Ideo-Lab

POSTGRESQL OPTIMIZATION & SETTINGS

Guillaume oneill

2th October 2025

Version 2.1

TUNING PLAN

TUNING AGENDA

- 1) Collecte d’entrées & cadrage
- 2) Stratégie d’architecture
- 3) Paramétrage PostgreSQL (formules & profils)
- 4) Réglages OS & FS
- 5) Modélisation & paramètres par objets
- 6) Journaux, traçage & observabilité
- 7) Recettes par profil
- 8) Exemple de gabarit postgresql.conf
- 9) Runbook de mise en prod (pas-à-pas)
- 10) Check-list rapide selon entrées fournies
- 11) Pièges courants & bonnes pratiques
- 12) Option “formulaire d’entrée → sortie calculée”

1) Collecte d'entrées & cadrage

1.1. Inventaire matériel & OS

- **RAM totale**, **CPU** (cœurs/threads, freq, NUMA on/off), **stockage** (NVMe/SAS/SATA, RAID, EBS), **réseau** (latence, MTU).
- **Type**: bare-metal / VM / cloud, présence de **bruit de noisy neighbors**, **crédits CPU** (cloud).
- **OS**: distribution & version (Ubuntu/Debian/RHEL/Alma), kernel (`uname -r`), THP, HugePages.
- **FS**: ext4/XFS, options de montage (barrier, noatime, discard), scheduler I/O (mq-deadline, none).

1.2. Profil applicatif & charge

- **OLTP** (petites transactions, forte concurrence), **OLAP** (scans volumineux), **mixte**.
- **Taille jeu de données** active & totale, **ratio data vs RAM**.
- **I/O**: aléatoire vs séquentiel, write-intensive vs read-heavy, latence cible.
- **Pics**: nombre de connexions simultanées, requêtes/s, batchs/crons, ETL.
- **Exigences**: RPO/RTO (sauvegardes/HA), contraintes **durabilité/latence** (synchronous_commit).

2) Stratégie d'architecture

2.1. Connexions & pooling

- **PgBouncer** en mode `transaction` (OLTP) ou `session` (OLAP outillage) pour limiter `max_connections` (ex: 50–200) et éviter l'explosion de `work_mem`.
- **max_connections** bas côté PostgreSQL (évite la contention mémoire & LWLocks), **scaling via pool**.

2.2. Layout stockage

- Séparer (si possible) : **DATA**, **WAL**, **temp**. NVMe pour WAL/temp si write-heavy.
- RAID10 > RAID5 pour latence. Désactiver cache contrôleur incohérent; BBU/FBWC si contrôleur.
- XFS (généralement très bon par défaut) ou ext4; `noatime`, **alignment** correct, stripe size adapté RAID.

2.3. HA, sauvegardes, observabilité

- **Sauvegardes**: base + WAL (pgBackRest/Barman). Chiffrage & vérif de restauration.
- **Réplication**: streaming async (latence), sync (durabilité). **Slots** logiques/physiques dimensionnés.
- **Monitoring**: pg_stat_statements, pg_stat_io (>=15), views *pg_stat_*, exporter Prometheus, traces EXPLAIN.

3) Paramétrage PostgreSQL (formules & profils)

> Notation :
> **RAM** = mémoire totale serveur.
> **RAM_PG** = budget mémoire pour PostgreSQL (typiquement 60–75% de RAM si serveur dédié).
> Les valeurs ci-dessous sont **points de départ** à valider par tests.

3.1. Mémoire principale

- `shared_buffers`
  - OLTP: **RAM_PG × 0,25** (jusqu'à 8–16 Go raisonnable)
  - OLAP/mixte: **RAM_PG × 0,15–0,25** (éviter trop haut si dataset >> RAM)
- `effective_cache_size`
  - ≈ **RAM_PG × 0,75–0,8** (estimation du cache FS + shared_buffers)
- `work_mem` *(par trie/hash, par worker)*
  - Point de départ: **(RAM_PG × 0,05) / max_active_ops**, où `max_active_ops` ≈ connexions simultanées × 1,5 (OLTP) ou nb. workers OLAP.
  - Ex: RAM_PG=32 Go, 100 ops actives ⇒ (32×0,05)/150 ≈ **10 MB**. Ajuster après mesure des `temporary file` dans logs.
- `maintenance_work_mem`
  - 512 MB à **2 GB** selon RAM (vacuum/reindex/CREATE INDEX). Ne pas déraisonnablement gonfler.
- `wal_buffers`
  - `-1` (auto-tune) ou **64–256 MB** si write-heavy constant.

3.2. WAL, checkpoints, durabilité

- `wal_level` = `replica` (par défaut), `logical` si CDC nécessaire.
- `synchronous_commit`
  - OLTP faible latence: `off` (accepte perte <1 tx), ou `local` / `remote_write` selon RPO.
  - OLAP: souvent `on`.
- `max_wal_size` & `min_wal_size`
  - Point de départ: **min_wal_size=1–4 GB**, **max_wal_size=8–32 GB** (NVMe → plus haut). Ajuster pour raréfier checkpoints.
- `checkpoint_timeout` = 15–30 min (write-heavy → vers 15–20 min).
- `checkpoint_completion_target` = **0,9** (étale les I/O).
- `full_page_writes` = `on` (production) sauf FS/journal garantis (rarement off).

3.3. Background writer & autovacuum

- `bgwriter_*` (surtout ≤14): conserver défauts récents ou augmenter `bgwriter_lru_maxpages` (ex 1000) si churn.
- `autovacuum` = `on` (toujours).
  - `autovacuum_max_workers` = 3–10 (RAM/IO).
  - `autovacuum_naptime` = 10–30s.
  - `autovacuum_vacuum_scale_factor` = **0,05** (ou plus bas **0,01–0,02** pour grosses tables).
  - `autovacuum_analyze_scale_factor` = **0,02–0,05**.
  - Fixer **per-table** via `ALTER TABLE ... SET (autovacuum_vacuum_scale_factor=...)` pour contenance.
  - `autovacuum_vacuum_cost_limit` / `delay` selon I/O.
- **Freeze**: vérifier `age(datfrozenxid)`, planifier `VACUUM (FREEZE)` hors-pics si nécessaire.

3.4. Planner & parallélisme

- `random_page_cost` :
  - NVMe: **1,1–1,5** ; RAID10 SSD: **1,5–2,5** ; HDD: **3–4**.
- `seq_page_cost` : **1,0** (standard).
- `effective_io_concurrency` : SSD/NVMe **64–256** ; HDD **1–4**.
- Parallélisme (>=9.6):
  - `max_parallel_workers_per_gather` = 2–4 (OLAP) ; OLTP → 1–2.
  - `max_parallel_workers` = nb cœurs/2 (plafonné).
  - `parallel_leader_participation` = on (défaut).
- JIT (>=11):
  - OLTP: souvent **off** (latence).
  - OLAP: **on**, avec `jit_above_cost` élevé et `jit_inline_above_cost` pour cibler gros plans.

3.5. Temp & tri

- Déplacer `temp_tablespaces` sur volume rapide si disponible.
- Surveiller `log_temp_files` (ex `0` en pré-prod, `10MB` en prod) pour calibrer `work_mem`.

3.6. Verrouillage, timeouts & robustesse

- `idle_in_transaction_session_timeout` (ex 60–300 s) pour tuer sessions bloquantes.
- `lock_timeout` / `statement_timeout` par rôle/app (OLTP: 1–5 s sur requêtes interactives).
- `max_locks_per_transaction` si nombreux objets (migrations massives).

4) Réglages OS & FS

4.1. Kernel & mémoire

- **THP**: désactiver (Transparent Huge Pages) → latence.
- **HugePages** classiques: **oui** si gros `shared_buffers` (réduire overhead TLB).
- `vm.swappiness` = **1–10** (éviter swap), `vm.dirty_ratio` / `vm.dirty_background_ratio` prudents sur HDD; sur SSD laisser défauts récents.
- `vm.overcommit_memory` = 0/1 (tester avec workload).
- `kernel.sched_migration_cost_ns` (par défaut OK sur kernels récents).

4.2. I/O & filesystems

- Scheduler I/O: `none` / `mq-deadline` (SSD/NVMe).
- `noatime` sur FS, activer **discard** si SSD (ou TRIM périodique).
- Aligner **stripe** avec RAID; taille bloc 4k par défaut OK.

4.3. Réseau

- MTU 9001 (si infra end-to-end), `tcp_fin_timeout` standard, `somaxconn` augmenté (poolers), `net.core.rmem_max` / `wmem_max` si réplication distante très rapide.

5) Modélisation & paramètres par objets

5.1. Tables & index

- **Fillfactor**: tables à UPDATE fréquents → **90–95** ; index sensibles aux updates → **90–95** pour limiter bloat.
- **Partitionnement**: par date/range/hash si tables > 100 M lignes ou maintenance lourde.
- **Index**: éviter doublons, index partiels (`WHERE active`), multicolonnes (ordre selon sélectivité), BRIN pour gros scans ordonnés.
- **TOAST**: limiter champs très larges, compresser au client si pertinent.

5.2. Statistiques & planificateur

- `ALTER TABLE ... ALTER COLUMN ... SET STATISTICS N` (50–200) sur colonnes très corrélées/distribuées.
- `default_statistics_target` global: 100–200 (OLAP), 100 (mixte), 50–100 (OLTP).

5.3. Paramètres par rôle/session

- `SET work_mem` plus haut pour jobs ETL/analyses (via rôles ou `SET LOCAL`).
- `search_path` maîtrisé; `statement_timeout` par rôle app.

6) Journaux, traçage & observabilité

- `logging_collector=on`, `log_line_prefix` riche (time, pid, user, db, app, client).
- `log_statement` = `none` (prod), mais `log_min_duration_statement` (ex 200–500 ms OLTP).
- `log_checkpoints=on`, `log_autovacuum_min_duration` (ex 1000 ms), `log_lock_waits=on`.
- **Extensions**: `pg_stat_statements`, `pg_wait_sampling` (si dispo), `auto_explain` (en QA), `pg_stat_io` (PG15+) pour I/O.
- Dashboards: latence P50/P95/P99, TPS, temp files, cache hit, bloat, autovacuum activity, WAL write rate.

7) Recettes par profil

7.1. OLTP (latence & concurrence)

- **PgBouncer obligatoire**, `max_connections` PostgreSQL ≤ **200**.
- `shared_buffers` 25% RAM_PG (8–16 Go plafond), `effective_cache_size` 75–80% RAM_PG.
- `work_mem` modéré (4–16 MB), `maintenance_work_mem` 512 MB–1 GB.
- `synchronous_commit=off` (si acceptable) ou `local`.
- `max_wal_size` 8–16 GB ; `checkpoint_timeout` 15–20 min ; `completion_target=0,9`.
- Planner: `random_page_cost` bas (SSD), `effective_io_concurrency` 64–128.
- JIT off. `statement_timeout` 1–3 s. Autovacuum agressif par table.

7.2. OLAP (débit & scans)

- Connexions limitées mais lourdes; parallélisme **on** (2–4 workers/gather).
- `work_mem` élevé **par job** (64–512 MB), réglé **par session**.
- `shared_buffers` 15–20% RAM_PG (cache OS utile pour gros scans).
- `synchronous_commit=on`, `wal_level=replica`, `max_wal_size` élevé (16–64 GB).
- JIT **on** avec seuils élevés. `random_page_cost` bas sur SSD.
- Tables partitionnées, index BRIN, `effective_io_concurrency` haut.

7.3. Mixte

- Séparer **pools** (PgBouncer) et **rôles** avec `work_mem`/timeouts distincts.
- `shared_buffers` 20–25%, `max_wal_size` moyen (16–32 GB), `checkpoint` 20–30 min.
- Parallélisme modéré (1–2). JIT ciblé.

9) Runbook de mise en prod (pas-à-pas)

1. **Mesures de base (avant):** TPS, latences, hit cache, WAL/s, temp files, bloat, top N requêtes (pg_stat_statements).
2. **Appliquer réglages OS** (THP off, HugePages si needed, FS options, scheduler). Redémarrage si nécessaire.
3. **Configurer PgBouncer**, réduire `max_connections` PostgreSQL.
4. **Ajuster mémoire & WAL** selon formules (section 3).
5. **Calibrer autovacuum par table** (grosse volumétrie → scale_factor bas).
6. **Déployer observabilité** (exporter + dashboards + alertes).
7. **Tests de charge** (pgbench & charges réelles) :
   - Varier `work_mem`, `max_wal_size`, `random_page_cost`, parallélisme.
   - Observer P95/P99 latence, temp files, I/O wait.
8. **Boucle d'itération**: changer **un paramètre à la fois**, comparer 30–60 min de trafic.
9. **Plan de maintenance**: `VACUUM (VERBOSE, ANALYZE)` périodique, reindex ciblé, `pg_repack` si besoin, contrôle freeze.
10. **Revue trimestrielle**: top requêtes lentes, nouveaux index/partitions, croissance des tables, paramètres planner.

10) Check-list rapide selon entrées fournies

> À remplir lors du cadrage pour calculer les valeurs initiales.

- **RAM**: ___ Go → RAM_PG = ___ Go → `shared_buffers` = ___ Go ; `effective_cache_size` = ___ Go.
- **Type serveur**: bare-metal / VM / cloud → scheduler I/O ___, NUMA policy ___, credits CPU ___.
- **OS**: version ___ ; THP off ? HugePages ? FS ___ with `noatime` ?
- **Profil I/O/app**: OLTP / OLAP / mixte ; dataset actif ___ Go ; connexions simultanées ___ ;
  - `max_connections` (PG) ___ ; `work_mem` cible ___ MB ; `maintenance_work_mem` ___ MB.
  - WAL: `max_wal_size` ___ GB ; `checkpoint_timeout` ___ ; `sync_commit` ___.
  - Planner: `random_page_cost` ___ ; `effective_io_concurrency` ___ ; parallélisme ___ ; JIT ___.
- **Autovacuum par grosses tables**: scale_factor (vacuum/analyze) = ___ / ___ ; naptime ___ s ; workers ___.
- **Observabilité**: `pg_stat_statements` on ; `log_min_duration_statement` ___ ms ; dashboards OK.

---

11) Pièges courants & bonnes pratiques

- **Trop de connexions** sans pooler → explosion mémoire et context switch.
- `work_mem` trop haut **globalement** → temp files ou OOM quand charge monte.
- `shared_buffers` gigantesque **sans HugePages** → fragmentation & overhead.
- `random_page_cost` mal réglé → plans sous-optimaux (index vs seq scan).
- Autovacuum trop timide sur grosses tables → **bloat**, plans pessimistes.
- **Pas de tests** avant/après → tuning “aveugle”.
- **Sauvegardes non testées**: RPO/RTO théoriques. Toujours tester la **restauration**.

12) Option “formulaire d’entrée → sortie calculée”

À partir des 4 entrées (RAM, type serveur, version OS, profil I/O & app), produire automatiquement :

- Un **jeu de paramètres** (`postgresql.conf` + `pg_hba.conf` indicatif)
- Une **check-list OS/FS**
- Des **overrides par table** (autovacuum + statistics target)
- Des **reco requêtes** (timeouts, pooler)
- Un **plan de test** (pgbench + scénarios réels)
- Un **tableau des KPI** à suivre (seuils & alertes)

EXEMPLE DE SETTINGS

```
# Connections & memory
max_connections = 150
shared_buffers = 8GB
effective_cache_size = 24GB
work_mem = 8MB
maintenance_work_mem = 1GB
wal_buffers = -1

# WAL & checkpoints
wal_level = replica
synchronous_commit = local
full_page_writes = on
checkpoint_timeout = 20min
checkpoint_completion_target = 0.9
min_wal_size = 2GB
max_wal_size = 16GB

# Planner & I/O (SSD/NVMe)
random_page_cost = 1.2
effective_io_concurrency = 128
max_parallel_workers_per_gather = 2
max_parallel_workers = 8
parallel_leader_participation = on
jit = off
```

```
# Autovacuum
autovacuum = on
autovacuum_max_workers = 6
autovacuum_naptime = 15s
autovacuum_vacuum_scale_factor = 0.02
autovacuum_analyze_scale_factor = 0.02
log_autovacuum_min_duration = 1000

# Logging
logging_collector = on
log_line_prefix = '%m [%p] %u@%d %a %h '
log_min_duration_statement = 300
log_checkpoints = on
log_lock_waits = on
log_temp_files = 10MB

# Timeouts
statement_timeout = 3000
idle_in_transaction_session_timeout = 120000
```

POSTGRESQL SETTINGS

MAIN SERVER SETTINGS

- **shared_buffers** = 45% of total DRAM
- temp_buffers = 64 to **256** MB
- **work_mem** = **256** MB
- maintenance_work_mem = 4GB
- **fsync** = off
- commit_delay = 1000 (micro second)
- checkpoint_timeout = 10min (*to be tested*)
- **default_statistics_target** = 1000

- ***show shared_buffers;***
- ***alter system set shared_buffers = '32GB';***

MAIN SERVER SETTINGS

- **random_page_cost** = **1.1** (*Index scan*)
- **seq_page_cost** = **0.5**
- **effective_cache_size** = **40GB** or *75% of memory*
- **synchronous_commit** = off
- **commit_siblings** = 10 – 20
- **checkpoint_completion_target** = **0.9**
- **checkpoint_timeout** = ‘3min’ (*to be tested)*

WHEN DYNAMIC ALTER OF A SETTINGS IT’S LOCATED HERE :

/var/lib/postgresql/16/main
postgresql.auto.conf

ERROR LOG IS HERE :
/VAR/LOG/POSTGRESQL
RELOAD SETTINGS : SELECT pg_reload_conf();

WAL & PARTITION Settings

- wal_sync_method = *open_datasync*
- wal_writer_delay = 500ms
- full_page_writes = off (*For Batch Process*)
- wal_buffers = 64 to 256MB
- max_wal_size = 10GB

- effective_io_concurrency = **200** (*for high speed ssd*)
- **max_parallel_workers** = **16**
- max_parallel_maintenance_workers = 4
- **max_parallel_workers_per_gather** = **4**
- enable_partitionwise_aggregate = 'on'
- enable_partition_pruning = 'on'
- enable_partitionwise_join = 'on'
- autovacuum_max_workers = 5
- plan_cache_mode = auto
- parallel_tuple_cost/parallel_setup_cost **= 0.05 / 1000**

WAL SETTINGS - extended

- max_wal_size = 10GB
- min_wal_size = 1GB
- archive_mode = ‘off’
- archive_command = ‘’

- *If noreplica process we can increase this value to avoid frequent checkpoints which are slowing the Batch process!*
- *No Archive activated !!*

Index Monitoring

- pg_stat_user_indexes()
- pg_stat_activity()
- pg_relation_size()
- idx_scan
- EXPLAIN_ANALYZE
- **REINDEX** DATABASE kocliko;
- **analyze**;
- CREATE EXTENSION pg_stat_statements;
- ***SELECT * FROM pg_stat_statements ORDER BY***
- ***total_time DESC LIMIT 10;***

VACUUM SETTINGS

- autovacuum_vacuum_scale_factor = 0.05*;*
- autovacuum_analyze_scale_factor = 0.02;
- autovacuum_max_workers = 5;
- autovacuum_vacuum_cost_limit = 2000
- autovacuum_vacuum_scale_factor = 0.05
- *vacuum_cost_limit = 2000*
- *vacuum_cost_delay = 20ms*

WRITES OPTIMIZATION

✅ Recommandations de réglages :

- Si tu as **beaucoup de RAM** et une charge modérée d'écriture :

```postgresql
checkpoint_timeout = 15min
max_wal_size = 4GB
checkpoint_completion_target = 0.9
```

- Si tu as une forte activité d'écriture (nombreux updates fréquents) :

```postgresql
checkpoint_timeout = 5min
max_wal_size = 2GB
checkpoint_completion_target = 0.9
```

SQL PRE WARMING

- CREATE EXTENSION **pg_prewarm** ;
- SELECT pg_prewarm('xxxxxxxxx')

- *EXAMPLE :*
  - select pg_**prewarm**('api_accomodationTemperature');
  - select pg_**prewarm**('api_accomodationConsumption');
  - select pg_**prewarm**('ai_apiDataSensor');

  - return the Number of 8K pages allocated in « shared_buffers »

OPTIMIZE YOUR SQL QUERIES

- **Sous-requêtes IN/EXISTS** : Préférez l'utilisation de `EXISTS` plutôt que `IN` dans les sous-requêtes lorsque vous travaillez avec des tables volumineuses.
  - Exemple :

sql — Copier le code

```sql
SELECT *
FROM table_a
WHERE EXISTS (
    SELECT 1
    FROM table_b
    WHERE table_b.id = table_a.id
);
```

OPTIMIZE YOUR SQL QUERIES

9. Plan d'exécution (EXPLAIN)

- Toujours vérifier le plan d'exécution avec `EXPLAIN` ou `EXPLAIN ANALYZE` pour comprendre comment PostgreSQL exécute une requête. Vous pouvez identifier les problèmes comme les scans séquentiels inutiles ou les mauvaises estimations de coûts.

sql Copier le code

```sql
EXPLAIN ANALYZE SELECT * FROM table WHERE condition;
```

OPTIMIZE YOUR SQL QUERIES

4. Optimisation des jointures

- **Évitez les jointures inutiles** : Rejoindre des tables qui ne sont pas nécessaires dans une requête peut ralentir le processus. Analysez toujours les plans d'exécution des requêtes.
- **Favorisez les jointures sur des colonnes indexées** : Les jointures entre des colonnes non indexées peuvent entraîner des scans séquentiels coûteux.

5. Évitez les full table scans (scan séquentiels)

- Utilisez des index pour éviter que PostgreSQL scanne toute la table lorsqu'une petite partie des données est demandée.
- Utilisez les clauses `LIMIT` et `OFFSET` avec parcimonie, car elles peuvent parfois entraîner un scan complet de la table.

OPTIMIZE YOUR SQL QUERIES

4. Optimisation des jointures

Les jointures mal configurées sur de grandes tables peuvent provoquer des scans coûteux.

Recommandations :

- **Utilisez des index sur les colonnes utilisées dans les jointures.** Cela permet d'accélérer la correspondance des lignes dans les tables jointes.
  - **Exemple :**

```sql
CREATE INDEX idx_customer_id ON orders (customer_id);
```

- **Évitez les jointures non nécessaires :** Limitez les jointures aux tables strictement nécessaires pour votre requête.

Comparaison IN vs EXISTS :

- `EXISTS` est généralement plus rapide que `IN` pour les sous-requêtes car PostgreSQL arrête la recherche dès qu'il trouve une correspondance, tandis que `IN` doit parcourir toutes les valeurs.

OPTIMIZE YOUR SQL QUERIES

5. Plan d'exécution (EXPLAIN et EXPLAIN ANALYZE)

L'outil **EXPLAIN** vous montre comment PostgreSQL planifie d'exécuter une requête. C'est fondamental pour comprendre pourquoi une requête est lente et quels ajustements sont nécessaires.

- **EXPLAIN** vous donne un aperçu théorique du plan d'exécution.
- **EXPLAIN ANALYZE** exécute réellement la requête et affiche le temps réel pris par chaque étape.
    - **Exemple :**

```sql
EXPLAIN ANALYZE SELECT * FROM orders WHERE customer_id = 123;
```

Vous pouvez voir si PostgreSQL utilise un scan séquentiel ou un index scan. Si vous voyez un **Seq Scan** (scan séquentiel), cela signifie que PostgreSQL parcourt chaque ligne de la table, ce qui est lent pour les grandes tables. Un **Index Scan** serait beaucoup plus rapide.

Tuning guidelines

- https://www.postgresql.org/docs/current/performance-tips.html
- https://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server
- https://www.postgresql.org/docs/16/populate.html#POPULATE-MAX-WAL-SIZE
- https://amperecomputing.com/tuning-guides/postgreSQL-tuning-guide
- https://medium.com/@talhakhalid101/tuning-tips-to-maximize-your-postgresql-performance-2a78996ee666
- https://www.postgresql.org/docs/current/runtime-config-wal.html#GUC-COMMIT-SIBLINGS
- https://www.cybertec-postgresql.com/en/bulk-load-performance-in-postgresql/
- https://blog.gitguardian.com/10-tips-to-optimize-postgresql-queries-in-your-django-project/

BEST QUERY PLANNER SETTINGS

General considerations

Chaque environnement est unique, donc il est important de surveiller régulièrement les performances des requêtes à l'aide d'outils comme `EXPLAIN ANALYZE`. Vous pouvez ajuster les paramètres ci-dessus progressivement, tester leurs impacts, et utiliser les retours du planificateur pour affiner davantage la configuration.

La configuration du *query planner* dans PostgreSQL 16 peut avoir un impact significatif sur la performance des requêtes. Pour optimiser au mieux les performances, il faut ajuster certains paramètres spécifiques. Voici une liste des principaux réglages du *query planner* que vous pouvez envisager d'ajuster en fonction de votre charge de travail.

1. `effective_cache_size`

- **Description** : Ce paramètre estime la taille de la mémoire cache disponible pour PostgreSQL. Il aide le *planner* à décider s'il doit utiliser des index ou non.
- **Valeur recommandée** : En général, vous pouvez régler ce paramètre à environ 50 % à 75 % de la mémoire totale disponible sur votre serveur.

bash Copier le code

```
effective_cache_size = 8GB  # Exemple pour un serveur avec 16 Go de RAM
```

2. `work_mem`

- **Description** : Ce paramètre définit la quantité de mémoire utilisée par les opérations de tri et les *hash joins* avant que les données ne soient écrites sur disque.
- **Valeur recommandée** : Augmentez cette valeur si vous effectuez des tris ou des agrégations importantes, mais soyez prudent, car elle s'applique à chaque opération.

```bash
work_mem = 64MB
```

3. `random_page_cost`

- **Description** : Ce paramètre influence les décisions du *planner* sur l'utilisation des index. Une valeur plus faible indique que l'accès aléatoire au disque (par index) est moins coûteux.
- **Valeur recommandée** : Sur des systèmes avec des disques SSD, vous pouvez abaisser cette valeur à 1.1 ou 1.5. Sur des disques plus lents, gardez la valeur par défaut de 4.0.

```bash
random_page_cost = 1.2  # Pour un serveur avec SSD
```

4. `seq_page_cost`

- **Description** : Similaire à `random_page_cost`, mais pour les lectures séquentielles. Si vous avez un stockage rapide (comme des SSD), vous pouvez réduire cette valeur.
- **Valeur recommandée** : Pour des disques SSD, vous pouvez descendre à environ 0.5.

```bash
seq_page_cost = 0.5  # Pour un serveur avec SSD
```

5. `cpu_tuple_cost`, `cpu_index_tuple_cost`, `cpu_operator_cost`

- **Description** : Ces paramètres indiquent au *planner* le coût de traitement d'une ligne ou d'une opération. Si vos CPU sont rapides, vous pouvez réduire légèrement ces valeurs pour encourager l'utilisation des CPU par rapport aux lectures disque.
- **Valeur recommandée** : Généralement, des ajustements mineurs suffisent, mais vous pouvez expérimenter des valeurs légèrement inférieures à celles par défaut.

```bash
cpu_tuple_cost = 0.03
cpu_index_tuple_cost = 0.001
cpu_operator_cost = 0.002
```

6. `parallel_setup_cost` et `parallel_tuple_cost`

- **Description** : Ces paramètres affectent l'utilisation du traitement parallèle. Des valeurs plus faibles peuvent encourager le *planner* à exécuter plus de requêtes en parallèle.
- **Valeur recommandée** : Sur des systèmes multi-cœurs, vous pouvez expérimenter avec des valeurs réduites.

```bash
parallel_setup_cost = 1000
parallel_tuple_cost = 0.1
```

7. `geqo_threshold` et `geqo_effort`

- **Description** : Le *Genetic Query Optimizer* (GEQO) est activé pour les requêtes complexes (beaucoup de jointures). En réglant correctement ces paramètres, vous pouvez améliorer les performances des requêtes complexes.
- **Valeur recommandée** : Pour les requêtes impliquant plus de 12 jointures, vous pouvez ajuster `geqo_threshold` et expérimenter avec `geqo_effort`.

```bash
geqo_threshold = 12
geqo_effort = 5
```

8. Autres paramètres parallèles

- `max_parallel_workers_per_gather` : Ce paramètre définit combien de processus de travail parallèle peuvent être utilisés pour chaque opération de type *gather*.
- `max_parallel_workers` : Le nombre total maximal de processus de travail parallèle disponibles pour le serveur.
- **Valeur recommandée** : Ajustez s[illegible] le nombre de cœurs CPU disponibles.

```bash
max_parallel_workers_per_gather = 4
max_parallel_workers = 8
```

9. `jit` (Just-in-Time Compilation)

- **Description** : PostgreSQL 12+ propose la compilation JIT qui peut améliorer les performances des requêtes complexes.
- **Valeur recommandée** : Pour les requêtes complexes et les systèmes avec de la puissance CPU, activez-le, sinon désactivez-le.

```bash
jit = on  # Activez si vous avez des requêtes lourdes et beaucoup de CP
```

10. Statistiques

- Il est essentiel d'avoir des statistiques de haute qualité pour que le *query planner* puisse prendre de bonnes décisions. Assurez-vous d'analyser régulièrement vos tables avec `ANALYZE` et ajustez le paramètre `default_statistics_target` en fonction de la complexité des colonnes.

```bash
default_statistics_target = 100
```

BEST TUNING TOOLS

1. pgTune

- **Description** : pgTune est un outil qui aide à générer un fichier `postgresql.conf` optimisé en fonction de la configuration matérielle et des besoins spécifiques de l'application.
- **Utilité** : Il est particulièrement utile pour ajuster automatiquement les paramètres tels que la mémoire, les buffers, la taille des fichiers de journalisation, etc.
- **Lien** : pgTune

2. pg_stat_statements

- **Description** : C'est une extension intégrée dans PostgreSQL qui collecte des statistiques sur les requêtes SQL exécutées, telles que le temps d'exécution, le nombre d'appels, et les coûts associés.
- **Utilité** : Cette extension permet d'identifier les requêtes les plus coûteuses afin de les optimiser (par exemple en ajoutant des index ou en réécrivant certaines requêtes).
- **Commandes** : `CREATE EXTENSION pg_stat_statements;`

3. pgBadger

- **Description** : pgBadger est un analyseur de logs qui produit des rapports détaillés sur l'utilisation des ressources, les requêtes lentes et d'autres aspects des performances de PostgreSQL.
- **Utilité** : Il permet de visualiser facilement les problèmes de performance en analysant les fichiers de log générés par PostgreSQL.

4. auto_explain

- **Description** : C'est une autre extension qui permet de générer automatiquement les plans d'exécution pour les requêtes lentes ou coûteuses.
- **Utilité** : Elle aide à comprendre les raisons des ralentissements en montrant les plans d'exécution des requêtes sans avoir à les exécuter manuellement avec `EXPLAIN`.
- **Commandes** : `CREATE EXTENSION auto_explain;`

5. pg_repack

- **Description** : pg_repack est un outil qui permet de reconstruire les index et de réorganiser les tables sans verrouiller la base de données, ce qui améliore les performances sans interrompre les opérations.
- **Utilité** : Cet outil est particulièrement utile pour les bases de données ayant des tables ou index fragmentés, ou des espaces disque gaspillés.
- **Lien** : pg_repack

6. EXPLAIN et EXPLAIN ANALYZE

- **Description** : Ce sont des commandes PostgreSQL qui permettent d'analyser le plan d'exécution des requêtes.
- **Utilité** : Elles fournissent des informations sur la manière dont une requête est exécutée, ce qui aide à identifier les optimisations possibles (utilisation des index, tri, etc.).

7. HypoPG

- **Description** : HypoPG est une extension qui permet de simuler la création d'index sans les appliquer réellement. Cela permet d'évaluer si un index améliorerait une requête avant de l'implémenter.
- **Utilité** : Très utile pour tester les effets des index hypothétiques sans les coûts de création et de maintenance réels.
- **Commandes** : `CREATE EXTENSION hypopg;`

8. pgbench

- **Description** : pgbench est un outil de benchmarking intégré dans PostgreSQL qui permet de tester les performances de la base de données sous diverses charges de travail.
- **Utilité** : Il est utilisé pour simuler des charges sur la base de données et tester la réactivité après optimisation.
- **Commandes** : `pgbench -i -s 10` (initialisation de test) ; `pgbench -c 10 -j 2 -T 60` (benchmark)

9. VACUUM et ANALYZE

- **Description** : Les commandes `VACUUM` et `ANALYZE` sont des outils essentiels intégrés dans PostgreSQL pour nettoyer les tuples obsolètes et mettre à jour les statistiques des tables.
- **Utilité** : Elles permettent de maintenir les performances de la base de données en supprimant les données inutiles et en optimisant les plans de requête.

10. **Prometheus et Grafana (pour la surveillance)**

- **Description** : Prometheus est un outil de surveillance open source qui collecte des métriques de performance, et Grafana est utilisé pour visualiser ces métriques dans des tableaux de bord.
- **Utilité** : Ces outils permettent de surveiller en temps réel les performances de PostgreSQL et d'identifier les goulots d'étranglement.
- **Lien** : Prometheus, Grafana

GRAFANA & POSTGRESQL

Qu'est-ce que Grafana ?

Grafana est un outil open-source de visualisation et de surveillance de données qui permet de créer des tableaux de bord interactifs et dynamiques à partir de diverses sources de données. En tant que solution de surveillance, Grafana est particulièrement apprécié pour son extensibilité, sa compatibilité avec de nombreux systèmes de bases de données et sa capacité à fournir des visualisations en temps réel.

Pour PostgreSQL, Grafana permet de surveiller divers aspects de la performance de la base de données tels que les temps de réponse, la charge, les connexions actives, les requêtes lentes, l'utilisation des disques et de la mémoire, etc.

Pourquoi utiliser Grafana pour PostgreSQL 16 ?

- **Surveillance en temps réel** : Grafana offre la possibilité de surveiller PostgreSQL en temps réel et de voir instantanément les effets de vos changements de configuration ou d'optimisations.
- **Visualisation des métriques** : Il permet de créer des graphiques et des tableaux de bord personnalisés qui affichent des métriques telles que les transactions par seconde, les lectures/écritures, la latence, et bien plus.
- **Détection proactive des problèmes** : Grâce aux alertes configurables et aux visualisations, vous pouvez détecter des problèmes potentiels avant qu'ils n'affectent les performances de votre base de données.
- **Support multi-source** : Grafana peut se connecter à plusieurs bases de données et services, ce qui permet de surveiller non seulement PostgreSQL, mais aussi d'autres composants de votre architecture tels que le serveur

Principales fonctionnalités de Grafana pour PostgreSQL

1. **Connecteur PostgreSQL natif** : Grafana inclut un connecteur pour PostgreSQL, vous permettant de vous connecter directement à votre base de données, d'exécuter des requêtes SQL et d'afficher les résultats sous forme de graphiques.
2. **Tableaux de bord personnalisés** : Vous pouvez créer des tableaux de bord interactifs qui visualisent diverses métriques, allant des statistiques de requêtes aux paramètres système tels que l'utilisation de la mémoire et du CPU.
3. **Alertes** : Grafana permet de définir des alertes basées sur des seuils de performance. Par exemple, vous pouvez configurer une alerte pour être notifié si le nombre de connexions actives dépasse un certain seuil ou si le temps de réponse des requêtes ralentit.
4. **Intégration avec Prometheus** : Pour une surveillance plus poussée, Grafana s'intègre parfaitement avec Prometheus, un autre outil de surveillance qui collecte des métriques spécifiques à PostgreSQL via des **exporters** (par exemple, `postgres_exporter`).
5. **Exploration des logs** : En plus des métriques, Grafana permet de visualiser les logs PostgreSQL en temps réel, facilitant ainsi le diagnostic des problèmes.

Configuration de Grafana pour PostgreSQL

1. **Installation de Grafana**

Grafana peut être installé sur diverses plateformes (Linux, Windows, Docker, etc.). Voici un aperçu rapide pour l'installation sous Linux :

bash Copier le code

```bash
sudo apt-get install -y software-properties-common
sudo add-apt-repository "deb https://packages.grafana.com/oss/deb stabl
sudo apt-get update
sudo apt-get install grafana
sudo systemctl start grafana-server
sudo systemctl enable grafana-server.service
```

2. **Ajout de PostgreSQL comme source de données**

Une fois Grafana installé et accessible via l'interface web, vous pouvez ajouter PostgreSQL comme source de données :

- Accédez à **Configuration** > **Data Sources**.
- Sélectionnez PostgreSQL parmi les options disponibles.
- Fournissez les détails de connexion à votre base de données PostgreSQL (hôte, base de données, utilisateur, mot de passe, etc.).
- Testez la connexion pour vous assurer que Grafana peut se connecter à PostgreSQL.

3. **Création d'un tableau de bord**

Après avoir ajouté la source de données PostgreSQL, vous pouvez commencer à créer des tableaux de bord :

1. **Créer un nouveau tableau de bord** : Cliquez sur **Create** puis sur **Dashboard**.
2. **Ajouter des panneaux** : Chaque panneau est un graphique, une jauge ou un tableau. Vous pouvez créer des panneaux pour surveiller :
   - Les transactions par seconde (requêtes `SELECT`, `INSERT`, `UPDATE`, etc.).
   - Le nombre de connexions actives.
   - L'utilisation de la mémoire et du CPU par PostgreSQL.
   - Les lectures/écritures sur disque.
   - Les requêtes lentes (à partir de `pg_stat_statements`).
3. **Personnaliser les graphiques** : Vous pouvez écrire des requêtes SQL dans Grafana pour récupérer des données spécifiques de PostgreSQL. Grafana vous permet de visualiser ces données dans des formats divers (graphes, diagrammes à barres, jauges, etc.).

4. **Surveillance via Prometheus et Postgres Exporter**

Pour une surveillance plus avancée, il est recommandé d'utiliser Prometheus et **postgres_exporter**, qui permet de récupérer des métriques spécifiques à PostgreSQL. Voici les étapes générales pour la mise en place :

1. **Installation de Prometheus** : Prometheus va collecter les métriques de PostgreSQL.
2. **Configuration de Postgres Exporter** : Cet exporter récupère des métriques PostgreSQL et les expose dans un format compréhensible par Prometheus.
3. **Connexion de Grafana à Prometheus** : Vous ajoutez Prometheus comme source de données dans Grafana, puis vous pouvez créer des tableaux de bord basés sur les métriques récupérées par `postgres_exporter`.

Exemple de métriques disponibles via `postgres_exporter` :

- Temps d'exécution des requêtes.
- Blocages dans la base de données.
- Taille des tables et index.
- Transactions par seconde.

Exemple de Tableau de Bord PostgreSQL

Un tableau de bord typique pour PostgreSQL dans Grafana peut inclure :

- **Nombre de connexions actives.**
- **Transactions par seconde (TPS).**
- **Temps moyen d'exécution des requêtes.**
- **Taille des bases de données.**
- **Utilisation du CPU par PostgreSQL.**
- **Métriques des tables et des index** (nombre d'insertions, suppressions, etc.).

Conclusion

Grafana est un outil puissant pour surveiller PostgreSQL, surtout lorsqu'il est utilisé en combinaison avec Prometheus et `postgres_exporter`. Il offre une grande flexibilité en matière de visualisation et d'alertes, ce qui le rend idéal pour surveiller les performances de PostgreSQL 16 en temps réel et diagnostiquer les problèmes de manière proactive.

BENCHMARK

COMPARATIVE EC2 / Residence 4198 - ARM

EC2	RESTORE DB	UPDATE UR	DAILY TASK	CHARGE MGNT	EXPORT	PRECALCUL	PRICE
C7G-12X LARGE	> 35 mn		2,04 mn				0,93 $
M7G-8xLarge	32 mn		2,20 mn				0,69 $
R6g-8xLarge	> 38 mn		2,38 mn				0,81 $

COMPARATIVE EC2 / Residence 4198 - compute

EC2	memory	UPDATE UR	DAILY TASK	HTTPS - WEB	EXPORT	PRECALCUL	PRICE
g6.8xLarge		4,43 mn	2,10 mn			3,01 mn	1,17 $
Inf 1.6	48 gb	6,35 mn	2,24 mn	4,05 sec	6 sec	3,31 mn	
m6i,8xLarge	128 gb	5,38 mn	1,54 mn	2,44 sec	10 sec	2,52 mn	0,81 $ *
m6a.8xlarge	128 gb	4,50 mn	2,04 mn	2,61 sec	11 sec	3,14 mn	0,73 $
gr6,4xLarge	128 gb	4.43 mn	2,01 mn	2,34 sec	11 sec	3.43 mn	0,89 $
Inf 2.8	128 gb	5,36 mn	1,55 mn	2,57 sec		2,55 mn	1,27 $ *
g4dn.8xLarge		5.49 mn	2,33 mn			3.49 mn	1,23 $

COMPARATIVE EC2 / Residence 4198 - memory

EC2	Memory	RESTORE	UPDATE UR	DAILY TASK	https	EXPORT	PRECALCUL	PRICE
r6g.8xLarge								
r6i.8xLarge	128 gb		5,38 mn	1,51 mn			2,51 mn	1,07 $ *
r7i.4Xlarge	128 gb		6,58 mn	2,32 mn			4,10 mn	1,24 $ *
r6gd.4xLarge	128 gb							1,08 $
r6i.4xLarge	128 gb							1,18 $

COMPARATIVE EC2 / Residence 4198 - General

EC2	Memory	RESTORE DB	UPDATE UR	DAILY TASK	CHARGE MGNT	EXPORT	PRECALCUL	PRICE
m7i.8xLarge								