Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

Оптимізація роботи СУБД PostgreSQL: від рівня сервера до рівня запитів. Демонстрація результатів на тестових даних

Текстова частина до курсової роботи
за спеціальністю 122 «Комп'ютерні науки»

Керівник курсової роботи
к.н., ст.викл. Захоженко П.О.
_____
(підпис)
"____" _____ 2021 р.
Виконав студент Смальченко Н.Г.
"____" _____ 2021 р.

Київ 2021

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав.кафедри інформатики,
Доцент., к. ф.-м. н. С.С.Гороховський
(підпис)
„____”_____20201р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

студенту Смальченко Наталії Григорівні факультету інформатики 4-го курсу
ТЕМА Оптимізація роботи СУБД PostgreSQL: від рівня сервера до рівня запитів. Демонстрація результатів на тестових даних
Зміст ТЧ до курсової роботи:
Індивідуальне завдання
Календарний план
Вступ
Прискорення наповнення БД даними
Налаштування конфігурації сервера PostgreSQL
Оптимізація на рівні запитів
Оптимізація, що загрожує стабільності
Висновок
Список використаних джерел.

Дата видачі „___” _____ 2020 р. Керівник _____
(підпис)
Завдання отримав _____
(підпис)

Календарний план виконання роботи:

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання теми курсової роботи.	23.10.2020	
2.	Огляд основниго матеріалу за темою	30.11.2020	
3.	Аналіз алгоритмів, рекомендацій	15.03.2021	
4.	Написання теоретичної частини	05.03.2021	
5.	Доповнення тестовими прикладами	20.03.2021	
6.	Написання останнього розділу	03.04.2021	
7.	Створення презентації та написання доповіді для захисту роботи.	04.04.2021	
8.	Корегування роботи згідно із зауваженнями керівника	11.04.2021	

Студенту Смальченко Н.Г.

Керівник Захоженко П.О.

“_____”_____

Зміст

Вступ

Будь-якому користувачу і розробнику хочеться щоб їх програма працювала швидко, займала не дуже багато пам'яті і ресурсів системи, але при цьому ще і була надійною. В багатьох випадках робота програмних продуктів не обходиться без баз даних.

Оптимізація СУБД потрібна для досягнення певних цілей. До них належать:

- Збільшити швидкість виконання запитів.
- Підвищити загальну продуктивність сервера.
- Зменшити час очікування завантаження сторінок ресурсу.
- Знизити споживання серверних потужностей хостингу.
- Знизити обсяг займаного дискового простору.

Швидкість роботи, взагалі кажучи, не є основною причиною використання реляційних СУБД. Більш того, перші реляційні бази працювали повільніше своїх попередників. Вибір цієї технології був викликаний скоріше можливістю покласти підтримку цілісності даних на СУБД; незалежністю логічної структури даних від фізичної. Ці особливості дозволяють сильно спростити написання програм, але вимагають для своєї реалізації додаткових ресурсів. Таким чином, перш, ніж шукати відповідь на питання «як змусити РСУБД працювати швидше в моїй задачі?» слід відповісти на питання «чи немає більш підходящого засобу для вирішення моєї завдання, ніж РСУБД?» Іноді використання іншого засобу зажадає менше зусиль, ніж настройка продуктивності.[2]

Проте в багатьох задачах використовувати реляційну СКБД краще. Тому все ж розглянемо ці питання детальніше. Оскільки ця тема досить обширна, описати всі можливості і випробувати їх в рамках однієї роботи не вдасться. Тому розглянемо лише основне.

1 Прискорення наповнення БД даними

Коли ви поступово додаєте дані невеликими частинками, майже непомітно те, що це працює не так швидко, як хотілось би. Для цього існують різні способи. Можна змінити сам скрипт вставлення, а можна змінити конфігурацію сервера. Є ще певні способи розглянемо їх всі.

1.1 Зменшення швидкості наповнення БД шляхом зміни конфігурації сервера

Прискорити завантаження великих обсягів даних можна, збільшивши параметр конфігурації maintenance_work_mem на час завантаження. Це призведе до збільшення швидкодії команд створення індексів та зовнішніх ключів: CREATE INDEX і ALTER TABLE ADD FOREIGN KEY. Проте це не вплине на швидкість самої команди COPY .[7]

Також масову завантаження даних можна прискорити, змінивши на час завантаження параметр конфігурації max_wal_size. Завантажуючи великі обсяги даних, PostgreSQL змушений збільшувати частоту контрольних точок в порівнянні зі звичайною (яка задається параметром checkpoint_timeout), а значить і частіше буде скидати «брудні» сторінки на диск. Тимчасово збільшивши параметр max_wal_size, можна зменшити частоту контрольних точок і пов'язаних з ними операцій введення-виведення.

Крім того, для завантаження великих обсягів даних в середовищі, де використовується архівація WAL або потоковая реплікація, швидше буде зробити копію бази даних після завантаження даних, ніж обробляти безліч операцій змін в WAL. Щоб відключити передачу змін через WAL в процесі завантаження, вимкніть архівацію і потокову реплікацію. Для цього треба встановити наступні значення для параметрів:

- wal_level = minimal

- archive_mode = off
- max_wal_senders = 0

Але при цьому треба розуміти, що змінені параметри вступлять в силу тільки після перезапуску сервера.

Це не тільки допоможе заощадити час архівації та передачі WAL, але і безпосередньо прискорить деякі команди, які можуть зовсім не використовувати WAL, якщо wal_level дорівнює minimal. Такі команди можуть гарантувати безпеку при збої, без запису всі зміни в WAL, а виконавши тільки fsync в кінці операції, що буде набагато дешевше. Це відноситься до наступних команд:

• CREATE TABLE AS SELECT

• CREATE INDEX (і подібні команди, як наприклад ALTER TABLE ADD PRIMARY KEY)

• ALTER TABLE SET TABLESPACE

• CLUSTER

• COPY FROM, коли цільова таблиця була створена або спустошена раніше в тій же транзакції. [7]

1.2 Зменшення швидкості наповнення БД шляхом зміни SQL-запиту

Команда INSERT є транзакцією, тому після кожного її виконання відбувається її фіксація. Це досить затратна операція, Щоб цього уникнути можна переписати запит, у якому для вставки окремого рядка таблиці використовується окрема команда INSERT з використанням тільки однієї команди для багатьох рядків.

Іншим способом є додавання команд BEGIN до початку процеса вставки, і COMMIT після нього. Деякі клієнтські бібліотеки можуть робити це

автоматично, в таких випадках потрібно переконатися, що це так. Виконувати всі операції в одній транзакції добре ще й тому, що в разі помилки додавання однієї з рядків відбудеться відкат до вихідного стану і ви не опинитеся в складній ситуації з частково завантаженими даними.

Ще одним варіантом пришвидшення є використання команди COPY, замість серії INSERT. Команда COPY оптимізована для завантаження великої кількості рядків; хоча вона не так гнучка, як INSERT, але при завантаженні великих обсягів даних вона тягне набагато менше зайвих витрат. Так як COPY - це одна команда, застосовуючи її, немає необхідності відключати автофіксації транзакцій.

Але є випадки, коли COPY не підходить для використання, тоді може бути корисно створити підготовлений оператор INSERT за допомогою PREPARE, а потім виконувати EXECUTE стільки раз, скільки буде потрібно. Це дозволить уникнути накладних витрат, пов'язаних з розглядом та аналізом кожної команди INSERT. У різних інтерфейсах це може виглядати по-різному; за подробицями зверніться до опису «підготовлених операторів» в документації конкретного інтерфейсу.

Але треба зауважити, що за допомогою COPY велика кількість рядків практично завжди завантажується швидше, ніж за допомогою INSERT, навіть якщо використовується PREPARE і серія операцій додавання укладена в одну транзакцію.[7]

COPY працює швидше в тому випадку, якщо команда виконується в одній транзакції з командами CREATE TABLE або TRUNCATE. У таких випадках записувати WAL не потрібно, оскільки в разі помилки файли, що містять завантажувані дані, будуть все одно видалені. Однак це зауваження справедливо тільки, коли параметр wal_level дорівнює minimal, так як в противному випадку всі команди повинні записувати свої зміни в WAL.

Щоразу, коли розподіл даних в таблиці значно змінюється, рекомендується виконувати ANALYZE. Ця рекомендація стосується і завантаження в таблицю великого обсягу даних. Виконавши ANALYZE (або VACUUM ANALYZE), ви тим самим поновіте статистику по цій таблиці для планувальника. Коли планувальник не має статистики або вона не відповідає дійсності, він не зможе правильно планувати запити, що призведе до зниження швидкодії при роботі з відповідними таблицями. Проте, якщо включений потік автоочищення, він може запускати ANALYZE автоматично.

1.3 Видалення індексів та зовнішніх ключів

Якщо ви завантажуєте дані в щойно створену таблицю, швидше за все скористаєтесь командою COPY, а потім створите всі необхідні для неї індекси. На створення індексу для вже існуючих даних піде менше часу, ніж на послідовне його оновлення при додаванні кожного рядка.

Якщо ви додаєте дані в існуючу таблицю, може мати сенс видалити індекси, завантажити таблицю, а потім перебудувати індекси. Звичайно, при цьому треба враховувати, що тимчасова відсутність індексів може негативно вплинути на швидкість роботи інших користувачів. Крім того, слід двічі подумати, перш ніж видаляти унікальні індекси, так як без них відповідні перевірки ключів не будуть виконуватися.

Як і з індексами, перевірки, пов'язані з обмеженнями зовнішніх ключів, вигідніше виконувати «масово», а не для кожного рядка окремо. Тому може бути корисно видалити зовнішні ключі, завантажити дані, а потім відновити колишні обмеження. І в цьому випадку теж доводиться вибирати між швидкістю завантаження даних і ризиком допустити помилки під час відсутності обмежень.

Більш того, коли ви завантажуєте дані в таблицю з існуючими обмеженнями зовнішнього ключа, для кожної нової рядки додається запис в чергу подій тригера (так як саме спрацьовує тригер перевіряє такі обмеження для рядка). При завантаженні багатьох мільйонів рядків в чергу подій тригера може використати усю вільну пам'ять, що призведе до неприпустимою навантаження на файл підкачки або навіть до збою команди. Таким чином, завантажуючи великі обсяги даних, може бути не просто бажано, а необхідно видаляти, а потім відновлювати зовнішні ключі. Якщо ж тимчасове відключення цього обмеження неприйнятно, єдино можливим рішенням може бути поділ всієї операції завантаження на менші транзакції.[7]

1.4 Певні зауваження щодо відновлення даних

Якщо ми відновити дані, використовуються команди pg_dump або pg_restore.

У скриптах завантаження даних, які вона генерує, автоматично враховуються деякі, але не всі з цих рекомендацій. Щоб завантажити дані, які вивантажив pg_dump, максимально швидко, вам потрібно буде виконати деякі додаткові дії вручну.

За замовчуванням pg_dump використовує команду COPY і коли вона вивантажує повністю схему і дані, в створеному скрипті вона спочатку завбачливо завантажує дані, а потім створює індекси і зовнішні ключі. Так що в цьому випадку частина рекомендацій виконується автоматично. Залишається врахувати тільки наступне:

• Встановлення необхідних значеннь для maintenance_work_mem і max_wal_size.

• Якщо використовується архівація WAL або потокова реплікація, по можливості треба їх вимкнути їх на час відновлення. Для параматрів

archive_mode, wal_level та max_wal_senders треба встановити відповідні значення. Закінчивши відновлення, варто повернути їх звичайні значення і зробити свіжу резервну копію БД.

• Можна поекспериментувати з режимами паралельного копіювання і відновлення команд pg_dump і pg_restore, і підберіть оптимальне число паралельних завдань. Паралельне копіювання і відновлення даних, кероване параметром -j, має дати значний виграш в швидкості в порівнянні з послідовним режимом.

• Якщо це можливо, варто відновити всі дані в рамках однієї транзакції. Для цього передається параметр -1 або --single-transaction команді psql або pg_restore. Але треба врахувати, що в цьому режимі навіть незначна помилка приведе до відкочення всіх змін і час відновлення буде витрачений даремно. Залежно від того, наскільки взаємопов'язані дані, може бути краще "вичистити" їх вручну.

• Якщо на сервері баз даних встановлено кілька процесорів, корисним може виявитися параметр --jobs команди pg_restore. З його допомогою можна виконати завантаження даних і створення індексів паралельно.

• Після завантаження даних варто запустити ANALYZE.

1.5 Випробування на практиці

Оскільки для виконання цієї курсової роботи будуть потрібні певні тестові дані, було знайдено дві бази даних sportdb (можна знайти за посиланням: http://www.sportsstandards.org/sd/samples) та pagila (копія БД sakila для postgres; посилання: https://github.com/devrimgunduz/pagila).

Файли скриптів для завантаження даних для sportdb має розмір 15,6MB. При першій спробі завантажити на цю дію було витрачено 55 хвилин. При

цьому параметри maintenance_work_mem, wal_level replica, max_val_size, max_wal_senders, archive_mode мали наступні значення:

- maintenance_work_mem = 64MB
- wal_level = replica
- max_val_size = 1GB
- max_wal_senders = 10
- archive_mode = off

В цьому випадку завантаження даних тривало близько 55хв. Для проведення перевірки наведені параметри було змінено наступним чином:

- maintenance_work_mem = 512MB
- wal_level = minimal
- max_val_size = 2GB
- max_wal_senders = 0

Оскільки параметр archive_mode і так дорівнював off, зміни його не торкнулись.

Після всіх змін ще раз спробували завантаження тривало близько 50хв. Витрачений час зменшився на 9%.

Потім всім параметрам були повернуті початкові значення і проведено експерименти з частковою зміною SQL файлу. В нього було додано команду BEGIN перед групою команд INSERT і команду COMMIT після неї. В цьому випадку база завантажувалась близько хвилини і навідміну від попередніх разів часу на виконання команд INSERT було використано манше ніж на решту команд, до змін майже весь час виконувались команди INSERT. Значне покращення після невеликої зміни говорить саме за себе.

Щоб протестувати швидкість комади COPY було використано БД pagila. Якщо просто порвіняти розмір двох файлів з використанням команди INSERT (5.4MB) і з використанням команди COPY (3.0MB) стає очевидно, що для перенесення великої БД краще вивантажити дані з командою COPY хоча б задля економії місця на диску.

Результати експерименту такі:

- З використанням команди INSERT - 28хв
- З використанням команди INSERT, BEGIN та COMMIT - 23с
- З використанням COPY - 17с

Отже найбільше вплинуло саме зміна запиту дл зменшення дій фіксації транзакцій.

2 Налаштування конфігурації сервера PostgreSQL

Після проектування бази даних у розробника вже будуть певні вимоги до кількості та розміру таблиць, до запису та читання інформації. Залежно від цього потрібно налаштувати сервер для власних потреб. Для такого налаштування зазвичай використовується конфігураційний файл, який має повну адресу “/etc/postgresql/12/main/postgresql.conf”. Замість числа “12” може бути інша версія сервера. При цьому після зміни конфігураційного файлу необхідно зупинити сервер postgres і потім запустити знову задля того, щоб нові параметри вступили в силу, оскільки файл postgresql.conf зчитується один раз перед запуском сервера.

Розглянемо основні параметри, які можуть допомогти покращити продуктивність роботи з базою даних в PostgreSQL.

2.1 Параметр shared_buffers.

PostgreSQL використовує свій власний буфер, а також використовує буферизоване ядро вводу-виводу. Це означає, що дані зберігаються в пам'яті двічі, спочатку в буфері PostgreSQL, а потім в буфері ядра. На відміну від інших баз даних, PostgreSQL не забезпечує пряме введення-виведення. Це називається подвійний буферизацією. Буфер PostgreSQL називається shared_buffer. Цей параметр є неефективно налаштованим параметром для більшості сучасних операційних систем. Цей параметр встановлює, скільки виділеної пам'яті буде використовуватися PostgreSQL для кешування.

Значення за замовчуванням для shared_buffer встановлено дуже низьким, і майже не впливає на роботу сервера. Це зроблено через те, що деякі комп’ютери і операційні системи не підтримують більш високі значення. Але в більшості сучасних комп’ютерів необхідно збільшити це значення для кращої продуктивності.

Якщо використовується сервер з об'ємом оперативної пам'яті 1 ГБ і більше, розумним початковим значенням shared_buffers буде 25% від обсягу пам'яті. Існують варіанти навантаження, при яких ефективні будуть і ще більші значення shared_buffers, але так як PostgreSQL використовує і кеш операційної системи, виділяти для shared_buffers більше 40% ОЗУ навряд чи буде корисно. При збільшенні shared_buffers зазвичай потрібно відповідно збільшити max_wal_size, щоб розтягнути процес запису великого обсягу нових або змінених даних на більш тривалий час.

У системах з об'ємом ОЗУ менше 1 ГБ варто обмежитися меншим відсотком ОЗУ, щоб залишити достатньо місця операційній системі. [11]

Якщо розмір робочого набору даних невеликий і вони можуть легко поміститися в вашу оперативну пам'ять, можна збільшити значення shared_buffer, щоб воно містило всю вашу базу даних і щоб весь робочий набір даних міг перебувати в кеші. Тим не менш, резервувати всю оперативну

пам'ять для PostgreSQL не варто, оскільки певна частина має бути відведена для роботи системи і самого сервера СКБД.

У виробничих середовищах велике значення для shared_buffer дійсно дає хорошу продуктивність, хоча для досягнення правильного балансу завжди слід проводити тести.[8]

Для перевірки значення shared_buffer використовуються наступні команди:

```
postgres=# SHOW shared_buffers;
 shared_buffers
----------------
 128MB
(1 row)
```

2.2 Параметр wal_buffers

PostgreSQL спочатку записує записи в WAL(журнал передзаписів) в буфери, а потім ці буфери записуються на диск. Розмір буфера за замовчуванням, визначений параметром wal_buffers і становить 16 МБ. Але якщо при роботі з БД багато одночасних підключень, то більш високе значення цього параметра може підвищити продуктивність.

```
postgres=# SHOW wal_buffers;
 wal_buffers
-------------
 16MB
(1 row)
```

Вміст буферів WAL записується на диск при фіксуванні кожної транзакції, тому дуже великі значення навряд чи принесуть значну користь. Однак значення як мінімум в декілька мегабайт може збільшити швидкодію при записі

на навантаженому сервері, коли відразу безліч клієнтів фіксують транзакції. Автоналаштування, що діє при значенні за замовчуванням (-1), в більшості випадків обирає розумні значення. [3]

2.3 Параметр effective_cache_size

effective_cache_size надає оцінку пам'яті, доступної для кешування диска. Цей параметр не вказує скільки фактично виділено пам’яті чи кеша, але повідомляє оптимізатору обсяг кеша, який доступний в ядрі. Якщо значення цього параметра встановлено дуже низьким, планувальник запитів може прийняти рішення не використовувати деякі індекси, навіть якщо вони будуть корисні. Тому установка великого значення буде необхідною.

```
postgres=# SHOW effective_cache_size;
 effective_cache_size
----------------------
 4GB
(1 row)
```

Зважаючи на розмір цього параметра, характеристики системи і на невеликі розміри БД, що зберігаються в СКБД, зміна параметра не тестувалась через неможливість виявити зміни.

2.4 Параметр work_mem

Цей параметр використовується для складного сортування. Якщо потрібно виконати складне сортування, варто збільшити значення work_mem для отримання гарних результатів. Сортування в пам'яті відбувається набагато швидше, ніж сортування даних на диску.

Треба звернути увагу, що складних запитах одночасно можуть виконуватися декілька операцій сортування або хеширования, і при цьому зазначений обсяг пам'яті може використовуватися в кожній операції, перш ніж дані почнуть витіснятися в тимчасові файли. Крім того, такі операції можуть виконуватися одночасно в різних сеансах. Таким чином, загальний обсяг пам'яті може багаторазово перевершувати значення work_mem; це слід враховувати, вибираючи відповідне значення. Операції сортування використовуються для ORDER BY, DISTINCT і з'єднань злиттям. Хеш-таблиці використовуються при з'єднаннях і агрегування по хешу, а також обробці підзапитів IN із застосуванням хешу.

Установка цього параметра глобальним може привести до дуже високого використання пам'яті. Тому рекомендується змінювати його на рівні сеансу.

```
postgres=# SHOW work_mem;
 work_mem
----------
 4MB
(1 row)
```

Зменшимо work_mem до 64kB

```
sportdb=# SET work_mem to '64';
SET
sportdb=# EXPLAIN ANALYZE SELECT * FROM stats ORDER BY stat_repository_type;
                        QUERY PLAN
-----------------------------------------------------------------
 Sort  (cost=1611.16..1634.66 rows=9398 width=67)
   Sort Key: stat_repository_type
   ->  Seq Scan on stats  (cost=0.00..213.98 rows=9398 width=67)
```

```
(3 rows)
```

Збільшимо work_mem до 64MB

```
sportdb=# SET work_mem to '64MB';
SET
sportdb=# EXPLAIN SELECT * FROM stats ORDER BY stat_repository_type;
                        QUERY PLAN
-----------------------------------------------------------------
 Sort  (cost=834.16..857.66 rows=9398 width=67)
   Sort Key: stat_repository_type
   ->  Seq Scan on stats  (cost=0.00..213.98 rows=9398 width=67)
(3 rows)
```

Як бачимо вартість операції знизиласьз 1611,16 до 834,16

2.5 Параметр maintenance_work_mem

Це параметр пам'яті, який використовується для задач обслуговування. Значення за замовчуванням становить 64 МБ. Установка великого значення допомагає в таких завданнях, як VACUUM, RESTORE, CREATE INDEX, ADD FOREIGN KEY і ALTER TABLE.

Розглянемо тестові результати:

```
sportdb=# \timing on
Timing is on.
sportdb=# SET maintenance_work_mem to '1024';
SET
sportdb=# DROP INDEX idx_stats_1;
DROP INDEX
```

```
sportdb=# CREATE INDEX idx_stats_1 ON stats USING btree (stat_repository_type);
CREATE INDEX
Time: 105,794 ms
sportdb=# SET maintenance_work_mem to '256MB';
SET
sportdb=# DROP INDEX idx_stats_1;
DROP INDEX
sportdb=# CREATE INDEX idx_stats_1 ON stats USING btree (stat_repository_type);
CREATE INDEX
Time: 78,414 ms
```

Якщо розглянемо наведені вище приклади, побачимо, що час виконання операцій становить 105,794 мс, якщо параметр maintenance_work_mem дорівнює 1 МБ, але цей час зменшується до 78,414 мс, коли ми збільшуємо значення параметра maintenance_work_mem до 256 МБ.

2.6 Параметр synchronous_commit

Цей параметр визначає, після завершення якого рівня обробки WAL сервер буде повідомляти про успішне виконання операції. Можна вибрати зі значень remote_apply (застосовано віддалено), on (вкл., За замовчуванням), remote_write (записано віддалено), local (локально) і off (викл.).

Якщо значення synchronous_standby_names не задано, для даного параметра мають сенс тільки значення on і off; з варіантами remote_apply, remote_write і local буде вибраний той же рівень синхронізації, що і з on. Локальна дія всіх відмінних від off режимів полягає в очікуванні локального скидання WAL на диск. У режимі off очікування відсутнє, тому може утворитися вікно від моменту, коли клієнт дізнається про успішне завершення,

до моменту, коли транзакція дійсно гарантовано захищена від збою. (Максимальний розмір вікна дорівнює потрійному значенню wal_writer_delay.) На відміну від fsync, значення off цього параметра не загрожує цілісності даних: збій операційної системи або бази даних може привести до втрати останніх транзакцій, які вважалися зафіксованими, але стан бази даних буде точно таким же, як і в разі штатного переривання цих транзакцій. Тому виключення режиму synchronous_commit може бути корисною альтернативою відключення fsync, коли продуктивність важливіше, ніж надійна гарантія збереження кожної транзакції.

Якщо значення synchronous_standby_names не порожнє, параметр synchronous_commit також визначає, чи повинен сервер при фіксуванні транзакції чекати, поки відповідні записи WAL будуть оброблені на відомому сервері (серверах).Зі значенням remote_apply фіксування завершується тільки після отримання відповідей від поточних синхронних ведених серверів, які говорять, що вони отримали запис про фіксування транзакції, зберегли її в надійному сховищі, а також застосували транзакцію, так що вона стала видна для запитів на цих серверах. З таким варіантом затримка при фіксуванні виявляється більше, так як необхідно чекати відтворення WAL. Зі значенням on фіксування завершується тільки після отримання відповідей від поточних синхронних ведених серверів, що підтверджують, що вони отримали запис про фіксування транзакції і передали її в надійному сховищі. Це гарантує, що транзакція не буде втрачена, якщо тільки база даних не буде пошкоджена і на провідному, і на всіх синхронних ведених серверах. Зі значенням remote_write фіксування завершується після отримання відповідей від поточних синхронних серверів, які говорять, що вони отримали запис про фіксування транзакції і зберегли її в своїх ФС. Цей варіант дозволяє гарантувати збереження даних у разі відмови веденого сервера PostgreSQL, але не у випадку збою на рівні ОС, так як дані можуть ще не досягти надійного сховища на цьому сервері. Зі значенням local фіксування завершується після локального скидання даних, не

чекаючи реплікації. Зазвичай це небажаний варіант при синхронної реплікації, але він представлений для повноти.

Цей параметр можна змінити в будь-який час; поведінка кожної конкретної транзакції визначається значенням, які у час її фіксування. Таким чином, є можливість і сенс фіксувати деякі транзакції синхронно, а інші - асинхронно. Наприклад, щоб зафіксувати одну транзакцію з декількох команд асинхронно, коли за замовчуванням обраний протилежний варіант, виконайте в цій транзакції SET LOCAL synchronous_commit TO OFF.[3]

```
sportdb=# SHOW synchronous_commit;
 synchronous_commit
--------------------
 on
(1 row)
```

2.7 Параметри checkpoint_timeout, checkpoint_completion_target

PostgreSQL записує зміни в WAL. Процес контрольної точки записує дані в файли. Ця дія виконується, коли виникає контрольна точка (CHECKPOINT). Це дорога операція, яка може викликати величезну кількість операцій вводу-виводу. Весь цей процес включає в себе дорогі операції читання і запису на диск. Користувачі завжди можуть запустити процес контрольної точки (CHECKPOINT), коли це необхідно, або автоматизувати запуск за допомогою параметрів checkpoint_timeout і checkpoint_completion_target.

Параметр checkpoint_timeout використовується для установки часу між контрольними точками WAL. Якщо буде встановлено менше значення, це зменшить час відновлення після збою, оскільки на диск записується більше даних, але це знизить продуктивність, оскільки кожна контрольна точка все ж споживає цінні системні ресурси.

checkpoint_completion_target - це значення проміжку часу між контрольними точками для завершення контрольної точки. Висока частота контрольних точок може вплинути на продуктивність. Для плавного виконання завдання контрольної точки, checkpoint_timeout повинен мати низьке значення. В іншому випадку ОС буде накопичувати всі непотрібні дані до певного значення, а потім виконає велике скидання.

checkpoint_completion_target задає цільовий час для завершення процедури контрольної точки, як коефіцієнт для загального часу між контрольними точками.[8]

```
sportdb=# SHOW checkpoint_timeout;
 checkpoint_timeout
--------------------
 5min
(1 row)
sportdb=# SHOW checkpoint_completion_target;
 checkpoint_completion_target
------------------------------
 0.5
(1 row)
```

3 Оптимізація на рівні запитів

3.1 Дослідження виконання запиту

Під час виконання будь-якого отриманий запиту, PostgreSQL розгортає для нього план виконання. Вибір правильного планування, відповідної структуру та характеристики даних, вкрай важливо для високої продуктивності, тому в системі працює складний планувальник, завдання якого - підібрати хороший

план. Дізнавшись, який план був вибраний для якого-небудь запрошення, можна за допомогою команд EXPLAIN та EXPLAIN ANALYZE. [4]

Приклади виконання цих команд:

```
sportdb=# EXPLAIN SELECT * FROM stats ORDER BY stat_repository_type;
 Index Scan using idx_stats_1 on stats  (cost=0.29..847.19 rows=9398 width=67)
sportdb=# EXPLAIN ANALYZE SELECT * FROM stats ORDER BY stat_repository_type;
 Index Scan using idx_stats_1 on stats  (cost=0.29..847.19 rows=9398 width=67) (actual time=34.902..54.626 rows=9398 loops=1)
 Planning Time: 0.294 ms
 Execution Time: 55.373 ms
```

З результату видно, що команда EXPLAIN ANALYZE показує ще додатково час планування та виконання запиту, фактичне число рядків.

Ще один приклад:

```
pagila_2=# EXPLAIN ANALYZE SELECT payment_date, amount, sum(amount) OVER (ORDER BY payment_date)
FROM (
  SELECT CAST(payment_date AS DATE) AS payment_date, SUM(amount) AS amount
  FROM payment
  GROUP BY CAST(payment_date AS DATE)
) p
ORDER BY payment_date;
 WindowAgg  (cost=534.35..537.85 rows=200 width=68) (actual time=167.070..167.183 rows=41 loops=1)
  -> Sort  (cost=534.35..534.85 rows=200 width=36) (actual time=167.028..167.042 rows=41 loops=1)
       Sort Key: ((payment_p2020_01.payment_date)::date)
```

Sort Method: quicksort Memory: 26kB

-> HashAggregate (cost=521.70..524.70 rows=200 width=36) (actual time=166.884..166.929 rows=41 loops=1)

Group Key: ((payment_p2020_01.payment_date)::date)

-> Append (cost=0.00..434.66 rows=17409 width=10) (actual time=102.721..156.253 rows=16049 loops=1)

-> Seq Scan on payment_p2020_01 (cost=0.00..23.46 rows=1157 width=10) (actual time=102.719..104.408 rows=1157 loops=1)

-> Seq Scan on payment_p2020_02 (cost=0.00..45.90 rows=2312 width=10) (actual time=10.302..12.798 rows=2312 loops=1)

-> Seq Scan on payment_p2020_03 (cost=0.00..112.55 rows=5644 width=10) (actual time=5.059..10.059 rows=5644 loops=1)

-> Seq Scan on payment_p2020_04 (cost=0.00..134.43 rows=6754 width=10) (actual time=0.653..6.308 rows=6754 loops=1)

-> Seq Scan on payment_p2020_05 (cost=0.00..4.28 rows=182 width=10) (actual time=19.584..19.729 rows=182 loops=1)

-> Seq Scan on payment_p2020_06 (cost=0.00..27.00 rows=1360 width=10) (actual time=0.008..0.008 rows=0 loops=1)

Planning Time: 0.523 ms

Execution Time: 167.342 ms

В цьому прикладі використаний більш складний запит, тому ми побачили більший план виконання

У цих прикладах використовується текстовий формат виводу EXPLAIN, прийнятий за умовою, як більш компактний та зручний для сприйняття людиною. Якщо ви виведете EXPLAIN, потрібно передати яку-небудь програму для подальшого аналізу, краще використовувати один із машинно-орієнтованих форматів (XML, JSON або YAML). [4]

3.2 ANLYZE та VACUUM ANALYZE

VACUUM вивільняє простір, pfqyznbq «мертвими» кортежами. При звичайних операціях Postgres кортежі, вилучені або застарілі в результаті оновлення, фізично не видаляються з таблиці; вони зберігаються в ній, поки не буде виконана команда VACUUM. Таким чином, періодично необхідно виконувати VACUUM, особливо для часто таблиць, які часто змінюються.

Без параметра команда VACUUM обробляє всі таблиці в поточній базі даних, які може очистити поточний користувач. Якщо в параметрі передається ім'я таблиці, VACUUM обробляє тільки цю таблицю.

VACUUM ANALYZE виконує очистку (VACUUM), а потім аналіз (ANALYZE) всіх зазначених таблиць. Це зручна комбінація для регулярного обслуговування БД. [5]

ANALYZE збирає статистичну інформацію про вміст таблиць в базі даних і зберігає результати в системному каталозі pg_statistic. Згодом планувальник запитів буде використовувати цю статистику для вибору найбільш ефективних планів виконання запитів. Рекомендується виконувати ANALYZE після зміни структури БД.

Без списку таблиць і стовпців команда ANALYZE обробляє всі таблиці і матеріалізовані представлення в поточній базі даних, на аналіз яких поточний користувач має право. Зі списком ANALYZE обробляє тільки зазначені в ньому таблиці. Додатково можна задати для таблиці список імен стовпців, в цьому випадку статистика буде збиратися тільки за цими стовпцями.[5]

Хоча в деяких випадках запити можуть працювати гірше після використання цієї команди. Наприклад:

До використання ANALYZE:

```
pagila_3=# EXPLAIN SELECT *
FROM (
```

```
  SELECT CAST(payment_date AS DATE) AS payment_date, SUM(amount) AS amount
  FROM payment
  GROUP BY CAST(payment_date AS DATE)
)
ORDER BY payment_date;
 Sort  (cost=532.32..532.82 rows=200 width=36)
   Sort Key: ((payment_p2020_01.payment_date)::date)
   ->  HashAggregate  (cost=519.68..522.68 rows=200 width=36)
         Group Key: ((payment_p2020_01.payment_date)::date)
         ->  Append  (cost=0.00..433.08 rows=17319 width=10)
               ->  Seq Scan on payment_p2020_01  (cost=0.00..23.46 rows=1157 width=10)
               ->  Seq Scan on payment_p2020_02  (cost=0.00..45.90 rows=2312 width=10)
               ->  Seq Scan on payment_p2020_03  (cost=0.00..112.55 rows=5644 width=10)
               ->  Seq Scan on payment_p2020_04  (cost=0.00..134.43 rows=6754 width=10)
               ->  Seq Scan on payment_p2020_05  (cost=0.00..4.28 rows=182 width=10)
               ->  Seq Scan on payment_p2020_06  (cost=0.00..25.88 rows=1270 width=10)
```

Після використання ANALYZE:

```
pagila_3=# EXPLAIN SELECT *
FROM (
  SELECT CAST(payment_date AS DATE) AS payment_date, SUM(amount) AS amount
  FROM payment
  GROUP BY CAST(payment_date AS DATE)
)
ORDER BY payment_date;
 GroupAggregate  (cost=1652.35..2038.27 rows=17069 width=36)
```

```
Group Key: ((payment_p2020_01.payment_date)::date)
->  Sort  (cost=1652.35..1695.64 rows=17319 width=10)
      Sort Key: ((payment_p2020_01.payment_date)::date)
      ->  Append  (cost=0.00..433.08 rows=17319 width=10)
            ->  Seq Scan on payment_p2020_01  (cost=0.00..23.46 rows=1157 width=10)
            ->  Seq Scan on payment_p2020_02  (cost=0.00..45.90 rows=2312 width=10)
            ->  Seq Scan on payment_p2020_03  (cost=0.00..112.55 rows=5644 width=10)
            ->  Seq Scan on payment_p2020_04  (cost=0.00..134.43 rows=6754 width=10)
            ->  Seq Scan on payment_p2020_05  (cost=0.00..4.28 rows=182 width=10)
            ->  Seq Scan on payment_p2020_06  (cost=0.00..25.88 rows=1270 width=10)
```

3.3 Використання індексів

Використання правильних індексів може значно пришвидшити запит. Наприклад:

До створення індекса

```
sportdb=# EXPLAIN SELECT * FROM stats ORDER BY stat_repository_type;
 Sort  (cost=834.16..857.66 rows=9398 width=67)
   Sort Key: stat_repository_type
   ->  Seq Scan on stats  (cost=0.00..213.98 rows=9398 width=67)
```

Після створення індекса

```
sportdb=# EXPLAIN SELECT * FROM stats ORDER BY stat_repository_type;
 Index Scan using idx_stats_1 on stats  (cost=0.29..847.19 rows=9398 width=67)
```

Але при їх використанні варто дотримуватись наступних правил:

Індекс для полів використовуваних в "JOIN", "ORDER BY" і "GROUP BY", "MAX ()" і "MIN ()" не менш важливий, ніж індекс для полів в "WHERE"

умовах (в PostgreSQL для "ORDER BY ", MIN (), MAX () часто використовується" Seq Scan "замість індексу, кожен випадок треба розглядати використовуючи EXPLAIN, іноді використання LIMIT допомагає" вибрати "index замість Seq Scan).

Краще не індексувати поля мають строковий тип, особливо поля типу text.

Індекс для LIKE і ~ (regex) корисний тільки коли маска не йде спочатку, тобто при вказівці '% text' індекс нічого очікувати використаний, а при 'text% буде (для regex індекс використовується тільки для масок виду' ^ text ... ').

Якщо в запиті використовуються функції LOWER / UPPER або ILIKE, то індекс буде використаний тільки якщо він був побудований з урахуванням регістра, тобто наприклад "CREATE INDEX news_ilike ON news (lower (title));"

Не слід індексувати boolean поля або числові поля мають невеликий розкид значень (флагової поля), в даному випадку індекс більше нашкодить, ніж надасть користь.

Для полів відображають дату / час і числових більше підходить btree індекс, для текстових - hash, для індексів за двома і більше полях можливо використовувати тільки tree. btree індекси краще підходять для операцій '<', '>', сортування, а hash для '=' і '<>'.

При створенні таблиці з UNIQUE і PRIMARY KEY індекси для цих полів створюються автоматично.

Оптимізатору сильно допомагає VACUUM ANALYZE при прийнятті рішення використовувати чи індекси;

На невеликих таблицях, прямий перебір буває швидше використання індексу, через зайвих дискових операцій (на невеликих таблиць оптимізатор може відмовитися від іспользоватнія індексів автоматично).

UNIQUE індекси швидше, ніж індекси не по унікальним полях.

Поля text в яких свідомо буде зберігатися великий обсяг даних (наприклад, текст статей) краще відокремити від інших атрибутів, таких як час, автор, розділ ...

Поля помічені як NOT NULL трохи економлять місце і виключають зайві перевірки.

Замість типу text краще використовувати character varying (N).

Чим менше розмір типу, тим краще.

При наявності дублюються складних і ресурсоекіх запитів можливе використання кешування через PREPARE / EXECUTE. Має сенс тільки в рамках поточної сесії (з'єднання з SQL сервером). [10]

3.4 Оптимізація запитів безпосередньо

Є ще деякі інші способи покращення запитів:

1. Використання конкретних імен стовпців після оператора select, замість «*» - це дозволить збільшити швидкість відпрацювання запиту і зменшення мережевого трафіку, оскільки не доведеться надсилати і відображати всі стовпчики.

2.Використовувати оператор IN обережно, оскільки на практиці він має низьку продуктивність і може бути ефективний тільки при використанні критеріїв фільтрації в підзапиті.

3. З'єднання таблиць в запиті також є критичним: у разі, коли з'єднання таблиць відбувається в правильному порядку, то загальне число рядків, необхідних для обробітку, значно скоротиться.

При з'єднанні основної та уточнюючої таблиць варто переконатися, що першою буде основна таблиця, в іншому випадку є ризик отримати обробку набагато більшого числа рядків, ніж необхідно.

Це можна побачити при використанні різних типів з'єднання

Приклад 1:

sportdb=# EXPLAIN SELECT * FROM (participants_events INNER JOIN persons ON persons.id=participants_events.participant_id) ORDER BY participant_id;

Merge Join (cost=0.57..526.32 rows=8700 width=221)

Merge Cond: (participants_events.participant_id = persons.id)

-> Index Scan using idx_participants_events_2 on participants_events (cost=0.29..412.46 rows=8700 width=36)

-> Index Scan using persons_id_key on persons (cost=0.28..147.34 rows=3937 width=185)

Приклад 2

sportdb=# EXPLAIN SELECT * FROM (participants_events LEFT JOIN persons ON persons.id=participants_events.participant_id) ORDER BY participant_id;

Merge Left Join (cost=0.57..526.32 rows=8700 width=221)

Merge Cond: (participants_events.participant_id = persons.id)

-> Index Scan using idx_participants_events_2 on participants_events (cost=0.29..412.46 rows=8700 width=36)

-> Index Scan using persons_id_key on persons (cost=0.28..147.34 rows=3937 width=185)

Приклад 3

sportdb=# EXPLAIN SELECT * FROM (participants_events RIGHT JOIN persons ON persons.id=participants_events.participant_id) ORDER BY participant_id;

Sort (cost=870.72..892.47 rows=8700 width=221)

Sort Key: participants_events.participant_id

-> Hash Right Join (cost=121.58..301.44 rows=8700 width=221)

Hash Cond: (participants_events.participant_id = persons.id)

-> Seq Scan on participants_events (cost=0.00..157.00 rows=8700 width=36)

-> Hash (cost=72.37..72.37 rows=3937 width=185)

-> Seq Scan on persons (cost=0.00..72.37 rows=3937 width=185)

В цьому випадку видно, що Приклад 1 і Приклад 2 працюють однаково, але Приклад 3 буде працювати набагато гірше (це можна зрозуміти із значення вартості).

4. Варто писати більш прості запити. Оптимізатор може не впоратися із занадто складними операторами. Крім того, іноді виконання декількох простих до неможливості операторів дає кращий результат у порівнянні зі складними і дозволяє домогтися кращої ефективності.

5. Варто оформлювати повторювані коди в призначену для користувача процедуру. Це може значно прискорити роботу, зменшити мережевий трафік. [1]

4 Оптимізація, що загрожує стабільності

Стабільність - це властивість бази даних, що гарантує, що результат зафіксованих транзакцій буде збережений навіть в разі збою сервера або відключення живлення. Однак забезпечується стабільність за рахунок значної додаткового навантаження. Тому, якщо ви можете відмовитися від такої гарантії, PostgreSQL можна прискорити ще більше, застосувавши такі методи оптимізації. Крім явно описаних винятків, навіть з такими змінами конфігурації при збої програмного ядра СУБД гарантія стабільності зберігається; ризик втрати або руйнування даних можливий тільки в разі раптової зупинки операційної системи.

- Помістити каталог даних кластера БД в файлову систему, розміщену в пам'яті (т. Е. В RAM-диск). Так ви виключіть всю активність введення / виведення, пов'язану з базою даних, якщо тільки розмір бази даних не перевищує обсяг вільної пам'яті (можливо, з урахуванням файлу підкачки).

- Вимкнути fsync; скидати дані на диск не потрібно.
- Вимкнути synchronous_commit; немає необхідності примусово записувати WAL на диск при фіксації кожної транзакції. Але врахуйте, це може привести до втрати транзакцій (хоча дані залишаться узгодженими) в разі збою бази даних.
- Вимкнути full_page_writes; захист від часткової записи сторінок не потрібна.
- Збільшити max_wal_size і checkpoint_timeout; це зменшить частоту контрольних точок, хоча обсяг / pg_wal при цьому зросте.
- Створити нежурналізовані таблиці для оптимізації запису в WAL (але врахуйте, що такі таблиці не захищені від збою). [9]

Висновок

У роботі було розглянуто питання оптимізації роботи СКБД PostgreSQL. Грамотна настройка і оптимізація PostgreSQL дозволяє досягти оптимально високих показників роботи сервера і додатків, розгорнутих на його основі. Цьому процесу сприяє запуск скриптів, які можуть швидко виявити проблеми, що впливають на продуктивність бази даних.

Є більше параметрів, які можна налаштувати, щоб отримати кращу продуктивність, але вони роблять менший вплив, ніж ті, які виділені тут. Зрештою, ми завжди повинні пам'ятати, що не всі параметри актуальні для всіх типів програм. Деякі програми працюють краще, під час налаштування параметрів, а деякі ні. Крім того такі налаштування дуже залежать від конкретної системи та харктеристик сервера. Всі ці налаштування краще тестувати вже з урахуванням наведеного вище.

При дослідженні вставки даних в БД було вивлено великий вплив оптимізації запита — використанні BEGIN перед групою команд INSERT та

COMMIT після неї або використання COPY замість INSERT. Хоча зміна конфігурації також мала вплив на результат.

Будь-яка СКБД намагається оптимізувати запит самостійно. Для дослідження цього використовуються команди EXPLAIN та EXPLAIN ANALYZE. Але користувачу теж не варто забувати, що СКБД може виправити не всі його прорахунки. Коли до БД підключено небагато користувачів чи БД невелика, це не так помітно, чим більше проект тим більше виявляється таких прорахунків.

Отже, враховувати треба все: конфігурацію сервера postgres, схему БД, додати індекси, обмеження, сам запит, а в деяких випадках корисним буде і покращити ресурси системи.

Список використаних джерел

1. NTA. Как оптимизировать запросы в SQL? [Електронний ресурс] / NTA. – 2020. – Режим доступу до ресурсу: https://vc.ru/newtechaudit/113408-kak-optimizirovat-zaprosy-v-sql.

2. Борзов А. PostgreSQL: настройка производительности [Електронний ресурс] / Алексей Борзов. – 2003. – Режим доступу до ресурсу: https://www.opennet.ru/docs/RUS/postgresql_tune/node1.html.

3. Журнал предзаписи [Електронний ресурс] – Режим доступу до ресурсу: https://postgrespro.ru/docs/postgrespro/12/runtime-config-wal.

4. Использование EXPLAIN [Електронний ресурс] – Режим доступу до ресурсу: https://postgrespro.ru/docs/postgresql/12/using-explain.

5. Команды SQL. ANALYZE [Електронний ресурс] – Режим доступу до ресурсу: https://postgrespro.ru/docs/postgresql/12/sql-analyze.

6. Команды SQL. VACUUM [Електронний ресурс] – Режим доступу до ресурсу: https://postgrespro.ru/docs/postgresql/12/sql-vacuum.

7. Наполнение базы данных [Електронний ресурс] – Режим доступу до ресурсу: https://postgrespro.ru/docs/postgresql/12/populate.

8. Настройка параметров PostgreSQL для оптимизации производительности [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://m.habr.com/ru/post/458952/.

9. Оптимизация, угрожающая стабильности [Електронний ресурс] – Режим доступу до ресурсу: https://postgrespro.ru/docs/postgresql/12/non-durability.

10. Оптимизация SQL запросов и борьба с deadlock [Електронний ресурс]. – 2003. – Режим доступу до ресурсу: https://www.opennet.ru/base/dev/postgresql_tune.txt.html.

11. Потребление ресурсов [Електронний ресурс] – Режим доступу до ресурсу: https://postgrespro.ru/docs/postgresql/12/runtime-config-resource.