Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження методів обробки розподілених транзакцій
в мікросервісній архітектурі на основі патерну SAGA
(тема)

Виконав:
студент (ка) 2 курсу, групи ІПЗм-22-2

Кузнецов Р.О.
(прізвище, ініціали)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-наукова

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

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

2024 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук
Кафедра програмної інженерії
Рівень вищої освіти другий (магістерський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми освітньо-наукова програма
Освітня програма Інженерія програмного забезпечення
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Кузнецову Роману Олександровичу
(прізвище, ім’я, по батькові)

1. Тема роботи «Дослідження методів обробки розподілених транзакцій в мікросервісній архітектурі на основі патерну SAGA»

Затверджена наказом по університету від 29.03.2024р. № 250 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 21.06.2024

3. Вихідні дані до роботи електронні ресурси за обраною тематикою, вимоги до реалізації розподілених транзакції на основі використання шаблону SAGA, база даних MySQL, середовище розробки IntelliJ IDEA, технології Java, Spring Boot, Spring Cloud, Apache Kafka, Eventuate Tram/Local, VisualVM, Docker Desktop

4. Перелік питань, що потрібно опрацювати в роботі аналіз та порівняння існуючих методів обробки розподілених транзакцій, вибір підходящих патернів для дослідження, проектування логічної моделі даних для проведення експериментальних досліджень, написання програмних рішень, проведення експериментів та аналіз отриманих результатів

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної галузі та постановка задачі	23.01 – 14.02.24	*виконано*
2	Аналіз та вибір шаблонів для дослідження	15.02 – 24.02.24	*виконано*
3	Аналіз та моделювання предметної області	17.02 – 28.02.24	*виконано*
4	Планування експериментів	25.02 – 28.02.24	*виконано*
5	Програмна реалізація кожного з обраних шаблонів під предметну галузь	28.02 – 01.04.24	*виконано*
6	Експериментальні дослідження	02.04 – 20.04.24	*виконано*
7	Аналіз результатів експериментальних досліджень та розробка рекомендацій	20.04 – 23.04.24	*виконано*
8	Написання та оформлення статті та тез доповіді	23.04 – 04.05.24	*виконано*
9	Підготовка пояснювальної записки	04.05 – 31.05.24	*виконано*
10	Підготовка презентації та доповіді	01.06 – 06.06.24	*виконано*
11	Нормоконтроль	07.06 – 11.06.24	*виконано*
12	Рецензування	11.06 – 16.06.24	*виконано*
13	Занесення диплома в електронний архів	18.06.2024	*виконано*
14	Попередній захист	18.06.2024	*виконано*
15	Допуск до захисту у зав. кафедри	19.06.2024	*виконано*

Дата видачі завдання 22 січня 2024р.

Студент _____ Кузнецов Р.О.
(підпис)

Керівник роботи _____ доц. Мазурова О.О.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ/ABSTRACT

Пояснювальна записка містить: 105 с., 27 рис., 11 табл., 38 джерел, 5 додатків.

БАЗА ДАНИХ, ЕЛЕКТРОНА КОМЕРЦІЯ, МІКРОСЕРВІС, РОЗПОДІЛЕНА СИСТЕМА, ТРАНЗАКЦІЯ, ЧЕРГА ПОВІДОМЛЕНЬ, DOCKER, EVENT SOURCING, JAVA, KAFKA, MYSQL, OUTBOX, SAGA, SPRING, VISUALVM.

Об’єктом дослідження є методи обробки розподілених транзакцій в мікросервісній архітектурі.

Метою роботи є отримання рекомендацій для інженерів, які спростять та прискорять такі процеси, як впровадження механізму забезпечення транзакційності в розподілених системах на основі мікросервісної архітектури.

Методами розробки та проектування є платформа Spring, мова програмування Java, СКБД MySQL, брокер повідомлень Apache Kafka, фреймворк для реалізації SAGA Eventuate, середовище розробки IntelliJ IDEA, ПЗ для контейнеризації Docker Desktop, профайлер JVM VisualVM.

В результаті роботи було досліджено метод обробки розподілених транзакцій SAGA на основі хореографії в поєднанні з шаблонами забезпечення атомарності, надійності та консистентності Outbox та Event Sourcing, проведено аналіз та моделювання предметної області, розроблено схему бази даних, програмно реалізовано мікросервіси під кожен шаблон, проведено експериментальне дослідження та складені рекомендації щодо доречності використання підходів в різних системах.

DATABASE, E-COMMERCE, MICROSERVICE, DISTRIBUTED SYSTEM, TRANSACTION, MESSAGE BROKER, DOCKER, EVENT SOURCING, JAVA, KAFKA, MYSQL, OUTBOX, SAGA, SPRING, VISUALVM.

The object of research are methods for processing distributed transactions in a microservice architecture.

The aim of the study is to obtain recommendations for engineers that will simplify and speed up processes such as the implementation of a mechanism for ensuring transactionalism in distributed systems based on microservice architecture.

The development and design methods used are the Spring platform, Java programming language, MySQL database, Apache Kafka message broker, SAGA Eventuate framework, IntelliJ IDEA development environment, Docker Desktop containerization software, and VisualVM JVM profiler.

As a result of the work, the method of processing distributed transactions SAGA based on choreography in combination with the Outbox and Event Sourcing patterns of ensuring atomicity, reliability and consistency was investigated, the subject area was analyzed and modeled, a database scheme was developed, microservices for each pattern were programmatically implemented, an experimental study was conducted and recommendations were made on the appropriateness of using approaches in various systems.

Я, Кузнецов Роман Олександрович, студент гр. ІПЗм-22-2, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Дослідження методів обробки розподілених транзакцій в мікросервісній архітектурі на основі патерну SAGA», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

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

ЗМІСТ

ВСТУП

У сучасному світі, де цифрова трансформація охоплює все більшу сферу нашого життя, ефективне управління транзакціями стає вирішальним фактором для забезпечення стабільності та надійності інформаційних систем. Транзакції є фундаментальною одиницею обробки даних, що забезпечує узгодженість та цілісність інформації в базах даних, що, у свою чергу, є життєво важливим для оперативного функціонування бізнес-процесів.

Зі зростанням масштабів та складності програмних продуктів, архітектура мікросервісів стала надзвичайно популярною, надаючи можливість розбиття програм на менші, незалежно розгортаємі та легше керовані компоненти. Це сприяло створенню більш гнучких та масштабованих систем, які можуть швидко адаптуватися до змінних вимог бізнесу та ринку. Мікросервісна архітектура, яка все частіше стає стандартом при проектуванні нових систем, дозволяє компаніям оптимізувати свої ресурси та прискорити час виведення продуктів на ринок.

Проте, така архітектура вносить певні виклики, особливо у контексті обробки розподілених транзакцій. В мікросервісних системах, де бізнес-операції можуть перетинати межі багатьох служб, забезпечення атомарності, узгодженості, ізоляції та стійкості даних – ACID властивостей – стає значно складнішим. Традиційні підходи до управління транзакціями часто не можуть бути застосовані безпосередньо у таких розподілених середовищах. Все більше і більше розробників стикаються із труднощами в розумінні та виборі правильного підходу до опрацювання, фіксації та відкату транзакцій в розподілених масштабованих сервісах. Існують ряд патернів та підходів, які можуть бути по-різному використані в тих чи інших варіаціях мікросервісної архітектури та комунікації сервісів.

Отже, для отримання рішення цих проблем була поставлена задача виконати дослідження методів обробки розподілених транзакцій в мікросервісній архітектурі, що передбачає аналіз та вибір певних шаблонів реалізації

розподілених транзакцій для порівняння, проектування запитів та транзакцій, та, відповідно, проведення експериментів та формування практично обґрунтованих рекомендацій на фундаменті предметної галузі у сфері електронної комерції.

Під час виконання кваліфікаційної роботи був проведений аналіз проблемної області обробки розподілених транзакцій на основі публікацій світових та вітчизняних фахівців в проблемній області (див. додаток А). Обрано варіацію мікросервісної архітектури в комбінації синхронної та асинхронної взаємодії разом із патерном «DB per service» для ефективного моделювання розподілених транзакцій. Був обраний патерн SAGA на основі хореографії як конкретний підхід для дослідження природи обробки розподілених транзакцій. Були обрані варіації реалізації атомарного оновлення бази та публікації повідомлень на основі Transactional Outbox та Event Sourcing

Також, було спроектовано процес проведення експерименту дослідження, що в свою чергу включає вибір предметної галузі, проектування схем та таблиць БД, проектування архітектури та способів комунікації мікросервісів, проектування бізнес-процесів для виконання розподілених транзакцій, вибір метрик, планування умов проведення експерименту, розробка допоміжного програмного забезпечення.

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

На основі плану проведення дослідження та його результатів було розроблено презентацію (див. додаток В). За результатами роботи були створені тези доповіді на 28-й Міжнародний молодіжний форум «Радіоелектроніка і молодь у XXI столітті» (див. додаток Г), а також підготовлено для подачі наукову статтю – «Research of choreography methods for processing distributed transactions in a microservice architecture based on the SAGA pattern» (див. додаток Г).

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

1 АНАЛІЗ ПРОБЛЕМНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз проблемної області дослідження

Впровадження високопродуктивних обчислювальних систем є критично важливим для підтримки зростаючих вимог сучасного цифрового бізнесу. У контексті неперервного прагнення до оптимізації обслуговування запитів клієнтів, здатність швидко та ефективно обробляти транзакції стає вирішальною для успіху багатьох компаній. З огляду на стрімке зростання обсягів даних і транзакцій, які вимірюються вже не тільки гігабайтами, але й терабайтами, атомарність та консистентність даних виступають як необхідні умови для забезпечення цілісності і надійності операцій, що відповідають фундаментальним ACID-принципам.

У відповідь на такі виклики, мікросервісна архітектура набуває все більшої популярності як засіб досягнення масштабованості, гнучкості та швидкості розгортання. Застосування мікросервісів дозволяє розбити монолітні системи на більш автономні компоненти, що сприяє незалежності розробки та деплойменту окремих сервісів. Такий підхід не лише полегшує ведення комплексних проектів, але й підвищує відмовостійкість системи в цілому.

Мікросервісна архітектура представляє собою передовий метод розробки програмного забезпечення, що базується на концепції поділу системи на множину взаємонезалежних сервісів [1]. Кожен з цих сервісів відповідає за виконання окремої бізнес-задачі та взаємодіє з рештою системи через стандартизовані, часто використовуючи HTTP API, інтерфейси. Унікальною особливістю кожного мікросервісу є його автономність – наявність власної бази даних та бізнес-логіки, що надає можливість окремо масштабувати та оновлювати кожен компонент системи без впливу на інші частини. Такий підхід знаходить застосування в широкому спектрі галузей, у тому числі в електронній торгівлі, медіа, фінансах та телекомунікаціях.

Розвиток мікросервісної архітектури є відповіддю на потреби сучасного динамічного бізнес-середовища, де швидкість, гнучкість і надійність розробки

програмного забезпечення стають вирішальними. У контексті DevOps практик та необхідності забезпечення безперервної розробки та впровадження, мікросервіси пропонують модель, що дозволяє командам працювати незалежно, фокусуючись на конкретних субдоменах або бізнес-функціях [2].

Оманливо проста концепція мікросервісів насправді приховує за собою велику композицію згрупованих логічно патернів [3], які вирішують ту чи іншу проблему в контексті інфраструктури, комунікації, обробки бізнес-логіки, безпеки, тестування, моніторингу, декомпозиції задач, надійності системи, масштабованості, зберігання та оброки даних тощо (див. рис. 1.1).

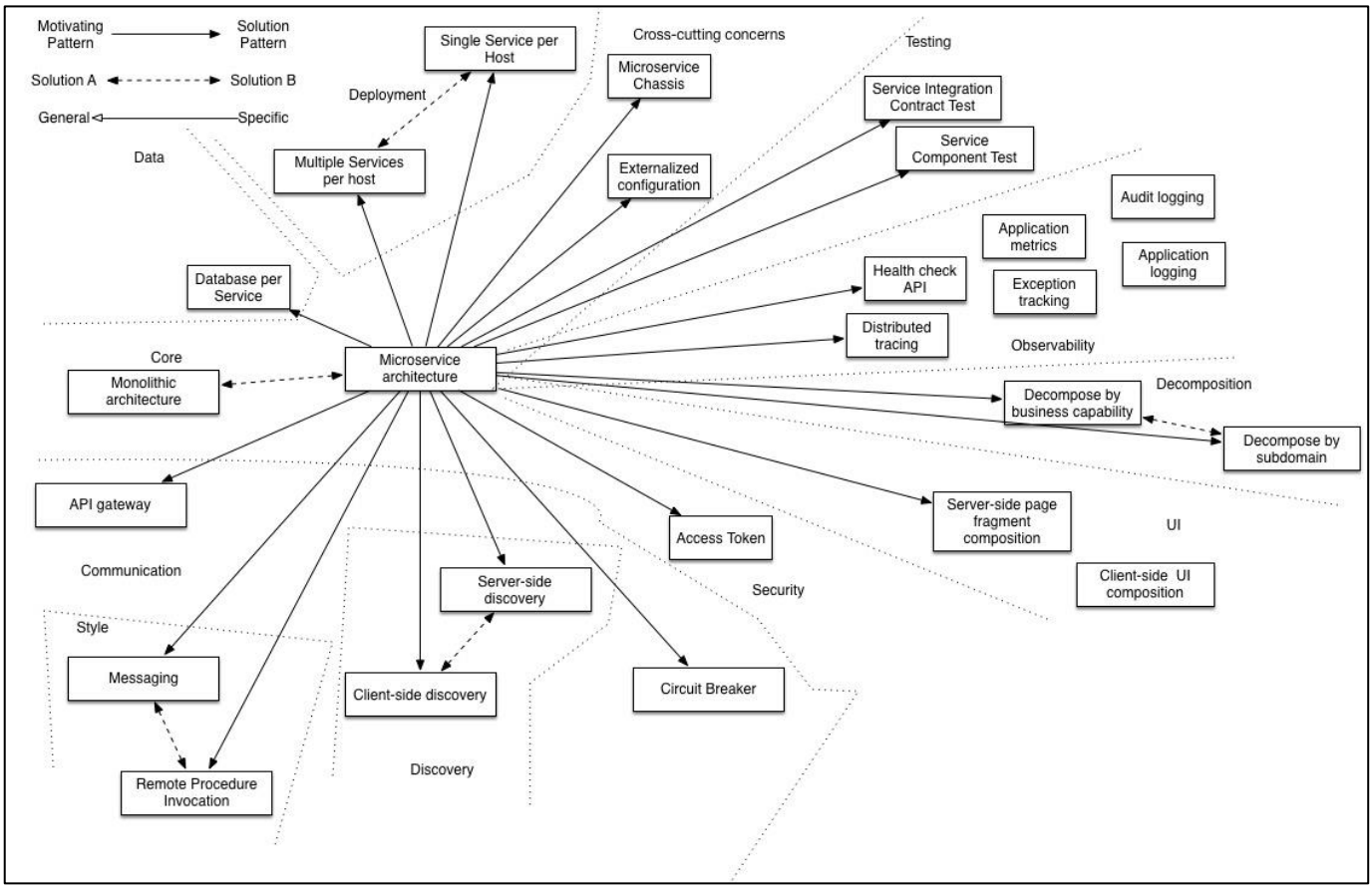

Рисунок 1.1 – Розподіл шаблонів проектування мікросервісів (за даними [3])

Великі технологічні компанії, такі як Netflix, Amazon та Uber, демонструють успішний перехід від монолітних архітектур до мікросервісів [4], підкреслюючи важливість такої трансформації для підтримки масштабованості, і гнучкості.

Переваги мікросервісної архітектури включають:

– простоту сервісів, кожен сервіс є легким для розуміння та підтримки;

– автономію команд, можливість розробки, тестування та розгортання сервісів незалежно від роботи інших команд;

– швидкість розгортання, маленькі сервіси можуть бути швидко протестовані та розгорнуті;

– підтримку різних технологічних стеків: сервіси можуть використовувати різні мови програмування та фреймворки, так само як і різні БД: на відміну від SQL, бази даних NoSQL можуть стати кращим вибором для великої кількості Web-систем, якщо є мета створення масштабованих баз даних з високою продуктивністю та широкою функціональністю [5].

Водночас, існують потенційні недоліки:

– складність розподілених операцій: деякі операції можуть бути важкими для розуміння та усунення проблем;

– потенційна неефективність, так як деякі розподілені операції можуть вимагати значного мережевого обміну даними;

– складність управління транзакціями: розподіл логіки і даних між окремими мікросервісами призводить до нових викликів, серед яких обробка розподілених транзакцій виокремлюється як одна з найбільших проблем.

Перш ніж розглянути проблематику розподілених транзакцій, варто проаналізувати два основних варіанти зберігання даних в наведеній архітектурі: «Shared database» та «Database per service».

Підхід «Shared database» передбачає використання спільної бази даних для декількох сервісів, що може спростити роботу з даними, але водночас створює тісний зв'язок між сервісами, обмежуючи їхню незалежність та масштабованість [6]. Цей підхід може призвести до ускладнення розгортання та оновлення сервісів, оскільки зміни в одному сервісі потенційно можуть вплинути на інші, що використовують ту саму базу даних.

Натомість, підхід «Database per service» полягає в тому, що кожен мікросервіс використовує власну базу даних [7], що забезпечує високий рівень автономії та спрощує розгортання та масштабування сервісів (див. рис. 1.2).

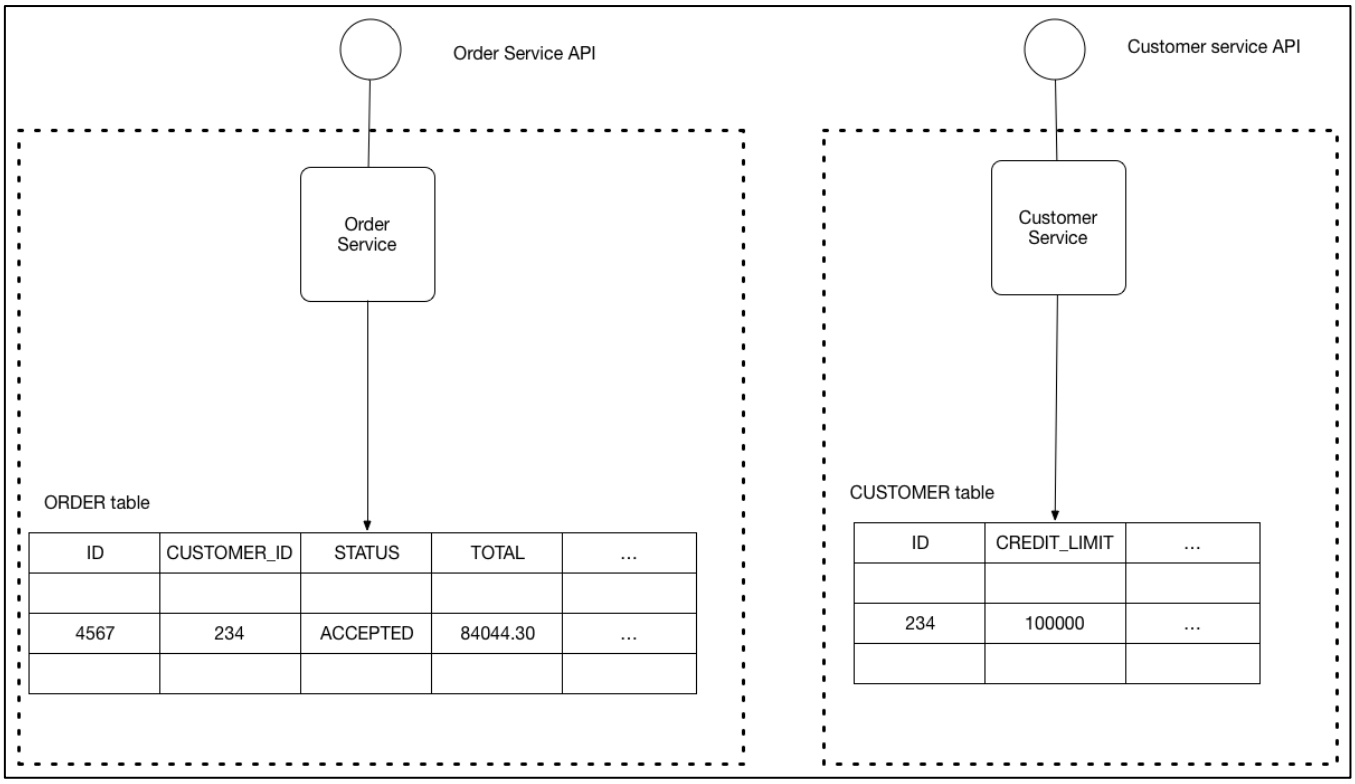

Рисунок 1.2 – Розподілення схем баз даних під відповідне API мікросервісу (за даними [7])

Цей підхід дозволяє кожному сервісу ефективно управляти своїми даними, оптимізувати схеми та індекси під конкретні запити та операції. Водночас, «Database per service» ставить перед розробниками виклик у координації транзакцій, що зачіпають декілька сервісів. В контексті мікросервісної архітектури підхід «Database per service» виявляється більш прийнятним, оскільки він відповідає основним принципам незалежності та гнучкості мікросервісів. Однак саме цей підхід і породжує потребу в ефективних рішеннях для обробки розподілених транзакцій.

Розподілена транзакція – це комплексна операція, що включає в себе кілька окремих транзакцій, виконуваних на різних вузлах баз даних або мікросервісах, але які мають бути розглянуті як єдине ціле з точки зору забезпечення атомарності,

консистентності, ізоляції та стійкості даних (ACID-властивості). Розподілена транзакція забезпечує, що всі складові транзакції будуть успішно завершені, і в цьому випадку зміни будуть зафіксовані в усіх задіяних системах, або, у разі виникнення помилки в будь-якій з частин, всі часткові транзакції будуть відкотити до початкового стану, забезпечуючи тим самим цілісність даних у розподіленій системі.

В умовах збільшення складності бізнес-процесів та розширення обсягів даних, що обробляються в цифрових системах, забезпечення ефективної обробки розподілених транзакцій стає не лише технологічним викликом, а й бізнес-необхідністю. Це пояснюється ще й зростанням складності бізнес-моделей, що вимагають інтеграції різноманітних сервісів та платформ. Здатність системи ефективно керувати розподіленими транзакціями без втрати продуктивності та надійності стає ключовою для підтримки високого рівня обслуговування клієнтів та забезпечення стійкого розвитку компанії. Таким чином, дослідження та впровадження рішень в області управління розподіленими транзакціями стають важливим елементом стратегії цифрової трансформації бізнесу, який обрав для себе світ мікросервісів.

1.2 Постановка задачі

Розглянувши сучасні виклики у сфері обробки розподілених транзакцій в мікросервісній архітектурі, сформулюємо задачу. Задачею є дослідження методів обробки розподілених транзакцій в мікросервісній архітектурі. Передбачено розглянути ці методи для найбільш популярних варіацій архітектур, комунікації, способів зберігання даних, порівняти ключові аспекти, які саме варіації найбільш підходять для забезпечення ACID та тих чи інших бізнес-правил та процесів, на що саме слід звертати увагу при проектуванні системи.

Об’єктами дослідження є різноманітні способи обробки розподілених транзакцій, підходи до їх реалізації, реляційні та нереляційні бази даних, мікросервіси, інтегровані з цими методами, та їх взаємодія у цьому контексті, особливості

використання та технічні обмеження.

Дослідження передбачає вирішення наступних задач:

– провести аналіз та вибір методів обробки розподілених транзакцій;
– провести планування експериментального дослідження обраних методів, а саме, розробити схему БД, обрати метрики для замірів під час експериментів, спроектувати архітектуру мікросервісного додатку, спроектувати транзакційні розподілені запити для дослідження;
– реалізувати програмну систему та обрані методи для спроектованих запитів;
– провести експериментальне дослідження;
– оцінити якість методів обробки розподілених транзакцій та виробити рекомендації стосовно їх вибору та використання.

Дослід включатиме проведення експериментів на одній локальній машині при однакових умовах та обчислювальній можливості з використанням технологій контейнеризації програмного забезпечення. Для додаткового підґрунтя реалізованих методів обробки розподілених транзакцій обрано реальну предметну область в сфері електронної комерції із можливостями додавання товарів у корзину, створення замовлень, оновлення каталогу.

2 ОПИС ПРИЙНЯТИХ ПРОЕКТНИХ РІШЕНЬ

2.1 Аналіз та вибір методів обробки розподілених транзакцій для подальшого дослідження

Перед проведенням власне дослідження, варто проаналізувати наявну проблематику, розглянути існуючі способи вирішення та зупинитись на варіанті, який влаштує всі компроміси. Перше за все, потреба у використанні механізмів контролю розподілених транзакцій з'являється при застосуванні підходу «подвійного запису», коли має відбутись фіксація даних в декількох незалежних сховищах.

Іншими словами, проблема «подвійного запису» виникає, коли необхідно забезпечити синхронізоване оновлення даних у декількох системах, що може призвести до неконсистентності стану в разі виникнення збоїв. Ця ситуація є звичайною для розробки розподілених систем, де кожен компонент, такий як, наприклад, база даних, індекс пошукової системи чи кеш, повинен бути оновлений як частина єдиної бізнес-транзакції. Сценарії включають:

– необхідність мікросервісу повідомити інший мікросервіс про зміни та одночасно оновити власну базу даних;
– бізнес-транзакції, які охоплюють кілька мікросервісів, вимагаючи кожним з них виконувати свою частку операції;
– ідемпотентність операцій, необхідність для клієнтів повторювати операції у випадку їх невдачі.

Проблема стає ще складнішою, коли розглядається сценарій, в якому клієнтська програма ініціює зміну, викликаючи мікросервіс, який має оновити базу даних та взаємодіяти з іншими службами. Ризики подвійного запису в такому сценарії полягають у можливості виникнення збоїв після фіксації змін у базі даних, але перед повідомленням іншого мікросервісу, залишаючи систему у неузгодженому стані.

Ілюстрація на рисунку 2.1 демонструє цю дилему: використання шаблону "закріпити локально, потім опублікувати" несе ризик того, що збій у програмі після запису у базу даних, але до відправлення повідомлення, може призвести до втрати синхронності стану даних [8]. Ні вид бази даних, ні протокол комунікації мікросервісів не впливають на перебіг проблеми [9].

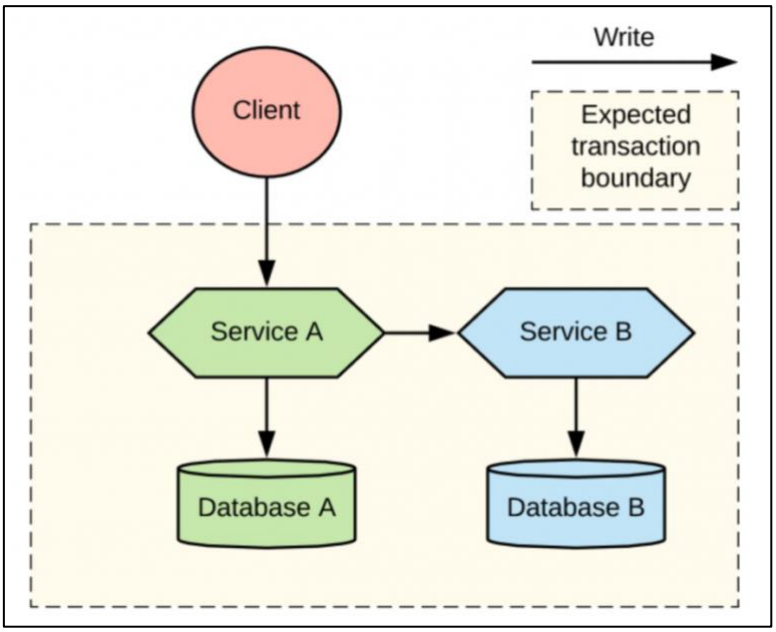

Рисунок 2.1 – Проблематика механізму подвійного запису (за даними [10])

Ймовірність збою між записом у базу даних та відправленням події в чергу може призвести до неконсистентного стану даних, якщо мікросервіс А фіксує зміни до своєї бази даних, але через непередбачені обставини не може доставити повідомлення до мікросервісу В.

Задача вибору стратегії обробки розподілених транзакцій буде розглянута як задача багатокритеріального прийняття рішень. Для цього, перш за все, варто описати множину альтернатив та критеріїв вибору.

Існують наступні варіації обробки розподілених транзакцій, або проблеми «подвійного запису», які виступлять в якості множини альтернатив:

– модульний моноліт;

– двофазний коміт (2PC);

– оркестрація транзакцій;

– хореографія транзакцій;

– паралельні пайплайни.

Для вибору варіації для дослідження варто проаналізувати кожен з них, виділити переваги та недоліки, провести проекцію через «призму» мікросервісної архітектури, проаналізувати, наскільки задовольняються критерії вибору. В якості множини критеріїв вибору розглянемо:

– виконавче середовище: єдине чи розподілене;
– сховище даних: єдине чи гетерогенне;
– координація: центральна чи розподілена;
– транзакційні властивості: ACID, блокуючі або неблокуючі (синхронні або асинхронні);
– реалізація: легкість впровадження, гнучкість, масштабованість.

Почнемо з модульного моноліту. Попри загальну тенденцію до розподіленої архітектури, існують ситуації, коли розробка модульного моноліту може бути доцільною. Модульний моноліт є альтернативою мікросервісам, яка підходить, коли вища важливість приділяється строгій консистентності даних, ніж можливості незалежного розгортання та масштабування.

У модульному моноліті кожен компонент розробляється та інтегрується як бібліотека в єдиному середовищі виконання. Такий підхід дозволяє ефективно управляти транзакціями, які включають декілька модулів, забезпечуючи атомарність та консистентність за допомогою локальних транзакцій. Відповідно, оновлення бази даних та інші операції відбуваються в рамках цієї ж транзакції, уникнувши проблем узгодженості.

Розгортання модулів у такій архітектурі також відрізняється від мікросервісів: замість незалежного розгортання кожного мікросервісу, модулі розгортаються як частини більшого додатку, з можливістю використання спільних транзакцій. Цей метод вимагатиме перетворення мікросервісів A і B на бібліотечні модулі, які можна

інсталювати в спільному середовищі виконання.

Розподіл ресурсів і відповідальності в модульному моноліті також може бути ефективним, навіть у спільному середовищі виконання. Можлива ізоляція коду та даних, де різні команди управляють своїми модулями та базами даних, навіть якщо фізично вони розміщені на тому ж сервері (див. рис. 2.2). Шляхом класифікації таблиць відповідно до умов іменування, схем, баз даних або навіть серверів баз даних можна досягти часткової ізоляції даних.

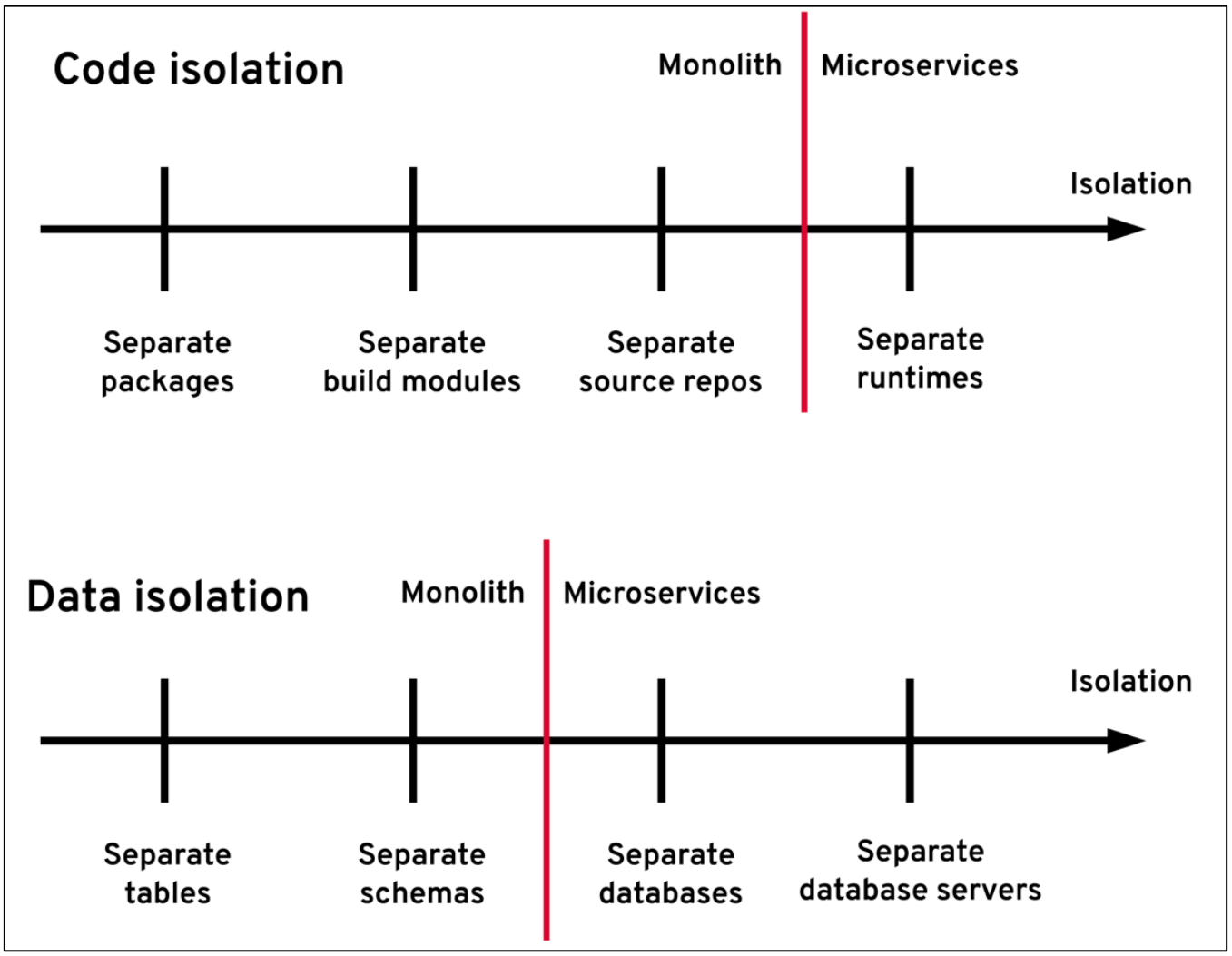

Рисунок 2.2 – Ізоляція коду та даних (за даними [11])

Для прикладу мікросервіси A і B перетворюються на бібліотеки та встановлюються в спільному середовищі виконання (див. рис. 2.3).

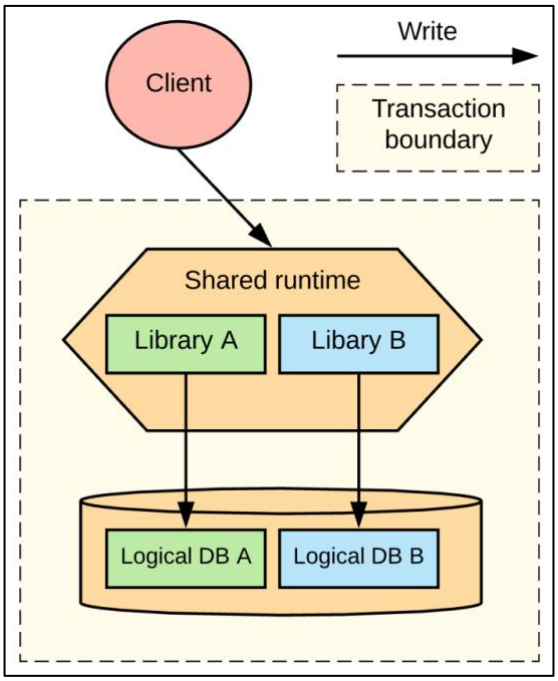

Рисунок 2.3 – Модульний моноліт (за даними [11])

Використання простої локальної семантики транзакцій для забезпечення узгодженості даних, читання записів, відкатів тощо є головною перевагою модульного моноліту. Але є недоліки, наприклад, спільне середовище виконання, що перешкоджає ізоляції помилок і незалежному розгортанню та масштабуванню модулів.

Двофазні транзакції, в свою чергу, зазвичай використовуються в таких ситуаціях:

- коли необхідно записувати в різнорідні сховища даних;
- якщо система не може бути перероблена для забезпечення ідемпотентних операцій, але потрібна гарантія одноразової обробки повідомлення;
- під час включення специфікації двофазної транзакції в сторонні застарілі системи.

Двофазні транзакції можна брати до уваги в усіх цих сценаріях, де відсутність горизонтальної масштабованості не є проблемою. Розподілений менеджер транзакцій і надійний рівень зберігання для журналів транзакцій необхідні для

технічного функціонування двофазної фіксації. Кеші, брокери повідомлень і системи керування реляційними базами даних, які підтримують розподілені транзакції, також необхідні. Журнал, який підтримує стан транзакцій, завжди повинен бути доступний менеджеру транзакцій [12].

Щоб усунути можливість дублювання або втрати повідомлень, мікросервіс A у прикладі (див. рис. 2.4) виконує всі оновлення бази даних і черги повідомлень за допомогою двофазних транзакцій. Подібним чином мікросервіс B може використовувати повідомлення та передавати їх до бази даних B в одній транзакції без дублікатів за допомогою двофазних транзакцій.

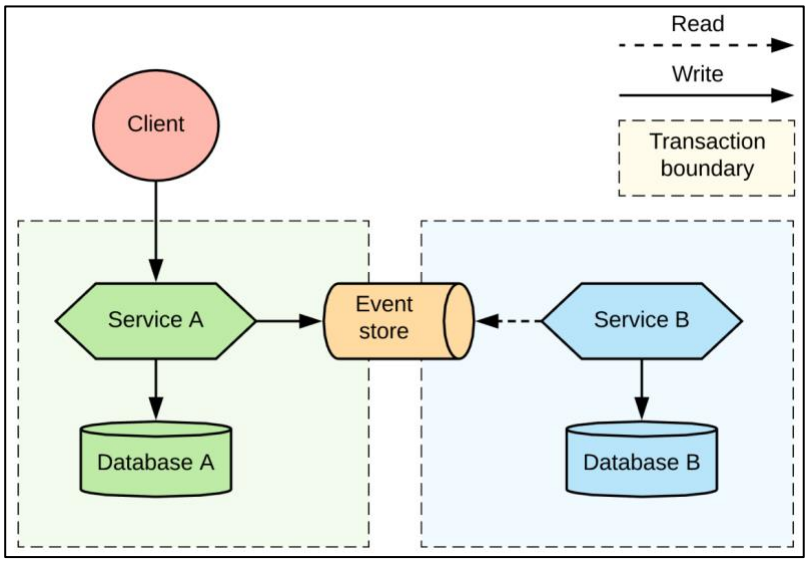

Рисунок 2.4 – Двофазні транзакції (за даними [13])

За кількома помітними винятками, двофазові транзакції забезпечують порівняні гарантії з локальними транзакціями в модульно-монолітному підході. Атомарне оновлення, що включає два або більше різних сховища даних, може призвести до збою та заблокувати транзакцію [14]. Однак, порівняно з іншими методами, які будуть розглянуті надалі, відновлення стану розподіленої системи все одно просте завдяки центральному координатору.

З модульним монолітом використовуються локальні транзакції, і статус системи завжди доступний. Послідовний стан додатково забезпечується

розподіленими транзакціями на основі двофазного протоколу фіксації. Однак що було би, якби ми могли координувати розподілену систему з одного місця, знаючи її загальний стан, тим самим зменшивши вимоги до узгодженості? У цій ситуації ми можемо застосувати оркестрований підхід, коли мікросервіс координує всю розподілену зміну стану (див. рис. 2.5).

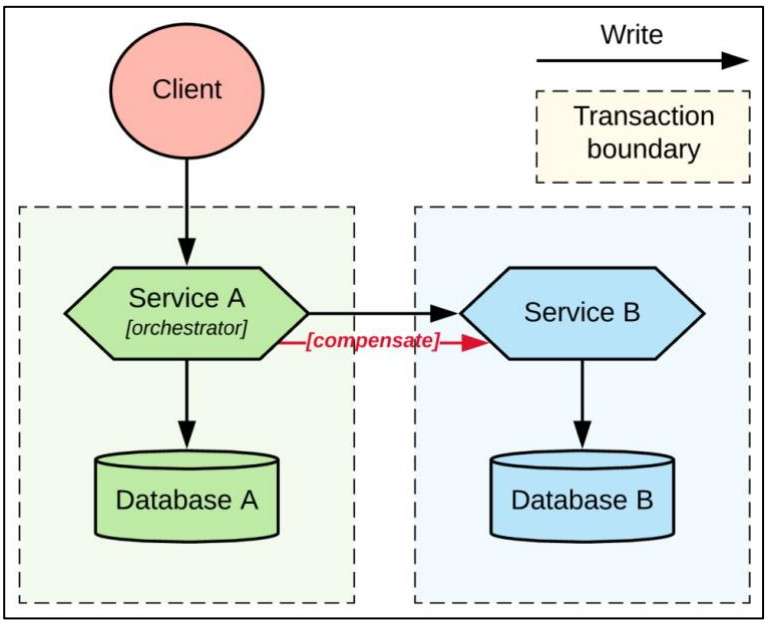

Рисунок 2.5 – Оркестрація транзакцій (за даними [15])

Мікросервіс оркестратора відповідає за виклик інших мікросервісів, доки вони не досягнуть необхідного стану, або, у разі збою, ініціює коригувальну або компенсувальну дію. Оркестратор відповідає за відновлення після збою та відстежує зміни стану за допомогою своєї локальної бази даних.

Як координатор станів у прикладі, мікросервіс А відповідає за виклик мікросервісу В і, якщо необхідно, компенсує збої шляхом виконання операції компенсації. Ключовим компонентом цієї стратегії є той факт, що і мікросервіс А, і мікросервіс В використовують локальні транзакції, тоді як мікросервіс А відповідає за нагляд і керування всім комунікаційним потоком. Є варіації для використання синхронної взаємодії або черги повідомлень, яка підтримує двофазові транзакції для досягнення зв’язку.

Повторні спроби та відкати можуть знадобитися, щоб привести систему в узгоджений стан, оскільки оркестровка транзакцій гарантує лише узгодженість результату. Хоча оркестровка усуває потребу в двофазних транзакціях, вона вимагає участі мікросервісу та ідемпотентності транзакцій у випадку, якщо координатору потрібно повторити транзакцію. Основною перевагою цього методу є його здатність використовувати лише локальні транзакції для переведення неоднорідних мікросервісів, які можуть не підтримувати двофазні транзакції, у скоординований стан.

Координатору та мікросервісам, які беруть участь, потрібні лише локальні транзакції. Навіть у випадках, коли система лише частково узгоджена, завжди можна надіслати запит координатору про поточний стан системи. Іншими засобами це зробити неможливо.

Переваги оркестровки транзакцій включають:

– координація стану розрізнених компонентів;

– двофазові транзакції не потрібні; визнаний розосереджений стан на рівні координатора.

Недоліки оркестровки транзакцій:

– складна модель розподіленого програмування;

– можуть знадобитися компенсаційні транзакції від спільних мікросервісів і підтримка ідемпотентності;

– тільки кінцевий продукт демонструє консистенцію;

– потенційно незворотні помилки під час процесу компенсації.

В свою чергу, на перевагу оркестрації, хореографія – це інший вид координації мікросервісів, у якому користувачі спілкуються подіями без єдиної точки контролю [16]. Використовуючи цей шаблон, кожен мікросервіс виконує локальну транзакцію та публікує події, які змушують інші мікросервіси виконувати локальні транзакції. Замість того, щоб залежати від однієї точки контролю, кожна частина системи бере участь у вирішенні того, як здійснюється робочий процес бізнес-транзакцій.

Використання асинхронного рівня обміну повідомленнями між мікросервісами є найпопулярнішим способом реалізації техніки хореографії транзакцій (див. рис. 2.6).

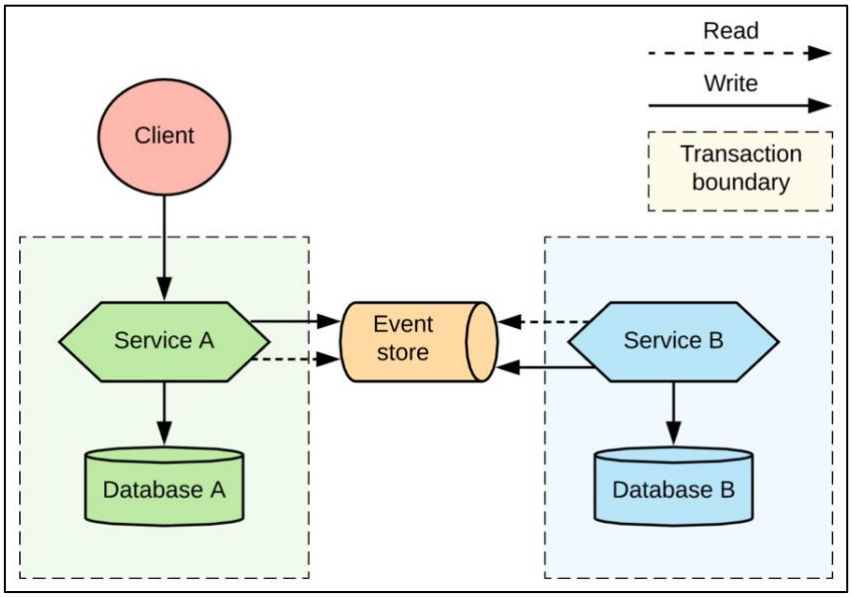

Рисунок 2.6 – Хореографія транзакцій (за даними [17])

Кожен мікросервіс, який бере участь, повинен виконати локальну транзакцію та викликати наступний мікросервіс, опублікувавши команду або подію в інфраструктурі обміну повідомленнями, щоб хореографія на основі повідомлень працювала. Подібним чином завершення локальної транзакції та споживання повідомлень вимагаються від інших залучених мікросервісів.

Ця проблема подвійного запису є підпроблемою проблеми подвійного запису вищого рівня. Хореографію транзакцій можна реалізувати у двофазовій транзакції, яка охоплює брокер повідомлень і локальну базу даних під час розробки рівня обміну повідомленнями подвійного запису. Крім того, можна використовувати наступні методи:

– перш ніж здійснювати локальну транзакцію, повідомлення мають бути опубліковані. Незважаючи на явні переваги, цей варіант має деякі недоліки. Наприклад, ідентифікатор, створений під час фіксації транзакції, часто вимагається для публікації, але він не буде опублікований. Крім того,

помилка може завершити локальну транзакцію, але опубліковане повідомлення не можна скасувати. Оскільки йому бракує семантики, необхідної для читання нещодавно зафіксованих записів, цей метод непрактичний для більшості сценаріїв транзакцій;

– перед публікацією повідомлення необхідно зафіксувати локальну транзакцію. Існує невелика ймовірність того, що це не працюватиме після транзакції, але до того, як повідомлення буде опубліковано. Тим не менш, транзакцію можна повторити, а мікросервіси зробити ідемпотентними. Це передбачає публікацію повідомлення після повторного виконання транзакції. Ця стратегія може бути ефективною, якщо є змога зробити клієнтів ідемпотентами.

Різні реалізації хореографічної архітектури вимагають від кожного мікросервісу запису в єдине сховище даних через локальну транзакцію і ніде більше. Варто подумати, як це може функціонувати без дублювання роботи. Припустимо, що запит отримано мікросервісом А і він записується лише в базу даних А. Мікросервіс В регулярно опитує мікросервіс А, щоб визначити будь-які оновлення. Мікросервіс В оновлює власну базу даних і індекс або позначку часу останньої зміни щоразу, коли вносяться зміни. Важливо те, що кожен мікросервіс виконує локальну транзакцію та записує виключно у свою власну базу даних. Цю методологію (див. рис. 2.7) можна охарактеризувати як конвеєрну передачу даних або хореографію.

У найпростішому випадку мікросервіс В підключається до бази даних мікросервісу А і зчитує її таблиці.

З поважних причин індустрія уникає такого рівня спілкування: будь-які модифікації способу впровадження мікросервісу А можуть призвести до збою мікросервісу В. Цей сценарій можна трохи покращити, наприклад, надавши мікросервісу А таблицю, яка служить загальнодоступним інтерфейсом, і застосувавши шаблон «Вихідні».

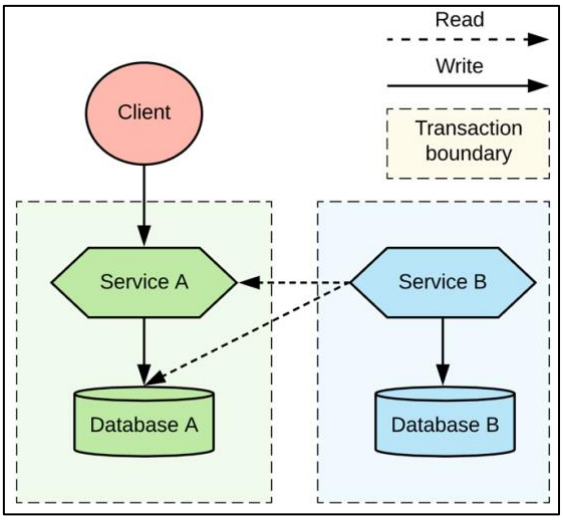

Рисунок 2.7 – Хореографія транзакцій без подвійного запису (за даними [17])

Якщо цього все ще недостатньо, було б навіть краще, якби мікросервіс B використовував API для запиту мікросервісу A щодо будь-яких змін, а не встановлював пряме з’єднання з базою даних A.

По суті, у всіх цих варіаціях є один недолік: мікросервіс B повинен постійно опитувати мікросервіс A. Це може призвести до непотрібної затримки отримання оновлень або непотрібного безперервного навантаження на систему. Оскільки опитувати мікросервіс щодо змін може бути складно, варто подумати про те, як покращити цю архітектуру.

Іншим застосуванням методології мікросервісної хореографії є пошук подій, заснований підході Event Sourcing [18]. Стан об’єкта записується за допомогою цього шаблону як серія подій про зміну стану у спеціальне сховище подій. У разі появи нового оновлення до списку подій додається подія, а не стан об’єкта, що оновлюється. Атомарна операція, що виконується в локальній транзакції, додає нові події до сховища подій. За потреби отримати актуальний стан доменного об’єкту, або ж агрегату [19] (графу пов’язаних між собою логічно сутностей та об’єктів) в термінах DDD [20], зі сховища даних вигружаються всі події та наново «відіграються», застосовуючи всі зміни до пустого об’єкту.

Однією з найкращих переваг методу Event Sourcing (див. рис. 2.8) є те, що інші

мікросервіси, які отримують оновлення в реальному часі зі сховища подій, також можуть взаємодіяти з ним у вигляді серії повідомлень. Окрім того, Event Sourcing зачасту інтегрується з шаблоном CQRS для запровадження read-only репліки бази даних [21].

Запити клієнтів додаються до сховища подій лише під час його використання. Відтворюючи події, Microservice A може реконструювати свій поточний стан. Microservice B також може підписатися на ті самі події оновлення через сховище подій. Використовуючи цей метод, мікросервіс A спілкується з іншими мікросервісами через рівень зберігання. Хоча цей механізм є дуже розумним і вирішує проблему постійної публікації подій щоразу, коли відбувається зміна стану, він створює додаткову складність, пов’язану зі стисненням повідомлень і реконструкцією стану, що потребує спеціалізованих сховищ даних, а також новий стиль програмування, незнайомий багатьом розробникам.

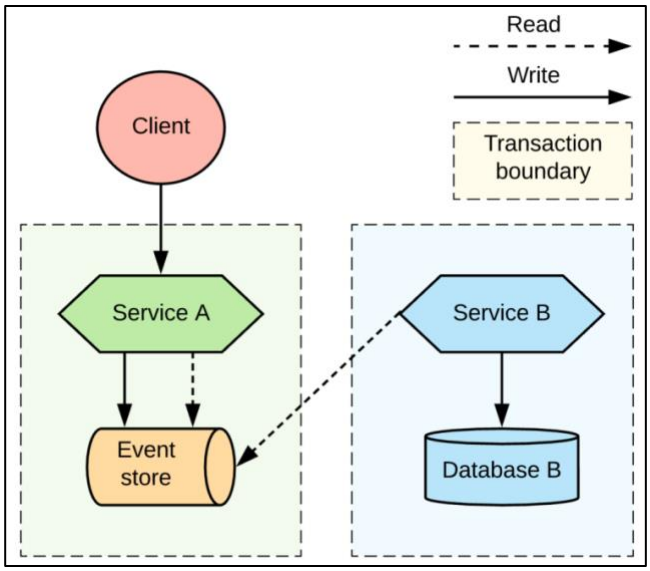

Рисунок 2.8 – Хореографія транзакцій без подвійного запису із застосуванням Event Sourcing (за даними [22])

Незалежно від способу отримання даних, хореографія зберігає записи окремо, дозволяє мікросервісам самостійно масштабуватися та підвищує загальну стабільність системи.

Недоліком цього підходу є його децентралізований потік рішень, що ускладнює ідентифікацію глобально розподілених станів. Коли є багато мікросервісів, може бути складно запитати кілька сховищ даних, щоб визначити статус запиту клієнта.

Переваги хореографії транзакцій включають:

– розрізняє комунікацію та бізнес-логіку;
– відсутність центрального координатора транзакцій, покращені властивості стійкості та масштабованості, практично миттєва співпраця;
– зменшення навантаження на систему через наявність Event Store.

Недоліки оркестровки транзакцій:

– логіка стану системи розпорошена серед усіх залучених сторін;
– тільки кінцевий продукт демонструє консистенцію.

Коли використовується хореографія, розподілена система поширюється через низку мікросервісів, а не через єдину точку запиту стану системи. Оскільки хореографія будує конвеєр мікросервісів послідовно, повідомлення гарантовано пройдуть через кожен крок, перш ніж досягнуть певної точки в загальному процесі.

Що сталося б, якби ми змогли пом’якшити це обмеження й виконувати кожен крок самостійно? У цьому випадку запит може бути оброблений мікросервісом B незалежно від того, чи був він оброблений мікросервісом A (див. рис. 2.9).

Маршрутизатор мікросервісу, який отримує запити та направляє їх до мікросервісів A і B через брокер повідомлень в одній локальній транзакції, встановлюється за допомогою паралельних конвеєрів [24].

Запити можуть оброблятися незалежно та одночасно обома мікросервісами. Цей шаблон найкраще працює в сценаріях, де немає прив’язки між мікросервісами, незважаючи на те, що його надзвичайно легко реалізувати. Мікросервіс Б повинен мати можливість обробити запит незалежно від того, чи обробив його мікросервіс А. Крім того, такий підхід вимагає додаткового маршрутизатора або знання клієнтом обох сервісів для відправки повідомлень.

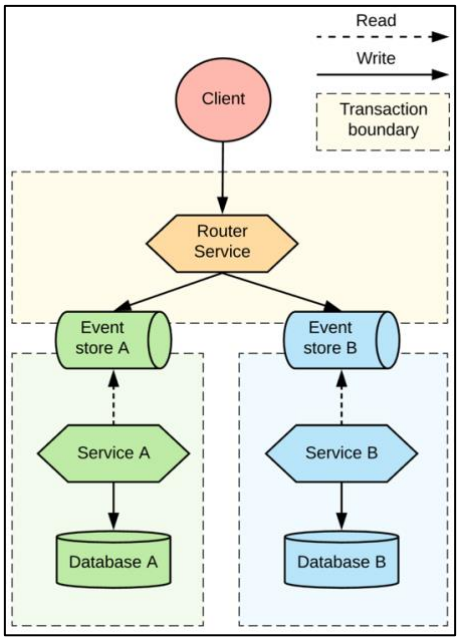

Рисунок 2.9 – Паралельний пайплайн (за даними [23])

Простіший спосіб зробити це – використовувати шаблон «Listen to yourself», в якому мікросервіс також виконує роль маршрутизатора. Використовуючи цей альтернативний метод, коли мікросервіс A отримує запит, він публікує його в системі обміну повідомленнями, яка потім передає повідомлення як мікросервісу B, так і мікросервісу A (див. рис. 2.10), замість запису в свою базу даних.

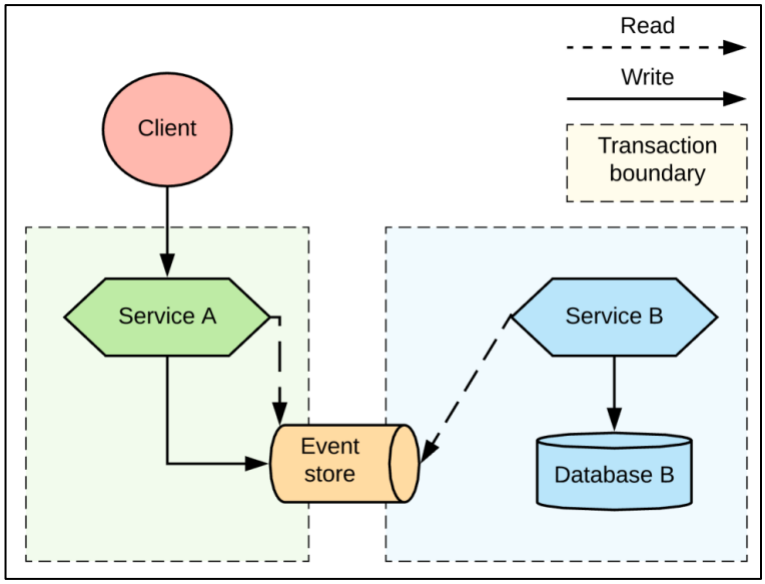

Рисунок 2.10 – Паралельний пайплайн із використанням шаблону «Listen to yourself» (за даними [24])

Мікросервіси B і A отримують повідомлення, які надходять у систему обміну повідомленнями в абсолютно різному контексті транзакцій. Мікросервіси A і B можуть незалежно обробляти запити та записувати дані у відповідні бази даних за допомогою цієї обробки.

У випадках, коли синхронізація між мікросервісами не потрібна для виконання транзакцій, паралельні конвеєри пропонують простий і масштабований засіб обробки запиту кількома мікросервісами одночасно, незалежно від того, як вони реалізовані.

Коли справа доходить до обробки розподілених транзакцій в архітектурі мікросервісу, немає правильного чи неправильного шаблону. Кожен має переваги та недоліки. Кожен шаблон створює нові проблеми, вирішуючи інші. Основні характеристики підходів, які використовуються для вирішення проблеми подвійного запису, зібрані в таблиці 2.1.

Таблиця 2.1 – Порівняння методів обробки розподілених транзакцій (таблиця виконана самостійно)

	Виконавче середовище	Сховища даних	Координація	Транзакційні властивості	Реалізації
Модульний моноліт	Єдине	Єдине	Центральна	ACID, блокуючі, синхронні	Локальні транзакції
Двофазні транзакції	Єдине/Розподілене	Гетерогенні (двофазні)	Центральна	ACID, блокуючі, синхронні	XA
Оркестрація	Розподілене	Гетерогенні	Центральна	ACD, неблокуючі, (а)синхронні	SAGA, Outbox

Кінець таблиці 2.1

	Виконавче середовище	Сховища даних	Координація	Транзакційні властивості	Реалізації
Хореографія	Розподілене	Гетерогенні	Розподілена	ACD, неблокуючі, асинхронні	SAGA, Outbox, Event Sourcing
Паралельні пайплайни	Розподілене	Гетерогенні	Розподілена	ACD, неблокуючі, асинхронні	Listen to yourself

Повертаючись до задачі багатокритеріального прийняття рішень, проаналізуємо отриману таблицю та зведемо значення до числових показників.

Розглядаючи виконавче середовище, перевагу будемо надавати розподіленому, адже саме це є невід'ємною частиною мікросервісної архітектури. Тому модульний моноліт отримує 1 бал, всі інші стратегії – 2.

З приводу сховища даних, перевагу надамо гетерогенному, оскільки вони мають кращі показники відносно відмовостійкості та масштабування. Тому тут виходить аналогічно, модульний моноліт – 1 бал, всі інші – 2 бали.

Відносно координації, в даному досліджені нас цікавить саме розподілена, адже вона не дає системі мати вузьке місце в якості «Single point of failure», дозволяє зменшити зв'язність, незалежність компонентів, гарантувати надійність та безперебійність логіки системи [25]. Тому хореографія та паралельні пайплайни отримують 2 бали, модульний моноліт, двофазні транзакції та оркестрація – 1 бал.

Щодо транзакційних можливостей, 1 бал буде нараховано за кожну з ACID властивостей. Перевагу також надамо неблокуючим, або ж асинхронним підходам, адже вони дозволяють збільшити швидкодію програмної системи. Отже, модульний моноліт та двофазні транзакції – 4 бали за рахунок підтримки ізольованості, всі інші – теж 4 бали, але за рахунок асинхронної природи.

Останній пункт, реалізації, оцінимо наступним чином: локальні транзакції отримують 3 бали за поширеність, простоту реалізації та надійність, XA (eXtended Architecture) – 2 бали, зокрема нараховані за те, що підхід є певною специфікацією та розширенням можливостей 2PC [26], але бал знімається за «legacy» складову технології та складність її впровадження та обслуговування в сучасних системах. Оркестрація отримує 2 бали за наявність SAGA та Outbox, так само як і хореографія, що має той самий набір патернів та Event Sourcing, які є доволі популярними рішеннями, легкими для впровадження як в існуючі системи, так і в нові. Паралельні пайплайни отримують 1 бал за підхід «Listen to yourself», адже він має ряд критичних недоліків – клієнт не зможе отримати відповідь щодо власне запису в базу даних, так само цей метод не гарантує моментальної консистентності локальної бази даних при завершені транзакції [27]. Тому маємо наступні показники (див. табл. 2.2).

Таблиця 2.2 – Показники стратегій обробки розподілених транзакцій (таблиця виконана самостійно)

	Виконавче середовище	Сховища даних	Координація	Транзакційні властивості	Реалізації
Модульний моноліт	1	1	1	4	3
Двофазні транзакції	2	2	1	4	2
Оркестрація	2	2	1	4	2
Хореографія	2	2	2	4	2
Паралельні пайплайни	2	2	2	4	1

Для нормалізації даних по всім пунктам візьмемо формулу 2.1:

$$a_i = \frac{x_i}{max} \tag{2.1}$$

де a_i – нормалізоване значення,

max – максимальне значення серед наявної категорії,

x_i – початкове значення.

Задамо наші пріоритети вектором P = {0.2, 0.1, 0.25, 0.2, 0.25}, що задовольняє вимозі $\sum_{j=0}^{m} p_i = 1$ (p_i – пріоритет за критерієм). Маємо наступну таблицю з нормалізованими даними та даними про пріоритетами (див. табл. 2.3).

Таблиця 2.3 – Нормалізовані дані (таблиця виконана самостійно)

	Виконавче середовище	Сховища даних	Координація	Транзакційні властивості	Реалізації
Модульний моноліт	0.5	0.5	0.5	1	1
Двофазні транзакції	1	1	0.5	1	0.67
Оркестрація	1	1	0.5	1	0.67
Хореографія	1	1	1	1	0.67
Паралельні пайплайни	1	1	1	1	0.33
P	0.2	0.1	0.25	0.2	0.25

Вирішувати дану задачу будемо методом лінійної адитивної згортки з ваговими коефіцієнтами, за формулою 2.2:

$$Z^* = \max_{i=1,n} \sum_{j=0}^{m} p_i \, a_{j,} \tag{2.2}$$

де p_i – вага пріоритету,

a_j – нормована оцінка критерію.

Результати наведено нижче в порядку від найбільшого (див. табл. 2.4).

Таблиця 2.4 – Результат лінійної адитивної згортки (таблиця виконана самостійно)

Стратегія обробки розподілених транзакцій	Z*
Хореографія	0.9175
Паралельні пайплайни	0.8325
Оркестрація	0.7925
Двохфазний коміт	0.7925
Модульний моноліт	0.75

Таким чином, було з’ясовано, що найбільш релевантним методом обробки розподілених транзакцій для дослідження виявилась хореографія. Враховуючи, що до реалізації хореографії відносятся SAGA, Outbox та Event Sourcing, у дослідженні на обраній предметній галузі будуть реалізовані та порівнянні між собою бізнес-транзакції у двох підходах: SAGA + Transactional Outbox та SAGA + Event Sourcing.

2.2 Планування експериментальної частини дослідження

Планування експериментів перш за все зав’язане на предметній області, яка буде для цього обрана. Маючи за основу реальну прикладу предметну область, буде змога виконати проекцію на ту програму систему та ту архітектуру, яка найкращим чином буде підходити для моделювання розподілених транзакцій та їх обробки. Маючи загальну схему даних, архітектуру мікросервісів та принципи їх комунікації,

необхідно змоделювати відповідні бізнес-запити для виконання замірів по обраним метрикам.

Для проведення експериментального дослідження буде використана локальна машина Mac Book Pro з наступною конфігурацією: M3 Pro (12-core CPU з 6 performance ядрами до 4,06 GHz та 6 efficiency ядрами до 2,8 GHz), 36 Gb RAM. Всі мікросервіси та їхня інфраструктура будуть розгорнуті використовуючи Docker. Експеримент буде проведено залучаючи різні види навантаження (послідовне та паралельне виконання певної кількості запитів).

2.2.1 Вибір метрик з оцінювання якості методів обробки розподілених транзакцій

Для оцінки ефективності методів обробки розподілених транзакцій, таких як SAGA + Transactional Outbox та SAGA + Event Sourcing, важливо визначити конкретні метрики, які відображатимуть ключові аспекти їхньої роботи. Обрані метрики дозволять провести глибокий аналіз та порівняння обох підходів з урахуванням їхніх переваг та недоліків у різних умовах. Розглянемо п'ять основних метрик для цього порівняння.

Час виконання вимірює тривалість завершення однієї повної розподіленої транзакції, включаючи всі локальні операції та комунікацію між сервісами. Ця метрика важлива для розуміння загальної продуктивності системи. Вимірюється в мілісекундах (мс).

Розмір журналу подій показує залежність між виконанням транзакції та кількістю і розміром повідомлень, або подій, які були опубліковані для коректного відпрацювання всього ланцюжка SAGA. Ця метрика дасть змогу оцінити додаткове навантаження на сховище даних, спричинене використанням шаблону. Вимірюється в мегабайтах (Mb).

Споживання оперативної пам'яті вказує на кількість ресурсів пам'яті, необхідних віртуальній машині Java (JVM), в якій виконується мікросервіс, для

зберігання стану транзакцій, проміжних даних, кешування та обміну повідомленнями з іншими сервісами. Ефективне використання пам'яті може підвищити загальну продуктивність та масштабованість системи. Вимірюється в мегабайтах (Mb).

Споживання процесорного часу показує яке навантаження виконання транзакцій створюють для процесора, що виконує інструкції в середині контейнера. Розуміння цієї метрики допоможе побачити наскільки ефективні ресурси варто залучати для розгортання і чи є можливість зменшити фінансове навантаження за оренду інфраструктури. Вимірюється у відсотках використання (%).

Пропуска здатність мережі (Network I/O) показує, яке навантаження спричинене транзакціями на мережу і яка кількість даних була передана мережевими протоколами для загальної комунікації сервісів. Вимірюється як співвідношення вхідних до вихідних байтів (bytes/bytes).

Ці метрики дозволять провести всебічний аналіз та порівняння обраних методів обробки розподілених транзакцій з метою визначення їхньої придатності для конкретних сценаріїв використання. Під час дослідження важливо зосередитися на тому, як кожен з підходів впливає на зазначені метрики в різних умовах роботи системи, враховуючи зміни в обсязі транзакцій та навантаженні на систему.

Для часу виконання доречно буде провести серію експериментів, в яких певна кількість запитів буде виконана послідовно, а потім паралельно, тобто одночасно. Це дозволить оцінити, наскільки ефективно кожен з методів справляється зі зростаючим навантаженням.

Аналіз розміру журналу подій допоможе зрозуміти, який з патернів потребує більше місця та подій для мінімального виконання своєї роботи. Це буде корисним для систем де є обмеження на використання дискового простору для БД, або ж на кількість повідомлень в брокері повідомлень.

Для вимірювання обсягу залученої оперативної пам'яті та процесорного часу можна використати інструменти моніторингу ресурсів, які дозволять отримати дані

про використання пам'яті під час виконання транзакцій. Це допоможе ідентифікувати, який із методів ефективніше використовує ресурси системи.

Оцінку пропускної здатності мережі та навантаження на неї допоможе виконати інструмент моніторінгу вхідного та вихідного трафіку мікросервісів, яким також може слугувати відразу ж інструмент, що розгортає мікросервіси через механізм контейнеризації.

Результати аналізу цих метрик дозволять не тільки порівняти SAGA + Transactional Outbox і SAGA + Event Sourcing між собою, але й визначити, які з них краще підходять для конкретних умов застосування. Таке дослідження надасть цінну інформацію для архітекторів та розробників, які прагнуть оптимізувати роботу мікросервісних систем, забезпечуючи надійність та високу продуктивність.

Отже, підсумовуючи, при виконанні кожного етапу експериментального дослідження було вирішено збирати такі метрики:

– загальний час виконання від першого запиту до обробки останньої події в SAGA ланцюжку (мс);
– розмір журналу подій (Mb);
– споживання оперативної пам'яті (Mb);
– споживання процесорного часу (%);
– пропускна здатність мережі, Network I/O (bytes).

2.2.2 Аналіз та моделювання предметної області для дослідження

Для розробки структур баз даних для експерименту було обрано предметну області в сфері електронної комерції з покупки товарів за рахунок додавання їх до кошику покупця та оформлення замовлення. Дана предметна область з великою кількістю розподілених операцій, які залучають різні компоненти системи та мікросервіси. Саме тому вона чудово підходить для виконання даного дослідження. Розглядаючи предметну область електронної комерції, основою є схема бази даних, що охоплює користувачів, товарний кошик, замовлення, товари, категорії товарів, а

також властивості товарів. Електронна комерція включає великий об'єм транзакцій та вимагає ефективного управління даними для забезпечення гладкої взаємодії між користувачами та системою.

Товарний кошик відображає відносини між користувачами та товаром, що вони бажають купити. Кожен кошик має зв'язок з певним користувачем та включає інформацію про кількість товарів, загальну вартість тощо.

Замовлення фіксує завершені покупки, включаючи деталі про ціну, статус замовлення, тип доставки, спосіб оплати та адресу доставки. Формування замовлення виконується за рахунок використання товарів із кошику та очищення кошику при успішній оплаті.

Товари включають в себе таку інформацію, як назва, опис, ціна, кількість на складі. Вони мають зв'язок з категорією та властивостями товару. Категорії служать для класифікації товарів, полегшуючи пошук та навігацію для користувачів. Властивості товарів визначають характеристики та специфікації товарів, що допомагає користувачам у прийнятті рішення про покупку.

Електронна комерція вимагає гнучкості та масштабованості від системи управління базами даних для обробки зростаючих обсягів даних та забезпечення високого рівня доступності та швидкості обслуговування. Проблематика обробки розподілених транзакцій в такій системі стає особливо актуальною при забезпеченні консистентності даних між кошиками, замовленнями та статусами товарів. Використання методів хореографії, таких як SAGA з Transactional Outbox та Event Sourcing, дозволяє ефективно керувати складними бізнес-транзакціями, забезпечуючи консистентність даних при одночасному мінімізуванні залежностей між різними частинами системи.

Для системи електронної комерції критично важливим є забезпечення високої пропускної здатності та низької затримки при обробці замовлень, особливо під час пікових навантажень, наприклад, під час розпродажів або святкових періодів. Використання розподілених транзакцій дозволяє збалансувати навантаження між

різними вузлами системи, забезпечуючи стабільну та ефективну роботу.

Система повинна бути здатною швидко обробляти замовлення, оновлюючи статуси товарів у кошику та відслідковуючи загальну кількість та ціну замовлення. Актуальність даних про запаси товарів критично важлива для запобігання перепродажу товарів, що вимагає реалізації механізмів, які гарантують атомарність та консистентність даних. Ефективне використання даних про користувачів надає змогу для надання персоналізованих рекомендацій товарів та пропозицій, що залежить від здатності системи швидко обробляти великі обсяги даних.

Проаналізувавши загальну діаграму класів, далі будуть спроектовані логічна схема даних та фізичні моделі БД з урахуванням обмежень, описаних в цьому розділі. Розробка схеми бази даних для електронної комерції вимагає врахування специфіки бізнес-процесів, обсягів даних та вимог до швидкості обробки.

2.2.3 Розробка логічної моделі БД

Проектування логічної схеми БД базується на попередньому аналізі предметної галузі, її особливостей використання, транзакційних процесів, потреб користувачів, бізнесу. Було розроблено 9 таблиць БД (див. рис. 2.11).

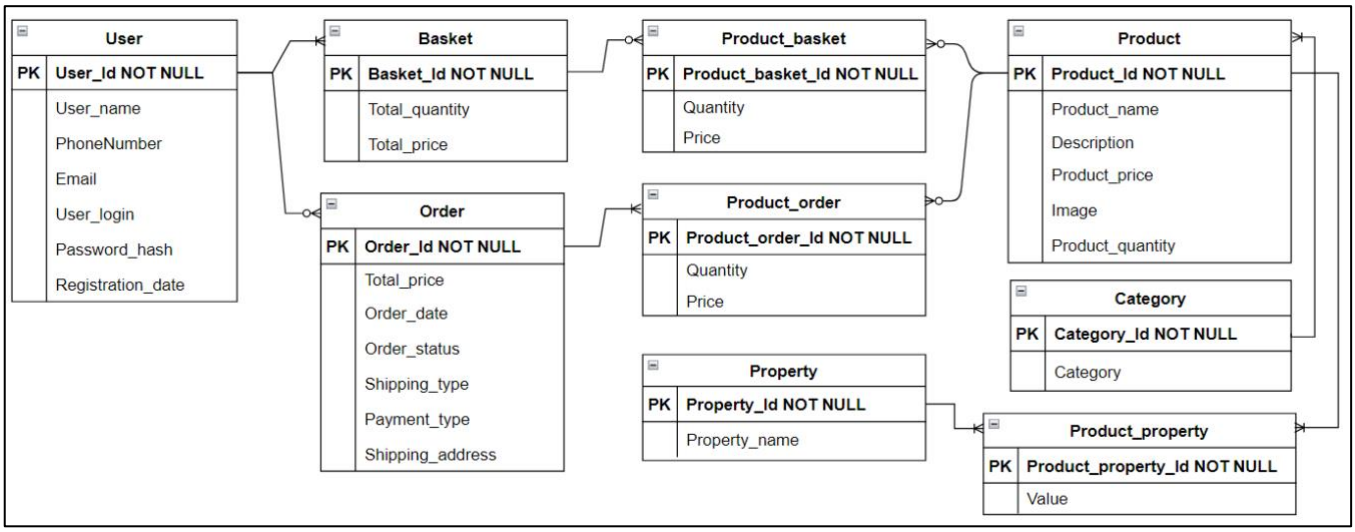

Рисунок 2.11 – ER-діаграма в сфері електронної комерції (рисунок створено самостійно)

Варто зазначити, що наразі діаграма відображає всі наявні в системі дані та зв’язки між ними. Проте, надалі при проектуванні мікросервісної архітектури, БД буде розбита згідно з шаблоном «DB per service».

Користувач може мати багато замовлень та багато створених кошиків. Кошик може містити багато товарів, так само як і товар може бути присутнім в багатьох кошиках. До категорії можуть відноситись багато товарів. Замовлення може мати багато товарів, так само як і один товар може бути присутнім у багатьох замовленнях. Властивість товару може бути присутня у багатьох товарів, так само як і багато товарів можуть мати одну і ту саму властивість.

2.2.4 Проектування мікросервісної архітектури програмної системи

Проектування мікросервісної архітектури для електронної комерції з використанням патерну «DB per service» (адже без його використання відпадає потреба в розподілених транзакціях) полягає у створенні декількох незалежних сервісів, кожен з яких взаємодіє з власною базою даних та спілкується з іншими сервісами шляхом синхронної HTTP взаємодії та асинхронної завдяки брокеру повідомлень. Такий підхід дозволяє забезпечити модульність, легкість управління, масштабування та розвиток кожного сервісу окремо. З урахуванням наявної інформації про предметну галузь та логічну модель даних, можемо сформувати наступні сервіси:

– Product Service: відповідає за управління інформацією про продукти. використовує базу даних, яка зберігає дані про продукти, їхні категорії та властивості; сутності, які належать локальній базі даних, наступні: Product, Category, Product_propety, Property;
– Order Service: керує процесами замовлень, включно з їх створенням, зміною статусу, оплатою, і доставкою; використовує окрему базу даних для збереження інформації про замовлення і взаємодіє з користувачами та кошиками; сутності, які належать локальній базі даних, наступні: Order,

Product_order;

– Basket Service: управління кошиками користувачів, які зберігають товари, додані до кошика, перш ніж буде здійснене замовлення; сервіс зберігає інформацію про кожен кошик та його вміст; сутності, які належать локальній базі даних, наступні: Basket, Product_basket;

– User Service: сервіс для управління даними користувачів, включаючи автентифікацію, реєстрацію та збереження персональних даних користувачів; база даних зберігає лише таблицю User.

Окрім того, в архітектурі мають бути наявні асинхронні канали зв'язку для передачі повідомлень або ж подій. Надавати такі канали зв'язку буде черга повідомлень. На рисунку 2.12 наведена високорівнева мікросервісна архітектура системи електронної комерції, яка буде використана для проведення експериментального дослідження.

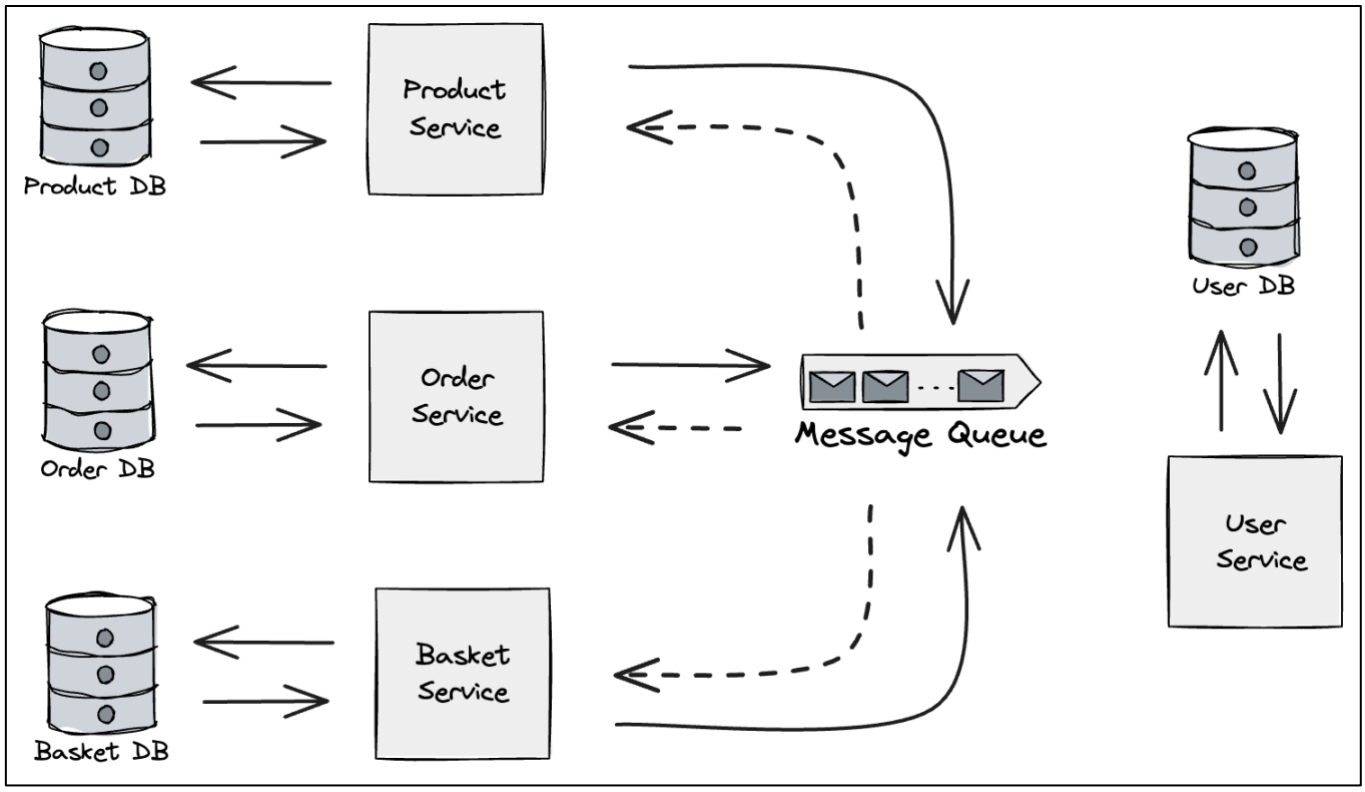

Рисунок 2.12 – Мікросервісна архітектура системи для проведення експериментів (рисунок створено самостійно)

Зробивши декомпозицію загальної схеми бази даних, можемо навести діаграму, яка ілюструє розподіл таблиць відповідно окремих баз даних кожного сервісу та зв’язки всередині кожної схеми (див. рис. 2.13).

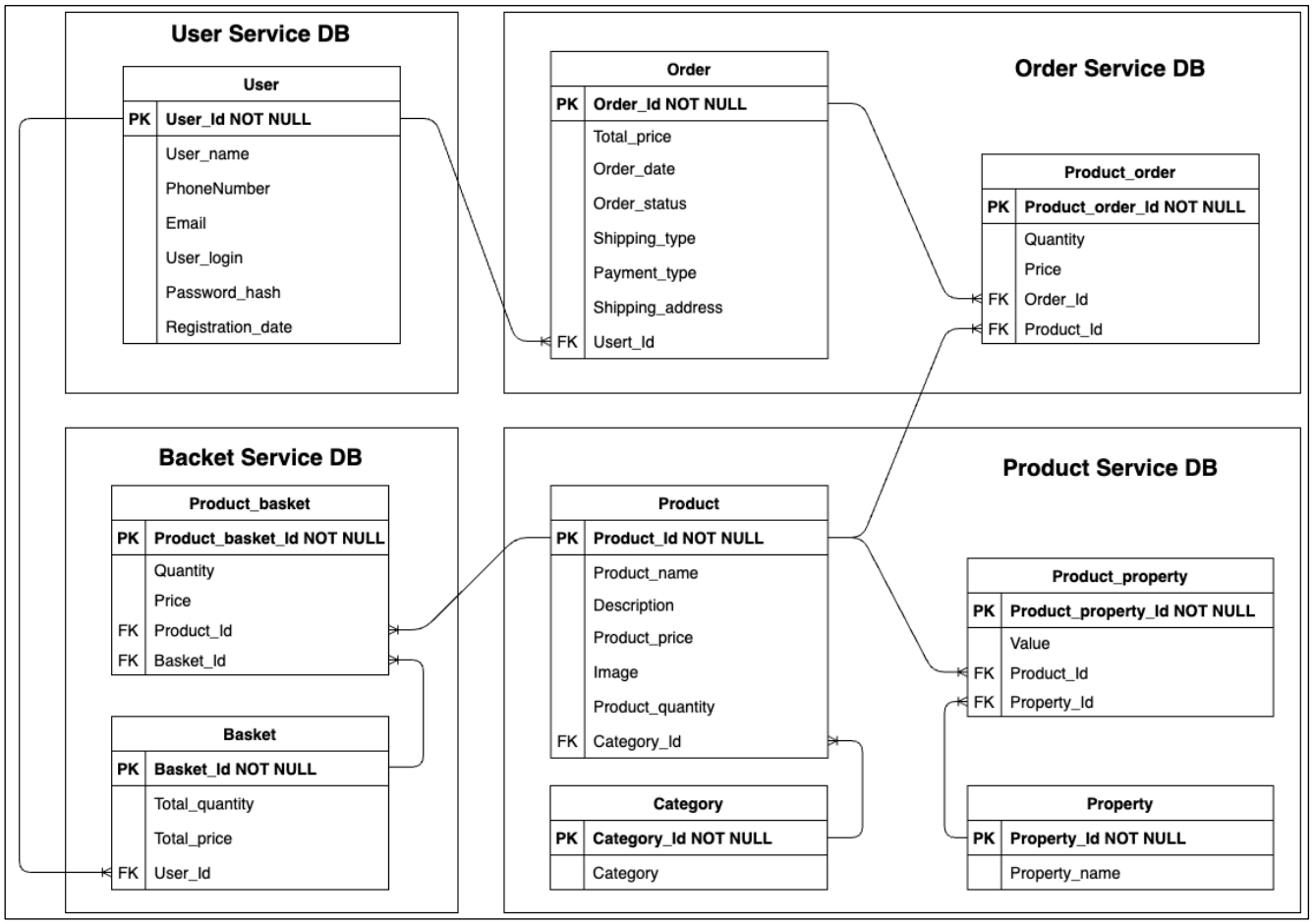

Рисунок 2.13 – Ілюстрація розподілення баз під кожен мікросервіс (рисунок створено самостійно)

Такий підхід до проектування системи не лише підходить для обробки розподілених транзакцій і забезпечує високу ступінь децентралізації та гнучкості для кожного компонента архітектури, але й відповідає принципам DevOps, забезпечуючи швидкі ітерації розгортання та легкість внесення змін.

Розглянемо високорівнево застосування даної архітектури в контексті обраних методів обробки розподілених транзакцій:

– SAGA + Transactional Outbox: кожен мікросервіс під час виконання локальної

транзакції записує події в «транзакційний вихідний ящик», або ж Outbox таблицю в базі, яка потім використовується для публікації подій у загальнодоступну чергу повідомлень;

– SAGA + Event Sourcing: всі зміни в стані сутностей та агрегатів системи виконуються через збереження послідовності подій, які можуть бути відтворені для відновлення стану системи або виконання асинхронних бізнес-операцій.

Обидва підходи SAGA забезпечують високу надійність обробки розподілених транзакцій у складі мікросервісної архітектури і можуть бути впроваджені залежно від специфічних вимог до консистентності даних і потреб бізнес-процесів.

Такий підхід вимагає ретельного планування і проектування взаємодії між сервісами, включаючи контракти API, моделювання черг повідомлень та механізмів відновлення після збоїв. На етапі проектування також важливо передбачити стратегії моніторингу та логування для забезпечення збору статистики відносно досліджувальних метрик.

Подальші кроки в реалізації системи на основі отриманої мікросервісної архітектури передбачають детальне специфікування кожного сервісу, вибір технологій і платформ, документування протоколів комунікації, а саме конкретних кроків виконання хореографічної бізнес-транзакції та конкретних компенсаційних транзакцій, а також планування інфраструктури, необхідної для підтримки високодоступної, масштабованої та відмовостійкої системи на основі обраних технологій та фреймворків.

2.2.5 Проектування запитів для проведення експериментів над транзакціями

Беручи за основу існуючий аналіз предметної галузі, логічну модель даних, спроектовану архітектуру програмної системи, можемо приступити до проектування запитів, які будуть виконані на системі під час проведення експериментів.

На основі аналізу бізнес-правил предметної області та найбільш поширених

варіантів використання, можемо зупинитись на наступних бізнес-транзакціях:

а) транзакція «Додавання товару у кошик»:

1) Basket: отримує від клієнта запит на додавання товару до кошика, оновлює дані у власній БД, після чого відправляє подію ProductAddedToBasketEvent про необхідність валідації ціни продукту у Catalog;
2) Catalog: отримує подію, перевіряє ціну продукту, надсилає відповідь ProductBasketAdditionValidatedEvent, якщо все добре, ProductBasketPriceHasChangedEvent, якщо змінилась ціна, або ProductBasketAdditionValidationFailedEvent, якщо товару взагалі не існує назад до Basket;
3) Basket: підтверджує додавання товару в кошик користувача з врахуванням отриманих даних, або ж компенсує зміни шляхом оновлення вартості товарів в кошику відповідно до отриманих даних;

б) транзакція «Оформлення замовлення»:

1) Ordering: на отримує запит створення замовлення з деталями доставки та оплати; зберігає цю інформацію, надсилає OrderPlacementRequestedEvent до Basket Service та очікує деталі кошика;
2) Basket Service: отримує подію, відправляє товари з кошика у вигляді повідомлення BasketCheckedOutEvent до Ordering;
3) Ordering: отримує дані замовлення з події; змінює стан замовлення на PENDING; зберігає деталі замовлених продуктів, надсилає ідентифікатор замовлення у вигляді OrderPlacedEvent назад до Basket;
4) Basket Service: очищає кошик користувача після отримання ідентифікатора замовлення;

в) транзакція «Підтвердження оплати»:

1) Ordering: підтверджує оплату, змінює стан замовлення на PENDING_PAYMENT_CONFIRMED і надсилає повідомлення для

оновлення запасів до Catalog;

2) Catalog: отримує подію, зменшує кількість наявних продуктів у базі даних, надсилає повідомлення ProductQuantityReservedEvent;

3) Ordering: оновлює статус замовлення на APPROVED після отримання підтвердження про зменшення запасів;

г) транзакція «Скасування замовлення»:

1) Ordering: отримує запит на скасування замовлення, змінює статус на CANCELLATION_REQUESTED, надсилає OrderCancellationRequestedEvent до Catalog із списком товарів для відновлення їх наявності на складі;

2) Catalog: отримує подію, відновлює запаси продуктів на складі (скасовує резервування для клієнта), надсилає подію ProductQuantityRestoredEvent до Ordering;

3) Ordering: отримує подію та змінює статус замовлення на CANCELED.

Під час кожного кроку в SAGA транзакціях, стан системи оновлюється послідовно, і кожен мікросервіс відповідає за свою частину логіки в межах транзакції. За допомогою SAGA, система забезпечує консистентність даних навіть в умовах розподілених операцій, а Message Queue дозволяє реалізувати асинхронний обмін повідомленнями між сервісами, що забезпечує високу доступність і відмовостійкість системи.

Кожен з цих кроків включає взаємодію з базою даних відповідного мікросервісу, де зберігається стан об'єктів, які взаємодіють (наприклад, кошик, замовлення, товари), а також обмін подіями через Message Queue.

Для забезпечення коректної логіки транзакцій та консистетності даних в системі взагалом, важливо також розробити механізми компенсації для відкату транзакцій у випадку помилок, що дозволить повернути систему до консистентного стану без втрати даних. Реалізація як і основних транзакцій, так і компенсаційних, детально буде розглянути в наступному розділі.

3 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

3.1 Вибір технологій для програмної реалізації

Перш ніж приступити до розробки програмної системи та реалізації розподілених транзакцій, варто вирішитись та зупинитись на конкретних технологіях та інструментах, необхідних для виконання цієї роботи та створення фундаменту для проведення експериментів.

При виборі технологій для програмної реалізації мікросервісної архітектури важливо зосередитись на надійності, гнучкості, продуктивності та зручності інтеграції з іншими компонентами системи. Вибрані технології повинні відповідати сучасним вимогам і стандартам розробки, а також підтримувати концепції розподіленої обробки та високої доступності.

Java як основна мова програмування обирається завдяки своїй зрілості, стабільності та широкій підтримці в індустрії. Java надає багатий набір бібліотек і фреймворків, які спрощують розробку складних мікросервісних додатків [28], а також має велику спільноту, що може надати підтримку у вирішенні проблем.

Spring Boot та Spring Cloud вибрані для розробки мікросервісів через їхню простоту у створенні незалежних самодостатніх сервісів, що можуть ефективно спілкуватись у межах розподіленої системи. Ці фреймворки пропонують зручні шаблони конфігурації, вбудовані механізми виявлення та врівноваження навантаження, а також забезпечують інтеграцію з хмарними платформами [29]. Кожен із спроектованих раніше мікросервісів буде реалізований як окремий Spring Boot сервіс. Spring Cloud ж надаватиме інфраструктурні та комунікаційні інструменти, з його допомогою буде можливість швидко розгорнути конфігураційний сервіс для збереження налаштувань та конфіденційних даних, а також Discovery сервіс для налаштування і збереження IP-адрес наявних в архітектурі сервісів для забезпечення міжсервісної комунікації.

Apache Kafka використовується як механізм обміну подіями та

повідомленнями завдяки його високій пропускній здатності, надійності та здатності до масштабування. Kafka дозволяє забезпечити стійкість обробки повідомлень і є відмінним рішенням для побудови надійних асинхронних комунікаційних каналів у мікросервісних архітектурах [30]. Окрім того, для коректної роботи Kafka кластеру варто використати Zookeeper, який є інструмент управління вузлами брокера.

Eventuate Tram [31] обраний як фреймворк для реалізації патерну Transactional Outbox. Він з легкістю інтегрується з вже обраними Apache Kafka та MySQL та надає можливості для реалізації подій і легкої їх публікації до інших сервісів. Підтримка Spring Boot дає змогу легкої інтеграції та конфігурації. Також він надає вбудований CDC (Change Data Capture) сервіс, який буде слідкувати за таблицею з повідомленнями та публікувати їх до брокера. Детальну архітектуру та взаємодію буде розглянуто в наступних підрозділах.

Eventuate Local [32] використовується як Event Store: комбінація реляційної СКБД та Message брокера. В якості меседж брокера Eventuate використовує вже обраний Apache Kafka, в якості реляційної СКБД використовує MySQL. В реляційній БД фреймфорк буде зберігати інформацію про події та сутності, а за допомогою Apache Kafka він буде відправляти ці події споживачам. Ця технологія є ключовою для реалізації SAGA + Event Sourcing підходу, вибір якої дозволить з легкістю створювати та управляти послідовностями подій, які є основою для розподілених транзакцій в SAGA. Це також забезпечує збереження історії всіх змін у системі, що є важливим для забезпечення відмовостійкості та аудиту.

Docker та, власне, Docker Desktop обрані як зручний та доступний інструмент для контейнеризації та розгортання кластеру пов’язаних сервісів. З його допомогою буде легко налаштувати та запускати мікросервіси під кожен патерн, БД під кожен патерн та необхідні інфраструктурні компоненти [33].

Окрім того, потрібно мати певний механізм, який дозволить відстежувати централізоване логування, адже кожен крок SAGA може виконуватись в різних мікросервісах, і важливо бачити етапи виконання в правильному порядку. Для цього

буде використано так званий ELK стек технологій: ElasticSearch, Logstash та Kibana [34]. Logstash буде вичитувати логи з файлів на мікросервісах та відправляти їх до ElasticSearch, звідки вони будуть потрапляти до Kibana – інструменту перегляду логів.

Вибір цих технологій для реалізації програмної системи заснований на балансі між продуктивністю та здатністю до масштабування. Вони спільно створюють міцний фундамент для побудови високодоступної та гнучкої мікросервісної системи, що може адаптуватися до змінних вимог бізнесу та технологічного середовища.

3.2 Розробка фізичної моделі даних

Перш ніж розробляти та впроваджувати мікросервіси та SAGA, варто створити схеми баз даних та всі необхідні сутності відповідно до проведених раніше аналізів. Нижче наведено скрипти створення таблиць для MySQL бази даних:

```
CREATE TABLE IF NOT EXISTS User (
    User_Id INT NOT NULL PRIMARY KEY,
    User_name VARCHAR(255) NOT NULL,
    PhoneNumber VARCHAR(20),
    Email VARCHAR(255),
    User_login VARCHAR(255) NOT NULL,
    Password_hash VARCHAR(255) NOT NULL,
    Registration_date DATE NOT NULL
);

CREATE TABLE IF NOT EXISTS Basket (
    Basket_Id INT NOT NULL PRIMARY KEY,
    Total_quantity INT,
    Total_price DECIMAL,
    User_Id INT NOT NULL
);

CREATE TABLE IF NOT EXISTS Product_Basket (
    Product_basket_Id INT NOT NULL PRIMARY KEY,
    Quantity INT NOT NULL,
    Price DECIMAL NOT NULL,
    Product_Id INT NOT NULL,
    Basket_Id INT NOT NULL,
    FOREIGN KEY (Basket_Id) REFERENCES Basket(Basket_Id)
);
```

```
CREATE TABLE IF NOT EXISTS Category (
    Category_Id INT NOT NULL PRIMARY KEY,
    Category_name VARCHAR(255) NOT NULL
);

CREATE TABLE IF NOT EXISTS Product (
    Product_Id INT NOT NULL PRIMARY KEY,
    Product_name VARCHAR(255) NOT NULL,
    Description TEXT,
    Product_price DECIMAL NOT NULL,
    Image TEXT,
    Product_quantity INT NOT NULL,
    Category_Id INT,
    FOREIGN KEY (Category_Id) REFERENCES Category(Category_Id)
);

CREATE TABLE IF NOT EXISTS Property (
    Property_Id INT NOT NULL PRIMARY KEY,
    Property_name VARCHAR(255) NOT NULL
);

CREATE TABLE IF NOT EXISTS Product_Property (
    Product_property_Id INT NOT NULL PRIMARY KEY,
    Value VARCHAR(255) NOT NULL,
    Product_Id INT NOT NULL,
    Property_Id INT NOT NULL,
    FOREIGN KEY (Product_Id) REFERENCES Product(Product_Id),
    FOREIGN KEY (Property_Id) REFERENCES Property(Property_Id)
);

CREATE TABLE IF NOT EXISTS `Order` (
    Order_Id INT NOT NULL PRIMARY KEY,
    Total_price DECIMAL NOT NULL,
    Order_date DATE NOT NULL,
    Order_status VARCHAR(50),
    Shipping_type VARCHAR(50),
    Payment_type VARCHAR(50),
    Shipping_address TEXT,
    User_Id INT NOT NULL
);

CREATE TABLE IF NOT EXISTS Product_Order (
    Product_order_Id INT NOT NULL PRIMARY KEY,
    Quantity INT NOT NULL,
    Price DECIMAL NOT NULL,
    Product_Id INT NOT NULL,
    Order_Id INT NOT NULL,
    FOREIGN KEY (Order_Id) REFERENCES `Order`(Order_Id)
);
```

Слід зазначити, що в даному скрипті зовнішні ключі присутні для таблиць, що знаходяться в рамках однієї схеми однієї бази даних під конкретний сервіс.

Посилальна цілісність тих колонок, для яких вказаний зовнішній ключ на діаграмі, проте в скрипті він відсутній, буде забезпечуватись не на рівні СУБД, а на рівні програмної системи.

Також наведемо скрипти створення таблиць, необхідних для роботи шаблону Transactional Outbox:

```
CREATE TABLE `message` (
  `id` varchar(255) NOT NULL,
  `destination` longtext NOT NULL,
  `headers` longtext CHARACTER SET utf8mb4
      COLLATE utf8mb4_unicode_ci NOT NULL,
  `payload` longtext CHARACTER SET utf8mb4
      COLLATE utf8mb4_unicode_ci NOT NULL,
  `published` smallint DEFAULT '0',
  `message_partition` smallint DEFAULT NULL,
  `creation_time` bigint DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `message_published_idx` (`published`,`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

CREATE TABLE `received_messages` (
  `consumer_id` varchar(255) NOT NULL,
  `message_id` varchar(255) NOT NULL,
  `creation_time` bigint DEFAULT NULL,
  `published` smallint DEFAULT '0',
  PRIMARY KEY (`consumer_id`,`message_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

Прокоментуємо найбільш важливі колонки. Message містить в собі id, destination (назва агрегату, який очікує на цю подію), headers (певна мета-інформація у форматі JSON, яка включає в себе тип події, дата та час публікації тощо), payload (данні необхідні для правильної роботи бізнес-логіки), creation_time (timestamp). При виконанні локальної транзакцій, мікросервіс в рамках однієї ACID транзакцій оновлює потрібні данні та публікує подію до таблиці. Після чого ця подія відслідковується CDC сервісом, який перевіряє бінарні логи транзакцій MySQL та при появі нового повідомлення – записує його в таблицю Received_messages (для уникнення дублювання) та відправляє деталі повідомлення в якості події до брокера повідомлень, наприклад, Apache Kafka. Після чого це повідомлення може бути

отримано відповідним мікросервісом для подальшої обробки та запуску того самого набору дій.

Наведемо скрипти створення таблиць, необхідних для роботи шаблону Event Sourcing:

```
CREATE TABLE `events` (
  `event_id` varchar(255) NOT NULL,
  `event_type` longtext,
  `event_data` longtext CHARACTER SET utf8mb4
      COLLATE utf8mb4_unicode_ci NOT NULL,
  `entity_type` varchar(255) NOT NULL,
  `entity_id` varchar(255) NOT NULL,
  `triggering_event` longtext,
  `metadata` longtext,
  `published` tinyint DEFAULT '0',
  PRIMARY KEY (`event_id`),
  KEY `events_idx` (`entity_type`,`entity_id`,`event_id`),
  KEY `events_published_idx` (`published`,`event_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

CREATE TABLE `entities` (
  `entity_type` varchar(255) NOT NULL,
  `entity_id` varchar(255) NOT NULL,
  `entity_version` longtext NOT NULL,
  PRIMARY KEY (`entity_type`,`entity_id`),
  KEY `entities_idx` (`entity_type`,`entity_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

Event Sourcing, навпаки, має інший, незвичний в стандартному плані, підхід до обробки транзакцій. Перш за все, змінюється підхід до зберігання даних. Будь-яка дія над сутністю тепер представляється у форматі події (Event). Тобто, наприклад, для кошика, який створили і додали в нього один товар, може бути наступний набір подій: BasketCreatedEvent, ProductAddedToBasketEvent. В базі додаються дві таблиці: Events та Entities. Entities містить наступні поля: entity_type, entity_id, entity_version (для Optimistic Locking). Основні поля Events це event_id, event_type, event_data (дані для оновлення стану та передачі між сервісами), entity_type, entity_id. Тобто вся інформація для відтворення стану сутності зберігається у вигляді набору подій, які треба вибрати з бази та «переграти» у порядку, в якому вони виконувались. Відмовостійкість та консистентність між оновленням даних та подіями виконується

завдяки тому, що тепер подія – це основний і головний інструмент як зберігання даних, так і передачі їх у вигляді повідомлень іншим сервісам.

3.3 Розробка логіки обробки розподілених транзакцій за патерном SAGA

Розглянемо деталі реалізації та механізмів роботи обраних патернів в програмній системі. Перш за все, варто зазначити спільну риску двох патернів – CDC сервіс, за який було згадано раніше. Розглянемо його призначення та принцип роботи.

Сервіс призначений для того, аби вичитувати повідомлення, або події, з відповідних таблиць в БД та відправляти їх до брокера Apache Kafka. Тобто мікросервіси напряму не взаємодіють з Apache Kafka в контексті публікації в неї повідомлень, а лише отримують повідомлення з неї. За доставку повідомлень до Kafka та правильне оновлення метаданих з приводу того які повідомлення були опубліковані, а які ні, відповідає саме CDC сервіс (див. рис. 3.1).

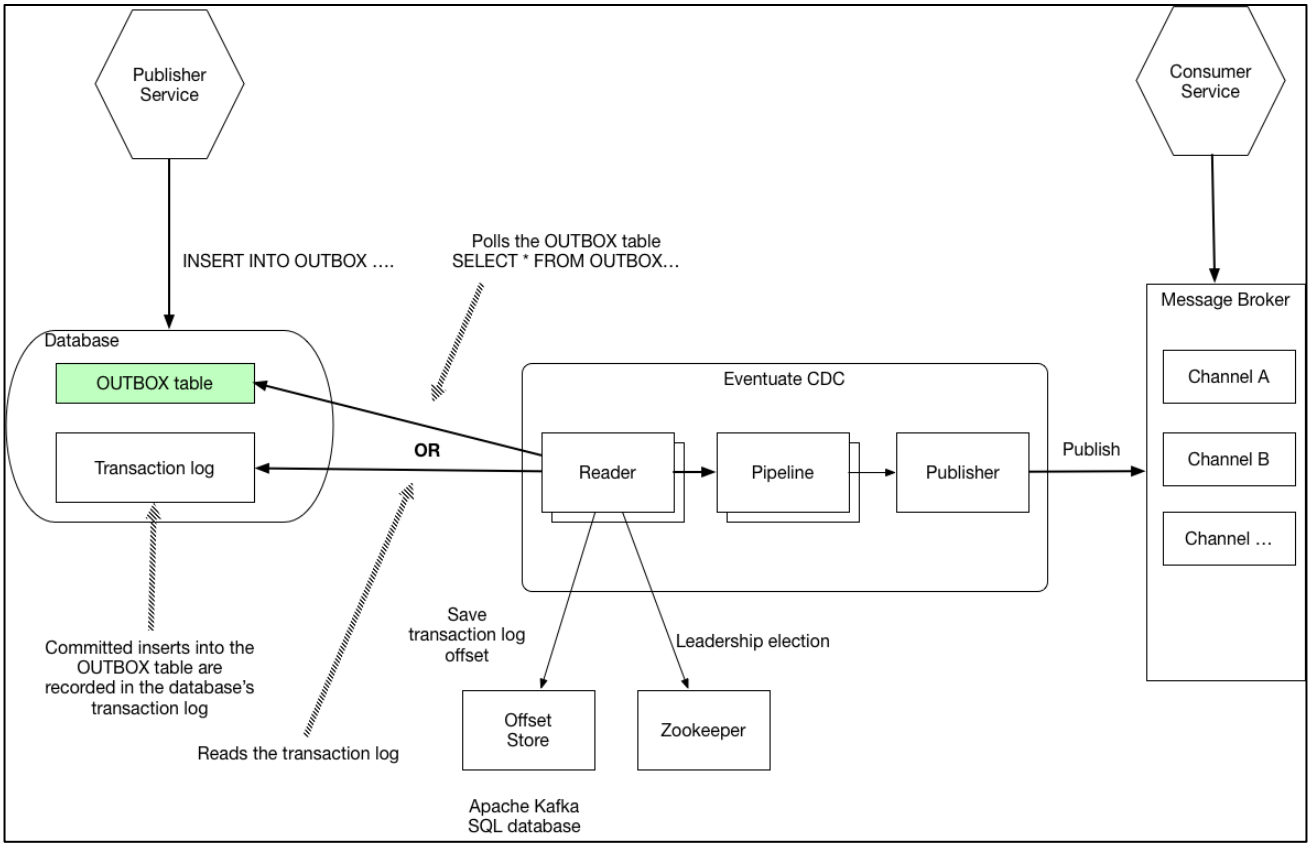

Рисунок 3.1 – Архітектура використання CDC сервісу (за даними [35])

Прокоментуємо ключові компоненти на наведеній діаграмі:

– Publisher service – сервіс, який додає повідомлення/подію у таблицю OUTBOX, якою може бути або Message, або Events, в залежності від патерну, що використовується;

– Consumer service – сервіс, який отримує повідомлення/подію;

– таблиця OUTBOX – узагальнена назва, таблиця Message або Events, в залежності від патерну, що використовується;

– Eventuate CDC-service – сервіс, який публікує повідомлення/події до брокера повідомлень;

– Reader – компонент CDC, який отримує події доданих повідомлень, використовуючи або «хвіст» журналу транзакцій (останні записи), або зчитування за допомогою оператора SELECT;

– Pipeline – викликається зчитувачем для публікації повідомлення;

– Offset Store – використовується зчитувачем для запису поточного зсуву журналу транзакцій, наприклад, MySQL binlog;

– Publisher – використовується конвеєром для публікації повідомлення/події до брокера повідомлень;

– Zookeeper – використовується зчитувачем для обрання лідера при кластеризації.

Даний сервіс грає ключові роль у механізмі виконання двох шаблонів. В наступних підрозділах розглянемо відмінності між ними та деталі впровадження в програмну систему.

3.3.1 Впровадження патерну Transactional Outbox

Розглянемо повний потік виконання транзакції «Додавання товару до кошика», реалізованої під шаблон Transactional Outbox.

Перш за все, запит на додавання товару до кошика надходить та оброблюється BasketServiceController, де клієнту відповідь повертається відразу:

```
@PostMapping(value = "/baskets/{basketId}/products")
@Operation(summary = "[Add Product to Basket SAGA] starting point")
public BasketDtoResponse addProductToBasket(
    @PathVariable Long basketId,
    @RequestBody AddProductToBasketRequest addProductToBasketRequest) {
    logStartTime(logger, ADD_PRODUCT_TO_BASKET_PREFIX, basketId);
    Basket basket = basketService.addProductToBasket(
            basketId,
            addProductToBasketRequest.getProductId(),
            addProductToBasketRequest.getQuantity(),
            addProductToBasketRequest.getPricePerUnit()
    );
    return convertToBasketDtoResponse(basket);
};
```

Контролер викликає сервіс, а саме addProductToBasket метод, реалізація якого наведена далі:

```
@Transactional
public Basket addProductToBasket(Long basketId,
                                 String productId,
                                 Long quantity,
                                 Money pricePerUnit) {
    log(logger, "{} Adding product {} for basket {}",
            ADD_PRODUCT_TO_BASKET_PREFIX, productId, basketId);

    Basket basket = basketRepository.findById(basketId).orElseThrow();

    addProductEntryToBasket(basket, productId, quantity, pricePerUnit);
    publishProductAddedToBasketEvent(basketId, productId, pricePerUnit);

    return basket;
};

private void publishProductAddedToBasketEvent(Long basketId,
                                              String productId,
                                              Money pricePerUnit) {
    ProductAddedToBasketEvent event =
        new ProductAddedToBasketEvent(productId, pricePerUnit);

    log(logger,
        "{} Product {} added to basket {} successfully, publishing {}",
         ADD_PRODUCT_TO_BASKET_PREFIX, productId,
         basketId, event.getClass().getSimpleName());

    domainEventPublisher.publish(Basket.class, basketId,
                                 Collections.singletonList(event));
}
```

Як видно із коду, addProductEntryToBasket виконує власне оновлення сутності в БД, додаючи товар до кошика, а domainEventPublisher.publish() публікує ProductAddedToBasketEvent до відповідної таблиці повідомлень.

Тепер розглянемо Product Service, а саме місце в коді, яке слухає відповідну подію:

```
public DomainEventHandlers domainEventHandlers() {
    return DomainEventHandlersBuilder
        .forAggregateType("ua.nure.sagaresearch.baskets.domain.Basket")
                .onEvent(ProductAddedToBasketEvent.class,
                        this::productAddedToBasketEventHandler)
        .andForAggregateType("ua.nure.sagaresearch.orders.domain.Order")
                .onEvent(OrderPaymentConfirmedEvent.class,
                        this::handleOrderPaymentConfirmedEvent)
                .onEvent(OrderCancellationRequestedEvent.class,
                        this::handleOrderCancellationRequestedEvent)
            .build();
}

private void productAddedToBasketEventHandler(
   DomainEventEnvelope<ProductAddedToBasketEvent> domainEventEnvelope) {
   ProductAddedToBasketEvent event = domainEventEnvelope.getEvent();
   long basketId = Long.parseLong(domainEventEnvelope.getAggregateId());
   log(logger, "{} Handling {} for basket {}",
       ADD_PRODUCT_TO_BASKET_PREFIX,
       event.getClass().getSimpleName(), basketId);

   productService.validateProductAddedToBasket(
           event.getProductId(),
           basketId,
           event.getPricePerUnit()
    );
}
```

Клас DomainEventHandlers надає змогу вказати обробників подій для тих чи інших агрегатів. Як видно, на подію ProductAddedToBasketEvent реагує відповідний EventHandler метод, в якому виконується зчитування події, ідентифікатора кошику, та ідентифікатора товару. Далі виконання переходить до productService, який валідує ціну товару:

```
public void validateProductAddedToBasket(String productId,
                                          Long basketId,
```

```
                                                Money pricePerUnit) {
    productRepository.findById(productId)
            .map(product -> {
                Money actualProductPrice = product.getProductPrice();
                if (Objects.equals(pricePerUnit, actualProductPrice)) {
                    return new
                        ProductBasketAdditionValidatedEvent(basketId);
                }
                return new
                  ProductBasketPriceHasChangedEvent(basketId,
                                                    actualProductPrice);
            })
            .ifPresentOrElse(
               event ->  publishProductAddedToBasketValidationResult(
                            event, productId),
               () -> publishProductAddedToBasketValidationResult(
                            new ProductBasketAdditionValidationFailedEvent(
                               basketId), productId));
}

private void publishProductAddedToBasketValidationResult(
    ProductEvent event, String productId) {
    log(logger, "{} Finished validation, publishing {} for id {}",
        ADD_PRODUCT_TO_BASKET_PREFIX, event.getClass().getSimpleName(),
        productId);
    domainEventPublisher.publish(Product.class, productId,
                                 Collections.singletonList(event));
}
```

В цьому коді відбувається валідація вартості товару, та, відповідно, від результати валідації публікуються події про успішну валідацію ціни, неуспішну та про відсутність товару як такого, які оброблюються фінально Basket сервісом, що завершує транзакцію:

```
public void handleProductBasketAdditionValidatedEvent(
  DomainEventEnvelope<ProductBasketAdditionValidatedEvent> envelope) {
    var event = envelope.getEvent();
    Long basketId = event.getBasketId();
    log(logger,
            "{} Received {}, transaction completed successfully for
                productId {} and basketId {}",
            ADD_PRODUCT_TO_BASKET_PREFIX,
            event.getClass().getSimpleName(), envelope.getAggregateId(),
            basketId);
    logEndTime(logger, ADD_PRODUCT_TO_BASKET_PREFIX, basketId);
}

private void handleProductBasketPriceHasChangedEvent(
```

```
    DomainEventEnvelope<ProductBasketPriceHasChangedEvent> envelope) {
      var event = envelope.getEvent();
      String productId = envelope.getAggregateId();
      Long basketId = event.getBasketId();
      Money actualPricePerUnit = event.getActualPricePerUnit();

      log(logger,
       "{} Received {}, updating price of product {} in basket {} to {}$",
        ADD_PRODUCT_TO_BASKET_PREFIX, event.getClass().getSimpleName(),
         productId, basketId,
         actualPricePerUnit.getAmount());

         basketService.actualizeProductEntryPrice(basketId, productId,
                                                   actualPricePerUnit);
}
```

Як видно, фінальні обробники подій ведуть себе по-різному: якщо валідація пройшла успішно, ця подія просто логується і ланцюжок SAGA завершується. Якщо валідація не пройшла успішно: вартість товару оновлюється відповідно до нових даних, які передались із Product сервісу.

Запустимо транзакцію та перевіримо її виконання, проаналізувавши логи в сервісі Kibana (див. рис. 3.2).

	message	path
15:15:51.504	[OUTBOX Add Product to Basket SAGA] Start time: 7624368366089 | ID: 101	/usr/share/logstash/logs/basket-service.log
15:15:51.507	[OUTBOX Add Product to Basket SAGA] Adding product 09e87578-9988-4e53-aa05-bb0f34460f07 for basket 101	/usr/share/logstash/logs/basket-service.log
15:15:51.530	[OUTBOX Add Product to Basket SAGA] Product 09e87578-9988-4e53-aa05-bb0f34460f07 added to basket 101 successfully, publishing ProductAddedToBasketEvent	/usr/share/logstash/logs/basket-service.log
15:15:51.754	[OUTBOX Add Product to Basket SAGA] Handling ProductAddedToBasketEvent for basket 101	/usr/share/logstash/logs/product-service.log
15:15:51.817	[OUTBOX Add Product to Basket SAGA] Finished validation, publishing ProductBasketAdditionValidatedEvent for productId 09e87578-9988-4e53-aa05-bb0f34460f07	/usr/share/logstash/logs/product-service.log
15:15:51.866	[OUTBOX Add Product to Basket SAGA] Received ProductBasketAdditionValidatedEvent, transaction completed successfully for productId 09e87578-9988-4e53-aa05-bb0f34460f07 and basketId 101	/usr/share/logstash/logs/basket-service.log
15:15:51.867	[OUTBOX Add Product to Basket SAGA] End time: 7624731464464 | ID: 101	/usr/share/logstash/logs/basket-service.log

Рисунок 3.2 – Логування виконання транзакції «Додавання товару у кошик» (рисунок створено самостійно)

Як видно із наявних логів, вони були зібрані із двох розташувань: basket-service.log та product-service.log, що відповідає дійсності. Окрім того, централізоване логування показує правильний порядок виконання всіх кроків.

3.3.2 Впровадження патерну Event Sourcing

Розглянемо повний потік виконання транзакції «Оформлення замовлення», реалізованої під шаблон Event Sourcing.

Проте перед тим, як перейти до деталей реалізації, розглянемо високорівневі відмінності цього шаблону. Традиційним способом збереження сутності є збереження її поточного стану. Event Sourcing використовує радикально інший, орієнтований на події підхід до збереження. Бізнес-об'єкт зберігається шляхом зберігання послідовності подій, що змінюють його стан. Щоразу, коли стан об'єкта змінюється, до послідовності подій додається нова подія. Оскільки це одна операція, вона є атомарною за своєю суттю. Поточний стан об'єкта відновлюється шляхом відтворення його подій.

Щоб побачити, як працює Event Sourcing, розглянемо сутність Order. Традиційно, кожне замовлення зіставляється з рядком у таблиці Order разом з рядками в іншій таблиці, наприклад, таблиці Product_Order (що було реалізовано при використанні Transactional Outbox). Але при Event Sourcing Order-сервіс маніпулює замовленням, зберігаючи події, що змінюють його стан: Created, Approved, Shipped, Cancelled (див. рис. 3.3). Кожна подія міститиме достатньо даних, щоб відновити стан замовлення.

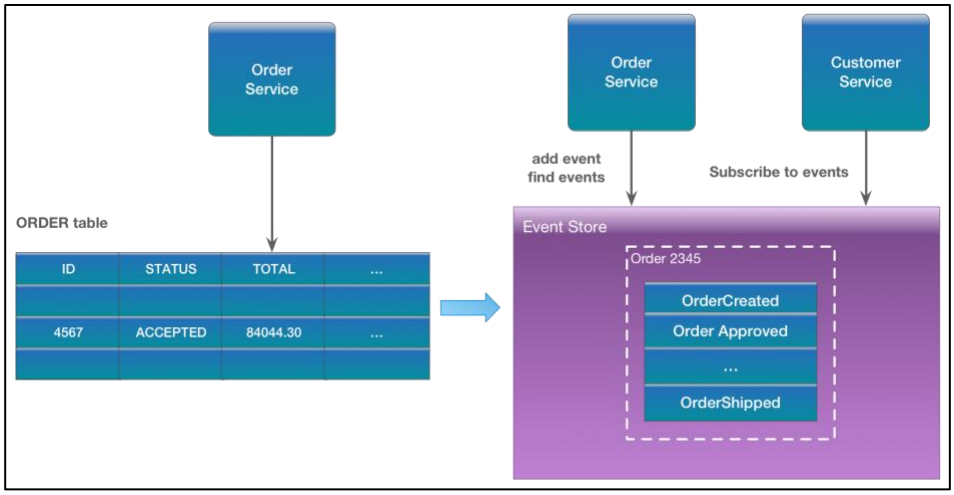

Рисунок 3.3 – Порівняння традиційного способу зберігання та сховища подій (за даними [36])

Події зберігаються в сховищі подій. Сховище подій не тільки діє як база даних подій, воно також поводиться як брокер повідомлень. Він надає API, який дозволяє сервісам підписуватися на події. Кожна подія, яка зберігається у сховищі подій, доставляється сховищем подій усім зацікавленим підписникам. Сховище подій є основою архітектури мікросервісів, керованих подіями.

У цій архітектурі запити на оновлення сутності (або зовнішній HTTP-запит, або подія, опублікована іншим сервісом) обробляються шляхом отримання подій сутності зі сховища подій, реконструкції поточного стану сутності, оновлення сутності та збереження нових подій. На рисунку 3.4 наведена високорівнева ілюстрація процесу обробки запиту на оновлення замовлення.

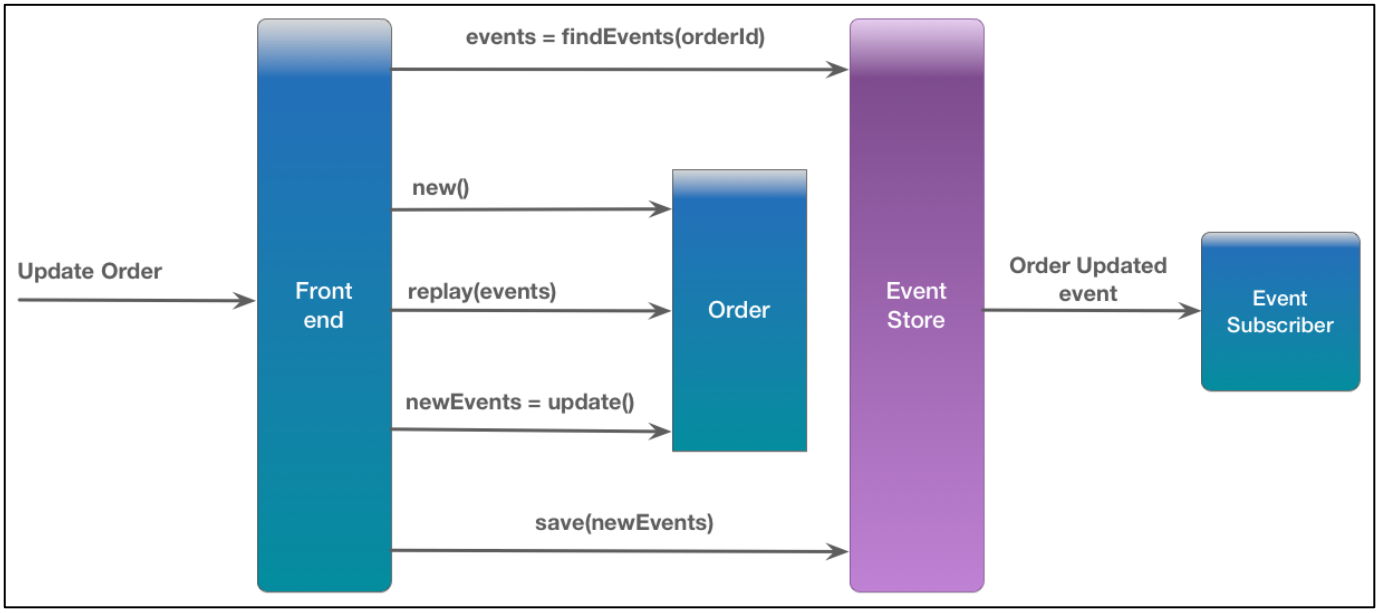

Рисунок 3.4 – Процес обробки запиту при використанні Event Sourcing (за даними [36])

Перейдемо до розгляду деталей реалізації транзакції «Оформлення замовлення». Перш за все, запит на оформлення замовлення традиційно оброблює контролер:

```
@PostMapping(value = "/orders")
@Operation(summary = "[Place Order SAGA] starting point")
```

```
public CreateOrderResponse placeOrder(
        @RequestBody CreateOrderRequest createOrderRequest) {
    long startTime = System.nanoTime();
    EntityWithIdAndVersion<Order> entity =
       sourcingOrderService.createOrder(
            createOrderRequest.getBasketId(),
            createOrderRequest.getShippingType(),
            createOrderRequest.getPaymentType(),
            createOrderRequest.getShippingAddress());
    String entityId = entity.getEntityId();
    logStartTime(LOGGER, EVENT_SOURCING_PLACE_ORDER_PREFIX, startTime,
                 entityId);
    return new CreateOrderResponse(entityId);
}
```

Подивимось як реалізований метод createOrder() в sourcingOrderService:

```
private final AggregateRepository<Order, OrderCommand> orderRepository;

public EntityWithIdAndVersion<Order> createOrder(String basketId,
                                                  String shippingType,
                                                  String paymentType,
                                                  String shippingAddress) {
    return orderRepository.save(new CreateOrderCommand(
                basketId, shippingType, paymentType, shippingAddress));
}
```

Як видно, певний репозиторій агрегатів виконує метод save(), де на вхід передається певна команда CreateOrderCommand. Це перша відмінність між традиційним підходом: ми зберігаємо не самі дані, а певні команди. Розглянемо наступне місце в коді, яке стане обробником цієї команди:

```
public List<Event> process(CreateOrderCommand cmd) {
    var event = new SourcingOrderPlacementRequestedEvent(
            cmd.getBasketId(),
            cmd.getShippingType(),
            cmd.getPaymentType(),
            cmd.getShippingAddress());
    logAggregateProcessMethod(LOGGER, this.getClass(),
            cmd, EVENT_SOURCING_PLACE_ORDER_PREFIX, event);
    return EventUtil.events(event);
}
```

Метод process розташований в класі сутності Order. Він буде викликаний

фреймоворком через механізм Java Reflection API тоді, коли відповідна команда, які він приймає на вхід, буде збережена, що було виконано попереднім кодом. В цьому методі ми створюємо відповідну подію про те, що був прийнятий запит на створення замовлення, та повертаємо із методу його у вигляді списку з одним елементом. Ця подія буде направлена до сервісів, які її очікують, але також ці подія буде оброблена самим же об'єктом Order, що буде виконано через виклик спеціального методу apply(), реалізація якого наведена нижче:

```
public void apply(SourcingOrderPlacementRequestedEvent event) {
    this.state = OrderState.PENDING;
    this.orderDetails = new OrderDetails(
            event.getBasketId(),
            event.getShippingType(),
            event.getPaymentType(),
            event.getShippingAddress());
    this.totalPrice = Money.ZERO;
    this.creationDate = LocalDate.now();
    this.productEntries = new HashMap<>();
}
```

Як видно, це перше місце, де ми якимось чином зберігаємо дані, або ж «відновлюємо» їх, змінюючи стан об'єкту за рахунок інформації, наявної в події, що прийшла на вхід. Тут ми створюємо початкове пусте замовлення. Розглянемо місце в коді на стороні BasketService, яке отримує на вхід цю ж подію:

```
@EventHandlerMethod
public CompletableFuture<EntityWithIdAndVersion<Basket>>
    handleOrderPlacementRequested(
        EventHandlerContext<SourcingOrderPlacementRequestedEvent> ctx) {
    SourcingOrderPlacementRequestedEvent event = ctx.getEvent();
    String basketId = event.getBasketId();
    String orderId = ctx.getEntityId();

    return ctx.update(Basket.class, basketId,
                      new CheckoutBasketCommand(orderId));
}
```

Це event handler метод, який доволі схожий на те, що ми використовували в патерні Transactional Outbox. Проте, по аналогії, ми оновлюємо контекст шляхом

публікації команди CheckoutBasketCommand, передаючи ідентифікатор як кошика, так і замовлення. Розглянемо як оброблюється ця команда:

```
public List<Event> process(CheckoutBasketCommand cmd) {
    String orderId = cmd.getOrderId();
    var event = new SourcingBasketCheckedOutEvent(orderId, totalPrice,
                                                  productEntries);
    logAggregateProcessMethod(LOGGER, this.getClass(), cmd,
                              EVENT_SOURCING_PLACE_ORDER_PREFIX, event);
    return EventUtil.events(event);
}

public void apply(SourcingBasketCheckedOutEvent event) {

}
```

В методі process ми створюємо подію, в яку передаємо ідентифікатор замовлення, вартість товарів в кошику та всі товари з кошику у вигляді змінної productEntries. Варто зауважити, що хоч метод apply і присутній, його тіло пусте. Це зумовлене тим, що на обробці даної події ніяких змін до самої сутності Basket оброблювати не треба, виконана лише передача даних з кошика до замовлення. Розглянемо як це оброблюється на стороні Order service, проаналізувавши код методів eventHandler, process та apply:

```
@EventHandlerMethod
public CompletableFuture<?> handleBasketCheckedOut(
        EventHandlerContext<SourcingBasketCheckedOutEvent> ctx) {
    SourcingBasketCheckedOutEvent event = ctx.getEvent();
    Money totalPrice = event.getTotalPrice();
    Map<String, ProductBasketEntry> productEntries =
        event.getProductEntries();
    String basketId = ctx.getEntityId();
    String orderId = event.getOrderId();

    return ctx.update(Order.class, orderId,
            new FillOrderWithProductEntriesCommand(
                    basketId, totalPrice, productEntries));
}

public List<Event> process(FillOrderWithProductEntriesCommand cmd) {
    var event = new SourcingOrderPlacedEvent(
            cmd.getBasketId(),
            cmd.getTotalPrice(),
```

```
            cmd.getProductEntries()
    );
    logAggregateProcessMethod(LOGGER, this.getClass(),
            cmd, EVENT_SOURCING_PLACE_ORDER_PREFIX, event);
    return EventUtil.events(event);
}

public void apply(SourcingOrderPlacedEvent event) {
    this.productEntries =
        convertToProductOrderEntries(event.getProductEntries());
    this.totalPrice = event.getTotalPrice();
}
```

Команда створює іншу команду на наповнення замовлення елементами з кошику, що виконується в методі apply. Подія про те, що замовлення було створене та наповнене, відправляється до Basket service. Розглянемо останній крок в ланцюжку, а саме обробники подій та команд на стороні Basket Service, які виконують очищення кошику після успішного створення замовлення:

```
@EventHandlerMethod
public CompletableFuture<EntityWithIdAndVersion<Basket>>
    handleOrderPlacement(
        EventHandlerContext<SourcingOrderPlacedEvent> ctx) {
    SourcingOrderPlacedEvent event = ctx.getEvent();
    String basketId = event.getBasketId();
    String orderId = ctx.getEntityId();

    return ctx.update(Basket.class, basketId,
                      new ClearBasketCommand(orderId));
}

public List<Event> process(ClearBasketCommand cmd) {
    String orderId = cmd.getOrderId();
    SourcingBasketClearedEvent event =
            new SourcingBasketClearedEvent(orderId);
    logAggregateProcessMethod(LOGGER, this.getClass(),
            cmd, EVENT_SOURCING_PLACE_ORDER_PREFIX, event);
    logEndTime(LOGGER, EVENT_SOURCING_PLACE_ORDER_PREFIX, orderId);
    return EventUtil.events(event);
}

public void apply(SourcingBasketClearedEvent event) {
    this.totalQuantity = 0L;
    this.totalPrice = Money.ZERO;
    this.productEntries = new HashMap<>();
}
```

Таким чином була повністю реалізована транзакція «Оформлення замовлення» з використанням методу Event Sourcing. Відповідні логи наведені на рисунку 3.5.

message	path
[Place Order SAGA] Event Sourcing. Aggregate Order, processing CreateOrderCommand, propagating SourcingOrderPlacementRequestedEvent	/usr/share/logstash/logs/order-service.log
[Place Order SAGA] Event Sourcing. Start time: 11634521667418 | ID: 0000018fd92f0df8-0242ac14000f0000	/usr/share/logstash/logs/order-service.log
[Place Order SAGA] Event Sourcing. Aggregate Basket, processing CheckoutBasketCommand, propagating SourcingBasketCheckedOutEvent	/usr/share/logstash/logs/basket-service.log
[Place Order SAGA] Event Sourcing. Aggregate Order, processing FillOrderWithProductEntriesCommand, propagating SourcingOrderPlacedEvent	/usr/share/logstash/logs/order-service.log
[Place Order SAGA] Event Sourcing. Aggregate Basket, processing ClearBasketCommand, propagating SourcingBasketClearedEvent	/usr/share/logstash/logs/basket-service.log
[Place Order SAGA] Event Sourcing. End time: 11634635859835 | ID: 0000018fd92f0df8-0242ac14000f0000	/usr/share/logstash/logs/basket-service.log

Рисунок 3.5 – Логування виконання транзакції «Оформлення замовлення» (рисунок створено самостійно)

При спробі прочитати якесь замовлення по ідентифікатору, сервіс знайде всі події для цього замовлення, створить пустий об’єкт та для кожної події в тому ж порядку викличе відповідний метод apply(). Таким чином буде відтворений правильний стан об’єкта.

3.4 Опис програмного середовища з проведення експериментів

Всі мікросервіси та їхня інфраструктура (сервери БД, Kafka, Zookeeper, CDC сервіс, Zipkin, Kibana, Logstash, ElasticSearch) були розгорнуті використовуючи Docker, а саме інструмент Docker Compose для автоматизованого розгортання більше ніж одного контейнера. Всі контейнери мають спільну локальну мережу, що дозволить без проблем взаємодіяти між собою, проте в якості хоста використано відповідне значення, що вказується в конфігурації файлу docker-compose.yaml. Окрім того, варто правильно налаштувати Docker Engine на спільне використання ресурсів хостової операційної системи. Якщо ресурсів буде надало замало: контейнери просто не зможуть стартувати для виконання своїх дій. Для Docker була використана конфігурація, наведена на рисунку 3.6.

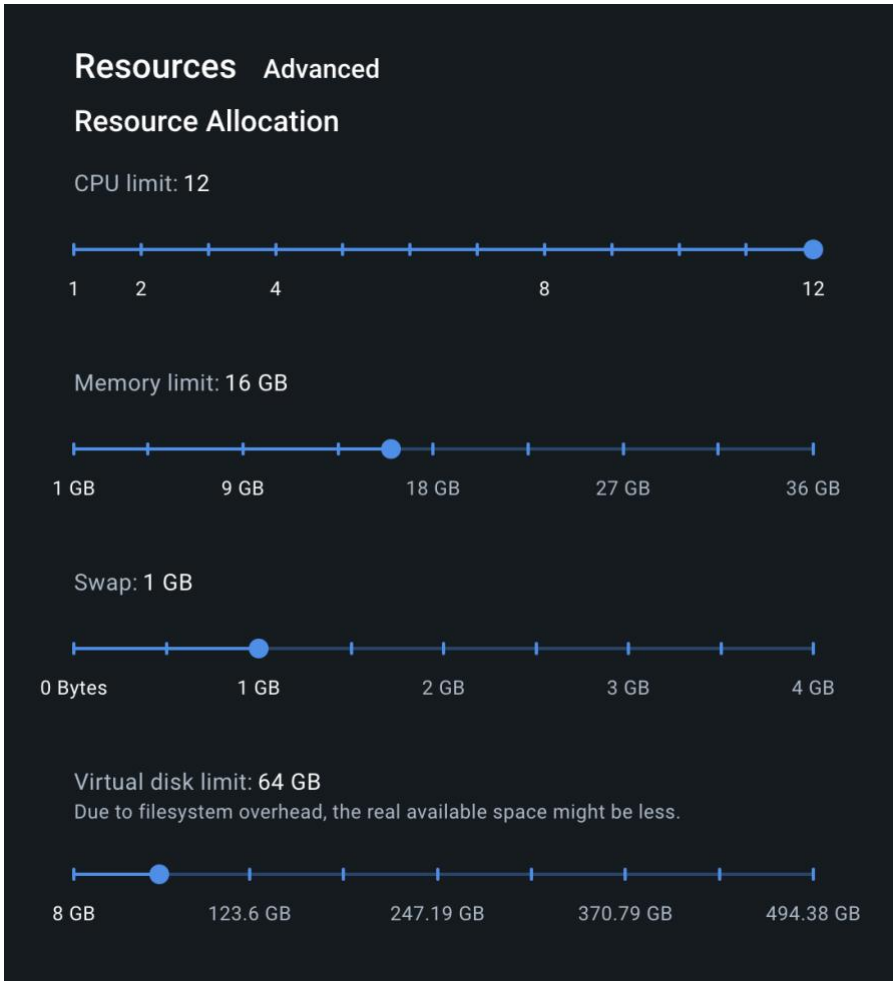

Рисунок 3.6 – Конфігурація ресурсів Docker для проведення експерименту (рисунок створено самостійно)

Окремо була запроваджена конфігурація на рівні JVM для сервісів та інфраструктурних компонентів, що наведена в таблиці 3.1. Обмеження максимального розміру JVM Heap є важливою складовою для подальших експериментів та їх точності, адже необмежений розмір Heap значить, що при збільшенні кількості об’єктів в пам’яті розмір Heap буде зростати допоки не дійде до ліміту ресурсів машини хоста, що є неоптимальним в контексті сучасних систем та може призвести до Memory Leaks та OutOfMemoryError. Окрім того, при необмеженому розмірі Heap Garbage Collector (GC) буде викликатись набагато рідше, що також зробить експерименти більш віддаленими від реальних систем.

Таблиця 3.1 – Конфігурації JVM для мікросервісів та інфраструктури (таблиця виконана самостійно)

Тип конфігурації	Кількість Mb
Core Services Max Heap Size	64
Zookeeper Max Heap Size	64
Kafka Max Heap Size	192
CDC service Max Heap Size	64

Для зручності та автоматизації процесу проведення експериментального дослідження був спроектований та реалізований ще один невеликий мікросервіс – Experiments сервіс. Він передбачає виконання 6 кроків, на кожному з яких ми можемо передати параметр N, тобто кількість ітерацій для експерименту:

– створення набору з N пустих кошиків;

– створення набору з N товарів в середньому наповненні (кількість на складі N, довільна вартість, 2 довільних атрибути товару);

– додавання N товарів в N кошиків (перша транзакція);

– створення N замовлень на основі попередньо наповнених N кошиків (друга транзакція);

– підтвердження оплати для N створених замовлень (третя транзакція);

– скасування N підтверджених замовлень (четверта транзакція).

REST ендпоінти 6 зазначених кроків, що доступні в інтерфейсі Swagger UI, наведені на рисунку 3.6.

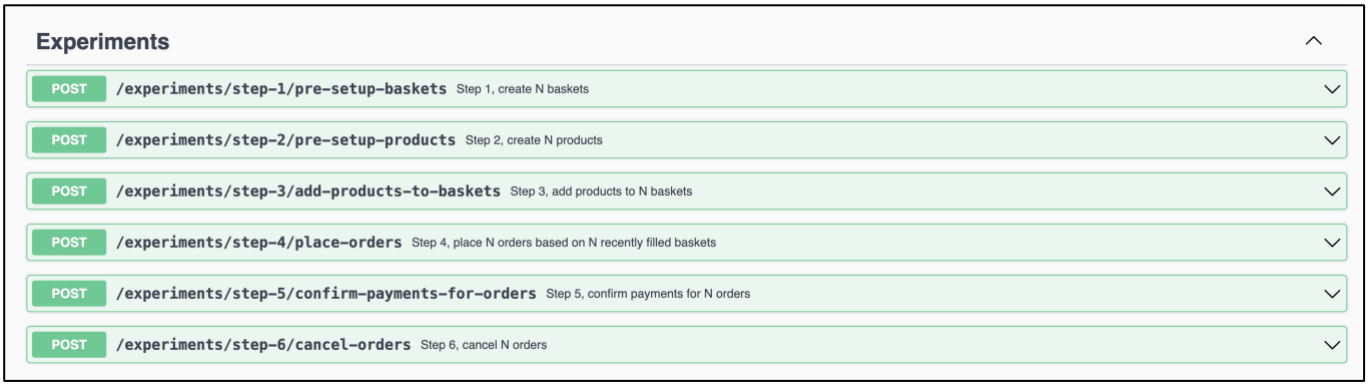

Рисунок 3.6 – Інтерфейс кроків в Experiments сервісі (рисунок створено самостійно)

Як видно із послідовності експериментальних кроків, спроектовані транзакції дуже зручно використовуються в дослідженні, адже вхідні дані кожного кроку, починаючи з 3 кроку – це результат виконання попереднього кроку.

Окрім того, кожен крок приймає не тільки параметр N, а й параметр experimentType (OUTBOX чи EVENT_SOURCING) та executionType (ITERATIVE чи CONCURRENT). Інтерфейс запуску та передачі параметрів для запиту наведено на рисунку 3.7.

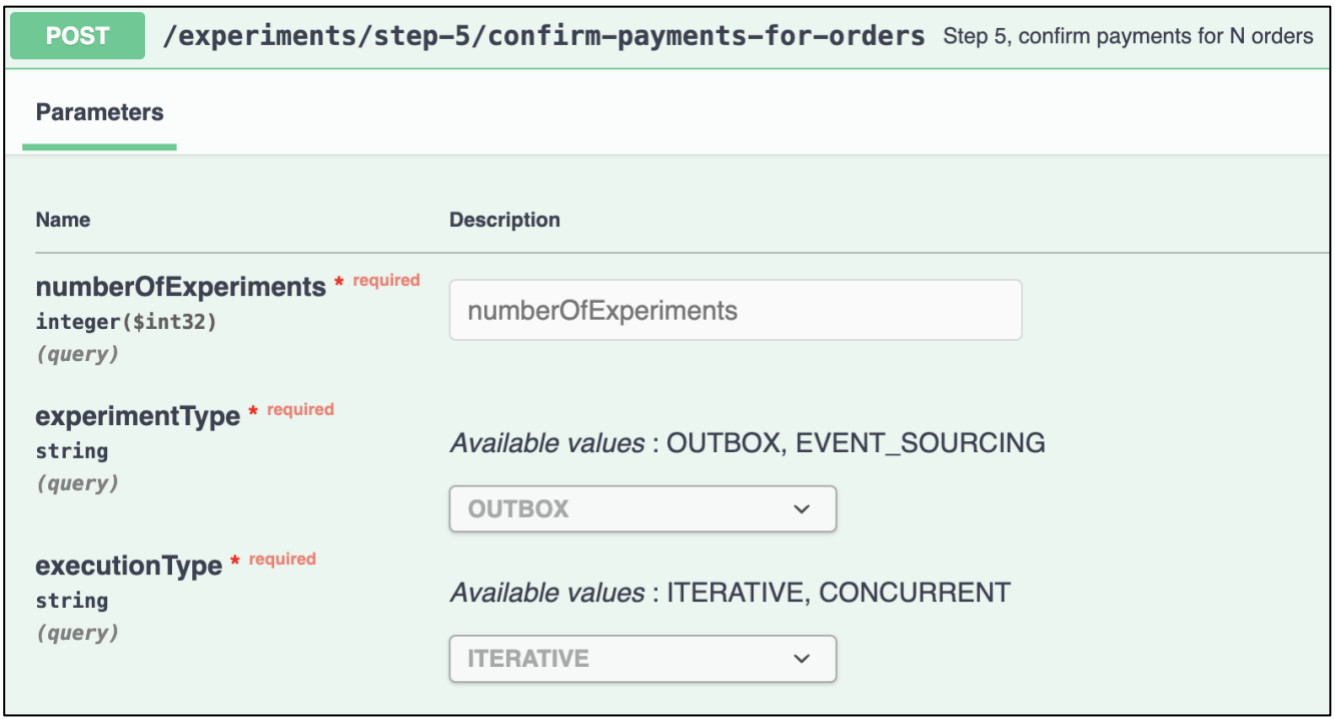

Рисунок 3.7 – Програмна панель запуску експерименту з передачею параметрів (рисунок створено самостійно)

При обранні конфігу запуску ITERATIVE всі N транзакцій будуть виконані послідовно, тобто одна за одною, після завершення кожної буде запущена наступна. Такий експеримент може показати кращі результати по часу та використанню ресурсів, проте він не в повній мірі буде відображати реальні умови використання таких запитів в системах, де можуть бути сотні чи тисячі одночасних запитів в секунду. Саме тому при виборі конфігу CONCURRENT всі запити будуть виконані паралельно з використанням CachedThreadPool, який створить максимум N потоків

виконання і зможе перевикористовувати їх при наступних кроках експерименту. Таким чином буде виконана симуляція реального великого навантаження на розподілену систему.

Для проведення експериментів було вирішено обрати значення N = 100, що є балансом між піковим навантаженням та малим навантаженням і дозволить отримати якісніше середнє значення та вкластись в наявні обмеження по конфігурації максимальної пам’яті для JVM Heap.

Перейдемо до опису інструментів, що будуть використані для збору результатів за визначеними метриками. Для відстеження метрики «Час виконання» був реалізований додатковий функціонал в Experiments сервіс. Перш за все, кожен мікросервіс логує час початку виконання транзакції та час завершення виконання транзакції у вигляді, наведеному на рисунку 3.8.

	message	path
19:14:07.580	[OUTBOX Add Product to Basket SAGA] Start time: 2264145078445 | ID: 1	/usr/share/logstash/logs/basket-service.log
19:14:07.892	[OUTBOX Add Product to Basket SAGA] End time: 2264456970904 | ID: 1	/usr/share/logstash/logs/basket-service.log
19:14:11.334	[OUTBOX Place Order SAGA] Start time: 2267792942864 | ID: 1	/usr/share/logstash/logs/order-service.log
19:14:11.445	[OUTBOX Place Order SAGA] End time: 2268010439572 | ID: 1	/usr/share/logstash/logs/basket-service.log
19:14:14.061	[OUTBOX Confirm Payment SAGA] Start time: 2270626282073 | ID: 1	/usr/share/logstash/logs/order-service.log
19:14:14.200	[OUTBOX Confirm Payment SAGA] End time: 2270765490198 | ID: 1	/usr/share/logstash/logs/order-service.log
19:14:16.788	[OUTBOX Cancel Order SAGA] Start time: 2273352887200 | ID: 1	/usr/share/logstash/logs/order-service.log
19:14:16.852	[OUTBOX Cancel Order SAGA] End time: 2273417218366 | ID: 1	/usr/share/logstash/logs/order-service.log

Рисунок 3.8 – Фільтрація логів по всім транзакціям за початковим та кінцевим часом виконання (рисунок створено самостійно)

Ці логи, окрім відображення у Kibana, зберігаються у локальних файлах basket-service.log, order-service.log, product-service.log. Сервіс проведення експериментів також має доступ до зчитування цих логів. Враховуючи ці деталі, сервіс експериментів має змогу зчитувати файли в зворотньому порядку від кінця до початку, шукати N входжень початкового та кінцевого часу за необхідною транзакцією, групувати їх за ідентифікатором, виділяти звідти значення часу та знаходити загальний час виконання транзакції. Після цього є можливість або

відображати загальний час по всім транзакціям, або відображати середній час виконання.

В сервісі автоматизації експериментів також було реалізовано 16 ендпоінтів для основного контролера, де під кожну транзакцію можна отримати загальний час для всіх N запусків та середній час виконання для N останніх запусків експерименту (див. рис. 3.9). Кожен з ендпоінтів відповідно викликає метод exctractExecutionTimeStatistic, проте для кожної варіації транзакцій передає необхідні налаштування, такі як префікс, по якому можна ідентифікувати транзакцію в логах, розташування лог файлів для початкових записів часу та кінцевих, та власне параметр N.

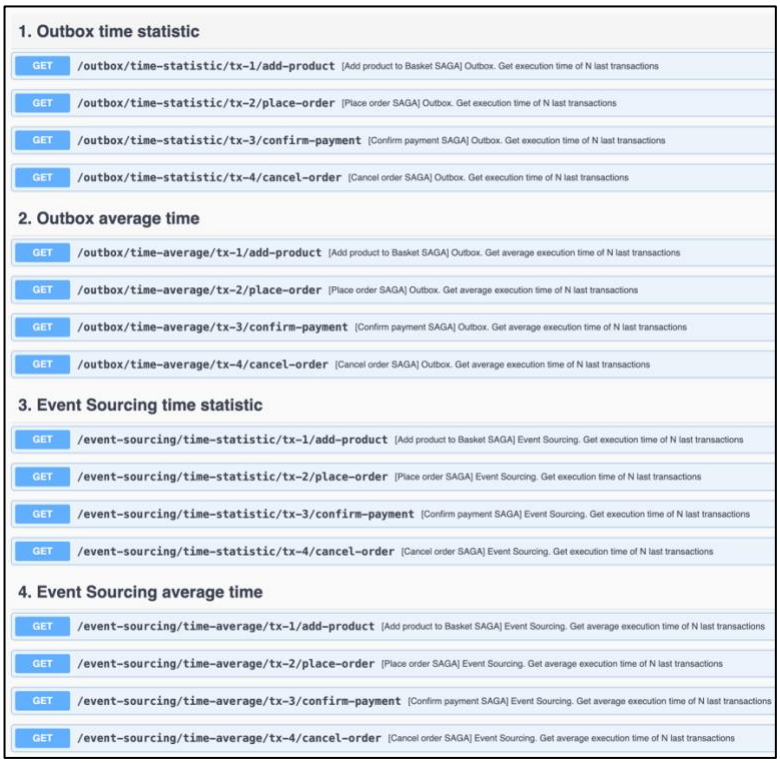

Рисунок 3.9 – Методи для отримання статистики по часу виконання експериментів (рисунок створено самостійно)

Для визначення розміру журналу подій буде використано наступний скрипт для MySQL:

```
CHECK TABLE message;
ANALYZE TABLE message;
```

```
CHECK TABLE received_messages;
ANALYZE TABLE received_messages;

CHECK TABLE events;
ANALYZE TABLE events;

SELECT
    table_name AS `Table`,
    round(((data_length + index_length)) / 1024 / 1024, 4) `MB`
FROM information_schema.TABLES
WHERE table_schema = 'eventuate'
  AND table_name IN ('message', 'received_messages, events');
```

Команди CHECK TABLE та ANALYZE TABLE використовуються для очищення кешів СКБД, тобто для отримання точних результатів. Наведений SELECT запит буде виводити назву таблиці та кількість місця в мегабайтах, яке вона займає на диску.

Для метрик «Споживання оперативної пам’яті» та «Споживання процесорного часу» буде використано інструмент VisualVM, який дозволяє виконувати профайлінг віртуальної машини Java, аналізувати пам’ять, навантаження, кількість потоків тощо. На рисунку 3.10 наведений приклад моніторингу навантаження на процесор та використання оперативної пам’яті для Order Service.

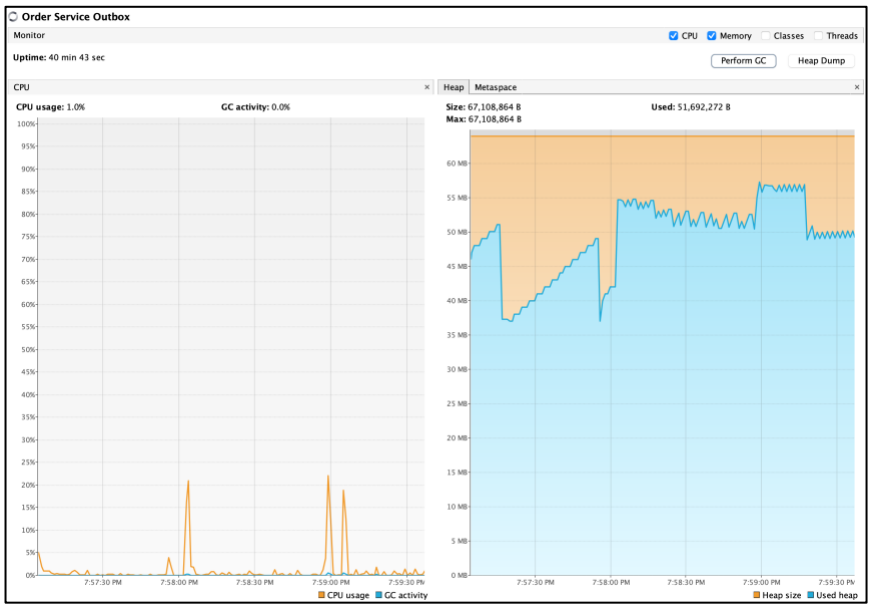

Рисунок 3.10 – Моніторинг процесорного часу та пам’яті в VisualVM (рисунок створено самостійно)

Для заміру останньої метрики, «Пропускної здатності мережі», буде використаний вбудований в Docker Desktop інструмент для моніторингу кількості байтів, що передавались до контейнеру на вхід та кількості байтів, які відправлялись з контейнера (див. рис. 3.11).

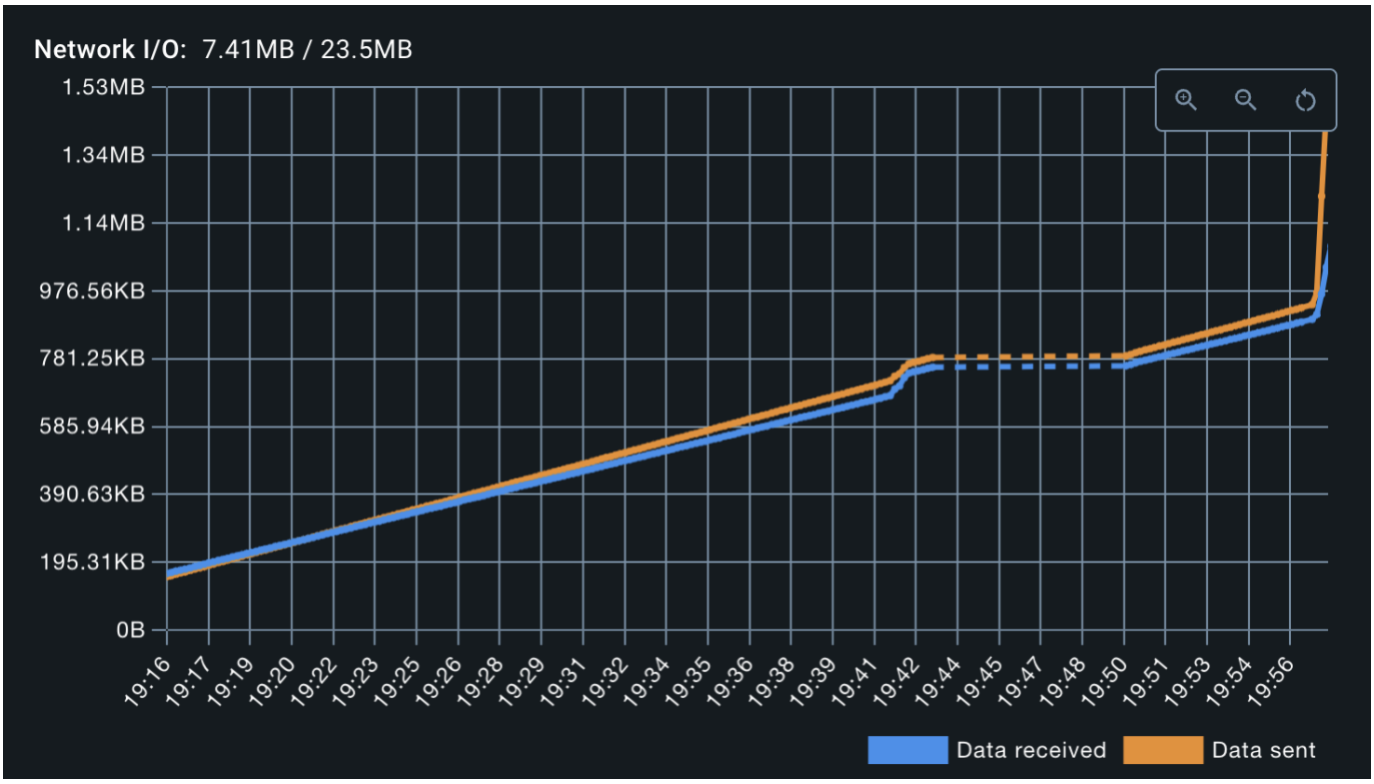

Рисунок 3.11 – Моніторинг мережі в Docker Desktop (рисунок створено самостійно)

Зазначимо, що всі запити будуть виконані на «прогрітих» мікросервісах та БД, тобто з виділеними у внутрішній кеш даними, створеними заздалегідь об’єктами фреймворків, набором підключень до БД, Kafka, тощо.

4 ПРОВЕДЕННЯ ЕКСПЕРИМЕНТАЛЬНОГО ДОСЛІДЖЕННЯ

4.1 Результати експериментальних досліджень

Розглянемо основні тренди продуктивності проведених експериментів з дослідження спроектованих запитів під Transactional Outbox та Event Sourcing. Насамперед, порівняємо середній час виконання 100 запитів в послідовному та паралельному режимах, результати яких наведено в таблицях 4.1 та 4.2.

Таблиця 4.1. Результати за метрикою «Час виконання», середній час для 100 послідовних запитів (таблиця виконана самостійно)

Транзакція / Патерн	Transactional Outbox (мс)	Event Sourcing (мс)
«Додавання товару у кошик»	24,9	22,47
«Оформлення замовлення»	28,24	23,55
«Підтвердження оплати»	22,33	20,38
«Скасування замовлення»	23,11	19,64

Таблиця 4.2. Результати за метрикою «Час виконання», середній час для 100 одночасних запитів (таблиця виконана самостійно)

Транзакція / Патерн	Transactional Outbox (мс)	Event Sourcing (мс)
«Додавання товару у кошик»	655,06	587,51
«Оформлення замовлення»	929,75	801,52
«Підтвердження оплати»	633,79	567,9
«Скасування замовлення»	557,8	571,9

З наведених даних видно, що майже у всіх випадках, окрім «Скасування замовлення» при одночасних запитах, Event Sourcing показує трохи кращі показники, що зумовлено тим, що при використанні цього патерну подія несе в собі два призначення: і запуск наступного обробника, і оновлення даних сутності, тоді як в Outbox ми маємо окремо оновлювати таблицю сутності і таблицю повідомлень.

Варто зазначити, що Event Sourcing показує кращі результати ще тому, що це

був перший цикл наповнення кошика та створення замовлення на його основі. При всіх наступних циклах мали б програватись події всіх попередніх ітерацій, які вже неактуальні, тобто всі наповнення та очищення кошиків. Але при використанні механізму Snapshots ця проблема в перегравання непотрібних подій була б вирішена [37].

Також відразу видно, що приблизно в 30 разів погіршилась швидкодія при 100 одночасних запитах, відповідно до послідовних. Це зумовлено більшим споживанням ресурсів, одночасним запускам потоків обробки, витратами на синхронізацію критичних секцій в коді, базі даних та меседж брокері, накладок на механізм локування та забезпечення консистентності даних. Оптимізація часу виконання при такій кількості запитів разом із забезпеченням ACID властивостей є підґрунтям для іншого дослідження.

Це може слугувати проблемним фактором для систем, де є критичною low latency та high throughput, де зазвичай є вимога, що транзакція має бути виконана за приблизно 200-300 мс. Але зазвичай для таких систем патерн SAGA і не обирається через наявність Eventual Consistency. Розглянемо результати виконання за наступною метрикою: розмір згенерованого журналу подій (див. табл. 4.3).

Таблиця 4.3. Результати за метрикою «Розмір журналу подій», 100 одночасних запитів (таблиця виконана самостійно)

Транзакція / Патерн	Transactional Outbox (mb)	Event Sourcing (mb)
«Додавання товару у кошик»	0,14	0,06
«Оформлення замовлення»	0,23	0,32
«Підтвердження оплати»	0,20	0,05
«Скасування замовлення»	1,08	0,08

Як видно, більший журнал подій у всіх випадках, окрім «Оформлення замовлення», спостерігається за Outbox, адже, як було зазначено раніше, Event Sourcing зберігає в цих подіях весь стан сутностей, тоді як для Outbox транзакцій в

подіях потрібно передавати певну кількість мета-даних для виконання наступного кроку, що може бути затратно по пам’яті. Журнал для Event Sourcing складається з однієї таблиці Events, для Outbox – з двох: Message та Received_Messages. Використання пам’яті може бути також оптимізовано для Outbox, адже є можливість очищувати Message та Received_Messages при однаковій кількості рядків, наприклад, раз на добу. З Event Sourcing очищення подій повністю заборонено, адже це прирівнюється до втрати всіх даних сутностей.

Окремо варто зауважити, що «Підтвердження оплати» та «Скасування замовлення» доволі схожі за своєю будовою і мають займати приблизно однакову кількість пам’яті в журналі подій, що ми і бачимо для Event Sourcing, проте видно, що для Outbox «Скасування замовлення» займає трохи більше 1 Mb місця. Це свідчить про те, що в програмній логіці передається мета-інформація, яка може бути зайвою. Це може призводити до збільшення затрат на зберігання даних, зменшення пропускної здатності мережі та збільшення затрат на очищення Heap Memory JVM, тому варто звертати увагу на цей нюанс під час початкової фази розробки.

Розглянемо специфіку споживання оперативної пам’яті в залучених сервісах (див. табл. 4.4).

Таблиця 4.4. Результати за метрикою «Споживання оперативної пам’яті», 100 одночасних запитів (таблиця виконана самостійно)

Транзакція / Патерн	Transactional Outbox (mb)	Event Sourcing (mb)
«Додавання товару у кошик»	(B) + (C) = 3,31 + 1,97 = 5,28	(B) + (C) = 5,64 + 9,00 = 14,64
«Оформлення замовлення»	(O) + (B) = 4,94 + 2,80 = 7,74	(O) + (B) = 17,61 + 1,41 = 19,02
«Підтвердження оплати»	(O) + (C) = 2,14 + 1,25 = 3,39	(O) + (C) = 1,25 + 7,29 = 8,54
«Скасування замовлення»	(O) + (C) = 2,94 + 1,13 = 4,07	(O) + (C) = 6,06 + 11,02 = 17,08

Умовні скорочення наступні: (B) – Basket, (C) – Catalog, (O) – Ordering. Для відповідних значень було використано суму споживаної оперативної пам’яті мікросервісів, залучених у виконанні транзакції, тобто якщо для «Додавання товару

у кошик» залучені Basket та Catalog, відповідні показники пам’яті знято з них та просумовано.

Як видно із результатів, більшу кількість оперативної пам’яті залучає Event Sourcing. Це передусім зумовлено тим, що при кожному застосуванні нових змін потрібно «переграти» попередні події, для чого кожен раз створюються пусті об’єкти в пам’яті, на яких ці зміни і виконуються. Також можна побачити, що у всіх транзакціях Catalog використовує набагато більше пам’яті. Це зумовлено тим, що, наприклад, при «Підтвердженні оплати» на кроці зменшення кількості товарів на складі ми не можемо виконати за одну сесію роботи з БД виконати певні UPDATE запити, нам треба створювати подію на кожен окремий товар, адже окремий товар є агрегатом і оновлюється теж через набір подій. Отже при наявності складних зв’язків Event Sourcing буде використовувати набагато більше ресурсів RAM.

Розглянемо як споживався процесорний час при виконані транзакцій (див. табл. 4.5).

Таблиця 4.5. Результати за метрикою «Споживання процесорного часу», 100 одночасних запитів (таблиця виконана самостійно)

Транзакція / Патерн	Transactional Outbox (%)	Event Sourcing (%)
«Додавання товару у кошик»	(B) + (C) = 33,3 + 9,7 = 43,0	(B) + (C) = 28,4 + 8,4 = 36,8
«Оформлення замовлення»	(O) + (B) = 41,9 + 12,8 = 54,8	(O) + (B) = 31,4 + 13,1 = 44,5
«Підтвердження оплати»	(O) + (C) = 21,7 + 9,2 = 30,9	(O) + (C) = 23,8 + 10,9 = 34,7
«Скасування замовлення»	(O) + (C) = 17,9 + 8,7 = 26,6	(O) + (C) = 23,3 + 8,5 = 31,8

Для замірів використовувався так само VisualVM, де було взято піковий процесорний час при виконанні транзакції з кожного залученого мікросервісу та прораховано їх суму для отримання показника загального використання ресурсів CPU. Як видно, однозначно відповісти хто краще споживає процесорний час не так легко. Але видно, що найбільш затратні транзакції по попереднім метрикам, такі як «Додавання товару у кошик» та «Оформлення замовлення», використовують менше

інструкцій процесора для Event Sourcing. І навпаки для останніх двох транзакцій, але, як було наведено для попередньої метрики, це зумовлено більшою кількістю подій під кожен товар у замовленні.

Розглянемо останню метрику: пропускну здатність мережі, тобто співвідношення байтів, які були отримані та надіслані сервісом (I/O) через мережу (див. табл. 4.6).

Таблиця 4.6. Результати за метрикою «Пропускна здатність мережі», 100 одночасних запитів (таблиця виконана самостійно)

Транзакція / Патерн	Transactional Outbox (mb)	Event Sourcing (mb)
«Додавання товару у кошик»	1,07 / 1,20	1,22 / 1,13
«Оформлення замовлення»	0,91 / 0,99	1,35 / 1,14
«Підтвердження оплати»	0,83 / 1,23	1,69 / 1,35
«Скасування замовлення»	1,18 / 1.23	2,01 / 1,31

Для замірів використовувався інструмент Docker Desktop, який дозволяє моніторити запущені в кластері Docker Compose контейнери та відслідковувати тенденцію зростання навантаження на мережу з плином часу. Метрика рахувалась як сума I/O мегабайтів за використаними сервісами, тобто кількість всього трафіку, який був переданий по мережі в рамках виконання транзакції.

Як видно, менше навантаження на мережу було виконано транзакціями при використанні патерну Outbox. Дані, що передавались, були враховані в REST запитах на запуск транзакції, запитах до БД, передачі повідомлень через брокер повідомлень. Більша кількість подій в Event Sourcing зумовлює також дублювання певної мета-інформації, як наприклад ідентифікаторів сутностей, що в Outbox використовується більш ефективно та дозволяє передавати в загальному менше корисного навантаження в мережі.

4.2 Оцінка якості та вироблені рекомендації

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

В системах, де вимогою є швидкодія транзакцій, варто надати перевагу Event Sourcing. Кожна зміна стану сутності в базі є лише подією, яка швидко додається в кінець таблиці за константний час. Так само швидко можуть бути відтворені всі події для конкретної сутності при ефективній реалізації сховища подій. Окрім того, Event Sourcing підходить для систем, де є вимога наявності чіткого аудиту подій – ця вимога легко покривається тим, що в нас є наявність відслідкувати до найменших деталей як змінювався стан сутності з плином часу. Разом із цим, патерн оптимізовує використання пам’яті БД, так як не потребує окремого місця для розділення бізнес-даних та опублікованих подій.

Event Sourcing добре підходить до транзакцій, де залучаються зв’язки «Один до одного». Як було видно із результатів досліджень, транзакції із залученням зв’язків «Один до багатьох» відпрацьовували гірше у зв’язку з необхідністю публікації подій під кожен зв’язок, що було неоптимальним у контексті використання оперативної пам’яті та навантаження на мережу. Тому для систем, де використання ресурсів пам’яті є критично важливим, варто зробити вибір в сторону іншого патерну.

Transactional Outbox добре підходить при використанні стандартних реляційних СКБД та стане гарним вибором для систем, де є вимога в збереженні ресурсів виконання мікросервісу, а саме оперативної пам’яті віртуальної машини виконання та меншого навантаження на мережу. Проте варто з обережністю ставитись до його використання в системах, де є вимога по оптимізації кількості місця, що займає БД на диску, та залучати додаткові зусилля на розробку механізму очищення непотрібних повідомлень з часом.

Обидва патерни показали приблизно однакові показники використання CPU, але все одно вони дещо зависокі. Це зумовлене великою кількістю запитів та подій для обробки зв’язків сутностей. Дослідження патернів на величезному навантаженні також може стати підґрунтям для іншого дослідження. В такому випадку варто розглянути альтернативи обробки транзакцій, які дозволять залучити не тільки реляційні БД, а й NoSQL БД, такі як MongoDB та Redis, що можуть значно зменшити CPU затрати та покращити пропускну здатність системи в цілому. Окрім того, варто також взяти до уваги яким саме чином можливо реалізувати ACID властивості в NoSQL БД [38].

Отже, якщо підсумувати:

– Event Sourcing рекомендується використовувати в системах з високими вимогами до швидкодії транзакцій та необхідністю ретельного аудиту подій; підходить для систем з одноразовими зв’язками (один до одного) та ефективним використанням пам’яті;
– Transactional Outbox рекомендується для систем, де важлива економія ресурсів оперативної пам’яті та мережевого трафіку; підходить для використання з реляційними СКБД, де критично важлива надійність збереження даних.

В майбутньому варто дослідити додаткові методи обробки розподілених транзакцій на великих навантаженнях та з залученням NoSQL баз даних. Це може включати дослідження нових підходів та патернів, що дозволять покращити ефективність системи та зменшити витрати на ресурси, такі як CPU та пам'ять. Подібні дослідження можуть забезпечити більш глибоке розуміння та дати можливість створення ще більш оптимізованих систем для різноманітних бізнес-вимог.

ВИСНОВКИ

В ході виконання кваліфікаційної роботи магістра була проаналізована проблемна область обробки розподілених транзакцій в мікросервісній архітектурі та досліджені конкретні методи реалізації, а саме хореографічна варіація патерну SAGA в поєднанні з Transactional Outbox та Event Sourcing.

Було опрацьовано наступні питання:

– проведено аналіз проблемної області та розроблена постановка задачі;

– обрано конкретні методи для подальшого дослідження;

– обрано реальну предметну галузь в сфері електронної комерції, спроектовано схеми баз даних, побудовано мікросервісну архітектуру, описано бізнес-транзакції для проведення експериментів;

– сплановано експериментальне дослідження, обрано технології для реалізації програмної системи для проведення експериментів, розроблено скрипти для створення фізичної моделі даних;

– спроектовано середовище для проведення експериментів, налаштовано обмеження по ресурсами для більш чітких показників метрик, реалізовано допоміжний програмний комплекс для автоматизації проведення експерименту;

– проведено експеримент та виконано аналіз результатів, надано рекомендації.

Для кваліфікаційної роботи була обрана реальна прикладна область зі сфери електронної комерції, в контексті якої була розроблена логічна модель бази даних.

В ході роботи було складено план експериментального дослідження, де було виявлено основні метрики якості, а саме час виконання, розмір журналу подій, обсяг залученої оперативної пам’яті, обсяг використання процесорного часу, пропускну здатність мережу. Були спроектовані сценарії бізнес-транзакцій, які відповідають реальним варіантам використання системи, такі як додавання товару до кошику, оформлення замовлення, оновлення запасів товарів після покупки та скасування

замовлення.

Було реалізовано допоміжне програмне рішення, яке дало змогу швидко та автоматизовано запускати довільний набір транзакцій в двох режимах: послідовному та паралельному. Окрім того, було реалізовано рахування середнього часу виконання транзакцій. Було проведено замір кожної з метрик для 100 послідовних чи паралельних запитів, використовуючи при цьому такі інструменти, як VisualVM та Docker Desktop.

Було отримано результати дослідження та сформована оцінка якості та рекомендації стосовно вибору конкретного шаблону для реалізації методу обробки розподілених транзакцій. Підсумовуючи головні тези рекомендацій, Transactional Outbox доцільно використовувати в системах, де ключовими вимогами є економія оперативної пам'яті та мінімізація мережевого трафіку, він підходить для реляційних СКБД, що потребують високої надійності збереження даних. Event Sourcing рекомендовано для систем з високими вимогами до швидкості транзакцій та необхідністю детального аудиту подій. Цей підхід ефективний для систем з одноразовими зв'язками (один до одного) та оптимальним використанням пам'яті.

За результатами роботи було опубліковано тези доповіді «Дослідження методів обробки розподілених транзакцій в мікросервісній архітектурі» на 28-й Міжнародний молодіжний форум «Радіоелектроніка і молодь у XXI столітті» (див. додаток Г), а також підготовлено для подачі наукову статтю – «Research of choreography methods for processing distributed transactions in a microservice architecture based on the SAGA pattern» (див. додаток Г).

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. What is Microservices? [Електронний ресурс] – URL: https://www.linkedin.com/pulse/what-microservices-ashish-ranjan/ (дата звернення: 07.03.2024).

2. Newman S. "Building Microservices: Designing Fine-Grained Systems", 2d ed. – O'Reilly Media, 2021. – 616 p.

3. Microservices: Related patterns [Електронний ресурс] – URL: https://microservices.io/patterns/microservices.html#related-patterns (дата звернення: 08.03.2024).

4. Why and How Netflix, Amazon, and Uber Migrated to Microservices: Learn from Their Experience [Електронний ресурс] – URL: https://www.hys-enterprise.com/blog/why-and-how-netflix-amazon-and-uber-migrated-to-microservices-learn-from-their-experience/ (дата звернення: 10.03.2024).

5. Mazurova O., Syvolovskyi I., Syvolovska O. NoSQL database logic design methods for mongodb and Neo4j. Innovative Technologies and Scientific Solutions for Industries. 2022. No. 2 (20). P. 52–63. Doi: 10.30837/itssi.2022.20.052.

6. Pattern: Shared database [Електронний ресурс] – URL: https://microservices.io/patterns/data/shared-database.html (дата звернення: 11.03.2024).

7. Pattern: Database per service [Електронний ресурс] – URL: https://microservices.io/patterns/data/database-per-service.html (дата звернення: 11.03.2024).

8. G. Chockler і A. Gotsman Multi-shot distributed transaction commit / G. Chockler і A. Gotsman. - Distrib. Comput. - вип. 34, вип. 4, Сер 2021. - с. 301–318.

9. G. Sharma і C. Busch Distributed transactional memory for general networks / G. Sharma і C. Busch. - Distrib. Comput. - вип. 27, вип. 5, Жов 2014. - с. 329–362.

10. Distributed transaction patterns. The dual write problem [Електронний ресурс] – URL: https://developers.redhat.com/articles/2021/09/21/distributed-transaction-patterns-microservices-compared#the_dual_write_problem (дата звернення: 12.03.2024).

11. Distributed transaction patterns. The modular monolith [Електронний ресурс] – URL: https://developers.redhat.com/articles/2021/09/21/distributed-transaction-patterns-microservices-compared#the_modular_monolith-h2 (дата звернення: 12.03.2024).

12. V. Chanron і K. Lewis Convergence and Stability in Distributed Design of Large Systems / V. Chanron і K. Lewis. - presented at the ASME 2004 International 73 Design Engineering Technical Conferences and Computers and Information in Engineering Conference. - 2008. - с. 593–603.

13. What’s the most important part of Event-Driven Architecture? [Електронний ресурс] – URL: https://dev.to/bikodes/whats-the-most-important-part-of-event-driven-architecture-m2a (дата звернення: 13.03.2024).

14. D. Surya Sai Venkatesh і S. Agarwal Data Access Pattern Recommendations for Microservices Architecture / D. Surya Sai Venkatesh і S. Agarwal. - in 2022 IEEE 15th International Conference on Cloud Computing (CLOUD). - 2022. - с. 241–243.

15. Distributed transaction patterns. Orchestration [Електронний ресурс] – URL: https://developers.redhat.com/articles/2021/09/21/distributed-transaction-patterns-microservices-compared#orchestration (дата звернення: 12.03.2024).

16. T. Zhou і Y. Wei Database replication technology having high consistency requirements / T. Zhou і Y. Wei. - in 2013 IEEE Third International Conference on Information Science and Technology (ICIST). - 2013. - с. 793–797.

17. Distributed transaction patterns. Choreography [Електронний ресурс] – URL: https://developers.redhat.com/articles/2021/09/21/distributed-transaction-patterns-microservices-compared#choreography (дата звернення: 12.03.2024).

18. Pattern: Event Sourcing [Електронний ресурс] – URL: https://microservices.io/patterns/data/event-sourcing.html (дата звернення: 15.03.2024).

19. Pattern: DDD Aggregate [Електронний ресурс] – URL: https://microservices.io/patterns/data/aggregate.html (дата звернення: 15.03.2024).

20. Evans E. "Domain-Driven Design: Tackling Complexity in the Heart of Software", 1st ed. – Addison-Wesley Proffesional, 2003. – 560 p.

21. Pattern: CQRS [Електронний ресурс] – URL: https://microservices.io/patterns/data/cqrs.html (дата звернення: 16.03.2024).

22. Distributed transactions. Choreography with Event Sourcing [Електронний ресурс] – URL: https://developers.redhat.com/articles/2021/09/21/distributed-transaction-patterns-microservices-compared#choreography-h2 (дата звернення: 12.03.2024).

23. Distributed transactions patterns. Parallel pipelines [Електронний ресурс] – URL: https://developers.redhat.com/articles/2021/09/21/distributed-transaction-patterns-microservices-compared#parallel_pipelines (дата звернення: 12.03.2024).

24. S. Kapferer і O. Zimmermann Domain-specific Language and Tools for Strategic Domain-driven Design, Context Mapping and Bounded Context Modeling / S. Kapferer і O. Zimmermann. - presented at the 8th International Conference on Model-Driven Engineering and Software Development. - 2022. - с. 299–306.

25. Kleppmann M. "Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems", 1st ed. – O'Reilly Media, 2017. – 611 p.

26. Two-Phased Commit and eXtended Architecture. Understanding Distributed Transactions with 2PC and XA [Електронний ресурс] – URL: https://blog.sofwancoder.com/two-phased-commit-and-extended-architecture-the-basics (дата звернення: 16.03.2024).

27. Listen to yourself design pattern for event driven microservices [Електронний ресурс] – URL: https://medium.com/@odedia/listen-to-yourself-design-pattern-for-event-driven-microservices-16f97e3ed066 (дата звернення: 17.03.2024).

28. Java Microservices [Електронний ресурс] – URL: https://hazelcast.com/glossary/java-microservices/ (дата звернення: 25.03.2024).

29. Spring Cloud [Електронний ресурс] – URL: https://spring.io/projects/spring-cloud (дата звернення: 25.03.2024).

30. Apache Kafka: Use Cases [Електронний ресурс] – URL: https://kafka.apache.org/uses (дата звернення: 25.03.2024).

31. Eventuate Tram [Електронний ресурс] – URL: https://eventuate.io/abouteventuatetram (дата звернення: 25.03.2024).

32. Eventuate Local [Електронний ресурс] – URL: https://github.com/eventuate-local/eventuate-local (дата звернення: 25.03.2024).

33. Docker [Електронний ресурс] – URL: https://www.docker.com/ (дата звернення: 26.04.2024).

34. Elastic Stack [Електронний ресурс] – URL: https://www.elastic.co/elastic-stack (дата звернення: 26.04.2024).

35. Configuring the Eventuate CDC Service [Електронний ресурс] – URL: https://eventuate.io/docs/manual/eventuate-tram/latest/cdc-configuration.html (дата звернення: 25.05.2024).

36. About Event Sourcing [Електронний ресурс] – URL: https://eventuate.io/whyeventsourcing.html (дата звернення: 26.05.2024).

37. Consistent retrospective snapshots in distributed event-sourced systems / B. Erb et al. 2017 International Conference on Networked Systems (NetSys), Gottingen, 13–16 March 2017. 2017. URL: https://doi.org/10.1109/netsys.2017.7903947 (дата звернення: 26.05.2024).

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