ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ АВТОМАТИЗОВАНИХ СИСТЕМ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка системи тестування максимального навантаження баз даних часових рядів»

на здобуття освітнього ступеня магістра
зі спеціальності 126 Інформаційні системи та технології
(код, найменування спеціальності)
освітньо-професійної програми Інформаційні системи та технології
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

________________ Володимир Старцев
(підпис) *Ім'я, Прізвище здобувача*

Виконав:
здобувач вищої освіти
група ІСДМ-62
Володимир Старцев

Керівник:
науковий ступінь, вчене звання
Юлія Глущенко
к.фм.н., доцент

Рецензент:
науковий ступінь, вчене звання
____
Ім'я, Прізвище

Київ 2023

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення автоматизованих систем

Ступінь вищої освіти Магістр

Спеціальність Інформаційні системи та технології

Освітньо-професійна програма Інформаційні системи та технології

ЗАТВЕРДЖУЮ
Завідувач кафедрою ІПЗАС

_______________ Каміла СТОРЧАК
«_____» ___________ 2023 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Старцеву Володимиру Михайловичу
(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: Розробка системи тестування максимального навантаження баз даних часових рядів

керівник кваліфікаційної роботи Юлія Глущенко к.фм.н., доцент,
(Ім'я, Прізвище науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «19» 10.2023р. №145

2. Строк подання кваліфікаційної роботи «29» грудня 2023р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, методи тестування, бази даних часових рядів.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
Дослідження методів тестування
Аналіз баз даних часових рядів
Розробка системи тестування навантаження баз даних часових рядів

5. Перелік графічного матеріалу: *презентація*
6. Дата видачі завдання «19» жовтня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	19.10-05.11.23	виконано
2	Вивчення матеріалів для аналізу методів тестування	05.11-12.11.23	виконано
3	Дослідження баз даних	13.11-19.11.23	виконано
4	Аналіз особливостей баз даних часових рядів	20.11-25.11.23	виконано
5	Розробка системи тестування баз даних часових рядів	27.11-03.12.23	виконано
6	Проведення тестування	04.12-10.12.23	виконано
7	Оформлення роботи: вступ, висновки, реферат	11.12-20.12.23	виконано
8	Розробка демонстраційних матеріалів	21.12-29.12.23	виконано

Здобувач вищої освіти _______________ Володимир Старцев
(підпис) *(Ім'я, Прізвище)*

Керівник
кваліфікаційної роботи _______________ Юлія Глущенко
(підпис) *(Ім'я, Прізвище)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 80 стор., 1 дод., 28 рис., 37 джерел.

Мета роботи – побудова системи тестування максимального навантаження баз даних часових рядів.

Об'єкт дослідження – процес тестування бази данних часових рядів. *Предмет дослідження* – визначення можливостей баз даних часових рядів

Короткий зміст роботи: У роботі проведено розробку системи тестування максимального навантаження баз даних часових рядів. Проаналізовано основні методи тестування. Проаналізовано основні типи баз даних. Проаналізовано роботу MQTT брокера.

КЛЮЧОВІ СЛОВА: ТЕСТУВАННЯ, БАЗИ ДАНИХ, ЧАСОВИЙ РЯД, MQTT.

ABSTRACT

Text part of the master's qualification work: 80 pages, 1 add., 28 pictures, 37 sources.

The aim of the work is to develop a testing system for the maximum load of time series databases.

Research object – the process of testing time series databases.

Research subject – determining the capabilities of time series databases.

Brief content of the work: The paper includes the development of a testing system for the maximum load of time series databases. The main testing methods are analyzed. The main types of databases are analyzed. The operation of an MQTT broker is analyzed.

KEYWORDS: TESTING, DATABASES, TIME SERIES, MQTT.

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут _________________________

ПОДАННЯ
ГОЛОВІ ЕКЗАМЕНАЦІЙНОЇ КОМІСІЇ
ЩОДО ЗАХИСТУ КВАЛІФІКАЦІЙНОЇ РОБОТИ
на здобуття освітнього ступеня бакалавра (магістра)

Направляється здобувач(ка) ______________________ до захисту кваліфікаційної роботи
(Прізвище та ініціали)

за спеціальністю ________________________________
(код, найменування спеціальності)

освітньо-професійної програми ____
(назва)

на тему: «____».

Кваліфікаційна робота і рецензія додаються.

Директор ННІ ________________ ____________________
(підпис) *(Ім'я, Прізвище)*

Висновок керівника кваліфікаційної роботи

Здобувач(ка) ____
____
____
____

Все це дозволяє оцінити виконану кваліфікаційну роботу здобувача(ки) ____________________ на оцінку «_____________» та присвоїти йому(їй) кваліфікацію ______________________________.

Керівник кваліфікаційної роботи _________ __________________
(підпис) *(Ім'я, Прізвище)*

«____» ____________ 20___ року

Висновок кафедри про кваліфікаційну роботу

Кваліфікаційна робота розглянута. Здобувач(ка) _*прізвище та ініціали*_ допускається до захисту даної роботи в Екзаменаційній комісії.

Завідувач кафедрою __________________ ______________ __________________
(назва) *(підпис)* *(Ім'я, Прізвище)*

ВІДГУК РЕЦЕНЗЕНТА на кваліфікаційну роботу на здобуття освітнього ступеня магістра

здобувача(ки) вищої освіти ____

(прізвище, ім'я, по батькові)

на тему «____»

Актуальність.

____

____

____

Позитивні сторони.

1.
2.
3.

Недоліки.

1.
2.

Відзначені зауваження не впливають на загальну позитивну оцінку кваліфікаційної магістерської роботи.

Висновок: *кваліфікаційна робота на здобуття ступеня магістр) заслуговує оцінку "______", а здобувач(ка) ___________________ заслуговує присвоєння кваліфікації: ……………………………..*

Рецензент: ________________ ________________________

науковий ступінь, вчене звання *підпис* *Ім'я, Прізвище*

ЗМІСТ

ВСТУП
1 ТЕОРЕТИЧНІ ОСНОВИ ТЕСТУВАННЯ БАЗ ДАНИХ 12
1.1 Загальні поняття тестування баз даних 12
1.1.1 Визначення та роль тестування у розробці баз даних 12
1.1.2 Процес тестування: етапи та взаємозв'язок 13
1.2 Види, методи та техніки тестування 20
1.2.1 Види тестування баз данних 20
1.2.2 Методи тестування 25
1.2.3 Техніки тестування 26
1.2.4 Рівні тестування 28
1.2.5 Автоматизація тестування: переваги та недоліки 29
1.2.6 Вибір та використання інструментів тестування 30
1.3 Бази даних 32
1.3.1 Типи бази даних та їхзастосування 32
1.3.2 Масштабованість баз даних 38
1.4 Інтернет речей (IoT) 41
1.4.1 Архітектура інтернет речей 41
1.4.2 Модулі інтернета речей 45
1.4.3 IoT протоколи передачі данних 46
2. АНАЛІЗ ВИКОРИСТАННЯ БАЗ ДАНИХ ЧАСОВИХ РЯДІВ 48
2.1 Робота з часовими рядами в базах даних 48
2.1.1 Особливості часових рядів та їхнє використання 48
2.1.2 Аналіз можливостей баз даних для роботи з часовими рядами 51
2.1.3 Технології зберігання та опрацювання часових рядів в базах даних 55
2.1.4 Вибір баз даних для зберігання часових рядів 57
2.1.5 Оптимізація та індексація для ефективного пошуку 62
2.1.6 Взаємодія з графіками та візуалізація даних 64

3 РОЗРОБКА МОДЕЛІ ТА ТЕСТУВАННЯ МАКСИМАЛЬНОГО НАВАНТАЖЕННЯ БАЗИ ДАНИХ ЧАСОВИХ РЯДІВ 66
3.1 Планування та проектування моделі 66
3.1.1 Планування та аналіз вимог до системи розробки тестування бази даних часових рядів 66
3.1.2 Дизайн системи тестування бази даних часових рядів 68
3.2 Архітектура моделі та її розробка 71
3.2.1 Робота з часовими рядами в Python 71
3.2.2 Застосування проколу MQTT для одночасного керування групою серверів. 74
3.2.3 Розробка та впровадження системи тестування максимального навантаження бази даних часових рядів 79
3.3 Тестування бази даних часових рядів 81
3.3.1 Тестування функціональності та навантаження 81
3.3.2 Аналіз результатів тестування 85
ВИСНОВКИ 88
ПЕРЕЛІК ПОСИЛАНЬ 90
ДОДАТКИ 93
ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ (Презентація) 94

ВСТУП

Наш світ знаходиться в дуже динамічному стані. Особливо це можна помітити в розвитку мобільних пристоїв, безпроводних технологій, зокрема інтернет речей. Все це потребує налагодження певної інфраструктури для взаємодіїї між різними пристороями, базами данних та комунікаторами. Потрібно побудувати так систему взаємодій щоб вона була ефективною, гнучкою, безпечною та при цьому недорогою, тобто було використано необхідну кількість ресурсів, які зможуть опрацювати кількість данних, що надійшли від користувача або змінилися за певний проміжок часу.

В даній дипломній роботі я хочу розкрити тему "Розробка системи тестування максимального навантаження баз даних часових рядів".

Актуальність теми: зі збільшенням кількості IoT пристроїв та збільшенням споживання ресурсів для цих пристроїв потрібно точно знати скільки ресурсів необхідно для підтримки житездатності IoT, що можливо забезпечити за допомогою тестування навантаження баз даних часових рядів.

Метою магістерської роботи є побудова системи тестування максимального навантаження баз даних часових рядів.

Завданнями є:

- проаналізувати підходи та методи тестування;
- проаналізувати, визначити особливості вибраної бази даних часових рядів
- експерементально перевірити дану систему тестування бази данних
- визначети тенденціїї розвитку в майбутньому;
- розробити рекомендаціїї щодо використання баз даних часових рядів.

Об'єкт дослідження: процес тестування бази данних часових рядів

Предметом дослідження є визначення можливостей баз даних часових рядів.

Методи дослідження. Емпіричні та теоретичні.

Науковою новизною одержаних результатів є розробка системи тестування баз даних часових рядів, що допоможе визначити оптимальні характеристики бази данних для використання під певні потреби.

1 ТЕОРЕТИЧНІ ОСНОВИ ТЕСТУВАННЯ БАЗ ДАНИХ

1.1 Загальні поняття тестування баз даних

1.1.1 Визначення та роль тестування у розробці баз даних

Тестування у розробці баз даних є важливою складовою процесу створення надійних та ефективних інформаційних систем. Воно охоплює широкий спектр аспектів, спрямованих на перевірку функціональності, продуктивності, безпеки та відновлення даних.

Тестування бази даних - це тип тестування програмного забезпечення, під час якого перевіряється схема, таблиці, тригери та інші компоненти тестованої бази даних. Воно також перевіряє цілісність та узгодженість даних. Це може включати створення складних запитів для завантаження/стрес-тестування бази даних та перевірки її реакції.

Тестування бази даних є важливою складовою тестування програмного забезпечення, оскільки воно гарантує, що значення даних та інформації, які отримані та збережені в базі даних, є правдивими. Тестування бази даних допомагає уникнути втрати даних, зберігає дані від перерваних транзакцій та запобігає несанкціонованому доступу до інформації. База даних є важливою для будь-якого програмного застосункуюю, так як об'єктивна необхідність у такому тестуванні БД, насамперед, для перевірки цілісності даних та їхньої достовірності, оскільки зберігаються дані, необхідні для роботи додатка.

Для спрощення взаємодії з бекендом використовують процедури перегляду та зберігані процедури. Ці процедури стосуються критично важливих повсякденних завдань; зокрема, вставки у базу даних користувачів (імена, контактні дані) або фінансових даних (про продажі та ціни). Такі процедури повинні бути тестовані кілька разів і на різних рівнях.

У базу даних можуть завантажуватися дані з кількох джерел, включаючи менш надійні, і завжди існує ймовірність надходження до бази даних некоректних

або шкідливих даних. Тому регулярна перевірка цілісності та узгодженості бази даних є необхідною.[1]

1.1.2 Процес тестування: етапи та взаємозв'язок

Життєвий цикл розробки програмного забезпечення (SDLC) - це економічний та швидкий процес, який використовують групи розробників для проектування та створення високоякісного програмного забезпечення. Мета SDLC - мінімізувати ризики проекту за рахунок попереднього планування, завдяки чому програмне забезпечення буде відповідати очікуванням клієнтів під час виробництва та на інших етапах. У цій методології описано кілька етапів, які розбивають процес розробки ПЗ на завдання, які можна розподіляти, виконувати та оцінювати.

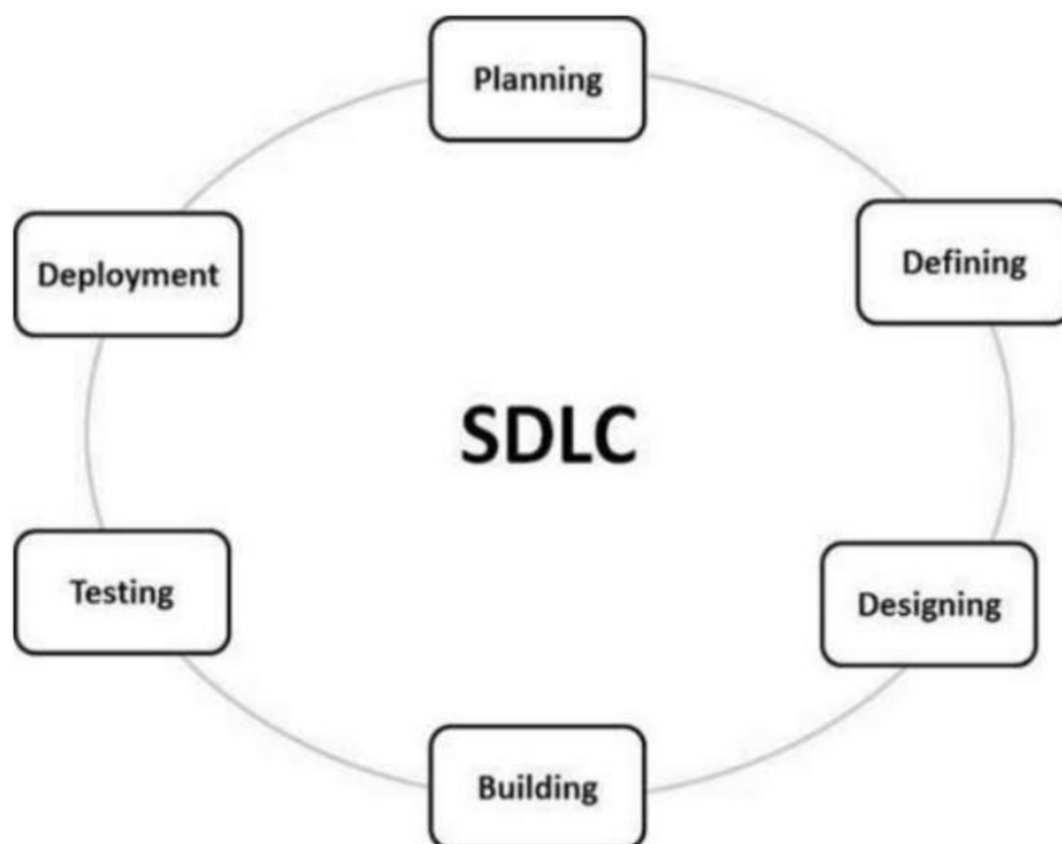

Рис. 1.1 Модель SDLC

SDLC включає наступні етапи розробки програмного забезпечення:

Етап 1: Планування та Аналіз Вимог

Аналіз вимог є найважливішим і фундаментальним етапом у SDLC. Виконується старшими членами команди за участю замовника, відділу збуту, результатів маркетингових досліджень та експертів в галузі промисловості. Цю інформацію використовують для планування основного підходу до проекту та проведення економічного, операційного та технічного аналізу його доцільності.

Також на етапі планування розробляються стратегії забезпечення якості та визначаються ризики, пов'язані з проектом. Результат технічного аналізу доцільності полягає у визначенні різних технічних підходів, які можуть бути використані для успішної реалізації проекту з мінімальними ризиками.

Етап 2: Визначення Вимог

Після завершення аналізу вимог наступним кроком є чітке визначення та документування вимог продукту та їх затвердження замовником чи аналітиками ринку. Це виконується через документ SRS (Специфікація Вимог до Програмного Забезпечення), який містить всі вимоги до продукту, що будуть розроблятися та вдосконалюватися протягом життєвого циклу проекту.

Етап 3: Проектування Архітектури Продукту

Специфікація вимог (SRS) є посиланням для архітекторів продукту для визначення найкращої архітектури продукту, що буде розроблятися. На основі вимог, визначених у SRS, зазвичай пропонується більше одного підходу до проектування архітектури продукту і документується в DDS - Документі Специфікації Проектування.[2]

Цей DDS переглядається всіма важливими зацікавленими сторонами, і на основі різних параметрів, таких як оцінка ризиків, стійкість продукту, модульність дизайну, обмеження бюджету та часу, обирається найкращий дизайн для продукту.

Дизайн підходу чітко визначає всі архітектурні модулі продукту разом із представленням потоку комунікації та обміну даними з зовнішніми та сторонніми

модулями (якщо такі є). Внутрішній дизайн всіх модулів запропонованої архітектури повинен бути чітко визначений з усіма дрібницями в DDS.

Етап 4: Будівництво чи Розробка Продукту

На цьому етапі SDLC розпочинається фактична розробка, і продукт будується. Програмний код генерується відповідно до DDS. Якщо дизайн виконаний деталізовано та організовано, генерацію коду можна виконати без зайвих труднощів.

Розробники повинні дотримуватися правил кодування, визначених їх організацією, і використовувати інструменти програмування, такі як компілятори, інтерпретатори, засоби відлагодження та інше для генерації коду. Для написання коду використовуються різні високорівневі мови програмування, такі як C, C++, Pascal, Java та PHP. Вибір мови програмування здійснюється з урахуванням типу розроблюваного програмного забезпечення.[3]

Етап 5: Тестування продукту

Цей етап зазвичай є підмножиною всіх етапів, оскільки в сучасних моделях SDLC тестувальні дії в основному включені в усі етапи SDLC. Однак цей етап вказує лише на етап тестування продукту, де повідомляються, відстежуються, виправляються та знову тестуються дефекти продукту, поки продукт не досягне стандартів якості, визначених у SRS.

Етап 6: Розгортання на ринку та обслуговування

Після того, як продукт випробований і готовий до розгортання, його офіційно випускають на відповідний ринок. Іноді розгортання продукту відбувається поетапно відповідно до бізнес-стратегії організації. Продукт може спочатку бути випущений в обмеженому сегменті та випробуваний в реальному бізнес-середовищі (UAT - тестування на прийняття користувачем).

Потім, враховуючи зворотний зв'язок, продукт може бути випущений таким, яким він є, або з пропонованими вдосконаленнями в цільовому сегменті ринку. Після випуску продукту на ринок здійснюється його обслуговування для існуючої бази клієнтів.[4]

Введення тестування на більш ранніх етапах допомогає зменшити кількість дефектів та допомогає заощадити ресурси.

Так само як і розробка ПЗ, так і тестування має свій життєвий цикл.

Життєвий цикл тестування програмного забезпечення (STLC) - це послідовність конкретних дій, які виконуються під час тестування з метою досягнення визначених цілей якості програмного продукту. Цей цикл включає в себе як верифікацію, так і валідацію. Навіть при тому, що тестування програмного забезпечення не є просто відокремленою дією, воно складається з ряду заходів, проведених методологічно для сертифікації вашого програмного продукту. STLC представляє собою життєвий цикл тестування програмного забезпечення.

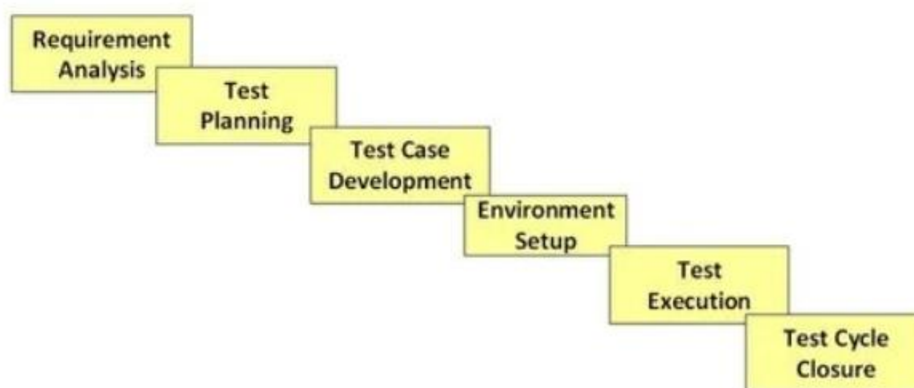

Рис. 1.2 Модель STLC

Тестування програмного забезпечення включає наступні етапи:

1. Аналіз вимог
2. Планування тестування
3. Розробка тестового випадку
4. Налаштування тестового середовища
5. Виконання тесту
6. Закриття тестового циклу

Фаза тестування вимог, також відома як аналіз вимог, полягає в тому, що група тестування вивчає вимоги з точки зору тестування, щоб визначити перевіряємі вимоги. Група забезпечення якості може взаємодіяти з різними зацікавленими сторонами для детального розуміння вимог. Вимоги можуть бути

функціональними чи нефункціональними. На цьому етапі також проводиться техніко-економічне обгрунтування проекту тестування.

Дії на етапі тестування вимог:

- Визначити типи тестів, які необхідно виконати.
- Зібрати детальну інформацію про пріоритети та фокус тестування.
- Підготувати матрицю відстеження вимог (RTM).

Визначити деталі тестового середовища, де планується проводити тестування.

Аналіз можливості автоматизації (при необхідності).

Результати етапу тестування вимог:

- RTM.
- Техніко-економічне обгрунтування автоматизації (якщо було рішення про її застосування).

Етап планування тестування в STLC - це етап, на якому вищий менеджер з забезпечення якості визначає стратегію плану тестування, а також зусилля та бюджетні витрати на проект. Крім того, також визначаються ресурси, тестове середовище, обмеження тестування та графік тестування. План тестування готується та ухвалюється на цьому етапі.

Дії по плануванню тестування:

- Підготовка плану/стратегії тестування для різних типів тестування.
- Вибір тестового інструменту.
- Оцінка зусиль з тестування.
- Планування ресурсів та визначення ролей та обов'язків.
- Вимоги до навчання.

Результати планування тестування:

- Документ плану/стратегії тестування.
- Документ із оцінкою зусиль.

Етап розробки тестового прикладу

Етап розробки тестового прикладу включає створення, перевірку та уточнення тестових прикладів і сценаріїв після того, як готовий план тестування.

Спочатку ідентифікуються дані для тестування, потім створюються, аналізуються і, в кінцевому рахунку, уточнюються на основі початкових умов. Після цього команда забезпечення якості розпочинає процес розробки тест-кейсів для окремих модулів.

Дії на етапі розробки тестових прикладів:

- Створення тест-кейсів, сценаріїв автоматизації (за необхідності).

- Огляд і базові тестові сценарії та сценарії.

- Створення тестових даних (якщо тестове середовище доступне).

Результати розробки тест-кейсу:

- Тестові випадки/скрипти.

- Тестові дані.

Налаштування тестового середовища

Налаштування тестового середовища визначає умови програмного та апаратного забезпечення, при яких тестується робочий продукт. Це один з найважливіших аспектів процесу тестування, який можна виконувати паралельно з етапом розробки тестового сценарію. Група тестування не може брати участь у цій діяльності, якщо група розробників надає тестове середовище. Група тестування зобов'язана провести перевірку готовності (димове тестування) даного середовища.

Дії на етапі налаштування тестового середовища:

- Вивчення необхідної архітектури, налаштування середовища та підготовка переліку вимог до обладнання та програмного забезпечення для тестового середовища.

- Налаштування тестового середовища та тестових даних.

- Виконання димового тестування збірки.

Результати налаштування тестового середовища:

- Готове середовище із налаштуванням тестових даних.

- Результати димового тестування.

Етап виконання тесту здійснюється тестувальниками, на яких тестування збірки програмного забезпечення здійснюється на основі підготовлених планів

тестування та тестових прикладів. Цей процес включає виконання тестового сценарію, його обслуговування та надсилання звітів про помилки. Якщо повідомляється про помилки, інформація повертається команді розробників для виправлення, і проводиться повторне тестування.[5]

Дії по виконанню тестів:

- Виконання тестів відповідно до плану.

- Документування результатів тестів та реєстрація дефектів у разі невдалих випадків.

- Порівняння дефектів з тестовими прикладами в RTM.

- Повторне тестування виправлених дефектів.

- Відстеження дефектів до закриття.

Результати виконання тесту:

- Завершений RTM із статусом виконання.

- Оновлені тестові приклади з результатами.

- Звіти про дефекти.

- Завершення іспитувального циклу

Етап завершення іспитувального циклу включає в себе декілька дій, таких як звіт про завершення тесту, збір матриць завершення тесту та результатів тесту. Члени команди тестування зустрічаються, обговорюють і аналізують артефакти тестування, щоб визначити стратегії, які необхідно реалізувати в майбутньому, виходячи з уроків поточного циклу тестування. Ідея полягає в усуненні звужених місць процесу для майбутніх циклів тестування.

Дії по завершенню тестового циклу:

- Оцінка критеріїв завершення циклу на основі часу, тестового покриття, вартості, програмного забезпечення, критичних бізнес-цілей та якості.

- Підготовка тестових метрик на основі вищезазначених параметрів.

- Документування результатів навчання в рамках проекту.

- Підготовка звіту про завершення тесту.

- Якісна і кількісна звітність про якість робочого продукту перед замовником.

- Аналіз результатів тестування для визначення розподілу дефектів за типами та серйозністю.

Результати завершення іспитувального циклу:

- Звіт про завершення тесту.
- Тестові метрики.[6]

1.2 Види, методи та техніки тестування

1.2.1 Види тестування баз данних

Виділяют три види тестування баз даннних:

I - структурне

II - функціональне

III - нефункціональне

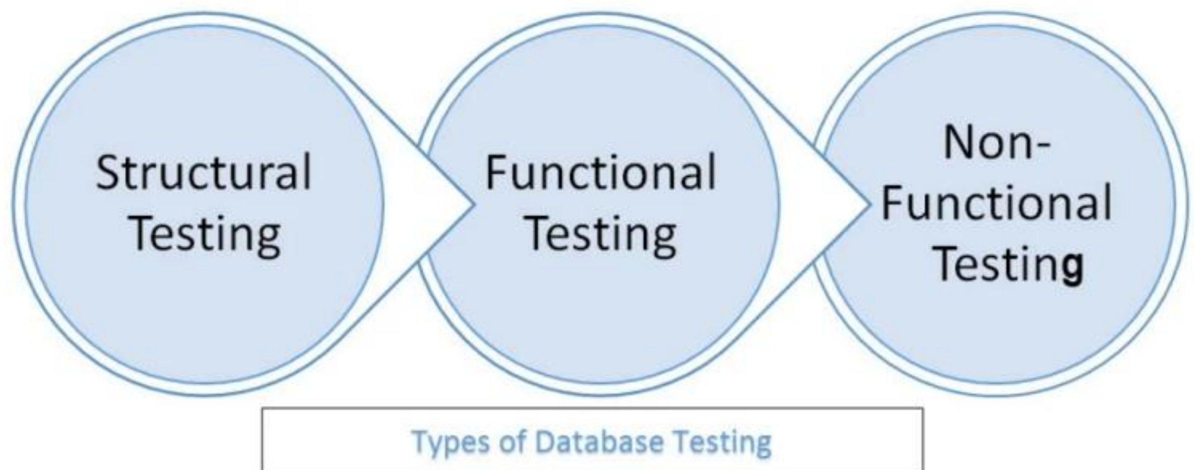

Рис. 1.3 Види тестування баз даних

I. Структурне тестування бази даних – це метод тестування бази даних, що перевіряє всі елементи усередині сховища даних, які в основному використовуються для зберігання інформації і якими кінцеві користувачі не можуть безпосередньо маніпулювати. Перевірка серверів баз даних також є важливим фактором при структурному тестуванні бази даних.

Структурне тестування включає в себе:

- тестування схеми,

- тестування ключів та індексів,
- тестування збережених процедур,
- тригерне тестування,
- перевірка сервера бази даних.

1. Тестування схеми бази даних включає в себе процес перевірки правильності та ефективності структури бази даних, включаючи таблиці, зв'язки між ними, обмеження та інші елементи, що визначають організацію та взаємодію збережених даних.

Основні етапи тестування схеми бази даних включають:

- Перевірка таблиць: Визначення правильності створення таблиць, їх полів і типів даних.

- Аналіз зв'язків: Перевірка коректності визначення зв'язків між таблицями та їх правильне використання.

- Тестування обмежень: Перевірка роботи обмежень, таких як унікальність, зовнішні ключі, перевірки на цілісність даних.

- Проведення тестів на вставку та оновлення даних: Перевірка, як схема реагує на операції вставки, оновлення та видалення даних.

- Тестування індексів: Визначення ефективності використання індексів для прискорення операцій з даними.

- Валідація відносин з іншими об'єктами системи: Перевірка взаємодії бази даних з іншими компонентами системи.

- Тестування забезпечення безпеки: Перевірка прав доступу та заходів безпеки схеми бази даних.

Цей процес дозволяє забезпечити, що схема бази даних відповідає вимогам проекту, сприяє ефективному використанню ресурсів та забезпечує надійність та інтегрітет даних.

2. Тестування ключів та індексів у базі даних включає в себе перевірку правильності та ефективності використання ключів і індексів для організації та оптимізації доступу до даних.

Основні етапи тестування ключів та індексів включають:

- Перевірка унікальності ключів: Визначення, чи використовуються унікальні ключі належним чином та чи вони забезпечують унікальні значення.

- Тестування зовнішніх ключів: Перевірка коректності визначення та функціонування зовнішніх ключів, що вказують на зв'язки між таблицями.

- Валідація індексів: Перевірка, як індекси впливають на продуктивність операцій вибірки, сортування та фільтрації даних.

- Тестування швидкодії використання індексів: Перевірка того, як ефективно індекси прискорюють операції пошуку та фільтрації даних.

- Перевірка використання автоматичних індексів: Визначення, чи ефективно використовуються автоматично створені індекси базою даних.

- Тестування впливу на продуктивність системи: Оцінка впливу ключів та індексів на продуктивність операцій та завантаження бази даних.

- Тестування операцій вставки, оновлення та видалення: Перевірка впливу ключів та індексів на операції зміни даних.

Цей процес дозволяє забезпечити ефективне та надійне використання ключів та індексів для забезпечення швидкого та точного доступу до даних у базі даних.

3. Тестування збережених процедур – це процес перевірки та валідації функціональності збережених процедур в базі даних. Збережені процедури представляють собою набір інструкцій або операцій, які виконуються в базі даних. Тестування хранимых процедур включає в себе перевірку правильності їх виконання, взаємодії з іншими об'єктами бази даних, а також валідацію введених і виведених даних.

4. Тригери в базах даних - це збережені фрагменти коду, які автоматично виконуються (спрацьовують), коли відбуваються певні події або умови в базі даних.

Тестування тригерів включає в себе перевірку їхньої правильності та ефективності у виконанні певних дій при виникненні відповідних подій. Основні аспекти тестування тригерів включають:

- Перевірка коректності коду: Визначення, чи виконують тригери правильні дії та чи не містять помилок.

- Тестування спрацювання тригерів: Перевірка, чи вони активуються при виникненні певних подій чи умов.

- Валідація результатів виконання: Перевірка коректності змін, які вносяться тригерами в базу даних.

- Тестування обмежень і безпеки: Перевірка, чи враховуються обмеження доступу та безпеки при виконанні тригерів.

- Тестування ефективності: Оцінка впливу тригерів на продуктивність бази даних та чи вони викликають зайве навантаження.

- Таке тестування допомагає впевнитися, що тригери працюють як очікується та не порушують цілісність та безпеку даних.

Перевірка сервера бази даних - це процес, під час якого оцінюється працездатність та правильність функціонування сервера, на якому розміщена база даних. Це важливий етап в загальному процесі тестування бази даних і включає в себе декілька ключових аспектів:

- Запуск та зупинка сервера: Перевірка, чи коректно запускається та зупиняється сервер бази даних за вказаними процедурами.

- Налаштування та конфігурація: Визначення, чи встановлені та налаштовані параметри сервера бази даних відповідно до вимог.

- Валідація доступу: Перевірка, чи можуть користувачі та програми отримувати доступ до сервера та бази даних відповідно до встановлених прав доступу.

- Моніторинг ресурсів: Оцінка використання ресурсів сервера, таких як CPU, пам'ять та диск, під час роботи з базою даних.

- Тестування відмовостійкості: Перевірка, як сервер реагує на відмови та відновлення роботи після аварійного відключення.

- Тестування безпеки: Перевірка, чи дотримуються заходи безпеки на рівні сервера та чи виявляються та вирішуються потенційні загрози.

- Тестування резервного копіювання та відновлення: Перевірка можливості виконання резервного копіювання та відновлення даних на сервері бази даних.

Цей процес дозволяє впевнитися в тому, що сервер бази даних працює надійно, ефективно та забезпечує необхідну продуктивність та безпеку для коректної роботи бази даних.[6]

II. Функціональне тестування

Функціональне тестування — це вид тестування програмного забезпечення, спрямований на оцінку функціональності системи та перевірку відповідності її функцій визначеним вимогам. Основна мета функціонального тестування полягає в перевірці того, чи виконує програма ті функції, для яких вона була розроблена, та чи робить це правильно.

У ході функціонального тестування емулюються різноманітні сценарії використання програми, тестуються різні входи та перевіряється коректність виходів. Також перевіряється взаємодія програми з користувачем, зовнішніми системами, обробка даних та виконання операцій.

Функціональне тестування включає в себе створення тестових сценаріїв, виконання тестів та аналіз результатів з метою виявлення дефектів та підтвердження відповідності програми визначеним вимогам. Цей процес допомагає забезпечити, що програмне забезпечення відповідає очікуванням користувачів та виконує свої функції ефективно та надійно.

Функціональне тестування баз даних включає в себе оцінку того, як база даних взаємодіє з додатками та системами. Перевіряється правильність операцій створення, читання, оновлення та видалення даних. Також перевіряється валідація SQL-запитів, забезпечення коректного зберігання та обробки різноманітних типів даних, виконання транзакцій та використання індексів та ключів. Тестується також схема бази даних, включаючи правильність взаємозв'язків та безпеку даних. Функціональне тестування допомагає переконатися в тому, що база даних відповідає функціональним вимогам та забезпечує надійну обробку даних.

III. Нефункціональне тестування

Нефункціональне тестування важливе для оцінки якості бази даних, оскільки воно вивчає аспекти, які не впливають безпосередньо на її функціональність, але мають важливе значення для ефективності, стабільності та відповідності вимогам

та стандартам. У контексті баз даних нефункціональне тестування охоплює оцінку аспектів, таких як продуктивність, безпека, доступність, стійкість до помилок та інші фактори, які визначають загальний досвід користувача та роботу системи.

Нефункціональне тестування охоплює дуже багато різних тестів:

- стрес тест
- тест продуктивності
- тест на безпеку
- тест масштабованості
- тест на сумісность
- тест на доступність

Для забезпечення якісного тестування потрібно виконувати вимоги щодо техніки тестування.[7]

1.2.2 Методи тестування

Основні методи тестування:

- Тестування методом чорного ящика (Black-box testing)
- Тестування методом білого ящика (White-box testing)
- Ручне тестування
- Автоматизоване тестування (auto testing)

Тестування "білого ящика" передбачає, що тестувальник має повний доступ до вихідного коду програми та може аналізувати його для виявлення помилок і недоліків. Зазвичай цей метод використовується розробниками для перевірки свого коду на правильність.

Тестування "чорного ящика" полягає в перевірці функціональності програми без доступу до її вихідного коду. Тестувальник оцінює роботу програми, ґрунтуючись на зовнішніх проявах та поведінці, не знаючи, як вона устроєна всередині.

Тестування "сірого ящика" - це комбінація двох попередніх методів. Тестувальник має частковий доступ до вихідного коду або знає деякі деталі його роботи, але в основному оцінює поведінку програми ззовні.

Ручне тестування - це процес перевірки програмного забезпечення вручну, без використання автоматизованих інструментів. Тестувальник самостійно взаємодіє з програмою, перевіряючи її працездатність та правильність виконання завдань.

Автоматизоване тестування - це використання спеціалізованих програм і інструментів для автоматизації процесу тестування. Тестувальник розробляє набір тестових сценаріїв, які потім виконуються автоматично.[8]

1.2.3 Техніки тестування

До основних технік тестування відносять:

- тестування еквівалентності або техніка розділення еквівалентних класів передбачає розбиття тестових даних на класи, де всі елементи схожі якимось чином. Ця методика має сенс лише тоді, коли компоненти схожі та можуть вписатися в спільну групу. Вибір цієї техніки означає, що ми будемо тестувати лише кілька значень з кожної групи. Слід пам'ятати, що це не гарантує, що решта значень, які не охоплені тестами, будуть вільні від помилок. Ми лише припускаємо, що використання кількох елементів з групи буде достатньо ілюстративним. Розділення еквівалентних класів є гарним рішенням для випадків, коли ви маєте справу з великим обсягом вхідних даних або численними ідентичними варіаціями введення. У іншому випадку може мати сенс покрити продукт тестами більш докладно.

- тестування граничних значень схоже на попередню техніку, тут також йде розбивка на класи, але тестується значення не з певного класу, а перевіряється значення на межах, тобто тих, які знаходяться на "межах" класів. Та ж сама логіка прекрасно працює для інтеграційного тестування. Ми перевіряємо менші елементи під час модульного тестування, і на наступному рівні помилки ймовірно виявляться на з'єднаннях модулів.

- попарне тестування, або комбінаторне тестування, використовує всі можливі комбінації пар параметрів системи для забезпечення високої покриття тестів при економії часу та ресурсів.

- таблиця прийняття рішень є інструментом для організації і визначення тестових випадків відповідно до різних умов або варіантів введення.

- діаграма станів та переходів використовується для візуального відображення різних станів системи та можливих переходів між ними.

- тестування варіантів використання Ця техніка орієнтована на перевірку, чи відповідає система вимогам та очікуванням кінцевих користувачів у конкретних сценаріях використання.

- доменне тестування, що фокусується на визначенні та перевірці меж вхідних даних, зокрема на границях і винятках, що може призвести до помилок або неправильного функціонування системи.

- вгадування помилок - це найбільш експериментальна практика серед усіх, зазвичай застосовується разом із іншою технікою проектування тестів. При вгадуванні помилок інженер з забезпечення якості передбачає місця, де ймовірно з'являться помилки, покладаючись на попередній досвід, знання системи та вимоги до продукту. Таким чином, спеціаліст із забезпечення якості повинен ідентифікувати місця, де дефекти зазвичай накопичуються, і приділяти підвищену увагу цим областям.

- State Transition (перехід стану) - це метод тестування, який візуалізує стани програмної системи в різні часові рамки та етапи її використання. Візуальна інформація легше сприймається порівняно з вербальним описом. Таким чином, метод переходу стану дозволяє швидко розробити остаточне тестування для систем з багатьма варіаціями станів. Цей метод ефективний при створенні тестових наборів для систем, які мають багато варіацій станів. Він буде корисним, якщо ви тестируєте послідовність подій з кількістю варіантів введення.[9]

1.2.4 Рівні тестування

Основні рівні тестування:

- Модульне тестування (Unit testing)
- Інтеграційне тестування (Integration testing)
- Системне тестування (System testing)
- Приймальне тестування (Acceptance testing)

1. Модульне тестування (Unit testing) - це вид тестування програмного забезпечення, який спрямований на перевірку і валідацію індивідуальних модулів або компонентів програми, щоб переконатися в їх коректності та відповідності вимогам. Мета полягає в перевірці правильності роботи окремих "одиниць" коду, зазвичай функцій чи методів, незалежно від інших частин програми. У випадку баз даних це означає тестування окремих одиниць бази даних, таких як таблиці, процедури, тригери, функції, індекси тощо.

2. Інтеграційне тестування (Integration testing) - це вид тестування програмного забезпечення, який спрямований на перевірку взаємодії та інтеграції між різними модулями, компонентами чи системами. Мета полягає в виявленні помилок та недоліків, що можуть виникнути під час взаємодії між різними частинами програми чи різними системами. Інтеграційне тестування дозволяє переконатися в тому, що компоненти взаємодіють правильно та спільно працюють в рамках великої системи.

3. Системне тестування (System testing) - це етап тестування програмного забезпечення, який оцінює повне функціональне поведінку системи як єдиного елементу. Мета системного тестування - переконатися в тому, що весь продукт, включаючи всі його компоненти та модулі, працює відповідно до визначених вимог та специфікацій.[3]

Під час системного тестування перевіряються різні аспекти системи, такі як функціональність, продуктивність, навантаження, безпека та інші. Тестування проводиться в реальному або максимально наближеному до реального

середовищі, щоб виявити та усунути можливі недоліки перед випуском продукту на ринок чи введенням його в експлуатацію.

Цей етап тестування допомагає забезпечити, що весь продукт в цілому відповідає вимогам та готовий до використання або впровадження.

4. Приймальне тестування (Acceptance Testing) - це етап тестування програмного забезпечення, яке проводиться для визначення того, чи відповідає система вимогам та чи може бути прийнята замовником або кінцевим користувачем.

На цьому етапі визначається, чи виконуються усі вимоги та функціональність, передбачені у специфікації, і чи задовольняє продукт потреби користувачів. Приймальне тестування може включати різні види тестів, такі як функціональні тести, тести ефективності, тести навантаження та інші, залежно від вимог та характеру програмного продукту.

Цей етап є критичним для визначення готовності продукту до випуску на ринок або використання в реальних умовах, і його результат визначає прийняття чи відмову від продукту замовником чи користувачем.[5]

1.2.5 Автоматизація тестування: переваги та недоліки

Автоматизація тестування має свої переваги та недоліки.

Переваги автоматизації тестування:

- швидкість виконання тестів: Автоматичні тести можуть бути виконані значно швидше, ніж їхні ручні аналоги, що дозволяє швидше отримати результати тестування.

- повторюваність: Автоматизовані тести завжди виконуються однаковим чином, що забезпечує стабільність результатів і дозволяє легше виявляти проблеми.

- покриття тестування: Автоматичні тести можуть включати велику кількість тестових сценаріїв, що дозволяє досягти широкого покриття функціональності продукту.

- ефективність ресурсів: При великій кількості тестів та частих змінах коду автоматизація може зекономити час і зусилля.[10]

Недоліки автоматизації тестування:

- високі витрати на розробку: Створення автоматизованих тестів може вимагати значних витрат часу та ресурсів, особливо для складних проектів.

- обмеженість виявлення дефектів: Автоматизовані тести, як правило, тестують те, що вже відомо, і можуть пропустити невідомі проблеми або нові функції.

- підтримка тестів: Зміни в коді можуть призвести до необхідності адаптації автоматизованих тестів, що вимагає постійного обслуговування.

- необхідність навчання команди: Впровадження автоматизації вимагає, щоб команда була ознайомлена зі специфічними інструментами та методологіями автоматизації.

- неможливість повного заміщення ручного тестування: Деякі аспекти тестування, такі як тестування UX або виявлення складних бізнес-проблем, можуть бути ефективніше виявлені ручним тестуванням.[11]

1.2.6 Вибір та використання інструментів тестування

Інструменти тестування - це програмні додатки, призначені для поліпшення, автоматизації та спрощення процесу тестування в різних галузях та сферах застосування, таких як розробка програмного забезпечення, IoT-пристрої, веб- та мобільні додатки і т.д.

Існує багато інструментів для тестування баз даних, які надають різноманітні можливості для перевірки їхньої продуктивності, надійності та відповідності стандартам. Ось деякі популярні інструменти тестування баз даних.

- Unit. Це інструмент для тестування, який широко використовується в середовищі Java. Він може бути використаний для написання тестів для баз даних, які використовують Java.

- TestNG. Це фреймворк для тестування, який підтримує тестування баз даних та може використовуватися для автоматизації тестів.[12]

- Selenium. Хоча Selenium частіше використовується для автоматизації тестів веб-інтерфейсів, він також може бути використаний для тестування баз даних через відповідні бібліотеки.[13]

- DBUnit. Це Java-бібліотека, спеціально розроблена для тестування баз даних. Вона дозволяє вам створювати консистентний стан бази даних для тестування.[14]

- JUnitDB. Це розширення JUnit, призначене для тестування баз даних. Воно надає аннотації та інструменти для легкої інтеграції тестів баз даних в JUnit.

- Postman. Це інструмент для тестування API, але він також може бути використаний для тестування баз даних через SQL-запити.[15]

- Apache JMeter. Це інструмент для тестування продуктивності, але його також можна використовувати для тестування баз даних.[16]

- Pytest - це фреймворк для тестування на Python, який підтримує автоматизоване тестування баз даних. Він дозволяє легко писати тести та використовувати розширені можливості.

- SQLAlchemy - це бібліотека для роботи з базами даних у Python. Вона включає можливості тестування та допомагає легко виконувати SQL-запити.

- Django TestCase має вбудований функціонал тестування, такий як Django TestCase, надасть вам інструменти для тестування баз даних у веб-застосунках.

- Pytest-django - це розширення для Pytest, яке надає можливості тестування Django-застосунків, включаючи взаємодію з базою даних.[17]

- Factory Boy допомагає створювати тести з об'єктами для вашого коду, включаючи тестування моделей баз даних.

- Faker генерує випадкові дані, що може бути корисним для створення тестових наборів даних для тестування баз даних.

1.3 Бази даних

1.3.1 Типи бази даних та їх застосування

Якщо сказати в загальному, то можно бази даних поділити на реляційні та нереляційні бази даних. IT технології розвиваються дуже динамічно, бази даних також не відстають в цьому і тому на даний час вже існує велика кількість різних типів систем управління базами даних.

Виділемо основні типи баз даних:

- Реляційні
- Ключ-значення (key-value)
- Документо-орієнтовані
- Бази даних часових рядів
- Графові бази даних
- Пошукові бази даних
- Об'єктно-орієнтовані бази даних
- RDF (Resource Description Framework)
- Wide Column Stores
- Мультимодальні СУБД
- Native XML СУБД
- GEO/GIS (просторові) та спеціалізовані СУБД
- Event СУБД (бази даних переходів станів)
- Контентні СУБД
- Навігаційні (Navigational) СУБД
- Векторні бази даних

1. Реляційні бази даних - це структуровані бази даних, які базуються на концепції реляційної моделі даних. У цих базах даних дані представлені у вигляді таблиць, які складаються з рядків і стовпців. Кожна таблиця має унікальний ідентифікатор - первинний ключ, який дозволяє встановити взаємозв'язки між різними таблицями. Вони широко використовуються в сучасних системах управління базами даних для зберігання та організації даних.

Реляційні бази даних можуть мати різний спосіб організації зберігання даних: за рядками або за стовпцями. Наприклад, PostgreSQL використовує організацію за рядками, в той час як ClickHouse та Vertica використовують організацію за стовпцями. Бази даних з організацією за стовпцями, як правило, ефективніше використовуються для аналітичних завдань, тоді як бази даних з організацією за рядками можуть бути більш підходящими для транзакційних завдань.[18]

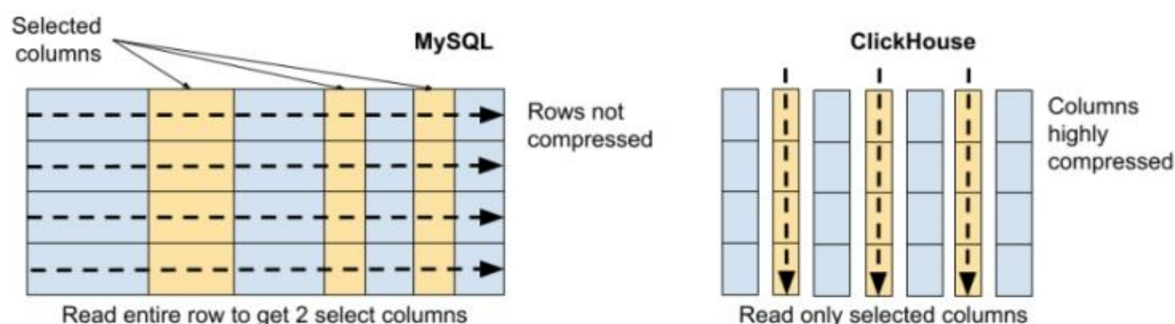

Рис. 1.4 Реляційні СУБД

2. Ключ-значення (key-value) тип баз даних призначений для швидких, практично миттєвих запитів для завдань, таких як кешування, відображення балансу тощо. Висока швидкість досягається завдяки зберіганню даних за принципом "ключ-значення", і у більшості випадків це здійснюється за участю оперативної пам'яті.

Словники містять колекцію об'єктів або записів, причому об'єкти мають низку різних полів, кожне з яких містить дані. Записи зберігаються та витягуються за допомогою ключа, який однозначно ідентифікує запис і використовується для швидкого пошуку даних.

Основним застосуванням є прискорення відображення даних для кінцевих користувачів і зниження навантажень, включаючи операції введення/виведення на інфраструктуру організацій.[19]

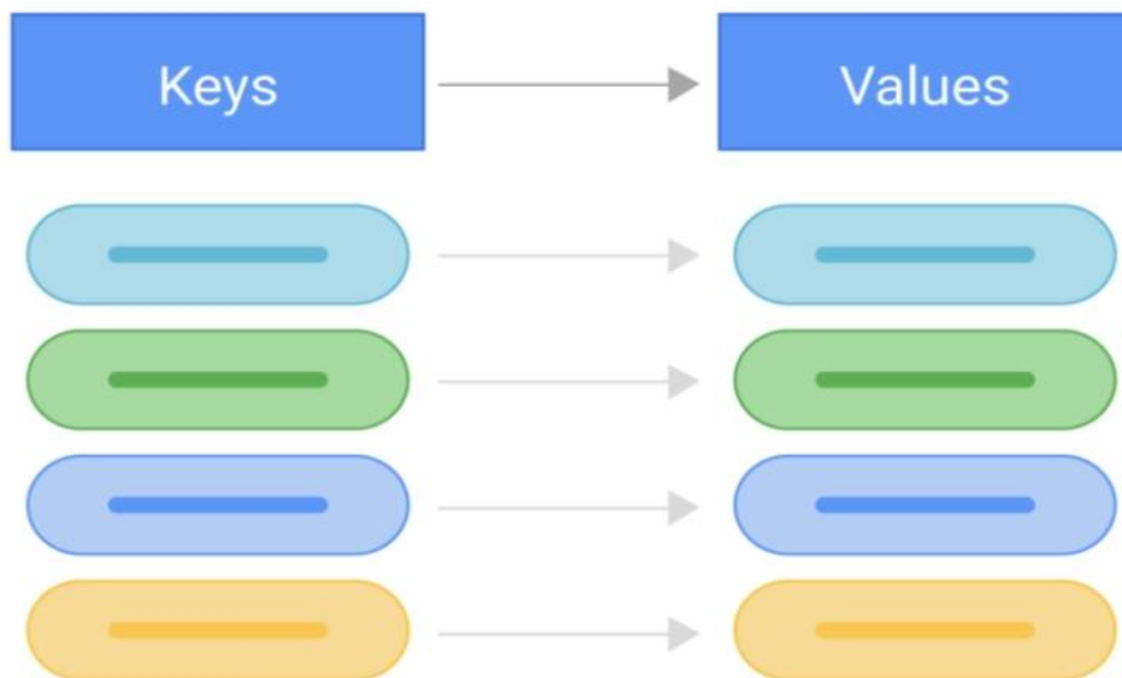

Рис.1.5 База даних типу Key-value

3. Бази даних часових рядів створені для збору та управління елементами, що змінюються з плиннем часу. Більшість таких баз даних організовані у структури, які фіксують значення для окремого елемента. Наприклад, можна створити таблицю для відстеження температури процесора. У кожному значенні буде часова мітка та показник температури. У таблиці може бути кілька метрик.

Особливості:

- орієнтовані на запис;

- призначені для обробки постійного потоку вхідних даних;

- продуктивність залежить від кількості відстежуваних елементів, інтервалу - опитування між записом нових значень та фактичного корисного навантаження даних.

Найпопулярнішими базами даних для часових рядів є Prometheus, InfluxDB, Graphite.

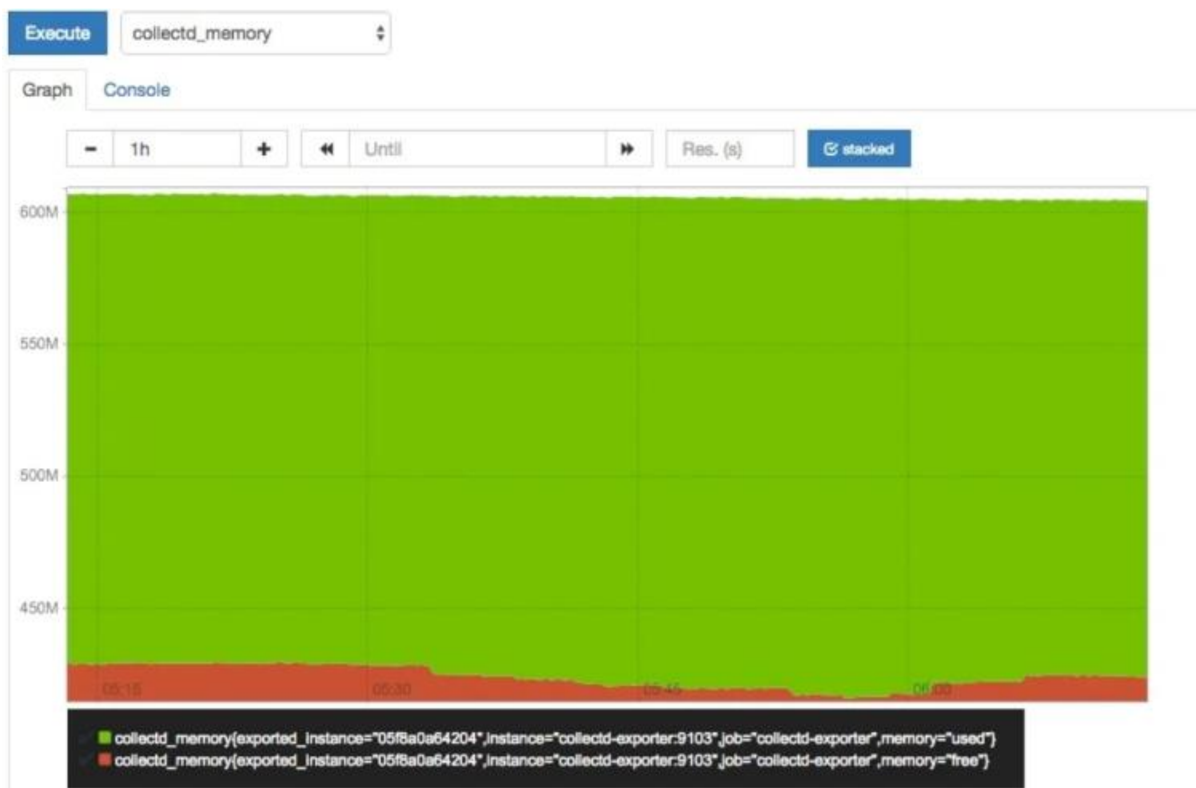

Рис. 1.6 Графічне представлення роботи БД часових рядів

4. Документо-орієнтовані бази даних призначені для зберігання ієрархічних структур даних (документів). Основою цих баз є документні сховища, які мають структуру дерева або лісу. Дерева починаються з кореневого вузла і можуть містити кілька внутрішніх і листових вузлів. Листові вузли містять дані, які при додаванні документа заносяться в індекси, що дозволяє навіть при досить складній структурі знаходити шлях до шуканих даних. У відміну від сховищ типу "ключ-значення", вибірка за запитом до документного сховища може містити частини великої кількості документів без повного завантаження цих документів в оперативну пам'ять.

На сьогоднішній день найпопулярнішою документо-орієнтованою базою даних є MongoDB.

Із особливойте документо-орієнтованої БД є:

- База даних не визначає певний формат або схему.

- Кожен документ може мати власну внутрішню структуру.

- Документні бази даних є відмінним вибором для швидкої розробки.

- В будь-який момент можна змінювати властивості даних, не змінюючи структуру чи самі дані.

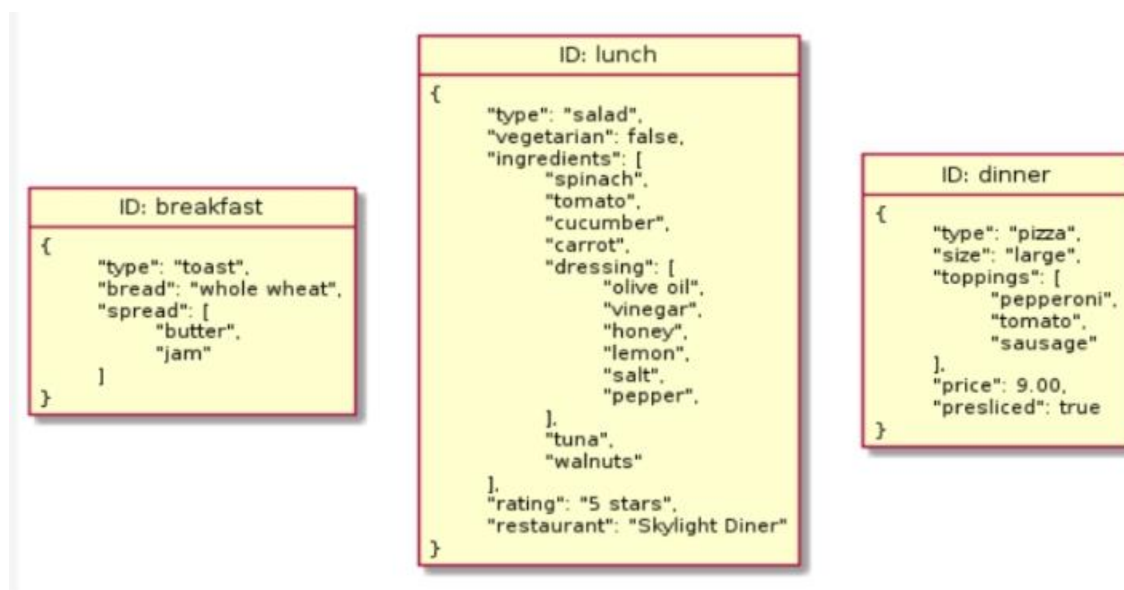

Рис. 1.7 Документо-орієнтована БД

5. Графова база даних - це систематичний набір даних, в якому акцентується на взаємозв'язках між різними сутностями даних. Ця база даних використовує математичну теорію графів для відображення зв'язків з даними. На відміну від реляційних баз даних, які зберігають дані в жорстких табличних структурах, графові бази даних зберігають дані у вигляді мережі сутностей та відносин. У результаті такі бази даних часто забезпечують більшу продуктивність і гнучкість, оскільки краще підходять для моделювання реальних сценаріїв.

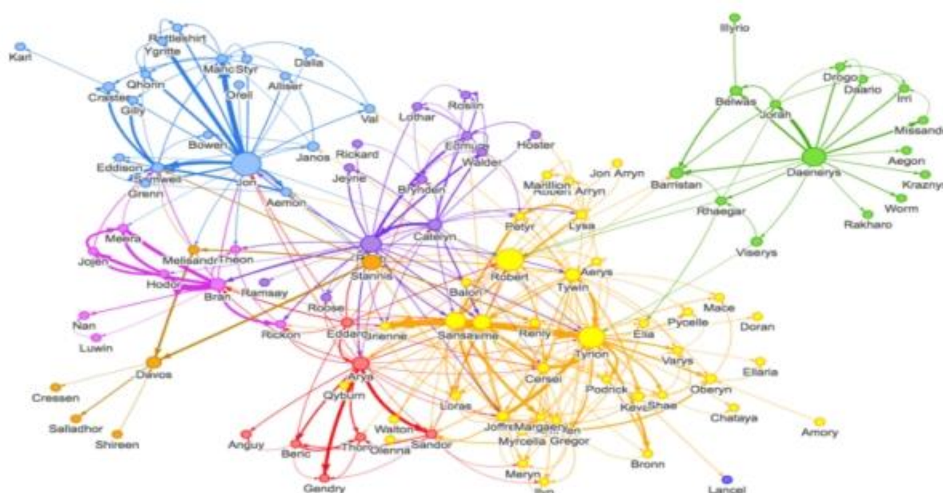

Рис. 1.8 Приклад візуалізації графа

6. Об'єктно-орієнтовані бази даних представляють собою системи управління базами даних, де інформація подається у вигляді об'єктів, аналогічно тому, як це відбувається у мовах програмування з об'єктно-орієнтованим підходом.

Ці бази даних виникли як засіб для нативної взаємодії коду, написаного за допомогою об'єктно-орієнтованих мов програмування, з базою даних.

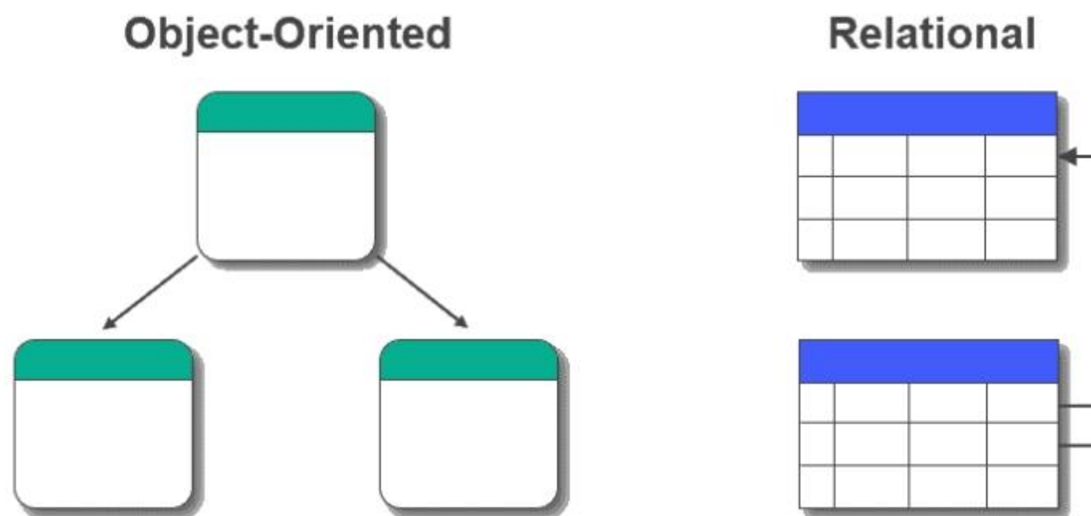

Рис. 1.9 Обєктно-орієнтована база даних та реляційна база даниз

7. Колоночні бази даних, також відомі як нереляційні колоночні сховища або бази даних з широкими стовпцями, відносяться до сімейства баз даних NoSQL, але зовні схожі на реляційні бази даних. Подібно до реляційних баз, колоночні бази даних зберігають дані, використовуючи рядки та стовпці, але з іншим відношенням між елементами.

У реляційних базах даних всі рядки повинні відповідати фіксованій схемі. Схема визначає, які стовпці будуть у таблиці, типи даних та інші критерії. У колоночних базах даних, замість таблиць, існують структури – "колоночні родини". Родини містять рядки, кожен з яких визначає власний формат. Рядок складається з унікального ідентифікатора, який використовується для пошуку, за яким слідують набори імен і значень стовпців.[20]

1.3.2 Масштабованість баз даних

Масштабованість в інформатиці означає здатність системи, мережі або процесу впоратися з збільшенням робочого навантаження (збільшувати свою продуктивність) за умови додавання ресурсів (зазвичай апаратних). Це важливий аспект електронних систем, програмних комплексів, систем баз даних, маршрутизаторів, мереж і т. д., якщо для них важлива здатність працювати під великим навантаженням. Систему називають масштабованою, якщо вона може збільшувати продуктивність пропорційно додатковим ресурсам. Масштабованість можна оцінити через співвідношення зростання продуктивності системи до зростання використовуваних ресурсів. У системи з поганою масштабованістю додавання ресурсів призводить лише до невеликого підвищення продуктивності, і з певного "порогового" моменту додавання ресурсів не призводить до ніякого корисного ефекту.

Існують два основних підходи до масштабування: вертикальне і горизонтальне.

Вертикальне масштабування (вглиб) - це традиційний метод збільшення обчислювальних потужностей. Замість невеликого сервера придбається більший багатопроцесорний сервер (або кластер серверів). Якщо цього недостатньо, можливо, знадобиться мейнфрейм або навіть суперкомп'ютер. Однак цей підхід вимагає значних витрат часу і фінансових ресурсів.

Горизонтальне масштабування (вшир) - це активно розвиваючийся підхід до збільшення обчислювальних потужностей системи за допомогою додавання нових обчислювальних вузлів. Замість одного сервера велика кількість взаємопов'язаних серверів розподіляє навантаження між собою. Цей підхід дозволяє оперативно збільшувати продуктивність системи. Обмеження масштабованості накладається лише архітектурою системи.

Також потрібно дослідити види масштабованості.

Партіціонування (partitioning) - це розбиття великих таблиць на логічні частини за обраними критеріями. Рекомендується використовувати

партіціонування, якщо в базі даних є декілька великих таблиць, більшість запитів при цьому стосується лише останньої частини (наприклад, останніх 5-10 записів). Це дозволяє розподіляти навантаження на таблицю відповідно до її партіцій. Наприклад, блог, де перша сторінка (останні 5-10 постів) становить 40–50% всього навантаження, є добрим прикладом.

Реплікація - це синхронне/асинхронне копіювання даних з головних серверів на ведучі (або можливо також головні) сервери. Головні сервери називають майстрами (master), ведучі - слейвами (slave). Для вирішення проблеми читання використовується майстер-слейв реплікація. Для вирішення проблеми запису потрібно використовувати майстер-майстер реплікацію. Реплікація також допомагає ізолювати навантаження, виниклі внаслідок важких запитів.

Шардування (шардинг) - це розділення даних на рівні ресурсів відповідно до вимог до навантаження.

Вертикальний шардинг - розподіл однієї бази даних веб-застосунка на дві або більше баз даних за допомогою виділення окремих модулів, без зміни логіки роботи веб-застосунка.

Горизонтальний шардинг - розподіл однотипних даних веб-застосунка між окремими базами даних. Наприклад, розташування рахунків користувачів на різних серверах з розподілом за їх ідентифікатором.

Шардування відмінно від вертикального масштабування, яке передбачає збільшення обчислювальних можливостей і обсягу носіїв інформації одного сервера баз даних, маючи об'єктивні фізичні обмеження. Шардування забезпечує кілька переваг, головна з яких - зниження витрат на забезпечення узгодженого читання та ізоляція навантаження.[21]

Згідно теореми CAP будь-яка реалізація розподілених обчислень можливо забезпечити не більше двох з трьох наступних властивостей:

- Спільність даних
- Доступність
- Стійкість до розподілення

Акронім CAP у назві теореми сформований з перших літер англійських найменувань цих трьох властивостей: (C)onsistency, (A)vailability, (P)artition tolerance. Принцип був запропонований професором Каліфорнійського університету в Берклі Еріком Брюером у липні 2000 року і отримав широку популярність серед фахівців з розподілених обчислень. Концепція NoSQL (Not only SQL), в рамках якої створюються розподілені нетранзакційні системи управління базами даних, часто використовує цей принцип як обґрунтування необхідності відмови від спільності даних.

Спільність даних (Consistency). Спільність даних передбачає, що в усіх обчислювальних вузлах в один момент часу дані не суперечать одне одному. Іншими словами, як тільки ми успішно записали дані в наше розподілене сховище, будь-який клієнт при запиті отримає ці останні дані, незалежно від того, до якого саме вузла він звернувся.

Доступність (Availability). Властивість доступності означає, що будь-який запит до розподіленої системи завершується коректною відповіддю, незалежно від того, до якого вузла був виконаний запит. Іншими словами, в будь-який момент часу клієнт може отримати дані з розподіленого сховища відповідно до свого запиту, або отримати відповідь про їх відсутність, якщо ці дані відсутні взагалі в сховищі.

Стійкість до розділення (Partition tolerance). Стійкість до розділення передбачає, що розщеплення розподіленої системи на декілька ізольованих секцій не призводить до некоректності відповіді від кожної з секцій. Іншими словами, втрата повідомлень між компонентами системи (можливо, навіть втрата всіх повідомлень) не впливає на працездатність системи. Тут дуже важливий момент полягає в тому, що якщо деякі компоненти виходять з ладу, це також підпадає під цей випадок, оскільки можна вважати, що ці компоненти просто втрачають зв'язок з усією іншою системою.

Згідно з теоремою CAP, розподілені системи, залежно від пари практично підтримуваних властивостей з трьох можливих, розпадаються на три класи.

Система, в усіх вузлах якої дані спільні і забезпечена доступність, жертвує стійкістю до розділення. Такі системи можливі на основі технологічного програмного забезпечення, яке підтримує транзакційність відповідно до вимог ACID (Атомарність, Спільність, Ізоляція, Надійність). Прикладами таких систем можуть бути рішення на основі систем управління базами даних, таких як Microsoft SQL Server, або розподілена служба каталогів LDAP.

Розподілена система, яка в кожний момент забезпечує цілісний результат і здатна функціонувати в умовах розділення, заради доступності може не видачувати відповідь. Стійкість до розділення вимагає забезпечення дублювання змін у всіх вузлах системи; у цьому контексті відзначається практична доцільність використання в таких системах розподілених песимістичних блокувань для збереження цілісності.

Розподілена система, яка відмовляється від цілісності результату. Хоча системи такого типу відомі задовго до формулювання принципу CAP (наприклад, розподілені веб-кеші або DNS), зростання популярності систем із цим набором властивостей пов'язане саме з поширенням теореми CAP. Завданням при побудові системи типу AP стає забезпечення якогось практично доцільного рівня цілісності даних, у цьому змісті про системи AP говорять як про "цілісні в кінцевому підсумку" або як про "слабко цілісні".[22]

1.4 Інтернет речей (IoT)

1.4.1 Архітектура інтернет речей

Інтернет речей (IoT) — це концепція, що передбачає підключення до Інтернету різноманітних фізичних об'єктів і пристроїв з метою обміну даними та взаємодії між ними. У звичайному розумінні це означає, що різні пристрої, які оточують нас у повсякденному житті, можуть бути підключені до мережі Інтернет, обмінюючись інформацією та виконуючи різноманітні функції без прямого участі людини.

Ідея полягає в тому, щоб забезпечити об'єктам і пристроям можливість "розуміти" і "комунікувати" між собою, що в свою чергу дозволяє створювати автоматизовані та інтелектуальні системи. Прикладами таких пристроїв можуть бути датчики, побутові прилади, автомобілі, вбрання, медичні пристрої тощо.

IoT використовується в різних галузях, таких як побутові технології (розумний будинок), промисловість (розумне виробництво), охорона здоров'я (медичні пристрої), транспорт (розумний транспорт), сільське господарство (precision farming) та інші. Однією з ключових характеристик IoT є можливість збору, обробки та аналізу великої кількості даних для отримання корисної інформації та оптимізації різних аспектів життєдіяльності.[25]

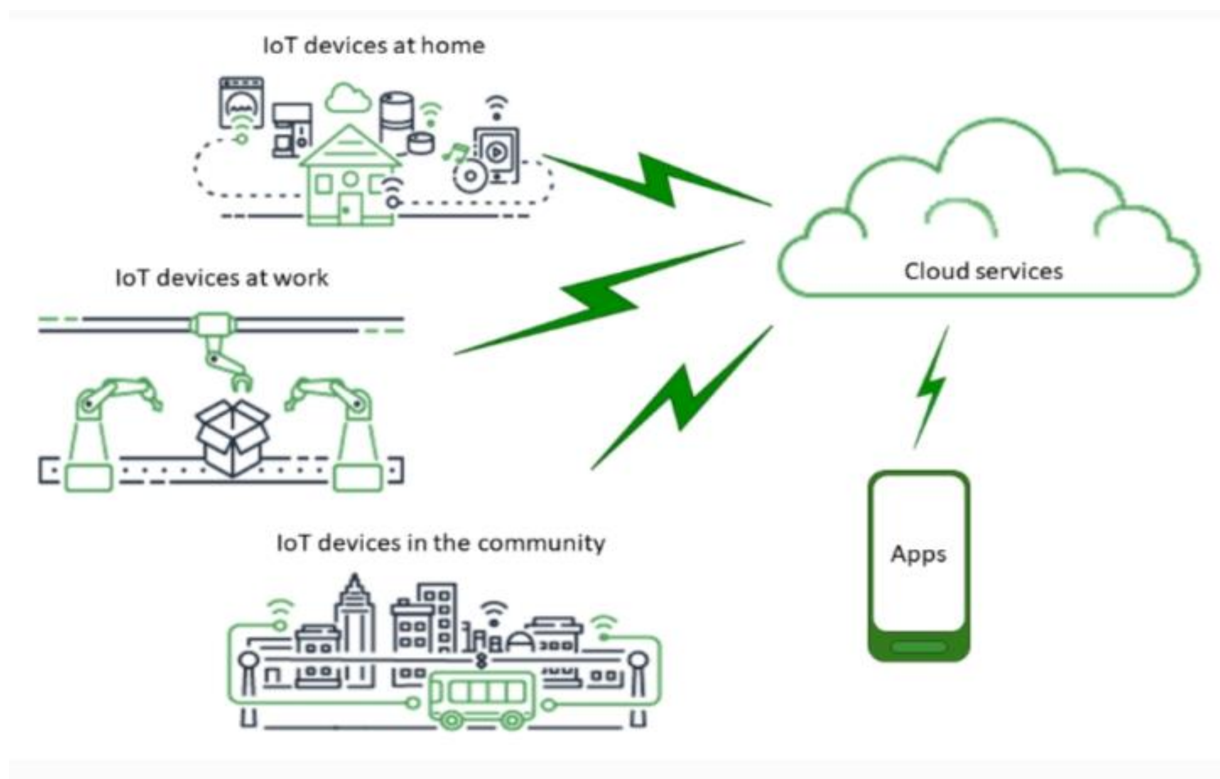

Рис. 1.10 Використання IoT

Спочатку, архітектура IoT базується на мережі фізичних сенсорів, які вимірюють різні параметри навколишнього середовища. Вони генерують дані, які стають основою для подальшого аналізу та взаємодії з іншими складовими системи. Актуатори відповідають за вплив на навколишнє середовище на основі отриманих даних.

Отримані дані передаються через мережу передачі даних, яка може включати в себе різноманітні технології, такі як бездротові мережі (Wi-Fi, Bluetooth, Zigbee), мережі передачі даних (LTE, 5G) та мережі короткого діапазону (NFC).

Отримані дані з сенсорів надходять до хмарних сервісів та обчислювальних центрів для подальшого аналізу, обробки та зберігання. Ці центри володіють великою обчислювальною потужністю та можливостями аналізу великих обсягів даних.

Для винесення розуміння та виведення корисних висновків з отриманих даних використовуються аналітичні та інтелектуальні алгоритми. Штучний інтелект (ШІ) грає ключову роль у виявленні закономірностей, прогнозуванні та прийнятті рішень.

На основі аналізу та висновків система може генерувати команди для актуаторів, які здійснюють вплив на фізичний світ. Цей зворотній зв'язок дозволяє системі реагувати на зміни та оптимізувати свою діяльність.

У зв'язку з обробкою та передачею важливих даних, архітектура IoT включає ефективні засоби захисту даних, шифрування та механізми автентифікації для забезпечення конфіденційності та цілісності інформації.

Важливою складовою архітектури є системи управління та моніторингу, які забезпечують централізовану контрольну точку для керування та нагляду за всією системою IoT.

Така комплексна архітектура Інтернету речей створює умови для розвитку інтелектуальних та ефективних рішень в різних галузях, від побутових систем до промислових та наукових застосувань.

Система Інтернету речей також передбачає можливість взаємодії з користувачем. Це включає в себе створення інтерфейсів для моніторингу та керування пристроями, додатками та послугами, що використовують дані, зібрані системою.

Архітектура IoT підтримує використання стандартів та протоколів, що дозволяє забезпечити взаємодію між пристроями різних виробників. Це сприяє інтеграції та розширенню екосистеми Інтернету речей.

Оскільки багато пристроїв IoT можуть працювати в умовах обмежених ресурсів енергопостачання, архітектура передбачає енергоефективні рішення, такі як оптимізація передачі даних, сплячий режим, та інші методи для забезпечення тривалого функціонування.[24]

Архітектура повинна бути гнучкою та легко розширюватися, дозволяючи додавати нові пристрої, сервіси та функції без необхідності значних змін у загальній структурі системи.

Забезпечення можливості взаємодії різних компонентів системи вимагає високого рівня інтероперабельності. Системи Інтернету речей мають працювати разом, незалежно від їхнього виробника чи функціонального призначення.

Архітектура включає елементи для управління взаємодією та функціонуванням усіх складових системи. Це важливо для забезпечення цілісності та ефективності дії системи Інтернету речей.

З урахуванням великої кількості зібраних даних, архітектура повинна включати механізми для забезпечення етичного використання інформації та захисту особистих прав та конфіденційності користувачів.

Аспекти тестування та валідації мають велике значення в архітектурі Інтернету речей, оскільки дозволяють перевірити працездатність, безпеку та відповідність стандартам всіх компонентів системи.

Ця архітектура IoT враховує різноманітні вимоги та виклики, з якими стикається Інтернет речей у сучасному світі, та намагається створити базовий фреймворк для створення ефективних та інтегрованих рішень в цій області.[26]

Архітектура Інтернету речей – це комплексне та системне підход до створення інфраструктури, що дозволяє забезпечувати зв'язок та взаємодію між мільярдами підключених пристроїв. Це вимагає розгляду всіх аспектів від збору даних до їхнього аналізу та використання. Ця архітектура створює умови для створення інтелектуальних та інноваційних рішень, які можуть впливати на різні сфери життя від приватного сектору до промисловості та міського управління.

1.4.2 Модулі інтернета речей

Модулі IoT є ключовими будівельними блоками, які роблять цей обмін даними можливим.

Так сенсорні модулі відіграють критичну роль у виявленні фізичних параметрів навколишнього середовища. Вони включають в себе різноманітні сенсори, такі як температурні, вологості, світлові, акустичні тощо, що дозволяють отримувати точні та розгорнуті дані про навколишній світ.

Вузлові модулі є центральною частиною IoP, яка відповідає за збір, обробку та передачу даних від сенсорних модулів до центральних серверів чи хмарних платформ. Вони мають вбудовані мікроконтролери, бездротові модулі передачі даних та можливості взаємодії з іншими вузловими модулями.

Комунікаційні модулі забезпечують передачу даних між різними складовими IoP. Вони можуть використовувати різноманітні технології зв'язку, такі як Wi-Fi, Bluetooth, Zigbee чи LoRa, для забезпечення ефективного обміну даними між вузловими модулями та інфраструктурою Інтернету.

Центральні серверні модулі відповідають за централізоване зберігання, обробку та аналіз даних, отриманих від вузлових модулів. Вони використовують великі обчислювальні потужності та бази даних для створення значущих інсайтів і підтримки прийняття рішень.

Модулі безпеки є важливою складовою систем IoT. Вони включають в себе механізми шифрування, аутентифікації та захисту від несанкціонного доступу.

Аспект енергозабезпечення є критичним для ефективності модулів IoT. Багато модулів працюють на основі батарей або акумуляторів, тому ефективне використання енергії та можливість періодичного поновлення є ключовими аспектами.

Модулі IoT визначають майбутнє розвитку технологій, прискорюючи обмін інформацією та надаючи інструменти для реалізації концепції "розумних" систем. Їх інтеграція в різноманітні галузі, починаючи від побутових пристроїв і

закінчуючи промисловими комплексами, дозволяє оптимізувати процеси та створювати більш стійкі та зручні технологічні рішення.[26]

1.4.3 IoT протоколи передачі данних

В інтернеті речей протоколи передачі даних є ключовою складовою для забезпечення ефективного обміну інформацією між пристроями та системами. Ці протоколи визначають стандарти взаємодії та формати даних, що передаються у мережі. Розглянемо деякі з основних IoT протоколів у деталях:

MQTT (Message Queuing Telemetry Transport) є легковаговим протоколом повідомлень, розробленим для передачі даних у мережах з обмеженим ресурсами. Він використовує модель "видавець-підписник", де пристрої можуть підписуватися на отримання повідомлень на конкретні теми. Цей протокол ефективно використовується в IoT для забезпечення асинхронного обміну даними.

CoAP (Constrained Application Protocol). Розроблений для пристроїв з обмеженими ресурсами, CoAP є протоколом, який дозволяє пристроям здійснювати обмін ресурсами та керувати ними. Він працює поверх протоколу UDP, що робить його особливо підходящим для мереж з великою кількістю мобільних та енергоефективних пристроїв.

HTTP/HTTPS (Hypertext Transfer Protocol/Secure). Хоча HTTP є стандартним протоколом для передачі даних в інтернеті, його застосування в IoT зазвичай включає в себе використання HTTPS для забезпечення безпеки. Протокол HTTP забезпечує взаємодію "запит-відповідь" та є широко використовуваним у веб-заснованих застосунках IoT.

DDS (Data Distribution Service) є протоколом для реального часу, який спрямований на обмін даними між пристроями у розподіленому середовищі. Він забезпечує механізми для публікації-підписки та розподіленої обробки подій, що робить його ефективним у вимогливих до часу задачах.

AMQP (Advanced Message Queuing Protocol) є протоколом для передачі повідомлень між застосунками. Він забезпечує стандартизований спосіб обміну

повідомленнями між пристроями та визначає структуру даних та механізми доставки.[27]

Ці протоколи грають важливу роль у створенні стандартів та забезпеченні сумісності в розмаїтті пристроїв, що складають Інтернет речей.

2 АНАЛІЗ ВИКОРИСТАННЯ БАЗ ДАНИХ ЧАСОВИХ РЯДІВ

2.1 Робота з часовими рядами в базах даних

2.1.1 Особливості часових рядів та їхнє використання

Часові ряди та їх аналіз стають дуже популярними у світі обробки даних через їх швидке розповсюдження. Це зумовлено різними чинниками, такими як розвиток Інтернету речей, перехід до електронного документообігу та вдосконалення технологій у розумних містах. Протягом найближчих років очікується подальший зріст попиту на технології обробки даних у формі часових рядів.

Зі збільшенням потреби у постійному моніторингу та зборі даних зростає і вимога до інструментів аналізу часових рядів, які використовують як статистичні, так і машинні методи навчання. Фактично, найбільш перспективні моделі базуються на обох цих технологіях.

Аналіз часових рядів включає у себе отримання важливої узагальненої та статистичної інформації з точок даних, які відомі у хронологічному порядку. Ця операція дозволяє не лише вивчати минулі стани, але й передбачати положення майбутніх точок даних.

Використання часових рядів є важливим інструментом в різних галузях, де вивчається динаміка подій у часі. Цей підхід знаходить широке застосування в наукових дослідженнях, аналізі даних та прогнозуванні у різноманітних сферах.

Аналіз часових рядів дозволяє вивчати зміни в економіці та фінансових ринках. Прогнозування та виявлення патернів у фінансових даних є важливим для прийняття рішень у фінансовому секторі.[28]

В контексті Інтернету речей, часові ряди грають важливу роль у зборі та аналізі даних в реальному часі. Моніторинг та аналіз сигналів з IoT-пристроїв дозволяє виявляти тенденції, здійснювати передбачення та підтримувати оптимальне функціонування мережі підключених пристроїв.

Технічне обслуговування та Промисловість 4.0. В контексті виробничої сфери використання часових рядів дозволяє відстежувати ефективність обладнання, планувати технічне обслуговування та уникати непередбачених збоїв у виробництві.

Використання часових рядів у медичних дослідженнях дозволяє відстежувати зміни в показниках здоров'я пацієнтів та розвивати методи прогнозування захворювань.

Моніторинг та аналіз часових рядів в енергетичному секторі допомагає вирішувати питання ефективного використання ресурсів та планування енергетичних систем.

Використання часових рядів у транспортних системах сприяє в оптимізації маршрутів, прогнозуванні трафіку та покращенні ефективності транспортних засобів.

Вивчення часових рядів у кліматології допомагає розуміти зміни в кліматі, прогнозувати погодні умови та впливати на стратегії адаптації.

У вивченні соціальних явищ інформація, представлена у вигляді часових рядів, може бути використана для аналізу змін у споживацьких та поведінкових тенденціях. Це допомагає розуміти соціокультурні процеси та формувати стратегії соціального розвитку.

Слід зазначити, що дані, які використовуються для створення часових рядів, можуть містити ряд дефектів та неперфектностей, які впливають на якість та достовірність аналізу. Декілька факторів може призвести до дефектів у часових рядах:

Відсутність даних (пропуски). Деякі точки даних можуть бути відсутніми через технічні проблеми, втрату інформації або інші причини. Пропуски можуть ускладнити аналіз та спотворити результати.

Аномальні значення (викиди). Надмірно великі чи низькі значення деяких точок даних можуть впливати на точність моделей та прогнозів, зроблених на основі часового ряду.

Шум у даних. Деякий рівень випадкового шуму може бути присутнім у часових рядах через різноманітні фактори, такі як помилки вимірювань або випадкові впливи.

Зміни в структурі даних. Іноді може відбутися зміна у властивостях або характері даних, що впливає на їхню структуру та розподіл.

Систематичні помилки. Помилки апаратного забезпечення, неправильне збирання даних або інші систематичні фактори також можуть призвести до дефектів у часових рядах.

Важливо враховувати ці можливі дефекти та вживати відповідні заходи для їхнього виявлення та корекції під час аналізу часових рядів. Застосування методів обробки даних та виправлення помилок може допомогти покращити якість та достовірність аналізу. Для цього використовують наступні методи:

- заміна попереднім значенням
- рухоме середнє
- інтерполяція

Заміна попереднім значенням - це метод обробки даних, при якому відсутні чи пошкоджені значення в часовому ряді замінюються значенням, що випереджає їх у послідовності. Такий підхід використовує попередні відомі дані для заповнення пропусків або відновлення пошкоджених значень, зберігаючи при цьому структуру та порядок часового ряду.

Рухоме середнє - це метод аналізу часових рядів, що використовується для згладжування випадкових змін та виокремлення трендів чи патернів. Цей метод включає в себе обчислення середнього значення для певної кількості послідовних точок даних. Рухоме середнє розраховується на кожному кроці за допомогою певного вікна (або періоду), переміщаючи це вікно вздовж ряду. Це дозволяє створювати плавніше значення, яке відображає середню тенденцію даних, а не їхній випадковий коливання. Рухоме середнє допомагає виокремлювати загальні зміни та робить ряд більш зрозумілим для аналізу.

Інтерполяція - це метод обробки даних, який використовується для визначення проміжних значень в часовому ряді на основі відомих значень в інших

точках. Цей підхід дозволяє заповнювати пропуски в даних або відновлювати пошкоджені значення, використовуючи математичні методи апроксимації. Інтерполяція може бути корисною для підтримання структури часового ряду та отримання повнішої та безперервної інформації в областях, де виникають втрати даних.

Загальний вплив використання часових рядів у різних галузях засвідчує їхню універсальність та значущість. Завдяки статистичному аналізу, моделям прогнозування та застосуванню сучасних методів машинного навчання, вони стають ефективним інструментом для вирішення складних завдань та прийняття обґрунтованих рішень у різних областях.[29]

2.1.2 Аналіз можливостей баз даних для роботи з часовими рядами

Аналіз можливостей баз даних для роботи з часовими рядами включає оцінку їхніх функціональностей та здатності ефективно опрацьовувати та зберігати дані з часовою компонентою.

Декілька ключових аспектів для такого аналізу включають: підтримка часових типів даних, індексація часових стовпців, підтримка агрегаційних функцій, оптимізації для операцій з часовими рядами, підтримка операцій з часовими вікнами, масштабованість, підтримка мов запитів.

Найпоширенішими примітивними типами даних для часових рядів є дата та час. Такі типи, як TIMESTAMP чи DATETIME, дозволяють точно вказувати момент часу. Для зберігання інтервалів часу можуть використовуватися типи INTERVAL. Крім того, TIMEZONE додає можливість робити дані орієнтованими на часові зони.

Для точного представлення деталей у часових рядах, важливо використовувати типи даних, які можуть охоплювати мілісекунди, мікросекунди чи ще більш дрібні часові відрізки. Типи, як TIMESTAMP WITH TIMEZONE, забезпечують додаткову гнучкість та точність у відображенні часових значень.

У великих обсягах часових даних ефективне управління ресурсами є ключовим завданням. Типи даних, як TIMESTAMP або BIGINT (для епохального часу), дозволяють ефективно працювати з великими обсягами даних, забезпечуючи оптимальну продуктивність.

Сучасні системи управління базами даних надають додаткові характеристики для роботи з часовими рядами. Наприклад, функції агрегації, які працюють з часовими інтервалами, або можливість використання індексів для прискорення операцій з великими обсягами часових даних.

Використання відповідних типів даних у базах часових рядів є важливим для аналізу та прогнозування. Точність та гнучкість типів даних визначають можливість отримання точних та надійних результатів.[30]

Операції з часовими рядами вимагають специфічних підходів та оптимізацій для ефективного використання ресурсів та отримання точних результатів. Цей розділ розглядає ключові аспекти оптимізацій при роботі з часовими рядами у контексті баз даних.

Однією з ключових стратегій є використання індексів та сортування даних за часовою відміткою. Індекси дозволяють ефективно знаходити та фільтрувати дані за часом, що прискорює операції вибірки. Сортування даних за часом сприяє поліпшенню швидкодії операцій.

Кешування активно використовується для збереження попередньо обчислених результатів операцій над часовими рядами. Це особливо ефективно для операцій складного аналізу, таких як статистичні розрахунки чи складні моделі прогнозування. Попереднє завантаження дозволяє заздалегідь завантажувати дані, що може покращити час відгуку при обробці запитань користувача.

Використання паралельної обробки та розподіленої архітектури може розгрузити систему при операціях з великими обсягами часових даних. Розподілені обчислення дозволяють використовувати різні вузли для обробки частин даних, що призводить до прискорення операцій над часовими рядами.

Ефективність операцій значно залежить від правильно складених запитань. Використання індексів, обрання оптимальних алгоритмів та використання умов фільтрації можуть суттєво покращити продуктивність операцій з часовими рядами.

Зменшення розміру даних за допомогою технік компресії може поліпшити якість зберігання та зменшити час передачі даних. Компресія може бути особливо корисною при роботі з великими обсягами часових рядів, де ефективне використання простору є важливим аспектом.

Індексація часових стовпців в базах даних є стратегічно важливим елементом для покращення продуктивності та ефективності операцій з часовими рядами. Цей аспект дозволяє швидше здійснювати пошук, фільтрацію та агрегацію даних, що є ключовим у великих обсягах часових даних.

Існують різні типи індексів, придатних для часових стовпців. Перш за все, кластерні індекси, які групують дані в заздалегідь відсортовані блоки, сприяють ефективній вибірці за часом. Також використовуються бітові карти, що дозволяють швидко виділяти записи, які відповідають конкретним діапазонам часу.

Індексація часових стовпців сприяє ефективному пошуку в межах конкретних інтервалів часу. Застосування індексів у поєднанні з оптимізованими алгоритмами пошуку дозволяє прискорити операції, такі як вибірка та фільтрація за часовими параметрами.

Індексація допомагає ефективно виконувати агрегації та операції з великими обсягами часових даних. Здійснення обчислень за допомогою індексів дозволяє швидко отримувати результати складних аналітичних запитань, що забезпечує швидку відповідь системи.

Для аналітичних операцій та прогнозування часових рядів важливо мати ефективні індекси. Вони забезпечують швидку обробку складних аналітичних запитань та сприяють точній роботі з часовими даними для прогнозування майбутніх подій.

З індексами легше впоратися з великим обсягом часових даних. Вони оптимізують роботу із запитами та дозволяють системі більш ефективно управляти великими наборами даних, що важливо у вимірах, де часові ряди мають велику роздільну

Підтримка агрегаційних функцій у базах даних визначається їхнім здатністю обчислювати суми, середні значення, мінімальні та максимальні значення, що є ключовим для аналізу часових рядів. Цей аспект грає важливу роль у виявленні та розумінні патернів в часових даних.

Агрегаційні функції, такі як SUM та AVG, дозволяють обчислювати суму та середнє значення часових рядів. Це важливо для виявлення загальних тенденцій та коливань великих обсягів часових даних.

Функції MIN та MAX дозволяють ідентифікувати найменші та найбільші значення в часових рядах. Це корисно для виявлення екстремальних точок та аналізу аномалій в часових даних.

У великих обсягах часових даних ефективні агрегаційні функції допомагають здійснювати оперативний аналіз та отримання значень на різних рівнях деталізації. Це забезпечує швидке та точне отримання результатів.

Агрегаційні функції можна успішно використовувати для групування даних за часовими інтервалами. Це корисно при аналізі та представленні часових рядів на більш високому рівні деталізації.

Для аналізу трендів та циклів в часових рядах важливо використовувати функції підсумовування. Це дозволяє виявити довгострокові зміни та розвиток подій в часі.

Однією з ключових операцій з часовими вікнами є рухомі середні, які дозволяють згладжувати коливання та виділяти тренди в часових рядах. Експоненційне згладжування додає вагу більш актуальним даним, полегшуючи виявлення швидких змін.

Підтримка операцій з часовими вікнами дозволяє проводити агрегацію даних в заданих часових інтервалах. Це корисно для отримання статистики та сумарної інформації за конкретні періоди.

Операції з часовими вікнами підтримують функції порівняння та фільтрації, що дозволяє визначати, які дані входять в задані часові рамки. Це особливо важливо для виявлення подій та аномалій.

Великі обсяги часових даних вимагають ефективної роботи з операціями часових вікон. Вони дозволяють здійснювати обчислення та агрегації на великій кількості даних, що важливо для великих корпоративних систем та аналітичних платформ.

Операції з часовими вікнами важливі для прогнозування та аналізу трендів у часових рядах. Вони дозволяють створювати моделі та визначати динаміку змін в часі для більш точного аналізу та прогнозу.[31]

2.1.3 Технології зберігання та опрацювання часових рядів в базах даних

Серед різноманітних варіантів зберігання часових рядів існують вагомі причини відмовитися від використання окремих файлів на користь баз даних, особливо при роботі з новими наборами даних. У базах даних, особливо в нереляційних, дані перебувають у гнучкому стані. Наявність жорсткої структурної організації є ще однією перевагою баз даних: це дозволяє значно прискорити запуск проектів із часовими рядами порівняно з рішеннями, що базуються на окремих файлах. Навіть якщо ви віддаєте перевагу зберіганню даних часових рядів у файлах (на відміну від більшості дослідників), база даних допоможе вам визначитися із структурою файлів, кількість яких неодмінно збільшується з ускладненням проекту та поповненням часових рядів новими даними.

При роботі з часовими рядами можуть використовуватися як реляційні, так і нереляційні технології (SQL та noSQL) управління базами даних. Однак обидва випадки пов'язані з рядом труднощів при управлінні базами даних часових рядів.

Початково SQL-рішення були призначені для обробки транзакційних даних, що були достатніми для повного опису дискретної події. У транзакціях обробляються дані атрибутів, які відображені в багатьох первинних ключах, таких як призначення, ім'я клієнта, дата виконання та вартість транзакції. Існують дві

важливі особливості транзакційних даних, які відрізняють їх від даних часових рядів:

- Точки даних часто оновлюються.

- Доступ до даних здійснюється випадковим чином, оскільки порядок їх обробки ніяк не регламентується.

Якщо дані часового ряду описують історію певного процесу, то транзакція вказує лише кінцевий стан системи. Отже, дані часових рядів зазвичай не вимагають оновлення, що робить операції запису даних із випадковим доступом дуже малоймовірними. Таким чином, вимоги до продуктивності інструментів, які були ключовими протягом багатьох десятиліть розробки СУБД, виявляються незначущими при роботі з часовими рядами. Фактично, висуваючи вимоги до систем зберігання часових рядів, ми переслідуємо кілька інших цілей, оскільки управління такими сховищами базується на принципах, відмінних від прийнятих в реляційних базах даних.

Ключові Вимоги до Засобів Зберігання Часових Рядів:

- Операції запису переважають над операціями читання.

- Дані записуються, читаються і оновлюються не випадковим чином, а в хронологічному порядку.

- Одночасне читання даних виконується набагато частіше, ніж в разі проведення транзакцій.

- Крім часу, допускається використовувати (якщо вони визначені) лише кілька первинних ключів.

- Масове видалення даних виконується частіше, ніж видалення окремих точок даних.

Більшість цих функцій підтримується NoSQL-рішеннями — багато нереляційних баз даних загального призначення пропонують багато того, що потрібно реалізувати в сховищі часових рядів — перш за все, що стосується переважання операцій запису над операціями читання.

Концептуально технології NoSQL гармонійно взаємодіють з вимогами, які ставляться до засобів зберігання часових рядів, відображаючи заздалегідь багато

спеціальних можливостей, наприклад, заповнення полів даних не для всіх точок даних. Гнучкість структури NoSQL виявляється природною для часових рядів. І, правду кажучи, стрімкий інтерес до нереляційних технологій в значній мірі викликаний потребою в системах управління та зберігання часових рядів.

З тієї ж причини готові NoSQL-рішення показують кращі результати в операціях запису, ніж реляційні СУБД.

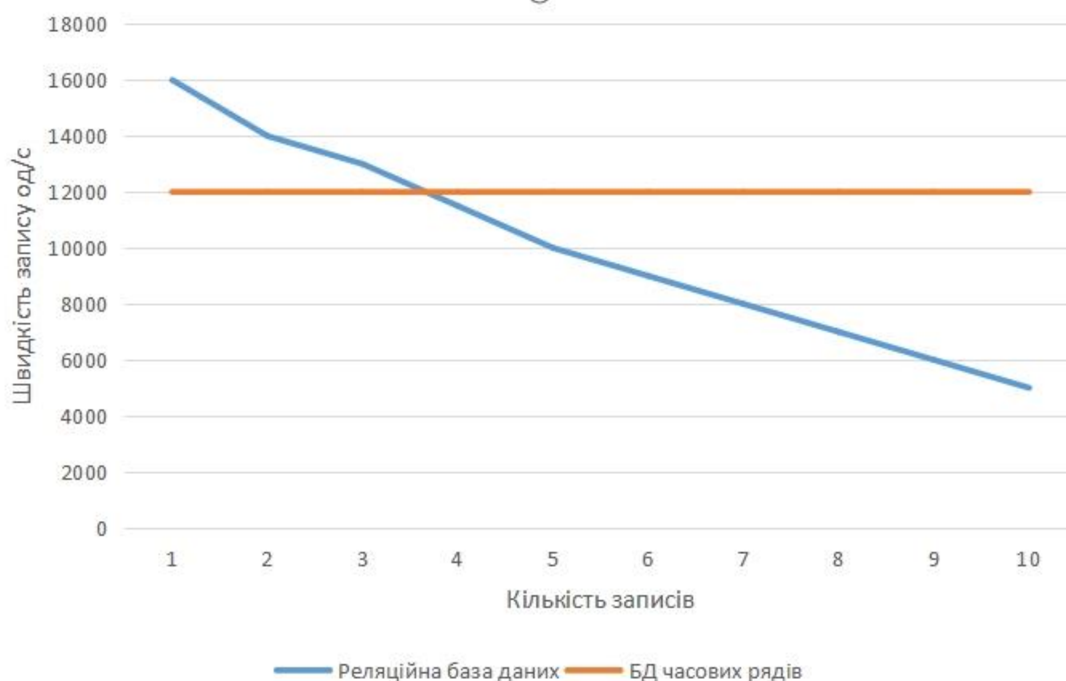

Рис. 2.1 Порівняння швидкості запису реляційною БД та БД часових рядів

Нереляційні бази даних мають більшу швідксть запису, також вони ідеально підходять для розробки функціональних та високопродуктивних рішень у системах, де мало що відомо про майбутні дані.[32]

2.1.4 Вибір баз даних для зберігання часових рядів

На даний час створено достатньо велику кількість баз даних часових рядів. Майже всі знаходяться в постійному вдосконаленні та розвитку.

Основні бази даних часових рядів:

InfluxDB. Тип даних: Часовий ряд, багатомодальний.

Prometheus. Тип даних: Часовий ряд.

Graphite. Тип даних: Часовий ряд.

VictoriaMetrics.Тип даних: Часовий ряд.

Kdb. Багатомодальний.

TimescaleDB. Тип даних: Часовий ряд, багатомодальний.

DolphinDB. Тип даних: Часовий ряд, багатомодальний.

RRDtool. Тип даних: Часовий ряд.

Apache Druid. Багатомодальний.

TDengine. Тип даних: Часовий ряд, багатомодальний.

QuestDB. Тип даних: Часовий ряд, багатомодальний.

OpenTSDB. Тип даних: Часовий ряд.

GridDB. Тип даних: Часовий ряд, багатомодальний.

Fauna. Багатомодальний.

Amazon Timestream. Тип даних: Часовий ряд.

M3DB. Тип даних: Часовий ряд.

Heroic. Тип даних: Часовий ряд.

eXtremeDB. Багатомодальний.

CrateDB. Багатомодальний.

Apache IoTDB. Тип даних: Часовий ряд.

KairosDB. Тип даних: Часовий ряд.

ITTIA. Тип даних: Часовий ряд, багатомодальний.

Raima Database Manager. Багатомодальний.

Axibase. Тип даних: Часовий ряд.

Riak TS. Тип даних: Часовий ряд.

Machbase. Тип даних: Часовий ряд.

CnosDB. Тип даних: Часовий ряд.

ArcadeDB. Багатомодальний.

Bangdb. Багатомодальний.

IRONdb. Тип даних: Часовий ряд.

Quasardb. Тип даних: Часовий ряд.

SiteWhere. Тип даних: Часовий ряд.

NSDb. Тип даних: Часовий ряд.

Alibaba Cloud TSDB. Тип даних: Часовий ряд.

IBM Db2 Event Store. Багатомодальний.

Tigris. Багатомодальний.

GreptimeDB. Тип даних: Часовий ряд.

Hawkular Metrics. Тип даних: Часовий ряд.

SiriDB. Тип даних: Часовий ряд.

Blueflood. Тип даних: Часовий ряд.

Warp 10. Тип даних: Часовий ряд.

Yanza. Тип даних: Часовий ряд.

Newts. Тип даних: Часовий ряд.

При розробці системі тестування баз даних часових рядів було використано VictoriaMetrics.

VictoriaMetrics - це потужна база даних часових рядів, призначена для високопродуктивного моніторингу та аналітики. В багатьох відношеннях вона схожа на Prometheus, проте пропонує ряд переваг, включаючи кращу продуктивність, масштабованість та стиснення даних. VictoriaMetrics також має відкритий вихідний код і є безкоштовною для використання.

VictoriaMetrics може відстежувати широкий спектр систем, включаючи кластери Kubernetes, сервери, додатки, компоненти інфраструктури та навіть пристрої Інтернету речей.

VictoriaMetrics - це система моніторингу, повністю сумісна з Prometheus. Вона пропонує багато з тих самих функцій, що й Prometheus, але з деякими додатковими перевагами, такими як вища продуктивність та масштабованість.

Однією з ключових відмінностей між VictoriaMetrics та Prometheus є те, що VictoriaMetrics включає функцію виявлення аномалій. Цю функцію можна використовувати для виявлення незвичайних змін у даних метрик, що може бути корисно для виправлення несправностей та виявлення потенційних проблем.

VictoriaMetrics має наступні важливі особливості:

- VictoriaMetrics можна використовувати як довгострокове сховище для Prometheus.

- Її можна використовувати як заміну Prometheus в Grafana, оскільки він підтримує API запитів Prometheus.

- VictoriaMetrics можна використовувати як заміну Graphite в Grafana, оскільки він підтримує Graphite API. Він також дозволяє зменшити витрати на інфраструктуру більше ніж в 10 разів порівняно з Graphite.

- Легко налаштовується та використовується. VictoriaMetrics складається з одного невеликого виконуваного файлу без зовнішніх залежностей. Вся конфігурація виконується за допомогою явних прапорців командного рядка з розумними значеннями за замовчуванням. Всі дані зберігаються в одному каталозі, вказаному прапором -storageDataPath командного рядка.

- Використовується аналогічна мову запитів PromQL, відому як MetricsQL, яка надає поліпшену функціональність поверх PromQL.

- Забезпечує глобальний запит, де кілька екземплярів Prometheus або будь-яких інших джерел даних можуть надсилати дані до VictoriaMetrics, і пізніше ці дані можна буде запитати за допомогою одного запиту.

- Забезпечує високу продуктивність та добру вертикальну і горизонтальну масштабованість як для прийому даних, так і для їх запиту. Він перевершує InfluxDB та TimescaleDB практично в 20 разів.

- Ефективне використання оперативної пам'яті. Victoriametriks використовує в 10 разів менше оперативної пам'яті, ніж InfluxDB, і в 7 разів менше оперативної пам'яті, ніж Prometheus, Thanos або Cortex при роботі з мільйонами унікальних часових рядів.

- Високий ступінь стиснення даних. Згідно з тестами, в обмеженому сховищі можна зберігати до 70 разів більше точок даних порівняно з TimescaleDB, а для зберігання потрібно в 7 разів менше місця порівняно з Prometheus, Thanos або Cortex.

- Оптимізована для високих затримок вводу-виводу та низької кількості операцій вводу-виводу в секунду

- Вона захищає сховище від пошкодження даних при некоректному завершенні роботи (наприклад, OOM, апаратному скиданні тощо).

- VictoriaMetrics підтримує різноманітні протоколи для збору, прийому заповнення метрик, такі як Prometheus Exposition, OpenTSDB, Graphite, JSON, DataDog, OpenTelemetry та інші.

- Вона підтримує потужну агрегацію потоків, яку можна використовувати як альтернативу statsd.

- Підтримується тегування метрик для більш ефективного організації даних.

- Забезпечує рішення проблем з високою мощністю та високою частотою відтоку за допомогою обмежувача серій.

- Вона працює ідеально з великими обсягами даних часових рядів з APM, Kubernetes, IoT-датчиків, підключених автомобілів, промислової телеметрії, фінансових даних та різноманітних корпоративних робочих навантажень.

- Є версія кластера з відкритим вихідним кодом для високої доступності та масштабованості.

- Може зберігати дані в сховищах на основі NFS, таких як Amazon EFS та Google Filestore.

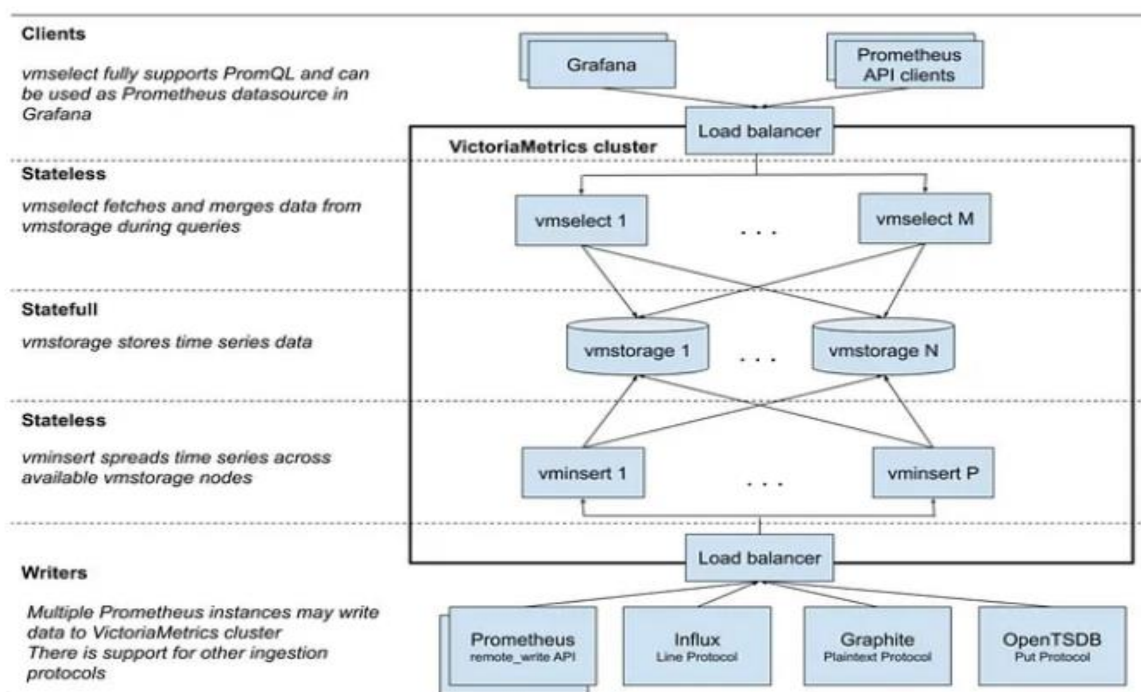

Рис. 2.2 Схема роботи Victoriametriks

VictoriaMetrics існує у двох версіях: одноузловій та кластерній. Переваги кластерної версії у швидкості запису, масштабованості та надійності відмітні.

Архітектура кластерної версії VictoriaMetrics включає:

- VM storage – зберігає дані.
- VM insert – відповідає за запис даних в сховище.
- VM select – використовується для виконання запитів на читання з сховища.

Ці особливості роблять VictoriaMetrics потужним інструментом для роботи з даними часових рядів у високонавантажених та складних системах моніторингу та аналітики, який має ряд переваг над іншими базами даних часових рядів та який може буьти доповненням інших баз даних.[33]

2.1.5 Оптимізація та індексація для ефективного пошуку

Індексація в контексті баз даних відноситься до процесу оптимізації операцій пошуку даних шляхом створення та підтримки структури даних, яка встановлює відповідність різних елементів даних їхнім фізичним розташуванням в базі даних. Основною метою індексації є значуще скорочення часу та обчислювальних ресурсів, необхідних для запитів та доступу до даних, тим самим підвищуючи загальну ефективність та продуктивність систем баз даних. Ефективні стратегії індексації вирішально важливі для великошкальних застосунків, оскільки вони можуть обробляти величезні об'єми даних та вимагають швидкого пошуку та обробки.

В основі індексації лежить концепція структур даних, таких як B-дерева, хеш-індекси та бітові індекси, які полегшують організацію та управління індексами бази даних. Наприклад, індекси B-дерева забезпечують швидкий доступ до даних як у порядку зростання, так і у порядку спадання, одночасно збалансовуючи операції вставки, видалення та пошуку. З іншого боку, індекси на основі хешу особливо корисні для пошуку за рівністю і можуть ефективно використовуватися для кешування часто використовуваних даних. Растрові індекси, як правило, використовуються для стовпців з низькою кардинальністю,

де кількість різних значень відносно невелика порівняно з загальною кількістю записів. Вибір відповідного механізму індексації в кінцевому підсумку залежить від характеру, розміру та шаблонів доступу до базових даних, а також від конкретних вимог додатка до запитів та обробки.

Крім цих структур загального використання для обслуговування конкретних областей застосунків також застосовуються спеціалізовані методи індексації, такі як повнотекстова індексація, просторова індексація та індексація часових рядів. Наприклад, повнотекстова індексація призначена для оптимізації текстового пошуку, що дозволяє ефективно обробляти складні запити, що включають текстові шаблони, ранжування, близькість та інше. Просторова індексація призначена для додатків, які працюють з географічними, геометричними або багатовимірними даними, що дозволяє швидко запитувати та вилучати об'єкти в межах певного діапазону чи близькості. Як вказує саме назва, індексація часових рядів адаптована для даних з відмітками часу і широко використовується в фінансових, моніторингових та аналітичних застосунках.

Реалізація індексації в системі баз даних може мати серйозні наслідки для продуктивності застосунків, які використовують цю систему. Ефективна стратегія індексації може забезпечити швидке виконання запитів до великих обсягів даних, поліпшуючи час реакції всієї системи. Проте важливо відзначити, що підтримка та оновлення індексів також може призвести до накладних витрат з точки зору вимог до зберігання та обробки. Надмірне використання індексів або неоптимальні конфігурації можуть призвести до неефективності і навіть знизити продуктивність операцій по обробці даних, таких як вставка, оновлення та видалення.

Моніторинг та точна настройка стратегій індексації мають вирішальне значення, оскільки дані застосунки та шаблони запитів можуть змінюватися з часом. У таких випадках корисно використовувати інструменти, які відстежують та аналізують продуктивність запитів та використання індексів. Крім того, регулярний порівняльний аналіз, періодичне обслуговування та постійне удосконалення стратегії індексації можуть ще більше підвищити ефективність та

швидкодію застосунків, які використовують бази даних, дозволяючи компаніям повністю використовувати потенціал своїх активів даних.

Індексація — це фундаментальний аспект управління та оптимізації баз даних, що тісно пов'язаний з продуктивністю, ефективністю та масштабованістю будь-якого застосунку, який використовує систему баз даних.[34]

2.1.6 Взаємодія з графіками та візуалізація даних

Взаємодія з графіками та візуалізація даних баз даних часових рядів відіграють ключову роль у розумінні та аналізі великих обсягів часових даних. Графіки та візуалізації є ефективними інструментами для представлення та сприйняття змін у динаміці даних з плином часу.

Інтерфейси візуалізації даних можуть бути вбудовані безпосередньо в системи управління базами даних часових рядів або використовувати зовнішні інструменти, такі як Grafana, Kibana, Power BI, Tableau та інші. За допомогою графіків, діаграм та інших візуальних елементів можна ефективно відтворювати та аналізувати різноманітні характеристики часових рядів.

Grafana — це платформа з відкритим вихідним кодом для візуалізації, моніторингу та аналізу даних. Grafana дозволяє користувачам створювати інформаційні панелі (дашборди), кожна з яких відображає певні показники протягом встановленого періоду часу. Кожний дашборд універсальний, тому його можна налаштувати для конкретного проекту або врахувати будь-які потреби розробки та бізнесу.

Для взаємодії з графіками можна використовувати різні методи, такі як вибір конкретного проміжку часу, збільшення або зменшення масштабу, фільтрація за конкретними параметрами, додавання або вилучення певних рядів часових даних. Також можуть бути доступні інші функції візуалізації, наприклад, відображення трендів, визначення аномалій, відслідковування паттернів тощо.

Важливою є можливість взаємодії з графіками в реальному часі, особливо для систем моніторингу та аналітики, де швидке реагування на зміни може бути

критичним. Використання інтерактивних елементів, таких як зум, фільтри, анімація та інші, може значно полегшити взаємодію з даними та зробити її більш інтуїтивно зрозумілою.

Графіки та візуалізація даних баз даних часових рядів допомагають користувачам отримувати цінні інсайти з великих обсягів часових даних та ефективно приймати управлінські рішення на основі цих аналізів.[35]

3 РОЗРОБКА МОДЕЛІ ТА ТЕСТУВАННЯ МАКСИМАЛЬНОГО НАВАНТАЖЕННЯ БАЗИ ДАНИХ ЧАСОВИХ РЯДІВ

3.1 Планування та проектування моделі

3.1.1 Планування та аналіз вимог до системи розробки тестування бази даних часових рядів

Згідно SDLC початок розробки починається з планування. Планування є фундаментальним етапом в розробці програмного продукту, так як саме на цьому єтапі визначається мета та обсяг проекту, визначення ресурсів, команди проекту, витрати та ризики.

Так метою проекту є побудова системи тестування баз даних часових рядів.

Обсяг проекту повинен знаходитись в межах задач, а саме: створення системи тестування баз даних часових рядів, перевірка даної систему тестування бази данних та аналіз результатів тестування.

Наступним етапом є аналіз вимог до системи розробки тестування баз даних часових рядів. Дана система вимагає ретельного аналізу вимог для забезпечення ефективності, надійності та високої продуктивності. Основні аспекти аналізу вимог можна розглядати в контексті функціональних та нефункціональних вимог системи.

Функціональні вимоги:

1. Зберігання та обробка даних часових рядів:

- Система повинна забезпечувати можливість зберігання великих обсягів часових рядів.

- Ефективна обробка та агрегація даних часових рядів для забезпечення швидкого доступу та виконання аналізу.

2. Масштабованість та витривалість:

- Система повинна бути масштабованою, здатною обробляти збільшення обсягів даних з часом.

- Забезпечення високої доступності та витривалості для уникнення втрати даних.

3. Запити та аналітика:

- Підтримка складних запитів та аналітичних операцій для забезпечення гнучкості аналізу даних.

- Здатність виконання агрегованих операцій та обчислень для отримання цінних висновків.

4. Інтеграція з іншими інструментами:

- Можливість інтеграції з різноманітними інструментами візуалізації, аналізу та моніторингу даних.

Нефункціональні вимоги:

1. Швидкодія та відмовостійкість:

- Забезпечення швидкої реакції на запити та операції.

- Механізми відмовостійкості та відновлення для забезпечення стабільної роботи системи.

2. Безпека та захист даних:

- Забезпечення механізмів аутентифікації та авторизації для контролю доступу до даних.

- Шифрування даних в транспорті та зберіганні для захисту конфіденційності.

3. Масштабованість та гнучкість:

- Можливість динамічного масштабування ресурсів в залежності від навантаження.

- Гнучкість у використанні різних типів баз даних та систем зберігання даних.

4. Моніторинг та логування:

- Забезпечення системи моніторингу для слідкування за продуктивністю та виявленням аномалій.

- Логування подій для належного аналізу та вирішення проблем.

Аналіз вимог до системи розробки та тестування баз даних часових рядів важливий для створення високопродуктивної, стійкої та функціональної системи, яка задовольняє потреби користувачів у галузі обробки та аналізу часових рядів. Від якості первинного аналізу залежить щвідкість розробки, наявність баг в документах та вартість продукту, бо саме баги на самих перших етапах є найдорожчими та найскладнішими для виправлення.[5]

3.1.2 Дизайн системи тестування бази даних часових рядів

В дизайні повинна бути зазначено архітектура системи розробки, стратегія та планування тестування. Основним видом тестування бази даних часвоих рядів, для нашої системи, будет функціональне тестування, тобто безпосередня перевірка можливостей системи здійснити тестування навантаження бази даних часових рядів та навантажувальне тестування в різних варіаціях, як вид нефункціонального тестування.

Для здійснення навантажувального тестування потрібно реалізувати навантаження на базу даних з декількох серверів, які будуть приймати запити з MQTT брокера.

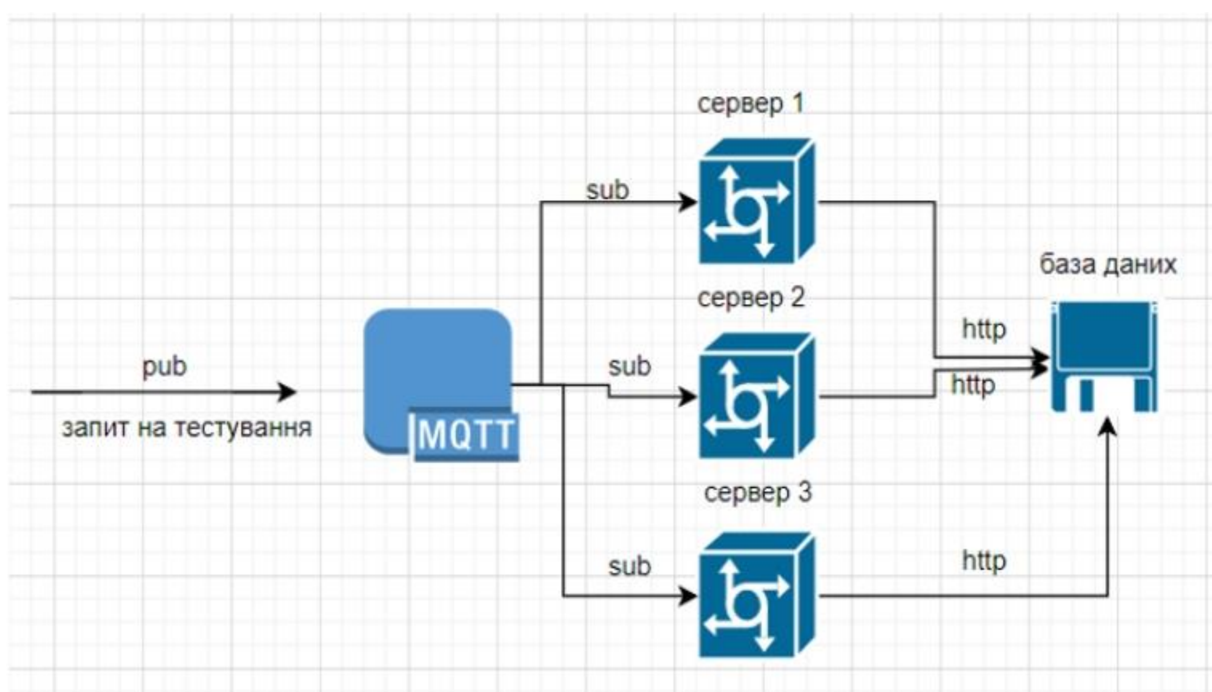

Рис. 3.1 Архітектура системи тестування БД часових рядів

Для тестування бази даних часових рядів потрібно розробити тестовий план.

Зобраземо тестовий план за допомогою Mind Map (додаток 1). Mind map або діаграма зв'язків, інтелект-карта, асоціативна карта чи карта мислення, представляє собою метод візуалізації і структуризації ідей та концепцій за допомогою графічного підходу. Це зображення, яке включає в себе центральну тему чи ідею, оточену вузлами або ключовими словами, які пов'язані лініями чи гілками. Майндмеп допомагає візуалізувати і структурувати інформацію, роблячи зручним аналіз, планування та запам'ятовування концепцій.

Цей тест-план описує стратегію та план дій для тестування максимального навантаження баз даних часових рядів на платформі Victoriametrix.

Метою тестування є оцінка продуктивності та стабільності системи при максимальному навантаженні, забезпечення роботи з великою кількістю часових рядів та визначення можливих обмежень.

Згідно тестового плана потрібно визначити об’єкт тестування - це база даних часових рядів VictoriaMetriks та компоненти для тестування.

Також визначаються метрики тестування:

- кількість оброблених часових рядів за одиницю часу;

- середній час відгуку системи на запити.

Сценарії тестування, критерії прийняття та план тестування також визначаються в документі.

Було визначено основні сценарії тестування:

- запуск системи без навантаження на базу даних,

- запуск системи під максимальним навантаженням

- запуск системи в стрес режимі, тобто навантаження в рази більше спроможності БД.

Також визначаємо середовище де проводиться тестування та хто буде проволити тестування бази даних часових рядів.

Тестування буде проводитися по методології Agile.

Зобразимо робочий процес тестування у вигляді схеми .

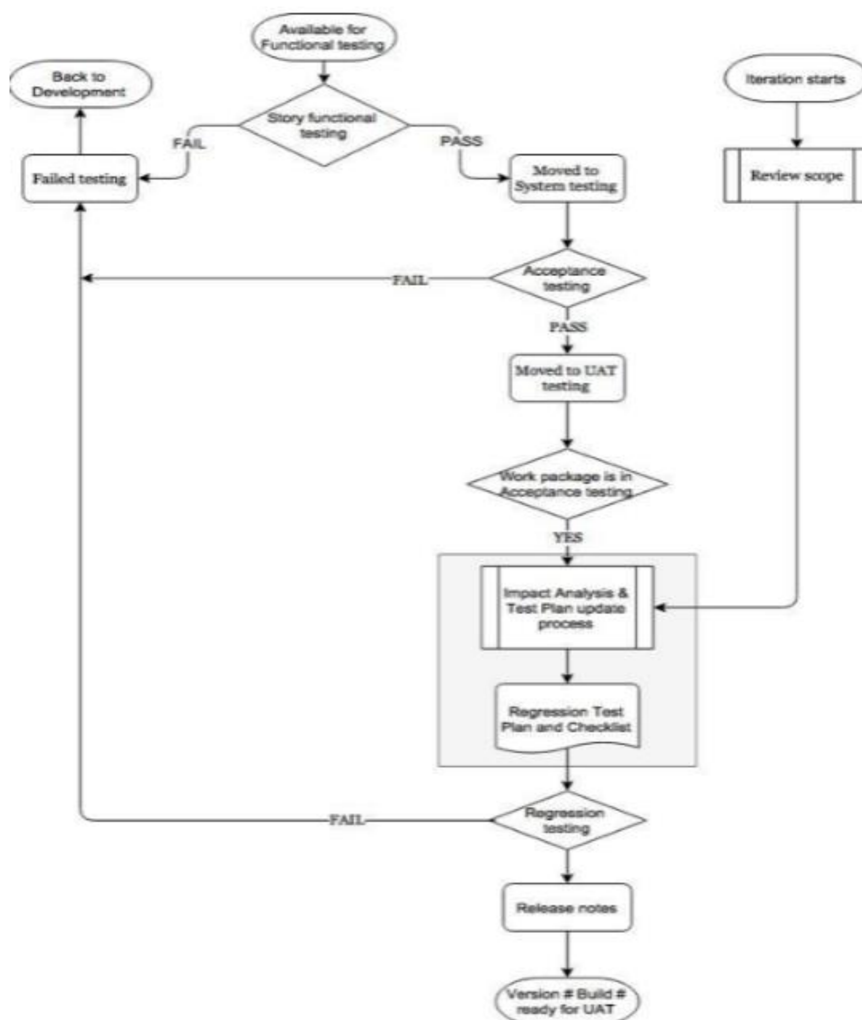

Рис 3.2 QA процес

Також потрібно створити чек лист для тестування та основні тестові кейси. (Рис. 3.3)

Згідно чек листа нам потрібно перевірити роботу бази даних часових рядів при мінімальному навантаженні. Також потрібно перевірити поводження бази даних при середньому навантаженні тривалий час, потім дещо збільшуючи навантаження бази даних до тих пір поки воно не стане максимальним. Після цього потрібно ще збільшити навантаження (кількіть запитів та потоків) щоб перевірити як себе буде поводити база даних, коли потік даних будет більше, ніж може обробити база даних часових рядів.- При цьому потрібно перевяряти навантаження баз даних коли використовувався один сревер та коли декілька.

	Чек лист
1	Створити навантаження 10 запитів/секунду в 10 потоків з одного сервера
2	Створити навантаження 10 запитів/секунду в 10 потоків з двох серверів
3	Створити навантаження 10 запитів/секунду в 10 потоків з трьох серверів
4	Створити навантаження 10 запитів/секунду в 10 потоків з п'яти серверів
5	Створити навантаження 10 запитів/секунду в 20 потоків з одного сервера
6	Створити навантаження 10 запитів/секунду в 20 потоків з двох серверів
7	Створити навантаження 10 запитів/секунду в 20 потоків з трьох серверів
8	Створити навантаження 10 запитів/секунду в 20 потоків з п'яти серверів
9	Створити навантаження 100 запитів/секунду в 10 потоків з одного сервера
10	Створити навантаження 100 запитів/секунду в 10 потоків з двох серверів
11	Створити навантаження 100 запитів/секунду в 10 потоків з трьох серверів
12	Створити навантаження 100 запитів/секунду в 10 потоків з п'яти серверів
13	Створити навантаження 100 запитів/секунду в 20 потоків з одного сервера
14	Створити навантаження 100 запитів/секунду в 20 потоків з двох серверів
15	Створити навантаження 100 запитів/секунду в 20 потоків з трьох серверів
16	Створити навантаження 100 запитів/секунду в 20 потоків з п'яти серверів
17	Створити навантаження 100 запитів/секунду в 50 потоків з п'яти серверів
18	Створити навантаження 100 запитів/секунду в 100 потоків з п'яти серверів
19	Створити таке навантаження, коли кількість надісланих запитів буде більше кількості звписів, що були записані в БД

Рис. 3.3 Чек лист

3.2 Архітектура моделі та її розробка

3.2.1 Робота з часовими рядами в Python

Python став невід'ємним інструментом для аналізу даних завдяки своїй простоті, універсальності та активній підтримці від спільноти. Зрозумілий синтаксис та розгалужена бібліотечна екосистема роблять цю мову програмування ефективним рішенням для вирішення різноманітних завдань.

Незалежно від того, чи розробляєте ви застосунок з великим обсягом даних чи співпрацюєте з досвідченим фахівцем з аналізу даних, Python забезпечує надійну основу для дослідження, візуалізації та моделювання часових рядів.

Python має інтуїтивно зрозумілий синтаксис, що дозволяє легко освоювати його навіть новачкам. Чиста структура коду Python сприяє ефективним методам

кодування, дозволяючи вам фокусуватися на аналізі часових рядів, а не на вивченні складних концепцій програмування.

Однією з великих переваг Python також є те, що це мова програмування з відкритим вихідним кодом. Це означає, що він доступний для безкоштовного використання, постійно удосконалюється і підтримується активною спільнотою розробників. Відкритий вихідний код Python дозволяє фахівцям з даних отримувати доступ до безлічі ресурсів, інструментів і бібліотек для аналізу часових рядів без додаткових витрат.

Python має обширну колекцію спеціалізованих бібліотек і інструментів, спеціально розроблених для аналізу часових рядів. Ці бібліотеки, такі як pandas, NumPy, statsmodels і scikit-learn, надають різноманітні функції та інструменти, адаптовані до унікальних завдань роботи з даними, що залежать від часу. Вони спрощують складні операції, дозволяючи вам зосередитися на витягуванні значущої інформації, а не на вигадуванні велосипедів.

Завдяки своєму широкому поширенню, Python має обширну базу коду, якою можуть скористатися вчені, які працюють з даними, та розробники додатків для своїх потреб у аналізі часових рядів.

Python пропонує широкий спектр можливостей для обробки часових рядів, дозволяючи виконувати різноманітні завдання з аналізу та вилучення інформації з ваших даних.

Побудова графіка часових рядів є важливим етапом візуалізації закономірностей, тенденцій та аномалій. Python надає бібліотеку Matplotlib, яка включає модуль Pyplot для створення різних типів графіків, включаючи лінійні графіки, діаграми розсіювання та гістограми.

Python надає багато бібліотек і методів для прогнозування часових рядів, і одним з популярних методів є модель авторегресійного інтегрованого ковзного середнього (ARIMA).

ARIMA - це потужний та широко використовуваний підхід, який об'єднує три наступні компоненти для виявлення закономірностей та тенденцій у часових рядах:

Авторегресія (AR)

Диференціювання (I)

Ковзне середнє (MA)

Клас ARIMA в бібліотеці statsmodels.tsa.arima.model для застосування в Python. Цей клас дозволяє вказувати порядок компонентів AR, I і MA та підігнати модель до ваших історичних даних. Після підбору моделі ви можете використовувати її для прогнозування майбутніх значень, викликавши метод predict та вказавши дати початку і закінчення прогнозованого періоду.

Очищення даних відіграє вирішальну роль під час аналізу часових рядів, оскільки забезпечує точність та надійність даних, які використовуються для подальшого аналізу та моделювання. У Python ви можете використовувати різні методи та бібліотеки для очищення даних часових рядів і вирішення поширених проблем, таких як відсутні значення, викиди чи несумісності.

Очищення даних при аналізі часових рядів зазвичай включає наступні кроки:

Обробка відсутніх значень. Відсутні значення можуть виникати в даних часових рядів з різних причин, наприклад, через збої датчиків, проблеми з передачею даних або помилки людей. У Python існують бібліотеки, такі як pandas, які пропонують методи для обробки відсутніх значень, такі як інтерполяція, пряме заповнення, зворотне заповнення або видалення рядків з відсутніми значеннями.

Виявлення та обробка викидів. Викиди - це екстремальні значення, які значно відхиляються від нормальних закономірностей в часовому ряді. Виявлення та обробка викидів важлива для уникнення спотворень в аналізі. Бібліотеки Python, такі як pandas, NumPy або scikit-learn, надають методи виявлення та обробки викидів, такі як статистичні методи чи підходи, засновані на машинному навчанні.

Робота з протиріччями чи неправильними даними. Дані часових рядів іноді можуть містити протиріччя або неправильні значення, наприклад, протиріччя в одиницях вимірювання, неприпустимі типи даних чи помилки введення даних. Python пропонує функціональні можливості для очищення та виправлення таких

несумісних даних, включаючи перетворення типів даних, нормалізацію даних чи застосування бізнес-правил для виявлення та виправлення помилкових даних.

Python має безліч переваг для аналізу часових рядів. Це зручна мова програмування. Вона широко поширена в світі відкритого програмного забезпечення. У неї обширна бібліотечна підтримка. Вона може використовувати існуючий код для повторного використання.[37]

3.2.2 Застосування проколу MQTT для одночасного керування групою серверів.

Протокол MQTT (Message Queuing Telemetry Transport) був розроблений у 1999 році для застосування в нафтогазовій промисловості. Інженерам був потрібен протокол із мінімальною пропускною здатністю та мінімальним розрядженням акумулятора для відстеження нафтопроводу з супутника. Спочатку протокол називався "телеметричним транспортом черги повідомлень" за назвою продукту IBM MQ Series, який підтримував його у початковій фазі. У 2010 році IBM випустила MQTT 3.1 як безкоштовний та відкритий протокол, який може реалізувати будь-який клієнт. У 2013 році він був відправлений до підрозділу зі специфікаціями Організації з удосконалення стандартів структурної інформації (OASIS) для доробки. У 2019 році OASIS випустила вдосконалену версію MQTT 5. MQTT більше не є скороченням, але вважається офіційною назвою протоколу.

MQTT (Message Queuing Telemetry Transport) – це протокол обміну повідомленнями, який базується на стандартах і використовується для взаємодії між комп'ютерами. Інтелектуальні датчики, портативні пристрої та інші пристрої Інтернету речей (IoT) зазвичай передають та отримують дані через мережі з обмеженими ресурсами та пропускною здатністю. Ці пристрої Інтернету речей використовують MQTT для передачі даних через мережу, оскільки він легко впроваджується та ефективно обмінюється даними IoT. MQTT підтримує передачу повідомлень від пристроїв до хмари та у зворотньому напрямку.

Протокол MQTT став стандартом для передачі даних IoT завдяки таким перевагам:

1. Легкість та Ефективність.

Реалізація MQTT на пристрої IoT вимагає мінімальних ресурсів і, отже, може використовуватися на невеликих мікроконтролерах.

Мінімальне управлінське повідомлення MQTT може складатися всього з двох байтів даних, а заголовки повідомлень MQTT також невеликі, що дозволяє оптимізувати пропускну здатність мережі.

2. Масштабованість.

Для реалізації MQTT потрібно мінімальну кількість коду, для роботи якого потрібно дуже мало енергії.

Протокол вбудовує функції для взаємодії з великою кількістю пристроїв IoT, що дозволяє реалізувати його для підключення мільйонів таких пристроїв.

3. Надійність. ;Засоби зменшення часу, який потрібно пристроєві IoT для відновлення з'єднання з хмарою, вбудовані в MQTT.

Визначаються три різних рівня обслуговування якості для забезпечення надійності використання IoT: "максимум один раз", "мінімум один раз" і "рівно один раз".

4. Безпека. MQTT спрощує для розробників завдання шифрування повідомлень та аутентифікації пристроїв та користувачів за допомогою сучасних протоколів аутентифікації, таких як OAuth, TLS1.3 та інші.

5. Добра підтримка. Деякі мови програмування, наприклад Python, надають добру підтримку протоколу MQTT, що дозволяє розробникам швидко впроваджувати його з мінімальною кількістю коду у будь-якому типі додатка.

Протокол MQTT працює за моделлю "видавець-підписник". У традиційному мережевому взаємодії клієнти та сервери спілкуються прямо між собою. Клієнти запитують ресурси або дані у сервера, сервер обробляє запит і повертає відповідь. Але MQTT використовує шаблон "видавець-підписник" для відокремлення відправника повідомлення (видавця) від отримувача (підписника). Взаємодію між видавцями та підписниками керує третій компонент - брокер повідомлень.

Завдання брокера - фільтрувати всі вхідні повідомлення від видавців і відправляти їх відповідним підписникам.

Протокол MQTT використовує модель "видавець-підписник" для організації взаємодії між різними пристроями чи клієнтами. В цій моделі взаємодія відбувається через брокера, який виступає посередником між видавцями (клієнтами, що надсилають повідомлення) та підписниками (клієнтами, що отримують повідомлення).

Брокер відокремлює видавців від підписників наступним чином:

1. Розділення в просторі. Коли клієнт публікує повідомлення (виступає як видавець), він не повинен знати IP-адресу або номер порту іншого клієнта (підписника). Брокер відповідає за маршрутизацію цього повідомлення до всіх зацікавлених сторін.

2. Розділення в часі. Видавці та підписники можуть приєднуватися та від'єднуватися від брокера у будь-який момент. Це означає, що вони не обов'язково повинні бути в мережі одночасно, щоб обмінюватися повідомленнями.

3. Роздільна синхронізація. Видавці можуть надсилати повідомлення без очікування відповіді чи підтвердження від кожного підписника. Це дозволяє реалізовувати асинхронний обмін даними, а підписники можуть отримувати та обробляти повідомлення в зручний для них спосіб.

Клієнтом MQTT може бути будь-яке пристрій, від сервера до мікроконтролера, із бібліотекою MQTT. Якщо клієнт надсилає повідомлення, він працює як видавець, а якщо отримує їх, то як підписник. Загалом будь-яке пристрій, яке використовує протокол MQTT для мережевої взаємодії, можна назвати клієнтським пристроєм MQTT.

Брокер MQTT - це серверна система, яка координує повідомлення між різними клієнтами. Брокер відповідає, зокрема, за отримання та фільтрацію повідомлень, ідентифікацію клієнтів, підписаних на кожне повідомлення, і надсилання їм повідомлень. Також він відповідає за виконання деяких інших завдань, наприклад:

- авторизацію та аутентифікацію клієнтів MQTT;

- передачу повідомлень іншим системам для подальшого аналізу;

- обробку пропущених повідомлень та клієнтських сеансів.

Клієнти та брокери починають взаємодію за допомогою підключення MQTT. Клієнт ініціює підключення, надсилаючи брокеру MQTT повідомлення CONNECT. Брокер підтверджує, що підключення встановлено, відповідаючи повідомленням CONNACK. Для взаємодії між клієнтом та брокером MQTT потрібен стек TCP/IP. Клієнти з'єднуються один з одним не прямо, а лише через брокера.

Принцип роботи MQTT:

- клієнт MQTT встановлює з'єднання з брокером MQTT.

- після підключення клієнт може публікувати повідомлення і (або) підписуватися на певні повідомлення.

- коли брокер MQTT отримує повідомлення, він пересилає його зацікавленим підписникам.

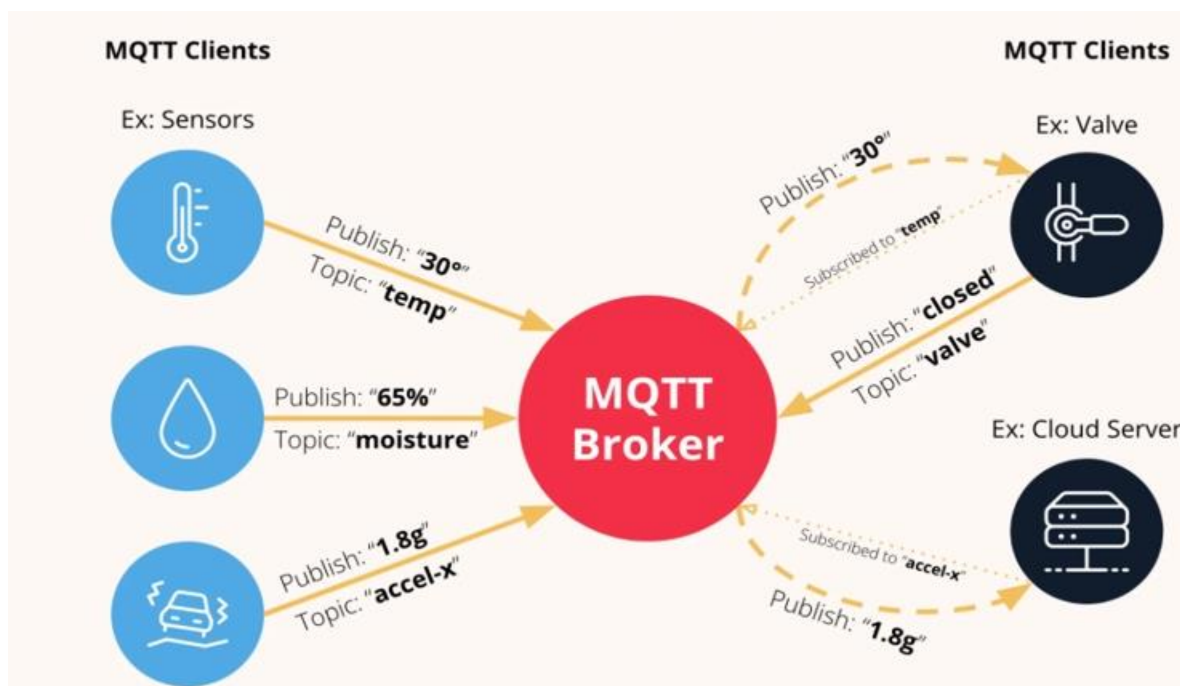

Рис. 3.4 Схема роботи MQTT брокера

Термін "тема" в MQTT означає слова, які використовуються для фільтрації повідомлень, які призначені для клієнтів MQTT. Теми подаються у вигляді

ієрархії, аналогічної файлам та текам. Розглянемо це на прикладі системи моніторингу здоров'я людини з розумними пристроями для вимірювання різних показників.

В даному випадку брокер MQTT може організувати теми наступним чином:

ourbody/hand/pressure

ourbody/hand/temperature

Такий підхід до створення тем дозволяє системі впорядковувати та категоризувати дані, щоб забезпечити ефективний обмін інформацією між різними частинами системи. Теми служать своєрідними мітками або адресами, за допомогою яких клієнти можуть підписуватися на конкретні категорії повідомлень та публікувати їх.

Клієнти MQTT публікують повідомлення, які містять тему та дані у байтовому форматі. Клієнт визначає формат даних файлу: текстовий, бінарний, XML або JSON. Як приклад лампа в системі може опублікувати повідомлення "увімк." у темі "bedroom/light".

Клієнт MQTT відправляє повідомлення SUBSCRIBE брокеру MQTT, щоб отримувати сповіщення про цікаву тему. Це повідомлення містить унікальний ідентифікатор та список підписників.

MQTT по протоколу WebSocket (WSS) - це реалізація MQTT для отримання даних безпосередньо в інтернет-браузері. Протокол MQTT визначає клієнт JavaScript для підтримки WSS у браузерах. У цьому випадку протокол діє як зазвичай, але додає додаткові заголовки до повідомлень MQTT для підтримки протоколу WSS. Це можна уявити як повідомлення MQTT в конверті WSS.

Взаємодія з використанням MQTT використовує протокол SSL для захисту конфіденційних даних, які передаються пристроями Інтернету речей (IoT). Ви можете реалізувати ідентифікацію, аутентифікацію та авторизацію між клієнтами та брокерами за допомогою сертифікатів SSL та/або паролів.

Підтримка MQTT REST: MQTT не є RESTful. Протокол передачі репрезентативного стану (REST) - це архітектурний підхід до мережевої взаємодії, використовуючи шаблон "запит-відповідь" між відправниками та отримувачами.

Навпаки, MQTT використовує модель взаємодії "видавець-підписник" на рівні додатків та вимагає стійкого підключення TCP для передачі push-сповіщень. Однак у MQTT версії 5 додано новий метод "запит-відповідь", який діє подібно до REST і дозволяє видавцеві прикріпити спеціальну тему відповіді, яку підписник обробляє, а потім генерує відповідь.

У підсумку, протокол MQTT, завдяки своїм технічним перевагам та гнучкій архітектурі, визначає стандарти для створення надійних, масштабованих та безпечних застосувань в області Інтернету речей, що забезпечує прискорену реалізацію різноманітних сценаріїв у різних галузях промисловості та технологій.[36]

3.2.3 Розробка та впровадження системи тестування максимального навантаження бази даних часових рядів

Згідно нашого дизайну запит буде йди на MQTT брокер, потім сервери,які підписані на даний брокер отримают запит та згенерують необхідну кількість запитів в базу даних. Сервери будуть на мові Python, база даних Victoriametrics.

Почнемо зі встановлення пакетів для Python:

- python -m pip install paho-mqtt requests

де paho-mqtt - це бібліотека MQTT для Python, яка надає інструменти для роботи з протоколом MQTT, а requests - це бібліотека для відправлення HTTP-запитів.

Поставимо MQTT сервер:

sudo apt install mosquitto

sudo apt install mosquitto-client

Встановлюємо IP сервера та вказуємо його в Python скрипт.

На іншому сервері встановимо victoriametriks BD:

sudo apt update

sudo apt install snapd

sudo snap install victoriametrics.

Запустимо БД: sudo snap start victoriametrics

Перевіримо статус бази даних: sudo snap services victoriametrics

Визначимо айпи сервера с бд: Ip a

Отриманий IP вставимо в Python скрипт.

Виконаємо Python скрипт.

```
1  import paho.mqtt.subscribe as subscribe
2  import paho.mqtt.publish as publish
3
4  from multiprocessing import Pool
5  import requests
6  import time
7  from datetime import datetime
8  import json
9
10 def on_message_print(client, userdata, message):
11     print("%s %s" % (message.topic, message.payload))
12     payload = message.payload.decode()
13     try:
14
15         workers, queryes = payload.split('x')
16
17         response = requests.get('''http://192.168.1.92:8428/api/v1/admin/tsdb/delete_series?match={__name__="test"}''')
18         print(response)
19
20         test_list = []
21         for i in range(int(queryes)*int(workers)):
22             test_list.append(i)
23
24         print(len(test_list))
25         ts_start = datetime.today().timestamp()
26         with Pool(int(workers)) as p:
27             p.map(worker, test_list)
28         ts_end = datetime.today().timestamp()
29
30         print(ts_end-ts_start)
```

```
31
32         time.sleep(5)
33         try:
34             response = requests.get('''http://192.168.1.92:8428/api/v1/export?match={__name__="test"}''')
35             out = response.json()["values"]
36             print(len(out))
37             publish.single("run", "server1.read: "+str(len(out))+" time: "+ str(ts_end-ts_start)+" sec", hostname="192.168.1.56")
38
39         except:
40             print("error export")
41
42     except:
43         publish.single("log", "query error", hostname="192.168.1.56")
44
45 def worker(par):
46
47     response = requests.post("http://192.168.1.92:8428/api/v1/import/prometheus", data='''test{test1="test1"} 123''')
48     #print(response)
49
50
51
52 if __name__ == '__main__':
53     subscribe.callback(on_message_print, "run", hostname="192.168.1.56")
```

Рис. 3.5 Код на Python системи тестування БД

Додамо деталей щодо роботи Python скрипт.

Спочатку імпортуємо бібліотеки для роботи з MQTT брокером.

Підписуємся на MQTT брокер. Визначаємо необхідну кількість воркерів.

Надсилаємо запит на адрессу нашої бази даних в кількості воркер помножити на кількість запитів. Потім отримуємо відповідь про кількість записів в БД. Також потрібно не забувати прокоманду, яка видаляє дані з бази даних чтоб кожний новий запит ніс інформацію саме про останній запис.

Таким чином мі можемо конролювати кількість запитів та кількість потоків до бази даних та отримати відповідь про реальну кількість записів в БД.

3.3 Тестування бази даних часових рядів

3.3.1 Тестування функціональності та навантаження

Тестування нашої системи для визначення навантаження на сервер є однією з важливіштх речей нашого проекту. Саме по результатам тестування ми зможемо визначити скільки саме можно обробити наша база даних та за який проміжок часу, також вирахувати середню кількість запитів за одну секунду при різному навантаженні та з використанням різнох кількості серверів.

Виконаємо команди в консолі сервера:

mosquitto_sub -t run

Виконаємо наші тестові сценарії.

Почнемо з надсилання мінімальної кількісті запитів(до 100 штук).

Для цього виконаємо команду:

mosquitto_pub -t run -m 5x5, де перша цифра це кількість запитів, а друга кількість воркерів.

Почнемо для одного сервера, поступово збільшуючи кількість запитів та серверів.

Також перевіримо поводження бази даних якщо стабільне навантаження буде йти певний час.

```
5x5
server1.read: 25 time: 0.4089090824127197 sec
```

```
10x10
server1.read: 100 time: 0.8979380130767822 sec
20x20
server1.read: 400 time: 3.5052871704101562 sec
30x30
server1.read: 900 time: 6.75911808013916 sec
50x50
server1.read: 2500 time: 17.416495084762573 sec
1x50
server1.read: 50 time: 0.825502872467041 sec
50x1
server1.read: 50 time: 1.3343350887298584 sec
1x500
server1.read: 500 time: 8.748691082000732 sec
500x1
```

Рис.3.6 Результати тестування з використанням 1 сервера

Виконаємо одну й ту ж кількість запитів напротязі певного часу.

```
server1.read: 100 time: 0.8459300994873047 sec
server1.read: 100 time: 0.8473520278930664 sec
server1.read: 100 time: 0.810265064239502 sec
server1.read: 100 time: 1.756941795349121 sec
server1.read: 100 time: 4.60046911239624 sec
server1.read: 100 time: 0.7511539459228516 sec
server1.read: 100 time: 0.9448111057281494 sec
server1.read: 100 time: 0.8625988960266113 sec
server1.read: 100 time: 0.7704651355743408 sec
server1.read: 100 time: 0.7344040870666504 sec
server1.read: 100 time: 0.8335421085357666 sec
server1.read: 100 time: 0.9299960136413574 sec
```

Рис. 3.7 Продовження результатів тестування з використанням 1 сервера

```
70x70
server1.read: 4900 time: 36.61128497123718 sec
80x80
server1.read: 6400 time: 44.1324200630188 sec
```

Рис. 3.8 Продовження результатів тестування з використанням 1 сервера

```
1x50
server1.read: 50 time: 0.825502872467041 sec
50x1
server1.read: 50 time: 1.3343350887298584 sec
1x500
server1.read: 500 time: 8.748691082000732 sec
500x1
```

Рис. 3.9 Продовження результатів тестування з використанням 1 сервера

Тепер та ж кількість запитів, але для 2 серверів

```
5x5
server1.read: 50 time: 0.5180258750915527 sec
server2.read: 50 time: 1.9370601177215576 sec
10x10
server1.read: 200 time: 1.316498041152954 sec
server2.read: 200 time: 3.3516790866851807 sec
```

Рис.3.10 Результати тестування з використанням 2 серверів

```
20x20
server1.read: 800 time: 5.589592933654785 sec
30x30
server2.read: 193 time: 8.926727056503296 sec
server1.read: 1018 time: 7.953088998794556 sec
server2.read: 1407 time: 12.025331974029541 sec
30x30
server1.read: 1793 time: 10.912477016448975 sec
server2.read: 1800 time: 15.420891046524048 sec
50x50
server1.read: 4291 time: 22.260270833969116 sec
server2.read: 5000 time: 30.70643186569214 sec
```

Рис. 3.11 Продовження результатів тестування з використанням 2 серверів

Використаємо 3 сервери для запитів.

```
5x5
server3.read: 75 time: 1.2560839653015137 sec
server2.read: 75 time: 2.393418073654175 sec
server1.read: 75 time: 3.502366781234741 sec
10x10
server3.read: 300 time: 3.921492099761963 sec
server2.read: 300 time: 5.112623929977417 sec
server1.read: 300 time: 9.049300193786621 sec
20x20
server1.read: 817 time: 3.66451096534729 sec
server3.read: 1200 time: 6.579342842102051 sec
server2.read: 1200 time: 10.314777135848999 sec
30x30
server1.read: 1991 time: 9.348246812820435 sec
server3.read: 2113 time: 10.560313940048218 sec
server2.read: 2700 time: 17.72895908355713 sec
50x50
server1.read: 5310 time: 23.568445920944214 sec
server3.read: 5501 time: 24.350739002227783 sec
server2.read: 7500 time: 41.633142948150635 sec
```

Рис.3.12 Результати тестування з використанням 3 серверів

Використаємо 4 сервери для запитів.

```
5x5
server1.read: 100 time: 0.5170481204986572 sec
server3.read: 100 time: 1.7843940258026123 sec
server4.read: 100 time: 2.05979585647583 sec
server2.read: 100 time: 2.893030881881714 sec
10x10
server1.read: 322 time: 1.1975560188293457 sec
server4.read: 400 time: 3.8425700664520264 sec
server3.read: 400 time: 4.309694051742554 sec
server2.read: 400 time: 6.417781829833984 sec
20x20
server1.read: 1180 time: 4.324438095092773 sec
server3.read: 1409 time: 9.127754926681519 sec
server4.read: 1409 time: 9.310077905654907 sec
server2.read: 1600 time: 14.877569913864136 sec
30x30
server1.read: 2014 time: 9.691956996917725 sec
server3.read: 3060 time: 17.164741039276123 sec
server4.read: 3109 time: 17.240314960479736 sec
server2.read: 3600 time: 26.114917993545532 sec
50x50
server1.read: 5466 time: 28.798716068267822 sec
server4.read: 7945 time: 40.25099492073059 sec
server3.read: 7945 time: 40.170501947402954 sec
server2.read: 10000 time: 57.721603870391846 sec
```

Рис.3.13 Результати тестування з використанням 4 серверів

Протестуємо для 5 серверів

```
5x5
server1.read: 125 time: 0.8282229900360107 sec
server1.read: 125 time: 0.8022520542144775 sec
server3.read: 125 time: 1.935175895690918 sec
server4.read: 125 time: 2.046757936477661 sec
server2.read: 125 time: 3.039051055908203 sec
10x10
server1.read: 500 time: 2.1954500675201416 sec
server1.read: 500 time: 2.27231502532959 sec
server4.read: 500 time: 3.2535760402679443 sec
server3.read: 500 time: 4.332871913909912 sec
server2.read: 500 time: 6.093374013900757 sec
20x20
10x10
server1.read: 500 time: 2.131765127182007 sec
server1.read: 500 time: 2.328115940093994 sec
server3.read: 500 time: 3.7825980186462402 sec
server4.read: 500 time: 3.786223888397217 sec
server2.read: 500 time: 5.505100965499878 sec
20x20
server1.read: 1861 time: 7.849038124084473 sec
server1.read: 1861 time: 7.858565092086792 sec
server3.read: 2000 time: 9.262087106704712 sec
server4.read: 2000 time: 9.296523094177246 sec
server2.read: 2000 time: 12.923449993133545 sec
30x30
server1.read: 3716 time: 17.914443016052246 sec
server1.read: 3716 time: 18.022972106933594 sec
server4.read: 3977 time: 20.406509160995483 sec
server3.read: 4084 time: 20.941370964050293 sec
server2.read: 4500 time: 28.51231598854065 sec
50x50
server1.read: 10272 time: 40.988866090774536 sec
server1.read: 10272 time: 41.29880690574646 sec
server3.read: 10272 time: 41.86353898048401 sec
server4.read: 10272 time: 42.181320905685425 sec
```

Рис.3.14 Результати тестування з використанням 5 серверів

3.3.2 Аналіз результатів тестування

Згідно раніше розробленого тестового плану та сценаріїв було протестовано нашу базу даних Victoriametriks під різним навантаженням.

Аналізуючи результати тестування ми можемо сказати, що при збільшенні кількості запитів сбільшується и загальний час на виконная цих запитів, те ж саме при збільшенні серверів. Так для 25 запитів при схемі 5х5 для одного сервера необхідно 0,4 секунди, для 100 запитів при схемі 10х10 - 0,89 с., для 900 -6,75, для 2500 - 17,41, тобто час зростає при збільшенні навантаженні, але середнє значення на обробку одного запиту падає при збільшенні кількості запитів та потоків на перших порах та в подальшому є стабільним. Найбільший приріст продуктивності спостерігається при збільшенні кількості запитів з 25 штук до 100 штук. В подальшому продуктивніть майже не змінюється та коливается в межах 0,07 с. для одного запиту. При надсиланні більше 7 тисяч запитів БД перестає відповідати.

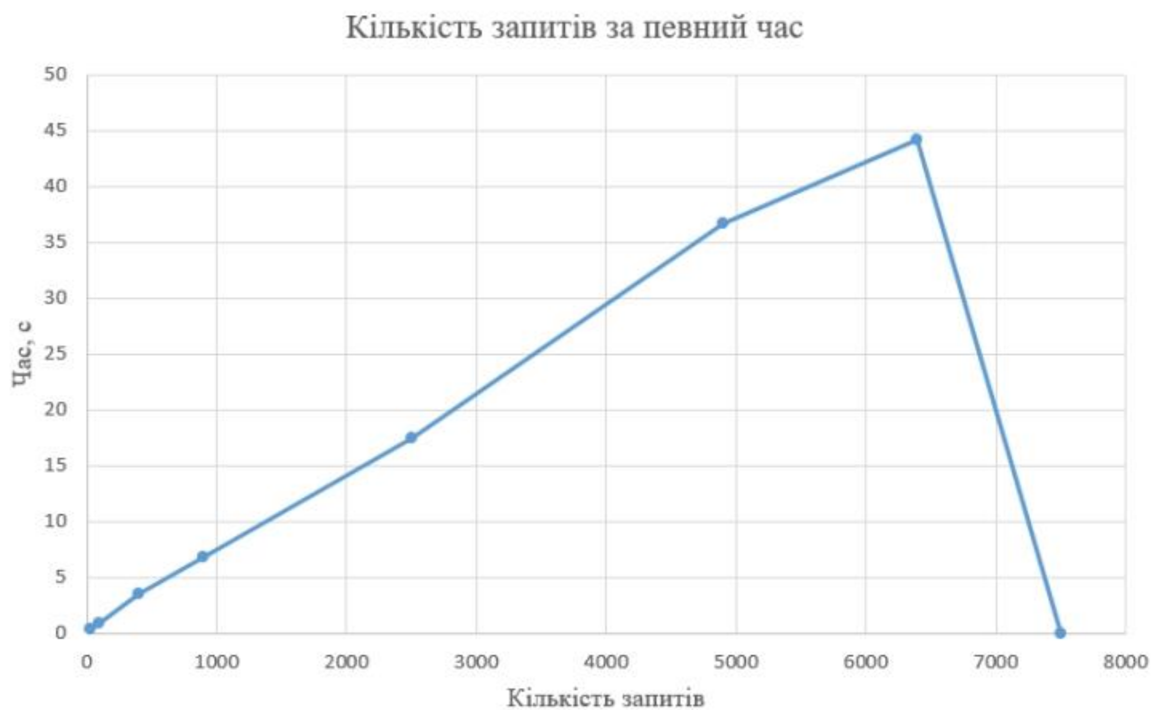

Рис. 3.15 Графік залежності кількості запитів за певний час, один сервер

Побудуємо графік залежності часу на виконная одного запиту та кількості запитів.

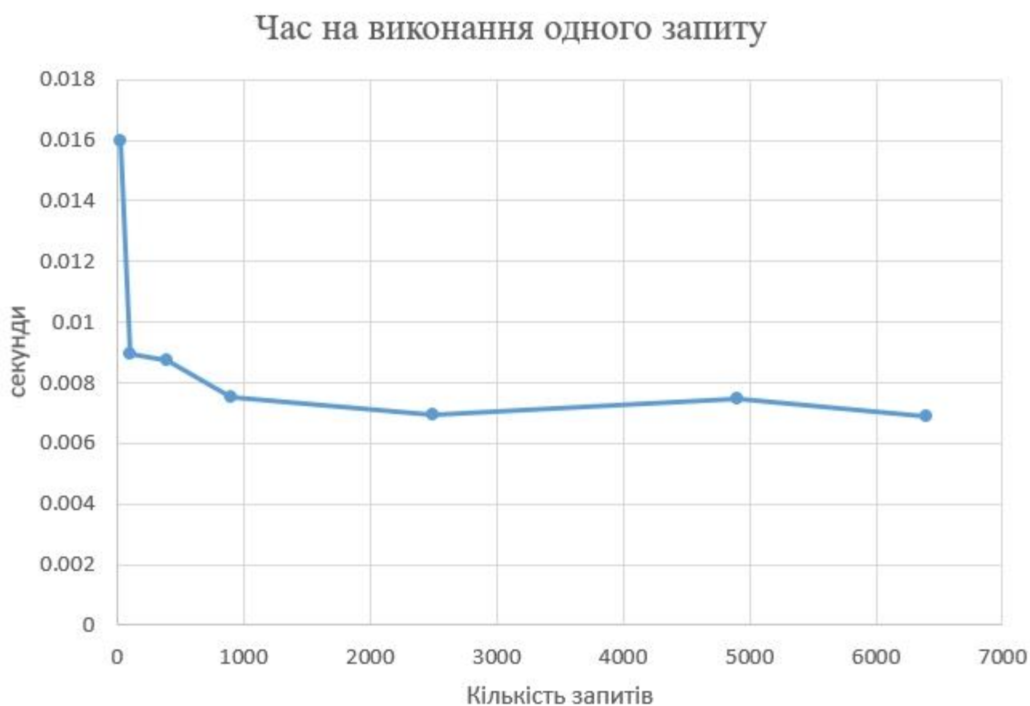

Рис. 3.16 Час на виконання одного запиту при різному навантаженні, один сервер.

При збільшенні кількості серверів кількість запитів до бази даних зростає, при цьому час на виконання одного запиту не суттєво, але при максимальному навантаженны зменшується до 0,06 с. для одного запиту. Тобто база даних показує стабільну роботу при збільшенні навантаження, але при перевищенні загальної кількості в понад 7000 запитів з одного сервера, час обробки зростає в рази та база даних перестає відповідати.

Також було помічено залежність часу обробки одного запиту від способу генерації запитів - чим більше запитів і менше потоків, тим більше потрібен час на обробку запитів. Так 50 запитів надісланих як один запит в 50 потоків виконується за 0,8 с., а 50 запитів в один потік за 1,34 с., тобто зростання на 67,5%. І чим більше запитів буде в одному потоці, тим більша різниця.

ВИСНОВКИ

Магістерська робота присвячена розробці системи тестування максимального навантаження бази даних часових рядів.

Дана робота представляє собою глибоке дослідження сучасних баз даних, спрямованих на обробку та зберігання часових рядів. У процесі аналізу було виявлено, що існує багато підходів та технологій у цьому напрямі, кожен з яких має свої переваги та обмеження. На основі цього дослідження вдалося визначити оптимальні рішення для конкретних вимог щодо обробки часових рядів. Також було досліджено різні метології та підходи до тестування, як одного з головних складових перевірки якості розробленої моделі.

Розроблена система тестування виявилася надзвичайно корисною для оцінки продуктивності бази даних часових рядів в умовах максимального навантаження. Вона дозволила не лише визначити оптимальні параметри для використання бази даних у реальних умовах, але й виявити області для подальшої оптимізації та підвищення ефективності.

Для створення ситеми тестування було використано декілька серверів реалізованих на мові Python, MQTT брокер та база даних Victorismetriks, яка має декілька переваг перед іншими базами даних часових рядів.

Проведені експерименти та аналіз результатів дали змогу зрозуміти, як база даних часових рядів веде себе при великому обсязі запитів та які чинники впливають на її продуктивність. Це може слугувати цінною інформацією для подальших розробок та оптимізацій.

Важливою частиною магістерської роботи було практичне використання розробленої системи тестування для моніторингу та аналізу продуктивності бази даних у реальному часі. Це відкриває можливості для активного вдосконалення та використання отриманих результатів у великих та високонавантажених IoT проектах.

Загалом, магістерська робота відкриває перспективи для подальших досліджень у галузі оптимізації баз даних часових рядів, використання їх у великих інформаційних системах та впровадження нових технологій для підвищення продуктивності та масштабованості.

ПЕРЕЛІК ПОСИЛАНЬ

1. Doug Vucevic, Wayne Yaddow. "Testing the Data Warehouse Practicum: Assuring Data Content, Data Structures and Quality". John Wiley & Sons, 2019. 576 с.
2. "Guide to the Software Engineering Body of Knowledge", v.4, 2022.
3. Майк Коеніґ. "Lean Software Development in Action". Manning Publications, 2019. 376 с.
4. R.E. Fairley, "Managing and Leading Software Projects", Wiley-IEEE Computer Society Press, 2020.
5. Lee Copeland. "A Practitioner's Guide to Software Test Design". Artech House, 2019. 368 с.
6. Gerardus Blokdyk. "Database testing", 5STARCooks, 2020. 87 с.
7. Dorothy Graham, Erik van Veenendaal. "Foundations of Software Testing", Cengage Learning, 2019. 432 с.
8. Вільям Еванс. "Чорна скринька. Методика та інструментарій тестування програмного забезпечення". Київ: Видавництво "К", 2021. 304 с.
9. Сэм Канер, Джек Фолк. "Testing Computer Software", Print2print, 2019. 544 с.
10. Кенґлін Лі. "Ефективна автоматизація тестування програмного забезпечення". Київ: Видавництво "К", 2018. 320 с.
11. Ron Jones. "Effective Software Test Automation: Developing an Automated Software Testing Tool". Wiley, 2020. 640 с.
12. TestNG [Електронний ресурс] – Режим доступу до ресурсу: https://www.testng.org
13. Selenium [Електронний ресурс] – Режим доступу до ресурсу: https://www.selenium.dev
14. DBUnit [Електронний ресурс] – Режим доступу до ресурсу: https://www.dbunit.org
15. Postman [Електронний ресурс] – Режим доступу до ресурсу: https://www.postman.com

16. Apache Jmeter [Електронний ресурс] – Режим доступу до ресурсу: https://www.apache.jmeter.org
17. Pytest-Django [Електронний ресурс] – Режим доступу до ресурсу: https://pytest-django.readthedocs.io
18. Abraham Silberschatz, Henry F. Korth, S. Sudarshan. Database System Concepts. McGraw-Hill Education, 2019. 1376 с.
19. C.J. Date. "SQL and Relational Theory: How to Write Accurate SQL Code". O'Reilly Media, 2019. 582 с.
20. DB engines [Електронний ресурс] – Режим доступу до ресурсу: https://db-engines.com/en/
21. Берко А.Ю., Верес О.М. "Системи баз даних та знань, книга 2: системи управління базами даних та знань." Київ: Видавництво «Магнолія 2006», 2021. 584 с.
22. Martin Kleppmann. "Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems". Видавництво: O'Reilly Media, 2017. 616 с.
23. Андреас Шолц, Лайза Гроссманн. "Інтернет речей: Додатки та розробка з Node-RED". Київ: Видавництво "БХВ-Петербург", 2020. 416 с.
24. Kaitlyn Hanson, Michael Kraft. "Internet of Things: Architectures, Protocols, and Standards". Видавництво: CRC Press, 2021. 424 с.
25. James Groff, Andrew Hellmann. "Programming the Internet of Things with Node.js". Видавництво: O'Reilly Media, 2019. 336 с.
26. Adrian McEwen, Hakim Cassimally. "Designing the Internet of Things". Wiley, 2019. 336 с.
27. O'Reilly Media. "The Internet of Things: What it is and why you should care". O'Reilly Media, 2018. 120 с.
28. William S. Klein. "Аналіз часових рядів: прогнозування та контроль". Видавництво Wiley, 2019. 784 с.
29. Scott Shell. "Time Series Analysis: A Time Domain Approach". John Wiley & Sons, 2019. 512 с.

30. Chris J. Date. "An Introduction to Database Systems". Pearson, 2021. 1256 с.
31. Michael Stonebraker, Joseph Hellerstein. "Readings in Database Systems". Morgan Kaufmann, 2018. 792 с
32. Hank Young. "Time Series Databases in Action". Manning Publications, 2020. 320 с.
33. Victoriametrics [Електронний ресурс] – Режим доступу до ресурсу: https://victoriametrics.com/
34. Lee Vo, Hoang Tran. "TimescaleDB for Timeseries: Scalable and Reliable Real-time Analytics". O'Reilly Media, 2021. 300 с.
35. Grafana [Електронний ресурс] – Режим доступу до ресурсу: https://grafana.com/
36. Dave Smith. "Programming the Internet of Things with MQTT". Manning Publications, 2018. 300
37. Lucy Folkingston. "Python for Time Series Analysis". O'Reilly Media, 2019. 300 с.

ДОДАТОК А

Mind map Тест-плану для Тестування Навантаження БД часових рядів