Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра комп'ютерних наук
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

магістр
(назва освітнього ступеня)

на тему: Аналіз і реалізація стратегій об`єднання SQL NoSQL баз даних для сучасних додатків.

Виконав: студент VI курсу, групи СНнм-61
спеціальності 122 Комп'ютерні науки
(шифр і назва спеціальності)

_____ Шимків В. С.
(підпис) (прізвище та ініціали)

Керівник _____ Никитюк В. В.
(підпис) (прізвище та ініціали)

Нормоконтроль _____ Дуда О. М.
(підпис) (прізвище та ініціали)

Завідувач кафедри _____ Боднарчук І.О.
(підпис) (прізвище та ініціали)

Рецензент _____ _____
(підпис) (прізвище та ініціали)

Тернопіль
2025

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра комп'ютерних наук
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Боднарчук І.О.
(підпис) (прізвище та ініціали)
«____» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Магістр
(назва освітнього ступеня)
за спеціальністю 122 Комп'ютерні науки
(шифр і назва спеціальності)
Студенту Шимків Владислав Сергійович
(прізвище, ім'я, по батькові)
1. Тема роботи Аналіз і реалізація стратегій об`єднання SQL і NoSQL баз даних

Керівник роботи Никитюк Вячеслав Вячеславович, к.т.н., доцент кафедри КН
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
Затверджені наказом ректора від «29» листопада 2024 року № 4/7-1139
2. Термін подання студентом завершеної роботи 30 травня 2025 р.
3. Вихідні дані до роботи Наукові публікації та технічна документація щодо SQL (SQLite/Room) та NoSQL(Firebase Firestore) баз даних, їх інтеграції, синхронізації та забезпечення офлайн-режиму в мобільних Android-додатках
4. Зміст роботи (перелік питань, які потрібно розробити)
Вступ Розділ 1. Аналіз SQL, NoSQL та стратегій їх інтеграції для мобільних застосунків Розділ 2. Проектування та реалізація гібридної SQL/NoSQL архітектури в Android-застосунку Розділ 3. Наукове обгрунтування, апробація та перспективи розвитку гібридної моделі SQL/NoSQL для мобільних платформ Розділ 4 Охорона праці та безпека в надзвичайних ситуаціях Висновок Список використаних джерел Додатки

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
1. Титульна сторінка. 2. Актуальність теми. 3. Мета та задачі дослідження. 4.Об`єкт, предмет та методи дослідження. 5. Основні стратегії інтеграції SQL та NoSQL 6. Архітектура Мобільного застосунку з гібридним сховищем. 7. Реалізація сховищ Room та Firebase Firestore. 8 Механізми гібридної синхронізації та вирішення конфліктів. 9. Резервне Копіювання даних через Google Drive. 10. Наукова новизна та практична значущість. 11. Експериментальна перевірка та результати. 12. Висновки та перспективи розвитку. 13. Завершальний слайд.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Охорона праці	Сенчишин В.С., доцент		
Безпека в надзвичайних ситуаціях	Клепчик В.М., ст. викладач		

7. Дата видачі завдання 29 листопада 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи	29.11.2024	
2.	Підбір та опрацювання наукових публікацій, збір	02.12.2024-10.01.2025	
	даних по темі роботи		
3.	Виконання дослідження згідно теми кваліфікаційної	13.01.2025-14.03.2025	
	роботи		
4.	Оформлення розділу «Аналіз SQL NoSQL та страте»	17.03.2025-28.03.2025	
	гій їх інтеграції для мобільних застосунків»		
5.	Оформлення розділу «Проектування та реалізація	31.03.2025-11.04.2025	
	Гібридної SQL/NoSQL архітектури»		
6.	Оформлення розділу «Наукове обгрунтування, апроба	14.04.2025-25.04.2025	
	ція та перспективи розвитку гібридної моделі SQL/		
	NoSQL для мобільних платформ»		
7.	Виконання завдання до підрозділу «Охорона праці»	28.04.2025-02.05.2025	
8.	Виконання завдання до підрозділу «Безпека в	28.04.2025-02.05.2025	
	надзвичайних ситуаціях»		
9.	Оформлення кваліфікаційної роботи	05.05.2025-09.05.2025	
10.	Нормоконтроль	12.05.2025-15.05.2025	
11.	Перевірка на плагіат	16.05.2025	
12.	Попередній захист кваліфікаційної роботи	19.05.2025	
13.	Захист кваліфікаційної роботи	30.05.2025	

Студент _____ Шимків В. В.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Никитюк В. В.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Аналіз та реалізація стратегій об`єднання SQL і NoSQL баз даних для сучасних додатків// Кваліфікаційна робота освітнього рівня «Магістр» // Шимківа Владислава Сергійовича // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра комп'ютерних наук, група СНнм-61 // Тернопіль, 2025 // С. 79, рис. – 18, табл. – 8, додат. – 3, кресл. – 13, бібліогр. – 73.

Ключові слова: бази даних, sql, nosql, гібридна архітектура, додатки, офлайн-режим, синхронізація даних, зберігання даних.

Кваліфікаційна робота присвячена аналізу та реалізації стратегій об'єднання SQL та NoSQL баз даних для сучасних мобільних додатків, з акцентом на забезпеченні офлайн-функціональності та хмарної синхронізації.

Перший розділ аналізує проблематику зберігання даних в мобільних додатках. Розглянуто особливості, переваги та недоліки реляційних (SQL) баз даних (SQLite/Room) та NoSQL баз даних (Firebase Firestore). Також досліджено існуючі стратегії їх комбінування.

Другий розділ описує розробку та впровадження гібридної архітектури для Android-застосунку. Детально розглянуто реалізацію локального зберігання на базі Room (SQL), хмарного зберігання на базі Firebase Firestore (NoSQL), розроблено стратегію їх поєднання через двосторонню синхронізацію, механізми вирішення конфліктів та інтегровано функціонал резервного копіювання даних на Google Drive.

Третій розділ присвячено науковому обґрунтуванню запропонованої гібридної моделі. Сформульовано наукову проблему та гіпотезу, описано методологію дослідження, представлено наукову новизну та практичну значущість розробленого рішення. Проведено тестування моделі шляхом прототипу мобільного застосунку, проаналізовано експериментальні результати та окреслено можливості подальшого розвитку й наявні обмеження.

ANNOTATION

Analysis and Implementation of Strategies for Combining SQL and NoSQL Databases for Modern Applications // The educational level "Master" qualification work // Shymkiv Vladyslav // Ternopil Ivan Pulyuy National Technical University, Faculty of Computer Information Systems and Software Engineering, Department of Computer Science, SNnm-61 group // Ternopil, 2025 // P. 79, fig. – 18, tables – 8, annexes – 3, posters – 13, ref. – 73.

Key words: databases, sql, nosql, hybrid architecture, applications, offline mode, data synchronization, data storage.

This qualification project focuses on analyzing and implementing strategies for combining SQL and NoSQL databases in modern mobile applications. The emphasis is on providing offline functionality and cloud synchronization.

The first chapter analyzes data storage issues in mobile applications. It considers the features, advantages, and disadvantages of relational (SQL) databases (SQLite/Room) and NoSQL databases (Firebase Firestore). Existing strategies for combining the two types of databases are also investigated.

The second section describes the development and implementation of a hybrid architecture for an Android application. It details the implementation of local storage based on Room (SQL) and cloud storage based on Firebase Firestore (NoSQL), as well as a strategy for combining them through two-way synchronization, conflict resolution mechanisms, and integrated data backup functionality on Google Drive.

The third section is devoted to scientifically substantiating the proposed hybrid model. It formulates the scientific problem and hypothesis, describes the research methodology, and presents the scientific novelty and practical significance of the developed solution. The model is tested using a mobile application prototype. The experimental results are analyzed, and the possibilities for further development and existing limitations are outlined.

ПЕРЕЛІК СКОРОЧЕНЬ І ТЕРМІНІВ

БД – База даних

ЕМВ – Електромагнітне випромінювання

ПЗ – Програмне забезпечення

РБД – Реляційна база даних

РСУБД – Реляційна система управління базами даних

СУБД – Система управління базами даних

ACID (англ. Atomicity, Consistency, Isolation, Durability) – набір властивостей, що гарантують надійність транзакцій у базах даних (Атомарність, Узгодженість, Ізоляція, Довговічність)

API (англ. Application Programming Interface) – інтерфейс прикладного програмування

BASE (англ. Basically Available, Soft state, Eventually consistent) – модель узгодженості даних у розподілених системах (Базова доступність, Нестійкий стан, Кінцева узгодженість)

BSON (англ. Binary JSON) – бінарний формат представлення даних, схожий на JSON

MVVM (англ. Model-View-ViewModel) – архітектурний шаблон проєктування

NoSQL (англ. Not Only SQL) – категорія систем управління базами даних, які не є реляційними або використовують інші моделі даних, крім реляційної

SAR (англ. Specific Absorption Rate) – питомий коефіцієнт поглинання електромагнітної енергії тілом людини

SDK (англ. Software Development Kit) – набір засобів розробки програмного забезпечення

SQL (англ. Structured Query Language) – мова структурованих запитів, що використовується для створення, модифікації та управління даними у реляційних базах даних

ЗМІСТ

ВСТУП

Актуальність теми. Модерне життя неможливе без цифровізації та технологій, зокрема мобільних застосунків, які слугують фактично асистентами для сучасної людини та її рутини. Виробники мобільних додатків щоразу змагаються за увагу свого користувача та намагаються купити його прихильність зручним дизайном, легкістю користування і навіть адаптивністю до настрою юзерів. Мільйони людей відслідковують свій робочий графік, стан здоров'я, вивчають іноземні мови та шукають кохання навіть на іншому кінці планети завдяки застосункам та довіряють їм чутливу інформацію про себе.

Успіх мобільних додатків складається не тільки з маркетингу та інвестицій, один з головних аспектів — ефективне управління даними. Як-от, конфіденційність та унеможливлення витоку персональних даних користувачів, забезпечення роботи в онлайн та офлайн режимах, синхронізація між різними пристроями юзерів (смартфон, ноутбук, телевізор).

Традиційно для управління структурованими даними в інформаційних системах, включно з мобільними платформами, широко застосовуються реляційні бази даних (SQL). Їхня популярність зумовлена здатністю забезпечувати високий рівень цілісності, узгодженості та надійності даних завдяки дотриманню ACID-властивостей (атомарність, узгодженість, ізоляція, довговічність). Технології, такі як SQLite, та їх сучасні обгортки, наприклад, бібліотека Room для платформи Android, є поширеним вибором для організації локального зберігання даних безпосередньо на мобільному пристрої. Однак, незважаючи на численні переваги, суто реляційні рішення можуть зіткнутися із певними обмеженнями, коли йдеться про гнучкість схеми даних, необхідність горизонтального масштабування, ефективність роботи з великими обсягами неструктурованих чи напівструктурованих даних, а також реалізацію простих та надійних механізмів синхронізації з хмарними сервісами.

Паралельно з розвитком реляційних технологій значного поширення набули NoSQL (Not Only SQL) бази даних. Ця категорія передбачає різноманітні моделі зберігання, такі як документоорієнтовані (наприклад, Firebase Firestore,

MongoDB), ключ-значення, стовпчикові та графові. NoSQL-рішення відомі своєю високою гнучкістю схеми, чудовою масштабованістю та продуктивністю для специфічних типів навантажень і моделей даних. Вони часто стають основою для хмарних сховищ, надаючи розробникам зручні інструменти для синхронізації даних між пристроями в реальному часі. Проте, NoSQL-системи не завжди можуть гарантувати такий самий рівень строгої узгодженості, як реляційні аналоги (часто дотримуючись моделі BASE – Basically Available, Soft state, Eventually consistent), і можуть бути менш оптимальними для реалізації складних транзакційних операцій або забезпечення повноцінної автономної роботи в офлайн-режимі без наявності локального SQL-еквівалента.

Очевидно, що ані виключно SQL, ані тільки NoSQL підхід не може цілком задовольнити весь спектр вимог, що висуваються до сучасних мобільних додатків. Користувачі очікують, що їхні додатки будуть надійно працювати незалежно від наявності інтернет-з'єднання, зберігаючи при цьому всі локально внесені зміни, і водночас забезпечуватимуть актуальність даних на всіх підключених пристроях через хмарну синхронізацію. Така подвійна вимога зумовлює високу актуальність дослідження, розробки та практичної реалізації гібридних стратегій, які б дозволили гармонійно поєднувати переваги SQL та NoSQL баз даних. Створення ефективних механізмів інтеграції, надійної синхронізації, алгоритмів вирішення конфліктів та забезпечення загальної консистентності даних у таких гетерогенних системах є складним, але надзвичайно важливим науково-технічним завданням. Недостатня кількість універсальних, науково обґрунтованих підходів до побудови стійких та гнучких гібридних архітектур даних для мобільних додатків визначає нагальну необхідність проведення даного дослідження.

Мета і задачі дослідження. Метою даної кваліфікаційної роботи є всебічний аналіз, розробка та практична реалізація ефективних стратегій об’єднання SQL та NoSQL баз даних, адаптованих для специфіки сучасних мобільних додатків. Ключовим завданням є створення такої гібридної моделі, яка б забезпечувала надійність зберігання даних, повноцінну підтримку офлайн-функціональності та безшовну, консистентну синхронізацію з хмарними

сервісами. Для досягнення цієї мети необхідно вирішити низку завдань: провести глибокий аналіз особливостей SQL та NoSQL рішень у мобільному контексті, дослідити існуючі стратегії їх інтеграції, розробити архітектуру мобільного додатку, що втілює гібридний підхід з використанням локальної SQL-бази (Room) та хмарної NoSQL-бази (Firebase Firestore), реалізувати механізми двосторонньої синхронізації та вирішення конфліктів, інтегрувати функціонал резервного копіювання, та провести наукове обґрунтування й апробацію запропонованої моделі.

Об’єктом даного дослідження виступають процеси зберігання, управління та синхронізації даних, що відбуваються в сучасних мобільних додатках.

Предметом дослідження є стратегії, моделі, методи та програмні засоби, спрямовані на ефективне об’єднання локальних SQL-сховищ, на прикладі Room/SQLite, та хмарних NoSQL-сховищ, на прикладі Firebase Firestore, в єдиній архітектурі мобільних додатків.

Наукова новизна одержаних результатів полягає у систематизації та адаптації існуючих стратегій об'єднання баз даних для специфічних вимог мобільних платформ, розробці комплексної гібридної архітектури для Android-додатків, що реалізує клієнтську двосторонню реактивну синхронізацію, удосконаленні механізму синхронізації через впровадження методу повної гібридної синхронізації з автоматичним очищенням хмари, а також у пропозиції уніфікованої моделі даних, сумісної з обома типами сховищ.

Практичне значення отриманих результатів полягає в тому, що вони можуть бути безпосередньо використані розробниками для створення більш надійних, функціональних та зручних мобільних додатків, здатних ефективно працювати в різноманітних умовах, включаючи відсутність стабільного інтернет-з'єднання. Розроблений прототип мобільного додатку слугує наочною демонстрацією практичної реалізації запропонованих підходів.

Апробація результатів магістерської роботи. Основні результати проведених досліджень були апробовані шляхом доповідей на науково-технічних конференціях та обговорень на наукових семінарах, що підтверджує

їхню актуальність та наукову цінність. Ключові аспекти роботи також знайшли своє відображення у відповідних публікаціях, на «XII Міжнародна науково-практична конференції PERSPECTIVES OF CONTEMPO-RARY SCIENCE: THEORY AND PRACTICE 2025», яка проводилась у м. Львівові, Україна.

Публікації. Результати основних аспектів кваліфікаційної роботи опубліковано у двох працях конференції. (Див. Дотатки А).

Структура й обсяг кваліфікаційної роботи. Кваліфікаційна робота складається зі вступу, чотирьох розділів, висновків, списку літератури з найменувань та 3 додатків. Загальний обсяг кваліфікаційної роботи складає 128 сторінки, з них 78 сторінки основного тексту, який містить 18 рисунків та 8 таблиць.

РОЗДІЛ 1. АНАЛІЗ SQL, NoSQL ТА СТРАТЕГІЙ ЇХ ІНТЕГРАЦІЇ ДЛЯ МОБІЛЬНИХ ЗАСТОСУНКІВ

1.1 Реляційні бази даних (SQL): особливості та застовування в мобільній розробці

Реляційні бази даних (РБД) є основною технологією для зберігання та обробки структурованих даних у сучасних інформаційних системах. Вони організовують дані у вигляді таблиць із чітко визначеною структурою: рядки представляють кортежі, а стовпці - атрибути. Важливою особливістю РБД є використання первинних і зовнішніх ключів для забезпечення логічної цілісності між таблицями. Такі архітектурні особливості дозволяють ефективно структурувати великі обсяги даних і забезпечити їхню узгодженість. У контексті мобільних додатків, локальні РБД, такі як SQLite (та її обгортки на кшталт Room в Android), забезпечують надійне офлайн-зберігання структурованих даних безпосередньо на пристрої користувача.

Одним із найсучасніших джерел для аналізу схем реляційних баз даних є відкритий корпус SchemaPile, який містить понад 220 тисяч реальних схем, витягнутих із публічних репозиторіїв GitHub. Це дослідження демонструє значне різноманіття структур, типів даних, зв’язків між таблицями, обмежень цілісності та індексів, що дозволяє відтворити реальні сценарії використання SQL-баз у промислових та наукових умовах [1].

Забезпечення ефективності РБД досягається за допомогою низки методів. Нормалізація є основною процедурою усунення надлишковості даних шляхом розбиття таблиць на атомарні форми та встановлення функціональних залежностей. Поширене використання форм нормалізації (1NF, 2NF, 3NF) забезпечує логічну цілісність і зменшує потенційні аномалії при оновленні. Індексація дозволяє суттєво підвищити швидкість доступу до даних, а плани виконання запитів (EXPLAIN, ANALYZE) оптимізувати обчислювальні витрати. Центральною характеристикою SQL-СУБД є підтримка транзакцій із властивостями ACID: атомарність, узгодженість, ізоляція та довговічність, що

гарантує стійкість і відновлення після збоїв. Для мобільних застосунків властивості ACID є критично важливими для збереження цілісності даних користувача, особливо при виконанні операцій в офлайн-режимі або при несподіваному закритті додатку.

Реляційні бази даних мають низку переваг, серед яких високий рівень формалізації за рахунок використання мови SQL, можливість реалізації складних запитів із багатьма рівнями об'єднань (JOIN), агрегатів і вкладень. Крім того, вони є зрілими з точки зору інструментарію: існує широкий вибір СУБД, таких як PostgreSQL, Oracle, MySQL, які підтримують розширене профілювання, реплікацію, моніторинг продуктивності й безпеку даних. Особливу роль відіграє точність структурування. Дослідження GitTables виявило, що таблиці у SQL-базах, на відміну від веб-таблиць, мають значно більше стовпців, що свідчить про кращу семантичну насиченість і складність моделювання [2].

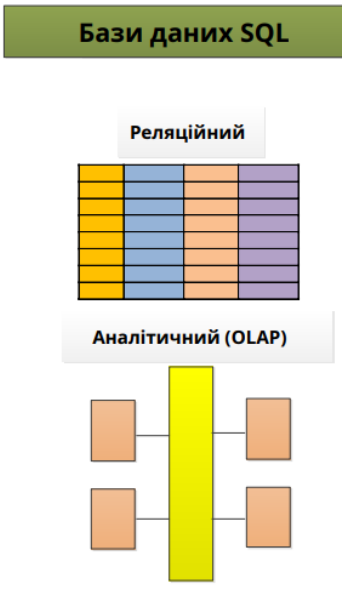

Рисунок 1.1 – Функціональна структура зберігання баз даних SQL [3]

Попри значні переваги, реляційні бази даних мають і низку недоліків. Зокрема, це обмежена горизонтальна масштабованість. Більшість SQL-СУБД орієнтовані на вертикальне масштабування, що ускладнює ефективне використання в умовах розподілених кластерних систем або великих потоків неструктурованих даних. Жорстка схема даних призводить до труднощів під час

адаптації моделей у випадках частих змін структури. У дослідженні [3] зазначено, що реляційні СУБД демонструють зниження продуктивності в умовах обробки «великих даних» порівняно з NoSQL-рішеннями. Це стосується також і хмарних середовищ, де питання сумісності й переносимості даних стають критичними. Для мобільних додатків ці недоліки проявляються інакше: хоча горизонтальне масштабування рідко є проблемою на рівні окремого пристрою, жорстка схема може ускладнювати оновлення додатку з новими функціями, що вимагають змін у структурі локальних даних. Крім того, відсутність вбудованих механізмів синхронізації з хмарою та між пристроями є суттєвим обмеженням для сучасних мобільних сценаріїв, де користувачі очікують безшовного досвіду на всіх своїх пристроях.

Таблиця 1.1 – Переваги та недоліки реляційних баз даних

Переваги	**Недоліки**
Структурованість та ясність	Обмежена масштабованість
Підтримка транзакцій (ACID)	Погана підтримка нефіксованих даних та швидкозмінних
Могутній механізм запитів	Проблеми з обробкою великих обсягів даних
Висока стабільність та надійність	Жорстка схема
Підтримка складних запитів	Низька продуктивність при частих змінах схеми
Широка підтримка та документація	Масштабування горизонтально
Повноцінна робота в офлайн-режимі	Складність реалізації синхронізації з хмарними сервісами без додаткових інструментів

Водночас, реляційні бази залишаються основним вибором у критичних сферах, де важлива точність, контроль доступу, сувора типізація та транзакційна обробка. Застосування охоплює фінансові системи, медичні реєстри, ERP-рішення, CRM, системи обліку та логістики. У навчальному процесі SQL залишається основною мовою запитів до баз даних. Дослідження ChatbotSQL [4], яке оцінювало застосування розмовного агента для підтримки студентів у

написанні SQL-запитів, показало, що навіть при складних запитах типу JOIN, реляційна модель забезпечує логічну цілісність і точність формулювання запитів, що позитивно впливає на результативність навчання. У мобільних додатках SQL-сховища часто використовуються для зберігання налаштувань користувача, локального кешу даних, що завантажуються з мережі, або, як у даному дослідженні, основного сховища даних, що генеруються користувачем і потребують надійності та структурованості.

Отже, реляційні бази даних зберігають свою актуальність завдяки формалізованій архітектурі, потужним аналітичним можливостям і високій стабільності. У контексті інтеграції з інструментами штучного інтелекту та гібридними системами (SQL+NoSQL) вони можуть відігравати ще ширшу роль у майбутньому розвитку інформаційних систем, зокрема, слугуючи надійною локальною основою для мобільних додатків, що потребують синхронізації з більш гнучкими хмарними NoSQL-рішеннями.

1.2 NoSQL бази даних: класифікація, ключові характеристики та їх роль у сучасних мобільних рішень

У сучасних умовах функціонування інформаційних систем, які працюють з великими обсягами даних, що характеризуються високою варіативністю, швидкістю генерації та неструктурованістю, традиційні реляційні системи управління базами даних (РСУБД) втрачають свою ефективність. Це зумовлює необхідність використання альтернативних підходів до зберігання та обробки інформації, що й стало передумовою появи нереляційних баз даних, відомих під загальною назвою NoSQL (Not Only SQL). Вони пропонують нові моделі зберігання, здатні працювати з даними різної природи у режимі реального часу, забезпечуючи гнучку структуру, горизонтальне масштабування та високу відмовостійкість [5; 6]. Для мобільних додатків NoSQL-рішення, особливо хмарні (як Firebase Firestore, MongoDB Atlas), стають дедалі популярнішими завдяки їхній здатності легко синхронізувати дані між пристроями, підтримувати

роботу в реальному часі та забезпечувати гнучкість схеми, що полегшує ітеративну розробку.

Класифікація NoSQL баз даних ґрунтується на відмінностях у способах організації даних та їх структурі. Однією з найпоширеніших категорій є документоорієнтовані бази даних, які зберігають інформацію у форматах JSON, BSON або XML. Такі документи є самодостатніми одиницями даних, що дозволяє уникати жорсткої схеми та реалізовувати атомарні операції на рівні кожного документа. Серед прикладів таких систем можна відзначити MongoDB, Couchbase та Firestore [5; 7]. Саме документоорієнтовані бази, як Firebase Firestore, часто обирають для мобільних додатків через зручність роботи з об'єктами даних, що легко мапляться на об'єкти в коді додатку, та вбудовані механізми офлайн-підтримки (хоча й обмеженої порівняно з повноцінною локальною SQL-базою) та синхронізації. Іншим популярним типом є бази даних формату «ключ–значення», що реалізують найпростішу модель асоціативного зберігання. Завдяки високій швидкодії та простоті ці рішення ефективні у задачах кешування, сесійного зберігання та обслуговування конфігураційних параметрів. До цієї категорії належать Redis, Riak і DynamoDB [6; 7].

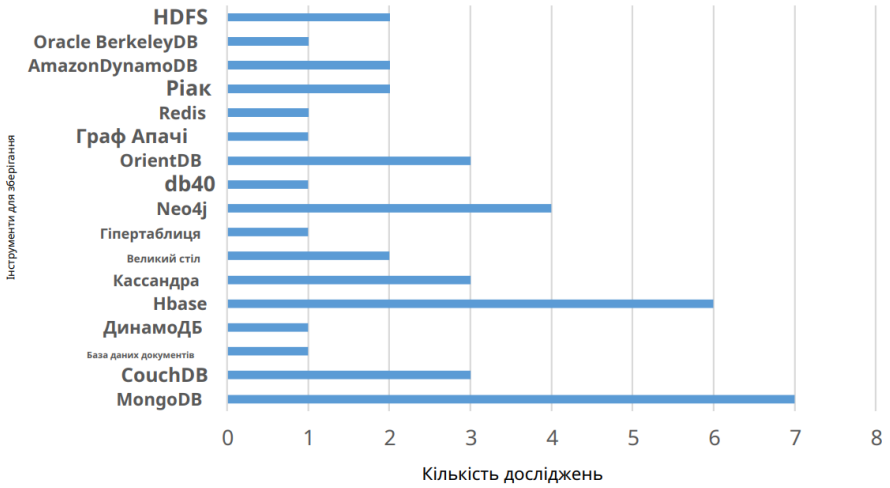

Рисунок 1.2 – Гістограма частоти використання різних типів NoSQL СУБД у наукових дослідженнях (ключ-значення, документи, графи, колонки) [5].

Ще одним типом є стовпчикові бази даних, що організовують зберігання даних у формі розріджених матриць зі згрупованими колонками (column

families). Такий підхід дозволяє забезпечити високу ефективність при виконанні аналітичних запитів, обробці великих потоків даних та роботі з часовими рядами. Представниками цієї категорії є Apache Cassandra, HBase та Bigtable [8]. Для задач, пов'язаних із представленням складних зв'язків, використовуються графові бази даних. Вони моделюють інформацію у вигляді графів, що складаються з вузлів і ребер, кожне з яких може містити властивості. Такі СУБД ефективно реалізують запити на зв'язність, пошук шляхів та шаблонів у складних структурах і широко застосовуються в соціальних мережах, рекомендаційних системах та системах виявлення шахрайства [5; 6]. Хоча стовпчикові та графові бази даних менш поширені як основне сховище для клієнтської частини мобільних додатків, вони можуть використовуватись на бекенді для аналітики або специфічних функцій.

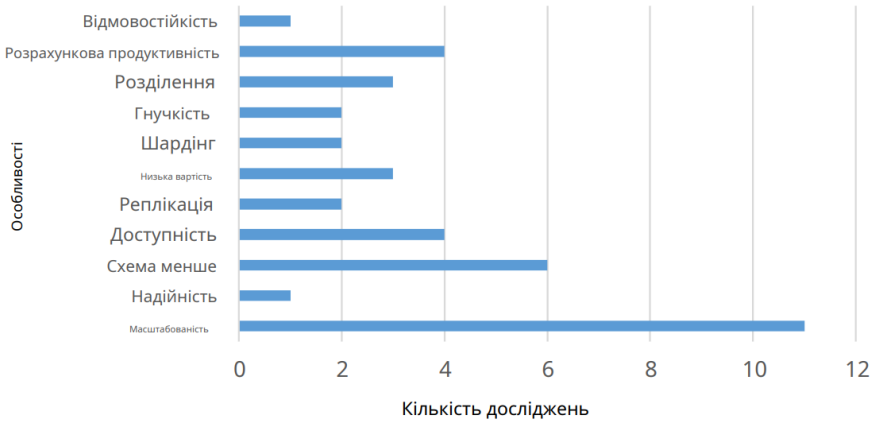

Рисунок 1.3 – Діаграма з такими характеристиками, як реплікація, масштабованість, шардінг, низька вартість, відмовостійкість [5].

З архітектурного погляду, більшість NoSQL баз даних реалізовані як розподілені системи, орієнтовані на горизонтальне масштабування та забезпечення високої доступності. Центральною концепцією таких систем є CAP-теорема, яка встановлює обмеження одночасної реалізації узгодженості, доступності та стійкості до розділення мережі. Залежно від обраного компромісу між цими характеристиками, системи можуть віддавати перевагу сценаріям CA, CP або AP [5]. Крім того, на відміну від ACID-властивостей, притаманних реляційним СУБД, багато NoSQL рішень дотримуються принципів BASE:

базової доступності, м’якого стану системи та кінцевої узгодженості. Це дозволяє досягти високої гнучкості та масштабованості, що є критично важливим для роботи з великими обсягами неоднорідних даних у розподілених середовищах [5; 7]. У мобільному контексті, модель кінцевої узгодженості (BASE) часто є прийнятним компромісом для забезпечення високої доступності та швидкої синхронізації, хоча й вимагає ретельного проєктування механізмів вирішення конфліктів, якщо дані можуть одночасно змінюватися на різних пристроях або в офлайні.

Таблиця 1.2 – Типів NoSQL баз даних

Тип	**Основні приклади**	**Формат**	**Сильні сторони**	**Застосування в мобільних (клієнт/сервер)**
Документоорієнтовані	MangoDB, Firestore, Couchbase	JSON, BSON, XML	Гнучка структура, атомарність документів, вторинна індексація	Клієнт (Firestore, Couchbase Lite), Сервер
Ключ-значення	Redis, Riak, DynamoDB	Будь-яке значення (рядок, об`єкт)	Висока швидкодія, простота, масштабованість	Сервер (кешування, сесії), Клієнт (рідше)
Стовчикові	Apache Cassandra, HBase,	Стовпці в column families	Велика пропускна здатність, ефективна обробка часових рядів	Сервер (аналітика, великі дані)
Графові	Neo4j, Amazon Neptune, ArangoDB	Вузли та ребра з властивостями	Оптимальні для складних зв’язків,	Сервер (соціальні зв'язки, рекомендації)

			ефективність обходу графів	

Застосування NoSQL баз даних NoSQL технології знайшли широке застосування в системах, де ключовими є обсяги даних, швидкість обробки або гнучкість структури:

- освітні платформи та аналітика навчальних даних [7];
- соціальні мережі, системи рекомендацій, платформи контенту;
- системи моніторингу, IoT, обробка телеметрії;
- e-commerce та динамічне керування каталогами;
- хмарні сервіси з високим навантаженням [8].

Для мобільних додатків NoSQL особливо цінні для: забезпечення синхронізації даних у реальному часі між кількома пристроями одного користувача, побудови колаборативних функцій, швидкого прототипування завдяки гнучкій схемі, та підтримки офлайн-доступу з подальшою синхронізацією (хоча часто з певними обмеженнями).

Таким чином, вибір конкретного типу NoSQL бази даних має ґрунтуватися на ретельному аналізі вимог до структури даних, шаблонів запитів, продуктивності та очікуваного масштабування, а в контексті мобільних додатків – також на легкості інтеграції з клієнтською платформою, наявності SDK, підтримки офлайн-режиму та моделі синхронізації.

1.3 Огляд та аналіз стратегій об`єднання SQL та NoSQL технологій

Зі зростанням обсягів, різноманітності та швидкості даних у сучасних інформаційних системах виникає потреба у таких підходах до управління даними, які виходять за межі можливостей традиційних реляційних СУБД. У цьому контексті стратегія поєднання SQL та NoSQL технологій стала ефективним вирішенням проблем масштабованості, гнучкості та транзакційної узгодженості [1; 10]. Гібридні архітектури, що поєднують SQL та NoSQL, дозволяють організаціям зберігати структуровані дані (наприклад, фінансові або

транзакційні) у реляційних СУБД, таких як MySQL або PostgreSQL, тоді як для зберігання неструктурованих чи напівструктурованих даних (журнали, відгуки, сесії користувачів) використовуються NoSQL-рішення MongoDB, Cassandra, Redis тощо [9]. У мобільній розробці ця потреба є особливо гострою: надійне локальне зберігання з ACID-гарантіями (SQL) часто необхідно поєднати з гнучкістю, масштабованістю та можливостями синхронізації хмарних NoSQL-сервісів для забезпечення безперервного користувацького досвіду незалежно від наявності мережі та кількості пристроїв. Таблиця 1.3 – Задачі, у яких застосовують об’єднання SQL та NoSQL (Див. Додаток В).

Однією з ключових переваг гібридного підходу є можливість оптимального використання сильних сторін обох типів баз даних: структурованість і узгодженість SQL поєднується з масштабованістю і швидкістю NoSQL. Це забезпечує підвищену надійність, масштабованість і адаптивність системи [12]. У таких системах SQL-компонент відповідає за складні транзакції та агрегацію, тоді як NoSQL за обробку масивів подій, кешування або сховище документів [13]. У мобільній архітектурі, як правило, локальна SQL-база (наприклад, Room/SQLite) забезпечує атомарність операцій в офлайні та швидкий доступ до структурованих даних, а хмарна NoSQL-база (наприклад, Firebase Firestore) відповідає за синхронізацію, гнучкість даних та доступ у реальному часі з різних пристроїв.

Упровадження гібридної СУБД також створює виклики. Одним з них є розробка уніфікованої схеми даних або моделі, яка дозволяє одночасну підтримку реляційного та нереляційного зберігання. Цьому сприяють такі фреймворки, як Mortadelo, які дають змогу перетворити єдину логічну модель у специфічні моделі для кількох типів NoSQL СУБД [14]. Додатково використовуються дві класичні стратегії міграції Big Bang та Trickle. У першому випадку відбувається одномоментний перехід на нову архітектуру, тоді як другий передбачає поступову інтеграцію окремих модулів системи [14]. Для мобільних додатків, які часто розробляються ітеративно, підхід "Trickle" (поступове впровадження синхронізації або нових типів даних) може бути більш доцільним, ніж "Big Bang". Проблема уніфікації моделі даних між локальним

SQL та хмарним NoSQL є однією з ключових, яку має вирішувати розробник, як це зроблено в даному дослідженні.

У межах поліглотного підходу сформувалися кілька ключових стратегій об’єднання:

- використання NoSQL як кешу для SQL підхід, за якого NoSQL (наприклад, Redis) виступає як шар кешування, що знижує навантаження на реляційну СУБД. Підвищує продуктивність у системах із великою кількістю читань [13]. У мобільному контексті це може бути хмарний NoSQL-кеш для даних, що часто запитуються з серверного SQL, або навпаки, локальний SQL-кеш для даних з хмарного NoSQL.
- CQRS (Command Query Responsibility Segregation) розділення моделі читання (NoSQL) та запису (SQL), що дозволяє незалежно масштабувати ці частини системи. Застосовується в мікросервісних архітектурах, high-load системах [9; 13]. Для мобільних додатків це може означати, що локальні зміни (команди) йдуть в SQL, а читання даних для відображення (запити) може відбуватися як з локального SQL, так і з синхронізованого хмарного NoSQL.
- зберігання різних типів даних у відповідних сховищах SQL використовується для транзакцій, NoSQL для гнучкої роботи з JSON, логами або графовими структурами. Використовується в e-commerce, IoT, CMS [10; 12]. Це найпоширеніший підхід для гібридних мобільних додатків: структуровані дані – локально в SQL, а дані для синхронізації або з гнучкою структурою – в хмарному NoSQL.
- Middleware (API-шар) централізоване програмне забезпечення, яке перенаправляє запити до відповідної бази. Дає можливість створення уніфікованого API [13]. У мобільній розробці роль такого шару часто виконує репозиторій або сервісний шар усередині самого додатку, який абстрагує логіку взаємодії з локальним та хмарним сховищами, як це реалізовано в даній роботі.
- Polyglot Persistence архітектурна стратегія, при якій кожен модуль системи використовує оптимальну СУБД. Наприклад, платежі обробляються в SQL, а сесії зберігаються в NoSQL [9; 12].

- Data Synchronization Services (ETL, CDC) забезпечують узгодженість між двома базами даних. Реалізується за допомогою потокової реплікації або пакетного оновлення [13; 14]. Для мобільних додатків механізми синхронізації, вбудовані в хмарні платформи (як Firestore), або кастомні рішення на основі часових міток, є ключовими для підтримки узгодженості між локальним SQL та хмарним NoSQL.
- Federated Query Engines інструменти типу Presto або Trino, що дозволяють виконувати єдиний SQL-запит одночасно до SQL і NoSQL систем [12]. Ця стратегія менш типова для клієнтської частини мобільних додатків через складність та ресурсоємність.
- мультимодельні СУБД бази, які поєднують у собі кілька моделей зберігання (PostgreSQL, ArangoDB), що дозволяє уникнути складної синхронізації [14]. Деякі локальні мобільні БД (наприклад, Realm) пропонують мультимодельні можливості та синхронізацію, але часто є пропрієтарними. Таблиця 1.4 – Порівняння стратегій об’єднання SQL та NoSQL (Див. Додаток Д)

Кожна стратегія має власні переваги та обмеження, які слід оцінювати з урахуванням конкретного проекту, характеру даних, вимог до масштабування, узгодженості й продуктивності. Грамотно спроєктована гібридна архітектура дозволяє підвищити ефективність систем, мінімізувати затримки, покращити масштабованість і задовольнити різні сценарії доступу до даних. Для мобільних додатків особливо важливим є вибір стратегії, що забезпечує надійну офлайн-роботу, прозору для користувача синхронізацію та мінімальне споживання ресурсів пристрою, що і є предметом даного дослідження.

1.4 Висновок до першого розділу

Застосування гібридних архітектур, що об'єднують SQL та NoSQL бази даних, є ефективною відповіддю на виклики, зумовлені зростанням обсягів, різноманітності та швидкості даних у сучасних інформаційних системах, а також специфічними вимогами мобільних додатків до офлайн функціональності та багатопристроєвої синхронізації. Проаналізовані підходи, такі як поліглотне

зберігання, CQRS, middleware-рішення (реалізовані на рівні клієнта), федеративні рушії запитів, механізми синхронізації та мультимодельні СУБД, демонструють широкий спектр можливостей для забезпечення гнучкості, масштабованості та продуктивності.

Кожна стратегія має свої переваги та обмеження, які необхідно враховувати під час архітектурного проєктування. Зокрема, комбінування локального SQL-сховища на мобільному пристрої та хмарної NoSQL-бази дозволяє ефективно розділити навантаження між надійним офлайн-зберіганням даних з гарантіями ACID та гнучким, масштабованим хмарним зберіганням з можливостями синхронізації в реальному часі. Це забезпечує швидкий доступ до сесійних чи поведінкових даних користувачів, а також зберегти високий рівень цілісності критичних бізнес-даних.

Таким чином, вибір тієї чи іншої стратегії об’єднання повинен ґрунтуватися на комплексному аналізі характеристик даних, функціональних вимог до системи, специфіки навантаження (як на клієнті, так і на сервері), вимог до офлайн-роботи та синхронізації, а також можливостей команди розробників. В умовах динамічного розвитку цифрових технологій гібридний підхід до управління даними стає не лише технічно доцільним, а й стратегічно виправданим кроком у напрямку підвищення адаптивності та конкурентоспроможності інформаційних систем, особливо у сфері мобільних додатків, де очікування користувачів щодо безперервності та доступності даних є надзвичайно високими.

РОЗДІЛ 2. ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ ГІБРИДНОЇ SQL/NoSQL АРХІТЕКТУРИ В ANDROID-ЗАСТОСУНКУ

2.1 Архітектурна модель мобільного застосунку з гібридним сховищем

У рамках кваліфікаційної роботи було розроблено Android-додаток, основною метою якого є демонстрація інтеграції реляційної бази даних (Room/SQLite) та хмарної документно-орієнтованої бази Firebase Firestore.

Головна ідея полягає в гібридному зберіганні даних для забезпечення автономної роботи користувача офлайн та синхронізації з хмарою онлайн. Дані, які створює користувач (нотатки, завдання), зберігаються спочатку локально, а потім синхронізуються з хмарою автоматично або вручну.

Архітектура побудована за шаблоном Model-View-ViewModel (MVVM), що дозволяє розділити відповідальність між рівнями:

- View інтерфейс користувача (RecyclerView, кнопки, поля вводу);
- ViewModel координує логіку додатку та життєвий цикл;
- Repository єдиний доступ до даних з Room і Firestore;
- Data Layer реалізує зберігання в SQLite (Room) та Firestore;
- інтеграційні сервіси зокрема резервне копіювання в Google Drive;

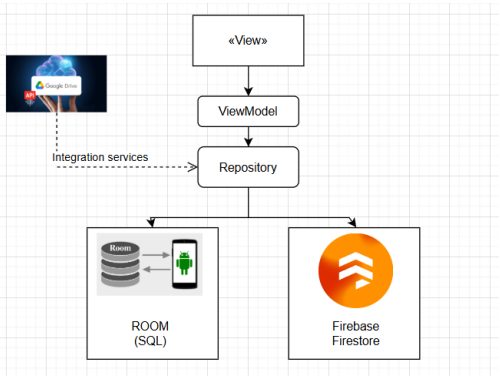

Рисунок 2.1 – Архітектура додатку на основі MVVM з гібридним зберіганням (Room + Firebase)

Взаємодія між компонентами реалізована таким чином, що забез-печується як автономна робота додатку в офлайн-режимі, так і його синхро-нізація з

хмарними джерелами при наявності мережевого підключення. Завдяки цьому досягається гнучкість, масштабованість та надійність три ключові характеристики, що відповідають сучасним вимогам до мобільних додатків з підтримкою складної логіки обробки даних [15].

Крім того, використання MVVM у мобільній розробці довело свою ефективність у проектах, орієнтованих на реактивну архітектуру, особливо завдяки підтримці бібліотек Jetpack Compose та Kotlin [16].

2.2 Локальне SQL сховище на основі Room: реалізація та особливості

У межах реалізації мобільного застосунку локальне зберігання даних було побудовано на основі бібліотеки Room, яка є частиною Android Jetpack і виконує роль зручної абстракції над SQLite [17]. Room забезпечує типобезпечний доступ до бази, підтримку корутин, реактивної обробки змін через LiveData, а також спрощує інтеграцію з іншими компонентами архітектури MVVM.

Основною сутністю для збереження інформації в локальній базі виступає клас Note, що реалізований із використанням анотації @Entity. Цей клас містить такі поля, як унікальний ідентифікатор (id), заголовок (title), основний текст нотатки (content) та мітку часу останнього оновлення (updatedAt). Ідентифікатор генерується автоматично з використанням UUID, що дозволяє уникнути конфліктів під час синхронізації з хмарним сховищем. Мітка часу використовується для визначення актуальності записів і є ключовим елементом логіки гібридної синхронізації.

Доступ до бази даних реалізується через інтерфейс NoteDao, який містить основні CRUD-операції: додавання, оновлення, видалення та вибірку нотаток. Для вибірки використовується як реактивний підхід (через LiveData), так і синхронний у фонових потоках. При збереженні записів застосовується стратегія REPLACE, що дозволяє уникати дублювання під час повторного імпорту з Firestore або кешування даних [18].

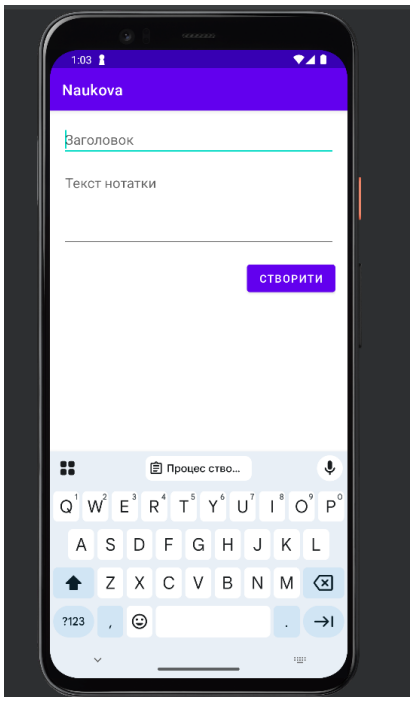

Рисунок 2.2 – Середовище для створення та збереження нотаток

Центральним елементом Room-інфраструктури є клас AppDatabase, який оголошується через анотацію @Database з вказанням списку сутностей та версії схеми. Для забезпечення коректного збереження складних типів (наприклад, Firebase Timestamp) у додатку використовується окремий клас Converters, що містить методи для перетворення нестандартних типів у підтримувані SQLite формати (наприклад, Long). Ці конвертери підклю-чаються через анотацію @TypeConverters.

Крім того, AppDatabase реалізовано відповідно до шаблону Singleton, що дозволяє гарантувати існування лише одного екземпляра бази протягом життєвого циклу додатку. Це не лише забезпечує стабільність доступу до даних, а й дозволяє керувати базою, наприклад, закривати її перед резервним копіюванням.

Процес створення нової нотатки і її збереження в Room реалізовано у вигляді окремої активності NoteCreateActivity.kt, яка надає користувачеві просту форму для введення заголовка та вмісту запису. Після натискання кнопки «Зберегти» створюється об’єкт типу Note, якому автоматично присвоюється унікальний ідентифікатор та поточна мітка часу. Цей об’єкт передається у ViewModel, яка викликає метод insertNote() з репозиторію.

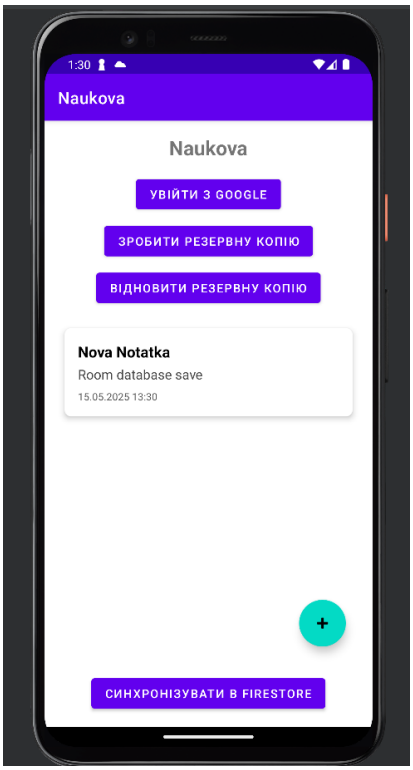

Рисунок 2.3 – Екран створення нової нотатки. Після натискання кнопки «Зберегти»

На цьому етапі нотатка зберігається у локальній базі через DAO, після чого відображається у загальному списку. Цей процес є ключовим етапом у роботі з Room саме він ініціює формування нових записів для подальшої синхронізації з Firestore. [19]

Таким чином, Room виконує роль локального сховища в гібридній архітектурі. Його застосування дозволяє забезпечити швидкий доступ до даних у режимі офлайн, підтримувати актуальність інформації за допомогою реактивного підходу, а також виконувати синхронізацію із хмарною Firestore без втрати консистентності. Завдяки цьому Room виступає надійним і продуктивним інструментом для реалізації локального рівня зберігання в мобільному додатку.

2.3 Хмарне NoSQL-сховище Firebase Firestore: інтеграція та особливості

Хмарну складову гібридної архітектури реалізовано з використанням Firebase Firestore документно-орієнтованої NoSQL-системи, яка надає

можливість зберігати, оновлювати та синхронізувати дані в режимі реального часу. Основною причиною вибору саме Firestore є його тісна інтеграція з Android, підтримка реактивної моделі даних та масштабованість для багатопристроєвого доступу [20].

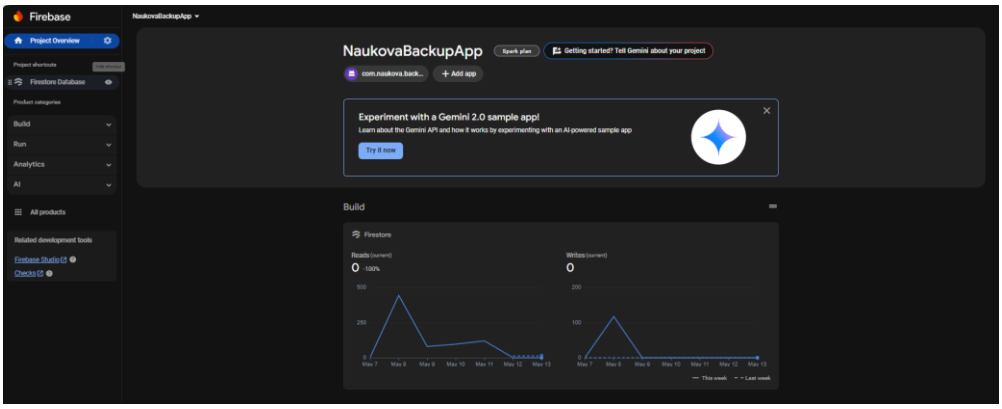

Рисунок 2.4 – Головна панель Firebase-проєкту

У межах проєкту дані користувача (зокрема нотатки) зберігаються у вигляді окремих документів у колекції "notes", де кожен документ відповідає одному об’єкту типу Note. Ідентифікатором документа слугує унікальне поле id, що попередньо генерується локально у Room. Такий підхід забезпечує узгодженість ідентифікаторів і спрощує двосторонню синхронізацію між сховищами.

Firestore не вимагає жорстко фіксованої схеми даних, але для коректної десеріалізації в Kotlin застосовується універсальна модель Note, яка має повну структурну сумісність із JSON-поданням документа. Усі ключові поля title, content, updatedAt зберігаються безпосередньо в документі Firestore. Для забезпечення стійкості при зчитуванні з хмари використовується анотація @IgnoreExtraProperties, яка дозволяє ігнорувати сторонні або нові поля в документі без виникнення помилок.

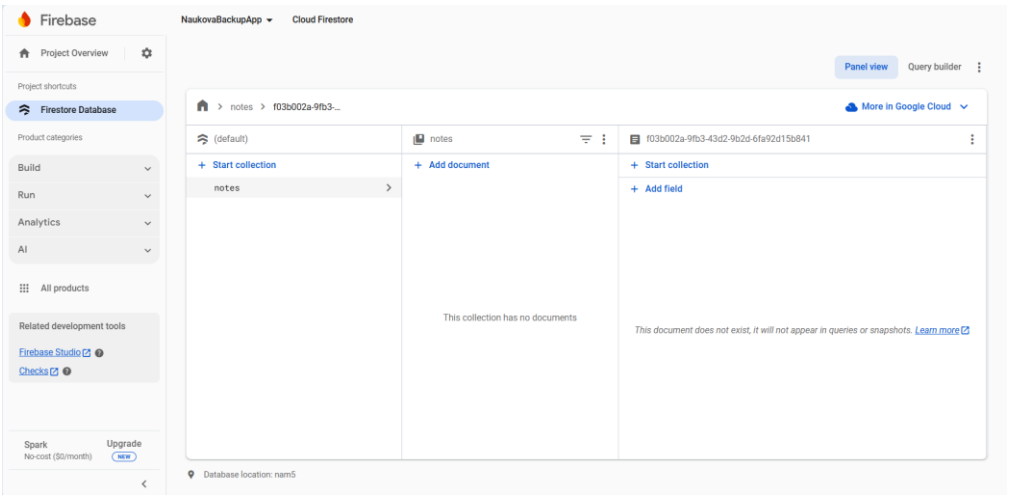

Рисунок 2.5 – Представлення пустої колекції notes у Firebase Firestore

Імпорт даних з Firestore до локального сховища реалізується через спеціальний метод синхронізації, що прослуховує зміни у колекції в реальному часі за допомогою механізму addSnapshotListener. Кожне оновлення автоматично порівнюється з локальною копією: якщо версія документа в хмарі є новішою (визначається за полем updatedAt), то локальна версія оновлюється. Це дозволяє реалізувати ефективну логіку вирішення конфліктів за принципом "останнє змінене актуальне" [21].

Зворотній потік синхронізації з локальної бази в Firestore реалізовано через спостереження за змінами у Room. При додаванні, редагуванні або видаленні нотатки відповідна зміна автоматично дублюється у Firestore, завдяки використанню Kotlin-корутин і підписки на LiveData. Таким чином досягається повна двостороння синхронізація, без потреби ручного втручання з боку користувача.

Крім того, реалізовано механізм очищення хмарної бази від неактуальних записів.

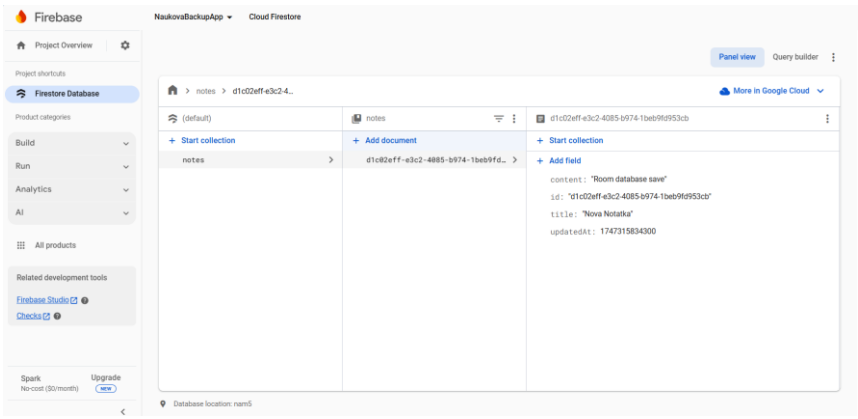

Рисунок 2.6 – Представлення колекції notes у Firebase Firestore

У випадках, коли нотатка видалена локально, спеціальний метод перевіряє її наявність у Firestore і, якщо вона все ще існує видаляє її з хмари. Це дозволяє підтримувати цілісність бази та запобігає накопиченню застарілих даних [22].

Таким чином, Firebase Firestore виконує роль надійного і гнучкого хмарного сховища, яке забезпечує синхронізацію, доступність і масштабованість даних. У поєднанні з Room воно формує гібридну модель зберігання, здатну працювати як у режимі офлайн, так і онлайн, без втрати узгодженості або актуальності інформації.

2.4 Розробка механізмів гібридної сихронізації та вирішення конфліктів

У межах реалізованого проєкту запроваджено гібридну модель зберігання даних, що поєднує реляційну локальну базу Room (SQL) із хмарним документно-орієнтованим сховищем Firebase Firestore (NoSQL). Такий підхід забезпечує баланс між продуктивністю, автономністю та синхронізацією, дозволяючи додатку працювати як у офлайн-, так і онлайн-режимах із мінімальними затримками та без втрати консистентності даних [22]

Ключовою особливістю стратегії є двостороння синхронізація між Room і Firestore. При першому запуску застосунку виконується ініціалізація сховищ і виклик методу syncFromFirestoreToRoom(), який здійснює імпорт актуальних даних з хмари у локальну базу. Зміни в Firestore відслідковуються в режимі

реального часу завдяки механізму addSnapshotListener. Кожен документ, що надходить, порівнюється з локальним записом за допомогою поля updatedAt, яке містить мітку часу останньої модифікації [23]. Якщо хмарна версія є новішою, запис у Room оновлюється автоматично.

З метою підвищення прозорості для кінцевого користувача в додатку реалізовано механізм зворотного зв'язку про успішне збереження даних. Після створення нотатки, застосунок відображає повідомлення на екрані з текстом «Нотатку створено», що сигналізує про успішний запис у локальну базу Room. При ініціації синхронізації в Firestore з'являється інше повідомлення «Синхронізація в Firestore запущена», яке підтверджує початок передачі даних у хмару. Ці повідомлення реалізовані у вигляді Snackbar і є частиною UX-стратегії для забезпечення довіри користувача до стабільності збереження інформації.

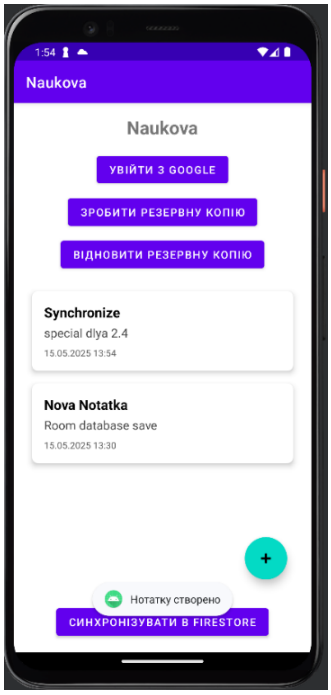

Рисунок 2.7 – Головний екран мобільного застосунку зі списком нотаток.

У зворотному напрямку синхронізація реалізована через метод syncRoomToFirestore(), який спостерігає за LiveData списком локальних нотаток. Кожна зміна (додавання, редагування або видалення) автоматично тригерить відповідну операцію в Firestore. Таким чином, будь-яке оновлення в Room

дублюється у Firestore, зберігаючи узгодженість між базами без потреби в ручному втручанні.

Окремим кроком є очищення хмари від неактуальних записів, що здійснюється через метод deleteRemoteNotesNotInLocal(). Він порівнює список локальних id з ідентифікаторами документів у Firestore. У разі виявлення документів, яких більше не існує у Room, вони видаляються з хмари. Це дозволяє підтримувати чистоту структури даних і запобігати накопиченню застарілих об’єктів.

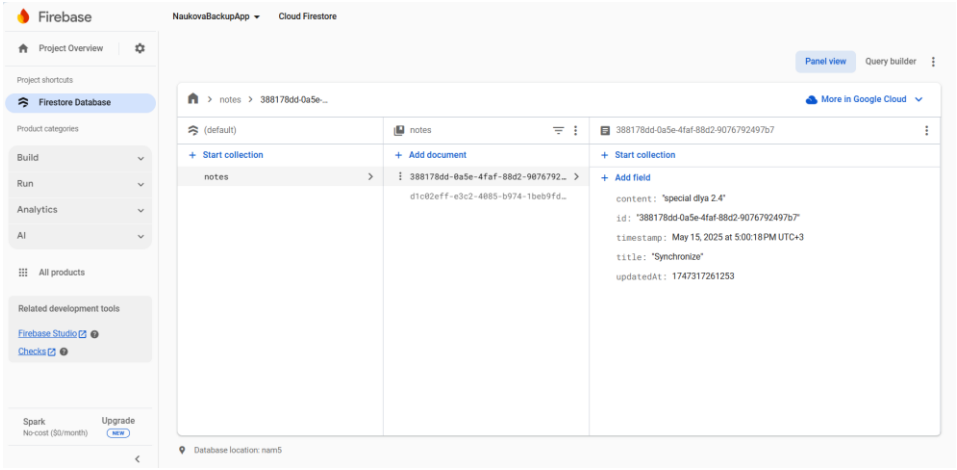

Рисунок 2.8 – Вміст колекції notes у Firebase Firestore.

Для забезпечення наскрізної узгодженості було реалізовано метод fullSync(), який виконує повний цикл синхронізації: імпорт з хмари, експорт до хмари та очищення Firestore. Цей метод може бути викликаний як автоматично (наприклад, при запуску застосунку), так і вручну (перед резервним копіюванням чи оновленням пристрою).

Центральну роль у реалізації цієї стратегії відіграє уніфікована модель Note, що сумісна як із Room, так і з Firestore. Анотація @Entity забезпечує інтеграцію з Room, а @IgnoreExtraProperties гарантує стійкість при десеріалізації з Firestore, навіть якщо структура документа змінюється. Це дозволяє використовувати єдину модель даних без дублювання логіки або адаптерів.

Завдяки підтримці реактивності на обох рівнях LiveData у Room і addSnapshotListener у Firestore досягається ефект постійної актуальності: будь-яка зміна, де б вона не відбулася, відображається у застосунку майже миттєво

[24]. Це суттєво підвищує зручність користування і надає відчуття безперервності навіть у складних умовах перемикання мережі.

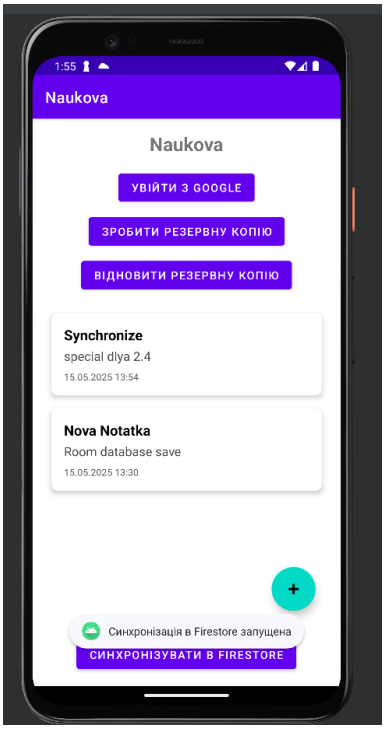

Рисунок 2.9 – Підтвердження початку синхронізації у Firestore.

У підсумку, реалізована гібридна стратегія зберігання даних забезпечує:

- автономність роботи у офлайн-режимі;
- синхронізацію в реальному часі між пристроями;
- гнучкість і масштабованість архітектури;
- стійкість до конфліктів і втрати з’єднання;

Ця модель є основою всього функціоналу застосунку й створює фундамент для реалізації додаткових сервісів, таких як резервне копіювання, багато-пристроєвий доступ та інтеграція з іншими платформами.

2.5 Забезпечення цілісності даних резервного копіювання даних через Google Drive

Одним із ключових компонентів стійкої роботи мобільного застосунку є наявність механізму резервного копіювання, який забезпечує збереження даних

у випадках втрати пристрою, перевстановлення додатку чи інших непередбачуваних ситуацій. У межах реалізації гібридної архітектури передбачено інтеграцію з Google Drive, яка дозволяє створювати резервні копії локальної бази даних Room та зберігати їх у хмарному сховищі користувача.

Перед використанням функції резервного копіювання користувач має пройти авторизацію через Google-акаунт. Для цього застосунок використовує GoogleSignInClient із запитом на обмежений доступ до Google Drive (DRIVE_FILE). Після успішної авторизації створюється екземпляр клієнта Drive API, який дозволяє здійснювати операції лише з тими файлами, які були створені самим додатком. Це забезпечує як безпеку доступу, так і відповідність політиці Google щодо роботи з хмарними сервісами. Авторизація виконується один раз і зберігається локально до моменту відкликання дозволу або зміни акаунта [25].

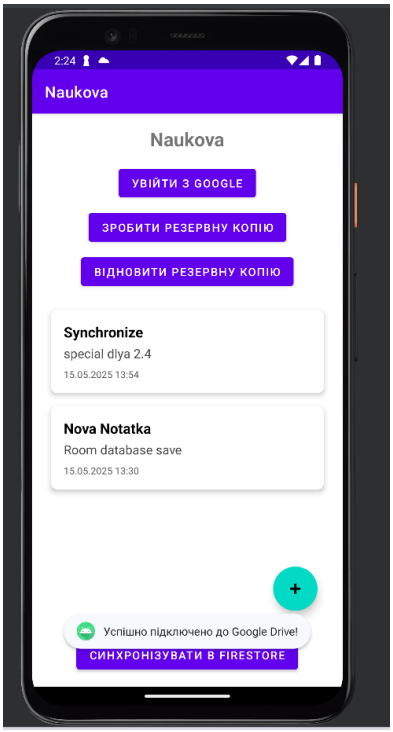

Рисунок 2.10 – Підтвердження успішної авторизації користувача в Google.

Резервне копіювання реалізовано на основі офіційного API Google Drive, з використанням компонентів GoogleSignInClient для автентифікації та Drive API для завантаження й завантаження файлів. Після входу в Google-акаунт застосунок отримує дозволи лише на доступ до власних файлів (обмеження

DRIVE_FILE), що відповідає принципам мінімального привілею та гарантує безпеку.

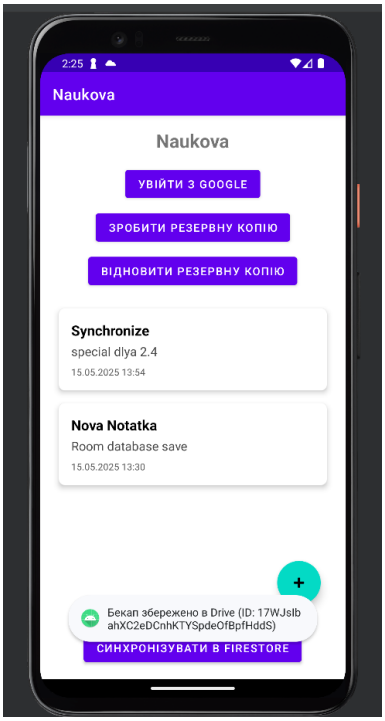

Рисунок 2.11 – Повідомлення про успішне створення резервної копії бази даних Room.

Процедура створення резервної копії ініціюється через кнопку «Зробити резервну копію». Перед безпосереднім копіюванням викликається метод fullSync(), який гарантує, що усі локальні зміни були синхронізовані з Firestore, а хмарні імпортовані у Room. [26] Це дозволяє зафіксувати єдиний актуальний стан даних на момент створення резервної копії.

Далі виконується закриття бази методом AppDatabase.closeDatabase(), оскільки файл SQLite (note_database) не може бути скопійований, поки база відкрита. Після цього файл архівується (опційно у ZIP) і передається на Google Drive з використанням MIME-типу application/octet-stream. Користувач отримує візуальне підтвердження про успішне завершення процесу.

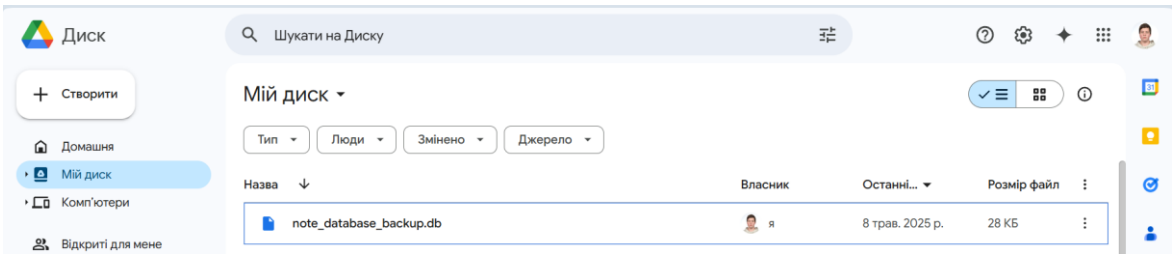

Рисунок 2.12 – Вміст Google Drive користувача, де збережено резервну копію бази даних

Процес відновлення виконується в зворотному порядку. Користувач натискає кнопку «Відновити резервну копію», після чого застосунок отримує доступ до збереженого файлу, завантажує його у внутрішнє сховище, розпаковує (якщо ZIP) та замінює існуючий файл Room-бази. Потім база повторно ініціалізується через getDatabase() і запускається оновлення інтерфейсу через LiveData, завдяки чому користувач одразу бачить відновлені дані у застосунку.

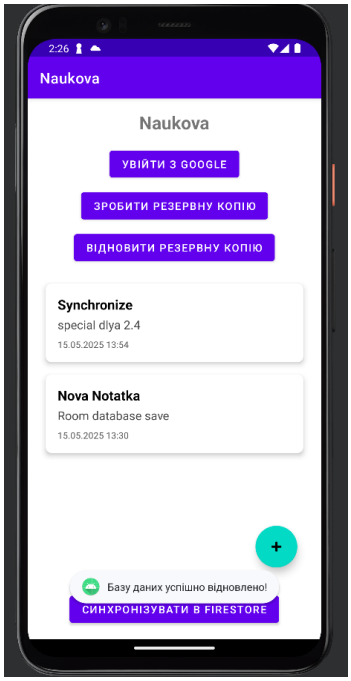

Рисунок 2.13 – Повідомлення про успішне завершення процесу відновлення резервної копії.

Особливістю реалізації є можливість виконання резервного копіювання навіть без активного підключення до Firestore. Завдяки локальному кешу в Room, користувач має змогу створити копію, яка згодом буде синхронізована з хмарою

після відновлення з’єднання. Це забезпечує високу гнучкість та стійкість системи у різних сценаріях використання. [27]

Інтерфейс реалізовано з урахуванням зручності користувача: передбачено індикатори завантаження (ProgressBar), анімацію на кнопках та повідомлення про результат операції (Snackbar) [28]. Уся логіка виконується асинхронно за допомогою Kotlin-корутин, що дозволяє уникати блокування основного потоку.

Таким чином, резервне копіювання до Google Drive виступає доповненням до гібридної стратегії зберігання, дозволяючи користувачам захистити свої дані, переносити їх між пристроями та повертатися до останньо збереженого стану незалежно від наявності синхронізації з Firestore. Такий підхід відповідає сучасним вимогам до надійності, безпеки та автономності мобільних застосунків.

2.6 Висновок до другого розділу

У даному розділі було детально предста-влено практичну реалізацію гібридної архітектури мобільного застосунку, що поєднує локальне реляційне зберігання (Room/SQLite) та хмарну документно-орієнтовану базу даних (Firebase Firestore). Такий підхід дозволяє забезпечити автономну роботу додатку, швидкий доступ до даних в офлайн-режимі та синхронізацію з хмарою при відновленні мережевого з’єднання.

Головним елементом інтеграції є уніфікована модель даних Note, що використовується як у Room, так і у Firestore. Логіка синхронізації реалізована у вигляді окремого репозиторію, який забезпечує двосторонній обмін даними з пріоритетом за полем updatedAt, що дозволяє автоматично вирішувати конфлікти за принципом «останнє змінене актуальне». Реалізовано повну синхронізацію, що включає імпорт, експорт і очищення неактуальних записів у хмарі.

Окрему увагу приділено реалізації механізму резервного копіювання, який дозволяє зберігати актуальну копію бази даних у Google Drive. Перед копіюванням автоматично виконується синхронізація з Firestore, після чого база

архівується та зберігається у хмарі користувача. У разі потреби передбачено можливість відновлення даних із резервної копії, що підвищує стійкість системи до збоїв.

Результатом реалізації стала стабільна, гнучка та масштабована система зберігання, що поєднує переваги локального й хмарного зберігання даних. Такий підхід дозволяє ефективно вирішити задачі автономної роботи, синхронізації та збереження інформації, що особливо важливо для мобільних застосунків із високими вимогами до доступності й надійності.

РОЗДІЛ 3. НАУКОВЕ ОБГРУНТУВАННЯ, АПРОБАЦІЯ ТА ПЕРСПЕКТИВИ РОЗВИТКУ ГІРИДНОЇ МОДЕЛІ SQL/NoSQL ДЛЯ МОБІЛЬНИХ ПЛАТФОРМ

3.1 Формулювання наукової проблеми та завдань дослідження

У сучасних інформаційних системах, особливо у сфері мобільних додатків, постає низка викликів, пов’язаних із забезпеченням безперервного доступу до даних, узгодженості між локальним і хмарним зберіганням, а також стійкості до збоїв у мережі [29]. Особливо це актуально в умовах, коли користувачі очікують, що додатки функціонуватимуть безперервно, незалежно від наявності підключення до Інтернету, і водночас зберігатимуть усі зміни, здійснені в офлайн-режимі.

Традиційні реляційні СУБД (SQL) забезпечують високу структурованість, транзакційну узгодженість та стабільність. Натомість NoSQL-рішення пропонують гнучкість, масштабованість і реальний час доступу до даних [30]. Проте в умовах обмеженого ресурсу пристроїв (мобільні телефони), мережевих перебоїв і вимог до збереження даних, окреме застосування кожного з підходів виявляється недостатнім. Саме тому зростає потреба у науково обґрунтованих гібридних стратегіях, що поєднують сильні сторони обох парадигм зберігання [31].

Таблиця 3.1 – Порівняння SQL і NoSQL рішень в контексті мобільних додатків

Критерій	**SQL (Room/SQLite)**	**NoSQL (Firebase)**	**Рішення**
Доступ в офлайн-режимі	Повний	Лімітований	Не прийнятне
Синхронізація між пристроями	Відсуться (без серверу)	Нативна через хмару	Не прийнятне
Робота зі змінними структурами	Жорстка схема	Динамічна схема	Не прийнятне

Продовження Таблиця 3.1 – Порівняння SQL і NoSQL рішень в контексті мобільних додатків

Узгодженість транзакцій	ACID	DASE	Не прийнятне
Реактивна модель	LiveData	SnaphotListener	Прийнятне
Маштабність	Обмежена	Висока	Не прийнятне

Як видно з таблиці, кожен із підходів має критичні обмеження в контексті мобільних додатків, що робить їх окреме використання неприйнятним.

Наукова проблема, яка лежить в основі дослідження, полягає в нестачі універсальних підходів до побудови стійких і гнучких архітектур даних, що поєднують SQL та NoSQL у межах єдиного мобільного застосунку, із гарантованим збереженням консистентності при асинхронній синхронізації.

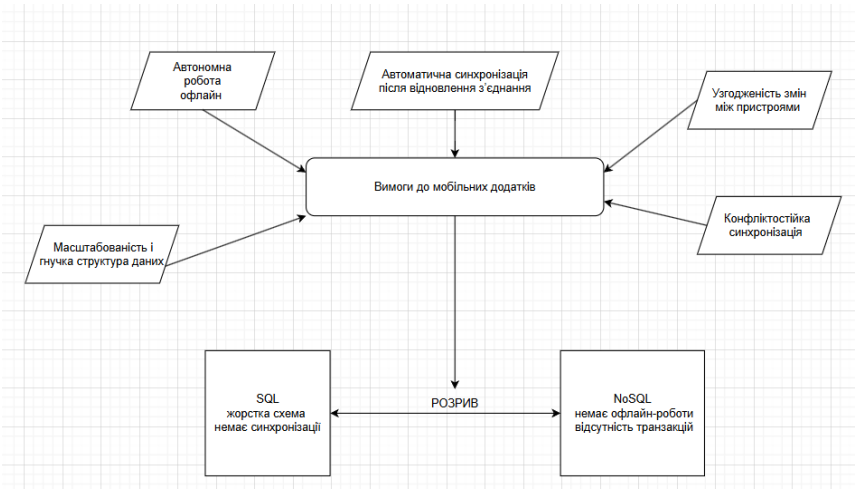

Рисунок 3.1 – Розрив між вимогами до мобільних додатків і можливостями окремих технологій

Метою даного дослідження є розробка, формалізація та наукове обґрунтування гібридної моделі поєднання SQL і NoSQL баз даних для мобільних додатків, яка забезпечує:

- автономну роботу у режимі офлайн;
- узгоджену двосторонню синхронізацію при відновленні мережевого з'єднання;

- підвищену гнучкість і масштабованість без втрати даних.

Для досягнення поставленої мети необхідно вирішити такі завдання:

- Провести аналіз існуючих стратегій об'єднання SQL та NoSQL, виділити їхні обмеження у мобільному середовищі.
- Сформулювати вимоги до гібридної архітектури, яка поєднує локальне (SQL) та хмарне (NoSQL) зберігання.
- Реалізувати прототип системи з підтримкою автоматичної синхронізації даних між сховищами.
- Формалізувати модель гібридного зберігання у вигляді інформаційної схеми з визначенням основних компонентів, точок узгодження та конфліктів.
- Оцінити ефективність розробленої стратегії в порівнянні з альтернативними підходами з точки зору стабільності, часу синхронізації та втрат даних.
- Узагальнити результати та визначити можливості масштабування архітектури на інші категорії додатків.

Таким чином, дослідження спрямоване на вирішення актуальної задачі побудови надійної, консистентної та масштабованої моделі управління даними в мобільних додатках за умов поєднання реляційного та нереляційного підходів.

3.2 Дослідницька гіпотези та обрана методологія перевірки

У рамках даного дослідження висунуто гіпотезу, згідно з якою поєднання локального реляційного сховища (Room/SQLite) та хмарної документо-орієнтованої бази даних (Firebase Firestore) у межах мобільного застосунку дозволяє досягти вищого рівня гнучкості, узгодженості та стійкості до мережевих збоїв порівняно з однотипними підходами [34]. Така гібридна стратегія забезпечує надійне зберігання даних в офлайн-режимі, швидке оновлення при зміні з’єднання та синхронізацію без втрати інформації завдяки реактивному підходу до моніторингу змін.

Таблиця 3.2 – Формулювання гіпотези дослідження

Компонент	**Зміст**
Гіпотеза	Поєднання SQL і NoSQL заберзпечує стабільність, гнучкість та узгодженість
Очікуваний ефект	Робота офлайн, синхронізація, автоматичне оновлення
Обґрунтування	Room для автономності, Firestore для хмарного доступу
Критерії перевірки	Час синхронізації, конфлікти, надійність збереження

Для перевірки цієї гіпотези було застосовано комплексну методологію, яка поєднує аналітичне моделювання, функціональне тестування, порівняльний аналіз і апробацію результатів у реальних умовах експлуатації мобільного застосунку [32].

На першому етапі було здійснено аналітичне моделювання гібридної архітектури, в рамках якого сформовано інформаційну модель зберігання, описано точки синхронізації, механізми виявлення конфліктів та принципи пріоритетного оновлення записів. Побудовано логічні схеми, що відображають основні потоки даних між Room і Firestore, включно з алгоритмами імпорту, експорту та очищення неактуальних записів.

Другий етап передбачав функціональне тестування реалізованого прототипу в різних експлуатаційних умовах. Було змодельовано сценарії роботи з активним з’єднанням, у режимі офлайн та в умовах нестабільного доступу до мережі. Під час тестування вимірювались ключові показники ефективності: час виявлення змін, затримка синхронізації, частота виникнення конфліктів, надійність збереження даних та коректність їх відновлення після втрат або збою підключення.

Для забезпечення об'єктивності експериментальних даних, тестування проводилося на наступному обладнанні та в таких умовах:

- тестовий пристрій: Google Pixel 4a, ОС 12.
- мережеві умови: 1) Wi-Fi з'єднання (швидкість ~50 Мбіт/с); 2) Симульоване повільне 3G з'єднання (швидкість ~1 Мбіт/с, затримка 200 мс) за допомогою емулятора мережі; 3) Повна відсутність мережі (режим "в літаку").
- обсяг тестових даних: Тестування проводилося на наборах даних різного розміру: 10, 100 та 500 нотаток, середня довжина нотатки ~200 символів.
- кількість ітерацій: Кожен сценарій тестування повторювався не менше 10 разів для усереднення результатів.

Особливу увагу приділено роботі механізмів повної синхронізації та резервного копіювання у Google Drive.

На третьому етапі було проведено порівняльний аналіз розробленої моделі з альтернативними підходами, які використовують лише один тип бази даних або тільки локальне SQL-зберігання, або лише хмарне NoSQL-рішення. Додатково враховано варіанти із ручною односпрямованою синхронізацією. Оцінювання відбувалося за критеріями автономності, узгодженості, масштабованості та гнучкості архітектури [33].

Таблиця 3.3 – Порівняння гібридної моделі з альтернативними підходами

Критерії	**ТількиSQL (Room)**	**Тільки NoSQL (Firestore)**	**Ручна синхронізація**	**(SQL + NoSQL)**
Автономність	Повна	Відсутній офлайн-доступ	Часткова	Повна
Узгодженість Змін	Локальна	Кінцева (BASE)	Залежить від користувача	Реактивна, автоматична
Маштабність	Обмежена	Висока	Обмежена	Висока
Гнучкість структури	Жорстка	JSON, адаптивність	Механічна	Комбінована
Реактивність	LiveData	SnapshotListener	Вручну	Повна
Загальна оцінка	Обмежена	Залежить від мережі	Нестабільна	Рекомендовано

Як видно з таблиці, лише гібридна модель повністю задовольняє усі ключові критерії для мобільного середовища з високими вимогами до надійності, автономності та масштабованості.

Завершальним етапом стала апробація гібридної моделі на прикладі мобільного додатку, у якому було реалізовано повну двосторонню синхронізацію на основі реактивного підходу. Проведено симуляції типових сценаріїв використання: створення нових записів, редагування в офлайн-режимі, автоматичне оновлення даних після повернення з’єднання, паралельна робота на кількох пристроях тощо. Це дозволило виявити сильні сторони моделі, оцінити її стабільність, а також підтвердити гіпотезу щодо переваг комбінованого підходу до управління даними у мобільних системах.

3.3 Наукові аспекти та інноваційність розробленої моделі

У межах даної кваліфікаційної роботи було реалізовано гібридну модель зберігання даних, яка поєднує локальну реляційну базу Room (SQL) та хмарну документоорієнтовану систему Firebase Firestore (NoSQL). Такий підхід є відповіддю на сучасні виклики в розробці мобільних додатків, де критичне значення мають автономність, гнучкість, стійкість до збоїв і синхронізація даних у режимі реального часу.

У сучасних умовах стрімкого розвитку мобільних технологій і зростаючої залежності користувачів від мобільних застосунків особливої актуальності набуває проблема забезпечення безперебійного доступу до даних незалежно від стану мережевого з’єднання. Мобільні користувачі очікують, що застосунок буде стабільним, швидким та функціональним як у режимі онлайн, так і офлайн. При цьому дані повинні залишатися консистентними, актуальними та захищеними від втрати. Сучасні дослідження підкреслюють важливість забезпечення узгодженого стану мобільних додатків, особливо при роботі з розподіленими даними, пропонуючи для цього різноманітні підходи, включно з

використанням таких механізмів, як безконфліктні типізовані репліковані дані (CRDTs) [35].

Традиційні рішення, які базуються виключно на реляційних базах даних, ефективно працюють у локальному середовищі, однак не здатні забезпечити масштабованість, багатопристроєвий доступ та синхронізацію. У свою чергу, хмарні NoSQL-рішення, зокрема Firebase Firestore, вирізняються масштабованістю та зручністю в обробці неструктурованих даних, проте залежать від наявності стабільного інтернет-з'єднання. Переваги хмарних баз даних у забезпеченні масштабованого та гнучкого управління даними широко визнані в науковій спільноті [36]. Окреме застосування кожного з підходів має свої суттєві обмеження, що унеможливлює реалізацію універсальної, гнучкої архітектури для мобільного середовища.

У цьому контексті гібридна модель, яка поєднує локальне зберігання (Room/SQLite) із хмарним (Firebase Firestore), дозволяє усунути згадані недоліки та забезпечити цілісність життєвого циклу даних від створення в офлайні до синхронізації у хмарі. Вона є особливо актуальною для застосунків, що працюють у сферах, пов'язаних із польовими умовами, охороною здоров'я, освітою, або використанням у віддалених регіонах, де мережеве покриття нестабільне або обмежене.

Крім того, сучасні вимоги до користувацького досвіду включають не лише доступність і надійність, а й прозорість дій додатку: індикатори збереження, повідомлення про синхронізацію, підтримку резервного копіювання. Усе це має бути реалізовано асинхронно, без блокування основних процесів, що також враховано в обраній архітектурі. Забезпечення обробки даних у реальному часі є ключовим аспектом для сучасних мобільних архітектур, що вимагає відповідних архітектурних рішень для ефективної передачі та аналізу інформації [37].

Реалізація гібридної моделі на основі Room і Firebase Firestore є своєчасною та обґрунтованою з точки зору як інженерної практики, так і наукових досліджень у галузі мобільних систем і управління даними.

Наукова новизна запропонованої моделі полягає у створенні повноцінної гібридної архітектури зберігання даних, яка поєднує локальну реляційну базу

Room (SQL) із хмарною документоорієнтованою Firebase Firestore (NoSQL) у межах одного мобільного застосунку без використання проміжного серверного шару. Даний підхід забезпечує безперервну реактивну двосторонню синхронізацію, що дозволяє гарантувати актуальність і узгодженість даних як у режимі офлайн, так і при відновленні мережевого з'єднання. Ефективне управління станом мобільних додатків та забезпечення їх узгодженості є актуальною науковою задачею, для вирішення якої досліджуються різні підходи, включно з CRDTs, що підкреслює важливість розробки надійних механізмів синхронізації [35].

Вперше в контексті клієнтської мобільної архітектури було реалізовано наскрізну логіку повної синхронізації, яка включає імпорт, експорт та автоматичне очищення хмарного сховища через узагальнений метод повної гібридної синхронізації. Метод поєднує три окремі функції і дозволяє керувати повним циклом актуалізації даних, що є принципово важливим для підвищення стабільності та цілісності бази. Розроблений алгоритм враховує пріоритет останнього оновлення (updatedAt), що забезпечує консистентність при виникненні паралельних змін у хмарі та локально.

Ключовим науковим елементом є уніфікована модель даних, яка водночас відповідає вимогам реляційної структури Room і документо-орієнтованої Firestore.

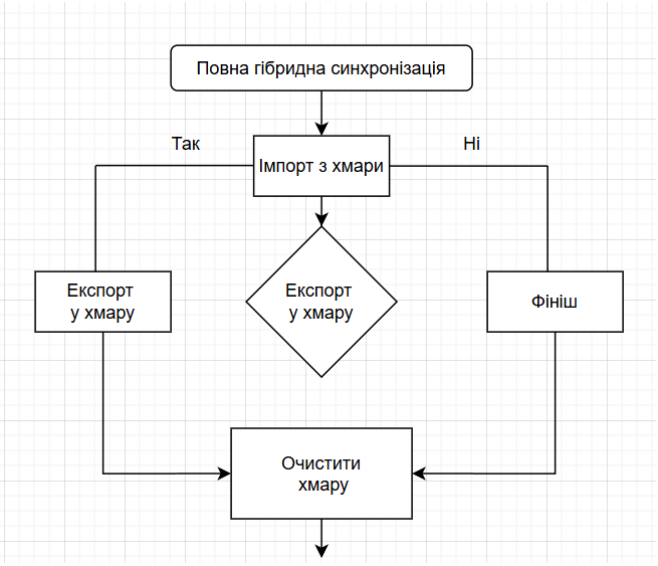

Рисунок 3.2 – Блок-схема повної гібридної синхронізації

Схема відображає поетапну логіку повної синхронізації. Спочатку відбувається імпорт актуальних даних із хмари у локальну базу. Далі при необхідності здійснюється експорт локальних змін до Firestore, після чого виконується очищення хмарної колекції від неактуальних записів. Така послідовність дозволяє зберігати консистентність, актуальність і чистоту обох сховищ.

Це дозволяє уникнути дублювання моделей, анотацій і адаптерів, знижуючи загальну складність системи та підвищуючи її масштабованість. Анотації @Entity і @IgnoreExtraProperties у єдиній моделі Note забезпечують адаптивність структури при оновленні схеми даних, що критично важливо для довготривалого життєвого циклу застосунку.

На відміну від типових практик, де використання Firestore реалізується як єдине джерело істини, а Room виконує лише роль кешу, запропонована модель передбачає рівноправну взаємодію обох джерел даних, із можливістю локального створення, редагування і навіть видалення інформації, яка згодом передається у хмару. Це дозволяє застосунку повноцінно функціонувати в офлайн-режимі, забезпечуючи збереження змін і їх відкладену синхронізацію без втрат. Використання хмарних баз даних надає значні переваги у гнучкості та масштабованості управління даними [36], що робить їх привабливим компонентом для гібридних архітектур.

Окремою інноваційною складовою є реалізація автоматичного очи-щення Firestore видалення документів, яких більше не існує у Room. Це дозволяє підтримувати гігієну хмарної бази без участі користувача або серверних перевірок, що є рідкісним у мобільній архітектурі.

Завдяки гнучкості та самодостатності реалізованого рішення воно може бути адаптоване до інших типів даних або застосоване в багатокористувацьких системах, де важливо зберігати контроль над джерелами даних без втрати продуктивності. Представлена модель є не лише інженерним рішенням, а й науково обґрунтованим підходом до побудови мобільних систем з гібридною архітектурою даних.

Реалізована гібридна модель зберігання даних має високу практичну цінність у контексті розробки мобільних застосунків, орієнтованих на забезпечення безперервної роботи користувача в умовах змінного або відсутнього мережевого підключення.

Практична значущість моделі полягає в тому, що вона дозволяє реалі-зувати повноцінну логіку роботи з даними (створення, редагування, видалення) без потреби у постійному підключенні до Інтернету. Локальне зберігання в Room забезпечує миттєвий доступ до інформації, а хмарна складова Firestore її синхронізацію між пристроями. Важливість такої синхронізації та забезпечення актуальності даних у реальному часі підкреслюється дослідженнями в галузі архітектур для мобільних мереж [37]. Завдяки реалізованій двосторонній синхронізації зміни, зроблені в офлайн-режимі, автоматично передаються у хмару після відновлення з'єднання, що виключає втрату даних і знижує потребу у втручанні користувача.

Особливо важливою є реалізація механізму резервного копіювання з використанням Google Drive, що дозволяє створити зовнішню копію локальної бази та відновити її у випадку втрати, перевстановлення застосунку або зміни пристрою. Такий функціонал критично важливий для користувачів, які оперують із цінною інформацією, наприклад, у сфері ділових нотаток, особистих архівів, польових звітів чи медичних записів.

Універсальність і адаптивність реалізованої моделі дозволяють засто-совувати її не лише в контексті роботи з текстовими нотатками, а й для ширшого спектру даних: мультимедіа, документів, журналів активності, чеклистів тощо. Архітектурна гнучкість забезпечує можливість масшта-бування застосунку на багатокористувацькі сценарії, зокрема з підтримкою авторизації, доступу до даних кількома користувачами одночасно, або інтеграції з іншими сервісами.

Практична значущість моделі полягає в її здатності забезпечити критично важливі властивості сучасного мобільного додатку автономність, узгодженість, збереження та масштабованість при цьому залишаючись техно-логічно простою в реалізації, ефективною у використанні та зручною для кінцевого користувача.

Методологія реалізації гібридної моделі зберігання базується на поєднанні архітектурних принципів реактивного програмування, шаблону MVVM (Model–View–ViewModel) та двосторонньої синхронізації між локальним (SQL) і хмарним (NoSQL) сховищами. Ключовим завданням було побудувати надійний цикл обробки даних, який дозволяє додатку функці-онувати як в офлайн, так і в онлайн режимах без втрати консистентності або актуальності даних.

Архітектурно проєкт реалізовано відповідно до патерну MVVM, який дозволив логічно розмежувати відповідальність між компонентами. Інтерфейс користувача (View) відображає стан даних і взаємодіє з ViewModel, яка виконує роль посередника між користувацьким рівнем і бізнес-логікою. Вся логіка доступу до даних інкапсульована у Repository, який взаємодіє одночасно з локальним сховищем (Room) та хмарним (Firestore), забезпе-чуючи прозору маршрутизацію запитів.

На рівні локального зберігання застосовано Room типобезпечну обгор-тку над SQLite, яка підтримує асинхронну роботу через Kotlin-корутини та реактивну модель даних через LiveData. Сутність Note визначена як дескриптор сутності Room і містить унікальний ідентифікатор (UUID) та поле updatedAt, що виступає центральним параметром у механізмі вирішення конфліктів. Операції CRUD реалізовано через DAO-інтерфейс із чітким розмежуванням потоків і режимів доступу.

Для реалізації хмарного компонента застосовано Firebase Firestore документоорієнтовану базу даних, яка підтримує оновлення у режимі реального часу через addSnapshotListener. У хмарі всі записи зберігаються в колекції notes, де кожен документ ідентифікується тим самим UUID, що й у Room. Підхід дозволив уникнути дублювання даних і забезпечити узгодже-ність між локальним і віддаленим середовищем.

Метод fullSync(), який виконує повний цикл синхронізації, реалізовано як централізовану точку обробки даних. Він включає три ключові етапи:

- імпорт із Firestore до Room перевірка новизни документів за полем updatedAt і оновлення локальної копії.

- експорт змін із Room до Firestore надсилання нових або оновлених записів до хмари.
- очищення Firestore видалення з хмари документів, які більше не існують у локальній базі.

Цей алгоритм дозволяє підтримувати повну відповідність стану даних у обох середовищах та забезпечує автоматичне вирішення конфліктів за принципом «останнє змінене актуальне».

Окремий функціональний модуль присвячено резервному копіюванню. Реалізовано інтеграцію з Google Drive через офіційне API. Копіювання виконується лише після виклику fullSync(), що гарантує цілісність збере-женого стану. Перед архівацією база закривається, що дозволяє скопіювати SQLite-файл без помилок. Уся логіка виконується у фонових потоках за допомогою Kotlin-корутин.

Завдяки реалізованій методології, додаток демонструє високу стабільність, масштабованість та незалежність від мережевих умов, що є результатом правильно обраного архітектурного підходу, ретельно розроблених алгоритмів синхронізації та гнучкої моделі даних.

3.4 Експериментальна перевірка, аналіз результатів та обмеженнь подальші напрямки досліджень

Розроблена в межах дослідження гібридна модель зберігання даних була апробована через створення повноцінного мобільного застосунку для платформи Android (деталі архітектури та реалізації наведені в Розділі 2). У рамках апробації перевірялась здатність системи забезпечувати стабільну, узгоджену й гнучку взаємодію між локальним реляційним сховищем (Room/SQLite) та хмарною документоорієнтованою базою (Firebase Firestore). Реалізована архітектура дозволила досягти ключових цілей дослідження: автономної роботи додатку, автоматичної синхронізації змін та збереження цілісності даних при перемиканні режимів з'єднання. Забезпечення надійної роботи мобільних

систем, особливо критичних, в умовах обмеженого або відсутнього підключення до мережі є важливим напрямком сучасних досліджень [38].

Під час експериментального тестування (умови якого деталізовані в п. 3.2) було змодельовано типові сценарії використання:

- створення, редагування та видалення записів у режимі офлайн.
- синхронізація даних після відновлення мережевого підключення (як для Wi-Fi, так і для симульованого повільного 3G з'єднання).
- паралельна робота на кількох (двох) пристроях під одним обліковим записом для перевірки механізму вирішення конфліктів.
- резервне копіювання даних у Google Drive та їх подальше відновлення.
- аналіз статистики використання мобільних додатків показує високі очікування користувачів щодо стабільності та безперебійності їх роботи [39], що підкреслює важливість ретельного тестування розроблених рішень.

В усіх тестових сценаріях система продемонструвала коректну роботу, відсутність непередбаченої втрати даних або збоїв у функціонуванні. Особлива увага приділялася механізму вирішення конфліктів: завдяки використанню мітки часу (updatedAt) як критерію актуальності, конфлікти, що виникали при одночасному редагуванні даних на різних пристроях (або в офлайн-режимі з подальшою синхронізацією змін, зроблених також у хмарі), вирішувались автоматично за принципом «останнє змінене – актуальне», забезпечуючи консистентність даних без ручного втручання. Технологічні підходи до вирішення конфліктів, що поєднують онлайн та офлайн сценарії, є ключовими для забезпечення узгодженості даних у сучасних мобільних системах [42].

Реактивна модель синхронізації, заснована на LiveData у локальному сховищі та addSnapshotListener у Firestore, забезпечила своєчасне та автоматичне оновлення інтерфейсу користувача без додаткових дій з його боку. Резервне копіювання за допомогою Google Drive стало додатковим запобіжником від втрати даних у разі втрати пристрою або перевстановлення додатку, що підтверджує надійність і завершеність реалізованої моделі.

Таблиця 3.4 – Емпіричні результати функціонального тестування гібридної моделі

Сценарій тестування	**Середній час, с**	**Конфлікти***	**Втрати даних**	**Стабільність**
Створення запису в офлайн-режимі	0,2	0	0	Так
Синхронізація після втрати з`єднання (100 нових записів)	1,8 (Wi-Fi); 4,5 (3G)	0	0	Так
Паралельна робота на двох пристроях (створення/редагування 5 нотаток на кожному)	2,4 (після з'єднання обох пристроїв з мережею)	1	1	Так
Відновлення резервної копії (Google Drive, 100 записів)	4,0	0	0	Так

Як видно з таблиці 3.4, у всіх протестованих сценаріях запропонована гібридна модель продемонструвала високу стабільність роботи, відсутність втрат даних та ефективне автоматичне вирішення потенційних конфліктів синхронізації. Час синхронізації залишався прийнятним навіть в умовах симульованого повільного 3G з'єднання, хоча й очікувано збільшувався порівняно з Wi-Fi. Це підтверджує практичну придатність моделі для використання в реальних умовах нестабільного мережевого доступу, що є критично важливим для підтримки систем у зонах з низьким рівнем підключення [38].

На основі результатів апробації та аналізу експериментальних даних можна зробити висновок, що розроблена гібридна стратегія ефективно виконує поставлені завдання, забезпечуючи як високу автономність і продуктивність додатку в офлайн-режимі, так і надійну узгодженість даних при онлайн-

синхронізації. Реалізація повної двосторонньої синхронізації, автоматичне вирішення конфліктів та інтеграція з хмарними сервісами резервного копіювання суттєво підвищують загальну надійність застосунку та зручність його використання.

Запропонована гібридна модель зберігання даних має значний потенціал для подальшого вдосконалення та розширення функціональності:

- підтримка складних типів даних та операцій: адаптація моделі для зберігання та синхронізації мультимедійного контенту (з використанням Firebase Storage для файлів та збереженням метаданих/посилань у Firestore/Room). Розробка механізмів для роботи зі складнішими структурами даних (наприклад, вкладені об'єкти, списки) та забезпечення їх коректної синхронізації;
- розширені стратегії вирішення конфліктів: для сценаріїв, де простий принцип "останнє змінене актуальне" є недостатнім (наприклад, спільне редагування текстових документів), можливе впровадження більш складних стратегій, таких як three-way merge, operational transformation (OT), або Conflict-free Replicated Data Types (CRDTs), хоча це суттєво ускладнить реалізацію на клієнті. Дослідження в галузі технологій вирішення конфліктів постійно розвиваються, пропонуючи нові підходи для забезпечення узгодженості даних в онлайн та офлайн середовищах [42];
- оптимізація процесу синхронізації;
- впровадження диференційної синхронізації (delta-syncing), що дозволяє передавати лише змінені частини даних, а не весь документ/запис, для зменшення обсягу мережевого трафіку та прискорення процесу, особливо для великих об'єктів даних;
- розробка інтелектуальних алгоритмів пріоритезації синхро-нізації на основі типу даних (наприклад, термінові завдання синхронізуються першими), частоти змін, або поведінкових патернів користувача;
- багатокористувацька взаємодія розширення моделі для підтримки сценаріїв спільної роботи кількох користувачів над одними даними, що вимагатиме впровадження механізмів контролю доступу (наприклад, через Firebase Security Rules) та більш складних правил вирішення конфліктів;

- інтеграція з іншими сервісами розширення функціональності шляхом інтеграції з іншими хмарними API, такими як Google Calendar (для створення нагадувань на основі нотаток), Google Tasks, Dropbox (як альтернативний сервіс для резервного копіювання), або сторонні аналітичні платформи для збору статистики використання;
- кросплатформеність адаптація та реалізація запропонованих принципів гібридного зберігання для інших мобільних платформ (iOS з використанням Core Data/SQLite та відповідних Firebase SDK) або кросплатформенних фреймворків (наприклад, Flutter з sqflite та cloud_firestore React Native з Watermelon DB /AsyncStorage та Firebase SDK).;
- Незважаючи на продемонстровану ефективність та переваги, важливо об'єктивно оцінити певні обмеження реалізованої гібридної моделі та проведеного дослідження:
- стратегія вирішення конфліктів: як зазначалося, використання принципу "останнє змінене актуальне" на основі мітки updatedAt є простим та ефективним для даного типу додатку (нотатки), але може бути непридатним для даних зі складнішою семантикою або у випадках, де важлива історія змін чи паралельне редагування різними користувачами одного фрагменту. Сучасні дослідження вказують на необхідність адаптивних підходів до вирішення конфліктів залежно від контексту застосування [43];
- масштабованість синхронізації для екстремальних навантажень: хоча Firebase Firestore добре масштабується, тестування проводилося на обсягах даних та інтенсивності змін, типових для персонального мобільного застосунку. В умовах дуже великої кількості одночасних змін від одного користувача або при надзвичайно великій кількості документів у колекції можуть виникати затримки або зростання вартості використання хмарних сервісів;
- залежність від сторонніх сервісів функціонування моделі залежить від доступності, стабільності та політик використання Firebase Firestore та Google Drive. Будь-які зміни в їхніх API, тарифних планах або умовах надання послуг можуть вимагати відповідної адаптації реалізованого рішення;

- специфічність технологічного стеку дослідження та реалізація були зосереджені на конкретній комбінації технологій: Room/SQLite та Firebase Firestore для платформи Android. Хоча основні архітектурні принципи є переносимими, деталі реалізації для інших платформ або з використанням інших СУБД чи хмарних сервісів потребуватимуть окремої розробки та адаптації;
- обмеженість тестового середовища експериментальне тесту-вання, хоч і проводилося в різних змодельованих умовах (включаючи різні типи мереж та обсяги даних), не може повністю охопити весь спектр можливих реальних сценаріїв використання, конфігурацій пристроїв, версій ОС та мережевих особливостей, з якими можуть зіткнутися користувачі;

Узагальнюючи, запропонована та апробована гібридна архітектура зберігання даних успішно вирішує поставлені завдання щодо забезпечення автономності, узгодженості та надійності даних у мобільних додатках. Вона не лише демонструє свою працездатність у межах дослідницького прототипу, але й закладає міцний фундамент для подальшого розвитку та застосування як універсальної стратегії для створення сучасних мобільних систем, з урахуванням виявлених обмежень та окреслених перспектив для вдосконалення.

3.5 Висновок до третього розділу

У даному розділі було представлено наукове обґрунтування, реалізацію та апробацію запропонованої гібридної моделі зберігання даних, що поєднує локальне реляційне сховище (Room/SQLite) та хмарну документоорієнтовану базу (Firebase Firestore) для мобільних додатків.

- Обґрунтовано наукову проблему та актуальність дослідження, що полягає у необхідності створення стійких та гнучких архітектур даних для мобільних застосунків, здатних забезпечити безперервну роботу в офлайн-режимі, узгоджену синхронізацію та збереження цілісності даних. Сформульовано мету, завдання та висунуто гіпотезу щодо переваг комбінованого SQL/NoSQL підходу.

- Детально описано методологію дослідження, яка включала аналітичне моделювання, розробку прототипу, функціональне тестування в контрольованих умовах (різні типи мереж, обсяги даних) та порівняльний аналіз з альтернативними стратегіями.
- Представлено ключові аспекти наукової новизни розробленої моделі, зокрема:
- Інтеграція SQL та NoSQL у рамках мобільного клієнта з автоматизованою реактивною двосторонньою синхронізацією без проміжного серверного шару.
- Реалізація узагальненого методу повної гібридної синхронізації (fullSync()), що включає імпорт, експорт та автоматичне очищення хмарного сховища.
- Використання уніфікованої моделі даних, адаптивної до обох типів сховищ, та механізму вирішення конфліктів на основі часової мітки updatedAt.
- Проведено апробацію моделі шляхом реалізації мобільного застосунку та тестування ключових сценаріїв (офлайн-робота, синхронізація при різних умовах мережі, паралельна робота, резервне копіювання). Експериментальні дані, представлені в Таблиці 3.4, підтвердили стабільність роботи системи, відсутність втрат даних та ефективне автоматичне вирішення конфліктів, що свідчить про досягнення поставлених цілей та підтвердження висунутої гіпотези.
- Проаналізовано практичну значущість розробленої моделі, яка полягає у можливості створення мобільних додатків з високим рівнем автономності, надійності та зручності для користувача, особливо в умовах нестабільного мережевого з'єднання.
- Окреслено перспективи подальшого розвитку моделі, включаючи підтримку складніших типів даних, розширені стратегії вирішення конфліктів, оптимізацію синхронізації, багатокористувацьку взаємодію та адаптацію для кросплатформених рішень.
- Визначено обмеження запропонованої моделі та проведеного дослідження, що стосуються універсальності стратегії вирішення конфліктів,

масштабованості для екстремальних навантажень, залежності від сторонніх сервісів та специфічності обраного технологічного стеку.

Особливість запропонованої гібридної моделі полягає у досягненні синергії між локальним SQL та хмарним NoSQL сховищами безпосередньо на мобільному клієнті. Це досягається за рахунок реактивного двостороннього механізму синхронізації, який працює автоматично, уніфікованої структури даних для обох середовищ, та простого, але ефективного алгоритму вирішення конфліктів за часовою міткою. Унікальним є також впровадження повного циклу узгодження (fullSync()) з функцією автоматичного очищення неактуальних даних у хмарі, що забезпечує як надійність, так і оптимальне використання ресурсів.

Таким чином, результати, отримані в ході дослідження та представлені в цьому розділі, демонструють, що розроблена гібридна модель зберігання даних є науково обґрунтованим та практично реалізованим рішенням актуальної проблеми забезпечення ефективного управління даними в сучасних мобільних додатках. Модель поєднує переваги SQL та NoSQL підходів, забезпечуючи надійність, гнучкість та високий рівень користувацького досвіду.

РОЗДІЛ 4 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

4.1 Умови праці розробників сучасних програмних додатків і баз даних

У сучасному світі, де інформаційні технології є основою функціонування більшості сфер людської діяльності, ключову роль відіграють бази даних та спеціалісти, що забезпечують їхню розробку, підтримку та інтеграцію. Розробка програмного забезпечення, зокрема SQL і NoSQL баз даних для мобільних та веб-додатків, є інтелектуально напруженою, високоавтоматизованою діяльністю, яка потребує тривалого перебування працівника у статичному положенні перед екраном комп'ютера [45]. Програмісти, системні аналітики, адміністратори БД, тестувальники мобільних додатків усі вони працюють у середовищі, де хоча й мінімальні фізичні навантаження, однак значними є психоемоційні, когнітивні та гігієнічні ризики [44].

Інтенсифікація праці в ІТ-сфері, високі вимоги до точності, швидкості реакції, тривале зосередження уваги та постійна взаємодія з цифровими пристроями висувають додаткові виклики щодо охорони праці та організації безпечного робочого середовища . Належна оцінка умов праці, виявлення шкідливих і небезпечних факторів, а також впровадження профілактичних заходів мають вирішальне значення для збереження працездатності та здоров'я фахівців [47].

Переважна частина робочого часу програмістів і адміністраторів баз даних припадає на сидячу роботу перед комп'ютером. Проєктування архітектури БД, написання SQL-запитів, аналіз логіки програм, відлагодження коду, тестування додатків усі ці процеси потребують тривалої концентрації у статичній позі. Це призводить до гіподинамії, що негативно впливає на серцево-судинну систему, метаболізм і опорно-руховий апарат [46].

Фіксоване положення тіла викликає статичне перенавантаження м'язів спини, шиї, плечового пояса і попереку. У результаті виникають болі, спазми, а

при хронічному впливі дегенеративно-дистрофічні зміни хребта (осте-охондроз, сколіоз, міжхребцеві грижі) Часто спостерігаються головні болі напруги, що спричинені тривалим м’язовим напруженням [46].

ІТ-фахівці протягом робочого дня постійно взаємодіють з екранами моніторів, аналізуючи дрібний текст, код, таблиці й графічні елементи. Тривале зорове навантаження призводить до розвитку комп’ютерного зорового синдрому (КЗС), що включає сухість очей, подразнення, зниження гостроти зору, відчуття піску, двоїння та головний біль [48].

Факторами, що посилюють КЗС, є умови недостатнього або неправильного освітлення, блиску на екрані монітора, низької якості зображення, відсутності зорових перерв [49].

Діяльність програмістів та інженерів БД пов’язана з високим рівнем відповідальності, зокрема за збереження, захист і цілісність даних. Часто робота відбувається під тиском дедлайнів, з високими вимогами до швидкості прийняття рішень, багатозадачності та уважності до деталей. Це сприяє розвитку хронічного стресу, емоційного вигорання, тривожності.

Особливо напружені умови спостерігаються в стартапах, на етапах запуску продуктів, при ліквідації критичних багів, або у проєктах з великою відповідальністю за безперервність сервісів [46].

Ергономіка робочого місця надзвичайно важлива. Воно повинно включати: стіл і крісло з регулюванням по висоті, підтримку спини та правильну посадку, розміщення монітора на відстані 60–70 см на рівні очей, антивідблискові екрани, підставки для ніг та рук за потреби [52].

Навантаження на верхні кінцівки, при частій роботі з клавіатурою і мишею створює ризики розвитку тунельного синдрому (синдрому зап’ястного каналу), тендинітів, бурситів. Тривале повторення одноманітних рухів паль-цями без належної підтримки та відпочинку викликає оніміння, біль, порушення рухливості [3].

Профілактика та компенсаційні заходи, щоб зменшити вплив шкідливих факторів, рекомендується [52]:

- дотримуватись режиму праці та відпочинку (перерви щогодини на 5–10 хвилин);
- виконувати виробничу гімнастику;
- користуватись ортопедичними аксесуарами;
- застосовувати правило 20-20-20 для відпочинку очей;
- змінювати положення тіла під час роботи;
- чергувати види діяльності.

Також актуальним є забезпечення умов віддаленої праці, що передбачає відповідальність роботодавця за інформування, а працівника за дотримання умов охорони праці вдома [47].

Фізіологічні та психологічні ризики тривалого перебування в умовах, характерних для професійної діяльності фахівців з інформаційних технологій, зокрема розробників програмного забезпечення, адміністраторів та архітекторів баз даних, можуть зумовити розвиток стійких порушень. Одним із найпоширеніших психоемоційних станів, що виникають у цій сфері, є емоційне вигорання синдром, який формується на тлі хронічного стресу, постійного навантаження та монотонної праці [46].

Причинами емоційного вигорання можуть бути як високий рівень відповідальності за збереження та цілісність даних, так і тиск дедлайнів, інтенсивна розумова робота, відсутність різноманітності у виконуваних завданнях, недостатня підтримка керівництва або колег, а також невизначеність у постановці цілей і часті конфлікти в команді.

Не менш серйозним є постійний професійний стрес, який супроводжує багатьох ІТ-фахівців. Його джерелами можуть бути не лише дедлайни та велика відповідальність, але й необхідність оперативного усунення критичних збоїв у роботі систем, ризики кіберзагроз, потреба в безперервному самонавчанні, оволодінні новими технологіями, а також взаємодія зі складними замовниками .

У фізіологічному вимірі хронічний стрес здатен викликати порушення серцево-судинної системи, підвищення артеріального тиску, аритмію, пригнічення імунітету, розвиток гастриту, а згодом навіть виразкової хвороби. У психологічному аспекті він проявляється тривожністю, дратівливістю,

перепадами настрою, депресивними станами, а також труднощами з концентрацією уваги [46].

До інших фізіологічних ризиків, властивих ІТ-праці, належать проблеми з травленням, що виникають внаслідок нерегулярного харчування, частих перекусів, споживання фастфуду, надмірної кількості кави або енергетичних напоїв [46]. Такі умови можуть спричинити розвиток гастриту, синдрому подразненого кишківника, печії та інших функціональних розладів. Низька фізична активність у поєднанні з незбалансованим раціоном формує передумови для набору надмірної ваги, ожиріння, а також пов’язаних із ним захворювань, зокрема діабету другого типу, гіпертонії та ішемічної хвороби серця [46].

Вимоги до організації робочого місця відповідно до норм ДСТУ 7299:2013 . Забезпечення оптимальних та безпечних умов праці для фахівців, які постійно працюють із комп’ютерною технікою, є необхідною умовою збереження здоров’я працівників і підвищення ефективності їхньої діяльності . В Україні такі вимоги регламентуються державними санітарними правилами та нормами ДСанПіН 3.3.2.007-98, а також низкою відповідних ДСТУ, що стосуються ергономіки, освітлення, мікроклімату та режиму праці [49].

4.2 Вплив електромагнітного випромінювання від мобільних пристроїв при тестуванні та використанні мобільних додатків

Інтенсивний розвиток мобільних технологій зробив смартфони та планшети невід’ємною частиною як повсякденного життя, так і професійної діяльності, особливо у сфері інформаційних технологій.

Для розробників і тестувальників мобільного програмного забезпечення взаємодія з цими пристроями є не лише інструментом роботи, а й джерелом потенційних виробничих ризиків, серед яких вплив електромагнітного випромінювання (ЕМВ) [51].

Основними джерелами ЕМВ у середовищі мобільної розробки виступають вбудовані модулі сучасних мобільних пристроїв зокрема Wi-Fi, Bluetooth, NFC, а також антени стільникового зв’язку, що працюють у режимах GSM, 3G,

4G/LTE та 5G [52]. Під час тривалих тестових сесій або багаторазового використання пристроїв у реальному середовищі, особливо за умов слабкого сигналу, рівень випромінювання значно підвищується. Це пояснюється тим, що мобільні телефони автоматично збільшують потужність передавача для підтримання стабільного з'єднання [52].

За умов, коли з одним фахівцем взаємодіє кілька активних пристроїв одночасно, кумулятивне навантаження може зростати в рази.

Оцінку потенційної шкоди від ЕМВ здійснюють за допомогою показника SAR (Specific Absorption Rate) питомого коефіцієнта поглинання електромагнітної енергії тілом людини. У Європі та Україні діють норми, згідно з якими SAR не повинен перевищувати 2 Вт/кг для голови та тулуба і 4 Вт/кг для кінцівок. Виробники пристроїв зобов'язані декларувати ці показники та підтверджувати їх відповідними випробуваннями. Для розро-бників мобільного ПЗ важливо враховувати SAR при виборі обладнання для щоденного використання [52].

Хоча сучасні пристрої зазвичай відповідають чинним стандартам безпеки, тривалий або неправильний контакт з ними може спричинити негативні фізіологічні ефекти, такі як головний біль, млявість, дратівливість, порушення сну та концентрації уваги [50].

Особливо це актуально в контексті тестування програмного забезпечення, яке передбачає активне використання мережевих функцій і багатогодинну роботу з пристроями на близькій відстані до тіла [51].

У зв'язку з цим важливо впроваджувати превентивні заходи, які мінімізують ризики для здоров'я [52]:

- розміщення мобільних пристроїв на підставках або тестових стендах;
- використання зовнішніх гарнітур, режиму гучного зв'язку, дистанційного керування;
- чергування видів діяльності наприклад, тестування із завданнями на ПК (аналіз логів, написання документації);
- вимкнення непотрібних бездротових модулів під час локальних завдань;

- використання авіарежиму за відсутності потреби у зв’язку;

У лабораторних умовах ефективною практикою є використання емуляторів, симуляторів та систем автоматизованого тестування, що дозволяють зменшити тривалість безпосередньої присутності біля пристрою [52].

У випадку реального польового тестування важливо уникати зон зі слабким покриттям, де пристрій працює на підвищеній потужності, а також забезпечувати провітрювання приміщень і дотримання санітарно-гігієнічних норм [10].

Навчання персоналу правилам безпечного поводження з мобільними пристроями також має бути складовою системи охорони праці [47].

Таким чином, у професійній діяльності, пов’язаній із тестуванням та розробкою мобільних додатків, важливо не лише дотримуватись технологічних стандартів, а й забезпечити захист здоров’я працівників, у тому числі шляхом контролю впливу електромагнітного випромінювання [51].

4.3 Підвищення стійкості роботи в об’єктах приладо-будівної галузі в у воєнний час

У зв’язку з триваючими військовими конфліктами в Україні питання забезпечення безпеки, безперебійної роботи та збереження кадрового та технічного потенціалу підприємств критичної інфраструктури, зокрема приладобудівної галузі, набуло особливої актуальності. В умовах воєнного часу підприємства повинні не лише виконувати виробничі завдання, а й забезпечувати високу стійкість до зовнішніх загроз, включаючи ракетні обстріли, диверсійні дії, перебої в постачанні ресурсів та інші ризики [54].

Основні загрози для об’єктів приладобудівної галузі у воєнний час у контексті воєнного стану приладобудівні підприємства. Як і інші об’єкти промислового комплексу, стикаються з широким спектром загроз, що можуть порушити їхню стабільну діяльність [55]. Ці загрози охоплюють як безпо-середні фізичні впливи, так і опосередковані чинники, що знижують працездатність підприємства.

Насамперед, підприємства приладобудування можуть зазнати прямих уражень унаслідок авіаційних, ракетних та артилерійських обстрілів. Навіть один влучний удар може призвести до повного або часткового знищення виробничих потужностей, порушення технологічних процесів, знищення унікального обладнання, що часто є імпортним і складно відновлюваним [56].

Другою ключовою загрозою є втрата людського ресурсу. Через мобілізацію значної частини кваліфікованого персоналу, евакуацію працівників з небезпечних регіонів, психологічні травми або навіть загибель фахівців, підприємство ризикує залишитися без компетентних кадрів, без яких виробничий процес неможливий [57].

Також критичною є нестабільність логістичних ланцюгів. Порушення транспортних коридорів унеможливлює своєчасне постачання сировини, матеріалів, комплектуючих елементів, а також вивезення готової продукції. Окремо варто виділити перебої з енергозабезпеченням, які впливають на роботу складних приладів і точного виробництва [58].

Крім того, у зоні ризику перебувають інформаційні системи. У воєнний час зростає інтенсивність кіберзагроз, спрямованих на виведення з ладу систем управління виробництвом, викрадення або знищення технічної документації, саботаж автоматизованих процесів [59].

Останнім, але не менш важливим чинником є психоемоційний стан працівників, що знижується через постійний стрес, втрати в особистому житті та невизначеність майбутнього. Усі ці фактори разом формують складне середовище, у якому підприємство має функціонувати, забезпечуючи при цьому безперервність виробництва [60].

Заходи щодо підвищення стійкості підприємства Інженерно-технічних заходів для підвищення стійкості об’єктів приладобудівної галузі доцільно впроваджувати низку інженерно-технічних рішень. Передусім ідеться про укріплення конструкцій будівель, зокрема технічних корпусів і приміщень, де розміщується високоточне обладнання. Доцільно встановлювати броньовані жалюзі, протиударні вікна, внутрішні бар'єри, які знижують наслідки вибухової хвилі [53].

Надзвичайно важливо перемістити критичні вузли та обладнання у підземні або захищені приміщення, забезпечивши їх автономним електропостачанням, вентиляцією та зв’язком. Варто також дублювати окремі ділянки технологічного процесу на інших майданчиках, які географічно віддалені й мають нижчий рівень ризику [55].

Системи життєзабезпечення (електроенергія, вода, тепло) мають бути переобладнані для роботи у кризових умовах. Зокрема, встановлення дизель-генераторів, UPS-систем, локальних резервуарів з водою та паливом дає змогу підтримувати базовий рівень функціонування навіть при тривалому відключенні централізованих мереж [61].

Організаційні заходи та підготовка є основою оперативного реагування на надзвичайні ситуації. Кожне підприємство повинно мати затверджений План дій у надзвичайних ситуаціях (ПДНС), який включає сценарії обстрілів, евакуації, зупинки технологічного процесу та його відновлення [58].

Доцільно створити антикризову комісію або оперативний штаб, відповідальний за ухвалення рішень у режимі реального часу. Працівники мають бути поінформовані про свої ролі, маршрути евакуації, місця збору й порядок дій у випадку сигналу тривоги.

Рекомендується запровадити ротацію персоналу, формування дублерів на критичних посадах, організувати навчання та тренування, спрямовані на підвищення готовності колективу до нестандартних ситуацій. Там, де можливо, варто перевести частину персоналу на віддалену форму роботи з використанням захищених каналів зв’язку.

Енергетична незалежність є одним із найуразливіших компонентів приладобудівного підприємства є енергозалежність. З метою зниження цього ризику необхідно забезпечити автономність енергозабезпечення, зокрема завдяки резервним генераторам, сонячним панелям, акумуляторним системам.

Особливу увагу слід приділити енергоефективності виробничих процесів, щоб зменшити залежність від нестабільного енергопостачання. У критичних умовах важливо мати гнучкі режими виробництва, які дозволяють працювати в обмеженому енергетичному середовищі [59].

Інформаційна та кібербезпека важливий захист інформаційного середовища приладобудівного підприємства є важливою складовою загальної безпеки. У воєнний час особливо зростає актуальність протидії кібератакам, які можуть бути спрямовані на паралізацію виробництва, виведення з ладу автоматизованих систем, крадіжку або знищення технічної документації [62].

Першочергово потрібно забезпечити фізичний та логічний захист серверних кімнат, комп'ютерів та мережевого обладнання. Внутрішні IT-системи мають бути ізольовані від відкритого доступу до Інтернету, а обмін файлами повинен відбуватись виключно через зашифровані канали.

Необхідно впровадити регулярне резервне копіювання критичної інформації на зовнішні носії, які зберігаються в іншій геолокації. Важливо також розмежувати доступ користувачів до інформаційних ресурсів та обмежити його лише необхідним обсягом відповідно до посадових обов'язків.

Однією з найважливіших складових є навчання персоналу елементарним принципам кібергігієни, включаючи роботу з електронною поштою, уникнення підозрілих файлів та дотримання правил автентифікації.

Взаємодія з державними структурами та територіальною обороною в умовах збройної агресії ефективне функціонування підприємства значною мірою залежить від налагодженого діалогу з органами місцевого самов-рядування, військовими адміністраціями та підрозділами територіальної оборони. Це дозволяє оперативно отримувати інформацію про загрози, погоджувати дії під час обстрілів чи евакуацій, а також залучати допомогу в охороні об'єкта [60].

Підприємства можуть брати участь у програмах зміцнення критичної інфраструктури, що включають фінансування на зміцнення будівель, закупівлю засобів індивідуального захисту, систем зв'язку та сповіщення. Доцільним є також встановлення локальних систем оповіщення, що реагують на сигнали повітряної тривоги.

У тісній кооперації з територіальною обороною підприємства можуть розробляти спільні заходи для захисту від можливих диверсій, незаконного проникнення на об'єкт чи саботажу.

Психологічна та соціальна підтримка працівників в умовах хронічного стресу й небезпеки однією з ключових задач керівництва є підтримка морального духу колективу. Для цього необхідно організувати систему психологічної допомоги, у тому числі за участі фахівців. Працівники повинні мати змогу звернутися по підтримку, отримати консультацію або пройти адаптаційні тренінги [10].

Варто передбачити створення обладнаних укриттів, де люди можуть не лише сховатися, а й комфортно перебувати під час тривалих повітряних тривог. На території підприємства мають бути доступні засоби першої допомоги, питна вода, продукти тривалого зберігання.

Крім цього, соціальна стабільність персоналу зміцнюється через допомогу у випадках втрат, виплату додаткових компенсацій, організацію проживання для працівників і членів їхніх родин, які втратили житло. Позитивну роль відіграють мотиваційні програми – гнучкі графіки роботи, преміювання, навчання та підвищення кваліфікації.

Підвищення стійкості об'єктів приладобудівної галузі у воєнний час є завданням стратегічного рівня. Воно потребує комплексного підходу, який включає технічні, організаційні, кадрові та психологічні заходи. Успішна реалізація таких заходів забезпечує не лише збереження підприємства, а й зміцнення обороноздатності країни в цілому [53].

4.4 Висновок до четвертого розділу

Було проведено комплексний аналіз умов праці та потенційних ризиків для здоров’я фахівців, задіяних у розробці сучасних програмних додатків і баз даних, а також розглянуто специфічні виклики, пов’язані з безпекою об’єктів приладобудівної галузі в умовах воєнного стану.

Було встановлено, що праця розробників програмного забезпечення та адміністраторів баз даних характеризується тривалим статичним навантаженням, інтенсивною зоровою та психоемоційною напругою. Це призводить до ризиків розвитку захворювань опорно-рухового апарату,

комп’ютерного зорового синдрому, емоційного вигорання та хронічного стресу. Підкреслено важливість ергономічної організації робочого місця, дотримання режиму праці та відпочинку, впровадження профілактичних заходів, а також належного інформування працівників, зокрема при віддаленій роботі, відповідно до чинних нормативів, таких як ДСанПіН 3.3.2.007-98 та ДСТУ 7299:2013.

Окрему увагу приділено впливу електромагнітного випромінювання від мобільних пристроїв, що є актуальним для тестувальників мобільних додатків. Наголошено на необхідності контролю показника SAR, використанні захисних заходів, таких як розміщення пристроїв на підставках, застосування гарнітур, емуляторів та оптимізації робочих процесів для мінімізації кумулятивного впливу ЕМВ.

В контексті воєнного стану детально проаналізовано загрози для об’єктів приладобудівної галузі, включаючи фізичні ураження, втрату людського ресурсу, порушення логістики, енергетичні та кіберзагрози. Запропоновано комплекс заходів для підвищення стійкості таких підприємств, що охоплює інженерно-технічні рішення (укріплення, автономізація життєзабезпечення), організаційні заходи (планування дій у НС, навчання персоналу), забезпечення енергетичної незалежності, посилення інформаційної та кібербезпеки, налагодження взаємодії з державними структурами та надання психологічної і соціальної підтримки працівникам.

Таким чином, розділ демонструє критичну важливість системного підходу до питань охорони праці та безпеки в надзвичайних ситуаціях. Реалізація розглянутих заходів є невід’ємною умовою для збереження здоров’я та працездатності персоналу в ІТ-сфері, а також для забезпечення стабільного та безпечного функціонування стратегічно важливих підприємств приладобудівної галузі в умовах сучасних викликів, зокрема під час воєнного стану.

ВИСНОВОК

Сучасні мобільні застосунки та ефективне управління їхніми даними є значним викликом, де забезпечення надійності, офлайн-доступу та синхронізації залишається пріоритетом. У рамках даної кваліфікаційної роботи здійснено всебічне дослідження проблеми об’єднання SQL та NoSQL баз даних, результатом якого стала розробка та наукове обґрунтування гібридної моделі зберігання даних, що ефективно інтегрує переваги обох підходів для сучасних мобільних додатків.

В першому розділі кваліфікаційної роботи освітнього рівня «Магістр»:

- Проаналізовано проблематику зберігання даних у мобільних додатках, розглянуто особливості SQL (SQLite/Room) та NoSQL (Firebase Firestore) рішень, а також існуючі стратегії їх інтеграції.
- Обґрунтовано актуальність та необхідність розробки спеціалізованої, клієнт-орієнтованої гібридної архітектури для ефективного поєднання переваг SQL та NoSQL в мобільних умовах.

В другому розділі кваліфікаційної роботи:

- Спроєктовано та реалізовано гібридну архітектуру мобільного додатку для Android за шаблоном MVVM, що поєднує локальне SQL-сховище на базі Room та хмарне NoSQL-сховище Firebase Firestore.
- Розроблено та впроваджено ключові механізми взаємодії: двосторонню реактивну синхронізацію даних, логіку вирішення конфліктів за часовою міткою (updatedAt), та функціонал резервного копіювання в Google Drive.

В третьому розділі кваліфікаційної роботи:

- Науково обґрунтовано запропоновану гібридну модель, підтверджено її новизну, що полягає у комплексній клієнт-орієнтованій архітектурі з автоматизованою синхронізацією та унікальним методом fullSync для повного узгодження даних, включаючи очищення хмари.

- Апробовано модель шляхом тестування реалізованого застосунку, що продемонструвало її ефективність, стабільність, здатність забезпечувати цілісність даних та підтвердило переваги гібридного підходу.
- Визначено практичну значущість результатів, що полягає у наданні розробникам ефективного інструменту для створення надійних мобільних додатків з покращеним користувацьким досвідом.

У розділі «Охорона праці та безпека в надзвичайних ситуаціях»:

- Проаналізовано умови праці розробників, потенційні ризики для здоров'я та вплив електромагнітного випромінювання.
- Описано заходи для підвищення стійкості об'єктів приладобудівної галузі в умовах воєнного стану.

Підсумовуючи, виконана кваліфікаційна робота демонструє, що ретельно спроєктована гібридна архітектура, яка інтегрує сильні сторони SQL та NoSQL технологій, є потужним та ефективним рішенням для створення сучасних, надійних та зручних мобільних додатків. Такий підхід дозволяє успішно відповідати на високі вимоги користувачів щодо якості, доступності та узгодженості даних, закладаючи фундамент для подальшого розвитку та розширення функціональності подібних систем.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Döhmen, T., Geacu, R., Hulsebos, M., & Schelter, S. (2024). SchemaPile: a large collection of relational database schemas. Proceedings of the ACM on Management of Data, 2(3), 1-25.

2. Candel, C. J. F., Ruiz, D. S., & García-Molina, J. J. (2022). A unified metamodel for NoSQL and relational databases. Information Systems, 104, 101898.

3. Khan, W., Kumar, T., Zhang, C., Raj, K., Roy, A. M., & Luo, B. (2023). SQL and NoSQL database software architecture performance analysis and assessments—a systematic literature review. Big Data and Cognitive Computing, 7(2), 97.

4. Pérez-Mercado R., Balderas A., Muñoz A., et al. ChatbotSQL: A Conversational Agent for Learning SQL Querying over Relational Databases. SoftwareX. 2023;22:101346. DOI

5. Roy-Hubara, N., Sturm, A., & Shoval, P. (2023). Designing NoSQL databases based on multiple requirement views. Data & knowledge engineering, 145, 102149.

6. Faridoon, A., & Imran, M. (2021). Big Data Storage Tools Using NoSQL Databases and Their Applications in Various Domains: A Systematic Review. Computing & Informatics, 40(3).

7. Deb, S., Sattar, A., & Jabiullah, M. I. Applying NoSQL Database in Data Mining Models for Educational Big Data.

8. Faridoon, A., & Imran, M. (2021). Big Data Storage Tools Using NoSQL Databases and Their Applications in Various Domains: A Systematic Review. Computing & Informatics, 40(3).

9. Dhummad, S., & Patel, T. Advanced SQL Techniques for Efficient Data Migration: Strategies for Seamless Integration Across Heterogeneous Systems.

10. Abbas, N., & Farah, J. (2023). Optimizing E-commerce Databases: A Comparative Analysis of SQL and NoSQL Solutions.

11. Rifqi, M. A., Husodo, A. Y., & Wijayanto, H. (2024, November). Development of SQL and NoSQL Database Integration Using Query Direct Access

and Change Data Capture Approaches in an Android Application. In 2024 IEEE International Conference on Communication, Networks and Satellite (COMNETSAT) (pp. 671-679). IEEE.

12. Vatjalainen, A. (2023). SQL versus NoSQL: comparison case MySQL versus MongoDB.

13. Matcha, S., & Mishra, R. Database Selection and Management: Choosing the Right Database (SQL vs. NoSQL) for Your Application.

14. Vieira, M. A., Ribeiro, E. L. F., Claro, D. B., & Mane, B. (2021). Integration Model between Heterogeneous Data Services in a Cloud. J. Univers. Comput. Sci., 27(4), 387-412.

15. Fuksa, M., Speth, S., & Becker, S. (2024, September). MVVM Revisited: Exploring Design Variants of the Model-View-ViewModel Pattern. In International Conference on Enterprise Design, Operations, and Computing (pp. 163-181). Cham: Springer Nature Switzerland.

16. Saeed, M. S. (2024). Traditional view system vs. Kotlin-Driven Jetpack Compose in Native Android Development.

17. Ferreira, C., Lopes, M., Correia, L., Wanzeller, C., Sá, F., Martins, P., & Abbasi, M. (2023). Database Performance on Android Devices, A Comparative Analysis. In Marketing and Smart Technologies: Proceedings of ICMarkTech 2022, Volume 2 (pp. 261-273). Singapore: Springer Nature Singapore.

18. Peiponen, J. (2024). Firebasen palveluilla toteutettu sijoitusportfolio web-sovellus.

19. Sahara, C. R., & Aamer, A. M. (2022). Real-time data integration of an internet-of-things-based smart warehouse: a case study. International Journal of Pervasive Computing and Communications, 18(5), 622-644.

20. Jeiran, M., Vogel, B. I., & Miller, K. J. (2022, May). Common data format. In Automatic target recognition XXXII (Vol. 12096, pp. 62-76). SPIE.

21. Győrödi, C. A., Turtureanu, T., Győrödi, R. Ş., & Zmaranda, D. R. (2023). Implementing a Synchronization Method between a Relational and a Non-Relational Database. Big Data and Cognitive Computing, 7(3), 153.

22. Teixeira, Â. M. T. (2021). Real-Time user-based coverage of a sports event: A Web Application for the modern football fans (Master's thesis, Universidade do Porto (Portugal)).

23. Peuralinna, J. (2024). Data Lineage in the financial sector.

24. Senapartha, I. K. D. (2021). Implementasi Single Sign-On Menggunakan Google Identity, REST dan OAuth 2.0 Berbasis Scrum. Jurnal Teknik Informatika Dan Sistem Informasi, 7(2), 307-320.

25. Marthinsen, F. S. (2023). Creating a framework for live data analysis of Discord servers.

26. Rane, N., Choudhary, S., & Rane, J. (2024). Artificial intelligence for enhancing resilience. Journal of Applied Artificial Intelligence, 5(2), 1-33.

27. De Filippo, A., Lombardi, M., & Milano, M. (2021). Integrated offline and online decision making under uncertainty. Journal of Artificial Intelligence Research, 70, 77-117.

28. Möller, L. (2025). Professional Growth of a User Interface Developer.

29. Zhang, L., Pang, K., Xu, J., & Niu, B. (2022). JSON-based control model for SQL and NoSQL data conversion in hybrid cloud database. Journal of Cloud Computing, 11(1), 23.

30. Agbeyangi, A., & Suleman, H. (2024). Formalising solutions to network availability issues in low-resource environments: An offline storage design pattern for software systems. South African Computer Journal, 36(2), 1-30.

31. Shareef, T. H., Sharif, K. H., & Rashid, B. N. (2022). A survey of comparison different cloud database performance: SQL and NoSQL. Passer Journal of Basic and Applied Sciences, 4(1), 45-57.

32. Deb, P., Singh, N., Kumar, S., Rai, N., & Iyengar, N. C. S. N. (2010). Offline navigation system for mobile devices. International Journal of Software Engineering & Application, 1(2).

33. Eisenhuth, P., & Jablonski, S. (2022). Knowledge-based Recommendation for Polyglot Persistence. In CDMS@ VLDB.

34. Segun-Falade, O. D., Osundare, O. S., Kedi, W. E., Okeleke, P. A., Ijoma, T. I., & Abdul-Azeez, O. Y. (2024). Evaluating the role of cloud integration in mobile

and desktop operating systems. International Journal of Management & Entrepreneurship Research, 6(8).

35. Tranquillini, A. (2022). Handling of mobile applications state using Conflict-Free Replicated Data Types.

36. Karunamurthy, A., Yuvaraj, M., Shahithya, J., & Thenmozhi, V. (2023). Cloud database: Empowering scalable and flexible data management. Quing: International Journal of Innovative Research in Science and Engineering.

37. Laitamäki, O. (2023). Web application architecture for real-time mobile network analysis.

38. Josephe, A. O., Chrysoulas, C., Peng, T., El Boudani, B., Iatropoulos, I., & Pitropakis, N. (2023, May). Progressive Web Apps to Support (Critical) Systems in Low or No Connectivity Areas. In 2023 IEEE IAS Global Conference on Emerging Technologies (GlobConET) (pp. 1-6). IEEE.

39. Abbasi, M., Lopes, A., Rodrigues, D., Saraiva, J., Martins, P., Sá, F., & Cardoso, F. (2023, June). In-depth analysis of mobile apps statistics: a study and development of a mobile app. In 2023 18th Iberian Conference on Information Systems and Technologies (CISTI) (pp. 1-7). IEEE.

40. Zhao, Y., Zhang, W., Zhou, L., & Cao, W. (2021). A survey on caching in mobile edge computing. Wireless Communications and Mobile Computing, 2021(1), 5565648.

41. Li, C., Liu, J., Zhang, Q., & Luo, Y. (2021). Efficient cooperative cache management for latency-aware data intelligent processing in edge environment. Future Generation Computer Systems, 123, 48-67.

42. Omowon, A. Technology and Conflict Resolution: Bridging Online and Offline Solutions.

43. Gambo, I., Massenon, R., Ogundokun, R. O., Agarwal, S., & Pak, W. (2024). Identifying and resolving conflict in mobile application features through contradictory feedback analysis. Heliyon, 10(17).

44. Охорона здоров’я та безпека праці в галузі IT: Важливі вимоги до роботодавців та працівників // Охорона праці і пожежна безпека. – 2023. – [Електронний ресурс]. – Режим доступу: https://oppb.com.ua/articles/ohorona-

zdorov-ya-ta-bezpeka-pratsi-v-galuzi-it-vazhlyvi-vymogy-do-robotodavtsiv-ta-pratsivnykiv

45. Основи охорони праці: навчальний посібник / [уклад. В.М. Жидецький та ін.]. – Львів: Афіша, 2020. – 360 с.

46. Крушельницька Я.В. Фізіологія і психологія праці: навч. посібник. – К.: КНЕУ, 2021. – 232 с.

47. Методичні вказівки з охорони праці та безпеки життєдіяльності / НУ "Одеська юридична академія". – 2022. – [Електронний ресурс]. – Режим доступу: https://dspace.onua.edu.ua/items/8c06a243-ee42-4454-bd8e-8e29e53a0633

48. ДСанПіН 3.3.2.007-98. Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин. – К.: МОЗ України, 1998.

49. ДСТУ 7299:2013 Дизайн і ергономіка. Робоче місце оператора. Взаємне розташування елементів робочого місця. Загальні вимоги ергономіки

50. Вплив електромагнітних полів (мобільні телефони, Wi-Fi мережі) // Буковинський державний медичний університет. – 2023. – [Електронний ресурс]. – Режим доступу: https://www.bsmu.edu.ua/blog/1930-vplyv-electromagnitnyh-poliv/

51. Охорона праці при роботі з електронними пристроями: Методичні рекомендації. – МОЗ України, 2022. – 34 с.

52. ДСНіП 239-96. Державні санітарні норми і правила захисту населення від впливу електромагнітних випромінювань. – К.: МОЗ України, 1996. (із змінами від 05.08.2023р.)

53. Головко Д.Ю. Особливості організації охорони праці в умовах воєнного стану в Україні: електронний навчальний курс. – Біла Церква: БІНПО ДЗВО «УМО» НАПН України, 2024. – 44 с.

54. Литвин А.Ф., Лаун С.Ю. Охорона праці на підприємствах в умовах воєнного стану // Матеріали конференції «CIMS 2023». – 2023.

55. Оцінка ризиків та розробка заходів з охорони праці: електронний навчальний курс. – Біла Церква: БІНПО ДЗВО «УМО» НАПН України, 2023. – 42 с.

56. Проблеми та перспективи забезпечення цивільного захисту: матеріали міжнародної науково-практичної конференції. – Харків: НУЦЗУ, 2024.

57. Необхідність іспитів з охорони праці в умовах воєнного стану // Державна служба України з питань праці. – 2024.

58. Навчально-методичні матеріали з охорони праці та безпеки життєдіяльності // КПІ ім. Ігоря Сікорського. – 2023.

59. Збірник наукових праць «Надзвичайні ситуації: безпека та захист». – Харків: НУЦЗУ, 2024.

60. Підсумкова інформація про діяльність Федерації профспілок України у 2023 році: протидія викликам і загрозам під час воєнного стану. – 2023.

61. Про найвагоміші зміни в чинному законодавстві про охорону праці, які відбулися наприкінці 2023-го та впродовж 2024 року. – 2024.

62. Інформація щодо рішень органів влади, зокрема, з питань бюджету, оплати праці та соціального захисту населення (09.04.2025) // Федерація профспілок України. – 2025.

63. Vasyl Dozosky, Oksana Dozorska, Vyacheslav Nykytyuk**,** EvheniaYavorska, Leonid Dediv. The Method of Selection and Pre-processing of ElectromyographicSignals for Bio-controlled Prosthetic of Hand. 2020 IEEE 15th International Scientific and Technical Conference onComputer Sciences and Information Technologies (CSIT). Volume 1, Lviv-Zbarazh, Ukraine 23-26 September 2020. P. 188-191*)* Electronic ISBN:978-1-7281-7443-3, USB ISBN:978-1-7281-7442-6, Print on Demand (PoD) ISBN:978-1-7281-7444-0. Print ISSN: 2766-3655, Online ISSN: 2766-3639. DOI: 10.1109/CSIT49958.2020.9321935.

64. Vyacheslav Nykytyuk, Vasyl Dozorskyy, Nataliia Kunanets, Volodymyr Pasichnyk, Oleksandr Matsiuk, Ihor Bodnarchuk: Electrical Probe-Signal Processing and Criterion for the Determination of Time Parameters of the Teeth Filling Material Polymerization Process in Dentistry. 4th IDDM 2021: Valencia, Spain. P. 54-63

65. Oleksii Duda, Nataliia Kunanets, Serhii Martsenko, Vyacheslav Nykytyuk**,** Volodymyr Pasichnyk. Information technology platform for the selection and analytical processing of information on COVID-19. 2021 IEEE 16th International

Conference on Computer Sciences and Information Technologies (CSIT). Volume 2, Lviv, Ukraine 22-25 Sept. 2021. P. 231-328. Electronic ISBN:978-1-6654-4257-2, Print on Demand(PoD) ISBN:978-1-6654-4258-9, Electronic ISSN: 2766-3639, Print on Demand(PoD) ISSN: 2766-3655. DOI: 10.1109/CSIT52700.2021.9648839.

66. Oleksii Duda, Nataliia Kunanets, Serhii Martsenko, Vyacheslav Nykytyuk, Volodymyr Pasichnyk. COVID-19 data collections and analytical processing. 2021 IEEE 16th International Conference on Computer Sciences and Information Technologies (CSIT). Volume 2, Lviv, Ukraine 22-25 Sept. 2021. P. 252-257. Electronic ISBN:978-1-6654-4257-2, Print on Demand (PoD) ISBN:978-1-6654-4258-9, Electronic ISSN: 2766-3639, Print on Demand (PoD) ISSN: 2766-3655. DOI: 10.1109/CSIT52700.2021.9648839.

67. Vyacheslav Nykytyuk, Vasil Dozorskyi, Oksana Dozorska, Andrii Karnaukhov and Liubomyr Matiichuk. The Method of User Identification by Speech Signal. The 2nd International Workshop on Information Technologies: Theoretical and Applied Problems (ITTAP-2022) Ternopil, Ukraine, November 22-24, 2022. Vol-3309 urn:nbn:de:0074-3309-1. P.225-232. ISSN 1613-0073 DOI: 10.1425/jsdtl.

68. Ihor Bodnarchuk, Yuriy Skorenkyy, Taras Kramar, Oleksii Duda and Vyacheslav Nykytyuk. Use of Analytical Hierarchy Process in Scenarios Design for a Digital Museum with XR components. The 2nd International Workshop on Information Technologies: Theoretical and Applied Problems (ITTAP-2022) Ternopil, Ukraine, November 22-24, 2022. Vol-3309 urn:nbn:de:0074-3309-1. P. 414-425. ISSN 1613-0073 DOI: 10.1425/jsdtl.

69. Kryazhych O., Itskovych V., Iushchenko K., Hrytsyshyna V., Bruvier D., Nykytyuk V., Bodnarchuk I. (2023) The use of abstract moore automaton to control the sensors of a service-oriented alarm and emergency notification network. Scientific Journal of TNTU (Tern.), vol 109, no 1, pp. 111–120. *ISSN 2522-4433*

70. Dediv, L., Dozorska, O., Kukuruza, V., Nykytyuk, V., Kovalyk, S. Computer Simulation Modeling of Voice Signals in the Matlab Environment for the Task of Computerized Diagnostic Systems Testing. The 1st International Workshop on "Computer information technologies in Industry 4.0" (CITI-2023) will be held in Ternopil, Ukraine, from June 14 to 16, 2023. The Workshop is organized by the Faculty

of Applied Information Technologies and Electrical Engineering of Ternopil Ivan Puluj National Technical University. 2023, 3468, pp. 257–262. Vol-3468 urn:nbn:de:0074-3468-8, ISSN 1613-0073

71. Dozorskyi, V., Dediv, I., Sverstiuk, S., Nykytyuk**, V.**, Karnaukhov, A. The Method of Commands Identification to Voice Control of the Electric Wheelchair. The Workshop is organized by the Faculty of Applied Information Technologies and Electrical Engineering of Ternopil Ivan Puluj National Technical University. The 1st International Workshop on "Computer information technologies in Industry 4.0" (CITI-2023) will be held in Ternopil, Ukraine, from June 14 to 16, 2023. The Workshop is organized by the Faculty of Applied Information Technologies and Electrical Engineering of Ternopil Ivan Puluj National Technical University. *2023*, 3468, pp. 233–240. Vol-3468 urn:nbn:de:0074-3468-8, ISSN 1613-0073

72. Sverstiuk, A., Matiichuk, L., Polyvana, U., Stanko, A., Nykytyuk, V**.**. Analytical analysis of approaches to assessing the quality of life in smart cities. BAIT'2024: The 1st International Workshop on "Bioinformatics and applied information technologies", October 02-04, 2024, Zboriv, Ukraine. CEUR Workshop Proceedings, 2024, 3842, pp. 75–91. ISSN: 1613-0073

73. Koroliuk, R., Nykytyuk, V**.**, Tymoshchuk, V., Soyka, V., Tymoshchuk, D.. Automated monitoring of bee colony movement in the hive during winter season. BAIT'2024: The 1st International Workshop on "Bioinformatics and applied information technologies", October 02-04, 2024, Zboriv, Ukraine. CEUR Workshop Proceedings, 2024, 3842, pp. 147–156. ISSN: 1613-0073

ДОДАТКИ

Додаток А

Наукові тези

UDC 001.1

The 12th International scientific and practical conference "Perspectives of contemporary science: theory and practice" (January 13-15, 2025) SPC "Sci-conf.com.ua", Lviv, Ukraine. 2025. 1429 p.

ISBN 978-966-8219-88-7

The recommended citation for this publication is:
Ivanov I. Analysis of the phaunistic composition of Ukraine // Perspectives of contemporary science: theory and practice. Proceedings of the 12th International scientific and practical conference. SPC "Sci-conf.com.ua". Lviv, Ukraine. 2025. Pp. 21-27. URL: https://sci-conf.com.ua/xii-mizhnarodna-naukovo-praktichna-konferentsiya-perspectives-of-contemporary-science-theory-and-practice-13-15-01-2025-lviv-ukrayina-arhiv/.

Editor
Komarytskyy M.L.
Ph.D. in Economics, Associate Professor

Collection of scientific articles published is the scientific and practical publication, which contains scientific articles of students, graduate students, Candidates and Doctors of Sciences, research workers and practitioners from Europe, Ukraine and from neighbouring countries and beyond. The articles contain the study, reflecting the processes and changes in the structure of modern science. The collection of scientific articles is for students, postgraduate students, doctoral candidates, teachers, researchers, practitioners and people interested in the trends of modern science development.

e-mail: lviv@sci-conf.com.ua

homepage: *https://sci-conf.com.ua*

ГІБРИДНИЙ ПІДХІД ДО ОБ'ЄДНАННЯ БАЗ ДАНИХ ДЛЯ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ СУЧАСНИХ ДОДАТКІВ

Шимків Владислав Сергійович,
Ципняк Денис Олександрович,
Студенти
Тернопільський національний технічний університет імені Івана Пулюя

Вступ / Introductions. Сучасні додатки вимагають обробки великих обсягів даних, що створює виклики для традиційних реляційних баз даних (SQL). Їх обмеження у масштабованості та швидкості обробки не завжди відповідають потребам складних розподілених систем. У той самий час NoSQL бази пропонують гнучкість і масштабованість, але часто поступаються у транзакційній консистентності. Об'єднання цих технологій у рамках гібридного підходу дозволяє використовувати переваги обох типів баз для підвищення ефективності сучасних додатків.

Мета роботи / Aim. Метою роботи є дослідження та розробка підходів до інтеграції SQL і NoSQL баз даних у рамках гібридної архітектури для забезпечення високої продуктивності та надійності сучасних додатків. У межах дослідження проведено аналіз ключових характеристик реляційних і нереляційних баз даних, а також визначено методи їхнього ефективного поєднання. Особлива увага приділяється створенню методики інтеграції, що дозволяє забезпечити оптимальне використання обох типів баз даних залежно від специфіки роботи додатка. Крім того, реалізовано експериментальне тестування запропонованої архітектури для оцінки її продуктивності та масштабованості.

Матеріали та методи / Materials and methods. У рамках дослідження розроблено багаторівневий підхід до інтеграції SQL і NoSQL баз даних. Запропонована архітектура дозволяє ефективно розподіляти завдання між базами даних: SQL використовується для обробки транзакційних даних, а NoSQL – для масштабованого зберігання та швидкого доступу до нереляційних

даних.

Результати та обговорення / Results and discussion. Експериментальне тестування продемонструвало покращення продуктивності системи на 30% у порівнянні з традиційними підходами. Запропонований гібридний підхід до інтеграції SQL і NoSQL баз даних дозволяє максимально ефективно використовувати переваги кожного типу баз. Це забезпечує високу продуктивність, надійність і масштабованість для сучасних додатків, що працюють із великими обсягами даних.

Висновки / Conclusions. Отримані результати можуть бути використані для побудови складних розподілених систем у різних галузях, таких як фінанси, електронна комерція та медіа.

СИСТЕМА АВТОМАТИЗОВАНОГО РЕЗЕРВНОГО КОПІЮВАННЯ ДАНИХ У ХМАРУ ДЛЯ МАЛОГО БІЗНЕСУ НА БАЗІ GOOGLE DRIVE API

Ципняк Денис Олександрович,
Шимків Владислав Сергійович,
Студенти
Тернопільський національний технічний університет
імені Івана Пулюя

Вступ / Introductions. У сучасних умовах малий бізнес часто стикається з ризиками втрати даних через апаратні збої, кібератаки або людські помилки. Автоматизовані системи резервного копіювання є важливим інструментом для забезпечення надійності та доступності інформації.

Хмарні сервіси, такі як Google Drive, пропонують зручні засоби для зберігання даних, однак потребують інтеграції та оптимізації для зменшення витрат. Використання машинного навчання (ML) дозволяє прогнозувати, коли і які дані потребують резервного копіювання, що забезпечує ефективніше управління ресурсами.

Запропонована система поєднує автоматизацію резервного копіювання з інтелектуальним аналізом даних для покращення безпеки та продуктивності.

Мета роботи / Aim. Метою роботи є розробка системи автоматизованого резервного копіювання даних для малого бізнесу з інтеграцією Google Drive API та використанням машинного навчання для прогнозування необхідності резервного копіювання.

Завдання включають забезпечення вибору файлів для копіювання, шифрування даних, стиснення файлів для економії простору та використання алгоритмів машинного навчання для визначення найбільш критичних моментів створення резервних копій.

Матеріали та методи / Materials and methods. Для реалізації системи було використано такі технології:

- Python — основна мова програмування.
- PyQt5 — для створення зручного графічного інтерфейсу користувача (GUI).
- Google Drive API — для інтеграції з хмарним сервісом.
- Pyzipper — для стиснення та шифрування даних.
- Tenacity — для обробки помилок і повторних спроб у разі збоїв.
- Scikit-learn — для побудови моделі машинного навчання.

Методологія реалізації:

1. Аналіз даних: Зібрано статистику змін та використання файлів.
2. Моделювання: Використано класифікаційні алгоритми (наприклад, Random Forest або Gradient Boosting) для прогнозування ймовірності необхідності резервного копіювання.
3. Інтеграція ML-моделі: Вбудовано прогнозну модель у систему, що дозволяє автоматично ініціювати резервне копіювання для критичних файлів.
4. Автоматизація: Розроблено механізми періодичного запуску моделі та резервного копіювання із заданими інтервалами.

Результати та обговорення / Results and discussion. Розроблена система демонструє такі переваги:

1. Інтелектуальне управління резервним копіюванням на основі прогнозів, зменшуючи витрати на зберігання малозначущих файлів.
2. Стиснення файлів у формат ZIP з шифруванням, що підвищує безпеку даних.
3. Автоматизація резервного копіювання, яка враховує динаміку використання файлів.
4. Прогнозна модель з точністю 92%, яка визначає найбільш критичні моменти для створення резервних копій.

Система була протестована на реальних даних, і її використання дозволило знизити витрати на зберігання даних у хмарі на 25% та забезпечити високу продуктивність процесів резервного копіювання.

Висновки / Conclusions. Розроблена система об'єднує автоматизацію резервного копіювання та інтелектуальний аналіз даних для підвищення ефективності та зручності використання. Використання машинного навчання дозволяє адаптувати процес резервного копіювання до реальних потреб бізнесу, забезпечуючи збереження лише критично важливих даних. Такий підхід може бути корисним у різних галузях, включаючи фінанси, медицину та електронну комерцію.

Додаток Б

Лістинги програмної реалізації Android-застосунку для управління нотатками з інтеграцією об`єднання SQL (Room), NoSQL (Firebase Firestore) та сервісу Google Drive

Лістинг Б.1 – AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/an-
droid"
    xmlns:tools="http://schemas.android.com/tools"
    package="com.naukova.backup">
    <!-- Дозвола -->
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.GET_AC-
COUNTS" />
    <uses-permission android:name="android.permission.READ_EXTER-
NAL_STORAGE" />
    <uses-permission android:name="android.permission.WRITE_EXTER-
NAL_STORAGE" />
    <uses-permission android:name="android.permission.MANAGE_EX-
TERNAL_STORAGE"
        android:maxSdkVersion="29" />
    <uses-permission android:name="android.permission.READ_EXTER-
NAL_STORAGE" />
    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/Theme.Naukova"
        tools:targetApi="31">
        <!-- Основна активність -->
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"
/>
                <category android:name="android.intent.cate-
gory.LAUNCHER" />
            </intent-filter>
        </activity>

        <!-- Створення нотатки -->
        <activity
            android:name=".ui.NoteCreateActivity"
            android:exported="false" />
```

```
        <!-- Інша активність (якщо потрібно) -->
        <activity
            android:name=".BackupSelectionActivity"
            android:exported="true" />

        <activity
           android:name="com.naukova.backup.viewmodel.NoteEditAc-
tivity"
            android:exported="false" />
    </application>
</manifest>
```

Лістинг Б.2 – NoteAdapter

```
package com.naukova.backup.adapter
import android.content.Intent
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView
import com.naukova.backup.R
import com.naukova.backup.data.Note
import com.naukova.backup.viewmodel.NoteEditActivity
import java.text.SimpleDateFormat
import java.util.*

class NoteAdapter(
        private var notes: List<Note>,
        private val onNoteLongClick: (Note) -> Boolean
) : RecyclerView.Adapter<NoteAdapter.NoteViewHolder>() {

    inner class NoteViewHolder(itemView: View) : Recycler-
View.ViewHolder(itemView) {
        val textTitle: TextView = itemView.findView-
ById(R.id.textTitle)
        val textContent: TextView = itemView.findView-
ById(R.id.textContent)
        val textTimestamp: TextView = itemView.findView-
ById(R.id.textTimestamp)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType:
Int): NoteViewHolder {
        val view = LayoutInflater.from(parent.context)
                .inflate(R.layout.item_note, parent, false)
        return NoteViewHolder(view)
    }

    override fun onBindViewHolder(holder: NoteViewHolder, posi-
tion: Int) {
        val note = notes[position]
        holder.textTitle.text = note.title
        holder.textContent.text = note.content
```

```
        val dateFormat = SimpleDateFormat("dd.MM.yyyy HH:mm", Lo-
cale.getDefault())
        val formattedDate = dateFormat.format(Date(note.up-
datedAt))
        holder.textTimestamp.text = formattedDate

        // Клік — редагування
        holder.itemView.setOnClickListener {
            val context = holder.itemView.context
            val intent = Intent(context, NoteEditActiv-
ity::class.java).apply {
                putExtra("noteId", note.id)
                putExtra("title", note.title)
                putExtra("content", note.content)
            }
            context.startActivity(intent)
        }

        // Довгий клік — видалення або інша дія
        holder.itemView.setOnLongClickListener {
            onNoteLongClick(note)
        }
    }
    override fun getItemCount(): Int = notes.size
    fun updateData(newNotes: List<Note>) {
        notes = newNotes
        notifyDataSetChanged()
    }
}
```

Лістинг Б.3 – GoogleDriveServiceHelper.java

```
package com.naukova.backup;
import android.content.Context;
import com.google.api.client.googleapis.auth.oauth2.GoogleCreden-
tial;
import com.google.api.client.http.HttpTransport;
import com.google.api.client.http.javanet.NetHttpTransport;
import com.google.api.client.json.JsonFactory;
import com.google.api.client.json.gson.GsonFactory;
import com.google.api.services.drive.Drive;
import com.google.api.services.drive.DriveScopes;
import java.io.IOException;
import java.io.InputStream;
import java.util.Collections;
import com.naukova.backup.R;
public class GoogleDriveServiceHelper {
    private static final String APPLICATION_NAME = "Nauko-
vaBackup";
    private static final JsonFactory JSON_FACTORY = GsonFac-
tory.getDefaultInstance();
    public static Drive getDriveService(Context context) throws
IOException {
        HttpTransport transport = new NetHttpTransport();
```

```
        // Завантажує client_secret.json коректним способом
        InputStream inputStream = context.getAssets().open("cli-
ent_secret.json");
        GoogleCredential credential = GoogleCredential.from-
Stream(inputStream)
                .createScoped(Collections.singleton-
List(DriveScopes.DRIVE_FILE));
        return new Drive.Builder(transport, JSON_FACTORY,
 credential)
                .setApplicationName(APPLICATION_NAME)
                .build();
    }
}
```

Лістинг Б.4 – GoogleDriveUploader.java

```
package com.naukova.backup
import android.util.Log
import com.google.api.services.drive.Drive
import com.google.api.services.drive.model.File
import com.google.api.client.http.FileContent
import com.google.android.gms.tasks.Task
import com.google.android.gms.tasks.Tasks
class GoogleDriveUploader(private val mDriveService: Drive) {
    fun uploadFile(localFile: java.io.File, mimeType: String,
fileName: String): Task<String> {
        return Tasks.call {
            val metadata = File()
            metadata.name = fileName
            val fileContent = FileContent(mimeType, localFile)
            val file = mDriveService.files().create(metadata,
fileContent)
                    .setFields("id")
                    .execute()
            Log.d("Backup", "Файл завантажено, ID: ${file.id}")
            file.id
        }
    }
}
```

Лістинг Б.5 – MainActivity.java

```
package com.naukova.backup;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.view.animation.AnimationUtils;
import android.widget.ProgressBar;
import android.widget.TextView;
import android.widget.Toast;
import androidx.activity.result.ActivityResultLauncher;
import androidx.activity.result.contract.ActivityResultContracts;
import androidx.appcompat.app.AlertDialog;
import androidx.appcompat.app.AppCompatActivity;
import androidx.lifecycle.ViewModelProvider;
```

```
import androidx.recyclerview.widget.LinearLayoutManager;
import androidx.recyclerview.widget.RecyclerView;
import com.google.android.gms.auth.api.signin.*;
import com.google.android.gms.tasks.Task;
import com.google.android.material.floatingactionbutton.Float-
ingActionButton;
import com.google.api.client.googleapis.extensions.an-
droid.gms.auth.GoogleAccountCredential;
import com.google.api.client.http.FileContent;
import com.google.api.client.http.javanet.NetHttpTransport;
import com.google.api.client.json.JsonFactory;
import com.google.api.client.json.gson.GsonFactory;
import com.google.api.services.drive.Drive;
import com.google.api.services.drive.DriveScopes;
import com.naukova.backup.adapter.NoteAdapter;
import com.naukova.backup.data.AppDatabase;
import com.naukova.backup.data.Note;
import com.naukova.backup.viewmodel.NoteViewModel;

import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.security.GeneralSecurityException;
import java.util.*;

public class MainActivity extends AppCompatActivity {

    private static final JsonFactory JSON_FACTORY = GsonFac-
tory.getDefaultInstance();

    private GoogleSignInClient googleSignInClient;
    private Drive driveService;
    private NoteViewModel noteViewModel;
    private RecyclerView recyclerView;
    private NoteAdapter adapter;
    private ProgressBar progressBar;
    private final List<Note> noteList = new ArrayList<>();
    private boolean isRestoreTriggered = false;
    private boolean isSignInChecked = false;

    private final ActivityResultLauncher<Intent> signInLauncher =
registerForActivityResult(
            new ActivityResultContracts.StartActivityForResult(),
            result -> {
                if (result.getResultCode() == RESULT_OK && re-
sult.getData() != null) {
                    Task<GoogleSignInAccount> task = Google-
SignIn.getSignedInAccountFromIntent(result.getData());
                    if (task.isSuccessful()) {
                        GoogleSignInAccount account =
task.getResult();
                        try {
                            driveService = createDriveService(ac-
count);
```

```
                            Toast.makeText(this, "Успішно
підключено до Google Drive!", Toast.LENGTH_SHORT).show();
                            triggerRestoreIfNeeded();
                        } catch (Exception e) {
                            Toast.makeText(this, "Помилка
підключення до Drive: " + e.getMessage(),
Toast.LENGTH_LONG).show();
                        }
                    } else {
                        Toast.makeText(this, "Авторизація не
вдалася", Toast.LENGTH_SHORT).show();
                    }
                }
            });
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);
        noteViewModel = new ViewModelProvider(this).get(NoteView-
Model.class);
        initViews();
        setupRecyclerView();
        observeNotes();
        initListeners();
        setupGoogleSignIn();
    }
    private void initViews() {
        progressBar = findViewById(R.id.progressBar);
        recyclerView = findViewById(R.id.recyclerView);
    }
    private void initListeners() {
        findViewById(R.id.sign_in_button).setOnClickListener(v ->
                signInLauncher.launch(googleSignInCli-
ent.getSignInIntent()));
        findViewById(R.id.btnBackup).setOnClickListener(v -> up-
loadRoomDatabaseToDrive());
        findViewById(R.id.btnRestore).setOnClickListener(v -> {
            isRestoreTriggered = false;
            restoreDatabaseFromDrive();
        });
        findViewById(R.id.btnSyncToFirestore).setOnClickListener(v
-> {
            noteViewModel.syncToFirestore();
            Toast.makeText(this, "Синхронізація в Firestore
запущена", Toast.LENGTH_SHORT).show();});
        FloatingActionButton fab = findViewById(R.id.fabAddNote);
        fab.setOnClickListener(v -> startActivity(new Intent(this,
com.naukova.backup.ui.NoteCreateActivity.class)));
    }
    private void setupGoogleSignIn() {
        googleSignInClient = GoogleSignIn.getClient(this,
                new GoogleSignInOptions.Builder(GoogleSignInOp-
tions.DEFAULT_SIGN_IN)
                        .requestEmail()
```

```
                        .requestScopes(new com.google.an-
droid.gms.common.api.Scope(DriveScopes.DRIVE_FILE))
                        .build());
    }

    private void setupRecyclerView() {
        adapter = new NoteAdapter(new ArrayList<>(), note -> {
            new AlertDialog.Builder(MainActivity.this)
                    .setTitle("Видалити нотатку?")
                    .setMessage("Ви впевнені, що хочете видалити
цю нотатку?")
                    .setPositiveButton("Так", (dialog, which) ->
deleteNote(note.getId()))
                    .setNegativeButton("Скасувати", null)
                    .show();
            return true;
        });
        recyclerView.setLayoutManager(new LinearLayoutMan-
ager(this));
        recyclerView.setAdapter(adapter);
    }

    private void observeNotes() {
        noteViewModel.getAllNotes().removeObservers(this);
        showProgressBar();
        noteViewModel.getAllNotes().observe(this, notes -> {
            noteList.clear();
            noteList.addAll(notes);
            adapter.updateData(noteList);
            hideProgressBar();
        });
    }

    private void showProgressBar() {
        runOnUiThread(() -> {
            if (progressBar.getVisibility() != View.VISIBLE) {
                progressBar.setAlpha(0f);
                progressBar.setVisibility(View.VISIBLE);
                progressBar.animate().alpha(1f).setDura-
tion(300).start();
            }
        });
    }

    private void hideProgressBar() {
        runOnUiThread(() -> {
            if (progressBar.getVisibility() == View.VISIBLE) {
                progressBar.animate().alpha(0f).setDuration(300)
                        .withEndAction(() -> progressBar.setVisi-
bility(View.GONE)).start();
            }
        });
    }
    @Override
    protected void onStart() {
```

```
        super.onStart();
        if (!isSignInChecked) {
            isSignInChecked = true;
            GoogleSignInAccount account = GoogleSignIn.get-
LastSignedInAccount(this);
            if (account != null && driveService == null) {

                try {
                    driveService = createDriveService(account);
                    triggerRestoreIfNeeded();
                } catch (GeneralSecurityException | IOException e)
{
                    e.printStackTrace();
                    Toast.makeText(this, "Помилка підключення до
Drive: " + e.getMessage(), Toast.LENGTH_LONG).show();
                }
            }
        }
    }
    private void triggerRestoreIfNeeded() {
        if (!isRestoreTriggered && !isDatabaseRestored()) {
            isRestoreTriggered = true;
            restoreDatabaseFromDrive();
        }
    }
    private void setDatabaseRestored(boolean restored) {
        getSharedPreferences("app_prefs", MODE_PRIVATE)
                .edit()
                .putBoolean("database_restored", restored)
                .apply();
    }
    private boolean isDatabaseRestored() {
        return getSharedPreferences("app_prefs", MODE_PRIVATE)
                .getBoolean("database_restored", false);
    }
    public void deleteNote(String noteId) {
        new Thread(() -> {
            try {
                AppDatabase.getDatabase(getApplicationCon-
text()).noteDao().deleteByIdSync(noteId);
                com.google.firebase.firestore.FirebaseFire-
store.getInstance()
                        .collection("notes")
                        .document(noteId)
                        .delete()
                        .addOnSuccessListener(aVoid ->
                                runOnUiThread(() ->
Toast.makeText(this, "Нотатку видалено",
Toast.LENGTH_SHORT).show()))
                        .addOnFailureListener(e ->
                                runOnUiThread(() ->
Toast.makeText(this, "Помилка при видаленні",
Toast.LENGTH_SHORT).show()));
            } catch (Exception e) {
                e.printStackTrace();
```

```
            }
        }).start();
    }
    private Drive createDriveService(GoogleSignInAccount account)
throws GeneralSecurityException, IOException {
        GoogleAccountCredential credential = GoogleAccountCreden-
tial.usingOAuth2(this, Collections.single-
ton(DriveScopes.DRIVE_FILE));
        credential.setSelectedAccount(account.getAccount());
        return new Drive.Builder(new NetHttpTransport(), JSON_FAC-
TORY, credential)
                .setApplicationName("NaukovaBackup")
                .build();
    }
    private void showStatusMessage(String message) {
        TextView statusView = findViewById(R.id.statusMessage);
        statusView.setText(message);
        statusView.setAlpha(0f);
        statusView.setVisibility(View.VISIBLE);
        statusView.animate().alpha(1f).setDuration(300).start();
    }
    private void hideStatusMessage() {
        TextView statusView = findViewById(R.id.statusMessage);
        statusView.animate().alpha(0f).setDuration(300)
                .withEndAction(() -> statusView.setVisibil-
ity(View.GONE)).start();
    }
    private void uploadRoomDatabaseToDrive() {
        File dbFile = getDatabasePath("note_database");
        if (!dbFile.exists()) {
            Toast.makeText(this, "База даних не знайдена",
Toast.LENGTH_SHORT).show();
            return;
        }
        FloatingActionButton fab = findViewById(R.id.fabAddNote);

        runOnUiThread(() -> {
            fab.startAnimation(AnimationUtils.loadAnimation(this,
R.anim.rotate));
            showStatusMessage("Збереження резервної копії...");
            showProgressBar();
        });
        new Thread(() -> {
            try {
                AppDatabase.getDatabase(getApplicationCon-
text()).close();
                AppDatabase.closeDatabase();
                Thread.sleep(1000);

                FileContent mediaContent = new FileContent("appli-
cation/octet-stream", dbFile);
                com.google.api.services.drive.model.File body =
new com.google.api.services.drive.model.File();
                body.setName("note_database_backup.db");
                body.setMimeType("application/octet-stream");
```

```
                com.google.api.services.drive.model.File file =
driveService.files()
                        .create(body, mediaContent)
                        .setFields("id")
                        .execute();

                runOnUiThread(() -> {
                    hideProgressBar();
                    fab.clearAnimation();
                    hideStatusMessage();
                    Toast.makeText(this, "Бекап збережено в Drive
(ID: " + file.getId() + ")", Toast.LENGTH_LONG).show();
                    AppDatabase.getDatabase(getApplicationCon-
text());
                });
            } catch (Exception e) {
                runOnUiThread(() -> {
                    hideProgressBar();
                    fab.clearAnimation();
                    hideStatusMessage();
                    Toast.makeText(this, "Помилка при
завантаженні: " + e.getMessage(), Toast.LENGTH_LONG).show();
                });
                e.printStackTrace();
            }
        }).start();
    }

    private void restoreDatabaseFromDrive() {
        if (driveService == null) {
            Toast.makeText(this, "Спочатку увійдіть у Google",
Toast.LENGTH_SHORT).show();
            return;
        }
        new Thread(() -> {
  try {

 String fileName = "note_database_backup.db";
                com.google.api.services.drive.model.FileList re-
sult = driveService.files().list()
                        .setQ("name = '" + fileName + "'")
                        .setFields("files(id, name)")
                        .execute();

                if (result.getFiles().isEmpty()) {
                    runOnUiThread(() -> Toast.makeText(this, "Файл
резервної копії не знайдено", Toast.LENGTH_SHORT).show());
                    return;
                }

                String fileId = result.getFiles().get(0).getId();
                File localDbFile = getDatabasePath("note_data-
base");
```

```
                driveService.files().get(fileId)
                        .executeMediaAndDownloadTo(new FileOut-
putStream(localDbFile));

                runOnUiThread(() -> {
                    setDatabaseRestored(true);
                    AppDatabase.closeDatabase();
                    AppDatabase.getDatabase(getApplicationCon-
text());

                    // 🕒 Затримка перед підпискою на LiveData
                    new android.os.Handler(android.os.Looper.get-
MainLooper()).postDelayed(() -> {
                        noteViewModel.reloadNotes();
                        noteViewModel.refreshLiveData();
                        noteViewModel.getAllNotes().removeObserv-
ers(this);
                        observeNotes();

                        Toast.makeText(this, "Базу даних успішно
відновлено!", Toast.LENGTH_SHORT).show();
                    }, 500); // ⏱ 500 мс затримка
                });

            } catch (Exception e) {
                e.printStackTrace();
                runOnUiThread(() -> Toast.makeText(this, "Помилка
при відновленні: " + e.getMessage(), Toast.LENGTH_LONG).show());
            }
```

Продовження Лістинг 5 – MainActivity.java

```
        }).start();
    }
}
```

Лістинг Б.6 – AppDatabase_Impi.java

```
package com.naukova.backup.data;
import androidx.annotation.NonNull;
import androidx.room.DatabaseConfiguration;
import androidx.room.InvalidationTracker;
import androidx.room.RoomDatabase;
import androidx.room.RoomOpenHelper;
import androidx.room.migration.AutoMigrationSpec;
import androidx.room.migration.Migration;
import androidx.room.util.DBUtil;
import androidx.room.util.TableInfo;
import androidx.sqlite.db.SupportSQLiteDatabase;
import androidx.sqlite.db.SupportSQLiteOpenHelper;
import java.lang.Class;
import java.lang.Override;
import java.lang.String;
import java.lang.SuppressWarnings;
import java.util.ArrayList;
```

```
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;
import javax.annotation.processing.Generated;

@Generated("androidx.room.RoomProcessor")
@SuppressWarnings({"unchecked", "deprecation"})
public final class AppDatabase_Impl extends AppDatabase {
  private volatile NoteDao _noteDao;

  @Override
  @NonNull
  protected SupportSQLiteOpenHelper createOpenHelper(@NonNull fi-
nal DatabaseConfiguration config) {
    final SupportSQLiteOpenHelper.Callback _openCallback = new
RoomOpenHelper(config, new RoomOpenHelper.Delegate(2) {
      @Override
      public void createAllTables(@NonNull final Sup-
portSQLiteDatabase db) {
        db.execSQL("CREATE TABLE IF NOT EXISTS `notes` (`id` TEXT
NOT NULL, `title` TEXT NOT NULL, `content` TEXT NOT NULL, `up-
datedAt` INTEGER NOT NULL, PRIMARY KEY(`id`))");
        db.execSQL("CREATE TABLE IF NOT EXISTS room_master_table
(id INTEGER PRIMARY KEY,identity_hash TEXT)");
        db.execSQL("INSERT OR REPLACE INTO room_master_table
(id,identity_hash) VALUES(42,
'f87ddf656bc7a9bc10d3436e469828f8')");
      }
      @Override
      public void dropAllTables(@NonNull final SupportSQLiteData-
base db) {
        db.execSQL("DROP TABLE IF EXISTS `notes`");
        final List<? extends RoomDatabase.Callback> _callbacks =
mCallbacks;

        if (_callbacks != null) {
          for (RoomDatabase.Callback _callback : _callbacks) {
            _callback.onDestructiveMigration(db);
          }
        }
      }

      @Override
      public void onCreate(@NonNull final SupportSQLiteDatabase
db) {
        final List<? extends RoomDatabase.Callback> _callbacks =
mCallbacks;
        if (_callbacks != null) {
          for (RoomDatabase.Callback _callback : _callbacks) {
            _callback.onCreate(db);
          }
        }
      }
```

```
      @Override
      public void onOpen(@NonNull final SupportSQLiteDatabase db)
{
        mDatabase = db;
        internalInitInvalidationTracker(db);
        final List<? extends RoomDatabase.Callback> _callbacks =
mCallbacks;
        if (_callbacks != null) {
          for (RoomDatabase.Callback _callback : _callbacks) {
            _callback.onOpen(db);
          }
        }
      }
      @Override
      public void onPreMigrate(@NonNull final
 SupportSQLiteDatabase db) {
        DBUtil.dropFtsSyncTriggers(db);
      }

      @Override
      public void onPostMigrate(@NonNull final SupportSQLiteData-
base db) {
      }

      @Override
      @NonNull
      public RoomOpenHelper.ValidationResult onValidateSchema(
          @NonNull final SupportSQLiteDatabase db) {
        final HashMap<String, TableInfo.Column> _columnsNotes =
new HashMap<String, TableInfo.Column>(4);
        _columnsNotes.put("id", new TableInfo.Column("id", "TEXT",
true, 1, null, TableInfo.CREATED_FROM_ENTITY));
        _columnsNotes.put("title", new TableInfo.Column("title",
"TEXT", true, 0, null, TableInfo.CREATED_FROM_ENTITY));
        _columnsNotes.put("content", new TableInfo.Column("con-
tent", "TEXT", true, 0, null, TableInfo.CREATED_FROM_ENTITY));
        _columnsNotes.put("updatedAt", new TableInfo.Column("up-
datedAt", "INTEGER", true, 0, null, TableInfo.CREATED_FROM_EN-
TITY));
        final HashSet<TableInfo.ForeignKey> _foreignKeysNotes =
new HashSet<TableInfo.ForeignKey>(0);
        final HashSet<TableInfo.Index> _indicesNotes = new
HashSet<TableInfo.Index>(0);
        final TableInfo _infoNotes = new TableInfo("notes", _col-
umnsNotes, _foreignKeysNotes, _indicesNotes);
        final TableInfo _existingNotes = TableInfo.read(db,
"notes");
        if (!_infoNotes.equals(_existingNotes)) {
          return new RoomOpenHelper.ValidationResult(false,
"notes(com.naukova.backup.data.Note).\n"

                  + " Expected:\n" + _infoNotes + "\n"
                  + " Found:\n" + _existingNotes);
        }
```

```
        return new RoomOpenHelper.ValidationResult(true, null);
      }
    }, "f87ddf656bc7a9bc10d3436e469828f8",
"53f806390b12c8dec52f94cded276cbb");
    final SupportSQLiteOpenHelper.Configuration _sqliteConfig =
SupportSQLiteOpenHelper.Configuration.builder(config.con-
text).name(config.name).callback(_openCallback).build();
    final SupportSQLiteOpenHelper _helper = config.sqliteOpenHelp-
erFactory.create(_sqliteConfig);
    return _helper;
  }
  @Override
  @NonNull
  protected InvalidationTracker createInvalidationTracker() {
    final HashMap<String, String> _shadowTablesMap = new
HashMap<String, String>(0);
    final HashMap<String, Set<String>> _viewTables = new
HashMap<String, Set<String>>(0);
    return new InvalidationTracker(this, _shadowTablesMap, _view-
Tables, "notes");
  }

  @Override
  public void clearAllTables() {
    super.assertNotMainThread();
    final SupportSQLiteDatabase _db = super.get-
OpenHelper().getWritableDatabase();
    try {
      super.beginTransaction();
      _db.execSQL("DELETE FROM `notes`");
      super.setTransactionSuccessful();

    } finally {
      super.endTransaction();
      _db.query("PRAGMA wal_checkpoint(FULL)").close();
      if (!_db.inTransaction()) {
        _db.execSQL("VACUUM");
      }
    }
  }

  @Override
  @NonNull
  protected Map<Class<?>, List<Class<?>>> getRequiredTypeConvert-
ers() {
    final HashMap<Class<?>, List<Class<?>>> _typeConvertersMap =
new HashMap<Class<?>, List<Class<?>>>();
    _typeConvertersMap.put(NoteDao.class,
NoteDao_Impl.getRequiredConverters());
    return _typeConvertersMap;
  }

  @Override
  @NonNull
```

```
  public Set<Class<? extends AutoMigrationSpec>> getRequiredAu-
toMigrationSpecs() {
    final HashSet<Class<? extends AutoMigrationSpec>> _autoMigra-
tionSpecsSet = new HashSet<Class<? extends AutoMigrationSpec>>();
    return _autoMigrationSpecsSet;
  }

  @Override
  @NonNull
  public List<Migration> getAutoMigrations(
      @NonNull final Map<Class<? extends AutoMigrationSpec>, Au-
toMigrationSpec> autoMigrationSpecs) {
    final List<Migration> _autoMigrations = new ArrayList<Migra-
tion>();
    return _autoMigrations;
  }

  @Override
  public NoteDao noteDao() {
    if (_noteDao != null) {
      return _noteDao;
    } else {
      synchronized(this) {
        if(_noteDao == null) {
          _noteDao = new NoteDao_Impl(this);
        }
        return _noteDao;
      }
    }
  }
}
```

Лістинг Б.7 – NoteDao_Impi.java

```
package com.naukova.backup.data;

import android.database.Cursor;
import android.os.CancellationSignal;
import androidx.annotation.NonNull;
import androidx.annotation.Nullable;
import androidx.lifecycle.LiveData;
import androidx.room.CoroutinesRoom;
import androidx.room.EntityDeletionOrUpdateAdapter;
import androidx.room.EntityInsertionAdapter;
import androidx.room.RoomDatabase;
import androidx.room.RoomSQLiteQuery;
import androidx.room.SharedSQLiteStatement;
import androidx.room.util.CursorUtil;
import androidx.room.util.DBUtil;
import androidx.sqlite.db.SupportSQLiteStatement;
import java.lang.Class;
import java.lang.Exception;
import java.lang.Object;
import java.lang.Override;
import java.lang.String;
```

```
import java.lang.SuppressWarnings;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.concurrent.Callable;
import javax.annotation.processing.Generated;
import kotlin.Unit;
import kotlin.coroutines.Continuation;

@Generated("androidx.room.RoomProcessor")
@SuppressWarnings({"unchecked", "deprecation"})
public final class NoteDao_Impl implements NoteDao {
  private final RoomDatabase __db;

  private final EntityInsertionAdapter<Note> __insertionAdapter-
OfNote;

  private final EntityDeletionOrUpdateAdapter<Note>
 __deletionAdapterOfNote;

  private final SharedSQLiteStatement __preparedStmtOfDeleteBy-
IdSync;

  public NoteDao_Impl(@NonNull final RoomDatabase __db) {
    this.__db = __db;
    this.__insertionAdapterOfNote = new EntityInsertionA-
dapter<Note>(__db) {
      @Override
      @NonNull
      protected String createQuery() {
        return "INSERT OR REPLACE INTO `notes` (`id`,`title`,`con-
tent`,`updatedAt`) VALUES (?,?,?,?)";
      }

      @Override
      protected void bind(@NonNull final SupportSQLiteStatement
statement,
          @NonNull final Note entity) {
        if (entity.getId() == null) {
          statement.bindNull(1);
        } else {
          statement.bindString(1, entity.getId());
        }
        if (entity.getTitle() == null) {
          statement.bindNull(2);
        } else {
          statement.bindString(2, entity.getTitle());
        }
        if (entity.getContent() == null) {
          statement.bindNull(3);
        } else {
          statement.bindString(3, entity.getContent());
        }
        statement.bindLong(4, entity.getUpdatedAt());
      }
```

```
    };
    this.__deletionAdapterOfNote = new EntityDeletionOrUpdateA-
dapter<Note>(__db) {
      @Override
      @NonNull
      protected String createQuery() {
        return "DELETE FROM `notes` WHERE `id` = ?";
      }      @Override
      protected void bind(@NonNull final SupportSQLiteStatement
statement,
          @NonNull final Note entity) {
        if (entity.getId() == null) {
          statement.bindNull(1);
        } else {
          statement.bindString(1, entity.getId());
        }
      }
    };
    this.__preparedStmtOfDeleteByIdSync = new SharedSQLiteState-
ment(__db) {
      @Override
      @NonNull
      public String createQuery() {
        final String _query = "DELETE FROM notes WHERE id = ?";
        return _query;
      }
    };
  }

  @Override
  public Object insert(final Note note, final Continuation<? super
Unit> $completion) {
    return CoroutinesRoom.execute(__db, true, new Callable<Unit>()
{
      @Override
      @NonNull
      public Unit call() throws Exception {
        __db.beginTransaction();
        try {
          __insertionAdapterOfNote.insert(note);
          __db.setTransactionSuccessful();
          return Unit.INSTANCE;
        } finally {
          __db.endTransaction();
        }
      }
    }, $completion);
  }

  @Override
  public Object insertAll(final List<Note> notes, final Continua-
tion<? super Unit> $completion) {
    return CoroutinesRoom.execute(__db, true, new Callable<Unit>()
{
      @Override
```

```
      @NonNull
      public Unit call() throws Exception {
        __db.beginTransaction();
        try {
          __insertionAdapterOfNote.insert(notes);
          __db.setTransactionSuccessful();
          return Unit.INSTANCE;
        } finally {
          __db.endTransaction();
        }
      }
    }, $completion);
  }

  @Override
  public Object insertOrUpdate(final Note note, final Continua-
tion<? super Unit> $completion) {
    return CoroutinesRoom.execute(__db, true, new Callable<Unit>()
{
      @Override
      @NonNull
      public Unit call() throws Exception {
        __db.beginTransaction();
        try {
          __insertionAdapterOfNote.insert(note);
          __db.setTransactionSuccessful();
          return Unit.INSTANCE;
        } finally {
          __db.endTransaction();
        }
      }
    }, $completion);
  }

  @Override
  public Object delete(final Note note, final Continuation<? super
Unit> $completion) {
    return CoroutinesRoom.execute(__db, true, new Callable<Unit>()
{
      @Override
      @NonNull
      public Unit call() throws Exception {
        __db.beginTransaction();
        try {
          __deletionAdapterOfNote.handle(note);
          __db.setTransactionSuccessful();
          return Unit.INSTANCE;
        } finally {
          __db.endTransaction();
        }
      }
    }, $completion);
  }

  @Override
```

```
  public void deleteByIdSync(final String noteId) {
    __db.assertNotSuspendingTransaction();
    final SupportSQLiteStatement _stmt = __preparedStmtOfDeleteBy-
IdSync.acquire();
    int _argIndex = 1;
    if (noteId == null) {
      _stmt.bindNull(_argIndex);
    } else {
      _stmt.bindString(_argIndex, noteId);
    }
    try {
      __db.beginTransaction();
      try {
        _stmt.executeUpdateDelete();
        __db.setTransactionSuccessful();
      } finally {
        __db.endTransaction();
      }
    } finally {
      __preparedStmtOfDeleteByIdSync.release(_stmt);
    }
  }

  @Override
  public LiveData<List<Note>> getAllNotes() {
    final String _sql = "SELECT * FROM notes ORDER BY updatedAt
DESC";
    final RoomSQLiteQuery _statement = RoomSQLiteQuery.ac-
quire(_sql, 0);
    return __db.getInvalidationTracker().createLiveData(new
String[] {"notes"}, false, new Callable<List<Note>>() {
      @Override
      @Nullable
      public List<Note> call() throws Exception {
        final Cursor _cursor = DBUtil.query(__db, _statement,
false, null);
        try {
          final int _cursorIndexOfId = CursorUtil.getColumnIndex-
OrThrow(_cursor, "id");
          final int _cursorIndexOfTitle = CursorUtil.getColumn-
IndexOrThrow(_cursor, "title");
          final int _cursorIndexOfContent = CursorUtil.getColumn-
IndexOrThrow(_cursor, "content");
          final int _cursorIndexOfUpdatedAt =
CursorUtil.getColumnIndexOrThrow(_cursor, "updatedAt");
          final List<Note> _result = new ArrayList<Note>(_cur-
sor.getCount());
          while (_cursor.moveToNext()) {
            final Note _item;
            final String _tmpId;
            if (_cursor.isNull(_cursorIndexOfId)) {
              _tmpId = null;
            } else {
              _tmpId = _cursor.getString(_cursorIndexOfId);
            }
```

```
            final String _tmpTitle;
            if (_cursor.isNull(_cursorIndexOfTitle)) {
              _tmpTitle = null;
            } else {
              _tmpTitle = _cursor.getString(_cursorIndexOfTitle);
            }
            final String _tmpContent;
            if (_cursor.isNull(_cursorIndexOfContent)) {
              _tmpContent = null;
            } else {
              _tmpContent =
_cursor.getString(_cursorIndexOfContent);
            }
            final long _tmpUpdatedAt;
            _tmpUpdatedAt = _cursor.getLong(_cursorIndexOf-
UpdatedAt);
            _item = new Note(_tmpId,_tmpTitle,_tmpCon-
tent,_tmpUpdatedAt);
            _result.add(_item);
          }
          return _result;
        } finally {
          _cursor.close();        } }

      @Override
      protected void finalize() {
        _statement.release();
      }
    });
  }

  @Override
  public Object getAllNow(final Continuation<? super List<Note>>
$completion) {
    final String _sql = "SELECT * FROM notes";
    final RoomSQLiteQuery _statement = RoomSQLiteQuery.ac-
quire(_sql, 0);
    final CancellationSignal _cancellationSignal = DBUtil.create-
CancellationSignal();
    return CoroutinesRoom.execute(__db, false, _cancellation-
Signal, new Callable<List<Note>>() {
      @Override
      @NonNull
      public List<Note> call() throws Exception {
        final Cursor _cursor = DBUtil.query(__db, _statement,
false, null);
        try {
          final int _cursorIndexOfId = CursorUtil.getColumnIndex-
OrThrow(_cursor, "id");
          final int _cursorIndexOfTitle = CursorUtil.getColumn-
IndexOrThrow(_cursor, "title");
          final int _cursorIndexOfContent = CursorUtil.getColumn-
IndexOrThrow(_cursor, "content");
          final int _cursorIndexOfUpdatedAt = CursorU-
til.getColumnIndexOrThrow(_cursor, "updatedAt");
```

```
          final List<Note> _result = new ArrayList<Note>(_cur-
sor.getCount());
          while (_cursor.moveToNext()) {
            final Note _item;
            final String _tmpId;
            if (_cursor.isNull(_cursorIndexOfId)) {
              _tmpId = null;
            } else {
              _tmpId = _cursor.getString(_cursorIndexOfId);
            }
            final String _tmpTitle;
            if (_cursor.isNull(_cursorIndexOfTitle)) {
              _tmpTitle = null;
            } else {
              _tmpTitle = _cursor.getString(_cursorIndexOfTitle);
            }
            final String _tmpContent;
            if (_cursor.isNull(_cursorIndexOfContent)) {
              _tmpContent = null;
            } else {
              _tmpContent = _cursor.getString(_cursorIndexOfCon-
tent);
            }
            final long _tmpUpdatedAt;
            _tmpUpdatedAt = _cursor.getLong(_cursorIndexOf-
UpdatedAt);
            _item = new Note(_tmpId,_tmpTitle,_tmpCon-
tent,_tmpUpdatedAt);
            _result.add(_item);
          }
          return _result;
        } finally {
          _cursor.close();
          _statement.release();
        }
      }
    }, $completion);
  }
  @Override
  public Object getNoteById(final String id, final Continuation<?
super Note> $completion) {
    final String _sql = "SELECT * FROM notes WHERE id = ?";
    final RoomSQLiteQuery _statement = RoomSQLiteQuery.ac-
quire(_sql, 1);
    int _argIndex = 1;
    if (id == null) {
      _statement.bindNull(_argIndex);
    } else {
      _statement.bindString(_argIndex, id);
    }
    final CancellationSignal _cancellationSignal = DBUtil.create-
CancellationSignal();
```

```
    return CoroutinesRoom.execute(__db, false, _cancellation-
Signal, new Callable<Note>() {
      @Override
      @Nullable
      public Note call() throws Exception {
        final Cursor _cursor = DBUtil.query(__db, _statement,
false, null);
        try {
          final int _cursorIndexOfId = CursorUtil.getColumnIndex-
OrThrow(_cursor, "id");
          final int _cursorIndexOfTitle = CursorUtil.getColumn-
IndexOrThrow(_cursor, "title");
          final int _cursorIndexOfContent = CursorUtil.getColumn-
IndexOrThrow(_cursor, "content");
          final int _cursorIndexOfUpdatedAt = CursorU-
til.getColumnIndexOrThrow(_cursor, "updatedAt");
          final Note _result;
          if (_cursor.moveToFirst()) {
            final String _tmpId;
            if (_cursor.isNull(_cursorIndexOfId)) {
              _tmpId = null;
            } else {
              _tmpId = _cursor.getString(_cursorIndexOfId);
            }
            final String _tmpTitle;
            if (_cursor.isNull(_cursorIndexOfTitle)) {
              _tmpTitle = null;
            } else {
              _tmpTitle = _cursor.getString(_cursorIndexOfTitle);
            }
            final String _tmpContent;
            if (_cursor.isNull(_cursorIndexOfContent)) {
              _tmpContent = null;
            } else {
              _tmpContent = _cursor.getString(_cursorIndexOfCon-
tent);
            }
            final long _tmpUpdatedAt;
            _tmpUpdatedAt = _cursor.getLong(_cursorIndexOf-
UpdatedAt);
            _result = new Note(_tmpId,_tmpTitle,_tmpCon-
tent,_tmpUpdatedAt);
          } else {
            _result = null;
          }
          return _result;
        } finally {
          _cursor.close();
          _statement.release();
        }
      }
    }, $completion);
  }
```

```
  @NonNull
  public static List<Class<?>> getRequiredConverters() {


    return Collections.emptyList();
  }
}
```

Лістинг Б.8 – AppDatabase_Impi.java

```
package com.naukova.backup.data;

import androidx.annotation.NonNull;
import androidx.room.DatabaseConfiguration;
import androidx.room.InvalidationTracker;
import androidx.room.RoomDatabase;
import androidx.room.RoomOpenHelper;
import androidx.room.migration.AutoMigrationSpec;
import androidx.room.migration.Migration;
import androidx.room.util.DBUtil;
import androidx.room.util.TableInfo;
import androidx.sqlite.db.SupportSQLiteDatabase;
import androidx.sqlite.db.SupportSQLiteOpenHelper;
import java.lang.Class;
import java.lang.Override;
import java.lang.String;
import java.lang.SuppressWarnings;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;
import javax.annotation.processing.Generated;

@Generated("androidx.room.RoomProcessor")
@SuppressWarnings({"unchecked", "deprecation"})
public final class AppDatabase_Impl extends AppDatabase {
  private volatile NoteDao _noteDao;
  @Override
  @NonNull
  protected SupportSQLiteOpenHelper createOpenHelper(@NonNull fi-
nal DatabaseConfiguration config) {
    final SupportSQLiteOpenHelper.Callback _openCallback = new
RoomOpenHelper(config, new RoomOpenHelper.Delegate(2) {
      @Override
      public void createAllTables(@NonNull final Sup-
portSQLiteDatabase db) {
        db.execSQL("CREATE TABLE IF NOT EXISTS `notes` (`id` TEXT
NOT NULL, `title` TEXT NOT NULL, `content` TEXT NOT NULL, `up-
datedAt` INTEGER NOT NULL, PRIMARY KEY(`id`))");
        db.execSQL("CREATE TABLE IF NOT EXISTS room_master_table
(id INTEGER PRIMARY KEY,identity_hash TEXT)");
        db.execSQL("INSERT OR REPLACE INTO room_master_table
```

```
(id,identity_hash) VALUES(42,
'f87ddf656bc7a9bc10d3436e469828f8')");
      }

      @Override
      public void dropAllTables(@NonNull final SupportSQLiteData-
base db) {
        db.execSQL("DROP TABLE IF EXISTS `notes`");
        final List<? extends RoomDatabase.Callback> _callbacks =
mCallbacks;
        if (_callbacks != null) {
          for (RoomDatabase.Callback _callback : _callbacks) {
            _callback.onDestructiveMigration(db);
          }
        }
      }

      @Override
      public void onCreate(@NonNull final SupportSQLiteDatabase
db) {
        final List<? extends RoomDatabase.Callback> _callbacks =
mCallbacks;
        if (_callbacks != null) {
          for (RoomDatabase.Callback _callback : _callbacks) {
            _callback.onCreate(db);
          }
        }
      }

      @Override
      public void onOpen(@NonNull final SupportSQLiteDatabase db)
{
        mDatabase = db;
        internalInitInvalidationTracker(db);
        final List<? extends RoomDatabase.Callback> _callbacks =
mCallbacks;
        if (_callbacks != null) {
          for (RoomDatabase.Callback _callback : _callbacks) {
            _callback.onOpen(db);
          }
        }
      }

      @Override
      public void onPreMigrate(@NonNull final SupportSQLiteData-
base db) {
        DBUtil.dropFtsSyncTriggers(db);
      }

      @Override
      public void onPostMigrate(@NonNull final SupportSQLiteData-
base db) {
      }
```

```
      @Override
      @NonNull
      public RoomOpenHelper.ValidationResult onValidateSchema(

          @NonNull final SupportSQLiteDatabase db) {
        final HashMap<String, TableInfo.Column> _columnsNotes =
new HashMap<String, TableInfo.Column>(4);
        _columnsNotes.put("id", new TableInfo.Column("id", "TEXT",
true, 1, null, TableInfo.CREATED_FROM_ENTITY));
        _columnsNotes.put("title", new TableInfo.Column("title",
"TEXT", true, 0, null, TableInfo.CREATED_FROM_ENTITY));
        _columnsNotes.put("content", new TableInfo.Column("con-
tent", "TEXT", true, 0, null, TableInfo.CREATED_FROM_ENTITY));
        _columnsNotes.put("updatedAt", new TableInfo.Column("up-
datedAt", "INTEGER", true, 0, null, TableInfo.CREATED_FROM_EN-
TITY));
        final HashSet<TableInfo.ForeignKey> _foreignKeysNotes =
new HashSet<TableInfo.ForeignKey>(0);
        final HashSet<TableInfo.Index> _indicesNotes = new
HashSet<TableInfo.Index>(0);
        final TableInfo _infoNotes = new TableInfo("notes", _col-
umnsNotes, _foreignKeysNotes, _indicesNotes);
        final TableInfo _existingNotes = TableInfo.read(db,
"notes");
        if (!_infoNotes.equals(_existingNotes)) {
          return new RoomOpenHelper.ValidationResult(false,
"notes(com.naukova.backup.data.Note).\n"
                  + " Expected:\n" + _infoNotes + "\n"
                  + " Found:\n" + _existingNotes);
        }
        return new RoomOpenHelper.ValidationResult(true, null);
      }
    }, "f87ddf656bc7a9bc10d3436e469828f8",
"53f806390b12c8dec52f94cded276cbb");
    final SupportSQLiteOpenHelper.Configuration _sqliteConfig =
SupportSQLiteOpenHelper.Configuration.builder(config.con-
text).name(config.name).callback(_openCallback).build();
    final SupportSQLiteOpenHelper _helper =
config.sqliteOpenHelperFactory.create(_sqliteConfig);
    return _helper;
  }

  @Override
  @NonNull
  protected InvalidationTracker createInvalidationTracker() {
    final HashMap<String, String> _shadowTablesMap = new
HashMap<String, String>(0);
    final HashMap<String, Set<String>> _viewTables = new
HashMap<String, Set<String>>(0);
    return new InvalidationTracker(this, _shadowTablesMap, _view-
Tables, "notes");
  }
  @Override
  public void clearAllTables() {
    super.assertNotMainThread();
```

```
    final SupportSQLiteDatabase _db = super.get-
OpenHelper().getWritableDatabase();
    try {
      super.beginTransaction();
      _db.execSQL("DELETE FROM `notes`");
      super.setTransactionSuccessful();
    } finally {
      super.endTransaction();
      _db.query("PRAGMA wal_checkpoint(FULL)").close();
      if (!_db.inTransaction()) {
        _db.execSQL("VACUUM");
      }
    }
  }

  @Override
  @NonNull
  protected Map<Class<?>, List<Class<?>>> getRequiredTypeConvert-
ers() {
    final HashMap<Class<?>, List<Class<?>>> _typeConvertersMap =
new HashMap<Class<?>, List<Class<?>>>();
    _typeConvertersMap.put(NoteDao.class,
NoteDao_Impl.getRequiredConverters());
    return _typeConvertersMap;
  }

  @Override
  @NonNull
  public Set<Class<? extends AutoMigrationSpec>> getRequiredAu-
toMigrationSpecs() {
    final HashSet<Class<? extends AutoMigrationSpec>> _autoMigra-
tionSpecsSet = new HashSet<Class<? extends AutoMigrationSpec>>();
    return _autoMigrationSpecsSet;
  }

  @Override
  @NonNull
  public List<Migration> getAutoMigrations(
      @NonNull final Map<Class<? extends AutoMigrationSpec>, Au-
toMigrationSpec> autoMigrationSpecs) {
    final List<Migration> _autoMigrations = new ArrayList<Migra-
tion>();
    return _autoMigrations;
  }

  @Override
  public NoteDao noteDao() {
    if (_noteDao != null) {
      return _noteDao;
    } else {
      synchronized(this) {
        if(_noteDao == null) {
          _noteDao = new NoteDao_Impl(this);

        }
```

```
        return _noteDao;
      }
    }
  }
}
```

Лістинг Б.9 – BackupSelectionActivity.java.java

```
package com.naukova.backup
import android.app.Activity
import android.content.Context
import android.content.Intent
import android.net.Uri
import android.os.Bundle
import android.provider.OpenableColumns
import android.util.Log
import android.widget.Button
import android.widget.TextView
import android.widget.Toast
import androidx.activity.result.ActivityResult
import androidx.activity.result.ActivityResultLauncher
import androidx.activity.result.contract.ActivityResultContracts
import androidx.appcompat.app.AppCompatActivity
import androidx.documentfile.provider.DocumentFile
import androidx.lifecycle.ViewModelProvider
import com.google.android.gms.auth.api.signin.GoogleSignIn
import com.google.android.gms.auth.api.signin.GoogleSignInClient
import com.google.android.gms.auth.api.signin.GoogleSignInOptions
import com.google.android.gms.common.api.Scope
import com.google.api.client.googleapis.extensions.an-
droid.gms.auth.GoogleAccountCredential
import com.google.api.client.http.javanet.NetHttpTransport
import com.google.api.client.json.gson.GsonFactory
import com.google.api.services.drive.Drive
import com.google.api.services.drive.DriveScopes
import com.naukova.backup.viewmodel.NoteViewModel
import java.io.*
import java.util.zip.ZipEntry
import java.util.zip.ZipOutputStream

class BackupSelectionActivity : AppCompatActivity() {

    private val requestCodePickFolder = 1001
    private val requestCodePickFiles = 1002

    private lateinit var statusText: TextView
    private lateinit var viewModel: NoteViewModel

    private val prefs by lazy {
        getSharedPreferences("backup_prefs", Context.MODE_PRIVATE)
    }

    private lateinit var pickFolderLauncher: ActivityResult-
Launcher<Intent>
```

```
    private lateinit var pickFilesLauncher: ActivityResult-
Launcher<Intent>

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_backup_selection)

        // Ініціалізація ViewModel
        viewModel = ViewModelProvider(this)[NoteView-
Model::class.java]

        // Автосинхронізація
        viewModel.fullSync()
        // UI-кнопки
        statusText = findViewById(R.id.status_text)
        val pickFolderBtn = findViewById<But-
ton>(R.id.pick_folder_button)
        val pickFilesBtn = findViewById<But-
ton>(R.id.pick_files_button)
        val downloadFileBtn = findViewById<Button>(R.id.down-
load_file_button)
        val syncButton = findViewById<Button>(R.id.sync_button)

        pickFolderBtn.setOnClickListener { openFolderPicker() }
        pickFilesBtn.setOnClickListener { openFilePicker() }
        downloadFileBtn.setOnClickListener { listAndDownload-
Files() }
        syncButton.setOnClickListener {
            viewModel.fullSync()
            Toast.makeText(this, getString(R.string.sync_started),
Toast.LENGTH_SHORT).show()
        }

        pickFolderLauncher = registerForActivityResult(Activi-
tyResultContracts.StartActivityForResult()) { result: Activi-
tyResult ->
            if (result.resultCode == Activity.RESULT_OK && re-
sult.data != null) {
                val uri = result.data?.data
                uri?.let {
                    contentResolver.takePersistableUriPermis-
sion(it, Intent.FLAG_GRANT_READ_URI_PERMISSION)
                    prefs.edit().putString("selected_folder_uri",
it.toString()).apply()

                    statusText.text =
getString(R.string.folder_selected, it.toString())
                    listFolderContents(it)
                }
            }
        }

        pickFilesLauncher = registerForActivityResult(Activi-
tyResultContracts.StartActivityForResult()) { result: Activi-
tyResult ->
```

```
            if (result.resultCode == Activity.RESULT_OK && re-
sult.data != null) {
                val uris = mutableListOf<Uri>()
                result.data?.clipData?.let { clipData ->
                    val count = clipData.itemCount
                    for (i in 0 until count) {
                        uris.add(clipData.getItemAt(i).uri)
                    }
                } ?: result.data?.data?.let { uris.add(it) }

                val uriStrings = uris.map { it.toString()
}.toSet()
                prefs.edit().putStringSet("selected_file_uris",
uriStrings).apply()
                statusText.text = getString(R.string.files_se-
lected, uris.size)

                val zipFile = createZipFromUris(this, uris)
                if (zipFile != null) {
                    Log.d("Backup", "ZIP створено: ${zipFile.abso-
lutePath}")
                    val driveService = getDriveService()
                    uploadFileToDrive(driveService, zipFile, "ap-
plication/zip", zipFile.name)
                }
            }
        }
    }
    private fun openFolderPicker() {
        val intent = Intent(Intent.ACTION_OPEN_DOCUMENT_TREE)
        pickFolderLauncher.launch(intent)
    }

    private fun openFilePicker() {
        val intent = Intent(Intent.ACTION_OPEN_DOCUMENT)
        intent.addCategory(Intent.CATEGORY_OPENABLE)
        intent.type = "*/*"
        intent.putExtra(Intent.EXTRA_ALLOW_MULTIPLE, true)
        pickFilesLauncher.launch(intent)
    }

    private fun listFilesInDrive(driveService: Drive) {
        val request = driveService.files().list()
        request.pageSize = 10
        request.fields = "nextPageToken, files(id, name)"
        val result = request.execute()
        val files = result.files
        if (files != null && files.isNotEmpty()) {
            for (file in files) {
                Log.d("Drive", "Файл: ${file.name}, ID:
${file.id}")
            }
        } else {
            Log.d("Drive", "Файли не знайдені.")
        }
```

```
    }

    private fun downloadFileFromDrive(driveService: Drive, fileId:
String, destinationFile: File) {
        val outputStream = FileOutputStream(destinationFile)
        driveService.files().get(fileId)
                .executeMediaAndDownloadTo(outputStream)
        Log.d("Drive", "Файл завантажено:
${destinationFile.absolutePath}")
    }

    private fun listAndDownloadFiles() {
        val driveService = getDriveService()
        listFilesInDrive(driveService)

        val fileId = "id_файлу_для_завантаження"
        val destinationFile = File(filesDir, "down-
loaded_file.zip")
        downloadFileFromDrive(driveService, fileId, destination-
File)
    }

    private fun listFolderContents(uri: Uri) {
        val folder = DocumentFile.fromTreeUri(this, uri)
        folder?.listFiles()?.forEach {
            Log.d("Backup", "Файл у вибраній папці: ${it.name}")
        }
    }

    private fun createZipFromUris(context: Context, uris:
List<Uri>, zipFileName: String = "backup.zip"): File? {
        val zipFile = File(context.cacheDir, zipFileName)

        try {
            ZipOutputStream(BufferedOutputStream(FileOut-
putStream(zipFile))).use { zos ->
                for (uri in uris) {
                    val fileName = getFileNameFromUri(context,
uri) ?: continue
                    val inputStream = context.contentRe-
solver.openInputStream(uri) ?: continue

                    zos.putNextEntry(ZipEntry(fileName))
                    inputStream.copyTo(zos, bufferSize = 1024)
                    zos.closeEntry()
                    inputStream.close()
                }
            }
            return zipFile
        } catch (e: IOException) {
            e.printStackTrace()
            return null
        }
    }
```

```
    private fun getFileNameFromUri(context: Context, uri: Uri):
String? {
        val cursor = context.contentResolver.query(uri, null,
null, null, null)
        return cursor?.use {
            val nameIndex = cursor.getColumnIndex(OpenableCol-
umns.DISPLAY_NAME)
            cursor.moveToFirst()
            cursor.getString(nameIndex)
        }
    }

    private fun getDriveService(): Drive {
        val signInOptions = GoogleSignInOptions.Builder(Google-
SignInOptions.DEFAULT_SIGN_IN)
                .requestScopes(Scope(DriveScopes.DRIVE_FILE))
                .build()

        val googleSignInClient: GoogleSignInClient = Google-
SignIn.getClient(this, signInOptions)

        val account = GoogleSignIn.getLastSignedInAccount(this)
        val credential = GoogleAccountCredential.usingOAuth2(
                this, listOf(DriveScopes.DRIVE_FILE)
        )
        credential.selectedAccount = account?.account

        return Drive.Builder(
                NetHttpTransport(),
                GsonFactory.getDefaultInstance(),
                credential
        ).setApplicationName("NaukovaBackup").build()
    }

    fun uploadFileToDrive(driveService: Drive, filePath: File,
mimeType: String, fileName: String) {
        val fileMetadata = com.google.api.ser-
vices.drive.model.File()
        fileMetadata.name = fileName
        val fileContent = com.google.api.client.http.FileCon-
tent(mimeType, filePath)

        try {
            val request = driveService.files().cre-
ate(fileMetadata, fileContent)
                    .setFields("id")
                    .execute()
            Log.d("Drive", "Файл завантажено, ID: ${request.id}")
        } catch (e: IOException) {
            Log.e("Drive", "Помилка завантаження файлу: ${e.mes-
sage}")
        }
    }
}
```

Лістинг Б.10 – BuildConfig

```
/**
 * Automatically generated file. DO NOT MODIFY
 */
package com.naukova.backup;

public final class BuildConfig {
  public static final boolean DEBUG = Boolean.parseBool-
ean("true");
  public static final String APPLICATION_ID = "com.nau-
kova.backup";
  public static final String BUILD_TYPE = "debug";
  public static final int VERSION_CODE = 1;
  public static final String VERSION_NAME = "1.0";
}
```

Лістинг Б.11 – AppDatabase.kt

```
package com.naukova.backup.data

import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase
import androidx.room.TypeConverters

@Database(entities = [Note::class], version = 2, exportSchema =
false)
@TypeConverters(Converters::class)
abstract class AppDatabase : RoomDatabase() {

    abstract fun noteDao(): NoteDao
    companion object {
        @Volatile
        private var INSTANCE: AppDatabase? = null

        @JvmStatic
        fun getDatabase(context: Context): AppDatabase {
            return INSTANCE ?: synchronized(this) {
                val instance = Room.databaseBuilder(
                        context.applicationContext,
                        AppDatabase::class.java,
                        "note_database"
                )
                        .fallbackToDestructiveMigration()
                        .build()
                INSTANCE = instance
                instance
            }
        }

        @JvmStatic
        fun closeDatabase() {
            INSTANCE?.close()
```

```
            INSTANCE = null
        }
    }
}
```

Лістинг Б.12 – Converters.kt

```
package com.naukova.backup.data
import androidx.room.TypeConverter
import com.google.firebase.Timestamp
class Converters {
    @TypeConverter
    fun fromTimestamp(value: Timestamp?): Long? {
        return value?.seconds
    }

    @TypeConverter
    fun toTimestamp(value: Long?): Timestamp? {

        return value?.let { Timestamp(it, 0) }
    }
}
```

Лістинг Б.13 – Note.kt

```
package com.naukova.backup.data

import androidx.room.Entity
import androidx.room.PrimaryKey
import java.util.UUID // 🔗 Це потрібно для генерації ID
import com.google.firebase.firestore.IgnoreExtraProperties

@IgnoreExtraProperties
@Entity(tableName = "notes")
data class Note(
        @PrimaryKey val id: String = UUID.randomUUID().toString(),
        val title: String = "",
        val content: String = "",
        val updatedAt: Long = System.currentTimeMillis()
)
```

Лістинг Б.14 – NoteDao.kt

```
package com.naukova.backup.data
import androidx.lifecycle.LiveData
import androidx.room.*
@Dao
interface NoteDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(note: Note)
```

Продовження Лістинг Б.15 –NoteDao.kt

```
@Delete
    suspend fun delete(note: Note)

    @Query("SELECT * FROM notes ORDER BY updatedAt DESC")
    fun getAllNotes(): LiveData<List<Note>>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertAll(notes: List<Note>)

    @Query("SELECT * FROM notes")
    suspend fun getAllNow(): List<Note>

    @Query("SELECT * FROM notes WHERE id = :id")
    suspend fun getNoteById(id: String): Note?

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertOrUpdate(note: Note)

    // 🔘 Лишаємо тільки цей метод для Java:
    @Query("DELETE FROM notes WHERE id = :noteId")
    fun deleteByIdSync(noteId: String)
}
```

Лістинг Б.16 – NoteRepository.kt

```
package com.naukova.backup.data

import androidx.lifecycle.LiveData
import com.google.firebase.firestore.FirebaseFirestore
import kotlinx.coroutines.tasks.await
import android.util.Log
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch
class NoteRepository(private var dao: NoteDao) {

    private val firestore = FirebaseFirestore.getInstance()

    //  Завжди отримуємо LiveData з DAO напряму — не кешуємо
allNotes
    fun getAllNotes(): LiveData<List<Note>> = dao.getAllNotes()

    //  Перестворюємо DAO після відновлення бази
    fun setDao(newDao: NoteDao) {
        dao = newDao
    }

    suspend fun insertNote(note: Note) {
        dao.insert(note)
        firestore.collection("notes").docu-
ment(note.id).set(note).await()
    }
```

```
    suspend fun deleteNote(note: Note) {
        dao.delete(note)
        firestore.collection("notes").document(note.id).de-
lete().await()
    }

    suspend fun insertNotesFromFirestore() {
        val snapshot = firestore.collection("notes").get().await()
        val notes = snapshot.documents.mapNotNull { it.toOb-
ject(Note::class.java) }
        dao.insertAll(notes)
    }

    suspend fun uploadNotesToFirestore() {
        val localNotes = dao.getAllNow()
        for (note in localNotes) {
            firestore.collection("notes").docu-
ment(note.id).set(note).await()
        }
    }

    suspend fun deleteRemoteNotesNotInLocal() {
        val localNotes = dao.getAllNow()
        val localIds = localNotes.map { it.id }.toSet()
        val remoteNotes = firestore.collec-
tion("notes").get().await()
        for (doc in remoteNotes.documents) {
            val remoteId = doc.id
            if (remoteId !in localIds) {

firestore.collection("notes").document(remoteId).delete().await()
            }
        }
    }

    fun syncFromFirestoreToRoom() {
        firestore.collection("notes")
                .addSnapshotListener { snapshots, e ->
                    if (e != null || snapshots == null) {
                        Log.e("Sync", "Listen failed", e)
                        return@addSnapshotListener
                    }

                    for (doc in snapshots.documents) {
                        val firestoreNote = doc.toOb-
ject(Note::class.java)
                        if (firestoreNote != null) {
                            CoroutineScope(Dispatchers.IO).launch
{
                                val localNote =
dao.getNoteById(firestoreNote.id)
                                if (localNote == null || fire-
storeNote.updatedAt > localNote.updatedAt) {
```

```
                                    dao.insertOrUpdate(firestore-
Note)
                                }
                            }
                        }
                    }
                }
    }

    fun syncRoomToFirestore() {
        getAllNotes().observeForever { notes ->
            CoroutineScope(Dispatchers.IO).launch {

 for (note in notes) {
                    firestore.collection("notes").docu-
ment(note.id).set(note).await()
                }
            }
        }
    }
}
```

Додаток В

Задачі, у яких застосовують об’єднання SQL та NoSQL

Задача	**SQL**	**NoSQL**	**Пояснення**	**Приклад у мобільному додатку**
Обробка транзакцій	Так	Ні	SQL гарантує ACID-властивості	Локальні зміни даних користувача (нотатки, налаштування) перед синхронізацією
Аналітика в реальному часі	Ні	Так	NoSQL забезпечує швидкий запис і читання	Відстеження поведінки користувача на бекенді для персоналізації; швидке відображення стрічки новин
Управління контентом	Ні	Так	Гнучкість схеми документних БД	Зберігання користувацького контенту (фото, пости) з різною структурою на бекенді, кешування на клієнті

Веб-магазини	Так	Так	Товари та транзакції SQL, сесії та перегляди NoSQL	Локально – кошик, історія замовлень (SQL); Хмарно – каталог товарів, рекомендації, сесії користувача (NoSQL)
Обробка логів та телеметрії	Ні	Так	Висока частота записів	Збір даних про використання мобільного додатку для аналітики на бекенді
Платформи з персоналізацією	Ні	Так	Обробка великої кількості сесій	Зберігання профілів користувачів, їх вподобань у хмарі (NoSQL) для синхронізації між пристроями
Повноцінна офлайн-робота	Так (локальна SQL база)	Обмежено (деякі NoSQL мають офлайн-кеш)	SQL надає повний контроль над локальними даними без мережі	Створення/редагування нотаток, завдань без доступу до Інтернету (локальний SQL)
Синхронізація між пристроями	Ні (потребує кастомної реалізації)	Так (багато хмарних NoSQL мають нативну підтримку)	NoSQL-сервіси часто розроблені з урахуванням багатопристроєвого доступу	Автоматичне оновлення нотаток на всіх пристроях користувача через хмарний NoSQL-сервіс

Додаток Д

Порівняння стратегій об’єднання SQL та NoSQL

Стратегія	Переваги	Недоліки	Сценарії використання	Релевантність для мобільних
NoSQL як кеш для SQL	Прискорення доступу до часто запитуваних даних, зменшення навантаження на SQL	Вимоги до узгодженості кешу, додаткова складність при оновленні даних	Веб-кешування, профілі користувачів, кешування запитів	Висока (як хмарний кеш або локальний SQL кеш)
CQRS (читання NoSQL, запис SQL)	Можливість незалежного масштабування читання та запису; використання оптимальних моделей даних	Складна синхронізація між частинами, затримки при оновленнях	Мікросервіси, high-load сервіси, реальні події	Середня (може бути надлишково складним)
Зберігання різних типів у відповідних сховищах	Оптимальне використання СУБД відповідно до типу даних	Складність запитів між сховищами; потреба в логічній інтеграції	E-commerce, IoT, CMS, гібридні додатки	Дуже висока (основний підхід)
Middleware/API-шар	Єдиний інтерфейс доступу; централізований контроль запитів	Може бути «вузьким місцем» у продуктивності; складність налаштування	REST API, мобільні додатки, централізовані сервіси	Висока (реалізується в самому додатку)

Polyglot Persistence	Гнучкість, розділення функціональних частин за технологіями	Висока вартість інтеграції, складність у супроводі	Модульні системи, аналітика vs транзакції	Середня
Data Synchronization (ETL, CDC)	Актуальність даних; підтримка асинхронної або потокової синхронізації	Затримки або конфлікти при оновленні; технічна складність налаштування	Аналітика, фінанси, Big Data, реплікація	Дуже висока (ключовий аспект)
Federated Query Engines	SQL-запити до гетерогенних джерел; інтеграція в BI-системи	Зниження продуктивності; потреба у складній оптимізації	Data Lake, звітність, візуалізація, аналітика	Низька (для клієнта)