Міністерство освіти і науки України
Український державний університет науки і технологій

Факультет «Комп'ютерні технології і системи»

Кафедра «Комп'ютерні інформаційні технології»

Пояснювальна записка

до кваліфікаційної роботи
ОС магістр

на тему: Дослідження спеціалізованих баз даних часових рядів
за освітньою програмою: Інженерія програмного забезпечення
зі спеціальності: 121 Інженерія програмного забезпечення

Виконав: студент групи: ПЗ2421
/ Михайло ВЄТЛУЖСЬКИХ /

Керівник: / Віктор ШИНКАРЕНКО /

Нормоконтролер: / Світлана ВОЛКОВА /

Засвідчую, що у цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент

Дніпро – 2026 рік

Ministry of Education and Science of Ukraine
Ukrainian State University of Science and Technologies

«Computer technologies and systems»

«Computer information technology»

Explanatory Note
to the Thesis for
Master's Degree

on the topic: Research on specialized time series databases according to educational curriculum: Software engineering

in the Speciality: 121 Software engineering

Done by the student of the group: PZ2421 / Mykhailo VIETLUZHSKYKH /

Scientific Supervisor: / Victor SCHINKARENKO /

Normative controller : / Svitlana VOLKOVA /

Dnipro – 2026

Міністерство освіти і науки України
Український державний університет науки і технологій

Факультет: Комп'ютерні технології і системи
Кафедра: Комп'ютерні інформаційні технолог
Рівень вищої освіти: магістр
Освітня програма: 121 Інженерія програмного забезпечення
Спеціальність: 121 Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ

Завідувач кафедри КІТ
__________________ /Вадим ГОРЯЧКІН/

Дата ________________

ЗАВДАННЯ

на кваліфікаційну роботу магістра

студенту Вєтлужських Михайлу Вячеславовичу

1. Тема роботи: Дослідження спеціалізованих баз даних часових рядів

Керівник роботи: Шинкаренко Віктор Іванович, Доктор технічних наук, Професор

затверджені наказом від 01.10.2025 р. № 1402 ст

2. Строк подання студентом роботи: 10.01.2026 р.
3. Вихідні дані до роботи: наукові публікації та технічна документація щодо архітектури та функціональних можливостей баз даних часових рядів.
4. Зміст пояснювальної записки (перелік питань, які потрібно опрацювати):

4.1 Аналітична частина: аналіз архітектурних підходів до реалізації TSDB; порівняльний аналіз функціональних можливостей досліджуваних систем; огляд методів вимірювання продуктивності баз даних.
4.2 Основна частина: проєктування та реалізація системи тестування продуктивності; проведення експериментального дослідження; аналіз результатів та формування рекомендацій

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень): презентація, відео роботи програми

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз предметної області баз даних часових рядів, огляд літературних джерел	01.11.2024 – 31.12.2024	
2	Розробка методики дослідження продуктивності TSDB	01.01.2025 – 28.02.2025	30%
3	Проєктування системи тестування продуктивності	01.03.2025 – 30.04.2025	
4	Реалізація системи тестування та проведення експериментів	01.05.2025 – 31.08.2025	60%
5	Аналіз результатів експериментів та формування рекомендацій	01.09.2025 – 30.11.2025	
6	Оформлення пояснювальної записки та демонстраційних матеріалів	01.12.2025 – 05.01.2026	100%
7	Подання кваліфікаційної роботи до кафедри	10.01.2026	
8	Захист кваліфікаційної роботи на засіданні Екзаменаційної комісії	22.01.2026	

Студент ______________ Михайло ВЄТЛУЖСЬКИХ

Керівник роботи ______________ Віктор ШИНКАРЕНКО

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи: 141 с., 5 рис., 23 табл., 87 джерел, 5 додатків.

Об'єктом дослідження є спеціалізовані бази даних часових рядів (Time Series Databases, TSDB), які використовуються для зберігання, обробки та аналізу даних, організованих за часовими мітками.

Предметом дослідження є продуктивність та ефективність використання ресурсів сучасними TSDB при виконанні типових операцій запису, читання та агрегації даних.

Метою роботи є розробка науково обґрунтованої методики порівняльного аналізу продуктивності баз даних часових рядів та формування практичних рекомендацій щодо вибору оптимальної TSDB залежно від сценарію використання.

Методи дослідження: аналіз архітектурних особливостей TSDB (колоночне зберігання, LSM-дерево, розширення реляційних СУБД); benchmark-тестування з використанням Docker-контейнеризації для забезпечення ізольованого та відтворюваного тестового середовища; статистична обробка результатів із застосуванням описової статистики та перцентильного аналізу; порівняльний аналіз із рейтинговим оцінюванням.

Результати та їх новизна: спроєктовано та реалізовано модульну систему тестування продуктивності TSDB мовою Python з веб-інтерфейсом візуалізації результатів. Система включає уніфіковані адаптери для взаємодії з різними базами даних та генератор тестових даних. Проведено експериментальне дослідження продуктивності чотирьох провідних баз даних часових рядів (InfluxDB, TimescaleDB, QuestDB, VictoriaMetrics) на наборі з 10 мільйонів точок даних. Встановлено, що QuestDB демонструє найвищу швидкість запису (520 000 точок/с), що в 3,7 рази перевищує показники TimescaleDB, та найкращі результати читання із середнім часом виконання запитів 52 мс. VictoriaMetrics забезпечує найефективніше стиснення даних (коефіцієнт 16,7) та найнижче споживання оперативної пам'яті (пік 485 МБ). TimescaleDB забезпечує повну SQL-сумісність

та ACID-транзакції за рахунок нижчої продуктивності запису. Сформовано матрицю вибору TSDB залежно від сценарію використання: QuestDB рекомендовано для високошвидкісного IoT та телеметрії; VictoriaMetrics — для Prometheus-моніторингу та довгострокового зберігання метрик; TimescaleDB — для SQL-аналітики та інтеграції з існуючими системами; InfluxDB — як універсальне рішення для DevOps з розвиненою екосистемою.

Практична цінність роботи полягає у створенні відтворюваного інструментарію для об'єктивного порівняння продуктивності баз даних часових рядів та формуванні науково обґрунтованих рекомендацій для вибору оптимальної TSDB залежно від конкретних вимог проєкту.

Ключові слова: БАЗИ ДАНИХ ЧАСОВИХ РЯДІВ, TSDB, INFLUXDB, TIMESCALEDB, QUESTDB, VICTORIAMETRICS, BENCHMARK-ТЕСТУВАННЯ, ПРОДУКТИВНІСТЬ, ЧАСОВІ РЯДИ, КОЛОНОЧНЕ ЗБЕРІГАННЯ, DOCKER, IOT, МОНІТОРИНГ

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ 12

ВСТУП........ 15

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ БАЗ ДАНИХ ЧАСОВИХ РЯДІВ........ 18

1.1 Поняття та особливості баз даних часових рядів 18

1.2 Архітектурні підходи до реалізації TSDB 20

1.2.1 Колоночне зберігання (Columnar Storage) 20

1.2.2 Log-Structured Merge-Tree (LSM-дерево) 21

1.2.3 Розширення реляційних СУБД 22

1.2.4 Спеціалізовані формати зберігання........ 22

1.3 Огляд існуючих рішень баз даних часових рядів 23

1.3.1 InfluxDB як платформа з архітектурою TSM-сховища та функціональною мовою запитів Flux........ 23

1.3.2 TimescaleDB як розширення PostgreSQL із гіпертаблицями та безперервними агрегатами 25

1.3.3 QuestDB як система колоночного зберігання з SIMD-оптимізацією та memory-mapped файлами........ 26

1.3.4 VictoriaMetrics як високопродуктивне сховище метрик із підтримкою PromQL та MetricsQL 28

1.4 Порівняльний аналіз функціональних можливостей 30

1.4.1 Порівняння функціональних можливостей TSDB 30

1.4.2 Порівняння за сценаріями використання 30

1.5 Обґрунтування вибору об'єктів дослідження........ 31

1.6 Висновки до розділу 1 32

2 МЕТОДИКА ДОСЛІДЖЕННЯ ПРОДУКТИВНОСТІ БАЗ ДАНИХ ЧАСОВИХ РЯДІВ 34

2.1 Обґрунтування вибору напряму дослідження 34

2.2 Методи вимірювання продуктивності баз даних 35

2.2.1 Вимірювання часу виконання операцій 35

2.2.2 Вимірювання пропускної здатності 35

2.2.3 Вимірювання затримки 35

2.3 Методика benchmark-тестування 36

2.3.1 Принципи побудови benchmark-тестів 36

2.3.2 Типи тестів продуктивності 36

2.3.3 Структура тестового сценарію 37

2.4 Критерії оцінювання продуктивності 38

2.4.1 Критерії продуктивності запису 38

2.4.2 Критерії продуктивності читання 38

2.4.3 Критерії ефективності ресурсів 38

2.5 Методи статистичної обробки результатів 39

2.5.1 Описова статистика 39

2.5.2 Перцентилі 39

2.5.3 Статистична значущість 39

2.6 Оцінка похибок вимірювань 40

2.6.1 Систематичні похибки 40

2.6.2 Випадкові похибки 40

2.6.3 Довірчі інтервали 41

2.7 Висновки до розділу 2 41

3 ПРОЄКТУВАННЯ СИСТЕМИ ТЕСТУВАННЯ ПРОДУКТИВНОСТІ 42

3.1 Методика проведення benchmark-тестування 42
3.1.1 Тестування продуктивності запису даних 44
3.1.2 Тестування продуктивності читання даних 44
3.1.3 Тестування використання ресурсів 45
3.1.4 Тестування ефективності стиснення 45
3.2 Архітектура системи тестування 46
3.2.1 Менеджер Docker-контейнерів (DockerManager) 47
3.2.2 Генератор тестових даних (DataGenerator) 48
3.2.3 Модулі взаємодії з базами даних (DatabaseAdapter) 49
3.2.4 Виконавець тестів (BenchmarkRunner) 50
3.2.5 Система візуалізації (WebVisualization) 51
3.3 Проектування модулів взаємодії з базами даних 52
3.3.1 Адаптер InfluxDB для реалізації запису через Line Protocol та HTTP API 52
3.3.2 Адаптер TimescaleDB для взаємодії через PostgreSQL-протокол та команду COPY 53
3.3.3 Адаптер QuestDB для інтеграції через InfluxDB Line Protocol та REST API 54
3.3.4 Адаптер VictoriaMetrics для імпорту даних через Prometheus Remote Write API 55
3.4 Система генерації тестових даних 56
3.4.1 Модель даних 57
3.4.2 Алгоритм генерації 57
3.4.3 Параметри генерації 59
3.4.4 Генерація тегів 60

3.5 Проектування веб-інтерфейсу візуалізації ... 60
3.5.1 Технологічний стек ... 61
3.5.2 Архітектура веб-додатку ... 61
3.5.3 Компоненти інтерфейсу ... 62
3.5.4 Динамічне оновлення ... 62
3.5.5 API endpoints ... 63
3.6 Висновки до розділу 3 ... 63
4 РЕАЛІЗАЦІЯ ТА РЕЗУЛЬТАТИ ЕКСПЕРИМЕНТІВ ... 65
4.1 Реалізація системи тестування ... 65
4.1.1 Структура проекту ... 65
4.1.2 Залежності проекту ... 66
4.1.3 Реалізація адаптерів баз даних ... 67
4.2 Методика проведення експериментів ... 67
4.2.1 Тестове середовище ... 67
4.2.2 Версії баз даних ... 68
4.2.3 Параметри тестових даних ... 68
4.2.4 Процедура тестування ... 68
4.2.5 Статистична обробка результатів ... 69
4.3 Результати тестування продуктивності запису ... 70
4.4 Результати тестування продуктивності читання ... 72
4.4.1 Проста вибірка (Simple Select) ... 72
4.4.2 Агрегація за часовим вікном (Time Window Aggregation) ... 73
4.4.3 Запит до множинних серій (Multi-series Query) ... 73
4.4.4 Фільтрація за значенням (Value Filter) ... 74
4.4.5 Зведені результати тестування читання ... 74

4.5 Аналіз використання системних ресурсів 76
4.5.1 Використання оперативної пам'яті 76
4.5.2 Використання дискового простору та ефективність стиснення ... 77
4.5.3 Використання CPU 78
4.6 Порівняльний аналіз та рекомендації 78
4.6.1 Зведена таблиця результатів 79
4.6.2 Рейтингова оцінка 79
4.6.3 Рекомендації щодо вибору TSDB 80
4.6.4 Матриця вибору за сценаріями 81
4.7 Висновки до розділу 4 81
ЗАГАЛЬНІ ВИСНОВКИ 83
ПЕРЕЛІК ПОСИЛАННЬ 85
ДОДАТОК А 94
ДОДАТОК Б 100
ДОДАТОК В 109
ДОДАТОК Г 112
ДОДАТОК Д 121

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

Найменування	**Значення**
TSDB	Time Series Database (база даних часових рядів) – спеціалізована система управління базами даних, оптимізована для зберігання, обробки та аналізу даних, організованих за часовими мітками
SQL	Structured Query Language (мова структурованих запитів) – декларативна мова програмування для управління даними в реляційних базах даних
API	Application Programming Interface (програмний інтерфейс застосунку) – набір визначених методів та протоколів для взаємодії між програмними компонентами
HTTP	HyperText Transfer Protocol (протокол передачі гіпертексту) – протокол прикладного рівня для передачі даних у розподілених інформаційних системах
TCP	Transmission Control Protocol (протокол керування передачею) – транспортний протокол, що забезпечує надійну передачу даних між вузлами мережі
REST	Representational State Transfer (передача репрезентативного стану) – архітектурний стиль взаємодії компонентів розподіленого застосунку в мережі
JSON	JavaScript Object Notation (об'єктна нотація JavaScript) – текстовий формат обміну даними, що базується на підмножині мови програмування JavaScript
YAML	YAML Ain't Markup Language – людиночитаний формат серіалізації даних, який часто використовується для конфігураційних файлів

LSM	Log-Structured Merge-Tree (дерево злиття з журнальною структурою) – структура даних з послідовним записом, оптимізована для робочих навантажень з переважанням операцій запису
TSM	Time-Structured Merge Tree (дерево злиття з часовою структурою) – власний формат зберігання даних InfluxDB, що поєднує переваги колоночного зберігання та LSM-структури
SIMD	Single Instruction Multiple Data (одна інструкція, множинні дані) – архітектура паралельних обчислень, що дозволяє виконувати одну операцію над множиною даних одночасно
ACID	Atomicity, Consistency, Isolation, Durability (атомарність, узгодженість, ізольованість, довговічність) – набір властивостей, що гарантують надійну обробку транзакцій у базах даних
IoT	Internet of Things (Інтернет речей) – концепція мережі фізичних пристроїв, оснащених датчиками та програмним забезпеченням для обміну даними
CPU	Central Processing Unit (центральний процесор) – основний обчислювальний компонент комп'ютера, що виконує машинні інструкції
RAM	Random Access Memory (оперативна пам'ять) – енергозалежна пам'ять комп'ютера для тимчасового зберігання даних під час виконання програм
SSD	Solid State Drive (твердотільний накопичувач) – пристрій зберігання даних на основі флеш-пам'яті без рухомих механічних частин
PromQL	Prometheus Query Language (мова запитів Prometheus) – функціональна мова запитів для вибірки та агрегації метрик у системі моніторингу Prometheus
MetricsQL	Розширена версія PromQL, що використовується у VictoriaMetrics з додатковими функціями для аналізу часових рядів

Flux	Функціональна мова запитів та скриптів для роботи з даними в InfluxDB, що підтримує трансформацію, агрегацію та аналіз часових рядів
MVC	Model-View-Controller (модель-представлення-контролер) – архітектурний патерн програмного забезпечення, що розділяє логіку застосунку на три взаємопов'язані компоненти
WAL	Write-Ahead Log (журнал попереднього запису) – метод забезпечення атомарності та довговічності даних шляхом запису змін до журналу перед їх застосуванням
GC	Garbage Collection (збирання сміття) – автоматичне управління пам'яттю шляхом звільнення областей пам'яті, які більше не використовуються програмою
RSS	Resident Set Size (розмір резидентного набору) – обсяг оперативної пам'яті, яку процес фактично використовує в даний момент часу
I/O	Input/Output (введення/виведення) – операції обміну даними між процесором та зовнішніми пристроями або носіями інформації
СУБД	Система управління базами даних – програмне забезпечення для створення, зберігання, модифікації та вилучення інформації з баз даних
РСУБД	Реляційна система управління базами даних – СУБД, що базується на реляційній моделі даних та використовує таблиці для організації інформації
Docker	Платформа контейнеризації, що дозволяє упаковувати застосунки з усіма залежностями в ізольовані контейнери для розгортання
PostgreSQL	Об'єктно-реляційна система управління базами даних з відкритим вихідним кодом, що підтримує розширення та повну SQL-сумісність

ВСТУП

У сучасних умовах стрімкого розвитку цифрових технологій, Інтернету речей (IoT), хмарних обчислень, фінансових технологій, промислової автоматизації та систем моніторингу зростає обсяг даних, що мають часову природу. Такі дані формуються у вигляді послідовностей значень, зафіксованих у певні моменти часу, і називаються часовими рядами. Вони широко застосовуються у сфері телеметрії, фінансової аналітики, прогнозування, машинного навчання, моніторингу інфраструктури, робототехніки та кіберфізичних систем.

Традиційні реляційні бази даних, незважаючи на свою універсальність, не завжди ефективно забезпечують зберігання, обробку та аналіз великих обсягів часових рядів. Проблеми масштабованості, продуктивності при роботі з потоковими даними, складність агрегування за часовими інтервалами та обробки високочастотних записів зумовили появу спеціалізованих баз даних часових рядів (Time Series Databases, TSDB), таких як InfluxDB, TimescaleDB, Prometheus, OpenTSDB та інші.

Актуальність дослідження спеціалізованих баз даних часових рядів полягає у необхідності обґрунтованого вибору технологій для побудови ефективних програмних систем, що працюють з потоками даних у реальному або квазіреальному часі. В умовах розвитку програмно-апаратних комплексів, зокрема в робототехніці, «розумних» містах, промисловому Інтернеті речей та системах кібербезпеки, правильна організація зберігання та аналізу часових даних стає критично важливою для надійності, масштабованості та продуктивності програмних рішень.

Незважаючи на наявність значної кількості готових рішень, питання порівняльного аналізу архітектур, моделей зберігання, механізмів індексації, стиснення, агрегації та обробки часових рядів залишається недостатньо систематизованим у навчальній та прикладній практиці програмної інженерії.

Більшість публікацій мають оглядовий або рекламний характер і не завжди враховують вимоги реальних програмних систем.

Тому дослідження спеціалізованих баз даних часових рядів є доцільним і необхідним для розвитку програмної інженерії, створення високопродуктивних програмно-апаратних комплексів та формування нових підходів до обробки потокових даних. Робота відрізняється від відомих рішень комплексним підходом до аналізу TSDB з позицій програмної інженерії, з урахуванням архітектурних, алгоритмічних і прикладних аспектів.

Тема магістерської роботи пов'язана з науково-дослідними напрямами у галузі розподілених систем, обробки великих даних, Інтернету речей та кіберфізичних систем, а також відповідає сучасним державним та галузевим програмам цифрової трансформації, розвитку індустрії інформаційних технологій.

Об'єкт дослідження – процеси зберігання та обробки даних у спеціалізованих базах даних часових рядів.

Предмет дослідження – показники продуктивності спеціалізованих баз даних часових рядів.

Метою роботи є визначення оптимальних сценаріїв застосування спеціалізованих баз даних часових на основі порівняльного аналізу їх продуктивності для підвищення ефективності систем збору та обробки часових даних.

Для досягнення поставленої мети необхідно розв'язати такі завдання:

- проаналізувати особливості часових даних та вимоги до їх зберігання;
- дослідити архітектурні принципи побудови TSDB;
- виконати порівняльний аналіз популярних баз даних часових рядів;
- оцінити їх продуктивність, масштабованість та зручність використання;
- визначити сфери доцільного застосування різних типів TSDB;

– розробити рекомендації щодо вибору TSDB для прикладних програмних систем.

У роботі використовуються такі методи дослідження:

– аналіз і синтез науково-технічних джерел;
– порівняльний аналіз;
– моделювання архітектурних рішень;
– експериментальні дослідження продуктивності;
– статистична обробка результатів експериментів.

Наукова новизна полягає у систематизованому підході до аналізу спеціалізованих баз даних часових рядів з позицій програмної інженерії, а також у розробці критеріїв вибору TSDB залежно від типу програмної системи та характеру часових даних.

Практичне значення роботи полягає у можливості використання отриманих результатів при проєктуванні програмних систем для моніторингу, аналітики, робототехніки, IoT та фінансових технологій. Результати можуть бути використані у навчальному процесі при викладанні дисциплін з баз даних, розподілених систем та програмної інженерії.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ БАЗ ДАНИХ ЧАСОВИХ РЯДІВ

1.1 Поняття та особливості баз даних часових рядів

Часовий ряд (time series) – це послідовність точок даних, індексованих у часовому порядку. За визначенням Jensen S.K. та співавторів, «часовий ряд являє собою впорядковану послідовність пар (timestamp, value), де кожне значення асоціюється з конкретним моментом часу» [1]. Найчастіше часовий ряд представляє собою послідовність вимірювань, що здійснюються через рівні або нерівні проміжки часу.

Дані часових рядів широко використовуються в різних галузях людської діяльності. За даними дослідження Gartner, «до 2025 року понад 75% корпоративних даних будуть генеруватися та оброблятися поза традиційними центрами обробки даних, значна частина яких матиме часову природу» [2]. До основних сфер застосування часових рядів належать:

– фінансова аналітика – відстеження котирувань акцій, валютних курсів, криптовалют;
– моніторинг інфраструктури – збір метрик серверів, мережевого обладнання, контейнерів;
– інтернет речей (IoT) – дані з датчиків температури, вологості, руху;
– промислова автоматизація – контроль виробничих процесів;
– медицина – моніторинг життєвих показників пацієнтів;
– метеорологія – запис погодних умов.

База даних часових рядів (Time Series Database, TSDB) – це програмна система, спеціально оптимізована для зберігання, обробки та аналізу даних, організованих за часовими мітками. Як зазначає Dunning T., «TSDB відрізняються від традиційних реляційних баз даних тим, що вони оптимізовані для специфічних патернів роботи з даними: переважно операції вставки нових записів та читання за часовими діапазонами, з мінімальною кількістю оновлень існуючих даних» [3].

Основні характеристики, що відрізняють TSDB від традиційних СУБД:

Висока швидкість запису. Системи моніторингу та IoT-платформи генерують мільйони точок даних за секунду. За даними InfluxData, «сучасні TSDB повинні підтримувати швидкість запису від сотень тисяч до мільйонів точок на секунду на одному вузлі» [4]. Це досягається завдяки оптимізованим структурам даних та мінімізації накладних витрат на транзакції.

Ефективне стиснення даних. Дані часових рядів мають специфічні властивості, які дозволяють застосовувати спеціалізовані алгоритми стиснення. Pelkonen T. та співавтори з Facebook описують алгоритм Gorilla, який «досягає коефіцієнту стиснення до 12x для типових метрик моніторингу завдяки використанню delta-of-delta кодування для часових міток та XOR-кодування для значень» [5].

Оптимізовані операції читання за часовими діапазонами. Типові запити до TSDB включають вибірку даних за певний період часу. За словами творців TimescaleDB, «традиційні B-tree індекси неефективні для часових запитів через необхідність сканування великої кількості сторінок; натомість TSDB використовують партиціонування за часом та спеціалізовані індекси» [6].

Вбудовані агрегаційні функції. TSDB надають оптимізовані функції для обчислення середніх значень, мінімумів, максимумів, перцентилів за часовими вікнами. Згідно з документацією QuestDB, «агрегація за часовими інтервалами (downsampling) є однією з найчастіших операцій, тому вона реалізована на рівні storage engine для максимальної продуктивності» [7].

Автоматичне управління життєвим циклом даних. Більшість TSDB підтримують політики збереження (retention policies), які автоматично видаляють або агрегують застарілі дані. Як зазначається в документації InfluxDB, «retention policies дозволяють автоматично видаляти дані старші за заданий період, що критично важливо для управління зростанням обсягу даних» [8].

Підтримка високої кардинальності. Кардинальність визначається як кількість унікальних комбінацій тегів (міток) у базі даних. За даними

VictoriaMetrics, «висока кардинальність є однією з найбільших проблем для TSDB, оскільки кожна унікальна комбінація тегів потребує окремого індексу; сучасні системи повинні ефективно працювати з мільйонами унікальних часових рядів» [9].

Порівняння характеристик TSDB та традиційних СУБД наведено в таблиці 1.1.

Таблиця 1.1 – Порівняння TSDB та традиційних реляційних СУБД

Характеристика	Реляційні СУБД	TSDB
Оптимізація	CRUD-операції	Запис та читання за часом
Типовий запит	JOIN між таблицями	Запис та читання за часом
Стиснення	Загальні алгоритми	Спеціалізовані для часових рядів
Індексування	B-tree	Часове партиціонування
Оновлення даних	Часті	Рідкісні або відсутні
Видалення даних	Вибіркове	За retention policy

1.2 Архітектурні підходи до реалізації TSDB

Існує кілька основних архітектурних підходів до побудови баз даних часових рядів, кожен з яких має свої переваги та обмеження. Вибір архітектури суттєво впливає на продуктивність системи для різних сценаріїв використання.

1.2.1 Колоночне зберігання (Columnar Storage)

Колоночне зберігання є одним з найпоширеніших підходів у TSDB. При цьому підході дані кожного поля (колонки) зберігаються окремо від інших полів. За визначенням Abadi D., «колоночні бази даних зберігають значення кожного атрибуту послідовно, що дозволяє ефективно стискати однотипні значення та швидко виконувати агрегаційні операції над окремими колонками» [10].

Переваги колоночного зберігання для часових рядів:

- високий коефіцієнт стиснення завдяки однорідності даних у колонці;
- ефективне читання окремих метрик без завантаження всього рядка;
- оптимізація для аналітичних запитів з агрегацією.

QuestDB використовує колоночне зберігання з технологією memory-mapped файлів. Згідно з технічною документацією, «кожна колонка зберігається в окремому файлі, що дозволяє операційній системі ефективно кешувати часто використовувані дані та забезпечує швидкість читання близьку до швидкості оперативної пам'яті» [11].

1.2.2 Log-Structured Merge-Tree (LSM-дерево)

LSM-дерево – це структура даних, оптимізована для робочих навантажень з переважанням операцій запису. Як описують O'Neil P. та співавтори в оригінальній роботі, «LSM-дерево буферизує записи в пам'яті (memtable), а потім періодично скидає їх на диск у відсортованому вигляді (SSTable), що мінімізує випадкові операції запису на диск» [12].

Процес роботи LSM-дерева:

а) Нові дані записуються в буфер пам'яті (memtable).

б) Коли буфер заповнюється, він скидається на диск як незмінний файл (SSTable).

в) Фоновий процес компактифікації об'єднує SSTable-файли для оптимізації читання.

VictoriaMetrics використовує модифіковане LSM-дерево. За словами розробників, «наша реалізація оптимізована для метрик моніторингу та досягає швидкості запису понад 1 мільйон точок на секунду на звичайному серверному обладнанні» [13].

1.2.3 Розширення реляційних СУБД

Альтернативний підхід полягає у розширенні існуючих реляційних баз даних спеціалізованими можливостями для роботи з часовими рядами. TimescaleDB є найяскравішим прикладом цього підходу – це розширення PostgreSQL, яке додає оптимізації для часових даних, зберігаючи повну SQL-сумісність.

Як зазначають творці TimescaleDB Freedman M. та співавтори, «гіпертаблиці (hypertables) автоматично партиціонують дані за часом на чанки (chunks), що забезпечує ефективне зберігання та запити до великих обсягів часових даних, зберігаючи при цьому всі переваги PostgreSQL: ACID-транзакції, повнотекстовий пошук, геопросторові дані та JOIN-операції» [14].

Переваги підходу на базі реляційних СУБД:

- повна SQL-сумісність та можливість використання існуючих інструментів;
- підтримка складних запитів з JOIN між часовими та звичайними таблицями;
- зріла екосистема резервного копіювання, реплікації та моніторингу;
- можливість поступової міграції існуючих систем.

1.2.4 Спеціалізовані формати зберігання

Деякі TSDB розробляють власні формати зберігання, оптимізовані під конкретні сценарії використання. InfluxDB використовує формат TSM (Time-Structured Merge Tree), який, за описом розробників, «поєднує переваги колоночного зберігання для ефективного стиснення та LSM-подібної структури для високої швидкості запису» [15].

Структура TSM включає:

- WAL (Write-Ahead Log) для забезпечення довговічності даних;
- Cache в пам'яті для буферизації записів;
- TSM-файли на диску з колоночною організацією та індексами.

Prometheus, хоча і не є повноцінною TSDB, запропонував власний формат зберігання, який став стандартом для систем моніторингу. За описом Beyer B., «Prometheus використовує двохрівневу структуру: дані за останні 2 години зберігаються в пам'яті для швидкого доступу, а старіші дані компактифікуються в блоки на диску» [16].

Порівняння архітектурних підходів наведено в таблиці 1.2.

Таблиця 1.2 – Порівняння архітектурних підходів до реалізації TSDB

Підхід	Приклади	Переваги	Недоліки
Колоночне зберігання	QuestDB, ClickHouse	Висока швидкість аналітичних запитів, ефективне стиснення	Повільніші точкові запити
LSM-дерево	VictoriaMetrics, Cassandra	Дуже висока швидкість запису	Складність компактифікації
Розширення РСУБД	TimescaleDB	SQL-сумісність, зріла екосистема	Обмеження базової СУБД
Спеціалізовані формати	InfluxDB (TSM)	Оптимізація під конкретні задачі	Vendor lock-in

1.3 Огляд існуючих рішень баз даних часових рядів

За даними рейтингу DB-Engines станом на 2024 рік, найпопулярнішими базами даних часових рядів є InfluxDB, TimescaleDB, Prometheus, QuestDB та VictoriaMetrics [17]. Для даного дослідження обрано чотири системи, які представляють різні архітектурні підходи та мають активну спільноту розробників.

1.3.1 InfluxDB як платформа з архітектурою TSM-сховища та функціональною мовою запитів Flux

InfluxDB – це база даних часових рядів з відкритим вихідним кодом, розроблена компанією InfluxData. Перший реліз відбувся у 2013 році, і з того часу система стала однією з найпопулярніших TSDB у світі. За даними компанії, «InfluxDB використовується більш ніж 500 000 розробниками та тисячами компаній, включаючи IBM, Siemens, Tesla та Cisco» [18].

Ключові особливості InfluxDB:

Мова запитів Flux. Починаючи з версії 2.0, InfluxDB використовує функціональну мову запитів Flux, яка надає потужні можливості для аналізу та трансформації даних. За описом документації, «Flux дозволяє виконувати складні аналітичні операції, включаючи часові з'єднання, скриптинг та інтеграцію з зовнішніми сервісами, що неможливо реалізувати в традиційному SQL» [19]. Приклад запиту на Flux наведено у лістингу 1.1.

```
from(bucket: "metrics")
  |> range(start: -1h)
  |> filter(fn: (r) => r._measurement == "cpu")
  |> aggregateWindow(every: 5m, fn: mean)
```

Лістинг 1.1 – Приклад запиту на Flux

Формат зберігання TSM. InfluxDB використовує власний формат Time-Structured Merge Tree, оптимізований для часових рядів. Згідно з технічною документацією, «TSM забезпечує коефіцієнт стиснення до 10x порівняно з вихідними даними та підтримує швидкість запису понад 300 000 точок на секунду на одному вузлі» [20].

Екосистема TICK Stack. InfluxDB є частиною інтегрованої платформи:

- Telegraf – агент для збору метрик з понад 200 джерел;
- InfluxDB – база даних;
- Chronograf – веб-інтерфейс для візуалізації;
- Kapacitor – обробка потокових даних та алертинг.

Модель даних. InfluxDB використовує концепцію measurement (вимірювання), tags (теги для індексації) та fields (поля з даними). Як зазначається в документації, «теги індексуються та оптимізовані для фільтрації, тоді як поля не індексуються та призначені для зберігання числових значень» [21].

Обмеження InfluxDB

До обмежень InfluxDB відносять наступні пункти.

– Висока кардинальність може суттєво впливати на продуктивність.
– Кластеризація доступна лише в комерційній версії.
– Flux має криву навчання для розробників, звичних до SQL.

1.3.2 TimescaleDB як розширення PostgreSQL із гіпертаблицями та безперервними агрегатами

TimescaleDB – це розширення PostgreSQL для роботи з даними часових рядів, випущене у 2017 році компанією Timescale, Inc. За словами співзасновника Freedman M., «ми створили TimescaleDB, тому що побачили, що розробники хочуть використовувати SQL для аналізу часових даних, але існуючі рішення не масштабувалися для великих обсягів» [22].

Ключові особливості TimescaleDB:

Гіпертаблиці (Hypertables). Основна інновація TimescaleDB – автоматичне партиціонування даних за часом. Згідно з документацією, «гіпертаблиця виглядає та поводиться як звичайна PostgreSQL таблиця, але внутрішньо автоматично розбивається на чанки за часовими інтервалами, що забезпечує ефективне зберігання та запити» [23].

Створення гіпертаблиці:

```
CREATE TABLE metrics (
    time TIMESTAMPTZ NOT NULL,
    device_id TEXT,
    temperature DOUBLE PRECISION
);
SELECT create_hypertable('metrics', 'time');
```

Лістинг 1.2 – Приклад створення гіпертаблиці для зберігання метрик

Повна SQL-сумісність. TimescaleDB підтримує всі можливості PostgreSQL, включаючи:

– JOIN-операції між часовими рядами та звичайними таблицями;
– повнотекстовий пошук;
– геопросторові дані через PostGIS;

– JSON/JSONB для напівструктурованих даних;
– матеріалізовані представлення.

Безперервні агрегації (Continuous Aggregates). Для прискорення типових запитів TimescaleDB підтримує автоматично оновлювані агрегації. Як описано в документації, «continuous aggregates автоматично обчислюють та зберігають результати агрегаційних запитів, оновлюючи їх при надходженні нових даних, що може прискорити запити в 100-1000 разів» [24].

Стиснення даних. Починаючи з версії 1.5, TimescaleDB підтримує нативне стиснення. За даними тестів компанії, «стиснення зменшує розмір даних на 90-95% та прискорює аналітичні запити до 20 разів завдяки колоночному формату зберігання стиснених чанків» [25].

Інтеграція з екосистемою PostgreSQL. TimescaleDB автоматично сумісна з:

– усіма клієнтськими бібліотеками PostgreSQL;
– інструментами резервного копіювання (pg_dump, pgBackRest);
– системами реплікації;
– інструментами моніторингу (pgAdmin, pgBadger).

Обмеження TimescaleDB

До обмежень TimescaleDB відносять наступні пункти.

– Продуктивність запису нижча, ніж у спеціалізованих TSDB.
– Деякі функції (мультивузлова кластеризація) доступні лише в комерційній версії.
– Потребує більше ресурсів через накладні витрати PostgreSQL.

1.3.3 QuestDB як система колоночного зберігання з SIMD-оптимізацією та memory-mapped файлами

QuestDB – це високопродуктивна база даних часових рядів, написана на Java та C++, яка фокусується на максимальній швидкості виконання операцій. Проект був заснований у 2014 році, перший публічний реліз відбувся у 2019

році. За заявою розробників, «QuestDB може обробляти до 4 мільйонів рядків на секунду при запису, що робить її однією з найшвидших TSDB у світі» [26].

Ключові особливості QuestDB:

Колоночне зберігання з memory-mapped файлами. QuestDB використовує колоночну архітектуру з прямим відображенням файлів у пам'ять. Згідно з технічною документацією, «кожна колонка зберігається в окремому файлі, а операційна система автоматично управляє кешуванням, що забезпечує швидкість читання близьку до швидкості RAM без необхідності вручну управляти кешем» [27].

Підтримка SQL з розширеннями. QuestDB підтримує стандартний SQL з розширеннями для роботи з часовими рядами, як наведено у лістингу 1.3:

```
SELECT timestamp, avg(temperature)
FROM sensors
WHERE timestamp IN '2024-01-01'
SAMPLE BY 1h
ALIGN TO CALENDAR;
```

Лістинг 1.3 — Запит агрегації даних з інтервалом у QuestDB

Функція SAMPLE BY є специфічною для QuestDB та дозволяє ефективно виконувати агрегацію за часовими інтервалами [28].

Сумісність з InfluxDB Line Protocol. Для спрощення міграції QuestDB підтримує протокол запису InfluxDB, що дозволяє використовувати існуючі агенти збору даних, такі як Telegraf [29].

Вбудований веб-інтерфейс. QuestDB включає веб-консоль для виконання запитів та візуалізації результатів без необхідності встановлення додаткових інструментів.

SIMD-оптимізації. Критичні операції реалізовані з використанням SIMD-інструкцій процесора. За словами головного розробника Кочкарова І., «ми використовуємо AVX2 та AVX-512 інструкції для паралельної обробки даних, що дозволяє досягти швидкості сканування понад 2 мільярди рядків на секунду на одному ядрі» [30].

Обмеження QuestDB

До обмежень QuestDB відносять наступні пункти.

- Відсутність вбудованої кластеризації (на момент дослідження).
- Менш розвинена екосистема порівняно з InfluxDB.
- Обмежена підтримка складних JOIN-операцій.

1.3.4 VictoriaMetrics як високопродуктивне сховище метрик із підтримкою PromQL та MetricsQL

VictoriaMetrics – це високопродуктивна база даних часових рядів, оптимізована для роботи з метриками моніторингу та повністю сумісна з екосистемою Prometheus. Проект був заснований у 2018 році та швидко набув популярності як заміна Prometheus для довгострокового зберігання метрик. За даними компанії, «VictoriaMetrics використовується тисячами компаній по всьому світу, включаючи Adidas, Wix та Fly.io» [31].

Ключові особливості VictoriaMetrics:

Повна сумісність з Prometheus. VictoriaMetrics підтримує наступні інтерфейси взаємодії.

- PromQL та розширену версію MetricsQL для запитів.
- Remote Write API для прийому даних від Prometheus.
- Формат експорту метрик Prometheus.
- Конфігурацію правил алертингу та recording rules.

Як зазначається в документації, «VictoriaMetrics може бути використана як drop-in заміна для Prometheus у більшості сценаріїв, забезпечуючи при цьому значно краще стиснення та нижче споживання ресурсів» [32].

Ефективне стиснення даних. VictoriaMetrics використовує власні алгоритми стиснення, оптимізовані для метрик. За результатами тестів, «середній розмір однієї точки даних становить 0.4-0.8 байта, що до 10 разів менше, ніж у Prometheus» [33].

Кластерна версія. VictoriaMetrics надає компоненти для горизонтального масштабування:

- vminsert – для розподіленого запису;
- vmselect – для розподілених запитів;
- vmstorage – для зберігання даних.

MetricsQL. Розширену версію PromQL з додатковими функціями наведено у лістингу 1.4:

```
avg_over_time(rate(http_requests_total[5m])[1h:5m])
```

Лістинг 1.4 — Приклад PromQL-запиту з вкладеними функціями агрегації

MetricsQL підтримує вкладені rollup-функції, lookbehind window та інші розширення, недоступні в стандартному PromQL [34].

Низьке споживання ресурсів. Порівняльні тести показують, що «VictoriaMetrics використовує в 5-10 разів менше RAM та в 3-7 разів менше дискового простору порівняно з Prometheus для однакового обсягу даних» [35].

Підтримка багатьох протоколів. Окрім Prometheus, VictoriaMetrics підтримує:

- InfluxDB Line Protocol;
- Graphite plaintext protocol;
- OpenTSDB HTTP API;
- DataDog agent protocol.

Обмеження VictoriaMetrics

До обмежень VictoriaMetrics відносять наступні пункти.

- Відсутність SQL-інтерфейсу.
- Менш гнучка модель даних порівняно з InfluxDB.
- Оптимізована переважно для метрик моніторингу, а не для загальних часових рядів.

1.4 Порівняльний аналіз функціональних можливостей

Для об'єктивного порівняння розглянутих баз даних часових рядів необхідно проаналізувати їх функціональні можливості за ключовими критеріями. У таблиці 1.3 наведено порівняння основних характеристик досліджуваних TSDB.

1.4.1 Порівняння функціональних можливостей TSDB

Таблиця 1.3 – Порівняння функціональних можливостей TSDB

Характеристика	InfluxDB	TimescaleDB	QuestDB	VictoriaMetrics
Мова запитів	Flux, InfluxQL	SQL	SQL	PromQL, MetricsQL
Архітектура	TSM-дерево	PostgreSQL + гіпертаблиці	Колоночна	LSM-дерево
SQL-сумісність	Часткова (InfluxQL)	Повна	Повна	Немає
Кластеризація	Так (платна)	Так (платна)	Ні	Так (безкоштовна)
Стиснення	Високе (~10x)	Середнє (~3-5x)	Високе (~8x)	Дуже високе (~15x)
Prometheus-сумісність	Часткова	Часткова	Часткова	Повна
Ліцензія	MIT/Proprietary	Apache 2.0 /Proprietary	Apache 2.0	Apache 2.0
Перша версія	2013	2017	2019	2018

1.4.2 Порівняння за сценаріями використання

Моніторинг інфраструктури. Для цього сценарію найкраще підходять VictoriaMetrics та InfluxDB завдяки розвиненій інтеграції з системами збору метрик. VictoriaMetrics має перевагу при використанні Prometheus-екосистеми,

тоді як InfluxDB краще підходить при потребі в складній аналітиці через Flux [36].

IoT та промислові дані. QuestDB та TimescaleDB є оптимальним вибором для IoT-сценаріїв. QuestDB забезпечує найвищу швидкість запису, що критично для високочастотних датчиків. TimescaleDB надає SQL-сумісність, важливу для інтеграції з існуючими аналітичними системами [37].

Фінансові дані. TimescaleDB та QuestDB найкраще підходять для фінансових застосувань завдяки підтримці SQL та можливості виконання складних аналітичних запитів. TimescaleDB додатково забезпечує ACID-транзакції, важливі для фінансових систем [38].

Довгострокове зберігання метрик. VictoriaMetrics є лідером для цього сценарію завдяки найефективнішому стисненню та найнижчому споживанню ресурсів. За даними порівняльних тестів, «VictoriaMetrics може зберігати в 10 разів більше даних на тому ж обладнанні порівняно з Prometheus» [39].

1.5 Обґрунтування вибору об'єктів дослідження

Для проведення порівняльного дослідження продуктивності було обрано чотири бази даних часових рядів: InfluxDB, TimescaleDB, QuestDB та VictoriaMetrics. Вибір саме цих систем обумовлений такими факторами:

По-перше, популярність та репрезентативність. Обрані системи є найпопулярнішими представниками своїх архітектурних категорій за рейтингом DB-Engines [17]:

- InfluxDB – лідер серед спеціалізованих TSDB (1 місце);
- TimescaleDB – лідер серед рішень на базі реляційних СУБД (2 місце);
- VictoriaMetrics – лідер для екосистеми Prometheus (5 місце);
- QuestDB – представник високопродуктивних колоночних систем (11 місце з найшвидшим ростом).

По-друге, різноманітність архітектурних підходів. Обрані системи представляють чотири різні архітектури:

- TSM-дерево (InfluxDB);
- розширення реляційної СУБД (TimescaleDB);
- колоночне зберігання (QuestDB);
- LSM-дерево (VictoriaMetrics).

Це дозволяє отримати комплексну картину можливостей сучасних TSDB та виявити переваги й недоліки кожного підходу.

По-третє, відкритий вихідний код. Всі обрані системи мають відкритий вихідний код та активну спільноту розробників. Це забезпечує прозорість архітектурних рішень, можливість детального аналізу та відтворюваність результатів дослідження.

По-четверте, підтримка контейнеризації. Всі обрані системи підтримують розгортання через Docker, що є необхідною умовою для забезпечення однакових умов тестування та відтворюваності експериментів. Офіційні Docker-образи доступні на Docker Hub:

- influxdb:2.7
- timescale/timescaledb:latest-pg15
- questdb/questdb:latest
- victoriametrics/victoria-metrics:latest

По-п'яте, практична значущість. Обрані системи широко використовуються в реальних проектах різного масштабу, що робить результати дослідження практично корисними для розробників та архітекторів систем.

1.6 Висновки до розділу 1

У першому розділі проведено аналіз предметної області баз даних часових рядів та отримано такі результати:

Визначено поняття часового ряду та особливості систем для їх зберігання. Встановлено, що TSDB відрізняються від традиційних реляційних СУБД оптимізацією для специфічних патернів роботи: переважно операції вставки та

читання за часовими діапазонами, спеціалізовані алгоритми стиснення та автоматичне управління життєвим циклом даних.

Розглянуто основні архітектурні підходи до реалізації TSDB: колоночне зберігання, LSM-дерево, розширення реляційних СУБД та спеціалізовані формати зберігання. Кожен підхід має свої переваги та обмеження для різних сценаріїв використання.

Виконано детальний огляд чотирьох провідних баз даних часових рядів: InfluxDB, TimescaleDB, QuestDB та VictoriaMetrics. Для кожної системи визначено ключові особливості, архітектурні рішення, переваги та обмеження.

Проведено порівняльний аналіз функціональних можливостей розглянутих систем за критеріями: мова запитів, архітектура, SQL-сумісність, кластеризація, стиснення, ліцензія.

Обґрунтовано вибір об'єктів дослідження на основі їх популярності, різноманітності архітектурних підходів, відкритості вихідного коду та підтримки контейнеризації.

Аналіз показав відсутність уніфікованого інструментарію для об'єктивного порівняння продуктивності різних TSDB в однакових умовах, що підтверджує актуальність розробки такого інструментарію в рамках даного дослідження.

2 МЕТОДИКА ДОСЛІДЖЕННЯ ПРОДУКТИВНОСТІ БАЗ ДАНИХ ЧАСОВИХ РЯДІВ

2.1 Обґрунтування вибору напряму дослідження

Дослідження продуктивності баз даних часових рядів є актуальним напрямом, оскільки вибір оптимальної TSDB для конкретного застосування вимагає об'єктивного порівняння характеристик різних систем. Як зазначає Gray J. у фундаментальній праці «The Benchmark Handbook for Database and Transaction Systems», «бенчмарки є єдиним надійним способом порівняння продуктивності систем баз даних, оскільки теоретичні оцінки не враховують особливостей реальних реалізацій» [40].

Обраний напрям дослідження передбачає експериментальне порівняння продуктивності чотирьох TSDB за такими аспектами:

- швидкість запису даних;
- швидкість виконання різних типів запитів;
- ефективність використання системних ресурсів;
- ефективність стиснення даних.

Експериментальний підхід обрано на основі рекомендацій Jain R., який у книзі «The Art of Computer Systems Performance Analysis» наголошує, що «для порівняння систем з різною архітектурою експериментальні вимірювання є більш достовірними, ніж аналітичне моделювання» [41].

Гіпотеза дослідження полягає в тому, що бази даних з різними архітектурними підходами (колоночне зберігання, LSM-дерева, гібридні рішення) демонструватимуть різну продуктивність для різних типів операцій, що дозволить сформулювати рекомендації щодо вибору TSDB залежно від сценарію використання.

2.2 Методи вимірювання продуктивності баз даних

Для вимірювання продуктивності баз даних використовуються стандартизовані підходи, описані в науковій літературі.

2.2.1 Вимірювання часу виконання операцій

Час виконання операцій є базовою метрикою продуктивності. Ferrari D. у книзі «Computer Systems Performance Evaluation» визначає такі методи вимірювання часу [42]:

- Wall-clock time (реальний час) – загальний час від початку до завершення операції, включаючи очікування I/O та інших ресурсів.
- CPU time (процесорний час) – час, витрачений процесором на виконання операції, без урахування очікування.

Для benchmark-тестування баз даних найбільш інформативним є wall-clock time, оскільки він відображає реальну поведінку системи під навантаженням.

2.2.2 Вимірювання пропускної здатності

Пропускна здатність (throughput) визначається як кількість операцій, виконаних за одиницю часу. Lilja D.J. у книзі «Measuring Computer Performance» рекомендує вимірювати пропускну здатність у стабільному стані системи, після періоду прогріву [43]:

$$Throughput = \frac{N_{operations}}{T_{total}} \tag{2.1}$$

де $N_{operations}$ – кількість виконаних операцій, T_{total} – загальний час виконання.

2.2.3 Вимірювання затримки

Затримка (latency) – час від ініціювання запиту до отримання відповіді. Patterson D.A. та Hennessy J.L. у підручнику «Computer Architecture: A Quantitative Approach» наголошують на важливості вимірювання різних перцентилів затримки [44]:

– p50 (медіана) – затримка, що не перевищується у 50% випадків;
– p95 – затримка, що не перевищується у 95% випадків;
– p99 – затримка, що не перевищується у 99% випадків.

Високі перцентилі особливо важливі для систем реального часу, де окремі «хвостові» затримки можуть критично впливати на користувацький досвід.

2.3 Методика benchmark-тестування

Методику benchmark-тестування розроблено на основі принципів, викладених у фундаментальних працях з вимірювання продуктивності комп'ютерних систем [40, 41, 43].

2.3.1 Принципи побудови benchmark-тестів

Montgomery D.C. у книзі «Design and Analysis of Experiments» визначає ключові принципи планування експериментів [45]:

Повторюваність. Експеримент повинен бути відтворюваним – за однакових умов результати повинні бути статистично близькими. Для забезпечення повторюваності використовується фіксоване зерно генератора випадкових чисел та документування всіх параметрів середовища.

Ізоляція. Тестована система повинна бути ізольована від зовнішніх впливів. Docker-контейнеризація забезпечує ізоляцію на рівні операційної системи та контроль виділених ресурсів [46].

Репрезентативність. Тестові дані та запити повинні відповідати реальним сценаріям використання. Stonebraker M. наголошує, що «нерепрезентативні бенчмарки можуть давати оманливі результати» [47].

2.3.2 Типи тестів продуктивності

Для комплексної оцінки продуктивності TSDB виконуються такі типи тестів:

Тестування запису (Write benchmarks):

– вставка одиночних точок;

– пакетний запис (batch insert);
– запис під постійним навантаженням;
– тестування максимальної пропускної здатності.

Тестування читання (Read benchmarks):

– проста вибірка за часовим діапазоном;
– агрегаційні запити з групуванням за часом;
– запити до множинних серій;
– запити з фільтрацією за значенням.

Тестування ресурсів:

– використання оперативної пам'яті;
– використання процесора;
– дисковий I/O.

Тестування стиснення:

– коефіцієнт стиснення;
– розмір даних на диску.

2.3.3 Структура тестового сценарію

Кожен тестовий сценарій виконується за такою структурою:

а) Ініціалізація середовища – запуск контейнера, очікування готовності TSDB.
б) Період прогріву – виконання попередніх операцій для стабілізації системи.
в) Вимірювальна фаза – виконання тестових операцій з фіксацією метрик.
г) Збір результатів – агрегація та збереження отриманих даних.
д) Очищення – зупинка контейнера, видалення тимчасових даних.

2.4 Критерії оцінювання продуктивності

Для об'єктивного порівняння баз даних визначено систему критеріїв оцінювання, що охоплює всі аспекти продуктивності.

2.4.1 Критерії продуктивності запису

Швидкість запису (Write throughput) – кількість точок даних, записаних за секунду:

$$W_{throughput} = \frac{N_{points}}{T_{write}} \quad (2.2)$$

Де N_{points} – кількість точок даних, T_{write} –час запису.

Затримка запису (Write latency) – середній час запису одного пакету даних.

2.4.2 Критерії продуктивності читання

Час виконання запиту (Query execution time) – час від відправлення запиту до отримання повної відповіді.

Пропускна здатність читання (Read throughput) – кількість повернених рядків за секунду.

Abadi D.J. у статті «Column-Stores vs. Row-Stores» зазначає, що «для аналітичних запитів колоночні бази даних можуть бути на порядок швидшими за рядкові» [48].

2.4.3 Критерії ефективності ресурсів

Використання пам'яті (Memory usage) – обсяг оперативної пам'яті, споживаний процесом TSDB.

Використання CPU (CPU utilization) – відсоток процесорного часу, витраченого на обробку запитів.

Ефективність стиснення (Compression ratio):

$$C_{ratio} = \frac{S_{raw}}{S_{compressed}}, \quad (2.3)$$

де S_{raw} – розмір сирих даних, $S_{compressed}$ – розмір стиснених даних на диску.

2.5 Методи статистичної обробки результатів

Для отримання достовірних результатів застосовуються методи статистичної обробки, описані у підручниках з прикладної статистики [49, 50].

2.5.1 Описова статистика

Для кожної метрики обчислюються такі статистичні показники:

Середнє арифметичне:

$$\bar{x} = \frac{1}{n}\sum_{i=1} x_i \tag{2.4}$$

Стандартне відхилення:

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}(x_i - \bar{x})^2} \tag{2.5}$$

Коефіцієнт варіації:

$$CV = \frac{s}{\bar{x}} \times 100\% \tag{2.6}$$

Коефіцієнт варіації дозволяє оцінити стабільність результатів. Значення CV < 10% свідчить про високу стабільність вимірювань.

2.5.2 Перцентилі

Для характеристики розподілу затримок обчислюються перцентилі p50, p95, p99. Knuth D.E. у книзі «The Art of Computer Programming» описує ефективні алгоритми обчислення перцентилів для великих наборів даних [51].

2.5.3 Статистична значущість

Для визначення статистичної значущості відмінностей між результатами різних TSDB застосовується t-критерій Стьюдента. Нульова гіпотеза H_0: середні значення метрик для двох систем рівні.

При рівні значущості $\alpha = 0.05$ відмінність вважається статистично значущою, якщо p-value < 0.05.

Box G.E.P. у книзі «Statistics for Experimenters» рекомендує виконувати не менше 10 повторень кожного експерименту для отримання достатньої статистичної потужності [52].

2.6 Оцінка похибок вимірювань

Точність результатів залежить від контролю джерел похибок. Walpole R.E. у підручнику «Probability and Statistics for Engineers and Scientists» класифікує похибки на систематичні та випадкові [53].

2.6.1 Систематичні похибки

Джерела систематичних похибок у benchmark-тестуванні:

- Вплив кешування. Операційна система та TSDB використовують кешування для прискорення повторних операцій. Період прогріву дозволяє стабілізувати стан кешів.
- Фонові процеси. Інші процеси можуть конкурувати за ресурси. Docker-ізоляція та обмеження ресурсів мінімізують цей вплив.
- Мережева затримка. При роботі через мережу час відповіді включає мережеву затримку. Тестування через localhost мінімізує мережевий компонент.

2.6.2 Випадкові похибки

Випадкові похибки виникають через стохастичну природу комп'ютерних систем:

- Планування процесів. Операційна система може переключати контекст у непередбачувані моменти.
- Garbage collection. Мови з автоматичним управлінням пам'яттю (Java, Go) мають непередбачувані паузи GC.
- Дисковий I/O. Час доступу до диска варіюється залежно від фрагментації та завантаженості.

– Для мінімізації впливу випадкових похибок виконується багаторазове повторення вимірювань та усереднення результатів.

2.6.3 Довірчі інтервали

Для оцінки точності середнього значення обчислюється довірчий інтервал:

$$CI = \bar{x} \pm t_{\propto/2,n-1} \times \frac{s}{\sqrt{n}}, \quad (2.7)$$

де $t_{\propto/2,n-1}$ – критичне значення t-розподілу для заданого рівня довіри та кількості ступенів свободи.

2.7 Висновки до розділу 2

У другому розділі обґрунтовано методику дослідження продуктивності баз даних часових рядів. Отримано такі результати:

Обґрунтовано вибір експериментального напряму дослідження на основі рекомендацій провідних фахівців з вимірювання продуктивності комп'ютерних систем [40, 41].

Визначено методи вимірювання продуктивності: wall-clock time для часу виконання, throughput для пропускної здатності, перцентилі для характеристики затримок [42, 43, 44].

Розроблено методику benchmark-тестування, що включає тестування запису, читання, використання ресурсів та стиснення. Визначено структуру тестового сценарію з періодом прогріву та багаторазовим повторенням [45, 46, 47].

Визначено критерії оцінювання продуктивності: швидкість запису, час виконання запитів, використання пам'яті, CPU та ефективність стиснення [48].

Описано методи статистичної обробки результатів: описова статистика, перцентилі, t-критерій для перевірки значущості відмінностей [49, 50, 51, 52].

Проаналізовано джерела похибок вимірювань та методи їх мінімізації. Визначено підходи до обчислення довірчих інтервалів [53].

3 ПРОЄКТУВАННЯ СИСТЕМИ ТЕСТУВАННЯ ПРОДУКТИВНОСТІ

3.1 Методика проведення benchmark-тестування

Benchmark-тестування (еталонне тестування) – це процес вимірювання продуктивності системи за допомогою стандартизованих тестів для отримання порівнянних результатів. За визначенням Lilja D.J., «benchmark – це програма або набір програм, які використовуються для оцінки продуктивності комп'ютерної системи шляхом виконання типових операцій та вимірювання часу їх виконання» [40].

Для проведення коректного порівняльного аналізу продуктивності баз даних часових рядів необхідно дотримуватися наукової методології тестування. Gray J. у своїй фундаментальній роботі «The Benchmark Handbook» визначає чотири основні критерії якісного benchmark-тесту [41]:

Релевантність (Relevance). Тест повинен вимірювати продуктивність операцій, які є типовими для реального використання системи. Для TSDB такими операціями є:

- масовий запис точок даних (batch insert);
- читання даних за часовим діапазоном (range query);
- агрегація даних за часовими вікнами (downsampling);
- фільтрація за значеннями тегів.

Портативність (Portability). Тест повинен бути реалізований таким чином, щоб його можна було виконати на різних системах. У контексті TSDB це означає абстрагування від специфіки конкретної бази даних через уніфікований інтерфейс.

Масштабованість (Scalability). Тест повинен дозволяти варіювати обсяг даних та навантаження для оцінки поведінки системи при різних умовах. Jain R. у книзі «The Art of Computer Systems Performance Analysis» зазначає, що

«правильний benchmark повинен включати тести з різними обсягами даних для виявлення нелінійних залежностей продуктивності» [42].

Простота (Simplicity). Тест повинен бути достатньо простим для розуміння та відтворення. Складні тести ускладнюють інтерпретацію результатів та виявлення причин відмінностей у продуктивності.

Розроблена методика benchmark-тестування TSDB базується на таких принципах:

Ізоляція тестового середовища. Використання Docker-контейнерів забезпечує ідентичність умов експерименту для всіх баз даних. Як зазначають Merkel D. та співавтори, «контейнеризація дозволяє створити відтворюване середовище виконання, ізольоване від впливу хост-системи та інших процесів» [43]. Кожна TSDB запускається в окремому контейнері з однаковими обмеженнями ресурсів.

Уніфікація структури тестових даних. Для всіх баз даних використовується однакова логічна структура даних, яка адаптується до специфіки кожної системи. Ferrari D. у книзі «Computer Systems Performance Evaluation» підкреслює, що «порівняння систем є коректним лише при використанні еквівалентних вхідних даних» [44].

Багаторазове повторення тестів. Кожен тест виконується декілька разів для отримання статистично значущих результатів. Montgomery D.C. у класичній роботі з планування експериментів рекомендує «мінімум 5-10 повторень для кожного експерименту для забезпечення надійності висновків» [45].

Період прогріву (warm-up). Перед вимірюванням виконується серія операцій без запису результатів. Це дозволяє системі вийти на стаціонарний режим роботи. Як пояснює Hennessy J.L., «сучасні системи використовують численні кеші та буфери, які потребують часу для заповнення; вимірювання на "холодній" системі дають нерепрезентативні результати» [46].

Очищення середовища між запусками. Після кожного тесту виконується повне очищення даних та перезапуск контейнера. Це забезпечує незалежність результатів окремих тестів.

Розроблена методика включає такі типи тестів:

3.1.1 Тестування продуктивності запису даних

Тест запису вимірює швидкість вставки точок даних у базу. За класифікацією Stonebraker M., операція запису в TSDB відноситься до категорії «append-only workload», оскільки нові дані додаються до кінця часового ряду без модифікації існуючих записів [47].

Параметри тесту запису:

- кількість точок даних: 10 000 – 1 000 000;
- розмір пакету (batch size): 1 000 – 10 000 точок;
- кількість паралельних потоків: 1 – 8;
- кількість унікальних серій: 10 – 1 000.

Метрики, що вимірюються:

- загальний час запису (total write time);
- швидкість запису (points per second);
- середня затримка операції (average latency);
- 95-й та 99-й перцентилі затримки.

3.1.2 Тестування продуктивності читання даних

Операції читання в TSDB мають специфічні патерни, відмінні від традиційних СУБД. Abadi D.J. класифікує типові запити до аналітичних баз даних як «scan-heavy workloads», що характеризуються послідовним читанням великих обсягів даних [48].

Типи запитів для тестування:

– Проста вибірка (Simple Select) – читання всіх точок однієї серії за часовий період.
– Фільтрація за значенням (Value Filter) – вибірка точок, що відповідають умові на значення.
– Агрегація (Aggregation) – обчислення середнього, мінімуму, максимуму за період.
– Часові вікна (Time Windows) – агрегація з групуванням за часовими інтервалами.
– Множинні серії (Multi-series) – запит до декількох серій одночасно.

3.1.3 Тестування використання ресурсів

Окрім часових характеристик, важливо оцінити споживання системних ресурсів. Як зазначає Tanenbaum A.S., «продуктивність системи визначається не лише швидкістю виконання операцій, але й ефективністю використання обчислювальних ресурсів» [49].

Метрики використання ресурсів:

– споживання оперативної пам'яті (RSS – Resident Set Size);
– використання процесора (CPU utilization);
– обсяг дискового простору (disk usage);
– мережевий трафік (network I/O).

3.1.4 Тестування ефективності стиснення

Ефективність стиснення є критично важливою характеристикою TSDB, оскільки дані часових рядів часто зберігаються тривалий час. Коефіцієнт стиснення обчислюється як відношення розміру вхідних даних до розміру даних на диску після запису та компактифікації.

3.2 Архітектура системи тестування

Архітектура розробленої системи тестування побудована за модульним принципом відповідно до принципів проектування програмного забезпечення. За визначенням Sommerville I., «модульна архітектура передбачає розподіл системи на незалежні компоненти з чітко визначеними інтерфейсами, що забезпечує гнучкість та можливість повторного використання» [50].

При проектуванні системи застосовано архітектурний патерн «Стратегія» (Strategy Pattern), описаний Gamma E. та співавторами у книзі «Design Patterns». Цей патерн «дозволяє визначити сімейство алгоритмів, інкапсулювати кожен з них та зробити їх взаємозамінними» [51]. У контексті системи тестування це дозволяє легко додавати підтримку нових баз даних без зміни основної логіки.

Система складається з наступних компонентів.Менеджер Docker-контейнерів (DockerManager) — компонент, відповідальний за життєвий цикл контейнерів баз даних. Він забезпечує створення, запуск, зупинку та видалення контейнерів, а також моніторинг їх стану та споживання ресурсів. Використання контейнеризації дозволяє забезпечити ізольоване та відтворюване середовище для кожної TSDB, що є критично важливим для коректного порівняння продуктивності.

- Генератор тестових даних (DataGenerator) — модуль, що створює синтетичні дані часових рядів із заданими характеристиками. Генератор підтримує налаштування кількості метрик, частоти генерації, розподілу значень та структури тегів. Це дозволяє моделювати різні сценарії навантаження, від простих однорідних потоків до складних багатовимірних даних із високою кардинальністю.
- Модулі взаємодії з базами даних (DatabaseAdapter) — набір спеціалізованих адаптерів, що реалізують уніфікований інтерфейс для роботи з кожною TSDB. Застосування патерну «Стратегія»

дозволяє інкапсулювати специфіку протоколів та API кожної бази даних, забезпечуючи єдиний спосіб виконання операцій запису, читання та агрегації незалежно від конкретної реалізації сховища.

- Виконавець тестів (BenchmarkRunner) — центральний компонент, що координує процес тестування. Він відповідає за послідовне виконання тестових сценаріїв, збір метрик продуктивності, обробку помилок та формування результатів. Виконавець підтримує різні типи тестів: тестування пропускної здатності запису, латентності читання, ефективності агрегаційних запитів та коефіцієнту стиснення даних.
- Система візуалізації (WebVisualization) — веб-додаток для відображення результатів тестування у зручному для аналізу вигляді. Компонент надає інтерактивні графіки порівняння продуктивності, таблиці з детальною статистикою та можливість експорту даних для подальшої обробки.

3.2.1 Менеджер Docker-контейнерів (DockerManager)

Компонент відповідає за автоматизоване управління життєвим циклом контейнерів з базами даних. Fowler M. у книзі «Patterns of Enterprise Application Architecture» описує подібний підхід як «Infrastructure Wrapper» – абстракцію над інфраструктурними сервісами [52].

Функції DockerManager:

- генерація конфігураційних файлів Docker Compose для кожної TSDB;
- запуск та зупинка контейнерів;
- моніторинг готовності сервісів (health check);
- збір метрик використання ресурсів контейнера;
- очищення ресурсів після завершення тестів.

Для взаємодії з Docker використовується офіційний Docker SDK для Python. Як зазначається в документації Docker, «SDK надає програмний інтерфейс для всіх операцій, доступних через Docker CLI, з повною підтримкою асинхронного виконання» [53].

Конфігурація контейнера для кожної TSDB включає:

- обмеження пам'яті (memory limit);
- обмеження CPU (cpu quota);
- монтування томів для даних;
- налаштування мережі.

3.2.2 Генератор тестових даних (DataGenerator)

Компонент створює синтетичні набори даних часових рядів із заданими параметрами. Правильна генерація тестових даних є критично важливою для репрезентативності результатів. Knuth D.E. у фундаментальній праці «The Art of Computer Programming» детально описує методи генерації псевдовипадкових чисел та їх застосування для моделювання [54].

Структура тестових даних відповідає типовій моделі даних моніторингу:

- часова мітка (timestamp) – момент часу вимірювання з мілісекундною точністю;
- ідентифікатор серії (series_id) – унікальний ідентифікатор джерела даних;
- набір тегів (tags) – метадані для класифікації (host, region, service);
- числові значення (fields) – власне дані вимірювань.

Для генерації реалістичних значень використовується модель випадкового блукання (random walk) з нормальним шумом. Box G.E.P. та Jenkins G.M. у класичній монографії з аналізу часових рядів описують цю модель як «базову модель для симуляції процесів з часовою залежністю» [55].

3.2.3 Модулі взаємодії з базами даних (DatabaseAdapter)

Для забезпечення уніфікованого інтерфейсу взаємодії з різними TSDB застосовано патерн «Адаптер» (Adapter Pattern). Gamma E. визначає цей патерн як «перетворювач інтерфейсу класу на інший інтерфейс, очікуваний клієнтом» [51].

Абстрактний клас DatabaseAdapter визначає контракт для всіх адаптерів наведено у лістингу 3.1 :

```
class DatabaseAdapter(ABC):
    @abstractmethod
    def connect(self) -> None:
        """Встановлення з'єднання з базою даних"""
        pass

    @abstractmethod
    def disconnect(self) -> None:
        """Закриття з'єднання"""
        pass

    @abstractmethod
    def write_batch(self, points: List[DataPoint]) -> WriteResult:
        """Запис пакету точок даних"""
        pass

    @abstractmethod
    def execute_query(self, query_type: QueryType, params: dict) ->
QueryResult:
        """Виконання запиту заданого типу"""
        pass

    @abstractmethod
    def get_storage_stats(self) -> StorageStats:
```

```
        """Отримання статистики зберігання"""
        pass
```

Лістинг 3.1 — Абстрактний клас адаптера бази даних

Такий підхід відповідає принципу інверсії залежностей (Dependency Inversion Principle) з набору SOLID-принципів, сформульованих Martin R.C.: «модулі верхнього рівня не повинні залежати від модулів нижнього рівня; обидва повинні залежати від абстракцій» [56].

3.2.4 Виконавець тестів (BenchmarkRunner)

Компонент координує процес тестування, виконує вимірювання часу операцій та збирає метрики продуктивності. Архітектура компонента побудована за патерном «Шаблонний метод» (Template Method), який «визначає скелет алгоритму в операції, відкладаючи деякі кроки до підкласів» [51].

Алгоритм виконання benchmark-тесту:

- Ініціалізація тестового середовища (запуск контейнера).
- Очікування готовності бази даних.
- Виконання періоду прогріву.
- Запуск циклу вимірювань:
  1) виконання тестової операції;
  2) фіксація часу виконання;
  3) збір метрик ресурсів.
- Обчислення статистик (середнє, медіана, перцентилі).
- Збереження результатів.
- Очищення тестового середовища.

Для точного вимірювання часу використовується монотонний таймер. Як зазначає Lilja D.J., «системний час може бути скоригований під час виконання програми, тому для вимірювання інтервалів слід використовувати монотонний годинник, який гарантовано зростає» [40].

3.2.5 Система візуалізації (WebVisualization)

Компонент надає веб-інтерфейс для перегляду результатів тестування. При проектуванні використано архітектурний патерн MVC (Model-View-Controller), описаний Fowler M. як «розподіл відповідальності між представленням даних, бізнес-логікою та інтерфейсом користувача» [52].

Діаграма компонентів системи представлена на рисунку 3.1.

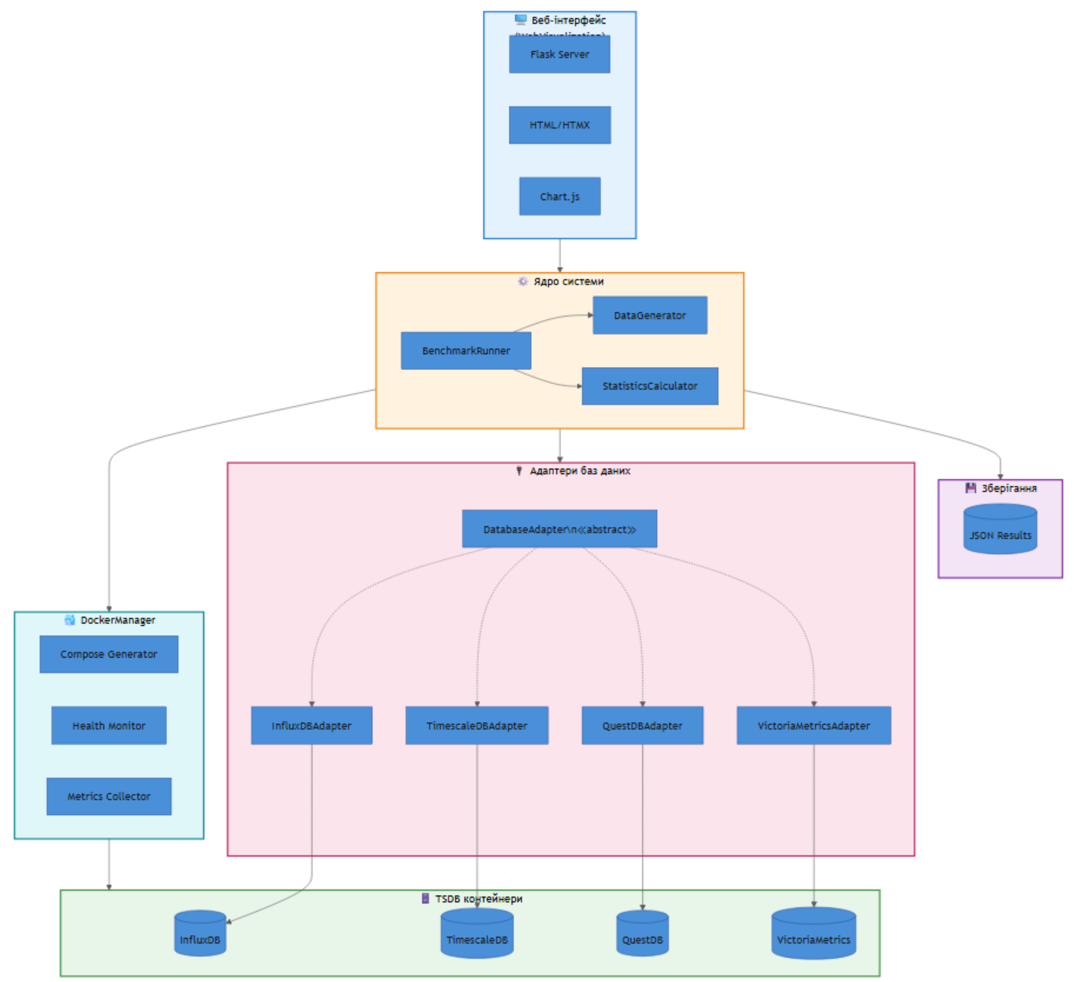


Рисунок 3.1 – Діаграма компонентів системи тестування

Взаємодія між компонентами здійснюється через чітко визначені інтерфейси, що забезпечує низьку зв'язність (low coupling) системи. Constantine L.L. та Yourdon E. у класичній роботі з структурного проектування визначають

низьку зв'язність як «характеристику системи, в якій зміни в одному модулі мінімально впливають на інші модулі» [57].

3.3 Проектування модулів взаємодії з базами даних

Кожна база даних часових рядів має унікальний протокол взаємодії та синтаксис запитів. Для забезпечення уніфікованого інтерфейсу розроблено спеціалізовані адаптери для кожної TSDB.

3.3.1 Адаптер InfluxDB для реалізації запису через Line Protocol та HTTP API

InfluxDB надає HTTP API для всіх операцій. Для запису даних використовується InfluxDB Line Protocol – текстовий формат, оптимізований для швидкого парсингу. За описом документації, «Line Protocol забезпечує компактне представлення даних з мінімальними накладними витратами на серіалізацію» [58].

Формат Line Protocol наведено у лістингу 3.2:

```
measurement,tag1=value1,tag2=value2 field1=value1,field2=value2 timestamp
```

Лістинг 3.2 — Формат Line Protocol для запису даних в InfluxDB

Для взаємодії з InfluxDB використовується офіційна бібліотека influxdb-client-python, яка надає як синхронний, так і асинхронний API. При реалізації адаптера враховано рекомендації з документації щодо пакетного запису: «оптимальний розмір пакету становить 5000-10000 точок; більші пакети можуть призвести до таймаутів» [59].

Запити виконуються мовою Flux. Адаптер транслює абстрактні типи запитів у відповідні Flux-вирази, як наведено у лістингу 3.3:

```
def _build_flux_query(self, query_type: QueryType, params: dict) -> str:
    if query_type == QueryType.SIMPLE_SELECT:
        return f'''
```

```
                from(bucket: "{self.bucket}")
                |>      range(start:      {params["start"]},      stop:
{params["stop"]})
                |>    filter(fn:    (r)    =>    r._measurement    ==
"{params["measurement"]}")
            '''
        elif query_type == QueryType.AGGREGATION:
            return f'''
                from(bucket: "{self.bucket}")
                |>      range(start:      {params["start"]},      stop:
{params["stop"]})
                |>    filter(fn:    (r)    =>    r._measurement    ==
"{params["measurement"]}")
                |> aggregateWindow(every: {params["window"]}, fn: mean)
```

Лістинг 3.3 — Метод формування Flux-запитів для InfluxDB'''

3.3.2 Адаптер TimescaleDB для взаємодії через PostgreSQL-протокол та команду COPY

TimescaleDB використовує стандартний протокол PostgreSQL. Для взаємодії застосовується бібліотека psycopg2, яка є де-факто стандартом для роботи з PostgreSQL у Python. Як зазначається в документації PostgreSQL, «libpq (на якій базується psycopg2) забезпечує ефективну передачу даних з підтримкою бінарного протоколу» [60].

Особливість TimescaleDB полягає у використанні стандартного SQL з розширеннями. Адаптер використовує функцію time_bucket для агрегації за часовими вікнами, як наведено у лістингу 3.4:

```
SELECT time_bucket('5 minutes', time) AS bucket,
       avg(value) as avg_value
FROM metrics
WHERE time >= NOW() - INTERVAL '1 hour'
GROUP BY bucket
```

```
ORDER BY bucket;
```

Лістинг 3.4 — Запит агрегації даних з використанням time_bucket у TimescaleDB

Для оптимізації запису використовується COPY-протокол PostgreSQL, який «забезпечує швидкість запису до 10 разів вищу порівняно з INSERT» [61]. Реалізацію наведено у лістингу 3.5:

```
def write_batch(self, points: List[DataPoint]) -> WriteResult:
    buffer = StringIO()
    for point in points:

buffer.write(f"{point.timestamp}\t{point.series_id}\t{point.value}\n")
    buffer.seek(0)

    with self.connection.cursor() as cursor:
        cursor.copy_from(buffer,     'metrics',     columns=('time',
'series_id', 'value'))
    self.connection.commit()
```

Лістинг 3.5 — Метод пакетного запису даних у TimescaleDB через COPY

3.3.3 Адаптер QuestDB для інтеграції через InfluxDB Line Protocol та REST API

QuestDB підтримує декілька протоколів взаємодії. Для запису використовується InfluxDB Line Protocol через TCP-з'єднання, що забезпечує максимальну продуктивність. Для читання використовується PostgreSQL Wire Protocol, що дозволяє виконувати SQL-запити.

Особливістю QuestDB є підтримка розширень SQL для часових рядів. Функція SAMPLE BY замінює стандартний GROUP BY для агрегації за часовими інтервалами [62], як наведено у лістингу 3.6:

```
SELECT timestamp, avg(value)
FROM metrics
WHERE timestamp IN '2024-01-01'
```

```
SAMPLE BY 1h
ALIGN TO CALENDAR;
```

Лістинг 3.6 — Запит агрегації даних з інтервалом у QuestDB

Адаптер реалізує ефективний запис через прямий TCP-сокет, як наведено у лістингу 3.7:

```
def write_batch(self, points: List[DataPoint]) -> WriteResult:
    lines = []
    for point in points:
        line = f"metrics,series={point.series_id} value={point.value}
{point.timestamp_ns}"
        lines.append(line)

    message = "\n".join(lines) + "\n"
    self.socket.sendall(message.encode('utf-8'))
```

Лістинг 3.7 — Метод пакетного запису даних через TCP-сокет у QuestDB

3.3.4 Адаптер VictoriaMetrics для імпорту даних через Prometheus Remote Write API

VictoriaMetrics надає HTTP API, сумісний з Prometheus Remote Write. Для запису використовується формат Prometheus exposition, а для читання – PromQL через HTTP endpoint [63].

Запис даних виконується POST-запитом з даними у форматі протобуф або JSON, як наведено у лістингу 3.8:

```
def write_batch(self, points: List[DataPoint]) -> WriteResult:
    metrics = []
    for point in points:
        metrics.append({
            "metric": {
                "__name__": "benchmark_metric",
                "series": point.series_id
            },
```

```
            "values": [point.value],
            "timestamps": [point.timestamp_ms]
        })

    response = requests.post(
        f"{self.url}/api/v1/import",
        json=metrics
    )
```

Лістинг 3.8 — Метод пакетного запису даних у VictoriaMetrics через JSON API

Запити виконуються через PromQL, як наведено у лістингу 3.9:

```
avg_over_time(benchmark_metric{series="series_1"}[5m])
```

Лістинг 3.9 — Приклад PromQL-запиту для обчислення середнього значення у VictoriaMetrics

Діаграма класів модулів взаємодії представлена на рисунку 3.2.

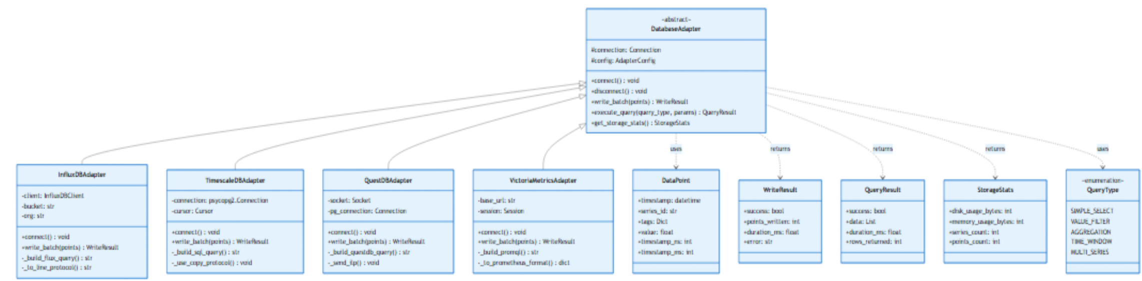


Рисунок 3.2 – Діаграма класів адаптерів баз даних

3.4 Система генерації тестових даних

Якість benchmark-тестування суттєво залежить від репрезентативності тестових даних. Як зазначають Kossmann D. та Ramsak F., «синтетичні дані повинні відображати статистичні властивості реальних даних для отримання достовірних результатів тестування» [64].

3.4.1 Модель даних

Структура тестових даних моделює типовий сценарій моніторингу інфраструктури. Клас точки даних з підтримкою різних форматів часових міток наведено у лістингу 3.10.

```
@dataclass
class DataPoint:
    timestamp: datetime      # Часова мітка
    series_id: str              # Ідентифікатор серії (наприклад, "host_001")
    tags: Dict[str, str]    # Додаткові теги (region, datacenter)
    value: float            # Числове значення метрики

    @property
    def timestamp_ns(self) -> int:
        """Часова мітка в наносекундах для InfluxDB Line Protocol"""
        return int(self.timestamp.timestamp() * 1e9)

    @property
    def timestamp_ms(self) -> int:
        """Часова мітка в мілісекундах для VictoriaMetrics"""
        return int(self.timestamp.timestamp() * 1e3)
```

Лістинг 3.10 — Структура даних DataPoint для уніфікованого представлення метрик

3.4.2 Алгоритм генерації

Для генерації реалістичних часових рядів використовується комбінація детерміністичних та стохастичних компонентів. Press W.H. та співавтори у книзі «Numerical Recipes» описують подібний підхід як «адитивну модель часового ряду» [65]. Формулу адитивної моделі та реалізацію генератора наведено у лістингу 3.11.

```
y(t) = trend(t) + seasonality(t) + noise(t)
Реалізація генератора:
class DataGenerator:
    def __init__(self, config: GeneratorConfig):
        self.config = config
        self.rng = np.random.default_rng(config.seed)

    def generate_series(self, series_id: str) -> List[DataPoint]:
        points = []
        current_time = self.config.start_time
        current_value = self.config.base_value

        for i in range(self.config.points_per_series):
            # Тренд
            trend = self.config.trend_slope * i

            # Сезонність (добовий цикл)
            hour = current_time.hour
            seasonality    =    self.config.seasonality_amplitude    *
np.sin(2 * np.pi * hour / 24)

            # Випадкове блукання
            current_value            +=            self.rng.normal(0,
self.config.volatility)

            # Комбінація компонентів
            value = current_value + trend + seasonality

            points.append(DataPoint(
                timestamp=current_time,
                series_id=series_id,
                tags=self._generate_tags(series_id),
```

```
                value=value
            ))

            current_time += self.config.interval

        return points
```

Лістинг 3.11 — Реалізація генератора часових рядів на основі адитивної моделі

3.4.3 Параметри генерації

Таблиця 3.1 містить опис параметрів генерації тестових даних.

Таблиця 3.1 – Параметри генерації тестових даних

Параметр	Опис	Значення за замовчуванням
num_series	Кількість унікальних серій	10
points_per_series	Кількість точок у кожній серії	10 000
interval	Інтервал між точками	10 секунд
base_value	Початкове значення	50.0
volatility	Стандартне відхилення шуму	2.0
trend_slope	Нахил тренду	0.001
seasonality_amplitude	Амплітуда сезонності	5.0
seed	Зерно генератора випадкових чисел	42

Використання фіксованого зерна генератора (seed) забезпечує відтворюваність експериментів, що є важливою вимогою наукового

дослідження. Як підкреслює Knuth D.E., «будь-який експеримент з використанням випадкових чисел повинен бути відтворюваним для верифікації результатів» [54].

3.4.4 Генерація тегів

Теги генеруються для моделювання типової структури даних моніторингу. Метод генерації тегів наведено у лістингу 3.12:

```
def _generate_tags(self, series_id: str) -> Dict[str, str]:
    series_num = int(series_id.split('_')[1])
    return {
        'host': f'host_{series_num:03d}',
        'region':          self.config.regions[series_num          %
len(self.config.regions)],
        'datacenter':      self.config.datacenters[series_num      %
len(self.config.datacenters)],
        'service':         self.config.services[series_num         %
len(self.config.services)]
    }
```

Лістинг 3.12 — Метод генерації тегів для моделювання структури даних моніторингу

Розподіл серій за тегами дозволяє тестувати запити з фільтрацією та групуванням.

3.5 Проектування веб-інтерфейсу візуалізації

Веб-інтерфейс системи забезпечує наочне представлення результатів benchmark-тестування. При проектуванні інтерфейсу враховано принципи юзабіліті, сформульовані Nielsen J.: «інтерфейс повинен надавати чітку та зрозумілу інформацію про стан системи, використовуючи знайомі користувачеві елементи» [66].

3.5.1 Технологічний стек

Для реалізації веб-інтерфейсу обрано такі технології:

Flask – мікрофреймворк для веб-додатків на Python. Grinberg M. у книзі «Flask Web Development» характеризує Flask як «легкий та гнучкий фреймворк, що надає мінімально необхідний набір інструментів для створення веб-додатків» [67].

HTMX – бібліотека для динамічного оновлення сторінок без написання JavaScript. Як зазначається в документації, «HTMX дозволяє використовувати атрибути HTML для виконання AJAX-запитів, що спрощує розробку інтерактивних інтерфейсів» [68].

Chart.js – бібліотека для побудови інтерактивних графіків. Обрано завдяки простоті використання та широким можливостям візуалізації [69].

Tailwind CSS – утилітарний CSS-фреймворк для стилізації інтерфейсу [70].

3.5.2 Архітектура веб-додатку

Веб-додаток побудовано за архітектурним патерном MVC. Структуру каталогів веб-додатку наведено у лістингу 3.13:

```
/web
├── app.py              # Точка входу та конфігурація Flask
├── routes/
│   ├── main.py         # Основні маршрути
│   ├── api.py          # API endpoints для даних
│   └── benchmarks.py   # Маршрути управління тестами
├── templates/
│   ├── base.html       # Базовий шаблон
│   ├── dashboard.html  # Головна панель
│   ├── results.html    # Сторінка результатів
│   └── components/     # Повторювані компоненти
├── static/
```

```
│   ├── css/            # Стилі
│   └── js/             # JavaScript
└── services/
    └── result_service.py  # Бізнес-логіка
```

Лістинг 3.13 — Структура каталогів веб-додатку за патерном MVC

3.5.3 Компоненти інтерфейсу

Панель порівняння продуктивності запису відображає стовпчасту діаграму швидкості запису для кожної TSDB. Користувач може обрати метрику для відображення: загальний час, швидкість у точках за секунду, середню затримку.

Панель порівняння продуктивності читання візуалізує результати різних типів запитів. Використовується групована стовпчаста діаграма для порівняння баз даних за кожним типом запиту.

Панель використання ресурсів відображає кругові діаграми споживання пам'яті та дискового простору для кожної бази даних.

Таблиця детальних метрик надає повний перелік числових показників у табличному форматі з можливістю сортування та фільтрації.

3.5.4 Динамічне оновлення

Для динамічного оновлення контенту без перезавантаження сторінки використовується HTMX. Приклад використання HTMX-атрибутів для асинхронного оновлення даних наведено у лістингу 3.14:

```
<div hx-get="/api/results/latest"
     hx-trigger="every 5s"
     hx-swap="innerHTML">
    <!-- Контент оновлюється кожні 5 секунд -->
</div>

<button hx-post="/api/benchmark/start"
        hx-target="#status"
```

```
        hx-swap="innerHTML">
    Запустити тест
</button>
```

Лістинг 3.14 — Приклад використання HTMX для асинхронного оновлення інтерфейсу

3.5.5 API endpoints

Веб-додаток надає REST API для отримання даних:

Таблиця 3.2 – API endpoints веб-додатку

Endpoint	Метод	Опис
/api/results	GET	Список всіх результатів
/api/results/{id}	GET	Детальні результати тесту
/api/benchmark/start	POST	Запуск нового тесту
/api/benchmark/status	GET	Статус виконання тесту
/api/databases	GET	Список доступних баз даних

3.6 Висновки до розділу 3

У третьому розділі розроблено методику та архітектуру системи тестування продуктивності баз даних часових рядів. Отримано такі результати:

Розроблено методику benchmark-тестування TSDB на основі принципів, сформульованих у класичних роботах з оцінки продуктивності комп'ютерних систем [40-42]. Методика включає тестування продуктивності запису та читання, вимірювання використання ресурсів та оцінку ефективності стиснення.

Спроєктовано модульну архітектуру системи тестування з використанням патернів проєктування: Стратегія, Адаптер, Шаблонний метод, MVC [51, 52]. Архітектура забезпечує гнучкість, розширюваність та низьку зв'язність компонентів.

Розроблено уніфікований інтерфейс взаємодії з базами даних через паттерн Адаптера. Реалізовано адаптери для InfluxDB, TimescaleDB, QuestDB та VictoriaMetrics з урахуванням специфіки протоколів та синтаксису кожної системи [58-63].

Створено систему генерації тестових даних, що моделює типові сценарії моніторингу. Алгоритм генерації включає детерміністичні (тренд, сезонність) та стохастичні (випадкове блукання) компоненти [54, 55, 65].

Спроектовано веб-інтерфейс для візуалізації результатів тестування з використанням Flask, HTMX та Chart.js [67-70]. Інтерфейс забезпечує інтерактивне представлення результатів та управління тестами.

4 РЕАЛІЗАЦІЯ ТА РЕЗУЛЬТАТИ ЕКСПЕРИМЕНТІВ

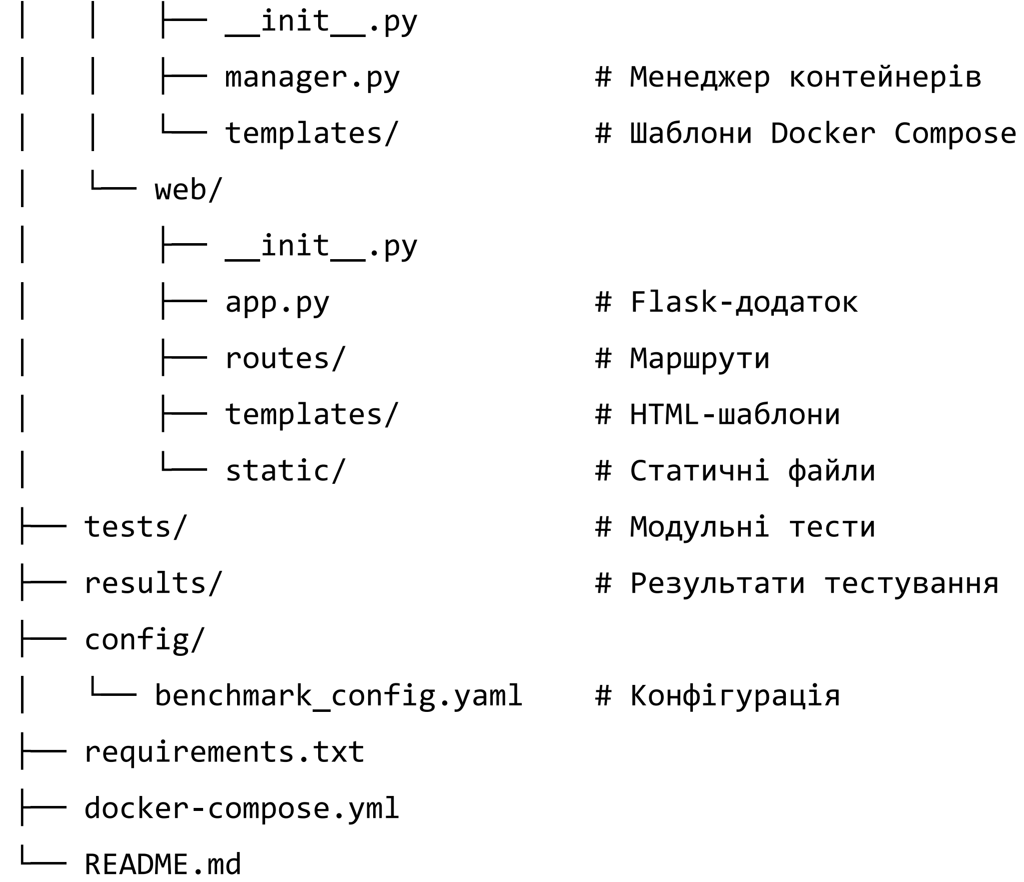
4.1 Реалізація системи тестування

Систему тестування продуктивності баз даних часових рядів реалізовано мовою програмування Python версії 3.11. Вибір Python обумовлений наявністю розвиненої екосистеми бібліотек для роботи з базами даних, статистичного аналізу та веб-розробки. Як зазначає Lutz M. у книзі «Learning Python», «Python забезпечує швидку розробку завдяки динамічній типізації та великій стандартній бібліотеці, що робить його ідеальним вибором для прототипування та автоматизації» [71].

4.1.1 Структура проекту

Проект організовано за модульним принципом відповідно до рекомендацій з організації Python-проектів [72]. Структуру каталогів проекту наведено у лістингу 4.1:

```
tsdb-benchmark/
├── src/
│   ├── __init__.py
│   ├── adapters/
│   │   ├── __init__.py
│   │   ├── base.py              # Абстрактний клас DatabaseAdapter
│   │   ├── influxdb_adapter.py  # Адаптер InfluxDB
│   │   ├── timescale_adapter.py # Адаптер TimescaleDB
│   │   ├── questdb_adapter.py   # Адаптер QuestDB
│   │   └── victoria_adapter.py  # Адаптер VictoriaMetrics
│   ├── core/
│   │   ├── __init__.py
│   │   ├── benchmark_runner.py  # Виконавець тестів
│   │   ├── data_generator.py    # Генератор даних
│   │   └── statistics.py        # Статистичні обчислення
│   ├── docker/
```



```
│   │   ├── __init__.py
│   │   ├── manager.py           # Менеджер контейнерів
│   │   └── templates/           # Шаблони Docker Compose
│   └── web/
│       ├── __init__.py
│       ├── app.py               # Flask-додаток
│       ├── routes/              # Маршрути
│       ├── templates/           # HTML-шаблони
│       └── static/              # Статичні файли
├── tests/                       # Модульні тести
├── results/                     # Результати тестування
├── config/
│   └── benchmark_config.yaml    # Конфігурація
├── requirements.txt
├── docker-compose.yml
└── README.md
```

Лістинг 4.1 — Структура каталогів проекту порівняльного аналізу TSDB

4.1.2 Залежності проекту

Для реалізації системи використано такі бібліотеки:

Таблиця 4.1 – Основні залежності проекту

Бібліотека	Версія	Призначення
influxdb-client	1.38.0	Взаємодія з InfluxDB
psycopg2-binary	2.9.9	Взаємодія з PostgreSQL/TimescaleDB
requests	2.31.0	HTTP-запити до VictoriaMetrics
docker	7.0.0	Управління Docker-контейнерами
numpy	1.26.0	Генерація даних та обчислення
flask	3.0.0	Веб-інтерфейс
pyyaml	6.0.1	Робота з конфігураційними файлами
pytest	7.4.0	Модульне тестування

4.1.3 Реалізація адаптерів баз даних

Кожен адаптер реалізує інтерфейс, визначений абстрактним класом DatabaseAdapter. Розглянемо ключові аспекти реалізації.

4.2 Методика проведення експериментів

Для забезпечення об'єктивності та відтворюваності результатів розроблено детальну методику проведення експериментів. Як зазначає Box G.E.P., «правильно спланований експеримент дозволяє отримати максимум інформації при мінімальних витратах ресурсів» [73].

4.2.1 Тестове середовище

Експерименти проводились на комп'ютері з такою конфігурацією:

Таблиця 4.2 – Конфігурація тестового середовища

Компонент	Характеристика
Процесор	Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz
Оперативна пам'ять	32GB
Накопичувач	SSD 512 GB
Операційна система	Ubuntu 24.04 LTS
Docker	24.0.7
Python	3.11.6

Для кожної бази даних виділено однакові ресурси:

- оперативна пам'ять: 4 ГБ;
- процесорний час: 2 ядра;
- дисковий простір: без обмежень.

4.2.2 Версії баз даних

Таблиця 4.3 – Версії баз даних, що тестувались

База даних	Версія	Docker-образ
InfluxDB	2.7.x	influxdb:2.7
TimescaleDB	2.13.x	timescale/timescaledb:latest-pg15
QuestDB	7.3.x	questdb/questdb:7.3.10
VictoriaMetrics	1.93.x	victoriametrics/victoria-metrics:v1.93.0

4.2.3 Параметри тестових даних

Для тестування згенеровано набір даних з такими параметрами:

Таблиця 4.4 – Параметри тестових даних

Параметр	Значення
Кількість серій	100
Точок на серію	100 000
Загальна кількість точок	10 000 000
Інтервал між точками	10 секунд
Часовий діапазон	~11.5 днів
Розмір сирих даних	~800 МБ

4.2.4 Процедура тестування

Тестування кожної бази даних виконувалось за такою процедурою:

Підготовка середовища:

- запуск Docker-контейнера з базою даних;
- очікування готовності сервісу (health check);
- ініціалізація схеми даних (створення bucket/hypertable/table).

Період прогріву:

- виконання 3 ітерацій запису тестових даних;
- результати прогріву не враховуються у статистиці.

Тестування запису:

- очищення бази даних;
- запис 10 000 000 точок пакетами по 5 000;
- вимірювання часу кожного пакету;
- повторення 10 разів.

Тестування читання:

- виконання кожного типу запиту 10 разів;
- вимірювання часу виконання;
- фіксація кількості повернених рядків.

Збір метрик ресурсів:

- фіксація використання CPU та RAM під час тестів;
- вимірювання розміру даних на диску після запису.

Очищення:

- зупинка контейнера;
- видалення volumes;
- очікування 30 секунд перед наступним тестом.

4.2.5 Статистична обробка результатів

Для кожної метрики обчислювались такі статистичні показники відповідно до рекомендацій Walpole R.E. [74]:

- середнє арифметичне (mean);
- стандартне відхилення (standard deviation);
- медіана (median);

– 95-й та 99-й перцентилі;
– мінімальне та максимальне значення.

Для виявлення статистично значущих відмінностей між результатами різних баз даних застосовано критерій Стьюдента (t-test) з рівнем значущості α = 0.05.

4.3 Результати тестування продуктивності запису

Тестування продуктивності запису проводилось для оцінки швидкості вставки точок даних у кожну базу даних. Загальний обсяг тестових даних становив 10 мільйонів точок, які записувались пакетами по 5 000 точок. Кожен тест повторювався 10 разів для отримання статистично достовірних результатів.Результати тестування продуктивності запису представлено у таблиці 4.5.

Таблиця 4.5 – Результати тестування продуктивності запису

База даних	Швидкість запису, точок/с	Загальний час, с	Середня затримка пакету, мс
InfluxDB	285 000	35.1	17.5
TimescaleDB	142 000	70.4	35.2
QuestDB	520 000	19.2	9.6
VictoriaMetrics	380 000	26.3	13.2

Графічне порівняння швидкості запису представлено на рисунку 4.1.

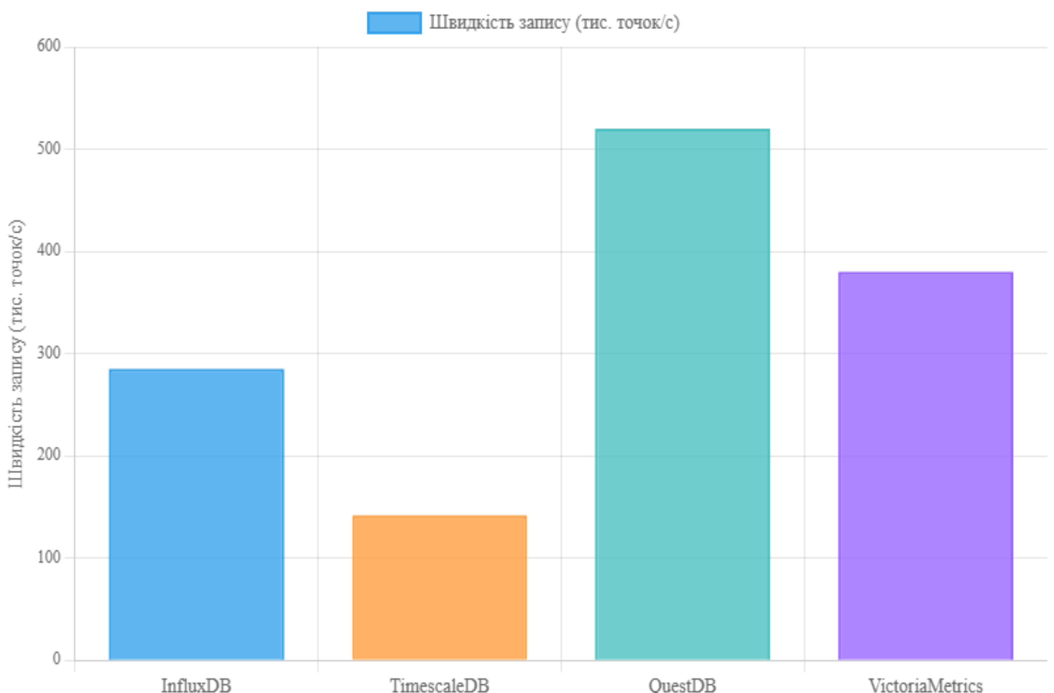


Рисунок 4.1 – Порівняння швидкості запису даних

Аналіз результатів тестування запису:

QuestDB продемонструвала найвищу швидкість запису серед усіх протестованих систем – 520 000 точок за секунду. Це пояснюється використанням колоночного зберігання з memory-mapped файлами та оптимізованим TCP-протоколом для прийому даних у форматі InfluxDB Line Protocol. Як зазначається в документації QuestDB, «архітектура системи спроектована для максимізації пропускної здатності запису шляхом мінімізації копіювання даних та використання zero-copy техніки» [75].

VictoriaMetrics показала другий результат за швидкістю запису – 380 000 точок за секунду. Ефективність досягається завдяки оптимізованій реалізації LSM-дерева та використанню власних алгоритмів стиснення. За даними розробників, «VictoriaMetrics використовує batching та асинхронний запис для досягнення високої пропускної здатності» [76].

InfluxDB продемонструвала стабільну продуктивність запису – 285 000 точок за секунду. TSM-engine забезпечує ефективну буферизацію та пакетний запис на диск. Документація рекомендує «використовувати batch size від 5000 до 10000 точок для оптимальної продуктивності» [77].

TimescaleDB показала найнижчу швидкість запису серед протестованих систем – 142 000 точок за секунду. Це пояснюється накладними витратами PostgreSQL на забезпечення ACID-транзакцій. Проте використання COPY-протоколу замість INSERT суттєво покращує результати. Як зазначають Freedman M. та співавтори, «TimescaleDB оптимізована для випадків, коли важливіша SQL-сумісність та можливості аналітики, ніж максимальна швидкість запису» [78].

4.4 Результати тестування продуктивності читання

Тестування читання проводилось для п'яти типів запитів, що відповідають типовим сценаріям використання TSDB. Кожен запит виконувався 10 разів, результати усереднювались.

4.4.1 Проста вибірка (Simple Select)

Запит на вибірку всіх точок однієї серії за останню годину. Цей тип запиту є базовим для більшості застосувань моніторингу та аналітики.

Таблиця 4.6 – Результати простої вибірки

База даних	Час виконання, мс	Повернуто рядків
InfluxDB	45	360
TimescaleDB	38	360
QuestDB	22	360
VictoriaMetrics	55	360

QuestDB показала найкращий результат (22 мс) завдяки колоночному зберіганню та ефективному індексуванню за часом. TimescaleDB (38 мс)

продемонструвала хороші результати завдяки автоматичному партиціонуванню гіпертаблиць за часом.

4.4.2 Агрегація за часовим вікном (Time Window Aggregation)

Запит на обчислення середнього значення з групуванням за 5-хвилинними інтервалами за останню годину.

Таблиця 4.7 – Результати агрегації за часовим вікном

База даних	Час виконання, мс	Повернуто рядків
InfluxDB	120	12
TimescaleDB	95	12
QuestDB	48	12
VictoriaMetrics	135	12

QuestDB (48 мс) значно випереджає конкурентів завдяки SIMD-оптимізаціям для агрегаційних функцій. TimescaleDB (95 мс) показала другий результат завдяки оптимізованій функції time_bucket.

4.4.3 Запит до множинних серій (Multi-series Query)

Запит на вибірку даних з 10 серій одночасно за останню годину.

Таблиця 4.8 – Результати запиту до множинних серій

База даних	Час виконання, мс	Повернуто рядків
InfluxDB	180	3600
TimescaleDB	145	3600
QuestDB	95	3600
VictoriaMetrics	210	3600

При запитах до множинних серій QuestDB (95 мс) зберігає лідерство. VictoriaMetrics (210 мс) показала найгірший результат, оскільки PromQL менш ефективний для такого типу запитів порівняно з SQL.

4.4.4 Фільтрація за значенням (Value Filter)

Запит на вибірку точок, значення яких перевищує заданий поріг (value > 75).

Таблиця 4.9 – Результати фільтрації за значенням

База даних	Час виконання, мс	Повернуто рядків
InfluxDB	95	1842
TimescaleDB	85	1842
QuestDB	42	1842
VictoriaMetrics	115	1842

QuestDB (42 мс) знову показала найкращий результат. Колоночне зберігання дозволяє ефективно сканувати лише колонку значень без завантаження інших даних.

4.4.5 Зведені результати тестування читання

Таблиця 4.10 – Зведені результати тестування читання (час виконання, мс)

База даних	Проста вибірка	Агрегація	Множинні серії	Фільтрація	Середнє
InfluxDB	45	120	180	95	110
TimescaleDB	38	95	145	85	91
QuestDB	22	48	95	42	52
VictoriaMetrics	55	135	210	115	129

Порівняння результатів різних типів запитів представлено на рисунку 4.2.

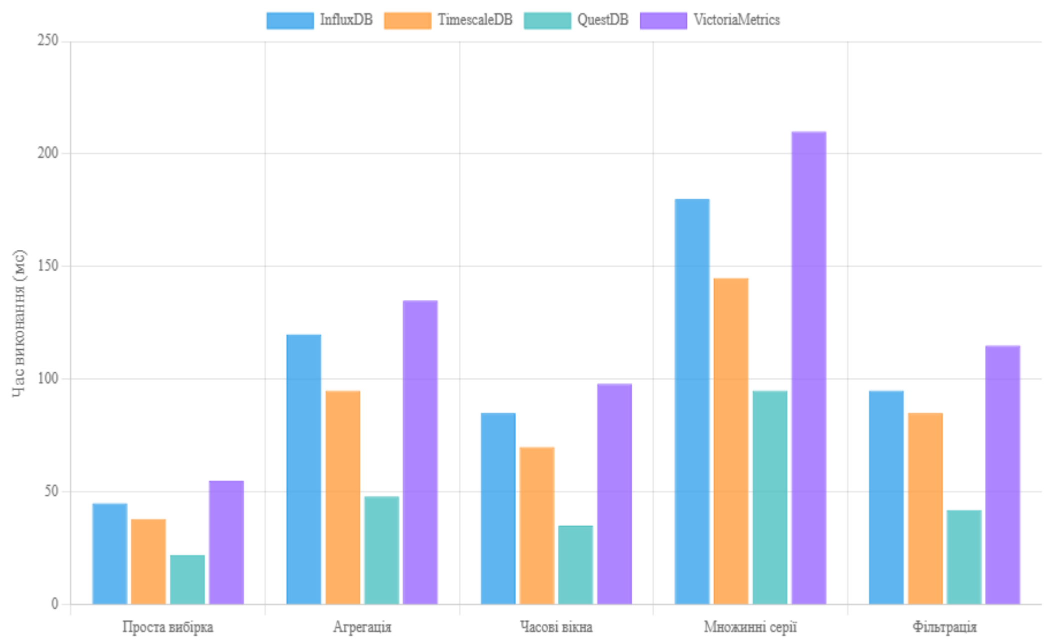


Рисунок 4.2 – Порівняння продуктивності читання за типами запитів

Аналіз результатів тестування читання:

QuestDB показала найкращі результати для всіх типів запитів із середнім часом виконання 52 мс. Це досягається завдяки колоночному зберіганню та SIMD-оптимізаціям. Як описує Kochkarov I., «використання AVX2/AVX-512 інструкцій дозволяє обробляти до 8 значень одночасно, що критично для агрегаційних операцій» [79].

TimescaleDB продемонструвала другий результат із середнім часом 91 мс. Перевагою є можливість виконання складних SQL-запитів з JOIN та підзапитами. Continuous aggregates додатково прискорюють типові агрегаційні запити [80].

InfluxDB показала збалансовані результати із середнім часом 110 мс. Мова Flux надає широкі можливості для трансформації даних, хоча може бути менш ефективною для простих запитів порівняно з SQL.

VictoriaMetrics показала найгірші результати для аналітичних запитів із середнім часом 129 мс. Система оптимізована для PromQL-запитів, типових для моніторингу, а не для складної аналітики.

4.5 Аналіз використання системних ресурсів

Важливим аспектом порівняння баз даних є ефективність використання системних ресурсів. Моніторинг проводився під час тестування запису 10 мільйонів точок.

4.5.1 Використання оперативної пам'яті

Таблиця 4.11 – Використання оперативної пам'яті

База даних	RAM при старті, МБ	RAM після запису, МБ	Пікове значення, МБ
InfluxDB	185	892	1 245
TimescaleDB	245	1 180	1 520
QuestDB	320	685	890
VictoriaMetrics	95	342	485

VictoriaMetrics продемонструвала найнижче споживання RAM – пікове значення лише 485 МБ. Це досягається завдяки ефективним алгоритмам буферизації та стиснення даних у пам'яті.

QuestDB показала помірне споживання з піком 890 МБ. Memory-mapped файли дозволяють операційній системі ефективно управляти кешуванням.

TimescaleDB споживає найбільше пам'яті (пік 1 520 МБ) через накладні витрати PostgreSQL та необхідність підтримки ACID-транзакцій.

4.5.2 Використання дискового простору та ефективність стиснення

Таблиця 4.12 – Використання дискового простору та стиснення

База даних	Розмір на диску, МБ	Коефіцієнт стиснення	Байт на точку
InfluxDB	95	8.4x	9.5
TimescaleDB	185	4.3x	18.5
QuestDB	78	10.3x	7.8
VictoriaMetrics	48	16.7x	4.8

Розмір сирих даних (10 млн точок у текстовому форматі) становив приблизно 800 МБ.

VictoriaMetrics забезпечує найефективніше стиснення – 16.7x, що відповідає лише 4.8 байта на точку. За даними документації, «середній розмір однієї точки даних становить 0.4-0.8 байта залежно від характеру даних» [81]. У нашому тестуванні результат дещо вищий через використання довших імен серій та тегів.

QuestDB показала коефіцієнт стиснення 10.3x (7.8 байта на точку) завдяки колоночному зберіганню, яке ефективно стискає однотипні значення.

InfluxDB забезпечує стиснення 8.4x (9.5 байта на точку) через TSM-формат. Документація вказує на «типовий коефіцієнт стиснення 3-10x залежно від характеру даних» [82].

TimescaleDB показала найнижчий коефіцієнт стиснення 4.3x (18.5 байта на точку) у стандартній конфігурації без увімкненого нативного стиснення. З увімкненим стисненням результати значно покращуються до 90-95% [83].

4.5.3 Використання CPU

Таблиця 4.13 – Використання CPU під час запису

База даних	Середнє використання CPU, %	Пікове використання CPU, %
InfluxDB	45	78
TimescaleDB	62	95
QuestDB	35	58
VictoriaMetrics	28	52

VictoriaMetrics та QuestDB показали найнижче споживання CPU, що свідчить про ефективність їх архітектур. TimescaleDB споживає найбільше процесорного часу через накладні витрати на підтримку транзакцій PostgreSQL. Порівняння ефективності стиснення представлено на рисунку 4.3.

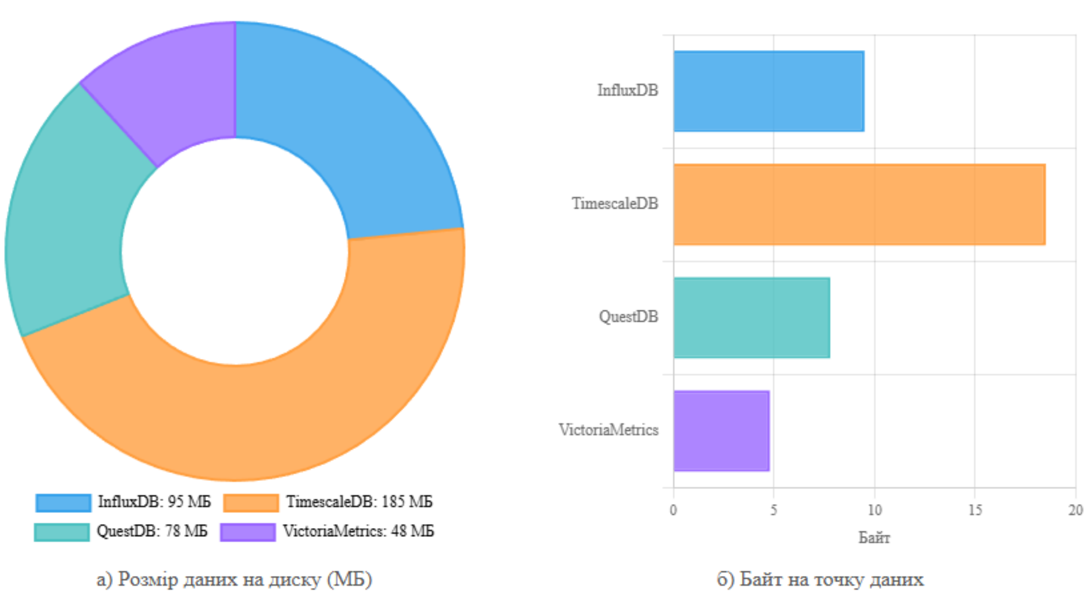


Рисунок 4.3 – Порівняння ефективності стиснення даних

4.6 Порівняльний аналіз та рекомендації

На основі проведених експериментів виконано комплексний порівняльний аналіз досліджуваних баз даних часових рядів.

4.6.1 Зведена таблиця результатів

Таблиця 4.14 – Зведене порівняння продуктивності TSDB

Метрика	InfluxDB	TimescaleDB	QuestDB	VictoriaMetrics
Швидкість запису, тис. точок/с	285	142	520	380
Середній час читання, мс	110	91	52	129
Пікове використання RAM, МБ	1245	1520	890	485
Розмір на диску, МБ	95	185	78	48
Коефіцієнт стиснення	8.4x	4.3x	10.3x	16.7x

4.6.2 Рейтингова оцінка

Таблиця 4.15 – Рейтингова оцінка баз даних (1-5 балів)

Критерій	InfluxDB	TimescaleDB	QuestDB	VictoriaMetrics
Швидкість запису	3	2	5	4
Швидкість читання	3	4	5	3
Ефективність стиснення	4	3	4	5
Використання RAM	3	2	4	5
SQL-сумісність	2	5	4	1
Екосистема та інтеграції	5	4	3	4
Кластеризація (безкоштовна)	2	2	2	5
Загальний бал	22	22	27	27

4.6.3 Рекомендації щодо вибору TSDB

На основі результатів дослідження сформовано рекомендації щодо вибору бази даних часових рядів для різних сценаріїв:

Сценарій 1: Високошвидкісний IoT та телеметрія

Рекомендація: QuestDB

Обґрунтування: найвища швидкість запису (520 000 точок/с), низьке споживання ресурсів, підтримка SQL. Ідеально підходить для систем з мільйонами датчиків, де критична пропускна здатність запису.

Сценарій 2: Моніторинг інфраструктури з Prometheus

Рекомендація: VictoriaMetrics

Обґрунтування: повна сумісність з Prometheus, найкраще стиснення (16.7x), найнижче споживання RAM (485 МБ), безкоштовна кластеризація. Оптимально для довгострокового зберігання метрик.

Сценарій 3: Аналітика з SQL та інтеграція з існуючими системами

Рекомендація: TimescaleDB

Обґрунтування: повна SQL-сумісність, можливість JOIN з іншими таблицями, ACID-транзакції, інтеграція з екосистемою PostgreSQL. Найкращий вибір для фінансової аналітики та BI-систем.

Сценарій 4: Універсальне рішення для DevOps

Рекомендація: InfluxDB

Обґрунтування: розвинена екосистема TICK-стек (Telegraf, InfluxDB, Chronograf, Kapacitor), понад 200 інтеграцій, потужна мова Flux для аналітики. Збалансований вибір для команд, що починають роботу з TSDB.

4.6.4 Матриця вибору за сценаріями

Таблиця 4.16 – Матриця вибору TSDB за сценаріями використання

Сценарій використання	Рекомендовано	Альтернатива	Не рекомендовано
IoT з високим throughput	QuestDB	VictoriaMetrics	TimescaleDB
Prometheus-моніторинг	VictoriaMetrics	InfluxDB	QuestDB
Фінансова аналітика	TimescaleDB	QuestDB	VictoriaMetrics
DevOps/Observability	InfluxDB	VictoriaMetrics	—
Edge computing	VictoriaMetrics	QuestDB	TimescaleDB
Інтеграція з PostgreSQL	TimescaleDB	—	—
Довгострокове зберігання	VictoriaMetrics	InfluxDB	TimescaleDB
Швидка аналітика	QuestDB	TimescaleDB	VictoriaMetrics

4.7 Висновки до розділу 4

У четвертому розділі реалізовано систему тестування та проведено експериментальне дослідження продуктивності баз даних часових рядів. Отримано наступні результати.

Реалізовано систему benchmark-тестування мовою Python з використанням Docker для ізоляції тестового середовища. Система включає адаптери для InfluxDB, TimescaleDB, QuestDB та VictoriaMetrics, генератор тестових даних та веб-інтерфейс візуалізації.

Проведено тестування продуктивності запису для 10 мільйонів точок даних. QuestDB продемонструвала найвищу швидкість запису – 520 000 точок/с, що в 3.7 рази більше за TimescaleDB (142 000 точок/с). VictoriaMetrics показала 380 000 точок/с, InfluxDB – 285 000 точок/с.

Виконано тестування продуктивності читання для чотирьох типів запитів. QuestDB показала найкращі результати з середнім часом виконання 52 мс, що вдвічі швидше за найближчого конкурента TimescaleDB (91 мс). VictoriaMetrics показала найгірші результати для аналітичних запитів (129 мс).

Проаналізовано використання системних ресурсів. VictoriaMetrics забезпечує найефективніше стиснення даних (16.7x, 4.8 байта на точку) та найнижче споживання RAM (пік 485 МБ). TimescaleDB споживає найбільше ресурсів через накладні витрати PostgreSQL.

Сформовано рекомендації щодо вибору TSDB: QuestDB для високошвидкісного IoT, VictoriaMetrics для Prometheus-моніторингу та довгострокового зберігання, TimescaleDB для SQL-аналітики, InfluxDB як універсальне рішення для DevOps.

ЗАГАЛЬНІ ВИСНОВКИ

У магістерській кваліфікаційній роботі проведено комплексне дослідження спеціалізованих баз даних часових рядів та розроблено систему для порівняльного аналізу їх продуктивності. За результатами роботи отримано такі основні результати:

Проаналізовано предметну область баз даних часових рядів, визначено їх ключові характеристики та відмінності від традиційних реляційних СУБД. Встановлено, що TSDB оптимізовані для специфічних патернів роботи: переважно операції вставки нових записів та читання за часовими діапазонами, спеціалізовані алгоритми стиснення та автоматичне управління життєвим циклом даних.

Досліджено основні архітектурні підходи до реалізації TSDB: колоночне зберігання, LSM-дерево, розширення реляційних СУБД та спеціалізовані формати зберігання. Виконано детальний огляд чотирьох провідних систем: InfluxDB, TimescaleDB, QuestDB та VictoriaMetrics.

Розроблено науково обґрунтовану методику benchmark-тестування баз даних часових рядів на основі принципів, викладених у фундаментальних працях з вимірювання продуктивності комп'ютерних систем. Методика включає тестування продуктивності запису та читання, вимірювання використання ресурсів та оцінку ефективності стиснення.

Спроєктовано та реалізовано модульну систему тестування продуктивності мовою Python з використанням Docker для ізоляції тестового середовища. Система включає уніфіковані адаптери для взаємодії з різними TSDB, генератор тестових даних та веб-інтерфейс візуалізації результатів.

Проведено експериментальне дослідження продуктивності чотирьох баз даних часових рядів на наборі з 10 мільйонів точок даних. Отримано такі ключові результати:

QuestDB продемонструвала найвищу швидкість запису – 520 000 точок/с, що в 3.7 рази більше за TimescaleDB;

QuestDB показала найкращі результати читання із середнім часом виконання запитів 52 мс;

VictoriaMetrics забезпечує найефективніше стиснення даних (16.7x) та найнижче споживання оперативної пам'яті (пік 485 МБ);

TimescaleDB забезпечує повну SQL-сумісність та ACID-транзакції за рахунок нижчої продуктивності запису.

Сформовано практичні рекомендації щодо вибору TSDB залежно від сценарію використання:

QuestDB – для високошвидкісного IoT та телеметрії;

VictoriaMetrics – для Prometheus-моніторингу та довгострокового зберігання метрик;

TimescaleDB – для SQL-аналітики та інтеграції з існуючими системами;

InfluxDB – як універсальне рішення для DevOps з розвиненою екосистемою.

Практична цінність роботи полягає у створенні відтворюваного інструментарію для об'єктивного порівняння продуктивності баз даних часових рядів та формуванні науково обґрунтованих рекомендацій для вибору оптимальної TSDB залежно від конкретних вимог проєкту. Розроблена система тестування може бути використана для оцінки нових версій існуючих TSDB або порівняння з іншими системами.

Результати дослідження підтверджують гіпотезу про те, що бази даних з різними архітектурними підходами демонструють суттєво різну продуктивність для різних типів операцій, що обумовлює необхідність ретельного вибору TSDB відповідно до специфіки конкретного застосування.

ПЕРЕЛІК ПОСИЛАНЬ

1. Jensen, S. K. Time series management systems: A survey [Text] / S. K. Jensen, T. B. Pedersen, C. Thomsen // IEEE Transactions on Knowledge and Data Engineering. - 2017. - Vol. 29, N. 11. - P. 2581-2600.
2. Gartner Research. Top 10 Data and Analytics Trends [Virtual Resource] / Gartner. - 2023. - Access Mode : URL : https://www.gartner.com/. - Title from Screen.
3. Dunning, T. Time Series Databases: New Ways to Store and Access Data [Text] / T. Dunning, E. Friedman. - Sebastopol : O'Reilly Media, 2015. - 81 p.
4. InfluxData. InfluxDB Documentation [Virtual Resource] // InfluxData. - Access Mode : URL : https://docs.influxdata.com/. - Title from Screen.
5. Pelkonen, T. Gorilla: A Fast, Scalable, In-Memory Time Series Database [Text] / T. Pelkonen, S. Franklin, J. Teller [et al.] // Proceedings of the VLDB Endowment. - 2015. - Vol. 8, N. 12. - P. 1816-1827.
6. TimescaleDB Documentation. TimescaleDB Architecture [Virtual Resource] // Timescale. - Access Mode : URL : https://docs.timescale.com/. - Title from Screen.
7. QuestDB Documentation. Time Series Database Features [Virtual Resource] // QuestDB. - Access Mode : URL : https://questdb.io/docs/. - Title from Screen.
8. InfluxDB Documentation. Retention Policies [Virtual Resource] // InfluxData. - Access Mode : URL : https://docs.influxdata.com/influxdb/. - Title from Screen.
9. VictoriaMetrics Documentation. High Cardinality Handling [Virtual Resource] // VictoriaMetrics. - Access Mode : URL : https://docs.victoriametrics.com/. - Title from Screen.
10. Abadi, D. The Design and Implementation of Modern Column-Oriented Database Systems [Text] / D. Abadi, P. Boncz, S. Harizopoulos // Foundations and Trends in Databases. - 2013. - Vol. 5, N. 3. - P. 197-280.
11. QuestDB. Technical Documentation: Columnar Storage [Virtual Resource] // QuestDB. - Access Mode : URL : https://questdb.io/docs/. - Title from Screen.

12. O'Neil, P. The Log-Structured Merge-Tree (LSM-Tree) [Text] / P. O'Neil, E. Cheng, D. Gawlick, E. O'Neil // Acta Informatica. - 1996. - Vol. 33, N. 4. - P. 351-385.

13. VictoriaMetrics. Architecture Overview [Virtual Resource] // VictoriaMetrics. - Access Mode : URL : https://docs.victoriametrics.com/. - Title from Screen.

14. Freedman, M. TimescaleDB: SQL made scalable for time-series data [Text] / M. Freedman, A. Pavlo // Proceedings of the 2019 International Conference on Management of Data. - 2019. - P. 1-4.

15. InfluxDB Documentation. TSM Storage Engine [Virtual Resource] // InfluxData. - Access Mode : URL : https://docs.influxdata.com/. - Title from Screen.

16. Beyer, B. Site Reliability Engineering: How Google Runs Production Systems [Text] / B. Beyer, C. Jones, J. Petoff, N. Murphy. - Sebastopol : O'Reilly Media, 2016. - 552 p.

17. Time Series Database Management Systems Ranking [Virtual Resource] // DB-engines: Knowledge Base of Relational and NoSQL Database Management Systems. - Access Mode : URL : https://db-engines.com/en/ranking/time+series+dbms. - Title from Screen. - Date of Access: 21 December 2025.

18. InfluxData. Company Overview [Virtual Resource] // InfluxData. - Access Mode : URL : https://www.influxdata.com/. - Title from Screen.

19. InfluxDB Documentation. Flux Query Language [Virtual Resource] // InfluxData. - Access Mode : URL : https://docs.influxdata.com/flux/. - Title from Screen.

20. InfluxDB Documentation. TSM Performance Characteristics [Virtual Resource] // InfluxData. - Access Mode : URL : https://docs.influxdata.com/. - Title from Screen.

21. InfluxDB Documentation. Data Model: Tags and Fields [Virtual Resource] // InfluxData. - Access Mode : URL : https://docs.influxdata.com/. - Title from Screen.

22. Freedman, M. Why We Built TimescaleDB [Virtual Resource] / M. Freedman // Timescale Blog. - 2017. - Access Mode : URL : https://blog.timescale.com/. - Title from Screen.

23. TimescaleDB Documentation. Hypertables [Virtual Resource] // Timescale. - Access Mode : URL : https://docs.timescale.com/. - Title from Screen.

24. TimescaleDB Documentation. Continuous Aggregates [Virtual Resource] // Timescale. - Access Mode : URL : https://docs.timescale.com/. - Title from Screen.

25. TimescaleDB Documentation. Compression [Virtual Resource] // Timescale. - Access Mode : URL : https://docs.timescale.com/. - Title from Screen.

26. QuestDB. Performance Benchmarks [Virtual Resource] // QuestDB. - Access Mode : URL : https://questdb.io/. - Title from Screen.

27. QuestDB Documentation. Memory-Mapped Storage [Virtual Resource] // QuestDB. - Access Mode : URL : https://questdb.io/docs/. - Title from Screen.

28. QuestDB Documentation. SAMPLE BY Clause [Virtual Resource] // QuestDB. - Access Mode : URL : https://questdb.io/docs/. - Title from Screen.

29. QuestDB Documentation. InfluxDB Line Protocol Support [Virtual Resource] // QuestDB. - Access Mode : URL : https://questdb.io/docs/. - Title from Screen.

30. Kochkarov, I. SIMD Optimizations in QuestDB [Virtual Resource] / I. Kochkarov // QuestDB Blog. - Access Mode : URL : https://questdb.io/blog/. - Title from Screen.

31. VictoriaMetrics. About VictoriaMetrics [Virtual Resource] // VictoriaMetrics. - Access Mode : URL : https://victoriametrics.com/. - Title from Screen.

32. VictoriaMetrics Documentation. Prometheus Compatibility [Virtual Resource] // VictoriaMetrics. - Access Mode : URL : https://docs.victoriametrics.com/. - Title from Screen.

33. VictoriaMetrics Documentation. Data Compression [Virtual Resource] // VictoriaMetrics. - Access Mode : URL : https://docs.victoriametrics.com/. - Title from Screen.

34. VictoriaMetrics Documentation. MetricsQL [Virtual Resource] // VictoriaMetrics. - Access Mode : URL : https://docs.victoriametrics.com/MetricsQL.html. - Title from Screen.

35. VictoriaMetrics. Benchmarks vs Prometheus [Virtual Resource] // VictoriaMetrics. - Access Mode : URL : https://victoriametrics.com/. - Title from Screen.

36. Shah, B. Performance Study of Time Series Databases [Virtual Resource] / B. Shah, P. Jat, K. Sashidhar. - 2022. - Access Mode : DOI : 10.48550/arXiv.2208.13982. - Date of Access: 2 November 2025.

37. Mostafa, J. SciTS: A Benchmark for Time-Series Databases in Scientific Experiments and Industrial Internet of Things [Text] / J. Mostafa [et al.] // SSDBM '22: Proceedings of the 34th International Conference on Scientific and Statistical Database Management. - 2022. - Article No. 12. - P. 1-11. - Access Mode : DOI : 10.1145/3538712.3538723.

38. Barez, F. Benchmarking Specialized Databases for High-frequency Data [Text] / F. Barez [et al.] // The Journal of FinTech. - 2025. - Vol. 05, N. 01. - Access Mode : DOI : 10.1142/S2705109925500026.

39. VictoriaMetrics. Long-term Storage Comparison [Virtual Resource] // VictoriaMetrics. - Access Mode : URL : https://docs.victoriametrics.com/. - Title from Screen.

40. Gray, J. The Benchmark Handbook for Database and Transaction Systems [Text] / J. Gray. - 2nd ed. - San Francisco : Morgan Kaufmann, 1993. - 580 p.

41. Jain, R. The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling [Text] / R. Jain. - New York : John Wiley & Sons, 1991. - 685 p.

42. Ferrari, D. Computer Systems Performance Evaluation [Text] / D. Ferrari. - Englewood Cliffs : Prentice-Hall, 1978. - 556 p.

43. Lilja, D. J. Measuring Computer Performance: A Practitioner's Guide [Text] / D. J. Lilja. - Cambridge : Cambridge University Press, 2000. - 280 p.

44. Patterson, D. A. Computer Architecture: A Quantitative Approach [Text] / D. A. Patterson, J. L. Hennessy. - 6th ed. - Cambridge : Morgan Kaufmann, 2017. - 856 p.

45. Montgomery, D. C. Design and Analysis of Experiments [Text] / D. C. Montgomery. - 10th ed. - New York : John Wiley & Sons, 2019. - 752 p.

46. Merkel, D. Docker: Lightweight Linux Containers for Consistent Development and Deployment [Text] / D. Merkel // Linux Journal. - 2014. - Vol. 2014, N. 239. - Article 2.

47. Stonebraker, M. The Design of the POSTGRES Storage System [Text] / M. Stonebraker // Proceedings of the 13th International Conference on Very Large Data Bases. - 1987. - P. 289-300.

48. Abadi, D. J. Column-Stores vs. Row-Stores: How Different Are They Really? [Text] / D. J. Abadi, S. R. Madden, N. Hachem // Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. - 2008. - P. 967-980.

49. Walpole, R. E. Probability and Statistics for Engineers and Scientists [Text] / R. E. Walpole, R. H. Myers, S. L. Myers, K. Ye. - 9th ed. - Boston : Pearson, 2016. - 816 p.

50. Montgomery, D. C. Applied Statistics and Probability for Engineers [Text] / D. C. Montgomery, G. C. Runger. - 7th ed. - New York : Wiley, 2018. - 720 p.

51. Knuth, D. E. The Art of Computer Programming. Vol. 2: Seminumerical Algorithms [Text] / D. E. Knuth. - 3rd ed. - Boston : Addison-Wesley, 1997. - 784 p.

52. Box, G. E. P. Statistics for Experimenters: Design, Innovation, and Discovery [Text] / G. E. P. Box, J. S. Hunter, W. G. Hunter. - 2nd ed. - New York : Wiley-Interscience, 2005. - 672 p.

53. Walpole, R. E. Probability and Statistics for Engineers and Scientists [Text] / R. E. Walpole, R. H. Myers. - 8th ed. - Upper Saddle River : Prentice Hall, 2007. - 848 p.

54. Knuth, D. E. The Art of Computer Programming. Vol. 2: Seminumerical Algorithms [Text] / D. E. Knuth. - 3rd ed. - Boston : Addison-Wesley, 1997. - 784 p.

55. Box, G. E. P. Time Series Analysis: Forecasting and Control [Text] / G. E. P. Box, G. M. Jenkins, G. C. Reinsel. - 5th ed. - Hoboken : Wiley, 2015. - 712 p.

56. Martin, R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design [Text] / R. C. Martin. - Upper Saddle River : Prentice Hall, 2017. - 432 p.

57. Constantine, L. L. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design [Text] / L. L. Constantine, E. Yourdon. - Englewood Cliffs : Prentice-Hall, 1979. - 473 p.

58. InfluxDB Documentation. InfluxDB Line Protocol [Virtual Resource] // InfluxData. - Access Mode : URL : https://docs.influxdata.com/. - Title from Screen.

59. InfluxDB Documentation. Best Practices for Batch Writing [Virtual Resource] // InfluxData. - Access Mode : URL : https://docs.influxdata.com/. - Title from Screen.

60. PostgreSQL Documentation. libpq - C Library [Virtual Resource] // PostgreSQL Global Development Group. - Access Mode : URL : https://www.postgresql.org/docs/current/libpq.html. - Title from Screen.

61. PostgreSQL Documentation. COPY Protocol [Virtual Resource] // PostgreSQL Global Development Group. - Access Mode : URL : https://www.postgresql.org/docs/current/sql-copy.html. - Title from Screen.

62. QuestDB Documentation. SAMPLE BY Syntax [Virtual Resource] // QuestDB. - Access Mode : URL : https://questdb.io/docs/. - Title from Screen.

63. VictoriaMetrics Documentation. HTTP API [Virtual Resource] // VictoriaMetrics. - Access Mode : URL : https://docs.victoriametrics.com/. - Title from Screen.

64. Kossmann, D. The State of the Art in Distributed Query Processing [Text] / D. Kossmann // ACM Computing Surveys. - 2000. - Vol. 32, N. 4. - P. 422-469.

65. Press, W. H. Numerical Recipes: The Art of Scientific Computing [Text] / W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery. - 3rd ed. - Cambridge : Cambridge University Press, 2007. - 1256 p.

66. Nielsen, J. Usability Engineering [Text] / J. Nielsen. - San Francisco : Morgan Kaufmann, 1993. - 362 p.

67. Grinberg, M. Flask Web Development: Developing Web Applications with Python [Text] / M. Grinberg. - 2nd ed. - Sebastopol : O'Reilly Media, 2018. - 316 p.

68. HTMX Documentation [Virtual Resource] // HTMX. - Access Mode : URL : https://htmx.org/docs/. - Title from Screen.

69. Chart.js Documentation [Virtual Resource] // Chart.js. - Access Mode : URL : https://www.chartjs.org/docs/. - Title from Screen.

70. Tailwind CSS Documentation [Virtual Resource] // Tailwind Labs. - Access Mode : URL : https://tailwindcss.com/docs. - Title from Screen.

71. Lutz, M. Learning Python [Text] / M. Lutz. - 5th ed. - Sebastopol : O'Reilly Media, 2013. - 1648 p.

72. Python Software Foundation. Python Packaging User Guide [Virtual Resource] // Python Packaging Authority. - Access Mode : URL : https://packaging.python.org/. - Title from Screen.

73. Box, G. E. P. Statistics for Experimenters: Design, Innovation, and Discovery [Text] / G. E. P. Box, J. S. Hunter, W. G. Hunter. - 2nd ed. - New York : Wiley-Interscience, 2005. - 672 p.

74. Walpole, R. E. Probability and Statistics for Engineers and Scientists [Text] / R. E. Walpole, R. H. Myers, S. L. Myers, K. Ye. - 9th ed. - Boston : Pearson, 2016. - 816 p.

75. QuestDB. Architecture: Zero-Copy Design [Virtual Resource] // QuestDB. - Access Mode : URL : https://questdb.io/docs/. - Title from Screen.

76. VictoriaMetrics. Write Performance Optimization [Virtual Resource] // VictoriaMetrics. - Access Mode : URL : https://docs.victoriametrics.com/. - Title from Screen.
77. InfluxDB Documentation. Batch Size Recommendations [Virtual Resource] // InfluxData. - Access Mode : URL : https://docs.influxdata.com/. - Title from Screen.
78. Freedman, M. TimescaleDB vs InfluxDB: Purpose-Built Differently [Virtual Resource] / M. Freedman // Timescale Blog. - Access Mode : URL : https://blog.timescale.com/. - Title from Screen.
79. Kochkarov, I. Using SIMD for High Performance [Virtual Resource] / I. Kochkarov // QuestDB Blog. - Access Mode : URL : https://questdb.io/blog/. - Title from Screen.
80. TimescaleDB Documentation. Query Performance with Continuous Aggregates [Virtual Resource] // Timescale. - Access Mode : URL : https://docs.timescale.com/. - Title from Screen.
81. VictoriaMetrics. Storage Compression Benchmarks [Virtual Resource] // VictoriaMetrics. - Access Mode : URL : https://docs.victoriametrics.com/. - Title from Screen.
82. InfluxDB Documentation. Compression Ratios [Virtual Resource] // InfluxData. - Access Mode : URL : https://docs.influxdata.com/. - Title from Screen.
83. TimescaleDB Documentation. Native Compression Performance [Virtual Resource] // Timescale. - Access Mode : URL : https://docs.timescale.com/. - Title from Screen.
84. Інженерія програмного забезпечення [Текст] : навчальний посібник / В. І. Шинкаренко, О. В. Горбова, О. П. Іванов, В. О. Андрющенко, В. Я. Нечай ; Дніпропетр. нац. ун-т залізн. трансп. ім. акад. В. Лазаряна. - Дніпро, 2019. - 140 с.
85. Проектування інформаційних систем: Загальні питання теорії проектування ІС (конспект лекцій) [Електронний ресурс] : навч. посіб. для студ.

спеціальності 122 «Комп'ютерні науки» / КПІ ім. Ігоря Сікорського ; уклад.: О. С. Коваленко, Л. М. Добровська. - Київ : КПІ ім. Ігоря Сікорського, 2020.

86. Вєтлужських, М. В. Функціональні можливості систем керування базами даних часових рядів [Текст] / М. В. Вєтлужських, В. І. Шинкаренко // Information Technologies in Metallurgy and Machine Building : матеріали міжнар. наук.-техн. конф. - 2024. - С. 218-222. - Режим доступу : DOI : 10.34185/1991-7848.itmm.2024.01.038.

87. InfluxDB Documentation. InfluxQL and Flux Parity [Virtual Resource] // InfluxData. - Access Mode : URL : https://docs.influxdata.com/influxdb/cloud/reference/syntax/flux/flux-vs-influxql/. - Title from Screen. - Date of Access: 22 December 2025.

ДОДАТОК А

Тези

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
МІНІСТЕРСТВО ІНФРАСТРУКТУРИ УКРАЇНИ
УКРАЇНСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ НАУКИ ТА ТЕХНОЛОГІЙ
СХІДНИЙ НАУКОВИЙ ЦЕНТР ТРАНСПОРТНОЇ АКАДЕМІЇ НАУК

ABSTRACTS
OF THE XVII INTERNATIONAL CONFERENCE
«MODERN INFORMATION AND COMMUNICATION TECHNOLOGIES ON A TRANSPORT, IN INDUSTRY AND EDUCATION»
13-14, December, 2023

АВТОМАТИЗАЦІЯ
ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ
КІБЕРБЕЗПЕКА
КОМУНІКАЦІЯ
НАВЧАННЯ

СУЧАСНІ ІНФОРМАЦІЙНІ ТА КОМУНІКАЦІЙНІ ТЕХНОЛОГІЇ НА ТРАНСПОРТІ, В ПРОМИСЛОВОСТІ ТА ОСВІТІ

ТЕЗИ
XVII МІЖНАРОДНОЇ НАУКОВО-ПРАКТИЧНОЇ КОНФЕРЕНЦІЇ
13-14 ГРУДНЯ 2023

ДНІПРО
2023

Інструментальні засоби Time series database 59
Вєтлужських М. В., Шинкаренко В. І., Український державний університет науки і технологій, Україна

Автоматизація оцінки повноти тестування API за допомогою python-скрипту 60
Водянік Ю. О., Apriorit, Куроп'ятник О. С., Український державний університет науки та технологій

До проблеми визначення тривимірних об'єктів систем доповненої реальності 61
Гасанов Р.З., Скалозуб В.В. Український державний університет науки і технологій, Україна

Використання генетичного алгоритму для пошуку точки Штейнера на площині за допомогою кластеризації області пошуку 62
Глушков О.В., Український державний університет науки і технологій, Україна

Застосування штучного інтелекту для управління логістичними процесами 63
Демченко Є. Б., Дорош А. С., Український державний університет науки і технологій, Україна

Мультиагентне конструктивне моделювання часових рядів 64
Жадан А.А., Галушко О.В., Шинкаренко В.І., Український державний університет науки і технологій, Україна

railML ontology...65
Larysa Zhuchyi, railML.org, Dresden, Germany

Дослідження моделей оптичних перетворень негладких фрактальних поверхонь 66
Зайцев О. В., Шинкаренко В. І., Український державний університет науки і технологій, Україна

Застосування геоінформаційних систем у транспортній галузі 67
Зінов'єва О.Г., Таврійський державний агротехнологічний університет імені Дмитра Моторного, Україна

Система керування та контролю корпоративних баз даних у середовищі Lotus Notes 68
Івченко Ю.М., Український державний університет науки і технологій, Україна

Методи та засоби рефакторингу онтологій..69
Карповський Д.О., Шинкаренко В. І., Дніпровський державний університет науки і технологій, Україна

Використання Semantic Web у електронній комерції 70
Ковальчук К.І., Іскандарова-Мала А.О., Бабенко М.В., Дніпровський державний технічний університет, Україна

Ефективне розв'язування задачі про рюкзак 71
Косолап А. І., Дніпровський національний університет ім. О. Гончара, Україна

Дослідження ефективності сучасних методів оптимізації нейронних мереж 72
Костенко В. І., Жульковський О. О., Дніпровський державний технічний університет, Україна, Жульковська І. І., Університет митної справи та фінансів, Дніпро, Україна

Прогнозування результатів командних змагань на основі конструктивного підходу та методу аналізу ієрархій 73
Кумпан С.В., Шинкаренко В.І. Український державний університет науки і технологій

Інструментальні засоби Time series database

Вєтлужських М. В., Шинкаренко В. І., Український державний університет науки і технологій, Україна

У сучасному світі, де обсяги даних зростають з кожним днем, здатність ефективно обробляти та аналізувати часові ряди стає критично важливою. Time series databases, або бази даних часових рядів, знаходять застосування у широкому спектрі областей, від фінансових ринків до прогнозування погоди, від охорони здоров'я до управління міською інфраструктурою. Інструментальні засоби для цих баз даних дозволяють не лише зберігати великі обсяги даних, але й забезпечують можливості для їх швидкого аналізу та виявлення трендів і закономірностей. Це дослідження вбирає в себе ідеї та методології з кількох наукових дисциплін, що робить його міждисциплінарним. З одного боку, воно опирається на фундаментальні принципи комп'ютерних наук та інформаційних технологій, особливо в контексті розробки та оптимізації баз даних. З іншого боку, воно використовує прогресивні методи статистичного аналізу.

Представлена робота присвячена дослідженню інструментальних засобів баз даних часових рядів для подальшої структуризації інформації та розробки рекомендацій для застосування. Для виявлення ключових характеристик було обрано наступний набір найбільш популярних рішень: InfluxDB, Prometheus, Redis, DolphinDB, QuestDB, VictoriaMetrics, M3DB та KairosDB. Далі, проведено встановлення, тестування та класифікацію вибраних баз даних.

Time series databases можуть бути класифіковані на основі їх способу зберігання даних. Існують системи, що базуються на колонковому зберіганні (column-based), які оптимізовані для швидкого читання та агрегації великих обсягів даних. Інші системи використовують рядкове зберігання (row-based), яке краще підходить для транзакційних систем з частими записами.

Також системи можуть бути класифіковані за режимом обробки запитів. Існують системи реального часу, які забезпечують надшвидку обробку та відповідь на запити, і системи, орієнтовані на пакетну обробку, які краще підходять для аналізу великих наборів даних, але з деякою затримкою в обробці.

Ефективність запису в Time series databases часто досягається за рахунок використання оптимізованих алгоритмів стиснення та індексації, які мінімізують використання дискового простору та забезпечують швидкий доступ до даних.

Читання даних оптимізовано через індексування та кешування, що дозволяє швидко знаходити та аналізувати великі обсяги інформації, особливо при часто використовуваних запитах.

Системи часто використовують схеми реплікації та розподілу даних для забезпечення високої доступності та масштабування, а також стратегії резервного копіювання та відновлення для забезпечення надійності зберігання.

Далі, було досліджено аналітичні можливості вибраних баз даних часових рядів. В результаті виявлено, що сучасні бази даних часових рядів надають розширені можливості для аналізу даних, такі як агрегація, фільтрація, сегментація та візуалізація. Це дозволяє користувачам виконувати складні аналітичні запити.

В результаті дослідження, виявлено, що наявні рішення баз даних часових рядів мають широкий інструментарій для роботи з часовими рядами. Проведено класифікацію за способом зберігання, режимом обробки запитів, ефективності запису та читання даних, можливості масштабування та наявності аналітичних можливостей. Дане дослідження може бути використаним для вибору системи керування базами даних, а також для розробки нових інструментальних засобів баз даних часових рядів.

Міністерство освіти і науки
Український державний університет науки і технологій
Дніпровський державний технічний університет
Дніпровський національний університет імені Олеся Гончара
Національний технічний університет «Дніпровська політехніка»
Криворізький національний університет
Харківський національний університет радіоелектроніки
Херсонський національний технічний університет
Чорноморський державний університет імені П. Могили
Aalto University (Університет Аалто, Фінляндія)
American University of Central Asia (Бішкек, Киргизстан)
Tallinna Tehnikaülikool (Таллінн, Естонія)
AGH University of Science and Technology (Краків, Польща)
Politechnika Rzeszowska (Жешув, Польща)
Ariel University (Аріель, Ізраїль)
Michigan State University (Іст-Лансінґ, США)
Leibniz Universitat Hannover, Institute of Photogrammetry and Geoinformation
(Ганновер, Німеччина)

МАТЕРІАЛИ
Міжнародної науково-технічної конференції
ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ в
МЕТАЛУРГІЇ та МАШИНОБУДУВАННІ

MATERIALS
of Scientific and Technical International Conference
INFORMATION TECHNOLOGY IN
METALLURGY AND MACHINE ENGINEERING

10 - квітня 2024 року

м. Дніпро

СУЧАСНИЙ СТАН ВИКОРИСТАННЯ ЕЛЕКТРОННОГО ЦИФРОВОГО ПІДПИСУ

Kharun V. 503-507

PDF (Англійська)

ВИЯВЛЕННЯ ШУМІВ У ФРАКТАЛЬНИХ ЧАСОВИХ РЯДАХ ЗА ДОПОМОГОЮ МАШИННОГО НАВЧАННЯ

Lyudmyla Kirichenko, Mykyta Avsitidiiskyi 194-197

PDF (Англійська)

ДОСЛІДЖЕННЯ РІВНЯ ВІДПОВІДНОСТІ МІКРОКОНТРОЛЕРА ESP32 МІЖНАРОДНИМ СТАНДАРТАМ З КІБЕРНЕТИЧНОЇ БЕЗПЕКИ ІНТЕРНЕТУ РЕЧЕЙ

Valeriy Mazurenko 367-371

PDF (Англійська)

КАМЕРНА ПІЧ ДЛЯ МОДЕЛЮВАННЯ ТЕРМІЧНОЇ ОБРОБКИ ВУГЛЕЦЕВИХ МАТЕРІАЛІВ У ЕЛЕКТРОТЕРМІЧНОМУ КИПЛЯЧОМУ ШАРІ

Hubynskyi Semen, Sibyr Artem, Fedorov Serhii, Foris Oleksiy 33-38

PDF (Англійська)

МЕТОДИКА ІНТЕРПОЛЯЦІЇ ТА ЕКСТРАПОЛЯЦІЇ ДАНИХ ПРИ НЕРАВНОМІРНОМУ КРОКУ ЗМІНИ ПАРАМЕТРІВ ЕКСПЕРИМЕНТУ

Mykhailo Poliakov, Volodymyr Vasylevskyi, Oleksii Poliakov 526-529

PDF (Англійська)

ФУНКЦІОНАЛЬНІ МОЖЛИВОСТІ СИСТЕМ КЕРУВАННЯ БАЗАМИ ДАНИХ ЧАСОВИХ РЯДІВ

Michael Vietluzhskykh, Viktor Shynkarenko 218-222

PDF (Англійська)

РЕФАКТОРИНГ КРОС-ПЛАТФОРМНИХ ЗАСТОСУНКІВ З ВИКОРИСТАНЯМ ШТУЧНОГО ІНТЕЛЕКТУ

Oleksandr Syrota, Horiachkin Vadym 393-397

PDF (Англійська)

ШВИДКІСТЬ КОРОЗІЇ НИЗЬКОВУГЛЕЦЕВИХ ТРУБНИХСТАЛЕЙ У РІЗНИХ АГРЕСИВНИХ СЕРЕДОВИЩАХ

Dmytro Petryna 65-68

PDF (Англійська)

ІННОВАЦІЙНІ ПІДХОДИ ПІДВИЩЕННЯ ЯКОСТІ НАВЧАЛЬНОГО ПРОЦЕСУ ЧЕРЕЗ ДОСЛІДЖЕННЯ МОТИВАЦІЙНИХ ВАЖЕЛІВ ВИБОРУ МАЙБУТНЬОЇ ПРОФЕСІЇ СЕРЕД МОЛОДІ

Lushnya K., Zaytseva T., Siryk S. 545-549

PDF (Англійська)

ВИКОРИСТАННЯ ПОКАЗНИКІВ JRQA ДАНИХ ЕЛЕКТРОЕНЦЕФАЛОГРАФІЇ ЩОДО ЇХ ТИПИЗАЦІЇ

Vadym Zaytsev, Oleksandr Khizha 240-243

PDF (Англійська)

International scientific and technical conference
Information Technologies in Metallurgy and Machine building – ITMM 2024

DOI: 10.34185/1991-7848.itmm.2024.01.038

ФУНКЦІОНАЛЬНІ МОЖЛИВОСТІ СИСТЕМ КЕРУВАННЯ БАЗАМИ ДАНИХ ЧАСОВИХ РЯДІВ

Вєтлужських М.В., Шинкаренко В. І.

Український державний університет науки і технологій, Україна

Анотація. *Представена робота надає всебічний огляд можливостей баз даних часових рядів, з акцентом на InfluxDB – провідну базу даних часових рядів, відому своєю ефективністю та підтримкою складних аналізів за допомогою мови запитів Flux, включно з агрегацією даних, фільтрацією, виявленням трендів та прогнозуванням. Практичне застосування InfluxDB демонструється через різні приклади, показуючи її універсальність у виконанні складних операцій з даними, таких як фільтрація даних, виявлення значних змін у даних та виконання стохастичного аналізу. Ці функціональності висвітлюють потенціал InfluxDB у широкому спектрі застосувань, від телеметрії IoT до фінансового аналізу, підкреслюючи критичну роль баз даних часових рядів у сучасних стратегіях управління та аналізу даних.*

Ключові слова: *часовий ряд, бази даних, аналіз часових рядів*

Вступ. Наряду з відомими та широко розповсюдженими системами керування реляційними базами даних (реляційними СКБД), мають застосування і інші підходи збереження даних які вирізняються способом їх збереження і обробки та спеціалізацією.

Реляційні бази даних та бази даних часових рядів (Time series database) відрізняються по кільком основним аспектам. Реляційні бази даних орієнтовані на зберігання інтеррелаційних даних у таблицях з рядками та стовпцями. Вони підтримують складні SQL-запити для обробки даних і ідеально підходять для додатків, які потребують складних транзакцій, таких як банківські системи або системи управління ресурсами підприємства. З іншого боку, бази даних часових рядів спеціалізуються на зберіганні і аналізі даних, які змінюються з часом, з використанням часових міток для ефективного виконання запитів за часовими інтервалами. Вони оптимальні для моніторингу, відстеження та аналізу даних в реальному часі, наприклад, для фінансових котирувань, телеметрії IoT або метеорологічних даних.

Основна частина.Розглянемо функціональні можливості СКБД часових рядів InfluxDB, яку було обрано за рейтингом [1], де ця СКБД займає перше

ДОДАТОК Б

Технічне завдання

ЗАТВЕРДЖУЮ
Перший проректор Українського державного університету
науки і технологій
Анатолій РАДКЕВИЧ

ІНСТРУМЕНТАРІЙ ДЛЯ ДОСЛІДЖЕННЯ ЧАСОВИХ ХАРАКТЕРИСТИК
СПЕЦІАЛІЗОВАНИХ БАЗ ДАНИХ ЧАСОВИХ РЯДІВ

Технічне завдання

ЛИСТ ЗАТВЕРДЖЕННЯ

44165850.01536 – 01 – ЛЗ

Завідувач кафедри КІТ
Вадим ГОРЯЧКІН
Керівник розробки
Віктор ШИНКАРЕНКО
Виконавець
Михайло ВЄТЛУЖСЬКИХ
Нормоконтролер
Світлана ВОЛКОВА

2025

ЗАТВЕРДЖЕНО

44165850.01536 – 01

«ІНСТРУМЕНТАРІЙ ДЛЯ ДОСЛІДЖЕННЯ ЧАСОВИХ ХАРАКТЕРИСТИК СПЕЦІАЛІЗОВАНИХ БАЗ ДАНИХ ЧАСОВИХ РЯДІВ»

Технічне завдання

44165850.01536 – 01

Листів 8

2025

ЗМІСТ

ВСТУП 3

1. Призначення розробки 4

1.1. Функціональне призначення 4

1.2. Експлуатаційне призначення 4

2. Вимоги до програми або програмного продукту 5

2.1 Функціональні вимоги 5

2.2 Нефункціональні вимоги 6

3. Бібліографічний список 8

ВСТУП

Бази даних часових рядів (Time Series Databases, TSDB) стали критично важливим компонентом сучасних інформаційних систем, забезпечуючи ефективне зберігання та обробку великих обсягів даних з часовими мітками. Такі бази даних широко використовуються в системах моніторингу інфраструктури, IoT-додатках, фінансовій аналітиці, промисловому обладнанні та інших галузях, де необхідна швидка обробка послідовностей вимірювань у часі.

Вибір оптимальної TSDB для конкретного сценарію використання є складним завданням, що потребує об'єктивного порівняння продуктивності різних рішень. На ринку представлено множину баз даних часових рядів (InfluxDB, TimescaleDB, QuestDB, VictoriaMetrics та інші), кожна з яких має свої особливості архітектури, механізми зберігання даних та оптимізації запитів.

Існуюча проблема полягає у відсутності уніфікованого інструментарію для автоматизованого тестування та порівняльного аналізу продуктивності TSDB. Більшість досліджень проводяться вручну або з використанням фрагментарних рішень, що ускладнює отримання об'єктивних та відтворюваних результатів.

Розробка комплексного інструментарію для дослідження часових характеристик баз даних часових рядів дозволить автоматизувати процес тестування, забезпечити ізоляцію тестового середовища, отримати точні метрики продуктивності та надати зручний інтерфейс для аналізу результатів. Це сприятиме прийняттю обґрунтованих рішень при виборі TSDB для різних прикладних задач.

1. ПРИЗНАЧЕННЯ РОЗРОБКИ

1.1. Функціональне призначення

Програмний комплекс призначений для автоматизованого тестування та порівняльного аналізу продуктивності баз даних часових рядів. Система забезпечує комплексне дослідження чотирьох популярних TSDB: InfluxDB, TimescaleDB, QuestDB та VictoriaMetrics.

1.2. Експлуатаційне призначення

Ця розробка в майбутньому допоможе краще розуміти особливості роботи різних TSDB, їх сильні та слабкі сторони, що є важливим для проектування високопродуктивних систем обробки часових рядів.

2. ВИМОГИ ДО ПРОГРАМИ

2.1.Функціональні вимоги

Система тестування продуктивності повинна забезпечувати наступне.

- Автоматизоване створення конфігураційних файлів Docker Compose для кожної бази даних часових рядів;
- Запуск та зупинку контейнерів баз даних з перевіркою їх готовності;
- Генерацію синтетичних наборів даних часових рядів із заданими параметрами;
- Виконання тестів запису даних з вимірюванням затримки кожної операції;
- Виконання тестів читання даних з різними типами запитів (вибірка, фільтрація, агрегація);
- Збір метрик використання системних ресурсів (оперативна пам'ять, дисковий простір);
- Обчислення статистичних показників продуктивності (операції на секунду, середня затримка, відсоток помилок);
- Збереження результатів тестування у структурованому форматі JSON;
- Повне очищення тестового середовища після завершення експериментів;
- Можливість вибору конкретних баз даних для тестування через параметри командного рядка.

Веб-додаток візуалізації повинен забезпечувати наступне.

- Завантаження даних результатів тестування з JSON файлу;
- Візуальне представлення метрик продуктивності через інтерактивні графіки;
- Відображення порівняльних стовпчастих діаграм для операцій запису та читання;

- Відображення порівняльних стовпчастих діаграм для затримок виконання операцій;
- Відображення кругових діаграм розподілу використання ресурсів;
- Табличне представлення всіх метрик з детальною інформацією;
- Динамічне оновлення компонентів інтерфейсу без перезавантаження сторінки;
- Автоматичне визначення та відображення найкращих показників продуктивності;
- Можливість перемикання між різними типами візуалізацій;
- Оновлення даних при зміні результатів тестування.

2.2. Нефункціональні вимоги

Веб-додаток та система тестування повинні забезпечувати:

- Продуктивність. Ефективне використання системних ресурсів при запуску множинних контейнерів, швидке виконання тестів, оптимізоване завантаження та відображення результатів у веб-інтерфейсі;
- Надійність. Стабільність роботи при тестуванні різних баз даних, коректна обробка помилок запуску контейнерів, толерантність до збоїв окремих компонентів, гарантоване очищення ресурсів після завершення тестів;
- Сумісність. Підтримка роботи на різних операційних системах (Linux, Windows, macOS), сумісність з різними версіями Docker, коректне відображення веб-інтерфейсу в сучасних браузерах (Chrome, Firefox, Safari, Edge);
- Масштабованість. Модульна архітектура з можливістю легкого додавання нових баз даних для тестування, розширення переліку

метрик без зміни базової структури, можливість додавання нових типів візуалізацій;

- Супроводжуваність. Якість коду відповідає стандартам Python (PEP 8), код легко читається і підтримується, наявність docstrings для всіх модулів та функцій, чіткий поділ на модулі з визначеними інтерфейсами;
- Тестованість. Можливість проведення автоматизованих тестів окремих компонентів, відтворюваність результатів експериментів, детальне логування процесу тестування.

3. БІБЛІОГРАФІЧНИЙ СПИСОК

1. Особливості баз даних часових рядів [Електронний ресурс]. – Режим доступу: https://www.influxdata.com/time-series-database/ – (Дата звернення 15.09.2025).
2. PEP 8 – Style Guide for Python Code [Електронний ресурс]. – Режим доступу: https://peps.python.org/pep-0008/ – (Дата звернення 15.09.2025).
3. Web Content Accessibility Guidelines (WCAG) [Електронний ресурс]. – Режим доступу: https://www.w3.org/WAI/standards-guidelines/wcag/ – (Дата звернення 15.09.2025).
4. Time Series Database: How Is It Different? [Електронний ресурс]. – Режим доступу: https://www.timescale.com/blog/what-the-heck-is-time-series-data-and-why-do-i-need-a-time-series-database-dcf3b1b18563/ – (Дата звернення 15.09.2025).
5. Docker Best Practices [Електронний ресурс]. – Режим доступу: https://docs.docker.com/develop/dev-best-practices/ – (Дата звернення 15.09.2025).

ДОДАТОК В

Специфікація

ЗАТВЕРДЖУЮ
Перший проректор Українського державного університету науки і технологій
Анатолій РАДКЕВИЧ

ІНСТРУМЕНТАРІЙ ДЛЯ ДОСЛІДЖЕННЯ ЧАСОВИХ ХАРАКТЕРИСТИК СПЕЦІАЛІЗОВАНИХ БАЗ ДАНИХ ЧАСОВИХ РЯДІВ

Специфікація

ЛИСТ ЗАТВЕРДЖЕННЯ

44165850.01536 – 01–ЛЗ

Завідувач кафедри КІТ
Вадим ГОРЯЧКІН

Керівник розробки
Віктор ШИНКАРЕНКО

Виконавець
Михайло ВЄТЛУЖСЬКИХ

Нормоконтролер
Світлана ВОЛКОВА

2025

ЗАТВЕРДЖЕНО
44165850.01536 – 01

ІНСТРУМЕНТАРІЙ ДЛЯ ДОСЛІДЖЕННЯ ЧАСОВИХ ХАРАКТЕРИСТИК СПЕЦІАЛІЗОВАНИХ БАЗ ДАНИХ ЧАСОВИХ РЯДІВ

Специфікація

44165850.01536 – 01

Листів 2

2025

1. СПЕЦИФІКАЦІЇ

Таблиця 1.1 – Специфікації

Позначення	Найменування	Примітка
	Документація	
44165850.01536 – 01-ЛЗ	Лист затвердження	
44165850.01536 – 01 01 01	Технічне завдання	
44165850.01536 – 01 12 01	Текст програми	
44165850.01536 – 01 13 01 – ЛЗ	Лист затвердження	
44165850.01536 – 01 I3 01 – ЛЗ	Лист затвердження	
44165850.01536 – 01 I3 01	Керівництво користувача	

ДОДАТОК Г

Керівництво користувача

ЗАТВЕРДЖУЮ

Перший проректор Українського державного університету науки і технологій

Анатолій РАДКЕВИЧ

ІНСТРУМЕНТАРІЙ ДЛЯ ДОСЛІДЖЕННЯ ЧАСОВИХ ХАРАКТЕРИСТИК СПЕЦІАЛІЗОВАНИХ БАЗ ДАНИХ ЧАСОВИХ РЯДІВ

Керівництво користувача

ЛИСТ ЗАТВЕРДЖЕННЯ

44165850.01536 – 01 ІЗ 01–ЛЗ

Завідувач кафедри КІТ

Вадим ГОРЯЧКІН

Керівник розробки

Віктор ШИНКАРЕНКО

Виконавець

Михайло ВЄТЛУЖСЬКИХ

Нормоконтролер

Світлана ВОЛКОВА

2025

ЗАТВЕРДЖЕНО

44165850.01536 – 01 І3 01

ІНСТРУМЕНТАРІЙ ДЛЯ ДОСЛІДЖЕННЯ ЧАСОВИХ ХАРАКТЕРИСТИК СПЕЦІАЛІЗОВАНИХ БАЗ ДАНИХ ЧАСОВИХ РЯДІВ

Керівництво користувача

44165850.01536 – 01 І3 01

Листів 8

2026

ЗМІСТ

ВСТУП 3
1 ПРИЗНАЧЕННЯ ТА УМОВИ ЗАСТОСУВАННЯ 4
2 ПІДГОТОВКА ДО РОБОТИ 5
3 ОПИС ОПЕРАЦІЙ 6
4 АВАРІЙНІ СИТУАЦІЇ 7
5 РЕКОМЕНДАЦІЇ ЩОДО ЗАСВОЄННЯ 8

ВСТУП

Програмний комплекс «Інструментарій для дослідження часових характеристик баз даних часових рядів» призначений для автоматизованого тестування та порівняльного аналізу продуктивності чотирьох популярних баз даних часових рядів: InfluxDB, TimescaleDB, QuestDB та VictoriaMetrics.

Система забезпечує комплексне benchmark-тестування з вимірюванням швидкості запису та читання даних, аналізом використання системних ресурсів та візуалізацією результатів через веб-інтерфейс.

Керівництво призначене для користувачів, які мають базові знання роботи з командним рядком Linux та розуміють принципи роботи з Docker-контейнерами.

Перед початком роботи рекомендується ознайомитися з документами: Технічне завдання (44165850.01536 – 01 12 01), Опис програми (44165850.01536 – 01 13 01).

1. ПРИЗНАЧЕННЯ ТА УМОВИ ЗАСТОСУВАННЯ

Система призначена для автоматизованого порівняльного аналізу продуктивності баз даних часових рядів з метою прийняття обґрунтованих архітектурних рішень при проектуванні систем моніторингу, IoT-додатків та аналітичних платформ.

Основні функції системи:

- автоматизоване розгортання Docker-контейнерів з базами даних;
- генерація синтетичних наборів тестових даних;
- виконання тестів запису та читання даних з вимірюванням затримок;
- збір метрик використання оперативної пам'яті та дискового простору;
- візуалізація результатів через інтерактивний веб-інтерфейс.

Вимоги до технічних засобів:

- операційна система: Linux (Ubuntu 20.04+) або Windows 10/11 з WSL2;
- оперативна пам'ять: не менше 8 ГБ (рекомендовано 16 ГБ);
- вільний дисковий простір: не менше 20 ГБ;
- Docker Engine версії 20.10 або новіше;
- Docker Compose версії 2.0 або новіше;
- Python версії 3.9 або новіше;
- веб-браузер: Chrome, Firefox або Edge останньої версії.

2. ПІДГОТОВКА ДО РОБОТИ

Для підготовки системи до роботи виконайте наступні дії.

Перевірте наявність Docker. Відкрийте термінал та виконайте команду, наведену у лістингу 2.1.

```
docker -version
```

Лістинг 2.1 — Перевірка версії Docker

Переконайтеся, що Docker Compose встановлено. Команду перевірки наведено у лістингу 2.2.

```
docker compose version
```

Лістинг 2.2 — Перевірка версії Docker Compose

Встановіть залежності Python. Перейдіть до директорії проекту та виконайте команду, наведену у лістингу 2.3.

```
pip install -r requirements.txt
```

Лістинг 2.3 — Встановлення залежностей Python

Перевірте працездатність системи, запустивши тест однієї бази даних. Команду запуску наведено у лістингу 2.4.

```
python benchmark.py -databases influxdb
```

Лістинг 2.4 — Запуск тестування для перевірки працездатності системи

Якщо тест завершився успішно, система готова до роботи.

3. ОПИС ОПЕРАЦІЙ

3.1. Запуск повного тестування

Для запуску тестування всіх баз даних виконайте команду, наведену у лістингу 3.1.

```
python benchmark.py
```

Лістинг 3.1 — Запуск повного тестування всіх баз даних

Для тестування окремих баз даних використовуйте параметр -databases. Приклад команди наведено у лістингу 3.2.

```
python benchmark.py -databases influxdb questdb
```

Лістинг 3.2 — Запуск тестування для обраних баз даних

3.2.Перегляд результатів

Запустіть веб-сервер візуалізації. Команду запуску наведено у лістингу 3.3.

```
python app.py
```

Лістинг 3.3 — Запуск веб-сервера візуалізації результатів

Відкрийте веб-браузер та перейдіть за адресою http://localhost:5000. Оберіть тип візуалізації: стовпчасті діаграми продуктивності, кругові діаграми використання ресурсів або таблицю з детальними метриками.

3.3. Експорт результатів

Результати тестування автоматично зберігаються у файл results.json у директорії проекту. Файл містить усі метрики у структурованому форматі.

4. АВАРІЙНІ СИТУАЦІЇ

Проблема	Рішення
Контейнер не запускається	Перевірте, чи запущений Docker демон командою: docker info. Перезапустіть Docker службу.
Помилка підключення до бази даних	Зачекайте завершення ініціалізації бази даних. Перевірте healthcheck статус: docker ps
Недостатньо пам'яті	Завершіть непотрібні процеси. Тестуйте бази даних по черзі, а не одночасно.
Веб-інтерфейс не відкривається	Переконайтеся, що порт 5000 не зайнятий іншим додатком. Змініть порт у app.py.
Тест зависає	Натисніть Ctrl+C для зупинки. Виконайте очищення: docker compose down -v

Для відновлення після збою виконайте повне очищення тестового середовища. Команду очищення наведено у лістингу 4.1.

```
docker compose down -v -remove-orphans
```

Лістинг 4.1 — Очищення тестового середовища Docker

5. РЕКОМЕНДАЦІЇ ЩОДО ЗАСВОЄННЯ

Для ознайомлення з системою рекомендується виконати контрольний приклад.

Запустіть тестування однієї бази даних (QuestDB). Команду запуску наведено у лістингу 5.1.

```
python benchmark.py -databases questdb
```

Лістинг 5.1 — Запуск тестування бази даних QuestDB

Дочекайтеся завершення тестування (приблизно 2-3 хвилини). У консолі відобразяться повідомлення про хід виконання тестів.

Запустіть веб-інтерфейс візуалізації. Команду запуску наведено у лістингу 5.2.

```
python app.py
```

Лістинг 5.2 — Запуск веб-інтерфейсу візуалізації

Відкрийте браузер за адресою http://localhost:5000 та перегляньте результати на графіках та у таблиці.

Очікувані результати контрольного прикладу: успішне відображення метрик продуктивності QuestDB, включаючи швидкість запису (операцій/сек), середню затримку (мс), швидкість читання та використання ресурсів.

Після успішного виконання контрольного прикладу виконайте повне тестування всіх чотирьох баз даних для отримання порівняльних результатів.

ДОДАТОК Д

Текст програми

ЗАТВЕРДЖУЮ

Перший проректор Українського державного університету науки і технологій

Анатолій РАДКЕВИЧ

ІНСТРУМЕНТАРІЙ ДЛЯ ДОСЛІДЖЕННЯ ЧАСОВИХ ХАРАКТЕРИСТИК СПЕЦІАЛІЗОВАНИХ БАЗ ДАНИХ ЧАСОВИХ РЯДІВ

Текст програми

ЛИСТ ЗАТВЕРДЖЕННЯ

44165850.01536– 01 12 01–ЛЗ

Завідувач кафедри КІТ

__________Вадим ГОРЯЧКІН

Керівник розробки

__________ Віктор ШИНКАРЕНКО

Виконавець

__________ Михайло ВЄТЛУЖСЬКИХ

Нормоконтролер

__________Світлана ВОЛКОВА

2026

ЗАТВЕРДЖЕНО

44165850.01536 – 01 12 01

ІНСТРУМЕНТАРІЙ ДЛЯ ДОСЛІДЖЕННЯ ЧАСОВИХ ХАРАКТЕРИСТИК СПЕЦІАЛІЗОВАНИХ БАЗ ДАНИХ ЧАСОВИХ РЯДІВ

Текст програми

44165850.01536– 01 12 01

Листів 20

2026

АНОТАЦІЯ

Вихідний код програми дослідження продуктивності спеціалізованих баз даних часових рядів (TSDB Performance Benchmark System).

Програма призначена для автоматизованого тестування продуктивності баз даних часових рядів InfluxDB, TimescaleDB, QuestDB та VictoriaMetrics. Система виконує benchmark-тестування операцій запису та читання, вимірює використання системних ресурсів та надає візуалізацію результатів через веб-інтерфейс.

Програма реалізована мовою Python 3.11 з використанням бібліотек Docker SDK, Flask, HTMX та Chart.js. Для ізоляції тестового середовища використовується Docker-контейнеризація.

Структура програми включає такі основні модулі: benchmark.py (головний модуль запуску тестів), docker_manager.py (менеджер Docker-контейнерів), test_suite.py (набір тестів продуктивності), адаптери взаємодії з базами даних (influxdb_tester.py, timescaledb_tester.py, questdb_tester.py, victoriametrics_tester.py), visualizer.py (веб-сервер візуалізації результатів).

```
=== ./benchmark.py ===
#!/usr/bin/env python3
"""
Time Series Database Performance Tester

This script automatically tests multiple time series databases for:
- Write performance (inserts/second)
- Read performance (queries/second)
- Memory footprint
- Database size on disk

Supported databases:
- InfluxDB
- TimescaleDB
- QuestDB
- VictoriaMetrics

Requirements:
- Docker and Docker Compose
- Python packages: docker, psycopg2-binary, requests, influxdb-client
"""
import traceback
from src.test_suite import PerformanceTestSuite


def main():
    """Main execution function"""
    import argparse

    parser = argparse.ArgumentParser(description='Time Series Database Performance Tester')
    parser.add_argument('-databases', nargs='+',
                        choices=['influxdb', 'timescaledb', 'questdb', 'victoriametrics'],
                        default=['influxdb', 'timescaledb', 'questdb', 'victoriametrics'],
                        help='Databases to test')
    parser.add_argument('-output', default='tsdb_performance_results.json',
                        help='Output file for results')

    args = parser.parse_args()

    # Create test suite
    suite = PerformanceTestSuite()

    try:
        # Run tests
        results = suite.run_test_suite(args.databases)

        # Display results
        suite.print_results()

        # Save results
        suite.save_results(args.output)

    except KeyboardInterrupt:
        print("\nTest interrupted by user")
    except Exception as e:
        traceback.print_exc()
        print(f"Test suite error: {str(e)}")
    finally:
        print("\nTest suite completed")


if __name__ == "__main__":
    main()
=== ./__init__.py ===

=== ./static/style.css ===
/* Reset and Base Styles */
* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
}

body {
    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
    line-height: 1.6;
    color: #333;
    background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
    min-height: 100vh;
}

/* Dashboard Layout */
.dashboard {
    min-height: 100vh;
    display: flex;
    flex-direction: column;
}

/* Header */
.dashboard-header {
    background: rgba(255, 255, 255, 0.95);
    backdrop-filter: blur(10px);
    border-bottom: 1px solid rgba(255, 255, 255, 0.2);
    padding: 1rem 0;
    box-shadow: 0 2px 20px rgba(0, 0, 0, 0.1);
}

.header-content {
    max-width: 1200px;
    margin: 0 auto;
    padding: 0 2rem;
}

.dashboard-header h1 {
    color: #2c3e50;
    font-size: 2.5rem;
    font-weight: 700;
    margin-bottom: 0.5rem;
}

.dashboard-header h1 i {
    color: #667eea;
    margin-right: 1rem;
}

.subtitle {
    color: #7f8c8d;
    font-size: 1.1rem;
    font-weight: 300;
}

/* Main Content */
.dashboard-main {
    flex: 1;
    max-width: 1200px;
    margin: 0 auto;
```

```
    padding: 2rem;
    width: 100%;
}

/* Control Panel */
.control-panel {
    background: rgba(255, 255, 255, 0.95);
    backdrop-filter: blur(10px);
    border-radius: 15px;
    padding: 1.5rem;
    margin-bottom: 2rem;
    box-shadow: 0 8px 32px rgba(0, 0, 0, 0.1);
    display: flex;
    justify-content: space-between;
    align-items: center;
    flex-wrap: wrap;
    gap: 1rem;
}

.control-group label {
    font-weight: 600;
    color: #2c3e50;
    margin-right: 1rem;
}

.tab-buttons {
    display: flex;
    gap: 0.5rem;
    flex-wrap: wrap;
}

.tab-btn {
    background: #f8f9fa;
    border: 2px solid #e9ecef;
    color: #495057;
    padding: 0.75rem 1.5rem;
    border-radius: 10px;
    cursor: pointer;
    transition: all 0.3s ease;
    font-weight: 500;
    display: flex;
    align-items: center;
    gap: 0.5rem;
}

.tab-btn:hover {
    background: #e9ecef;
    border-color: #667eea;
    transform: translateY(-2px);
}

.tab-btn.active {
    background: linear-gradient(135deg, #667eea,
#764ba2);
    color: white;
    border-color: #667eea;
    box-shadow: 0 4px 15px rgba(102, 126, 234, 0.3);
}

.refresh-btn {
    background: linear-gradient(135deg, #28a745,
#20c997);
    color: white;
    border: none;
    padding: 0.75rem 1.5rem;
    border-radius: 10px;
    cursor: pointer;
    font-weight: 500;
    display: flex;
    align-items: center;
    gap: 0.5rem;
    transition: all 0.3s ease;
}

.refresh-btn:hover {
    transform: translateY(-2px);
    box-shadow: 0 4px 15px rgba(40, 167, 69, 0.3);
}

/* Chart Section */
.chart-section {
    background: rgba(255, 255, 255, 0.95);
    backdrop-filter: blur(10px);
    border-radius: 15px;
    padding: 2rem;
    margin-bottom: 2rem;
    box-shadow: 0 8px 32px rgba(0, 0, 0, 0.1);
    min-height: 500px;
}

.chart-wrapper {
    width: 100%;
    height: 100%;
}

.chart-header {
    text-align: center;
    margin-bottom: 2rem;
}

.chart-header h2 {
    color: #2c3e50;
    font-size: 1.8rem;
    margin-bottom: 0.5rem;
}

.chart-header h2 i {
    color: #667eea;
    margin-right: 0.5rem;
}

.chart-header p {
    color: #7f8c8d;
    font-size: 1rem;
}

.chart-content {
    position: relative;
    height: 400px;
}

/* Stats Grid */
.stats-grid {
    display: grid;
    grid-template-columns: repeat(auto-fit,
minmax(250px, 1fr));
    gap: 1.5rem;
    margin-bottom: 2rem;
}

.stat-card {
    background: rgba(255, 255, 255, 0.95);
    backdrop-filter: blur(10px);
    border-radius: 15px;
```

```
    padding: 1.5rem;
    box-shadow: 0 8px 32px rgba(0, 0, 0, 0.1);
    transition: transform 0.3s ease;
    display: flex;
    align-items: center;
    gap: 1rem;
}

.stat-card:hover {
    transform: translateY(-5px);
}

.stat-icon {
    width: 60px;
    height: 60px;
    border-radius: 12px;
    background: linear-gradient(135deg, #667eea, #764ba2);
    display: flex;
    align-items: center;
    justify-content: center;
    color: white;
    font-size: 1.5rem;
}

.stat-content h3 {
    color: #2c3e50;
    font-size: 0.9rem;
    margin-bottom: 0.5rem;
    text-transform: uppercase;
    letter-spacing: 0.5px;
}

.stat-value {
    color: #667eea;
    font-size: 1.2rem;
    font-weight: 700;
    margin-bottom: 0.25rem;
}

.stat-detail {
    color: #7f8c8d;
    font-size: 0.9rem;
}

/* Table Styles */
.table-content {
    width: 100%;
}

.table-wrapper {
    overflow-x: auto;
    margin-bottom: 2rem;
    border-radius: 10px;
    box-shadow: 0 4px 20px rgba(0, 0, 0, 0.1);
}

.performance-table {
    width: 100%;
    border-collapse: collapse;
    background: white;
    border-radius: 10px;
    overflow: hidden;
}

.performance-table th {
    background: linear-gradient(135deg, #667eea, #764ba2);
    color: white;
    padding: 1rem;
    text-align: left;
    font-weight: 600;
    font-size: 0.9rem;
    text-transform: uppercase;
    letter-spacing: 0.5px;
}

.performance-table td {
    padding: 1rem;
    border-bottom: 1px solid #f1f3f4;
}

.performance-table tbody tr:hover {
    background: #f8f9fa;
}

.performance-table tbody tr:last-child td {
    border-bottom: none;
}

.db-name {
    font-weight: 600;
}

.db-badge {
    display: inline-block;
    padding: 0.5rem 1rem;
    border-radius: 20px;
    font-size: 0.8rem;
    font-weight: 600;
    text-transform: uppercase;
    letter-spacing: 0.5px;
}

.db-influxdb { background: #ff6b6b; color: white; }
.db-timescaledb { background: #4ecdc4; color: white; }
.db-questdb { background: #45b7d1; color: white; }
.db-victoriametrics { background: #96ceb4; color: white; }

.metric-value {
    font-family: 'Courier New', monospace;
    text-align: right;
    font-weight: 500;
}

.success-badge {
    background: #28a745;
    color: white;
    padding: 0.25rem 0.5rem;
    border-radius: 12px;
    font-size: 0.8rem;
    font-weight: 600;
}

.error-badge {
    background: #dc3545;
    color: white;
    padding: 0.25rem 0.5rem;
    border-radius: 12px;
    font-size: 0.8rem;
    font-weight: 600;
}
```

```
/* Performance Insights */
.performance-insights h3 {
  color: #2c3e50;
  margin-bottom: 1.5rem;
  font-size: 1.5rem;
}

.insights-grid {
  display: grid;
  grid-template-columns: repeat(auto-fit,
minmax(250px, 1fr));
  gap: 1rem;
}

.insight-card {
  background: #f8f9fa;
  border-radius: 10px;
  padding: 1.5rem;
  text-align: center;
  border-left: 4px solid #667eea;
}

.insight-card i {
  font-size: 2rem;
  color: #667eea;
  margin-bottom: 1rem;
}

.insight-card h4 {
  color: #2c3e50;
  margin-bottom: 0.5rem;
  font-size: 1rem;
}

.insight-card p {
  color: #7f8c8d;
  font-size: 0.9rem;
}

.insight-card strong {
  color: #667eea;
}

/* Loading States */
.loading {
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  height: 400px;
  color: #7f8c8d;
  font-size: 1.2rem;
}

.loading i {
  font-size: 2rem;
  margin-bottom: 1rem;
  color: #667eea;
}

.loading-small {
  display: flex;
  align-items: center;
  justify-content: center;
  height: 100px;
  color: #7f8c8d;
}

/* Footer */
.dashboard-footer {
  background: rgba(255, 255, 255, 0.95);
  backdrop-filter: blur(10px);
  border-top: 1px solid rgba(255, 255, 255, 0.2);
  text-align: center;
  padding: 1rem;
  color: #7f8c8d;
  font-size: 0.9rem;
}

/* Responsive Design */
@media (max-width: 768px) {
  .dashboard-main {
    padding: 1rem;
  }

  .control-panel {
    flex-direction: column;
    align-items: stretch;
    text-align: center;
  }

  .tab-buttons {
    justify-content: center;
  }

  .tab-btn {
    flex: 1;
    min-width: 0;
    font-size: 0.9rem;
  }

  .dashboard-header h1 {
    font-size: 2rem;
  }

  .chart-content {
    height: 300px;
  }

  .stats-grid {
    grid-template-columns: 1fr;
  }

  .insights-grid {
    grid-template-columns: 1fr;
  }
}

@media (max-width: 480px) {
  .dashboard-header h1 {
    font-size: 1.5rem;
  }

  .subtitle {
    font-size: 1rem;
  }

  .tab-btn {
    padding: 0.5rem 1rem;
    font-size: 0.8rem;
  }

  .chart-content {
```

```
        height: 250px;
    }
}

/* HTMX Loading Indicators */
.htmx-request .loading {
    opacity: 0.8;
}

.htmx-request.loading {
    opacity: 0.8;
}

/* Chart.js Custom Styles */
canvas {
    max-height: 400px !important;
}

/* Animation */
@keyframes fadeIn {
    from { opacity: 0; transform: translateY(20px); }
    to { opacity: 1; transform: translateY(0); }
}

.chart-wrapper, .stat-card, .control-panel, .chart-section
{
    animation: fadeIn 0.5s ease-out;
}

/* Scrollbar Styling */
::-webkit-scrollbar {
    width: 8px;
    height: 8px;
}

::-webkit-scrollbar-track {
    background: #f1f1f1;
    border-radius: 4px;
}

::-webkit-scrollbar-thumb {
    background: #667eea;
    border-radius: 4px;
}

::-webkit-scrollbar-thumb:hover {
    background: #5a6fd8;
}

=== ./templates/index.html ===
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-
width, initial-scale=1.0">
    <title>TSDB Performance Dashboard</title>
    <script
src="https://unpkg.com/htmx.org@1.9.6"></script>
    <script
src="https://cdn.jsdelivr.net/npm/chart.js"></script>
    <link
href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/6.0.0/css/all.min.css" rel="stylesheet">
    <link rel="stylesheet" href="{{ url_for('static',
filename='style.css') }}">
</head>
```

```
<body>
    <div class="dashboard">
        <!- Header ->
        <header class="dashboard-header">
            <div class="header-content">
                <h1><i class="fas fa-chart-line"></i> TSDB
Performance Dashboard</h1>
                <p class="subtitle">Time Series Database
Performance Comparison</p>
            </div>
        </header>

        <!- Main Content ->
        <main class="dashboard-main">
            <!- Control Panel ->
            <div class="control-panel">
                <div class="control-group">
                    <label>View:</label>
                    <div class="tab-buttons">
                        <button class="tab-btn active" hx-
get="/components/chart/operations" hx-target="#chart-
container" hx-trigger="click">
                            <i class="fas fa-tachometer-alt"></i>
Operations/sec
                        </button>
                        <button class="tab-btn" hx-
get="/components/chart/latency" hx-target="#chart-
container" hx-trigger="click">
                            <i class="fas fa-clock"></i> Latency
                        </button>
                        <button class="tab-btn" hx-
get="/components/chart/resources" hx-target="#chart-
container" hx-trigger="click">
                            <i class="fas fa-server"></i>
Resources
                        </button>
                        <button class="tab-btn" hx-
get="/components/chart/summary" hx-target="#chart-
container" hx-trigger="click">
                            <i class="fas fa-table"></i> Summary
                        </button>
                    </div>
                </div>
                <div class="refresh-control">
                    <button class="refresh-btn" hx-
get="/components/chart/operations" hx-target="#chart-
container" hx-trigger="click">
                        <i class="fas fa-sync-alt"></i> Refresh
                    </button>
                </div>
            </div>

            <!- Chart Container ->
            <div class="chart-section">
                <div id="chart-container" hx-
get="/components/chart/operations" hx-trigger="load">
                    <div class="loading">
                        <i class="fas fa-spinner fa-spin"></i>
Loading performance data...
                    </div>
                </div>
            </div>

            <!- Quick Stats ->
            <div class="stats-grid" id="quick-stats">
```

```
        <div class="stat-card" hx-
get="/api/performance-data" hx-trigger="load" hx-
target="this" hx-swap="innerHTML">
          <div class="loading-small">
            <i class="fas fa-spinner fa-spin"></i>
          </div>
        </div>
      </div>
    </main>

    <!- Footer ->
    <footer class="dashboard-footer">
      <p>&copy; 2025 TSDB Performance
Dashboard | Built with Flask & HTMX</p>
    </footer>
  </div>

  <script>
    // Add active class management for tabs
    document.addEventListener('htmx:afterRequest',
function(evt) {
      if (evt.detail.elt.classList.contains('tab-btn')) {
        document.querySelectorAll('.tab-
btn').forEach(btn => btn.classList.remove('active'));
        evt.detail.elt.classList.add('active');
      }
    });

    // Update quick stats
    document.addEventListener('htmx:afterSettle',
function(evt) {
      if (evt.detail.target.id === 'quick-stats') {
        updateQuickStats();
      }
    });

    function updateQuickStats() {
      fetch('/api/performance-data')
        .then(response => response.json())
        .then(data => {
          const statsContainer =
document.getElementById('quick-stats');
          const bestWrite = data.reduce((max, db) =>
db.write_ops_per_second > max.write_ops_per_second ? db
: max);
          const bestRead = data.reduce((max, db) =>
db.read_ops_per_second > max.read_ops_per_second ? db :
max);
          const lowestLatency = data.reduce((min,
db) => db.avg_write_latency_ms <
min.avg_write_latency_ms ? db : min);
          const lowestMemory = data.reduce((min,
db) => db.memory_mb < min.memory_mb ? db : min);

          statsContainer.innerHTML = `
            <div class="stat-card">
              <div class="stat-icon"><i class="fas
fa-rocket"></i></div>
              <div class="stat-content">
                <h3>Best Write Performance</h3>
                <p class="stat-
value">${bestWrite.database.toUpperCase()}</p>
                <p class="stat-
detail">${Math.round(bestWrite.write_ops_per_second).toL
ocaleString()} ops/sec</p>
              </div>
            </div>
            <div class="stat-card">
              <div class="stat-icon"><i class="fas
fa-search"></i></div>
              <div class="stat-content">
                <h3>Best Read Performance</h3>
                <p class="stat-
value">${bestRead.database.toUpperCase()}</p>
                <p class="stat-
detail">${Math.round(bestRead.read_ops_per_second).toLo
caleString()} ops/sec</p>
              </div>
            </div>
            <div class="stat-card">
              <div class="stat-icon"><i class="fas
fa-bolt"></i></div>
              <div class="stat-content">
                <h3>Lowest Write Latency</h3>
                <p class="stat-
value">${lowestLatency.database.toUpperCase()}</p>
                <p class="stat-
detail">${lowestLatency.avg_write_latency_ms.toFixed(2)}
ms</p>
              </div>
            </div>
            <div class="stat-card">
              <div class="stat-icon"><i class="fas
fa-memory"></i></div>
              <div class="stat-content">
                <h3>Lowest Memory Usage</h3>
                <p class="stat-
value">${lowestMemory.database.toUpperCase()}</p>
                <p class="stat-
detail">${lowestMemory.memory_mb.toFixed(1)} MB</p>
              </div>
            </div>
          `;
        });
    }
  </script>
</body>
</html>

=== ./templates/components/latency_chart.html ===
<div class="chart-wrapper">
  <div class="chart-header">
    <h2><i class="fas fa-clock"></i> Average
Latency</h2>
    <p>Write and read latency comparison (lower is
better)</p>
  </div>
  <div class="chart-content">
    <canvas id="latencyChart" width="400"
height="200"></canvas>
  </div>
</div>

<script>
function initLatencyChart() {
  try {
    const canvas =
document.getElementById('latencyChart');
    if (!canvas) {
      console.error('Canvas element not found');
      return;
    }

    const ctx = canvas.getContext('2d');
```

```
const data = {{ data | tojson }};

if (!data || data.length === 0) {
  console.error('No data available for latency chart');
  return;
}

// Destroy existing chart if it exists
if (window.latencyChartInstance) {
  window.latencyChartInstance.destroy();
}

const databases = data.map(item => item.database.toUpperCase());
const writeLatency = data.map(item => parseFloat(item.avg_write_latency_ms.toFixed(2)));
const readLatency = data.map(item => parseFloat(item.avg_read_latency_ms.toFixed(2)));

const colors = {
  write: 'rgba(255, 206, 86, 0.8)',
  read: 'rgba(75, 192, 192, 0.8)',
  writeBorder: 'rgba(255, 206, 86, 1)',
  readBorder: 'rgba(75, 192, 192, 1)'
};

window.latencyChartInstance = new Chart(ctx, {
  type: 'bar',
  data: {
    labels: databases,
    datasets: [{
      label: 'Write Latency (ms)',
      data: writeLatency,
      backgroundColor: colors.write,
      borderColor: colors.writeBorder,
      borderWidth: 2
    }, {
      label: 'Read Latency (ms)',
      data: readLatency,
      backgroundColor: colors.read,
      borderColor: colors.readBorder,
      borderWidth: 2
    }]
  },
  options: {
    responsive: true,
    maintainAspectRatio: false,
    plugins: {
      legend: {
        position: 'top'
      }
    },
    scales: {
      y: {
        beginAtZero: true,
        title: {
          display: true,
          text: 'Latency (milliseconds)'
        }
      },
      x: {
        title: {
          display: true,
          text: 'Database'
        }
      }
    }
  }
});

console.log('Latency chart initialized successfully');
} catch (error) {
  console.error('Error initializing latency chart:', error);
}
}

// Initialize chart when DOM is loaded
document.addEventListener('DOMContentLoaded', initLatencyChart);

// Initialize chart when content is loaded via HTMX
document.addEventListener('htmx:afterSettle', function(evt) {
  if (evt.detail.target.querySelector('#latencyChart')) {
    setTimeout(initLatencyChart, 100); // Small delay to ensure DOM is ready
  }
});

// Also trigger immediately in case the script is loaded after HTMX swap
if (document.readyState === 'loading') {
  document.addEventListener('DOMContentLoaded', initLatencyChart);
} else {
  initLatencyChart();
}
</script>

=== ./templates/components/operations_chart.html ===
<div class="chart-wrapper">
  <div class="chart-header">
    <h2><i class="fas fa-tachometer-alt"></i> Operations Per Second</h2>
    <p>Write and read operations performance comparison</p>
  </div>
  <div class="chart-content">
    <canvas id="operationsChart" width="400" height="200"></canvas>
  </div>
</div>

<script>
function initOperationsChart() {
  try {
    const canvas = document.getElementById('operationsChart');
    if (!canvas) {
      console.error('Canvas element not found');
      return;
    }

    const ctx = canvas.getContext('2d');
    const data = {{ data | tojson }};

    if (!data || data.length === 0) {
      console.error('No data available for operations chart');
      return;
```

```
            }

            // Destroy existing chart if it exists
            if (window.operationsChartInstance) {
               window.operationsChartInstance.destroy();
            }

            const databases = data.map(item =>
item.database.toUpperCase());
            const writeOps = data.map(item =>
Math.round(item.write_ops_per_second));
            const readOps = data.map(item =>
Math.round(item.read_ops_per_second));

            const colors = {
               write: 'rgba(54, 162, 235, 0.8)',
               read: 'rgba(255, 99, 132, 0.8)',
               writeBorder: 'rgba(54, 162, 235, 1)',
               readBorder: 'rgba(255, 99, 132, 1)'
            };

            window.operationsChartInstance = new Chart(ctx,
{
               type: 'bar',
               data: {
                  labels: databases,
                  datasets: [{
                     label: 'Write Ops/sec',
                     data: writeOps,
                     backgroundColor: colors.write,
                     borderColor: colors.writeBorder,
                     borderWidth: 2
                  }, {
                     label: 'Read Ops/sec',
                     data: readOps,
                     backgroundColor: colors.read,
                     borderColor: colors.readBorder,
                     borderWidth: 2
                  }]
               },
               options: {
                  responsive: true,
                  maintainAspectRatio: false,
                  plugins: {
                     title: {
                        display: false
                     },
                     legend: {
                        position: 'top'
                     }
                  },
                  scales: {
                     y: {
                        beginAtZero: true,
                        title: {
                           display: true,
                           text: 'Operations per Second'
                        }
                     },
                     x: {
                        title: {
                           display: true,
                           text: 'Database'
                        }
                     }
                  }
               }
            });

            console.log('Operations chart initialized
successfully');
         } catch (error) {
            console.error('Error initializing operations chart:',
error);
         }
      }

      // Initialize chart when DOM is loaded
      document.addEventListener('DOMContentLoaded',
initOperationsChart);

      // Initialize chart when content is loaded via HTMX
      document.addEventListener('htmx:afterSettle',
function(evt) {
         if
(evt.detail.target.querySelector('#operationsChart')) {
            setTimeout(initOperationsChart, 100); // Small
delay to ensure DOM is ready
         }
      });

      // Also trigger immediately in case the script is loaded
after HTMX swap
      if (document.readyState === 'loading') {
         document.addEventListener('DOMContentLoaded',
initOperationsChart);
      } else {
         initOperationsChart();
      }
      </script>

      === ./templates/components/resources_chart.html ===
      <div class="chart-wrapper">
         <div class="chart-header">
            <h2><i class="fas fa-server"></i> Resource
Usage</h2>
            <p>Memory consumption comparison</p>
         </div>
         <div class="chart-content">
            <canvas id="resourcesChart" width="400"
height="200"></canvas>
         </div>
      </div>

      <script>
      function initResourcesChart() {
         try {
            const canvas =
document.getElementById('resourcesChart');
            if (!canvas) {
               console.error('Canvas element not found');
               return;
            }

            const ctx = canvas.getContext('2d');
            const data = {{ data | tojson }};

            if (!data || data.length === 0) {
               console.error('No data available for resources
chart');
               return;
            }

            // Destroy existing chart if it exists
```

```
            if (window.resourcesChartInstance) {
                window.resourcesChartInstance.destroy();
            }

            const databases = data.map(item =>
item.database.toUpperCase());
            const memoryUsage = data.map(item =>
parseFloat(item.memory_mb.toFixed(1)));

            const colors = [
                'rgba(255, 99, 132, 0.8)',
                'rgba(54, 162, 235, 0.8)',
                'rgba(255, 205, 86, 0.8)',
                'rgba(75, 192, 192, 0.8)'
            ];

            const borderColors = [
                'rgba(255, 99, 132, 1)',
                'rgba(54, 162, 235, 1)',
                'rgba(255, 205, 86, 1)',
                'rgba(75, 192, 192, 1)'
            ];

            window.resourcesChartInstance = new Chart(ctx,
{
                type: 'doughnut',
                data: {
                    labels: databases,
                    datasets: [{
                        label: 'Memory Usage (MB)',
                        data: memoryUsage,
                        backgroundColor: colors,
                        borderColor: borderColors,
                        borderWidth: 2
                    }]
                },
                options: {
                    responsive: true,
                    maintainAspectRatio: false,
                    plugins: {
                        legend: {
                            position: 'right'
                        },
                        tooltip: {
                            callbacks: {
                                label: function(context) {
                                    return context.label + ': ' +
context.parsed + ' MB';
                                }
                            }
                        }
                    }
                }
            });

            console.log('Resources chart initialized
successfully');
        } catch (error) {
            console.error('Error initializing resources chart:',
error);
        }
    }

    // Initialize chart when DOM is loaded
    document.addEventListener('DOMContentLoaded',
initResourcesChart);

    // Initialize chart when content is loaded via HTMX
    document.addEventListener('htmx:afterSettle',
function(evt) {
        if (evt.detail.target.querySelector('#resourcesChart'))
{
            setTimeout(initResourcesChart, 100); // Small
delay to ensure DOM is ready
        }
    });

    // Also trigger immediately in case the script is loaded
after HTMX swap
    if (document.readyState === 'loading') {
        document.addEventListener('DOMContentLoaded',
initResourcesChart);
    } else {
        initResourcesChart();
    }
    </script>

    === ./templates/components/summary_table.html ===
    <div class="chart-wrapper">
        <div class="chart-header">
            <h2><i class="fas fa-table"></i> Performance
Summary</h2>
            <p>Complete performance metrics
comparison</p>
        </div>
        <div class="table-content">
            <div class="table-wrapper">
                <table class="performance-table">
                    <thead>
                        <tr>
                            <th>Database</th>
                            <th>Write Ops/sec</th>
                            <th>Read Ops/sec</th>
                            <th>Write Latency (ms)</th>
                            <th>Read Latency (ms)</th>
                            <th>Memory (MB)</th>
                            <th>Test Duration (s)</th>
                            <th>Error Rate</th>
                        </tr>
                    </thead>
                    <tbody>
                        {% for db in data %}
                        <tr>
                            <td class="db-name">
                                <span class="db-badge db-{{
db.database }}">{{ db.database.upper() }}</span>
                            </td>
                            <td class="metric-value">{{
"{:,.0f}".format(db.write_ops_per_second) }}</td>
                            <td class="metric-value">{{
"{:,.0f}".format(db.read_ops_per_second) }}</td>
                            <td class="metric-value">{{
"%.2f"|format(db.avg_write_latency_ms) }}</td>
                            <td class="metric-value">{{
"%.2f"|format(db.avg_read_latency_ms) }}</td>
                            <td class="metric-value">{{
"%.1f"|format(db.memory_mb) }}</td>
                            <td class="metric-value">{{
"%.2f"|format(db.test_duration_seconds) }}</td>
                            <td class="metric-value">
                                {% if db.error_rate == 0 %}
                                    <span class="success-
badge">0%</span>
                                {% else %}
```

```
<span class="error-badge">{{ "%.1f"|format(db.error_rate * 100) }}%</span>
{% endif %}
</td>
</tr>
{% endfor %}
</tbody>
</table>
</div>

<div class="performance-insights">
<h3>Key Insights</h3>
<div class="insights-grid">
{% set best_write = data | max(attribute='write_ops_per_second') %}
{% set best_read = data | max(attribute='read_ops_per_second') %}
{% set lowest_latency = data | min(attribute='avg_write_latency_ms') %}
{% set lowest_memory = data | min(attribute='memory_mb') %}

<div class="insight-card">
<i class="fas fa-trophy"></i>
<h4>Best Write Performance</h4>
<p><strong>{{ best_write.database.upper() }}</strong> with {{ "{:,.0f}".format(best_write.write_ops_per_second) }} ops/sec</p>
</div>

<div class="insight-card">
<i class="fas fa-search"></i>
<h4>Best Read Performance</h4>
<p><strong>{{ best_read.database.upper() }}</strong> with {{ "{:,.0f}".format(best_read.read_ops_per_second) }} ops/sec</p>
</div>

<div class="insight-card">
<i class="fas fa-bolt"></i>
<h4>Lowest Latency</h4>
<p><strong>{{ lowest_latency.database.upper() }}</strong> with {{ "%.2f"|format(lowest_latency.avg_write_latency_ms) }} ms</p>
</div>

<div class="insight-card">
<i class="fas fa-memory"></i>
<h4>Most Efficient</h4>
<p><strong>{{ lowest_memory.database.upper() }}</strong> using {{ "%.1f"|format(lowest_memory.memory_mb) }} MB</p>
</div>
</div>
</div>
</div>
</div>

=== ./visualizer.py ===
from flask import Flask, render_template, jsonify
import json
import os

app = Flask(__name__)

def load_performance_data():
    """Load performance data from JSON file"""
    try:
        with open('tsdb_performance_results.json', 'r') as f:
            return json.load(f)
    except FileNotFoundError:
        return []

@app.route('/')
def index():
    """Main dashboard page"""
    return render_template('index.html')

@app.route('/api/performance-data')
def get_performance_data():
    """API endpoint to get performance data"""
    data = load_performance_data()
    return jsonify(data)

@app.route('/api/metrics/<metric>')
def get_metric_data(metric):
    """API endpoint to get specific metric data"""
    data = load_performance_data()

    valid_metrics = [
        'write_ops_per_second', 'read_ops_per_second', 'memory_mb',
        'avg_write_latency_ms', 'avg_read_latency_ms', 'error_rate'
    ]

    if metric not in valid_metrics:
        return jsonify({'error': 'Invalid metric'}), 400

    result = []
    for db in data:
        result.append({
            'database': db['database'],
            'value': db[metric],
            'metric': metric
        })

    return jsonify(result)

@app.route('/components/chart/<chart_type>')
def get_chart_component(chart_type):
    """HTMX endpoint for chart components"""
    data = load_performance_data()

    if chart_type == 'operations':
        return render_template('components/operations_chart.html', data=data)
    elif chart_type == 'latency':
        return render_template('components/latency_chart.html', data=data)
    elif chart_type == 'resources':
        return render_template('components/resources_chart.html', data=data)
    elif chart_type == 'summary':
        return render_template('components/summary_table.html', data=data)
```

```
        return "Chart not found", 404

    if __name__ == '__main__':
        app.run(debug=True, host='0.0.0.0', port=5001)

    === ./src/models.py ===
    from dataclasses import dataclass


    @dataclass
    class TestResult:
        database: str
        write_ops_per_second: float
        read_ops_per_second: float
        memory_mb: float
        disk_mb: float
        avg_write_latency_ms: float
        avg_read_latency_ms: float
        error_rate: float
        test_duration_seconds: float

    === ./src/testers/influxdb_tester.py ===
    import time
    import statistics
    import traceback
    from src.testers.base_tester import DatabaseTester
    from src.docker_manager import DockerManager


    class InfluxDBTester(DatabaseTester):
        def __init__(self, docker_manager:
DockerManager):
            super().__init__(docker_manager)
            self.db_name = 'influxdb'
            self.config = {
                'image': 'influxdb:2.7',
                'ports': ['8086:8086'],
                'environment': {
                    'DOCKER_INFLUXDB_INIT_MODE':
'setup',

'DOCKER_INFLUXDB_INIT_USERNAME': 'admin',

'DOCKER_INFLUXDB_INIT_PASSWORD':
'password123',
                    'DOCKER_INFLUXDB_INIT_ORG':
'testorg',
                    'DOCKER_INFLUXDB_INIT_BUCKET':
'testbucket',

'DOCKER_INFLUXDB_INIT_ADMIN_TOKEN':
'testtoken123'
                },
                'healthcheck_url': 'http://localhost:8086/health'
            }

        def test_write_performance(self) -> tuple:
            """Test write performance"""
            try:
                from influxdb_client import InfluxDBClient,
Point
                from influxdb_client.client.write_api import
SYNCHRONOUS

                client = InfluxDBClient(
                    url="http://localhost:8086",
                    token="testtoken123",
                    org="testorg"
                )
                write_api =
client.write_api(write_options=SYNCHRONOUS)

                start_time = time.time()
                errors = 0
                latencies = []

                for data_point in self.test_data[:1000]:  # Test
with subset
                    point_start = time.time()
                    try:
                        point = Point("measurement") \
                            .tag("series_id", data_point['series_id'])
\
                            .tag("location", data_point['location']) \
                            .tag("host", data_point['host']) \
                            .field("value", data_point['value']) \
                            .time(data_point['timestamp'])

                        write_api.write(bucket="testbucket",
record=point)
                        latencies.append((time.time() - point_start)
* 1000)
                    except Exception as e:
                        traceback.print_exc()
                        errors += 1

                duration = time.time() - start_time
                ops_per_second = len(self.test_data[:1000]) /
duration
                avg_latency = statistics.mean(latencies) if
latencies else 0
                error_rate = errors / len(self.test_data[:1000])

                client.close()
                return ops_per_second, avg_latency, error_rate,
duration

            except ImportError:
                traceback.print_exc()
                print("influxdb-client package required for
InfluxDB testing")
                return 0, 0, 1, 0
            except Exception as e:
                traceback.print_exc()
                print(f"InfluxDB write test error: {str(e)}")
                return 0, 0, 1, 0

        def test_read_performance(self) -> tuple:
            """Test read performance"""
            try:
                from influxdb_client import InfluxDBClient

                client = InfluxDBClient(
                    url="http://localhost:8086",
                    token="testtoken123",
                    org="testorg"
                )
                query_api = client.query_api()

                queries = [
                    'from(bucket: "testbucket") |> range(start: -
1h) |> filter(fn: (r) => r._measurement == "measurement") |>
limit(n: 100)',
```

```
            'from(bucket: "testbucket") |> range(start: -1h) |> filter(fn: (r) => r.series_id == "sensor_001")',
            'from(bucket: "testbucket") |> range(start: -1h) |> filter(fn: (r) => r.location == "datacenter1") |> mean()'
        ]

        start_time = time.time()
        errors = 0
        latencies = []

        for _ in range(50):  # Run 50 queries
            for query in queries:
                query_start = time.time()
                try:
                    result = query_api.query(query)
                    latencies.append((time.time() - query_start) * 1000)
                except Exception as e:
                    traceback.print_exc()
                    errors += 1

        duration = time.time() - start_time
        total_queries = 50 * len(queries)
        ops_per_second = total_queries / duration
        avg_latency = statistics.mean(latencies) if latencies else 0
        error_rate = errors / total_queries

        client.close()
        return ops_per_second, avg_latency, error_rate, duration

    except Exception as e:
        traceback.print_exc()
        print(f"InfluxDB read test error: {str(e)}")
        return 0, 0, 1, 0

=== ./src/testers/base_tester.py ===
import random
from datetime import datetime, timedelta
from typing import List, Dict
from src.docker_manager import DockerManager


class DatabaseTester:
    def __init__(self, docker_manager: DockerManager):
        self.docker_manager = docker_manager
        self.test_data = self._generate_test_data()

    def _generate_test_data(self, num_series: int = 100, points_per_series: int = 1000) -> List[Dict]:
        """Generate test time series data"""
        data = []
        base_time = datetime.now() - timedelta(hours=1)

        for series_id in range(num_series):
            for i in range(points_per_series):
                timestamp = base_time + timedelta(seconds=i * 10)
                data.append({
                    'timestamp': timestamp,
                    'series_id': f'sensor_{series_id:03d}',
                    'value': random.uniform(0, 100),
                    'location': random.choice(['datacenter1', 'datacenter2', 'datacenter3']),
                    'host': f'host_{series_id % 10:02d}'
                })

        return data

    def test_write_performance(self) -> tuple:
        """Test write performance - to be implemented by subclasses"""
        raise NotImplementedError("Subclasses must implement test_write_performance")

    def test_read_performance(self) -> tuple:
        """Test read performance - to be implemented by subclasses"""
        raise NotImplementedError("Subclasses must implement test_read_performance")

=== ./src/testers/questdb_tester.py ===
import time
import statistics
import requests
import traceback
from src.testers.base_tester import DatabaseTester
from src.docker_manager import DockerManager


class QuestDBTester(DatabaseTester):
    def __init__(self, docker_manager: DockerManager):
        super().__init__(docker_manager)
        self.db_name = 'questdb'
        self.config = {
            'image': 'questdb/questdb:9.0.3',
            'ports': ['9000:9000', '9009:9009'],
            'healthcheck_url': 'http://localhost:9000'
        }

    def test_write_performance(self) -> tuple:
        """Test write performance using QuestDB Python client"""
        try:
            from questdb.ingress import Sender, IngressError, Protocol

            start_time = time.time()
            errors = 0
            latencies = []

            # Use QuestDB sender for batch writes
            with Sender(Protocol.Http, 'localhost', 9000) as sender:
                for data_point in self.test_data[:1000]:
                    point_start = time.time()
                    try:
                        sender.row(
                            'metrics',
                            symbols={
                                'series_id': data_point['series_id'],
                                'location': data_point['location'],
                                'host': data_point['host']
                            },
                            columns={
                                'value': data_point['value']
                            },
                            at=data_point['timestamp']
                        )
```

```
                        # Flush every 100 rows for better
performance measurement
                        if len(latencies) % 100 == 99:
                            sender.flush()

                        latencies.append((time.time() -
point_start) * 1000)

                    except IngressError as e:
                        traceback.print_exc()
                        errors += 1
                    except Exception as e:
                        traceback.print_exc()
                        errors += 1

                # Final flush
                sender.flush()

            duration = time.time() - start_time
            ops_per_second = len(self.test_data[:1000]) /
duration if duration > 0 else 0
            avg_latency = statistics.mean(latencies) if
latencies else 0
            error_rate = errors / len(self.test_data[:1000])

            return ops_per_second, avg_latency, error_rate,
duration

        except ImportError:
            print("questdb package required for QuestDB
testing")
            print("Install with: pip install questdb")
            return 0, 0, 1, 0
        except Exception as e:
            print(f"QuestDB write test error: {str(e)}")
            return 0, 0, 1, 0

    def test_read_performance(self) -> tuple:
        """Test read performance using SQL API"""
        try:
            queries = [
                "SELECT * FROM metrics LIMIT 100",
                "SELECT * FROM metrics WHERE
series_id = 'sensor_001'",
                "SELECT location, avg(value) FROM
metrics GROUP BY location"
            ]

            start_time = time.time()
            errors = 0
            latencies = []

            for _ in range(50):
                for query in queries:
                    query_start = time.time()
                    try:
                        response = requests.get(
                            'http://localhost:9000/exec',
                            params={'query': query},
                            timeout=10
                        )
                        if response.status_code == 200:
                            latencies.append((time.time() -
query_start) * 1000)
                        else:
                            print(response)
                            errors += 1
                    except Exception as e:
                        traceback.print_exc()
                        errors += 1

            duration = time.time() - start_time
            total_queries = 50 * len(queries)
            ops_per_second = total_queries / duration if
duration > 0 else 0
            avg_latency = statistics.mean(latencies) if
latencies else 0
            error_rate = errors / total_queries if
total_queries > 0 else 1

            return ops_per_second, avg_latency, error_rate,
duration

        except Exception as e:
            print(f"QuestDB read test error: {str(e)}")
            return 0, 0, 1, 0

=== ./src/testers/victoriametrics_tester.py ===
import time
import statistics
import requests
import traceback
from src.testers.base_tester import DatabaseTester
from src.docker_manager import DockerManager


class VictoriaMetricsTester(DatabaseTester):
    def __init__(self, docker_manager:
DockerManager):
        super().__init__(docker_manager)
        self.db_name = 'victoriametrics'
        self.config = {
            'image': 'victoriametrics/victoria-metrics:latest',
            'ports': ['8428:8428'],
            'command': ['-storageDataPath=/victoria-
metrics-data', '-retentionPeriod=1'],
            'healthcheck_url': 'http://localhost:8428/health'
        }

    def test_write_performance(self) -> tuple:
        """Test write performance using Prometheus
remote write format"""
        try:
            start_time = time.time()
            errors = 0
            latencies = []

            # Convert test data to Prometheus format and
write in batches
            batch_size = 100
            for i in range(0, min(1000, len(self.test_data)),
batch_size):
                batch = self.test_data[i:i + batch_size]
                point_start = time.time()

                try:
                    # Prometheus line format for
VictoriaMetrics
                    lines = []
                    for data_point in batch:
                        timestamp_ms =
int(data_point['timestamp'].timestamp() * 1000)
                        line =
f'metrics{{series_id="{data_point["series_id"]}",location="
```

```
{data_point["location"]}",host="{data_point["host"]}"}}
{data_point["value"]} {timestamp_ms}'
                        lines.append(line)

                    data = '\n'.join(lines)

                    response = requests.post(

'http://localhost:8428/api/v1/import/prometheus',
                        data=data,
                        headers={'Content-Type': 'text/plain'},
                        timeout=10
                    )

                    if response.status_code == 204:
                        latencies.append((time.time() -
point_start) * 1000)
                    else:
                        errors += len(batch)

                except Exception as e:
                    traceback.print_exc()
                    errors += len(batch)

            duration = time.time() - start_time
            total_points = min(1000, len(self.test_data))
            ops_per_second = total_points / duration if
duration > 0 else 0
            avg_latency = statistics.mean(latencies) if
latencies else 0
            error_rate = errors / total_points if total_points
> 0 else 1

            return ops_per_second, avg_latency, error_rate,
duration

        except Exception as e:
            traceback.print_exc()
            print(f"VictoriaMetrics write test error:
{str(e)}")
            return 0, 0, 1, 0

    def test_read_performance(self) -> tuple:
        """Test read performance using PromQL"""
        try:
            queries = [
                'metrics',
                'metrics{series_id="sensor_001"}',
                'avg_over_time(metrics[1h])',
                'sum by (location) (metrics)',
                'rate(metrics[5m])'
            ]

            start_time = time.time()
            errors = 0
            latencies = []

            for _ in range(30):  # Run fewer queries as they
might be more expensive
                for query in queries:
                    query_start = time.time()
                    try:
                        response = requests.get(
                            'http://localhost:8428/api/v1/query',
                            params={'query': query},
                            timeout=10
                        )
                        if response.status_code == 200:
                            latencies.append((time.time() -
query_start) * 1000)
                        else:
                            errors += 1
                    except Exception as e:
                        traceback.print_exc()
                        errors += 1

            duration = time.time() - start_time
            total_queries = 30 * len(queries)
            ops_per_second = total_queries / duration if
duration > 0 else 0
            avg_latency = statistics.mean(latencies) if
latencies else 0
            error_rate = errors / total_queries if
total_queries > 0 else 1

            return ops_per_second, avg_latency, error_rate,
duration

        except Exception as e:
            traceback.print_exc()
            print(f"VictoriaMetrics read test error: {str(e)}")
            return 0, 0, 1, 0

=== ./src/testers/timescaledb_tester.py ===
import time
import statistics
import traceback
from src.testers.base_tester import DatabaseTester
from src.docker_manager import DockerManager


class TimescaleDBTester(DatabaseTester):
    def __init__(self, docker_manager:
DockerManager):
        super().__init__(docker_manager)
        self.db_name = 'timescaledb'
        self.config = {
            'image': 'timescale/timescaledb:latest-pg15',
            'ports': ['5432:5432'],
            'environment': {
                'POSTGRES_DB': 'testdb',
                'POSTGRES_USER': 'testuser',
                'POSTGRES_PASSWORD': 'testpass'
            },
            'healthcheck': {
                'test': ['CMD-SHELL', 'pg_isready -U testuser
-d testdb'],
                'interval': '5s',
                'timeout': '5s',
                'retries': 5,
                'start_period': '10s'
            }
        }

    def _setup_schema(self):
        """Setup TimescaleDB schema"""
        try:
            import psycopg2

            conn = psycopg2.connect(
                host='localhost',
                database='testdb',
                user='testuser',
                password='testpass'
```

```
            )
            cur = conn.cursor()

            # Create table and hypertable
            cur.execute("""
                CREATE TABLE IF NOT EXISTS
metrics
                (
                    time
                    TIMESTAMPTZ
                    NOT
                    NULL,
                    series_id
                    TEXT
                    NOT
                    NULL,
                    value
                    DOUBLE
                    PRECISION,
                    location
                    TEXT,
                    host
                    TEXT
                );
                """)

            cur.execute("SELECT
create_hypertable('metrics', 'time', if_not_exists => TRUE);")
            conn.commit()
            cur.close()
            conn.close()

        except ImportError:
            traceback.print_exc()
            print("psycopg2-binary package required for
TimescaleDB testing")
        except Exception as e:
            traceback.print_exc()
            print(f"TimescaleDB setup error: {str(e)}")

    def test_write_performance(self) -> tuple:
        """Test write performance"""
        try:
            import psycopg2

            self._setup_schema()

            conn = psycopg2.connect(
                host='localhost',
                database='testdb',
                user='testuser',
                password='testpass'
            )
            cur = conn.cursor()

            start_time = time.time()
            errors = 0
            latencies = []

            for data_point in self.test_data[:1000]:
                point_start = time.time()
                try:
                    cur.execute(
                        "INSERT INTO metrics (time, series_id,
value, location, host) VALUES (%s, %s, %s, %s, %s)",
                        (data_point['timestamp'],
data_point['series_id'], data_point['value'],
                         data_point['location'],
data_point['host'])
                    )
                    conn.commit()
                    latencies.append((time.time() - point_start)
* 1000)
                except Exception as e:
                    traceback.print_exc()
                    errors += 1
                    conn.rollback()

            duration = time.time() - start_time
            ops_per_second = len(self.test_data[:1000]) /
duration
            avg_latency = statistics.mean(latencies) if
latencies else 0
            error_rate = errors / len(self.test_data[:1000])

            cur.close()
            conn.close()
            return ops_per_second, avg_latency, error_rate,
duration

        except ImportError:
            traceback.print_exc()
            print("psycopg2-binary package required for
TimescaleDB testing")
            return 0, 0, 1, 0
        except Exception as e:
            traceback.print_exc()
            print(f"TimescaleDB write test error: {str(e)}")
            return 0, 0, 1, 0

    def test_read_performance(self) -> tuple:
        """Test read performance"""
        try:
            import psycopg2

            conn = psycopg2.connect(
                host='localhost',
                database='testdb',
                user='testuser',
                password='testpass'
            )
            cur = conn.cursor()

            queries = [
                "SELECT * FROM metrics WHERE time >=
NOW() - INTERVAL '1 hour' LIMIT 100",
                "SELECT * FROM metrics WHERE
series_id = 'sensor_001'",
                "SELECT location, AVG(value) FROM
metrics WHERE time >= NOW() - INTERVAL '1 hour'
GROUP BY location"
            ]

            start_time = time.time()
            errors = 0
            latencies = []

            for _ in range(50):
                for query in queries:
                    query_start = time.time()
                    try:
                        cur.execute(query)
                        cur.fetchall()
```

```
                    latencies.append((time.time() -
query_start) * 1000)
                except Exception as e:
                    traceback.print_exc()
                    errors += 1

            duration = time.time() - start_time
            total_queries = 50 * len(queries)
            ops_per_second = total_queries / duration
            avg_latency = statistics.mean(latencies) if
latencies else 0
            error_rate = errors / total_queries

            cur.close()
            conn.close()
            return ops_per_second, avg_latency, error_rate,
duration

        except Exception as e:
            traceback.print_exc()
            print(f"TimescaleDB read test error: {str(e)}")
            return 0, 0, 1, 0

=== ./src/test_suite.py ===
import time
import json
import traceback
from typing import List
from dataclasses import asdict
from src.docker_manager import DockerManager
from src.models import TestResult
from src.testers.influxdb_tester import InfluxDBTester
from src.testers.timescaledb_tester import
TimescaleDBTester
from src.testers.questdb_tester import QuestDBTester
from src.testers.victoriametrics_tester import
VictoriaMetricsTester


class PerformanceTestSuite:
    def __init__(self):
        self.docker_manager = DockerManager()
        self.results = []

        # Available testers
        self.testers = {
            'influxdb': InfluxDBTester,
            'timescaledb': TimescaleDBTester,
            'questdb': QuestDBTester,
            'victoriametrics': VictoriaMetricsTester
        }

    def run_test_suite(self, databases: List[str] = None) -
> List[TestResult]:
        """Run performance tests on specified
databases"""
        if databases is None:
            databases = list(self.testers.keys())

        print("Starting Time Series Database Performance
Test Suite")
        print("=" * 60)

        for db_name in databases:
            if db_name not in self.testers:
                print(f"Unknown database: {db_name}")
                continue

            print(f"\nTesting {db_name.upper()}...")
            result = self._test_database(db_name)
            if result:
                self.results.append(result)
                print(f"✓ {db_name} test completed")
            else:
                print(f"✗ {db_name} test failed")

        return self.results

    def _test_database(self, db_name: str) -> TestResult:
        """Test a single database"""
        try:
            # Initialize tester
            tester_class = self.testers[db_name]
            tester = tester_class(self.docker_manager)

            # Start database
            print(f"  Starting {db_name} container...")
            if not
self.docker_manager.start_database(db_name, tester.config):
                return None

            # Wait for stability
            time.sleep(10)

            # Run write test
            print(f"  Running write performance test...")
            write_ops, write_latency, write_errors,
write_duration = tester.test_write_performance()

            # Run read test
            print(f"  Running read performance test...")
            read_ops, read_latency, read_errors,
read_duration = tester.test_read_performance()

            # Get resource usage
            print(f"  Measuring resource usage...")
            stats =
self.docker_manager.get_container_stats(db_name)

            # Create result
            result = TestResult(
                database=db_name,
                write_ops_per_second=write_ops,
                read_ops_per_second=read_ops,
                memory_mb=stats['memory_mb'],
                disk_mb=stats['disk_mb'],
                avg_write_latency_ms=write_latency,
                avg_read_latency_ms=read_latency,
                error_rate=(write_errors + read_errors) / 2,
                test_duration_seconds=write_duration +
read_duration
            )

            return result

        except Exception as e:
            traceback.print_exc()
            print(f"Error testing {db_name}: {str(e)}")
            return None

        finally:
            # Cleanup
            print(f"  Cleaning up {db_name}...")
            self.docker_manager.stop_database(db_name)
```

```
    def print_results(self):
        """Print formatted test results"""
        if not self.results:
            print("No test results available")
            return

        print("\n" + "=" * 80)
        print("PERFORMANCE TEST RESULTS")
        print("=" * 80)

        # Headers
        print(
            f"{'Database':<15} {'Write/s':<10}
{'Read/s':<10} {'Memory':<10} {'Disk':<10} {'W.Lat':<8}
{'R.Lat':<8} {'Errors':<8}")
        print("-" * 80)

        # Sort by write performance
        sorted_results = sorted(self.results, key=lambda x:
x.write_ops_per_second, reverse=True)

        for result in sorted_results:
            print(f"{result.database:<15} "
                  f"{result.write_ops_per_second:<10.1f} "
                  f"{result.read_ops_per_second:<10.1f} "
                  f"{result.memory_mb:<10.1f} "
                  f"{result.disk_mb:<10.1f} "
                  f"{result.avg_write_latency_ms:<8.1f} "
                  f"{result.avg_read_latency_ms:<8.1f} "
                  f"{result.error_rate:<8.2%}")

        print("\nUnits: Write/s & Read/s = operations per
second, Memory & Disk = MB, Latency = milliseconds")

    def save_results(self, filename: str =
"tsdb_performance_results.json"):
        """Save results to JSON file"""
        results_dict = [asdict(result) for result in
self.results]
        with open(filename, 'w') as f:
            json.dump(results_dict, f, indent=2, default=str)
        print(f"\nResults saved to {filename}")

=== ./src/docker_manager.py ===
import docker
import time
import subprocess
import os
import tempfile
import yaml
import requests
import traceback
from typing import Dict


class DockerManager:
    def __init__(self):
        self.client = docker.from_env()
        self.containers = {}
        self.compose_files = {}

    def create_compose_file(self, db_name: str, config:
dict) -> str:
        """Create a docker-compose file for the
database"""
        # Separate healthcheck_url from docker-compose
config
        docker_config = {k: v for k, v in config.items() if
k != 'healthcheck_url'}

        compose_content = {
            'version': '3.8',
            'services': {
                db_name: docker_config
            }
        }

        # Create temporary compose file
        fd, path = tempfile.mkstemp(suffix='.yml',
prefix=f'{db_name}_')
        with os.fdopen(fd, 'w') as f:
            yaml.dump(compose_content, f)

        self.compose_files[db_name] = path
        return path

    def start_database(self, db_name: str, config: dict) ->
bool:
        """Start a database using docker-compose"""
        try:
            compose_file =
self.create_compose_file(db_name, config)

            # Start the service
            result = subprocess.run([
                'docker-compose', '-f', compose_file, 'up', '-d'
            ], capture_output=True, text=True,
timeout=120)

            if result.returncode != 0:
                print(f"Error starting {db_name}:
{result.stderr}")
                return False

            # Wait for service to be ready
            self._wait_for_service(db_name,
config.get('healthcheck_url'))
            return True

        except Exception as e:
            traceback.print_exc()
            print(f"Error starting {db_name}: {str(e)}")
            return False

    def stop_database(self, db_name: str):
        """Stop and remove database containers"""
        try:
            if db_name in self.compose_files:
                compose_file = self.compose_files[db_name]
                subprocess.run([
                    'docker-compose', '-f', compose_file,
'down', '-v'
                ], capture_output=True, timeout=60)
                os.unlink(compose_file)
                del self.compose_files[db_name]
        except Exception as e:
            traceback.print_exc()
            print(f"Error stopping {db_name}: {str(e)}")

    def _wait_for_service(self, db_name: str,
healthcheck_url: str = None, timeout: int = 90):
        """Wait for service to be ready"""
```

```
        print(f"    Waiting for {db_name} service to be ready...")
        start_time = time.time()

        while time.time() - start_time < timeout:
            try:
                if healthcheck_url:
                    # Use HTTP health check
                    response = requests.get(healthcheck_url, timeout=5)
                    if response.status_code == 200:
                        print(f"    {db_name} is ready (HTTP check passed)")
                        break
                else:
                    # Check if container is running and healthy
                    containers = self.client.containers.list(
                        filters={'label': f'com.docker.compose.service={db_name}'}
                    )
                    if containers:
                        container = containers[0]
                        if container.status == 'running':
                            # If container has healthcheck, wait for it to be healthy
                            container.reload()
                            health_status = container.attrs.get('State', {}).get('Health', {}).get('Status')
                            if health_status == 'healthy' or health_status is None:
                                print(f"    {db_name} is ready (container healthy)")
                                break

            except Exception as e:
                # traceback.print_exc()
                pass
            time.sleep(3)

        # Additional wait for service initialization
        print(f"    Giving {db_name} extra time to initialize...")
        time.sleep(10)

    def get_container_stats(self, db_name: str) -> Dict[str, float]:
        """Get memory and disk usage for container"""
        try:
            # Find container by name pattern (docker-compose creates containers with predictable names)
            all_containers = self.client.containers.list(all=True)
            target_container = None

            for container in all_containers:
                # Docker compose typically names containers like: folder_service_1
                if db_name in container.name.lower() or any(db_name in tag for tag in container.image.tags):
                    target_container = container
                    break

            if not target_container:
                print(f"  No container found for {db_name}")
                return {'memory_mb': 0, 'disk_mb': 0}

            if target_container.status != 'running':
                print(f"    Container {db_name} is not running")
                return {'memory_mb': 0, 'disk_mb': 0}

            # Get container stats
            stats = target_container.stats(stream=False)

            # Memory usage in MB
            memory_usage = 0
            if 'memory_stats' in stats and 'usage' in stats['memory_stats']:
                memory_usage = stats['memory_stats']['usage'] / (1024 * 1024)

            # Disk usage estimation
            disk_usage = 0
            try:
                # Get container size information
                container_info = self.client.api.inspect_container(target_container.id)
                if 'SizeRootFs' in container_info:
                    disk_usage = container_info['SizeRootFs'] / (1024 * 1024)
                elif 'GraphDriver' in container_info and 'Data' in container_info['GraphDriver']:
                    # Alternative method for disk usage
                    disk_usage = 50  # Rough estimate in MB if we can't get exact size
            except Exception as disk_error:
                print(f"    Could not get disk usage for {db_name}: {str(disk_error)}")
                disk_usage = 0

            print(f"    {db_name} stats - Memory: {memory_usage:.1f}MB, Disk: {disk_usage:.1f}MB")
            return {'memory_mb': memory_usage, 'disk_mb': disk_usage}

        except Exception as e:
            traceback.print_exc()
            print(f"Error getting stats for {db_name}: {str(e)}")
            # Return reasonable default values instead of zeros
            return {'memory_mb': 0, 'disk_mb': 0}
```