МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ІВАНА ФРАНКА

Факультет прикладної математики та інформатики
(повне найменування назва факультету)

Кафедра програмування
(повна назва кафедри)

Магістерська робота

ДОСЛІДЖЕННЯ МЕТОДІВ ПОБУДОВИ EVENT SOURCING СИСТЕМ

Виконав: студент групи ПМІм-21
спеціальності
122-"Комп'ютерні науки"
(шифр і назва спеціальності)

(підпис) Доскач Д. Я.
(прізвище та ініціали)

Керівник (підпис) Музичук А. О.
(прізвище та ініціали)

Консультант (підпис) Нобіс В.В.
(прізвище та ініціали)

Рецензент підпис) Шунькін Ю.В.
(прізвище та ініціали)

Львів – 2022

ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ІВАНА ФРАНКА

Факультет *прикладної математики та інформатики*

Кафедра *програмування*

Освітньо-кваліфікаційний рівень *магістр*

Спеціальність *122 – комп'ютерні науки*

«ЗАТВЕРДЖУЮ»

Зав. кафедрою *~~доц.~~ Ярошко С.А.*

« *13* » *вересня* 2022 р.

З А В Д А Н Н Я

НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Доскач Денис Ярославович

(прізвище, ім'я, по батькові)

1. Тема роботи

Дослідження методів побудови Event Sourcing систем

керівник роботи *доц. Музичук А.О.*

затверджена Вченою радою факультету від « *13* » *вересня* 2022 р., № *15*

2. Строк подання студентом роботи *12.12.2022 р.*

3. Вихідні дані до роботи

Література та інтернет-ресурси за тематикою роботи

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

Проаналізувати існуючу інформацію про підхід Event Sourcing, зрозуміти його область застосування та деталі реалізації. Визначити, які фактори впливають на можливість використання різних типів баз даних для реалізації сховища подій. Дослідити можливість застосування моделі акторів для оптимізації Event Sourcing систем. Реалізувати набір компонент (бібліотек) для створення Event Sourcing систем з використанням SQL та NoSQL баз даних, а також із застосуванням моделі акторів. Виконати тестування швидкодії отриманих реалізацій у різних сценаріях використання, порівняти результати та зробити висновки про доцільність використання кожної реалізації.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

Схеми, діаграми

Презентація дипломної роботи

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання *05.09.22 р.*

КАЛЕНДАРНИЙ ПЛАН

№	Найменування етапів дипломної (кваліфікаційної) роботи	Строк виконання етапів роботи	Примітка
1.	*Огляд існуючих систем та технологій*	*Вересень*	
2.	*Постановка задачі*	*Вересень*	
3.	*Розробка Event Sourcing системи на основі MS SQL*	*Жовтень*	
4.	*Розробка Event Sourcing системи на основі Mongo DB*	*Жовтень*	
5.	*Застосування моделі акторів до розроблених систем*	*Листопад*	
7.	*Тестування реалізації, аналіз та порівняння результатів*	*Листопад*	
8.	*Оформлення роботи*	*Грудень*	

Студент _____ (підпис) Доскач Д.Я.

Керівник роботи _____ (підпис) доц. Музичук А.С

ЗМІСТ

ВСТУП **5**

РОЗДІЛ 1. EVENT SOURCING СИСТЕМИ **6**

1.1 Опис підходу 6

РОЗДІЛ 2. ПАРАЛЕЛЬНА РОБОТА З ДАНИМИ **10**

2.1 Проблеми паралелізму у Event Sourcing системах 10
2.2 Способи вирішення конфліктів паралелізму 12
2.3 Оптимістичне вирішення конфліктів паралелізму 13
2.4 Модель акторів 15

РОЗДІЛ 3. РЕАЛІЗАЦІЯ СХОВИЩА ПОДІЙ **18**

3.1 Аналіз вимог 18
3.2 Абстракції сховища подій 18
3.3 Реалізація сховища подій у реляційній базі даних 26
3.4 Реалізація сховища подій у нереляційній базі даних 32
3.5 Застосування моделі акторів у Event Sourcing системі 42

РОЗДІЛ 4. АНАЛІЗ ТА ПОРІВНЯННЯ **47**

4.1 Принципи тестування 47
4.2 Тест 1: Запис подій 47
4.3 Тест 2: Регідрація об’єкта з заданою кількістю подій 49
4.4 Тест 3: Регідрація об’єкта певної версії із заданою кількістю подій 51
4.5 Тест 4: Зміна об’єкта 52
4.6 Тест 5: Одночасна зміна об’єкта кількома користувачами 57

ВИСНОВОК **60**

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ **61**

ВСТУП

Однією із найважливіших складових будь-яких сучасних інформаційних систем є спосіб збереження та доступу до даних. З технічної точки зору, від нього залежить швидкодія системи, її підтримуваність, масштабованість та стабільність. Окрім цього, від типу інформації, що ми зберігаємо, залежить можливість її використання для проведення статистичних досліджень, аналітики та оптимізації процесів. Найбільш поширеним способом збереження даних є використання однієї бази даних, що використовується і для читання, і для запису, яка містить інформацію про поточний стан сутностей. Хоч цей спосіб і підходить для вирішення більшості повсякденних завдань, він має два важливих недоліки, які проявляються при розробці складних систем.

Перший недолік полягає у тому, що кожного разу, коли стан сутності змінюється, дані про його попередній стан безповоротно втрачаються. Як наслідок, втрачається цінна інформація, яка може бути використана для аналізу та оптимізації існуючих процесів, збору статистичної інформації та інших цілей.

Другий недолік полягає у неможливості незалежної оптимізації читання та запису даних. Типові засоби оптимізації одного аспекту часто негативно впливають на швидкодію іншого. До прикладу, індекси у реляційних базах даних пришвидшують зчитування інформації, проте уповільнюють запис, а нормалізація даних допомагає забезпечувати цілісність даних при змінах, проте уповільнює читання. Для вирішення цієї проблеми був створений принцип розподілу відповідальності запитів та команд (eng. CQRS – Command Query Responsibility Segregation), у якому сховища даних, що використовуються для читання та для запису розділяються, що дозволяє виконувати незалежну оптимізацію кожного з них. Проте, виникає проблема синхронізації даних у сховищі для запису і у сховищі для читання.

Event Sourcing це підхід, який дозволяє вирішити обидві проблеми. Він полягає у тому, що замість збереження поточного стану сутності, ми зберігаємо послідовність

подій, що призвела до цього стану. Таким чином ми зберігаємо усю інформацію про зміни, що відбулися з сутністю із моменту її створення. Окрім того, використання сховища подій в якості сховища для запису (у принципі CQRS) дозволяє ефективно вирішити проблему синхронізації даних. Для цього, події не лише зберігаються, а і обробляються так званими проекціями, які у відповідь на подію змінюють дані у сховищі для читання.

Метою цієї роботи є дослідження способів реалізації сховища подій, яке є основним компонентом будь-якої Event Sourcing системи. Для виконання роботи буде визначено необхідні вимоги до баз даних для його реалізації, проаналізована можливість використання SQL та NoSQL баз даних, а також можливість застосування моделі акторів. Після цього буде проведено тестування швидкодії кожного способу реалізації і зроблено відповідні висновки про переваги та недоліки описаних підходів, а також доцільність їх використання у різних випадках.

РОЗДІЛ 1. EVENT SOURCING СИСТЕМИ

1.1 Опис підходу

Традиційним підходом до реалізації сховища даних у програмних системах є так звана CRUD модель (eng. Create Read Update Delete). Вона полягає у тому, що функціонал системи дозволяє створювати дані, зчитувати їх, змінювати існуючі дані, а також виконувати операцію видалення. Event Sourcing це альтернативний підхід, у якому замість актуального стану окремої сутності, зберігаються усі зміни, які відбулися з нею [1]. Коротко, відмінність між цими підходами можна описати наступним чином: традиційний підхід може дати відповідь лише на запитання «У якому стані зараз знаходиться сутність?», тоді як у системі, що використовує Event Sourcing можна також отримати відповідь на запитання, як ми опинилися у цьому стані. Таким чином не втрачається важлива інформація, яка може використоуватись

для аналізу та оптимізації процесів за допомогою статистики, або й інших методів обробки даних.

У контексті Event Sourcing системи, зміна, що відбулася із сутністю називається подією (eng. Event) [1]. Події володіють наступними властивостями [2]:

а) Подія – це інформація про зміну в минулому.

б) Подія є незмінною. Тобто після того, як вона була збережена, її не можна змінити чи видалити. Ця властивість слідує з пункту а.

в) Події можуть публікуватися та оброблятися іншими частинами системи, які можуть бути в них зацікавлений.

г) Подія описує зміну з точки зору процесу, у якому вона відбулася. Наприклад, події «Готівка знята з банкомату» та «Заробітна плата нарахована» містять у собі більше цінної інформації, аніж «Баланс користувача змінився».

Подію можна розглядати як інформацію про перехід із одного стану до іншого. Зрозуміло, що для отримання кінцевого стану, нам також потрібен механізм опрацювання подій. Зазвичай для цього використовують так звані агрегати [3]. Цей термін походить з предметно-орієнтованого програмування (eng. DDD – domain driven design) і означає сукупність пов’язаних між собою сутностей які вигідно розглядати як одне ціле. Дані всередині агрегату завжди повинні знаходитися у несуперечливому стані. Щоб цього досягти, в агрегаті визначається один об’єкт, який називається коренем агрегату. Через корінь відбувається взаємодія зі всіма сутностями агрегату за допомогою його методів, які повинні забезпечувати узгодженність даних. Прикладом агрегату може бути замовлення а також інформація про товари у ньому. Хоч вони і є окремими об’єктами, їх зручно розглядати як одне ціле та інкапсулювати доступ до інформації про товари замовлення через об’єкт замовлення. У контексті Event Sourcing, ми можемо змінити стан агрегату, надіславши його кореню команду. Агрегат при цьому опрацьовує команду та створює події, що описують виконані зміни. Після цього ці події зберігаються у вигляді потоку подій. Зазвичай події

записується у так зване сховище подій, яке описане далі. Підсумовуючи, події у Event Sourcing системах асоціюються з агрегатами, які є сукупністю сутностей, цілісність та узгодженість яких підтримується разом. Уся взаємодія з агрегатом відбувається через його корінь, який зберігає усі зміни стану у вигляді подій. Події, що стосуються одного агрегату називаються потоком і зберігаються у сховищі подій.

Для того, щоб отримати стан агрегату у будь-який момент часу необхідно отримати зі сховища подій потік, що містить події конкретного агрегату та виконати операцію, що називається агрегацією або регідрацією. Оскільки кожна подія містить інформацію про перехід із одного стану до іншого, операція агрегації зводиться до послідовного виконання кожного переходу, починаючи зі стану нової сутності. На практиці, час отримання актуального стану агрегату зростатиме із збільшенням кількості подій у його потоці. Це означає, що швидкодія такої системи буде повільно, але стабільно зменшуватися. Для вирішення цієї проблеми використовують так звані знімки [4] (eng. Snapshot), які представляють стан агрегату у певний момент часу. Тоді, операція агрегації пришвидшується, оскільки потрібно прочитати лише події, які сталися після останнього знімку, замість того, щоб зчитувати події від початку існування агрегату. Використання знімків не позбавлене недоліків, оскільки вони ускладнюють підтримку та версіонування моделі. Існують різні стратегії створення знімків. Їх можна створювати після кожних N подій, раз в певний проміжок часу, чи після події певного типу. Кожен з цих підходів має свої переваги та недоліки, які, проте, у цій роботі не розглядаються.

Для демонстрації описаних концепцій у лістингу 1.1 представлено простий агрегат, який описує вимірювання температури з певного датчика. Агрегат складається із однієї сутності, яка представлена класом *TemperatureMeasurement*, яка також є коренем агрегату. Окрім того, доменна модель містить дві події *TemperatureMeasurementStarted* і *TemperatureMeasurementRecorded,* які відповідно містять інформацію про початок вимірювань, а також про надходження нових даних про температуру.

```
public class TemperatureMeasurement : Aggregate
{
    1 reference
    public DateTimeOffset Started { get; private set; }
    1 reference
    public DateTimeOffset? LastRecorded { get; private set; }
    2 references
    public List<decimal> Mesurements { get; private set;  } = default!;

    1 reference
    private TemperatureMeasurement(Guid id)
    {
        var @event = TemperatureMeasurementStarted.Create(id);
        Enqueue(@event);
        Apply(@event);
    }

    13 references
    public void Record(decimal temperature)
    {
        if (temperature < -273) throw new ArgumentOutOfRangeException(nameof(temperature));
        var @event = TemperatureRecorded.Create(Id, temperature);
        Enqueue(@event);
        Apply(@event);
    }

    2 references
    private void Apply(TemperatureMeasurementStarted @event)
    {
        Id = @event.MeasurementId;
        Started = @event.StartedAt;
        Mesurements = new List<decimal>();
    }

    2 references
    private void Apply(TemperatureRecorded @event)
    {
        Mesurements.Add(@event.Temperature);
        LastRecorded = @event.MeasuredAt;
    }
    other members
}
```

Лістинг 1.1 – Реалізація простого агрегату

Бачимо, що об’єкт містить інформацію про стан, а також методи для його модифікції. До прикладу, метод *Record* записує інформацію про нове вимірювання. При цьому він впевнюється, що дані передані ззовні – валідні, що забезпечує правильність доменної моделі. Після цього, він створює подію *TemperatureRecorded,* додає її до внутрішньої колекції подій за допомогою методу *Enqueue,* після чого викликає метод *Apply,* який виконує зміну стану агрегату, що описана подією *TemperatureRecorded.* В подальшому ця подія буде збережена у сховище подій, а

відновити стан агрегату можна буде прочитавши її та викликавши той самий метод *Apply*.

РОЗДІЛ 2. ПАРАЛЕЛЬНА РОБОТА З ДАНИМИ

2.1 Проблеми паралелізму у Event Sourcing системах

Одним із важливих аспектів при реалізації Event Sourcing системи є спосіб вирішення конфліктів при одночасному доступі до даних. Розглянемо спочатку цю ситуацію у випадку застосування класичного підходу (зберігання поточного стану), а потім спробуємо зрозуміти принципові відмінності у випадку, коли інформація зберігається у формі потоку подій.

Розглянемо процес зміни даних про об’єкт у загальному випадку (Рис. 2.1). На першому кроці потрібно прочитати дані про об’єкт, що зберігаються у базі даних. Таким чином, ми завантажуємо у пам’ять репрезентацію тої інформації, що міститься у базу. Наступним кроком, ми змінюємо нашу репрезентацію в пам’яті. І врешті решт зберігаємо змінені дані у базі даних, перезаписуючи при цьому попередній стан.

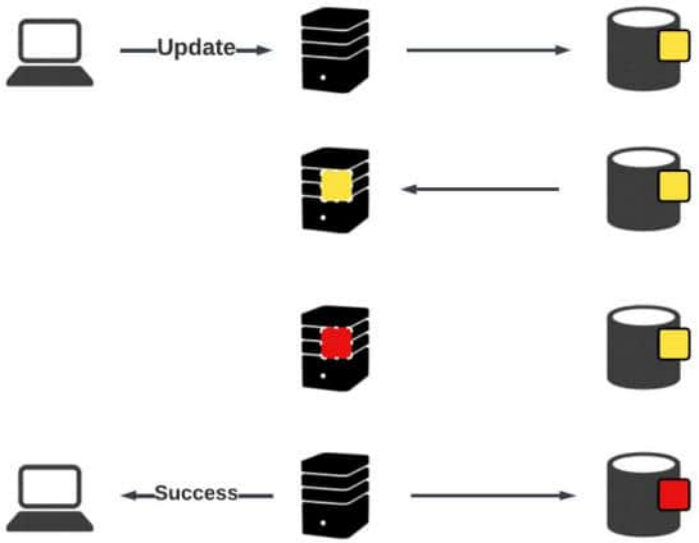

Рис. 2.1 – Процес зміни даних

При цьому, при виконанні одночасної зміни одного і того самого об’єкта у класичному підході, збережеться та зміна, що викоаналась останньою (Рис. 2.2).

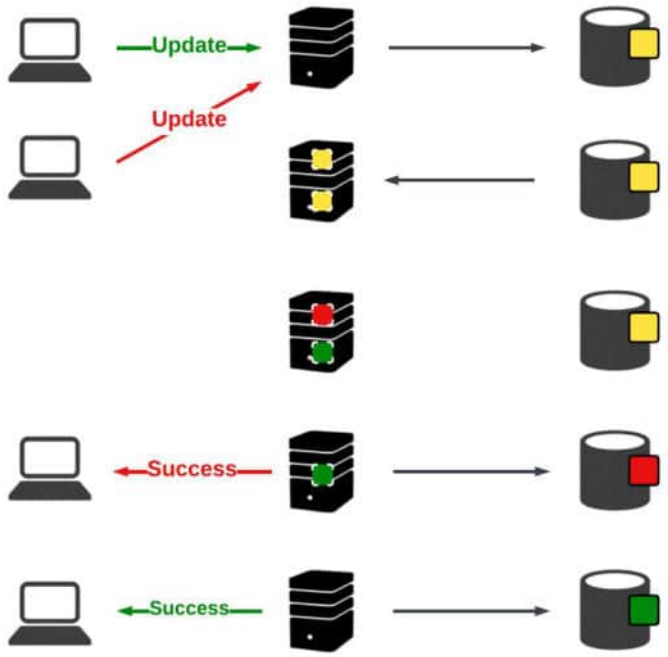

Рис. 2.2 – Одночасна зміна даних при збереженні поточного стану

Хоча у такому випадку ми втрачаємо одну із змін, цілісність даних при цьому не порушується.

Розглянемо, що станеться у тій самій ситуації у випадку, коли дані зберігаються у вигляді потоку подій, а операція зміни об'єкта полягає у додаванні нової події до потоку. Нехай ми змінюємо об’єкт, поточна версія якого – 2. Це означає, що його потік містить дві події. Обидва користувачі виконають операцію регідрації, завдяки чому кожен з них отримає репрезентацію об’єкта у пам’яті з версією 2. Після цього кожен користувач вносить свою зміну. Таким чином буде створено дві події з версією 3, які описуватимуть різні зміни. При збереженні змін першого користувача, його подія додається у сховище у наслідок чого, поточна версія об’єкта стає 3 і містить три події з версіями 1, 2 та 3. Врешті решт, другий користувач зберігає свою зміну, додаючи ще одну подію версії 3, та встановлюючи версію потоку рівною 3. В результаті ми

отримуємо потік версії 3, що містить 4 події з версіями 1, 2, 3 та 3. Бачимо, що в результаті виконання такої операції, порушується цілісність даних у сховищі, що є недопустимим (Рис. 2.3).

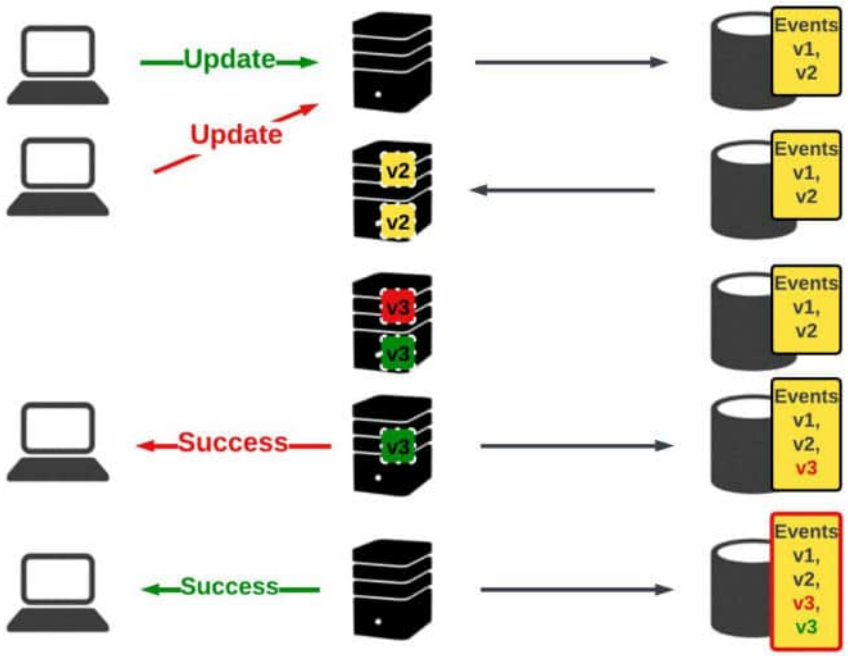

Рис. 2.3 – Одночасна зміна даних при збереженні подій

Ситуація що описана вище називається конфліктом паралелізму (eng. Concurrency Conflict). Способи вирішення таких проблем описані у наступному розділі.

2.2 Способи вирішення конфліктів паралелізму

Існує два основних способи вирішення конфліктів паралелізму [5, 6]. Перший спосіб називається Песимістичним, другий – Оптимістичним. Песимістичний спосіб полягає у блокуванні доступу до об’єкта з моменту читання, до моменту запису змін у базу даних. Таким чином перший користувач отримує ексклюзивний доступ до об’єкта, при цьому другий користувач зможе його прочитати лише після того, як перший збереже свої зміни у базу. Хоч такий підхід і має місце у загальному випадку,

в контексті нашої системи він має серйозні недоліки. Проблема полягає у тому, що операція зміни може займати значний проміжок часу, наприклад якщо для виконання зміни потрібно отримати додаткові дані із сторьоннього сервісу. Протягом усього цього часу, доступ до об’єкта буде неможливим, тому швидкодія такого способу буде на низькому рівні. Окрім того, такий спосіб є доволі складним у реалізації, особливо у нереляційних базах даних, які часто підтримують ACID транзакції лише на дуже обмеженому рівні. Через це застосування песимістичного підходу є обмеженим у Event Sourcing системах і в цій роботі не розглядається.

2.3 Оптимістичне вирішення конфліктів паралелізму

Як було згадано вище, одним із підходів до вирішення конфліктів паралелізму є оптимістичний підхід. Його основою є допущення про те, що такого роду конфлікти у системі виникатимуть рідко. Якщо це не так, то варто розглянути інші підходи. Ідея цього підходу в тому, що при збереженні змін до об’єкта ми також передаємо інформацію про очікуваний стан цього об’єкта. Очікуваний стан – це стан у якому знаходився об’єкт в той момент, коли ми його прочитали для виконання зміни. Під час збереження ми порівнюємо очікуваний стан із реальним. Якщо вони однакові, то це означає, що з моменту, коли ми прочитали об’єкт і до моменту запису жодних змін не відбулося. У цьому випадку наші зміни можуть бути збереженими. Якщо ж стани не співпадають, то це означає, що об’єкт був змінений іншим користувачем після того, як ми його прочитали. У цьому випадку збереження даних може порушити цілісність даних, тому користувачу повертається повідомлення про помилку з проханням повторити операцію.

Для визначення стану об’єкта використовується поле, що називається маркером паралелізму (eng. Concurrency token). Інформація у цьому полі змінюється кожного разу, коли відбувається зміна об’єкта. Для цього можна зберігати у ньому час зміни об’єкта у формі UNIX timestamp, випадкову стрічку, або версію, що інкрементується при кожному записі. В результаті, реалізація оптимістичного підходу зводиться до

операції *запис за умови*. Умовою в даному випадку є рівність очікуваного і реального маркеру паралелізму. Іншими словами, якщо значення очікуваного та реального маркеру паралелізму співпадають – виконуємо зміну даних. Якщо ні, повертаємо повідомлення про помилку. Важливо, щоб описана операція відповідала принципам ACID, оскільки в іншому випадку можлива ситуація, коли інший користувач змінить дані об’єкта після того, як було перевірено, що маркери паралелізму співпадають, але до того, як збережуться зміни.

У контексті Event Sourcing систем, операція запису за умови полягає у додаванні подій до потоку за умови, що поточна версія потоку співпадає з тією, якою вона була при зчитуванні для модифікації. **Можливість реалізації такої операції в базі даних є необхідною умовою для того, щоб її можна було використати в якості сховища подій.**

Варто також згадати про спосіб покращення досвіду користувача при використанні оптимістичного підходу у Event Sourcing системах. Як було описано вище, при виникненні конфлікту паралелізму, користувачу повертається помилка з проханням повторити дію. В якості додаткового рішення, можна перевірити чи конфліктують дві зміни з точки зору доменної моделі. Для цього підтримується реєстр подій, що суперечать чи не суперечать одна одній [7]. У випадку, якщо суперечності немає, ту саму зміну можна повторити автоматично. Наприклад, якщо ми намагаємось зберегти подію «Пароль змінено», а між тим хтось уже зберіг подію «Персональні дані змінено», тим самим спровокувавши конфлікт версій, ми можемо перевірити, що ці дві події не конфліктують між собою, та зберегти подію «Пароль змінено», змінивши при цьому її версію та очікувану версію. З іншої сторони, якщо ми хочемо внести подію «Замовлення скасовано», а між тим хтось уже вніс подію «Замовлення відправлено», то за допомогою реєстру конфліктуючих подій ми можемо перевірити, що ці дві події несумісні, та показати користувачу повідомлення про помилку.

2.4 Модель акторів

Альтернативним способом вирішення конфліктів, що виникають при одночасному доступі до об’єктів може бути їх недопускання. У цьому контексті у роботі також розглядається можливість застосування підходу під назвою модель акторів [8], який був описаним у 1973 році Карлом Хюітом і використовується для реалізації паралельних та розподілених систем без використання таких низькорівневих засобів, як критичні секції (lock section), а тому не призводить до таких проблем, як взаємне блокування чи простоювання потоків.

У цій моделі основним поняттям є поняття актора. Актором називають примітивну одиницю обчислень, що може виконувати дії у межах своєї області відповідальності. Поняття актора складається з трьох частин:

а) Стан – актор може володіти певною інформацією.
б) Процедури – актор містить засоби, що дозволяють змінювати його стан.
в) Комунікація – актори можуть взаємодіяти між собою за допомогою обміну повідомленнями.

Фундаментальною основою цієї моделі є те, що один актор в один момент часу може обробляти лише одне повідомлення, тобто кожен окремий актор опрацьовує свої повідомлення у послідовній манері. Спрощено, актора можна уявити як об’єкт в пам’яті, який підтримує певний стан і якому можна надсилати повідомлення для зміни цього стану. При цьому, повідомлення попадають в чергу повідомлень, звідки послідовно опрацьовуються.

Розглянемо, як можна застосувати цей підхід у контектсі Event Sourcing систем [9, 10, 11]. Якщо проаналізувати процеси, що описані в розділі 2.1, стає зрозуміло, що головною причиною виникнення конфліктів паралелізму є те, що зміни вносяться не до реальних даних, а до їх репрезентацій. Тим часом, з моменту, коли ми отримали репрезентацію у пам’яті і до моменту, коли ми її зберігаємо, реальні дані можуть бути зміненими іншими коритувачами, що і призводить до конфліктів паралелізму. Ідея полягає в тому, щоб перенести реальні дані із бази даних у пам’ять і працювати із

ними напряму. При цьому база даних буде використовуватись лише для зберігання даних на випадок, якщо дані у пам’яті буде втрачено (наприклад при перезавантаженні сервера). Модель акторів чудово підходить для реалізації описаної ідеї. Кожен об’єкт може бути представленим у вигляді актора. Тоді дані об’єкта будуть представлені як стан актора, а операції над станом будуть представлені у вигляді повідомлень, що обробляються об’єктом. При кожній зміні об’єкта актор також зберігатиме ці зміни у базу даних (у вигляді послідовності подій) для того, щоб стан можна було відновити при потребі. Описаний підхід зображено на рисунку 2.4. Така реалізація дає нам декілька вагомих переваг у порівнянні із іншими способами:

Перша перевага полягає у зменшенні вимог до баз даних. Оскільки у пам’яті міститься реальний об’єкт, а не його репрезентація, а усі повідомлення щодо зміни стану цього об’єкта обробляються послідовно – виключена можливість виникнення конфліктів паралелізму. Це знімає усю відповідальність за контроль паралелізму з бази даних, а значить у якості сховища подій може використовуватись будь-який тип баз даних.

Друга перевага полягає у збільшенні швидкодії системи. Оскільки реальний об’єкт одразу знаходиться у пам’яті, зникає необхідність виконувати операцію регідрації (яка раніше використовувалась для отримання репрезентації об’єкта) для того, щоб виконати зміни. Потенційно, це може значно збільшити швидкодію системи. Ця гіпотеза перевіряється далі в роботі. Зазначимо, що регідрація все ж потрібна при ініціалізації актора, тобто при першому звернені до об’єкта, коли його ще не має у пам’яті. Така ситуація може викникнути наприклад після перезавантаження сервера, та у деяких інших випадках, які описані далі.

На перший погляд, значним недоліком такої системи є високі вимоги до кількості оперативної пам’яті на сервері, оскільки дані про усі об’єкти переносяться саме туди. На практиці така проблема легко вирішується за допомогою обмеження часу життя актора. Для прикладу, ми можемо видаляти актора з пам’яті, якщо на протязі десяти хвилин до нього не надходило нових повідомлень. Таким чином,

вимоги до пам’яті у системі стають лише незначно більшими порівняно із іншими реалізаціями. При повторному зверненні до актора, його можна буде відновити з бази даних, використавши операцію регідрації.

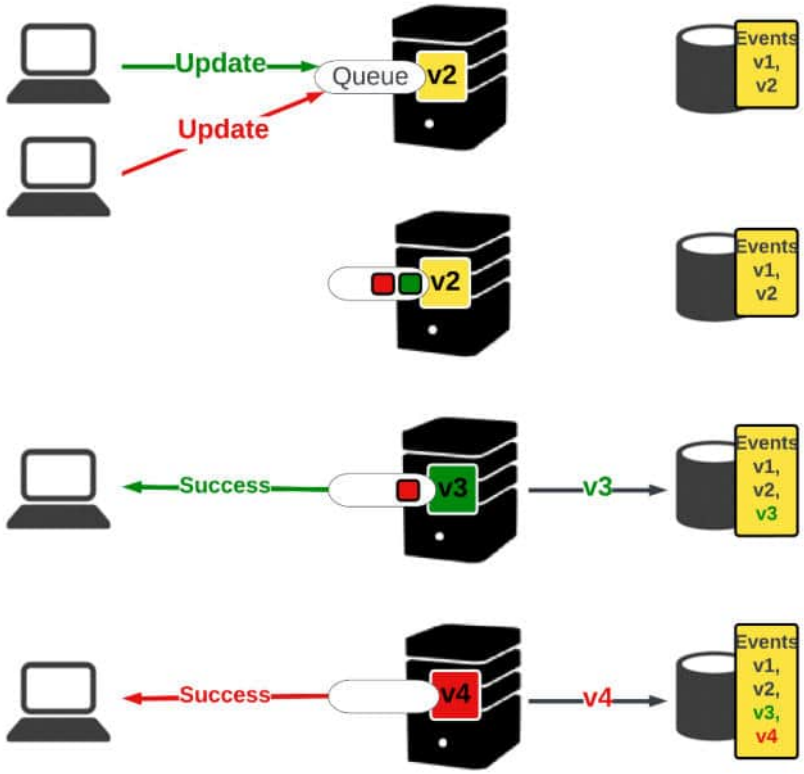

Рис. 2.4 – Одночасна зміна об’єкта при застосуванні моделі акторів

РОЗДІЛ 3. РЕАЛІЗАЦІЯ СХОВИЩА ПОДІЙ

3.1 Аналіз вимог

Із інформації, що наведена у попередніх розділах, можемо визначити наступні вимоги до сховища подій:

а) Повинна існувати можливість зберігати зміни у вигляді послідовності подій у хронологічному порядку.

б) Повинна існувати можливість зчитувати події, що стосуються одного об’єкта у хронологічному порядку.

в) У випадку, якщо не застосовується модель акторів, повинна існувати можливість реалізувати операцію запису за умови, що відповідає принципам ACID.

3.2 Абстракції сховища подій

Частиною цієї роботи є порівняння ефективності декількох реалізацій сховища подій. Для цього в коді було визначено декілька абстракцій, а також реалізацій за замовчуванням, які дозволять у зручній манері переключатися між різними сховищами подій. Розглянемо кожну з них детально.

Першою абстракцією є *IStore,* яка зображена у лістингу 3.1.

```
public interface IStore
{
    6 references
    Task AppendEventsAsync(EventStream stream, IEnumerable<Event> events);
    6 references
    Task<IEnumerable<Event>> GetEventsAsync(Guid streamId, long? fromVersion = null, long? toVersion = null);
    8 references
    Task<Snapshot> GetNearestSnapshotAsync(Guid streamId, long? version = null);
    7 references
    Task<EventStream> GetStreamAsync(Guid streamId);
    6 references
    Task SaveSnapshot(Snapshot snapshot);
}
```

Лістинг 3.1 – Абстракція *IStore*

Цей інтерфейс визначає абстракцію над сховищем подій та містить методи, які необхідні для роботи з ним. Метод *AppendEventsAsync* використовується для додавання подій до потоку. *GetEventsAsync* дозволяє отримати події, що належать до певного потоку, причому є можливість вказати мінімальну та максимальну версію події, яку ми хочемо отримати. Обмеження мінімальної версії використовується як частина реалізації механізму знімків і дозволяє зчитувати не усі події потоку, а лише ті, які відбулися після створення останнього знімку. Обмеження максимальної версії використовується тоді, коли потрібно отримати інформацію про стан об'єкта у певній версії. Метод *GetNearestSnapshot* використовується для отримання найближчого до певної версії знімка. *GetStreamAsync* можна використати для того, щоб отримати інформацію про потік за його ідентифікатором. І останній *SaveSnapshot* використовується для збереження знімку певної версії. У наступних розділах буде наведено реалізації інтерфейсу *IStore* для реляційної та нереляційної бази даних.

Ще однією абстракцією дещо вищого рівня є *IEventStore* (лістинг 3.2). Вона визначає операції для роботи з доменними сутностями, та визначає операції регідрації сутності із потоку подій, а також збереження сутності у вигляді потоку подій.

```
public interface IEventStore
{
    10 references
    Task<TAggregate> Rehydrate<TAggregate>(Guid streamId, long? version = null) where TAggregate : IAggregate;
    17 references
    Task Store<TAggregate>(TAggregate aggregate) where TAggregate : IAggregate;
}
```

Лістинг 3.2 – Абстракція *IEventStore*

У системі створена стандартна реалізація *IEventStore* під назвою *DefaultEventStore,* проте перед тим, як перейти до неї, розглянемо ще одну допоміжну абстракцію для кращого розуміння згаданої реалізації.

Інтерфейс *ISnapshotPredicate* (лістинг 3.3) використовується для реалізації різних стратегій створення знімків у системі. Він визначає лише один метод, який приймає інформацію про контекст, у якому використовується і на її основі приймає рішення про необхідність створення знімку. Інформація контекст містить час створення останнього знімку, версію останнього знімку, та іншу інформацію, яку можна використати для прийняття рішення.

```
public interface ISnapshotPredicate
{
    3 references
    bool ShouldTakeSnapshot(SnapshotContext context);
}
```

Лістинг 3.3 – Абстракція *ISnapshotPredicate*

У системі існує реалізація цього інтерфейсу під назвою *EachNEventsSnapshotPedicate* (лістинг 3.4), яку можна використовувати для створення знімків кожні *N* подій.

```
public class EachNEventsSnapshotPredicate : ISnapshotPredicate
{
    private readonly int _n;

    2 references
    public EachNEventsSnapshotPredicate(int n)
    {
        _n = n;
    }

    3 references
    public bool ShouldTakeSnapshot(SnapshotContext context)
    {
        return context.Aggregate.Version % _n == 0;
    }
}
```

Лістинг 3.4 – Приклад предикату для створення знімків

Розглянемо тепер реалізацію *DefaultEventStore* більш детально. У лістингу 3.5 наведена структура цього класу, а також список його залежностей.

```
internal class DefaultEventStore : IEventStore
{
    private readonly IStore _store;
    private readonly IEnumerable<IProjection> _projections;
    private readonly EventStoreOptions _eventStoreOptions;

    0 references
    public DefaultEventStore(IStore store, IEnumerable<IProjection> projections, EventStoreOptions eventStoreOptions)
    {
        _store = store;
        _projections = projections;
        _eventStoreOptions = eventStoreOptions;
    }

    8 references
    public async Task<TAggregate> Rehydrate<TAggregate>(
        Guid aggregateId,
        long? version = null) ...

    15 references
    public async Task Store<TAggregate>(
        TAggregate aggregate) ...

    1 reference
    private async Task<EventStream> GetStream<TAggregate>(TAggregate aggregate, long initialVersion) ...

    1 reference
    private async Task HandleShnapshots<TAggregate>(TAggregate aggregate, List<Event> eventsToStore) ...

    1 reference
    private void HandleProjections(IEnumerable<object> events)...
}
```

Лістинг 3.5 – Структура класу *DefaultEventStore*

Бачимо, що клас реалізує методи *Rehydrate* та *Store,* що визначені в інтерфейсі *IEventStore.* Окрім того, містить деякі допоміжні приватні методи, які буде описано згодом. Важливо звернути увагу на те, що реалізація цього класу використовує *IStore* як одну із своїх залежностей. Передаючи цьому класу різні реалізації *IStore* можна легко змінювати тип сховища подій, що використовується у системі. Окрім того, у конструктор передається колекція *IProjection,* які є обробниками подій, що використовуються для реалізації незалежної моделі для читання (у принципі CQRS), а також об’єкт *EventStoreOptions,* який дозволяє налаштувати спосіб серіалізації подій та створення знімків. Розглянемо реалізацію *DefaultEventStore* більш детально. У лістингу 3.6 наведено реалізацію методу *Store.*

```
55      public async Task Store<TAggregate>(
56          TAggregate aggregate) where TAggregate : IAggregate
57      {
58          var events = aggregate.DequeueUncommittedEvents();
59          var initialVersion = aggregate.Version - events.Count();
60
61          var eventStream = await GetStream(aggregate, initialVersion);
62
63          var currentEventVersion = initialVersion;
64
65          var eventsToStore = events.Select(x => new Event
66          {
67              StreamId = aggregate.Id,
68              Type = x.GetTypeName(),
69              Data = _eventStoreOptions.EventSerializer.Serialize(x),
70              Version = ++currentEventVersion,
71              Stream = eventStream
72          }).ToList();
73
74          eventStream.Version = currentEventVersion;
75
76          await _store.AppendEventsAsync(eventStream, eventsToStore);
77
78          HandleProjections(events);
79          await HandleShnapshots(aggregate, eventsToStore);
80      }
81
        1 reference
82      private async Task<EventStream> GetStream<TAggregate>(TAggregate aggregate, long initialVersion)
83              where TAggregate : IAggregate
84      {
85          var eventStream = await _store.GetStreamAsync(aggregate.Id);
86
87          if (eventStream != null && eventStream.Version != initialVersion)
88          {
89              throw new EventStoreConcurrencyException();
90          }
91
92          eventStream ??= new EventStream()
93          {
94              StreamId = aggregate.Id,
95          };
96          return eventStream;
97      }
```

Лістинг 3.6 – Реалізація методу *Store* у класі *DefaultEventStore*

Як було згадано раніше, метод *Store* використовується для збереження об’єкта у вигляді послідовності подій. На 58 рядку ми отримуємо список усіх змін, що ще не були збережені у сховищі. Далі використовується метод *GetStream* для того, щоб отримати інформацію про потік з бази даних. Звернімо увагу на перевірку на рядку

87. Якщо потік існує, проте його версія не співпадає з версією нашого об’єкта до модифікації, ми повертаємо помилку. Ця перевірка є оптимізацією оптимістичної моделі паралелізму і використовується для того, щоб не відправляти дані до бази даних, якщо ми заздалегідь знаємо, що потік було змінено. Зазначимо, що ця оптимізація не виключає необхідності виконувати аналогічну перевірку на стороні бази даних, оскільки потік усе ще можуть змінити у проміжку між тим, як ми його прочитали у рядку 85 та зберегли нові події у рядку 76. Після отримання потоку, ми записуємо змінні у форматі, в якому вони будуть збережені до бази даних. При цьому виконується серіалізація даних події та присвоєння послідовних версій (рядки 65 - 72). Далі змінюється версія потоку і усі дані зберігаються у сховище подій (рядки 74 - 76). Метод *HandleProjections* викликає обробники подій та виконує зміни даних у моделі для читання. Врешті-решт, викликається метод *HandleSnapshots,* який за потреби створює знімок нашого об’єкта. Його реалізація наведена на лістингу 3.7.

```
99  private async Task HandleShnapshots<TAggregate>(TAggregate aggregate, List<Event> eventsToStore)
100             where TAggregate : IAggregate
101 {
102     if (_eventStoreOptions.SnapshotPredicate == null) return;
103
104     var latestSnapshot = await _store.GetNearestSnapshotAsync(aggregate.Id);
105
106     var snapshotContext = new SnapshotContext(
107         aggregate,
108         eventsToStore,
109         latestSnapshot?.CreatedAt,
110         latestSnapshot?.Version);
111
112     if (!_eventStoreOptions.SnapshotPredicate.ShouldTakeSnapshot(snapshotContext)) return;
113
114     var snapshot = new Snapshot
115     {
116         StreamId = aggregate.Id,
117         Version = aggregate.Version,
118         Data = _eventStoreOptions.EventSerializer.Serialize(aggregate)
119     };
120
121     await _store.SaveSnapshot(snapshot);
122 }
```

Лістинг 3.7 – Реалізація методу *HandleSnapshots*

На початку методу відбувається перевірка, чи налаштований механізм знімків для нашої системи. Далі, за допомогою метода *GetNearestSnapshotAsync* отримуємо дані про останній знімок об’єкта. Далі створюється об’єкт *SnapshotContext,* і приймається рішення про необхідність створення змінку. Якщо рішення позитивне, то стан об’єкта серіалізується і знімок зберігається у базі даних.

Тепер, маючи певне уявлення про те, як реалізоване збереження об’єкта, розглянемо як відбувається відновлення поточного стану із потоку подій.

```
22  public async Task<TAggregate> Rehydrate<TAggregate>(
23      Guid aggregateId,
24      long? version = null) where TAggregate : IAggregate
25  {
26      var stream = await _store.GetStreamAsync(aggregateId);
27      if (stream == null) return default;
28
29      var latestSnapshot = _eventStoreOptions.SnapshotPredicate == null
30          ? null
31          : await _store.GetNearestSnapshotAsync(aggregateId, version);
32
33      var aggregate = latestSnapshot == null
34          ? Activator.CreateInstance<TAggregate>()
35          : (TAggregate)_eventStoreOptions.EventSerializer.Deserialize(latestSnapshot.Data, typeof(TAggregate));
36
37      var events = await _store.GetEventsAsync(aggregateId, latestSnapshot?.Version + 1 ?? null, version);
38
39      events.ToList().ForEach(e =>
40      {
41          var domainEvent = _eventStoreOptions.EventSerializer.Deserialize(
42              e.Data,
43              TypeHelper.GetType(e.Type)
44          );
45
46          aggregate.When(domainEvent);
47      });
48
49      var aggregateVersion = events.Any() ? events.Max(x => x.Version) : latestSnapshot.Version;
50      aggregate.SetVersion(aggregateVersion);
51
52      return aggregate;
53  }
```

Лістинг 3.8 – Реалізація операції регідрації у класі *DefaultEventStore*

На початку отримуємо інформацію про потік із вказаним ідентифікатором. Якщо такого потоку не існує, значить об’єкту не існує і метод повертає значення за замовчуванням для типу об’єкта. Далі, якщо механізм знімків налаштований, пробуємо знайти знімок, що якнайближче знаходиться до версії об’єкта, яку ми

хочемо отримати. Якщо такий знімок знайдено, десеріалізуємо його і отримуємо стан об’єкта в моменту створення цього знімка. Якщо ж ні – створюємо новий об’єкт у пам’яті. Далі, завантажуємо у пам’ять потрібні події. Для цього в залежності від того, чи існує знімок, завантажуємо або усі події від початку існування об’єкта, або лише ті, які є новішими за останній знімок. Далі десеріалізуємо дані кожної події та виконуємо зміну стану об’єкта за допомогою методу *When.* Реалізацію методу *When* для класу, який було наведено у лістингу 1.1 наведено у лістингу 3.9.

```
public override void When(object @event)
{
    switch (@event)
    {
        case TemperatureMeasurementStarted x:
            Apply(x);
            break;
        case TemperatureRecorded x:
            Apply(x);
            break;
    }
}
```

Лістинг 3.9 – Реалізація методу *When* агрегату *TemperatureMeasurement*

Бачимо, що реалізація є достатньо простою і в залежності від типу події, яку ми обробляємо, викликається відповідний метод *Apply,* що і виконує зміну стану об’єкта згідно з інформацією у події (лістинг 1.1).

Врешті решт, визначається поточна версія об’єкта і агрегат повертається користувачу. Таким чином, *DefaultEventStore* містить код для реалізації Event Sourcing системи, який є незалежним від типу сховища подій. У наступних розділах, розглянемо реалізації *IStore,* які інкапсулюють роботу з різними типами сховищ.

3.3 Реалізація сховища подій у реляційній базі даних

Розглянемо тепер реалізацію сховища подій, що використовує реляційну модель даних. Як було пояснено вище, для цього у системі потрібно створити реалізацію інтерфейсу *IStore,* що працюватиме із SQL базою даних [12]. У якості реляційної бази даних було використано MS SQL Server. Реалізація містить три таблиці: *EventStreams*, *Events* та *Snapshots*

Таблиця *Events* зберігає події, що описують зміни сутностей. Кожен запис у таблиці описує одну подіїю. Таблиця містить наступні колонки:

а) **Id:** унікальний ідентифікатор запису (сурогатний ключ);
б) **StreamId:** ідентифікатор об’єкта, якого стосується подія;
в) **Created:** час створення події;
г) **Type:** назва/тип події;
д) **Data:** серіалізована інформація про зміни;
е) **Version:** версія події;

Таблиця *EventStreams* містить інформацію про об’єкти у системі. Для кожного об’єкта повинен існувати запис у цій таблиці. Окрім того, вона містить денормалізовану поточну версію кожного об’єкта. Загалом цю інформацію можна отримати і з таблиці з подіями, проте через її великий розмір, це було б не надто ефективно. *EventStreams* містить наступні колонки:

а) **StreamId:** ідентифікатор об’єкта у системі;
б) **Version:** актуальна версія об’єкта;
в) **Timestamp:** час останньої зміни об’єкта. Використовується для реалізації оптимістичної моделі паралелізму;

Врешті решт, таблиця *Snapshots* використовується для зберігання знімків, тобто стану об’єкта в певний момент часу. Її структура є аналогічною до таблиці *Events,* за винятком того, що відсутня колонка *Type* і поле *Data* містить серіалізований стан, а не інформацію про зміни.

Приклад даних, що можуть зберігатися у цих таблицях наведено у таблицях 3.1, 3.2 і 3.3.

Таблиця 3.1 – таблиця *EventStreams*

StreamId	Version	Timestamp
5584DDC547AE	2	1665520249
C737DA3567B2	1	1665510003

Таблиця 3.2 – таблиця *Events*

Id	StreamId	Created	Type	Data	Version
1	5584DDC547AE	16.02.2000 10:11:12	UserCreated	name=’Denys’	1
2	5584DDC547AE	16.02.2000 12:13:14	PersonalInfoUpdated	age=22	2
3	C737DA3567B2	16.02.2000 14:15:16	UserCreated	name=’Petro’	1

Таблиця 3.3 – таблиця *Snapshots*

Id	StreamId	Created	Data	Version
1	5584DDC547AE	16.02.2000 12:13:14	name=’Den’; age=22	2

Розглянемо тепер реалізацію методів *IStore,* що використовує описану модель. Усі методи напряму виконують SQL запити, використовуючи бібліотеку для .Net під назвою Dapper, тому для простоти сприйняття наведемо лише їх.

```
1
2   SELECT *
3   FROM Events
4   WHERE StreamId = @streamId
5       AND (@fromVersion IS NULL OR Version >= @fromVersion)
6       AND (@toVersion IS NULL OR Version <= @toVersion);
7
```

Лістинг 3.10 – Реалізація *GetEventsAsync*

SQL скрипт, що використовується в методі *GetEventsAsync* наведено в лістингу 3.10. Реалізація є достатньо простою та очевидною: виконується фільтрація по

індентифікатору об’єкта, який ми хочемо отримати, а також при наявності параметрів *fromVersion* та *toVersion,* здійснюється фільтрація версій подій, які нам потрібно отримати.

```
1
2    SELECT *
3    FROM EventStreams
4    WHERE StreamId = @streamId
5
```

Лістинг 3.11 – Реалізація *GetStreamAsync*

Реалізація методу *GetStreamAsync,* що використовується для отримання інформації про об’єкт та його потік подій є також очевидною (лістинг 3.11).

```
1
2    SELECT TOP(1) *
3    FROM Snapshots
4    WHERE StreamId = @streamId AND (@version IS NULL OR Version <= @version)
5    ORDER BY Version DESC;
6
```

Лістинг 3.12 – Реалізація *GetNearestSnapshotAsync*

Реалізацію *GetNearestSnapshotAsync* наведено в лістингу 3.12. Тут відбувається фільтраціїя по ідентифікатору об’єкта, а також при наявності параметра *version,* вибираємо ті знімки, версія яких є не більшою ніж параметр *version.* Після цього відбувається сортування по зростанюю та вибирається перший елемент отриманої множини. Таким чином отримуємо найближчий до заданої версії знімок.

```
1
2    INSERT INTO Snapshots(StreamId, Data, Version, CreatedAt)
3    VALUES (@StreamId, @Data, @Version, @CreatedAt);
4
```

Лістинг 3.13 – Реалізація *SaveSnapshot*

Реалізація методу *SaveSnapshots* зводиться до виконання операції вставки і є очевидною.

```
1
2   DECLARE @StreamTimestamp ROWVERSION;
3
4   SELECT @StreamTimestamp = Timestamp
5   FROM EventStreams WITH (UPDLOCK, ROWLOCK)
6   WHERE StreamId = @StreamId;
7
8   IF @StreamTimestamp IS NOT NULL AND @StreamTimestamp != @Timestamp
9       THROW 424242, 'Concurrency check failed', 1;
10
11  IF @StreamTimestamp IS NULL
12      INSERT INTO EventStreams(StreamId, Version)
13      VALUES (@StreamId, @Version);
14  ELSE
15      UPDATE EventStreams
16      SET Version = @Version
17      WHERE StreamId = @StreamId;
18
19  INSERT INTO Events(StreamId, Created, Type, Data, Version)
20  VALUES (...),
21      (...),
22      (...);
23
```

Лістинг 3.14 – Реалізація *AppendEventsAsync*

Найбільш складною є реалізація методу *AppendEventsAsync,* який виконує додавання нових подій об’єкта. Розглянемо детальніше скрипт, що наведено в лістингу 3.14. Тут реалізована операція запису за умови, про яку йшла мова раніше. У рядках 2-6 записуємо поточне значення у колонці *timestamp* об’єкта з вказаним ідентифікатором. У 8 рядку виконується перевірка чи значення існує і чи рівне воно очікуваному (очікуване значення передається за допомогою SQL параметра *@Timestamp)*. Нагадаємо, що очікуване значення, це значення, яке було в цього об’єкта в момент читання. Далі, якщо значення не співпадають, повертаємо користувачеві помилку, оскільки це означає, що запис був змінений після читання.

Зауважимо, що значення *@StreamTimestamp* може бути рівним *NULL* лише у випадку, коли такий об'єкт не існує. Користуючись цим фактом, у рядках 11-17, створюємо новий запис у таблиці *EventStream,* або модифікуємо існуючий, змінюючи його версію. При виконанні цих операцій, MSSQL Server автоматично змінює значення у колонці *Timestamp,* забезпечуючи правильне функціонування операції запису за умови. Врешті решт, у рядку 19 відбувається запис усіх подій у таблицю *Events*. Варто звернути увагу на те, що операція *INSERT* в MSSQL Server дозволяє одночасний запис лише 999 рядків. Якщо за одну операцію додаватиметься більше ніж 999 рядків, потрібно виконувати деілька операцій *INSERT* в межах однієї транзакції. Зазначимо, що описаний скрипт повинен виконуватися в межах SQL транзакції. Для забезпечення ACID властивостей у контексті скрипта у лістингу 3.14, достатнім рівнем ізоляції транзакції є *READ COMMITTED* (оскільки виникнення фантомних читань у цьому скрипті неможливе). Транзакція ініціюється та завершується інструментами .Net (лістинг 3.15).

```
try
{
    using (var transaction = _context.Connection.BeginTransaction())
    {
        var result = await _context.Connection.ExecuteAsync(
            finalSql,
            new {
                stream.StreamId,
                stream.Version,
                stream.Timestamp
            },
            transaction);

        transaction.Commit();
    }
}
catch(SqlException ex) when (ex.ErrorCode == 424242)
{
    throw new EventStoreConcurrencyException();
}
```

Лістинг 3.15 – Спосіб виконання скрипта з лістингу 3.14

Звернімо увагу на запис у 5 рядку лістингу 3.14: *WITH (UPDLOCK, ROWLOCK).* Перед тим, як пояснити його функцію, розглянемо проблему, що виникає при його відсутності. Уявімо, що одночасно виконується дві транзакції, що змінюють один і той самий об'єкт. Можлива наступна ситуація. *Транзакція 1* виконала перевірку у рядках 4-9 і наступним кроком буде виконувати зміну у рядку 15. При цьому *транзакція 2* виконуючи перевірку у рядках 4-9 також успішно її проходить, оскільки транзакція 1 іще не встигла зберегти свої зміни. Далі обидві транзакції додають свої події і виконують *COMMIT,* внаслідок чого отримуємо декілька подій з однаковими версіями у сховищі, що порушує цілісність даних. Проаналізувавши описану ситуацію, розуміємо, що проблема виникає тому, що *транзакція 2* може прочитати значення *@StreamTimestamp* після того, як це зробила *транзакція 1*, не зважаючи на те, що *транзакція 1* планує змінити значення цього поля. Така поведінка не суперечить ідеї транзакції, проте є недопустимою у нашій системі. Саме для вирішення цієї проблеми і використовується інструкція *WITH (UPDLOCK, ROWLOCK).* Вона описує намір змінити дані, які ми прочитали у виразі *SELECT.* Говорячи термінами MSSQL Server, для запису, який ми прочитали у рядках 4-6, буде отримано *UPDATE LOCK* [13]. У цьому випадку, після того як *транзакція 1* отримала *UPDATE LOCK* для рядка з ідентифікатором *@StreamId,* спроба отримати *UPDATE LOCK* для того самого рядка в *транзакції 2* призведе до її блокування доти, доки *транзакція 1* не завершиться. Зауважимо, що наявність *UPDATE LOCK* на рядку НЕ блокує читання цього рядка без наміру його змінювати. Тобто операція в лістингу 3.16 виконається успішно при наявності *UPDATE LOCK* на рядку. При цьому, операції, що передбачають зміну рядка (лістинг 3.17) будуть заблоковані до завершення транзакції, що утримує *UPDATE LOCK.* Така реалізація дозволяє уникнути непотрібних блокувань у тих випадках, коли інформація про потік отримується лише з метою читання.

```
1
2    SELECT *
3    FROM EventStreams
4    WHERE StreamId = @StreamId;
5
```

Лістинг 3.16 – Читання без наміру змінювати

```
1
2    SELECT *
3    FROM EventStreams WITH(UPDLOCK, ROWLOCK)
4    WHERE StreamId = @StreamId;
5
6    UPDATE EventStreams
7    SET Version = 42
8    WHERE StreamId = @StreamId;
9
```

Лістинг 3.17 – Операції, що передбачають зміну рядка

Таким чином, ми отримали увесь необхідний набір операції для використання SQL бази даних у якості сховища подій.

3.4 Реалізація сховища подій у нереляційній базі даних

Розглянемо реалізацію сховища подій з використанням NoSQL бази даних. У якості такої бази даних було обрано документно-орієнтовану базу даних MongoDB. Вибір документно-орієнтованої бази пояснюється тим, що сфера їх застосування у великій мірі схожа зі сферою застосування реаляційних баз даних, на противагу іншим типам нереляційних баз даних, таких як графові чи колонкові, сфера застосування яких є вузькоспеціалізованою. Окрім того, MongoDB є однією з найбільш популярних документно-орієнтованих баз даних, тому надалі робота відбуватиметься саме з нею.

Перше, що потрібно зрозуміти для використання MongoDB у якості сховища подій – це яким чином події будуть у ній зберігатися . Перша опція полягає у збереженні кожної події у вигляді окремого документа в колекці. Тоді подія буде

містити версію та ідентифікатор об’єкта, до якого вона належить. Цей підхід практично аналогічний тому, який ми використовували у реліційній базі даних. Друга опція полягає у збереженні подій у вигляді внутрішнього масиву документа, що описує об’єкт [14]. Таким чином у системі існуватиме колекція з об’єктами, структура яких наведена у лістингу 3.18. Аналогічним чином можемо зберігати і знімки.

```
{
    "StreamId": "12345",
    "Version": 1,
    "Events": [
        {
            "Created": "16.02.2000 16:45:01",
            "Type": "DataRecorded",
            "Data": "Serialized Event Data",
            "Version": 1
        }
    ]
}
```

Лістинг 3.18 – спосіб збереження подій у документі

Оскільки MongoDB (як і більшість документо-орієнтованих баз даних) підтримує ефективні ACID транзакції лише на рівні одного документа [15], то нашим потребам відповідає лише друга опція. Зауважимо, що максимальний розмір документа в MongoDB не може перевищувати 16 мегабайт, що потрібно враховувати при прийнятті рішення про використання того чи іншого підходу.

Тепер, коли ми визначились з підходом до збереження даних, розглянемо реалізацію деяких методів інтерфейса *IStore.* Для взаємодії з MongoDB використовується офіційна бібліотека для .Net – MongoDB Driver. Окрім того, класи *Event, Snapshot* та *EventStream*, що відповідають документам у базі даних, дещо відрізняються від тих, які використовуються системою, тому створено додаткові методи для мапінгу структури класів MongoDB у структуру, що вимагається інтерфейсом *IStore.*

У лістингу 3.19 наведено реалізацію методу *GetStreamAsync*. Він використовує приватний метод *GetStreamAsync* (лістинг 3.20) для отримання даних про потік, а також виконує мапінг структури сутності бази даних у доменну сутність.

```
public async Task<EventStream> GetStreamAsync(Guid streamId)
{
    var eventStream = await GetStreamAsync(streamId, includeEvents: false);
    return eventStream?.ToDomain();
}
```

Лістинг 3.19 – Реалізація публічного методу *GetStreamAsync*

```
106  private async Task<Entities.EventStream> GetStreamAsync(Guid streamId, bool includeEvents)
107  {
108      var collection = GetEventStreamsCollection();
109
110      var filter = Builders<Entities.EventStream>.Filter.Eq(x => x.StreamId, streamId);
111      var options = new FindOptions<Entities.EventStream>();
112      var projection = Builders<Entities.EventStream>.Projection.Exclude(x => x.Snapshots);
113
114      if (!includeEvents)
115      {
116          projection = projection.Exclude(x => x.Events);
117      };
118
119      options.Projection = projection;
120
121      var eventStreamCollection = await collection.FindAsync(filter, options);
122      var eventStream = await eventStreamCollection.FirstOrDefaultAsync();
123
124      return eventStream;
125  }
```

Лістинг 3.20 – Реалізація приватного методу *GetStreamAsync*

Проаналізуємо детальніше реалізацію приватного методу *GetStreamAsync*. У рядку 108 ми отримуємо об’єкт для доступу до колекції MongoDB. Після цього створюємо фільтр, який шукає документи, значення поля *StreamId* яких рівне *streamId*. Пам'ятаємо, що події та знімки знаходяться всередині документа, тому для збільшення ефективності завантаження даних, у рядках 111-119 створюємо проекцію, яка виключить знімки з результуючого об'єкта, а також виключить події у випадку, якщо агрумент *includeEvents* є рівним *false*. Надалі буде показано, як і для чого

застосовується цей параметр. Врешті решт, ми відправляємо створений запит до бази даних і повертаємо інформацію про потік.

Розглянемо тепер реалізацію методу завантеження подій для об'єкта. Вона наведена у лістингу 3.21

```
47  public async Task<IEnumerable<Event>> GetEventsAsync(Guid streamId, long? fromVersion = null, long? toVersion = null)
48  {
49      var collection = GetEventStreamsCollection();
50
51      IEnumerable<Entities.Event> events = null;
52
53      if(fromVersion == null && toVersion == null)
54      {
55          var stream = await GetStreamAsync(streamId, true);
56          events = stream?.Events;
57      }
58      else
59      {
60          var stream = await collection.Aggregate()
61              .Match(x => x.StreamId == streamId)
62              .Unwind<Entities.EventStream, EventStreamEventFlattened>(x => x.Events)
63              .Match(x => (fromVersion == null || x.Events.Version >= fromVersion)
64                          && (toVersion == null || x.Events.Version <= toVersion))
65              .Group(x => x.StreamId,
66                     x => new FilteredEventsCollection
67                     {
68                         StreamId = x.Key,
69                         Events = x.Select(e => e.Events)
70                     })
71              .FirstOrDefaultAsync();
72
73          events = stream?.Events;
74      }
75
76      return events?.Select(x => x.ToDomain()).ToList() ?? Enumerable.Empty<Event>();
77  }
```

Лістинг 3.21 – Реалізація методу *GetEventsAsync*

Існує два можливих способи використання цього методу: без фільтрації подій за версією, та з ним. У першому випадку для завантаження усіх подій об’єкта використовується метод *GetStreamAsync* (лістинг 3.20) з параметром *includeEvents = true.* Таким чином ми завантажуємо цілий документ, який містить колекцію усіх подій. Більш складним є випадок, коли потрібно виконати фільтрацію подій за версіями. MongoDB не має готових інструментів для того, щоб вибрати лише деякі елементи з внутрішньої колекції (не варто плутати з фільтрацією документів по елементах внутрішніх колекцій), тому тут використовується aggregation pipeline, яка послідовно застосовує перетворення до документа. Розглянемо кожен крок більш детально. Використаємо для цього інструмент, що доступний в застосунку MongoDB

Compas. На рисунку 3.1 зображено результат виконання операції *match,* яка виконує фільтрацію документів.

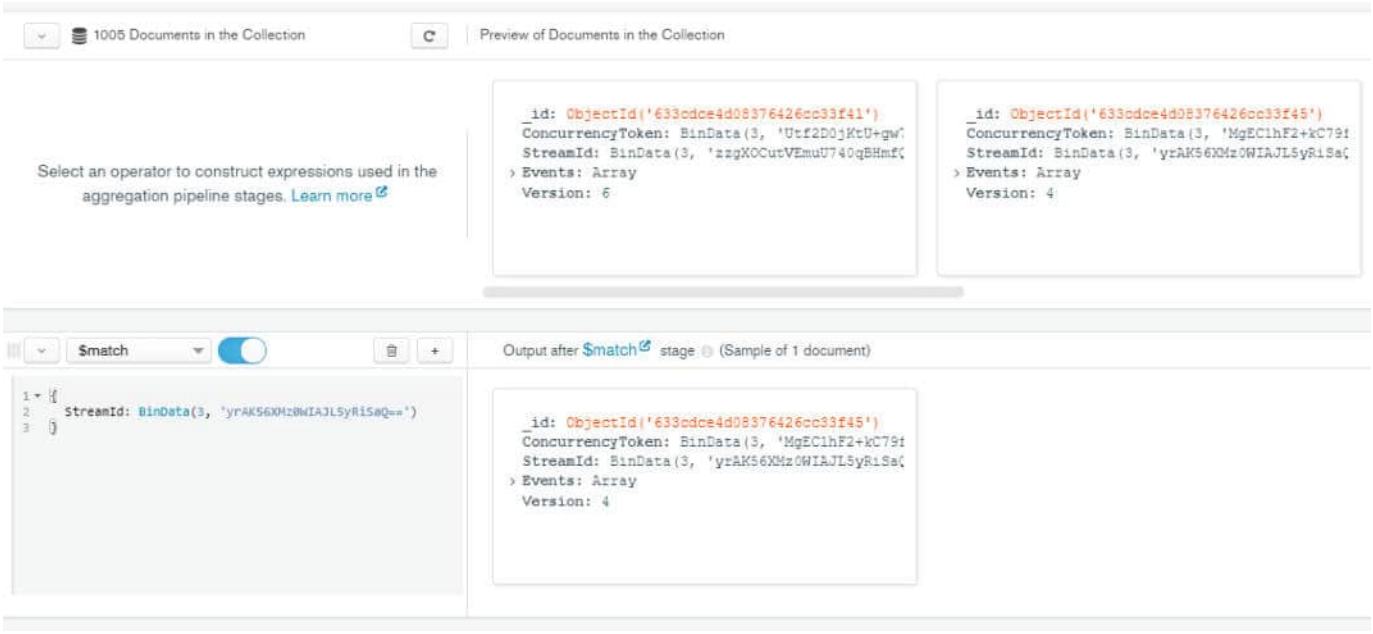

Рис. 3.1 – Результат операції *match*

Наступним кроком є операція *unwind* (рис. 3.2), яка для кожного елемента внутрішньої колекції створює окремий документ, тобто виконує сплющування (eng. flatten).

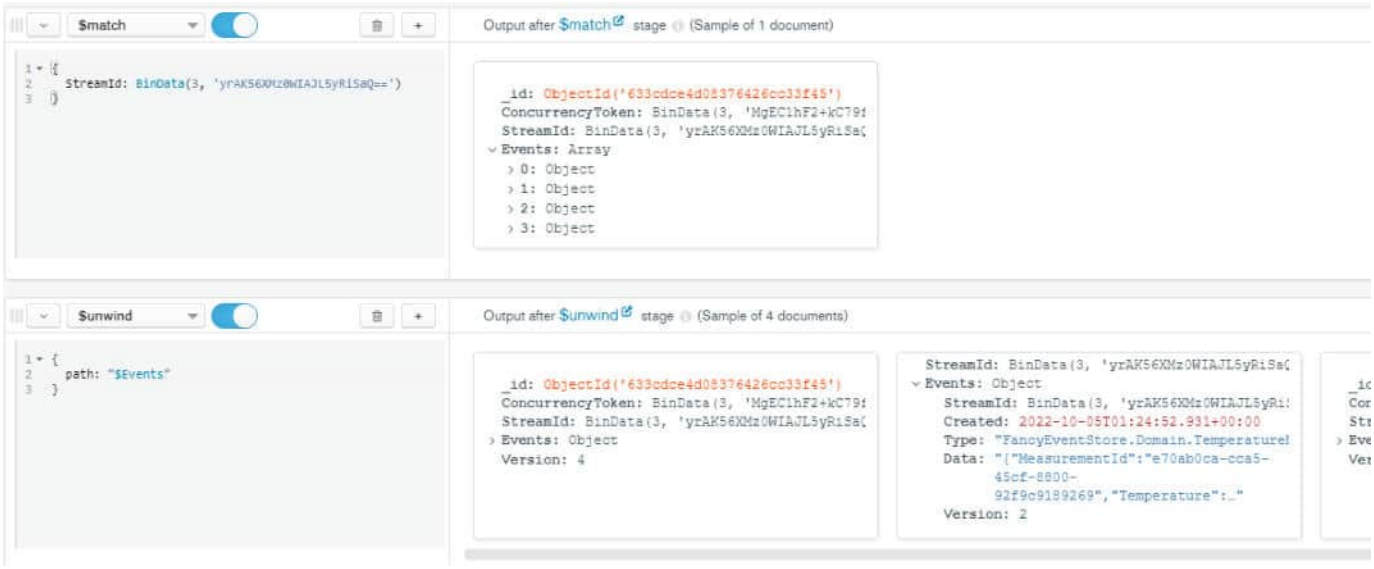

Рис. 3.2 – Результат операції *unwind*

Далі за допомогою оператора *match* виконується фільтрація по версіях подій (рис. 3.3) (тепер ми це можемо зробити, оскільки подіє є внутрішніми об’єктами документа, а не елементами внутрішнього масиву).

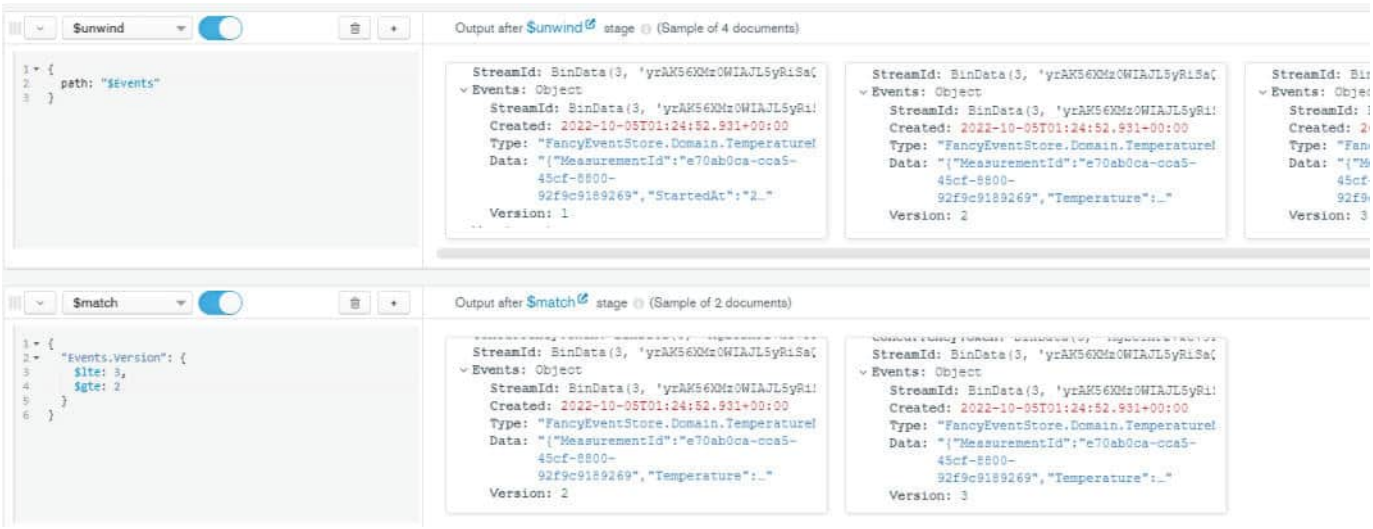

Рис. 3.3 – Фільтрація за версією подій

Останнім кроком ми виконуємо операцію групування (рис. 3.4), яка у цьому контексті є протилежною операцією до *unwind*.

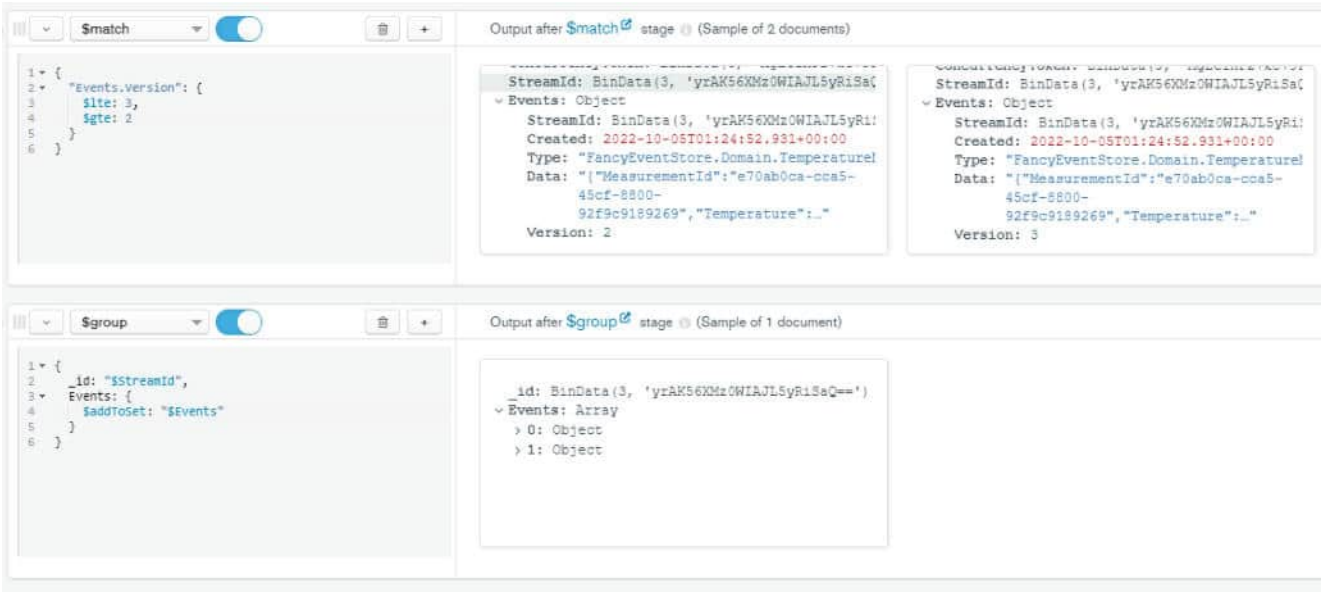

Рис. 3.4 – Результат групування

Таким чином ми отримуємо відфільтровані за версією події. Повертаючись до лістингу 3.21 стає зрозуміло, що різний спосіб завантаження подій в залежності від необхідності фільтрації використовується з метою покращення ефективності. Aggregation pipeline могла б підійти і для випадку, коли фільтрація не потрібна, проте тоді доводилось би виконувати багато зайвих операцій.

Для збереження знімків було реалізовано два підходи. Перший підхід полягає у збереженні знімків у внутрішньому масиві документа, по аналогії до того, як реалізоване збереження подій. Проте, оскільки збереження знімків не потребує ACID транзакцій, можемо також збегірати їх у окремій колекцій, по аналогії до реляційної моделі.

У першому випадку, метод для отримання найближчого знімку працює схожим чином до отримання подій певної версії (лістинг 3.22). Відмінність полягає у тому, що після фільтрації за версією ми не виконуємо групування, а сортуємо за спаданням версії і завантажуємо лише перший елемент колекції.

```
79  public async Task<Snapshot> GetNearestSnapshotAsync(Guid streamId, long? version = null)
80  {
81      var collection = GetEventStreamsCollection();
82
83      var snapshot = await collection.Aggregate()
84          .Match(x => x.StreamId == streamId)
85          .Unwind<Entities.EventStream, EventStreamSnapshotFlattened>(x => x.Snapshots)
86          .Match(x => version == null || x.Snapshots.Version <= version)
87          .SortByDescending(x => x.Snapshots.Version)
88          .Project(x => new FilteredSnapshot { Id = x.Id, Snapshots = x.Snapshots})
89          .FirstOrDefaultAsync();
90
91      return snapshot?.Snapshots?.ToDomain();
92  }
```

Лістинг 3.22 – Завантаження найближчого знімку у першому підході

Збереження знімків реалізоване за допомогою операції *Push,* яка дозволяє додати елемент у внутрішній масив документа (лістинг 3.23).

```
100     public async Task SaveSnapshot(Snapshot snapshot)
101     {
102         var collection = GetEventStreamsCollection();
103         var snapshotEntity = snapshot.ToEntity();
104
105         var update = Builders<Entities.EventStream>.Update
106             .Push(x => x.Snapshots, snapshotEntity);
107
108         await collection.UpdateOneAsync(x => x.StreamId == snapshot.StreamId, update);
109     }
110
```

Лістинг 3.23 – Збереження знімків у першому підході

У другому підході, метод *SaveSnapshot* виконує додавання нового документа у колекцію знімків (лістинг 3.24).

```
public async Task SaveSnapshot(Snapshot snapshot)
{
    var collection = GetSnapshotsCollection();
    var snapshotEntity = snapshot.ToEntity();

    await collection.InsertOneAsync(snapshotEntity);
}
```

Лістинг 3.24 – Збереження знімків у другому підході

Завантаження найближчого знімку відбувається за допомогою простої операції фільтрації (лістинг 3.25).

```
public async Task<Snapshot> GetNearestSnapshotAsync(Guid streamId, long? version = null)
{
    var collection = GetSnapshotsCollection();
    var filterBuilder = Builders<Entities.Snapshot>.Filter;
    FilterDefinition<Entities.Snapshot> filter = null;

    if (version != null)
    {
        filter = filterBuilder.And(
            filterBuilder.Lte(x => x.Version, version),
            filterBuilder.Eq(x => x.StreamId, streamId)
        );
    }
    else
    {
        filter = filterBuilder.Eq(x => x.StreamId, streamId);
    }

    var snapshot = await collection.Find(filter).SortByDescending(x => x.Version).FirstOrDefaultAsync();
    return snapshot?.ToDomain();
}
```

Лістинг 3.25 – Завантаження найближчого знімку у другому підході

Врешті решт, розглянемо реалізацію методу *AppendEventsAsync.* По аналогії з реляційною базою даних, він використовує оптимістичну модель паралелізму, яка, проте, працює дещо іншим чином. Реалізацію методу наведено у лістингу 3.26.

```
25  public async Task AppendEventsAsync(EventStream stream, IEnumerable<Event> events)
26  {
27      var collection = GetEventStreamsCollection();
28      var eventStream = MongoEntitiesMapper.ToEntity(events);
29
30      var update = Builders<Entities.EventStream>.Update
31          .Set(x => x.StreamId, eventStream.StreamId)
32          .Set(x => x.Version, eventStream.Version)
33          .Set(x => x.ConcurrencyToken, Guid.NewGuid())
34          .PushEach(x => x.Events, eventStream.Events);
35
36      var isUpsert = eventStream.Events.Count() - eventStream.Version == 0;
37
38      var result = await collection.UpdateOneAsync(
39          x => x.StreamId == eventStream.StreamId && x.ConcurrencyToken == eventStream.ConcurrencyToken,
40          update,
41          new UpdateOptions { IsUpsert = isUpsert }
42      );
43
44      if (!isUpsert && result.ModifiedCount == 0) throw new EventStoreConcurrencyException();
45  }
```

Лістинг 3.26 – Реалізація *AppendEventsAsync*

Для того, щоб зберети дані про об’єкт, потрібно додати події до масиву подій, якщо документ уже існує. Якщо ж ні, то окрім того потрібно створити новий документ. Зміни, що при цьому відбуваються описані в рядках 30-34. Далі використовується операція *UpdateOneAsync,* яка виконує зміну документа з вказаним *StreamId.* При цьому, якщо кількість подій, які ми хочемо зберегти відповідає версії об’єкта, то це означає що ми створюємо новий об’єкт і параметру *isUpsert* при цьому присвоюється значення *true,* що дозволяє методу *UpdateOneAsync* створити об’єкт, якщо його не існує. Розглянемо детальніше як реалізована оптимістична модель паралелізму. У рядку 33 при кожній зміні об’єкта ми записуємо унікальне значення у поле під назвою *ConcurrencyToken.* При цьому у рядку 39 під час виконання фільтрації ми перевіряємо не лише рівність *StreamId,* а і співпадіння реального та очікуваного *ConcurrencyToken.* Далі ми перевіряємо скільки документів було модифіковано. Якщо жодного і при цьому ми виконували саме операцію зміни, а не створення, то повертаємо помилку.

Іншими словами, ми намагаємось модифікувати документ з вказаним *StreamId* та тим самим *ConcurrencyToken,* який у нього був в момент читання. Якщо між читанням та записом інший користувач змінив цей документ, то це призведе до зміни *ConcurrencyToken* (рядок 33). Тоді фільтр у рядку 39 не поверне жодних результатів і значення *ModifiedCount* буде рівним нулю, що і свідчить про конфлікт паралелізму.

Фінальна структура документа, що зберігається у базі даних наведена у лістингу 3.27.

```
{
    "_id": "ObjectId(...)"
    "StreamId": "3423jk4324n23",
    "ConcurrencyToken": "sdfsdf434gg3445hj6j563t3t"
    "Version": 1,
    "Events": [
        {
            "Created": "16.02.2000 16:45:01",
            "Type": "DataRecorded",
            "Data": "Serialized Event Data",
            "Version": 1
        }
    ],
    "Snapshots": []
}
```

Лістинг 3.27 – Фінальна структура документа у MongoDB сховищі подій

Отже, нам вдалося побудувати увесь необхідний набір інструментів для того, щоб використовувати MongoDB у якості сховища подій.

3.5 Застосування моделі акторів у Event Sourcing системі

Для реалізації Event Sourcing системи із використанням моделі акторів було використано бібліотеку AntActor, що була розроблена незалежно від цієї роботи і тут не розглядається. Розглянемо лише її застосування у контексті нашої системи. Як було описано у попередньому розділі, модель акторів дозволяє перенести контроль паралелізму з рівня бази даних, на рівень застосунку. Цей підхід не є самостійною реалізацією сховища подій, а скоріше надбудовою над існуючими сховищами, яка, проте, дозволяє повністю зняти відповідальність за контроль паралелізму з бази даних. В розділі 2.4 було запропоновано ідею про те, що для кожного об'єкта у системі можна створити актора, через який відбуватиметься уся взаємодія з цим об'єктом за допомогою відправки повідомлень.

Розглянемо реалізацію актора для агрегату, що був описаний у лістингу 1.1. Для зручності продублюємо його у лістингу 3.28.

```
public class TemperatureMeasurement : Aggregate
{
    1 reference
    public DateTimeOffset Started { get; set; }
    1 reference
    public DateTimeOffset? LastRecorded { get; set; }
    2 references
    public List<decimal> Mesurements { get; set; } = default!;

    1 reference
    public TemperatureMeasurement(Guid id)
    {
        var @event = TemperatureMeasurementStarted.Create(id);

        Enqueue(@event);
        Apply(@event);
    }

    13 references
    public void Record(decimal temperature)
    {
        if (temperature < -273)
            throw new ArgumentOutOfRangeException(nameof(temperature));

        var @event = TemperatureRecorded.Create(Id, temperature);

        Enqueue(@event);
        Apply(@event);
    }

    Other Members
}
```

Лістинг 3.28 – Агрегат *TemperatureMeasurement*

Із визначення типу бачимо, що основними операціями, які змінюють цей агрегат є операції створення об’єкта *TemperatureMeasurement,* а також додавання результату вимірювання температури. Визначимо типи повідомлень, що будуть надсилатися актору для виконання цих операцій (лістинг 3.29).

```
public interface ITemperatureMeasurementAction
{
}

3 references
public class StartMeasurementAction: ReplyableMessageBase<bool>, ITemperatureMeasurementAction
{
    1 reference
    public StartMeasurementAction(ReplyChanel<bool> replyChannel): base(replyChannel)
    {
    }
}

3 references
public class RecordMeasurementAction : ReplyableMessageBase<bool>, ITemperatureMeasurementAction
{
    2 references
    public decimal Temperature { get; }

    1 reference
    public RecordMeasurementAction(decimal temperature, ReplyChanel<bool> replyChannel) : base(replyChannel)
    {
        Temperature = temperature;
    }
}

3 references
public class CreateAndRecord: ReplyableMessageBase<bool>, ITemperatureMeasurementAction
{
    2 references
    public IEnumerable<decimal> Measurements { get; }

    1 reference
    public CreateAndRecord(IEnumerable<decimal> measurements, ReplyChanel<bool> replyChanel): base(replyChanel)
    {
        Measurements = measurements;
    }
}
```

Лістинг 3.29 – Типи повідомлень актора *TemperatureMeasurementAnt*

Бачимо, що кожне повідомлення містить усю необхідну інформацію для виконання відповідної дії. Розглянемо тепер реалізацію самого класу актора у лістингу 3.30.

```
38  public class TemperatureMeasurementAnt : AbstractAnt<ITemperatureMeasurementAction>
39  {
40      private readonly Guid _id;
41      private TemperatureMeasurement _measurement;
42
43      private readonly IEventStore _eventStore;
44
        0 references
45      public TemperatureMeasurementAnt(IEventStore eventStore, string id)
46      {
47          _eventStore = eventStore;
48          _id = Guid.Parse(id);
49      }
50
        3 references
51      public override async Task OnActivateAsync()
52      {
53          _measurement = await _eventStore.Rehydrate<TemperatureMeasurement>(_id);
54      }
55
        2 references
56      public async Task StartMeasurement()
57          => await PostAndReply<bool>(rc => new StartMeasurementAction(rc));
        2 references
58      public async Task Record(decimal temperature)
59          => await PostAndReply<bool>(rc => new RecordMeasurementAction(temperature, rc));
        3 references
60      public async Task CreateAndRecord(IEnumerable<decimal> temperature)
61          => await PostAndReply<bool>(rc => new CreateAndRecord(temperature, rc));
62
        2 references
63      protected override async Task HandleMessage(ITemperatureMeasurementAction message)...
88  }
```

Лістинг 3.30 – Реалізація актора *TemperatureMeasurementAnt*

У рядках 40-41 поля, що містять ідентифікатор актора, який дозволяє отримати до нього доступ, а також об’єкт *TemperatureMeasurement,* доступ до якого контролює цей актор. Ідентифікатор актора та об’єкта співпадає, тому все що потрібно для того щоб отримати актора – це знати id об’єкта, доступ до якого він контролює. Відповідно для кожного об’єкта класу *TemperatureMeasurement* буде створюватись відповідний йому об’єкт класу *TemperatureMeasurementAnt.* Також актор містить посилання на імплементацію *IEventStore*, яку він використовує для збереження інформації про об’єкт в базі даних. Важливо звернути увагу на метод *OnActivatedAsync.* Цей метод викликається лише один раз при початковій ініціалізації актора. Він виконує операцію регідрації, отримуючи при цьому актуальний стан актора. Після початкової ініціалізації операція регідрації більше використовуватись не буде і уся взаємодія з об’єктом буде відбуватись у пам’яті. Також актор визначає методи *StartMeasurement,*

Record та *CreateAndRecord,* які використовуються для зручної відправки повідомлень, що були визначені у лістингу 3.29. При цьому повідомлення попадають у внутрішню чергу, звідки вони по одному опрацьовуються. Для обробки повідомлень використовується метод *HandleMessage,* реалізація якого наведена у лістингу 3.31.

```
63  protected override async Task HandleMessage(ITemperatureMeasurementAction message)
64  {
65      switch (message)
66      {
67          case StartMeasurementAction a:
68              if (_measurement != null) throw new Exception("Measurement already started");
69              _measurement = TemperatureMeasurement.Start(_id);
70              break;
71          case RecordMeasurementAction a:
72              if (_measurement == null) throw new Exception("Measurement not started");
73              _measurement.Record(a.Temperature);
74              break;
75          case CreateAndRecord a:
76              _measurement = TemperatureMeasurement.Start(_id);
77
78              foreach(var measurement in a.Measurements)
79              {
80                  _measurement.Record(measurement);
81              }
82              break;
83      }
84
85      await _eventStore.Store(_measurement);
86      (message as ReplyableMessageBase<bool>).ReplyChannel.Reply(true);
87  }
```

Лістинг 3.31 – Реалізація методу *HandleMessage*

Тут в залежності від типу повідомлення викликаються відповідні методи внутрішнього об’єкта *TemperatureMeasurement,* які виконують реальні зміни. Після цього, усі зміни зберігаються у сховище подій для того, щоб можна було відновити стан актора після його видалення з пам’яті з тих чи інших причин. З цієї реалізації стає очевидною перевага, що описувалась раніше – застосування моделі акторів дозволяє позбутися усіх операції регідрації об’єкта, окрім першої. При цьому усі операції запису усе ще залишаються. Зауважимо, що таку реалізацію методу *HandleMessage* у лістингу 3.31 не варто використовувати в реальних проектах, оскільки вона порушує принципи SOLID (а саме Single Responsibility Principle та Open Closed Principle) і

наведена лише для простоти сприйняття. В реальному коді варто розділити обробку кожного повідомлення, наприклад використавши патерн Стратегія. Для використання актора достатньо отримати його об’єкт, що у використаній бібліотеці досягається за допомогою класу *Anthill* (лістинг 3.32).

```
[HttpPost("{id}/temperatures")]
0 references
public async Task<IActionResult> Record(Guid id, [FromBody] decimal temperature)
{
    var measurementAnt = await _anthill.GetAnt<TemperatureMeasurementAnt>(id.ToString());
    await measurementAnt.Record(temperature);

    return Ok();
}
```

Лістинг 3.32 – Приклад використання створеного актора

Метод *GetAnt* класу *Anthill* повертає існуючого актора, якщо він уже міститься у пам’яті; якщо ж ні – створює нового та викликає метод *OnActivateAsync*.

З використанням описаного підходу можна створити акторів для взаємодії із усіма об’єктами системи. При цьому очевидними є додаткові затрати на етапі розробки. Порівняння цього підходу із двома іншими, буде виконано далі для оцінки доцільності його використання.

РОЗДІЛ 4. АНАЛІЗ ТА ПОРІВНЯННЯ

4.1 Принципи тестування

Для порівняння ефективності різних реалізацій сховища подій була написана серія тестів, які оцінюють час виконання тих чи інших дій. Тести запускалися локально з метою мінімізації впливу затримок мережі на кінцевий результат. Характеристики комп'ютера наступні:

а) Процесор: Intel(R) Core(TM) i7 2.60GHz, 6 ядер
б) Оперативна пам'ять: 16 GB DDR4
в) Диск: 1 TB SSD

Усі бази даних, що використовувались в тестах запускалися локально у вигляді docker контейнерів. Це дозволило забезпечити максимально рівноцінні умови для усіх реалізацій. Кожен тест повторювався деяку кількість разів і в результуючих графіках використовується середнє значення часу виконання усіх спроб. У тестах перевірялась ефективність виконання типових для сховища подій операцій.

Для сховища на основі MS SQL Server використовувалась база даних версії 2019. Для MongoDB використовувалась версія 6.0.1. Окрім того, в деяких тестах виконується порівняння власних реалізацій сховища подій з комерційним інструментом EventStoreDB, що працює на базі PostgreSQL. Версія EventStoreDB – 21.10. Реалізація, що використовує модель акторі працює на базі MS SQL Server.

4.2 Тест 1: Запис подій

У першому тесті порівнювалась швидкість запису різної кількості подій у сховище. Для цього у пам'яті створювався новий об'єкт, та виконувалась певна кількість змін цього об'єкта. Зрозуміло, що кожна зміна створювала нову подію у пам'яті. Далі, усі події зберігалися у сховище за один раз. Для кожного числа подій тест повторювався 5 разів. При цьому, перед початком кожної спроби з бази даних видалялися усі дані, що там знаходились для того, щоб мінімізувати вплив попередніх

спроб на швидкість виконання наступних. Далі, база заповнювалась тисячею об’єктів з різною кількістю подій для створення умов близьких до реальних. Час, що вимірювався це час виконання методу *Store* з інтерфейсу *IEventStore* в залежності від кількості подій. Результати тестування наведено на рисунку 4.1.

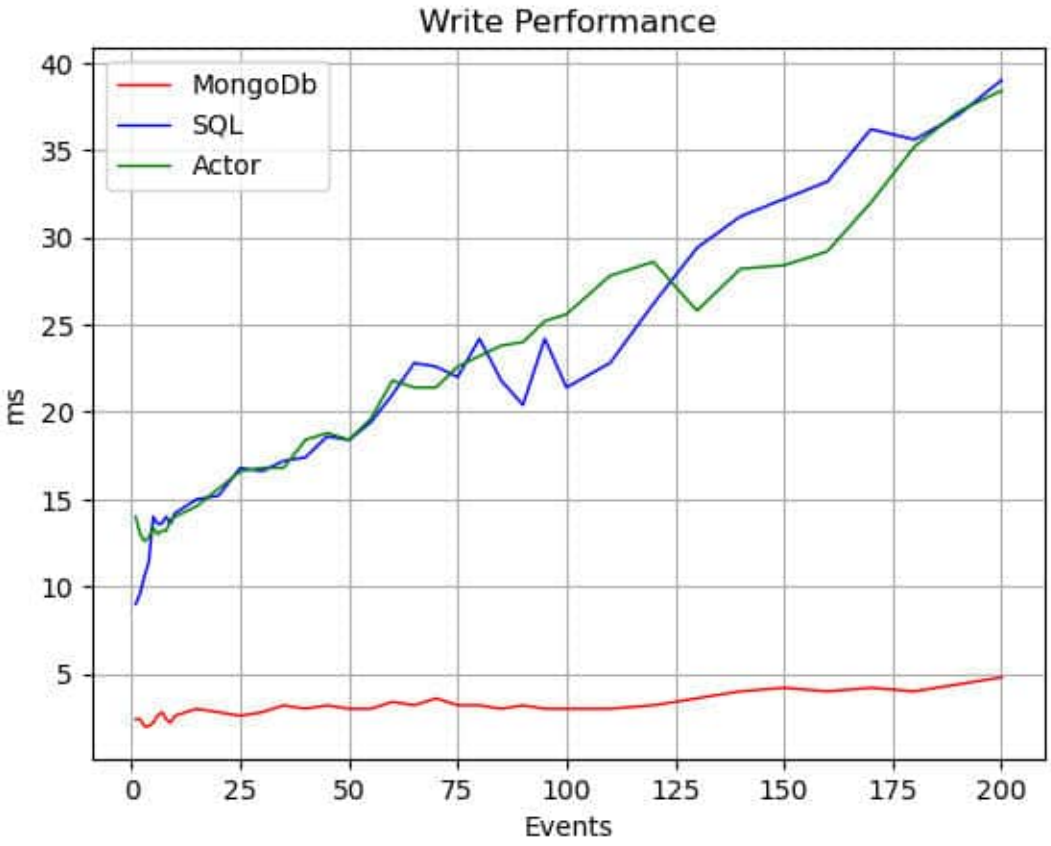

Рис. 4.1 – Швидкість збереження змін в залежності від кількості подій

Бачимо, що запис подій у сховище на базі MongoDB був значно швидшим ніж при використанні SQL бази даних, причому деградація швидкодії при збільшенні кількості подій була незначною, порівняно з реляційною базою даних. Оскільки реалізація, що використовує модель акторів була побудована на базі реляційного сховища, бачимо схожий час виконання у випадках застосування самостійної SQL бази даних, а також моделі акторів. Цей графік підтверджує твердження з розділів 2.4 та 3.5 про те, що застосування моделі акторів не впливає на швидкість запису даних у

базу. Також зауважимо, що при застосуванні реаляційної бази даних як у випадку самостійної реалізації, так і у випадку використання моделі акторів, швидкодія дещо покращувалась поблизу 100 подій в першому випадку та 125 подій в другому випадку. Скоріше за все це пов’язано з внутрішніми механізмами оптимізація MS SQL Server, наприклад механізмом статистик, проте ця девіація не впливає на загальну картину. Тут варто зазначити, що основна мета виконання тесту аж для 200 подій за одну операцію була у тому, щоб зрозуміти характер залежності. Така кількість змін за один раз є малоімовірною у реальних сценаріях роботи зі сховищем подій, тому при прийнятті рішень варто зважати лише на значення швидкодії в околі 1-5 подій за одну операцію.

4.3 Тест 2: Регідрація об’єкта з заданою кількістю подій

У другому тесті порівнювалась швидкість виконання операції регідрації об’єкта із заданою кількістю подій. Для цього перед початком тестування, уся інформація з бази даних видалялась, після чого відбувалось створення тисячі об’єктів з різною кількістю подій для створення умов близьких до реальних. Після цього створювався новий об’єкт із заданою кількістю подій та зберігався у сховище. Далі, оцінювався час виконання операції *Rehydrate* інтерфейсу *IEventStore,* без вказання необхідної версії об’єкта (тобто завантажувались усі події). Для кожної кількості подій операція регідрації виконувалась 5 разів з подальшим усередненням результатів. Графік результатів для кожної реалізації сховища наведено на рисунку 4.2.

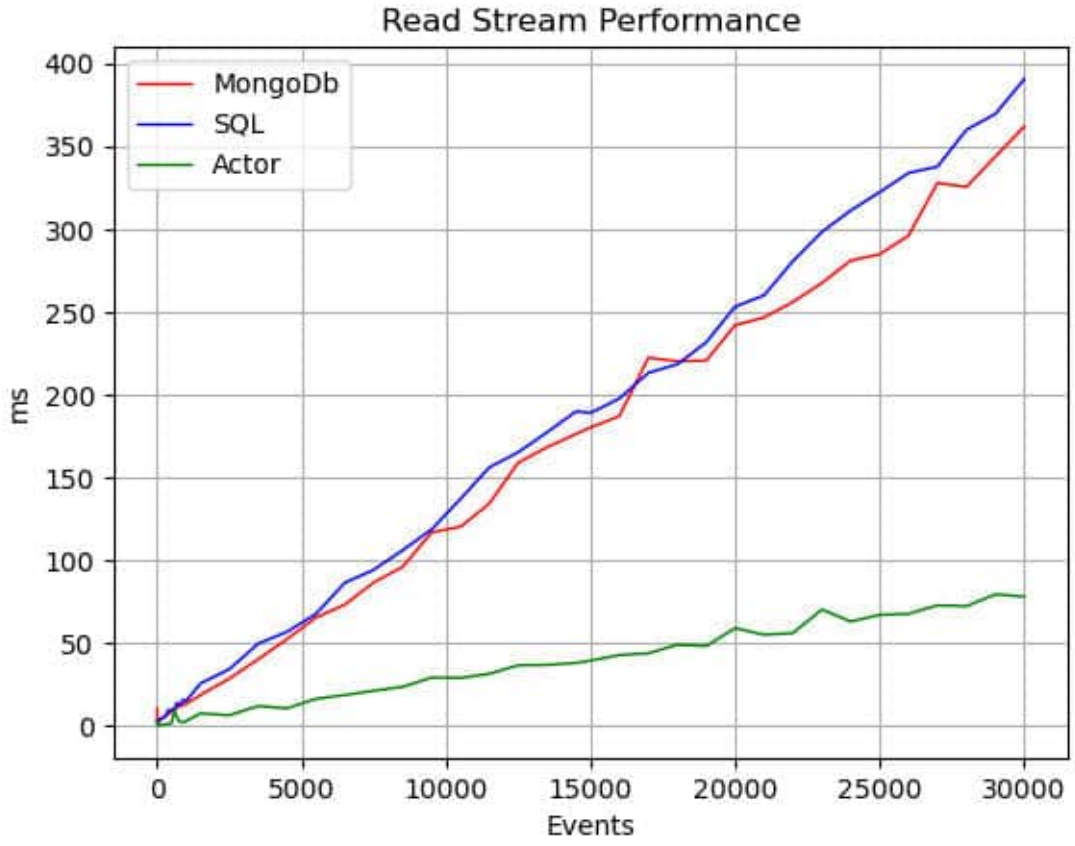

Рис. 4.2 – Час регідрації поточної версії об’єкта залежно від кількості подій

У цьому випадку результати дещо відрізняються від тесту 1. У всіх реалізаціях спостерігаємо лінійну залежність часу читання від кількості подій. При цьому реалізації, що використовують MongoDB та MS SQL показали практично однаковий результат, хоча MongoDB все ж була дещо швидшою. При цьому реалізація, що використовувала модель акторів показала значно кращий результат порівняно з іншими. Проаналізуємо її графік більш детально. Можна зауважити, що для кожної кількості подій, швидкодія моделі акторів є приблизно в 5 разів кращою, аніж при використанні SQL бази даних. Так відбувається тому, що при першому зчитуванні об’єкта з бази даних, швидкодія самостійної SQL бази даних та моделі акторів, що працює поверх SQL бази даних буде однакова. Проте, при кожному наступному читанні, реалізація на основі моделі акторів уже міститиме об’єкт у пам’яті, тому швидкість його читання буде знаходитися в околі декількох мілісекунд. Оскільки при

виконанні цього тесту операція читання одного і того самого об’єкта повторювалась 5 разів, після чого обчислювалось середнє значення усіх спроб, то графік моделі акторів можна отримати діленням часу виконання першого читання (який відповідає графіку самостійної SQL бази даних) на число спроб, тобто 5. Очевидно, що при зростанні кількості спроб, графік для моделі акторів прямуватиме до значення в околі 0.

4.4 Тест 3: Регідрація об’єкта певної версії із заданою кількістю подій

Дизайн тесту 3 є таким самим, як і тесту 2 з єдиною різницею в тому, що вимірюється час завантаження певної версії об’єкта, замість поточної версії. Тобто методу *Rehydrate* передається параметр з потрібною версією (передавалась версію на 1 менша за останню). Результати наведено на рисунку 4.3.

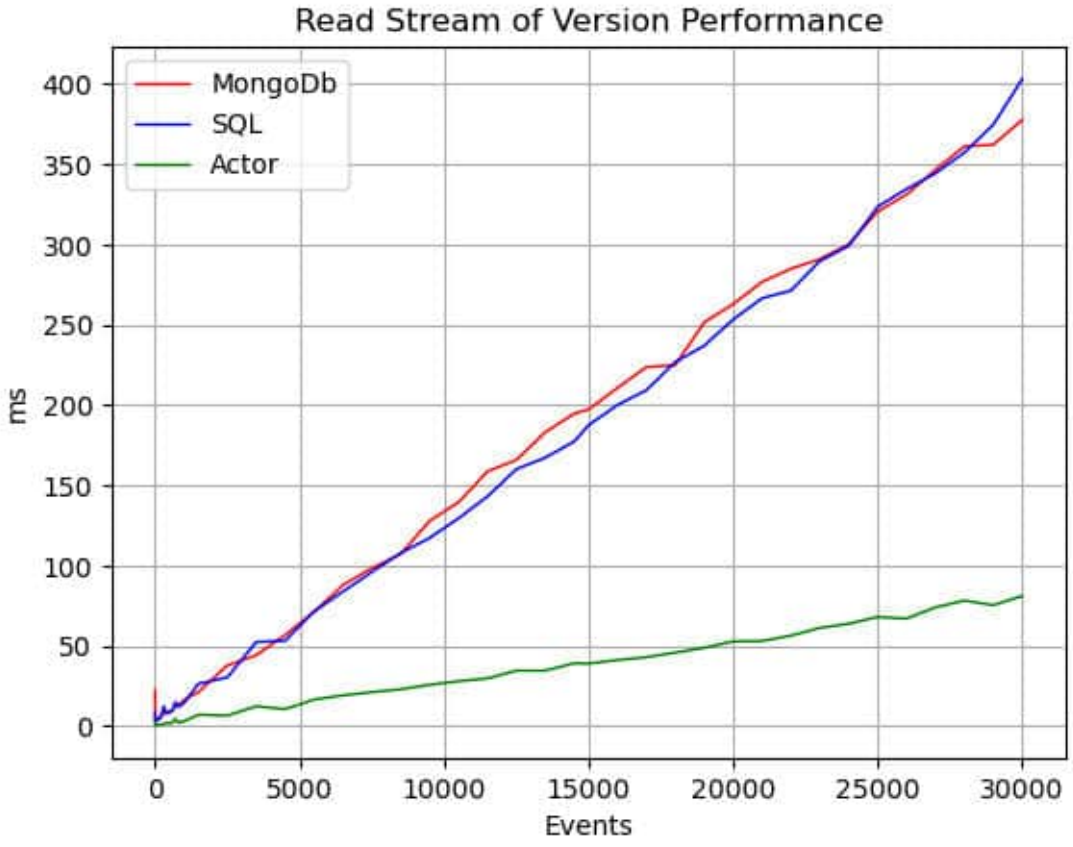

Рис. 4.3 – Регідрація заданої версії об’єкта залежно від кількості подій

Бачимо, що принципово картина не помінялася порівняно із завантаженням усіх подій об’єкта, проте, реалізація на базі MongoDB все ж показала деяке погіршення швидкодії. Це пов’язано з необхідністю використання aggregation pipeline для виконання цієї операції, як було описано у розділі 3.4.

4.5 Тест 4: Зміна об’єкта

Тести 4 полягає у послідовному виконанні операції зміни об’єкта певну кількість разів. На першому кроці об’єкт створюється у сховищі. У всіх наступних кроках, відбувається одинична зміна цього об’єкта. Операція зміни полягає у зчитуванні об’єкта з бази даних, виконанні зміни та збереженні модифікованого об’єкта назад у сховище. Для того самого об’єкта виконувалось 5000 послідовних операцій зміни, при цьому вимірювався час виконання кожної такої операції. Зрозуміло, що на кожному кроці зростатиме кількість подій, що описують стан об’єкта. Тест повторювався 5 разів, при цьому в межах однієї спроби виконувались усі 5000 операцій. Після цього обчислювалось середнє значення часу виконання зміни для кожного кроку. Перед початком тестування, база очищалась та заповнювалась тисячею об’єктів з різною кількістю подій. Описаний тест є максимально наближеним до реальних сценаріїв роботи зі сховищем подій, оскільки операція зміни об’єкта є найтиповішою. Результат виконання тесту наведено на рисунку 4.5.

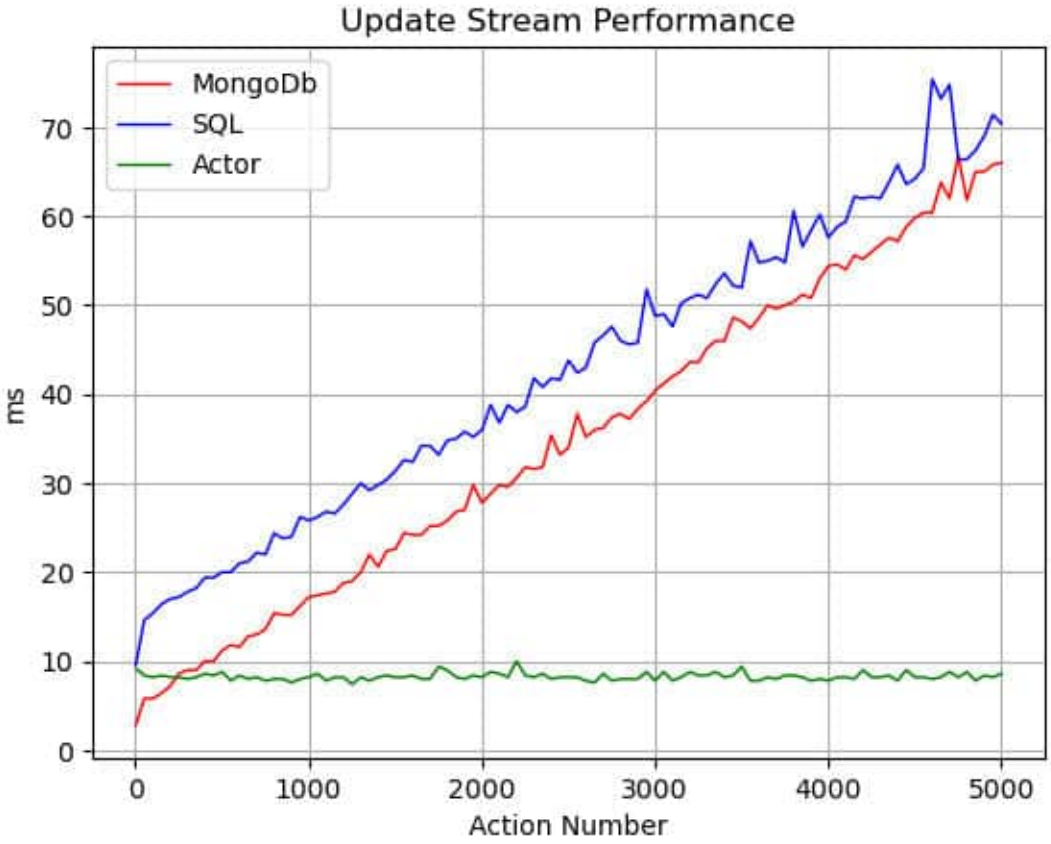

Рис. 4.5 – Час зміни об’єкта

Бачимо, що найбільш ефективною у цьому сценарії використання виявилась реалізація на основі моделі акторів, швидкодія якої не залежить від номеру зміни. Реалізації що використовують MongoDB та MS SQL Server показали однаковий характер залежності, проте MongoDB все ж виявилась швидшою. Проаналізуємо причини таких результатів. Як було сказано вище, операція зміни полягає у послідовному виконанні операції регідрації, модифікації, та запису даних. Оскільки на кожному кроці відбувався запис однієї події, то справедливим є припущення про те, що оримані значення складаються з часу запису однієї події та часу регідрації об’єкта із певною кількістю подій. Для MongoDB час запису одного об’єкта знаходиться в околі 2.5 мс (рис. 4.1). Для MS SQL Server це значення в околі 10 мс. Виходить, що отриманий графік є сумою значень графіка на рисунку 4.2 та значення у точці 1 графіка на рисунку 4.1. Що стосується моделі акторів, то як було описано в

розділі 4.3, вплив операції регідрації при ініціалізації актора мінімізується при зростанні кількості звернень до актора. Бачимо, що швидкодія на першому кроці співпадає зі швидкодією з SQL базою даних. Далі, вона залишається на рівні 10мс, що відповідає швидкості запису однієї події в SQL базу даних, оскільки операції регідрації стають не потрібними.

У межах цього тесту також було виконане порівняння реалізацій з цієї роботи з комерційним інструментом EventStoreDB (рис. 4.6). Аналіз результатів не проводився, оскільки EventStoreDB є закритою системою і зрозуміти їх причини є неможливим.

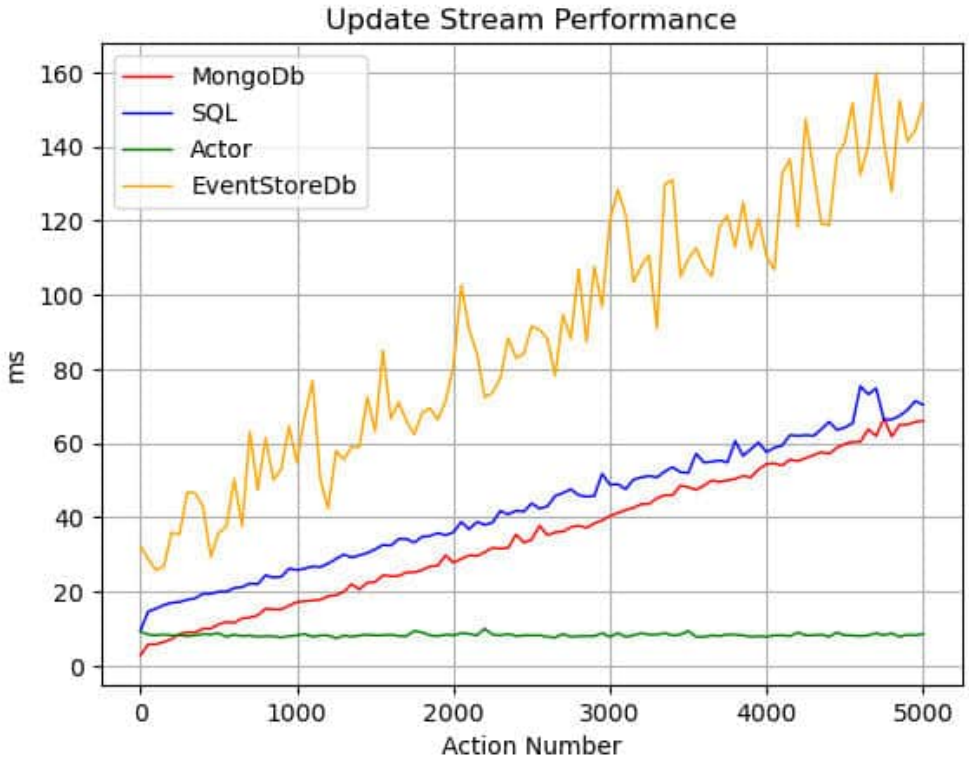

Рис. 4.6 – Порівняння власних реалізацій сховища подій з EventStoreDB

У розділі 1.1 був описаний підхід для оптимізації читання об’єкта під назвою знімки. Перевіримо, як застосування цього механізму вплине на ефективність виконання сценарію тесту 4. Для цього будемо створювати знімки кожні 200 подій. Число 200 обране тому, що згідно з графіком на рисунку 4.5 час зчитування 200 подій

становить близько 15 мілісекунд, що в абсолютних величинах не сильно відрізняється від результатів, отриманих з використанням моделі акторів. Пам’ятаємо, що для MongoDB було реалізовано два способи збереження знімків, побудуємо графіки для кожного з них. Графіки часу виконання змін об’єкта з використанням знімків наведено на рисунках 4.7, 4.8.

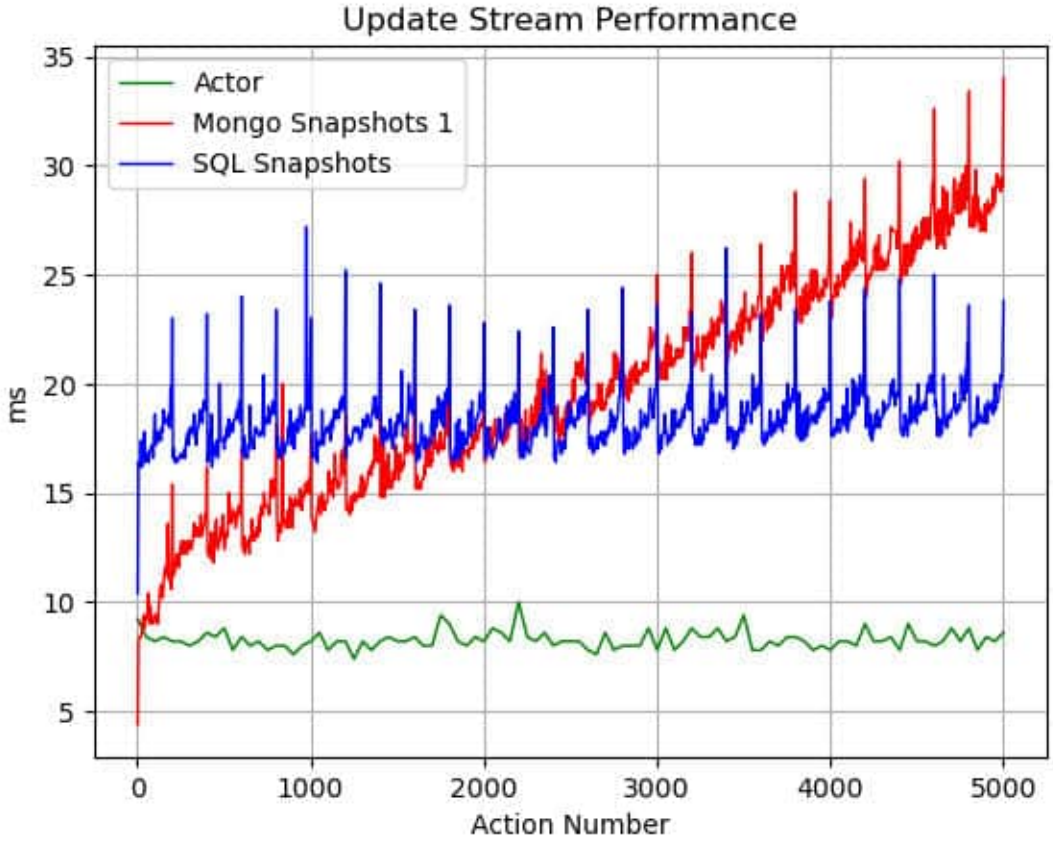

Рис. 4.7 – Час зміни об’єкта з викорстанням знімків (перший підхід в MongoDB)

На графіку видно, що кожні 200 подій з’являється зубець, який означає зниження швидкодії. Причина його появи це додаткові затрати на створення та збереження знімку. При цьому, використання механізму знімків в SQL базі даних дозволяє добитися константної залежності від кількості змін, хоч і загальна швидкодія є дещо нижчою ніж при використані моделі акторів. Щодо нереляційної бази даних, бачимо, що механізм знімків також дозволив добитися значного покращення

швидкодії. Скоріше за все ще пов’язано з затратами на пошук потрібного знімку у внутрішньому масиві документа, як було описано у розділі 3.4.

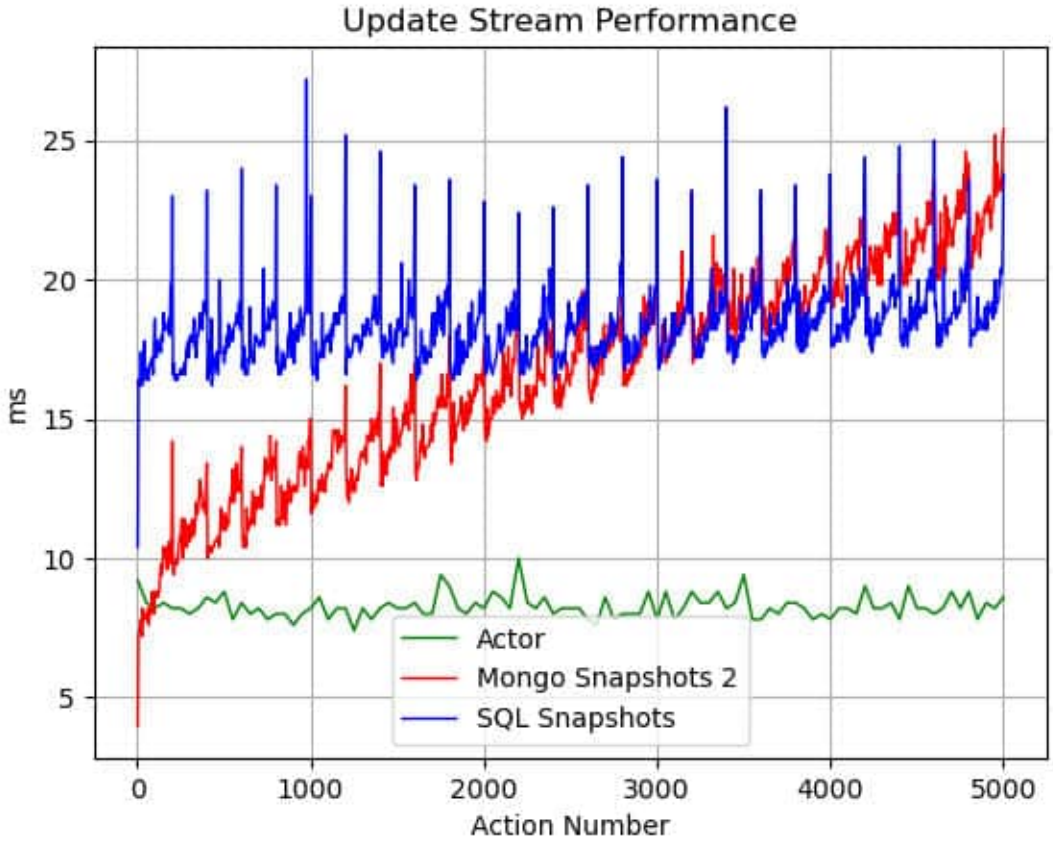

Рис. 4.8 - Час зміни об’єкта з використанням знімків (другий підхід в MongoDB)

При використанні другого підходу до створення знімків у MongoDB, швидкодія стала кращою та швидкість зростання часу виконання операцій знизилась. Проте, залежність усе ще залишається лінійною. Причини такої поведінки знаходяться у внутрішніх механізмах MongoDB або MongoDB Driver і потребують додаткових досліджень.

4.6 Тест 5: Одночасна зміна об’єкта кількома користувачами

Як було описано у попередніх розділах, основною складністю при реалізації сховища подій є створення механізму вирішення конфліктів паралелізму. В контексті цієї роботи розглядався оптимістичний підхід. Також, в розділі 2.3 згадується про те, що оптимістичний підхід базується на допущенні, що конфлікти у системі ставатимуться рідко, у інших випадках він може виявитись неефективним. Перевіримо це твердження на практиці. Тест 5 полягає у вимірюванні часу виконання 2000 операцій зміни одного і того самого об’єкта. При цьому ці операції розділятимуться між декількома користувачами. На першому кроці вимірюється час виконання 2000 операцій одним користувачем. На другому, двома користувачами. У цьому випадку кожен з користувачів буде намагатися виконати 1000 операцій зміни того самого об’єкта. На 4 кроці – кожен користувач виконуватиме по 500 операцій і тд. При виникненні конфлікту паралелізму, користувач повторюватиме операцію, допоки не вдасться виконати її успішно. Знову ж таки зрозуміло, що висока частота змін того самого об’єкта є малоймовірною у реальному житті, проте тест дозволяє отримати залежність часу змін, а також кількості помилок від рівня паралелізму, на основі якої можна прийняти рішення про використання того чи іншого підходу. На рисунку 4.9 зображено графік залежності часу виконання 2000 операцій змін об’єкта в залежності від рівня паралелізму.

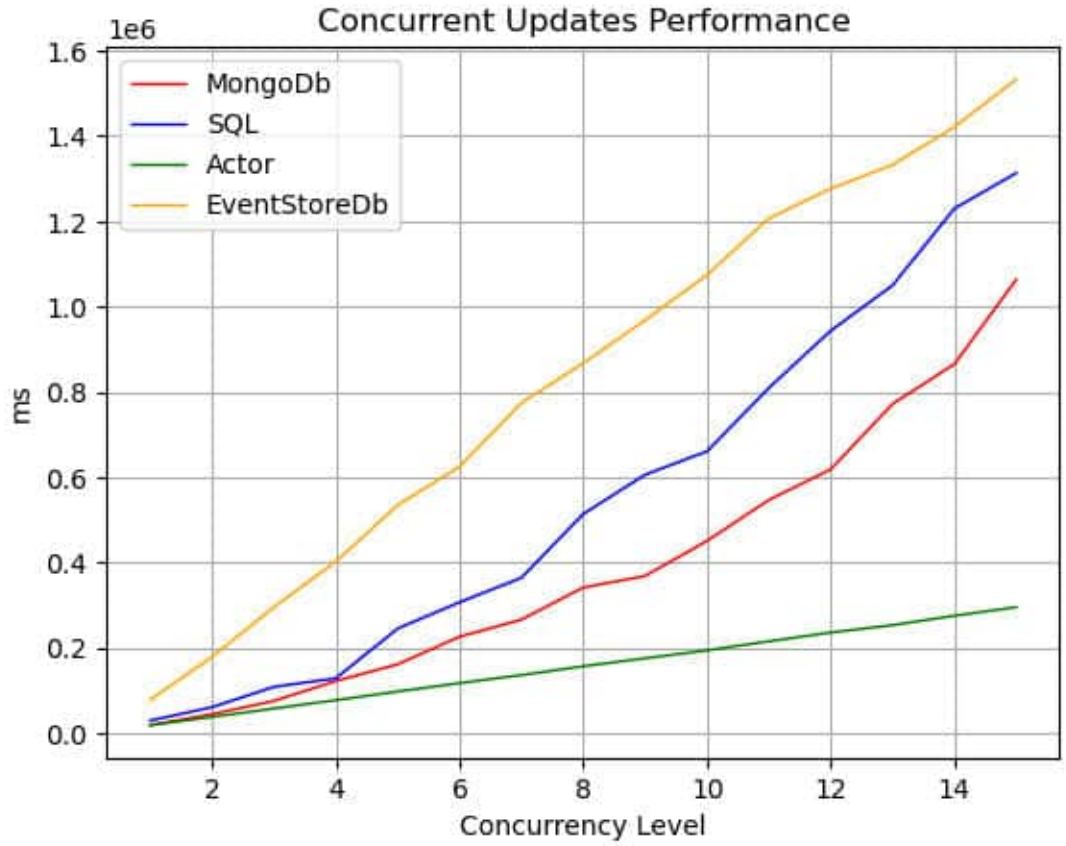

Рис. 4.9 – Залежність часу виконання змін від рівня паралелізму

Бачимо, що при використанні оптимістичної моделі, час виконання операцій в MS SQL та MongoDB зростає нелінійно. Event Store DB також використовує оптимістичну модель паралелізму, хоча в його випадку залежність часу виконання від рівня паралелізму все ж залишається лінійною. І лише модель акторів показує хорошу швидкодію і чітку лінійну залежність. Лінійна залежність у випадку з моделлю акторів пов’язана з тим, що повідомлення попадають у чергу, тому користувачі повинні чекати не лише на завершення власних операцій, а і на інші, які знаходяться у черзі. Також можемо побудувати графік залежності кількості конфліктів паралелізму, що виникли від рівня паралелізму. Цю залежність наведено на рисунку 4.10.

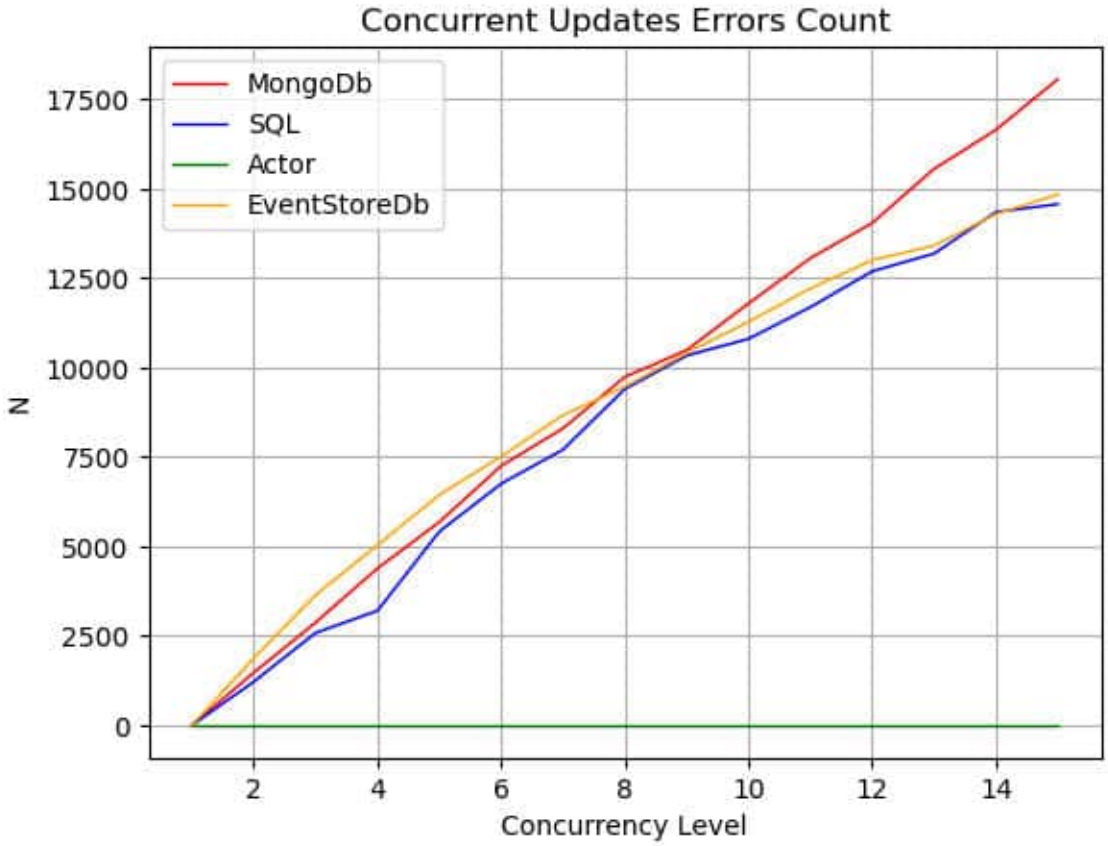

Рис. 4.10 – Залежність кількості конфліктів паралелізму від рівня паралелізму

Бачимо, що залежність кількості помилок в реалізації на основі MongoDB є практично лінійною, тому швидкість зростання функції на рисунку 4.6 також є найбільшою. Реалізації на основі MS SQL Server та Event Store DB виглядають практично однаково, що може свідчити про використання схожих підходів до реалізації операції запису за умови. Проте, графік Event Store DB на рисунку 4.6 зростає повільніше, що свідчить про використання додаткових засобів оптимізації у цьому інструменті. У реалізації, що використовує модель акторів, як і очікувалось, конфліктів паралелізму не виникало.

ВИСНОВОК

У результаті виконання роботи вдалося реалізувати сховище подій на основі SQL та NoSQL баз даних, а також перевірити можливість застосування моделі акторів при їх реалізації. Найбільш ефективною у реалістичних сценаріях використання виявилась система, що використовує модель акторів, оскільки дозволяє досягти сталого часу зміни об’єктів при будь-якій кількості подій у потоці. При цьому, серед реалізацій що порівнювались, лише модель акторів показала хорошу ефективність при високому рівні паралелізму. Швидкодія системи, що використовує оптимістичну моделі вирішення конфліктів стрімко деградує із збільшенням кількості одночасних змін об’єкта. У випадках, коли одночасний доступ до об’єктів відбувається рідко, як альтернативу моделі акторів, можна розглядати реалізацію на основі реляційної бази даних із використанням механізму знімків. Такий підхід потребує менше зусиль на етапі розробки і дозволяє досягти сталого часу зміни об’єктів. При цьому, цей час є все ж більшим, ніж при застосуванні моделі акторів. У реалізації, що використовує нереляційну базу даних MongoDB, не вдалося досягти сталого часу зміни об’єктів, тому її застосування є недоцільним у загальному випадку. Також не варто забувати про обмеження розміру документа у такій системі, яке на даний момент становить 16 MB. Таке сховище можна розглядати лише у випадках, коли розмір об’єктів, а також час протягом якого з ними відбувається будь-яка взаємодія є чітко передбачуваними. У тих випадках, коли кількість подій об’єкта є меншою 4000 ця реалізація показала кращу ефективність, аніж реалізація на основі SQL бази даних. При більшій кількості подій, швидкодія лінійно погіршується.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

[1] Event Sourcing. [Електронний ресурс]: Martin Fowler – 2005 – Режим доступу: https://martinfowler.com/eaaDev/EventSourcing.html

[2] Betts D. Exploring CQRS and Event Sourcing – Microsoft, 2012 – С. 235-246

[3] Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software – Addison-Wesley Professional; 1st edition (August 20, 2003).

[4] Snapshots in Event Sourcing. [Електронний ресурс]: Oskar Dudycz – 2021 – Режим доступу: https://www.eventstore.com/blog/snapshots-in-event-sourcing

[5] Concurrent commands in event sourcing. [Електронний ресурс]: Michiel Rook – 2016 – Режим доступу: https://www.michielrook.nl/2016/09/concurrent-commands-event-sourcing/

[6] Event Sourcing and Concurrent Updates. [Електронний ресурс]: Teiva Harsanyi – 2017 – Режим доступу: https://teivah.medium.com/event-sourcing-and-concurrent-updates-32354ec26a4c

[7] Handling Concurrency Conflicts in a CQRS and Event Sourced system. [Електронний ресурс]: Daniel Whittaker – Режим доступу: https://danielwhittaker.me/2014/09/29/handling-concurrency-issues-cqrs-event-sourced-system/

[8] Actor model. [Електронний ресурс] – Режим доступу: https://en.wikipedia.org/wiki/Actor_model

[9] Actor Model and Event Sourcing. [Електронний ресурс]: Andrzej Ludwikowski – 2021 – Режим доступу: https://blog.softwaremill.com/actor-model-and-event-sourcing-aa00993d2f1e

[10] Akka, DDD, CQRS, Event Sourcing and Me. [Електронний ресурс]: Andrew Easter – 2013 – Режим доступу: https://medium.com/@dreweaster/akka-ddd-cqrs-event-sourcing-and-me-dad400ab62d1

[11] Alexev Zimarev - DDD, Event Sourcing and Actors. [Електронний ресурс]: Alexey Zimarev – Режим доступу: https://www.youtube.com/watch?v=roCffcadWAU

[12] Building an Event Storage. [Електронний ресурс] – Режим доступу: https://cqrs.wordpress.com/documents/building-event-storage/

[13] Korotkevitch D. Pro SQL Server Internals – Appress, 2014 – C. 375-386

[14] Rothsberg J. Evaluation of using NoSQL databases in an event sourcing system: Final thesis: 24.11.2015 / Linkoping University. Department of Computer and Information Science – C. 26

[15] MongoDB Documentation. [Електронний ресурс] – Режим доступу: https://www.mongodb.com/docs/manual/faq/fundamentals/

[16] Доскач Д. Дослідження методів побудови Event Sourcing систем // Інновації та перспективи світової науки. Тези доповідей п'ятнадцятої Міжнародної наукової конференції (12-14 жовтня 2022р., Ванкувер, Канада). – С. 21-27. URL: https://sci-conf.com.ua/xv-mizhnarodna-naukovo-praktichna-konferentsiya-innovations-and-prospects-of-world-science-12-14-10-2022-vankuver-kanada-arhiv/