Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)
Кафедра Системотехніки
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)
«Дослідження моделей транзакцій в контексті проектування розподілених систем»
(тема)

Виконав:
студент 2 курсу, групи ІТПм-22-2
Савенков О. А.
(прізвище, ініціали)

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

Освітня програма Інформаційні технології проектування

Керівник доцент Білова Т. Г.
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри _____ _____
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук

Кафедра Системотехніки

Рівень вищої освіти другий (магістерський)

Спеціальність 122-Комп'ютерні науки

Тип програми освітньо-професійна

Назва програми Інформаційні технології проектування

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Савенкову Олексію Аркадійовичу

(прізвище, ім'я, по батькові)

1. Тема роботи «Дослідження моделей транзакцій в контексті проектування розподілених систем» затверджена наказом університету від «20» лютого 2023 р. №1373 Ст

2. Термін здачі студентом закінченої роботи 07 січня 2024 року

3. Вихідні дані до проекту аналіз проблем, пов'язаних з підтримкою консистенції даних у розподілених системах; розгляд і порівняння моделей транзакцій ACID та BASE, та їхніх впливів на продуктивність та доступність системи; дослідження сценаріїв, коли доцільно поєднувати моделі транзакцій ACID та BASE в одній розподіленій системі, вивчення переваг такого поєднання та його впливу на функціональність та продуктивність системи; реалізація розподіленої системи, що поєднує одночасно моделі транзакцій ACID та BASE проведення практичного експерименту, де порівнюються швидкодія та затримка консистентності даних у розподілений системі, що використовує моделі транзакцій ACID та BASE одночасно, та розподілених системах, що використовують лише модель ACID або модель BASE; розробка рекомендацій щодо вибору та реалізації моделі транзакції ACID, BASE або поєднання ACID та BASE в одній розподіленій системі;

4. Перелік питань, що потрібно опрацювати в роботі 4.1 Аналіз предметної області 4.2 Порівняльний аналіз методів підтримки консистенції даних у розподілених системах 4.3 Реалізація розподіленої системи, що використовує

одночасно моделі транзакцій ACID та BASE 4.4 Експериментальне порівняння швидкодії та консистентності даних у моделях транзакцій ACID та BASE

5. Перелік графічного матеріалу (з точним визначенням обов'язкових креслень):
5.1. Схема бекенду онлайн гри з використанням моделі транзакцій ACID 5.2 Схема бекенду онлайн гри з використанням моделі транзакцій BASE 5.3 Схема бекенду онлайн гри з використанням змішаної моделі транзакцій 5.4 Графік порівняння часу читання рядка 5.5 Графік порівняння часу запису рядка 5.6 Графік порівняння часу затримки консистенції бази даних

6. Консультанти розділів роботи

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Основний розділ	Доц. Білова Т.Г.		

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Постановка задачі. Збирання інформації за темою роботи.	09.09-15.09.2023	
2	Попередня рубрикація змістовної частини пояснювальної записки на розділи, підрозділи, пункти, підпункти зі стислими відомостями про їх зміст	16.09-24.09.2023	
3	Написання першого розділу роботи	25.09-06.10.2023	
4	Написання наступних розділів роботи	07.10-15.11.2023	
5	Формулювання загальних висновків	16.11-28.11.2023	
6	Оформлення матеріалів кваліфікаційної роботи	29.11.2023-31.12.2023	
7	Подання роботи до ЕК та її захист	01.01-12.01.2024	

Дата видачі завдання 7 вересня 2023 р.

Студент _____ Савенков Олексій Аркадійович
(підпис) (посада, прізвище, ініціали)

Керівник роботи (проекту) _____ доцент Білова Тетяна Георгіївна
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Робота містить: 103 сторінки, 27 рисунка, 3 таблиці, 8 додатки, 26 джерел.

КОНСИСТЕНТНІСТЬ ДАНИХ, МІКРОСЕРВІСИ, МОДЕЛІ ТРАНЗАКЦІЙ, ПРОЕКТУВАННЯ РОЗПОДІЛЕНИХ СИСТЕМ, РОЗПОДІЛЕНІ СИСТЕМИ, ACID, APACHE CASSANDRA, BASE, DYNAMODB.

Об'єктом дослідження є розподілені системи, що складаються з різних вузлів або серверів.

Предметом дослідження є підтримка консистенції даних у розподілених системи.

Мета дослідження полягає в аналізі, порівнянні та визначенні ефективних методів та стратегій для забезпечення консистенції даних в розподілених системах.

Методи дослідження – системний аналіз, практичний експеримент, спостереження, класифікація.

Наукова новизна дослідження полягає в тому, що воно пропонує поєднання різних моделей транзакцій ACID та BASE в одній розподіленій системі, визначає сценарії де таке поєднанная є доцільним, вивчає переваги такого поєднання на вплив та функціональність розподіленої системи, порівнюючи з моделями ACID та BASE окремо.

Галузь застосування – проектування розподілених систем з розподіленим сховищем даних.

ABSTRACT

Thesis contains: 103 pages, 27 images, 3 tables, 8 annexes, 26 sources.

ACID, APACHE CASSANDRA, BASE, DATA CONSISTENCY, DESIGN OF DISTRIBUTED SYSTEMS, DISTRIBUTED SYSTEMS, DYNAMODB., MICROSERVICES, TRANSACTION MODELS.

The object of research is distributed systems consisting of different nodes or servers.

The subject of the study is the maintenance of data consistency in distributed systems.

The purpose of the study is to analyze, compare and determine effective methods and strategies for ensuring data consistency in distributed systems.

Research methods - system analysis, practical experiment, observation, classification.

The scientific novelty of the research lies in the fact that it proposes a combination of different ACID and BASE transaction models in one distributed system, defines scenarios where such a combination is appropriate, studies the advantages of such a combination on the impact and functionality of a distributed system, comparing it with ACID and BASE models separately.

The field of application is the design of distributed systems with distributed data storage.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

ACID – модель транзакцій до бази даних, що включає атомарність, узгодженість, ізольованість, довговічність.

BASE – модель транзакцій до бази даних, що розшифровується як Basically Available, Soft state, Eventually consistent, та фокусується на доступності та швидкодії системи, а не на консистентності.

Моно-модель транзакцій – використання лише однієї моделі транзакції (ACID або BASE) у всій розподіленій системі.

Змішана модель транзакцій – використання ACID та BASE в одній розподіленій системі одночасно.

Лоад-балансер - load balancer.

ACID на розподіленому рівні – модель транзакцій у розподіленій базі даних, де усі принципи ACID (атомарність, узгодженість, ізольованість, довговічність) гарантуються на рівні всіх вузлів розподіленої бази даних, тобто транзакція має ізольовано атомарно узгоджено довговічно виконуватися одночасно на всіх вузлах розподіленої системи, та не може бути такого, що в якийсь момент часу транзакція виконалася на одному, але ще не виконалася на будь-якому іншому.

ВСТУП

Розподілені системи є важливою складовою сучасної інформаційної технології та комп'ютерних наук. Вони використовуються для обробки даних та виконання обчислень в розподіленому середовищі, де різні комп'ютери та сервери співпрацюють, об'єднуючи свої ресурси та дані.

Розподілені системи мають своє коріння в децентралізованих обчислювальних системах, які були поширені в середині 20-го століття. Однак справжній прорив в цій області стався в 1970-1980-х роках, коли з'явилися перші розподілені операційні системи та мережі, такі як UNIX та ARPANET [1]. Це відкрило двері до розвитку розподілених обчислювальних середовищ.

Моделі транзакцій ACID і BASE є двома основними підходами до управління консистентністю даних в розподілених системах [2]. Кожна з цих моделей має свою історію.

ACID (Atomicity, Consistency, Isolation, Durability) - ця модель була вперше впроваджена Едгаром Коддом в 1983 році і стала стандартом для багатьох транзакційних систем [3]. Atomicity вказує на те, що транзакція виконується як єдине ціле - вона або виконується повністю, або не виконується взагалі.

В контексті розподілених систем найпопулярнішим ACID методом підтримки консистенції транзакцій є протокол 2PC (двохфазний коміт) [4, 15]. Його основною метою є забезпечення атомарності транзакцій у розподілених середовищах, де різні компоненти системи можуть взаємодіяти з різними джерелами даних. Метод 2PC складається з двох основних фаз: підготовки та підтвердження. У фазі підготовки, координатор транзакції збирає підтвердження від усіх учасників транзакції щодо їх готовності виконати операції. У фазі підтвердження, якщо всі учасники підтвердили свою готовність, то координатор видає команду для усіх учасників на фінальне виконання транзакції.

BASE (Basically Available, Soft state, Eventually consistent) - ця модель є альтернативою ACID та набула популярності у розподілених системах з великим обсягом даних і високою доступністю [5]. BASE була розроблена як відповідь на вимогу до максимальної доступності та відсутності блокувань в розподілених середовищах.

Найпопулярнішим методом підтримки консистенції даних в розподілених системах використовуючи модель BASE є SAGA [6]. SAGA - це архітектурний шаблон для керування розподіленими транзакціями в мікросервісних системах. Цей підхід був вперше запропонований в 1987 році та востаннє отримав популярність завдяки розвитку мікросервісної архітектури. Основна ідея SAGA полягає в тому, що замість класичних ACID-транзакцій, які надають гарантовану консистентність даних, використовується покроковий підхід до вирішення конфліктів у розподілених середовищах.

Існує багато наукових джерел, де порівнюються моделі транзакцій ACID та BASE [7, 8, 9], а також методи підтримки консистенції 2PC та SAGA [10, 11, 12]. В усіх них висновком порівняння ACID та BASE є те, що якщо проект вимагає строгого керування транзакціями та потрібна гарантія високої цілісності даних, то рекомендовано вибирати ACID підхід. У випадках, коли доступність та швидкість є більш важливими, BASE є більш підходящим варіантом.

Проте, на даний момент відсутні дослідження сценаріїв, коли доцільно використовувати одночасно ACID та BASE в одній розподіленій системі, та порівняння такого поєднання з використанням лише ACID або BASE з точки зору функціональності та доступності розподіленої системи.

У даному дослідженні будуть проаналізовані переваги, недоліки використання поєднання моделей ACID та BASE в різних сценаріях розподілених систем, та буде розроблено довідник з рекомендаціями щодо вибору моделей транзакцій ACID, BASE та поєднання ACID та BASE в

одній розподіленій системі. Також, буде проведений практичний експеримент, що порівнює у конкретному сценарії швидкодію та консистентність даних у розподіленій системі, що використовує одночасно моделі транзакцій ACID та BASE, та двох аналогічних по функціоналу систем з моделями транзакцій ACID та BASE відповідно.

Результати дослідженння було апробовано на IV Всеукраїнської студентської наукової конференції "Експериментальні та теоретичні дослідження в контексті сучасної науки", Чернігів, 2023 рік [7].

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Опис сучасного стану розвитку розподілених систем

Розподілені системи стають все більш популярними в сучасному інформаційному середовищі [8], і їх використання вимагає ефективних методів керування консистенцією даних [8]. У цьому розділі будуть розглянуті основні поняття та принципи, які стосуються сучасного стану розподілених систем та забезпечення консистентності даних.

Розподілені системи - це складні інформаційні системи, які складаються з різних компонентів, що розташовані на різних фізичних машинах або в різних локаціях, і взаємодіють між собою через мережу [9]. Ці системи розподілені з метою підвищення масштабованості, доступності та надійності обробки даних та обчислень [9]. Застосування розподілених систем розповсюджуються на широкий спектр галузей і сфер. У сучасному світі розподілені системи стали невід'ємною частиною багатьох аспектів нашого життя. Ось декілька прикладів їх застосування: Інтернет-сервіси, Банківські системи [10], Системи управління складами та логістикою [11], Системи штучного інтелекту та аналізу даних [12] тощо. Основна ідея розподілених систем полягає в розділенні завдань між вузлами мережі та координації їхньої роботи [9]. Розподілені системи розділяють складні завдання на менші підзадачі, які можуть бути вирішені окремими вузлами. Це дозволяє досягти паралельності та рівномірного розподілу навантаження між різними частинами системи. Взаємодія між вузлами розподіленої системи здійснюється через мережу. Завдяки цьому можливо обмінюватися даними та повідомленнями між вузлами для спільного вирішення завдань. Синхронізація обчислень та обміну даними є важливим аспектом роботи розподілених систем. Це дозволяє уникнути конфліктів та забезпечити правильність результатів. Дані можуть бути розподілені між різними вузлами та зберігатися в розподіленій базі даних або системі збереження. Це забезпечує доступність та надійність даних. Розподілені системи повинні бути масштабованими, тобто можуть легко збільшувати

обчислювальні та зберігаючі ресурси для вирішення зростаючого обсягу завдань. Оптимізація використання ресурсів, які включають в себе обчислювальну потужність, мережу та зберігання даних, є ключовою частиною роботи розподілених систем.

Порівнюючи розподілену систему із сервером-монолитом, слід враховувати, що сервер-моноліт підходить для менших додатків та проектів з обмеженими ресурсами, де важлива простота розробки, тоді як розподілена система часто вибирається для великих та вимогливих за масштабом проектів, де висока продуктивність та надійність є першочерговими завданнями [9]. Вибір між ними залежить від конкретних потреб та вимог проекту, і обидві архітектури мають свої власні переваги та обмеження.

Масштабування сервера-моноліта є вертикальним (рисунок 1.1) і полягає в збільшенні ресурсів на одному сервері, наприклад, додаванні більш потужного процесора чи пам'яті [13]. Це підходить для додатків, які можуть бути оптимізовані за допомогою збільшення обсягу ресурсів на одному сервері. Однак цей метод має обмеження в масштабованості, і настає момент, коли подальше збільшення ресурсів стає надто витратним або фізично неможливим.

Масштабування розподіленої системи є горизонтальним (рисунок 1.1) полягає в додаванні нових вузлів до мережі з метою збільшення продуктивності та надійності [13].

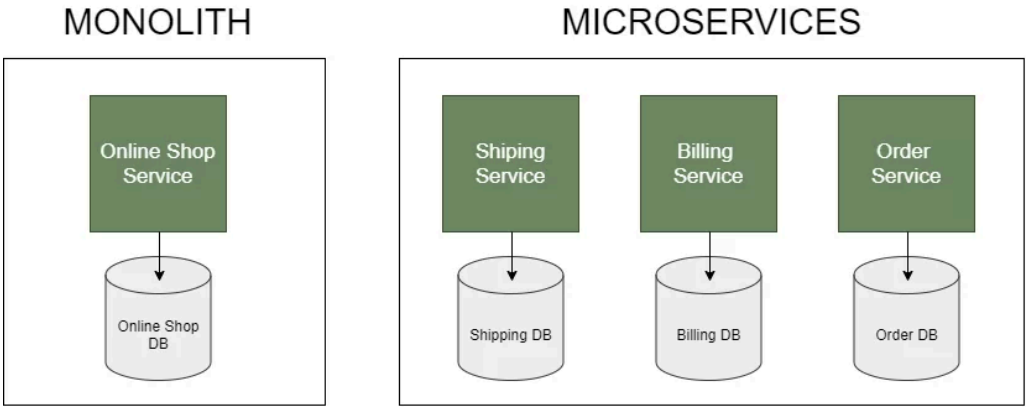


Рисунок 1.1 – Схеми архітектур сервера-моноліта та розподіленої системи на прикладі мікросервісної архітектури

В системі, що масштабується горизонтально, можна забезпечити високу доступність та стійкість до відмов [13].

Порівнюючи їх, можна сказати, що вертикальне масштабування підходить для менших і менш складних проектів, де обсяг роботи може бути обслугований на одному сервері [13]. Горизонтальне масштабування є більш практичним для великих та розподілених систем, де потрібно забезпечити високий рівень доступності та ефективно реагувати на зростання завдань [12]. У багатьох випадках системи використовують комбінацію обох підходів для досягнення оптимальних результатів.

Рисунок 1.2 – Візуалізація вертикального та горизонтального масштабування

Мікросервісна архітектура - це один із важливих та сучасних видів розподілених систем, який набуває популярності завдяки своїм перевагам у розробці, масштабованості та підтримці [14]. У мікросервісній архітектурі програмне забезпечення розбивається на невеликі і незалежні компоненти, відомі як мікросервіси [14]. Кожен мікросервіс відповідає за конкретну функціональність, і він може бути розвинутий та підтриманий окремо від інших мікросервісів [14]. Ця архітектура розроблена з урахуванням принципів декомпозиції, де складні системи розбиваються на менші та керовані частини для полегшення розробки та підтримки [14]. Такий підхід дозволяє розробникам розбити великі та складні системи на менші, автономні компоненти, що полегшує розробку, тестування та підтримку програмного

забезпечення. Мікросервісна архітектура також сприяє гнучкості в розгортанні нових функцій та вдосконаленнях, а також полегшує ведення коду в актуальному стані завдяки відокремленню функціональних блоків. Всі ці переваги роблять мікросервіси популярним вибором для розробки сучасних, масштабованих та легко управляємих програмних систем.

Мікросервіси можуть бути масштабовані незалежно один від одного, що дозволяє забезпечити ефективну роботу великих проектів [14]. Кожен мікросервіс може бути розроблений, тестований та впроваджений окремо, що сприяє прискоренню розробки та виправленню помилок. Помилки в одному мікросервісі не повинні впливати на решту системи, оскільки кожен сервіс функціонує незалежно. Додавання нових функцій може бути виконано без перекомпіляції всього додатку, що спрощує процес впровадження. Проте, мікросервісна архітектура також має свої виклики, включаючи управління багатьма мікросервісами, забезпечення комунікації між ними та докладну моніторинг і налагодження. Тим не менше, за правильних умов та при ретельному проектуванні, мікросервісна архітектура може сприяти створенню високоефективних, гнучких та масштабованих систем, які відповідають сучасним вимогам розвитку програмних продуктів.

Масштабування мікросервісів є ключовою складовою успішної їх імплементації [14]. Однією з важливих переваг такої архітектури є можливість гнучкого та ефективного масштабування окремих сервісів відповідно до їх завдань та обсягу навантаження. Кожен мікросервіс може бути масштабований незалежно від інших, що дозволяє оптимізувати ресурси та забезпечити відповідність вимогам конкретної частини системи. Масштабування може відбуватися горизонтально (додаванням нових інстанцій сервісу) або вертикально (покращенням ресурсів для окремого сервісу). Застосування мікросервісної архітектури дозволяє легко адаптуватися до змін у величині обсягу оброблюваних запитів та забезпечує високий рівень доступності системи. Однак важливо управляти цим процесом з урахуванням аспектів моніторингу, балансування навантаження та ефективного використання

ресурсів для досягнення оптимальної продуктивності та відповідності потребам користувачів.

Горизонтальне масштабування мікросервісів має кілька ключових переваг, що роблять його рекомендованим підходом для реалізації масштабованих та гнучких архітектур [14]. Воно надає гнучкість збільшувати чи зменшувати кількість інстанцій конкретного сервісу в залежності від навантаження без необхідності перегляду або зміни архітектури системи. Горизонтальне масштабування також дозволяє розподіляти навантаження між багатьма інстанціями мікросервісів. Це особливо важливо в умовах зростання обсягу користувацьких запитів, оскільки система може легко адаптуватися та забезпечувати високу продуктивність. Розподілення навантаження на багато інстанцій сприяє високій доступності системи. У випадку відмови одного сервісу, інші можуть продовжувати працювати, що забезпечує стабільну роботу системи. Горизонтальне масштабування дозволяє використовувати ресурси ефективніше, оскільки можна додавати нові інстанції на базі потреб кожного окремого сервісу. Це дозволяє оптимізувати використання обчислювальних та мережевих ресурсів.

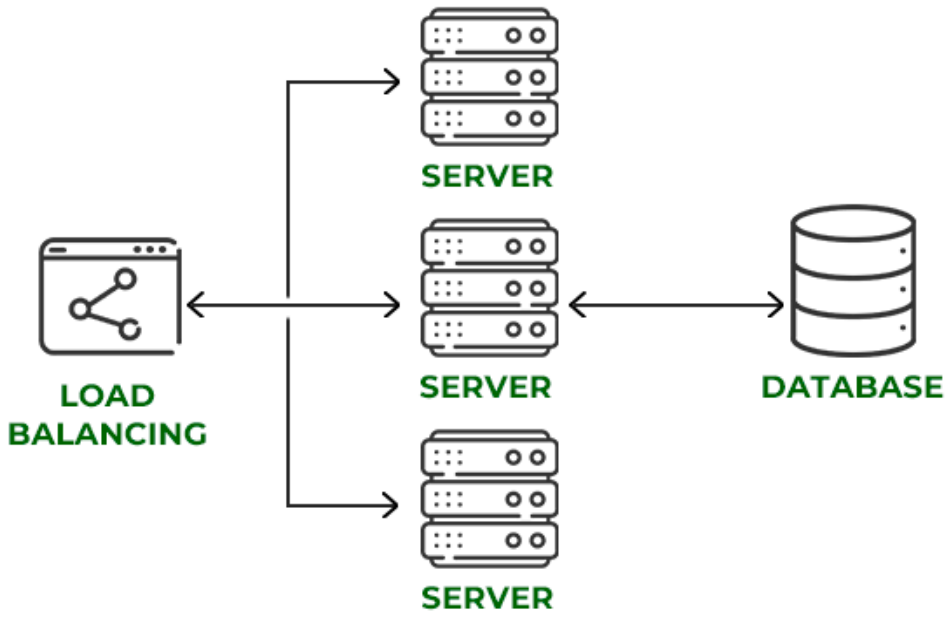


Рисунок 1.3 – Схема типового горизонтального масштабування серверів

У схемі типового горизонтального масштабування серверів (рисунок 1.3) застосунок працює наступним чином [13]:

— лоад-балансер виступає як центральна точка входу для запитів. Він розподіляє вхідний трафік між різними серверами в кластері, використовуючи різні алгоритми балансування навантаження (наприклад, Round Robin, Least Connections, або інші). Коли лоад-балансер отримує запит, він направляє його до одного з серверів у кластері. Сервер обробляє запит, можливо, взаємодіючи зі спільною базою даних;

— кластер серверів складається з багатьох інстанцій мікросервісів чи додатків, які обробляють вхідний трафік. Система може динамічно масштабувати кількість серверів в кластері в залежності від обсягу навантаження. Нові інстанції можуть додаватися або видалятися автоматично відповідно до потреб;

— у цьому сценарії використовується одна спільна база даних для всього кластера. Це може бути відносно проста база даних або розподілена система баз даних.

Ця конфігурація дозволяє системі ефективно розміщати велику кількість користувачів, динамічно масштабувати ресурси, забезпечувати високу доступність та забезпечувати спільний доступ до даних через централізовану базу даних. Однак важливо враховувати питання синхронізації та консистентності даних при такій конфігурації.

Масштабування серверів вирішує проблему недостатньої кількості обчислювальних ресурсів при збільшенні кількості одночасних запитів або просто збільшенні кількості обчислювального завдання. Але це є не єдиною точкою відмови. Навіть при достатній кількості обчислювальних ресурсів (серверів), при збільшенні кількості запитів до бази даних або обсягу даних масштабування лише серверів може не допомогти, бо можлива точка відмови у базі даних.

Для масштабування бази даних також існують вертикальні та

горизонтальні підходи [15]. Вертикальне масштабування передбачає збільшення продуктивності або обсягу роботи шляхом додавання ресурсів (потужності, пам'яті тощо) до існуючого сервера бази даних [15]. З іншого боку, горизонтальне масштабування полягає у використанні розподіленої архітектури бази даних, що може бути досягнуте шляхом створення кластера серверів бази даних, які працюють разом для обробки запитів [15].

Розподілені бази даних пропонують ряд значних переваг, які сприяють їх широкому використанню в сучасних інформаційних системах [15]. По-перше, розподілені системи надають велику масштабованість, оскільки можливість розподілити дані між різними серверами або вузлами дозволяє ефективно масштабувати інфраструктуру для обробки зростаючого обсягу інформації. Це робить розподілені бази даних ідеальними для великих підприємств та проектів з високими обчислювальними вимогами. Друга перевага полягає в високій доступності та надійності. Розподілена структура дозволяє вирішувати проблеми відмов чи відновлювати дані з резервних копій, що робить систему менш вразливою до виходу з ладу окремих компонентів. Це особливо важливо для бізнес-систем, які вимагають неперервної доступності до даних. Третя перевага пов'язана з географічною розподіленістю. Розподілені бази даних можуть бути розташовані в різних регіонах чи дата-центрах, що дозволяє ефективно обслуговувати користувачів по всьому світу та зменшувати час відповіді.

У прикладі розподіленої бази даних, що показано на рисунку 1.4, розподілена база даних складається з 3 кластерів. Один знаходитсья в Америці, другий в Європі, третій в Азії. Коли до застосунку звертається користувач, то його запит направляється лоад балансером на бекенд сервер, що знаходиться ближче всього до нього, а цей сервер робить запит до бази даних у цьому ж регіоні. Для підтримки консистенції між кластерами бази даних у різних регіонах відбувається асинхронна реплікація.

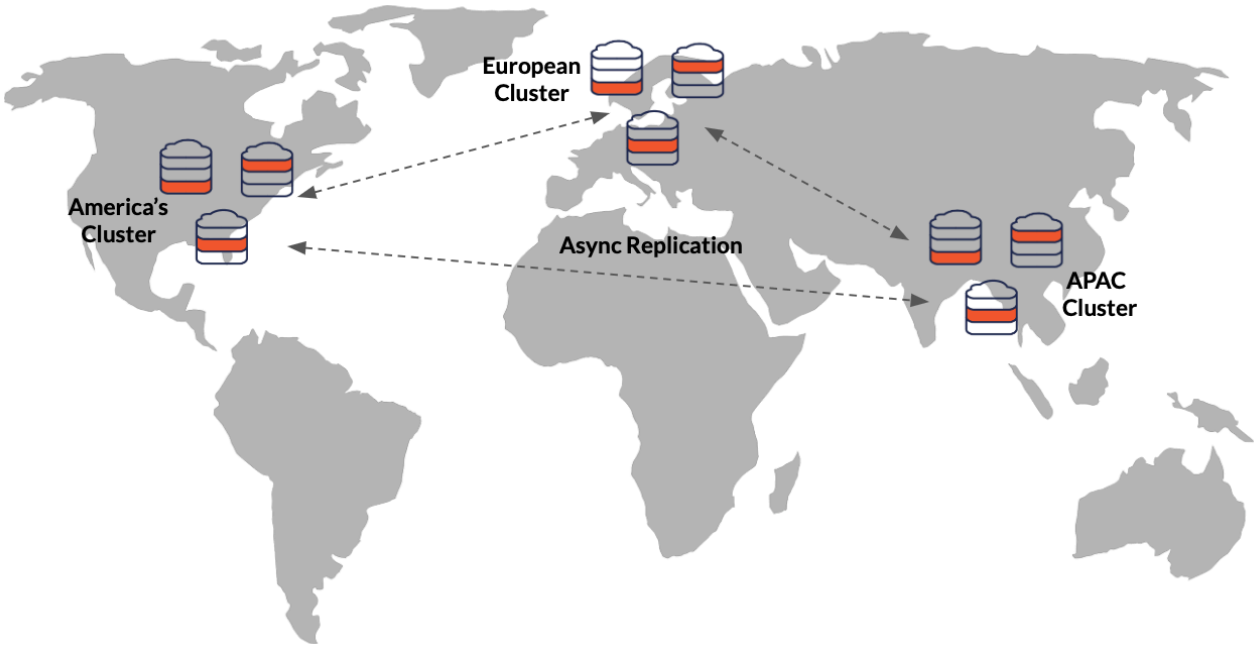


Рисунок 1.4 – Приклад розподіленої бази даних, що розташована у декількох регіонах світу

Асинхронна реплікація даних є однією з технік, які використовуються в розподілених базах даних для забезпечення високої доступності та стійкості системи [15]. У цьому підході зміни в базі даних передаються між вузлами чи серверами не негайно, а з певним затримкою, що дозволяє зменшити вплив мережі та інших факторів на продуктивність системи.

Основна перевага асинхронної реплікації полягає в тому, що вона може підвищити продуктивність та знизити витрати ресурсів, так як затримка дозволяє групувати зміни та передавати їх у пакетах. Це особливо корисно в великих системах з великою кількістю транзакцій.

Однак, незважаючи на переваги, асинхронна реплікація також може призводити до можливих конфліктів, оскільки часовий інтервал між змінами на різних вузлах може призвести до розбіжностей даних. Також, у випадку виходу з ладу одного з вузлів протягом затримки, дані на ньому можуть застаріти.

В цілому, вибір між асинхронною та іншими стратегіями реплікації залежить від конкретних потреб системи, вимог до доступності та консистентності даних.

Проблема консистентності даних у розподіленій базі даних є однією з ключових викликів, з якими стикаються розробники при роботі з розподіленими системами [16]. Консистентність в цьому контексті означає, що дані в базі повинні відображати поточний стан системи та її об'єктів у будь-який момент часу. Однак у розподілених середовищах, де дані розташовані на різних серверах чи вузлах, забезпечення консистентності стає важкою задачею. Проблеми виникають через можливість виникнення конфліктів при одночасних змінах даних на різних частинах системи. Наприклад, якщо два користувачі вносять зміни до одного й того ж об'єкта одночасно, може виникнути конфлікт, який ускладнює збереження консистентного стану.

Окрім мікросервісної архітектури та розподіленої архітектури бази даних, існує багато інших видів розподілених систем, які використовуються для різних завдань та областей застосування:

– системи для обробки BigData [17] (Apache Hadoop, Apache Spark, та Apache Flink тощо);

– розподілені файлові системи [18] (Hadoop Distributed File System (HDFS), Ceph тощо);

– системи для розподіленого обчислення [19] (Kubernetes, Docker Swarm, Apache Mesos тощо);

– розподілені сховища даних [20] (Amazon S3, Google Cloud Storage, Microsoft Azure Blob Storage тощо).

Розподілені системи можуть складатися з розподілених підсистем декількох видів, наприклад у одному серверному застосунку може використовуватися одночасно і мікросервісна архітектура, і розподілена база даних, і розподілене сховище даних тощо.

У цьому дослідженні в прикладах розподілених систем будуть використовуватися переважно серверні застосунки з мікросервісною архітектурою та розподіленою базою даних, хоча проблеми та методи їх вирішення, що розглядаються у дослідженні, притаманні усім видам розподілених систем.

Розподілені системи є актуальними та важливими в сучасному світі інформаційних технологій, оскільки вони створюють інфраструктуру для обробки та обміну великими обсягами даних, забезпечують гнучкість та масштабованість, а також сприяють ефективному використанню обчислювальних ресурсів [9]. Розподілені системи дозволяють компаніям та організаціям забезпечити високу доступність та надійність своїх послуг, навіть у випадку відмови окремих компонентів чи серверів.

У сучасному світі зростає кількість даних, які необхідно зберігати, обробляти та аналізувати, і розподілені системи грають ключову роль у вирішенні цього завдання [9]. Розподілені бази даних та системи обробки Big Data дозволяють компаніям ефективно керувати великими обсягами інформації та використовувати її для прийняття рішень.

Хмарні обчислення є ще однією актуальною областю, де розподілені системи використовуються для надання обчислювальної потужності та зберігання даних через Інтернет [13]. Це дає можливість компаніям зберігати та обробляти дані в децентралізованому середовищі, що сприяє ефективності та масштабованості.

Розподілені системи також стали невід'ємною частиною Інтернету речей (IoT), де підключені до мережі пристрої збирають великі обсяги даних [12]. Обробка цих даних вимагає розподілених систем для ефективного відстеження та аналізу інформації.

Проте, разом із своїми перевагами, розподілені системи також постають перед численними викликами. Проблеми такі, як безпека даних, синхронізація, керування ресурсами та складність розробки, вимагають ретельного вивчення та управління [9]. Вирішення цих викликів є ключовим завданням для подальшого розвитку та успіху розподілених систем у сучасному світі.

1.2. Аналіз проблем, пов’язаних з підтримкою консистенції даних у розподілених системах

У розподіленій архітектурі застосунку, що складається із декількох мікросервісів, які у свою чергу являють собою горизонтально масштабовані кластери серверів, що взаємодіють із розподіленою горизонтально масштабованою базою даних, проблеми пов’язані з підтримкою консистенції даних можна умовно поділити на наступні види:

— проблеми консистенції даних, що виникають у мікросервісній архітектурі між мікросервісами [21];

— проблеми консистенції даних, що виникають у межах одного сервісу між серверами одного горизонтально масштабованого кластеру [21];

— проблеми консистенції даних, що виникають у розподіленій базі даних між вузлами бази даних [21].

Далі розберемо кожен вид проблеми окремо.

1.2.1 Проблеми консистенції даних у мікросервісній архітектурі між мікросервісами

Мікросервісна архітектура, яка стала популярною в розробці програмного забезпечення, принесла багато переваг, таких як гнучкість, масштабованість та легка розгортка. Однак, разом з усім цим, виникають і певні проблеми, зокрема проблеми консистенції даних між мікросервісами.

Однією з основних проблем є відсутність централізованого сховища даних [21], оскільки кожен мікросервіс має свою власну базу даних. Якщо одні і ті самі дані зберігаються у декількох мікросервісах, це може призводити до розбіжностей у даних між сервісами, що ускладнює управління консистентністю.

Ще однією проблемою є атомарність операцій [21]. Якщо одна операція повинна змінити дані в кількох мікросервісах, виникає ризик того, що операція

в одному сервісі виконається успішно, а в іншому – ні. Це може привести до неконсистентного стану системи.

Також, асинхронні комунікації між мікросервісами можуть впливати на консистентність даних [21]. Затримки або втрати повідомлень можуть викликати ситуації, коли один сервіс працює з застарілими або неправильними даними.

Для вирішення цих проблем потрібно або використовувати механізм розподіленої атомарної транзакції [4], наприклад двохфазний коміт, що зменшить швидкодію системи, або перепроектувати систему таким чином, можливо об’єднавши мікросервіси, щоб не було потреби у синхронизації між ними.

1.2.2 Проблеми консистенції даних у межах одного сервісу між серверами одного горизонтально масштабованого кластеру

Проблеми консистенції даних у межах одного сервісу між серверами одного горизонтально масштабованого кластеру можуть виникати у таких випадках:

— якщо дані зберігаються у розподіленій базі даних, і різні сервери одного кластера працюють з різними вузлами бази даних [21]. Ця проблема буде розглянута у підрозділі 1.2.3;

— якщо якісь дані зберігаються на жорстких дисках або в оперативній пам’яті окремих серверів [21]. Дана проблема може трапитись, якщо даний сервіс раніше був монолітом і зберігав якісь дані в умовному локальному кеші на жорсткому диску або оперативній пам’яті. Після перетворення сервісу на масшатобваний кластер декількох серверів, без змін до логіки локального кешу він може почати працювати неправильно, бо він фактично перетворюється на багато окремих локальних кешів, що працюють в контексті лише окремого

серверу. Щоб він працював правильно та не потребував сихнронізації між серверами, потрібно використовувати кеш не прив'язаний до конкретного серверу, наприклад Redis.

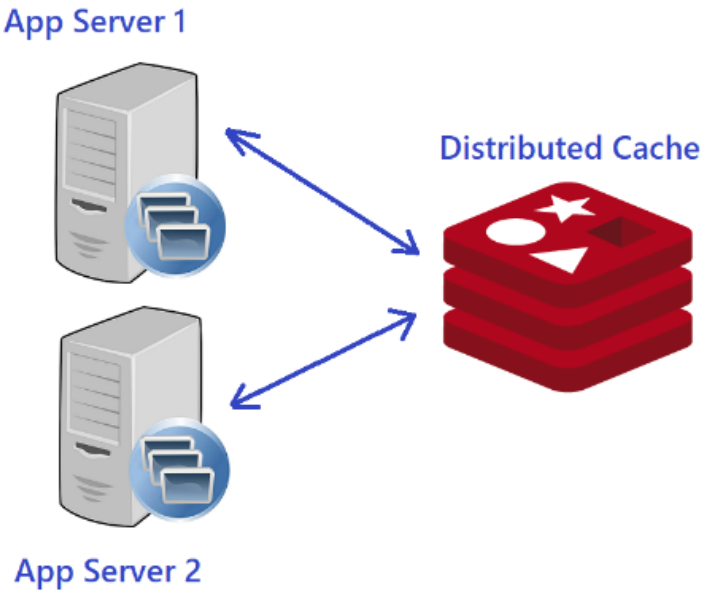


Рисунок 1.5 Приклад реалізації кешу у розподіленій системі

1.2.3 Проблеми консистенції даних у розподіленій базі даних між вузлами бази даних

Проблеми консистенції даних у розподіленій базі даних між вузлами можуть виникнути внаслідок асинхронності, затримок у мережі та розподіленої природи системи [22]. Однією з основних проблем є забезпечення узгодженості даних між різними вузлами бази даних. При асинхронній реплікації або розподіленому зберіганні даних, може виникнути ситуація, коли один вузол має більш актуальні дані, ніж інший, що призводить до неузгодженості.

Затримки у мережі можуть також призводити до ситуацій, коли вузли бази даних не мають негайного доступу до оновлень, і це може викликати розбіжності в станах даних [22]. Чим більше розподілені системи, тим більше ймовірність затримок і труднощів у збереженні консистентності.

Додатково, проблема консистенції може виникнути внаслідок конфліктів

при паралельних записах або читаннях декількох вузлів бази даних. Без відповідних механізмів синхронізації це може вести до суперечливих результатів та неоднозначності в структурі даних.

Можливі варіанти вирішення проблем консистенції в розподіленій базі даних:

— використання розподіленої бази даних, де можливо виконувати транзакції у всіх вузлах бази даних одночасно без тимчасової втрати консистенції, за рахунок втрати швидкодії операції. Це може буде, наприклад розподілена база даних Apache Cassandra;

— якщо тимчасова неконсистентність даних не призведе до серйозних наслідків, можна її ігнорувати.

1.3 Постановка задачі

1.3.1 Об’єкт та предмет дослідження

Об'єктом дослідження є розподілені системи, що складаються з різних вузлів або серверів.

Предметом дослідження є підтримка консистенції даних у розподілених системи. Дослідження в цьому контексті стосується аналізу і порівняння різних підходів та стратегій для забезпечення консистентності даних та/або швидкодії у системах, які працюють у розподіленому середовищі. Основні аспекти об'єкта дослідження включають:

– транзакції в розподіленому середовищі – вивчення того, як системи обробляють транзакції, що включають в себе різні операції над даними, які можуть виконуватися на різних вузлах чи серверах;

– консистентність даних – дослідження методів забезпечення, що дані залишаються консистентними після виконання розподілених транзакцій у великих і складних системах;

– моделі транзакцій – дослідження двох різних транзакційних моделей

ACID та BASE, які використовуються для забезпечення консистентності у розподілених системах.

1.3.2 Вхідні та вихідні дані

Дослідження передбачає збір, аналіз та порівняння великої кількості інформації про розподілені системи, транзакції та консистентність даних.

Вхідні дані:

– загальна інформація про розподілені системи, обчислення та транзакції;

– опис моделей транзакцій ACID та BASE;

– опис методу підтримки консистенції даних у розподілених даних 2PC, що використовується в моделі транзакції ACID;

– опис методу підтримки консистенції даних у розподілених даних SAGA, що використовується в моделі транзакції BASE.

Вихідні дані:

– порівняльний аналіз – оцінка і порівняльний аналіз моделей транзакцій ACID, BASE, а також поєднання ACID та BASE в одній розподіленій системі;

– результати аналізу доцільності поєднання ACID та BASE в одній розподіленій системі;

– рекомендації щодо вибору та реалізації моделі транзакцій в розподіленій системі, рекомендації з найкращими практиками щодо вибору ACID, BASE або ACID-BASE Hybrid для конкретної розподіленої системи;

– реалізована розподілена система, що поєднує одночасно моделі транзакцій ACID та BASE;

– результати практичного експерименту, де порівнюються швидкодія та затримка консистентності даних у розподілений системі у моделях транзакцій ACID та BASE.

1.3.3 Мета та задачі дослідження

Мета дослідження полягає в аналізі, порівнянні та визначенні ефективних методів та стратегій для забезпечення консистенції даних в розподілених системах.

В рамках поставленої мети слід вирішити наступні задачі:

– аналіз і вивчення проблем, пов'язаних з консистенцією даних в розподілених системах; вивчення різних підходів до досягнення консистенції даних;

– розгляд і порівняння моделей транзакцій ACID та BASE, та їхніх впливів на продуктивність та доступність системи;

– дослідження сценаріїв, коли доцільно поєднувати моделі транзакцій ACID та BASE в одній розподіленій системі, вивчення переваг такого поєднання та його впливу на функціональність та продуктивність системи;

– реалізація розподіленої системи, що поєднує одночасно моделі транзакцій ACID та BASE;

– проведення практичного експерименту, де порівнюються швидкодія та затримка консистентності даних у розподілений системі, що використовує моделі транзакцій ACID та BASE одночасно, та розподілених системах, що використовують лише модель ACID або модель BASE;

– розробка рекомендацій щодо вибору та реалізації моделі транзакції ACID, BASE або поєднання ACID та BASE в одній розподіленій системі.

1.3.4 Актуальність дослідження

Тема дослідження є актуальною через постійний розвиток і розширення розподілених систем[9]. Сучасні розподілені системи стають все більш складними та розгалуженими, включаючи хмарні обчислення[13], мікросервісну архітектуру[14], інтернет речей[12] тощо. Ця складність призводить до нових викликів у забезпеченні консистентності даних. Обсяги

даних та обчислювальні завдання у розподілених системах постійно зростають. Забезпечення консистентності даних у великих обсягах стає складнішим завданням. Дані є одними з найцінніших активів для багатьох організацій. Невірні або несумісні дані можуть призвести до фінансових втрат, порушення законодавства та надання послуг низької якості. Забезпечення консистентності даних також пов'язане з питаннями безпеки та приватності, які є важливими аспектами у світі, де дані можуть бути піддаються злому або незаконному доступу. Загалом актуальність базується на кількох ключових чинниках: зростаюча складність та актуальність розподілених систем, підвищення обсягів даних і завдань, важливість даних.

1.3.5 Наукова новизна дослідження

Наукова новизна дослідження полягає в тому, що воно пропонує поєднання різних моделей транзакцій ACID та BASE в одній розподіленій системі, визначає сценарії де таке поєднанная є доцільним, вивчає переваги такого поєднання на вплив та функціональність розподіленої системи, порівнюючи з моделями ACID та BASE окремо. Також результатом дослідження є реалізована розподілена система, що поєднує одночасно моделі транзакцій ACID та BASE, а також проведений практичний експеримент, де порівнюються швидкодія та затримка консистентності даних у розподілений системі, що використовує моделі транзакцій ACID та BASE одночасно, та розподілених системах, що використовують лише модель ACID або модель BASE. На основі аналізу та практичного експерименту були розроблені рекомендації з найкращими практиками і рекомендаціями щодо вибору та реалізації моделей транзакцій ACID, BASE та поєднання ACID та BASE в одній розподіленій системі.

2 ПОРІВНЯЛЬНИЙ АНАЛІЗ МЕТОДІВ ПІДТРИМКИ КОНСИСТЕНЦІЇ ДАНИХ У РОЗПОДІЛЕНИХ СИСТЕМАХ

2.1 Порівняльний аналіз моделей транзакцій ACID та BASE

ACID (Atomicity, Consistency, Isolation, Durability) і BASE (Basically Available, Soft state, Eventually consistent) - це дві основні філософії в підтримці консистентності даних в розподілених системах. Обидва підходи створені для вирішення проблеми збереження даних в розподіленому середовищі, але підходять до неї з різних точок зору[2].

ACID - це традиційний підхід до забезпечення консистентності даних, який встановлює високі вимоги до транзакцій:

- атомарність (A): транзакція є атомарною, тобто вона виконується цілком або не виконується зовсім, що гарантує, що система залишається в консистентному стані після виконання транзакції, навіть в разі відмови або помилки під час виконання;
- консистентність (C): після виконання транзакції система повинна переходити в інший консистентний стан, де всі правила та обмеження щодо даних зберігаються та дотримуються;
- ізоляція (I): транзакції мають бути ізольованими одна від одної, тобто виконання однієї транзакції не повинно впливати на інші транзакції, які виконуються паралельно;
- тривалість (Durability): зміни, внесені транзакцією, повинні бути стійкими та зберігатися в системі навіть після відмови або перезапуску системи.

BASE, навпаки, виходить з ідеї, що абсолютної консистентності може бути важко досягнути у деяких випадках і, можливо, навіть не потрібно її досягати. Основні принципи BASE включають:

- Basically Available (Базова доступність) – система завжди готова відповідати на запити, хоча вона може повертати неостанні дані або

незавершені результати;

- Soft state (М'який стан) – система може змінювати свій стан і не вимагає інстантанеального оновлення в усіх частинах, деякі частини системи можуть бути більш актуальними, ніж інші;
- Eventually Consistent (Зрештою консистентна) – система врешті-решт забезпечує консистентність, але це може займати деякий час, дані можуть бути асинхронно синхронізовані між частинами системи.

Вибір між ACID та BASE залежить від конкретних вимог проекту[2]. Різниця між ACID і BASE полягає у тому, як вони балансують консистентність та доступність в розподілених системах. ACID надає більш суворі гарантії стосовно консистентності та стійкості даних, але за ціною затримок та обмежень у доступності. BASE надає більшу доступність і швидкість, але може призвести до тимчасової нестійкості даних. Вибір між цими підходами залежить від конкретних вимог проекту та бажаних компромісів між консистентністю і доступністю [2].

У певних випадках, таких як системи для соціальних мереж або онлайн ігор, BASE може бути прийнятнішим вибором, оскільки користувачі можуть допустити певний рівень неоднаковості даних, якщо це допомагає забезпечити швидку доступність та масштабованість. Однак у критичних застосуваннях, де точність та консистентність даних – це першочергові вимоги, ACID може залишатися найкращим вибором [5]. Більший список сценаріїв, де доцільно використовувати ACID та BASE наведено у таблиці 2.1.

Таблиця 2.1 – Приклади сценаріїв розподілених систем, розділених за доцільністю використання ACID та BASE

ACID	BASE
Банківські системи: В розподілених банківських системах, де велика кількість транзакцій відбувається одночасно, ACID гарантує, що перекази грошей або інші фінансові операції відбуваються надійно.	**Системи соціальних мереж:** Системи, як Facebook або Twitter, можуть використовувати BASE для забезпечення доступності та надійності. Вони можуть допускати певну консистентність, якщо коментарі або лайки

Продовження таблиці 2.1

ACID	BASE
Atomicity гарантує, що операція виконується або повністю, або не виконується взагалі, і ця гарантія є критично важливою в банківському секторі.	з'являються з деякою затримкою, але гарантують, що користувачі можуть завжди звертатися до своїх профілів та взаємодіяти з іншими користувачами.
Системи управління запасами: В розподілених системах для управління запасами ACID може бути використаний для контролю стану товарів та забезпечення транзакційної цілісності. Наприклад, при оновленні інформації про залишки товарів на складі, Atomicity гарантує, що операція виконується безпомилково.	**Системи аналітики:** При обробці великих обсягів даних, наприклад, в системах аналізу великих даних (Big Data), головним завданням може бути швидкість та доступність даних, а не жорстка консистентність. Системи можуть використовувати BASE для оптимізації швидкості обробки та доступності результатів аналізу.
Системи бронювання та резервування: У розподілених системах бронювання (наприклад, авіаквитків або готелів), ACID гарантує, що клієнти отримують надійну та коректну інформацію про доступність та резервації. Consistency забезпечує, що жодна одиниця не може бути зарезервована більше одного разу.	**Ігрові системи:** В онлайн-іграх важливо, щоб гравці могли негайно взаємодіяти один з одним та з ігровим світом. BASE може бути використаний для забезпечення низької латентності та доступності гри, навіть якщо деякі дії гравців можуть з'являтися з затримкою на сервері.

Прикладом такої системи є онлайн гра, у якій можливо купити віртуальні ігрові речі справжніми грошима. На відміну від онлайн ігор, де присутні окремі сервери і гравці можуть взаємодіяти між собою лише в межах одного сервера на якому знаходяться, у цю гру грають одночасно по всьому світу ли та немає розділення ігрового світу на окремі сервери, тому необхідно, щоб точки доступу до бекенду та репліки баз даних були на всіх континентах для швидкого доступу гравців до стану гри, де б вони не були. Припустимо, що серверна частина гри складається з наступних мікросервісів:

- мікросервіс ігрового світу – тут відбувається вся логіка та зберігається вся інформація, що пов’язана з ігровим світом, а саме локація, стан ігрових та неігрових персонажів (NPC), стан об’єктів;
- мікросервіс ігрового інвентарю – тут відбувається вся логіка та

зберігається вся інформація, що пов’язана з інвентарем гравців. Всі предмети в інвентарі вважаються власністю гравців. Можливі дії з інвентарем включають отримання гравцем нового предмета, у тому числі через придбання його за реальні кошти, передача ігрового предмету від одного гравця до іншого тощо;

- мікросервіс оплат – тут відбувається вся логіка та зберігається вся інформація, що пов’язана з доступними для купівлі за справжні кошти віртуальні ігрові предмети, та з безпосередньо купівлею гравцями цих предметів. Основні дії що відбуваються в мікросервісі це перевірка чи предмет доступний для купівлі (моживо таке, що є обмежена кількість певного типу ігрового предмету для купівлі, і треба перевірити чи не був він куплений іншим гравцем), та купівля гравцем нового ігрового предмета.

Спочатку розглянемо переваги та недоліки вибору моделі ACID для всіх мікросервісів (рис. 2.1, табл. 2.2). У всіх наступних схемах для позначення мікросервісів використовуються прямокутники, а для позначення баз даних використовуються куби.

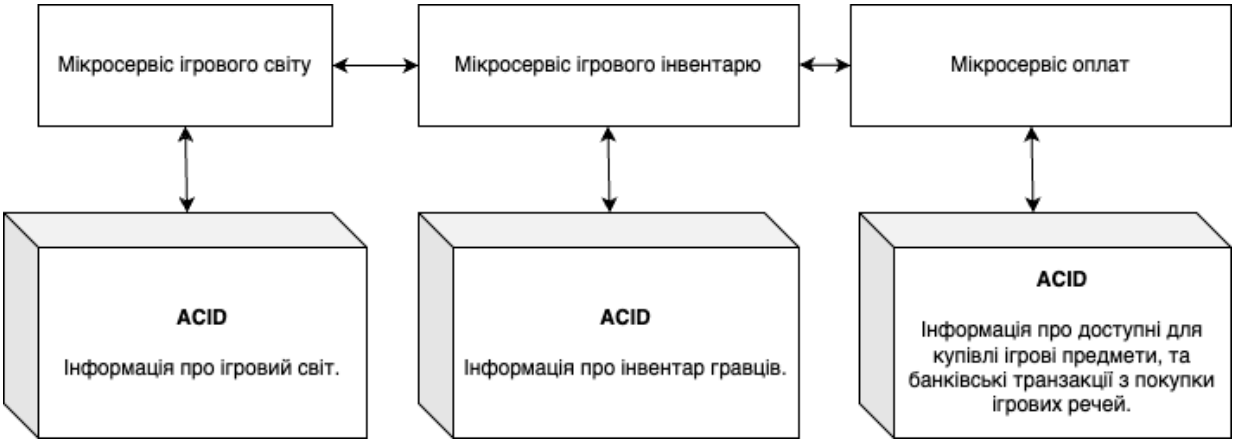


Рисунок 2.1 – Схема бекенду онлайн гри з використанням моделі транзакцій ACID

Таблиця 2.2. Переваги та недоліки вибору моделі транзакцій ACID для всіх мікросервісів бекенду онлайн-гри

ACID	Переваги	Недоліки
Ігровий світ	Дані будуть завжди консистентні у всіх репліках розподіленої бази даних. Це означає, що в один момент часу якщо всі гравці одночасно зроблять запит на стан ігрового світу, то у всіх гравців він буде на 100% однаковий.	Так як стан ігрового світу може змінюватися дуже багато разів на секунду, то підтримка консистенції всіх цих транзакцій дуже складна задача. Скоріш за все ці транзакції стануть у нескінченну чергу, та стан ігрового світу буде оновлюватися дуже повільно.
Ігровий інвентар	Стан ігрового інвентарю буде завжди консистентний у всіх гравців. Наприклад, якщо якийсь гравець подарує ігровий предмет іншому гравцю, то виконається ACID транзакція у всіх репліках бази даних одночасно, і неможливе таке, що якийсь короткий час предмет якийсь час буде у двох гравців одночасно тому, що дані у різних репліках бази даних не консистентні. Також порівняно з BASE дуже мала ймовірність, що якщо під час транзакції станеться помилка, то предмет виявиться у обох гравців на постійній основі, або ж навпаки - пропаде повністю з гри під час передачі предмета від одного гравця до іншого.	Операції з додаванням або видаленням речей з інвентарю будуть виконуватися довше порівнюючи з BASE через необхідність виконання транзакцій на всіх репліках одночасно.
Оплата	Порівняно з BASE дуже мала ймовірність того, що предмет буде оплачений, але гравець не отримає його через помилку. Також неможливо те, що якщо у платного предмета є обмежена кількість, то буде продано більше предметів ніж доступно	Операції з оплатою предметів будуть виконуватися довше порівнюючи з BASE через необхідність виконання транзакцій на всіх репліках одночасно.

Далі розглянемо переваги та недоліки вибору моделі BASE для всіх мікросервісів (рис. 2.2, табл. 2.3).

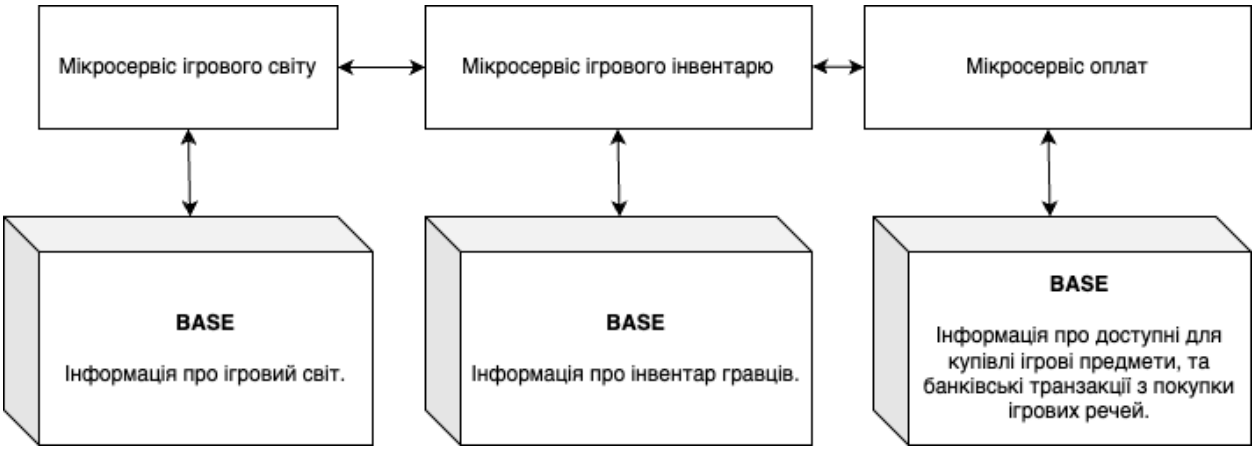


Рисунок 2.2 - Схема бекенду онлайн гри з використанням моделі транзакцій BASE

Таблиця 2.3. Переваги та недоліки вибору моделі транзакцій BASE для всіх мікросервісів бекенду онлайн-гри

BASE	Плюси	Мінуси
Ігровий світ	Оновлення ігрового світу гравцем відбувається дуже швидко в репліці бази даних, куди зробив запит гравець (далі ця зміна асинхронно пропагується в інші репліки, не блокуючи інші зміни)	Якщо в один момент часу всі гравці одночасно зроблять запит на стан ігрового світу, то у всіх гравців він може бути не однаковий на 100%. Деякі гравці можуть отримати застарілий стан. Якщо після внесення зміни в одну репліку бази даних станеться помилка і зміна не пропагуєтсья в інші, тоді можлива розсинхронізація стану ігрового світу у різних гравців
Ігровий інвентар	Оновлення ігрового інвентарю гравцем відбувається дуже швидко в репліці бази даних, куди зробив запит гравець (далі ця зміна асинхронно пропагується в інші репліки,	Стан ігрового інвентарю може бути не консистентний у всіх гравців. Наприклад, якщо якийсь гравець подарує ігровий предмет іншому гравцю, то можливе таке, що якийсь короткий час предмет якийсь час буде у двох

Продовження таблиці 2.3

	Плюси	Мінуси
Ігровий інвентар	не блокуючи інші зміни)	гравців одночасно тому, що дані у різних репліках бази даних не консистентні. Також порівняно з ACID існує більша ймовірність, що якщо під час транзакції станеться помилка, то предмет виявиться у обох гравців на постійній основі, або ж навпаки - пропаде повністю з гри під час передачі предмета від одного гравця до іншого.
Оплата	Операції з оплатою предметів будуть виконуватися швидше порівнюючи з ACID підходом через необхідність виконання транзакцій на всіх репліках одночасно.	Порівняно з ACID існує більша ймовірність того, що предмет буде оплачений, але гравець не отримає його через помилку у коді. Також можливе таке, що якщо у платного предмета є обмежена кількість, то буде продано більше предметів ніж доступно

Порівнюючи ACID та BASE підходи, що розповсюджуються на всю систему загалом, можна сказати, що жоден з них не підходить для побудови бекенду для такої масштабної онлайн гри, або подібної іншої розподіленої системи.

У ACID підході ігровий світ буде оновлюватися дуже повільно і скоріш за все у гру буде неможливо грати багатьом гравцям одночасно. Справа в тому, що світ може оновлюватися діями багатьох гравців дуже багато разів на секунду, і якщо для кожного такого оновлення виконувати двохфазний коміт або подібний ACID алгоритм для всіх реплік бази даних, тоді може утворитися нескінченна черга з таких транзакцій, які блокують одна одну.

У BASE підході критичним мінусом є те, що операції пов’язані зі справжніми грошима або предметами в інвентарі, що були куплені за справжні гроші, можуть призвести до неконсистентності даних, наприклад може бути

куплено більше предметів різними гравцями ніж доступно для покупки, або при передачі предмету від одного гравця до іншого на певний час (або на постійно якщо виникла помилка, що не була оброблена належним чином) цей предмет може бути присутній у обох гравців одночасно.

У даному випадку було б доцільно використовувати ACID підхід у мікросервісах, де критично важлива консистентність, та BASE підхід, де консистентність є менш важлива, ніж швидкість виконання операцій. У наступному підрозділі буде розглянуто такий змішаний підхід.

2.2 Проектування розподілених систем, що використовують моделі транзакцій ACID та BASE одночасно

У випадку масштабної онлайн-гри, яку було розглянуто у попередньому підрозділі, якщо вибирати ACID та BASE модель для кожного мікросервісу окремо, тоді має сенс:

Для мікросервісу ігрового світу обрати BASE підхід, тому що швидкість виконання великої кількість операцій багатьма гравцями в ігровому світі є більш пріоритетно, ніж можливість тимчасової неконсистентності між репліками сховища даних, що призведе до можливості неконсистентності ігрового світу між гравцями. Час який сховище буде не консистенте залежатиме від швидкості синхронізації вузлів сховища даних. Для мікросервісу ігрового світу як сховище даних можна обрати наприклад Redis, горизонтально масштабований кеш, з можливістю деплою в Redis Multi-Region Cluster [25] - декілька вузлів одного й того самого екземпляру кешу у різних регіонах світу. Кеш Redis не дотримується строгої консистенції [26], а тому вважається BASE підходом. Redis - це кеш, дані якого зберігаються в оперативній пам’яті, а не на диску, а тому операції запису виконуються набагато швидше ніж у базу даних, що нам і необхідно для мікросервісу ігрового світу з дуже великою кількістю записів.

Для мікросервісу оплат обрати ACID підхід. Фінансові транзакції повинні

виконуватися або повністю, або не виконуватися зовсім (гарантія цього у ACID підході є вищою, ніж у BASE). Це важливо для того, щоб уникнути ситуацій, де лише часткова транзакція виконана, і система залишається в непрогнозованому стані. Після успішного виконання транзакції її результати мають бути надійно збережені і витраиати будь-які збої системи (гарантія цього у ACID підході є вищою, ніж у BASE). У фінансовому контексті це особливо важливо для уникнення втрати даних та забезпечення невід'ємності операцій. Для мікросервісу оплат потрібно використовувати розподілену базу даних із строгою консистенцією між вузлами, наприклад Apache Cassandra.

Для мікросервісу інвентарів гравців обрати ACID підхід, тому що предмети в інвентарі можуть бути результатом фінансових транзакцій, тому для них важлива консистентність. З ACID стан ігрового інвентарю буде завжди консистентний у всіх гравців. Наприклад, якщо якийсь гравець подарує ігровий предмет іншому гравцю, то виконається ACID транзакція у всіх репліках бази даних одночасно, і неможливе таке, що якийсь короткий час предмет якийсь час буде у двох гравців одночасно тому, що дані у різних репліках бази даних не консистентні. Також порівняно з BASE дуже мала ймовірність, що якщо під час транзакції станеться помилка, то предмет виявиться у обох гравців на постійній основі, або ж навпаки - пропаде повністю з гри під час передачі предмета від одного гравця до іншого. Для мікросервісу інвентарю потрібно використовувати розподілену базу даних із строгою консистенцією між вузлами, наприклад Apache Cassandra.

Зі змішаним ACID/BASE підходом, описаним раніше, схема бекенду онлайн гри виглядатиме наступним чином (рис. 2.3).

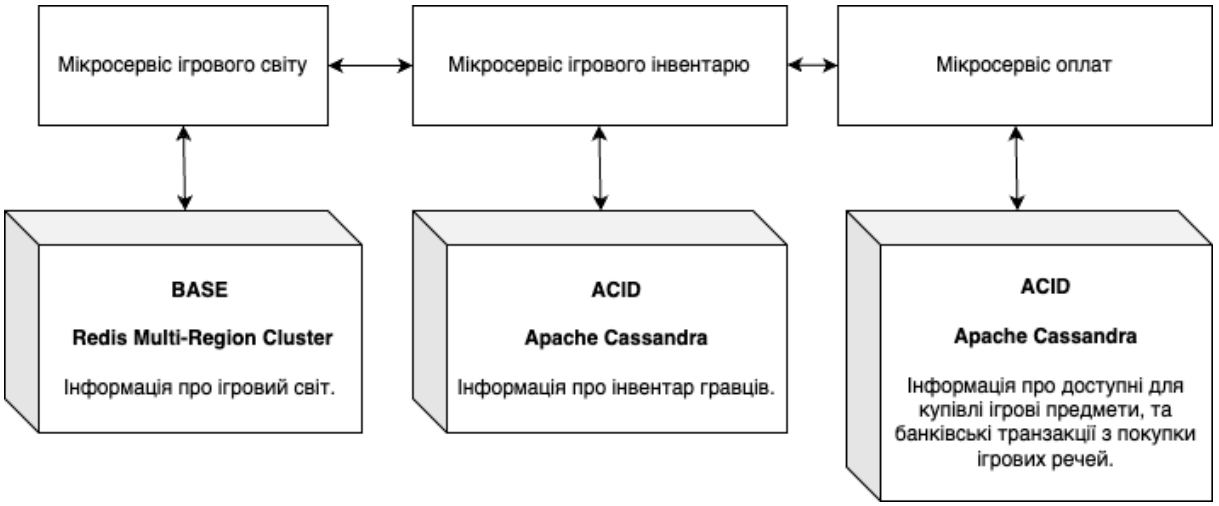


Рисунок 2.3 – Схема бекенду онлайн гри з використанням змішаної моделі транзакцій

Перевагою такого змішаного підходу є те, що баланс між консистентністю та швидкістю/доступністю можна налаштовувати у кожній підсистемі великої системи окремо. Такий підхід можна використовувати у будь-якій подібній розподіленій системі, не лише в масштабній онлайн-грі, що є лише поодиноким наочним прикладом.

Отже, вибір моделі транзакцій ACID, BASE або змішаної ACID/BASE у розподіленій системі можна зробити наступним чином:

– розбити розподілену систему на окремі логічні частини, наприклад мікросервіси;

– для кожної логічної частини системи описати плюси та мінуси вибору ACID та BASE для цієї конкретної підсистеми, та вплив на користувачів. У випадку ACID зазвичай основним плюсом є гарантія консистентності даних, а мінусом зменшення швидкості/доступності системи. Тут потрібно враховувати кількість та об’єм записів у сховище даних. Чим частіше та більші за об’ємом записи, тим серйозніше негативний вплив на швидкість/доступність розподіленої консистентної бази даних. У випадку BASE зазвичай основним плюсом є швидкість та доступність системи, а мінусом більша ймовірність неконсистентності даних;

– оцінити серйозність плюсів та мінусів кожного підходу у кожній

окремій підсистемі;

– якщо у всіх підсистемах можливість неконсистентності даних не може призвести до серйозних непоправних проблем, наприклад до фінансових втрат, або купівлі користувачем того, чого немає в наявності, тоді для простоти реалізації та підтримки системи програмістами має сенс обрати BASE підхід для всіх підсистем розподіленої системи;

– якщо у всіх підсистемах консистентність даних дуже важлива, наприклад система є банком або інтернет магазином, тоді має сенс обрати ACID підхід для всіх підсистем розподіленої системи;

– якщо у кожній підсистемі розподіленої системи вимоги відрізняються - у деяких критично важлива швидкодія, а в інших - консистентність, тоді має сенс обрати різні розподілені сховища даних для окремих підсистем, та використовувати різні моделі транзакцій в мікросервісах (або інших компонентах розподіленої системи).

3 РЕАЛІЗАЦІЯ РОЗПОДІЛЕНОЇ СИСТЕМИ, ЩО ВИКОРИСТОВУЄ ОДНОЧАСНО МОДЕЛІ ТРАНЗАКЦІЙ ACID ТА BASE

У даному розділі наведено приклад реалізації онлайн-гри, що було спроектовано з використанням моделей транзакцій ACID та BASE одночасно у попередньому розділі.

Реалізований код не є повною робочою онлайн грою, а фокусується лише на найбільш важливих моментах в контексті підтримки консистенції, швидкодії та виборі прийнятної моделі транзакції.

3.1 Реалізація мікросервісу оплат за ігрові предмети

Під час проектування гри у попередньому розділі було вирішено, що прийнятною моделлю транзакцій для мікросервісу оплат є ACID. ACID та strong consistency на рівні всієї багаторегіональної розподіленої системи буде досягнуто за рахунок використання розподіленої бази даних Apache Cassandra, у якій можна налаштовувати баланс між консистенцією та швидкодією, а отже її можна налаштувати на максимальну консистенцію за рахунок більш повільного виконання транзакцій. Це необхідно в мікросервісі оплат за ігрові предмети для того, щоб була якомога менше ймовірність що коли предмет буде оплачений, то гравець не отримає його через помилку в системі, або отримає із затримкою через помилку або затримку при перенесенні інформації з одного вузла бази даних до іншого. Також потрібно щоб було неможливе таке, що якщо у платного предмета є обмежена кількість, то буде продано більше предметів ніж доступно, що надійно реалізувати більш складно у BASE ніж ACID.

Python-фреймворк для обробки веб запитів, що буде використовуватися в мікросервісі, буде Flask.

Для роботи з базою даних Cassandra буде використовуватись бібліотека cassandra-driver, що встановлюється наступною командою пакетного менеджера pip (див. лістинг 5.1).

Лістинг 5.1 – Команда, що встановлює pip-пакет cassandra-driver

```
pip install Flask cassandra-driver
```

На початку програми імпортуємо потрібні модулі (див лістинг 5.2).

Лістинг 5.2 – Імпортування необхідних модулів для роботи з фреймворком Flask та базою даних Cassandra

```
from flask import Flask, request, jsonify
from cassandra.cluster import Cluster
from cassandra.auth import PlainTextAuthProvider
```

Далі ініціалізуємо фреймворк Flask (див. лістинг 5.3).

Лістинг 5.3 – Ініціалізація серверу Flask

```
app = Flask(__name__)
```

Далі ініціалізуємо зв’язок з базою Cassandra (див. лістинг 5.4).

Лістинг 5.4 – Ініціалізація зв’язку з базою даних Cassandra

```
# Cassandra connection setup
cassandra_auth = PlainTextAuthProvider(username='your_username',
password='your_password')
cassandra_cluster = Cluster(['127.0.0.1'],
auth_provider=cassandra_auth)
cassandra_session = cassandra_cluster.connect()
```

Далі ключовим для підтримки консистенції даних є налаштування зв’язку з базою даних таким чином, щоб за замовчуванням у всіх операціях читання та запису передбачався максимальний рівень консистенції (див. лістинг 5.5).

Лістинг 5.5 – Налаштування рівня консистенції QUORUM у зв'язку з базою даних Cassandra

```
cassandra_session.default_consistency_level =
cassandra.cluster.ConsistencyLevel.QUORUM
```

На рисунку 3.1 наведено список усіх варіантів рівня консистенції бази даних Cassandra.

Consistency. Level	Read Level Implication	Write Level Implication
ANY	Not Applicable	Ensure that the value is written to a minimum of one replica node before returning to the client, allowing hints to count as a write.
ONE, TWO, THREE	Immediately return the record held by the first node(s) that respond to the query. The record is checked against the same record on other replicas. If any are out of date, a read repair is then performed to sync them all to the most recent value.	Ensure that the value is written to the commit log and memtable of at least one, two, or three nodes before returning to the client.
LOCAL_ONE	Similar to ONE, with the additional requirement that the responding node is in the local data center.	Similar to ONE, with the additional requirement that the responding node is in the local data center.
QUORUM	Query all nodes. Once a majority of replicas **((replication factor / 2) + 1)** respond, return to the client the value with the most recent timestamp. Then, if necessary, perform a read repair on all remaining replicas.	Ensure that the write was received by at least a majority of replicas **((replication factor / 2) + 1).**
LOCAL_QUORUM	Similar to **QUORUM**, where the responding nodes are in the local data center.	Similar to **QUORUM** , where the responding nodes are in the local data center.
EACH_QUORUM	Ensure that a **QUORUM** of nodes respond in each data center.	Ensure that a **QUORUM** of nodes respond in each data center.
ALL	Query all nodes. Wait for all nodes to respond, and return to the client the record with the most recent timestamp. Then, if necessary, perform a read repair. If any nodes fail to respond, fail the read operation.	Ensure that the number of nodes specified by replication factor received the write before returning to the client. If even one replica is unresponsive to the write operation, fail the operation.

Рисунок 3.1 – Список усіх варіантів рівня консистенції операцій читання та запису бази даних Cassandra

У таблиці найвищим рівнем є ALL, а не QUORUM що був обраний у мікросервісі оплат. ALL передбачає те, що при виконанні запису запис відбувався зразу у абсолютно всіх серверах усіх датацентрів, а при виконанні читання воно відбувалося з усіх серверів усіх датацентрів, та повертався результат з сервера з останнім часом оновлення.

QUORUM у свою чергу передбачає що запис відбувається зразу у більшість серверів ((кількість серверів / 2) + 1), а запис відбувається також з більшості серверів ((кількість серверів / 2) + 1) та повертався результат з сервера з останнім часом оновлення.

З попереднього абзаца випливає, що якщо обрати рівень QUORUM і для

запису і для читання одночасно, то запит у базу гарантовано поверне останню версію даних, тому що як мінімум один сервер з довільного набору серверів читання ((кількість серверів / 2) + 1) та довільного набору серверів попереднього запису ((кількість серверів / 2) + 1) буде перетинатися.

Отже, рівень ALL не є необхідний, і достатньо обрати рівень QUORUM для запису і для читання.

Створимо Keyspace (аналог database у базах даних Postgresql, mysql тощо) game_payment та таблиці transactions та inventory (див. лістинг 5.6)

Лістинг 5.6 – Створення Keyspace game_payment та таблиць transactions і inventory

```
# Keyspace and table creation
cassandra_session.execute("""
    CREATE KEYSPACE IF NOT EXISTS game_payment
      WITH REPLICATION = {'class':'NetworkTopologyStrategy',
'us-east-1':'3','eu-west-1':'3'}
""")
cassandra_session.set_keyspace('game_payment')
cassandra_session.execute("""
    CREATE TABLE IF NOT EXISTS transactions (
        transaction_id UUID PRIMARY KEY,
        user_id UUID,
        item_id UUID,
        amount DECIMAL,
        status TEXT
    )
""")
cassandra_session.execute("""
    CREATE TABLE IF NOT EXISTS inventory (
        item_id UUID PRIMARY KEY,
        available_quantity INT
    )
```

"""）

У команді CREATE KEYSPACE параметр REPLICATION = {'class':'NetworkTopologyStrategy', 'us-east-1':'3','eu-west-1':'3'} означає що KEYSPACE у регіонах us-east-1 та eu-west-1 будуть існувати по 3 вузла бази даних.

Таблиця transactions зберігає інформація про виконані оплати.

Таблиця inventory зберігає інформацію про доступні предмети для продажу (їх кількість може бути обмежена).

Далі реалізуємо точку доступу HTTP, що оброблює оплату за віртуальних предмет. Повний код реалізації наведено у додатку Б.

Вказана реалізація гарантує, що предметів не буде продано більше ніж їх доступно в магазині онлайн гри. Це гарантується за рахунок:

— виконання перевірки наявності предметів та їх резервація в одній транзакції;

— налаштування консистенції даних рівня QUORUM у базі даних Cassandra. Це гарантує що резервація предметів одразу пропагується у більшість довільних вузлів бази даних, і при перевірці наявності предметів інший сервер зробить запит у більшість довільних вузлів, що гарантовано поверне останню інформацію про наявність.

3.2 Реалізація мікросервісу ігрового інвентарю

Під час проектування гри у попередньому розділі було вирішено, що прийнятною моделлю транзакцій для мікросервісу ігрового інвентарю, як і для мікросервісу оплат, є ACID. Це необхідно в мікросервісі інвентарю також, тому що операції в інвентарі можуть відбуватися з предметами, купленими за справжні гроші, а тому консистенція та запобігання втрат даних інвентарю є дуже важливими, що набагато простіше реалізувати в контексті моделі ACID ніж BASE. Консистенція у мікросервісі ігрового інвентарю буде досягнуто за

рахунок використання розподіленої бази даних Apache Cassandra, у якій можна налаштовувати баланс між консистенцією та швидкодією, а отже її можна налаштувати на максимальну консистенцію за рахунок більш повільного виконання транзакцій.

Python-фреймворк для обробки веб запитів, що буде використовуватися в мікросервісі, буде Flask.

Для роботи з базою даних Cassandra буде використовуватись бібліотека cassandra-driver, що встановлюється наступною командою пакетного менеджера pip:

Лістинг 5.7 – Команда, що встановлює pip-пакет cassandra-driver

```
pip install Flask cassandra-driver
```

Повний код реалізації наведено у додатку Б.

Найбільш ризикованою операцією у мікросервісі інвентарів гравців в контексті консистенції даних серед вказаних прикладів є обмін предметами гравцями, тому що необхідно щоб обмін відбувся або повністю або не відбувся. Це гарантується за рахунок:

— виконання всіх чотирьох операцій обміну предметами в одній транзакції;

— налаштування консистенції даних рівня QUORUM у базі даних Cassandra. Це гарантує що обмін предметами пропагується у більшість довільних вузлів бази даних, і при перевірці наявності предметів у гравця інший сервер зробить запит у більшість довільних вузлів, що гарантовано поверне останню інформацію про наявність у гравця. Це виключає сценарій, коли сервер в іншому регіоні може повернути старий стан інвентарю гравця через те, що вузли бази даних ще не синхронизувалися.

3.3 Реалізація мікросервісу ігрового світу

Під час проектування гри у попередньому розділі було вирішено, що прийнятною моделлю транзакцій для мікросервісу ігрового світу є BASE, тому що швидкодія є набагато важливіша ніж тимчасова неконсистентність даних між вузлами. Багато гравців можуть робити дуже багато операцій в секунду і підтримка консистенції всіх цих операцій призведе до значного збільшення часу виконання запитів до бази даних, що може негантивно вплинути на процес гри. Високий рівень швидкодії без строгої консистенції буде досягнуто за рахунок використання розподіленого кешу Redis.

Python-фреймворк для обробки http та websocket запитів, що буде використовуватися в мікросервісі, буде Flask та Flask_SocketIO.

Для роботи з кешом Redis буде використовуватись бібліотека redis-py, що встановлюється наступною командою пакетного менеджера pip (див. лістинг 5.8).

Лістинг 5.8 – Команда, що встановлює pip-пакет cassandra-driver

```
pip install Redis-Py
```

Повний код реалізації наведено у додатку Б.

4 ЕКСПЕРИМЕНТАЛЬНЕ ПОРІВНЯННЯ ШВИДКОДІЇ ТА КОНСИСТЕНТНОСТІ ДАНИХ У МОДЕЛЯХ ТРАНЗАКЦІЙ ACID ТА BASE

4.1 Визначення критеріїв оцінювання швидкодіїї та консистентності даних у мікросервісі

У цьому розділі будуть розгорнуті у хмарному середовищі, запущені та протестовані на швидкодію та консистентність даних фрагменти розподіленої онлайн-гри, код яких був реалізований у попередньому розділі.

Будуть протестовані час запису у базу даних, час читання з бази даних, та час затримки консистенції для моделей транзакцій ACID та BASE.

У кожному тесті модель транзакцій буде тестуватися наступним чином:

— розподілена база даних мікросервісу буде розгорнута у двох регіонах: Європі і Північній Америці;

— одна AWS Lambda функція, що буде розгорнута в одному з регіонів, буде записувати дані у базу даних, а друга AWS Lambda функція, що буде розгонута у другому регіоні, буде читати записані дані з бази даних.

Результати кожного тестування будуть представлені наступними полями, за якими можна наглядно оцінити швидкодію та консистентність даних:

— мінімальна, середня та максимальна швидкість обробки запиту запису у базу даних;

— мінімальна, середня та максимальна швидкість обробки запиту читання з бази даних;

— мінімальна, середня та максимальна затримка консистенції між запитами запису та читання.

Модель транзакцій ACID в експерименті буде представлена базою даних Apache Cassandra налаштованою на максимальну консистентність. Модель транзакцій BASE буде представлена базою даних DynamoDB.

На основі результатів проведеного експерименту будуть зроблені висновки щодо доцільності використання обох моделей транзакцій одночасно на прикладі онлайн-гри, спроектованої у попередньому розділі.

4.2. Тестування моделі транзакцій ACID

Перед початком тестування, потрібно розгорнути розподілену базу даних Apache Cassandra у двох регіонах. Для розгортання середовища тестування буде використовуватися переважно хмарний сервіс Amazon Web Services.

Одним з найпростіших способів розгортання бази даних Apache Cassandra є створення її у сервісі Amazon Web Services Keyspaces. Проте після спроби розгортання у AWS Kespaces (див. рис. 4.1, 4.2, 4.3) та запуску фрагментів коду з розділу 3.1 було виявлено що база даних створена у цьому сервісі не підтримує рівні консистенції QUORUM та ALL (див. рис. 4.4), що унеможливлює конфігурацію розподіленої бази даних зі строгою консистентністю даних та моделлю транзакції ACID на розподіленому рівні відповідно.

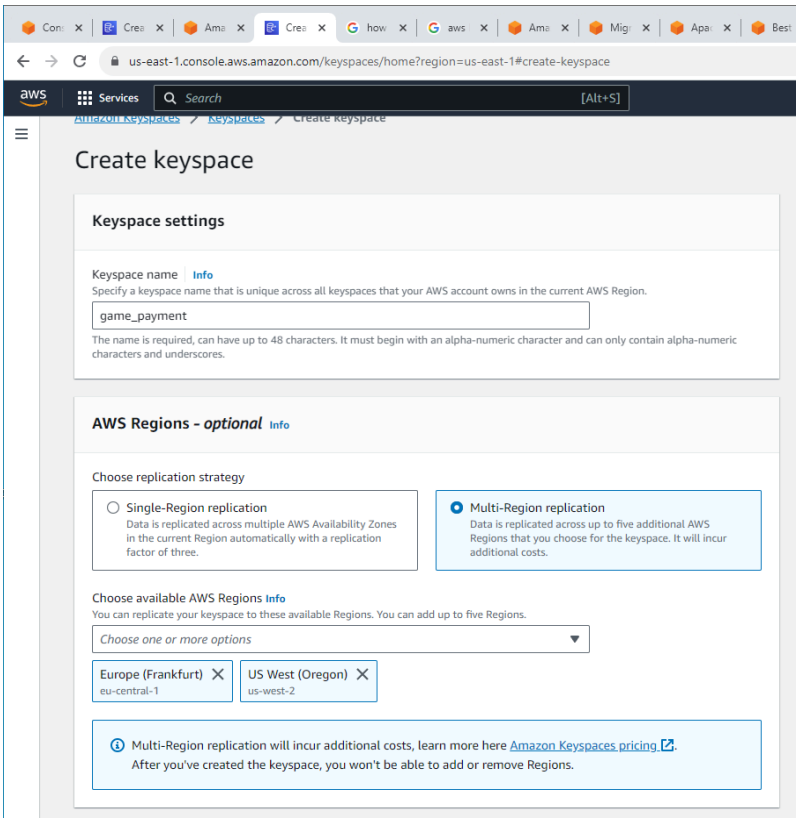


Рисунок 4.1 – Процес створення бази даних Cassandra у хмарному сервісі Amazon Web Services Keyspaces

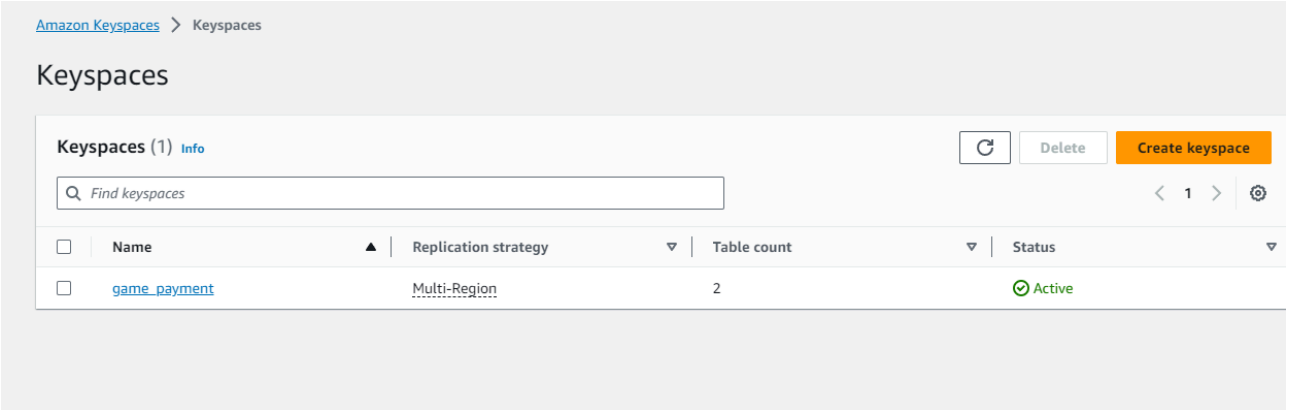


Рисунок 4.2 – Створений Keyspace (аналог database у інших SQL базах даних) у хмарному сервісі Amazon Web Services Keyspaces

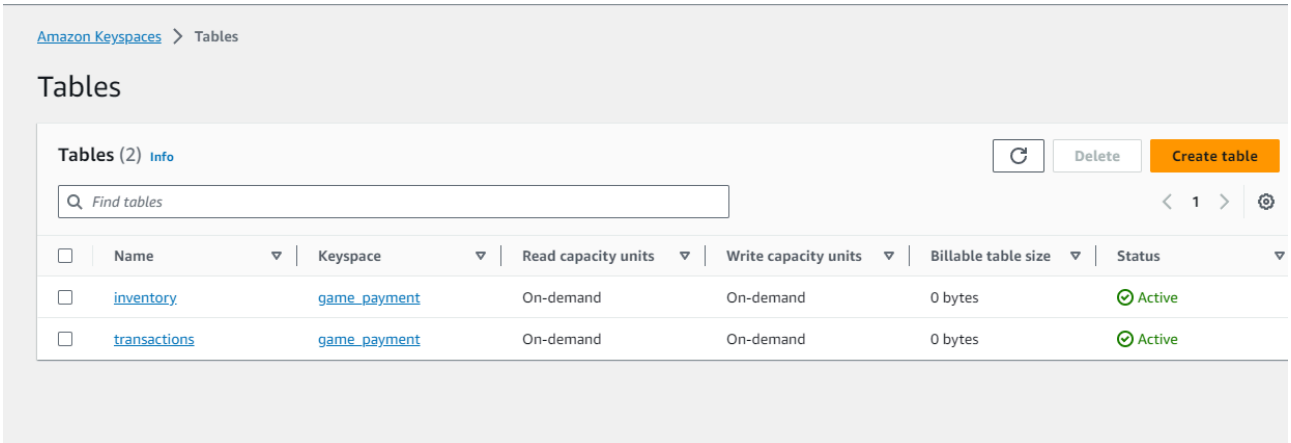


Рисунок 4.3 – Створені таблиці inventory та transactions у хмарному сервісі Amazon Web Services Keyspaces

Помилка показана на рис. 4.4 має опис “Supported consistency levels are: ONE, LOCAL_QUORUM, LOCAL_ONE”, що означає, що Amazon Web Services Keyspaces підтримує лише вірні консистенції ONE, LOCAL_QUORUM, LOCAL_ONE. Цього недостатньо для проведення експерименту з моделлю транзакції ACID на розподіленому рівні, тому будуть розглянуті інші альтернативи для розготання розподіленої бази даних Cassandra.

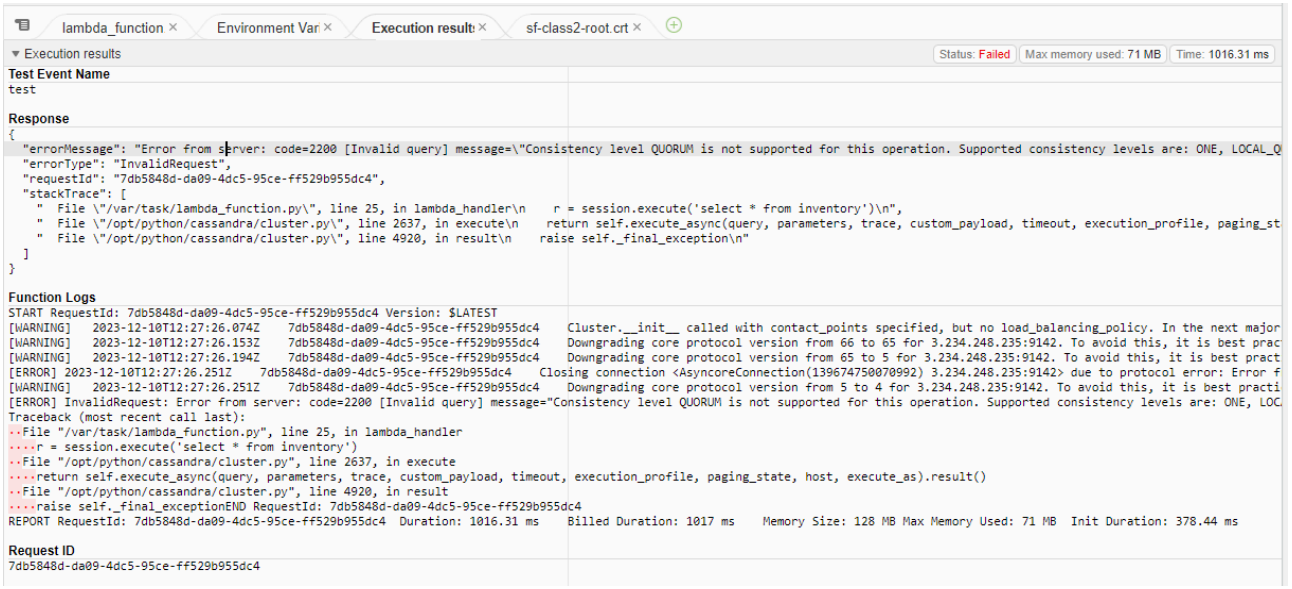


Рисунок 4.4 – Результат спроби створення зв’язку з базою даних з рівнем консистенції даних QUORUM розгорнутої у хмарному сервісі Amazon Web Services Keyspaces

Однією з альтернатив хмарного сервісу, що дозволяє розгортання бази даних Cassandra є Astra DB [22]. Ця альтернатива підтримує рівень консистенції QUORUM, необхідний для проведення експерименту.

Було створено обліковий запис у Astra DB, а також створено базу даних game_payment у двох регіонах (North America (us-east-1) та Europe (eu-central-1)) (див. рис. 4.5) аналогічно за конфігурацією до бази, створеної у AWS що показано на рисунках 4.1, 4.2.

Зв’язок з базою даних Astra DB було налаштовано у коді згідно інструкції на офіційному сайті. Фрагмент коду з підрозділу 3.1, що створює таблиці для мікросервісу оплат було додано та виконано без помилок з налаштуванням консистенції даних на рівні QUORUM (див. рис. 4.6, 4.7). Це означає, що рівень консистенції QUORUM дійсно підтримується хмарною розподіленою базою даних Astra DB, та її можна налаштувати на модель транзакції ACID на розподіленому рівні, що необхідно для проведення експерименту.

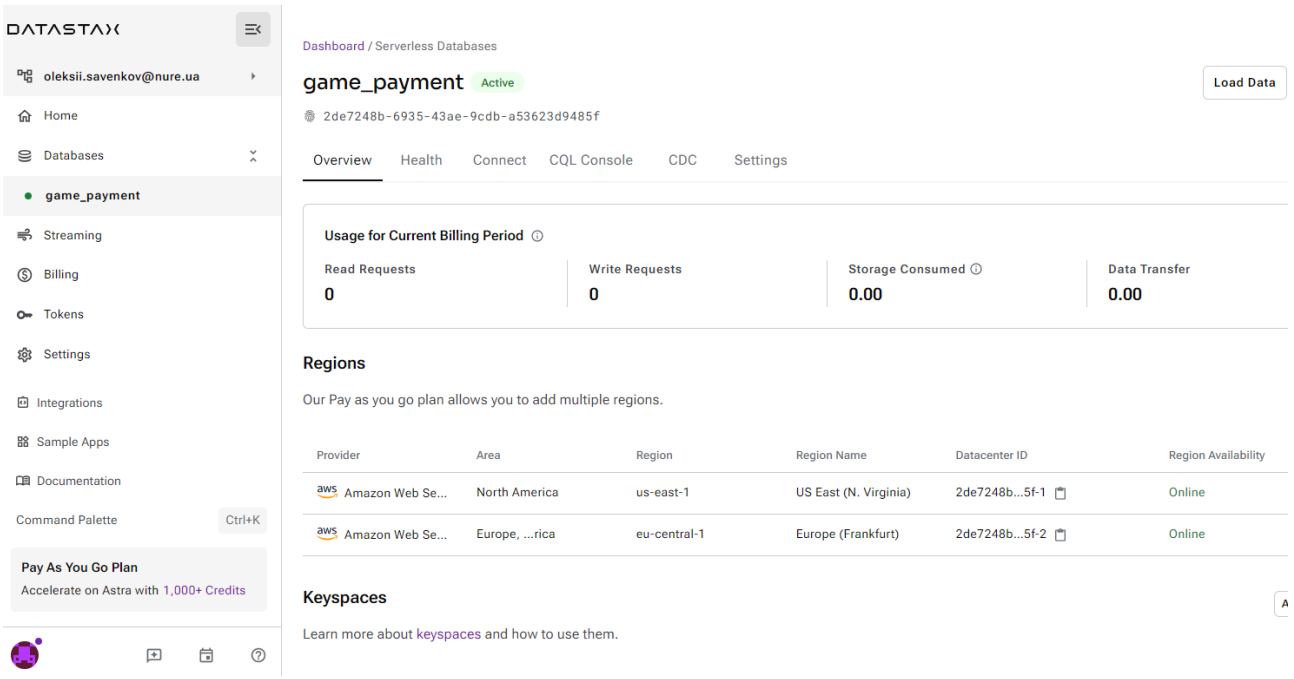


Рисунок 4.5 – База даних Apache Cassandra створена у хмарному сервісі Datastax (Astra DB), розгорнута у двох регіонах (Північній Америці та Європі)

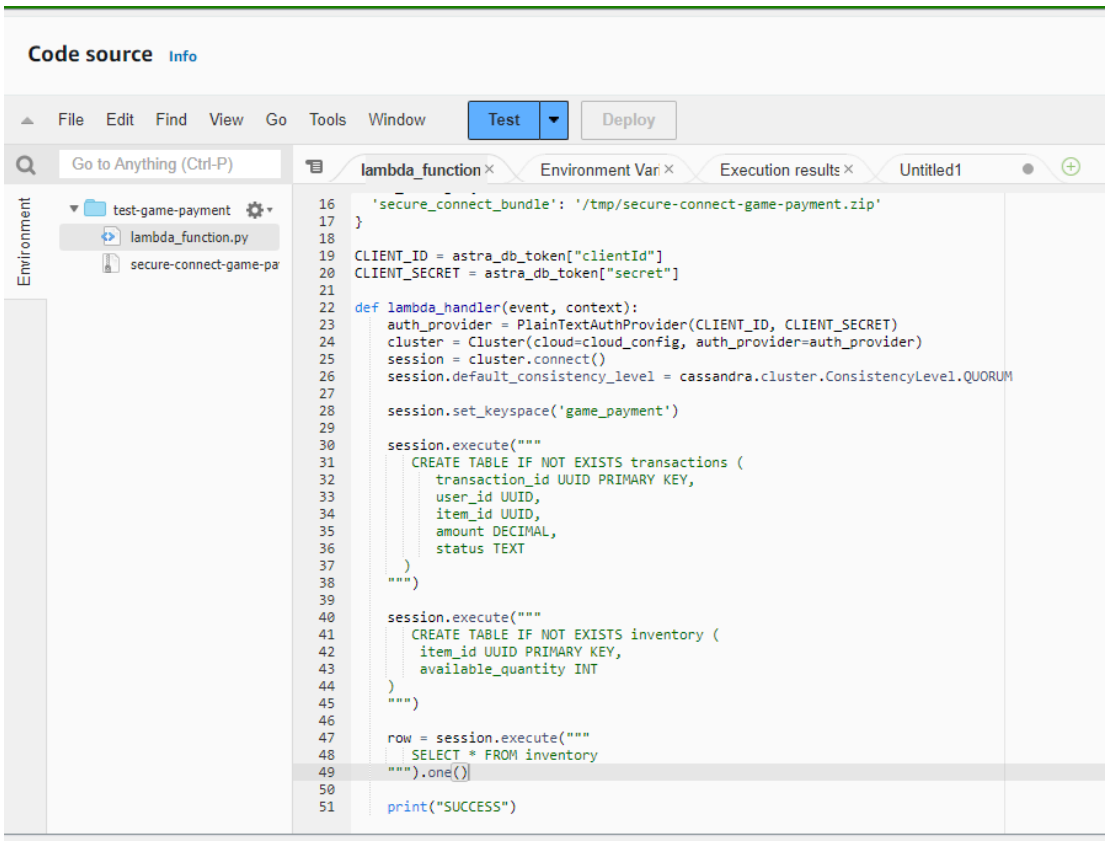


Рисунок 4.6 – Фрагмент коду, що налаштовує зв’язок з базою даних Apache Cassandra (хмарною реалізацією Astra DB) з рівнем консистенції даних QUORUM та створює таблиці для мікросервісу оплат онлайн-гри

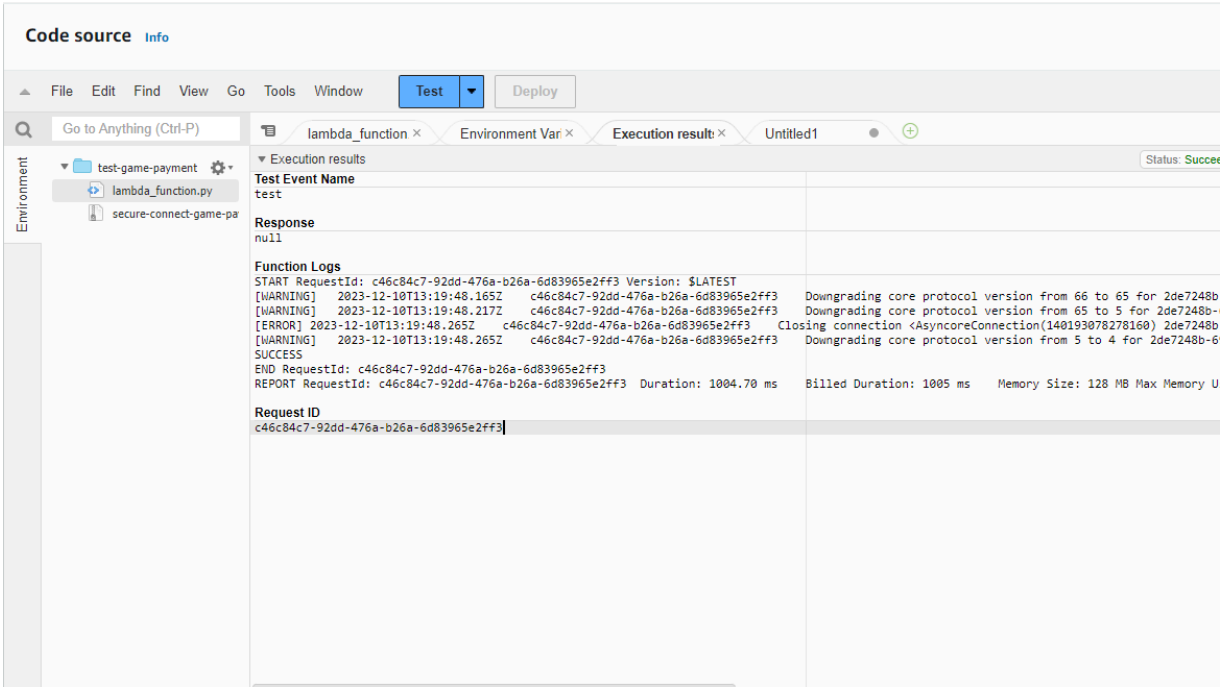


Рисунок 4.7 – Результат успішного виконання коду з попереднього рисунку 4.6

Тепер необхідно провести експеримент, що оцінює затримку консистенції даних та швидкодію мікросервісу оплат з використанням налаштованого зв’язку із базою.

Експеримент буде реалізований за допомогою двох функцій Amazon Web Services Lambda. Перша функція буде оновлювати дані у таблиці з одного регіону. Друга функція буде читати дані з таблиці з другого регіону та рахувати затримку консистенції даних. Обидві функції також будуть рахувати час запитів до бази даних для подальшого його порівняння з іншої моделлю транзакцій.

Повний код функції, що оновлює дані у таблиці inventory наведений у додатку Б.

Перша функція, що оновлює дані, після виконання виводить MIN UPDATE TIME (мінімальний час оновлення рядку в таблиці), AVG UPDATE TIME (середній арифметичний час оновлення рядку в таблиці), MAX UPDATE TIME (максимальний час оновлення рядку в таблиці), update_timestamps (масив моментів часу у форматі epoch, коли дані були оновлені), та цей самий масив але списком, щоб можна було легко вставити в аркуш excel та порівняти з

масивом моментів часу з другої функції.

Повний код другої функції AWS Lambda наведений у додатку Б.

Друга функція, що зчитує дані з іншого регіону, після виконання виводить MIN READ TIME (мінімальний час читання рядку з таблиці), AVG READ TIME (середній арифметичний час читання рядку з таблиці), MAX READ TIME (максимальний час читання рядку з таблиці), read_timestamps (масив останніх моментів часу у форматі epoch, коли для відповідного рядка читання з бази даних повернуло попередній результат до оновлення. Якщо порівнювати відповідні елемети масиву read_timestamps та update_timestamps, то щоб була строга консистентність даних, портібно щоб моменти часу у update_timestamps були більш пізніми, ніж відповідні елементи у масиві read_timestamps. Якщо це буде навпки, то це свідчиме про затримку консистентності даних, та можливість невірного оновлення даних (що в контексті мікросервісу оплат у онлайн-грі може привести до продажу більшої кількості предметів, ніж є в наявності)), та цей самий масив але списком, щоб можна було легко вставити в аркуш excel та порівняти з масивом моментів часу з першої функції.

Обидві функції тестують фрагмент функціоналу мікросервісу оплат онлайн-гри, що відповідає за перевірку наявності предмету в магазині та зменшення кількості предметів у наявності після покупки. Функції тестують 5 предметів в магазині, кожен предмет спочатку має наявність 5 в магазині, та впродовж тестування кількість зменшується до 0.

Після запуску обох функцій одночасно, вони вивели наступні результати (див. рис. 4.8, 4.9).

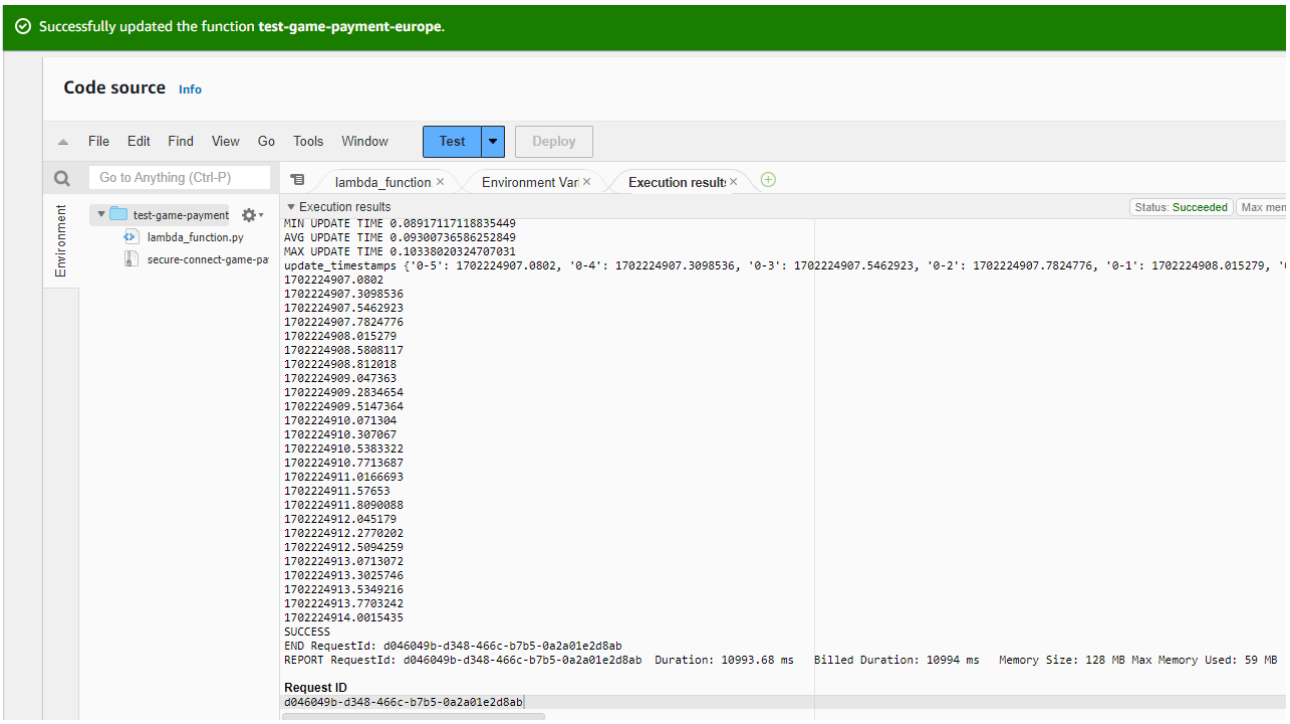


Рисунок 4.8 – Результат виконання AWS Lambda-функції test-game-payment-europe, що оновлює дані з європейського регіону, та виводить статистику по часу оновлення, та масив моментів часу у форматі epoch, коли відповідне оновлення було завершене

Мінімальний час оновлення рядку серед 25 тестів був 89 мілісекунд. Максимальний – 103 мілісекунди. Середнє арифметичне – 93 мілісекунди.

Також функція вивела масив моментів часу у форматі epoch, коли відповідне оновлення було завершене, для порівняння з результатом другої функції для оцінки затримки консистентності.

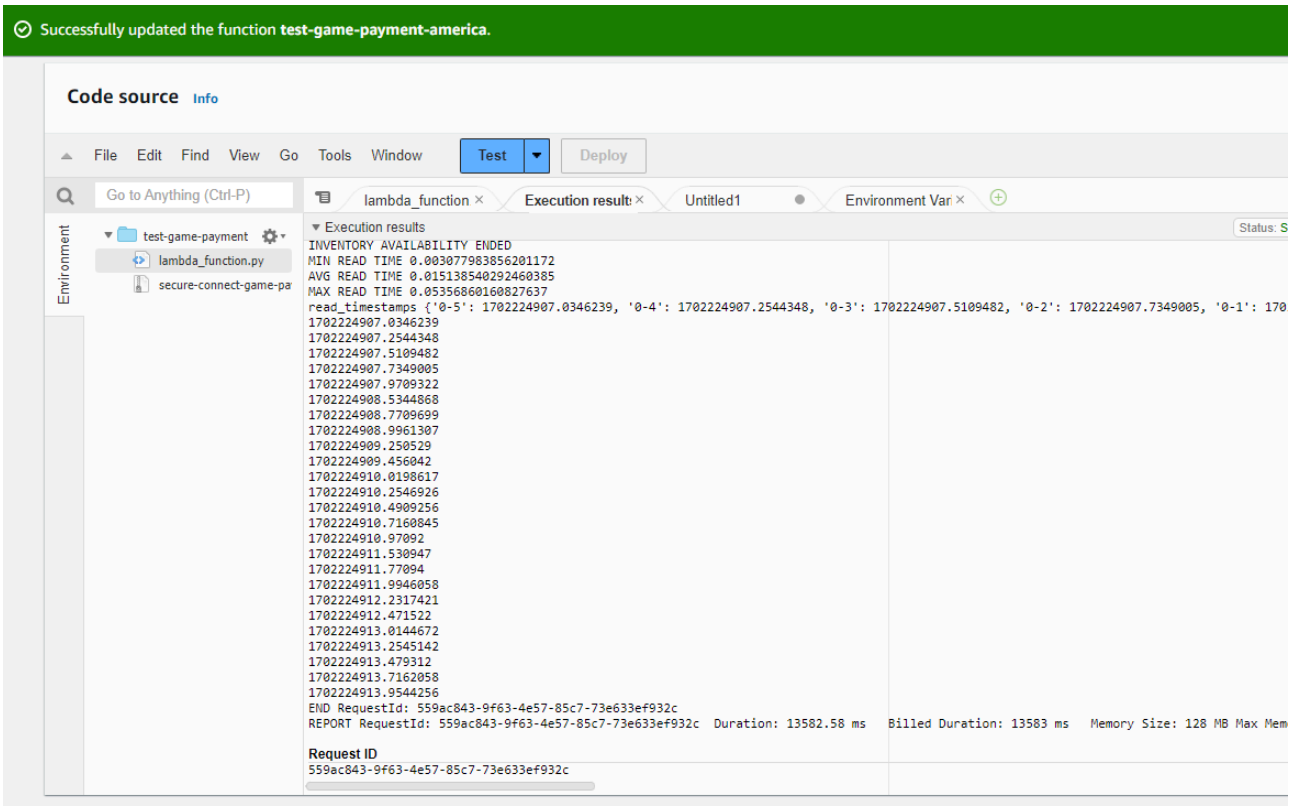
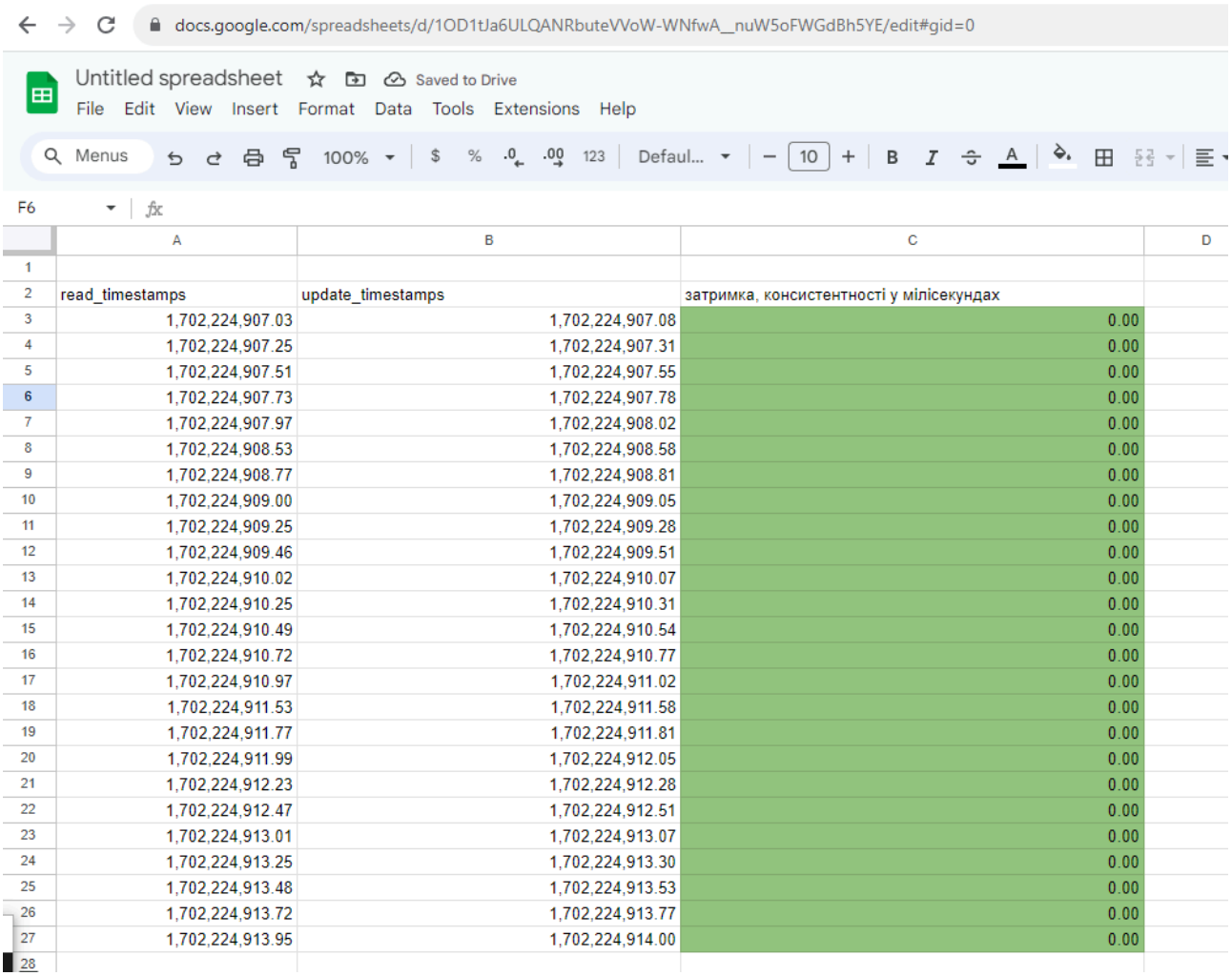


Рисунок 4.9 – Результат виконання AWS Lambda-функції test-game-payment-america, що зчитує дані з північноамериканського регіону, та виводить статистику по часу читання, та масив моментів часу у форматі epoch, коли відповідне читання повернуло попереднє значення

Мінімальний час читання рядку серед усіх тестів був 3 мілісекунди. Максимальний – 53 мілісекунди. Середнє арифметичне – 15 мілісекунд.

Занесемо виведені масиви моментів часу read_timestamps та update_timestamps у аркуш excel для оцінки затримки консистентності даних за допомогою формул Excel (див. рис. 4.10).



	A	B	C	D
1				
2	read_timestamps	update_timestamps	затримка, консистентності у мілісекундах	
3	1,702,224,907.03	1,702,224,907.08	0.00	
4	1,702,224,907.25	1,702,224,907.31	0.00	
5	1,702,224,907.51	1,702,224,907.55	0.00	
6	1,702,224,907.73	1,702,224,907.78	0.00	
7	1,702,224,907.97	1,702,224,908.02	0.00	
8	1,702,224,908.53	1,702,224,908.58	0.00	
9	1,702,224,908.77	1,702,224,908.81	0.00	
10	1,702,224,909.00	1,702,224,909.05	0.00	
11	1,702,224,909.25	1,702,224,909.28	0.00	
12	1,702,224,909.46	1,702,224,909.51	0.00	
13	1,702,224,910.02	1,702,224,910.07	0.00	
14	1,702,224,910.25	1,702,224,910.31	0.00	
15	1,702,224,910.49	1,702,224,910.54	0.00	
16	1,702,224,910.72	1,702,224,910.77	0.00	
17	1,702,224,910.97	1,702,224,911.02	0.00	
18	1,702,224,911.53	1,702,224,911.58	0.00	
19	1,702,224,911.77	1,702,224,911.81	0.00	
20	1,702,224,911.99	1,702,224,912.05	0.00	
21	1,702,224,912.23	1,702,224,912.28	0.00	
22	1,702,224,912.47	1,702,224,912.51	0.00	
23	1,702,224,913.01	1,702,224,913.07	0.00	
24	1,702,224,913.25	1,702,224,913.30	0.00	
25	1,702,224,913.48	1,702,224,913.53	0.00	
26	1,702,224,913.72	1,702,224,913.77	0.00	
27	1,702,224,913.95	1,702,224,914.00	0.00	
28				

Рисунок 4.10 – Результати виведення обох функцій, занесені у аркуш Excel у перші дві колонки, та обчислена затримка консистенції між записом та читанням у третій колонці

Результати обчислення показують що затримка консистенції даних у всіх тестах відсутня.

4.3. Тестування моделі транзакцій BASE

Перед початком тестування, потрібно розгорнути розподілену базу даних Amazon DynamoDB у двох регіонах.

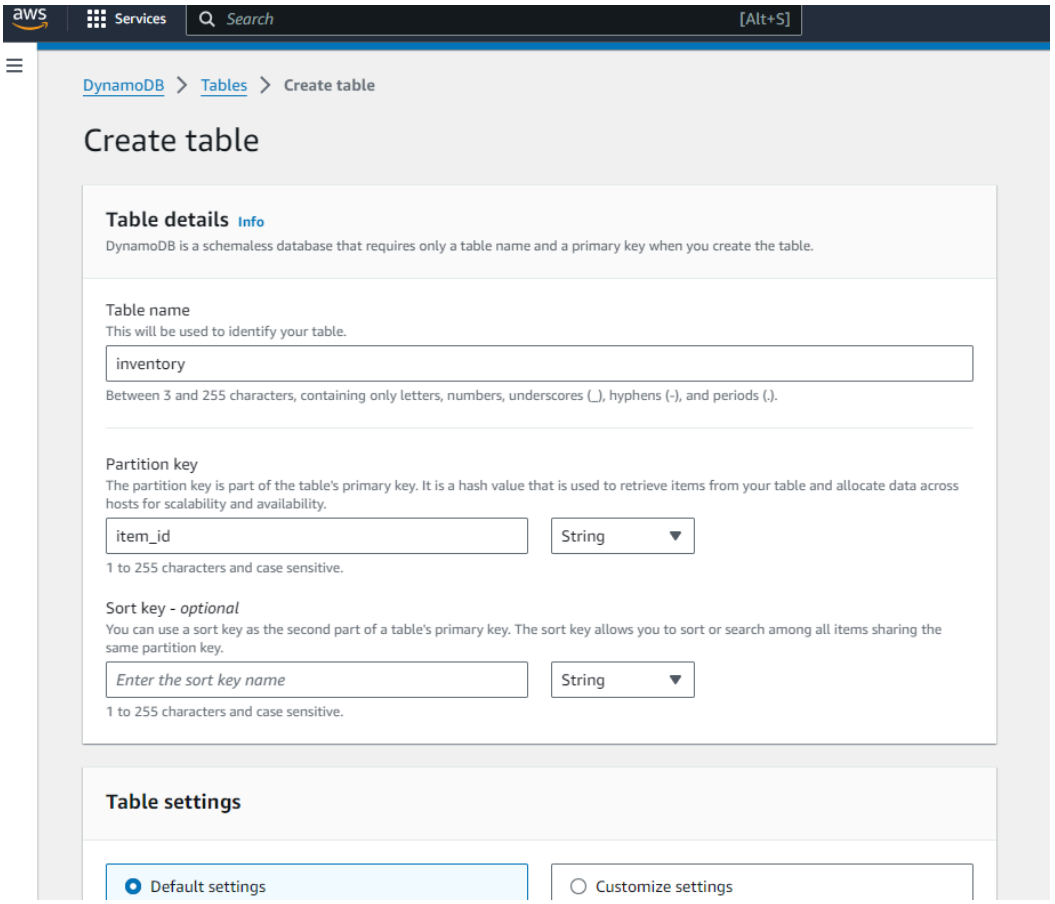


Рисунок 4.11 – Створення таблиці inventory в хмарній базі даних DynamoDB з partition key item_id в регіоні Північна Америка

За замовчування таблиці бази даних DynamoDB розгортаються лише в одному регіоні [23]. Було додано репліку таблиці в Європейському регіоні (рис. 4.12)

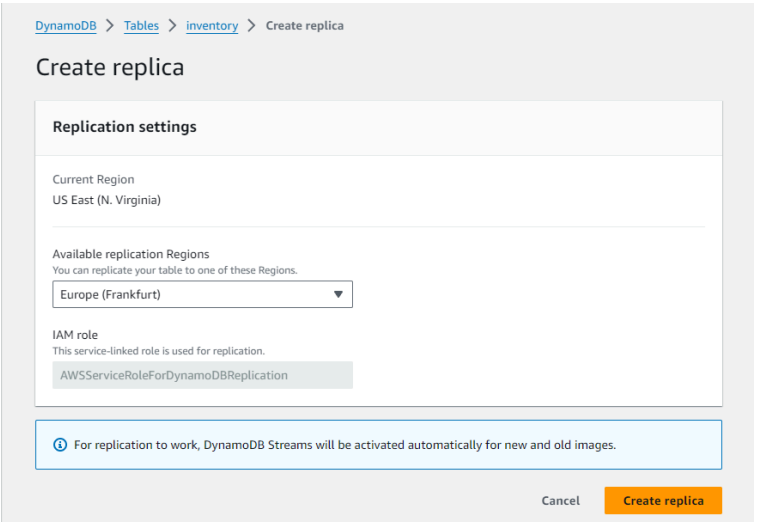


Рисунок 4.12 – Створення репліки таблиці inventory в регіоні Frankfurt

Повний код функції, що оновлює дані у таблиці inventory в DynamoDB, наведений у додатку Б.

Перша функція, що оновлює дані, після виконання виводить MIN UPDATE TIME (мінімальний час оновлення рядку в таблиці), AVG UPDATE TIME (середній арифметичний час оновлення рядку в таблиці), MAX UPDATE TIME (максимальний час оновлення рядку в таблиці), update_timestamps (масив моментів часу у форматі epoch, коли дані були оновлені), та цей самий масив але списком, щоб можна було легко вставити в аркуш excel та порівняти з масивом моментів часу з другої функції.

Повний код другої функція AWS Lambda наведений у додатку Б.

Друга функція, що зчитує дані з іншого регіону, після виконання виводить MIN READ TIME (мінімальний час читання рядку з таблиці), AVG READ TIME (середній арифметичний час читання рядку з таблиці), MAX READ TIME (максимальний час читання рядку з таблиці), read_timestamps (масив останніх моментів часу у форматі epoch, коли для відповідного рядка читання з бази даних повернуло попередній результат до оновлення. Якщо порівнювати відповідні елемети масиву read_timestamps та update_timestamps, то щоб була строга консистентність даних, портібно щоб моменти часу у update_timestamps були більш пізніми, ніж відповідні елементи у масиві read_timestamps. Якщо це буде навпки, то це свідчиме про затримку консистентності даних, та можливість невірного оновлення даних (що в контексті мікросервісу оплат у онлайн-грі може привести до продажу більшої кількості предметів, ніж є в наявності)), та цей самий масив але списком, щоб можна було легко вставити в аркуш excel та порівняти з масивом моментів часу з першої функції.

Обидві функції тестують фрагмент функціоналу мікросервісу оплат онлайн-гри, що відповідає за перевірку наявності предмету в магазині та зменшення кількості предметів у наявності після покупки. Функції тестують 5 предметів в магазині, кожен предмет спочатку має наявність 5 в магазині, та впродовж тестування кількість зменшується до 0.

Після запуску обох функцій одночасно, вони вивели наступні результати

(див. рис. 4.13, 4.14).

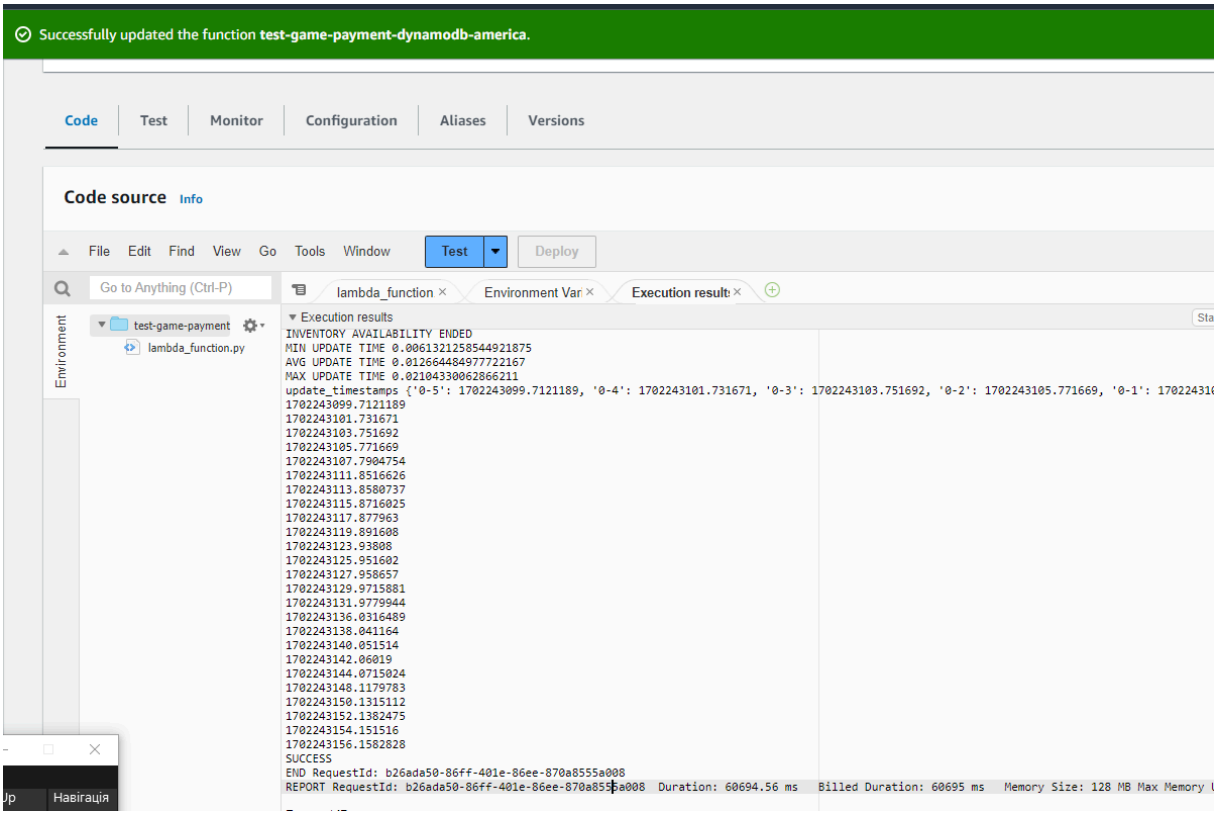


Рисунок 4.13 – Результат виконання AWS Lambda-функції test-game-payment-dynamodb-america, що оновлює дані з американського регіону, та виводить статистику по часу оновлення, та масив моментів часу у форматі epoch, коли відповідне оновлення було завершене

Мінімальний час оновлення рядку серед 25 тестів був 6 мілісекунд. Максимальний – 21 мілісекунда. Середнє арифметичне – 12 мілісекунд.

Також функція вивела масив моментів часу у форматі epoch, коли відповідне оновлення було завершене, для порівняння з результатом другої функції для оцінки затримки консистентності.

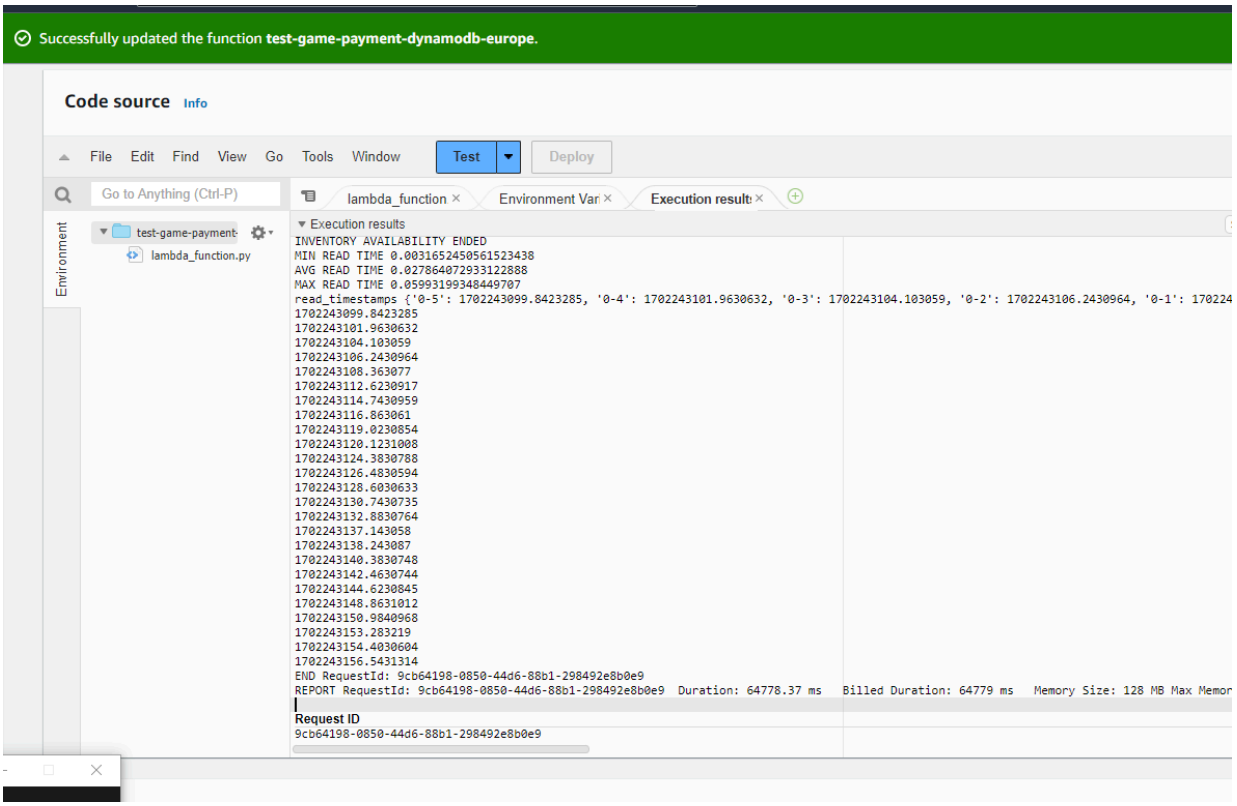
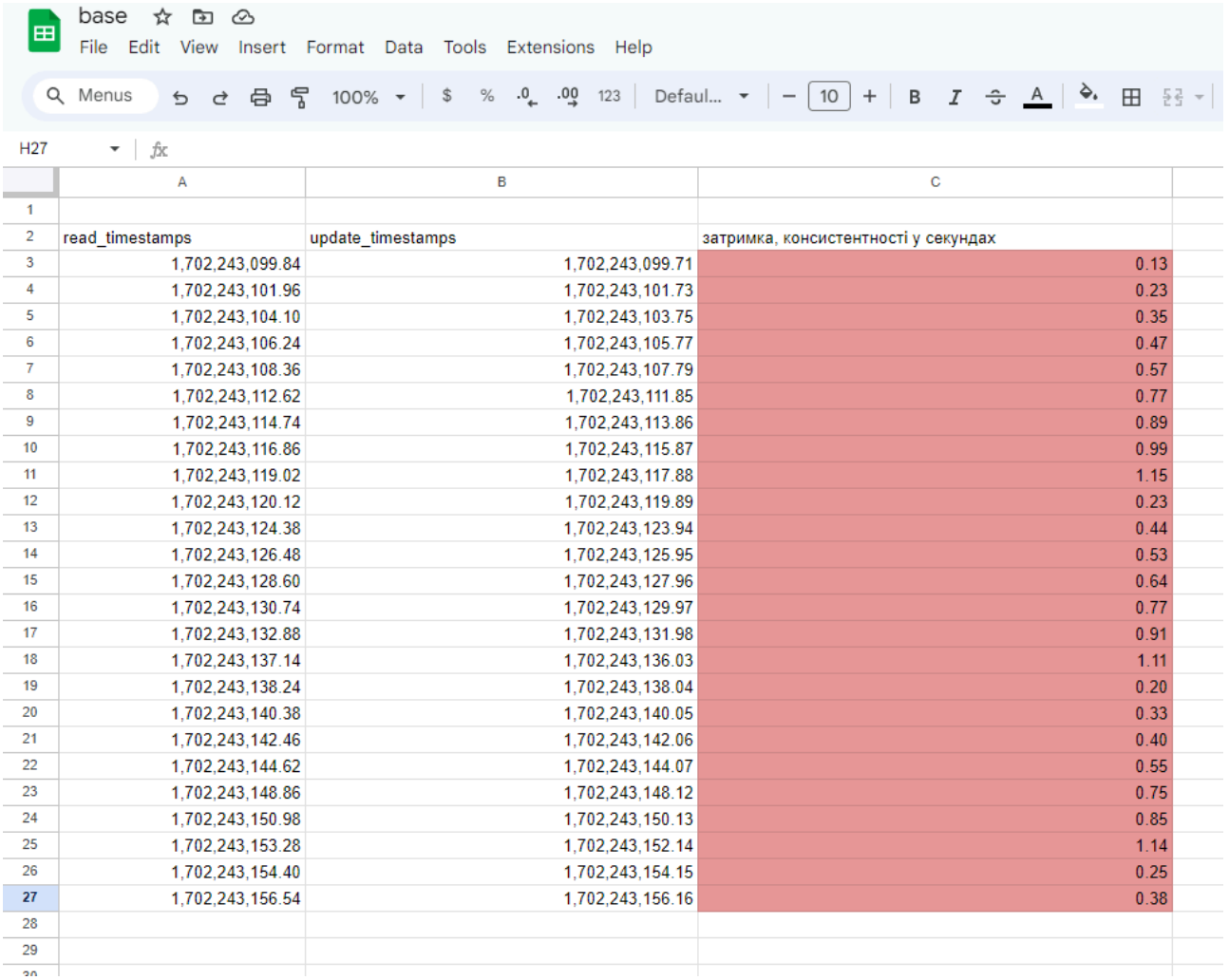


Рисунок 4.14 – Результат виконання AWS Lambda-функції test-game-payment-dynamodb-europe, що зчитує дані з європейського регіону, та виводить статистику по часу читання, та масив моментів часу у форматі epoch, коли відповідне читання повернуло попереднє значення

Мінімальний час читання рядку серед усіх тестів був 3 мілісекунди. Максимальний – 59 мілісекунди. Середнє арифметичне – 27 мілісекунд.

Занесемо виведені масиви моментів часу read_timestamps та update_timestamps у аркуш excel для оцінки затримки консистентності даних за допомогою формул Excel (див. рис. 4.15).



base

File Edit View Insert Format Data Tools Extensions Help

H27

	A	B	C
1			
2	read_timestamps	update_timestamps	затримка, консистентності у секундах
3	1,702,243,099.84	1,702,243,099.71	0.13
4	1,702,243,101.96	1,702,243,101.73	0.23
5	1,702,243,104.10	1,702,243,103.75	0.35
6	1,702,243,106.24	1,702,243,105.77	0.47
7	1,702,243,108.36	1,702,243,107.79	0.57
8	1,702,243,112.62	1,702,243,111.85	0.77
9	1,702,243,114.74	1,702,243,113.86	0.89
10	1,702,243,116.86	1,702,243,115.87	0.99
11	1,702,243,119.02	1,702,243,117.88	1.15
12	1,702,243,120.12	1,702,243,119.89	0.23
13	1,702,243,124.38	1,702,243,123.94	0.44
14	1,702,243,126.48	1,702,243,125.95	0.53
15	1,702,243,128.60	1,702,243,127.96	0.64
16	1,702,243,130.74	1,702,243,129.97	0.77
17	1,702,243,132.88	1,702,243,131.98	0.91
18	1,702,243,137.14	1,702,243,136.03	1.11
19	1,702,243,138.24	1,702,243,138.04	0.20
20	1,702,243,140.38	1,702,243,140.05	0.33
21	1,702,243,142.46	1,702,243,142.06	0.40
22	1,702,243,144.62	1,702,243,144.07	0.55
23	1,702,243,148.86	1,702,243,148.12	0.75
24	1,702,243,150.98	1,702,243,150.13	0.85
25	1,702,243,153.28	1,702,243,152.14	1.14
26	1,702,243,154.40	1,702,243,154.15	0.25
27	1,702,243,156.54	1,702,243,156.16	0.38
28			
29			

Рисунок 4.15 – Результати виведення обох функцій, занесені у аркуш Excel у перші дві колонки, та обчислена затримка консистенції між записом та читанням у третій колонці

Результати обчислення показують що затримка консистенції даних у моделі транзакцій BASE на прикладі DynamoDB може сягати більше 1 секунди, а в середньому дорівнює приблизно півсекунди.

4.4. Висновки тестування швидкодії і консистентності даних різних моделей транзакцій

Візуалізуємо результати дослідження з попередніх підрозділів за допомогою графіків (див. рис. 4.16, 4.17, 4.18).

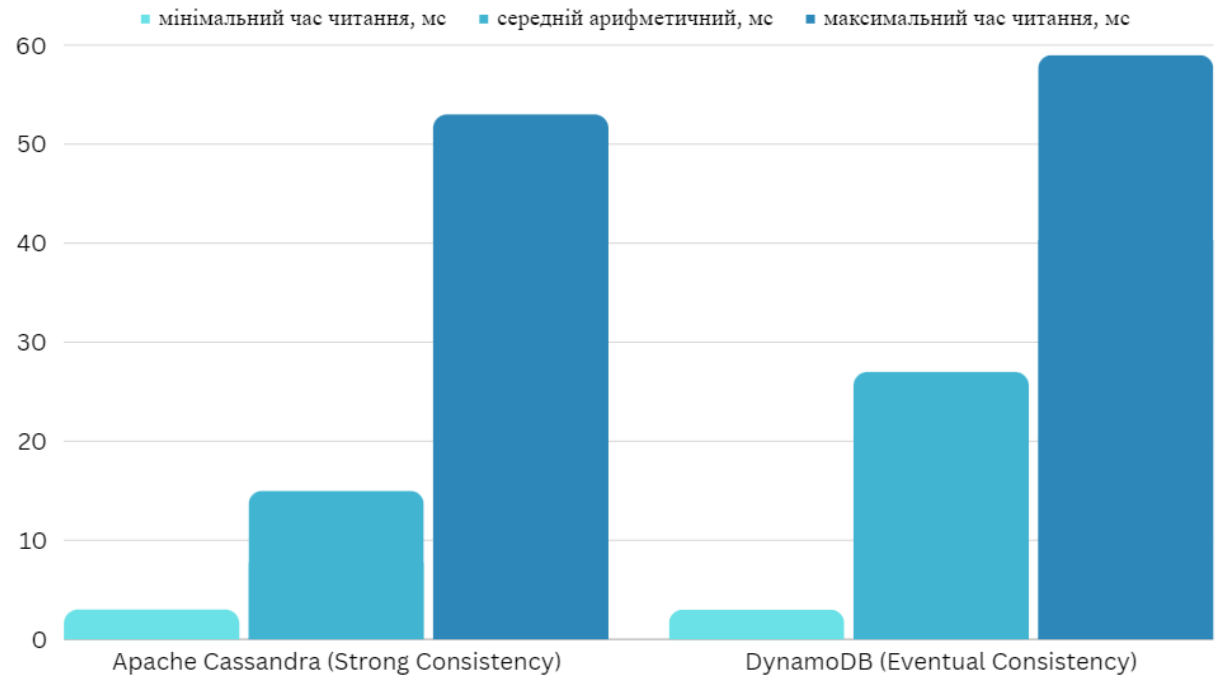


Рисунок 4.16 – Графік порівняння мінімального, середнього арифметичного та максимального часу читання рядка з бази даних Apache Cassandra (ACID на розподіленому рівні) та DynamoDB (BASE)

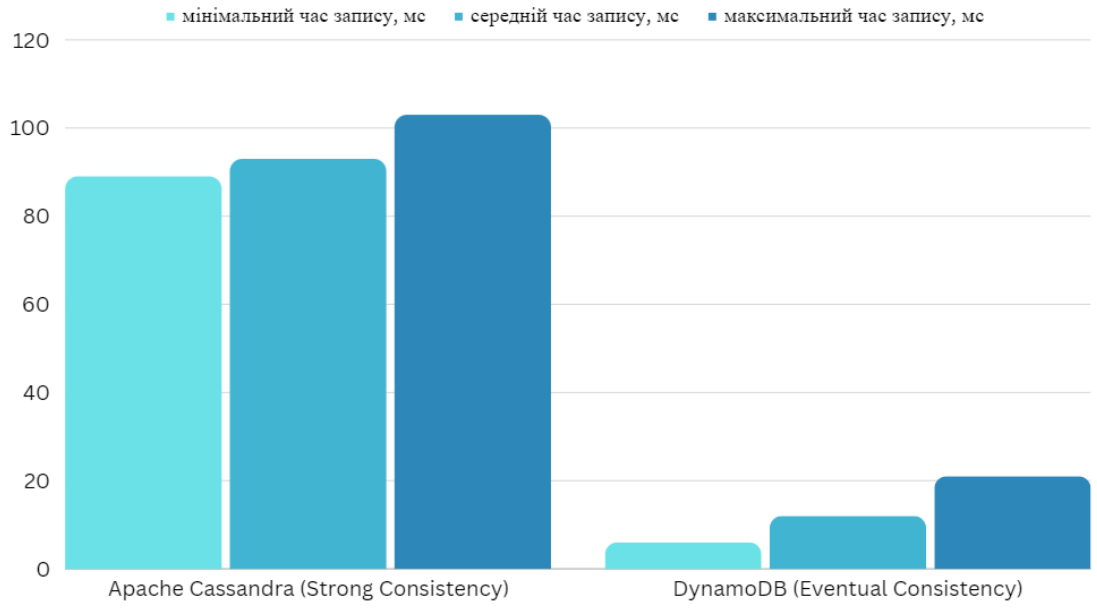


Рисунок 4.17 – Графік порівняння мінімального, середнього арифметичного та максимального часу запису рядка в базу даних Apache Cassandra (ACID на розподіленому рівні) та DynamoDB (BASE)

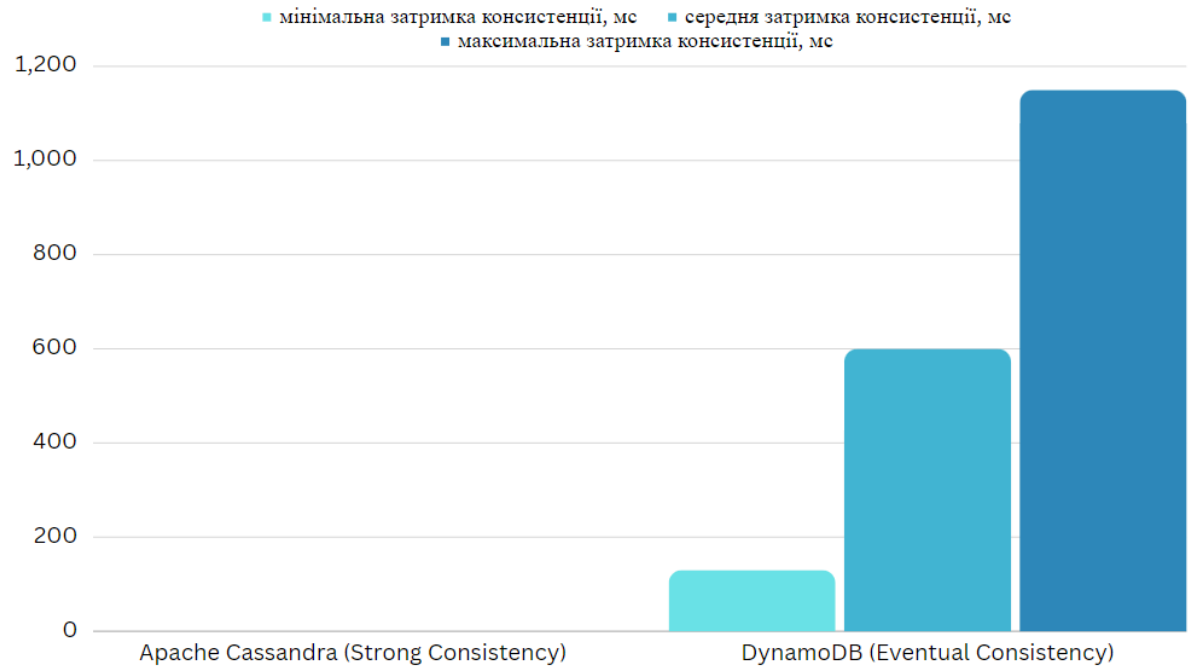


Рисунок 4.18 – Графік порівняння мінімального, середнього арифметичного та максимального часу затримки консистенції в базі даних Apache Cassandra (ACID на розподіленому рівні) та DynamoDB (BASE)

Час читання рядка з бази даних є приблизно однаковим в розподіленій DynamoDB та Apache Cassandra з налаштуванням консистенції QUORUM для читання та запису. Час запису рядку є значно повільнішим в Apache Cassandra ніж в DynamoDB. Затримка консистенції повністю відсутня в Cassandra на відміну від DynamoDB.

Результати проведеного практичного експерименту підтверджують те, що модель транзакцій ACID завжди підтримує розподілену систему в консистентному стані, а в моделі транзакцій BASE можлива затримка пропагування змін між вузлами розподіленої система, та певний час система може знаходитися не в консистентному стані. Це підтверджує рекомендацію, наведену у розділі 2 про те, що якщо консистентність є більш важливою ніж швидкодія, тоді варто обрати модель транзакцій ACID. Якщо постійна консистентність даних не є дуже важливою, і швидкодія є більш важливою, тоді варто обрати модель транзакцій BASE.

Результати експерименту також підтверджують доцільність використання обох моделей транзакцій (ACID та BASE) в одній розподіленій системі на прикладі системи онлайн-гри, що була спроектована в підрозділі 3.2. Якщо окремі компоненти розподіленої системи мають різні пріоритети щодо швидкодії і консистентності даних, тоді варто обрати змішану модель транзакцій, і використовувати і ACID і BASE в окремих компонентах.

Результати проведеного експерименту можуть бути використані архітекторами систем або програмістами при виборі моделі транзакції та бази даних для розподілених систем, які вони проектують або розробляють.

Практичний експеримент був проведений лише з розподіленими базами даних Cassandra, що при певних налаштуваннях представляє модель транзакцій ACID на рівні всієї розподіленої системи, та DynamoDB, що представляє модель транзакцій BASE. Темами для подальших досліджень можуть бути порівняння швидкодії та консистентності інших розподілених баз даних, що не були детально розглянуті у цьому дослідженні. При виборі інших баз даних результати експерименту можуть дещо відрізнятися.

ВИСНОВКИ

У ході дослідження були розглянуті різні підходи до підтримки консистенції даних у розподілених системах, а саме моделі ACID та BASE.

ACID (Atomicity, Consistency, Isolation, Durability) і BASE (Basically Available, Soft state, Eventually consistent) - це дві основні філософії в підтримці консистентності даних в розподілених системах. Обидва підходи створені для вирішення проблеми збереження даних в розподіленому середовищі, але підходять до неї з різних точок зору.

ACID - це традиційний підхід до забезпечення консистентності даних, який встановлює високі вимоги до транзакцій, у тому числі до консистентності даних. BASE, навпаки, виходить з ідеї, що абсолютної консистентності може бути важко досягнути у деяких випадках і, можливо, навіть не потрібно її досягати.

Якщо проект вимагає строгого керування транзакціями та потрібна гарантія високої цілісності даних, то рекомендовано вибирати ACID підхід. У випадках, коли доступність та швидкість є більш важливими, BASE є більш підходящим варіантом.

Якщо у кожній підсистемі розподіленої системи вимоги відрізняються - у деяких критично важлива швидкодія, а в інших - консистентність, тоді має сенс обрати різні розподілені сховища даних для окремих підсистем, та використовувати різні моделі транзакцій в мікросервісах (або інших компонентах розподіленої системи). У даному дослідженні продемонстровано використання моделей ACID та BASE одночасно на прикладі проектування масштабної онлайн-гри з розподіленою архітектурою бекенду, що включає в себе можливість гравців купляти віртуальні предмети за справжні гроші.

Проведено практичний експеримент, що досліджує швидкодію та затримку консистентності даних в моделях транзакцій ACID та BASE на прикладі розподілених баз даних Apache Cassandra та DynamoDB.

Експеримент сам по собі є новим, і він підтверджує теоретичну інформацію наведену в джерелах.

Досліджено доцільність використання обох моделей транзакцій одночасно в одній розподіленій системі на прикладі проектування онлайн-гри, у яку можуть одночасно грати гравці з різних регіонів світу, та створено рекомендації щодо використання обох моделей транзакцій в одній мікросервісній системі.

Сформовано рекомендації щодо вибору моно-моделі транзакцій (ACID або BASE) або змішаної моделі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Chesson G. L. The network Unix system. ACM SIGOPS Operating Systems Review. 1975. Vol. 9, no. 5. P. 60–66. URL: https://doi.org/10.1145/1067629.806522 (date of access: 28.10.2023).
2. Banothu N., Bhukya S., Sharma K. V. Big-data: Acid versus base for database transactions. 2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT), Chennai, India, 3–5 March 2016. 2016. URL: https://doi.org/10.1109/iceeot.2016.7755401 (date of access: 28.10.2023).
3. Little M. Transactions and Web services. Communications of the ACM. 2003. Vol. 46, no. 10. P. 49–54. URL: https://doi.org/10.1145/944217.944237 (date of access: 28.10.2023).
4. Analysis and Comparison of Atomic Commit Protocols for Adaptive Usage in Wireless Sensor Networks / C. Reinke et al. 2010 IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing, Newport Beach, CA, USA, 7–9 June 2010. 2010. URL: https://doi.org/10.1109/sutc.2010.12 (date of access: 28.10.2023).
5. Pritchett D. BASE: An Acid Alternative. Queue. 2008. Vol. 6, no. 3. P. 48–55. URL: https://doi.org/10.1145/1394127.1394128 (date of access: 28.10.2023).
6. Štefanko M., Chaloupka O., Rossi B. The Saga Pattern in a Reactive Microservices Environment. 14th International Conference on Software Technologies, Prague, Czech Republic, 26–28 July 2019. 2019. URL: https://doi.org/10.5220/0007918704830490 (date of access: 28.10.2023).
7. Савенков О.А. Дослідження методів підтримки консистенції у розподілених транзакціях. Експериментальні та теоретичні дослідження в контексті сучасної науки : матеріали IV Всеукраїнської студентської наукової конференції, м. Чернігів, 29 вересня, 2023 рік / ГО «Молодіжна наукова ліга». — Вінниця : ТОВ «УКРЛОГОС Груп», 2023. С 63-65.
8. Zhao X., Haller P. Consistency types for replicated data in a higher-order

distributed programming language. The Art, Science, and Engineering of Programming. 2020. Vol. 5, no. 2. URL: https://doi.org/10.22152/programming-journal.org/2021/5/6 (date of access: 22.12.2023).

9. van Steen M., Tanenbaum A. S. A brief introduction to distributed systems. Computing. 2016. Vol. 98, no. 10. P. 967–1009. URL: https://doi.org/10.1007/s00607-016-0508-7 (date of access: 22.12.2023).
10. Apostu A., Rednic E., Puican F. Modeling Cloud Architecture in Banking Systems. Procedia Economics and Finance. 2012. Vol. 3. P. 543–548. URL: https://doi.org/10.1016/s2212-5671(12)00193-1 (date of access: 22.12.2023).
11. Otto S., Kirn S. Adaption in distributed systems. the 8th annual conference, Seattle, Washington, USA, 8–12 July 2006. New York, New York, USA, 2006. URL: https://doi.org/10.1145/1143997.1144031 (date of access: 22.12.2023).
12. The role of Artificial Intelligence and distributed computing in IoT applications / ed. by S. Rodríguez González et al. Ediciones Universidad de Salamanca, 2020. URL: https://doi.org/10.14201/0aq0287 (date of access: 22.12.2023).
13. Vertical/Horizontal Resource Scaling Mechanism for Federated Clouds / C.-Y. Liu et al. 2014 International Conference on Information Science and Applications (ICISA), Seoul, South Korea, 6–9 May 2014. 2014. URL: https://doi.org/10.1109/icisa.2014.6847479 (date of access: 22.12.2023).
14. Thones J. Microservices. IEEE Software. 2015. Vol. 32, no. 1. P. 116. URL: https://doi.org/10.1109/ms.2015.11 (date of access: 22.12.2023).
15. Kuhlenkamp J., Klems M., Röss O. Benchmarking scalability and elasticity of distributed database systems. Proceedings of the VLDB Endowment. 2014. Vol. 7, no. 12. P. 1219–1230. URL: https://doi.org/10.14778/2732977.2732995 (date of access: 22.12.2023).
16. Kuhlenkamp J., Klems M., Röss O. Benchmarking scalability and elasticity of distributed database systems. Proceedings of the VLDB Endowment. 2014.

Vol. 7, no. 12. P. 1219–1230. URL: https://doi.org/10.14778/2732977.2732995 (date of access: 22.12.2023).

17. Big Data Processing Technologies in Distributed Information Systems / N. Shakhovska et al. Procedia Computer Science. 2019. Vol. 160. P. 561–566. URL: https://doi.org/10.1016/j.procs.2019.11.047 (date of access: 22.12.2023).
18. Levy E., Silberschatz A. Distributed file systems: concepts and examples. ACM Computing Surveys. 1990. Vol. 22, no. 4. P. 321–374. URL: https://doi.org/10.1145/98163.98169 (date of access: 22.12.2023).
19. Nguyen N. D., Kim T. Balanced Leader Distribution Algorithm in Kubernetes Clusters. Sensors. 2021. Vol. 21, no. 3. P. 869. URL: https://doi.org/10.3390/s21030869 (date of access: 22.12.2023).
20. On construction of a distributed data storage system in cloud / C.-T. Yang et al. Computing. 2014. Vol. 98, no. 1-2. P. 93–118. URL: https://doi.org/10.1007/s00607-014-0399-4 (date of access: 22.12.2023).
21. Microservices: architecture, container, and challenges / G. Liu et al. 2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C), Macau, China, 11–14 December 2020. 2020. URL: https://doi.org/10.1109/qrs-c51114.2020.00107 (date of access: 22.12.2023).
22. Transactions and consistency in distributed database systems / I. L. Traiger et al. ACM Transactions on Database Systems. 1982. Vol. 7, no. 3. P. 323–342. URL: https://doi.org/10.1145/319732.319734 (date of access: 22.12.2023).
23. Astra DB | DataStax. DataStax. URL: https://www.datastax.com/products/datastax-astra (date of access: 22.12.2023).
24. Sivasubramanian S. Amazon dynamoDB. the 2012 international conference, Scottsdale, Arizona, USA, 20–24 May 2012. New York, New York, USA, 2012. URL: https://doi.org/10.1145/2213836.2213945 (date of access: 22.12.2023).

25. Redis Multi-Region Cluster. URL: https://www.virtuozzo.com/application-platform-docs/redis-multi-region-cluster/ (дата звернення: 12.11.2023)
26. Redis Scaling. URL: https://redis.io/docs/management/scaling/ (дата звернення: 12.11.2023)