МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

Факультет Комп'ютерних наук
Кафедра Програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

другий (магістерський)
(рівень вищої освіти)

Дослідження методів логічного моделювання NoSQL баз даних для ігрових серверних систем

Виконав:
Студент 2 курсу групи ІПЗм-20-3
Сиволовський І.М.
(прізвище, ініціали)

Спеціальність 121 – Інженерія програмного забезпечення

Тип програми Освітньо-наукова
Керівник доц. Мазурова О.О.
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри _____ проф. Дудар З.В.

2022 р.

Харківський національний університет радіоелектроніки

Факультет_Комп'ютерних наук_____
Кафедра_Програмної інженерії_____
Рівень вищої освіти - другий (магістерський)_____

Спеціальність _121-Інженерія програмного забезпечення_____
(код і повна назва)
Тип програми освітньо-наукова програма_____

Освітня програма_Інженерія програмного забезпечення _____

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___»_____ 20 ___ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові _ Сиволовському Іллі Михайловичу_____
(прізвище, ім'я, по батькові)

1. Тема роботи _Дослідження методів логічного моделювання NoSQL баз даних для ігрових серверних систем_____
затверджена наказом університету від "_24_" _03_2022 р № __412 Ст_
заповнюється вручну після отримання наказу

2. Термін подання студентом роботи до екзаменаційної комісії
20 травня 2022 р.

3. Вихідні дані до роботи *електронні ресурси за обраною тематикою, методи моделювання NoSQL баз даних, бази даних Neo4j, MongoDB нормалізована та денормалізована, середовище розробки Visual Studio 2022, клієнт Studio3t, мови Cypher, MQL, C#*

4. Перелік питань, що потрібно опрацювати в роботі *аналіз предметної та проблемної області, постановка задачі, аналіз методів логічного проектування, розробка логічних та фізичних моделей, розробка запитів для експериментів, розробка програмного забезпечення для виконання дослідження, проведення експериментів, аналіз отриманих результатів, розробка рекомендацій щодо використання досліджуваних методів.*

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз проблемної області дослідження	17.01.22 – 30.01.22	Виконано
2	Розробка постановки задачі	30.01.22 – 07.02.20	Виконано
3	Аналіз та вибір документних та графових NoSQL СКБД	07.02.22 – 14.02.22	Виконано
4	Аналіз методів логічного проектування	14.02.22 – 21.02.22	Виконано
5	Планування експериментального дослідження	21.02.22 – 25.02.22	Виконано
6	Аналіз та моделювання предметної області	25.02.22 – 01.03.22	Виконано
7	Розробка моделей баз даних на основі обраних методів	01.03.22 – 07.03.22	Виконано
8	Розробка запитів для експериментального дослідження	07.03.22 – 14.03.22	Виконано
9	Проектування та розробка ПЗ	14.03.22 – 21.03.22	Виконано
10	Проведення експериментів	21.03.22 – 01.04.22	Виконано
11	Підготовка пояснювальної записки	1.04.22 – 01.05.22	Виконано
12	Підготовка презентації та доповіді	1.05.22 – 15.05.22	Виконано
13	Нормоконтроль	18.05.22	Виконано
14	Рецензування	19.05.22	Виконано
15	Занесення диплома в електронний архів	20.05.22	Виконано
16	Попередній захист	20.05.22	Виконано
17	Допуск до захисту у зав. кафедри	21.05.22	Виконано

Дата видачі завдання __17_ _____січня_____ 2022р.

Студент _____ Сиволовський І. М.

(підпис)

Керівник роботи _____ доц. Мазурова О.О.

(підпис)

РЕФЕРАТ/ABSTRACT

Кваліфікаційна робота магістра містить: 113 с., 23 рис., 21 табл., 22 джер.

БАЗА ДАНИХ, ПРОЕКТУВАННЯ БАЗИ ДАНИХ, СКБД, NOSQL, MONGODB, BSON, NEO4J, CYPHER, C#, .NET.

Мета роботи – дослідження існуючих методів логічного моделювання NoSQL баз даних і порівняння їх ефективності під час вирішення завдань у сфері ігрових серверних систем.

Методи розробки та проектування базуються на платформі .NET 6 та мові програмування C#, СКБД MongoDB 5.0.5, Neo4j 4.4.3, середовищі розробки Visual Studio 2022.

В результаті роботи були досліджені методи логічного проектування NoSQL баз даних MongoDB та Neo4j, розроблені фізичні моделі БД та запити для кожного з них, проведене експериментальне дослідження та сформовані висновки щодо їх ефективності.

DATABASE, DATABASE DESIGN, NOSQL, MONGODB, BSON, NEO4J, CYPHER, C#, .NET.

The aim of the work – research of existing methods of logical modeling of NoSQL databases and comparison of their efficiency in solving problems in the field of game server systems.

Development and design methods are based on the .NET 6 platform and C# programming language, DBMS MongoDB 5.0.5, Neo4j 4.4.3, Visual Studio 2022 IDE.

As a result, various methods of logical design of NoSQL databases MongoDB and Neo4j were investigated, physical models of databases and queries for each of them were developed, an experimental study was conducted and conclusions were drawn about their effectiveness.

Умови публікації пояснювальної записки

Я, Сиволовський Ілля Михайлович, студент гр. ІПЗм-20-3, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Дослідження методів логічного моделювання NoSQL баз даних для ігрових серверних систем», 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

БД – база даних;

СКБД – система керування базами даних;

JSON – JavaScript Object Notation;

BSON – Binary JavaScript Object Notation;

API – Application Programming Interface;

ER – Entity Relationship;

ACID – Atomicity, Consistency, Isolation, Durability;

NPC – Non-Player Character;

MQL – MongoDB Query Language.

ВСТУП

Кількість даних у мережі Інтернет зростає з величезною швидкістю, тому що активні користувачі щомиті додають у соціальні мережі сотні гігабайт даних. Не дивно, що з такими сучасними масивами інформації реляційні бази даних не справляються. Хоча протягом кількох десятиліть вони успішно реалізовували завдання обробки інформаційних даних.

Ця проблема призвела до необхідності впровадження нових підходів щодо обробки інформації у великих системах. З цим завданням впоралися NoSQL бази даних, які дозволили замінити дороге вертикальне масштабування ефективним горизонтальним масштабуванням на кластерах. Також, вони мали вищу продуктивність, більш гнучку модель даних, а також відкритий вихідний код СУБД.

Особливе місце серед серверних систем займають ігрові сервісні системи через свою специфіку: складність реалізації, велике та нерівномірне навантаження. З вирішенням подібних завдань на боці бази даних зараз успішно справляються NoSQL БД зі своїми численними можливостями реплікації.

Але, без уніфікованих підходів до вибору бази даних та реалізації схеми, безліч розробників роблять помилки вже на стадії проектування системи, що може призвести до додаткових витрат та проблем далі.

Метою даної роботи є дослідження існуючих методів логічного моделювання NoSQL баз даних та порівняння їх продуктивності при вирішенні типових завдань у сфері ігрових серверних систем.

Під час виконання магістерської кваліфікаційної роботи було проведено аналіз проблемної області проектування NoSQL баз даних на основі публікацій світових та вітчизняних фахівців в проблемній області (див. додаток А).

На основі аналізу проблемної області були обрані цільові СКБД та методи логічного моделювання, і було проведено планування експериментального дослідження.

Також, було проведено аналіз і моделювання предметної області ігрових серверних систем, у результаті якого були розроблені фізичні моделі під цільові СКБД та набори запитів для проведення експериментального дослідження. Далі, було проведене експериментальне дослідження і на основі його результатів були створені рекомендації щодо використаня тих чи інших методів логічного проектування у контексті предметної області ігрових серверних систем.

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

На основі плану проведення дослідження та його результатів було розроблено презентацію (див. додаток В). За результатами роботи були створені тези доповіді на дванадцяту міжнародну науково-технічну конференцію «Сучасні напрями розвитку інформаційно-комунікаційних технологій та засобів управління», а також, підготовлено до подачі наукову статтю «Методи логічного проектування NoSQL баз даних для MongoDB та Neo4J» у журналі «Сучасний стан наукових досліджень і технологій в промисловості» (див. додаток Г).

Також, робота перевірена на відповідність вимогам оформлення (див. додаток Д).

1 АНАЛІЗ ПРОБЛЕМНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз проблемної області дослідження

Наразі людство переживає «інформаційний вибух». За п'ять попередніх років людством було зроблено інформації більше, ніж за попередню історію [1].

Реляційні бази даних, які вже більше 40 років використовуються в проектах різного профілю, вже не справляються з такими потоками даних. Також, це підсилює те, що вони сами по собі не дуже ефективно підтримують важливий у сучасності функціонал для великих даних: шардинг та реплікацію. Тому, в теперішній ситуації, для ефективного збереження неструктурованих і слабоструктурованих даних все частіше використовуються нереляційні бази даних NoSQL.

Швидке і широке розповсюдження баз даних NoSQL відбувається за рахунок простоти їх розробки в будь-яких масштабах, функціональності та продуктивності цих баз даних. Бази даних NoSQL зручно використовувати для багатьох сучасних додатків, що мають на меті застосування масштабованих баз даних, які мають високу продуктивність, широкі функціональні можливості, здатність забезпечення максимальної зручності використання. Це мобільні, ігрові, інтернет-додатки тощо.

Основними перевагами NoSQL баз даних є наступні:

- відсутність чіткої схеми та гнучкості, що призводить до більш швидкої розробки і забезпечує можливості з поетапної реалізації проектів;
- масштабованість: бази даних NoSQL не треба встановлювати на дорогі надійні сервери, бо вони розраховані на масштабування з використанням розподілених кластерів. Також, всі постачальники хмарних послуг реалізують такі операції у фоновому режимі для забезпечення повністю керованого сервісу;

- висока продуктивність: на відміну від реляційних, бази даних NoSQL дозволяють досягати більш високої продуктивності, бо вони є оптимізованими для конкретних шаблонів доступу та моделей даних;
- широкі функціональні можливості: бази даних NoSQL надають API, що мають широку функціональність і є спеціально розробленими для відповідних моделей даних.

В залежності від моделі даних NoSQL бази даних поділяються на кілька типів. До основних видів NoSQL баз даних належать наступні:

- бази даних типу «ключ/значення»: вони мають можливість зберігання данних у будь-якому форматі за певним ключем. Забезпечують високу розподільність і підтримують безпрецедентне горизонтальне масштабування, яке є недосяжним при використанні інших типів баз даних. Ці бази даних найчастіше зустрічаються в проектах з високим навантаженням на читання. І тому вони потребують кєшування блоків інформації. Прикладами таких СКБД є: Redis, Amazon DynamoDB;
- документоорієнтовані; це бази даних, які дозволяють розробникам за допомогою тієї ж документної моделі, що вони використали в коді програми, зберігати і запитувати дані в БД. Кожний збережений запис має вигляд окремого документа з своїм власним набором полів. Документи характеризуються гнучкістю і ієрархічністю, що дозволяє їм розвиватися у відповідості до зростаючих потреб додатків. MongoDB, CouchDB та Couchbase – це все приклади найпоширеніших документних баз даних. Вони мають на меті надання функціональних та інтуїтивно зрозумілих API для гнучкої розробки;
- графові бази даних – це бази, в яких представлення даних реалізується у вигляді вузлів та ребер, які є відношеннями між вузлами. Такі бази даних реалізують легку обробку складнопов’язаних даних і обчислення специфічних властивостей графів, таких як шлях від одного вузла до іншого та його довжина. До типових прикладів

використання графових баз даних відносяться соціальні мережі та сервіси;

– колонкові бази даних; це бази даних, де дані зберігаються у вигляді комірок, які згруповані не в рядки даних, як в реляційних базах, а в колонки. Ці колонки логічно групуються в сімейства, які мають практично необмежену кількість колонок. Для читання і запису використовуються колонки, а не рядки. Такі бази даних, як правило, застосовуються для зберігання та обробки аналітичної інформації. Google BigTable – це приклад першої колонкової СКБД для наступних сучасних колоночних баз даних, а саме: Cassandra, HBase та ClickHouse.

На даний момент найпопулярнішими NoSQL базами даних є документні, зокрема MongoDB, який стрімко наздоганяє популярні реляційні бази даних: Microsoft SQL Server, Oracle, MySQL і PostgreSQL [2, 3]. З цієї причини документні БД були обрані у якості об’єкта дослідження.

В загальному вигляді, проектування баз даних є процесом створення схеми бази даних, а також визначення необхідних обмежень цілісності [4]. До основних завдань проектування БД належать наступні:

а) забезпечення зберігання у базі даних всієї необхідної інформації;
б) зменшення дублювання і надмірності даних;
в) забезпечення можливості отримання інформації по всім потрібним запитам;
г) забезпечення цілісності даних, тобто виключення втрат даних і суперечностей.

Проектування баз даних включає чотири основних етапи.

Концептуальне проектування – процес створення концептуальної (семантичної) моделі предметної області, яка не спирається на СКБД або модель даних, яка вже існує.

Концептуальна модель має містити:

- описання інформаційних об’єктів системи та зв’язків, які виникають між ними;
- описання обмежень цілісності, тобто вимог, які висуваються до допустимих значень даних та зв’язків, які виникають між ними і так далі.

Для реалізації цього етапу проектування баз даних зазвичай використовують текстовий опис та нотації «вільного» вигляду, схожі на діаграму об'єктів UML. Якщо модель предметної області не дуже складна або велика, цей етап можна пропустити.

Наступний етап проектування БД – інфологічне проектування. Це процес створення ER-моделі предметної області. Модель може бути як у вигляді діаграми, так і в текстовому вигляді. Модель включає: сутності, їхні атрибути та ключі (первинні), зв’язки між сутностями та їх кардинальність.

Результатом проектування є ER-діаграма в нотації Чена [5], Баркера, «Crow’s Foot» [6] або IDEF1X.

Логічне або даталогічне проектування – це створення конкретної логічної моделі даних, а саме: реляційної, документної або графової моделі, але без урахування особливостей конкретних СКБД. Нерідко метою даного етапу є саме створення схеми бази даних. Щодо реляційної моделі баз даних – то це набір відношень з вказанням первинних ключів та їхніх зв’язків, графової моделі – набір вершин, зв'язків та їх атрибутів і так далі.

Фізичне проектування є процесом створення фізичної моделі бази даних вже для конкретної СКБД. Тобто на цьому етапі проектування вказується тип даних, що підтримується, створюються індекси, управління фізичним простором зберігання даних та інше. Результатом цього етапу проектування є діаграма та/або скрипт створення бази даних.

Якщо модель баз даних є реляційною, то вона повинна бути нормалізована згідно до нормальних форм [7]. Але, в контексті нереляційних

моделей даних, наприклад, документних або графових моделей, на практиці використовується зворотне перетворення – денормалізація [8].

Денормалізація – це процес модифікації бази даних, при якому зменшується порядок її нормалізації. Необхідність виконання цього процесу мотивується необхідністю підвищення продуктивності БД.

При цьому, в реальних додатках кількість операцій читання зазвичай набагато більше, ніж операцій на запис. Також, дані сутностей для потрібних операцій часто з’єднуються для зменшення кількості запитів, за допомогою оператора JOIN в реляційних СКБД [9] або його еквіваленту в NoSQL базах даних.

Процес денормалізації відбувається за допомогою наступного набору правил:

- об’єднання сутностей, що мають зв’язки «1:1»;
- дублювання неключових атрибутів в сутностях для зменшення кількості зв’язків;
- створення сутностей-агрегатів, що містять дані з інших сутностей.

При проектуванні бази даних для будь-якої NoSQL-системи від розробника вимагається чітке розуміння роботи бази даних та інструментів, які пропонує СКБД. Оскільки на практиці такого розуміння може не відбуватись, багато комерційних проектів не наважуються перейти на нові NoSQL бази даних, бо реалізація такого переходу потребує багато часу на моделювання продуктивності та міграцію інформації.

Також, досить часто, розробники не знають, як ефективніше моделювати схему (сутності, зв'язку) під певну модель даних, які індекси яких даних працюють ефективніше тощо.

Таким чином, дослідження методів та підходів логічного моделювання NoSQL баз даних є актуальним.

1.2 Постановка задачі

Метою цієї роботи є дослідження існуючих методів логічного моделювання NoSQL баз даних та порівняння ефективності використання даних підходів для вирішення задач предметної галузі ігрових серверних систем.

В ході виконання роботи треба виконати наступні завдання:

- проаналізувати існуючі NoSQL СКБД та обрати цільові СКБД для дослідження;
- провести аналіз методів логічного проектування для обраних NoSQL баз даних;
- провести планування експериментального дослідження методів для обраних СКБД, а саме:
    1) провести аналіз та моделювання предметної області ігрових серверних систем;
    2) розробити інфологічну модель бази даних (ER-діаграму) для експериментального дослідження;
    3) розробити логічні моделі для обраних СУБД, використовуючи різні методи логічного проектування;
    4) розробити запити для виконання експериментального дослідження.
- виконати реалізацію програмного забезпечення для проведення експериментального дослідження;
- провести експериментальне дослідження методів;
- розробити рекомендації щодо проектування NoSQL баз даних за результатами дослідження.

2 ОПИС ПРИЙНЯТИХ ПРОЕКТНИХ РІШЕНЬ

2.1 Аналіз та вибір документних та графових NoSQL СКБД для дослідження

На даний момент, існують наступні популярні документні СКБД, які можна використати для серверних додатків:

- MongoDB;
- DynamoDB;
- CosmosDB (у режимі документної бази даних);
- Couchbase.

Для вибору оптимальної бази даних для дослідження серед них використовуватимуться такі критерії:

- популярність (або розповсюдженість) СКБД – чим популярніша СКБД, тим більше інформації по ній можна знайти та більше шанс, що її підтримкою будуть займатися довше. Для визначення популярності СКБД, пропонується використання ресурсу db-engines.com.
- ліцензія – чи є СКБД open source проектом, або має commercial ліцензію.
- тип розгортання СКБД – чи є СКБД тільки хмарним рішенням або має опцію розгортання у хмарі;
- наявність драйверів під різні мови програмування;

Для зазначених критеріїв були обрані такі шкали:

а) популярність СКБД: оцінка з db-engines.com, від 0 до нескінченності (дані взяті на грудень 2021 року);

б) ліцензія:

1) open source та має enterprise версію – 4 бали;
2) open source – 3 бали;

3) частковий open source – 2 бали;
4) commercial (private) license – 1 бал.

в) тип розгортання СКБД:
1) standalone та в хмарі – 3 бали;
2) standalone – 2 бали;
3) тільки хмарне розгортання – 1 бал;

г) драйвери: оцінка від 1 до 5, яка показує охоплення мов програмування, відносно інших документних БД, що розглядаються.

Відсутність прив'язки до певного хмарного сервісу також є великою перевагою, тому що це робить архітектуру системи гнучкішою і дає можливість будувати свою інфраструктуру на окремому сервері (або набору серверів) під будь-які потреби проекту.

З точки зору ліцензії, open source бази мають перевагу над коммерційними через відкритість вихідного коду. Наявність опціональної enterprise версії СКБД з додатковою функціональністю щодо захисту даних та резервного копіювання є значною перевагою в великих проектах.

Наявність драйверів на багато мов програмування дає більшу гнучкість при виборі технологій при побудові серверної частини.

Таким чином, спираючись на ці критерії та їхні значення, можна побудувати таблицю документних СКБД зі значеннями відповідних критеріїв (див. табл. 1).

Таблиця 1 – Документні СКБД зі значеннями критеріїв

	Популярність	Ліцензія	Тип розгортання	Драйвери
MongoDB	484.67	Open Source + Enterprise	Standalone + хмарне	5
DynamoDB	77.63	Commercial	Тільки хмарне	2

Кінець таблиці 1

	Популярність	Ліцензія	Тип розгортання	Драйвери
CosmosDB	39.71	Commercial	Тільки хмарне	5
Couchbase	28.45	Open Source + Enterprise	Standalone + хмарне	3

Слід зазначити, що, на даний момент, всі представлені СКБД підтримують такі важливі концепції, як:

- підтримка шардування даних;
- підтримка реплікації (source-replica, multi-source);
- підтримка ACID транзакцій.

Саме тому, ці елементи функціональності БД не були винесені в критерії.

Враховуючи вид вхідних даних, вибір кращої СУБД може бути зведений до задачі лінійного програмування. Задачі ЛП можна вирішити безліччю різних методів, в залежності від характеру задачі. Найбільш популярними є згорткові методи на базі теорії корисності.

Тому, для вирішення задачі, було обрано лінійну згортку. Так як одні параметри є важливішими за інші і мають різні розмірності, має сенс використовувати лінійну адитивну згортку з ваговими коефіцієнтами. Дана згортка універсальна і підходить під різні види задач [10]:

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

$$\alpha_j = \frac{1}{\sum_{i=1}^{m} a_{ij}}$$

де α_j – нормуючі множники;

β_j – вагові коефіцієнти.

Слід зазначити, що використання нормуючих множників прибирає потребу попередньої нормалізації вхідних даних.

Далі, треба визначити ваговий коефіцієнт під кожний критерій. Для вирішення цієї проблеми існує багато методів, наприклад, експертна оцінка.

Основна вимога – всі коефіцієнти повинні бути більшими за 0, а їх сума повинна дорівнювати 1.

У нашому випадку, добре підходить пропорційний метод. Тобто, приймемо, що популярність бази і тип розгортки в 2 рази важливіші, ніж доступні драйвери та ліцензія. За цими правилами, розраховуємо вагові коефіцієнти:

- популярність БД: 0.33;
- ліцензія: 0.17;
- тип розгортання: 0.33;
- драйвери: 0.17.

Використовуючи коефіцієнти, побудуємо таблицю нормалізованих значень критеріїв з урахуванням вагових коефіцієнтів та нормуючих множників (див. табл. 2).

Таблиця 2 – Документні СКБД з значеннями критеріїв у числовому форматі

	Популярність	Ліцензія	Тип розгортання	Драйвери	Z
MongoDB	1	1	1	1	0.5
DynamoDB	0.11	0	0	0	0.12
CosmosDB	0.02	0	0	1	0.14
Couchbase	0	1	1	0.33	0.24

Кінець таблиці 2

	Популярність	Ліцензія	Тип розгортання	Драйвери	Z
Вагові коефіцієнти	0.33	0.17	0.33	0.17	
Нормуючий множник	0.00157	0.1	0.125	0.0667	

Останній стовпець таблиці містить результати функції згортки. Як видно з результатів, максимальне значення отримала СКБД MongoDB. Тому, можна зробити висновок, що вона є найкращою альтернативою для дослідження ефективності фізичних моделей, побудованих на основі логічних моделей, з використанням різних методів проектування баз даних.

Якщо розглядати графові бази даних, кількість можливих варіантів СУБД доволі низька. На даний момент існують такі графові БД:

- Neo4j;
- CosmosDB (у режимі графової бази даних);
- Amazon Neptune;
- ArangoDB;
- OrientDB;
- TigerGraph;

Але, слід зазначити, що тільки Neo4j, Amazon Neptune і TigerGraph мають підтримують суто графову модель, решта СКБД є мультимодельними. За підтримки багатьох видів моделей даних, не гарантується необхідна продуктивність та функціонал СКБД. Також, Amazon Neptune є пропрієтарною БД, яка може бути розгорнута лише у хмарі, що може стати перешкодою для багатьох розробників. Багато СКБД зі списку, такі як ArangoDB, OrientDB і TigerGraph є ще досить новими розробками і досі розвиваються, що робить їх ризикованим вибором для production рішень.

У такій ситуації, тільки Neo4j є прийнятним вибором, якщо серверному додатку потрібна швидка робота з даними, які можна подати у вигляді графа.

2.2 Аналіз методів логічного проектування під СКБД MongoDB

Як було згадано, процес проектування бази даних складається з 4 етапів. Кожен із етапів приносить певний артефакт, на якому далі ґрунтується наступний етап проектування. Перед логічним проектуванням відбувається інфологічне, тобто, побудова логічної моделі ґрунтується на ER-діаграмі бази даних.

Алгоритми переходу від ER-діаграм до логічної моделі у контексті реляційних БД вже давно формалізовані. Проте, дані алгоритми не застосовні до NoSQL баз даних, які засновані на інших структурах даних, ніж таблиці (відносини).

Але, навіть при такому розкладі, способи моделювання сутностей і зв'язків реляційних баз даних можна накласти на NoSQL бази, хоча і з поправками та доповненнями [11].

Традиційно, документні бази даних не передбачають зв'язки у реляційному розумінні, тому реалізація подібного функціоналу та цілісності даних цілком лежить на розробників серверної (або клієнтської, якщо це локальна база даних) частини, що використовують базу даних.

Для створення схожого зі зв'язками функціоналу, використовується метод «Manual reference», який передбачає збереження поля «_id» одного документа в полі іншого об'єкта, подібно до зовнішнього ключа в реляційних БД, але без самого зв'язку (див. рис. 2.1) [12].

За допомогою даного методу, формується зв'язок 0:M, який вже розробниками може використовуватися як зв'язки 1:1, 1:M та похідні від них.

Таким чином, використовуючи додатковий запит, можна отримати інший документ, на який посилається поточний. Але, це призводить до проблеми «N+1», бо потребує додатковий запит або приєднання даних через JOIN-подібну операцію.

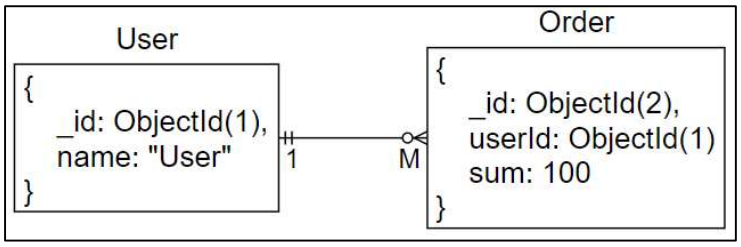

Рисунок 2.1 – Метод «Manual reference»

Для вирішення подібних проблем документні БД пропонують використання композиції у вигляді вкладених об'єктів або масивів об'єктів (див. рис. 2.2).

Рисунок 2.2 – Метод композиції об’єктів

Цей підхід добре підходить, якщо зв'язок між об'єктами можна висловити словом «включає».

Таким методом можна моделювати зв'язки:

– 1:1, але потрібно враховувати, що вкладений об'єкт збільшить вагу документа, що уповільнює його вивантаження з БД на клієнт;

– 1:M, але якщо M є не особливо великим числом та вкладені об’єкти не дуже великі.

Але, при такому підході треба пам'ятати, що:

– максимальний розмір вкладеності – 100 рівнів;
– максимальний розмір документа – 16 МБ;
– якщо в полі документа (масив) постійно додаються нові записи, то розмір документа постіно зростатиме. Це може викликати проблеми з продуктивністю через переміщення документа в іншу ділянку пам'яті, тому що йому вже нема куди зростати в поточній ділянці (дефрагментація).

Відсутність зв'язків також означає відсутність JOIN у реляційному розумінні. Але, згодом, MongoDB додали два способи приєднання даних:

– $lookup – операція, що працює аналогічно LEFT OUTER JOIN в реляційних БД (додана у версії 3.2);
– $graphLookup – створює колекцію записів, які показують ієрархію об'єктів від деякого до поточного, подібно до пошуку в графових БД (доданий у версії 3.4).

Проте, слід зазначити, що ці методи мають потенційно низьку продуктивність, бо виконують додаткові запити на потрібні дані. Тому в контексті документних БД композиція є більш пріоритетною. Але, дуже часто без цього функціоналу не можна обійтися, тому необхідно створити правильний індекс, щоб замінити етап «COLLSCAN» (повний перебір) на «IXSCAN» (пошук за індексом). Таким чином, якщо проводити аналогію з реляційними СКБД, ми перетворюємо найповільніший тип JOIN (LOOKUP) на швидкий MERGE JOIN.

Також, в MongoDB була додана функціональність, яка має зовнішню схожість зі зв'язками – DBRef. Але, ця функціональність не є зв'язком, це допоміжна функція для драйверів клієнта, яка вказує, що в цьому полі можливий зв'язок 1:M, тому можливо зробити додатковий запит на вилучення

сутностей. Проте, цей функціонал має досить низьку продуктивність, підтримується не всіма драйверами і призначений більше для «зв'язування» у вкладеному масиві сутностей з різних колекцій, бо посилання містить ім'я колекції, та, опціонально, ім'я бази даних.

З сутностями, що мають зв’язок M:M все складніше. Відомо, що зв'язок M:M можна виразити у вигляді двох зв'язків 1:M та проміжного об'єкта, який містить ідентифікатори тих двох об'єктів, що на нього посилаються. Цей підхід може бути реалізований у MongoDB без особливих проблем, крім створення індексів на поля з ідентифікаторами. При такому підході всі допоміжні атрибути зв'язку M:M будуть розташовані в окремому об'єкті.

Але, якщо розробнику потрібно з'єднати дані такого зв'язку, то йому доведеться використовувати дві JOIN-подібні операції ($lookup), що в контексті документних БД дуже дорого.

Для вирішення подібної проблеми, практично завжди, в MongoDB використовується інший підхід: композиція даного проміжного об'єкта в один з об'єктів зв'язку M:M у вигляді масиву. На рисунку 2.3 зазначені обидва підходи: через допоміжну сутність (зверху) та скорочений підхід з композицією (знизу).

Широке використання другого підходу пов'язана з тим, що в 2017 році операція $lookup отримала підтримку використання масивів ідентифікаторів як вхідні дані для приєднання даних. При такому підході для з'єднання даних необхідна всього одна JOIN-подібна операція замість двох. Але, при його використанні, необхідно виділити «головний» об'єкт із зв'язку M:M, який міститиме масив ідентифікаторів.

По суті, метод «Manual reference» робить схему більш подібною до реляційних БД, тому його можна позначити як «нормалізуючий» метод. Метод вкладеного документа, навпаки, знижує рівень нормалізації через композицію, отже його можна назвати «денормалізуючим». Вищезазначений метод

проектування зв’язку M:M є «змішаним», з нахилом до «нормалізуючого», бо використовує обидві концепції, але все ще залежить від JOIN-подібної операції.

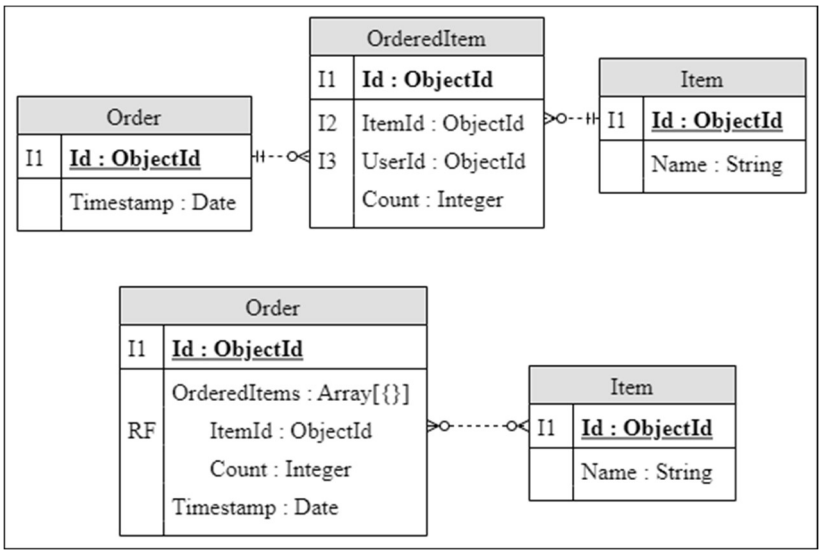

Рисунок 2.3 – Методи проектування зв’язку M:M в документних БД

Таким чином, однією з цілей дослідження є порівняння ефективності нормалізованої та денормалізованої моделі, що були створені за допомогою цих методів.

2.3 Аналіз методів логічного проектування під СКБД Neo4j

Тепер розглянемо методи логічного проектування під графові СКБД, зокрема Neo4j. У цій СКБД база даних є одним великим графом без поділу сутностей за типом (як таблиці чи колекції). Цей граф складається з: вершин (сутностей) і ребер (зв'язків). На відміну від документних, графові бази даних підтримують зв'язки, хоч вони значно відрізняються від реляційних зв'язків.

У Neo4j кожен зв'язок є сутністю спеціального типу, яка зберігає посилання на вихідну та вхідну сутність. Таким чином, зв'язки мають свої імена, можуть містити атрибути та на них можуть бути створені індекси.

Як і всі бази даних NoSQL, дана СУБД не має механізмів обмеження цілісності, це має вирішувати лише розробник на рівні додатку. Але, кожен зв'язок у графі обов'язково повинен мати вихідну та вхідну сутності.

Звідси випливає, що кожен зв'язок Neo4j за замовчуванням має кардинальність 1:M, яку можна через обмеження на унікальність або на програмному рівні «перетворити» на зв'язок 1:1.

Таким чином, зв'язок M:M можна потенційно промоделювати двома способами: через допоміжну сутність (як у реляційних БД) і безпосередньо, зберігаючи додаткові дані як атрибути зв'язку. На рисунку 2.4 ці методи моделювання проілюстровані графічно: як це виглядає у реляційних БД (зверху), через допоміжну сутність (посередині) та через атрибути зв’язку (знизу).

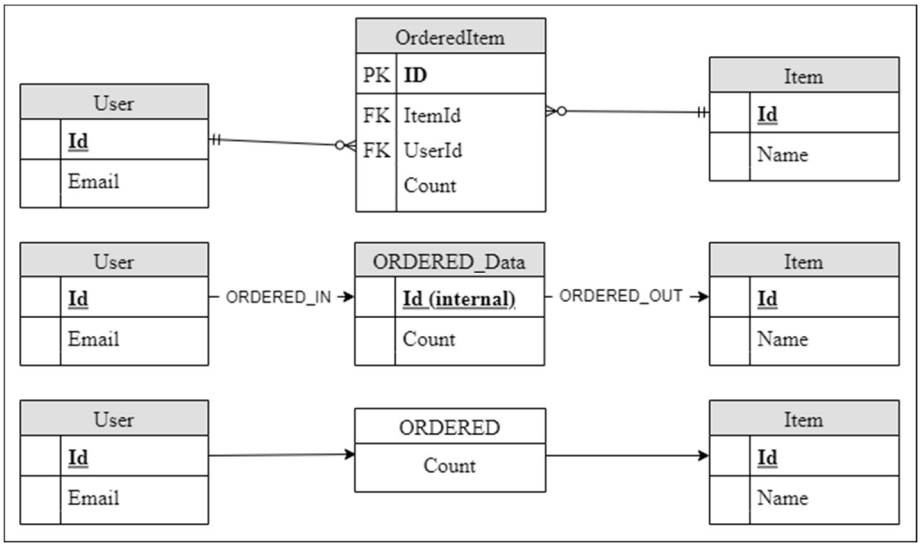

Рисунок 2.4 – Порівняння методів проектування зв’язку M:M в графових СКБД

Слід зазначити, що практично всі графові СКБД мають лише односпрямовані зв'язки. Стандартизована мова запитів Gremlin, яку

підтримують всі графові СКБД, також не має підтримки двонаправлених зв'язків. Таким чином, для створення двонаправлених зв'язків необхідно зробити два зв'язки в обидві сторони.

Але, якщо враховувати, що зв'язок теж є одним із об'єктів СКБД, то варіант із проміжною сутністю є неефективним з самого початку, бо сильно засмічує БД зайвою сутністю та зв’язками, збільшує вагу БД за рахунок зайвих об'єктів і потенційно збільшує час виконання навіть базових запитів.

Таким чином, у подальшому порівнянні методів логічного проектування прийматиме лише «традиційний» метод для СКБД Neo4j. Дане порівняння має показати різницю щодо ефективності використання документної та графової СКБД та у яких ситуаціях їх використання найефективніше.

2.4 Планування експериментального дослідження

2.4.1 Аналіз та моделювання предметної області

Впродовж майже 50 років існування відеоігор було створено безліч різних жанрів, підходів до реалізації ігрових механік та ігрового процесу. З поширенням загальнодоступності Інтернету, ігрові студії почали створювати все більше ігор з опцією багатокористувацького геймплею, крім одиночного [13].

Пізніше, з розвитком мобільного Інтернету, з'явилися ігри, розраховані тільки на багатокористувацькій геймплей та прив'язку до Інтернету. На даний момент, саме ці ігри є найбільш прибутковими та популярними, а поширеність ігор у житті людини тільки зростає. Саме тому як предметну область було обрано ігрові серверні системи.

Слід зазначити, що, в залежності від жанру гри та її основних механік, сильно відрізняється внутрішня реалізація мультиплеєра. Це можуть бути як

засновані на Peer-to-Peer реалізації, якщо гра має обмежену кількість гравців в ігровій сесії та змагальний характер, так і різні реалізації з виділеним сервером [14].

Також, вже існують готові реалізації для мультиплеєра, такі як Photon Unity Networking для ігрового рушія Unity або різноманітні модулі для рушія Unreal Engine, але через їхню ціну, складність інтеграції в гру або прив'язки до їх серверів, не всі студії віддадуть перевагу їх використанню власним розробкам [15].

У якості об’єкта предметної області виберемо розраховану на багато користувачів гру жанру action-adventure з елементами RPG та виділеним сервером.

У грі подібного жанру та реалізації мультиплеєра, у будь-якому випадку необхідно реалізувати базу даних для зберігання стану світу та прогресу гравця. База даних має зберігати таку інформацію:

- інформацію про обліковий запис гравця (валюта, дані гравця, інформацію про його входи в ігру);
- стан та інформація про його персонажів в ігровому світі, їхні здібності;
- повний список завдань, завдання персонажа та стан їх виконання (етап);
- список противників (монстрів) у грі та пов’язана з ними інформація (або інформація про їхнє місцезнаходження, якщо їх генерує серверна частина);
- список неігрових персонажів (NPC) та пов’язана з ними інформація (ім’я, координати, завдання, що вони видають);
- історія подій у грі (купівля внутрішньоігрової валюти, перемога над супротивниками, виконання завдань та ін.).

На основі цієї інформації складемо загальну діаграму класів предметної області, яка описує сутності ігрової системи та взаємозв'язок між ними (див. рис. 2.5).

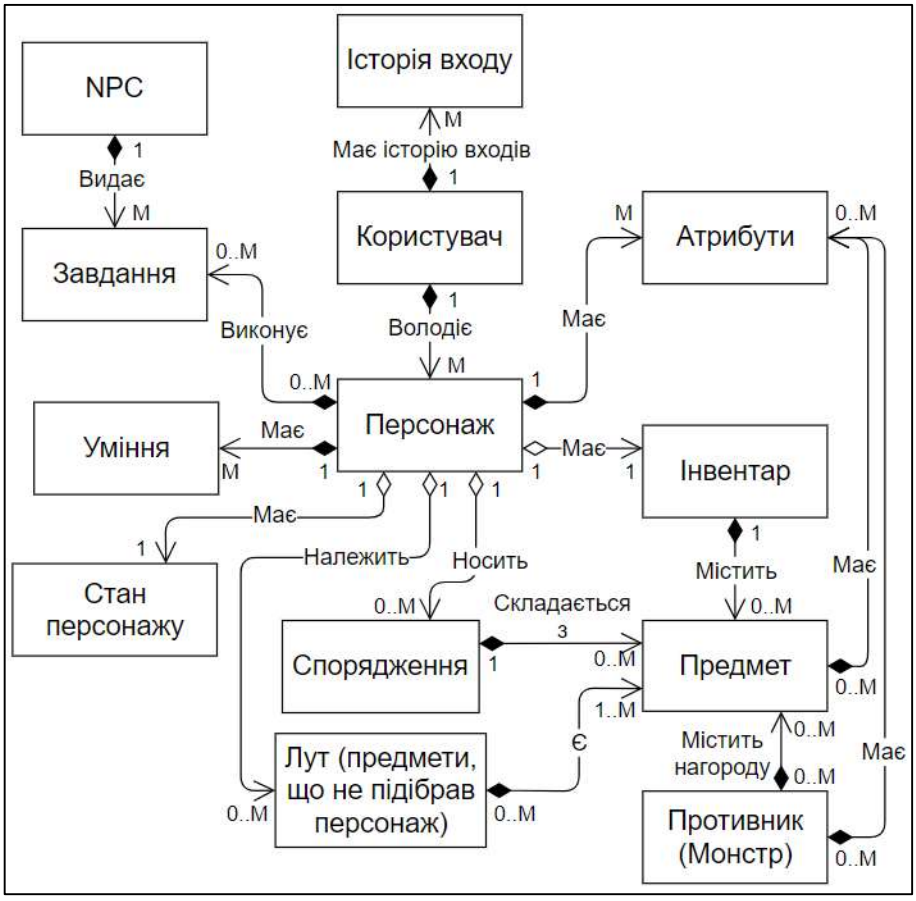

Рисунок 2.5 – Загальна діаграма класів предметної області

На основі даної схеми, обмежень цілісності та атрибутів складемо ER-діаграму бази даних [16], яка відповідає етапу інфологічного проектування баз даних (див. рис. 2.6).

Для побудови діаграми будемо використовувати нотацію «Crow's foot»[17], бо вона використовується частіше за інших нотацій і краще візуально сприймається.

Перелічені вище дані є базовими, всі можливі інші сутності та його властивості залежать від деталей ігрових механік гри, що розробляється. Але для моделювання предметної області з метою дослідження продуктивності

достатньо даних сутностей, тому що вони описують абстрактну гру подібного жанру і механік.

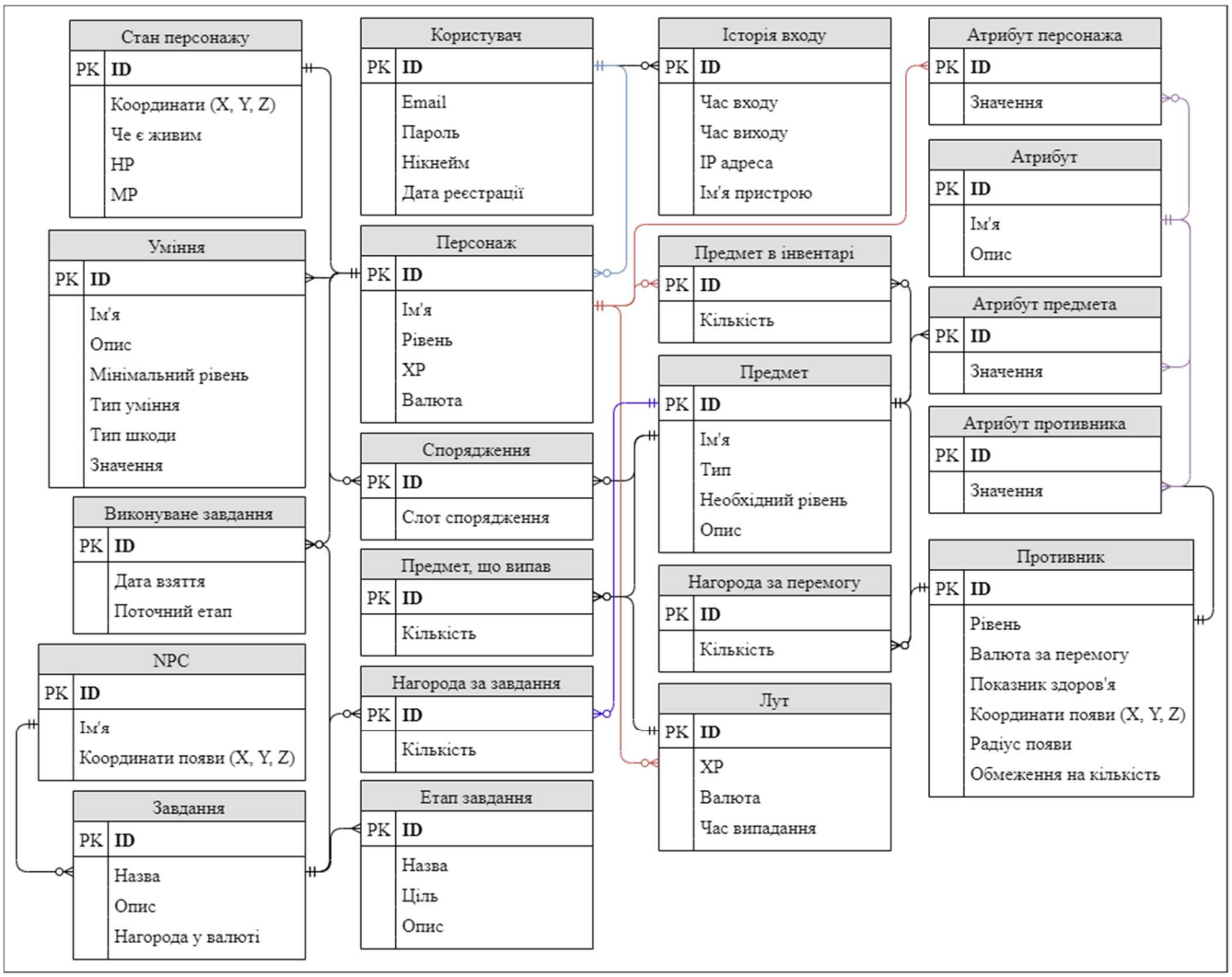

Рисунок 2.6 – ER-діаграма бази даних предметної області

Також, варто уточнити, що в ER діаграмі опущені поля, які вказують на внутрішні ідентифікатори в грі (імена моделей, скриптів і т.д), оскільки це залежить від рушія реалізації гри і дослідження не стосується.

Таким чином, була розроблена ER-діаграма бази даних, яка відповідає предметній області деякої багатокористувацької гри.

2.4.2 Розробка логічних моделей на основі обраних методів

Тепер на основі ER-діаграми слід створити логічні моделі бази даних, по одній на кожний метод проектування: графова, документна нормалізована та документна денормалізована.

Щоб перетворити ER-діаграму на певну логічну модель, необхідно керуватися деяким алгоритмом або набором правил. Деякі правила перетворення притаманні всім алгоритмам, що будуть розглянуті далі, наприклад:

- видалення дублікатів атрибутів в сутностях, якщо є;
- перетворення сутностей ER-діаграми в сутності логічної моделі бази даних;
- додавання індексів на первинні ключі.

Основна відмінність даних алгоритмів полягає у різному перетворенні зв'язків між сутностями: 1:1, 1:M та M:M.

Насамперед спроектуємо графову логічну модель. Стандартизованої нотації для побудови логічної моделі даного типу на даний момент немає, тому буде використовуватися модифікація реляційної. Слід зазначити, що Neo4j має підтримку атрибутів у зв'язках, що може значно скоротити кількість сутностей і спростить модель.

Алгоритм перетворення ER-діаграми в графову логічну модель сильно відрізняється від документних через іншу структуру зберігання даних. Він має наступні кроки:

- об’єднати в одну сутність ті сутності, що мають зв’язок 1:1 між собою. Слід пам’ятати, що Neo4j вкладені об’єкти не підтримує;
- перетворити зв’язки 1:M на графові зв’язки без атрибутів;
- замінити проміжні сутності, що створюють зв’язок M:M, на графові зв’язки (з атрибутами, якщо є).

На рисунку 2.7 зображено графову логічну модель, отриману в результаті проектування.

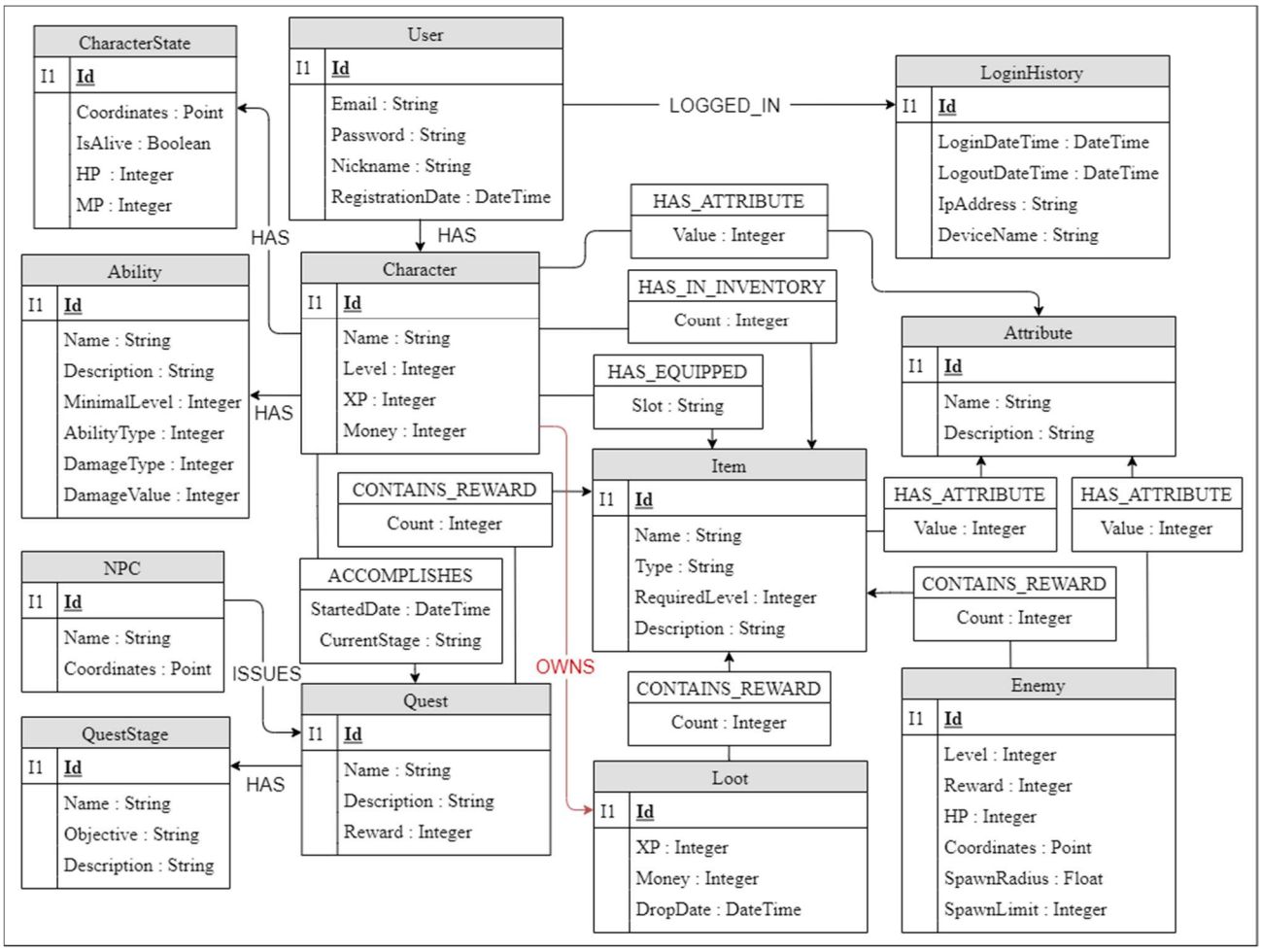

Рисунок 2.7 – Графова логічна модель бази даних предметної області

У моделі фігурують два види зв'язків: з атрибутами та без них. Для відображення зв'язків з атрибутами було виділено окрему нотацію у вигляді прозорого блоку.

Використання атрибутів зв'язків дозволило усунути всі сутності, які використовувалися для моделювання зв'язку M:M, що значно скоротило модель. Але через те, що графові БД не підтримують вкладеність об'єктів, зв'язок 1:1 необхідно підтримувати на програмному рівні, наприклад, зв'язок між персонажем і його станом [18].

Тепер, спроектуємо документну нормалізовану логічну модель. Слід зазначити, що вона є найбільш подібною до вихідної ER-діаграми. Для побудови цієї моделі у графічному вигляді також немає стандартизованої нотації. Ускладнює процес також те, що MongoDB об'єкти можуть мати до 100 рівнів вкладення і це проблематично відобразити в графічному вигляді, хоч це і трапляється на практиці не дуже часто. Тому для опису документної логічної моделі пропонується модифікація нотації для реляційних логічних моделей з додатковим функціоналом, який властивий документним БД.

Спроектована нормалізована логічна модель зображена на рисунку 2.8.

На рисунку позначкою RF зображені всі «умовні» зовнішні ключі, що були побудовані за методом «manual reference». Отримані зв'язки несуть суто умовний характер через те, що MongoDB не має механізмів обмеження цілісності та функціоналу «зв'язування» документів загалом. Завдання контролю цілісності даних повністю лежить на розробникові.

Алгоритм перетворення ER-діаграми в нормалізовану документну логічну модель є схожим до аналогічного алгоритму в контексті реляційних БД.

Алгоритм має наступні кроки:

- сутності, які мають зв’язок 1:1: додати поле з ідентифікатором одного документа в інший документ (залежний);
- сутності, які мають зв’язок 1:M: додати поле з ідентифікатором головного документа (1) у залежні (M);
- сутності, які мають зв’язок M:M мають використовувати один з зазначених раніше методів: або через виділення проміжної сутності з ідентифікаторами об’єктів, що посилаються на нього (неефективно), або через композицію цього проміжного об’єкта в «головному» об’єкті у вигляді масиву (ефективно).

Тепер спроектуємо денормалізовану документну логічну модель. Оскільки вже існує нормалізована модель, то цю модель можна спроектувати

двома способами: денормалізуючи існуючу нормалізовану модель та через перетворення з ER-діаграми.

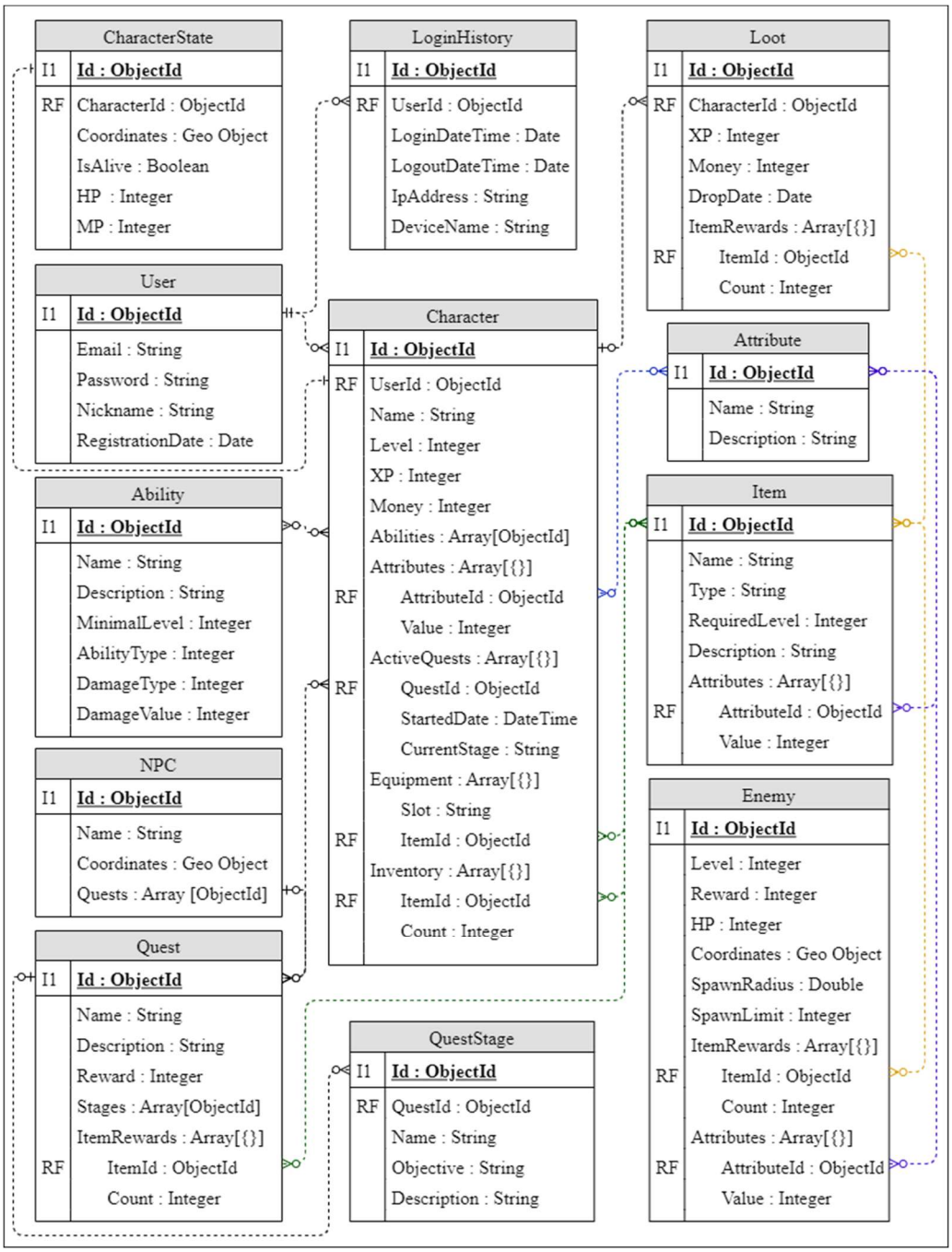

Рисунок 2.8 – Нормалізована документна логічна модель БД предметної області

Кроки по денормалізації моделі в контексті документних БД майже не відрізняються від реляційних:

- злиття сутностей зв'язків 1:1 в одну сутність (тільки в документних БД ще можна створити вкладений об'єкт);
- дублювання даних заради зменшення кількості зв'язків та ін.

Проте, у роботі розглядається саме процес логічного моделювання, тобто. перехід від ER-діаграми до логічної моделі. Тому розглянемо цей алгоритм для денормалізованої моделі:

- сутності, які мають зв’язок 1:1: створити в головному документі поле та вкласти в нього залежний документ із подальшим видаленням залежної сутності (або вкласти його поля в головний документ). Рекомендується додати до нього індекс, якщо планується вибирати ці сутності окремо від головного об’єкта;
- сутності, які мають зв’язок 1:M: додати поле-масив у головний документ (1), яке буде містити всі залежні (M) об’єкти. Якщо семантично головний об’єкт може не мати зв’язків до усіх залежних об’єктів (немає зв’язку «володіє»), то є необхідність створити окрему колекцію без зв’язків, що буде містити всі екземпляри залежних об’єктів.
- сутності, які мають зв’язок M:M: додати поле-масив у головний документ, яке містить всі залежні документи. Випадки, у яких потрібно створювати додаткову колекцію аналогічні з 1:M.

Керуючись цим алгоритмом, була спроектована денормалізована документна логічна модель предметної області (див. рис. 2.9).

Дана модель має максимально можливу ступінь денормалізації в контексті даної системи.

Єдина сутність, що не зазнала денормалізації – це Item. Це пов’язано з тим, що у ігрових додатках подібного жанру звернення до всіх предметів здійснюється за його ідентифікатором та набір всіх предметів вивантажується під час запуску гри.

Таким чином, JOIN-подібні операції із сутністю Item проводитися на практиці не будуть та різниці в продуктивності теж не буде.

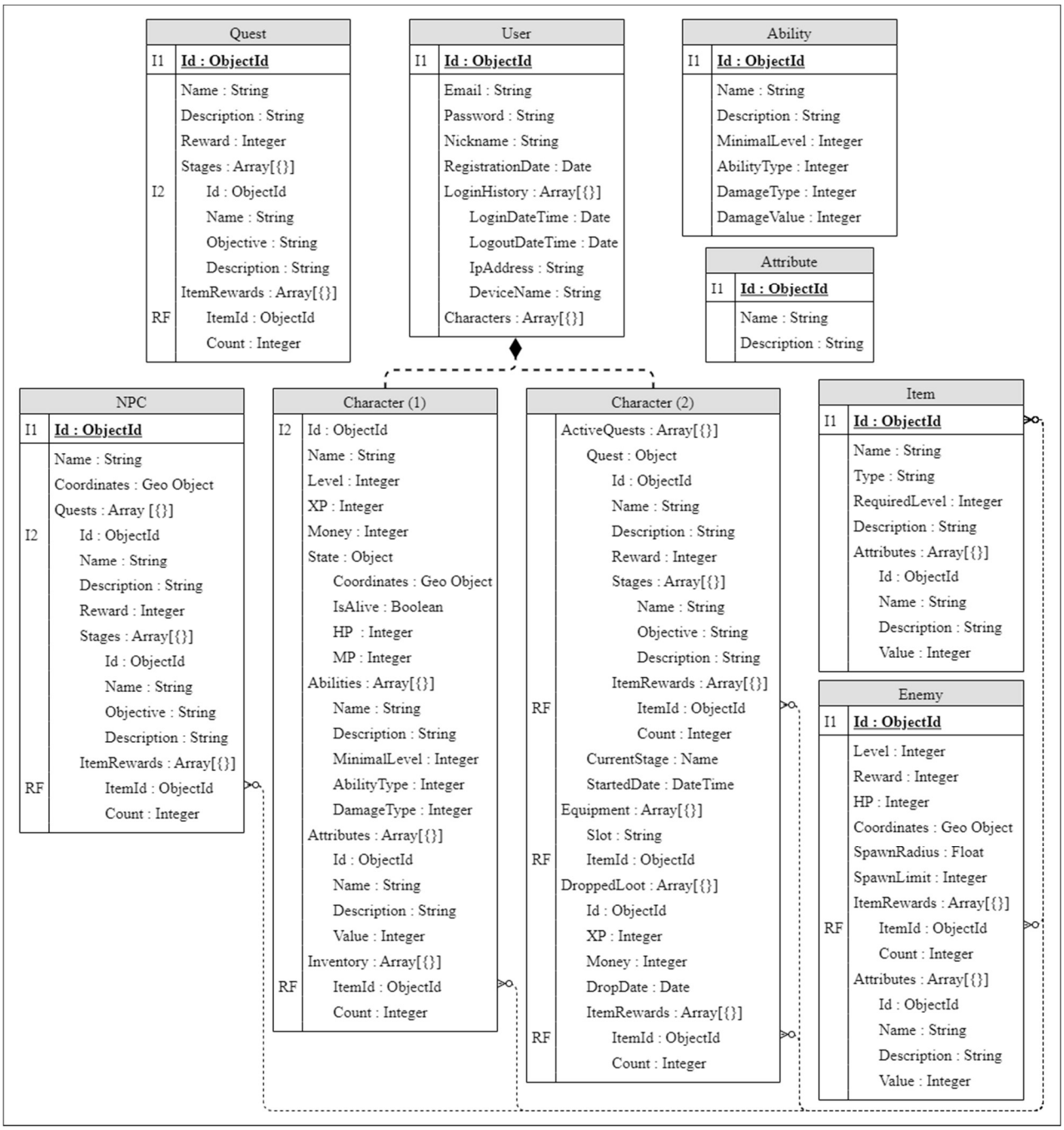

Рисунок 2.9 – Денормалізована документна логічна модель БД предметної області

Для зручнішого візуального відображення діаграми, вкладений об'єкт Character був винесений окремо і розбитий на дві частини.

Для позначення вкладеності Character в User був використаний зв'язок із діаграм класів UML «включає».

Таким чином, в результаті аналізу та моделювання предметної області були розроблені логічні моделі: нормалізована та денормалізована документні та графова. На основі цих моделей буде спроектовано фізичні моделі БД.

2.4.3 Вибір критеріїв та обмежень для експериментального дослідження

Перед початком експериментального дослідження варто описати обмеження експерименту: апаратні конфігурації серверів БД та тип розміщення СКБД.

Почнемо з типу розміщення серверів БД, або виду розподіленості БД. Для дослідження використовуватимуться кластери із серверів БД, чи реплікація типу «source-replica», бо такий підхід добре підходить ігрових серверів, які мають специфіку великого читання [19].

Таким чином, всі виміри будуть виконані на кластерах серверів баз даних, незалежно від конфігурації. Розташовуватимуться вони у хмарному сервісі Azure на віртуальних машинах різного розміру (див. табл. 3) [20].

Таблиця 3 – Конфігурації серверів БД для експериментів

Тип конфігурації	**Small**	**Medium**	**Large**
Назва машини	Standard_B2s	Standard_B4ms	Standard_B8ms
vCPU	2	4	8
RAM (ГБ)	4	16	32
Кількість вузлів	2	4	6
Кількість підключень	20	50	100

В якості операційної системи буде використовуватися Ubuntu 20.04 LTS Minimal для мінімізації споживання ресурсів системою.

Також тип конфігурації машини впливає і на кількість сутностей в БД, які будуть використовуватися в експериментах (див. табл. 4).

Ґрунтуючись на іграх аналогічного жанру, деякі сутності не можуть бути у великій кількості і не змінюватимуться залежно від конфігурації, наприклад:

– Attribute (обрано константну кількість – 6);
– CharacterState (одна сутність на кожного персонажа).

Тепер слід виділити метрики, які ми порівнюватимемо під час проведення експерименту. При виконанні кожного етапу експериментального дослідження збиратимуться дані метрики:

– S – місце, яке займає БД на диску (МБ);
– T – час виконання запиту (мс);
– M – споживання оперативної пам'яті (МБ);
– C – споживання процессорного часу (%).

Таблиця 4 – Кількість сутностей БД для експериментів за конфігураціями

Тип конфігурації	**Small**	**Medium**	**Large**
Користувачі / персонажів у користувача	3000 / 1	5000 / 2	8000 / 3
Кількість входів в гру на користувача	25	50	75
Предметів усього / в інвентарі гравців	100 / 50	200 / 100	300 / 150
Уміння / слотів спорядження	10 / 4	25 / 6	50 / 8

Кінець таблиці 4

Тип конфігурації	**Small**	**Medium**	**Large**
NPC / завдання, що вони видають	50 / 100	150 / 200	300 / 500
Противники	100	200	500
Лут / кількість предметів в ньому	5000 / 3	15000 / 5	30000 / 8
Кількість об’єктів БД у «найгіршому» випадку (денормалізація)	776 967	4 701 983	16 362 659

Також всі запити необхідно виконувати на «прогрітій» базі даних, тобто з виділеними у внутрішній кеш даними.

Відповідно, одним із завдань дослідження буде визначення залежності зміни представлених метрик від розміру датасета та конфігурації обладнання, на яких запущено кластер бази даних.

2.4.4 Розробка запитів для експериментального дослідження

На відміну від стандартних бізнес-додатків, описати дії користувача у вигляді одного або кількох ланцюжків дій в ігрових системах не так просто. Якщо виключити стандартні дії користувача на зразок авторизації або купівлі валюти, дії гравця у грі можна описати терміном, який геймдизайнери називають «ігровий цикл». В залежності від деталізації, ігровий цикл може представляти як узагальнені дії користувача чи механіки гри, так і докладні дії користувача під час гри. Але, в будь-якому випадку, будь-яка дія користувача приводить його на «точку старту», так що можна вважати, що в будь-який

момент часу може відбуватися будь-яка дія, або навіть кілька дій одночасно з боку будь-якого користувача, що потрібно враховувати під час проведення експериментів.

У процесі моделювання предметної області було виділено всі необхідні дані абстрактної моделі гри, але, важливо відзначити, що не всі ці дані мають постійне читання або запис з боку клієнтів. Одна з специфік розробки бази даних під ігрову серверну систему полягає у дуже нерівномірному навантаженні (читання/запис) на різні таблиці або колекції:

- стан гравця має часте оновлення та читання, наприклад, щоб гравець опинився на тому ж місці при наступному вході в гру;
- різні списки персонажів і монстрів мають низьку частоту запису і можуть завантажуватися на клієнт при запуску гри, але якщо їхня кількість велика, це може займати деякий час, що потребує інших підходів;
- інвентар та інформація про персонажа має «середню» за частотою читання та рідкісні запити на запис, але додавання нових предметів (наприклад, при підбиранні нагороди з сильного противника) може викликати масивну операцію на запис та оновлення.

Маючи цю інформацію, складемо набір запитів, на яких проводитимемо дослідження:

- читання інформації персонажа, його стан та інвентар;
- оновлення стану персонажа у світі (XP, координати, HP, MP, рівень);
- видалення предметів із інвентарю;
- поява (додавання) лута у світі під час перемоги над противниками;
- підбір нагороди (предметів) в інвентар персонажа (запис та видалення).

Як видно зі списку, передбачені запити на читання, запис, видалення та оновлення даних із БД. Також частина запитів має на увазі кілька операцій різного плану (наприклад, запис та видалення), що виконуються в одній

транзакції. Таким чином, з цим набором запитів можна повно протестувати ефективність досліджуваних логічних моделей.

2.5 Проектування програмного забезпечення для експерименту

Для проведення експерименту потрібно розробити програмне забезпечення, яке б виконувало всі необхідні операції та заміри метрік. Для його реалізації було обрано мову програмування C# і платформу .NET 6, що робить ПЗ підтримуваним на всіх розповсюджених платформах.

У якості архітектури ПЗ була обрана стандартна для практично всіх програм на платформі .NET – багатошарова (чиста) архітектура (див. рис. 2.10).

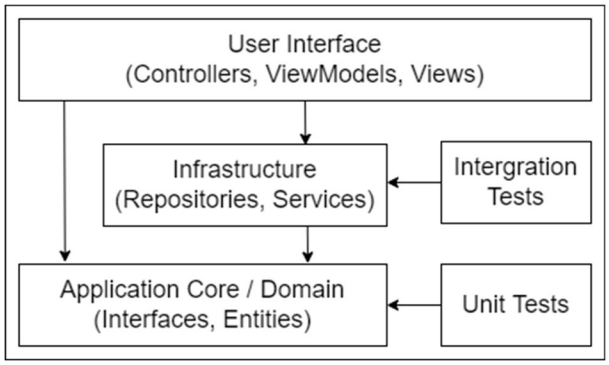

Рисунок 2.10 – Чиста (Onion) архітектура програмного забезпечення

Таким чином, додаток буде мати:

- ядро (сутності, агрегати, інтерфейси, специфікації тощо);
- інфраструктуру (репозиторії, послуги, служби інфраструктури, наприклад, логування);
- рівень подання (контролери, уявлення та їх моделі, фільтри, ПЗ для проміжного шару);
- інтеграційні та unit тести.

Дана архітектура добре підходить для дослідження, так як програмне забезпечення для експериментів можна декомпозувати на окремі частини. Таким чином, кілька екземплярів шарів представлення матимуть спільні шари ядра та інфраструктури.

Сам процес виконання дослідження можна представити у вигляді діаграми діяльності (див. рис. 2.11).

Рисунок 2.11 – Діаграма діяльності процесу дослідження

Очікується, що після виконання всіх запитів формується звіт з результатами виконання у форматі JSON. Даний звіт далі можна обробити

вручну або за допомогою додаткової утиліти, яка перетворить їх на діаграми (переважно, стовпчикові).

Таким чином, можна виділити такі частини функціоналу, що мають підтримуватися програмним забезпеченням:

- розгортання віртуальних машин у хмарному сервісі Azure та їхнє налаштування;
- наповнення розгорнутих БД тестовими даними;
- виконання набору запитів та замірювання необхідних метрик;
- візуалізація отриманих даних у вигляді графіків.

Дані частини можуть бути реалізовані як у вигляді окремих утиліт, так і у вигляді одного комплексного додатка.

Створення віртуальних машин в Azure повністю автоматизовано за рахунок бібліотеки Azure.Fluent, що дозволяє керувати ресурсами в хмарі дистанційно з коду.

Для генерації тестових даних використовується бібліотека Bogus, що може генерувати досить реалістичні дані, які співпадають з семантикою поля сутності: імена, email і реальні адреси і так далі.

Для виконання запитів були розроблені класи-репозиторії, що базуються на клієнтських бібліотеках для роботи з досліджуваними СКБД:

- офіційний дравер MongoDB для .NET (MongoDB .NET Driver);
- неофіційний драйвер (open-source) Neo4j для .NET (Neo4jClient).

Даний неофіційний драйвер Neo4j повністю побудований на офіційному, але значно розширює його функціонал: більш проста робота з транзакціями, написання запитів у .NET стилі (LINQ), автоматичний мапінг об'єктів.

3 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

3.1 Розробка фізичних моделей під СКБД MongoDB

Як було зазначено раніше, етап логічного моделювання не є останнім етапом проектування БД і використовувати цю модель у контексті СКБД не можна. Тепер необхідно провести останній етап – фізичного проектування під конкретні СКБД. Результатом цього етапу є фізична модель бази даних, яка містить:

- всі об’єкти БД, їхні атрибути з вказаними типами даних;
- індекси та всі ключі – первинні та зовнішні (якщо є);
- обмеження цілісності (якщо є).

Для дослідження будуть використовуватися три моделі під дві СКБД – MongoDB та Neo4j. Насамперед, спроектуємо фізичні моделі для СКБД MongoDB.

У контексті реляційних СКБД, фізичною моделлю зазвичай є скрипт створення всіх таблиць, ключів, індексів тощо. Але, однією з особливостей багатьох NoSQL СКБД є відсутність точної моделі даних, що породжує відсутність стандартного підходу до створення фізичної моделі.

Але, в СКБД MongoDB вже довгий час підтримується і допрацьовується певний функціонал для підтримки «точності» схеми – використання описової мови для визначення схеми. У якості описової мови використовується формат JSONSchema (draft 4).

Цей формат заснований на форматі JSON і спочатку створювався для створення анотацій для моделей та валідації JSON-документів. MongoDB використовує модифікацію цього формату з підтримкою своїх типів даних.

Отже, схему у цьому форматі можна використовувати як фізичну модель БД, бо при її використанні СКБД заборонятиме клієнту відхилятися від схеми – обов’язкова відповідність всіх атрибутів та типів даних.

Слід зазначити, що схема прописується кожної окремої колекції, отже фізичної моделлю можна вважати масив даних схем. Повні фізичні моделі (як нормалізованої, так і денормалізованої) містять кілька сотень рядків даних у форматі JSON, тому вони будуть наведені частково. Деякі моделі мають однаковий вигляд в обох моделях: Attribute та Ability.

```
{
 "$jsonSchema": {
   "title": "Attribute",
   "required": ["_id", "name", "description"],
   "properties": {
     "_id": { "bsonType": "objectId" },
     "name": { "bsonType": "string" },
     "description": { "bsonType": "string" }
   }
 }
}
{
  "$jsonSchema": {
    "title": "Ability",
    "required": [
      "_id",
      "name",
      "description",
      "minimalLevel",
      "abilityType",
      "damageType",
      "damageValue"
    ],
    "properties": {
      "_id": { "bsonType": "objectId" },
      "name": { "bsonType": "string" },
      "description": { "bsonType": "string" },
      "minimalLevel":
{"bsonType":"int","minimum":1,"maximum":100 },
      "abilityType": { "bsonType": "int" },
      "damageType": { "bsonType": "int" },
      "damageValue": { "bsonType": "int" }
    }
  }
}
```

Відмінність між денормалізованою та нормалізованою моделлю практично повністю полягає у реалізації «зв'язків» між документами. Така сутність як CharacterState має однакову структуру в обох моделях, тільки в нормалізованій версії вона є окремим документом, а в денормалізованій – вкладеною в Character. Нижче наведена версія для нормалізованої схеми.

```
{
  "$jsonSchema": {
```

```
        "title": "CharacterState",
      "required":["_id","characterId","coordinates","isAlive","HP","MP"],
        "properties": {
          "_id": { "bsonType": "objectId" },
          "characterId": { "bsonType": "objectId" },
          "coordinates": {
            "bsonType": "object","properties": {
              "type": { "bsonType": "string" },
              "coordinates": {
                "bsonType":"array","minItems":{"$numberInt":"3"},
                "maxItems":{"$numberInt":"3"},"items":{"type":"double"}
              }
            }
          },
          "isAlive": { "bsonType": "bool" },
          "HP": { "bsonType": "int", "minimum": 0 },
          "MP": { "bsonType": "int", "minimum": 0 }
        }
      }
      }
```

Через процес денормалізації, в БД з'явилося значне дублювання даних неключових атрибутів, що збільшує розмір об'єкта, але прибирає потребу в JOIN-подібній операції. Знизу представлені схеми сутності Item у нормалізованому та денормалізованому вигляді. Для стислості в денормалізованій моделі були прибрані атрибути, що присутні в нормалізованій моделі.

```
      {"$jsonSchema": {
          "title": "Item",
          "required":["_id","name","type","requiredLevel",  "description",
"attributes"],
          "properties": {
           "_id": { "bsonType": "objectId" },
           "name": { "bsonType": "string" },
           "type": { "bsonType": "string" },
           "requiredLevel": {"bsonType":"int","minimum":1,"maximum":100},
           "description": { "bsonType": "string" },
           "attributes": {
              "bsonType": "array",
              "items": {
                "bsonType": "object",
                "required": ["attributeId", "value"],
                "properties": {
                  "attributeId": { "bsonType": "objectId" },
                  "value": { "bsonType": "int", "minimum": 1 }
                }
              }
            }
          }
        }}
      {"$jsonSchema": {
          "title": "Item",
```

```
    "properties": {...
      "attributes": {
        "bsonType": "array",
        "items": {
          "bsonType": "object",
          "required": ["_id", "name", "description", "value"],
          "properties": {
            "_id": { "bsonType": "objectId" },
            "name": { "bsonType": "string" },
            "description": { "bsonType": "string" },
            "value": { "bsonType": "int", "minimum": 1 }
          }
        }
      }
    }
  }}
```

Подібним чином, були реалізовані JSON-схеми для всіх сутностей обох моделей – нормалізованої та денормалізованої.

3.2 Розробка фізичних моделей під СКБД Neo4j

Далі необхідно розробити фізичну модель для СКБД Neo4j на основі розробленої раніше логічної моделі. Ця СКБД так само не має суворої типізації об'єктів БД, як і стандартного способу описати фізичну модель.

Але, присутність індексу або обмеження на сутності повідомляє СУБД, що існуватимуть екземпляри сутності з цим ім'ям і, як мінімум, даним полем. Також, якщо в СУБД існує хоча б один екземпляр сутності, то СУБД (а також її клієнти) «знатиме» про існування цієї сутності з даними атрибутами. Всі ці особливості також стосуються і зв’язків.

Таким чином, фізичну модель для СУБД Neo4j можна представити у вигляді двох наборів скриптів:

- один створює тестову базу даних, яка містить усі види сутностей (див. рис. 3.1).
- другий створює всі потрібні індекси та обмеження на унікальність (див. рис. 3.2),

Слід зазначити, що єдиний тип обмеження, який є доступним у Community версії СУБД – це обмеження на унікальність значення поля.

```
CREATE
(user:User { Id: '1', Email: 'test@gmail.com', Password: 'base64string',
             Nickname: 'Nickname', RegistrationDate: datetime('2020-01-01T00:00:00.000Z') }),
(loginHistory:LoginHistory { Id: '2', LoginDateTime: datetime('2020-01-02T00:00:00.000Z'),
                             LogoutDateTime: datetime('2020-01-03T00:00:00.000Z'),
                             IpAddress: '192.168.0.1', DeviceName: 'DeviceName' }),
(character:Character { Id: '3', Name: 'Name', Level: 1, XP: 0, Money: 0 }),
(characterState:CharacterState { Id: '4', Coordinates: point({x:0,y:0,z:0}),
                                 IsAlive: true, HP: 100, MP: 100 }),
(ability:Ability { Id: '5', Name: 'Name', Description: 'Description',
                   MinimalLevel: 1, AbilityType: 0, DamageType: 0, DamageValue: 20 }),
(npc:NPC { Id: '6', Name: 'Name', Coordinates: point({x:0,y:0,z:0}) }),
(quest:Quest { Id: '7', Name: 'Name', Description: 'Description', Reward: 100}),
(questStage:QuestStage { Id: '8', Name: 'Name', Objective: 'Objective',
                         Description: 'Description' }),
(item:Item { Id: '9', Name: 'Name', Type: 'Weapon', RequiredLevel: 1,
                      Description: 'Description' }),
(loot:Loot { Id: '10', XP: 100, Money: 100, DropDate: datetime('2020-01-02T10:00:00.000Z') }),
(attribute:Attribute { Id: '11', Name: 'Name', Description: 'Description' }),
(enemy:Enemy { Id: '12', Level: 1, Reward: 100, HP: 100, Coordinates: point({x:2,y:2,z:1}),
               SpawnRadius: 1.5, SpawnLimit: 10 }),
(loginHistory)<-[:LOGGED_IN]-(user)-[:HAS]->(character),
(characterState)<-[:HAS]-(character)-[:HAS]->(ability),
(quest)<-[:ACCOMPLISHES { StartedDate: datetime('2020-01-02T10:00:00.000Z'),
                          CurrentStage: 'Stage1' }]-(character),
(item)<-[:HAS_EQUIPPED {Slot: 'Weapon'}]-(character)-[:HAS_IN_INVENTORY {Count: 1}]->(item),
(npc)-[:ISSUES]->(quest)-[:HAS]->(questStage),
(character)-[:HAS_ATTRIBUTE {Value: 1}]->(attribute)<-[:HAS_ATTRIBUTE {Value: 5}]-(enemy),
(enemy)-[:CONTAINS_REWARD {Count: 1}]->(item)-[:HAS_ATTRIBUTE {Value: 3}]->(attribute),
(quest)-[:CONTAINS_REWARD {Count: 1}]->(item)<-[:CONTAINS_REWARD {Count: 1}]
                                              -(loot)<-[:OWNS]-(character);
```

Рисунок 3.1 – Скрипт створення тестової БД

Такі обмеження як «існування поля в сутності чи зв'язку» доступні лише в Enterprise версії БД, тому в цій моделі не використовуються.

```
CREATE CONSTRAINT User_Id IF NOT EXISTS FOR (n:User) REQUIRE (n.Id) IS UNIQUE;
CREATE CONSTRAINT LoginHistory_Id IF NOT EXISTS FOR (n:LoginHistory) REQUIRE (n.Id) IS UNIQUE;
CREATE CONSTRAINT Character_Id IF NOT EXISTS FOR (n:Character) REQUIRE (n.Id) IS UNIQUE;
CREATE CONSTRAINT CharacterState_Id IF NOT EXISTS FOR (n:CharacterState) REQUIRE (n.Id) IS UNIQUE;
CREATE CONSTRAINT Ability_Id IF NOT EXISTS FOR (n:Ability) REQUIRE (n.Id) IS UNIQUE;
CREATE CONSTRAINT NPC_Id IF NOT EXISTS FOR (n:NPC) REQUIRE (n.Id) IS UNIQUE;
CREATE CONSTRAINT Quest_Id IF NOT EXISTS FOR (n:Quest) REQUIRE (n.Id) IS UNIQUE;
CREATE CONSTRAINT QuestStage_Id IF NOT EXISTS FOR (n:QuestStage) REQUIRE (n.Id) IS UNIQUE;
CREATE CONSTRAINT Item_Id IF NOT EXISTS FOR (n:Item) REQUIRE (n.Id) IS UNIQUE;
CREATE CONSTRAINT Loot_Id IF NOT EXISTS FOR (n:Loot) REQUIRE (n.Id) IS UNIQUE;
CREATE CONSTRAINT Attribute_Id IF NOT EXISTS FOR (n:Attribute) REQUIRE (n.Id) IS UNIQUE;
CREATE CONSTRAINT Enemy_Id IF NOT EXISTS FOR (n:Enemy) REQUIRE (n.Id) IS UNIQUE;

CREATE INDEX User_Email_Password IF NOT EXISTS FOR (n:User) ON (n.Email, n.Password);
CREATE CONSTRAINT User_Email IF NOT EXISTS FOR (n:User) REQUIRE (n.Email) IS UNIQUE;
CREATE CONSTRAINT User_Nickname IF NOT EXISTS FOR (n:User) REQUIRE (n.Nickname) IS UNIQUE;
CREATE CONSTRAINT Character_Name IF NOT EXISTS FOR (n:Character) REQUIRE (n.Name) IS UNIQUE;
```

Рисунок 3.2 – Скрипт створення обмежень та індексів

В результаті проектування були розроблені три фізичні моделі БД під СКБД MongoDB та Neo4j: дві схеми для валідації об’єктів та скрипт на

створення БД. Ці моделі далі будуть використовуватися для проведення експериментального дослідження.

3.3 Реалізація запитів для експериментального дослідження

3.3.1 Реалізація запитів під СКБД MongoDB

Відтепер, коли були розроблені всі фізичні моделі БД для експериментів, необхідно розробити набір запитів для виконання самого процесу дослідження. Спочатку, створимо запити для СКБД MongoDB.

Для роботи з СУБД та виконання запитів, MongoDB використовує мову MQL (MongoDB Query Language), яка має велику схожість із мовою програмування JavaScript.

Перед тим, як розпочати розробку запитів, слід нагадати, що запити розробляються для двох документних моделей: нормалізованої та денормалізованої. Тому кожний запит буде наводитися у двох виглядах для кожної з моделей.

Почнемо з досить базового запита – отримання повної інформації про деякого персонажа. Спочатку розробимо цей запит для нормалізованої моделі:

```
db.getCollection("Character").aggregate([
  { $match: { "_id": {$eq:ObjectId("62700367922baedc1c3bed05")} }
},
  { $lookup:{from:"CharacterState", localField:"_id",
foreignField:"characterId", as:"characterState" } },
  { $set: { characterState: { $arrayElemAt: ["$characterState", 0]
} } },
  { $lookup: { from: "Quest", localField: "activeQuests.questId",
foreignField: "_id", as: "characterActiveQuests" } },
  { $lookup: { from: "Item", localField: "equipment.itemId",
foreignField: "_id", as: "equippedItems" } },
  { $lookup: { from: "Item", localField: "inventory.itemId",
foreignField: "_id", as: "inventoryItems" } },
  { $lookup: { from: "Attribute", localField:
"attributes.attributeId", foreignField:"_id", as:"characterAttributes" }
}
], { allowDiskUse: true })
```

А тепер, цей самий запит для денормалізованої моделі:

```
db.getCollection("Character").find({_id:ObjectId("Id")})
```

Як видно з поданих запитів, денормалізована версія значно коротша і використовує лише одну операцію за рахунок великого дублювання даних. У нормалізованій версії, всі пов'язані дані ще потрібно зібрати докупи.

Але важливо відзначити, що, хоча, офіційний драйвер MongoDB для .NET і може виконувати запити в такому вигляді, стандартний підхід до написання запитів передбачає використання LINQ-подібного синтаксису. Таким чином, ці запити можна записати на мові C# як:

```
collection.Aggregate(new AggregateOptions { AllowDiskUse = true })
  .Match(x => x.Id == id)
  .Lookup<CharacterState, MongoNormalizedCharacterFull>
("CharacterState", "_id", "characterId", "characterState")
  .AppendStage<MongoNormalizedCharacterFull>
(BsonDocument.Parse("{ $set: { characterState: { $arrayElemAt:
[\"$characterState\", 0] } } }"))
  .Lookup<MongoNormalizedQuest, MongoNormalizedCharacterFull>
("Quest", "activeQuests.questId", "_id", "characterActiveQuests")
  .Lookup<MongoNormalizedItem, MongoNormalizedCharacterFull>
("Item", "equipment.itemId", "_id", "equippedItems")
  .Lookup<MongoNormalizedItem, MongoNormalizedCharacterFull>
("Item", "inventory.itemId", "_id", "inventoryItems")
  .Lookup<Attribute, MongoNormalizedCharacterFull>
("Attribute", "attributes.attributeId", "_id",
"characterAttributes")
  .FirstOrDefault();

collection.FindSync(
  Builders<MongoDenormalizedCharacter>.Filter.Eq(x => x.Id, id)
).FirstOrDefault();
```

За представленим кодом видно, що LINQ-подібний синтаксис зберіг усі ключові слова MongoDB, назви операцій тощо. Тому, далі запити будуть наведені мовою MQL.

Далі, розробимо запит для отримання нагороди (лута) персонажем. Цей запит передбачає додавання даних до одної колекції та видалення з іншої колекцій за один раз. Раніше, MongoDB був дуже обмежений інструментарій для роботи з транзакціями, але у версії 4.0 інструментарій був значно розширений [21]. Тому, розробимо цей запит для MongoDB, використуючи транзакції:

```
var session = db.getMongo().startSession()
```

```
session.startTransaction({
    "readConcern": { "level": "snapshot" },
    "writeConcern": { "w": "majority" }
})

db.getCollection("Character").updateOne(
{
    _id: { $eq: ObjectId("6270427f642c33e9415ec10a") }
},
{
    $inc: { XP: NumberInt(100), money: NumberInt(100) },
    $push: { "inventory": { $each: [ { itemId:
ObjectId("6270427f642c33e9415ebead"), count: 10 } ] } }
})

db.getCollection("Loot").deleteOne({ _id:
ObjectId("6270427f642c33e9415ed87a") })

session.commitTransaction()
```

Слід зазначити, що даний запит є майже однаковим, як для нормалізованої БД, так і для денормалізованої. Єдина різниця – в денормалізованій моделі видалення буде виконуватися з вкладеного масиву. Комбінація параметрів «readConcern: snapshot» та «writeConcern: majority» означає, що при паралельному читанні повертатимуться дані з того snapshot, що зафіксований на більшості реплік.

Тепер спроектуємо запит на додавання даних – створення об'єкта з нагородою при перемозі над супротивником. Цей запит дуже відрізняється в залежності від схеми, бо додавання відбувається у різні колекції. Запит для нормалізованої моделі:

```
db.getCollection("Loot").insertOne({
    "XP" : NumberInt(1011011124),
    "money" : NumberInt(1850518016),
    "dropDate" : ISODate("2022-04-09T08:36:55.692+0000"),
    "characterId" : ObjectId("6270427f642c33e9415ec415"),
    "itemRewards" : [
        {
            "itemId" : ObjectId("6270427f642c33e9415ebecd"),
            "count" : NumberInt(44)
        }
    ]
})
```

І той самий запит для денормалізованої моделі:

```
db.getCollection("Character").updateOne({ _id: { $eq:
ObjectId("6270427f642c33e9415ec10a") }},
{
    $push: { "droppedLoot": {
```

```
            "XP" : NumberInt(1011011124),
            "money" : NumberInt(1850518016),
            "dropDate" : ISODate("2022-04-09T08:36:55.692+0000"),
            "characterId" : ObjectId("6270427f642c33e9415ec415"),
            "itemRewards" : [
            {
                "itemId" : ObjectId("6270427f642c33e9415ebecd"),
                "count" : NumberInt(44)
            }
        ]
       } }
})
```

Також, в одному випадку це є операцією редагування, а в іншому – додавання.

Аналогічним чином, використовуючи мову MQL, були розроблені і інші запити для нормалізованої та денормалізованої схем.

3.3.2 Реалізація запитів під СКБД Neo4j

Наступним етапом є розробка запитів для СКБД Neo4j. Для виконання запитів до цієї СКБД рекомендується використовувати мову їхньої власної розробки – Cypher. Мова Gremlin, яка є стандартною для всіх графових СКБД, також підтримується, але, на відміну від неї, Cypher має значно чистіший та інтуітивно зрозумілий синтаксис, що робить запити теж схожими на графи.

Почнемо з того ж запиту, що і для документних БД – отримання повної інформації про персонажа:

```
     MATCH (state:CharacterState)<-[:HAS]-(character:Character {Id:
"3"})
     OPTIONAL MATCH (character)-[:ACCOMPLISHES]->(quest:Quest)
     OPTIONAL MATCH (character)-[:HAS_IN_INVENTORY]->(invItem:Item)
     OPTIONAL MATCH (character)-[:HAS_EQUIPPED]->(equippedItem:Item)
     OPTIONAL MATCH (character)-[:HAS_ATTRIBUTE]->
(characterAttribute:Attribute)
     RETURN character, state, collect(quest) AS quests, collect(invItem)
AS invItems, collect(equippedItem) AS equippedItems,
collect(characterAttribute) AS characterAttributes
```

Якщо проводити аналогію з кардинальністю зв'язків між сутностями: MATCH є еквівалентом зв'язку 1: M, а OPTIONAL MATCH – 1:0/M. В

залежності від типа зв’язку між сутностями, змінюється вигляд конструкції в RETURN:

- якщо зв’язок є 1:1, то просто вказується ім’я змінної;
- якщо зв’язок є 1:M і треба повернути всі сутності – викликається функція collect() для збору всіх результатів у колекцію (масив).

Але, цей запит у такому вигляді не може бути використаний навіть офіційним драйвером Neo4j для .NET, що робить його непридатним для експериментів. Тепер потрібно перетворити даний запит на код на C# з використанням драйвера Neo4jClient:

```
      var results = repository.Client.Cypher
        .Match("MATCH (state:CharacterState)<-[:HAS]-
(character:Character)")
        .Where("character.Id = $id")
        .WithParam("id", id)
        .OptionalMatch("OPTIONAL MATCH (character)-[:ACCOMPLISHES]-
>(quest:Quest)")
        .OptionalMatch("OPTIONAL MATCH (character)-[:HAS_IN_INVENTORY]-
>(invItem:Item)")
        .OptionalMatch("OPTIONAL MATCH (character)-[:HAS_EQUIPPED]-
>(equippedItem:Item)")
        .OptionalMatch("OPTIONAL MATCH (character)-[:HAS_ATTRIBUTE]-
>(characterAttribute:Attribute)")
        .Return((state, character, quest, invItem, equippedItem,
characterAttribute) => new
        {
          State = state.As<Neo4jCharacterState>(),
          Character = state.As<Character>(),
          Quests = state.CollectAs<Quest>(),
          InventoryItems = state.CollectAs<Item>(),
          EquippedItems = state.CollectAs<Item>(),
          CharacterAttributes =
state.CollectAs<Models.Entities.BaseEntities.Attribute>()
        })
      .ResultsAsync.GetAwaiter().GetResult();
```

Як видно з фрагмента коду, однією з проблем роботи з Neo4j є мапінг об'єктів. Навіть із додатковим функціоналом мапінгу від драйвера Neo4jClient, доводиться використовувати проміжний об'єкт для отримання даних із запиту, а потім, з'єднувати їх вручну.

Далі, розробимо запит оновлення інформації про персонажа в світі:

```
      MATCH (character:Character {Id: "3"})-[:HAS]-
>(state:CharacterState)
      SET state +=
```

```
      { HP:78, MP:100, IsAlive:true, Coordinates:point({x: 1, y: 1, z:1})
}
      SET character += { XP: 100, Level: 1  }
      RETURN character, state

      var query = repository.Client.Cypher
        .Match("MATCH (character:Character)-[:HAS]-
>(state:CharacterState)")
        .Where("character.Id = $id")
        .WithParam("id", character.Id)
        .Set("state += { HP: $hp, MP: $mp, IsAlive: $isAlive,
Coordinates: point({ x: 1, y: 1, z: 1}) }")
        .WithParams(new { hp = state.HP, mp = state.MP, isAlive =
state.IsAlive })
        .Set("character += { XP: $xp, Level: $level  }")
        .WithParams(new { xp = character.XP, level = character.Level })
        .Return(character => character.As<Character>())
        .ResultsAsync
        .GetAwaiter().GetResult();
```

Для скороченого запису в цьому запиті використовується оператор мутації "+=". Його поведінка полягає в тому, що він замінює значення полів, якщо вони існують або додає їх, якщо вони відсутні.

Таким чином, були розроблені всі потрібні запити для виконання експериментальних досліджень.

4 ОПИС ЕКСПЕРИМЕНТАЛЬНИХ ДОСЛІДЖЕНЬ

4.1 Дослідження для MongoDB

Тепер, за допомогою розробленого програмного забезпечення, можна провести серію експериментів для порівняння запропонованих методів. Після проведення даних експериментів було отримано необхідні метрики, які далі будуть представлені.

Перед описом результатів, для стислості та наочності, введемо умовні позначення (ідентифікатори) для всіх досліджуваних запитів:

- GetCharacter – читання інформації персонажа та його стан;
- UpdateCharacter – оновлення стану персонажа;
- RemoveFromInventory – видалення предметів із інвентарю;
- CreateLootInstance – додавання лута (нагороди);
- PickUpLoot – підбір нагороди в інвентар персонажа.

В першу чергу, будуть розглянуті метрики моделей для СКБД MongoDB. Після заповнення баз даних тестовими даними були отримані розміри баз даних досліджуваних моделей (див. табл. 5).

Таблиця 5 – Порівняння розміру бази даних моделей під СКБД MongoDB

	Розмір даних денормалізованої моделі (МБ)	Реальний розмір БД денормалізованої моделі (МБ)	Розмір даних нормалізованої моделі (МБ)	Реальний розмір БД нормалізованої моделі (МБ)
Small	29.7	13.5	21	8.8
Medium	116.2	48.8	90	33.5
Large	356.7	137.7	266.1	91.0

Слід враховувати, що MongoDB використовує внутрішні механізми стискання даних, що призводить до значного меншого розміру файлу БД.

Далі наведемо результати виконання запитів. Почнемо із запиту GetCharacterFull для денормалізованої моделі (див. табл. 6).

Таблиця 6 – Результати виконання запиту GetCharacterFull з використанням денормалізованої схеми

	Мінімальний час (с)	Середній час (с)	Максимальний час (с)	RAM / CPU (ГБ / %)
Small	0.039	0.190	1.330	1.9 / 14
Medium	0.040	0.049	0.534	4.4 / 28
Large	0.046	0.073	0.724	8.8 / 32

І, також, результати аналогічних замірів для нормалізованої моделі (див. табл. 7).

Таблиця 7 – Результати виконання запиту GetCharacterFull з використанням нормалізованої схеми

	Мінімальний час (с)	Середній час (с)	Максимальний час (с)	RAM / CPU (ГБ / %)
Small	0.047	0.187	1.035	1.8 / 21
Medium	0.051	0.122	1.195	4.12 / 40
Large	0.089	0.245	1.421	8.1 / 46

Видно, що нормалізована модель використовує більше ресурсів та програє за продуктивністю через використання JOIN-подібних операцій, коли денормалізована модель використовує звичайний пошук за ключем.

Наступним за списком є запит на оновлення даних персонажа (UpdateCharacter). Нижче наведено виміри для денормалізованої моделі (див. табл. 8).

Таблиця 8 – Результати виконання запиту UpdateCharacter з використанням денормалізованої схеми

	Мінімальний час (с)	Середній час (с)	Максимальний час (с)	RAM / CPU (ГБ / %)
Small	0.040	0.383	1.544	2.0 / 12
Medium	0.040	0.048	0.530	4.4 / 19
Large	0.062	0.351	1.192	8.8 / 22

Далі, наводимо результати вимірів метрик для нормалізованої моделі (див. табл. 9).

Таблиця 9 – Результати виконання запиту UpdateCharacter з використанням нормалізованої схеми

	Мінімальний час (с)	Середній час (с)	Максимальний час (с)	RAM / CPU (ГБ / %)
Small	0.094	0.772	2.361	1.9 / 33
Medium	0.085	0.107	0.859	4.2 / 50
Large	0.261	1.251	2.513	8.2 / 55

На цей раз денормалізована схема теж виграє у нормалізованої: перша використовує операції над однією колекцією, а друга - над декількома з використанням транзакцій.

Наступним йде запит видалення предметів з інвентарю персонажа (RemoveFromInventory). Як і минулі рази, спочатку наводимо результати для денормалізованої моделі (див. табл. 10).

Таблиця 10 – Результати виконання запиту RemoveFromInventory з використанням денормалізованої схеми

	Мінімальний час (с)	Середній час (с)	Максимальний час (с)	RAM / CPU (ГБ / %)
Small	0.046	0.386	1.704	2.0 / 19
Medium	0.040	0.047	0.770	4.4 / 24
Large	0.052	0.276	1.252	8.8 / 28

Слід за ним приводимо метріки запиту для нормалізованої моделі (див. табл. 11).

Таблиця 11 – Результати виконання запиту RemoveFromInventory з використанням нормалізованої схеми

	Мінімальний час (с)	Середній час (с)	Максимальний час (с)	RAM / CPU (ГБ / %)
Small	0.057	0.404	1.538	1.9 / 20
Medium	0.041	0.057	1.033	4.2 / 26
Large	0.062	0.304	1.492	8.2 / 27

Як видно з результатів, продуктивність обох моделей приблизно однакова з огляду на те, що їхні запити теж практично однакові. Різниця лежить у межах похибки.

Наступним по списку йде запит створення об'єкта нагороди (CreateLootInstance). Враховуючи, що логіку вибору нагороди визначає серверна частина, це чисте порівняння продуктивності операцій додавання. Нижче наведено метрики для денормалізованої моделі (див. табл. 12).

Таблиця 12 – Результати виконання запиту CreateLootInstance з використанням денормалізованої схеми

	Мінімальний час (с)	Середній час (с)	Максимальний час (с)	RAM / CPU (ГБ / %)
Small	0.040	0.383	1.358	2.0 / 14
Medium	0.040	0.048	0.315	4.4 / 15
Large	0.042	0.234	0.924	8.8 / 17

Після метрик денормалізованої моделі, традиційно наведемо метрики запитів до денормалізованої (див. табл. 13).

Таблиця 13 – Результати виконання запиту CreateLootInstance з використанням нормалізованої схеми

	Мінімальний час (с)	Середній час (с)	Максимальний час (с)	RAM / CPU (ГБ / %)
Small	0.048	0.381	1.732	1.9 / 14
Medium	0.044	0.081	1.244	4.2 / 15
Large	0.052	0.241	1.101	8.2 / 16

Тут ситуація схожа на попередній запит, але, що примітно, операція додавання в масив виявилася в середньому швидше, ніж додавання об'єкта. Це можна виправдати тим, що об'єктів персонажа в БД значно менше, ніж об'єктів лута, що призводить до більш швидкого пошуку.

Останнім запитом є запит отримання нагороди персонажем (PickUpLoot). Важливо відзначити, що цей запит має на увазі транзакцію в кожній з моделей. Наведемо результати денормалізованої моделі (див. табл. 14).

Таблиця 14 – Результати виконання запиту PickUpLoot з використанням денормалізованої схеми

	Мінімальний час (с)	Середній час (с)	Максимальний час (с)	RAM / CPU (ГБ / %)
Small	0.040	0.193	1.045	2.0 / 24
Medium	0.040	0.049	0.325	4.4 / 42
Large	0.060	0.116	0.746	8.8 / 51

Також наводимо метрики для нормалізованої моделі (див. табл. 15).

Таблиця 15 – Результати виконання запиту PickUpLoot з використанням нормалізованої схеми

	Мінімальний час (с)	Середній час (с)	Максимальний час (с)	RAM / CPU (ГБ / %)
Small	0.040	0.848	2.734	2 / 27
Medium	0.040	0.260	0.659	4.4 / 50
Large	0.046	0.428	1.572	8.3 / 58

У цій ситуації денормалізована схема також є більш ефективною через те, що в ній виконуються операції над однією колекцією, а в нормалізованій – над декількома.

Слід зазначити, що значення RAM для обох моделей протягом серій запитів мало змінюються, бо СКБД на момент виконання запитів вже має у пам'яті кеш всіх необхідних даних (прогріта).

Таким чином, були проведені серії експериментів для СУБД MongoDB.

4.2 Дослідження для Neo4j

Тепер переходимо до експериментів над СУБД Neo4j. На відміну від MongoDB, стиснення даних, що зберігаються в БД, немає, так само як і способу дізнатися розмір БД за допомогою команд СУБД. Після заповнення БД тестовими даними, було отримано значення розміру бази даних, кількості сутностей та зв’язків (див. табл. 16).

Таблиця 16 – Порівняння розміру баз даних моделей під СКБД Neo4j

	Кількість сутностей	Кількість зв’язків	Розмір бази даних (МБ)
Small	29666	222950	309
Medium	68624	642200	515
Large	116842	1304200	1090

Відразу помітно, що отримані бази даних важать значно більше, ніж аналогічні тестові бази MongoDB. Слід зазначити, що Neo4j не надає зберігання процедур або команд для отримання статусу сервера, ваги бази даних і так далі.

Тому замір розміру БД здійснюється вручну через різницю розміру порожньої БД та заповненої.

Перейдемо до результатів виконання запитів. Отримані метрики запитів наводитимемо в тому ж порядку, що і при експериментах з MongoDB.

Почнемо із запиту отримання інформації про персонажа (див. табл. 17).

Таблиця 17 – Результати виконання запиту GetCharacterFull з використанням Neo4j

	Середній час (с)	Використання RAM (ГБ)	Використання CPU (%)
Small	0.186	2.8	70
Medium	0.210	5.6	52
Large	0.225	14.6	54

Відразу помітно, що ця СУБД використовує набагато більше ресурсів, ніж MongoDB, хоч і тримає планку в плані продуктивності.

Наступним є запит на оновлення даних про персонажа «UpdateCharacter» (див табл. 18).

Таблиця 18 – Результати виконання запиту UpdateCharacter з використанням Neo4j

	Середній час (с)	Використання RAM (ГБ)	Використання CPU (%)
Small	0.982	3.0	30
Medium	1.005	6	34
Large	1.064	15	29

Слід зазначити, що запити на оновлення даних проходять однаково ефективно (а, в середньому, навіть краще) з нормалізованою схемою MongoDB. При збільшенні кількості типів сутностей, що оновлюються, дана різниця була б більш помітна. Використання CPU все ще високе, хоч і менше, ніж було.

Наступним наведемо метрики запиту видалення предметів з інвентарю «RemoveFromInventory» (див. табл. 19).

Таблиця 19 – Результати виконання запиту RemoveFromInventory з використанням Neo4j

	Середній час (с)	Використання RAM (ГБ)	Використання CPU (%)
Small	0.721	3.0	40
Medium	0.586	6	42
Large	0.692	15	41

Час виконання запиту теж є прийнятним даної предметної області. Слід окремо відзначити, що операція видалення предмета у цій схемі передбачає видалення сутностей, що робить цей запит потенційно дуже ефективним, якщо предметів потрібно прибрати з інвентарю багато.

Далі наведемо метріки запиту на створення нагороди (див. табл. 20).

Таблиця 20 – Результати виконання запиту CreateLootInstance з використанням Neo4j

	Середній час (с)	Використання RAM (ГБ)	Використання CPU (%)
Small	0.428	3.2	35
Medium	0.386	6.1	38
Large	0.402	15	42

Створення нових сутностей теж не є сильною стороною СКБД Neo4j: використовується досить багато ресурсів і час виконання досить великий, хоч і лежить у межах норми для подібних систем.

Насамкінець, перейдемо до метрик останнього запиту – операція на підбір нагороди гравцем.

Таблиця 21 – Результати виконання запиту PickUpLoot з використанням Neo4j

	Середній час (с)	Використання RAM (ГБ)	Використання CPU (%)
Small	0.962	3.4	78
Medium	0.568	6.2	69
Large	0.684	15.2	72

Тут слід окремо відзначити, що в Community версії Neo4j транзакції не підтримуються, що необхідно для цього запиту. Проте, дана СКБД має підтримку виконання кількох операцій у межах одного запиту, що є прийнятним у даному випадку. СКБД справляється із цим запитом досить добре, хоч і використовує багато ресурсів.

Так були проведені всі необхідні експерименти над розробленими моделями для СКБД MongoDB та Neo4j.

5 АНАЛІЗ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

5.1 Порівняння продуктивності методів та СКБД

Тепер, після проведення експериментального дослідження, можна порівняти отримані результати (метрик). Найбільш наочно це можна здійснити за допомогою візуалізації, зокрема, за допомогою діаграм.

Насамперед, порівняємо розміри БД при заповнених тестових даних (рис. 5.1).

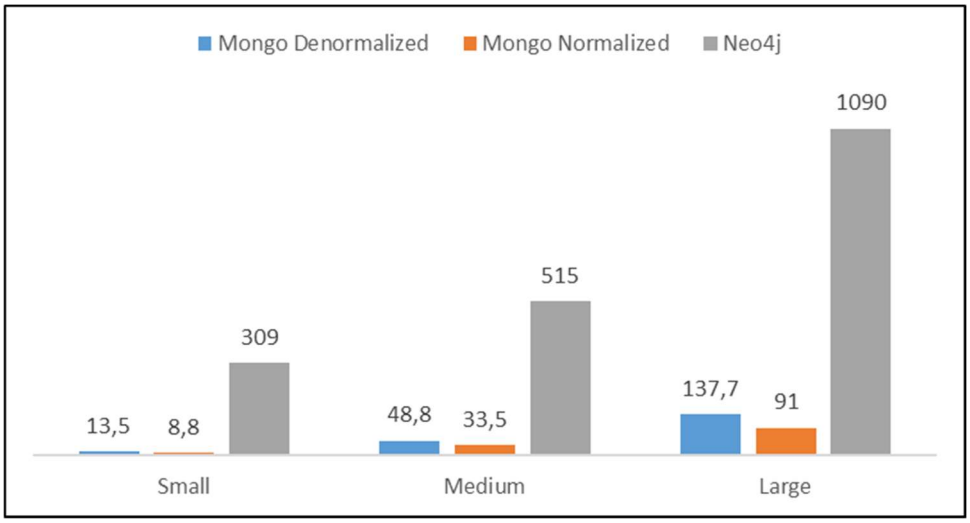

Рисунок 5.1 – Порівняння розмірів БД (на диску)

З діаграми чітко видно, що СУБД Neo4j споживає величезну кількість місця на диску і це зростання практично лінійне кількості сутностей. Слід зазначити, що в ході експериментів було чітко видно, що ця "вага БД" формують саме сутності, самі зв'язки практично не займають місце на диску.

Денормалізована модель MongoDB важить на 30-35% більше за нормалізовану, що, очевидно, викликане надмірністю даних. Проте, у плані ваги БД MongoDB однозначно виграє у Neo4j.

Тепер перейдемо до порівняння ефективності запитів. Першим йде запит отримання даних про персонажа (див. рис. 5.2). На мінімальній конфігурації

результати рівні через малу кількість сутностей: пошук відпрацьовує дуже швидко з малим додаванням витрат часу на пересилання даних.

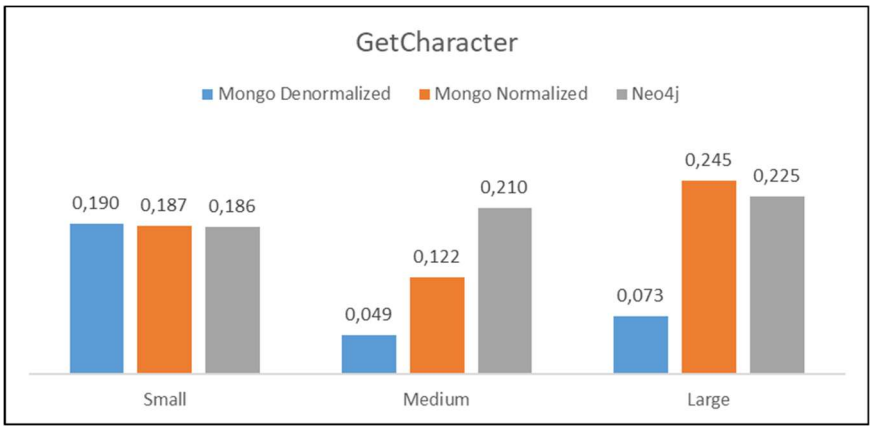

Рисунок 5.2 – Порівняння середнього часу виконання «GetCharacter»

На високих конфігураціях однозначно видно, що денормалізована MongoDB модель виграє у всіх і має найменше падіння продуктивності зі збільшенням кількості об'єктів БД. Але, це не можна сказати про нормалізовану модель, яка має найбільше падіння. Зі збільшенням числа даних, не дуже ефективний lookup починає сильніше програвати графовим зв'язкам.

Наступний запит – оновлення інформації про персонажа (див. рис. 5.3).

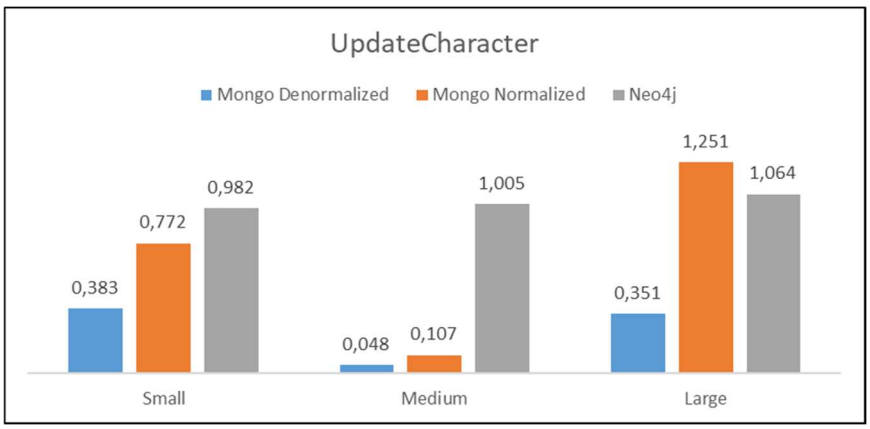

Рисунок 5.3 – Порівняння середнього часу виконання «UpdateCharacter»

Ситуація дуже схожа на попередній запит, але час виконання запитів менше.

Оновлення кількох сутностей у MongoDB відбувається повільніше, ніж в одній, що обумовлює різницю у часі виконання.

Наступним наведемо порівняння часу виконання запитів на видалення предметів з інвентарю (див. рис. 5.4).

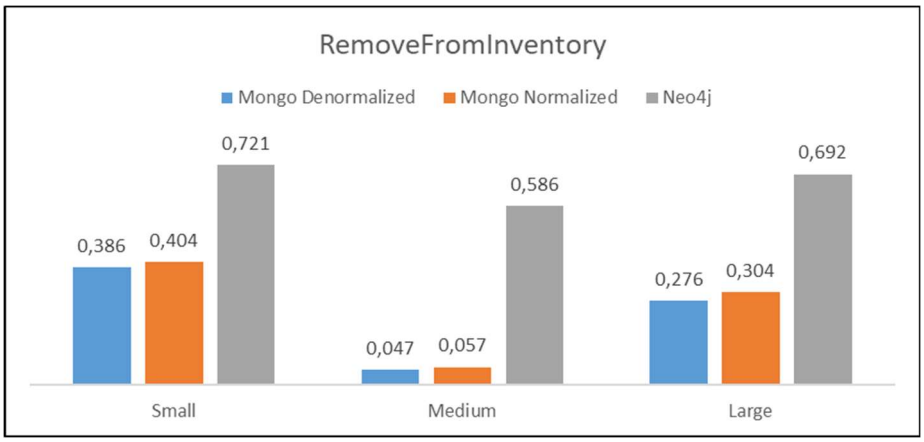

Рисунок 5.4 – Порівняння середнього часу виконання запиту RemoveFromInventory

Незважаючи на те, що СУБД Neo4j не виконує видалення сутностей, тільки зв'язків, продуктивність не така висока, як очікувалося. Ймовірно, тут вплинув час пошуку сутностей, між якими на які вказують зв'язки, що видаляються. Також, не варто забувати про час пересилання запиту до БД та результату з неї.

Високі результати виконання запитів MongoDB при конфігурації Medium обумовлені високим ставленням доступних ресурсів до «потреб» СКБД.

Наступний запит на порівняння – створення об'єктів нагороди для гравця (див. рис. 5.5).

Ситуація в даному запиті трохи схожа на запит GetCharacter – продуктивність на низькій та високій конфігураціях досить схожа. В цілому можна сказати, що Neo4j досить непогано справляється з поодинкими операціями вставки.

Але тут слід зазначити, що для створення зв'язків при вставці необхідно здійснити пошук сутностей, на які посилатимуться зв'язки, що може значно збільшити час виконання запиту.

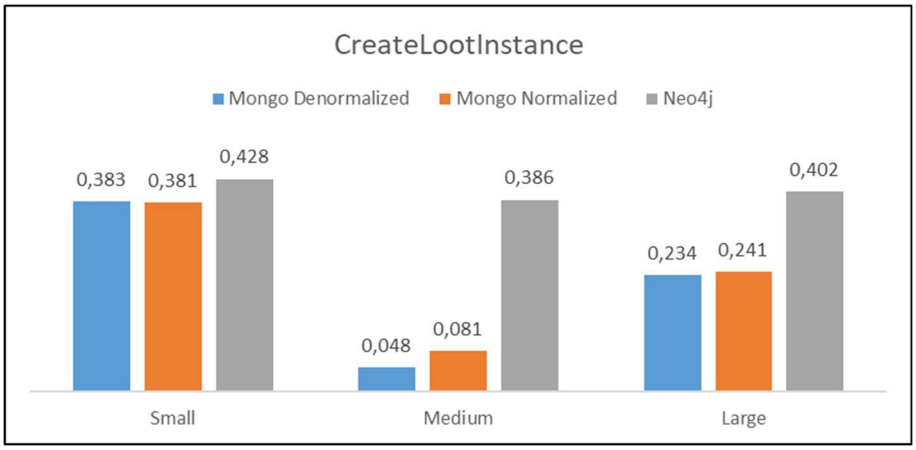

Рисунок 5.5 – Порівняння середнього часу виконання запиту CreateLootInstance

Тепер ми переходимо до метрік виконання останнього виду запитів – підбір нагороди гравцем (див. рис. 5.6).

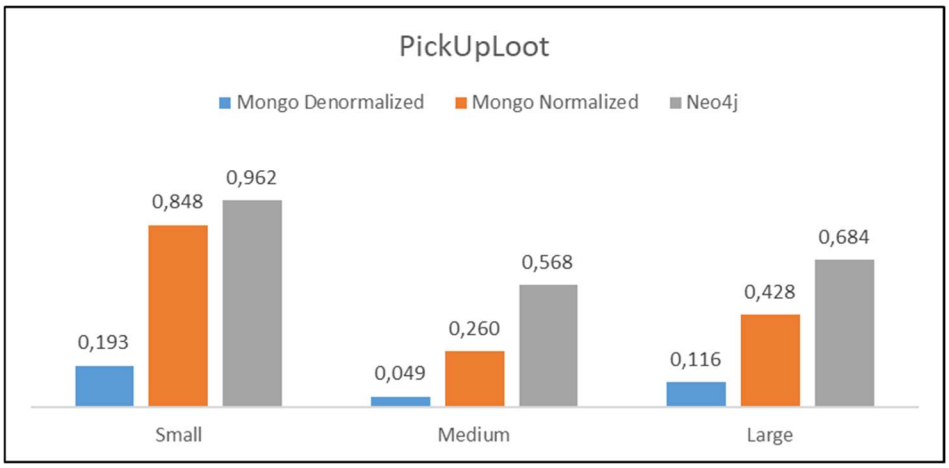

Рисунок 5.6 – Порівняння середнього часу виконання запиту PickUpLoot

Загалом, можна сказати, що тут проявляється одна з сильних сторін СКБД Neo4j і відносно слабка частина MongoDB: запити, що складаються з кількох команд та транзакції.

Neo4j досить непогано справляється з цим запитом. Але сама структура БД дозволяє денормалізованій моделі значно покращити свою продуктивність відносно інших моделей, шляхом зменшення складності операцій.

Перейдемо до порівняння споживання оперативної пам'яті сервером СУБД (див. рис. 5.7).

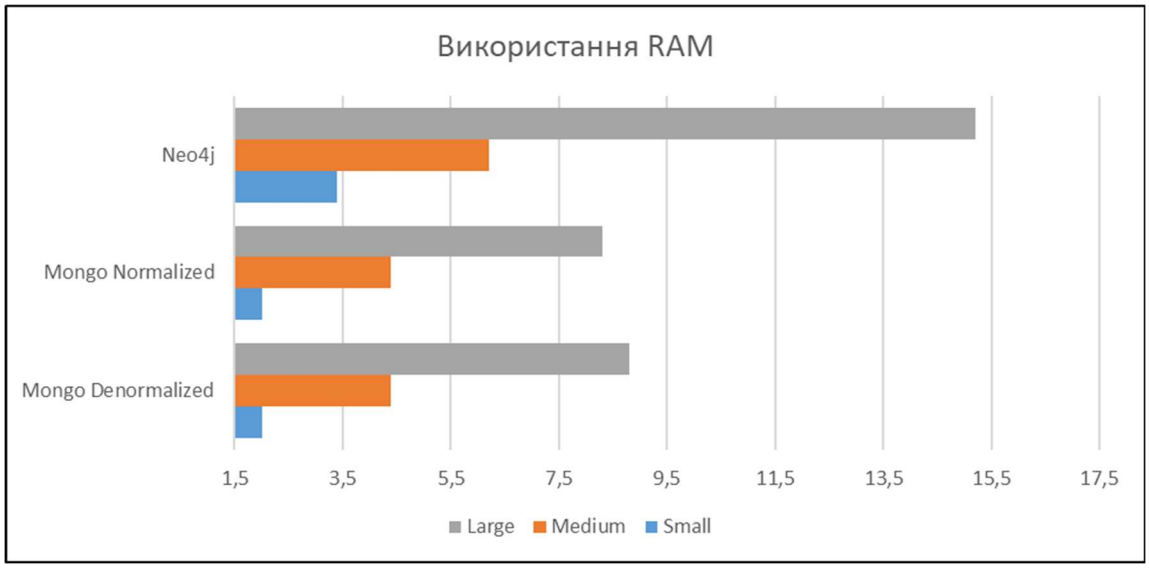

Рисунок 5.7 – Використання RAM

На гістограмі однозначно видно, що Neo4j використовує значно більше оперативної пам'яті, ніж MongoDB. У цій ситуації ще грає роль логіка виділення пам'яті в MongoDB – СУБД не може використовувати більше ніж 50% від оперативної пам'яті системи, тоді як Neo4j, за потреби, може використати всю доступну їй.

Також, Neo4j має умовно мінімальну кількість RAM для коректної роботи – 2 гігабайти, коли рекомендована кількість – взагалі близько 8 гігабайт.

Далі порівняємо використання CPU сервером СУБД. У дослідженні запропоновано використання трьох конфігурацій, тому дані метрики будуть представлені для всіх трьох конфігурацій систем.

Спочатку наведемо споживання CPU конфігурацією Small (див. рис. 5.8).

При не дуже великій кількості ресурсів, споживання СУБД Neo4j оперативної пам'яті та CPU здається доволі великим на тлі СУБД MongoDB.

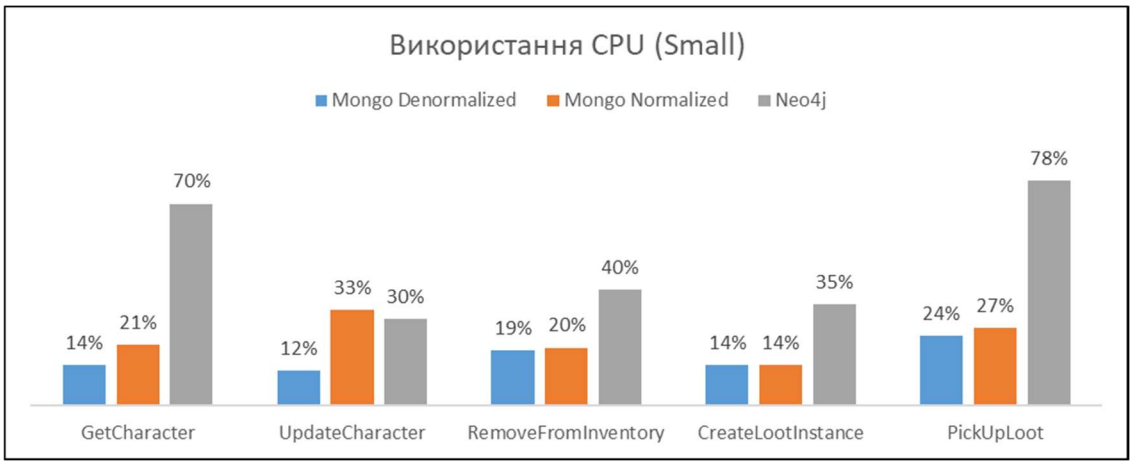

Рисунок 5.8 – Використання CPU (конфігурація Small)

З попередніх порівнянь і цих даних можна зробити однозначний висновок, що для невеликих проектів або проектів на стадії MVP Neo4j використовувати не особливо ефективно.

Тепер перейдемо до конфігурації Medium (див. рис. 5.9), вона вже більше відповідає реальним конфігураціям машин для проектів середніх розмірів.

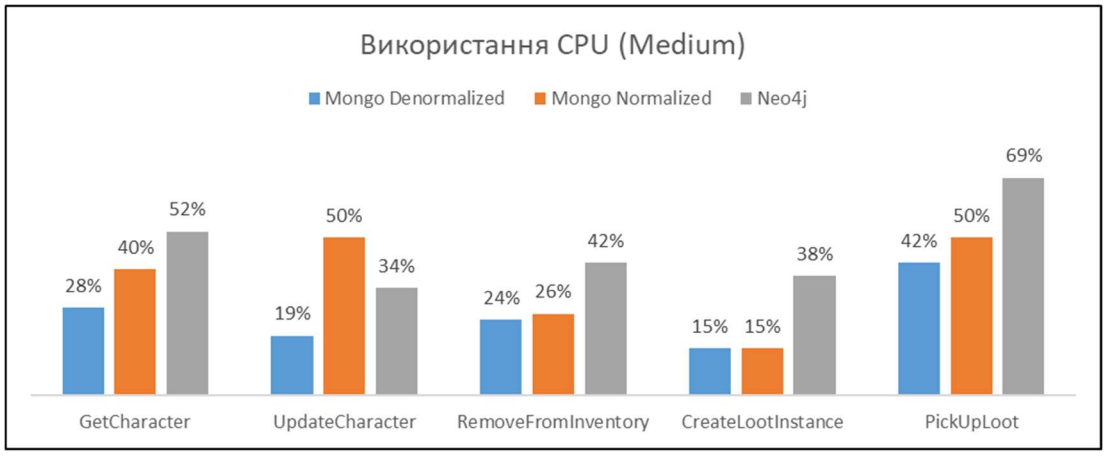

Рисунок 5.9 – Використання CPU (конфігурація Medium)

На такій кількості ресурсів ситуація для Neo4j вирівнялася щодо MongoDB, тепер дані СКБД приблизно рівні.

В цілому, саме на цій конфігурації, Neo4j почала працювати «без обмежень». Можна навіть зробити висновок, що виділених їй ресурсів навіть забагато для того навантаження, що було на неї виділено.

Насамкінець, переходимо до останньої конфігурації – Large (див. рис. 5.10).

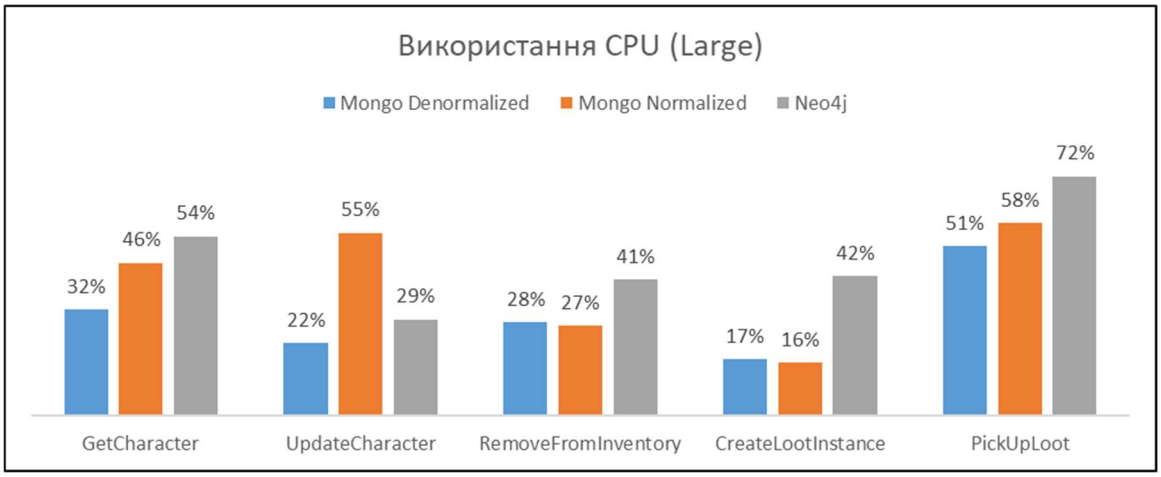

Рисунок 5.10 – Використання CPU (конфігурація Large)

Загалом, ситуація дуже схожа з результатами для конфігурації Medium.

Ресурсів стало більше, але пропорційно їм виросло навантаження та розмір БД. Тут теж можна сказати, що виділених СУБД ресурсів більше, ніж їм потрібно для стабільної роботи на даних навантаженнях.

Протягом усіх порівнянь видно певну картину – Neo4j споживає велику кількість ресурсів по відношенню до MongoDB, денормалізована модель працює швидше нормалізованої в контексті досліджуваних запитів, так само як і потребує менше ресурсів.

Після всіх порівнянь можна однозначно сказати, що для досліджуваної предметної області (моделі БД та запитів) найприйнятнішим варіантом є денормалізована модель MongoDB. Ця схема призвела до найменших витрат ресурсів машини за умови задовільної продуктивності.

5.2 Розробка рекомендацій щодо використання методів

Після проведення порівняння всіх отриманих результатів, можна зробити певні висновки та розробити деякі рекомендації щодо використання того чи іншого методу у певній ситуації.

Спочатку наведемо рекомендації для СКБД Neo4j. Ця СКБД знаходиться в розробці вже 15 років, протягом яких набула великого набору функціоналу, значного поліпшення продуктивності і так далі. Але, на практиці, все стає не так добре: обмеження Community версії, величезне споживання ресурсів CPU та RAM, велика вага БД порівняно з іншими СКБД і середня продуктивність у тривіальних завданнях роблять цю СУБД не особливо привабливою для проектів малого або середнього розмірів. Також слід зазначити, що вимогливість СКБД ще й пов'язана з її використанням JVM, що відразу накладає обмеження на мінімальну RAM для сервера СКБД.

Однією з особливостей СКБД також є те, що вона використовує максимум виділених їй ресурсів, так що їх необхідно жорстко обмежити до певних значень. Але, слід зазначити, що дана БД здатна легко виконувати операції, які складно виконувати в інших СКБД або взагалі неможливо.

Таким чином, рекомендується використовувати Neo4j у випадках, коли:

- схема бази даних містить велику кількість зв’язків M:M і логіка серверної частини передбачає часту вибірку декількох пов'язаних даних одночасно;
- схема БД має невелику частку сутностей на велику частку зв'язків і логіка додатку здебільшого полягає у видаленні та додаванні зв'язків між об'єктами БД;
- система велика, має велику кількість користувачів і компанія має в своєму розпорядженні велику кількість ресурсів;

– серверна частина потребує специфіки графових БД, наприклад, знаходження глибини зв'язків.

Тепер переходимо до СКБД MongoDB. У межах роботи розглядалося два варіанти моделі: нормалізована та денормалізована. Але є деякі моменти, які притаманні самій СУБД: досить низьке споживання ресурсів, висока швидкість читання, простота розгортання. Як таких, великих недоліків у СКБД немає, крім відсутності підтримки транзакцій, коли СКБД працює не у режимі кластера. Але цей недолік нівелюється тим, що MongoDB сьогодні рідко використовується в режимі Standalone.

Розглянемо спочатку метод нормалізації моделі. Використання цього методу призвела до нульової надмірності даних у БД, що позитивно позначилося на її вазі. Також, оскільки об'єкти БД значно менші, ніж у денормалізованій моделі, то вони мають між собою більше «схожості», СУБД більш ефективно застосувало механізм стиснення даних, що зберігаються (в середньому, на 5-15%).

Але аналізовані операції в досліджуваній серверній системі часто вимагали або з'єднання даних, або виконання операцій над кількома колекціями одночасно, що вимагає використання транзакцій або JOIN-подібних операцій. Це призвело до зниженої продуктивності порівняно з денормалізованою моделлю.

Таким чином, слід використовувати метод нормалізації, якщо:

– дані мають високу «самостійність» та планується нечасте оновлення/видалення декількох даних за один запит;
– зв'язки у схемі переважно мають кардинальність «0», що прибирає потребу штучно підтримувати цілісність даних за допомогою транзакцій (традиційний підхід «eventually consistent»);
– очікується, що у зв'язку 1:M, число M буде великим (та/або вага об'єкта велика). Це пов’язане з обмеженням розміру об'єкта в 16 MB;

- над об’єктами «M» у зв'язку 1:M будуть проводитися складні операції оновлення або фільтрації, оскільки масиви MongoDB мають менший інструментарій, порівняно з операціями над самими документами. Або, ці об’єкти необхідно більш складно проіндексувати;
- система раніше використовувала реляційну СУБД та потрібен швидкий перехід на MongoDB.

Тепер, розглянемо метод денормалізації. Використання цього підходу в досліджуваній серверній системі показало себе найефективнішим з погляду продуктивності. Але, надмірність даних відчутно збільшило вагу БД. Також деякі операції системи було досить важко реалізувати використовуючи операції над масивами (а деякі, потенційно, неможливо), що не притаманне нормалізованій моделі.

Цей метод слід застосовувати, якщо:

- дуже важлива швидкість операції читання і кожен запит має вибирати багато даних за зв’язками 1:1 або 1:M;
- кількість об'єктів «M» у зв'язку 1:M не особливо велика (до 1000) або залежні об'єкти не можуть існувати без головного (простіша «штучна» підтримка цілісності даних);
- ті об'єкти, дані яких продубльовані в інших об'єктах СКБД, рідко змінюються (операція Update), бо дана операція спричинить зміну даних у всіх них, що потребує багато ресурсів (CPU, запис на диск), якщо БД велика.

Таким чином, обидва дані підходу MongoDB не можна назвати однозначно оптимальними. Найбільш оптимальним буде змішаний підхід – комбінація нормалізації та деноралізації, подібності до реляційних зв'язків та композиції.

Загалом, можна однозначно сказати, що обидві досліджувані СУБД мають хорошу продуктивність, хоч і орієнтовані на різні завдання.

Якщо невідомо заздалегідь, як швидко буде розвиватися система, скільки в неї буде користувачів і так далі, то універсальним вибором є використання MongoDB. Дана СКБД має дуже широкий функціонал і здатність до горизонтального і вертикального масштабування, що робить її хорошим вибором для прототипів і нещодавно створених систем.

Таким чином, на основі результатів експериментального дослідження були сформовані рекомендації щодо використання досліджуваних методів. Ці рекомендації можуть бути використані для проектування реальних систем, зокрема в сфері ігрових серверів.

ВИСНОВКИ

У результаті кваліфікаційної роботи було досліджено проблемну область проектування NoSQL баз даних, зокрема, етап логічного проектування та різні існуючі методи логічного моделювання NoSQL баз даних.

Для виконання дослідження було обрано предметну область ігрових серверних систем. На прикладі деякої рольової гри було виконано моделювання предметної області, в ході якої була спроектована ER-діаграма БД та набір запитів, який використовує ця система.

Було обрано три способи логічного проектування БД: документний нормалізований, документний денормалізований та графовий. Для кожного з методів була розроблена відповідна логічна модель предметної галузі. На їх основі були розроблені відповідні фізичні моделі для СУБД MongoDB і Neo4j, а також набори запитів під дані моделі.

Під час дослідження було проведено планування експериментів: виділено основні метрики для порівняння ефективності методів та різні конфігурації серверів БД та даних.

Для виконання експериментів було розроблено ПЗ, яке реалізує функціонал для проведення дослідження: автоматичне розгортання віртуальних машин, заповнення їх тестовими даними, виконання запитів та замір необхідних метрик та формування звіту з результатами.

В результаті експериментів для кожної конфігурації були отримані необхідні метрики при виконанні запитів. На їх основі були розроблені рекомендації щодо практичного застосування досліджуваних методів для можливого їх використання при проектуванні баз даних для реальних ігрових систем.

За результатами дослідження було опубліковано тези «Дослідження методів логічного моделювання NoSQL баз даних для ігрових серверних

систем» на дванадцяту міжнародну науково-технічну конференцію «Сучасні напрями розвитку інформаційно-комунікаційних технологій та засобів управління» [22] та підготовлено до подачі наукову статтю «Методи логічного проектування NoSQL баз даних для MongoDB та Neo4J» у журналі «Сучасний стан наукових досліджень і технологій в промисловості» (див. додаток Г).

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Wikipedia: Информационный взрыв : веб-сайт . URL: https://ru.wikipedia.org/wiki/ Информационный_взрыв.

2. Graph Database. What’s the Big Deal? Towards Data Science : веб-сайт . URL: https://towardsdatascience.com/graph-databases-whats-the-big-deal-ec310b1bc0ed (дата звернення 02.05.2020).

3. Wikipedia: Проектирование баз данных : веб-сайт . URL: https://ru.wikipedia.org/wiki/Проектирование_баз_данных

4. Halpin T. Entity Relationship modeling from an ORM perspective: Part 1 // [Електронний ресурс]: Object Role Modeling. Електрон. текст, дані. 1999. URL: http://www.orm.net/pdf/JCM11.pdf (дата звернення 25.03.2022).

5. Дейт, К. Введение в системы баз данных. / К. Дейт. – М.: Наука, 1980. – 464 с.

6. Wikipedia: Denormalization : веб-сайт . URL: https://en.wikipedia.org/wiki/Denormalization

7. Sanders G. L., Shin S. K. Denormalization effects on performance of RDBMS / Proceedings of the 34th Annual Hawaii International Conference on System Sciences. 2001. 9 P.

8. Meier A., Kaufmann M. SQL & NoSQL Databases: Models, Languages, Consistency Options and Architectures for Big Data Management. – Springer Vieweg, 2019. – 248 P. – ISBN 978-3658245481.

9. Teorey T., Lightstone S., Nadeau T. Database Modeling and Design. – Elsevier, 2006. – 296 P. – ISBN 978-0-12-685352-0.

10. O. Mazurova, O. Samantsov, O. Topchii and M. Shirokopetleva, "A Study of Optimization Models for Creation of Artificial Intelligence for the Computer Game in the Tower Defense Genre," 2020 IEEE International Conference on Problems of Infocommunications. Science and Technology (PIC S&T), 2020, pp.

491-496, doi: 10.1109/PICST51311.2020.9468057.

11. Sullivan D. NoSQL for Mere Mortals. 1st Edition. – Addison-Wesley Professional, 2015. – 544 P. – ISBN 978-013-402-321-2.

12. MongoDB Manual. Data Model Design : веб-сайт . URL: https://www.mongodb.com/docs/manual/core/data-model-design/#embedded-data-models.

13. Glazer J., Madhav S. Multiplayer Game Programming: Architecting Networked Games (Game Design). – Addison-Wesley Professional, 2015. – 384 P. – ISBN 978-013-403-430-0.

14. Thor A. Massively Multiplayer Game Development (Game Development Series). – Charles River Media, 2004. – 484 P. – ISBN 978-158-450-243-2.

15. Stagner A. R. Unity Multiplayer Games. – Packt Publishing, 2013. – 242 P. – ISBN 978-184-969-232-8.

16. Bagui S., Earp R. Database Design Using Entity-Relationship Diagrams (Foundations of Database Design). – Auerbach Publications, 2011. – 371 P. – ISBN 978-143-986-177-6.

17. Date C.J. Database Design and Relational Theory: Normal Forms and All That Jazz. – Apress, 2019. – 470 P. – ISBN 978-148-425-539-1.

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

19. Vukotic A., Watt N., Abedrabbo T., Fox. D., Partner J. Neo4j in Action. – Manning, 2014. – 304 P. – ISBN 978-161-729-076-3.

20. Toroman M. Hands-On Cloud Administration in Azure: Implement, monitor, and manage important Azure services and components including IaaS and PaaS. – Packt Publishing, 2018. – 390 P. – ISBN 978-178-913-496-4.

21. Mazurova, O. Research of ACID transaction implementation methods for distributed databases using replication technology / Mazurova, O., Naboka, A., Shirokopetleva, M. Innovative technologies and scientific solutions for industries, (2 (16), pp. 19-31. Doi: 10.30837/ITSSI.2021.16.019.

22. Мазурова О.О., Сиволовський І.М. Дослідження методів логічного моделювання NoSQL баз даних для ігрових серверних систем // Сучасні напрями розвитку інформаційно-комунікаційних технологій та засобів управління.- Квітень 2022.- С. 148.