

UTILISER PGBADGER POUR ANALYSER LES PERFORMANCES POSTGRESQL



SOMMAIRE

1. Introduction à PgBadger
2. Mise en place
 - Configuration des logs PostgreSQL
 - Installation de l'outil
3. Génération de rapports
 - Ligne de commande et options
 - Cas d'usage
4. Lecture et interprétation
5. Bonnes pratiques
6. Conclusion et récapitulatif

INTRODUCTION À PGBADGER

QU'EST-CE QUE PGBADGER ? 🤔

- Outil écrit en [Perl](#).
- Génère des rapports HTML interactifs, texte, CSV ou JSON.
- Centralise et visualise l'activité de PostgreSQL issue des logs
- Open source, gratuit, largement utilisé.
- Analyse les logs PostgreSQL (également pgBouncer)
- Aucune extension ni dépendance côté serveur
- Idéal pour une analyse rétrospective et post-mortem

POURQUOI L'UTILISER ?

- Identifier rapidement les **requêtes lentes**.
- Comprendre **qui fait quoi** dans la base (utilisateurs, applications, IPs).
- Suivre les tendances de charge (pics d'activité, périodes creuses).
- Améliorer la communication entre **DBA et développeurs**.
- Complémentaire aux autres outils de monitoring.
- Facile à intégrer dans des workflows automatisés (cron, CI/CD).
- Permet une analyse rétrospective (logs historiques).

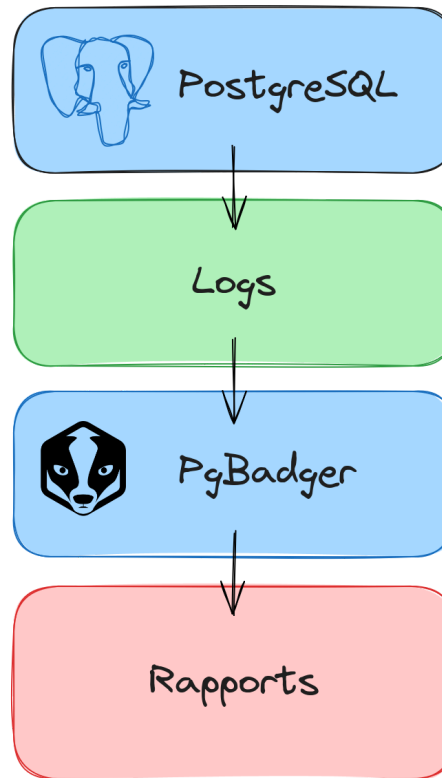
COMPARAISON AVEC OUTILS NATIFS

- `pg_stat_statements` → analyse en temps réel, extension SQL.
- `pg_stat_activity` → donne l'activité en temps réel
- `auto_explain` → enregistre les plans d'exécution dans les logs.
- `PgBadger` → analyse rétrospective des logs.

COMPARAISON AVEC OUTILS DE MONITORING

Outils	Analyse	Temporalité
Prometheus + Grafana	CPU, RAM, I/O, ...	Direct
Datadog	CPU, RAM, I/O, Connexions, requêtes, Wal, ...	Direct
Zabbix	CPU, RAM, I/O, + sonde pg	Direct
services managés (RDS, Azure, etc.)	CPU, RAM, I/O, ...	Direct
pgwatch2	CPU, RAM, I/O, Connexion, activité sql, vacuum, Wal	Direct
pgBadger	Connexions, Wal, erreurs, requête	a posteriori

SCHÉMA : RÔLE DE PGBADGER



MISE EN PLACE

CONFIGURATION DES LOGS POSTGRESQL

Exemple de configuration dans postgresql.conf :

```
logging_collector = on
log_directory = 'pg_log'
log_filename = 'postgresql-%Y-%m-%d.log'
log_line_prefix = '%t [%p]: [%l-1] user=%u,db=%d,app=%a,client=%h '
log_statement = 'none'
log_min_duration_statement = 500      # Requetes > 500 ms
log_checkpoints = on
log_connections = on
log_disconnections = on
log_lock_waits = on
log_temp_files = 0
lc_messages='en_US.UTF-8'
lc_messages='C'
```

`logging_collector = 'on'`

- 👉 Active le collecteur de logs interne de PostgreSQL.
- ⚠ Sans ça, PostgreSQL écrit les logs directement dans stderr ou le journal système (syslog, journald).
- 👉 Avec on, les logs sont capturés et écrits dans des fichiers gérés par PostgreSQL.

Indispensable pour centraliser les logs dans des fichiers définis par `log_directory` et `log_filename`.

log_directory = 'pg_log'

- Définit le répertoire (relatif au PGDATA, sauf si chemin absolu) où PostgreSQL enregistre les fichiers de log.
- Ici, les logs seront dans : \$PGDATA/pg_log/
- Le répertoire doit être accessible en écriture par l'utilisateur PostgreSQL.

log_filename = 'postgresql-%Y-%m-%d.log'

- Schéma de nommage des fichiers de log.
 - %Y → année
 - %m → mois
 - %d → jour
- Exemple : le 26 août 2025 → postgresql-2025-08-26.log
- Permet de changer de fichier log chaque jour (rotation quotidienne).

**log_line_prefix = '%t [%p]: [%l-1]
user=%u,db=%d,app=%a,client=%h'**

- Définit le préfixe ajouté au début de chaque ligne de log.
- Les variables insérées :
 - %t → horodatage (timestamp)
 - %p → PID du processus serveur
 - %l → numéro de ligne dans le log pour cette session (commence à 1)
 - %u → utilisateur PostgreSQL
 - %d → base de données utilisée
 - %a → nom de l'application (fourni par le client)
 - %h → adresse du client

EXEMPLE DE LIGNE DE LOG :

```
2025-08-26 14:12:35.123 CEST [12345]: [1-1] user=alice,db=mydb,app=psql, client=192.168.1.10
```

`log_statement = 'none'`

- Contrôle le niveau de verbosité des logs SQL.
- Options possibles :
 - `none` → ne trace aucune requête complète
 - `ddl` → trace uniquement les commandes DDL (CREATE, ALTER, DROP, ...)
 - `mod` → trace DDL + commandes qui modifient les données (INSERT, UPDATE, DELETE, ...)
 - `all` → trace toutes les requêtes SQL (très verbeux et coûteux !)

 Ici : `none`, donc on ne trace pas toutes les requêtes, mais on préfère cibler via `log_min_duration_statement`.

Log_min_duration_statement = 500

- trace uniquement les requêtes qui prennent plus de 500 ms.
- Très utile pour identifier les requêtes lentes.
- Valeurs possibles :
 - -1 → désactivé
 - 0 → trace toutes les requêtes avec leur durée (utile pour profiling mais coûteux)
 - >0 → trace uniquement si temps d'exécution supérieur à la valeur (en ms).



Ici : toutes les requêtes > 500 ms seront tracées.



Commencez avec un seuil de 500 ms, puis ajustez selon votre contexte.

`log_checkpoints = 'on'`

- Active le log de chaque checkpoint.
- Permet de surveiller la fréquence et la durée des checkpoints.

👉 Utile pour diagnostiquer des problèmes de performance liés à l'I/O. ou le paramétrage de `checkpoint_timeout`, `max_wal_size`

log_connections = 'on'

- Trace chaque nouvelle connexion au serveur PostgreSQL.
- Exemple de message :

```
connection authorized: user=alice database=mydb  
application=psql client=192.168.1.10
```

- Utile pour la traçabilité (auditer qui se connecte et quand).

`log_disconnections = 'on'`

- Trace chaque déconnexion de session.
- Donne aussi la durée de la session.

Exemple :

```
disconnection: session time: 1:23:45.678  
user=alice database=mydb host=192.168.1.10
```

`log_lock_waits = 'on'`

- Trace quand une requête attend un verrou trop longtemps.
- La durée de déclenchement dépend de `lock_timeout` (par défaut 1s).

👉 Très utile pour diagnostiquer les contentions et problèmes de concurrence.

log_temp_files = 0

- Log tous les fichiers temporaires
- Mémoire opération > work_mem ou maintenance_work_mem
 - →  opérations sur disque (fichiers temporaires)
 - ⇒  ralentissement.
- Usual suspects :
 - tri (ORDER BY, DISTINCT, GROUP BY)
 - hash join ou hash aggregate
 - CTE / sous-requêtes complexes
 - création d'index

lc_messages

Ce paramètre détermine la langue des messages système et d'erreur de PostgreSQL.

⚠ vos messages doivent être rédigés en anglais

```
# anglais américain  
lc_messages='en_US.UTF-8'  
  
# ou  
  
# locale standard, toujours en anglais,  
# avec tri et messages simples.  
lc_messages='C'
```

INSTALLATION DE PGBADGER

DEBIAN/UBUNTU

Les packages sont disponible dans les [dépôts PostgreSQL](#)

```
sudo apt install curl ca-certificates
sudo install -d /usr/share/postgresql-common/pgdg
sudo curl -o /usr/share/postgresql-common/pgdg/apt.postgresql.org.asc \
  --fail https://www.postgresql.org/media/keys/ACCC4CF8.asc

. /etc/os-release
sudo sh -c "echo 'deb [signed-by=/usr/share/postgresql-common/pgdg/apt.postgresql.org.asc] \
  https://apt.postgresql.org/pub/repos/apt $VERSION_CODENAME-pgdg main' \
  > /etc/apt/sources.list.d/pgdg.list"

# Update the package lists:
sudo apt update
sudo apt install pgbadger
```

RHEL / CENTOS

Pour les paquets RPM, sont disponible dans le dépôt [yum de PostgreSQL](#).

```
# Install the repository RPM:
sudo dnf install -y \
  https://download.postgresql.org/pub/repos/yum/\
  repopms/EL-10-x86_64/pgdg-redhat-repo-latest.noarch.rpm

sudo dnf install pgbadger
```

DEPUIS LES SOURCES

```
wget https://github.com/darold/pgbadger/archive/refs/tags/v13.1.zip  
tar xvf v13.1.tar.gz  
cd pgbadger-13.1  
perl Makefile.PL && make && sudo make install
```

GÉNÉRATION DE RAPPORTS

COMMANDE DE BASE

```
pgbadger /var/log/postgresql/postgresql-2025-08-25.log -o rapport.html
```

OPTIONS UTILES

- -o → sortie (HTML, JSON, texte, CSV).
- -p → motif de parsing des fichiers.
- -I → mode incrémental (mise à jour continue).
- -j → parallélisation de l'analyse.

Exemple :

```
pgbadger -j 4 -p '%t [%p]: [%l-1] user=%u,db=%d,app=%a,client=%h ' \  
/var/log/postgresql/postgresql-*.log -o rapport.html
```

CAS D'USAGE

RAPPORT JOURNALIER

```
pgbadger /var/log/postgresql/postgresql-$(date +%Y-%m-%d).log \  
-o $(date +%Y-%m-%d).html
```

RAPPORT PAR BASE DE DONNÉES :

```
-E --explode
```

- rapport par base de données
- informations globales non liées dans le fichier de sortie.

```
pgbadger -E /var/log/postgresql/postgresql-$(date +%Y-%m-%d).log \  
-o pgbadger.html  
ls -l  
pgbadger.html  
cqs_pgbadger.html  
rdsadmin_pgbadger.html
```

EXCLURE DES REQUÊTES INTERNES DE MONITORING

Si un monitoring est en place comme des sondes Nagios, Zabbix ou Prometheus, leurs requêtes reviennent sans cesse et peuvent cacher des informations utiles.

```
pgbadger /var/log/postgresql/postgresql.log \  
--exclude-query="SELECT 1" \  
--exclude-query="SHOW.*" \  
--exclude-appname="check_postgres"
```

👉 Permet de supprimer du rapport toutes les requêtes de supervision, pour se concentrer sur les vraies requêtes applicatives.

EXCLUDE DES UTILISATEURS SPÉCIFIQUES

Par exemple, un utilisateur d'ETL (Airflow, Talend, etc.) fait tourner des batchs massifs. On souhaite ne voir que les sessions web/app.

```
pgbadger /var/log/postgresql/postgresql.log \  
--exclude-user="etl_user"
```

EXCLUDE CERTAINES BASES

On veut analyser les logs liés aux bases de prod :

```
pgbadger /var/log/postgresql/postgresql.log \  
--exclude-database="template1" \  
--exclude-database="testdb"
```

EXCLUDE CERTAINES ERREURS "CONNUES"

Par exemple des erreurs de duplicate key gérées côté app (pas besoin de les voir dans le rapport) :

```
pgbadger /var/log/postgresql/postgresql.log \  
--exclude-error="duplicate key value violates unique constraint"
```

RAPPORT INCRÉMENTAL

PRINCIPE

- pgBadger garde un état dans un répertoire (`--incremental --outdir /path/to/dir`)
- Un rapport par jour + mise à jour du rapport hebdomadaire
- Chaque nouvelle exécution va traiter uniquement les nouveaux logs et mettre à jour le rapport.

👉 C'est très pratique sur un serveur en production où les logs sont lourds (plusieurs Go/jour).



EXEMPLE SIMPLE

Supposons que les logs PostgreSQL soient dans `/var/log/postgresql/`.

```
pgbadger /var/log/postgresql/postgresql-*.log \  
--incremental \  
--outdir /var/www/html/pgbadger \  
--jobs 4 \  
--quiet
```

Dans `/var/www/html/pgbadger/` on aura un répertoire par semaine et un répertoire par jour contenant chacun un fichier `index.html`.

RAPPORT MENSUEL

- Par défaut : rapports quotidiens et hebdomadaires
 - Commande distincte pour les rapports mensuels cumulés
- Par exemple, pour créer un rapport pour le mois d'août 2025 :

```
pgbadger --month-report 2025-08 /var/www/pg_reports/
```

- Nouveau lien dans le calendrier des rapports
- Reconstruit à chaque execution
- L'option (-E | --explode) doit être appelée si utilisée pour les rapports journaliers

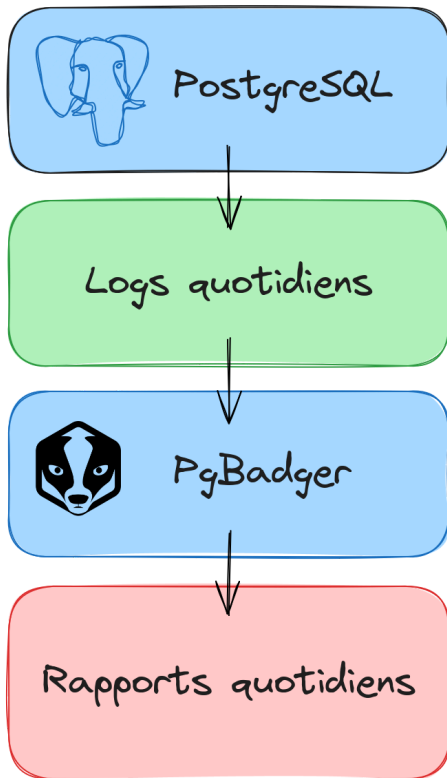
```
pgbadger -E --month-report 2025-08 /var/www/pg_reports/
```

- Idem pour l'option de reconstruction (-R | --rebuild).

EXEMPLE DE CRON (QUOTIDIEN À 1H DU MATIN)

```
0 1 * * * pgbadger /var/log/postgresql/postgresql-*.log \  
--incremental \  
--outdir /var/www/html/pgbadger \  
--jobs 4 --quiet
```

SCHÉMA : CYCLE D'UTILISATION TYPIQUE



- 🔍 Développeurs consultent les requêtes lentes
- 🐾 DBAs ajustent la config et détectent anomalies

LECTURE ET INTERPRÉTATION

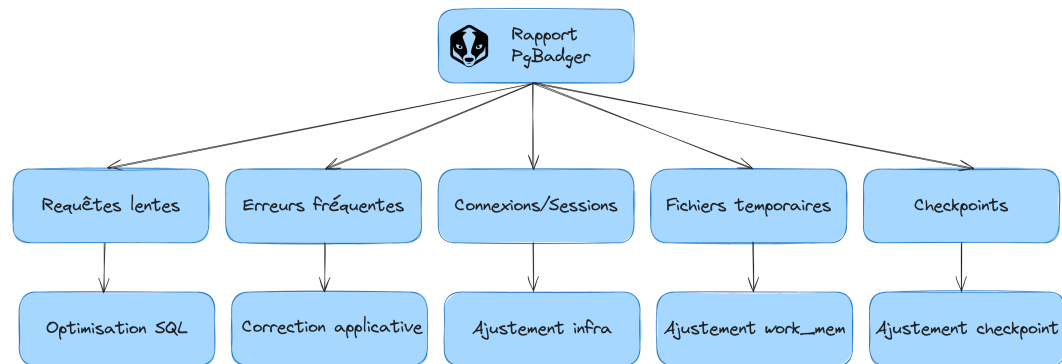
SECTIONS CLÉS DU RAPPORT

- **Résumé général** : Connexions, durée totale, taille des logs.
- **Connections** : Connections par base, utilisateur, host.
- **Sessions** : Sessions par base, utilisateur, host, application.
- **Checkpoints** : Détails sur les checkpoints (nombre, taille, distance, causes)
- **Fichiers temporaires** : Nombre et taille totale, requêtes qui génèrent le plus de fichiers temporaires.
- **Vacuum** : Activité, distribution des vacuums, analyses
- **Locks** : Requêtes ayant attendu le longtemps.
- **Queries** : Distributions des requêtes par base, utilisateur, host, application.
- **Top** : TOP N queries les plus lentes, plus fréquentes.
- **Events** : Liste et classification des evenements rencontrées (erreurs, warning)

EXEMPLES DE DÉTECTION

- Requête SELECT prenant 5s plusieurs fois par jour.
- Application sans pooling → surcharge de connexions.
- Locks fréquents → besoin d'optimiser les transactions.
- Fichier temporaire → optimiser les requêtes, éventuellement augmenter le `work_mem`
- Checkpoints trop fréquents → augmenter le `max_wal_size`

SCHÉMA : LOGIQUE D'INTERPRÉTATION



BONNES PRATIQUES

- Calibrer correctement `log_min_duration_statement`.
- Ne pas tracer toutes les requêtes en production (surcharge).
- Automatiser la génération de rapports (cron, systemd timer).
- Partager les rapports entre équipes **Dev** et **Ops**.
- Utiliser PgBadger comme **point d'entrée** puis approfondir avec EXPLAIN, `pg_stat_statements`, `auto_explain`.

CONCLUSION

Maintenant, vous savez :

- Configurer PostgreSQL pour générer des logs exploitables par PgBadger.
- Installer et exécuter PgBadger.
- Lire et interpréter les rapports générés.
- Identifier les requêtes problématiques.
- Favoriser le diatrace entre Dev et DBA grâce à des données objectives.

EXEMPLE

Allons voir la [Demo](#)

MERCI!



Merci à tous pour votre attention!

