МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
Факультет електроніки та інформаційних технологій

Кафедра комп'ютерних наук

Кваліфікаційна робота бакалавра

ОПТИМІЗАЦІЯ ПРОЦЕДУРИ РЕЗЕРВНОГО КОПІЮВАННЯ ДЛЯ СЕРВІСУ DBASS ELEPHANT

Здобувач освіти гр. ІНз – 81с Сергій ГАЛЕВИЧ

Науковий керівник,
кандидат технічних наук Борис КУЗІКОВ

Завідувач кафедри
доктор технічних наук, професор. Анатолій ДОВБИШ

Суми 2022

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

Центр заочної, дистанційної і вечірньої форм навчання

Кафедра комп'ютерних наук

Затверджую_____

Зав. кафедрою Довбиш А.С.

"_____" _____2022 р.

ЗАВДАННЯ
до кваліфікаційної роботи бакалавра

Здобувача освіти четвертого курсу, групи ІНз-81с спеціальності "122 – Комп'ютерні науки" заочної форми навчання Галевича Сергія Сергійовича.

Тема: "**Оптимізація процедури резервного копіювання для сервісу DBasS Elephant**"

Затверджена наказом по СумДУ

№_____ від _____2022 р.

Зміст пояснювальної записки: 1) аналітичний огляд методів створення резервних копій реляційних баз даних; 2) постановка завдання й формування завдань дослідження; 3) проектування підсистеми резервного копіювання DBaaS Elephant; 5) аналіз результатів та порівняння ефективності запропонованих підходів.

Дата видачі завдання "_____"_____2022 р.

Керівник випускної роботи _____Кузіков Б.О.

Завдання прийняв до виконання _____Галевич С.С.

РЕФЕРАТ

Записка: 39 стор., 9 рис., 8 табл., 1 додаток, 13 джерел.

Об'єкт дослідження —інформаціна технологія резервного копіювання реляціних баз даних.

Мета роботи — оптимізація підсистеми створення та відновлення резервних копій DBaaS Elephant за критерієм часу.

Методи дослідження — метод функціонально-статистичних випробувань.

Результати — був обраний логічний тип резервної копії, а в якості утиліт для реалізації вбудовані засоби PostgreSQL. Був проведений їхній огляд, були розглянуті переваги та недоліки кожної утиліти та сформовані параметри, які потенційно впливають на швидкодію. При огляді програми DBaaS Elephant були знайдені та розглянуті методи створення та відновлення баз даних, які в ній використовуються, а також спроектована UML діаграма взаємодії з користувачем. Було змінено та дописано код програми. Була проведена підготовка для тестування, а також розроблений скрипт для тестового створення таблиць і заповнення їх згенерованими даними. Проаналізувавши та порівнявши отримані результати, вдалося покращити швидкість створення та відновлення копії приблизно у 4 рази.

DBASS ELEPHANT, POSTGRESQL, СТВОРЕННЯ РЕЗЕРВНИХ КОПІЙ, ВІДНОВЛЕННЯ РЕЗЕРВНІХ КОПІЙ, PG_DUMP, PG_RESTORE

ЗМІСТ

ВСТУП

Одним із обов’язкових елементів розробки сучасного програмного забезпечення є забезпечення його якості. Забезпечення якості - це складний процес, що втілюється на всіх етапах створення програмного продукту, зокрема на етапі тестування. Одним із популярних підходів до тестування є реалізація End-to-End сценарію, що дозволяє покривати потреби smoke, sanity, new feature та regression тестування. E2E-тестування може бути виконано як тестувальником, так і автоматичному режимі. У обох випадках процес тестування вимагає підготовку вхідних даних та їх видалення після виконання тесту. Одним із способів швидко задати вхідні данні, особливо у веб-орієнтованих системах – додати їх безпосередньо у базу даних. Для демонстрації підходу у рамках курсу «Технічна підтримка програмного забезпечення» для студентів 4-го курсу денної форми навчання спеціальності “122 Комп’ютерні науки СумДУ” було розроблено модельне програмне забезпечення для баз даних, як сервіс Elephant, яке надає можливість за допомогою Rest API створювати, заповнювати бази даних у СУБД PostgreSQL, створювати їх резервні копії та повертатися до них. Розроблений продукт дозволяє підвищити ефективність тестування завдяки автоматизації підготовки вхідних даних та виконання тестового сценарію та їх скидання по завершенню. Час, що витрачається на виконання цих дій є частиною часу тестування, тому подальше його скорочення дозволить знизити витрати на тестування. Виходячи із актуальності проблеми метою роботи є оптимізація підсистеми створення та відновлення резервних копій DBaaS Elephant за критерієм часу.

Для досягнення поставленої мети сформульовано наступні задачі роботи:

1) виконати аналітичний огляд методів створення резервних копій реляційних баз даних;

2) провести проектування підсистеми резервного копіювання DBaaS Elephant з урахуванням можливості застосування кількох підходів до створення та відновлення резервних копій;
3) розробити необхідне програмне забезпечення;
4) провести аналіз результатів та виконати порівняння ефективності запропонованих підходів.

РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМИ. ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕННЯ

1.1. Аналітичній огляд методів резервного копіювання реляційних баз даних

Мною було розглянуто і проаналізовано, які резервні копії існують взагалі. Їх можна поділити на наступні типи:

1) логічні;
2) фізичні;
3) відновлення на момент часу (PITR);
4) знімки файлової системи.

Логічні копії являють собою цілісні дані в форматі, які система управління базами даних (СУБД) може виконати як SQL-команди, це може бути як текстовий формат зрозумілий користувачу так і бінарний. Здебільшого використовується для відновлення даних з нуля. Серед переваг логічних копій можна відмітити те, що з них легко відновлювати дані, для створення такого бекапу не потрібно зупиняти роботу бази даних, вони більш гнучкі, бо отримані дані можна відновити на іншому сервері, іншій версії СУБД, а іноді навіть і на іншій СУБД. Також можна створити копію окремого об’єкта, наприклад бази чи таблиці. За допомогою великої кількості параметрів, їх можна легко налаштовувати. Але є і недоліки, для їх створення потрібні ресурси сервера, особливо для відновлення, порівняно невелика швидкість, особливо для великих БД, також іноді розмір файлу більший, через текстовий формат даних.

Фізична копія – це копіювання файлів, які СУБД використовує для зберігання даних. Але при простому копіюванні можуть ігноруватися і блокуватися транзакції, в результаті вони можуть бути неправильно збережені і порушені, тож при спробі приєднати такий файл він буде в неналежному стані і це призведе до помилок. Отже, якщо ми бажаємо

зробити фізичну копію власноруч, потрібно зупинити базу даних чи ввімкнути спеціальну функцію в СУБД, а тільки потім копіювати, чи скористатися спеціальними програмами, які можуть зробити копію не призупиняючи роботи СУБД. Ще одним очевидним недоліком є те, що неможливо відновити якісь конкретні дані, а тільки увесь кластер повністю, і обов'язково необхідно буде вимикати СУБД для будь-якого фізичного відновлення. Також, можуть бути складнощі з переносом копії на іншу машину/систему. Перевагами цього типу є: велика швидкість створення, так як просто копіюються файли (швидкість зводиться, до швидкості їхнього копіювання) і немає необхідності зчитувати вміст файлу.

Відновлення на момент часу (Point in time recovery) ще називають інкрементним резервним копіюванням або архівним резервним копіюванням. Фактично це фізична копія, але з деякими додатковими налаштуваннями, які ми далі розглянемо. PITR забезпечує неперервне резервне копіювання даних таблиць. СУБД записує всі транзакції користувачів (додавання, оновлення, видалення) до спеціальних файлів-журналів, кожен раз, коли до баз даних вносяться зміни. Первинно, цей журнал існував для безпечного відновлення після краху сервера. Якщо відбувався крах, то СУБД могла відновити свою цілісність за допомогою записів цього журналу. Однак, наявність такого журналу дозволила з'явитися новій стратегії резервного копіювання, за допомогою якої можна комбінувати фізичне резервне копіювання з відновленням на момент часу. Якщо потрібне відновлення, то спочатку воно виконується з резервної копії файлової системи, а потім відновлюється з резервної копії журналів. Такий підхід більш складніший, чим наведені типи вище, але він має деякі значні переваги:

1) якщо неперервно передавати усі файли-журнали на іншу машину, яка буде завантажена з того ж файлу резервної копії бази даних, то вийде резервна система: в будь який момент ми зможемо запустити другу машину і вона буде мати практично поточну копію бази даних;

2) немає необхідності відтворювати записи файлу-журналу від початку і до кінця. Можна зупинити відтворювання в будь-якій точці і отримати цілісний знімок бази даних на цей час. Таким чином ця технологія підтримує відновлення на момент часу: можна відновити базу даних до її стану на будь-який час з моменту виконання резервної копії;
3) оскільки для відтворення можна комбінувати файли-журнали за необмежено довгий час, неперервне резервне копіювання може виконуватись за допомогою неперервної архівації цих файлів-журналів. Це особливо важливо для великих баз даних, де може бути не зручно створювати повні резервні копії досить часто;
4) немає необхідності робити спочатку цілісну резервну копію файлової системи. Будь яка внутрішня нецілісність у резервній копії буде відкоригована при відтворюванню журналу (це не сильно відрізняється від того, що відбувається при відновленні після краху).

Серед недоліків, можна виділити неможливість відновлення окремого компоненту, а знов таки тільки кластер повністю, хоч і на будь-який момент часу, а також складності з налаштуванням.

Знімком файлової системи є стан системи у певний проміжок часу. Такий знімок можна зробити не на кожній файловій системі, система повинна бути для цього призначена. Особливістю таких систем є наявність механізму копіювання при записуванні (Copy-On-Write) – метод для ефективної реалізації операції дублювання чи копіювання. Ідея цього механізму в тому, що при читанні області даних використовується загальна копія, а у випадку зміни даних створюється нова копія. Найпопулярнішими серед таких систем є LVM, ZFS та BTRFS.

Розглянемо на прикладі LVM (Менеджер Логічних Томів). LVM є підсистемою операційної системи Linux, та дозволяє використовувати різні ділянки одного жорсткого диску чи ділянки з різних жорстких дисків як один

логічний том. Іншими словами LVM додає рівень абстракції між фізичними/логічними дисками та файловою системою.

Переваги: це дуже швидкий спосіб для створення резервної копії.

Недоліки: необхідна вільна пам’ять для створення віртуального диску і знімків, якщо неправильно розрахувати і вичерпати вільне місце, то дані на диску і збережені знімки будуть пошкоджені.

Під час огляду ми зрозуміли, що нам не підходять фізична копія та PITR, тому що через такі відновлення, неможливо відновити конкретну базу, а тільки увесь кластер повністю і це розбігається з функціями Elephant. Знімки файлової системи, також не зовсім підходять. Підхід з їхнім використанням можливий, але ми стикаємося з тим, що для кожного користувача, потрібно створювати новий диск і тільки тоді це буде працювати. Тому у випадку коли буде багато користувачів, у яких буде багато баз даних, то і відповідно кількість створених дисків буде дуже значною, а це в свою чергу дає велике навантаження на операційну систему. Отже, у своїй праці я буду розглядати логічні методи резервного копіювання.

1.2 Вбудовані в PostgreSQL засоби резервного копіювання і відновлення

pg_dump – це інструмент для створення резервних копій, яким можна зробити логічну копію однієї бази даних чи окремого елемента бази. Утиліта створює копію як у бінарному форматі, так і в форматі sql-запитів. Також в неї вбудовані функції для архівування, наприклад ми можемо отримати формат вихідного файлу gzip чи tar.

Переваги:

1. копії створюються паралельно роботі бази даних, база залишається доступною і для читання, і для запису
2. дуже висока сумісність між різними версіями СУБД PostgreSQL
3. гнучке налаштування за допомогою великої кількості ключів

4. підтримує паралелізм за рахунок збільшення навантаження на систему

Недоліки:

1. невисока швидкість роботи з великими базами даних

Потенційно на швидкодію можуть вплинути ключі, які ми розглянемо далі.

Ключ --format=значення зі значеннями custom або directory. Цей ключ відповідає за формат виводу. При значенні custom на виході буде отриманий файл у спеціальному архівному форматі. За замовчуванням відбувається стиснення. При значенні directory, отримаємо дані у форматі каталогу. В даному каталозі для кожного великого об’єкту та таблиці буде створено окремий файл. Може працювати у декілька потоків і також за замовчуванням стискається.

Ключ --jobs=кількість завдань. Ключ jobs виконує створення копії з вказаним числом потоків, підтримується тільки формат directory. Цей ключ може вплинути на швидкодію, але необхідно розуміти, що зросте навантаження на ресурси машини.

Ключ --compress=0..9. Встановлює рівень стиснення вихідних даних. Працює тільки зі спеціальним форматом та форматом каталогу. При значенні 0 стиснення буде вимкнуте.

Утиліта pg_dumpall відповідає за створення логічних резервних копій усіх баз, в тому числі і системних. Тобто ця утиліта працює за принципом pg_dump але з набагато меншою кількістю ключів і ми не можемо обрати конкретну базу, тому ця утиліта не підходить.

pg_restore - програма для відновлення баз даних, які були створені програмою pg_dump. Швидкість відновлення, можливо, буде залежати від вихідного файлу, наприклад, якщо ми стиснемо базу даних, відповідно знадобиться час для розархівування. Серед ключів, які можуть вплинути на

швидкодію можна відмітити ключ --jobs, який виконує аналогічну функцію як і в pg_dump, - відкриває паралельні з’єднання.

pg_basebackup - утиліта для створення фізичної резервної копії повністю усього кластеру. В результаті її роботи отримуємо набір файлів у бінарному форматі. Налаштування ключами обмежені і немає жодних ключів, щоб якось прискорити процес створення копії. СУБД PostgreSQL обмежує користувачам які не мають права “REPLICATION” створювати копії. Для безпечного використання, будь-яка фізична копія повинна бути узгодженою.

Переваги:

1) швидке створення резервної копії;
2) не складне відновлення;
3) можливість створення резервної копії через мережу;

Недоліки:

1) неможливо створити резервну копію однієї бази;
2) потрібно мати додаткові права і виконати певні налаштування;
3) для відновлення потрібне вимикання СУБД;
4) для збереження копії і журналів потрібно багато вільного місця;

Таблиця 1.1 Узагальнення розглянутих програм

Програма	Тип копії	Копія окремого елемента	Багатопо-точність	Потрібне вимикання СУБД при розгортанні?	Підтримка PITR
pg_dump	Логічна	Так	Так	Ні	Ні
pg_dumpall	логічна	Ні	Ні	Ні	Ні
pg_basebackup	фізична	Ні	Ні	Так	Так

1.3 Постановка задачі

Метою роботи є підвищення ефективності існуючого програмного забезпечення за критерієм часу. Огляд наявних рішень показав, що найбільш доцільним буде використання наявних у PostgreSQL вбудованих засобів створення резервних копій. Вони мають різну ефективність в залежності від налаштувань. Для досягнення поставленої мети необхідно:

- Внести зміни у існуючий продукт, так щоб мати змогу використовувати різні підходи до створення резервних копій без перекомпіляції додатку. Для цього розширимо алгоритм створення та відновлення резервних копій завантаженням параметрів із наявного конфігураційного файлу.
- Провести порівняння за критерієм часу впливу параметрів format, compress та jobs для наступних множин вхідних даних: малий обсяг даних (300МБ), середній обсяг (700МБ), великий (2500МБ).

РОЗДІЛ 2. ПРОЕКТУВАННЯ ТА ІМПЛЕМЕНТАЦІЯ ЗАСОБУ РЕЗЕРВНОГО КОПІЮВАННЯ

Основною функцією сервісу Elephant є полегшення роботи, яка виникає при тестуванні програмних продуктів, а саме: наповнення продукту тестовими даними та швидкий перехід між різними версіями цих даних. Перехід між різними версіями реалізовано за допомогою створення резервних копій та скидання до них. Дуже корисним і важливим є наявність API запитів, які дають можливість максимально автоматизувати роботу з сервісом. На Рисунку 2.1 наведено вигляд головної сторінки.

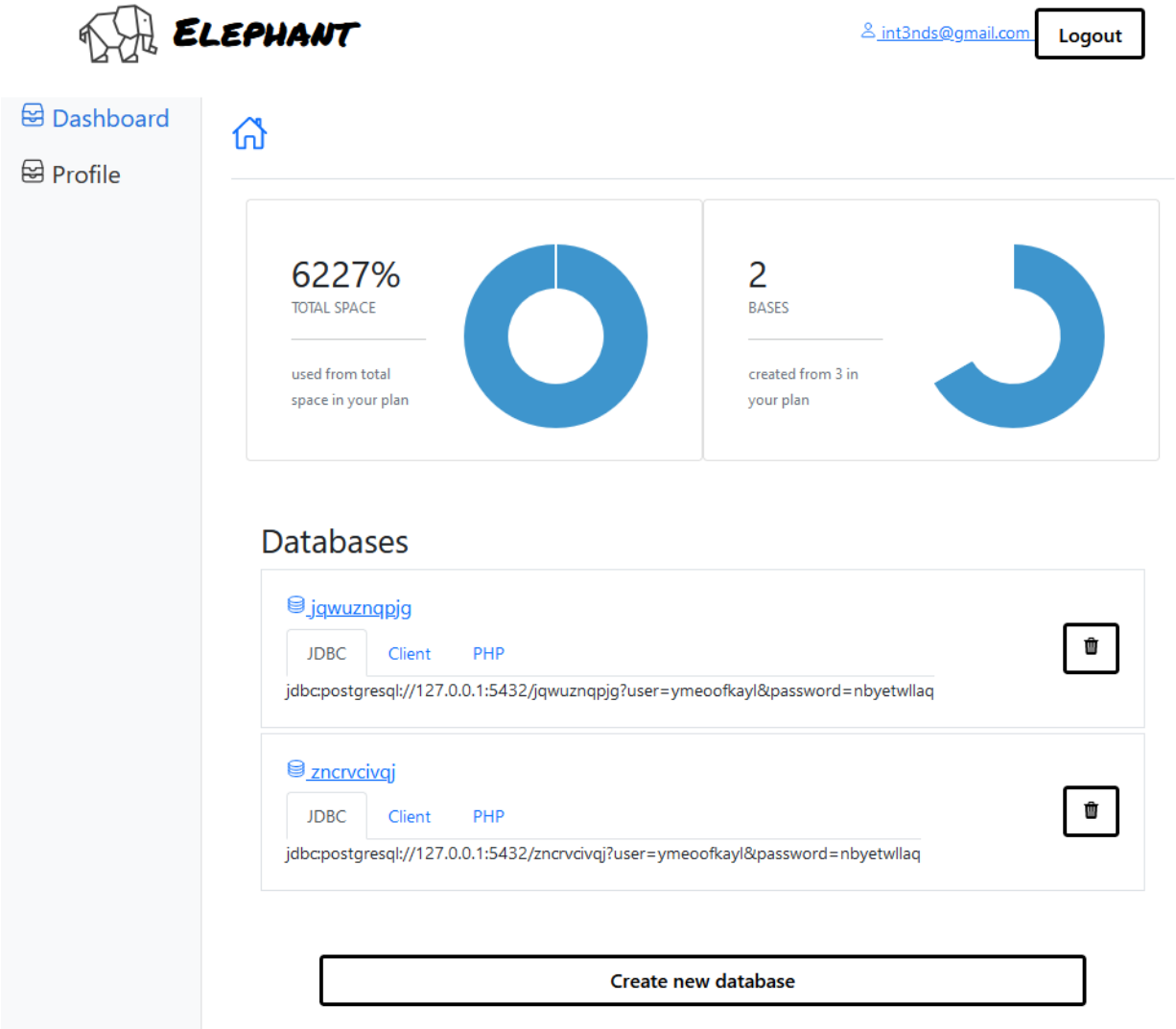

Рисунок 2.1 — Головна сторінка сервісу Elephant

Натиснувши, на одну зі створених баз, або створивши нову, переміщуємося на сторінку з можливими діями, по відношенню до обраної бази (Рисунок 2.2).

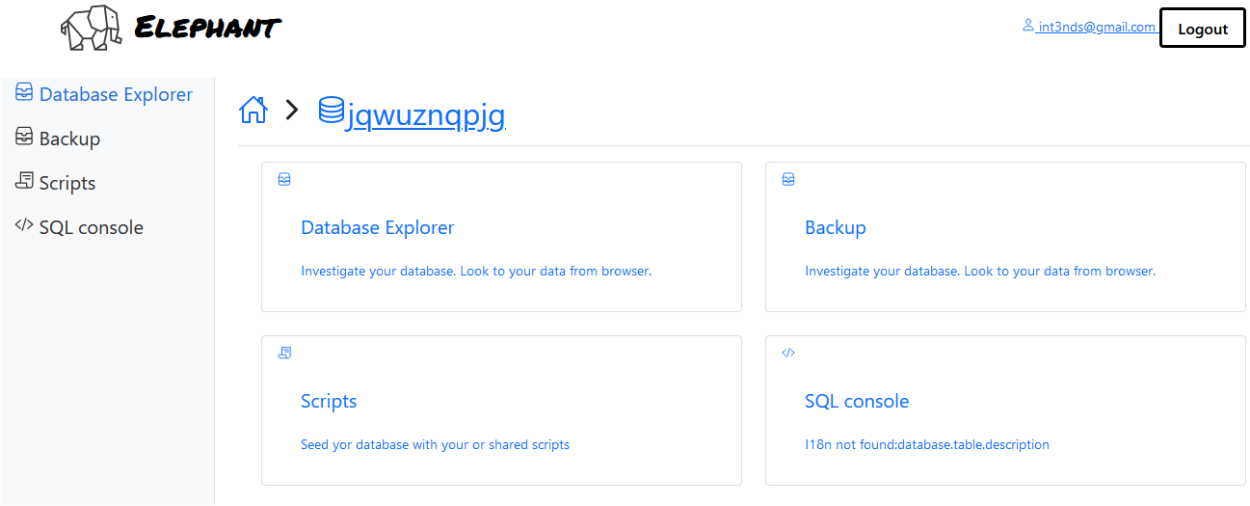

Рисунок 2.2 — Домашня сторінка бази даних

Для більш детального огляду процесу створення і відновлення резервної копії, була спроектована діаграма (Рисунок 2.3). На ній ми бачимо, що є клієнти які займаються тестуванням. Вони з'єднуються безпосередньо зі своєю базою даних (авторизація засобами БД). А є клієнти, які управляють даними (через API або UI). Вони ініціюють створення та відновлення резервних копій. Всі їхні дії авторизуються та відмічаються у БД сервісу, для того, щоб ми знали перелік точок відновлення та могли ними керувати. Створення резервної копії відбувається онлайн, а ось відновлення - потребує переривання з'єднань із цільовою базою даних.

У рамках роботи потрібно доопрацювати те, що відбувається на рівні операційної системи при запитах на створення та відновлення резервних копій. Системи програмного забезпечення серверу додатку знаходиться на тій самій фізичній машині, що і PostgreSQL, тому ми допрацьовуємо сам додаток. Якщо б це було не так, то потрібно було б мати окремий агент, якій виконує команди від сервера додатку на цільовій операційній системі.

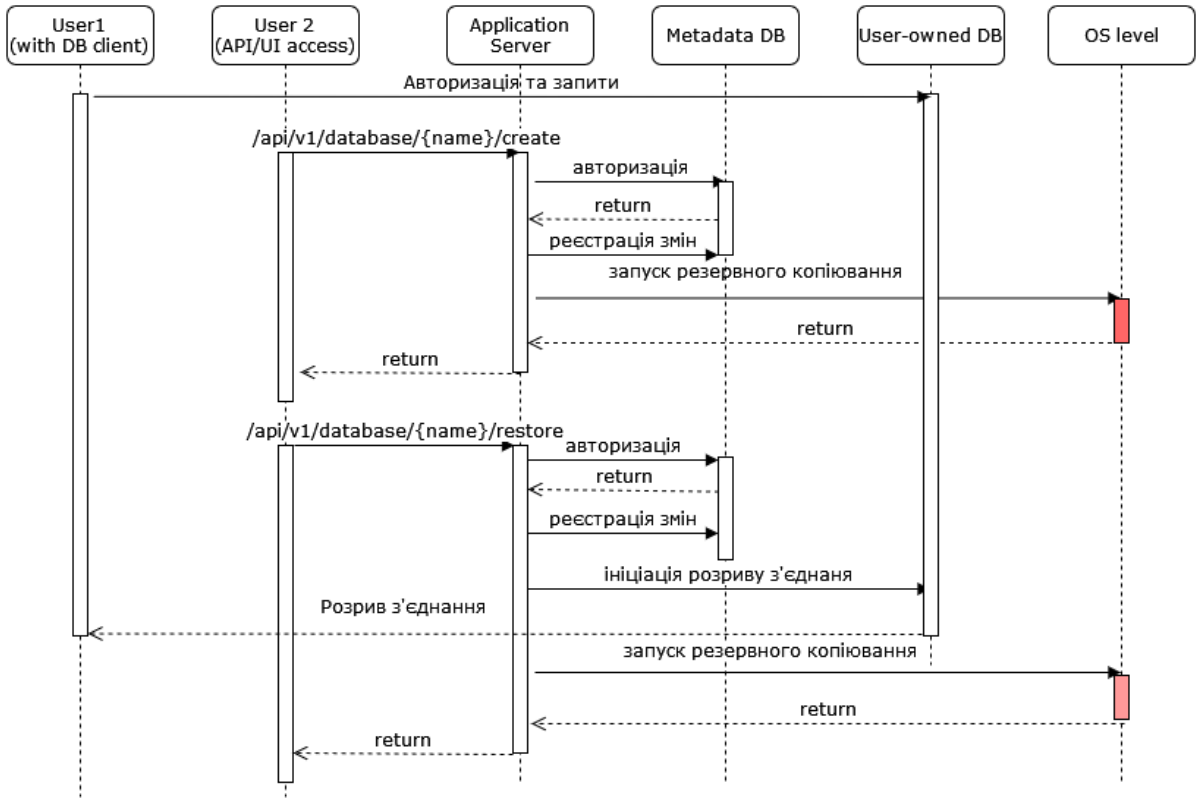

Рисунок 2.3 — UML Діаграма огляду взаємодії

Проект Elephant написаний на мові програмування Java. Після ознайомлення з кодом програми www.github.com/potapuff/elephant , було знайдено, за допомогою яких саме методів виконується резервне копіювання та відновлення.

Нижче наведено метод createBackup, який використовується для створення резервних копій.

```
private static void createBackup(String owner, String database, String
pointName) {
   String path = filePath(owner, database, pointName);
   try {
       FileUtils.forceMkdirParent(new File(path));
   } catch (Exception ex) {
       throw new HttpError500(ex);
   }
   CmdUtil.exec(String.format("pg_dump --format=custom --dbname=%s
   --file=%s", DBPool.dbUtilUrl(database), path));
}
```

З коду видно, що використовується програма pg_dump, яку ми розглядали вище, з ключем --format=custom. Отже, ми отримуємо вихідний файл у спеціальному архівному форматі.

Тепер потрібно розглянути метод для розгортання створених копій restoreBackup.

```
private static void restoreBackup(String owner, String database,
String     pointName) {
    String path = filePath(owner, database, pointName);
    var recreate = DatabaseService.exists(database) ? "--clean" : "";
    CmdUtil.exec(String.format("pg_restore %s --create --dbname=%s
%s",
    recreate, DBPool.dbUtilUrl(DBPool.DEFAULT_DATABASE), path));
}
```

Використовується програма pg_restore з ключем --create, функція якого створити базу даних перш ніж починати відновлення та опціональним ключем --clean, який видаляє об’єкти бази даних, перед тим як створювати їх. Також ці два ключі взаємодіють між собою, доповнюючи один одного.

Для того, щоб значно скоротити час тестування і після кожної зміни в методах createBackup та restoreBackup знову і знову не компілювати програму був розроблений окремий метод, який дозволяє у окремому конфігураційному файлі config.properties зберігати параметри до утиліт pg_dump і pg_restore. Заповнення цього файлу повинне бути у такому форматі

```
create_backup=pg_dump --format=d -Z 0 -j 8 --dbname=${dbname} --
file=${path}
restore_backup=pg_restore --format=d -j 8 --create --dbname=${dbname}
${path}
restore_backup_recreate=pg_restore --clean --create --dbname=${dbname}
${path}
```

Нижче наведено код створеного методу, а повний код BackupService.java (бо сам файл також зазнав незначних змін) наведено у Додатку 1.

```
    private static void initConfig() {
        try {
```

```
            Path root = Paths.get("").toAbsolutePath();
            Path configFile = root.resolve("config.properties");

            if (!Files.isRegularFile(configFile)) {
                throw  new  RuntimeException("File  'config.properties'
does not exists!");
            } else {
                // read
                var map = new HashMap<String, String>();
                for (String line : Files.readAllLines(configFile)) {
                    String[] lineArr = line.split("=", 2);
                    map.put(lineArr[0], lineArr[1]);
                }

                if (map.containsKey("create_backup"))
                    CREATE_BACKUP_COMMAND = map.get("create_backup");
                if (map.containsKey("restore_backup"))
                    RESTORE_BACKUP_COMMAND                              =
map.get("restore_backup");
                if (map.containsKey("restore_backup_recreate"))
                    RESTORE_BACKUP_RECREATE_COMMAND                     =
map.get("restore_backup_recreate");
            }
        } catch (Exception ex) {
            throw new RuntimeException(ex);
        }
    }
```

РОЗДІЛ 3 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Для тестування був використаний linux-сервер з переліченими (Таблиця 3.1) характеристиками.

Таблиця 3.1 Характеристики серверу

Кількість ядер процесора	4
Об’єм оперативної пам’яті	8 Гб
Об’єм і тип накопичувача	50 Гб, SSD
Операційна система	Ubuntu 20.04 (LTS) x64
Версія PostgreSQL	14.3
Версія Java	17.0.3
Версія Apache Maven	3.8.5

Для того, щоб заповнити бази даних для тесту, потрібно згенерувати дані. В PostgreSQL є чудові функції, якими ми скористаємось, це функції generate_series та md5. Функція generate_series(start, stop) приймає на вхід 2 значення і видає проміжок цілих чисел від start до stop, з кроком 1 (якщо не вказане третє значення функції яке відповідає за крок). Функція md5(string) приймає на вхід текстове значення, обраховує MD5-хеш і повертає результат у шістнадцятковій системі числення. Скористаємось можливостями Elephant щоб швидко створити і заповнити базу. Для цього натискаємо на кнопку “Create new database”.

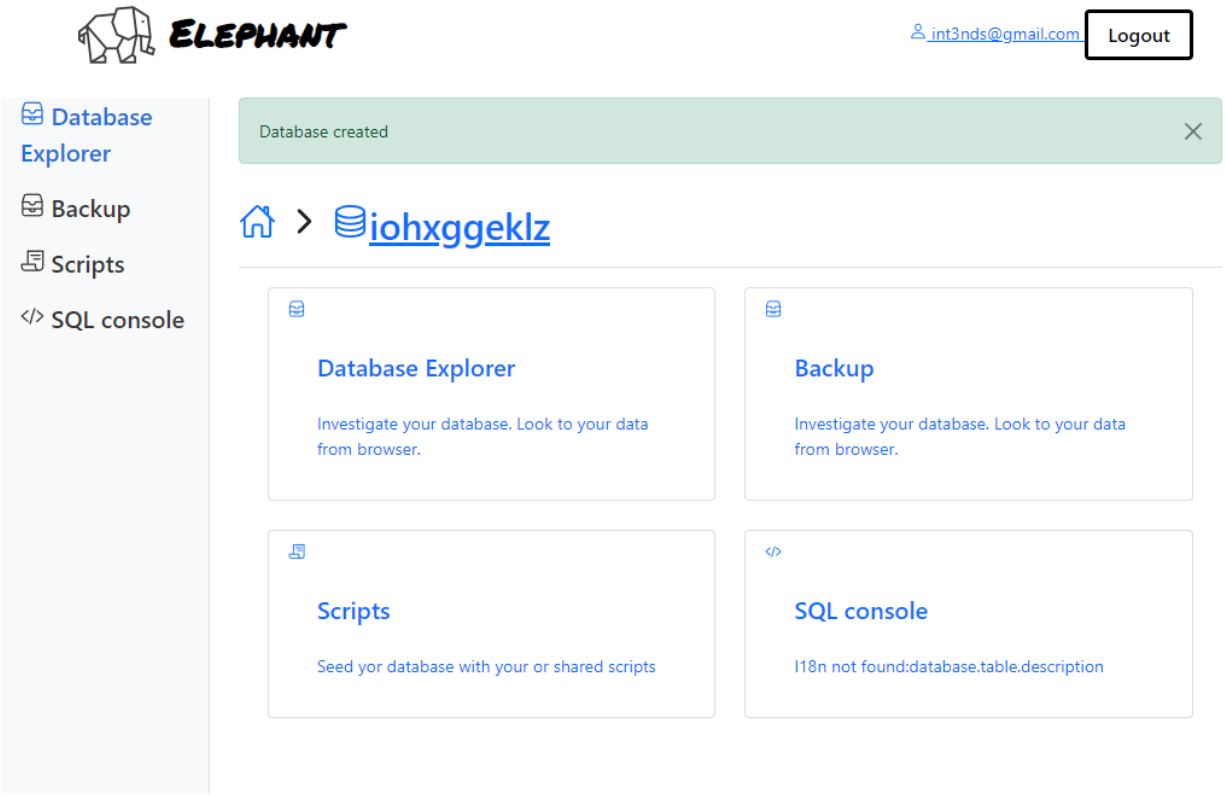

Рисунок 3.1 — Успішне створення бази даних

Отримали повідомлення, що база даних була створена. Тепер у розділі “SQL console” виконуємо код для створення і заповнення тестової таблиці. Заповнимо кількістю у 500 тисяч рядків.

```
CREATE TABLE table1 (
      id VARCHAR(50),
      name VARCHAR(50),
      surname VARCHAR(50),
      city VARCHAR(50));

INSERT INTO table1 (id,name,surname,city)
SELECT id,
      md5('name' || id),
      md5('surname' || id),
      md5('city' || id)
      from generate_series(1,500000) as gs(id);
```

Перейдемо в “Database Explorer”, для того, щоб подивитись результат. Отримали таблицю зі згенерованими значеннями Рисунок 3.2.

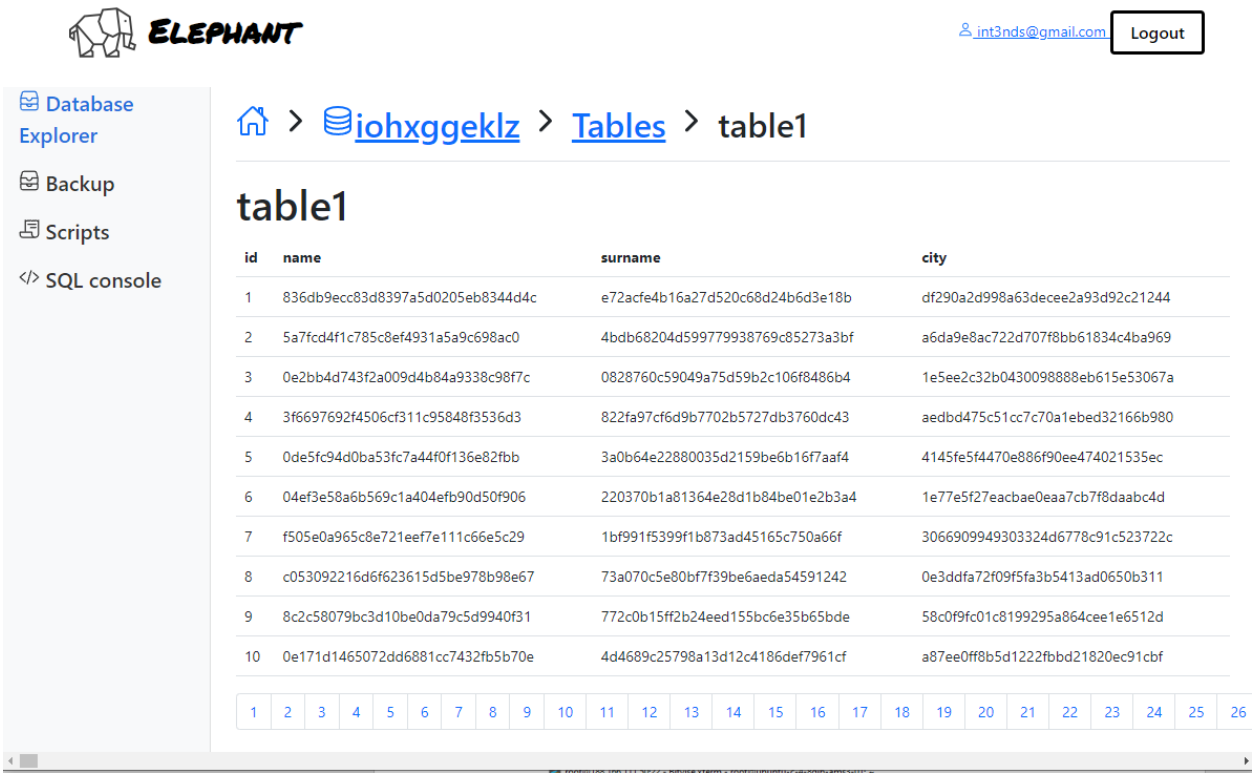

Рисунок 3.2 — Огляд згенерованої таблиці

Тепер необхідно виміряти розмір бази даних, для цього створимо копію за допомогою pg_dump без стиснення, використовуючи ключ compress=0 (Рисунок 3.3).

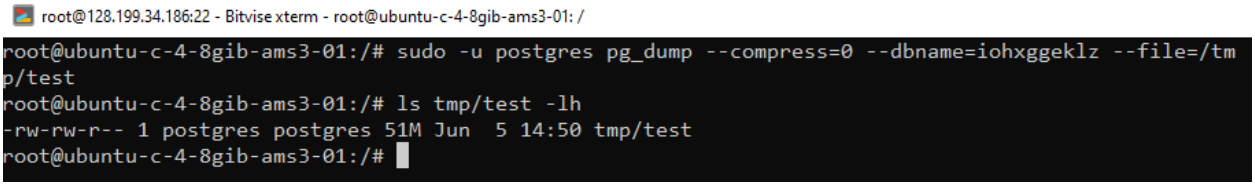

Рисунок 3.3 — Перевірка розміру тестової БД

Отримана копія має розмір 51 МБ. Отже роблю висновок, що 500 тисяч згенерованих рядків приблизно дорівнюють 51 МБ. Посилаючись на поставку до задачі, необхідно мати 3 бази даних розмірами: 300 МБ, 700 МБ і 2500 МБ, у згенерованих рядках це буде 3 мільйони, 7 мільйонів і 25 мільйонів рядків відповідно. Створюємо 3 бази даних та для кожної запускаємо окремий скрипт (наведені у Додатку 1) для створення таблиць і

заповнення згенерованими значеннями. Перевіряю розмір баз, створивши 3 копії і вимірявши їхній розмір (Рисунок 3.4).

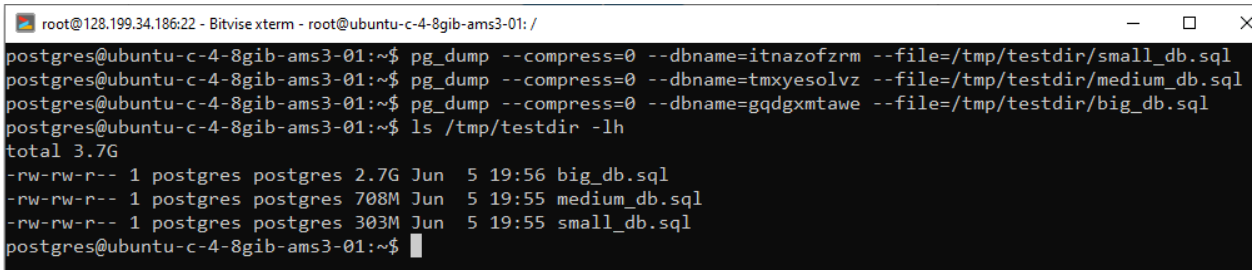

Рисунок 3.4 — Перевірка розміру створених для тестування БД

Розмір баз співпадає с потрібним для тесту. На Рисунку 3.5 наведено схему створених баз. Зв'язки між таблицями не було додано, так як для тестування потрібно перевірити можливість паралельного оновлення таблиць. І в даному випадку, це ніяк не впливає на результат, бо спочатку база заповнюється даними, а вже потім перевіряються зв'язки.

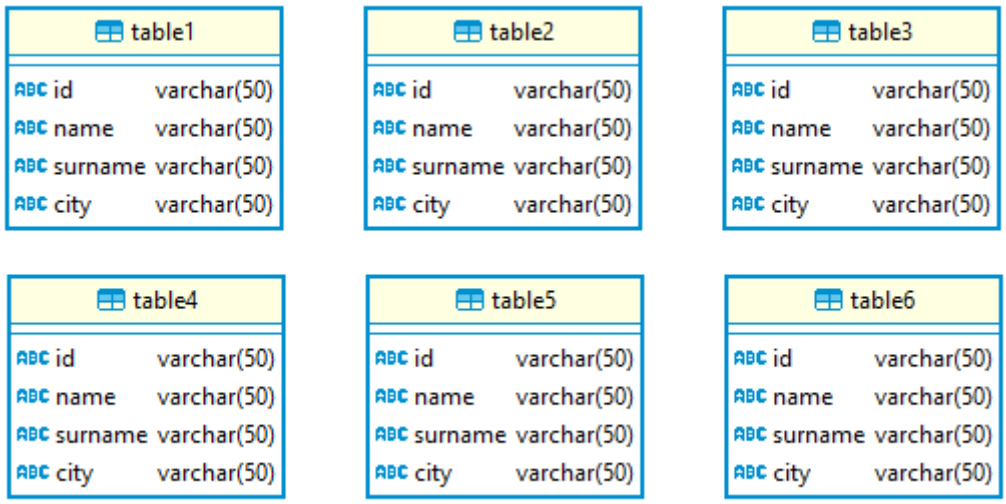

Рисунок 3.5 — Схема створеної БД

Далі необхідно перевірити час створення резервної копії у двох форматах виводу, custom та directory, з різними ключами. Формат directory цікавий перш за все, тим що підтримує багато потоків, в порівнянні з форматом custom. Усі заміри будемо повторювати 10 разів для кожного результату, далі

шукати середнє значення і заносити його в таблицю. Занесемо отримані результати в Таблицю 3.2.

3.2 Резервне копіювання pg_dump

Ключ	Мала(~300Мб)	Середня(~700Мб)	Велика(~2500Мб)
format=custom	16,4 с	39,8 с	2 хв 30 с
format=custom, compress=0	2,2 с	7,4 с	19,5 с
format=directory	16,5 с	38 с	2 хв 23 с
format=directory, compress=0	2,8 с	5,3 с	14,2 с
format=directory, jobs=8	7 с	19,5 с	54,6 с
format=directory, compress=0, jobs=8	1,7 с	3,4 с	10,2 с

Бачимо, що лідером по швидкості на всіх трьох базах, є ключ directory + compress + jobs.

Тепер знайдемо час відновлення, для кожної створеної резервної копії, спочатку без ключів, як і у програмі Elephant.

Таблиця 3.3 Відновлення pg_restore без ключів

Ключі, з яким створювалася копія	Мала(~300Мб)	Середня(~700Мб)	Велика(~2500Мб)
format=custom	5,6 с	11,5 с	48,5 с
format=custom, compress=0	4,9 с	12,7 с	52,1 с
format=directory	4,3 с	11,6 с	52,2 с
format=directory, compress=0	4,8 с	11,7 с	49,2 с

Ключі, з яким створювалася копія	Мала(~300Мб)	Середня(~700Мб)	Велика(~2500Мб)
format=directory, jobs=8	4,7 с	12,6 с	51,6 с
format=directory, compress=0, jobs=8	4,8 с	12,9 с	52,6 с

Виходячи з Таблиці 3.3 бачимо, що значення в межах баз з однаковим розміром практично однакові. У малій базі, різниця між найшвидшим і найдовшим відновленням становить 1,3 с, у середній 1,4 с, а у великій 4,1 с. Тепер перевіримо відновлення з ключем, який відповідає за кількість потоків (Таблиця 3.4).

Таблиця 3.4 Відновлення pg_restore з ключем --jobs=8

Ключі, з яким створювалася копія	Мала(~300Мб)	Середня(~700Мб)	Велика(~2500Мб)
format=custom	3,4 с	8,3 с	38,8 с
format=custom, compress=0	3 с	7,9 с	37,9 с
format=directory	3,6 с	8,1 с	39,5 с
format=directory, compress=0	2,8 с	7,6 с	40,6 с
format=directory, jobs=8	3,3 с	9 с	39,8 с
format=directory, compress=0, jobs=8	2,9 с	8,5 с	38,5 с

Порівнюючи результати pg_restore без ключів і з ключем jobs, бачимо, що з ключем jobs, швидкість всіх результатів помітно краща, отже для відновлювання його стовідсотково потрібно використовувати. Також робимо висновок, що формат виводу і навіть відсутність стискання, не впливає на швидкість відновлення. Тепер візьмемо кількість завдань, рівну кількості ядер процесора (Таблиця 3.5) і подивимось, чи є якась різниця, якщо перевищити цю кількість, як ми зробили у попередньому тесті, виставивши кількість рівну 8.

Таблиця 3.5 Відновлення pg_restore з ключем --jobs=4

Ключі, з яким створювалася копія	Мала(~300Мб)	Середня(~700Мб)	Велика(~2500Мб)
format=custom	3,3 с	8,1 с	37,9 с
format=custom, compress=0	3,2 с	7,7 с	37,2 с
format=directory	3,6 с	7,9 с	40,5 с
format=directory, compress=0	2,7 с	7,4 с	41,4 с
format=directory, jobs=8	3,1 с	9,2 с	39,9 с
format=directory, compress=0, jobs=8	3 с	8,3 с	39 с

При порівнянні результатів тесту відновлення з кількістю задач 4 та 8. Бачимо що результати майже не відрізняються, різниця до 1 секунди, отже встановлювати кількість задач, більшу за кількість ядер процесора не має сенсу. Для того, щоб зробити остаточний висновок, потрібно знайти сумарний час створення та відновлення копії. Додамо результати між собою і відобразимо їх у Таблиці 3.6.

Таблиця 3.6 Сумарний час резерву та відновлення

Ключі	Мала(~300Мб)	Середня(~700Мб)	Велика(~2500Мб)
format=custom	19,8 с	48,1 с	3 хв 8,8 с
format=custom, compress=0	5,2 с	15,3 с	57,4 с
format=directory	20,1 с	46,1 с	3 хв 2,5 с
format=directory, compress=0	5,6 с	12,9 с	54,8 с
format=directory, jobs=8	10,3 с	28,5 с	1 хв 34,4 с
format=directory, compress=0, jobs=8	4,6 с	11,9 с	48,7

Відобразимо отримані результати на стовпчастій діаграмі Рисунок 3.6.

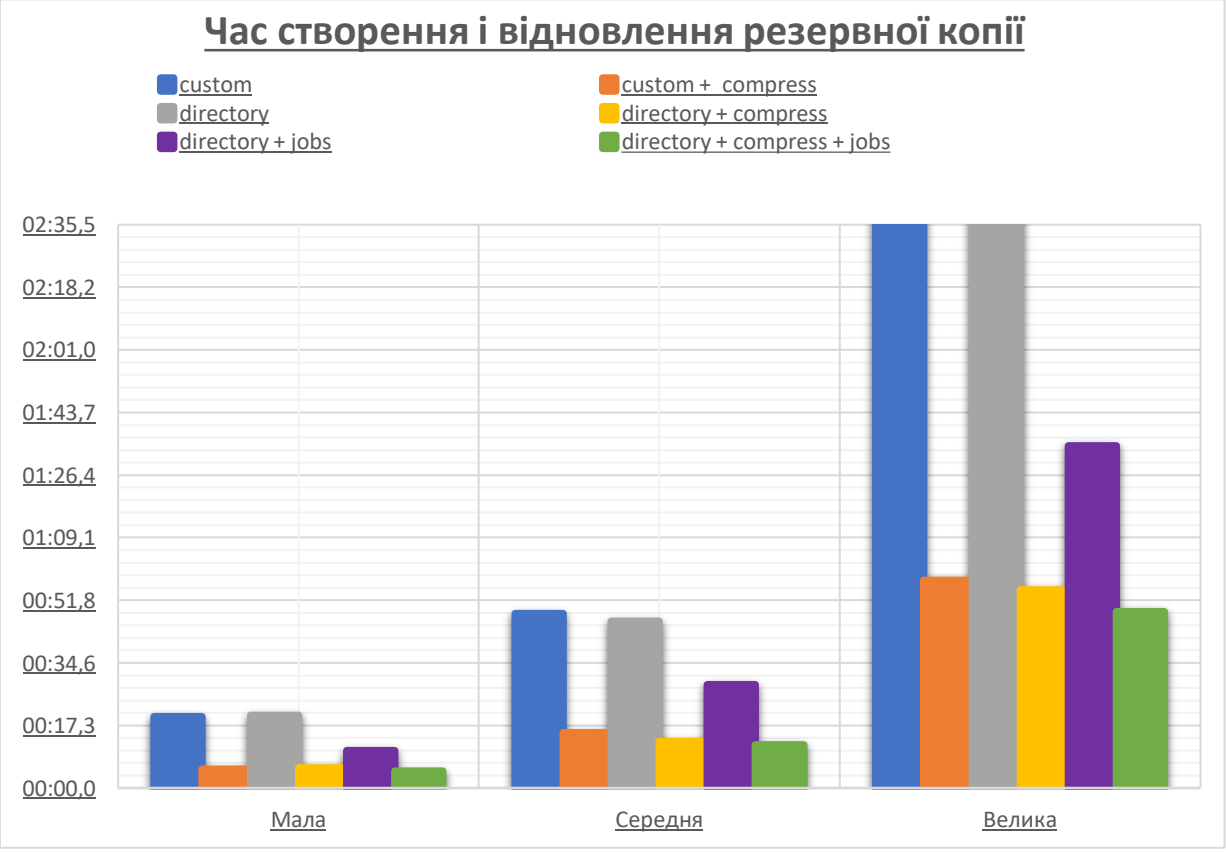

Рисунок 3.6 — Діаграма витраченого часу на створення і відновлення копії

З діаграми добре видно, що найкращі і більш-менш однакові результати показали ключі custom + compress, directory + compress та directory + compress + jobs. Але найкращим з них в усіх трьох таблицях є custom + compress + jobs, при чому ми впевнились, якщо кількість завдань (jobs) більша за кількість ядер процесора, це ніяк не пришвидшує процес. Тепер порахуємо час створення та відновлення копії, з початковими ключами Elephant. Мала база – 22 с, середня – 51,3 с, велика – 3 хв 18,5 с. Отже, мені вдалось покращити результат при базі рівній 300 МБ на 17, 4 с, базі рівній 700 МБ на 39 секунд, та базі рівній 2500 МБ на 2 хв 30 секунд.

ВИСНОВКИ

При порівнянні доступних методів створення резервних копій, було обрано логічний тип резервної копії з застосуванням вбудованих у PostgreSQL утиліт. Був проведений їхній огляд, були розглянуті переваги та недоліки кожної утиліти та сформовані параметри, які потенційно впливають на швидкодію.

При огляді програми DBaaS Elephant були знайдені та розглянуті методи створення та відновлення баз даних, які в ній використовуються, а також спроектована UML діаграма взаємодії з користувачем. Було змінено та дописано код програми. Була проведена підготовка для тестування, а також розроблений скрипт для тестового створення таблиць і заповнення їх згенерованими даними.

Проаналізувавши та порівнявши отримані результати, вдалося покращити швидкість створення та відновлення копії приблизно у 4 рази.

СПИСОК ЛІТЕРАТУТИ

1. Мікула М., Коцюк Ю., Мікула О. Організація баз даних та знань. Вид-во Нац. ун-ту «Острозь-ка акад.», 2021.
2. Рогов Е. В., PostgreSQL изнутри. ,М.: ДМК Пресс, 2022. — 660 с.
3. PostgreSQL: Documentation [Електронний ресурс] – Режим доступу: https://www.postgresql.org/docs/
4. Böszörmenyi Z., Schönig H.-J. PostgreSQL Replication. Packt Publishing, 2013. 250 с.
5. Group T. P. G. D. The PostgreSQL Reference Manual Volume 1: SQL Language Reference. Network Theory Ltd., 2007. 716 с.
6. Perkins J. PostgreSQL (Linux). Muska & Lipman/Premier-Trade, 2001. 450 с.
7. Postgresql Server Programming. Packt Publishing Limited, 2013.
8. Shaik B. Best Ways to Install PostgreSQL. PostgreSQL Configuration. Berkeley, CA, 2020.
9. Stinson B. PostgreSQL essential reference. Indianapolis: New Riders, 2002.371 с.
10. Thomas S. M. PostgreSQL High Availability Cookbook - Second Edition. Packt Publishing - ebooks Account, 2017. 536 с.
11. Silva R. Essential Postgres: Database Development using PostgreSQL
12. Douglas, Korry. Postgresql. Addison-Wesley, 2020.
13. Momjian, Bruce. PostgreSQL : Introduction and Concepts. Boston, Ma, Addison-Wesley, 2001.

ДОДАТОК 1

Скрипти створення таблиць та заповнення їх згенерованими даним

```
-- Скрипт створення та заповнення таблиці розміром 300 МБ

CREATE TABLE table1 (
      id VARCHAR(50),
      name VARCHAR(50),
      surname VARCHAR(50),
      city VARCHAR(50));

CREATE TABLE table2 (
      id VARCHAR(50),
      name VARCHAR(50),
      surname VARCHAR(50),
      city VARCHAR(50));

CREATE TABLE table3 (
      id VARCHAR(50),
      name VARCHAR(50),
      surname VARCHAR(50),
      city VARCHAR(50));

CREATE TABLE table4 (
      id VARCHAR(50),
      name VARCHAR(50),
      surname VARCHAR(50),
      city VARCHAR(50));

CREATE TABLE table5 (
      id VARCHAR(50),
      name VARCHAR(50),
      surname VARCHAR(50),
      city VARCHAR(50));

CREATE TABLE table6 (
      id VARCHAR(50),
      name VARCHAR(50),
      surname VARCHAR(50),
      city VARCHAR(50));

INSERT INTO table1 (id,name,surname,city)
SELECT id,
      md5('name' || id),
      md5('surname' || id),
      md5('city' || id)
      from generate_series(1,500000) as gs(id);
```

```
INSERT INTO table2 (id,name,surname,city)
SELECT id,
      md5('name' || id),
      md5('surname' || id),
      md5('city' || id)
      from generate_series(1,500000) as gs(id);

INSERT INTO table3 (id,name,surname,city)
SELECT id,
      md5('name' || id),
      md5('surname' || id),
      md5('city' || id)
      from generate_series(1,500000) as gs(id);

INSERT INTO table4 (id,name,surname,city)
SELECT id,
      md5('name' || id),
      md5('surname' || id),
      md5('city' || id)
      from generate_series(1,500000) as gs(id);

INSERT INTO table5 (id,name,surname,city)
SELECT id,
      md5('name' || id),
      md5('surname' || id),
      md5('city' || id)
      from generate_series(1,500000) as gs(id);

INSERT INTO table6 (id,name,surname,city)
SELECT id,
      md5('name' || id),
      md5('surname' || id),
      md5('city' || id)
      from generate_series(1,500000) as gs(id);

-- Скрипт створення та заповнення таблиці розміром 700 МБ

CREATE TABLE table1 (
      id VARCHAR(50),
      name VARCHAR(50),
      surname VARCHAR(50),
      city VARCHAR(50));

CREATE TABLE table2 (
      id VARCHAR(50),
      name VARCHAR(50),
      surname VARCHAR(50),
      city VARCHAR(50));

CREATE TABLE table3 (
```

```
      id VARCHAR(50),
      name VARCHAR(50),
      surname VARCHAR(50),
      city VARCHAR(50));

CREATE TABLE table4 (
      id VARCHAR(50),
      name VARCHAR(50),
      surname VARCHAR(50),
      city VARCHAR(50));

CREATE TABLE table5 (
      id VARCHAR(50),
      name VARCHAR(50),
      surname VARCHAR(50),
      city VARCHAR(50));

CREATE TABLE table6 (
      id VARCHAR(50),
      name VARCHAR(50),
      surname VARCHAR(50),
      city VARCHAR(50));

INSERT INTO table1 (id,name,surname,city)
SELECT id,
      md5('name' || id),
      md5('surname' || id),
      md5('city' || id)
      from generate_series(1,2000000) as gs(id);

INSERT INTO table2 (id,name,surname,city)
SELECT id,
      md5('name' || id),
      md5('surname' || id),
      md5('city' || id)
      from generate_series(1,1000000) as gs(id);

INSERT INTO table3 (id,name,surname,city)
SELECT id,
      md5('name' || id),
      md5('surname' || id),
      md5('city' || id)
      from generate_series(1,1000000) as gs(id);

INSERT INTO table4 (id,name,surname,city)
SELECT id,
      md5('name' || id),
      md5('surname' || id),
      md5('city' || id)
      from generate_series(1,1000000) as gs(id);
```

```
INSERT INTO table5 (id,name,surname,city)
SELECT id,
      md5('name' || id),
      md5('surname' || id),
      md5('city' || id)
      from generate_series(1,1000000) as gs(id);

INSERT INTO table6 (id,name,surname,city)
SELECT id,
      md5('name' || id),
      md5('surname' || id),
      md5('city' || id)
      from generate_series(1,1000000) as gs(id);

-- Скрипт створення та заповнення таблиці розміром 2500 МБ

CREATE TABLE table1 (
      id VARCHAR(50),
      name VARCHAR(50),
      surname VARCHAR(50),
      city VARCHAR(50));

CREATE TABLE table2 (
      id VARCHAR(50),
      name VARCHAR(50),
      surname VARCHAR(50),
      city VARCHAR(50));

CREATE TABLE table3 (
      id VARCHAR(50),
      name VARCHAR(50),
      surname VARCHAR(50),
      city VARCHAR(50));

CREATE TABLE table4 (
      id VARCHAR(50),
      name VARCHAR(50),
      surname VARCHAR(50),
      city VARCHAR(50));

CREATE TABLE table5 (
      id VARCHAR(50),
      name VARCHAR(50),
      surname VARCHAR(50),
      city VARCHAR(50));

CREATE TABLE table6 (
      id VARCHAR(50),
      name VARCHAR(50),
```

```
      surname VARCHAR(50),
      city VARCHAR(50));

INSERT INTO table1 (id,name,surname,city)
SELECT id,
      md5('name' || id),
      md5('surname' || id),
      md5('city' || id)
      from generate_series(1,7000000) as gs(id);

INSERT INTO table2 (id,name,surname,city)
SELECT id,
      md5('name' || id),
      md5('surname' || id),
      md5('city' || id)
      from generate_series(1,4000000) as gs(id);

INSERT INTO table3 (id,name,surname,city)
SELECT id,
      md5('name' || id),
      md5('surname' || id),
      md5('city' || id)
      from generate_series(1,4000000) as gs(id);

INSERT INTO table4 (id,name,surname,city)
SELECT id,
      md5('name' || id),
      md5('surname' || id),
      md5('city' || id)
      from generate_series(1,4000000) as gs(id);

INSERT INTO table5 (id,name,surname,city)
SELECT id,
      md5('name' || id),
      md5('surname' || id),
      md5('city' || id)
      from generate_series(1,4000000) as gs(id);

INSERT INTO table6 (id,name,surname,city)
SELECT id,
      md5('name' || id),
      md5('surname' || id),
      md5('city' || id)
      from generate_series(1,4000000) as gs(id);
```

Програмний код зміненого файлу BackupService.java

```
package edu.sumdu.tss.elephant.model;

import java.io.File;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
import java.util.Date;
import java.util.HashMap;
import java.util.List;
import java.util.StringJoiner;

import org.apache.commons.io.FileUtils;
import org.sql2o.Connection;
import org.sql2o.Sql2oException;

import edu.sumdu.tss.elephant.helper.DBPool;
import edu.sumdu.tss.elephant.helper.exception.BackupException;
import edu.sumdu.tss.elephant.helper.exception.HttpError500;
import edu.sumdu.tss.elephant.helper.exception.NotFoundException;
import edu.sumdu.tss.elephant.helper.utils.CmdUtil;
import edu.sumdu.tss.elephant.helper.utils.ParameterizedStringFactory;

public class BackupService {

    private static final String LIST_BACKUP_SQL = "SELECT * from
backups where database = :database";
    private static final String GET_BY_NAME_SQL = "SELECT * from
backups where database = :database and point = :point";
    private static final String INSERT_SQL =
            """
                    insert into backups(database, point, status,
"createdAt", "updatedAt")
                    values (:database, :point, :status, :createdAt,
:updatedAt)  ON CONFLICT(database, point) DO UPDATE\s
                    SET status = excluded.status,\s
                       "updatedAt" = now();""";
    private static final String DELETE_BACKUP = "DELETE FROM backups
WHERE database = :database and point = :point;";
    private static final ParameterizedStringFactory DROP_DB = new
ParameterizedStringFactory("DROP DATABASE :database WITH (FORCE);");

    private static String CREATE_BACKUP_COMMAND;
    private static String RESTORE_BACKUP_COMMAND;
    private static String RESTORE_BACKUP_RECREATE_COMMAND;

    static {
        initConfig();
```

```
    }

    public static List<Backup> list(String dbName) {
        try (Connection con = DBPool.getConnection().open()) {
            return
con.createQuery(LIST_BACKUP_SQL).addParameter("database",
dbName).executeAndFetch(Backup.class);
        }
    }

    public static Backup byName(String database, String point) {
        try (Connection con = DBPool.getConnection().open()) {
            Backup                          backup                          =
con.createQuery(GET_BY_NAME_SQL).addParameter("database",
database).addParameter("point",
point).executeAndFetchFirst(Backup.class);
            if (backup == null) {
                throw new NotFoundException("Point not found");
            }
            return backup;
        }
    }

    public static void perform(String owner, String database, String
pointName) throws BackupException {
        try (Connection con = DBPool.getConnection().open()) {
            Backup point;
            try {
                point = byName(database, pointName);
            } catch (NotFoundException ex) {
                point = new Backup();
                point.setDatabase(database);
                point.setPoint(pointName);
                point.setCreatedAt(new Date());
            }
            point.setUpdatedAt(new Date());
            point.setStatus(Backup.BackupState.PERFORMED.name());
            BackupService.save(point);
            BackupService.createBackup(owner, database, pointName);
            point.setUpdatedAt(new Date());
            point.setStatus(Backup.BackupState.DONE.name());
            BackupService.save(point);
        } catch (Sql2oException ex) {
            throw new BackupException(ex);
        }
    }

    private static void save(Backup point) {
        try (Connection con = DBPool.getConnection().open()) {
```

```
            con.createQuery(INSERT_SQL,
false).bind(point).executeUpdate();
        }
    }

    static public void restore(String owner, String dbName, String
pointName) throws BackupException {
        try (Connection con = DBPool.getConnection().open()) {
            Backup point = con.createQuery(GET_BY_NAME_SQL)
                    .addParameter("database", dbName)
                    .addParameter("point", pointName)
                    .executeAndFetchFirst(Backup.class);
            if (point == null) {
                point = new Backup();
                point.setDatabase(dbName);
                point.setPoint(pointName);
                point.setCreatedAt(new Date());
            }
            point.setUpdatedAt(new Date());
            point.setStatus(Backup.BackupState.PERFORMED.name());
            BackupService.save(point);
            BackupService.restoreBackup(owner, dbName, pointName);
            point.setUpdatedAt(new Date());
            point.setStatus(Backup.BackupState.DONE.name());
            BackupService.save(point);
        } catch (Sql2oException ex) {
            throw new BackupException(ex);
        }
    }

    public static void delete(String owner, String database, String
point) {
        String path = filePath(owner, database, point);
        DBPool.getConnection().open().createQuery(DELETE_BACKUP,
false)
                .addParameter("database", database)
                .addParameter("point", point)
                .executeUpdate();
        if (!new File(path).delete()) {
            throw new RuntimeException("File not deleted");
        }
    }

    private static void createBackup(String owner, String database,
String pointName) {
        String path = filePath(owner, database, pointName);
        try {
            FileUtils.forceMkdirParent(new File(path));
        } catch (Exception ex) {
            throw new HttpError500(ex);
```

```
        }
        var command =
                CREATE_BACKUP_COMMAND
                        .replace("${dbname}",
DBPool.dbUtilUrl(database))
                        .replace("${path}", path);
        CmdUtil.exec(command);
    }

    private static void restoreBackup(String owner, String database,
String pointName) {
        String path = filePath(owner, database, pointName);
        boolean recreate = DatabaseService.exists(database);
        var command =
                (recreate    ?    RESTORE_BACKUP_RECREATE_COMMAND    :
RESTORE_BACKUP_COMMAND)
                        .replace("${dbname}",
DBPool.dbUtilUrl(DBPool.DEFAULT_DATABASE))
                        .replace("${path}", path);
        CmdUtil.exec(command);
    }

    public static String filePath(String owner, String database,
String pointName) {
        return UserService.userStoragePath(owner) +
                File.separator + "backups" +
                File.separator + database +
                File.separator + pointName.replaceAll("\s","\\ ");
    }

    private static void initConfig() {
        try {
            Path root = Paths.get("").toAbsolutePath();
            Path configFile = root.resolve("config.properties");

            if (!Files.isRegularFile(configFile)) {
                throw new RuntimeException("File 'config.properties'
does not exists!");
            } else {
                // read
                var map = new HashMap<String, String>();
                for (String line : Files.readAllLines(configFile)) {
                    String[] lineArr = line.split("=", 2);
                    map.put(lineArr[0], lineArr[1]);
                }

                if (map.containsKey("create_backup"))
                    CREATE_BACKUP_COMMAND = map.get("create_backup");
                if (map.containsKey("restore_backup"))
```

```
                    RESTORE_BACKUP_COMMAND                                  =
map.get("restore_backup");
                if (map.containsKey("restore_backup_recreate"))
                    RESTORE_BACKUP_RECREATE_COMMAND                         =
map.get("restore_backup_recreate");
            }
        } catch (Exception ex) {
            throw new RuntimeException(ex);
        }
    }
}
```