Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

Курсова робота

Освітній ступінь – бакалавр

на тему: «**Порівняльна характеристика мобільних баз даних для iOS**»

за спеціальністю «Інженерія програмного забезпечення» - 121

Керівник курсової роботи

асистент Франків О.О.

_____

(підпис)

“____” _____ 2022 р.

Виконала студентка 3 курсу

Зубрицька І.І.

“____” _____ 2022 р.

Київ 2022

Календарний план виконання роботи

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання теми курсової роботи	Квітень 2022	
2.	Ознайомлення з базами даних під iOS	Квітень 2022	
3.	Визначення характеристик для порівняння	Квітень 2022	
4.	Огляд літератури та існуючих бенчмарків	Квітень 2022	
5.	Проведення власного дослідження	Травень 2022	
6.	Написання текстової частини	Травень 2022	
7.	Коригування роботи	Червень 2022	
8.	Здача курсової роботи	07.06.2022	

Зміст

Анотація

Ця курсова робота присвячена порівнянні баз даних для мобільних додатків під операційну систему iOS.

Спочатку у ній розглядаються такі бази даних/фреймворки як CoreData, Realm та Firebase, а також надається інформація про основні особливості при роботі з ними. Після цього проводиться ознайомлення з бенчмарками: що це таке та навіщо вони потрібні. І насамкінець робота надає порівняльну характеристику перелічених баз даних/фреймворків.

Вступ

Зберігання даних – одна з важливих функцій кожного додатку чи програми, без якої вони не можуть обійтися.

Мобільним додаткам майже завжди потрібно зберігати дані і робити це можна різними способами – за допомогою реляційної або нереляційної бази даних, нативного фреймворку чи зовнішньої бібліотеки. Проте виникає питання, яку з баз даних правильно обрати для того чи іншого додатку? Важливо розуміти, які з них краще працюють з невеликими даними, а які можуть зберігати тисячі даних. Або які працюють більш ефективно, а які повільніше. Які більш надійні з точки зору втрати даних, а які менш стійкі. Все це важливо, щоб додаток правильно і ефективно працював і задовольняв потреби як користувача, так і програміста.

Для додатків під iOS існує багато рішень для зберігання даних. Дечим вони схожі, дечим відрізняються, а також кожне з них має переваги та недоліки.

Тому метою цієї письмової роботи є ознайомитися з існуючими базами даних для мобільних додатків під iOS та порівняти їхню продуктивність. На основі отриманих даних можна буде зрозуміти, чи є суттєва різниця у тому, яку саме базу даних використовувати, і в яких випадках варто обирати ту чи іншу базу даних для мобільних пристроїв.

1. Огляд баз даних та їхні особливості

1.1. CoreData

CoreData — це об’єктно-реляційна система відображення(Object-Relational Mapping, або ORM) для мобільних додатків. Основною функцією Core Data є керування об’єктами даних у об’єктному графі. Другорядною, але не менш важливою функцією, є збереження даних на диску.

CoreData API, або CoreData стек, має набір класів, які підтримують шар моделі застосунку. Вони включають в себе NSManagedObjectModel, NSPersistentStoreCoordinator і NSManagedObjectContext. Усі ці частини працюють разом, щоб дозволити програмі отримувати та зберігати екземпляри NSManagedObject. CoreData стек має багато властивостей, які можна налаштовувати вручну, в тому числі й тип сховища для даних.

Екземпляр класу **NSManagedObjectModel** описує сутності об’єктного графу разом з їхніми властивостями та зв’язками[1]. Це скомпільована двійкова версія моделі даних, яку ми створюємо графічно в редакторі Xcode. Коли ми говоримо, що маніпулюємо об’єктною моделлю, то маємо на увазі, що редагуємо вихідний файл у Xcode з розширенням .xcdatamodeld, який буде скомпільовано та використано NSManagedObjectModel[3]. З точки зору бази даних, цей файл представляє схему бази даних.

Екземпляр класу **NSPersistentStoreCoordinator** є посередником між сховищем і контекстом та відповідає за збереження, завантаження та кешування даних[1]. В роботі дуже мало працюють з NSPersistentStoreCoordinator безпосередньо, зазвичай під час ініціалізації. Але майже ніколи не торкаються його знову протягом життя програми.

Екземпляр класу **NSManagedObjectContext** відслідковує зміни у сутностях, керує ними[1].

NSPersistentStore – це абстрактний клас для сховищ CoreData[1]. CoreData надає декілька типів(NSPersistentStore.StoreType) сховищ для даних: in-memory сховище та постійні сховища на диску. Зокрема, розглянемо постійні сховища. Вони бувають атомарні та інкрементальні, або NSAtomicStore та NSIncrementalStore відповідно.

Перш ніж перейти до розгляду кожного з типів сховищ Core Data, з'ясуємо загальний принцип їхньої роботи – що є відповідальністю Core Data, а що є відповідальністю розробника.

Core Data відповідає за:

- Ініціалізацію сховища
- Обробку запитів
- Керування контекстами

У свою чергу розробник відповідає за:

- Створення бази – сутностей і зв'язків між ними
- Генерацію унікальних ідентифікаторів для екземплярів сутностей
- Визначення метаданих для сховища

На рисунку наведене порівняння різних типів сховищ CoreData:

Store type	Speed	Object graph in memory	Other factors
XML (atomic)	Slow	Whole	Externally parsable
Binary (atomic)	Fast	Whole	N/A
SQLite	Fast	Partial	N/A

РИСУНОК 1 - ПОРІВНЯННЯ СХОВИЩ CORE DATA[2]

1.1.1. Атомарний тип сховища NSAtomicStore

Атомарні сховища надають перевагу простоті над продуктивністю. Існуючі рішення включають в себе двійкові(NSPersistentStore.StoreType.binary) та XML(NSPersistentStore.StoreType.xml, але він не підтримується OSX) сховища. Вони записуються на диск атомарно, тобто під час кожного збереження даних весь файл перезаписується. Такий підхід має свої переваги та недоліки. На жаль, вони гірше масштабуються в порівнянні з іншими сховищами і коли до них звертатися, вони повністю завантажуються в пам'ять, що перевантажує програму. Проте, оскільки дані завжди знаходяться в оперативній пам’яті, робота з ними дуже швидка. Тому атомарні сховища чудово підходять для програм, які використовують невеликі обсяги даних.

Крім використання вже існуючих сховищ, за допомогою API атомарних сховищ їх можна налаштовувати вручну і вибрати зручний формат файлу для збереження даних, наприклад, JSON, HTML, XML чи CSV файл. У разі реалізації власного сховища, його тип часто залежить від самої моделі даних, що зберігається. Але важливо пам’ятати, що API атомарних сховищ не підтримує інтеграцію з реляційними базами даних або подібними сховищами на базі SQL.

1.1.2. Інкрементальний тип сховища NSPersistentStore

Напротивагу атомарним сховищам, з інкрементальними може бути складніше працювати, але вони дозволяють ефективно працювати з великим обсягом даних. Тобто в пам'ять заносяться лише ті дані, з якими потрібно працювати, а не весь вміст сховища. Аналогічно, при збереженні змін сховище записує лише нові дані, а не перезаписує цілий файл. Тому їх доцільно використовувати, коли занесення всіх даних в пам'ять неможливе, тобто при роботі з великими даними. Проте робота інкрементальних

сховищ з невеликими даними досить неефективна, бо якщо потрібно часто звертатися до даних, кожен раз програма буде доступатися до постійної пам’яті.

Єдиним існуючим типом інкрементальних сховищ CoreData у мобільних додатках є сховище SQLite. Воно використовується і для iOS, і для OS X. Це програмна бібліотека, яка реалізує автономний, безсерверний, транзакційний механізм баз даних SQL з нульовою конфігурацією[3] і має свої переваги.

Однією з переваг є масштабування. Коли програма використовує реляційну базу даних, вона може масштабуватися до великих розмірів. Сам SQLite був протестований терабайтами даних і може обробляти майже все, що програмісти можуть реально розробити[3]. Оскільки ми завантажуємо лише ті дані, які нам потрібні в певний момент, SQLite зберігає обсяг пам’яті нашої програми досить низьким. Так само SQLite ефективно використовує свій дисковий простір і, отже, займає невелику площу на диску.

Наступна причина використовувати SQLite це можливість налаштовувати різні параметри продуктивності. Працюючи з базою даних замість звичайного файлу, ми маємо доступ до різноманітних параметрів для налаштування бази даних. Наприклад, можна індексувати стовпці в наших сутностях, щоб увімкнути швидші предикати. Також можна контролювати, що саме завантажується в пам'ять. Можна отримати лише кількість об’єктів, лише унікальні ідентифікатори для об’єктів тощо. Ця гнучкість дозволяє налаштовувати продуктивність програми більше, ніж будь-який інший тип сховища[3].

1.2. Realm

Realm Database – це система керування базами даних на основі NoSQL з відкритим кодом. Дані зберігаються у вигляді JSON у файлах. Написана на мові C++, ця СКБД є крос-платформною і використовується не тільки в iOS, а й в Android і веб розробці. Якщо Core Data це більш традиційний та надійний спосіб зберігати дані, то Realm відносно нова СКБД. Незважаючи на свій вік, вона вже активно використовується розробниками завдяки її багатьом перевагам, зокрема легкість у використанні.

Модель даних

Модель даних Realm визначається як звичайні класи у Swift, що наслідуються від класу Object[6] (він є частиною бібліотеки RealmSwift). Вони мають звичні для класів Swift властивості, можуть конформити протоколи і реалізовувати потрібні методи. Єдине обмеження, яке накладається, це те, що вони можуть використовуватися лише в тому потоці, у якому були створені.

Властивості класів обов’язково мають мати значення за замовчуванням або бути опціональними. Також вони мають бути позначені атрибутами @objc dynamic. Це потрібно, щоб поля були доступні у часі виконання Objective-C через Dynamic Dispatch. У нових версіях можна використовувати обгортку @Persisted для заміни @objc dynamic.

Через те, що Realm є крос-платформною, то і властивості класів мають підтримуватися різними платформами. Тому Realm підтримує лише типи такі, як Bool, Int, Double, Float, String, Date, Data. З них опціоналами можуть бути тільки 3 останні, для інших використовується обгортка RealmOptional<T>.

Для ефективності роботи з базою при читанні чи оновленні даних, зручно позначити якийсь атрибут первинним ключем для ідентифікації об’єкту. Він незмінний, тому встановити можна тільки раз.

Приклад класу з первинним ключем:

```
class MyObject: Object {
  @objc dynamic var id = UUID()
  @objc dynamic var name: String = “”
ч
  override static func primaryKey() -> String? {
        “id”
  }
}
```

Щоб почати роботу з Realm, потрібно доступитися до екземпляру класу Realm, за допомогою якого можна буде робити зміни в базі даних. Це робиться наступним чином:

```
let realm = try! Realm()
```

Будь-яка зміна в базі даних – створення, оновлення та видалення даних – має відбуватися всередині транзакції запису. Оскільки це синхронізована операція, за можливості варто робити кілька змін одночасно. Такий підхід є оптимізованішим, аніж створювати багато транзакцій і робити невеликі зміни. Приклад з використанням транзакції запису:

```
try! realm.write {

  realm.add(someObject)

}
```

У програмі ми можемо створювати і використовувати об’єкти моделі даних Realm і вони будуть називатися некерованими до тих пір, поки ми не збережемо їх до бази даних. Але як тільки Realm збереже об’єкти, вони називатимуться керованими, і це означає, що тепер зміни у цих об’єктах будуть відслідковуватися і зберігатися у базі.

Читання даних

Щоб отримати дані з бази, використовуються запити. Для цього потрібно викликати метод objects() на екземплярі класу Realm і вказати тип, який потрібно зчитати. За потреби дані можна фільтрувати, сортувати. Метод повертає Results<T>, що є узагальненою колекцією даних, схожою на масив, її також можна ітерувати. Крім того, вона має свої особливості:

- Дані, що зчитуються, не копіюються, ми отримуємо прямі посилання на них. І при зміні даних, вони будуть відразу зберігатися.
- Дані автоматично оновлюються. Якщо мати посилання на одні і ті ж дані у кількох частинах програми, то зміни в одній частині призведуть до змін в іншій.
- Дані зчитуються ліниво(lazy data loading) і завантажуються тільки тоді, коли ми доступаємося до них[7].

Приклад читання даних:

```
let objects = realm.objects(ClassName.self)
```

Підтримка зв’язків

За допомогою Realm можна визначити будь-який зв'язок: one-to-one, one-to-many, many-to-many. В такому випадку властивості не треба позначати @objc dynamic, оскільки типи, що використовуються, самі по собі є типу Object.

One-to-one зв'язок мусить бути опційним, інакше програма аварійно завершиться під час виконання. Для позначення one-to-many зв’язку використовується список потрібного типу. У прикладі маємо батька класу Parent, який може мати багатьох дітей, і позначимо цю властивість так:

```
let children = List<Child>()
```

Отримання двонапрямленого зв’язку досягається наступним полем у класі Child:

```
let parent = LinkingObjects(fromType: Parent.self, property: “children”)
```

Які переваги Realm? Основними з них є:

- Гнучкість. Через те, що Realm працює на основі NoSQL, вона має можливість легко горизонтально розширюватися. З точки зору бізнесу, це можливість робити часті оновлення додатку і виходити на ринок.
- Швидкість. Це забезпечується тим, що: не потрібно відображати дані з БД у об’єкти Swift, так як ми зберігаємо і використовуємо один і той самий клас; немає копіювання, ми працюємо з прямими посиланнями; ліниве завантаження даних.
- Крос-платформність. Як вище було зазначено, Realm використовується в iOS, Android і веб розробці.
- Простота у використанні.
- Дані можна зручно переглядати за допомогою допоміжних додатків, наприклад, MongoDB Realm Studio.

1.3. Firebase

Firebase – це backend-as-a-service(BAAS) платформа для iOS, Android та веб додатків. Вона багатофункціональна і містить компоненти такі, як Realtime Database, Firestore, Firebase Authentication, Firebase Crashlytics, AdMob та інші.

В межах цієї роботи розглядаються бази даних, які пропонує Firebase, а саме Firebase Realtime Database і Firebase Cloud, або ж Firestore. Початково існувала тільки Realtime Database, згодом з'явився Firestore. Вони обидві є NoSQL базами даних, які працюють в хмарі, тому вони гнучкі, крос-платформні і мають можливість ділитися даними між кількома пристроями.

Щоб почати роботу з однією з цих баз даних, потрібно виконати наступні кроки:

- Зареєструватися на google.firebase
- Створити новий проект. За замовчуванням(безкоштовно) можна створити до 15 проектів. По суті це ми додаємо бекенд для додатку
- Вибрати платформу iOS
- Ввести потрібні дані(Bundle id, назва проекту, AppStore id)
- Зареєструвати свій додаток до створеного проекту на Firebase
- Завантажити .plist файл
- Додати його до свого додатку
- Додати залежність Firebase до свого додатку за допомогою одного з пакетних менеджерів

- Налаштувати сервіс Firebase за допомогою виклику методу FirebaseApp.configure() в методі класу AppDelegate application(_:didFinishLaunchingWithOptions:)

Коли початкові налаштування виконано і тепер в додаток інтегрований Firebase, потрібно вирішити, яку саме базу даних використовувати. Для цього у консолі Firebase потрібно вибрати одну з опцій – Realtime Database або Firebase Cloud.

Дані, що зберігаються у базі, мають бути захищеними. Для цього потрібно налаштувати правила захисту даних в базі. Є 3 типи правил:

- Private – читання і редагування даних заборонене
- Public – читання і редагування даних відкрите для будь-кого(не рекомендується, хіба що для навчальних проектів)
- User – оптимальний варіант, коли користувач має доступ до читання і редагування тільки своїх даних[8]

1.3.1. Firebase Realtime Database

База даних Firebase Realtime Database зберігає усі дані в одному великому JSON-дереві. З назви Realtime Database зрозуміло, що зміни даних відслідковуються в реальному часі. Для бази даних можна увімкнути збереження даних в локальному сховищі на пристрої користувача, коли він не під’єднаний до мережі. А коли зв'язок відновиться, дані синхронізуються з хмарою і збережуться в основну базу даних.

У JSON-дереві дані можна вкладати одні в одні, але у Firebase Realtime Database існує обмеження на кількість цих вкладень, тому варто бути обережним з додаванням даних «вглиб». Натомість гарним тоном є робити базу максимально «плоскою», тобто уникаючи багатьох вкладень.

Так буде і простіше, і ефективніше працювати з базою, оскільки при читанні даних зчитуються всі дочірні гілки.

Щоб працювати з даними, потрібно встановити зв'язок з базою даних і робиться це наступним чином:

```
let reference = Database.database().reference()
```

Читання відбувається за допомогою відстеження змін даних. Для цього викликається метод observe(_:with:), куди передається подія, яку потрібно відслідкувати, і замикання. Існують такі події:

- .value – відстежує зміни всіх всього вмісту заданого шляху разом з вкладеними даними
- .childAdded – відстежує додавання даних на вказаному шляху
- .childChanged – відстежує зміну даних
- .childRemoved – відстежує видалення даних
- .childMoved – відстежує зміну порядку даних[8]

Приклад виклику:

```
reference.observe(.value) { [weak self] snapshot in
    // process snapshot data
}
```

Параметр snapshot у замиканні – це всі дані з шляху. Якщо даних немає, то snapshot є nil.

Щоб додати дані, використовують метод setValue(_:) на потрібному шляху. Якщо дані до запису там вже існували, вони будуть перезаписані, в іншому випадку – створяться. Передати можна число, стрічку, масив або словник значень. Приклад:

```
reference.child("path").setValue(["key":"some value"])
```

Для оновлення даних так само можна використовувати setValue(_:), якщо є сенс повністю перезаписати дані. Або ж на потрібному шляху викликається метод updateChildUpdates(_:), якому передаються нові значення у вигляді словника. При цьому оновлюються тільки певні значення. Видалення реалізується через removeValue() на заданому шляху.

Як було сказано, Firebase може зберігати дані локально, якщо користувач є офлайн. Для цього потрібно встановити значення isPersistenceEnabled на true:

```
reference.isPersistenceEnabled = true
```

1.3.2. Firebase Cloud(Firestore)

Firebase Cloud, або Firestore, є схожою на Realtime Database, вона так само зберігає дані в реальному часі, відслідковує зміни і може обробляти дані навіть коли користувач офлайн. Але це новіша і покращена альтернатива. На відміну від Realtime Database, дані зберігаються в колекціях документів, тому їх легше організовувати, і, крім того, має кращу продуктивність.

На найвищому рівні база даних Firestore містить колекцію або кілька. Вони можуть містити у собі тільки документи. Колекції і документи мають мати унікальні імена. У свою чергу документи містять дані у вигляді ключ-значення. Значення підтримують більше типів, ніж у Firestore Database, а саме: String, Bool, Array, Bytes, Date and Time, Number, Geographical Point, nil, reference, Map[9]. Також документи можуть містити цілі підколекції даних. Проте вони не можуть містити у собі документів. Таким чином будується гнучка структура даних.

Щоб додати дані, потрібно створити хоча в одну колекцію і документ. Спочатку створюється сама база, після чого в неї додаються дані. Приклад створення колекції, документу в ній та даних у вигляді словника:

```
let reference = Firestore.firestore()

reference.collection("collection name").document("doc name").setData(["key": "value"])
```

Так само як у Realtime Database, setData(_:) створить нові дані або перезапише старі, якщо вони вже існують на тому шляху. Щоб дані лише оновити, то потрібно додати параметр, вказавши, що дані треба об'єднати:

```
setData([], merge: true)
```

Дані зчитуються з документу або кількох на певному шляху. Щоб побудувати шлях, використовується collection(_:) або document(_:). Дані можна зчитувати один раз або постійно відслідковувати зміни.

Для видалення цілого документу викликається метод delete() на потрібному документі. Для видалення лише якогось поля, для певного ключа встановлюється значення FieldValue.delete(). Встановлення nil нічого не дасть, так як це підтримуваний тип, і дані все одно будуть зберігатися. Приклад видалення:

```
reference.collection("a").document("b").updateData(["k":FieldValue.delete()])
```

2. Бенчмарки

2.1. Визначення бенчмарків і бенчмаркінгу

Бенчмарк - термін, що може мати кілька значень. Він прийшов з галузі економіки і перекладається як орієнтир. Якщо коротко, то це тест продуктивності. У комп'ютерній термінології бенчмарк може мати також наступні визначення:

- Спеціалізоване ПЗ, яке забезпечує вимірювання продуктивності конкретної ОС чи застосунку
- Відомий продукт, знайомий багатьом користувачам, такий що з ним можна порівнювати нові продукти
- Набір критеріїв, яким має відповідати продукт

Відповідно, бенчмаркінг - це процес порівняння продукту, сервісу чи процесу за певними критеріями з показниками інших компаній, зазвичай більш успішних[10]. Його головною метою є визначення можливостей покращення продукту. Методів вдосконалення існує 2 - неперервні(continuous) та драматичні(dramatic). Неперервне вдосконалення відбувається поступово і передбачає постійні невеликі зміни аби досягти кращого результату. Драматичне вдосконалення відбувається за повного реінжинірингу робочого процесу.

2.2. Етапи бенчмаркінгу

Бенчмаркінг продукту відбувається у кілька етапів, розглянемо їх детальніше.

Перший очевидний крок - це вибрати об'єкт для тестування. Це може бути продукт, сервіс чи й ціла компанія. У нашому випадку об'єктом є конкретна мобільна база даних чи фреймворк.

Наступним кроком є пошук та аналіз конкурентів, або можна називати їх бенчмарками. При пошуку важливо обрати саме ті аналоги власному продукту, які мають максимально схожий функціонал. Після цього йде збір даних про них з метою отримання інформації щодо продуктивності роботи та інша статистика. Потім ці дані аналізуються.

Коли дані вже отримані, час порівняти їх з якістю свого продукту. Це допоможе визначити, де продукт має недостатньо хороші показники аби бути конкурентноспроможним. Коли слабкі місця виявлено, починається робота над вдосконаленням. Це супроводжується запровадженням нових процесів та правил, які зможуть оптимізувати роботу продукту та наблизитися до бажаного результату.

2.3. Переваги та недоліки бенчмарків

Переваги

Бенчмаркінгом займаються більшість американських та європейських компаній, адже він має багато переваг. Зокрема, це встановлення нових цілей і постійне вдосконалення, які дуже важливі у світі, що швидко розвивається. Крім того, порівняльний аналіз власного продукту з конкурентами та отримані результати допомагають вирішити проблеми продукту.

Неочевидною, але не менш важливою особливістю бенчмаркінгу є також кооперація та тімбілдинг. Спільна праця над вдосконаленням продукту спонукає більшу зацікавленість працівників у його розробці. Це безумовно новий досвід і певний освітній процес - знайомство членів команди з метриками ефективності та способами покращення продукту/процесу.

Цікавий факт. Бенчмарки надважливі при розробці процесорів саме для мобільних телефонів, оскільки їх неможливо або майже неможливо замінити в телефоні в порівнянні з ПК, тому вони мають бути дуже надійними.

Недоліки

Як би багато переваг не мали бенчмарки, у них також є і недоліки. До прикладу, бенчмаркінг - це спосіб знайти слабкі місця продукту, де він програє своїм аналогам. Але він не дає відповіді як вирішити ці проблеми, це залишається відповідальністю команди розробників.

Іншою складністю є те, що часто інформація про конкурентів важкодоступна, тому потрібно добре постаратися, аби її роздобути. Іноді це бувають двосторонні домовленості, аби взаємно обмінятися інформацією.

Крім того, з бенчмаркінгом виникає проблема у браці часу, адже цей процес може бути складним і затяжним. Тому він потребує багато сил та часу на опрацювання. А за браку досвідчених фахівців, це також можуть бути фінансові витрати аби найняти компетентних людей.

2.4. Бенчмарки для баз даних

Ознайомившись у загальному, що таке бенчмарки, можна перейти до конкретного прикладу. Так як у цій роботі розглядаються мобільні бази даних під iOS, визначимо, за якими саме критеріями будемо надавати порівняльну характеристику.

Для початку, кілька порад, як підготуватися до оцінки продуктивності бази даних:

- Визначити потреби. Що є пріоритетом - безпека при роботі з БД, швидкість виконання запитів, підтримка різних типів зв'язків, кількість використовуваної пам'яті?

- Підготувати ПК. Це означає звільнити пам'ять, обмежити процеси, які відбуваються на бекграунді, бо це все має вплив на результат тестування.

- Оптимізувати. Для порівняння роботи різних БД важливо, щоб вони, скажімо, були в однакових умовах. Тому найкраще їх порівнювати, коли робота з кожною базою реалізована найбільш оптимізовано.

- Багато тестувати. Щоб отримати найбільш точні результати, тести потрібно запускати багато разів, це зменшить ймовірні побічні ефекти.

Люди проводять досить багато часу в мобільних телефонах з метою роботи/розваги чи будь-якої іншої активності. Звичайно вони очікують, що застосунки, які вони використовують, мають мати високу продуктивність, адже вони зацікавлені у економії свого часу.

З точки зору розробників, окрім продуктивності запитів, цікаво порівняти бази даних за підтримкою різних зв'язків між сутностями. Так як дані можуть мати найрізноманітнішу структуру, залежно від предметної області, програмісти зацікавлені у тому, як найзручніше і найефективніше зберігати дані програми.

Саме тому в межах цієї письмової роботи, цікаво порівняти мобільні бази даних за швидкістю виконання запитів.

3. Дослідження та аналіз отриманих даних

3.1. Підтримка зв’язків

У наведеній нижче таблиці наведені підтримувані зв’язки у кожній з баз даних, що розглядаються:

База даних	1-to-1	1-to-m	m-to-n
CoreData SQLite	+	+	+
Realm	+	+	+
Firebase Database	+	+	+
Firestore	+	+	+

Можна довести, що кожна з баз даних підтримує усі зв’язки, реалізувавши її у кожній з них. Розглядатимемо базу даних з сутностями Студент, Профіль студента, Курс, Предмет. Маємо зв’язки:

- 1 до 1 – кожен студент має створений профіль з електронною поштою
- 1 до багатьох – один курс має кілька предметів, але предмет може належати тільки до одного курсу
- Багато до багатьох – запис студентів на курси

У випадку використання CoreData з СКБД SQLite очевидно, що підтримуються всі потрібні типи зв’язків, оскільки дані зберігаються реляційній базі даних. На рисунку 1 приклад реалізованої моделі у XCode.

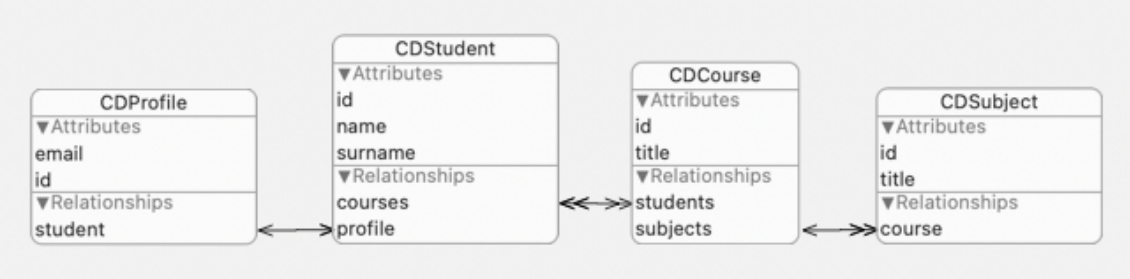

РИСУНОК 2 - МОДЕЛЬ ДАНИХ У CORE DATA

Оскільки Realm, Realtime Database та Firestore – NoSQL бази даних, то для них будуть діяти схожі правила для підтримки зв’язків. Це можна зробити кількома способами, реалізація залежить від потреб(наприклад, важливий розмір бази даних і мінімальна кількість дублікатів чи швидкий доступ до даних) і вирішується в конкретному випадку.

Про реалізацію різних зв’язків у Realm йдеться у розділі 1.2. Таким чином була створена база даних на рисунку 2. Відкрита за допомогою MongoDB Realm Studio.

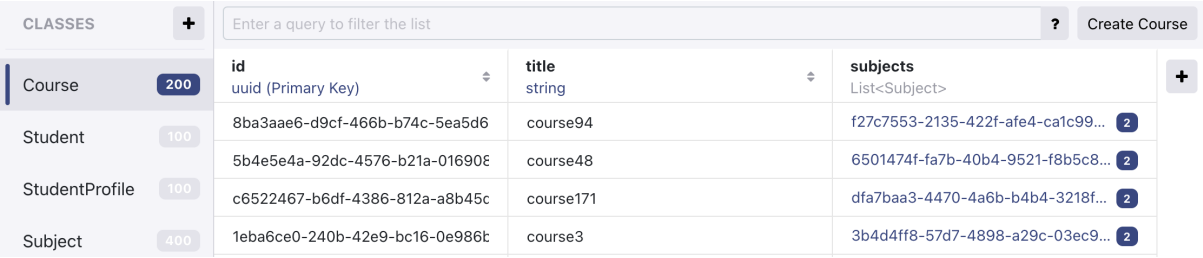

РИСУНОК 3 - МОДЕЛЬ ДАНИХ У REALM

Вміст баз даних Firebase можна переглядати у веб-браузері. На рисунку 3 зображено як виглядає модель даних у Realtime Database. Для дослідження профіль студента знаходиться тільки у самому студенті. Але якщо в реальному житті довелося би часто доступатися до електронних скриньок студентів, було би доречніше ще й самі профілі тримати окремо для швидшого доступу. Нехай це дублювання даних, але воно не критичне.

РИСУНОК 4 - МОДЕЛЬ ДАНИХ У REALTIME DATABASE

Така сама модель зі студентами, реалізована у Firestore, має вигляд, як зображено на рисунках 4 та 5.

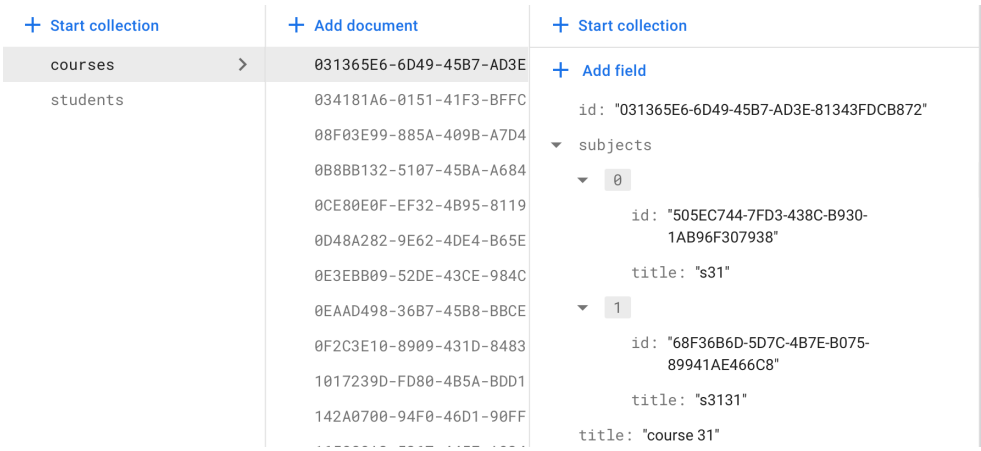

РИСУНОК 5 - МОДЕЛЬ ДАНИХ У FIRESTORE Ч.1

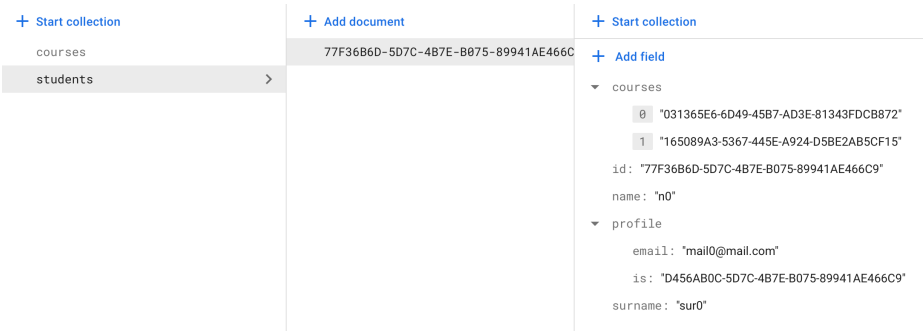

РИСУНОК 6 - МОДЕЛЬ ДАНИХ У FIRESTORE Ч.2

3.2. Час виконання запитів

Порівняння CoreData і Realm. На рисунках 6-9 результати виконання CRUD-операцій у базі даних, що містить 1000 об’єктів з одним полем «test». Вони протестовані на даних розміром від 1 до 10000 об’єктів на симуляторі iPhone 6S.

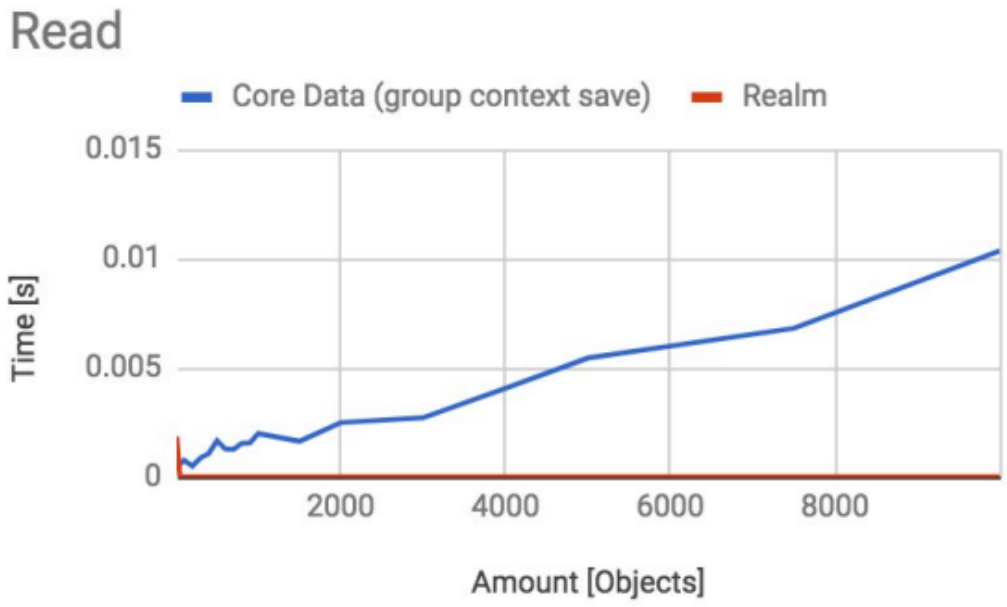

РИСУНОК 7 - ПОРІВНЯННЯ ШВИДКОСТІ ЧИТАННЯ COREDATA І REALM [11]

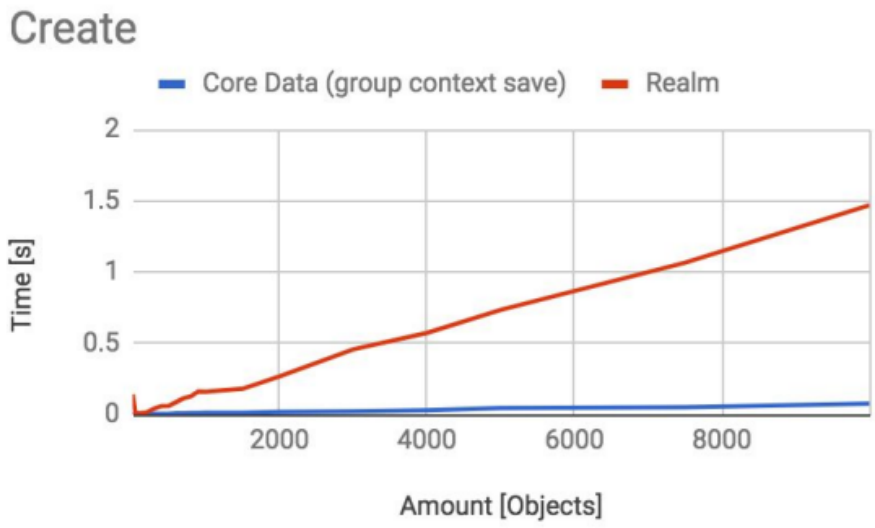

РИСУНОК 8 - ПОРІВНЯННЯ ШВИДКОСТІ ЗАПИСУ COREDATA І REALM[11]

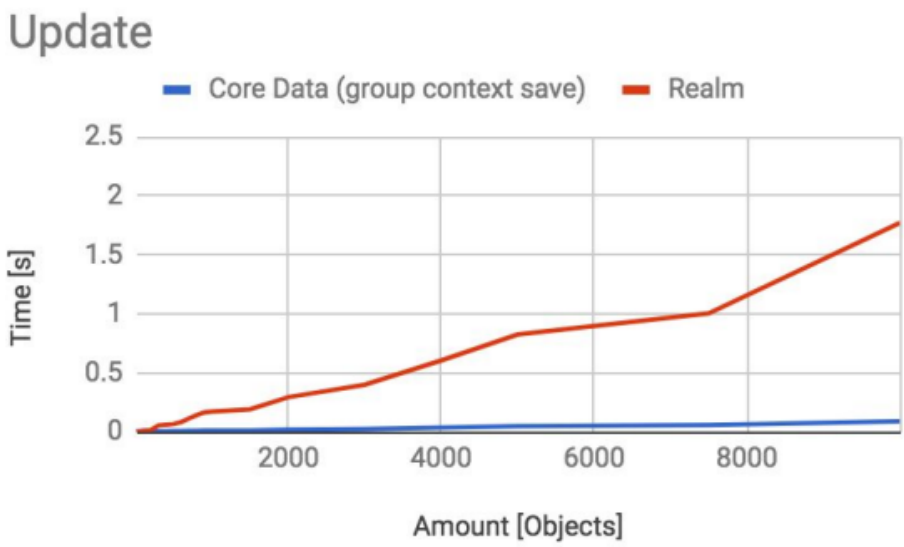

РИСУНОК 9 - ПОРІВНЯННЯ ШВИДКОСТІ ОНОВЛЕННЯ REALM І COREDATA [11]

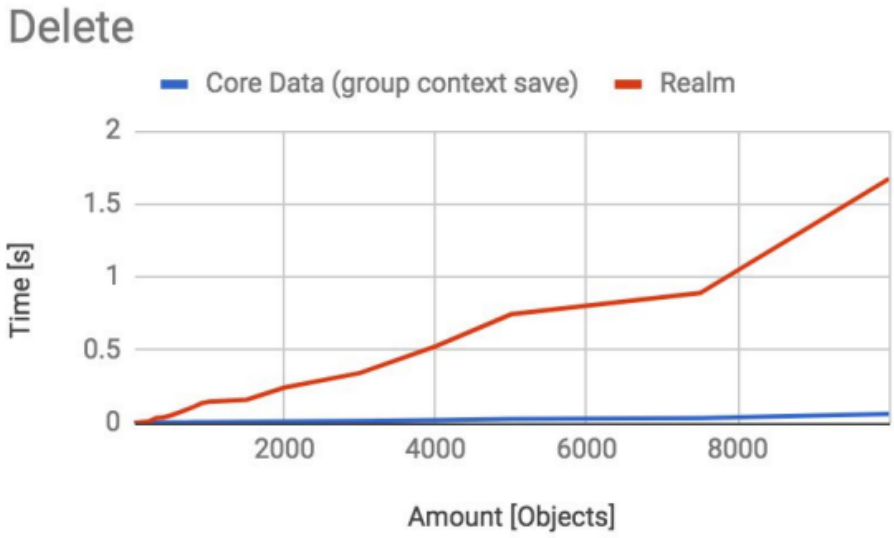

РИСУНОК 10 - ПОРІВНЯННЯ ШВИДКОСТІ ВИДАЛЕННЯ REALM І COREDATA [11]

Порівняння Realm і Realtime Database. Ці дві бази даних були протестовані даними про студентів та їхні курси розміром від 1 до 10000 об’єктів і на симуляторі iPhone 11. Для чистоти експерименту тестування відбувалося багато разів, щоб переконатися, що щоразу результати

виходять наближені одні до одних. Для заміру часу запитів використовувався метод CACurrentMediaTime() .

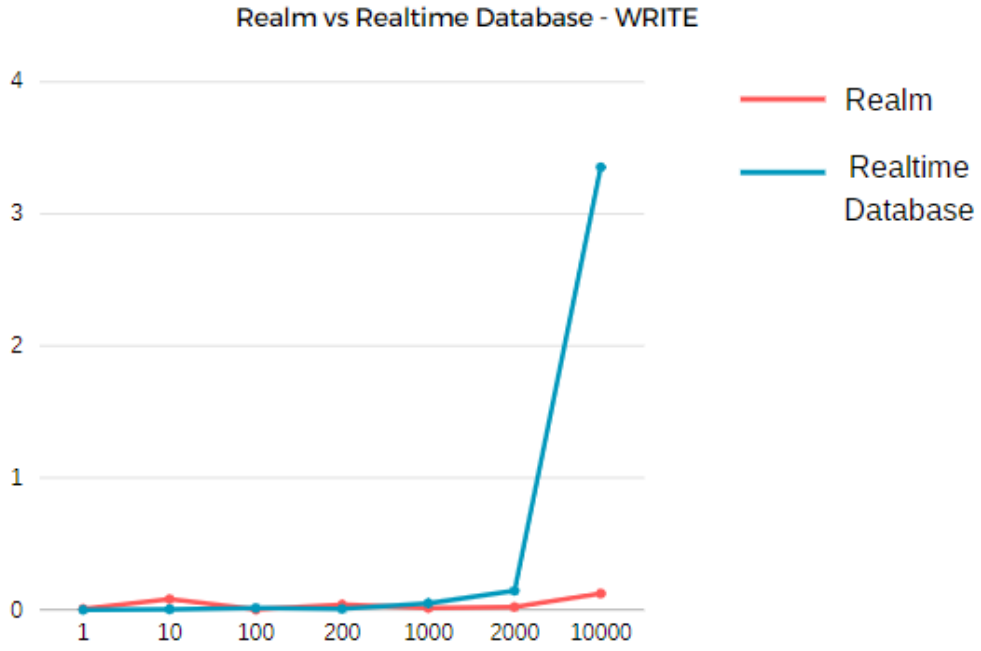

РИСУНОК 11 - ПОРІВНЯННЯ ШВИДКОСТІ ЗАПИСУ REALM І REALTIME DATABASE

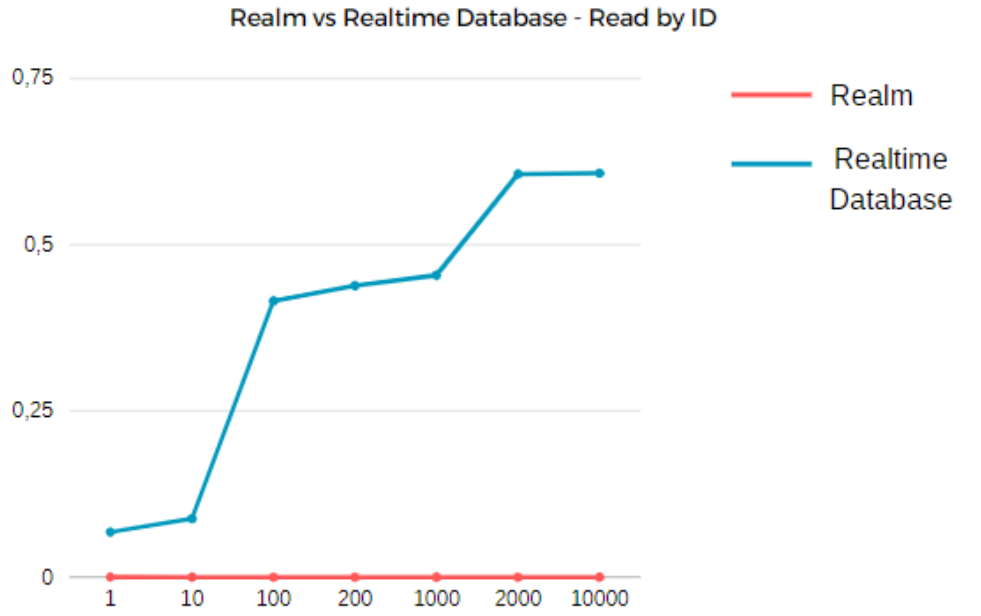

РИСУНОК 12 - ПОРІВНЯННЯ ШВИДКОСТІ ЧИТАННЯ ЗА ІДЕНТИФІКАТОРОМ REALM І REALTIME DATABASE

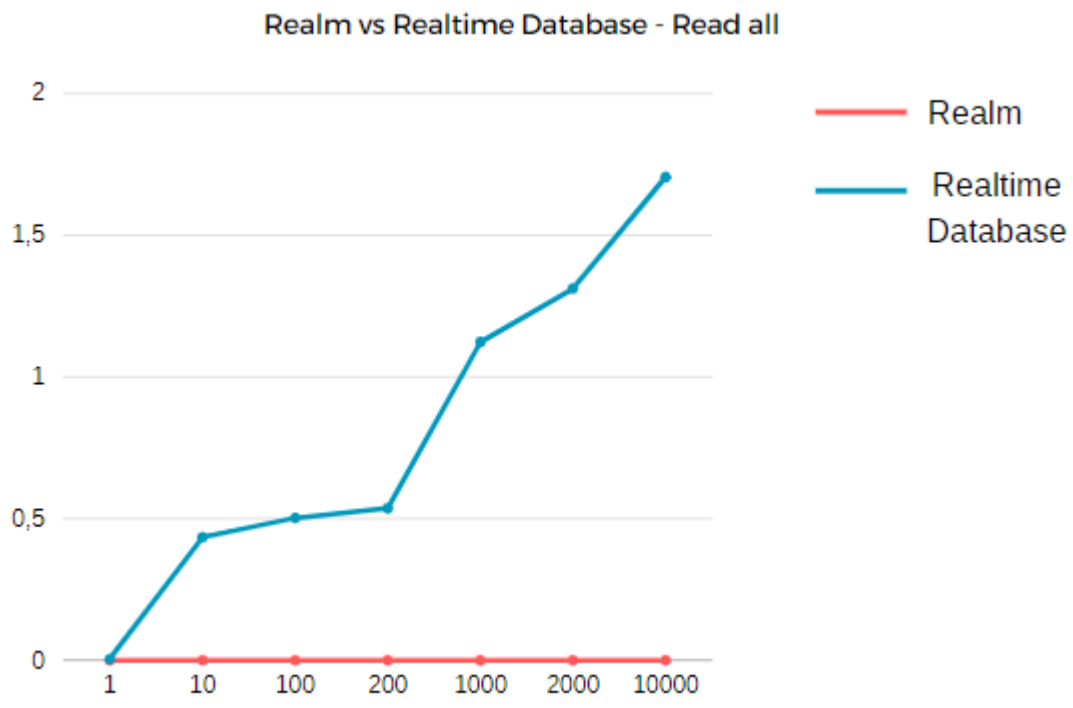

РИСУНОК 13 - ПОРІВНЯННЯ ШВИДКОСТІ ЧИТАННЯ ВСІХ ОБ'ЄКТІВ REALTIME DATABASE

На основі отриманих результатів, можна зробити висновок, що найшвидше працює база даних Realm. Крім того, вона проста у використанні і не потребує багато коду. Але досі хорошими залишається варіант CoreData як надійний спосіб зберігання даних в реляційній базі даних. Але так як з нею складніше працювати, на її розробку може йти більше часу. Продуктивність Realtime Database з усіх порівнюваних баз даних є середньою. Читання відбувається досить швидко, а от запис на 10000 об’єктах вже трохи довга.

Висновки

У межах виконання курсової роботи було описано і порівняно 4 різні бази даних/фреймворки для iOS розробки, таких як CoreData, Realm, Firebase Realtime Database та Cloud Firestore.

У першій частині детально розглядаються усі запропоновані бази даних з їхніми основними компонентами та принципами роботи. Для кожної бази даних описується модель та як працювати з даними – читати, записувати, видаляти.

У другій частині є ознайомлення з бенчмарками, бенчмаркінгом, чому вони потрібні. Крім того, визначилися умови і технічні характеристики для бенчмаркінгу баз даних.

У третій частині бази даних порівнюються за типами зв’язків, які можна в них реалізувати, а також замірами тривалості запитів. Для візуального надані графіки, за допомогою яких легко робити висновки. Для порівняння було використано як і існуючі, так і власні бенчмарки.

У подальшому можна буде дослідити цю тему ще глибше і порівняти ці бази даних і за іншими характеристиками та метриками.

Список використаних джерел

1. Core Data – Apple Developer Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://developer.apple.com/documentation/coredata
2. Core Data Programming Guide [Електронний ресурс] – Режим доступу до ресурсу: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CoreData/index.html#//apple_ref/doc/uid/TP40001075-CH2-SW1
3. Marcus S. Zarra. (2016). Core Data in Swift [Електронний ресурс] – Режим доступу до ресурсу: https://learning.oreilly.com/library/view/core-data-in/9781680502046/
4. Avi Tsadok (2022). Core Data – Building a Custom Store [Електронний ресурс] – Режим доступу до ресурсу: https://betterprogramming.pub/core-data-building-a-custom-store-84d19f39dec4
5. Tobias Andersson (2018). Analysis and quantitative comparison of storage, management, and scalability of data in Core Data system in relation to Realm [Електронний ресурс] – Режим доступу до ресурсу: https://www.diva-portal.org/smash/get/diva2:1204209/FULLTEXT01.pdf
6. Realm Swift SDK [Електронний ресурс] – Режим доступу до ресурсу: https://www.mongodb.com/docs/realm/sdk/swift/
7. Aasif Khan(2021). Getting Started With Realm Database in Swift [Електронний ресурс] – Режим доступу до ресурсу: https://www.appypie.com/realm-database-swift-getting-started
8. Realtime Database Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://firebase.google.com/docs/database

9. Cloud Firestore Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://firebase.google.com/docs/firestore

10. Що таке бенчмаркінг і як його використовувати [Електронний ресурс] – Режим доступу до ресурсу: https://www.interkassa.com/blog/chto-takoe-benchmarking-i-kak-ego-primenyat-dlya-internet-magazina/

11. Analysis and quantitative comparison of storage, management, and scalability of data in Core Data system in relation to Realm [Електронний ресурс] – Режим доступу до ресурсу: https://www.diva-portal.org/smash/get/diva2:1204209/FULLTEXT01.pdf