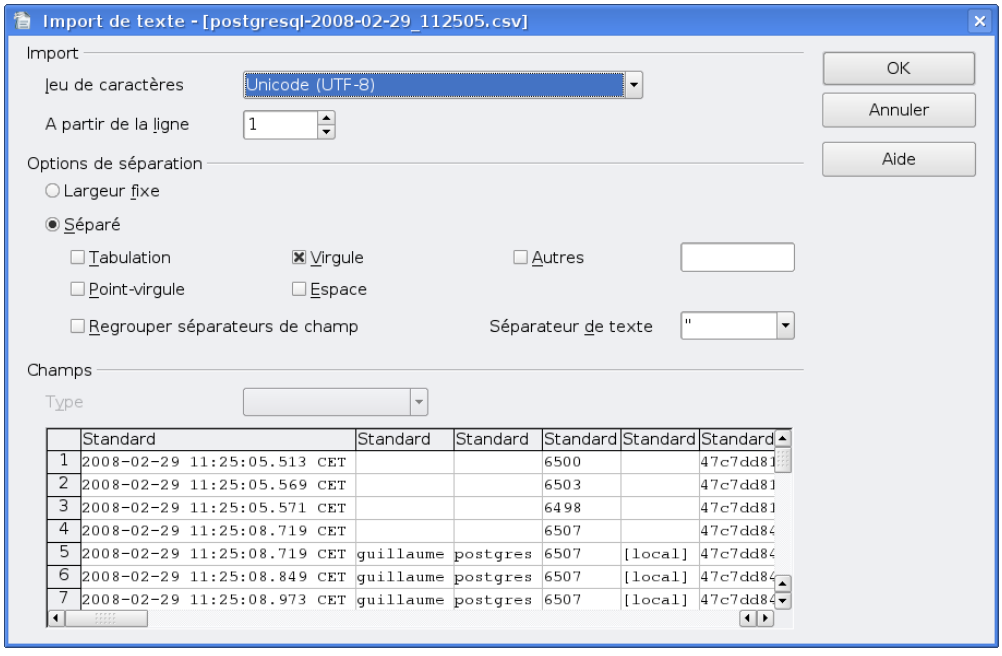
Le format CSV a été conçu notamment pour permettre des échanges entre tableurs. Prenons l'exemple de Calc, le tableur d'OpenOffice.org. Voici les étapes pour récupérer le contenu du journal dans Calc :

- démarrer Calc ;
- aller dans Fichier/Ouvrir ;
- sélectionner le journal ;
- le dialogue qui s'ouvre doit avoir par défaut les bons paramètres (jeu de caractères Unicode, lire depuis la ligne 1, et la virgule comme champ séparateur… voir la figure 1) ;
- cliquer sur OK.



	A	B	C	D	E	F	G	
19	2008-02-27 10:13:28.447 CET	guillaume	tests	6736	[local]	47c529b8.1a50	2	authentication
20	2008-02-27 10:13:29.596 CET	guillaume	tests	6736	[local]	47c529b8.1a50	3	idle
21	2008-02-27 10:13:41.274 CET	guillaume	tests	6736	[local]	47c529b8.1a50	4	idle
22	2008-02-27 10:14:41.408 CET	guillaume	tests	6736	[local]	47c529b8.1a50	5	idle
23	2008-02-27 10:14:48.233 CET	guillaume	tests	6736	[local]	47c529b8.1a50	6	idle
24	2008-02-27 10:15:59.983 CET	guillaume	tests	6736	[local]	47c529b8.1a50	7	idle
25	2008-02-27 10:16:00.515 CET	guillaume	tests	6736	[local]	47c529b8.1a50	8	SELECT
26	2008-02-27 10:16:10.585 CET	guillaume	tests	6736	[local]	47c529b8.1a50	9	idle
27	2008-02-27 10:16:15.634 CET	guillaume	tests	6736	[local]	47c529b8.1a50	10	idle
28	2008-02-27 10:24:35.144 CET	guillaume	tests	6736	[local]	47c529b8.1a50	11	idle

Et voilà. Vous devriez aboutir au résultat de la figure 2. Il n'y a pas d'en-tête sur le tableau mais le contenu des colonnes est décrit dans le tableau 1. Vous pouvez ensuite utiliser les fonctionnalités du tableur pour trier, filtrer, construire un rapport, embellir le journal.



	Standard	Standard	Standard	Standard	Standard	Standard
1	2008-02-29 11:25:05.513 CET			6500		47c7dd81
2	2008-02-29 11:25:05.569 CET			6503		47c7dd81
3	2008-02-29 11:25:05.571 CET			6498		47c7dd81
4	2008-02-29 11:25:08.719 CET			6507		47c7dd84
5	2008-02-29 11:25:08.719 CET	guillaume	postgres	6507	[local]	47c7dd84
6	2008-02-29 11:25:08.849 CET	guillaume	postgres	6507	[local]	47c7dd84
7	2008-02-29 11:25:08.973 CET	guillaume	postgres	6507	[local]	47c7dd84

Intégration du journal dans une table d'une base PostgreSQL

Utiliser Calc est un moyen simple. Le gros inconvénient est qu'il faut charger en permanence les nouveaux fichiers et qu'il est plus difficile d'automatiser certaines recherches. Un administrateur de bases de données préfère souvent garder l'information dans une base SQL qu'il pourra accéder de partout. Le tout est de pouvoir intégrer facilement les données du fichier au format CSV dans une table. Sous PostgreSQL, c'est possible. Il existe même une instruction SQL pour cela, la commande COPY.

Cette commande existe depuis bien longtemps, mais ce n'est qu'à partir de la version 8.0 que le mode CSV a été ajouté. Elle permet principalement d'importer (COPY FROM) et d'exporter (COPY TO) des données dans un format CSV ou s'en rapprochant. Cette instruction est principalement utilisée dans les sauvegardes au format SQL car elle permet un enregistrement très rapide des données dans une table.

Il ne nous reste plus qu'à avoir la structure de la table qui va accueillir les données. Le manuel PostgreSQL donne directement l'instruction SQL de création. La voici pour mémoire :

```
CREATE TABLE postgres_log
(
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
  sql_state_code text,
  message text,
  detail text,
  hint text,
  internal_query text,
  internal_query_pos integer,
  context text,
  query text,
  query_pos integer,
  location text,
  PRIMARY KEY (session_id, session_line_num)
);
```

Une fois la table créée, il ne nous reste plus qu'à importer les données avec la commande :

```
COPY postgres_log FROM '/chemin/complet/vers/le/journalapplicatif.csv' WITH csv;
```

Il est donc très facile d'automatiser cette commande via un script cron. L'important est de savoir quels ont été les journaux déjà importés, pour pouvoir importer les autres. Il est possible d'utiliser une autre table, qui contiendra une ligne par journal importé.

```
CREATE TABLE imported_logs
(
  log_filename text,
  imported_time  timestamp with time zone DEFAULT now(),
  PRIMARY KEY (log_filename)
);
```

Le script va tout d'abord récupérer la liste des journaux qui se trouvent dans cette table et rechercher tous ceux qui ne correspondent pas à cette liste. Lorsque le script en trouve un, il l'importe immédiatement. Voici un exemple du script bash en question :

```
#!/bin/sh

# Personnalisation possible
PGDATA=${PGDATA:-/chemin/complet/vers/le/repertoire/des/donnees}
# Fin de la personnalisation possible du script

cd $PGDATA/pg_log

requete_journal="SELECT count(*) FROM imported_logs WHERE log_filename="
for fichier in *.csv
do
  existe=$(psql -Atc "$requete_journal '$fichier'" logs)
  if test $existe -eq "0"
  then
    psql logs <<_EOF_SQL_
BEGIN;
COPY postgres_log FROM '$PGDATA/pg_log/$fichier' WITH csv;
INSERT INTO imported_logs (log_filename) VALUES ('$fichier');
COMMIT;
_EOF_SQL_
  fi
done
```

En ajoutant ce script dans un cron quotidien, les journaux sont automatiquement importés dans la table.

Note : Ce script, bien que fonctionnel, n'est pas écrit dans les règles de l'art. Il serait déjà préférable de l'écrire dans un langage plus évolué comme Perl ou Python. Il serait bien vu aussi de vérifier les erreurs éventuelles. Sans compter qu'il est loin d'être performant. En effet, il est préférable de récupérer la liste des journaux importés en une seule requête et de comparer cette liste au fichier en cours de traitement, plutôt que d'exécuter une requête pour vérifier la présence de chaque fichier. Cela étant dit, ce n'est pas le but de cet article, l'auteur laisse donc l'amélioration de ce script en exercice à ses lecteurs.

Utilisation de la table

Une fois que les données sont dans la table, leur gestion est très simple. Il suffit d'utiliser les ordres SQL habituels.

Par exemple, si un administrateur veut récupérer tous les messages d'erreur, il peut utiliser cette requête :

```
SELECT log_time, error_severity, message
FROM postgres_log
WHERE error_severity = 'ERROR';
ce qui lui donnera un résultat ressemblant à ceci :
          log_time          | error_severity |           message
----------------------------+----------------+-----------------------------
 2008-03-12 09:16:47.521+01 | ERROR          | syntax error at or near ";"
(1 ligne)
```

S'il souhaite récupérer toutes les traces des trois dernières heures, il lui suffit d'exécuter cette requête :

```
SELECT log_time, message
FROM postgres_log
WHERE log_time > now() - '3 hours'::interval;
```

et voici un résultat possible :

```
          log_time         |                       message
---------------------------+---------------------------------------------------
 2008-03-12 09:16:12.872+01 |  connexion reçue : hôte=[local]
 2008-03-12 09:16:12.872+01 |  connexion autorisée : utilisateur=guillaume, base de données=tests
 2008-03-12 09:16:18.292+01 |  instruction : show lc_messages;
[...]
```

Les tris sont aussi possibles, la plus fréquente étant par date :

```
SELECT log_time, message
FROM postgres_log
WHERE log_time>now() - '3 hours'::interval
ORDER BY log_time;
```

La clause de regroupement permet de savoir, par exemple, le nombre d'erreurs par jour, mais aussi le nombre de messages par jour et par sévérité :

```
logs=# SELECT log_time::date, count(*) FROM postgres_log
logs-# WHERE error_severity='ERROR' GROUP BY 1;
  log_time  | count
------------+-------
 2008-03-12 |     1
(1 ligne)

logs=# SELECT log_time::date, error_severity, count(*) FROM postgres_log
```

```
logs-# GROUP BY 1, 2 ORDER BY 1, 2;
  log_time  | error_severity | count
------------+----------------+-------
 2008-02-27 | FATAL          |     2
 2008-02-27 | INFO           |     7
 2008-02-27 | LOG            |   102
 2008-02-28 | ATTENTION      |     1
 2008-02-28 | FATAL          |     1
 2008-02-28 | LOG            |  1203
 2008-02-29 | FATAL          |     1
 2008-02-29 | LOG            |    64
 2008-03-12 | ERREUR         |    58
 2008-03-12 | INFO           |     2
 2008-03-12 | LOG            |   446
```

Il est aussi possible d'utiliser les vues pour conserver des rapports intéressants, voire même les combiner. Bref, les possibilités sont nombreuses.

Conclusion

Cette vue complète des possibilités de journalisation applicative de PostgreSQL montre ses grandes capacités dans ce domaine ainsi qu'une possibilité importante de configuration. C'est le genre de capacités qui plaira à tout administrateur de bases de données. Libre à lui d'utiliser des outils supplémentaires, comme pgFouine, pour faciliter son travail sur les traces enregistrées par PostgreSQL.