КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет комп'ютерних наук та кібернетики
Кафедра математичної інформатики

Кваліфікаційна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Інформатика»
за спеціальністю 122 Комп'ютерні науки

на тему:

ТИПИ NoSQL БАЗ ДАНИХ І ЇХ ПОРІВНЯННЯ З РЕЛЯЦІЙНИМИ БД

Виконав студент 4 курсу
Ілля ЯКОВЕНКО

(підпис)

Науковий керівник:
асистент, кандидат технічних наук
Олексій ФЕДОРУС

(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент

(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри математичної інформатики
"____" _____
2023 р., протокол №_____
Завідувач кафедри
Василь ТЕРЕЩЕНКО

(підпис)

Київ – 2023

РЕФЕРАТ

Обсяг роботи 47 сторінок, 17 ілюстрацій, 3 таблиці, 12 джерел посилань, 3 додатки. БАЗИ ДАНИХ, ВИМІРЮВАННЯ, КЛАСИ, НАВАНТАЖЕННЯ, СИСТЕМИ УПРАВЛІННЯ БАЗАМИ ДАНИХ, ТЕСТУВАННЯ, ШВИДКОДІЯ.

Об’єктами роботи є класи систем управління базами даних, моделі транзакцій цих класів. Предметом роботи є виконання тестування навантаженням для отримання показників швидкодії СУБД.

Метою роботи є встановлення класів СУБД і визначення різниць між ними, отримання даних про швидкодію СУБД під навантаженням.

Методи дослідження: класифікація та тестування. Мова програмування - Java, інструмент – Yahoo! Cloud Benchmark Service, середа розробки – IntelliJ IDEA.

Результати роботи: виконано детальний огляд видів СУБД і проведено їх класифікацію, розглянуто моделі транзакцій та поняття вимірювання, підхід до нього; отримано ключові показники швидкодії СУБД під час навантаження, виконано їх порівняльний аналіз. За допомогою отриманих результатів проведено актуалізацію стану досліджень, оновлено дані що до порівняльних характеристик.

Результати роботи будуть корисні архітекторам інформаційних систем під час прийняття рішень що до побудови серверної частини та бази даних.

Задачі по класифікації баз даних та оцінці ефективності їх роботи виникають в усіх сферах інформаційних технологій де використовуються бази даних. Кожен раз при проектуванні системи спеціалісти повинні приймати рішення опираючись на вимоги до системи та функції доступних баз даних.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

ACID – **A**tomicity, **C**onsistency, **I**solation, **D**urability; атомарність, узгодженість, ізоляція, стійкість;

BASE – **Ba**sically available, **S**oft state, **E**ventually consistent; легкодоступність, динамічність, кінцева узгодженість;

NoSQL – Not Only Structured Query Language, не тільки структурована мова запитів;

SQL – Structured Query Language, структурована мова запитів;

YCSB – Yahoo! Cloud Benchmark Service;

БД – база даних;

ПЗ – програмне забезпечення;

РСУБД – реляційна система управління базами даних;

СУБД – система управління базами даних;

ВСТУП

Оцінка сучасного стану об’єкта дослідження. Сучасні інформаційні технології неперервно розвиваються, зростає потреба у зберіганні та обробці значних обсягів комплексних даних, тому змінюються вимоги до їх структури, а разом з цим і підходи до проектування та функціональності систем управління базами даних, покликанням цього є подолання недоліків наявних програмних рішень. Отже, з формуванням та ростом потреби у базах даних, що можуть бути легко масштабованими, або географічно розподіленими і швидшими у виконанні конкретних задач, на ринку з’являються програмні продукти, що спеціалізуються на розв’язанні конкретних проблем. Відповідно нагальним стає питання що до класифікації та порівняння СУБД між собою, для чіткого розрізнення випадків, де один програмний продукт має перевагу над іншим [1].

Актуальність роботи та підстави для її виконання. З огляду на те, що є потреба у висвітленні цього питання та підтриманні актуальності наявних даних, для того щоб архітектори програмних систем знали у яких ситуаціях слід краще використовувати конкретний тип СУБД, випливають вагомі підстави до виконання, як конкретно цієї роботи, так й інших подібних робіт.

Мета і завдання роботи. Метою цієї роботи є встановлення класів існуючих СУБД, визначення різниць між цими класами, вимог до них та підходів до проектування і створення програмних продуктів, порівняння їх швидкодії. Для досягнення цієї мети було поставлено завдання – провести класифікацію СУБД, розглянути різницю між класами, експериментально порівняти швидкодію представників програмних продуктів, як у середині класів, так і між ними; розробити рекомендації що до вибору СУБД [2].

Об’єкти, методи та засоби дослідження. Об’єктами вивчення у цій роботі є класи систем управління базами даних – NoSQL та реляційні, підходи до моделей побудови транзакцій у цих класах – BASE та ACID, різні представники СУБД з цих класів. До методів дослідження, що були застосовані у цій роботі, відносяться:

класифікація та аналіз для теоретичної частини роботи, експеримент і аналіз – для практичної. До засобів можна віднести Yahoo! Cloud Serving Benchmark (далі YCSB) – програмну специфікацію з відкритим вихідним кодом, що застосовується для виконання порівняння характеристик швидкодії СУБД[6].

Можливі сфери застосування. До можливих сфер відносяться усі сфери інформаційних технологій, де використовуються бази даних, а також результати цієї роботи можуть бути використані для отримання архітекторами ПЗ інформації, що допоможе розв’язати проблему вибору доцільної СУБД для проекту тощо.

РОЗДІЛ 1. ОГЛЯД NOSQL СУБД

NoSQL (від “**N**ot **O**nly **SQL**”) – це клас, що об’єднує велику кількість різноманітних СУБД, які мають відмінні, від класичних реляційних систем, механізми зберігання, обробки та доступу до даних та їх структуру. Ці СУБД покликані вирішити задачі, які типові реляційні СУБД не в змозі виконати, або не можуть виконати достатньо ефективно, тож було створено та розвинуто наступні основні спеціалізації NoSQL[3]:

- Документо-орієнтовні — призначені для роботи з документами, їх зберіганням, пошуком вкладеної інформації та керуванням нею. Такі бази даних, за будовою можна вважати БД типу ключ-значення (про цей тип далі буде), але документо-орієнтовні бази даних додатково аналізують внутрішню структуру документа (рисунок 1), щоб отримати метадані, за допомогою яких надалі буде виконано оптимізацію запитів; на відміну від звичайних БД типу ключ-значення, де дані, що містяться у значенні, додатково не оброблюються.

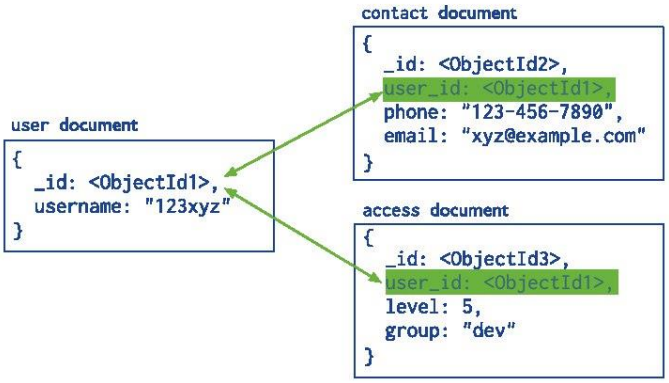

Рисунок 1 – Аналіз внутрішньої структури документа

- Репозиторії “ключ-значення” — призначені для зберігання таких структурних даних, як словники або хеш-таблиці. Словники складаються з наборів записів, які, у свою чергу, складаються з різних полів, що можуть мати відмінні один від одного типи даних (рисунок 2). Безпосередні записи зберігаються та відшукуються за допомогою ключа, який є унікальним ідентифікатором. За описом зрозуміло, що такі БД не мають визначеної наперед структури, на відміну від реляційних БД.

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

Рисунок 2 – Різні типи полів у записах

- Колонко-орієнтовні — зберігають таблиці даних не за рядками, як класичні реляційні БД, а за стовпцями (рисунок 3). Цей підхід дає перевагу у доступі до даних, коли запитуються лише конкретні підмножини стовпців; і усуває потребу в обробці непотрібних стовпців, що не є цільовими, дає більші можливості що до стиснення даних.

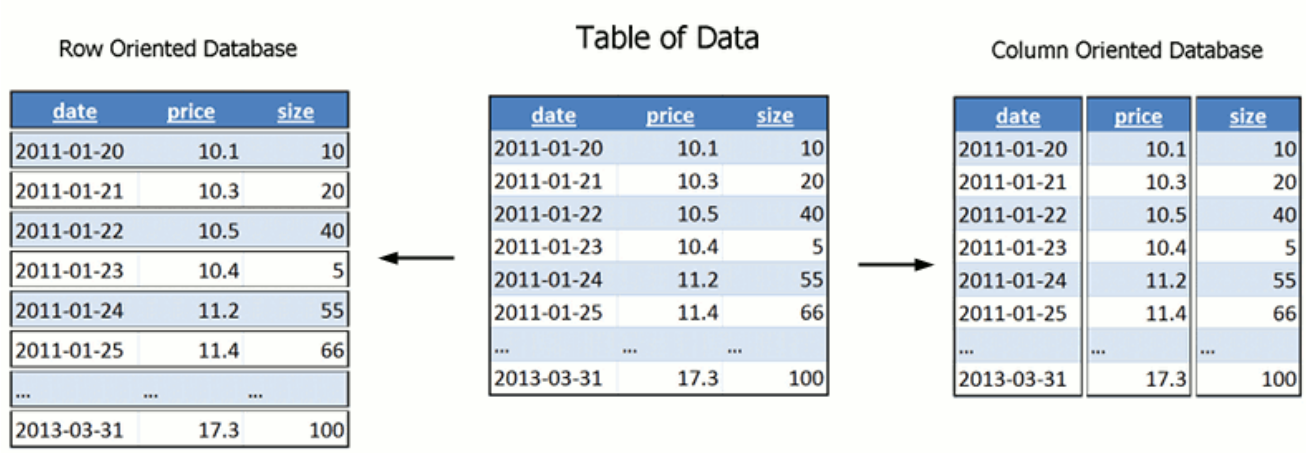

Рисунок 3 – Різниця у представленні даних між рядково-орієнтовними та колонко-орієнтовними базами даних

- Графові бази даних — тип, що використовує таку структуру даних, як граф; це дає змогу виконувати семантичні запити над вузлами та ребрами, можливості для представлення та зберігання комплексних даних. Граф відображає зв'язок елементів даних за допомогою колекції вузлів та ребер; де ребра, у свою чергу, характеризують типи відношень між вузлами (рисунок 4). Зв’язки дозволяють безпосередньо з'єднувати дані в сховищі, таким чином даючи можливість отримувати їх однією операцією. Відповідно, якщо ваші дані щільно зв'язані нетривіальним чином, то слід обрати цей тип БД.

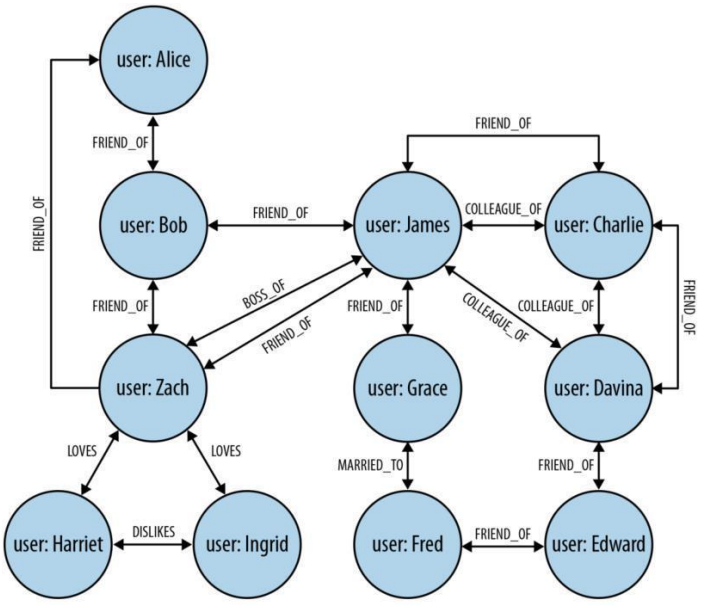

Рисунок 4 – Візуальне представлення різновидів зв’язків у графових базах даних

Також слід зауважити, що NoSQL відрізняються від реляційних БД не тільки описаним вище. Вони також можуть мати такі важливі відмінності[4]:

- Відмінні від SQL мови виклику операцій, відсутність затверджених стандартів що до них.
- Місце присутності даних – оперативна пам’ять чи диск.
- Можливості що до зберігання та розподілення даних по декільком серверам, у тому числі фізичним.
- Відсутність уніфікованої схеми даних.
- Слабша модель паралелізму.

РОЗДІЛ 2. ОГЛЯД РЕЛЯЦІЙНИХ СУБД

Реляційна СУБД — програмний додаток, який дозволяє створювати, керувати та маніпулювати реляційними базами даних. Він призначений для зберігання і організації великої кількості структурованих даних, забезпечуючи ефективний і надійний доступ до інформації. Такі СУБД широко використовуються в різних галузях і додатках, починаючи від невеликих особистих проектів і закінчуючи системами корпоративного рівня.

В основі РСУБД лежить концепція реляційної бази даних, яка складається з однієї або кількох таблиць. Кожна таблиця складається з рядків (також відомих як записи або кортежі) і стовпців (так званих полів чи атрибутів). Рядки представляють окремі екземпляри даних, тоді як стовпці визначають типи даних, з яких складаються рядки таблиці. Зв’язки між таблицями встановлюються за допомогою ключів - первинних та зовнішніх.

Однією з ключових переваг РСУБД є її здатність забезпечити цілісність і узгодженість даних. Для цього надаються механізми для визначення правил і обмежень щодо даних, гарантуючи, що зберігається лише дійсна та значуща інформація. Наприклад, можна встановити обмеження первинного ключа, щоб гарантувати, що кожен рядок у таблиці має унікальний ідентифікатор, запобігаючи повторюваним або конфліктним даним. Крім того, у реляційних СУБД важливим є механізм транзакцій, які є блоками виконання роботи, що забезпечують цілісність модифікацій даних їх атомарність, узгодженість, ізольованість та стійкість (ACID характеристики).

Додатковою важливою особливістю реляційних СУБД є мова структурованих запитів. SQL — це стандартизована мова, що використовується для взаємодії з реляційними базами даних. Вона надає широкий спектр потужних і гнучких операцій для отримання, зміни та керування даними. За допомогою SQL користувачі можуть писати запити для вилучення певної інформації з бази даних, виконувати складні об’єднання між таблицями, агрегувати дані та визначати представлення.

Реляційні СУБД пропонують численні переваги, включаючи масштабованість, незалежність і безпеку даних. Вони можуть обробляти велику кількість інформації і підтримувати одночасний доступ для кількох користувачів. Незалежність даних означає, що логічна структура бази даних відокремлена від фізичного сховища, що дозволяє полегшити обслуговування та зміни, не впливаючи на програми, які використовують базу даних. Крім того, РСУБД забезпечують вбудовані механізми безпеки, включаючи автентифікацію користувачів, контроль доступу та шифрування даних, для захисту конфіденційної інформації.

Загалом реляційні СУБД є надійним та ефективним рішенням для керування структурованими даними. Вони пропонують чітко визначену модель даних, забезпечення цілісності даних, потужні можливості запитів і масштабованість, що робить їх широко застосовуваними у різноманітних сферах.

2.1 Первинні та зовнішні ключі

У сфері реляційних баз даних первинні та зовнішні ключі відіграють фундаментальну роль у забезпеченні цілісності даних і встановленні зв’язків між таблицями. Розглянемо кожне з цих понять докладніше:

Первинний ключ — це унікальний ідентифікатор, який однозначно визначає кожен запис у таблиці. Слугує засобом, що відрізняє один рядок таблиці від іншого. При визначенні первинного ключа, гарантується, що кожен запис матиме унікальний ідентифікатор, який є важливим для цілісності даних і їх швидкого

пошуку. Первинний ключ може бути одним стовпцем або комбінацією кількох стовпців, відомою як складений ключ. Як правило, первинний ключ визначається під час створення таблиці та його подальше функціонування забезпечується алгоритмами СУБД.

Наприклад, розглянемо таблицю під назвою "Customers" (рисунок 5) з такими стовпцями, як: **CustomerID**, *CustFirstName, CustLastName, CustStreetAddress, CustCity, CustState, CustZipCode, CustAreaCode, CustPhoneNumber*. Щоб призначити стовпець CustomerID як первинний ключ, треба позначити його таким під час створення таблиці. Обмеження первинного ключа гарантує, що кожне значення CustomerID, як і рядок, якому він відповідає; є унікальним, що дозволяє однозначно ідентифікувати кожного клієнта в таблиці.

Рисунок 5 – Структура таблиці Customers

Зовнішній ключ — це поле або комбінація полів у таблиці, яке посилається на первинний ключ іншої таблиці. Він встановлює логічний зв’язок (або асоціацію) між двома таблицями на основі їхніх спільних значень ключів. Зовнішній ключ представляє рядок таблиці, що залежить від іншого рядку іншої таблиці, що має у первинному ключі аналогічне значення. Зовнішні ключі дозволяють встановлювати між таблицями різні типи зв’язків, наприклад “один до багатьох” і “багато до багатьох” (про ці та інші зв'язки буде надано інформацію нижче). Вони забезпечують цілісність посилань, тобто під час зв’язування таблиць зберігається узгодженість даних.

Щоб проілюструвати це, розглянемо дві таблиці на Рисунку 6: попередню "Customers" та нову "Orders", яка є залежною від першої і має наступні стовпці: **OrderID** (тут це первинний ключ), *OrderDate, ShipDate,* **CustomerID** же буде стовпцем зовнішнього ключа, що посилається на стовпець первинного ключа CustomerID у таблиці "Customers".

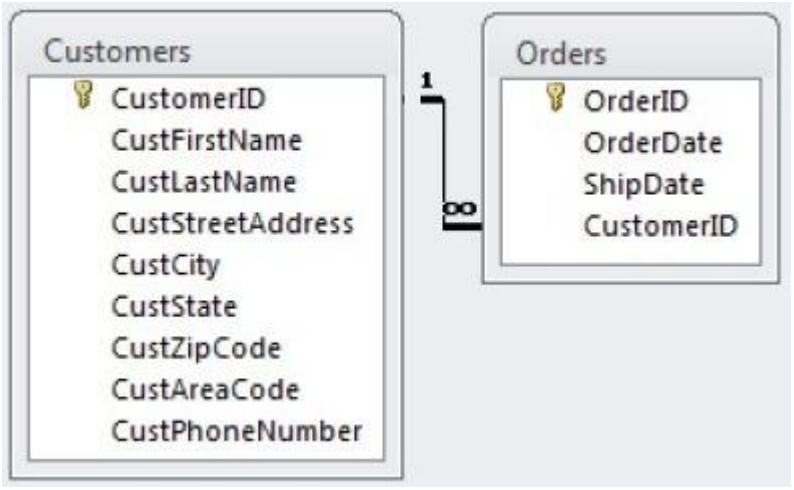

Рисунок 6 – Структура таблиці Orders, зв’язок з таблицею Customers

Встановлюючи цей зв’язок зовнішнього ключа, ви гарантуєте, що кожне замовлення в таблиці "Orders" пов’язане з дійсним клієнтом, присутнім у таблиці "Customers". Обмеження зовнішнього ключа гарантує, що значення в стовпці "CustomerID" таблиці "Orders" існують як значення первинного ключа в таблиці "Customers", підтримуючи цілісність даних і запобігаючи втрату зв`язку записів.

Таким чином, первинні ключі забезпечують унікальний ідентифікатор для кожного запису в таблиці, тоді як зовнішні ключі встановлюють зв’язки між таблицями на основі значень первинного ключа. Ці ключові поняття мають вирішальне значення для підтримки узгодженості даних, забезпечення ефективного пошуку даних і підтримки загальної цілісності реляційних баз даних.

2.2 Типи реляцій

У контексті реляційних баз даних існує кілька типів зв’язків, які можуть існувати між таблицями. Далі наведено поширені типи відношень:

1. Відношення один до багатьох (1:N): у зв’язку "один до багатьох" один запис у таблиці пов’язаний з кількома записами в іншій таблиці. Це найпоширеніший тип зв'язку, який зустрічається в розробці баз даних. Наприклад, розглянемо базу даних, у якій є дві таблиці: “Матері” та “Діти”. Жінка може мамою декількох дітей, але кожна дитина має лише одну матір. У цьому випадку зв’язок між таблицею “Матері” та “Діти” є зв’язком “один до багатьох”. Таблиця “Матері” матиме стовпець первинного ключа (наприклад, **MotherID**), а таблиця “Діти” матиме стовпець зовнішнього ключа **MotherID**, який посилатиметься на первинний ключ таблиці “Матері” (рисунок 7).

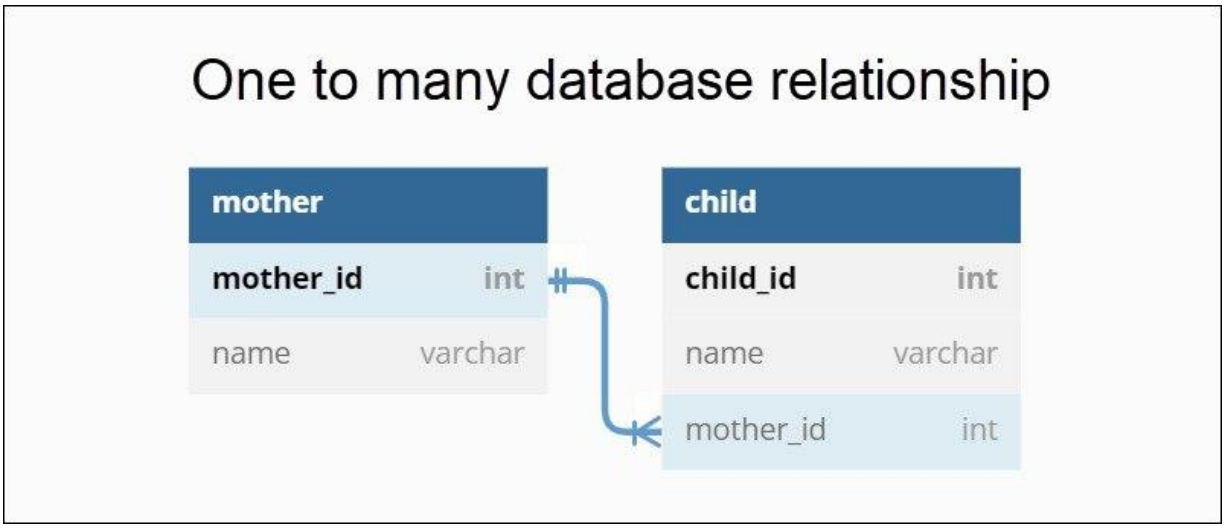

Рисунок 7 – Структура таблиці з зв’язком один-до-багатьох

2. Відношення “багато до багатьох” (N:N): відношення “багато до багатьох” існує, коли кілька записів в одній таблиці пов’язані з декількома записами в іншій таблиці. У такому зв’язку третя таблиця, яка називається таблицею з’єднання або асоціативною таблицею, використовується для з’єднання двох таблиць. Для прикладу розглянемо базу даних для книжкового магазину. У нас може бути дві таблиці: “Книги” та “Автори”. Кілька книг можуть бути написані декількома авторами, і кожен автор може мати декілька книг. У цьому випадку використовується сполучна таблиця, яка часто називається “BookAuthors” або подібна, яка містить зовнішні ключі з обох таблиць “Books” і “Authors”. Ця сполучна таблиця встановлює зв’язок “багато до багатьох” між книгами та авторами (рисунок 8).

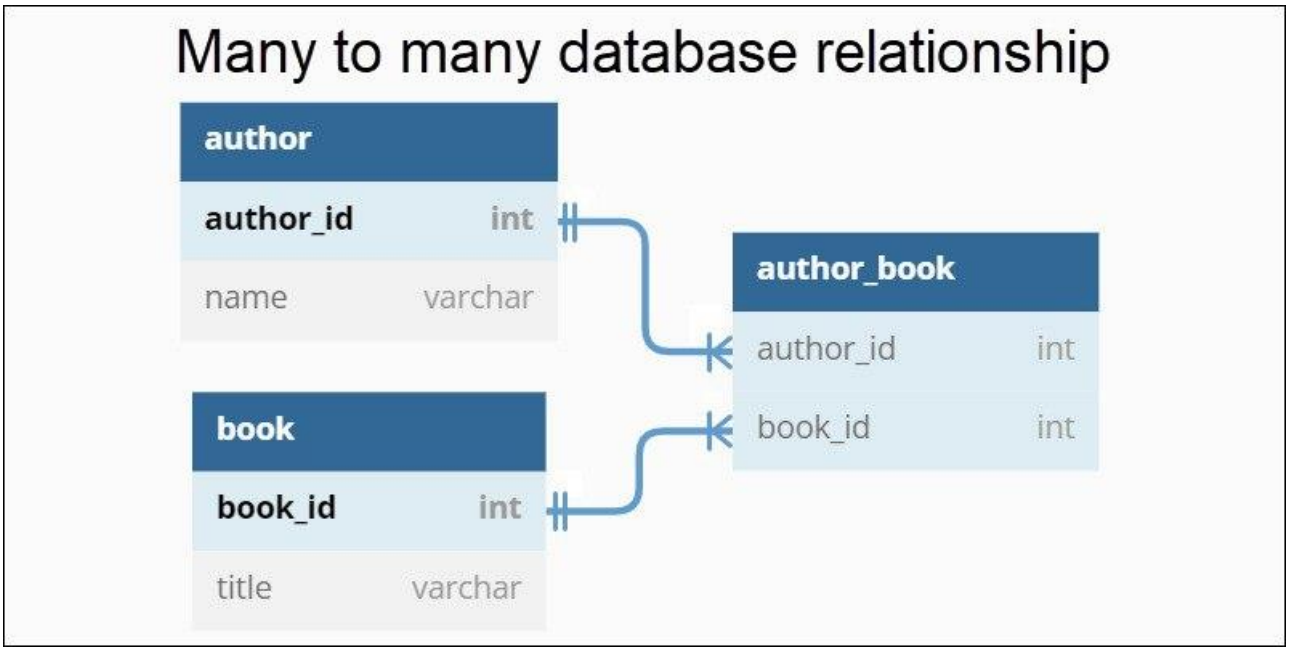

Рисунок 8 – Структура таблиць з зв’язком багато-до-багатьох

3. Відношення один до одного (1:1): зв’язок один-до-одного виникає, коли кожен запис в одній таблиці пов’язаний точно з одним записом в іншій таблиці (рисунок 9). У цьому типі зв’язку первинний ключ однієї таблиці пов’язаний з зовнішнім ключем іншої таблиці. Одним із практичних прикладів може бути база даних для записів про країни, де таблиця “Міста” містить стовпець первинного ключа (наприклад, **CityID**), а таблиця “Країни” має стовпець зовнішнього ключа (наприклад, **CityID**), що посилається на

первинний ключ таблиці “Міста”. Таке розташування дозволяє зберігати в окремій таблиці додаткові відомості про міста (наприклад кількість населення, дату заснування, щільність населення; все це окремо від потенційної кількості населення країни, дати незалежності і т. п.).

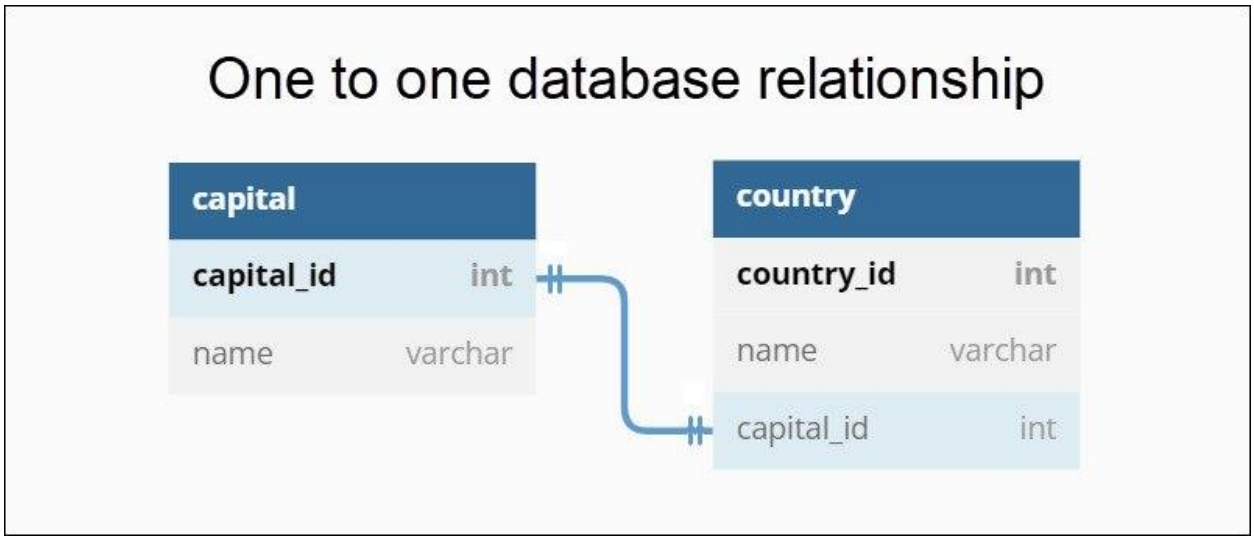

Рисунок 9 – Структура таблиць з зв’язком один-до-одного

4. Відношення “Сам-до-Себе”: самопосилання існує, коли таблиця має зв’язок із самою собою. Це відбувається, коли записи в одній таблиці пов’язані один з одним. Наприклад, розглянемо організаційну діаграму, де кожен працівник має керівника, який також є працівником. Таблиця, що представляє співробітників, міститиме стовпець зовнішнього ключа, який посилається на первинний ключ іншого запису в тій самій таблиці для встановлення ієрархічного зв’язку (рисунок 10).

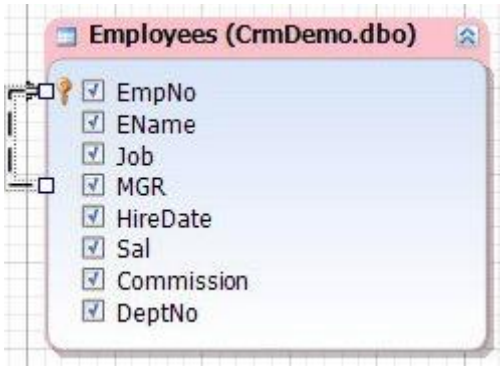

Рисунок 10 – Ілюстрація відношення сам-до-себе

Також варто звернути увагу на те, що існують модифікації цих реляцій, де зв'язки можуть бути опціональні, у такому випадку з відповідної сторони буде

позначення як; 1:0vN, 1:0v1 і т. п. Для того щоб вказати що може існувати нуль або декілька, один або жодного з записів. Типи зв’язків допомагають представити різні асоціації та залежності між таблицями в реляційній базі даних. Вони забезпечують механізми організації та зв’язування даних, забезпечуючи ефективний пошук і підтримуючи цілісність. Розуміння та правильне визначення цих зв’язків має вирішальне значення для розробки ефективної схеми бази даних.

РОЗДІЛ 3. МОДЕЛІ ТРАНЗАКЦІЙ

Модель транзакцій описує спосіб, у який можна використовувати транзакції в потоках повідомлень для виконання певних завдань і результатів[5].

Потік повідомлень складається з наступних складових частин:

- Джерело вхідних даних
- Конвеєр обробки повідомлень або ж логіка, яка визначається послідовністю вузлів
- Нуль або більше зовнішніх ресурсів, доступ до яких здійснюється під час потоку
- Нуль або більше вихідних цілей

Кроки нижче представляють типову послідовність подій потоку повідомлень у транзакції:

1. Повідомлення береться з джерела вхідних даних; наприклад, черги.
2. Дані зчитуються з або записуються на один чи більше зовнішніх ресурсів; наприклад, базу даних.
3. Повідомлення надсилається до вихідної цілі; наприклад, черги.
4. Система переходить у стан очікування наступного вхідного повідомлення.

Під час виконання цієї послідовності дій стан даних у системі змінюється, незалежно від кількості зовнішніх ресурсів, до яких отримує доступ потік повідомлень, і від того, чи створюється вихідне повідомлення.

3.1 ACID модель

Серед типів моделей обробки транзакцій ACID модель, слідує наступним характеристикам:

1. **Atomicity** (Атомарність): Гарантує, що всі зміни даних розглядаються як єдина неподільна операція. Це гарантує, що або всі зміни в транзакції успішно виконаються, або жодна з них. Наприклад, у додатку для переказу коштів між рахунками, атомарність гарантуватиме те, що якщо дебетування здійснено з одного рахунку, відповідний кредит також буде зараховано на інший рахунок.
2. **Consistency** (Узгодженість): Передбачає, що дані залишаються в узгодженому стані на початку та в кінці транзакції. Це забезпечує дотримання обмежень цілісності та відповідності загальному стану даних протягом транзакції. Аналогічний приклад переказу коштів - узгодженість гарантує, що загальна сума коштів на обох рахунках залишається незмінною до та після кожної транзакції.
3. **Isolation** (Ізоляція): Регулює одночасне виконання транзакцій, гарантуючи, що вони працюють незалежно та не заважають одна одній. Транзакції виглядають так, ніби вони виконуються послідовно, навіть якщо обробляються одночасно. У сценарії переказу коштів ізоляція гарантує, що транзакції спостерігають за переказаними коштами в одному або іншому обліковому записі, але не в обох або жодному.
4. **Durability** (Стійкість): Гарантує постійність змін даних після успішного завершення транзакції, незалежно від системних або програмних збоїв. Це гарантує, що зміни, внесені під час транзакції, збережуться й не будуть скасовані. Таким чином кошти на рахунках збережуться навіть якщо сервер дасть збій і відключиться.

Властивості ACID сукупно забезпечують надійність, цілісність і послідовність транзакційних операцій у системі баз даних. Дотримуючись цих принципів, системи управління базами даних забезпечують впевненість користувачів у збереженні цілісності даних.

3.2 BASE модель

Серед типів моделей обробки транзакцій ACID модель, слідує наступним характеристикам:

1. **Basically available** (Легкодоступна): Цей принцип підкреслює постійну доступність системи бази даних для задоволення запитів користувачів, навіть якщо миттєвий доступ до всіх даних не може бути забезпечений. Хоча можуть виникати тимчасові періоди недоступності, система розроблена таким чином, щоб мінімізувати такі випадки та швидко відновлюватися після збоїв. Завдяки пріоритетності доступності, користувачі можуть постійно взаємодіяти з базою даних, хоча й з потенційними змінами часу відповіді або доступності даних.
2. **Soft state** (Динамічна): Принцип вказує, що стан бази даних може динамічно змінюватися з часом, навіть за відсутності явних дій користувача. Такі фактори, як фонові процеси та оновлення даних, сприяють змінюванню цього стану. Очікується, що добре спроектована система бази даних точно і ефективно керуватиме такими змінами, запобігаючи пошкодженню чи втраті даних.
3. **Eventually consistent** (Кінцева узгодженість): Цей принцип зосереджує на тому, що в кінці-кінців дані будуть консистентні, враховуючи зміни, які будуть відбуватися з часом, навіть після обробки всіх запитів від користувача. Хоча миттєва узгодженість, яку забезпечують традиційні ACID бази даних, може не бути гарантованою, система баз даних поступово рухається до узгодженого стану. Це дає час для розсилки оновлень даних (по різним вузлам, наприклад).

Таким чином, властивості BASE утворюють основу сучасних систем керування базами даних, забезпечуючи високу доступність, пристосовуючись до часто змінюваних станів і досягаючи кінцевої узгодженості, хоч і з плином часу. Їх впровадження має вирішальне значення для підтримки великомасштабних стійких систем у різних сферах, сприяння бездоганній взаємодії з користувачем та ефективного керування даними.

3.3 Порівняння ACID та BASE

Ознайомившись з моделями транзакцій, що були описані вище, їх сильними та слабкими сторонами, можна підсумувати (таблиця 3.1):

Таблиця 3.1

Критерій	**ACID**	**BASE**
Узгодженість даних	Сильно узгоджені	Слабо узгоджені
Схема даних	Комплексна	Проста
Фокус	Стійка	Легкодоступна
Ізоляція	Транзакції незалежні	Останній запис виграє
Автентифікація	Вдосконалена	Базова
Шифрування	Мають вбудовані механізми шифрування	Не мають великих можливостей для шифрування
Масштабованість	Можливості масштабування обмежені	Надаються механізми для легкого масштабування

З цією порівняльною характеристикою можна мати уяву що до можливостей та потенційно обрати тип СУБД, що задовольнить вашим потребам.

РОЗДІЛ 4. ВИМІРЮВАННЯ ШВИДКОДІЇ

Порівняння швидкодії та ефективності різних СУБД у розв’язку схожих задач – це суть процесу. Він є вкрай важливим для того, щоб інженери програмного забезпечення мали повне представлення що до характеристик роботи СУБД, оскільки треба відслідковувати вразливі міста системи, чи обирати з поміж декількох програмних продуктів. І тому, якщо на початковому етапі було зроблено неправильний вибір, не було взято до уваги слабкі сторони та обмеження СУБД, не передбачено потреби, які можуть з’явитись у майбутньому, то в подальшому знадобиться багато цінного часу на виправлення недоліків, або ж міграцію на іншу СУБД. Тому краще заздалегідь обирати правильний варіант.

Саме тому порівняння швидкодії СУБД як у межах одного класу, так і між представниками різних класів, є надзвичайно важливим для того, щоб мати повну картину. Це допоможе обрати систему управління базами даних, що підходить до вимог проекту.

Вимірювання швидкодії передбачає створення сценаріїв навантаження СУБД, вимірювання показників продуктивності та порівняння отриманих результатів. В процесі будуть отримані відповідні метрики: час відгуку, пропускна здатність ті інші. На основі них і можна буде робити висновки що до потреби оптимізації існуючої програмної системи, чи просто зроблені висновки що до переваги одної СУБД над іншою.

Підсумовуючи, порівняння ефективності СУБД може стати важливою підмогою для розробників ПЗ, з цим вони зможуть зрозуміти яка СУБД більше підходить під потреби. Наприклад, для високої структурованості даних, гарантії безпеки транзакцій і узгодженості даних – обираємо класичні реляційні СУБД з ACID моделлю. По іншу сторону – NoSQL СУБД засновані на BASE моделі, які дають широкі можливості що до масштабування, легкодоступні та гнучкі.

Але, варте уваги те, що ключові показники ефективності не є єдиним фактором при прийнятті рішення. Дані, отримані від вимірювання швидкодії БД, повинні розглядатися в контексті конкретного проекту і вимог до нього. Варто враховувати і такі нюанси, як інструменти, що застосовуються для розробки та, наприклад, бібліотеки мов програмування, їх зручність та актуальність документації тощо. Усе це впливає на продуктивність процесу розробки програмної системи.

4.1 Процес порівняння швидкодії

Під час нього оцінюються та порівнюються характеристики швидкодії та стабільності різних СУБД, це все може проводитись як в штучно створених, так і в реальних умовах. Для цього створюються та виконуються сценарії навантажень, під час яких і замірюються потрібні показники, аналізуються отримані дані.

Під сценаріями навантажень ми розуміємо набір різних операцій: створення записів, їх читання або видалення тощо. Таким чином моделюються обсяги роботи, з якими БД може стикатися, важливим є розуміння розмірів допустимих навантажень і утримання їх у реальних рамках, тих що дійсно можуть з’являтися при використанні системи наживо.

Далі познайомимось з ключовими показниками швидкодії:

1. Час виконання – відображає довжину часового проміжку, під час якого виконувався, та був виконаний обраний сценарій навантаження. Окремо від інших показників він дає мало інформації, але може бути порівняний з часом виконання аналогічного сценарію на іншій СУБД.
2. Пропускна здатність – дуже важлива характеристика системи, особливо для такої, що використовується великою кількістю користувачів. Характеризує кількість операцій, що можуть бути оброблені програмною системою за одиницю часу. За допомогою неї можна оцінювати працездатність системи під час інтенсивних навантажень, порівнюючи її з кількістю запитів, що генерують користувачі під час користування.

3. Затримка (середня, мінімальна та максимальна) – відображає час за який може оброблюватися запит. Правильний моніторинг цієї характеристики може допомогти виявити слабкі місця системи, наприклад повільні модулі, або ж може свідчити про утворення великих черг запитів. Виявлення і усунення таких недоліків є вкрай значущим для систем, що працюють у реальному часі.
4. Код повернення – для кожної з виконаних операцій означає її успішність, збирання статистики що до цього показника допоможе виявляти запити, що не обробились. Таким чином з’явиться можливість знайти спільну проблему, при її наявності, та усунути її.
5. Тип операції – для показників під номерами 2, 3 та 4 можна виконати поділ за типом операції: INSERT, READ, UPDATE тощо. За допомогою цього буде отримана більш детальна статистика, яка допоможе конкретизувати та обмежити місця пошуку слабкого модуля системи.

Наведений перелік не є повним, можливо отримати ще такі показники, як відсоткове завантаження ядер процесору (може свідчити про неправильне використання можливостей багатопоточності), обсяги оперативної пам’яті (вказує на можливі витоки пам’яті), кількість активних з’єднань тощо. Таким чином ми маємо широкий набір інструментів, які є вкрай корисними для аналізу продуктивності системи.

4.2 Типові помилки

Під час порівняння СУБД треба уникати помилок, які ведуть до отримання неправильних та не пов’язаних з реальністю даних. А саме слід дотримуватись наступного:

1. Формувати сценарії навантажень бази даних таким чином, щоб вони відповідали можливим навантаженням, що зустрічаються під час експлуатації системи.
2. Перед запуском треба перевіряти параметри БД та налаштування оточення, таким чином можна пересвідчитись, що робота бази даних під час виконання сценарію буде максимально схожа на роботу у продуктовому оточенні.
3. Не використовувати старі та неактуальні версії СУБД та бібліотек, таким чином ви зменшите шанс отримання недостовірної інформацію через те, що у новій версії БД чи бібліотеки були виправлені суттєві недоліки попередніх версій.
4. Незнання теоретичної бази для тестування та некоректне використання інструментів може призвести до отримання фіктивних даних, чи їх невірного трактування, що призведе до прийняття неправильних рішень.
5. У випадку, якщо у сценарії навантаження є виконання операцій, які потребують наявності початкових даних у БД, наприклад операції READ, то час, який буде витрачено на наповнення бази даних і її підготовки до виконання сценарії, матиме вплив на загальну статистику виконання навантаження.
6. Використання недостатньої, або невідповідної дійсності на продуктовому оточенні, кількості ресурсів таких як: кількість та швидкість пам’яті, потужність процесору та кількість його ядер, пропускної здатності комп’ютерних мереж. Ця помилка теє може призвести до отримання невідповідних даних.

7. Недостатній розмір навантаження або його протяжність. Таким чином можуть залишитися прихованими проблеми з формуванням черг обробки запитів, накопиченням фрагментованих даних та так званих “шийок пляшки” – “вузьких” місць системи з недостатньою пропускною здатністю. Тому треба уважно створювати сценарії навантажень, це допоможе уникнути подібних моментів.

Загалом, якщо пам’ятати про описані вище типові помилки та правильним чином трактувати діагностичні дані, то розробники зможуть мати актуальну та правильну інформацію що до стійкості системи до навантажень, стабільності та швидкості її роботи, що значно полегшить прийняття рішень.

РОЗДІЛ 5. ОБ'ЄКТИ ТА ЗАСОБИ ДОСЛІДЖЕННЯ

Для дослідження у даній роботі були обрані наступні об'єкти та засоби, що надають необхідну для порівняння різних типів СУБД функціональність. До обраних засобів належить Yahoo! Cloud Serving Benchmark (YCSB) – фреймворк для порівняння різних видів баз даних. Цей вибір був зумовлений наступними критеріями: фреймворк реалізовано на популярній об'єктно-орієнтовній мові програмування Java, він підтримує багато різних СУБД та дає можливість легко додати підтримку нових, має гнучку конфігурацію сценаріїв навантажень. До об'єктів дослідження належать різні представники NoSQL та реляційних СУБД, у ході подальшої роботи, за допомогою YCSB буде проведено тестування за допомогою декількох сценаріїв навантажень, які виконують операції над даними, та зібрано статистику, що до ключових показників швидкодії цих систем. Перелік систем та їх опис буде викладено нижче.

5.1 Yahoo! Cloud Serving Benchmark

YCSB – Java фреймворк для тестування швидкодії різних СУБД, він надає інформацію про такі метрики: пропускну здатність, затримки виконання запитів – тобто показники, що дають прозору характеристику швидкодії БД.

Основою роботи з цим фреймворком є конфігурація сценаріїв навантажень (workloads) (додаток А), вони покривають основні операції: створення, читання, оновлення записів[7]. Ці сценарії є універсальними та можуть застосовуватися до усіх типів СУБД, що підтримуються фреймворком. За рахунок того, що користувачу доступний вихідний програмний код, відкривається можливість для додавання підтримки інших СУБД та розширення існуючих функцій.

Для того щоб додати підтримку нової системи управління базами даних користувачу потрібно реалізувати клас бази даних (додаток В) з методами читання, сканування, оновлення, вставки і видалення, безпосередньою логікою виконання запитів. Все інше бере на себе YCSB – він управляє підключеннями, запускає сценарії навантажень, формує дані для виконання цих сценаріїв. Від користувача потрібна лише реалізація дуже специфічною логіки.

Таким чином, YCSB надає базові класи і інтерфейси на базі яких кінцевий користувач за потреби може реалізовувати або модифікувати, йому не треба буде самостійно писати багато коду.

5.2 PostgreSQL

PostgreSQL – це багатофункціональна та високопродуктивна система управління реляційними базами даних з відкритим вихідним кодом, вона є популярною завдяки своїй надійності та можливостям масштабування[8]. Була обрана як представник класу реляційних СУБД тому, що широко застосується розробниками ПЗ і пропонує їм широкий набір функцій, а саме:

- Потужний механізм запитів: PostgreSQL реалізовує повний набір стандартних SQL функцій та включає застосування складних запитів, підзапитів, загальних табличних виразів та віконних функцій. Також пропонує такі нетривіальні для реляційних СУБД функції, як підтримку повнотекстового пошуку та JSON формату даних.
- Розширюваність: ця СУБД дає користувачам інструменти для створення користувацьких типів даних, операторів та функцій над ними; таким чином збільшуючи можливість адаптування системи під конкретні потреби, значно спрощуючи розробку.

- Управління паралелізмом: встановлює стійкий багатоверсіонний контроль паралелізму, переосмислюючи концепцію транзакцій. Таким чином дозволяючи багатокористувацьким системам виконувати транзакції ізольовано від поточного стану бази даних тому, що вони оперують над так званим знімком бази даних, який може відрізнятися від реального стану БД.
- Реплікація: існують включені в систему механізми реплікацію, підтримки синхронних і асинхронних її методів, що забезпечує високу відмовостійкість, легку доступність даних та їх резервування. І все це можливо налаштувати одразу.
- Безпеку даних та їх цілісність: СУБД надає розширені механізми шифрування даних, автентифікації користувачів, SSL та можливість налаштування доступу до даних на рівні окремих об’єктів. Також реалізовані класичні методи підтримання цілісності даних: первинні та зовнішні ключі, обмеження.
- Діалекти популярних мов програмування: у PostgreSQL можна визначати користувацькі функції, написані на таких мовах програмування: C#, C++, C, Java, Python, PHP тощо, це широко покриває ринкове різноманіття, надаючи можливість розробникам серверних частин систем лаконічно впроваджувати потрібний функціонал.

Всі ці потужні засоби роблять PostgreSQL виключно конкурентоспроможним у сфері РСУБД.

До сильних сторін слід віднести:

- Розширюваність: разом з можливістю визначення користувацьких функцій і типів, система управління дозволяє розподіляти логіку між шарами бази даних і сервером.
- Масштабованість: за допомогою як горизонтального масштабування, яке потребує від розробників високих професійних навичок; так і вертикального за допомогою збільшень розрахункових потужностей серверу, базу даних на PostgreSQL можна вкрай ефективно пришвидшувати і збільшувати у розмірах, реалізовувати розподіленість.
- Відкритий вихідний код: не обмежує користувачів рамками, надає можливість вбудовувати потрібні функції.
- Сумісність з різними платформами: PostgreSQL працює та отримає регулярні оновлення для Windows, macOS та Linux, легко встановлюється та розгортається на них.

До слабких сторін належать:

- Комплексність: БД на PostgreSQL можуть бути складними в обслуговуванні та підтримці.
- Крива навчання: СУБД пропонує вкрай великий набір різноманітних функцій, які можуть бути складними, як для початківців, так і для досвідчених розробників.
- Ресурси: PostgreSQL при неправильному налаштуванні та використанні може використовувати невиправдано великий обсяг оперативної пам’яті та ресурсів процесору тощо

Таким чином, PostgreSQL — це потужна реляційна СУБД із широкими можливостями, що забезпечують гнучкість, масштабованість і надійність. Йому віддають перевагу велика кількість розробників та компаній.

5.3 Redis

Redis – це NoSQL система управління базами даних з відкритим вихідним кодом, що працює з структурами даних типу “ключ-значення”.[9] Важливо зазначити, що Redis є представником in-memory БД (тобто тих, що зберігаються в оперативній пам’яті). Часто її застосовують у ролі кеша, або брокера повідомлень, її специфіка побудови дає можливість ефективно використовувати у цих задачах.
Плюсами цієї системи є:

- Швидкість: Redis має дуже низькі затримки виконання операцій тому що дані зберігаються у оперативній пам’яті – легкодоступному і швидкому ресурсу комп’ютера.
- Брокер повідомлень: підтримка можливості підписки на повідомлення дає змогу використовувати Redis у ролі брокера повідомлень – модулі системи, за допомогою якого інші модулі здійснюють обмін інформацією. Це виключно корисне архітектурне рішення.
- Кешування: завдяки тому, що Redis є in-memory репозиторієм, його зручно використовувати для зберігання даних, які можуть бути постійно потрібними.
- Кластеризація: Redis підтримує функцію горизонтального масштабування, дозволяючи розділяти дані по кільком вузлам, що дає змогу використовувати його у високонавантажених системах.
- Структури даних: має широкий набір таких структур, як: хеш-таблиці, списки, відсортовані списки тощо. Таке різноманіття робить систему більш гнучкою і швидкою, при умові, що ці структури даних використовуються коректно, як і алгоритми над ними.

До недоліків можна віднести:

- Синтаксис запитів: ця система управління базами даних відносить до типу NoSQL і не має стандартизованої структури запитів, як наприклад SQL, до того ж існуючі операції, у порівнянні з SQL операціями, можуть вважатися простими.
- Розмір: через те, що дані зберігаються у оперативній пам’яті, репозиторій має вкрай обмежений розмір, у випадку якщо порівнювати з базами даних, які зберігаються на диску; він значно програє. Це може стати суттєвим недоліком, якщо системі потрібен великий обсяг легкодоступних даних, наприклад.
- Стійкість до збоїв: знову ж таки, через те, що дані зберігаються у оперативній пам’яті, то при вимкненні сервера вони пропадають. Але Redis пропонує можливість відвантаження даних на диск, що частково компенсує цей недолік.

Підсумовуючи, Redis — це потужне сховище даних у оперативній пам’яті, яке вирізняється продуктивністю, простотою та можливостями кешування. У своїй ніші він є одним з лідерів.

5.4 Memcached

Memcached — це широко система розподіленого кешування, розроблена для підвищення продуктивності та масштабованості шляхом кешування даних, до яких часто звертаються, у оперативній пам’яті[10]. Далі розглянемо особливості цієї системи, її сильні та слабкі сторони.

Особливості:

- Розподілене кешування: Memcached працює як система розподіленого кешування, що дозволяє кільком вузлам працювати разом у кластері. Цей розподілений характер забезпечує горизонтальну масштабованість, оскільки

дані можна розподіляти між декількома серверами, покращуючи продуктивність і ємність.

- Місце зберігання даних: зберігає дані в оперативній пам’яті, що забезпечує надзвичайно швидкий час доступу. Зберігаючи дані, до яких часто звертаються, у оперативній пам’яті, виключається потреба в операціях запису/читання з диска, що призводить до зменшення затримки.
- Модель даних: використовує просту модель даних “ключ-значення”, де дані зберігаються та витягуються на основі унікальних ключів. Ця простота полегшує роботу з даними та забезпечує швидкі операції читання та запису.
- Термін зберігання даних: Memcached підтримує налаштування часу дії кешованих даних. Ця функція дозволяє розробникам контролювати тривалість, протягом якої дані залишаються в оперативній пам’яті, гарантуючи, що неактуальні дані не будуть займати місце. Крім того, коли обсяг кешу досягає ліміту, Memcached видаляє дані, до яких не часто звертаються.

Сильні сторони:

- Пришвидшення: кешуючи часто використовувані дані в пам’яті, Memcached значно скорочує час, необхідний для їх. Це призводить до покращення продуктивності системи.
- Масштабованість: розподілена архітектура Memcached дозволяє додаткам масштабуватися горизонтально, додаючи більше серверів до кластера. Це дає змогу обробляти більші обсяги даних і збільшити трафік користувачів без втрати продуктивності.
- Висока пропускна здатність: Memcached розроблено для обробки великої кількості одночасних запитів, що робить його придатним для додатків із високим трафіком. Його ефективне зберігання в пам’яті та легка конструкція сприяють його здатності виконувати великий обсяг операцій.
- Зменшення навантаження на базу даних: шляхом кешування даних у пам’яті Memcached зменшує навантаження на серверні бази даних або системи зберігання. Це призводить до скорочення часу запиту до бази даних,

зменшення дискового введення/виведення та покращення загальної продуктивності бази даних.

Слабкі сторони:

- Обмежені операції над даними: Memcached в основному розроблено для зберігання та пошуку ключів-значення, не має розширених можливостей запитів або підтримки складних структур даних. Він може не підійти для програм, які вимагають складних операцій обробки даних.
- Нестійкість даних: не має механізмів, що копіюють дані з оперативної пам’яті на диск, що може призвести до втрати даних.
- Відсутність функцій безпеки даних: Memcached не надає вбудованих механізмів автентифікації чи шифрування. Контроль доступу та безпека даних повинні бути реалізовані на рівні системи або мережі.
- Фрагментація даних: оскільки дані можуть розподілятися між кількома вузлами в кластері, алгоритм розподілу може призвести до фрагментації даних. Це може викликати збільшення мережевого трафіку, що вплине на загальну продуктивність.

Таким чином, Memcached має достатній набір функцій для використання його у ролі швидкого кешу, навіть разом з усіма його слабкими сторонами, він має достатню конкурентоспроможність.

5.5 MongoDB

MongoDB – це документо-орієнтовна NoSQL система управління базами даних. Вона призначена для обробки неструктурованих та частково структурованих даних, що забезпечує гнучкість для використання у сучасних додатках[11].

Особливості:

- Масштабованість: має механізми горизонтального масштабування за рахунок виконання сегментування даних.
- Гнучка модель даних: у MongoDB структура даних можуть змінюватись від документа до документа.
- Багата мова запитів: підтримує індексування, сортування та агрегування. Також підтримує повнотекстовий пошук, що дозволяє зберігати та обробляти складні дані.

Плюси:

- Гнучкість: безсхемна природа MongoDB і її динамічна модель даних роблять її придатною для використання у системах, де дані мають різний формат.
- Масштабованість: автоматичне сегментування реалізує горизонтальне масштабування, що робить СУБД придатною для обробки великих обсягів даних та використання у системах з великим навантаженням.
- Продуктивність: завдяки механізму зберігання з обробкою внутрішньої структури документу забезпечується ефективне індексуванню даних, що значно прискорює їх пошук
- Реплікація: має вбудовані можливості реплікації, що збільшують їх стійкість.

Мінуси:

- Використання пам'яті: механізм зберігання з обробкою внутрішньої структури документу може споживати значний обсяг оперативної пам'яті, особливо під час роботи з великою кількістю вкладених даних.
- Крива навчання: MongoDB відхиляється від традиційних реляційних баз даних, користувачам може знадобитися більше часу, щоб освоїти її механізми.

Підсумовуючи, документо-орієнтована модель MongoDB, масштабованість, продуктивність і гнучкість роблять її популярним вибором для сучасних додатків.

5.6 Aerospike

Aerospike - це високопродуктивна NoSQL система управління базами даних, призначена для роботи з системами, що інтенсивно використовують дані[12]. Вона пропонує широкий набір функцій та можливостей з обробки JSON формату, що робить її придатною для використання у ролі репозиторію документів.

До особливостей СУБД відносяться:

- Модель “ключ-значення”: Aerospike побудований на моделі даних “ключ-значення”, що дозволяє ефективно отримувати та зберігати дані на основі унікальних ключів. Ця простота забезпечує швидку та передбачувану продуктивність операцій читання та запису.
- Документоорієнтовність: незважаючи на те, що Aerospike побудований на моделі даних “ключ-значення”, він використовується як репозиторій для документів.
- Флеш-оптимізованість: може використовувати твердотільні накопичувачі як основний носій даних, забезпечуючи високошвидкісний доступ до даних та операції з малою затримкою.
- Швидке індексування: Aerospike використовує передові методи індексування, включаючи первинні та вторинні індекси, бітові індекси та індекси діапазону. Ці механізми індексації полегшують ефективний пошук даних.

До сильних сторін можна віднести:

- Висока продуктивність: Aerospike забезпечує виняткову продуктивність, операції з малими затримками та високу пропускну спроможність. Його ефективні методи зберігання даних та управління пам'яттю забезпечують швидкий доступ до даних та їх обробку.
- Масштабованість: розподілена архітектура Aerospike забезпечує плавну масштабованість за рахунок додавання вузлів у кластер, розміщення зростаючих обсягів даних та обробки зростаючих навантажень трафіку, що забезпечує стабільну продуктивність у міру розширення системи.

До слабких сторін належать:

- Складність конфігурації: встановлення та налаштування Aerospike для оптимальної продуктивності може бути складним завданням, яке потребує ретельного розгляду різних параметрів, таких як розподіл даних, коефіцієнт реплікації та конфігурації сховища.
- Обмежені можливості запитів: хоча Aerospike і підтримує первинні та вторинні індекси, насамперед він оптимізований для шаблонів доступу “ключ-значення”. Складні запити, що включають кілька об'єднань або агрегацій, можуть мати малу продуктивність.
- Пропрієтарність: комерційна ліцензія Aerospike та структура ціноутворення можуть бути проблемою для організацій з бюджетними обмеженнями.

Таким чином, модель даних Aerospike “ключ-значення”, масштабованість, оптимізація для флеш-пам'яті; затверджують її як високопродуктивну бази даних NoSQL. Однак слід враховувати можливі складнощі конфігурації, криву навчання, обмежені можливості запитів.

РОЗДІЛ 6. ЕКСПЕРИМЕНТАЛЬНЕ ПОРІВНЯННЯ СУБД

Для виконання цього було обрано такі СУБД: PostgreSQL, Redis, Memcached, MongoDB, Aerospike. Вони є представниками реляційних та NoSQL систем управління базами даних. PostgreSQL підтримує формат даних JSON, тому у цій роботі вона використовується як база даних для документів, без застосування реляцій. Для кожного класу СУБД, а також для кожного типу у NoSQL класі було проведено групове тестування трьома типами навантажень з різним розміром даних:

1. Навантаження операціями READ – у ході цього навантаження у фазі підготовки у базу завантажуються дані (параметри швидкодії під час цього не вимірюються), після цього велику кількість разів виконується їх зчитування.
2. Навантаження операціями INSERT – під час цього навантаження у фазі підготовки не виконується ніяких дій, після цього у базу записуються дані та заміряються параметри швидкодії операції запису даних, на цьому навантаження завершується.
3. Навантаження операціями UPDATE – у ході даного навантаження під часи фази підготовки у базу завантажуються дані, під час їх завантаження не заміряються показники швидкодії; після цього проводиться виконання операцій оновлення цих даних і збирається статистика.

Всі бази даних було конфігуровано таким чином, щоб вони зберігали документ формату JSON, відповідно виконувалось порівняння швидкодії цих СУБД, у випадку використання їх у ролі сховища для зберігання документів.

6.1 Схема даних

Для отримання показників швидкодії на різних сценаріях навантажень, за умови того що СУБД тестуються у ролі репозиторіїв для документів, було обрано документ формату JSON (рисунок 10), що має наступну структуру:

```
{
    field0: "...",
    field1: "...",
    field2: "...",
    field3: "...",
    field4: "...",
    field5: "...",
    field6: "...",
    field7: "...",
    field8: "...",
    field9: "...",
}
```

Рисунок 10 – Структура обраного JSON документа.

Документ складається з десяти полів, що містять рядки довжиною 100 символів. Формат та зміст документу було обрано з огляду на підтримку його обраними СУБД та вплив розміру даних на швидкість виконання операцій. Роль первинного ключа для зберігання записів виконує рядок довжиною 255 символів, тому схема даних (рисунок 11) має наступний вигляд:

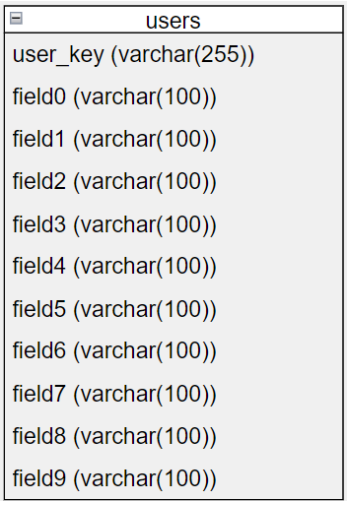

Рисунок 11 – Схема даних

6.2 Хід тестування

Для тестування над базами даних виконувались наступні типи операцій: вставка запису (INSERT), читка запису (READ), оновлення запису (UPDATE).
Під час тестування у кожну СУБД відправлялося по 1000 запитів одного з трьох типів. Загальний сценарій тестування мав наступний вигляд:

1. Фаза підготовки (load) – заповнення бази даних відповідно до обраних розміру та схеми даних.
2. Зчитування записів (run workload read) – зчитування заданої кількості записів з БД.
3. Оновлення записів (run workload update) – оновлення змісту заданої кількості записів з БД.
4. Вставка записів (run workload insert) – вставка заданої кількості записів у БД. Виконувалась останньою для того щоб унеможливити вплив розміру БД на результати тестувань інших операцій.

У результаті виконання сценарію над кожною з СУБД було отримано файли (додаток Б), які містили дані, що до наступних характеристик швидкодії: часу виконання навантаження (мілісекунд), пропускної здатності (операцій/секунду); середньої, мінімальної та максимальної затримки виконання операції (мікросекунд); а також кодів повернення (індикатор успішності виконання операції). На основі отриманих даних далі було проведено порівняння швидкодії СУБД для різного розміру бази даних та типу операції, побудовані діаграми.

6.3 Результати тестування

В результаті виконання навантажень над кожним екземпляром систем управління базами даних, було зібрано статистику (таблиця 6.1) що до пропускної здатності СУБД.

Таблиця 6.1

Пропускна здатність (оп/сек)	**Тип операції**		
Тип СУБД	READ	UPDATE	INSERT
PostgreSQL	1505	562	547
Redis	1862	1937	995
Memcached	1246	1285	1221
MongoDB	777	780	795
Aerospike	1745	1814	1754

Також було зібрано статистику (таблиця 6.2) що до показників середньої затримки виконання.

Таблиця 6.2

Середня затримка виконання (мкс)	**Тип операції**		
Тип СУБД	READ	UPDATE	INSERT
PostgreSQL	515	1191	1218
Redis	492	472	955
Memcached	685	660	701
MongoDB	912	909	876
Aerospike	514	491	507

Далі, інформацію було розділено на три групи за призначенням: реляційні БД, репозиторії “ключ-значення” та документо-орієнтовні бази даних. У групу реляційних БД увійшла PostgreSQL (яка надалі буде використовуватись як еталон), у групу репозиторіїв “ключ-значення” - Redis та Memcached, до документо-орієнтовних баз даних – MongoDB та Aerospike.

6.3.1 Порівняння репозиторіїв “ключ-значення”

По отриманим даним (рисунок 12), Redis має значно вищу пропускну здатність у сценаріях з виконанням операцій READ та UPDATE, але у випадку з INSERT, він незначно поступається Memcached.

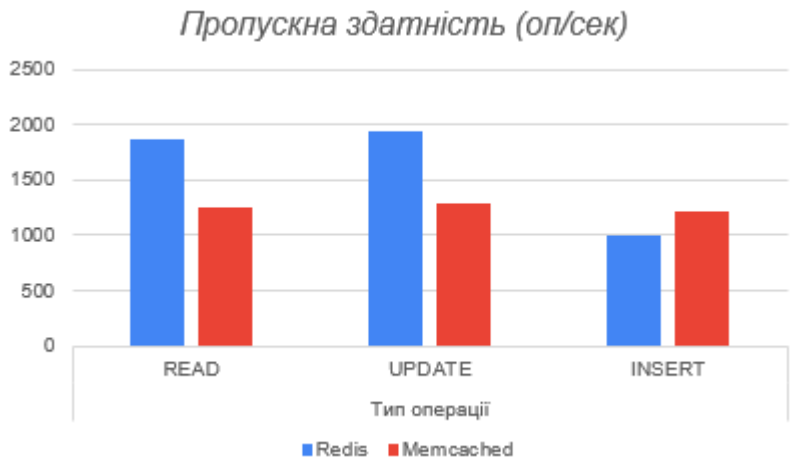

Рисунок 12

Згідно статистики (рисунок 13), Redis має значно нижчу середню затримку виконання у сценаріях з виконанням операцій READ та UPDATE, але у випадку з INSERT, він має затримку, що вища ніж у Memcached.

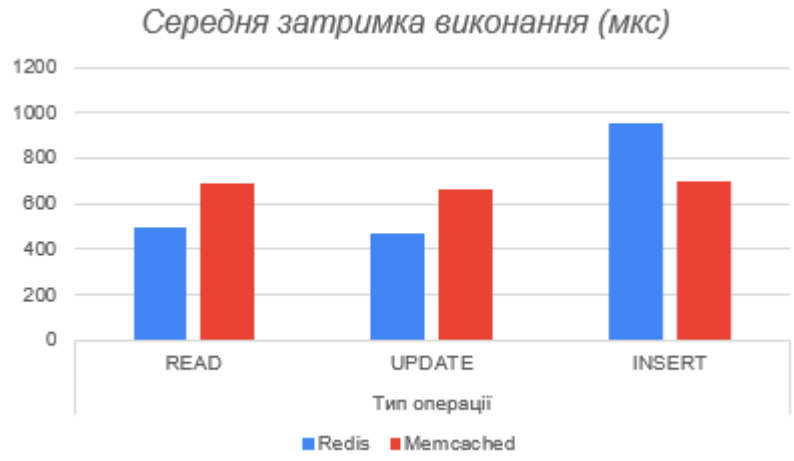

Рисунок 13

Отже, з огляду на те, що Redis в більшості випадків швидший за Memcached, то в групі репозиторіїв “ключ-значення” Redis є лідером.

6.3.2 Порівняння документо-орієнтовних БД

Згідно отриманих даних (рисунок 14), у сценаріях за всіма типами операцій над даними MongoDB має значно меншу пропускну здатність.

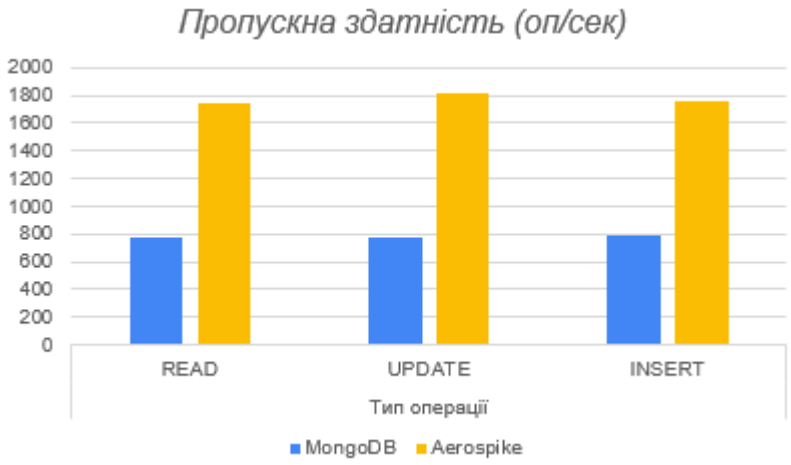

Рисунок 14

Статистика (рисунок 15) каже про те, що Aerospike у всіх випадках має майже в два рази меншу затримку виконання.

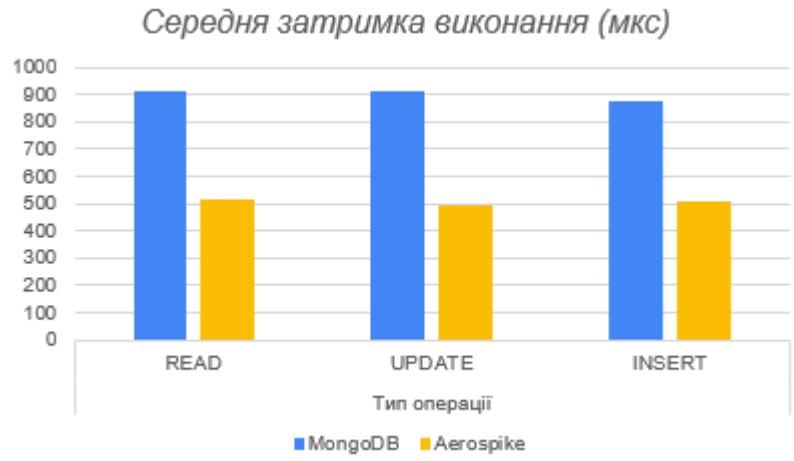

Рисунок 15

Отже, Aerospike є беззаперечним лідером у групі документо-орієнтовних БД; схоже, що це досягається завдяки використанню їм моделі даних “ключ-значення”.

6.3.3 Загальне порівняння

Для того, щоб визначити яка з обраних СУБД найкраще виконала задачу, було проведено загальне порівняння їх пропускної здатності та затримок виконання операцій, отримано наступні результати.

У випадку з пропускною здатністю (рисунок 16) кандидатами на загальне лідерство є Redis та Aerospike, але Redis у сценарії INSERT показав, що має замалу пропускну здатність.

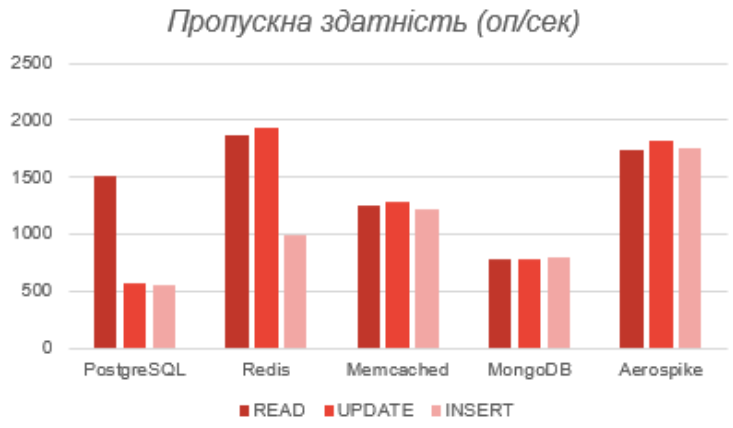

Рисунок 16

У випадку з середньою затримкою виконання (рисунок 17) у сценаріях READ, UPDATE Redis має показники схожі на відповідні у Aerospike, але Redis був повільніший під час вставки нових записів у базу даних.

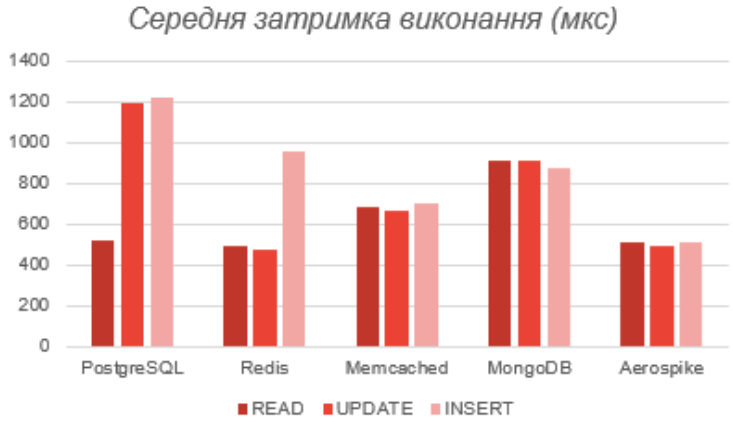

Рисунок 17

Отже, загальний лідером серед усіх розглянутих система управління базами даних є Aerospike, він найкраще виконав задачі по зчитуванню, оновленню і вставці JSON документів у базу даних.

ВИСНОВКИ

У цій роботі було виконано класифікацію систем управління базами даних, розглянуто особливості класів NoSQL та реляційних баз даних, визначено різницю між ними. Для цього було проаналізовано зміст наукових статей на цю тему.
Також було розглянуто існуючі види моделей транзакцій, а саме BASE та ACID, викладено їх особливості та зазначено класи СУБД, в яких ці моделі частіше всього застосовуються.

Додатково, у роботі було викладено теоретичні відомості про порівняння ефективності СУБД, порядок та особливості його виконання. І також представлено вичерпну інформацію що до об’єктів та засобів дослідження, а саме про такі СУБД: Redis, Memcached, MongoDB, Aerospike та PostgreSQL; програмний застосунок для виконання порівняння продуктивності СУБД – Yahoo! Cloud Benchmark Service. За допомогою нього проведено тестування навантаженням у випадку, коли обрані БД використовувались як репозиторії для JSON документів. На основі даних, отриманих під час тестування, зроблено висновки що до продуктивності СУБД у обраному сценарії.

Оцінка одержаних результатів. Представлені у роботі класи СУБД повністю покривають різноманіття існуючих на момент виконання роботи систем управління баз даних. Наведено вичерпну характеристику цих класів, детально вивчено різницю між ними. Детально описано моделі транзакцій, що використовуються у сучасних СУБД та принципи, які були у них покладені, наведено приклади, що розкривають ці принципи.

Докладно було розкрито поняття вимірювання продуктивності, цілі, заради яких його виконують, та типові помилки, яких слід уникати для отримання правильних результатів. Керуючись викладеними відомостями, було проведено тестування обраних СУБД та отримано актуальні відомості про їх швидкодію. А на основі зібраної статистики зроблено висновки про те як сучасні бази даних виконують поширену в наш час задачу – зберігання JSON документів.

Наукова та науково-технічна значущість роботи. У даній роботі укомплектовано наукові відомості про види СУБД, проаналізовано різницю між ними. На основі викладеної інформації, такі технічні спеціалісти, як архітектори програмних систем, зможуть приймати рішення що до вибору системи управління базами даних у проекті.

Доцільність продовження досліджень за відповідною тематикою. З постійним розвитком інформаційних систем змінюються потреби до систем управління базами даних, розробляються нові підходи до їх побудови. У майбутньому можуть виникнути нові класи баз даних, які треба буде визначити та розглянути їх особливості, порівняти з базами даних, що існували до цього. Тому дослідження з тематикою роботи треба проводити регулярно, щоб мати актуальне представлення про сферу систем управління баз даних.

СПИСОК ЛІТЕРАТУРИ

1. Gudivada, Venkat & Ramaswamy, Srini & Srinivasan, Seshadhri. Data Management Issues in Cyber-Physical Systems. – 2018.
2. Zhang, Chao & Lu, Jiaheng. Holistic evaluation in multi-model databases benchmarking. Distributed and Parallel Databases. – 2021.
3. Types Of NoSQL Databases – MongoDb. [Електронний ресурс] – Режим доступу до ресурсу: https://www.mongodb.com/scale/types-of-nosql-databases
4. Difference between SQL and NoSQL – GeeksforGeeks. [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/difference-between-sql-and-nosql/
5. The transactional model – IBM. [Електронний ресурс] – Режим доступу до ресурсу: https://www.ibm.com/docs/en/integration-bus/10.0?topic=transactions-transactional-model
6. GitHub репозиторій YCSB. [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/brianfrankcooper/YCSB
7. Документація та опис YCSB. [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/brianfrankcooper/YCSB/wiki
8. Офіційний сайт СУБД PostgreSQL. [Електронний ресурс] – Режим доступу до ресурсу: https://www.postgresql.org/
9. Офіційний сайт СУБД Redis. . [Електронний ресурс] – Режим доступу до ресурсу: https://redis.io/
10. Офіційний сайт СУБД Memcached. [Електронний ресурс] – Режим доступу до ресурсу: http://memcached.org/about
11. Офіційний сайт СУБД MongoDB. [Електронний ресурс] – Режим доступу до ресурсу: https://www.mongodb.com/what-is-mongodb
12. Офіційний сайт СУБД Aerospike. [Електронний ресурс] – Режим доступу до ресурсу: https://aerospike.com/products/database/

ДОДАТОК А

Приклад файлу з конфігурацією сценарію навантаження

```
# Copyright (c) 2010 Yahoo! Inc. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License"); you
# may not use this file except in compliance with the License. You
# may obtain a copy of the License at
#
# http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied. See the License for the specific language governing
# permissions and limitations under the License. See accompanying
# LICENSE file.


# Yahoo! Cloud System Benchmark
# Workload A: Update heavy workload
#   Application example: Session store recording recent actions
#
#   Read/update ratio: 50/50
#   Default data size: 1 KB records (10 fields, 100 bytes each, plus key)
#   Request distribution: zipfian

recordcount=1000
operationcount=1000
workload=site.ycsb.workloads.CoreWorkload

readallfields=true

readproportion=0.5
updateproportion=0.5
scanproportion=0
insertproportion=0

requestdistribution=zipfian
```

ДОДАТОК Б

Приклад вихідного файлу в результаті виконання сценарію навантаження

```
[OVERALL], RunTime(ms), 537
[OVERALL], Throughput(ops/sec), 1862.1973929236499
[TOTAL_GCS_G1_Young_Generation], Count, 0
[TOTAL_GC_TIME_G1_Young_Generation], Time(ms), 0
[TOTAL_GC_TIME_%_G1_Young_Generation], Time(%), 0.0
[TOTAL_GCS_G1_Concurrent_GC], Count, 0
[TOTAL_GC_TIME_G1_Concurrent_GC], Time(ms), 0
[TOTAL_GC_TIME_%_G1_Concurrent_GC], Time(%), 0.0
[TOTAL_GCS_G1_Old_Generation], Count, 0
[TOTAL_GC_TIME_G1_Old_Generation], Time(ms), 0
[TOTAL_GC_TIME_%_G1_Old_Generation], Time(%), 0.0
[TOTAL_GCs], Count, 0
[TOTAL_GC_TIME], Time(ms), 0
[TOTAL_GC_TIME_%], Time(%), 0.0
[READ], Operations, 1000
[READ], AverageLatency(us), 492.64
[READ], MinLatency(us), 357
[READ], MaxLatency(us), 10911
[READ], 95thPercentileLatency(us), 719
[READ], 99thPercentileLatency(us), 918
[READ], Return=OK, 1000
[CLEANUP], Operations, 1
[CLEANUP], AverageLatency(us), 872.0
[CLEANUP], MinLatency(us), 872
[CLEANUP], MaxLatency(us), 872
[CLEANUP], 95thPercentileLatency(us), 872
[CLEANUP], 99thPercentileLatency(us), 872
```

ДОДАТОК В

Приклад реалізації класу клієнта для бази даних

```java
/**
 * Copyright (c) 2012 YCSB contributors. All rights reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License"); you
 * may not use this file except in compliance with the License. You
 * may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
 * implied. See the License for the specific language governing
 * permissions and limitations under the License. See accompanying
 * LICENSE file.
 */

/**
 * Redis client binding for YCSB.
 *
 * All YCSB records are mapped to a Redis *hash field*.  For scanning
 * operations, all keys are saved (by an arbitrary hash) in a sorted set.
 */

package site.ycsb.db;

import site.ycsb.ByteIterator;
import site.ycsb.DB;
import site.ycsb.DBException;
import site.ycsb.Status;
import site.ycsb.StringByteIterator;
import redis.clients.jedis.BasicCommands;
import redis.clients.jedis.HostAndPort;
import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisCluster;
import redis.clients.jedis.JedisCommands;
import redis.clients.jedis.Protocol;

import java.io.Closeable;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import java.util.HashSet;
import java.util.Iterator;
import java.util.List;
import java.util.Properties;
import java.util.Set;
import java.util.Vector;

/**
 * YCSB binding for <a href="http://redis.io/">Redis</a>.
 *
 * See {@code redis/README.md} for details.
 */
public class RedisClient extends DB {

  private JedisCommands jedis;

  public static final String HOST_PROPERTY = "redis.host";
  public static final String PORT_PROPERTY = "redis.port";
  public static final String PASSWORD_PROPERTY = "redis.password";
  public static final String CLUSTER_PROPERTY = "redis.cluster";
  public static final String TIMEOUT_PROPERTY = "redis.timeout";

  public static final String INDEX_KEY = "_indices";

  public void init() throws DBException {
    Properties props = getProperties();
    int port;

    String portString = props.getProperty(PORT_PROPERTY);
    if (portString != null) {
      port = Integer.parseInt(portString);
    } else {
      port = Protocol.DEFAULT_PORT;
    }
    String host = props.getProperty(HOST_PROPERTY);

    boolean clusterEnabled = Boolean.parseBoolean(props.getProperty(CLUSTER_PROPERTY));
    if (clusterEnabled) {
      Set<HostAndPort> jedisClusterNodes = new HashSet<>();
      jedisClusterNodes.add(new HostAndPort(host, port));
      jedis = new JedisCluster(jedisClusterNodes);
    } else {
      String redisTimeout = props.getProperty(TIMEOUT_PROPERTY);
      if (redisTimeout != null){
        jedis = new Jedis(host, port, Integer.parseInt(redisTimeout));
      } else {
        jedis = new Jedis(host, port);
      }
      ((Jedis) jedis).connect();
    }

    String password = props.getProperty(PASSWORD_PROPERTY);
    if (password != null) {
      ((BasicCommands) jedis).auth(password);
    }
  }

  public void cleanup() throws DBException {
    try {
      ((Closeable) jedis).close();
    } catch (IOException e) {
      throw new DBException("Closing connection failed.");
    }
  }

  /*
   * Calculate a hash for a key to store it in an index. The actual return value
   * of this function is not interesting -- it primarily needs to be fast and
   * scattered along the whole space of doubles. In a real world scenario one
   * would probably use the ASCII values of the keys.
   */
  private double hash(String key) {
    return key.hashCode();
  }

  // XXX jedis.select(int index) to switch to `table`

  @Override
  public Status read(String table, String key, Set<String> fields,
      Map<String, ByteIterator> result) {
    if (fields == null) {
      StringByteIterator.putAllAsByteIterators(result, jedis.hgetAll(key));
    } else {
      String[] fieldArray =
          (String[]) fields.toArray(new String[fields.size()]);
      List<String> values = jedis.hmget(key, fieldArray);

      Iterator<String> fieldIterator = fields.iterator();
      Iterator<String> valueIterator = values.iterator();

      while (fieldIterator.hasNext() && valueIterator.hasNext()) {
        result.put(fieldIterator.next(),
            new StringByteIterator(valueIterator.next()));
      }
      assert !fieldIterator.hasNext() && !valueIterator.hasNext();
    }
    return result.isEmpty() ? Status.ERROR : Status.OK;
  }

  @Override
  public Status insert(String table, String key,
      Map<String, ByteIterator> values) {
    if (jedis.hmset(key, StringByteIterator.getStringMap(values))
        .equals("OK")) {
      jedis.zadd(INDEX_KEY, hash(key), key);
      return Status.OK;
    }
    return Status.ERROR;
  }

  @Override
  public Status delete(String table, String key) {
    return jedis.del(key) == 0 && jedis.zrem(INDEX_KEY, key) == 0 ? Status.ERROR
        : Status.OK;
  }

  @Override
  public Status update(String table, String key,
      Map<String, ByteIterator> values) {
    return jedis.hmset(key, StringByteIterator.getStringMap(values))
        .equals("OK") ? Status.OK : Status.ERROR;
  }

  @Override
  public Status scan(String table, String startkey, int recordcount,
      Set<String> fields, Vector<HashMap<String, ByteIterator>> result) {
    Set<String> keys = jedis.zrangeByScore(INDEX_KEY, hash(startkey),
        Double.POSITIVE_INFINITY, 0, recordcount);

    HashMap<String, ByteIterator> values;
    for (String key : keys) {
      values = new HashMap<String, ByteIterator>();
      read(table, key, fields, values);
      result.add(values);
    }

    return Status.OK;
  }

}
```