Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп’ютерних наук
(повна назва)

Кафедра Програмної інженерії
(повна назва)

# КВАЛІФІКАЦІЙНА РОБОТА

## Пояснювальна записка

рівень вищої освіти другий (магістерський)

**Дослідження методів реалізації ACID транзакцій для розподілених баз даних за технологією реплікування**

**(тема)**

Виконав:
Студент 2 курсу, групи ІПЗм-19-2
Набока А.О.
(прізвище,ініціали)

Спеціальність 121- Інженерія програмного забезпечення
(код і повна назва спеціальності)

Тип програми Освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Керівник доц. Мазурова О.О.
(посада,прізвище)

Допускається до захисту
Зав. кафедри ______________ З.В. Дударь
(підпис) (прізвище, ініціали)

2021р.

Харківський національний університет радіоелектроніки

Факультет Комп’ютерних наук
(повна назва)

Кафедра Програмної інженерії
(повна назва)

Рівень вищої освіти другий (магістерський)

Спеціальність 121- Інженерія програмного забезпечення
(код і повна назва спеціальності)

Тип програми Освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Інженерія програмного забезпечення
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ________________
(підпис)
« 26 » березня 2021 р.

**ЗАВДАННЯ**
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студента Набоки Артема Олександровича
(прізвище, ім’я, по батькові)

1. Тема роботи Дослідження методів реалізації ACID транзакцій для розподілених баз даних за технологією реплікування
затверджена наказом університету від 26.03.2021 № 385
2. Термін подання роботи до екзаменаційної комісії 10 травня 2021р.
3. Вихідні дані до роботи *електронні ресурси за обраною тематикою, вимоги до реалізації розподілених транзакцій за принципом ACID, бази даних MongoDB, VoltDB, SQL Server, середовища NoSQL Manager, SQL Server Management Studio, Visual Studio 2019, мови VoltSQL, T-SQL, C#.*
4. Перелік питань, що потрібно опрацювати в роботі *аналіз проблемної області і постановка задачі, розгляд методів реалізації ACID транзакцій, розробка структур баз даних для досліду, проектування запитів і транзакцій, розробка програмного забезпечення для отримання замірів, проведення експерименту, порівняння обраних методів, формування рекомендацій.*
5. Перелік графічного матеріалу із зазначенням креслеників, схем, слайдів, ілюстрацій *актуальність проблеми, методи масштабування, постановка задачі, аналіз проблемної області, планування експерименту, проектування БД, архітектура і реалізація програмного забезпечення, результати, рекомендації, висновки.*

6. Консультанти розділів роботи

| Найменування розділу | Консультант (посада, прізвище, ім’я, по батькові) | Позначка консультанта про виконання розділу | |
|---|---|---|---|
| | | підпис | дата |
| Спецчастина | доц. Мазурова О.О. | | 08.05.21 |
| | | | |

**КАЛЕНДАРНИЙ ПЛАН**

| № | Назва етапів роботи | Терміни виконання етапів роботи | Примітка |
|---|---|---|---|
| 1 | Аналіз проблемної області дослідження | 25.01.21 – 8.02.21 | виконано |
| 2 | Аналіз аналогів | 5.02.21 – 9.02.21 | виконано |
| 3 | Розробка постановки задачі | 9.02.21 – 16.02.21 | виконано |
| 4 | Дослідження існуючих методів реалізації розподілених транзакцій | 16.02.21 – 20.02.21 | виконано |
| 5 | Планування експериментального дослідження | 20.02.21 – 23.02.21 | виконано |
| 6 | Розробка структур баз даних | 23.02.21 – 01.03.21 | виконано |
| 7 | Проектування ПЗ для експерименту | 01.03.21 – 15.03.21 | виконано |
| 8 | Розробка запитів та транзакцій | 15.03.21 – 02.04.21 | виконано |
| 9 | Проведення експерименту та аналіз результатів | 02.04.21 – 16.04.21 | виконано |
| 10 | Оформлення статті | 16.04.21 – 20.04.21 | виконано |
| 11 | Підготовка пояснювальної записки | 02.04.21 – 30.04.21 | виконано |
| 12 | Підготовка презентації та доповіді | 30.04.21 – 03.05.21 | виконано |
| 13 | Нормоконтроль | 03.05.21 – 11.05.21 | виконано |
| 14 | Рецензування | 03.05.21 – 11.05.21 | виконано |
| 15 | Занесення диплома в електронний архів | 08.05.21 | виконано |
| 16 | Попередній захист | 08.05.21 | виконано |
| 17 | Допуск до захисту у зав. кафедри | 12.05.21 | виконано |

Дата видачі завдання 25 січня 2021р.

Студент ____________________

(підпис)

Керівник роботи ____________ доц. Мазурова О.О.

(підпис) (посада, прізвище, ініціали)

# РЕФЕРАТ/ABSTRACT

Кваліфікаційна робота магістра містить: 155 с., 36 рис., 56 табл., 30 джер.

МЕТРИКА ЕФЕКТИВНОСТІ, НАВАНТАЖЕННЯ БАЗИ, РОЗПОДІЛЕНА БАЗА ДАНИХ, СУБД, ТРАНЗАКЦІЯ, ACID, BSON, C#, MONGODB, NEWSQL, NOSQL, SQL, SQL SERVER, T-SQL, TWO-PHASE COMMIT, VOLTDB.

Метою роботи є дослідження існуючих методів реалізації ACID транзакцій для розподілених за допомогою технології реплікації баз даних, а також порівняння ефективності цих методів для різного роду запитів та транзакцій.

Методи розробки та проектування базуються на платформі .Net Core 3.1, мові TypeScript та фреймворку Angular 10, СУБД MS SQL Server 2017, MongoDB 4.4, VoltDB 6.1, середовищі розробки Visual Studio 2019 та Visual Studio Code.

У результаті роботи були обрані методи реалізації розподілених транзакцій, розроблена база даних для кожного з них та складені висновки щодо ефективності виконання запитів для кожного з методів.

ACID, BSON, C#, DATABASE LOAD, DBMS, DISTRIBUTED DATABASE, EFFICIENCY METRICS, MONGODB, NEWSQL, NOSQL, SQL, SQL SERVER, T-SQL, TRANSACTIONS, TWO-PHASE COMMIT, VOLTDB.

The purpose of the work is to investigate the existing methods of implementing ACID transactions for distributed using replication technology databases, as well as to compare the efficiency of these methods for different queries and transactions.

Development and design methods are based on the .Net Core 3.1 platform, TypeScript language and Angular 10 framework, DBMS MS SQL Server 2017, MongoDB 4.4, VoltDB 6.1, Visual Studio 2019 IDE and Visual Studio Code.

As a result, the methods of implementation of distributed transactions were selected, a database was developed for each of them, and conclusions were made on the efficiency of query execution for each of the methods.

Я, Набока Артем Олександрович, студент гр. ІПЗм-19-2, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Дослідження методів реалізації ACID транзакцій для розподілених баз даних за технологією реплікування», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElAr KhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

# ЗМІСТ

Перелік умовних скорочень ........ 8
Вступ ........ 9
1 Аналіз проблемної області та постановка задачі ........ 11
1.1 Аналіз проблемної області дослідження ........ 11
1.2 Аналіз існуючих аналогів ........ 15
1.3 Постановка задачі ........ 16
2 Опис прийнятих проектних рішень ........ 19
2.1 Аналіз методів реалізації розподілених транзакцій ........ 19
2.2 Математичний опис проблемної області дослідження ........ 25
2.3 Планування експериментального дослідження ........ 27
2.3.1 Розробка структур баз даних для експерименту ........ 27
2.3.2 Обмеження експерименту ........ 32
2.3.3 Моделювання предметної області та бізнес-процесів ........ 36
2.4 Проектування програмного забезпечення для експерименту ........ 37
3 Опис програмної реалізації ........ 43
3.1 Розробка фізичної моделі реляційної БД ........ 43
3.2 Розробка транзакцій ........ 44
3.3 Опис програмного інтерфейсу ........ 55
4 Експериментальне дослідження методів підтримки транзакцій ........ 57
4.1 Дослідження для MongoDB ........ 57
4.2 Дослідження для VoltDB ........ 67
4.3 Дослідження для Two-Phase Commit ........ 78

5 Аналіз результатів дослідження ........................................................................ 89

5.1 Порівняння продуктивності методів ........................................................... 89

5.2 Розробка рекомендацій щодо використання ................................................ 110

Висновки ................................................................................................................ 116

Перелік джерел посилань ...................................................................................... 118

Додаток А Перелік джерел посилання за науковими напрямами керівника та науковців кафедри програмної інженерії ........................................... 121

Додаток Б Звіт результатів Перевірки кваліфікаційної роботи на унікальність тексту ........................................................................................................... 122

Додаток В Слайди презентації .............................................................................. 123

Додаток Г Апробація результатів роботи ........................................................... 141

Додаток Д Експертний Висновок результатів Перевірки кваліфікаційної роботи на відповідність оформлення Вимоги ДСТУ 3008: 2015 ........................ 156

# ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

ACID – atomicity, consistency, isolation, durability;

BSON – binary javascript object notation;

IP – internet protocol;

JSON – javascript object notation;

TCP – transmission control protocol.

# ВСТУП

Бази даних стали невід’ємною частиною сучасних додатків, а в деяких – центральним елементом. На даний момент бази даних використовуються у переважній більшості предметних областей програмного забезпечення: електронна комерція, фінансовий сектор, охорона здоров’я, соціальні мережі, системи відео та аудіо-хостингу і т.д. Вони використовуються в невеликих специфічних додатках, так і в додатках з мільйонами користувачів. Разом зі збільшення популярності баз даних, збільшувався і об’єм самих даних, що потрібно зберігати. Наприклад, банки та інтернет-магазини зберігають мільйони записів про дії їх користувачів та цих даних з кожним днем стає набагато більше. Саме тому постає проблема масштабування великих баз даних задля забезпечення адекватної швидкості відповіді додатку.

Існує два основних методи масштабування баз даних – горизонтальне, що передбачає розбиття всієї бази даних на кілька об’єднаних мережею фізичних серверів, які зберігають частину свою частину даних, та вертикальне, що включає в себе збільшення ресурсів єдиного сервера баз даних. Але через надто значне збільшення інформації останній метод перестає бути ефективним, оскільки потребував би недосяжних ресурсів. Саме тому у більшості випадків використовується саме горизонтальне масштабування.

У той же час більшість сучасних додатків потребують від баз даних підтримки транзакційності за принципом ACID: переказ коштів, збереження замовлення і т.д. Однак, через те, що під час горизонтального масштабування дані можуть знаходитися на різних фізичних серверах, неможливо забезпечити транзакційне збереження даних класичними методами, бо мережеві протоколи самі по собі не підтримують розподілену транзакційність. Найбільш складно реалізувати атомарність транзакції, а також узгодженість всіх серверів. Саме тому в даній роботі розглянуті різні способи забезпечення транзакційності за принципом ACID для горизонтально масштабованих баз даних.

Під час виконання магістреської кваліфікаційної роботи був проведений аналіз проблемної області реалізації розподілених транзакцій на основі публікацій світових та вітчизняних фахівців в проблемній області (див. додаток А), розглянуті основні існуючі методи реалізації розподілених транзакцій, а також обрані найбільш актуальні методи для проведення дослідження. У результаті була сформована постановка задачі, а також спроектовано весь процес проведення експерименту дослідження, включаючи проектування схем та структур баз даних, запитів до цих баз. Крім того, були описані умови та обмеження проведення експерименту, сформовані та обґрунтовані метрики, за якими порівнювалися обрані методи здійснення транзакцій у розподілених базах даних.

Робота прошла успішну перевірку на академічну доброчесність (див. додаток Б).

На основі плану проведення дослідження та його результатів було розроблено презентацію (див. додаток В). Також за результатами роботи були написані тези доповіді на участі у науково-технічній конференції «Інноваційні технології-2020, а також до науково-технічного журналу «АСУ та прибори автоматики» подано до опублікування статтю «Дослідження методів реалізації розподілених ACID транзакцій за технологією реплікації» (див. додаток Г).

Також робота відповідає всім критеріям нормоконтролю (див. додаток Д).

# 1 АНАЛІЗ ПРОБЛЕМНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

## 1.1 Аналіз проблемної області дослідження

Через збільшення кількості інформації, що необхідно зберігати, вертикальне масштабування серверів баз даних перестає бути ефективним, оскільки збереження на одному сервері терабайтів даних та забезпечення високої доступності цих даних потребувало би недосяжних ресурсів. Саме тому великої популярності набули розподілені бази даних, суть яких полягає у горизонтальному масштабуванні ресурсів.

Архітектура розподілених баз даних передбачає набір серверів-вузлів, об'єднаних в кластери, на яких зберігається певна БД, а також певну систему, що займається маршрутизацією запитів до вузлів (див. рис. 1.1). Система маршрутизації може бути як частиною СУБД, так і окремим додатком.

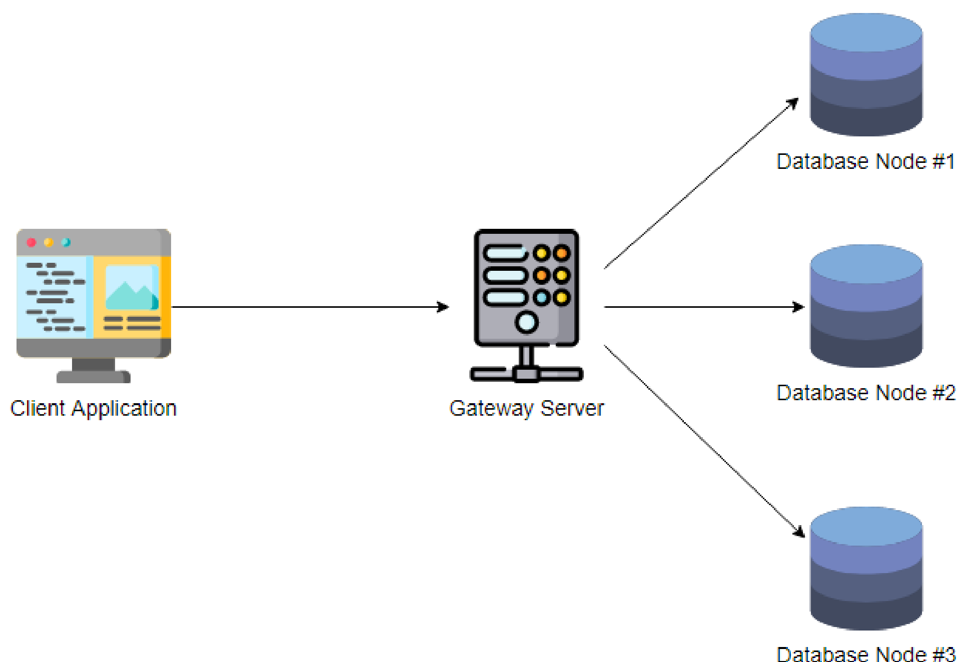


Рисунок 1.1 – Ілюстрація взаємодії додатку з розподіленою базою даних

Кожен вузол може зберігати або власну певну частину вертикально або горизонтально поділених даних для зменшення навантаження на конкретний

сервер – такий вузол називається шардом, або може зберігати копію загальних даних для забезпечення швидкого отримання даних при великому навантаженні – такий тип вузлу називається реплікою [1]. Також два зазначених методи можуть комбінуватися.

Разом із горизонтальним масштабуванням постає проблема реалізації у розподілених базах транзакцій, які б відповідали властивостям ACID [2]. ACID транзакції життєво необхідні для додатків, де частково зафіксовані результати призведуть систему в некоректний стану: прикладами є фінансові додатки, додатки з продажу та бронювання ресурсів. Дана задача є нетривіальною, адже вузли кластеру бази даних зазвичай знаходяться на різних фізичних серверах, з’єднання до яких здійснюється за допомогою TCP/IP протоколу, що сам по собі не має ніяких інструментів для реалізації транзакційності.

Причиною створення розподіленої бази даних у більшості випадках є забезпечення високої доступності до даних додатку, що означає необхідність швидко отримувати та обробляти дані. В свою чергу найбільш популярними рішеннями, що реалізують ACID транзакції, у свій час були класичні реляційні СУБД: SQL Server, Oracle, MySQL, Postgres [3]. Але під час підвищення навантаження на СУБД та збільшення збережених даних класичні реляційні бази даних масштабуються вертикально та не мають вбудованих механізмів до горизонтального масштабування на шарди та репліки [4]. А тому класичні реляційні СУБД здебільшого не можуть надати високу доступність до великих даних під час роботи великої кількості користувачів.

З іншого боку з необхідністю розгортування розподілених баз даних стали популярні NoSQL СУБД [5]. За своєю натурою NoSQL бази даних зазвичай працюють швидше під час запиту і аналізу даних та краще масштабуються, ніж класичні SQL бази даних, бо вони в першу чергу не підтримуються реляційну схему даних – немає структурованих таблиць, специфічної схеми даних, немає фізичних зв’язків між сутностями, немає необхідності підтримувати цілісність посилань зовнішніх ключів, – а також NoSQL бази даних підтримують транзакційність лише на рівні одного запису (тому немає можливості проводити

транзакції над даними, що знаходяться на різних вузлах) [6]. Саме тому NoSQL бази даних використовуються у пошукових та аналітичних системах [7], але не можуть використовуватися в додатках, де транзакційність критично важлива – переводи коштів, бронювання місць, створення замовлень і т.д.

Проаналізувавши все вищеописане, можна більш широко сформулювати проблему, що досліджується, яка полягає в реалізації системи з розподіленою базою даних, що надавала би високу доступність та найбільш швидкий час виконання запитів при роботі з великими даними, а також підтримувала транзакційність за принципом ACID для критичних для програмної системи бізнес-операцій.

Для цієї реалізації, очевидно, потрібно виявити кілька методів, що будуть досліджуватися. При цьому ці методи повинні бути досить популярними та щоб їх можна було легко застосувати для розробки сучасних додатків. Звісно, що ці методи в першу чергу всі повинні підтримувати ACID принцип, тобто, дані транзакції повинні одночасно зафіксуватися на всіх вузлах кластера, при чому повинні бути зафіксовані атомарно. Це означає, якщо певний запит завершився помилкою на одному з вузлів, то вся транзакція повинна відкотитися до початкового стану. Консистентність кластеру полягатиме у тому, що будь-який запит, який виконується після успішної фіксації транзакції, повинен мати доступ до раніше зафіксованих у цій транзакції даних, звертаючись до будь-якого вузлу кластера [8]. За допомогою цієї консистеності досягається висока доступність даних та відсутність аномалій при виконанні певного роду бізнес-операцій: наприклад, бронювання декількома клієнтами одного і того ж номеру в готелі, придбання того ж самого товару або списання коштів з рахунку. Звісно, що також важливими аспектами є ізольованість транзакцій та їх довговічність, що означає їх фіксування на постійний носій, таким чином що зафіксовані транзакцією дані будуть доступні після перезавантаження серверу бази.

Але обраний набір найбільш актуальних методів реалізації розподілених транзакцій сам по собі не несе цінності для процесу проектування та розробки програмного забезпечення. Набагато важливіше знати, як ці методи

відрізняються між собою, які сильні та слабкі сторони в плані ефективності використання має той чи інший метод. А в свою чергу ефективність методів у загальному випадку полягає в їх продуктивності виконання різноманітних запитів, а також ресурсів, необхідних їм для цього виконання. Саме ці знання дадуть можливість правильно підібрати модель виконання транзакцій у розподілених базах даних. Наприклад, для системи трасування та формування статистики потрібен метод з найбільшою пропускною здатність виконання запитів вставки, адже ця операція є найбільш часто виконуваємою в таких системах. З іншого боку, якщо є дуже обмежений бюджет для оренди віртуальної машини серверу бази, то буде розглядатися підхід з найменшим споживанням, наприклад, оперативної пам'яті. Виходячи з усього вищесказаного стає очевидною необхідність експериментального порівняння різних методів.

Іншим аспектом даної області є метод розподілення бази даних. Як вже було загадано на початку, то основними з них є реплікація та шардінг, а також їх гібрид. Тут одразу варто зазначити, що більш у даній роботі акцент сконцентровано саме на технології реплікації. По-перше, тому що для шардингу зазвичай зв'язані між собою дані зберігаються на одному вузлі, а значить далеко не всі транзакції будуть дійсно розподіленими. По-друге, саме технологія реплікації забезпечує високу доступність даних, тобто якщо один сервер бази даних по певним причинам перестав бути доступним або не витримує навантаження, то вся система не падає, а запити перенаправляються на інший вузол [9]. Саме ця технологія життєво важлива для відомих гігантів, як Amazon або Ebay. Дуже часто за допомогою реплікації сервери бази даних розташовані в різних куточках планети та обслуговують найбільш близьких за геолокацією користувачів [10]. Тут і виникає та проблема, коли, наприклад, два користувачі з Північної Америки та Азії в один і той самий момент намагуються забронювати одне й те ж саме місце на борту літака. За відсутності ACID транзакційності може статися аномалія одночасного володіння квитка одразу двома особами, що в свою чергу призведе до довгого розгляду ситуації. Саме тому так важливо дослідити розподілену транзакційність для реплікованої розподіленої бази даних.

## 1.2 Аналіз існуючих аналогів

Існує декілька підходів для розподілених транзакцій. Більшість з них орієнтовані не на транзакційність безпосередньо бази даних, а транзакційність розподілених програмних додатків системи. Такі підходи дуже часто використовуються в мікросервісній архітектурі, де потрібно синхронізувати стан декількох мікросервісів. Але кожен мікросервіс може мати свою СУБД різного типу. Для мікросервісної системи існує навіть спеціальний патерн Saga [11], що дозволяє проводити розподілені транзакції та може бути реалізований двома методами:

- хореографія (Choreography) – коли кожен мікросервіс передає дані про успішне або помилкове завершення транзакції лише наступному та попередньому мікросервісу [12];
- оркестрація (Orchestration) – коли існує окремий мікросервіс, який керує фіксуванням транзакцій на кожному з інших мікросервісів. При цьому ці інші мікросервіси більше не кумунікують один з одним, а лише з «оркестратором».

Приклади інструментів для транзакційності розподілених транзакцій:

- Fescar від компанії Alibaba;
- Seata – open source рішення [13];
- OpenSleigh – рішення для підтримки Saga шаблону для .Net додатків.

Всі вищезазначені рішення є досить ефективними у своїй сфері, але вони напряму не стосуються як такової транзакційності на рівні бази даних. Безпосередньо для забезпечення транзакційності на рівні бази даних можуть використовуватися інші підходи, зокрема найбільш елементарний підхід Local Transaction. Його суть полягає у паралельному виконанні класичних транзакцій на кожному з вузлів кластера. Очевидно, що даний підхід не забезпечує глобальну ACID транзакційність, адже дані можуть зафіксуватися на різних

вузлах в різний час, що призведе до втрати узгодженості. А також у випадку помилки виконання запиту на будь-якому з вузлів, ці запити можуть успішно зафіксуватися на інших, що також суперечить правилу атомарності та узгодженості.

Іншим підходом є так званий Flexible Transaction підхід, що по своїй суті дуже схожий на Saga шаблон для мікросервісів, в ньому по черзі на кожному вузлі відкриваються (але не затверджуються) класичні транзакції та виконуються запити [14]. Лише після того, як запити були успішно виконані на всіх вузлах, транзакції починаються одна за одною затверджуватися на кожному вузлі. Очевидно, що даний підхід надає більш ефективні розподілені транзакції, ніж Local Transaction, але все одно при ньому транзакції залишаються відкритими досить довгий час, що призведе до зменшення ефективності при високому паралелізмі.

Розглянувши різні аналоги реалізації розподілених транзакцій можна оцінити їх недоліки та під час дослідження обрати методи, що нівелюють ці недоліки повність або хоча б більшу їх частину.

## 1.3 Постановка задачі

Проаналізувавши проблемну область та виявивши основні проблеми існуючих систем, можна скласти перелік задач для виконання дослідження. У першу чергу потрібно виявити найбільш підходящі для масового використання методи транзакційності для розподілених транзакцій та перевірити функціонал транзакцій на той, що відповідає принципам ACID. Після цього найбільш важливим фактором є швидкість обробки запитів та транзакцій за допомогою обраних методів, оптимальність використовуваних ресурсів. Ці фактори надзвичайно важливі, адже безпосередньо впливають на користувацькі

очікування щодо швидкості відповіді додатку, а також на вартість хостингу сервер.

Для досліду, звісно, необхідно буде створити кластер бази даних зі змінною кількістю реплік, які повинні бути в узгодженому стані після виконання транзакції. Ефективність підходу також буде залежати від кількості реплік у кластері.

Задля коректного порівняння підходів, для кожного з них база даних повинна містити одні й ті ж дані та відносини між сутностями, хоча її схема може відрізнятися від типу СУБД (SQL, NoSQL). В якості предметної області бази даних найкраще обрати реальну область використання, наприклад, електронна комерція, і для цього напрямку створити спрощену базу даних. Для цієї бази даних необхідно буде створити набір запитів різноманітного характеру, включаючи, звісно, запити в транзакції. На прикладі цих запитів буду робитися подальші заміри.

Дані заміри повинні бути зроблені з різною кількістю даних та паралельних користувачів, які в один і той же момент будуть звертатися до бази даних. Для більшої точності кожен запит або транзакція повинні бути заміряні декілька разів. Мірятися будуть наступні величини:

- час виконання запитів при змінній кількості користувачів, реплік та даних;
- дисковий простір для зберігання бази даних різного розміру;
- оперативну пам’ять, що потребує сервер бази даних;
- процесорний час, що витрачає сервер бази даних.

Після оцінки зазначених факторів та визначення оптимальності кожного підходу потрібно буде їх порівняти для того, щоб зробити висновки щодо доцільності використання того чи іншого методу для конкретного типу додатку. Висновки повинні будуватися на моделях залежності часу виконання запитів або кількості споживаних ресурсів від кількості даних в базі, кількості реплік у кластері, а також кількості одночасних користувачів.

Отже, під час дослідження необхідно вирішити наступні глобальні задачі:

– провести аналіз та обрати актуальні методи реалізації розподілених транзакцій, які б могли масово використовуватися в сучасних додатках;
– розробити математичний опис для дослідження, визначити критерії оцінювання ефективності методів;
– розробити план експериментального дослідження обраних методів (схему БД, транзакції, обмеження тощо);
– спроектувати та розробити програмне забезпечення для проведення дослідження;
– провести експериментальне дослідження методів та на базі його результатів розробити рекомендації щодо застосування методів підтримки транзакцій.

# 2 ОПИС ПРИЙНЯТИХ ПРОЕКТНИХ РІШЕНЬ

## 2.1 Аналіз методів реалізації розподілених транзакцій

При виникненні перших горизонтально масштабованих баз даних ще не було ніяких готових рішень для проведення розподілених транзакцій одразу з коробки СУБД. Тому найпершим запропонованим способом реалізації розподілених транзакцій стала реалізація транзакцій на рівні додатку, яка на той час базувалася на шаблоні Two-phase-commit [15]. В основі даного шаблону стоїть прошарок між додатком та базою даних TransactionManager, який реалізує розподілену транзакцію в два кроки: перевірка, чи може транзакція бути зафіксована для всіх вузлів кластеру, а потім у випадку позитивного результату безпосередньо ізольована фіксація транзакції на всіх вузлах (див. рис. 2.1).

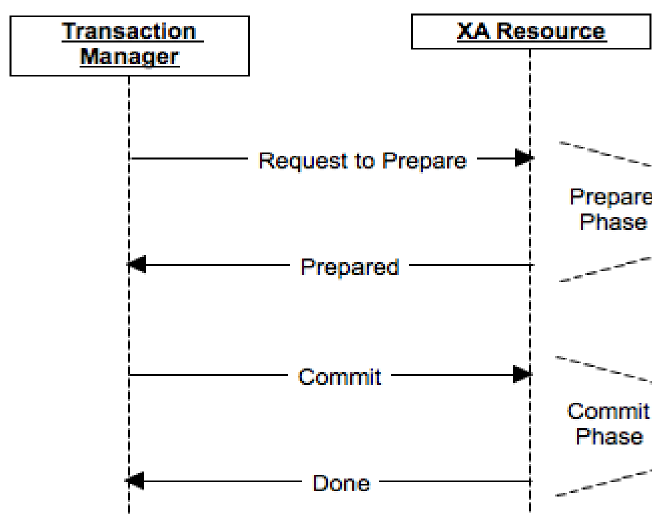


Рисунок 2.1 – Ілюстрація роботи шаблону Two-Phase-commit

Перевага даного методу у незалежності від типу бази даних та можливість повної кастомізації транзакційної системи під потреби додатку. Саме тому даний метод буде розглядатися у дослідженні та буде перевірена його ефективність при роботі з різною кількістю даних під різним навантаженням. В якості конкретної реалізації обрано реляційну СУБД SQL Server та мову програмування C#.

NoSQL СУБД спочатку не були призначені для виконання складних транзакцій, через що не могли бути повноцінно використані у додатках, де транзакційність критично важлива (грошові перекази, бронювання і т.д.). Розуміючи цю проблему розробники NoSQL СУБД стараються додати підтримку ACID транзакцій для своїх продуктів. Тому далі йде дослідження вбудованих можливостей щодо транзакцій у NoSQL базах даних. Але бази даних такого типу не мають спільної мови запиту або уніфікованої схеми зберігання даних. Саме тому необхідно за певними критеріями обрати одну базу даних, на якій буде проводитися дослідження.

Логічно, що обирати необхідно з найбільш популярних у використанні NoSQL базах даних, до яких входять наступні:

- MongoDB;
- Cassandra;
- HBase;
- Redis;
- Couchbase.

Для вибору NoSQL бази використовуватимуться наступні критерії:

- популярність бази – тобто ми повинні запевнитися, що дану базу дійсно використовують розробники в реальних проектах і тому є сенс досліджувати її. Для визначення популярності буде використовуватися db-engines ресурс, в якому вже виставлені оцінки кожній базі;
- принципи CAP – кількість принципів з теореми CAP, що підтримуються базою даних;
- наявність вбудованої системи транзакційності;
- вид ліцензії – чи є open source проектом, чи захищена певною ліцензією;
- пропускна здатність – максимальна кількість операцій читання та запису за одну секунду. Буде враховуватися для кластеру з двох вузлів.

Для обраних критеріїв були обрані наступні шкали:

а) популярність бази: оцінка з db-engines;

б) принципи CAP (C – Consistency, A – Availability, P – Partition Tolerance): зазвичай база підтримує не менше двох принципів даної теореми. Тому існують наступні варіанти підтримки:
   1) CAP – 5 балів;
   2) CP – 4 балів;
   3) AP – 3 балів;
   4) CA – 2 балів.

в) вбудована транзакційність:
   1) повне виконання ACID – 10 балів;
   2) лише атомарність для серії запитів – 5 балів;
   3) транзакційність лише на рівні запису – 1 бал.

г) тип ліцензії:
   1) open source – 3 бали;
   2) partial (частковий) open source – 2 бали;
   3) under private license – 1 бал.

д) пропускна здатність: кількість операцій за секунду.

Окремо необхідно сказати, що для шкали принципів CAP найбільш важливим принципом є P – Partition-Tolerance, адже саме цей принцип забезпечує можливість масштабування бази даних, її розподіленості. Другим за важливістю є принцип C – Consistency, який забезпечує транзакційність даних. Принцип A – Availability гарантує, що кожен запит до бази даних буде супроводжуватися відповіддю сервера бази даних незалежно від того, чи був цей запит успішним або закінчився помилкою.

Для вбудованої транзакційності розрив у балах зумовлений складністю реалізації того чи іншого рішення.

Для шкали типу ліцензії перевага надається відкритим проектам, що можуть бути використані будь-ким.

Таким чином після отримання необхідних значень кожного критерія можна побудувати таблицю NoSQL баз даних зі значеннями відповідних критеріїв (див. табл. 1).

Таблиця 1 – NoSQL бази даних зі значеннями критеріїв

| | Популяр-ність | Принципи CAP | Вбудована транзакційність | Ліцензія | Пропускна здатність |
|---|---|---|---|---|---|
| MongoDB | 457.73 | CP | ACID | Partial Open Source | 13,462.51 |
| Cassandra | 118.84 | CP | На рівні запису | Open Source | 31,144.24 |
| HBase | 46.92 | AP | На рівні запису | Open Source | 23,373.93 |
| Redis | 153.63 | AP | Атомарність для серії запитів | Open Source | 74,239.05 |
| Couchbase | 31.82 | CP | ACID | Open Source | 26,140.82 |

Вирішувати дану задачу вибору NoSQL бази будемо за допомогою лінійної адитивної згортки з ваговими коефіцієнтами, що активно використовується для вирішення задач вибору в різних областях [16], за формулою:

$$Z^* = max \sum_{j=1}^{n} \alpha_j \, \beta_j a_{ij}$$

де $\alpha_j$ – нормуючі множники,

$\beta_j$ – вагові коефіцієнти.

Для початку потрібно нормувати всі значення критеріїв, тобто привести їх із різних шкал до однієї в діапазоні [0; 1] з урахуванням min і max:

$$f = \frac{f_{\text{вимір}} - f_{min}}{f_{max} - f_{min}}$$

Далі потрібно визначити вагові коефіцієнти. Було прийнято рішення робити це за допомогою пропорційного методу. Як вже було згадано, то дуже важливо, щоб базу даних використовувало якомога більше людей, а оскільки ми досліджуємо в першу чергу транзакційність, то також дуже важливим є максимальна функціональність вбудованої транзакційності. Саме тому вважаємо, що популярність бази та вбудована транзакційність в 4 рази важливіші, ніж ліцензія та пропускна здатність, та в 2 рази важливіші, ніж слідування CAP принципам (адже вони все ж досить важливі з точки зору архітектури БД).

Тому в кінці отримуємо наступні вагові коефіцієнти:

- популярність: 0.33.
- принципи CAP: 0.17.
- вбудована транзакційність: 0.33.
- ліцензія: 0.08.
- пропускна здатність: 0.08.

Тепер можна побудувати таблицю з урахування вагових коефіцієнтів та нормуючих множників (див. табл. 2).

Таблиця 2 – NoSQL бази даних зі значеннями критеріїв у числовому форматі

| | Популяр-ність | Принципи CAP | Вбудована транзакційність | Ліцензія | Пропускна здатність | $Z^*$ |
|---|---|---|---|---|---|---|
| MongoDB | 457.73 | 4 | 10 | 2 | 13,462.51 | 0.83 |
| Cassandra | 118.84 | 4 | 1 | 3 | 31,144.24 | 0.34 |
| HBase | 46.92 | 3 | 1 | 3 | 23,373.93 | 0.11 |
| Redis | 153.63 | 3 | 5 | 3 | 74,239.05 | 0.4 |
| Couchbase | 31.82 | 4 | 10 | 3 | 26,140.82 | 0.6 |
| Вагові коеф. | 0.33 | 0.17 | 0.33 | 0.08 | 0.08 | |

В останньому стовпці відображено результати функції згортки. Максимальне значення 0.83 стоїть біля бази даних MongoDB, а тому можна зробити висновок, що вона буде найкращою альтернативою для дослідження транзакційності в розподілених NoSQL базах даних [17].

Як уже було пригадано, то класичні реляційні СУБД не призначені для горизонтального масштабування. Для подолання цієї проблеми були винайдені NoSQL бази даних, основною задачею яких була якраз гнучкість та масштабованість. А відносно недавно був розроблений новий тип баз даних NewSQL, основною метою якого було збереження реляційної моделі даних, але в той же час надати гнучке масштабування властиве NoSQL СУБД [18]. Тому при дослідженні реалізації розподілених транзакцій обов'язково повинен бути розглянути даний тип СУБД. Для досліду також необхідно обрати певну NewSQL базу даних. Хоча вибір NewSQL є дуже вузьким, але двома найбільш популярними базами даних даного типу є Clustrix та VoltDB. Ці бази даних здебільшого мають однаковий функціонал, особливо в плані транзакцій та масштабування, а відрізняються лише внутрішньою реалізацією. Для досліду буде використана VoltDB база даних лише через більшу її популярність та детальну документацію, а також інтеграцію з великою кількістю мов програмування.

Отже, для проведення дослідження обрані наступні методи:

– дослідження реалізації розподілених транзакцій за допомогою шаблону Two-Phase Commit;
– дослідження та використання вбудованих можливостей розподіленої транзакційності в NoSQL базах даних на прикладі MongoDB;
– дослідження та використання NewSQL баз даних на прикладі VoltDB.

Як вже було згадано, то всі три підходи будуть у першу чергу перевірятися на відповідність принципам ACID, де окрема увага буде приділятися першим двом принципам атомарності та узгодженості (дані, змінені цією транзакцією, можуть бути одразу запитані) транзакцій, а також буде вимірюватися ефективність кожного підходу.

## 2.2 Математичний опис проблемної області дослідження

Для формалізації процесу виконання транзакцій та більшого розуміння цього процесу необхідно побудувати математичну модель. Як уже було зазначено, у досліді буде визначатися залежність часу виконання запитів на читання даних і розподілених транзакцій, а також кількість споживаних ресурсів для усіх трьох підходів за наступними параметрами: кількість даних у базі, кількість одночасних підключень до бази, кількість вузлів-реплік кластері.

Для всіх зазначених параметрів необхідно ввести умовні позначення:

- V (data **v**olume) – кількість даних у базі;
- C (count of **c**onnections) – кількість одночасних підключень;
- RP (count of **rep**licas) – кількість вузлів-реплік;

Під час досліду будуть вимірюватися наступні величини:

- $T_R$ (**t**ime of **r**ead operations) – час виконання запитів на зчитування;
- $T_T$ (**t**ime of **t**ransaction execution) – час виконання транзакцій;
- R (consumed **r**esources) – споживані ресурси ЕВМ.

Тобто, метою досліду буде визначення залежності зазначених величин від зазначених параметрів. В узагальненому вигляді це описується наступним чином:

$$T_R = f(V, C, RP) + \varepsilon, \quad T_T = f(V, C, RP) + \varepsilon, \quad R = f(V, C, RP) + \varepsilon$$

де ε – похибка (відхилення) при вимірюванні.

Причому емпіричним шляхом можливо визначити напрямок залежності (зростає чи спадає разом з параметром) кожної величини від певного параметра:

$$T_R \sim V, \quad T_R \sim C, \quad T_R \sim RP$$
$$T_T \sim V, \quad T_T \sim C, \quad T_T \sim RP$$
$$R \sim V, \quad R \sim C, \quad R \sim RP$$

У свою чергу споживані ресурси є на множиною різних типів ресурсів: місце на постійному носії, оперативна пам’ять, процесорний час:

$$R = \{R_{HDD}, R_{RAM}, R_{CPU}\} = R_{HDD} \cup R_{RAM} \cup R_{CPU}$$

Це означає, що необхідно визначити наступні залежності:

$$R_{HDD} = f(V, C, RP) + \varepsilon$$
$$R_{RAM} = f(V, C, RP) + \varepsilon$$
$$R_{CPU} = f(V, C, RP) + \varepsilon$$

В узагальненому вигляді є 5 величин, залежність яких буде визначатися під час досліду для кожного методу:

- для реалізації транзакційності на рівні додатку: $T_{R\ App}, T_{T\ App}, R_{HDD\ App}, R_{RAM\ App}, R_{CPU\ App}$;
- для досліду вбудованих можливостей транзакційності в MongoDB: $T_{R\ Mongo}, T_{T\ Mongo}, R_{HDD\ Mongo}, R_{RAM\ Mongo}, R_{CPU\ Mongo}$;
- для досліду можливостей транзакційності в VoltDB: $T_{R\ VoltDB}, T_{T\ VoltDB}, R_{HDD\ VoltDB}, R_{RAM\ VoltDB}, R_{CPU\ VoltDB}$.

Основною задачею досліду після визначення функцій залежностей є порівняння цих функцій з метою визначення  того, в якому з методів функція зростає/спадає швидше. Завдяки цьому порівнянню і можна буде робити висновки щодо особливостей використання того чи іншого методу. Оскільки необхідно знайти швидкість зростання функції, то логічно шукати похідну, а для порівняння швидкостей зростання функцій необхідно застосувати ліміти. В кінці кінців необхідно знайти швидкість зростання функції при зміні кожного параметру використовуючи правило Лопіталя (звісно, вважаючи, що всі необхідні умови виконані).

## 2.3 Планування експериментального дослідження

Планування експерименту повинно початися з вибору предметної області, а саме щоб вона була актуальною для сучасних систем, і тому дослід мав би прикладний характер [19]. Після цього під обрану предметну область необхідно спроектувати спрощену версію реальної структури БД з різними видами зв'язків в залежності від обраного методу. І далі для цієї схем потрібно спроектувати транзакційні запити, ефективність виконання яких буде досліджуватися.

### 2.3.1 Розробка структур баз даних для експерименту.

Під час планування експерименту було прийнято рішення, що в якості предметної області було обрано такий реальний сегмент ринку, як електронна комерція. Дана предметна область як раз здебільшого пов'язана з великим навантаженням та великими даними [20]. Також більшість систем, що продають товари або послуги онлайн потребують транзакційності бази даних, адже вони зазвичай надають функціональність створення замовлення та їх оплати, що є самі по собі повинні виконуватися однією транзакцією. Отже, проаналізувавши вищесказане, можна зробити висновок, що область електронної комерції ідеально підходить для реалізації транзакційності в розподіленій базі даних.

Тепер необхідно проаналізувати сутності в базі даних та їх відношення. Звісно, що кожна система електронної комерції має свою унікальну схему бази даних з різною кількістю сутностей та відношень. Але для даного експерименту досить виділити основну частину баз даних електронної комерції та створити свою спрощену базу, структури якої буде досить для проведення експерименту.

Було прийнято рішення моделювати БД на основі продажу деяких товарів в інтернет магазині. У такій базі все починається безпосередньо з сутності товарів,

що матиме свою назву, описання, доступну кількість, ціну, категорію і набір властивостей, що характеризують різні сторони даного товару.

З іншого боку є клієнт сайту, що може замовляти наявні товари у певній кількості. Цей процес генерує нову сутність елементу замовлення клієнтом певного продукту у певній кількості. Набір елементів замовлення складають сутність замовлення, яке також повинне мати дату доставки та можливу знижку. Кожне замовлення в свою чергу належить одному певному клієнту.

На основі наведеного аналізу сутностей та зв'язків між ними можна навести ER-діаграму [21], що візуально демонструє ці зв'язки (див. рис. 2.2).

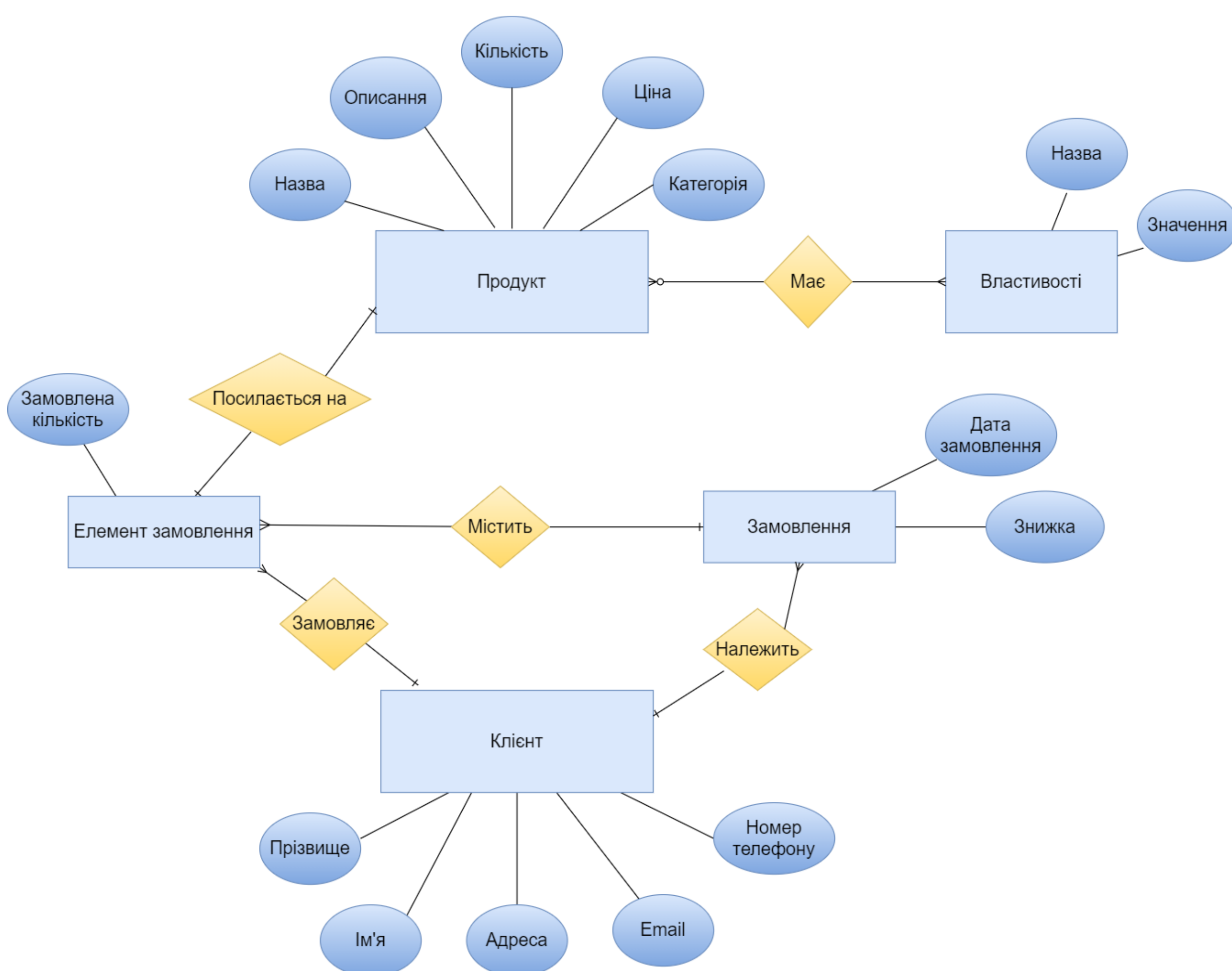


Рисунок 2.2 – ER-діаграма бази даних, що проектується

Далі доречно почати проектувати базу даних для реляційної моделі. Основні таблиці реляційної бази даних співпадатимуть з сутностями ER-моделі, а зв'язки

між цими таблицями зі зв'язками між сутностями. Також можна проаналізувати, що одній категорії (наприклад, «Ноутбуки» або «Телефони») можуть належати багато продуктів. Саме тому для реляційної моделі логічно створити окрему таблицю категорій та виділити відношення один-до-багатьох з таблицею продуктів. Також можна помітити, що в даній моделі на відміну від ER-діаграми таблиця клієнтів не потребує окремого зв'язку з таблицею елементів замовлення, адже ці елементи можуть бути знайдені через таблицю замовлень, на яку в свою чергу посилаються записи з таблиці елементів замовлення.

Отже, реляційна база даних матиме наступні таблиці та відношення:

- таблиця клієнтів (Customers);
- таблиця категорій товарів (Categories);
- таблиця продуктів (Products) – посилається на таблицю категорій та має з нею відношення один-до-багатьох: одна категорія може належати багатьом продуктам;
- таблиця властивостей продуктів (ProductProperties) – посилається на таблицю продуктів та має з нею відношення один-до-багатьох: один продукт може мати багато властивостей;
- таблиця замовлень (Orders) – посилається на таблицю клієнтів та має з нею відношення один-до-багатьох: один клієнт може мати багато замовлень;
- таблиця елементів замовлення (OrderItems) – посилається на таблицю замовлень та має з нею відношення один-до-багатьох: одне замовлення може мати багато елементів. Також дана таблиця посилається на таблицю товарів та має з нею відношення один-до-одного: один елемент замовлення представляє собою один певний товар.

Проаналізувавши все вище сказане також можна помітити, що між таблицями клієнтів та товарів є зв'язок багато-до-багатьох: один клієнт може замовити багато товарів, а один товар замовити багато клієнтів. У базі даних це

відношення реалізується через дві проміжні таблиці замовлень та елементів замовлень.

Отже, в результаті аналізу була спроектована реляційна база даних для експерименту та для кращого її розуміння надана її візуалізація (див. рис. 2.3). У реалізованій базі даних присутні відношення усіх типів, а також сама база знаходиться у 3-ій нормальній формі.

Наступним кроком є проектування структури бази даних для NoSQL підходу. Важливим зауваження є те, що всі NoSQL СУБД зберігають дані у своєму окремому форматі, тобто не існує уніфікованої структури бази на відміну від реляційних бази даних.

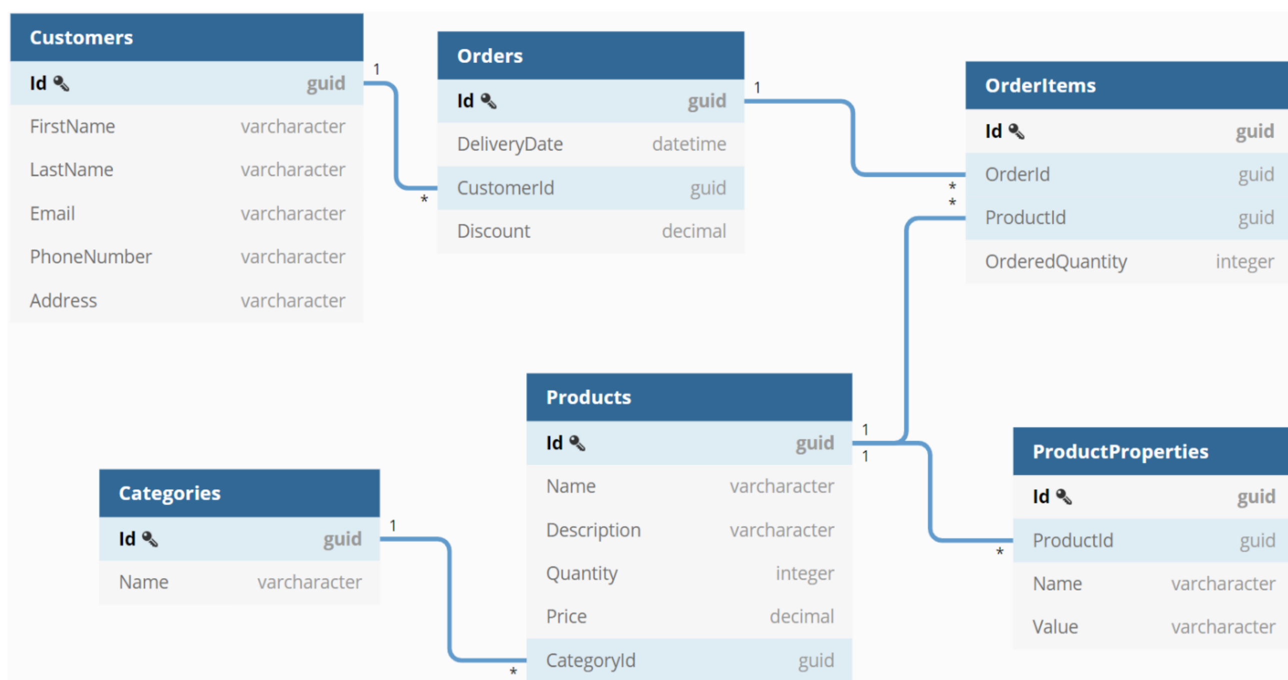


Рисунок 2.3 – Реляційна схема бази даних

Оскільки в якості NoSQL СУБД було обрано MongoDB, то проектування буде здійснюватися саме під цю базу. Дана СУБД замість таблиць визначає колекції, а замість записів документи. Документи на відміну від записів у SQL базах даних може не мати жорстко прописаної структури. MongoDB зберігає документи в колекціях у так званому BSON форматі, логічним представленням

якого є відомий формату обміну даних JSON. Тому проектування та візуалізація структури бази даних для MongoDB буде здійснюватися на основі формату JSON.

Зіставляючи ER-діаграму з необхідним форматом, можна побачити, що для сутності елементу замовлення не потрібна окрема колекція, адже вона існує тільки в рамках сутності замовлення, тому елементи замовлення логічно зробити вкладеним масивом документу замовлення. Те ж саме стосується і властивостей товарів, адже вони існують тільки в рамках конкретного продукту, то їх також доречно зробити вкладеним масивом у документі товару.

Для наглядного демонстрування структури бази даних для MongoDB доречно навести її візуально (див. рис. 2.4).

```
"Categories"
{
  "id": "ObjectId",
  "name": "string"
}

"Products"
{
  "id": "ObjectId",
  "name": "string",
  "description": "string",
  "quantity": "number",
  "price": "decimal",
  "category": "ObjectId",
  "properties": [{
    "name": "string",
    "value": "string"
  },
  ...
  ]
}
```

```
"Customers"
{
  "id": "ObjectId",
  "first_name": "string",
  "last_name": "string",
  "email": "string",
  "phone_number": "string",
  "address": "string"
}

"Orders"
{
  "id": "ObjectId",
  "delivery_date": "Date",
  "customer": {
    "id": "ObjectId",
    "email": "string"
  },
  "items": [{
    "product_id": "ObjectId",
    "product_name": "string",
    "ordered_quantity": "number"
  },
  ...
  ]
}
```

Рисунок 2.4 – Структура колекцій для MongoDB

Тобто, можна виділити наступні колекції бази даних для MongoDB:

– клієнти (clients);

- категорії (categories);
- замовлення (orders) – документи цієї колекції зберігають як дані самого замовлення, так і його елементів;
- товари (products) – документи цієї колекції зберігають дані про товар, а також його властивості.

У результаті аналізу було розроблено структури баз даних для реляційної моделі даних, а також для BSON формату, що використовується в MongoDB.

### 2.3.2 Обмеження експерименту.

Перед за все необхідно описати обмеження дослідження. Першим з таких обмежень є використання рівня ізоляції Read Committed за замовченням для кожного підходу. Для MongoDB цей рівень ізоляції досягається за допомогою встановлення параметрів read concern та write concern. Write concern параметр повинен мати значення, рівним кількості вузлів у кластері, read concern значення snapshot, щоб зчитувати тільки узгоджені дані [22].

Наступним обмеженням є вид розподіленої бази даних. Вона може розподілятися за допомогою реплікація, або шардингу, або комбінації цих двох підходів. Для дослідження було обрано реплікацію, адже саме на такій розподіленій базі більш доречно перевіряти атомарність та консистентність транзакцій, бо ці транзакції повинні атомарно та консистентно зафіксуватися на всіх вузлах кластеру. А для шардингу в якості правила розподілення найбільш часто використовується геолокація, де зв'язані між собою дані зберігаються на одному вузлі, а тому і немає потреби у розподіленій фіксації транзакції.

Заміри будуть робитися для кластеру з віртуальним машин-серверів баз даних. Кожен сервер працюватиме в хмарі Azure на віртуальній машині типу B2s

[23, 24], що містить 2 CPU, 4 GB оперативної пам'яті та 8 GB жорсткого диску. В якості операційної системи використовуватиметься Windows Server 2016.

Останнім обмеженням є кількість даних, кількість одночасних підключень та кількість реплік, для яких будуть проводитися заміри. Очевидно, що всі ці величини можуть бути нескінченними, а тому для експерименту потрібно виділити певні дискретні величини, на яких будуть проводитися заміри. Було прийнято рішення виділити певні режими, що представлятимуть собою конкретні дискретні значення вищезазначених величин.

Перед наведенням режимів необхідно описати, як повинні змінюватися задані величини. Починаючи з даних, доречно виявити, дані яких саме сутностей будуть зростати з тим чи іншим темпом. Наприклад, категорії товарів є досить статичними сутностями, їх кількість невелика та вони рідко змінюються. Такими ж досить статичними сутностями є властивості товарів. Для спрощення дослідження кожен товар матиме всього 4 властивості. Більш цікавими у цьому плані є сутності клієнтів, товарів, замовлень та їх елементів, кількість яких зростає у дослідженні при замірах буде зростати певною мірою. При цьому під час наповнення бази даними будуть дотримуватися пропорції, що справедливі для більшості систем електронної комерції: товарів зазвичай більше, ніж клієнтів, а сумарна кількість замовлень та їх елементів набагато більша за кількість товарів. Виходячи з цього, було виділено наступні режими з точки зору кількості даних:

– Basic;
– Basic+;
– Basic++;
– Medium;
– Intensive.

Режими у списку відсортовані у порядку зростання кількості сутностей. На кожному наступному режимі розрив у кількості сутностей з попереднім значно збільшується. Нижче наведена таблиця кількості даних щодо кожної сутності для кожного режиму (див. табл. 3).

Таблиця 3 – Кількість даних для різних режимів

| | Категорії | Клієнти | Товари | Замовлення | Елементи замовлення | Всього сутностей |
|---|---|---|---|---|---|---|
| Basic | 20 | 40 | 2 000 | 4 000 | 30 | 122 064 |
| Basic+ | 20 | 40 | 8 000 | 14 000 | 40 | 568 064 |
| Basic++ | 20 | 40 | 16 000 | 25 000 | 50 | 1 266 064 |
| Medium | 20 | 400 | 20 000 | 40 000 | 60 | 2 420 064 |
| Intensive | 20 | 4 000 | 50 000 | 80 000 | 100 | 8 050 064 |

Далі розглядаються значення режимів для метрики кількості одночасних запитів. Дана метрика напряму залежить від кількості користувачів у системі та є індивідуальною для кожного додатку. Існують популярні системи типу Amazon, які мають близько 1000 запитів за секунду, а є локальні інтернет-магазини, що мають до 10 користувачів у день. Дослідження буде проводитися на середній кількості одночасних запитів, тобто на десятках та тисячах з’єднань в один і той же момент. Для цієї метрики було обрано ті ж 5 режимів, при цьому режими типу Basic будуть між собою відрізнятися на декілька десятків, а режими Medium та Intensive відрізнятимуться у декілька разів порівняно з іншими. При цьому з кожним вищим режимом ця різниця буде збільшуватися. Нижче наведена таблиця з кількістю одночасних підключень до бази, в рамках яких виконуються запити (див. табл. 4).

Таблиця 4 – Кількість одночасних підключень для різних режимів

| | Кількість одночасних підключень |
|---|---|
| Basic | 10 |
| Basic+ | 30 |
| Basic++ | 50 |
| Medium | 100 |
| Intensive | 300 |

Останнім кроком є визначення значень кількості вузлів у розподіленій базі даних. Ця метрика також є дуже індивідуальною для кожного додатку, але кількість вузлів зростає дуже повільно, зазвичай при підвищенні навантаження додають по одному вузлу. В силу технічних обмежень було прийнято рішення ввести наступні чотири режими, в кожен з яких буде відрізнятися від іншого на 1-2 вузли:

- Basic;
- Basic+;
- Medium;
- Intensive.

Нижче наведена таблиця з кількістю розгорнути вузлів у розподіленій базі даних (див. табл. 5).

Таблиця 5 – Кількість вузлів у кластері

| | Кількість вузлів |
|---|---|
| Basic | 2 |
| Basic+ | 3 |
| Medium | 4 |
| Intensive | 6 |

Для кожного фактору навантаження (кількість даних, підключень або вузлів) будуть проводитися заміри двох типів:

- змінюються значення лише даного фактору, а значення інших факторів залишаються статичними. Такий підхід потрібен для визначення залежності часу виконання та споживання ресурсів від конкретного фактору;
- змінюються одночасно всі фактори. Цей підхід потрібен для визначення сумарного впливу всіх факторів. При цьому значення всіх факторів будуть збільшуватися на один рівень одночасно.

2.3.3 Моделювання предметної області та бізнес-процесів.

Різні системи електроної комерції, насправді пропонують різний функціонал своїм користувачам. Але в цьому розділі будуть виділені найбільш поширені бізнес-операції, на основі яких будуватимуться запити. Деякі операції стосуватимуться безпосередньо клієнтів системи, а інші її адміністраторів. У результаті було виділено наступні операції:

– вивід найбільш популярних товарів. Зазвичай переходячи на головну сторінку будь-якого інтернет-магазину ми бачимо список товарів, які купують найчастіше за певний останній період. Такий підхід дозволяє пропонувати найрелевантніші товари і робити на цьому більше продаж;
– звісно, що для клієнтів таких систем найбільш актуальним функціоналом є робота із замовленнями, тобто їх створення та перегляд;
– зміна кількості товарів, наприклад, зменшення у результаті успішного виконання замовлення, або збільшення при поповненні товару на складі;
– зміна ціни товару. Така операція може знадобитися для різного роду знижок або ціна може змінюватися в залежності від попиту на товар;
– додавання або зміна знижки на певне замовлення. Частіше за все ця відбувається при вводі користувачем промо-коду або коли користувач є постійним клієнтом системи;
– додавання нових категорій та товарів для них. Дана операція знадобляється при розширенні асортименту, виникненні нових іноваційних та актуальних товарів;
– зміна дати замовлення як адміністратором, так і клієнтом у випадку виявлення форс-мажорних обставин з кожного боку.

Звісно, що наведеними операціями не обмежують різні системи електронної комерції, але вони є базовими для таких додатків та на основі цих операцій у подальшому будуть спроектовані відповідні запити до бази даних.

У результаті були виявлені основні бізнес-операції для клієнта та адміністратора системи електронної комерції. Для більш наглядного демонстрування цих операцій нижче наведена діаграма прецедентів (див. рис. 2.5).

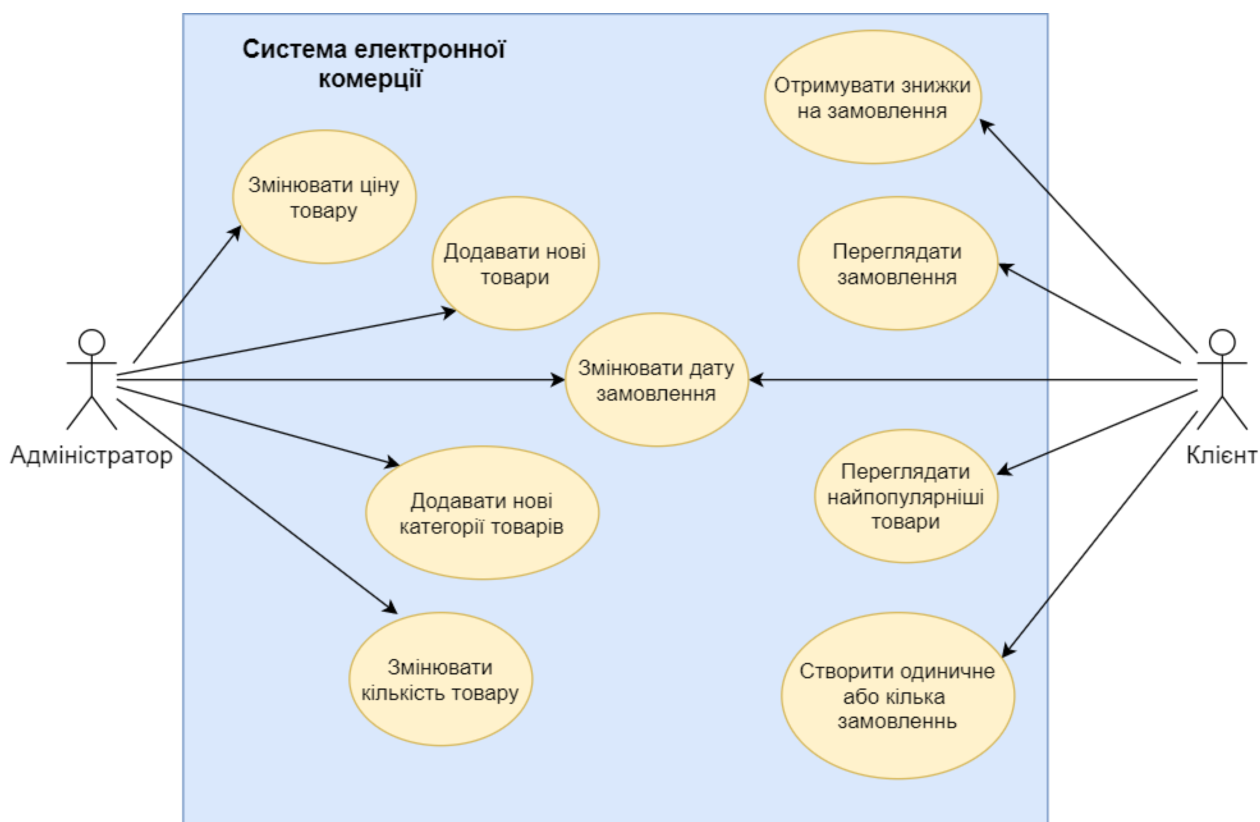


Рисунок 2.5 – Діаграма прецедентів основних операцій у електронній комерції

У результаті були промодельовані основні операції, що відбуваються в більшості систем електронної комерції.

## 2.4 Проектування програмного забезпечення для експерименту

Для проведення експерименту було прийнято рішення розробити програмне забезпечення, яке б робило необхідні заміри при заданому навантаженні та

автоматично б розгортало кластер розподіленої бази даних. Розгортання буде проходити через командний рядок та CLI, що надає СУБД. Оскільки кластер бази та програма для вимірювання будуть працювати на операційних системах із сімейства Windows, то для розробки ПЗ було прийнято рішення використовувати мову програмування C# та платформу .Net Core 3.1, що мають багато вбудованого функціоналу для роботи з Windows ОС.

Оскільки ПЗ не є кінцевою метою дослідження, а тільки інструментом для його виконання, то в якості архітектури обрана звичайна класична трьох-шарова архітектура (див. рис. 2.6): ПЗ матиме презентаційний рівень, рівень бізнес-логіки та доступу до даних [25].

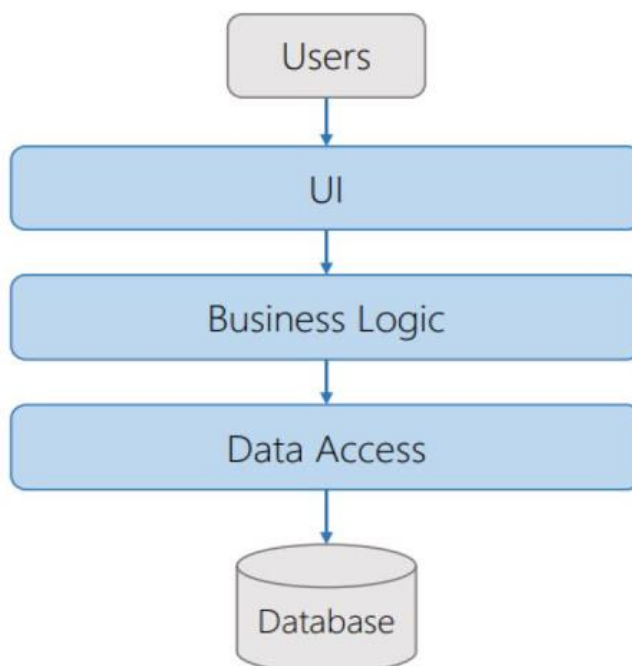


Рисунок 2.6 – Трьох-шарова архітектура ПЗ

Найбільш цікавим представляється шар бізнес-логіки, адже саме в ньому буде реалізована логіка автоматичного розгортування розподіленої БД та замірювання значень обраних метрик. Варто зазначити, що вимірювання часу виконання запиту буде здійснюватися за допомого спеціального C# класу Stopwatch, що має під собою низькорівневу реалізацію заміру на основі тіків процесора, що надають найменшу можливу гранулярність часу [26]. Ці тіки легко

можуть бути конвертовані в секунди через ділення їх на частоту процесора. Для визначення розміру бази даних будуть просто заміряні розміри всіх файлів цієї бази. При вимірюванні кількості процесорного часу та оперативної пам’яті, що потребує розподілена база, використовуватимуться спеціальні лічильники, що є частиною платформи Windows [27].

Наступним аспектом шару бізнес-логіки є спосіб виконання послідовності дій з базою даних, а саме: спочатку необхідно розгорнути кластер, далі заповнити його даними, потім послідовно виповнити та заміряти запити для обраного режиму. Для цього буде використовуватися так звана конвеєрна архітектура модулів додатку, що дозволяє писати масштабований код [28] (див. рис. 2.7). Суть її полягає в тому, щоб розбити всю логіку систему на окремі модулі-фази конвеєру, при цьому ці модулі виконується послідовно один за одним у строго заданому порядку [29]. Результатом виконання певного модуля можуть користуватися усі наступні модулі.

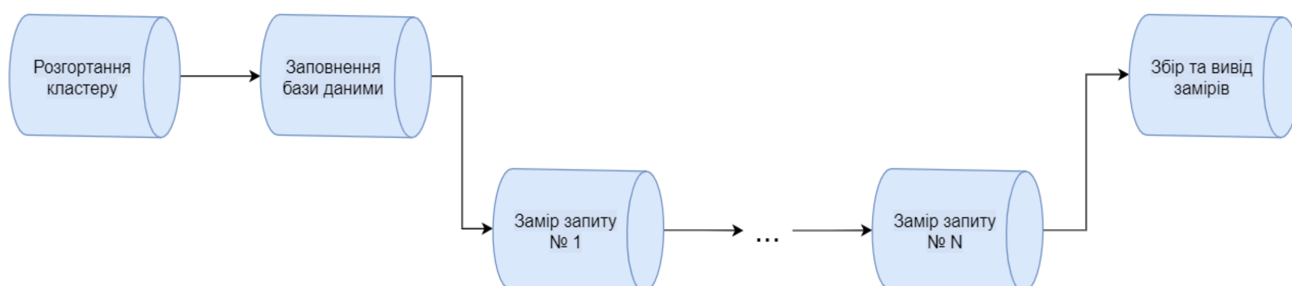


Рисунок 2.7 – Конвеєрна архітектура бізнес-логіки ПЗ

Далі доречно було б більш детально описати потік виконання замірів програмним забезпеченням. Перш за все програма повинна отримати на вхід значення режимів для кожного фактору навантаження. Потім програма зчитує конкретні величини, що відповідають цим факторам. Після цього зчитування першим кроком є безпосередньо розгортування розподіленої бази даних з кількістю вузлів, що відповідає заданому раніше режиму. Коли БД успішно розгорнута, необхідно спочатку заміряти її розмір як одну із метрик, а потім заповнити її початковими даними, при цьому кількість даних у кожній

таблиці/колекції визначається кількістю сутностей для обраного режиму. Після початкового заповнення БД стартує циклічний процес виконання та заміру запитів один за одним. При замірі кожного запиту його будуть одночасно виконувати декілька різних потоків додатку, кількість яких теж регулюється заданим режимом. Також варто зазначити, що після виконання певних запитів потрібне перезаповнення бази перед виконанням наступного запиту. Це потрібно для того, щоб всі запити виконувалися на одному фіксованому об’ємі даних. І в кінці після виконання всі запитів всі заміри групуються та з них виділяється середнє значення часу виконання, затребуваний процесорний час та оперативна пам’ять.

Для візуалізації цього процесу наведено діаграму активності (див. рис. 2.8).

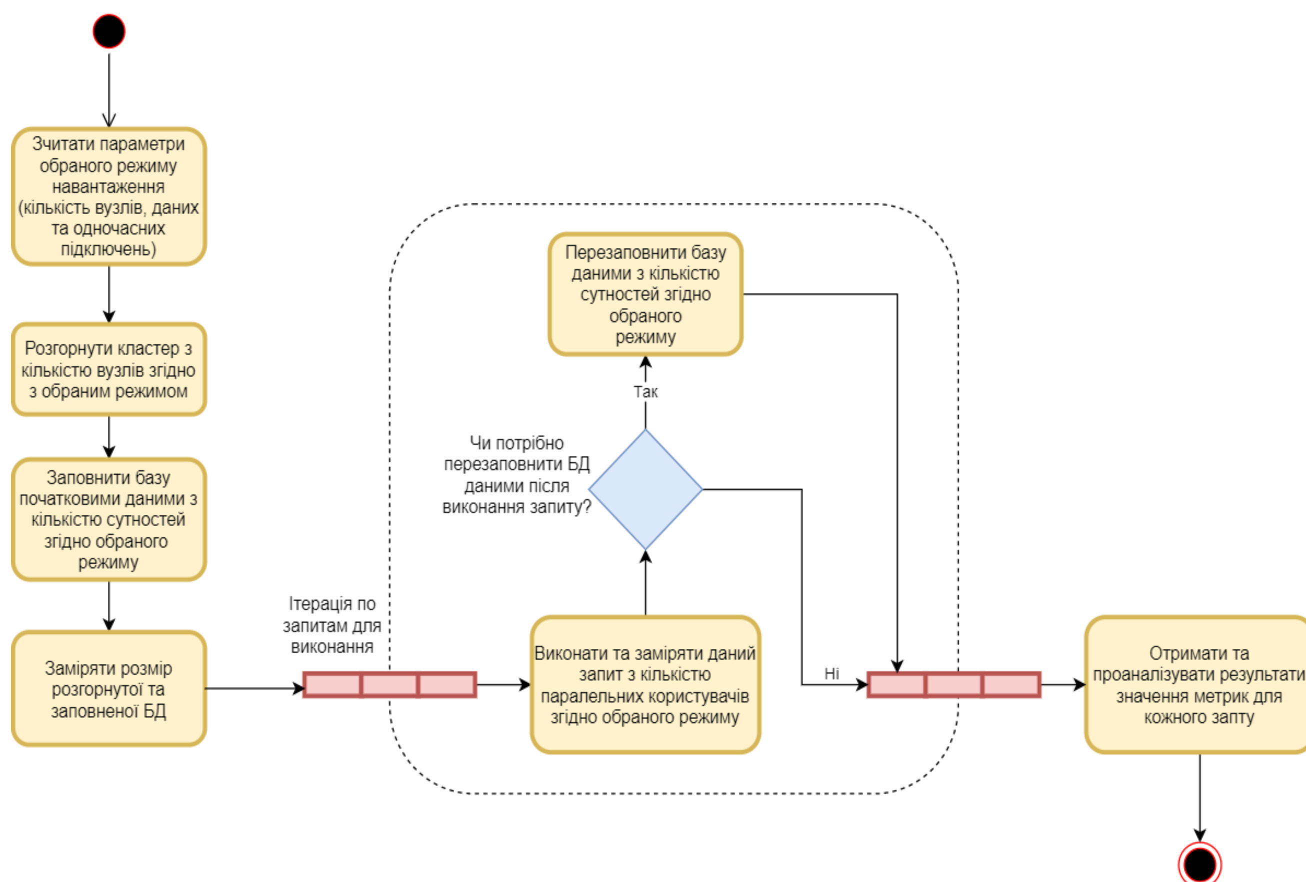


Рисунок 2.8 – Діаграма активності процесу виконання замірів

Також окремим моментом є те, що після завершення роботи додаток повністю деконструює створений у самому початку кластер бази даних та фізично

видаляє всі файли, пов’язані з базою даних. За рахунок цього попереджується захламлення робочої машини.

Важливо зазначити, що для заповнення бази даних беруться не випадкові незначимі дані, а за допомогою бібліотеки Bogus для всіх таблиць/колекцій генеруються реалістичні дані, що співпадають із предметною областю бази даних (див. рис. 2.9). Дана бібліотека підтримує генерацію реальних імен, номерів телефонів, адрес, назв товарів та багато іншого [30].

| | |
|---|---|
| [1] (id="6044072b160ac3cbf93b2cbd") | |
| _id | 6044072b160ac3cbf93b2cbd |
| firstName | Enrique |
| lastName | Larson |
| email | Enrique.Larson@yahoo.com |
| phoneNumber | 512-380-9813 x0192 |
| address | Heaneyborough, 315 Mertz Flats, 65532-0796 |
| [2] (id="6044072b160ac3cbf93b2cbe") | |
| _id | 6044072b160ac3cbf93b2cbe |
| firstName | Penny |
| lastName | Kris |
| email | Penny.Kris@gmail.com |
| phoneNumber | 697.591.6539 |
| address | New Ignatius, 4781 Delmer Trace, 44726 |
| [3] (id="6044072b160ac3cbf93b2cbf") | |
| _id | 6044072b160ac3cbf93b2cbf |
| firstName | Felix |
| lastName | Russel |
| email | Felix.Russel@yahoo.com |
| phoneNumber | 541-591-4698 |
| address | Kuvalischester, 238 Ferry Cape, 83396 |

Рисунок 2.9 – Приклад генерації даних бібліотекою Bogus для колекції клієнтів

Також необхідно сказати, що кожен вузол-репліка буде знаходитися на окремому сервері бази даних. Множина цих серверів формує сам кластер розподіленої БД. Додаток, що робить заміри знаходитиметься на окремій фізичній машині та буде взаємодіяти з кластером через TCP протокол.

У цій архітектурі буде присутній так званий Primary вузол бази даних, який відповідатиме за коректну роботу всього кластеру. Але запити зчитування та запису будуть відсилатися на перший вільний вузол.

Описана архітектура проілюстрована на діаграмі розгортування (див. рис. 2.10).

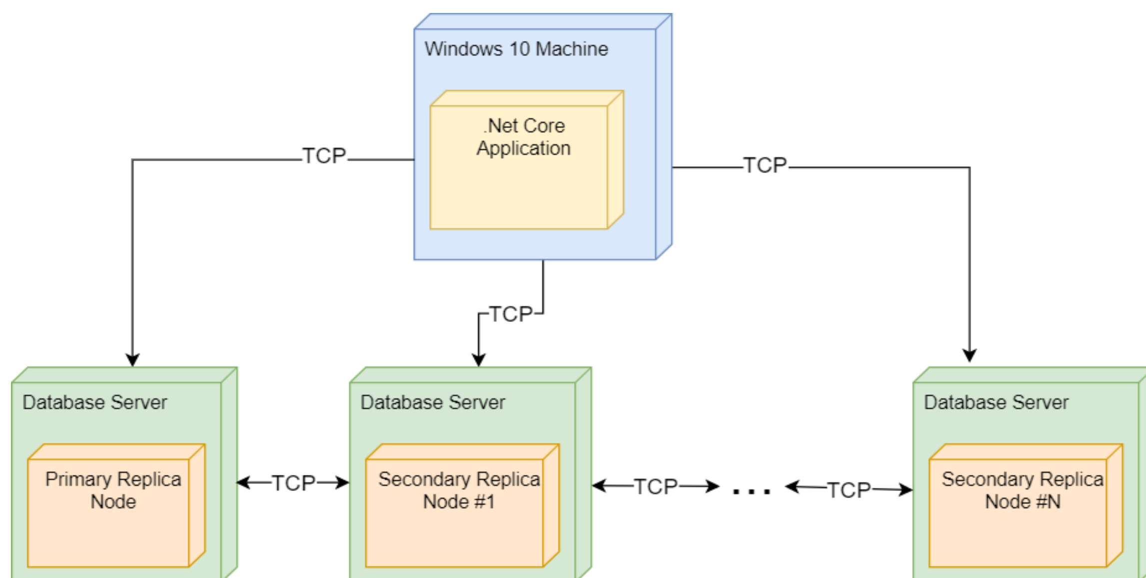


Рисунок 2.10 – Діаграма розгортування інфраструктури для дослідження

Таким чином, було спроектовано програмне забезпечення, необхідне для проведення експерименту.

# 3 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

## 3.1 Розробка фізичної моделі реляційної БД

Оскільки реляційна модель має свою схему даних, то в першу чергу перед проведенням експерименту потрібно задекларувати SQL таблиці, для яких будуть виконуватися запити. Скрипти для їх створення наведені на мові T-SQL, але вони будуть схожі і для інших СУБД, тільки деякі типи даних можуть відрізнятися.

Нижче наведені скрипти створення таблиць, що відносяться до домену товарів, а саме таблиці категорій, товарів та їх властивостей.

```
CREATE TABLE Categories (
   Id UNIQUEIDENTIFIER PRIMARY KEY,
   Name NVARCHAR(255) NOT NULL
);

CREATE TABLE Products (
   Id UNIQUEIDENTIFIER PRIMARY KEY,
   Name NVARCHAR(255) NOT NULL,
   Description NVARCHAR(4096),
   Quantity INTEGER NOT NULL,
   Price DECIMAL NOT NULL,
   CategoryId UNIQUEIDENTIFIER NOT NULL,
   CONSTRAINT FK_Categories_Products FOREIGN KEY (CategoryId)
REFERENCES Categories.Id
);

CREATE TABLE ProductProperties (
   Id UNIQUEIDENTIFIER PRIMARY KEY,
   Name NVARCHAR(255) NOT NULL,
   ProductId UNIQUEIDENTIFIER NOT NULL,
   CONSTRAINT FK_Products_Properties FOREIGN KEY (ProductId)
REFERENCES Products.Id
);
```

У таблиць вище можна побачити два відношення один-до-багатьох, а саме таблиця товарів посилається на таблицю категорій, а в свою чергу таблиця властивостей товару на товари.

Далі наведено скрипт створення таблиці клієнтів, яка сама по собі незалежна і не посилається на інші таблиці.

```
CREATE TABLE Customers (
   Id UNIQUEIDENTIFIER PRIMARY KEY,
   FirstName NVARCHAR(255) NOT NULL,
   LastName NVARCHAR(255) NOT NULL,
   Email NVARCHAR(255) NOT NULL,
```

```
   PhoneNumber NVARCHAR(128) NOT NULL,
   Address NVARCHAR(255)
);
```

Останнім кроком є об’єднання таблиць клієнтів та товарів зв’язком багато-до-багатьох через таблиці замовлень та елементів замовлення. Замовлення посилатимуться на певного клієнта, а його елементи на конкретне замовлення та конкретний товар.

```
CREATE TABLE Orders (
   Id UNIQUEIDENTIFIER PRIMARY KEY,
   DeliveryDate DATETIME2 NOT NULL,
   CustomerId UNIQUEIDENTIFIER NOT NULL,
   Discount REAL NOT NULL,
   CONSTRAINT FK_Customer_Orders FOREIGN KEY (CustomerId) REFERENCES
Customers.Id
);

CREATE TABLE OrderItems (
   Id UNIQUEIDENTIFIER PRIMARY KEY,
   OrderId UNIQUEIDENTIFIER NOT NULL,
   ProductId UNIQUEIDENTIFIER NOT NULL,
   CONSTRAINT FK_Order_Items FOREIGN KEY (OrderId) REFERENCES
Orders.Id,
   CONSTRAINT FK_Product_Item FOREIGN KEY (ProductId) REFERENCES
Products.Id
);
```

Отже, в результаті були розроблені скрипти для створення таблиць для реляційних баз даних, у результаті чого в подальшому будуть розроблені запити до цих таблиць. На основі цих скриптів будуть створені таблиці для реляційних методів реалізації розподілених ACID транзакцій, а саме

## 3.2 Розробка транзакцій

Також для проведення експерименту необхідно описати запити, що будуть виконуватися для бази даних. Очевидно, що ці запити повинні бути різносторонніми, щоб у результаті замірів можна було зробити висновки щодо ефективності виконання тих чи інших запитів. Запити будуть реалізовуватися

таким чином, щоб подібні операції потенційно могли використовуватися у сфері електронної комерції.

Основними для заміру запитами є транзакційні запити, але також для більш всеосяжного аналізу ефективності тієї чи іншої СУБД доречно також заміряти операції на зчитування, що не потребують транзакцій.

Першим таким запитом було вирішено зробити зчитування даних про замовлення за його ідентифікатором. Даний запит дозволить виміряти ефективність пошуку за індексом у розподіленій базі даних. І для прикладного використання цей запит є однозначно корисним, адже користувачі зазвичай мають можливість переглядати деталі свого замовлення на окремій сторінці. На мові SQL даний запит матиме наступний вигляд:

```
SELECT * FROM Orders WHERE Id = @OrderId
```

У свою чергу цей запит у BSON форматі, що використовується в MongoDB, мав би наступний вигляд:

```
db.orders.find({ "_id": @OrderId })
```

Далі доречно було б реалізувати більш складний запит зчитування зі складним предикатом та групуванням. З точки зору прикладного використання таким запитом могло би бути отримання трьох найбільш популярних категорій товарів за останній місяць Популярність категорії визначається кількістю товарів цієї категорії, що була замовлена впродовж останніх 30 днів. Цей запит міг би бути корисний як для адміністраторів або власників системи при аналізі найбільш перспективних напрямків продажу, так і для користувачів, щоб на головній сторінці пропонувати їм товари з цих категорій. На процедурній мові T-SQL цей запит має наступний код:

```
WITH Sales AS (
  SELECT i.ProductId, SUM(i.Quantity) AS Total FROM OrderItems i
  JOIN Orders o ON i.OrderId = o.Id
  WHERE o.DeliveryDate >= DATEADD(month, -1, GETDATE())
  GROUP BY i.ProductId
)
SELECT TOP 3 c.* FROM Categories c
JOIN Products p ON p.CategoryId = c.Id
JOIN Sales s ON p.ProductId = s.ProductId
ORDER BY Total DESC
```

Той же самий запит для MongoDB матиме наступний формат з більшою кількістю коду:

```
today = new Date();
date = today.setMonth(today.getMonth() - 1);
db.orders.aggregate([
{
   "$match":{
      "deliveryDate":{
         "$lte": date
      }
   }
},
{
   "$unwind":"$items"
},
{
   "$project":{
      "_id":0,
      "items":"$items"
   }
},
{
   "$group":{
      "_id":"$items.productId",
      "count":{
         "$sum":1
      }
   }
},
{
   "$lookup":{
      "from":"inventory",
      "localField":"_id",
      "foreignField":"_id",
      "as":"products"
   }
},
{
   "$unwind":"$products"
},
{
   "$group":{
      "_id":"$products.categoryId",
      "count":{
         "$sum":"$count"
      }
   }
},
{
   "$sort":{
      "count":-1
   }
},
{
   "$limit":3
```

```
},
{
   "$lookup":{
      "from":"categories",
      "localField":"_id",
      "foreignField":"_id",
      "as":"categories"
   }
},
{
   "$unwind":"$categories"
},
{
   "$project":{
      "categoryName":"$categories.name",
      "count":"$count",
      "_id":0
   }
},
{
   "$sort":{
      "count":-1
   }
}
])
```

Наступними запитами будуть прості однотипні транзакційні запити з операціями вставки, оновлення та видалення. Першим з таким запитів може бути вставка замовлення зі змінною кількістю елементів замовлення. Кількість елементів замовлень для вставки логічно зробити еквівалентною кількості елементів для даного режиму. Запит для створення замовлення є одним з найбільш поширених в електронній комерції, адже саме цю операцію виконують клієнти системи. На мові T-SQL запити вставки замовлення і його елементів мають наступний вигляд:

```
BEGIN TRAN;
DECLARE @OrderId UNIQUEIDENTIFIER;
INSERT INTO Orders (DeliveryDate, CustomerId, Discount)
   OUTPUT INSERTED.Id INTO @OrderId
   VALUES (@DeliveryDate, @CustomerId, @Discount);
INSERT INTO OrderItems (OrderId, ProductId, OrderedQuantity)
   VALUES (@OrderId, @ProductId, @OrderdQuantity), ...;
COMMIT TRAN;
```

Цей же запит для MongoDB виглядає таким чином:

```
session = db.getMongo().startSession();
session.startTransaction();
orderCollection = session.getDatabase('inventory').orders;
orderCollection.insertOne({
   delivery_date: @DeliveryDate,
   discount: @Discount,
   customer: {
```

```
         id: @OrderCustomerId,
         email: @OrderCustomerEmail
      },
      items: [
         {
            product_id: @ProductId,
            product_name: @ProductName,
            ordered_quantity: @OrderedQuantity
         },
         ...
      ]
   });
   session.commitTransaction();
```

Наступним запитом було прийнято рішення реалізувати операцію оновлення товарів, а саме зменшити ціну тим товарам на 10%, кількість яких на складі менша за 10 одиниць. Такий запит має сенсе, бо дуже часто ті ж електронні магазини, як і звичайні, зменшують ціну товару, що закінчується, щоб якомога скоріше розпродати всю партію. Далі наведено код оновлення на мові SQL:

```
UPDATE Products
   SET Price = Price * 0.9
   WHERE Quantity < 10
```

Для MongoDB цей же запит має наступний вигляд:

```
session = db.getMongo().startSession();
session.startTransaction();
productCollection = session.getDatabase('inventory').products;
productCollection.update(
   { quantity: { $lt: 10 } },
   { $mul: { price: 0.9 } }
);
session.commitTransaction();
```

Ключовим запитом наступної транзакції є видалення. Операція видалення, насправді, дуже рідко використовується в електронній комерції, адже, зазвичай, у такій сфері важливо мати та відслідковувати усі зміни даних. Тим паче, ще важче знайти складний запит видалення, щоб можна було оцінити ефективність роботи СУБД. Саме тому для досліду взято навчальний запит, суть якого полягає у видаленні всіх замовлень, загальна сума яких знаходиться на певному відрізку $[x; x + 500]$, де $x \in [8\ 000; 13\ 000]$. Саме в цьому діапазоні знаходяться більшість замовлень. На мові T-SQL цей запит має наступний вигляд:

```
DELETE FROM Orders WHERE Id IN (
   SELECT OrderId FROM OrderItems
   GROUP BY OrderId
   HAVING SUM(Price) BETWEEN @MinPrice AND @MaxPrice)
```

Цей же запит для MongoDB виглядає так:

```
session = db.getMongo().startSession();
session.startTransaction();
orderCollection = session.getDatabase('inventory').orders;
ordersToDelete = orderCollection.aggregate([
{
   "$unwind" : "$items"
},
{
   "$lookup" : {
     "from" : "inventory",
     "localField" : "items.productId",
     "foreignField" : "_id", "as" : "products"
   }
},
{
   "$unwind" : "$products"
},
{
   "$group" : {
     "_id" : "$_id",
     "total" : {
          "$sum" : "$products.price"
     }
   }
},
{
   "$match" : {
     "total" : {
          "$lte" : 10922,
          "$gte" : 10700
     }
   }
},
{
   "$project": { "_id": "$_id" }
}]).toArray();

orderCollection.deleteMany({
   "_id": { $in: ordersToDelete }
});
session.commitTransaction();
```

Далі було прийнято рішення перевірити ефективність виконання транзакцій з різними типами запитами. Для цих комплексних транзакцій не так важливо, які саме запити виконуються, як те, за який час та ресурси СУБД зможе атомарно та консистентно зафіксувати кілька різноманітних запитів. Для дослідження будуть використовуватися дві комплексні транзакції.

Першою з них є транзакція, що містить два запити на оновлення та один запит на вставку. Для вставки можна використати запит створення замовлення,

але створювати не одне, а наприклад п’ять замовлень за раз. Перший запитом на оновлення було обрано збільшення кількості товарів на складі, поточна кількість яких є меншою за 15. Така операція доречна в електронній комерції для поповнення запасів на складі. І другою операцією оновлення є переніс дати замовлення у певному діапазоні на кілька днів вперед. Даний запит може знадобитися для переносу замовлень у випадку форс-мажорних обставин. У цілому дана транзакція на мові T-SQL матиме наступний вигляд:

```
BEGIN TRAN;

DECLARE @DeliveryDate DATE;
DECLARE @CustomerId UNIQUEIDENTIFIER;
DECLARE @Discount FLOAT;

DECLARE OrdersCursor CURSOR FOR SELECT * FROM @OrdersToInsert;
OPEN OrdersCursor;

FETCH NEXT FROM OrdersCursor INTO @DeliveryDate, @CustomerId,
@Discount;

WHILE @@FETCH_STATUS = 0
BEGIN
     DECLARE @OrderId INTEGER;
     DECLARE @ProductId UNIQUEIDENTIFIER;
     DECLARE @OrderedQuantity INTEGER;

     INSERT INTO Orders (DeliveryDate, CustomerId, Discount)
          OUTPUT INSERTED.Id INTO @OrderId
          VALUES(@DeliveryDate, @CustomerId, @Discount);

     DECLARE ItemsCursor CURSOR FOR
          SELECT * FROM @ItemsToInsert WHERE OrderIndex =
@@CursorRows;
     OPEN ItemsCursor;

     FETCH NEXT FROM ItemsCursor INTO @ProductId, @OrderedQuantity;

     WHILE @@FETCH_STATUS = 0
     BEGIN
          INSERT INTO OrderItems(OrderId, ProductId,
OrderedQuantity)
               VALUES (@OrderId, @ProductId, @OrderdQuantity);

          FETCH NEXT FROM ItemsCursor INTO @ProductId,
@OrderedQuantity;
     END;

     CLOSE ItemsCursor;
     DEALLOCATE ItemsCursor;
     FETCH NEXT FROM OrdersCursor INTO @DeliveryDate, @CustomerId,
@Discount;
```

```
END;

CLOSE OrdersCursor;
DEALLOCATE OrdersCursor;

UPDATE Products SET Quantity = Quantity + @Diff
   WHERE Quantity < 15;

UPDATE Orders SET DeliveryDate = DATEADD(DD, 2, DeliveryDate)
   WHERE DeliveryDate BETWEEN @StartDate AND @EndDate;

COMMIT TRAN;
```

Цей же запит для MongoDB матиме наступний вигляд:

```
session = db.getMongo().startSession();
session.startTransaction();

orderCollection = session.getDatabase('inventory').orders;
productCollection = session.getDatabase('inventory').products;

orderCollection.insertMany([{
   delivery_date: @DeliveryDate,
   discount: @Discount,
   customer: {
     id: @OrderCustomerId,
     email: @OrderCustomerEmail
   },
   items: [
     {
        product_id: @ProductId,
        product_name: @ProductName,
        ordered_quantity: @OrderedQuantity
     },
     ...
   ]
},
...
]);

productCollection.updateMany(
   { quantity: { $lt: 15 } },
   { $inc: { quantity: @Diff } });

orderCollection.updateMany({
   deliveryDate: { $lt: @EndDate, $gt: @StartDate }
}, {
   $set: { deliveryDate: { $add: ["$date", 2 * 24 * 60 * 60000] } }
});
session.commitTransaction();
```

У наступній транзакції буде 4 операції вставки та тільки одна для оновлення. Першими двома операціями вставки є запити створення нових категорій (у тій же кількості, що вже є у базі даних, адже цих категорій зберігається невелика кількість) та товарів для цих категорій (для цих категорій

буде вставлено товарів у кількості 2% від загальної кількості для даного режиму). Іншими операціями вставки є створення нових користувачів (у кількості 50% від загальної кількості користувачів для даного режиму) та замовлень для них (у кількості 2% від загальної кількості замовлень для даного режиму). Операція оновлення полягатиме у зміні знижки для тих замовлень, чия дата доставки належить певному відрізку. Для такої комплексної транзакцій T-SQL код має наступний вигляд:

```
BEGIN TRAN;

DECLARE @CategoryName NVARCHAR(MAX);
DECLARE @CustomerId UNIQUEIDENTIFIER;
DECLARE @Discount FLOAT;

DECLARE CategoryCursor CURSOR FOR SELECT * FROM @CategoriesToInsert;
OPEN CategoryCursor;

FETCH NEXT FROM CategoryCursor INTO @CategoryName;

WHILE @@FETCH_STATUS = 0
BEGIN

      DECLARE @CategoryId INTEGER;
      DECLARE @ProductName NVARCHAR(MAX);
      DECLARE @ProductDescription NVARCHAR(MAX);
      DECLARE @ProductQuantity INTEGER;
      DECKARE @ProductPrice DECIMAL;

      INSERT INTO Categories (Name)
            OUTPUT INSERTED.Id INTO @CategoryId
            VALUES(@CategoryName);

      DECLARE ProductCursor CURSOR FOR SELECT * FROM @ProductsToInsert
      WHERE CategoryIndex = @@CursorRows;
      OPEN ProductCursor;

      FETCH NEXT FROM ProductCursor INTO @ProductName,
            @ProductDescription, @ProductQuantity, @ProductPrice;

      WHILE @@FETCH_STATUS = 0
      BEGIN
            INSERT INTO Products(Name, Description, Quantity, Price,
      CategoryId)
                  VALUES (@ProductName, @ProductDescription,
@ProductQuantity, @ProductPrice, @CategoryId);

            FETCH NEXT FROM ProductCursor INTO @ProductName,
                  @ProductDescription, @ProductQuantity, @ProductPrice;
      END;

      CLOSE ProductCursor;
      DEALLOCATE ProductCursor;
```

```
		FETCH NEXT FROM CategoryCursor INTO @CategoryName;
	END;

	CLOSE CategoryCursor;
	DEALLOCATE CategoryCursor;

	DECLARE @FirstName NVARCHAR(MAX);
	DECLARE @LastName NVARCHAR(MAX);
	DECLARE @Email NVARCHAR(MAX);
	DECLARE @PhoneNumber NVARCHAR(MAX);
	DECLARE @Address NVARCHAR(MAX);

	DECLARE CustomerCursor CURSOR FOR SELECT * FROM @CustomersToInsert;
	OPEN CustomerCursor;

	FETCH NEXT FROM CustomerCursor INTO @FirstName, @LastName, @Email,
@PhoneNumber, @Address;

	WHILE @@FETCH_STATUS = 0
	BEGIN
		DECLARE @CustomerId INTEGER;

		INSERT INTO Customers (FirstName, LastName, Email, PhoneNumber,
	Address)
			OUTPUT INSERTED.Id INTO @CustomerId
			VALUES(@FirstName, @LastName, @Email, @PhoneNumber,
	@Address);

	DECLARE @DeliveryDate DATE;
	DECLARE @Discount FLOAT;

	DECLARE OrdersCursor CURSOR FOR SELECT * FROM @OrdersToInsert;
	OPEN OrdersCursor;

	FETCH NEXT FROM OrdersCursor INTO @DeliveryDate, @Discount;

	WHILE @@FETCH_STATUS = 0
	BEGIN

		DECLARE @OrderId INTEGER;
		DECLARE @ProductId UNIQUEIDENTIFIER;
		DECLARE @OrderedQuantity INTEGER;

		INSERT INTO Orders (DeliveryDate, CustomerId, Discount)
			OUTPUT INSERTED.Id INTO @OrderId
			VALUES(@DeliveryDate, @CustomerId, @Discount);

		DECLARE ItemsCursor CURSOR FOR
			SELECT * FROM @ItemsToInsert WHERE OrderIndex =
	@@CursorRows;
		OPEN ItemsCursor;

		FETCH NEXT FROM ItemsCursor INTO @ProductId, @OrderedQuantity;

		WHILE @@FETCH_STATUS = 0
		BEGIN
```

```
           INSERT      INTO      OrderItems(OrderId,      ProductId,
OrderedQuantity)
                VALUES (@OrderId, @ProductId, @OrderdQuantity);

           FETCH    NEXT    FROM    ItemsCursor    INTO    @ProductId,
@OrderedQuantity;
     END;

     CLOSE ItemsCursor;
     DEALLOCATE ItemsCursor;
     FETCH NEXT FROM OrdersCursor INTO @DeliveryDate, @CustomerId,
@Discount;
     END;

     CLOSE OrdersCursor;
     DEALLOCATE OrdersCursor;

END;

CLOSE CustomerCursor;
DEALLOCATE CustomerCursor;

UPDATE Orders SET Discount = Discount + @DiscountDiff
   WHERE DeliveryDate BETWEEN @StartDate AND @EndDate;

COMMIT TRAN;
```

Нижче наведено запит для MongoDB:

```
session = db.getMongo().startSession();
session.startTransaction();

customerCollection = session.getDatabase('inventory').customers;
categoryCollection = session.getDatabase('inventory').categories;
orderCollection = session.getDatabase('inventory').orders;
productCollection = session.getDatabase('inventory').products;

categoryCollection.insertMany([{
   name: @CategoryName
},
...
]);

productCollection.insertMany([{
   name: @ProductName,
   description: @ProductDescription,
   quantity: @ProductDescription,
   price: @ProductPrice
   category: @CategoryId
},
...
]);

customerCollection.insertMany([{
   first_name: @CustomerFirstName,
   last_name: @CustomerLastName,
   email: @CustomerEmail,
```

```
    phone_number: @CustomerPhoneNumber,
    address: @CustomerAddress
},
...
]);

orderCollection.insertMany([{
    delivery_date: @DeliveryDate,
    discount: @Discount,
    customer: {
        id: @OrderCustomerId,
        email: @OrderCustomerEmail
    },
    items: [
        {
            product_id: @ProductId,
            product_name: @ProductName,
            ordered_quantity: @OrderedQuantity
        },
        ...
    ]
},
...
]);

orderCollection.updateMany({
    deliveryDate: { $lt: @EndDate, $gt: @StartDate }
}, {
    $set: { discount: { $add: @DiscountDiff } }
});

session.commitTransaction();
```

Таким чином були спроектовані та написані на мовах T-SQL та форматі BSON запити, що будуть використовуватися для замірів.

## 3.3 Опис програмного інтерфейсу

Останнім кроком у виконанні бізнес-логіки додатку є аналіз та вивід результатів. Для цих цілей можуть використовуватися декілька інтерфейсів, наприклад, консольний інтерфейс та веб-сторінка. На цих інтерфейсах виведено вже оброблені значення, а саме середній час виконання запиту, розмір бази даних, максимально затребуваний процесорний час у відсотках та максимально потрібна

оперативна пам’ять. Нижче наведено результат виводу для консольного інтерфейсу (див. рис. 3.1).

```
D:\ХНУРЭ\Диплом магистратура\DatabaseMetrics\DatabaseMetrics.Mongo.Console\bin\Release\netcoreapp3.1\DatabaseMetrics.Mongo.Console.exe
MongoDB shell version v4.4.3
---------------------------------------------------FullBasic---------------------------------------------------
Size bytes: 314644389
connecting to: mongodb://127.0.0.1:27018/admin?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("7632a4eb-810e-46ad-af5c-bb5919f071a0") }
MongoDB server version: 4.4.3
{
        "ok" : 1,
        "$clusterTime" : {
                "clusterTime" : Timestamp(1613068760, 1),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        },
        "operationTime" : Timestamp(1613068760, 1)
}
Ram bytes: 9531392
CPU percents (%): 0
MONGO - Get top 3 categories that are sold for next 30 days: 3776900 ticks
MONGO - Get order by id : 179846 ticks
Create a new order with 30 item(s): 1330876 ticks
Reduce products with quantity less than 10 by 10%: 8651757 ticks
Delete all orders which sum price is in some range: 158678210 ticks
Insert N orders with different items count. Update quantity of K products. Update orders date.: 4712819 ticks
Insert K new categories and N products for them. Insert L customers and orders for them. Update discount for orders.: 41
53048 ticks
MongoDB shell version v4.4.3
MongoDB shell version v4.4.3
connecting to: mongodb://127.0.0.1:27018/admin?compressors=disabled&gssapiServiceName=mongodb
```

Рисунок 3.1 – Вивід результатів для консольного інтерфейсу

На базі розробленого програмного забезпечення будуть робитися заміри щодо продуктивності роботи обраних методів.

# 4 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ МЕТОДІВ ПІДТРИМКИ ТРАНЗАКЦІЙ

## 4.1 Дослідження для MongoDB

Після виконання ПЗ та експерименту для СУБД MongoDB було отримано ряд результатів замірів. Почати доречно із замірів, де змінюється лише фактор кількості даних, а фактори кількості одночасних підключень та кількості реплік залишаються статичними та мають стандартний режим Basic.

Нижче наведена таблиця з метриками розміру бази даних, процесорного часу та оперативної пам’яті, що були необхідні для виконання запитів (див. табл. 6).

Таблиця 6 – Нечасові метрики при змінній кількості даних для MongoDB

| | Розмір бази даних (МБ) | Максимальний процесорний час (%) | Максимальна кількість ОП (байти) |
|---|---|---|---|
| Basic | 308 | 1.1 | 95313923 |
| Basic+ | 341 | 3.5 | 100474884 |
| Basic++ | 394 | 3.7 | 94044162 |
| Medium | 421 | 7.1 | 101621766 |
| Intensive | 1708 | 7.2 | 103424003 |

З таблиці досить легко побачити, що всі величини зростають при збільшенні кількості даних, а тому і є залежними від цього фактору.

Далі слід навести середній час виконання запитів. І спочатку наведено цей час для запитів зі зчитуванням даних. Таких запитів всього два і час по ним наведений у таблиці нижче (див. табл. 7). Час наведено в мілісекундах з округленням до 2 знаків .

Таблиця 7 – Час виконання запитів зчитування при змінній кількості даних для MongoDB

| | Час отримання 3-ьох найпопулярніших категорій (мс) | Час отримання замовлення за ідентифікатором (мс) |
|---|---|---|
| Basic | 377.69 | 14.98 |
| Basic+ | 1992.34 | 15.4 |
| Basic++ | 3036.25 | 14.73 |
| Medium | 4367.34 | 18.46 |
| Intensive | 12577.61 | 15.32 |

Із вищезазначеної таблиці можна побачити, що зі збільшенням даних час отримання найпопулярніших категорій досить помітно зростає. У той же час отримання замовлення за ідентифікатором здається, що не насправді не сильно залежить від кількості даних, а різниця більш зумовлена простою похибкою.

Далі вже йдуть транзакційні запити, а саме запити з операцією одного типу: вставка, оновлення або видалення. Всього таких запитів три і їх час виконання наведено в таблиці нижче (див. табл. 8).

Таблиця 8 – Час виконання однотипних транзакцій при змінній кількості даних для MongoDB

| | Час вставки замовлення зі змінною кількістю елементів (мс) | Зменшення ціни на продукти, кількість яких менша за певний поріг (мс) | Час видалення замовлень, сумарна ціна яких менша за певну суму (мс) |
|---|---|---|---|
| Basic | 133.09 | 865.18 | 15867.82 |
| Basic+ | 86.76 | 1141.03 | 75958.16 |
| Basic++ | 100.05 | 1101.22 | 167801.26 |
| Medium | 91.39 | 1220.66 | 330337.52 |
| Intensive | 121.12 | 2741.36 | 1083499.95 |

Аналізуючи заміри можна зробити висновок, що на операцію вставки фактор кількості даних не впливає, а різниця між режимами також пояснюється випадковою похибкою. У той же час для операції оновлення можна побачити зростання часу виконання, хоча й не дуже стрімке зростання: спочатку на декілька сотень мілісекунд, а потім більш ніж у 2 рази, тобто все ж у цієї операції нелінійне зростання. І в кінці кінців для операції видалення кількість даних грає велику роль, адже зростання часу виконання відбувається від 2 до 5 разів.

Останніми замірами при змінній кількості даних є заміри часу виконання комплексних транзакцій з різними типами операцій. Всього таких транзакційних запитів було спроектовано два. Час виконання цих транзакцій наведено в таблиці нижче (див. табл. 9).

Таблиця 9 – Час виконання комплексних транзакцій при змінній кількості даних для MongoDB

| | Транзакція №1 (мс) | Транзакція №2 (мс) |
|---|---|---|
| Basic | 332.38 | 415.3 |
| Basic+ | 297.46 | 424.65 |
| Basic++ | 386.05 | 780.19 |
| Medium | 525.11 | 738.96 |
| Intensive | 1024.07 | 2213.16 |

Як можна побачити, то для першої транзакції, де більшість запитів на вставку спочатку для режимів типу Basic зростання майже немає та навіть немає чіткого росту, але вже для наступних режимів, час зростає в 1.5 та 2 рази, тому зі збільшенням кількості даних час транзакції буде досить швидко зростати. Для другої транзакції теж характерне зростання часу виконання, але для перших чотирьох режимів він складає декілька сотень мілісекунд, а для режиму Intensive приблизно в 3 рази. Тому і для цієї транзакції видна чітка залежність часу виконання від кількості даних.

Наступним кроком є наведення замірів при змінній кількості одночасних підключень, коли інші фактори знаходитимуться в режимі Basic. Нижче наведені нечасові метрики, окрім розміру бази даних (див. табл. 10).

Таблиця 10 – Нечасові метрики при змінній кількості підключень для MongoDB

| | Максимальний процесорний час (%) | Максимальна кількість ОП (байти) |
|---|---|---|
| Basic | 1.1 | 87203841 |
| Basic+ | 2.5 | 87408645 |
| Basic++ | 3.6 | 90521605 |
| Medium | 3.5 | 90849289 |
| Intensive | 3.7 | 95641603 |

Аналізуючи наведені дані можна легко виявити, що фактор кількості підключень корелюється із CPU часом та кількістю оперативної пам’яті. Але кількість CPU часу зростає не так швидко як при збільшенні об’єму даних. А кількість затребуваної оперативної пам’яті зростає приблизно з тією ж швидкістю.

Далі в таблиці наведений час для запитів зчитування (див. табл. 11).

Таблиця 11 – Час виконання запитів зчитування при змінній кількості підключень для MongoDB

| | Час отримання 3-ьох найпопулярніших категорій (мс) | Час отримання замовлення за ідентифікатором (мс) |
|---|---|---|
| Basic | 406.5 | 15.06 |
| Basic+ | 773.03 | 17.87 |
| Basic++ | 1157.26 | 17.48 |
| Medium | 2127.15 | 21.18 |
| Intensive | 5734.47 | 25.53 |

Для запитів зчитування як і випадку змінної кількості даних одразу видно пропорційну залежність. Хоча й час отримання найпопулярніших категорій досить швидко зростає, але все ж в декілька разів повільніше, ніж у випадку збільшення даних. З іншої сторони, при збільшенні кількості підключень помітний також ріст часу отримання замовлення за ідентифікатором, хоча й невеликий.

Далі наведено значення часу виконання однотипних запитів на вставку, оновлення та видалення даних. Для операцій вставки для перших режимах тренд не дуже зрозумілий, але зі значним збільшенням підключень час запиту все ж росте в більш, ніж 2 рази. Для операції оновлення помітний тренд збільшення часу виконання при збільшенні кількості підключень, при чому цей фактор впливає на цю операцію сильніше, ніж фактор кількості даних. А на відміну від оновлення час виконання операції видалення зростає з кожним режимом, але повільніше, ніж при рості даних (див. табл. 12).

Таблиця 12 – Час виконання однотипних транзакцій при змінній кількості підключень для MongoDB

| | Час вставки замовлення зі змінною кількістю елементів (мс) | Зменшення ціни на продукти, кількість яких менша за певний поріг (мс) | Час видалення замовлень, сумарна ціна яких менша за певну суму (мс) |
|---|---|---|---|
| Basic | 127.5 | 958.81 | 15699.24 |
| Basic+ | 152.16 | 1527.61 | 33606.71 |
| Basic++ | 151.54 | 2341.4 | 55363.79 |
| Medium | 139.51 | 2649.51 | 105683.58 |
| Intensive | 325.16 | 30385.01 | 357715.86 |

І останнім кроком для даного фактору є наведення часу виконання комплексних транзакцій (див. табл. 13).

Таблиця 13 – Час виконання комплексних транзакцій при змінній кількості підключень для MongoDB

| | Транзакція №1 (мс) | Транзакція №2 (мс) |
|---|---|---|
| Basic | 332.38 | 225.4 |
| Basic+ | 342.62 | 1054.83 |
| Basic++ | 414.36 | 591.82 |
| Medium | 795.48 | 1126.16 |
| Intensive | 2935.62 | 938.33 |

Для обох транзакцій склалась різна ситуація: якщо для першої видно зростання часу при збільшенні підключень, то для другої час то збільшується, то зменшується з кожним наступним режимом, що свідчить про відсутність тренду.

Далі будуть наведені заміри, коли змінною буде кількість реплік у кластері, а інші фактори залишаться статичними в режимі Basic. Необхідно нагадати, що для даного фактору на відміну від інших було спроектовано всього 4 режими. Нижче наведені заміри нечасових метрів при змінній кількості вузлів у кластері бази (див. табл. 14).

Таблиця 14 – Нечасові метрики при змінній кількості вузлів для MongoDB

| | Максимальний процесорний час (%) | Максимальна кількість ОП (байти) |
|---|---|---|
| Basic | 1.1 | 96460807 |
| Basic+ | 1.2 | 96911365 |
| Medium | 1.2 | 99860487 |
| Intensive | 1.1 | 99983362 |

Одразу можна помітити, що процесорний час ніяк не корелює зі збільшенням кількості вузлів, та приймає значення на рівні статистичної похибки. А у випадку затребуваної оперативної пам’яті є дуже повільний ріст, а саме

споживання трохи більше, ніж у випадку змінної кількості підключень та трохи менше ніж у випадку змінної кількості даних. Далі наведено час виконання операцій зчитування в залежності від заданого фактору кількості вузлів (див. табл. 15).

Таблиця 15 – Час виконання запитів зчитування при змінній кількості вузлів для MongoDB

| | Час отримання 3-ьох найпопулярніших категорій (мс) | Час отримання замовлення за ідентифікатором (мс) |
|---|---|---|
| Basic | 378.62 | 12.16 |
| Basic+ | 373.38 | 12.19 |
| Medium | 384.15 | 12.07 |
| Intensive | 381.98 | 16.57 |

Для обох запитів можна побачити майже повну відсутність кореляції, тобто на час виконання запитів зчитування не впливає кількість вузлів у кластері. Це можна пояснити тим фактом, що ці запити не потребують транзакційності, а дані беруться з першого найближчого вузла.

У наступній таблиці наведено заміри часу виконання однотипних транзакцій (див. табл. 16).

Таблиця 16 – Час виконання однотипних транзакцій при змінній кількості вузлів для MongoDB

| | Час вставки замовлення зі змінною кількістю елементів (мс) | Зменшення ціни на продукти, кількість яких менша за певний поріг (мс) | Час видалення замовлень, сумарна ціна яких менша за певну суму (мс) |
|---|---|---|---|
| Basic | 122.33 | 888.23 | 15674.11 |
| Basic+ | 146.83 | 1043.86 | 15931.15 |

Кінець таблиці 16

| | Час вставки замовлення зі змінною кількістю елементів (мс) | Зменшення ціни на продукти, кількість яких менша за певний поріг (мс) | Час видалення замовлень, сумарна ціна яких менша за певну суму (мс) |
|---|---|---|---|
| Medium | 178.45 | 1614.6 | 16061.42 |
| Intensive | 239.62 | 2349.92 | 16513.8 |

Вже для транзакційних запитів, що потребують синхронізації на всіх вузлах, можна побачити характерне зростання часу заміру, хоча й зростання не є стрімким – геометричним. Але з кожним наступним режимом розрив часу виконання з попереднім режимом збільшується.

І останнім кроком для цього фактору є наведення часу виконання комплексних транзакцій (див. табл. 17).

Таблиця 17 – Час виконання комплексних транзакцій при змінній кількості підключень для MongoDB

| | Транзакція №1 (мс) | Транзакція №2 (мс) |
|---|---|---|
| Basic | 479.37 | 317.36 |
| Basic+ | 752.68 | 601.49 |
| Medium | 1080.68 | 785.13 |
| Intensive | 1801.23 | 1222.05 |

Для комплексних транзакцій також характерне зростання часу виконання зі збільшенням кількості реплік і хоча спочатку час не дуже сильно зростає, але з кожним режимом розрив збільшується. Тобто час виконання цих транзакцій точно є залежним від даного фактору.

Після наведення замірів при змінному одному факторі та статичних інших двох також доречно було б дослідити, як одночасне збільшення кожного фактору

впливатиме на метрики. Адже це одночасне збільшення може дати цікаві результати щодо ефективності того чи іншого рішення. Тобто далі будуть йти заміри, де кожен фактор матиме спочатку режим Basic, потім Basic+ і так далі. Оскільки для вузлів використовуються не чотири, а 5 режимів, то для загального режиму Basic++ для цього фактору використовуватиметься значення Basic+.

Нижче наведені значення нечасових метрик при одночасній зміні всіх факторів (див. табл. 18).

Таблиця 18 – Нечасові метрики при одночасній зміні всіх факторів для MongoDB

| | Максимальний процесорний час (%) | Максимальна кількість ОП (байти) |
|---|---|---|
| Basic | 1.1 | 95313927 |
| Basic+ | 1.9 | 98304002 |
| Basic++ | 6.3 | 98730569 |
| Medium | 10.2 | 100720643 |
| Intensive | 16.5 | 7287603202 |

При одночасному зростанні всіх факторів одразу можна побачити стрімкий ріст використаного процесорного часу. Кількість споживаної оперативної пам’яті також помітно зростає та приймає високе значення для максимального режиму.

Далі наведено заміри для запитів зчитування даних (див табл. 19).

Таблиця 19 – Час виконання запитів зчитування при одночасній зміні всіх факторів для MongoDB

| | Час отримання 3-ьох найпопулярніших категорій (мс) | Час отримання замовлення за ідентифікатором (мс) |
|---|---|---|
| Basic | 377.69 | 14.98 |
| Basic+ | 4384.8 | 15.22 |

Кінець таблиці 19

| | Час отримання 3-ьох найпопулярніших категорій (мс) | Час отримання замовлення за ідентифікатором (мс) |
|---|---|---|
| Basic++ | 10723.34 | 17.3 |
| Medium | 30272.29 | 18.98 |
| Intensive | 80733.55 | 41.32 |

Для обох запитів зчитування помітний різкий ріст. Особливо для отримання популярних категорій ріст в рази швидший, ніж при будь-якій іншій конфігурації, що замірялася до цього. Також досить швидко зростає отримання замовлення через індекс, не дивлячись на те, що ця операція має логарифмічну складність.

Далі наведено заміри для однотипних транзакцій (див. табл. 20).

Таблиця 20 – Час виконання однотипних транзакцій при одночасній зміні всіх факторів для MongoDB

| | Час вставки замовлення зі змінною кількістю елементів (мс) | Зменшення ціни на продукти, кількість яких менша за певний поріг (мс) | Час видалення замовлень, сумарна ціна яких менша за певну суму (мс) |
|---|---|---|---|
| Basic | 133.09 | 865.18 | 15867.82 |
| Basic+ | 173.14 | 3193.3 | 165536.76 |
| Basic++ | 233.4 | 5511.42 | 609625.27 |
| Medium | 267.91 | 20374.17 | 2334308.82 |
| Intensive | 1012.74 | 54177.78 | 121576543.4 |

Для однотипних запитів одразу можна побачити динамічний ріст для кожної транзакції, особливо для операції видалення, де для майже кожного наступного режиму час виконання збільшується на порядок відносно попереднього режиму.

І останнім кроком є наведення зміни часу виконання комплексних транзакцій в залежності від зміни всіх факторів (див. табл. 21).

Таблиця 21 – Час виконання комплексних транзакцій при одночасній зміні всіх факторів для MongoDB

| | Транзакція №1 (мс) | Транзакція №2 (мс) |
|---|---|---|
| Basic | 332.38 | 415.3 |
| Basic+ | 1662.03 | 1348.81 |
| Basic++ | 1791.04 | 1660.43 |
| Medium | 2173.09 | 6163.25 |
| Intensive | 16980.45 | 25183.41 |

Як і очікувалося, час виконання обох транзакцій дуже помітно зростає, навість на декілька порядків. Тобто, одночасне збільшення кожного фактору сильно впливає на транзакцію кожного типу.

У результаті були наведені заміри обраних для MongoDB для різноманітних конфігурація. Надалі ці дані будуть використані для аналізу та порівняння з ефективністю інших методів реалізації розподілених транзакцій.

## 4.2 Дослідження для VoltDB

Далі за допомогою розробленого програмного забезпечення були отримані заміри щодо ефективності виконання запитів та транзакцій для VoltDB. Всі заміри проводилися в тих самих умовах, що й для попереднього методу, з тими самими запитами та режимами навантаження.

Нижче наведена таблиця з метриками розміру бази даних, процесорного часу у відсотках та оперативної пам’яті, що були необхідні для виконання запитів для VoltDB (див. табл. 22).

Таблиця 22 – Нечасові метрики при змінній кількості даних для VoltDB

| | Розмір бази даних (МБ) | Максимальний процесорний час (%) | Максимальна кількість ОП (байти) |
|---|---|---|---|
| Basic | 221 | 1 | 228921756 |
| Basic+ | 268 | 1.1 | 243224303 |
| Basic++ | 319 | 1.3 | 285491652 |
| Medium | 356 | 1.4 | 346281738 |
| Intensive | 1360 | 2 | 587658901 |

Для даного метода також характерний ріст всі нечасових метрик зі збільшенням об’єму даних, але при цьому помітне низьке споживання процесорного часу та досить велике споживання оперативної пам’яті.

Далі наведено середній час виконання запитів на зчитування даних (див. табл. 23).

Таблиця 23 – Час виконання запитів зчитування при змінній кількості даних для VoltDB

| | Час отримання 3-ьох найпопулярніших категорій (мс) | Час отримання замовлення за ідентифікатором (мс) |
|---|---|---|
| Basic | 3936.31 | 128.87 |
| Basic+ | 18953.79 | 123.27 |
| Basic++ | 43030.67 | 126.3 |
| Medium | 81332.71 | 125.63 |
| Intensive | 294374.23 | 129.73 |

Як і для попереднього метода швидкість пошуку замовлення за ідентифікатором не залежить від об’єму даних, у той час як для отримання найпопулярніших категорій видно ріст у декілька разів при переході між режимами.

Далі наведено час виконання однотипних транзакцій вставки, оновлення та видалення даних (див. табл. 24).

Таблиця 24 – Час виконання однотипних транзакцій при змінній кількості даних для VoltDB

| | Час вставки замовлення зі змінною кількістю елементів (мс) | Зменшення ціни на продукти, кількість яких менша за певний поріг (мс) | Час видалення замовлень, сумарна ціна яких менша за певну суму (мс) |
|---|---|---|---|
| Basic | 537.14 | 125.5 | 1850.01 |
| Basic+ | 803.87 | 137.61 | 9354.58 |
| Basic++ | 943 | 153.92 | 21001.97 |
| Medium | 1205.16 | 186.43 | 38275.32 |
| Intensive | 1883.38 | 203.19 | 54321.94 |

Для всіх наведених транзакцій час виконання залежить від об’єму даних у базі. При цьому помітне мінімальне зростання часу для запиту оновлення, та досить велике зростання для запиту видалення: від 1.5 до 5 разів для різних режимів. Час виконання для транзакції вставки також зростає, але скоріше в арифметичній прогресії, ніж геометричній.

Наступними і останніми замірами при змінній кількості даних є заміри часу виконання двох комплексних транзакцій з багатьма операціями вставки та оновлення даних (див. табл. 25).

Таблиця 25 – Час виконання комплексних транзакцій при змінній кількості даних для VoltDB

| | Транзакція №1 (мс) | Транзакція №2 (мс) |
|---|---|---|
| Basic | 1654.39 | 3774 |
| Basic+ | 2117.78 | 8661.2 |
| Basic++ | 2404.56 | 10307.58 |
| Medium | 3632.14 | 17030.16 |
| Intensive | 8363.25 | 40693.78 |

Для обох комплексних транзакцій характерне зростання часу зі збільшенням об’єму даних. Але у випадку першої транзакції з рівною кількістю операцій вставки та оновлення час зростає не так стрімко, як у випадку другої комплексної транзакції, де швидкість росту часу є досить високою.

Далі будуть наведені результати замірів для VoltDB при змінній кількості одночасних підключень. І нижче одразу наведені значення кількості процесорного часу та оперативної пам’яті для кожного з режимів (див. табл. 26).

Таблиця 26 – Нечасові метрики при змінній кількості підключень для VoltDB

| | Максимальний процесорний час (%) | Максимальна кількість ОП (байти) |
|---|---|---|
| Basic | 1 | 228921756 |
| Basic+ | 1.4 | 293569618 |
| Basic++ | 2 | 380104605 |
| Medium | 4.2 | 618475292 |
| Intensive | 10 | 1193602908 |

З таблиці вище видно, що при змінній кількості підключень наявний стрімки ріст споживання процесорного часу. Різниця між найнижчим та

найвищим рівнями сягає 10 разів, що свідчить про пряму залежність споживання процесорного часу від кількості одночасних підключень. Кількість споживаної оперативної пам'яті також зростає, при чому швидкість росту часу також зростає, хоча й досить повільно.

Наступним кроком є наведення замірів для запитів зчитування при змінній кількості одночасних підключень (див. табл. 27).

Таблиця 27 – Час виконання запитів зчитування при змінній кількості підключень для VoltDB

| | Час отримання 3-ьох найпопулярніших категорій (мс) | Час отримання замовлення за ідентифікатором (мс) |
|---|---|---|
| Basic | 4027.64 | 123.68 |
| Basic+ | 9651.07 | 146.38 |
| Basic++ | 14138.01 | 179.38 |
| Medium | 25476.42 | 187.07 |
| Intensive | 52859.35 | 5993.58 |

Із замірів можна одразу побачити, що на відміну від об'єму даних час отримання замовлення за ідентифікатором помітно залежить від кількості одночасних підключень. При цьому для найвищого режиму різниця з попереднім у часі сягає аж 1 порядок. Запит отримання категорій товарів також залежить у часі від кількості підключень і при зміні режиму статично зростає приблизно в 2 рази.

Нижче наведені заміри часу виконання однотипних транзакцій на вставку, оновлення та видалення даних (див. табл. 28). Час операції вставки в такому випадку спочатку зростає дуже повільно, і тільки на максимальному режимі є значний скачок у часі в 4 рази. Для операції оновлення характерний більш статичний ріст часу приблизно у два рази на кожному режимі. Для транзакції видалення також помітний ріст часу у декілька разів при зміні режимів.

Таблиця 28 – Час виконання однотипних транзакцій при змінній кількості підключень для VoltDB

| | Час вставки замовлення зі змінною кількістю елементів (мс) | Зменшення ціни на продукти, кількість яких менша за певний поріг (мс) | Час видалення замовлень, сумарна ціна яких менша за певну суму (мс) |
|---|---|---|---|
| Basic | 537.01 | 125.19 | 1886.83 |
| Basic+ | 578.51 | 206.43 | 5643.46 |
| Basic++ | 622.65 | 263.63 | 9228.94 |
| Medium | 598.18 | 460.55 | 17304.94 |
| Intensive | 2122.96 | 804.29 | 56622.07 |

І останнім кроком для даного фактору є наведення часу виконання комплексних транзакцій (див. табл. 29).

Таблиця 29 – Час виконання комплексних транзакцій при змінній кількості підключень для VoltDB

| | Транзакція №1 (мс) | Транзакція №2 (мс) |
|---|---|---|
| Basic | 1592.37 | 3194.82 |
| Basic+ | 4215.1 | 5980.61 |
| Basic++ | 7294.21 | 8496.66 |
| Medium | 15636.51 | 14616.95 |
| Intensive | 46122.82 | 42720.91 |

Час обох транзакцій зростає по мірі збільшення кількості одночасних підключень. При цьому час для першої комплексної транзакції спочатку менший за час другої, але його ріст швидший, тому перша транзакцій на останніх режимах виконується навіть довше за другу.

Далі будуть наведені заміри при змінній кількості реплік у кластері бази даних. Першими наведені значення нечасових метрик, а саме процесорного часу та кількості оперативної пам’яті (див. табл. 30).

Таблиця 30 – Нечасові метрики при змінній кількості вузлів для VoltDB

| | Максимальний процесорний час (%) | Максимальна кількість ОП (байти) |
|---|---|---|
| Basic | 1 | 228921756 |
| Basic+ | 1.1 | 234190632 |
| Medium | 1.1 | 231822746 |
| Intensive | 1.2 | 237182904 |

Одразу можна помітити, що для процесорного часу важко визначити наявність кореляції з кількістю реплік, адже різниця між найменшим та найвищим рівнями складає всього 0.2 %. Для оперативної пам’яті скоріш за все кореляція відсутня, адже при підвищенні режиму значення то йдуть вгору, то спадають.

Наступним кроком є наведення часу виконання запитів зчитування при змінній кількості вузлів-реплік (див. табл. 31).

Таблиця 31 – Час виконання запитів зчитування при змінній кількості вузлів для VoltDB

| | Час отримання 3-ьох найпопулярніших категорій (мс) | Час отримання замовлення за ідентифікатором (мс) |
|---|---|---|
| Basic | 4027.64 | 123.68 |
| Basic+ | 3861.71 | 124.83 |
| Medium | 3328.6 | 123.37 |
| Intensive | 3763.21 | 125.01 |

Як і очікувалося, для обох запитів зчитування відсутня кореляція з кількістю реплік, адже дані операції не потребують роботи з усіма вузлами кластеру. А тому різниця в часі пояснюється простою статистичною похибкою замірів.

Далі наведено час виконання однотипних транзакцій при зміні кількості вузлів-реплік у кластері (див. табл. 32).

Таблиця 32 – Час виконання однотипних транзакцій при змінній кількості вузлів для VoltDB

| | Час вставки замовлення зі змінною кількістю елементів (мс) | Зменшення ціни на продукти, кількість яких менша за певний поріг (мс) | Час видалення замовлень, сумарна ціна яких менша за певну суму (мс) |
|---|---|---|---|
| Basic | 847.57 | 125.19 | 1886.83 |
| Basic+ | 847.55 | 120.31 | 1994.64 |
| Medium | 771.9 | 126.19 | 1983.99 |
| Intensive | 798.11 | 125.34 | 2004.13 |

Для транзакційних запитів також майже не помітна різниця в часі між режимами при збільшенні кластеру. Для транзакцій вставки та оновлення числа часу то зростають, то спадають, що свідчить про відсутність кореляції з кількістю реплік. Для транзакції видалення є послідовний ріст, але він дуже слабкий і складає близько 120 мс, що є незначним результатом, коли час самого виконання вимірюється в секундах.

Наступним і останнім кроком для даного фактору є наведення часу виконання комплексних транзакцій (див. табл. 33).

Таблиця 33 – Час виконання комплексних транзакцій при змінній кількості підключень для VoltDB

| | Транзакція №1 (мс) | Транзакція №2 (мс) |
|---|---|---|
| Basic | 1592.37 | 3194.82 |
| Basic+ | 1883.75 | 3593.48 |
| Medium | 1942.05 | 3343.35 |
| Intensive | 2285.46 | 4492.47 |

Вже для комплексних транзакцій з багатьма операціями на запис стає помітною залежність часу виконання від кількості реплік, адже всі ці зміни потрібно зафіксувати на всіх вузлах. Але хоча ріст і помітний, тим не менш його прогресія є арифметичною, а не геометричною.

Після отримання метрик при зміні тільки одного фактору необхідно розглянути ситуацію, коли змінюються всі фактори навантаження одночасно. Це дасть змогу оцінити продуктивність того чи іншого підходу, коли заздалегідь не зрозуміло, який саме фактор переважатиме для конкретного додатку.

Нижче наведені значення нечасових метрик при одночасній зміні всіх факторів (див. табл. 34).

Таблиця 34 – Нечасові метрики при одночасній зміні всіх факторів для VoltDB

| | Максимальний процесорний час (%) | Максимальна кількість ОП (байти) |
|---|---|---|
| Basic | 1 | 228921756 |
| Basic+ | 1.4 | 337321045 |
| Basic++ | 2.2 | 460380542 |
| Medium | 4.7 | 725801783 |
| Intensive | 13 | 1435729031 |

Для такого режиму споживання процесорного часу досить стрімко зростає, хоча й на піковому навантаженні і є меншим за час попереднього методу. Кількість споживаної оперативної пам’яті також зростає приблизно в 1.5-2 рази та на максимальному режимі сягає майже 1.5 Гб.

Наступним кроком є наведення часу виконання запитів на зчитування даних при зміні всіх факторів одночасно (див. табл. 35).

Таблиця 35 – Час виконання запитів зчитування при одночасній зміні всіх факторів для VoltDB

| | Час отримання 3-ьох найпопулярніших категорій (мс) | Час отримання замовлення за ідентифікатором (мс) |
|---|---|---|
| Basic | 3936.31 | 128.87 |
| Basic+ | 21542.83 | 151.75 |
| Basic++ | 61572.88 | 183.57 |
| Medium | 94824.04 | 197.45 |
| Intensive | 358923.22 | 6451.23 |

Обидва запити дуже характерно зростають, особливо на максимальному режимі навантаження. При цьому запит для отримання категорій зростає між режимами в геометричній прогресії в декілька разів між режимами. З іншого боку час отримання замовлення за ідентифікатором на першим рівнях зростає всього на декілька десятків мілісекунд, але вже при найвищому навантаженні зростає на один порядок. При цьому час виконання обох запитів на зчитування на порядок вище, ніж час виконання цих же операцій для MongoDB.

Далі наведено результати замірів для однотипних транзакцій на вставку, оновлення та видалення (див. табл. 36).

Таблиця 36 – Час виконання однотипних транзакцій при одночасній зміні всіх факторів для VoltDB

| | Час вставки замовлення зі змінною кількістю елементів (мс) | Зменшення ціни на продукти, кількість яких менша за певний поріг (мс) | Час видалення замовлень, сумарна ціна яких менша за певну суму (мс) |
|---|---|---|---|
| Basic | 537.14 | 125.5 | 1850.01 |
| Basic+ | 846.43 | 258.31 | 14190.24 |
| Basic++ | 1017.06 | 435.23 | 39421.29 |
| Medium | 1422.01 | 923.91 | 64325.84 |
| Intensive | 3052.32 | 2593.51 | 100485.72 |

Для всіх трьох однотипних запитів помітний ріст зі збільшенням навантаження. Але в той же час цей ріст досить стабільний і складає приблизно 2 рази для кожної операції, при цьому на найвищих режимах немає ніякої аномалії в часі виконання.

І останнім кроком для даного методу є наведення зміни часу виконання комплексних транзакцій в залежності від одночасної зміни всіх факторів навантаження (див. рис. 37).

Таблиця 37 – Час виконання комплексних транзакцій при одночасній зміні всіх факторів для VoltDB

| | Транзакція №1 (мс) | Транзакція №2 (мс) |
|---|---|---|
| Basic | 1654.39 | 3774 |
| Basic+ | 4943.22 | 9620,03 |
| Basic++ | 8181.02 | 11691.62 |
| Medium | 17930.21 | 21084.46 |
| Intensive | 675549.34 | 613669.34 |

Для обох комплексних транзакцій є помітний ріст, але для першої транзакції його швидкість зростає стрімкіше, адже маючи менший час на перших режимах, ця транзакція має більший, порівняно з другою, на останніх режимах. В цілому для обох цих транзакцій різниця між найвищим та найнижчим рівнями складає декілька порядків.

Таким чином були наведені результати замірів часу виконання запитів і транзакцій та споживані при цьому ресурси для VoltDB. Ці значення будуть використані при порівнянні з іншими підходами.

## 4.3 Дослідження для Two-Phase Commit

Останнім методом дослідження реалізації розподілених транзакцій є реалізація на рівні додатку за допомогою шаблону Two-Phase Commit. Дана метод виконується на класичній реляційні СУБД SQL Server 2017, а також мови програмування C# за допомогою інфраструктури класу TransactionScope, що реалізує зазначений шаблон.

Нижче наведена таблиця з метриками розміру бази даних, процесорного часу у відсотках та оперативної пам’яті, що були необхідні для виконання запитів для Two-Phase Commit підходу (див. табл. 38).

З таблиці одразу можна побачити, що всі три нечасові метрики мають залежність від об’єму даних, що зберігаються в базі. Розмір бази даних відрізняється в 5 разів між найнижчим та найвищим рівнями та при максимальному об’ємі даних майже сягає 3 Гб. Процесорний час також збільшується, але не критично, всього в 3 рази. Кількість оперативної пам’яті зростає дуже повільно, в арифметичній прогресії, але все ж її кількість на порядок більша, ніж у випадку з MongoDB.

Таблиця 38 – Нечасові метрики при змінній кількості даних для Two-Phase Commit

| | Розмір бази даних (МБ) | Максимальний процесорний час (%) | Максимальна кількість ОП (байти) |
|---|---|---|---|
| Basic | 592 | 1.7 | 363855872 |
| Basic+ | 656 | 2 | 407896064 |
| Basic++ | 912 | 2.2 | 426770432 |
| Medium | 1168 | 3.1 | 516947968 |
| Intensive | 2896 | 4.2 | 609222656 |

Далі наведено середній час виконання запитів на зчитування даних (див. табл. 39).

Таблиця 39 – Час виконання запитів зчитування при змінній кількості даних для Two-Phase Commit

| | Час отримання 3-ьох найпопулярніших категорій (мс) | Час отримання замовлення за ідентифікатором (мс) |
|---|---|---|
| Basic | 390.42 | 13.05 |
| Basic+ | 1031.89 | 6.43 |
| Basic++ | 1613.87 | 9.54 |
| Medium | 3246.09 | 25.58 |
| Intensive | 11697.18 | 6.75 |

Для запиту отримання замовлення за ідентифікатором час виконання не залежить від об’єму даних, адже йде зчитування за індексом. У той же час є характерний зріст часу отримання категорій товарів, а також час зростає у геометричній прогресії.

Далі наведено час виконання однотипних транзакцій вставки, оновлення та видалення даних (див. табл. 40).

Таблиця 40 – Час виконання однотипних транзакцій при змінній кількості даних для Two-Phase Commit

| | Час вставки замовлення зі змінною кількістю елементів (мс) | Зменшення ціни на продукти, кількість яких менша за певний поріг (мс) | Час видалення замовлень, сумарна ціна яких менша за певну суму (мс) |
|---|---|---|---|
| Basic | 749.04 | 375.64 | 4575.53 |
| Basic+ | 1127.57 | 744.61 | 8188.54 |
| Basic++ | 1759.04 | 1165.24 | 15087.72 |
| Medium | 2650.62 | 2409.81 | 30208.72 |
| Intensive | 5477.22 | 5025.99 | 96059.97 |

Для всіх наведених транзакцій час виконання залежить від об’єму даних у базі. При чому час вставки зростає не так швидко, як інші методи, а найбільший ріст має час транзакція видалення. Найменший абсолютний час має оновлення транзакція.

Останнім кроком для фактору зміни даних є наведення часу виконання двох комплексних транзакцій з багатьма операціями вставки та оновлення даних (див. табл. 41).

Таблиця 41 – Час виконання комплексних транзакцій при змінній кількості даних для Two-Phase Commit

| | Транзакція №1 (мс) | Транзакція №2 (мс) |
|---|---|---|
| Basic | 1654.39 | 3774 |
| Basic+ | 2117.78 | 8661.2 |
| Basic++ | 2404.56 | 10307.58 |

Кінець таблиці 41

| | Транзакція №1 (мс) | Транзакція №2 (мс) |
|---|---|---|
| Medium | 3632.14 | 17030.16 |
| Intensive | 8363.25 | 40693.78 |

В обох випадках час виконання транзакції залежить від кількості даних в базі, але друга транзакція зростає набагато жвавіше.

Наступним кроком є наведення значень замірів при змінній кількості одночасних підключень. Першими замірами є нечасові метрики процесорного часу та оперативної пам’яті (див. табл. 42).

Таблиця 42 – Нечасові метрики при змінній кількості підключень для Two-Phase Commit

| | Максимальний процесорний час (%) | Максимальна кількість ОП (байти) |
|---|---|---|
| Basic | 1.7 | 363855872 |
| Basic+ | 2.3 | 452984832 |
| Basic++ | 3.8 | 525336576 |
| Medium | 9.2 | 695205888 |
| Intensive | 17 | 1285939456 |

Одразу із замірів видно досить високий ріст процесорного часу, що збільшився між в 10 разів між найвищим та найнижчим режимами. Цей показник досить високий та перевищує відповідні показники для MongoDB та VoltDB. Кількість споживаної оперативної пам’яті також зростає зі збільшенням кількості підключень та на піковому навантаженні сягає більше 1.2 Гб.

Далі наведені заміри для запитів зчитування при змінній кількості одночасних підключень (див. табл. 43).

Таблиця 43 – Час виконання запитів зчитування при змінній кількості підключень для Two-Phase Commit

| | Час отримання 3-ьох найпопулярніших категорій (мс) | Час отримання замовлення за ідентифікатором (мс) |
|---|---|---|
| Basic | 390.42 | 5.01 |
| Basic+ | 2395.75 | 6.45 |
| Basic++ | 2627.16 | 13.64 |
| Medium | 4234.23 | 79.32 |
| Intensive | 9240.13 | 966.57 |

Обидва запити досить помітно залежні від кількості одночасних підключень, при чому для часу отримання категорій ріст більш статичний, а для часу отримання замовлення за ідентифікатором проявляється різке зростання часу на піковому навантаженні.

Нижче наведені заміри часу виконання однотипних транзакцій на вставку, оновлення та видалення даних (див. табл. 44).

Таблиця 44 – Час виконання однотипних транзакцій при змінній кількості підключень для Two-Phase Commit

| | Час вставки замовлення зі змінною кількістю елементів (мс) | Зменшення ціни на продукти, кількість яких менша за певний поріг (мс) | Час видалення замовлень, сумарна ціна яких менша за певну суму (мс) |
|---|---|---|---|
| Basic | 749.04 | 375.64 | 4452.37 |
| Basic+ | 1294.76 | 689.63 | 9374.02 |
| Basic++ | 2001.74 | 2426.05 | 18530.65 |
| Medium | 4263.9 | 8021.28 | 42902.59 |
| Intensive | 8803.93 | 33749.51 | 117820.39 |

З таблиці видно, що час виконання всіх транзакцій досить сильно залежить від кількості підключень, а найменший ріст у транзакції на вставку.

І останнім кроком для даного фактору є наведення часу виконання комплексних транзакцій (див. табл. 45).

Таблиця 45 – Час виконання комплексних транзакцій при змінній кількості підключень для Two-Phase Commit

| | Транзакція №1 (мс) | Транзакція №2 (мс) |
|---|---|---|
| Basic | 3859.25 | 5265.33 |
| Basic+ | 9084.92 | 9940.17 |
| Basic++ | 16381.05 | 18730.47 |
| Medium | 35433.09 | 42672 |
| Intensive | 108501.14 | 131612.74 |

На замірах видно, що обидві комплексні транзакції мають жорстку залежність часу виконання від кількості одночасних підключень.

Наступним кроком є наведення замірів при змінній кількості реплік-вузлів у кластері. Першими наведені значення нечасових метрик (див. табл. 46).

Таблиця 46 – Нечасові метрики при змінній кількості вузлів для Two-Phase Commit

| | Максимальний процесорний час (%) | Максимальна кількість ОП (байти) |
|---|---|---|
| Basic | 1.7 | 363855872 |
| Basic+ | 1.9 | 427819008 |
| Medium | 2.2 | 460324864 |
| Intensive | 2.4 | 487587840 |

Обидві метрики залежні від кількості вузлів у кластері, але ріст часу в обох випадках дуже повільний, а різниця між будь-якими рівнями не більше 2 разів.

Далі наведено значення метрики часу для запитів на зчитування при змінній кількості вузлів-реплік (див. табл. 47).

Таблиця 47 – Час виконання запитів зчитування при змінній кількості вузлів для Two-Phase Commit

| | Час отримання 3-ьох найпопулярніших категорій (мс) | Час отримання замовлення за ідентифікатором (мс) |
|---|---|---|
| Basic | 523.47 | 17.22 |
| Basic+ | 343.41 | 7.72 |
| Medium | 354.67 | 20 |
| Intensive | 432.88 | 19.21 |

Як і випадку інших методів, час зчитування даних не залежить від кількості вузлів у кластері.

Далі наведено час виконання однотипних транзакцій при зміні кількості вузлів-реплік у кластері (див. табл. 48).

Таблиця 48 – Час виконання однотипних транзакцій при змінній кількості вузлів для Two-Phase Commit

| | Час вставки замовлення зі змінною кількістю елементів (мс) | Зменшення ціни на продукти, кількість яких менша за певний поріг (мс) | Час видалення замовлень, сумарна ціна яких менша за певну суму (мс) |
|---|---|---|---|
| Basic | 749.04 | 375.64 | 4363 |
| Basic+ | 973.41 | 459.95 | 7509.03 |
| Medium | 1426.75 | 814.68 | 14753 |
| Intensive | 1703.57 | 1075.91 | 22085.37 |

На відміну від двох інших розглянутих методів всі однотипні транзакції залежать від кількості вузлів реплік у випадку реалізації транзакційності на рівні додатку. Але тим не менш цей ріст не є стрімким і це зростання більше схоже на арифметичну прогресію, ніж геометричну.

Останнім кроком для даного фактору є наведення часу виконання комплексних транзакцій (див. табл. 49).

Таблиця 49 – Час виконання комплексних транзакцій при змінній кількості підключень для Two-Phase Commit

| | Транзакція №1 (мс) | Транзакція №2 (мс) |
|---|---|---|
| Basic | 3897.67 | 5004.18 |
| Basic+ | 5018.38 | 7035.07 |
| Medium | 7207.35 | 10917.59 |
| Intensive | 10948.3 | 17952.1 |

У цьому випадку також обидві комплексні транзакції залежать від кількості реплік у кластері, але ріст часу обчислюється не в разах і не є надто стрімким.

Далі будуть наведені заміри при одночасній зміні всіх факторів навантаження.

Нижче наведені значення нечасових метрик при одночасній зміні всіх факторів (див. табл. 50).

Таблиця 50 – Нечасові метрики при одночасній зміні всіх факторів для Two-Phase Commit

| | Максимальний процесорний час (%) | Максимальна кількість ОП (байти) |
|---|---|---|
| Basic | 1.7 | 363855872 |
| Basic+ | 2.6 | 441450496 |
| Basic++ | 5.9 | 564133888 |

Кінець таблиці 50

| | Максимальний процесорний час (%) | Максимальна кількість ОП (байти) |
|---|---|---|
| Medium | 11 | 765460480 |
| Intensive | 32 | 1514143744 |

При одночасній зміні всі факторів реалізація розподілених транзакцій на рівні додатку потребує дуже великої кількості ресурсів. На піковому навантаженні споживається майже третина процесорних ресурсів та півтора гігабайти оперативної пам’яті, що є вище, ніж для інших методів .

Наступним кроком є наведення часу виконання запитів на зчитування даних при зміні всіх факторів одночасно (див. табл. 51).

Таблиця 51 – Час виконання запитів зчитування при одночасній зміні всіх факторів для Two-Phase Commit

| | Час отримання 3-ьох найпопулярніших категорій (мс) | Час отримання замовлення за ідентифікатором (мс) |
|---|---|---|
| Basic | 411.46 | 19.32 |
| Basic+ | 2709.69 | 20.74 |
| Basic++ | 10221.51 | 49.02 |
| Medium | 20884.01 | 108.17 |
| Intensive | 113856.34 | 1539.32 |

Обидва запити зростають дуже швидко, майже на порядок при переході з одного рівня навантаження на інший. Це означає сума всіх факторів навантаження дає значний ефект на час виконання транзакцій, при їх реалізації на рівні додатку.

Наступним кроком наведені результати замірів для однотипних транзакцій на вставку, оновлення та видалення (див. табл. 52).

Таблиця 52 – Час виконання однотипних транзакцій при одночасній зміні всіх факторів для Two-Phase Commit

| | Час вставки замовлення зі змінною кількістю елементів (мс) | Зменшення ціни на продукти, кількість яких менша за певний поріг (мс) | Час видалення замовлень, сумарна ціна яких менша за певну суму (мс) |
|---|---|---|---|
| Basic | 749.04 | 375.64 | 4436.43 |
| Basic+ | 1973.84 | 976.75 | 12483.97 |
| Basic++ | 2874.02 | 3913.79 | 27050.64 |
| Medium | 6626.03 | 15881.05 | 58362.82 |
| Intensive | 13734.17 | 87716.21 | 149037.78 |

Для всіх трьох однотипних запитів помітний ріст зі збільшенням навантаження. При чому в усіх трьох випадках ріст є геометричним та його швидкість найвища для транзакцій оновлення та видалення даних.

І останнім кроком для даного методу є наведення зміни часу виконання комплексних транзакцій в залежності від одночасної зміни всіх факторів навантаження (див. рис. 53).

Таблиця 53 – Час виконання комплексних транзакцій при одночасній зміні всіх факторів для Two-Phase Commit

| | Транзакція №1 (мс) | Транзакція №2 (мс) |
|---|---|---|
| Basic | 3944.22 | 5025.67 |
| Basic+ | 11040.57 | 12579.28 |
| Basic++ | 19305.57 | 21573.36 |
| Medium | 43880.35 | 62350.21 |
| Intensive | 739043.99 | 164176.99 |

Обом комплексним транзакціям притаманний дуже стрімкий ріст, тобто, час їх виконання зростає в декілька разів між рівнями. Також варто зазначити, що ці показники часу виконання набагато більші, ніж ті, що були наведені для інших методів.

У результаті були проведені заміри для методу реалізації розподілених транзакцій на рівні додатку за допомогою шаблону Two-Phase Commit. На основі отриманих замірів буде проходити порівняння з іншими методами.

# 5 АНАЛІЗ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

## 5.1 Порівняння продуктивності методів

Після отримання результатів замірів можна навести порівняння основних отриманих метрик для всіх підходів. Далі будуть наведені основні найбільш цікаві тренди продуктивності на основі проведених експериментів з підтримки ACID транзакцій для NoSQL СУБД MongoDB, NewSQL СУБД VoltDB та транзакційності на рівні додатку для SQL Server та мови C#.

У першу чергу показано порівння споживання ресурсів кожного з методів. Нижче наведено порівняльну гістограму, що для кожного режиму навантаження зі зміною кількості даних показує, скільки мегабайт пам'яті займає база даних для кожного методу. При цьому на кожному режимі для всіх трьох методів збергіється одна й та сама кількість сутностей.

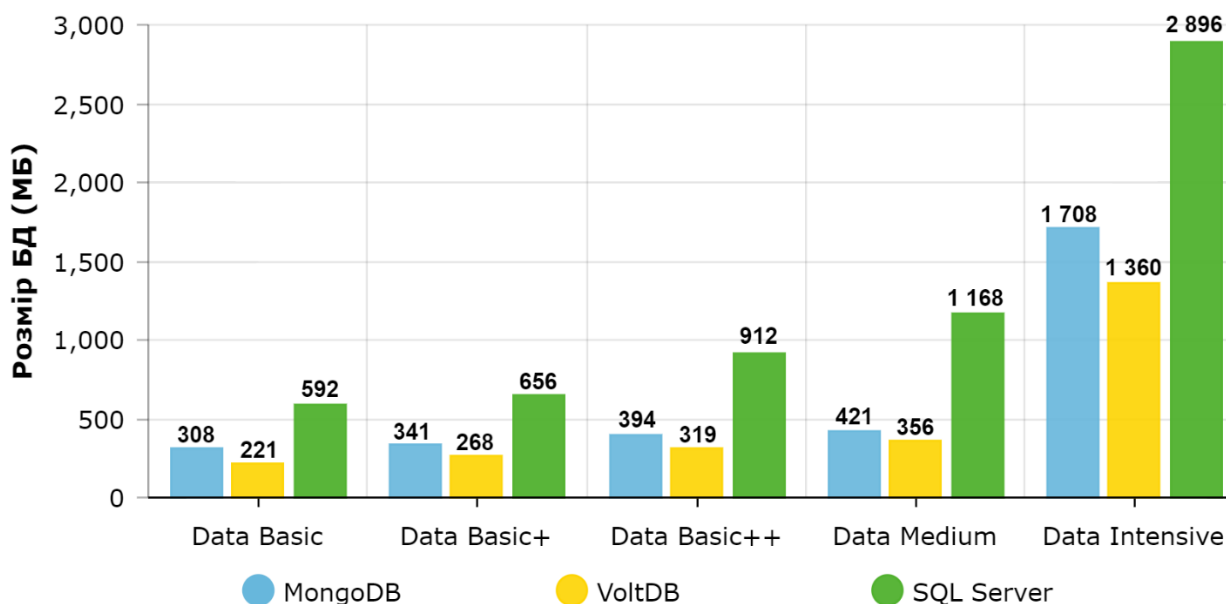


Рисунок 5.1 – Порівняльна гістограма розміру бази даних

З гістограми чітко видно, що найбільше пам'яті в абсолютних числах споживає SQL Server, а найменше потребує VoltDB. При цьому зі збільшенням кількості сутностей у базі розрив між SQL Server та іншими двома базами тільки

зростає. У цей же час розрив між VoltDB та MongoDB для перших рівнів навантаження залишається досить статичним і не зростає. Але вже на піковому навантаженні стає більш помітним розрив, а, отже, для MongoDB все ж розмір бази даних зростає швидше, ніж у випадку VoltDB. З усього цього можна зробити висновок, що для зберігання однієї сутності найбільше пам’яті споживає SQL Server, на другому місці MongoDB, а найменше потребує VoltDB.

Далі йтиме порівняння кількості процесорного часу, що споживається різними методами при виконанні запитів на різних режимах навантаження. Тут одразу потрібно одразу сказати, що споживання процесорного часу для всіх методів залежить від об’єму даних у базі. Далі наведено графік порівняння росту споживання процесорного часу при поступовому збільшенні об’єму даних (разом із новим режимом), що зберігається в БД.

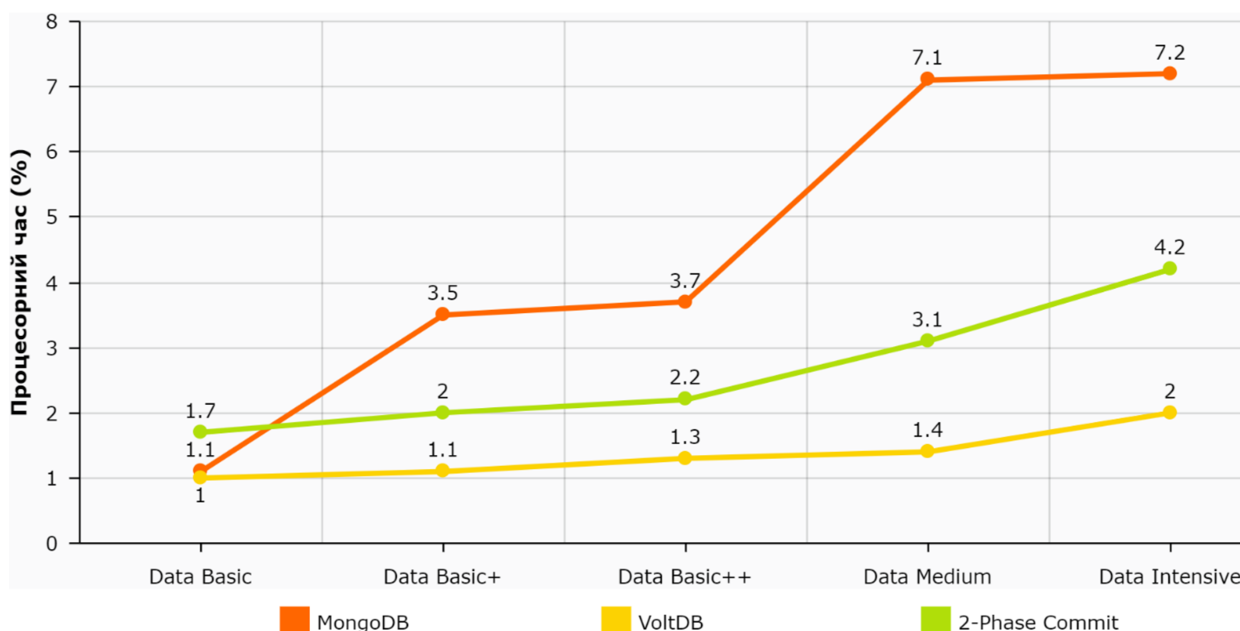


Рисунок 5.2 – Порівняльний графік споживаного процесорного часу при змінному об’ємі даних

На порівняльному графіку видно, що найбільшу залежність від кількості даних у базі має MongoDB, для якої присутнє ступінчасте зростання процесорного часу, який у загальному вигляді зростає в 7 раз між найнижчим та

найвищим рівнями. Другим за споживанням є метод Two-Phase Commit, в якому майже для всіх режимів як в абсолютних числах значення споживання менше за попередній метод, так і не такий швидкий ріст споживання. І найбільш ефективно використовує процесорний час VoltDB, для якої ріст при переході між режимами складає кілька десятих відсотка і цей ріст прогресує дуже повільно.

З іншого боку дуже цікаві результати замірів при змінній кількості одночасних підключень до бази. У цьому випадку, навпаки, MongoDB показує дуже ефективне використання процесорного часу, у той час як для SQL методів споживання цього ресурсу досить швидко зростає зі збільшенням підключень. Відповідні тренди показані на рисунку 5.3.

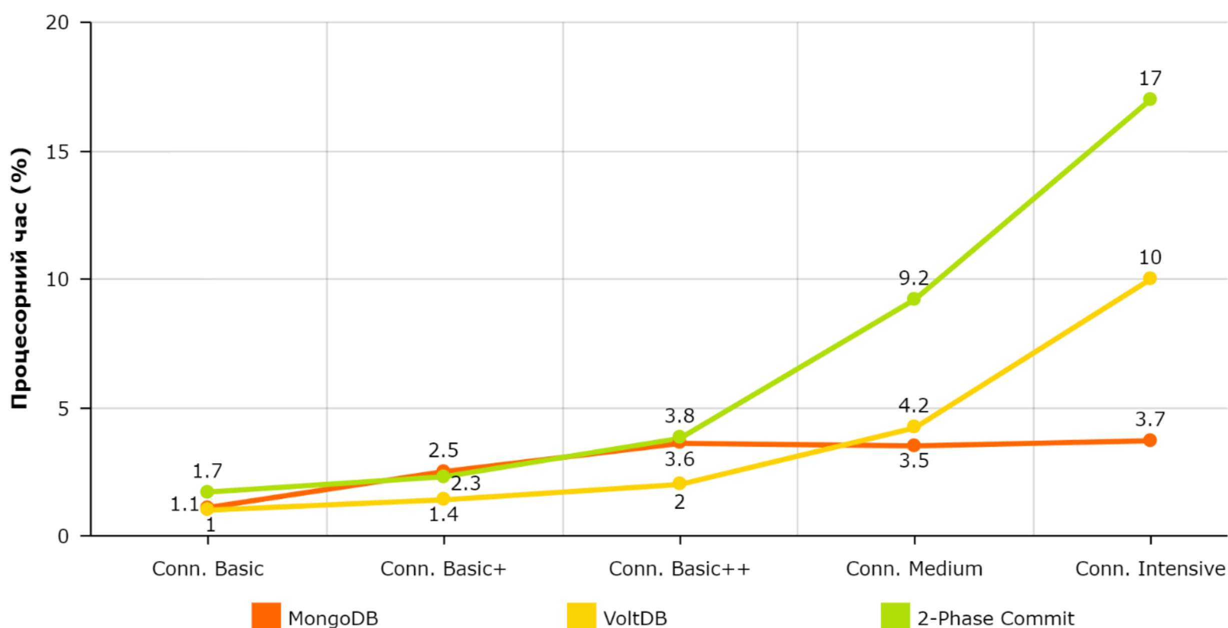


Рисунок 5.3 – Порівняльний графік споживаного процесорного часу при змінній кількості одночасних підключень

На графіку видно, що споживання процесорного часу з боку MongoDB зростає дуже повільно, в той час як для VoltDB та Two-Phase Commit це споживання зростає дуже жваво та складає приблизно 10 разів між найнижчим та найвищим рівнями. При цьому найбільше CPU поглинає Two-Phase Commit метод, а також для нього споживання зростає трохи швидше, ніж для VoltDB, у випадку збільшення кількості одночасних підключень.

Коли заздалегідь невідомі пропорції кількості даних та користувачів або припускається характерне зростання обох величин, то для цього краще порівняти метрики CPU коли всі фактори навантаження одночасно та поступово збільшуються. Порівняльна гістограма споживання процесорного часу при одночасній зміні всіх факторів навантаження наведена на рисунку 5.4.

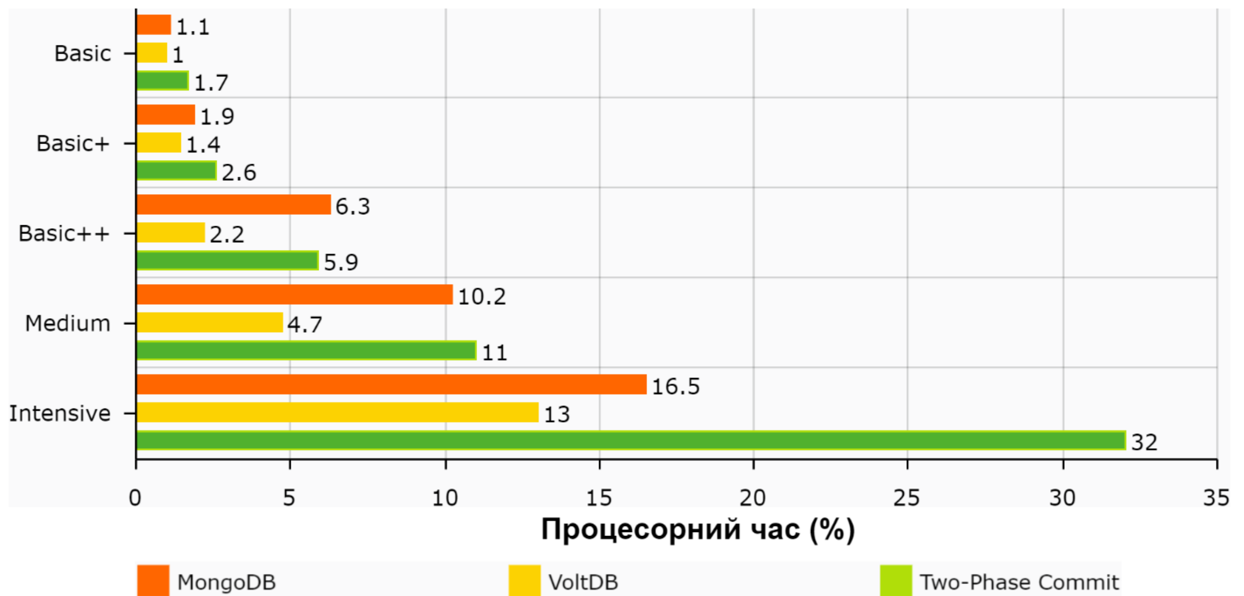


Рисунок 5.4 – Порівняльна гістограма споживання процесорного часу при одночасній зміні всіх факторів навантаження

Гістограма показує, що при одночасній зміні всіх факторів навантаження споживання CPU стрімко зростає для всіх методів. Але найменш ефективно використовує ці ресурси Two-Phase Commit підхід, де на піковому навантаженні споживання в два рази більше, ніж в інших методах. Трохи менше CPU споживає MongoDB, а найбільш ефективним з точки зору використання цього ресурсу є метод реалізації транзакцій через VoltDB.

Також потрібно сказати, що процесорний час для всіх методів практично не залежить від кількості реплік у кластері. Від цього фактору прослідковується залежність тільки для Two-Phase Commit, але різниця між найвищим та найнижчим рівнями складає менше 1%.

Далі йтиме порівняння використання останнього виду ресурсів – оперативної пам’яті. Із самого старту зрозуміло, що MongoDB споживає в декілька разів менше оперативної пам’яті, ніж VoltDB та Two-Phase Commit. Так, при зміні кількості даних MongoDB споживає від 91 MB оперативної пам’яті для найнижчого рівня та до 99 MB для найвищого рівня, а при змінній кількості підключень від 83 MB до 97 MB. Навіть при наявності певної статистичної похибки видно, що ріст, якщо і є, то дуже повільний. У той же час, VoltDB споживає при зміні даних від 218 MB до 560 MB, а при зміні підключень від 218 MB до 1138 MB. А найбільше потребує реалізація через Two-Phase та SQL Server, яка споживає від 347 MB до 581 MB при зміні даних та від 347 MB до 1226 MB при зміні даних. Відповідний порівняльний графік наведений на рисунку 5.5.

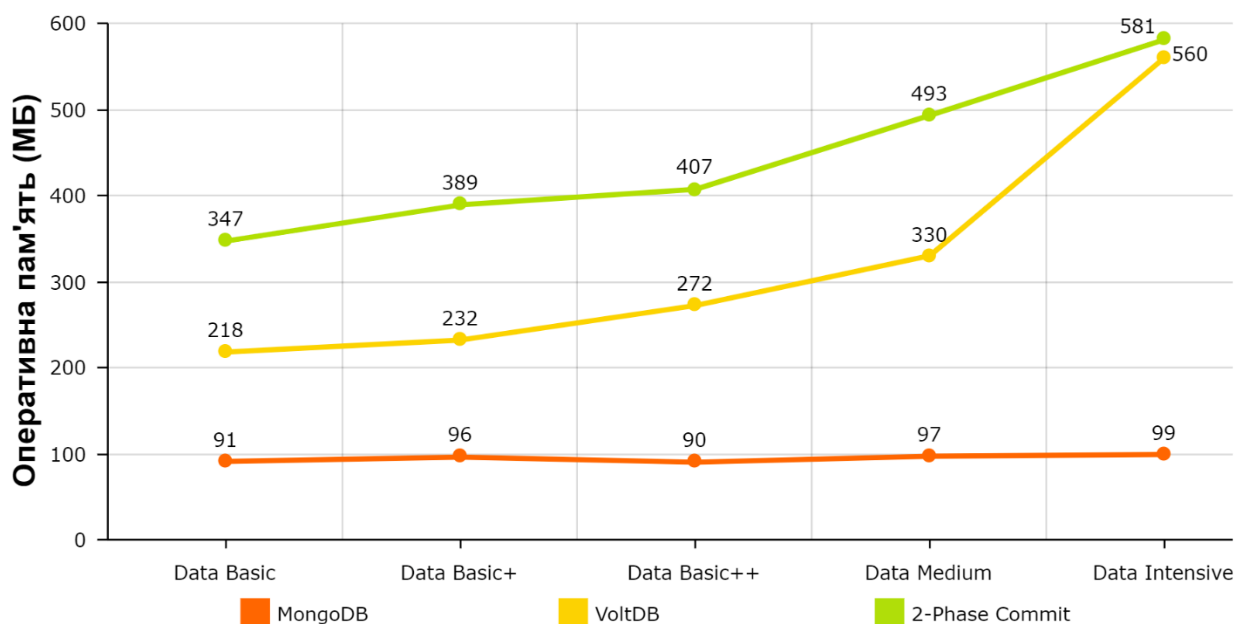


Рисунок 5.5 – Порівняльний графік споживання оперативної пам’яті при зміні об’єму даних

На графіку видно, що хоча на початкових рівнях для VoltDB споживання RAM менше, ніж для випадку Two-Phase, але ріст більш жвавий і на піковому навантаженні дані два методи мають майже однакові значення.

При зміні кількості одночасних підключень зберігаються ті ж самі тренди, що і в попередньому випадку. MongoDB споживає до 100 МБ оперативної пам'яті на всіх рівнях навантаження, що нижче на кілька порядків, ніж інші методи. VoltDB та Two-Phase Commit мають дуже близькі значення та зростають у часі приблизно з однаковою швидкістю, але все в абсолютних числах VoltDB використовує оперативну пам'ять ефективніше. Відповідіні результати відображені на рисунку 5.6.

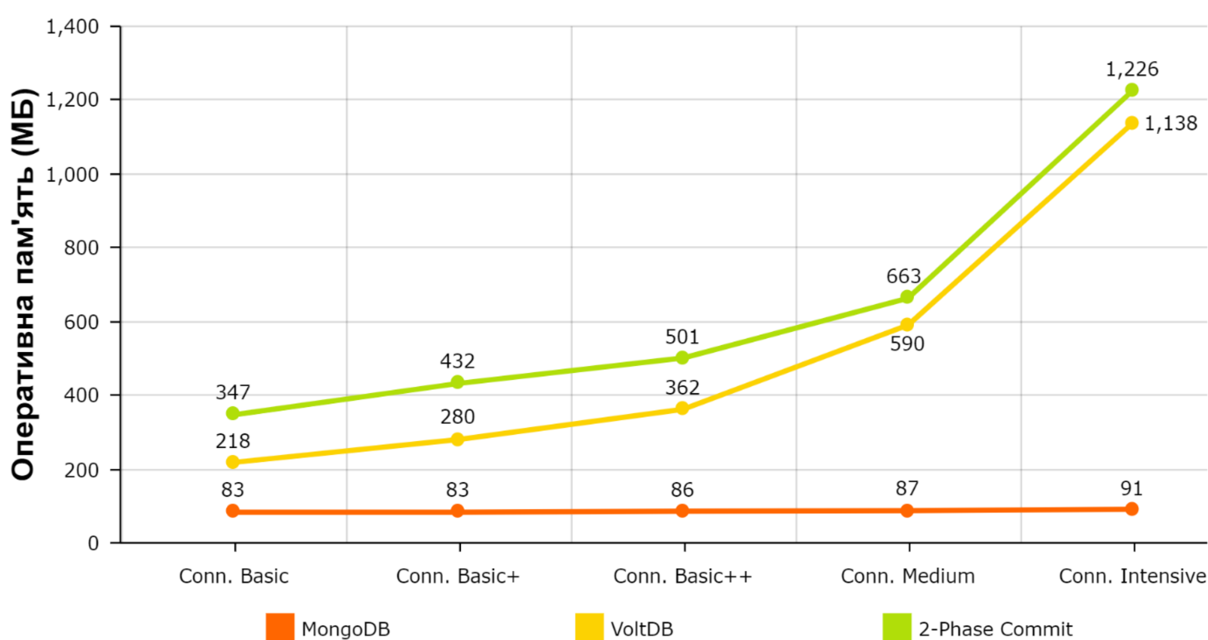


Рисунок 5.6 – Порівняльний графік споживання оперативної пам'яті при зміні кількості одночасних підключень

Схожі результати отримані при одночасній зміні всіх факторів навантаження, але в цьому випадку споживання оперативної пам'яті для MongoDB все ж помітно збільшується, тим не менш досі набагато менше, ніж для інших двох методів. Тренди споживання та росту для VoltDB та Two-Phase Commit залишаються такими ж, але при цьому останній метод споживає в абсолютних числах більше RAM. Відповідні результати наведені на гістограмі на рисунку 5.7.

Також необхідно сказати, що всі 3 методи дуже з точки зору споживання оперативної пам'яті дуже слабко залежать від кількості реплік у кластері.

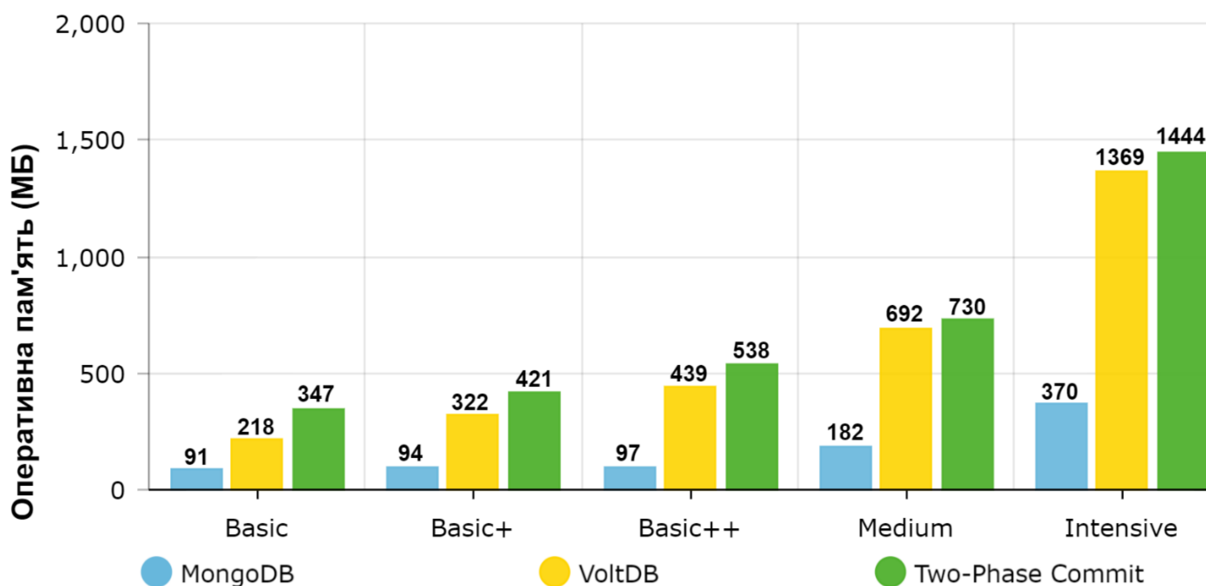


Рисунок 5.7 – Порівняльна гістограма споживання оперативної пам'яті при зміні всіх факторів навантаження

З отриманих результатів можна зробити висновок, що для будь-якого типу навантаження найбільш ефективно оперативну пам'ять використовує MongoDB, наступним йде VoltDB, а найбільше пам'яті витрачає метод Two-Phase Commit.

Також варто зазначити, що крім ресурсів, які споживають самі сервери СУБД, для Two-Phase Commit підходу споживаються набагато більше ресурсів при виконанні додатку, який надсилає запити до бази даних, ніж для MongoDB та VoltDB (див. табл. 54).

Таблиця 54 – Порівняння споживаних ресурсів додатком, що надсилає запити до бази даних

| | Two-Phase Commit | MongoDB та VoltDB |
|---|---|---|
| RAM | ~2.1-2.5 Гб | ~0.8-1.2 Гб |
| CPU | ~45-55% | ~9-12% |

З таблиці видно, що крім більшого навантаження на сервер СУБД Two-Phase Commit також більше навантажує сам додаток.

Після проведення аналізу стає зрозумілим, що метод Two-Phase Commit споживає більше усіх типів ресурсів за інші два підходи майже на всіх рівнях навантаження. З іншого боку метод з VoltDB показав себе більш оптимальним з точки зору використання місця на диску та споживання процесорного часу. А MongoDB найбільш ефективно витрачає оперативну пам’ять сервера.

Далі будуть порівнюватися результати часу виконання запитів та транзакцій. У першу чергу будуть проаналізовані результати для запитів на зчитування даних. Тут одразу потрібно сказати, що для всіх трьох методів час виконання цього запиту не залежить від об’єму даних у базі, адже операція зчитування за індексом має логарифмічну складність, та від кількості реплік у кластері, бо для зчитування потрібно встановити з’єднання лише з однією базою. З іншого боку час виконання для всіх методів залежить від кількості одночасних підключень, що відображено на рисунку 5.8.

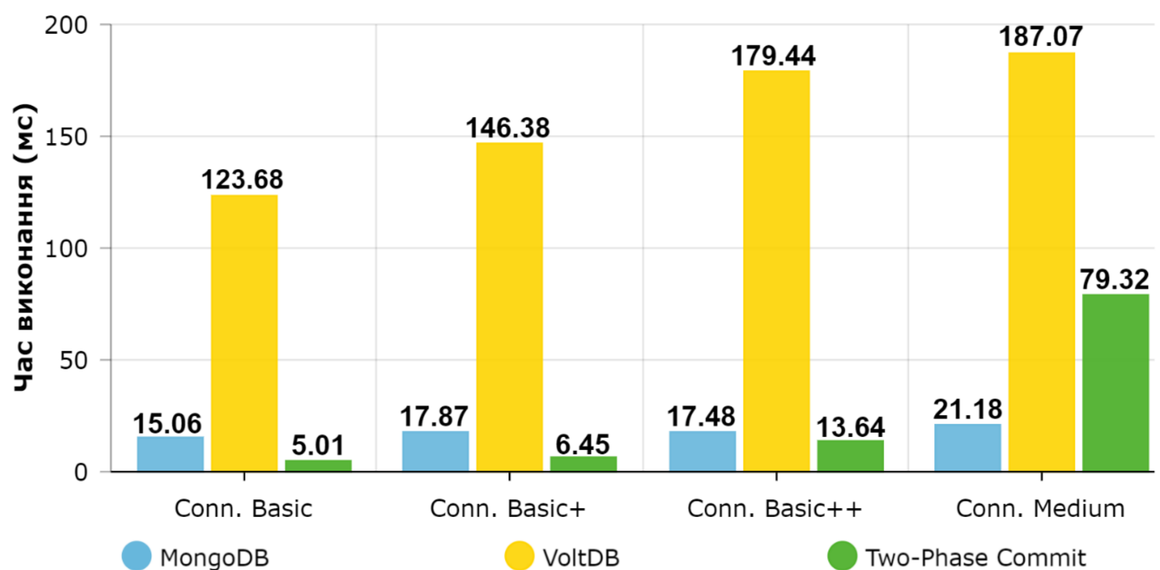


Рисунок 5.8 – Порівняльна гістограма часу виконання запиту отримання замовлення за ідентифікатором при зміні кількості підключень

З гістограми чітко видно, що найбільше часу потребує виконання для VoltDB, що у свою чергу демонструє сильно чутливість при до навантажень з

точки зору кількості підключень. На порядок швидше виконується цей запит на зчитування для MongoDB та Two-Phase Commit через SQL Server. І хоча для перших режимів MongoDB показує більший час виконання, але цей метод має меншу швидкість росту цього часу, ніж у випадку Two-Phase Commit, який вже на Medium режимі потребує в кілька разів більше часу. Це свідчить про стійкість MongoDB до цього виду навантаження.

На гістограмі вище показані результати для всіх режимів, крім Intensive, адже на цьому режимі час виконання стає на порядок вищим, що відображено в таблиці 55.

Таблиця 55 – Порівняння часу виконання запиту на зчитування замовлення за ідентифікатором для режиму Connection Intensive

| | MongoDB | VoltDB | Two-Phase Commit |
|---|---|---|---|
| Час виконання (мс) | 41.32 мс | 6451.23 мс | 1539.32 мс |
| Відносні показники | 1x | 156x | 37x |

Із таблиці чітко видно, що тренди залишаються такими ж: MongoDB виконує зчитування за індексом швидше всіх, далі йде SQL Server, а найдовше запит виконується для VoltDB.

При одночасній зміні всіх факторів навантаження абсолютні значення часу вище, ніж при зміні тільки кількості підключень, але всі зазначені тренди продуктивності залишаються такими ж. Із цього слідує, що при будь-якому типі навантаження найбільш оптимальним методом для зчитування записів за індексом є MongoDB.

Наступним запитом на зчитування йде отримання трьох найпопулярніших категорій товарів, що включає операції групування, сортування та лімітування даних. При виконанні цього запиту для всіх методів присутній характерний ріст часу виконання від об’єму даних та кількості одночасних підключень.

При зміні об'єму даних у базі найбільш стрімко зростає час виконання для VoltDB, для якого час сягає аж до 5 хвилин. З іншого боку MongoDB та Two-Phase Commit мають приблизно однакові значення (при цьому в абсолютних значення Two-Phase Commit відпрацьовує швидше), що відрізняються приблизно на 1 секунду, а швидкість росту час для обох методів однакова (див. рис. 5.9).

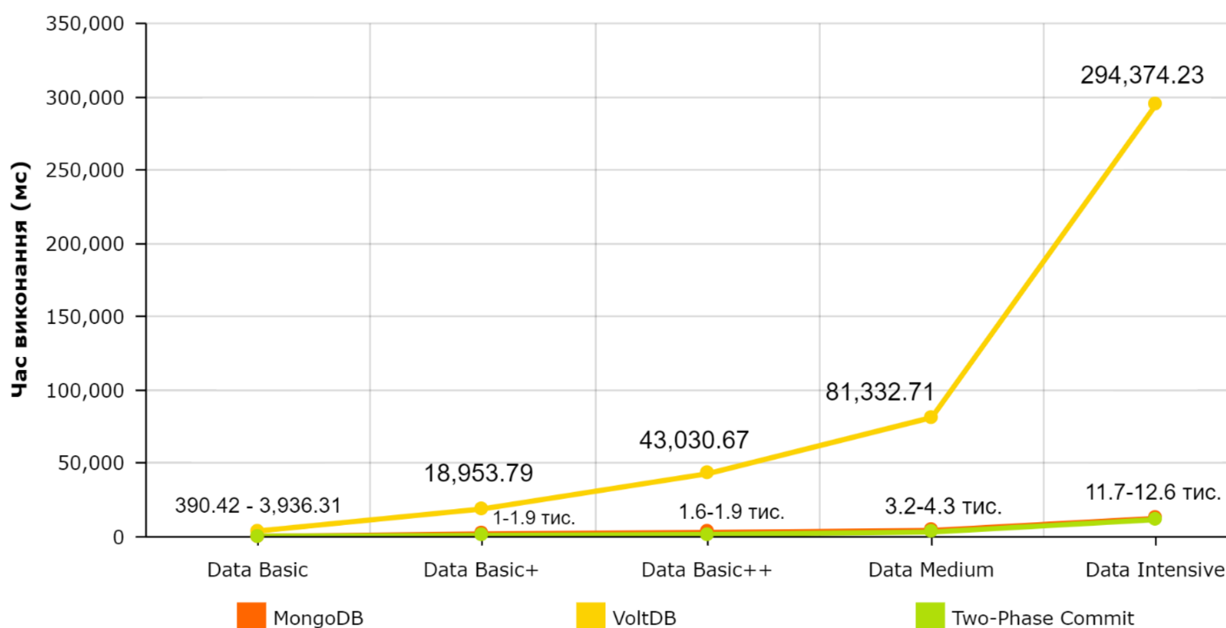


Рисунок 5.9 – Порівняльний графік часу виконання запиту отримання категорій товарів при зміні об'єму даних

На графіку вище видно, що при зміні даних MongoDB та Two-Phase Commit є найбільш оптимальними методами для виконання складної операції зчитування даних.

Трохи інша ситуація з часом виконання даного запиту при зміні фактору кількості одночасних підключень. VoltDB як і в попередньому випадку виконує запит на всіх рівнях повільніше, ніж інші два методи. Але з іншого боку теперь MongoDB виконує запит швидше, ніж Two-Phase Commit, а також ця різниця з кожним наступним рівнем стає все більш помітною. Це вкотре свідчить про ефективність MongoDB при високих навантаженнях з боку одночасних підключень користувачів.

Відповідні результати замірів наведено на порівняльному графіку на рисунку 5.10.

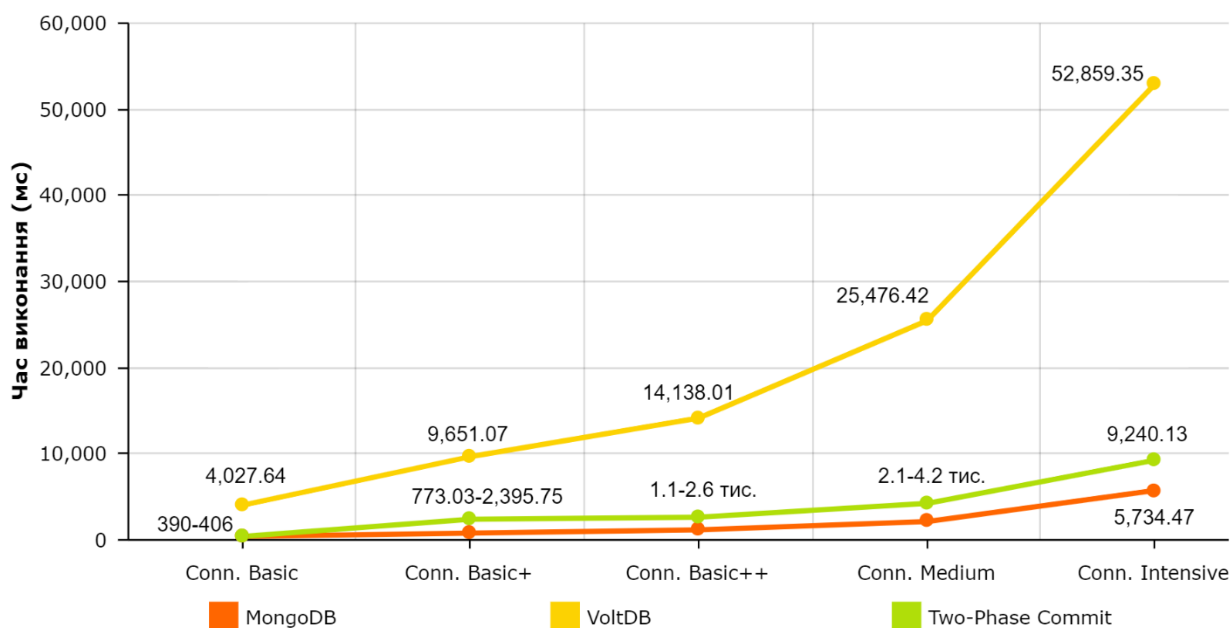


Рисунок 5.10 – Порівняльний графік часу виконання запиту отримання категорій товарів при зміні кількості підключень

Графік зміни часу виконання даного запиту при одночасній зміні всіх факторів навантаження наведено на рисунку 5.11. На ньому можна легко побачити, що VoltDB залишається найменш ефективним методом для цього складного запиту зчитування, а у же час Two-Phase Commit метод через SQL Server є трохи оптимальнішим на перших 4 рівнях, але на піковому навантаженні значно поступається MongoDB. Це означає, що Two-Phase Commit метод з SQL Server можна обрати для цього типу запиту, коли заздалегідь відомий максимальний поріг навантаження, а для більшої надійності та масштабованості, звісно, що краще обрати MongoDB, яка показує дуже високі результати на пікових навантаження, а також програє у часі при на деяких режимах з мінімальним відривом.

Таким чином були розглянуті запити різні запити на зчитування даних, для яких при високому навантаженні найбільш оптимальним є використання Mongo.

Далі будуть розглянуті з точки зору часу виконання різні транзакційні запити на запис даних.

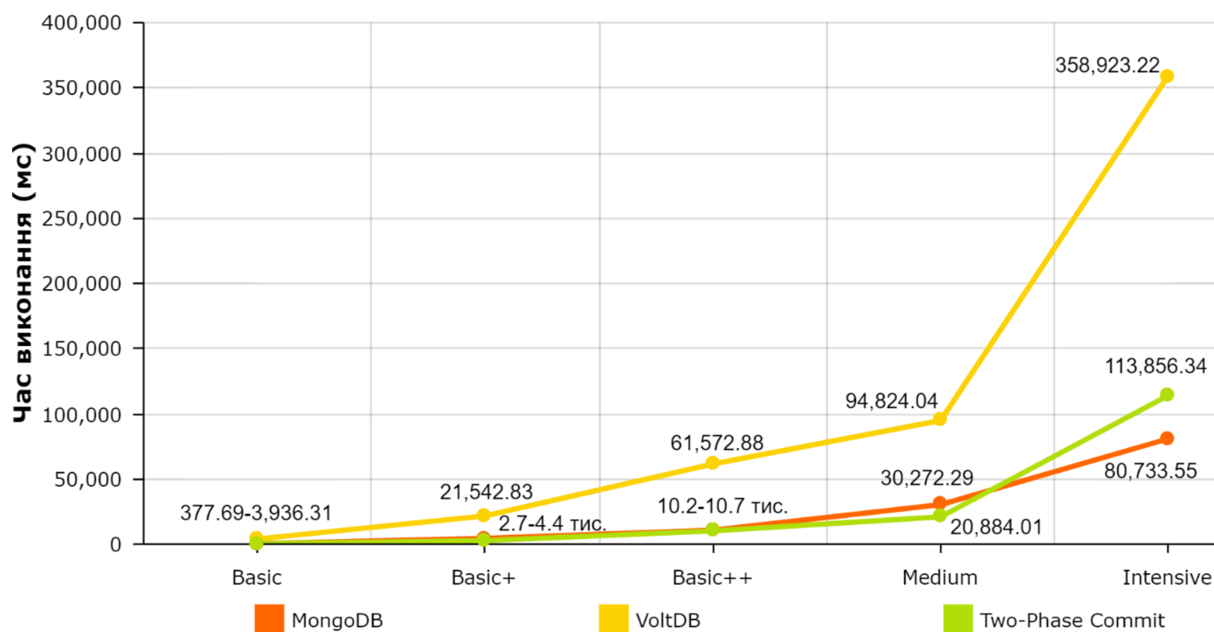


Рисунок 5.11 – Порівняльний графік часу виконання запиту отримання категорій товарів при зміні всіх факторів навантаження

Першими транзакціями будуть розглянуті однотипні операції на вставку, оновлення та видалення даних.

Спочатку буде проведено аналіз для транзакції вставки замовлення з його елементами. Експеримент показав, що MongoDB виконує цю транзакцію в декілька разів швидше, ніж VoltDB та Two-Phase Commit при зміні будь-якого фактору. Також було виявлено, що час виконання вставки для MongoDB не залежить від об’єму даних у БД та вар’юється від 86.76 мс до 133.09 мс на різних рівнях. У той же час VoltDB та Two-Phase Commit методи залежать у часі виконання від об’єму даних у базі, але в абсолютних значення та швидкості росту VoltDB програє, адже запити для цього методу виконуються набагато довше (див. рис. 5.12).

Далі розглядатиметься час виконання транзакції на вставку при зміні кількості одночасних підключень. У цьому випадку більш менш стабільно зростає

час виконання для всіх методів. Найдовше транзакцію виконує Two-Phase Commit, далі йде VoltDB, а найшвидше виконує MongoDB.

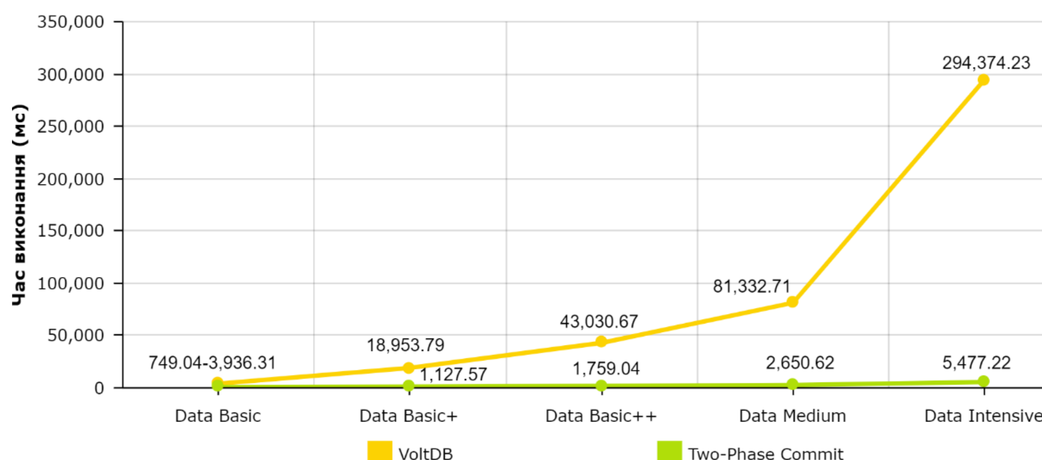


Рисунок 5.12 – Порівняльний графік часу виконання транзакції на вставку при зміні об’єму даних

Відповідні тренди росту при зміні кількості підключень відображено рисунку 5.13.

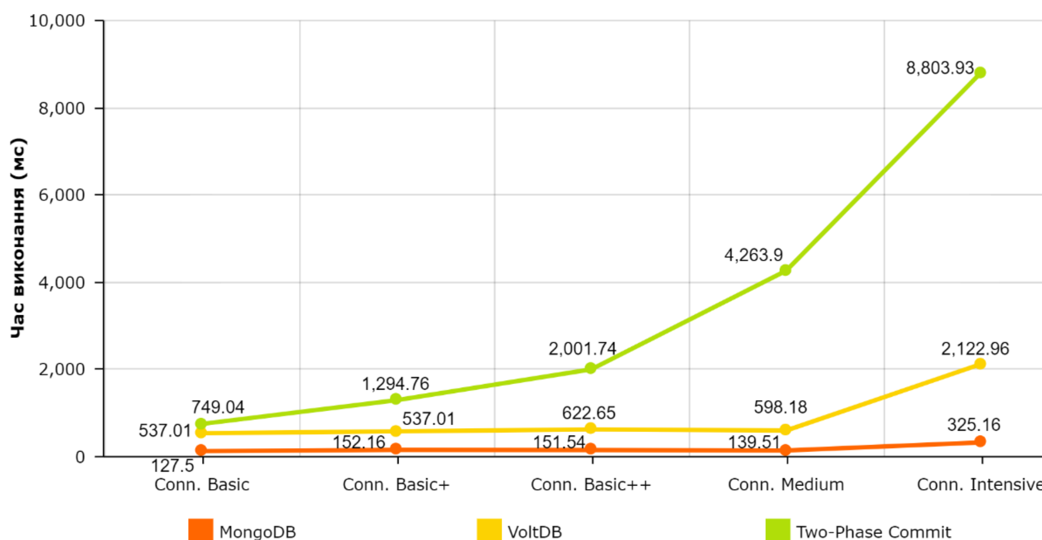


Рисунок 5.13 – Порівняльний графік часу виконання транзакції на вставку при зміні кількості підключень

На графіку видно, що й час швидкість росту часу також найбільша у Two-Phase Commit, а на другому місці у VoltDB. Також варто сказати, що при

одночасній зміні всіх факторів навантаження абсолютні значення трохи вищі, але загальний тренд залишається таким же, як і випадку зміни кількості підключень.

Також варто зазначити від фактору кількості реплік залежать тільки MongoDB та Two-Phase Commit (див. рис. 5.14). VoltDB не залежить від цього фактору і виконує транзакцію при зміні цього фактора від 771.9 до 847.57.

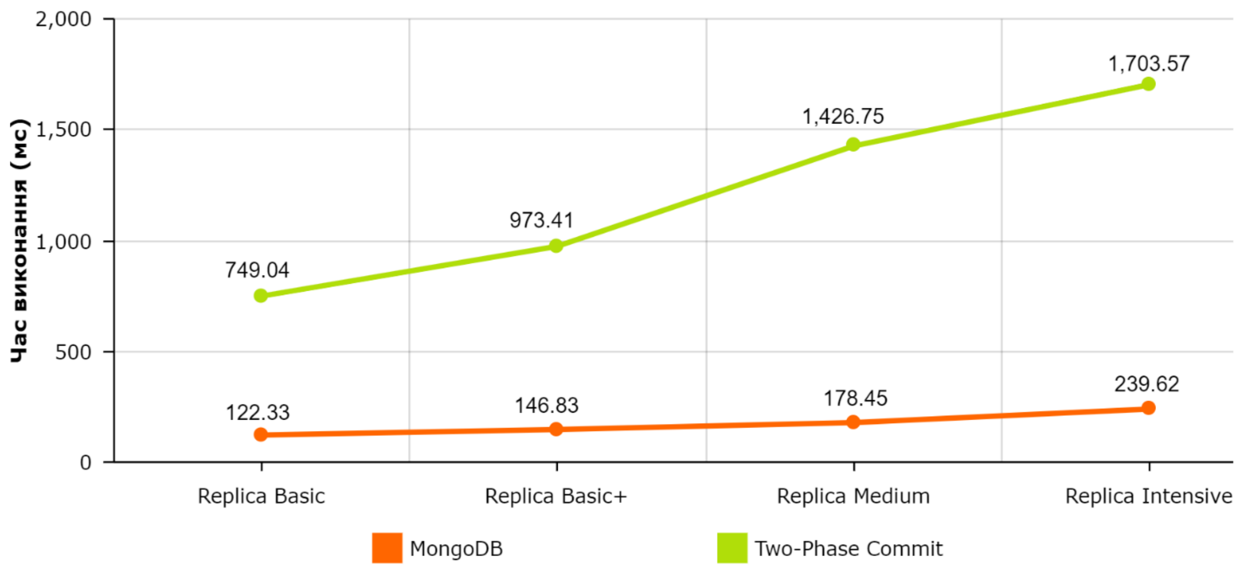


Рисунок 5.14 – Порівняльний графік часу виконання транзакції на вставку при зміні кількості реплік

Таким чином видно, що для MongoDB ріст дуже повільний і складає всього 100 мс між крайніми рівнями, у той час як для Two-Phase Commit це ріст є більш жвавим, але не геометричним.

На основі отриманих замірів можна побудувати порівняльну гістограму пропускної здатності для транзакції вставки, тобто, скільки сутностей можна транзакційно вставити в базу даних за 1 секунду (див. рис. 5.15). З гістограми видно, що найбільше сутностей може обробити MongoDB, потім VoltDB, а найменше можна вставити за допомогою Two-Phase Commit.

Наступним кроком розглядається транзакція оновлення ціни продуктів, кількість яких менша за певний ліміт. Для виконання даної транзакції необхідне буде сканування всіє таблиці або колекції продуктів. При змінній кількості даних у базі час для всіх методів зростає. При цьому найшвидше цей запит виконує

VoltDB, а Two-Phase Commit працює швидше за MongoDB лише на початкових рівнях навантаження, але має набагато жвавіший ріст цього часу.

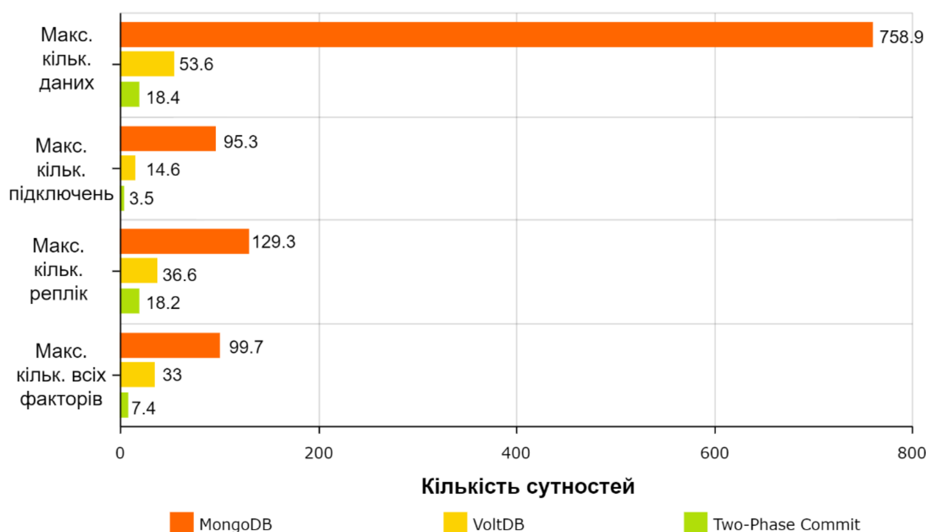


Рисунок 5.15 – Порівняльна гістограма пропускної здатності для вставки

Відповідні тренди зображені на порівняльному графіку росту часу при зміні об’єму даних у БД (див. рис. 5.16).

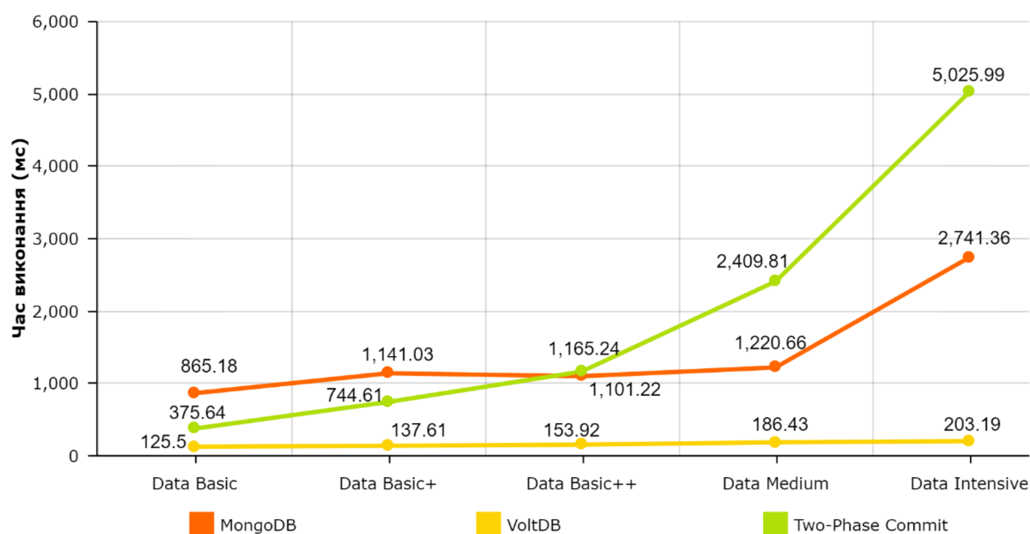


Рисунок 5.16 – Порівняльний графік часу виконання транзакції на оновлення при зміні об’єму даних

Далі розглядається ситуація, коли буде поступово збільшуватися кількість одночасних підключень до бази. Хоча значення виросли, але тренди збереглися:

найшвидше запит виконує VoltDB, потім на початкових рівнях швидше працює Two-Phase Commit, але має більш швидкий ріст часу (див. рис. 5.17).

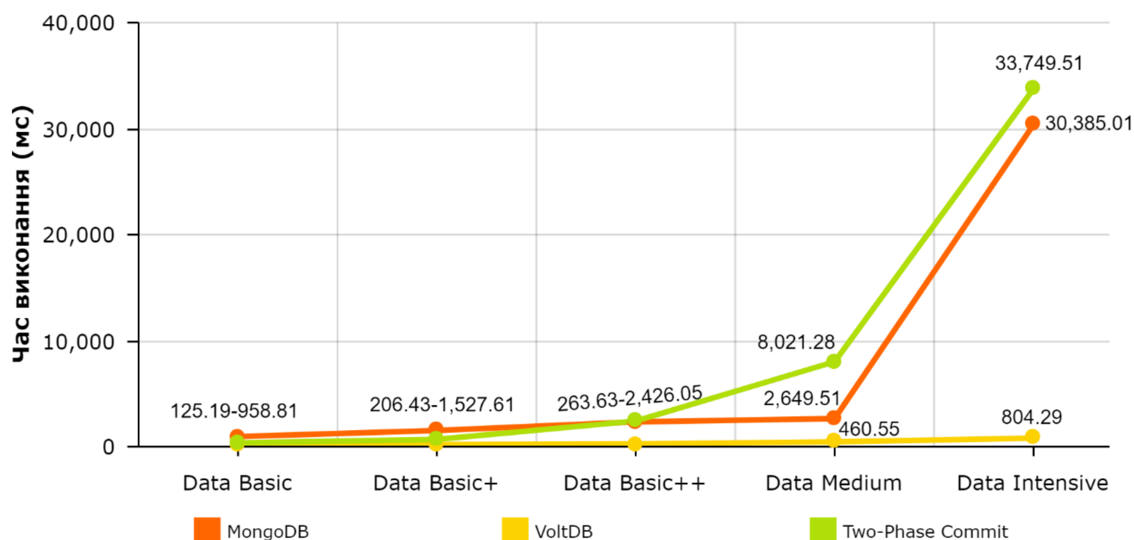


Рисунок 5.17 – Порівняльний графік часу виконання транзакції на оновлення при зміні кількості підключень

Нижче наведено порівняння часу виконання запиту оновлення для всіх методів при змінній кількості реплік у кластері бази (див. рис. 5.18).

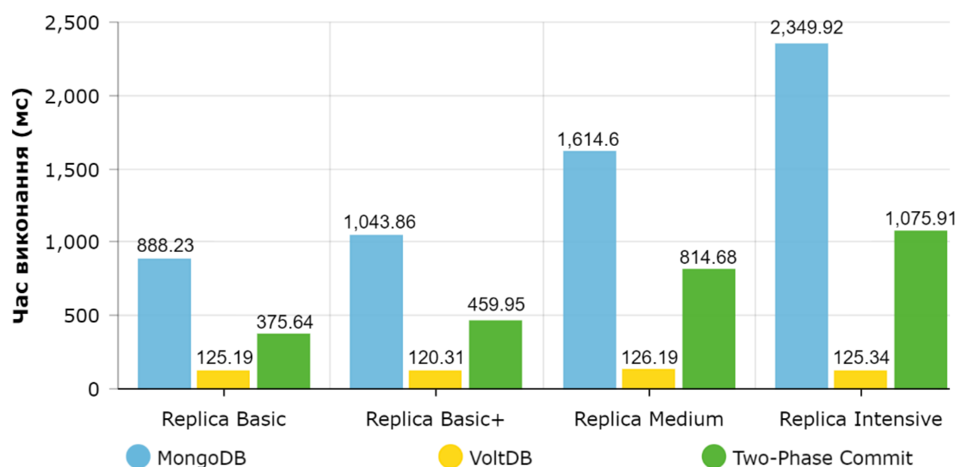


Рисунок 5.18 – Порівняльна гістограма часу виконання транзакції на оновлення при зміні кількості реплік

З гістограми видно, що VoltDB майже не залежить від кількості реплік у кластері БД. У цей же час інші два методи помітно зростають у часі, але при зміні

кількості реплік швидше виконує запит Two-Phase Commit, що має швидкість росту часу менше за MongoDB.

Маючи результати для всіх методів, можна порівняти пропускну здатність, тобто максимальну кількість сутностей, що можуть бути транзакційно оновлені за 1 секунду, для всіх методів при піковому навантаженню кожного типу (див. табл. 56).

Таблиця 56 – Порівняння пропускної здатності на пікових навантаженнях

| | MongoDB (кільк. сутностей) | VoltDB (кільк. сутностей) | Two-Phase Commit (кільк. сутностей) |
|---|---|---|---|
| При макс. об'ємі даних | 1824 (1.8x) | 93283 (93.8x) | 994.8 (1x) |
| При макс. кількості підключень | 7 (1.2x) | 248 (42x) | 5.9 (1x) |
| При макс. кількості реплік | 85 (1x) | 1585 (18.6x) | 185.9 (2.2x) |
| При макс. значеннях всіх факторів | 92 (1.6x) | 1927 (33.8x) | 57 (1x) |

З таблиці видно, що найбільш ефективним є метод реалізації через VoltDB, також для більшості типів навантаження друге місце посідає MongoDB.

Наступним кроком є розгляд ефективності виконання транзакції на видалення замовлень, чия загальна сума знаходиться у певному діапазоні. Цей запит потребує сканування кількох таблиць, а також групування даних.

Експеримент показав, що, як і в попередньому випадку, швидше всього виконується запит для методу VoltDB, наступним йде Two-Phase Commit, найдовше виконується для MongoDB. Але швидкість росту часу виконання при зміні об'єму даних та кількості підключень краще буде дослідити через графічне

відображення відповідних функцій лінійної регресії. Нижче наведено графік залежностей часу виконання транзакції від об'єму даних (див. рис. 5.19).

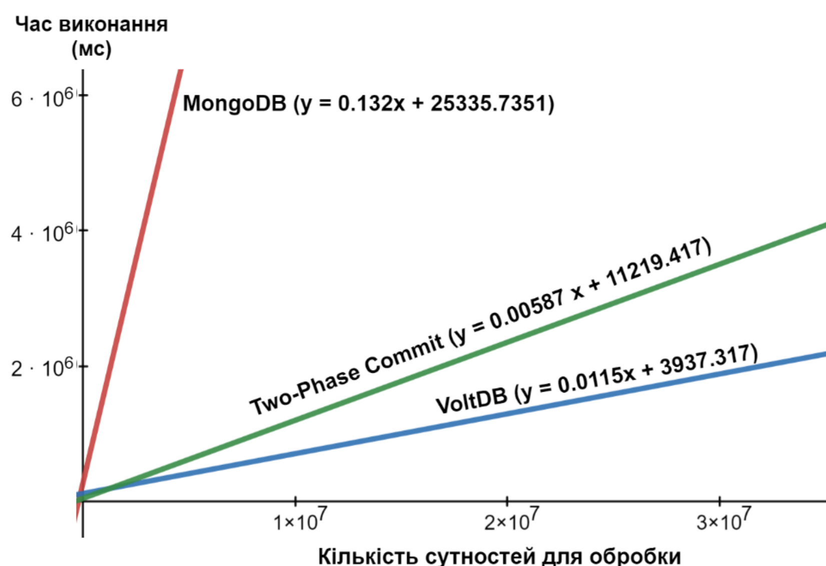


Рисунок 5.19 – Графік функцій росту часу в залежності від об'єму даних

Також на рисунку 5.20 наведено функції залежності часу виконання транзакції від кількості одночаних підключень (див. рис. 5.19).

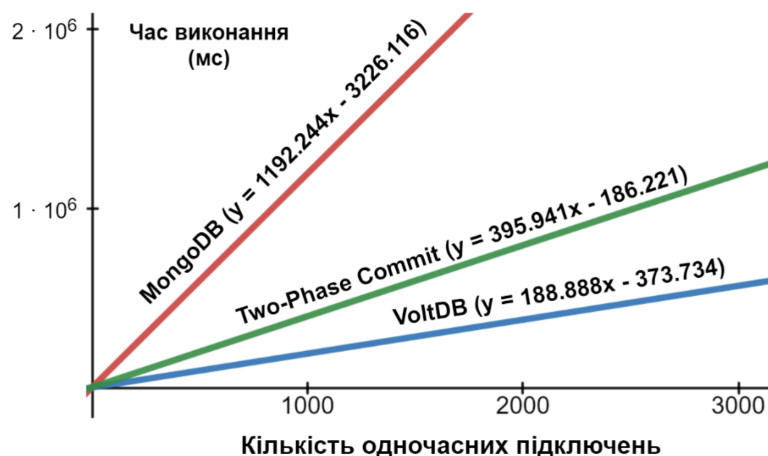


Рисунок 5.20 – Графік функцій росту часу в залежності від кількості підключень

На обох графіках чітко видно, що найшвидше зростає час виконання для MongoDB, а це значить, що цей метод не є найоптимальнішим для виконання

цього типу запиту під високими навантаженнями. З іншої сторони набагато повільніше зростає функція для Two-Phase, а найповільніше вона зростає для VoltDB, що робить цей підхід найбільш ефективним для високого навантаження з боку об'єму даних та кількості підключень.

При зміні кількості вузлів у кластері час виконання кожного з методів збільшується. Але у випадку VoltDB та MongoDB цей час зростає досить повільно, коли для Two-Phase Commit цей ріст набагато помітніший, адже цей метод переганяє у часі MongoDB при найбільшій кількості реплік (див. рис. 5.21)

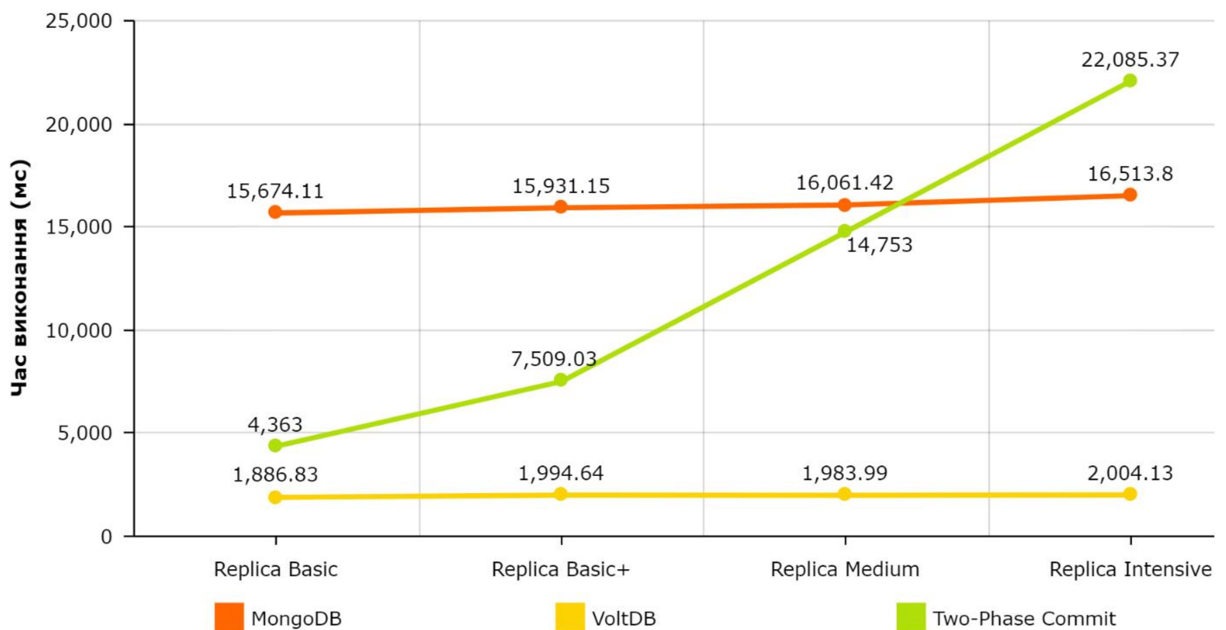


Рисунок 5.21 – Порівняльний графік часу виконання транзакції видалення при зміні кількості реплік

Цей факт демонструє сильну залежність Two-Phase Commit від кількості реплік у кластері БД. При одночасній зміні всіх факторів для транзакції видалення залишається той тренд, що VoltDB виконує її найшвидше, далі за швидкістю йде Two-Phase Commit, а найповільніше запит виконується для MongoDB.

Наступним і останнім кроком є йде розгляд порівняння часу виконання комплексних транзакцій. Суть замірів для таких транзакцій полягає у тому, щоб перевірити, який метод найшвидше вміє фіксувати транзакції, що включають багато запитів на запис, адже такі типи транзакцій найбільш поширені в сучасних

системах. Перша з таких транзакції містить дві операції вставки та дві операції оновлення. Із замірів стає зрозуміло, що MongoDB виконує цю транзакцію в десятки разів швидше за інші методи. А найдовше транзакція виконується через Two-Phase Commit. Швидкість росту кожного методу не змінює ці результати при жодному факторі навантаження, а тому на рисунку 5.22 зображена порівняльна гістограма часу виконання при максимальному навантаженні кожного фактору.

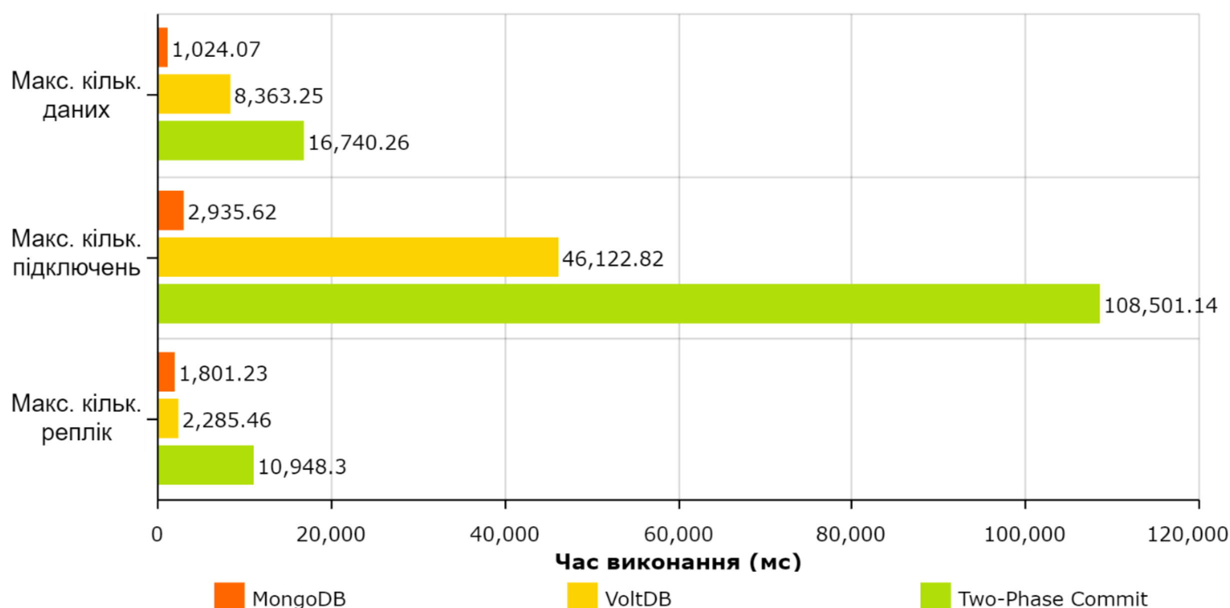


Рисунок 5.22 – Порівняльна гістограма часу виконання комплексної транзакції на максимальних навантаженнях

З гістограми одразу видно, що MongoDB набагато ефективніше фіксує комплексні транзакції за два інших методи. При максимальному навантаженні з боку всіх факторів тренди залишаються такими ж (MongoDB виконує найшвидше, Two-Phase Commit найповільніше), але час виконання в абсолютних числах збільшується в кілька разів (див. рис. 5.23).

Далі розглядається друга комплексна транзакція, яка містить 6 операцій вставки та 1 операцію оновлення. Дану транзакцію MongoDB, як і в попередньому випадку, виконує дану транзакцію на порядок швидше, ніж VoltDB та Two-Phase Commit. При цьому цей тренд зберігається при зміні будь-якого фактору навантаження. З іншого боку VoltDB виконує транзакцію швидше за Two-Phase

Commit при максимальному об’ємі даних у базі, а також при максимальній кількості реплік у кластері (див. рис. 5.24)

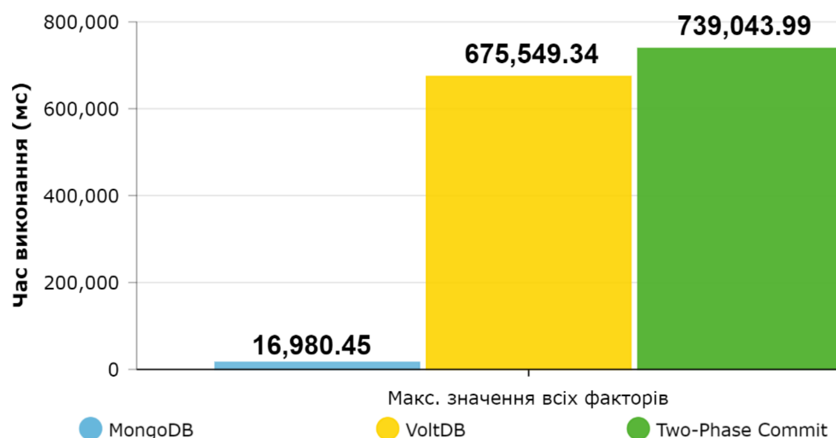


Рисунок 5.23 – Порівняльна гістограма часу виконання комплексної транзакції на максимальних значеннях всіх факторів навантаження

Отже, друга комплексна транзакція довела, що MongoDB дуже ефективно використовувати, коли транзакції повинні мати багато запитів на запис даних.

Рисунок 5.24 – Порівняльна гістограма часу виконання другої комплексної транзакції намаксимальних навантаженнях

У результаті було проведено порівняння ефективності всіх методів, на основі чого далі будуть розроблені практичні рекомендації по їх використанню.

## 5.2 Розробка рекомендацій щодо використання

Після проведення порівняльного аналізу всіх методів реалізації розподілених транзакцій та виявлення слабких сторін кожного з них можна розробити практичні рекомендації щодо використання того чи іншого методу в конкретній ситуації.

Спочатку будуть надані рекомендації для методу Two-Phase Commit за допомогою СУБД SQL Server та мови програмування C#. По-перше, потрібно сказати, що цей підхід потребує набагато більше ресурсів усіх типів: місця на постійному носії для зберігання даних, кількості процесорного часу та оперативної пам’яті. Особливо багато порівняно з іншими методами серверами СУБД витрачається процесорних ресурсів, у кілька разів більше, ніж у випадку MongoDB та VoltDB. Крім того сам додаток, що взаємодіє з базою даних, для даного методу витрачає набагато більше RAM та CPU ресурсів. Все це натякає на висновок, що Two-Phase Commit є дуже неефективним з точки зору використання ресурсів, а його впровадження на проект потребуватиме великих коштів на оренду потужних віртуальних машин для розміщення серверів баз даних та додатків, що взаємодіють з цими БД.

По-друге, в плані виконання транзакцій даний метод також потребує в більшості випадків або найбільше часу для виконання, або є другим за ефективністю з невеликим відривом (наприклад, для однотипних транзакцій оновлення та видалення). Для комплексних транзакцій Two-Phase Commit взагалі витрачав у багато разів більше часу, ніж інші методи. Також у багатьох випадках, цей метод виконував запит швидше за певного конкурента, але ріст цього часу при збільшенні навантаження був жвавішим, а тому на високих навантаженнях, Two-Phase Commit знову виконував ці запити найповільніше. Найкращу продуктивність цей метод показував лише на запитах зчитування лише при певного типу навантаження (наприклад, при зміні об’єму даних для отримання

найпопулярніших категорій товарів), але і там виграш у часі порівняно з найближчим конкурентом був мінімальний.

Two-Phase Commit шаблон був одним з перших методів реалізації розподілених ACID транзакцій, а тому і має багато недоліків. А враховуючи показники продуктивності часу виконання запитів і транзакцій та споживання ресурсів, то можна зробити висновок, що при розробці додатку будь-якого типу краще за все не використовувати цей метод реалізації розподілених ACID транзакцій, адже це впровадження буде коштувати досить дорого і не дасть очікуваної продуктивності додатку. Єдиною причиною використання даного методу є гостра потреба в кастомізації алгоритму проведення та фіксування транзакцій у розподіленій БД. Наприклад, коли потрібно імплементувати ексклюзивний рівень ізоляції транзакцій, що не підтримується певною СУБД, або коли потрібно фіксувати транзакції на вузлах у певному порядку (наприклад, в залежності від геолокації сервера). Це все можливо завдяки даному методу, адже керування транзакцією повністю в руках додатка та його розробників.

Далі наведено рекомендації щодо використання VoltDB та MongoDB в якості метода реалізації розподілених ACID транзакцій. Кількість споживаних ресурсів напряму впливає на апаратне забезпечення віртуальної машини, на якій розміщується сервер БД, а тому й на ціну цієї машини. З експериментів видно, що VoltDB витрачає менше за інші методи місця для збереження даних та затребує трохи менше процесорного часу, ніж MongoDB, в той час як при будь-якому режимі навантаження витрачає в декілька разів більше оперативної пам’яті.

Отже, менше використання пам’яті на диску з боку VoltDB дозволяє обрати цю СУБД у випадку:

- коли БД повинна зберігати десятки гігабайт даних, які при цьому не дуже часто або інтенсивно обробляються, коли швидкість цієї обробки не є пріоритетом. Прикладами таких ситуацій є зберігання бекапів БД або застарілих даних, що запитуються с дуже низькою вірогідністю та частотою. Це дозволить зекономити на розмірі постійного носія віртуальної машини;

- коли на ті й же машині зберігаються дані інших баз або просто інші файли. При цьому також швидкість обробки не повинна бути пріоритетом. Це також дозволить зекономити на розмірі постійного носія.

З іншого більш низьке споживання CPU ресурсів з боку VoltDB дозволяє обрати цей підхід у випадку, коли на одній віртуальній машині із сервером бази даних також розгортуються один або декілька додатків, діяльність яких потребує постійних обчислень з боку процесора та багатопоточної середи. Прикладами таких додатків є системи комп'ютерного зору, в яких для вирішення задачі класифікації необхідно багато процесорного часу. А також додатки, що знаходяться в постійному активному стані та споживають потоки та ресурси процесора, наприклад, антивірусні системи або пошукові боти. За допомогою даного підходу можна зекономити на ресурсах процесора, які коштують дорожче всього.

Але через менше споживання оперативної пам'яті більш доречно використовувати MongoDB у випадках:

- коли на ті й же віртуальній машині повинні розміщуватися один або декілька додатків, для постійної роботи яких потрібно багато оперативної пам'яті. Такими додатками в першу чергу є системи, які для більш швидкого реагування використовують різні рівні кешу, які зберігаються в RAM. Прикладом таких систем є веб-сервери, які кешують відповідь клієнту (особливо якщо це цілі веб-сторінки) задля прискорення відповіді;
- коли на ті й же віртуальній машині повинні розміщуватися інші сховища, для роботи яких необхідна велика кількість RAM. Прикладами таких сховищ є In-Memory бази даних, такі як Redis або Memcached.

Такі заходи дозволять при оренді віртуальних машин економити на відносно дорогій оперативній пам'яті.

Наступним кроком є розробка рекомендацій з урахуванням середнього часу виконання методами тієї або іншої транзакції. Виходячи з результатів замірів реалізація розподілених ACID транзакцій за допомогою VoltDB більш доречна у випадках:

– коли операції оновлення переважають над іншими операціями за критичністю з точки зору продуктивності та частотою виконання. VoltDB як показали заміри на порядок швидше виконує такий тип транзакцій, ніж інші методи. Прикладами таких систем є додатки спільного редагування документів, де кілька користувачів одночасно та постійно вносять правки в один і той же документ, а також додатки для фінансових бірж, де постійно оновлюються курси валют та ціни на акції компаній;
– коли операція видалення виконується з такою ж частотою, як і операція вставки, адже VoltDB показала час на декілька порядків менший для виконання складних операцій видалення. Також даний метод є пріоритетним, коли дуже важлива швидкість видалення даних. Такими системами є, наприклад, системи передачі секретних значень через одноразові сторінки, де одразу після першого зчитування даних вони повинні бути видалені, а також системи з показом різних даних та метрик у режимі реального часу, де застарілі за часом дані повинні якомога швидше видалятися, щоб не займати місце на диску;
– коли очікується велика кількість реплік у кластері. VoltDB показала, що фактор кількості вузлів-реплік у кластері не впливає на швидкість роботи більшості запитів. А для запитів, для яких все ж характерний ріст часу при збільшенні кількості реплік, то цей ріст мінімальний і коливається в районі кількох відсотків. Прикладом такого додатку з великою кількістю реплік є системи бронювання готелів по всьому світу, де сервери баз даних розміщуються в різних географічних зонах.

З іншого боку використання MongoDB для реалізації розподілених ACID транзакцій більш ефективно у випадках:

- коли операції зчитування більш пріоритетні, ніж інші операції на запис, адже MongoDB на порядок швидше виконує як прості запити на зчитування за індексом, так і складні запити з групуванням, сортування та лімітуванням. Якщо індексований пошук використовується у всіх додатках, то такі складні запити дуже часто використовуються в різних аналітичних системах (Azure Application Insights), системах обробки даних (Google Analytics), інтернет-магазинах (Amazon, Adorama), пошукових системах (Google, Bing);
- коли операції вставки переважають над операціями оновлення та видалення. Такий функціонал притаманний системам збору даних, які потім аналізуються. Зазвичай, в таких ситуаціях йде обробка початкових даних, які більше не змінюються. Прикладами таких додатків є системи централізованого логування даних, системи контролю різних метрик навколишнього середовища за допомогою Інтернету речей, бронювання квитків (Booking.com, Ryanair);
- коли в рамках ACID транзакції повинні виконуватися багато записів на вставку, оновлення або видалення. MongoDB швидше виконує та фіксує на всіх вузлах транзакції з декількома окремими запитами. При цьому чим більше цих запитів, тим більше різниця у часі порівняно з іншими методами. Існує багато систем з комплексними транзакціями, наприклад, банкові додатки з функцією переводу коштів, додатки для побудови різних графічних об’єктів, графіків та діаграм на основі числових даних (Draw.io, DBDiagram.io), а також додатки з функцією проходження тестувань та автоматизованого оцінювання знань користувачів (Moodle);
- коли критичною є швидкість будь-яких операцій зміни даних під високим навантаженням від користувачів. MongoDB дуже ефективно

справляється з великою кількістю одночасних підключень для більшості операцій зміни даних. Більшу частину таких запитів MongoDB виконує швидше за VoltDB та Two-Phase Commit, а для таких транзакцій, що все ж швидше виконуються в інших методах, все одно розрив з MongoDB менший, ніж для інших факторів навантаження. Прикладами таких додатків є системи електронної комерції, в яких існує пікове навантаження користувачів під час свят або системи проведення онлайн-змагань, де для участі в певний час повинні під'єднатися одразу сотні користувачів.

Також бувають ситуації, коли заздалегідь для додатка, що проектується, невідомо, який тип навантаження переважатиме для нього. Для таких випадків більш доречним буде використання MongoDB для реалізації розподілених ACID транзакцій, адже при одночасній зміні всіх факторів навантаження для переважної більшості запитів та транзакцій цей метод виконує їх швидше, ніж будь-який з інших, що розглядалися.

Таким чином на основі результатів порівняння різних аспектів продуктивності методів реалізації розподілених ACID транзакцій були розроблені практичні рекомендації щодо їх використання для конкретного типу додатку. Ці рекомендації можуть бути використані при проектуванні та розробці прикладного програмного забезпечення.

# ВИСНОВКИ

У результаті роботи була досліджена проблемна область розподілених транзакцій, а також різні методи реалізації ACID транзакцій для розподілених за допомогою реплікації баз даних. Серед них були обрані найбільш актуальні методи, для яких експериментально досліджувалася ефективність виконання різноманітних запитів та транзакцій.

Для дослідження була обрана прикладна предметна область бази даних електронної комерції, в рамках якої була розроблена діаграма зв’язків сутностей та промодельовані основні бізнес-операції цієї області. Також для кожного з методів була обрана певна СУБД, для якої в подальшому була спроектована структура або схема бази даних та розроблені запити для виконання. Крім цього було розроблено план розгортування різних вузлів розподіленого кластеру, на яких знаходяться сервери баз даних.

Під час дослідження було складено план експерименту, в рамках якого були виділені основні метрики, за якими порівнюється ефективність методів реалізації транзакцій, а також складені режими навантаження, для яких виконувалися подальші заміри.

Також для автоматизації процесу заміру ефективності методів реалізації розподілених транзакцій під різним навантаженням було спроектовано програмне забезпечення, яке дозволятиме в автоматичному режимі розгортати кластер розподіленої БД, заповнювати його даними, виконувати запити та заміряти різні метрики під час виконання.

У результаті для кожного режиму навантаження були отримані необхідні значення часу виконання запитів і транзакцій та метрик споживання ресурсів. На основі цих значень були витягнуті основні тренди щодо продуктивності обраних методів та на їх основі розроблені практичні рекомендації щодо використання того чи іншого методу. Ці рекомендації без проблем можуть бути впроваджені при проектуванні та розробці прикладного програмного забезпечення.

За результатами дослідження опубліковано тези «Дослідження методів реалізації ACID транзакцій для розподілених баз даних» у науково-технічній конференції «Інноваційні технології – 2020» та розроблено і подано до опублікування в науково-технічний журнал «АСУ та прибори автоматики» статтю «Дослідження методів реалізації розподілених ACID транзакцій за технологією реплікації».

## ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. M. Tamer Özsu. Principles of Distributed Database Systems. – Springer International Publishing, 2020. – 674 p.

2. Gerardus Blokdyk. ACID Transactions Second Edition. – 5STARCooks, 2018. – 282 p.

3. Anthony Molinaro. SQL Cookbook: Query Solutions and Techniques for Database Developers. – O'Reilly Media, 2005. – 877 с.

4. Matt Allen. Relational Databases Are Not Designed For Scale. URL: https://www.marklogic.com/blog/relational-databases-scale/ (дата звернення: 26.01.2021).

5. Kuzochkina A., Z. Dudar, Shirokopetleva M. Analyzing and Comparison of NoSQL DBMS. // International Scientific and Practical Conference «Problems of Infocommunications. Science and Technology» (PIC S&T-2018). 2018. P. 560.

6. Pramod Sadalage, Martin Fowler. NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence, 1st Edition. – Addison-Wesley Professional, 2012. – 192 p.

7. A B M Moniruzzaman and Syed Akhter Hossain. NoSQL Database: New Era of Databases for Big data Analytics – Classification, Characteristics and Comparison. // International Journal of Database Theory and Application. 2012. № 4. P. 1.

8. Daniel Abadi, Matt Freels. Achieving ACID Transactions in a Globally Distributed Database. URL: https://fauna.com/blog/acid-transactions-in-a-globally-distributed-database (дата звернення: 28.01.2021).

9. Sandeep Verma. Data Replication in distributed systems. URL: https://medium.com/@sandeep4.verma/data-replication-in-distributed-systems-part-1-13f52410faa3 (дата звернення: 28.01.2021).

10. M.Priya, R.Kalpana. Distributed processing of location based spatial query through vantage point transformation. // Future Computing and Informatics Journal. 2018. № 2. P. 296.

11. Saga distributed transactions. Документація. URL: https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga (дата звернення: 30.01.2021).

12. Saga distributed transactions. URL: https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga (дата звернення 30.01.2021).

13. Seata documentation. URL: http://seata.io/en-us/docs/overview/what-is-seata.html (дата звернення 31.01.2021).

14. Sami Bhir. Overview of Transactional Patterns : Combining Workflow Flexibility and Transactional Reliability for Composite Web Services. // Business Process Management. 2005.

15. Teresa Kwamboka Abuya. Handling distributed transaction failures: A practical approach. – LAP LAMBERT Academic Publishing, 2016. – 500 p.

16. Oleksandr Topchii, Oleksandr Samantsov, Oksana Mazurova, Mariia Shirokopetleva. A Study of Optimization Models for Creation of Artificial Intelligence for The Computer Game in The Tower Defense Genre. // Problem of Infocommunications. Science and Technology (PIC S&T′2020), Kharkiv, Ukraine. 2020.

17. Kristina Chodorow. Scaling MongoDB: Sharding, Cluster Setup, and Administration. – O'Reilly Media, 2011. – 66 p.

18. Chris Foot. NewSQL databases: The bridge between SQL and NoSQL. URL: https://searchdatamanagement.techtarget.com/feature/NewSQL-databases-The-bridge-between-SQL-and-NoSQL (дата звернення: 05.02.2021).

19. Pritam Roy. Distributed transactions and why you should care. URL: https://towardsdatascience.com/distributed-transactions-and-why-you-should-care-116b6da8d72 (дата звернення: 09.02.2021).

20. Leshchynskyi Volodymyr. Principles of explanation in e-commerce system based on sales dynamics. // COMPUTER AND INFORMATION SYSTEMS AND TECHNOLOGIES KHARKIV. 2020. P. 76.

21. Entity Relationship Diagram. Документація. URL: https://www.smartdraw.com/entity-relationship-diagram/ (дата звернення: 15.02.2021).

22. MongoDB Causal Consistency and Read and Write Concerns. Документація. URL: https://docs.mongodb.com/manual/core/causal-consistency-read-write-concerns/ (дата звернення: 19.02.2021).

23. Windows Virtual Machines Pricing. Документація. URL: https://azure.microsoft.com/en-us/pricing/details/virtual-machines/windows/ (дата звернення: 22.02.2021).

24. Natalia Kravets, Tseshkovskyi N., Liutova K. S. Containers and virtual machinesin microsoft Azure. // Science, society, education: topical issues and development prospects. Abstracts of the 7th International scientific and practical conference. 2020. P. 278.

25. Mark Richards, Neal Ford. Fundamentals of Software Architecture: An Engineering Approach 1st Edition. – O'Reilly Media, Inc., 2020. – 432 p.

26. Тимур Гуєв. Під капотом у Stopwatch. URL: https://habr.com/ru/post/226279/ (дата звернення: 05.03.2021).

27. About Performance Counters. Документація. URL: https://docs.microsoft.com/en-us/windows/win32/perfctrs/about-performance-counters (дата звернення: 05.03.2021).

28. Dudar Z.V., Falatuk H., Shirokopetleva M. Investigation of Architecture and Technology Stack for e-Archive System. // International Scientific-Practical Conference on Problems of Infocommunications Science and Technology. 2019. P. 229.

29. Guillaume Bonnot. The Pipeline Design Pattern — From Zero to Hero. URL: https://medium.com/@bonnotguillaume/software-architecture-the-pipeline-design-pattern-from-zero-to-hero-b5c43d8a4e60 (дата звернення: 07.03.2021).

30. Christos Matskas. Creating .NET fakes using Bogus. URL: https://cmatskas.com/creating-net-fakes-using-bogus-2/ (дата звернення: 11.03.2021).