Engineer

Transaction

groupe de changements en base validés ou annulés ensemble

Ce que c’est

Une transaction est un groupe de changements qu’une base de données applique comme une seule unité : soit tous prennent effet, soit aucun. On précise transaction de base de données pour la distinguer d’un paiement, qui partage le même mot. Le tutoriel de PostgreSQL décrit une transaction comme ce qui « regroupe plusieurs étapes dans une seule opération, tout ou rien », sans qu’aucun état intermédiaire soit visible par les autres.

Dans PostgreSQL, une transaction commence par `BEGIN` et se termine par `COMMIT`, qui rend ses changements permanents et visibles aux autres sessions, ou par `ROLLBACK`, qui les abandonne. Une instruction envoyée seule s’exécute tout de même dans une transaction : PostgreSQL l’entoure d’un `BEGIN` implicite et, si elle réussit, d’un `COMMIT` implicite. Une longue transaction peut poser un `SAVEPOINT`, puis y revenir avec `ROLLBACK TO`, en abandonnant le travail effectué depuis tout en conservant ce qui précède.

Les garanties d’une transaction se résument par ACID. L’atomicité signifie que la transaction s’achève comme un tout ou pas du tout, sans qu’un résultat partiel survive à un plantage. La cohérence signifie que les contraintes déclarées sur les données sont respectées à la validation ; une transaction dont la validation violerait une contrainte est annulée. L’isolation signifie que ses changements restent invisibles aux transactions concurrentes jusqu’à la validation. La durabilité signifie qu’un changement validé survit au plantage du serveur.

L’isolation se décline en plusieurs niveaux, et le standard SQL en définit quatre : Read Uncommitted, Read Committed, Repeatable Read et Serializable. PostgreSQL commence chaque transaction en Read Committed sauf instruction contraire ; son Read Uncommitted se comporte exactement comme Read Committed. En Read Committed, chaque instruction voit les données validées

avant son propre début : deux requêtes de la même transaction peuvent donc voir des chiffres différents. En Repeatable Read, toute la transaction travaille sur un instantané pris à sa première instruction qui ne soit pas une instruction de contrôle de transaction. Serializable va plus loin et fait produire aux transactions concurrentes le même résultat que si elles s'étaient exécutées l'une après l'autre.

Utilisez le concept pour prendre une décision

Utilisez « Qu'est-ce qu'une transaction ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. 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 →

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. 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 →

Pourquoi c'est important

Presque toute action digne d'être enregistrée touche plus d'une ligne. Une commande est une ligne dans une table, une diminution dans une autre et une entrée dans un registre. Sans transaction, un plantage, un délai dépassé ou une contrainte violée entre la deuxième et la troisième instruction laisse une commande dont le stock n'a jamais été retiré, ou du stock retiré pour une commande inexistante. Quelqu'un le découvre ensuite dans un rapport et le répare à la main. Avec une transaction, cet état incomplet n'existe jamais aux yeux des autres ; c'est la base, pas l'application, qui le garantit.

La deuxième raison est la concurrence. Deux requêtes arrivant ensemble s'exécutent côte à côte ; le niveau d'isolation décide de ce que chacune voit, tandis que les verrous déterminent laquelle attend. C'est là que les équipes sont le plus souvent surprises : entourer le code de `BEGIN` et `COMMIT` n'empêche pas, à lui seul, deux requêtes de lire le même solde et de le dépenser toutes deux. En Read Committed, chaque instruction lit ce qui était validé à son démarrage ; entre une vérification faite par une instruction et une écriture faite par la suivante, une autre session peut donc valider. Il faut d'abord verrouiller la ligne lors de la vérification, ou intégrer celle-ci à l'écriture.

Les transactions ont aussi un coût, payé en temps. Les verrous de lignes pris par `UPDATE`, `DELETE` et `SELECT ... FOR UPDATE` durent normalement jusqu'à la fin de la transaction ; revenir à un point de sauvegarde libère les verrous acquis après ce

point. Une session demandant un verrou incompatible doit attendre, sauf si une limite de temps, une annulation ou la résolution d'un interblocage l'arrête. Une transaction ouverte empêche aussi `VACUUM` de retirer d'anciennes versions de lignes dont elle pourrait encore avoir besoin : la laisser ouverte pendant des heures fait donc grossir les tables. La question de conception n'est jamais seulement de savoir s'il faut une transaction : il faut en fixer les limites et réduire ce qui se passe entre elles.

Enfin, une transaction locale ordinaire de base de données n'inclut pas, à elle seule, les effets externes. Le retour arrière de la base n'annule ni un appel à un prestataire de paiement, ni un courriel, ni un message envoyé dans une file distincte. Coordonner d'autres systèmes exige un travail de conception supplémentaire ; un `BEGIN` englobant ne suffit pas. Décider de ce qui entre dans les limites, et de la manière dont le monde extérieur apprend la validation, constitue l'essentiel de ce travail de conception.

À quoi cela ressemble en pratique

Un membre d'un programme de fidélité dépense 250 points pour un bon de cinéma. Trois opérations doivent se faire ensemble : le portefeuille perd les points, le registre en consigne la raison et l'opération d'échange est créée. Dans une seule transaction, cela donne :

```
BEGIN;

UPDATE wallet
SET    points = points - 250
WHERE  member_id = 42
AND    points >= 250
RETURNING points;
-- no row back: not enough points, so ROLLBACK and stop here

INSERT INTO points_entry (member_id, delta, reason)
VALUES (42, -250, 'redeemed: cinema voucher');

INSERT INTO redemption (member_id, reward_id)
VALUES (42, 7);

COMMIT;
```

L'`UPDATE` prend un verrou sur la ligne du portefeuille du membre 42. Jusqu'au `COMMIT`, toute autre session lisant le portefeuille voit encore l'ancien solde, puisque les verrous de lignes ne bloquent pas les lectures et que les changements non validés sont invisibles. Un second échange pour le même membre, envoyé d'un autre onglet peu après, atteint son propre `UPDATE` et attend. Lorsque la première transaction est validée, PostgreSQL en Read Committed réévalue la clause `WHERE` de la seconde sur la ligne telle qu'elle est devenue. S'il ne reste que 100 points, la condition échoue, aucune ligne n'est renvoyée et la seconde requête est annulée sans aucune écriture. Si c'est plutôt le dernier `INSERT` qui échoue, par exemple parce que la récompense 7 a été retirée et qu'une contrainte de clé étrangère empêche l'insertion, le client annule et les points retrouvent exactement leur état initial.

C'est le principe du portefeuille de fidélité avec ligues et badges. Chaque changement d'un portefeuille y prend un verrou sur sa ligne avant de la lire : une seconde requête attend donc la fin de la première au lieu d'entrer en concurrence avec elle. Comme le portefeuille est dans la même base que tout le reste, le solde et ce qu'il a acheté sont validés ensemble ou pas du tout. La page expose le prix : le verrouillage des lignes est la solution plus lente, et les membres attendent leur tour au lieu de dépenser deux fois les mêmes points.

La note sur la vente du dernier billet exactement une fois applique le même outil à une vente éclair. Sa correction réunit la vérification du stock et sa diminution dans un seul `UPDATE` conditionnel : garder un verrou pendant que l'application réfléchit entre un `SELECT` et un `UPDATE` fait attendre l'acheteur suivant pendant chaque milliseconde de cet intervalle. Le premier conseil pour le reste du travail est de garder tout effet externe hors de la transaction : appel de paiement, requête HTTP ou message publié doivent se placer avant ou après.

Où les problèmes surviennent

Une session reste inactive dans une transaction et tout attend derrière elle. L'application ouvre une transaction, modifie une ligne, puis attend une API lente, le prochain clic d'un utilisateur ou un processus de traitement qui a planté sans jamais envoyer `COMMIT`. Les verrous restent en place ; avec les limites de temps désactivées, PostgreSQL fait attendre indéfiniment une session demandant un verrou incompatible, sauf s'il détecte un interblocage ou si l'attente est annulée. La documentation déconseille de garder une transaction ouverte en attendant une saisie utilisateur. `pg_stat_activity` affiche ces sessions dans l'état `idle in transaction`, et `xact_start` indique le début de la transaction :

```
SELECT pid, now() - xact_start AS open_for, state, query
FROM pg_stat_activity
WHERE state LIKE 'idle in transaction%'
ORDER BY xact_start;
```

Sortez le travail lent de la transaction et fixez une limite au serveur pour qu'une transaction oubliée ne puisse garder des verrous pendant des heures. `idle_in_transaction_session_timeout` termine toute session restée inactive dans une transaction ouverte au-delà de sa valeur. Sa valeur prédéfinie est 0, ce qui le désactive : il faut donc le régler, par exemple pour le rôle de l'application avec `ALTER ROLE app SET idle_in_transaction_session_timeout = '30s';` .

Deux requêtes dépensent toutes deux le même solde. Le code lit une valeur dans une instruction, décide dans l'application et écrit le résultat dans une autre. Deux sessions lisent le même nombre, réussissent la vérification et écrivent toutes deux : un changement écrase silencieusement l'autre. Un `BEGIN` englobant ne l'empêche pas en Read Committed. Intégrez la vérification à l'écriture avec une condition dans l' `UPDATE` , comme dans l'exemple ci-dessus, ou verrouillez la ligne avant lecture avec `SELECT ... FOR UPDATE` , ce qui fait attendre la seconde session jusqu'à la fin de la première.

Des erreurs apparaissent sous charge en Repeatable Read ou Serializable. À ces niveaux, PostgreSQL refuse une transaction qui violerait la garantie du niveau et signale un échec de sérialisation, SQLSTATE `40001` , avec des messages comme `could not serialize access due to concurrent update` . C'est le fonctionnement prévu, pas un bogue ; la documentation précise que les applications utilisant ces niveaux doivent prévoir des reprises. Interceptez `40001` et réexécutez toute la transaction depuis `BEGIN` , lectures comprises, car sa décision reposait sur des données qui ont changé depuis. Ne reprendre que l'instruction en échec reproduit l'erreur.

Deux transactions s'interbloquent. L'une verrouille la commande 1 puis la commande 2, l'autre la commande 2 puis la commande 1, et chacune attend l'autre. PostgreSQL détecte le cycle et annule une transaction pour que l'autre puisse finir ; il est difficile de prévoir laquelle sera choisie. La défense documentée consiste à faire acquérir les verrous dans le même ordre à chaque chemin de code, par exemple en mettant à jour les lignes par ordre croissant de leur clé primaire, et à réexécuter une transaction annulée en raison d'un interblocage.

Un courriel part pour une commande annulée. L'application envoie une confirmation ou publie un événement depuis la transaction, puis une instruction ultérieure échoue. Le retour arrière supprime la commande, mais ne peut rappeler le courriel. Enregistrez le message dans une table outbox dans la même transaction que le changement qu'il décrit, puis laissez un processus distinct l'envoyer après la validation. Le message existe alors exactement lorsque la commande existe.

Pour aller plus loin

- Transactions — le chapitre du tutoriel PostgreSQL : tout ou rien, transactions implicites et points de sauvegarde, illustrés par un virement bancaire.
- Isolation des transactions — les quatre niveaux standards, ce que chacun voit dans PostgreSQL et les échecs de sérialisation qui demandent une reprise.
- Verrouillage explicite — verrous de lignes et de tables, durée de détention, détection et prévention des interblocages.
- Valeurs par défaut des connexions clientes — `default_transaction_isolation` , `idle_in_transaction_session_timeout` , `lock_timeout` et les autres limites d'une session.
- Nettoyage régulier par VACUUM — pourquoi les anciennes versions de lignes restent tant qu'une transaction pourrait les voir et les effets des transactions longues sur le nettoyage.

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'une transaction ? » 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/transaction/