Engineer

Vendre le dernier billet exactement une fois

13 octobre 2026

Une vente éclair présente dix mille unités à un million de personnes à la fois. La défaillance intéressante n'est pas que la boutique tombe en panne. C'est qu'elle reste disponible et en vende dix mille quarante. Le réflexe qui suit — la base est trop petite, répartissons-la entre serveurs — répond à une question que personne n'a posée. La base n'a jamais été trop grosse. Chaque acheteur écrivait sur la même ligne.

Nous analysons ici le problème avant qu'il n'atteigne un projet : d'où vient le plafond, l'instruction unique qui en élimine l'essentiel et ce qu'il faut construire lorsqu'une seule base ne suffit réellement plus.

Appliquez l'ingénierie à votre système

L'ingénierie derrière « Vendre le dernier billet exactement une fois » 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 →

Une ligne, chaque acheteur

Vingt millions de lignes réparties sur une grappe constituent un problème résolu. Une ligne que chaque requête doit modifier n'en est pas un, et ajouter des machines ne change rien, car cette ligne réside sur l'une d'elles. Les stockages

distribués le disent clairement : DynamoDB accorde environ 1 000 unités d'écriture par seconde à une partition, et chaque écriture sur un élément atteint sa partition, quel que soit le nombre des autres.

Un théorème sous-tend cette structure. Bailis et ses collègues qualifient un invariant d'invariant-confluent lorsque les répliques peuvent accepter des écritures seules puis fusionner dans un état valide. `stock >= 0` est le cas d'école d'un invariant qui ne l'est pas : deux répliques voyant chacune une unité restante la vendront chacune, et aucune fusion ne peut annuler la seconde vente. Ne jamais survendre nécessite de la coordination. La seule question est la part de coordination payée par chaque vente.

Le nombre qui décide

Le plafond pour un article correspond approximativement au nombre de ventes couvertes par un verrou, divisé par la durée de maintien de ce verrou :

```
max sales/s per item ≈ sales per lock ÷ lock hold time
```

Ce qui retient la ligne	Plafond par ligne
Verrou, puis appel au paiement	~2/s
Verrou, puis logique applicative	~100–200/s
Un `UPDATE` conditionnel	~500–2 000/s
Lots de 100 par commit	~50 000/s
Un seul processus d'écriture en mémoire	~100 000+/s

Ce sont des ordres de grandeur, pas des mesures de référence ; le matériel les déplace. Il ne change pas la structure. L'essentiel d'un verrou court est l'attente, lors du commit, de l'arrivée du journal sur disque. C'est pourquoi le traitement par lots gagne deux ordres de grandeur : cent ventes partagent une seule écriture forcée.

Il reste quatre leviers, et chaque conception ci-dessous en utilise un :

- **Garder le verrou moins longtemps** — une instruction, aucun échange réseau à l'intérieur.
- **Faire davantage par verrou** — regrouper les décréments.
- **Avoir davantage de verrous** — répartir le stock sur plusieurs clés.
- **Envoyer moins de requêtes au verrou** — refuser la plupart des acheteurs avant qu'ils l'atteignent.

  Le dernier compte plus qu'il n'y paraît. Avec un million d'acheteurs et dix mille unités, 990 000 requêtes doivent échouer. Les refus représentent l'essentiel du trafic ; « épuisé » doit donc être la réponse la moins coûteuse du système : une salle d'attente avant la vente, un contrôle d'admission qui laisse entrer environ autant d'acheteurs qu'il y a d'unités, un indicateur `sold_out` servi par le CDN sans demander à personne et une limite de débit par compte.

Corriger d'abord la base unique

La version qui se traîne ou survend ressemble généralement à ceci :

```
BEGIN;
SELECT available FROM inventory
 WHERE item_id = :item FOR UPDATE;
-- the application checks the number,
-- and perhaps calls a payment provider
UPDATE inventory SET available = :new
 WHERE item_id = :item;
COMMIT;
```

Le verrou est correct. Ce qu'il couvre ne l'est pas : chaque milliseconde passée par l'application entre le `SELECT` et l' `UPDATE` est une milliseconde d'attente pour l'acheteur suivant. Sans `FOR UPDATE` , c'est pire. Deux acheteurs lisent 1, écrivent tous deux 0, et une unité est vendue deux fois.

La correction réunit vérification et décrément dans une instruction :

```
UPDATE inventory
   SET available = available - :qty
 WHERE tenant_id = :tenant
   AND item_id = :item
```

```
   AND available >= :qty
RETURNING available;
-- one row back: reserved. none: sold out.
```

Dans PostgreSQL avec READ COMMITTED, un second acheteur qui atteint la ligne pendant que le premier la verrouille attend, puis réévalue la clause `WHERE` sur la ligne validée par le premier. Si cette vente a pris la dernière unité, la condition échoue et le second acheteur ne reçoit aucune ligne — épuisé — plutôt qu'un nombre négatif. Avec REPEATABLE READ ou SERIALIZABLE, il reçoit à la place un échec de sérialisation, sûr mais transformant la contention en nouvelles tentatives ; cette instruction est écrite pour READ COMMITTED. InnoDB de MySQL arrive à la même réponse, car un `UPDATE` y lit la dernière ligne validée à tout niveau. Ajoutez aussi la contrainte pour qu'un bogue ailleurs échoue de manière visible plutôt que de vendre du stock inexistant :

```
ALTER TABLE inventory
  ADD CONSTRAINT available_not_negative
  CHECK (available >= 0);
```

Puis le reste du travail sur une base unique, approximativement par ordre de rendement :

- **Rien d'externe dans la transaction.** Aucun appel de paiement, aucune requête HTTP, aucun message publié. Ils vont avant la transaction ou après.
- **Toucher la ligne très sollicitée en dernier.** Insérez d'abord la commande et tout le reste, pour que le verrou sur la ligne soumise à contention ne vive que pendant l'instruction finale et le commit.
- **Aucun verrouillage optimiste sur un compteur très sollicité.** Une colonne de version transforme la contention en nouvelles tentatives et, pendant une vente, ces tentatives constituent la charge.
- **Servir les lectures ailleurs.** La mention « 12 restants » de la page produit peut venir d'une réplique ou d'un cache en retard d'une seconde. Seule la vente nécessite le primaire.
- **Donner au stock sa propre base**, pour qu'une vente n'attende jamais derrière le catalogue sur le même disque.
- **Mutualiser les connexions.** Cent mille acheteurs devraient représenter quelques dizaines de connexions via PgBouncer, pas des milliers de processus serveur attendant un verrou.

Les sièges présentent une autre structure, car chaque unité est distincte, et nécessitent une autre instruction. Chaque siège est une ligne, et un acheteur saute les lignes retenues par une autre transaction plutôt que de faire la queue derrière elles :

```
UPDATE seats
   SET held_by = :order,
       held_until = now() + interval '10 min'
 WHERE id = (
   SELECT id FROM seats
    WHERE event_id = :event
      AND sold_at IS NULL
      AND (held_until IS NULL
           OR now() > held_until)
    LIMIT 1
    FOR UPDATE SKIP LOCKED)
RETURNING id;
```

Une réservation expirée redevient simplement disponible ; aucun nettoyage n'est donc nécessaire pour que la réponse soit correcte. `now()` est l'horloge de la base ; des serveurs applicatifs aux horloges décalées ne peuvent donc pas diverger sur l'expiration d'une réservation.

Pour de nombreuses boutiques, cette section constitue toute la correction. Appliquez-la avant la suite, car la répartition entre serveurs est une porte à sens unique : facile à franchir, coûteuse à retraverser.



Réserver d'abord, payer ensuite

Un acheteur qui prend la dernière unité puis abandonne le formulaire de carte ne l'a pas achetée. Une vente n'est donc pas une écriture, mais un court cycle de vie :

```
AVAILABLE → HELD, for minutes → SOLD
                              → RELEASED
```

- **La réservation est indexée par demande, pas par tentative.** Chaque passage en caisse porte une clé d'idempotence et la table des réservations comporte `UNIQUE (tenant_id, idempotency_key)` ; un clic répété récupère donc sa réservation existante plutôt qu'une seconde unité. Un détail de PostgreSQL peut piéger ici : `INSERT ... ON CONFLICT DO NOTHING RETURNING id` ne renvoie rien si la ligne existait déjà ; « aucune ligne » signifie donc « la rechercher », pas « échec ».
- **Chaque transition est conditionnelle.** Confirmer une vente fait passer une réservation de `held` à `sold` uniquement si elle est encore `held`, et une libération ne recrédite le stock que si sa propre mise à jour a modifié exactement une ligne. Une libération exécutée deux fois — expiration et bouton d'annulation, par exemple — restitue l'unité une seule fois.
- **Les réservations expirent en quelques minutes.** Cinq à dix est habituel. Prolongez une fois la réservation au début du paiement, pour qu'une carte lente ne fasse pas perdre l'unité qu'elle règle.
- **Plafonner les unités par acheteur lors de la réservation**, pour qu'un compte ne retienne pas tout le stock pendant sa réflexion.

Sous l'ensemble se trouve une ligne, la plus utile de la conception :

```
available + held + sold = total
```

Elle reste vraie après chaque transaction, une requête la vérifie et, lorsqu'elle cesse d'être vraie, quelque chose a survendu ou perdu une unité — avant qu'un client ne le signale.



Quand une seule base ne suffit pas

Parfois, elle ne suffit réellement pas : de nombreuses organisations clientes, de nombreuses ventes simultanées, plusieurs régions. La structure qui passe à l'échelle conserve une règle — chaque article a exactement un responsable de son stock, un service de stock, et rien d'autre ne décide — et répartit ce service selon `(tenant_id, item_id)`, pour que les écritures d'un article ne couvrent jamais deux fragments. Les plus grandes organisations clientes reçoivent leurs propres cellules, pour que leur vente ne prive personne d'autre de ressources. Seule la réservation doit être synchrone ; tout ce qui suit peut être un message.

Pour l'article très sollicité à l'intérieur d'un fragment, quatre outils utilisent chacun l'un des leviers précédents.

Un contrôle d'admission en amont. Un compteur Redis, décrémenté par un script Lua pour que vérification et décrément ne forment qu'une étape, admet environ autant d'acheteurs qu'il y a d'unités et refuse les autres en quelques microsecondes. Il n'est pas l'arbitre. Redis réplique de manière asynchrone ; après un basculement, le nouveau primaire peut donc avoir oublié des admissions et en laisser passer un peu trop. L'`UPDATE` conditionnel de la base décide toujours, et un processus de rapprochement réinitialise l'admission à partir du nombre de la base.

Diviser le stock. Dix mille unités peuvent devenir cent compartiments de cent unités, chacun avec sa ligne ou son fragment, chaque acheteur étant dirigé vers l'un d'eux : cent verrous plutôt qu'un. L'idée est ancienne — les transactions escrow d'O'Neil en 1986, puis le protocole de démarcation de Barbará et Garcia-Molina, puis les compteurs bornés de Balegas et ses collègues — et son seul véritable coût est la fin de vente. Le compartiment d'un acheteur est vide alors que d'autres ont encore du stock ; il faut donc en essayer un autre ou déplacer les restes, puis revenir à un seul vers la fin. La même idée attribue un quota à chaque région, pour que la plupart des ventes n'aient jamais à traverser un océan pour demander.

Un seul processus d'écriture par article. Placez chaque demande pour un article dans une partition Kafka, lue par un consommateur unique qui garde le décompte en mémoire et valide ses décisions par lots. Un fil d'exécution, aucun verrou : la structure LMAX. Les demandes portent un identifiant, et le processus d'écriture ignore celles sur lesquelles il a déjà décidé, car une file d'attente peut livrer deux fois.

Les unités comme lignes. Lorsque les unités sont peu nombreuses et distinctes — sièges, éditions numérotées — créez une ligne pour chacune à l'avance et réservez-les avec `SKIP LOCKED` comme ci-dessus, ou conservez leurs identifiants dans un ensemble Redis et retirez-en un avec `SPOP`. Il ne reste aucun compteur disputé.

La suite du passage en caisse

Le paiement, la commande et le reçu sont des systèmes distincts, qu'aucune transaction ne couvre ensemble. La validation en deux phases est la réponse des manuels et la mauvaise ici : elle bloque dès qu'un participant est lent, ajoute des allers-retours au parcours le plus sollicité et n'est pas prise en charge par les prestataires de paiement. Une saga fait l'affaire : une suite d'étapes locales, chacune avec une étape qui l'annule.

```
1. reserve stock       undo: release the hold
2. pending order       undo: cancel the order
3. extend the hold     (expires on its own)
4. authorise payment   undo: void it
5. confirm HELD → SOLD
6. capture payment
```

Autoriser d'abord et capturer à la fin signifie qu'un paiement arrivant après expiration de sa réservation est annulé plutôt que remboursé : l'argent n'a jamais été déplacé. Chaque étape écrit son message dans une table outbox au sein de la même transaction que la modification annoncée, et la capture des changements de données le publie ; rien ne se perd donc entre enregistrement et envoi. Chaque consommateur est idempotent, indexé par commande. Les messages arrivent au moins une fois, et l'idempotence rend leur traitement effectif unique.

Un verrou distribué n'a sa place nulle part ici. Kleppmann explique que le détenteur d'un verrou peut rester suspendu au-delà de son bail — ramasse-miettes, machine virtuelle bloquée — puis continuer à écrire, et que le remède est un jeton de protection vérifié par le stockage. Mais un stockage capable de vérifier un jeton peut vérifier `available >= :qty` ; il n'a alors jamais eu besoin du verrou.

Suspendre l'article plutôt que le survendre

La cohérence et la disponibilité se choisissent par question, pas par système. « Puis-je vendre cette unité ? » exige de la cohérence : lorsque le responsable du stock d'un article est inaccessible, cet article cesse de se vendre. Tout le reste — le catalogue, le badge « 3 restants », l'historique des commandes — peut rester disponible avec un léger retard.

La survente la plus discrète vient d'un basculement. Avec une réplication asynchrone, un primaire confirme une vente puis meurt avant que sa réplique ne l'ait ; la réplique est promue, la vente a disparu et l'unité est vendue à nouveau. La correction consiste à rendre une vente durable sur plusieurs machines avant de la confirmer : une réplique de secours synchrone dans PostgreSQL (`synchronous_commit` avec `synchronous_standby_names`), ou un stockage construit sur Raft. Cela ajoute de la latence à chaque commit, ce qui allonge précisément la durée de verrou utilisée au dénominateur du plafond. Les lots et le contrôle d'admission compensent ce coût. Limitez aussi la suspension : un fragment en panne doit arrêter ses articles, pas la boutique.

Le démontrer

Rien de cela ne doit être accepté sur parole, y compris ce qui vient de nous.

- **Surveillez l'invariant.** Exécutez `available + held + sold = total` régulièrement, représentez-le et déclenchez une alerte d'astreinte à toute dérive. Zéro survente est un objectif de niveau de service assorti d'une alerte, pas un espoir.
- **Provoquez délibérément la concurrence.** Dix mille acheteurs simultanés pour cent unités, dans la suite de tests, en exigeant exactement cent ventes et aucun nombre négatif.
- **Provoquez des pannes pendant cette concurrence.** Arrêtez le primaire, coupez le réseau, suspendez un processus — à la manière de Jepsen — puis vérifiez l'invariant.
- **Testez les refus en charge.** Le parcours « épuisé » doit rester rapide lorsqu'il représente 99 % du trafic, et la périphérie doit l'absorber.
- **Placez les métriques sur un seul tableau de bord :** attentes de verrous, réservations expirées et écart entre le décompte de l'admission et celui de la base.

Dans quel ordre

- Mesurer d'abord. `pg_stat_statements` et les attentes de verrous indiquent si la ligne très sollicitée est réellement le problème.
- Réunir l'écriture dans une seule instruction conditionnelle.
- Délester : salle d'attente, contrôle d'admission, « épuisé » en cache.
- Donner au stock son propre service et son propre responsable.

- Répartir selon `(tenant_id, item_id)`, uniquement si les quatre premières étapes n'ont pas suffi.
- Ajouter un mode pour article très sollicité, des compartiments ou un processus d'écriture unique pour la vente prévue.
- Les régions en dernier, avec des quotas.

Ce que nous dirions à la prochaine personne

- **La répartition entre serveurs répond à une autre question.** Elle répartit de nombreuses lignes, alors qu'une vente survendue concerne une ligne. Déterminez votre cas avant de construire.
- **Réunissez la vérification et le décrément dans une instruction.** L'essentiel de l'écart entre bon fonctionnement et survente est le temps entre la lecture d'un nombre et son écriture.
- **La majeure partie du trafic est la réponse non.** Faites du refus l'action la moins coûteuse du système.
- **Suspendez avant de survendre.** Un article qui cesse de se vendre trente secondes demande des excuses. Un article vendu deux fois signifie un remboursement, une file d'assistance et un client qui s'en souvient.
- **Écrivez l'invariant et surveillez-le.** Il tient en une ligne et c'est la seule chose qui atteste du fonctionnement de la conception en production plutôt qu'en revue.

Sources

- PostgreSQL : isolation Read Committed
- PostgreSQL : `FOR UPDATE` et `SKIP LOCKED`
- PostgreSQL : réplication synchrone
- Bailis et al., Coordination Avoidance in Database Systems
- Balegas et al., Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants
- Martin Kleppmann : How to do distributed locking
- Martin Fowler : The LMAX Architecture
- DynamoDB : conception des clés de partition
- DynamoDB : partitions et distribution des données
- microservices.io : Saga
- microservices.io : Transactional outbox
- Redis : réplication
- Jepsen

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 « Vendre le dernier billet exactement une fois » 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://engineer.company/fr/notes/selling-limited-stock-without-overselling/