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

Курсова робота

На тему:

Використання LMDB для роботи з великими обсягами даних в iOS застосунках

Виконав: студент 3-го року навчання,
Спеціальності:
121 Інженерія програмного забезпечення
Тарасенко Михайло Сергійович
Керівник: старший викладач
Франків О.О.

Київ-2025

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Освітній ступінь бакалавр

Спеціальність 121 «Інженерія Програмного Забезпечення»

Освітня програма бакалавр

ЗАТВЕРДЖУЮ

Завідувач кафедри інформатики

Гороховський С. С.

“10” жовтня 2024 року

ЗАВДАННЯ
ДЛЯ КУРСОВОЇ РОБОТИ СТУДЕНТУ

Тарасенку Михайлу

1. Тема роботи «Використання LMDB для роботи з великими обсягами даних в iOS застосунках», керівник роботи Франків Олександр Олександрович, магістр комп’ютерних наук, старший викладач

2. Строк подання студентом роботи 5 травня 2025

3. План роботи

Анотація

Вступ

Розділ 1. Аналіз проблематики та стандартних рішень для збереження великих даних в iOS

1.1 Поняття ”Великого обсягу даних”

1.2 Джерела та приклади великих обсягів даних в iOS додатках

Тема: Використання LMDB для роботи з великими обсягами даних в iOS застосунках

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на кваліфікаційну роботу.	02.11.2024	
2.	Огляд технічної літератури за темою роботи.	15.11.2024	
3.	Виконати аналіз сучасних рішень ...	25.11.2024	
4.	Розробка власного персистентного сховища	1.01.2025	
5.	Програмування власного персистентного сховища	1.02.2025	
6.	Проведення порівняльного дослідження	1.03.2025	
7.	Аналіз отриманих результатів	15.03.2025	
8.	Остаточне оформлення технічної частини	1.04.2025	
9.	Створення презентації та написання доповіді	5.05.2025	
10.	Захист курсової роботи	12.05.2025	

Виконавець курсової роботи: Тарасенко Михайло

Науковий керівник: Франків Олександр Олександрович

Зміст

Анотація

У даній роботі розглянуто можливість створення власних постійних сховищ у фреймворку Core Data з використання Sqligtning. В роботі детально описано механізми збереження даних в iOS застосунках та їх переваги та недоліки. Значну увагу приділено архітектурним рішенням та особливостям реалізації LMDB та Core Data. Детально описано особливості реалізації NSIncrementalStore. В рамках даної роботи розроблено власне постійне сховище для зберігання даних та створено додатковий проєкт для тестування готового рішення та порівняння зі стандартним NSSQLiteStore. Результати дослідження підтверджують можливість поєднання Core Data з більш продуктивним рушієм для покращення ефективності застосунку.

Вступ

У сучасному цифровому середовищі мобільні пристрої посідають ключову роль у повсякденному житті людини. Вони слугують не лише засобами комунікації, а платформою для локального зберігання інформації. Існує підклас застосунків, які акумулюють великі обсяги даних користувача, наприклад нотатки, Apple Health тощо, і не мають поширювати їх через вимоги конфіденційності, тобто великі обсяги інформації мають зберігатися на пристрої.

Для таких додатків ключовим є ефективне зберігання та швидкість отримання даних у базі даних, оскільки це напряму впливатиме на швидкість роботи застосунку, а отже на користувацький досвід. Оскільки часто цим додаткам потрібно обробляти та зберігати великі обсяги даних, то одним із рішень, котре може бути запропоновано є використання проміжних обчислень з подальшим збереженням. Проте, воно не завжди є зручним і не може використовуватись для відображення графіків у реальному часі або виведення детальної статистики. Як приклад, такого застосунку, можна навести Apple Health, шо збирає дані про активність, хвороби, сон та інші дані користувача та надає детальну статистику по ним.

Тож, постає питання у розробці оптимізованого для таких завдань бази даних.

Робота складається з трьох основних розділів.

У першому розділі розглядається поняття великих обсягів даних та джерела їх виникнення в iOS застосунках. Особлива увага акцентується на механізмах зберігання інформації, їх перевагах та недоліках. Оглядається проблема продуктивності Core Data при використанні NSSQLiteStore.

В другому розділі детально розглядається архітектура та специфіки реалізації LMDB та Core Data. Розбирається принципи інтеграції LMDB, як бекенд частини для SQLite.

У третьому розділі розглядаються особливості інтеграції при під’єднанні Sqlightning до проєкту. Детально описується реалізація власного NSIncrementalStore та варіанти його підключення до координатора персистентних сховищ. Та проведено порівняльний аналіз, в якому зіставляється стандартне рішення NSSQLiteStore та кастомного рішення, а також можливості для вдосконалення.

1.0 Аналіз проблематики та стандартних рішень для зберігання великих даних в iOS

1.1. Поняття "великого обсягу даних"

Сучасні інформаційні системи – від мобільних застосунків до складних корпоративних програм – дедалі частіше функціонують в умовах, що потребують обробки та зберігання великих обсягів даних. Слід визначити, що саме охоплює дане в межах цієї роботи? Це значна кількість інформації, що накопичується у базах даних, логах, файлах і потребує ефективного зберігання та обробки. Як приклад, можна навести мільйони записів у реляційних базах даних, журнали подій або статистика роботи додатку. Виникає це внаслідок ускладнення чи автоматизації програмних систем, збільшення кількості користувачів, розвитку і зростання кількості пристроїв Інтернету, тощо.

Поняття "великого обсягу" у цьому контексті є відносним і залежить від можливостей конкретної інфраструктури. Для невеликих систем, таких як мобільні пристрої, обсяги, що можуть сягати кількох гігабайтів даних можуть виявитися проблемою, як з точку зору продуктивності, так і місцем на сховищі, тоді як для мікросервісних застосунків з великою кількістю користувачів це є типовим.

Варто зазначити, що великі обсяги даних у нашому контексті – це не те саме, що Big Data. Різниця між цими поняттями доволі чітка: якщо перший – структуровані дані, що можуть зберігатися та оброблятися звичайними методами, то друге – це ціла категорія задач і технологій, котрі пов’язані з високошвидкісною та ефективною обробкою різноманітних даних, обсяги яких перевищують можливості традиційних рішень.

Отже, великі обсяги даних – це один з викликів розробки програмного забезпечення, що вимагає ефективного використання ресурсів та оптимізацію додатку.

1.2. Джерела та приклади великих обсягів даних в iOS додатках

Сучасні мобільні застосунки характеризуються обробкою і зберіганням великих обсягів даних. Незалежно від категорії застосунку – система моніторингу здоров'я, соціальна мережа, нотатки, синхронізація та швидкість обробки даних є критичними. У контексті мобільної розробки великі обсяги це сотні мегабайт або гігабайти, що вже є суттєвим для мобільного пристрою через обмеженість його ресурсів.

Серед джерел виникнення значних обсягів даних в iOS застосунках варто виокремити медіа та файли, котрі користувач створив сам або імпортував з інших джерел. До цієї категорії відносяться застосунки з функціоналом для створення або редагування фото та відео, що може призвести до утворення чи копіювання вже існуючих файлів. Як приклад можна навести редактори фото/відео (Lightroom Photo & Video Editor, Snapsed, VSCO: Photo & Video Editor), соціальні мережі (Instagram, Tik Tok), застосунки для нотаток (GoodNotes, Evernote), месенджери (Telegram, WhatsApp, Viber). Крім того, застосунки для роботи з аудіо, теж слід відносити до цієї категорії. Зокрема, такі додатки для створення музики (GarageBand) чи диктофон. Також до цієї категорії слід відносити професійні рішення для створення 3d моделей (3D modeling: Design my model, uMake: 3D CAD Modeling, Design), креслень (AutoCAD), презентацій (Keynote, Microsoft PowerPoint) та документів (Pages, Microsoft Word).

Є велика кількість застосунків, де для зручності користувачів існує функціонал збереження даних на локальний пристрій, для доступу без Інтернету. До

них можна віднести: стрімінгові сервіси (YouTube, Netflix), програми для прослуховування музики (Apple Music, Spotify), навігаційні застосунки (Google Maps; MAPS.Me: Offline Maps, GpsNav), додатки для читання книг (Kindle, Apple Books). Також, багато застосунків використовують кешовані дані для пришвидшення роботи, та підвищення продуктивності. Зокрема соціальні мережі, котрі можуть кешувати фото та відео, сторінки профілів користувачів або ж зберігати історії спілкування (Instagram, Telegram). В свою чергу, браузери можуть кешувати сторінки, картинки, відповіді від сервера та багато чого іншого.

Часто додатки генерують та зберігають інформацію під час своєї роботи, це може бути, як історія взаємодії користувача, так і дані, отримані з інших пристроїв, таких як смарт-годинники, фітнес-браслети тощо. Гарним прикладом є Apple Health, який збирає, аналізує та зберігає велику кількість персональних даних про фізичну активність, медичні показники, сон та інші.

Таким чином, джерелами великих обсягів даних в iOS застосунках можуть бути мультимедійні матеріали, великі файли та проєкти, кеші, інформація генерована сенсорами та іншими пристроями. В умовах ресурсів мобільного середовища це створює виклики щодо продуктивності, оптимізації зберігання, опрацювання та контролю за даними, що має враховуватись під час розробки.

1.3. Огляд стандартних механізмів зберігання даних в iOS та їх обмеження

В iOS існують різні вбудовані механізми для локального зберігання даних додатків. Вибір конкретного механізму визначається типом даних, їх обсягом, структурою, а також вимогами до безпеки та продуктивності.

1.3.1 Файлова система

З міркувань безпеки у кожної програми в iOS є своє ізольоване середовище (sandbox), де і зберігаються файли. Під час встановлення нової програми інсталятор створює каталоги-контейнери для програми всередині sandbox середовища, кожен ж яких має певну роль. Bundle Container містить файли програми, і варто зазначити, що він є незмінним. В свою чергу, Data Container зберігає дані програми та користувача. [1]

Розглянемо більш детально структуру Data Container, який складається з трьох підкаталогів: Documents, Library та Temp (рис. 1).

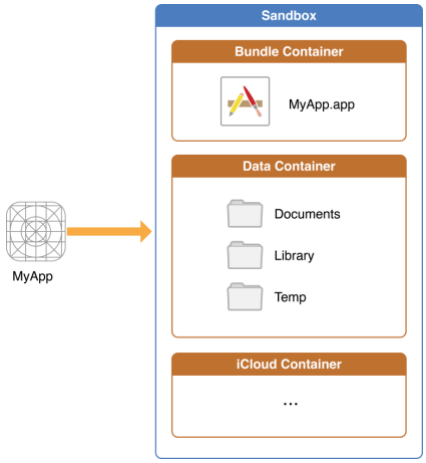

Рис. 1 Приклад ізольованого середовища iOS застосунку [1]

Documents – призначений для зберігання даних, створених користувачем, найкраще він підходить для великих файлів, таких як текстові документи, медіа файли. Вміст цього каталогу резервно копіюється в iTunes та iCloud. [1]

Library – це каталог верхнього рівня, який має стандартні підкаталоги в яких і розміщуються дані, з якими користувач не має взаємодіяти чи бачити. Прикладами стандартних каталогів є – Application Support та Caches. Також, варто зазначити, що

можна створювати і власні підкаталоги. Вміст каталогу Library (за винятком підкаталогу Caches) резервно копіюється в iTunes та iCloud. [1]

Temp – призначений для зберігання тимчасових файлів, які потрібні лише на короткий час роботи програми. Слід зазначити, що додаток має сам керувати видаленням цих файлів, але система теж може очистити цю директорію, коли програма не працює. Дані не копіюються в iTunes та iCloud. [1]

Перевагою використання файлової системи для зберігання даних є прямий контроль над файлами та їх вмістом, що може бути зручним для створення власної системи, так як розробник сам обиратиме структуру, формат та кодування для зберігання інформації. Особливо це зручно робити для великих за обсягом файлів, наприклад, для зображень, відео, аудіо, текстових документів тощо. Ще однією перевагою є легкий та зрозумілий API для роботи з файловою системою – FileManager.

Недоліки такого підходу стають суттєвими при зростанні складності та обсягу даних. Відсутність готових механізмів для роботи зі структурованими даними, таких як транзакції чи система запитів можуть стати значною перешкодою для подальшої розробки, оскільки розробник повинен реалізовувати всі ці функції самостійно, на відміну від баз даних, в яких це вже реалізовано.

Отже, файлова система може слугувати ефективним засобом для зберігання великих медіа файлів чи документів користувача, але не підходить для зберігання великих обсягів структурованих даних, що потребують частого доступу та системи складних запитів.

1.3.2. SQLite.

SQLite – Це вбудована, легка, файлова база даних, що зберігається як один файл. Вона, надзвичайно популярна і широко використовується у мобільній розробці, зокрема в iOS.

Серед переваг використання SQLite можна зазначити: реляційну структуру що дозволяє організовувати дані у вигляді взаємопов’язаних таблиць; підтримку SQL, забезпечуючи зручну роботу з складними запитами; транзакційність (ACID), що гарантує надійність та цілісність даних у базі даних; кросплатформність та простоту використання.

Попри перелічені переваги, це рішення має і певні обмеження. Серед них є необхідність у ручному перетворенні даних, що вимагає написання SQL-запитів у вигляді рядків та перетворення результатів в об'єкти мови програмування (Swift/Objective-C) і навпаки, що може призвести до великої кількості шаблонного коду, складності підтримки та потенційних помилок. Крім того, важливим недоліком є відсутність автоматизованого механізму для управління схемами та міграціями бази даних. Розробник має контролювати створення та зміну структури таблиць у всіх потрібних файлах, що може стати проблемою у великому проєкті через те, що таблицю можна змінити в одному з компонентів без відповідного оновлення в інших частинах, виникає ризик несинхронізованих даних та помилки під час роботи програми. Також, із зростанням обсягів даних можуть спостерігатися зниження продуктивності, через недостатню оптимізацію.

Таким чином SQLite є потужним і надійним інструментом для невеликих застосунків або додатків, що потребують повного контролю над базою даних. Однак він вимагає більше часу та зусиль на розробку порівняно з більш високорівневими фреймвокрами, такими як CoreData чи Realm.

1.3.3. Realm.

Realm – це реактивна об’єктно-орієнтована база даних з власним рушієм, створена спеціально для мобільних пристроїв.

Серед переваг Realm можна виокремити такі ключові аспекти: простота використання, висока продуктивність, об’єктно орієнтований підхід, реактивність, кросплатформність та масштабованість.

Попри свою зручність і простоту використання, Realm має низку недоліків, які можуть стати критичними у деяких проєктах. По-перше, база даних не підтримує SQL, що може стати обмеженням, при потребі у складних запитах. По-друге, Realm є стороннім (third-party) фреймворком, що означає залежність від зовнішньої компанії та можливі ризики, пов’язані з невизначеністю майбутнього цієї технології. По-третє, ще одним суттєвим мінусом є великий розмір бібліотеки, що негативно впливає на розмір кінцевого застосунку та значно збільшує час компіляції.

Отже, Realm є сучасним та привабливим рішенням для мобільних пристроїв, що пропонує зручний API та високу продуктивність. Однак, відсутність SQL, великий розмір та залежність від стороннього фреймворку, можуть створити суттєві обмеження.

1.3.4. Core Data.

Core Data – це потужний та гнучкий фреймворк розроблений компанією Apple, який фактично є графом об’єктів. Він управляє об’єктами в пам’яті і може зберігати їх до вибраного сховища. Для роботи із SQL сховищем фреймворк надає NSSQLiteStore.

Core Data має низку переваг, що вирізняють її з-поміж конкурентів та робить його надзвичайно зручною для використання у роботі з локальними даними в середовищі Apple. Робота з даними відбувається через контекст керованих об’єктів, що відповідає за відстеження змін, додавання та видалення нових об’єктів. Core Data розроблений з урахуванням архітектурний властивостей та принципів екосистеми Apple, що дозволяє йому тісно інтегруватися з такими технологіями як SwiftUI та мати розширений функціонал для роботи у середовищі розробки Xcode. Однією з головних механік фреймворку є ліниве завантаження даних (faulting), що зменшує навантаження на пам’ять пристрою, оскільки потрібні дані будуть завантажуватись тільки за потреби. Крім того, Core Data підтримує механізм міграцій, що забезпечує безпечне оновлення моделі бази даних для .

На противагу цьому, фреймворк має складний для опанування API та велику кількість нюансів, що може стати перешкодою для програмістів-початківців.

Core Data є популярним рішенням для роботи з БД в iOS застосунках, але при роботі з великими обсягами даних розробники можуть зіштовхнутися з необхідністю оптимізації програм або потребою у використанні власних рішень.

1.4. Проблеми продуктивності Core Data + NSSQLiteStore на великих наборах даних

Після розгляду можливих варіантів методів збереження даних в iOS застосунках, можна зробити висновок, що фреймворк Core Data є одним з найефективніших та найпопулярніших в екосистемі Apple.

Хоча Core Data має численні переваги, він має низку проблем при роботі з великими обсягами даних, більшість з яких пов’язані з тим, що в якості основи для зберігання даних найчастіше використовується SQLite. Найсуттєвішим із недоліків є падіння продуктивності при збільшенні обсягів даних, оскільки SQLite має

однопоточність при записі, то при одночасній спробі модифікування даних, зміни будуть вставати в чергу, і тоді, фактично, продуктивність додатку буде залежати від того, з якою швидкістю база даних зможе обробляти ці запити. Схожі проблеми можуть з’явитися і при зчитуванні даних, це спричинено значним збільшенням кількості fetch запитів, через механізм лінивого завантаження даних. Це зумовлено тим, що спочатку Core Data буде забирати тільки ідентифікатори об’єктів з бази даних, і тільки при необхідності використання даних буде робити запит на отримання детальної інформації.

Таким чином, більшість недоліків Core Data можна звести до недоліків SQLite – системи, спроектованої для мобільних пристроїв із відносно невеликим обсягом даних.

2. Поглиблений аналіз технологій LMDB та Core Data

2.1 База даних LMDB: Огляд технології

LMDB (Lightning Memory-Mapped Database) — це вбудована транзакційна база даних типу "ключ-значення", що вирізняється надзвичайною швидкістю, особливо при читанні даних, високою надійністю та компактністю (розмір об’єктного коду займає всього 32 кб, що надзвичайно мало). Розроблена компанією Symas для використання у проєкті OpenLDAP як заміна для Berkeley DB (надалі BDB). [2]

Існуючий на той час BDB не задовольняв розробників, через низку своїх недоліків. По-перше, складність правильного налаштування для отримання оптимальної продуктивності та надійності. По-друге, проблеми продуктивності та значні труднощі розробки. Причиною цього є декілька кешів, котрі витрачають пам’ять через можливість дублювання даних, і кожен з яких потребує ретельного налаштування та узгодження. Також через проблему узгодження компонентів

з’являються блокування, що уповільнюють роботу. По-третє, великі витрати на адміністрування, так ,наприклад, BDB використовує журнали попереднього запису для управління транзакціями. Файли журналів постійно зростають, а операції їх видалення займають багато часу і є небезпечними. [2]

Тож, розробники прийшли до рішення – розробити власну базу даних, котра б мала такі властивості: максимальні швидкості читання, надійна та стійка до збоїв, легка у використанні, ефективно використовувала ресурси системи, забезпечувала ACID.

Отже, LMDB розроблялася як замінник BDB для використання у проєкті OpenLDAP, та мала забезпечити низку властивостей, необхідних необхідних для проєкту OpenLDAP.

2.2. Архітектура та принципи роботи LMDB

Як було зазначено у попередньому розділі, LMDB дозволяє досягти виняткової продуктивності та надійності, що забезпечується поєднанням ключових технологій та підходів до роботи з файлами та даними.

Файли, відображені у пам'ять (memory-mapped-files [3]): це фундаментальна відмінність LMDB від багатьох інших баз даних. Це механізм, за допомогою якого файл (або його частина) відображається у віртуальну пам’ять процесу. Тож, програма працює з вмістом файлу так, ніби це оперативна пам’ять, без необхідності використовувати таку операцію як read().

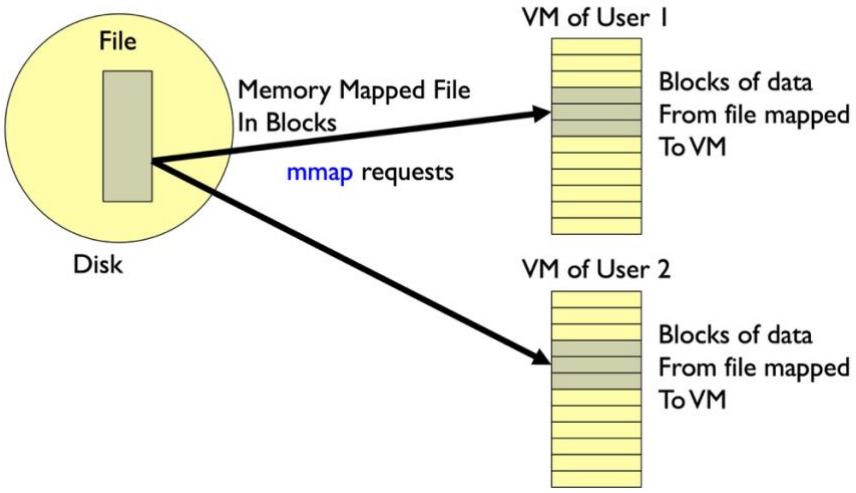

Рис. 2 Схема відображення файлу у пам'ять [3]

Використання цього механізму має багато переваг, серед яких: прямий доступ до даних; використання кешу ОС, що спрощує розробку, оскільки операційна система буде сама вирішувати, які дані тримати в оперативній пам'яті та як ними управляти, що усуває необхідність у розробці власного рішення. Ще однією перевагою є нульове копіювання (Zero-Copy) – через те, що дані вже знаходяться в кеші OS, то немає потреби у їхньому копіюванні, що значно зменшує затримки.

Паралельне керування багатьма версіями (MVCC [4]): Основна ідея цього методу полягає у тому, що оновлення даних ніколи не перезаписують існуючі дані; замість цього створюється копія сторінки, котра і змінюється і таким чином створює нову версію БД. Такий механізм забезпечує ізольованість читачів та записувачів. Кожна транзакція читання працює зі кореневою сторінкою, яка була на момент її старту. Якщо ж записувач робить якісь зміни, то це не буде впливати на вже існуючі транзакції читання, оскільки сторінки з якими вони працюють ніяк не змінилися. Через цю ізоляцію доступ для читання не потребує блокуваннь, вони завжди мають консистентний набір даних. Та завдяки збереженню попередніх версій, база даних має підвищену відмовостійкість.

В основі механізму зберігання лежить концепція append-only Btree, котра працює за принципом, що будь-яка зміна структури дерева (наприклад, додавання або оновлення) не змінює існуючі сторінки, а створює нові, після завершення транзакції змінюється метасторінка (з посиланням на новий корінь дерева). Це дозволяє зберігати історію змін, уникати колізій між транзакціями та забезпечує стійкість до збоїв. Однак у цього підходу є низка недоліків: файл бази даних постійно зростає, що призводить до необхідності у періодичному збиранні сміття.

LMDB ж реалізувало це по-іншому: замість нескінченного зростання, використовуються дві фіксовані метасторінки, котрі по черзі зберігають актуальний стан бази даних – посилання на корені двох B+ дерев – основного дерева даних та дерева вільних сторінок (free list). [2]

Тож такий механізм забезпечує атомарність, відмовостійкість та високу швидкодію. Також це прибирає проблему постійного зростання файлу, адже минулі сторінки, котрі вже не використовуються можуть бути перевикористані для записування, замість створення нових.

2.3. SQLite з lmdb: Огляд інтегрованого рішення

Як було зазначено у першому розділі SQLite є надзвичайно популярною та ефективною при роботі з малими обсягами даних, але має низку проблем при великих обсягах. SQLite працює на власному B-tree рушії, і існують експериментальні проєкти, що досліджують заміну цього стандартного рушія на інші key-value сховища. LMDB є одним з таких рішень, завдяки швидкості читання, надійності та ефективності використання пам'яті. Тож, ідея інтеграції полягає в тому, щоб поєднати SQLite з перевагами архітектури LMDB на рівні зберігання даних.

Як приклад таких проєктів можна навести: SQLightning [6] та LumoSQL [7]

2.3.1. Принцип інтеграції та потенційні переваги

SQLite має багатошарову архітектуру. Верхні шари відповідають за роботу з SQL-запитами (Tokenizer, Parser, Code Generator), а нижній шар (B-tree) відповідає безпосередньо за зберігання даних на диску (Рис. 3) [5]. Принцип інтеграції SQLite з LMDB полягає в заміні стандартного B-tree на власний модуль що використовує LMDB.

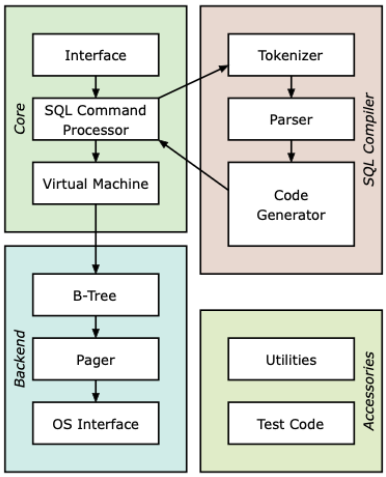

Рис. 3 Архітектура SQLite [5]

Поєднання SQLite та LMDB дозволить зберегти логіку роботи бази даних та залишить SQL, що є дуже зручним для користувачів так, як їхня взаємодія бібліотекою ніяк не зміниться. А зміна рівня зберігання даних на LMDB дозволить використати всі переваги перелічені у минулому розділі.

Тож, інтеграція LMDB як бекенду для SQLite може принести такі переваги: підвищена продуктивність читання, вища надійність та стійкість до збоїв, більш ефективне використання пам’яті.

Нижче наведено результати замірів що порівнюють продуктивність стандартного SQLite(3.7.17) та версії з бекендом LMDB (Рис. 4)

	SQLite	SQLightning
Operation times in microseconds, lower is better		
Sync Seq Write	8175.371	6171.233
Sync Rand Write	8308.706	6231.249
Seq Write	25.587	31.778
Batch Seq Write	7.402	7.087
Rand Write	33.235	32.902
Batch Rand Write	18.847	13.754
Rand Read	22.645	7.685
Seq Read	7.557	1.551
Rev Seq Read	7.456	1.531

Рис. 4 Порівняння продуктивності SQLightning та SQLite [6]

2.4. Детальний огляд архітектури Core Data

Core Data – призначений для керування об'єктами у пам'яті (граф об'єктів) та зберіганням їх у персистентне сховище. Core Data впроваджує чотири різні типи сховищ – SQLite, Binary, XML та In-Memory. У цій роботі, сконцентруємось на імплементації саме з SQLite. Базовий стек фреймворку складається з чотирьох основних частин: керовані об'єкти (NSManagedObject), контекст керованих об'єктів (NSManagedObjectContext), координатор постійного сховища (NSPersistentStoreCoordinator) та постійне сховище (Persistent Store). (Рис. 5) [8]

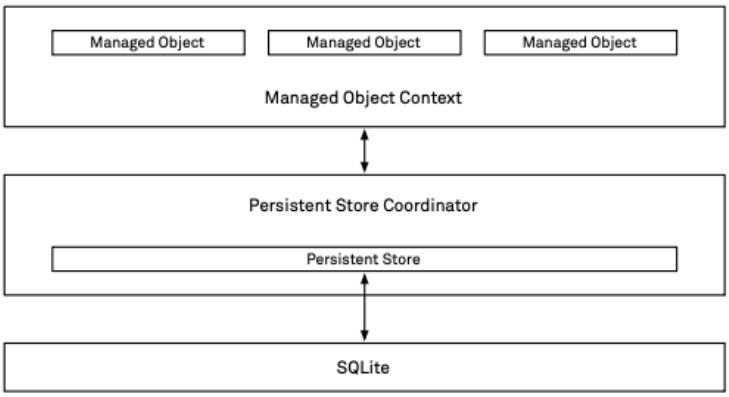

Рис. 5 Архітектура Core Data [8]

Managed Objects – це класи, що представляють дані, які зберігаються в постійному сховищі. Для створення керованих об'єктів спершу потрібно створити

файл моделі даних .xcdatamodeld. В ньому створюються та налаштовується (додаються потрібні атрибути, зв'язки з іншими сутностями …) сутності. Далі потрібно створити ці самі керовані об'єкти, Xcode надає три варіанти на вибір, як це можна зробити: Manual/None, Class Defifnition, Category/Extension. Слід зазначити, що для реалізації керованого об'єкту вам знадобиться два файли: класу та властивостей, який буде доповнювати клас створений у першому файлі. В першому варіанті Xcode не генерує жодних класів для обраної сутності, тож розробник має реалізувати це самостійно. Це потрібно, якщо керований об'єкт необхідно більш точно налаштувати, наприклад змінити модифікатори доступу, додати нові методи. Вибираючи другий варіант, Xcode згенерує все необхідне автоматично, але ви не побачите цих файлів у вашому вихідному коді, так як це відбувається під час збирання проєкту і вони знаходяться у build директорії. Обираючи цей варіант варто розуміти, що редагування згенерованих реалізацій неможливе. Category/Extension – згенерує тільки файл властивостей, що дасть вам змогу додавати додаткові можливості, так як клас сутності реалізовуєте ви самі. [9]

Managed Object Context – це клас для управління керованими об'єктами. Ви використовуєте його для отримання, створення та відстеження змін у Managed Objects. Контекст керованих об'єктів пов'язаний зі сховищем, зазвичай це Persistent Store Coordinator. Тож, коли відбувається запит на отримання об'єктів контекст просить сховище надати йому об'єкти, що відповідають запиту. При створенні та оновлені, зміни одразу не зберігаються у постійне сховище, для цього потрібно явно зберегти контекст. Також варто зазначити, що в межах одного контексту (їх може бути і декілька) може існувати тільки один керований об'єкт для представлення відповідного запису з бази даних. [10]

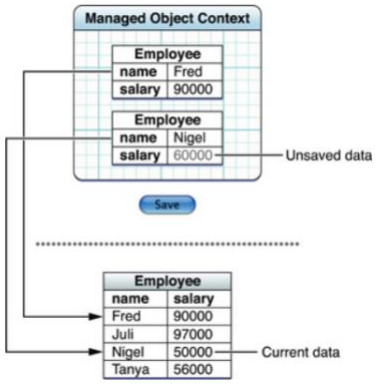

Рис. 6 Схематичне зображення роботи контексту керованих об'єктів [10]

Persistent Store Coordinator: Це центральна частина Core Data. Він керує постійними сховищами та надає фасад уніфікованого сховища для контекстів керованих об'єктів. [11]

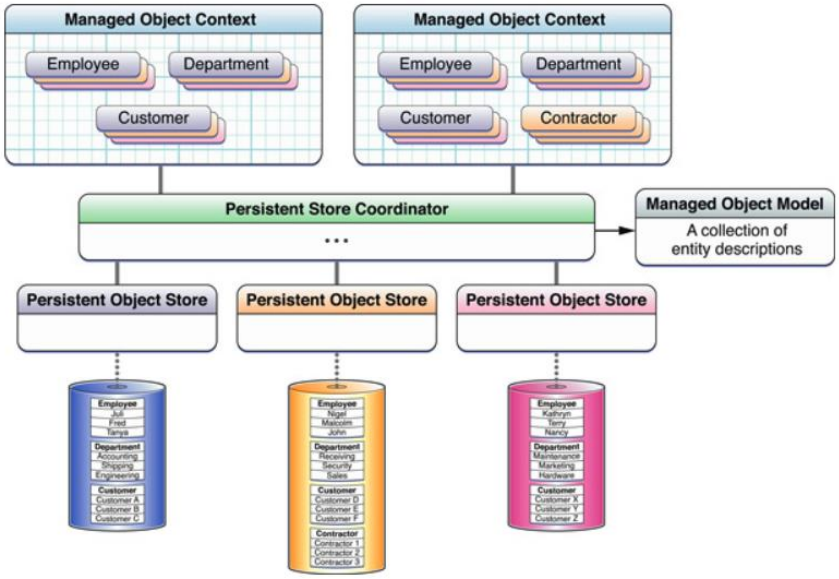

Рис. 7 Схема роботи координатору постійних сховищ [11]

Persistent Store – це репозиторій, у якому зберігаються дані. Core Data має три типи постійних сховищ: бінарне, XML та SQLite. Також можна реалізовувати власні сховища. [12]

2.5. Механізм NSIncrementalStore: Інтерфейс для власного сховища

NSIncrementalStore – це клас який репрезентує постійне сховище в Core Data, яке дозволяє отримувати дані зі сховища порціями, а не все одразу. Таким же чином,

і відбувається запис та оновлення даних, тобто змінюються лише необхідні дані, а не сховище цілком. Це дає змогу тримати в оперативній пам’яті лише ті об’єкти, котрі потрібні для роботи, що ідеально підходить для роботи з великими обсягами даних. Як приклад, можна навести базову реалізація на SQLite.

NSIncrementalStore має виконувати запити для отримання чи збереження даних до сховища, перевіряти та завантажувати метадані, які використовує координатор постійного сховища. Для цього він має методи, зображені на Лістингу 1.

```
import CoreData

class CustomIncrementalStore: NSIncrementalStore {

    override func loadMetadata() throws {
        code
    }

    override func execute(_ request: NSPersistentStoreRequest, with context: NSManagedObjectContext?)
        throws -> Any {
        code
    }

    override func newValuesForObject(with objectID: NSManagedObjectID, with context:
        NSManagedObjectContext) throws -> NSIncrementalStoreNode {
        code
    }

    override func obtainPermanentIDs(for array: [NSManagedObject]) throws -> [NSManagedObjectID] {
        code
    }
}

```

Лістинг 1 Базове Api NSIncrementalStore

Розглянемо більш детально кожний метод [13].

loadMetadata – відповідає за ініціалізацію та перевірку метаданих, перевірку URL адреси сховища, завантаження та перевірку хешів версій моделей для підтримки міграцій.

execute – викликається при запиті на отримання або збереження даних. На вхід методу подається request та context (в якому відбулися зміни). Request – описує що саме треба зробити та з якими об’єктами. Наприклад, NSSaveChangesRequest, що інкапсулює колекції змін, котрі мають бути виконані. Ще один тип запитів – це NSFetchRequest, що репрезентує вибірку необхідних об’єктів.

newValuesForObject – використовується, коли керованому об’єкту стають потрібні значення його властивостей, координатор постійного сховища викличе цей метод для їх отримання. Ви отримаєте objectId, що є унікальним ідентифікатором об’єкта.

obtainPermanentIDs –викликається при збережені нового керованого об’єкту. При створенні об’єкту присвоюється тимчасовий ідентифікатор без залучення постійного сховища. І тільки при збережені контексту буде викликано цей метод для надання id, після чого координатор викличе метод execute для обробки кожної транзакції.

3. Створення власного NSIncrementalStore

3.1 Загальний опис

У попередніх розділах ми розглядали механізми зберігання даних в iOS, зокрема, файлова система, SQLite, Realm, Core Data. Фреймворк Core Data є популярним та зручним для роботи зі структурованими даними та SQL. Попри це, він має ряд недоліків при роботі з великими обсягами даних. Проаналізувавши архітектуру та специфіки реалізації, ми дійшли висновку, що проблеми продуктивності виникають зокрема через використання SQLite. Логічним висновком буде заміна на більш швидкий рушій або використання оптимізованої версії цієї бази даних.

3.2 Під'єднання Sqlightning до проєкту

Однією з перших проблем, з якою довелося зіштовхнутися в процесі розробки власного PersistentStore є під'єднання бібліотеки бази даних. Підключення сторонніх бібліотек – це завдання, яке може здаватися легким на перший погляд, але в реальності є справжнім викликом для розробника. Під час впровадження бібліотеки слід очікувати низки технічних, концептуальних та інфраструктурних проблем, які потребують розуміння механізмів збірки та інтерфейсів взаємодії між мовами.

Однією з найпоширеніших проблем є налаштування шляху до заголовкових (.h) та бібліотечних (.a) файлів. Середовище розробки Xcode може не бачити ці файли, якщо вони знаходяться не в стандартних директоріях, і тоді проєкт просто не буде збиратися. Якщо ж знайти необхідні файли вдалося, то наступним випробуванням буде створення правильного прошарку для Swift, так як він не може викликати методи з C напряму, найчастішими рішеннями є bridging header [14] або module map [15]. На даному етапі, важко зрозуміти де саме виникають помилки, так як бібліотека може імпортуватися, а проєкт збиратися. Однак це все одно не гарантує можливості імпортувати потрібні вам класи.

Зіткнення з подібними проблемами стало одним із ключових викликів на початковому етапі реалізації проєкту, що суттєво вплинуло на темпи його розробки. Незважаючи на значну кількість спроб вирішення цих труднощів за допомогою різних підходів та технічних рішень причина помилки залишилася незрозумілою. Що вказує на необхідність в більш детальному огляді методів під'єднання сторонніх бібліотек.

Розглянемо детальніше процес підключення Sqlightning, який було реалізовано у цьому проєкті. По-перше нам необхідно зібрати бібліотеку, щоб з'явилися заголовкові файли, котрі зможе використовувати Xcode. Після чого, потрібно додати повний шлях до них у Header Search Paths (як зазначено на Рис. 8), та визначачити Other Linker Flags – -lsqlite3 (Рис. 9). Ці властивості знаходяться у налаштуваннях збірки проєкту. Після цих дій я зміг імпортувати sqlightning та використовувати її методи.

Рис. 8 Приклад повного шляху до дерикторії бібліотеки, де знаходяться заголовний файл

Рис. 9 Приклад налаштування Other Linker Flags

3.3 Написання власного NSIncrementalStore з використанням SQLightning

В минулому розділі, розглядалося, як можна під'єднати бібліотеку бази даних до проєкту, отже наступним кроком є реалізація власного Incremental Store. Для цього потрібно зрозуміти, які задачі та як саме він має виконувати. Знаходження цієї інформації та прикладів реалізації може бути важким завданням, так як більшість

прикладів чи туторіалів реалізовують найпростішу логіку, котра насправді відрізняється від правильної реалізації і може вводити в оману.

На мою думку, найкращим джерелом для розуміння роботи та допомоги у реалізації – є документація від компанії Apple ”Implementation Strategy” [14], тому у наступних етапах розробки я буду використовувати саме цей ресурс.

Першим кроком є створення клаус який називається CustomStore та наслідує NSIncrementalStore.

Перший метод, який потрібно реалізувати, це – loadMetadata(). Для цього необхідно імплементувати: перевірку коректності шляху до бази даних та встановлення зв’язку з нею; перевірити відповідність хешів моделей та табличок, і в разі їх відмінності виконати міграції; проініціалізувати метадані, в яких зазначимо тип сховища, унікальній UUID-ключ та поточне значення хешів моделей.

Для реалізації першого пункту, розроблений внутрішній клас Database, що приймає шлях до БД та встановлюватиме з’єднання за допомогою метода sqlite3_open_v2. Таким чином, було реалізовано обгортку, яка забезпечує доступ до вказівника на базу даних.

Для реалізації другого пункту, було реалізовано допоміжні методи:

loadModelHashesFromDatabase – тут відбувається запит до БД на отримання поточної версії моделей. Якщо ж такого запису або таблиці ще не існує, то потрібно правильно обробити результати та повернути nil, що буде означати, що база даних пуста.

updateDatabaseSchema – метод перевіряє відповідність хешів, отриманих з бази даних, та тих, що ми отримали у моделі об’єктів. При не співпадінні цих значень для конкретної сутності необхідно, або оновити табличку, якщо дані є, але не співпадають, або видалити, якщо записів про неї вже немає у моделі об’єктів.

Також потрібно створити таблички для новостворених сутностей, яких, ще немає у базі даних.

Наступним етапом буде реалізація функціоналу fetch запитів. Для цього реалізовано перевизначення методу execute(_ request:with:). Перший параметр має тип NSPersistentStoreRequest та інкапсулює різні типи запитів. У поточному етапі реалізації, необхідно оброблювати тільки NSFetchRequest (Лістинг 2).

```
// MARK: - execute method
override func execute(_ request: NSPersistentStoreRequest, with context: NSManagedObjectContext?) throws -> Any {
    switch request {
    case let fetchRequest as NSFetchRequest<NSFetchRequestResult>:
        return try executeFetchRequest(entity: fetchRequest.entity!, in: context!)
    default:
        throw NSError(
            domain: "CustomStore", code: 3,
            userInfo: [
                NSLocalizedDescriptionKey: "Unsupported request type"
            ])
    }
}
```

Лістинг 2 Приклад реалізації метода execute для fetch запиту

executeFetchRequest – приймає entity: NsEntityDescription (з якого отримується назва сутності) та context, в котрому потрібно ці сутності ініціалізувати. Ця функція має повернути список NSManagedObject, котрі будуть так звані fault [14]. У спрощеному вигляді – це об’єкти у яких визначені тільки ідентифікатори. Це зроблено для поліпшення використання пам’яті, відкладаючи матерелізацію даних до того моменту, коли вони стануть потрібні користувачеві. Тож, необхідно зробити запит до бази даних на отримання id’s певної сутності. Для створення керованого об’єкту у певному контексті потрібно спочатку отримати для нього NSManagedObjectID, після чого можна створити сам об’єкт.

Виникає питання щодо отримання всіх інших властивостей об’єктів. Для цього існує метод newValuesForObject(with:with:), координатор постійного сховища викличе цей метод коли користувачеві знадобляться ці дані, і попросить повернути NSIncrementalStoreNode, в котру передаються: ідентифікатор об’єкта; значення його властивостей у вигляді словнику, де ключ – це назва властивості і відповідне

значення; а також версію об’єкту, це необхідно для виявлення та вирішення конфліктів. (Лістинг 3)

```
//MARK: - New values for object
override func newValuesForObject(with objectID: NSManagedObjectID, with context: NSManagedObjectContext) throws ->
    NSIncrementalStoreNode {
    let primaryKey = referenceObject(for: objectID) as! Int

    let (values, version) = try fetchAttributesFromDatabase(for: objectID.entity, primaryKey: primaryKey)

    let node = NSIncrementalStoreNode(objectID: objectID, withValues: values, version: version)

    return node
}
```

Лістинг 3 Приклад реалізації методу newValuesForObject

На даному етапі вже реалізовано запити на отримання даних з власного IncrementalStore, для цього необхідно до координатора постійного сховища додати CustomStore, детальніше це буде розглядатися у наступному розділі. Оскільки ще не імплементований функціонал для збереження даних, то потрібно вручну заповнити дані у базі даних. Для цього екобхідно запустити проєкт, щоб відбулися міграції і з’явилися відповідні таблички. Після їх заповнення виконати fetch request, і переконаємося, що все працює. (Лістинг 4)

```
models = try context.fetch(SampleEntity.fetchRequest())
```

Лістинг 4 Приклад запиту для отримання даних

Наступним кроком є реалізації обробки запитів на збереження даних. Для цього необхідно розширити оброблюванні типи запитів у методі execute додавши NSSaveChangesRequest (Лістинг 5).

```
case let saveRequest as NSSaveChangesRequest:
    return try executeSaveRequest(saveRequest)
```

Лістинг 5 Обробка запитів на збереження даних

Як наведено у лістингу, реалізовано додатковий метод executeSaveRequest(_: NSSaveChangesRequest). Першим кроком є відкриття транзакції до бази даних, для підвищення швидкості опрацювання запитів (перша реалізація, була написана без

використання транзакцій, і була значно повільніша). Після цього зчитуються дані полів об’єкту запиту (4 різних сети об’єктів), котрі репрезентують: об’єкти для створення (.insertedObjects), оновлення (updatedObjects), видалення (.deletedObjects) та для оптимістичного блокування (.lockObjects). Оптимістичне блокування – це механізм для виявлення конфліктів та їх вирішення, це відбувається коли різні частини програми одночасно намагаються змінити одні й ті самі дані. Реалізація збереження, оновлення, видалення та блокування є доволі тривіальними, окрім запиту на створення, який буде більш детально розглянутий у наступному абзаці. Слід зазначити лише те, що в цих реалізаціях необхідно працювати з метаданими керованих об’єктів для правильного мапінгу їх у SQL запити. Ще однією важливою деталлю є оновлення версії запису при редагуванні та блокуванні сутності, та встановлення 1 – для нових об’єктів.

Створення об’єктів відрізняється за логікою роботи у CoreData від оновлення. При створенні керованого об’єкта в оперативній пам’яті, йому надається тимчасовий ідентифікатор для однозначного визначення, без використання персистентного сховища. Після ж збереження контексту Core Data спочатку попросить надати постійні ідентифікатори, викликавши метод obtainPermanentIDs(for:), і лише після цього викличе метод execute для збереження всіх даних.

На початковому етапі реалізації виникли труднощі зумовлені відсутністю чіткого розуміння – як можна отримати id, не додаючи записи до бази даних. Ще однією проблемою стало те, що метож отримує масив [NSManagedObject], тобто в ньому можуть бути сутності різного типу, і потрібно знати який наступний вільний ідентифікатор для конкретної сутності. Оптимальним вирішенням, котре було реалізовано, стало отримання найбільшого ідентифікатора для певного типу сутності, та запису цього значення до словника. Після цього, все що залишається

зробити – це інкрементально збільшити значення збережуваного ідентифікатора. Таким чином, підтримується унікальність ідентифікаторів відповідних табличок.

3.4 Додавання власного персистентного сховища до координатора

У цьому розділі описані методи під'єднання CustomStore до координатора персистентних сховищ.

Розглянемо два варіанти:

З використанням UIKit та обраним середовищем зберігання Core Data, при створенні проєкту. Спершу необхідно зареєструвати імплементацію NSIncrementalStore у NSPersistentStoreCoordinator. Наступним кроком є визначення зберігання файлу бази даних (детальніше про розташування файлів в iOS застосунку було написано у розділі 1.3.1). Останнім кроком є визначення правильного опису та завантаження описаного сховища. Відповідна логіка реалізовується у класі AppDelegate (Лістинг 6).

```
76      lazy var persistentContainer: NSPersistentContainer = {
77          let container = NSPersistentContainer(name: "CoreDataExample")
78          _ = CustomStore.registerStoreType
79          let storeURL = FileManager.default.urls(
80              for: .documentDirectory,
81              in: .userDomainMask
82          ).first!.appendingPathComponent("Store.sqlite")
83
84          let description = NSPersistentStoreDescription(url: storeURL)
85          description.type = CustomStore.storeType()
86          container.persistentStoreDescriptions = [description]
87          container.loadPersistentStores { _, error in
88              if let error = error {
89                  fatalError("Failed to load store: \(error)")
90              }
91          }
92          return container
93      }()
```

Лістинг 6 Приклад налаштування координатору персистентних сховищ

Другим варіантом є додавання фреймворку вже після створення проєкту в середовищі розробки Xcode. У такому випадку код буде трохи інший. Додатково слід визначити модель даних для координатора. Для цього необхідно отримати посилання на файл моделі у бандлі проєкту, після чого створити об'єкт NSManagedObjectModel, передавши у конструктор вищезгадане посилання. Все, що

залишилося, це створити NSPersistanceStoreCoordinator передавши в нього модель та налаштувати його.

3.5 Порівняння результатів власного рішення зі стандартним NSSQLiteStore та аналіз результатів

Концепція тестування досить проста. За моєю гіпотезою, персистентне сховище, реалізоване на Sqlightning має бути швидшим при запитах на отримання даних та повільнішим при записуванні даних.

Ще одне припущення є те, що при збільшенні кількості транзакцій Sqlightning має бути більш продуктивною, порівнюючи зі швидкістю запису без роздроблення, через більш оптимізовану структуру зберігання даних. Тобто, наприклад, ви можете створити 100_000 об’єктів постів у соцмережі, і зберегти їх як одна транзакція, або навпаки зберігати їх меншими частинками, таким чином зменшуючи кількість записаних об’єктів в одній операції та збільшуючи кількість транзакцій.

Тож, для порівняння цих двох реалізацій було реалізовано окремий проєкт – iStoreBanchmark.

В проєкті створено два NSPersistentStoreCoordinator: перший відповідає за CustomStore, а другий за NSSQLiteStore. Для забезпечення коректності тестування до бандлу проєкту додані приклади файлів баз даних. Для кожної версії існує один файл з ініціалізованими таблицями, але без жодних записів, він буде використовуватися для тестування створення сутностей. Другий – вже заповнена база даних, котра буде використовуватися для тестування отримання, оновлення та видалення даних. Такий підхід забезпечить більшу достовірність тестам, так як кожного разу при запуску кожного окремого тесту буде створюватися новий контекст керованих об’єктів зі скопійованим сховищем.

Для тестування реалізовано уніфікований клас TestRunner, в конструктор передається тип сховища, котре потрібно протестувати та кількість запусків

кожного тесту. Для запуску самих тестів потрібно викликати метод run () в якому і запускаються всі випробування: insertTest(), fetchTest(), updatetest() та deleteTest(). Є чотири типи тестувань, за розміром транзакції: додавання мільйона об’єктів однією транзакцією, кожні десять тисяч, кожну тисячу, кожні сто та по одному об’єкту.

Для тестування створено сутність TestEntity, котра має такі поля: вік (Integer), дата народження (Date), місто (String), країна (String), електрона адреса (String), дохід (Double), ім’я (String).

Проведемо тестування та розглянемо результати (Таблиця 1). Слід зазначити, що кожен тест запускався 10 разів і результат вимірюється як середнє значення.

	NSSQLiteStore	CustomStore
insertAllObjects	18.81	26.1
insertPerTenThousend	16.16	24.68
insertPerThousend	16.01	29.75
insertPerHundred	17.21	123.11
insertPerObject	91.62	–
fetch	1.06	0.77
updateAllObjects	18.36	69.33
updatePerTenThousend	17.88	67.7
updatePerThousend	17.4	68
updatePerHundred	18	67.98
updatePerObject	18.26	67.34
deleteAllObjects	5.68	51.01
deletePerTenThousend	4.92	48.71
deletePerThousend	5.2	47.48
deletePerHundred	6.8	47.43
deletePerObject	6.64	47.21

Таблиця 1Результати відпрацювання save, fetch, update, delete запитів у NSSQLiteStore та CustomStore

Тестування збереження нових об’єктів (insert), швидше працює в стандартній версії. В обох реалізаціях спостерігається пришвидшення роботи при збільшенні кількості транзакцій, але до певної межі. Так, наприклад для NSSQLiteStore найоптимальнішим значенням серед протестованих є запис кожні тисячу об’єктів, а для CustomStore – кожні десять тисяч. Також слід зазначити, що існує певна межа

після котрої швидкість запису помітно деградує і ці значення також відрізняються поміж втілень. Тож, це може бути цікавою темою для дослідження оптимального розміру транзакції для покращення продуктивності системи.

Тестування на отримання даних із бази даних (fetch). З результатів тестів можна зробити висновок у покращенні швидкості при даному типі запита, більше ніж на 25%. Це є суттєвим покращенням порівнюючи зі NSSQLiteStore, що підтверджує гіпотезу. Такий результат може стати критичним при виборі рушія для Core Data, у застосунках, котрі потребують більш швидкого отримання даних, та можуть знехтувати іншими недоліками.

Тестування оновлення даних (update) та видалення (delete), як і insert працюють швидше у стандартній реалізації. Слід зазначити, що різниця у цьому більш помітна порівнюючи з минулими. Також, вже немає помітного деградування продуктивності при зменшенні розміру транзакції до одного об’єкту.

Отже, за результатами цього експерименту, можна зробити висновок, що таке дослідження є вдалим, і гіпотеза справдилася. Таким чином, довівши можливість реалізовувати кастомні NSIncrementalStore для пришвидшення запитів на отримання даних. Також, цілком зрозуміло недоліки такої реалізації, що полягають у більш повільному виконанні створення, оновлення та видалення.

На основі аналізу виявлених проблем було сформовано висновок про те, що продуктивність інших операцій могла знизитись через неправильну під’єднану бібліотеку бази даних, що потребує додаткового дослідження. Ще одним, потенційним варіантом є прихованні оптимізації, що використовує NSSQLiteStore. Прийняття такого рішення стало результатом детального аналізу часу виконання операції видалення. Під час дослідження було зафіксовано надзвичайно короткий час її виконання, що викликало обґрунтовані сумніви щодо безпосереднього виклику відповідних методів SQLite.

У наступних версіях Custom Store планується покращення роботи з міграціями, реалізувавши перенесення даних до нової таблиці. Також, передбачається розширення функціоналу, додавши обробку зв’язків між сутностями. Потенційним розвитком – є дослідження та впровадження використання кешів для ще більш швидкого отримання даних.

Висновки

Результатом цього дослідження стало створення власної реалізації NSIncrementalStore, використовуючи Sqlightning в якості бази даних. Наведено перелік найпопулярніших на сьогодні механізмів зберігання даних, та проаналізовано їх переваги та недоліки у контексті роботи з великими обсягами структурованої інформації. В роботі розглянуто LMDB, її архітектуру та принципи роботи. Оглянуто спосіб її інтеграції у SQLite, як заміна стандартного B-tree рушія. Проаналізовано архітектуру фреймворку Core Data та інтерфейс власного постійного сховища. Детально описано нюанси та виклики при реалізації.

Розроблена реалізація постійного сховища є спробою пришвидшення відпрацювання запитів на отримання великої кількості даних. Таким чином, вдалося покращити роботу застосунків завдяки прискоренню fetch операції на ~ 25%, для яких швидке отримання інформації та подальший її аналіз є критично необхідним. Водночас, операції для зберігання даних (save, update, delete) відпрацьовують повільніше ніж в стандартній реалізації NSSQLiteStore, що не є критичним у цьому контексті.

В ході експерименту було знайдено безліч способів покращення новоствореного постійного сховища, що може свідчити про його потенційний розвиток.

Список використаних джерел

1. Apple Inc. File System Programming Guide. *Apple Developer*. URL: https://developer.apple.com/library/archive/documentation/FileManagement/Conceptual/FileSystemProgrammingGuide/FileSystemOverview/FileSystemOverview.html.
2. Howard C. MDB: A Memory-Mapped Database and Backend for OpenLDAP. *symas.com*. URL: http://www.lmdb.tech/media/20120322-UKUUG-MDB-txt.pdf.
3. University of Illinois. Interprocess Communication: Memory mapped files and pipes. *courses.grainger.illinois.edu*. 04.04.2014. URL: https://courses.grainger.illinois.edu/cs241/sp2014/lecture/27-IPC.pdf.
4. PHILIP A. B., NATHAN G. Multiversion Concurrency Control-Theory and Algorithms. *ACM Transactions on Database Systems*. 1983. Т. 8, вип. 4. С. 465–483. URL: https://www.cs.cmu.edu/~15712/papers/bernstein83.pdf.
5. SQLite. Architecture of SQLite. *www.sqlite.org*. URL: https://www.sqlite.org/arch.html.
6. SQLightning. *GitHub*. URL: https://github.com/LMDB/sqlightning.
7. LumoSQL. *Github*. URL: https://github.com/LumoSQL/lumosql.
8. Florian K., Daniel E. Core Data. objc.io, 2017.
9. Apple Inc. Managed object. *Apple Developer*. URL: https://developer.apple.com/library/archive/documentation/DataManagement/Devpedia-CoreData/managedObject.html#//apple_ref/doc/uid/TP40010398-CH23-SW1.
10. Apple Inc. Managed object context. *Apple Developer*. URL: https://developer.apple.com/library/archive/documentation/DataManagement/Devpedia-CoreData/managedObjectContext.html.
11. Apple Inc. Persistent store coordinator. *Apple Developer*. URL: https://developer.apple.com/library/archive/documentation/DataManagement/Dev

pedia-CoreData/persistentStoreCoordinator.html#//apple_ref/doc/uid/TP40010398-CH27-SW1.

12. Apple Inc. Persistent store. *Apple Developer*. URL: https://developer.apple.com/library/archive/documentation/DataManagement/Devpedia-CoreData/persistentStore.html#//apple_ref/doc/uid/TP40010398-CH29-SW1.
13. Apple Inc. Incremental Store Programming Guide. *Apple Developer*. URL: https://developer.apple.com/library/archive/documentation/DataManagement/Conceptual/IncrementalStorePG/ImplementationStrategy/ImplementationStrategy.html.
14. Apple Inc. Importing Objective-C into Swift. *Apple Developer*. URL: https://developer.apple.com/documentation/swift/importing-objective-c-into-swift.
15. Apple Inc. Wrapping C/C++ Library in Swift. *Swift.org*. URL: https://www.swift.org/documentation/articles/wrapping-c-cpp-library-in-swift.html.
16. Howard C. LDAP at Lightning Speed. *BuildStuffLT*, Vilnius, 20 листоп. 2014. URL: http://www.lmdb.tech/media/20141120-BuildStuff-Lightning.pdf.