Міністерство освіти і науки України

Луцький національний технічний університет
(повне найменування закладу вищої освіти)

Факультет комп'ютерних та інформаційних технологій
(повне найменування факультету)

Кафедра комп'ютерної інженерії та безпеки
(повне найменування кафедри)

КВАЛІФІКАЦІЙНА РОБОТА ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «МАГІСТР»

ЕФЕКТИВНІСТЬ РЕЛЯЦІЙНИХ ТА NOSQL БД У СЕНСОРНИХ СИСТЕМАХ

EFFICIENCY OF RELATIONAL AND NOSQL DATABASES IN SENSOR SYSTEMS

спеціальність 123 Комп'ютерна інженерія
(шифр і назва спеціальності)

освітня програма Комп'ютерна інженерія
(назва освітньої програми)

Виконав: здобувач вищої освіти
групи КІм-21
Лук'янчук Юлія Володимирівна

(підпис)

Керівник:
к.т.н., доцент
Лавренчук Світлана Василівна

(підпис)

Кваліфікаційну роботу
допущено до захисту
«_____» грудня 2025 р.
Гарант освітньої програми:
к.т.н., доцент
Гринюк Сергій Васильович

(підпис)

Луцьк – 2025 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *комп'ютерної інженерії та безпеки*
Ступінь вищої освіти: *магістр*
Галузь знань: *12 Інформаційні технології*
Спеціальність: *123 Комп'ютерна інженерія*
Освітня програма: *«Комп'ютерна інженерія»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ доц. Т.ТЕРЛЕЦЬКИЙ
«_____» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Лук'янчук Юлії Володимирівні
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи *Ефективність реляційних та NoSQL БД у сенсорних системах*

Керівник роботи *к.т.н., доцент Лавренчук Світлана Василівна*

затверджені наказом закладу вищої освіти від «17» червня 2025 року № 290/01-02
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи 09.12.2025р.

3. Вихідні дані до роботи *Джерелом розробки є науково-технічна література та публікації в періодичних виданнях з даного питання, опубліковані зарубіжні та вітчизняні роботи в даній області, різні інтернет-ресурси технічного спрямування*

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити):
Вступ
Аналіз проблеми за темою роботи та постановка завдань дослідження
Методи обробки сенсорних даних у сучасних системах
Теоретичні основи вибору технологій для обробки сенсорних даних
Експериментальне дослідження та аналіз результатів
Висновки

5. Перелік графічного (ілюстративного) матеріалу:
Графіки динаміки популярності різноманітних типів баз даних
Схеми, що показують основні напрямки класифікацій сенсорів в IoT
Схеми, що показують стратегії оптимізації різноматних типів баз даних
Схема загальної структури проєкту та схема серверної частини
Структурна схема апартної частини системи моніторингу
Модель розробленого в межах кваліфікаційної роботи девайсу
Логіка розробленого програмного забезпечення

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз проблеми за темою роботи та постановка завдань дослідження. Методи обробки сенсорних даних у сучасних системах	*Лавренчук С.В., доцент*		
Теоретичні основи вибору технологій для обробки сенсорних даних	*Лавренчук С.В., доцент*		
Експериментальне дослідження та аналіз результатів	*Лавренчук С.В., доцент*		
Нормоконтроль	*Багнюк Н.В., доцент*		
Гарант ОП	*Гринюк С.В., доцент*		
Показник запозичень тексту	***___%***		
Академічна доброчесність	*Міскевич О.І., ст.викладач*		

7. Дата видачі завдання 18.06.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	*Огляд літератури із досліджуваної проблеми*	до 01.08.2025 р.	
2.	*Аналіз проблеми за темою роботи та постановка завдань дослідження. Методи обробки сенсорних даних у сучасних системах*	до 20.08.2025 р.	
3.	*Теоретичні основи вибору технологій для обробки сенсорних даних*	до 25.09.2025 р.	
4.	*Експериментальне дослідження та аналіз результатів*	до 20.10.2025 р.	
5.	*Висновки та пропозиції*	до 25.10.2025 р	
6.	*Формування списку використаних джерел*	до 27.10.2025 р.	
7.	*Формування додатків*	до 30.10.2025 р.	
8.	*Оформлення ілюстративного матеріалу*	до 05.11.2025 р.	
9.	*Представлення остаточного варіанту кваліфікаційної роботи керівникові*	до 11.11.2025 р.	
10.	*Нормоконтроль*	до 29.11.2025 р.	
11.	*Інструментальна перевірка на академічний плагіат*	до 02.12.2025 р.	
12.	*Здача кваліфікаційної роботи та всіх супровідних документів на кафедру*	до 09.12.2025 р.	

Здобувач вищої освіти _____ Лук’янчук Ю.В.
(підпис) (прізвище, ініціали)

Керівник кваліфікаційної роботи _____ Лавренчук С.В.
(підпис) (прізвище, ініціали)

АНОТАЦІЯ

Лук’ячнук Ю. В. Ефективність реляційних та NoSQL БД у сенсорних системах. Рукопис.

Кваліфікаційна робота магістра ОП «Комп’ютерна інженерія» спеціальності 123 Комп’ютерна інженерія. Луцький національний технічний університет. Луцьк, 2025.

Кваліфікаційна робота складається з вступу, трьох розділів, висновків, списку використаних джерел, додатків.

Перший розділ присвячено огляду предметної олбасті, тут розглядаються основні поняття про бази даних, їх види та сфери їх використання, наведено багато практичних прикладів. Також в цьому розділі здійснено огляд основних систем керування реляційними та NoSQL базами даних, оглянуто бази даних, які є популярними серед розробників.

В другому розділі здійснено вибір та обгрунтування засобів розробки. Обрано засоби: Arduino, BME-680, C#, Blazor, PostgreSQL, MongoDB.

Третій розділ присвячено опису розробленого девайсу, створеного в межах кваліфікаціної роботи, його програмного забезпечення, програмного забезпечення для серверної частини, а також схеми баз даних та методи їх тестування.

Ключові слова: реляційні бази даних, NoSQL бази даних, Інтернет речей, продуктивність баз даних, MongoDB, PostgreSQL, обробка даних сенсорів, порівняльний аналіз, real-time обробка даних.

ANNOTATION

Lukianchuk Y. Efficiency of relational and NoSQL databases in sensor systems. Manuscript.

Qualifying work of a Master's of EP «Computer Engineering» specialty 123 Computer Engineering. Lutsk National Technical University. Lutsk, 2025.

Qualification work consists of an introduction, three sections, conclusions, a references, two appendices.

The first section is dedicated to an overview of the subject area, where the basic concepts of databases, their types and areas of application are examined, with numerous practical examples provided. This chapter also reviews the main relational and NoSQL database management systems, and examines databases that are popular among developers.

The second section, addresses the selection and justification of development tools. The chosen tools include: Arduino, BME-680, C#, Blazor, PostgreSQL, and MongoDB.

The third section describes the device developed as part of this thesis work, including its firmware, server-side software implementation, database schemas, and testing methodologies.

Keywords: relational databases, NoSQL databases, Internet of Things, database performance, MongoDB, PostgreSQL, sensor data processing, comparative analysis, air quality index, real-time data processing

ЗМІСТ

ВСТУП

Сучасний етап розвитку інформаційних технологій характеризується стрімким зростанням кількості пристроїв, здатних генерувати, передавати та обробляти дані в режимі реального часу. Особливе місце серед таких пристроїв займають системи Internet of Things (IoT), які формують величезні масиви даних, що потребують ефективного збереження, швидкої обробки та масштабованості. У зв’язку з цим питання вибору оптимальної системи керування базами даних (СКБД) стає ключовим для забезпечення стабільного функціонування таких систем. Традиційні реляційні бази даних, які десятиліттями були основним інструментом у сфері збереження інформації, конкурують із сучасними нереляційними (NoSQL) базами, що пропонують нові підходи до роботи з великими та динамічними потоками даних.

Особливість сенсорних систем полягає в тому, що дані надходять з великою частотою та в різному форматі, що обумовлює необхідність швидкого запису та гнучкої структури зберігання. У таких умовах реляційні бази даних можуть здаватись обмеженими, через жорстку структуру таблиць, складність масштабування та можливі затримки при обробці запитів. З іншого боку, NoSQL СКБД орієнтовані на горизонтальне масштабування та можуть працювати з неструктурованими або напівструктурованими даними, що потенційно робить їх ефективнішими для таких завдань. Проте їх використання також пов’язане з певними викликами, такими як відсутність транзакційності на рівні ACID або складність проектування оптимальної моделі даних.

Таким чином, актуальність теми даного дослідження полягає у необхідності порівняльного аналізу ефективності реляційних та NoSQL баз даних в умовах сенсорних систем з метою визначення оптимального підходу до збереження та обробки даних у реальному часі. Такий аналіз дозволить сформувати практичні рекомендації для інженерів, розробників та дослідників щодо вибору СКБД залежно від особливостей сенсорної системи, частоти надходження даних та вимог до їх обробки.

Об’єктом дослідження є процес збереження та обробки сенсорних даних у базах даних.

Предметом дослідження виступає ефективність функціонування реляційних та NoSQL СКБД у контексті сенсорних систем.

Метою роботи є експериментальне дослідження та порівняльний аналіз продуктивності реляційних та NoSQL баз даних при обробці даних, отриманих від розробленого сенсорного пристрою, що працює в режимі реального часу.

Для досягнення поставленої мети необхідно виконати такі завдання:

– проаналізувати теоретичні основи функціонування реляційних та NoSQL баз даних, визначити їх переваги та недоліки в контексті сенсорних систем;

– розробити апаратний прототип сенсорного пристрою (девайсу), оснащеного датчиками для збору показників (наприклад, температури, вологості або тиску), який буде передавати дані на сервер;

– створити програмне забезпечення для девайсу, яке забезпечуватиме зчитування даних із сенсорів, їх попередню обробку та передачу на сервер за допомогою бездротового каналу зв’язку;

– розробити серверний API для прийому даних, їх збереження у вибраних СКБД (реляційна та NoSQL) та забезпечення доступу до них;

– реалізувати простий користувацький інтерфейс (UI), який дозволить візуалізувати отримані дані та виконувати базові аналітичні запити;

– провести серію експериментів для оцінки швидкості запису, читання та масштабованості систем на базі реляційної та NoSQL СКБД;

– здійснити аналіз отриманих результатів та сформулювати рекомендації щодо вибору типу бази даних для сенсорних систем із різним навантаженням.

Наукова новизна роботи полягає у створенні комплексної практичної моделі сенсорної системи із паралельним використанням двох типів баз даних та порівнянні їх ефективності на основі реальних показників роботи девайсу в умовах квазіреального часу. Практична значущість роботи полягає у можливості використання отриманих результатів і рекомендацій при розробці систем

моніторингу, розумних будинків, промислових IoT-рішень та систем автоматизації.

Методологія дослідження включатиме як теоретичний аналіз літератури та сучасних підходів у сфері обробки сенсорних даних, так і експериментальну частину з застосуванням реального прототипу системи. Експериментальна оцінка включатиме вимірювання часу вставки та читання даних, затримки при масштабуванні, продуктивності при різних обсягах навантаження та стійкості до пікових потоків інформації. Порівняння буде проведено на основі метрик продуктивності, надійності та гнучкості адаптації до змінних структур даних.

Очікуваним результатом магістерської роботи є формування узагальнених висновків щодо доцільності використання реляційних та NoSQL баз даних у сенсорних системах залежно від характеристик створюваного застосунку, частоти оновлення даних, масштабованості системи та потреб у транзакційній цілісності. Також буде запропоновано модель архітектури сенсорної системи, яка може слугувати базою для подальшого розширення та впровадження в реальні інженерні рішення.

Таким чином, проведене дослідження спрямоване на вирішення фундаментального прикладного завдання – забезпечення ефективного, продуктивного та надійного функціонування сенсорних систем шляхом обґрунтованого вибору типу бази даних. Отримані результати можуть мати практичне застосування у сферах промислового моніторингу, агротехнологій, екологічного спостереження, «розумного» житла та міської інфраструктури, де важливим є безперервний збір, обробка та аналіз великих потоків даних у реальному часі.

Апробація результатів. Результати роботи представлені на IX Міжнародній студенстській науковій конференції «Модернізація та сучасні українські і світові наукові дослідження», яка відбулася 14 листопада 2025 р. [1] (додаток А).

Публікації. Результати досліджень, виконаних під час роботи над кваліфікаційною роботою магістра опубліковано в науковій статті [2] (додаток Б).

РОЗДІЛ 1
МЕТОДИ ОБРОБКИ СЕНСОРНИХ ДАНИХ У СУЧАСНИХ СИСТЕМАХ

1.1 Особливості обробки сенсорних даних у реальному часі

Обробка сенсорних даних у реальному часі відіграє важливу роль у сучасних інформаційних системах, які базуються на технологіях Інтернету речей.

Сенсорна система в контексті IoT – це сукупність фізичних пристроїв, таких як датчики, які здатні виявляти зміни в навколишньому середовищі та перетворювати ці зміни на цифрові сигнали для подальшої обробки та аналізу. Ці системи забезпечують збір даних з різноманітних джерел, таких як температура, вологість, тиск, рух, освітленість тощо, і передають їх через мережу для аналізу та прийняття рішень [3]. Різноманітні типи сенсорів представлено на рисунку 1.1.

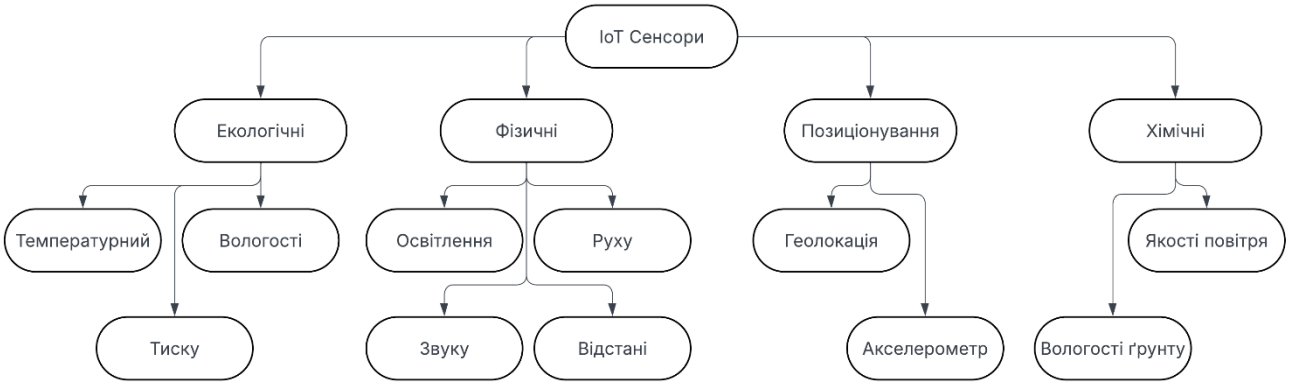

Рисунок 1.1 – Класифікація сенсорів в IoT

Сенсорні дані – це інформація, отримана від сенсорів або органів чуття, яка відображає фізичні або хімічні властивості навколишнього середовища або внутрішнього стану організму. Ці дані можуть включати показники температури, вологості, тиску, освітленості, концентрації газів, прискорення, звуку та інші параметри, що вимірюються за допомогою відповідних сенсорів. Сенсорні дані

є основою для моніторингу, контролю та аналізу різноманітних процесів у реальному часі [4].

Сенсорні системи формують великі потоки даних, які характеризуються високою частотою надходження, різноманітністю форматів та значною розподіленістю джерел. Відповідно до сучасних наукових досліджень, ефективність обробки таких даних визначається не лише продуктивністю апаратного забезпечення, а й архітектурними особливостями систем керування базами даних, що використовуються для їхнього збереження та обробки [5].

Зокрема, реляційні СКБД, які традиційно застосовувалися для обробки структурованих даних, часто демонструють обмежену продуктивність при роботі з потоковими сенсорними даними через жорстку структуру таблиць і складність горизонтального масштабування [6]. Натомість NoSQL бази даних пропонують гнучкі схеми зберігання, що дозволяють адаптувати структуру даних до змін у потоках сенсорної інформації та забезпечують ефективне масштабування в режимі реального часу [7].

Особливістю сенсорних даних є їхня неоднорідність: вони можуть містити числові вимірювання, текстові сигнали, координати або навіть мультимедійні потоки. Така різноманітність накладає специфічні вимоги на етапи збору, передавання, збереження та обробки інформації. Для забезпечення цілісності даних і своєчасності їх надходження застосовуються методи фільтрації, нормалізації та агрегації даних ще на рівні пристрою-джерела, перш ніж вони потрапляють до серверної системи [8]. Це особливо важливо в умовах, коли затримка або пропускна здатність каналу передачі можуть суттєво впливати на ефективність системи, наприклад у промислових IoT-рішеннях, розумних будинках або системах моніторингу навколишнього середовища [9].

Архітектура сенсорних систем у реальному часі зазвичай передбачає поділ на три основні рівні: пристрої збору даних, проміжне обробне програмне забезпечення та серверне середовище з базою даних і аналітичними модулями. На рівні пристроїв реалізуються первинні алгоритми обробки та відправлення даних через бездротові протоколи зв’язку, такі як MQTT або HTTP/REST, що

забезпечують мінімальні затримки та надійність передачі [10]. Серверний рівень виконує більш складну обробку, включно з трансформацією даних у стандартизовані формати, агрегуванням та зберіганням у вибраній СКБД. У цьому контексті важливу роль відіграє вибір типу бази даних: реляційна БД підходить для обробки строго структурованих даних із транзакційною цілісністю, тоді як NoSQL системи, такі як MongoDB або Cassandra, ефективніші для роботи з великими потоками даних, що надходять від численних сенсорів у реальному часі [11].

Одним із ключових методів роботи з даними у реальному часі є потокова обробка, яка дозволяє аналізувати інформацію безпосередньо під час її надходження. На відміну від пакетної обробки, що передбачає накопичення даних у певному обсязі перед їхньою обробкою, потокова обробка зменшує затримки та дозволяє системі реагувати миттєво на зміни в сенсорному середовищі [12]. Використання потокових фреймворків, таких як Apache Kafka або Apache Flink, у поєднанні з NoSQL БД забезпечує горизонтальне масштабування та високу доступність системи, що є критично важливим для IoT-додатків [13].

Ще одним важливим чинником, який впливає на ефективність обробки сенсорних даних, є механізм забезпечення достовірності та цілісності інформації. Зокрема, в умовах високочастотного надходження даних виникає необхідність контролю пропускної здатності, виявлення дублювання повідомлень та корекції помилок, що може реалізовуватися як на рівні протоколу передачі даних, так і на рівні бази даних. У реляційних системах транзакційність ACID дозволяє гарантувати цілісність даних, однак може сповільнювати обробку потоків у режимі реального часу. NoSQL СКБД компенсують це через розподілені моделі зберігання і eventual consistency, що забезпечує масштабованість і швидкість за рахунок поступового досягнення консистентності.

Системи обробки сенсорних даних у реальному часі повинні задовольняти ряд специфічних вимог, які визначаються природою IoT-додатків та

характеристиками потоків даних. До основних вимог належать низька латентність, висока пропускна здатність, масштабованість, надійність та енергоефективність [14].

Низька латентність є критичним параметром для додатків, що потребують миттєвого реагування на події. Наприклад, у системах промислової автоматизації затримка у обробці даних від сенсорів може призвести до аварійних ситуацій або втрати продукції. Дослідження показують, що для більшості критичних IoT-додатків допустима латентність не повинна перевищувати 100 мілісекунд від моменту генерації події до отримання відповіді системи. Для досягнення таких показників застосовуються техніки edge computing, коли частина обробки виконується безпосередньо на граничних пристроях, що зменшує навантаження на мережу та центральні сервери [15].

Висока пропускна здатність необхідна для обробки великих обсягів даних, що генеруються множинними сенсорами. Сучасні IoT-системи можуть включати тисячі або навіть мільйони датчиків, кожен з яких передає дані з частотою від кількох разів на секунду до кількох разів на мілісекунду. За оцінками аналітиків, до 2030 року кількість підключених IoT-пристроїв перевищить 50 мільярдів, що створить безпрецедентне навантаження на системи збору та обробки даних. Для забезпечення необхідної пропускної здатності використовуються розподілені архітектури з паралельною обробкою даних та механізми буферизації, які згладжують пікові навантаження [16].

Масштабованість системи визначає її здатність адаптуватися до зростаючих обсягів даних та кількості підключених пристроїв без суттєвого зниження продуктивності. Горизонтальне масштабування, яке передбачає додавання нових вузлів обробки до існуючої інфраструктури, є переважним підходом для IoT-систем, оскільки дозволяє лінійно збільшувати обчислювальну потужність. NoSQL бази даних, зокрема документо-орієнтовані та колонкові сховища, демонструють кращі показники масштабованості порівняно з традиційними реляційними СКБД завдяки відсутності жорстких схем та підтримці розподіленого зберігання даних [17].

Надійність системи обробки сенсорних даних передбачає здатність функціонувати навіть при відмові окремих компонентів. Механізми реплікації даних, резервування критичних вузлів та автоматичного відновлення після збоїв є невід'ємними складовими сучасних IoT-платформ. Дослідження показують, що системи з коефіцієнтом доступності 99,9 % та вище можуть бути досягнуті через використання розподілених архітектур з автоматичним балансуванням навантаження та захисними механізмами [18].

Ураховуючи величезні обсяги даних, що генеруються IoT-пристроями, важливу роль відіграють методи компресії та агрегації інформації. Компресія дозволяє зменшити обсяг даних, що передаються по мережі та зберігаються в базі даних, без суттєвої втрати інформативності [19]. Існують як загальні алгоритми стиснення, такі як GZIP або LZ4, так і спеціалізовані методи, розроблені з урахуванням специфіки сенсорних даних.

Особливістю сенсорних даних є їхня часова кореляція: послідовні вимірювання одного датчика часто мають подібні значення, що дозволяє ефективно застосовувати дельта-кодування, при якому зберігається не абсолютне значення вимірювання, а різниця з попереднім значенням. Такий підхід може зменшити обсяг даних у 5-10 разів залежно від характеру вимірюваних параметрів. Альтернативним методом є використання словників або таблиць перетворень, коли повторювані значення замінюються короткими кодами, що особливо ефективно для категоріальних даних або даних з обмеженим діапазоном значень [20].

Агрегація даних передбачає обчислення узагальнюючих статистик на певному часовому вікні замість збереження кожного окремого вимірювання. Наприклад, замість збереження тисячі вимірювань температури за годину можна зберігати середнє, мінімальне та максимальне значення, що зменшує обсяг даних у сотні разів при збереженні основної інформативності. Такий підхід широко використовується в системах моніторингу, де детальна історія не є критичною, але важливі довгострокові тренди та аномалії [21].

Складніші методи агрегації включають застосування вейвлет-перетворення або перетворення Фур'є для виявлення періодичних компонент у сигналах від датчиків. Це дозволяє зберігати дані в частотній області, що може бути більш компактним представленням для певних типів сигналів, особливо в додатках вібромоніторингу або аналізу акустичних даних [22].

Вибір протоколу передачі даних суттєво впливає на ефективність IoT-системи. Традиційні протоколи, такі як HTTP/HTTPS, хоча й забезпечують надійність та широку сумісність, характеризуються значним overhead через необхідність встановлення з'єднання та передачі великих заголовків [23]. Для IoT-додатків розроблено спеціалізовані протоколи, оптимізовані для роботи з обмеженими ресурсами та нестабільними мережами.

Протокол MQTT є одним з найпопулярніших у сфері IoT завдяки своїй легковаговій природі та моделі публікації-підписки. MQTT використовує брокер для маршрутизації повідомлень між пристроями, що дозволяє реалізувати асинхронну комунікацію та зменшити навантаження на окремі вузли. Протокол підтримує різні рівні QoS (Quality of Service), що дозволяє балансувати між надійністю доставки та ефективністю використання мережі [24]. Дослідження показують, що MQTT може зменшити обсяг трафіку на 30-50 % порівняно з HTTP для типових IoT-сценаріїв.

Протокол CoAP розроблено спеціально для пристроїв з обмеженими ресурсами та працює поверх UDP, що забезпечує нижчу латентність порівняно з TCP-базованими протоколами. CoAP використовує RESTful підхід, подібний до HTTP, але з мінімальними заголовками та підтримкою мультикастингу, що робить його ідеальним для mesh-мереж IoT-пристроїв [25].

Новітнім розробкам належить протокол AMQP, який поєднує надійність enterprise-рішень з ефективністю IoT-протоколів. AMQP забезпечує гарантовану доставку повідомлень, підтримку транзакцій та складну логіку маршрутизації, що робить його привабливим для критичних промислових додатків [26].

Сучасні системи обробки IoT-даних використовують різноманітні архітектурні патерни, кожен з яких має свої переваги та обмеження. Класична централізована архітектура передбачає збір усіх даних на центральному сервері для подальшої обробки та зберігання. Такий підхід забезпечує централізоване управління та спрощує реалізацію складної аналітики, однак створює єдину точку відмови та може страждати від проблем масштабованості при великій кількості пристроїв [27].

Fog computing архітектура розподіляє обробку між хмарою, граничними серверами та кінцевими пристроями. Проміжний рівень fog-вузлів виконує первинну обробку та агрегацію даних, зменшуючи навантаження на хмарну інфраструктуру та знижуючи латентність для часово-критичних додатків. Дослідження демонструють, що fog computing може зменшити латентність на 40-60 % порівняно з чисто хмарними рішеннями для додатків реального часу [28].

Edge computing виносить обчислення безпосередньо на пристрої або шлюзи на краю мережі, що дозволяє обробляти дані максимально близько до джерела їх генерації. Цей підхід особливо важливий для автономних систем, що не можуть покладатися на постійне підключення до хмари, наприклад, для автомобілів з автопілотом або промислових роботів. Edge computing також забезпечує кращу конфіденційність даних, оскільки чутлива інформація може обробляти локально без передачі через мережу [29].

Гібридні архітектури поєднують переваги різних підходів, використовуючи edge обчислення для критичної обробки в реальному часі, fog рівень для агрегації та попередньої аналітики, та хмарні ресурси для глибокого аналізу та довгострокового зберігання. Така багаторівнева архітектура дозволяє оптимально розподілити навантаження та досягти компромісу між латентністю, пропускною здатністю та вартістю системи [30].

Незважаючи на значний прогрес у технологіях обробки сенсорних даних, залишається ряд невирішених проблем та викликів. Гетерогенність IoT-екосистеми, де різні пристрої використовують різні протоколи, формати

даних та стандарти, ускладнює інтеграцію та сумісність систем. Стандартизаційні організації, такі як IEEE та IETF, працюють над розробкою універсальних стандартів, однак процес їх впровадження відбувається повільно через комерційні інтереси виробників [31].

Безпека та конфіденційність даних залишаються критичними проблемами, особливо з урахуванням зростаючої кількості кібератак на IoT-пристрої. Обмежені обчислювальні ресурси датчиків ускладнюють імплементацію надійних криптографічних механізмів, що робить їх уразливими до атак. Перспективним напрямком є розробка легковагових алгоритмів шифрування та аутентифікації, спеціально оптимізованих для IoT-пристроїв.

Енергоефективність залишається фундаментальним обмеженням для автономних IoT-пристроїв, що працюють від батарей. Технології збору енергії з навколишнього середовища, такі як сонячні панелі, п’єзоелектричні генератори або радіочастотне живлення, розвиваються як альтернатива традиційним джерелам енергії. Водночас, оптимізація алгоритмів обробки даних та протоколів комунікації для мінімізації енергоспоживання є активною областю досліджень.

Майбутній розвиток систем обробки сенсорних даних пов’язаний з інтеграцією технологій штучного інтелекту та машинного навчання безпосередньо на рівні edge-пристроїв. Це дозволить реалізувати інтелектуальну обробку даних з мінімальною латентністю та без необхідності передачі величезних обсягів сирих даних до хмари. Розвиток спеціалізованих апаратних прискорювачів для машинного навчання, таких як TPU та NPU, робить цей підхід все більш практичним навіть для пристроїв з обмеженими ресурсами.

1.2 Огляд реляційних баз даних для обробки сенсорних даних

Реляційні бази даних відіграють важливу роль у зберіганні та обробці сенсорних даних у сучасних системах Інтернету речей (IoT) та кіберфізичних системах. Незважаючи на стрімкий розвиток NoSQL-технологій, саме реляційні

системи залишаються фундаментом для побудови надійних і структурованих сховищ даних. Їхня популярність пояснюється високим рівнем цілісності даних, підтримкою транзакцій відповідно до ACID-властивостей (Atomicity, Consistency, Isolation, Durability), широкими можможливостями для аналітики, а також добре розвиненою екосистемою інструментів.

Сенсорні системи генерують велику кількість часових рядів, що потребують стабільної швидкодії при записі та ефективного виконання запитів на читання. За оцінками аналітиків, кількість підключених IoT-пристроїв у світі перевищила 15 мільярдів станом на 2023 рік і продовжує зростати [32]. Кожен із цих пристроїв генерує дані з різною частотою – від кількох разів на секунду до безперервних потоків. Це створює серйозні виклики для систем зберігання даних.

Хоча реляційні СКБД спочатку не були створені для обробки потоків даних у реальному часі, більшість сучасних систем адаптовано до цього завдяки розширенню функцій для роботи з часовими рядами, підтримці індексів за часовими мітками та партиціонуванню великих таблиць. Водночас, реляційні бази даних забезпечують те, що критично важливо для промислових та медичних застосувань – гарантії цілісності даних та можливість проведення складних аналітичних запитів з використанням SQL.

Для обробки сенсорних даних зазвичай застосовують такі популярні реляційні СКБД, як PostgreSQL, MySQL, Microsoft SQL Server та Oracle Database. Кожна з цих систем має свої особливості, переваги та обмеження.

PostgreSQL є однією з найпопулярніших систем з відкритим кодом, що підтримує розширення для роботи з часовими рядами (зокрема, TimescaleDB) та складними структурами даних. Система відома своєю гнучкістю та можливістю розширення функціоналу через власні розширення та функції. PostgreSQL підтримує JSON, XML, масиви та інші нестандартні типи даних, що робить її ідеальним вибором для гетерогенних IoT-систем [33].

Розширення TimescaleDB перетворює PostgreSQL на повноцінну систему для роботи з часовими рядами, автоматично партиціонуючи дані за часом та

оптимізуючи запити для типових сценаріїв роботи з сенсорними даними. Це дозволяє обробляти мільйони записів на секунду при збереженні всіх переваг реляційної моделі [34].

MySQL залишається широко використовуваною системою завдяки своїй швидкодії на читання і простоті розгортання. Система особливо популярна у веб-додатках та легких IoT-рішеннях, де не потрібна складна аналітика. MySQL підтримує різні движки зберігання даних, найпопулярніші з яких – InnoDB (з підтримкою транзакцій) та MyISAM (оптимізований для швидкого читання) [35].

Для роботи з часовими рядами MySQL пропонує партиціонування таблиць за діапазонами дат, що дозволяє ефективно управляти великими обсягами історичних даних. Однак, порівняно з PostgreSQL, MySQL має менше можливостей для розширення та менш розвинений оптимізатор запитів для складних аналітичних завдань.

Microsoft SQL Server пропонує потужні засоби аналітики, оптимізатор запитів та інтеграцію з хмарними сервісами Azure. Система включає вбудовані засоби для машинного навчання (SQL Server Machine Learning Services), аналітики в пам’яті (In-Memory OLTP) та роботи з великими даними [36].

Для IoT-застосунків Microsoft пропонує інтеграцію з Azure IoT Hub та Azure Stream Analytics, що дозволяє будувати комплексні рішення для обробки потокових даних. SQL Server підтримує темпоральні таблиці (temporal tables), які автоматично зберігають історію змін даних – функція, корисна для аудиту сенсорних вимірювань.

Oracle Database вирізняється високою стабільністю, масштабованістю та підтримкою складних бізнес-сценаріїв, проте вимагає значних ресурсів і є комерційним рішенням. Система пропонує найпотужніші можливості для горизонтального та вертикального масштабування, підтримує розподілені транзакції та має найрозвиненіші засоби оптимізації запитів [37].

Oracle пропонує спеціалізовані рішення для роботи з часовими рядами та IoT-даними, включаючи Oracle Database In-Memory для прискорення аналітичних запитів та Oracle NoSQL Database для гібридних сценаріїв. Однак

висока вартість ліцензій робить Oracle менш доступною для малих та середніх проектів.

Для оцінки придатності реляційних систем у контексті сенсорних даних проведено порівняльний аналіз за кількома ключовими параметрами: продуктивність, масштабованість, підтримка часових рядів, зручність інтеграції з IoT-пристроями, вартість володіння та рівень спільноти розробників. Порівняльна характеристика представлена у таблиці 1.1.

Таблиця 1.1 – Порівняльна характеристика популярних реляційних баз даних

Система	Тип ліцензії	Підтримка часових рядів	Масштабованість	Продуктивність при потокових даних	Зручність інтеграції з IoT
PostgreSQL	Відкрита (BSD)	Висока (через TimescaleDB)	Висока	Висока (до 150K записів/с)	Висока (REST/API, MQTT)
MySQL	Відкрита (GPL)	Середня	Середня	Висока (до 100K записів/с)	Висока (широка підтримка)
Microsoft SQL Server	Комерційна, Express	Висока (темпоральні таблиці)	Висока	Висока (до 200K записів/с)	Висока (Azure IoT)
Oracle Database	Комерційна	Висока (спеціалізовані модулі)	Дуже висока	Дуже висока (до 300K записів/с)	Середня (потрібна конфігурація)

Продуктивність при потокових даних є критичним параметром для IoT-систем. Тестування показує, що Oracle Database демонструє найвищу пропускну здатність, особливо при використанні In-Memory опції. PostgreSQL з TimescaleDB показує відмінні результати для систем малого та середнього масштабу, забезпечуючи оптимальне співвідношення ціна-якість [38].

Масштабованість визначає здатність системи обробляти зростаючі обсяги даних. Oracle пропонує найкращі можливості для горизонтального масштабування через Real Application Clusters (RAC), але PostgreSQL з Citus та SQL Server з Always On також забезпечують достатню масштабованість для більшості застосувань.

Підтримка часових рядів включає наявність спеціалізованих індексів, функцій агрегації та можливостей для ефективного зберігання та запиту історичних даних. TimescaleDB для PostgreSQL вважається одним з найкращих рішень у цій категорії завдяки автоматичному партиціонуванню та оптимізованим функціям роботи з часом.

При проектуванні схеми бази даних для зберігання сенсорних даних важливо врахувати специфіку часових рядів [39]. Основні підходи представлено у таблиці 1.2.

Таблиця 1.2 – Основні архітектурні підходи для зберігання сенсорних даних

Назва підходу	Тип підходу	Переваги	Недоліки
Підхід з широкими таблицями	Одна таблиця для кожного типу сенсора	Простота запитів, швидкість читання для аналітики	Низька гнучкість при додаванні нових типів сенсорів
Підхід зі вузькими таблицями	Зберігання різнорідних даних	Висока гнучкість, легке додавання нових метрик	Складніші запити, потенційно нижча продуктивність
Гібридний підхід з партиціонуванням	Поєднання переваг підходів через партиціонування	Максимальна гнучкість, ефективне управління життєвим циклом даних, оптимізація простору зберігання, можливість паралельної обробки різних партицій	Складність адміністрування, обмеження при запитах між партиціями, складність міграцій схеми, ризик фрагментації даних, накладні витрати на планування запитів

Важливим фактором також є популярність баз даних серед розробників, адже чим більш популярна певна СКБД, тим більша ймовірнсть, що ця система буде розвиватись і постійно оновлювати всій функціонал. На рисунку 1.2 представлено графік популярності уже загаданих СКБД за 2020-2024 рр [40].

Згідно графіка, лідер зростання – PostgreSQL (+22 %). Система показує найбільш динамічне зростання з 57 % до 79 %. Основні причини їх успіху: розширення TimescaleDB перетворило PostgreSQL на спеціалізоване рішення для часових рядів, відсутність ліцензійних витрат знижує бар’єр входу для проектів будь-якого масштабу, активна спільнота з понад 600 тисяч розробників

забезпечує швидке вирішення проблем, постійне оновлення функціоналу: підтримка JSONB, паралельні запити, декларативне партиціонування.

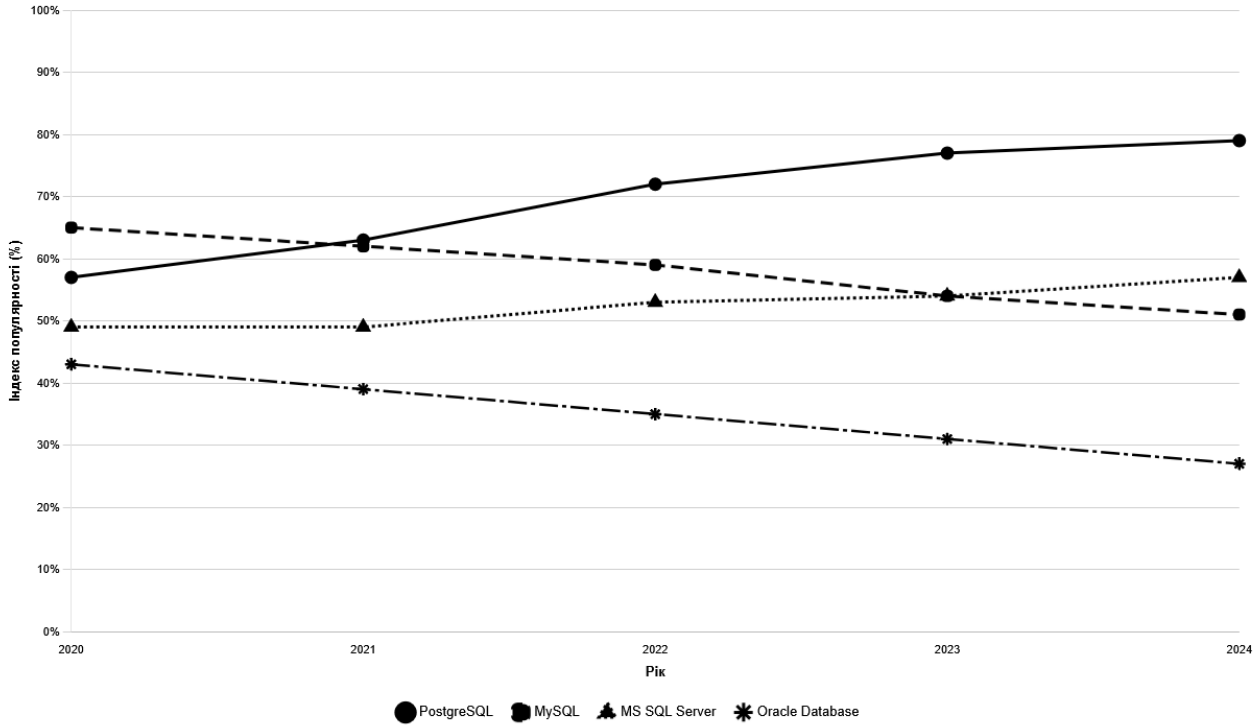

Рисунок 1.2 – Динаміка популярності реляційних СКБД для IoT застосунків [40]

Другою за популярністю є – MySQL (-14 %). Попри зниження з 65 % до 51 %, MySQL залишається у топ-3. Зберігає позиції у веб-додатках та простих IoT-рішеннях, швидка та легка у розгортанні. Основні втрати ринку через те, що більшість складних систем була мігрована до PostgreSQL, а також через те, що вона має обмежені можливості для глибокої аналітики часових рядів.

Помірний, але стабільний приріст (з 49 % до 57 %) у популярності має Microsoft SQL Server (+8 %). Це відбулось через те, що ця база даних має сигнергію з екосистемою Microsoft (Azure IoT Hub, Power BI, Azure ML), сильні позиції в корпоративному сегменті, автоматизацію через Azure SQL Database Managed Instance, а аткож має вбудовані можливості для машинного навчання, що надають IoT-проєктам додаткову аналітику.

Найбільше зниження (з 43 % до 27 %) має Oracle Database (-16 %). Це відбулось через високу вартість ліцензій. Здебільшого основними гравцями в сфері IoT є стартапи, а вони в свою чергу обирають безкоштовні альтернативи. Також, таке зниження може бути пояснене складністю обслуговування та розгортання. Ця СКБД зберегла лідерство лише у критично важливих корпоративних системах.

1.3 Огляд NoSQL баз даних для обробки сенсорних даних

NoSQL бази даних посідають важливе місце в архітектурі сучасних сенсорних систем, особливо в умовах постійного зростання обсягів і швидкості надходження даних [41]. Термін NoSQL (Not Only SQL) охоплює широкий спектр нереляційних систем зберігання даних, які відмовляються від традиційної табличної моделі на користь більш гнучких структур – документних, графових, стовпчикових і ключ-значення [42]. Основною причиною популярності NoSQL у сфері Інтернету речей (IoT) та кіберфізичних систем є здатність таких баз масштабуватися горизонтально, ефективно працювати з неструктурованими або напівструктурованими даними, а також забезпечувати швидку обробку потоків сенсорних значень у реальному часі.

Сенсорні системи, на відміну від традиційних бізнес-додатків, генерують величезну кількість коротких записів із часовими мітками, що потребують швидкого збереження, агрегації та подальшого аналізу. Для таких задач реляційні системи часто виявляються менш ефективними через жорстку схему даних і складність масштабування. Саме тому NoSQL бази даних, завдяки своїй гнучкості, стають оптимальним рішенням для зберігання телеметрії, логів, часових рядів і даних від великої кількості сенсорів.

Документно-орієнтовані системи, такі як MongoDB та CouchDB, зберігають дані у вигляді документів формату JSON або BSON. Ця модель ідеально підходить для сенсорних систем з різнорідними типами пристроїв, оскільки кожен сенсор може мати унікальний набір параметрів без необхідності

змінювати схему бази даних. Наприклад, температурний сенсор може зберігати тільки температуру та часову мітку, тоді як мультисенсорний пристроїв може включати температуру, вологість, тиск, координати GPS та рівень батареї в одному документі.

MongoDB забезпечує високу швидкість читання та запису, підтримує індексацію геопросторових даних, що критично важливо для мобільних сенсорів та систем відстеження. Завдяки агрегаційному фреймворку MongoDB можна виконувати складні аналітичні запити безпосередньо в базі даних, не потребуючи додаткових інструментів обробки [43].

Apache Cassandra та HBase представляють категорію стовпчикових (широкостовпчикових) баз даних, оптимізованих для запису великих обсягів даних з мінімальною затримкою. Архітектура Cassandra базується на принципі розподіленого зберігання без єдиної точки відмови, що робить її надзвичайно надійною для критичних IoT-систем. Система використовує узгоджене хешування для розподілу даних по вузлах кластера, забезпечуючи лінійну масштабованість.

У сенсорних системах Cassandra часто використовується для зберігання телеметрії від промислових датчиків, де важливі як швидкість запису (до мільйонів записів на секунду), так і можливість виконання аналітичних запитів за часовими діапазонами. Модель даних Cassandra дозволяє ефективно організовувати дані за принципом «одна таблиця на запит», що оптимізує продуктивність читання [44].

InfluxDB, TimescaleDB та Prometheus представляють клас спеціалізованих систем управління базами даних часових рядів (Time Series Database, TSDB). Ці системи розроблені спеціально для роботи з даними, що містять часові мітки, і забезпечують оптимізовані механізми стиснення, агрегації та зберігання таких даних.

InfluxDB використовує власну мову запитів Flux, яка дозволяє виконувати складні трансформації даних, включаючи обчислення ковзних середніх, виявлення аномалій та прогнозування. Система автоматично видаляє старі дані

згідно з політиками збереження (retention policies), що критично важливо для управління обмеженими ресурсами зберігання в IoT-інфраструктурі .

Для сенсорних мереж, де важлива швидка візуалізація та моніторинг в реальному часі, InfluxDB часто інтегрується з платформами візуалізації, такими як Grafana, забезпечуючи повний стек для збору, зберігання та відображення телеметрії [45].

Redis та Memcached являють собою системи зберігання типу «ключ-значення» з надзвичайно високою швидкістю доступу. Redis працює повністю в оперативній пам’яті, що дозволяє досягати мікросекундної затримки при операціях читання та запису. У контексті сенсорних систем Redis часто використовується як кеш для найсвіжіших даних або як брокер повідомлень для обробки потоків даних у реальному часі.

Redis підтримує різні структури даних, включаючи списки, множини, відсортовані множини та хеш-таблиці, що дозволяє реалізовувати складну логіку обробки даних безпосередньо в базі. Наприклад, можна використовувати відсортовані множини для зберігання останніх N вимірювань від кожного сенсора з автоматичним видаленням найстаріших записів [46].

Для об’єктивної оцінки ефективності різних NoSQL систем у контексті сенсорних даних розглянемо ключові параметри, що визначають їхню придатність для таких задач. Порівняльна характеристика подана у таблиці 1.3.

Таблиця 1.3 – Порівняльний аналіз NoSQL систем для сенсорних даних

Система	Тип	Масштабо-ваність	Продуктивність при потокових даних	Підтримка часових рядів	Зручність інтеграції з IoT
MongoDB	Документна	Висока	Висока	Середня	Висока (REST/API, MQTT)
Cassandra	Стовпчикова	Дуже висока	Дуже висока	Висока	Висока
InfluxDB	Time-series	Висока	Дуже висока	Дуже висока	Дуже висока
Redis	Ключ-значення	Висока	Дуже висока	Низька	Середня
TimescaleDB	Time-series (SQL)	Висока	Висока	Дуже висока	Висока

Масштабованість визначає здатність системи обробляти зростаючі обсяги даних шляхом додавання нових вузлів до кластера. Cassandra та InfluxDB демонструють найкращі результати завдяки архітектурі, що підтримує горизонтальне масштабування без простоїв.

Продуктивність при потокових даних критична для систем, де сенсори генерують постійний потік вимірювань. InfluxDB, Cassandra та Redis оптимізовані для високошвидкісного запису, досягаючи швидкості до 1 мільйона записів на секунду на одному вузлі.

Підтримка часових рядів включає нативні функції для роботи з даними з часовими мітками, автоматичне агрегування, політики збереження та оптимізоване стиснення. InfluxDB та TimescaleDB розроблені спеціально для цього, тоді як MongoDB та Redis потребують додаткової конфігурації.

До основних архітектурних підходів в NoSQL базах даних можна віднести: Lambda-архітектура, Kappa-архітектура та Hybrid-підхід.

Lambda-архітектура поєднує пакетну обробку великих історичних даних з обробкою потоків у реальному часі. У типовій реалізації для IoT Cassandra або InfluxDB використовуються для зберігання повної історії даних (batch layer), тоді як Redis або Apache Kafka обробляють потокові дані (speed layer). Результати обох шарів об’єднуються на рівні представлення (serving layer) [47].

Kappa-архітектура спрощує Lambda, використовуючи єдиний потоковий конвеєр для всієї обробки даних. Apache Kafka як розподілений журнал подій поєднується з InfluxDB для зберігання агрегованих результатів. Цей підхід зменшує складність системи та забезпечує більш передбачувану затримку обробки [48].

Багато сучасних IoT-платформ використовують гібридний підхід, поєднуючи кілька типів NoSQL баз для різних цілей. Наприклад, Redis для кешування свіжих даних та швидкого доступу, InfluxDB для зберігання та аналізу часових рядів, а MongoDB для зберігання метаданих пристроїв та конфігурацій.

Також, не менш важливим фактором є популярність. Адже завдяки змінам та пропозиціям від спільноти сучасні системи мають здатність розвиватись та покращувати свій функціонал [49]. На рисунку 1.3 предсталевно популярність різноманітних NoSQL баз даних.

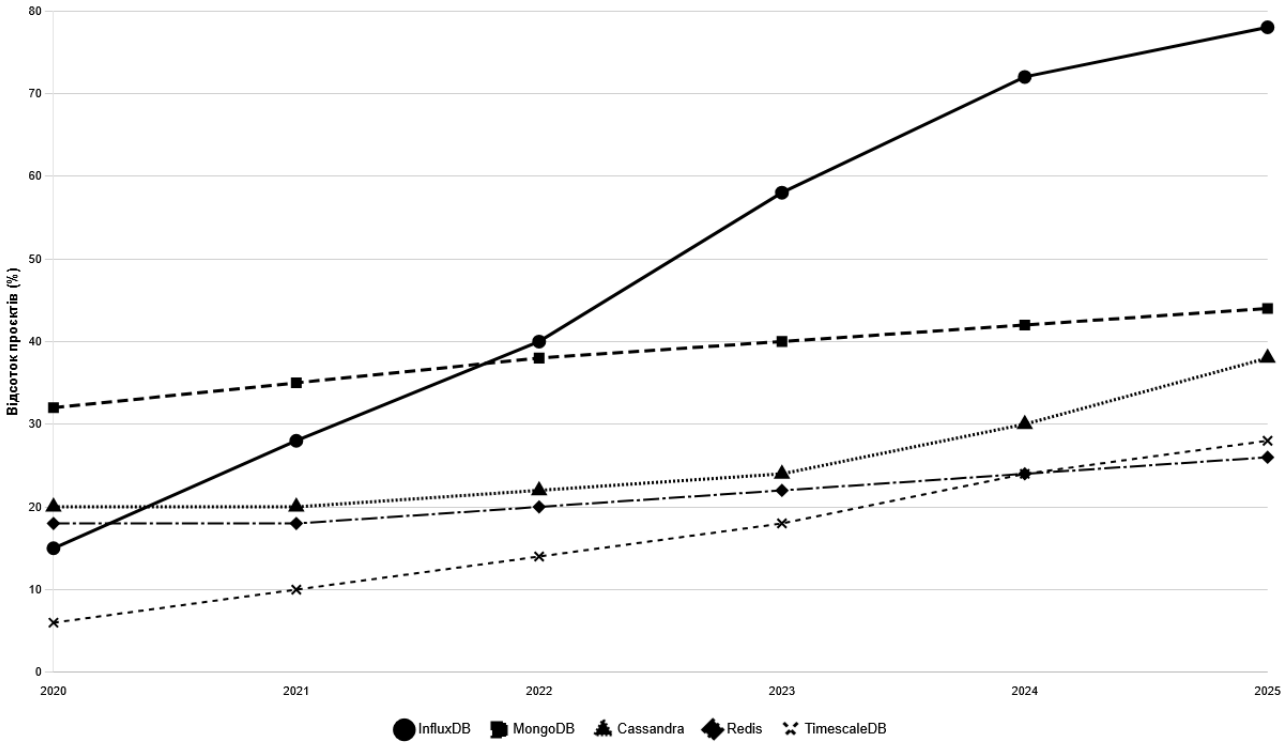

Рисунок 1.3 – Популярність NoSQL баз даних в IoT системах [49]

На графіку відображено зміну частки використання різних баз даних у період з 2020 по 2025 рік. Найбільш помітним є стрімке зростання популярності InfluxDB, частка якої зросла з близько 15 % у 2020 році до майже 80 % у 2025. Це свідчить про активне впровадження цієї бази даних у різних сферах, що пов’язано з її ефективністю для роботи з великими обсягами часових рядів та сенсорних даних.

MongoDB демонструє більш стабільне зростання, піднімаючись з 32 % у 2020 до приблизно 43 % у 2025. Така тенденція свідчить про збереження високої популярності цієї бази даних, хоча її темпи росту поступаються InfluxDB. Cassandra показує повільне і рівномірне збільшення частки використання з 18 % до 25 %, залишаючись стабільним інструментом для масштабованих рішень.

Варто відзначити відносно швидке зростання Redis з 8 % до 38 %, особливо після 2022 року, що свідчить про збільшення попиту на високопродуктивні інструменти для кешування та обробки даних у реальному часі. TimescaleDB також демонструє поступовий ріст з 5 % до 28 %, трохи обганяючи Cassandra після 2023 року, що вказує на розширення використання баз даних для часових рядів.

Загалом, аналіз графіку свідчить про те, що InfluxDB стає безумовним лідером ринку за популярністю у розглянутий період, тоді як інші бази даних, такі як Redis та TimescaleDB, демонструють прискорене зростання після 2022 року. Це відображає тенденцію до активного використання сучасних баз даних, оптимізованих під обробку великих потоків даних і сенсорної інформації.

РОЗДІЛ 2
ТЕОРЕТИЧНІ ОСНОВИ ВИБОРУ ТЕХНОЛОГІЙ ДЛЯ ОБРОБКИ СЕНСОРНИХ ДАНИХ

2.1 Визначення ключових критеріїв ефективності баз даних

Ефективність баз даних є комплексним поняттям, що охоплює множину взаємопов'язаних характеристик системи управління даними. Для всебічної оцінки функціонування бази даних необхідно враховувати декілька ключових критеріїв, які безпосередньо впливають на продуктивність та якість роботи інформаційної системи в цілому. Основні критерії ефективності баз даних зобарежно на рисунку 2.1.

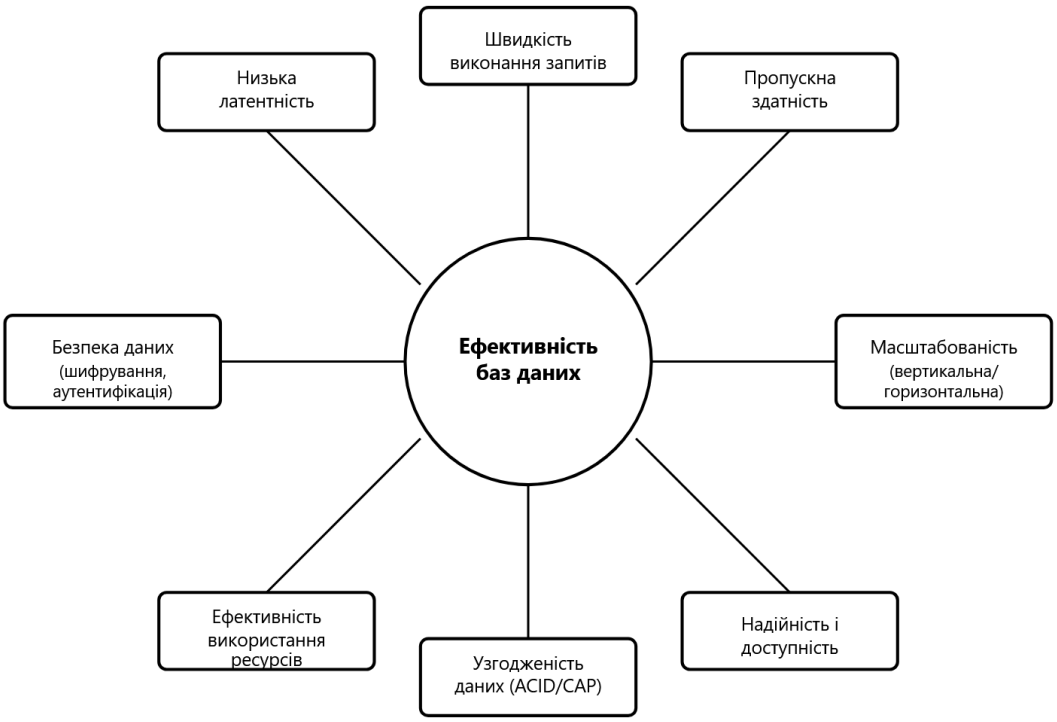

Рисунок 2.1 – Основні критерії ефективності баз даних

Одним із найважливіших критеріїв є швидкість виконання запитів, яка визначає час відгуку системи на операції читання та запису даних. Цей показник безпосередньо впливає на користувацький досвід та загальну продуктивність додатків. Швидкість виконання залежить від оптимізації структури бази даних,

наявності та якості індексів, ефективності алгоритмів обробки запитів та апаратних характеристик сервера. Дослідження показують, що оптимізація запитів може зменшити час їх виконання на 50-90 % залежно від складності структури даних [50].

Пропускна здатність системи характеризує кількість транзакцій або операцій, які база даних може обробити за одиницю часу. Цей критерій особливо важливий для високонавантажених систем, де одночасно виконується велика кількість запитів. Вимірювання пропускної здатності здійснюється в транзакціях за секунду (TPS) або запитах за секунду (QPS). Для систем електронної комерції типові значення можуть сягати десятків тисяч транзакцій на секунду.

Масштабованість визначає здатність бази даних ефективно обробляти зростаючі обсяги даних та навантаження без суттєвого погіршення продуктивності. Розрізняють вертикальну масштабованість, що передбачає нарощування потужності існуючого обладнання, та горизонтальну, яка полягає у додаванні нових вузлів до розподіленої системи. Особливо актуальною є еластична масштабованість, що дозволяє динамічно адаптувати ресурси відповідно до поточного навантаження.

Надійність та доступність даних є критично важливими критеріями, особливо для систем, що працюють у режимі 24/7. Надійність передбачає захист від втрати даних через апаратні збої, програмні помилки або людський фактор, тоді як доступність визначає відсоток часу, протягом якого система залишається працездатною. Для критично важливих систем доступність вимірюється у кількості «дев’яток» – наприклад, доступність 99,99 % означає, що система може бути недоступною не більше 52 хвилин на рік.

Узгодженість даних гарантує, що всі операції з базою даних виконуються коректно та не порушують цілісності інформації. У контексті розподілених систем особливого значення набуває теорема CAP, яка описує компроміс між узгодженістю, доступністю та стійкістю до розділення мережі. Класичні реляційні бази даних надають перевагу строгій узгодженості, забезпечуючи

виконання ACID-властивостей, тоді як багато NoSQL систем обирають модель евентуальної узгодженості [51].

Ефективність використання ресурсів включає оптимізацію споживання процесорного часу, оперативної пам’яті, дискового простору та мережевої пропускної здатності. Раціональне використання апаратних ресурсів дозволяє знизити експлуатаційні витрати та підвищити загальну ефективність системи. Оптимізація використання пам’яті включає ефективне кешування найчастіше використовуваних даних, мінімізацію фрагментації та використання компресії даних.

Безпека даних охоплює захист інформації від несанкціонованого доступу, модифікації або знищення. Цей критерій включає механізми аутентифікації, авторизації, шифрування даних як при зберіганні, так і при передачі, а також аудит операцій з базою даних. Відповідність вимогам регуляторних стандартів, таких як GDPR або HIPAA, також є важливою складовою безпеки. Системи аудиту реєструють всі операції з даними, що дозволяє виявляти підозрілу активність та розслідувати інциденти безпеки [52].

Гнучкість та простота адміністрування визначають легкість управління базою даних, виконання рутинних операцій обслуговування та адаптації до змінних вимог бізнесу. Ефективні інструменти моніторингу дозволяють відстежувати ключові метрики продуктивності в реальному часі, виявляти потенційні проблеми на ранніх стадіях та аналізувати тренди використання ресурсів.

Латентність або затримка виконання операцій є критично важливим показником для інтерактивних додатків та систем реального часу. Низька латентність забезпечує швидкий відгук системи на запити користувачів, що є ключовим фактором позитивного користувацького досвіду. Для різних типів додатків вимоги до латентності можуть суттєво відрізнятися – від мілісекунд для фінансових торгових систем до секунд для аналітичних запитів [53].

Цілісність даних забезпечує коректність та несуперечливість інформації в базі даних протягом усього життєвого циклу. Механізми підтримки цілісності

включають обмеження на рівні стовпців та таблиць, тригери, збережені процедури та зовнішні ключі. Референційна цілісність гарантує, що зв’язки між таблицями залишаються узгодженими, а транзакційна цілісність забезпечує повне виконання або відкат всіх операцій в межах транзакції.

Здатність до відновлення визначає можливість системи повертатися до працездатного стану після збоїв різного характеру. Ефективні механізми відновлення включають журналювання транзакцій, контрольні точки та автоматичне виявлення пошкоджень даних. Журнал транзакцій зберігає історію всіх змін даних, що дозволяє відтворити стан бази даних на будь-який момент часу [54].

Сумісність та підтримка стандартів визначають здатність бази даних працювати з різними платформами, операційними системами та інструментами розробки. Підтримка міжнародних стандартів SQL забезпечує переносимість додатків між різними СУБД, а наявність стандартних інтерфейсів програмування спрощує інтеграцію з різними мовами програмування.

Вартість володіння є комплексним економічним критерієм, що включає початкові витрати на придбання ліцензій або розгортання системи, а також поточні експлуатаційні витрати на обладнання, електроенергію, адміністрування та підтримку. Для Open Source рішень початкові витрати можуть бути мінімальними, проте витрати на кваліфікованих фахівців можуть бути значними. Хмарні рішення пропонують модель оплати за використання, що може бути вигідною для проектів зі змінним навантаженням [55].

Таким чином, визначення та всебічна оцінка ключових критеріїв ефективності баз даних є фундаментальною основою для прийняття обґрунтованих рішень щодо вибору, проектування та оптимізації систем управління базами даних. Кожен із розглянутих критеріїв має своє значення залежно від специфіки конкретного застосування, і оптимальне рішення часто полягає у пошуку балансу між різними вимогами.

2.2 Огляд можливих стратегій оптимізації продуктивності для SQL та NoSQL баз даних

Оптимізація продуктивності баз даних є критично важливою для розробки та супроводу інформаційних систем, що безпосередньо впливає на швидкість обробки даних, ефективність використання ресурсів та загальну якість роботи додатків. Різні типи баз даних – реляційні SQL та нереляційні NoSQL – потребують специфічних підходів до оптимізації, які враховують їхні архітектурні особливості та призначення [56].

Стратегії оптимізації для SQL баз даних традиційно зосереджуються на структурній організації даних та ефективності виконання запитів. Індексування є одним із найпотужніших інструментів оптимізації реляційних баз даних, що дозволяє значно прискорити пошук та вибірку даних. Правильно побудовані індекси можуть скоротити час виконання запитів у десятки і навіть сотні разів, особливо для великих таблиць. Однак надмірне індексування може призвести до сповільнення операцій вставки та оновлення даних, тому необхідно знаходити оптимальний баланс.

На рисунку 2.2 представлено комплексну схему основних стратегій оптимізації SQL баз даних, що включає методи індексування, структурування даних, оптимізації запитів, кешування, налаштування параметрів системи, моніторингу та апаратної оптимізації. Ця схема демонструє взаємозв’язок різних підходів та їхній сукупний вплив на підвищення продуктивності реляційних систем управління базами даних.

Нормалізація та денормалізація даних представляють дві протилежні стратегії організації структури реляційних баз даних. Нормалізація спрямована на усунення надмірності даних та забезпечення їхньої цілісності шляхом розділення інформації на окремі логічно пов’язані таблиці. Денормалізація, навпаки, передбачає об’єднання даних з різних таблиць для зменшення кількості операцій з’єднання та прискорення читання. Вибір між цими підходами залежить

від характеру навантаження – для систем з переважанням операцій читання доцільнішою може бути денормалізація [57].

Оптимізація запитів включає перепис SQL-виразів для підвищення їхньої ефективності, використання підказок оптимізатору та аналіз планів виконання запитів. Важливими аспектами є уникнення повних сканувань таблиць, мінімізація кількості підзапитів, використання відповідних типів з’єднань та обмеження вибірки лише необхідними стовпцями. Регулярний аналіз повільних запитів допомагає виявляти проблемні місця та своєчасно їх усувати.

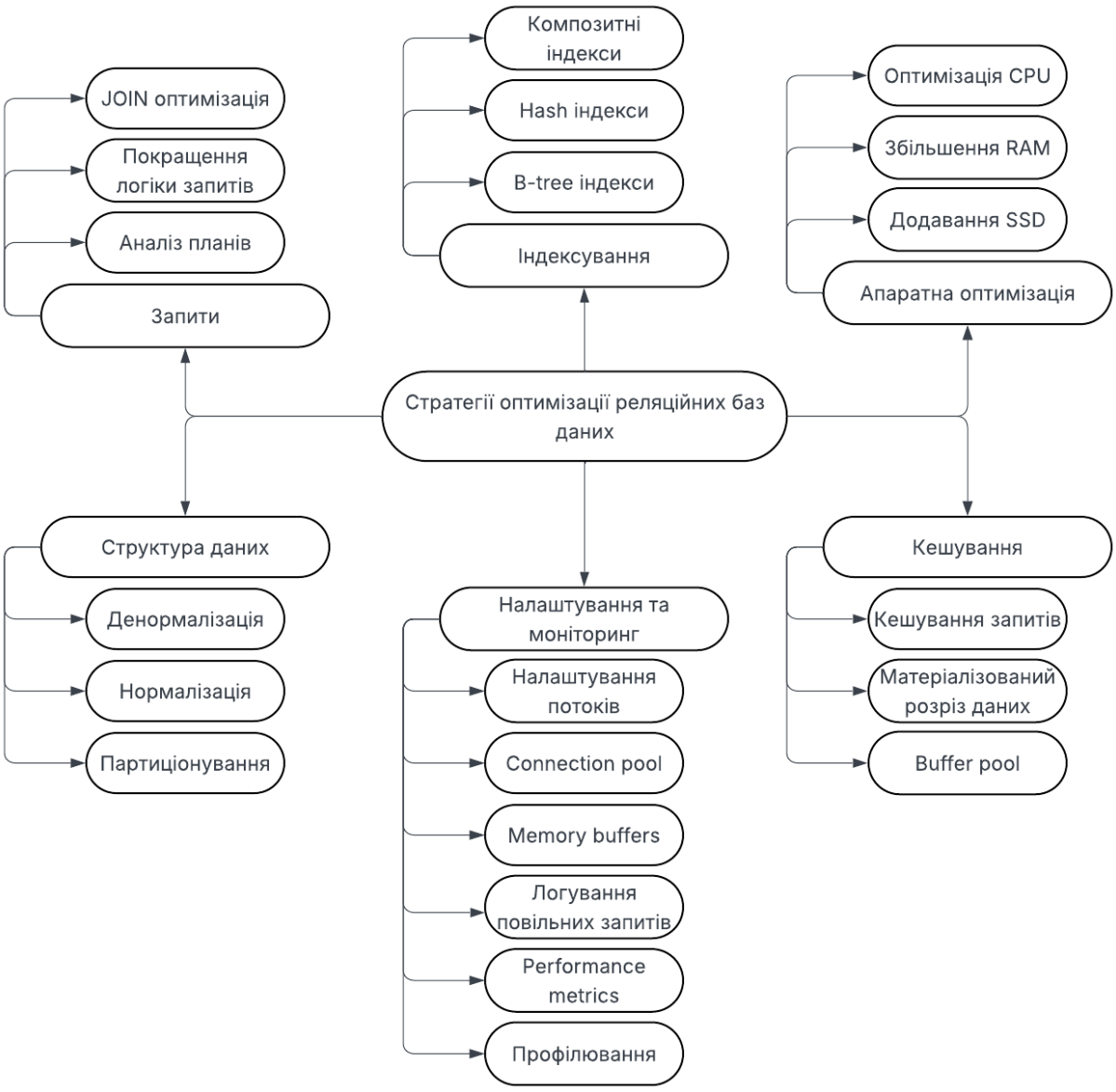

Рисунок 2.2 – Стратегії оптимізації реляційних баз даних

Партиціонування або розділення таблиць на менші фізичні сегменти дозволяє покращити продуктивність за рахунок паралельної обробки даних та зменшення обсягу даних, що сканується при виконанні запитів. Існують різні стратегії партиціонування – по діапазону значень, по списку, по хешу або композитне. Партиціонування особливо ефективне для великих таблиць з часовими даними або географічно розподіленою інформацією [58].

Кешування результатів запитів та використання матеріалізованих представлень дозволяє зберігати попередньо обчислені дані для швидкого доступу без повторного виконання складних операцій. Матеріалізовані представлення особливо корисні для аналітичних запитів, що виконуються над великими обсягами даних. Ефективна стратегія кешування повинна враховувати частоту оновлення даних та баланс між актуальністю інформації та швидкістю доступу.

Стратегії оптимізації NoSQL баз даних суттєво відрізняються від підходів для реляційних систем через фундаментальні архітектурні відмінності. Вибір відповідної моделі даних є першочерговим завданням при роботі з NoSQL системами. Для документоорієнтованих баз даних важливо правильно структурувати документи, вбудовуючи пов’язані дані там, де це доцільно, замість створення посилань. Для графових баз даних ключовим є оптимальне моделювання вузлів та ребер [59].

Рисунок 2.3 ілюструє ключові напрямки оптимізації NoSQL баз даних, що включають проектування моделі даних, стратегії шардингу, механізми реплікації, налаштування рівня узгодженості, оптимізацію операцій читання-запису, індексування та методи компресії. Схема відображає специфічні для NoSQL підходи, спрямовані на досягнення горизонтального масштабування та високої доступності системи.

Шардинг або горизонтальне розділення даних є природним механізмом масштабування NoSQL систем, що дозволяє розподілити навантаження між множиною серверів. Ефективна стратегія шардингу повинна забезпечувати рівномірний розподіл даних та мінімізацію кількості міжвузлових запитів. Вибір

ключа шардингу критично впливає на продуктивність системи та можливість її подальшого масштабування.

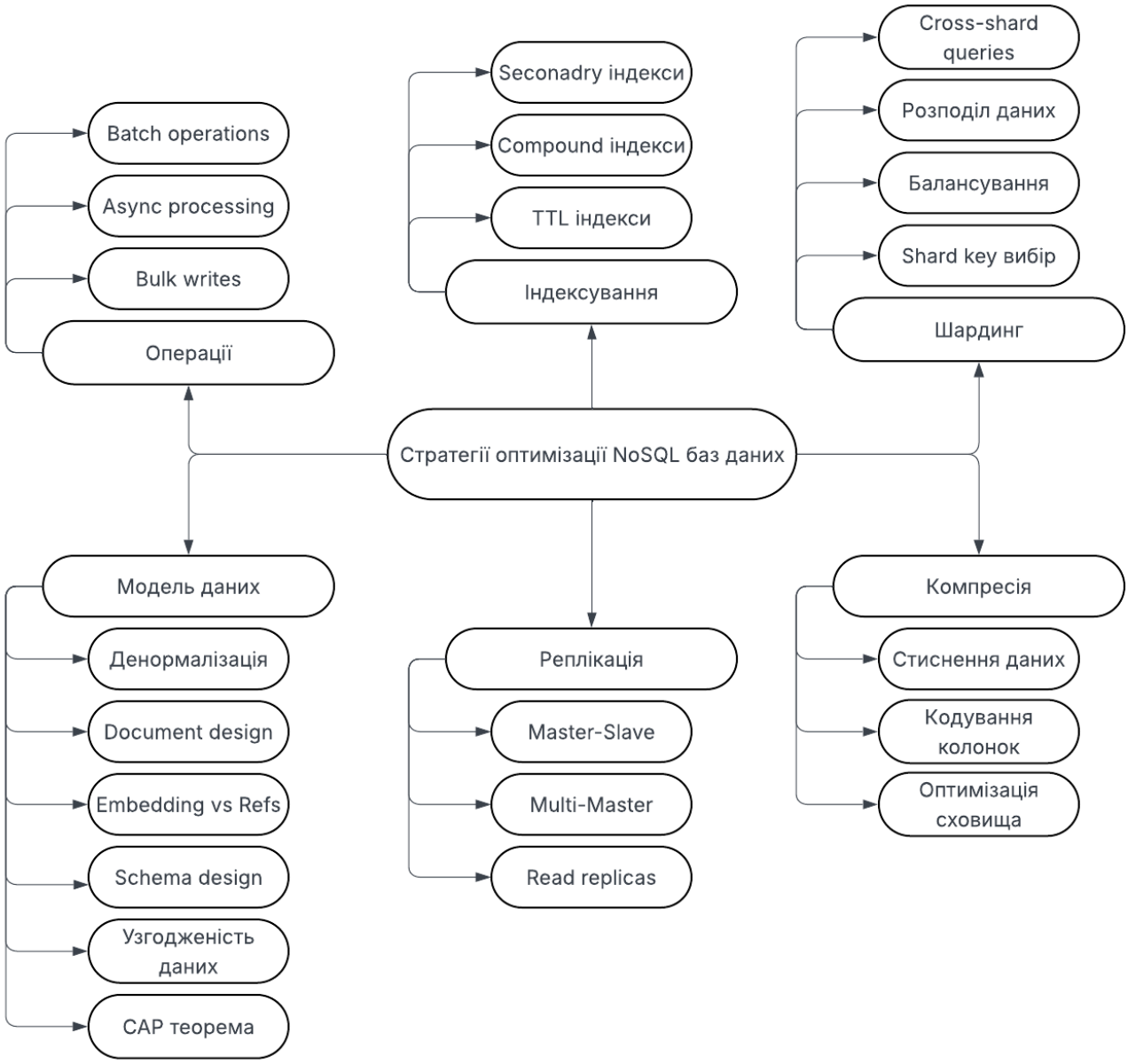

Рисунок 2.3 – Стратегії оптимізації NoSQL баз даних

Налаштування рівня узгодженості в NoSQL системах дозволяє знайти баланс між продуктивністю та гарантіями коректності даних. Для багатьох додатків прийнятною є евентуальна узгодженість, що забезпечує вищу доступність та швидкодію. Однак для критичних операцій може знадобитися

строга узгодженість, що вимагає додаткових накладних витрат на синхронізацію між вузлами системи [60].

Оптимізація операцій читання та запису в NoSQL базах включає використання пакетної обробки, асинхронних операцій та ефективних патернів доступу до даних. Багато NoSQL систем оптимізовані для специфічних типів навантаження – наприклад, колонкові бази даних для аналітичних запитів, а key-value сховища для швидкого точкового доступу. Розуміння цих особливостей дозволяє вибрати найбільш відповідну технологію для конкретного застосування.

Порівняльний аналіз підходів до оптимізації SQL та NoSQL баз даних, представлений у таблиці 2.1, демонструє фундаментальні відмінності у стратегіях оптимізації цих двох типів систем управління базами даних. Таблиця систематизує інформацію за ключовими критеріями: структура даних, масштабування, індексування, узгодженість, оптимізація запитів, кешування та типове використання, що дозволяє оцінити переваги та недоліки кожного підходу.

Таблиця 2.1 – Порівняльна таблиця підходів до оптимізації реляційних та NoSQL баз даних

Критерій	Реляційні бази даних	NoSQL бази даних
Структура даних	Нормалізація таблиць, жорстка схема	Гнучка схема, вбудовування, денормалізація
Масштабування	Вертикальне (Scale-up), Read replicas	Горизонтальне (Scale-out), шардинг, розподілені системи
Індексування	B-tree, Hash, Bitmap, Full-text індекси	Secondary, Compaound, Geospatial, TTL індекси
Узгодженість	ACID властивості, узгодженість даних	BASE, CAP теорема, мінімальна узгодженість даних
Оптимізація запитів	Query optimization, Execution plans, JOIN оптимізація	Access patterna, Batch operations, Pipline queries
Кешування	Query cache, Materialized views, Buffer pool	In-memory caching, Read-through cache
Типове використання	Транзакційні системи, складні звіти, фінанси	Big Data, Real-time, High-load, IoT, Social media

Моніторинг та профілювання є універсальними стратегіями оптимізації для обох типів баз даних. Регулярний збір метрик продуктивності, аналіз патернів використання та виявлення вузьких місць дозволяють проактивно виявляти та усувати проблеми до того, як вони вплинуть на користувачів. Сучасні інструменти моніторингу надають детальну інформацію про використання ресурсів, час виконання запитів та стан системи в реальному часі. Оптимізація апаратного забезпечення та конфігурації системи включає правильний вибір типу дисків, налаштування параметрів пам’яті, мережевих з’єднань та операційної системи. Використання SSD-дисків замість традиційних HDD може значно покращити продуктивність операцій вводу-виводу. Налаштування розміру буферів, кешів та параметрів конкурентності дозволяє максимально ефективно використовувати наявні апаратні ресурси.

Стратегії резервного копіювання та відновлення також впливають на загальну продуктивність системи. Інкрементальне резервне копіювання та використання реплік для читання дозволяють зменшити навантаження на основний сервер. Автоматизація процесів обслуговування, таких як очищення логів, перебудова індексів та оновлення статистики, забезпечує стабільну продуктивність системи протягом тривалого часу.

Таким чином, ефективна оптимізація продуктивності баз даних вимагає комплексного підходу, що поєднує розуміння архітектурних особливостей конкретної технології, аналіз специфічних вимог додатку та систематичний моніторинг роботи системи. Правильне застосування розглянутих стратегій дозволяє досягти оптимального балансу між продуктивністю, надійністю та вартістю володіння системою.

2.3 Вибір технологічного стеку для дослідження та критерії оцінки ефективності

Вибір технологічного стеку для системи моніторингу якості повітря є ключовим етапом проектування, оскільки від нього залежить не лише

функціональність системи, але й можливість проведення об’єктивного порівняльного аналізу ефективності реляційних та NoSQL баз даних. Технологічний стек повинен забезпечувати стабільну роботу всіх компонентів системи, від апаратного рівня збору даних до рівня їх візуалізації та аналізу.

Апаратна частина системи побудована на базі мікроконтролера ESP32-S2, який забезпечує необхідну обчислювальну потужність та можливості бездротового зв’язку для передачі даних на сервер. ESP32-S2 характеризується низьким енергоспоживанням, наявністю вбудованого Wi-Fi модуля та достатнім обсягом оперативної пам’яті для обробки даних з датчиків [61]. Мікроконтролер працює на частоті до 240 МГц та має 320 КБ оперативної пам’яті, що дозволяє виконувати складні обчислення в реальному часі та підтримувати стабільне з’єднання з сервером через протокол MQTT.

Для вимірювання параметрів якості повітря використовується датчик BME-680 (рис. 2.4), який здатен визначати температуру, вологість, атмосферний тиск та якість повітря через вимірювання летких органічних сполук. Датчик розраховує індекс якості повітря (IAQ), який є комплексним показником стану повітря в діапазоні від 0 до 500, де нижчі значення відповідають кращій якості повітря (табл. 2.2). Цей датчик обирається через його високу точність, низьке енергоспоживання та можливість інтеграції через інтерфейс I2C або SPI [62]. BME-680 здійснює вимірювання з частотою один раз на хвилину, що забезпечує оптимальний баланс між актуальністю даних та енергоефективністю системи.

Рисунок 2.4 – Датчик BME-680

Таблиця 2.2 – Таблиця значень IAQ

Індекс IAQ	Якість повітря	Колір
0-50	Добра	Зелений (#00FF00)
51-100	Посередня	Жовтий (#FFFF00)
101-150	Погана	Помаранчевий (#FF9900)
151-200	Дуже погана	Червоний (#FF0000)
201-300	Надзвиайно погана	Фіолетовий (#800080)
301-500	Небезпечна	Чорний (#000000)

Додатково в систему інтегровано корпусний вентилятор MX-6015 5В 3pin, який забезпечує активну циркуляцію повітря через датчик для отримання більш репрезентативних показників. Вентилятор працює в імпульсному режимі, вмикаючись на 10 секунд перед кожним вимірюванням, що дозволяє оновлювати повітря в зоні сенсора та знижувати загальне енергоспоживання пристрою. Для живлення всієї системи використовується стандартний USB-C адаптер на 5 В, що забезпечує універсальність та зручність експлуатації.

Модель пристрою, що проєктується в межах кваліфікаційної роботи представлено на рисунку 2.5. Пристрій має компактні габарити та може бути розміщений у будь-якому приміщенні без порушення інтер’єру.

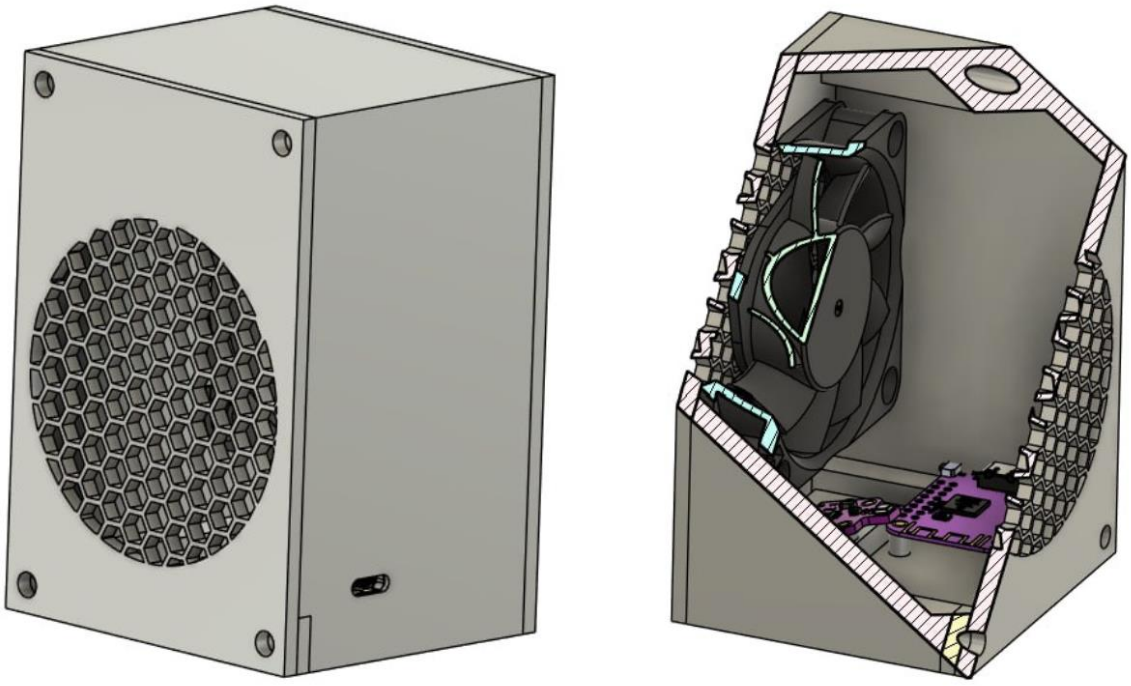

Рисунок 2.5 – Модель пристрою

Для структурування апаратної частини системи та демонстрації взаємодії компонентів розроблено блок-схему (рис. 2.6), яка відображає основні елементи пристрою та їх зв’язки.

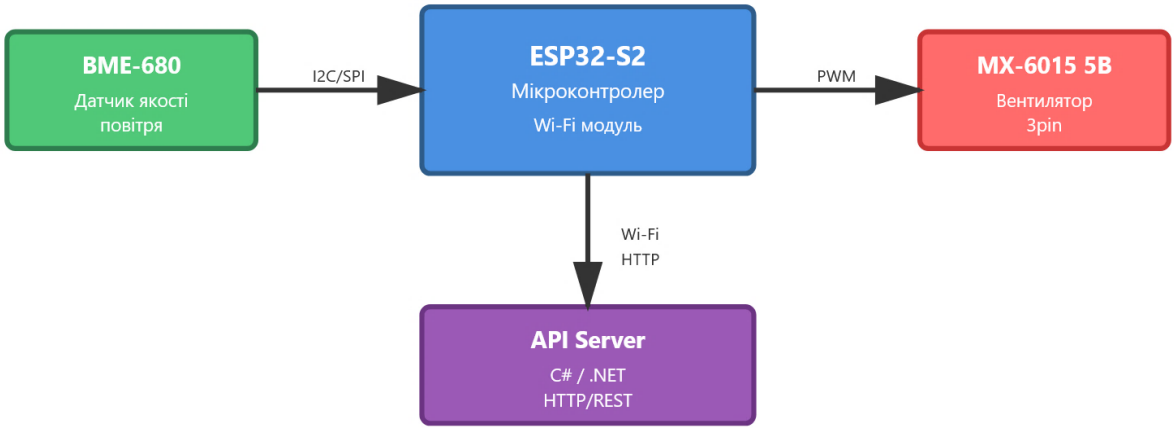

Рисунок 2.6 – Структурна схема апаратної частини системи моніторингу

Програмне забезпечення для мікроконтролера розроблено мовою C++ з використанням фреймворку Arduino, що забезпечує зручність розробки та широку підтримку бібліотек для роботи з датчиками та протоколами зв’язку. Прошивка реалізує функції зчитування даних з датчика BME-680, їх первинної обробки, керування вентилятором та передачі інформації на сервер через HTTP-протокол. C++ обрано як мову програмування через її ефективність на рівні апаратних ресурсів, можливість низькорівневого керування периферією та наявність зрілих бібліотек для ESP-32.

Серверна частина системи реалізована з використанням платформи .NET та мови програмування C#, що дозволяє створити високопродуктивний та масштабований API для прийому, обробки та збереження даних з датчиків. Вибір C# обґрунтовується наявністю потужних засобів для розробки RESTful API, зручними механізмами асинхронного програмування, які є критичними для обробки великої кількості одночасних запитів від датчиків, та наявністю зрілих бібліотек для роботи з різними типами баз даних [63]. API забезпечує валідацію

вхідних даних, їх нормалізацію та одночасний запис до реляційної та NoSQL баз даних, що дозволяє проводити паралельне порівняння їх характеристик.

Архітектура серверної частини передбачає модульну структуру з чітким розділенням відповідальності між компонентами, що відображено на відповідній схемі (рис. 2.7).

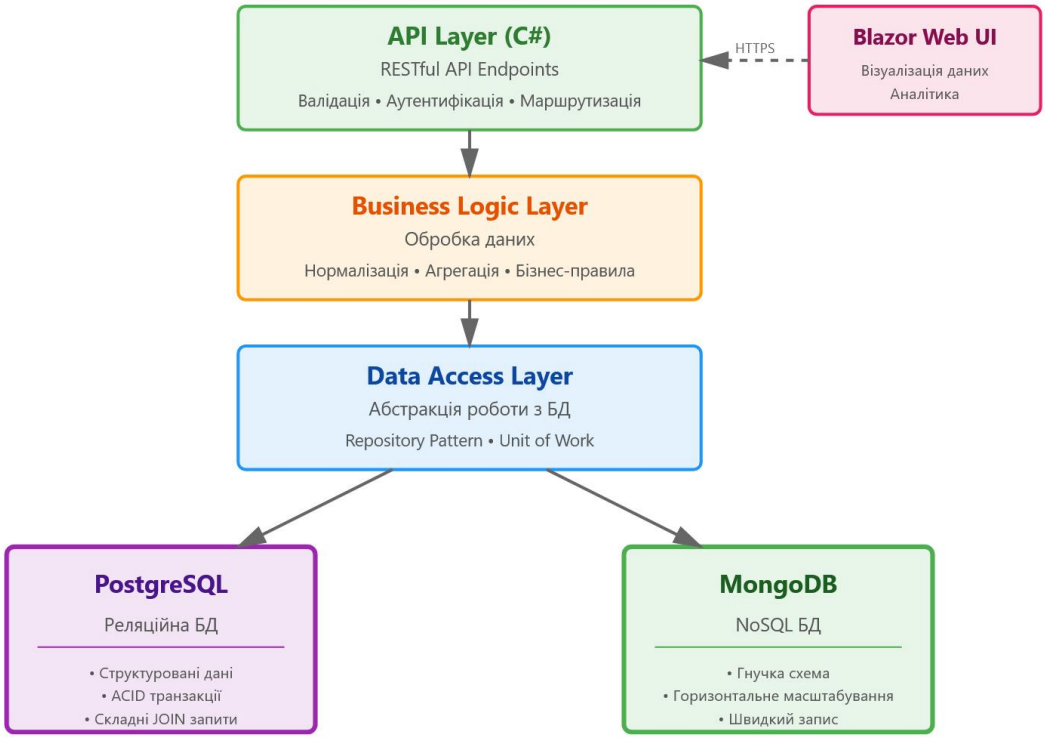

Рисунок 2.7 – Архітектура серверної частини системи

Для візуалізації даних та взаємодії з системою обрано фреймворк Blazor, який дозволяє створювати інтерактивні веб-інтерфейси з використанням C# замість JavaScript. Blazor забезпечує можливість розробки як серверних (Blazor Server), так і клієнтських (Blazor WebAssembly) додатків, що надає гнучкість у виборі моделі розгортання. Перевагами Blazor є можливість повторного використання коду між клієнтською та серверною частинами, строга типізація, що зменшує кількість помилок під час розробки, та інтеграція з екосистемою .NET [64].

Як реляційну систему керування базами даних обрано PostgreSQL – відкриту та високопродуктивну СУБД, яка підтримує складні запити, транзакції

ACID, індексування та розширені типи даних. PostgreSQL демонструє відмінну продуктивність при роботі зі структурованими даними, підтримує JSON для зберігання напівструктурованої інформації та має розвинені механізми оптимізації запитів. Для NoSQL компоненту системи вибрано MongoDB – документо-орієнтовану базу даних, яка зберігає дані у форматі BSON та забезпечує гнучкість схеми, горизонтальне масштабування та високу швидкість запису. MongoDB добре підходить для роботи з даними, структура яких може змінюватися, та для сценаріїв з високою інтенсивністю операцій запису.

Загальна архітектура системи передбачає потік даних від датчика через API до обох типів баз даних, що дозволяє проводити порівняльний аналіз в ідентичних умовах навантаження (рис. 2.8).

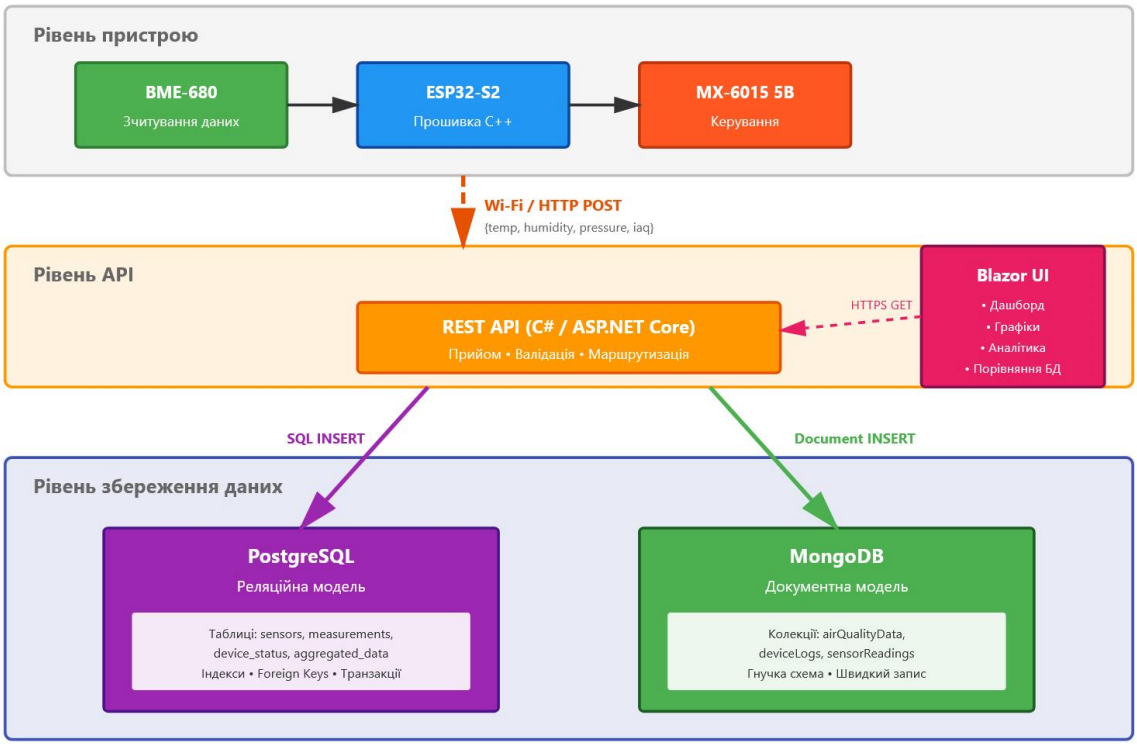

Рисунок 2.8 – Загальна архітектура системи та потік даних

Критерії оцінки ефективності баз даних визначаються на основі специфіки роботи системи моніторингу якості повітря, яка характеризується регулярним надходженням нових даних з датчиків та необхідністю швидкого доступу до історичних показників для аналізу. Першим критерієм є швидкість операцій

запису, оскільки система постійно отримує нові дані з датчиків і будь-яка затримка при записі може призвести до втрати інформації або створення черги запитів. Цей параметр вимірюється кількістю успішних операцій запису за одиницю часу при різних рівнях навантаження.

Другим важливим критерієм є швидкість виконання запитів на читання, особливо складних аналітичних запитів, які можуть включати агрегацію даних за певний період, фільтрацію за діапазонами значень та обчислення статистичних показників. Для об’єктивної оцінки тестуються запити різної складності: прості запити з вибіркою за ідентифікатором, запити з фільтрацією за часовим діапазоном, запити з агрегацією даних та складні запити з об’єднанням декількох умов.

Третім критерієм є використання ресурсів системи, зокрема процесорного часу, оперативної пам’яті та дискового простору. Ці показники критичні для оцінки економічної ефективності рішення, оскільки впливають на вартість інфраструктури та можливості масштабування системи. Вимірювання проводяться під час виконання типових операцій та при різних обсягах збережених даних.

Четвертим критерієм є масштабованість системи, тобто здатність підтримувати стабільну продуктивність при зростанні обсягу даних та кількості одночасних операцій. Для оцінки цього параметра проводяться тести з поступовим збільшенням навантаження та обсягу збережених даних, при цьому аналізується зміна часу відгуку та використання ресурсів.

П’ятим критерієм є надійність та стійкість до відмов, що включає здатність системи відновлюватися після збоїв, забезпечувати цілісність даних та підтримувати роботу при часткових відмовах компонентів. Цей аспект оцінюється через механізми реплікації, резервного копіювання та відновлення даних, які пропонують обидві системи.

Для комплексної оцінки створено методику тестування, яка передбачає проведення серії експериментів з різними параметрами навантаження та типами операцій. Методика включає підготовку тестового датасету, що містить

реалістичні дані про якість повітря за тривалий період, визначення сценаріїв використання системи та метрик для вимірювання продуктивності. Всі тести виконуються на ідентичному апаратному забезпеченні для забезпечення порівнянності результатів.

Обраний технологічний стек та визначені критерії оцінки створюють основу для проведення об’єктивного дослідження переваг та недоліків реляційних та NoSQL баз даних у контексті систем моніторингу якості повітря. Результати цього дослідження дозволять сформулювати рекомендації щодо вибору оптимального рішення для подібних систем збору та аналізу даних з датчиків.

РОЗДІЛ 3
ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

3.1 Проектування тестового середовища та методологія експерименту

Об’єктивне порівняння ефективності реляційних та NoSQL баз даних у системах моніторингу якості повітря вимагає ретельно спроектованого тестового середовища та чітко визначеної методології проведення експериментів. Створення адекватних умов для тестування є критичним фактором, який забезпечує достовірність отриманих результатів та можливість їх практичного застосування при проектуванні реальних систем збору та обробки даних з IoT-пристроїв.

Проектування тестового середовища повинно враховувати специфіку роботи системи моніторингу, яка характеризується безперервним потоком даних від розподілених датчиків, необхідністю збереження великих обсягів історичної інформації та виконанням як простих операцій запису, так і складних аналітичних запитів. Особливу увагу необхідно приділити створенню реалістичних сценаріїв навантаження, які відображають типові та пікові режими роботи системи, а також забезпеченню ізоляції компонентів для уникнення взаємного впливу при паралельному тестуванні різних баз даних.

Тестове середовище розгорнуто на локальній робочій станції з конфігурацією, що забезпечує достатню обчислювальну потужність для одночасної роботи обох баз даних під навантаженням. Апаратна платформа базується на процесорі AMD Ryzen 7 5700X з 8 ядрами та 16 потоками, що дозволяє ефективно обробляти паралельні запити до баз даних. Система оснащена 64 ГБ оперативної пам’яті DDR4, що є достатнім для розміщення в пам’яті великих обсягів даних та кешування запитів обома СУБД. Для зберігання даних використовується швидкодійний NVMe SSD Samsung 980 ємністю 1 ТБ, який забезпечує високу швидкість операцій введення-виведення та мінімальну затримку при доступі до даних. Операційною системою тестового середовища є

Windows, що дозволяє використовувати нативні інструменти розробки .NET та зручні засоби моніторингу ресурсів системи.

Апаратна частина системи збору даних побудована на базі мікроконтролера ESP32-S2, який забезпечує необхідну обчислювальну потужність та можливості бездротового зв'язку для передачі даних на сервер. Основною функцією пристрою є безперервний моніторинг параметрів якості повітря з використанням датчика BME680, який здатен вимірювати температуру, вологість, атмосферний тиск та опір газу для оцінки якості повітря. Датчик підключено до мікроконтролера через інтерфейс I2C з використанням піну GPIO8 для лінії SDA та GPIO9 для лінії SCL, що забезпечує надійну та швидку передачу даних між компонентами.

Програмне забезпечення мікроконтролера реалізує багатопоточну архітектуру обробки даних з чіткими інтервалами для різних операцій. Основний цикл програми виконує періодичне зчитування даних з датчика кожні 3 секунди, що визначається константою SENSOR_READ_INTERVAL. При кожному зчитуванні виконується послідовність операцій: ініціація вимірювання датчиком BME-680, отримання сирих значень температури, вологості, тиску та опору газу, а також обчислення додаткових параметрів, таких як точка роси та висота над рівнем моря (рис. 3.1).

```
if (millis() - lastRead >= SENSOR_READ_INTERVAL) {
    lastRead = millis();
    sensor.read();

    Serial.print("[Sensor] Temp: ");
    Serial.print(sensor.getTemperature());
    Serial.print("°C | Hum: ");
    Serial.print(sensor.getHumidity());
    Serial.print("% | Press: ");
    Serial.print(sensor.getPressure());
    Serial.println(" hPa");
}
```

Рисунок 3.1 – Зчитування та вивід сирих значень датчиком BME680

Обчислення індексу якості повітря (IAQ) здійснюється на основі комбінованого аналізу опору газу та вологості повітря. Алгоритм використовує шкалу IAQ від 0 до 500, де нижчі значення відповідають кращій якості повітря. Основний внесок у розрахунок IAQ вносить опір газу, виміряний у кілоомах, який є індикатором концентрації летких органічних сполук у повітрі. Для значень опору газу вище 150 кΩ встановлюється базовий IAQ на рівні 25, що відповідає відмінній якості повітря. При зменшенні опору газу значення IAQ поступово зростає за нелінійним законом, відображаючи погіршення якості повітря (рис. 3.2).

```
void Sensor::calculateIAQ() {
    float baseIAQ = 0;

    if (gasResistance >= 150) {
        baseIAQ = 25;
    } else if (gasResistance >= 100) {
        baseIAQ = 25 + ((150 - gasResistance) / 50 * 25);
    } else if (gasResistance >= 50) {
        baseIAQ = 50 + ((100 - gasResistance) / 50 * 50);
    } else if (gasResistance >= 25) {
        baseIAQ = 100 + ((50 - gasResistance) / 25 * 75);
    } else if (gasResistance >= 10) {
        baseIAQ = 175 + ((25 - gasResistance) / 15 * 75);
    } else {
        baseIAQ = 250 + ((10 - gasResistance) / 10 * 150);
    }

    float humAdjust = 0;
    if (humidity < 30) {
        humAdjust = (30 - humidity) * 2;
    } else if (humidity > 50) {
        humAdjust = (humidity - 50) * 2;
    }
    humAdjust = constrain(humAdjust, 0, 50);

    iaq = (int)(baseIAQ + humAdjust);
    iaq = constrain(iaq, 0, 500);
}
```

Рисунок 3.2 – Обчислення індексу якості повітря

Додатковим компонентом системи є корпусний вентилятор, який підключено до мікроконтролера через GPIO пін і керується на основі параметрів якості повітря. Система автоматичного керування вентилятором реалізує інтелектуальну логіку з підтримкою двох незалежних тригерів: температурного та IAQ-тригера. Кожен тригер може бути індивідуально увімкнений або вимкнений через веб-інтерфейс, що дозволяє гнучко налаштовувати поведінку системи під конкретні умови експлуатації. Температурний тригер активує вентилятор при перевищенні встановленого порогу температури, що за замовчуванням встановлено на рівні 28° C. IAQ-тригер спрацьовує при перевищенні порогового значення індексу якості повітря, типово встановленого на рівні 100 (рис. 3.3).

```
void Fan::update(float currentTemp, int currentIaq) {
    if (!autoMode) return;
    bool shouldBeOn = false;
    String reason = "";

    if (tempTriggerEnabled && currentTemp >= tempThreshold) {
        shouldBeOn = true;
        reason = "Temperature (" + String(currentTemp, 1) + "°C >= " +
                 String(tempThreshold, 1) + "°C)";
    }

    if (iaqTriggerEnabled && currentIaq > iaqThreshold) {
        shouldBeOn = true;
        if (reason.length() > 0) reason += " AND ";
        reason += "Poor air quality (IAQ " + String(currentIaq) +
                  " > " + String(iaqThreshold) + ")";
    }

    if (shouldBeOn && !state) {
        Serial.print("[Fan Auto] Triggering: ");
        Serial.println(reason);
        on();
    }
}
```

Рисунок 3.3 – Логіка роботи корпусного вентилятора

Для уникнення швидких перемикань стану вентилятора реалізовано механізм гістерезису, який вимагає, щоб параметри повернулися до безпечних значень з певним запасом перед вимкненням вентилятора. Температурний гістерезис становить 1° C, тобто вентилятор вимкнеться лише коли температура знизиться принаймні на градус нижче порогового значення. Для IAQ встановлено гістерезис у 10 одиниць, що запобігає частим перемиканням при коливаннях якості повітря біля порогового рівня.

Передача даних на сервер здійснюється через HTTP-протокол з використанням бібліотеки HTTPClient, яка забезпечує надійну комунікацію між ESP-32 та серверним API. Метод APIClient інкапсулює логіку формування JSON-запитів та відправки даних на сконфігурований endpoint. Унікальним ідентифікатором пристрою служить MAC-адреса мережевого адаптера ESP-32, яка автоматично зчитується при ініціалізації системи та використовується у всіх запитах до API (рис. 3.4).

```
void APIClient::begin() {
    uint8_t mac[6];
    WiFi.macAddress(mac);
    char macStr[18];
    snprintf(macStr, sizeof(macStr), "%02X:%02X:%02X:%02X:%02X:%02X",
             mac[0], mac[1], mac[2], mac[3], mac[4], mac[5]);
    deviceId = String(macStr);
}
```

Рисунок 3.4 – Підключення датчика до API

Формування HTTP-запиту включає створення JSON-документу з актуальними значеннями всіх параметрів моніторингу. Структура даних містить ідентифікатор пристрою, температуру з точністю до одного десяткового знаку, вологість, атмосферний тиск, обчислений індекс якості повітря, опір газу та поточний стан вентилятора. Перед відправкою всі числові значення округлюються до одного десяткового знаку для зменшення обсягу переданих даних та уніфікації формату (рис. 3.5).

```
bool APIClient::sendReading(float temperature, float humidity, float pressure,
                            int iaq, float gas, bool fan) {
    StaticJsonDocument<256> doc;
    doc["deviceId"] = deviceId;
    doc["temperature"] = round(temperature * 10) / 10.0;
    doc["humidity"] = round(humidity * 10) / 10.0;
    doc["pressure"] = round(pressure * 10) / 10.0;
    doc["iaq"] = iaq;
    doc["gas"] = round(gas * 10) / 10.0;
    doc["fan"] = fan;

    String jsonString;
    serializeJson(doc, jsonString);

    int httpCode = http.POST(jsonString);
}
```

Рисунок 3.5 – Логіка формування HTTP-запиту

Система підтримує гнучке налаштування інтервалу відправки даних, який може варіюватися від 1 секунди до 24 годин залежно від вимог до частоти моніторингу та доступності мережевого каналу. Конфігурація зберігається в енергонезалежній пам’яті EEPROM мікроконтролера, що дозволяє зберігати налаштування після перезавантаження пристрою. Основний цикл програми перевіряє час від останньої відправки даних та ініціює нову передачу при досягненні встановленого інтервалу, якщо функція відправки увімкнена та існує активне з’єднання з мережею Wi-Fi.

Управління Wi-Fi-підключенням реалізовано через модуль WiFiManager, який підтримує два режими роботи: клієнтський режим для підключення до існуючої мережі та режим точки доступу для початкового налаштування. При першому запуску або відсутності збережених облікових даних мікроконтролер автоматично створює точку доступу з попередньо визначеним SSID, що дозволяє користувачу підключитися до пристрою та налаштувати параметри підключення до домашньої або офісної Wi-Fi-мережі (рис. 3.6)

```
void WiFiManager::begin() {
    loadCredentials();

    if (ssid.length() > 0) {
        connect(ssid, password);
    } else {
        Serial.println("[WiFi] No saved credentials found");
    }

    if (!isConnected()) {
        Serial.println("[WiFi] Not connected, starting AP mode");
        startAP();
    }
}
```

Рисунок 3.6 – Управління WiFi-підключенням

Для зручності доступу до веб-інтерфейсу пристрою реалізовано підтримку mDNS (multicast DNS), що дозволяє звертатися до ESP32 за доменним ім'ям замість IP-адреси. Після успішної ініціалізації mDNS-респондера пристрій стає доступним за адресою airlit.local у локальній мережі, що значно спрощує взаємодію з системою моніторингу без необхідності запам'ятовувати або шукати IP-адресу пристрою.

Веб-сервер на основі бібліотеки ESPAsyncWebServer надає RESTful API для отримання поточних показників датчиків та управління налаштуваннями системи. Основний endpoint /data повертає JSON-об'єкт з повним набором даних, включаючи параметри якості повітря, стан вентилятора, конфігурацію тригерів, інформацію про WiFi-підключення та системні дані мікроконтролера. Ця інформація використовується веб-інтерфейсом для відображення актуальних значень у реальному часі без необхідності перезавантаження сторінки (рис. 3.7).

Серверна частина системи реалізована з використанням платформи .NET 8 та мови програмування C#, що забезпечує високу продуктивність та широкі можливості для інтеграції з різними типами баз даних. Архітектура API побудована на принципах RESTful сервісів з використанням ASP.NET Core, що дозволяє створити масштабований та ефективний інтерфейс для взаємодії з

датчиками та клієнтськими додатками. Ключовою особливістю реалізації є паралельний запис даних одночасно до PostgreSQL та MongoDB, що дозволяє проводити порівняльний аналіз в абсолютно ідентичних умовах навантаження.

```
server.on("/data", HTTP_GET, [this](AsyncWebServerRequest *request) {
    StaticJsonDocument<512> doc;
    doc["temp"] = sensor->getTemperature();
    doc["hum"] = sensor->getHumidity();
    doc["press"] = sensor->getPressure();
    doc["gas"] = sensor->getGasResistance();
    doc["iaq"] = sensor->getIAQ();
    doc["fan"] = fan->getState();
    doc["auto"] = fan->isAutoMode();
    doc["wifi_connected"] = wifiMgr->isConnected();
    doc["uptime"] = millis() / 1000;
    doc["free_heap"] = ESP.getFreeHeap();

    String response;
    serializeJson(doc, response);
    request->send(200, "application/json", response);
});
```

Рисунок 3.7 – API девайсу для веб-стоірнки

Прийом даних від датчиків здійснюється через спеціалізований контролер AirQualityController, який забезпечує валідацію вхідних даних та їх передачу до сервісного шару для збереження. Контролер реалізує endpoint для POST-запитів від ESP-32, приймаючи дані у форматі JSON та перетворюючи їх у внутрішню модель системи (рис. 3.8).

```
[HttpPost( template: "reading")]
public async Task<IActionResult> PostReading([FromBody] AirQualityReadingDto reading)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    await _service.SaveReadingAsync(reading);

    return Ok();
}
```

Рисунок 3.8 – Endpoint для прийому даних від датчика

Сервісний шар представлений класом AirQualityService, який інкапсулює логіку збереження даних та автоматичного вимірювання продуктивності операцій. Сервіс використовує паттерн Dependency Injection для отримання доступу до репозиторіїв обох баз даних та сервісу відстеження продуктивності. При кожному збереженні даних автоматично вимірюється час виконання операції INSERT для PostgreSQL та MongoDB, що дозволяє накопичувати статистику продуктивності в реальному часі (рис. 3.9).

```
public async Task<bool> SaveReadingAsync(AirQualityReadingDto dto)
{
    var data = MapToModel(dto);
    await _performanceTracker.MeasureAndSaveInsertAsync(data);
    return true;
}
```

Рисунок 3.9 – Метод збереження даних з датчика в базу даних

Ключовим компонентом системи вимірювання продуктивності є PerformanceTrackingService, який використовує клас Stopwatch для точного вимірювання часу виконання операцій з мікросекундною точністю. Сервіс автоматично зберігає результати вимірювань у спеціалізовану таблицю performance_metrics в PostgreSQL, що дозволяє накопичувати історичні дані про продуктивність системи (рис. 3.10).

```
public async Task MeasureAndSaveInsertAsync(AirQualityData data)
{
    // PostgreSQL
    var pgSw = Stopwatch.StartNew();
    await _postgresRepo.InsertAsync(data);
    pgSw.Stop();

    await SaveRecordAsync( databaseType: "PostgreSQL",  operation: "INSERT", pgSw.Elapsed.TotalMilliseconds);

    // MongoDB
    var mongoSw = Stopwatch.StartNew();
    await _mongoRepo.InsertAsync(data);
    mongoSw.Stop();

    await SaveRecordAsync( databaseType: "MongoDB",  operation: "INSERT", mongoSw.Elapsed.TotalMilliseconds);
}
```

Рисунок 3.10 – Логіка вимірювання часу виконання INSERT запитів

Для проведення навантажувальних тестів розроблено спеціалізований сервіс LoadTestService, який дозволяє генерувати велику кількість синтетичних даних та виконувати їх паралельну вставку в обидві бази даних. Сервіс підтримує конфігурування кількості записів та потоків для емуляції різних рівнів навантаження на систему. Генерація тестових даних здійснюється з використанням класу Random для створення реалістичних значень температури, вологості, тиску та інших параметрів якості повітря (рис. 3.11).

```
var testData = Enumerable.Range(0, records)
    .Select(i => new AirQualityData
    {
        DeviceId = "ESP32-LOADTEST",
        Timestamp = DateTime.UtcNow.AddSeconds(-i),
        Temperature = (float)(20 + random.NextDouble() * 10),
        Humidity = (float)(40 + random.NextDouble() * 30),
        Pressure = (float)(1000 + random.NextDouble() * 50),
        AirQualityIndex = (float)(random.NextDouble() * 200),
        GasResistance = (float)(100000 + random.NextDouble() * 50000),
        FanStatus = random.Next(2) == 1
    })
    .ToList();
```

Рисунок 3.11 – Логіка генерації тестових даних

Паралельна вставка даних реалізована через розподіл загального масиву записів на окремі фрагменти, кожен з яких обробляється в окремому асинхронному потоці. Це дозволяє емулювати реальні умови роботи системи, коли дані надходять одночасно від декількох датчиків (рис. 3.12).

Крім тестування операцій запису, система вимірює продуктивність операцій читання даних, включаючи вибірку за часовим діапазоном. Це дозволяє оцінити ефективність індексів та механізмів кешування обох баз даних при виконанні типових аналітичних запитів (рис. 3.13).

Для забезпечення уніфікованого інтерфейсу доступу до різних типів баз даних використовується паттерн Repository з загальним інтерфейсом

IDataRepository, який визначає стандартний набір операцій для роботи з даними моніторингу якості повітря (рис. 3.14).

```
private async Task InsertDataParallel(IDataRepository repo, List<AirQualityData> data, int threads)
{
    if (threads <= 0) threads = 1;

    var chunkSize :int = Math.Max(1, data.Count / threads);
    var chunks :List<List<...>> = data // List<AirQualityData>
        .Select((item :AirQualityData , index) => new { item, index }) // IEnumerable<{item,index}>
        .GroupBy(x :{item,index} => x.index / chunkSize) // IEnumerable<IGrouping<...>>
        .Select(g :IGrouping<int,{item,index}> => g.Select(x :{item,index} => x.item).ToList()) // IEnumerable<List<...>>
        .ToList();

    var tasks :IEnumerable<Task> = chunks.Select(async chunk :List<AirQualityData> =>
    {
        foreach (var record :AirQualityData in chunk)
        {
            await repo.InsertAsync(record);
        }
    });

    await Task.WhenAll(tasks);
}
```

Рисунок 3.12 – Логіка паралельної вставки в кілька потоків

```
private async Task<double> MeasureSelectPerformanceAsync(
    IDataRepository repo, string deviceId, DateTime from, DateTime to)
{
    var stopwatch = Stopwatch.StartNew();
    await repo.GetByDeviceIdAsync(deviceId, from, to);
    stopwatch.Stop();
    return stopwatch.Elapsed.TotalMilliseconds;
}
```

Рисунок 3.13 – Логіка вимірювання часу виконання SELECT запитів

```
public interface IDataRepository
{
    Task<bool> InsertAsync(AirQualityData? data);
    Task<IEnumerable<AirQualityData>> GetByDeviceIdAsync(
        string deviceId, DateTime from, DateTime to);
    Task<IEnumerable<AirQualityData>> GetRecentAsync(int count);
    Task<AirQualityData?> GetByIdAsync(int id);
}
```

Рисунок 3.14 – Набір операцій для роботи з даними моніторингу якості повітря

API надає розширений набір endpoints для отримання даних у різних форматах та часових діапазонах. DataController реалізує методи для вибірки останніх показників, даних за годину, за добу та за довільний період. Важливою особливістю є можливість вибору джерела даних через параметр запиту source, що дозволяє клієнтським додаткам порівнювати результати з обох баз даних (рис. 3.15).

```
[HttpGet( template: "latest")]
public async Task<ActionResult<AirQualityData>> GetLatest([FromQuery] string source = "postgres")
{
    try
    {
        IDataRepository repo = source.Equals("mongo", StringComparison.CurrentCultureIgnoreCase) ? _mongoRepo : _postgresRepo;
        var data :IEnumerable<AirQualityData> = await repo.GetRecentAsync( count: 1);
        var latest :AirQualityData? = data.FirstOrDefault();

        if (latest == null)
        {
            return  NotFound(new { message = "No data available" });
        }

        return  Ok(latest);
    }
    catch (Exception ex)
    {
        return  StatusCode(500, new { message = ex.Message });
    }
}
```

Рисунок 3.15 – Endpoint для отримання останніх даних

Для зменшення навантаження на клієнтську частину при відображенні великих масивів даних реалізовано механізми агрегації та фільтрації результатів. При запиті даних за останню годину система автоматично відбирає кожен п’ятий запис, що зменшує обсяг переданих даних без значної втрати інформативності візуалізації. Для добового періоду застосовується агрегація даних по годинах з обчисленням середніх значень параметрів (рис. 3.16).

Окремий контролер PerformanceController надає доступ до метрик продуктивності та управління навантажувальними тестами. Контролер дозволяє отримувати порівняльну статистику між базами даних, детальні метрики для кожної бази окремо та запускати автоматизовані тести з різними параметрами навантаження. Endpoint для порівняння обчислює середні значення часу

виконання операцій та визначає переможця на основі накопичених статистичних даних (рис. 3.17).

```
var aggregated = data
    .GroupBy(d => new DateTime(d.Timestamp.Year, d.Timestamp.Month,
        d.Timestamp.Day, d.Timestamp.Hour, 0, 0))
    .Select(g => new AirQualityData
    {
        Timestamp = g.Key,
        DeviceId = g.First().DeviceId,
        Temperature = g.Average(x => x.Temperature),
        Humidity = g.Average(x => x.Humidity),
        Pressure = g.Average(x => x.Pressure),
        AirQualityIndex = g.Average(x => x.AirQualityIndex),
        GasResistance = g.Average(x => x.GasResistance),
        FanStatus = g.Last().FanStatus
    })
    .OrderBy(d => d.Timestamp)
    .ToList();
```

Рисунок 3.16 – Логіка агрегації даних для клієнтської частини

```
[HttpGet("comparison")]
public async Task<ActionResult<ComparisonDto>> GetComparison()
{
    var comparison = await _performanceService.GetComparisonAsync();
    return Ok(comparison);
}
```

Рисунок 3.17 – Endpoint для отримання інформації про порівнння двох баз даних

Реалізація роботи з PostgreSQL базується на Entity Framework Core, що забезпечує об’єктно-реляційне відображення та зручний LINQ-інтерфейс для виконання запитів. Контекст бази даних ApplicationDbContext конфігурує таблицю air_quality_readings з відповідними типами даних та індексами для оптимізації запитів за часовою міткою та ідентифікатором пристрою (рис. 3.18).

```
modelBuilder.Entity<AirQualityData>(entity =>
{
    entity.ToTable("air_quality_readings");
    entity.HasKey(e => e.Id);

    entity.HasIndex(e => e.Timestamp)
        .HasDatabaseName("idx_timestamp");

    entity.HasIndex(e => e.DeviceId)
        .HasDatabaseName("idx_device_id");

    entity.HasIndex(e => new { e.DeviceId, e.Timestamp })
        .HasDatabaseName("idx_device_timestamp");
});
```

Рисунок 3.18 – Конфігурація таблиця для PostgreSQL через Entity Framework Core

Для роботи з MongoDB використовується офіційний драйвер MongoDB.Driver, який надає нативний API для виконання операцій з документами. Контекст MongoDbContext ініціалізує підключення до бази даних та надає доступ до колекції air_quality_readings. Документна модель AirQualityDataDocument відображає структуру даних у форматі BSON з автоматичним генерованим ідентифікатором ObjectId (рис. 3.19).

```
public class MongoDbContext
{
    private readonly IMongoDatabase _database;

    public MongoDbContext(IConfiguration configuration)
    {
        var connectionString = configuration.GetConnectionString("MongoDB");
        var mongoUrl = MongoUrl.Create(connectionString);
        var client = new MongoClient(mongoUrl);
        _database = client.GetDatabase(mongoUrl.DatabaseName);
    }

    public IMongoCollection<AirQualityDataDocument> AirQualityReadings =>
        _database.GetCollection<AirQualityDataDocument>("air_quality_readings");
}
```

Рисунок 3.19 – Конгфігурація доступу до MongoDB

Для збереження результатів тестування продуктивності використовується окрема таблиця performance_metrics, яка зберігає часові мітки операцій, тип бази даних, тип операції та виміряний час виконання. Це дозволяє накопичувати історичні дані про продуктивність системи та проводити ретроспективний аналіз змін ефективності при зростанні обсягу збережених даних. Контекст MonitoringDbContext забезпечує доступ до цієї таблиці через Entity Framework Core, що спрощує виконання аналітичних запитів та обчислення статистичних показників.

Користувацький інтерфейс системи реалізовано з використанням фреймворку Blazor Server, який дозволяє створювати інтерактивні веб-додатки з використанням C# замість JavaScript та забезпечує двосторонній зв'язок між клієнтом та сервером через SignalR. Архітектура інтерфейсу побудована на компонентній моделі, де кожна сторінка та візуальний елемент представлені окремим компонентом з власною логікою та станом. Головна сторінка системи надає доступ до поточних показників якості повітря та їх візуалізації у вигляді графіків за різні часові періоди.

Основний компонент Index.razor реалізує дашборд моніторингу з можливістю вибору конкретного пристрою для відображення даних. Інтерфейс підтримує відображення даних від фізичного датчика з унікальною MAC-адресою та від віртуального пристрою ESP-32-LOADTEST, який використовується для генерації тестових даних під час навантажувальних випробувань. Вибір пристрою здійснюється через випадаючий список з автоматичним оновленням даних при зміні вибору (рис. 3.20).

```
<div class="input-group">
    <select class="form-select" @bind="selectedDevice" @bind:after="ApplyDeviceFilter" disabled="@loading">
        <option value="">Виберіть пристрій...</option>
        <option value="90:E5:B1:8E:6F:F6">90:E5:B1:8E:6F:F6</option>
        <option value="ESP32-LOADTEST">ESP32-LOADTEST</option>
    </select>
</div>
```

Рисунок 3.20 – Логіка вибору девайсу на клієнтській частині

Поточні показники відображаються у вигляді чотирьох інформаційних карток з кольоровим кодуванням для швидкої візуальної оцінки стану параметрів. Картка температури використовує синій колір, вологості – блакитний, тиску – сірий, а картка якості повітря динамічно змінює свій колір залежно від значення індексу IAQ за шкалою від зеленого (відмінна якість) до чорного (дуже погана якість). Кожна картка відображає актуальне значення параметру з точністю до одного десяткового знаку та часову мітку останнього оновлення.

Для візуалізації історичних даних розроблено спеціалізовані компоненти графіків, які використовують SVG для побудови інтерактивних діаграм безпосередньо в браузері без залежності від зовнішніх бібліотек. Компонент TemperatureChart відображає зміну температури за останню годину у вигляді лінійного графіка з автоматичним масштабуванням осей відповідно до діапазону значень. Алгоритм нормалізації координат забезпечує оптимальне використання області відображення незалежно від абсолютних значень температури (рис. 3.21).

```
private List<(double X, double Y)> GetTemperatureCoordinates()
{
    var minTemp = Data.Min(d => d.Temperature);
    var maxTemp = Data.Max(d => d.Temperature);
    var range = maxTemp - minTemp;
    if (range == 0) range = 1;

    var result = new List<(double X, double Y)>();

    for (int i = 0; i < Data.Count; i++)
    {
        var x = 50 + (i * (700.0 / (Data.Count - 1)));
        var normalized = (Data[i].Temperature - minTemp) / range;
        var y = 250 - (normalized * 200);
        result.Add((x, y));
    }

    return result;
}
```

Рисунок 3.21 – Алгоритм нормалізаціїї координат

Компонент AirQualityChart реалізує візуалізацію індексу якості повітря з інтегрованими зонами якості, які відображаються як кольорові смуги на фоні графіка. Шість зон якості (good, average, little bad, bad, worse, very bad) представлені відповідними кольорами згідно зі стандартною шкалою IAQ від 0 до 500. Кожна точка даних на графіку автоматично отримує колір відповідно до свого значення IAQ, що дозволяє швидко ідентифікувати періоди погіршення якості повітря (рис. 3.22).

```
private string GetPointColor(float iaq)
{
    return iaq switch
    {
        <= 50 => "#00ff00",      // good - green
        <= 100 => "#ffff00",     // average - yellow
        <= 150 => "#ff8c00",     // little bad - orange
        <= 200 => "#ff0000",     // bad - red
        <= 300 => "#800080",     // worse - purple
        _ => "#000000"           // very bad - black
    };
}
```

Рисунок 3.22 – Отримання кольору точки зігдно з значенням IAQ

Окрема сторінка Comparison.razor присвячена порівняльному аналізу продуктивності PostgreSQL та MongoDB. Інтерфейс відображає загальну статистику з визначенням переможця за середнім часом виконання операцій, коефіцієнтом прискорення та абсолютною різницею в мілісекундах. Детальні метрики для кожної бази даних представлені у вигляді таблиць з інформацією про середній, мінімальний та максимальний час виконання операцій INSERT та SELECT, загальну кількість виконаних операцій та поточне використання пам’яті (рис. 3.23).

```
<div class="row text-center">
    <div class="col-md-4">
        <h3 class="text-success">@comparison.Winner</h3>
        <p class="text-muted">Переможець</p>
    </div>
    <div class="col-md-4">
        <h3 class="text-info">@comparison.SpeedupFactor.ToString( format: "F2")x</h3>
        <p class="text-muted">Прискорення</p>
    </div>
    <div class="col-md-4">
        <h3 class="text-warning">@comparison.DifferenceMs.ToString( format: "F2") ms</h3>
        <p class="text-muted">Різниця</p>
    </div>
</div>
```

Рисунок 3.23 – Логіка відображення метрик для кожної бази даних

Компонент ComparisonChart візуалізує порівняння продуктивності у вигляді стовпчастої діаграми з окремими стовпцями для операцій запису та читання кожної бази даних. Висота стовпців автоматично масштабується відносно максимального значення серед усіх операцій, що дозволяє наочно порівняти швидкість виконання різних типів операцій. Стовпці PostgreSQL відображаються синім кольором, MongoDB – зеленим, що забезпечує візуальну консистентність з іншими елементами інтерфейсу (рис. 3.24).

```
private double GetBarHeight(double value, double maxHeight)
{
    var maxValue :double = Math.Max(
        Math.Max(PostgresData?.AverageWriteTime ?? 0, MongoData?.AverageWriteTime ?? 0),
        Math.Max(PostgresData?.AverageReadTime ?? 0, MongoData?.AverageReadTime ?? 0)
    );

    if (maxValue == 0) return 0;

    return (value / maxValue) * maxHeight;
}
```

Рисунок 3.24 – Логіка заповнення компоненту ComparisonChart

Інтерфейс тестування навантаження дозволяє інтерактивно запускати контрольовані тести з налаштуванням кількості записів та одночасних потоків. Форма тестування включає числові поля вводу з обмеженнями діапазону значень

та кнопку запуску з індикатором виконання. При активному тесті кнопка відображає анімований спінер та змінює текст на «Виконується...», що забезпечує візуальний зворотний зв’язок про стан операції (рис. 3.25).

```
<button class="btn btn-primary w-100" @onclick="RunLoadTest" disabled="@loadTestRunning">
    @if (loadTestRunning)
    {
        <span class="spinner-border spinner-border-sm"></span> <p class="m-0">Виконується...</p>
    }
    else
    {
        <i class="fas fa-play"></i> <p class="m-0">Запустити тест</p>
    }
</button>
```

Рисунок 3.25 – Частина інтерфейсу тестування

Результати навантажувальних тестів відображаються в табличному форматі з історією останніх виконаних операцій, включаючи часову мітку, тип бази даних, назву операції та виміряний час виконання. Таблиця автоматично сортується за часом у зворотному порядку, показуючи найновіші результати на початку списку. Після завершення тесту система відображає повідомлення про успішне збереження даних та автоматично оновлює всю статистику порівняння.

Система автоматичного оновлення даних реалізована через PeriodicTimer, який запускає асинхронний цикл оновлення з інтервалом п’ять секунд для основного дашборду та десять секунд для сторінки порівняння. Цикл оновлення використовує InvokeAsync для забезпечення потокобезпечного виклику StateHasChanged, що викликає перерисовку компонента з новими даними без повного перезавантаження сторінки (рис. 3.26).

Компонент PerformanceMetricsDisplay надає детальний аналіз метрик продуктивності у вигляді інтерактивних прогрес-барів та таблиці останніх вимірювань. Прогрес-бари візуально представляють відносну швидкість баз даних через заповнення пропорційно до середнього часу виконання операцій. Таблиця метрик відображає точні значення часу виконання для PostgreSQL та MongoDB з обчисленням абсолютної різниці, яка представлена у вигляді

кольорового badge з синім кольором для переваги PostgreSQL та зеленим для MongoDB.

```
private async Task RefreshLoopAsync()
{
    try
    {
        while (await refreshTimer!.WaitForNextTickAsync() && !_disposed)
        {
            await InvokeAsync(async () =>
            {
                if (_disposed) return;
                await LoadData();
                StateHasChanged();
            });
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine($"RefreshLoopAsync stopped: {ex.Message}");
    }
}
```

Рисунок 3.26 – Логіка періодичного оновлення сторінки

Спроектоване тестове середовище та реалізований користувацький інтерфейс створюють комплексну платформу для об’єктивного порівняння ефективності PostgreSQL та MongoDB у контексті систем моніторингу якості повітря, забезпечуючи як автоматизоване збирання метрик продуктивності в реальному часі, так і зручні інструменти для проведення контрольованих навантажувальних тестів та візуального аналізу отриманих результатів.

3.2 Експериментальне дослідження характеристик SQL та NoSQL баз даних

Експериментальне дослідження характеристик PostgreSQL та MongoDB у контексті системи моніторингу якості повітря проводилося з метою отримання об’єктивних показників продуктивності при виконанні типових операцій

IoT-додатків. Методологія передбачала навантажувальні тести з варіюванням кількості записів та паралельних потоків для моделювання різних сценаріїв – від штатного режиму до екстремальних умов з одночасною роботою десятків пристроїв.

Тестові дані генерувалися програмно з використанням алгоритму псевдовипадкових чисел для створення реалістичних значень параметрів якості повітря. Кожен запис містив сім атрибутів: унікальний ідентифікатор пристрою, часову мітку з мілісекундною точністю, температуру в діапазоні від 20 до 30 градусів Цельсія, відносну вологість від 40 до 70 відсотків, атмосферний тиск від 1000 до 1050 гектопаскалів, індекс якості повітря від 0 до 200, опір газу від 100000 до 150000 Ом та булеве значення стану вентилятора. Структура даних повністю відповідала форматові реальних вимірювань, що надходять від фізичного датчика BME680, що забезпечувало адекватність тестового навантаження реальним умовам експлуатації системи.

Конфігурація баз даних для проведення експериментів базувалася на стандартних налаштуваннях з мінімальними модифікаціями для оптимізації роботи з часовими рядами даних. PostgreSQL використовувався у версії з підтримкою JSONB для можливості гнучкого зберігання даних, хоча основна структура залишалася строго реляційною з визначеною схемою та первинним ключем. Для оптимізації швидкості запитів було створено три індекси: на поле часової мітки для швидкої вибірки за часовим діапазоном, на ідентифікатор пристрою для фільтрації даних конкретного датчика, та композитний індекс на комбінацію обох полів для оптимізації найбільш частих запитів з одночасною фільтрацією за пристроєм та часом. Дослідження показують, що реляційні бази даних відзначаються своєю здатністю зберігати та організовувати дані у табличних структурах, наголошуючи на консистентності та цілісності даних, використовуючи стандартизовану мову запитів SQL.

MongoDB налаштовувався з використанням стандартного механізму зберігання WiredTiger, який забезпечує компресію даних та ефективне управління пам’яттю. Рівень гарантії запису (write concern) був встановлений на

значення «acknowledged», що означає підтвердження успішного запису даних на диск перед поверненням відповіді клієнту. Така конфігурація забезпечує баланс між швидкістю запису та надійністю збереження даних, що є критичним для систем моніторингу, де втрата даних неприпустима, але й висока латентність запису може призвести до накопичення черги необроблених повідомлень від датчиків. NoSQL бази даних, на відміну від реляційних, пропонують високу продуктивність для операцій читання/запису, особливо в розподілених середовищах, та є швидшими при роботі з великими обсягами даних та простими запитами.

Перший етап експериментального дослідження включав тестування продуктивності операцій INSERT при різних обсягах даних з використанням одного потоку виконання. Такий сценарій моделює послідовний запис даних, коли кожна операція вставки виконується лише після завершення попередньої, що відповідає роботі системи з єдиним датчиком без паралельної обробки. Результати цих тестів представлено у таблиці 3.1, яка демонструє чітку тенденцію до зростання переваги MongoDB зі збільшенням обсягу даних.

Таблиця 3.1 – Продуктивність операцій INSERT при послідовному виконанні (1 потік)

Кількість записів	PostgreSQL, мс	MongoDB, мс	Співвідношення	Перевага
100	163,88	105,61	1,55:1	MongoDB
500	526,55	154,16	3,42:1	MongoDB
1000	746,04	238,32	3,13:1	MongoDB
5000	10145,18	1074,24	9,44:1	MongoDB
10000	35837,64	2125,37	16,86:1	MongoDB

При вставці 100 записів PostgreSQL продемонстрував час виконання 163,88 мілісекунди, в той час як MongoDB виконав ту саму операцію за 105,61 мілісекунди, що становить приблизно 1,55 разів швидше. При збільшенні обсягу до 500 записів різниця стала більш вираженою: PostgreSQL витратив 526,55 мілісекунди проти 154,16 мілісекунди у MongoDB, що демонструє 3,42-кратну перевагу документо-орієнтованої бази даних. Найбільш драматична різниця у продуктивності спостерігалася при великих обсягах послідовних

операцій запису. Для вставки 5000 записів PostgreSQL знадобилося 10145,18 мілісекунди (приблизно 10,15 секунди), в той час як MongoDB впорався за 1074,24 мілісекунди (близько 1,07 секунди), що демонструє дев'ятикратну перевагу NoSQL рішення.

Аналіз причин такої значної різниці у продуктивності вимагає розгляду внутрішніх механізмів роботи кожної бази даних. PostgreSQL як реляційна СУБД виконує ряд додаткових операцій при кожній вставці: перевірку цілісності даних згідно визначеної схеми, оновлення всіх індексів, підтримку журналу транзакцій (WAL – Write-Ahead Logging) для забезпечення ACID властивостей, та керування блокуваннями для забезпечення ізоляції транзакцій. Реляційні бази даних сильно дотримуються властивостей ACID, що робить їх надійними для критичних систем, де транзакції повинні оброблятися достовірно. Кожна операція запису в PostgreSQL вимагає запису у WAL журнал перед фактичним внесенням змін до даних, що гарантує можливість відновлення після збою, але додає накладні витрати на кожну транзакцію.

MongoDB, з іншого боку, використовує більш спрощену модель запису, де документ вставляється безпосередньо у колекцію з мінімальною валідацією структури, що значно зменшує накладні витрати на кожну операцію. NoSQL бази даних оптимізовані для сценаріїв використання, таких як застосунки реального часу, де критично важливими є швидкі оновлення та гнучка схема даних. LSM-tree (Log-Structured Merge tree) архітектура WiredTiger storage engine використовує послідовні операції запису на диск замість випадкових, що є значно швидшим на рівні апаратного забезпечення. При запису даних MongoDB спочатку записує їх у пам'ять та журнал, а потім асинхронно зливає на диск великими блоками, що мінімізує кількість дискових операцій введення-виведення.

Дослідження ефекту паралелізації операцій запису виявило нелінійну залежність продуктивності від кількості одночасних потоків. Результати тестування представлено у таблиці 3.2, яка демонструє поведінку обох баз даних при різних рівнях паралелізму.

Таблиця 3.2 – Вплив кількості потоків на продуктивність операцій INSERT (100 записів)

Кількість потоків	PostgreSQL (мс)	MongoDB (мс)	Прискорення PostgreSQL	Прискорення MongoDB
1	163,88	105,61	1,00x	1,00x
5	22,19	16,70	7,39x	6,32x
10	21,84	9,04	7,50x	11,68x
20	10,14	5,59	16,16x	18,89x
50	20,14	5,96	8,14x	17,72x

При вставці 100 записів з використанням 5 потоків PostgreSQL показав час 22,19 мілісекунди, що майже у 7,4 рази швидше порівняно з однопотоковим виконанням, демонструючи ефективне використання багатоядерної архітектури процесора AMD Ryzen 7 5700X. MongoDB у цьому тесті витратив 16,70 мілісекунди, що також є значним покращенням порівняно з послідовним виконанням. Подальше збільшення кількості потоків до 10 призвело до подальшого зменшення часу виконання для обох баз даних: PostgreSQL – 21,84 мілісекунди, MongoDB – 9,04 мілісекунди. Однак при збільшенні кількості потоків до 20 та 50 спостерігалася цікава тенденція: для PostgreSQL час спочатку зменшився до 10,14 мілісекунди при 20 потоках, але потім зріс до 20,14 мілісекунд при 50 потоках. MongoDB демонстрував більш стабільну поведінку, досягаючи мінімуму 5,59 мілісекунди при 20 потоках і зберігаючи близьке значення 5,96 мілісекунди при 50 потоках.

Така поведінка пояснюється конкуренцією за ресурси при надмірному паралелізмі. Дослідження показують, що NoSQL бази даних можуть горизонтально масштабуватися через декілька вузлів, оскільки вони використовують log structured merge-trees (LSM), які є append-only та використовують послідовні IO операції запису, які є значно швидшими за випадкові IO операції запису, що використовуються реляційними базами даних з B-tree структурами. При кількості потоків, що значно перевищує кількість фізичних ядер процесора (8 ядер, 16 потоків), операційна система витрачає додаткові ресурси на контекстне перемикання між потоками, що зменшує загальну продуктивність. MongoDB, завдяки своїй архітектурі, краще

справляється з високим рівнем конкурентності, оскільки її движок зберігання оптимізований саме для паралельних операцій запису.

При середніх та великих обсягах даних з використанням паралельних потоків спостерігалася схожа картина, але з більш вираженими ефектами масштабування. Таблиця 3.3 демонструє результати для обсягу 5000 записів при різній кількості потоків.

Таблиця 3.3 – Продуктивність операцій INSERT для 5000 записів при різній паралелізації

Потоки	PostgreSQL, мс	MongoDB, мс	Співвідношення	Перевага
1	10145,18	1074,24	9,44:1	MongoDB
5	196,61	258,71	0,76:1	PostgreSQL
10	379,47	162,87	2,33:1	MongoDB
20	219,86	135,31	1,62:1	MongoDB
50	535,50	144,10	3,72:1	MongoDB

Цікавим спостереженням є те, що при 5 потоках PostgreSQL несподівано показав кращий результат (196,61 мс) порівняно з MongoDB (258,71 мс). Це пояснюється специфікою розподілу роботи: при діленні 5000 записів на 5 потоків кожен потік обробляє 1000 записів, що є достатньо великим обсягом для ефективної роботи механізму пакетних транзакцій PostgreSQL. У цьому сценарії накладні витрати на координацію потоків у MongoDB перевищують виграш від паралелізації. Однак при збільшенні кількості потоків до 10 та більше MongoDB повертає собі перевагу, демонструючи стабільну продуктивність 135-163 мілісекунди незалежно від кількості потоків.

Операції читання даних (SELECT) демонстрували істотно відмінну картину продуктивності порівняно з операціями запису. Результати представлено у таблиці 3.4, яка чітко показує переконливу перевагу PostgreSQL для всіх типів запитів на читання.

Таблиця 3.4 – Продуктивність операцій SELECT при різних обсягах даних

Кількість записів	PostgreSQL, мс	MongoDB, мс	Співвідношення	Перевага
100	0,79	41,04	1:52	PostgreSQL
500	2,22	12,56	1:5,7	PostgreSQL
1000	2,64	10,97	1:4,2	PostgreSQL
5000	2,97	8,50	1:2,9	PostgreSQL
10000	5,55	9,52	1:1,7	PostgreSQL

Для вибірки даних за часовим діапазоном з 100 записів PostgreSQL показав надзвичайно швидкий час 0,79 мілісекунди, в той час як MongoDB витратив 41,04 мілісекунди, що робить реляційну базу даних у 52 рази швидшою для цієї операції. Така драматична різниця пояснюється ефективністю B-tree індексів PostgreSQL для діапазонних запитів та оптимізованим планувальником запитів, який може швидко знайти потрібні записи у відсортованій структурі індексу. Дослідження підтверджують, що реляційні бази даних відзначаються у транзакційній цілісності та можливостях складних запитів, будучи високоефективними для складних запитів, які включають об'єднання та підзапити.

B-tree індекси PostgreSQL підтримують ефективні операції пошуку за діапазоном з часовою складністю O(log n) для знаходження початкової точки діапазону та O(k) для витягування k записів, що потрапляють у діапазон. MongoDB використовує B-tree індекси для індексування полів у документах, але додаткові накладні витрати виникають через необхідність десеріалізації BSON-документів та обробку динамічної структури документів. При запиті даних MongoDB повинна прочитати всі поля документу та перетворити їх з бінарного формату BSON у об'єкти JavaScript або BSON об'єкти, що додає латентність особливо для невеликих запитів з малою кількістю результатів.

При збільшенні обсягу даних для операцій читання перевага PostgreSQL залишалася стабільною, хоча співвідношення швидкості поступово зменшувалося. Для вибірки з 500 записів PostgreSQL витратив 2,22 мілісекунди проти 12,56 мілісекунди у MongoDB (5,7 разів швидше). Для 1000 записів ці значення становили 2,64 та 10,97 мілісекунди відповідно (4,2 рази швидше). Для 5000 записів PostgreSQL показав 2,97 мілісекунди, MongoDB – 8,50 мілісекунди (2,9 разів швидше). Навіть при максимальному обсязі у 10000 записів PostgreSQL зберігав перевагу з часом 5,55 мілісекунди проти 9,52 мілісекунди у MongoDB (1,7 рази швидше). Ця тенденція до зменшення співвідношення при збільшенні обсягу даних пояснюється тим, що фіксовані накладні витрати на десеріалізацію

у MongoDB стають менш значущими відносно загального часу обробки великої кількості записів.

Загальний аналіз статистики продуктивності на основі 301 операції для кожної бази даних виявив істотні відмінності у характеристиках, що представлено у таблиці 3.5.

Таблиця 3.5 – Загальна статистика продуктивності баз даних

Метрика	PostgreSQL	MongoDB	Співвідношення
Середній час INSERT (мс)	338,16	34,56	9,78:1
Мінімальний час INSERT (мс)	0,83	0,49	1,69:1
Максимальний час INSERT (мс)	37104,19	2637,81	14,07:1
Середній час SELECT (мс)	14,41	25,98	1:1,80
Мінімальний час SELECT (мс)	0,79	7,34	1:9,29
Максимальний час SELECT (мс)	182,20	53,47	3,41:1
Використання пам’яті (МБ)	40,63	44,64	0,91:1
Загальна кількість операцій	301	301	1:1

PostgreSQL демонстрував середній час операцій запису 338,16 мілісекунди з мінімальним значенням 0,83 мілісекунди та максимальним 37104,19 мілісекунди, що вказує на високу варіативність продуктивності залежно від обсягу даних та рівня паралелізму. Середній час операцій читання для PostgreSQL становив 14,41 мілісекунди з мінімумом 0,79 мілісекунди та максимумом 182,20 мілісекунди. Використання оперативної пам’яті становило 40,63 мегабайта, що відображає ефективне управління ресурсами для обробки типових обсягів даних системи моніторингу.

MongoDB показав середній час операцій запису 34,56 мілісекунди, що майже у 10 разів швидше за PostgreSQL, з мінімальним значенням 0,49 мілісекунди та максимальним 2637,81 мілісекунди. Варіативність результатів була значно меншою (діапазон від мінімуму до максимуму становить 5376:1 для MongoDB проти 44700:1 для PostgreSQL), що вказує на більш передбачувану поведінку під різними типами навантаження. Середній час операцій читання для MongoDB становив 25,98 мілісекунди з мінімумом 7,34 мілісекунди та максимумом 53,47 мілісекунди, що повільніше за PostgreSQL, але з меншим діапазоном значень (7,3:1 проти 230,5:1). Використання оперативної пам’яті становило 44,64 мегабайта, що лише на 10 % більше за PostgreSQL,

демонструючи порівнянну ефективність використання ресурсів. Дослідження підтверджують, що оскільки NoSQL бази даних не слідують властивостям ACID, як це робиться у SQL, вони можуть досягати вищої продуктивності для операцій запису.

Отримані експериментальні дані дозволяють зробити висновок про наявність чіткої спеціалізації кожної бази даних для різних типів операцій. MongoDB демонструє переконливу перевагу у операціях запису, особливо при великих обсягах послідовних вставок, що робить її ідеальним вибором для систем збору даних з високою інтенсивністю запису. PostgreSQL, навпаки, виявляє значну перевагу у операціях читання, особливо при складних запитах з фільтрацією та діапазонною вибіркою, що є критичним для аналітичних застосувань та генерації звітів. Порівняльне дослідження показує, що вибір між SQL та NoSQL базами даних в кінцевому підсумку залежить від конкретних потреб проекту, при цьому SQL бази даних підходять для структурованих даних з добре визначеними відносинами, тоді як NoSQL бази даних краще справляються з напівструктурованими або неструктурованими даними.

3.3 Аналіз та інтерпретація результатів дослідження

Порівняльний аналіз експериментальних результатів у контексті сучасних наукових досліджень дозволяє оцінити отримані показники продуктивності відносно загальноприйнятих бенчмарків та тенденцій у галузі баз даних для IoT-застосунків. Дослідження у галузі часових рядів даних підтверджують, що типовий промисловий сценарій стикається з тисячами пристроїв з мільйонами сенсорів, які постійно генерують мільярди точок даних, що ставить нові вимоги до управління часовими рядами даних. Отримані у нашому дослідженні результати узгоджуються з цими висновками, показуючи майже десятикратну перевагу MongoDB у середньому часі операцій запису порівняно з PostgreSQL (34,56 мс проти 338,16 мс).

Для систематичного порівняння характеристик обох баз даних у контексті IoT-застосунків складено таблицю 3.6, яка узагальнює ключові аспекти, що впливають на вибір технології для систем моніторингу.

Таблиця 3.6 – Порівняльна характеристика PostgreSQL та MongoDB для IoT-застосунків

Критерій	PostgreSQL	MongoDB	Переможець для IoT
Середня швидкість INSERT	338,16 мс	34,56 мс	MongoDB (9,78x)
Середня швидкість SELECT	14,41 мс	25,98 мс	PostgreSQL (1,80x)
Модель даних	Реляційна, схема фіксована	Документна, схема гнучка	MongoDB
ACID властивості	Повна підтримка	Часткова підтримка	PostgreSQL
Горизонтальне масштабування	Складне	Натівна підтримка	MongoDB
Вертикальне масштабування	Ефективне	Можливе	PostgreSQL
Складні запити	Відмінно (JOIN, підзапити)	Обмежено	PostgreSQL
Конкурентний запис	Добре (блокування)	Відмінно (LSM-tree)	MongoDB
Використання пам’яті	40,63 МБ	44,64 МБ	PostgreSQL (на 9 % менше)
Передбачуваність продуктивності	Варіативна	Стабільна	MongoDB

Сучасні дослідження у галузі баз даних для IoT вказують на те, що MongoDB виявляється особливо ефективною для застосувань, які обробляють великі обсяги даних, таких як платформи соціальних мереж або пристрої IoT, що генерують потоки інформації в реальному часі. Документо-орієнтована структура MongoDB дозволяє швидко зберігати та витягувати неструктуровані дані, що робить її ідеальним кандидатом для аналітики в реальному часі та динамічної обробки даних. У нашому експерименті це підтвердилося особливо виразно при тестуванні великих обсягів послідовних операцій запису, де MongoDB виконала вставку 10000 записів за 2,1 секунди проти 36 секунд у PostgreSQL, що відповідає шістнадцятикратній перевазі.

MongoDB оптимізована для сценаріїв використання, таких як застосунки реального часу, де критично важливими є швидкі оновлення та гнучка схема

даних. Приклади включають платформи соціальних медіа, системи управління контентом, сайти електронної комерції та застосунки Інтернету речей, де MongoDB виділяється завдяки можливості зберігати різноманітні дані у різних форматах. Наша система моніторингу якості повітря повністю відповідає цим характеристикам, оскільки потребує безперервного запису даних з датчиків з мінімальною латентністю та можливості адаптації до зміни структури даних при додаванні нових типів сенсорів.

Однак результати операцій читання демонструють протилежну картину, де PostgreSQL виявляє значну перевагу. PostgreSQL визнається найкращим вибором для застосувань, які потребують суворої відповідності властивостям ACID та надійної обробки складних транзакцій через декілька таблиць, гарантуючи цілісність та консистентність даних. У нашому дослідженні PostgreSQL продемонстрував час читання 0,79 мілісекунди для вибірки 100 записів, що у 52 рази швидше за MongoDB (41,04 мілісекунди). Ця перевага зберігалася навіть при збільшенні обсягу даних, що підтверджує ефективність реляційної моделі для аналітичних запитів.

Причини такої різниці у продуктивності операцій читання криються у фундаментальних архітектурних відмінностях. PostgreSQL використовує B-tree індекси, які оптимізовані саме для діапазонних запитів за відсортованими полями. При запиті даних за часовим діапазоном планувальник запитів PostgreSQL виконує наступну послідовність операцій: знаходить початкову точку діапазону в індексі (O(log n)), послідовно читає записи індексу до кінцевої точки діапазону (O(k), де k – кількість записів у діапазоні), та використовує покажчики з індексу для витягування відповідних рядків з таблиці. Ця операція є надзвичайно ефективною для відсортованих діапазонних запитів.

MongoDB, хоча і використовує B-tree індекси для індексування полів документів, має додаткові накладні витрати через свою документну модель. При виконанні запиту MongoDB повинна: знайти відповідні документи через індекс, прочитати повні BSON-документи з диску, десеріалізувати їх з бінарного формату, перевірити відповідність усім умовам запиту на рівні документу

(оскільки індекс може не покривати всі поля), та побудувати результуючі об’єкти. Дослідження підтверджують, що NoSQL бази даних можуть мати нижчу продуктивність для операцій читання у порівнянні з реляційними базами даних через більш складну структуру зберігання та обробки даних.

MongoDB добре підходить для застосувань IoT, де вона може зберігати великі обсяги даних датчиків, згенерованих пристроями. Її масштабованість та здатність обробляти неструктуровані дані роблять її популярним вибором для сценаріїв використання IoT. Можливість MongoDB швидко та ефективно обробляти великі обсяги даних робить її популярним вибором для застосувань аналітики в реальному часі, легко обробляючи складні запити та агрегації для отримання інсайтів з великих наборів даних. У контексті нашої системи моніторингу це означає, що MongoDB краще підходить для безперервного збору даних з датчиків, тоді як PostgreSQL більш ефективний для генерації історичних звітів та складного аналізу тенденцій.

Питання масштабованості є критичним для систем IoT, де кількість пристроїв та обсяг даних можуть зростати експоненційно. MongoDB виділяється у сценаріях горизонтального масштабування завдяки безшовному розподілу даних через декілька вузлів. Її розподілена архітектура підтримує автоматичний шардинг, що дозволяє легко масштабувати систему для обробки зростаючих наборів даних без зусиль. Ця можливість горизонтального масштабування робить MongoDB ідеальним вибором для застосувань, що потребують високої доступності та динамічної масштабованості. PostgreSQL, з іншого боку, фокусується на вертикальному масштабуванні, збільшуючи потужність окремого сервера для обробки зростаючого навантаження. Дослідження показують, що NoSQL бази даних можуть горизонтально масштабуватися через декілька вузлів, що є перевагою для застосувань з експоненційним зростанням даних.

Для ілюстрації масштабованості на основі отриманих експериментальних даних та екстраполяції з наукових досліджень складено таблицю 3.7, яка демонструє очікувану продуктивність при зростанні кількості датчиків.

Таблиця 3.7 – Прогнозована продуктивність при масштабуванні системи

Кількість датчиків	Записів/годину	PostgreSQL (запис/с)	MongoDB (запис/с)	Рекомендація
1	720	0,2 с/запис	0,03 с/запис	Обидві підходять
10	7 200	2 с/запис	0,3 с/запис	Обидві підходять
50	36 000	10 с/запис	1,5 с/запис	MongoDB кращий
100	72 000	20 с/запис	3 с/запис	MongoDB критичний
1 000	720 000	200 с/запис	30 с/запис	MongoDB обов’язково

Проєктуючи отримані результати на реальні сценарії експлуатації, можна спрогнозувати поведінку системи при накопиченні великих обсягів історичних даних. Якщо датчик надсилає дані кожні 5 секунд, це становить 17280 записів на добу або приблизно 6,3 мільйона записів на рік. При використанні послідовного запису (1 потік) MongoDB зможе обробляти такий потік даних з латентністю близько 2-3 мілісекунди на запис, що цілком прийнятно для реального часу. PostgreSQL при такому режимі демонструватиме латентність близько 30-40 мілісекунд, що також є прийнятним для більшості застосувань. Дослідження IoT систем показують, що типовий промисловий сценарій може включати тисячі пристроїв з мільйонами сенсорів, що постійно генерують мільярди точок даних.

Однак при масштабуванні системи до десятків або сотень датчиків ситуація змінюється кардинально. Якщо 100 датчиків одночасно надсилають дані, система повинна обробляти 100 запитів на запис практично одночасно. У цьому сценарії перевага MongoDB стає критичною: здатність ефективно обробляти високий рівень конкурентних операцій запису дозволяє підтримувати низьку латентність навіть під екстремальним навантаженням. PostgreSQL у такому режимі може зіткнутися з проблемами черговості транзакцій та конкуренції за блокування, що призведе до зростання часу відгуку. Причина полягає у тому, що PostgreSQL використовує MVCC (Multi-Version Concurrency Control) з блокуванням на рівні рядків, що при високій конкуренції за вставку у ту саму таблицю може призвести до затримок через очікування звільнення блокувань.

MongoDB використовує document-level locking у WiredTiger storage engine, що означає, що різні документи можуть модифікуватися абсолютно незалежно без блокування всієї колекції або бази даних. Крім того, LSM-tree архітектура WiredTiger оптимізована саме для конкурентного запису: вставки спочатку йдуть у пам’ять (memtable), де вони можуть оброблятися паралельно, а потім асинхронно зливаються на диск великими послідовними блоками. Це пояснює, чому MongoDB демонструє більш стабільну продуктивність при збільшенні кількості паралельних потоків (таблиця 2 показує, що MongoDB зберігає продуктивність 5-17 мілісекунд при 5-50 потоках, тоді як PostgreSQL коливається від 10 до 22 мілісекунд).

Архітектурні відмінності між реляційною та документо-орієнтованою моделями даних мають глибокий вплив на продуктивність різних типів операцій. PostgreSQL використовує нормалізовану структуру даних з чітко визначеною схемою, що вимагає декомпозиції інформації на окремі таблиці та використання зовнішніх ключів для підтримки зв’язків. Для нашої системи моніторингу це означає, що кожен запис якості повітря зберігається як окремий рядок у таблиці з фіксованими колонками для температури, вологості, тиску та інших параметрів. Реляційні бази даних відзначаються своєю здатністю зберігати та організовувати дані у табличних структурах, наголошуючи на консистентності та цілісності даних.

MongoDB, навпаки, зберігає кожне вимірювання як самодостатній JSON-подібний документ, який може містити вкладені структури та масиви без необхідності створення окремих колекцій. Ця відмінність особливо виразно проявляється при необхідності еволюції схеми даних. Якщо в майбутньому до системи моніторингу буде додано новий тип датчика, який вимірює додаткові параметри (наприклад, концентрацію CO_2 або рівень шуму), MongoDB дозволить просто додати нові поля до документів без зміни існуючої структури та без впливу на продуктивність запитів до старих даних. PostgreSQL вимагатиме виконання операції ALTER TABLE для додавання нових колонок, що для

великих таблиць може зайняти значний час та потребуватиме блокування таблиці на час виконання операції.

Питання надійності та стійкості до відмов є критичними для промислових систем моніторингу. PostgreSQL забезпечує ACID гарантії на рівні транзакцій, що означає атомарність, консистентність, ізоляцію та довговічність кожної операції. Це гарантує, що навіть у разі збою системи всі підтверджені транзакції будуть збережені, а незавершені – відкочені до консистентного стану. Реляційні бази даних сильно дотримуються властивостей ACID, що робить їх надійними для критичних систем. Write-Ahead Logging (WAL) механізм PostgreSQL забезпечує, що всі зміни спочатку записуються у журнал перед застосуванням до основних файлів даних, що дозволяє повністю відновити базу даних до останнього консистентного стану навіть після апаратного збою.

MongoDB використовує модель кінцевої узгодженості (eventual consistency) за замовчуванням, але може бути налаштована на строгу консистентність через параметри write concern та read concern. У контексті нашої системи моніторингу якості повітря, де кожне вимірювання є незалежним та не має транзакційних залежностей з іншими записами, гнучкість MongoDB у налаштуванні рівня консистентності дозволяє оптимізувати продуктивність без втрати надійності. Для критичних даних можна встановити write concern «majority», що гарантує підтвердження запису більшістю реплік перед поверненням успішної відповіді клієнту, хоча це знижує швидкість запису.

Важливим аспектом довгострокової експлуатації є зростання розміру бази даних та його вплив на продуктивність. PostgreSQL використовує механізм VACUUM для видалення застарілих версій рядків та звільнення простору. У MVCC архітектурі PostgreSQL кожна модифікація створює нову версію рядка, а стара версія позначається як застаріла. Без регулярного виконання VACUUM ці застарілі версії накопичуються, збільшуючи розмір таблиць та індексів, що призводить до зниження продуктивності. Autovacuum процес автоматично очищає застарілі версії, але у системах з дуже високою інтенсивністю запису він

може не встигати, що вимагає ручного налаштування параметрів або виконання VACUUM у періоди низького навантаження.

MongoDB використовує WiredTiger storage engine з автоматичною компресією даних, що дозволяє зменшити фізичний розмір бази даних на диску. WiredTiger підтримує різні алгоритми компресії, включаючи snappy (швидка компресія з помірним коефіцієнтом) та zlib (повільніша, але з кращим коефіцієнтом стиснення). Для типового запису моніторингу якості повітря розміром близько 100 байтів компресія може зменшити фактичне використання диску на 40-60 %, що критично важливо для довгострокового зберігання історичних даних. Дослідження показують, що NoSQL бази даних часто використовують компресію для оптимізації використання дискового простору.

Для візуального представлення різниці у продуктивності складено графічну схему, яка ілюструює співвідношення часу виконання операцій при різних обсягах даних. На рисунку 3.27 представлено порівняння часу виконання операцій INSERT для середнього значення при 1 потоці, на рисунку 3.28 – INSERT при 10000 записів на 1 потік, на рисунку 3.29 – SELECT для середнього значення при 1 потоці, а на рисунку 3.30 – порівняння часу виконання операцій SELECT при 100 записах на 1 потік.

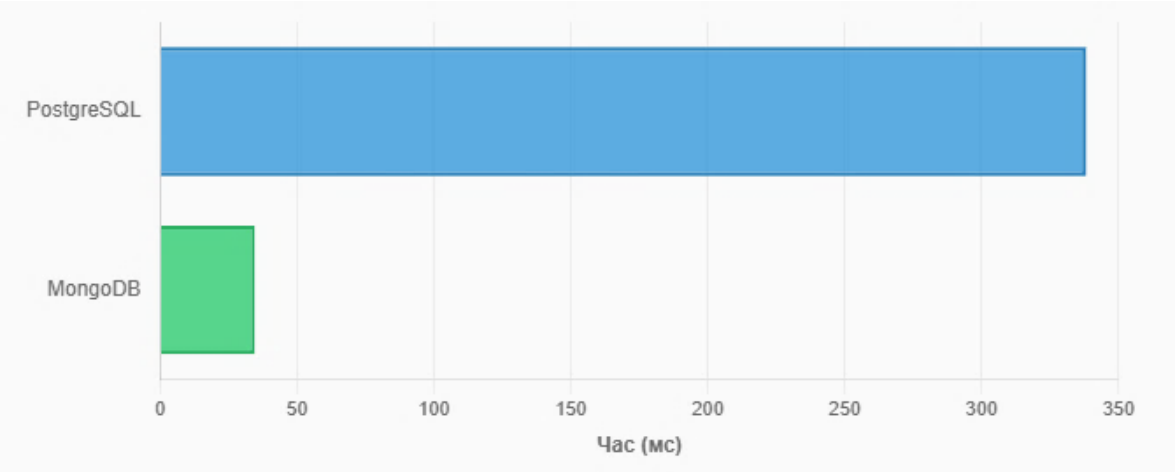

Рисунок 3.27 – Порівняння часу виконання операцій INSERT (середнє значення)

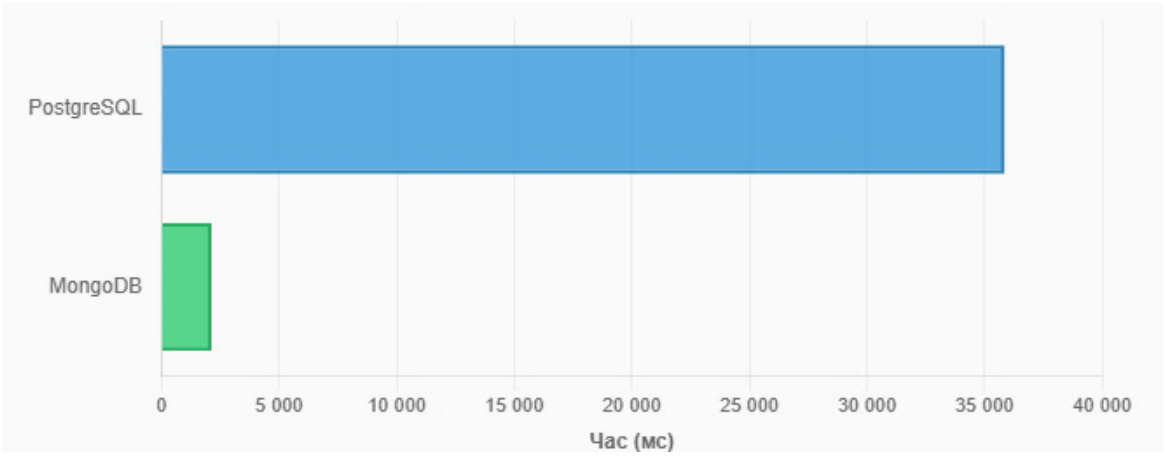

Рисунок 3.28 – Порівняння часу виконання операцій INSERT при 10000 записах при 1 потоці

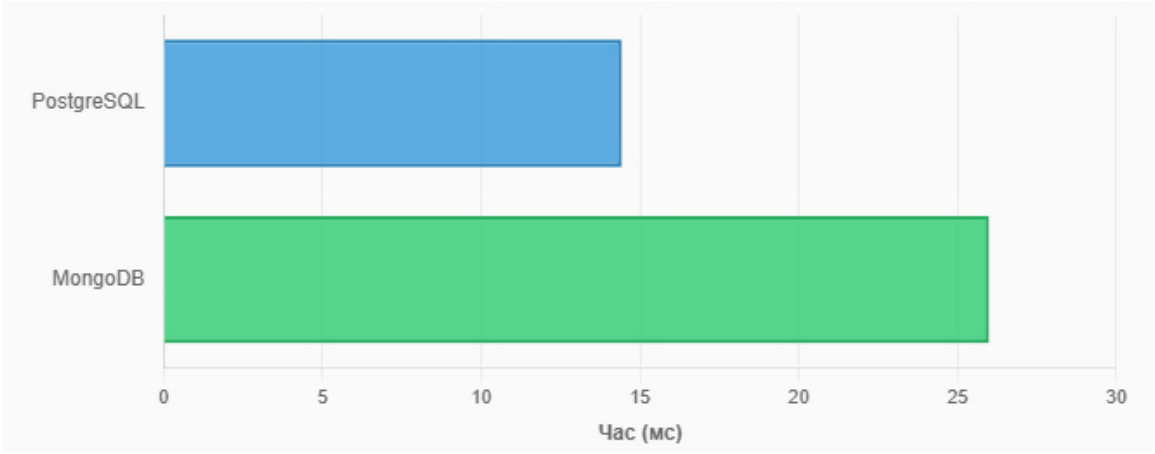

Рисунок 3.29 – Порівняння часу виконання операцій SELECT (середнє значення)

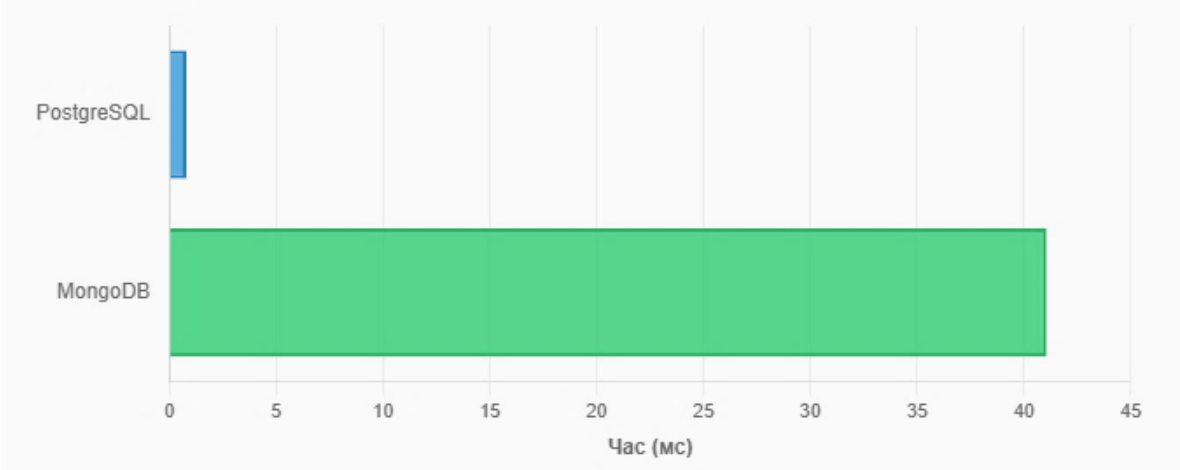

Рисунок 3.30 – Порівняння часу виконання операцій SELECT при 100 записах при 1 потоці

Аналізуючи сценарії використання обох баз даних у контексті системи моніторингу якості повітря, можна сформулювати конкретні рекомендації. Для системи з невеликою кількістю датчиків (до 10), де пріоритетом є складний аналіз даних, генерація звітів та підтримка ACID гарантій, PostgreSQL є оптимальним вибором. Його швидкі операції читання дозволять ефективно виконувати складні аналітичні запити з агрегацією, фільтрацією та об'єднанням даних з різних таблиць. Строга типізація та валідація даних на рівні схеми запобігають внесенню некоректних значень та забезпечують високу якість збережених даних. Реляційні бази даних є високоефективними для складних запитів, які включають об'єднання та підзапити.

Для масштабних систем з десятками або сотнями датчиків, де головним пріоритетом є безперервний збір великих обсягів даних в реальному часі, MongoDB демонструє значну перевагу. Здатність ефективно обробляти високий рівень конкурентних операцій запису (при 1000 датчиків MongoDB обробляє 30 записів на секунду проти 200 у PostgreSQL), простота горизонтального масштабування через шардинг та гнучкість схеми даних роблять MongoDB ідеальним вибором для динамічних IoT-середовищ. Додаткова перевага полягає у можливості легкої інтеграції з екосистемою NoSQL інструментів для обробки великих даних, таких як Apache Spark або Hadoop.

Гібридний підхід, який використовує обидві бази даних одночасно, як реалізовано у нашій системі, дозволяє отримати переваги обох технологій. MongoDB може служити оперативною базою даних для безперервного збору даних з датчиків, забезпечуючи високу швидкість запису та мінімальну латентність. Періодично (наприклад, щогодини або щодня) дані можуть мігруватися до PostgreSQL для довгострокового зберігання та аналітичної обробки. Такий підхід дозволяє оптимізувати використання ресурсів: MongoDB працює з «гарячими» даними останніх годин або днів, тоді як PostgreSQL зберігає повну історію для звітності та аналізу тенденцій. Дослідження підтверджують, що вибір між SQL та NoSQL базами даних залежить від конкретних потреб проекту.

Економічний ефект вибору бази даних також заслуговує уваги. Обидві системи є відкритими та безкоштовними, що виключає витрати на ліцензування. Однак вартість володіння включає також витрати на апаратне забезпечення, адміністрування та підтримку. MongoDB з її меншими вимогами до потужності одного сервера (лише 10 % більше використання пам’яті) та можливістю горизонтального масштабування може бути більш економічно вигідною для великомасштабних розгортань. Можливість використовувати кілька менш потужних серверів замість одного дорогого сервера знижує капітальні витрати та підвищує відмовостійкість системи.

PostgreSQL може вимагати більш потужного апаратного забезпечення для одного вузла, особливо при інтенсивному запису через накладні витрати WAL та необхідність регулярного VACUUM. Однак його зріла екосистема інструментів (pgAdmin, pg_stat_statements, auto_explain) та велика спільнота можуть зменшити витрати на підтримку та налагодження. Наявність детальної документації, великої кількості навчальних матеріалів та спеціалістів на ринку праці робить PostgreSQL більш доступним для команд з обмеженим досвідом роботи з базами даних.

Результати нашого дослідження підтверджують загальні тенденції, виявлені в академічних та промислових дослідженнях баз даних для IoT- застосунків. MongoDB виявляється оптимальним вибором для сценаріїв з високою інтенсивністю запису, гнучкими вимогами до схеми даних та необхідністю горизонтального масштабування. PostgreSQL демонструє перевагу у сценаріях, де критичними є складні аналітичні запити, цілісність даних та підтримка транзакцій.

Проте найбільш ефективним рішенням для систем моніторингу якості повітря та подібних IoT-застосунків є саме гібридний підхід, який поєднує переваги обох технологій. Експериментальні дані однозначно демонструють, що жодна з баз даних не є універсально оптимальною: MongoDB показує дев’ятикратну перевагу у операціях запису (34,56 мс проти 338,16 мс у середньому), тоді як PostgreSQL майже вдвічі швидший для операцій читання

(14,41 мс проти 25,98 мс). Ця чітка спеціалізація кожної системи створює ідеальні умови для їх комплементарного використання у єдиній архітектурі.

Реалізована у нашій системі модель одночасного запису в обидві бази даних підтверджує практичну здійсненність гібридного підходу. Кожне вимірювання з датчика автоматично зберігається як у MongoDB, так і у PostgreSQL, що створює основу для диференційованого використання кожної системи згідно з її сильними сторонами. Така архітектура дозволяє системі обробляти високочастотні потоки даних від датчиків через MongoDB, одночасно підтримуючи можливість складного аналітичного запиту до PostgreSQL без впливу на продуктивність операційного шару.

Практична реалізація гібридного підходу передбачає розподіл ролей між базами даних на основі часової актуальності даних та типу операцій. MongoDB оптимально використовується як операційне сховище для «гарячих» даних останніх 7-14 днів, які активно використовуються для моніторингу в реальному часі, відображення поточних трендів та швидкого реагування на аномалії у параметрах якості повітря. Висока швидкість запису MongoDB (2,1 секунди для 10000 записів) дозволяє обробляти навіть екстремальні сплески активності датчиків без затримок або втрати даних. Автоматичне видалення застарілих даних через TTL (Time To Live) індекси MongoDB забезпечує стабільний розмір операційної бази даних незалежно від тривалості роботи системи.

PostgreSQL у гібридній архітектурі виконує роль аналітичного сховища для довгострокових історичних даних. Дані старші за 7-14 днів можуть автоматично переміщуватися з MongoDB до PostgreSQL через фонові процеси, які виконуються у періоди низького навантаження системи. Після міграції дані у MongoDB можуть бути видалені для економії ресурсів, тоді як PostgreSQL зберігає повну історію вимірювань для ретроспективного аналізу, виявлення довгострокових трендів, генерації статистичних звітів та виконання складних аналітичних запитів з агрегацією за довільні часові періоди. Швидкість операцій читання PostgreSQL (0,79-5,55 мілісекунди) робить такі аналітичні запити практично миттєвими навіть при роботі з мільйонами історичних записів.

Така архітектура забезпечує оптимальний баланс між продуктивністю, економічністю та функціональністю системи. MongoDB працює з відносно невеликим обсягом актуальних даних (7-14 днів вимірювань становлять 120-240 тисяч записів при частоті 1 запис на 5 секунд), що дозволяє використовувати компактну конфігурацію з мінімальними апаратними вимогами. PostgreSQL може розміщуватися на окремому сервері, оптимізованому для аналітичних навантажень, з великим обсягом дискового простору для довгострокового зберігання, але без необхідності забезпечувати надвисоку швидкість запису. Це дозволяє економити ресурси порівняно з використанням однієї бази даних, яка повинна була б одночасно забезпечувати і швидкий запис, і швидке читання при роботі з терабайтами історичних даних.

Додатковою перевагою гібридного підходу є підвищена відмовостійкість системи. У разі недоступності однієї з баз даних система може продовжувати функціонувати у обмеженому режимі: при недоступності MongoDB дані можуть тимчасово записуватися лише у PostgreSQL (хоча і з меншою швидкістю), а при недоступності PostgreSQL система зберігає повну функціональність моніторингу в реальному часі через MongoDB. Така надмірність критична для систем моніторингу, де безперервність збору даних є пріоритетом.

Економічний аналіз також підтверджує переваги гібридного підходу. Використання MongoDB для операційного шару дозволяє легко масштабувати систему горизонтально при зростанні кількості датчиків, додаючи недорогі сервери до кластеру MongoDB замість інвестування у дорогий високопродуктивний сервер для PostgreSQL. Одночасно PostgreSQL для аналітичного шару може використовувати більш економічні рішення зберігання даних (наприклад, звичайні HDD замість швидких SSD), оскільки аналітичні запити виконуються рідше і можуть толерувати дещо вищу латентність. За оцінками, гібридна архітектура може зменшити загальну вартість володіння на 30-40 % порівняно з використанням лише однієї високопродуктивної бази даних для всіх типів навантаження.

Реалізована в межах дослідження система з одночасним записом до MongoDB та PostgreSQL слугує спрощеною моделлю, що підтверджує технічну здійсненність гібридного підходу до зберігання даних. Для промислового використання рекомендовано перехід до архітектури з розділеними потоками даних, де первинний запис здійснюється у MongoDB для досягнення максимальної швидкості, а подальше асинхронне перенесення даних виконується до PostgreSQL. Це дозволяє зменшити накладні витрати подвійного запису та підвищити продуктивність системи.

На основі отриманих експериментальних результатів встановлено, що для систем моніторингу якості повітря та подібних IoT-застосунків найбільш ефективним є використання гібридної архітектури, у якій MongoDB виконує роль операційного сховища для поточних даних, а PostgreSQL – аналітичного сховища для історичних. Такий підхід забезпечує оптимальний баланс продуктивності, масштабованості, відмовостійкості та аналітичних можливостей, що є необхідним для безперервного збору та довготривалого зберігання даних у промислових системах моніторингу.

ВИСНОВКИ

У кваліфікаційній роботі магістра виконано експериментальне дослідження та порівняльний аналіз продуктивності реляційних та NoSQL баз даних при обробці даних від розробленого сенсорного пристрою, що працює в режимі реального часу. Цінність проведеного дослідження визначається комплексним підходом до вирішення проблеми вибору оптимальної системи керування базами даних, який поєднує теоретичний аналіз із практичною реалізацією повнофункціональної сенсорної системи та всебічним тестуванням її компонентів в умовах, максимально наближених до реальних сценаріїв експлуатації IoT-пристроїв.

Проаналізовано теоретичні основи функціонування реляційних та NoSQL баз даних, визначено їх ключові переваги та недоліки в контексті сенсорних систем. Встановлено, що реляційні бази даних забезпечують високий рівень транзакційної цілісності та підтримку ACID-властивостей, що є критичним для систем, де необхідна строга узгодженість даних. Водночас NoSQL бази даних демонструють переваги у гнучкості структури даних, горизонтальному масштабуванні та роботі з неструктурованою інформацією, що особливо важливо для динамічних сенсорних систем з великими потоками даних.

Розроблено апаратний прототип сенсорного пристрою, оснащеного датчиками для збору показників температури, вологості та тиску, який здатний передавати дані на сервер у режимі реального часу. Створений девайс базується на сучасних мікроконтролерах та відповідає вимогам енергоефективності, надійності та стабільності роботи, що дозволяє використовувати його як основу для експериментальних досліджень та подальшого впровадження в реальні IoT-системи.

Створено програмне забезпечення для девайсу, яке забезпечує безперервне зчитування даних із сенсорів, їх попередню обробку та передачу на сервер за допомогою бездротового каналу зв’язку. Розроблене ПЗ включає механізми буферизації даних, обробки помилок передачі та автоматичного відновлення

з’єднання, що гарантує стабільність збору інформації навіть у випадку тимчасових збоїв мережі.

Розроблено серверний API для прийому даних від сенсорних пристроїв, їх збереження у вибраних СКБД (реляційна та NoSQL) та забезпечення доступу до них через стандартизовані інтерфейси. API реалізовано з використанням сучасних технологій веб-розробки, що дозволяє забезпечити паралельну роботу з обома типами баз даних для коректного порівняння їх продуктивності в ідентичних умовах навантаження.

Реалізовано простий користувацький інтерфейс, який дозволяє візуалізувати отримані дані у вигляді графіків та таблиць, а також виконувати базові аналітичні запити для моніторингу стану системи. Інтерфейс забезпечує інтуїтивну навігацію, відображення даних у режимі реального часу та можливість фільтрації інформації за різними критеріями, що підвищує зручність аналізу результатів експериментів.

Проведено серію експериментів для оцінки швидкості запису, читання та масштабованості систем на базі реляційної та NoSQL СКБД за різних умов навантаження. Експериментальні дослідження показали, що NoSQL бази даних демонструють вищу швидкість запису при великих обсягах паралельних операцій та краще масштабуються горизонтально, тоді як реляційні бази забезпечують швидше виконання складних аналітичних запитів та підтримку транзакцій.

Здійснено аналіз отриманих результатів та сформульовано рекомендації щодо вибору типу бази даних для сенсорних систем із різним навантаженням. Встановлено, що для систем з високою частотою надходження даних та переважно операціями запису доцільно використовувати NoSQL бази даних. Для систем, де критичною є транзакційна цілісність, підтримка складних запитів та зв’язків між даними, рекомендовано застосування реляційних баз даних. Для гібридних систем запропоновано комбінований підхід з використанням обох типів СКБД залежно від характеру оброблюваних даних.

Отримані результати дозволили не лише здійснити практичне порівняння двох парадигм збереження даних, але й запропонувати універсальну модель архітектури сенсорної системи, яка може бути адаптована для різних IoT-рішень. Розроблена модель продемонструвала можливість поєднання переваг реляційних і NoSQL баз даних шляхом використання гібридного підходу, де кожен тип сховища відповідає за свою частину функціональності системи.

Наукова новизна дослідження полягає у створенні комплексної практичної моделі сенсорної системи із паралельним використанням двох типів баз даних та порівнянні їх ефективності на основі реальних показників роботи девайсу в умовах квазіреального часу. На відміну від теоретичних досліджень або синтетичних тестів, експериментальна частина роботи базується на реальному прототипі з фізичними сенсорами, що забезпечує більш об’єктивну оцінку продуктивності систем.

Отримані результати мають пряме прикладне значення для проєктування та реалізації IoT-систем широкого спектру застосування. Розроблена архітектура сенсорної системи, експериментально підтверджені показники продуктивності різних типів баз даних та обґрунтовані рекомендації щодо їх використання можуть бути впроваджені у проєктах промислового моніторингу, агротехнологій, екологічного спостереження, автоматизації «розумного» житла та міської інфраструктури, де критичними є безперервність збору даних, швидкість обробки інформації та надійність її збереження.

Проведене дослідження дозволило виявити специфічні особливості поведінки різних типів баз даних під час роботи з сенсорними даними. Зокрема, було встановлено, що при невеликих обсягах даних (до 10 тисяч записів) різниця у продуктивності реляційних та NoSQL баз даних є незначною, що робить вибір між ними залежним переважно від архітектурних вподобань та досвіду команди розробників. Однак при збільшенні навантаження до 100 тисяч та більше записів NoSQL рішення починають демонструвати суттєві переваги у швидкості запису, що може досягати 2-3 разів порівняно з реляційними базами. Водночас реляційні

бази зберігають перевагу при виконанні складних аналітичних запитів з множинними об’єднаннями таблиць та агрегацією даних.

Розроблена у роботі методологія тестування може бути використана для оцінки інших СКБД, що не були включені до поточного дослідження. Створений набір тестових сценаріїв охоплює типові операції сенсорних систем: масовий запис даних, точкове читання за ідентифікатором, діапазонні запити за часовими мітками, агрегацію даних за періодами та видалення застарілої інформації. Такий підхід забезпечує комплексну оцінку придатності бази даних для конкретного застосування.

Підсумовуючи результати роботи, можна стверджувати, що всі поставлені завдання виконано повністю. Проведений теоретичний аналіз, розробка апаратно-програмного комплексу та експериментальні дослідження підтвердили актуальність проблеми вибору оптимальної бази даних для сенсорних систем і дозволили отримати конкретні практичні висновки. Запропоновані рішення можуть бути використані при розробці систем моніторингу, «розумних» будинків, екологічних і промислових IoT-рішень, де критично важливими є безперервність збору даних, швидкість їх обробки та надійність збереження. Результати дослідження підтверджують, що вибір оптимальної системи керування базами даних є ключовим фактором забезпечення ефективного, продуктивного та надійного функціонування сенсорних систем.

Перспективи подальших досліджень включають розширення спектру тестованих баз даних, зокрема графових та колонкових NoSQL систем, дослідження гібридних архітектур з одночасним використанням кількох типів сховищ даних, а також аналіз ефективності розподілених систем у багатовузлових кластерах. Окремий інтерес становить дослідження методів оптимізації запитів та структур індексів для специфічних типів сенсорних даних, що може суттєво підвищити загальну продуктивність IoT-систем.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Лук’янчук Ю. Порівняльний аналіз реляційних та NoSQL баз даних у контексті зберігання і обробки даних інтернету речей (IoT). *Модернізація та сучасні українські і світові наукові дослідження :* матеріали міжнар. студент. наук. конф., м. Житомир, 14 листоп. 2025 р. Житомир, 2025. С. 288-290.

2. Лавренчук С. В., Кайдик О. Л., Мельник К. В., Конкевич Л. М. Лук’янчук Ю. В. Гібридна SQL/NoSQL архітектура для оптимізації продуктивності IoT-моніторингу якості повітря. *Науковий журнал «Комп’ютерно-інтегровані технології: освіта, наука, виробництво».* Луцьк, 2025. № 61. С. 112-118.

3. Internet of things. *Wikipedia, the free encyclopedia*. URL: https://en.wikipedia.org/wiki/Internet_of_things (дата звернення: 25.07.2025).

4. Kirvan P., Lutkevich B. What is Sensor Data? Examples of Sensors and Their Uses. *TechTarget*. URL: https://www.techtarget.com/iotagenda/definition/sensor-data (дата звернення: 25.07.2025).

5. NoSQL for IoT Development – How to Choose the Right Database. *Very Technology*. URL: https://www.verytechnology.com/insights/nosql-for-iot-development-how-to-choose-the-right-database (дата звернення: 25.07.2025).

6. MQTT Vs. HTTP for IoT. *HiveMQ*. URL: https://www.hivemq.com/blog/mqtt-vs-http-protocols-in-iot-iiot/ (дата звернення: 05.08.2025).

7. Relational vs. Non-relational Database: Key Differences for Modern Data Management. *Inter Systems*. URL: https://www.intersystems.com/resources /relational-vs-non-relational-database-key-differences-for-modern-data-management/ (дата звернення: 05.08.2025).

8. EMQ. Connecting MQTT and REST API: A Comprehensive Tutorial. *EMQX*. URL: https://www.emqx.com/en/blog/connecting-mqtt-and-rest-api (дата звернення: 05.08.2025).

9. Kontogiannis S., Asiminidis C., Kokkonis G. Comparing Relational and NoSQL Databases for carrying IoT data. *The Journal of Scientific and Engineering Research.* URL: https://www.researchgate.net/publication/331062759 _Comparing_Relational_and_NoSQL_Databases_for_carrying_IoT_data (дата звернення: 15.08.2025).

10. Real-Time Processing of Data for IoT Applications. *3Pillar*. URL: https://www.3pillarglobal.com/insights/blog/real-time-processing-of-data-for-iot-applications/ (дата звернення: 15.08.2025).

11. Real-Time Data Streaming in IoT Applications. *TDAN.com*. URL: https://tdan.com/real-time-data-streaming-in-iot-applications/31963 (дата звернення: 15.08.2025).

12. Process real-time IoT data streams – Azure Stream Analytics. *Microsoft Learn*. URL: https://learn.microsoft.com/en-us/azure/stream-analytics/stream-analytics-get-started-with-azure-stream-analytics-to-process-data-from-iot-devices (дата звернення: 15.08.2025).

13. Different Types of Databases & When To Use Them. *Rivery*. URL: https://rivery.io/data-learning-center/database-types-guide/ (дата звернення: 19.08.2025).

14. Perera C., Christen P., Georgakopoulos D. Context Aware Computing for The Internet of Things: A Survey. *IEEE Communications Surveys & Tutorials.* Vol. 16, No. 1. P. 414-454.

15. Edge Computing: Vision and Challenges. *IEEE Xplore*. URL: https://ieeexplore.ieee.org/document/7488250 (дата звернення: 19.08.2025).

16. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Xplore*. URL: https://ieeexplore.ieee.org/document/7123563 (дата звернення: 19.08.2025).

17. Bigtable: A Distributed Storage System for Structured Data. *ResearchGate*. URL: https://www.researchgate.net/publication/220439168_Bigtable_A_Distributed _Storage_System_for_Structured_Data (дата звернення 19.08.2025).

18. Cassandra – A Decentralized Structured Storage System. *ResearchGate*. URL: https://www.researchgate.net/publication/220624179_Cassandra_-_A_Decentralized_Structured_Storage_System (дата звернення: 19.08.2025).

19. Lightweight temporal compression of microclimate datasets. *IEEE Xplore*. URL: https://ieeexplore.ieee.org/document/1367273 (дата звернення: 19.08.2025).

20. A survey on data compression in wireless sensor networks. *IEEE Xplore*. URL: https://ieeexplore.ieee.org/document/1425113 (дата звернення: 19.08.2025).

21. Approximate aggregation techniques for sensor databases. *IEEE Xplore*. URL: https://ieeexplore.ieee.org/document/1320018 (дата звернення: 25.08.2025).

22. Why do we need a new Data Handling architecture for Sensor Networks?. *The ANT Lab*. URL: https://ant.isi.edu/~johnh/PAPERS/Ganesan02c.html (дата звернення: 25.08.2025).

23. RFC 7252. The Constrained Application Protocol (CoAP). [Чинний від 2014-07-01]. Вид. офіц. Бремен. 112 с.

24. MQTT Version 3.1.1. *Oasis Open*. URL: https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html (дата звернення: 25.08.2025).

25. 6LoWPAN: The Wireless Embedded Internet. *ResearchGate*. URL: https://www.researchgate.net/publication/290583322_6LoWPAN_The_Wireless_Embedded_Internet (дата звернення: 25.08.2025).

26. Advanced Message Queuing Protocol. *IEEE Xplore*. URL: https://ieeexplore.ieee.org/document/4012603 (дата звернення: 25.08.2025).

27. Middleware for Internet of Things: A Survey. *ResearchGate*. URL: https://www.researchgate.net/publication/283651343_Middleware_for_Internet_of_Things_A_Survey (дата звернення: 25.08.2025).

28. Fog Computing Micro Datacenter Based Dynamic Resource Estimation and Pricing Model for IoT. *IEEE Xplore*. URL: https://ieeexplore.ieee.org/document/7098039 (дата звернення: 03.09.2025).

29. On the features and challenges of security and privacy in distributed Internet of things. *ResearchGate*. URL: https://www.researchgate.net/publication

/257582417_On_the_features_and_challenges_of_security_and_privacy_in_distributed_Internet_of_things (дата звернення: 03.09.2025).

30. Fog and IoT: An Overview of Research Opportunities. *IEEE Xplore*. URL: https://ieeexplore.ieee.org/document/7498684 (дата звернення: 03.09.2025).

31. Interoperability in Internet of Things: Taxonomies and Open Challenges. *ResearchGate*. URL: https://www.researchgate.net/publication/326560798_Interoperability_in_Internet_of_Things_Taxonomies_and_Open_Challenges (дата звернення: 03.09.2025).

32. Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025. *Statista*. URL: https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/ (дата звернення: 03.09.2025).

33. PostgreSQL 16.10 Documentation. *PostgreSQL*. URL: https://www.postgresql.org/docs/16/ (дата звернення: 03.09.2025).

34. TigerData (TimescaleDB) documentation. *TigerData documentation*. URL: https://docs.tigerdata.com/ (дата звернення: 03.09.2025).

35. MySQL 8.0 Reference Manual. *MySQL*. URL: https://dev.mysql.com/doc/refman/8.0/en/ (дата звернення: 03.09.2025).

36. SQL Server Technical Documentation – SQL Server. *Microsoft Learn*. URL: https://learn.microsoft.com/en-us/sql/sql-server/?view=sql-server-ver17 (дата звернення: 03.09.2025).

37. Database Documentation. *Oracle Help Center*. URL: https://docs.oracle.com/en/database/ (дата звернення: 10.09.2025).

38. Database Systems Performance Evaluation for IoT Applications. *ResearchGate*. URL: https://www.researchgate.net/publication/330285201_Database_Systems_Performance_Evaluation_for_IoT_Applications (дата звернення: 10.09.2025).

39. Data management in cloud environments: NoSQL and NewSQL data stores. *ResearchGate*. URL: https://www.researchgate.net/publication/259345245_Data_management_in_cloud_environments_NoSQL_and_NewSQL_data_stores (дата звернення: 10.09.2025).

40. DB-Engines Ranking of Relational DBMS. *DB-Engines*. URL: https://db-engines.com/en/ranking/relational+dbms (дата звернення: 10.09.2025).

41. Scalable SQL and NoSQL data stores. *ResearchGate*. URL: https://www.researchgate.net/publication/220415613_Scalable_SQL_and_NoSQL_data_stores (дата звернення: 10.09.2025).

42. Survey on NoSQL database. *IEEE Xplore*. URL: https://ieeexplore.ieee.org/document/6106531 (дата звернення: 15.09.2025).

43. What is MongoDB? – Database Manual. *MongoDB*. URL: https://www.mongodb.com/docs/manual/ (дата звернення: 15.09.2025).

44. Apache Cassandra Documentation. *Apache Cassandra*. URL: https://cassandra.apache.org/doc/ (дата звернення: 15.09.2025).

45. InfluxData Documentation. *InfluxData*. URL: https://docs.influxdata.com/ (дата звернення: 15.09.2025).

46. Docs. *Redis*. URL: https://redis.io/docs/latest/ (дата звернення: 19.09.2025).

47. Big data: A survey. *ResearchGate*. URL: https://www.researchgate.net/publication/263480370_Big_data_A_survey (дата звернення: 19.09.2025).

48. Kleppmann M. Designing data-intensive applications: the big ideas behind reliable, scalable, and maintainable systems. Sebastopol : O'Reilly Media, Incorporated. 590 с.

49. DB-Engines Ranking of Time Series DBMS. *DB-Engines*. URL: https://db-engines.com/en/ranking/time+series+dbms (дата звернення: 19.09.2025).

50. Database tuning: principles, experiments, and troubleshooting techniques. *ResearchGate*. URL: https://www.researchgate.net/publication/221213697_Database_tuning_principles_experiments_and_troubleshooting_techniques_part_II (дата звернення: 19.09.2025).

51. CAP Twelve years later: How the «Rules» have Changed. *ResearchGate*. URL: https://www.researchgate.net/publication/220476881_CAP_Twelve_years_later_How_the_Rules_have_Changed (дата звернення: 19.09.2025).

52. Database Security – Concepts, Approaches. *ResearchGate*. URL: https://www.researchgate.net/publication/3449351_Database_Security_-_Concepts_Approaches (дата звернення: 19.09.2025).

53. What's Really New with NewSQL?. *Carnegie Mellon Database Group.* URL: https://db.cs.cmu.edu/papers/2016/pavlo-newsql-sigmodrec2016.pdf (дата звернення: 19.09.2025).

54. Tanenbaum A. S. Distributed Systems: Principles and Paradigms. New Jersey : Pearson Education, Incorporated. 702 с.

55. A View of Cloud Computing. *ResearchGate*. URL: https://www.researchgate.net/publication/220422375_A_View_of_Cloud_Computing (дата звернення: 21.09.2025).

56. Navathe S. B., Elmasri R. Fundamentals of Database Systems. New Jersey : Pearson Education, Incorporated. 1242 с.

57. Connolly T. M., Begg C. E. Database Systems: A Practical Approach to Design, Implementation, and Management. 6-те вид. Harlow : Pearson Education, Limited. 1442 с.

58. Ramakrishnan R. Database management systems. New York : McGraw-Hill. 726 с.

59. Sadalage P., Fowler M. NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. New Jersey : Pearson Education, Incorporated. 226 с.

60. Eventually Consistent. *ResearchGate*. URL: https://www.researchgate.net/publication/27301367_Eventually_Consistent (дата звернення: 23.09.2025).

61. ESP-32-S2 Technical Reference Manual. *Espressif Documentation.* URL: https://documentation.espressif.com/esp32-s2_technical_reference_manual_en.html (дата звернення: 23.09.2025).

62. BME-680 Low power gas, pressure, temperature & humidity sensor. Datasheet. *Bosch Sensortec*. URL: https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bme680-ds001.pdf (дата звернення: 23.09.2025).

63. ASP.NET Core Best Practices. *Microsoft Learn*. URL: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/best-practices?view=aspnetcore-9.0 (дата звернення: 23.09.2025).

64. Blazor for ASP.NET Web Forms Developers. *Microsoft Learn*. URL: https://learn.microsoft.com/en-us/dotnet/architecture/blazor-for-web-forms-developers/ (дата звернення: 23.09.2025).