Міністерство освіти і науки України
Національний університет «Києво-Могилянська академія»
Факультет інформатики
Кафедра інформатики

Магістерська робота

Освітній ступінь – магістр

на тему: **«Гібридні алгоритми синхронізації даних веб-застосунків з колаборативним редагуванням»**

Виконав: студент 2-року навчання

Спеціальності
121 Інженерія програмного забезпечення

Шевчук Кіріл Юрійович

Керівник Гороховський С. С.

Доцент, кандидат наук

Рецензент _____

Магістерська робота захищена

з оцінкою _____

Секретар ЕК _____

«___» _____2026

Київ – 2026

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

БД	-	База даних
OT	-	Operational Transformation
HLC	-	Hybrid Logical Clock
SQL	-	Structured Query Language
CRDT	-	Conflict-Free Replicated Data Types
CRUD	-	Create, Read, Update, Delete, базові операції над даними
OPFS	-	Origin Private File System
JSON	-	JavaScript Object Notation
UI	-	Користувацький інтерфейс
WASM	-	WebAssembly

ВСТУП

Сучасне програмне забезпечення для мережевих платформ пройшло великий шлях від простих статичних сторінок до надскладних систем, що за функціоналом та продуктивністю не поступаються настільним програмним рішенням. Вебзастосунки сьогодення характеризуються величезною кількістю взаємозалежних функцій, динамічним інтерфейсом, що миттєво змінюється від дій користувачів, та необхідністю обробки великих масивів даних у реальному часі. Оскільки веб програми стали основним інструментом для професійної діяльності у різних індустріях, вимоги користувачів до їх надійності та швидкодії зросли.

Однією з головних проблем сучасного вебу залишається залежність від стабільності мережевого з’єднання. Традиційна архітектура «тонкого клієнта» часто призводить до затримок та неможливості роботи в офлайн-режимі. У відповідь на це виникла концепція Local-first software, яка передбачає, що дані зберігаються локально, а мережа використовується лише для синхронізації. Яскравим прикладом успішної реалізації такої моделі є система Linear [1], яка демонструє безпрецедентну швидкість відгуку завдяки локальному кешуванню та фоновій синхронізації. Проте реалізація таких систем ускладнюється необхідністю забезпечення цілісності даних, адже тепер дані розподіляються між клієнтами, кожен з яких володіє власною копією.

Актуальність роботи зумовлена необхідністю пошуку ефективних механізмів вирішення конфліктів у розподілених системах. Існуючі підходи — Operational Transformation (OT) та Conflict-free Replicated Data Types (CRDT) — самі по собі мають значну складність в реалізації і інтеграції в існуючі програми, адже вимагають написання складної логіки і змін до підходів моделювання даних. Сучасні готові бібліотеки, по типу Replicache,

ElectricSQL, LiveStore, Tinybase, RxDB та інші зазвичай не є повним рішенням проблеми, а лише частковим, і накладають додаткові обмеження на інфраструктуру, вимагають використовувати окрему базу даних чи сторонній сервіс.

Обґрунтування необхідності виконання роботи випливає з потреби розробників у доступному інструментарії для створення offline-first застосунків. Існуючий поріг входження у розробку колаборативного ПЗ є надто високим через необхідність глибоких знань у теорії розподілених систем. Розробка бібліотеки, яка б автоматизувала трансформацію стандартних SQL-операцій у синхронізовані події, дозволить значно спростити процес створення сучасних вебзастосунків.

Призначення магістерської роботи полягає у дослідженні та розробці гібридного методу синхронізації даних, який інтегрує CRDT-події безпосередньо в реляційну модель SQLite. Результатом роботи є програмний комплекс sqlite-sync — програмна бібліотека, що забезпечує повний цикл життєдіяльності даних: від локального збереження і реактивного відображення в інтерфейсі, до їх синхронізації між клієнтами.

Постановка задачі:

1. Проаналізувати сучасні алгоритми і готові рішення для синхронізації даних
2. Розробити комплексну програмну бібліотеку з використанням гібридного підходу, зокрема:
   - Синхронізація реляційних таблиць і автоматичне перетворення SQL запитів на CRDT операції
   - Event Sourcing для централізованого впорядкування подій і подальшої реплікації
   - Можливість легкої реалізації Operational transformation (OT) для розв’язання більш складних конфліктів
3. Провести експериментальну перевірку на прикладних застосунках

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ ПІДХОДІВ ДО СИНХРОНІЗАЦІЇ ДАНИХ

1.1 Особливості сучасних веб застосунків і проблема синхронізації даних

Однією з ключових особливостей сучасних вебзастосунків є високий рівень інтерактивності і динамічності. Користувач очікує, що будь-яка його дія матиме майже миттєвий відгук. У традиційній серверній моделі, де всі операції запису та читання виконуються через сервер, швидкодія системи безпосередньо залежить від часу відповіді сервера та стабільності каналу передачі даних. Навіть незначні затримки можуть суттєво погіршувати користувацький досвід, особливо у випадках, коли робота з інтерфейсом передбачає часті зміни даних.

Ще однією важливою характеристикою вебзастосунків є необхідність підтримки складного локального стану. У сучасному інтерфейсі одночасно можуть співіснувати серверні дані, тимчасові зміни користувача, кешовані результати запитів, локальні черги операцій та результати фонової синхронізації. Усе це формує надскладну модель даних, яку дуже важко тримати узгодженою у випадках втрати мережі, оновлення сторінки, відкриття декількох вкладок або одночасної роботи кількох користувачів із тими самими сутностями.

У відповідь на ці обмеження сформувалася концепція offline-first або local-first програмного забезпечення, запропонована групою дослідників з Ink & Switch [2]. Її основна ідея – дані мають зберігатися і змінюватися локально, застосунок може працювати без мережі, адже вона використовується лише для фонової синхронізації. Це дозволяє забезпечити високу швидкодію інтерфейсу і значно спростити логіку застосунку, оскільки клієнт володіє даними і може зчитувати і записувати миттєво. Водночас саме тут виникає

головна складність: якщо одна й та сама інформація існує у кількох копіях, необхідно гарантувати її коректне узгодження після обміну змінами.

Проблема синхронізації даних у розподілених вебзастосунках полягає саме в забезпеченні цілісності, узгодженості та передбачуваної поведінки системи за умов, коли пов’язані зміни можуть виникати незалежно у різних вузлах. Кожен клієнт може мати власний локальний стан, виконувати операції у довільний момент часу, тимчасово працювати офлайн, а після відновлення зв’язку надсилати накопичені зміни на сервер. Якщо два або більше клієнтів змінюють одну й ту саму сутність паралельно, виникає конфлікт, який потрібно або автоматично розв’язати, або коректно представити на рівні прикладної логіки. Простий принцип «остання зміна перемагає» не завжди є прийнятним, оскільки він може призводити до втрати даних і руйнування семантики предметної області. У веб застосунках ця проблема ще більше ускладнюється тим, що користувачі можуть легко відкрити застосунок в декількох вкладках, які також треба якось між собою синхронізувати.

Ще одним важливим аспектом є необхідність підтримки правильного порядку змін. Використання фізичного часу не гарантує коректного впорядкування, оскільки годинники різних пристроїв можуть відрізнятися, і мережеві затримки можуть змінювати порядок доставки повідомлень. Це зумовлює потребу у спеціальних механізмах логічного або гібридного впорядкування подій, які є основою багатьох сучасних алгоритмів синхронізації.

1.2 Аналіз підходів до синхронізації: Operational Transformation, CRDT, Event Sourcing

Існує декілька базових підходів до організації синхронізації даних, серед яких найбільш відомими є Operational Transformation (OT), Conflict-free

Replicated Data Types (CRDT) та Event Sourcing. Хоча ці підходи виникли в різному контексті і розв’язують дещо різні задачі, на практиці вони часто розглядаються разом, оскільки всі спрямовані на забезпечення узгодженості стану системи за умов паралельних змін, мережевих затримок та часткової автономності вузлів.

Operational Transformation є одним із найперших і найвідоміших підходів до підтримки колаборативного редагування. Він був вперше запропонований у 1989 С. Еллісом і С. Гіббсом [3], і набув більшої популярності коли його запровадили інженери гугл у 2009 [4]. Його основна ідея полягає в тому, що одночасні операції користувачів не просто застосовуються до спільного стану, а трансформуються централізованим сервером таким чином, щоб зберігати семантично коректний результат. Класичним прикладом є спільне редагування тексту, коли вставка або видалення символів одним користувачем змінює позиції, до яких прив’язані операції інших користувачів. У такому випадку система повинна перерахувати координати операцій так, щоб після їх послідовного застосування всі учасники отримали однаковий і логічно узгоджений документ.

Перевагою ОТ є те, що цей підхід добре зберігає наміри користувача. Однак реалізація ОТ є дуже складною і схильною до помилок, оскільки вимагає визначення правил трансформації для кожної пари або класу операцій. Для складних структур даних кількість таких правил швидко зростає, а доведення коректності алгоритму стає нетривіальним. Крім того, ОТ зазвичай потребує централізованого контролю версій, підтвердження порядку операцій або принаймні складної координації між вузлами. Це унеможливлює застосування підходу в offline-first системах, де клієнти повинні тривалий час працювати автономно і синхронізувати зміни лише після відновлення зв’язку.

Альтернативою ОТ стали Conflict-free Replicated Data Types [5,6]. CRDT являють собою спеціальні структури даних, для яких операції зміни

стану визначені таким чином, що злиття різних версій завжди відбувається детерміновано і без конфліктів. Головна властивість CRDT полягає в тому, що незалежно від порядку доставки повідомлень, повторної доставки або часткової затримки синхронізації всі репліки зрештою збігаються до одного консистентного стану. Це робить підхід особливо привабливим для розподілених систем з високою автономністю клієнтів. Додатковою перевагою є те, що багато CRDT не потребують складної централізованої координації, а отже краще відповідають розподіленій природі вебзастосунків. Детально CRDT описані у статті Джейка Лазароф [7].

Водночас CRDT також мають обмеження. По-перше, вони зазвичай вимагають спеціального моделювання даних, оскільки не будь-яка бізнес-сутність природно зводиться до стандартної CRDT-структури. По-друге, для частини сценаріїв автоматичне безконфліктне злиття може не збігатися з очікуваною прикладною семантикою. Наприклад, для складних доменних обмежень, агрегатів або транзакційних залежностей між кількома таблицями потрібна додаткова логіка, яка виходить за межі базових властивостей CRDT.

Також відносно нещодавно почав набувати популярності Event Sourcing як рішення для local-first застосунків. Загалом цей термін виник ще у двохтисячних роках [8], і використовувався переважно щодо збереження серверних даних. У моделі Event Sourcing стан системи розглядається як результат послідовного застосування подій, що відображають усі зміни. Замість збереження лише поточного стану база фіксує історію операцій, а актуальне представлення може бути відтворене шляхом повторного програвання журналу подій.Такий підхід добре поєднується з реплікацією, оскільки саме події стають основною одиницею обміну між клієнтом і сервером.

В контексті local-first цей термін набув популярності з випуском LiveStore у 2024 [9]. Сильна сторона Event Sourcing полягає у наявності повного журналу змін, що полегшує аудит, відновлення, дедуплікацію,

повторну синхронізацію і побудову похідних представлень даних. У контексті offline-first систем це особливо корисно, оскільки клієнт може накопичувати локальні події, а після відновлення зв’язку передавати їх на сервер для впорядкування і подальшої реплікації іншим вузлам.

Однак, сам по собі, Event Sourcing не визначає як саме розв’язувати конфлікти і потребує обмеженого моделювання даних.

Всі розглянуті підходи мають свої особливості і потребують повного розуміння для правильного використання, однак жоден з цих підходів не є повним рішенням синхронізації даних в веб застосунках, і для побудови практичної системи доцільним є їх комбінування. Найбільш перспективною виглядає модель, у якій Event Sourcing використовується як механізм фіксації та передавання змін, CRDT забезпечує автоматичне безконфліктне злиття більшості операцій, а OT або подібна до нього прикладна трансформаційна логіка може застосовуватися для окремих складних випадків, де потрібно зберегти семантичний намір користувача.

1.3 Огляд існуючих рішень: ShareDB, Yjs, LiveStore, ElectricSQL

Перед тим як розробляти власне рішення, доцільно розглянути кілька відомих підходів до розв’язання проблеми синхронізації даних. Вони репрезентують різні архітектурні моделі: OT-орієнтовний SharedDB, популярну CRDT бібліотеку - Yjs, event sourcing орієнтовну бібліотеку LiveStore і та ElectricSQL (Postgres Sync), яка вирішує проблему синхронізації реляційних баз даних [9; 10; 11; 12].

Почнемо з SharedDB – це комплексна бібліотека яка вирішує проблему конфліктів даних через operational transformation (OT). Вона надає Node.js сервер для координації та фіксації змін, а також JavaScript-клієнт для

браузера або Node.js, однак сама по собі не вирішує конфлікти і не реалізує правила трансформації самостійно, а делегує це спеціальним OT типам, які можуть бути кастомізовані залежно від домену задачі. Основними недоліками є документно-орієнтована модель даних замість реляційної, залежність від централізованої координації і необхідність окремо проектувати та доводити коректність OT-логіки для нетривіальних сценаріїв, що взагалі не є тривіальної задачею. Тому ShareDB добре підходить для синхронізації JSON-документів, але значно гірше узгоджується з ідеєю прозорої синхронізації звичайних SQL-операцій, і як результат не набула популярності, хоча і вийшла ще на початку 2010 року.

Другий готовий підхід – Yjs, це одна з найпопулярніших реалізацій різних CRDT структур [11], написана на мові програмування JavaScript. Вона представляє з себе набір структур даних і типів, зокрема Y.Map, Y.Array, Y.Text та Y.Xml, що можуть безконфліктно синхронізуватися незалежно від того, який канал для передачі даних використовується (HTTP, WebSocket, WebRTC, тощо). Сильною стороною Yjs є відсутність потреби в єдиному центральному вузлі для розв’язання конфліктів, хороша підтримка офлайн-роботи і велика екосистема розширень з готовими інтеграціями текстових редакторів і інших програм. Саме тому бібліотека Yjs особливо ефективна для задач одночасного редагування тексту, rich-text контенту або інших деревоподібних структур. Водночас Yjs є саме CRDT-ядром, а не повним прикладним рішенням: мережевий транспорт, збереження, авторизація, доменна модель та логіка зчитування даних мають добудовуватися окремо. Через це інтеграція Yjs у реляційну модель даних є нетривіальною, оскільки розробник працює не з SQL-таблицями, а зі спеціалізованими CRDT-типами. Yjs добре вирішує проблему безконфліктного злиття у конкретних випадках, але не є самодостатнім рішенням для повноцінної синхронізації вебзастосунків, і також вимагає розуміння для правильної інтеграції в існуючі застосунки.

LiveStore вже позиціонується як повний local-first data layer на основі реактивної SQLite та event sourcing [9]. Ця бібліотека вийшла відносно нещодавно (перша бета версія була опублікована лише у 2024), однак швидко набула популярності. Основна ідея – стан застосунку матеріалізується у локальну SQLite-базу, а всі зміни фіксуються як впорядкований журнал подій, який реплікується між клієнтами через центральний сервер. Такий підхід дозволяє зробити зчитування і запис даних локальними, що робить сервіси стійкими до втрати мережі та полегшує створення справжнього реактивного користувацького інтерфейсу. До переваг LiveStore також належить синхронізація між вкладками на одному пристрої, та сильний акцент на полегшення розробки local-first застосунків. LiveStore навіть включає в себе окремий набір інструментів для розробників (devtools), які дозволяють легко переглядати і маніпулювати логом подій. Однак, ця бібліотека також не є магічним вирішенням всіх проблем, і містить низку важливих обмежень:

- система вимагає центрального backend для глобального порядку подій, не підтримує повністю децентралізовану або P2P-модель.
- Система ніяк не обробляє конфлікти даних, а лише намагається їх уникнути. Якщо якісь зміни від двох різних користувачів не є сумісними, система просто їх ігнорує. Крім того, оскільки стан формується через materializer-и, які перетворюють події на SQL-зміни, семантична коректність системи значною мірою залежить від правильності реалізації цих функцій розробником. Один неправильно написаний materializer може зламати всю систему.
- Також трохи обмежуючим є адаптованість і кастомізація серверу. Бібліотека включає готові реалізації простих серверів під різні платформи, такі як NodeJS і Cloudflare Workers, однак інтеграція у власний сервер не є тривіальною, оскільки система вимагає чіткого впорядкування усіх подій.

Отже хоча бібліотека і є дуже простою у встановленні і використанні, при роботі з нею дуже легко допускатись помилок, які можуть призвести до повного краху системи.

Electric (Postgres Sync) – описується як двигун синхронізації для зчитування даних з Postgres баз даних, що реплікує підмножини даних із PostgreSQL у локальні застосунки та сервіси через Shapes [12]. Сильна сторона Electric полягає в тому, що він добре інтегрується з існуючою Postgres-інфраструктурою, підтримує часткову реплікацію (не треба надсилати користувачам усю БД) і передає зміни через HTTP API, яке може масштабуватися через CDN. Це робить його привабливим для задач реплікації даних, де потрібно доставляти релевантні підмножини записів на клієнт або інші вузли. Водночас поточна документація показує, що Electric зосереджений саме на читанні, тоді як матеріалізація даних у локальне сховище та підтримка інваріантів залишаються на боці клієнта. Більше того, у документації прямо зазначено, що синхронізація в локальну базу даних є складнішим сценарієм і не покривається бібліотекою. Також додатково описуються обмеження shape-based синхронізації: неповна підтримка деяких SQL запитів, відсутність підтримки рекурсивних таблиць і спеціальні правила для збереження referential integrity. Ще одним обмеженням є те, що для Electric необхідно запускати окремий сервер написаний на Elixir, який сам по собі ніяк не інтегрується в існуючі сервери.

Таким чином з аналізу випливає, що, хоча і є велика кількість рішень, жодне з них не вирішує проблему повністю, а лише добре покриває одну частину, накладаючи обмеження на інші. Саме це обґрунтовує доцільність розробки нової бібліотеки sqlite-sync, як гібридного підходу, що поєднує найкращі елементи існуючих бібліотек для побудови повної комплексної системи, що може бути використана для полегшення розробки local-first застосунків.

2 ПРОЕКТУВАННЯ ГІБРИДНОГО МЕТОДУ СИНХРОНІЗАЦІЇ ТА АРХІТЕКТУРИ БІБЛІОТЕКИ SQLITE-SYNC

Для вирішення проблем local-first веб застосунків необхідне створення такого підходу до синхронізації, який поєднує переваги локальної реляційної бази даних та автоматичного безконфліктного злиття змін. На відміну від систем, де розробник змушений вручну працювати зі спеціалізованими CRDT-структурами, у запропонованій моделі основною одиницею роботи залишається звичайна SQLite-таблиця, з якою велика кількість розробників уже добре знайомі. Це дозволяє зберегти звичний спосіб моделювання даних, SQL-запити та інтеграцію з прикладною логікою, водночас доповнивши реляційну модель автоматичними методами фонової реплікації і синхронізації.

2.1 Поєднання CRDT з реляційною моделлю

Однією з головних проблем використання CRDT у прикладних веб застосунках є те, що більшість класичних CRDT-структур орієнтовані на спеціальні типи даних, такі як лічильники, множини, списки або текстові документи. Натомість реальні інформаційні системи зазвичай будуються на реляційній моделі, де дані організовані у таблиці, зв’язані між собою зовнішніми ключами, а прикладна логіка виражається через SQL запити. Через це пряме використання CRDT часто вимагає повного переосмислення структури даних, що значно ускладнює інтеграцію у звичайні веб додатки.

Основна ідея гібридного підходу полягає в тому, що локальна база даних розглядається як уже матеріалізоване представлення стану, тоді як синхронізація виконується не як пряме копіювання рядків, а через обмін подіями змін. Ці події мають CRDT-властивості на рівні окремих полів

запису, а також зберігаються у вигляді впорядкованого журналу, що наближає систему до моделі Event Sourcing.

Практичний спосіб подолання цієї проблеми добре ілюструє підхід, популяризований James Long у доповіді CRDTs for Mortals [13]. Його суть полягає в тому, що реляційний рядок можна інтерпретувати не як неподільний об’єкт, а як множина пар ключ-значення, де кожна пара має власну часову мітку останнього оновлення. Тоді операції над рядками можна розкласти на менші, незалежні зміни окремих атрибутів. Така інтерпретація дозволяє використати принцип Last-Write-Wins для кожного поля окремо, не відмовляючись від реляційного подання даних. Самі ж таблиці можна представити як Grow-only множини, тобто ті, в яких нові записи можна заносити, але не видаляти. Тоді ці множини легко синхронізувати між собою простою операцією об’єднання. Видалення ж можна реалізувати через додаткову булеву колонку - tombstone, тоді операція видалення просто перетворюється на просту операцію оновлення одного атрибуту.

2.2 Автоматичне перетворення SQL-запитів у послідовність CRDT-подій

Ключовою ідеєю sqlite-sync є розділення матеріалізованого реляційного стану від журналу подій синхронізації. На рівні прикладного коду розробник працює зі звичайними SQLite-таблицями та SQL-запитами, однак усі операції зміни даних автоматично трансформуються у послідовність CRDT-подій, які зберігаються, реплікуються та повторно застосовуються до локального (інші вкладки) або віддаленого сховища. Такий підхід поєднує зручність реляційної моделі з фоновою синхронізацією.

У реалізованій бібліотеці кожна синхронізована сутність описується через дві таблиці. Перша є базовою фізичною таблицею для зберігання

матеріалізованого стану, наприклад _todo або _item. Цю таблицю розробники не мають використовувати напряму. Друга є CRDT-представленням цієї таблиці, наприклад todo або item, через яку прикладний код може вносити зміни і зчитувати актуальну версію даних. Друга таблиця є лише автоматично створеним SQLite View, і не вимагає збереження чи якогось оновлення, а операції запису використовують SQLite тригери з ключовим словом INSTEAD OF [14,15]. Таким чином, для прикладного коду CRDT-таблиця виглядає як звичайна таблиця, хоча фактично є спеціальним шаром над базовим сховищем.

Ця реалізація накладає на базову таблицю мінімальні структурні вимоги. Кожна CRDT-таблиця повинна мати поле id типу TEXT, яке виступає глобальним ідентифікатором запису, а також поле tombstone типу BOOLEAN або INTEGER, яке використовується для маркування видалення. На етапі ініціалізації функція makeCrdtTable перевіряє наявність цих колонок і створює SQL VIEW, що відображає лише активні записи, тобто рядки з tombstone = 0. Саме тому видалені записи фізично залишаються у базі, але не потрапляють у звичайні запити читання. Це дозволяє реплікувати факт видалення так само, як і будь-яке інше оновлення, не порушуючи властивостей збіжності.

Перетворення SQL-операцій у CRDT-події реалізоване через INSTEAD OF тригери, визначені на CRDT-view. Для кожної такої view автоматично створюються три тригери: на INSERT, UPDATE і DELETE. Вони не змінюють базову таблицю напряму, а викликають спеціальні SQLite scalar-функції handle_item_created, handle_item_updated і handle_item_deleted. Отже, коли прикладна логіка виконує звичайний SQL-запит на кшталт insert into todo ..., фактично спрацьовує тригер, який формує відповідну CRDT-подію і передає її до шару синхронізації.

Для операції створення запису тригер серіалізує весь новий рядок у JSON-об’єкт і породжує подію типу item-created. Така подія містить назву набору даних (dataset), ідентифікатор запису (item_id) та повний запис нового

об’єкта. Для операції оновлення система працює більш вибірково. Функція handle_item_updated порівнює старе і нове значення кожної колонки та формує JSON лише з тими атрибутами, які дійсно змінилися. Якщо різниці немає, подія взагалі не створюється. Це зменшує розмір журналу подій і дозволяє реалізувати злиття на рівні окремих полів, а не цілого рядка. Операція видалення не породжує окремий тип події item-deleted; натомість вона зводиться до події item-updated з корисним навантаженням { "tombstone": 1 }. Таким чином, видалення інтерпретується як звичайне оновлення одного атрибута.

Такі операції потім легко передати і застосувати на інших віддалених вузлах – для цього використовується функція createCrdtApplyFunction. Вона працює разом зі окремою службовою таблицею crdt_update_log, у якій для кожного запису та кожного поля зберігається час останнього успішного оновлення. Якщо надходить подія item-created, система вставляє новий рядок у базову таблицю та ініціалізує для всіх його атрибутів однакову часову мітку. Якщо надходить item-updated, система розбирає JSON-подію на окремі атрибути і для кожного з них незалежно порівнює часову мітку події з часовою міткою останнього оновлення цього поля. Атрибут змінюється лише тоді, коли нова мітка є більшою. В іншому випадку зміна ігнорується як застаріла. Саме цей механізм утворює реалізацію Last-Write-Wins на рівні окремих полів, що і забезпечує CRDT-властивість детермінованого безконфліктного злиття.

Таким чином, у sqlite-sync SQL-запити не є кінцевими операціями над базою, а виступають лише зручним і уже знайомим інтерфейсом для автоматичного створення CRDT-подій. Така організація дозволяє зберегти звичну реляційну модель даних і водночас реалізувати синхронізацію на основі послідовних подій, придатну для журналювання, реплікації, дедуплікації та безконфліктного злиття.

2.3 Використання Hybrid Logical Clock для впорядкування подій

Однією з ключових вимог до запропонованої системи синхронізації є наявність стабільного механізму впорядкування подій. У розподіленому та offline-first покладатися лише на фізичний час пристрою є поганою ідеєю, оскільки годинники різних клієнтів можуть бути не синхронізованими, а мережеві затримки здатні змінювати порядок доставки повідомлень. У такій ситуації одна і та сама подія може бути створена раніше, але отримана пізніше, ніж інша. Якщо система буде спиратися тільки на звичайні часові мітки, це може призвести до неправильного злиття змін і втрати причинно-наслідкових зв'язків між ними.

Для розв'язання проблеми впорядкування подій у sqlite-sync використано Hybrid Logical Clock (HLC) – підхід, запропонований Sandeep S. Kulkarni, Murat Demirbas, Deepak Madeppa, Bharadwaj Avva та Marcelo Leone у 2014 році [16]. У своїй роботі автори запропонували поєднати фізичний час із логічним лічильником, щоб одночасно зберігати читабельність реального часу та гарантувати коректне впорядкування подій у розподіленій системі, навіть якщо декілька вузлів створюють записи одночасно.

У загальному випадку HLC можна подати як трійку: t, c, n.

- t – фізичний час на пристрої у мілісекундах
- c – логічний лічильник
- n – унікальний ідентифікатор вузла, на якому було створено подію

У бібліотеці sqlite-sync кожна подія синхронізації отримує власну HLC-мітку часу під час запису в журнал подій. На клієнтській стороні окремий екземпляр годинника підтримується для кожної вкладки браузера, яка виступає незалежним вузлом синхронізації. На серверній стороні

аналогічний механізм може використовуватися для впорядкування подій у межах віддаленого сховища.

Алгоритм формування нової HLC-мітки працює таким чином. Якщо поточний фізичний час пристрою є більшим за попереднє значення t, то нова мітка отримує це нове значення часу, а логічний лічильник c скидається до нуля. Якщо ж фізичний час не збільшився, тобто нова подія виникла в межах того самого моменту або локальний годинник відстає, тоді фізичний компонент залишається незмінним, а логічний лічильник збільшується на одиницю. У результаті навіть кілька подій, створених практично одночасно на одному вузлі, все одно отримують унікальні та впорядковані часові позначки.

Не менш важливим є механізм злиття HLC при отриманні віддалених подій. Коли вузол отримує подію з іншого клієнта або сервера, він не лише застосовує її до локального стану, а й оновлює власний HLC так, щоб усі подальші локальні події мали мітки, більші за вже отриману. Якщо віддалена мітка часу є більшою за локальну, вузол переходить на це значення часу і продовжує відлік лічильника від нього. Якщо часові значення збігаються, використовується найбільший лічильник із подальшим збільшенням. Якщо ж локальний годинник уже випереджає віддалений, тоді збільшується лише локальний логічний лічильник. Така схема гарантує монотонність часових міток і запобігає появі нових локальних подій, які могли б отримати “старіший” час, ніж уже оброблені віддалені зміни.

У sqlite-sync саме HLC використовується для підтримки Last-Write-Wins. Для кожного запису та для кожного окремого атрибута в службовій таблиці crdt_update_log зберігається час останнього успішного оновлення. Коли надходить нова подія типу item-updated, система порівнює її HLC-мітку з уже наявною міткою для конкретного поля. Якщо нова мітка є більшою, значення поля оновлюється; якщо ж вона виявляється застарілою, зміна ігнорується. У випадку офлайн-роботи це особливо важливо, оскільки

клієнт може накопичувати події локально, а після відновлення зв’язку передавати їх на сервер без втрати їхнього порядку виникнення.

2.4 Архітектура системи: in-memory SQLite, OPFS Worker, Cloudflare Durable Objects

Архітектура sqlite-sync побудована як багаторівнева система зберігання даних, у якій кожен рівень виконує окрему роль:

1. in-memory SQLite у вкладці відповідає за миттєву взаємодію з інтерфейсом
2. OPFS-backed SQLite у Web Worker забезпечує довготривале локальне збереження та координацію між вкладками [17]
3. Cloudflare Durable Objects виступають віддаленим вузлом синхронізації між різними клієнтами [18]

Така побудова дозволяє поєднати низьку затримку локальної роботи з надійною реплікацією та централізованим розповсюдженням подій. Фактично система утворює багато рівневу ієрархію main-replica: для користувацького інтерфейсу головною є локальна база у пам’яті, для групи вкладок одного браузера координатором є worker-рівень, а для множини клієнтів глобальним вузлом виступає сервер – Durable Object.

Розглянемо кожний з цих рівнів більш детально. Перший рівень архітектури реалізовано як in-memory SQLite, що працює безпосередньо у вкладці браузера на головному потоці. Саме цей шар використовується іншими компонентами для синхронних запитів читання, локальних записів та реактивного оновлення інтерфейсу. Його головна перевага полягає в тому, що всі операції, критичні для UX, виконуються без очікування мережі або

міжпотокової комунікації. Під час ініціалізації вкладка не створює базу з нуля, а отримує снапшот збереженого стану від worker-рівня, після чого розгортає його у пам’яті та реєструє CRDT-представлення таблиць, view та тригери. Завдяки цьому прикладний код працює зі звичайними SQL-таблицями і реактивними запитами, не взаємодіючи напряму з журналом подій. Усі локальні insert, update і delete спершу застосовуються саме тут, що дає майже миттєвий відгук інтерфейсу навіть у повністю офлайн-режимі.

Другий рівень становить Web Worker, у якому розміщено SQLite-базу, яка зберігає дані в OPFS (Origin Private File System) [19]. Цей шар виконує роль довготривалого локального сховища, яке зберігається між перезапуском сторінки, а також бере на себе всю фонову синхронізацію. Використання worker-а ізолює важчі операції введення-виведення та обміну подіями від головного потоку, а OPFS дає змогу працювати з реальною файловою SQLite-базою всередині браузера. На цьому рівні підтримуються службові таблиці системи, журнал подій, стан синхронізації та міграції схеми. Після запуску worker ініціалізує базу з OPFS, застосовує системні та прикладні міграції, а далі на запит активної вкладки формує снапшот, який завантажується в in-memory-копію. Коли користувач виконує локальні зміни, вкладка не лише оновлює власну копію, а й передає сформовані CRDT-події до worker-а. Той зберігає їх у стійкому журналі, застосовує до локального матеріалізованого стану, а після цього розсилає сповіщення іншим вкладкам. Таким чином worker виступає спільним фоновим координатором для всіх вкладок одного браузера.

Одним з обмежень OPFS є те, що ця система не підтримує одночасного доступу до файлової системи. Один файл може читати або змінювати лише один клієнт. Тому для уникнення конфліктного доступу до OPFS використовується механізм Web Locks [20], що дозволяє фактично підтримувати один активний worker для конкретної бази даних і коректно завершувати його роботу, коли клієнтські вкладки закриті.

Третій рівень архітектури – координаційний сервер – реалізовано на основі Cloudflare Durable Objects, однак за необхідності його можна легко адаптувати під інші платформи, або взагалі інтегрувати в існуючий сервіс. Цей шар відповідає за віддалену синхронізацію між різними клієнтами, які працюють з однією логічною сутністю, наприклад окремим списком чи документом. Durable Object є природним вибором для такого завдання, оскільки він надає одиночний серверний екземпляр із власним SQLite-сховищем і послідовною обробкою запитів. Завдяки цьому всі вхідні події проходять через один вузол, де можуть бути збережені, дедуплікувані, застосовані до серверного матеріалізованого стану та одразу розіслані підключеним клієнтам через WebSocket. На серверному рівні також ведеться власний журнал crdt_events, зберігаються метадані останніх оновлень для реалізації Last-Write-Wins, виконуються міграції схеми та формується потік подій для подальшої відправки клієнтами.

Взаємодія між цими трьома рівнями утворює повний цикл життєдіяльності даних. Після відкриття вкладки OPFS Worker ініціалізує локальну і стійку базу та передає її снапшот у in-memory SQLite. Користувач працює з локальною копією синхронно і без мережевих затримок. Кожна зміна автоматично перетворюється на CRDT-подію, яка спершу застосовується локально, потім зберігається й реплікується у worker, а вже далі надсилається на серверний рівень у Cloudflare Durable Object. Після обробки сервер розсилає інформацію про нові події іншим клієнтам, а їхні worker-и виконують pull, застосовують зміни до своїх OPFS-баз і повідомляють вкладки про оновлення. У результаті користувач отримує систему, що одночасно забезпечує миттєвий локальний відгук, стійкість до перезавантаження і втрати мережі, синхронізацію між вкладками та коректну реплікацію між віддаленими клієнтами, а розробник може розробляти застосунок із знайомими технологіями і спрощеною моделлю даних, адже вся синхронізація виконується автоматично на фоні. Саме така багатошарова

архітектура і є практичною основою запропонованого гібридного методу синхронізації в бібліотеці sqlite-sync.

2.5 Можливості розширення системи для складніших правил злиття, наближених до OT

Базова модель sqlite-sync спирається на CRDT-події та правило Last-Write-Wins на рівні окремих атрибутів, що добре працює для більшості стандартних сценаріїв синхронізації і дозволяє уникнути конфліктів. Однак у прикладних системах часто виникають ситуації, де простого безконфліктного злиття недостатньо, оскільки важливо зберегти не лише технічну узгодженість стану, а й семантичний намір користувача. Саме тому запропонована архітектура передбачає можливість розширення серверного рівня додатковими правилами обробки подій, які за своїм призначенням наближаються до Operational Transformation.

Ідея такого розширення полягає в тому, що серверний вузол синхронізації не обмежується пасивним прийманням і ретрансляцією подій. Оскільки всі зміни проходять через централізований координаційний рівень, сервер може реагувати на події певного типу, що стосуються конкретних таблиць, колонок або сутностей, і одразу після застосуванням змін виконувати додатковий аналіз контексту. Це дає змогу реалізовувати не лише дедуплікацію, валідацію або нормалізацію даних, а й більш складну логіку трансформації змін залежно від поточного стану системи.

Практично це може бути реалізовано у вигляді серверних хуків або обробників доменних подій. Після отримання чергової CRDT-події сервер спочатку визначає, чи належить вона до набору сутностей, для яких задано спеціальні правила злиття. Якщо так, то перед записом у серверний журнал і

розсиланням клієнтам запускається додатковий етап обробки. На цьому етапі система може виконати пошук пов’язаних записів, перевірити бізнес-інваріанти, перетворити одну вхідну подію на кілька похідних, об’єднати її з уже наявними змінами або навіть відхилити як некоректну. Важливо, що результатом такої обробки все одно залишаються звичайні події синхронізації, які можуть бути журналізовані, реплiковані та повторно відтворені на інших вузлах.

Наприклад, якщо два клієнти майже одночасно створюють сутності з дуже схожими назвами, сервер може виконати дедуплікацію за наперед визначеними правилами: порівняти нормалізовані імена, знайти ймовірний збіг, об’єднати записи або створити службову подію, яка позначить їх як подібні. Аналогічно можна реалізувати і більш складну логіку перевірки референтної цілісності або збереження складних доменних обмежень, які не покриваються звичайним LWW-злиттям.

Таким чином, серверний рівень у sqlite-sync може виступати не лише як засіб централізованого впорядкування подій, а і як точка впровадження спеціалізованої трансформаційної логіки. У найпростішому випадку система працює як CRDT-орієнтований механізм автоматичного безконфліктного злиття. У складніших прикладних сценаріях вона може бути розширена правилами, які за своєю суттю наближаються до OT, оскільки враховують тип операції, контекст змін і бажаний семантичний результат. Саме така гібридність є однією з ключових переваг запропонованого підходу: стандартні випадки обробляються автоматично, тоді як складні конфлікти можуть розв’язуватися через цільові серверні механізми без необхідності повністю відмовлятися від реляційної моделі чи базової CRDT-архітектури.

3 РЕАЛІЗАЦІЯ ПРОГРАМНОГО КОМПЛЕКСУ SQLITE-SYNC

3.1 Структура monorepo та склад основних пакетів

Програмний комплекс sqlite-sync реалізовано у вигляді pnpm monorepo, що дозволило об’єднати в одному репозиторії як модульні пакети бібліотеки, так і прикладні застосунки, які використовуються для демонстрації, тестування та експериментальної перевірки запропонованого підходу. Така організація є доцільною для системи цього типу, оскільки всі її складові мають спільну кодову базу, тісно пов’язані між собою на рівні типів і контрактів та розвиваються синхронно. Monorepo-підхід також спрощує повторне використання коду, локальну розробку пакетів без окремої публікації, контроль версій та узгоджене оновлення залежностей [21].

На верхньому рівні репозиторію розміщено корневі файли конфігурації: package.json, pnpm-workspace.yaml, спільний tsconfig.json, а також Biome, який використовується для форматування та статичного аналізу коду.

Основні команди моно репозиторію охоплюють збирання (pnpm build), перевірку правильності типізації (pnpm typecheck), форматування (pnpm format) та публікацію бібліотечних пакетів. Завдяки цьому весь програмний комплекс розглядається як єдина система, а не набір ізольованих підпроектів.

Структурно monorepo поділено на дві головні частини: packages і apps. Директорія packages містить основні бібліотечні модулі, які утворюють ядро платформи sqlite-sync, тоді як apps включає прикладні застосунки та середовища для тестування архітектурних рішень (Рис 3.1).

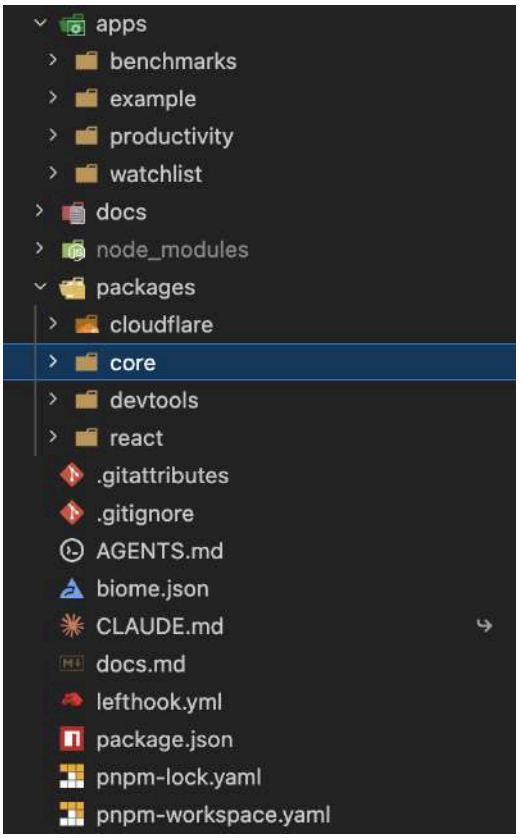

Рисунок 3.1 - Файлова структура проекту

До складу бібліотечних пакетів входять такі основні модулі:

1. Пакет @sqlite-sync/core

   Це центральний пакет усієї системи, у якому реалізовано базову логіку синхронізації. Саме тут зосереджені основні механізми роботи з CRDT-подіями, Hybrid Logical Clock, локальними SQLite-базами, міграціями, створенням синхронізованих таблиць, застосуванням подій до матеріалізованого стану та запуском worker-рівня. Фактично @sqlite-sync/core є ядром платформи, від якого залежать усі інші модулі.

2. Пакет @sqlite-sync/react

   Цей пакет містить React-інтеграцію для бібліотеки. Його призначення полягає в тому, щоб надати зручний механізм підключення синхронізованої бази даних до React-застосунку спеціальні хуки. Зокрема, пакет інкапсулює створення DbProvider, доступ до бази через

useDb, а також реактивні механізми зчитування даних на кшталт useDbQuery і useDbState. Наявність окремого React-пакета дозволяє відокремити UI-залежну частину від платформного ядра та зберегти універсальність core, який може використовуватися і поза React-середовищем. За необхідністю бібліотеку можна розширити додавши підтримку інших UI-бібліотек.

3. Пакет @sqlite-sync/cloudflare

   Цей модуль реалізує серверну інтеграцію для платформи Cloudflare, насамперед для Durable Objects і D1. У ньому містяться адаптери та допоміжні інструменти для створення віддаленого вузла синхронізації, що приймає CRDT-події від клієнтів, зберігає їх у серверному SQLite-сховищі, виконує дедуплікацію, застосовує до матеріалізованого стану та розсилає іншим підключеним клієнтам. Окремо винесено підтримку міграцій і виконання типізованих SQL-запитів у середовищі Cloudflare. Завдяки цьому серверна частина синхронізації не змішується з клієнтською логікою, а реалізується як самостійний модуль, який також може бути прикладом для реалізації підтримки інших платформ.

Окрім бібліотечних пакетів, monorepo містить кілька прикладних застосунків, що розміщені в директорії apps:

1. apps/example – це демонстраційний застосунок, який показує базове використання sqlite-sync у браузері разом із Web Worker, React та Cloudflare-сумісним сервером синхронізації. Він виконує роль мінімального прикладу інтеграції бібліотеки в реальний клієнтський застосунок.
2. apps/watchlist – повноцінний прикладний застосунок для керування спільними списками перегляду фільмів чи серіалів. Він побудований на базі TanStack Router, oRPC, Cloudflare D1 та пакетів sqlite-sync. Саме на

цьому застосунку перевіряється практична придатність архітектури для колаборативної роботи з даними в умовах offline-first.

3. apps/benchmarks – окремий застосунок для проведення експериментальних вимірювань продуктивності. У ньому використовуються tinybench та @sqlite-sync/core для оцінювання характеристик локальних операцій, роботи CRDT-механізмів та інших критичних частин системи [22].

Така організація спрощує супровід системи, повторне використання коду та подальше розширення бібліотеки новими платформними адаптерами або прикладними рішеннями.

3.2 Реалізація @sqlite-sync/core: CRDT, журнал подій, міграції, sync engine

Пакет @sqlite-sync/core реалізує технологічне ядро всієї системи синхронізації і публікується як окремий модуль monorepo з трьома точками входу: основний клієнтський API, worker-runtime та server-side контракт для мережевого обміну. На рівні коду цей пакет побудовано не як один великий модуль, а як набір вузькоспеціалізованих підсистем: обгортка над SQLite WASM, механізм реактивного in-memory доступу до БД, CRDT-шар, журнал подій, міграційний механізм і рушій синхронізації. Такий поділ дозволив ізолювати складність окремих частин і водночас повторно використовувати однакові примітиви як у вкладці браузера, так і у worker-рівні.

Базовим інфраструктурним компонентом пакета є клас SQLiteDbWrapper, який приховує низькорівневу роботу з @sqlite.org/sqlite-wasm і надає уніфікований інтерфейс для виконання SQL-запитів, чи то звичайних, чи то скомпільованих Kysely-виразів [23].

Окремо в ньому реалізовано кешування підготовлених запитів, інвалідацію метаданих схеми після міграцій, створення снапшотів бази та реєстрацію scalar-функцій SQLite, які використовуються CRDT-шаром. Саме ця абстракція зробила можливим однаково працювати з in-memory базою у вкладці та з OPFS-backed базою у worker без дублювання логіки застосування подій чи міграцій.

Реалізація CRDT побудована навколо декларативного опису синхронізованої схеми через createSyncDbSchema. Цей конструктор не лише збирає список CRDT-таблиць, а й формує типізовані проекції схеми для різних контекстів виконання. У результаті одна і та сама конфігурація виступає одночасно джерелом метаданих і TypeScript-контрактом для запитів. Під час ініціалізації in-memory бази функція createMemoryDb застосовує ініціалізує спеціальну внутрішню схему, після чого для кожної синхронізованої таблиці викликається makeCrdtTable, яка автоматично створює VIEW та INSTEAD OF тригери. Тут важливим є саме спосіб інкапсуляції: прикладний код працює зі звичайною таблицею з точки зору SQL, тоді як перехоплення змін повністю сховане у згенерованому шарі SQLite-об’єктів.

Додатково реалізовано важливу гарантію цілісності: події вважаються остаточно створеними лише після transaction-committed; якщо транзакція відкочується, прапорець застосування очищується, і жодне сповіщення про нові події не розсилається. Отже, CRDT-журнал синхронізується не з окремими SQL-інструкціями, а з фактом успішного завершення транзакції.

Журнал подій реалізований через універсальний компонент createCrdtStorage. Саме він відповідає за присвоєння sync_id, фіксацію походження події (local, own, remote), її статусу (pending, applied, failed, skipped), а також за послідовне застосування черги змін. Спочатку подія записується як pending, далі проходить через чергу обробки: за потреби зливається з локальним HLC-станом, мігрується до актуальної версії схеми,

застосовується до матеріалізованої таблиці та після цього отримує фінальний статус.

Окрему роль у пакеті відіграє механізм міграцій. У @sqlite-sync/core мігрується не лише фізична схема БД, а й сам потік подій. Для цього використано конструктор createMigrations, який дозволяє описувати міграції декларативно через операції на кшталт createTable, renameTable, renameColumn, addColumn, dropColumn та dropTable. Кожний такий крок компілюється у SQL і, за потреби, одночасно генерує eventTransformer для історичних CRDT-подій. Наприклад, при перейменуванні таблиці трансформується поле dataset, при перейменуванні колонки переноситься відповідний ключ у payload, при видаленні колонки її значення вилучається зі старих подій, а якщо після цього item-updated втрачає весь зміст, така подія може бути відкинута. Це важлива властивість системи, оскільки вона дає змогу змінювати прикладну схему без розриву сумісності з уже накопиченим журналом синхронізації. In-memory база при цьому не виконує повноцінні міграції самостійно, а отримує вже підготовлений снапшот від worker.

Рушій синхронізації у core об’єднує локальне сховище, worker і віддалене джерело в одну керовану сесію. Функція createSyncedDb координує повний життєвий цикл клієнтської БД: створює ідентифікатори екземпляра та вкладки, підключається до worker через BroadcastChannel, отримує снапшот бази, розгортає його в SQLiteReactiveDb, ініціалізує CRDT-сховище і запускає двосторонню синхронізацію з worker-рівнем. Для узгодження доступу використовуються Web Locks: вкладка тримає shared-lock клієнта, а worker отримує exclusive-lock на конкретну базу. Такий підхід унеможливлює одночасний конфліктний доступ до OPFS і дозволяє коректно завершувати worker після закриття всіх вкладок.

Сама web worker-реалізація зосереджена у startDbWorker. Під час запуску вона ініціалізує OPFS SAH pool VFS, відкриває основну базу та окремо під’єднує службову worker-базу через ATTACH DATABASE. Далі застосовуються PRAGMA-налаштування, системні та прикладні міграції,

формується CrdtStorage, а поверх нього запускається createCrdtSyncRemoteSource, який відповідає за pull/push цикл з віддаленим джерелом. Важливо, що sync engine реалізовано незалежним від механізму передачі даних: віддалений вузол описується як remoteFactory з методами pullEvents і pushEvents, а WebSocket-підключення є лише однією з можливих реалізацій, оформленою окремим адаптером createWsRemoteSource. Завдяки цьому ядро пакета не прив'язане жорстко до конкретного протоколу чи бекенду.

Таким чином core модуль реалізований як набір незалежних модулів, що можна легко компонувати.

3.3 Реалізація @sqlite-sync/react: реактивна SQLite база даних

Пакет @sqlite-sync/react реалізує тонкий інтеграційний шар між React-застосунком і клієнтською базою даних, створеною в @sqlite-sync/core. Його призначення полягає у зручному підключенні вже ініціалізованої реактивної SQLite-бази користувацького інтерфейсу та автоматичному оновленні інтерфейсу при зміні даних.

Основною точкою входу пакета є функція createDbContext, яка створює React-context для конкретної синхронізованої бази даних. На її основі формуються DbProvider, useDb, useDbQuery та useDbState. Компонент useDb надає прямий доступ до клієнтської БД, useDbState дозволяє отримувати стан worker-рівня та мережевої синхронізації, а useDbQuery використовується для реактивного виконання SQL-запитів.

Реактивність побудована навколо механізму live query, який надається з @sqlite-sync/core. Коли компонент викликає useDbQuery, переданий запит компілюється до SQL-рядка і набору параметрів, після чого для нього

створюється або повторно використовується відповідна live query-підписка. Якщо декілька компонентів використовують один і той самий SQL-запит з однаковими параметрами, пакет не створює дубльовані підписки, а ділить одну спільну підписку між кількома споживачами. Це зменшує кількість зайвих обчислень і робить реактивний шар ефективнішим.

На рівні @sqlite-sync/core реактивність реалізована в компоненті SQLiteReactiveDb, який працює поверх in-memory SQLite і відстежує зміни безпосередньо через нативні функції SQLite. Під час будь-якої модифікації даних, незалежно від того, чи вона походить із локального SQL-запиту, чи з застосування віддаленої CRDT-події, sqlite3_update_hook накопичує назви змінених таблиць у межах поточної транзакції. Після успішного commit спрацьовує sqlite3_commit_hook, який фіксує завершення транзакції та асинхронно розсилає повідомлення лише для тих таблиць, які дійсно були змінені. Якщо ж транзакція відкочується, sqlite3_rollback_hook очищає чергу змін, і жодні реактивні оновлення не публікуються [24].

Основою реактивного читання є механізм live query. Коли прикладний код створює реактивний SQL-запит, система компілює його в SQL і визначає, від яких таблиць він залежить. Для цього використовується SQLite-функція tables_used(...) [25], а для окремих випадків на кшталт DELETE з повним очищенням таблиці додатково застосовується аналіз інструкцій EXPLAIN, оскільки стандартний механізм не завжди коректно повертає таку залежність. Після цього live query підписується на події змін відповідних таблиць. Коли одна з них оновлюється, запит перевиконується, кеш рядків замінюється новим результатом, а підписники отримують сповіщення про зміну. Для зменшення кількості зайвих перерахунків оновлення проходять через короткий debounce, тому серія швидких змін об’єднується в один цикл оновлення.

Важливо, що реактивність у @sqlite-sync/core працює на рівні самої бази даних, а не UI-фреймворку. Пакет @sqlite-sync/react лише адаптує цей механізм до useSyncExternalStore, тоді як уся логіка відстеження змін,

визначення залежних таблиць, кешування prepared statements і перевиконання запитів зосереджена саме в core. Це дозволяє використовувати однакову модель реактивного доступу до SQLite незалежно від конкретного інтерфейсного шару.

3.4 Реалізація @sqlite-sync/cloudflare: приклад готового серверного адаптера

Пакет @sqlite-sync/cloudflare реалізує готовий серверний адаптер для розгортання віддаленого вузла синхронізації на платформі Cloudflare. Його головне призначення полягає в тому, щоб поєднати готові CRDT-механізми з @sqlite-sync/core із середовищем Durable Objects, яке надає послідовну обробку запитів, вбудоване SQLite-сховище та зручну модель довготривалого серверного стану [18]. У результаті цей пакет виступає прикладом повноцінної серверної реалізації, яка може приймати події від клієнтів, зберігати їх, застосовувати до матеріалізованого стану та розсилати оновлення іншим підключеним вузлам.

Архітектурно пакет складається з трьох основних частин: SQL-виконавця для DurableObjectStorage, механізму міграцій і власне адаптера серверної синхронізації.

Всередині адаптера створюється серверне CRDT-сховище на базі createCrdtStorage. Воно використовує HLCCounter для генерації часових міток, веде монотонний sync_id, зберігає журнал подій у таблиці crdt_events і застосовує події до матеріалізованого стану через createCrdtApplyFunction.

Для інформування клієнтів про появу нових змін використовується окремий механізм сповіщення. Після того як чергова подія успішно застосована і отримує статус applied, createCrdtSyncProducer викликає broadcastPayload(...) з коротким повідомленням events-applied, яке містить

лише новий sync_id. Такий підхід дозволяє не розсилати повний вміст подій усім клієнтам одразу. Натомість підключені вкладки отримують сигнал про наявність нових змін і самостійно виконують pull, завантажуючи лише ті записи, яких їм бракує. Це зменшує обсяг дубльованого мережевого трафіку і добре узгоджується з ідеєю журналу подій як основного каналу реплікації.

4 ЕКСПЕРИМЕНТАЛЬНА ПЕРЕВІРКА ТА ОЦІНЮВАННЯ SQLITE-SYNC

4.1 Мета експериментальної перевірки та критерії оцінювання

Метою експериментальної перевірки є підтвердження практичної придатності розробленої бібліотеки sqlite-sync для побудови offline-first вебзастосунків із колаборативним редагуванням. Експерименти мають показати, що запропонований гібридний підхід, забезпечує коректну синхронізацію даних, збіжність реплік після обміну подіями та прийнятну продуктивність у реальних сценаріях використання.

У межах експериментальної перевірки оцінюються кілька основних аспектів системи. По-перше, перевіряється функціональна коректність синхронізації: локальні зміни мають коректно перетворюватися на CRDT-події, передаватися між вкладками та клієнтами, а також відтворювати однаковий фінальний стан у всіх репліках. По-друге, оцінюється стійкість системи до офлайн-режиму, тобто здатність накопичувати зміни локально та правильно синхронізувати їх після відновлення з’єднання. По-третє, аналізується продуктивність основних операцій, зокрема накладні витрати CRDT-шару порівняно зі звичайними SQL-операціями, швидкість обробки подій та вартість ініціалізації бази зі снапшота.

Усі вимірювання продуктивності виконувалися на одному й тому самому апаратному та програмному середовищі: Macbook air M1 (2020), MacOS 14.8 - 8gb оперативної пам'яті і 8 ядер процесор. Для кожного сценарію проводилося N незалежних раундів. Перед кожним раундом стан бази даних відновлювався з однакового снапшота, щоб усунути вплив попередніх операцій на результат. Час виконання вимірювався за допомогою performance.now() лише для цільової операції; етап підготовки середовища не

включався до вимірювання. Для кожного сценарію наведено середнє значення часу.

Таким чином, експериментальна частина роботи спрямована не лише на демонстрацію працездатності sqlite-sync, а й на кількісне та якісне оцінювання того, наскільки запропонована архітектура є придатною для використання у практичних веб-застосунках.

4.2 Практичне використання sqlite-sync для розробки веб-застосунку

4.2.1 apps/example як базовий сценарій інтеграції бібліотеки в застосунок

Для демонстрації практичного використання бібліотеки sqlite-sync у складі monorepo було реалізовано застосунок apps/example. Його основне призначення полягає в тому, щоб показати типовий і мінімально необхідний шлях інтеграції бібліотеки у звичайний клієнтський веб-застосунок, побудований на React. На відміну від складніших прикладних систем, цей застосунок не перевантажений доменною логікою, серверною авторизацією чи великою кількістю сутностей. Завдяки цьому він добре підходить для демонстрації базових принципів роботи sqlite-sync і того, як саме розробник повинен використовувати бібліотеку під час створення offline-first застосунку.

У прикладі реалізовано простий список задач todo. Незважаючи на простоту предметної області, цей сценарій охоплює всі ключові механізми бібліотеки: створення синхронізованої схеми, ініціалізацію локальної бази, підключення worker-рівня, виконання реактивних SQL-запитів, локальні зміни даних та їх автоматичну реплікацію.

Інтеграція починається з опису схеми даних. Для цього використовується декларативний механізм міграцій createMigrations(...), у якому розробник визначає фізичну SQLite-таблицю _todo. Вона містить мінімально необхідний набір колонок: id, title, completed і tombstone. Після цього на основі міграцій створюється синхронізована схема через createSyncDbSchema(...), де явно вказується, що базова фізична таблиця _todo має бути представлена в прикладному коді як CRDT-таблиця todo.

Наступний крок інтеграції полягає в ініціалізації локальної синхронізованої бази. У apps/example для цього використовується функція createSyncedDb(...), якій передаються ідентифікатор бази, екземпляр worker-а та опис синхронізованої схеми. На цьому етапі бібліотека створює повний клієнтський стек: in-memory SQLite для миттєвої взаємодії з інтерфейсом, окремий Web Worker для довготривалого збереження в OPFS та канал синхронізації з віддаленим вузлом через WebSocket. Для самого прикладного коду вся ця багаторівнева архітектура схована за відносно простою процедурою ініціалізації.

Читання даних у apps/example реалізовані через useDbQuery(...). Наприклад, список задач отримується звичайним SQL запитом до таблиці todo, а статистика обчислюється іншим запитом із функціями count та sum. У цьому випадку важливо, що розробник формулює саме декларативний запит до SQLite, а не працює з окремим клієнтським сховищем у стилі state-management бібліотек. При зміні відповідних таблиць бібліотека автоматично перевиконує live query і оновлює інтерфейс. Таким чином, локальна SQLite-база фактично стає основним реактивним шаром даних застосунку.

Записи в базу також виконуються у звичний спосіб. Додавання задачі реалізовано через insert, зміна стану виконання через update, а видалення через delete. Ці операції спрямовані на таблицю todo, тобто на CRDT-представлення сутності, однак з точки зору прикладного коду вони виглядають як звичайні SQL-запити. Це і є одна з центральних ідей усього

підходу: розробник не працює напряму ні з CRDT-структурами, ні з журналом подій, ні з логікою дедуплікації чи впорядкування. Усі ці технічні аспекти залишаються всередині бібліотеки. При цьому кожна локальна операція одразу застосовується до in-memory копії, що забезпечує миттєвий відгук інтерфейсу, а вже потім у фоновому режимі потрапляє до worker-а, зберігається в журналі подій і передається на сервер.

Отже, apps/example виконує роль навчального та архітектурного зразка. Він показує, що sqlite-sync може бути використана як повноцінний локальний data layer для веб-застосунку, де всі читання і записи виконуються локально, інтерфейс оновлюється динамічно, а синхронізація з іншими вузлами відбувається автоматично у фоновому режимі. Саме на цьому прикладі добре видно головну практичну перевагу бібліотеки: складні механізми CRDT, event sourcing і offline-first синхронізації вбудовуються під уже знайому для розробника модель SQL-таблиць і простих CRUD-операцій.

4.2.2 apps/watchlist як складніший приклад використання з авторизацією та системою доступу

Якщо apps/example демонструє мінімальний сценарій використання бібліотеки, то застосунок apps/watchlist показує, як sqlite-sync може бути використана в набагато більш реалістичному й прикладному середовищі. Це повноцінний веб-застосунок для керування списками перегляду фільмів і серіалів, у якому поєднуються локальна синхронізована база, віддалений бекенд, авторизація користувачів, розмежування доступу та спільна робота з даними. Саме цей застосунок найбільш наочно демонструє, що бібліотека придатна не лише для простих експериментів, а і для побудови складніших local-first систем із колаборативним редагуванням.

Архітектурно watchlist складається з двох логічно пов’язаних, але різних частин. Перша частина є традиційною серверною інфраструктурою

застосунку: таблиці користувачів, сесій, списків і зв’язків між користувачами та списками, а також набір серверних процедур для авторизації, створення списків, запрошення учасників та перевірки прав доступу. Друга частина є локальною синхронізованою базою даних окремого списку, яка працює поверх sqlite-sync і відповідає вже за самі елементи watchlist, їх редагування, сортування, позначення перегляду та інші зміни вмісту.

Така побудова є дуже показовою. Вона демонструє, що в реальному застосунку не всі дані обов’язково мають бути синхронізовані через один і той самий механізм. Метадані системи, пов’язані з обліковими записами, сесіями, членством у списках і контролем доступу, зберігаються у звичайній серверній базі. Натомість дані, для яких важливі офлайн-робота, реактивність і колаборативне редагування, винесені в окремий синхронізований простір на основі sqlite-sync. Це дозволяє використовувати бібліотеку саме там, де її модель дає найбільшу користь, не намагаючись примусово переносити до CRDT-шару всю систему загалом.

У застосунку реалізовано автентифікацію двома способами: через magic link на електронну пошту та через Google OAuth. Після успішної перевірки користувача створюється запис сесії у серверній базі, а в браузері зберігається підписаний cookie з ідентифікатором сесії. Подальші захищені серверні процедури виконуються лише після перевірки цієї сесії.

Сама модель доступу до даних у watchlist побудована доволі просто. На сервері існує таблиця list, яка представляє логічні списки, і таблиця user_to_list, що задає зв’язок між користувачами та списками, до яких вони мають доступ. Кожен створений список має автора createdBy, а також набір учасників, які можуть працювати з його вмістом. Усі звернення до конкретного списку проходять через перевірку наявності такого зв’язку. Якщо користувач не є учасником списку, доступ до нього забороняється.

На практиці це означає, що sqlite-sync працює в умовах уже не абстрактної, а контрольованої багатокористувацької системи. Синхронізація відбувається не між усіма клієнтами глобально, а лише між тими вузлами, які

працюють із конкретною сутністю та мають на це право. У watchlist такою сутністю є окремий список перегляду. Саме тут проявляється ще одна сильна сторона архітектури бібліотеки: синхронізацію можна організовувати не на рівні всієї програми, а на рівні конкретного документа, списку, проекту чи іншого ізольованого простору даних.

У клієнтській частині це реалізовано через ініціалізацію окремої локальної бази для кожного listId. Коли користувач відкриває певний список, застосунок спочатку через серверний шар перевіряє, чи має він право доступу до цього списку. Якщо доступ дозволений, клієнт ініціалізує локальну синхронізовану базу даних саме для цього списку. Ідентифікатор списку передається у worker, а той, своєю чергою, підключається до окремої кімнати синхронізації на віддаленому сервері. Тобто кожен список має власний ізольований канал реплікації, власну локальну БД і власний простір подій.

Під час роботи зі списком користувач може додавати нові елементи, змінювати їхній стан, редагувати окремі атрибути, додавати оцінки й теги. Усі ці зміни спочатку відображаються в локальній in-memory базі, що забезпечує швидкий інтерфейс без очікування сервера. Далі події потрапляють у worker-рівень, зберігаються в локальному журналі та реплікуються на віддалений вузол синхронізації, реалізований як окремий серверний екземпляр для конкретного списку. Якщо інший користувач, що має доступ до того самого списку, відкриє його в іншому браузері чи на іншому пристрої, він отримає той самий потік подій і побачить узгоджений стан даних.

Цікавим є і те, що серверний вузол у watchlist використовується не лише для пасивної реплікації. Поверх нього розміщено додаткові прикладні серверні процедури, пов’язані, наприклад, із налаштуваннями списку або AI-обробкою даних. Коли новий елемент додається до списку, сервер автоматично запускає фоновий аналіз тегів цього нового елементу, і потім асинхронно оновлює їх. Через фонову синхронізацію користувачі бачать ці зміни одразу.

З погляду практичного проєктування watchlist демонструє дуже важливий висновок: local-first синхронізація найкраще працює тоді, коли вона поєднується зі звичайним серверним рівнем, що відповідає за автентифікацію, авторизацію, організацію доступу та інші системні функції. Бібліотека sqlite-sync у такій моделі не замінює бекенд повністю, а бере на себе саме ту частину задачі, яка пов’язана з локальним збереженням, реактивною роботою з даними та реплікацією змін між учасниками. Побачити як виглядає застосунок watchlist можна на рисунку 4.1

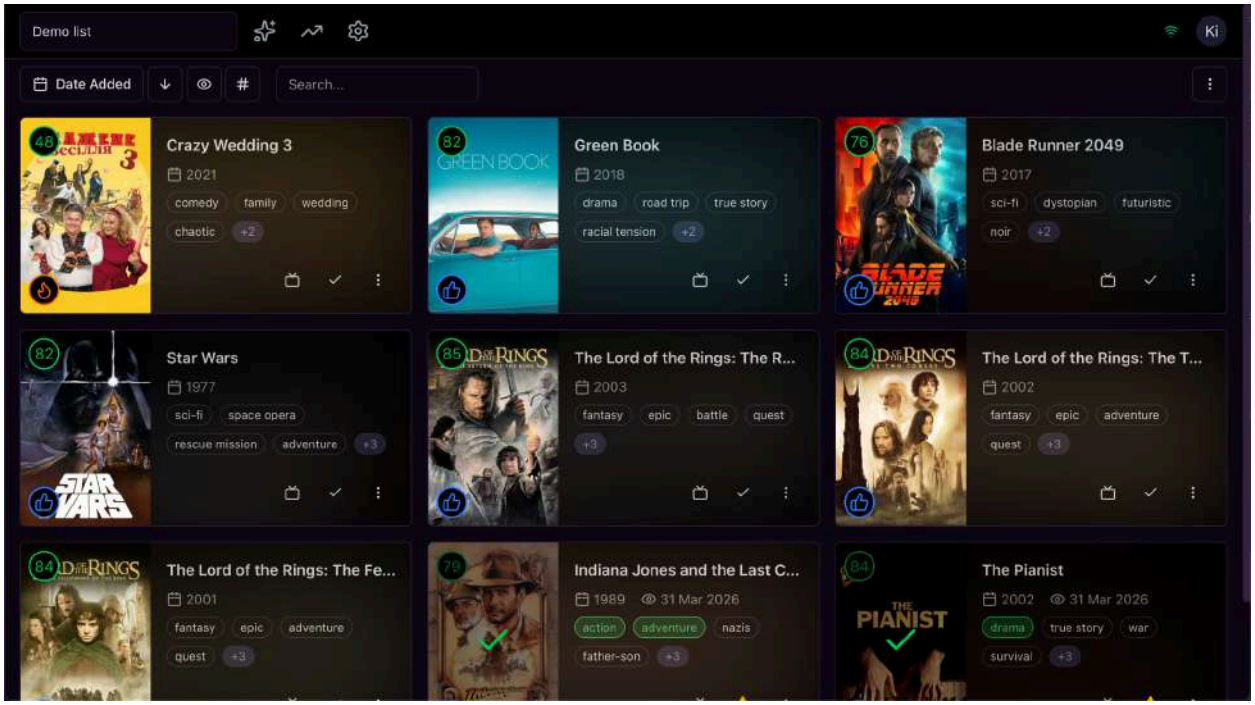

Рисунок 4.1 Застосунок watchlist

Отже, apps/watchlist є більш складним і значно ближчим до реального використання прикладом застосування sqlite-sync. Він показує, що бібліотека здатна працювати в умовах багатокористувацької системи з авторизацією, сесіями, членством у групах і розмежуванням прав доступу. Водночас принцип інтеграції з боку клієнтської логіки залишається таким самим, як і в простому прикладі: розробник описує синхронізовану схему, ініціалізує локальну SQLite-базу для потрібного простору даних і далі працює з нею як

зі звичайною реляційною базою. Саме ця комбінація простоти інтеграції та можливості використання в реалістичному багатокористувацькому середовищі і демонструє практичну цінність бібліотеки sqlite-sync.

4.3 Перевірка коректності синхронізації реплік

Одним із ключових етапів експериментальної перевірки бібліотеки sqlite-sync є підтвердження того, що запропонований механізм синхронізації дійсно забезпечує коректне поширення змін між вузлами та приводить усі копії даних до узгодженого фінального стану. Необхідно також запевнетись, що система зберігає працездатність за умов кількох одночасно відкритих вкладок, тимчасової відсутності мережевого з’єднання та паралельних змін одних і тих самих даних.

У межах цієї роботи під коректністю синхронізації розуміється така поведінка системи, за якої будь-яка локальна зміна, виконана в одній репліці, після завершення обміну подіями коректно відображається в інших репліках відповідно до правил обробки подій. Під збіжністю реплік, своєю чергою, розуміється досягнення однакового матеріалізованого стану даних на всіх вузлах після того, як усі накопичені події були доставлені та застосовані. Тобто після завершення синхронізації всі клієнти повинні містити однаковий набір записів і однакові значення їхніх атрибутів, незалежно від порядку доставки подій або тимчасової автономної роботи окремих вузлів.

Для перевірки цих властивостей було розглянуто три характерні сценарії, які відображають типові умови використання local-first веб-застосунку: синхронізація між кількома вкладками та клієнтами, робота в офлайн-режимі та паралельне редагування сутностей.

4.3.1 Синхронізація між різними клієнтами і вкладками

Перший сценарій перевірки моделює звичайну роботу користувачів у мережевому середовищі, коли один і той самий набір даних відкрито одразу у кількох контекстах. Для цього було використано два клієнти, і на першому клієнті одночасно відкрито дві різні вкладки браузера. Таким чином, перевірка охоплює два рівні синхронізації: локальну між вкладками одного браузера та віддалену між різними клієнтами.

У межах цього сценарію в першій вкладці клієнта A послідовно виконувалися операції створення, редагування та видалення записів. Очікувана поведінка системи полягала в тому, що друга вкладка того самого клієнта повинна отримати ці зміни через локальний, а вкладка клієнта B повинна отримати ті самі події через сервер.

Результати цього сценарію показали, що sqlite-sync коректно підтримує багаторівневу реплікацію (Рис. 4.2).

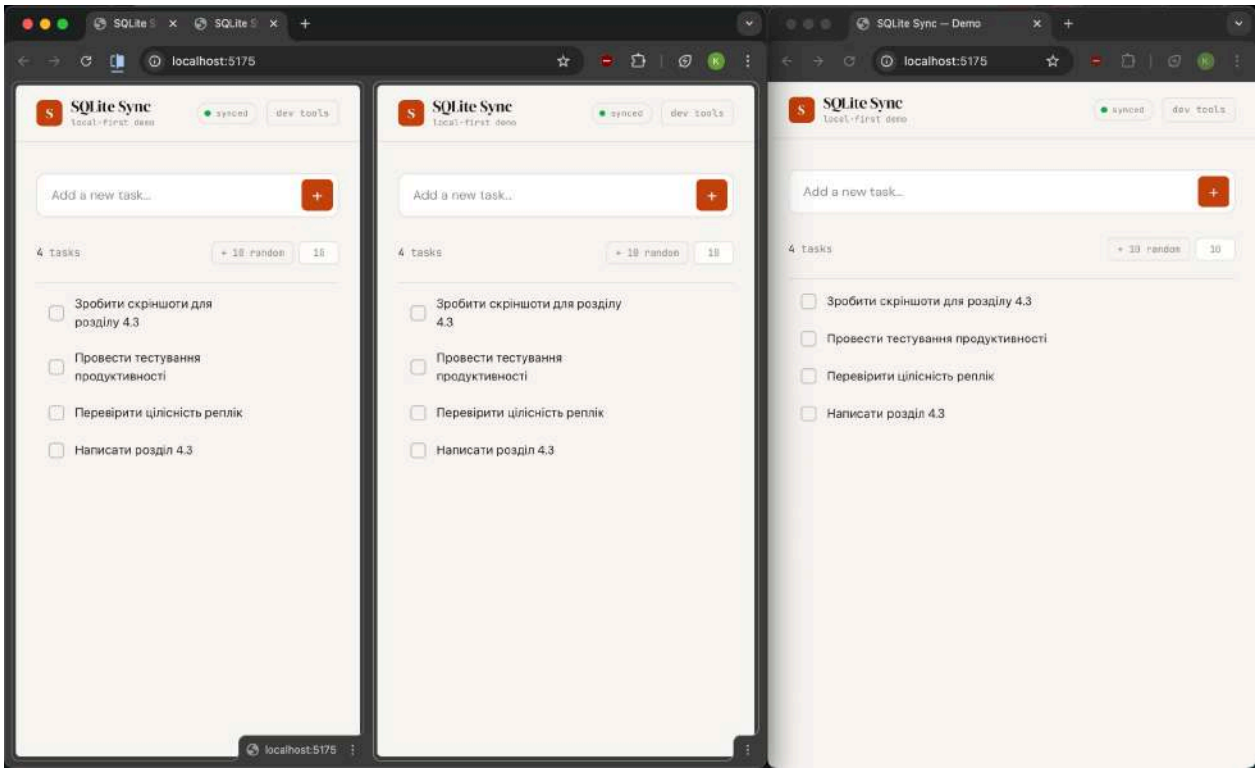

Рисунок 4.2 Дві синхронізовані вкладки одного клієнта і окрема вкладка другого клієнта

Зміни, виконані в одній вкладці, майже одразу ставали видимими в іншій вкладці того самого клієнта, що підтверджує правильну роботу worker-рівня як локального координатора стану. Після проходження через віддалений вузол синхронізації ці ж самі зміни відображалися і на іншому клієнті. Важливо, що після завершення обміну подіями всі три репліки містили однаковий стан даних. Це підтверджує, що система коректно поєднує локальну синхронізацію між вкладками та глобальну синхронізацію між різними клієнтами без порушення узгодженості

4.3.2 Синхронізація в офлайн-режимі

Другий сценарій спрямований на перевірку однієї з головних властивостей offline-first архітектури, а саме здатності системи коректно працювати за відсутності з’єднання з мережею та відновлювати узгоджений стан після повторного підключення. Для цього один із клієнтів тимчасово переводився в офлайн-режим, після чого на ньому виконувалася серія локальних змін: створення нових записів, редагування наявних і видалення частини даних. У цей самий час інший клієнт залишався онлайн і продовжував працювати зі своєю копією даних. На рисунку 4.3 можна побачити два синхронізованих клієнта, де перший клієнт від’єднався від мережі, але ще не виконував ніякі зміни.

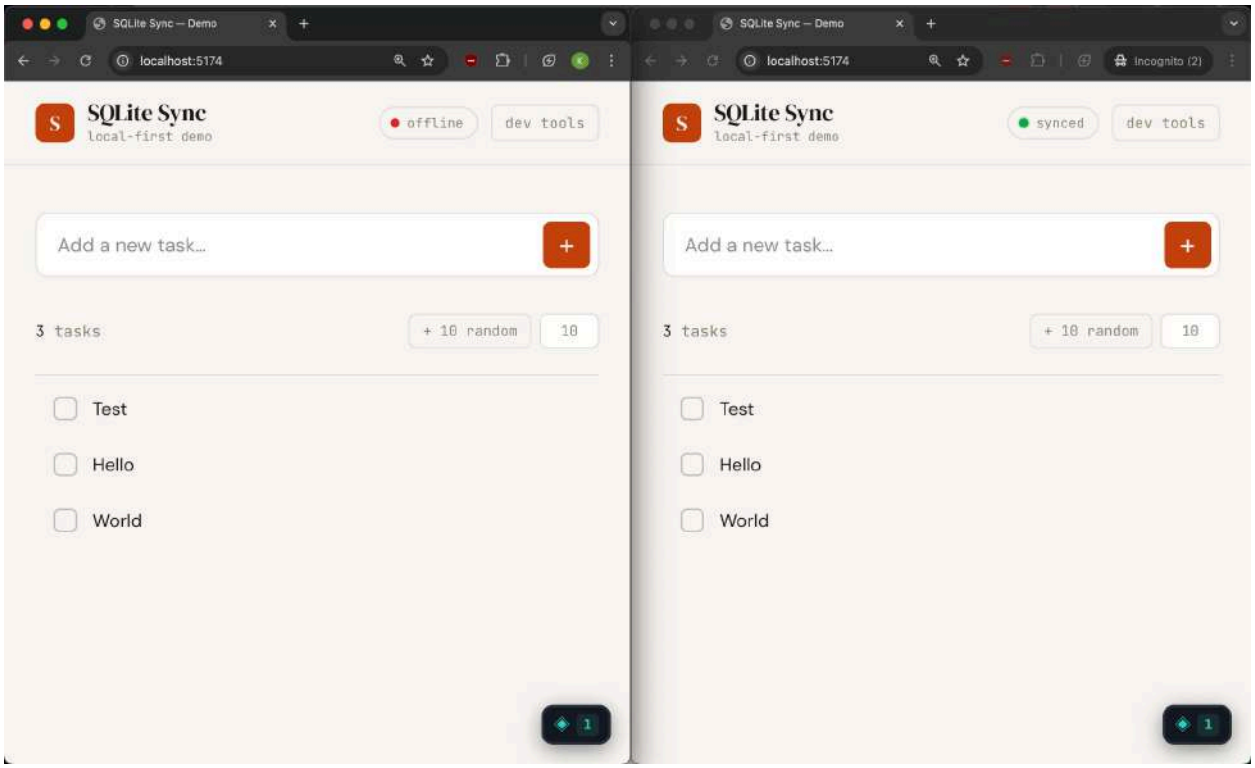

Рис 4.3 Клієнти синхронізовані. Лівий - оффлайн, правий - онлайн

Обидва клієнта можуть асинхронно вносити зміни, і для обох застосунок працює як треба – зміни одразу відображаються, що видно на рисунку 4.4.

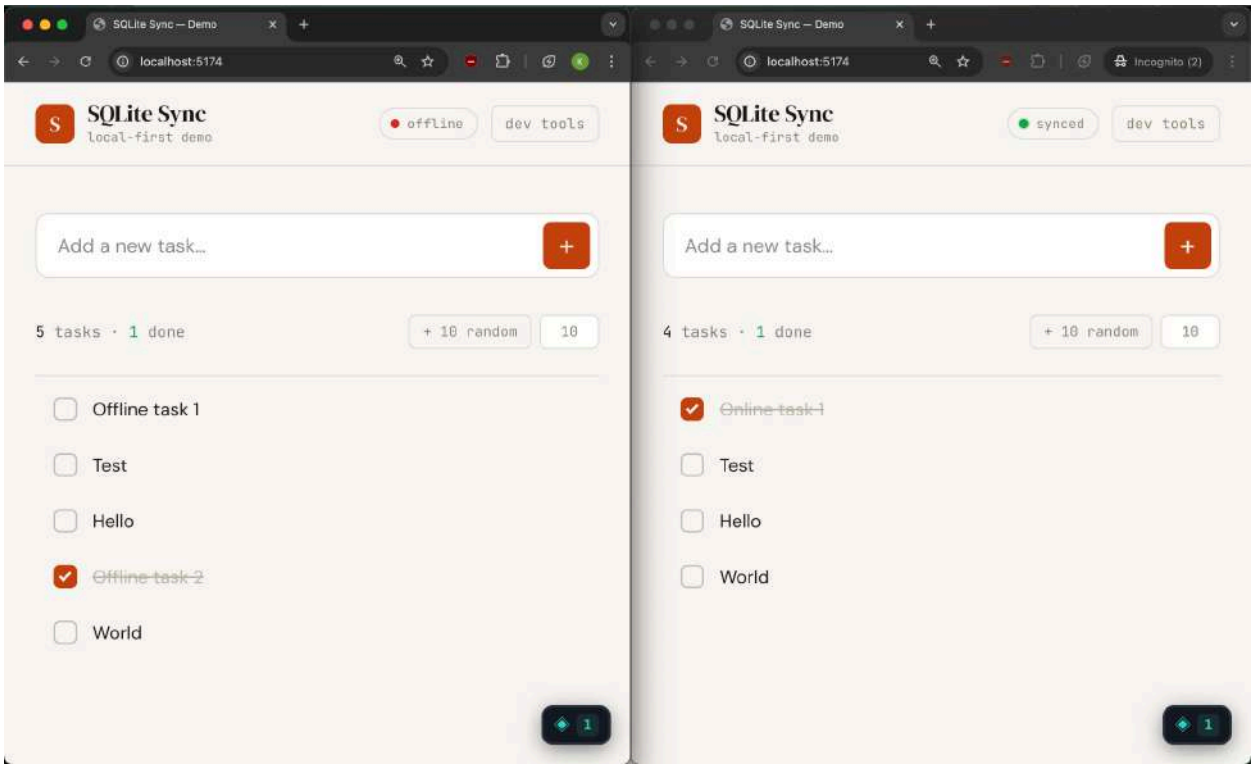

Рис 4.4 Обидва клієнта внесли зміни

Далі, лівого клієнта можна знову під’єднати до серверу, після чого він одразу надішле свої зміни, і завантажить сторонні зміни з сервера. Правий клієнт отримує сповіщення, що сервер отримав нові зміни, після чого правий клієнт їх завантажує. В результаті – обидва клієнта приходять до однакового стану, що видно на рисунку 4.5.

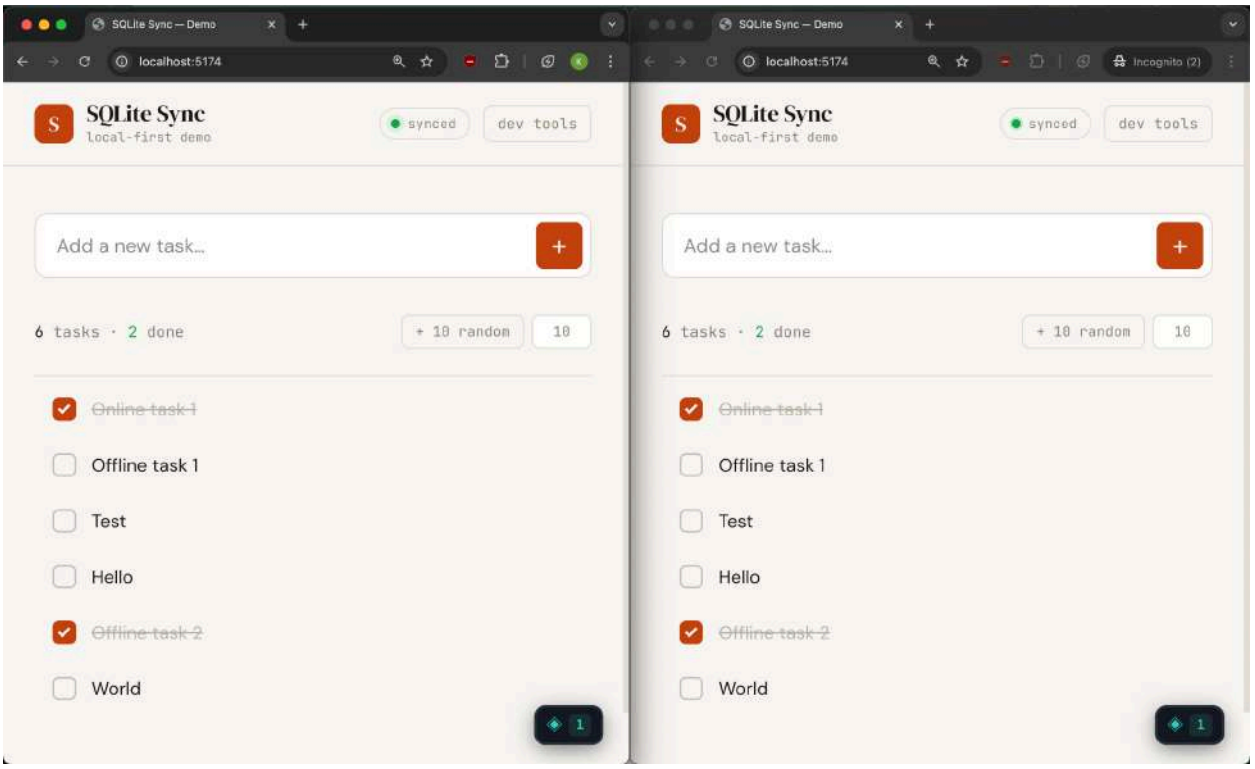

Рис 4.5 Клієнти успішно синхронізуються

Експеримент показав, що така модель працює коректно. Під час відсутності мережі застосунок залишався повністю функціональним, оскільки всі операції запису й читання виконувалися локально в in-memory SQLite та worker-базі. Після повторного підключення накопичені події були успішно синхронізовані із сервером, а потім доставлені на інші клієнти. У результаті після завершення обміну всі репліки знову збігалися за станом. Це підтверджує, що система не лише підтримує офлайн-роботу, але й коректно інтегрує відкладені зміни у загальний потік синхронізації без втрати даних і без потреби у ручному втручанні користувача.

4.3.3 Паралельне редагування сутностей

Окрему увагу під час експериментальної перевірки було приділено сценаріям паралельного редагування сутностей, оскільки саме вони є найбільш критичними для систем із колаборативною роботою. Якщо кілька

реплік майже одночасно змінюють одну й ту саму сутність, система синхронізації повинна не лише доставити всі події, а й детерміновано обчислити однаковий фінальний стан на всіх вузлах. Для перевірки цієї властивості в межах пакета @sqlite-sync/core було реалізовано автоматизовані інтеграційні тести, які моделюють роботу двох незалежних реплік, локальне внесення змін у кожній з них та подальший обмін CRDT-подіями між цими репліками.

Такі тести неможливо зробити вручну, оскільки вони вимагають неймовірної точності аж до мікросекунди. Автоматизовані тести ж дозволяють точно відтворювати конфліктні сценарії, контролювати часові мітки HLC і перевіряти фінальний стан даних після завершення синхронізації. У тестах використовувалася модель двох реплік, кожна з яких мала власне локальне сховище подій, власний лічильник Hybrid Logical Clock і власну матеріалізовану копію даних. Після виконання локальних змін у кожній репліці події експортувалися, передавалися іншій репліці та повторно застосовувалися через той самий механізм CRDT-обробки, який використовується в реальній бібліотеці. Критерієм успішності в усіх випадках було досягнення однакового фінального стану запису в обох репліках.

Перший тестовий сценарій перевіряв паралельне редагування різних полів одного запису. В одній репліці змінювалося поле title, а в іншій незалежно від нього поле completed. Результати тесту показали, що після обміну подіями обидві репліки містили однаковий запис, у якому було збережено обидва оновлення одночасно. Це підтверджує, що система не втрачає зміни, якщо вони стосуються різних полів однієї сутності, і коректно реалізує CRDT-злиття на рівні атрибутів.

Другий тестовий сценарій перевіряв більш конфліктний випадок, коли дві репліки одночасно змінюють одне й те саме поле. В обох вузлах редагувалося поле title, з HLC мітками які відрізняються лише ідентифікатором вузла. У цьому випадку система повинна застосувати

правило Last-Write-Wins, тобто залишити значення з більшою часовою міткою та відкинути застаріле. Автоматичний тест підтвердив саме таку поведінку: після синхронізації обидві репліки містили однакове фінальне значення поля title, що відповідало події з більшою HLC-міткою. Таким чином було підтверджено, що механізм Hybrid Logical Clock разом із Last-Write-Wins забезпечує детерміноване розв’язання конфлікту при конкурентному редагуванні одного атрибута.

Третій сценарій перевіряв конфлікт між оновленням запису та його видаленням. В одній репліці змінювалося поле title, а в іншій майже одночасно для того самого запису встановлювався атрибут tombstone, який у sqlite-sync представляє логічне видалення сутності. Результати тесту показали, що після синхронізації обидві репліки сходилися до одного й того самого стану: зміна заголовка зберігалася, а запис одночасно позначався як видалений через tombstone. Це підтверджує, що навіть у випадку конфлікту між модифікацією та логічним видаленням система зберігає збіжність реплік і не породжує неоднозначного фінального результату.

Отже, автоматизовані інтеграційні тести підтвердили коректність роботи sqlite-sync у трьох основних сценаріях паралельного редагування: при незалежній зміні різних атрибутів, при конкурентній зміні одного поля та при конфлікті між оновленням і видаленням. У всіх випадках після завершення обміну подіями обидві репліки переходили до однакового матеріалізованого стану. Це дає підстави стверджувати, що реалізований у бібліотеці механізм CRDT-подій, доповнений Hybrid Logical Clock, забезпечує детерміноване злиття змін і збіжність реплік у типових сценаріях колаборативного редагування.

4.4 Оцінювання продуктивності та бенчмарки

Для оцінювання продуктивності розробленої бібліотеки було проведено серію бенчмарків, спрямованих на вимірювання накладних витрат CRDT-шару, швидкості застосування подій синхронізації та вартості ініціалізації локальної бази зі збереженого снапшота. Основна мета цього етапу полягала не лише у фіксації абсолютних числових показників, а й у визначенні того, наскільки додатковий рівень автоматичної синхронізації впливає на локальні SQL-операції та чи залишається продуктивність системи придатною для практичного використання.

4.4.1 Накладні витрати локальних операцій запису

Перший набір бенчмарків порівнював середній час локальних операцій insert, update та delete у звичайній SQLite і в sqlite-sync. Вимірювання виконувалися для трьох типових розмірів навантаження: 1, 10 і 1000 рядків за один workload. Результат можна побачити на таблиці 4.6

Кількість подій	Операція	SQL, середній час	SQLite-sync, середній час	Оверхед
1	insert	0.050	0.187	3.7x
1	update	0.040	0.474	11.9x
1	delete	0.039	0.182	4.7x
10	insert	0.119	0.728	6.1x
10	update	0.155	1.393	9.0x
10	delete	0.137	0.973	7.1x
1000	insert	4.400	60.540	13.8x

1000	update	13.860	100.020	7.2x
1000	delete	10.620	86.380	8.1x

Таблиця 4.6 Результати першого бенчмарку

Отримані результати підтверджують наявність помітного, але очікуваного оверхеду. Для одиничних операцій абсолютний час усе ще залишається дуже малим: вставка одного запису в sqlite-sync у середньому займає 0.187 мс, а видалення 0.182 мс. Найдорожчою операцією виявилося оновлення одного запису, що пояснюється необхідністю порівняння старого і нового стану, формування часткового payload, оновлення журналу подій і службової таблиці crdt_update_log.

Зі збільшенням розміру workload накладні витрати зростають, що також є очікуваним, оскільки для кожного рядка система виконує додаткові дії, пов’язані із CRDT-представленням. Найбільший оверхед спостерігається для масової вставки 1000 рядків: 60.54 мс проти 4.4 мс у plain SQLite. Водночас навіть у цьому випадку мова йде про десятки мілісекунд, а не секунди, що є прийнятним для багатьох локальних застосунків, особливо якщо такі масові операції виконуються не в головному потоці інтерфейсу.

З практичної точки зору ці результати означають, що sqlite-sync не прагне бути швидшою за чисту SQLite, а свідомо обмінює частину продуктивності на автоматичне журналювання, безконфліктне злиття та подальшу синхронізацію між вузлами. Для звичайних користувацьких сценаріїв, де зміни здебільшого є дрібними й відбуваються по одному або невеликими пакетами, такий оверхед виглядає прийнятним.

4.4.2 Пропускна здатність застосування CRDT-подій

Другий набір бенчмарків оцінював пропускну здатність застосування віддалених CRDT-подій до матеріалізованого стану бази даних. У цьому тесті

вимірювалося, наскільки швидко система може обробити великий пакет із 10 000 подій різних типів. Результати можна побачити на рисунку 4.7.

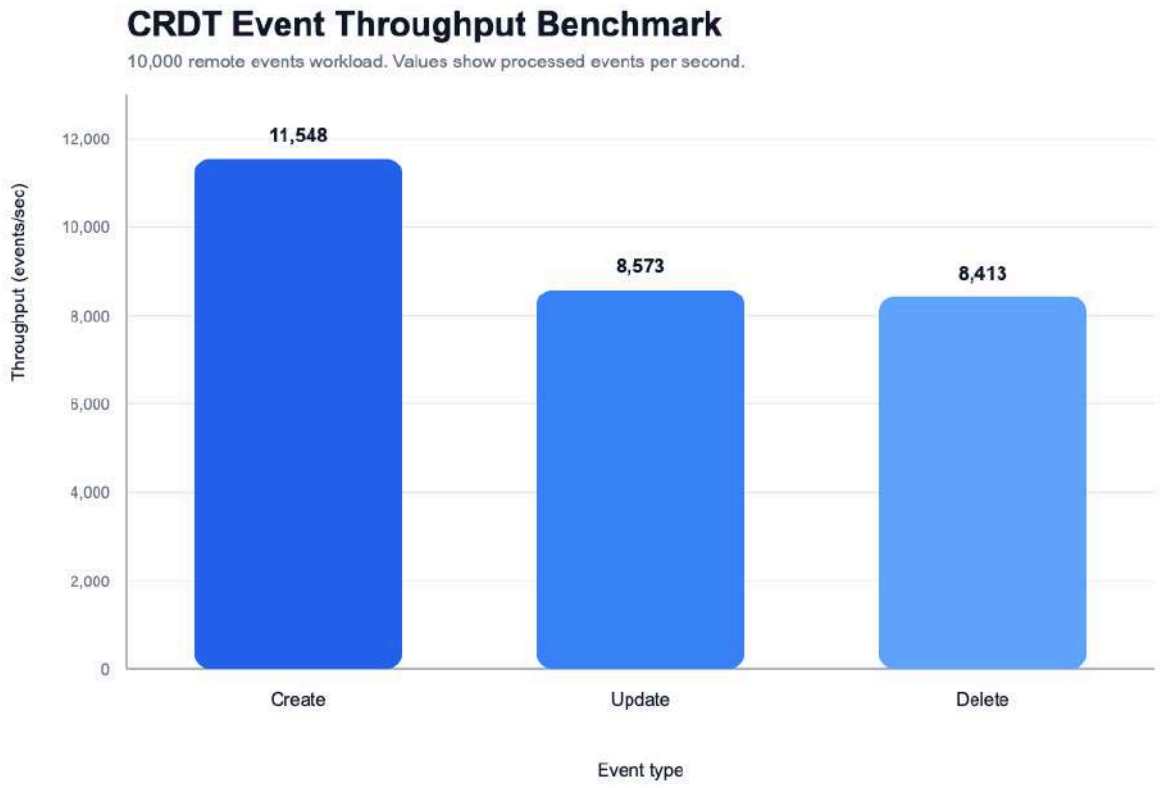

Рисунок 4.7 Результати бенчмарку пропускної здатності системи

Як видно з результатів, найвищу пропускну здатність мають події створення: понад 11.5 тис. подій за секунду. Це логічно, оскільки для item-created система виконує відносно прямолінійне вставлення нового запису та ініціалізацію часових міток для його атрибутів. Оновлення і видалення виявилися дещо дорожчими: приблизно 8.4–8.6 тис. подій за секунду. У цьому випадку система повинна не просто записати нові значення, а й для кожного поля порівняти часову мітку події з уже наявною міткою останнього оновлення, тобто фактично виконати CRDT-злиття на рівні атрибутів.

Такі результати свідчать, що навіть при масовому надходженні подій система зберігає високу пропускну здатність. Це є важливим для сценаріїв

початкової синхронізації, повторного pull після офлайн-режиму або реплікації великої кількості змін між кількома клієнтами. З урахуванням того, що типове користувацьке навантаження в прикладних застосунках значно менше за десятки тисяч подій за раз, отримана продуктивність виглядає достатньою для практичного використання.

4.4.3 Ініціалізація бази даних зі снапшота

Третій набір бенчмарків спрямований на оцінку того, наскільки швидко система може експортувати локальний снапшот бази та відновити з нього нову або вже існуючу in-memory SQLite-базу. Цей аспект є особливо важливим для архітектури sqlite-sync, оскільки під час відкриття нової вкладки клієнт отримує саме снапшот від worker-рівня, а не повністю відтворює стан з нуля.

Для оцінки було використано кілька розмірів локальної бази. На рисунку 4.8 можна побачити результати.

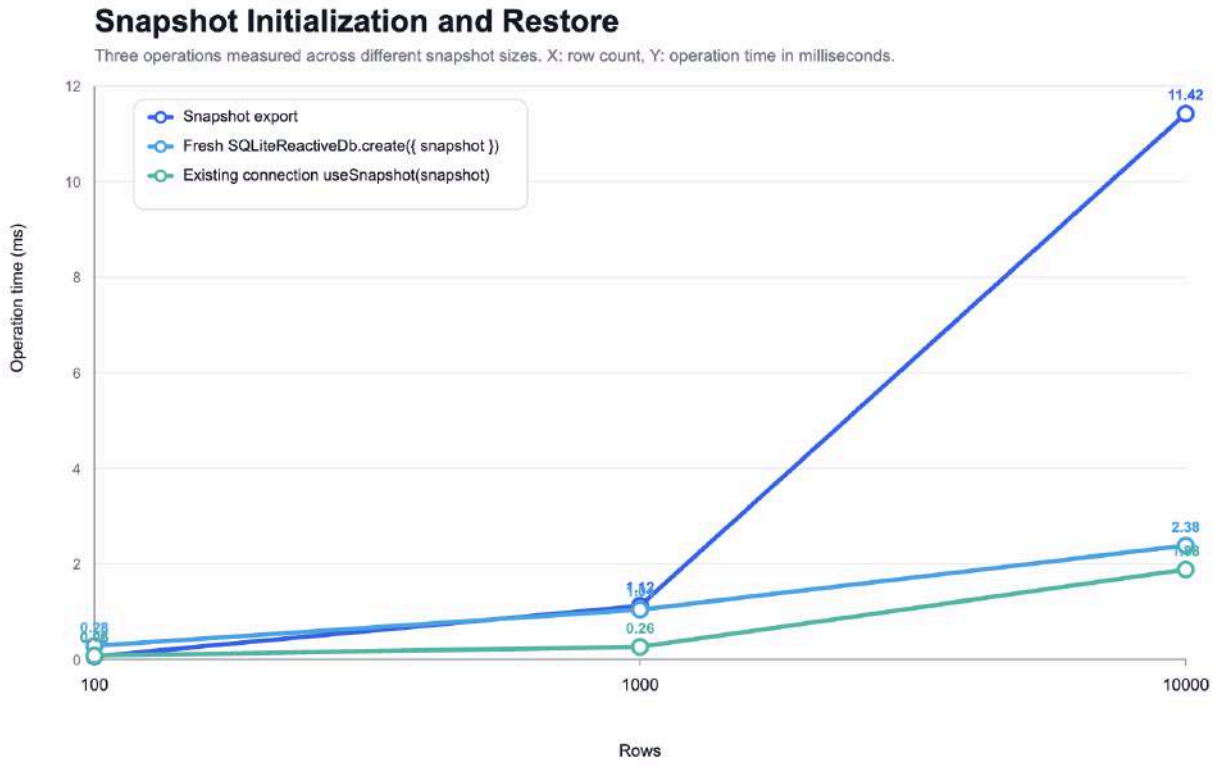

Рисунок 4.8 Виміри часу ініціалізації

Для невеликих обсягів даних операції експорту та відновлення займають частки мілісекунди або одиниці мілісекунд. Навіть для 10 000 рядків повне створення нової реактивної бази зі снапшота займає в середньому лише 2.38 мс, а застосування снапшота до вже відкритого з'єднання 1.88 мс. Найдорожчою операцією в цьому випадку є сам експорт снапшота, який для 10 000 рядків займає 11.42 мс.

З часом і з відповідним збільшенням розміру бази даних буде і збільшуватись час ініціалізації застосунку. Це серйозна проблема, яку можна вирішувати через механізм компакції – коли дані з часом будуть оптимізуватись, прибираючи непотрібні івенти з локальної бази даних.

Загалом, проведене оцінювання показало, що sqlite-sync дійсно створює помітний оверхед порівняно зі звичайною SQLite під час локальних операцій запису, але це і було очікувано. Цей результат є природним наслідком додаткової логіки, пов'язаної з автоматичним перетворенням SQL-змін у CRDT-події, веденням журналу та підтримкою службових часових міток.

Проте в абсолютних значеннях для типових малих workload-ів ці витрати залишаються невеликими й придатними для інтерактивних застосунків.

4.5 Аналіз результатів та обмеження підходу

Проведена експериментальна перевірка дає підстави стверджувати, що розроблена бібліотека sqlite-sync є зручним інструментом для побудови offline-first вебзастосунків із колаборативним редагуванням. Запропонований гібридний підхід, який поєднує CRDT, Event Sourcing та можливість подальшого розширення логікою, наближеною до OT, дозволяє забезпечити коректну синхронізацію даних між кількома клієнтами, із збереженням працездатності в офлайн режимі.

Автоматизовані інтеграційні тести також підтвердили, що запропонований механізм Hybrid Logical Clock у поєднанні з Last-Write-Wins на рівні окремих атрибутів забезпечує детерміноване злиття змін і однаковий фінальний стан на всіх репліках.

Загалом, до сильних сторін бібліотеки можна віднести такі аспекти:

- збереження звичної реляційної моделі даних і можливість працювати зі звичайними SQLite-таблицями та SQL-запитами без переходу на спеціалізовані CRDT-структури;
- прозоре автоматичне перетворення локальних SQL-операцій у CRDT-події за рахунок view і SQLite-trigger’ів;
- підтримка offline-first моделі, за якої всі читання і записи виконуються локально, а синхронізація відбувається у фоновому режимі;
- коректна багаторівнева синхронізація між in-memory SQLite, OPFS worker-рівнем і віддаленим серверним сховищем;

- підтримка колаборативного редагування з детермінованим безконфліктним злиттям змін на рівні окремих атрибутів;
- гнучка система міграцій, яка адаптує як уже існуючі дані, так і самі події оновлення цих даних
- модульна архітектура проекту, що дозволяє розділити core-логіку, React-інтеграцію та серверні адаптери;
- наявність реактивної обгортки над звичайними SQL запитами, що значно полегшує побудову динамічних користувацьких інтерфейсів

Водночас проведене дослідження виявило і низку обмежень запропонованого підходу.

Першим важливим недоліком є додатковий оверхед автоматичної CRDT-системи порівняно зі звичайною SQLite. Цей оверхед виникає через необхідність перехоплення записів через INSTEAD OF trigger’и, серіалізації змін у події, ведення службових таблиць та порівняння HLC-міток для кожного атрибута. Хоча абсолютні показники продуктивності залишаються прийнятними для більшості інтерактивних сценаріїв, система все ж істотно повільніша за звичайний SQLite при масових операціях. Потенційно це обмеження можна зменшити завдяки більш нативній інтеграції механізмів синхронізації у WebAssembly-рівень SQLite, без проміжного шару JSON-подій і scalar-функцій.

Другим суттєвим обмеженням є відсутність повноцінної підтримки unique constraints і foreign keys для синхронізованих CRDT-таблиць. У класичній реляційній моделі такі обмеження забезпечуються централізовано і синхронно, тоді як у розподіленому середовищі клієнти можуть незалежно створювати конфліктні записи або посилання на ще не синхронізовані сутності. Через це стандартні механізми SQLite не можуть бути безпосередньо перенесені в CRDT-шар без ризику порушення збіжності реплік. Теоретично це можна розв’язати шляхом винесення перевірки таких інваріантів на серверний рівень та реалізації додаткової трансформаційної

логіки, подібної до OT, коли сервер аналізує контекст і коригує або відхиляє конфліктні події.

Третім недоліком є нескінченне зростання журналу подій. Оскільки вся історія змін зберігається як послідовність CRDT-подій, з часом обсяг локальної та серверної бази неминуче збільшується. Це може негативно впливати швидкість початкової синхронізації та вартість відновлення стану на нових клієнтах. Практичним способом розв’язання цієї проблеми є компакція журналу: агрегування застарілих подій у снапшоти, видалення або архівація подій, які вже не потрібні для відтворення актуального стану, а також періодична оптимізація локального сховища.

Окрім наведених недоліків, варто також зазначити, що запропонований підхід найкраще працює для широкого класу типових CRUD-сценаріїв, однак не завжди гарантує збереження складної прикладної семантики лише за рахунок базового Last-Write-Wins. У випадках, де важливим є не лише фінальний стан, а й точний намір користувача, можуть знадобитися додаткові серверні правила злиття, валідації або трансформації подій.

Напрямки подальшого розвитку бібліотеки sqlite-sync можуть включати:

- реалізацію механізмів компакції журналу подій і створення снапшотів для зменшення обсягу історичних даних;
- більш нативну інтеграцію CRDT-шару з SQLite WASM для зниження накладних витрат локальних операцій;
- підтримку складніших правил злиття на сервері, включно з доменною логікою, наближеною до Operational Transformation;
- дослідження способів підтримки unique constraints, foreign keys та інших реляційних інваріантів у distributed-середовищі;
- розвиток часткової синхронізації та вибіркової реплікації підмножин даних;

- розширення набору серверних адаптерів, зокрема для інших платформ, окрім Cloudflare;
- розширення набору клієнтських фреймворків, які підтримуються бібліотекою;
- подальше експериментальне оцінювання на більших навантаженнях і в складніших багатокористувацьких сценаріях;

Отже, запропонований підхід не є універсальним вирішенням усіх проблем синхронізації даних, однак він демонструє, що поєднання реляційної SQLite-моделі, CRDT-подій, Event Sourcing та гібридного впорядкування подій через HLC дозволяє побудувати практичну платформу для local-first вебзастосунків. Основна цінність sqlite-sync полягає в тому, що складні механізми розподіленої синхронізації інтегруються у звичну для розробника модель SQL-таблиць і CRUD-операцій, суттєво знижуючи поріг входження у створення колаборативного програмного забезпечення.

ВИСНОВКИ

В даній магістерській роботі було проведено комплексний аналіз різного роду проблем, що виникають у сучасних веб-застосунках із колаборативним редагуванням, та розроблено програмний комплекс sqlite-sync, який реалізує гібридний підхід до побудови offline-first систем.

Актуальність роботи зумовлена тим, що традиційна серверно-орієнтовна модель погано відповідає вимогам сучасних застосунків щодо швидкодії, автономності та стійкості до втрати мережевого з’єднання, тоді як існуючі підходи до синхронізації або є надто складними для практичної інтеграції, або накладають суттєві архітектурні обмеження.

В роботі було обґрунтовано доцільність комбінування різних підходів до синхронізації даних в межах єдиної архітектури, де CRDT використовується для автоматичного безконфліктного злиття змін, Event Sourcing для журналювання та реплікації подій, а OT-подібна логіка може бути застосована для складніших прикладних конфліктів на серверному рівні.

Основним результатом роботи стала розробка готової до використання бібліотеки sqlite-sync, що включає ядро синхронізації, React-інтеграцію, серверний адаптер для Cloudflare та набір застосунків-прикладів. У запропонованій системі звичайні SQL-операції автоматично перетворюються на CRDT-події через SQLite view і INSTEAD OF trigger-и, а синхронізація відбувається не шляхом прямого копіювання стану, а через реплікацію та повторне застосування журналу подій. Для впорядкування подій у розподіленому середовищі було використано Hybrid Logical Clock, що забезпечує монотонність часових міток і коректну реалізацію правила Last-Write-Wins на рівні окремих атрибутів. На клієнтській стороні було запроваджено реактивну обгортку над SQLite базою даних, яка дозволяє легко будувати динамічний користувацький інтерфейс.

Експериментальна перевірка підтвердила працездатність і практичну придатність розробленого підходу, а також виявила деякі його обмеження. Однак ці обмеження не заперечують практичної цінності розробленої системи, а радше визначають напрями її подальшого розвитку.

Отже, поставлена в магістерській роботі мета досягнута. Було не лише досліджено сучасні підходи до синхронізації даних у вебзастосунках, а й розроблено та експериментально перевірено власний гібридний метод, реалізований у вигляді бібліотеки sqlite-sync. Практичне значення роботи полягає в тому, що запропонований підхід дозволяє інтегрувати складні механізми розподіленої синхронізації у звичну для розробника реляційну модель SQLite. Таким чином, отримані результати можуть бути використані як основа для подальших досліджень і розвитку інструментів local-first програмного забезпечення.

СПИСОК ДЖЕРЕЛ

1. Yatsenko D. Linear App Built $400M Issue Tracker With Next to No Marketing: Here's How [Електронний ресурс] / D. Yatsenko // Eleken. - Updated: 05.02.2026. - URL: https://www.eleken.co/blog-posts/linear-app-case-study.
2. Kleppmann M., Wiggins A., van Hardenberg P., McGranaghan M. Local-First Software: You Own Your Data, in Spite of the Cloud / M. Kleppmann, A. Wiggins, P. van Hardenberg, M. McGranaghan // Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '19). - New York, NY, USA : ACM, 2019. - P. 154-178. - DOI: 10.1145/3359591.3359737.
3. Ellis C. A., Gibbs S. J. Concurrency Control in Groupware Systems / C. A. Ellis, S. J. Gibbs // ACM SIGMOD Record. - 1989. - Vol. 18, no. 2. - P. 399-407.
4. Wang D., Mah A., Lassen S. Google Wave Operational Transformation [Електронний ресурс] / D. Wang, A. Mah, S. Lassen. - 2010. - URL: https://web.archive.org/web/20090531063923/http://www.waveprotocol.org/whitepapers/operational-transform.
5. Letia M., Preguiça N. M., Shapiro M. Consistency without concurrency control in large, dynamic systems / M. Letia, N. M. Preguiça, M. Shapiro // ACM SIGOPS Operating Systems Review. - 2010. - Vol. 44, no. 2. - P. 29-34. - DOI: 10.1145/1773912.1773921.
6. Shapiro M., Preguiça N., Baquero C., Zawirski M. A comprehensive study of Convergent and Commutative Replicated Data Types [Електронний ресурс] / M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski. - INRIA Research Report RR-7506. - 2011. - 50 p. - URL: https://hal.inria.fr/inria-00555588.

7. Lazaroff J. An interactive intro to crdts. [Електронний ресурс] URL: https://jakelazaroff.com/words/an-interactive-intro-to-crdts/.
8. Fowler M. Event Sourcing [Електронний ресурс] / M. Fowler // Martin Fowler. - 12.12.2005. - URL: https://martinfowler.com/eaaDev/EventSourcing.html
9. LiveStore. Event Sourcing [Електронний ресурс]. - URL: https://docs.livestore.dev/evaluation/event-sourcing/
10. ShareDB. Introduction [Електронний ресурс]. - URL: https://share.github.io/sharedb/
11. Yjs. Introduction [Електронний ресурс]. - URL: https://docs.yjs.dev/
12. Electric. Postgres Sync [Електронний ресурс]. - URL: https://electric-sql.com/primitives/postgres-sync
13. Long J. CRDTs for Mortals [Video resource] / J. Long // dotJS. - 2019. - URL: https://www.youtube.com/watch?v=DEcwa68f-jY
14. SQLite Tutorial. SQLite INSTEAD OF Triggers [Електронний ресурс]. - URL: https://www.sqlitetutorial.net/sqlite-instead-of-triggers/
15. SQLite. CREATE TRIGGER [Електронний ресурс]. - URL: https://www.sqlite.org/lang_createtrigger.html
16. Kulkarni S., Demirbas M., Madeppa D., Avva B., Leone M. Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases [Електронний ресурс] / S. Kulkarni, M. Demirbas, D. Madeppa, B. Avva, M. Leone. - Technical Report 2014-04. - 2014. - URL: https://cse.buffalo.edu/tech-reports/2014-04.pdf
17. SQLite WebAssembly & JavaScript. Persistent Storage Options [Електронний ресурс]. - URL: https://sqlite.org/wasm/doc/trunk/persistence.md#vfs-opfs-sahpool
18. Cloudflare. Durable Objects docs: Overview [Електронний ресурс]. - URL: https://developers.cloudflare.com/durable-objects/
19. MDN Web Docs. Origin private file system [Електронний ресурс]. - URL:

https://developer.mozilla.org/en-US/docs/Web/API/File_System_API/Origin_private_file_system

20. MDN Web Docs. Web Locks API [Електронний ресурс]. - URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Locks_API
21. pnpm. Workspace [Електронний ресурс]. - URL: https://pnpm.io/workspaces
22. tinylibs. tinybench [Електронний ресурс]. - URL: https://github.com/tinylibs/tinybench
23. Kysely. Splitting query building and execution [Електронний ресурс]. - URL: https://kysely.dev/docs/recipes/splitting-query-building-and-execution
24. SQLite. Commit And Rollback Notification Callbacks [Електронний ресурс]. - URL: https://sqlite.org/c3ref/commit_hook.html
25. SQLite. The Bytecode() And Tables_Used() Table-Valued Functions [Електронний ресурс]. - URL: https://sqlite.org/bytecodevtab.html