Engineer Софтвер

Продати останній квиток рівно один раз

13 жовтня 2026 р.

Під час флеш-розпродажу мільйон людей одночасно намагається купити десять тисяч одиниць товару. Цікавий збій тут не той, коли магазин падає, а той, коли магазин працює й продає десять тисяч сорок одиниць. Перша реакція — «база замала, шардуймо її» — відповідає на питання, якого ніхто не ставив. Справа ніколи не була в розмірі бази. Кожен покупець писав у той самий рядок.

Тут ми розбираємо проблему ще до того, як вона трапиться в проєкті: звідки береться стеля, яка єдина інструкція усуває більшу її частину і що будувати, коли однієї бази справді замало.

Застосуйте досвід до своєї системи

Робота, описана в «Продати останній квиток рівно один раз», може допомогти з наступною зміною. Ми можемо дослідити подібну поведінку вашого програмного забезпечення, даних або інфраструктури. Структура, що відповідає реальній роботі бізнесу, тож потрібні згодом запитання до даних усе ще можна буде поставити.

Переглянути наші послуги Зв'язатися з нами →

Простежте міркування та їхні межі

Прочитайте міркування та їхні межі в цій нотатці. Ми можемо перевірити, чи справджується той самий діагноз у вашому середовищі. Спроєктувала й підтримувала 6 пайплайнів ETL/ELT — 6 пайплайнів ETL на трьох рушіях баз даних

Спроєктувала й підтримувала 6 пайплайнів ETL/ELT Зв'язатися з нами →

Один рядок, усі покупці

Двадцять мільйонів рядків, розкладених по кластеру, — розв’язана задача. Один рядок, який мусить змінити кожен запит, — ні, і додаткові машини на нього не впливають, бо рядок живе на одній із них. Розподілені сховища

прямо про це пишуть: DynamoDB дає одному розділу близько 1 000 одиниць запису на секунду, і кожен запис для одного товару потрапляє в розділ цього товару, скільки б інших розділів не було.

За цією закономірністю стоїть теорема. Бейліс і колеги називають інваріант інваріантно-конфлюентним, коли репліки можуть приймати записи незалежно одна від одної й усе одно зливатися в коректний стан. `stock >= 0` — хрестоматійний приклад інваріанта, який таким не є: дві репліки, кожна з яких бачить останню одиницю, продадуть її обидві, і жодне злиття не скасує другий продаж. Щоб ніколи не продати більше, ніж є, потрібна координація. Питання лише в тому, скільки цієї координації оплачує кожен продаж.

Число, яке все вирішує

Стеля для одного товару — це приблизно кількість продажів за одне блокування, поділена на час його утримання:

```
макс. продажів/с на товар ≈
  продажів за одне блокування ÷ час утримання блокування
```

Що тримає рядок	Стеля на рядок
Блокування, потім оплата	~2/с
Блокування, потім логіка	~100–200/с
Один умовний `UPDATE`	~500–2 000/с
Пакети по 100 на коміт	~50 000/с
Один записувач у пам’яті	~100 000+/с

Це порядки величин, а не бенчмарки: залізо зсуває цифри, але не саму картину. Більшу частину короткого блокування займає коміт, що чекає, поки журнал запишеться на диск, тому пакетування й дає виграш у два порядки: сто продажів ділять один скид на диск.

Лишаються чотири важелі, і кожен дизайн нижче тягне за один із них:

- **Тримати блокування коротше** — одна інструкція, нічого мережевого всередині.
- **Робити більше за одне блокування** — пакетувати списання.
- **Мати більше блокувань** — розділити запас між кількома ключами.
- **Надсилати до блокування менше запитів** — відвертати більшість покупців, перш ніж вони до нього дійдуть.

  Останній важить більше, ніж здається. З мільйоном покупців і десятьма тисячами одиниць 990 000 запитів мусять отримати відмову. Відмови — це більша частина трафіку, тож «розпродано» має бути найдешевшою відповіддю системи: зала очікування перед розпродажем, шлюз, що пропускає приблизно стільки покупців, скільки є одиниць, прапорець `sold_out`, який CDN віддає, нікого не питаючи, і обмеження частоти запитів на обліковий запис.

Спершу виправте одну базу

Версія, яка ледве повзе або продає більше, ніж є, зазвичай має такий вигляд:

```
BEGIN;
SELECT available FROM inventory
 WHERE item_id = :item FOR UPDATE;
-- застосунок перевіряє число
-- і, можливо, викликає платіжний сервіс
UPDATE inventory SET available = :new
 WHERE item_id = :item;
COMMIT;
```

Блокування правильне. Неправильне те, що воно охоплює: кожна мілісекунда, яку застосунок проводить між `SELECT` і `UPDATE`, — це мілісекунда, яку чекає наступний покупець. Без `FOR UPDATE` гірше. Двоє покупців читають 1, обидва пишуть 0, і одну одиницю продано двічі.

Виправлення — об’єднати перевірку і списання в одну інструкцію:

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

```
   AND available >= :qty
RETURNING available;
-- повернувся рядок: зарезервовано. жодного: розпродано.
```

У PostgreSQL на рівні READ COMMITTED другий покупець, який дістався рядка, поки його тримає перший, чекає, а тоді знову перевіряє умову `WHERE` на рядку, який закомітив перший. Якщо той продаж забрав останню одиницю, умова не виконується, і другий покупець не отримує жодного рядка — розпродано — замість від'ємного числа. На REPEATABLE READ чи SERIALIZABLE другий покупець натомість отримує помилку серіалізації: це безпечно, але перетворює конкуренцію на повтори; READ COMMITTED — саме той рівень, під який написано цю інструкцію. InnoDB у MySQL дає той самий результат, бо `UPDATE` там читає останній закомічений рядок на будь-якому рівні. Додайте ще й обмеження, щоб помилка деінде в коді спричиняла явний збій, а не продаж запасу, якого немає:

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

Далі решта роботи в межах однієї бази, приблизно в порядку віддачі:

- **Нічого зовнішнього всередині транзакції.** Жодного виклику оплати, жодного HTTP-запиту, жодного надісланого повідомлення. Їм місце до транзакції або після неї.
- **Гарячий рядок — останнім.** Спершу вставте замовлення й усе інше, щоб блокування на рядку, за який конкурують, трималося лише впродовж останньої інструкції та коміту.
- **Жодного оптимістичного блокування на гарячому лічильнику.** Стовпець версії перетворює конкуренцію на повтори, а під час розпродажу повтори і є навантаженням.
- **Читання — з іншого місця.** «Залишилося 12» на сторінці товару може йти з репліки чи кешу, що відстає на секунду. Лише продажу потрібна основна база.
- **Дайте запасу окрему базу,** щоб розпродаж ніколи не стояв у черзі за каталогом до того самого диска.
- **Пул з'єднань.** Сто тисяч покупців мають перетворюватися на кілька десятків з'єднань через PgBouncer, а не на тисячі бекендів у черзі до одного блокування.

З місцями все інакше: кожна одиниця унікальна, тож потрібна інша інструкція. Кожне місце — це рядок, і покупець пропускає рядки, які тримає інша транзакція, замість того щоб чекати, доки їх звільнять:

```
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;
```

Резерв, що сплив, просто знову вільний, тож для правильної відповіді не потрібен жоден фоновий процес прибирання. А `now()` — це годинник бази, тож сервери застосунку, чиї годинники розходяться, не можуть розійтися в тому, коли резерв сплив.

Багатьом магазинам цього розділу й досить. Зробіть це раніше за все, що нижче, бо шардування — двері, які відчиняються в один бік: увійти легко, вийти дорого.

Спершу резерв, потім оплата

Покупець, що взяв останню одиницю й покинув форму оплати карткою, її не купив. Тож продаж — це не один запис, а короткий життєвий цикл:

```
AVAILABLE → HELD, на хвилини → SOLD
                             → RELEASED
```

- **Резерв прив'язаний до запиту, а не до спроби.** Кожне оформлення має ключ ідемпотентності, а таблиця резервів має `UNIQUE (tenant_id, idempotency_key)`, тож повторний клік отримує вже наявний резерв, а не ще одну одиницю. Тут є підступна деталь PostgreSQL: `INSERT ... ON CONFLICT DO NOTHING RETURNING id` нічого не повертає, коли рядок уже існував, тож «рядок не повернувся» означає «знайди його», а не «не вдалося».
- **Кожен перехід умовний.** Підтвердження продажу переводить резерв із `held` у `sold` лише якщо він досі `held`, а звільнення повертає запас, лише якщо його власне оновлення змінило рівно один рядок. Звільнення, що спрацювало двічі — скажімо, тайм-аут і кнопка скасування, — повертає одиницю один раз.
- **Резерви спливають за хвилини.** Зазвичай за п'ять–десять. Продовжте резерв один раз, коли починається оплата, щоб покупець із повільною оплатою не втратив одиницю, за яку саме платить.
- **Обмежте кількість одиниць на покупця вже під час резерву,** щоб один обліковий запис не тримав увесь запас, поки вагається.

В основі всього цього — один рядок, найкорисніший у всьому дизайні:

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

Рівність справджується після кожної транзакції, її перевіряє один запит, і коли вона перестає справджуватися, десь продали зайве чи загубили одиницю, — і ви дізнаєтеся про це раніше за клієнта.

Коли однієї бази замало

Іноді її справді замало: багато орендарів, багато розпродажів одночасно, кілька регіонів. Структура, яка масштабується, тримається одного правила — кожен товар має рівно одного власника свого запасу, сервіс запасів, і ніхто інший не вирішує — і шардує цей сервіс за `(tenant_id, item_id)`, щоб записи одного товару ніколи не охоплювали двох шардів. Найбільші орендарі отримують власні комірки, щоб їхній розпродаж не забирав ресурси в інших. Синхронним має бути лише резервування; усе після нього може бути повідомленням.

Для гарячого товару всередині шарда є чотири інструменти, і кожен тягне за один із важелів вище.

Шлюз попереду. Лічильник у Redis, який зменшує Lua-скрипт так, що перевірка і списання — один крок, пропускає приблизно стільки покупців, скільки є одиниць, а решту відвертає за мікросекунди. Остаточно вирішує не він. Redis реплікує асинхронно, тож після перемикання на репліку нова основна може забути частину пропусків і впустити трохи забагато. Рішення все одно ухвалює умовний `UPDATE` бази, а процес звірки відновлює шлюз за числом із бази.

Розділіть запас. Десять тисяч одиниць можна поділити на сотню сегментів по сто, кожен своїм рядком або своїм шардом, і кожного покупця скеровують в один: сто блокувань замість одного. Ідея стара — escrow-транзакції О'Ніла 1986 року, потім демаркаційний протокол Барбари та Гарсія-Моліни, потім обмежені лічильники Балегаша й колег — і єдина її справжня ціна — хвіст. Сегмент покупця спорожнів, а в інших ще є запас, тож або спробуйте інший сегмент, або перенесіть залишки, а під кінець розпродажу знову зведіть усе в один. Та сама ідея дає кожному регіону власну квоту, щоб більшості продажів не доводилося питати дозволу через океан.

Один записувач на товар. Кладіть кожен запит на товар в один розділ Kafka, який читає один споживач: він тримає число в пам'яті й комітить свої рішення пакетами. Один потік і жодних блокувань — архітектура в стилі LMAX. Запити мають ідентифікатор, і записувач відкидає той, щодо якого вже ухвалив рішення, бо черга може доставити повідомлення двічі.

Одиниці як рядки. Коли одиниць небагато і вони унікальні — місця, нумеровані видання, — створіть по рядку на кожну заздалегідь і забирайте їх через `SKIP LOCKED`, як вище, або тримайте ідентифікатори одиниць у множині Redis і беріть одну через `SPOP`. Тоді не лишається лічильника, за який битися.

Решта оформлення

Оплата, замовлення й квитанція — окремі системи, і жодна транзакція їх не охоплює. Двофазний коміт — хрестоматійна відповідь, але тут хибна: він блокує, поки будь-який учасник повільний, додає обміни повідомленнями на найгарячіший шлях, а платіжні сервіси в ньому не беруть участі. Із задачею впорається сага — послідовність локальних кроків, кожен зі своїм кроком відкату:

```
1. резерв запасу          відкат: звільнити резерв
2. замовлення в очікуванні  відкат: скасувати його
3. продовжити резерв      (спливе сам)
4. авторизувати оплату    відкат: анулювати
5. підтвердити HELD → SOLD
6. списати оплату
```

Оскільки оплату спершу авторизують, а списують наприкінці, платіж, що надійшов уже після того, як резерв сплив, анулюють, а не повертають: гроші так і не було списано. Кожен крок записує своє повідомлення в таблицю outbox у тій самій транзакції, що й зміну, про яку воно сповіщає, а change data capture його публікує, тож між «збережено» і «надіслано» нічого не губиться. Кожен споживач ідемпотентний, з ідентифікатором замовлення як ключем. Повідомлення приходять щонайменше один раз, а завдяки ідемпотентності кожне фактично обробляється один раз.

Розподіленому блокуванню тут не місце ніде. Думка Клеппмана в тому, що власник блокування може зависнути довше, ніж триває його оренда — збирання сміття, призупинена віртуальна машина, — і далі писати, а ліки від цього — fencing-токен, який перевіряє сховище. Але сховище, яке вміє перевірити токен, уміє перевірити й `available >= :qty`, і тоді блокування йому ніколи не було потрібне.

Поставте товар на паузу, але не продавайте понад залишок

Узгодженість і доступність обирають для кожного питання, а не для системи загалом. «Чи можна продати цю одиницю?» мусить бути узгодженим: коли власник запасу товару недосяжний, цей товар перестає продаватися. Усе інше — каталог, позначка «залишилося 3», історія замовлень — може лишатися доступним і трохи застарілим.

Найнепомітніше зайву одиницю продають під час перемикання на репліку. За асинхронної реплікації основна база підтверджує продаж і падає, перш ніж репліка його отримала; репліку підвищують, продажу немає, і одиницю продають знову. Виправлення — надійно записати продаж більш ніж на одну машину, перш ніж його підтвердити: синхронний standby у PostgreSQL (`synchronous_commit` із `synchronous_standby_names`) або сховище на Raft. Це додає

затримку до кожного коміту, а отже, подовжує саме той час утримання блокування, на який ділять у формулі стелі, — і пакетування та шлюз цю затримку компенсують. І обмежте паузу: шард, що впав, має зупиняти товари на ньому, а не весь магазин.

Як це довести

Усе це варто перевірити самим — і наші слова теж.

- **Стежте за інваріантом.** Запускайте `available + held + sold = total` за розкладом, будуйте графік і піднімайте тривогу на будь-яке розходження. Нуль продажів понад залишок — це ціль рівня обслуговування (SLO) з алертом, а не надія.
- **Влаштуйте гонку навмисно.** Десять тисяч одночасних покупців на сто одиниць, у тестах, з перевіркою: продано рівно сто і жодного числа нижче нуля.
- **Ламайте все під час гонки.** Вбийте основну базу, розріжте мережу, призупиніть процес — у дусі Jepsen — і перевірте інваріант після цього.
- **Перевірте відмови під навантаженням.** Шлях «розпродано» мусить лишатися швидким, коли це 99% трафіку, і периферія мережі (edge) мусить його витримати.
- **Зберіть метрики на одній панелі:** очікування блокувань, резерви, що спливли, і розрив між числом шлюзу та числом бази.

У якому порядку

- Спершу виміряйте. `pg_stat_statements` і очікування блокувань покажуть, чи проблема взагалі в гарячому рядку.
- Зробіть запис однією умовною інструкцією.
- Скиньте навантаження: зала очікування, шлюз, закешоване «розпродано».
- Дайте запасу власний сервіс і власного власника.
- Шардуйте за `(tenant_id, item_id)`, і лише якщо перших чотирьох замало.
- Додайте режим гарячого товару, сегменти чи один записувач, для розпродажу, про який ви знаєте заздалегідь.
- Регіони — наостанок, із квотами.

Що ми порадили б тим, хто йде слідом

- **Шардування відповідає на інше питання.** Воно розносить багато рядків, а розпродаж, що продав більше, ніж мав, — це один рядок. З'ясуйте, що з цього у вас, перш ніж будувати.
- **Зробіть перевірку і списання однією інструкцією.** Більша частина відстані між «усе гаразд» і «продано більше, ніж є» — це час між читанням числа і його записом.
- **Більша частина трафіку — це відповідь «ні».** Зробіть відмову найдешевшою операцією системи.
- **Краще пауза, ніж продаж понад залишок.** Товар, що на тридцять секунд перестав продаватися, — це вибачення. Товар, проданий двічі, — це повернення коштів, черга в підтримці й клієнт, який запам'ятає.
- **Запишіть інваріант і стежте за ним.** Це один рядок, і лише він показує, що дизайн працює в продакшні, а не тільки на рев'ю.

Джерела

- PostgreSQL: ізоляція Read Committed
- PostgreSQL: `FOR UPDATE` і `SKIP LOCKED`
- PostgreSQL: синхронна реплікація
- 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: проєктування ключів розділу
- DynamoDB: розділи й розподіл даних
- microservices.io: Saga
- microservices.io: Transactional outbox
- Redis: реплікація
- Jepsen

Усі дописи блогу

Виміряйте перед змінами

Надайте репрезентативний набір даних, запитання до нього та поточне вузьке місце. Узгодимо вимірювання якості, пропускної здатності та відновлення.

Дивитися, скільки це коштує Зв'язатися з нами →

Обговоріть свої спостереження

Обговорімо «Продати останній квиток рівно один раз» у контексті вашого проєкту. Розкажіть, що ви спостерігали і який результат був би корисним.

Переглянути наші послуги

welcome@engineer.company +45 50 10 61 84 Копенгаген, Данія

Візьміть цю сторінку із собою

Візитівка

Дбайливо зроблено з любов'ю — D.

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

Авторське право © 2025 – дотепер · Engineer ApS · Усі права захищено.

Надруковано з https://software.engineer.company/uk/notes/selling-limited-stock-without-overselling/