Engineer Données

Optimiser la base avant de la répartir

14 octobre 2026

Lorsqu'une base de données ralentit sous la charge, la première proposition consiste généralement à ajouter des machines. C'est parfois la bonne réponse. Plus souvent, la lenteur vient d'une poignée d'instructions que personne n'a encore lues, et répartir les données entre serveurs répartit ces instructions sur davantage de matériel sans en rendre une seule moins coûteuse. Un problème apparemment insoluble n'est souvent qu'à un plan de requête d'une réponse élégante, et le passage à l'échelle horizontal est parfois choisi précisément pour que personne n'ait à lire la base de données dont vous disposez déjà.

Voici notre ordre de travail avant toute division : observer, optimiser, retirer les lectures du primaire, partitionner les tables autour du point de contention et ne répartir entre serveurs que lorsqu'une machine est remplie de travail utile. La vente éclair où chaque acheteur écrit sur une seule ligne est détaillée dans Vendre le dernier billet exactement une fois. Cette note décrit la méthode qui l'entoure, et sa dernière section examine la proposition inverse : vendre depuis chaque serveur et s'excuser pour les unités vendues deux fois.

Appliquez l'ingénierie à votre système

L'ingénierie derrière « Optimiser la base avant de la répartir » peut éclairer votre prochain changement. Nous pouvons étudier un comportement similaire dans vos logiciels, vos données ou votre infrastructure. Une structure fidèle au fonctionnement de l'entreprise, qui permet encore de poser demain les questions dont vous aurez besoin.

Découvrir nos prestations Contactez-nous →

Suivez le raisonnement et ses limites

Découvrez le raisonnement et ses limites dans cette note. Nous pouvons vérifier si le même diagnostic s'applique à votre environnement. 6 pipelines ETL/ELT conçus et exploités — 6 pipelines ETL sur trois moteurs de bases de données

6 pipelines ETL/ELT conçus et exploités Contactez-nous →

Observer d’abord

PostgreSQL enregistre déjà l’essentiel de ce qu’il faut savoir. Avant toute modification, posez-lui quatre questions.

Quelles instructions coûtent le plus au total ? `pg_stat_statements`, une extension chargée via `shared_preload_libraries`, conserve chaque instruction normalisée avec son nombre d’appels et sa durée. Triez par total, pas par moyenne : une requête de 2 ms exécutée un million de fois par heure coûte plus qu’un rapport de deux secondes exécuté deux fois.

```
SELECT calls,
       round(total_exec_time) AS total_ms,
       round(mean_exec_time::numeric, 1)
         AS mean_ms,
       left(query, 60) AS query
  FROM pg_stat_statements
 ORDER BY total_exec_time DESC
 LIMIT 10;
```

Que fait réellement la première ? `EXPLAIN (ANALYZE, BUFFERS)` exécute l’instruction et montre le plan choisi, les lignes réellement renvoyées par chaque étape par rapport aux lignes attendues, et le nombre de pages lues. Parcourir un million de lignes pour en renvoyer dix, ou une estimation fausse d’un facteur cent, explique généralement tout. `ANALYZE` exécute ce qu’il explique ; encadrez donc une écriture par `BEGIN` et `ROLLBACK`.

Qui attend, et qui le bloque ?

```
SELECT pid, wait_event_type, wait_event,
       pg_blocking_pids(pid) AS blocked_by,
       left(query, 60) AS query
  FROM pg_stat_activity
 WHERE wait_event_type = 'Lock';
```

Si des lignes apparaissent ici pendant une minute de forte activité, cela signifie que les transactions font la queue les unes derrière les autres, plutôt que d’attendre le matériel. Une machine plus puissante ne supprime pas la nécessité

d'exécuter une à une les mises à jour d'une même ligne partagée. Des serveurs applicatifs supplémentaires peuvent ajouter à cette file des clients qui effectuent des écritures.

Le serveur travaille-t-il ou jongle-t-il ? Comparez le nombre de sessions actives dans `pg_stat_activity` au nombre de cœurs. Lorsqu'il y a plusieurs fois plus de sessions actives que de cœurs, le serveur passe son temps à basculer de l'une à l'autre ; un pool de connexions corrige cela avant tout ajout matériel. Consultez ensuite `n_dead_tup` dans `pg_stat_user_tables` : une table avec davantage de lignes mortes que vivantes est parcourue à travers ses propres déchets. C'est autovacuum qui prend du retard, pas une base trop petite.

Consignez vos constats. Ils constituent la référence à laquelle chaque étape ultérieure sera comparée, et contiennent souvent déjà la réponse honnête à « devons-nous répartir la base entre serveurs ? ».



Optimiser l'existant

Approximativement par ordre de rendement de l'effort.

Des index qui correspondent aux questions. Les plans ci-dessus nomment les parcours. Un index sur les colonnes filtrées par une requête très sollicitée transforme un parcours en recherche directe, et un index partiel ne contient que les lignes que la requête peut vouloir — par exemple les unités encore en vente. Il rétrécit donc au fil de la vente plutôt que de croître :

```
CREATE INDEX units_unsold
    ON units (item_id)
 WHERE NOT sold;
```

Chaque index a aussi un coût à chaque écriture ; supprimez donc ceux dont `idx_scan` dans `pg_stat_user_indexes` reste à zéro.

Des types qui disent ce que représente la valeur. Les schémas écrits rapidement, par une personne ou un modèle, choisissent `TEXT` là où `SMALLINT` ou `BOOLEAN` suffiraient. Un `BOOLEAN` occupe un octet, un `SMALLINT` deux, et `'yes'` stocké comme texte quatre. PostgreSQL ajoute aussi du remplissage

à chaque colonne de largeur fixe selon son alignement ; un ordre de colonnes choisi au hasard coûte donc également des octets. Une unité de stock, écrite rapidement :

```
CREATE TABLE units (
  sold        text,  -- 'yes' or 'no'
  id          bigint,
  reserved    text,  -- 'yes' or 'no'
  reserved_at timestamptz,
  item_id     bigint,
  order_id    bigint
);
```

Et la même unité avec des types qui expriment les valeurs, les plus larges en premier :

```
CREATE TABLE units (
  id          bigint PRIMARY KEY,
  reserved_at timestamptz,
  order_id    bigint,
  item_id     integer NOT NULL,
  sold        boolean NOT NULL DEFAULT false,
  reserved    boolean NOT NULL DEFAULT false
);
```

Sur PostgreSQL 18, la première ligne mesure 72 octets et la seconde 54, soit un quart de moins ; une plus grande partie de la table tient donc en mémoire. Les types refusent aussi des valeurs telles que `'maybe'`, dont l'application devrait autrement se protéger. Une contrainte `CHECK` fait de même pour les plages de valeurs : `quantity smallint CHECK (quantity > 0)`.

Moins d'allers-retours et le travail là où sont les données. Chaque instruction envoyée par l'application coûte un aller-retour et une analyse. Une boucle qui lit cent lignes puis les met à jour une par une représente cent un allers-retours là où une instruction ensembliste suffirait. Une page qui récupère chaque unité d'un article pour compter celles qui sont libres transfère des milliers de lignes pour calculer un nombre. Notre règle : ce qui doit voir les données va dans la base — filtrage, comptage, écriture dépendant de l'existant — et les règles métier restent dans l'application, car la puissance de calcul de la base est celle qu'il est le plus difficile d'augmenter dans le système.

Des mises à jour qui restent sur leur page. Une ligne modifiée plusieurs fois par minute laisse à chaque fois une nouvelle version d'elle-même. Lorsqu'une mise à jour ne modifie aucune colonne indexée et que la page a de la place, PostgreSQL garde la nouvelle version sur la même page sans toucher aux index : un heap-only tuple, ou une mise à jour HOT. Gardez donc les colonnes fréquemment modifiées hors des index et laissez de l'espace sur les pages :

```
ALTER TABLE units SET (fillfactor = 90);
```

Ce réglage s'applique aux pages écrites ensuite ; définissez-le donc avant le chargement des lignes. Sur la table ci-dessus, 2 000 réservations modifiant `reserved` , `reserved_at` et `order_id` ont toutes été HOT. Sur 2 000 mises à jour modifiant `sold` , lu par l'index partiel, aucune ne l'a été. Une unité n'est vendue qu'une fois ; sa réservation peut se répéter. Le rapport entre `n_tup_hot_upd` et `n_tup_upd` dans `pg_stat_user_tables` montre cette proportion sur un système en production.



Une ligne désigne le premier arrivé

Une unité revient au premier acheteur dont l'instruction l'atteint, et la réservation expire selon l'horloge de la base. Deux colonnes la portent : `reserved` , un booléen, et `reserved_at` , l'instant où il a été défini.

```
UPDATE units
   SET reserved = true,
       reserved_at = now(),
       order_id = :order
 WHERE id = :unit
   AND NOT sold
   AND (NOT reserved
        OR reserved_at
           < now() - interval '10 minutes')
RETURNING id;
```

Une ligne renvoyée : cet acheteur l'a obtenue. Aucune : quelqu'un l'a précédé ou l'unité est vendue. Deux réservations de la même unité attendent sur le verrou de sa ligne. Avec READ COMMITTED, le niveau d'isolation prédéfini de PostgreSQL, la seconde revérifie la clause `WHERE` sur la ligne validée par la

première ; exactement une des deux obtient donc une ligne en retour. Une réservation de plus de dix minutes est considérée comme libre ; aucun nettoyage n'est nécessaire pour que la réponse soit correcte.

Le booléen seul donne donc une réponse incomplète. `reserved` reste vrai après expiration, et seules les deux colonnes réunies indiquent si une unité est retenue ; toute requête doit donc interroger les deux. Un nettoyage périodique qui réinitialise les réservations expirées maintient la table cohérente pour les lecteurs de `reserved` seul, mais la correction du résultat ne l'attend jamais. Une contrainte maintient la cohérence de la paire :

```
ALTER TABLE units
  ADD CONSTRAINT reserved_has_time
  CHECK (NOT reserved
         OR reserved_at IS NOT NULL);
```

La note sur les billets stocke la date limite, `held_until`, plutôt que le début ; c'est la même conception. Lorsque l'acheteur choisit l'unité — un siège, un tirage numéroté — l'instruction la nomme, comme ici. Lorsque n'importe quelle unité convient, la forme `SKIP LOCKED` de cette note en choisit une que personne d'autre ne retient.

Retirer les lectures du primaire

La majeure partie du trafic d'une boutique est de la lecture : la page produit, la mention « 3 restants », le panier, l'historique des commandes. Rien de cela ne nécessite le primaire, cette machine qui doit rester disponible pour les écritures. Une réplique en flux continu prend ces lectures, et en ajouter une relève de la configuration là où une répartition entre serveurs constitue un projet.

Deux précautions l'accompagnent. Une réplique a un léger retard ; un acheteur qui vient de réserver une unité puis y lit son panier peut donc ne pas voir la réservation : lisez sur le primaire ce qu'un utilisateur vient d'écrire, ou envoyez-le vers le primaire pendant quelques secondes après chaque écriture. Un rapport long sur une réplique peut aussi être annulé s'il retarde les changements que celle-ci rejoue. La documentation PostgreSQL appelle cela des conflits de requêtes et énumère les réglages qui arbitrent entre ces conflits et le retard de réplication.

La lecture la moins coûteuse est celle qui n'atteint jamais une base. Lorsque chaque unité d'un article est réservée, personne ne peut l'acheter avant l'expiration ou la libération d'une réservation, et la base peut indiquer exactement quand cela arrivera :

```
SELECT count(*) FILTER (
         WHERE NOT reserved
            OR reserved_at
               < now() - interval '10 minutes')
         AS free,
       min(reserved_at) + interval '10 minutes'
         AS until
  FROM units
 WHERE item_id = :item
   AND NOT sold;
```

Lorsque `free` vaut zéro, `until` est l'expiration de la réservation la plus proche : mettez « entièrement réservé » en cache jusqu'à cet instant, jamais au-delà, et tous les visiteurs jusque-là sont refusés par le cache plutôt que par la base. Lorsque `free` vaut zéro et que `until` est vide, toutes les unités sont vendues et la réponse est définitive. Une réservation libérée plus tôt — carte refusée, panier abandonné — supprime l'entrée de cache, et l'unité est immédiatement remise en vente.

Le cache ne dit jamais que non. Un oui vient toujours de l' `UPDATE` conditionnel ci-dessus ; un cache périmé peut donc faire perdre une vente pendant quelques secondes, mais jamais vendre deux fois une unité. C'est le « épuisé » mis en cache de la note sur les billets, avec une date limite.



Partitionner les tables autour de la ligne très sollicitée

Le partitionnement divise une table en plusieurs selon une clé — un mois, une organisation cliente — derrière un seul nom. Il faut être clair sur ce qu'il ne fait pas : il n'accélère pas une ligne soumise à contention. Cette ligne reste dans une partition, sur une machine, derrière un verrou.

Il maintient en revanche petites les tables autour du point de contention. Les commandes, l'historique des réservations et les événements croissent indéfiniment, et la plupart des requêtes veulent la partie récente :

```
CREATE TABLE orders (
  id          bigint NOT NULL,
  created_at  timestamptz NOT NULL
              DEFAULT now(),
  item_id     integer NOT NULL,
  quantity    smallint NOT NULL
              CHECK (quantity > 0),
  paid        boolean NOT NULL DEFAULT false,
  PRIMARY KEY (id, created_at)
) PARTITION BY RANGE (created_at);

CREATE TABLE orders_2026_10
  PARTITION OF orders
  FOR VALUES FROM ('2026-10-01')
              TO ('2026-11-01');
```

Une requête indiquant une plage de dates ne lit que les partitions concernées, et les index du mois courant restent assez petits pour tenir en mémoire. La conservation ne passe plus par un `DELETE` de millions de lignes, avec le gonflement et le nettoyage qui suivent, mais par une instruction :

```
ALTER TABLE orders
  DETACH PARTITION orders_2025_10;
```

La clé primaire doit inclure la clé de partition, d'où `(id, created_at)`. Une requête qui ne nomme pas cette clé lit chaque partition ; partitionnez donc selon les filtres déjà utilisés par les requêtes.



Quand la répartition entre serveurs est raisonnable

Il est une limite que cette répartition ne peut pas déplacer : chaque article a un seul responsable. Une unité ne peut pas être réservée sur deux serveurs à la fois ; on déplace donc des articles entiers, jamais la moitié d'un article. Lorsqu'un

article dépasse les capacités de sa machine, les compartiments de la note sur les billets divisent son stock en morceaux plus petits, chacun ayant à nouveau exactement un responsable.

Dans ce cadre, nous jugeons la répartition raisonnable lorsqu'une des conditions suivantes est remplie, pas avant :

- **Une machine est pleine de travail utile.** Les instructions sont déjà optimisées — une écriture conditionnelle chacune, aucun appel externe dans une transaction, des mises à jour restant sur leur page — et le processeur ou le journal d'écriture anticipée du primaire reste saturé, sur la plus grosse machine qu'il soit raisonnable d'acheter, pour de nombreux articles plutôt qu'un seul. Un article très sollicité relève des compartiments, pas de la répartition entre serveurs.
- **Les données ont dépassé les capacités de maintenance.** Une sauvegarde ne peut plus être restaurée dans le délai promis à l'entreprise, un index ne peut plus être construit ou une table nettoyée pendant les heures creuses, ou l'ensemble de travail ne tient plus en mémoire.
- **L'isolation est l'exigence.** Une organisation cliente dont le contrat ou le régulateur impose des données séparées, ou dont la vente ne doit partager aucun disque avec celle d'un autre.

Le coût se reporte sur tout le reste. Les migrations, sauvegardes, restaurations et rapports doivent tenir compte des fragments distincts ; une jointure entre fragments peut être déplacée dans le code applicatif ; une requête ignorant sa clé de répartition peut devoir les interroger tous. C'est la facture que les sections précédentes cherchent à éviter et sur laquelle il est difficile de revenir une fois payée.

Quand des excuses coûtent moins cher qu'un verrou

Il existe un argument sérieux en faveur de la voie inverse. Laisser chaque serveur vendre le stock qu'il voit, accepter qu'une unité revienne parfois à deux acheteurs, puis laisser l'assistance rembourser l'un d'eux avec des excuses. La logique métier devient à cohérence éventuelle, conçue pour les 99,9 % de requêtes sans collision, et une boucle asynchrone détecte les collisions après coup et les répare. En échange, la boutique n'a jamais à dire « réessayez ». Helland et Campbell présentent la version la plus forte de cet argument dans *Building on*

Quicksand : un système qui accuse réception d'un travail avant que sa sauvegarde l'ait rend toute promesse probabiliste, et les entreprises ont toujours réglé par des excuses les promesses qu'elles ne pouvaient tenir.

Nous sommes d'accord avec l'essentiel. Nous divergeons sur l'endroit où appliquer ce compromis.

La majeure partie d'une boutique devrait être à cohérence éventuelle. Le catalogue, l'index de recherche, le badge « 3 restants », le panier, les recommandations, les courriels, l'export vers l'entrepôt et les rapports peuvent tous avoir quelques secondes de retard. Les faire attendre le primaire coûte de la vitesse sans rien apporter. Dynamo a été conçu précisément pour cela : le panier Amazon n'oublie ni ne refuse jamais un ajout, même pendant une panne de machines, au prix de fusions susceptibles de réintroduire un article supprimé par l'acheteur. Après la réservation, le paiement de la note sur les billets est déjà asynchrone : une saga d'étapes locales, des messages livrés au moins une fois et des consommateurs idempotents, indexés par commande. Une boucle de rapprochement a sa place dans toute conception qui communique avec un prestataire de paiement, car des webhooks se perdent.

La réservation est le seul endroit où cela ne tient pas. « Rare » suppose des collisions rares, alors que dans une vente éclair, les collisions sont la vente. Un serveur qui vend depuis sa propre vue continue jusqu'à apprendre les ventes des autres ; la survente est donc approximativement le taux de tentatives multiplié par le retard de réplication. Dans une simulation simple, deux régions se partagent 2 000 tentatives par seconde, chacune vend depuis son propre décompte et apprend les ventes de l'autre 200 millisecondes plus tard. Sur cinq exécutions d'une vente de 10 000 unités, chacune a vendu entre 191 et 205 unités inexistantes. Le calcul prédit 200 : le taux de tentatives, multiplié par le retard, multiplié par la part des ventes effectuées dans l'autre région. Avec un seul responsable par article, le même modèle a vendu exactement le stock. Cela arrive à chaque épuisement de stock, qu'une panne ait eu lieu ou non, au point culminant, lorsque le plus d'acheteurs regardent.

Le gain de disponibilité est plus faible qu'il n'y paraît. Gilbert et Lynch ont démontré le compromis de la cohérence éventuelle : pendant une partition réseau, un système peut répondre ou rester cohérent, pas les deux. Le modèle PACELC d'Abadi ajoute l'autre moitié. Sans partition, le compromis oppose latence et cohérence, et un responsable unique par article avec des instructions optimisées est déjà rapide. Aucune conception n'atteint 100 % : le prestataire de

paiement, le DNS et les réseaux qui les relient connaissent leurs propres pannes. Ce que sacrifie la conception stricte est limité. Lorsque le responsable du stock d'un article est inaccessible, cet article cesse de se vendre pendant les secondes nécessaires au basculement, et la section de la note sur la suspension limite celle-ci aux articles de la machine en panne. Le reste de la boutique continue à vendre.

Des excuses ne sont pas un remboursement. Le remboursement est la partie peu coûteuse. L'acheteur a reçu un oui et a peut-être déjà réservé son train, posé un jour de congé et prévenu ses amis. Entendre « épuisé » une seconde trop tôt est une déception. Apprendre l'annulation d'une commande confirmée est une promesse rompue. L'assistance est dimensionnée pour une journée ordinaire, et les surventes arrivent ensemble, pendant la minute la plus chargée de l'année. Avant de prévoir le retrait de commandes confirmées, demandez à un juriste à quoi une confirmation engage la boutique.

Gardez l'asynchronisme et placez-le avant le oui. La proposition a raison sur un point : l'acheteur n'a pas à attendre sur un verrou. Acceptez chaque clic dans un journal durable et répliqué, avec une partition par article, et indiquez à l'acheteur que la réponse arrive. L'acceptation d'un clic dépend alors uniquement du journal, et un journal répliqué sur trois machines continue d'accepter les écritures si l'une échoue. Un seul responsable par article lit le journal dans l'ordre et décide à la vitesse de la mémoire ; la réponse revient par message en quelques secondes. Lorsque ce responsable bascule, les requêtes attendent dans le journal plutôt que d'être refusées ou vendues deux fois. Le oui est une réservation, suivie du paiement comme dans la note sur les billets ; personne ne paie donc une unité non obtenue. Le rapprochement existe toujours, mais avant la promesse plutôt qu'après.

Partagez le stock, pas le décompte. Lorsque l'activité exige que les régions vendent sans se consulter, attribuez à chacune un quota de stock et laissez-la le vendre sans demander à personne. Une région à court de quota en demande davantage et, en fin de vente, les quotas se regroupent sous un responsable unique. Bailis et ses collègues déterminent les règles que des serveurs peuvent respecter sans communiquer ; un stock qui ne doit pas devenir négatif n'en fait pas partie. Les quotas n'éliminent pas cette communication ; ils la déplacent hors du chemin critique, au rare moment de leur réapprovisionnement. C'est la conception à cohérence éventuelle qui conserve l'invariant. Son seul coût est la fin de vente, où un acheteur est refusé dans une région alors qu'une autre a encore du stock.

Quand la survente est la bonne politique. Certains biens peuvent être reproduits : un tirage à la demande, un tee-shirt encore disponible chez le fournisseur, une clé de licence. Vendre au-delà du décompte devient alors une commande différée avec une date honnête, et la cohérence éventuelle est la bonne conception. Les compagnies aériennes pratiquent volontairement la surréservation. Elles connaissent les taux de non-présentation, chiffrent le risque et indemnisent selon une règle écrite à l'avance. C'est le critère. Une survente est acceptable lorsque l'entreprise a choisi son ampleur et son remède avant la vente. C'est un défaut lorsqu'elle retombe sur la personne qui a cliqué en dernier.

Si vous choisissez les excuses, construisez-les comme les paiements. La boucle décrite par la proposition est un processus de remboursement. Elle nécessite les mêmes choses que tout système déplaçant de l'argent :

- **Un identifiant sur chaque vente**, pour qu'une nouvelle tentative soit la même vente, et un registre de ventes en ajout seul. Le nombre en stock est dérivé du registre, jamais modifié.
- **Une règle désignant qui conserve l'unité**, explicable à l'acheteur perdant. Ordonnez selon le journal, pas selon les horloges des serveurs, où quelques millisecondes de décalage décideraient du remboursement.
- **Des compensations idempotentes et journalisées :** annuler, annuler l'autorisation ou rembourser, prévenir l'acheteur. Ce sont les étapes d'annulation de la saga, exécutées par une machine.
- **Une échéance.** La boucle doit trouver chaque survente avant toute expédition et avant que l'acheteur ne la découvre.
- **Un budget mesuré.** Les surventes par vente, comparées au nombre accepté par l'entreprise, à côté de l'invariant sur le même tableau de bord, avec une alerte lorsque le budget est épuisé.

Notre réponse consiste à adopter la cohérence éventuelle partout autour de la réservation, la rigueur au moment de réserver et l'asynchronisme pour transmettre la réponse à l'acheteur. Concevez pour le parcours habituel, mais dans une vente éclair, la collision est ce parcours habituel. Si l'entreprise accepte quelques surventes en échange d'un avantage, faites-en un nombre qu'elle a choisi, avec un remède écrit avant la vente, et mesurez-le.

Ce que nous dirions à la prochaine personne

- **Lisez avant d'acheter.** `pg_stat_statements` et un `EXPLAIN` répondent à davantage de questions de capacité qu'un nouveau serveur.
- **Faites exprimer aux types la nature des données.** Un booléen stocké comme booléen est plus petit et refuse les absurdités.
- **Envoyez à la base ce qui doit voir les données, rien d'autre.** Le filtrage et le comptage y ont leur place. Les règles métier non.
- **Le cache dit non ; seule la base dit oui.** Un cache périmé coûte alors des secondes, jamais une unité.
- **Le partitionnement est de l'entretien.** Il garde petites les tables autour d'une ligne très sollicitée et ne change rien pour cette ligne.
- **Répartissez entre serveurs lorsqu'une machine est pleine de travail utile**, pas lorsqu'elle est pleine d'attente.
- **Rapprochez avant le oui, pas après.** Tout ce qui entoure la réservation peut être à cohérence éventuelle. La réservation elle-même, conçue ainsi, entraîne à chaque vente une survente égale au taux de tentatives multiplié par le retard.

Sources

- PostgreSQL : pg_stat_statements
- PostgreSQL : utilisation d'EXPLAIN
- PostgreSQL : le système de statistiques cumulatives
- PostgreSQL : types numériques
- PostgreSQL : types de caractères
- PostgreSQL : type booléen
- PostgreSQL : index partiels
- PostgreSQL : heap-only tuples (HOT)
- PostgreSQL : partitionnement des tables
- PostgreSQL : réplication en attente active et conflits de requêtes
- Pat Helland et Dave Campbell, Building on Quicksand
- Seth Gilbert et Nancy Lynch, Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services
- Daniel Abadi, Consistency Tradeoffs in Modern Distributed Database System Design
- DeCandia et al., Dynamo: Amazon's Highly Available Key-value Store
- Hector Garcia-Molina et Kenneth Salem, Sagas

- Bailis et al., Coordination Avoidance in Database Systems

Tous les articles du blog

Mesurez avant de modifier

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 →

Discutez de vos observations

Discutez de « Optimiser la base avant de la répartir » dans le contexte de votre propre projet. Dites-nous ce que vous avez observé et à quoi ressemblerait un résultat utile.

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://data.engineer.company/fr/notes/optimise-before-you-shard/