Engineer Софтвер

Оптимізуйте базу, перш ніж її шардувати

14 жовтня 2026 р.

Коли база сповільнюється під навантаженням, першою пропозицією зазвичай стає більше машин. Іноді це правильно. Частіше повільна частина — жменька інструкцій, яких ще ніхто не прочитав, і розкидання даних по серверах розкидає ці інструкції по більшій кількості заліза, не роблячи жодну з них дешевшою. Проблему, що здається нерозв'язною, від акуратної відповіді часто відділяє лише один план запиту, а горизонтальне масштабування іноді обирають саме для того, щоб нікому не довелося читати базу, яка у вас уже є.

Ось порядок, у якому ми працюємо, перш ніж щось ділити: подивитися, підтягнути, зняти читання з основної бази, розділити таблиці навколо гарячої точки на партиції і шардувати лише тоді, коли одна машина повністю завантажена корисною роботою. Флеш-розпродаж, де кожен покупець пише в один рядок, розібрано в нотатці Продати останній квиток рівно один раз. Ця нотатка — про ширший метод, частиною якого він є, а її останній розділ розбирає протилежну пропозицію: продавати з кожного сервера й вибачатися за одиниці, продані двічі.

Застосуйте досвід до своєї системи

Робота, описана в «Оптимізуйте базу, перш ніж її шардувати», 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 →

Спершу подивіться

PostgreSQL уже записує більшу частину того, що треба знати. Перш ніж щось змінювати, поставте їй чотири питання.

Які інструкції коштують найбільше сумарно? `pg_stat_statements`, розширення, яке вмикають через `shared_preload_libraries`, зберігає кожну нормалізовану інструкцію з кількістю викликів і часом. Сортуйте за сумою, а не за середнім: запит на 2 мс, що виконується мільйон разів на годину, коштує більше, ніж двосекундний звіт, запущений двічі.

```
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;
```

Що насправді робить перший у списку? `EXPLAIN (ANALYZE, BUFFERS)` виконує інструкцію й показує обраний план, скільки рядків кожен крок справді повернув проти очікуваних і скільки сторінок прочитав. Сканування мільйона рядків заради десяти або оцінка, що помиляється в сто разів, — зазвичай і пояснює все. `ANALYZE` виконує те, що пояснює, тож обгортайте запити на запис у `BEGIN` і `ROLLBACK`.

Хто чекає і на кого?

```
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';
```

Якщо в пікову хвилину тут є рядки, транзакції чекають одна на одну, а не на апаратні ресурси, і більша машина не усуває потреби виконувати оновлення одного спільного рядка послідовно. Додаткові сервери застосунку можуть

збільшити кількість клієнтів, що чекають у цій черзі на запис.

Сервер працює чи жонглює? Порахуйте активні сесії в `pg_stat_activity` і порівняйте з кількістю ядер. Коли активних у багато разів більше, ніж ядер, сервер витрачає час на перемикання між ними, і пул з'єднань виправить це раніше, ніж будь-яке залізо. Потім прочитайте `n_dead_tup` у `pg_stat_user_tables` : таблицю, де мертвих рядків більше, ніж живих, доводиться сканувати крізь її власне сміття: це autovacuum не встигає, а не база замала.

Запишіть, що знайшли. Це точка відліку, з якою порівнюють кожен наступний крок, і чесна відповідь на «чи треба нам шардувати?» часто вже в ній.

Підтягніть те, що є

Приблизно в порядку віддачі на вкладені зусилля.

Індекси, що відповідають питанням. Плани вище показують, де відбуваються сканування. Індекс на стовпцях, за якими фільтрує гарячий запит, перетворює сканування на пошук, а частковий індекс містить лише рядки, які запит може шукати, — скажімо, одиниці, що ще в продажу, — тож під час розпродажу він зменшується, а не росте:

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

За кожен індекс також доводиться платити під час кожного запису, тож видаліть ті, у яких `idx_scan` у `pg_stat_user_indexes` так і лишається нулем.

Типи, що кажуть, чим є значення. Схеми, написані поспіхом, людиною чи моделлю, тягнуться до `TEXT` там, де вистачило б `SMALLINT` або `BOOLEAN` . `BOOLEAN` займає один байт, `SMALLINT` — два, а `'yes'` як текст — чотири. PostgreSQL ще й доповнює кожен стовпець фіксованої ширини до його власного вирівнювання, тож випадковий порядок стовпців теж коштує байтів. Одна одиниця запасу, написана поспіхом:

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

І та сама одиниця з типами, що кажуть, чим є значення, найширші спершу:

```
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
);
```

На PostgreSQL 18 перший рядок займає 72 байти, а другий — 54, на чверть менше, тож у пам’ять уміщується більша частина таблиці. Типи також відкидають значення на кшталт `'maybe'`, від яких інакше мусив би захищатися застосунок, а обмеження `CHECK` робить те саме для діапазонів: `quantity smallint CHECK (quantity > 0)`.

Менше звернень, а робота — там, де дані. Кожна інструкція, яку надсилає застосунок, коштує звернення туди й назад і розбору. Цикл, що читає сто рядків і оновлює їх по одному, — це сто одне звернення там, де вистачило б однієї інструкції над множиною, а сторінка, що вибирає всі одиниці товару, щоб порахувати вільні, пересилає тисячі рядків заради одного числа. Наше правило: те, що мусить бачити дані, іде в базу — фільтрація, підрахунок, запис, що залежить від наявного, — а бізнес-правила лишаються в застосунку, бо процесор бази — той, якого в системі найважче додати.

Оновлення, що лишаються на своїй сторінці. Рядок, що змінюється багато разів на хвилину, щоразу лишає нову версію себе. Коли оновлення не змінює жодного проіндексованого стовпця, а на сторінці є місце, PostgreSQL кладе нову версію на ту саму сторінку й не торкається жодного індексу: це heap-only tuple, або HOT-оновлення. Тож тримайте стовпці, які змінює часте оновлення, поза індексами й лишайте місце на сторінках:

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

Налаштування діє на сторінки, записані після цього, тож задайте його до завантаження рядків. На таблиці вище всі 2 000 резервувань, що задавали `reserved`, `reserved_at` і `order_id`, були HOT, а з 2 000 оновлень, що задавали `sold`, який читає частковий індекс, — жодне. Кожну одиницю продають один раз, а резервувати її можуть багато разів. Співвідношення `n_tup_hot_upd` до `n_tup_upd` у `pg_stat_user_tables` показує, як це виглядає на живій системі.

Один рядок каже, хто був першим

Одиниця дістається першому покупцеві, чия інструкція до неї дійде, а резерв спливає за годинником бази. Його несуть два стовпці: `reserved`, булеве значення, і `reserved_at`, момент, коли його поставили.

```
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;
```

Повернувся один рядок: одиниця в цього покупця. Жодного: хтось інший був першим, або її продано. Два резерви на ту саму одиницю стають у чергу на блокування її рядка, і на рівні READ COMMITTED, типовому для PostgreSQL, другий ще раз перевіряє умову `WHERE` проти рядка, який закомітив перший, тож рядок повертається рівно одному з них. Резерв, старший за десять хвилин, вважається вільним, тож для правильної відповіді його не треба нічим прибирати.

Через це саме булеве значення — неповна відповідь. `reserved` лишається `true` після того, як резерв сплив, і лише обидва стовпці разом кажуть, чи одиницю утримують, тож кожен запит, що про це питає, мусить питати про

обидва. Періодичне прибирання, що скидає прострочені резерви, підтримує таблицю в актуальному стані для тих, хто читає лише `reserved`, але правильність ніколи на нього не чекає. Обмеження тримає пару узгодженою:

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

Нотатка про квитки зберігає кінцевий термін, `held_until`, а не початок; це той самий дизайн. Коли покупець обирає одиницю — місце, нумерований принт, — інструкція її називає, як тут. Коли підійде будь-яка, форма зі `SKIP LOCKED` з тієї нотатки вибирає ту, яку ніхто інший не утримує.

Винесіть читання з основної бази

Більшість трафіку магазину — читання: сторінка товару, «залишилося 3», кошик, історія замовлень. Жодному з цих читань не потрібна основна база — та єдина машина, яка має лишатися вільною для записів. Ці читання бере на себе потокова репліка, і додати її — справа конфігурації, тоді як шард — цілий проєкт.

Тут є два застереження. Репліка трохи відстає, тож покупець, який щойно зарезервував одиницю, а потім читає свій кошик із репліки, може не побачити резерву: читайте те, що користувач щойно записав, з основної бази або на кілька секунд після кожного запису спрямовуйте цього користувача туди. А довгий звіт на репліці можуть скасувати, коли він затримує зміни, які репліка відтворює. Документація PostgreSQL називає це конфліктами запитів і перелічує налаштування, що обмінюють відставання репліки на менше таких конфліктів.

Найдешевше читання — те, що ніколи не доходить до бази. Коли всі одиниці товару зарезервовано, ніхто не може його купити, доки резерв не спливе або його не звільнять, і база може точно сказати, коли це станеться:

```
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;
```

Коли `free` дорівнює нулю, `until` — момент, коли спливає найраніший резерв: кешуйте «повністю зарезервовано» з таким терміном, не довше, і до того часу кожного відвідувача відвертає кеш, а не база. Коли `free` дорівнює нулю, а `until` порожнє, усі одиниці продано, і відповідь остаточна. Резерв, звільнений раніше, — відхилена картка, покинутий кошик — видаляє запис у кеші, тож одиниця одразу знову в продажу.

Кеш завжди каже лише «ні». «Так» завжди дає умовний `UPDATE` вище, тож застарілий кеш може на кілька секунд завадити продажу, але ніколи не продасть одиницю двічі. Це закешоване «розпродано» з нотатки про квитки, тільки з терміном.



Поділіть таблиці на партиції навколо гарячого рядка

Партиціонування ділить одну таблицю на кілька за ключем — місяцем, орендарем — під одним ім'ям. Варто прямо сказати, чого воно не робить: воно не пришвидшує рядок, за який конкурують транзакції. Цей рядок і далі живе в одній партиції, на одній машині, за одним блокуванням.

Натомість воно тримає таблиці навколо гарячої точки малими. Замовлення, історія резервів і події ростуть безкінечно, а більшості запитів потрібна свіжа частина:

```
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');
```

Запит, що називає діапазон дат, читає лише партиції всередині нього, а індекси цього місяця лишаються досить малими, щоб жити в пам'яті. Видалення старих даних перестає бути `DELETE` мільйонів рядків із подальшим розбуханням і вакуумом і стає однією інструкцією:

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

Первинний ключ мусить містити ключ партиціонування, тому він складається з `(id, created_at)`. Запит, що не називає ключа, читає всі партиції, тож розділяйте за тим, за чим запити вже фільтрують.

Коли шардування доречне

Є одна межа, якої шардування не зсуне: кожен товар має одного власника. Одиницю не можна зарезервувати на двох серверах одночасно, тож шардування переносить товари цілими, ніколи не половину одного. Коли один товар переростає свою машину, сегменти з нотатки про квитки ділять його запас на менші частини, і кожна частина знову має рівно одного власника.

У цих межах ми вважаємо шардування доречним, коли справджується одна з таких умов, і не раніше:

- **Одна машина повністю завантажена корисною роботою.** Інструкції вже ощадні — один умовний запис кожна, нічого зовнішнього всередині транзакції, оновлення, що лишаються на своїй сторінці, — а процесор або журнал WAL основної бази все одно завантажені до межі — на найбільшій машині, яку варто купувати, і з навантаженням на багато товарів, а не на один. Один гарячий товар — це задача для сегментів, а не для шардів.

- **Дані переросли своє обслуговування.** Резервну копію не вдається відновити за час, обіцяний бізнесу, індекс не встигає побудуватися чи таблиця пройти вакуум у тихі години, або робочий набір більше не вміщується в пам'ять.
- **Вимога — ізоляція.** Орендар, чий договір чи регулятор вимагає, щоб його дані жили окремо, або чий розпродаж не може ділити диск ні з чиїм іншим.

Ціна лягає на все інше. Міграції, резервне копіювання, відновлення та звіти мають враховувати окремі шарди; join між шардами може перейти в код застосунку; а запит, що не знає свого ключа шарда, може потребувати звернення до всіх шардів. Саме цього рахунку й допомагають уникнути розділи вище, і коли його сплачено, повернути назад важко.



Коли вибачення дешевше за блокування

Є серйозний аргумент на користь протилежного шляху. Нехай кожен сервер продає із запасу, який бачить, змириться з тим, що час від часу одиниця дістанеться двом покупцям, і нехай служба підтримки поверне гроші одному з них разом із вибаченням. Бізнес-логіка стає узгодженою з часом (eventual consistency), збудованою для тих 99,9% запитів, що ніколи не зіткнуться, а асинхронний цикл згодом знаходить зіткнення й виправляє їх. Натомість магазинові ніколи не доводиться казати «спробуйте ще раз». Найсильнішу форму цього аргументу дають Гелланд і Кемпбелл у *Building on Quicksand*: система, що підтверджує роботу до того, як її отримав резервний сервер, робить кожну обіцянку ймовірнісною, а бізнес завжди залагоджував обіцянки, яких не зміг дотримати, вибаченнями.

Ми згодні з більшою частиною цього. Ми не згодні щодо того, де це витрачати.

Більша частина магазину має бути узгодженою з часом. Каталог, пошуковий індекс, позначка «залишилося 3», кошик, рекомендації, листи, фід для складу й звіти можуть відставати на секунди, а змушувати їх чекати на основну базу — це втрата швидкості без жодного виграшу. Dynamo збудували саме для цього: кошик Amazon ніколи не забуває й не відхиляє «додати в кошик», навіть коли машини відмовляють, а ціна в тому, що злиття може повернути товар, який покупець видалив. Після резерву оформлення з нотатки про квитки вже асинхронне: сага з локальних кроків, повідомлення,

що приходять щонайменше один раз, і ідемпотентні споживачі із замовленням як ключем. Цикл звірки потрібен у будь-якому дизайні, що працює з платіжним сервісом, бо вебхуки губляться.

Резерв — єдине місце, де це не працює. «Рідко» передбачає, що зіткнення рідкісні, а у флеш-розпродажі зіткнення і є розпродажем. Сервер, що бачить лише власні дані, продає далі, доки не дізнається, що продали інші, тож кількість проданого понад залишок приблизно дорівнює частоті спроб, помноженій на відставання реплікації. У спрощеній симуляції два регіони ділять 2 000 спроб на секунду, кожен продає за власним лічильником і дізнається про продажі іншого через 200 мілісекунд. За п'ять прогонів розпродажу на 10 000 одиниць кожен прогін продав від 191 до 205 одиниць, яких не існувало. Арифметика передбачає 200: частота спроб, помножена на відставання й на частку продажів, зроблених в іншому регіоні. З одним власником на товар та сама модель продала рівно запас. Це стається щоразу, коли запас розпродано до нуля, незалежно від того, чи щось зламалося, і припадає на кульмінацію, коли дивиться найбільше покупців.

Аптайм менший, ніж здається. Гілберт і Лінч довели, яким компромісом є узгодженість з часом: поки мережа розділена, система може або відповідати, або бути узгодженою, але не те й інше разом. PACELC Абаді додає другу половину. Коли мережа ціла, вибір — між затримкою й узгодженістю, а один власник на товар зі струнками інструкціями й так швидкий. Жоден із двох дизайнів не досягає 100%: платіжний сервіс, DNS і мережі між ними мають власні збої. У строгому дизайні ця поступка обмежена. Коли власник запасу одного товару недосяжний, цей товар перестає продаватися на ті секунди, які триває перемикання на репліку, а розділ нотатки про квитки про паузу для товару обмежує паузу товарами на машині, що відмовила. Решта магазину продає далі.

Вибачення — це не повернення коштів. Повернення — дешева частина. Покупці отримали «так» і, можливо, уже купили квитки на потяг, взяли вихідний і розповіли друзям. Почути «розпродано» на секунду зарано — це розчарування. Почути, що підтверджене замовлення відкликали, — це порушена обіцянка. Підтримка укомплектована під звичайний день, а продажі понад залишок приходять усі разом, у найнапруженішу хвилину року. Перш ніж планувати відкликання підтверджених замовлень, спитайте юриста, до чого підтвердження зобов'язує магазин.

Залиште асинхронну частину й перенесіть її до «так». У чому пропозиція має рацію, так це в тому, що покупцеві не треба чекати на блокування. Приймайте кожен клік у довговічний реплікований лог, одна партиція на товар, і кажіть покупцеві, що відповідь уже в дорозі. Тоді прийняття кліку залежить лише від логу, а лог, реплікований на три машини, і далі приймає записи, коли одна з них відмовляє. Один власник на товар читає лог по порядку й вирішує зі швидкістю пам'яті, а відповідь повертається повідомленням за кілька секунд. Коли цього власника перемикають на репліку, запити чекають у лозі, а не відхиляються й не продаються двічі. «Так» — це резерв, а оплата йде за ним, як в оформленні з нотатки про квитки, тож ніхто не платить за одиницю, якої не отримав. Звірка однаково відбувається, але до обіцянки, а не після неї.

Діліть запас, а не лічильник. Коли бізнесу потрібно, щоб регіони продавали, не питаючи один одного, дайте кожному регіону квоту запасу, і нехай він продає зі своєї квоти, ні в кого не питаючи. Регіон, чия квота вичерпалась, просить ще, а ближче до кінця розпродажу квоти знову зводяться до одного власника. Бейліс і колеги з'ясовують, які правила сервери можуть виконувати, не розмовляючи між собою, а запас, який не може опуститися нижче нуля, — не з таких. Квоти не прибирають цієї розмови; вони виносять її з кожного продажу в рідкісну мить, коли квоту поповнюють. Це дизайн, узгоджений з часом, що все ж тримає інваріант. Його єдина ціна — хвіст, коли покупцеві відмовляють в одному регіоні, хоча в іншому ще є запас.

Коли продаж понад залишок — правильна політика. Деякі товари можна зробити ще раз: друк на замовлення, футболка, яку постачальник ще має, ліцензійний ключ. Продаж понад лічильник тоді — це відкладене замовлення з чесною датою, і узгодженість з часом — правильний дизайн. Авіакомпанії навмисно продають більше квитків, ніж місць. Вони знають, скільки пасажирів не з'являється, закладають ризик у ціну й компенсують за правилом, написаним заздалегідь. Це і є тест. Продаж понад залишок прийнятний, коли бізнес обрав його розмір і спосіб виправлення до розпродажу. Це дефект, коли він падає на того, хто випадково клікнув останнім.

Якщо обираєте вибачення, будуйте їх як платежі. Цикл, який описує пропозиція, — це процес повернення коштів, і йому потрібне те саме, що й усьому, що рухає гроші:

- **Ідентифікатор для кожного продажу,** щоб повторна спроба була тим самим продажем, і журнал продажів, у який лише дописують. Число запасу виводять із журналу, а не редагують.
- **Правило, хто залишає одиницю собі,** яке ви змогли б пояснити покупцеві, що програв. Упорядковуйте за логом, а не за годинниками різних серверів, де кілька мілісекунд розбіжності вирішували б, хто отримає повернення.
- **Компенсації, ідемпотентні й записані в журнал:** скасувати, анулювати або повернути кошти, повідомити покупця. Це кроки відкату саги, які виконує машина.
- **Дедлайн.** Цикл має знайти кожен продаж понад залишок до того, як щось відвантажать, і раніше, ніж це помітить покупець.
- **Бюджет, який вимірюють.** Кількість продажів понад залишок за розпродаж проти числа, під яким бізнес підписався, поруч з інваріантом на тому самому дашборді, з алертом, коли бюджет вичерпано.

Наша відповідь: бути узгодженими з часом усюди навколо резерву, строгими в самому резерві й асинхронними в тому, як відповідь доходить до покупця. Проєктуйте для звичайного шляху, але у флеш-розпродажі зіткнення і є звичайним шляхом. Якщо бізнес готовий продати кілька одиниць понад залишок в обмін на щось, зробіть це числом, яке бізнес обрав, зі способом виправлення, написаним до розпродажу, і вимірюйте його.



Що ми порадили б тим, хто йде слідом

- **Читайте, перш ніж купувати.** `pg_stat_statements` і один `EXPLAIN` відповідають на більше питань про потужність, ніж новий сервер.
- **Нехай типи кажуть, чим є дані.** Булеве значення, збережене як `boolean`, менше, і воно відкидає нісенітниці.
- **Надсилайте базі те, що мусить бачити дані, і нічого більше.** Фільтрації й підрахунку там місце. Бізнес-правилам — ні.
- **Кеш каже «ні»; «так» каже лише база.** Тоді застарілий кеш коштує секунд, а не одиниці.
- **Партиціонування — це хатнє прибирання.** Воно тримає таблиці навколо гарячого рядка малими й нічого не робить для самого рядка.
- **Шардуйте, коли одна машина повністю завантажена корисною роботою,** а не очікуванням.

- **Звіряйте до «так», а не після.** Усе навколо резерву може бути узгодженим з часом. Сам резерв, зроблений так, продає понад залишок стільки, скільки дає частота спроб, помножена на затримку, на кожному розпродажі.

Джерела

- PostgreSQL: pg_stat_statements
- PostgreSQL: використання EXPLAIN
- PostgreSQL: система накопичувальної статистики
- PostgreSQL: числові типи
- PostgreSQL: символьні типи
- PostgreSQL: тип boolean
- PostgreSQL: часткові індекси
- PostgreSQL: heap-only tuples (HOT)
- PostgreSQL: партиціонування таблиць
- PostgreSQL: hot standby і конфлікти запитів
- Pat Helland і Dave Campbell, Building on Quicksand
- Seth Gilbert і 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 і Kenneth Salem, Sagas
- Bailis et al., Coordination Avoidance in Database Systems

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 «Оптимізуйте базу, перш ніж її шардувати» 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/optimise-before-you-shard/