











Découvrez nos magazines :







- GNU/Linux Magazine
  -  [Dernier Numéro Paru](#)
  -  [Tous les anciens Numéros](#)
  -  [Abonnements : GNU/Linux Magazine](#)
  -  [Toutes les offres d'abonnement avec ce titre](#)







- GNU/Linux Magazine HORS-SÉRIE
  -  [Dernier Numéro Paru](#)
  -  [Tous les anciens Numéros](#)
  -  [Abonnements : GNU/Linux Magazine HS](#)
  -  [Toutes les offres d'abonnement avec ce titre](#)







- Linux Pratique
  -  [Dernier Numéro Paru](#)
  -  [Tous les anciens Numéros](#)
  -  [Abonnements : Linux Pratique](#)
  -  [Toutes les offres d'abonnement avec ce titre](#)



- Linux Pratique HORS-SÉRIE
  -  [Dernier Numéro Paru](#)
  -  [Tous les anciens Numéros](#)
  -  [Abonnements : Linux Pratique HS](#)
  -  [Toutes les offres d'abonnement avec ce titre](#)



- Linux Pratique Essentiel
  -  [Dernier Numéro Paru](#)
  -  [Tous les anciens Numéros](#)
  -  [Abonnements : Linux Pratique Essentiel](#)
  -  [Toutes les offres d'abonnement avec ce titre](#)

- 

Linux Pratique Essentiel HORS-SÉRIE
  -  [Dernier Numéro Paru](#)
  -  [Tous les anciens Numéros](#)
  -  [Abonnements : Linux Pratique Essentiel HS](#)
  -  [Toutes les offres d'abonnement avec ce titre](#)
- 

Misc
  -  [Dernier Numéro Paru](#)
  -  [Tous les anciens Numéros](#)
  -  [Abonnements : MISC](#)
  -  [Toutes les offres d'abonnement avec ce titre](#)
- 

Misc HORS-SÉRIE
  -  [Dernier Numéro Paru](#)
  -  [Tous les anciens Numéros](#)
  -  [Abonnements : MISC HS](#)
  -  [Toutes les offres d'abonnement avec ce titre](#)
- 

Open Silicium
  -  [Dernier Numéro Paru](#)
  -  [Tous les anciens Numéros](#)
  -  [Abonnements : Open Silicium](#)
  -  [Toutes les offres d'abonnement avec ce titre](#)

Il y a 1,110 articles/billets en ligne.

Publié(s) dans le(s) magazine(s) :

GNU/Linux Magazine	Linux Pratique	Linux Pratique Essentiel	Misc
GNU/Linux Magazine Hors-séries	Linux Pratique Hors-séries	Linux Pratique Essentiel Hors-séries	Misc Hors-séries

Mots-clés

Categories

[Administration système](#)

# Les vues système sous PostgreSQL 8.3

08/04/2009

Posté par [La rédaction](#) | Signature : Guillaume LelargeTags : [GLMF](#)[0 Commentaire](#) | [Ajouter un commentaire](#)Retrouvez cet article dans : [Linux Magazine 106](#)

La journalisation applicative permet à tout administrateur de bases de données de retrouver des informations importantes, notamment en cas de problème sur le serveur. Le cadre est principalement historique : retrouver une trace pouvant fournir des informations importantes sur un problème particulier, y compris quinze jours après. Mais, il est aussi essentiel de savoir ce qui se passe en temps réel. La commande `ps` ne fournira pas tous les renseignements dont on a besoin. PostgreSQL propose donc des informations qu'il est possible de retrouver grâce à des tables et à des vues système.

# 1. Surveillance de l'activité système

La première surveillance à exercer concerne l'activité des clients sur le serveur : qui est connecté, sur quelle base de données, quelles requêtes sont-elles en cours d'exécution, quels verrous ont-ils été posés, etc. ? Il est aussi important de savoir si des verrous sont posés, de quel type sont ces verrous.

## 1.1 Processus

L'un des points essentiels de la surveillance d'un serveur de bases de données concernent les connexions aux bases de données. Il s'agit principalement de surveiller le nombre de connexions établies, mais aussi de connaître les utilisateurs connectés, les bases utilisées, les requêtes en cours d'exécution, etc. Le nombre de connexions peut évidemment se retrouver avec la commande `ps` en ne récupérant que les lignes comprenant le texte `postgres` :

```
guillaume@laptop:~$ ps xfo args | grep [p]ostgres
/opt/postgresql-8.3/bin/postgres
\_ postgres: logger process
\_ postgres: writer process
\_ postgres: wal writer process
\_ postgres: autovacuum launcher process
\_ postgres: stats collector process
\_ postgres: guillaume benchs [local] idle
\_ postgres: guillaume pagila 127.0.0.1(40808) idle
\_ postgres: guillaume tests 127.0.0.1(40809) INSERT
\_ postgres: postgres benchs [local] idle in transaction
\_ postgres: guillaume pagila [local] idle
```

De cette sortie, nous apprenons plusieurs choses. Tout d'abord, le type d'activité du processus : ~~logger process~~ pour le processus de journalisation applicative, ~~writer process~~ pour le processus d'écriture sur disque des fichiers de données, ~~wal writer process~~ pour le processus d'écriture des journaux de transaction, ~~autovacuum launcher process~~ pour le processus autovacuum principal, ~~autovacuum worker process~~ pour les processus autovacuum qui exécutent les opérations de maintenance `VACUUM` et `ANALYZE`, ~~stats collector~~ pour le récupérateur de statistiques, et les autres qui s'occupent chacun d'une connexion d'un client à une base. Pour ceux-là, nous avons plusieurs informations :

- le nom de l'utilisateur PostgreSQL connecté ;
- la base de connexion ;
- le type de connexion (local dans le cas d'une connexion par socket Unix, l'adresse IP et le port externe du client pour une connexion TCP/IP) ;
- l'action en cours par ce client.

Cette action peut indiquer ~~idle~~, ce qui signifie que le client n'a demandé l'exécution d'aucune requête, ~~idle in transaction~~, qui signifie que le client n'a demandé l'exécution d'aucune requête, mais qu'il se trouve dans une transaction, ou le mot clé de l'instruction SQL qu'il exécute en ce moment (par exemple ~~INSERT~~, ~~SELECT~~ ou ~~VACUUM~~). Le mot ~~waiting~~ peut être attaché à la suite de l'action pour indiquer que l'exécution de la requête est en attente, car un verrou d'un autre processus l'empêche d'accéder à un objet dont elle a besoin. Attention à bien activer le paramètre ~~update\_process\_title~~ pour avoir l'information de l'action. Ce paramètre n'est disponible qu'à partir de la version 8.2. Les versions précédentes utilisent ~~stats\_command\_string~~.

Ce sont des informations très intéressantes, mais le moyen de les récupérer demande d'avoir un accès shell à la machine, ce qui n'est pas toujours simple à avoir. Il est souvent préférable de pouvoir obtenir ces informations à partir d'un catalogue système. Ce catalogue s'appelle ~~pg\_stat\_activity~~. Le nombre de lignes proposées par cette vue est inférieur au résultat de la commande ~~ps~~ précédente, car seules les connexions à une base de données sont retenues. Autrement dit, les processus d'écriture ou de trace ne sont pas reflétés dans cette vue. Malgré tout, elle propose un peu plus d'informations par processus :

```
pagila=# SELECT * FROM pg_stat_activity LIMIT 1;
-[ RECORD 1 ]-+-----
datid         | 16432
datname       | benchs
procpid       | 5541
usesysid     | 10
username     | guillaume
current_query | <IDLE>
waiting       | f
xact_start    |
query_start   |
backend_start | 2008-04-14 14:14:21.087267+02
client_addr   |
client_port   | -1
```

Les premières informations concernent la connexion : qui est connecté et sur quoi. La colonne ~~datid~~ correspond à l'OID (Object IDentifier) de la base de données. Le nom de la base se trouve dans la colonne ~~datname~~. ~~procpid~~ présente le PID (Process IDentifier) du processus postgres. ~~usesysid~~ est l'OID de l'utilisateur connecté, et ~~username~~ est le nom de connexion de cet utilisateur. Tout en fin de tableau se trouvent les colonnes ~~client\_addr~~ correspondant à l'adresse IP de connexion et le port du client (respectivement ~~NULL~~ et ~~-1~~ dans le cas d'une connexion locale).

Ensuite, les informations présentées concernent l'action en cours du processus. Les colonnes ~~xact\_start~~ (nouvelle colonne en 8.3), ~~query\_start~~, ~~backend\_start~~ sont des horodatages, respectivement du début de la transaction, du début d'exécution de la requête, du début de connexion du client. ~~current\_query~~ est la requête en cours d'exécution. Il s'agit cette fois de la requête complète. ~~waiting~~, elle-aussi ajoutée en 8.3, indique si le processus est en attente d'un verrou pour exécuter la requête. Lorsqu'un client envoie sa requête au SGBD, celle-ci est analysée par PostgreSQL. Pour répondre à la demande du client, PostgreSQL peut avoir à poser certains

verrous. Si des verrous conflictuels sont déjà posés par d'autres processus sur les mêmes objets, l'exécution de la requête est mise en attente de la disponibilité des objets souhaités. Tant qu'ils ne sont pas utilisables par notre processus serveur, la requête est en attente. Une fois les objets disponibles, la requête s'exécute. Autrement dit, le temps d'exécution d'une requête dépend du temps d'attente de la disponibilité des objets et du temps d'exécution proprement dit. Plus le temps d'attente est long, plus le client attend le retour, plus il a l'impression que la requête s'exécute lentement, et donc que le moteur n'est pas rapide. C'est ce qu'on appelle la réactivité du moteur. Cette colonne vous permet de "mesurer" cela. Plus vous avez de lignes avec une colonne `waiting` à `true`, plus la réactivité du serveur peut laisser à désirer. Dans ce cas, inutile dans un premier temps de récrire vos requêtes. Commencez par chercher pourquoi vous avez autant de verrous bloquant. Par contre, si vous n'avez aucun verrou et que vos requêtes sont longues à s'exécuter, c'est que le problème se trouve ailleurs. Peut-être faut-il récrire vos requêtes ou mieux configurer le serveur.

La colonne `current_query` a un fonctionnement particulier. Elle n'est renseignée que si le paramètre `update_process_title` est activé. Attention, ce paramètre existait déjà avant la version 8.3 sous le nom de `stats_command_string` pour les versions précédentes. Récupérer chaque requête à exécuter prend du temps et cela a un coût au niveau des performances générales du système. Le fait de pouvoir l'activer ou le désactiver, suivant l'impact détecté, donne un grand contrôle à l'administrateur de bases de données. Il peut choisir de la désactiver la majorité du temps et ne l'activer que dans certains cas, par exemple pour détecter des requêtes qui entrent en conflit pour des verrous. Il est à noter que des améliorations ont eu lieu sur les performances, ce qui a permis d'activer par défaut ce paramètre pour toute base de données en 8.3. Enfin, dernier point, la sécurité est bien évidemment gérée sur cette vue. Seuls les superutilisateurs ont droit de voir les requêtes des autres utilisateurs. Un utilisateur "simple" voit les requêtes dont il est l'exécuteur, et voit la chaîne `<insufficient privilege>` dans tous les autres cas. Les colonnes `waiting`, `xact_start`, `query_start`, `backend_start`, `client_addr` et `client_port` sont à `NULL` dans ce dernier cas.

En connaissant le PID du processus (colonne `procpid`) et la requête, il est très simple d'annuler l'exécution d'une requête qui impacte fortement les performances, soit parce qu'elle bloque d'autres tables (voir pour cela le chapitre suivant), soit parce qu'elle utilise beaucoup les disques. La fonction `pg_cancel_backend()` est prévue pour cela. Il suffit de lui indiquer le PID du processus. Un simple `SELECT pg_cancel_backend(21234);` annule la requête en cours d'exécution par le processus de PID 21234 (à condition que celui qui demande l'annulation de la requête soit un superutilisateur). Si la requête annulée était exécutée à l'intérieur d'une transaction explicite, toute modification entreprise par la transaction sera aussi annulée.

## 1.2 Verrous

Comme dit ci-dessus, le temps d'exécution d'une requête dépend aussi du temps d'attente des objets verrouillés par d'autres processus. Il est donc important de pouvoir connaître les verrous posés en temps réel, savoir lesquelles sont actifs, lesquels sont en attente, etc. La vue `pg_locks` fournit toutes ces informations et même plus :

```
pagila=# SELECT locktype, database::regclass, relation::regclass,
pagila=# page, tuple, virtualxid, transactionid, classid, objid,
pagila=# objsubid, virtualtransaction, pid, mode, granted
```

```

pagila=# FROM pg_locks
pagila=# WHERE relation=10969;
-[ RECORD 1 ]-----+-----
locktype          | relation
database          | 18018
relation          | pg_locks
page              |
tuple             |
virtualxid        |
transactionid     |
classid           |
objid             |
objsubid          |
virtualtransaction | 1/12
pid               | 455
mode              | AccessShareLock
granted           | t

```

### Note

La notation `database::regclass` indique que l'OID doit être converti vers le type `regclass`, ce qui correspond au nom de l'objet. Du coup, `database::regclass` transforme l'OID contenu dans le champ `database` par le nom de la base de données.

Le premier champ indique le type de l'objet verrouillable. En voici la liste : `relation`, `extend`, `page`, `tuple`, `transactionid`, `virtualxid`, `object`, `userlock`, `advisory`. Les plus courants sont `relation` quand le verrou est sur une table, `page` (quand le verrou concerne une page disque), `tuple` (principalement quand une ou plusieurs lignes d'une table sont visées), `transactionid` (l'objet verrouillé est l'identifiant d'une transaction explicite qui a modifié le contenu de la base de données), `virtualxid` (verrouillage de l'identifiant d'une transaction qui n'a pas encore modifié le contenu de la base de données). `userlock` et `advisory` concernent les verrous informatifs disponibles respectivement dans le module `contrib-userlock` et depuis la version 8.2 de PostgreSQL.

Le champ `database` indique l'OID de la base de données où le verrou a été posé, alors que `relation` précise l'OID de la relation dans le cas où le champ `locktype` vaut `relation`. Le champ `page` pousse la précision jusqu'au numéro de page, et le champ `tuple` va même jusqu'à indiquer la ligne concernée (pour les `locktype` impliqués).

Les deux champs suivants sont des informations sur la transaction qui a posé le verrou. Dans les versions antérieures à la 8.3, chaque transaction avait son identifiant, qu'elle fasse des modifications dans une base ou pas. Des identifiants de transactions pouvaient être consommés alors qu'ils ne traçaient pas des modifications en base. Comme ces identifiants sont en nombre limités, ce comportement était considéré comme inutile, voire gênant pour les performances et l'espace disque. En 8.3 est apparu un nouveau concept, celui du `lazy-xid` (identifiant de transaction paresseux). En gros, un identifiant de transaction est généré quand une transaction modifie le contenu de la base de données. Néanmoins, un numéro de transaction virtuel doit être utilisé dans les autres cas. Cet identifiant ne sera pas enregistré. Il est généré uniquement pour identifier temporairement la transaction. L'identifiant virtuel est donc indiqué dans cette vue grâce au champ `virtualxid`. Dès qu'une modification a lieu, un numéro de transaction "réel" est généré. Ce dernier apparaît dans le champ `transactionid`.

Les trois prochains champs précisent encore l'objet verrouillé. `classid` est l'OID du catalogue système contenant l'objet verrouillé. `objid` est l'OID de l'objet verrouillé. Pour une colonne de table,

~~objsubid~~ est le numéro de la colonne.

~~virtualtransaction~~ est l'identifiant virtuel de la transaction qui détient ou attend le verrou, ~~pid~~ est le PID du processus qui a demandé le verrou.

~~mode~~ peut avoir les valeurs suivantes :

- ~~Access Share~~, accès partagé, tout ~~SELECT~~ acquiert un verrou de ce type ;
- ~~Row Share~~, acquis par les commandes ~~SELECT FOR UPDATE~~ et ~~SELECT FOR SHARE~~ ;
- ~~Row Exclusive~~, acquis par les commandes ~~UPDATE~~, ~~DELETE~~ et ~~INSERT~~ ;
- ~~Share Update Exclusive~~, protège une table contre les modifications concurrentes de schéma et l'exécution d'un ~~VACUUM~~, acquis par ~~VACUUM~~ (sans le mode ~~FULL~~), ~~ANALYZE~~ et ~~CREATE INDEX CONCURRENTLY~~ ;
- ~~Share~~, protège une table contre les modifications de données concurrentes, acquis par ~~CREATE INDEX~~ (sans ~~CONCURRENTLY~~) ;
- ~~Share Row Exclusive~~, jamais acquis automatiquement avec les commandes SQL ;
- ~~Exclusive~~, ne concerne que certains catalogues système ;
- Access Exclusive, garantit un accès exclusif, acquis par les commandes ~~ALTER TABLE~~, ~~DROP TABLE~~, ~~TRUNCATE~~, ~~REINDEX~~, ~~CLUSTER~~, ~~VACUUM FULL~~.

Enfin, ~~granted~~ indique si le verrou est posé ou non. À ~~true~~, le verrou est détenu par le processus indiqué dans le champ ~~pid~~. À ~~false~~, le processus est en attente du verrou.

Dans cet exemple, la ligne affichée correspond au verrou posé sur la table ~~pg\_locks~~ pour la lecture nécessaire pour répondre à la requête. Le verrou est de type ~~AccessShare~~. Bien que le processus 455 le détient, le type même de ce verrou fait que tous les autres processus peuvent accéder à cette table, en lecture comme en écriture (en dehors du fait qu'il s'agit dans ce cas d'un catalogue système).

## 1.3 Processus d'écriture en tâche de fond

Ce processus, aussi appelé ~~bgwriter~~, s'occupe de copier les données modifiées en mémoire vive (en quelque sorte le cache disque de PostgreSQL) dans les fichiers de données. Il exécute son traitement à chaque ~~CHECKPOINT~~ manuel et automatique. Quand le cache disque est plein et qu'un processus postgres a besoin d'y stocker des données, ce dernier s'occupe lui-même d'enregistrer suffisamment de données sur disque pour pouvoir les remplacer par ses propres données. Si un processus doit faire cela en plus de son travail, les performances en pâtiront. Il faut donc s'assurer que la configuration de ~~bgwriter~~ est bien faite et, pour cela, les développeurs de PostgreSQL nous proposent depuis la version 8.3 cette nouvelle vue système appelée ~~pg\_stat\_bgwriter~~.

Cette vue ne renvoie qu'une seule ligne :

```
pagila=> SELECT * FROM pg_stat_bgwriter ;
-[ RECORD 1 ]-----
checkpoints_timed | 122
checkpoints_req   | 4
buffers_checkpoint | 11131
buffers_clean     | 6527
maxwritten_clean  | 52
buffers_backend   | 20848
buffers_alloc     | 17627
```

Le premier champ, `checkpoints_timed`, indique le nombre de `CHECKPOINT` dû au dépassement du délai précisé par le paramètre `checkpoint_timeout`. Le deuxième indique le nombre de `CHECKPOINT` demandé. Cela peut survenir suite à plusieurs événements :

- Dépassement d'un certain nombre de journaux de transactions écrits depuis le dernier `CHECKPOINT` manuel ou automatique. Ce nombre correspond au paramètre `checkpoint_segments`. Cela survient principalement pendant une grosse charge de données, par exemple lors de la restauration de la sauvegarde d'une base.
- Arrêt du serveur. La commande d'arrêt du serveur exécute un `CHECKPOINT` (pour les modes d'arrêt `smart` et `fast`).
- Exécution de la commande `SQL CHECKPOINT` par un utilisateur.

Le champ `buffers_checkpoint` indique le nombre de pages disque qui ont été écrites suite à un `CHECKPOINT`. En divisant `buffers_checkpoint` par `checkpoints_timed`, on obtient en moyenne le nombre de pages disque écrites toutes les `checkpoint_timeout` secondes, ce qui donne une idée des entrées/sorties réalisées sur le disque.

Le champ `buffers_clean` précise le nombre de pages disque modifiées nettoyées par `bgwriter`.

`maxwritten_clean` indique le nombre de fois où `bgwriter` a rencontré sa limite de pages disque avant d'avoir écrit toutes les pages modifiées sur disque. Un grand nombre ici indique que les valeurs des paramètres `bgwriter_lru_maxpages` et `bgwriter_lru_multiplier` sont à revoir, généralement à la baisse.

`buffers_backend` représente le nombre de pages disque écrites non pas par `bgwriter`, mais par les processus `postgres`. Dit autrement, `bgwriter` est soit trop occupé et ne peut donc pas écrire toutes les pages disque modifiées en mémoire sur disque (trop grande valeur du `checkpoint_segments`, trop petites valeurs de `bgwriter_lru_maxpages` et/ou `bgwriter_lru_multiplier`, voire trop grand `shared_buffers`), soit trop espacé (trop grande valeur pour `checkpoint_segments` et `checkpoint_timeout`).

`buffers_alloc` indique la quantité totale de pages disque placées en mémoire.

## 2. Statistiques sur les objets

Surveiller l'activité des différents processus relatifs au serveur de bases de données ne doit pas faire oublier la surveillance de l'état des objets de la base. PostgreSQL propose des vues de l'état et des actions réalisées sur chacun des objets qui ont une existence sur disque.

### 2.1 Bases de données

La vue nommée `pg_stat_databases` donne des statistiques sur les bases de données du serveur PostgreSQL. Elle renvoie une ligne par base :

```
pagila=# SELECT * FROM pg_stat_database WHERE datname='pagila';
-[ RECORD 1 ]-+-----
datid         | 18018
datname       | pagila
numbackends   | 2
xact_commit   | 1761
```



xact_rollback	11
blks_read	589
blks_hit	141702
tup_returned	845133
tup_fetched	27222
tup_inserted	0
tup_updated	206
tup_deleted	400

Pour identifier la base de données, les deux premiers champs, ~~datid~~ et ~~datname~~, précisent respectivement l'OID de la base de données et son nom.

~~numbackends~~ indique le nombre de clients actuellement connectés à cette base. En surveillant cette vue pour toutes les bases de données (par exemple, avec la requête `SELECT sum(numbackends) FROM pg_stat_database`), vous pouvez vous assurer que vous n'approchez pas trop de la limite fixée par le paramètre `max_connections`.

~~xact\_commit~~ donne le nombre de transactions qui ont bien été validées, alors que ~~xact\_rollback~~ est le nombre de transactions annulées. L'annulation peut être implicite ou explicite. Le nombre d'annulations doit être très inférieur à celui des validations. Dans le cas contraire, cela indique généralement une mauvaise programmation, ce qui aboutit à coup sûr à des performances médiocres.

~~blks\_read~~ correspond au nombre de pages disque lues sur disque (plus exactement, supposé lues sur disque, car rien ne dit que le noyau du système d'exploitation n'a pas déjà mis en cache ces pages disque), ~~blks\_hit~~ au nombre de pages récupérées dans le cache disque de PostgreSQL. La taille du cache disque se configure avec le très célèbre paramètre `shared_buffers`. Si une majorité des pages disque sont lues sur le disque, cela peut être la cause d'une mauvaise configuration de ce paramètre. Il est important d'avoir un ratio `blks_read/blks_hit` aussi petit que possible.

Les cinq prochains champs sont une nouveauté de la version 8.3. ~~tup\_returned~~ indique le nombre de lignes lues par des parcours séquentiels alors que ~~tup\_fetched~~ précise le nombre de lignes récupérées suite à un parcours de bitmap (donc en mémoire), ainsi que le nombre de lignes récupérées par des parcours d'index simples. ~~tup\_inserted~~, ~~tup\_updated~~, ~~tup\_deleted~~ sont respectivement le nombre de lignes insérées, mises à jour et supprimées pour cette base de données. Ces informations permettent de mieux apprécier l'activité en modification de la base de données.

## 2.2 Tables

Un administrateur de bases de données aime connaître l'impact des modifications au niveau de la fragmentation de la table (donc le nombre de lignes insérées, supprimées, modifiées), ainsi que l'impact sur le disque (le nombre de pages disque écrites). PostgreSQL fournit des informations statistiques sur tous ces points.

Trois catalogues système permettent de surveiller l'activité sur le contenu des tables :

~~pg\_stat\_all\_tables~~, ~~pg\_stat\_user\_tables~~, ~~pg\_stat\_sys\_tables~~. Elles ont exactement les mêmes colonnes. La première renvoie toutes les tables, la seconde uniquement les tables créées par des utilisateurs et la dernière les tables système. Chaque ligne prend en considération une table.

```

pagila=# SELECT * FROM pg_stat_user_tables WHERE relname='actor';
-[ RECORD 1 ]-----+-----
relid          | 18024
schemaname     | public
relname        | actor
seq_scan       | 10
seq_tup_read   | 1022
idx_scan       | 0
idx_tup_fetch  | 0
n_tup_ins      | 0
n_tup_upd      | 202
n_tup_del      | 400
n_tup_hot_upd  | 2
n_live_tup     | 200
n_dead_tup     | 0
last_vacuum    |
last_autovacuum | 2008-04-15 11:56:11.622041+02
last_analyze   |
last_autoanalyze | 2008-04-15 11:56:11.622041+02

```

Les trois premiers champs identifient la table par son OID (~~relid~~), le nom de son schéma (~~schemaname~~) et celui de la table (~~relname~~).

Le reste des colonnes est divisible en trois parties : les parcours, les modifications et les opérations de maintenance.

~~seq\_scan~~ comptabilise le nombre de parcours séquentiels réalisés sur cette table alors que ~~idx\_scan~~ s'occupe des parcours d'index. ~~seq\_tup\_read~~ correspond au nombre de lignes lues suite à un parcours séquentiel. Enfin ~~idx\_tup\_fetch~~ compte le nombre de récupérations de lignes par un index.

Dans les modifications sont tracées les insertions (~~n\_tup\_ins~~), les suppressions (~~n\_tup\_del~~), les mises à jour "standards" (~~n\_tup\_upd~~). Les trois colonnes suivantes datent de la version 8.3. Les mises à jour utilisant la nouvelle technique HOT sont tracées spécifiquement via la colonne ~~n\_tup\_hot\_upd~~. De plus, les lignes vivantes et mortes sont tracées respectivement par les colonnes ~~n\_live\_tup~~ et ~~n\_dead\_tup~~ (auparavant, il était nécessaire d'utiliser le module ~~contrib/pgstattuple~~ pour avoir le même type d'informations).

Enfin, il reste quelques informations sur les opérations de maintenance. Il est intéressant de savoir quand un ~~VACUUM~~ et un ~~ANALYZE~~ sont exécutés sur une table, surtout pour ceux qui sont exécutés automatiquement via l'autovacuum, car il n'existe aucun autre moyen de le savoir. Pour cela, quatre colonnes précisent la date et l'heure de la dernière exécution d'un ~~VACUUM~~ manuel et automatique, et celle d'un ~~ANALYZE~~ manuel et automatique.

La récupération de ces informations est liée à l'activation du paramètre ~~track\_counts~~. Ce paramètre avait pour nom ~~stats\_row\_level~~ pour les versions antérieures à la 8.3.

Trois autres tables sont disponibles sur le même principe si ce n'est qu'elles fournissent des informations sur les pages disque écrites. Elles se nomment ~~pg\_statio\_all\_tables~~, ~~pg\_statio\_user\_tables~~ et ~~pg\_statio\_sys\_tables~~. Voici un exemple de leur contenu :

```

pagila=# SELECT * FROM pg_statio_all_tables WHERE relname='actor';
-[ RECORD 1 ]-----+-----
relid          | 18024
schemaname     | public
relname        | actor
heap_blks_read | 3

```

```

heap_blks_hit | 1831
idx_blks_read | 6
idx_blks_hit  | 412
toast_blks_read |
toast_blks_hit |
tidx_blks_read |
tidx_blks_hit  |

```

On retrouve les trois mêmes champs de désignation de la table.

~~heap\_blks\_read~~ et ~~heap\_blks\_hit~~ concernent la lecture de la table même. Dans le premier cas, la lecture se fait sur disque, dans le second sur le cache disque. ~~toast\_blks\_read~~ et ~~toast\_blks\_hit~~ donnent les mêmes informations concernant les lectures de la table ~~TOAST~~.

### Note

À chaque table peut être associé une table ~~TOAST~~. Cette table contiendra toutes les valeurs de cellules dont la taille dépasse 2 Ko. Ceci permet un parcours séquentiel plus rapide de la table principale (appelée aussi table heap).

Quatre autres champs donnent le même type d'informations pour l'ensemble des index de la table.

La récupération de ces informations est liée à l'activation du paramètre ~~track\_counts~~. Ce paramètre avait pour nom ~~stats\_block\_level~~ pour les versions antérieures à la 8.3.

Les vues sur les entrées/sorties disque sont particulièrement utiles pour déterminer l'efficacité du cache disque de PostgreSQL. Quand le nombre de lectures disque réelles est bien inférieur au nombre de lectures en cache, c'est bien que le cache satisfait la plupart des demandes de lecture.

### Attention

Ces statistiques ne donnent qu'une image partielle. À cause de la façon dont PostgreSQL gère les entrées/sorties disque, les données qui ne sont pas dans le cache de PostgreSQL peuvent toujours faire partie du cache disque du noyau et pourraient toujours être récupérées sans nécessiter une lecture physique sur disque.

## 2.3 Index

Là aussi, trois catalogues système donnent une vue sur l'activité des index au niveau ligne :

~~pg\_stat\_all\_indexes~~, ~~pg\_stat\_user\_indexes~~ et ~~pg\_stat\_sys\_indexes~~.

```

pagila=# SELECT * FROM pg_stat_user_indexes
pagila=# WHERE indexrelname='film_actor_pkey';
-[ RECORD 1 ]-+-----
reloid      | 18061
indexrelid  | 18229
schemaname  | public
relname     | film_actor
indexrelname | film_actor_pkey
idx_scan    | 2
idx_tup_read | 38
idx_tup_fetch | 0

```

Les cinq premiers champs renseignent sur l'index concerné par les trois champs statistiques qui suivent. ~~reloid~~ est l'OID de la table à laquelle est attachée l'index, ~~indexrelid~~ est l'OID de l'index proprement dit. ~~schemaname~~, ~~relname~~ et ~~indexrelname~~ sont les noms, respectivement, du schéma,

de la table et de l'index.

~~idx\_scan~~ indique le nombre de parcours de cet index. Il est intéressant de vérifier si vous avez des index avec cette colonne à zéro. Dans ce cas, cela sous-entend que l'index n'est pas utilisé, et que sa présence est du coup un point négatif pour les performances, car chaque écriture dans la table associée doit mettre à jour l'index, alors que ce dernier n'est jamais utilisé.

~~idx\_tup\_read~~ indique le nombre de lignes lues dans l'index, et ~~idx\_tup\_fetch~~ indique le nombre de lignes récupérées grâce à l'index.

Les tables ~~pg\_statio\_all\_indexes~~, ~~pg\_statio\_user\_indexes~~ et ~~pg\_statio\_sys\_indexes~~ donnent deux informations intéressantes sur le nombre de pages disque lues sur disque (~~idx\_blks\_read~~) et sur celles récupérées dans le cache disque de PostgreSQL (~~idx\_blks\_hit~~).

```
pagila=# SELECT * FROM pg_statio_all_indexes WHERE indexrelname='actor_pkey';
-[ RECORD 1 ]-----
reloid      | 18024
indexreloid | 18217
schemaname  | public
relname     | actor
indexrelname| actor_pkey
idx_blks_read| 2
idx_blks_hit | 201
```

Évidemment, les champs d'identification de l'index y figurent aussi.

Pour toutes ces statistiques, la configuration indiquée dans le chapitre "Tables" s'applique de la même façon.

## 2.4 Séquences

En ce qui concernent les séquences, il n'existe que les tables d'information au niveau bloc :

~~pg\_statio\_all\_sequences~~, ~~pg\_statio\_user\_sequences~~, ~~pg\_statio\_sys\_sequences~~.

```
pagila=# SELECT * FROM pg_statio_all_sequences LIMIT 1;
-[ RECORD 1 ]-----
reloid      | 18022
schemaname  | public
relname     | actor_actor_id_seq
blks_read   | 0
blks_hit    | 0
```

~~reloid~~ est l'OID de la séquence, ~~schemaname~~ et ~~relname~~ sont respectivement le nom du schéma et celui de l'index.

Les informations statistiques sont les, maintenant très connus, ~~blks\_read~~ et ~~blks\_hit~~.

## 2.5 Autres objets

Trois autres vues donnent une indication de l'activité sur le système. Elles concernent des fonctionnalités très spécifiques de PostgreSQL.

La vue ~~pg\_prepared\_statements~~ permet à un utilisateur de récupérer la liste des requêtes préparées via l'instruction `PREPARE`. Cette instruction permet à PostgreSQL de calculer le plan d'exécution de

la requête et de stocker ce plan. L'utilisateur peut ensuite exécuter cette requête en fournissant si nécessaire les arguments de la requête préparée. L'exécution sera plus rapide, car toute l'étape de calcul du plan d'exécution est déjà réalisée. Cela a peu d'importance sur une requête peu utilisée, mais lorsque cette dernière est exécutée de nombreuses fois, cela fait une différence appréciable. Voici un exemple d'utilisation :

```
pagila=# PREPARE select_un_acteur (integer) AS SELECT * FROM actor WHERE actor_id=$1;
PREPARE
pagila=# \x
Affichage étendu activé.
pagila=# SELECT * FROM pg_prepared_statements;
-[ RECORD 1 ]-----+-----
name           | select_un_acteur
statement      | PREPARE select_un_acteur (integer) AS SELECT * FROM actor
                | WHERE actor_id=$1;
prepare_time   | 2008-04-19 13:43:01.535469+02
parameter_types | {integer}
from_sql       | t
```

Le premier champ indique le nom de la requête préparée, le second l'instruction qui a permis sa préparation, le troisième le moment où elle a été préparée, le quatrième les types des arguments et enfin le dernier si cette requête préparée provient de l'exécution d'une instruction SQL. Comme une instruction préparée appartient seulement à une session, seule cette session la verra. Autrement dit, ne se voient dans cette vue que les requêtes qui ont été préparées au sein de la session qui a demandé l'affichage de la vue.

Dans le même genre, il existe la vue `pg_prepared_xacts`. Cela se passe dans le cadre du Two Phase Commit, dont l'acronyme, 2PC, est souvent plus connu. L'intérêt du Two Phase Commit est de permettre à un processus de préparer une transaction et que cette dernière soit réellement validée par un autre processus. Cette vue affiche toutes les transactions préparées sur une base de données, quelle que soit la session, quel que soit l'utilisateur connecté.

```
pagila=# BEGIN;
BEGIN
pagila=# UPDATE actor SET last_name = upper (last_name);
UPDATE 200
pagila=# PREPARE TRANSACTION 'yop';
PREPARE TRANSACTION
pagila=# SELECT * FROM pg_prepared_xacts;
-[ RECORD 1 ]-----+-----
transaction | 1816
gid         | yop
prepared   | 2008-04-19 14:06:57.121776+02
owner      | guillaume
database   | pagila
```

Le champ `transaction` précise l'identifiant de la transaction concernée, alors que le champ `gid` indique son nom (déclaré avec l'instruction `PREPARE TRANSACTION`). `prepared` est l'horodatage, `owner` le propriétaire de la transaction préparée et `database` la base de données concernée par la transaction.

La dernière vue, `pg_cursors`, liste les curseurs disponibles. Quand le curseur a une durée de vie limitée à la transaction qui a permis sa création, sa visibilité n'est possible qu'au sein de cette transaction.

```
pagila=# BEGIN;
```

```

BEGIN
pagila=# DECLARE x CURSOR FOR SELECT * FROM actor;
DECLARE CURSOR
pagila=# SELECT * FROM pg_cursors;
-[ RECORD 1 ]-+-----
name          | x
statement     | DECLARE x CURSOR FOR SELECT * FROM actor;
is_holdable   | f
is_binary     | f
is_scrollable | t
creation_time | 2008-04-19 14:18:09.348622+02

```

~~name~~ est le nom du curseur (indiqué lors de la création du curseur), ~~statement~~ est la requête SQL qui a créé le curseur, ~~creation\_time~~ est un horodatage précisant le moment où le curseur a été créé. ~~is\_holdable~~ précise si le curseur survit à la fin de la transaction qui l'a créé. ~~is\_binary~~ indique si le curseur est binaire, ~~is\_scrollable~~ précise si les lignes peuvent être récupérées aléatoirement.

### 3. Petit récapitulatif sur la configuration

Certaines de ces statistiques nécessitent une configuration particulière du serveur PostgreSQL. Voici un tableau récapitulatif :

De ce tableau, on peut déduire que la version 8.3 ne peut pas désactiver le collecteur de statistiques. En effet, l'impact sur les performances a été tellement minimisé que les développeurs ont décidé de ne plus permettre la désactivation du collecteur.

### 4. Une utilisation intéressante des vues système

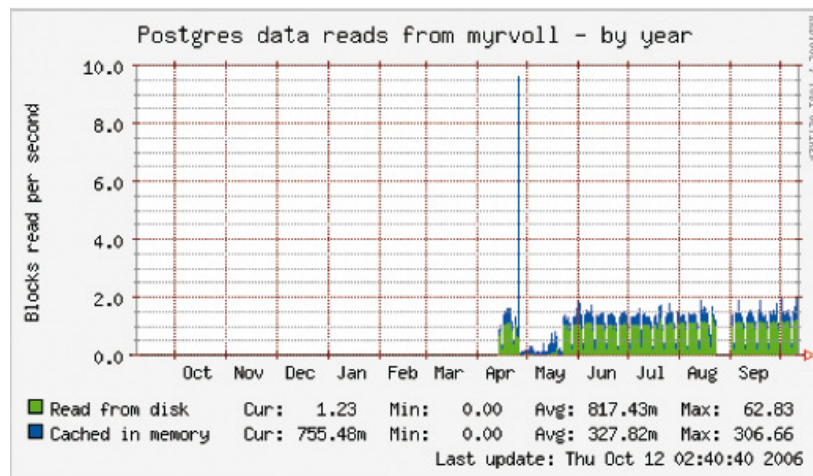
Avec toutes ces informations, il devient très simple de créer des scripts de surveillance d'une base de données PostgreSQL. Le but du script peut permettre d'alerter un administrateur si un point critique survient (par exemple, un grand nombre de transactions annulées) ou de créer des graphes indiquant l'évolution de la situation. L'outil ~~munin~~ est tout à fait indiqué pour ce dernier cas. Des plugins ont été écrits par la société Dalibo pour tracer un grand nombre de cas et vous en trouverez d'autres sur le site MuninExchange (<http://muninexchange.projects.linpro.no/>). Voici quelques exemples de plugins existants :

- ratio lecture sur disque/lecture en cache ;
- ratio des transactions validées/transactions annulées ;
- nombre total de connexions ;
- nombre de connexions par bases de données ;

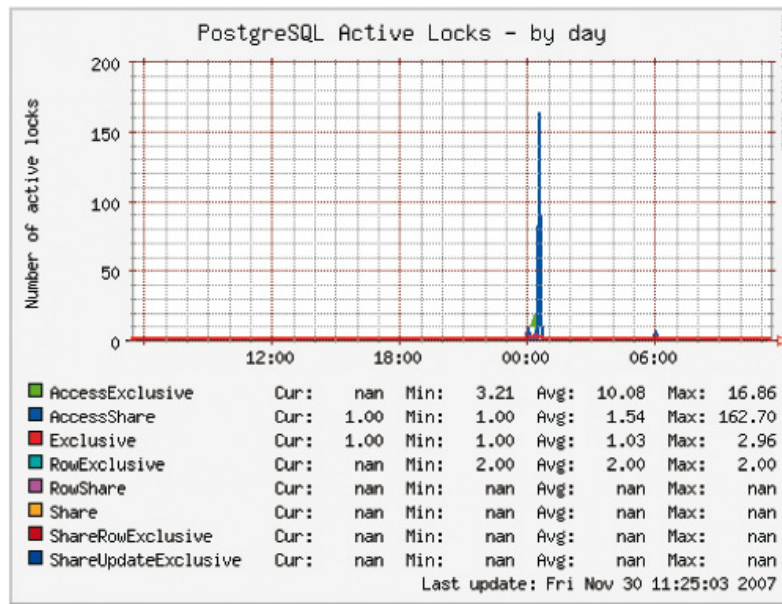
Version	Paramètre	Par défaut	Commentaires
8.3	<code>track_activities</code>	on	Si activé, met à jour les colonnes <code>current_query</code> , <code>waiting</code> et <code>query_start</code> de la vue système <code>pg_stat_activity</code> .
	<code>track_counts</code>	on	Si activé, met à jour les colonnes des vues système <code>pg_stat_all_tables</code> et <code>pg_statio_all_tables</code> , ainsi que celles concernant les index et les séquences.
	<code>update_process_title</code>	on	Si activé, met à jour le titre du processus pour que <code>ps</code> affiche l'activité du processus PostgreSQL.
8.2	<code>stats_command_string</code>	on	Si activé, met à jour la colonne <code>current_query</code> de la vue système <code>pg_stat_activity</code> .
	<code>update_process_title</code>	on	Si activé, met à jour le titre du processus pour que <code>ps</code> affiche l'activité du processus PostgreSQL.
	<code>stats_start_collector</code>	on	Si activé, lance le processus d'enregistrement des statistiques ( <code>stats collector process</code> en sortie de la commande <code>ps</code> ).
	<code>stats_block_level</code>	off	Équivalent de <code>track_count</code> en 8.3 mais ne met à jour que les colonnes des vues système <code>pg_statio_all_tables</code> , ainsi que celles concernant les index et les séquences.
	<code>stats_row_level</code>	off	Équivalent de <code>track_count</code> en 8.3 mais ne met à jour que les colonnes des vues système <code>pg_stat_all_tables</code> , ainsi que celles concernant les index.
	<code>stats_reset_on_server_start</code>	off	Si activé, réinitialise les statistiques au démarrage du serveur PostgreSQL.
8.1, 8.0 et 7.4	<code>stats_command_string</code>	off	Voir description en 8.2
	<code>stats_start_collector</code>	on	Voir description en 8.2
	<code>stats_block_level</code>	off	Voir description en 8.2
	<code>stats_row_level</code>	off	Voir description en 8.2
	<code>stats_reset_on_server_start</code>	on	Voir description en 8.2

- place disque occupée par tout le serveur ;
- place disque occupée par chaque base de données ;
- évolution du nombre d'insertions/modifications/suppressions ;
- etc.

Voici, par exemple, le graphe du nombre de lectures sur disque combiné avec celui du nombre de lectures provenant du cache :

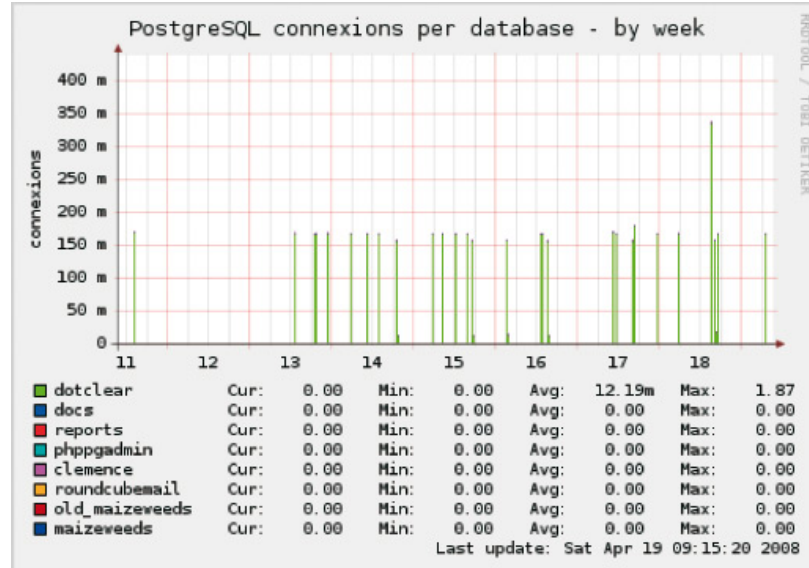


Il est aussi possible de consulter les verrous créés dans le temps :



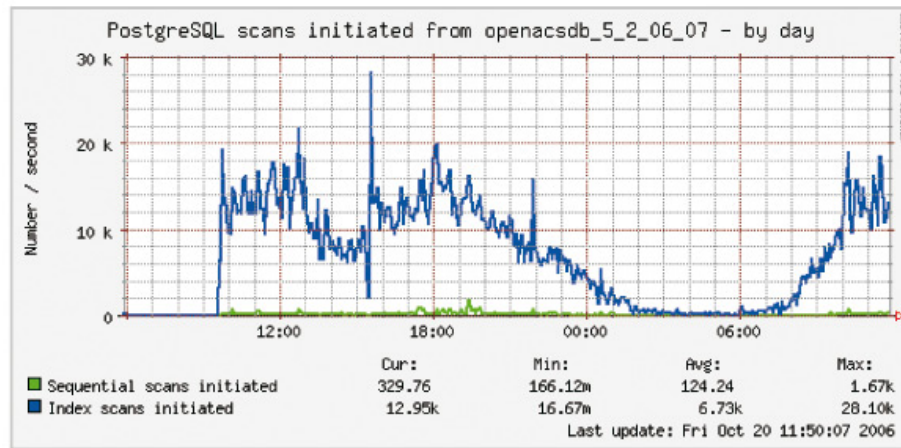
On remarque ici que le nombre de verrous exclusifs créés est très bas par rapport au nombre de verrous partagés (comme **AccessShare**), ce qui est bon signe pour la réactivité du moteur.

Sur ce nouveau graphe, on aperçoit le nombre de connexions par bases de données :



Enfin, ce dernier graphe montre le nombre de lignes récupérées suite à un parcours séquentiel et suite à un parcours d'index :





Le nombre de parcours séquentiels est bien inférieur à celui des parcours d'index, ce qui est un plus évident pour les performances globales du système.

Un projet est en cours de création sur pgFoundry depuis la réécriture de certains plugins en Perl. Il est prévu de mettre à jour les plugins et de tirer profit des informations statistiques proposées par les dernières versions de PostgreSQL. N'hésitez pas à aller consulter l'avancée de ce nouveau projet sur <http://pgfoundry.org/projects/muninpgplugins>.

Évidemment, ce n'est pas la seule utilisation possible de ces catalogues système. D'autres outils en tirent profit. Par exemple, voici une copie d'écran de la fenêtre "État du serveur" de pgAdmin :

PID	Base de données	Utilisateur	Client	Démarrage du client	Détail
9817	template1	guillaume	127.0.0.1:41984	2008-04-19 12:34:03...	

L'onglet Statut passe par la vue `pg_stat_activity`, l'onglet Verrous par `pg_locks`. Quant aux informations spécifiques aux objets, elles sont généralement disponibles dans l'onglet Statistiques une fois l'objet voulu sélectionné dans le navigateur.

phpPgAdmin utilise les catalogues système de la même façon.

## Conclusion

Les informations renvoyées par les catalogues système de statistiques sont essentiels pour mesurer en temps réel la bonne santé du système et l'adéquation de la configuration par rapport à l'activité des bases et au matériel utilisé.

La version 8.3 apporte de nombreuses informations supplémentaires facilitant la vie de l'administrateur de bases de données tout en améliorant les performances du collecteur de statistiques.

## Référence

Chapitre 26, Surveiller les activités de la base de données : <http://docs.postgresqlfr.org/8.3/monitoring.html>

Retrouvez cet article dans : [Linux Magazine 106](#)

## Laissez une réponse

Vous devez avoir ouvert une [session](#) pour écrire un commentaire.

[Identifiez-vous](#)

[Inscription](#)

[S'abonner à UNIX Garden](#)

Catégories

[Administration réseau](#)

[Administration système](#)

[Agenda-Interview](#)

[Audio-vidéo](#)

[Bureautique](#)

[Comprendre](#)

[Distribution](#)

[Embarqué](#)

[Environnement de bureau](#)

[Graphisme](#)

[Jeux](#)

[Matériel](#)

[News](#)

[Programmation](#)

[Réfléchir](#)

[Sécurité](#)

[Utilitaires](#)

[Web](#)

## • Pages

- - [A propos](#)
  - [Nuage de Tags](#)
  - [Contribuer](#)

## • Archives

- - [avril 2011](#)
  - [mars 2011](#)
  - [février 2011](#)
  - [janvier 2011](#)
  - [décembre 2010](#)
  - [novembre 2010](#)
  - [octobre 2010](#)
  - [septembre 2010](#)

- [août 2010](#)
- [juillet 2010](#)
- [juin 2010](#)
- [mai 2010](#)
- [avril 2010](#)
- [mars 2010](#)
- [février 2010](#)
- [janvier 2010](#)
- [décembre 2009](#)
- [novembre 2009](#)
- [octobre 2009](#)
- [septembre 2009](#)
- [août 2009](#)
- [juillet 2009](#)
- [juin 2009](#)
- [mai 2009](#)
- [avril 2009](#)
- [mars 2009](#)
- [février 2009](#)
- [janvier 2009](#)
- [décembre 2008](#)
- [novembre 2008](#)
- [octobre 2008](#)
- [septembre 2008](#)
- [août 2008](#)
- [juillet 2008](#)
- [juin 2008](#)
- [mai 2008](#)
- [avril 2008](#)
- [mars 2008](#)
- [février 2008](#)
- [janvier 2008](#)
- [décembre 2007](#)
- [novembre 2007](#)
- [février 2007](#)

[GNU/Linux Magazine](#)

[GNU/Linux Magazine HS](#)

[Linux Pratique](#)

[Linux Pratique HS](#)

[Linux Pratique Essentiel](#)

[Linux Pratique Essentiel HS](#)

[Misc](#)

[Misc HS](#)

## • Articles secondaires

- 15/3/2009

[Smart Middle Click 0.5.1 : ouvrez les liens JavaScript dans des onglets](#)

Tout d'abord, un petit raccourci utile : quand vous voulez ouvrir un lien dans un onglet, plutôt que d'utiliser le menu contextuel, cliquez simplement dessus avec le bouton du milieu. Hop, c'est ouvert ! C'est simple et diablement efficace, parfois un peu trop.....

[Voir l'article...](#)

30/10/2008

[Google Gears : les services de Google offline](#)

Lancé à l'occasion du Google Developer Day 2007 (le 31 mai dernier), Google Gears est une extension open source pour Firefox et Internet Explorer permettant de continuer à accéder à des services et applications Google, même si l'on est déconnecté....

[Voir l'article...](#)

7/8/2008

[Trois questions à...](#)

Alexis Nikichine, développeur chez IDM, la société qui a conçu l'interface et le moteur de recherche de l'EHM....

[Voir l'article...](#)

11/7/2008

[Protéger une page avec un mot de passe](#)

En général, le problème n'est pas de protéger une page, mais de protéger le répertoire qui la contient. Avec Apache, vous pouvez mettre un fichier `.htaccess` dans le répertoire à protéger....

[Voir l'article...](#)

6/7/2008

[hypermail : Conversion mbox vers HTML](#)

Comment conserver tous vos échanges de mails, ou du moins, tous vos mails reçus depuis des années ? mbox, maildir, texte... les formats ne manquent pas. ...

[Voir l'article...](#)

6/7/2008

[iozone3 : Benchmark de disque](#)

En fonction de l'utilisation de votre système, et dans bien des cas, les performances des disques et des systèmes de fichiers sont très importantes....

[Voir l'article...](#)

Les Éditions Diamond - Tous droits réservés.