БАКАЛАВРСЬКА РОБОТА

БР.ІП – 52.00.00.000 ІПЗ

Група ІП-22-2

Фуфалько Денис

2026

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Фуфалько Денис Іванович

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

БАКАЛАВРСЬКА РОБОТА

Роль баз даних у архітектурі програмних систем
(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня _____Фуфалько Д.І. _____
(підпис, ініціали та прізвище здобувача)

Науковий керівник ____Ксенич Андрій Іванович, доцент, к.т.н_____
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

_доц._____Бандура В.В.__
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доцент. В.В. Бандура
“ ____ ” _____2026 р.

ЗАВДАННЯ
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Фуфалько Денису Івановичу

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) "Роль баз даних у архітектурі програмних систем"

керівник проекту (роботи) доцент, к.т.н., Ксенич А.І.

затверджені наказом вищого навчального закладу від “ 21_ ” _квітня 2026 р. № ___179/7____

2. Строк подання студентом проекту (роботи) 10 червня 2026 р.

3. Вихідні дані до проекту (роботи) Результати і матеріали отримані під час проходження переддипломної практики

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1. Аналіз вимог до програмного забезпечення
2. Аналіз вимог і постановка задачі
3. Проектування системи
4. Реалізація проекту
5. Тестування та впровадження

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1 NoSQL — основні особливості. (рис. 2.1, ст. 29)

2 Комбінація CAP (рис. 2.2, ст. 31)

3 Принцип забезпечення узгодженості схеми (рис. 2.8, ст. 45)

4 Впровадження системи управління транзакціями (рис. 3.1, ст. 49)

5 Структури зберігання даних у базах даних SQL та NoSQL (рис. 3.7, ст. 61)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

2. Дата видачі завдання **2026 р.**

Керівник _____
(підпис)

Завдання прийняв до виконання_____ _____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Визначення та обгрунтування теми роботи	17.03.2026	виконано
2	Огляд існуючих концепцій, рішень та сервісів в даній області	21.04.2026	виконано
3	Побудова моделі або алгоритму власного рішення	01.05.2026	виконано
4	Документування реалізації власного оригінального рішення вибраними засобами	21.05.2026	виконано
5	Оформлення пояснювальної записки бакалаврської роботи	09.06.2026	виконано

Студент _____Фуфалько Д.І.
(підпис)

Керівник роботи _____. Ксенич А.І
(підпис)

АНОТАЦІЯ

Бакалаврська робота робота містить 86 сторінок, 23 рисунка, 5 таблиць, список використаних джерел із 40 найменувань.

Метою роботи є дослідження ролі баз даних у архітектурі програмних систем та аналіз підходів до їх ефективної інтеграції для забезпечення надійності, масштабованості та продуктивності програмного забезпечення.

Об'єкт дослідження: архітектура програмних систем із використанням баз даних.

Предмет дослідження: методи, моделі та інструменти інтеграції баз даних у програмні системи, включаючи архітектурні підходи, системи управління базами даних та технології обробки даних.

Результати дослідження: проведено аналіз ролі баз даних у побудові програмних систем, досліджено сучасні архітектурні підходи, розглянуто методи управління даними в розподілених середовищах.

У першому розділі розглянуто теоретичні основи баз даних, їх місце в архітектурі програмних систем, сучасні підходи до побудови архітектур та інтеграцію баз даних у гнучкі методології розробки.

У другому розділі досліджено методи та інструменти використання баз даних, включаючи архітектурні підходи, управління ресурсами в розподілених системах та еталонні архітектури систем управління базами даних.

У третьому розділі розглянуто практичні аспекти розробки та інтеграції баз даних, особливості роботи з напівструктурованими та мультимедійними даними, а також підходи до обробки великих даних у програмних системах.

Висновок: використання сучасних баз даних у архітектурі програмних систем дозволяє забезпечити високу продуктивність, масштабованість і надійність систем, однак потребує ретельного вибору архітектурних рішень.

КЛЮЧОВІ СЛОВА: БАЗИ ДАНИХ, АРХІТЕКТУРА ПРОГРАМНИХ СИСТЕМ, СУБД, РОЗПОДІЛЕНІ СИСТЕМИ, BIG DATA, NOSQL, МАСШТАБОВАНІСТЬ, ПРОДУКТИВНІСТЬ.

ANNOTATION

Bachelor's thesis work contains 86 pages, 23 figures, 5 tables, and a reference list with 40 sources.

The aim of the work is to study the role of databases in the architecture of software systems and to analyze approaches to their effective integration to ensure reliability, scalability, and performance of software solutions.

Research object: software system architectures that utilize databases.

Research subject: methods, models, and tools for integrating databases into software systems, including architectural approaches, database management systems, and data processing technologies.

Research results: an analysis of the role of databases in software system architecture was conducted, modern architectural approaches were studied, methods of data management in distributed environments were reviewed, and the effectiveness of different database types was evaluated.

The first section describes the theoretical foundations of databases, their role in software architecture, modern architectural approaches, and integration with agile development methodologies.

The second section analyzes methods and tools for database usage, including architectural patterns, resource management in distributed systems, and reference architectures of database management systems.

The third section covers the implementation and integration of databases, features of semi-structured and multimedia data handling, and approaches to big data processing in modern software systems.

Conclusion: the use of modern databases in software architecture improves system performance, scalability, and reliability, but requires careful selection of architectural solutions and technologies depending on system requirements.

KEY WORDS: DATABASES, SOFTWARE ARCHITECTURE, DBMS, DISTRIBUTED SYSTEMS, BIG DATA, NOSQL, SCALABILITY, PERFORMANCE.

ЗМІСТ

ВСТУП

У сучасному світі інформаційні системи є невід'ємною складовою практично всіх сфер діяльності, а ефективне управління даними відіграє ключову роль у забезпеченні їхньої продуктивності, надійності та масштабованості. Зі зростанням обсягів даних, появою розподілених систем та необхідністю обробки інформації в реальному часі, бази даних стають центральним елементом архітектури програмних систем. Вони забезпечують збереження, узгодженість і доступність даних, а також впливають на загальну продуктивність і якість програмного забезпечення. Сучасні виклики, пов'язані з інтеграцією різнорідних джерел даних, забезпеченням безпеки та підтримкою високих навантажень, вимагають глибокого розуміння ролі баз даних у програмній архітектурі.

Актуальність теми зумовлена необхідністю побудови ефективних архітектур програмних систем, у яких бази даних виступають не лише засобом зберігання інформації, а й активним компонентом, що визначає підходи до проєктування, інтеграції та масштабування систем. Сучасні програмні рішення, зокрема розподілені та мікросервісні архітектури, потребують використання різних типів систем управління базами даних, включаючи реляційні, NoSQL та мультимодельні рішення. Існуючі підходи часто стикаються з проблемами продуктивності, складності інтеграції та забезпечення узгодженості даних, що підкреслює необхідність дослідження ефективних методів використання баз даних у програмних системах.

Метою роботи є дослідження ролі баз даних у архітектурі програмних систем, аналіз сучасних підходів до їх використання та визначення ефективних методів інтеграції і управління даними для забезпечення високої продуктивності та надійності програмного забезпечення.

Завданнями дослідження є аналіз теоретичних основ архітектури програмних систем і ролі баз даних у них, дослідження сучасних методів і технологій управління даними, зокрема систем управління базами даних та підходів до побудови розподілених систем, вивчення інструментів інтеграції баз

даних у програмні системи, аналіз підходів до роботи з великими обсягами даних, а також оцінка ефективності різних архітектурних рішень.

Об’єктом дослідження є процеси проєктування та функціонування програмних систем із використанням баз даних, що охоплюють етапи аналізу вимог, проєктування архітектури, реалізації та інтеграції компонентів системи.

Предметом дослідження є методи, моделі та технології використання баз даних у архітектурі програмних систем, включаючи реляційні та нереляційні моделі даних, механізми інтеграції, управління транзакціями, забезпечення узгодженості та обробки великих обсягів даних.

Методи дослідження включають аналіз наукових джерел для вивчення сучасних підходів до побудови архітектури програмних систем, порівняльний аналіз різних типів систем управління базами даних, моделювання архітектурних рішень, а також експериментальні методи для оцінки ефективності використання баз даних у різних сценаріях.

У роботі розглядаються теоретичні основи ролі баз даних у програмній архітектурі, досліджуються сучасні методи та інструменти управління даними, а також аналізуються підходи до інтеграції баз даних у програмні системи різної складності. Особлива увага приділяється питанням масштабованості, продуктивності, узгодженості та безпеки даних, що є критично важливими для сучасних інформаційних систем. Очікується, що результати дослідження сприятимуть підвищенню ефективності проєктування програмних систем і оптимізації процесів управління даними.

Бакалаврська робота містить 86 сторінок, 23 рисунків, 5 таблиць, 3 розділи, висновки та список використаних джерел із 40 найменувань.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ РОЛІ БАЗ ДАНИХ У ПРОГРАМНИХ СИСТЕМАХ

1.1 Поняття баз даних та їх місце в архітектурі програмних систем

Для великих, складних програмних систем проектування загальної структури системи (архітектури програмного забезпечення) є суттєвим викликом. Архітектура *програмної* системи визначає цю систему з точки зору компонентів та зв'язків між цими компонентами. Не сам *проект* цієї системи є більш детальним. Архітектура показує відповідність між вимогами та побудованою системою, тим самим надаючи певне обґрунтування для проектних рішень. Цей рівень проектування розглядався різними способами, включаючи неформальні діаграми та описові терміни, мови взаємозв'язку модулів та фреймворки для систем, що обслуговують потреби конкретних областей застосування. Архітектура втілює рішення щодо властивостей якості. Вона являє собою найпершу можливість для оцінки цих рішень. Крім того, можливість повторного використання компонентів та послуг залежить від того, наскільки сильно вони пов'язані з іншими компонентами в архітектурі системи. Продуктивність, наприклад, значною мірою залежить від складності необхідної координації, зокрема, коли компоненти розподіляються через певну мережу. Архітектура зазвичай є першим артефактом, який досліджується, коли програміст (особливо програміст з обслуговування), незнайомий із системою, починає працювати над нею. Архітектура програмного забезпечення часто є першим артефактом дизайну, який відображає рішення щодо того, як мають бути досягнуті вимоги всіх типів. Як прояв ранніх проектних рішень, вона являє собою проектні рішення, які найважче змінити, і тому найбільше заслуговують на ретельне розгляд.

Минуле: Фокус на описі та повторному використанні архітектури

Задовго до того, як архітектура програмного забезпечення виникла як дисципліна, зазначили що програмній системі недостатньо просто виконувати

правильні функції. Інші якості програмного забезпечення, такі як надяійність та зручність обслуговування, також важливі та можуть бути досягнуті лише шляхом ретельного структурування програмних систем.

Концепція архітектури програмного забезпечення як окремої дисципліни почала формуватися в 1990-х роках з мовами опису архітектури, формалізацією та класифікацією архітектурних стилів. Вивчення архітектури програмного забезпечення розвинулося з основоположної роботи. Мови опису архітектури (ADL) з'явилися для забезпечення формальної точності представлення архітектури. Стандарт ANSI/IEEE, IEEE-Std-1471-2000, має на меті кодифікувати найкращі практики та знання як спільноти системної, так і програмної інженерії в галузі документації архітектури [2].

Структури, перевірені на практиці, були каталогізовані та пояснені як шаблони [1] . На рівні програмування повторне використання зазвичай здійснюється за допомогою конструкцій мов програмування високого рівня, бібліотек функцій або об'єктно-орієнтованих фреймворків класів. На рівні проектування шаблони проектування та встановлені архітектури програмного забезпечення є важливими. Шаблони проектування [3] – це «мікроархітектури», тоді як архітектури програмного забезпечення – це більш грубозернисті проекти. Шаблон проектування описує рішення повторюваної проблеми. Шаблони утворюють більші цілісні структури, як мови шаблонів, щоб забезпечити керівництво для вирішення складних проблем. Шаблони виражають розуміння, отримане з практики в проектуванні та створенні програмного забезпечення. Спільнота шаблонів каталогізує корисні фрагменти дизайну та контекст, який керує їх використанням.

Мова опису архітектури — це набір нотацій, мов, стандартів та домовленостей для архітектурних моделей. Обговорюється формалізація таких архітектурних моделей. Архітектурна модель фіксує частину знань про архітектуру окремої системи або сімейства систем у певній області (тобто еталонної архітектури). Обговорюється повторне використання еталонних архітектур у контексті розробки лінійки програмних продуктів.

Формалізація архітектурних моделей

У промисловій практиці архітектури програмного забезпечення зазвичай описуються неформально або напівформально за допомогою діаграм, що використовують прямокутники, кола та лінії разом із супровідним екстом. Текст пояснює діаграми та надає певне обґрунтування для обраної архітектури.

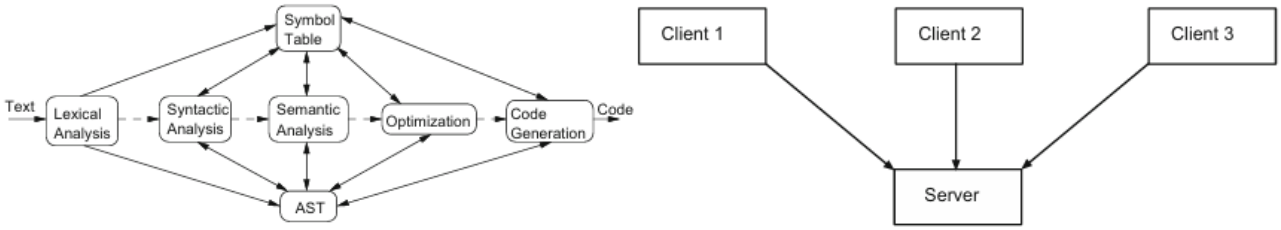

Рисунок 1.1 -Типова конвеєрна архітектура для різних етапів роботи компілятора (ліворуч) та архітектура «клієнт-сервер» для інформаційних систем (праворуч)

Типовими прикладами є *конвеєрні* архітектури для різних фаз компілятора або *клієнт-серверна* архітектура для інформаційних систем, як показано на рис. 1.1. Шоу та Клементс назвали цей метод «боксологією».

Такі рисунки часто дають інтуїтивне уявлення про конструкцію системи, але семантика компонентів та їхні зв'язки/взаємодії можуть інтерпретуватися різними людьми по-різному (через неформальність). Таким чином, такі описи - були піддані критиці через відсутність (формальної) семантики. Однак вони корисні для комунікації із зацікавленими сторонами та для планування проектів. Ступінь формальності залежить від цільового використання архітектурних моделей.

UML часто використовується для архітектурної документації. Однак UML — як загальна об'єктно-орієнтована мова моделювання — забезпечує лише обмежену підтримку архітектурної документації. Наприклад, їй все ще бракує базових архітектурних концепцій, таких як шари. Формалізації описів

архітектури, розроблених паралельно з розробкою мови [4] . Деякі конкретні переваги формальності в описі архітектури програмного забезпечення можна підсумувати наступним чином:

- Архітектури програмного забезпечення стають доступними для аналізу та оцінки [5] . Це допомагає оцінити архітектури та допомогти у виборі архітектурних варіацій як рішень конкретних проблем.
- Архітектури програмного забезпечення можуть бути основою для *повторного використання дизайну* за умови, що окремі елементи архітектурних описів визначені незалежно та точно. Архітектури, що підлягають повторному використанню, надають дизайнерам *план* розробки, допомагаючи їм уникати типових помилок проектування.
- Архітектури програмного забезпечення підтримують покращене розуміння програми як основу для еволюції системи, якщо її специфікація добре зрозуміла: збереження наміру розробника щодо організації системи повинно допомогти розробникам зберегти цілісність дизайну системи.
- Формальність може дозволити створення прототипів для ранньої оцінки дизайну [6] .
- Тестування може бути підкріплене шляхом виведення планів тестування з формальних архітектурних описів.
- Стає можливою належна інструментальна підтримка для проектування та аналізу архітектур програмного забезпечення.

Визнання того, що архітектурний аналіз повинен узгоджувати кілька поглядів, допомогло сформулювати вимоги до формалізму. ADL визначає набір нотацій (наприклад, діаграми, формальні мови, текстові поля природної мови) для кожного погляду, який включає ADL. Моделі архітектури надають один або кілька поглядів на архітектуру. Погляди виділяють певні типи інформації та приховують інші. Приклади відомих архітектурних поглядів включають діаграми керування потоком даних, діаграми переходів станів, моделі даних та діаграми сутностей-зв'язків, структурні діаграми та об'єктно-орієнтовані діаграми ієрархій.

Формалізація архітектурних стилів має на меті дозволити формальні перевірки відповідності між архітектурою та реалізацією, передбачити вплив змін та формально обґрунтувати архітектурний опис системи [7]. Було запропоновано різні підходи до формалізації архітектурних стилів. Формальне порівняння дозволяє детально проаналізувати подібності та відмінності між архітектурними варіаціями.

Лінійки програмних продуктів для повторного використання програмних компонентів

Архітектури програмного забезпечення зазвичай враховують атрибути якості окремих систем. Системи в одній області часто мають подібні архітектури, що відображають концепції області. Еталонні архітектури враховують архітектурну спільність кількох пов'язаних систем, тобто систем в одній області. Еталонні архітектури є центральними для повторного використання, специфічного для певної області, оскільки вони забезпечують основу для створення активів та побудови систем в області. Таким чином, інженерія області дозволяє розробляти лінійку продуктів, яка прагне досягти повторного використання в сімействі систем.

Лінійка програмних продуктів складається з програмних систем, які мають певну спільну функціональність та певну змінну функціональність. Інтерес до лінійок програмних продуктів виник у сфері повторного використання програмного забезпечення, коли розробники та менеджери зрозуміли, що вони можуть отримати набагато більші переваги повторного використання, повторно використовуючи архітектури програмного забезпечення, а не лише окремі компоненти програмного забезпечення. Основна філософія лінійок програмних продуктів полягає в повторному використанні шляхом чітко запланованого використання спільних рис пов'язаних продуктів та належного управління *змінністю* в програмних системах [8].

Розробка продуктів у лінійках програмних продуктів організована у два етапи: інженерія предметної області та інженерія додатків з повторним використанням програмних компонентів [9]. Ідея цього підходу до інженерії

додатків полягає в тому, що інвестиції, необхідні для розробки артефактів, придатних для повторного використання під час інженерії предметної області, переважуються перевагами отримання окремих продуктів під час інженерії додатків.

Референтні архітектури відіграють важливу роль в інженерії предметних областей. *Інженерія предметних областей* – це діяльність зі створення компонентів повторного використання, за допомогою якої вирішується систематичне створення моделей та архітектур предметних областей. Цілі інженерії предметних областей на підтримку *прикладної інженерії* , яка використовує моделі предметної області та еталонні архітектури для побудови конкретних систем, як показано на рис. 1.2 [10]. Модель предметної області характеризує *простір проблем,* тоді як *еталонна архітектура* звертається до *простору рішень* в предметній інженерії. Акцент робиться на повторному використанні та лінійках продуктів.

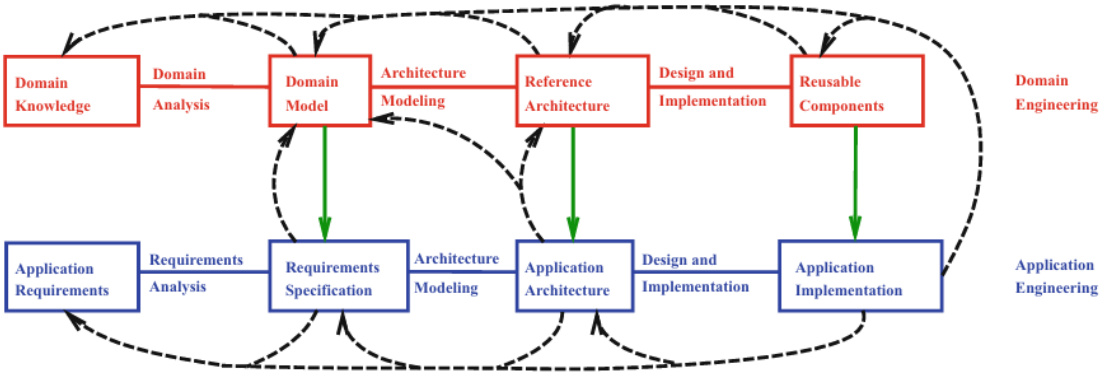

Рисунок 1.2 - Доменна та прикладна інженерія для лінійок програмних продуктів на основі компонентів.

У рамках прикладної інженерії програмні системи розробляються на основі компонентів, придатних для повторного використання, створених у процесі доменної інженерії. Як показано пунктирними стрілками, можливий багатоаспектний зворотний зв’язок.

1.2 Сучасні підходи до побудови архітектур із використанням баз даних

Як обговорювалося в попередньому підрозділі, раніше акцент робився переважно на загальних архітектурних стилях, таких як архітектури типу «труба-фільтр». Однак процес розробки доменно-орієнтованої архітектури програмного забезпечення (DSSA) був запроваджений на початку 1990-х років для сприяння чіткому розмежуванню між вимогами предметної області та вимогами програми. Доменно-орієнтована архітектура програмного забезпечення складається з моделі предметної області та еталонної архітектури. DSSA була запропонована для таких областей, як авіоніка.

Тим часом, для багатьох областей та застосувань створюються різні архітектури. Там, де ще не існує повних архітектурних рішень, часткові, безумовно, існують у вигляді каталогів архітектурних шаблонів, які допомагають вирішити багато проблем та досягти різних атрибутів якості. Різні предметно-орієнтовані архітектури з'явилися, зокрема, з промислової практики. Прикладами є архітектури сховищ даних для бізнес-аналітики та, останнім часом, мікросервісні архітектури для інтернет-сервісів. Як приклад, ми детальніше розглянемо нещодавні мікросервісні архітектури . Багато сучасних підходів до архітектури зосереджені на вимогах до якості.

Приклад: Мікросервісні архітектури

Мікросервіси – це архітектурний стиль програмного забезпечення, якому зараз приділяється велика увага як у промисловості, так і в академічних колах. Особливо так звані системи масштабу Інтернету використовують їх для задоволення своїх величезних вимог до масштабованості та швидкого надання нових функцій своїм користувачам. Як випливає з назви, сервіси є будівельними блоками та основним засобом модуляризації в архітектурах мікросервісів. Сервіси працюють в окремих контекстах процесів і можуть бути окремо розгорнуті, замінені та виведені з експлуатації. Сервіси будуються навколо бізнес-можливостей міжфункціональними командами, які відповідають

за кожен аспект сервісу, від розробки до продуктивної експлуатації.

Традиційно, інтеграція інформаційних систем спрямована на досягнення високої узгодженості даних між різнорідними джерелами інформації [11]. Однак великою проблемою інтегрованих баз даних є обмежена горизонтальна масштабованість управління транзакційними базами даних [12] . Однією з цілей мікросервісних архітектур є подолання обмеженої масштабованості таких монолітних архітектур [13]. Система має мікросервісну архітектуру, коли вона складається з багатьох взаємодіючих мікросервісів, зазвичай без централізованого керування. Мікросервіси побудовані навколо бізнес-можливостей і передбачають повноцінну реалізацію програмного забезпечення для цієї бізнес-сфери, включаючи окремі сховища даних.

Мікросервісні архітектури надають невеликі сервіси, які можна розгортати та масштабувати незалежно один від одного, і можуть використовувати різні стеки проміжного програмного забезпечення для їх реалізації. Мікросервісні архітектури мають на меті подолати недоліки - монолітних архітектур, де вся логіка та дані програми об'єднані в одному розгортаному блоці.

Рекомендується вертикальна декомпозиція на автономні системи (scs-architecture.org) вздовж бізнес-сервісів. Окрім масштабованості, відповідна модульна структура підтримує розуміння програми, стійкість (запобігання поширенню помилок) та автономні команди з добрим знанням своєї вертикальної області. Мікросервісні архітектури сприяють масштабованості, а також гнучкості та надійності [14].

Приклад вертикальної декомпозиції системи електронної комерції на автономні сервіси проілюстровано на рис. 1.3. Вертикальний мікросервіс – це частина платформи, яка відповідає за єдиний обмежений контекст у бізнес-домені [15] . Вертикалі можуть бути такими ж малими, як мікросервіс, але здебільшого вони є більш грубозернистими. Компроміс між багатьма малими компонентами та кількома великими компонентами необхідно враховувати під час проектування сервісів та систем [16]. Мікросервіси повинні дотримуватися

принципу «Спільного нічого»: вони не діляться станом, компонентами інфраструктури, базою даних чи іншими спільними ресурсами. Великими перевагами архітектур без спільного використання є горизонтальна масштабованість та покращена відмовостійкість. Причина цього очевидна: якщо два компоненти нічого не діляться, вони, очевидно, не можуть негативно впливати один на одного. Невеликі сервіси легше розгортати, і, оскільки вони автономні, менш схильні до системних збоїв, якщо вийдуть з ладу.

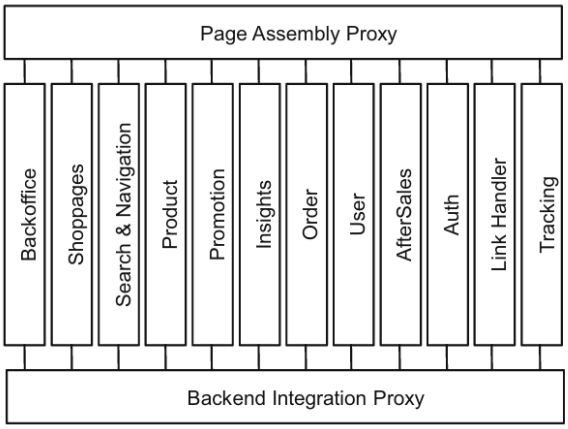

Рисунок 1.3 - Приклад вертикального розбиття системи електронної комерції на автономні мікросервіси

Відомий закон Конвея стверджує, що організації, які проектують системи, зобов'язані створювати проекти, що є копіями комунікаційних структур цих організацій [17] . Якщо організаційну структуру розкласти вертикально та відповідно до структури мікросервісів на міжфункціональні команди розробників, це дозволить масштабувати можливості розробки відповідно до змінних бізнес-вимог. Команди розробників повинні бути дуже незалежними, маючи членів усіх ролей та навичок, необхідних для створення та підтримки їхніх мікросервісів. Мікросервіси підсилюють модульну структуру, що особливо важливо для великих команд. Розділення команд так само актуальне, як і розділення програмних модулів.

Будучи високорозподіленою архітектурою, мікросервіси особливо схильні до часткових збоїв. Тому мікросервіси повинні бути розроблені таким

чином, щоб коректно справлятися з недоступністю необхідних сервісів, щоб запобігти каскадним збоям. Для цієї мети з'явилося кілька шаблонів [18], таких як шаблон вимикача. У цьому шаблоні залежності, такі як інші сервіси, обгорнуті в так званий об'єкт вимикача, який служить проксі-сервером для залежності та контролює її доступність. Якщо залежність стає недоступною або не реагує, вимикач спрацьовує та вживає відповідних заходів, таких як негайне повернення кешованих даних або значень за замовчуванням. Через деякий час вимикач може перевірити, чи залежність знову доступна, і якщо так, повернутися до режиму проксі-сервера.

Децентралізація відповідальності за дані між мікросервісами має наслідки для управління оновленнями. Традиційний підхід до обробки оновлень полягає у використанні транзакцій для гарантування узгодженості під час оновлення кількох ресурсів. Цей підхід часто використовується в межах монолітів. Використання транзакцій таким чином допомагає забезпечити узгодженість, але нав'язує значну зв'язність, що є проблематичним для розподілених сервісів. Розподілені транзакції, як відомо, складні в реалізації, і як наслідок, архітектури мікросервісів наголошують на безтранзакційній координації між сервісами, з явним визнанням того, що узгодженість може бути лише кінцевою узгодженістю, а проблеми вирішуються шляхом компенсації операцій.

Однак, майте на увазі, що мікросервісні архітектури також мають свої витрати. Підтримка узгодженості, моніторингу, тривоги та відмовостійкості є складною для розподіленої системи, а це означає, що вам доведеться керувати набагато складнішою системою, ніж у монолітних архітектурах. Вам потрібна зріла операційна команда для управління великою кількістю сервісів, які часто перерозгортаються.

Зосередьтеся на вимогах до якості

Вимоги до якості є найважливішими вимогами до архітектурної роботи.

Вивчення архітектури програмного забезпечення визнає залежність між архітектурою та атрибутами якості програмної системи, такими як

продуктивність, модифікація та безпека. Архітектурний аналіз та оцінка виникли, пов'язуючи атрибути якості та рішення щодо архітектурного проектування [19]. Для аналізу атрибутів якості використовуються моделі архітектури програмного забезпечення, щоб проаналізувати, чи може система відповідати своїм нефункціональним вимогам [20]. Цілями архітектурних оцінок є оцінки впливу архітектурних рішень на атрибути якості програмного забезпечення. Наприклад, методи оцінки архітектури програмного забезпечення на основі сценаріїв зазвичай оцінюють ремонтопридатність [21] . Архітектурна оцінка для проектування програмного забезпечення допомагає зрозуміти наслідки проектних рішень, дозволяє приймати обґрунтовані проектні рішення, допомагає виявляти компроміси, допомагає перевіряти відповідність та підтримує управління ризиками.

Якість може бути забезпечена за допомогою модельного або вимірювального підходу. Модельний аналіз якості надає інформацію на ранній стадії, тобто до впровадження системи. Такий підхід може виявити проблеми на ранній стадії та зменшити витрати на переробку. Однак розробка відповідних моделей вимагає додаткових зусиль та часу. Підходи до якості, засновані на вимірюваннях, виконують реальні вимірювання. Однак це застосовується лише тоді, коли система впроваджена, але може надати реальні дані.

Нефункціональні атрибути, такі як масштабованість та відмовостійкість для високої доступності, враховуються мікросервісними архітектурами. Наслідком використання мікросервісів є те, що програми повинні бути розроблені таким чином, щоб вони могли переносити збої окремих служб. Оскільки служби можуть вийти з ладу в будь-який час, важливо мати можливість швидко виявляти збої та, якщо можливо, автоматично відновлювати служби. Таким чином, мікросервісні програми приділяють велику увагу моніторингу програми в режимі реального часу, перевіряючи як технічні показники (наприклад, скільки запитів за секунду отримує база даних), так і бізнес-метрики (наприклад, скільки замовлень отримується за хвилину). Моніторинг може забезпечити систему раннього попередження про щось, що

йде не так, що спонукає команди розробників до подальших дій.

Якість не виникає сама по собі. Про неї потрібно думати та ретельно враховувати. Якщо не звертати уваги на якості системи, їх може бути важко досягти в останній момент.

1.3 Інтеграція баз даних у гнучкі методології розробки програмного забезпечення

Напруженість між Agile-спільнотою та архітектурною спільнотою все ще досить висока. Форда часто цитують за його твердження «Архітектура — це те, що важко змінити пізніше» та «Відкладаючи важливі архітектурні та дизайнерські рішення до останнього відповідального моменту, ви можете запобігти непотрібній складності, яка підриває ваші програмні проекти» [22]. Однак, архітектуру програмного забезпечення слід визнати ключовою основою гнучкої розробки програмного забезпечення, незважаючи на те, що її часто ігнорує гнучка спільнота, яка прозвала її BUFD (велике початкове проектування).

Тим часом, спільнота Agile спостерігає ренесанс з інноваціями в архітектурі програмного забезпечення: організації визнали, що «хмара» є фактичною платформою майбутнього, а переваги та гнучкість, які вона приносить, започаткували «ренесанс» в архітектурі програмного забезпечення. Одноразова інфраструктура хмари дозволила створити першу «хмарну нативну» архітектуру, а саме мікросервіси. Безперервна доставка, техніка, яка радикально змінює розвиток технологічних підприємств, посилює вплив хмари як архітектури. ThoughtWorks очікує, що архітектурні інновації продовжаться, а такі тенденції, як контейнеризація та програмно-визначені мережі, забезпечать ще більше технічних опцій та можливостей.

Як зазначено, сприяння гнучкості бізнесу вимагає обґрунтованого проектування архітектури. Питання полягає в тому, як найкраще досягти гнучкості за допомогою архітектури. Високомодульні архітектури, що

дозволяють швидке експериментування, є критично важливими для успішної інтеграції архітектурної роботи в гнучку розробку програмного забезпечення. Легкі, гнучкі методи є перспективними, оскільки вони дозволяють командам швидко навчатися, швидко зазнавати невдач, швидко змінюватися та ефективно спілкуватися. Ці фактори є важливими для самоорганізованих команд, що, у свою чергу, дозволяє створювати кращі проекти [23].

У підрозділі представлено передбачувану роль власника архітектури в Agile-командах, а потім у розділі обговорюється взаємозв'язок між розробкою програмного забезпечення та експлуатацією. Досягнення надійності за допомогою гнучкої розробки, адаптивності середовища виконання та підтримки актуальності знань про архітектуру для довговічних програмних систем обговорюються в розділах.

Інтеграція власників архітектури в Agile-команди

Розгляд архітектури як фази ігнорує її фундаментальну роль у розробці програмного забезпечення. Робота над архітектурою повинна бути інтегрована з усіма видами діяльності з розробки програмного забезпечення. За допомогою гнучкої розробки у вас може бути роль «Власника архітектури», який повинен залучати всю команду до прийняття обґрунтованих та прийнятних архітектурних рішень. Велике питання полягає в тому, скільки архітектурних та дизайнерських рішень потрібно прийняти заздалегідь (до Спринту 1)? Перспективним підходом є створення *бачення архітектури* у «нульовому» спринті. Вам потрібно прийняти обмеження, визначити та просувати бажані атрибути якості, призначити функціональні обов'язки елементам та керувати дизайном за допомогою шаблонів. Спробуйте встановити чітке бачення архітектури.

За допомогою гнучкої розробки програмного забезпечення ви можете проектувати архітектуру за допомогою історій та варіантів використання, тобто керуватися вимогами. Сценарії варіантів використання описують конкретні взаємодії між користувачем прикладної системи та самою системою. Ми повинні протестувати архітектуру за сценаріями, пов'язаними з вимогами до атрибутів якості системи, такими як продуктивність. Ви можете додати елементи журналу

невиконання технічних вимог та роботи, пов'язаної з якістю архітектури. Критерії прийнятності, пов'язані з якістю, можна додати до історій користувачів. Ви повинні контролювати якості системи, наприклад, за допомогою операційної панелі інструментів. Не хвилюйтеся про те, щоб ваша архітектура була правильною в перший день. Моделюйте та впроваджуйте поступово . Розставте пріоритети функцій архітектури та зменшуйте ключові ризики.

Валідація архітектури заохочує комунікацію між зацікавленими сторонами проекту. Постійна перевірка архітектури показує нам, чи маємо ми все ще правильну структуру команди. Найкращим підходом є раннє виправлення дефектів. Аспекти комунікації та співпраці в архітектурі так само важливі, як і її розробка. Оцінка архітектури значно допомагає у співпраці та комунікації. Власник архітектури повинен забезпечувати лідерство архітектора у співпраці та допомагати членам команди покращувати свої можливості в розумінні архітектурних принципів та пов'язаних з ними компромісів. Власник архітектури повинен бути інтегрований у щоденну розробку та звертати увагу на деталі.

Деякі рішення занадто важливі, щоб відкладати їх на *останній* відповідальний момент, тому оберіть *найвідповідальніший* момент. Оберіть архітектуру, перш ніж вона обере вас, як це може статися з *емерджентними* архітектурами.

Інтеграція розробки програмного забезпечення та операцій: DevOps

Рух DevOps продовжує те, що розпочав Agile. Рух DevOps має на меті покращити комунікацію, співпрацю та інтеграцію між розробниками програмного забезпечення (Dev) та фахівцями з IT-операцій (Ops). Автоматизація є ключем до успіху DevOps: автоматизоване створення систем з репозиторіїв керування версіями; автоматизоване виконання модульних тестів, інтеграційних та системних тестів; автоматизоване розгортання в тестовому та виробничому середовищах; включаючи тести продуктивності. DevOps — це набір практик, призначених для скорочення часу між внесенням змін до системи та їх введенням у нормальне виробництво, забезпечуючи при цьому високу якість [24] .

Конвеєр розгортання – це місце, де перетинаються архітектурні та процесні аспекти DevOps [2] . Архітектурний вибір повинен сприяти безперервній доставці. Мікросервісна архітектура розроблена для мінімізації - потреб координації та забезпечення незалежного розгортання. Кількома одночасними версіями можна керувати за допомогою перемикачів функцій та зворотної/прямої сумісності. Перемикачі функцій підтримують відкат, canary-тестування та A/B-тестування. Мікросервіси використовують безперервну інтеграцію та безперервне розгортання для просування DevOps. Мікросервісні архітектури та DevOps виграють один від одного [26] .

Досягнення надійності за допомогою гнучкої розробки програмного забезпечення

Для великих програмних систем основна складність автоматизованого регресійного тестування пов'язана з високими обчислювальними витратами на тести. Щоб забезпечити високе покриття коду, необхідно виконати потенційно експоненціальний набір конфігурацій тестів. Рішенням цієї проблеми може бути належна модуляризація програмного забезпечення, щоб компоненти програмного забезпечення стали тестованими окремо. Підходи до модуляризації, такі як мікросервіси, також можуть сприяти автоматизованому регресійному тестуванню великих програмних систем. Для задоволення потреб забезпечення якості необхідні інструменти та автоматизація. Поєднання модульних архітектур мікросервісів з автоматизованим забезпеченням якості дозволяє зберегти надійність за допомогою гнучкої розробки програмного забезпечення [27].

Багато Agile-команд зосереджуються лише на функціональному тестуванні, але є набагато більше для тестування, наприклад, продуктивність. Тести продуктивності можна автоматизувати в умовах безперервної інтеграції за допомогою регресійного бенчмаркінгу. Сильна модель тестування на основі архітектури, підкріплена формальним мисленням та відповідним інструментарієм, може мати значний економічний вплив на розробку програмних систем.

Використання архітектурних моделей для адаптивності середовища виконання

Масштабовані системи повинні дозволяти автоматично реагувати на зміну робочих навантажень за допомогою еластичного управління потужністю, як це пропонують хмарні інфраструктури [28]. Завдяки мікросервісним архітектурам можна динамічно реплікувати ці мікросервіси в хмарні інфраструктури, які перебувають під великим навантаженням. Немає потреби масштабувати всю систему, як це було б потрібно для монолітної системи.

Відмовостійкість призначена для забезпечення надання правильних послуг за наявності активних несправностей. Вона реалізується шляхом виявлення помилок та подальшого відновлення системи. Виявлення помилок виявляє помилковий стан системи. Подальше відновлення системи перетворює стан системи, що містить одну або декілька помилок, у стан без виявлених помилок. Можливим рішенням є динамічна адаптація. У разі помилок динамічна адаптація може забезпечити досягнення найкращої можливої функціональності системи та збереження критичних функцій (живучість). Реалізація живучості замість відмовостійкості забезпечує величезний потенціал для економії коштів, забезпечення безпеки системи та досягнення прийнятної доступності. Вищезгадана схема вимикача забезпечує такі властивості.

Щоб забезпечити таку адаптивність середовища виконання, нам потрібна інформація про архітектуру запущеної системи: опис архітектури у вигляді виконуваного файлу, який також називають models@runtime.

Підтримка знань про архітектуру в актуальному стані для довговічних програмних систем

Найбільші витрати в розробці програмного забезпечення, як правило, пов'язані з обслуговуванням системи та додаванням нових функцій. Якщо провести оцінку архітектури на ранній стадії, вона може зменшити ці витрати, виявивши наслідки для проекту [29] . Це, у свою чергу, може призвести до раннього виявлення помилок та до найбільш передбачуваних та економічно ефективних модифікацій системи протягом її життєвого циклу. Архітектура

програмного забезпечення фіксує та зберігає наміри розробників щодо структури та поведінки системи, тим самим забезпечуючи захист від занепаду проекту в міру старіння системи [30]. Якість та довговічність системи, що залежить від програмного забезпечення, значною мірою визначаються її архітектурою. Слід уникати технічного боргу. Неправильні архітектурні рішення є основним фактором, що сприяє виникненню технічного боргу.

Знання про архітектуру часто втрачаються, коли ми переходимо до коду. Найкраща архітектура нічого не варта, якщо код їй не відповідає. Для такого довговічного програмного забезпечення важливо підтримувати документацію та знання про програмне забезпечення в актуальному стані. Без відповідності між документацією про архітектуру та кодом документація про архітектуру стає неактуальною. Ми можемо встановити відповідність шляхом побудови — за допомогою модельно-орієнтованої програмної інженерії — або шляхом зворотного проектування (статичного або динамічного аналізу артефакту для визначення його архітектури). Розуміння програмної системи за допомогою моделей зворотного проектування архітектури є корисним у цьому контексті [31]. Успішні, великі системи мають довгий термін служби. Вони повинні розвиватися, щоб відповідати мінливим вимогам. Існуючі системи, які необхідно підтримувати, іноді називають застарілими системами. Потрібна модернізація застарілих програмних систем [32]. Коли програмна система розвивається, в ідеалі спочатку змінюється її прескриптивна архітектура. Однак на практиці система, а отже, і її описова архітектура, часто модифікується безпосередньо. Це відбувається через сприйняття коротких термінів,
необхідність оптимізації коду, неадекватну підтримку інструментів тощо.

Розуміння взаємозв'язку між архітектурними рішеннями та атрибутами якості системи показує, що оцінка архітектури програмного забезпечення є корисною стратегією зниження ризиків. Рішення є основним результатом (гнучої) архітектурної роботи, водночас зберігаючи відкладені архітектурні проблеми. Управління витратами та ризиками є основною бізнес-метою та

пріоритетними факторами для власників архітектури. Для довговічності рішення слід приймати в найвідповідальніший момент, а не в останній.

Архітектури програмного забезпечення є важливими для розробки та підтримки великомасштабних, довговічних програмних систем. Розуміння цих систем покращується завдяки абстрактному погляду на систему високого рівня. Архітектура підтримує повторне використання компонентів та фреймворків. Архітектура робить очікувану еволюцію явною та розділяє функціональність та механізми підключення компонентів і сервісів, щоб вони могли розвиватися індивідуально. Повна автоматизація забезпечення якості та розгортання програмного забезпечення дозволяє виявляти несправності та помилки на ранній стадії, тим самим скорочуючи час виправлення як під час розробки, так і під час експлуатації.

Знаходження правильного балансу для роботи з архітектурою – це мистецтво гнучкого володіння архітектурою. Ми можемо очікувати поєднання роботи з архітектурою та гнучких практик розробки програмного забезпечення. Власники архітектури повинні приймати рішення у *найвідповідальніший* момент, а не в *останній* .

1.4 Висновок по розділу

У першому розділі розглянуто теоретичні основи використання баз даних у програмних системах та визначено їх ключову роль в архітектурі сучасного програмного забезпечення. Проаналізовано основні підходи до побудови архітектурних рішень із використанням баз даних та особливості їх інтеграції у процеси розробки за гнучкими методологіями. Встановлено, що бази даних є центральним компонентом інформаційних систем, забезпечуючи надійне зберігання, обробку та доступ до даних, що безпосередньо впливає на ефективність і масштабованість програмних продуктів.

РОЗДІЛ 2. МЕТОДИ ТА ІНСТРУМЕНТИ ВИКОРИСТАННЯ БАЗ ДАНИХ В АРХІТЕКТУРІ СИСТЕМ

2.1 Архітектурні підходи до організації взаємодії з базами даних

Архітектура конкретного програмного застосунку враховує нефункціональні характеристики, такі як надійність, зручність використання, масштабованість, продуктивність, сумісність, портативність, адаптивність та шардування даних. Завжди існують компроміси між набором атрибутів якості, і архітектор програмного забезпечення стикається зі складним завданням їх балансування. Системи великих даних [33] є внутрішньо розподіленими. Труднощі з доступністю та узгодженістю даних виникають через шардування даних та реплікацію у величезних системах даних. Через зростаюче розширення застосунків для обробки даних технології баз даних зазнали суттєвих змін . Протягом понад десятиліття бази даних NoSQL зростали в геометричній прогресії, хоча класична автоматизація баз даних зберігалася. Традиційні макети змушують використовувати жорстку структуру схеми, що призводить до неясності масштабування та перешкоджає модифікації даних між кластерами. На противагу цьому, бази даних NoSQL підтримують прості прототипи. Основні властивості проектів баз даних NoSQL включають:

- Структура без схеми
- Дозвіл на ефективне та динамічне зростання представлень даних
- Горизонтальне масштабування шляхом колекцій реплікації даних та шардингу на масивних кластерах.

За останні роки численні організації накопичили величезні обсяги даних, які реляційні бази даних не можуть ефективно обробляти. За останні чотири десятиліття спостерігається величезне зростання використання реляційних баз даних. Вони дотримуються атрибута ACID (доступність, узгодженість, ізоляція та довговічність) та розроблені для структурованих даних. Хоча «великі дані» включають інструменти та технології, які керують величезними обсягами даних

у будь-якому масштабі, ці інструменти та технології є масштабованими. Великі дані включають 5V (об'єм, швидкість, різноманітність, достовірність та цінність) та величезну кількість неструктурованих даних з різноманітною природою. Численні фреймворки, включаючи Hadoop/MapReduce, Spark, Flink та Samza, використовуються для обробки великих даних [34].

Тема продуктивності та оптимізації SQL-запитів для корпоративних, виробничих, паралельних баз даних та великих даних [2] отримала підвищену увагу в останні роки. Неефективні та неоптимізовані запити можуть споживати системні та серверні ресурси, що призводить до блокування бази даних та проблем втрати даних. Інформаційний інтелектуальний аналіз передбачає вилучення фактів та логічної кореляційної структури з початкового дельта-множини, на відміну від самої інформації. Оптимізація запитів стосується вибору оптимальної стратегії виконання запитів з мінімальними витратами та споживанням системних ресурсів. Алгоритми інтелектуального аналізу даних виконують глибокі та розширені запити до бази даних для вилучення шаблонів та знань з комплексних даних [5]. Загальнопромислові методології, такі як XML та об'єктні бази даних, ніколи не досягали такого ж рівня популярності, як технологія RDBMS.

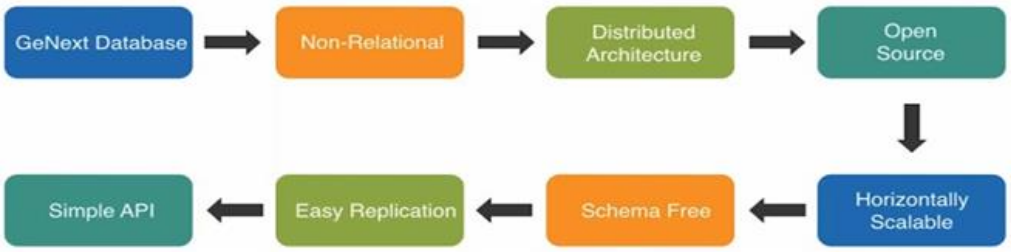

Рисунок 2.1 - NoSQL — основні особливості.

Протягом останнього десятиліття науковці та онлайн-продавці часто ставили під сумнів універсальний характер технології обробки даних. Такий підхід призвів до розробки нової альтернативної системи баз даних, відомої як NoSQL, що розшифровується як «не тільки SQL». NoSQL описує використання

веб-розробниками нереляційних баз даних [I]. У 1998 році термін NoSQL[e] був вперше використаний, і конференція з нереляційних баз даних у Сан-Франциско привернула до нього більше уваги. Рисунок. У розділі 1 описано ключові аспекти NoSQL баз даних.

Таблиця 2.1

Теорія CAP.

Узгодженість (Consistency)	**Доступність (Availability)**	**Стійкість до розділення (Partition Tolerance)**
● **Узгодженість** означає, що дані в базі даних залишаються цілісними після виконання операції.	● **Доступність** означає, що система не матиме простоїв (гарантується 100% часу безперебійної роботи).	● **Стійкість до розділення** означає, що система продовжує функціонувати, навіть якщо зв'язок між серверами ненадійний.
● Наприклад, після операції оновлення всі клієнти бачать однакові дані.	● Кожен вузол (якщо він не вийшов з ладу) завжди виконує запит.	● Сервери можуть бути розділені на кілька груп, які не можуть спілкуватися одна з одною.

Теоретично, неможливо виконати всі три вимоги. Тому CAP забезпечує основну потребу для розподіленої системи, щоб вона відповідала двом із трьох елементів. Отже, всі сучасні NoSQL бази даних підтримують різні комбінації C, A та P теорії CAP, як описано на рисунку 2.2.

Нижче наведено переваги NoSQL баз даних:

- Обсяг: Дані в стані спокою — терабайти до ексабайтів існуючих даних для обробки.
- Швидкість: Дані в русі — потокові дані, час відповіді від мілісекунд до секунд.
- Мінливість: дані в багатьох формах — структуровані, неструктуровані, текстові тощо.
- Достовірність: дані під сумнівом — невизначеність через затримку, обман, неоднозначності тощо.
- Не побудовано на таблицях і не використовує SQL для маніпулювання даними.

- Схема включає парі ключ-значення, документ, стовпцеву схему, графік тощо.
- Альтернатива традиційним реляційним базам даних.
- База даних для обробки неструктурованих, невпорядкованих та непередбачуваних даних.
- Корисно для роботи з великими наборами розподілених даних.

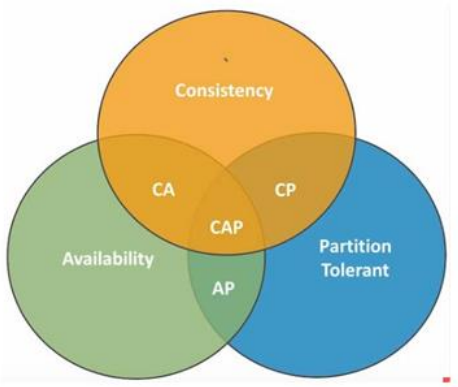

Рисунок 2.2 - Комбінація CAP

NoSQL-бази даних відрізняються від реляційних баз даних і мають власні моделі та архітектури. Під час зберігання NoSQL-баз даних у хмарі слід бути обережним через різноманітність цих баз даних та необхідність забезпечення сумісності, портативності та заходів безпеки. Хмарні постачальники послуг (CSP) [36] пропонують масштабованість , доступність та конфіденційність, але важливо шифрувати конфіденційні дані користувачів перед наданням доступу CSP. Це може бути складно через різноманітний характер NoSQL-баз даних. База даних як послуга (DBaaS) [35] – це платформа tloud, яка перетворює традиційні архітектури на хмарні.

Основний внесок цієї SLR arc IHC наступний:

- Цей односторонній документ пов'язаний з оцінкою архітектури баз даних SQL та NoSQL, можливостями масштабування та аналізом продуктивності, зокрема Oracle RDBMS та NoSQL Document Database (MongoDB). Крім того, досліджується переміщення даних між різними базами

даних на кількох хмарних платформах.

- Для досягнення дослідницьких цілей, згаданих раніше, було проаналізовано загалом 142 дослідження.
- У цій роботі визначено прогалини в дослідженнях пов'язаних архітектур та їх причини.

Поширення великих даних призвело до потреби в масштабованих системах, які можуть ефективно обробляти величезні обсяги даних. Реляційні бази даних, засновані на SQL, можуть певною мірою керувати структурованими, напівструктурованими та неструктурованими даними, але мають обмеження щодо масштабованості. З іншого боку, NoSQL бази даних, які відповідають властивості BASE та можуть розширювати свою ємність сховища горизонтально, краще підходять для керування величезними обсягами даних та адаптації до змін типу та структури даних.

Існує чотири типи NoSQL баз даних — бази даних типу «ключ-значення», документи, стовпці та графи — кожна з яких має свої особливості та застосування. Наприклад, графова база даних NoSQL зберігає інформацію у вузлах, а не в таблицях, і зберігає асоціації (з'єднання) між вузлами, що вимагає постійного часу виконання. Це дає їй перевагу над базами даних SQL під час обробки високо взаємопов'язаних даних.

MongoDB є прикладом NoSQL бази даних, яка ефективно керує величезними обсягами даних завдяки своїм чудовим характеристикам масштабованості. Однак перетворення з OLTP- баз даних до MongoDB може спричинити проблеми, такі як унікальні індекси, складені ключі, невідповідність даних та дублювання даних через складність їхньої схеми.

Ще однією важливою проблемою при передачі даних у хмарне сховище є захист конфіденційної інформації користувачів. Поточні проекти постачальників хмарних послуг (CSP) розробляються без урахування питань сумісності та портативності, що ускладнює пропонування та створення єдиного хмарного рішення для моделей NoSQL.

Нарешті, у цьому SLR детально розглядаються найсучасніші методи та

політики безпеки для NoSQL баз даних.

Метод

Цей SLR відповідає рекомендаціям PRISMA та має на меті синтезувати високоякісні первинні дослідження на основі доказів, пов'язаних з конкретними дослідницькими питаннями. SLR широко використовуються в дослідженнях у галузі програмної інженерії, а наше дослідження зосереджене на оцінці архітектури баз даних SQL та NoSQL та оцінці продуктивності, а також на переносимості даних та сумісності між кількома хмарними платформами. Характеристики, що відрізняють наше дослідження від існуючих, - це системний процес збору даних, вичерпний перелік охоплених досліджень, цілеспрямований обсяг дослідження та детальна класифікація та аналіз вибраних досліджень.

2.2 Управління ресурсами та транзакціями у розподілених інформаційних системах

Розподілені інформаційні системи є відображенням сучасних розподілених організаці– бізнесу, управління чи сфери послуг. Сьогодні бізнес-процеси розглядаються як центральна концепція організації способу ведення бізнесу. Тому проектування бізнес-процесів вважається серйозним завданням. Вимоги до розподілених інформаційних систем повинні бути головним результатом проектування.

Обмін інформацією є життєво важливою частиною бізнес-процесів. Бізнес-процеси працюють на географічних відстанях, часто у світовому масштабі, та використовують корпоративну пам'ять у вигляді величезних сховищ даних. Отже, ми обмежуємося поглядом на розподілені інформаційні системи, який зосереджується на тих функціях, що підтримують обмін інформацією. Оскільки обмін інформацією має *просторовий* та *часовий* аспекти, очікується, що розподілені інформаційні системи долають часові та просторові відстані. Це дає перший – тривіальний – архітектурний критерій – поділ цих двох аспектів на системи передачі даних та системи управління базами даних.

Більш абстрактно, з точки зору бізнес-процесу, інформаційну систему можна розглядати як набір інформаційних *ресурсів* разом із відповідним *менеджером (ресурсів*) для кожного. Просторовий обмін забезпечується менеджери, з локальними та глобальними мережами як ресурсами. Часовий обмін забезпечується менеджерами баз даних, з основним та периферійним сховищем як фізичними ресурсами та базами даних як логічними ресурсами. Спектр допомоги, яку менеджер ресурсів пропонує своїм клієнтам, називається його *послугою.* Бізнес-процеси, таким чином, використовують послуги зв'язку та управління базами даних розподіленої інформаційної системи.

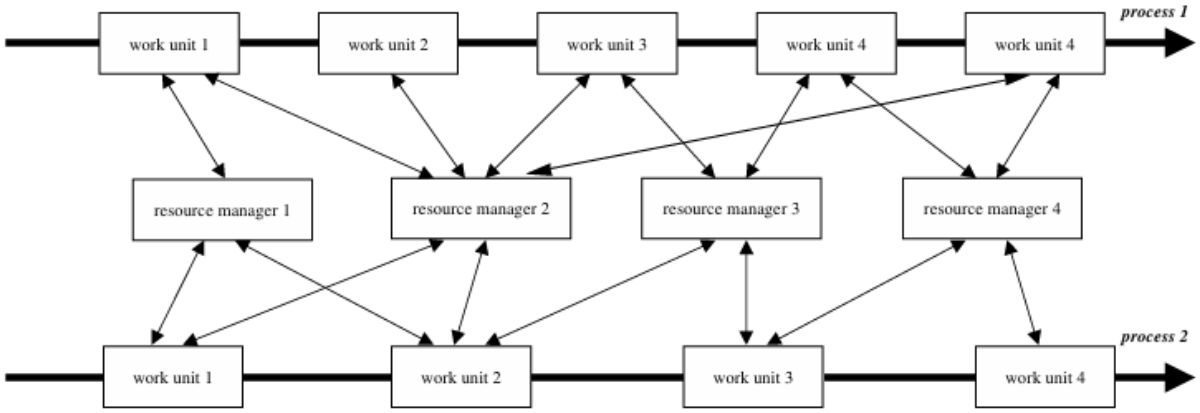

Рисунок 2.3 - Бізнес-процеси та менеджери ресурсів

На рисунку 2.3 показано основну структуру. Бізнес-процес складається з набору робочих одиниць. Кожна одиниця користується послугами одного або кількох менеджерів ресурсів, і різні одиниці можуть звертатися до одного й того ж менеджера. Для спрощення припустимо, що кожен бізнес-процес повністю впорядкований. Таким чином, у межах бізнес-процесу ресурси розподіляються в часовому порядку. Однак загалом велика кількість бізнес- процесів – в межах одного підприємства або між різними підприємствами – відбувається паралельно.

З точки зору послуг, ми називаємо менеджерів ресурсів *постачальниками послуг* , а робочі одиниці – *клієнтами послуг.*

Функції сервісу

З абстрактної точки зору, клієнт і постачальник укладають контракт, так звану *угоду про надання послуг.* Грубо кажучи, угода стосується двох аспектів: *що* має бути виконано та *наскільки добре* це має бути виконано. Щодо характеристик послуг, що складають перший аспект, постачальник повинен надати *абсолютні* гарантії, тоді як для характеристик другого аспекту можна обговорити *градуйовані гарантії* . Ми називаємо характеристики першого типу *функціональністю послуг* , а другого типу – *якістю послуг.* В ідеалі функціональність послуг повинна відповідати бажаним характеристикам, а якість послуг – обмеженням.

Почнемо з функціональності сервісів розподілених інформаційних систем, їх можна розділити на чотири широкі категорії.

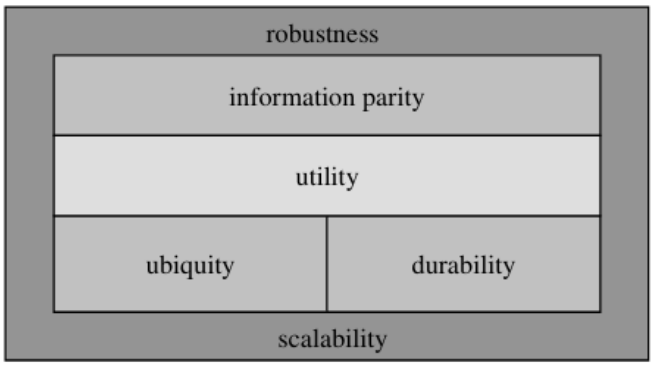

Рисунок 2.4 - Функціональність та якості послуг у розподілених інформаційних системах

1. *Повсюдність.* Обмін даними має бути можливим між будь-якою парою клієнтів у будь-який час у будь-якому місці. Доступ до даних має бути можливим для будь-якого клієнта у будь-який час з будь-якого місця.

2. *Довговічність.* Доступ до збережених даних – якщо вони не перезаписані – має залишатися можливим у будь-який час у необмеженому майбутньому.

3. *Корисність.* Це є raison d'être угоди. Вона описує набір функцій, за допомогою яких клієнт ініціює обмін просторовими та часовими даними.

4. *Інформаційна парність.* Дані несуть інформацію, але самі по собі вони не є інформацією. Для обміну інформацією відправник повинен закодувати свою інформацію як дані, а одержувач реконструює інформацію, інтерпретуючи ці дані. Будь-який обмін повинен забезпечувати, наскільки це можливо, узгодженість інтерпретацій відправника та одержувача, тобто збереження значення. Це вимагає деяких спільних домовленостей, наприклад, формальних рамок для інтерпретації. Ця вимога є більш суворою для часового обміну, ніж для просторового, оскільки перший не надає можливості для прямого спілкування для усунення будь-яких непорозумінь. Натомість, у часовому обміні домовленості мають бути повідомлені постачальнику послуг, щоб забезпечити незмінність інтерпретації під час створення та доступу до даних.

Для розподілених інформаційних систем якість обслуговування має два основні аспекти.

1. *Надійність.* Сервіс повинен залишатися надійним за будь-яких обставин, будь то помилки, перебої, збої, вторгнення, перешкоди. Надійність завжди повинна базуватися на *моделі відмов.*

2. *Масштабованість.* Сервіс повинен витримувати постійне зростання запитів на обслуговування, як для передачі даних, так і для зберігання або пошуку даних.

На рисунку 2.4 показано, як взаємодіють функції сервісу. Повсюдність – це, перш за все, відповідальність за передачу даних та довговічність управління базою даних. Всі інші функції є спільною відповідальністю обох сторін.

З точки зору контракту, архітектурне проектування є паралельним процесом до переговорів щодо контракту. Враховуючи характеристики послуг, системні розробники визначають, як вони впливають один на одного за умов подальших обмежувальних факторів, таких як обмеження фізичних ресурсів. Вони вирішують, як вони можуть бути взаємовигідно використані та, зрештою, чи можна укласти угоду, чи умови необхідно переглянути.

Динаміка послуг

З рис. 2.3 ми спостерігаємо три типи зв'язків в інформаційній системі.

1. *Клієнт-постачальник.* Клієнти надсилають *запити* на обслуговування постачальнику, викликаючи сервісну функцію. Постачальник автономно виконує запит і повертає результат клієнту. Клієнт і постачальник працюють в окремих процесах , зв'язок може бути синхронним або асинхронним.

2. *Клієнт-клієнт.* Завдяки спільному доступу до ресурсів, бізнес-процеси через свої робочі підрозділи можуть взаємодіяти. Якщо взаємодія бажана, оскільки процеси переслідують спільну мету, ми маємо випадок *співпраці.* Якщо вона небажана, ми маємо випадок *конфлікту* між процесами.

3. *Постачальник-постачальник.* Бізнес-процес може залучати послуги низки постачальників. Щоб процес досяг своєї кінцевої мети, залучені постачальники повинні *координувати* свою діяльність.

Уточнення гіпотези

Підрозділ надає нам загальну основу для вираження зовнішніх факторів, що визначають архітектурне проектування. Загалом ми виділили шість ознак, забагато, щоб розглянути їх усі одночасно. Отже, ми уточнюємо нашу гіпотезу до такої, яка припускає, що деякі ознаки мають більший вплив, ніж інші. Основні ознаки використовуються для розробки загальної архітектури. Мірою правильного вибору було б те, що інші ознаки впливають лише на один компонент загальної архітектури або додають до неї один компонент. Ми називаємо цю властивість *ортогональністю проектування.*

У наступних чотирьох розділах гіпотеза буде перевірена ретроспективно для різних рхітектур систем баз даних.

2.3 Еталонні архітектури систем управління базами даних (СУБД)

Наша спеціалізація — системи керування базами даних (СКБД). Щоб позначити цю спеціалізацію, ми використовуємо спеціалізовану термінологію для позначення її функцій.

1. *Модель даних.* Модель даних виражає корисність СУБД. Ця корисність є узагальненою: через величезні інвестиції, що вкладаються в

розробку, СУБД повинна бути здатною підтримувати великий та широкий ринок застосунків. Як така, модель даних надає колекцію примітивних просторів станів та операторів переходів, а також додаткові конструктори, які дозволяють групувати їх у складніші простори станів та процедури переходів відповідно. Більш формально кажучи, модель даних можна порівняти з поліморфною системою типів. Оператори та процедури відповідають службовим функціям, які можуть викликатися клієнтами. Масштабованість має функціональний аналог у конструкторі для динамічних наборів елементів зі структурою записів.

2. *Узгодженість*. Враховуючи стан ділового світу, метою є те, щоб база даних відображала відповідну абстракцію цього стану (міні-світу) в актуальній версії. Таким чином, інформаційна парність уточнюється до поняття обмеження вмісту та еволюції сховища даних чітко визначеним набором станів та переходів між станами, які вважаються значущими. Кожен стан бази даних слід інтерпретувати як певний стан справ у міні-світі. Оновлення бази даних, призначене для відображення певної зміни в міні-світі, дійсно інтерпретується як та сама зміна кожним спостерігачем бази даних. Однак, оскільки СУБД не може передбачити справжній стан міні-світу, було б занадто вимагати збереження значення в цьому ідеальному сенсі. Натомість, доводиться задовольнятися меншими гарантіями. *Статична узгодженість* означає, що поточний стан бази даних завжди визначається з попередньо визначеного набору узгоджених станів бази даних, які вважаються дійсними та однозначними описами можливих станів міні-світу. *Динамічна узгодженість* гарантує, що переходи станів перетворюють узгоджені стани на узгоджені. Щоб забезпечити ці два аспекти, відповідні набори станів та переходів станів необхідно визначити в *схемі бази даних.* Зазвичай схему бази даних можна розглядати як тип бази даних разом з подальшими обмеженнями стану. Поліморфні оператори переходу забезпечують узгодженість, дотримуючись схеми. Загалом кажучи, узгодженість стосується ступеня інформаційної парності між базою даних та міні-світом.

3. *Збереження.* Збереження передбачає збереження даних на енергонезалежному носії, тобто на носії з (принаймні концептуально)

необмеженим терміном служби. Більше того, збереження має бути обмежене тими станами бази даних, які вважаються несуперечливими. Такі стани називаються постійними. Як правило, лише *результат* виконання транзакційної процедури (так званої транзакції (бази даних)) вважається постійним.

4. *Стійкість.* Надійна СУБД повинна бути здатною відновлюватися після різноманітних збоїв, включаючи втрату нестабільного стану системи, несправності обладнання та втрату носія. Усі моделі збоїв для досягнення стійкості повинні базуватися на понятті узгодженості. Однак слід розрізняти, чи транзакція впливає окремо, чи через конфлікт з іншими. У першому випадку збою модель збою призводить до повернення бази даних до попереднього постійного стану або, якщо це виявляється неможливим, до досягнення альтернативного узгодженого стану. У другому випадку модель збою полягає в синхронізації конфліктуючих транзакцій таким чином, щоб результат був постійним як з точки зору окремої транзакції (внутрішня узгодженість), так і сукупності транзакцій (зовнішня узгодженість).

5. *Продуктивність.* Функціональність СУБД повинна масштабуватися для надзвичайно великих обсягів даних та великої кількості транзакцій. Масштабованість проявлятиметься в *часі відгуку* транзакції. Системні адміністратори натомість зосереджуватимуться на *пропускній здатності транзакцій.* Разом ці два поняття називають продуктивністю СУБД.

У класичних централізованих системах управління базами даних *повсюдність* зазвичай ігнорується. Ця тема стає ще важливішою в мережевих системах управління базами даних.

Фізичні та економічні вузькі місця

Як зазначалося, корисність є raison d'être (основою) для СУБД, але так само й довговічність. Більше того, обидва тісно пов'язані: перше передбачає друге. Отже, перший крок полягає в аналізі того, чи впливає якась із решти функцій безпосередньо на одну з них, а отже, опосередковано на іншу. Ми стверджуємо, що функцією, яка має основний вплив, є продуктивність .

Вплив продуктивності зумовлений обмеженнями, що виникають у

фізичних ресурсах. Навіть після десятиліть довговічність все ще майже виключно забезпечується магнітними дисками. Якщо використовувати швидкість процесора як мірило, фізичне вузьке місце – це те, що уповільнює обробку щонайменше на порядок. Найбільшим вузьким місцем, на шість порядків, є затримка доступу, яка складається з руху механічного механізму доступу для досягнення циліндра та затримки обертання, поки потрібний блок даних не з'явиться під головкою читання/запису. За цим вузьким місцем слідує друге вузьке місце, на три порядки величини – пропускна здатність передачі.

Зазначимо, що навіть основна пам'ять вносить один порядок. Комп'ютери намагаються подолати це, розміщуючи дані у швидкому кеш-сховищі. СУБД повинні дотримуватися аналогічної стратегії *розміщення даних* , переміщуючи дані досить рано з периферійної пам'яті до основної пам'яті. Але тепер ми спостерігаємо друге, економічне вузьке місце: ціна за біт для основної пам'яті на два-три порядки вища, ніж для магнітного диска. Таким чином, розміщення даних, яке знаходить відповідний баланс між фізичними та економічними вузькими місцями, має бути одним із принципів, що керують архітектурним проектуванням СУБД.

Основний компроміс: балансування моделі даних та продуктивності

Пропонується, що корисність та продуктивність повинні диктувати стратегію проектування, тобто що вони повинні мати перший пріоритет. Таким чином, ми вводимо два діаметрально протилежні напрямки проектування: низхідний напрямок відображення моделі даних на фізичні ресурси та висхідний напрямок зменшення вузьких місць шляхом поетапного розміщення даних (рис. 2.5). Це складне завдання, яке традиційно вимагає поетапного підходу, щоб розбити простір рішень на менші частини. Результатом є багаторівнева архітектура.

Визначення простору рішень для кожного рівня є першочерговим завданням. Наша гіпотеза полягає в тому, що нам потрібно знайти відповідні абстракції як для корисності, так і для продуктивності, щоб обидві збігалися з метою збалансування потреб. Корисність визначається з точки зору

відображення функцій, а продуктивність – з точки зору критеріїв для «досить раннього» розміщення даних. Рисунок 2.6 ілюструє цей підхід. Зміщення корисності вниз еквівалентно зменшенню виразності моделі даних. Зміщення продуктивності вгору еквівалентно розширенню контексту для визначення того, що означає досить рано.

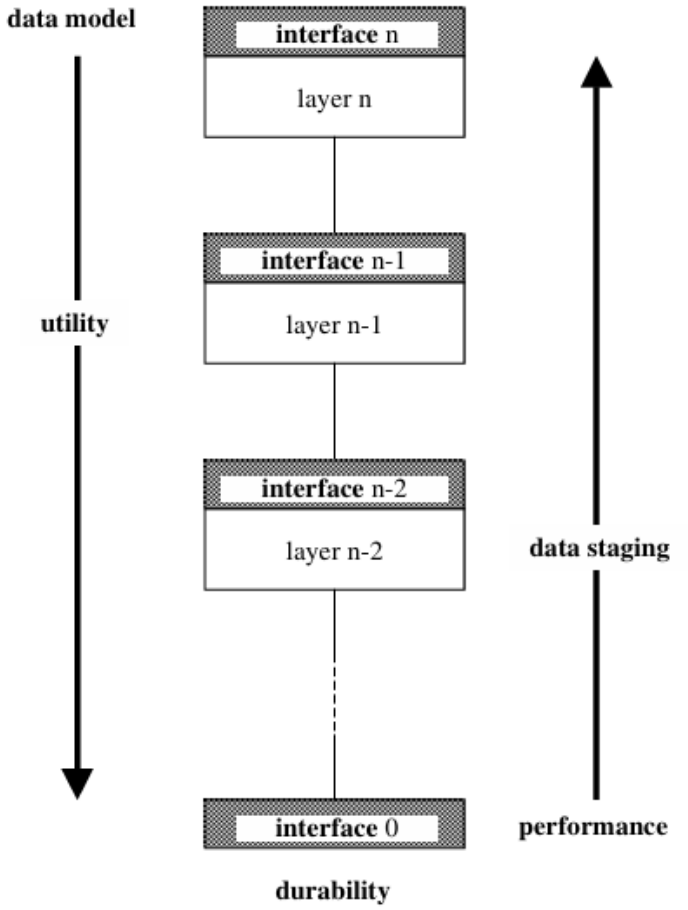

Рисунок 2.5 - Архітектурний проект, орієнтований на модель даних та продуктивність

Відома еталонна архітектура для СУБД використовує п'ять шарів. Архітектуру показано на рисунку 2.7. Ми демонструємо, що цю архітектуру можна пояснити з точки зору двох факторів: моделі даних та продуктивності, дотримуючись основних ідей, щойно згаданих.

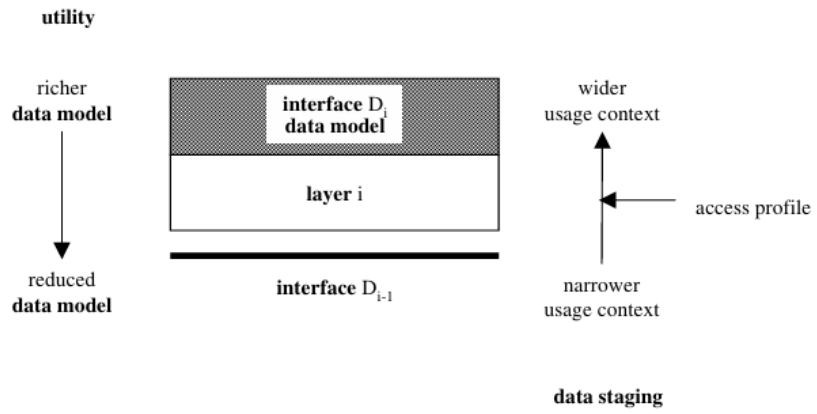

Рисунок 2.6 - Збалансування функціональності та продуктивності в межах одного рівня

1. Ми припускаємо модель даних рівня сервісу, яка має структуру множин/записів і використовує орієнтовану на множини, а отже, описову мову запитів, таку як SQL або OQL. Для розміщення даних на найвищому рівні ми припускаємо переважання запитів на читання. Відповідно, ми досліджуємо послідовність запитів на читання та аналізуємо результат на предмет частих шаблонів колективно отриманих даних (профіль читання). Потім дані будуть розміщені шляхом перевпорядкування бази даних відповідно до цих шаблонів. Це, у свою чергу, визначає модель даних для наступного нижчого рівня (внутрішня модель даних): Вона знову повинна мати структуру множин/записів.

2. Тепер перейдемо до функціонального відображення. Повинно відбутися деяке зниження виразності. Оскільки структурно різниця невелика, вона може полягати лише в операторах. Тепер вони будуть орієнтовані на записи, тобто навігаційні . Відображення охоплює три аспекти. Перший – це фактичне перегрупування зовнішніх структур. Другий стосується перетворення запитів у вирази алгебраїчних типів множин над перегрупованою базою даних, алгебраїчної та неалгебраїчної оптимізації виразів з функцією вартості, яка мінімізує оцінені розміри проміжних результатів. Третій забезпечує реалізацію алгебраїчних операторів у термінах наступної нижчої моделі даних.

3. Ми знову чергуємо процес з розміщенням даних. Ми аналізуємо весь профіль запиту, тепер з точки зору операторів запису. Наша надія — знайти

характерні шаблони послідовностей операторів для кожного внутрішнього набору. Кожен шаблон визначатиме відповідну організацію даних разом з реалізаціями операторів. Отже , модель даних на наступному нижчому рівні пропонує колекцію чогось подібного до класів (фізичних структур даних).

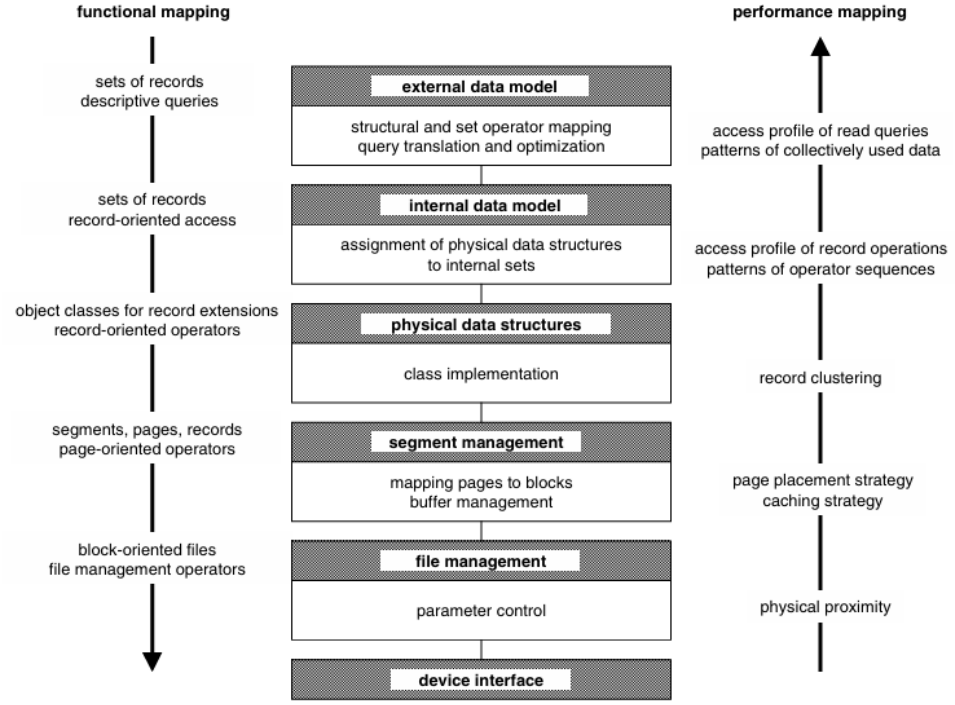

Рисунок 2.7 - Еталонна архітектура систем управління базами даних, орієнтованих на набори/записи

4. Повертаючись до функціонального відображення, ми в основному призначаємо фізичні структури даних внутрішнім наборам.

5. На цьому етапі ми змінюємо напрямок і починаємо знизу. Враховуючи пристрої зберігання даних, ми використовуємо фізичне управління файлами, як це передбачено операційними системами. Ми обираємо блочно-орієнтовану організацію файлів, оскільки вона робить найменше припущень щодо подальшого використання даних і пропонує однорідне представлення на всіх пристроях. Ми використовуємо налаштування параметрів для впливу на продуктивність. Параметри стосуються, серед іншого, розміру файлу та динамічного зростання, розміру блоку, розміщення блоків, адресації блоків

(віртуальної чи фізичної). Щоб закласти основу для поетапного розміщення даних, нам потрібен контроль над фізичною близькістю: нумерація сусідніх блоків повинна бути еквівалентна мінімальній затримці при послідовному доступі [37]. На жаль, не всі операційні системи пропонують такий ступінь контролю. Модель даних визначається класичними функціями управління файлами.

6. Наступний верхній рівень, управління сегментами, є особливо важливим з функціональної точки зору, оскільки він утворює місток між світом, позбавленим сервісного контенту та просто переміщуючим блоки байтів, і світом, який має готуватися до сервісів. Міст підтримує структуру блоків, але робить додаткове припущення, що блоки містять записи набагато меншого розміру. Це вимагає більш складного управління простором як всередині блоків, так і між ними, а також ефективної схеми динамічного розміщення та адресації записів. Через всю цю додаткову цінність модель даних називає блоки сторінками, а файли сегментами. Оператори визначають доступ до сторінок і записів, а також розміщення записів.

7. Керування сегментами також визначатиме успіх розміщення даних на верхніх рівнях. По-перше, якщо для сторінок має бути використана фізична близькість блоків, зіставлення сторінок з блоками є критично важливим (стратегія розміщення). Ще більш критичним є те, як рівень використовує основну пам'ять для покращення доступу до сторінок на три-п'ять порядків. Використовуються великі буфери сторінок. Окрім розміру буфера, вирішальним фактором є прогностична модель «досить раннього» завантаження сторінки (стратегія кешування).

Залишаються деталі рівня фізичних структур даних. Враховуючи сторінку, до всіх записів на ній можна отримати доступ зі швидкістю основної пам'яті. Оскільки кожна структура даних відображає певний шаблон операцій із записами, ми перетворюємо цей шаблон на стратегію розміщення спільно використовуваних записів на одній сторінці (кластеризація записів).

Функціональне відображення потім стосується алгоритмічних рішень для методів класу.

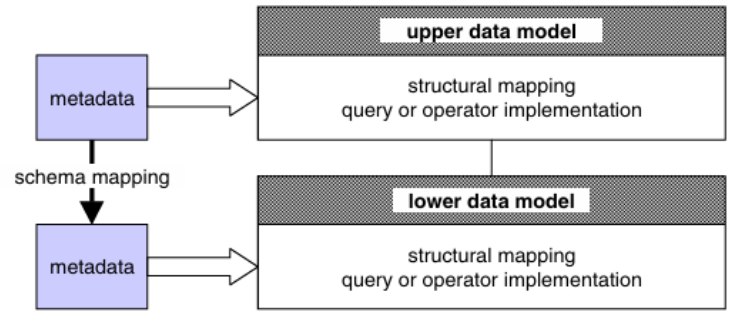

Рисунок 2.8 - Принцип забезпечення узгодженості схеми

Узгодженість схеми

Якщо модель даних та продуктивність справді є найважливішими факторами проектування, то решта функцій повинні мати лише ортогональний вплив.

Розглянемо функціональне відображення в розділі. Воно повністю базувалося на моделях даних і, отже, було універсальним. Однією з функцій, яку потрібно додати на цьому етапі, є узгодженість схеми. Узгодженість схеми еквівалентна безпеці типів у програмуванні. У універсальних рішеннях перевірка типів виконується під час виконання. Отже, інформація про тип, яку часто називають *метаданими*, повинна підтримуватися операціями на кожному рівні. Вміст метаданих на кожному рівні походить від метаданих на наступному верхньому рівні за допомогою відображення, яке визначається функціональним відображенням для моделей даних. На найвищому рівні схема бази даних являє собою метадані. Рисунок 2.8 ілюструє цей принцип і демонструє, що узгодженість дійсно ортогональна до моделі даних.

Оскільки з точки зору управління даними метадані – це просто дані, метадані всіх шарів часто збираються в окреме сховище, яке називається *словником даних,* таким чином удосконалюючи ортогональність (рис. 2.9).

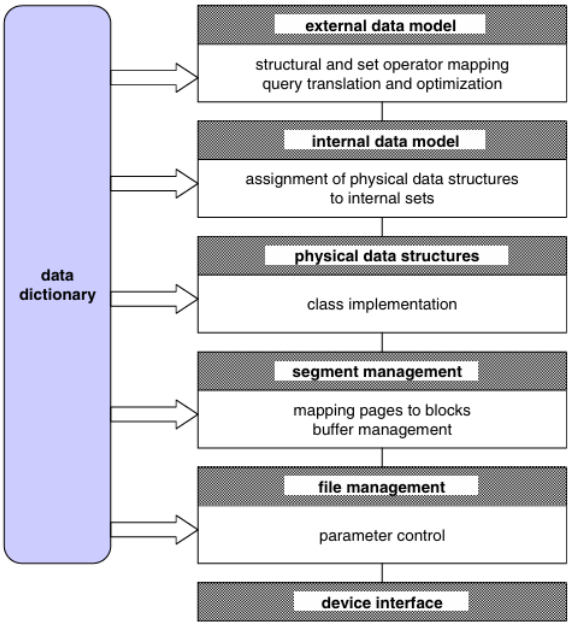

Рисунок 2.9 - Словник даних як ортогональний компонент системи

Послідовність, наполегливість та стійкість

Транзакції визначають досяжну узгодженість та стійкість. Вони також включають модель відмов для стійкості – атомарність для збоїв та ізоляцію для придушення конфліктів. Такі транзакції називаються властивостями ACID. Керування транзакціями складається з трьох компонентів: координатора транзакцій, який здійснює всю взаємодію з клієнтами та необхідний облік, планувальника, який упорядковує операції набору одночасних транзакцій для забезпечення серіалізованості та відновлюваності, та менеджера відновлення, який гарантує стійкість та стійкість.

Оскільки менеджери буферів відповідають за критично важливу частину продуктивності, вони працюють досить автономно. З іншого боку, атомарність вимагає тісної співпраці між менеджером відновлення (включаючи менеджер журналів) та менеджером буферів. Тому менеджери відновлення є невід'ємною частиною управління сегментами. Якщо ми хочемо досягти ортогональності, нам слід зосередити більшість або всі інші завдання управління транзакціями на цьому рівні, тобто в управлінні сегментами.

Як наслідок, ми також робимо планувальники частиною управління сегментами та синхронізуємо на основі звернень до сторінок. Координатори

транзакцій залишаються поза базовою архітектурою, оскільки вони не є критично важливими для продуктивності.

2.4 Висновки по розділу

У другому розділі досліджено методи та інструменти використання баз даних в архітектурі програмних систем. Розглянуто архітектурні підходи до організації взаємодії між програмними компонентами та базами даних, а також принципи управління ресурсами й транзакціями у розподілених системах. Проаналізовано еталонні архітектури систем управління базами даних та їх особливості. Визначено, що правильний вибір архітектурного підходу та механізмів управління даними сприяє підвищенню продуктивності, надійності та відмовостійкості програмних систем.

РОЗДІЛ 3. РОЗРОБКА, ІНТЕГРАЦІЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ БАЗ ДАНИХ У ПРОГРАМНИХ СИСТЕМАХ

3.1 Використання напівструктурованих баз даних у сучасних системах

У підрозділі ми визначили модель даних як одну з домінуючих характеристик. Якщо ми продовжимо перевіряти нашу уточнену гіпотезу, має сенс змінити припущення для моделі даних. У цьому розділі ми замінюємо модель даних наборів записів малого розміру та операцій з множинами моделлю даних для напівструктурованих баз даних. Модель відображає спроби

застосувати підхід до баз даних більш-менш структурованих баз даних документів Інтернету або класичного інформаційного пошуку.

1. *Модель даних.* Запис еталонної моделі замінюється графовою структурою. Вузол позначається неунікальною тегом, і навіть вузли-брати можуть мати ідентичні теги. Вузол може містити текст будь-якої довжини та, за бажанням, традиційний список атрибутів/значень. З точки зору системи, текст є атомарним. Пошук вважається переважною функцією обслуговування і здійснюється для всієї або частини структури графа, і навіть для різних наборів графів, шляхом зіставлення зі зразком. Зміни в структурі графа, незалежно від їхньої складності, розглядаються як атомарні.

Вибір моделі даних має наслідки для деяких інших функцій.

2. *Узгодженість.* Регулярність структурованих баз даних є основою узгодженості схеми. Чи може СУБД гарантувати парність інформації навіть в обмеженій формі узгодженості схеми, залежить від ступеня регулярності, який можна спостерігати для напівструктурованої бази даних. Регулярність необхідно буде визначити в менш обмежувальній формі регулярних виразів. Графи, додані до бази даних, повинні відповідати схемі. Оскільки узгодженість визначається на окремих графах, а оновлення до них вже є атомарними, транзакційні процедури, здається, відіграють меншу роль.

3. *Наполегливість.* З меншою потребою в транзакціях, довговічність зазвичай передбачає наполегливість.

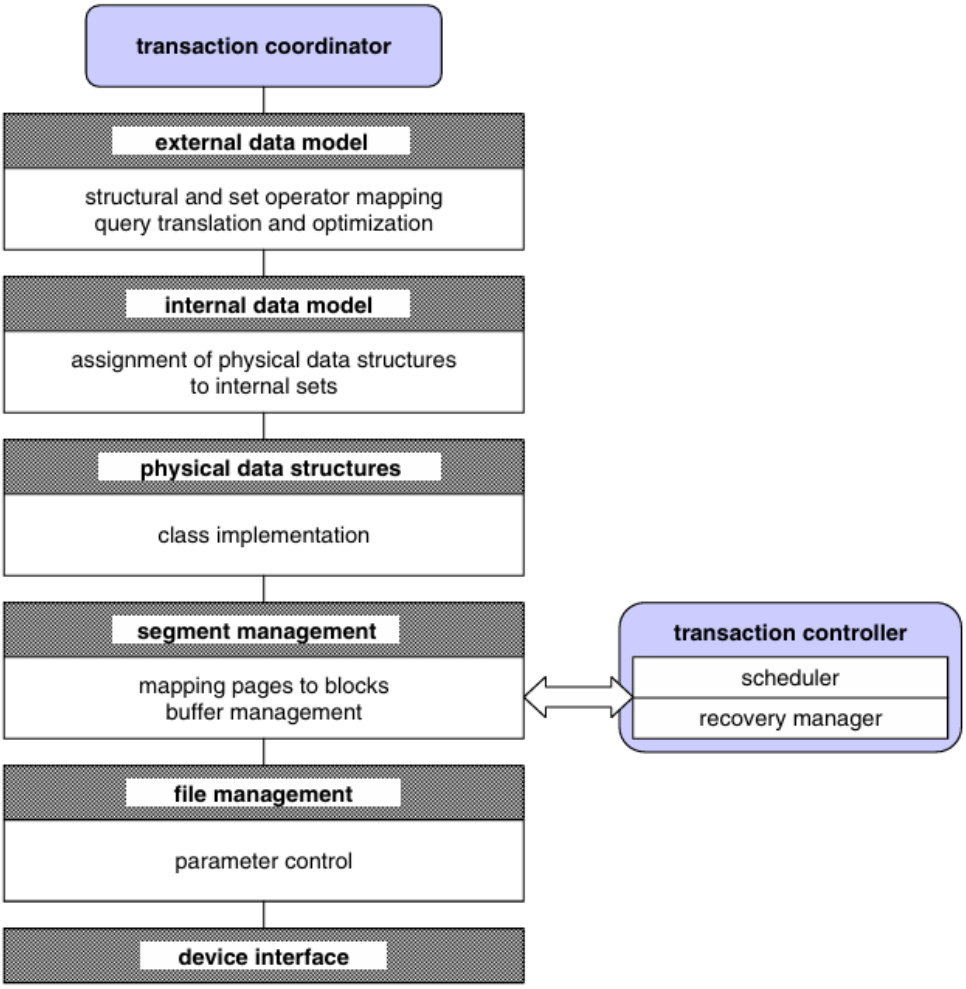

Рисунок 3.1 - Впровадження системи управління транзакціями

4. *Стійкість.* Без транзакцій має використовуватися інша модель відмов. Збої також можуть траплятися, але більше немає резервних позицій, які СУБД може автоматично розпізнавати. Натомість клієнти тепер повинні явно ідентифікувати постійні стани, так звані контрольні точки, до яких потрібно повернутися у разі збою. Оскільки зміни в структурі графа є атомарними, модель відмов не повинна враховувати перешкоди.

5. *Продуктивність.* Фізичне вузьке місце залишається тим самим. Отже, розміщення даних на певній стадії має залишатися основною стратегією підвищення продуктивності .

Архітектури

Низька узгодженість**.** Поняття схеми бази даних має сенс лише в середовищі з певним ступенем регулярності в структурах баз даних. Отже, там,

де немає такої регулярності, мало що можна було б забезпечити з точки зору узгодженості. Існують і інші недоліки. Мало що можна вказати на аналіз профілів доступу, а оскільки немає поняття розширення типу, існує мало можливостей для описового доступу між наборами і, отже, для оптимізації запитів. Отже, два верхні рівні з точки зору еталонної архітектури можуть мати дуже незначний вплив на продуктивність, і немає серйозних проблем із функціональним відображенням. Таким чином, ми можемо об'єднати рівні в один.

Однак існує сильне поняття фізичної близькості. Близькість визначається топологією графової структури, і оскільки доступ для читання здійснюється шляхом зіставлення зі зразком, топологічна інформація повинна бути кластеризована на якомога меншій кількості блоків файлів. Оскільки довгі текстові поля у вузлах є найбільшою перешкодою для кластеризації, їх слід відокремити та зберігати на інших блоках, знову ж таки, кластеризовано. На відміну від еталонної архітектури, де через невеликі розміри записів окрема сторінка є основним об'єктом занепокоєння, продуктивність тепер диктує, що кластери блоків враховуються.

Отже, залишаються два завдання. Для продуктивності кластери суміжних блоків даних повинні бути розміщені в основній пам'яті. Для корисності топологія графа та поля вузлів повинні бути зіставлені з кластерами блоків. Завдання стають частиною шару, який замінює шари фізичних структур даних та управління сегментами в еталонній архітектурі. За відсутності концепції транзакцій жодні інші фактори не впливають. Результат обговорення показано на рисунку 3.2.

Узгодженість схеми. Рішення попереднього розділу розглядає напівструктуровані бази даних ізольовано. Це суперечить останнім тенденціям поєднання традиційних структурованих баз даних з розділу 3 з напівструктурованими базами даних. Припустимо, що з міркувань *прозорості* ми підтримуємо єдину модель даних і погоджуємося з напівструктурованими даними як більш загальною з двох моделей даних. Однак, очевидно, що

включення структурованих баз даних змушує нас включати узгодженість як важливий фактор проектування. У свою чергу, нам потрібні графові структури, подібні з точки зору базових регулярних виразів. Їх потрібно буде класифікувати за типами. Отже, навіть якщо спостерігатиметься менша регулярність між їхніми екземплярами, як у традиційному випадку, існує поняття можливої схеми бази даних. Щоб отримати орієнтацію в множинах, доступ повинен охоплювати повні розширення типів.

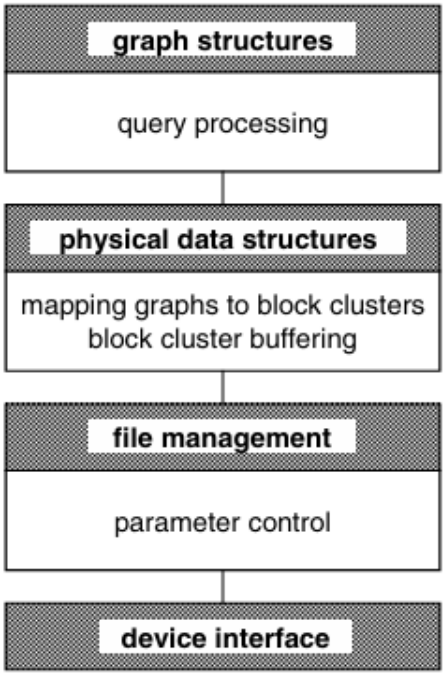

Рисунок 3.2 - Архітектура систем напівструктурованих баз даних з низьким рівнем узгодженості

Мови запитів знову можуть бути описовими, хоча їхня мета — лише пошук. Аналіз профілів пошуку можна проводити за типами графів. Усі ці аргументи свідчать на користь верхнього рівня, подібного до того, що в еталонній архітектурі. У деталях функціонального відображення цей рівень, найімовірніше, буде набагато потужнішим.

З точки зору продуктивності, шаблони пошуку повинні мати вплив. Наша гіпотеза полягає в тому, що поняття близькості знову може бути пов'язане з цими шаблонами. Якщо припустити, що доступ для читання обмежений

розширеннями одного типу, близькість можна перевести в підграфи, які відповідають шаблонам запитів. Отже, відображення продуктивності полягає в поділі вихідного графа на (можливо, перекриваючі) підграфи. Як і вище, графи слід розділити на топологію та довгі текстові поля або, загалом, довільні медіадані . Функціональне відображення тепер включає оптимізацію запитів.

Підграфи та довгі поля вимагають різних базових реалізацій. Фактично, при детальнішому розгляді підграфів ми можемо виявити, що деякі з них демонструють сильну регулярність традиційних структурованих баз даних. Отже, існує потреба в призначенні різних фізичних структур даних і, як наслідок, потреба в другому рівні нашої еталонної архітектури. Однак, хоча верхній рівень здається складнішим, ніж той, для другого рівня, здається, вірно зворотне.

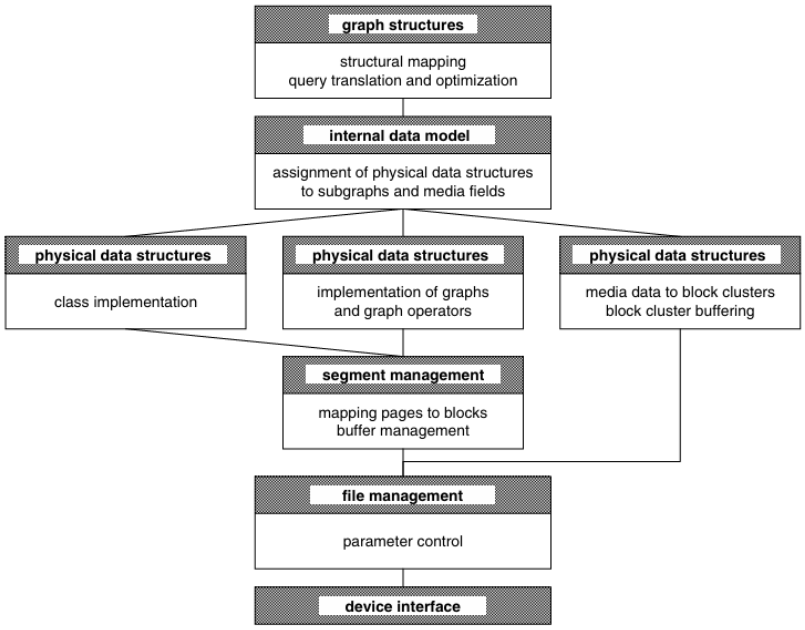

Рисунок 3.3 - Архітектура напівструктурованих систем баз даних із забезпеченням узгодженості схем

Для реалізації ми використовуємо двосторонній підхід. Існує велика ймовірність того, що підграфи мають розмір, який можна обмежити окремими блоками. Отже, їхня реалізація може відповідати еталонній архітектурі, починаючи з рівня фізичних структур даних і вниз, хоча в деталях методи можуть відрізнятися. Поля з неструктурованими медіаданими повинні відповідати архітектурі рис. 3.2. Рисунок 3.3 підсумовує обговорення.

3.2 Особливості мультимедійних баз даних та сервіс-орієнтованих рішень

Навіть якщо до вузла графа можна приєднати будь-який тип медіаданих, і ці типи можуть відрізнятися, напівструктуровані бази даних аж ніяк не є мультимедійними. Вони стають мультимедійними лише тоді, коли ми пропонуємо сервіси, які враховують взаємозв'язки між різними медіафайлами і, отже, здатні поєднувати два або більше медіафайлів. Нова функціональність сервісу, *перехресна синхронність,* гарантує, що вміст пов'язаних медіафайлів збігається в часовій шкалі. Візьмемо, наприклад, безперервне відтворення комбінованого відео та аудіо.

Перехресна синхронність залежить від *синхронності відтворення,* здатності системи представляти медіадані за часових умов, ідентичних тим, що були під час запису. Як функція послуги, синхронність відтворення балансує між функціональністю та якістю. Погана синхронність відтворення обмежує корисність, але існує певний допуск щодо швидкості, роздільної здатності та безперервності відтворення, в межах якого корисність зберігається. Отже, синхронність відтворення також можна розглядати як якість послуги.

Архітектура

Для вивчення архітектурних ефектів ми припускаємо, що перехресна синхронність є функціональністю сервісу , а синхронність відтворення – якістю сервісу. Оскільки перехресна синхронність є особливістю, що перевищує ті, що призвели до нашої еталонної архітектури , нам слід спробувати розглядати її як ортогональну особливість та інкапсулювати її в окремий шар або компонент десь на верхніх рівнях.

Синхронність відтворення, здається, тісно пов'язана з продуктивністю, оскільки вона має справу з тим самим типом вузького місця – периферійним сховищем. Однак умови ще суворіші. Відтворення має залишатися досить безперервним протягом кількох хвилин до години або близько того. Довші

перерви або тремтіння через переміщення механізмів доступу або конкуренцію з боку інших процесів повинні залишатися в гарантованих межах. Отже, залежно від швидкості використовуються спеціальні пристрої зберігання даних (потокові пристрої) або виділене дискове сховище. Спеціальні методи буферизації повинні вирівнювати залишки перерв та тремтіння. Шари вище повинні втручатися якомога менше. Отже, для безперервних носіїв права гілка на рис. 3.3 повинна стати ще більш спеціалізованою. Шар фізичних структур даних тепер повинен обмежуватися зберіганням вхідного потоку медіаданих та керуванням ним. Розривні медіадані та графові структури можуть слідувати архітектурі рис. 3.3.

Тепер, враховуючи синхронність відтворення кожних окремих медіаданих, верхні шари, показані на рис. 3.3, можуть бути збережені для обробки структурних аспектів взаємозв'язків між медіа, знову ж таки, втручаючись лише під час зберігання даних та під час фази налаштування відтворення. Зверху ми додаємо ще один шар, який контролює перехресну синхронність, зазвичай за зразком заданого сценарію. Рисунок 3.4 дає загальне уявлення про архітектуру.

Мережеві бази даних. Функції сервісу

Повертаємося до рис. 3.1. У розподілених інформаційних системах окремий бізнес-процес і навіть робоча одиниця стикаються з безліччю менеджерів ресурсів. Усі вони можуть бути розміщені в географічно різних місцях . Отже, *повсюдність* виступає як додаткова функціональність сервісу.

Функціональні можливості сервісу узгодженості та стійкості, а також якість сервісу та стійкість були пов'язані з поняттям транзакції бази даних і , отже, з єдиним менеджером ресурсів. Загалом, ці функції повинні бути пов'язані з бізнес-процесом в цілому. Технічно, ми називаємо бізнес-процес *транзакцією програми.* Зазвичай транзакція програми є розподіленою. Оскільки кожен компонент має справу з властивостями транзакцій індивідуально, *координація* вступає в гру як додаткова функціональність сервісу.

Клієнти послуг, а також постачальники послуг відрізняються за великою кількістю характеристик. Деякі з них мають виключно технічний характер, такі

як відмінності в апаратному забезпеченні та платформах операційних систем або протоколах передачі, які вони розуміють. Ці розбіжності називають технічною неоднорідністю.

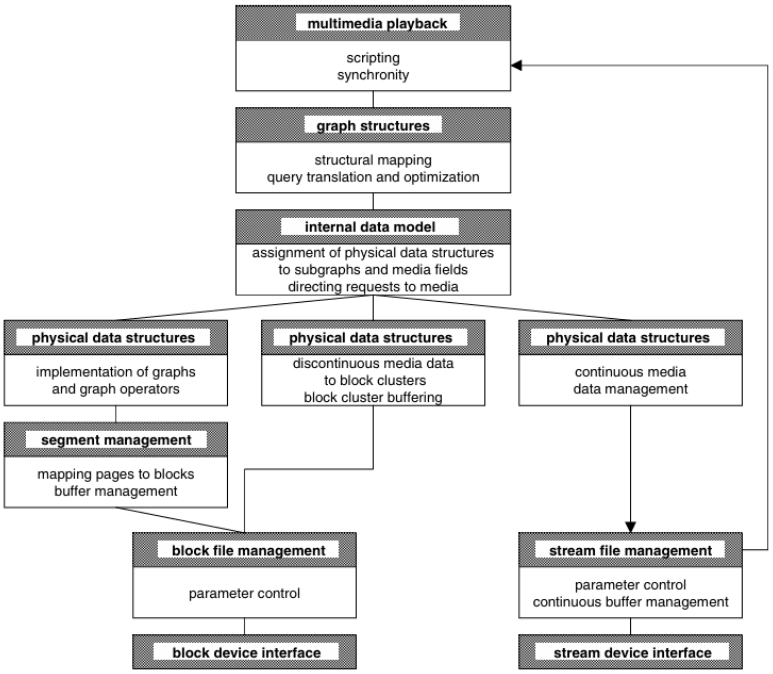

Рисунок 3.4 - Архітектура мультимедійних систем баз даних

Відмінності, що позначаються як семантична гетерогенність, пов'язані з функціональністю, наприклад, різними моделями даних, або узгодженістю, наприклад, різними схемами. Рішення про прийнятний рівень гетерогенності, здається, приймається поступово. Отже, ми додаємо нову якість послуг: *однорідність послуг.*

Проміжне програмне забезпечення

Якщо ми хочемо довести нашу гіпотезу, нам слід дотримуватися нашої попередньої архітектурної стратегії та реалізовувати нові функції ортогонально, коли це можливо. У випадку розподілених інформаційних систем ця стратегія зводиться до незначного торкання менеджерів ресурсів та додавання інфраструктури, яка відповідає за нові функції. Ця інфраструктура називається *проміжним програмним забезпеченням.*

Перша проблема, з якою потрібно розібратися, — це повсюдність.

Проблема вирішується шляхом використання інфраструктури передачі даних та застосування спільного протоколу високого рівня, наприклад, TCP/IP. Друга — це технічна однорідність. Проміжне програмне забезпечення забезпечує однорідність, встановлюючи внутрішній стандарт (візьмемо, наприклад, протокол IIOP CORBA) та вимагаючи локальних адаптерів.

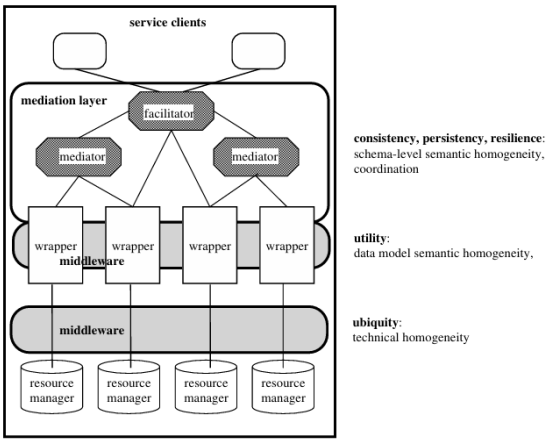

Рисунок 3.5 - Типова архітектура проміжного програмного забезпечення

Виробники підходять до семантичної однорідності, встановлюючи власний стандарт моделі даних (наприклад, HTML-документи для WWW, віддалені об'єкти для CORBA, DCOM та RMI).

Це залишає завдання, які набагато складніше стандартизувати, по суті всі ті, що пов'язані з контентом. Контент впливає на узгодженість та похідні характеристики, такі як стійкість або стійкість. Якщо неможливо створити стандарти, можуть допомогти метастандарти. Архітектурне представлення метастандартів здійснюється за допомогою фреймворків. Фреймворк, орієнтований на семантичну однорідність на рівні схеми та питання координації, - це архітектура І3 (Інтелектуальна інтеграція інформації). Рисунок 3.5 поєднує підходи проміжного програмного забезпечення та *в* єдину архітектуру.

Обгортки, медіатори та посередники є частиною фреймворку І3 . Обгортки адаптують своїх менеджерів ресурсів на семантичному рівні, зіставляючи локальну модель даних із загальною, стандартизованою моделлю даних, а схему – із такою, що виражається в термінах загальної моделі даних.

Медіатори гомогенізують схеми, долаючи розбіжності в термінології та структурі. Все частіше вони покладаються на онтології, представлені тезаурусами, словниками, каталогами або базами знань [39]. Інші медіатори приймають запити, що охоплюють кілька ресурсів, перетворюють їх відповідно до гомогенізованих схем, надсилають їх менеджерам ресурсів та збирають гомогенізовані результати. Посередники полегшують орієнтацію клієнтів сервісів у мережі. Прикладами посередників є веб-пошукові системи або каталоги джерел даних.

Модель даних, узгодженість та продуктивність

Локальна корисність та узгодженість є відповідальністю окремого менеджера ресурсів. Колективна корисність та узгодженість є сферою діяльності посередника. З точки зору окремої системи управління базами даних, усі аспекти глобальної корисності є додатковою функціональністю сервісу і, отже, віднесені до нового верхнього рівня. Рисунок 3.6 ілюструє цей логічний підхід. Цей підхід залишає відкритим питання, чи є посередник централізованим компонентом, чи розподіленим між менеджерами ресурсів.

Медіатори виконують на мережевому рівні те, що зовнішній рівень моделі даних виконує локально: переклад запитів, оптимізацію запитів, доставку запитів, збір та інтеграцію результатів. Це надзвичайно складні питання, але тим не менш ортогональні до всіх локальних завдань. Медіатор підтримується структурами метаданих, такими як каталоги та онтології, згадані раніше, або глобальною схемою (так званою федеративною схемою).

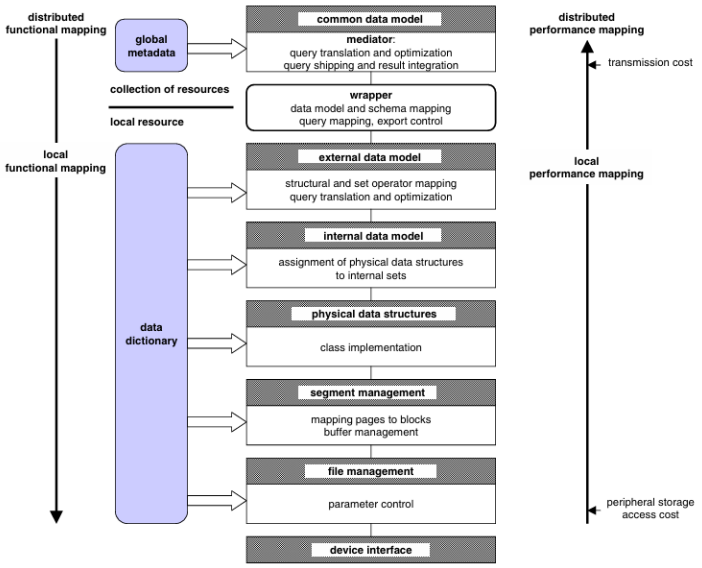

Рисунок 3.6 - Система управління базами даних у мережі

Обгортка, окрім зіставлення схеми, запитів та результатів із загальною моделлю даних, діє як своєрідний фільтр, щоб зробити видимими лише ті частини бази даних, які доступні глобально (так звана схема експорту).

Оскільки локальна система баз даних залишається незмінною, всі нові вузькі місця продуктивності виникають через розподіл. У минулому саме час передачі

домінувала у стратегіях посередництва, і залишається такою й сьогодні, тією мірою, якою пропускна здатність не встигає за швидкістю процесора. Незалежно від швидкості, затримка, тобто часова затримка між запитом і початком передачі, залишається серйозною перешкодою. Тому оптимізація запитів значною мірою визначається вартістю передачі.

Послідовність, наполегливість та стійкість

Глобально, функціональні можливості сервісів узгодженості та стійкості, а також якості сервісів паралельності та стійкості пов'язані з транзакцією програми. Локально, функціональні можливості та якості гарантуються транзакціями бази даних ACID. Ми обмежуємо наші розгляди транзакціями програми, які також є ACID. Отже, якщо для глобальної ізоляції та атомарності недостатньо покладатися виключно на локальну ізоляцію та атомарність, необхідно запровадити деякі глобальні, можливо, розподілені, механізми

координації.

З теорії транзакцій добре відомо, що локальна ізоляція є основою для будь-якого виду глобальної ізоляції. Отже, конфлікти між бізнес-процесами (див. рис. 3.1) обробляються виключно локально. Так само добре відомо, що глобальна атомарність – глобальна стійкість та стійкість – потребують координації у формі протоколів глобальної фіксації. Ці протоколи є централізованими в тому сенсі, що вони розрізняють координатора та його агентів, де агенти відображають тих менеджерів ресурсів, які оновили свої бази даних. Найбільш широко використовуваний протокол, Two-Phase-Commit (2PC), вимагає, щоб навіть після локальної фіксації агенти були здатні, під час так званої фази невизначеності, відкатити транзакцію [40]. У чисто локальному управлінні транзакціями фаза невизначеності не існує. Як наслідок, менеджер відновлення має бути відповідно адаптований.

Це залишає питання, де розмістити координацію та, до речі, моніторинг транзакцій. Обидва є універсальними завданнями. Якщо припустити, що транзакції додатків проходять через посередника, здається цілком природним розмістити ці завдання разом із проміжним програмним забезпеченням (рис. 3.7). І справді, саме це намагається зробити таке проміжне програмне забезпечення, як CORBA та DCOM. Фактично, сучасне розуміння багатьох ранніх систем обробки транзакцій є розумінням проміжного програмного забезпечення.

Підсумовуючи, облік розподіленого управління транзакціями все ще близький до ортогонального: додається глобальний компонент, а один локальний компонент адаптується.

3.3 Обробка великих даних у архітектурі програмних систем

Моделювання бази даних допомогло нам передбачити типи даних, які будуть у ній зберігатися, і як вони будуть зберігатися.

NoSQL, абревіатура від «Not only SQL» [22], – це підхід до управління базами даних, який чудово справляється з обробкою величезних обсягів

неструктурованих даних та аналітики великих даних [23] [14]. З цими базами даних можна використовувати всілякі мови запитів, і вони не дотримуються суворої, заздалегідь визначеної структури схеми. Однак протягом останніх кількох десятиліть реляційні бази даних використовували стандартну галузеву мову SQL. Документоорієнтовані бази даних є підмножиною баз даних NoSQL. Бази даних, що зосереджені на зберіганні та отриманні документів, включають MongoDB та CouchDB. Бази даних цього типу використовуються для зберігання та адміністрування даних, які переважно базуються на документах. Складні формати даних, такі як JSON, BSON, XML та PDF, використовуються для зберігання інформації в документоорієнтованих базах даних. Як MongoDB, так і CouchDB є безкоштовними та з відкритим вихідним кодом; однак MongoDB [24] більше підходить для розподіленого середовища та JSON. Дослідники в [25] досліджували кілька різних функцій баз даних NoSQL. Одна з популярних баз даних, створена спеціально з урахуванням JSON, - це MongoDB [26], яка використовує мову програмування C++. MongoDB використовує динамічні структури схем [17] замість попередньо визначених статичних документів. Аналіз та пошук даних є швидкими та точними завдяки покращенням обробки запитів, підтримці індексації та агрегації в пам'яті. Окрім повної безпеки, вона надає утиліти для відновлення та резервного копіювання. Бази даних SQL та NoSQL показано на рисунку. 3.7 разом із відповідною структурою зберігання даних.

SQL (Oracle, MySQL і SQL-Server) та NoSQL (MongoDB, Neo4j) були предметом численних досліджень, що порівнювали їхню структуру, дизайн та продуктивність. У цих дослідженнях використовувалися запропоновані методи для ситуацій, що стосуються баз даних SQL та NoSQL, та аналізувалися результати. Бази даних NoSQL мають різні функції та застосування. Через свою здатність до горизонтального масштабування бази даних NoSQL не можуть гарантувати властивість ACID. У цьому відношенні графова база даних Neo4j є чудовим варіантом. Було показано, що графові бази даних ефективно організовують та зберігають дані зі складними залежностями. Документи є

основним напрямком MongoDB. MongoDB використовує формат BSON, який є відгалуженням JSON.

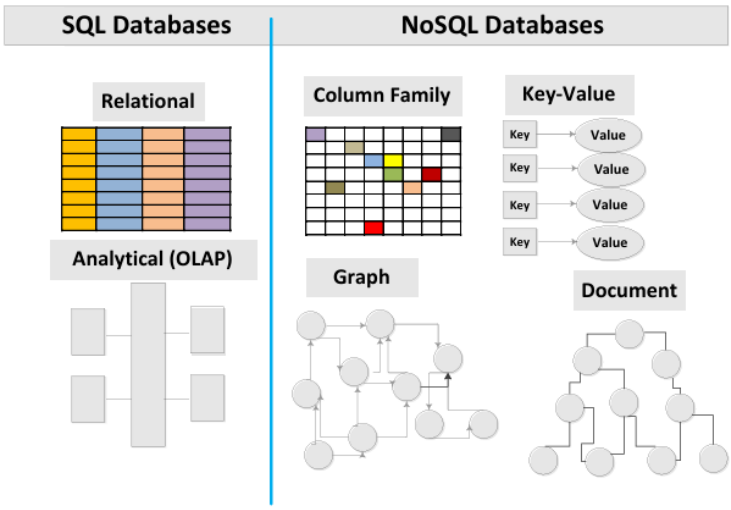

Рисунок 3.7 - Структури зберігання даних у базах даних SQL та NoSQL.

З величезними обсягами даних MongoDB добре справляється і неодноразово продемонстрував свою ефективність. У дослідженні порівнювали базу даних NoSQL MongoDB з реляційною базою даних Postgre SQL для даних сервісів соціальних мереж та потокових датчиків. У сценаріях, де порівнювалися дві бази даних, MongoDB вийшла на перше місце. На відміну від реляційних систем керування базами даних (RDBMS), база даних NoSQL добре підходить для використання в кластерному середовищі, де її гнучка та потужна архітектура може вмістити величезні обсяги даних у найрізноманітніших формах. MongoDB та MySQL порівнювали з канонічною системою керування базами даних у [35]. MongoDB перевершив MySQL у пошуку та вставці даних. На основі представлених порівнянь стало зрозуміло, що NoSQL MongoDB є кращим варіантом, ніж MySQL, при обробці великих обсягів даних. Для статей було проведено поглиблене дослідження передумов для ефективного управління величезними обсягами неструктурованих даних . Вони віддали перевагу використанню бази даних NoSQL MongoDB замість використання RDBMS MySQL. Реляційну та нереляційну моделі баз даних було детально розглянуто. Залежно від їхнього аналізу, база даних NoSQL MongoDB перевершила базу

даних MySQL. Вони припустили, що для невеликого набору даних з базовими запитами MySQL є ефективним, тоді як для великих наборів даних зі складними запитами MongoDB є більш прийнятним рішенням. Автори [40] порівняли продуктивність Oracle RDBMS з базою даних NoSQL MongoDB. В результаті свого дослідження вони дійшли висновку, що бази даних NoSQL не є життєздатною альтернативою базам даних SQL. Вони [40] стверджували, що компанії можуть вибрати базу даних, що відповідає їхнім потребам, враховуючи специфіку своєї компанії.

Що стосується масштабованості даних, високої продуктивності та доступності даних, ми виявили багато публікацій, які обговорюють міграцію з реляційної бази даних до бази даних NoSQL, а також інші статті на цю тему. Завдяки своїй масштабованості, цілісності, поширенню, безпеці та настроюваному дизайну, MongoDB добре підходить для обробки величезних обсягів даних. Згідно з цитованими роботами, перехід з реляційної бази даних на базу даних NoSQL MongoDB є найкращим рішенням. MongoDB використовує формат бінарної нотації об'єктів JSON (BSON) для зберігання своєї інформації. Оскільки він не потребує з'єднань, як реляційні бази даних, MongoDB здатний ефективно зберігати та отримувати велику кількість документів в одній колекції. MongoDB достатньо універсальний, щоб обробляти дані в широкому спектрі форматів, включаючи структуровані, напівструктуровані та неструктуровані дані. Реляційні бази даних були створені з чіткою метою обробки організованих даних, і як такі, вони можуть обробляти певний обсяг даних, що мають вертикальний характер. Зрозуміло, бази даних NoSQL не призначені для заміни реляційних баз даних, але їхня підвищена продуктивність та корисність означають, що вони мають місце поряд з реляційними базами даних у деяких контекстах. Згідно з дослідженням, реляційна система баз даних погано масштабується з величезними обсягами даних. Автоматичне зіставлення з RDBMS MySQL на базу даних NoSQL MongoDB з використанням метаданих, збережених у RDBMS MySQL. Бази даних NoSQL не обіцяють такої ж атомарності, узгодженості, ізоляції та довговічності, які є відмінними рисами

систем управління реляційними базами даних. Функція BASE спостерігається в базах даних NoSQL. NoSQL– це загальний термін для набору технологій, які мають спільні цілі: доступність даних, ефективне масштабування даних, ефективне управління сховищем та покращена продуктивність. У дослідженні [25] розглядаються мотиви переходу від реляційних систем управління базами даних до нереляційних документоорієнтованих баз даних. Тим часом, аспекти ACE (доступність, узгодженість та ефективність) системи великих даних були досліджені в обох базах даних.

Порівнювалася ефективність кількох NoSQL баз даних (включаючи - Couchbase, MongoDB, Re-thinkDB та in-memory) при використанні з реальними даними. Плануючи запропоновані ними випробування, автори враховували кілька факторів, таких як час відгуку різних баз даних. У роботі було проведено порівняльний аналіз продуктивності Oracle NoSQL та MongoDB. Бази даних вивчалися з різних точок зору, включаючи їхню модель зберігання, масштабованість, паралельність та реплікацію. Автори дійшли висновку, що загалом MongoDB була популярнішою, ніж NoSQL-пропозиція Oracle. Згідно з рейтингом DBEngine.com, MongoDB є п'ятою найкращою NoSQL-базою даних, тоді як Oracle NoSQL посідає 78 місце. У роботі стверджується, що порівняно з MapReduce MongoDB, агрегація RDBMS Oracle показала кращі результати. З іншого боку, щодо швидкості отримання відповідей на запити, MongoDB перевершила Oracle RDBMS. Агрегація Oracle перевершувала агрегацію MongoDB для робочих навантажень SUM, COUNT та AVG. У роботі використовувався набір даних приблизно *з* 500 000 записів, чого було недостатньо для аналізу великих даних. Блок-схема процесу оператора SQL Select в Oracle 11g RDBMS зображена на рисунку. 3.8 .

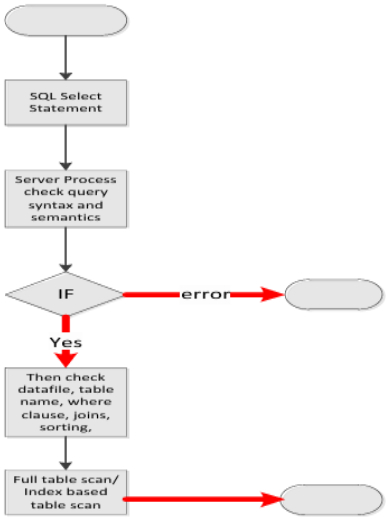

Рисунок 3.8 - Блок-схема виконання оператора SQL SELECT.

Отримання даних MongoDB є простим та легким завдяки структурі піддокументів і не вимагає перевірки обмежень або будь-яких умов, на відміну від оператора SQL Select в Oracle 11g RDBMS. Послідовність дій оператора SQL Insert показано на рисунку 3.9. Порівняно з Oracle RDBMS, вставка в MongoDB відбувається швидше, оскільки не потрібно перевіряти кроки, показані на рисунку. 3.9 оператора SQL INSERT.

На відміну від цього, MapReduce – це програмний модуль, який добре працює в сценаріях розподілу та більше підходить для аналізу великих даних порівняно з базовою агрегацією в кластерних (кілька серверних) контекстах. Перед початком етапу map, операції MapReduce можуть виконувати будь-яке довільне сортування та обмежувати документи в одній колекції, яку вони використовують як вхідні дані. Дві фази MapReduce, "Map" та "Reduce", необхідні для роботи алгоритму MapReduce; MongoDB використовує фазу map для кожного документа, який подається в MapReduce. Функція map повертає пару критичних значень. MongoDB використовує процес під назвою "reduction" для збору та стиснення агрегованих даних, а потім MongoDB зберігає результати в колекції. Наприклад, у наборі даних про злочинність у Чикаго функція map обчислює всі злочини отримання даних MongoDB є простим та легким завдяки структурі піддокументів і не вимагає перевірки обмежень або будь-яких умов, на

відміну від оператора SQL Select в Oracle 11g RDBMS. Послідовність дій оператора SQL Insert показано на рисунку 3.9.

Порівняно з Oracle RDBMS, вставка в MongoDB відбувається швидше, оскільки не потрібно перевіряти кроки, показані на рисунку 3.9 оператора SQL INSERT.

На відміну від цього, MapReduce – це програмний модуль, який добре працює в сценаріях розподілу та більше підходить для аналізу великих даних порівняно з базовою агрегацією в кластерних (кілька серверних) контекстах. Перед початком етапу map, операції MapReduce можуть виконувати будь-яке довільне сортування та обмежувати документи в одній колекції, яку вони використовують як вхідні дані. Дві фази MapReduce, "Map" та "Reduce", необхідні для роботи алгоритму MapReduce; MongoDB використовує фазу map для кожного документа, який подається в MapReduce. Функція map повертає пару критичних значень. MongoDB використовує процес під назвою "reduction" для збору та стиснення агрегованих даних, а потім MongoDB зберігає результати в колекції. Наприклад, у наборі даних про злочинність у місті функція map обчислює всі злочини.

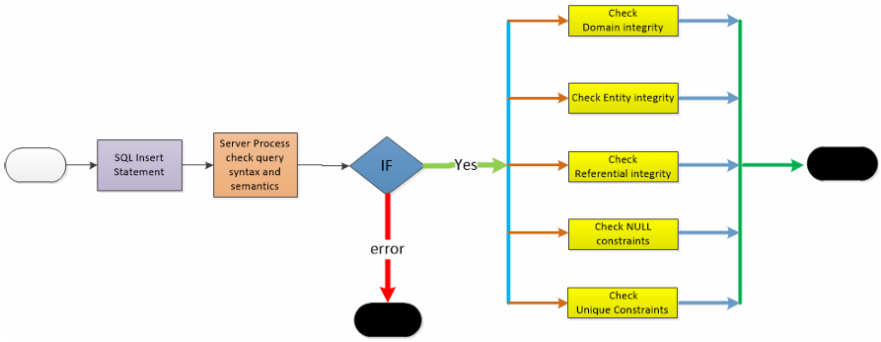

Рисунок 3.9 - Блок-схема виконання оператора SQL INSERT.

Протягом останнього десятиліття СУБД Oracle широко впроваджувалися компаніями будь-якого розміру. Через свою абстрактну системну архітектуру з одним образом, SQL СУБД погано справлялися з обробкою неструктурованих даних.)

У дослідженні досліджувалися кілька особливостей та класифікацій баз даних NoSQL, що працюють на базі Hadoo. У нашому дослідженні було виявлено, що бази даних NoSQL можна розділити на підкатегорії відповідно до таких факторів, як масштабованість та тип даних (підключені проти документних). Порівнюється Oracle RDBMS з MongoDB та перераховані їхні основні функції.

Ключові сильні сторони реляційної бази даних Oracle та NoSQL бази даних MongoDB описані в таблиці 3.1, порівнює бази даних NoSQL (MongoDB, Neo4j) з реляційними базами даних (Oracle, MySQL та SQL-Server).

Таблиця 3.1

Характеристики систем управління базами даних MongoDB та Oracle

Характеристика	**SQL (Реляційні БД)**	**MongoDB (NoSQL)**
Схема	Жорстка схема (Rigid schema)	Гнучка схема (Flexible schema)
Одиниця зберігання	Таблиця (Table)	Колекція (Collection)
Рядок / Запис	Рядок (Row)	Документ (Document)
Стовпець	Стовпець (Column)	Поле (Field)
Доступ до даних	Потребує багато об'єднань (**Joins**), що уповільнює доступ.	Дуже швидкий: всі дані зберігаються в одному документі.
Масштабованість	Вертикальна (Vertical scalability)	Горизонтальна (Horizontal scalability)
Шардинг	Складний розподіл даних	Простіший розподіл даних
Пріоритет	Ефективність зберігання даних	Швидкість, продуктивність, час розробника
Агрегація / Пошук	Добре працює з агрегацією	Краще працює з вибіркою (retrieval)
Big Data аналітика	Не дуже швидкий запис/читання	Дуже швидкий запис/читання через відображення пам'яті.
Алгебра	Підтримує реляційну алгебру	Підтримує реляційну алгебру

Згідно з цитованими роботами, спостерігається помітне збільшення кількості геопросторових та геолокаційних даних. Було розроблено багато програм для обробки та використання геопросторових даних, і це стосується багатьох різних галузей (управління надзвичайними ситуаціями, археологія , Інтернет речей та розумні міста). Для ефективної обробки величезних обсягів

географічних даних [27] абсолютно необхідна потужна система управління базами даних.

Під час обробки величезних обсягів даних, бази даних NoSQL, а не SQL, є найкращим варіантом для веб-застосунків. Геопросторові та геолокаційні дані також керувалися за допомогою баз даних NoSQL. У кількох звітах було висловлено припущення , що бази даних NoSQL здатні ефективно обробляти величезні обсяги неструктурованих даних. У дослідженні повідомляється , що дослідники вперше дізналися про просторові дані в фундаментальних роботах з геоінформаційних систем. У дослідженні досліджувалися дві проблеми з ефективною обробкою геопросторових запитів. Перша полягала в тому, що традиційні методи оптимізації були неадекватними для вирішення географічних запитів. З урахуванням великої аналітики стало зрозуміло, що кожен метод і підхід до географічних запитів, випробуваний досі, був неадекватним для величезних обсягів даних. На відміну від традиційних текстових даних, які містять лише обмежений набір якостей, геопросторові дані пропонують широкий спектр додаткових функцій. Згідно з кількома дослідженнями, обробка геопросторових даних вимагає добре налагоджених методів обробки величезної кількості геопросторових даних. У роботі було продемонстровано , що відомі системи управління реляційними СУБД стикалися з численними труднощами під час спроби аналізу географічних даних. Використовуючи порівняльний аналіз продуктивності, як NoSQL-база даних документів MongoDB порівнюється з реляційною системою управління базами даних PostGIS. Як показали їхні тести, MongoDB показала кращі результати, ніж PostGIS, при обробці геопросторових даних.

Модель просторового зберігання даних Oracle мала два основні компоненти: розташування та форму. Моделі зберігання, такі як Index Engine та Geometry Engine, що використовуються для аналізу запитів, використовували тип даних SDO_GEOMETRY. Для служб визначення місцезнаходження геокодер використовувався для перетворення адреси в інформацію SDO_GEOMETRY. Для візуалізації використовувалися Oracle Maps та Map

Viewer.

Таблиця 3.2

Геопросторові характеристики баз даних SQL та NoSQL.

Характеристика	RDBMS (Реляційні)	NoSQL
Дані	S (Структуровані)	SUSm (Структуровані, Неструктуровані, Напівструктуровані)
Схема	Фіксована (Fixed)	Динамічна (Dynamic)
Масштабованість	Вертикальна	Горизонтальна
Відповідність	**ACID** (Атомарність, Узгодженість, Ізоляція, Довговічність)	**BASE** (Базова доступність, Гнучкий стан, Кінцева узгодженість)
Архітектура	Централізована	Розподілена
Узгодженість	Сувора (Strict)	Кінцева (Eventual)
Мова запитів	SQL	OO API, SQL-подібні мови
Продуктивність	Повільна	Швидка

Дані зберігаються в шарді, а кластер MongoDB неможливо побудувати без одного або кількох шардів. Коли вузол виходить з ладу, інформація про цей шард зберігається його репліками. Транзакції даних (читання/запис) визначають, який шард використовується. Реплікований вузол використовує один або кілька серверів, а вторинний вузол моделюється як оригінальний сервер. У разі виходу з ладу основного сервера, його роботу візьмуть на себе резервні сервери. Сервер є центром, навколо якого обертаються всі операції (читання/запис). Зрештою, кластер буде синхронізовано з усіма розподіленими транзакціями читання. Набір серверів конфігурації в кластері відповідає за зберігання метаданих. Шарди даних ідентифікуються цими серверами, які також передають, який фрагмент даних належить до якого шарду. Служба підтримки клієнтів запитує, щоб дію виконали або сервери маршрутизації, або MongoDB. Перш ніж отримати підтвердження від клієнта, MongoDB призначає кожному користувачеві відповідний шард на основі завдання thpe, а потім об'єднує отриманий dcla. Оскільки Mongos [23] є бездержавними, їх можна використовувати в розподіленому середовищі.

Файли, що відображаються в пам'яті, використовуються MongoDB для

максимального використання доступної пам'яті, що, у свою чергу, підвищує продуктивність. Індексація в базі даних MongoDB використовує B-дерева. У кластері MongoDB користувач може заявити право власності на певну колекцію розділів, використовуючи ключ шарду.

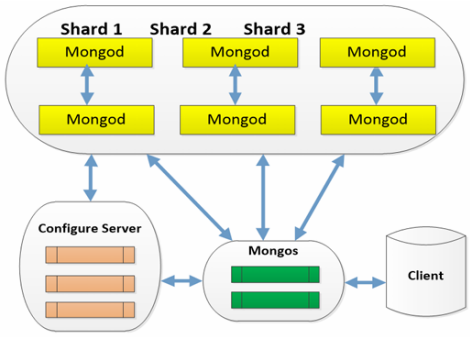

Рисунок 3.10 - Архітектура MongoDB

Для цієї роботи було проведено поглиблене дослідження літератури з архітектури DBaaS. Аналіз та вилучення даних показують, що хмарний підхід DBaaS, розроблений для реляційних баз даних, не є оптимальним для NoSQL баз даних. Час та зусилля експертів можна скоротити, а безпеку можна підвищити, якщо вчені та дослідники зможуть знайти єдине рішення DBaaS як для SQL, так і для NoSQL баз даних. «Також немає потреби переробляти програми для використання з різними постачальниками хмарних послуг (CSP) завдяки стандартним API (інтерфейсам API). Переносимість даних та сумісність між різними постачальниками хмарних послуг є основними перешкодами, які необхідно подолати. Суміжність визначається по-різному кожною з трьох парадигм: IaaS, PaaS та SaaS. Відкриті стандарти допомагають вирішити - проблему сумісності; однак наше рішення зосереджено саме на рівні IaaS. Під час передачі інформації між різними постачальниками хмарних послуг зазвичай необхідні уніфіковані API. Не існує єдиної моделі зберігання даних, яка використовується всіма хмарними сервісами. Коли розробник мігрує з одного CSP до іншого, дані передаються відповідно до високорівневої архітектури, зображеної на рисунку 3.11.

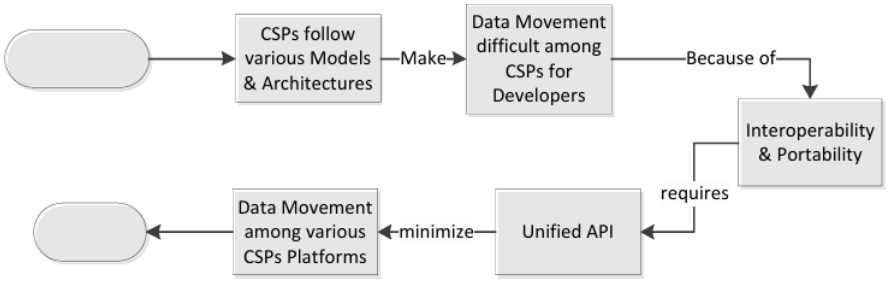

Рисунок 3.11 - Передача даних всередині хмарних сервісних платформ

Бази даних, які є одночасно узгодженими та пов'язаними, а також ефективне та результативне управління, є ключовими та критично важливими питаннями в сучасну епоху інформаційних технологій. Фундаментальними характеристиками системи баз даних є гарантія постійного сховища даних незалежності даних від базового фізичного носія зберігання, а також можливості обробки черг, що забезпечуються декларативними запитами (DBS). У секторі баз даних спостерігається широкий спектр підходів до різних областей. Кілька DMS, включаючи ACID, OOM, XML та сховища даних, базуються на реляційній моделі 8ata. Однак атрибут BASE підтримується в базах даних oSQL, які в першу чергу призначені для захисту великих обсягів даних.

Нові хмарні сервіси та можливості, що пропонуються постачальниками хмарних послуг клієнтам за низькою ціною та з високою ефективністю. Однак, кілька постачальників хмарних послуг пропонують однакові функції, використовуючи різні реалізації та інтерфейси користувача, що неминуче спричиняє проблеми зі сумісністю, несумісністю та портативністю. Це складні проблеми для постачальників хмарних послуг, які одночасно впроваджують та сприяють хмарним технологіям. Взаємодія I aaS, PaaS та S aaS – це різні терміни з різними значеннями та застосуваннями в сфері хмарних сервісів. Користувачів хмарних послуг слід переміщувати від одного CSP до іншого з таких причин: вища частота простоїв або збоїв, розірвання контракту, зміни корпоративних планів, кращі альтернативи з низькою вартістю та юридичні труднощі. Клієнти, які використовують кілька хмарних постачальників, не можуть легко переносити

свої дані між собою. Моделі хмарних сервісів прагнуть контролювати можливості клієнта, оскільки їхні ключові архітектури не мають сумісності. Ця проблема вирішується шляхом використання прив'язки до постачальника, що створює значну проблему безпеки для моделей на основі хмарних технологій. Портативність DST покращить ситуацію, коли все частіше постачальники хмарних послуг (CSP) впроваджують відкритий стандарт для вирішення проблем сумісності.

сумісності, необхідної для користувачів хмарних технологій та їхніх постачальників послуг інфраструктури. Використовуючи OCCI, можна зменшити сумісність між постачальниками послуг зв'язку (CSP), водночас забезпечуючи адекватні результати. Оскільки ці стандарти не включають функції сумісності в свої базові архітектури, вони не використовуються широко різними постачальниками послуг зв'язку. Крім того, існує проблема з доступністю стандартизованих інтерфейсів та API.

Шаблони проектування, хмарні проміжні інфраструктури, хмарна платформа надання послуг (SDCP) та інструменти міграції були розроблені іншими науковцями для полегшення переміщення даних між різними хмарами. Незважаючи на економію часу користувачів, ці методи не вирішують проблему переносимості між різними постачальниками послуг зв'язку, яка виникає через необхідність вивчення та впровадження кількох API. Amazon Web Service (AWS), Microsoft Azure та Google App Engine (GAE) є прикладами постачальників хмарних послуг, які допомагають споживачам створювати та запускати хмарні додатки. Крім того, вони надають хмарну платформу Database as a Service (DBaaS) для підтримки розробників баз даних. Перенесення даних у DBaaS, яке може відбуватися між базами даних SQL та NoSQL, а також всередині кожного типу бази даних NoSQL, може бути проблематичним через проблеми сумісності. Багато типів баз даних NoSQL дотримуються несумісних форматів зберігання та парадигм даних. Існує потреба в стандартизованих API, які можуть регулювати переміщення даних між різними платформами хмарного зберігання.

Для розміщення своїх даних і програм розробники програмного забезпечення можуть скористатися перевагами DBaaS, високомасштабованого та доступного хмарного сервісу backend. Як результат, DBaaS є найпривабливішим варіантом для хмарних клієнтів наразі. База даних як послуга (DBaaS) – це хмарний сервіс, що пропонується хмарними постачальниками (CSP), який полегшує перехід від традиційної архітектури баз даних до хмарної архітектури баз даних. SaaS забезпечує віддалений доступ до комп'ютерних програм, їх функцій і можливостей. PNUTS, HBase, SimpleDB та Google BigTable – це деякі інші хмарні сервіси баз даних . Потенційно, фреймворк DBaaS може добре справлятися з обробкою традиційних баз даних. Однак продуктивність DBaaS знижується в таких областях, як узгодженість даних, конфіденційність, цілісність, доступність та відсутність безпеки через різні підходи, яких дотримуються численні бази даних. Якби конфіденційність та безпека даних були гарантовані, модель аутсорсингу, відома як DBaaS, могла б мати фінансовий успіх. У дослідженні було запропоновано консультанта-експерта з віртуалізації для хмарних баз даних приватних таблиць. Хмарні обчислення дозволяють інтегрувати численні програми в основну структуру самого сервісу. Масштабованість та адаптивність хмарних обчислень мають нижчу ціну, ніж будь-коли раніше.

Щоб забезпечити більшу сумісність даних, портативність та безпеку під час перенесення даних, у роботі досліджувалися два уніфіковані фреймворки API, CDPort та Se-cloudDB, для баз даних SQL та NoSQL. Перед передачею третій стороні, запропонований ними API забезпечував конфіденційність критичної інформації користувачів (CSP). У рамках запланованого MCTool запити перетворюються на відповідні моделі, а потім передаються до відповідної бази даних, враховуючи моделі, які вона може обробляти. Ключі шифрування/дешифрування метаданих гарантують, що лише авторизовані користувачі та адміністратор баз даних мають доступ до даних, що зберігаються в різних хмарах, та можуть вносити зміни до них. Запропонований фреймворк підтримує шифрування та дешифрування.

Обговорення та класифікація

Дискусія SQL проти NoSQL не стосується реляційних та нереляційних баз даних, незважаючи на назви. Аналізуються та порівнюються моделі транзакцій обох баз даних. Під час виконання програми база даних виконує серію операцій, відомих як «транзакції ». Усі транзакції в базі даних SQL залежать від властивостей AC ID. У цьому випадку ACID посилається на принципи атомізму, узгодженості, ізоляції та довговічності. Однак розробники баз даних NoSQL дійшли висновку, що властивість ACID є марною перешкодою для захисту величезних даних. Отже, на початку 2000-х років професор Ерік Брюер запропонував нову теорію. Принципи CAP - це узгодженість, доступність та толерантність до розділів. Теорема стверджує, що всі ці якості можуть бути отримані одночасно розробниками, які працюють у розподіленому середовищі. Як компроміс для гарантованої узгодженості та доступності, розробники можуть реалізувати базу даних, толерантну до розділів, використовуючи модель CA. Якщо розробник бази даних надає більше значення доступності та толерантності до розділів, ніж узгодженості, то не можна сказати, що база даних базується на AP. Щоб досягти узгодженості та толерантності до розділів без шкоди для доступності, розгортається база даних на основі CP. Таблиця 3.3 описано основні функції, що підтримуються обома типами баз даних.

Стандартизація класифікації СУБД призвела до появи кількох підкатегорій. Основою будь-якої системи управління базами даних є модель даних, на якій вона побудована. Багато сучасних комерційних СУБД значною мірою спираються на реляційну модель даних. Об'єктна модель даних отримала мало поширення, хоча й використовувалася в невеликій кількості комерційних

Систем Багато застарілих програм продовжують працювати на системах баз даних, що базуються на ієрархічних та мережевих моделях даних. Ми можемо класифікувати бізнес-модель NoSQL як одну з наступних.

Таблиця 3.3

Функції СКБД.

База даних	Геометричні об'єкти	Функціональність	Просторові індекси
Oracle	Точка, Лінія, Полігон тощо	Підтримка OGC	B-Trees, R-tree
PostGIS	Повний набір OGC	Підтримка OGC	GiST, R-Tree, B-Tree
Azure SQL	Повний набір OGC	Підтримка OGC	B-Trees, 2D plane
MongoDB	Точка, Лінія, Полігон	Перетин, Дистанція	2D index, 2D sphere
DocumentDB	Точка, Лінія, Полігон	Дистанція/Близькість	2D plane, Quadtree

Системи баз даних NoSQL отримали різноманітні рекомендації, що можна пояснити тим фактом, що структура реалізації не включає традиційний SQL. Кілька різних типів баз даних NoSQL використовують унікальний набір процедур запитів. Розробники програмного забезпечення часто відповідають за належне проектування виконання запитів, а не покладаються на декларативну мову запитів, яка поєднує питання та забезпечує швидкі плани виконання запитів. Це пояснюється тим, що декларативна мова запитів може поєднувати питання. Користувачі системи відповідають за виконання додаткових завдань, таких як перевірка та інтерпретація зібраної інформації. Крім того, відповідальність за забезпечення узгодженості даних, реплікації даних та доступності під час одночасних змін у спільних та реплікованих базах даних все більше переходить до рук розробників. Поєднання масивних систем даних з базами даних NoSQL може мати кілька різних наслідків для проектування та архітектури програмного забезпечення.

Брюер сформулював набір вимог до розподілених баз даних, використовуючи свій метод CAP [7]. Ці вимоги слугують базовими. У випадку, коли є мережевий розділ (P: у кластері, коли інформація випадковим чином втрачається між вузлами), узгодженість (C: представлення однакових даних усім користувачам) повинна мати пріоритет над доступністю (доступністю для користувачів) (A: підтвердження має бути отримане кожним клієнтом як успіх або невдача). Якщо розділу немає (P), фреймворк сприятиме затримці.

Сьогодні недостатньо покладатися виключно на структуровані дані,

оскільки неструктурована природа астрономічних даних вимагає швидкої аналітики даних та методів вилучення інформації. Можливості баз даних SQL обмежені в їхній здатності ефективно обробляти різні форми великих даних. Можливості баз даних NoSQL дозволяють ефективно обробляти масивні набори даних. Бази даних NoSQL перевершують у сферах розширення ємності сховища, гнучкості схеми, масштабованості та доступу в режимі реального часу. Бази даних NoSQL дотримуються атрибута BASE. Замість того, щоб надавати пріоритет узгодженості та безпеці даних, бази даних NoSQL роблять акцент на покращенні ефективності читання/запису даних. Додаток відповідає за забезпечення узгодженості даних. З цієї причини це чудовий варіант для обробки великих наборів даних. Крім того, бази даних NoSQL не застосовують обмеження на рівні даних, стовпців або таблиць, як це використовується в структурованих базах даних.

У цій роботі проаналізовано приблизно 140 попередніх досліджень, які порівнювали корисність, продуктивність та надійність баз даних SQL з базами даних NoSQL. У цьому дослідженні розглядалися величезні обсяги даних. Згідно з нашими дослідженнями, бази даних NoSQL пропонують більшу можливість розширення [23, 37]. Бази даних SQL краще підходять для транзакційних систем і використовують більше ресурсів для забезпечення цілісності та узгодженості даних, тоді як бази даних NoSQL краще оснащені для обробки величезних та різноманітних наборів даних і використовують менше ресурсів для забезпечення цілісності та узгодженості даних. З іншого боку, бази даних NoSQL [24] відрізняються тим, що вони надають більшого пріоритету доступності даних. Результати нашого тестування показують, що реляційні бази даних не можуть бути ефективно замінені базами даних NoSQL. Оскільки обидві бази даних мають свої переваги та недоліки, база даних, яку вирішить використовувати організація, залежатиме від вимог, унікальних для цього бізнесу. Для прикладу, бази даних NoSQL, які дозволяють використовувати програмний модуль MapReduce, краще підходять для паралельних обчислень, коли вони реалізовані в кластерному середовищі.

Бази даних NoSQL побудовані на основі більш гнучкої та динамічної схеми, на відміну від реляційних баз даних, які сильно залежать від попередньо визначеної структури даних, що називається «схемою» (таблична форма). Наприклад, для відстеження даних студентів слід використовувати поля StdRegNo, StdName та StdAddress. Під час роботи з реляційними базами даних перше , що потрібно зробити, це побудувати схему, яка задовольняє всі необхідні вимоги домену та цілісності.

Після цього можна буде зберегти відповідні дані студентів та дотримуватися необхідних обмежень. Розглядаючи можливість розширення обсягу поточної бази даних шляхом додавання двох нових стовпців, слід врахувати це. Ті, хто буде відповідати за внесення змін до існуючої схеми, також будуть відповідати за перенесення даних з попередньої схеми до нової. Під час роботи з великими наборами даних процедура може стати непрактично повільною та затягнутою. Той факт, що бази даних NoSQL не оновлюються автоматично щоразу, коли до схеми вносяться нові зміни, є найважливішою проблемою для гнучкої розробки програмного забезпечення [25]. Під час роботи з базами даних NoSQL не обов'язково мати заздалегідь визначену схему для внесення будь-яких змін. Таблиця А4 Додатка А містить список кількох баз даних, до яких було здійснено доступ під час обраного дослідження.

Нормалізація даних для контролю викидів, реляційних схем (атрибутів), обмежень домену, обмежень перевірок, обмежень унікальності та обмежень Not NULL, серед іншого, сприяє безпечній інтеграції даних у реляційних базах даних [24 , 26 , 37] на відміну від баз даних NoSQL [24 , 26 , 26 , 37]. Реляційні бази даних забезпечують добре налагоджені системи безпеки та автентифікації для користувачів. Бази даних SQL перевершують бази даних NoSQL як за продуктивністю, так і за довговічністю. SQL є стандартним інтерфейсом для всіх реляційних баз даних; однак бази даних NoSQL не
використовують SQL. Натомість вони використовують інший інтерфейс.

Бази даних SQL та NoSQL використовують широкий спектр моделей. Крім того, існує різниця в базовій моделі даних, що використовується кожним

типом бази даних NoSQL. Враховуючи поширеність кількох постачальників послуг зв'язку (CSP), важко забезпечити перенесення даних. Швидке розширення комерційних баз даних створює ще одну труднощі для хмарного сховища. Натомість, AWS DBaaS [1] пропонує дещо обмежені можливості розширення сховища для хмарних баз даних Azure. Категорії продуктів, архітектури, типи баз даних та мови наведено в таблиці 3.4.

Таблиця 3.4

База даних: категорія продукту, тип архітектури та мови.

Назва БД	Категорія	Архітектура	Тип	Мова програмування
MongoDB	NoSQL (Документна)	Розподілена мультимодель	Open Source	C++, Go, JS, Python
MySQL	SQL	-	Open Source	C, C++
Oracle	SQL	-	Комерційна	Асемблер, C, C++
SQL Server	SQL	-	Комерційна	C, C++
Neo4j	NoSQL (Графова)	-	Open Source	Java
CouchDB	NoSQL (Документна)	Розподілена мультимодель	Open Source	Erlang, JS, C, C++
Cassandra	NoSQL (Широкостовпчикова)	Розподілена мультимодель	Open Source	Java
BigTable	NoSQL (Широкостовпчикова)	-	Комерційна	C++ (ядро), Java, Go, Ruby
PostgreSQL	SQL	-	Open Source	C
GraphDB	NoSQL (Графова)	Розподілена	-	Java
Redis	NoSQL (Ключ-значення)	-	Open Source	ANSI C
Azure SQL	SQL	-	Комерційна	C, C++
Azure DocumentDB	NoSQL (Документна)	-	Комерційна	SQL (Core) API

Прогалина в дослідженнях

Впровадження складних систем розподілу необхідне для досягнення значного рівня як доступності, так і масштабованості. Крім того, шардування та розподіл відбуваються на базових рівнях архітектури програмного забезпечення, які називаються рівнем додатків, рівнем кешування та рівнем серверного сховища відповідно. Однак досягнення високої масштабованості може бути складним завданням через атомарну абстракцію, яку представляє типовий

фреймворк, що використовує SQL як золотий стандарт непроцедурної мови. Крім того, програмне забезпечення має бути інтелектуальним, щоб вирішувати проблеми, що виникають з репліками даних, та невідповідності, спричинені колізіями між змінами в репліках, що відбуваються одночасно. Що стосується критеріїв якості, таких як масштабованість, узгодженість, продуктивність та довговічність, кожен різновид NoSQL бази даних має свій унікальний набір недоліків та обмежень. Крім того, архітектор принципово повинен проводити дослідження характеристик номінальної бази даних, щоб відповідати вимогам, що висуваються постачальником, під час вибору відповідної бази даних.

Термін «великі дані» часто використовується для позначення даних, отриманих з великомасштабних сенсорних мереж або інформації, згенерованої користувачами із соціальних мереж. Однак, оскільки багато людей не мають необхідних навичок, вони не знають, як обробляти дані для досягнення бажаного результату для свого бізнесу. Це пояснюється тим, що значна кількість технологій, методів та дисциплін, що стосуються великих даних, є відносно новими розробками. Реляційні сховища даних, з іншого боку, не здатні обробляти такий великий набір даних, що вимагає створення нових можливостей системи зберігання. Дані, що зберігаються в базі даних NoSQL, не потребують централізованої схеми. Завдяки своїй більшій масштабованості, доступності даних, відмовостійкості та швидкій обробці величезних обсягів неструктурованих даних, бази даних NoSQL підходять для обробки великих даних. Існує багато питань, які ще не вирішені через відмінності між реляційними базами даних та базами даних NoSQL.

У роботі робиться висновок, що NoSQL не є придатною альтернативою реляційним базам даних. Крім того, NoSQL є чудовим варіантом для гетерогенних великих даних. Кожна з цих галузей має численні незаповнені прогалини в дослідженнях. Простота, масштабованість та продуктивність проектів баз даних NoSQL є областями зі значним простором для дослідження. На відміну від суворої структури даних схеми реляційних баз даних, бази даних NoSQL використовують модель даних типу «ключ/значення» у вигляді плоского

файлу або типу «ключ/значення» (табличну форму). Бази даних NoSQL, завдяки своїй горизонтальній масштабованості, добре підходять для обробки величезної кількості та різноманітності даних, що використовуються наразі. Однак це не стосується баз даних SQL, які можуть масштабуватися вертикально. При порівнянні NoSQL з реляційними базами даних масштабованість та продуктивність є двома найважливішими факторами. Також залишається без відповіді дослідницьке питання про те, як інтегрувати функції реляційних баз даних з функціями нереляційних баз даних. Для баз даних NoSQL також бракує досліджень щодо проблеми проектування простих схем.

Розробка гнучких фреймворків для міграції даних з реляційної бази даних до NoSQL-сховища даних – це ще одна галузь досліджень, яка отримала підвищений інтерес в останні роки. На карту поставлено багато факторів щодо успіху передачі даних з SQL до NoSQL. Це має вирішальне значення, оскільки будь-якій компанії необхідно отримувати корисну інформацію з власних даних, таку як використання системних ресурсів та оцінки ефективності роботи співробітників.

Крім того, NoSQL є кращим варіантом порівняно з реляційними базами даних у хмарі через розподілену природу хмарної платформи. Щоб зменшити зусилля користувачів хмарних служб під час переміщення даних між багатьма хмарними платформами та контролювати проблеми взаємодії та переносимості даних, сучасний рівень розвитку вимагає уніфікованих фреймворків API.

Прогнозування та виникнення СУБД для конкретної СУБД

Наша система даних базується на таблиці . Спочатку ми створили пару даних (x, y). Потім ми використали формулу комбінації, щоб створити пару (x, y) для рядка з більш ніж двома іменами баз даних, таким чином побудувавши 301 пару. Ми використали кодувальник міток для кодування x та y у числа з рядка для попередньої електронної обробки та використали гауссівський наївний баєсівський алгоритм для навчання наших закодованих даних. Після того, як вони були підготовлені, ми отримали унікальне ім'я бази даних з x. Ми закодували ці унікальні імена, а потім протестували їх на окремих закодованих

даних. Потім ми отримали ймовірність усіх класів, що відповідають окремим закодованим даним, і представили результат як унікальне закодоване x, що відповідає решті різних ймовірностей бази даних. Таким чином, розмір таблиці дорівнює n на n, де n — це унікальна база даних.

Аналіз зібраних даних: На зібраних даних ми навчили гауссову наївну баєсівську модель. Тим не менш, головна проблема полягає в тому, що вона передбачала MongoDB для всіх імен баз даних, окрім себе. MongoDB у наборі міток (y) багато порівняно з іншими. Відсоток MongoDB містить понад 22% усіх даних. Таким чином, модель має упередженість щодо даних. Щоб вирішити цю проблему дисбалансу даних, ми згенерували дані.

Аналіз зібраних та згенерованих даних: на основі аналізу, , ми дійшли висновку, що бази даних "MongoDB, Cassandra, MySQL та HBase" зробили значний внесок у зібраний набір даних. Таким чином, ми застосували стратегію ігнорування цих імен баз даних під час створення комбінації інших імен баз даних. Відсоток віку цих імен баз даних потім можна зменшити, а ім'я іншої бази даних можна збільшити, що призведе до збалансування набору даних. Набір даних видається збалансованим, а результати відрізняються від тих, що ми отримали із зібраного набору даних.

Результати дослідження показують, що немає потреби переходити від реляційних баз даних до NoSQL. Оскільки обидва типи баз даних мають свої переваги та недоліки, компанія може вибрати СУБД, яка найкраще відповідає її вимогам. Організація може використовувати базу даних SQL, якщо вона надає пріоритет стандартизації та узгодженості даних. NoSQL слід використовувати, коли бізнес має велику кількість неструктурованих даних, а доступність даних є високою вимогою. Реляційна база даних може бути кращою за NoSQL для агрегації невеликих наборів даних і навпаки для аналітики великих даних. MapReduce найбільше підходить для використання в кластерах завдяки своїй розподіленій природі. Хоча MapReduce повільний під час процесу агрегації, він ефективніший у паралельних обчисленнях і був розроблений для обробки величезних обсягів неструктурованих даних. NoSQL бази даних, завдяки своїй

розподіленій та високомасштабованій природі, добре підходять для програм, які генерують об'ємні дані з різноманітними характеристиками. Коли йдеться про геопросторові дані, масштабованість реляційних баз даних перевершує масштабованість NoSQL баз даних. NoSQL-бази даних мають швидший час відгуку, ніж реляційні бази даних, особливо під час обробки величезних обсягів геопросторових даних.

NoSQL-сховища даних пропонують альтернативу традиційним реляційним СУБД, але ймовірно, що організації не одразу вирішать, яку з них використовувати. Оптимальна стратегія вибору належної NoSQL-бази даних полягає у визначенні того, що вимагає програма, чого не може забезпечити реляційна система керування базами даних. Якщо реляційна система керування базами даних (СУБД) може ефективно керувати даними, NoSQL-система зберігання може не знадобитися.

Більше того, загальновизнано, що бази даних NoSQL є новішою розробкою в просторі баз даних. Однак вони розробляються з використанням загальновизнаних теорій. Незважаючи на переваги, системи на основі NoSQL не позбавлені своїх недоліків, таких як відсутність загальноприйнятих стандартів або добре відомої мови запитів для використання з базами даних NoSQL. Кожна база даних має свої власні характеристики та способи роботи. На противагу цьому, ці набори даних є новими та динамічними. Немає гарантії, що всі дані будуть записані до сховища даних правильно, оскільки бази даних NoSQL не пропонують суворих характеристик ACID. Крім того, бази даних NoSQL спрощують швидку розробку завдяки своїй динамічній/гнучкій схемі. На відміну від жорсткої структури реляційних баз даних, моделі та архітектури баз даних NoSQL є більш адаптивними.

Перехід від однієї моделі зберігання до іншої є складним завданням для розробників через різні моделі, що використовуються базами даних NoSQL. Різні постачальники послуг зв'язку (CSP) використовують несумісні протоколи та інтерфейси, що робить їх несумісними один з одним. Оскільки кожен CSP розробив власні API для своїх конкретних послуг, між ними стає важко

обмінюватися даними. Для ефективного управління переносимістю даних та їх сумісністю бази даних NoSQL потребують єдиного стандартизованого хмарного рішення.

Майбутні напрямки роботи щодо структурованих даних включають прийняття денормалізованої стратегії для SQL RDBMS та порівняння результатів вставки, оновлення та отримання даних у MongoDB з результатами в іншій системі. Натомість ми можемо порівняти продуктивність MongoDB та SQL RDBMS у конкретному сценарії, стежачи за нормалізованим підходом, який застосовує перша, а потім оцінюючи результати вставки, оновлення та отримання даних. Паралельні геопросторові підходи, що використовуються базами даних NoSQL, потребують більшої уваги, якщо вони мають ефективно обслуговувати велику кількість користувачів. Комп'ютерний зір, виявлення об'єктів, класифікація сигналів та різні інші програми глибокого навчання – це лише деякі з програм на основі глибокого навчання, які можуть отримати вигоду від розширення методологій великих даних.

3.4 Висновки по розділу

У третьому розділі розглянуто сучасні напрями розвитку баз даних та їх практичне використання в архітектурі програмних систем. Досліджено особливості застосування напівструктурованих і мультимедійних баз даних, а також сервіс-орієнтованих рішень для роботи з великими обсягами інформації. Проаналізовано підходи до обробки великих даних та їх інтеграції в сучасні програмні системи. Отримані результати підтвердили, що ефективне використання різних типів баз даних забезпечує гнучкість, масштабованість і високу продуктивність програмних рішень в умовах зростання обсягів даних та складності інформаційних процесів.

ВИСНОВКИ

У ході виконання дипломної роботи було досліджено роль баз даних у архітектурі програмних систем як ключового компонента, що забезпечує зберігання, обробку та узгодженість даних. Розглянуто сучасні підходи до проєктування програмних систем, у яких бази даних виступають не лише сховищем інформації, а й важливим елементом, що визначає продуктивність, масштабованість та надійність системи. У роботі проаналізовано існуючі архітектурні рішення, досліджено переваги та недоліки різних типів систем управління базами даних, а також визначено їх вплив на ефективність програмного забезпечення.

На початковому етапі було проведено аналіз предметної області та визначено основні вимоги до сучасних програмних систем, зокрема вимоги до продуктивності, масштабованості, доступності та безпеки даних. Розглянуто базові поняття архітектури програмного забезпечення, а також сучасні тенденції розвитку, включаючи перехід до мікросервісної архітектури та інтеграцію баз даних у гнучкі процеси розробки. Це дозволило сформувати цілісне уявлення про місце баз даних у структурі програмних систем.

У другому розділі досліджено методи та інструменти роботи з базами даних, включаючи архітектурні підходи до їх інтеграції, використання менеджерів ресурсів у розподілених системах та застосування еталонних архітектур систем управління базами даних. Особливу увагу приділено порівнянню різних типів СУБД, включаючи реляційні, NoSQL та мультимодельні рішення, що дозволило визначити їх доцільність використання залежно від специфіки задачі.

У третьому розділі розглянуто практичні аспекти використання баз даних у програмних системах, зокрема роботу з напівструктурованими та мультимедійними даними, а також обробку великих обсягів інформації. Проаналізовано проблеми, що виникають при моделюванні та інтеграції даних, такі як забезпечення узгодженості, оптимізація продуктивності та управління

транзакціями у розподілених середовищах. Оцінка ефективності різних підходів дозволила визначити оптимальні рішення для побудови сучасних інформаційних систем.

У процесі дослідження встановлено, що правильний вибір та інтеграція бази даних у архітектуру програмної системи є критичними для досягнення високої продуктивності та надійності. Використання сучасних технологій, таких як розподілені бази даних і системи обробки великих даних, дозволяє забезпечити ефективну роботу систем навіть за умов значних навантажень. Водночас існують певні обмеження, пов’язані зі складністю налаштування, підтримки та забезпечення узгодженості даних у розподілених середовищах.

Загалом, результати роботи підтверджують, що бази даних відіграють фундаментальну роль у формуванні архітектури програмних систем. Вони впливають на всі етапи життєвого циклу програмного забезпечення — від проєктування до експлуатації. Подальший розвиток у цій галузі пов’язаний із вдосконаленням методів інтеграції, оптимізації продуктивності та використання новітніх підходів до обробки даних, зокрема технологій великих даних і хмарних обчислень.

СПИСОК ПОСИЛАНЬ НА ДЖЕРЕЛА

1. Date, C. J. (2019). *An Introduction to Database Systems*. pearson.com. https://www.pearson.com
2. Elmasri, R., & Navathe, S. (2016). *Fundamentals of Database Systems*. pearson.com. https://www.pearson.com
3. Silberschatz, A., Korth, H., & Sudarshan, S. (2020). *Database System Concepts*. mcgraw-hill.com. https://www.mheducation.com
4. Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. martinfowler.com. https://martinfowler.com/eaaCatalog/
5. Evans, E. (2004). *Domain-Driven Design*. domainlanguage.com. https://domainlanguage.com
6. Martin, R. C. (2017). *Clean Architecture*. pearson.com. https://www.pearson.com
7. Kleppmann, M. (2017). *Designing Data-Intensive Applications*. oreilly.com. https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/
8. Stonebraker, M. (2018). The case for shared nothing. *Communications of the ACM*, 61(12), 54–60. https://doi.org/10.1145/3226595
9. Oracle Database Documentation. (2025). oracle.com. https://docs.oracle.com/en/database/
10. Microsoft SQL Server Documentation. (2025). learn.microsoft.com. https://learn.microsoft.com/en-us/sql/
11. PostgreSQL Documentation. (2025). postgresql.org. https://www.postgresql.org/docs/
12. MySQL Documentation. (2025). mysql.com. https://dev.mysql.com/doc/
13. MongoDB Documentation. (2025). mongodb.com. https://www.mongodb.com/docs/

14. Cassandra Documentation. (2025). cassandra.apache.org. https://cassandra.apache.org/doc/

15. Redis Documentation. (2025). redis.io. https://redis.io/docs/

16. Neo4j Graph Database Documentation. (2025). neo4j.com. https://neo4j.com/docs/

17. Azure Cosmos DB Documentation. (2025). learn.microsoft.com. https://learn.microsoft.com/en-us/azure/cosmos-db

18. Google Cloud Spanner Documentation. (2025). cloud.google.com. https://cloud.google.com/spanner/docs

19. Amazon DynamoDB Documentation. (2025). aws.amazon.com. https://docs.aws.amazon.com/dynamodb/

20. Amazon RDS Documentation. (2025). aws.amazon.com. https://docs.aws.amazon.com/rds/

21. Chen, P. (1976). The entity-relationship model. *ACM Transactions on Database Systems*, 1(1), 9–36. https://doi.org/10.1145/320434.320440

22. Codd, E. F. (1970). A relational model of data. *Communications of the ACM*, 13(6), 377–387. https://doi.org/10.1145/362384.362685

23. Bernstein, P. A., & Newcomer, E. (2009). *Principles of Transaction Processing*. morganclaypool.com. https://doi.org/10.2200/S00196ED1V01Y200811DTM001

24. Hellerstein, J. M., & Stonebraker, M. (2005). *Readings in Database Systems*. mit.edu. http://www.redbook.io

25. Sadalage, P., & Fowler, M. (2012). *NoSQL Distilled*. pearson.com. https://www.pearson.com

26. Lakshman, A., & Malik, P. (2010). Cassandra architecture. *ACM SIGOPS*, 44(2), 35–40. https://doi.org/10.1145/1773912.1773922

27. Abadi, D. J. (2012). Consistency tradeoffs in distributed systems. *Computer*, 45(2), 37–42. https://doi.org/10.1109/MC.2012.33

28. Brewer, E. (2012). CAP theorem revisited. *Computer*, 45(2), 23–29. https://doi.org/10.1109/MC.2012.37

29. Pritchett, D. (2008). BASE: An ACID alternative. *Queue*, 6(3), 48–55. https://doi.org/10.1145/1394127.1394128

30. OWASP. (2024). Top Ten Security Risks. owasp.org. https://owasp.org/www-project-top-ten/

31. Microsoft Azure Architecture Center. (2025). learn.microsoft.com. https://learn.microsoft.com/en-us/azure/architecture/

32. Google Cloud Architecture Framework. (2025). cloud.google.com. https://cloud.google.com/architecture/framework

33. AWS Well-Architected Framework. (2025). aws.amazon.com. https://aws.amazon.com/architecture/well-architected/

34. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns*. pearson.com. https://www.pearson.com

35. Newman, S. (2021). *Building Microservices*. oreilly.com. https://www.oreilly.com/library/view/building-microservices/9781492034018/

36. Richardson, C. (2018). *Microservices Patterns*. manning.com. https://www.manning.com/books/microservices-patterns

37. Stonebraker, M., & Çetintemel, U. (2005). “One size fits all”: database myth. *CIDR Conference*. http://cidrdb.org

38. Dean, J., & Ghemawat, S. (2008). MapReduce. *Communications of the ACM*, 51(1), 107–113. https://doi.org/10.1145/1327452.1327492

39. Zaharia, M. et al. (2016). Apache Spark overview. *Communications of the ACM*, 59(11), 56–65. https://doi.org/10.1145/2934664

40. Gartner. (2024). Data management trends. gartner.com. https://www.gartner.com/en/information-technology

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: "Роль баз даних у архітектурі програмних систем"

Обсяг пояснювальної записки: 86 аркушів

Дата закінчення бакалаврської роботи 10 червня 2026р.

Підпис студента _____