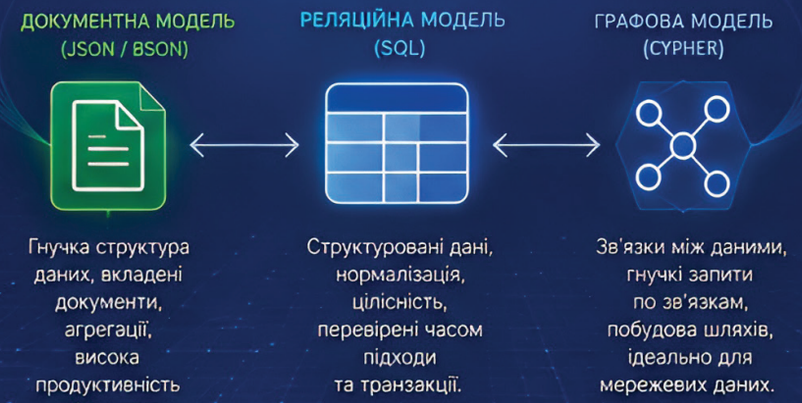
Навчальний посібник присвячено сучасним нереляційним моделям даних та особливостям їх практичного використання під час створення інформаційних систем.

У посібнику розглянуто теорему CAP, підходи ACID та BASE, документо-орієнтовану модель даних і СУБД MongoDB, графову модель даних і СУБД Neo4j, а також питання проєктування структур даних, індексування, агрегування та виконання запитів.

Особливістю видання є використання єдиної предметної області та наскрізних демонстраційних прикладів, що дозволяють порівняти різні підходи до організації даних і сформувати цілісне уявлення про сучасні технології зберігання інформації.

Для здобувачів вищої освіти IT-спеціальностей, аспірантів, викладачів та фахівців у галузі інформаційних технологій.



9 786177 980086

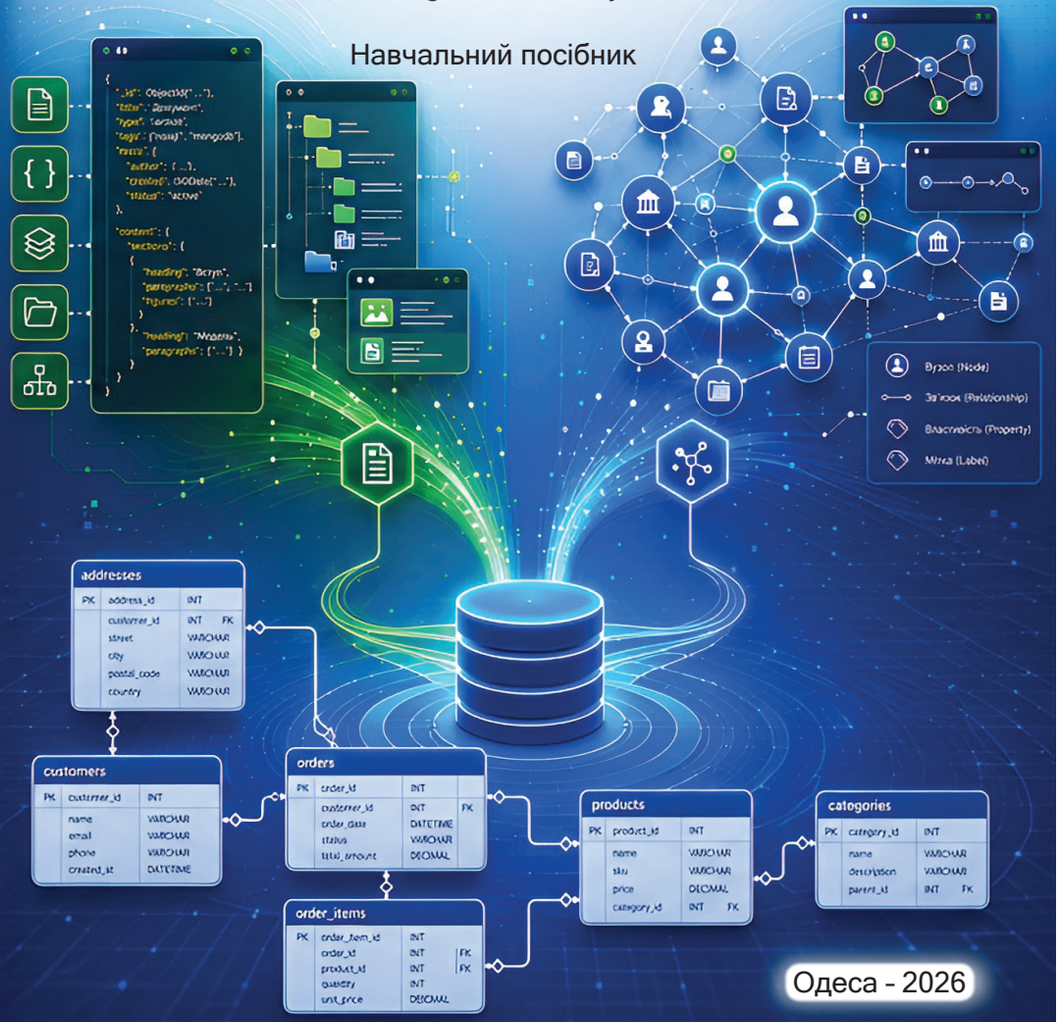
Малахов Є.В. NoSQL моделі та бази даних

МАЛАХОВ Є.В.

NoSQL моделі та бази даних

реалізація засобами MongoDB та Neo4j

Навчальний посібник



Одеса - 2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ І. І. МЕЧНИКОВА
ФАКУЛЬТЕТ МАТЕМАТИКИ, ФІЗИКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА МАТЕМАТИЧНОГО ЗАБЕЗПЕЧЕННЯ КОМП'ЮТЕРНИХ СИСТЕМ

Є.В. Малахов

NoSQL моделі та бази даних

реалізація засобами MongoDB та Neo4j

НАВЧАЛЬНИЙ ПОСІБНИК

ОДЕСА
«Маджента»
2026

УДК 004.652.2:004.658.2
М 181

Автор:
Малахов Є.В., д-р техн. наук, професор, завідувач кафедри математичного забезпечення комп'ютерних систем Одеського національного університету імені І. І. Мечникова

Рецензенти:
Арсірій О.О., д-р техн. наук, професор, завідувачка кафедри інформаційних систем Національного університету «Одеська політехніка»
Гожий О.П., д-р техн. наук, професор, професор кафедри інтелектуальних інформаційних систем Чорноморського національного університету імені Петра Могили

Рекомендовано до видання Вченою радою ОНУ імені І. І. Мечникова.
Протокол № 11 від 23 червня 2026 року

Малахов Є.В.

М 181 NoSQL моделі та бази даних реалізація засобами MongoDB та Neo4j : навч. посіб. / Є.В. Малахов. – Одеса : Маджента, 2026. – 219 с.
ISBN 978-617-7980-08-6

Навчальний посібник містить інформацію щодо особливостей нереляційних баз даних, розглянуто основні типи і структури, які притаманні нереляційним моделям даних, а також СУБД, що підтримують такі моделі, зокрема: документо-орієнтована модель, призначена для зберігання ієрархічних структур даних, з ілюстрацією реалізації в СУБД MongoDB, та графова модель з ілюстрацією реалізації в СУБД Neo4j. Наприкінці окремих розділів наведено питання для самоперевірки, завдання для самостійної роботи, а також наскрізних демонстраційних прикладів і взаємопов'язаних практичних завдань, що дозволяють послідовно простежити процес проєктування та реалізації баз даних різних типів на основі єдиної предметної області.

Для здобувачів вищої освіти з ІТ-спеціальностей.

УДК 004.652.2:004.658.2

ISBN 978-617-7980-08-6

Зміст

Вступ

При сучасних темпах розвитку інформаційних технологій, стрімкому зростанні обсягів даних та ускладненні інформаційних систем особливого значення набуває ефективне зберігання, обробка та аналіз інформації. Традиційні реляційні бази даних протягом багатьох років залишалися основним інструментом побудови інформаційних систем, однак розвиток веб-технологій, хмарних сервісів, соціальних мереж, систем аналітики та обробки великих даних призвів до появи нових вимог до моделей даних і систем управління базами даних.

Одним із результатів такого розвитку стало формування класу нереляційних баз даних, об'єднаних загальним терміном NoSQL. На відміну від класичних реляційних систем, NoSQL-підхід орієнтується на гнучкість моделей даних, горизонтальне масштабування, високу продуктивність та ефективну роботу зі слабкоструктурованими або взаємопов'язаними даними.

Сучасні NoSQL-системи охоплюють різні моделі даних, серед яких найбільш поширеними є: сховища типу «ключ-значення»; стовпцеві бази даних; документо-орієнтовані бази даних; графові бази даних.

Кожна з цих моделей має власні концептуальні особливості, переваги та обмеження, а також орієнтована на різні класи задач і предметних областей.

У даному навчальному посібнику основну увагу приділено документо-орієнтованій та графовій моделям даних, а також їх реалізації засобами СУБД MongoDB та Neo4j. Вибір саме цих систем обумовлений їх широким практичним використанням, зрілістю екосистем, доступністю інструментів розробки та наочністю демонстрації сучасних підходів до моделювання даних.

У посібнику розглянуто:

- основні принципи та особливості NoSQL-підходу;
- теорему CAP та підходи до узгодженості даних;
- документо-орієнтовану модель даних і СУБД MongoDB;
- графову модель даних і СУБД Neo4j;
- підходи до моделювання даних;
- принципи побудови розподілених NoSQL-систем;
- приклади створення, модифікації та обробки даних засобами MongoDB і Neo4j.

Особливістю посібника є поєднання теоретичного матеріалу з практичними прикладами, що базуються на єдиній предметній області «Навчальний процес». Це дозволяє продемонструвати відмінності між реляційною, документною та графовою моделями даних, а також показати особливості реалізації однакових задач у різних типах СУБД.

Для закріплення матеріалу наприкінці окремих розділів наведено питання для самоперевірки та завдання для самостійної роботи.

Навчальний посібник призначений для здобувачів вищої освіти ІТ-спеціальностей, аспірантів, а також усіх, хто цікавиться сучасними підходами до побудови інформаційних систем і технологіями нереляційних баз даних.

Передумови розвитку нереляційних моделей та баз даних

У сфері розробки програмного забезпечення реляційні бази даних за умовчанням були основним способом зберігання даних у серйозних проектах, особливо у галузі промислових застосунків [1]. Якщо ви – архітектор, який починає новий проект, то швидше за все виберете реляційну базу.

Це зумовлено низкою переваг РСУБД (таблиця 1)

Таблиця 1 – Переваги реляційних СУБД

№	Критерій	Опис
1	Сувора схема даних (Schema-on-Write)	– Чітка структура: Дані повинні відповідати заздалегідь визначеній схемі (таблиці, стовпці, типи даних, зв'язки, обмеження). Це забезпечує цілісність, узгодженість та передбачуваність даних. – Валідація на рівні БД: Обмеження (NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK) гарантують коректність даних до запису.
2	Гарантії ACID	Надійність транзакцій (ACID): Атомарність (Atomicity), Узгодженість (Consistency), Ізольованість (Isolation), Довговічність (Durability) – фундаментальні властивості для критично важливих операцій (банківські транзакції, фінансові системи). Гарантують, що операції або завершаться повністю, або зовсім не відбудуться, навіть при збоях.
3	Потужна мова запитів (SQL)	– Стандартизація: SQL – потужна, універсальна і добре стандартизована мова. – Складні JOIN-операції: ідеально підходить для виконання складних запитів, що поєднують дані з багатьох зв'язаних таблиць за допомогою операцій JOIN. – Агрегація та аналіз: багатий набір функцій для групування, агрегації (GROUP BY, SUM, AVG, COUNT тощо) та складної фільтрації даних.
4	Узгодженість даних (переважно Strong Consistency)	Точні дані: гарантує, що після завершення операції запису всі наступні операції читання побачать найактуальніше значення даних. Важливо для систем, де точність даних є критичною.
5	Зрілість та екосистема	– Перевірені часом: технології існують десятиліттями (PostgreSQL, MySQL, Oracle, SQL Server), дуже стабільні та надійні. – Багатий інструментарій: величезна кількість інструментів для адміністрування, моніторингу, проектування, ORM (Object-Relational Mapping), інтеграції з BI-системами.

Проте реляційні бази даних, попри всі переваги, не є універсальним рішенням [3].

Для розробників застосунків основною проблемою була втрата відповідності (impedance mismatch): різниця між реляційною моделлю та структурами даних у пам’яті прикладної програми. Реляційна модель подає дані у вигляді відношень і кортежів. Усі операції математично елегантної реляційної алгебри, реалізовані мовою SQL, відповідно до властивості замкнутості отримують та повертають відношення.

Використання відношень забезпечує певну елегантність та простоту, але одночасно накладає деякі обмеження, сформульовані, наприклад, у вимогах нормальних форм. Зокрема, відповідно до вимог 1НФ, значення реляційного кортежу повинні бути атомарні, тобто. не повинні містити складених або вкладених структур (записів, масивів тощо). Це обмеження не відноситься до структур даних, що знаходяться в пам'яті, які можуть бути набагато складнішими за відношення. В результаті, якщо ви захочете використати складну структуру даних, що знаходяться в пам'яті, то маєте перетворити реляційне подання для зберігання на диску. У результаті виникає втрата відповідності – два різні подання, що вимагають трансляції (рис. 1).

Втрата відповідності – основне джерело неприємностей для розробників застосунків. У 1990-х роках багато хто вважав, що це врешті-решт призведе до заміни РБД на бази, що реплікують структури даних, що зберігаються в пам'яті, на диск. Це десятиліття відзначалося зростанням об'єктно-орієнтованих мов програмування, а за ними з'явилися об'єктно-орієнтовані бази даних. Очікувалося, що вони домінуватимуть у галузі розробки програмного забезпечення у новому тисячолітті.

Проте в той час як об'єктно-орієнтовані мови зайняли лідируючі позиції в програмуванні, об'єктно-орієнтовані бази даних зникли в темряві, Реляційні бази даних відповіли на виклик, посиливши свою роль як інтегруючий механізм, підтриманий практично стандартною мовою для маніпулювання даними (SQL) і зростаючою професійною диференціацією.

Поява комбінованих РСУБД пом'якшила, але не усунула проблему втрати відповідності.

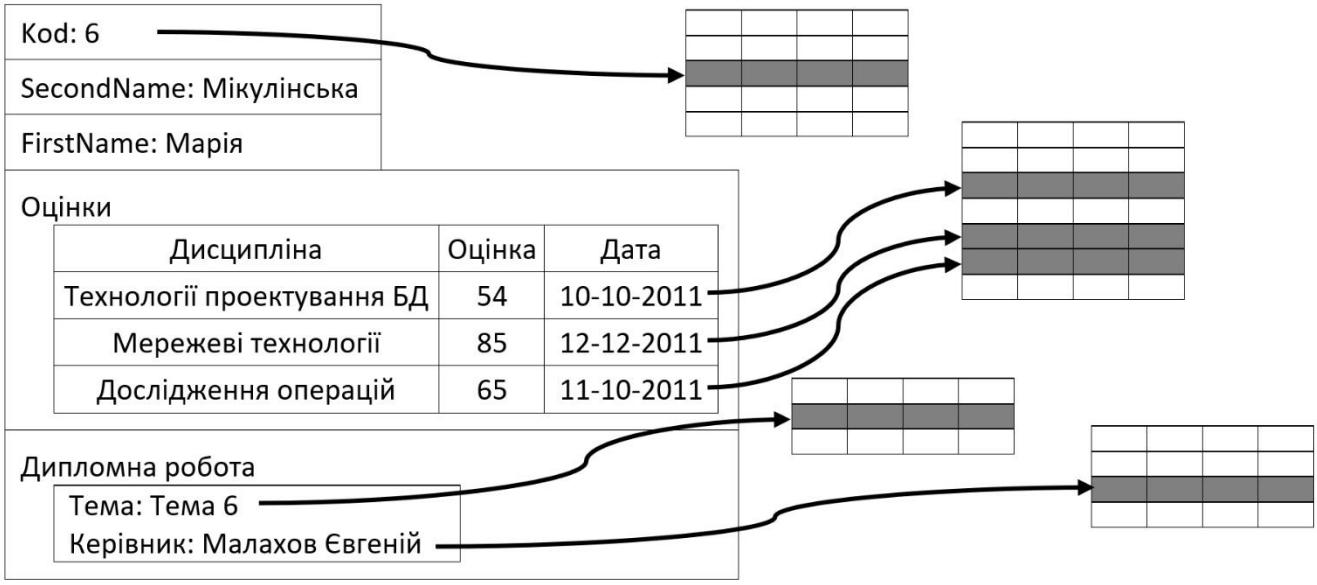


Рис. 1 – Картка студента, яка в інтерфейсі користувача виглядає як єдина агрегована структура, в реляційній базі даних поділяється на множину рядків з багатьох таблиць

Крім того, РСУБД мають й інші фундаментальні обмеження, які стають критичними у певних сценаріях і змушують шукати альтернативи (таблиця 2).

Таблиця 2 – Обмеження реляційних СУБД

№	Обмеження	Проблема	Наслідки
1	Складність та вартість горизонтального масштабування (Scale-Out)	РСУБД спочатку проектувалися для вертикального масштабування (посилення сервера: більше CPU, RAM, дисків). Горизонтальне масштабування (додавання незалежних серверів-нодів) вкрай складно, дорого і часто потребує екзотичних рішень.	При зростанні навантаження або обсягу даних до екстремальних рівнів (Big Data, високонавантажені веб-сервіси) вартість та складність підтримки кластера РСУБД стають непомірно високими. Шардування (ручний або автоматичний поділ даних по нодах) – це біль, що порушує ACID, ускладнює запити та адміністрування.
2	Продуктивність складних JOIN на великих обсягах	Хоча JOIN – потужний інструмент, його виконання на дуже великих таблицях, розташованих на різних фізичних нодах (у разі шардування), стає вкрай ресурсомістким та повільним.	Для досягнення високої продуктивності у розподілених системах часто доводиться денормалізовувати дані (дублювати інформацію у різних таблицях), що порушує принципи нормалізації РСУБД, збільшує обсяг сховища та складність підтримки узгодженості.
3	Неефективність для специфічних шаблонів доступу	РСУБД – універсальні інструменти, але вони можуть бути надмірними або неоптимальними для вузьких завдань: – прості операції "ключ-значення": навантаження по парсингу SQL, планувальнику запитів, менеджеру транзакцій для простого SELECT * FROM table WHERE id =? занадто велика порівняно зі спеціалізованими сховищами типу Redis; – дуже "широкі" або розріджені дані: зберігання мільйонів стовпців або рядків, де більшість значень NULL, є неефективним у табличній моделі; – складні ієрархії або графи: зберігання та запити до глибоко вкладених структур або граф з безліччю зв'язків вимагають складних SQL-запитів (рекурсивні CTE, множинні JOIN) і працюють повільніше, ніж у документних або графових БД.	

4	Жорсткість схеми даних (Schema Rigidity)	Необхідність строго визначати структуру даних перед записом (Schema-on-Write). Будь-яка зміна схеми (додавання/ видалення стовпця, зміна типу) вимагає операції ALTER TABLE, яка на великих таблицях може бути: – блокуючою – заморожує запис (а іноді і читання) в таблиці на час виконання. – довгою – на гігантських таблицях триває годинами або цілодобово. – ризикованою – помилка в міграції може призвести до втрати даних або простою.	Наслідки: сильно уповільнює розробку в середовищах з вимогами, що швидко змінюються, або при роботі з даними, структура яких заздалегідь невідома або неоднорідна (логи, користувальницький контент).
5	Обмеження Теореми CAP (у розподілених системах)	У розподіленій системі неможливо одночасно гарантувати всі три властивості – Узгодженість (Consistency), Доступність (Availability), Стійкість до поділу (Partition tolerance). У класичних ACID конфігураціях РСУБД пріоритет надається узгодженості (C) перед доступністю (A) при мережевих розділах (P).	При розриві зв'язку між нодами кластера РСУБД частина системи може бути недоступною для запису (чи навіть читання), щоб зберегти узгодженість. Для глобальних програм, що вимагають 100% доступності, це неприйнятно.
6	Висока вартість володіння (TCO)	Ліцензії комерційних РСУБД (Oracle, SQL Server Enterprise) можуть бути дуже дорогими, особливо для великих кластерів. Навіть для Open Source (PostgreSQL, MySQL) вартість потужного заліза для вертикального масштабування, висококваліфікованих DBA для налаштування, оптимізації та адміністрування складних кластерів дуже суттєва.	Для стартапів або проектів з обмеженим бюджетом та екстремальними вимогами до масштабованості TCO РСУБД може бути непідйомним.

Реляційні бази даних продовжують домінувати в галузі промислової обробки даних у 2000-х роках, але протягом першого десятиліття в стінах їхньої неприступної фортеці почали з'являтися тріщини.

Від РСУБД відмовляються не тому, що вони «погані», а тому, що їх архітектурні принципи (строга схема, ACID, реляційна модель) стають непереборною перешкодою при досягненні певних масштабів (Big Data, глобальна розподіленість), вимог до швидкості розробки (гнучка схема) або доступності (CAP). Це вимушений захід, коли переваги РСУБД переважуються їхніми фундаментальними обмеженнями у конкретному контексті.

У спробах подолати ці обмеження з’являлися інші технології баз даних, наприклад об'єктні бази даних у 1990-х роках, але ці альтернативи не набули широкого поширення. Після тривалого переважного переважання поява сильного конкурента як баз даних NoSQL стало сюрпризом.

Дійсно [2], проекти класу XTP (*extreme transaction processing* – високонавантажені застосунки) пред'являють нові вимоги до СУБД – сьогодні необхідні технології, що забезпечують низьку вартість масштабування та обробки великих обсягів даних. До таких технологій відносять СУБД класу NoSQL [1, 3], що обіцяють розробникам високу швидкість внесення змін до застосунків, низькі витрати на масштабування, інструменти обробки та зберігання великих обсягів даних, а також високу швидкість виконання на відносно недорогому обладнанні. NoSQL – це сімейство підходів до зберігання даних, відмінних від класичної реляційної моделі. Їх поєднують гнучкість схеми (частіше схема валідується на записи, а не фіксується заздалегідь DDL), орієнтація на горизонтальне масштабування та практична націленість на доступність та швидкість розробки. У базах NoSQL реалізується відмінна від традиційної (один сервер баз даних для декількох застосунків) парадигма, при якій одному застосунку або модулю програми пропонується окреме рішення щодо роботи з даними. За своєю суттю архітектура рішень NoSQL орієнтована на боротьбу або з великим обсягом даних, або їх підвищеною складністю [4].

Відмінні риси NoSQL СУБД – нереляційні моделі даних, прості API або протоколи доступу, здатність до горизонтального масштабування на вимогу для деякого набору операцій на багатьох серверах, розподілене зберігання даних, ефективне використання розподілених індексів та пам'яті для запитів, досить вільне поводження з такими непорушними для традиційних СУБД речами, як транзакції. Спільним для NoSQL-проектів є компроміси щодо взаємно суперечливих вимог – наприклад, ухиляння від підтримки стандартних правил для забезпечення транзакційної цілісності *ACID* на користь горизонтальної масштабованості. Відмова від подібних вимог була неможлива і є догмою для традиційних СУБД. У розподілених системах неможливо одночасно максимізувати узгодженість, доступність і стійкість до поділу мережі, проте у ряді конкретних випадків можливі варіанти компромісу.

Інше джерело компромісів – характер даних розв'язуваної задачі. Саме тут важлива вимога до технології, яка має максимально використовувати особливості предметної області. Наприклад, якщо можна розпаралелити обробку даних і використовувати принцип *shared nothing* («без поділу ресурсів»), потрібно це ефективно застосовувати як для зберігання, так і для виконання запитів. В цьому випадку треба будувати модель зберігання та розподілу даних, яка спирається на цю можливість, проте в реляційних базах свободи вибору майже немає і доводиться використовувати те, що є, скажімо, не можна зберігати дані по колонках на різних серверах (рис. 2). При цьому, на відміну від традиційних СУБД, технологія NoSQL дає розробнику більше свободи при врахуванні природних особливостей проектної задачі, які можуть бути використані, наприклад, горизонтального масштабування. Однак розробник несе більше відповідальності за прийняті архітектурні рішення.

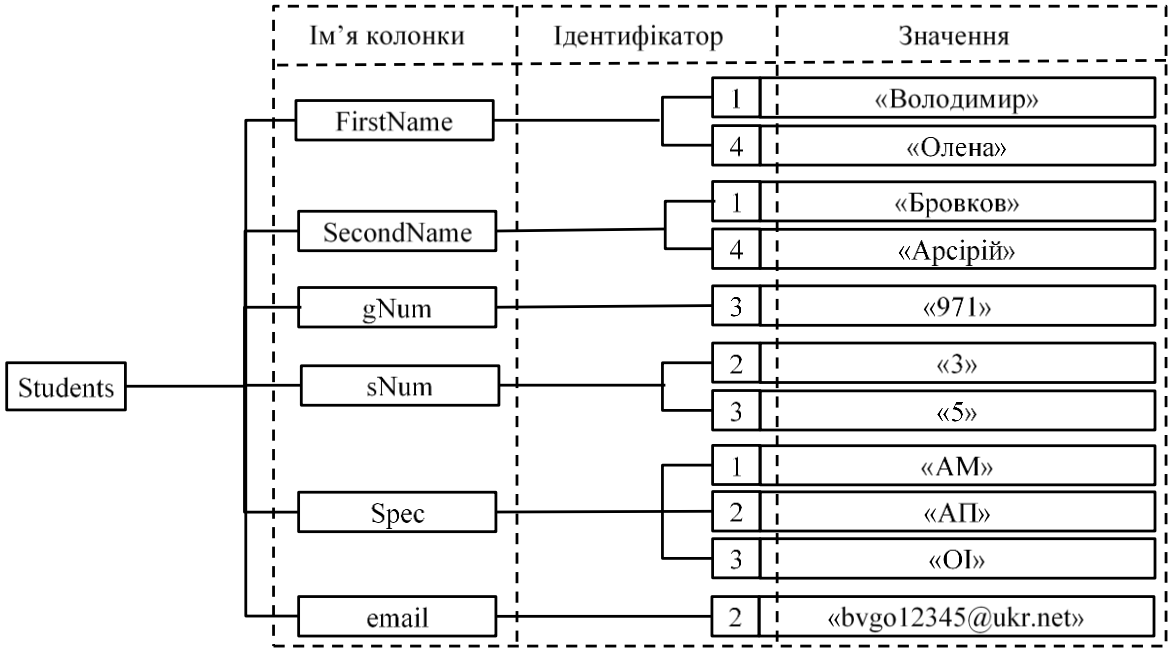


Рис. 2 – Реализація постовбцевого зберігання

Отже, NoSQL БД мають ряд переваг перед реляційними (таблиця 3).

Таблиця 3 – Переваги NoSQL СУБД

№	Критерій	Опис
1	Динамічна або керована схема (flexible schema – в MongoDB) чи гнучка схема даних (Schema-on-Read – в Hadoop)	1. Динамічна структура: дозволяє зберігати дані з різною структурою в одній «колекції» чи «таблиці». Легко адаптувати структуру даних під мінливі вимоги програми без дорогих операцій ALTER TABLE. 2. Підтимка швидкої ітерації: скорочує цикл внесення змін у структуру даних, особливо на ранніх етапах або під час роботи з напівструктурованими/неструктурованими даними (JSON, документи, графи).
2	Горизонтальне масштабування (Scale-Out)	1. Розподіл: легше масштабувати для обробки величезних обсягів даних і високих навантажень, додаючи недорогі сервери (ноди) в кластер. Шардування даних за нодами – ключовий принцип. 2. Висока доступність: Реплікація даних між нодами забезпечує стійкість до відмови – при падінні однієї ноди, її роль беруть на себе інші.
3	Підтримка напівструктурованих та неструктурованих даних	Природне зберігання: ідеально підходить для JSON, XML, логів, текстів, даних сенсорів, де структура може бути мінливою або складною.
4	Модель узгодженості (переважно Eventual Consistency)	Доступність і стійкість до поділу (CAP Theorem): часто жертвують суворою узгодженістю (Strong Consistency) на користь високої доступності (Availability) та стійкості до поділу мережі (Partition tolerance). Це дозволяє системі залишатися доступною для запису та читання навіть при збоях зв'язку між нодами, з гарантією, що дані зрештою стануть узгодженими (коли мережа відновиться).
5	Георозподіленість	Глобальні програми: багато NoSQL БД спочатку спроектовані для легкого розгортання та реплікації даних у кількох географічних регіонах, знижуючи затримку для користувачів у всьому світі.

6	Висока продуктивність за певних навантажень	1. Спеціалізація: оптимізовані для конкретних патернів доступу: – документні (MongoDB, Couchbase) – швидкі читання/запис по ключу або в межах документа. – ключ-Значення (Redis, DynamoDB) – екстремально швидкі операції з унікального ключа (кеш, сесії). – стовпцеві (Cassandra, HBase) – ефективне читання/запис великих обсягів даних по стовпцях, агрегація часових рядів. – графові (Neo4j) – блискавичне виконання складних запитів до пов'язаних даних (соціальні зв'язки, рекомендації). 2. Відсутність реляційних JOIN як центрального механізму моделі даних: часто відмова від складних JOIN на користь денормалізації чи вкладених структур веде до збільшення швидкості певних запитів.

Незважаючи на різноманітність NoSQL-проектів, досі немає жодного, який можна було б назвати універсальною та всеосяжною NoSQL-платформою. Тому, якщо у розробників виникла ідея скористатися NoSQL-підходами в наступному проекті, то перш за все доведеться відповісти на цілу низку питань і вирішити безліч ризиків, наприклад: яку модель даних вибрати; наскільки стабільна та відпрацьована обрана технологія; наскільки серйозними будуть зміни в коді у разі спроби зміни технології на іншу; чи буде мова запитів досить повною та технологічною для задоволення проектних вимог. Окремо варто відзначити, що багато NoSQL-технологій було створено спеціально в рамках конкретного проекту і є ймовірність того, що, закриваючи вимоги та завдання початкового проекту, вони можуть виявитися не дуже підходящими для іншого.

У результаті ***вибір моделі даних і СУБД залежить від завдання***!

1. Вибирайте РСУБД, коли: критично важливі транзакції ACID, сувора цілісність даних, складні запити з JOIN, узгодженість у реальному часі. Приклади: фінансові системи, ERP, CRM, системи обліку, де дані строго структуровані та взаємопов’язані.
2. Вибирайте NoSQL, коли: потрібні обробка великих обсягів даних, горизонтальне масштабування, гнучка схема даних, робота з напівструктурованими та неструктурованими даними, а висока доступність і стійкість до збоїв є важливішими за миттєву узгодженість. Приклади: каталоги товарів, профілі користувачів, системи IoT/телеметрії, кеші, рекомендацій, соціальні графи, аналітика великих даних. Приклади: каталоги товарів, профілі користувачів, системи IoT/телеметрії, кеші, рекомендацій, соціальні графи, аналітика великих даних.

Сучасні тренди: часто використовується гібридний підхід (*Polyglot Persistence*) [1], де в одному застосунку застосовують різні типи БД, оптимальні для конкретних підзадач. Також з’являються «NewSQL» БД, які намагаються об’єднати масштабованість NoSQL з гарантіями ACID та SQL-інтерфейсом РСУБД.

Теорема CAP

Визначення

Теорема CAP визначає фундаментальні обмеження, із якими стикаються розподілені інформаційні системи [3, 5]. Вона стверджує, що

Розподілена система зберігання даних не може одночасно гарантувати три властивості: узгодженість (Consistency), доступність (Availability) та стійкість до розділення мережі (Partition tolerance) [6].

Ці властивості визначаються так:

- **Узгодженість (Consistency)** означає, що після виконання операції запису всі наступні операції читання повертають однакове, актуальне значення даних.
- **Доступність (Availability)** означає, що система завжди повертає відповідь на запит, якщо хоча б один вузол системи працює.
- **Стійкість до розділення мережі (Partition tolerance)** означає, що система продовжує функціонувати навіть у випадку тимчасової втрати зв'язку між окремими вузлами мережі.

Відповідно до теореми CAP, у розподіленій системі можна гарантувати лише ***два з трьох*** зазначених властивостей. Тому під час проєктування системи необхідно обирати, які характеристики є пріоритетними.

На практиці можливі три основні варіанти компромісу (таблиця 4):

Таблиця 4 – Комбінації властивостей розподілених систем (теорема CAP)

Тип CAP	Характеристика
CP (Consistency + Partition tolerance)	система зберігає узгодженість даних навіть у разі розділення мережі, але може тимчасово втрачати доступність.
AP (Availability + Partition tolerance)	система залишається доступною для запитів, але може тимчасово повертати неузгоджені дані.
CA (Consistency + Availability)	система забезпечує узгодженість і доступність, але не розрахована на роботу в умовах розділення мережі (тобто фактично не є повністю розподіленою).

На практиці різні СУБД реалізують різні компроміси між цими властивостями [1, 6]. Конкретні приклади таких архітектур будуть розглянуті у наступних розділах.

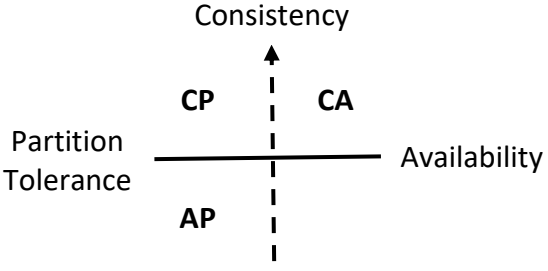


Рис. 3 – Трикутник CAP

У більшості сучасних розподілених систем властивість ***Partition tolerance*** вважається обов'язковою, оскільки мережеві збої є неминучими. Тому практичний вибір зазвичай здійснюється між ***узгодженістю*** та ***доступністю*** (рис. 3).

Узгодженість у підсумку (Eventual Consistency)

Одним зі способів досягнення високої доступності у розподілених системах є використання моделі ***узгодженості у підсумку (eventual consistency)*** [3, 14].

У цьому випадку система дозволяє обробляти запити навіть тоді, коли окремі вузли мають різні версії даних. Зміни, внесені в одному вузлі, поступово поширюються на інші вузли в фоновому режимі. У результаті система може бути тимчасово неузгодженою, але через певний час усі копії даних стають однаковими.

Класичним прикладом такої поведінки є система DNS, у якій зміни інформації про домени можуть поширюватися мережею протягом певного часу, але кожен сервер при цьому залишається доступним для запитів.

Теорема CAP на практиці

Теорема CAP відіграє важливу роль під час проєктування розподілених баз даних [3].

Різні системи керування базами даних роблять різні архітектурні вибори:

- деякі системи орієнтуються на ***узгодженість даних***;
- інші віддають перевагу ***доступності та масштабованості***;
- окремі системи дозволяють ***налаштовувати рівень узгодженості на рівні запиту***.

Таким чином, теорема CAP не визначає «кращу» архітектуру, але допомагає розуміти, які компроміси лежать в основі конкретної системи і які обмеження виникають у розподілених середовищах.

ACID та BASE: два підходи до узгодженості даних

У класичних реляційних системах управління базами даних транзакції описуються набором властивостей **ACID** (таблиця 5). Ця модель забезпечує високу надійність та строгий контроль узгодженості даних [7, 14].

Таблиця 5 – Властивості ACID

Властивість	Характеристика
Atomicity (атомарність)	транзакція виконується повністю або не виконується зовсім.
Consistency (узгодженість)	після завершення транзакції база даних переходить з одного коректного стану в інший.
Isolation (ізольованість)	паралельні транзакції не впливають одна на одну.
Durability (довговічність)	результати завершеної транзакції зберігаються навіть у разі збоїв системи.

Ці властивості забезпечують строгий контроль над даними, але можуть обмежувати масштабованість системи у розподілених середовищах.

У багатьох NoSQL-системах використовується альтернативний підхід, який часто описують абревіатурою **BASE** (таблиця 6).

Таблиця 6 – Властивості BASE

Властивість	Характеристика
Basically Available	система прагне залишатися доступною для запитів навіть у випадку часткових відмов.
Soft State	стан системи може тимчасово змінюватися без виконання нових операцій через фонову реплікацію та синхронізацію.
Eventual Consistency	дані можуть бути тимчасово неузгодженими, але з часом всі вузли системи приходять до однакового стану.

Модель BASE дозволяє досягти високої масштабованості та доступності у розподілених системах за рахунок ослаблення вимог до негайної узгодженості [1].

Таким чином, ACID і BASE відображають *два різні підходи до забезпечення узгодженості даних* (таблиця 7) [4]:

– ACID орієнтована на строгий контроль транзакцій;
– BASE орієнтована на масштабованість і доступність у розподілених системах.

Таблиця 7 – ACID vs BASE

ACID	BASE
строгі транзакції	ослаблена узгодженість
централізовані системи	розподілені системи
негайна узгодженість	eventual consistency
традиційні РСУБД	різноманітні NoSQL системи

На практиці сучасні системи часто використовують *комбіновані підходи*, поєднуючи елементи ACID і BASE залежно від архітектури системи та вимог до узгодженості даних.

Сучасні NoSQL моделі та СУБД

Сховища ключ-значення

Модель Key-Value [3] є мінімалістичним словником «ключ → значення», де значення може бути непрозорим для бази – найпростіша з усіх моделей. Як випливає з назви, ця модель зіставляє ключам відповідні значення як словник (або хеш-таблиця) в будь-якій популярній мові програмування. Деякі реалізації допускають як значення складові типи даних, наприклад хеші або списки. Прикладом KV-сховища можна вважати файлову систему, якщо розглядати шлях до файлу як ключ, яке вміст – як значення. Оскільки від подібної моделі потрібно так мало, бази даних цього типу можуть демонструвати неймовірно високу продуктивність, але в загальному випадку марні, коли потрібні складні запити і агрегування.

Як і у разі реляційних СУБД, є багато продуктів із відкритим вихідним кодом. Такі системи, як Redis, Amazon DynamoDB (в KV-режимі), Aerospike та Riak KV, надають дуже швидкий доступ по ключу і добре підходять для кешів, сесій та лічильників, проте вимагають продуманої денормалізації на рівні програми.

Riak – це не просто сховище ключів та значень. Система підтримує такі веб-технології, як HTTP та REST. Це точний повтор системи Dynamo, що використовується компанією Amazon, з деякими додатковими функціями, наприклад, векторні часові мітки для розв'язання конфліктів. Значенням Riak може бути що завгодно: простий текст, XML-документ, зображення тощо, а зв'язок між ключами описуються іменованими структурами, які називаються посиланнями (links).

Система ***Redis*** підтримує складові типи даних, зокрема відсортовані множини та хеші, а також базові комунікаційні засоби, у тому числі публікацію-підписку та блокуючі черги. Вона також має в своєму розпорядженні один з найрозвиненіших механізмів запитів серед усіх KV -СУБД. А за рахунок кешування операцій запису в пам'яті Redis досягає вражаючої продуктивності ціною підвищеного ризику втрати даних у разі апаратного збою. Завдяки цій характеристиці вона може бути непоганим засобом кешування некритичних даних, а також брокером повідомлень.

Наприклад, створення ключів для Redis – KV-модель для БД lectures:

ключ = student:<kod>,

значення = JSON/рядок з «карткою студента» (мінімальним профілем)

```
# Профіль студента (ключ -> JSON)
SET student:1 "{\"kod\":1,\"fullName\":\"Бровков Володимир
Георгійович\",\"spec\":\"КМ\",\"email\":\"bvgo12345@ukr.net\"}"

# Швидкий доступ до email по коду (ключ -> рядок)
SET student:1:email "bvgo12345@ukr.net"
```

```
# Приклад «списку оцінок» як окремого ключа (ключ -> JSON-масив)
SET student:1:ratings "[{\"discipline\":\"Технології проектування
БД\",\"mark\":82},{\"discipline\":\"Мережеві технології\",\"mark\":10}]"
```

Стовпцеві бази даних

Бази сімейства колонок [3] (wide column / column family) (стовпцеві, або орієнтовані на зберігання даних по стовпцях бази даних) отримали свою назву завдяки одному істотному аспекту дизайну: дані, що належать одному стовпцю (в сенсі двовимірних таблиць) зберігаються поруч. Такі СУБД організують дані навколо ключа-розділювача та сімейств колонок, дозволяючи ефективно читати та писати дуже великі обсяги за заздалегідь відомими шаблонами доступу. Навпаки, у рядкових базах даних (до яких відносяться реляційні) поруч зберігаються дані, що належать одному рядку. Переваги такого проектного рішення в наступному: у стовпцевих базах даних додавання нового стовпця обходиться дешево і проводиться рядково, у кожному рядку набір стовпців може бути різним, можливо навіть, що в деяких рядках стовпців взагалі немає, і, отже, таблиця може бути розріджена без накладних витрат на зберігання null значень. З точки зору структури, стовпцеві бази даних займають проміжне положення між реляційними та KV СУБД.

На ринку стовпцевих баз даних конкуренція менша, ніж серед реляційних СУБД та KV- сховищ. Найбільш популярні системи Apache Cassandra, Apache HBase та ScyllaDB, які демонструють високу стійкість до відмов та лінійне масштабування, але вимагають від запитів акуратного моделювання.

З усіх нереляційних СУБД ***HBase*** найближче до реляційної моделі. HBase спроектована за зразком системи Google BigTable, побудована на базі Hadoop (механізм *map-reduce – розподіл-редукція)* і призначена для горизонтального масштабування на кластерах, складених зі стандартного обладнання. HBase дає суворі гарантії несуперечності даних і надає таблиці, що складаються з рядків і стовпців, – шанувальникам SQL це доведеться до вподоби. Готова підтримка версіонування та стиску ставить цю СУБД на перше місце в категорії «великі дані».

У термінології ***Apache Cassandra*** застосунок працює з простором ключів (keyspace), що відповідає поняттю схеми бази даних (database schema) у реляційній моделі. У цьому просторі ключів можуть бути кілька колонкових сімейств (column family), що відповідає поняттю реляційної таблиці. У свою чергу колонкові сімейства містять колонки (column), які об’єднуються за допомогою ключа (row key) у записі (row). Колонка складається з трьох частин: імені (column name), мітки часу (timestamp) та значення (value). Колонки у межах запису впорядковані. На відміну від реляційної БД, жодних обмежень на те, щоб записи (а в термінах БД це рядки) містили колонки з такими ж іменами, як і в інших записах – немає.

Наприклад, Wide-column (Cassandra/HBase) для БД lectures:
«широка» таблиця оцінок,
де ключ партиції – студент,
а кластеризація – дисципліна/дата

```
-- Keyspace
CREATE KEYSPACE IF NOT EXISTS lectures
WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 1};

USE lectures;

-- Студенти (вузька таблиця-довідник)
CREATE TABLE IF NOT EXISTS students (
  kod int PRIMARY KEY,
  secondName text,
  firstName text,
  patronymic text,
  spec_code text,
  email text
);

-- Оцінки: широка таблиця під читання «всі оцінки студента»
CREATE TABLE IF NOT EXISTS ratings_by_student (
  student_kod int,
  discipline text,
  rating_date date,
  mark int,
  PRIMARY KEY ((student_kod), discipline, rating_date)
) WITH CLUSTERING ORDER BY (discipline ASC, rating_date DESC);

-- Індекс (якщо треба шукати студентів по spec_code)
CREATE INDEX IF NOT EXISTS idx_students_spec ON students(spec_code);
```

Документо-орієнтовані бази даних

У документоорієнтованих [3], або просто документних базах даних, що очевидно, зберігаються документи. Документ – це свого роду аналог хешу, у якому є поле унікального ідентифікатора, а значення можуть виступати дані довільного типу, зокрема інші хеші. Документи можуть містити вкладені структури і мають високу гнучкість, що робить їх придатними для застосування в різних предметних областях. Система накладає небагато обмежень на вхідні дані за умови, що вони задовольняють базові вимоги до представленості у вигляді документа. У різних документних базах даних застосовуються різні підходи до індексування, формулювання довільних запитів, реплікації, забезпечення узгодженості та інших аспектів.

⚠ *Для правильного вибору системи необхідно добре розуміти ці відмінності та їх вплив на конкретний сценарій використання.*

Цей клас зручний, коли агрегати даних складаються в один документ. Тобто, документна модель ефективна тоді, коли основні операції виконуються в межах одного агрегату. Проте у випадках, коли домінують запити до довгих ланцюжків взаємопов'язаних сутностей, денормалізація стає неефективною або призводить до дублювання даних.

З іншого боку, якщо модель предметної області визначається не вкладеністю, а зв'язками між великою кількістю автономних об'єктів (соціальні мережі, рекомендаційні системи, мережі постачань, графи залежностей), документна модель потребує або складних операцій $lookup (аналог операції з'єднання в реляційних БД), або перенесення логіки обходу об'єктів на рівень прикладного коду (застосунку).

До представників відносяться MongoDB, Couchbase, CouchDB та Azure Cosmos DB з API, сумісним із MongoDB.

MongoDB

СУБД MongoDB проектувалась для зберігання гігантських обсягів даних (mongo – частина слова humongous (об'єднання слів huge та monstrous) – «монструозний»). При налаштуванні сервера MongoDB дозволяє конфігурувати рівень узгодженості через параметри write concern та read concern, що визначають ступінь підтвердження запису та гарантії актуальності даних при читанні (до наступного оновлення). Ця особливість робить MongoDB привабливою альтернативою тим, хто має досвід роботи з РСУБД. Крім того, MongoDB підтримує атомарні операції читання-запису, у тому числі інкрементування значення та запити до вкладених документів. Завдяки JavaScript-подібному синтаксису запитів MongoDB підтримує як прості запити, так і складні завдання з розподілом-редукцією.

CouchDB

Система CouchDB розрахована на різноманітні сценарії розгортання від центру обробки даних до настільного ПК і навіть смартфона. Написана мовою Erlang, CouchDB може похвалитися дуже високим рівнем живучості: її файли даних майже неможливо пошкодити, при цьому CouchDB зберігає високу доступність навіть в умовах спорадичної втрати зв'язку або апаратних збоїв. Як і в MongoDB, мовою запитів у CouchDB є JavaScript. Подання описуються функціями map-reduce, які зберігаються як документи і реплікуються між вузлами як звичайні дані.

Графові бази даних

Переваги графових баз даних [3] яскраво виявляються під час обробки даних із великою кількістю зв'язків. Графова база складається з вузлів та зв'язків між ними. Як із вузлами, так і зі зв'язками можна асоціювати властивості – пари ключ-значення, у яких зберігаються дані. Справжня сила графових баз даних полягає у можливості обходу вузлів, дотримуючись зв'язків.

СУБД Neo4j, JanusGraph та Amazon Neptune дозволяють висловлювати складні патерни зв'язків, де реляційні JOIN стають громіздкими чи дорогими.

Neo4J

Операція, де інші бази даних часто здаються, – це обхід даних із посиланнями він або іншими складно влаштованими зв'язками. Саме тут переваги Neo4J проявляються у всьому блиску. Перевага графової бази даних у тому полягає, що забезпечується швидкий перегляд вузлів і зв'язків для пошуку потрібних даних. Такі бази часто використовуються у соціальних мережах та завоювали визнання за свою гнучкість.

Системи часових рядів

Ці системи [3] зосереджені на зберіганні та агрегуванні вимірів за часом, підтримують компресію та вниз вібрації та пропонують функції агрегування по вікнах. СУБД InfluxDB, VictoriaMetrics і TimescaleDB (хоча остання – розширення PostgreSQL) вирішують завдання телеметрії, моніторингу та аналітики подій із високою частотою.

Комбіновані системи

Пошукові та аналітичні документні движки …

… комбінують документну модель з інвертованими індексами для повнотекстового та фасетного пошуку [2]. Elasticsearch, OpenSearch і Solr забезпечують швидкий пошук, релевантність та агрегування, виступаючи в ролі спеціалізованого шару над вихідним сховищем.

Багатостороннє зберігання

Насправді різні бази даних часто використовують у поєднанні [1, 2]. Згодом все популярнішими стають комбінації різних баз даних, у яких сильні сторони кожної дозволяють створити екосистему, яка виявляється більш потужною, функціональною та надійною, ніж сума її частин. Цю практику, що отримала назву багатостороннє зберігання (polyglot persistence) ми намагалися застосовувати навіть у рамках кваліфікаційних робіт бакалаврів, створюючи та використовуючи гібридні інформаційні системи.

Питання для самоперевірки

(до розділів «Передумови розвитку нереляційних моделей та баз даних», «Сучасні NoSQL моделі та СУБД»)

1. Чим сучасні вимоги до інформаційних систем відрізняються від умов, у яких формувались класичні реляційні СУБД?
2. У чому полягають основні обмеження реляційної моделі даних при роботі з великими обсягами слабкоструктурованих даних?
3. Що означає горизонтальне масштабування і чому воно стало важливим для сучасних інформаційних систем?
4. У чому полягають переваги нереляційних баз даних?
5. Які компроміси між узгодженістю, доступністю та стійкістю до мережного розділення описує теорема CAP?
6. Чим підхід BASE відрізняється від ACID?
7. У яких випадках доцільно обирати реляційні чи нереляційні бази даних?
8. Які основні типи NoSQL-моделей даних розглядаються у сучасних СУБД?
9. У чому полягають концептуальні відмінності між сховищами «ключ–значення», стовпцевими, документними та графовими базами даних?
10. Для яких класів задач доцільно використовувати сховища типу «ключ–значення»?
11. У чому полягають особливості організації даних у стовпцевих базах даних?
12. Чому документно-орієнтовані бази даних добре підходять для роботи зі слабкоструктурованими даними?
13. Які переваги графових баз даних при роботі з взаємопов’язаними даними?
14. У чому полягає ідея комбінованих систем зберігання даних?

Документна модель даних та СУБД MongoDB

Документна модель є природнішою за реляційну, якщо в предметній області домінують складні вкладені об’єкти зі змінною структурою – наприклад, профілі, каталоги та налаштування, – документна модель позбавляє постійних міграцій схеми та полегшує розвиток [1]. У сценаріях подійного збору та журналування, де запис йде безперервним потоком та в режимі додавання (append-only), документне сховище забезпечує високу швидкість при помірній складності запитів. Під персоналізацію та кешування подань для UI/API зручно формувати документи так, щоб один запит закривав потреби екрана. Для помірних обсягів, де потрібний ще й текстовий або геопошук, вбудовані індекси можуть дозволити обійтися без додаткового стека.

Документ у цій моделі розглядається як агрегат, тобто як природна одиниця узгодженості та атомарності змін; саме на рівні документа СУБД (наприклад, MongoDB) гарантує атомарні операції. Замість жорсткого DDL застосовується валідація на рівні колекції: JSON Schema-подібні правила визначають обов’язкові поля, типи та діапазони значень. Денормалізація стає інструментом прискорення читання: дані, які часто читаються разом, містяться в один документ, щоб мінімізувати кількість з’єднань та проміжних вузлів (hops). Ключові запити визначають індексацію, тому проектування схеми починається з аналізу фільтрів, сортувань та частоти викликів API, а не з креслення ER-діаграм.

⚠ *Водночас початкове концептуальне моделювання (наприклад, у вигляді ER-діаграми) зберігає свою роль, але в такій СУБД вона динамічна (розглянемо наприкінці розділу).*

У розподілених системах доводиться вибирати баланс між узгодженістю, доступністю та стійкістю до поділу мережі. Насправді цей вибір у таких СУБД, як MongoDB, виражається через рівні write concern і read concern, і навіть через переваги читання з первинних чи з вторинних реплік [9]. SLA (Service Level Agreement – Угода про Рівень Обслуговування) за затримками та допустимими відхиленнями свіжості даних фіксує очікувану поведінку системи та дозволяють усвідомлено керувати ризиками.

Основи документної моделі даних

Концептуальними елементами документної моделі даних (на прикладі MongoDB) є такі [4]:

1. Сервер MongoDB концептуально аналогічний звичним серверам реляційних баз даних (PostgreSQL, Oracle та ін.). На сервері MongoDB може бути кілька баз даних, кожна з яких є контейнером для інших сутностей.
2. База даних містить кілька «колекцій». Колекція нагадує традиційну реляційну таблицю.
3. Колекції складаються із «документів». Знову ж таки, документ можна розглядати як «рядок».
4. Документ складається з одного або більше «полів», подібних до «колонок».
5. Механізми індексації в MongoDB концептуально подібні до індексів у реляційних СУБД.

6. Курсори також нагадують такі у реляційних БД. Тобто, при виконанні запиту MongoDB повертає курсор, із яким ми працюємо далі – підраховуємо, пропускаємо певну кількість попередніх записів, не завантажуючи при цьому самі дані.

Синтаксис та відмінності від реляційної СУБД (на прикладі PostgreSQL)

Зіставлення базових понять

У світі PostgreSQL таблиці відповідає колекція, а рядку – документ BSON, який має обмеження за розміром 16 МБ (таблиця 8). Стовпці реляційної таблиці аналогічні до полів документа, включаючи вкладені шляхи на кшталт address.city . Роль DDL та статичної типізації бере на себе валідатор колекції, що формулює правила на основі JSON Schema. Там, де в PostgreSQL застосовуються JOIN і зовнішні ключі, в MongoDB частіше використовуються денормалізація та стадія $lookup в агрегуючому пайплайні, але їхня вартість та обмеження суттєво відрізняються від реляційного світу.

Таблиця 8 – Зіставлення базових понять реляційної СУБД та MongoDB

РСУБД	MongoDB
База даних (Database)	База даних (Database)
Таблиця (Table)	Колекція (Collection)
Кортеж / Рядок (Tuple / Row)	Документ (Document)
Поле/Колонка (Field/Column)	Поле (Field)
З'єднання таблиць (Table Join)	Вбудовані документи (Embedded Documents)
Первинний ключ (Primary Key)	Первинний ключ Primary Key (за замовчуванням ключ _ id наданий самої MongoDB)
Сервер баз даних	
postgres/sqlservr/oracle	mongod

Еквівалентність CRUD-операцій

Між комендами та операторами PostgreSQL та MongoDB можна побачити певне відповідність (таблиця 9).

Таблиця 9 – Еквівалентність CRUD-операцій реляційної СУБД та MongoDB

Операція	PostgreSQL (SQL)	MongoDB	Коментар
Вставка	INSERT INTO ...	insertOne(), insertMany()	Документи можуть мати різну структуру (за правилами валідатора)
Вибірка	SELECT ... WHERE ...	find()	Використовуються фільтри та проекції полів, дозволяючи віддавати лише потрібні частини документа
Оновлення	UPDATE ... SET ... WHERE ...	updateOne(), updateMany()	Використовуються оператори: $set, $inc, $push, $addToSet, $unset
Видалення	DELETE FROM ... WHERE ...	deleteOne(), deleteMany()	Використовується та сама логіка фільтрації
Upsert	–	update(..., { upsert: true })	Поєднання вставки та оновлення

При фільтрації активно застосовуються оператори порівняння та логіки, а також роботи із вкладеними масивами (таблиця 10).

Таблиця 10 – Відповідність операторів фільтрації SQL та MongoDB

Тип	PostgreSQL	MongoDB	Призначення
Порівняння	=, <>, >, >=, <, <=	$eq, $gt, $gte, $lt, $lte	Порівняння значень
Множини	IN	$in, $nin	Перевірка належності до множини
Логічні	AND, OR, NOT	$and, $or, $not	Комбінування умов
Масиви	–	$elemMatch	Пошук у вкладених масивах

Індекси та унікальність

У MongoDB доступні одиночні та складові індекси, причому порядок полів у складовому індексі критичний через префіксне правило, що визначає, які запити зможуть використовувати індекс. Обмеження унікальності налаштовується так само, як у реляційних базах, та забезпечує недопущення дублікатів за заданим ключем. Часткові або фільтровані індекси дозволяють індексувати лише підмножину документів, що схоже на індекси з умовою WHERE у PostgreSQL і допомагає заощаджувати пам’ять. Sparse-індекси індексують лише документи, де поле присутнє, що важливо при розріджених даних. TTL-індекси автоматично видаляють застарілі документи та спрощують керування часом життя даних. Для завдань пошуку є текстові індекси, а для просторових запитів – геоіндекс типу 2dsphere. План виконання та кеш планів впливають на латентність, тому корисно періодично перевіряти explain() і при необхідності застосовувати hint() для вибору відповідного індексу.

Aggregation Framework проти SQL

Стадія $match грає роль фільтрації і відповідає умові WHERE, тому її намагаються розташовувати якомога раніше в пайплайні зменшення обсягу даних. Стадія $project відповідає за вибір та обчислення полів, дозволяючи формувати зручні подання та проводити перетворення безпосередньо в базі. Групування реалізується через $group, аналогом реляційного GROUP BY, за допомогою агрегатних функцій та складання масивів; частина віконних сценаріїв реалізується через комбінації стадій конвеєра або спеціалізовані оператори. Сортування та пагінація (pagination – розбиття на сторінки/фрагменти) виконуються стадіями $sort, $limit і $skip, проте для більших зсувів краще курсорна пагінація по ключу, оскільки офсети стають дорогими. Стадія $lookup забезпечує з’єднання колекцій і підтримує під-pipeline, що додає гнучкості, але вимагає уваги до обсягу даних, що передаються. Для роботи з масивами застосовується $unwind, що перетворює елементи масиву на окремі документи, а $facet і $bucket дозволяють розпаралелювати гілки обчислень і будувати гістограми в одному проході.

Транзакції та узгодженість

За промовчанням MongoDB гарантує атомарність на рівні одного документа, що покриває більшість сценаріїв при коректному моделюванні агрегатів. Багатодокументні транзакції підтримуються в replica set та шардованих кластерах, надаючи ACID-гарантії, але збільшуючи затримки і знижуючи пропускну здатність при конфліктах. Параметр write concern дозволяє вибрати силу підтвердження запису – від локального підтвердження до majority з журналуванням, – тоді як read concern управляє свіжістю даних, включаючи режим linearizable для найсуворіших вимог. Уподобання читання задають, звідки читати – з первинних або з вторинних вузлів, – що корисно для звітності, але при цьому необхідно враховувати можливе запізнення реплік.

Валідація схеми

Валідація в MongoDB формулюється на рівні колекції та використовує правила, сумісні з JSON Schema. Це дозволяє оголосити обов'язкові поля, типи значень, допустимі діапазони та перерахування. Добре зарекомендував себе підхід, при якому критичні для продуктивності та узгодженості поля визначаються суворо, а периферійні частини документа залишаються гнучкими та розширюваними, що дає баланс між безпекою та швидкістю розвитку.

Продуктивність та діагностика

Інструмент explain() розкриває етапи виконання запиту, показує план, що переміг, і дозволяє відрізнити індексні скани від повносканuючих проходів по колекції. Розміри документів впливають на поведінку зберігання: часті розширення великих документів призводять до перепакування та зайвих I/O, тому схеми слід проектувати із запасом. При виборі складових індексів варто керуватися емпіричним правилом порядку полів: спочатку рівності, потім діапазони і далі компоненти, що у сортуванні. Для пошуку регресій і точок зростання використовуються профільувальник та журнал повільних запитів, а також серверні метрики – завантаження CPU, інтенсивність I/O, обсяг робочої вибірки у пам'яті та частота page faults.

Обмеження та підводні каменці

Відсутність жорстких зовнішніх ключів переносить контроль посилальної цілісності на рівень логіки прикладної програми і в офлайн-процеси (пайплайни, конвеєри), тому механізми перевірки потрібно проектувати явно. Ліміт 16 МБ на документ і наявність великих вкладених масивів спонукають стежити за зростанням агрегатів і при необхідності виносити великі колекції в окремі сутності. Операції $lookup та інші міжколекційні обчислення дороги і особливо чутливі в шардованих інсталяціях через міжшардові пересилки. Пам'ять під індекси часто стає первинним обмежувачем, тому обсяг та склад індексів повинен відповідати робочому набору та частоті запитів.

Інструменти для роботи з MongoDB

Mongo Shell:

Як будь-який термінал (консоль) – найпростіший інструмент роботи з MongoDB. Він призначений для виконання адміністративних та прикладних команд. Перевага роботи через термінал полягає в тому, що ці ж команди можна легко інтегрувати у код в інших мовах програмування.

Консоль MongoDB (рис. 4) працює на JavaScript. Є декілька глобальних команд, наприклад, help або exit. Команди, які запускаються у поточній базі даних, виконуються з об'єктом db, наприклад *db.help()* або *db.stats()*. Команди, які запускаються стосовно конкретної колекції, виконуються з об'єктом *db.collection_name*, наприклад *db.collection_name.help()* або *db.collection_name.count()*

⚠ *Слід звернути увагу, що до інсталяцій останніх версій MongoDB термінал не включається і його треба скачати додатково.*

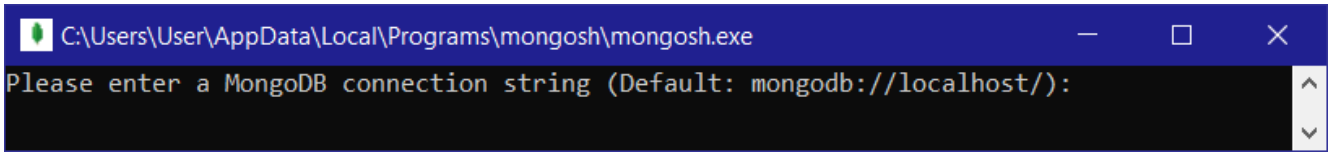


Рис. 4 – Початковий екран консолі Mongo Shell (mongosh)

MongoDB Compass:

Це спеціальна програма (середовище), яка надає візуальний інтерфейс для взаємодії з БД (рис. 6). Завдяки Compass можна створювати БД, додавати або видаляти значення, просто натискаючи кнопки та вводячи дані. Хоча це дуже зручно, проте будемо розглядати команди, що вводяться через термінал, для кращого зрозуміння їх логіки.

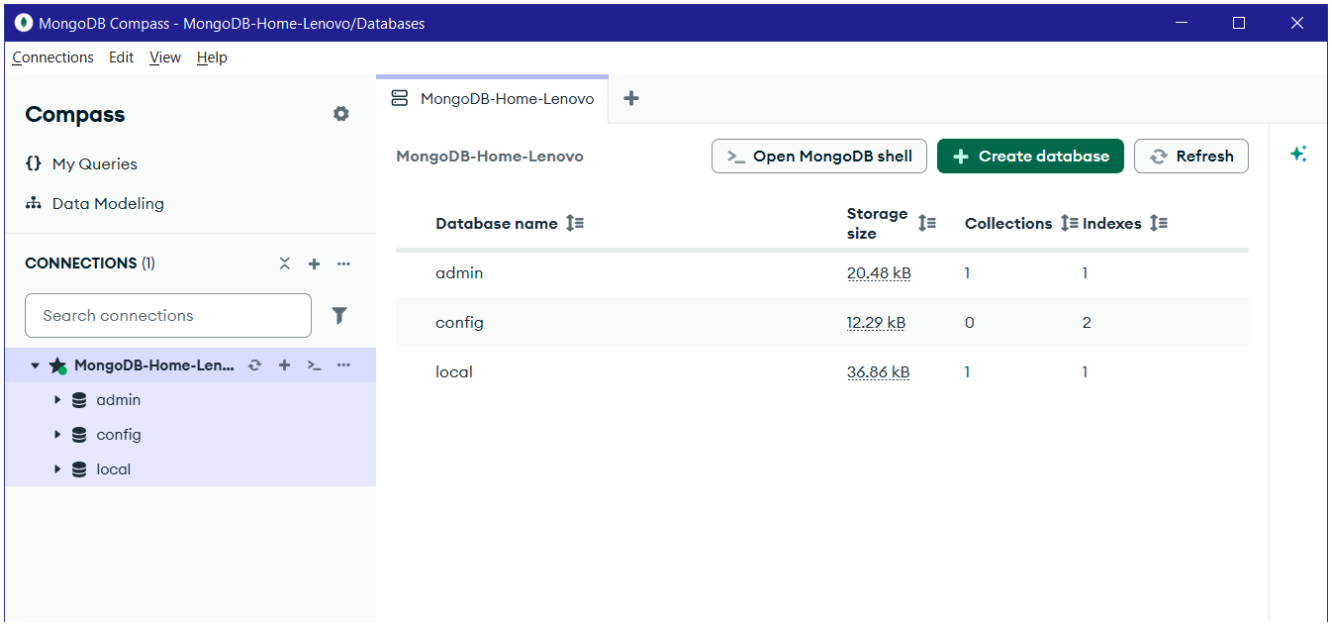


Рис. 5 – Початковий екран MongoDB Compass після під'єднання до сервера MongoDB

До речі, в середовищі MongoDB Compass також можна викликати вікно терміналу Mongo Shell і працювати безпосердньо в ньому (рис. 6).

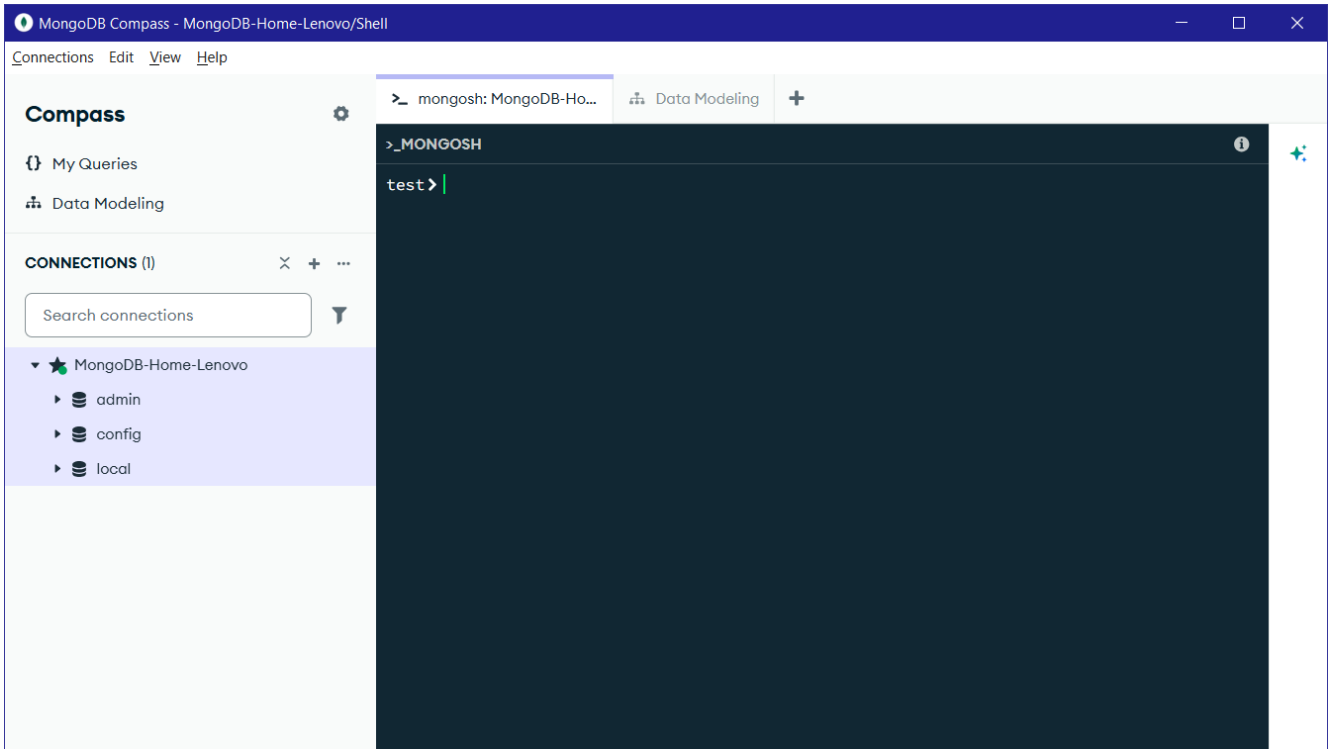


Рис. 6 – Вікно консолі Mongo Shell в середовищі MongoDB Compass

Підключення до сервера MongoDB

Підключення до хосту виконується з використанням стандартних значень. Зазвичай це порт **27017** (рис. 7). Після підключення видно три стандартні бази даних: admin, config та local. Вони створюються автоматично. Якщо в середровищі Compass їх видно одразу, то в терміналі треба ввести команду show dbs, яка поверне список БД, існуючих на сервері. Будь-яку зі стандарних БД можна видалити – вони найчастіше не потрібні для роботи, але й не заважають.

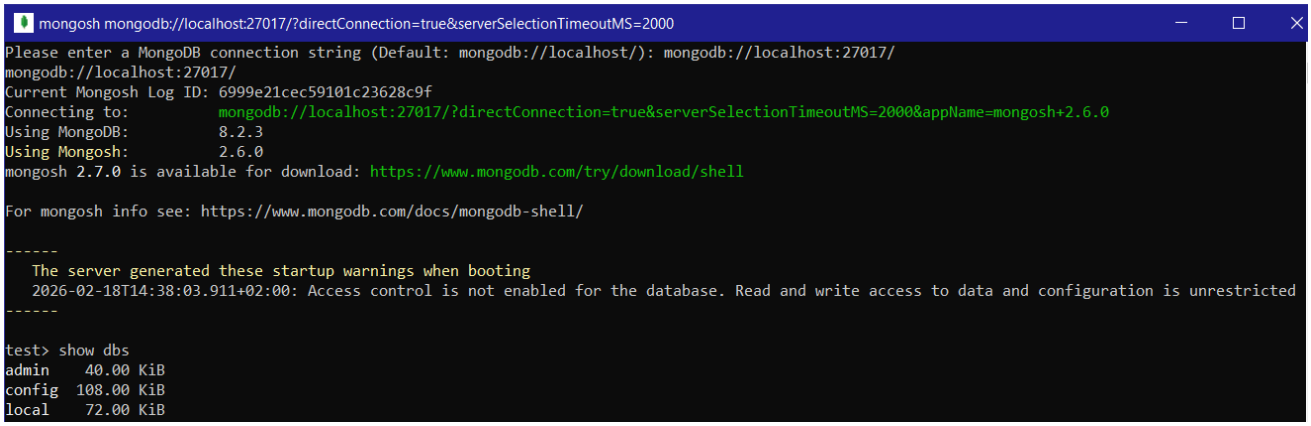


Рис. 7 – Під'єднання до сервера MongoDB з консолі Mongo Shell (mongosh)

Питання для самоперевірки

(до розділів «Основи документної моделі даних», «Синтаксис та відмінності від реляційної СУБД (на прикладі PostgreSQL)», «Інструменти для роботи з MongoDB»)

1. У яких випадках документна модель даних є природнішою за реляційну?
2. Що означає поняття «документ як агрегат» у MongoDB і чому документ розглядається як природна одиниця атомарності змін?
3. У чому полягають основні відмінності між документною та реляційною моделями даних?
4. Які переваги та недоліки денормалізації даних у MongoDB?
5. Що є концептуальними елементами документної моделі даних?
6. У чому полягає різниця між вкладеними документами та посиланнями між колекціями?
7. Що таке flexible schema і які переваги вона надає?
8. Які можливості надає Aggregation Framework і чим він концептуально відрізняється від SQL-запитів?
9. У чому полягають особливості підтримки транзакцій у MongoDB?
10. Для чого використовується валідація документів через JSON Schema?
11. Які засоби та інструменти застосовуються для роботи з MongoDB, і для яких задач доцільніше використовувати кожен із цих інструментів?
12. Які параметри та способи підключення можуть використовуватись для роботи з сервером MongoDB?

Розширення предметної області реляційної БД «Навчальний процес»

Для демонстрації доцільності використання документної моделі даних та створення наочних прикладів використання документної бази даних у СУБД MongoDB розширено предметну область «Навчальний процес», для якої в рамках курсу «Бази даних» було спроектовано та створено БД lectures на основі реляційної моделі у СУБД PostgreSQL. З ER-діаграмою цієї БД та SQL-скриптом її створення можна ознайомитись у конспекті лекцій з курсу «Організація баз даних».

Розширення предметної області полягає в тому, що викладачам додано їх наукові публікації та можливі активності типу участі у комітетах конференцій, редколегіях журналів тощо. При цьому в предметній області і в базі даних необхідно враховувати, що хоча множина типів публікацій є обмеженою та фіксованою, не всі типи публікацій, та й взагалі публікації, можуть бути у всіх викладачів. А перелік активностей, окрім участі у редколегіях журналів та комітетах конференцій, є повністю індивідуальним.

РБД тут може бути неефективною через те, що зберігання публікацій та індивідуальних активностей викладачів вимагало б створення додаткових таблиць і проміжних зв'язків (publication_types, lecturer_publications, activities, activity_types тощо), а також значної кількості операцій JOIN при формуванні повного профілю викладача. За умов, коли більшість запитів стосується перегляду профілю конкретного викладача разом із його публікаціями та активностями, реляційна модель призводить до фрагментації логічно пов'язаних даних у кількох таблицях. Це ускладнює схему, збільшує кількість з'єднань і знижує наочність моделі при відсутності суттєвої вигоди від нормалізації.

З огляду на те, що публікації тісно пов'язані з конкретним викладачем і нечасто запитуються окремо, а активність може мати складні взаємозв'язки та потребує гнучкості, для зберігання додаткової інформації про публікації та активність викладачів запропонована така структура:

1. «Публікації» створюються як вбудовані документи до колекції lecturers. При цьому використовується поліморфна структура як із загальними полями, так і специфічними для кожного типу, і додається поле type з категоризацією публікацій.
2. Для забезпечення гнучкості «Додаткова активність» створюється як окрема колекція activities. При цьому формуються посилання на викладачів через lecturerId і при вставці типу активності забезпечується автоматичне обчислення за правилами (логіка обробки та категоризації реалізується на рівні прикладної програми).

Інші об'єкти цієї предметної області також відкориговано відповідно до концептуальних особливостей документної моделі даних (Додаток А).

Завдання для самостійної роботи

Оберіть предметну область (ПрО) з наведеного переліку із задачами, для розв'язання яких доцільна документна модель даних, та:

– визначте початковий перелік колекцій для зберігання даних про сутності обраної ПрО;
– запропонуйте структуру основних документів;
– визначте, які дані доцільно зберігати у вкладених документах або масивах.

1. Електронна бібліотека навчальних матеріалів університету

1.1.Система призначена для:
- зберігання карток навчальних матеріалів;
- зберігання відомостей про авторів, кафедри та дисципліни;
- накопичення анотацій, ключових слів і метаданих;
- зберігання різних версій файлів навчальних матеріалів;
- формування персоналізованих добірок матеріалів для студентів.

1.2.Задачі:

1.2.1. Знайти навчальні матеріали певної дисципліни та року видання, відсортувати результати за датою оновлення та вивести лише назву, авторів і ключові слова.

1.2.2. Знайти навчальні матеріали за ключовим словом або фрагментом анотації, визначити релевантність результатів та передбачити використання текстового індексу, вивести показник релевантності (score) та відсортувати результати за спаданням релевантності.

1.2.3. Для кожного навчального матеріалу розгорнути масив версій файлів, визначити останню доступну версію та кількість оновлень документа.

1.2.4. Оновити інформацію про навчальний матеріал: додати нову версію файлу, змінити дату останнього оновлення та збільшити лічильник переглядів документа.

1.2.5. Сформувати список авторів, матеріали яких найчастіше використовуються студентами певної спеціальності, згрупувавши дані за авторами та дисциплінами.

2. Інтернет-магазин комп’ютерної техніки

2.1.Система призначена для:
- зберігання карток товарів різних категорій;
- накопичення технічних характеристик товарів;
- зберігання відгуків і оцінок користувачів;
- зберігання історії замовлень клієнтів;
- накопичення інформації про оплату та доставку.

2.2.Задачі:

2.2.1. Знайти ноутбуки з обсягом оперативної пам’яті не менше 16 ГБ, заданим типом процесора та ціною у вказаному діапазоні; результати відсортувати за рейтингом і ціною.

2.2.2. Виконати пошук товарів за текстом відгуків користувачів, ключовими словами опису та назвою бренду, передбачивши використання текстового індексу, вивести показник релевантності (score) та відсортувати результати за спаданням релевантності.

2.2.3. Для кожного товару розгорнути масив відгуків, визначити середню оцінку, кількість негативних відгуків та список найчастіше згадуваних недоліків.

2.2.4. Оновити картку товару: додати новий відгук користувача, змінити середню оцінку товару та збільшити кількість переглядів.

2.2.5. Визначити товари, які найчастіше купуються разом, проаналізувавши вкладені масиви позицій у документах замовлень клієнтів.

3. Медична інформаційна система приватної клініки

3.1.Система призначена для:

- зберігання електронних карток пацієнтів;
- накопичення історії звернень та лікування;
- зберігання результатів аналізів та обстежень;
- зберігання призначень лікарів;
- накопичення медичних документів і вкладених файлів.

3.2.Задачі:

3.2.1. Знайти пацієнтів із заданим діагнозом, певним лікарем та датою останнього звернення у вказаному часовому інтервалі.

3.2.2. Виконати пошук пацієнтів за фрагментом медичного висновку або результату аналізу, передбачивши використання текстового індексу, вивести показник релевантності (score) та відсортувати результати за спаданням релевантності.

3.2.3. Для кожного пацієнта розгорнути масив візитів, визначити кількість звернень, останній діагноз та перелік призначених процедур.

3.2.4. Оновити електронну картку пацієнта: додати новий візит, результати аналізів, призначення лікаря та змінити статус лікування.

3.2.5. Визначити лікарів, пацієнти яких найчастіше проходили повторне лікування, згрупувавши дані за лікарями та діагнозами.

4. Система управління подіями та реєстрацією учасників

4.1.Система призначена для:

- зберігання інформації про події різних типів;
- накопичення даних про учасників та спікерів;
- зберігання програм подій та розкладів;
- зберігання матеріалів доповідей і презентацій;
- накопичення інформації про реєстрацію та участь у заходах.

4.2.Задачі:

4.2.1. Знайти всі події певної тематики, дати або формату проведення; результати відсортувати за датою та кількістю зареєстрованих учасників.

4.2.2. Виконати пошук доповідей за ключовими словами назви, опису або матеріалів презентацій із використанням текстового індексу, вивести показник релевантності (score) та відсортувати результати за спаданням релевантності.

4.2.3. Для кожної події розгорнути масив секцій та доповідей, визначити кількість учасників, популярність секцій та список доповідачів.

4.2.4. Оновити документ події: додати нового учасника, змінити статус реєстрації та включити нову доповідь до програми заходу.

4.2.5. Визначити учасників, які найчастіше беруть участь у заходах певної тематики, згрупувавши дані за категоріями подій та ролями учасників.

5. Система моніторингу IoT-пристроїв у “розумній” будівлі

5.1.Система призначена для:

- зберігання інформації про IoT-пристрої;
- накопичення телеметричних даних;
- зберігання журналів подій та помилок;
- накопичення інформації про спрацювання датчиків;
- формування агрегованого стану приміщень і поверхів.

5.2.Задачі:

5.2.1. Знайти всі критичні спрацювання датчиків за заданий період часу, типом пристрою та приміщенням; результати відсортувати за рівнем критичності та часом події.

5.2.2. Виконати пошук повідомлень журналу подій за текстом помилки або типом аварійної ситуації, передбачивши використання текстового індексу, вивести показник релевантності (score) та відсортувати результати за спаданням релевантності.

5.2.3. Для кожного приміщення розгорнути масив телеметричних показників, визначити середні значення температури, вологості та енергоспоживання.

5.2.4. Оновити документ пристрою: додати нове телеметричне значення, журнал помилки та змінити поточний стан пристрою.

5.2.5. Визначити пристрої, які найчастіше переходять у стан помилки, згрупувавши дані за типами пристроїв та приміщеннями.

Якщо раніше виконувався проект в реляційній моделі, можна запропонувати розширення предметної області РБД за аналогією наведеного приклада ПрО «Навчальний процес».

Демонстраційний приклад

Завдання

Предметна область: Портфоліо студентських проєктів

ДД1. Система призначена для:

- зберігання інформації про студентів та їх навчальні проєкти;
- накопичення описів проєктів різних типів: вебзастосунків, мобільних застосунків, аналітичних систем, IoT-рішень, ML-проєктів тощо;
- зберігання інформації про авторів проєкту, використані технології та етапи виконання;

- накопичення посилань на репозиторії, презентації, демонстраційні матеріали та файли проєктів;
- зберігання відгуків керівників, оцінок та коментарів щодо виконаних проєктів.

ДД2. Задачі:

ДД2.1. Знайти проєкти, що використовують задану технологію або мову програмування, та відсортувати результати за датою оновлення.

ДД2.2. Виконати пошук проєктів за ключовими словами опису або назви, вивести показник релевантності (score) та відсортувати результати за спаданням релевантності.

ДД2.3. Для кожного проєкту розгорнути масив етапів виконання та визначити поточний стан реалізації проєкту.

ДД2.4. Оновити документ проєкту: додати новий етап виконання, файл або відгук керівника.

ДД2.5. Визначити технології, які найчастіше використовуються у студентських проєктах певної спеціальності або навчальної групи.

Розв'язання

Початковий перелік колекцій

1) ***students*** – колекція містить наступну інформацію про студентів:
   - ПІБ;
   - навчальна групу;
   - спеціальність;
   - контактні дані (номер телефону, адреса електроної пошти);
   - рік навчання (курс).
2) ***projects*** – основна колекція системи, що містить:
   - назву та опис проєкту;
   - тип проєкту;
   - авторів;
   - використані технології;
   - етапи виконання;
   - посилання на GitHub-репозиторії;
   - презентаційні матеріали;
   - вкладені файли;
   - відгуки керівників;
   - дату створення та оновлення.
3) ***project_tags*** – допоміжна колекція для зберігання тематичних тегів проєктів та їх категорій.

   Колекція може використовуватись для уніфікації тегів, формування тематичних добірок, реалізації навігації та фільтрації проєктів, пошуку проєктів за тематикою. Можливі варіанти тегів: web, mobile, machine-learning, database, cybersecurity, IoT, data-analysis тощо.

Мова даних MongoDB (CRUD-операції)

Розглянемо роботу з БД MongoDB, використовуючи аналогію з мовою запитів (типу SQL), де команди можна згрупувати в «мова визначення даних» (DDL, для створення, зміни, видалення всіх елементів БД, тобто структур), мову маніпулювання даними (DML, для роботи з вмістом БД, тобто і видалення, тобто управління даними (DCL, управління доступом, користувачами, транзакціями, безпекою тощо).

Застосовуючи цю аналогію до MongoDB, необхідно враховувати, що MongoDB – це документо-орієнтована NoSQL СУБД: її «мову» реалізовано не через текстові команди, як у SQL, а через методи та операції у JSON-подібному форматі, тобто на JavaScript-подібному синтаксисі (наприклад, з використанням оболонки mongosh) або в середовище MongoDB Compass.

Поняття «мова даних» у MongoDB

У класичних реляційних СУБД термін «мова даних» зазвичай безпосередньо асоціюється з SQL – стандартизованою текстовою мовою, що включає оператори CREATE, SELECT, UPDATE, GRANT та ін.

MongoDB не використовує єдину текстову мову запитів. Тим не менш, в архітектурному та навчальному контексті говорять про «мову даних MongoDB», маючи на увазі сукупність засобів опису структури даних та операцій над ними (таблиця 11).

Під цією мовою розуміються:

- формат зберігання та подання даних (BSON/JSON-подібна структура);
- декларативний синтаксис запитів та оновлень;
- система операторів та команд управління.

Таблиця 11 – «Мова даних» MongoDB у термінах DDL, DML, DCL (SQL vs MongoDB)

Категорія	SQL	MongoDB	Особливості
Мова визначення даних (DDL) – управління **структурами**: БД, колекції, індекси, схеми	CREATE, ALTER, DROP, TRUNCATE	**Неявний та частково явний DDL** Колекції та бази створюються автоматично при першому використанні, але можна управляти ними явно	
Мова маніпулювання даними (DML) – робота з **даними**: вставка, вибірка, оновлення, видалення	SELECT, INSERT, UPDATE, DELETE	**Повноцінний DML** реалізований через методи: insertOne, find, updateOne, deleteMany та ін.	Заснований на **JSON / BSON,** включає потужні оператори ($set, $push, $gt, $lookup)
Мова управління даними (DCL) – управління **доступом, безпекою, транзакціями**	GRANT, REVOKE, COMMIT, ROLLBACK	**Підтримка DCL** Користувачі, ролі, привілеї, транзакції (replica set / sharded cluster)	Управління через команди createUser, grantRolesToUser, db.runCommand (...)

JSON та BSON як основа представлення даних

Логічно документи MongoDB виглядають як JSON об’єкти. Це робить модель даних інтуїтивною для розробників, що працюють із JavaScript, Python, REST-API та веб-застосунками.

Фізично дані зберігаються у форматі BSON (Binary JSON), який є бінарним розширенням JSON і додає:

- сувору типізацію (ObjectId, Date, Decimal128 та ін .);
- компактне бінарне зберігання;
- підтримку ефективної індексації та серіалізації.

Важливо наголосити, що JSON та BSON не є мовами програмування. Це формати представлення даних, які використовуються декларативною мовою запитів MongoDB (таблиця 12).

Таблиця 12 – JSON та BSON: логічне та фізичне подання даних

Аспект	**Механізм, що використовується в MongoDB**	**Концептуальний аналог**
Формат логічного подання	JSON-подібні документи	JSON
Формат фізичного зберігання	BSON	Бінарний JSON
Типи даних	ObjectId, Date, Decimal128 та ін.	Розширення стандартних JSON-типів
Синтаксис запитів	JSON-подібні документи та оператори	JavaScript-об’єкти / DSL
Структура даних	Вкладені документи та масиви	Об’єкти в JS, словники для Python
Модель агрегації	Конвеєр операцій (pipeline)	Функціональне оброблення даних

Приклад логічного JSON-подібного документа MongoDB (рівень моделі даних):

```
{
 secondName: "Арсірій",
 firstName: "Олена",
 patronymic: "Олександрівна",
 spec: { code: "IT", name: "Інформаційні системи і технології" },
 email: "arsiriy@gmail.com",
 ratings: [
 { discipline: "Технології проектування БД", mark: 90 },
 { discipline: "Мережеві технології",  mark: 76 }
 ]
}
```

Видно, що структура документа безпосередньо відбиває предметну область і вимагає жорсткого попереднього опису схеми.

Приклад BSON-документа з розширеними типами даних (фізичний рівень зберігання):

```
{
  _id: ObjectId("65f000000000000000000002"),
  createdAt: ISODate("2011-10-10T00:00:00Z"),
  dkod: NumberInt(1),
  hours: NumberInt(72),
  coursework: true,
  salary: NumberDecimal("1500.00")
}
```

BSON розширює JSON, додаючи сувору типізацію та оптимізацію для зберігання та індексації.

Аналогія з SQL: DDL, DML, DCL у MongoDB

Незважаючи на відсутність SQL, концептуальний поділ операцій із призначення MongoDB зберігається. За аналогією з реляційними СУБД можна назвати групи операцій: DDL, DML і DCL (таблиця 13).

Таблиця 13 – DDL / DML / DCL у MongoDB

Категорія	Підтримка у MongoDB	Форма подання
DDL	Часткова (неявна + явна)	Методи: createIndex, drop, createCollection
DML	Повна та потужна	Методи: find, insert, update, delete, aggregate
DCL	Повна (користувачі, ролі, транзакції)	Команди: createUser, grantRoles, session API

Мова визначення даних (DDL) у MongoDB

MongoDB відноситься до схемно-гнучких СУБД. Це означає, що структура даних не задається жорстко заздалегідь.

Колекції та бази даних створюються автоматично при першому зверненні до них. Однак MongoDB надає засоби явного управління структурою:

- створення та видалення колекцій;
- управління індексами;
- завдання правил валідації документів через JSON Schema.

Таким чином, DDL в MongoDB існує (таблиця 14), але носить менш обов’язковий і більш гнучкий характер, ніж SQL.

Таблиця 14 – DDL-операції в SQL та MongoDB

SQL	MongoDB
CREATE DATABASE lectures;	use lectures (автоматично створює при першому insert)
CREATE TABLE students (...);	db.students.insertOne ({...}) (автоматично створює колекцію)
DROP TABLE students;	db.students.drop()
CREATE INDEX idx_name ON students (SecondName);	db.students.createIndex({ SecondName: 1 })

SQL	MongoDB
ALTER TABLE ... ADD COLUMN	не потрібно – документи гнучкі, поля додаються динамічно
Управління схемою	через **валідатори** та **сніпети схеми**: db.createCollection ("students", { validator : { $ jsonSchema : { ... } } })

Мова маніпулювання даними (DML) у MongoDB

DML є найрозвиненішою частиною MongoDB (таблиця 15). Всі операції маніпулювання даними описуються за допомогою JSON-подібних документів.

MongoDB підтримує:

- вибірку документів (find);
- вставку (insertOne, insertMany);
- оновлення (updateOne, updateMany з операторами $set, $push та ін);
- видалення (deleteOne, deleteMany).

Особливе місце займає агрегатний конвеєр (aggregate), який за виразністю можна порівняти з SELECT + JOIN + GROUP BY в SQL.

Таблиця 15 – DML-операции в SQL и MongoDB

SQL	MongoDB
SELECT * FROM rating WHERE mark > 60;	db.students.find({ ratings.mark: { $gt: 60 } })
INSERT INTO students VALUES (...);	db.students.insertOne({ name: "Євгеній", gNum: 842 })
UPDATE students SET gNum=211 WHERE FirstName="Євгеній";	db.students.updateOne({ FirstName: "Євгеній" }, { $set: { gNum=211 } })
DELETE FROM students WHERE name="Олена";	db.students.deleteOne({ name: "Олена" })
SELECT name, age FROM students;	db.students.find({}, { name: 1, age: 1 }) – проекція
Складні вибірки (JOIN, агрегації)	db.students.aggregate ([...]) – потужна конвеєрна обробка

Приклад операції вибірки (аналог SELECT з умовою):

```
// Знайти студентів, у яких є хоча б одна оцінка > 80
db.students.find({ "ratings.mark": { $gt: 80 } })
```

У цьому способі умови відбору задаються через оператори ($gt, $lt, $in), а чи не через WHERE.

Приклад оновлення документа:

```
// Оновити email студента з kod=1
db.students.updateOne(
  { kod: 1 },
  { $set: { email: "bvgo12345@ukr.net" } }
)
```

Як і SQL, оновлення описується декларативно і зачіпає інші поля документа.

Приклад агрегатного конвеєра (аналог GROUP BY):

```
// Середній бал по кожній дисципліні (по всім студентам)
db.students.aggregate([
 { $unwind: "$ratings" },
 {
 $group: {
 _id: "$ratings.discipline",
 avgMark: { $avg: "$ratings.mark" },
 cnt: { $sum: 1 }
 }
 },
 { $sort: { avgMark: -1 } }
])
```

Тут агрегація будується як послідовність етапів обробки даних.

Мова управління даними (DCL) у MongoDB

MongoDB підтримує керування доступом та безпекою через механізм користувачів та ролей (таблиця 16).

Адміністратор може:

- створювати користувачів;
- призначати ролі та привілеї;
- керувати правами доступу до баз та колекцій.

Крім того, MongoDB підтримує багатодокументні транзакції (при роботі в replica set і sharded cluster), що наближає до можливостей класичних СУБД.

Таблиця 16 – DCL-операції в SQL та MongoDB

SQL	MongoDB
GRANT SELECT ON students TO DeptHead;	db.grantRolesToUser ("DeptHead", ["read"])
CREATE USER DeptHead WITH PASSWORD '...';	db.createUser ({ user: "DeptHead", pwd: "...", roles: [...] })
REVOKE ...	db.revokeRolesFromUser (...)
COMMIT; ROLLBACK;	Підтримка **багатодокументних транзакцій** (replica set): session.startTransaction (), session.commitTransaction()
Управління ролями	Можна створювати кастомні ролі: db .createRole (...)

Порівняння РСУБД (SQL) та MongoDB на рівні мови даних

SQL є єдиною стандартизованою текстовою мовою.

MongoDB використовує предметно-орієнтовану декларативну мову (DSL), реалізовану через методи та JSON-подібні структури.

Незважаючи на відмінності синтаксису, логічна модель роботи з даними багато в чому збігається (таблиця 17).

Таблиця 17 – SQL проти MongoDB

Аспект	SQL	MongoDB
Синтаксис	Текстова мова (SELECT, WHERE)	JSON-подібні документи та методи JavaScript
Схема	Жорстка (DDL обов'язковий)	Гнучка (DDL не є обов'язковим, але можливим)
Зберігання	Таблиці, рядки, стовпці	Документи, колекції, поля
Мова	Єдиний стандарт (SQL)	Команди в mongosh (на основі JavaScript) та BSON

Висновки про мову даних MongoDB

Отже, СУБД MongoDB немає аналога SQL як єдиного текстового мови. Однак вона надає повноцінний функціональний еквівалент мови даних через JSON/BSON, декларативні операції та чіткий поділ відповідальності між DDL, DML та DCL.

Такий підхід забезпечує гнучкість, масштабованість та зручність інтеграції в сучасні розподілені та веб-орієнтовані системи.

Тепер розглянемо детальніше команди та методи мови даних MongoDB.

CD-операції (створення та видалення БД та її елементів)

Рівень БД

Для ***створення бази даних*** MongoDB використовується команда

use *db_name*

яка створить нову БД, якщо вона не існує або повертає існуючу БД (рис. 8).

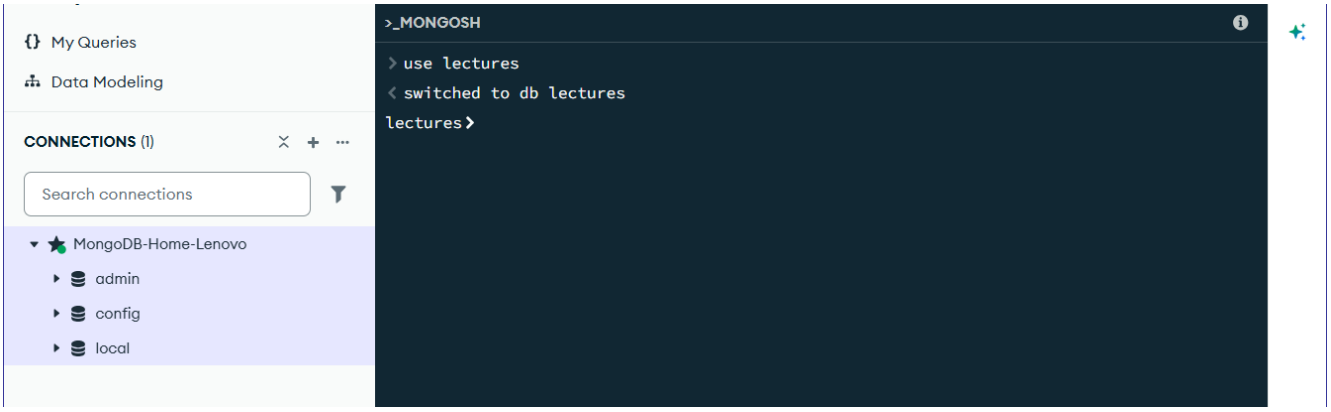


Рис. 8 – Створення БД lectures («Навчальний процес»)

Перевірити вибрані на даний момент бази даних можна за допомогою команди

db

А команда

show dbs

повертає список наявних баз даних. Однак «свіжостворені» БД у ньому не буде. Для цього необхідно вставити до неї принаймні один документ.

Видалення бази даних.

⚠ *Важливо знати, що база даних у MongoDB **автоматично видаляється,** якщо в ній **немає жодної колекції** . Тобто, якщо ви видалите всі колекції з бази даних, сама база даних також зникне.*

Рівень колекцій

Наступний крок після створення бази даних – створити в ній **колекції**.
Створити колекцію можна двома шляхами.

1. Явне створення через термінал.

   Для цього використовується команда

   db.createCollection("*ім'я_колекції*").

   Наприклад, для створення колекції *students*:

   db.createCollection("students").

Після виконання команди ви отримаєте підтвердження ok: 1, що означає успішне створення, і одразу можна побачити базу даних lectures з колекцією students (рис. 9).

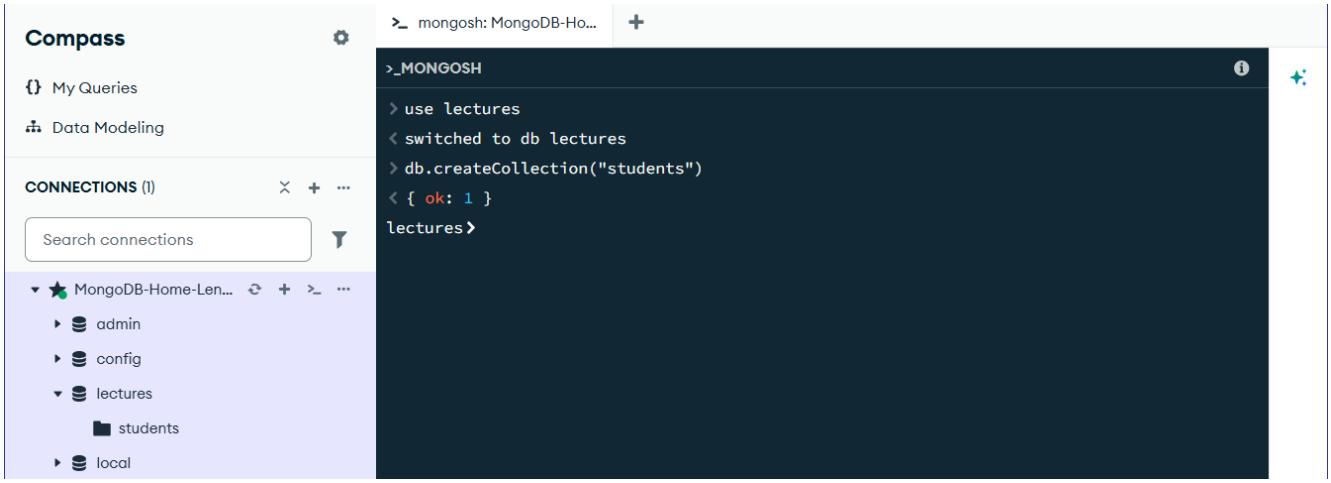


Рис. 9 – Створення колекції students в БД lectures

2. Автоматичне створення.

⚠ *Дуже зручною особливістю MongoDB є те, що колекція **автоматично створюється** при вставці до неї даних, коли **колекції ще не існує**.*

На відміну від реляційних баз даних, де таблиці мають фіксований набір полів, MongoDB надає ***еластичну структуру колекцій***. Це означає, що різні об'єкти (документи) в одній і тій же колекції ***можуть мати різну кількість та різні типи полів***, включаючи рядки, числа, булеві значення, масиви та вкладені об'єкти. Наприклад, один користувач може мати ім'я та email, а інший – ще й номер телефону або адресу, без необхідності змінювати структуру всієї колекції. Це забезпечує велику гнучкість у зберіганні даних.

Тобто, у створену вище колекцію можна вставити будь-який документ будь-якої структури. Наприклад:

```
db.students.insertOne({ name: "Євгеній", anything: 123 });
```

Проте можна визначити і певну жорстку або напівжорстку структуру документів в колекції. з різним рівнем контролю структури документів, що додаються. Наприклад:

– **напівжорстка без валідатора** – в такій колекції обмеження документів підтримуються виключно на рівні застосунку та індексами:

```
db.createCollection("departments");

// логична "жорсткість" через індекси
db.departments.createIndex({ deptKod: 1 }, { unique: true });
db.departments.createIndex({ deptName: 1 }, { unique: true });
```

проте можна вказати, що є рекомендуєма структура документу:

```
{
 deptKod: 1,
 deptName: "ІМФІТ",
 pKod: 2 // необов’язкове поле
}
```

– **напівжорстка з валідатором**, який контролює *часткову* структуру документів при їх додаванні – тобто є обов’язкове ядро, проте допускаються додаткові поля:

```
db.createCollection("departments", {
 validator: {
  $jsonSchema: {
   bsonType: "object",
   required: ["deptKod", "deptName"],   // обов’кові поля
   properties: {
    deptKod: { bsonType: "int", minimum: 1 },
    deptName: { bsonType: "string", minLength: 1, maxLength: 100 },
    pKod: { bsonType: "int" }                // необов’кове поле
   }
  }
 }
});
```

– **жорстка з валідатором**, тобто задана фіксована структура документу, а будь-які додаткові поля заборонені:

```
db.createCollection("publication_types", {
 validator: {
  $jsonSchema: {
   bsonType: "object",
   required: ["code", "label"],
   additionalProperties: false,   // заборона додаткових полів, окрім ти
x,
   properties: {                    // що вказані тут в properties
    _id: { bsonType: "objectId" },
    code: { bsonType: "string", minLength: 1, maxLength: 30 },
    label: { bsonType: "string", minLength: 1, maxLength: 200 }
   }
  }
 }
});
```

Щоб ***видалити колекцію*** необхідна команда

db.*ім'я_колекції*.drop().

Наприклад, для видалення колекції *students*:

db.students.drop().

Повідомлення *true* підтверджує успішне видалення.

⚠ *Нагадування: після видалення останньої колекції в базі даних, сама база даних також буде видалена.*

Робота з даними

Основи додавання даних

1. Будь-який запис у колекції MongoDB є, по суті, ***JavaScript об'єктом***. Це означає, що робота йде зі звичними структурами даних, які містять пари «ім'я параметра» та «значення». Наприклад,

   name: "Євгеній"

2. Кожен запис повинен мати **унікальний ідентифікатор (ID)**. Цей _id додається автоматично при створенні нового запису, хоча його можна вказати й вручну. Якщо ID не вказано, він генерується випадковим чином і завжди є унікальним.

Для ***додавання одного нового запису*** в колекцію використовується метод insertOne:

db.collectionName.insertOne({})

Всередині фігурних дужок прописуються поля та їхні значення для цього об'єкта.

Наприклад:

```
use lectures

db.students.insertOne({
  kod: 9,
  secondName: "Блажко",
  firstName: "Олександр",
  patronymic: "Анатолійович",
  snum: 2,
  gnum: 954,
  email: "blazko@ukr.net",
  spec: db.specialties.findOne({ code: "КІ" }, { _id: 0 }),
  address: { city: "м. Одеса", street: "ул. Місцева", house: 2, flat: 4 },
  letters: [],
  ratings: [],
```

```
 diploma: null
})
```

Типи даних MongoDB задаються автоматично при введенні через формат BSON (Binary JSON) у вигляді пар «ключ-значення» (таблиця 18). MongoDB сама визначає тип (рядок, число, масив, дата тощо) на основі синтаксису: лапки для рядків, відсутність лапок для чисел/булевих значень, квадратні дужки для масивів.

Таблиця 18 – Приклади типів даних MongoDB

Тип	Приклад	Коментар
Рядки	name: "Євгеній"	UTF-8
Числа	age: 25	32/64-бітові цілі або Double
Логічні	isMember: true	
Масиви	tags: ["mongodb", "nosql"]	
Вкладені документи (об'єкти)	address: {city: "Odessa", street: "Malinovskogo", house: NumberInt(10), flat: NumberInt(42)}	
Дата	created_at: New Date()	
Ідентифікатор	_id: ObjectId()	

Після виконання команди db.<collection>.insertOne({...}), MongoDB повідомить, що запис був доданий, і надасть згенерований _id.

Після додавання запису його можна переглянути в Compass. Там можна перемикатися між ***списковим (list)*** та ***табличним (table)*** виглядом. Хоча табличний вигляд може здатися звичнішим, списковий краще відображає різноманітність полів та їх типів, особливо коли є вкладені об’єкти або масиви, а також відсутність певних полів у деяких документах.

⚠ *Коментар: Якщо в Compass не видно доданих даних, це означає, що не вказано, з якою базою даних йде робота в терміналі. Щоб переключитися на потрібну базу даних треба ввести команду use назва_БД, наприклад, use lectures.*

Окрім додавання одного запису можна ***додати одразу декілька записів*** за допомогою команди insertMany. Синтаксис insertMany вимагає передачі *масиву об’єктів*. Навіть якщо треба додати лише один запис за допомогою insertMany, він все одно повинен бути всередині квадратних дужок, тобто у вигляді масиву. Наприклад:

```
db.collectionName.insertMany([{}, {}])
```

Наприклад:

```
db.disciplines.insertMany([
 { dkod: 6, name: "NoSQL Бази даних", hours: 72, semesters: 1,
coursework: true,
```

```
currentSemester: 7 },
 { dkod: 7, name: "Розподілені системи", hours: 36, semesters: 1,
coursework: false,
currentSemester: 7 }
])
```

При виконанні insertMany MongoDB повідомляє про успішне додавання та також надає ID для всіх доданих записів.

Для ***видалення даних*** існують дві функції:

– deleteOne, яка видаляє лише **один запис**, який відповідає фільтру (*перший знайдений*).
– deleteMany, яка видаляє **всі записи**, які відповідають заданому фільтру.

```
db.collectionName.delete[One/Many](filter)
```

Наприклад, щоб видалити всі об’єкти, для яких виконується умова:

```
// Видалити всіх студентів, у яких є оцінка 0 хоча б по одній дисципліні
db.students.deleteMany({ "ratings.mark": 0 })
// Видалити всіх студентів 1994-1996 вступу (включно)
db.students.deleteMany({ "students.gnum": { $gte: 940, $lt: 970 } })
```

Питання для самоперевірки

(до розділів «Мова даних MongoDB (CRUD-операції)», «CD-операції (створення та видалення БД та її елементів)»)

1. У чому полягають особливості «мови даних» MongoDB порівняно з класичними реляційними СУБД?
2. Яку роль відіграють JSON та BSON у MongoDB?
3. Чим BSON відрізняється від JSON і які додаткові типи даних він підтримує?
4. Які особливості реалізації DDL-операцій у MongoDB?
5. У чому полягають особливості реалізації DML-операцій у MongoDB?
6. Чому в MongoDB створення баз даних та колекцій може виконуватись неявно?
7. Які операції відносять до CD-операцій у MongoDB?
8. Які команди використовуються для створення та видалення баз даних у MongoDB?
9. У чому полягають особливості створення та видалення колекцій у MongoDB?
10. Для чого використовується метод createCollection()?
11. Які засоби MongoDB застосовуються для управління індексами?
12. Для чого використовується валідація документів при створенні колекцій?
13. У чому полягає різниця між гнучкою та жорсткою схемою колекції?

14. Чому в MongoDB можливе співіснування документів із різною структурою в межах однієї колекції?
15. Які переваги та ризики має автоматичне створення колекцій?
16. У чому полягають особливості роботи з даними при створенні перших документів у колекції?
17. Які операції MongoDB можна умовно віднести до DCL?
18. Чому поділ на DDL, DML та DCL у MongoDB є більш умовним, ніж у SQL-СУБД?

Завдання для самостійної роботи

Написати запити і, використовуючи інструментарій описаний в розділі «Інструменти для роботи з MongoDB», виконати їх для:

1) сворення бази даних ПрО, обраної в розділі «Завдання для самостійної роботи» (с. 31);
2) створення колекцій, визначених для обраної ПрО;
3) наповнення колекцій прикладами документів, притаманих обраній ПрО.

Демонстраційний приклад

Завдання

Написати запити для:

1) сворення бази даних ПрО, описаної в розділі «Демонстраційний приклад» (с.34);
2) створення колекцій, визначених для обраної ПрО;
3) наповнення колекцій прикладами документів, притаманих обраній ПрО.

Розв’язання

Створення та вибір БД

```
// База даних буде створена автоматично після додавання перших документів
use student_projects_db
```

Створення колекцій

```
// Колекції для основних сутностей предметної області
db.createCollection("students")

db.createCollection("projects")

// Окрема колекція для довідника тегів проєктів
db.createCollection("project_tags")
```

Наповнення колекції students

```
db.students.insertMany([
// Перший документ студента
{
```

```
  student_id: 1,                  // Унікальний ідентифікатор студента
  second_name: "Мікулінська",
  first_name: "Марія",
  group: "ІС-31",                 // Академічна група
  speciality: "F6 Інформаційні системи та технології",

  // Вкладений документ контактних даних
  contacts: {
    email: "m.mikulinskaya@university.edu.ua",
    phone: "+380501112233"
  },

  study_year: 3                   // Курс навчання
},
// Другий документ студента
{
  student_id: 2,
  second_name: "Лисенко",
  first_name: "Максим",
  group: "КН-41",
  speciality: "F3 Комп’ютерні науки",

  contacts: {
    email: "m.lysenko@university.edu.ua",
    phone: "+380974445566"
  },

  study_year: 4
}
])
```

Наповнення колекції project_tags

```
db.project_tags.insertMany([
{
  tag: "web"                          // Назва тегу,
  category: "software-development" // Категорія тегу
},
{
  tag: "machine-learning",
  category: "artificial-intelligence"
}
])
```

Наповнення колекції projects

```
db.projects.insertMany([
// Перший документ проєкту
{
   project_id: 101,          // Ідентифікатор проєкту
   title: "Система управління студентськими проєктами",     // Назва проєкту
   project_type: "web-application",

   // Масив авторів проєкту
   authors: [

      // Вкладений документ з інформацією про учасника
      {
         student_id: 1,
         role: "team-lead"
      },
      {
         student_id: 2,
         role: "backend-developer"
      }
   ],

   // Масив використаних технологій
   technologies: [
      "MongoDB",
      "Node.js",
      "React"
   ],

   // Масив тегів
   tags: [
      "web"
   ],

   // Масив етапів виконання проєкту
   stages: [

      // Вкладений документ етапу
      {
         stage_name: "Аналіз вимог",
         status: "completed",          // Поточний стан етапу
         date: ISODate("2026-02-10")   // Дата завершення або оновлення
```

```
      },

      {
        stage_name: "Розробка API",
        status: "in-progress",
        date: ISODate("2026-03-01")
      }
    ],

    // Посилання на репозиторій
    repository: "https://github.com/example/student-projects",

    // Масив відгуків керівників
    supervisor_reviews: [
      {
        supervisor: "доц. Шпінарева І.М.",
        rating: 95,                          // Оцінка проєкту
        comment: "Добре продумана архітектура системи."
      }
    ],

    created_at: ISODate("2026-02-01"),       // Дата створення документа
    updated_at: ISODate("2026-03-05")        // Дата останнього оновлення
},

// Другий документ проєкту
{
    project_id: 102,
    title: "Прогнозування успішності студентів",
    project_type: "machine-learning",

    authors: [
      {
        student_id: 1,
        role: "data-analyst"
      }
    ],

    technologies: [
      "Python",
      "MongoDB",
      "Scikit-learn"
```

```
  ],

  tags: [
    "machine-learning"
  ],

  stages: [
    {
      stage_name: "Підготовка даних",
      status: "completed",
      date: ISODate("2026-01-20")
    }
  ],

  repository: "https://github.com/example/student-ml-project",

  supervisor_reviews: [
    {
      supervisor: "проф. Волков В.Е.",
      rating: 98,
      comment: "Якісна аналітична частина."
    }
  ],

  created_at: ISODate("2026-01-15"),
  updated_at: ISODate("2026-02-12")
}
])
```

RU-операції (маніпулювання даними)

Вибірка даних з колекції

Для початку роботи з даними потрібно підключитися до бази даних, наприклад lectures. Після цього можна почати вибірку даних з колекції students.

Щоб отримати ***всі записи*** з колекції, використовується метод find(). Наприклад:

```
db.students.find()
```

з наступним результатом (рис. 10):

Рис. 10 – Записи колекції students в БД lectures

Якщо треба вивести лише певну кількість записів (наприклад, перші дві), можено додати метод limit(). Наприклад:

```
db.students.find().limit(2)
```

Іноді не треба відображати певні поля, такі як автоматично створений _id, оскільки вони займають багато місця і можуть зробити вивід менш лаконічним. Для цього можно вказати другі фігурні дужки у find() і задати _id: 0. Наприклад:

```
db.students.find({}, {_id: 0}).limit(2)
```

Функція find() дозволяє не тільки вибирати дані, але й сортувати їх.
Сортування можна виконувати **за одним полем**.

- 1 – для сортування за зростанням. Наприклад:

```
// Сортування студентів за прізвищем (зростання)
db.students.find({}, { _id: 0, kod: 1, secondName: 1, firstName:
1 }).sort({ secondName: 1 })
```

- -1 для сортування за спаданням. Наприклад:

```
// Сортування студентів за прізвищем (спадання)
db.students.find({}, { _id: 0, kod: 1, secondName: 1, firstName:
1 }).sort({ secondName: -1 })
```

Можна сортувати **за декількома полями одночасно**. MongoDB спочатку сортує за першим вказаним полем, а потім, якщо значення в першому полі однакові, сортує за наступним полем. Наприклад:

```
// Сортування викладачів: спочатку за кафедрою, потім за прізвищем
db.lecturers.find({}, { _id: 0, kod: 1, secondName: 1, "dept.deptName": 1 })
  .sort({ "dept.deptName": -1, secondName: 1 })
```

Фільтри дозволяють відображати записи, які відповідають певним критеріям. Фільтрація може виконуватись:

- **за дорівненістю:** значення вказується у перших фігурних дужках find(). Наприклад:

```
// Знайти дисципліну з dkod=1
db.disciplines.find({ dkod: 1 })
```

- **за кількома критеріями (логіка AND):** комбінуючи фільтри. Наприклад:

```
// Знайти студента по прізвищу та спеціальності
db.students.find({ secondName: "Арсірій", "spec.code": "IT" })
```

- **за кількома критеріями (логіка OR):** якщо треба знайти записи, які відповідають *хоча б одному* з декількох критеріїв, використовується оператор $or, який приймає масив об’єктів, де кожен об’єкт – це окремий критерій. Наприклад:

```
// Знайти студента по прізвищу або спеціальності
db.students.find({ $or: [{ secondName: "Арсірій" }, { "spec.code": "IT" }] })
```

Крім того, MongoDB надає оператори порівняння для перевірки значень на "більше", "менше", "рівно" тощо (таблиця 19).

Наприклад, знайти студентів, які мають хоча б одну оцінку нижче 60 з дисципліни «Технології проектування БД», і відсортувати їх за прізвищем:

```
db.students.find({ "ratings.mark": {$lt: 60}},
ratings.discipline: "Технології проектування
БД" }).sort({secondName: 1})
```

Таблиця 19 – Оператори порівняння MongoDB

Оператор	Зміст
$lt	less than
$gt	greater than
$gte	greater than or equal
$lte	less than or equal
$eq	equal
$ne	not equal

Оператори $in та $nin дозволяють вибирати або виключати елементи, які **відповідають одному з декількох значень у масиві**.

Наприклад, вивести студентів, у яких spec.code належить/не належить множині {«ІТ», «КІ», «КМ»}:

```
db.students.find({spec.code: {$in: [«ІТ», «КІ», «КМ»]}})
db.students.find({spec.code: {$nin: [«ІТ», «КІ», «КМ»]}})
```

Об’єкти також можна вибирати залежно від того, чи існує у них певне поле.

```
$exists: true / false
```

Наприклад, вивести об’єкти (викладачів), у яких є/немає поля publications:

```
db.lecturers.find({publications: {$exists: true}})
db.lecturers.find({publications: {$exists: false}})
```

Як и деякі діалекти SQL, MongoDB дозволяє працювати з полями, які є масивами, і фільтрувати за ними.

– **за розміром масиву ($size).** Наприклад, вивести всі об’єкти, у яких поле r_colors є масивом і має розмір, наприклад, два елементи:

```
db.rainbow.find({"r_colors": {$size: 2}})
```

– **за значенням елемента масиву за індексом** для перевірки значення конкретного елемента в масиві за його індексом. Наприклад, знайти об’єкти, де перший елемент (індекс 0) поля r_colors дорівнює "червоний":

```
db.rainbow.find({"r_colors.0": "червоний"})
```

– **фільтрація елементів масиву за операторами порівняння** для перевірки елементів масиву на "більше", "менше", "рівно" тощо (хоча це незручно для рядкових змінних, якщо масив містить числа). Наприклад (якщо numbers – це поле-масив з числами):

```
db.rainbow.find({"numbers": {$lt: 20}})
```

Наприкінці зазначимо, що для вибірки ***одного документа*** в MongoDB використовується метод findOne(). На відміну від методу find(), який повертає курсор на множину документів, findOne() повертає лише перший документ, що задовольняє умову пошуку.

Наприклад, вибірка інформації про викладача з кодом 3:

```
db.lecturers.findOne({ lectKod: 3 })
```

Метод findOne() доцільно використовувати у випадках, коли:

- очікується єдиний результат;
- виконується пошук за унікальним ключем, зокрема, при вбудовуванні в документ даних з іншої колекції (див. приклад розділу «Основи додавання даних») або формуванні посилань між документами різних колекцій;
- необхідно отримати лише один документ без необхідності обробки множини результатів.

Оновлення даних (Update)

Для оновлення даних необхідно вказати певний **фільтр**, подібний до тих, що використовувались для вибору даних. Це дозволяє точно визначити, які записи будуть змінені.

У MongoDB існують дві основні функції для оновлення даних:

- **updateOne**, яка дозволяє оновити лише **один запис**, навіть якщо фільтр відповідає кільком записам. Вона оновлює перший знайдений запис.
- **updateMany**, навпаки, дозволяє оновити **кілька записів**, які відповідають заданому фільтру.

Синтаксис для оновлення даних виглядає так:

```
db.collectionName.update[One/Many](filter, updateOperation)
```

Перший об’єкт – це **фільтр**. Він може бути будь-яким фільтром, який ми вивчали раніше, наприклад, фільтри "більше/менше" або перевірка кількох значень.

Другий об’єкт – це **операція оновлення**. Зазвичай використовується оператор $set.

Наприклад, щоб змінити оцінку 10 на 60 з дисципліни "Мережеві технології" для студента з kod=1:

```
db.students.updateOne(
 { kod: 1, "ratings.discipline": "Мережеві технології", "ratings.mark": 10 },
 { $set: { "ratings.$.mark": 60 } }
)
```

… або для всіх студентів з балом 10:

```
db.students.updateMany(
 { "ratings.discipline": "Мережеві технології", "ratings.mark": 10 },
```

```
  { $set: { "ratings.$.mark": 60 } }
)
```

Позиційний оператор $ використовується для доступу до елемента масиву, який відповідає умовам фільтра. Він позначає перший елемент масиву, що задовольняє умову, і дозволяє змінювати його поля без необхідності знати його індекс.

⚠ *Навіть у випадку updateMany оператор $ у кожному знайденому документі оновлює тільки ПЕРШИЙ відповідний елемент масиву.*

Для оновлення ВСІХ елементів масиву треба використати оператор arrayFilters:

```
db.students.updateMany(
 {},
 {
  $set: { "ratings.$[r].mark": 60 }
 },
 {
  arrayFilters: [
   { "r.discipline": "Мережеві технології", "r.mark": 10 }
  ]
 }
)
```

Важливо зазначити, що за допомогою $set можна змінювати **кілька полів одночасно**, і ці поля не обов'язково повинні бути тими, за якими було фільтрування.

Наприклад, студентам обраної спеціальності (наприклад, "КІ") масово оновити контактні дані (email) за заданим правилом (наприклад, замінити домен).

```
db.students.updateMany(
 { speciality: "КІ" },
 [
  {
   $set: {
    email: {
     $replaceOne: {
      input: "$email",
      find: "oldmail.com",
      replacement: "onu.edu.ua"
     }
    }
   } } ] )
```

Заміна об’єктів (Replace One)

На відміну від update, яка змінює певні поля, функція **replaceOne** дозволяє повністю **замінити об’єкт**. Ви вказуєте новий об’єкт з абсолютно новими полями та значеннями.

При використанні replaceOne вказується фільтр для пошуку об’єкта, який потрібно замінити, а потім надається **повністю новий об’єкт**, яким буде замінено існуючий. Наприклад:

```
// Повна заміна документа дисципліни dkod=5
db.disciplines.replaceOne(
 { dkod: 5 },
 { dkod: 5, name: "Дослідження операцій", hours: 72, semesters: 1,
coursework: false, currentSemester: 7 }
)
```

Якщо не вказати певні поля в новому об’єкті, вони будуть відсутні або встановлені як null/NAN в заміненому записі. Це означає, що старі поля, які не були вказані в новому об’єкті, будуть втрачені. Також, можна встановлювати **різні типи даних** для полів, оскільки MongoDB є дуже гнучкою базою даних.

Концепція bulkWrite …

… дозволяє об’єднати декілька команд. Оскільки команд може бути багато, bulkWrite приймає масив, де кожна команда прописується як окремий об’єкт.

Ось як це виглядає на початковому етапі:

```
db.lecturers.bulkWrite([ // тут будуть потрібні команди ])
```

Розглянемо приклади команд, які можна об’єднати.

Задача: Сформувати пакетну операцію bulkWrite для колекції students (та за необхідністю – disciplines), що буде містити наступні операції:

1) додати нового студента;
2) оновити контактні дані (email) студента;
3) видалити записи, які відповідають умові «сміттєвих/тестових даних»;
4) замінити (або створити) документ дисципліни із зазначеним dkod (upsert).

Відповідно, ***першою командою*** буде insertOne – для вставки одного нового об’єкта.

Для insertOne вказуємо поле document, в якому міститься сам об’єкт, що додається.

Наприклад:

```
{
 insertOne: {
 document: {
 kod: 99,
 secondName: "Філатова",
```

```
 firstName: "Тетяна",
 patronymic: "В’ячеславівна",
 snum: 15,
 gnum: 943,
 email: "filatova@ukr.net",
 spec: { code: "IT", name: "Інформаційні системи і технології" },
 address: { city: "м. Одеса", street: "вул. Невідомого", house: 15, flat: 1 },
 ratings: [],
 diploma: null
 } }
}
```

Наступна команда updateOne або updateMany, для якої прописано поле filter, потім додається поле update, де за допомогою $set вказується, які поля треба змінити та на які нові значення. Наприклад:

```
{
 updateMany: { // Замість updateOne, для оновлення багатьох
 filter: { kod: 1 },
 update: { $set: { email: "bvgo12345@ukr.net" } }
 }
}
```

Далі буде **видалення**, де, як і для оновлення, вказуємо filter.

⚠ *Важливо пам'ятати, що якщо фільтр буде повністю порожнім, це означатиме видалення всіх об'єктів у базі даних – треба бути обережними з цим!*

Наприклад:

```
{
 deleteMany: {
 filter: { $or: [{secondName: "Блажко"}, {gnum: { $gte: 990 }}] }
 }
}
```

І ***остання команда*** – це replaceOne: повна заміна одного об’єкта іншим.

Знову ж таки, вказується filter для вибору об’єкта, наприклад, dkod: 6. Однак, на відміну від updateMany, тут не використовується $set. Замість цього, після фільтра, просто вказується повністю новий об’єкт, на який буде замінено існуючий. Це дозволяє задати нові значення для всіх полів об’єкта, або ж опустити ті поля, які не потрібні в новому об’єкті. Наприклад:

```
{
 replaceOne: {
 filter: { dkod: 6 },
 replacement: {
```

```
  dkod: 6,
  name: "NoSQL Бази даних",
  hours: 90,
  semesters: 1,
  coursework: false,
  currentSemester: 6
  },
  upsert: true
  }
}
```

У навчальних прикладах зазвичай показують "replaceOne" як "повна заміна документа". Але для того, щоб операція була практичною, включаємо upsert: true, щоб за відсутності зазначеного dkod документ створювався.

Після того, як всі ці команди зібрани в один масив bulkWrite, його можна виконати. Система додає новий об'єкт, видаляє один, оновлює декілька записів і замінює один об'єкт – і все це відбувається *майже* за один запит!

«Майже» тому, що в прикладі йде робота з двома колекціями, і через це використовуються дві команди bulkWrite:

```
use lectures

db.students.bulkWrite([
  // Крок 1: insertOne – додати студента
  {
  insertOne: {
…
  },

  // Крок 2: updateMany – оновити email студенту kod=1
  {
  updateOne: {
…
  },

  // Крок 3: deleteMany – видалити тестові/сміттєві записи
  {
  deleteMany: {
…
  }
], { ordered: true })

// Повна заміна/створення дисципліни логічно відноситься до пакету,
```

```
// але виконується в колекції disciplines, тому оформлена окремим
bulkWrite:
db.disciplines.bulkWrite([
 // Крок 4: replaceOne + upsert – замінити/створити дисципліну dkod=6
 {
 replaceOne: {
…
 }
], { ordered: true })
```

Таким чином, bulkWrite є потужним інструментом в MongoDB, який дозволяє ефективно управляти даними, виконуючи безліч команд всього за один запит. Це значно оптимізує роботу з базою даних та спрощує логіку вашого коду.

Додаткові приклади обробки даних в MongoDB

В якості прикладів ефективного маніпулювання колекціями як обробки даних в MongoDB для отримання потрібної інформації розглянемо три ключові операції: підрахунок об’єктів, отримання унікальних значень та агрегацію даних.

Метод count дозволяє нам підраховувати об’єкти за певним фільтром.

Наприклад: дізнатися, скільки об’єктів (студентів) є на спеціальності "КМ":

```
db.students.countDocuments({ "spec.code": "КМ" })
```

Можна задавати будь-який фільтр, включаючи комбінації декількох полів або умови більше/менше. Метод count просто підраховує загальну кількість записів за заданим фільтром.

Метод distinct

Задача: знайти унікальні кафедри, на яких працюють викладачі, та вивести список цих значень (без повторів).

Якщо просто використати команду find, будуть отримані всі назви кафедр, включаючи дублікати, оскільки в нашому випадку декілька об’єктів мають однакові кафедри.

Щоб отримати лише унікальні значення без дублікатів, використовується функція distinct:

```
db.lecturers.distinct("dept.deptName")
```

Агрегація даних – це процес узагальнення, групування та обробки даних для отримання підсумкової або аналітичної інформації.

Наприклад:

Задача А1: Побудувати агрегований звіт: для студентів обраної спеціальності (наприклад, "КІ") обчислити *середній бал за всіма оцінками* та *кількість оцінок* (скільки записів оцінювання враховано). Результат треба надати, як один документ виду {avgMark, cnt}.

Задача А2: Побудувати агрегований звіт: для студентів обраної спеціальності (наприклад, "КІ") обчислити *середній бал окремо з кожної*

дисципліни та *кількість оцінок з кожної дисципліни*. Результат має мати вигляд набору документів виду { discipline, avgMark, cnt }, відсортований по avgMark за спаданням.

У MongoDB агрегатна обробка даних реалізується засобами ***Aggregation Framework*** – підсистеми обробки та аналізу даних, яка базується на послідовному виконанні етапів обробки, що формують так званий ***агрегатний конвеєр*** *(aggregation pipeline)*.

Кожний етап конвеєра виконує окрему операцію над потоком документів (фільтрацію, групування, сортування, перетворення тощо), а результат одного етапу передається на наступний.

Для агрегації використовується метод aggregate(), який приймає масив етапів (стадій) обробки даних:

```
db.students.aggregate([
 // Етап 1: $match (Фільтрація)
 // Етап 2: $group (Групування)
])
```

Етап 1: $match (Фільтрація)

```
{ $match: { "spec.code": "КІ" } }
```

Етап 2: $group (Групування)

Тут ключовий елемент – _id – це поле, за яким відбувається групування документів.

Наприклад, _id: "$ratings.discipline" означає, що об’єкти будуть групуватися за унікальним значенням поля discipline.

Саме у межах цього етапу ($group) можуть виконуватись **агрегатні обчислення.**

Для підрахунку середнього балу вказуємо нове поле, наприклад avgMark, та використовуємо оператор $avg для поля mark:

```
avgMark: { $avg: "$ratings.mark" }
```

Повний запит агрегації виглядатиме так:

```
db.students.aggregate([
 { $match: { "spec.code": "КІ" } },
 { $unwind: "$ratings" },
 {
 $group: {
 _id: null                    // для задачі А1, або "$ratings.discipline" – для
задачі А2,
 avgMark: { $avg: "$ratings.mark" },
 cnt: { $sum: 1 }
 }
```

```
  },
  { $sort: { avgMark: -1 } }
])
```

Оператор $unwind використовується для "розгортання" масиву: для кожного елемента масиву формується окремий документ. Тобто документ логічно дублюється, але масив замінюється одним із його елементів.

⚠ *$unwind збільшує кількість документів у потоці обробки, що може впливати на продуктивність.*

Питання для самоперевірки

(до розділу «RU-операції (маніпулювання даними)»)

1. Які операції MongoDB відносяться до RU-операцій?
2. У чому полягають особливості вибірки даних у MongoDB?
3. Яке призначення методу find()?
4. Чим findOne() відрізняється від find()?
5. Як у MongoDB задаються умови фільтрації документів?
6. Які оператори порівняння використовуються у MongoDB?
7. Як у MongoDB реалізуються логічні операції AND, OR та NOT?
8. У чому полягають особливості роботи з вкладеними документами та масивами при вибірці даних?
9. Як у MongoDB реалізується сортування та обмеження кількості результатів вибірки?
10. У чому полягають особливості оновлення документів у MongoDB?
11. Яке призначення операторів $set, $unset, $inc та $rename?
12. Для чого використовується positional operator $?
13. У яких випадках доцільно використовувати replaceOne()?
14. Чим replaceOne() концептуально відрізняється від updateOne()?
15. Для чого використовується bulkWrite()?
16. Які переваги та ризики пакетного виконання операцій через bulkWrite()?
17. У чому полягають особливості роботи MongoDB з масивами даних?
18. Які можливості надає Aggregation Framework для обробки та аналізу даних у MongoDB?
19. Чому Aggregation Framework часто розглядають як аналог агрегатної та аналітичної обробки даних у SQL?

Завдання для самостійної роботи

Написати запити для розв’язання задач, наведених для обраної ПрО розділу «Завдання для самостійної роботи» (с. 31) з використанням колекцій, створенних при виконанні «Завдання для самостійної роботи» (с. 48). Використовуючи інструментарій описаний в розділі «Інструменти для роботи з MongoDB», виконати створені запити для обробки документів, введених до БД при виконанні «Завдання для самостійної роботи» (с. 48).

Демонстраційний приклад

Завдання

Написати запити для розв'язання задач, наведених для ПрО розділу «Демонстраційний приклад» (с. 34) з використанням колекцій, створенних при виконанні «Демонстраційний приклад» (с. 48).

Розв'язання

```
// Вибір БД для роботи
use student_projects_db
```

ДД2.1. Пошук проєктів за технологією

```
// Пошук документів, у яких масив technologies містить значення
"MongoDB"
db.projects.find(
 { technologies: "MongoDB" },

 // Проекція: виводити лише потрібні поля
 {
  _id: 0,
  project_id: 1,
  title: 1,
  project_type: 1,
  technologies: 1,
  updated_at: 1
 } )

// Сортування результатів за датою останнього оновлення
.sort({ updated_at: -1 })
```

ДД2.2. Текстовий пошук з релевантністю (score)

Для реалізації повнотекстового пошуку MongoDB потребує створення ***текстового індексу***.

Тому розв'язання задачі буде наведено в «Демонстраційному прикладі» після розділу «Індекси» (с. 68).

ДД2.3. Розгортання масиву stages

```
db.projects.aggregate([

// Оператор $unwind розгортає масив stages:
// кожний елемент масиву стає окремим документом у потоці
обробки
{
 $unwind: "$stages"
},
```

```
// Формування структури результату
{
 $project: {
  _id: 0,
  project_id: 1,
  title: 1,

  // Вибір полів вкладеного документа stages
  stage_name: "$stages.stage_name",
  status: "$stages.status",
  date: "$stages.date"
 }
},

// Сортування етапів за датою
{
 $sort: {
  project_id: 1,
  date: 1
 }
}

] )
```

ДД2.4. Оновлення документа проєкту

```
// Оновлення документа проєкту
db.projects.updateOne(


 { project_id: 101 },              // Пошук документа за project_id

 {

  // Додавання нового етапу до масиву stages
  $push: {
   stages: {
    // Новий вкладений документ
    stage_name: "Тестування",
    status: "planned",
    date: ISODate("2026-03-20")
   }
  },
```

```
    // Оновлення дати останньої модифікації документа
    $set: {
     updated_at: ISODate("2026-03-20")
    }
   }
)

// Додавання нового відгуку керівника
db.projects.updateOne(

 { project_id: 101 },

 {

   // Додавання документа до масиву supervisor_reviews
   $push: {
    supervisor_reviews: {
          supervisor: "доц. Шпінарева І.М.",
          rating: 97,
         comment: "Реалізація проєкту відповідає поставленим
вимогам."
    }
   },

   $set: {
    updated_at: ISODate("2026-03-22")
   }
 }
)
```

ДД2.5. Найчастіше використані технології

```
db.projects.aggregate([

{
 $unwind: "$authors"          // Розгортання масиву authors
},

// З'єднання даних з колекцією students
{
 $lookup: {
   from: "students",
   localField: "authors.student_id",
```

```
    foreignField: "student_id",
    as: "student"
  }
},

{
  $unwind: "$student"            // Розгортання результату lookup
},

// Відбір студентів певної групи
{
  $match: {
    "student.group": "IC-31"
  }
},

{
  $unwind: "$technologies"     // Розгортання масиву technologies
},

// Групування за технологіями
{
  $group: {

    _id: "$technologies",

    // Підрахунок кількості використань технології
    count: {
      $sum: 1
    }
  }
},

// Сортування за спаданням частоти використання
{
  $sort: {
    count: -1
  }
}

] )
```

Індекси

Так само, як і в РСУБД, MongoDB для прискорення пошуку та обробки даних можна створювати індекси.

Індекс охоплює запит, коли містить всі поля, скановані запитом. Охоплений запит сканує індекс, а не колекцію, що збільшує продуктивність запиту. Індекси також можуть частково підтримувати запити, якщо підмножина полів, що запитуються, індексована.

В одній колекції може бути не більше ніж 64 індекси. Але,

⚠ *Як і в SQL, занадто велика кількість індексів може знизити продуктивність ще до того, як цей ліміт буде досягнуто. Для колекцій з високим співвідношенням операцій запису та читання індекси можуть знижувати продуктивність, оскільки кожна вставка також має оновлювати всі індекси.*

Створення та використання індексів складається з декількох етапів.

1. Визначення найпоширеніших запитів

У локальних інсталяціях це можна зробити через MongoDB Profiler (system.profile) або аналіз slow queries у логах (див. Додаток В).

У MongoDB Atlas можна використати агрегацію [{ $queryStats: {} }] (за наявності Query Insights), який повідомляє метрики форм запитів, що групують запити з урахуванням спільних полів.

2. Створення індексів підтримки поширених запитів.

Створити індекси під поля, що найчастіше використовуються у фільтрах/сортуваннях.

3. Аналіз використання індексу

Після того, як програма почне використовувати індекси, можна проаналізувати їх ефективність. Перевірити використання індексів можна через explain() для конкретних запитів, наприклад:

```
db.<collection>.find({...}).explain("executionStats")
```

та додатково подивитися статистику через:

```
db.<collection>.aggregate([{ $indexStats: {} }])
```

Приклади

Якщо програма виконує запити лише по одному ключу в цій колекції, необхідно створити ***простий індекс*** (за одним ключем) для цієї колекції. Наприклад:

```
// Індекс за кодом спеціальності студента
db.students.createIndex({ "spec.code": 1 })
```

Запит, який використовує цей індекс, такий:

```
db.students.find({ "spec.code": "КІ" })
```

Якщо програма виконує запити як по одному, так і по кількох ключах, ***складовий індекс*** ефективніший за індекс з одним ключем. Наприклад:

```
// Складовий індекс: спеціальність + прізвище (під фільтр +
сортування)
db.students.createIndex({ "spec.code": 1, secondName: 1 })
```

Складовий індекс підтримує запити щодо ***префіксів індексу,*** які є початковими підмножинами індексованих полів. Тобто. пошук *тільки* по першому полю *складового* індексу працює як *простий* індекс. Наприклад, попередній індекс підтримує такі запити:

```
db.students.find({ "spec.code": "IT" })
db.students.find({ "spec.code": "IT", secondName: "Мікулінська" })
```

Спеціальні види індексів

Індекси для підтримки текстового пошуку

Щоб MongoDB міг ефективно шукати текст, необхідно створити ***текстові індекси*** . Індекси вказують MongoDB, на яких полях потрібно відстежувати інформацію для текстового пошуку. Наприклад:

```
// Текстовий індекс по назві активності та її деталям
db.lecturers.createIndex(
{
  "activities.name": "text",
  "activities.role": "text",
  "activities.details.url": "text"
},
{ name: "tx_lecturers_activities" }
);
```

Ця команда створює індекси для зазначених полів, дозволяючи виконувати пошук за текстом, після чого можна використовувати спеціальний фільтр для пошуку фрагментів тексту. Для цього потрібно звернутись до функції find та використати оператор $text з параметром $search.

Наприклад, пошук активностей по слову «конференція»:

```
db.lecturers.find(
{ $text: { $search: "конференція" } },
{
  secondName: 1,
  firstName: 1,
  activities: 1
}
);
```

Цікавою особливістю є те, що при пошуку кількох слів (наприклад, "… …") MongoDB шукатиме записи, які містять **будь-яке** з цих слів, використовуючи логіку «АБО» (OR-семантика).

```
db.lecturers.find(
{ $text: { $search: "конференція Одеса" } },
{
 secondName: 1,
 firstName: 1,
 activities: 1
}
);
```

Цей функціонал дозволяє легко шукати об’єкти за одним або кількома ключовими словами у зазначених полях.

І нарешті, якщо з’ясувалося, що індекси не потрібні і негативно впливають на продуктивність, необхідно ***видалити індекси*** за допомогою методу db.collection.dropIndexes. Наприклад:

```
db.students.dropIndexes(["spec.code_1", "secondName_1"])
```

Багатоключові (multikey) індекси

Якщо поле, для якого створюється індекс, містить масив, MongoDB автоматично визначає цей індекс як **багатоключовий (multikey index)**.

Для кожного елемента масиву формується окремий індексний запис, що прискорює пошук документів за елементами масиву.

⚠ *Для кожного документа у складовому multikey-індексі масивом може бути не більше одного поля, що індексується*

Наприклад, на доданок до індексів БД lectures (див. Додаток А) створити multikey-індекси:

1) за дисциплінами у масиві ratings колекції students

```
db.students.createIndex({ "ratings.discipline": 1 })
```

який буде використаний наступним запитом

```
db.students.find(
 { "ratings.discipline": "Мережеві технології" },
 { _id: 0, secondName: 1, firstName: 1, ratings: 1 }
)
```

Запит знайде і поверне документи студентів, які мають оцінку з дисципліни «Мережеві технології», у такому вигляді:

```
{
 secondName: 'Філатова',
 firstName: 'Тетяна',
 ratings: [
  {
   dKod: 1,
   discipline: 'Технології проектування БД',
```

```
    mark: 54,
    mdate: 2024-10-10T00:00:00.000Z
   },
   {
    dKod: 2,
    discipline: 'Мережеві технології',
    mark: 85,
    mdate: 2024-12-12T00:00:00.000Z
   },
   {
    dKod: 5,
    discipline: 'Програмування',
    mark: 65,
    mdate: 2024-10-11T00:00:00.000Z
   }
  ]
}
```

2) за роком публікацій у масиві publications колекції lecturers

```
db.lecturers.createIndex({ "publications.year": -1 })
```

який охоплюється запитом

```
db.lecturers.find(
 { "publications.year": { $gte: 2020 } },
 { _id: 0, secondName: 1, firstName: 1, publications: 1 }
)
```

Результат буде включати документи викладачів, у яких є публікації за останні 7 років.

TTL-індекси (індекси часу життя)

TTL-індекси (***Time-to-Live***) в MongoDB – це спеціальні прості (однополеві) індекси, які автоматично видаляють документи з колекції після заданого часу або в конкретний момент часу. Це потужний інструмент для роботи з тимчасовими сесіями, журналами, кешем чи кодами підтвердження, які повинні зберігатися в базі даних лише протягом обмеженого періоду часу [9].

Для створення TTL-індексу в аргументах методу createIndex() вказується поле, що містить значення типу Date або масив таких значень. Причому, значення параметра expireAfterSeconds вказується у секундах (межах від 0 до 2147483647 включно).

Для демонстрації роботи такого типу індексів, до БД lectures (Додаток А) додамо «службову» колекцію, яка міститиме дані про події, що відбувались з цією БД (на кшталт журналу):

```
db.createCollection("event_log")
```

Приклади таких даних можуть бути наступні:

```
db.event_log.insertMany([
 {
  eventType: "import",
  message: "Завантажено дані студентів",
  createdAt: ISODate("2026-03-01T10:00:00Z")
 },
 {
  eventType: "query",
  message: "Виконано пошук студентів за спеціальністю",
  createdAt: ISODate("2026-03-02T12:30:00Z")
 }
] )
```

Далі можна створити TTL-індекс для поля автоматичного видалення службових подій через 30 днів:

```
db.event_log.createIndex(
 { createdAt: 1 },
 { expireAfterSeconds: 60 * 60 * 24 * 30 }
)
```

Видалення документів виконується в фоновому режимі (*фоновий потік сервера* mongod), який перевіряє кожен індекс TTL.

⚠ *Після створення TTL-індексу у ньому може бути дуже велика кількість кваліфікаційних документів для одночасного видалення. Відповідне навантаження може спричинити проблеми з продуктивністю сервера.*

Геопросторові індекси

Геопросторові індекси (геоіндекси) підтримують запити до даних, що зберігаються у вигляді об’єктів GeoJSON або застарілих пар координат, і використовуються для підвищення продуктивності запитів до геопросторових даних або для виконання спеціальних геопросторових запитів.

MongoDB надає два типи геопросторових індексів:

– двовимірні (2d індекси), що підтримують запити, що інтерпретують геометрію на плоскій евклідовій площині (у сучасних застосуваннях використовуються обмежено);
– сферичні двовимірні (2dsphere індекси), що підтримують запити, що інтерпретують геометрію на сфері з урахуванням кривизни поверхні Землі.

Для створення ***сферичного індексу*** в методі createIndex() вказується тип "2dsphere":

```
db.<колекція>.createIndex ({ <місцезнаходження> : "2dsphere" } )
```

Значення <місцезнаходження> повинно бути або об’єктом GeoJSON, або парою координат (застарілий варіант) [9, geospatial-queries/#std-label-geospatial-legacy].

Для демонстрації можливостей роботи з геоіндексами створимо додаткову колекцію розташування корпусів університета:

```
db.createCollection("campus_locations")
```

яка міститиме такі документи:

```
db.campus_locations.insertMany([
 {
  code: "КСТ",
  name: "Корпус фізики",
  kind: "building",
  location: {
   type: "Point",
   coordinates: [30.7315, 46.4850]
  }
 },
 {
  code: "main",
  name: "Головний корпус університету",
  kind: "building",
  location: {
   type: "Point",
   coordinates: [30.7390, 46.4875]
  }
 }
] )
```

У властивості coordinates першою йде довгота в діапазоні [-180, 180], другою – широта в діапазоні [-90, 90].

Індекси 2dsphere дозволяють виконувати наступні обчислення з даними про місцезнаходження на сфері:

– запит на розташування поблизу точки на сфері;
– запит на розташування, обмежені багатокутником;
– запит на розташування, що перетинаються з об’єктом GeoJSON;
– запит розташування в межах кола на сфері;

Для отримання даних про місцезнаходження об’єктів, що є поблизу вказаної точки на сфері, потрібен оператор $near, наприклад:

```
db.campus_locations.createIndex({ location: "2dsphere" })
```

```
db.campus_locations.find({
 location: {
  $near: {
   $geometry: {
    type: "Point",
    coordinates: [30.7350, 46.4860]
   },
   $maxDistance: 1000
  }
 }
})
```

У властивості maxDistance вказується максимальна відстань в метрах.

Ранжування за релевантністю (Score)

Часто при пошуку важливо не просто знайти записи, а й визначити, які з них є **найбільш релевантними** до запиту. MongoDB надає функціонал, схожий на пошукові системи, такі як Google, для ранжування результатів.

Для цього ми можна додати до результатів спеціальне поле під назвою score (оцінка релевантності). Це поле буде автоматично розраховуватися MongoDB і показуватиме, наскільки кожен документ відповідає пошуковому запиту. Чим більше слів із запиту збігаються, і чим більше їх у тексті, тим вищою буде оцінка.

```
// Додати score (textScore)
// score – автоматично обчислювана оцінка релевантності
// (чим більше збігів, тим вище значення)
db.lecturers.find(
{ $text: { $search: "Odessa Одеса" } },
{
score: { $meta: "textScore" },
secondName: 1,
firstName: 1,
activities: 1
}
).sort({ score: { $meta: "textScore" } });
```

Після виконання цієї команди, до кожного знайденого об'єкта буде додано поле score. Відповідно записи з більшою кількістю збігів матимуть вищий score (рис. 11).

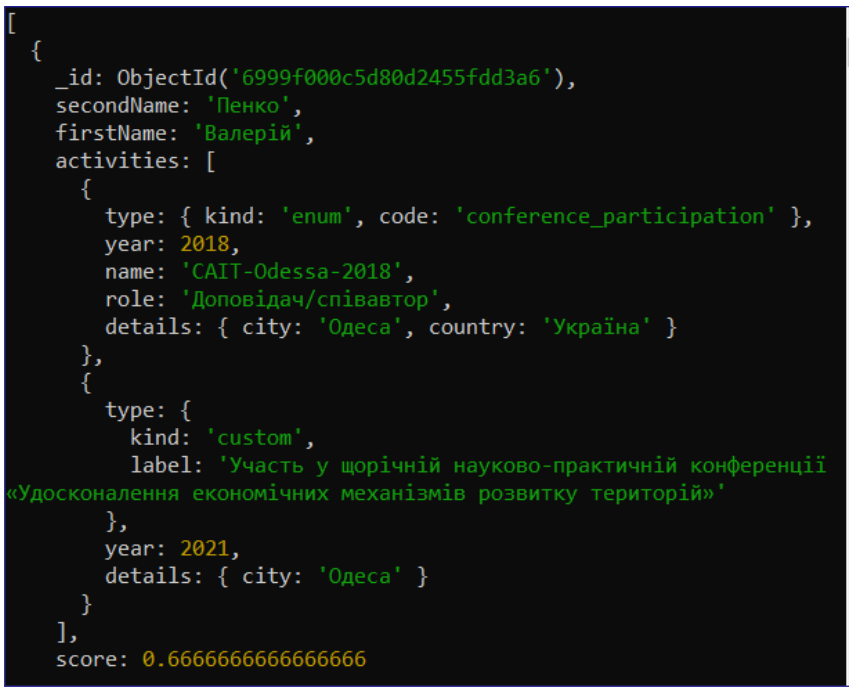


Рис. 11 – Додавання релевантності результатів пошуку

Маючи поле score, можна сортувати результати пошуку таким чином, щоб найбільш релевантні записи відображалися на початку. Це дозволяє користувачам одразу бачити найточніші збіги. Наприклад, сортування за полем score (за спаданням):

```
// Відсортувати за по релевантністю
db.lecturers.aggregate([
 { $match: { $text: { $search: "conference chair" } } },
 { $addFields: { score: { $meta: "textScore" } } },
 {
 $project: {
 score: 1,
 secondName: 1,
 firstName: 1,
 activities: {
$filter: {
input: "$activities",
as: "a",
cond: {
  $or: [
  { $regexMatch: { input: { $ifNull: ["$$a.name", ""] }, regex: /conference/i } },
  { $regexMatch: { input: { $ifNull: ["$$a.role", ""] }, regex: /chair|conference/i } }
  ]
}
}
 }
```

```
    }
    },
    { $sort: { score: -1 } }
  ]);
```

В результаті першим відображається найбільш релевантний запис, а далі – менш релевантні (рис. 12).

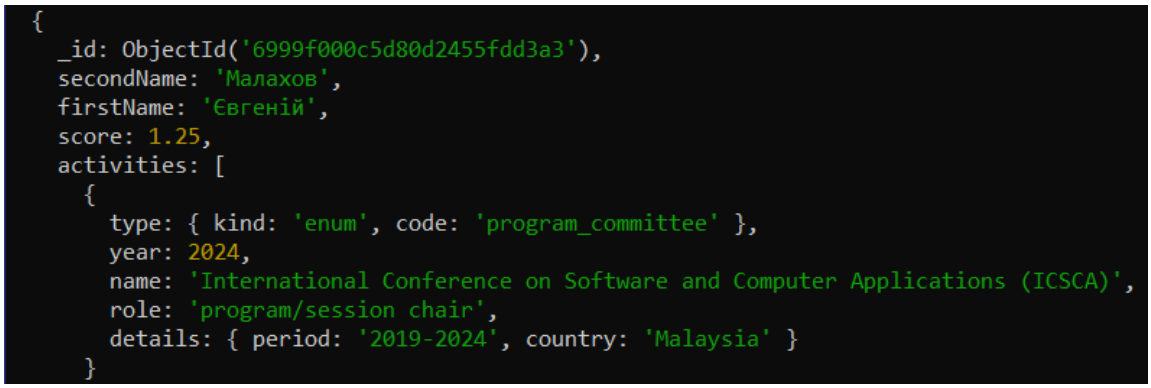


Рис. 12 – Сортування результатів пошуку за релевантністю

Практичні рекомендації

Отже, індекси є потужним інструментом для прискорення запитів MongoDB. Проте треба постійно аналізувати запити з точки зору необхідності індексації, ревізувати індекси і не індексувати все поспіль, тобто:

- індексуємо тільки те, що реально використовується у фільтрах та сортуваннях;
- застосовуємо складові індекси для найскладніших запитів;
- видаляємо індекси, що не використовуються (це економить місце і прискорює вставку);
- застосовуємо спеціальні види індексів лише тоді, коли вони відповідають характеру даних і запитів: multikey – для масивів, text – для повнотекстового пошуку, TTL – для тимчасових даних, 2dsphere – для геопросторових запитів.

Завдання для самостійної роботи

Запропонувати індекси для підвищення ефективності розв’язання задач, наведених для обраної ПрО розділу «Завдання для самостійної роботи» (с. 31) на основі колекцій, запропонованих при виконанні завдання цього розділу. Написати запити для їх побудови Використовуючи інструментарій, описаний в розділі «Інструменти для роботи з MongoDB», виконати створені запити.

Демонстраційний приклад

Завдання

Запропонувати індекси для підвищення ефективності розв’язання задач, наведених для обраної ПрО розділу «Демонстраційний приклад» (с. 34) на основі колекцій, запропонованих при виконанні завдання цього розділу. Написати запити для їх побудови Використовуючи інструментарій, описаний в розділі «Інструменти для роботи з MongoDB», виконати створені запити.

Розв’язання

Для колекцій БД ***«Портфоліо студентських проєктів»*** створимо створимо кілька типів індексів, що відповідають характеру запитів до колекцій демонстраційної БД.

1. B-tree-індекси

Через те, що ***B-tree***-індекси використовуються для пришвидшення пошуку проєктів за тематикою, реалізації тематичної навігації, ефективної роботи діапазонних умов та оптимізації фільтрації за окремими полями, для БД демонстраційного прикладу доцільними є наступні індекси:

Індекс для пошуку проєктів за технологіями

Запити, що розв’зуть задачі типу ДД2.1, часто виконують пошук за масивом technologies та сортування за датою оновлення:

```
db.projects.createIndex(
 {
  technologies: 1,        // multikey-індекс для масиву
  updated_at: -1          // підтримка сортування за датою оновлення
 }
)
```

Індекс для пошуку студентів за групою

```
db.students.createIndex(
 {
  group: 1
 }
)
```

Такий індекс прискорює відбір студентів певної групи і агрегування даних із фільтрацією документів за групою.

Індекс для тегів проєктів

```
db.projects.createIndex(
 {
  tags: 1
 }
)
```

Індекс оптимізує пошук за тематикою, фільтрацію проєктів та навігацію за тегами.

2. Текстові індекси

Для задачі ДД2.2 необхідно забезпечити пошук за назвою проєкту та/або за описом проєкту. Тому доцільно створити текстовий індекс для полів title та description:

```
db.projects.createIndex(
 {
  title: "text",
  description: "text"
 }
)
```

Саме тепер можна коректно ***розв'язати задачу ДД2.2***, оскільки для оператора `$text` у MongoDB є необхідний текстовий індекс.

ДД2.2. Пошук проєктів за ключовими словами

```
db.projects.find(

 {
   $text: {                          // виконує повнотекстовий пошук
   $search: "студентські проєкти"       // задає пошуковий рядок
  }
 },

 // Виведення score — показника релевантності документа
 {
  _id: 0,
  project_id: 1,
  title: 1,
  project_type: 1,

  score: {
   $meta: "textScore"    // повертає показник релевантності документа
  }
 }

)

// Сортування результатів за спаданням релевантності
.sort({
 score: {
  $meta: "textScore"
 }
})
```

Сортування за textScore дозволяє вивести найбільш релевантні документи першими.

Моделювання даних

Моделювання даних відноситься до організації даних у базі даних та зв'язків між пов'язаними сутностями.

Як було зазначено раніше, проектування документальної БД починається з визначення навантаження та аналізу фільтрів. Однак і в рамках цієї моделі доцільно спроектувати початкову схему. Процес проектування схеми допомагає підготувати ефективну роботу програми. При проектуванні схеми може знадобитися знайти баланс між продуктивністю і простотою. Іноді досягнення максимальної продуктивності потрібно безліч ітерацій і ретельне тестування. Залежно від програми та важливості оптимізації, можливо, варто створити просту схему, що охоплює базову функціональність, перш ніж витрачати час на оптимізацію. Дотримання цього процесу допомагає визначити, які дані необхідні вашому застосунку і як краще організувати їх для оптимізації продуктивності.

Надалі схему можна ітеративно змінювати в міру зміни потреб програми. MongoDB надає способи безперешкодної зміни схеми без простоїв. Однак зміна великомасштабних схем, що використовуються в робочому середовищі, може бути скрутною.

Як правило, документи в колекції мають схожу структуру. Для забезпечення узгодженості моделі даних можна створити правила перевірки схеми.

Гнучка модель даних дозволяє організувати дані відповідно до потреб програми і корисна у наступних сценаріях:

1. Компанія відстежує, у якому відділі працює кожен працівник. Можна вбудувати інформацію про відділ у колекцію співробітників, щоб повертати відповідну інформацію в одному запиті.
2. Програма електронної комерції показує п'ять останніх відгуків при відображенні товару. Можна зберігати останні відгуки в тій же колекції, що й дані про товар, а старіші відгуки – в окремій колекції, оскільки до них звертаються рідше.
3. Магазину необхідно створити односторінковий додаток для каталогу товарів. Різні товари мають різні атрибути і, отже, використовують різні поля документа. Однак, можна зберігати всі товари в одній колекції.

При проектуванні схеми для документної бази даних, як MongoDB, слід враховувати кілька важливих відмінностей від реляційних баз даних (таблиця 20).

Таблиця 20 – Відмінності при моделюванні реляційних та документних баз даних

Поведінка РБД	Поведінка БД документів
Перед вставкою даних слід визначити схему таблиці.	Схема може змінюватися з часом відповідно до змін потреб програми.
Часто доводиться об'єднувати дані з різних таблиць, щоб отримати дані, необхідні застосунку.	Гнучка модель даних дозволяє зберігати дані відповідно до того, як програма повертає дані, та уникати об'єднань. Відсутність об'єднань між кількома колекціями підвищує продуктивність та знижує навантаження на розгортання.

Планування схем

Щоб забезпечити логічну структуру моделі даних та оптимальну продуктивність, необхідно спланувати схему до використання бази даних у виробничих масштабах. Для визначення моделі даних використовується наступний процес проектування схеми:

1. Визначення робочого навантаження програми.
2. Зіставлення взаємозв'язків між об'єктами у колекціях.
3. Застосування шаблонів проектування.

При проектуванні моделі даних MongoDB необхідно враховувати структуру документів і те, як застосунок використовує дані з пов'язаних сутностей.

Щоб зв'язати пов'язані дані, можна:

– вбудовувати пов'язані дані до одного документа.
– зберігати пов'язані дані в окремій колекції та звертатися до неї за посиланням.

Вбудовані документи зберігають пов'язані дані у єдиній структурі документа. Документ може містити масиви та піддокументи із пов'язаними даними. Ці ***денормалізовані*** моделі даних дозволяють програмам витягувати пов'язані дані за одну операцію з базою даних.

```
{
 _id: ObjectId("65f00000000000000000101"),
 clientname: "xyz”,
 contact: {
  phone: "123-456-7890",  // Вбудований піддокумент
  email: "xyz@example.com”
 }
 access: {
  level: 5,                          // Вбудований піддокумент
  group: "manager"
 }
}
```

Для багатьох випадків використання MongoDB денормалізована модель даних є оптимальною.

Посилання зберігають зв'язки між даними, включаючи **посилання** з одного документа до іншого (рис. 13).

Програми можуть використовувати ці посилання для доступу до пов'язаних даних. У широкому сенсі це ***нормалізовані*** моделі даних.

На планування моделі даних можуть вплинути такі чинники.

1. Дублювання та узгодженість даних

При впровадженні пов'язаних даних до одного документа можна дублювати дані між двома колекціями. Дублювання даних дозволяє застосунку вимагати пов'язану інформацію про кілька сутностей в одному запиті, логічно розділяючи сутності моделі.

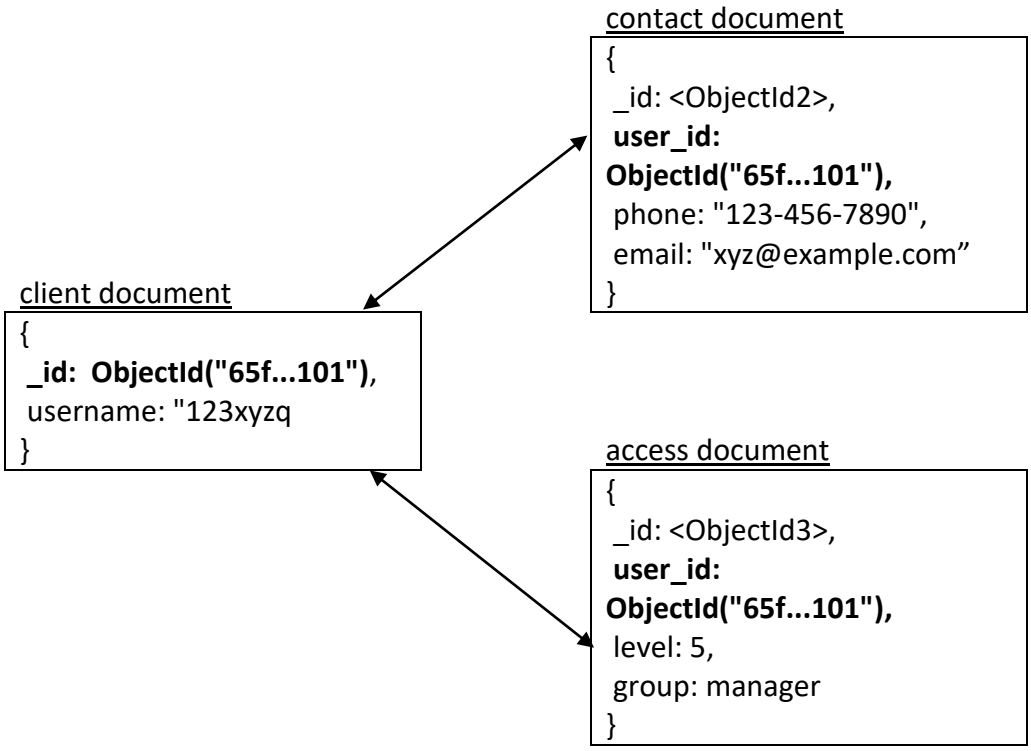


Рис. 13 – Приклад посилання з одного документа до іншого.

Наприклад, реалізувати дублювання «останніх 5 оцінок» усередині документа студента (у students.lastRatings), зберігати «всі оцінки» окремо (наприклад, у rating_events) і підтримувати узгодженість при додаванні нової оцінки.

При отриманні нової оцінки виконуються наступні дії:

- оцінку додано до колекції оцінок;
- масив останніх оцінок у колекції студентів оновлюється за допомогою $pop і $push.

Якщо дані, що дублюються, оновлюються нечасто, то для підтримки узгодженості двох колекцій знадобиться лише мінімальна додаткова робота. Однак, якщо дубльовані дані оновлюються часто, використання зв’язку зв’язаних даних може виявитися більш ефективним підходом.

Перш ніж копіювати дані, необхідно взяти до уваги такі фактори:

- як часто необхідно оновлювати дубльовані дані;
- виграш у продуктивності під час читання під час дублювання даних.

2. Індексація

Щоб підвищити продуктивність запитів, часто виконуваних застосунком, створюються індекси для полів, що часто запитуються. У міру зростання програми необхідно відстежувати використання індексів, щоб переконатися, що індекси, як і раніше, підтримують релевантні запити.

3. Апаратні обмеження

При проектуванні схеми необхідно враховувати апаратне забезпечення розгортання, особливо обсяг доступної оперативної пам’яті. Документи

більшого розміру використовують більше оперативної пам’яті, що може призвести до читання даних із диска та зниження продуктивності. По можливості схема проектується так, щоб запити повертали лише релевантні поля. Це гарантує, що робочий набір програми не буде надмірно розростатися.

4. Атомарність окремого документа

У MongoDB операція запису є атомарною лише на рівні одного документа, навіть якщо вона змінює кілька вкладених документів усередині одного документа. Це означає, що якщо операція оновлення торкається кількох вкладених документів, то всі ці вкладені документи будуть оновлені, або операція повністю завершиться невдачею і оновлення не будуть виконані.

Денормалізована модель даних із вбудованими даними поєднує всі пов’язані дані в одному документі замість нормалізації за декількома документами та колекціями. Ця модель даних допускає атомарні операції, на відміну нормалізованої моделі, де операції зачіпають кілька документів.

Розробка схеми

Процес проектування схеми складається з наступних етапів:

1) **визначення робочого навантаження,** тобто операцій, які програма виконує найчастіше.
2) **зіставлення відношень,** тобто визначення взаємозв’язків даних предметної області та, відповідно, слід зв’язати чи вбудувати пов’язані дані.
3) **застосування шаблонів проектування** для оптимізації читання та запису.
4) **створення індексів** для підтримки найпоширеніших шаблонів запитів.

Визначення робочого навантаження програми

Першим кроком у процесі проектування схеми є визначення операцій, які програма виконує найчастіше. Знання найчастіших запитів програми допоможе створити ефективні індекси та мінімізувати кількість звернень застосунка до БД.

При оцінці робочого навантаження програми враховуються сценарії, які вона підтримує нині, і сценарії, які може підтримувати у майбутньому. Схема має працювати всіх етапах розробки докладання.

Спочатку необхідно визначити, які дані необхідні застосунку з урахуванням наступних факторів:

- список користувачів програми та необхідна їм інформація;
- предметна сфера системи;
- журнали застосунків та часто виконувані запити.

Результат такого аналізу зручно оформити у вигляді наступної таблиці завдань, які має виконувати додаток (таблиця 21).

Таблиця 21 – Робоче навантаження застосунку

Дія	Тип запиту	Інформація	Частота	Пріоритет
Дія, яку користувач робить для запуску запиту	Тип запиту (читання чи запис)	Поля документа, які записуються або повертаються запитом	Як часто програма виконує запит: часто виконувані запити виграють від використання індексів і мають бути оптимізовані, щоб уникнути пошукових операцій	Наскільки важливим є запит для програми

Наприклад, задача: скласти таблицю «рабочого навантаження» для ПрО «Навчальний процес» (таблиця 22).

Таблиця 22 – Приклад робочого навантаження для ПрО «Навчальний процес»

№	Операція	Колекція	Тип	Частота	Пріоритет оптимізації	Коментар до схеми
1	Перегляд картки студента	students (ПІБ, спец., адреса, оцінки, диплом)	Читання	Дуже часто	Високий	Документ має бути «самодостатнім» (вбудовані ratings, diploma)
2	Пошук студентів за фахом	students	Читання	Часто	Високий	Індекс за spec.code
3	Пошук студента на прізвище	students	Читання	Часто	Середній	Індекс по secondName
4	Додавання оцінки студенту	students	Запис	Часто	Високий	$push до масиву ratings
5	Формування середнього балу з дисципліни	students	Читання (aggregate)	Періодично	Середній	$unwind + $group
6	Перегляд списку викладачів кафедри	lecturers	Читання	Часто	Високий	Індекс з dept.deptKod
7	Додавання нового викладача	lecturers	Запис	Рідко	Низький	Проста вставка
8	Пошук активностей за ключовим словом	activities	Читання	Періодично	Середній	Text index
9	Формування звіту за спеціальністю	students	Читання (aggregate)	Періодично	Середній	$match + $group
10	Пакетне оновлення (bulkWrite)	students, disciplines	Запис	Рідко	Середній	Використовується адміністратором

Зіставлення пов’язаних даних у схемі

Після визначення робочого навантаження програми наступним кроком у процесі проектування схеми є зіставлення пов’язаних даних у схемі.

При проектуванні схеми враховується, як додаток має вимагати та повертати пов’язані дані. Те, як відображаються зв’язки між сутностями даних, впливає на продуктивність та масштабованість програми.

Рекомендований спосіб обробки пов'язаних даних – вбудовування в піддокумент. Вбудовування пов'язаних даних дозволяє застосунку вимагати необхідні дані однією операцією читання, уникаючи повільних операцій пошуку.

У деяких випадках можна використовувати посилання, яке вказує на пов'язані дані в окремій колекції.

Щоб визначити, чи потрібно вбудовувати пов'язані дані або використовувати посилання, необхідно розглянути відносну важливість таких цілей для програми:

- якщо програма часто запитує одну сутність для повернення даних про іншу сутність, дані вбудовуються, щоб уникнути необхідності частих операцій `$lookup`;
- якщо програма повертає дані із зв'язаних сутностей разом, дані об'єднуються в одну колекцію;
- якщо програма часто оновлює пов'язані дані, доцільно зберігати дані в окремій колекції та використовувати посилання для доступу до неї. Використання посилань знижує навантаження на запис у застосунку, оскільки дані оновлюються лише в одному місці.

Отже:

1. **Визначаються пов'язані дані у схемі,** які запитує додаток, і як сутності співвідносяться друг з одним.

Для цього необхідно розглянути операції, визначені в робочому навантаженні програми на першому етапі проектування схеми, звернувши увагу на інформацію, яку записують і повертають ці операції, а також на те, яка інформація перетинається між декількома операціями.

2. Створюється схема/діаграма для пов'язаних даних.

Діаграма повинна відображати пов'язані поля даних і тип зв'язку між цими полями (один до одного, один до багатьох, багато до багатьох).

Схематична карта/діаграма може нагадувати модель «сутність-зв'язок».

3. Визначається, чи потрібно вбудовувати пов'язані дані чи використовувати посилання.

Рішення про вбудовування даних або використання посилань залежить від поширених запитів програми. Перегляньте запити, які ви визначили на етапі визначення робочого навантаження програми, щоб розробити схему для підтримки частих та критично важливих запитів.

Наприклад, розглянемо наступну схему фрагмента системи «Навчальний процес» (рис. 14)

Далі можна оптимізувати схему для різних запитів, залежно від потреб програми.

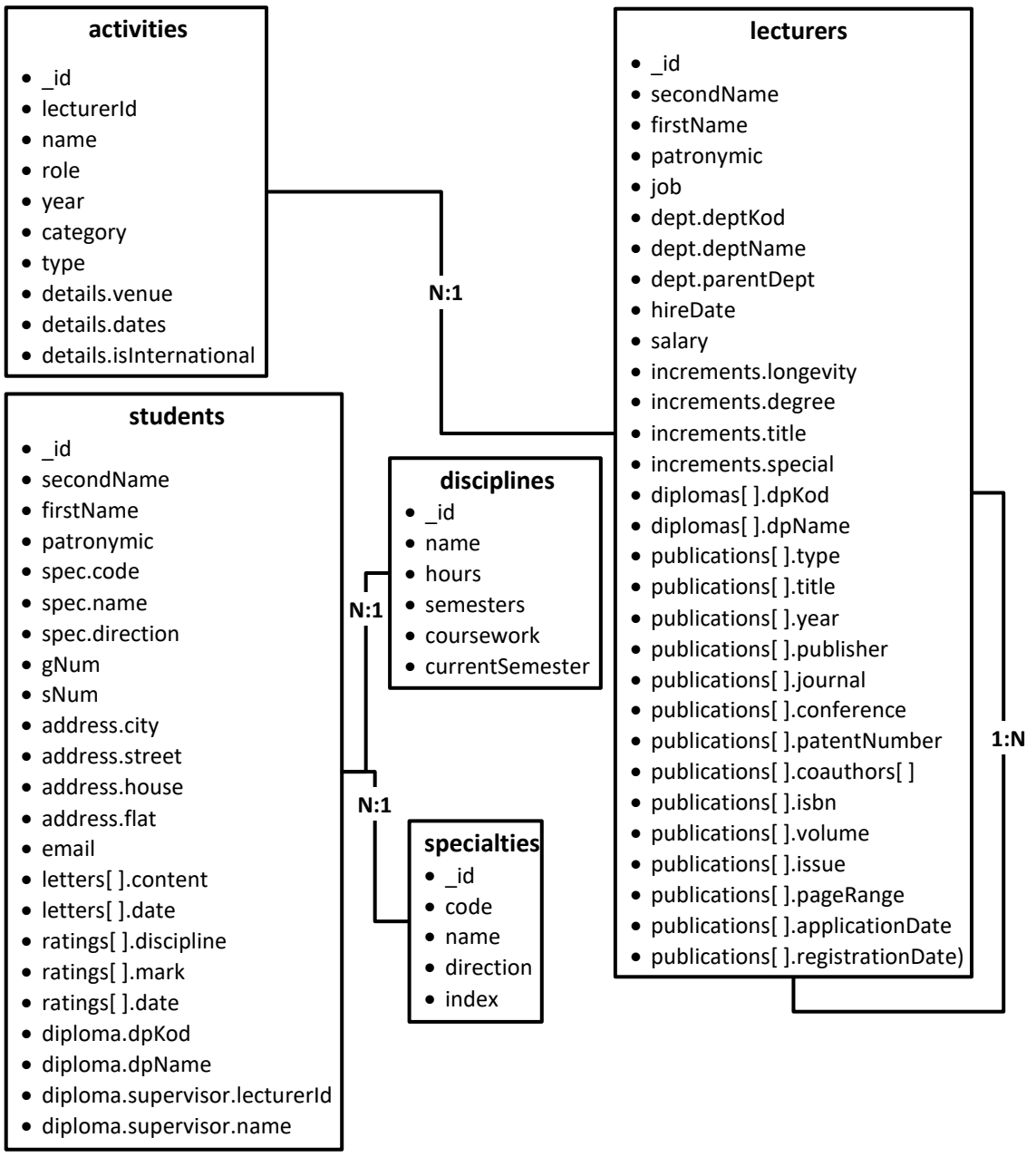


Рис. 14 – Фрагмент діаграми БД «Навчальний процес»

Наприклад, оптимізація схеми під запит «картка студента»: щоб одним запитом отримувати ПІБ, спеціальність, адресу, останні листи, оцінки та дипломну роботу (включаючи тему та керівника).

```
db.students.insertOne({  kod: 2,  secondName: "Арсірій",  firstName: "Олена",
 patronymic: "Олександрівна",
 spec: db.specialties.findOne({ code: "IT" }, { _id: 0 }),
 ratings: [  { discipline: "Технології проектування БД", mark: 90, date:
                                        ISODate("2011-10-10T00:00:00Z") }  ],
 diploma: {  dpkod: 1,  title: "Тема 1",
            supervisor: db.lecturers.findOne({ kod: 4 }, { _id: 0, kod: 1,
                                  secondName: 1, firstName: 1, patronymic: 1 })
          } } )
```

Інший приклад – оптимізація схеми під окремі запити «список викладачів кафедри» та «картка студента»: студент зберігає лише supervisorKod, а дані керівника беруться окремим запитом/через $lookup.

```
// students (тільки посилання на керівника)
db.students.insertOne({
 kod: 12, secondName: "Блажко", firstName: "Олександр", patronymic:
"Анатолійович",
 spec: { code: "КІ" },
 diploma: { dpkod: 9, title: "Тема 9", supervisorKod: 1 }
})

// lecturers
db.lecturers.insertOne({ kod: 1, secondName: "Малахов", firstName:
"Євгеній", patronymic: "Валерійович" })
```

Відповідно, запит для отримання зв’язаних диних може виглядати наступним чином:

```
db.students.aggregate([
 { $match: { kod: 12 } },
 {
  $lookup: {
   from: "lecturers",
   localField: "diploma.supervisorKod",
   foreignField: "kod",
   as: "supervisor_info"
  } } )
```

Застосування шаблонів проектування

Після зіставлення взаємозв’язків даних програми наступним кроком у процесі проектування схеми застосування шаблонів проектування для оптимізації схеми. Шаблони проектування схем – це методи оптимізації моделі даних з урахуванням особливостей доступу докладання. Вони підвищують продуктивність програми та знижують складність схеми. Шаблони проектування схем впливають на те, як зберігаються дані та які дані повертаються в додаток.

Перед тим, як впроваджувати шаблони проектування схем, необхідно оцінити проблему, яку потрібно вирішити. Кожен шаблон проектування схем має свої варіанти використання та компроміси щодо узгодженості даних, продуктивності та складності. Наприклад, деякі шаблони проектування схем підвищують продуктивність запису, інші – читання.

Реалізація шаблону без розуміння програми та необхідних йому даних може призвести до зниження продуктивності програми та викликати непотрібні ускладнення у дизайні схеми.

Шаблони проектування схем

Шаблони проектування схем використовуються для оптимізації моделі даних на основі того, як програма запитує і використовує дані. Наприклад:

- виконувати розрахунки в базі даних, щоб результати були готові до моменту запиту даних клієнтом;
- групувати дані в серії для підвищення продуктивності та обліку викидів;
- обробляти змінні поля документа та типи даних в одній колекції;
- підготувати до зміни схеми з урахуванням змінних технічних вимог;
- перемістити старі дані в окреме місце, щоб збільшити обсяг сховища та підвищити продуктивність там, де дані використовуються найчастіше.

Розглянемо кілька варіантів найпоширеніших шаблонів.

Обчислювані значення

Якщо потрібно повертати до програми обчислені значення даних, можна підвищити продуктивність, виконуючи обчислення в базі даних, а не за запитом даних. Застосунку можуть бути потрібні як точні обчислення, так і приблизні результати. Використовуючи шаблони схем ***обчислення*** та ***апроксимації,*** можна заздалегідь обчислити та зберегти отримані значення (наприклад, при вставці або за допомогою періодичного завдання), щоб вони були легко доступні при запиті даних. Наприклад (таблиця 23):

Таблиця 23 – Сценарії для шаблонів обчислення та апроксимації

Сценарій	Застосування шаблону проектування
Інформаційна система університету збирає події оцінювання студентів (кожна оцінка фіксується окремо). На основі цих подій періодично формуються зведені показники, такі як «середній бал студента» та «загальна кількість оцінок». Базові дані (події оцінювання) продовжують накопичуватись незалежно від формування зведеної інформації.	**Шаблон обчислення** (Computed Pattern) використовується для періодичного побудови та оновлення вітрини даних (student_stats) на основі колекції подій (rating_events). Сирі дані зберігаються окремо, а агреговані показники перераховуються з використанням агрегування та операції $merge.
Інформаційна система університету відстежує кількість активностей кафедри за навчальний рік (конференції, проекти, заходи). Значення показника може змінюватися часто, однак висока точність у кожний момент часу не потрібна – достатньо наближеного значення для статистичного моніторингу.	**Шаблон апроксимації** (Approximation Pattern) використовується для оновлення лічильника активності кафедри із заданим кроком округлення (наприклад, 10). Точне значення обчислюється за даними колекції activities, але нове значення записується в dept_counters тільки при переході в інший квант округлення. Обчислення є абсолютно точними, але статистично достовірні і зменшують частоту записів до бази даних.

Застосунку може знадобитися отримати значення, підраховане з даних, що зберігаються в базі даних. Обчислення нового значення може вимагати значних ресурсів процесора, особливо у разі великих наборів даних чи випадках, коли необхідно перевірити кілька документів.

Якщо обчислюване значення запитується часто, можливо ефективніше зберегти їх у базі даних заздалегідь. При запиті даних програмою потрібна лише одна операція читання.

Якщо операції читання відбуваються значно частіше, ніж записи, шаблон, що обчислюється, знижує частоту обчислень даних. Замість обчислення значень при кожному читанні програма зберігає обчислене значення та перераховує його за необхідності. Програма може перераховувати значення при кожному записі, що змінює вихідні дані, або в рамках періодичного завдання.

⚠ *При періодичних оновленнях обчислене значення, що повертається, не гарантує точності. Однак такий підхід може бути виправданим з точки зору підвищення продуктивності, якщо висока точність не є обов'язковою вимогою.*

Задача: створити колекцію «подій оцінювання» (кожна виставлена оцінка – окрема подія) і показати, як на їх основі отримувати статистику.

На 1-му етапі додаємо **зразок даних**:

```
// «Сирі події» - оцінки як окрема колекція подій (якщо потрібно)
db.rating_events.insertMany([
 { studentKod: 1, discipline: "Технології проектування БД", mark: 82,
date: ISODate("2011-10-10T00:00:00Z") },
 { studentKod: 1, discipline: "Мережеві технології", mark: 10, date:
ISODate("2011-10-12T00:00:00Z") }
])
```

Потім необхідно додати **обчислені дані**.

Наприклад: зробити вітрину student_stats (avgMark, marksCount, updatedAt) та оновлювати її за розкладом через aggregate + $merge .

Щоб уникнути виконання цього обчислення щоразу при запиті інформації, можна обчислити загальні значення та зберегти їх у колекції статистики разом із самим записом студента:

```
// "Вітрина" - агрегат по студенту (передрахування)
db.student_stats.insertOne({
 studentKod: 1,
 avgMark: 46.0,
 marksCount: 2,
 updatedAt: ISODate("2011-10-12T00:00:00Z")
})
```

І, нарешті, виконуємо **оновлення розрахункових даних.**

Для цього до колекції оцінок додаємо нову оцінку:

```
// Нова подія оцінки
db.rating_events.insertOne({
 studentKod: 1,
 discipline: "Дослідження операцій",
 mark: 75,
 date: ISODate("2011-10-11T00:00:00Z")
})
```

Розрахункові дані в основній колекції не відображають поточні дані про перегляд. Частота оновлення розрахункових даних залежить від програми:

– у середовищі з низьким обсягом запису обчислення можуть виконуватися одночасно з будь-яким оновленням даних оцінок.
– серед більш регулярних записів обчислення можна виконувати з заданими інтервалами (наприклад, кожну годину). Вихідні дані в оцінках не зачіпаються записом до колекції студентів, тому обчислення можна запускати у будь-який час.

Для оновлення обчислених даних на основі даних оцінок можна запустити таку агрегацію з регулярним інтервалом:

```
// Перерахунок вітрини та merge у student_stats
db.rating_events.aggregate([
 {
 $group: {
  _id: "$studentKod",
  avgMark: { $avg: "$mark" },
  marksCount: { $sum: 1 },
  updatedAt: { $max: "$date" }
          }
 },
 {
 $merge: {
  into: { db: "lectures", coll: "student_stats" },
  on: "_id",
  whenMatched: "replace",
  whenNotMatched: "insert"
           }
 }
] )
```

Щоб перевірити, чи оновлено колекцію статистики, можна виконати запит до колекції:

```
db.student_stats.find()
```

Шаблон **апроксимації** використовується, коли значення часто змінюються, але користувачам не потрібно знати точні значення. Замість того, щоб оновлювати значення при кожній зміні даних, шаблон апроксимації оновлює дані з більшим ступенем деталізації, що призводить до зменшення кількості оновлень та зниження навантаження на програму. Зокрема, при аналізі населення міста, відвідувань веб-сайту, потоку авіапасажирів.

Ці виміри зазвичай корисні, якщо вони наближені. Програма може заощадити час та ресурси, оновлюючи збережені значення на сотні або тисячі, залежно від масштабу даних.

Задача: створити наближений лічильник активності кафедри (наприклад, кількість активностей/заходів за рік) з кроком округлення.

Користувачам застосунка насамперед цікаві загальні тенденції, і їм не потрібно знати точну кількість активностей.

Вставляємо зразок даних.

```
// Приклад «знімку» наближеного лічильника активностей кафедри за
2025 рік
db.dept_counters.insertOne({
 deptKod: 2,
 year: 2025,
 approxActivities: 120, // округлено до кроку 10
 roundingStep: 10,
 computedAt: ISODate("2025-12-31T00:00:00Z")
})
```

Реалізуємо шаблон апроксимації.

Логіка програми наступна: новий запис вставляється в dept_counters лише тоді, коли зміна (округлене значення) перевищує заданий поріг.

Наприклад:

```
/*
 Перерахунок наближеного лічильника активностей кафедри за рік.
 Пишемо новий запис в dept_counters тільки якщо "наближене" значення
змінилося
 мінімум на roundingStep (тобто перейшли в інший "квант"). */

function updateApproxActivitiesCounter(deptKod, year, roundingStep = 10) {

 // 1) Точне значення: скільки активностей у викладачів кафедри за рік
 // Припускаємо: activities містить lecturerId, a lecturers містить dept.deptKod
 const exact = db.activities.aggregate([
 { $match: { year: year } },
 {
 $lookup: {
 from: "lecturers",
```

```
 localField: "lecturerId",
 foreignField: "_id",
 as: "lecturer"
  }
  },
  { $unwind: "$lecturer" },
  { $match: { "lecturers.dept.deptKod": deptKod } },
  { $count: "cnt" }
  ]).toArray()[0]?.cnt ?? 0;

  // 2) Округлення "квантуванням"
  const approx = Math.round(exact / roundingStep) * roundingStep;

  // 3) Беремо останнє збережене наближене значення
  const last = db.dept_counters.findOne(
  { deptKod, year },
  { sort: { computedAt: -1 }, projection: { approxActivities: 1 } }
  );

  const lastApprox = last?.approxActivities;

  // 4) Записуємо тільки якщо змінилося >= roundingStep (або запису ще не
було)
  if (lastApprox === undefined || Math.abs(approx - lastApprox) >= roundingStep)
{
  db.dept_counters.insertOne({
  deptKod,
  year,
  approxActivities: approx,
  roundingStep,
  exactActivities: exact, // можна прибрати, якщо потрібно зберігати лише
наближене
  computedAt: new Date()
  });
  }

  return { deptKod, year, exact, approx, lastApprox };
}

// приклад виклику
updateApproxActivitiesCounter(2, 2025, 10);
```

Групові дані

Якщо схема містить великий ряд даних, угруповання цих даних у декілька менших рядів може підвищити продуктивність.

Схема також може потребувати обробки викидів у ряді, які знижують продуктивність для більш поширених значень даних. Для підвищення продуктивності та організації груп даних можна використовувати шаблони ***контейнерів*** та ***викидів*** (таблиця 24)

Таблиця 24 – Сценарії для шаблонів контейнерів та викидів

Сценарій	Застосування шаблону проектування
У базі даних зберігається великий масив листів/повідомлень студентам, а застосунок організує зберігання листів сторінками по 10 штук у документі-контейнері.	**Шаблон «контейнер»** використовується для групування листів та керування пагінацією на сервері. Такий підхід знижує навантаження на додаток та спрощує логіку пагінації.
У базі даних зберігаються публікації викладачів. У черговому році один викладач має аномально велику кількість публікацій.	**Шаблон викидів** використовується виділення публікацій у викладачів-лідерів в окремі документи. При такому підході не буде одного великого документа, який заважатиме вилученню даних із дрібніших, типових документів.

Шаблон фрагментації/сегментування типу «Контейнер» поділяє довгі ряди даних на окремі об’єкти. Поділ великих рядів даних на дрібніші групи може покращити шаблони доступу до запитів та спростити логіку програми. Групування корисне, коли є схожі об’єкти, які стосуються центральної сутності, наприклад, угоди з акціями, вчинені одним користувачем.

Шаблон «контейнер» використовується для пагінації (розбиття на фрагменти/сторінки, аналогічно конструкції фрагментації у віконних функціях SQL), групуючи дані на основі елементів, що відображаються застосунком на кожній сторінці. Цей підхід використовує гнучку модель даних MongoDB для зберігання даних відповідно до потреб програми.

Наприклад, розглянемо контейнер-схему для листів студента та запити отримання 1-ї та N-ї сторінки (limit/skip або курсорна пагінація по _id). У початковій схемі не використовується шаблон «контейнер» і кожен лист зберігається в окремому документі.

```
db.letters.insertMany([
 { studentKod: 1, content: "Зателефонуйте мені...", date: ISODate("2011-10-09T00:00:00Z") },
 { studentKod: 2, content: "У задачі №4 відповідь 12.24", date: ISODate("2011-10-10T00:00:00Z") }
])
```

Застосунок відображає листи по 10 штук на сторінці. Щоб спростити логіку програми, можна використати шаблон контейнера для групування листів за ідентифікатором студента у групи по 10.

Для цього виконуємо таке.

Спершу **групуємо дані по studentKod.**

Реорганізуємо схему так, щоб для кожного studentkod був окремий документ:

```
db.letters_container.insertMany([
 {
 studentKod: 1,
 count: 1,
 history: [
 { content: "Зателефонуйте мені...", date: ISODate("2025-10-
09T00:00:00Z") }
 ],
 updatedAt: ISODate("2025-10-09T00:00:00Z")
 },
 {
 studentKod: 2,
 count: 1,
 history: [
 { content: "У задачі №4 відповідь 12.24", date: ISODate("2025-10-
10T00:00:00Z") }
 ],
 updatedAt: ISODate("2025-10-10T00:00:00Z")
 }
])
```

Із шаблоном контейнера:

- документи із загальним значенням studentKod об’єднуються в один документ, при цьому studentKod є полем верхнього рівня.
- листи для цього студента групуються у вбудованому полі масиву, що зветься *історією* .

Потім **додаємо ідентифікатор та підраховуємо кількість для кожного контейнера.**

```
db.letters_pages.insertMany([
 {
 _id: "student:1:page:1",    // Більш коректно { studentKod: 1, page: 1 }
 studentKod: 1,
 page: 1,
 pageSize: 10,
 count: 1,
 history: [
 { content: "Зателефонуйте мені...", date: ISODate("2025-10-
09T00:00:00Z") }
```

```
 ],
 updatedAt: ISODate("2025-10-09T00:00:00Z")
 },
 {
 _id: "student:2:page:1",
 studentKod: 2,
 page: 1,
 pageSize: 10,
 count: 1,
 history: [
 { content: "У задачі №4 відповідь 12.24", date: ISODate("2025-10-
10T00:00:00Z") }
 ],
 updatedAt: ISODate("2025-10-10T00:00:00Z")
 }
])
```

Значення поля _id являє собою об’єднання studentKod і першої сторінки листів . Коректніше використовувати не рядкове значення ключа "student: 1:page :1", а складовий ключ { studentKod: 1, page: 1 }.

Поле «count» вказує кількість елементів у масиві історії документа та використовується для реалізації логіки пагінації.

У оновленій схемі кожен документ містить дані для однієї сторінки програми. Поля _id та count можна використовувати, щоб визначити, як повертати та оновлювати дані.

Щоб запросити дані на потрібній сторінці, використовується запит з регулярним виразом для повернення даних за вказаним studentKod і функцію skip для повернення до даних потрібної сторінки. Запит з регулярним виразом _id використовує індекс _id за замовчуванням, що забезпечує високу продуктивність запитів без необхідності використання додаткового індексу.

Наступний запит повертає дані для першої «сторінки» листів студента 2:

```
db.letters_pages.find({ _id: /^2_/ }).sort({ _id: 1 }).limit(1)
```

Щоб отримати дані для наступних сторінок, вказується значення пропуску (skip) на одиницю менше, ніж сторінка, на яку потрібно показати дані. Наприклад, щоб показати дані для сторінки 3:

```
db.letters_pages.find({ _id: /^2_/ }).sort({ _id: 1 }).skip(2).limit(1)
```

Тепер, коли схема використовує шаблон контейнера, потрібно **оновити логіку програми, щоб додавати нові листи до потрібного контейнера**. Для цього використовується команда оновлення, наприклад, для додавання листа до контейнера з відповідним `studentKod` і контейнером.

Наприклад, вставка листа в сторінку, де count < 10, інакше upsert нової сторінки:

```
db.letters_pages.updateOne(
 { _id: /^2_/, count: { $lt: 10 } },
 {
 $push: { history: { content: "Новий текст...", date: ISODate("2011-10-
11T00:00:00Z") } },
 $inc: { count: 1 },
 $setOnInsert: { _id: "2_20111011", studentKod: 2 }
 },
 { upsert: true }
)
```

Якщо є документ, відповідний «`_id`»: `/^2_/`, і його кількість менше 10, команда оновлення вміщує нову угоду в масив історії (history) відповідного документа.

Якщо відповідного документа немає, команда update вставляє новий документ з новим листом (бо `upsert` має значення `true`).

Групування даних із шаблоном викидів.

Якщо в колекції зберігаються документи в цілому однакового розміру і форми, то документ (викид), що істотно відрізняється, може викликати проблеми з продуктивністю для поширених запитів.

Розглянемо колекцію, яка зберігає поле масиву. Якщо документ містить набагато більше елементів масиву, ніж інші документи в колекції, може знадобитися обробляти цей документ у схемі інакше.

Шаблон викидів використовується для того, щоб ізолювати документи, що не відповідають очікуваній формі, від решти колекції.

Перш ніж змінювати схему для обробки викидів, розглянемо плюси та мінуси шаблону викидів:

Плюси

- шаблон викидів підвищує продуктивність запитів, що часто виконуються: запити, що повертають типові документи, не обов'язково повинні повертати документи з великими викидами.
- шаблон викидів також обробляє прикордонні випадки у застосунку: наприклад, якщо програма зазвичай відображає 50 результатів з масиву, документ з 2000 результатами не заважатиме користувачеві.

Мінуси

Шаблон викидів вимагає складнішої логіки для обробки оновлень. Якщо часто доводиться оновлювати дані, можливо, варто розглянути інші шаблони проектування схем.

Як ***приклад,*** розглянемо схему відстеження публікацій викладачів. Для більшості викладачів кількість публікацій за рік помірна (типовий випадок). Такий документ зручний, бо його можна швидко читати повністю, оновлювати окремі елементи масиву, агрегувати за роками.

Типові документи в колекції викладача, де публікації за рік зберігаються у вбудованому масиві publications, виглядають так:

```
db.lecturers.insertOne({
 kod: 1,
 secondName: "Малахов",
 firstName: "Євгеній",
 patronymic: "Валерійович",
 dept: { deptKod: 2, deptName: "МЗКС" },

 publications: [
 {
 year: 2025,
 type: "article",
 title: "NoSQL-моделі в навчальних проектах",
 venue: "Вісник університету",
 coauthors: ["Петрушина Т.І."],
 pages: "12–18"
 },
 {
 year: 2025,
 type: "conference",
 title: "Практика MongoDB в курсі БД",
 venue: "Proc. of IT Education Conf",
 coauthors: [],
 pages: "55–60"
 } ],

 pubCountByYear: { "2025": 2 },
 updatedAt: ISODate("2025-12-31T00:00:00Z")
})
```

Масив publications *не обмежений,* тобто у міру публікацій викладача масив зростає. Для більшості документів це не проблема, оскільки система не очікує великого кількості публікацій.

Припустимо, що в черговому році якийсь викладач має аномально велику кількість публікацій.

Поточна схема призводить до роздуття документа, що негативно позначається на продуктивності. Щоб вирішити цю проблему, реалізується шаблон викидів для документів з нетиповою кількістю публікацій.

Спочатку **визначимо граничне значення** для викидів, тобто, враховуючи типову структуру документа схеми, визначимо, коли документ стає викидом. Порогове значення може залежати від вимог інтерфейсу користувача програми або запитів, які до своїх документів виконують користувачі.

У цьому прикладі поріг викиду publications (year = 2025) > 50.

Далі необхідно вирішити, **як чинити з викидами.**

При роботі з великими масивами поширеним способом обробки викидів є зберігання значень, що перевищують граничне значення, в окремій колекції.

Для більш ніж 50 публікацій збережемо додаткові значення publications в окремій колекції.

На наступному кроці **додамо індикатор документів, що випадають із загального списку.**

Наприклад, для викладачів, у яких понад 50 пудлікацій, додамо нове поле документа has_publications_extras і встановимо значення true. Це поле вказує на те, що окрема колекція зберігає додаткові публікації.

Отже, в документі викладача зберігатимемо лише «голову» (наприклад, перші 50 публікацій) та агрегати (pubCountByYear, has_publications_extras). Інші публікації будемо зберігати в колекції publications_extras (сторінками), щоб не створювати один занадто великий документ.

```
// Викид: у викладача аномально багато публікацій за 2025 рік
// У документі lecturers залишаємо лише «голову» масиву + агрегати т
а прапор
db.lecturers.insertOne({
 kod: 1,
 secondName: "Малахов",
 firstName: "Євгеній",
 patronymic: "Валерійович",
 dept: { deptKod: 2, deptName: "МЗКС" },

 // «Голова» – перші 50 публікацій
 publications: [
 { year: 2025, type: "article", title: "Paper 1", venue: "Journal A" },
 { year: 2025, type: "article", title: "Paper 2", venue: "Journal A" }
 // ... далі інші 48 публікацій
 ],

 pubCountByYear: { "2025": 356 },
 has_publications_extras: true,
 publicationsExtras: {
 year: 2025,
 pageSize: 100,
 pages: 4   // 356 публікацій => 4 сторінки по 100 (остання неповна)
 },

 updatedAt: ISODate("2025-12-31T00:00:00Z")
})
```

І нарешті, **збережемо викиди в окремій колекції .**

Для ***прикладу,*** створимо колекцію з назвою publications_extras для зберігання публікацій понад початкові 50 та зв'яжемо документи з колекції publications_extras з колекцією викладачів за допомогою посилання:

```
// Хвіст публікацій викладача зберігається окремо (сторінки по 100)
// Прив'язка по lecturerKod (можна замінити на lecturerId:ObjectId, якщ
о так у схемі)
db.publications_extras.insertMany([
 {
 lecturerKod: 1,
 year: 2025,
 page: 1,
 pageSize: 100,
 publications_extra: [
 { year: 2025, type: "article", title: "Paper 51", venue: "Journal B" }
 // ... до Paper 150
 ] },
 {
 lecturerKod: 11,
 year: 2025,
 page: 2,
 pageSize: 100,
 publications_extra: [
 { year: 2025, type: "conference", title: "Paper 151", venue: "Conf X" }
 // ... до Paper 250
 ] }
 // ... page 3, page 4
])
```

Шаблон викидів запобігає впливу нетипових документів на продуктивність запитів. Отримана схема дозволяє уникнути появи великих документів колекції, зберігаючи при цьому повний список документів.

Оновлення для документів викидів необхідно обробляти інакше, ніж для типових документів. Щоб виконати оновлення викидів для попередньої схеми, реалізується наступна логіка програми:

1) перевірити, чи для оновлюваного документа встановлено значення has_publications_extras рівним true;
2) якщо has_publications_extras відсутня або false, додати нові публикації до колекції lecturers;
3) якщо отриманий масив publications містить більше 50 елементів, встановити has_publications_extras значення true;
4) якщо has_publications_extras є true, додати нові публикації в колекцію publications_extras для відповідного lecturerKod.

Групування даних за допомогою шаблону атрибутів (Attribute)

Шаблон атрибутів – це шаблон проектування схеми, який допомагає організувати документи з великою кількістю подібних полів, особливо якщо ці поля мають загальні характеристики. Якщо потрібно сортувати або виконувати запити щодо цих підмножин схожих полів, шаблон атрибутів може оптимізувати схему. Він спрощує індексацію документів, поєднуючи кілька подібних полів документа в піддокумент типу «ключ-значення». Замість створення кількох індексів за кількома схожими полями шаблон атрибутів дозволяє створювати менше індексів, прискорюючи та спрощуючи написання запитів.

Шаблон атрибуту використовується, якщо до колекції можна застосувати хоча б одну з таких умов:

1) існують великі документи, що містять багато схожих полів із загальними характеристиками, які потрібно сортувати чи вимагати;
2) невелика підгрупа документів містить поля, необхідні сортування.

Наприклад, у БД зберігаються активності викладачів (конференції, проекти, комітети, курси, публікаційна активність тощо). Для різних типів активностей набір параметрів різниться: у конференції є venue/city/isInternational, проект – grant/customer/budget, курс – hours/semester тощо. Типові документи в колекції можуть виглядати так:

```
// activities: «пласка» схема, яка швидко розростається
db.activities.insertOne({
  lecturerKod: 1,
  year: 2025,
  type: "conference",
  name: "IT Education Conf",

  conf_city: "Одеса",
  conf_venue: "ONU",
  conf_isInternational: true,

  proj_customer: null,
  proj_budget: null,

  course_semester: null,
  course_hours: null
})
```

Є кілька полів виду conf_city, proj_budget, course_hours, …, «окреме» зберігання яких призводить до розростання схеми та індексів. Якщо потрібно знайти місто, бюджет тощо, доведеться переглядати відразу кілька полів. Без шаблону атрибутів довелося б створити кілька індексів для колекції активностей, щоб швидко виконувати пошук за ними:

```
db.activities.createIndex({ conf_city: 1 })
```

```
db.activities.createIndex({ conf_isInternational: 1 })
db.activities.createIndex({ proj_budget: 1 })
db.activities.createIndex({ course_hours: 1 })
```

Однак індекси коштують дорого і можуть знижувати продуктивність, особливо при операціях запису.

Наступна процедура демонструє, як можна застосувати шаблон атрибутів до колекції активностей, перемістивши підмножину інформації до масиву, що знижує потребу в індексації.

І тому спочатку **групуємо підмножини даних у масив,** тобто. реорганізуємо схему так, щоб перетворити різні поля властивостей активностей на масив пар ключ-значення:

```
db.activities.insertOne({
 lecturerKod: 1,
 year: 2025,
 type: "conference",
 name: "IT Education Conf",

 attrs: [
 { k: "city",   v: "Одеса" },
 { k: "venue",  v: "ONU" },
 { k: "isInternational", v: true }
 ]
})
```

А потім **створюємо один індекс по масиву** attrs. Цей індекс оптимізує запити щодо будь-якого з полів властивостей активностей.

```
db.activities.createIndex({ "attrs.k": 1, "attrs.v": 1 })
```

Якщо документ містить кілька полів, які відстежують однакові або схожі характеристики, шаблон атрибута дозволяє уникнути необхідності створення індексів кожного схожого поля. Поєднуючи схожі поля в масив і створюючи індекс для цього масиву, можна скоротити загальну кількість необхідних індексів та підвищити продуктивність запитів.

Запити, які використовують такий індекс, можуть виглядати так:

```
// Обрати всі активності в Одесі
db.activities.find({ attrs: { $elemMatch: { k: "city", v: "Одеса" } } })

// Знайти всі міжнародні активності
db.activities.find({ attrs: { $elemMatch: { k: "isInternational", v: true } } })
```

Шаблон атрибутів може бути корисним, коли в документі описуються характеристики товарів. Деякі товари можуть мати розміри, виражені як «маленький», «середній» або «великий». Інші товари з тієї ж колекції можуть бути виражені обсягом, а треті – фізичними розмірами чи вагою.

Наприклад, розглянемо колекцію пляшок із водою. Документ без використання шаблону атрибуту може виглядати так:

```
db.bottles.insertOne([
 {
 "_id": 1,
 "volume_ml": 500,
 "volume_ounces": 12,
 "height_cm": 30
 }
])
```

Оскільки поля volume_ml, volume_ounces і height_ cm у першому документі містять схожу інформацію, їх можна об’єднати в одне поле – specs, у якому перебуватиме інформація про технічні характеристики конкретної пляшки з водою, де поле *k* показує, що саме вимірюється, *v* – значення, а *u* – одиницю виміру.

Застосуємо шаблон атрибута до колекції пляшок:

```
db.bottles.insertOne([
 {
 "_id": 1,
 specs: [
 { k: "volume", v: "500", u: "ml" },
 { k: "volume", v: "12", u: "ounces" },
 { k: "height", v: "8", u: "cm" }
 ]
 }
])
```

З прикладу видно, що шаблон атрибутів дозволяє групувати схожі поля різними іменами. Вказуючи атрибути через пари «ключ-значення», наприклад поле k, яке визначає вимірюваний параметр, можете зберігати ширший спектр схожих полів в одному масиві, мінімізуючи кількість індексів, необхідних для ефективного запиту даних.

Використання пар «ключ-значення», таких як *k*, *v* і *u* забезпечує велику гнучкість при додаванні полів в масив. Чим більше полів можна об’єднати в масив, тим менше індексів потрібно створювати, що підвищує продуктивність запитів.

Антишаблони (антипатерни) проектування схем

Антипатерни проектування схем (таблиця 25) – неефективні методи структурування схеми бази даних. Вони можуть створювати непотрібну складність та викликати проблеми з продуктивністю. Розпізнавання та запобігання антипатернам проектування схем може допомогти створювати більш продуктивні програми.

Таблиця 25 – Антишаблони проектування схем

Антишаблон	Опис
Усунення необмежених масивів	У документі зберігається необмежений масив, який може стати надто великим. Такий великий масив може перевищити обмеження на розмір документа та призвести до зниження продуктивності індексу.
Зменшення кількості колекцій	Створюється велика кількість колекцій у базі даних. Занадто велика кількість колекцій може знизити продуктивність сховища.
Видалення непотрібних індексів	Колекція містить непотрібні індекси. Такі індекси займають додатковий дисковий простір та можуть знизити продуктивність запису.
Зменшення кількості роздутих документів	У колекції надто багато документів. Великий обсяг документів може знизити продуктивність найчастіших запитів.
Зменшення кількості операцій $lookup	Виконується надто багато операцій $lookup над даними. Це збільшує складність запитів та знижує їхню продуктивність.

Питання для самоперевірки

(до розділів «Моделювання даних», «Розробка схеми», «Шаблони проектування схем»)

1. У чому полягають основні особливості моделювання даних у MongoDB?
2. Чому при проектуванні документної БД важливо враховувати характер типових запитів до даних?
3. У чому полягає відмінність між моделюванням даних у реляційних та документних СУБД?
4. Що означає поняття «межі агрегату» у документній моделі даних і які фактори впливають на вибір меж документа у MongoDB?
5. У яких випадках доцільно використовувати вкладені документи, а в яких – посилання між документами?
6. Які переваги та недоліки денормалізації даних у MongoDB?
7. Які чинники можуть вплинути на планування моделі даних?
8. Чому при проектуванні MongoDB-систем важливо враховувати частоту оновлення даних?
9. У чому полягає взаємозв’язок між моделюванням даних та продуктивністю MongoDB-системи?
10. Що таке шаблони проектування схем у MongoDB?
11. Для чого використовуються шаблони проектування схем?
12. У чому полягає ідея шаблону Extended Reference?
13. У яких випадках доцільно використовувати шаблон Computed Pattern і чим від нього відрізняється шаблон апроксимації?
14. Які переваги та недоліки шаблону Bucket Pattern?
15. Для чого використовується шаблон Schema Versioning Pattern?
16. Що таке антипатерни проектування схем?
17. Чому одна й та сама предметна область може мати кілька коректних варіантів документної моделі?

Завдання для самостійної роботи

Для ПрО, обраної в розділі «Завдання для самостійної роботи» (с. 31) розробити схему БД:

1) визначити робоче навантаження;
2) уточнити взаємозв'язкі даних предметної області та визначити, чи є ефективним вибір зв'язування чи вбудовування пов'язані дані, і за необхідності переробити колекції БД;
3) визначити найбільш доцільні шаблони проектування для оптимізації читання та запису.

Демонстраційний приклад

Завдання

Для предметної області ***«Портфоліо студентських проєктів»*** з розділу «Демонстраційний приклад» (с.34) розробити схему БД:

1) визначити робоче навантаження;
2) уточнити взаємозв'язкі даних предметної області та визначити, чи є ефективним вибір зв'язування чи вбудовування пов'язані дані, і за необхідності переробити колекції БД;
3) визначити найбільш доцільні шаблони проектування для оптимізації читання та запису.

Розв'язання

1. Визначення робочого навантаження

№	Тип операції	Приклад задачі	Дані, що використовуються	Характер навантаження	Висновок для схеми
1	Пошук проєктів за технологією	ДД2.1: знайти проєкти, що використовують задану технологію або мову програмування	projects.technologies, projects.updated_at	Часті операції читання з фільтрацією та сортуванням	Технології доцільно зберігати масивом у документі проєкту
2	Повнотекстовий пошук	ДД2.2: пошук за ключовими словами опису або назви	projects.title, projects.description	Пошук за текстом із ранжуванням результатів	Назву та опис потрібно залишити у колекції projects; для них створюється текстовий індекс

№	Тип операції	Приклад задачі	Дані, що використовуються	Характер навантаження	Висновок для схеми
3	Аналіз етапів виконання	ДД2.3: розгорнути масив етапів і визначити поточний стан реалізації	projects.stages	Читання і агрегування даних у межах одного проєкту	Етапи доцільно вбудовувати в документ проєкту як масив вкладених документів
4	Оновлення проєкту	ДД2.4: додати етап, файл або відгук керівника	projects.stages, projects.files, projects. supervisor_reviews	Помірні операції запису з додаванням елементів у масиви	Пов’язані з проєктом елементи доцільно зберігати у самому документі проєкту
5	Аналітичне групування	ДД2.5: визначити найчастіше використані технології за групою або спеціальністю	projects.authors, students.group, students.speciality, projects.technologies	Агрегування з використанням зв’язку між projects і students	Дані студентів доцільно залишити в окремій колекції, а в проєкті зберігати посилання student_id

Отже, основне навантаження припадає на колекцію projects, оскільки саме документ проєкту є основним агрегатом даних. Більшість операцій читання та оновлення виконується навколо одного проєкту, його авторів, технологій, етапів, файлів і відгуків.

2. Уточнення взаємозв’язків даних та вибір між вбудовуванням і зв’язуванням

У початковій схемі були визначені три колекції: students, projects, project_tags. Після аналізу робочого навантаження цю структуру можна залишити, але уточнити спосіб зберігання пов’язаних даних.

2.1. Доцільність вбудовування

У документ projects доцільно вбудовувати:

- authors – масив учасників проєкту з посиланням на student_id та роллю в проєкті;
- technologies – масив використаних технологій;
- tags – масив тематичних тегів;
- stages – масив етапів виконання проєкту;
- files – масив посилань на презентації, демонстраційні матеріали або інші файли;

– supervisor_reviews – масив відгуків керівників.

Таке вбудовування є ефективним, оскільки ці дані зазвичай переглядаються разом із карткою проєкту і логічно належать саме до нього.

2.2. Доцільність зв'язування

Колекцію students доцільно залишити окремою, оскільки студент може брати участь у кількох проєктах, а його дані не слід дублювати в кожному документі проєкту. У документі projects достатньо зберігати student_id та роль студента в конкретному проєкті.

Колекцію project_tags також доцільно залишити окремою як довідникову, оскільки вона використовується для уніфікації тегів, тематичної навігації та фільтрації проєктів.

Отже, на рис. 15 наведено діаграму бази даних.

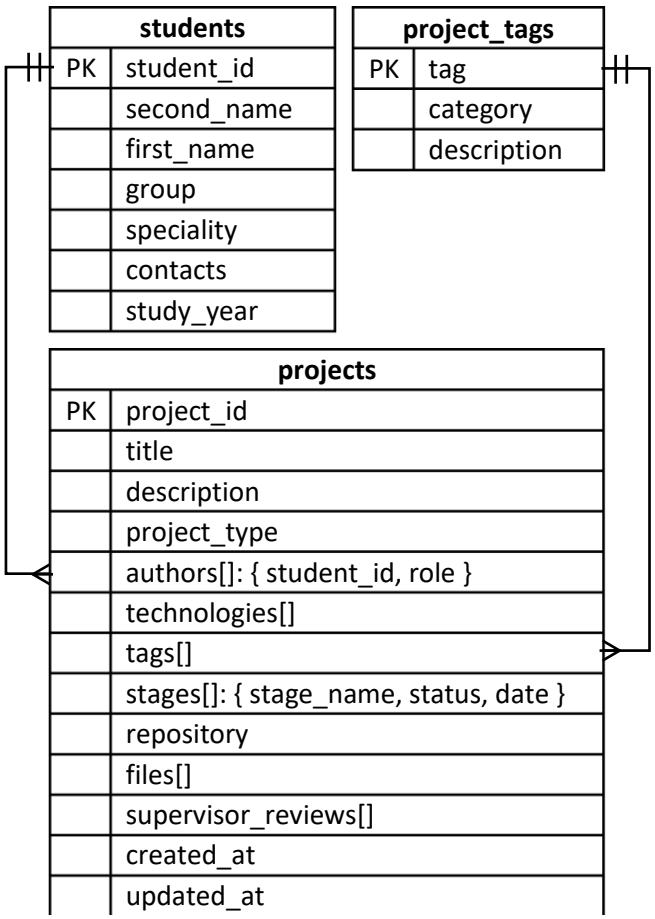


Рис. 15 – Діаграма документної БД «Портфоліо студентських проєктів»

3. Доцільні шаблони проектування

3.1. Шаблон вбудованих документів (Embedded Document Pattern)

Основним шаблоном для колекції projects є вбудовування пов'язаних даних у документ проєкту:

```
{
  project_id: 101,
  title: "Система управління студентськими проєктами",
```

```
 authors: [
  { student_id: 1, role: "team-lead" },
  { student_id: 2, role: "backend-developer" }
 ],
 technologies: ["MongoDB", "Node.js", "React"],
 stages: [
  { stage_name: "Аналіз вимог", status: "completed", date:
ISODate("2026-02-10") },
  { stage_name: "Розробка API", status: "in-progress", date:
ISODate("2026-03-01") }
 ]
}
```

Цей шаблон зменшує потребу у додаткових запитах і дозволяє отримати основні дані про проєкт однією операцією читання.

3.2. Шаблон посилань (Reference Pattern)

Для зв'язку проєктів зі студентами використовується посилання через student_id:

```
authors: [
 { student_id: 1, role: "team-lead" },
 { student_id: 2, role: "backend-developer" }
]
```

Цей шаблон доцільний, оскільки студент є самостійною сутністю і може бути пов'язаний із кількома проєктами.

3.3. Шаблон атрибутів (Attribute Pattern)

Для технологій і тегів доцільно використовувати масиви атрибутів:

```
technologies: ["MongoDB", "Node.js", "React"],
tags: ["web"]
```

Це дозволяє гнучко додавати нові технології та тематики без зміни структури документа.

3.4. Шаблон підмножини (Subset Pattern)

Якщо кількість файлів або відгуків зростатиме, у документі projects можна залишати тільки найважливіші або останні елементи, а повну історію винести в окремі колекції, наприклад project_files або project_reviews.

Для поточного навчального прикладу таке винесення не є обов'язковим, але цей шаблон варто розглядати як можливий напрям подальшої оптимізації.

3.5. Шаблон обчислень (Computed Pattern)

Для прискорення аналітичних запитів можна додати до документа проєкту попередньо обчислювані поля, наприклад:

```
avg_rating: 96,
reviews_count: 2,
```

```
stages_count: 3,
last_stage_status: "in-progress"
```

Це може зменшити кількість обчислень під час частого формування списків проєктів, рейтингів або звітів.

Висновок

Для БД «Портфоліо студентських проєктів» базову структуру колекцій students, projects, project_tags доцільно зберегти. Основним агрегатом є документ проєкту, тому етапи виконання, технології, теги, файли та відгуки керівників доцільно зберігати як вкладені масиви у колекції projects. Дані студентів і довідник тегів доцільно залишити в окремих колекціях, використовуючи посилання та значення тегів у документах проєктів.

Найбільш доцільними шаблонами проектування є *Embedded Document Pattern*, *Reference Pattern*, *Attribute Pattern* та *Computed Pattern*. *Subset Pattern* може бути використаний пізніше, якщо обсяг вкладених файлів або відгуків суттєво зросте.

MongoDB Sharding

Останнє, що ми розглянемо у цій СУБД – це її здатність до горизонтального масштабування.

Sharding (шардинг/шардування) – це процес зберігання записів даних на декількох комп'ютерах і це підхід MongoDB для задоволення потреб зростання даних. У міру збільшення розміру даних однієї станція може бути недостатньо для зберігання даних, ні для забезпечення прийнятної для читання і запису пропускної здатності. Шардинг вирішує проблему із сегментуванням – додаванням машин для підтримки зростання обсягів даних та потреби операцій читання та запису.

На рис. 16 показано схему sharded-кластера MongoDB.

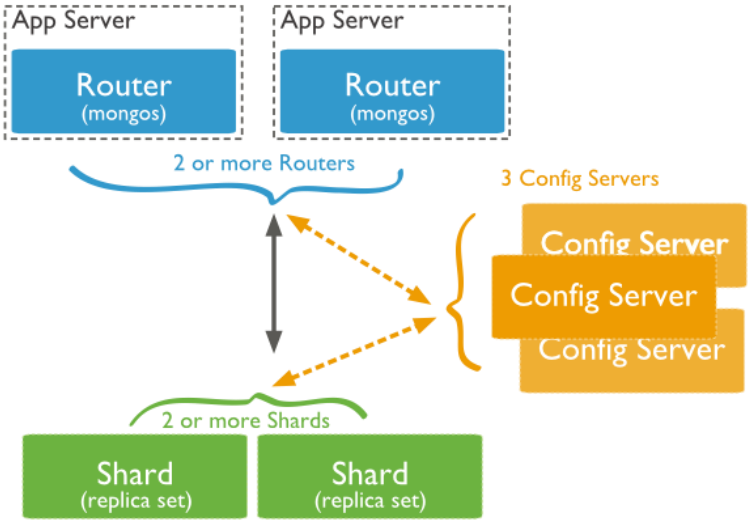


Рис. 16 – Розподілений кластер MongoDB

На цій схемі наведено три основні компоненти (таблиця 26).

Таблиця 26 – Компоненти sharded-кластера MongoDB

Компонент	Призначення	Опис
Shard (Replica Sets / уламки)	Використовуються для зберігання даних	Вони забезпечують високу доступність та цілісність даних. У виробничому середовищі кожен уламок є окремим набором реплік.
Config Servers	Сервери конфігурації для зберігання метаданих кластерів	Метадані містять відображення даних кластера, встановлених в уламках. Маршрутизатор запитів використовує ці метадані для цільових операцій із конкретними осколками. У виробничому середовищі розподілені кластери мають рівно 3 сервери конфігурації.
Query Routers	Маршрутизатори запитів (в основному *mongos)* – інтерфейс з клієнтськими застосунками та прямими операціями на відповідний уламок	Маршрутизатор запитів обробляє та розподіляє цільові операції уламків, а потім повертає результати для клієнтів. Sharded кластер може містити більше одного маршрутизатора запиту та розділяти навантаження запиту клієнта. Клієнт надсилає запити до одного маршрутизатора запиту. Зазвичай sharded кластер має багато маршрутизаторів запитів.

Вони вбудовані у стандартну інсталяцію, але працюють як різні процеси.

Коли встановлюється MongoDB (Community або Enterprise), ми отримуємо набір файлів, що виконуються (.exe у Windows або бінарники в Linux).

У каталозі встановленого MongoDB (зазвичай це C:\Program Files\MongoDB\Server\XX\bin) лежать два ключові файли, які роблять всю магію:

1) mongod.exe (Daemon) – такий собі «універсальний солдат», який запускається для зберігання даних і може бути
   - звичайним сервером БД;
   - шардом (якщо запущений із прапором --shardsvr);
   - конфіг-сервером (якщо запущено з прапором --configsvr).
2) mongos.exe (Sharding Router) – окрема легковажна програма, яка не вміє зберігати дані на диску (у нього навіть немає параметра dbPath). Його єдине завдання – знати, де лежать конфіг-сервери, і перенаправляти запити від вашого Python-скрипту до потрібних шардів.

Побудова розподіленої системи баз даних

Процес побудови розподіленої системи баз даних складається з наступних кроків.

1. Підготовка інфраструктури (конфігурація)

Спочатку запускається ***Config Server***. У сучасних версіях це обов'язково має бути Replica Set (навіть якщо там один сервер).

```
# Запуск Config Server
mongod --configsvr --replSet configReplSet --dbpath /data/configdb --port 27019
```

Потім запустимо звичайні екземпляри для **шардів**:

```
# Запуск першого шарда
mongod --shardsvr --replSet shard1RS --dbpath /data/shard1 --port 27018
```

2. Запуск роутера (*Mongos)*

Це «вхідні двері» до кластера. Він не зберігає дані, а лише перенаправляє запити.

```
# Вказуємо шлях до конфіг-серверів
mongos --configdb configReplSet/localhost:27019 --port 27017
```

3. Налаштування шардування в консолі

Тепер підключаємося до mongos через оболонку mongosh та реєструємо шарди.

```
// Підключаємося до mongos
// mongosh --port 27017

// 1. Додаємо шард до кластеру
sh.addShard("shard1RS/localhost:27018")
```

```
// 2. Включаємо шардування для конкретної бази даних
sh.enableSharding("testDB")

// 3. Вибираємо колекцію та ключ шардування (Shard Key)
// Важливо: ключ повинен бути індексований!
db.users.createIndex({ "country": 1, "userId": 1 })

// Розрізаємо колекцію по хешованому ключу (для рівномірного розподілу)
sh.shardCollection("testDB.users", { "country": 1, "userId": 1 })
```

Необхідно **звернути увагу** (такий собі чек-лист) на

1) **вибір ключа (Shard Key)**

   Це найкритичніше рішення. Якщо вибрати поганий ключ (наприклад, з низькою кардинальністю або ID, що монотонно зростає), отримаємо «гарячі шарди», де один сервер перевантажений, а решта відпочивають.

2) **вибір хешованого або діапазонного розподілу**

   * `{ "field": "hashed" }` – ідеально для рівномірного розподілу.
   `{ "field": 1 }` – краще для запитів за діапазонами (наприклад, за датами), але може створити навантаження на один шард.

3) **незворотність:**

   У багатьох версіях MongoDB змінити ключ шардування після того, як колекція вже «порізана», дуже складно чи неможливо без перестворення колекції.

Для **перевірки статусу шардів** можна використовувати команду:

```
sh.status()
```

Вона покаже, які шарди активні та як розподілені «чанки» (шматочки даних) між серверами.

Де і як запускати

У ***продакшені*** (реально експлуатованої системі) всі компоненти розносять так, щоб вихід з ладу одного сервера не «поклав» всю базу, тобто – це завжди різні фізичні або віртуальні машини (таблиця 27). У ***тестовому середовищі*** можна запустити все хоч на одному ноутбуці.

Таблиця 27 – Відмовостійка схема для Production

Компонент	Де запускається	Обґрунтування
Config Servers	Мінімум 3 окремі машини	Це «мозки» кластера: якщо вони «впадуть», mongos не знатиме, де лежать дані
Shard (Replica Sets)	Кожна копія (node) шарду – на окремому сервері	Щоб дані були доступні навіть якщо згорить стійка в дата-центрі
Mongos (Router)	**На серверах застосунків** (App Servers) або на окремих вузлах	Зазвичай mongos ставлять прямо там, де «крутиться» код (backend), щоб застосунок підключався до localhost:27017

Мінімальний кластер (економічна схема)

Якщо ресурсів мало, можна «підселяти» процеси один до одного, наприклад:

– **Сервер А:** Config Server №1 + Shard 1 (Primary) + Mongos
– **Сервер Б:** Config Server №2 + Shard 2 (Primary) + Mongos
– **Сервер В:** Config Server №3 + Вторинні копії шардів

Недоліки такої конфігурації: якщо Сервер А впаде, буде втрачено відразу і частину даних (поки не вибереться новий Primary), і один із трьох конфіг-серверів.

Роутеру `mongos` не потрібно багато пам'яті або диска (він не зберігає дані), тому його часто приклеюють «у навантаження» до інших сервісів. Але при цьому в мережі:

– `mongos` повинен бачити **всі** конфіг-сервери;
– `mongos` повинен бачити **всі** шарди;
– конфіг-сервери повинні бачити один одного (щоб підтримувати кворум).

⚠ *Ніколи не запускайте лише один Config Server у продакшені. MongoDB вимагає Replica Set з конфіг-серверів (зазвичай 3 вузли), інакше кластер працюватиме тільки в режимі читання або взагалі заблокується при збоях.*

Налаштування кластера

Для налаштування кластера потрібно *три різні типи конфігураційних файлів*. Основна відмінність – у секції `sharding` і `replication`.

Наприклад.

1. Конфігурація для Config Server (mongod-config.conf)

Ці сервери зберігають карту даних. Їм не потрібне багато місця, але важлива надійність.

```
storage:
  dbPath: /var/lib/mongodb-config
systemLog:
  destination: file
  path: /var/log/mongodb/configsvr.log
  logAppend: true
net:
  port: 27019
  bindIp: 0.0.0.0 # вказати конкретні IP
replication:
  replSetName: configReplSet
sharding:
  clusterRole: configsvr
```

2. Конфігурація для Shard Server (mongod-shard.conf)

Це робочі конячки, які зберігають ваші документи. Таких конфігів буде по одному на кожен примірник шарда.

```
storage:
  dbPath: /var/lib/mongodb-shard
systemLog:
  destination: file
  path: /var/log/mongodb/shardsvr.log
  logAppend: true
net:
  port: 27018
  bindIp: 0.0.0.0
replication:
  replSetName: shard1RS  # У кожного шарда своє им’я репліка-сета
sharding:
  clusterRole: shardsvr
```

3. Конфігурація для Mongos (Router) (mongos.conf)

У роутера немає секції storage, тому що він не зберігає дані на диску.

```
systemLog:
  destination: file
  path: /var/log/mongodb/mongos.log
  logAppend: true
net:
  port: 27017
  bindIp: 0.0.0.0
sharding:
  # Список ВСІХ конфіг-серверів через кому
  configDB:
configReplSet/cfg1.example.com:27019,cfg2.example.com:27019,cfg3.example
.com:27019
```

Порядок запуску

1. Запустити **всі Config Servers** зі своїм конфігом.
2. Зайти на один з них через mongosh --port 27019 та ініціалізувати їх як Replica Set:

```
rs.initiate({
  _id: "configReplSet",
  configsvr: true,
  members: [
  { _id: 0, host: "cfg1.example.com:27019" },
```

```
  { _id: 1, host: "cfg2.example.com:27019" },
  { _id: 2, host: "cfg3.example.com:27019" }
  ]
})
```

3. Запустити **Shard Servers** . Зайти на кожен і ініціалізувати їх внутрішні Replica Sets (аналогічно rs.initiate (...)).
4. Запустити **Mongos** . Він підчепить налаштування від конфіг-серверів.
5. Підключиться до mongos (порт 27017) і додати шарди в кластер командою sh.addShard ().

Безпека кластера

У реальній мережі обов'язково додайте секцію security з keyFile. Без загального ключа вузли кластера відмовляться спілкуватися один з одним, щоб хтось чужий не підсунув свій шард у вашу систему.

Для того, щоб вузли кластера (шарди, конфіг-сервери та роутери) довіряли один одному, MongoDB використовує ***Shared Key File***. Це свого роду "загальний пароль" для всього кластера. Без нього будь-хто в вашій мережі зможе прикинутися шардом і вкрасти або зіпсувати дані. Для організації цього процесу потрібно виконати такі дії.

1. Генерація KeyFile

Файл повинен містити від 6 до 1024 символів у кодуванні base64 і повинен бути **абсолютно (!)** однаковим на **всіх** серверах кластера (Config, Shards, Mongos)

```
# Генеруємо випадковий рядок і зберігаємо файл
openssl rand -base64 756 > /var/lib/mongodb/mongodb.key

# Встановлюємо суворі права доступу (ОБОВ'ЯЗКОВО!)
# MongoDB відмовиться запускатися, якщо файл доступний комусь, окрім власника
chmod 400 /var/lib/mongodb/mongodb.key
chown mongodb:mongodb /var/lib/mongodb/mongodb.key
```

2. Додавання конфігурації (mongod.conf **і** mongos.conf**)**

Тепер у кожен із трьох типів конфігів, наведених вище, необхідно додати секцію security.

```
security:
  keyFile: /var/lib/mongodb/mongodb.key
  authorization: enabled #   Включає рольову модель доступу
```

Для `mongos` параметр `authorization` не пишется окремо (він бере налаштування з конфіг-серверів), там достатньо тільки шляху до `keyFile`.

3. Створення першого адміністратора

⚠ *Коли keyFile буде увімкнено, база стане закритою. Щоб не замкнути зовні самого себе, потрібно створити адміністратора до того, як все буде остаточно перезапущено з включеною безпекою.*

1. Запустити один шард/конфіг без security.
2. Створити користувача:

```
use admin
db.createUser({
  user: "siteAdmin",
  pwd: "super-strong-password",
  roles: [ { role: "root", db: "admin" } ]
})
```

3. Перезапустити всі вузли з keyFile: ... в конфігу.

Найбільш поширена помилка – це **некоректні права доступу на файл ключа**. Якщо в логах з'являється Permissions on /.../mongodb.key are too open, то потрібно ще раз виконати chmod 400.

Коли все налаштовано, карта портів буде виглядати наступним чином (таблиця 28):

Таблиця 28 – «Карта» портів розподіленого кластера MongoDB

Сервіс	Порт за замовчуванням	З чим з'єднується
Mongos	27017	Застосунки (backend)
Shard SVR	27018	Mongos
Config SVR	27019	Mongos

Обмеження документної моделі

При всіх перевагах документна модель також не є універсальною і має низку обмежень, наприклад:

– вартість з'єднання ($lookup) зростає зі збільшенням кардинальності;
– денормалізація призводить до дублювання зв'язків;
– обхід довгих ланцюжків зв'язків потребує або вкладених запитів, або перенесення логіки в код.

У таких сценаріях ефективнішою стає модель, в якій на чільне місце поставлені зв'язки. Саме цю ідею реалізують графові СУБД, зокрема Neo4j.

Питання для самоперевірки

(до розділу «MongoDB Sharding»)

1. Що таке шардування і для чого воно використовується у MongoDB?
2. Чим горизонтальне масштабування через шардування відрізняється від вертикального масштабування?
3. Які основні компоненти шардуваного кластера MongoDB?
4. Яку роль у шардуваному кластері виконують config servers та mongos?
5. Що таке shard key і чому його вибір є критично важливим?
6. У чому полягають переваги та обмеження шардування?
7. Чому шардування не завжди є універсальним рішенням для підвищення продуктивності системи?

Графова модель даних та СУБД Neo4j

У більшості сучасних інформаційних систем дані не є ізольованими. Їхня цінність визначається не лише окремими фактами, а й структурою зв'язків між ними. У реляційній моделі зв'язки реалізуються через зовнішні ключі та операції з'єднання (JOIN). У документній моделі вони або інкапсулюються всередині агрегату, або відтворюються через механізми з'єднання на рівні запитів.

Однак існують предметні області, в яких зв'язок стає первинною сутністю. У таких випадках дані природно формують граф – структуру з вузлів і зв'язків, де саме відношення між об'єктами має самостійне значення.

Предметну область можна вважати *високозв'язною*, якщо:

– кількість зв'язків між екземплярами сутностей порівнянна або перевищує кількість самих сутностей;
– запити часто вимагають переходів на декілька рівнів глибини;
– важливими є не лише атрибути об'єктів, а й їхнє положення в структурі зв'язків.

Типові предметні області такого роду і, відповідно, *сценарії застосування графової моделі* і СУБД, що її підтримують:

– соціальні мережі;
– рекомендаційні системи;
– управління знаннями;
– наукові публікації та цитування;
– мережі постачань;
– контроль доступу;
– …

Наприклад, створюється БД соціальної мережі (рис. 17). З такої БД треба мати можливість отримати:

– список друзів конкретної особи;
– список друзів тих осіб, які є друзями конкретної особи (проект FOAF – «friend-of-a-friend»/«друг друга»);
– найкоротший шлях між двома особами;
– список спільних знайомих у двох осіб.

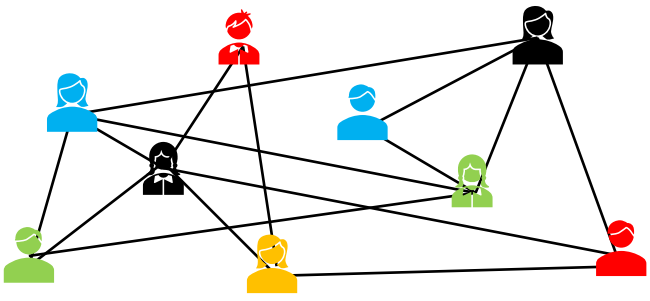


Рис. 17 – Зв'язки в предметній області соціальної мережі

Подібні запити не обмежуються одним рівнем вкладеності, а передбачають проходження по ланцюгах зв'язків довільної довжини. Відповідно, такий запит

може оброблятися довго в *реляційній базі даних*, оскільки необхідно мати зв'язані таблиці, в яких будуть представлені відношення між просто друзями та друзями друзів (рис. 18).

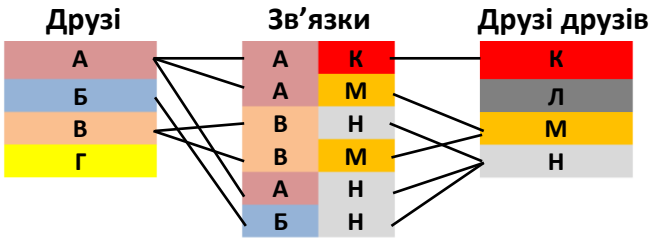


Рис. 18 – Формалізація зв'язків у ПрО соціальної мережі в реляційній моделі

Документна модель тут також буде неефективною через те, що структура соціальної мережі характеризується високою щільністю та глибиною зв'язків. Документна модель орієнтована на концепцію *агрегату* – логічно завершеного набору даних, що зберігається і обробляється як єдине ціле. Це забезпечує високу продуктивність при читанні та записі повного документа.

Проте у задачах соціальних мереж, наукових цитувань або аналізу керівництва студентами центральною сутністю стає не сам документ, а саме зв'язок між екземплярами сутностей. Тому у випадку соціальної мережі можливі два підходи до представлення зв'язків у документній БД:

– вкладення інформації про пов'язаних осіб у документ персони;
– збереження лише посилань (id) на пов'язані документи.

У першому випадку виникає дублювання даних, ускладнення підтримки узгодженості при зміні зв'язків та зростання розміру документа. У другому – для отримання багаторівневих зв'язків («друзі друзів», «спільні знайомі», «найкоротший шлях») необхідні багатоетапні запити з використанням $lookup та/або перенесення частина логіки переходів на рівень застосунку.

Таким чином, при зростанні глибини зв'язків складність запитів у документній моделі зростає непропорційно, що робить її менш придатною для таких предметних областей.

Відповідно, для таких ПрО та задач потрібна модель, яка природно описує и між екземплярами сутностей (значеннями). Математичною основою такої моделі є граф, тобто структура, що складається з об'єктів та зв'язків між ними.

Такий підхід змінює акцент моделювання:

– у реляційній моделі центральною є таблиця;
– у документній – агрегат;
– у графовій – зв'язок між сутностями.

Як зазначають Ян Робінсон, Джим Вебер та Еміль Ейфрем [10], практична цінність графової моделі проявляється у здатності ефективно обробляти великі масиви взаємопов'язаних даних без зростання складності запитів разом із масштабом системи.

Якщо у документній базі основна оптимізація полягає в мінімізації з'єднань, то в графовій – у мінімізації вартості проходження (traversal).

Відповідно, ***графова СУБД доцільна*** у випадках, коли:

- домінують запити до зв'язків, а не до окремих сутностей;
- необхідний аналіз багаторівневих залежностей;
- важливо визначати найкоротші шляхи або спільні зв'язки;
- дані мають високу зв'язність;
- структура зв'язків є динамічною і змінюється частіше, ніж атрибути сутностей;
- система повинна зберігати стабільну продуктивність при зростанні кількості переходів.

Таким чином, графова модель не є універсальною заміною реляційної чи документної, а виступає спеціалізованим інструментом для класу задач, у яких структура зв'язків має самостійне значення.

Таким чином, розглянуті приклади демонструють, що для предметних областей із високою зв'язністю даних доцільно використовувати графову модель. Далі розглянемо її формальні основи та особливості реалізації.

Математичні основи графової моделі даних

Поняття та властивості графу

Граф у загальному вигляді визначається як $G = [V, E]$, де V – множина вершин (vertices), E – множина ребер (edges) [11].

У контексті інформаційних систем *вершина* графу відповідає *екземпляру сутності* (наприклад, студенту, викладачу, публікації), а *ребро* відповідає *відношенню* між екземплярами (навчається у, керує, цитує тощо). При чому як вершини, так і зв'язки можуть мати певні властивості.

У подальшому при розгляді графових СУБД в якості прикладної інтерпретації «вершини» графа буде використовуватися термін «вузол».

Графи поділяються на:

- *неорієнтовані* – зв'язок між вузлами симетричний і позначається ребром графа або двоспрямованою стрілкою;
- *орієнтовані (digraph)* – зв'язок, що позначається дугою, має певний напрям від *початкового* вузла до *кінцевого*.

У більшості графових СУБД, зокрема у Neo4j, використовується орієнтована модель, де кожне відношення має чітко визначений напрям. Наприклад:

```
(Student)-[:STUDIES_IN]->(Group)
```

означає спрямований зв'язок від студента до групи.

Ключовими ***характеристиками*** графа є *ступінь вершини* та *шлях*.

Ступінь вершини/вузла (degree) – це кількість ребер, інцидентних даній вершині.

Для орієнтованих графів розрізняють:

– *вхідний* ступінь (in-degree) – кількість дуг, для яких вузол є кінцевим;
– *вихідний* ступінь (out-degree) – кількість дуг, для яких вузол є початковим.

У прикладних задачах високий ступінь вершини може означати популярність користувача у соціальній мережі або значну кількість цитувань у наукових графах.

Шлях (path) – це послідовність вершин, у якій кожна пара сусідніх вершин з'єднана ребром, тобто є суміжними. Відповідно, ***довжина шляху*** – кількість ребер у цьому шляху.

У задачах аналізу мереж важливими є пошук шляхів заданої довжини, пошук найкоротшого шляху чи визначення досяжності між вершинами.

Наприклад, при аналізі соціальних мереж чи наукового цитування часто необхідно виконувати пошук шляхів довжиною 2, 3 або більше. У традиційних реляційних СУБД такі задачі реалізуються через рекурсивні запити або багаторазові операції з'єднання, а у графових СУБД вони є природною операцією проходження по графу.

Граф називається *зв'язним*, якщо між будь-якими двома його вершинами існує шлях. У контексті інформаційних систем це відповідає можливості досягнення будь-якого об'єкта через ланцюг відношень.

У прикладних інформаційних системах аналіз зв'язності дозволяє визначати кластери користувачів, виявляти ізольовані підмережі, аналізувати структуру наукових співтовариств тощо.

Property Graph Model

На основі класичної теорії графів у сучасних інформаційних системах сформувалася прикладна інтерпретація графа – ***property graph model***, яка лежить в основі більшості графових СУБД [10, 12].

На відміну від класичної теорії графів, де вершини та ребра зазвичай розглядаються абстрактно і не мають внутрішньої структури, *property graph* передбачає наявність ***властивостей (атрибутів, properties)*** як у вершин, так і у зв'язків. Тобто, згідно [10, 15]

Property Graph характеризується трьома ключовими компонентами: вузлами (nodes), зв'язками (relationships) та властивостями (properties).

Вузол представляє окремий екземпляр сутності. Наприклад:

```
(:Student {secondName:"Мікулінська", firstName: "Марія"})
```

Крім того, вузол може мати одну або кілька міток (labels) та довільну кількість властивостей у вигляді пар «ключ–значення». Мітки дозволяють групувати вузли за типом (Student, Group, Publication тощо). У мові Cypher вузол зазвичай записується у круглих дужках, а мітка позначається після символу двокрапки (у попередньому прикладі міткою є Student).

Зв'язок є орієнтованим і з'єднує рівно два вузли.

```
(:Student)-[:STUDIES_IN]->(:Group)
```

Кожен зв’язок має тип (наприклад, STUDIES_IN, CITES, AUTHORED), напрям і може мати власні властивості. Наприклад:

```
(:Publication)-[:CITES {year:2022}]->(:Publication)
```

Таким чином, зв’язок у графовій моделі є самостійним структурним елементом, а не похідною конструкцією, реалізованою через зовнішній ключ.

Властивості – це атрибути вузлів або зв’язків, що зберігаються у вигляді пар «ключ–значення».

На відміну від реляційної моделі, де структура таблиці жорстко фіксована, *property graph* дозволяє різну кількість властивостей у різних вузлів та динамічне розширення моделі. Наприклад:

– вершина Student може мати властивості name, year, gpa;
– ребро [:CITES] може мати властивість citationYear.

⚠ *Ключова особливість property graph полягає у тому, що навігація між вузлами є природною операцією.*

Я. Робинсон, Д. Вебер та Э. Эйфрем у [10] підкреслюють, що ефективність графових СУБД ґрунтується на концепції ***index-free adjacency*** – прямого збереження зв’язків між вузлами, що дозволяє виконувати переходи без глобального пошуку. Це означає, що при проходженні від одного вузла до пов’язаного з ним система не виконує операцію пошуку у таблиці індексів, а використовує безпосередні посилання.

Відповідно, перевагами property graph є:

– явне представлення зв’язків;
– природна підтримка багаторівневих запитів;
– гнучкість структури;
– ефективність traversal.

Таким чином, *property graph* поєднує математичну строгість графової структури з практичними вимогами до зберігання атрибутів та виконання запитів.

Отже, графова структура природно відображає системи, де:

– важливою є навігація між об’єктами;
– запити передбачають переходи довільної глибини;
– зв’язки змінюються частіше, ніж атрибути сутностей.

Для наочності розглянемо узагальнену схему графової структури навчальної бази даних lectures, скрипт створення якої мовою Cypher наведано у Додатку В. У цій моделі вузли відповідають основним сутностям предметної області (викладачі, студенти, теми дипломів, публікації), а зв’язки відображають взаємозв’язки між ними. Спрощена схема графа цієї предметної області наведена на рис. 19.

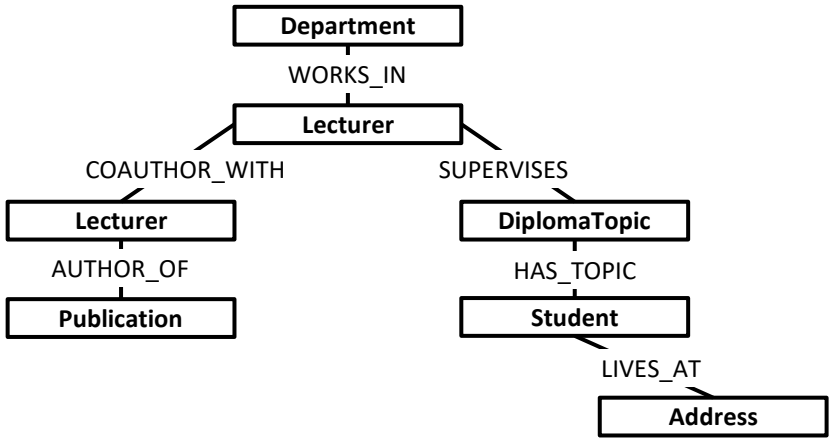


Рис. 19 – Спрощена схема графової структури навчальної бази даних lectures

Як видно зі схеми, графова модель дозволяє безпосередньо представляти складні взаємозв'язки між об'єктами предметної області. Це дає змогу ефективно виконувати запити, пов'язані з навігацією по зв'язках, наприклад пошук співавторів, студентів певного викладача або найкоротших ланцюгів співпраці.

Порівняльний аналіз моделей даних

З урахуванням розглянутих характеристик можна систематизувати ключові відмінності між трьома підходами: реляційною (PostgreSQL), документною (MongoDB) та графовою (Neo4j) моделями (таблиця 29).

Таблиця 29 – Ключові відмінності підходів реляційної, документної та графової моделей

Аспект	PostgreSQL (Реляційна)	MongoDB (Документна)	Neo4j (Графова)
Базова одиниця	Таблиця	Ієрархічний документ з вкладеннями (агрегат)	Вузол, зв'язок та їх властивості
Реалізація зв'язків	Зовнішні ключі (FK)	Денормалізація (вкладення) або посилання	Явні відношення першого класу
Механізм переходу	Операції з'єднання (JOIN)	Операція $lookup	Запити по шляхах (traversal)
Оптимізаційна стратегія	Індекси та оптимізатор запитів (план запиту)	Мінімізація операцій з'єднання	Index-free adjacency (безіндексна суміжність)
Глибокі зв'язки	Рекурсивні операції з'єднання (JOIN)	Багатоетапні запити $lookup	Природне проходження
Гнучкість схеми	Строга або, навіть, фіксована	Гнучка	Гнучка структура зв'язків

Реляційна модель забезпечує строгість та цілісність даних, однак складність запитів зростає зі збільшенням кількості з'єднань.

Документна модель оптимізована для роботи з агрегатами, але у задачах із високою зв'язністю потребує додаткових операцій з'єднання.

Графова модель орієнтована на структуру зв'язків і є доцільною у випадках, коли навігація по мережі відношень становить основну частину робочого навантаження системи.

Ці відмінності можна також відобразити схематично.
Реляційна модель (рис. 20):

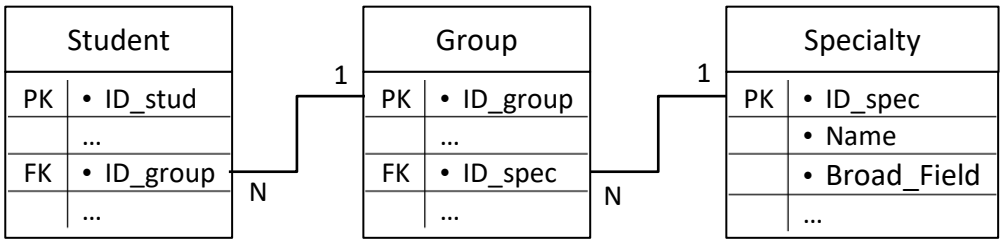


Рис. 20 – Діаграма реляційної моделі

У реляційній моделі зв'язки реалізуються через зовнішні ключі. Для отримання повної інформації про студента необхідне виконання операцій з'єднання між таблицями. При збільшенні кількості пов'язаних таблиць складність запиту зростає.

Документна модель:

```
StudentDocument {
   group: {...},
   specialty: {...},
   grades: [...]
}
```

У документній моделі пов'язані сутності можуть бути вкладені у структуру одного документа. Це забезпечує швидке читання агрегату, однак ускладнює підтримку узгодженості та повторне використання даних при складних зв'язках.

Графова модель:

```
(Student)-[:STUDIES_IN]->(Group)-[:SUBDIVISION_OF]->(Specialty)
(Student)-[:HAS_GRADE]->(Discipline)
(Specialty)-[:DETAILED_FIELD_OF]->(Broad_Field)
```

У графовій моделі зв'язки представлені явно та є повноцінними елементами структури. Це дозволяє ефективно виконувати запити довільної глибини без ускладнення синтаксису запиту та без необхідності явних з'єднань.

Основи СУБД Neo4j

Neo4j – тип NoSQL СУБД, який підтримує графову модель даних. Як випливає з назви, дані в ній зберігаються у вигляді графа саме в тому значенні, в якому він розглядається в математиці [3]. Відмінною особливістю таких баз даних є можливість малювати їх структуру у вигляді вузлів і ліній, що їх з'єднують. Все, що можна зобразити у такому вигляді, можна зберегти в Neo4j. Тобто у Neo4j упор робиться не на загальні характеристики значень (як і колекціях документів чи рядках таблиці), але на зв'язки між значеннями, дозволяючи в такий спосіб легко і природно зберігати абсолютно різнорідні дані.

⚠ Хоча Neo4j має невеликий розмір, в ній можна зберігати десятки мільярдів вузлів і стільки ж ребер. А враховуючи підтримку реплікації master-slave на безліч серверів, ця СУБД здатна впоратися із завданням практично будь-якого розміру.

Архітектура Neo4j

Neo4j є представником нативних графових СУБД, архітектура яких побудована безпосередньо навколо *property graph model*.

На відміну від реляційних систем, де графова структура може бути змодельована поверх таблиць, Neo4j зберігає вузли та зв'язки як первинні елементи структури даних. Такий підхід називають ***native graph storage*** [12].

Якщо у реляційних СУБД зв'язки реалізуються через зовнішні ключі, що вимагає виконання операцій з'єднання для відновлення структури зв'язків, то у Neo4j:

- вузли зберігаються як окремі записи;
- кожен зв'язок містить прямі посилання на початковий і кінцевий вузол;
- зв'язки організовані у вигляді пов'язаних структур.

Таким чином, графова структура не відтворюється під час виконання запиту – вона *фізично існує у сховищі*.

Як було зазначено вище, однією з ключових особливостей Neo4j є механізм ***index-free adjacency***, описаний в роботі [10]. Його суть полягає у тому, що:

- кожен вузол містить прямі посилання на пов'язані з ним зв'язки;
- перехід від одного вузла до іншого виконується без глобального пошуку;
- складність локального переходу не залежить від загального розміру графа.

У свою чергу, це означає, що при виконанні обходу (traversal) графа система не сканує таблиці або індекси для знаходження наступного вузла, а використовує вже наявні структурні посилання. У результаті продуктивність запитів, що передбачають проходження по зв'язках, є стабільною, а глибина переходів впливає на час виконання більше, ніж загальний обсяг даних.

Зазначена властивість забезпечує ефективність графових СУБД у задачах із високою зв'язністю. Слід зауважити, що index-free adjacency забезпечує ефективність локальних переходів між вузлами, однак загальна продуктивність запиту також залежить від структури графа та характеру traversal.

Для роботи з БД Neo4j використовує декларативну мову запитів Cypher, яка дозволяє описувати шаблони зв'язків у вигляді графових патернів. Наприклад:

```
MATCH (t:Teacher)-[:AUTHORED]->(p)<-[:AUTHORED]-(co)
RETURN DISTINCT co
```

Такий запит не вимагає явного опису операцій з'єднання. Система виконує навігацію по графу відповідно до заданого шаблону.

Neo4j підтримує властивості ACID (Atomicity-Consistency-Isolation-Durability). Це дозволяє використовувати систему не лише для аналітичних задач, але й у транзакційних сценаріях. Підтримка ACID відрізняє Neo4j від деяких інших NoSQL-підходів, орієнтованих переважно на eventual consistency.

Таким чином, Neo4j поєднує гнучкість графової структури, ефективність traversal і транзакційну надійність. Ця СУБД не є універсальною заміною інших підходів, але забезпечує природну та ефективну реалізацію задач, у яких структура зв'язків є домінуючою характеристикою даних.

Інструменти для роботи з Neo4j

Залежно від цілей розробника можна вибрати один із безкоштовних варіантів СУБД:

1. ***Neo4j Desktop***: Ідеально для навчання та розробки на ПК (Windows, MacOS, Linux). Включає безкоштовну ліцензію розробника для Enterprise Edition. Це графічна оболонка (IDE). Вона зручна для розробки, тому що дозволяє в пару кліків створювати різні проекти, перемикати версії БД та встановлювати плагіни.
2. ***Neo4j AuraDB***: Хмарна версія (SaaS). Нічого не потрібно встановлювати, є безкоштовний рівень (Free Tier) для невеликих проектів.
3. ***Docker***: Підійде, якщо ви віддаєте перевагу контейнеризації. Використовуйте офіційний образ neo4j на Docker Hub.
4. ***Neo4j Community Server***: Базова версія для встановлення на Linux/Windows сервери у вигляді сервісу. Це саме сервер БД. Його використовують, коли треба налаштувати Neo4j як системну службу (Service), запустити в мінімалістичному оточенні або сервері, де не потрібен зайвий інтерфейс.

Neo4j Desktop і Neo4j Community Server можуть працювати паралельно. Сервер із архіву можна запустити на одному порту, а базу в Desktop – на іншому. Також Desktop вміє підключатися до запущеного Community-сервера як до «віддаленої» БД (Remote Connection).

В якості інструментів навчальної роботи будемо використовувати Neo4j Desktop та Neo4j Browser, через те, що для більшості навчальних задач достатньо встановити Neo4j Desktop, який після інсталяції готовий до роботи (рис. 21).

Проте, якщо є необхідність встановлення Community Server, що можна зробити і після інсталяції Desktop, то в ОС Windows для цього потрібні:

- Neo4j Community Server, який поставляється у вигляді архіву і яки треба розпакувати на диск;
- Java – для версій сервера 2025+ та 2026+ потрібно Java 21 або Java 25 (треба переконайтись, що вона встановлена та змінна JAVA_HOME налаштована);
- командна консоль (наприклад, Windows PowerShell).

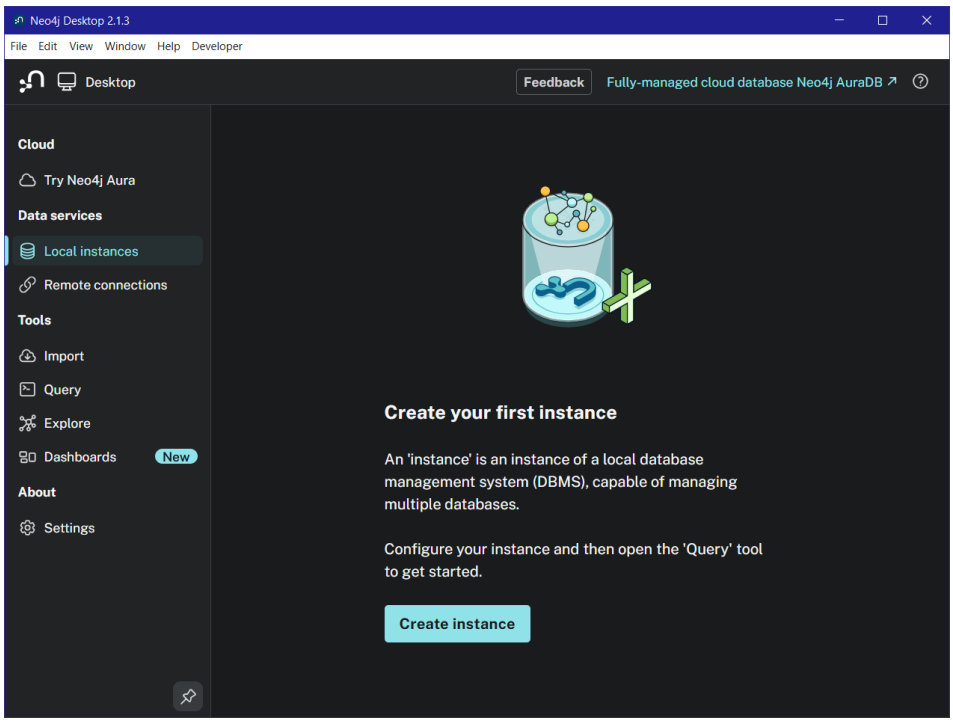


Рис. 21 – Початковий екран Neo4j Desktop

Для початку роботи

1. Відкрийте PowerShell або командний рядок від імені *адміністратора*.
2. Перейдіть до папки bin розпакованого архіву.
3. Виконайте команду (рис. 22): .\neo4j console

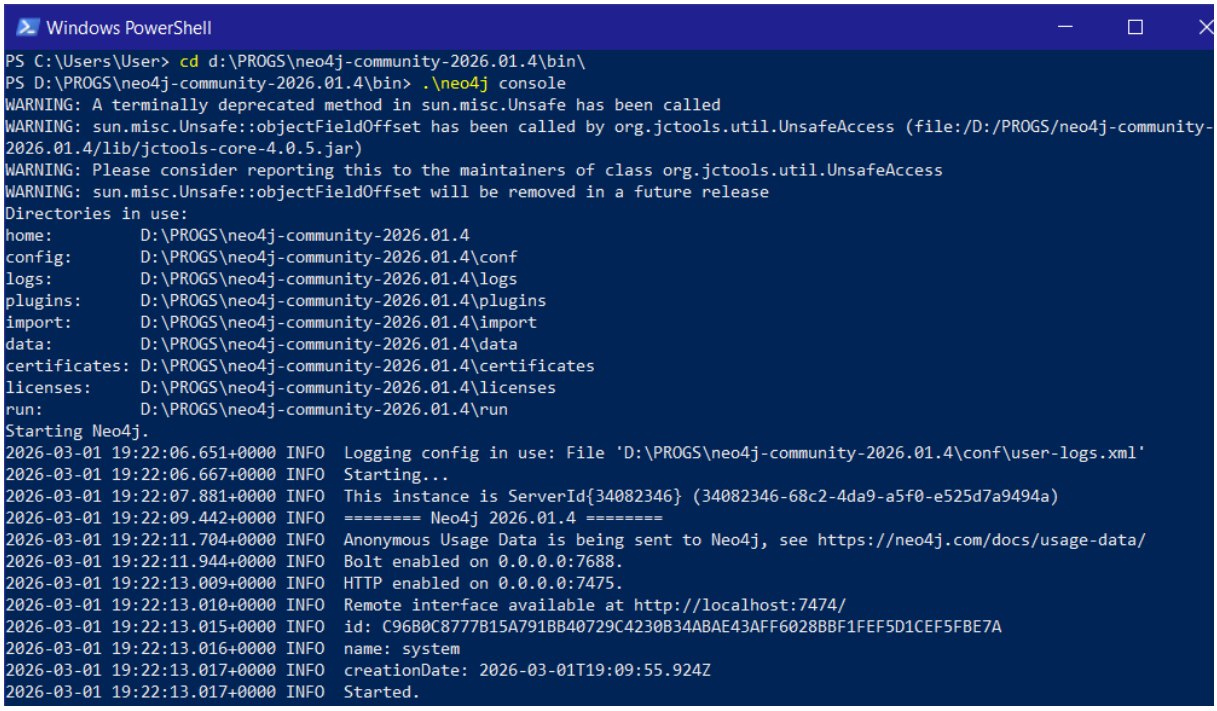


Рис. 22 – Запуск Neo4j Community Server

4. Доступ до бази: Після запуску в логах з'явиться адреса (зазвичай http://localhost:7474). Відкрийте його в браузері – це буде робочий інтерфейс Neo4j Browser (рис. 23).

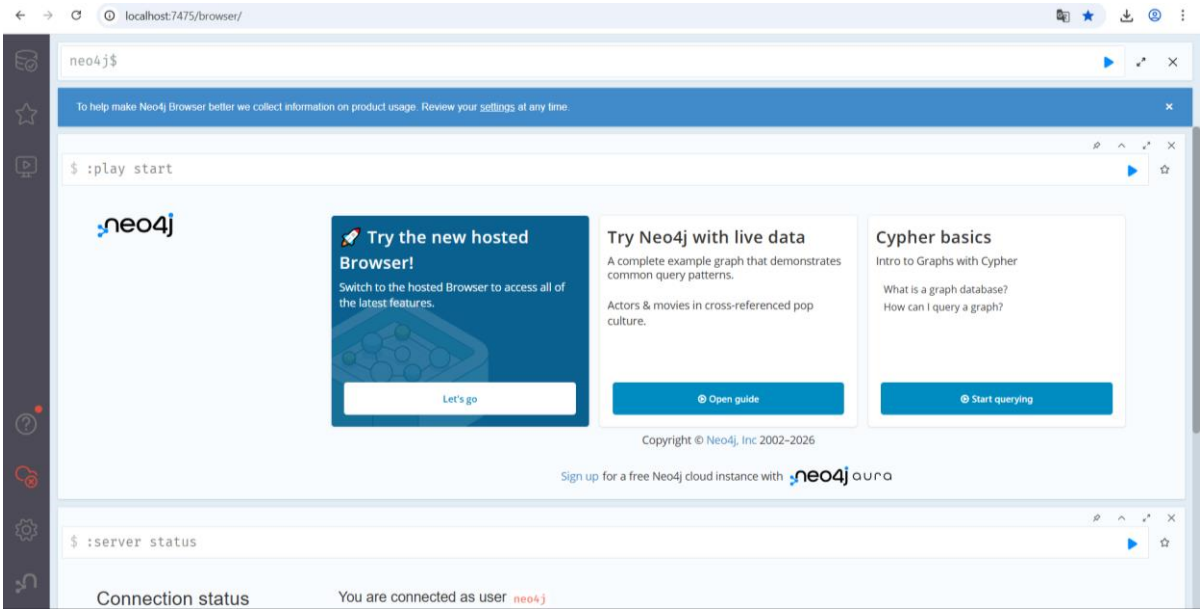


Рис. 23 – Вікно Neo4j Browser

При першому конекті треба буде ввести логин «neo4j» та пароль «neo4j», який буде запропоновано змінити на новий.

5. Встановлення як служби (опціонально): Щоб база працювала на тлі та запускалася разом з Windows, треба використати команду: .\neo4j install-service.

Для ***налаштування*** Neo4j Community Server так, щоб він працював незалежно і був доступний по мережі (або для підключення з Neo4j Desktop), необхідно відредагувати файл конфігурації, який знаходиться в папці, куди був розпакований архів:

<NEO4J_HOME>\conf\neo4j.conf

Якщо Neo4j Desktop запущено, він може займати стандартні порти (7474, 7687). Для Community-версії їх краще змінити (або переконатися, що Desktop-проекти вимкнені), для чого необхідно знайти та розкоментувати наступні рядки:

– ***Bolt (бінарний протокол)***:

server.bolt.listen_address=:7688 (слід змінити з 7687 на 7688, щоб уникнути конфліктів)

– ***HTTP (браузер)***:

server.http.listen_address=:7475 (аналогічно змінити з 7474 на 7475)

Далі необхідно дозволити доступ до мережі. За промовчанням Neo4j слухає лише localhost. Щоб база була видна іншим комп’ютерам у мережі, необхідно знайти і розкоментувати рядок

server.default_listen_address=0.0.0.0

Тепер можна додати запущений Community-сервер до інтерфейсу Desktop як «віддалену» базу. Для цього (рис. 24):

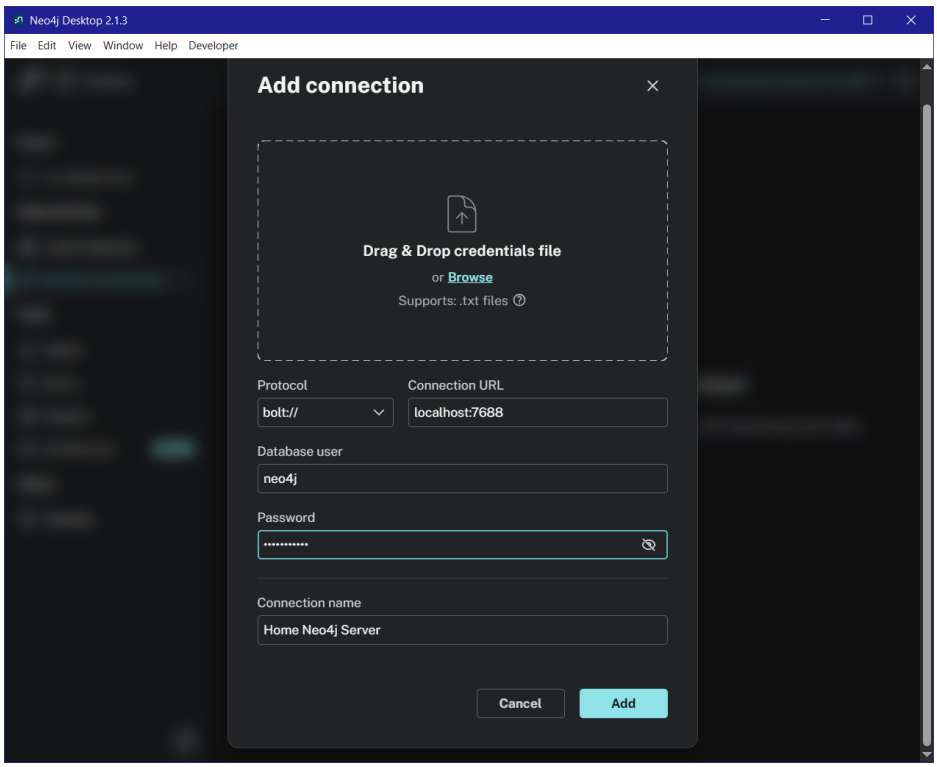


Рис. 24 – Під'єднання до Neo4j Community Server з Neo4j Desktop

1. У програмі Neo4j Desktop виберіть проект.
2. Натисніть Add -> Remote connection.
3. Введіть ім'я та адресу: bolt://localhost:7688.
4. Введіть логін «neo4j» та пароль, встановлений через браузер під час першого запуску Community-сервера.

Надалі з БД на сервері можна працювати і вводити Cypher-запити як через консоль Web-інтерфейсу (рис. 25), так і через консоль в Neo4j Desktop (рис. 26).

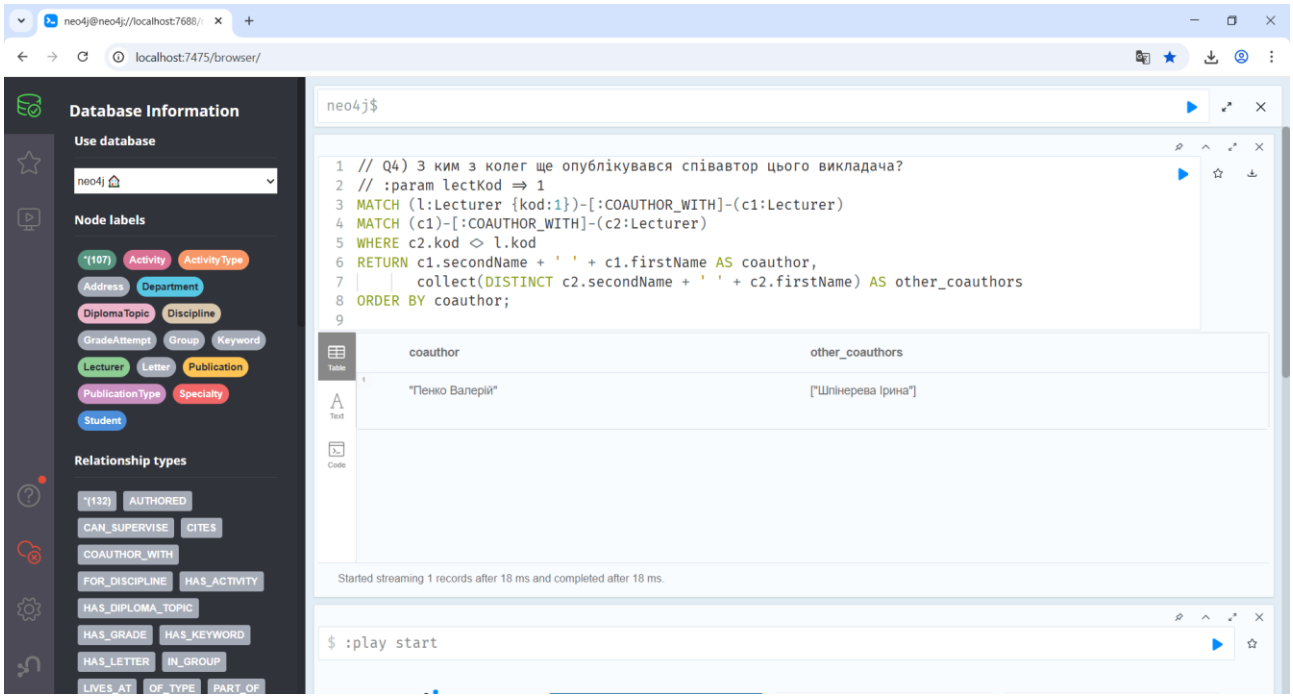


Рис. 25 – Web-консоль Neo4j

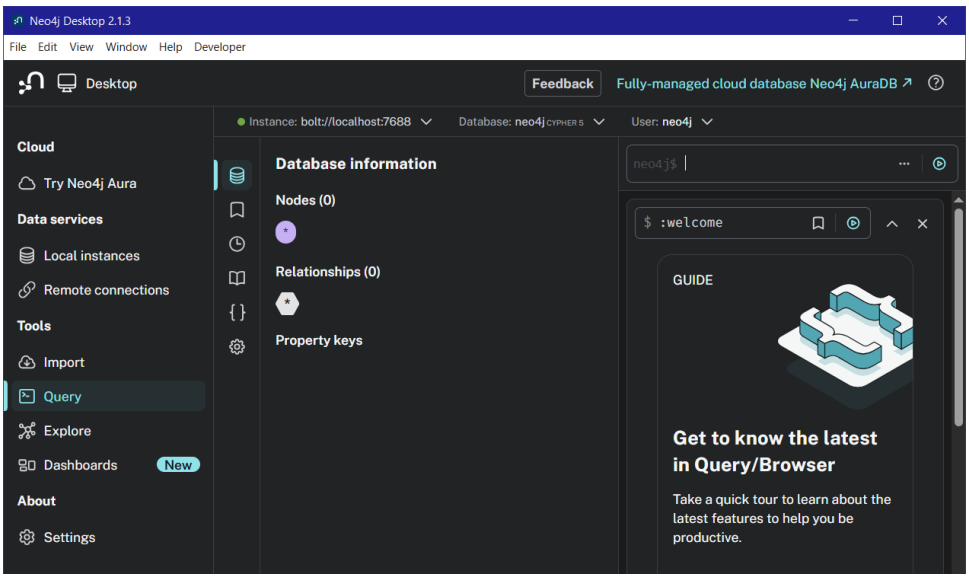


Рис. 26 – Консоль Neo4j Desktop

Питання для самоперевірки

(до вступної частини розділу «Графова модель даних та СУБД Neo4j» та розділів «Математичні основи графової моделі даних», «Основи СУБД Neo4j»)

1. У чому полягають основні відмінності між реляційною, документною та графовою моделями даних?
2. Що таке граф у математичному розумінні?
3. Які основні елементи графа використовуються у графових БД?
4. Що таке орієнтований граф і чому саме така модель використовується у Neo4j?
5. Для чого у Neo4j використовуються мітки (labels)?
6. У чому полягає концепція property graph?
7. Що таке index-free adjacency і чому цей принцип є важливим для Neo4j, зокрема при виконанні обходу графа (traversal)?
8. Що таке задачі типу «friend-of-a-friend», і як вони розв’язуються в реляційній, документній та графовій БД?
9. Які основні компоненти та інструменти екосистеми Neo4j використовуються для роботи з графовими БД?
10. У чому полягають особливості зберігання та навігації по зв’язках у Neo4j?

Друге розширення ПрО реляційної БД «Навчальний процес»

Використовуючи наведені інструменти, розглянемо реалізацію другого розширення предметної області «Навчальний процес» у графовій СУБД Neo4j.

На відміну від розширення, запропонованого для документної моделі, у графовій моделі змінюється не лише спосіб зберігання даних, а й характер запитів до них.

Нагадуємо, що вихідна РБД lectures описує ПрО «Навчальний процес» з відповідними сутностями: підрозділи, викладачі, студенти, дисципліни, оцінювання, дипломні роботи, листування. До того ж залишаємо додані в межах документної моделі активності викладачів та їх публікації, дані про які трохи розширимо.

Адже ***розширення ПрО*** полягає в тому, що ключовими стають *мережеві зв'язки* та *переходи по шляхах*, зокрема:

а) **освітні траєкторії розглядаються як мережа зв'язків**

Сутності залишаються ті ж самі, але завдання змінюється – аналіз буде виконуватись не «за таблицями», а «за зв'язками»

студент → спеціальність → дисципліни → оцінки → дипломна робота → керівник → кафедра

кафедра → викладач → публікації

кафедра → викладач → активності

що дозволяє також визначати «сусідів» студента: хто керував, які дисципліни спільні, які траєкторії подібні тощо.

Типовими запитами до такої структури можуть бути, наприклад:

- знайти студентів, які пов'язані з кафедрою через ланцюжок «дисципліна → викладач → підрозділ» (*знайти студентів, які навчались у X, слухали дисципліни у викладачів із підрозділу Y і далі потрапили на дипломи до керівника Z*);
- знайти студентів з подібним профілем дисциплін/оцінок (спільні вузли та ребра);
- виявити «центральних» викладачів за кількістю зв'язків (керівництво + дисципліни + комунікації);
- виявити, роботою кого ще зі студентів керує керівник цього дипломника;
- визначити, з ким з колег ще опублікувався співавтор цього викладача.

б) **висвітлюється спільність/перетин навчальних контекстів** (групи, спільні дисципліни, «спільні керівники», «ланцюжки впливу»)

Типовими запитами до такої структури можуть бути, наприклад:

- хто з студентів має найсильніший зв'язок з кафедрою через дисципліни + керівництво + листування.

в) **комунікації розглядаються як частина освітнього процесу**

Листи/повідомлення моделюються як **вузли подій** (а не просто TEXT у таблиці), і стають пов'язаними зі студентом та, за потреби, дисципліною або викладачем, що дозволяє аналізувати *контекст комунікацій*.

г) **оцінювання розглядається як властивість взаємодії**

Оцінка природно виступає *властивістю зв'язку* між Студентом і Дисципліною:

```
(Student)-[:HAS_GRADE {mark, mDate}]->(Discipline)
```

Так структура демонструє, що оцінювання – це не «окрема сутність», а «атрибут відношення».

д) **введене раніше додаткове розширення – науковий блок кафедри**, тобто публікації, співавторство та додатково: ключові слова/напрями, цитування.

Це взагалі класичний випадок, де «зв'язок» – центральна сутність.

Неефективність попередніх моделей даних при такій постановці задач в цій предметній області полягає в наступному.

Реляційна БД для таких задач вимагає довгих ланцюжків операції з'єднання (JOIN) через багато таблиць та проміжних відповідностей (на кшталт Student → Rating → Discipline → …; Student → DWExec → Dwlist → Lecturer → Department; Department → Department (PKod) … тощо), рекурсивних запитів (для ієрархій, транзитивних зв'язків), складної логіки запитів для «пошуку шляхів» і «зв'язків на відстані 2–4» (тобто для «глибоких переходів» у 2–5 кроків і більше), що погано масштабується під навігацію по зв'язках.

Тобто SQL гарний для фактологічних вибірок і звітності, але *погано читається* і *обтяжується* при навігації через мережу зв'язків.

У ***документній моделі*** такі задачі призводять до дилеми:

а) впроваджувати ***денормалізацію*** **і** зберігати «все у документі» (тобто, вбудовувати, наприклад, дисципліни/оцінки/керівників у документи студента), в результаті чого:
   – зростає дублювання (особливо для співавторства, загальних дисциплін, спільних подій);
   – оновлення стають складними;
   – запити «по зв'язках» переносяться у агрегації/$lookup чи до коду застосунку.

б) зберігати «за посиланням між колекціями» і в результаті фактично знову отримати JOIN-подібні операції, але вже без суворих гарантій цілісності та з ускладненням логіки застосування.

У Додатку В наведений скрипт створення графової БД мовою Cypher для відправлення через консоль. Створені елементи БД можна переглядати побачити як у Neo4j Desktop (рис. 27, рис. 28), так і у Web-інтерфейсі Neo4j (рис. 29),

змінюючи при цьому рівень деталізаціїї від «загального» погляду на всю ПрО до укрупненого виділення окремих областей графа.

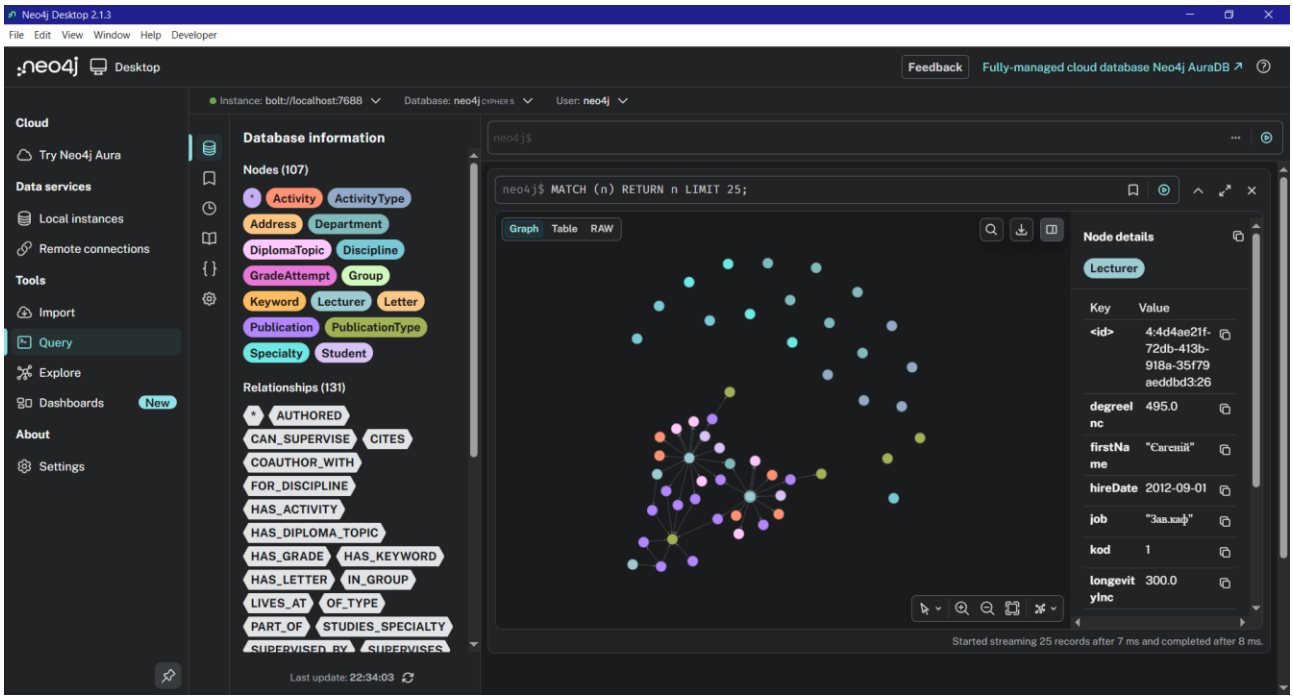


Рис. 27 – Вузли, зв'язки та їх властивості у Neo4j Desktop: початковий (загальний) вигляд графової моделі ПрО «Навчальний процес»

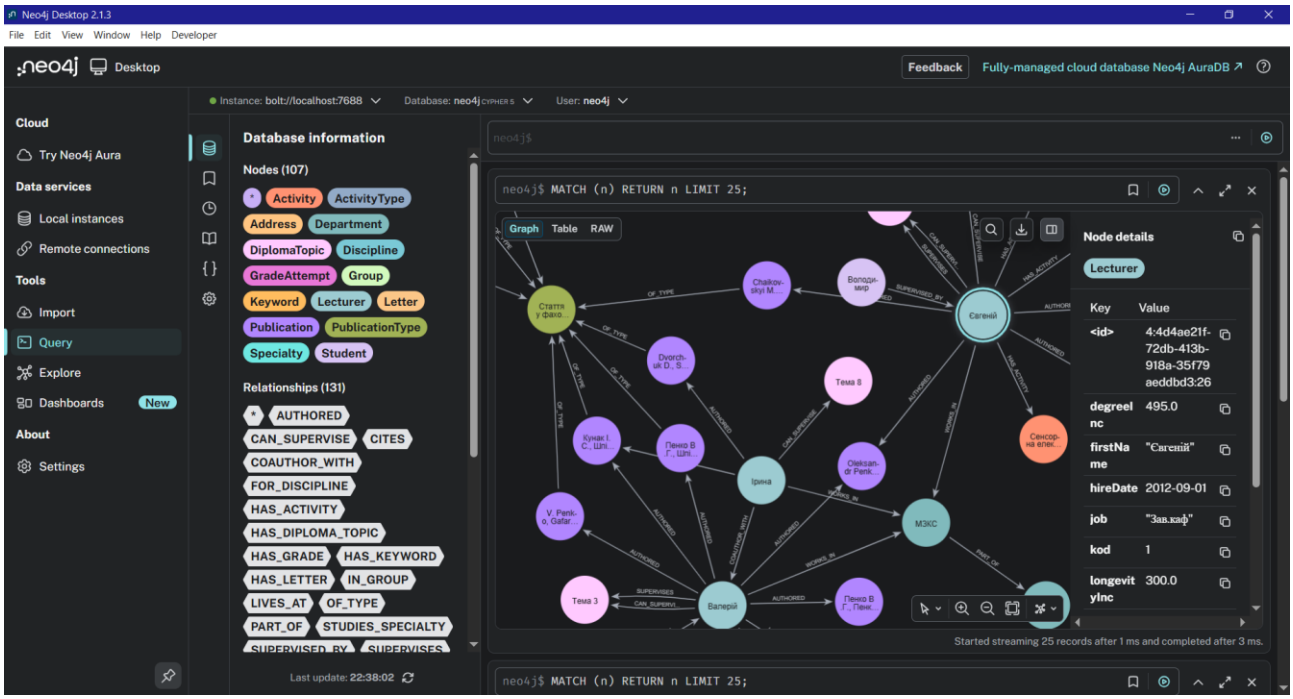


Рис. 28 – Вузли, зв'язки та їх властивості у Neo4j Desktop: деталізація вузлів та зв'язків графу ПрО «Навчальний процес»

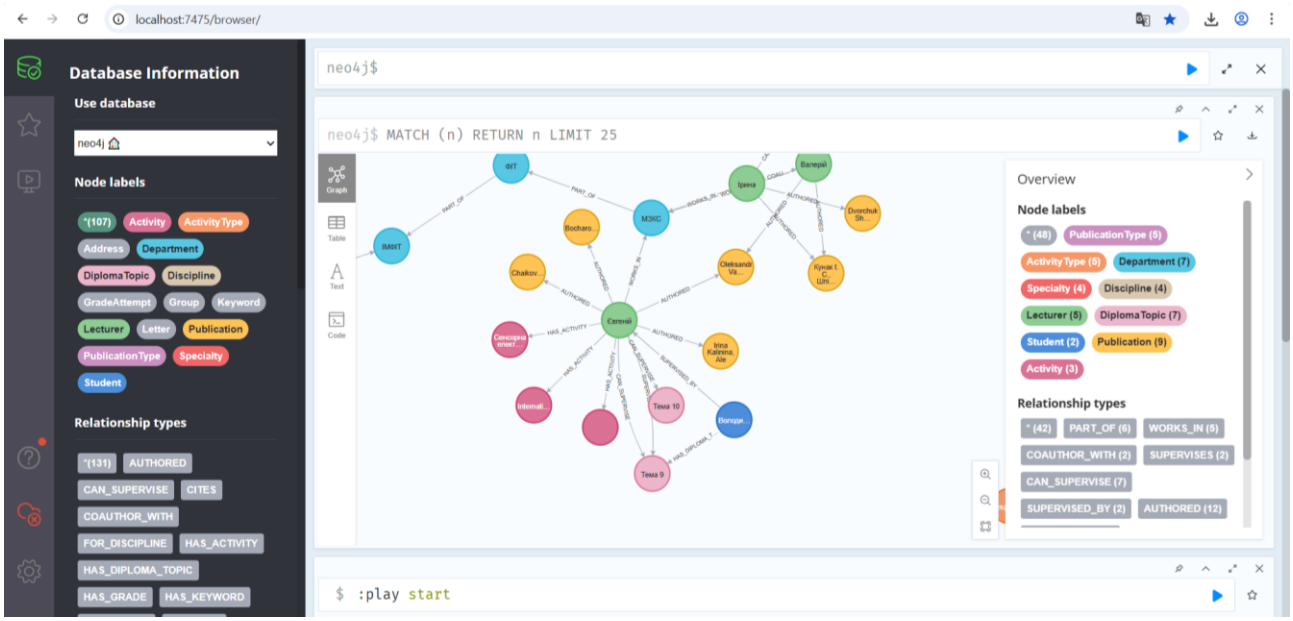


Рис. 29 – Вузли, зв'язки та їх властивості у Web-застосунку Neo4j

Завдання для самостійної роботи

Оберіть предметну область (ПрО) з наведеного переліку із задачами, для розв'зання яких доцільна графова модель даних, та:

– визначте початковий перелік типів сутностей обраної ПрО;
– визначте типи зв'язків між сутностями;
– запропонуйте фрагмент графової моделі для обраної ПрО.

1. Наукова співпраця викладачів та дослідників

1.1.Система призначена для:

– зберігання інформації про дослідників, кафедри, університети та наукові групи;
– накопичення відомостей про публікації, конференції, журнали та проєкти;
– зберігання інформації про співавторство;
– фіксації участі дослідників у наукових заходах;
– аналізу мережі наукової співпраці.

1.2.Задачі:

1.2.1. Знайти всіх співавторів заданого дослідника.
1.2.2. Визначити дослідників, які пов'язані із заданим дослідником через одного або двох спільних співавторів.
1.2.3. Знайти найкоротший шлях наукової співпраці між двома дослідниками.
1.2.4. Визначити наукові групи або кафедри, між якими існує найбільше спільних публікацій.
1.2.5. Знайти дослідників, які є центральними вузлами мережі співпраці.
1.2.6. Знайти дослідників, які виконували задану роль у спільних публікаціях або проєктах, наприклад автор, керівник, рецензент, учасник, та визначити кількість таких зв'язків.

2. Рекомендаційна система навчальних курсів

2.1.Система призначена для:

- зберігання інформації про студентів, курси, теми та компетентності;
- накопичення даних про проходження курсів студентами;
- зберігання зв'язків між курсами, темами та попередніми знаннями;
- формування рекомендацій щодо подальшого навчання;
- аналізу освітніх траєкторій студентів.

2.2.Задачі:

2.2.1. Знайти курси, які рекомендуються студенту на основі вже пройдених курсів.

2.2.2. Визначити курси, що мають спільні теми або компетентності.

2.2.3. Знайти навчальну траєкторію від базового курсу до спеціалізованого.

2.2.4. Визначити студентів зі схожими освітніми траєкторіями.

2.2.5. Знайти теми, які є найбільш пов'язаними з різними курсами.

2.2.6. Знайти курси, для яких не визначено попередні курси або пов'язані компетентності, та сформувати список таких неповних фрагментів освітньої траєкторії.

3. Соціальна мережа професійних контактів

3.1.Система призначена для:

- зберігання профілів користувачів;
- накопичення інформації про професійні навички, компанії та посади;
- зберігання зв'язків між користувачами;
- фіксації участі користувачів у проєктах і спільнотах;
- аналізу професійної мережі контактів.

3.2.Задачі:

3.2.1. Знайти спільних знайомих двох користувачів.

3.2.2. Побудувати ланцюжок контактів між двома користувачами.

3.2.3. Знайти користувачів із подібними навичками та досвідом.

3.2.4. Визначити найбільш пов'язаних користувачів у певній професійній спільноті.

3.2.5. Рекомендувати нові контакти на основі спільних проєктів, навичок або знайомих.

3.2.6. Знайти користувачів, які мають заданий тип професійного зв'язку з іншими користувачами, наприклад colleague, mentor, partner, та визначити найактивніші типи зв'язків.

4. Логістична мережа постачання

4.1.Система призначена для:

- зберігання інформації про постачальників, склади, транспортні вузли та клієнтів;
- накопичення даних про маршрути доставки;
- зберігання зв'язків між пунктами логістичної мережі;
- аналізу альтернативних маршрутів;

– виявлення критичних вузлів постачання.

4.2.Задачі:

4.2.1. Знайти всі можливі маршрути між складом і клієнтом.

4.2.2. Визначити найкоротший або найменш витратний маршрут доставки.

4.2.3. Знайти критичні вузли, відмова яких розриває логістичний ланцюг.

4.2.4. Визначити постачальників, які пов’язані з найбільшою кількістю клієнтів.

4.2.5. Знайти альтернативні маршрути у випадку недоступності певного транспортного вузла.

4.2.6. Знайти маршрути між двома пунктами з обмеженням максимальної кількості проміжних вузлів та порівняти їх за довжиною або вартістю.

5. Система аналізу кіберінцидентів

5.1.Система призначена для:

– зберігання інформації про користувачів, пристрої, облікові записи та мережеві ресурси;
– накопичення даних про події безпеки;
– зберігання зв’язків між інцидентами, пристроями, IP-адресами та користувачами;
– аналізу ланцюжків атак;
– виявлення підозрілих взаємозв’язків у мережі.

5.2.Задачі:

5.2.1. Знайти всі пристрої, пов’язані з певним кіберінцидентом.

5.2.2. Визначити шлях поширення атаки між пристроями або обліковими записами.

5.2.3. Знайти користувачів, які взаємодіяли з підозрілими ресурсами.

5.2.4. Виявити IP-адреси або пристрої, що беруть участь у кількох інцидентах.

5.2.5. Знайти найкоротший ланцюжок зв’язків між підозрілим обліковим записом і критичним ресурсом.

5.2.6. Знайти облікові записи або пристрої, які не мають прямого зв’язку з інцидентом, але пов’язані з ним через ланцюжок підозрілих взаємодій обмеженої довжини.

Якщо раніше виконувався проєкт в реляційній моделі, можна запропонувати розширення предметної області РБД за аналогією наведеного приклада ПрО «Навчальний процес».

Демонстраційний приклад

Завдання

Предметна область: «Портфоліо студентських проєктів» у графовій моделі

ДГ1. Система призначена для:

- зберігання інформації про студентів, навчальні проєкти, викладачів-керівників, технології та тематичні напрями;
- фіксації участі студентів у проєктах з різними ролями;
- зберігання зв'язків між проєктами, технологіями та темами;
- аналізу спільної участі студентів у проєктах;
- пошуку подібних проєктів, спільних технологій і потенційних рекомендацій.

ДГ2. Задачі:

ДГ2.1. Знайти всі проєкти, у яких брав участь заданий студент, із зазначенням його ролі, та, якщо є, їх керівників.

ДГ2.2. Визначити студентів, які працювали над спільними проєктами.

ДГ2.3. Знайти проєкти, подібні до заданого, за спільними технологіями або тематичними тегами.

ДГ2.4. Визначити технології, які найчастіше використовуються у проєктах студентів певної групи або спеціальності.

ДГ2.5. Знайти шлях зв'язку між двома студентами через спільні проєкти, технології або керівників.

ДГ2.6. Знайти студентів, які виконували задану роль у кількох проєктах.

Розв'язання

Початковий перелік типів сутностей

1) ***Student*** — студент.
   Основні властивості:
   student_id, second_name, first_name, group, speciality, study_year

2) ***Project*** — студентський проєкт.
   Основні властивості:
   project_id, title, description, project_type, created_at, updated_at

3) ***Technology*** — технологія, мова програмування, фреймворк або СУБД.
   Основні властивості:
   name, category

4) ***Tag*** — тематичний тег або напрям проєкту.
   Основні властивості:
   name, category

5) ***Supervisor*** — керівник або консультант проєкту.
   Основні властивості:
   supervisor_id, second_name, first_name, academic_position

Початковий перелік типів зв'язків

1) (:Student)-[**:PARTICIPATED_IN** {role, joined_at}]->(:Project)
   Студент бере участь у проєкті в певній ролі (role) з конкретного моменту (joined_at). В перспективі можна додати ще властивості «витрати часу/навантаження» (workload) та «над чим працював» (contribution).

2) (:Project)-[**:USES** {purpose}]->(:Technology)
   Проєкт використовує певну технологію з відповідним призначенням (purpose) застосування: «БД», «UI», «front-end» тощо.
3) (:Project)-[**:HAS_TAG**]->(:Tag)
   Проєкт належить до певного тематичного напряму.
4) (:Supervisor)-[**:SUPERVISES** {review_score}]->(:Project)
   Керівник супроводжує або оцінює проєкт «оцінкою» (review_score). За необхідністю можна додати властивості «дата оцінки» (review_date), «стан проєкту: схвалено/відхилено» (status).
5) (:Project)-[**:SIMILAR_TO**]->(:Project)
   Похідний (необов'язковий) зв'язок, який *доцільно* створювати між проєктами зі спільними технологіями або тегами. В якості властивостей можна додати «вагу» / «коефіцієнт схожості» (similarity) та її обґрунтування (reason).

Графовий шаблон предметної області

```
(:Student)-[:PARTICIPATED_IN {role, joined_at}]->(:Project)
(:Project)-[:USES  {purpose}]->(:Technology)
(:Project)-[:HAS_TAG]->(:Tag)
(:Supervisor)-[:SUPERVISES {review_score}]->(:Project)

(:Student)-[:PARTICIPATED_IN]->(:Project)<-[:PARTICIPATED_IN]-(:Student)
(:Project)-[:USES]->(:Technology)<-[:USES]-(:Project)
(:Project)-[:HAS_TAG]->(:Tag)<-[:HAS_TAG]-(:Project)
```

Пояснення вибору графової моделі

У ***документній*** моделі ***«Портфоліо студентських проєктів»*** основним агрегатом був *документ проєкту*. Це було зручно для зберігання картки проєкту, його авторів, технологій, етапів і відгуків.

У ***графовій*** моделі акцент зміщується з документа проєкту на *зв'язки між сутностями*:

- які студенти працювали разом, хто був team-lead та хто підключився пізніше;
- які проєкти використовують однакові технології;
- які технології поєднують різні тематичні напрями;
- які керівники супроводжують близькі за тематикою проєкти;
- які непрямі зв'язки існують між студентами, проєктами та технологіями.

Саме такі задачі природно формулюються як пошук графових шаблонів і обходи зв'язків, тому для цього варіанта предметної області графова модель є доцільною.

Мова Cypher

Cypher – це декларативна мова запитів для графових баз даних, розроблена для опису шаблонів зв'язків у property graph model [13].

На відміну від SQL, де запит формулюється як операція над таблицями, у Cypher користувач описує ***графовий патерн***, який система повинна знайти у базі даних.

Ключовою особливістю Cypher з точки зору Я. Робинсона, Д. Вебера та Э. Эйфрема [10] є орієнтація на ***pattern matching*** – зіставлення структур зв'язків.

Аналогія з SQL: DDL, DML, DCL у Cypher

В якості «мови даних» Neo4j використовує декларативну мову **Cypher**, призначену для роботи з property graph model. Хоча Cypher не поділяється формально на DDL, DML і DCL, у навчальному та архітектурному контексті доцільно застосувати *умовну* аналогію з SQL (таблиця 30).

Таблиця 30 – «Мова даних» Neo4j у термінах DDL, DML, DCL (SQL vs Cypher)

Категорія	SQL	Cypher (Neo4j)	Особливості
DDL	CREATE TABLE, ALTER, DROP	CREATE CONSTRAINT, CREATE INDEX	Граф не має жорсткої схеми; структура формується під час створення вузлів
DML	SELECT, INSERT, UPDATE, DELETE	MATCH, CREATE, MERGE, SET, DELETE	Патерновий підхід до роботи з даними
DCL	GRANT, REVOKE	CREATE USER, CREATE ROLE, GRANT, DENY	Рольова модель доступу

Особливістю формування структури графа є те, що на відміну від SQL, де DDL задає структуру таблиць до вставки даних, у Neo4j мітки вузлів (labels), типи зв'язків і властивості з'являються під час виконання команд CREATE або MERGE.

Наприклад:

```
CREATE (:Teacher {id:1, name:"Малахов"});
CREATE (:Publication {id:101, title:"Graph Databases in Education"});
```

Структура моделі «Teacher – AUTHORED – Publication» виникає без попереднього опису схеми.

Тому створення вузлів формально є DML-операцією, однак з точки зору побудови предметної області воно виконує роль ініціалізації структури графа.

Мова визначення даних (DDL) у Neo4j

Neo4j відноситься до схемно-гнучких СУБД. Це означає, що повний попередній опис структури не є обов'язковим.

Тому до мови визначення даних Cypher можна віднести тільки засоби забезпечення цілісності та прискорення пошуку початкових вузлів, тобто команди, що формують «каркас» бази даних: визначення обмежень унікальності, обмежень існування властивостей, типових обмежень та створення індексів.

До цієї групи віднесемо:

- визначення унікальності, існування властивості, типу властивості – CREATE CONSTRAINT ...;
- створення простих та складених індексів – CREATE INDEX ...;
- створення індексів повнотекстового пошуку та ін.

Наприклад, визначення унікальності вузлів Lecturer:

```
CREATE CONSTRAINT lecturer_kod_unique IF NOT EXISTS
FOR (l:Lecturer)
REQUIRE l.kod IS UNIQUE;
```

Таким чином, DDL у Neo4j існує, але носить менш жорсткий характер, ніж у SQL.

Мова маніпулювання даними (DML) у Neo4j

Команди і оператори маніпулювання даними (DML) є центральною частиною Cypher. До операцій, шо ними реалізуються, відносяться:

- створення вузлів та зв'язків (CREATE, MERGE);
- пошук вузлів та властивостей (MATCH, WHERE, RETURN);
- оновлення властивостей (SET);
- видалення елементів графу (DELETE, DETACH DELETE);
- масова обробка вузлів (UNWIND);
- агрегації (COUNT, COLLECT).

Наприклад, створення зв'язку

```
MATCH (t:Teacher {id:1}),
    (p:Publication {id:101})
CREATE (t)-[:AUTHORED]->(p);
```

або вибірка даних (аналог SELECT)

```
MATCH (t:Teacher)-[:AUTHORED]->(p)
RETURN t.name, p.title;
```

В Cypher не існує прямої реалізації операції з'єднання (JOIN) – замість цього описується *графовий патерн*.

Мова управління даними (DCL)у Neo4j

Neo4j підтримує керування користувачами та ролями, зокрема:

- створення/керування ролями й користувачами

```
CREATE ROLE reader;
```

- надання або заборона прав

```
GRANT MATCH ON GRAPH * TO reader;
DENY WRITE ON GRAPH * TO reader;
```

Також Neo4j підтримує механізм транзакцій з властивостями ACID.

Порівняння SQL та Cypher на рівні мови даних

Незважаючи на принципові відмінності реляційної та графової моделей даних, обидві системи мають повноцінні засоби опису структури, маніпулювання інформацією та управління доступом.

Однак різниця між SQL і Cypher полягає не лише у синтаксисі, а й у способі мислення щодо організації та обробки даних. SQL оперує таблицями та операціями над множинами рядків, тоді як Cypher працює з вузлами, зв’язками та графовими патернами.

Для узагальнення ключових відмінностей наведемо порівняльну характеристику мов SQL та Cypher (таблиця 31).

Таблиця 31 – SQL проти Cypher (Neo4j)

Аспект	SQL	Cypher
Синтаксис	Таблична текстова мова	Декларативна мова графових патернів
Схема	Жорстка	Гнучка
Зберігання	Таблиці	Вузли та зв’язки
Пошук	Вибірка, складні з’єднання	Pattern matching
Структурність	Стовпці визначаються заздалегідь	Властивості додаються динамічно

Висновки про «мову даних» Neo4j

Отже, СУБД Neo4j не має повного аналога SQL як стандартизованої табличної мови, однак Cypher забезпечує повноцінний функціональний еквівалент мови даних:

- декларативний синтаксис;
- підтримку цілісності та безпеки;
- орієнтацію на графову модель.

Ключова відмінність полягає в тому, що структура даних формується природно через створення вузлів і зв’язків, а не через попереднє визначення таблиць.

Через неможливість чітко окреслити «традиційні» групи команд мови даних будемо розглядати Cypher по підмножинах CRUD операцій: створення/модифікація/видалення структури графа (CD), маніпулювання даними (RU), а також управління даними.

CD-операції Cypher (створення та видалення елементів графа)

У реляційних СУБД структура даних визначається за допомогою таблиць та їхніх атрибутів. У Neo4j структура графа формується через мітки вузлів, типи зв'язків, властивості та обмеження. У цьому підрозділі розглянемо основні засоби опису та підтримки структури графа у Cypher.

Як зазначають М. Фаулер і П. Дж. Садаладж [1], у графовій БД створення графа є простою задачею – достатньо створити два вузли та зв'язок між ними. Наприклад:

```
CREATE (:Teacher {id:1, name:"Малахов"});
CREATE (:Publication {id:101, title:"Graph Databases in Education"});
```

Цим двом вузлам надано властивості: вузлу Teacher – «name», вузлу Publication – «title» зі значеннями «Малахов» і «Graph Databases in Education» відповідно.

Оскільки у графі тепер більше за один вузол, можна створити відношення або зв'язок AUTHORED:

```
MATCH (t:Teacher {id:1}),
      (p:Publication {id:101})
CREATE (t)-[:AUTHORED]->(p);
```

Таким чином, створення вузлів і зв'язків одночасно формує структуру графа і наповнює його даними.

Пакетне створення елементів графа

Наведений приклад демонструє «ручне» створення двох вузлів і зв'язку між ними. Проте на практиці, особливо під час ініціалізації БД, дані зазвичай завантажуються ***пакетно***, наприклад: список викладачів, кафедр, тем дипломів тощо. Для цього у Cypher використовується конструкція UNWIND, яка дозволяє перетворити список значень на послідовність рядків, що обробляються у запиті.

Сформуємо для вузлів викладачів невеликий список, який буде джерелом даних:

```
UNWIND [
 {kod: 1, secondName: "Малахов", firstName: "Євгеній", patronymic:
"Валерійович", job: "Зав.каф", hireDate: "2012-09-01T00:00:00Z", salary:
1500.0, deptKod: 3},
 {kod: 4, secondName: "Пенко",   firstName: "Валерій", patronymic:
"Георгійович", job: "Доц.",    hireDate: "2000-09-01T00:00:00Z", salary:
1000.0, deptKod: 3},
 {kod: 3, secondName: "Шпінарева", firstName: "Ірина", patronymic:
"Михайлівна", job: "Доц.",    hireDate: "2000-09-01T00:00:00Z", salary:
1250.0, deptKod: 3}
] AS row
```

```
CREATE (l:Lecturer {kod: row.kod})
SET l.secondName = row.secondName,
   l.firstName  = row.firstName,
   l.patronymic = row.patronymic,
   l.job       = row.job,
   l.hireDate   = date(substring(row.hireDate, 0, 10)),
   l.salary     = row.salary;
```

В цьому прикладі оператор UNWIND [...] AS row розкладає список об'єктів на «рядки» (row), які обробляються послідовно, команда CREATE (l:Lecturer {kod: row.kod}) створює вузол Lecturer з ключовою властивістю kod, оператор SET встановлює інші властивості вузла, а конструкція date(substring(...))перетворює рядок з `YYYY-MM-DD` у тип date.

Недоліком цього прикладу є те, що якщо виконати запит повторно (наприклад, при налагодженні БД), то повторно будуть створені вузли і з'являться їх дублікати.

Щоб зробити завантаження ***ідемпотентним*** (тобто, щоб повторний запуск не створював дублікати), замість CREATE використовується MERGE:

```
UNWIND [
  {kod: 1, secondName: "Малахов", firstName: "Євгеній", patronymic:
"Валерійович", job: "Зав.каф", hireDate: "2012-09-01T00:00:00Z", salary:
1500.0, deptKod: 3},
  {kod: 4, secondName: "Пенко",   firstName: "Валерій", patronymic:
"Георгійович", job: "Доц.",    hireDate: "2000-09-01T00:00:00Z", salary:
1000.0, deptKod: 3}
] AS row
MERGE (l:Lecturer {kod: row.kod})
SET l.secondName = row.secondName,
   l.firstName  = row.firstName,
   l.patronymic = row.patronymic,
   l.job       = row.job,
   l.hireDate   = date(substring(row.hireDate, 0, 10)),
   l.salary     = row.salary;
```

У такому запиті якщо вузол з певним kod існує, він буде знайдений; якщо ні – то створений.

Аналогічно піддокументами документної моделі даних в Neo4j можна створювати ***вкладені структури***. Наприклад, додамо викладачу вкладений об'єкт increments (надбавки):

```
UNWIND [
  {kod: 1, secondName: "Малахов", firstName: "Євгеній", patronymic:
"Валерійович", job: "Зав.каф",
   hireDate: "2012-09-01T00:00:00Z", salary: 1500.0, deptKod: 3,
```

```
  increments: {longevityInc: 300.0, degreeInc: 495.0, titleInc: 375.0,
specialInc: 675.0}},
 {kod: 4, secondName: "Пенко", firstName: "Валерій", patronymic:
"Георгійович", job: "Доц.",
  hireDate: "2000-09-01T00:00:00Z", salary: 1000.0, deptKod: 3,
  increments: {longevityInc: 200.0, degreeInc: 250.0, titleInc: 150.0,
specialInc: 390.0}}
] AS row
MERGE (l:Lecturer {kod: row.kod})
SET l.secondName = row.secondName,
   l.firstName  = row.firstName,
   l.patronymic = row.patronymic,
   l.job       = row.job,
   l.hireDate   = date(substring(row.hireDate, 0, 10)),
   l.salary     = row.salary,
   l.longevityInc = coalesce(row.increments.longevityInc, 0),
   l.degreeInc    = coalesce(row.increments.degreeInc, 0),
   l.titleInc     = coalesce(row.increments.titleInc, 0),
   l.specialInc   = coalesce(row.increments.specialInc, 0);
```

Конструкция row.increments.<елемент_структури> надає доступ до вкладених полів, а функция coalesce(x, Y) працює аналогічно такій SQL-функції, тобто якщо x відсутнє або NULL, буде взято значення константи Y.

Після створення вузлів Lecturer і попереднього створення за аналогічною технікою вузлів Department можна ***зв'язати*** ці вузли, виконавши кроки наступного алгоритму:

1) зберегти l і row між частинами запиту за допомогою оператора WITH, який "передає" вказані змінні в наступну частину запиту;
2) знайти кафедру командою MATCH;
3) створити зв'язок командою MERGE.

```
UNWIND [
 {kod: 1, secondName: "Малахов", firstName: "Євгеній", patronymic:
"Валерійович", job: "Зав.каф",
  hireDate: "2012-09-01T00:00:00Z", salary: 1500.0, deptKod: 3}
] AS row
MERGE (l:Lecturer {kod: row.kod})
SET l.secondName = row.secondName,
   l.firstName  = row.firstName,
   l.patronymic = row.patronymic,
   l.job       = row.job,
   l.hireDate   = date(substring(row.hireDate, 0, 10)),
   l.salary     = row.salary
```

```
WITH l, row
MATCH (d:Department {deptKod: row.deptKod})
MERGE (l)-[:WORKS_IN]->(d);
```

Останнім кроком для повноти відповідності ПрО (Додаток Г) треба пройти по списку тем дипломних робіт і зв'язати їх з викладачем як потенційним керівником, що їх запропонував:

```
UNWIND [
  {kod: 1, secondName: "Малахов", firstName: "Євгеній", patronymic:
"Валерійович", job: "Зав.каф",
   hireDate: "2012-09-01T00:00:00Z", salary: 1500.0, deptKod: 3,
   diplomas: [{dpKod: 9, dpName: "Тема 9"}, {dpKod: 10, dpName: "Тема
10"}]}
] AS row
MERGE (l:Lecturer {kod: row.kod})
SET l.secondName = row.secondName,
    l.firstName  = row.firstName,
    l.patronymic = row.patronymic
WITH l, row
MATCH (d:Department {deptKod: row.deptKod})
MERGE (l)-[:WORKS_IN]->(d)
WITH l, row
UNWIND coalesce(row.diplomas, []) AS dp
MERGE (t:DiplomaTopic {dpKod: dp.dpKod})
SET t.dpName = dp.dpName
MERGE (l)-[:CAN_SUPERVISE]->(t);
```

Обмеження значень властивостей

Як і інших СУБД, на значення властивостей вузлів та зв'язків можуть бути накладені певні обмеження, але, на відміну від SQL, для їх визначення у Cypher відсутні довільні логічні вирази або регулярні вирази.

Neo4j підтримує обмеження цілісності на рівні вузлів і властивостей, до яких належать:

– обмеження унікальності;
– обмеження існування властивості;
– обмеження типів (у версії Neo4j 5+).

Наприклад, наступний запит гарантує, що вузол Lecturer завжди має властивість secondName. (***обмеження існування властивості***):

```
CREATE CONSTRAINT lecturer_name_exists IF NOT EXISTS
FOR (l:Lecturer)
REQUIRE l.secondName IS NOT NULL;
```

У версії Neo4j 5+ з'явилась можливість забезпечити за допомогою

обмежень гарантування ***відповідності типу*** даних. Наприклад:

```
CREATE CONSTRAINT lecturer_salary_type IF NOT EXISTS
FOR (l:Lecturer)
REQUIRE l.salary IS :: FLOAT;
```

А ***обмеження унікальності*** (потенційного ключа) доцільно задати, щоб уникати дублювань вузлів. Наприклад:

```
// Унікальність викладача за kod
CREATE CONSTRAINT lecturer_kod_unique IF NOT EXISTS
FOR (l:Lecturer)
REQUIRE l.kod IS UNIQUE;
```

Через те, що Neo4j безпосередньо ***не підтримує*** визначення та перевірку обмеження на значення властивостей (на кшталт CHECK (salary > 0), регулярні вирази на рівні обмежень та складні логічні перевірки, такі перевірки виконуються на рівні прикладної логіки, через тригери (APOC) або у клієнтському коді.

Індексація

Як було зазначено, Neo4j підтримує механізм ***index-free adjacency***. Проте, щоб забезпечити швидке знаходження вершин в цій СУБД, аналогічно іншим, можна створювати індекси.

Neo4j використовує індекси для:

- швидкого знаходження стартових вузлів;
- оптимізації пошуку за властивостями.

⚠ *Обхід графа (traversal) по зв'язках не потребує додаткових індексів після знаходження стартового вузла.*

В цій СУБД існують два рівні індексації:

1. ***Стандартні індекси*** – для точного та швидкого пошуку вузлів за властивостями.
2. ***Повнотекстові індекси*** – для пошуку текстових значень з урахуванням лексичних особливостей.

Cypher дозволяє створювати наступні варінти індексів.

- ***прості індекси***

```
// індекси для пошуку за прізвищем
CREATE INDEX lecturer_secondName_idx IF NOT EXISTS
FOR (l:Lecturer)
ON (l.secondName);
```

- ***складені індекси***

```
CREATE INDEX lecturer_name_job_idx IF NOT EXISTS
FOR (l:Lecturer)
ON (l.secondName, l.job);
```

За замовчуванням Neo4j використовує ***B-tree індекси*** для пошуку вузлів за значеннями властивостей. Взагалі B-tree індекси широко використовуються в різних СУБД, зокрема в PostgreSQL та інших реляційних системах [14].

B-tree (B-дерево) – це збалансована багатошляхова структура даних, яка забезпечує:

– логарифмічну складність пошуку;
– ефективну вставку та видалення;
– збереження впорядкованості значень.

Такі індекси оптимізують операції точного зіставлення значень, наприклад:

```
MATCH (l:Lecturer {kod: 1})
RETURN l;
// або
MATCH (l:Lecturer)
WHERE l.secondName = "Малахов"
RETURN l;
```

Індекс використовується для швидкого знаходження **початкового вузла**, після чого обхід зв'язків виконується без додаткових індексів (механізм index-free adjacency).

– ***Повнотекстові індекси***

Окрім звичайних b-tree індексів, Neo4j підтримує ***повнотекстові індекси (full-text indexes)***, призначені для пошуку текстової інформації за словами, фрагментами слів та мовними правилами. На відміну від стандартного індексу, який забезпечує швидке точне зіставлення значень властивостей, повнотекстовий індекс дозволяє:

– виконувати пошук за частиною слова;
– враховувати регістр;
– використовувати логічні оператори (AND, OR);
– здійснювати ранжування результатів за релевантністю.

Зокрема, у БД «Навчальний процес» (Додаток Г) така індексація може бути корисно для:

– пошуку викладачів за прізвищем або його частиною;
– пошуку тем дипломів за ключовими словами;
– реалізації простого «пошуку по сайту кафедри».

Наприклад, звичайний індекс

```
MATCH (l:Lecturer {secondName:"Малахов"})
RETURN l;
```

дозволяє знайти певного викладача, але не дозволяє ефективно виконати запит: знайти всіх викладачів, у прізвищі яких міститься фрагмент "мал". Для цього використовується повнотекстовий індекс.

Для ***створення повнотекстового індексу*** необхідно виконати запит, наприклад

```
CALL db.index.fulltext.createNodeIndex(
  "lecturerFullText",
  ["Lecturer"],
  ["secondName", "firstName"]
);
```

де "lecturerFullText" – назва індексу;
["Lecturer"] – мітки вузлів;
["secondName", "firstName"] – текстові властивості, що індексуються.

Пошук за таким індексом здійснюється наступним чином:

```
CALL db.index.fulltext.queryNodes(
  "lecturerFullText",
  "Малахов"
)
YIELD node
RETURN node;
```

Варто зауважити, що процедура повнотекстового пошуку за замовчуванням повертає результати, вже відсортовані за релевантністю. Проте без використання YIELD score цей коефіцієнт залишається прихованим від користувача.

Завдяки створеному коефіцієнту релевантності score, результати пошуку автоматично ранжуються, і можна виконувати складні запити типу:

```
CALL db.index.fulltext.queryNodes(
  "lecturerFullText",
  "Малахов OR Пенко")
YIELD node, score
RETURN node.secondName, node.firstName, score
ORDER BY score DESC;
```

Технічні особливості повнотекстових індексів полягають в тому, що вони:

- побудовані на базі Apache Lucene – це потужна, високопродуктивна бібліотека з відкритим вихідним кодом для повнотекстового пошуку, написана на Java;
- використовують аналіз тексту (tokenization);
- підтримують мовні аналізатори;
- не застосовуються автоматично через MATCH, а викликаються процедурно через CALL.

⚠ *Важливо розуміти, що повнотекстовий індекс (таблиця 32):*

- *не замінює звичайні b-tree індекси;*
- *використовується для текстового пошуку, а не для точного зіставлення значень.*

Таблиця 32 – Різниця між B-tree та повнотекстовим індексами

Ознака	B-tree індекс	Повнотекстовий індекс
Тип пошуку	Точне співпадіння	Пошук за словами та фрагментами
Ранжування	Ні	Так (score)
Використання через MATCH	Так	Ні (через CALL)
Основа	Структура B-дерева	Apache Lucene

Наявність таких варіантів індексації розширює можливості графової СУБД, дозволяючи реалізувати як структурний аналіз зв’язків, так і текстовий пошук у межах єдиної системи.

Видалення елементів графа

Видалення вузлів графової БД – це фактично видалення екземплярів сутності, тобто значень. Тому просте видалення значень в реляційній чи документній БД в графовій призводить до модифікація структури БД. Наприклад:

```
MATCH (l:Lecturer {kod: 4})
DELETE l;
```

Така операція згенерує помилку, якщо існують зв’язки. Це пов’язано з тим, що Neo4j гарантує цілісність графа і не дозволяє видалити вузол, який має зв’язки. Якщо потрібно видалити вузол з урахуванням наявності зв’язків, то таку операцію необхідно виконувати з їх «розірванням», наприклад:

```
MATCH (l:Lecturer {kod: 4})
DETACH DELETE l;
```

Теж саме стосується і ***видалення зв’язків***. Наприклад:

```
MATCH (l:Lecturer {kod:1})-[r:CAN_SUPERVISE]->(t:DiplomaTopic {dpKod:
9})
DELETE r;
```

Наступний приклад виконує масове видалення вузлів:

```
MATCH (n:Lecturer)
DETACH DELETE n;
```

⚠ *Масове видалення без фільтра може призвести до втрати значної частини графа.*

Питання для самоперевірки

(до вступних підрозділів розділу «Мова Cypher»
та розділу «CD-операції Cypher (створення та видалення елементів графа)»)

1. У чому полягають основні особливості декларативного підходу мови Cypher?
2. Чим синтаксис Cypher концептуально відрізняється від SQL?
3. Яким чином у Cypher описуються вузли, зв’язки та шляхи?

4. Як у Cypher задаються мітки вузлів та властивості елементів графа?
5. Для чого у Cypher використовуються шаблони графа (graph patterns)?
6. Які операції Cypher можна умовно віднести до CD-операцій?
7. У чому полягають особливості створення вузлів та зв’язків у Cypher?
8. Як забезпечити ідемпотентне введення вузлів графа?
9. Як у Cypher створюються зв’язки між вузлами?
10. В яких випадках при створенні зв’язків у Cypher попередньо виконують пошук вузлів?
11. У чому полягають особливості видалення вузлів та зв’язків у Neo4j?
12. Для чого використовується команда DETACH DELETE?
13. У чому полягають особливості роботи Cypher із напрямленими зв’язками?
14. Як у Cypher можуть задаватись властивості під час створення вузлів та зв’язків?
15. Яким чином у Neo4j можуть моделюватись вкладені або ієрархічні структури даних?
16. Які типи обмежень (constraints) підтримуються у Neo4j?
17. Для чого у Neo4j використовуються обмеження унікальності та існування властивостей?
18. Для чого використовуються індекси у Neo4j?
19. Чим B-tree індекси відрізняються від full-text indexes у Neo4j?
20. У чому полягає особливість повернення результатів при використанні повнотекстових індексів?

Завдання для самостійної роботи

Написати запити і, використовуючи інструментарій описаний в розділі «Інструменти для роботи з Neo4j», виконати їх для сворення елементів графу бази даних ПрО, обраної в розділі «Завдання для самостійної роботи» (с. 132): вузлів та зв’зків з відповідними властивостями, обмежень на властивості, індексів та, за необхідності, похідних зв’язків.

Демонстраційний приклад

Завдання

Написати Cypher-запити для створення елементів графа бази даних, визначених у розділі «Демонстраційний приклад» (с. 134) для ПрО ***«Портфоліо студентських проєктів»***, а саме:

1) створити обмеження унікальності для основних типів вузлів;
2) створити індекси для полів, що можуть використовуватись у фільтрації;
3) створити вузли типів Student, Project, Technology, Tag, Supervisor;
4) створити зв’язки між вузлами відповідно до графового шаблону предметної області;
5) за необхідності створити похідні зв’язки між подібними проєктами.

Розв’язання

Послідовність створення елементів та обмежень графу, наведена в Завданні, не є випадковою: саме така послідовність (обмеження — індекси — вузли —

зв’язки) є коректною з інженерної точки зору і типовою для Neo4j/Cypher.

Обумовлено це наступними міркуваннями:

– обмеження унікальності починають контролювати дані одразу під час завантаження;
– Neo4j автоматично використовує створені індекси при виконанні MERGE та MATCH, тому масове завантаження зазвичай працює швидше;
– якщо дані мають дублікати, помилка виявиться відразу, а не після наповнення графа.

Я к не дивно, але це відповідає логіці «спочатку схема та визначення обмежень — потім введення даних», притаманній DDL SQL.

Очищення демонстраційного графа

```
// Рекомендується при відпрацюванні навчального прикладу
// перед повторним запуском скрипту очистити поточний граф
MATCH (n)
DETACH DELETE n;
```

1) Створення обмежень унікальності

```
// Унікальний ідентифікатор студента
CREATE CONSTRAINT student_id_unique IF NOT EXISTS
FOR (s:Student)
REQUIRE s.student_id IS UNIQUE;

// Унікальний ідентифікатор проєкту
CREATE CONSTRAINT project_id_unique IF NOT EXISTS
FOR (p:Project)
REQUIRE p.project_id IS UNIQUE;

// Унікальний ідентифікатор керівника
CREATE CONSTRAINT supervisor_id_unique IF NOT EXISTS
FOR (v:Supervisor)
REQUIRE v.supervisor_id IS UNIQUE;

// Унікальна назва технології
CREATE CONSTRAINT technology_name_unique IF NOT EXISTS
FOR (t:Technology)
REQUIRE t.name IS UNIQUE;

// Унікальна назва тегу
CREATE CONSTRAINT tag_name_unique IF NOT EXISTS
FOR (g:Tag)
REQUIRE g.name IS UNIQUE;
```

2) Створення додаткових індексів

```
// Індекс для фільтрації студентів за групою
CREATE INDEX student_group_index IF NOT EXISTS
FOR (s:Student)
ON (s.group);

// Індекс для фільтрації студентів за спеціальністю
CREATE INDEX student_speciality_index IF NOT EXISTS
FOR (s:Student)
ON (s.speciality);

// Індекс для фільтрації проєктів за типом
CREATE INDEX project_type_index IF NOT EXISTS
FOR (p:Project)
ON (p.project_type);
```

3) Створення вузлів типу Student

```
UNWIND [              // розкладання списку об’єктів студент на «рядки»
(row)
 {
  student_id: 1,
  second_name: "Мікулінська",
  first_name: "Марія",
  group: "ІС-31",
  speciality: "F6 Інформаційні системи та технології",
  study_year: 3
 },
 {
  student_id: 2,
  second_name: "Лисенко",
  first_name: "Максим",
  group: "КН-41",
  speciality: "F3 Комп’ютерні науки",
  study_year: 4
 },
 {
  student_id: 3,
  second_name: "Філатова",
  first_name: "Тетяна",
  group: "ІС-31",
  speciality: "F6 Інформаційні системи та технології",
  study_year: 3
```

```
  }
] AS row
MERGE (s:Student {student_id: row.student_id})      // ідемпотентне зава
нтаження
SET s += row;           // '+=' оператор масового оновлення властивостей
                        // див. розділ «Оновлення властивостей» Cypher
(с. 162)
```

Створення вузлів типу Project

```
UNWIND [
 {
  project_id: 101,
  title: "Система управління студентськими проєктами",
  description: "Вебзастосунок для обліку та аналізу студентських проєк
тів",
  project_type: "web-application",
  created_at: date("2026-02-01"),
  updated_at: date("2026-03-05")
 },
 {
  project_id: 102,
  title: "Прогнозування успішності студентів",
  description: "ML-проєкт для аналізу навчальних результатів",
  project_type: "machine-learning",
  created_at: date("2026-01-15"),
  updated_at: date("2026-02-12")
 },
 {
  project_id: 103,
  title: "Каталог студентських IT-проєктів",
  description: "Вебкаталог для пошуку проєктів за технологіями та тем
атикою",
  project_type: "web-application",
  created_at: date("2026-02-20"),
  updated_at: date("2026-03-18")
 }
] AS row
MERGE (p:Project {project_id: row.project_id})
SET p += row;
```

Створення вузлів типу Technology

```
UNWIND [
 {name: "MongoDB", category: "database"},
```

```
  {name: "Neo4j", category: "graph-database"},
  {name: "Node.js", category: "backend"},
  {name: "React", category: "frontend"},
  {name: "Python", category: "programming-language"},
  {name: "Scikit-learn", category: "machine-learning"}
] AS row
MERGE (t:Technology {name: row.name})
SET t += row;
```

Створення вузлів типу Tag

```
UNWIND [
  {name: "web", category: "software-development"},
  {name: "machine-learning", category: "artificial-intelligence"},
  {name: "database", category: "data-management"}
] AS row
MERGE (g:Tag {name: row.name})
SET g += row;
```

Створення вузлів типу Supervisor

```
UNWIND [
  {
    supervisor_id: 1,
    second_name: "Шпінарева",
    first_name: "Ірина",
    academic_position: "доцент"
  },
  {
    supervisor_id: 2,
    second_name: "Волков",
    first_name: "Віктор",
    academic_position: "професор"
  }
] AS row
MERGE (v:Supervisor {supervisor_id: row.supervisor_id})
SET v += row;
```

4) Створення зв'язків PARTICIPATED_IN

```
UNWIND [
  {student_id: 1, project_id: 101, role: "team-lead", joined_at: date("2026-01-19")},
  {student_id: 2, project_id: 101, role: "backend-developer", joined_at: date("2026-01-19")},
  {student_id: 1, project_id: 102, role: "data-analyst", joined_at: date("2026-
```

```
02-23")},
 {student_id: 1, project_id: 103, role: "team-lead", joined_at: date("2026-04-
10") },
 {student_id: 3, project_id: 103, role: "frontend-developer", joined_at: date
("2026-05-09") }
] AS row
MATCH (s:Student {student_id: row.student_id})
MATCH (p:Project {project_id: row.project_id})
MERGE (s)-[r:PARTICIPATED_IN]->(p)
SET r.role = row.role ,
r.joined_at = row.joined_at;
```

Створення зв'язків USES

```
UNWIND [
 {project_id: 101, technology: "MongoDB"},
 {project_id: 101, technology: "Node.js"},
 {project_id: 101, technology: "React"},
 {project_id: 102, technology: "Python"},
 {project_id: 102, technology: "MongoDB"},
 {project_id: 102, technology: "Scikit-learn"},
 {project_id: 103, technology: "Neo4j"},
 {project_id: 103, technology: "React"}
] AS row
MATCH (p:Project {project_id: row.project_id})
MATCH (t:Technology {name: row.technology})
MERGE (p)-[:USES]->(t);
```

Створення зв'язків HAS_TAG

```
UNWIND [
 {project_id: 101, tag: "web"},
 {project_id: 101, tag: "database"},
 {project_id: 102, tag: "machine-learning"},
 {project_id: 102, tag: "database"},
 {project_id: 103, tag: "web"},
 {project_id: 103, tag: "database"}
] AS row
MATCH (p:Project {project_id: row.project_id})
MATCH (g:Tag {name: row.tag})
MERGE (p)-[:HAS_TAG]->(g);
```

Створення зв'язків SUPERVISES

```
UNWIND [
 {supervisor_id: 1, project_id: 101},
```

```
  {supervisor_id: 2, project_id: 102},
  {supervisor_id: 1, project_id: 103}
] AS row
MATCH (v:Supervisor {supervisor_id: row.supervisor_id})
MATCH (p:Project {project_id: row.project_id})
MERGE (v)-[:SUPERVISES]->(p);
```

5) Створення похідних зв'язків SIMILAR_TO

```
// Проєкти вважаються подібними,
// якщо вони мають спільну технологію або спільний тематичний тег
MATCH (p1:Project)-[:USES|HAS_TAG]->(x)<-[:USES|HAS_TAG]-(p2:Projec
t)
WHERE p1.project_id < p2.project_id
MERGE (p1)-[r:SIMILAR_TO]->(p2)
SET r.reason = "common technology or tag";
```

RU-операції Cypher (маніпулювання даними: читання та оновлення)

Після створення структури графа та визначення обмежень цілісності виникає потреба у виконанні запитів до бази даних – пошуку, оновлення та аналізу інформації. Для цього у Cypher використовуються наступні команди і оператори маніпулювання даними [13]:

MATCH, WHERE, RETURN, SET, DELETE, UNWIND та інші.

На відміну від SQL, де запит формулюється як операція над таблицями, у Cypher користувач описує **графовий шаблон (pattern)** – структуру вузлів та зв'язків, яку система повинна знайти у графі (таблиця 33).

*Таблиця 33 – Основні елементи синтаксису **патерну***

Елемент	Позначення
вузол	( )
зв'язок	[ ]
напрямок	-> або <-
властивості	{ }

Пошук вузлів

Команда **MATCH** є центральною операцією Cypher [13, …/clauses/match/]. Вона задає шаблон (pattern) вузлів і зв'язків, відповідність якому система повинна знайти у графі. Команда має наступний загальний синтаксис:

```
MATCH (node_label {property:value})
RETURN node
```

Наприклад, знайти викладача з кодом 1:

```
MATCH (l:Lecturer {kod:1})
RETURN l
```

де l – змінна вузла,
Lecturer – мітка вузла
kod – властивість

Для швидкого знаходження таких вузлів Neo4j використовує індекси.

Нагадуємо, що головною особливістю графових СУБД є ефективна робота зі зв'язками. У Cypher зв'язок описується так:

```
(node)-[relationship]->(node)
```

На основі такого опису Neo4j виконує пошук зв'язаних вузлів. Наприклад, знайдемо викладачів та кафедри, на яких вони працюють:

```
MATCH (l:Lecturer)-[:WORKS_IN]->(d:Department)
RETURN l.secondName, d.deptName
```

Фактично, цей запит відповідає SQL-запиту зі з'єднанням (JOIN), але формулюється як ***патерн графа***.

Фільтрація результатів

Так само, аналогічно SQL-реалізації реляційної операції вибірки, у Cypher існує оператор **WHERE** [13, …/clauses/where/], який використовується для, так званої, фільтрації, тобто, для обмеження результатів пошуку.

Для визначення такого обмеження в Cypher використовуються оператори порівняння і логічні оператори. Наприклад:

```
MATCH (l:Lecturer)
WHERE l.salary > 1200
RETURN l.secondName, l.salary
```

або

```
MATCH (l:Lecturer)
WHERE l.salary > 1200 AND l.job = "Доц."
RETURN l
```

Повернення результатів

У всіх наведених вище прикладах пошуку даних фігурує оператор **RETURN**, який визначає перелік даних, що повертаються запитом.

Проте оператор RETURN у Cypher набагато потужніший, ніж просто вибір колонок.

В *першому прикладі* фільтрації (див. вище) оператор

```
RETURN l.secondName, l.salary
```

повертає ***конкретні властивості вузла***.

Результат має вигляд пласкої таблиці або списку кортежів. Це економно з точки зору трафіку, тому що застосунок забирає лише ті дані, які потрібні для звіту чи відображення:

```
{"l.secondName": "Малахов", "l.salary": 1500}.
```

В *другому прикладі* оператор

```
RETURN l
```

містить тільки мітку вузла і ***повертає весь об'єкт вузла*** повністю. Це включає в себе усі внутрішні властивості (поля) та метадані (внутрішній ID Neo4j, мітки вузла, наприклад: Lecturer).

Це зручно, коли робота йде з клієнтською бібліотекою (наприклад, Python або JS) і необхідно отримати повноцінний об'єкт для подальшої логіки.

Більш того, Cypher дозволяє вставляти у RETURN ***математику чи умови*** CASE. Наприклад:

```
RETURN l.secondName,
    CASE
```

```
    WHEN l.salary > 2000 THEN 'High'
    ELSE 'Standard'
  END AS SalaryLevel
```

Для краси чи зручності фронтенду Cypher дозволяє задавати *аліаси вихідних стовців* в результаті. Наприклад:

```
RETURN l.secondName AS LastName, l.salary * 1.1 AS SalaryWithBonus
```

Для виключення дублікати із результата, як і в SQL використовується оператор **DISTINCT**:

```
RETURN DISTINCT l.job
```

Агрегація та групування

Аналогічно більшості СУБД, Cypher підтримує [13, …/functions/aggregating/] наступні агрегатні функції (таблиця 34):

Таблиця 34 – Агрегатні функції Cypher

Функція	Опис
COUNT	підрахунок кількості вузлів
SUM	сума значень властивості
AVG	середнє значення властивості
MIN	мінімальне значення властивості
MAX	максимальне значення властивості
COLLECT	формування списку значень властивості

Наприклад, для підрахунку кількості викладачів на кафедрах потрібен наступний запит:

```
MATCH (l:Lecturer)-[:WORKS_IN]->(d:Department)
RETURN d.deptName, COUNT(l)
```

Робота з колекціями (формування вихідних списків)

У Cypher функція collect() працює як агрегатор (аналог GROUP BY в SQL, але більш гнучкий). Результат буде залежати від того, чи є у RETURN інші поля, окрім вказаних у «колекції».

Ось як це виглядає на практиці:

RETURN містить тільки колекції

Якщо ви пишете

```
RETURN collect(s.secondName) AS allNames
```

база даних візьме всі прізвища зі знайдених вузлів і «схлопне» в один масив (список).

Результат у JSON-виді буде такий:

```
{
  "allNames": ["Бровков", "Арсірій", "Любченко", "Філатова",
                "Журан", "Мікулінська", "Блажко", "Філиппова"]
}
```

Проте в інтерфейсі Neo4j Browser це буде виглядати як один рядок із квадратними дужками.

Групування (RETURN містить інші поля)

Якщо ж до RETURN додається поле, яке не обернене в collect(), Cypher автоматично згрупує дані, що стусуються нього.

Наприклад:

```
MATCH (l:Lecturer)
RETURN l.job AS Position, collect(l.firstName + " " + l.secondName) AS Staf
f
```

Результат буде таблицею:

Position	Staff
"Зав.каф"	["Євгеній Малахов"]
"Доц."	["Алла Рачинська", " Ірина Шпінарева", " Валерій Пенко"]
"Проф."	["Юрій Гунченко"]

Формування колекцій об'єктів

Більш того, в колекції можна збирати не лише рядки, а й цілі об'єкти (мапи):

```
MATCH (l:Lecturer)
WHERE l.job = "Доц."
RETURN collect({ name: l.secondName, money: l.salary }) AS staffDetails
```

Результат цього запиту виглядатиме так:

```
[
 { money: 1100.0, name: "Рачинська" },
 { money: 1250.0, name: "Шпінерева" },
 { money: 1000.0, name: "Пенко" }
]
```

Використовуючи такі можливості, можна скласти запит, який поверне список усіх кафедр і відразу вкладений список прізвищ усіх викладачів, які на них працюють, тобто згрупує викладачів по їхніх кафедрах у зручні списки.

Для цього треба, щоб між вузлами Lecturer та Department існував зв'язок [:WORKS_IN]:

```
MATCH (d:Department)<-[:WORKS_IN]-(l:Lecturer)
RETURN d.deptName AS Department,
    collect(l.secondName) AS Staff,
    count(l) AS TotalCount
ORDER BY TotalCount DESC
```

MATCH знаходить пари "Підрозділ – Викладач". Оскільки d.deptName не обернено у функцію, Cypher автоматично робить його "ключом" групування. Усі прізвища викладачів, що належать до конкретного d.deptName, упаковуються collect(l.secondName) в один масив. Паралельно count(l) рахує кількість людей у цьому списку:

Department	Staff	TotalCount
"МЗКС"	["Малахов", "Шпінерева", "Пенко"]	3
"МАІТ"	["Рачинська"]	1
"КСТ"	["Гунченко"]	1

Особливості роботи зі списками:

1. **Дублікати:** За замовчуванням collect() зберігає всі значення. Якщо два викладачі мають однакове прізвище, у списку будуть два таких прізвища. Щоб уникнути цього, використовується оператор DISTINCT:

```
RETURN collect(DISTINCT l.secondName)
```

2. **Порожні значення:** Якщо MATCH не знайшов відповідних вузлів, collect() поверне порожній список [], а не null. Це дуже зручно для обробки даних у коді (не потрібно робити перевірки на "null pointer").
3. **Порядок:** Порядок елементів у списку зазвичай відповідає порядку обходу вузлів базою даних, якщо ви не вказали WITH ... ORDER BY.

Обробка відсутніх зв'язків (збереження цілісності результуючого списку)

Оператор OPTIONAL MATCH [13, …/clauses/optional-match/] працює аналогічно LEFT JOIN у SQL. Тобто, якщо зв'язок відсутній, результат все одно повертається. Наприклад, отримати список студентів, дипломними роботами яких керує конкретний викладач:

```
MATCH (l:Lecturer)
OPTIONAL MATCH (l)<-[:SUPERVISED_BY]-(s:Student)
RETURN l.secondName AS Supervisor, collect(s.secondName) AS StudentsL
ist
```

Результат запиту буде такий:

Supervisor	StudentsList
"Малахов"	["Бровков"]
"Рачинська"	[]
"Шпінарева"	[]
"Пенко"	["Любченко", "Арсірій"]
"Гунченко"	["Журан"]

Інший приклад: необхідно оцінити "наукову" ефективність викладачів: скільки у них публікацій і яка частка статей у наукометричних базах.

Для цього знадобиться структура, де викладач (Lecturer) пов'язаний з публікаціями (Publication), а публікації мають властивість або мітку, що вказує на наукометричну базу (Scopus або WoS). Отже, треба додати відповідну інформацію про властивості публікацій:

```
// Стаття у фаховому журналі (Категорія Б)
MATCH (p:Publication {pubId: "doi:10.15276/imms.v11.no4.287"})
SET p.category = "Б";

// Конференція, що індексується у Scopus
MATCH (p:Publication)
WHERE p. pubId CONTAINS "CSIT"
SET p.category = "Scopus";

// Стаття у Scopus (Q3)
MATCH (p:Publication {pubId: "doi:10.15587/1729-4061.2024.313970"})
SET p.category = "Scopus";

// Стаття у Web of Science (WoS)
MATCH (p:Publication {pubId: "doi:10.15276/aait.09.2026.09"})
SET p.category = "WoS";
```

Наступний запит використовує існуючий зв'язок [:AUTHORED] та нову властивість isIndexed для оцінки наукової ефективності викладачів:

```
MATCH (l:Lecturer)
OPTIONAL MATCH (l)-[:AUTHORED]->(p:Publication)
WITH l,
   count(p) AS totalPubs,
   count(CASE WHEN p.category IN ["Scopus", "WoS"] THEN 1 END) AS ind
exedPubs

RETURN l.secondName AS Lecturer,
    totalPubs,
```

```
    indexedPubs,
    CASE
      WHEN totalPubs > 0 THEN round(toFloat(indexedPubs) / totalPubs *
100, 2)
      ELSE 0
    END AS IndexingRatePercentage
ORDER BY totalPubs DESC;
```

Якщо ж цікавим буде не тільки факт наявності такої публікації, а й, наприклад, наявність гранту чи проекту, в рамках якого подавали (було профінансовано) публікацію, то в OPTIONAL MATCH треба використати «ланцюжок» зв'язків.

Але попередньо треба модифікувати граф і додати інформацію про гранти/проекти у вигляді відповідних вузлів:

```
// Створюємо вузли грантів (яких немає в Додатку В)
MERGE (g1:Grant {id: "G-2024-01", source: "Horizon Europe", amount: 50000})
MERGE (g2:Grant {id: "G-2024-02", source: "МОН України", amount: 15000});

// Зв'язуємо публікації з грантами (ланцюжок для OPTIONAL MATCH)
MATCH (p:Publication {pubId: "doi:10.1109/CSIT65290.2024.10982587"})
MATCH (g:Grant {id: "G-2024-01"})
MERGE (p)-[:FUNDED_BY]->(g);

MATCH (p:Publication {pubId: "doi:10.15587/1729-4061.2024.313970"})
MATCH (g:Grant {id: "G-2024-02"})
MERGE (p)-[:FUNDED_BY]->(g);
```

Коли будується ланцюжок зв'язків в OPTIONAL MATCH, важливо розуміти: цей оператор працює за принципом *«все або нічого» для конкретного шаблону* . Тобто, якщо описується довгий ланцюжок зв'язків в одному `OPTIONAL MATCH`, і хоча б одна ланка в цьому ланцюжку відсутня – весь ланцюжок перетворюється на **`null`**. Наприклад, для ланцюжка Викладач -> Публікація -> Грант:

```
MATCH (l:Lecturer {secondName: "Рачинська"})
OPTIONAL MATCH (l)-[:AUTHORED]->(p:Publication)-[:FUNDED_BY]->(g:Grant)
RETURN l.secondName AS Lecturer, p.pubId, g.amount
```

Якщо певний викладач має статті, але ці статті не фінансувались грантами, то і p.pubId, і g.amount повернуть null. Хоча публікація існує, Cypher не зміг пройти шаблон до кінця, тому вся частина після OPTIONAL MATCH вважається невиконаною.

Якщо все ж таки треба, щоб на кожному етапі не губилися дані (побачити пулікацію, навіть якщо для неї немає гранту/проекту), то у нагоді буде використання «м'якого ланцюжка» (розбитого на частини) OPTIONAL MATCH, тобто декілька операторів.

Наступний запит з послідовним OPTIONAL MATCH використовує існуючий зв'язок викладача з публікацією [:AUTHORED] і доданий зв'язок [:FUNDED_BY] та розв'язує початкову задачу:

```
MATCH (l:Lecturer)

// Перша ланка: викладач -> публікація [cite: 57]
OPTIONAL MATCH (l)-[:AUTHORED]->(p:Publication)

// Друга ланка: публікація -> грант
OPTIONAL MATCH (p)-[:FUNDED_BY]->(g:Grant)

WITH l,
   count(p) AS totalPubs,                    // Загальна кількість публікацій
   // Далі підрахунок статтей за категоріями
   count(CASE WHEN p.category IN ["Scopus", "WoS"] THEN 1 END)
                                                    AS highRankPubs,
   collect(DISTINCT g.source) AS grantSources

RETURN l.secondName AS Lecturer,
     totalPubs,
     highRankPubs,
     // Використовуємо CASE для обробки порожнього списку collect()
     CASE
       WHEN size(grantSources) > 0 THEN grantSources
       ELSE ["Без грантового фінансування"]
     END AS FundingSource,
     CASE
       WHEN totalPubs > 0 THEN round(toFloat(highRankPubs) / totalPubs * 100, 2)
       ELSE 0
     END AS ScienceIndexPercentage
ORDER BY highRankPubs DESC;
```

В результаті будуть отримані всі викладачі, навіть якщо в них немає грантів чи публікацій, а рейтинг буде підрахований тільки по публікаціям в наукометричних базах:

Отже, один довгий OPTIONAL MATCH слід використовувати, якщо потрібний тільки повний шлях. Якщо ж треба «збирати» інформацію по крихтах, зберігаючи проміжні результати, використовуються кілька послідовних OPTIONAL MATCH.

Оновлення властивостей

Для ***зміни значень властивостей*** в Cypher, як і в SQL використовується оператор **SET** [13, …/clauses/set/]. Наприклад:

```
MATCH (l:Lecturer {kod:1})
SET l.salary = 1600
```

Цей же оператор дозволяє додавати нові властивості вузлам та зв'язкам:

```
MATCH (l:Lecturer {kod:1})
SET l.office = "305"
```

Додатковий оператор '+=' перетворює SET на *оператор масового оновлення властивостей* (***property map update***), який працює дещо інакше, ніж «звичайний» SET.

На відміну від «звичайного», наприклад, такого

```
MATCH (g:Grant {id: "G-2024-02"})
SET g.amount: 25000,
    g.FinDate: date("2030-12-31")
```

«розширений» оператор, фактично, є аналогом операції

```
node.properties ← node.properties ∪ map
```

тобто виконує злиття map із властивостями вузла:

```
MATCH (g:Grant {id: "G-2024-02"})
SET g += {amount: 25000, FinDate: date("2030-12-31")}
RETURN g
```

Такий варіант SET виконує додавання кількох властивостей відразу та/або оновлення існуючих властивостей. Його корисно застосовувати при оновленні декількох властивостей, оновленні з JSON чи оновленні з UNWIND.

Особливі графові шаблони

Обхід зв'язків графа (Traversal)

Traversal (обхід) – це процес навігації по графу, при якому алгоритм переміщається від одного вузла до іншого через зв'язки, що з'єднують їх, згідно з заданими правилами, фільтрами і напрямом [10].

Ключові аспекти цього процесу: *формування шляху*, *обмеження глибини*, *оптимізація* (таблиця 35).

Таблиця 35 – Ключові аспекти процесу обходу

Аспект	Зміст
Контекст (Path)	Обхід формує шлях – впорядковану послідовність вузлів та зв'язків
Глибина (Hops)	Обхід може бути обмежений кількістю «стрибків» (хопів) або продовжуватись доти, доки не буде досягнуто цільового вузла (*пошук найкоротшого шляху*)
Декларативність	В Cypher не пишеться цикл for, а просто описується шаблон (pattern), наприклад: (a)-[:REL1]->(b)-[:REL2]->(c). База даних сама оптимізує фізичний обхід

Traversal – це послідовний обхід зв'язків графа. У Neo4j такий обхід виконується дуже ефективно завдяки механізму ***index-free adjacency***, при якому кожен вузол безпосередньо зберігає посилання на свої зв'язки. Це дозволяє переходити між вузлами без додаткових індексних операцій.

Наприклад, знайдемо колег викладача «Малахов»:

```
MATCH (l:Lecturer {secondName:"Малахов"})-[:WORKS_IN]->(d)<-[:WORKS_IN]-
                                                        (colleague)
RETURN colleague.secondName
```

Цей запит знаходить усіх викладачів, які працюють на тій самій кафедрі.

Або більш складний приклад: знайти всі «онукові» підрозділи для департаменту (тобто, через два рівні підпорядкування):

```
MATCH (root:Department {deptName: "ІМФІТ"})-[:SUBDIVISION_OF*2]-(sub)
RETURN sub.deptName
```

де *2 – це інструкція обходу на глибину рівно за два кроки.

Neo4j підтримує дві ***стратегії обходу графа***: у ширину (BFS) і в глибину (DFS). Різниця між цими двома стратегіями – це, по суті, вибір між «скануванням шарів» та «зануренням углиб». У Cypher Neo4j найчастіше використовує ***DFS (Depth-First Search)*** для пошуку шляхів, оскільки це заощаджує пам'ять, але для розуміння структури важливо знати обидва підходи.

1. BFS (Breadth-First Search) – Пошук у ширину

Алгоритм відвідує всіх "сусідів" поточного вузла, перш ніж переходити до "онуків". Наприклад, для БД lectures:

Порядок обходу	Спочатку дивимося всі відділи 1-го рівня (прямі підрозділи), потім усі відділи 2-го рівня тощо.
Задача	Якщо потрібно знайти найкоротший шлях (наприклад, через скільки рукостискань персона знайома з людиною).
Результат застосування для навчальної БД	Спершу знаходимо всі факультети (ФІТ, ПМ), і тільки потім підемо дивитися кафедри всередині них.

2. DFS (Depth-First Search) – Пошук в глибину

Алгоритм йде по одній гілці до кінця (листа), а потім повертається назад (backtracking), щоб перевірити інші гілки. Наприклад, для БД lectures:

Порядок обходу	Спочатку дивимося гілку від кореня до листа, а потім переходимо до наступної гілки. Це природний спосіб побудови / обходу дерева папок або змісту книги.
Задача	Якщо потрібно обійти весь граф цілком або знайти рішення, приховане глибоко в одній із гілок.
Результат застосування для навчальної БД	Заходимо в гілку ІМФІТ → ФІТ → МЗКС. Коли гілка закінчилася, повертаємось до ФІТ та йдемо до КСТ.

Для визначення контексту використання співставимо ці стратегії (таблиця 36).

Таблиця 36 – Порівняння стратегій обходу графа у контексті Cypher

Характеристика	BFS (Ширина)	DFS (Глибина)
Пріоритет	Найближчі вузли	Найдальші вузли
Пам'ять	Потребує багато пам'яті (зберігає весь "фронтир")	Економніше (зберігає лише поточний шлях)
Типовий запит	shortestPath((a)-[*..10]-(b))	(a)-[:SUBDIVISION_OF*]->(b)

Наприклад, якщо виконати такий запит

```
MATCH (root:Department {deptName: "ІМФІТ"})<-[:SUBDIVISION_OF*]-(sub)
RETURN sub.deptName
```

то Neo4j за замовчуванням застосовуватиме ***DFS***: він спочатку буде рухатись вглиб гілки "ФІТ", витягне всі її кафедри, а потім повернеться і почне обробляти гілку "ПМ".

Шляхи змінної довжини

Cypher пітримує концепцію ***Variable Length Paths*** (шляхи змінної довжини), яка є однією з найпотужніших особливостей графових баз даних [13, …/patterns/], тому що в звичайному SQL для такого результату довелося б писати купу складних JOIN або рекурсивних CTE.

Ідея концепції VLP в тому, що ми не просто шукаємо сусіда, ми просимо базу «пройтися» ланцюжком зв'язків.

Для цього в опис зв'язку додається зірочка (*) та діапазон (n..m) і, тим самим, задаються правила «стрибків» (hops): зірочка вказує, що зв'язок може повторюватися, n – мінімальна кількість зв'язків (безпосередній сусід), m – максимальна кількість зв'язків у ланцюжку (таблиця 37).

Таблиця 37 – Варіанти опису діапазону

Синтаксис	Значення	Результат
[*2]	Рівно 2 кроки	Тільки "онуки" чи вузли через один рівень.
[*1..]	Від 1 до нескінченності	Весь ланцюжок до самого кінця (поки не вперется в глухий кут).
[*..5]	Від 1 до 5 кроків	Усі у радіусі 5 рукостискань.
[*0..2]	Від 0 до 2 кроків	0 означає сам вузол l плюс сусідів на 1 і 2 кроки.

Наприклад, запит

```
MATCH (l:Lecturer)-[:WORKS_IN]->(d:Department)-[:SUBDIVISION_OF*0..2]->(paren
t:Department)
RETURN l, d, parent
```

знайде:

Дистанція 1	Кафедру, де викладач працює безпосередньо.
Дистанція 2	Факультет, до складу якого входить ця кафедра.
Дистанція 3	Інститут (або університет), до складу якого належить цей факультет.

Якщо написати просто -[:WORKS_IN]->, то в результаті буде тільки Кафедра. А із зірочкою отримуємо всю вертикаль влади над цим викладачем.

⚠ *Важливе застереження (Пастка продуктивності): використання «нескінченних» шляхів на кшталт [*] на великих графах з мільйонами зв'язків може призвести до того, що запит буде виконуватися вічно або «з'їсть» усю оперативну пам'ять. Золоте правило: Завжди намагайтеся обмежувати максимальну глибину (наприклад, *1..5), якщо ви точно не впевнені у структурі даних.*

Деякі приклади маніпулювання даними у БД lectures

Розглянемо декілька прикладів запитів до навчальної БД **lectures**, структура якої наведена у Додатку В. Наведені приклади демонструють типові операції пошуку інформації у графовій базі даних.

У таблиця 38 наведено коротку характеристику запитів.

Таблиця 38 – Приклади запитів до БД lectures

№	Запит	Що демонструє
Q1	Спроби складання дисципліни студентом	Traversal через проміжний вузол та сортування результатів
Q2	Студенти та теми дипломних робіт викладача	Використання агрегації `collect()`
Q3	Студенти з одного міста	Фільтрація за властивістю вузла та групування
Q4	Співавтори співавторів викладача	Багатокроковий обхід графа

Приклад (Q1): *подивитись усі спроби студента складання однієї дисципліни (можливі декілька оцінок)*

```
// :param studKod => 1, :param dKod => 1
MATCH (s:Student {kod:$studKod})-[:HAS_GRADE]->(ga:GradeAttempt)-[:FO
R_DISCIPLINE]->(d:Discipline {dKod:$dKod})
RETURN s.secondName, d.name, ga.mark, ga.mdate
ORDER BY ga.mdate DESC;
```

У цьому запиті використовується ланцюжок зв’язків

Student → GradeAttempt → Discipline.

Така структура дозволяє зберігати декілька спроб складання дисципліни одним студентом.
Запит повертає всі оцінки студента для заданої дисципліни у порядку спадання дати.

Приклад (Q2): *роботою кого ще зі студентів керує цей викладач? (студенти + теми дипломних робіт)*

```
// :param lectKod => 1
MATCH (sup:Lecturer {kod:$lectKod})-[:SUPERVISES]->(t:DiplomaTopic)<-[:
HAS_DIPLOMA_TOPIC]-(s:Student)
RETURN sup.secondName AS supervisor, t.dpKod, t.dpName,
    collect(s.secondName + ' ' + s.firstName) AS students
ORDER BY t.dpKod;
```

Запит знаходить теми дипломних робіт, якими керує викладач, та формує список студентів для кожної теми, а функція collect() об’єднує декілька значень у список.

Приклад (Q3): *отримати список студентів з одного міста*

```
// :param city => "м. Одеса"
MATCH (a:Address {city:$city})<-[:LIVES_AT]-(s:Student)-[:IN_GROUP]->(g:
Group)
RETURN a.city, g.gNum, collect(s.secondName + ' ' + s.firstName) AS stude
nts
ORDER BY g.gNum;
```

У цьому прикладі використовується фільтрація за властивістю вузла Address. Запит формує список студентів кожної групи, що проживають у заданому місті.

Приклад (Q4): *з ким з колег ще опублікувався співавтор цього викладача?*

⚠ *Це «класичний» графовий запит!*

```
:param lectKod => 1
MATCH (l:Lecturer {kod:$lectKod})-[:COAUTHOR_WITH]-(c1:Lecturer)
MATCH (c1)-[:COAUTHOR_WITH]-(c2:Lecturer)
WHERE c2.kod <> l.kod
RETURN c1.secondName + ' ' + c1.firstName AS coauthor,
    collect(DISTINCT c2.secondName + ' ' + c2.firstName) AS other_coauth
ors
ORDER BY coauthor;
```

Запит виконує двокроковий обхід графа:

```
Lecturer → COAUTHOR_WITH → Lecturer → COAUTHOR_WITH → Lecture
r
```

Таким чином визначаються співавтори співавторів, що дозволяє аналізувати наукові зв’язки між дослідниками.

Подібні запити у реляційних СУБД потребують використання декількох JOIN-операцій, тоді як у графовій моделі вони природно формулюються через патерни зв’язків.

Наведені приклади демонструють, що запити у графових БД формулюються через опис структур зв’язків між об’єктами. Такий підхід отримав назву ***патернового мислення (pattern thinking)***, який буде розглянутий в розділі «Патернове мислення (Pattern Thinking)».

Питання для самоперевірки

(до розділу «RU-операції Cypher (маніпулювання даними: читання та оновлення)»)

1. Які операції Cypher можна умовно віднести до RU-операцій?
2. Для чого у Cypher використовується команда MATCH і чим вона концептуально відрізняється від SELECT у SQL?
3. Яким чином у Cypher задаються шаблони пошуку у графі?
4. Як у Cypher описуються напрямлені та ненапрямлені зв’язки?
5. У чому полягають особливості пошуку шляхів у графі засобами Cypher?
6. Для чого у Cypher використовуються змінні вузлів та зв’язків?
7. Як у Cypher виконується фільтрація результатів запитів?
8. Які оператори порівняння та логічні оператори підтримуються у Cypher?
9. Які особливості повернення результатів запитів у Cypher?
10. У чому полягають особливості роботи Cypher із NULL-значеннями?
11. Які можливості надає агрегатна функція collect()?
12. Для чого у Cypher використовується оператор WITH?
13. У чому полягає призначення оператора OPTIONAL MATCH?
14. Як у Cypher виконуються операції оновлення властивостей вузлів та зв’язків?
15. Для чого і як використовується оператор UNWIND у Cypher?
16. У чому полягають особливості оновлення властивостей з використанням UNWIND?
17. Як у Cypher здійснюється обхід зв’язків графа (traversal)?
18. Що таке і як реалізується концепція Variable Length Paths (шляхи змінної довжини)?

Завдання для самостійної роботи

Написати запити для розв’язання задач, наведених для обраної ПрО розділу «Завдання для самостійної роботи» (с. 132) з використанням елементів графу, створенних при виконанні «Завдання для самостійної роботи» (с. 148).

Використовуючи інструментарій описаний в розділі «Інструменти для роботи з Neo4j», виконати створені запити.

Демонстраційний приклад

Завдання

Написати запити для розв'язання задач ДГ2.1-ДГ2.6, наведених для ПрО ***«Портфоліо студентських проєктів»*** розділу «Демонстраційний приклад» (с. 134) з використанням елементів графу, створенних при виконанні завдання розділу «Демонстраційний приклад» (с. 148). У процесі розв'язання запропонувати рішення з використанням наступних можливостей мови Cypher:

- «проста» вибірка даних з графа;
- обробка відсутніх зв'язків через OPTIONAL MATCH;
- агрегація та формування списків, зокрема через collect();
- оновлення властивостей зв'язків;
- пошук подібних проєктів і шляхів між студентами.

Розв'язання

1. Розв'язання задач на основі графа, створеного при виконанні завдання «Демонстраційний приклад» на с. 148

ДГ2.1. Проєкти заданого студента з роллю та, якщо є, керівником

В результаті виконання запиту отримаємо список проєктів заданого студента, його роль у кожному проєкті, дату підключення до проєкту та список керівників, якщо вони визначені.

```
MATCH (s:Student {student_id: 1})-[r:PARTICIPATED_IN]->(p:Project)

// Керівник може бути відсутній, тому використовується
// оператор OPTIONAL MATCH, який дозволяє не втратити проєкт,
// навіть якщо для нього ще не створено зв'язок SUPERVISES
OPTIONAL MATCH (v:Supervisor)-[:SUPERVISES]->(p)

RETURN
  s.second_name + ' ' + s.first_name AS student,
  p.title AS project,
  r.role AS role,
  r.joined_at AS joined_at,
  collect(DISTINCT v.second_name + ' ' + v.first_name) AS supervisors
ORDER BY p.title;
```

ДГ2.2. Студенти, які працювали над спільними проєктами

```
MATCH (s1:Student)-[:PARTICIPATED_IN]->(p:Project)<-[:PARTICIPATED_IN]-(s
2:Student)
WHERE s1.student_id < s2.student_id
```

```
RETURN
  s1.second_name + ' ' + s1.first_name AS student_1,
  s2.second_name + ' ' + s2.first_name AS student_2,
  collect(DISTINCT p.title) AS common_projects
ORDER BY student_1, student_2;
```

ДГ2.3. Проєкти, подібні до заданого, за спільними технологіями або тегами

```
MATCH (p:Project {project_id: 101})-[:USES|HAS_TAG]->(x)<-[:USES|HAS_TAG]-(o
ther:Project)
WHERE p.project_id <> other.project_id

RETURN
  p.title AS source_project,
  other.title AS similar_project,
  collect(DISTINCT x.name) AS common_features,
  count(DISTINCT x) AS similarity_score
ORDER BY similarity_score DESC;
```

Запит поверне проєкти, що мають із заданим проєктом спільні технології або теги, а також список таких спільних ознак і їх кількість як умовний показник схожості.

ДГ2.4. Найчастіше використані технології у проєктах студентів певної групи

```
MATCH (s:Student {group: "IC-31"})-[:PARTICIPATED_IN]->(p:Project)-[:USES]->
(t:Technology)

WITH t, count(DISTINCT p) AS project_count, collect(DISTINCT p.title) AS projec
ts

RETURN
  t.name AS technology,
  project_count,
  projects
ORDER BY project_count DESC, technology;
```

ДГ2.5. Шлях зв'язку між двома студентами

```
MATCH (s1:Student {student_id: 1}), (s2:Student {student_id: 3})

MATCH path = shortestPath(
// Використовується шлях змінної довжини до 6 зв'язків.
// Перелічені типи зв'язків визначають, через які сутності
// може проходити шлях: проєкти, технології, теги, керівників
// та похідні зв'язки подібності між проєктами.
// Напрямок зв'язків не задається, тому обхід виконується в обидва бо
```

```
ки.
 (s1)-[:PARTICIPATED_IN|USES|HAS_TAG|SUPERVISES*..6]-(s2)
)

RETURN path;
```

Запит поверне не сам графічний шлях, а *впорядкований список назв вузлів*, через які проходить найкоротший шлях, та довжину цього шляху.

ДГ2.6. Студенти, які виконували задану роль у кількох проєктах

```
MATCH (s:Student)-[r:PARTICIPATED_IN]->(p:Project)
WHERE r.role = "team-lead"

WITH s, collect(p.title) AS projects, count(p) AS project_count
WHERE project_count >= 2

RETURN
 s.second_name + ' ' + s.first_name AS student,
 r.role AS role,
 project_count,
 projects;
```

В цьому запиті з логікою все коректно, проте є технічний нюанс: після `WITH` змінна `r` вже недоступна. Тому це' запит треба відкоригувати наступним чином:

```
MATCH (s:Student)-[r:PARTICIPATED_IN]->(p:Project)
WHERE r.role = "team-lead"

WITH s, r.role AS role, collect(p.title) AS projects, count(p) AS project_cou
nt
WHERE project_count >= 2

RETURN
 s.second_name + ' ' + s.first_name AS student,
 role,
 project_count,
 projects;
```

В результаті отримаємо студентів, які виконували задану роль у двох або більше проєктах, із переліком таких проєктів та датами початку участі.

2. Оновлення властивостей зв'язків

Нагадуємо, що у property graph властивості можуть мати не лише вузли, а й зв'язки.

Тому уточнимо властивості вже створених зв'язків.

Уточнення призначення використання технологій

```
UNWIND [
 {project_id: 101, technology: "MongoDB", purpose: "БД"},
 {project_id: 101, technology: "Node.js", purpose: "back-end"},
 {project_id: 101, technology: "React", purpose: "front-end"},
 {project_id: 102, technology: "Python", purpose: "аналіз даних"},
 {project_id: 102, technology: "MongoDB", purpose: "зберігання даних"},
 {project_id: 102, technology: "Scikit-learn", purpose: "ML"},
 {project_id: 103, technology: "Neo4j", purpose: "графова БД"},
 {project_id: 103, technology: "React", purpose: "UI"}
] AS row

MATCH (p:Project {project_id: row.project_id})-[r:USES]->
                                        (t:Technology {name: row.technology})
SET r.purpose = row.purpose;
```

Додавання оцінок керівників

```
UNWIND [
 {supervisor_id: 1, project_id: 101, review_score: 95,
                          review_date: date("2026-03-10"), status: "approved"},
 {supervisor_id: 2, project_id: 102, review_score: 98,
                          review_date: date("2026-02-20"), status: "approved"},
 {supervisor_id: 1, project_id: 103, review_score: 80,
                    review_date: date("2026-05-15"), status: "needs revision"}
] AS row

MATCH (v:Supervisor {supervisor_id: row.supervisor_id})-[r:SUPERVISES]->
                                        (p:Project {project_id: row.project_id})
SET
 r.review_score = row.review_score,
 r.review_date = row.review_date,
 r.status = row.status;
```

Створення / уточнення похідних зв'язків SIMILAR_TO

```
MATCH (p1:Project)-[:USES|HAS_TAG]->(x)<-[:USES|HAS_TAG]-(p2:Projec
t)
WHERE p1.project_id < p2.project_id

WITH
 p1,
 p2,
 collect(DISTINCT x.name) AS common_features,
```

```
  count(DISTINCT x) AS common_count

MERGE (p1)-[r:SIMILAR_TO]->(p2)
SET
  r.similarity = common_count,
  r.reason = "common technology or tag",
  r.common_features = common_features;
```

3. Повторне розв'язання задач з урахуванням оновлених властивостей

ДГ2.1. Проєкти студента з керівниками, ролями та оцінками

```
MATCH (s:Student {student_id: 1})-[part:PARTICIPATED_IN]->(p:Project)

OPTIONAL MATCH (v:Supervisor)-[sup:SUPERVISES]->(p)

RETURN
  s.second_name + ' ' + s.first_name AS student,
  p.title AS project,
  part.role AS student_role,
  part.joined_at AS joined_at,
  collect(DISTINCT {
    supervisor: v.second_name + ' ' + v.first_name,
    score: sup.review_score,
    review_date: sup.review_date,
    status: sup.status
  }) AS supervision
ORDER BY p.title;
```

ДГ2.3. Подібні проєкти через створений зв'язок SIMILAR_TO

```
MATCH (p:Project {project_id: 101})-[r:SIMILAR_TO]-(other:Project)

RETURN
  p.title AS source_project,
  other.title AS similar_project,
  r.similarity AS similarity,
  r.reason AS reason,
  r.common_features AS common_features
ORDER BY similarity DESC;
```

ДГ2.4. Технології за групою з урахуванням призначення використання

```
MATCH (s:Student {group: "IC-31"})-[:PARTICIPATED_IN]->(p:Project)-[u:USES]-
>(t:Technology)

WITH
```

```
  t.name AS technology,
  u.purpose AS purpose,
  collect(DISTINCT p.title) AS projects,
  count(DISTINCT p) AS project_count

RETURN
  technology,
  purpose,
  project_count,
  projects
ORDER BY project_count DESC, technology;
```

Додатковий аналітичний запит: середня оцінка проєктів за керівниками

```
MATCH (v:Supervisor)-[r:SUPERVISES]->(p:Project)
WHERE r.review_score IS NOT NULL

RETURN
  v.second_name + ' ' + v.first_name AS supervisor,
  avg(r.review_score) AS avg_score,
  count(p) AS project_count,
  collect(p.title) AS projects
ORDER BY avg_score DESC;
```

ДГ2.5. Шлях між студентами з поясненням вузлів шляху

```
MATCH (s1:Student {student_id: 1}), (s2:Student {student_id: 3})

MATCH path = shortestPath(
  (s1)-[:PARTICIPATED_IN|USES|HAS_TAG|SUPERVISES|SIMILAR_TO*..6]-
(s2)
)

RETURN
  [n IN nodes(path) |
    coalesce(n.second_name + ' ' + n.first_name, n.title, n.name)
  ] AS path_nodes,
  length(path) AS path_length;
```

ДГ2.6. Студенти, які виконували задану роль у кількох проєктах

```
MATCH (s:Student)-[r:PARTICIPATED_IN]->(p:Project)
WHERE r.role = "team-lead"

WITH
  s,
```

```
  r.role AS role,
  collect({
    project: p.title,
    joined_at: r.joined_at
  }) AS participations,
  count(p) AS project_count

WHERE project_count >= 2

RETURN
  s.second_name + ' ' + s.first_name AS student,
  role,
  project_count,
  participations
ORDER BY project_count DESC;
```

Наведені запити демонструють, що у графовій моделі основна увага приділяється не лише властивостям окремих вузлів, а й структурі зв'язків між ними. Властивості зв'язків role, joined_at, purpose, review_score, review_date, status, similarity дозволяють уточнювати зміст взаємодії між сутностями та виконувати аналітичні запити без додаткового перенесення логіки на рівень прикладної програми.

Управління даними в Cypher

Cypher також містить команди, що дозволяють управляти доступом до бази даних, користувачами та правами доступу. Такі команди іноді відносять до ***Data Control Language (DCL)***. Вони використовуються для:

- створення користувачів;
- управління ролями;
- надання або обмеження прав доступу;
- управління транзакціями.

На практиці більшість операцій адміністрування Neo4j виконується через Neo4j Browser або Neo4j Desktop, де управління користувачами і ролями реалізовано у вигляді графічного інтерфейсу. Тому DCL у прикладах Cypher можна побачити досить рідко.

Управління користувачами та ролями

У Neo4j адміністратор може створювати користувачів бази даних. Наприклад:

```
CREATE USER student1
SET PASSWORD '123456'
CHANGE NOT REQUIRED;
```

Ця команда створює нового користувача student1. Параметр

```
CHANGE [NOT] REQUIRED
```

означає, що користувач [не] зобов'язаний змінювати пароль при першому вході.

У Neo4j використовується ***рольова модель доступу***, тобто користувачам можуть бути призначені ролі. Наприклад, створюємо роль (на кшталт групової ролі в РСУБД Oracle, PostgreSQL, MS SQL):

```
CREATE ROLE analyst;
```

Надалі роль можна призначити користувачу:

```
GRANT ROLE analyst TO student1;
```

Neo4j дозволяє передивитись:

- список користувачів

```
SHOW USERS;
```

і отримати для кожного username, roles, suspended та passwordChangeRequired;

- список ролей

```
SHOW ROLES;
```

- список користувачів, які мають конкретну роль

```
SHOW USERS WITH ROLE analyst;
```

Видалення користувачів та ролей виконується командою DROP. Наприклад:

```
DROP USER student1;
DROP ROLE analyst;
```

Ролі дозволяють централізовано керувати правами доступу.

Управління доступом

Neo4j дозволяє гнучко управляти доступом до бази даних та елементів графа: вузлів, зв'язків та властивостей вузлів і зв'язків.

Управління доступом виконується призначенням та відкликанням привілеїв певним ролям за допомогою команд GRANT, DENY та REVOKE.

⚠ *Важливий нюанс: у Neo4j Community Edition розвинений механізм керування ролями та привілеями недоступний; повноцінне рольове адміністрування підтримується в Enterprise Edition та AuraDB. У безкоштовній версії роль лише одна – admin, і вона не налаштовується.*

Наприклад, щоб надати ролі analyst *право читання* ***всього графа***, тобто дозволити користувачам з роллю analyst виконувати запити читання будь-яких даних у графі, треба використати команду **GRANT** і вказати *тип ресурсів*, до яких надається доступ – ELEMENTS (всі сутності), NODES (тільки вузли) або RELATIONSHIPS (тільки зв'язки):

```
GRANT MATCH {*} ON GRAPH lectures
ELEMENTS * TO analyst;
```

Проте, якщо треба дозволити користувачам ***доступ лише до вузлів певного типу***, їх треба перелічити в команді GRANT. Наприклад, дозволити читання вузлів Lecturer та Publication *кільком ролям*:

```
GRANT MATCH {*} ON GRAPH lectures
NODES Lecturer, Publication TO analyst, researcher;
```

Це більш гнучке управління привілеями, ніж в SQL, де довелося б писати дві окремих команди GRANT – для кожної таблиці.

Аналогічно можна надати ***доступ до певного типу зв'язків***. Наприклад, дозволити користувачам проходити по зв'язках COAUTHOR_WITH:

```
GRANT TRAVERSE ON GRAPH lectures
RELATIONSHIP COAUTHOR_WITH TO analyst;
```

Більш того, на відміну від того ж SQL, де привілеї на доступ до окремих стовбців обмежені тільки UPDATE та REFERENCES, Cypher дозволяє дозволити навіть читати ***окремі властивості*** вузлів чи зв'язків (***Column-level Security***). Наприклад, дозволити бачити прізвища студентів, але заборонити бачити інші властивості:

```
GRANT MATCH {secondName} ON GRAPH lectures
NODES Student TO analyst;
```

Neo4j також дозволяє ***контролювати створення нових вузлів та зв'язків***.

Наступний запит, наприклад, надає аналітикам право створювати вузли типу Student:

```
GRANT CREATE ON GRAPH lectures
NODE Student TO analyst;
```

Надані права можуть бути скасовані командою **REVOKE**:

```
REVOKE MATCH {*} ON GRAPH lectures ELEMENTS * FROM researcher;
```

Треба розуміти, що в Neo4j команда REVOKE не є «відніманням» частини прав, він працює як видалення конкретного запису зі списку привілеїв.

Якщо був виданий доступ через ELEMENTS*, то в системі безпеки створився один запис: «Роль researcher може читати ЕЛЕМЕНТИ (вузли та зв'язки)».

Якщо потім спробувати виконати, наприклад, REVOKE ... RELATIONSHIPS *, Neo4j подивиться в список прав, не знайде там запису, виданого саме на RELATIONSHIPS, і нічого не зробить (або видасть помилку, що такого привілею немає).

Якщо необхідно залишити доступ до всього, але «заблокувати» конкретно, наприклад, зв'язки, можна накласти маску, що забороняє: скористатися командою **DENY**, яка має пріоритет над GRANT.

Наприклад, залишаємо спільний доступ, але вішаємо «замок» на зв'язок:

```
DENY MATCH {*} ON GRAPH lectures RELATIONSHIPS * TO researcher;
```

У цьому випадку у дослідника залишиться право на ELEMENTS, але при спробі звернутися до зв'язків система побачить DENY і заблокує запит до них.

Отже, на відміну від класичного SQL-підходу, Neo4j дозволяє керувати доступом не лише до «табличних» структур, а й до власне елементів графа – вузлів, зв'язків і їх властивостей. Така модель доступу дозволяє реалізувати досить детальний контроль над структурою графа та операціями користувачів, що є важливим для корпоративних інформаційних систем.

Для узагальнення відмінностей між традиційною реляційною моделлю безпеки та графовим підходом Neo4j наведемо порівняльну характеристику. (таблиця 39):

Таблиця 39 – Особливості підходів до безпеки: SQL vs Cypher

Особливість	SQL	Cypher (Neo4j)
Область дії	Зазвичай одна таблиця однією командою GRANT	Можна задавати привілеї одразу для кількох міток вузлів
Зв'язки	Реалізуються через таблиці та зовнішні ключі	Привілеї можна надавати окремо на типи зв'язків
Властивості / стовпці	Переважно SELECT / UPDATE для стовпців	Можна керувати доступом до окремих властивостей вузлів і зв'язків
Гранулярність	Часто потребує view або додаткової логіки	Підтримується тонке налаштування доступу на рівні графових елементів

Особливість	SQL	Cypher (Neo4j)
Заборона доступу	REVOKE знімає дозвіл; DENY реалізується не у всіх СУБД однаково	DENY має пріоритет над GRANT
Предметна орієнтація	Орієнтація на таблиці	Орієнтація на вузли, зв’язки, властивості та traversal

Управління транзакціями

Cypher підтримує механізм ***транзакцій***, що забезпечує цілісність даних.

Нагадую, що транзакція – це послідовність операцій, яка виконується як єдине ціле і діє за принципом «все або нічого». Як і в багатьох РСУБД, транзакція починається командою BEGIN і закінчується COMMIT. Наприклад:

```
BEGIN
CREATE (l:Lecturer {name:"Новий викладач"})
COMMIT
```

Якщо під час виконання виникає помилка, транзакція може бути скасована командою ROLLBACK.

Патернове мислення (Pattern Thinking)

Однією з ключових ідей графових баз даних є ***патернове мислення (pattern thinking)***.

Реляційне мислення (SQL thinking) vs графове мислення (Graph thinking)

На відміну від реляційної моделі, де запит формулюється як операції над таблицями, у графових БД користувач описує ***структуру зв'язків***, яку потрібно знайти у графі [10, 16]. Такий підхід дозволяє природно формулювати запити для задач, у яких структура зв'язків є частиною самої умови запиту.

В реляційній моделі (***реляційне мислення***) зв'язки між об'єктами зберігаються у вигляді зовнішніх ключів. Для отримання інформації необхідно з'єднувати таблиці за допомогою команд JOIN. Наприклад, для структури на рис. 30

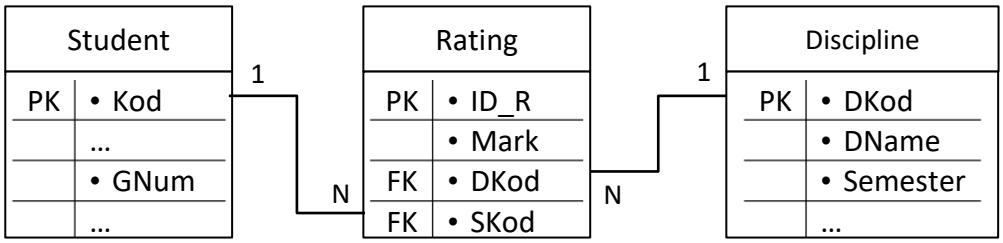


Рис. 30 – Діаграма реляційної моделі

може бути побудований такий запит:

```
SELECT …
FROM Student
JOIN Rating …
JOIN Discipline …
```

У графовій моделі (***графове мислення***) зв'язки є *частиною структури даних*.

```
(Student)-[:ENROLLED_IN]->(Course)
```

Запит формулюється як *опис патерну графа*.

```
MATCH (s:Student)-[:ENROLLED_IN]->(c:Course)
RETURN s, c
```

Тобто користувач описує ***структуру зв'язків***, яку необхідно знайти.

Один із класичних прикладів патернового мислення є пошук ***«друзів друзів» (friend-of-a-friend)***.

У соціальній мережі користувачі можуть бути пов'язані відношенням:

```
(Person)-[:FRIEND]->(Person)
```

що відповідає, наприклад, графу

Олена — FRIEND — Володимир — FRIEND — Марія

Тобто, *Марія* ***є другом друга*** *Олени*. Відповідно, запит

```
MATCH (p:Person {name:"Олена"})-[:FRIEND]->(f)-[:FRIEND]->(foaf)
WHERE foaf <> p
RETURN DISTINCT foaf.name;
```

знаходить **друзів друзів особи "**Олена**"**.

Тепер розглянемо подібну задачу у навчальній БД **lectures**.

Одним із типових прикладів є аналіз *мережі наукових співпраць*.

Наприклад, у графовій моделі наукових публікацій такі наукові співпраці можуть бути представлені як зв'язки

```
(Lecturer)-[:COAUTHOR_WITH]-(Lecturer)
```

що утворює *граф співпраці дослідників*.

Однією з типових задач є пошук **потенційних співпрацівників** – викладачів, які ще не публікувалися разом, але мають спільних співавторів.

```
MATCH (l:Lecturer {kod:1})-[:COAUTHOR_WITH]-(c1:Lecturer)
MATCH (c1)-[:COAUTHOR_WITH]-(c2:Lecturer)

WHERE c2.kod <> l.kod
AND NOT (l)-[:COAUTHOR_WITH]-(c2)

RETURN DISTINCT c2.secondName, c2.firstName;
```

Запит виконує двокроковий обхід графа:

Lecturer → COAUTHOR_WITH → Lecturer → COAUTHOR_WITH → Lecture r

і знаходить *співавторів співавторів викладача*. При цьому виключається початковий викладач і вже існуючі співпраці.

У БД lectures є наступні зв'язки за спільними публікаціями

Малахов – COAUTHOR_WITH – Пенко – COAUTHOR_WITH – Шпінарева

Таким чином, запит знайде, що *Шпінарева* ***є співавтором співавтора*** *Малахова*.

Як можна побачити, цей запит природно формулюється через ***патерн зв'язків***:

```
(l)-[:COAUTHOR_WITH]-(c1)-[:COAUTHOR_WITH]-(c2)
```

Знову ж таки, у реляційній моделі така задача потребує кількох JOIN-операцій над таблицею співавторів.

Пошук найкоротшого шляху між вузлами

Іншою типовою задачею є пошук ***найкоротшого ланцюга***. Cypher *автоматично* знаходить найкоротший шлях у графі за допомогою функції shortestPath(). Наприклад, знайдемо *найкоротший ланцюг співпраці*:

```
MATCH path = shortestPath(
  (l1:Lecturer {kod:1})-[:COAUTHOR_WITH*1..5]-(l2:Lecturer {kod:5})
)
RETURN path;
```

⚠ *Патернове мислення є фундаментальною особливістю графових БД.*

Замість виконання операцій над таблицями користувач описує структуру зв'язків, яку система повинна знайти у графі. Такий підхід значно спрощує роботу з сильно зв'язаними даними і дозволяє графовим СУБД особливо ефективно виконувати складні аналітичні запити (таблиця 40).

Таблиця 40 – Технології до розв'язання задач: SQL vs Cypher

задача	**SQL**	**Cypher**
знайти курс студента	JOIN	pattern
знайти друзів друзів	рекурсивні JOIN	traversal
знайти найкоротший шлях	складні алгоритми	shortestPath

Обмеження графової моделі

Незважаючи на значні переваги графових баз даних для моделювання сильно зв'язаних даних, графова модель не є універсальним рішенням для всіх інформаційних систем. Як і інші моделі даних, вона має певні обмеження, які необхідно враховувати під час проєктування баз даних та вибору СУБД. До основних обмежень графової моделі можна віднести:

1) неефективність для табличних і масових операцій;
2) складність виконання аналітичних і OLAP-операцій;
3) підвищені вимоги до пам'яті;
4) складність організації горизонтального масштабування;
5) не завжди природну відповідність предметній області;
6) слабшу стандартизацію екосистеми (залежність від конкретної СУБД).

Розглянемо ці обмеження детальніше.

Неефективність для табличних і масових операцій

Графові БД оптимізовані передусім для виконання traversal-операцій, а не для табличної аналітики.

Наприклад, задачі на кшталт *«обчислення середньої зарплати викладачів»*, *«визначення кількості студентів на курсі»*, *«розрахунок сумарного бюджету проєктів»*

– у реляційних базах даних легко реалізуються за допомогою агрегатних функцій SQL, наприклад:

```
SELECT AVG(salary) FROM Lecturer;
```

– у графових БД такі операції також можливі, проте вони не є їх сильною стороною.

Графові БД призначені насамперед для ефективної роботи зі зв'язками між сутностями, а не для обробки великих масивів однорідних табличних даних [1].

Складність аналітичних і OLAP-операцій

Для задач, пов'язаних із *статистикою*, *групуванням* та *агрегуванням* великих обсягів даних, графова модель зазвичай поступається іншим типам СУБД.

До таких задач належать, наприклад: *«аналіз продажів по роках»*, *«відстеження динаміки кількості студентів»*, *«формування фінансової звітності»*, *«побудова складних аналітичних звітів»*.

Для подібних задач більш придатними є:

– реляційні СУБД;
– колонкові (стовбцеві) бази даних;
– спеціалізовані OLAP-системи.

Графові БД не проектувалися як основний інструмент для табличної аналітики.

Підвищені вимоги до пам’яті

У графових базах даних зв’язки між об’єктами зберігаються як окремі структурні елементи графа. Кожен зв’язок містить посилання на початкову та кінцеву вершини, а також може мати власні властивості.

Це призводить до як до збільшення кількості посилань між елементами даних, так і до більш складної структури зберігання. У результаті графові бази даних можуть потребувати більше пам’яті, ніж реляційні або документні системи для зберігання тих самих даних. Особливо для таких структур, як дуже великих графів, *соціальні мережі, knowledge graphs*.

Складність горизонтального масштабування

Складність організації горизонтального масштабування є одним з найвідоміших недоліків графових БД.

Причина полягає в тому, що зв’язки між вершинами можуть вести до будь-якого елемента графа. Через це розподіл графа між декількома серверами є значно складнішим, ніж у реляційних або документних системах.

У документних та реляційних СУБД широко застосовуються механізми *шардування* (sharding) та *фрагментації* (partitioning) відповідно. У графових базах даних реалізація подібних підходів значно складніша, оскільки розрив зв’язків між частинами графа може негативно впливати на ефективність traversal-операцій.

Не завжди природна модель даних

Не кожна предметна область природно описується графом.

Наприклад, у таких ПрО, як *«бухгалтерський облік»*, *«складський облік»*, *«банківські транзакції»*, структура даних здебільшого має табличний характер і складається з відносно незалежних записів.

У таких випадках використання графової моделі може не лише не дати переваг, але й ускладнити реалізацію інформаційної системи.

Слабка стандартизація екосистеми

Ще одним обмеженням графових баз даних є відносно слабка стандартизація їх екосистеми.

На відміну від реляційних СУБД, де основною мовою роботи з даними є стандартизована мова SQL, у графових системах використовується декілька різних мов запитів, наприклад: Cypher, Gremlin, SPARQL.

Якщо, незважаючи на те, що кожна з реляційних СУБД має притаманний їй певний діалект, тобто розширення SQL, всі вони підтримують стандарт цієї мови, то наведені мови графових СУБД мають різні синтаксис і концепції, що може ускладнювати перенесення застосунків між різними системами.

Додаткові практичні обмеження

Як зазначають М. Фаулер та П. Дж. Садаладж [1], графові бази даних можуть бути не найкращим вибором у ситуаціях, коли необхідно виконувати масові зміни властивостей великої кількості сутностей.

Наприклад, якщо аналітична задача потребує одночасного оновлення властивостей у великій кількості вузлів, подібні операції можуть виконуватися менш ефективно, ніж у табличних системах.

Крім того, деякі графові СУБД можуть мати обмеження при роботі з дуже великими обсягами даних, особливо у випадках виконання глобальних операцій над усім графом.

Коли графова модель може бути недоцільною

Отже, застосування графової моделі може бути недоцільним у таких випадках [10]:

- структура даних є простою та ієрархічною;
- домінують агрегатні операції (SUM, AVG, GROUP BY);
- більшість запитів працюють із одним документом або таблицею;
- важливою є сувора таблична звітність;
- відсутні складні транзитивні зв'язки між сутностями.

У таких ситуаціях реляційні або документні бази даних можуть бути простішим і ефективнішим рішенням.

Висновок

Таким чином, різні моделі даних орієнтовані на різні типи задач.

Графова модель природно підтримує задачі, у яких структура зв'язків між сутностями є важливішою за ізольовані атрибути об'єктів. Водночас для інших типів задач – наприклад, зберігання напівструктурованих документів або виконання масових аналітичних операцій – більш доцільними можуть бути інші моделі даних, зокрема документні бази даних [3].

У наступному розділі розглянемо порівняльний приклад використання MongoDB та Neo4j для однієї і тієї ж предметної області.

Питання для самоперевірки

(до розділів «Патернове мислення (Pattern Thinking)», «Обмеження графової моделі»)

1. Що таке патернове мислення (Pattern Thinking) у контексті графових БД?
2. Чим патернове мислення у графових БД відрізняється від традиційного табличного підходу?
3. У чому полягає роль шаблонів графа у мові Cypher?
4. Які переваги має патерновий підхід при аналізі взаємопов'язаних даних?
5. У яких випадках графова модель даних може бути неефективною?
6. У чому полягають складності масштабування графових БД?
7. Чому при проектуванні графової БД важливо враховувати характер майбутніх traversal-запитів?

Порівняльний кейс Mongo vs Neo4j

Для кращого розуміння особливостей документної та графової моделей розглянемо один і той самий приклад предметної області – ***мережу наукових публікацій***.

У цій системі необхідно зберігати інформацію про: *викладачів*, *наукові публікації*, *співавторство* між викладачами.

Основні типи зв'язків:

викладач → автор публікації
викладач ↔ співавтор іншого викладача

Реалізація у MongoDB

У документній моделі дані можуть зберігатися у вигляді документів. Наприклад:

```
{
 "_id": 1,
 "name": "Євгеній Малахов",
 "publications": [
  {
   "title": "Graph Databases in Education",
   "year": 2023
  },
  {
   "title": "NoSQL Systems",
   "year": 2022
  }
 ],
 "coauthors": [
  "Пенко",
  "Шпінерьова"
 ]
}
```

У цьому випадку інформація про викладача, його публікації та співавторів зберігається в одному документі.

Такий підхід добре підходить для задач, у яких:

- дані природно групуються в один документ;
- більшість запитів працюють із одним об'єктом.

Наприклад, отримати інформацію про викладача можна простим запитом:

```
db.teachers.find({ name: "Малахов" })
```

Обмеження документної моделі

Однак у випадку складних зв'язків між авторами документна модель може ускладнювати виконання деяких запитів.

Наприклад, задача:

знайти співавторів співавторів певного викладача

потребує складної логіки обробки вкладених структур або кількох запитів.

Реалізація у Neo4j

У графовій моделі ті самі дані можуть бути представлені як граф.

```
(Малахов)-[:AUTHORED]->(Publication)

(Малахов)-[:COAUTHOR_WITH]->(Пенко)
(Пенко)-[:COAUTHOR_WITH]->(Шпінарева)
```

У Neo4j такі зв'язки є *першокласними елементами моделі даних*, на яких ґрунтується розв'язання відповідних задач . Наприклад, проста задача для багатьох типів БД – знайти співавторів викладача:

```
MATCH (l:Lecturer {name:"Малахов"})-[:COAUTHOR_WITH]-(c)
RETURN c
```

А знайти *співавторів співавторів* притаманна саме графовій БД:

```
MATCH (l:Lecturer {name:"Малахов"})-[:COAUTHOR_WITH]-(c1) -
[:COAUTHOR_WITH]-(c2)
RETURN c2
```

Тобто, подібні запити *є природними* для графової моделі.

Порівняльний аналіз MongoDB та Neo4j

Розглянуті приклади показують, що документна та графова моделі даних по-різному представляють одну й ту саму предметну область та орієнтовані на різні типи задач.

Для узагальнення основних відмінностей між підходами MongoDB та Neo4j наведено порівняння їхніх характеристик у таблиця 41.

Таблиця 41 – Основні відмінності документної та графової моделей (MongoDB і Neo4j)

Характеристика	**MongoDB**	**Neo4j**
Модель даних	документи	граф
Основна одиниця	документ	вузол та зв'язок
Сильна сторона	робота з документами	робота зі зв'язками
Типові задачі	зберігання напівструктурованих даних	аналіз мереж та зв'язків

Отже, документна та графова моделі не конкурують безпосередньо, а орієнтовані на різні типи задач. Розглянуті приклади демонструють, що одна й та сама предметна область може бути представлена різними моделями даних залежно від характеру майбутніх запитів та способу навігації між сутностями.

Саме тому у сучасних інформаційних системах часто застосовується підхід ***polyglot persistence***, коли система працює одночасно с різним типами баз даних, які застосовуються для різних задач [1].

Для «чого» та «коли» вибирати NoSQL (інженерна перспектива)

Як зазначає М. Клеппман [14], вибір технології зберігання даних повинен визначатися насамперед характером доступу до даних і вимогами до масштабування системи.

Критерії вибору на користь документної БД

За спостереженням М. Стоунбрейкера, системи керування даними нового покоління часто оптимізуються під конкретні типи операцій, що дозволяє досягти значно кращої масштабованості у порівнянні з універсальними СУБД [4].

Документна база виправдана, коли домінують вкладені та слабо структуровані дані, схема яких часто змінюється і не вкладається в жорстке табличне подання. Вона особливо ефективна при профілі навантаження з інтенсивним записом подій та при запитах за ключами та селективними фільтрами з вимогами до низької латентності. Часта еволюція продукту без важких міграцій DDL – ще один аргумент на користь MongoDB, як і потреба у горизонтальному масштабуванні та георозподілі. Додаткову перевагу дають функції з коробки – TTL, текстовий та геопошук, – які за помірних обсягів дозволяють не підключати окремі технології.

Шаблони моделювання в MongoDB

Вибір між вбудовуванням і посиланнями визначається характером доступу: вбудовування підходить для зв’язків «один-до-багатьох» з невеликим і відносно незмінним списком, коли дані часто читаються спільно, тоді як посилання переважні для великих колекцій або колекцій підлеглих сутностей, що повторно використовуються у різних контекстах і активно змінюються. Для часових рядів та телеметрії корисно агрегувати події у «контейнери» за часом та ключем доступу; розмір відра вибирається із компромісу між числом документів та вартістю оновлень, а політика TTL визначає термін зберігання. Паттерн outlier виділяє рідкісні та великі елементи в окрему колекцію, щоб основний документ залишався компактним. Передраховані представлення зберігають заздалегідь агреговані дані під критичні екрани та звіти, що скорочує латентність читання. При роботі з масивами важливо обмежувати їх зростання, використовувати $push з параметром $slice і в міру збільшення активності виносити гарячу частину в спеціалізовані колекції.

Антипатерни

Відсутність правил та єдиної домовленості за схемою при використанні однієї колекції для різношерстих даних призводить до хаосу та ускладнює індексацію. Перенесення реляційної нормалізації позбавляє документну модель її сильних сторін і різко збільшує залежність від $lookup. Створення гігантських документів з необмеженими списками призводить до гарячих точок запису та перепакування. Пагінація через skip на великих офсетах та агрегати без

відповідних індексів викликають деградацію продуктивності та нестабільні затримки.

Масштабування та експлуатація

Replica set забезпечує стійкість до відмов і можливість розвантажувати primary читаннями з вторинних вузлів, але вимагає усвідомленого ставлення до свіжості даних. Шардинг додає горизонтальне масштабування і спирається на вдалий вибір шард-ключа: він повинен мати достатню кардинальність, забезпечувати рівномірний розподіл і локальність запитів; монотонно зростаючі або гарячі ключі по можливості замінюють хешованими. Глобальні операції та $lookup у шардованому середовищі потрібно проектувати з розумінням міжшардових передач. Параметри writeConcern, readConcern та readPreference служать регуляторами SLA і дозволяють точково налаштовувати компроміси між свіжістю, надійністю та латентністю. Спостережуваність будується навколо профайлера, експорту метрик та оповіщень про зростання робочого набору та черг.

Доцільність використання графової БД

Графові бази даних доцільно застосовувати у системах, де запити передбачають багаторазові переходи між пов’язаними сутностями, наприклад пошук ланцюгів зв’язків, маршрутів або аналіз багаторівневих залежностей.

Як підкреслюють І. Робінсон, Дж. Вебер та Е. Ейфрем [10], графові бази даних особливо ефективні у задачах, де складність запитів визначається структурою зв’язків між даними, а не лише їх атрибутами. У таких випадках продуктивність значною мірою залежить від можливості швидко виконувати переходи (traversal) між пов’язаними елементами графа.

Особливості моделювання

Проєктування графової БД орієнтоване насамперед на *типи зв’язків між сутностями*, а не на структуру атрибутів.

Основними елементами моделі є:

- вузли (nodes);
- зв’язки (relationships);
- властивості (properties).

Як підкреслюють автори [10], ефективність графових СУБД значною мірою визначається прямим збереженням зв’язків між вузлами, що дозволяє виконувати переходи між пов’язаними елементами без пошуку через глобальні індекси.

Коли залишитися на PostgreSQL (або доповнити його)

Якщо система заснована на складних міжтабличних інваріантах і суворої цілісності, реляційна модель залишається природним вибором. Для багаторазової ad-hoc-аналітики, BI та важких JOIN/віконних функцій на онлайні SQL забезпечує виразність та оптимізаторну підтримку, які важко відтворити у

документній базі. Коли навантаження вкладається у вертикальний масштаб одного вузла, і простота експлуатації є важливішою за гнучкість схеми, PostgreSQL часто виявляється економічнішим. До того ж JSONB дозволяє отримати певну гнучкість схеми всередині PostgreSQL без переходу на інший стек [14].

Вартість володіння та нефункціональні вимоги

Економіка володіння визначається насамперед пам'яттю під індекси та робочий набір, де компресія може помітно змінити розрахунки. Вибір сховища впирається в баланс IOPS та об'єму: швидкі SSD дають виграш у латентності, а процедури снапшотів та бекапів формують основу для відновлення. Розподіл по зонах підвищує стійкість до відмов, але додає мережеву вартість і управлінську складність. У сферах із підвищеними вимогами до комплаєнсу важливі шифрування «на диску» та «у польоті», аудит дій та збереження журналів.

Для різних моделей баз даних структура витрат може істотно відрізнятися.

Реляційні СУБД зазвичай мають нижчі вимоги до пам'яті при табличному зберіганні даних, тоді як документні бази даних можуть потребувати додаткової пам'яті для зберігання вкладених структур і індексів.

Документні бази даних характеризуються зберіганням складних вкладених структур, що може призводити до збільшення обсягу документів і розміру індексів. Водночас така модель дозволяє зменшити кількість операцій з'єднання та підвищити швидкість доступу до пов'язаних даних.

Графові БД, у свою чергу, зберігають зв'язки як окремі елементи структури, що може збільшувати обсяг даних, але забезпечує високу ефективність навігаційних запитів.

Чек-лист прийняття рішення

Перед вибором технології корисно описати ключові запити, оцінити їх частоту та очікувані затримки, а також вирішити

1) для MongoDB:
   - чи достатньо вбудовування чи потрібні посилання та з'єднання з $lookup;
   - слід оцінити темпи зростання даних та кардинальність потенційних ключів, виділити сценарії, що вимагають транзакцій через кілька агрегатів, та зафіксувати цільові рівні P95/P99 за латентністю разом із доступним обсягом пам'яті під індекси;
   - чи критичні і покриваються «з коробки» такі важливі й спеціалізовані функції, як TTL, гео- та текстовий пошук, що спрощує архітектуру. Нарешті, варто заздалегідь продумати, як закриватимуться аналітичні потреби – через окреме сховище, пошуковий двигун або вітрини.
2) для Neo4j:
   - чи містить предметна область складні мережі зв'язків;
   - чи потрібен пошук шляхів або багаторівневих залежностей;
   - чи виконуються запити типу «друзі друзів» або «співавтори співавторів»;
   - чи є зв'язки важливішими за атрибути сутностей.

3) для PostgreSQL:
- чи містить система складні транзакційні інваріанти;
- чи потрібна строга підтримка цілісності даних;
- чи передбачається виконання складних аналітичних SQL-запитів;
- чи можуть дані бути природно представлені у вигляді таблиць;
- чи є потреба у стандартних SQL-інструментах аналітики та BI.

Як вже було зазначено раніше, у практичних системах вибір СУБД часто визначається типом задач, які домінують у предметній області (таблиця 40) [14]. Спрощену схему прийняття архітектурного рішення щодо вибору типу бази даних наведено на рис. 30.

Таблиця 42 – Вибір БД в залежності від задач ПрО

Тип задачі	**Рекомендована БД**
таблична аналітика	PostgreSQL
документні дані	MongoDB
мережі зв’язків	Neo4j
повнотекстовий пошук	Elasticsearch

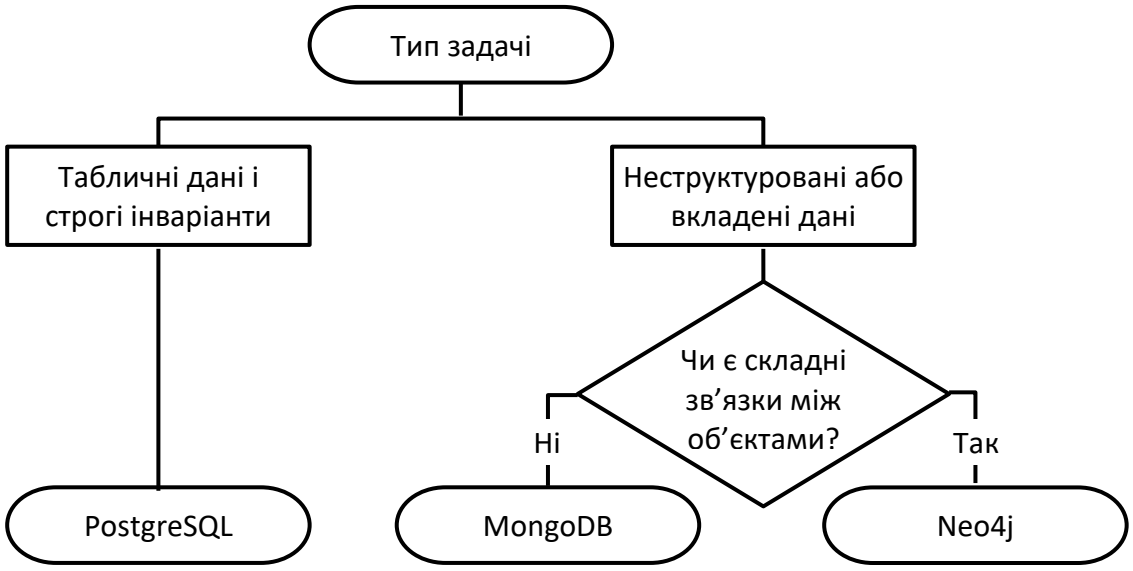


Рис. 31 – Архітектурне дерево вибору БД

Гібридні стратегії (polyglot persistence)

У складних інформаційних системах одна модель даних рідко може оптимально задовольнити всі вимоги. Тому на практиці часто використовується комбінування різних типів баз даних у межах єдиної архітектури.

У сучасних архітектурах поширеною є стратегія ***polyglot persistence***, коли різні типи баз даних використовуються для різних задач однієї системи [1, 14].

Одним з практичних підходів є рознесення обов’язків: транзакційний контур OLTP і суворі інваріанти залишити в PostgreSQL, а оперативні профілі, кеші та стрічки подій зберігати в MongoDB. Для пошуку за текстом та аналітичних розрізів добре працює зв’язка з Elasticsearch або OpenSearch, де

вихідні дані лежать у MongoDB або PostgreSQL, а індекс служить швидким поданням. Обмін між системами може будуватися через CDC-механіку на кшталт Debezium або через шаблон Transactional Outbox, що знижує ризик неузгодженості.

Таким чином, сучасна інженерія даних дедалі менше спирається на пошук «універсальної СУБД» і дедалі більше – на вибір моделі даних, що найкраще відповідає характеру задач та профілю навантаження системи.

Питання для самоперевірки

(до розділу «Для «чого» та «коли» вибирати NoSQL (інженерна перспектива)»)

1. Чому вибір моделі даних та СУБД є інженерним компромісом?
2. У яких випадках доцільно використовувати реляційні, документні та графові БД?
3. Чому NoSQL-СУБД не слід розглядати як універсальну заміну реляційних систем?
4. Які фактори необхідно враховувати при виборі моделі даних для інформаційної системи?
5. Як впливають структура даних та типи задач предметної області на архітектуру системи?
6. У чому полягає сутність гібридної стратегії (polyglot persistence) при проектуванні сучасних інформаційних систем?

Література

1. Sadalage P.J., Fowler M. NoSQL Distilled : A Brief Guide to the Emerging World of Polyglot Persistence. Upper Saddle River, NJ : Addison-Wesley Professional, 2012. 192 p.
2. Відкриті системи. СУБД №02, 2012. URL: https://books.google.com.ua/books?id=rYFADAAAQBAJ&printsec=frontcover&hl=ru#v=onepage&q&f=false
3. Redmond E., Wilson J. R. Seven Databases in Seven Weeks : A Guide to Modern Databases and the NoSQL Movement. Raleigh, NC : Pragmatic Bookshelf, 2012. 352 p.
4. Stonebraker M., Cattell R. 10 Rules for Scalable Performance in 'Simple Operation' Datastores. Communications of the ACM. 2011. Vol. 54, No. 6. P. 72–80.
5. Brewer E. A. Towards Robust Distributed Systems. Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC '00). New York, NY : ACM, 2000. P. 7.
6. Gilbert S., Lynch N. Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News. 2002. Vol. 33, No. 2. P. 51–59.
7. Gray J., Reuter A. Transaction Processing : Concepts and Techniques. San Francisco, CA : Morgan Kaufmann Publishers, 1992. 1070 p.
8. Karl Seguin, The Little MongoDB Book. [Electronic resource]. 23.04.2012. URL: https://github.com/jsmarkus/the-little-MongoDB-book/blob/master/ru/MongoDB.markdown (date of access: 26.02.2026)
9. MongoDB Documentation [Electronic resource]. URL: https://www.mongodb.com/docs/ (date of access: 26.02.2026).
10. Robinson I., Webber J., Eifrem E. Graph Databases : New Opportunities for Connected Data. 2nd ed. Sebastopol, CA : O'Reilly Media, 2015. 256 p.
11. West D. B. Introduction to Graph Theory. 2nd ed. Upper Saddle River, NJ : Prentice Hall, 2001. 588 p.
12. Neo4j Graph Database Documentation [Electronic resource]. URL: https://neo4j.com/docs/ (date of access: 26.02.2026).
13. Neo4j Cypher Manual [Electronic resource]. URL: https://neo4j.com/docs/cypher-manual/current/ (date of access: 26.03.2026).
14. Kleppmann M. Designing Data-Intensive Applications : The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Sebastopol, CA : O'Reilly Media, 2017. 616 p.
15. Angles R. The Property Graph Database Model. Proceedings of the 12th AMW (Alberto Mendelzon International Workshop on Foundations of Data Management). Cali, Colombia : CEUR Workshop Proceedings, 2018. Vol. 2100. P. 1–11.
16. Girba T., Lanza M. Graph Databases in Software Engineering. Graph-Based Tools (GraBaTs 2004): Proceedings of the 3rd International Workshop. Amsterdam : Elsevier, 2004. Vol. 127, Iss. 1. P.1-12.

ДОДАТОК А
Приклад скрипту створення БД на сервері MongoDB

Відмінності від реляційної моделі:

- денормалізація даних (замість JOIN – вкладені документи)
- відсутність схеми та зовнішніх ключів
- гнучка структура документів
- колекції замість таблиць
- вбудовані масиви для 1:N відношень

Особливості реалізації:

- використовується денормалізація даних (вкладені документи);
- спеціальності та дисципліни зберігаються в окремих колекціях;
- дані про підрозділи вбудовано у документи викладачів;
- оцінки та листи вбудовані у документи студентів;
- використовуються посилання через findOne( ) для зв'язків між колекціями;
- оптимізовано до читання цілісних документів.

Схема колекцій:

```
// ------------------------------------------------------------
// MongoDB create script (balanced schema)
// DB: lectures
// Strict core fields + conditional validation +
flexible details
// Dialect: mongosh
// ------------------------------------------------------------

db = db.getSiblingDB("lectures");
db.dropDatabase();

// Small helpers
const nonEmptyStr = { bsonType: "string",
minLength: 1 };
const posInt = { bsonType: "int", minimum: 1 };
const emailStr = { bsonType: "string", pattern:
"^[^\\s@]+@[^\\s@]+\\.[^\\s@]+$" };

// ------------------------------------------------------------
// 1) Reference dictionaries (enums)
// ------------------------------------------------------------

db.createCollection("publication_types", {
 validator: {
 $jsonSchema: {
 bsonType: "object",
 required: ["code", "label"],
 additionalProperties: false,
 properties: {
 _id: { bsonType: "objectId" },
 code: { ...nonEmptyStr, maxLength: 30 },
        // e.g. "article", "conference", "patent"
 label: { ...nonEmptyStr, maxLength: 200 }
                    // localized/human-readable
 }
 }
 }
});
db.publication_types.createIndex({ code: 1 }, {
unique: true });

db.createCollection("activity_types", {
 validator: {
 $jsonSchema: {
 bsonType: "object",
 required: ["code", "label"],
 additionalProperties: false,
 properties: {
 _id: { bsonType: "objectId" },
 code: { ...nonEmptyStr, maxLength: 30 },
              // e.g. "conference", "committee",
              // "grant", "course"
 label: { ...nonEmptyStr, maxLength: 200 }
 }
 }
 }
});
```

```
db.activity_types.createIndex({ code: 1 }, {
unique: true });

// ------------------------------------------------------------
// 2) Departments (оргструктура):
// всі 3 поля обов’язкові
// ------------------------------------------------------------

db.createCollection("departments", {
 validator: {
 $jsonSchema: {
 bsonType: "object",
 required: ["deptKod", "deptName"],
 additionalProperties: true,
 properties: {
deptKod: posInt,
deptName: { ...nonEmptyStr, maxLength: 100 },
pKod: { bsonType: "int" }
                    // НЕ required, parent dept kod
                    // (якщо корінь – то відсутній)
 }
 }
 }
});
db.departments.createIndex({ deptKod: 1 }, {
unique: true });
db.departments.createIndex({ deptName: 1 }, {
unique: true });

// ------------------------------------------------------------
// 3) Specialties: регламентовані
// Обов’язкові поля: code (напр. F6), name,
// sector, shifr (двухсимвольний код)
// ------------------------------------------------------------

db.createCollection("specialties", {
 validator: {
 $jsonSchema: {
 bsonType: "object",
 required: ["code", "name", "sector", "shifr"],
 additionalProperties: true,
 properties: {
code: { ...nonEmptyStr, maxLength: 10 },
                            // напр. "F6" или "121"
name: { ...nonEmptyStr, maxLength: 200 },
                            // назва спеціальності
sector: { ...nonEmptyStr, maxLength: 200 },
                            // галузь / напрям
shifr: { bsonType: "string", minLength: 2,
maxLength: 2 }
// університетський 2-символьний
 }
 }
 }
});
db.specialties.createIndex({ code: 1 }, { unique:
true });
db.specialties.createIndex({ shifr: 1 }, { unique:
true });

// ------------------------------------------------------------
// 4) Students
// Всегда есть: kod, ПІП, sNum, gNum,
// spec.code, email, address
// Может не быть: letters, ratings, diploma
// Якщо diploma є -> коректні поля
// обов'язкові
// Якщо rating є -> обов'язковий
// discipline + mdate
// ------------------------------------------------------------

db.createCollection("students", {
 validator: {
 $jsonSchema: {
 bsonType: "object",
 required: ["kod", "secondName", "firstName",
"sNum", "gNum", "spec", "email", "address"],
 additionalProperties: true,
 properties: {
kod: posInt,

secondName: { ...nonEmptyStr, maxLength: 60
},
firstName: { ...nonEmptyStr, maxLength: 60 },
patronymic: { bsonType: ["string", "null"],
maxLength: 60 },

sNum: posInt,
gNum: posInt,

spec: {
bsonType: "object",
required: ["code"],
additionalProperties: true,
properties: {
 code: { bsonType: "string", minLength: 2,
maxLength: 2 }, // shifr specialty, напр. "КІ"
 // за бажанням можна денормалізувати:
```

```
  name: { bsonType: ["string", "null"] },
  sector: { bsonType: ["string", "null"] }
}
},

email: emailStr,

address: {
bsonType: "object",
required: ["city", "street", "house", "flat"],
additionalProperties: true,
properties: {
  city: { ...nonEmptyStr, maxLength: 100 },
  street: { ...nonEmptyStr, maxLength: 200 },
  house: { bsonType: "int", minimum: 1 },
  flat: { bsonType: "int", minimum: 1 }
}
},

letters: {
bsonType: ["array", "null"],
items: {
  bsonType: "object",
  required: ["content"],
  additionalProperties: true,
  properties: {
  content: nonEmptyStr,
  date: { bsonType: ["date", "null"] }
  }
}
},

ratings: {
bsonType: ["array", "null"],
items: {
 bsonType: "object",
 required: ["discipline", "mdate", "mark"],
 additionalProperties: true,
 properties: {
 discipline: nonEmptyStr,    // обов'язковий
 mdate: { bsonType: "date" },   // обов'язковий
 mark: { bsonType: "int", minimum: 0,
                                    maximum: 100 },
   // можна зберігати і dKod за бажанням:
  dKod: { bsonType: ["int", "null"], minimum: 1 }
  }
}
},

 diploma: {
 bsonType: ["object", "null"],
 additionalProperties: true,
 required: ["dpKod", "dpName",
"supervisorKod"],
 properties: {
   dpKod: { bsonType: "int", minimum: 1 },
   dpName: nonEmptyStr,
   supervisorKod: { bsonType: "int", minimum: 1
},
   supervisorName: { bsonType: ["string", "null"]
}
 }
 }
  }
  }
  }
});

// ключі/індекси
db.students.createIndex({ kod: 1 }, { unique:
true });
db.students.createIndex({ "spec.code": 1,
gNum: 1, sNum: 1 }, { unique: true });
db.students.createIndex({ "spec.code": 1 });
db.students.createIndex({ secondName: 1 });
db.students.createIndex({ "diploma.dpKod": 1
});
db.students.createIndex({ "ratings.discipline": 1
});

// ------------------------------------------------------------
// 5) Lecturers
// Завжди є: kod, ПІП, job, dept, increments
// Може не бути: diplomas, publications,
// activities
//
// publications[]: кожен елемент повинен
// мати type + year, інше вільно
// activities[]: кожен елемент повинен
// мати тип, решта вільно, details вільний
// піддокумент
//
// type.kind = enum/custom
// enum -> type.code обов'язковий
// (з довідника; MongoDB не перевірить FK,
// але перевірить формат)
// custom -> type.label обов'язковий
// ------------------------------------------------------------
```

```
const typeSchema = {
  bsonType: "object",
  required: ["kind"],
  additionalProperties: true,
  properties: {
  kind: { enum: ["enum", "custom"] },
  code: { bsonType: ["string", "null"],
maxLength: 30 },
  label: { bsonType: ["string", "null"],
maxLength: 200 }
  },
  oneOf: [
  { // enum case
  properties: { kind: { enum: ["enum"] } },
  required: ["code"]
  },
  { // custom case
  properties: { kind: { enum: ["custom"] } },
  required: ["label"]
  }
  ]
};

db.createCollection("lecturers", {
  validator: {
  $jsonSchema: {
  bsonType: "object",
  required: ["kod", "secondName", "firstName",
"job", "dept", "increments"],
  additionalProperties: true,
  properties: {
 kod: posInt,
 secondName: { ...nonEmptyStr, maxLength: 60
},
 firstName: { ...nonEmptyStr, maxLength: 60 },
 patronymic: { bsonType: ["string", "null"],
maxLength: 60 },

 job: { ...nonEmptyStr, maxLength: 100 },

 dept: {
 bsonType: "object",
 required: ["deptKod", "deptName", "pKod"],
 additionalProperties: true,
 properties: {
   deptKod: posInt,
   deptName: { ...nonEmptyStr, maxLength: 100
},
   pKod: posInt
 }
 },

 increments: {
 bsonType: "object",
// в SQL increments могло би бути null,
// проте тут – "завжди є" як об’єкт
 additionalProperties: true,
 properties: {
   longevityInc: { bsonType: ["number", "null"],
                                          minimum: 0 },
   degreeInc: { bsonType: ["number", "null"],
                                          minimum: 0 },
   titleInc: { bsonType: ["number", "null"],
                                          minimum: 0 },
   specialInc: { bsonType: ["number", "null"],
                                          minimum: 0 }
 }
 },

 diplomas: {
 bsonType: ["array", "null"],
 items: {
   bsonType: "object",
   required: ["dpKod", "dpName"],
   additionalProperties: true,
   properties: {
   dpKod: { bsonType: "int", minimum: 1 },
   dpName: nonEmptyStr
   }
 }
 },

 publications: {
 bsonType: ["array", "null"],
 items: {
   bsonType: "object",
   required: ["type", "year", "title"],
   additionalProperties: true,
   properties: {
   type: typeSchema,
   year: { bsonType: "int", minimum: 1900,
                                    maximum: 2100 },
   title: nonEmptyStr,
   // будь-які додаткові поля – вільно:
   details: { bsonType: ["object", "null"] }
   }
 }
```

```
 },

 activities: {
 bsonType: ["array", "null"],
 items: {
  bsonType: "object",
  required: ["type"],
  additionalProperties: true,
  properties: {
  type: typeSchema,
  year: { bsonType: ["int", "null"],
              minimum: 1900, maximum: 2100 },
  name: { bsonType: ["string", "null"] },
  role: { bsonType: ["string", "null"] },
  details: { bsonType: ["object", "null"] }
               // повністю вільний піддокумент
  }
 }
 }
  }
  }
  }
});

// индексы

// ------------------------------------------------------------
print("DB 'lectures' created: balanced
strict+flex schema with conditional type.kind
enum/custom.");
```

Скрипт створення в MongoDB:

```
// Створення колекцій
db.createCollection("activities");
db.createCollection("lecturers")
db.createCollection("students")
db.createCollection("disciplines")
db.createCollection("specialties")

// Індекси для прискорення пошуку
db.lecturers.createIndex({ kod: 1 },
                                  { unique: true });
db.lecturers.createIndex({ secondName: 1,
                                  firstName: 1 });
db.lecturers.createIndex({ "dept.deptKod": 1 });
db.lecturers.createIndex({
                    "publications.type.kind": 1 });
db.lecturers.createIndex({
                    "publications.type.code": 1 });
db.lecturers.createIndex({ "publications.year":
                                                -1 });
db.lecturers.createIndex({
                     "activities.type.kind": 1 });
db.lecturers.createIndex({
                    "activities.type.code": 1 });
db.lecturers.createIndex({ "activities.year":
                                                -1 });
```

ДОДАТОК Б
Приклад JavaScript для заповнення БД на сервері MongoDB через термінал mongo shell

```
// ------------------------------------------------------------
// lectures DB - full insert script (balanced flexible schema)
// Source data: NoSQL DBs MongoDB - examples RDB.docx
// Extra: publication_types + activity_types + partial publications/activities for Malakhov & Pienko
// Dialect: mongosh
// ------------------------------------------------------------

db = db.getSiblingDB("lectures");
// db.dropDatabase();

// 1) Reference dictionaries
db.publication_types.insertMany([
 {
 "code": "article",
 "label": "Стаття у фаховому журналі"
 },
 {
 "code": "conference",
 "label": "Публікація у матеріалах конференції"
 },
 {
 "code": "thesis",
 "label": "Тези доповіді"
 },
 {
 "code": "monograph",
 "label": "Монографія / розділ монографії"
 },
 {
 "code": "patent",
 "label": "Патент / авторське свідоцтво"
 }
]);

db.activity_types.insertMany([
 {
 "code": "editorial_board",
 "label": "Член редакційної колегії"
 },
 {
 "code": "program_committee",
 "label": "Програмний/організаційний комітет конференції"
 },
 {
 "code": "conference_participation",
 "label": "Участь у конференції (доповідач/співавтор)"
 },
 {
 "code": "reviewing",
 "label": "Рецензування для журналу/конференції"
 },
 {
 "code": "council_membership",
 "label": "Членство у наукових/ експертних радах"
 }
]);

// 2) Departments
db.departments.insertMany([
 {
 "deptKod": 1,
 "deptName": "ІМФІТ"
 },
 {
 "deptKod": 2,
 "deptName": "ФІТ",
 "pKod": 1
 },
 {
 "deptKod": 3,
 "deptName": "МЗКС",
 "pKod": 2
 },
 {
 "deptKod": 4,
 "deptName": "КСТ",
 "pKod": 2
 },
 {
```

```
 "deptKod": 5,
 "deptName": "ПМ",
 "pKod": 1
 },
 {
 "deptKod": 6,
 "deptName": "ОКЕК",
 "pKod": 5
 },
 {
 "deptKod": 7,
 "deptName": "МАІТ",
 "pKod": 5
 }
]);

// 3) Specialties (code=government/regulatory
// code, shifr=2-letter university code)
db.specialties.insertMany([
 {
 "code": "E7",
 "name": "Математика",
 "sector": "Природничі науки",
 "shifr": "КМ"
 },
 {
 "code": "F7",
 "name": "Комп’ютерна інженерія",
 "sector": "Інформаційні технології",
 "shifr": "КІ"
 },
 {
 "code": "F6",
 "name": "Інформаційні системи і технології",
 "sector": "Інформаційні технології",
 "shifr": "ІТ"
 },
 {
 "code": "E5",
 "name": "Фізика та астрономія",
 "sector": "Природничі науки",
 "shifr": "ФА"
 }
]);

// 4) Disciplines
db.disciplines.insertMany([
 {
 "dKod": 1,
 "name": "Технології проектування БД",
 "numHours": 60,
 "numSemester": 1,
 "coursework": true,
 "curSemester": 7
 },
 {
 "dKod": 2,
 "name": "Мережеві технології",
 "numHours": 60,
 "numSemester": 1,
 "coursework": true,
 "curSemester": 7
 },
 {
 "dKod": 3,
 "name": "Дослідження операцій",
 "numHours": 30,
 "numSemester": 3,
 "coursework": false,
 "curSemester": 1
 },
 {
 "dKod": 4,
 "name": "Інтелектуальний аналіз даних",
 "numHours": 30,
 "numSemester": 1,
 "coursework": false,
 "curSemester": 5
 },
 {
 "dKod": 5,
 "name": "Програмування",
 "numHours": 60,
 "numSemester": 1,
 "coursework": false,
 "curSemester": 7
 }
]);

// 5) Lecturers
db.lecturers.insertMany([
{
 "kod": 1,
 "secondName": "Малахов",
 "firstName": "Євгеній",
 "patronymic": "Валерійович",
 "job": "Зав.каф",
 "dept": {
```

```
 "deptKod": 3,
 "deptName": "МЗКС",
 "pKod": 2
 },
 "hireDate": ISODate("2012-09-01T00:00:00Z"),
 "salary": 1500.0,
 "increments": {
 "longevityInc": 300.0,
 "degreeInc": 495.0,
 "titleInc": 375.0,
 "specialInc": 675.0
 },
 "diplomas": [
 {
 "dpKod": 9,
 "dpName": "Тема 9"
 },
 {
 "dpKod": 10,
 "dpName": "Тема 10"
 }
 ],
 "publications": [
 {
 "type": {
 "kind": "enum",
 "code": "conference"
 },
 "year": 2024,
 "title": "Oleksandr Penko, Valerii Pienko,
Eugene Malakhov, Multiagent Path Finding
using Dijkstra Algorithm",
 "venue": "Proceedings of IEEE 19th
International Conference on Computer Science
and Information Technologies (CSIT), 16-19
October 2024, Lviv, Ukraine. – прийнято до
публікації.  Sconus"
 },
 {
 "type": {
 "kind": "enum",
 "code": "conference"
 },
 "year": 2024,
 "title": "Irina Kalinina, Aleksandr Gozhyj,
Victoria Vysotska, Eugene Malakhov, Victor
Gozhyj, Iryna Tregubova, System Methodology
of Data Analysis and Pre-processing for Solving
Classification Problems",
 "venue": "Proceedings of IEEE 19th
International Conference on Computer Science
and Information Technologies (CSIT), 16-19
October 2024, Lviv, Ukraine. – прийнято до
публікації.  Sconus"
 },
 {
 "type": {
 "kind": "enum",
 "code": "conference"
 },
 "year": 2024,
 "title": "Bocharova, M., Malakhov, E. (2024).
Improving information theory of context-aware
phrase embeddings in HR domain. Eastern-
European Journal of Enterprise Technologies, 5
(2 (131)), 53–60. https:",
 "venue": "doi.org/10.15587/1729-
4061.2024.313970  Sconus (Q3)"
 }
 ],
 "activities": [
 {
 "type": {
 "kind": "enum",
 "code": "editorial_board"
 },
 "year": 2024,
 "name": "Сенсорна електроніка та
мікросистемні технології",
 "role": "Член редакційної колегії",
 "details": {
 "url": "http://semst.onu.edu.ua"
 }
 },
 {
 "type": {
 "kind": "enum",
 "code": "program_committee"
 },
 "year": 2024,
 "name": "International Conference on
Software and Computer Applications (ICSCA)",
 "role": "program/session chair",
 "details": {
 "period": "2019-2024",
 "country": "Malaysia"
 }
 },
```

```
 {
 "type": {
 "kind": "custom",
 "label": "Член Наукової Ради МОН України,
секції 2 «Інформатика та кібернетика»"
 },
 "year": 2024
 }
 ]
},
{
 "kod": 2,
 "secondName": "Рачинська",
 "firstName": "Алла",
 "patronymic": "Леонідівна",
 "job": "Доц.",
 "dept": {
 "deptKod": 7,
 "deptName": "МАІТ",
 "pKod": 5
 },
 "hireDate": ISODate("1995-09-01T00:00:00Z"),
 "salary": 1100.0,
 "increments": {
 "longevityInc": 220.0,
 "degreeInc": 275.0,
 "titleInc": 165.0,
 "specialInc": 330.0
 },
 "diplomas": [
 {
 "dpKod": 6,
 "dpName": "Тема 6"
 },
 {
 "dpKod": 7,
 "dpName": "Тема 7"
 }
 ]
},
{
 "kod": 3,
 "secondName": "Шпінарева",
 "firstName": "Ірина",
 "patronymic": "Михайлівна",
 "job": "Доц.",
 "dept": {
 "deptKod": 3,
 "deptName": "МЗКС",
 "pKod": 2
 },
 "hireDate": ISODate("2000-09-01T00:00:00Z"),
 "salary": 1250.0,
 "increments": {
 "longevityInc": 250.0,
 "degreeInc": 313.0,
 "titleInc": 188.0,
 "specialInc": 125.0
 },
 "diplomas": [
 {
 "dpKod": 8,
 "dpName": "Тема 8"
 }
 ]
},
{
 "kod": 4,
 "secondName": "Пенко",
 "firstName": "Валерій",
 "patronymic": "Георгійович",
 "job": "Доц.",
 "dept": {
 "deptKod": 3,
 "deptName": "МЗКС",
 "pKod": 2
 },
 "hireDate": ISODate("2000-09-01T00:00:00Z"),
 "salary": 1000.0,
 "increments": {
 "longevityInc": 200.0,
 "degreeInc": 250.0,
 "titleInc": 150.0,
 "specialInc": 390.0
 },
 "diplomas": [
 {
 "dpKod": 1,
 "dpName": "Тема 1"
 },
 {
 "dpKod": 2,
 "dpName": "Тема 2"
 },
 {
 "dpKod": 3,
 "dpName": "Тема 3"
 }
```

```
 ],
 "publications": [
 {
 "type": {
"kind": "enum",
"code": "article"
 },
 "year": 2021,
 "title": "Пенко В.Г., Шпінарева І.М., Ярощук О.В. Діагностика хвороби серця на основі дерева рішень",
 "venue": "\" Інформатика та математичні методи в моделюванні \". Науковий фаховий журнал. Том 11, №2.  - Одеса, ОНПУ. 2021, с.112-119"
 },
 {
 "type": {
"kind": "enum",
"code": "article"
 },
 "year": 2018,
 "title": "V. Penko, Gafar Abdula I. Approach to identifying plagiarism in multilingual texts",
 "venue": "\"Информатика и математические методы в моделировании\". Научный журнал. Том 8, №2.  - Одеса, ОНПУ. 2018, с.121-128"
 },
 {
 "type": {
"kind": "enum",
"code": "thesis"
 },
 "year": 2021,
 "title": "Пенко В.Г., Пенко О.В. Підхід до реалізації системи прийняття рішень для оптимізації управління на регіональному рівні",
 "venue": "Матеріали щорічної науково – практичної конференції «Удосконалення економічних механізмів розвитку територій», Одеський регіональний інститут державного управління Національної академії державного управління при Президентові України - 28 травня 2021 р"
 }
 ],
 "activities": [
 {
 "type": {
"kind": "enum",
"code": "conference_participation"
 },
 "year": 2018,
 "name": "CAIT-Odessa-2018",
 "role": "Доповідач/співавтор",
 "details": {
"city": "Одеса",
"country": "Україна"
 }
 },
 {
 "type": {
"kind": "custom",
"label": "Участь у щорічній науково-практичній конференції «Удосконалення економічних механізмів розвитку територій»"
 },
 "year": 2021,
 "details": {
"city": "Одеса"
 }
 }
 ]
},
{
 "kod": 5,
 "secondName": "Гунченко",
 "firstName": "Юрій",
 "patronymic": "Олександрович",
 "job": "Проф.",
 "dept": {
"deptKod": 4,
"deptName": "КСТ",
"pKod": 2
 },
 "hireDate": ISODate("2015-09-01T00:00:00Z"),
 "salary": 1300.0,
 "increments": {
"longevityInc": 260.0,
"degreeInc": 429.0,
"titleInc": 325.0,
"specialInc": 520.0
 },
 "diplomas": [
 {
```

```
 "dpKod": 4,
 "dpName": "Тема 4"
 },
 {
 "dpKod": 5,
 "dpName": "Тема 5"
 }
 ]
}
]);

// 6) Students
db.students.insertMany([
{
 "kod": 1,
 "secondName": "Бровков",
 "firstName": "Володимир",
 "patronymic": "Георгійович",
 "sNum": 3,
 "gNum": 971,
 "spec": {
 "code": "КМ",
 "name": "Математика",
 "sector": "Природничі науки",
 "govCode": "E7"
 },
 "email": "bvgo12345@ukr.net",
 "address": {
 "city": "м. Одеса",
 "street": "вул. Невідомого",
 "house": 5,
 "flat": 1
 },
 "letters": [
 {
 "ltKod": 4,
 "content": "Зателефонуйте мені за номером
(777) 777-77-77"
 }
 ],
 "ratings": [
 {
 "dKod": 1,
 "discipline": "Технології проектування БД",
 "mark": 82,
 "mdate": ISODate("2024-10-10T00:00:00Z")
 },
 {
 "dKod": 2,
 "discipline": "Мережеві технології",
 "mark": 10,
 "mdate": ISODate("2024-10-12T00:00:00Z")
 },
 {
 "dKod": 5,
 "discipline": "Програмування",
 "mark": 75,
 "mdate": ISODate("2024-10-11T00:00:00Z")
 }
 ],
 "diploma": {
 "dpKod": 9,
 "dpName": "Тема 9",
 "supervisorKod": 1,
 "supervisorName": "Малахов Євгеній"
 }
},
{
 "kod": 2,
 "secondName": "Арсірій",
 "firstName": "Олена",
 "patronymic": "Олександрівна",
 "sNum": 17,
 "gNum": 943,
 "spec": {
 "code": "IT",
 "name": "Інформаційні системи і технології",
 "sector": "Інформаційні технології",
 "govCode": "F6"
 },
 "email": "arsiriy@gmail.com",
 "address": {
 "city": "м. Одеса",
 "street": "вул. Друга",
 "house": 3,
 "flat": 1
 },
 "letters": [
 {
 "ltKod": 1,
 "content": "В завданні №4 відповідь 12.24"
 }
 ],
 "ratings": [
 {
 "dKod": 1,
 "discipline": "Технології проектування БД",
 "mark": 90,
```

```
  "mdate": ISODate("2024-10-10T00:00:00Z")
  },
  {
  "dKod": 2,
  "discipline": "Мережеві технології",
  "mark": 76,
  "mdate": ISODate("2024-10-12T00:00:00Z")
  },
  {
  "dKod": 5,
  "discipline": "Програмування",
  "mark": 20,
  "mdate": ISODate("2024-10-11T00:00:00Z")
  }
  ],
  "diploma": {
  "dpKod": 1,
  "dpName": "Тема 1",
  "supervisorKod": 4,
  "supervisorName": "Пенко Валерій"
  }
},
{
  "kod": 3,
  "secondName": "Любченко",
  "firstName": "Віра",
  "patronymic": "Вікторівна",
  "sNum": 3,
  "gNum": 102,
  "spec": {
  "code": "КМ",
  "name": "Математика",
  "sector": "Природничі науки",
  "govCode": "E7"
  },
  "email": "lubchenko@te.net.ua",
  "address": {
  "city": "м. Одеса",
  "street": "вул. 1-го травня",
  "house": 5,
  "flat": 1
  },
  "letters": [
  {
  "ltKod": 3,
  "content": "Збір біля актового залу о 17:00"
  }
  ],
  "ratings": [
  {
  "dKod": 1,
  "discipline": "Технології проектування БД",
  "mark": 46,
  "mdate": ISODate("2024-10-11T00:00:00Z")
  },
  {
  "dKod": 2,
  "discipline": "Мережеві технології",
  "mark": 85,
  "mdate": ISODate("2024-10-12T00:00:00Z")
  },
  {
  "dKod": 5,
  "discipline": "Програмування",
  "mark": 100,
  "mdate": ISODate("2024-10-11T00:00:00Z")
  }
  ],
  "diploma": {
  "dpKod": 3,
  "dpName": "Тема 3",
  "supervisorKod": 4,
  "supervisorName": "Пенко Валерій"
  }
},
{
  "kod": 4,
  "secondName": "Філатова",
  "firstName": "Тетяна",
  "patronymic": "В’ячеславівна",
  "sNum": 15,
  "gNum": 943,
  "spec": {
  "code": "IT",
  "name": "Інформаційні системи і технології",
  "sector": "Інформаційні технології",
  "govCode": "F6"
  },
  "email": "filatova@gmail.com",
  "address": {
  "city": "м. Одеса",
  "street": "вул. Невідомого",
  "house": 15,
  "flat": 1
  },
  "ratings": [
  {
  "dKod": 1,
```

```
 "discipline": "Технології проектування БД",
 "mark": 54,
 "mdate": ISODate("2024-10-10T00:00:00Z")
 },
 {
 "dKod": 2,
 "discipline": "Мережеві технології",
 "mark": 85,
 "mdate": ISODate("2024-12-12T00:00:00Z")
 },
 {
 "dKod": 5,
 "discipline": "Програмування",
 "mark": 65,
 "mdate": ISODate("2024-10-11T00:00:00Z")
 }
 ],
 "diploma": {
 "dpKod": 2,
 "dpName": "Тема 2",
 "supervisorKod": 4,
 "supervisorName": "Пенко Валерій"
 }
},
{
 "kod": 5,
 "secondName": "Журан",
 "firstName": "Олена",
 "patronymic": "Анатоліївна",
 "sNum": 4,
 "gNum": 971,
 "spec": {
 "code": "ФА",
 "name": "Фізика та астрономія",
 "sector": "Природничі науки",
 "govCode": "E5"
 },
 "email": "zhuran@ukr.net",
 "address": {
 "city": "м. Одеса",
 "street": "вул. Пересічна",
 "house": 45,
 "flat": 4
 },
 "letters": [
 {
 "ltKod": 2,
 "content": "Від’їзд на конференцію
відбудеться о 8:00 am"
 }
 ],
 "ratings": [
 {
 "dKod": 1,
 "discipline": "Технології проектування БД",
 "mark": 86,
 "mdate": ISODate("2024-10-10T00:00:00Z")
 }
 ],
 "diploma": {
 "dpKod": 5,
 "dpName": "Тема 5",
 "supervisorKod": 5,
 "supervisorName": "Гунченко Юрій"
 }
},
{
 "kod": 6,
 "secondName": "Мікулінська",
 "firstName": "Марія",
 "patronymic": "Геннадіївна",
 "sNum": 5,
 "gNum": 970,
 "spec": {
 "code": "IT",
 "name": "Інформаційні системи і технології",
 "sector": "Інформаційні технології",
 "govCode": "F6"
 },
 "email": "mmg12345@te.net.ua",
 "address": {
 "city": "м. Одеса",
 "street": "вул. Рожева",
 "house": 67,
 "flat": 8
 },
 "ratings": [
 {
 "dKod": 3,
 "discipline": "Дослідження операцій",
 "mark": 30,
 "mdate": ISODate("2024-10-14T00:00:00Z")
 }
 ],
 "diploma": {
 "dpKod": 10,
 "dpName": "Тема 10",
 "supervisorKod": 1,
```

```
  "supervisorName": "Малахов Євгеній"
  }
},
{
  "kod": 7,
  "secondName": "Блажко",
  "firstName": "Олександр",
  "patronymic": "Анатолійович",
  "sNum": 12,
  "gNum": 954,
  "spec": {
  "code": "КІ",
  "name": "Комп’ютерна інженерія",
  "sector": "Інформаційні технології",
  "govCode": "F7"
  },
  "email": "blazko@ukr.net",
  "address": {
  "city": "м. Одеса",
  "street": "вул. Місцева",
  "house": 2,
  "flat": 4
  },
  "letters": [
  {
  "ltKod": 6,
  "content": "Дайте відповідь за адресою
onu@onu.ua"
  }
  ],
  "ratings": [
  {
  "dKod": 4,
  "discipline": "Інтелектуальний аналіз даних",
  "mark": 97,
  "mdate": ISODate("2024-10-13T00:00:00Z")
  }
  ]
},
{
  "kod": 8,
  "secondName": "Філипова",
  "firstName": "Світлана",
  "patronymic": "Валеріївна",
  "sNum": 11,
  "gNum": 951,
  "spec": {
  "code": "ФА",
  "name": "Фізика та астрономія",
  "sector": "Природничі науки",
  "govCode": "E5"
  },
  "email": "filyppova@gmail.com",
  "address": {
  "city": "м. Одеса",
  "street": "вул. Крайня",
  "house": 56,
  "flat": 9
  },
  "letters": [
  {
  "ltKod": 5,
  "content": "Мій контактний телефон +38
(050) 111-11-11"
  }
  ],
  "ratings": [
  {
  "dKod": 1,
  "discipline": "Технології проектування БД",
  "mark": 0,
  "mdate": ISODate("2024-10-10T00:00:00Z")
  }
  ]
}
]);

// Sanity checks
print("departments:",
db.departments.countDocuments());
print("specialties:",
db.specialties.countDocuments());
print("disciplines:",
db.disciplines.countDocuments());
print("lecturers:",
db.lecturers.countDocuments());
print("students:",
db.students.countDocuments());
```

ДОДАТОК В
Визначення найпоширеніших запитів в прикладній базі даних MongoDB

Для виконання цього аналізу знадобляться інструменти для роботи з базою даних: консоль mongosh або середовище MongoDB Compass.

Метод 1. Використання MongoDB Profiler (system.profile)

Профайлер збирає дані про операції бази даних і зберігає їх у спеціальну колекцію system.profile у тій самій базі даних, для якої він увімкнений.

Крок 1. Увімкнення профайлеру

За промовчанням профайлер вимкнено (рівень 0). Щоб визначити найпоширеніші (найчастіші) запити, потрібно тимчасово включити профіль усіх операцій (рівень 2) або лише повільних (рівень 1).

У консолі треба вибрати базу даних і встановити рівень журналювання:

```
use <database_name>;

// Включаємо рівень 2 (збирає ВСІ запити).
// Увага! На високонавантаженому production це може сповільнити роботу,
// тому краще вмикати на короткий час!
db.setProfilingLevel(2);

// або включаємо рівень 1 (збирає лише запити, що виконуються довше 100 мс)
db.setProfilingLevel(1, { slowms: 100 });
```

Крок 2. Збирання статистики

Прикладна програма повинна попрацювати 5–15 хвилин, щоб профайлер зібрав репрезентативну статистику в system.profile.

Крок 3. Пошуку найчастіших операцій

Тепер можна звернутись до колекції `system.profile` і за допомогою агрегації згрупувати запити, щоб дізнатися, які з них найчастіше виконуються, за допомогою запиту:

```
db.system.profile.aggregate([
// Виключаємо службові запити самого профайлера
{ $match: { op: { $ne: "command" }, ns: { $not: /\.system\.profile$/ } } },

// Групуємо за типом операції (op) і простору імен (ns - колекція)
{ $group: {
_id: { operation: "$op", collection: "$ns" },
count: { $sum: 1 },                          // Рахуємо кількість
avgTime: { $avg: "$
millis" },                                   // Середній час виконання max$
```

```
}},

// Сортуємо за зменшенням частоти (найчастіші зверху)
{ $sort: { count: -1 } },

// Беремо, наприклад, топ-10
{$limit: 10}
]);
```

Цей запит покаже, до яких колекцій та які типи операцій (query, insert, update) відбуваються найчастіше, а також їхній середній час виконання.

Крок 4. Вимкнення профайлеру

Після збору даних треба вимкнути профайлер, щоб він не витрачав ресурси CPU та місце на диску.

```
db.setProfilingLevel(0);
```

Метод 2. Аналіз slow queries у журналах (логах)

Для використання цього методу необхідний доступ до файлу журналу (лог-файлу) з ім'ям `mongod.log`, який стандартно розташований:

- **Ubuntu/Debian:** /var/log/mongodb/mongod.log
- **RedHat/CentOS:** /var/log/mongo/mongod.log
- **Windows:** C:\Program Files\MongoDB\Server\XX\log\mongod.log (або <шлях_до_БД>\log\mongod.log)
- **macOS (Homebrew):** /usr/local/var/log/mongodb/mongo.log (або /opt/homebrew/var/log/mongodb/mongo.log)

При **аналізі логів** файл журналу можна переглядати вручну, але для виявлення найчастіших повільних запитів краще використовувати спеціалізовані інструменти, наприклад, утиліти mtools **(**зокрема, mloginfo**)**.

Для цього спочатку встановіть mtools через pip:

```
pip install mtools
```

А ддалі запустіть аналіз запитів за лог-файлом:

```
mloginfo /var/log/mongodb/mongod.log --queries
```

Ця команда розпарсує лог-файл, згрупує схожі запити та поверне таблицю із зазначенням того, які запити зустрічаються в лозі найчастіше (count), їхні патерни та середній час виконання.

Висновок-рекомендація щодо вибору методу

Якщо треба знайти **взагалі всі** найчастіші запити (навіть швидкі) – використовуйте **Profiler (рівень 2)** на короткий час.

Якщо треба знайти найчастіші з **проблемних/повільних** запитів, використовуйте парсинг **логів через mloginfo** або Profiler на рівні 1.

ДОДАТОК Г
Скрипт створення та заповнення БД на сервері Neo4j мовою Cypher

Відмінності від реляційної моделі:
- дані як вузли та відношення замість таблиць;
- властивості зберігаються на вузлах та відношеньах;
- відсутність явних зовнішніх ключів (зв'язки – першокласні сутності);
- оптимізація для зв'язкових запитів (шляхи, глибинні зв'язки).

Особливості реалізації
- використовуються вузли для всіх сутностей та відношення для зв'язків;
- кожен тип сутності має свої властивості;
- відносшення можуть мати властивості (наприклад, оцінки зберігаються як властивості відношень);
- листи створені як окремі вузли з відношеннями до студентів;
- використовуються складні шаблони MATCH-WHERE-CREATE для встановлення зв'язків;
- оптимізовано для запитів, які використовують зв'язок між сутностями.

Схема вузлів та відношень:

Навчання та організація

```
(Student) - [:STUDIES_SPECIALTY] -> (Specialty)
(Student) - [:IN_GROUP] -> (Group)
(Student) - [:LIVES_AT] -> (Address)
```

Оцінювання (з підтримкою кількох спроб)

```
(Student) - [:HAS_GRADE] -> (GradeAttempt)
(GradeAttempt) - [:FOR_DISCIPLINE] -> (Discipline) // GradeAttempt має
                                  // властивості: mark, mdate (дата спроби/оцінювання)
```

Дипломи та керівництво

```
(Student) - [:HAS_DIPLOMA_TOPIC] -> (DiplomaTopic)
(Student) - [:SUPERVISED_BY] -> (Lecturer)
(Lecturer) - [:SUPERVISES] -> (DiplomaTopic)
(Lecturer) - [:CAN_SUPERVISE] -> (DiplomaTopic) // "може керувати темою"
                                  // (з переліку тем, закріплених за викладачем)
```

Структура підрозділів та працевлаштування

```
(Lecturer) - [:WORKS_IN] -> (Department)
(Department) - [:SUBDIVISION_OF] -> (Department)  // Ієрархія підрозділів
                                  // (кафедра → факультет → інститут тощо)
```

Повідомлення/листи студентів

```
(Student) - [:HAS_LETTER] -> (Letter)          // Letter має властивості:
                                  // ltKod, content, sentAt
```

Наукова діяльність: публікації, ключові слова, цитування

(Lecturer) - [:AUTHORED] -> (Publication)
(Publication) - [:OF_TYPE] -> (PublicationType)
(Publication) - [:HAS_KEYWORD] -> (Keyword)
(Publication) - [:CITES] -> (Publication) // Орієнтований граф цитувань

Співавторство

(Lecturer) - [:COAUTHOR_WITH] -> (Lecturer) // Похідний зв'язок
// (обчислюється з "shared" публікацій),
// може мати властивість sharedPublications

Активності викладачів

(Lecturer) - [:HAS_ACTIVITY] -> (Activity)
(Activity) - [:OF_TYPE] -> (ActivityType)

Скрипт створення та заповнення БД в Neo4j (Cypher):

```
// ================================================================
// lectures graph DB (Neo4j) – build script v3.1.3.5
// ================================================================

// 0) Clean database
MATCH (n) DETACH DELETE n;

// 1) Constraints (Neo4j 5+)
CREATE CONSTRAINT dept_deptKod IF NOT EXISTS FOR (d:Department) REQUIRE d.deptKod IS UNIQUE;
CREATE CONSTRAINT spec_code IF NOT EXISTS FOR (s:Specialty) REQUIRE s.code IS UNIQUE;
CREATE CONSTRAINT disc_dKod IF NOT EXISTS FOR (d:Discipline) REQUIRE d.dKod IS UNIQUE;
CREATE CONSTRAINT lect_kod IF NOT EXISTS FOR (l:Lecturer) REQUIRE l.kod IS UNIQUE;
CREATE CONSTRAINT stud_kod IF NOT EXISTS FOR (s:Student) REQUIRE s.kod IS UNIQUE;
CREATE CONSTRAINT grp_gNum IF NOT EXISTS FOR (g:Group) REQUIRE g.gNum IS UNIQUE;
CREATE CONSTRAINT pubType_code IF NOT EXISTS FOR (pt:PublicationType) REQUIRE pt.code IS UNIQUE;
CREATE CONSTRAINT actType_code IF NOT EXISTS FOR (at:ActivityType) REQUIRE at.code IS UNIQUE;
CREATE CONSTRAINT pub_pubId IF NOT EXISTS FOR (p:Publication) REQUIRE p.pubId IS UNIQUE;
CREATE CONSTRAINT kw_value IF NOT EXISTS FOR (k:Keyword) REQUIRE k.value IS UNIQUE;
CREATE CONSTRAINT diploma_dpKod IF NOT EXISTS FOR (t:DiplomaTopic) REQUIRE t.dpKod IS UNIQUE;
CREATE CONSTRAINT letter_id IF NOT EXISTS FOR (m:Letter) REQUIRE m.id IS UNIQUE;
CREATE CONSTRAINT activity_id IF NOT EXISTS FOR (a:Activity) REQUIRE a.id IS UNIQUE;
CREATE CONSTRAINT grade_id IF NOT EXISTS FOR (ga:GradeAttempt) REQUIRE ga.id IS UNIQUE;
CREATE CONSTRAINT address_id IF NOT EXISTS FOR (a:Address) REQUIRE a.id IS UNIQUE;

// 2) Reference dictionaries

// 2.1 Publication types
UNWIND [
 {code: "article", label: "Стаття у фаховому журналі"},
 {code: "conference", label: "Публікація у матеріалах конференції"},
 {code: "thesis", label: "Тези доповіді"},
 {code: "monograph", label: "Монографія / розділ монографії"},
 {code: "patent", label: "Патент / авторське свідоцтво"}
] AS row
```

```
MERGE (pt:PublicationType {code: row.code})
SET pt.label = row.label;

// 2.2 Activity types
UNWIND [
 {code: "editorial_board", label: "Член редакційної колегії"},
 {code: "program_committee", label: "Програмний/організаційний комітет конференції"},
 {code: "conference_participation", label: "Участь у конференції (доповідач/співавтор)"},
 {code: "reviewing", label: "Рецензування для журналу/конференції"},
 {code: "council_membership", label: "Членство у наукових/експертних радах"}
] AS row
MERGE (at:ActivityType {code: row.code})
SET at.label = row.label;

// 3) Departments + hierarchy
UNWIND [
 {deptKod: 1, deptName: "ІМФІТ"},
 {deptKod: 2, deptName: "ФІТ", pKod: 1},
 {deptKod: 3, deptName: "МЗКС", pKod: 2},
 {deptKod: 4, deptName: "КСТ", pKod: 2},
 {deptKod: 5, deptName: "ПМ", pKod: 1},
 {deptKod: 6, deptName: "ОКЕК", pKod: 5},
 {deptKod: 7, deptName: "МАІТ", pKod: 5}
] AS row
MERGE (d:Department {deptKod: row.deptKod})
SET d.deptName = row.deptName, d.pKod = row.pKod;

// Department hierarchy: (child)-[:SUBDIVISION_OF]->(parent)
MATCH (child:Department)
WHERE child.pKod IS NOT NULL
MATCH (parent:Department {deptKod: child.pKod})
MERGE (child)-[:SUBDIVISION_OF]->(parent);

// 4) Specialties
UNWIND [
 {code: "КМ", govCode: "Е7", name: "Математика", sector: "Природничі науки"},
 {code: "КІ", govCode: "F7", name: "Комп’ютерна інженерія", sector: "Інформаційні технології"},
 {code: "ІТ", govCode: "F6", name: "Інформаційні системи і технології", sector: "Інформаційні технології"},
 {code: "ФА", govCode: "Е5", name: "Фізика та астрономія", sector: "Природничі науки"}
] AS row
MERGE (s:Specialty {code: row.code})
SET s.govCode = row.govCode, s.name = row.name, s.sector = row.sector;

// 5) Disciplines
UNWIND [
 {dKod: 1, name: "Технології проектування БД", numHours: 60, numSemester: 1, coursework: true, curSemester: 7},
 {dKod: 2, name: "Мережеві технології", numHours: 60, numSemester: 1, coursework: true, curSemester: 7},
 {dKod: 3, name: "Дослідження операцій", numHours: 30, numSemester: 3, coursework: false, curSemester: 1},
 {dKod: 4, name: "Інтелектуальний аналіз даних", numHours: 30, numSemester: 1, coursework: false, curSemester: 5},
 {dKod: 5, name: "Програмування", numHours: 60, numSemester: 1, coursework: false, curSemester: 7}
] AS row
MERGE (d:Discipline {dKod: row.dKod})
SET d.name = row.name, d.numHours = row.numHours, d.numSemester = row.numSemester,
   d.coursework = row.coursework, d.curSemester = row.curSemester;

// 6) Lecturers + WORKS_IN + CAN_SUPERVISE
UNWIND [
 {kod: 1, secondName: "Малахов", firstName: "Євгеній", patronymic: "Валерійович", job: "Зав.каф", hireDate: "2012-09-01T00:00:00Z", salary: 1500.0, deptKod: 3, increments: {longevityInc: 300.0, degreeInc: 495.0, titleInc: 375.0, specialInc: 675.0}, diplomas: [{dpKod: 9, dpName: "Тема 9"}, {dpKod: 10, dpName: "Тема 10"}]},
 {kod: 2, secondName: "Рачинська", firstName: "Алла", patronymic: "Леонідівна", job: "Доц.", hireDate: "1995-09-01T00:00:00Z", salary: 1100.0, deptKod: 7, increments: {longevityInc: 220.0, degreeInc: 275.0, titleInc: 165.0, specialInc: 330.0}, diplomas: [{dpKod: 6, dpName: "Тема 6
```

```
"}, {dpKod: 7, dpName: "Тема 7"}]},
  {kod: 3, secondName: "Шпінарева", firstName
: "Ірина", patronymic: "Михайлівна", job: "До
ц.", hireDate: "2000-09-01T00:00:00Z", salary:
1250.0, deptKod: 3, increments: {longevityInc:
250.0, degreeInc: 313.0, titleInc: 188.0, speciall
nc: 125.0}, diplomas: [{dpKod: 8, dpName: "Тем
а 8"}]},
  {kod: 4, secondName: "Пенко", firstName: "Ва
лерій", patronymic: "Георгійович", job: "Доц."
, hireDate: "2000-09-01T00:00:00Z", salary: 100
0.0, deptKod: 3, increments: {longevityInc: 200.
0, degreeInc: 250.0, titleInc: 150.0, specialInc: 3
90.0}, diplomas: [{dpKod: 1, dpName: "Тема 1"}
, {dpKod: 2, dpName: "Тема 2"}, {dpKod: 3, dpN
ame: "Тема 3"}]},
  {kod: 5, secondName: "Гунченко", firstName:
"Юрій", patronymic: "Олександрович", job: "П
роф.", hireDate: "2015-09-01T00:00:00Z", salar
y: 1300.0, deptKod: 4, increments: {longevityInc
: 260.0, degreeInc: 429.0, titleInc: 325.0, special
Inc: 520.0}, diplomas: [{dpKod: 4, dpName: "Те
ма 4"}, {dpKod: 5, dpName: "Тема 5"}]}
] AS row
MERGE (l:Lecturer {kod: row.kod})
SET l.secondName = row.secondName,
    l.firstName  = row.firstName,
    l.patronymic = row.patronymic,
    l.job        = row.job,
    l.hireDate   = date(substring(row.hireDate, 0,
10)),
    l.salary     = row.salary,
    l.longevityInc = coalesce(row.increments.long
evityInc, 0),
    l.degreeInc    = coalesce(row.increments.degr
eeInc, 0),
    l.titleInc     = coalesce(row.increments.titleInc
, 0),
    l.specialInc   = coalesce(row.increments.speci
alInc, 0)
WITH l, row
MATCH (d:Department {deptKod: row.deptKod}
)
MERGE (l)-[:WORKS_IN]->(d)
WITH l, row
UNWIND coalesce(row.diplomas, []) AS dp
MERGE (t:DiplomaTopic {dpKod: dp.dpKod})
SET t.dpName = dp.dpName
MERGE (l)-[:CAN_SUPERVISE]->(t);

// 7) Students
//    + Address as node
//    + GradeAttempt nodes for multiple
// grades per discipline
UNWIND [
  {kod: 1, secondName: "Бровков", firstName: "
Володимир", patronymic: "Георгійович", sNu
m: 3, gNum: 971, specCode: "КМ", email: "bvgo
12345@ukr.net", address: {id: "м. Одеса|вул.
Невідомого|5|1", city: "м. Одеса", street: "ву
л. Невідомого", house: 5, flat: 1}, letters: [{ltKo
d: 4, content: "Зателефонуйте мені за номеро
м (777) 777-77-77"}], ratings: [{dKod: 1, mark: 8
2, mdate: "2024-10-10T00:00:00Z"}, {dKod: 2,
mark: 10, mdate: "2024-10-12T00:00:00Z"}, {dK
od: 5, mark: 75, mdate: "2024-10-11T00:00:00Z
"}], diploma: {dpKod: 9, dpName: "Тема 9", sup
ervisorKod: 1, supervisorName: "Малахов Євге
ній"}},
  {kod: 2, secondName: "Арсірій", firstName: "О
лена", patronymic: "Олександрівна", sNum: 1
7, gNum: 943, specCode: "IT", email: "arsiriy@g
mail.com", address: {id: "м. Одеса|вул. Друга|
3|1", city: "м. Одеса", street: "вул. Друга", hou
se: 3, flat: 1}, letters: [{ltKod: 1, content: "В зав
данні №4 відповідь 12.24"}], ratings: [{dKod: 1
, mark: 90, mdate: "2024-10-10T00:00:00Z"}, {d
Kod: 2, mark: 76, mdate: "2024-10-12T00:00:00
Z"}, {dKod: 5, mark: 20, mdate: "2024-10-11T00
:00:00Z"}], diploma: {dpKod: 1, dpName: "Тема
1", supervisorKod: 4, supervisorName: "Пенко
Валерій"}},
  {kod: 3, secondName: "Любченко", firstName:
"Віра", patronymic: "Вікторівна", sNum: 3, gNu
m: 102, specCode: "КМ", email: "lubchenko@te
.net.ua", address: {id: "м. Одеса|вул. 1-го трав
ня|5|1", city: "м. Одеса", street: "вул. 1-го тра
вня", house: 5, flat: 1}, letters: [{ltKod: 3, conte
nt: "Збір біля актового залу о 17:00"}], ratings:
[{dKod: 1, mark: 46, mdate: "2024-10-11T00:00
:00Z"}, {dKod: 2, mark: 85, mdate: "2024-10-12
T00:00:00Z"}, {dKod: 5, mark: 100, mdate: "202
4-10-11T00:00:00Z"}], diploma: {dpKod: 3, dpN
ame: "Тема 3", supervisorKod: 4, supervisorNa
me: "Пенко Валерій"}},
  {kod: 4, secondName: "Філатова", firstName:
"Тетяна", patronymic: "В'ячеславівна", sNum:
15, gNum: 943, specCode: "IT", email: "filatova
```

```
@gmail.com", address: {id: "м. Одеса|вул. Невідомого|15|1", city: "м. Одеса", street: "вул. Невідомого", house: 15, flat: 1}, letters: [], ratings: [{dKod: 1, mark: 54, mdate: "2024-10-10T00:00:00Z"}, {dKod: 2, mark: 85, mdate: "2024-12-12T00:00:00Z"}, {dKod: 5, mark: 65, mdate: "2024-10-11T00:00:00Z"}], diploma: {dpKod: 2, dpName: "Тема 2", supervisorKod: 4, supervisorName: "Пенко Валерій"}},
  {kod: 5, secondName: "Журан", firstName: "Олена", patronymic: "Анатоліївна", sNum: 4, gNum: 971, specCode: "ФА", email: "zhuran@ukr.net", address: {id: "м. Одеса|вул. Пересічна|45|4", city: "м. Одеса", street: "вул. Пересічна", house: 45, flat: 4}, letters: [{ltKod: 2, content: "Від’їзд на конференцію відбудеться о 8:00 am"}], ratings: [{dKod: 1, mark: 86, mdate: "2024-10-10T00:00:00Z"}], diploma: {dpKod: 5, dpName: "Тема 5", supervisorKod: 5, supervisorName: "Гунченко Юрій"}},
  {kod: 6, secondName: "Мікулінська", firstName: "Марія", patronymic: "Геннадіївна", sNum: 5, gNum: 970, specCode: "IT", email: "mmg12345@te.net.ua", address: {id: "м. Одеса|вул. Рожева|67|8", city: "м. Одеса", street: "вул. Рожева", house: 67, flat: 8}, letters: [], ratings: [{dKod: 3, mark: 30, mdate: "2024-10-14T00:00:00Z"}], diploma: {dpKod: 10, dpName: "Тема 10", supervisorKod: 1, supervisorName: "Малахов Євгеній"}},
  {kod: 7, secondName: "Блажко", firstName: "Олександр", patronymic: "Анатолійович", sNum: 12, gNum: 954, specCode: "КІ", email: "blazko@ukr.net", address: {id: "м. Одеса|вул. Місцева|2|4", city: "м. Одеса", street: "вул. Місцева", house: 2, flat: 4}, letters: [{ltKod: 6, content: "Дайте відповідь за адресою opu@opu.ua"}], ratings: [{dKod: 4, mark: 97, mdate: "2024-10-13T00:00:00Z"}], diploma: null},
  {kod: 8, secondName: "Філиппова", firstName: "Світлана", patronymic: "Валеріївна", sNum: 11, gNum: 951, specCode: "ФА", email: "filyppova@gmail.com", address: {id: "м. Одеса|вул. Крайня|56|9", city: "м. Одеса", street: "вул. Крайня", house: 56, flat: 9}, letters: [{ltKod: 5, content: "Мій контактний телефон +38 (050) 111-11-11"}], ratings: [{dKod: 1, mark: 0, mdate: "2024-10-10T00:00:00Z"}], diploma: null}
] AS row

MERGE (s:Student {kod: row.kod})
SET s.secondName = row.secondName,
   s.firstName  = row.firstName,
   s.patronymic = row.patronymic,
   s.sNum       = row.sNum,
   s.gNum       = row.gNum,
   s.email      = row.email

// group + specialty
MERGE (g:Group {gNum: row.gNum})
MERGE (s)-[:IN_GROUP]->(g)
WITH s, row
MATCH (sp:Specialty {code: row.specCode})
MERGE (s)-[:STUDIES_SPECIALTY]->(sp)

// address as node
WITH s, row
FOREACH (_ IN CASE WHEN row.address IS NULL THEN [] ELSE [1] END |
  MERGE (a:Address {id: row.address.id})
  SET a.city = row.address.city, a.street = row.address.street, a.house = row.address.house, a.flat = row.address.flat
  MERGE (s)-[:LIVES_AT]->(a)
)

// letters as nodes (1 student – many letters)
WITH s, row
UNWIND coalesce(row.letters, []) AS lt
MERGE (m:Letter {id: toString(row.kod) + ':' + toString(lt.ltKod)})
SET m.ltKod   = lt.ltKod,
   m.content = lt.content,
   // deterministic pseudo-timestamp for demo
   // datasets: 2024-01-01T00:00Z + ltKod days
   m.sentAt  = datetime('2024-01-01T00:00:00Z') + duration({days: toInteger(lt.ltKod)})
MERGE (s)-[:HAS_LETTER]->(m)

// grades as GradeAttempt nodes:
// allow multiple attempts per discipline
WITH s, row
UNWIND coalesce(row.ratings, []) AS r
MATCH (d:Discipline {dKod: r.dKod})
MERGE (ga:GradeAttempt {id: toString(s.kod) + ':' + toString(r.dKod) + ':' + substring(r.mdate,0,10)})
SET ga.mark = r.mark, ga.mdate = date(substring(r.mdate,0,10))
```

```
MERGE (s)-[:HAS_GRADE]->(ga)
MERGE (ga)-[:FOR_DISCIPLINE]->(d)

// diploma: Student -> DiplomaTopic,
// supervised by Lecturer
// v3.1.3: compatible pattern without
// subquery importing WHERE.
// We pre-bind optional supervisor and then
// use FOREACH with only write clauses.
WITH s, row
OPTIONAL MATCH (sup:Lecturer {kod: row.diplo
ma.supervisorKod})
FOREACH (_ IN CASE WHEN row.diploma IS NUL
L THEN [] ELSE [1] END |
  MERGE (t:DiplomaTopic {dpKod: row.diploma.
dpKod})
  SET t.dpName = row.diploma.dpName
  MERGE (s)-[:HAS_DIPLOMA_TOPIC]->(t)
  FOREACH (__ IN CASE WHEN sup IS NULL THE
N [] ELSE [1] END |
    MERGE (s)-[:SUPERVISED_BY]->(sup)
    MERGE (sup)-[:SUPERVISES]->(t)
  )
)

WITH 1 AS _

// 8) Scientific-activity extension: publications, c
oauthors, keywords, citations
UNWIND [
  {pubId: "doi:10.1109/CSIT65290.2024.109825
87", kind: "shared", typeCode: "conference", ye
ar: 2024, title: "O. Penko, V. Pienko and E. Mala
khov. Multiagent Path Finding using Dijkstra Alg
orithm (CSIT 2024)", venue: "IEEE CSIT 2024, Lvi
v, Ukraine", doi: "10.1109/CSIT65290.2024.109
82587"},
  {pubId: "doi:10.15276/imms.v11.no4.287", kin
d: "shared", typeCode: "article", year: 2021, titl
e: "Кунак І. С., Шпінарєва І. М., Пенко В. Г. Іде
нтифікація особи у відеопотоці методами ма
шинного навчання", venue: "Інформатика та
математичні методи в моделюванні. Том 11,
№4. 2021. С. 287–295", doi: "10.15276/imms.v
11.no4.287"},
  {pubId: "doi:10.15276/aait.09.2026.09", kind:
"shared", typeCode: "article", year: 2026, title:
"Gunchenko Y. O.; Meshcheryakov V. I.; Prykho
dko S. B.; Rachinska A. L. Estimating the Numbe
r of Lines of Code ...", venue: "Applied Aspects
of Information Technology 9(1), 122–133", doi:
"10.15276/aait.09.2026.09"},
  {pubId: "doi:10.1109/CSIT65290.2024.109825
87", kind: "mongo", typeCode: "conference", ye
ar: 2024, title: "Oleksandr Penko, Valerii Pienko
, Eugene Malakhov, Multiagent Path Finding usi
ng Dijkstra Algorithm", venue: "Proceedings of I
EEE 19th International Conference on Compute
r Science and Information Technologies (CSIT),
16-19 October 2024, Lviv, Ukraine. – прийнято
до публікації.  Scopus", doi: "10.1109/CSIT652
90.2024.10982587"},
  {pubId: "doi:10.1109/CSIT65290.2024.109826
30", kind: "mongo", typeCode: "conference", ye
ar: 2024, title: "Irina Kalinina, Aleksandr Gozhyj,
Victoria Vysotska, Eugene Malakhov, Victor Goz
hyj, Iryna Tregubova, System Methodology of D
ata Analysis and Pre-processing for Solving Clas
sification Problems", venue: "Proceedings of IEE
E 19th International Conference on Computer S
cience and Information Technologies (CSIT), 16-
19 October 2024, Lviv, Ukraine. – прийнято до
публікації.  Scopus", doi: "10.1109/CSIT65290.
2024.10982630"},
  {pubId: "doi:10.15587/1729-4061.2024.31397
0", kind: "mongo", typeCode: "conference", yea
r: 2024, title: "Bocharova, M., Malakhov, E. (20
24). Improving information theory of context-a
ware phrase embeddings in HR domain. Eastern
-European Journal of Enterprise Technologies, 5
(2 (131)), 53–60. https:", venue: "doi.org/10.15
587/1729-4061.2024.313970  Scopus (Q3)", doi
: "10.15587/1729-4061.2024.313970"},
  {pubId: "doi:10.15276/imms.v11.no1-2.58", ki
nd: "mongo", typeCode: "article", year: 2021, ti
tle: "Пенко В.Г., Шпінарева І.М., Ярощук О.В.
Діагностика хвороби серця на основі дерева
рішень", venue: "\" Інформатика та математи
чні методи в моделюванні \". Науковий фахо
вий журнал. Том 11, №2.  - Одеса, ОНПУ. 202
1, с.112-119", doi: "10.15276/imms.v11.no1-2.
58"},
  {pubId: "Penko:4:2", kind: "mongo", typeCode:
"article", year: 2018, title: "V. Penko, Gafar Abd
ula I. Approach to identifying plagiarism in mult
ilingual texts", venue: "\"Информатика и мате
матические методы в моделировании\". Нау
чный журнал. Том 8, №2.  - Одеса, ОНПУ. 201
```

```
8, с.121-128", doi: null},
 {pubId: "Penko:4:3", kind: "mongo", typeCode: "thesis", year: 2021, title: "Пенко В.Г., Пенко О.В. Підхід до реалізації системи прийняття рішень для оптимізації управління на регіональному рівні", venue: "Матеріали щорічної науково –практичної конференції «Удосконалення економічних механізмів розвитку територій», Одеський регіональний інститут державного управління Національної академії державного управління при Президентові України - 28 травня 2021 р", doi: null},
 {pubId: "doi:10.15276/aait.08.2025.26", kind: "personal", typeCode: "article", year: 2025, title: "Chaikovskyi M. A.; Malakhov E. V. Vision-Based GUI Testing: A Systematic Review ...", venue: "Applied Aspects of Information Technology 8(4), 397–423", doi: "10.15276/aait.08.2025.26"},
 {pubId: "doi:10.31650/2618-0650-2025-7-2-160-169", kind: "personal", typeCode: "article", year: 2025, title: "Рачинська А.Л., Недєва О.А., Іщенко О.В. Інформаційна технологія візуалізації динаміки розвитку пожежі", venue: "(зі списку публікацій, 2025)", doi: "10.31650/2618-0650-2025-7-2-160-169"},
 {pubId: "doi:10.34229/2707-451X.25.2.7", kind: "personal", typeCode: "article", year: 2025, title: "Dvorchuk D., Shpinareva I. Analysis of Blockchain-Technology", venue: "Cybernetics and Computer Technologies. 2025. 2. P. 77–87", doi: "10.34229/2707-451X.25.2.7"},
 {pubId: "doi:10.15276/imms.v11.no1-2.58", kind: "personal", typeCode: "article", year: 2021, title: "Пенко В.Г., Шпінарєва І.М., Ярощук О.В. Діагностика хвороби серця на основі дерева рішень", venue: "Інформатика та математичні методи в моделюванні. Том 11, №2. 2021", doi: "10.15276/imms.v11.no1-2.58"},
 {pubId: "doi:10.17721/2519-481X/2022/76-08", kind: "personal", typeCode: "article", year: 2022, title: "Yurii Gunchenko et al. Analysis of the current state of the elements of ternary logic", venue: "Збірник наукових праць ВІКНУ. 2022. №76. С. 88–101", doi: "10.17721/2519-481X/2022/76-08"}
] AS p
MERGE (pub:Publication {pubId: p.pubId})
SET pub.kind = p.kind,
   pub.typeCode = p.typeCode,
   pub.year = p.year,
   pub.title = p.title,
   pub.venue = p.venue,
   pub.doi = p.doi;

MATCH (pub:Publication)
WHERE pub.typeCode IS NOT NULL
MATCH (pt:PublicationType {code: pub.typeCode})
MERGE (pub)-[:OF_TYPE]->(pt);

// Authorship
UNWIND [
 {lectKod: 1, pubId: "doi:10.1109/CSIT65290.2024.10982587"},
 {lectKod: 1, pubId: "doi:10.1109/CSIT65290.2024.10982630"},
 {lectKod: 1, pubId: "doi:10.15587/1729-4061.2024.313970"},
 {lectKod: 4, pubId: "doi:10.15276/imms.v11.no1-2.58"},
 {lectKod: 4, pubId: "Penko:4:2"},
 {lectKod: 4, pubId: "Penko:4:3"},
 {lectKod: 1, pubId: "doi:10.1109/CSIT65290.2024.10982587"},
 {lectKod: 4, pubId: "doi:10.1109/CSIT65290.2024.10982587"},
 {lectKod: 4, pubId: "doi:10.15276/imms.v11.no4.287"},
 {lectKod: 3, pubId: "doi:10.15276/imms.v11.no4.287"},
 {lectKod: 2, pubId: "doi:10.15276/aait.09.2026.09"},
 {lectKod: 5, pubId: "doi:10.15276/aait.09.2026.09"},
 {lectKod: 1, pubId: "doi:10.15276/aait.08.2025.26"},
 {lectKod: 2, pubId: "doi:10.31650/2618-0650-2025-7-2-160-169"},
 {lectKod: 3, pubId: "doi:10.34229/2707-451X.25.2.7"},
 {lectKod: 4, pubId: "doi:10.15276/imms.v11.no1-2.58"},
 {lectKod: 5, pubId: "doi:10.17721/2519-481X/2022/76-08"}
] AS a
MATCH (l:Lecturer {kod: a.lectKod})
MATCH (pub:Publication {pubId: a.pubId})
MERGE (l)-[:AUTHORED]->(pub);
```

```
// Coauthorship derived from ANY joint
// publication (recommended)
// Why: in real data 'kind' may be edited; coaut
horship should follow actual joint authorship.
MATCH ()-[r:COAUTHOR_WITH]-() DELETE r;

MATCH (l1:Lecturer)-[:AUTHORED]->(p:Publicat
ion)<-[:AUTHORED]-(l2:Lecturer)
WHERE l1.kod < l2.kod
MERGE (l1)-[r:COAUTHOR_WITH]->(l2)
ON CREATE SET r.sharedPublications = 1
ON MATCH  SET r.sharedPublications = coalesce
(r.sharedPublications,0) + 1;

// Keywords (minimal)
UNWIND [
  {pubId: "doi:10.1109/CSIT65290.2024.109825
87", keywords: ["multiagent", "path finding", "d
ijkstra"]},
  {pubId: "doi:10.15276/imms.v11.no4.287", ke
ywords: ["video stream", "machine learning", "i
dentification"]},
  {pubId: "doi:10.15276/aait.09.2026.09", keyw
ords: ["open source", "LOC", "regression"]}
] AS krow
MATCH (pub:Publication {pubId: krow.pubId})
UNWIND krow.keywords AS kw
MERGE (k:Keyword {value: kw})
MERGE (pub)-[:HAS_KEYWORD]->(k);

// Citations (example)
UNWIND [
  {fromId: "shpinareva:blockchain-analysis-2025
", toId: "penko-shpinareva:video-id-2021"},
  {fromId: "doi:10.17721/2519-481X/2022/76-0
8", toId: "doi:10.15276/aait.09.2026.09"}
] AS c
MATCH (pFrom:Publication {pubId: c.fromId})
MATCH (pTo:Publication {pubId: c.toId})
MERGE (pFrom)-[:CITES]->(pTo);

// 9) Activities
// NOTE: Neo4j properties cannot store maps;
// activity 'details' is stored as string
// via toString().
UNWIND [
  {lectKod: 1, typeKind: null, typeCode: null, typ
eLabel: null, year: 2024, name: "Сенсорна елек
троніка та мікросистемні технології", role: "Чл
ен редакційної колегії", details: {url: "http://se
mst.onu.edu.ua"}},
  {lectKod: 1, typeKind: null, typeCode: null, typ
eLabel: null, year: 2024, name: "International C
onference on Software and Computer Applicati
ons (ICSCA)", role: "program/session chair", det
ails: {period: "2019-2024", country: "Malaysia"}
},
  {lectKod: 1, typeKind: null, typeCode: null, typ
eLabel: null, year: 2024, name: null, role: null, d
etails: {}},
  {lectKod: 4, typeKind: null, typeCode: null, typ
eLabel: null, year: 2018, name: "CAIT-Odessa-2
018", role: "Доповідач/співавтор", details: {cit
y: "Одеса", country: "Україна"}},
  {lectKod: 4, typeKind: null, typeCode: null, typ
eLabel: null, year: 2021, name: null, role: null, d
etails: {city: "Одеса"}}
] AS ar
MATCH (l:Lecturer {kod: ar.lectKod})
MERGE (act:Activity {id: toString(l.kod)+':' + toS
tring(ar.year) + ':' + coalesce(ar.typeCode,'custo
m') + ':' + coalesce(ar.name,'NA')})
SET act.typeKind = ar.typeKind,
   act.typeCode = ar.typeCode,
   act.typeLabel = ar.typeLabel,
   act.year = ar.year,
   act.name = ar.name,
   act.role = ar.role,
   // details fields (Mongo 'details' is a map; Ne
o4j properties must be primitives)
   act.details_url     = ar.details.url,
   act.details_city    = ar.details.city,
   act.details_country = ar.details.country,
   act.details_period  = ar.details.period
MERGE (l)-[:HAS_ACTIVITY]->(act);

MATCH (act:Activity)
WHERE act.typeCode IS NOT NULL
MATCH (at:ActivityType {code: act.typeCode})
MERGE (act)-[:OF_TYPE]->(at);
```

ПРИКЛАДИ: запити для використання графової моделі

```
// Q1) Подивитись усі спроби студента складання однієї дисципліни
// (можливі декілька оцінок)
// :param studKod => 1, :param dKod => 1
/*
MATCH (s:Student {kod:$studKod})-[:HAS_GRADE]->(ga:GradeAttempt)-[:FOR_DISCIPLINE]->(d:Dis
cipline {dKod:$dKod})
RETURN s.secondName, d.name, ga.mark, ga.mdate
ORDER BY ga.mdate DESC;
*/

// Q2) Роботою кого ще зі студентів керує цей викладач? (студенти + теми дипломних робіт)
// :param lectKod => 1
/*
MATCH (sup:Lecturer {kod:$lectKod})-[:SUPERVISES]->(t:DiplomaTopic)<-[:HAS_DIPLOMA_TOPIC]-
(s:Student)
RETURN sup.secondName AS supervisor, t.dpKod, t.dpName,
    collect(s.secondName + ' ' + s.firstName) AS students
ORDER BY t.dpKod;
*/

// Q3) Отримати список студентів з одного міста
// :param city => "м. Одеса"
/*
MATCH (a:Address {city:$city})<-[:LIVES_AT]-(s:Student)-[:IN_GROUP]->(g:Group)
RETURN a.city, g.gNum, collect(s.secondName + ' ' + s.firstName) AS students
ORDER BY g.gNum;
*/

// Q4) З ким з колег ще опублікувався співавтор цього викладача?
// :param lectKod => 1
/*
MATCH (l:Lecturer {kod:$lectKod})-[:COAUTHOR_WITH]-(c1:Lecturer)
MATCH (c1)-[:COAUTHOR_WITH]-(c2:Lecturer)
WHERE c2.kod <> l.kod
RETURN c1.secondName + ' ' + c1.firstName AS coauthor,
    collect(DISTINCT c2.secondName + ' ' + c2.firstName) AS other_coauthors
ORDER BY coauthor;
*/
```

Навчальне видання

Малахов Євгеній Валерійович

NoSQL моделі та бази даних
реалізація засобами MongoDB та Neo4j

НАВЧАЛЬНИЙ ПОСІБНИК

В авторській редакції

Підп. до друку 16.06.2026. Формат 60x84/16.
Ум.-друк. арк. 12,2. Наклад 50 прим. Зам. № 2708/26.

ТОВ «Маджента»
пл. Бориса Дерев'янко, 1, м. Одеса, 65072
Свідоцтво суб'єкта видавничої справи ДК № 7133 від 23.09.2020

Тел.: (094) 930- 67-28
e-mail: magenta@mbx.in.ua

Надруковано з готового оригінал-макета