Engineer

Index de base de données

structure trouvant des lignes sans parcourir toute la table

Ce que c'est

Un index de base de données est une structure distincte, stockée à côté d'une table, qui permet à la base d'atteindre directement les lignes correspondant à une condition, au lieu de lire la table de la première à la dernière ligne. Dans l'usage courant et la plupart de la documentation PostgreSQL, on dit simplement index. La documentation le compare à l'index en fin de livre : le lecteur cherchant un sujet y trouve les pages pertinentes et s'y rend, au lieu de feuilleter tout le volume.

Prenez une table de commandes et la condition `WHERE customer_id = 42` . Sans index sur `customer_id` , PostgreSQL a une façon de répondre : un parcours séquentiel, lisant chaque ligne pour garder celles qui correspondent. Le coût est le même, que le client ait deux commandes ou deux mille, et augmente avec la table. Avec un index sur la colonne, PostgreSQL traverse quelques niveaux d'un arbre de recherche jusqu'aux entrées de 42 et ne récupère que ces lignes. Un index se crée en une seule instruction, `DROP INDEX` le supprime, et les deux opérations sont possibles à tout moment.

Une fois l'index créé, PostgreSQL le synchronise avec la table à chaque écriture. Le planificateur décide requête par requête si son usage coûte moins qu'un parcours. Rien dans le SQL ne nomme l'index ; le même peut donc accélérer un `SELECT` , un `UPDATE` ou `DELETE` avec condition de recherche, ainsi qu'une jointure.

PostgreSQL propose plusieurs types, chacun conçu pour une forme de question :

- **B-tree** est le type construit par `CREATE INDEX` sans autre indication. Il répond aux comparaisons d'égalité et de plage (`<` , `<=` , `=` , `>=` , `>`), à `BETWEEN` , `IN` , `IS NULL` et à un `LIKE` dont le motif est ancré au début. Ses entrées restent ordonnées ; il peut donc aussi renvoyer les lignes déjà triées.
- **Hash** stocke un code de hachage de 32 bits par valeur et ne répond qu'à l'égalité.
- **GiST** et **SP-GiST** sont des cadres d'indexation pour des données comme la géométrie, où l'on demande « chevauche », « contient » ou « le plus proche », plutôt que « égal ».
- **GIN** est un index inversé pour des valeurs composées de plusieurs parties, comme les tableaux. Il trouve les lignes dont la valeur contient un élément donné.
- **BRIN** ne garde qu'un résumé par plage de blocs de table. Il convient à une colonne dont les valeurs suivent l'ordre physique des lignes, comme souvent un horodatage d'insertion.

Au-delà du type, la forme de l'index peut s'adapter à la question. Un index unique refuse les valeurs en double. Un index multicolonne couvre plusieurs colonnes, jusqu'à 32. Un index partiel a une clause `WHERE` et ne contient que les lignes qui la satisfont. Un index d'expression stocke une valeur calculée comme `lower(email)` . Une liste `INCLUDE` contient des colonnes supplémentaires sans les utiliser pour la recherche ; une requête peut ainsi recevoir sa réponse du seul index.

Utilisez le concept pour prendre une décision

Utilisez « Qu'est-ce qu'un index de base de données ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. Identifier les requêtes et les index manquants dans un système alourdi au fil des ans, puis corriger ce qui compte.

Découvrir nos prestations Contactez-nous →

Vérifiez les hypothèses derrière le terme

Appuyez-vous sur l'explication et les réalisations liées pour formuler les questions de votre projet. Nous pouvons examiner les hypothèses avec vous avant le choix d'un outil. 30 jours de données électriques traités en 180 ms — 180 ms pour agréger un mois de relevés de compteurs

30 jours de données électriques traités en 180 ms Contactez-nous →

Pourquoi c'est important

L'écart produit par un index grandit avec la table. Sur une petite table, le parcours coûte peu, et le planificateur le préfère souvent à raison : lire quelques pages dans l'ordre coûte moins que traverser un arbre pour les atteindre. Sur des dizaines de millions de lignes, l'écart entre parcours et recherche ciblée sépare

une page qui se charge d'une page dont le chargement dépasse le délai d'attente. Une requête rapide au lancement peut devenir lente un an plus tard sans changement de code, simplement parce que sa table a grandi.

Les index ne sont pas gratuits ; le coût se paie lors des écritures. Chaque insertion ajoute une entrée à chaque index de la table. Seule une mise à jour de type heap-only tuple (HOT) peut éviter ces ajouts : la nouvelle version de la ligne reste sur la même page, et les entrées d'index existantes y mènent encore. PostgreSQL ne l'autorise que si la mise à jour ne touche aucune colonne utilisée par les index de la table, à l'exception des BRIN, qui ne portent qu'un résumé. Chaque index réduit donc les cas où cette optimisation est possible ; la documentation conseille de supprimer ceux qui servent rarement ou jamais. Une table ayant une douzaine d'index paie une douzaine de fois à chaque insertion.

Construire l'index a aussi un coût. Un `CREATE INDEX` ordinaire laisse les autres sessions lire la table, mais bloque leurs écritures jusqu'à la fin de la construction. La documentation prévient que sur une très grande table cela peut prendre de nombreuses heures.

Enfin, la valeur d'un index dépend de la décision du planificateur de l'utiliser, fondée sur les statistiques recueillies par `ANALYZE`. La question n'est pas « y a-t-il un index ? », mais « les requêtes importantes l'utilisent-elles ? ». `EXPLAIN` permet de le savoir.

À quoi cela ressemble en pratique

La page de compte d'une boutique en ligne montre les vingt dernières commandes d'un client :

```
SELECT id, created_at, total
FROM   orders
WHERE  customer_id = 42
ORDER  BY created_at DESC
LIMIT  20;
```

Avec seulement une clé primaire sur `id`, PostgreSQL doit lire tout `orders`, garder les lignes du client, les trier, puis ne conserver que les vingt premières. `EXPLAIN` le montre : un `Seq Scan` sur `orders`, avec la condition comme `Filter`

et un tri au-dessus. `EXPLAIN ANALYZE` exécute aussi la requête et affiche les nombres de lignes et durées réels à côté des estimations.

L'index adapté place la colonne d'égalité en premier et celle du tri en second :

```
CREATE INDEX CONCURRENTLY orders_customer_created_idx
ON orders (customer_id, created_at);
```

Relancez `EXPLAIN` : le parcours devient un parcours d'index utilisant `orders_customer_created_idx`, et `customer_id = 42` apparaît comme `Index Cond`, et non plus comme filtre. Le tri disparaît aussi. Un B-tree garde les entrées du client 42 ordonnées par `created_at` ; PostgreSQL peut les lire à rebours pour commencer par les plus récentes. Avec un `LIMIT`, il s'arrête après vingt entrées sans toucher au reste.

L'ordre des colonnes compte. Le même index sert une requête sur sa première colonne, `customer_id`, seule. Mais un rapport sur toutes les commandes d'hier, filtrant seulement `created_at`, en tire bien moins : dans un B-tree multicolonne, l'égalité sur les premières colonnes puis une plage sur la suivante réduisent la partie lue. PostgreSQL 18 sait faire un skip scan, répétant la recherche pour chaque valeur distincte de la première colonne. Le planificateur ne le choisit toutefois en général que lorsque la première colonne a peu de valeurs distinctes, et les identifiants clients en ont beaucoup. Si le rapport compte, il reçoit son index sur `created_at`. À l'inverse, `(created_at, customer_id)` disperserait les commandes du client 42 dans tout l'index.

La note sur l'optimisation de la base avant de la répartir place les index en tête des corrections classées par rendement, une fois que `EXPLAIN (ANALYZE, BUFFERS)` a montré les parcours d'une requête très sollicitée. Son exemple est un index partiel, `units_unsold`, ne gardant que les unités encore à vendre : il rétrécit donc au fil de la vente au lieu de grossir. La note compte aussi le prix. Dans sa table, les 2 000 demandes de réservation modifiant `reserved`, `reserved_at` et `order_id` étaient toutes des mises à jour HOT. Parmi les 2 000 mises à jour de `sold`, colonne lue par l'index partiel, aucune ne l'était. Elle conseille de supprimer les index dont `idx_scan` dans `pg_stat_user_indexes` reste à zéro.

L'étude de cas sur un meilleur choix de clé montre pourquoi les valeurs entrant dans un B-tree comptent autant que les colonnes. Les UUID aléatoires dispersent les insertions dans l'index, ce qui peut le fragmenter. La plateforme a retenu

comme choix standard des UUID v7 ordonnés dans le temps pour les clés primaires des entités, avec un horodatage dans les premiers bits pour transmettre cet ordre aux nouveaux identifiants. Réduire la fragmentation et accélérer les requêtes étaient des objectifs de conception. Le résultat consigné est la convention de clés configurée ; l'étude n'établit aucun benchmark de charge avant et après.

Où les problèmes surviennent

L'ajout d'index arrête les écritures. Un `CREATE INDEX` ordinaire sur une table active bloque insertions, mises à jour et suppressions pendant toute sa construction ; un déploiement courant devient une panne. Utilisez `CREATE INDEX CONCURRENTLY` : il parcourt la table deux fois et attend les transactions susceptibles d'utiliser l'index. Cela prend nettement plus longtemps, mais les écritures continuent. Il ne peut s'exécuter dans un bloc de transaction ; un outil de migration qui enveloppe chaque étape dans une transaction doit donc lancer cette instruction seule. Une seule construction concurrente peut s'exécuter à la fois sur une table.

Une construction concurrente échoue et laisse un index invalide. Un interblocage, ou un doublon pendant la construction d'un index unique, fait échouer `CREATE INDEX CONCURRENTLY`. L'index commencé reste toutefois dans le catalogue, marqué invalide. Les requêtes l'ignorent, mais chaque écriture paie encore son entretien, et un index unique invalide peut continuer à imposer l'unicité. Le `\d` de psql affiche `INVALID` après sa définition. Supprimez-le et relancez la construction concurrente, ou reconstruisez-le avec `REINDEX INDEX CONCURRENTLY`.

La requête cache la colonne dans une fonction. Un index sur `email` ne sert pas à `WHERE lower(email) = 'ana@example.com'`, car il contient `email`, pas `lower(email)`. Indexez l'expression de la requête :

```
CREATE INDEX users_lower_email_idx ON users (lower(email));
```

L'expression se calcule à chaque insertion et chaque mise à jour non HOT. La documentation recommande donc ces index quand la vitesse de lecture compte plus que celle d'écriture.

Les index inutilisés s'accumulent. Chaque ajout « au cas où » coûte à chaque écriture et peut empêcher les mises à jour HOT. `pg_stat_user_indexes` compte les parcours commencés sur chaque index dans `idx_scan` et enregistre l'horodatage du dernier dans `last_idx_scan` :

```
SELECT relname, indexrelname, idx_scan, last_idx_scan
FROM   pg_stat_user_indexes
WHERE  idx_scan = 0
ORDER  BY relname;
```

Vérifiez avant de supprimer. Un index derrière une clé primaire ou une contrainte unique fait son travail même si aucune requête ne le lit. Un compteur à zéro signifie peu sur un serveur qui vient de commencer à recevoir du trafic.

Le planificateur ignore un bon index. Après un chargement massif, les statistiques sont en retard sur les données, les estimations sont fausses et le parcours gagne sur le papier. Lancez `ANALYZE` ; la documentation qualifie d'« effort perdu » l'évaluation de l'usage des index sans lui. Testez aussi sur des volumes réalistes : sur cent lignes tenant sur une page disque, aucun index ne bat la lecture de cette page. Pour savoir si l'index est utilisable, définissez `enable_seqscan = off` dans une session de test, puis relancez `EXPLAIN`. Si le plan parcourt toujours la table, la condition ne correspond pas à l'index.

Pour aller plus loin

- Index : introduction — rôle, création et suppression d'un index, et surcoût de chaque écriture.
- Types d'index — B-tree, Hash, GiST, SP-GiST, GIN et BRIN, et les opérateurs auxquels chacun répond.
- Index multicolonnes — l'effet de l'ordre des colonnes sur la réduction de recherche dans un B-tree et les conditions du skip scan.
- CREATE INDEX — la syntaxe complète, dont `CONCURRENTLY`, `INCLUDE`, les index partiels et la récupération après construction invalide.
- Utiliser EXPLAIN — lire un plan : types de parcours, coûts estimés et valeurs réelles d' `EXPLAIN ANALYZE`.
- Examiner l'utilisation des index — les raisons de commencer par `ANALYZE` et les données réelles, et le test de l'utilisabilité d'un index.

Glossaire

Choisissez selon vos contraintes réelles

Présentez un jeu de données représentatif, les questions auxquelles il doit répondre et le goulot d'étranglement actuel. Nous pouvons convenir de la manière de mesurer la qualité, le débit et la reprise.

Consulter les tarifs Contactez-nous →

Rendez la terminologie concrète

Demandez comment « Qu'est-ce qu'un index de base de données ? » s'applique à votre organisation. Un exemple concret tiré de votre système actuel suffit pour engager la discussion.

Découvrir nos prestations

welcome@engineer.company +45 50 10 61 84 Copenhague, Danemark

Emportez cette page avec vous

Fiche de contact

Créé avec soin et amour par D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Droit d'auteur © 2025 – aujourd'hui · Engineer ApS · Tous droits réservés.

Imprimé depuis https://engineer.company/fr/glossary/database-index/