МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

О. О. Сергеєв-Горчинський, Л. С. Глоба, І. О. Мачалін

СИСТЕМИ БАЗ ДАНИХ
КОМП'ЮТЕРНИЙ ПРАКТИКУМ

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського як навчальний посібник для студентів, які навчаються за спеціальністю 122 «Комп'ютерні науки» і спеціалізацією «Інтелектуальні сервіс-орієнтовані розподілені обчислювання», спеціальністю 172 «Телекомунікації та радіотехніка» і спеціалізацією «Інформаційно-комунікаційні технології»

Київ
КПІ ім. Ігоря Сікорського
2021

Рецензенти *Мухін Вадим Євгенович,*
доктор технічних наук, професор
Скулиш Марія Анатоліївна,
доктор технічних наук, старший наук. співр.
Відповідальний редактор *Руренко Олександр Григорович,*
кандидат фізико-математичних наук, старший вик.

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 4 від 10.12.2020 р.)
за поданням Вченої ради ІТС
(протокол № 8 від 28.09.2020 р.)

Сергеєв-Горчинський Олексій Олександрович,
кандидат технічних наук
Глоба Лариса Сергіївна,
доктор технічних наук
Мачалін Ігор Олексійович,
доктор технічних наук

СИСТЕМИ БАЗ ДАНИХ
КОМП'ЮТЕРНИЙ ПРАКТИКУМ

Системи баз даних: Комп'ютерний практикум [Електронний ресурс] : навч. посіб. для студ. спеціальності 122 «Комп'ютерні науки» і спеціалізації «Інтелектуальні сервіс-орієнтовані розподілені обчислювання», спеціальності 172 «Телекомунікації та радіотехніка» і спеціалізації «Інформаційно-комунікаційні технології» / О. О. Сергеєв-Горчинський, Л. С. Глоба, І. О. Мачалін; КПІ ім. Ігоря Сікорського. – Електронні текстові дані (1 файл: 3 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2021. – 150 с.: Іл.

Зміст

Вступ

Еволюція моделей баз даних

За 20-ть років після широкого впровадження електронних обчислювальних машин (ЕОМ) сформувався ландшафт систем баз даних (СБД). Еволюція моделей баз даних (БД) охоплює три періоди: дореляційний (1950-72) – поява ранніх ЕОМ й стандарту CODASYL; реляційний (1972-2005) – створення реляційних СБД й стандарту SQL; постреляційний (2005-сьогодення) – експоненційне розростання кількості нереляційних моделей БД, обумовлене необхідністю розподіленого зберігання й підтримання безперервного доступу до даних [1].

Ранні системи баз даних

Хоча поняття «база даних» з'явилось тільки наприкінці 60-х років, збір і систематизація даних були невід'ємним фактором розвитку людської цивілізації й технологій. Текстові джерела, особливо структуровані – словники й енциклопедії – приклади масивів даних. Бібліотеки й інші індексовані архіви – еквіваленти сьогоднішніх СБД.

Еволюція цифрових баз даних відобразилась в появі перфокарт і інших фізичних носіїв, які мали б змогу зберігати інформацію в формі, придатній для механічної обробки. На початку 20-го століття ранні реалізації БД для зберігання даних застосовували магнітні стрічки.

В середині 50-х років з появою магнітних дисків програмно було реалізовано функції оновлення, дозаписування і видалення окремих екземплярів даних БД. В період появи магнітних дисків було розроблено метод послідовного доступу на базі індексів (Indexed Sequential Access Method, ISAM), котрий уможливив швидке зчитування екземплярів даних БД і сприяв створенню ранніх комп'ютерних систем обробки онлайн-транзакцій

(On-line Transaction Processing, OLTP).

Дореляційний період еволюції моделей баз даних

Функціонування будь-якого ПЗ містить елементи обробки даних, що в ранніх реалізаціях БД призводило до недостатньої продуктивності інформаційних систем. Будь-яке створюване програмне забезпечення мало повторно реалізовувати взаємодію ПЗ з базою даних. Помилки в коді ПЗ призводили до пошкодження або видалення даних.

Бажано було виокремити логіку взаємодії з БД з розробленого ПЗ в окрему систему керування вмістом БД, котре спрощувало б завдання програмування і забезпечило узгодженість й продуктивність виконання одночасних запитів до БД.

Ранні системи іменувались «навігаційними», оскільки потребували переміщення між об'єктами програм за вказівниками або посиланнями. Наприклад, для пошуку замовлення в ієрархічній базі даних необхідно було спочатку знайти об'єкт «Користувач», а потім переходити за посиланнями до наявних замовлень.

Бази даних першого покоління функціонували винятково на тогочасних ЕОМ – «мейнфреймах», зокрема виробництва компанії IBM. До початку 70-х років дві основні моделі СБД конкурували за першість. Особливості розробки мережевої моделі даних було конкретизовано в стандарті організації «Конференція щодо мов систем обробки даних» (COnference on DAta SYstems Language, CODASYL) і реалізовано компанією Hewlett Packard в СБД «Integrated Database Management System» («IDMS»). Ієрархічну модель було реалізовано компанією IBM в СБД «Information Management System» («IMS»).

Ієрархічні й мережеві системи баз даних підтримували функціонування переважної більшості ПЗ до кінця 70-х років, але містили певні вади. По-перше, навігаційні бази даних були надзвичайно негнучкими з точки зору

структур даних і запитів. Як правило, були можливі тільки ті запити, котрі можна було очікувати на початковому етапі проектування, і надзвичайно важко було дозаписувати нові екземпляри даних в існуючі БД.

По-друге, необхідність в комп'ютерному програмуванні статистичних звітів. В результаті існувала значна кількість програмістів, котрі мали формувати частково-ідентичні шаблони звітів на мовах програмування.

Реляційний період еволюції моделей баз даних

З 49-го року в IBM над розробкою ЕОМ працював Edgar Codd, котрий мав зауваження щодо тогочасних БД:

1. Структура баз даних були занадто складною для застосування спеціалістами без навичок комп'ютерного програмування.
2. Упорядкування даних немало теоретичного підґрунтя. Edgar Codd мислив з погляду формальних структур і логічних операцій щодо даних. Він вважав, що нежорстка структура БД була причиною незабезпечення логічної узгодженості значень окремих екземплярів даних БД.

В 70-х роках Edgar Codd опублікував серію наукових праць [2], в котрих описав умови формалізації реляційної моделі баз даних.

Реляційна теорія

Реляційна модель описує, як масив даних має бути відображений для користувача. Реляційна теорія включає поняття:

– кортеж – список значень атрибутів конкретного екземпляра даних. В реальній системі бази даних кортеж відповідає рядку, а атрибут – значенню стовпця;

– сутність (відношення) – послідовність кортежів, «таблиця» в реляційній реалізації;

– умова щодо значень атрибуту (стовпця) сутності – умова унікальності необхідна для ідентифікації кортежів і взаємозв'язків між сутностями.

Запити щодо множин екземплярів даних містять обчислення об'єднання, перетину, проекції тощо. Перераховані запити повертають тимчасові відношення. На практиці це означає, що запит щодо вмісту таблиці повертає дані в табличному форматі.

Рядок (кортеж) таблиці має бути ідентифікований за значенням первинного ключа (primary key). Списки, множини й інші структури даних, котрі містять вкладену інформацію, не підтримуються як стандартні типи значень в реляційних СБД.

Транзакції

Реляційна модель самостійно не визначає, яким чином база даних обробляє паралельні запити щодо оновлення даних. Корегування, котрі іменуються транзакціями, являють складність підтримання вмісту БД в узгодженому стані для більшості СБД.

Наприкінці 70-х років Jim Gray розробив модель транзакцій. Як він висловився, «Транзакція – оновлення поточного стану БД, характеризується «атомарністю» (atomicity) (виконання всіх складових запиту або невиконання всіх), «довговічністю» (durability) (відновлення стану БД при аварійному відключенні сервера) і «узгодженістю» (consistency) (перевіркою коректності значень всіх складових запиту)». Перераховані властивості було узагальнено в умови транзакційності ACID:

1. Атомарність (Atomic): Транзакція неподільна – або виконуються всі складові запиту, або всі не виконуються.

2. Узгодженість (Consistent): база даних залишається в узгодженому стані до й після виконання транзакції.

3. Ізольованість (Isolated): В той час як транзакції можуть виконуватись клієнтськими програмами одночасно, окрема транзакція не має бачити результат виконання інших транзакцій.

4. Довговічність (Durable): Після виконання транзакції (SQL-запит COMMIT) навіть при аварійному відключенні операційної системи (ОС) або апаратного забезпечення, стан БД має бути відновлено.

Ранні реляційні бази даних і мова SQL

В 74-у році IBM ініціювала дослідницьку програму розробки реляційної СБД «System R» з підтримкою «мови структурованих запитів» (Structured Query Language, SQL). Mike Stonebraker в 70-х роках в університеті Berkeley керував розробкою «діалогової системи графічного пошуку» (Interactive Graphics Retrieval System, скорочено «Ingres») – реляційної СБД з підтримкою мови запитів QUEL [3]. Larry Ellison в корпорації Amdahl, спеціалізованій на створенні обладнання мейнфреймів IBM, заснував компанію, котра створила комерційну реляційну СБД – «Oracle Database».

Боротьба розробників реляційних баз даних

Поява мінікомп'ютерів завершила період першості мейнфреймів. Порівнюючи з сучасними комп'ютерами, мінікомп'ютери кінця 70-х і початку 80-х років навряд були «міні». Але на відміну від мейнфреймів, їм взагалі не було потрібно спеціалізованого обладнання, і вони вперше забезпечили компанії середнього розміру приватною обчислювальною інфраструктурою.

В 81-у році IBM випустила комерційну реляційну СБД «SQL/DS», котра виконувалась тільки на ОС мейнфреймів виробництва IBM. СБД Oracle компанії Larry Ellison-а була розроблена в 79-у році й підтримувала різноманітні архітектури мінікомп'ютерів, зокрема компаній Digital і Data General.

В 86-у році в університеті Berkeley було реалізовано поліпшену версію СБД Ingres – «post-Ingres», скорочено Postgres. Oracle і Postgres боролись за першість серед реляційних СБД для мінікомп'ютерів.

Реляційні бази даних були настільки поширені, що розробники попередніх типів СБД почали доповнювати документацію описом реляційних властивостей. Дана обставина спонукала Edgar-a Codd-a сформулювати 12 правил (fidelity rules) (насправді 13 правил, починаючи з загального правила № 0) побудови реляційних БД [4].

Протягом наступних десятиліть було впроваджено різноманітні RDBMS, серед котрих «Sybase», «Microsoft SQL Server», «Informix», «MySQL», «DB2». В той час як більшість з перерахованих систем декларували продуктивність й економічність, вони були практично ідентичними за трьома складовими: реляційна модель БД, умови транзакційності ACID, мова запитів SQL.

Об'єднання вузлів комп'ютерної мережі

До кінця 80-х років реляційна модель явно здобула рішучу перемогу. Мінікомп'ютери були в певному сенсі «маленькими мейнфреймами»: обробка запитів відбувалась на мінікомп'ютері, користувач взаємодіяв з ПЗ засобами консолі клієнтської програми. Зростаюча поширеність мінікомп'ютерного апаратного забезпечення на базі стандарту IBM PC, і поява графічних користувацьких інтерфейсів, як то Microsoft Windows, вкупі призвели до появи моделі ПЗ «client-server».

Клієнтські програми виконувані на комп'ютері (Personal Computer, PC) обмінювались даними з сервером СБД, розміщеним на мінікомп'ютері. Функції прикладного програмного забезпечення, котрі формували запити до СБД, стандартно були розміщені на стороні клієнтських програм, але, зокрема, могли розміщуватись і на сервері СБД в вигляді збережених

процедур (Stored Procedures) на базі доповненого синтаксису мови SQL [5].

Об'єктно-орієнтоване програмування й OODBMS

В «процедурних» мовах програмування функції обробки й масиви даних розміщуються окремо. Процедури завантажують дані й виконують обробку в рамках логіки ПЗ. Об'єктно-орієнтоване програмування (ООП) поєднало опис даних й функції ПЗ в єдиний об'єкт. Найчастіше застосовувані принципи об'єктно-орієнтованого програмування:

1. Інкапсуляція – клас об'єктів визначає як дані, так і функції, котрі можна виконувати щодо даних. Клас об'єкту обмежує прямий доступ до даних, потребуючи, щоб редагування даних було можливе тільки застосовуючи функції об'єкту. Наприклад, клас «Службовець» може містити в собі функції «Отримання окладу» й «Оновлення окладу». Функція «Оновлення окладу» може містити умову щодо мінімального й максимального значень окладу, клас «Службовець» має не допускати оновлення окладу за межами відповідної функції.

2. Наслідування – клас об'єктів може наслідувати характеристики інших класів. Клас «Службовець» наслідує всі властивості наявного класу «Персона» (ПІПб, податковий ідентифікатор тощо) і специфічні властивості: зарплата, тип діяльності тощо.

Об'єктно-орієнтоване програмування значно підвищило продуктивність розробки й якість ПЗ. На початку 90-х більшість мов програмування були об'єктно-орієнтованими.

В середині 90-х років розробники програмного забезпечення (ПЗ) були збентежені невідповідністю між об'єктно-орієнтованим відображенням даних в створюваному ПЗ і реляційним відображенням даних в СБД. Наприклад, в ПЗ «Інтернет-магазин», об'єкт «Користувач» містив дані щодо окремого користувача з посиланням на об'єкт «Замовлення», котрий в свою чергу

містив посилання щодо окремих найменувань асортименту, за прикладом мережевої моделі CODASYL.

При дозаписуванні або зчитуванні об'єкта в реляційній СБД, для перетворення об'єктної сутності в реляційну необхідно було на мові SQL формувати послідовність запитів об'єднання, перетину й проекції, в результаті чого запити виконувались з певною часовою затримкою.

Було запропоновано розміщувати в БД об'єкти програм, а СБД, котрі іменувались OODBMS реалізовували завантаження об'єктів, без необхідності нормалізації і поділу на окремі елементні блоки. Структура OODBMS подібна до навігаційної моделі з дореляційного періоду – вказівники в складі об'єкта забезпечують переміщення за взаємозв'язаними об'єктами.

В середині 90-х років більшість експертів прогнозували, що OODBMS системи замістять реляційні СБД (RDBMS). Діючі розробники реляційних БД, в першу чергу Oracle, IBM, Informix і Sybase, швидко впровадили функції OODBMS.

Прихильники моделі OODBMS були зосереджені тільки на перевагах, пропонованих розробникам ПЗ, і ігнорували вади нової моделі щодо вирішення завдань обробки статистичних даних. OODBMS і реалізована модель даних набули поширення тільки серед спеціалістів з навичками програмування, так як не забезпечували інтерфейс для завдань аналізу й адміністрування даних. В подальшому для взаємодії з реляційними СБД були розроблені бібліотеки програм «об'єктно-реляційного відображення» (Object-Relational Mapping, ORM).

Формат RDF і мова SPARQL

Уніфікований опис ресурсів (Resource Description Framework, RDF) – Веб-стандарт, розроблений наприкінці 90-х років для моделювання Веб-ресурсів і взаємозв'язків між ними. Це ранній стандарт відображення й

обробки графів даних.

Інформація в RDF виражається в формі тверджень «суб'єкт–предикат–об'єкт». Масив RDF-тверджень формує орієнтований граф.

RDF-граф підтримує відображення на базі різноманітних форматів, включаючи XML, або навіть в реляційній моделі.

Для взаємодії з RDF розроблена подібна до SQL мова запитів «SPARQL Protocol and RDF Query Language» (SPARQL). RDF-документи зберігаються в базах даних, іменованих «triplestores», спеціалізоване ПЗ для редагування документів: «AllegroGraph», «Ontotext GraphDB», «StarDog», «Oracle Spatial».

Постреляційний період еволюції моделей баз даних

Після того як надмірна на той час увага до об'єктно-орієнтованих баз даних зменшилась, запит щодо застосування реляційних СБД залишався до другої половини 2000-х років. Протягом 1995-2005-го років не було розроблено нових типів баз даних: ринок СБД був насичений достатньою кількістю RDBMS систем, і те, що модель RDBMS трималась на ринку, означало відсутність нереляційних альтернатив.

Документні бази даних

Системи об'єктно-реляційного відображення тільки частково вирішили необхідність зберігання складноструктурованих об'єктів. З 2004 року збільшувалась кількість Веб-сайтів, котрі забезпечували інтерактивну взаємодію завдяки техніці програмування Asynchronous JavaScript і XML (AJAX), в котрій JavaScript всередині браузера взаємодіяв безпосередньо з backend-системою при передачі XML-повідомлень. Далі розробники почали застосовувати подібний до XML, але значно компактніший формат JavaScript Object Notation (JSON).

JSON перетворився в формат серіалізації (зберігання) програмних

об'єктів. Деякі Веб-сайти почали зберігати JSON-документи в стовпцях реляційних таблиць. Пізніше з'явились бази даних, в котрих можна було безпосередньо зберігати JSON-документи, наприклад, розроблена в 2005-у році СБД «CouchDB».

Соціальні мережі

Розробники СБД розподіляються в думках практично по кожному аспекту проектування баз даних, але вони згодні в тому, що БД виконує зберігання інформації щодо «об'єктів», будь то JSON-документи, таблиці або списки. Коли є потреба в формуванні гнучких взаємозв'язків між сутностями, тоді є сенс застосовувати графові СБД.

Графове відображення природне для соціальних мереж, як то Facebook. В Facebook інформація щодо користувачів була важливою, але сформований «соціальний граф» і взаємозв'язки між користувачами забезпечували тогочасну унікальність платформи. Існують певні вади відображення графа в реляційну модель:

1. Синтаксис SQL нерозрахований щодо запитів пошуку взаємозв'язків, особливо за необмеженої їх кількості.
2. Пошук ускладнюється при розростанні графа, в результаті чого запити виконуються з часовою затримкою.

Поняття «граф»

Порівнюючи зі значною кількістю нереляційних систем, реляційні і графові бази даних мають теоретичне підкріплення. Теорія графів – сформований розділ математики з різноманіттям практичних застосувань в медицині, фізиці, соціології, сервіс-орієнтованих обчисленнях, і, звісно, інформаційно-комунікаційних технологіях.

Теорія графів визначає наступні основні компоненти графа:

– вершини або «вузли» відображають різноманітні об'єкти;
– об'єкти з'єднуються ребрами;
– вершини і ребра можуть містити властивості.

В той час як «вершина» й «ребро» є поняттями теорії графів, в контексті графових СБД застосовують відповідно поняття «вузол» і «взаємозв'язок». Властивості вузлів є подібними до стовпців реляційних таблиць чи полів JSON-документів.

Системи графових баз даних

В той час як RDF-документи були складовою побудови графів, модель графа властивостей (Property Graph) забезпечувала гнучку структуру для відображення складноструктурованих даних, сформованих при об'єднанні вузлів і зв'язків за значеннями властивостей.

На базі графової моделі властивостей в 2007-у році була розроблена СБД «Neo4j». Neo4j реалізує сумісні з вимогами ACID транзакції, підтримує зберігання млрд. вузлів/взаємозв'язків й декларативну мову запитів «Cypher».

Cypher забезпечує простоту зчитування графів завдяки інтуїтивно-зрозумілому синтаксису, порівнянному з SQL або SPARQL, але оптимізованому для пошуку взамозв'язків в графових СБД. В 2015-у році компанія Neo4j анонсувала ініціативу щодо стандартизації мови Cypher – «openCypher», котру підтримали Oracle, Apache Spark і інші розробники ПЗ.

Системи стовпцевих баз даних

В 2000-х роках компанія Google усвідомила, що тогочасні реляційні СБД не мали змоги впоратись з обсягами й швидкістю дозаписування даних. Google започаткувала розробку апаратного й програмного забезпечення для зберігання й обробки даних щодо зростаючої кількості Веб-сайтів.

В 2003 році Google було опубліковано подробиці розподіленої файлової

системи GFS [6], в 2004 році – алгоритм розподіленої паралельної обробки «map-reduce» [7], який застосовувався для індексування Веб-сайтів, в 2006 році – модель даних розподіленої комерційної СБД «BigTable» [8].

Значна кількість концепцій, розроблених компанією Google, були реалізовані в рамках проекту Hadoop за участю компанії Yahoo [9]. В 2008-у році в рамках проекту Hadoop було розроблено стовпцеву СБД з відкритим вихідним кодом – «HBase», подібну за архітектурою до СБД BigTable [10].

Системи баз даних «ключ/значення»

Ранні сайти соціальних мереж, як то MySpace і пізніше Facebook, усвідомили необхідність масштабування обчислювальної інфраструктури з тис. до млн. користувачів. Тогочасні комерційні RDBMS не мали змоги забезпечити достатню масштабованість і тому подібні Веб-сайти застосовували різноманітні типи оптимізації ресурсів апаратного забезпечення; зберігання даних в оперативній пам'яті в вигляді кортежів «ключ-значення» для зменшення кількості запитів до зовнішньої пам'яті; реплікування для пришвидшення зчитування БД; шардування для розподілення даних між базами даних відповідно до інтервалів значень первинних ключів таблиць реляційних СБД.

Соціальні мережі Twitter і Facebook розподіляли дані користувачів між значною кількістю екземплярів СБД «MySQL», що забезпечувало швидке виконання запитів користувачів з відповідних клієнтських програм. Експлуатаційні витрати щодо розподілення даних між зовнішніми пристроями призвели до пошуку альтернатив RDBMS.

В 2009-у році створено СБД «ключ/значення» з відкритим вихідним кодом – «REmote DIctionary Server» (REDIS) [11]. В 2012-у році компанією Amazon створено комерційну СБД «ключ/значення» – «DynamoDB» [12].

Розділ 1. Системи реляційних баз даних

Системи реляційних баз даних (реляційні СБД) розроблені на базі теорії множин. Реляційна модель даних містить двовимірні структури «таблиці», котрі складаються з рядків і стовпців [13-19]. Стандартний спосіб взаємодії з реляційними СБД – декларативна мова структурованих запитів (Structured Query Language, SQL), котра забезпечує створення таблиць, дозаписування/зчитування екземплярів даних БД за умовами запиту. Значення даних є типізованими, містять числові, строкові, часові значення, неструктуровані бінарні об'єкти (Binary Large Object, BLOB) [17-19].

1.1. Випадки, коли рекомендовано застосовувати

Реляційні СБД має сенс застосовувати за необхідності в узгодженому зберіганні даних і формуванні довільних запитів за вмістом БД. Прикладами предметних областей, в котрих застосовують реляційні СБД є облік складських приміщень, Інтернет-магазини, комунальні сервіси тощо [19].

1.2. Випадки, коли не рекомендовано застосовувати

Якщо структура даних є тимчасовою, часто коригується або характеризується глибокою вкладеністю, тоді реляційні СБД – некращий варіант, так як для них необхідно заздалегідь зазначити схему бази даних (сутності, взаємозв'язки, наприклад, в завданні з опису всіх живих сутностей в природі), що є складним завданням, якщо значна кількість екземплярів даних відрізняються за структурою (стовпці, допустимі значення за стовпцями) (див. розд. «Документні СБД») [14; 19; 20].

1.3. СБД PostgreSQL

Існує чимало реляційних СБД з відкритим вихідним кодом – PostgreSQL, MySQL, H2, HSQLDB, SQLite тощо.

PostgreSQL – приклад системи, котра слідує традиціям реляційних СБД, підтримується значною кількістю фахівців, є стабільною і при належному налаштуванні вирішує різноманітні завдання. В PostgreSQL існують додаткові модулі, наприклад, для відображення запитів природною мовою, побудови індексів, опису типів даних, створення агрегатних функцій.

В СБД реалізовано черги завдань, наслідування таблиць, вкладені запити, властивості ACID – атомарність (англ. atomicity), узгодженість (англ. consistency), ізольованість (англ. isolation), довговічність (англ. durability) [13; 14; 17]. PostgreSQL притаманні швидкість, надійність, здатність до зберігання > 2 Тбайт даних.

СБД підтримує виконання сервера на різноманітних ОС, обробка SQL-запитів відповідає стандарту ANSI. Ефективність застосування PostgreSQL було підтверджено в високонавантажених інформаційних системах: Skype, Федеральне управління цивільної авіації США тощо [19].

1.3.1. Передісторія

PostgreSQL розроблена в університеті Берклі в 70-х роках і іменувалась як «діалогова система графічного пошуку» (Interactive Graphics Retrieval System, скорочено «Ingres»). В 80-х роках було випущено поліпшену версію post-ingres, скорочено Postgres.

В 1993-у році університет Берклі припинив підтримання програмного продукту, подальший розвиток продовжила opensource-спільнота, випустивши Postgres95. В 1996-у році Postgres було перейменовано в PostgreSQL, для підкреслення підтримки нових версій SQL.

Крім базової системи, реалізовано додаткові модулі: *tablefunc, dict_xsyn, fuzzystrmatch, pg_trgm, cube*. Інструкції щодо встановлення містяться на Веб-ресурсі *www.postgresql.org/docs/XX/static/contrib.html*, де *XX* – версія СБД.

1.3.2. Встановлення системи баз даних

Встановлюємо СБД в вигляді контейнера платформи Docker.

docker run --name some-postgres -e POSTGRES_USER=postgres -e POSTGRES_PASSWORD=postgres -d postgres

```
PS C:\Users\dbs> docker run --name some-postgres -e POSTGRES_USER=postgres -e POSTGRES_PASSWORD=postgres -d postgres
Unable to find image 'postgres:latest' locally
latest: Pulling from library/postgres
bf5952930446: Pull complete
9577476abb00: Pull complete
2bd105512d5c: Pull complete
b1cd21c26e81: Pull complete
34a7c86cf8fc: Pull complete
274e7b0c38d5: Pull complete
3e831b350d37: Pull complete
38fa0d496534: Pull complete
c989da35e5c0: Pull complete
26dc6fdd7b2d: Pull complete
3c5032512cf3: Pull complete
26910ececf99: Pull complete
0339413523e8: Pull complete
d61df7db53da: Pull complete
Digest: sha256:9f325740426d14a92f71013796d98a50fe385da64a7c5b6b753d0705add05a21
Status: Downloaded newer image for postgres:latest
3f3340faf6a463b48c5039b9db461994c5b36fa8ed44fae342aba93825f260bb
PS C:\Users\dbs>
```

Рисунок 1.1. Встановлено СБД PostgreSQL в вигляді контейнера

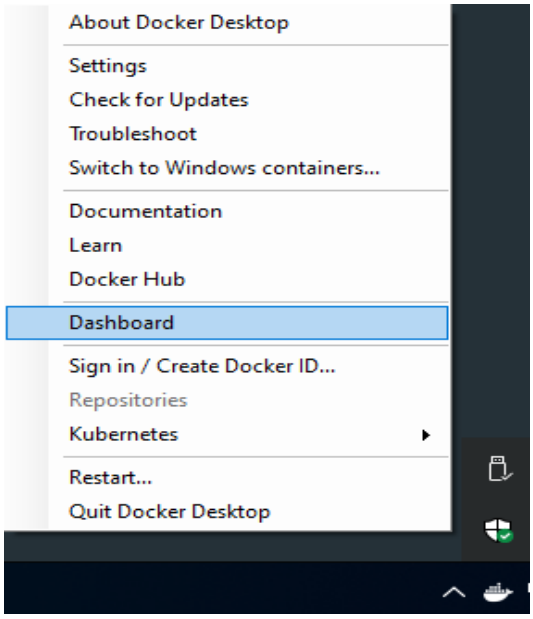


Рисунок 1.2. Список виконуваних контейнерів

Дивимось список виконуваних контейнерів й виконуємо запуск контейнера СБД PostgreSQL.

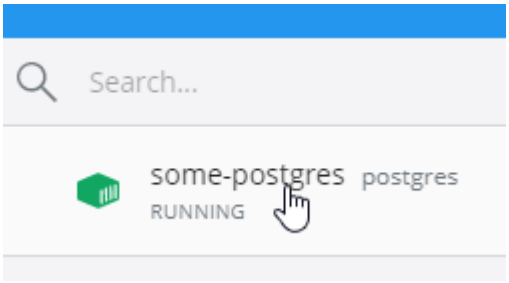


Рисунок 1.3. Виконано запуск контейнера

Виконуємо запуск консолі контейнеру.

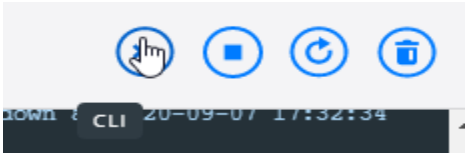


Рисунок 1.4. Виконано запуск консолі контейнеру

Перемістимось в обліковий запис *postgres*, наявний в ОС контейнеру.

```
# su postgres
postgres@7377b66591e8:/$
```

Рисунок 1.5. Виконано переміщення в обліковий запис

1.3.3. Створення бази даних

Як і більшість систем баз даних, СБД PostgreSQL містить сервер і консоль для підключення до сервера [15; 18].

Встановивши СБД, створимо базу даних «*booking*».

$ createdb booking

Включимо консоль PostgreSQL psql.

$ psql booking

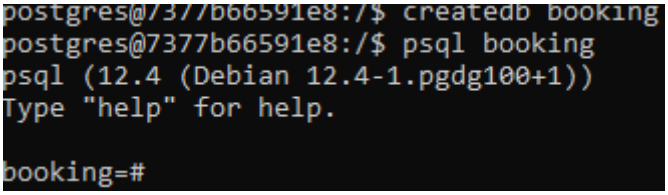


Рисунок 1.6. Виконано створення бази даних

Консоль відображає вітання, що складається з назви бази даних, далі слідує символ « # », якщо виконано автентифікацію адміністратора, або « $ » в протилежному випадку.

Команда « \h » відображає документацію щодо наявних запитів, « \q » – виконує закриття консолі.

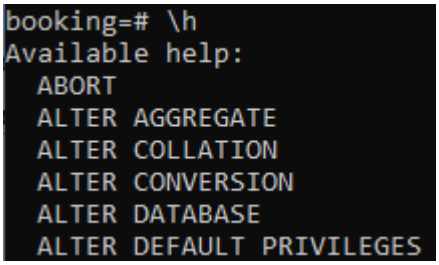


Рисунок 1.7. Відображено довідку щодо операторів SQL

Довідка щодо оператора SQL формується за шаблоном « *booking=# \h CREATE* » (див. рис. 1.8).

```
booking=# \h CREATE
Command:     CREATE ACCESS METHOD
Description: define a new access method
Syntax:
CREATE ACCESS METHOD name
    TYPE access_method_type
    HANDLER handler_function

URL: https://www.postgresql.org/docs/12/sql-create-access-method.html

Command:     CREATE AGGREGATE
Description: define a new aggregate function
Syntax:
CREATE [ OR REPLACE ] AGGREGATE name ( [ argmode ] [ argname ] arg_data_type [ , ... ] ) (
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , SSPACE = state_data_size ]
    [ , FINALFUNC = ffunc ]
    [ , FINALFUNC_EXTRA ]
    [ , FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , COMBINEFUNC = combinefunc ]
```

Рисунок 1.8. Відображено довідку щодо оператора *CREATE*

1.3.4. Мова запитів SQL

В PostgreSQL таблиці іменуються «сутностями» (TABLE), стовпці – «полями» (COLUMN), рядки – «записами» (ROW) [16-18]. В літературі також застосовують поняття: «*відношення*», «*атрибут*», «*кортеж*».

Проектування реляційної БД включає розробку трьох моделей: концептуальної (ER-модель конкретизує сутності і взаємозв'язки), логічної (реляційна модель конкретизує первинні і зовнішні ключі, типи полів таблиць), фізична (SQL-запити для створення схеми БД щодо конкретної СБД). SQL-запити формуються з врахуванням «реляційної алгебри», зокрема: вибірка – *WHERE*, проекція – *SELECT*, декартовий добуток – *FROM*.

Відповідність між реляційним виразом і послідовністю SQL-операторів зображено в табл. 1.1.

Після зазначення взаємозв'язків між таблицями, типів і умов щодо значень стовпців (логічний рівень проектування бази даних), їх опису мовою SQL (фізичний рівень проектування БД), починають дозаписувати дані, сумісні зі створеною схемою БД.

В кожній таблиці необхідно виокремити стовпець-ідентифікатор, котрий буде забезпечувати відмінність всіх рядків. Ідентифікатор містить унікальне значення і іменується «первинним ключем» *PRIMARY KEY*.

Таблиця 1.1. Відповідність між виразом реляційної алгебри і SQL-запитом

Вираз реляційної алгебри	$\pi_{country\_name}\left(\sigma_{country\_code=\omega}\left(\rho_x\left(countries\right)\right)\right)$
Проекція за стовпцем *country_name*	*SELECT x.country_name*
Створення змінної *x*	*FROM countries x*
Умова щодо значень стовпця *country_code*	*WHERE x.country_code IS NULL*
SQL-запит	*SELECT x.country_name* *FROM countries x* *WHERE x.country_code IS NULL*

Створимо таблицю *countries*.

CREATE TABLE countries (country_code char(2) PRIMARY KEY, country_name text UNIQUE);

```
booking=# CREATE TABLE countries ( country_code char(2) PRIMARY KEY, country_name text UNIQUE );
CREATE TABLE
```

Рисунок 1.9. Створено таблицю

1.3.5. CRUD-запити до бази даних

CRUD – абревіатура основних запитів до баз даних: *create* (дозаписування), *read* (зчитування), *update* (оновлення), *delete* (видалення). Значна кількість запитів до СБД, крім операторів *CREATE*, *UPDATE*, *DELETE*, виконують зчитування БД оператором *SELECT* [16; 17; 19].

Створена в попередньому підрозділі таблиця *countries* містить умови, за котрими кожен рядок країни, ідентифікується двозначним первинним ключем *country_code* і іменуванням *country_name* з умовою *UNIQUE*.

Створимо декілька екземплярів даних таблиці *countries*.

```
INSERT INTO countries ( country_code, country_name )
VALUES ('ua','Ukraine'),
('uk','United Kingdom'),
('us','United States'),
('mx','Mexico'),
('wl','Windland');
```

```
booking=# INSERT INTO countries ( country_code, country_name )
booking-# VALUES ('ua','Ukraine'),
booking-# ('uk','United Kingdom'),
booking-# ('us','United States'),
booking-# ('mx','Mexico'),
booking-# ('wl','Windland');
INSERT 0 5
```

Рисунок 1.10. Дозаписано в таблицю декілька екземплярів даних

СБД поверне помилку відповідно до умови унікальності при спробі дозаписування в таблицю *countries* екземплярів даних з ідентичними значеннями стовпця *country_name*.

Перевіряємо, чи виконується умова унікальності.

```
INSERT INTO countries VALUES ('ua', 'Ukraine');
```

```
booking=# INSERT INTO countries VALUES ('ua', 'Ukraine');
ERROR:  duplicate key value violates unique constraint "countries_pkey"
DETAIL:  Key (country_code)=(ua) already exists.
```

Рисунок 1.11. Повернуто помилку створення ідентичного екземпляра даних

Для з'ясування, які екземпляри даних наявні в таблиці, зчитаємо значення всіх стовпців SQL-запитом « *SELECT * FROM <table_name> ;* ».

*SELECT * FROM countries;*

```
booking=# SELECT * FROM countries;
 country_code |   country_name
--------------+-----------------
 ua           | Ukraine
 uk           | United Kingdom
 us           | United States
 mx           | Mexico
 wl           | Windland
(5 rows)
```

Рисунок 1.12. Зчитано повністю вміст таблиці

З повернутого результату слідує, що в таблиці є екземпляр даних з тестовою назвою «*Windland*». Для прикладу оператором *DELETE* видалимо екземпляр зі значенням «*wl*» в стовпці *country_code*.

DELETE FROM countries WHERE country_code = 'wl';

```
booking=# DELETE FROM countries WHERE country_code = 'wl';
DELETE 1
```

Рисунок 1.13. Видалено екземпляр даних таблиці

Створимо таблицю *cities*, параметр *REFERENCES* позначає умову відповідності значень зовнішнього ключа (англ. *foreign key*) *country_code*

таблиці *cities* до значень первинного ключа *country_code* таблиці *countries* (див. табл. 1.2).

PostgreSQL підтримує різноманіття типів даних. На рис. 1.14 зображено приклад застосування строкових типів даних: *text* – довільна послідовність з нефіксованою кількістю символів; *varchar*(9) – довільна послідовність з межею 9 символів; *char*(2) – фіксована послідовність з 2-х символів.

Таблиця 1.2. Зв'язування зовнішнім ключем

Створювана таблиця *cities*		
name	***postal_code***	***country_code***
Kyiv	01001	ua
…	01…	ua
…	…	us
…	…	us
…	01…	...

↔

Наявна таблиця *countries*	
country_code	***country_name***
ua	Ukraine
uk	United Kingdom
us	United States
mx	Mexico
…	…

```
CREATE TABLE cities (
name text NOT NULL,
postal_code varchar(9) CHECK (postal_code <> ''),
country_code char(2) REFERENCES countries,
PRIMARY KEY (country_code, postal_code)
);
```

```
booking=# CREATE TABLE cities (
booking(# name text NOT NULL,
booking(# postal_code varchar(9) CHECK (postal_code <> ''),
booking(# country_code char(2) REFERENCES countries,
booking(# PRIMARY KEY (country_code, postal_code)
booking(# );
CREATE TABLE
```

Рисунок 1.14. Створено таблицю зі складеним первинним ключем

Запит *CREATE TABLE* містить складений «*первинний ключ*» (англ. *primary key*) (*country_code*, *postal_code*), котрий однозначно ідентифікує

окремі екземпляри даних таблиці *cities*; умова *NOT NULL* позначає недопустимість невстановленого значення; послідовність « <> » позначає умову «*не дорівнює*».

Дозапишемо в *cities* екземпляр даних з тестовим значенням поля *name* – «*Tocity*» і зовнішнім ключем *country_code* – «*rr*».

INSERT INTO cities VALUES ('Tocity', '810900', 'rr');

```
booking=# INSERT INTO cities VALUES ('Tocity', '810900', 'rr');
ERROR:  insert or update on table "cities" violates foreign key constraint
"cities_country_code_fkey"
DETAIL:  Key (country_code)=(rr) is not present in table "countries".
```

Рисунок 1.15. Відхилено створення екземпляра таблиці з помилковим значенням зовнішнього ключа

Так як зовнішній ключ *country_code* і параметр *REFERENCES* зв'язують таблицю *cities* з *countries*, СБД визначає, чи міститься в таблиці *countries* екземпляр даних зі значенням «*rr*». Так як в таблиці *countries* немає даного значення, СБД поверне відповідь щодо наявності помилки в запиті.

Створимо екземпляр даних міста *Kyiv*.

INSERT INTO cities VALUES ('Kyiv','01011','ua');

```
booking=# INSERT INTO cities VALUES ('Kyiv','01011','ua');
INSERT 0 1
```

Рисунок 1.16. Дозаписано значення з врахуванням умови узгодженості за зовнішнім взаємозв'язком

Запит ймовірно буде виконано без помилки, але поштовий індекс *postal_code* міста *Kyiv* зазначено неточно, коректне значення – «*01001*». Замість видалення і повторного створення екземпляру, натомість необхідно оновити значення в стовпці *name*.

UPDATE cities SET postal_code = '01001' WHERE name = 'Kyiv';

```
booking=# UPDATE cities SET postal_code = '01001' WHERE name = 'Kyiv';
UPDATE 1
```

Рисунок 1.17. Оновлено екземпляр даних таблиці

1.3.6. Зв'язування таблиць

Внутрішнє зв'язування

«Зв'язування» – об'єднання екземплярів даних таблиць. Розповсюджене внутрішнє зв'язування *INNER JOIN* або скорочено *JOIN*.

SELECT cities., country_name*
FROM cities INNER JOIN countries
ON cities.country_code = countries.country_code;

Внутрішнє зв'язування об'єднує таблиці *cities* і *countries* за рядками, в котрих значення стовпців *country_code* є ідентичними. В відповідь на запит СБД сформує тимчасову проекцію зі значень всіх стовпців таблиці *cities.** і стовпця *country_code* таблиці *countries.*

```
booking=# SELECT cities.*, country_name
booking-# FROM cities INNER JOIN countries
booking-# ON cities.country_code = countries.country_code;
 name | postal_code | country_code | country_name
------+-------------+--------------+--------------
 Kyiv | 01001       | ua           | Ukraine
(1 row)
```

Рисунок 1.18. Зв'язано дві таблиці оператором *INNER JOIN*

Зв'язування також допускається стосовно таблиць, в котрих первинний ключ є складеним (див. рис. 1.14). Створимо таблицю громадських місць, взаємозв'язану складеним зовнішнім ключем з первинними ключем таблиці *cities*. Умова *MATCH FULL* гарантує, що значення стовпців зовнішнього ключа існують в таблиці *cities* або дорівнюють *NULL*.

CREATE TABLE venues (
venue_id SERIAL PRIMARY KEY,
name varchar(255),
street_address text,
type char(7) CHECK (type in ('public','private')) DEFAULT 'public',
country_code char(2),
postal_code varchar(9),
FOREIGN KEY (country_code, postal_code) REFERENCES cities (country_code, postal_code) MATCH FULL
);

```
booking=# CREATE TABLE venues (
booking(# venue_id SERIAL PRIMARY KEY,
booking(# name varchar(255),
booking(# street_address text,
booking(# type char(7) CHECK (type in ('public','private') ) DEFAULT
 'public',
booking(# country_code char(2),
booking(# postal_code varchar(9),
booking(# FOREIGN KEY (country_code, postal_code) REFERENCES cities
(country_code, postal_code) MATCH FULL
booking(# );
CREATE TABLE
booking=#
```

Рисунок 1.19. Створено таблицю з умовою *MATCH FULL* складеного зовнішнього ключа

Стовпець *venue_id* – первинний ключ, тип лічильник *SERIAL*. Так як початкове значення лічильника не вказано явно в запиті створення таблиці,

лічильник буде збільшуватись на одиницю починаючи зі значення «1» при дозаписуванні нових екземплярів даних.

```
INSERT INTO venues (name, postal_code, country_code)
VALUES ('Crystal Ballroom', '01001', 'ua');
```

```
booking=# INSERT INTO venues (name, postal_code, country_code)
booking-# VALUES ('Crystal Ballroom', '01001', 'ua');
INSERT 0 1
```

Рисунок 1.20. Дозаписано екземпляр даних зі складеним зовнішнім ключем

Для друку в консоль значення поля екземпляру, дозаписаного в БД, запит необхідно завершити оператором *RETURNING*.

```
INSERT INTO venues (name, postal_code, country_code)
VALUES ('Event Donuts', '01001', 'ua')
RETURNING venue_id;
```

```
booking=# INSERT INTO venues (name, postal_code, country_code)
booking-# VALUES ('Event Donuts', '01001', 'ua')
booking-# RETURNING venue_id;
 venue_id
----------
        2
(1 row)

INSERT 0 1
```

Рисунок 1.21. Зчитано результат запиту в консоль

СБД поверне відповідь зі значенням ідентифікатора *venue_id* створеного екземпляра даних.

При зв'язуванні таблиць *venues* і *cities* необхідно зазначити стовпці, котрі формують складений ключ. Для скорочення написання SQL-запитів,

назвам таблиць ставлять в відповідність змінні після дійсних назв стовпців і необов'язкового оператора *AS* (наприклад, *venues* v чи *venues AS v*).

SELECT v.venue_id, v.name, c.name
FROM venues v INNER JOIN cities c
ON v.postal_code=c.postal_code AND v.country_code=c.country_code;

СБД поверне відповідь в вигляді таблиці зі значеннями стовпців *v.venue_id, v.name, c.name*.

```
booking=# SELECT v.venue_id, v.name, c.name
booking-# FROM venues v INNER JOIN cities c
booking-# ON v.postal_code=c.postal_code AND v.country_code
=c.country_code;
 venue_id |       name       | name
----------+------------------+------
        2 | Event Donuts     | Kyiv
        1 | Crystal Ballroom | Kyiv
(2 rows)

booking=#
```

Рисунок 1.22. Зв'язано таблиці оператором *INNER JOIN*

Зовнішнє зв'язування

Крім внутрішнього зв'язування, в PostgreSQL існує оператор зовнішнього зв'язування *OUTER JOIN*, котрий включає в результат об'єднання всі рядки таблиці зліва чи справа від оператора, дивлячись за типом зв'язування, лівостороннє чи правостороннє. Створимо нову таблицю зі стовпцями: ідентифікатор заходу *event_id* (цілочисельний лічильник, тип *SERIAL*), назва *title* (строковий тип), часові позначки початку *starts* і завершення *ends* (тип *timestamp*), ідентифікатор місця проведення *venue_id* (зовнішній ключ, взаємозв'язаний з таблицею *venues*).

Тепер, подібно до вже виконаних запитів, необхідно створити таблицю *events*, доповнивши її трьома екземплярами даних, відповідно до табл. 1.3.

Таблиця 1.3. Таблиця events

title	***starts***	***ends***	***venue_id***	***event_id***
LARP Club	2012-02-14	2012-02-15	2	1
April Fools Day	2012-04-01	2012-04-09		2
Christmas Day	2012-12-20	2012-12-25		3

На рис. 1.23 зображено ER-модель (англ. Entity-Relationship, «сутність-зв'язок») створеної бази даних [15; 17; 18].

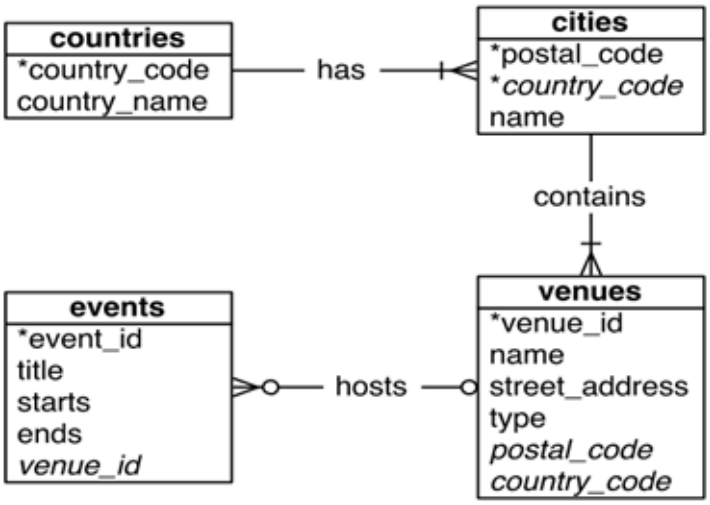


Рисунок 1.23. ER-модель бази даних

Для зчитування іменувань всіх заходів, навіть тих, для котрих місце проведення невідоме, тобто дорівнює *NULL*, потрібно застосувати оператор *LEFT OUTER JOIN* або скорочено *LEFT JOIN*.

SELECT e.title, v.name FROM events e LEFT JOIN venues v ON e.venue_id = v.venue_id;

СБД поверне тимчасову проекцію екземплярів даних таблиці *events* за стовпцем *title,* доповнену значеннями проекції таблиці *venues*, створеної за стовпцем *name*.

```
booking=# SELECT e.title, v.name FROM events e LEFT JOIN venues v ON e.venue_id = v.venue_id;
     title      |     name
----------------+--------------
 LARP Club      | Event Donuts
 April Fools Day |
 Christmas Day  |
(3 rows)
```

Рисунок 1.24. Зв'язано таблиці оператором *LEFT JOIN*

Правостороннє зовнішнє зв'язування *RIGHT OUTER JOIN* або скорочено *RIGHT JOIN* навпаки поверне тимчасову проекцію зі значеннями стовпця *venues.name*, доповнених значеннями стовпця *events.title*.

```
booking=# SELECT v.name, e.title FROM events e RIGHT JOIN venues v ON e.venue_id = v.venue_id;
      name       |   title
-----------------+-----------
 Event Donuts    | LARP Club
 Crystal Ballroom |
(2 rows)
```

Рисунок 1.25. Зв'язано дві таблиці оператором *RIGHT JOIN*

Для двостороннього зовнішнього зв'язування існує також оператор *FULL OUTER JOIN* або *FULL JOIN*, котрий об'єднує результати застосування операторів *LEFT JOIN* і *RIGHT JOIN* щодо вказаних таблиць.

```
booking=# SELECT e.title, v.name FROM events e FULL JOIN venues v ON e.venue_id = v.venue_id;
     title      |       name
----------------+------------------
 LARP Club      | Event Donuts
 April Fools Day |
 Christmas Day  |
                | Crystal Ballroom
(4 rows)
```

Рисунок 1.26. Зв'язано таблиці оператором *FULL JOIN*

Зчитаємо назви заходів і місць їх проведення застосовуючи оператор внутрішнього зв'язування. Параметр *INNER* в операторі *INNER JOIN* є необов'язковим, тому в SQL-запиті зазначимо *JOIN*.

SELECT e.title, v.name FROM events e JOIN venues v ON e.venue_id = v.venue_id;

Запит *JOIN* поверне відповідь в вигляді перетину екземплярів даних таблиць *events* і *venues* за стовпцем *venue_id*.

```
booking=# SELECT e.title, v.name FROM events e JOIN venues v ON e.venue_id = v.venue_id;
   title   |     name
-----------+--------------
 LARP Club | Event Donuts
(1 row)
```

Рисунок 1.27. Зв'язано таблиці оператором *INNER JOIN*

1.3.7. Створення індексів

Індекс – спеціалізована структура даних, котра усуває необхідність повного перебору екземплярів даних при виконанні запиту до БД. Індексні структури забезпечують оптимізацію запитів до СБД, зменшуючи кількість запитів до файлу бази даних, розміщеного в зовнішній пам'яті.

Прискорення перебору екземплярів даних БД теж має межі [13-15; 17]. Без *індексів* було б необхідно переглянути всі екземпляри даних, для з'ясування необхідності зчитування окремих рядків таблиці.

В відповідь на запит створення таблиці СБД друкує повідомлення «*CREATE TABLE*», PostgreSQL автоматично створює індекс за первинним ключем. Ключем індексу *events.index* є значення стовпця *events.event_id*, а

«значенням» – посилання щодо розташування в зовнішній пам'яті [16; 19; 21; 22].

Таблиця 1.4. Індекс за значенням первинного ключа

events.index	*events.title*	*venues.venue_id*	*events.event_id*
1	LARP Club	2	1
2	April Fools Day		2
3	Christmas Day		3

Для створення індексу за конкретним стовпцем зазначають умову *UNIQUE*.

Декларативно індекс створюється запитом *CREATE INDEX*.

CREATE INDEX events_index ON events USING hash (event_id);

```
booking=# CREATE INDEX events_index ON events USING hash (event_id);
CREATE INDEX
```

Рисунок 1.28. Створено хеш-індекс

За потреби в співставленні значень стовпців за операторами «<», «>», індекс необхідно зберігати в гнучкішій структурі даних, наприклад, B-дереві (див. рис. 1.29).

Сформуємо запит зчитування заходів з датою проведення «після 1-го квітня включно».

*SELECT * FROM events WHERE starts >= '2012-04-01';*

```
booking=# SELECT * FROM events WHERE starts >= '2012-04-01';
 event_id |      title      |   starts   |    ends    | venue_id
----------+-----------------+------------+------------+----------
        2 | April Fools Day | 2012-04-01 | 2012-04-09 |
        3 | Christmas Day   | 2012-12-20 | 2012-12-25 |
(2 rows)
```

Рисунок 1.30. Знайдено екземпляри даних за умовою щодо значень

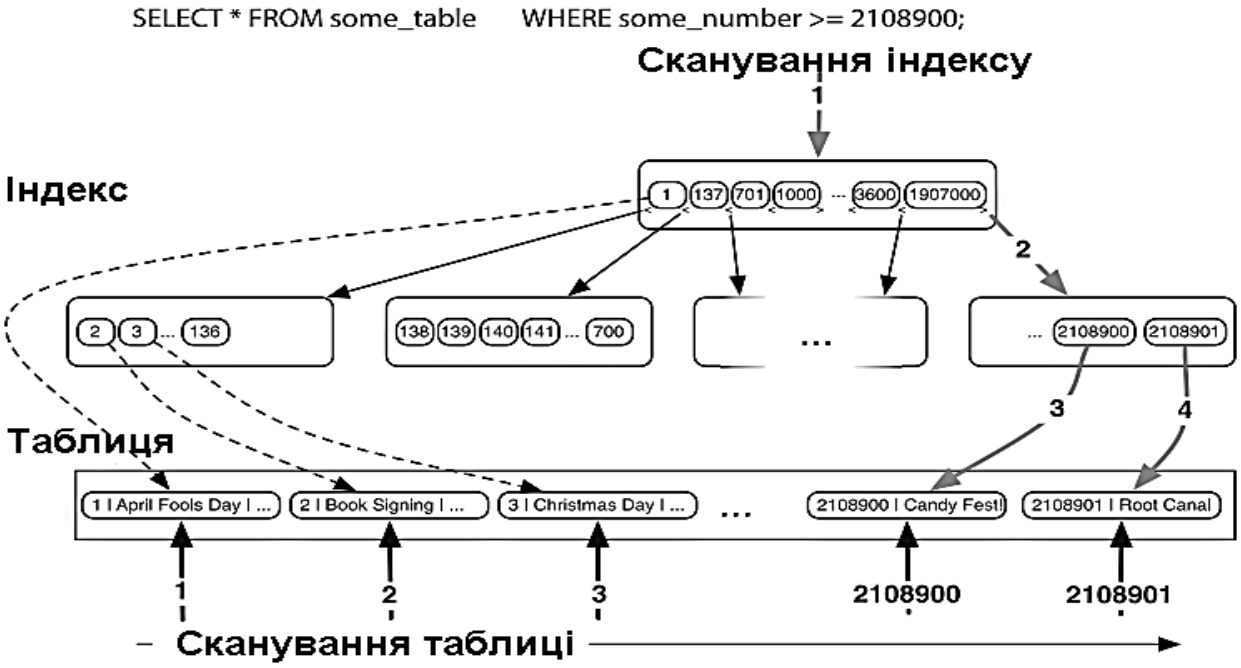


Рисунок 1.29. Зчитано значення за інтервалом з індексу
зі структурою B-дерево

Для даного запиту ідеальною структурою є дерево. Для побудови B-дерева (*btree*) за полем *events.starts* побудуємо відповідний індекс.

CREATE INDEX events_starts ON events USING btree (starts);

```
booking=# CREATE INDEX events_starts ON events USING btree (starts);
CREATE INDEX
```

Рисунок 1.31. Створено індекс зі структурою B-дерево

Тепер при виконанні запиту за діапазоном дат немає потреби в перегляді всієї таблиці, так як при перегляді млн. або млрд. екземплярів даних різниця в затримці відповіді СБД буде значною [19; 21].

СБД автоматично створює індекс за стовпцями з умовою *FOREIGN KEY*. Зчитаємо список всіх індексів бази даних – « *booking=# \di* ».

```
booking=# booking=# \di
                        List of relations
 Schema |             Name              | Type  |  Owner   |   Table
--------+-------------------------------+-------+----------+-----------
 public | cities_pkey                   | index | postgres | cities
 public | countries_country_name_key    | index | postgres | countries
 public | countries_pkey                | index | postgres | countries
 public | events_index                  | index | postgres | events
 public | events_pkey                   | index | postgres | events
 public | events_starts                 | index | postgres | events
 public | venues_pkey                   | index | postgres | venues
(7 rows)
```

Рисунок 1.32. Зчитано наявні індекси

Зупиняємо виконання контейнеру.

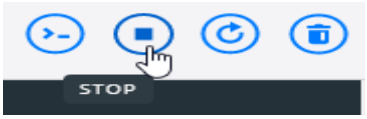


Рисунок 1.33. Зупинено виконання контейнеру

Вимикаємо контейнер.

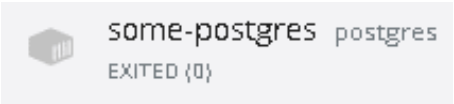


Рисунок 1.34. Вимкнено контейнер

1.4. Висновки

Поняття	**Значення**
Стовпець	Список значень, котрі характеризуються певним типом.

Рядок	Об'єкт, що складається з послідовності значень стовпців; іноді називається також «кортежем».
Таблиця	Послідовність рядків з відповідними значеннями стовпців.
Первинний ключ	Унікальне значення для ідентифікації конкретного рядка.
CRUD	Абревіатура Create, Read, Update, Delete.
SQL	Мова структурованих запитів (Structured Query Language).
Зв'язування	Об'єднання рядків таблиць з співставленням значень стовпців.
Лівостороннє зв'язування	Спосіб об'єднання двох таблиць, при котрому всі рядки з лівосторонньої таблиці включаються в тимчасову проекцію; за невідповідності значень стовпців з правосторонньої таблиці дозаписуються значення *NULL*.
Індекс	Структура даних для оптимізації пошуку екземплярів даних таблиці за умовами до значень стовпців.
B-дерево	Структура даних індексу; значення зберігаються в збалансованому дереві.

1.4.1. Теоретичне завдання

1. Встановіть в браузері закладки на документацію і FAQ PostgreSQL.
2. Прочитайте довідку за запитами « \? » і « \h ».
3. Зміст параметра *MATCH FULL* зовнішнього ключа.

1.4.2. Практичне завдання

1. Обчисліть кількість екземплярів даних окремо за всіма таблицями, відповідна «агрегатна» функція є в документації СБД.
2. Зчитайте назву країни за заходом LARP.
3. Доповніть наявну таблицю *venues* булевим стовпцем *venues.private* з попередньо встановленим значенням *TRUE*.

1.4.3. Питання для закріплення знань

1. Випадки, коли рекомендовано застосовувати реляційні СБД.
2. Випадки, коли не рекомендовано застосовувати реляційні СБД.

3. Сутність поняття «узгодженість взаємозв'язків». Назначення первинного і зовнішнього ключів.

4. Основні CRUD-запити, їх назначення. Типи даних в PostgreSQL.

5. Оператор внутрішнього зв'язування. Синтаксис, параметри, результат зв'язування.

6. Оператори зовнішнього зв'язування. Синтаксис, параметри, результат зв'язування.

7. Оператори реляційної алгебри, відповідні оператори мови SQL.

Розділ 2. Системи документних баз даних

Системи документних баз даних розміщують екземпляри даних в документах з деревоподібною структурою, подібна структура застосовується в різноманітних предметних областях. Два основних гравця серед документних СБД з відкритим вихідним кодом – CouchDB і MongoDB.

2.1. Випадки, коли рекомендовано застосовувати

Документні СБД застосовують при вирішенні завдань, котрі характеризуються мінливістю структури БД. Як і MongoDB, CouchDB зберігає документи в форматі JavaScript Object Notation (абрв. JSON). JSON-документи узгоджені з об'єктно-орієнтованою моделлю програмного забезпечення [19-22].

2.2. Випадки, коли не рекомендовано застосовувати

Для документних СБД невластива гнучкість формування запитів з врахуванням відношень між сутностями, подібно до реляційних баз даних. Документ містить всю чи переважну частину інформації щодо об'єкту. В реляційних СБД умовою узгодженого стану БД є нормалізований вміст таблиць, коли мінімізовано повторення значень в стовпцях, натомість в документних СБД «денормалізоване» зберігання даних є звичайною справою.

2.3. СБД CouchDB

Документ містить масив кортежів «ключ/значення» і унікальні поля зі значеннями ідентифікатора і версії створення/оновлення документу. Пошук документів реалізовано на базі «індексованих проекцій», створюваних за моделлю розподілених обчислень map-reduce [19; 20].

Так як окремі документи невзаємозв'язані, СБД даного типу має сенс застосовувати для завдань практично будь-якої складності. Документні СБД

здатні розподілятись вертикально, за умови поліпшення характеристик апаратного забезпечення серверу, або горизонтально – при розподілі БД між об'єднаними в кластер обчислювальними вузлами.

2.3.1. Передісторія

СБД розроблена в 2005-у році, як для застосування в центрах обробки даних (ЦОД), з'єднаних з мережею Інтернет, так і для мобільних пристроїв – ноутбуків, смартфонів [19; 21]. Модель зберігання даних допускає тільки дозаписування нових документів.

Видалення екземплярів даних залишає в БД всі попередні версії документу, завершальна версія позначається як «видалена». Подібна особливість зберігання даних гарантує відновлюваність стану БД в випадку зупинення сервера або відключення мережевого з'єднання.

2.3.2. Встановлення системи баз даних

Встановлюємо СБД в вигляді контейнера платформи Docker.

docker run --name some-couchdb -e 'COUCHDB_USER=couchdb' -e 'COUCHDB_PASSWORD=couchdb' -p 5984:5984 -d couchdb

```
PS C:\Users\dbs> docker run --name some-couchdb -e 'COUCHDB_USER=couchdb' -e 'COUCHDB_PASSWORD=couchdb' -p 5984:5984 -d couchdb
Unable to find image 'couchdb:latest' locally
latest: Pulling from library/couchdb
bf5952930446: Already exists
0133380524ef: Pull complete
9c1461e31bfd: Pull complete
6d4318fadf22: Pull complete
f59ba1e6f8e2: Pull complete
b2831b8f2c39: Pull complete
365bcd658e2c: Pull complete
ee4bfa72be70: Pull complete
9e0ad5d1ade1: Pull complete
acf89cee2ad7: Pull complete
f4d8f755ddaa: Pull complete
Digest: sha256:6f2c0001fa8074e81a69be7ac32193a2f835eebd4bf3ae3a37a5b53daf89bc62
Status: Downloaded newer image for couchdb:latest
fd2b20028e81453b6e20cea6d6bc3fb5d818b55c852ff6fa3d9be39cc6cf7a44
```

Рисунок 2.1. Встановлено СБД CouchDB в вигляді контейнера

Виконуємо запуск графічного інтерфейсу користувача.

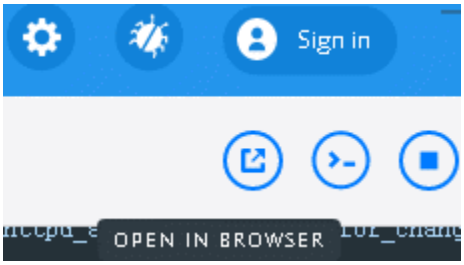


Рисунок 2.2. Виконано запуск Веб-інтерфейсу

2.3.3. Створення бази даних

Веб-інтерфейс Fauxton забезпечує виконання CRUD-запитів до БД, виконаємо автентифікацію за посиланням *http://localhost:5984/_utils/*.

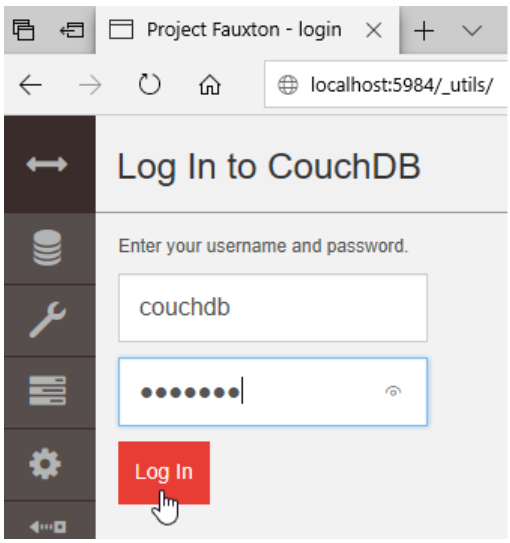


Рисунок 2.3. Інтерфейс автентифікації користувача

Після виконаної автентифікації СБД відображає Веб-сторінку зі списком баз даних.

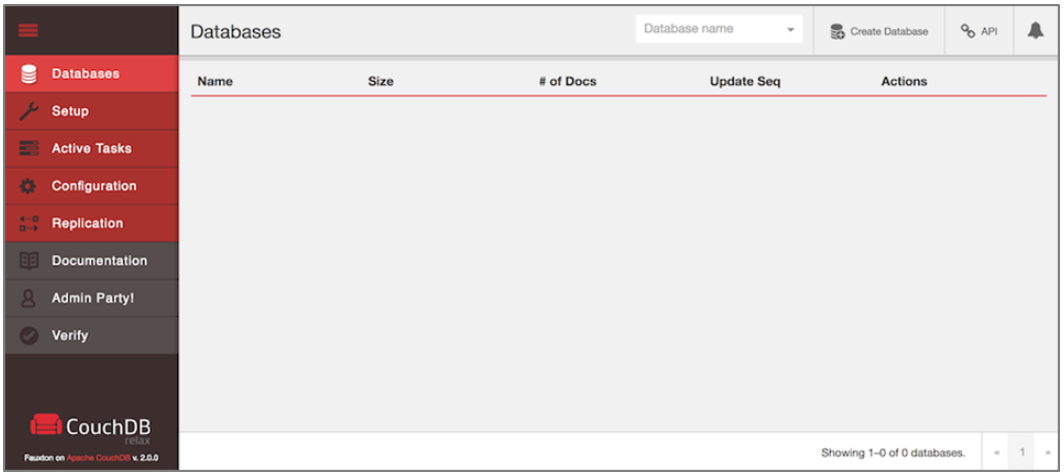


Рисунок 2.4. Відображено список баз даних

Веб-інтерфейс

Перш ніж дозаписувати документи, потрібно створити БД, наприклад, щодо музичних виконавців та їх альбомів. Курсором переміщуємось за посиланням *Create Database*, зазначимо іменування бази даних «*music*» і далі переміщуємось за посиланням *Create*.

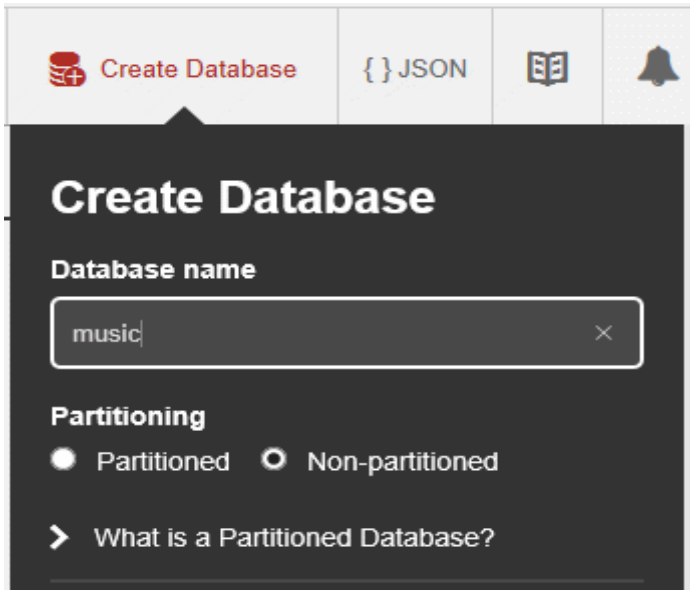


Рисунок 2.5. Створено базу даних

СБД відображає Веб-сторінку, котра містить посилання щодо створення нових або зчитування існуючих документів БД.

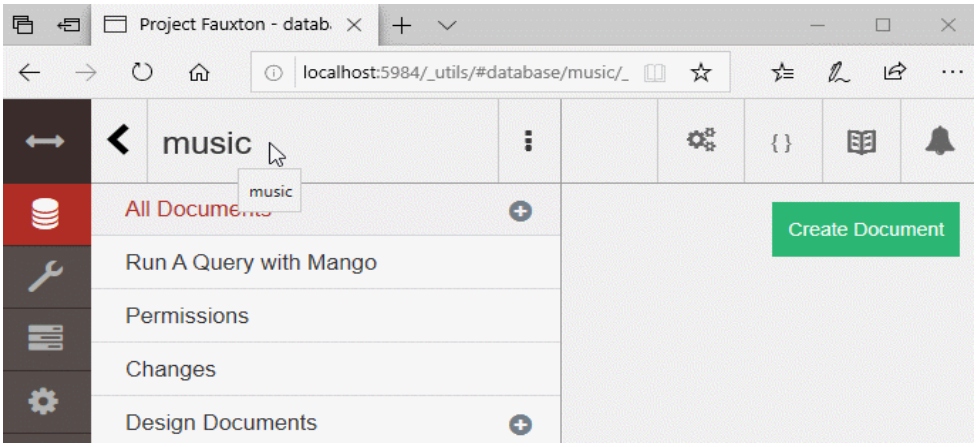


Рисунок 2.6. Відображено інтерфейс створеної бази даних

На сторінці БД «*music*» курсором спочатку переміщуємось за посиланням «плюс» поруч з *All Documents*, далі за *New Document*. СБД створить документ і відобразить Веб-сторінку редагування (див. рис. 2.7).

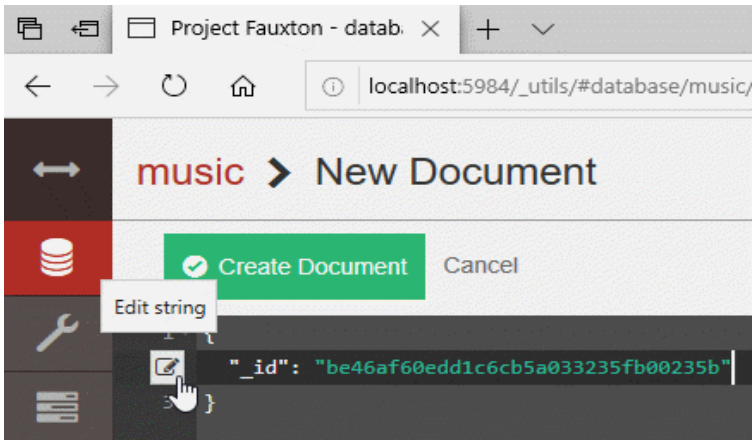


Рисунок 2.7. Відображено інтерфейс редагування документу

Вказано нове значення ідентифікатора документу.

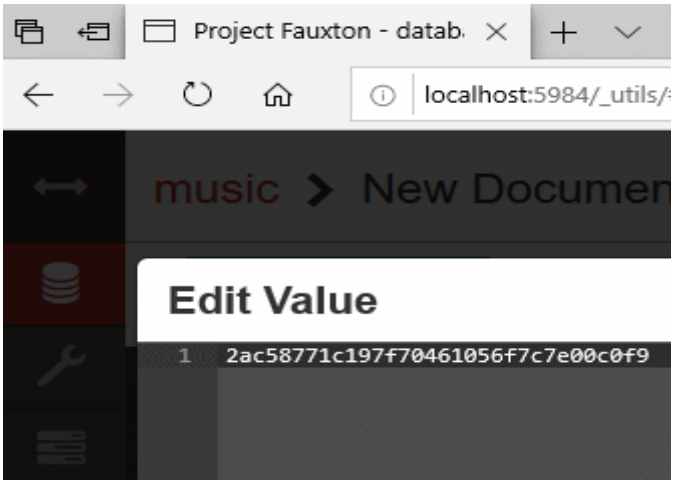


Рисунок 2.8. Вказано значення ідентифікатора документу

Оновлення документа супроводжується збільшенням на 1 значення кортежу з ключем « *_rev* ». Формат версії складається зі значення кількості оновлень, символу розмежування « - » і унікального строкового значення.

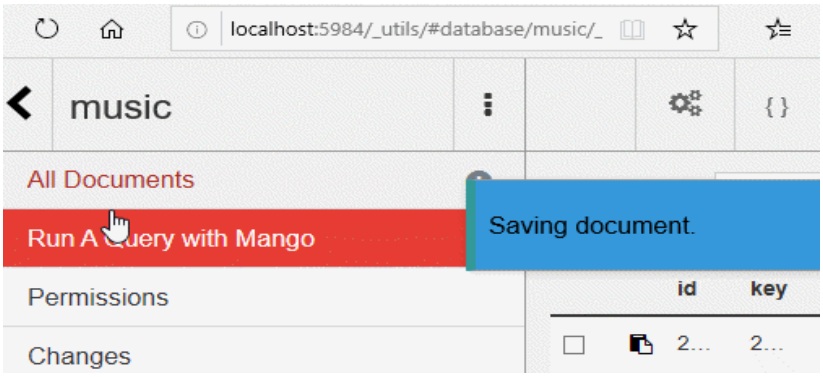


Рисунок 2.9. Оновлено ідентифікатор документу

Особливістю реалізації СБД CouchDB є усунення необхідності виконання транзакцій або блокувань. В процесі оновлення, СБД спочатку зчитує поточні значення полів *_id* і *_rev* і далі створює новий екземпляр документу збільшуючи на одиницю значення версії *_rev*.

При оновленні або видаленні існуючого документу, необхідно знати актуальні значення *_id* і *_rev*, в протилежному випадку СБД відхилить запит, гарантуючи оновлення документу тільки з завершальним номером версії.

Оновимо вміст JSON-документу, дозапишемо кортеж «ключ/значення» з ключем *name* і значенням «*The Beatles*», далі курсором переміщуємось за посиланням *Save Changes*.

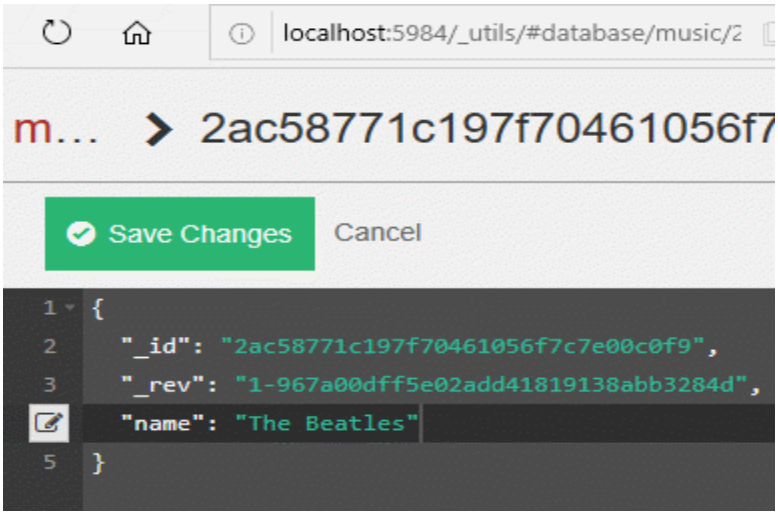


Рисунок 2.10. Оновлено структуру документу

Як було зазначено, СБД CouchDB може зберігати не тільки строкові значення, а і списки чи вкладені JSON-документи. Курсором переміщуємось за посиланням *Add Field*, створимо кортеж з ключем *albums*, і списком назв альбомів виконавця (список неповний) – « *["Help!", "Sgt. Pepper's Lonely Hearts Club Band", "Abbey Road"]* ».

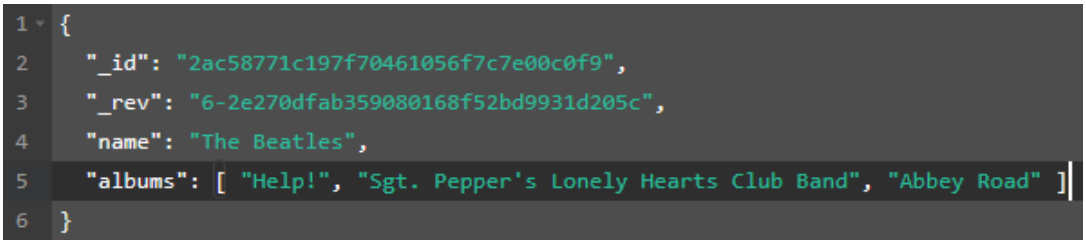


Рисунок 2.11. Вказано список в значенні кортежу

Для оновлення структури документу переміщуємось за посиланням *Save Document* (див. рис. 2.12).

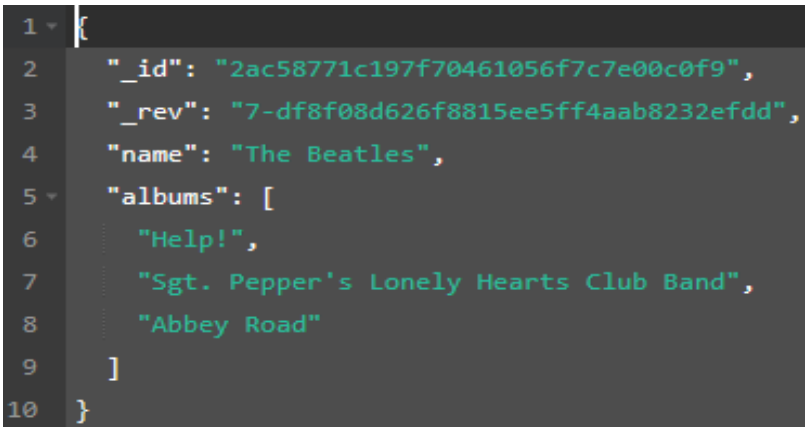


Рисунок 2.12. Оновлено вміст документу

Окрім назв, музичні альбоми характеризуються роками виконання. Оновимо значення кортежу *albums*.

[
{
"title":"Help!",
"year": 1965
}, {
"title":"Sgt. Pepper's Lonely Hearts Club Band",
"year": 1967
}, {
"title":"Abbey Road",
"year": 1969
}
]

Зчитаємо вміст оновленого документу.


```
{
  "_id": "2ac58771c197f70461056f7c7e00c0f9",
  "_rev": "8-e748ce5d01a98e4af7b42b85256c38b5",
  "name": "The Beatles",
  "albums": [
    {
      "title": "Help!",
      "year": 1965
    },
    {
      "title": "Sgt. Pepper's Lonely Hearts Club Band",
      "year": 1967
    },
    {
      "title": "Abbey Road",
      "year": 1969
    }
  ]
}
```


Рисунок 2.13. Зчитано документ з рівнями вкладеності

Виконуємо запуск консолі. Розберемо серію прикладів взаємодії з CouchDB за інтерфейсом REpresentational State Transfer (абрв. REST).

2.3.4. REST-інтерфейс

REST-інтерфейс забезпечує направлення запитів до СБД за мережевим протоколом HTTP.

Виконуємо запуск консолі контейнеру.

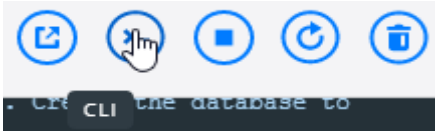


Рисунок 2.14. Виконано запуск консолі контейнеру

Утиліта *cURL* реалізує направлення HTTP-запитів до сервера СБД, без необхідності застосування Веб-інтерфейса.

Виконуємо запит *GET* утилітою *cURL*.

curl http://localhost:5984

Рисунок 2.15. Зчитано відповідь СБД щодо HTTP-запиту до сервера

2.3.5. CRUD HTTP-запити до бази даних

Зазначимо в консолі іменування і порт серверу, дані користувача для направлення запитів до СБД.

export COUCH_ROOT_URL=http://couchdb:couchdb@localhost:5984
curl $COUCH_ROOT_URL

В відповідь на запити *GET*, сервер зчитає інформацію щодо БД або документа, зазначеного в URL. Якщо вказано тільки іменування сервер, тоді СБД відобразить інформацію щодо особливостей виконання сервера і номера встановленої версії (див. рис. 2.16).

Зчитаємо інформацію щодо створеної бази даних «*music*».

curl $COUCH_ROOT_URL/music/

```
# export COUCH_ROOT_URL=http://couchdb:couchdb@localhost:5984

# curl $COUCH_ROOT_URL
{"couchdb":"Welcome","version":"3.1.0","git_sha":"ff0feea20",
"uuid":"ae104405fac594a8cf2f6441a04e16eb","features":["access
-ready","partitioned","pluggable-storage-engines","reshard","
scheduler"],"vendor":{"name":"The Apache Software Foundation"
}}
#
```

Рисунок 2.16. Створено змінну ОС контейнеру

```
# curl $COUCH_ROOT_URL/music/
{"db_name":"music","purge_seq":"0-g1AAAABPeJzLYWBgYMpgTmHgzcv
Py09JdcjLz8gvLskBCeexAEmGBiD1HwiyEhlwqEtkSKqHKMgCAIT2GV4","up
date_seq":"10-g1AAAABPeJzLYWBgYMpgTmHgzcvPy09JdcjLz8gvLskBCee
xAEmGBiD1HwiyEtlxqEtkSKoHK2DOAgCF8Rlo","sizes":{"file":57687,
"external":160,"active":939},"props":{},"doc_del_count":0,"do
c_count":1,"disk_format_version":8,"compact_running":false,"c
luster":{"q":2,"n":1,"w":1,"r":1},"instance_start_time":"0"}
#
```

Рисунок 2.17. Зчитано опис наявної бази даних

Відповідь містить інформацію щодо кількості документів в базі даних, тривалості виконання сервера, кількості оброблених запитів.

Зчитування документу HTTP-запитом GET

Для пошуку документу виконуємо запит *GET* до сервера, зазначивши іменування БД і ідентифікатор документу.

curl $COUCH_ROOT_URL/music/2ac58771c197f70461056f7c7e00c0f9

```
# curl $COUCH_ROOT_URL/music/2ac58771c197f70461056f7c7e00c0f9
{"_id":"2ac58771c197f70461056f7c7e00c0f9","_rev":"8-e748ce5d01a
98e4af7b42b85256c38b5","name":"The Beatles","albums":[{"title":
"Help!","year":1965},{"title":"Sgt. Pepper's Lonely Hearts Club
 Band","year":1967},{"title":"Abbey Road","year":1969}]}
#
```

Рисунок 2.18. Зчитано вміст за ідентифікатором документу

Для оновлення вмісту БД реалізовано HTTP-запити *PUT*, *POST*, *DELETE*.

Створення документу HTTP-запитом POST

Створюємо HTTP-запитом *POST* новий документ з зазначеним типом «*Content-Type: application/json*».

curl -i -XPOST $COUCH_ROOT_URL/music/ -H "Content-Type: application/json" -d '{ "name": "Wings" }'

Відповідь СБД містить технічну інформацію щодо з'єднання. Код «*201 Created*» позначає, що дозаписування виконано коректно.

```
# curl -i -XPOST $COUCH_ROOT_URL/music/ -H "Content-Type: app
lication/json" -d '{ "name": "Wings" }'
HTTP/1.1 201 Created
Cache-Control: must-revalidate
Content-Length: 95
Content-Type: application/json
Date: Wed, 16 Sep 2020 23:21:25 GMT
Location: http://localhost:5984/music/be46af60edd1c6cb5a03323
5fb00689b
Server: CouchDB/3.1.0 (Erlang OTP/20)
X-Couch-Request-ID: a4385d7dda
X-CouchDB-Body-Time: 1

{"ok":true,"id":"be46af60edd1c6cb5a033235fb00689b","rev":"1-2
fe1dd1911153eb9df8460747dfe75a0"}
#
```

Рисунок 2.19. Створено документ HTTP-запитом *POST*

Оновлення документу HTTP-запитом PUT

Як і для *GET*, URL-посилання в HTTP-запиті *PUT* містить іменування бази даних і значення ідентифікатора документу.

curl -i -XPUT
*$COUCH_ROOT_URL/music/be46af60edd1c6cb5a033235fb00689b *
*-H "Content-Type: application/json" *
-d '{ "_id": "74c7a8d2a8548c8b97da748f43000f1b",
"_rev": "1-2fe1dd1911153eb9df8460747dfe75a0",
"name": "Wings",
"albums": ["Wild Life", "Band on the Run", "London Town"] }'

Порівнюючи з СБД MongoDB, котра оновлює вміст існуючих документів, CouchDB при будь-якому оновленні створює нову версію документу [19; 20].

```
# curl -i -XPUT $COUCH_ROOT_URL/music/be46af60edd1c6cb5a03323
5fb00689b \
-H "Content-Type: application/json" \
-d '{ "_id": "74c7a8d2a8548c8b97da748f43000f1b",
"_rev": "1-2fe1dd1911153eb9df8460747dfe75a0",
"name": "Wings",
"albums": ["Wild Life", "Band on the Run", "London Town"] }'>
 > > > >
HTTP/1.1 201 Created
Cache-Control: must-revalidate
Content-Length: 95
Content-Type: application/json
Date: Wed, 16 Sep 2020 23:32:51 GMT
ETag: "2-17e4ce41cd33d6a38f04a8452d5a860b"
Location: http://localhost:5984/music/be46af60edd1c6cb5a03323
5fb00689b
Server: CouchDB/3.1.0 (Erlang OTP/20)
X-Couch-Request-ID: ca7203d581
X-CouchDB-Body-Time: 0

{"ok":true,"id":"be46af60edd1c6cb5a033235fb00689b","rev":"2-1
7e4ce41cd33d6a38f04a8452d5a860b"}
#
```

Рисунок 2.20. Оновлено документ HTTP-запитом *PUT*

При оновленні документу значення полів *_id* і *_rev* мають бути ідентичними до значень полів поточної версії вже існуючого документу, в протилежному випадку запит щодо екземпляру даних завершиться помилкою.

Виконуємо повторно запит *PUT*.

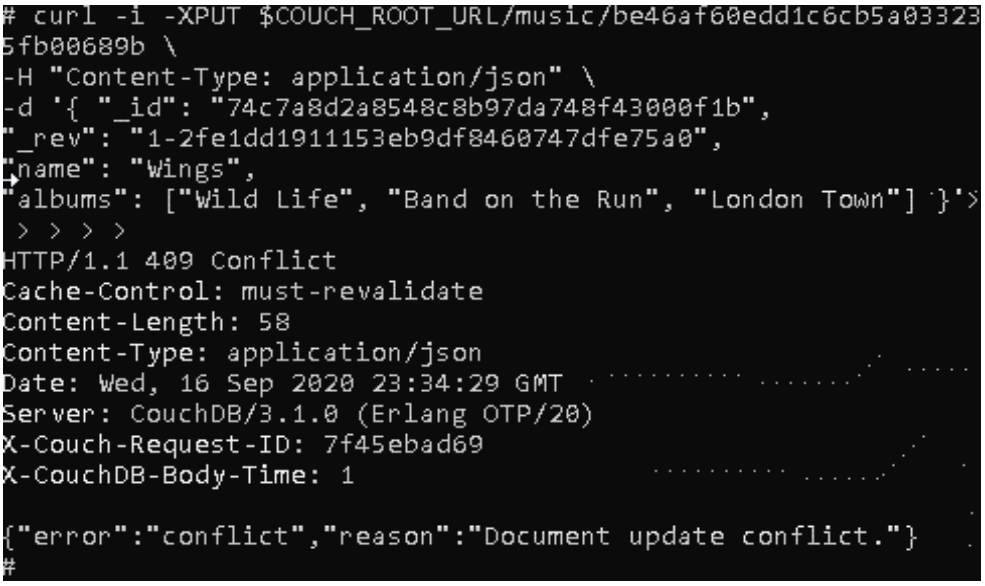


Рисунок 2.21. Відхилено оновлення за значенням попередньої версії документу

Забезпечуючи узгодженість вмісту бази даних, СБД поверне відповідь з кодом «*409 Conflict*» і JSON-описом помилки.

Видалення документу HTTP-запитом DELETE

HTTP-запит *DELETE* видаляє документ з бази даних.

curl -i -XDELETE $COUCH_ROOT_URL/music/
*be46af60edd1c6cb5a033235fb00689b *
-H "If-Match: 2-17e4ce41cd33d6a38f04a8452d5a860b"

Виконавши HTTP-запит *DELETE*, СБД видаляє незарезервовані поля, крім *_id* і *_rev*, і дозаписує поле *_deleted* зі значенням *true*.

```
# curl -i -XDELETE $COUCH_ROOT_URL/music/be46af60edd1c6cb5a03
3235fb00689b \
-H "If-Match: 2-17e4ce41cd33d6a38f04a8452d5a860b">
HTTP/1.1 200 OK
Cache-Control: must-revalidate
Content-Length: 95
Content-Type: application/json
Date: Wed, 16 Sep 2020 23:38:04 GMT
ETag: "3-42aafb7411c092614ce7c9f4ab79dc8b"
Server: CouchDB/3.1.0 (Erlang OTP/20)
X-Couch-Request-ID: dd0aaec79a
X-CouchDB-Body-Time: 0

{"ok":true,"id":"be46af60edd1c6cb5a033235fb00689b","rev":"3-4
2aafb7411c092614ce7c9f4ab79dc8b"}
#
```

Рисунок 2.22. Видалено документ

2.3.6. Створення проекцій

Проекції СБД CouchDB, в певному розумінні, «інтерфейс» доступу до документів, проіндексованих за полями.

Доступ до документів за проекціями

Процес створення проекцій включає кроки розподілення *map* і об'єднання *reduce*, котрі формують відсортований список кортежів «ключ/значення» [13-16; 19-21]. В CouchDB є «загальна» проекція *_all_docs*, котра створюється автоматично і містить значення поточних версій документів.

Для перегляду вмісту проекції *_all_docs* виконуємо запит *GET*.

curl $COUCH_ROOT_URL/music/_all_docs

```
# curl $COUCH_ROOT_URL/music/_all_docs
{"total_rows":1,"offset":0,"rows":[
{"id":"2ac58771c197f70461056f7c7e00c0f9","key":"2ac58771c197f
70461056f7c7e00c0f9","value":{"rev":"4-b91c5dda398293cb9d4fd2
6a01d4dd94"}}
]}
#
```

Рисунок 2.23. Зчитано вміст загальної проекції

Результат відображений в вигляді JSON-документу, котрий містить масив рядків *rows* з описом документів за трьома полями:

1. id – ідентифікатор документу *_id*;

2. key – ключ кортежу «ключ/значення» функції map-reduce;

3. value – значення кортежу «ключ/значення» функції map-reduce.

Загальна проекція *_all_docs* містить ідентичні значення полів *id* і *key*.

Для включення документів в поле *value*, необхідно в URL зазначити параметр *include_docs* зі значенням *true*.

curl $COUCH_ROOT_URL/music/_all_docs?include_docs=true

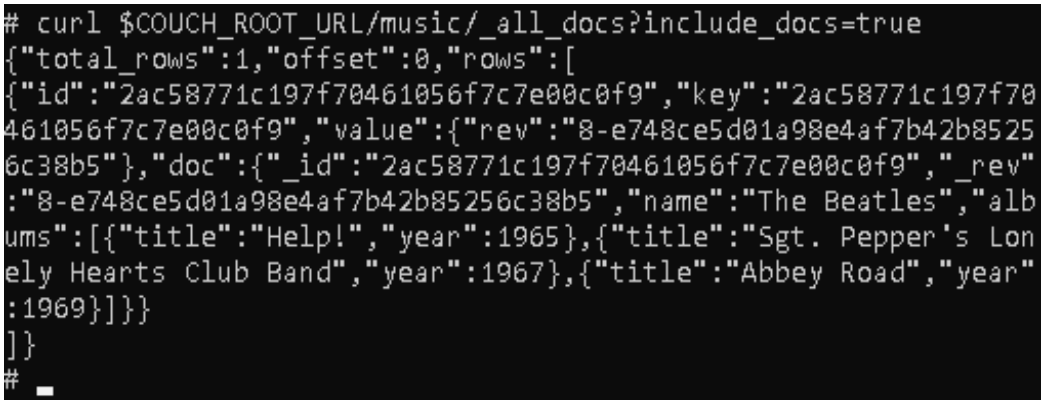


Рисунок 2.24. Зчитано проекцію з вмістом документів

Ознайомившись з загальною проекцію *_all_docs*, розберемо на прикладах створення тестових проекцій.

Функція емітеру

Для початку відтворимо проекцію *_all_docs*, далі ускладнимо завдання і будемо зчитувати значення індексованих полів документів.

Для створення проекції, виконуємо запуск Веб-інтерфейсу Fauxton за посиланням *http://localhost:5984/_utils/*. Курсором перемістимось позначкою «плюс» біля *Design Documents*, далі *New View*, відобразиться інтерфейс «*Map function*» з описом функції емітеру.

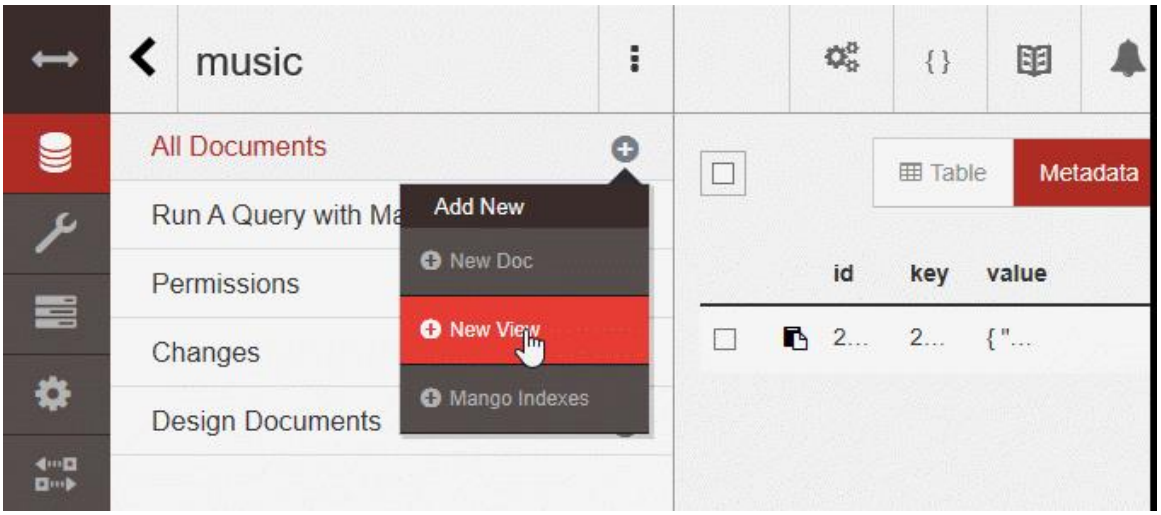


Рисунок 2.25. Відображено інтерфейс створення емітера

Параметрами функції *emit* є два аргументи – ключ і значення. Приклад, наявний в СБД, створює проекцію з кортежами, в котрих ключі дорівнюють ідентифікаторам документів, значення кортежів – число 1.

```
function(doc) {
    emit(doc._id, 1);
}
```

Спочатку зазначимо тестову функцію емітеру « *emit(null, doc)* » для створення проекції з ключем *null* і значенням – вміст документу.

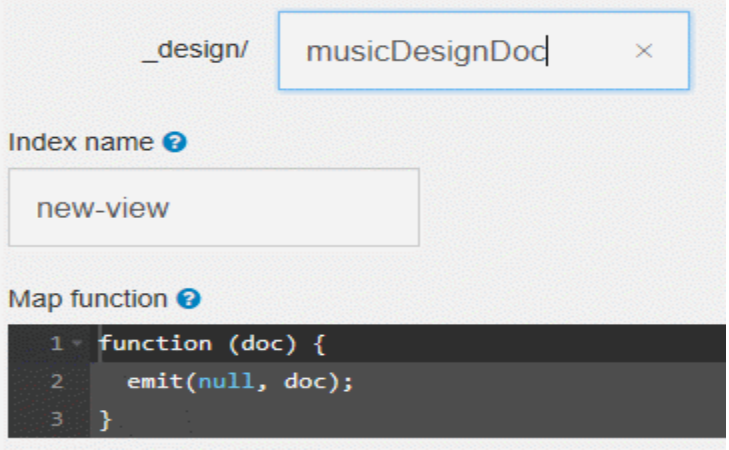


Рисунок 2.26. Зазначено вміст функції емітеру

В інтерфейсі «*Map function*» оновимо текст функції емітеру і курсором переміщуємось за посиланням *Run*.

Системою баз даних сформовано проекцію з ключем «*key*» – *null* і значенням «*value*» – вміст документу.

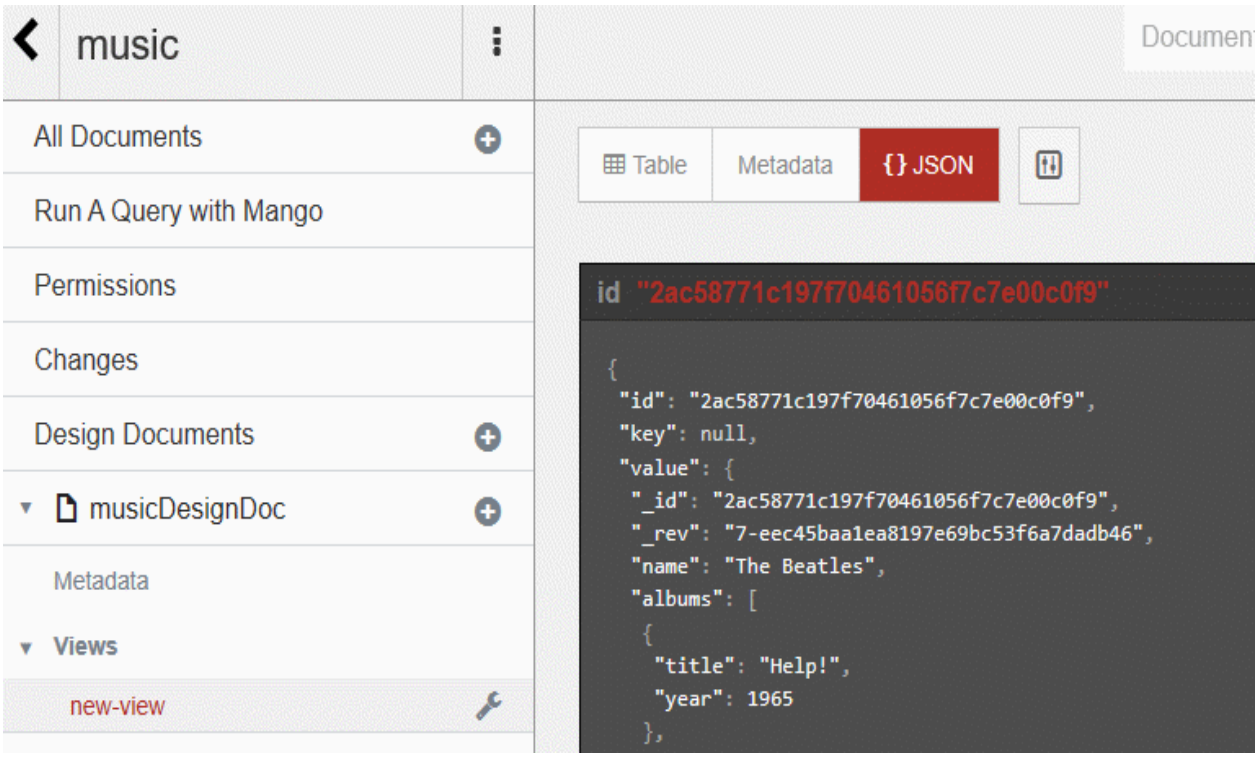


Рисунок 2.27. Зчитано вміст створеної проекції

_id	albums	language	name	views
2ac58771c197f70...	[{ "title": "Help!", "...		The Beatles	
_design/musicDes...		javascript		{ "new-view": { "m...

Рисунок 2.28. Зчитано наявні документи і проекції

Оновимо функцію емітеру для створення проекції, подібної до результату запиту за посиланням *_all_docs?include_docs=true*.

function (doc) { emit (doc._id, {rev: doc._rev}); }

Рисунок 2.29. Створено проекцію

Проектний документ

Створені проекцій зберігаються в «*проектних документах*» (*Design Documents*), котрі за структурою подібні до звичайних документів бази даних. URL-посилання проектних документів починаються з «*_design/*».

Проектний документ може складатись з декількох проекцій. Які проекції розмістити в проектному документі визначає розробник програмного

забезпечення, але рекомендовано групувати проекції за вмістом поля *value*, бо іменування ключів (наприклад, для музичної композиції «*track*» або «*title*») можуть відрізнятись в різноманітних версіях програмного забезпечення.

2.3.7. Створення проектних документів

Створення проекцій вже розібрано, тепер створимо проектні документи за вмістом документів.

Наприклад, поле *name* бази даних «*music*» містить інформацію щодо іменувань виконавців. Як знайти документ за полем *name* і значенням «*The Beatles*»? Потрібно створити відповідну проекцію.

Проекція виконавців

Зазначимо в інтерфейсі «*Map function*» функцію емітеру:

function(doc) { if ('name' in doc) { emit(doc.name, doc._id); } }

Рисунок 2.30. Зазначено умову в функції емітеру

Умова визначає, чи є поле «*name*» в поточному документі *doc*? За позитивної відповіді створюється кортеж «ключ/значення» зі значеннями полів документу – *name* і *_id*.

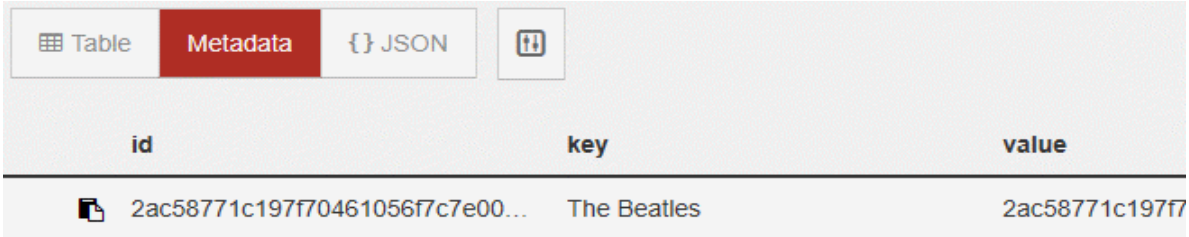


Рисунок 2.31. Створено проекцію зі значенням поля документу

Проекція альбомів

Розібравшись з пошуком музичних виконавців за полем *name*, створимо проекцію за альбомами. В «*_design*/» зазначимо назву проекції «*albums*», в «*Index name*» – значення «*by_name*», в «*Map function*» – функція емітеру:

function(doc) { if ('name' in doc && 'albums' in doc) { doc.albums.forEach(function(album) { var key=album.title || album.name, value={ by: doc.name, album: album }; emit(key, value); }); } }

Емітер перевіряє, чи є в поточному документі поля *name* і *albums*, за позитивної відповіді створює кортежі «ключ/значення» з описом альбомів, ключі – назви альбомів, значення – іменування музичних виконавців і опис альбомів (див. рис. 2.33 і рис. 2.35).

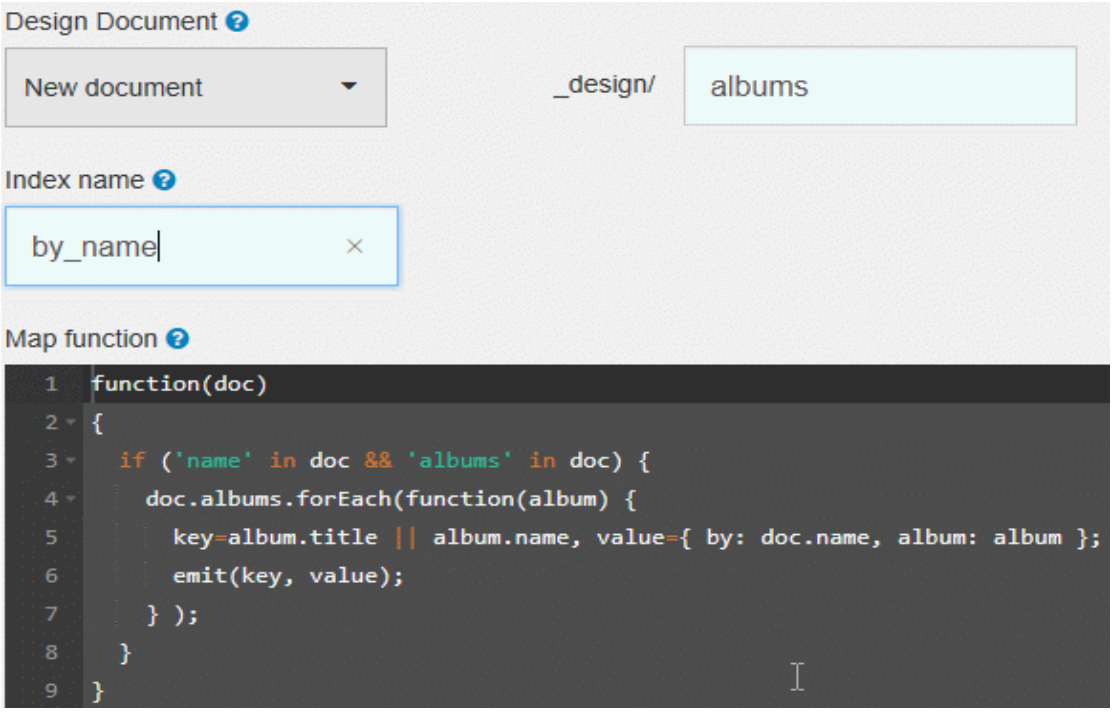


Рисунок 2.32. Зазначено умови щодо наявності полів документу

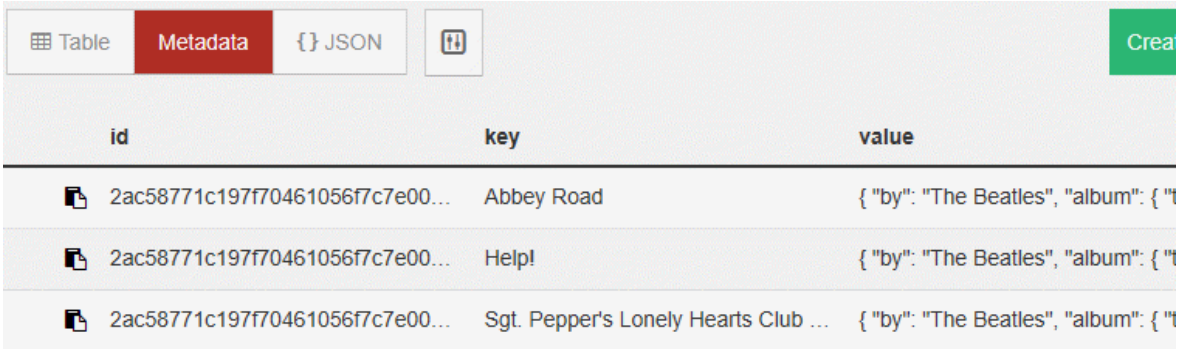


Рисунок 2.33. Виконано функцію емітеру

2.3.8. Зчитування проектних документів

Після створення проектного документу, перемістимось в консоль контейнеру і утилітою *cURL* зчитаємо вміст проекції за шаблоном: « */<database_name>/_design/<design_doc>/_view/<view_name>* ».

Зчитаємо проекцію (*<view_name>*) «*new-view*» проектного документу (*<design_doc>*) «*artistsByNameMapper*», створеного за вмістом бази даних (*<database_name>*) «*music*».

curl $COUCH_ROOT_URL/music/_design/artistsByNameMapper/_view/new-view

Рисунок 2.34. Зчитано проекцію за виконавцями

Зчитаємо проекцію щодо альбомів.

curl $COUCH_ROOT_URL/music/_design/albums/_view/by_name

Рисунок 2.35. Зчитано проекцію за альбомами

Як бачимо, при створенні проекції, CouchDB відсортовує екземпляри даних відповідно до значень ключів функції емітеру.

Вимикаємо контейнер.

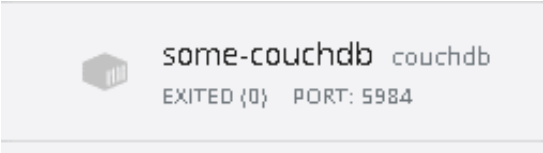


Рисунок 2.36. Вимкнено контейнер

2.4. Висновки

В інтерфейсі Fauxton і утилітою *cURL* виконано CRUD-запити до СБД. На базі прикладів розібрано поняття проекції і проектного документу.

2.4.1. Теоретичне завдання

1. Прочитайте документацію щодо HTTP Document API в CouchDB.
2. Які HTTP-запити, крім розібраних *GET*, *POST*, *PUT*, *DELETE*, реалізовані в СБД CouchDB?
3. Які типи ключів в функції емітеру підтримує CouchDB? Яку відповідь поверне при зазначенні списку в *key*?
4. Знайдіть опис підтримуваних параметрів URL (наприклад, *limit*, *startkey*), з'ясуйте їх назначення.

2.4.2. Практичне завдання

1. Утилітою *cURL* сформуйте запит *PUT* створення в базі даних «*music*» нового документу з явно вказаним значенням *_id*.
2. Утилітою *cURL* створіть і видаліть тестову базу даних.
3. Утилітою *cURL* створіть два нові документи зі структурою: *albums* (рівень № 1), *tracks* (рівень № 2), *tags* (рівень № 3).
4. Створіть проекції за полями *album*, *track*, *tag*. Сформуйте запит щодо зчитування значень рівня *tracks*.

2.4.3. Питання для закріплення знань

1. Випадки, коли рекомендовано застосовувати документні СБД.
2. Випадки, коли не рекомендовано застосовувати документні СБД.
3. Зарезервовані поля, забезпечення узгодженості вмісту бази даних СБД CouchDB.
4. REST HTTP-запити СБД CouchDB, коди відповідей серверу.
5. Поняття «проекція», назначення проектного документу.
6. Поняття «документ», пошук документів в документних СБД.
7. Формат, інтерфейси, протокол запитів до СБД CouchDB.

Розділ 3. Системи графових баз даних

Порівнюючи з СБД інших типів, обробка запиту до графової СБД потребує перегляду взаємозв'язків («ребер»), котрі з'єднують вузли («вершини») графу [19; 21; 22]. Графові бази даних застосовують для побудови соціальних сервісів, систем рекомендацій, автентифікації для доступу до Веб-ресурсів тощо.

3.1. Випадки, коли рекомендовано застосовувати

Окремі вузли/взаємозв'язки графу характеризуються типом і списком властивостей [19; 21]. Графове відображення узгоджене з об'єктно-орієнтованою моделлю програмування і реляційною моделлю баз даних реляційних СБД.

Так як графові СБД спроектовані для аналізу взаємозв'язків між вузлами, зчитування вузлів і взаємозв'язків, в тому числі рекурсивне, виконується за фіксований інтервал часу (якщо граф повнозв'язний, тоді для переміщення між вузлами A і C знадобиться крок), порівнюючи з реляційними базами даних, котрі спочатку виконують об'єднання даних для перевірки виконання умов щодо окремих екземплярів даних.

3.2. Випадки, коли не рекомендоване застосовувати

Архітектурні вади СБД Neo4j – апаратні межі розмірності графу; ускладненість розподілення графової бази даних між вузлами кластеру. Зчитування розподіленого графу уповільнюється затримками послідовного з'єднання між вузлами відповідно до наявних взаємозв'язків в графі [19].

3.3. СБД Neo4j

Neo4j концентрується на взаємозв'язках, ніж на унікальності окремих екземплярів даних, як реляційні СБД. Порівнюючи з документними базами

даних, графова СБД Neo4j забезпечує зберігання природним чином неструктурованих даних і взаємозв'язків між ними.

3.3.1. Передісторія

Neo4j – СБД з відкритим вихідним кодом, розроблена в 2007-у році. Позбавлена необхідності створення реляційної схеми бази даних і не містить умов щодо взаємозв'язків між даними.

СБД Neo4j підтримує мову запитів Cypher і REST-запити за протоколом HTTP, об'єм даних 34 млрд. вузлів і взаємозв'язків, котрих вистачає для більшості застосувань СБД [21]. Окрім простоти застосування, Neo4j характеризується значною швидкістю обробки запитів. Існує розподілена реалізація Neo4j за моделлю об'єднання вузлів комп'ютерної мережі «керуючий-службовий» – Neo4j High Availability (HA) [21].

3.3.2. Встановлення системи баз даних

Встановлюємо СБД в вигляді контейнера платформи Docker.

docker run --name some-neo4j -p 7474:7474 -p 7687:7687 -d neo4j

```
PS C:\Users\dbs> docker run --name some-neo4j -p 7474:7474 -p 7687:7687 -d neo4j
Unable to find image 'neo4j:latest' locally
latest: Pulling from library/neo4j
d121f8d1c412: Already exists
75deccc0fc24: Pull complete
690f480f5f48: Pull complete
9d192c8fdd60: Pull complete
2f13d06640lb: Pull complete
e41baf9d22a3: Pull complete
7786la68ea72: Pull complete
95d82e509265: Pull complete
Digest: sha256:1244a1496c9fcda4827db4fb9bf40278b9bc9332lflbf7c60963d76bd8472d03
Status: Downloaded newer image for neo4j:latest
fa0553aeaelb94ced2bbb12526ee1680d6fccbcc408c9d3fddd5ffb08lea27b4
PS C:\Users\dbs>
```

Рисунок 3.1. Встановлено СБД Neo4j в вигляді контейнера

3.3.3. Створення бази даних

Після встановлення, виконуємо запуск контейнеру.

Далі виконуємо запуск консолі контейнеру.

Рисунок 3.2. Виконано запуск консолі контейнеру

Виконуємо запуск серверу.

neo4j start

```
# neo4j status
Neo4j is not running
# neo4j start
Directories in use:
  home:          /var/lib/neo4j
  config:        /var/lib/neo4j/conf
  logs:          /logs
  plugins:       /var/lib/neo4j/plugins
  import:        /var/lib/neo4j/import
  data:          /var/lib/neo4j/data
  certificates: /var/lib/neo4j/certificates
  run:           /var/lib/neo4j/run
Starting Neo4j.
Started neo4j (pid 283). It is available at http://localhost:7474/
There may be a short delay until the server is ready.
See /logs/neo4j.log for current status.
#
```

Рисунок 3.3. Виконано запуск серверу СБД

Як і CouchDB, в СБД Neo4j реалізовано Веб-інтерфейс користувача. Виконуємо запуск графічного інтерфейсу користувача.

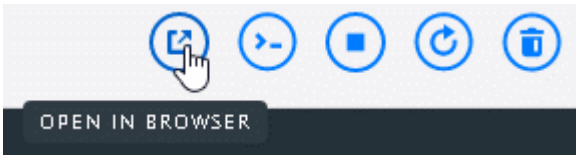


Рисунок 3.4. Виконано запуск Веб-інтерфейсу

Веб-інтерфейс

В браузері перемістившись за посиланням *http://localhost:7474/browser/*, відобразиться сторінка Веб-інтерфейсу СБД (див. рис. 3.5). В підрозділі «Connect to Neo4j» іменування користувача і пароль – « *neo4j* ».

Інтерфейс містить консоль біля позначки « $ » в лівій верхній частині. Для підключення до бази даних в консолі зазначається « *:server connect* », для довідки щодо операторів Cypher – « *:help* ».

3.3.4. Мова запитів Cypher

Існує декілька способів взаємодії з Neo4j. Крім клієнтських бібліотек різноманітних мов програмування, формувати запити до СБД можна за інтерфейсом REST і двома мовами – Gremlin і Cypher. В Gremlin реалізовані цікаві функції, але на сьогодні Cypher вважається стандартом.

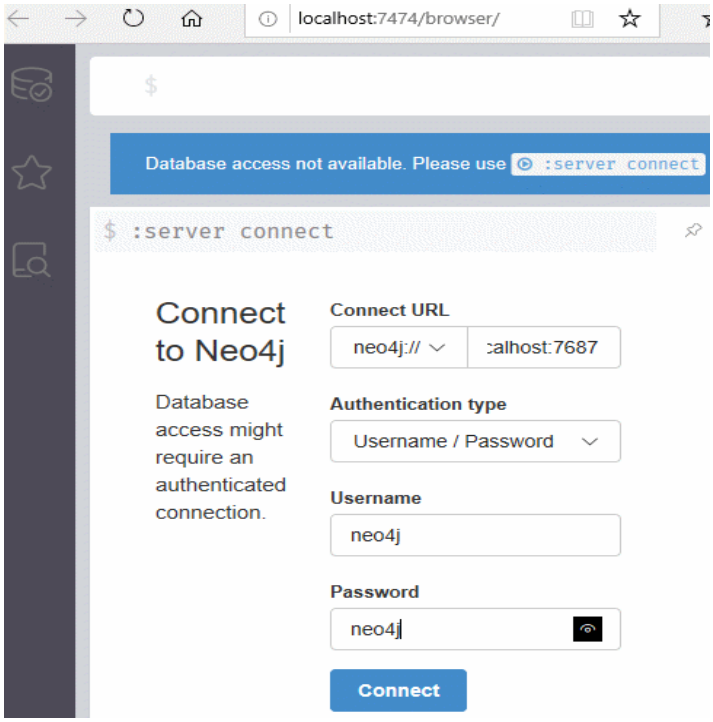


Рисунок 3.5. Відображено інтерфейс автентифікації

Результатом встановлення з'єднання є поява інформації щодо користувача і протоколу запитів до сервера СБД.

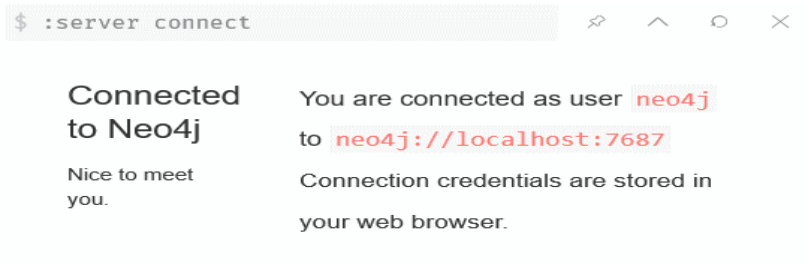


Рисунок 3.6. Встановлено з'єднання з СБД

Cypher – мова запитів до графової СБД. Порівнюючи з теорією графів, графи іменуються «вузлами» («вершинами»), з'єднання між вузлами – «взаємозв'язками» («ребрами»). Запити до графу формуються за шаблоном:

$ MATCH [some set of nodes and/or relationships]
WHERE [some set of properties holds]
RETURN [some set of results captured by the MATCH and WHERE clauses]

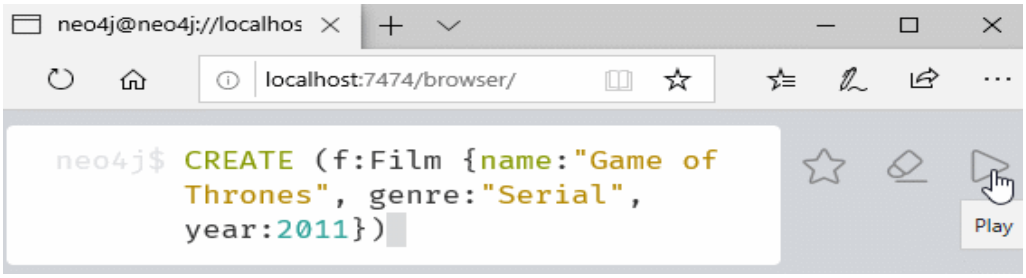


Рисунок 3.7. Сформовано запит мовою Cypher

3.3.5. CRUD-запити до бази даних

В мові Cypher, оператори *MATCH* і *CREATE* створюють нові вузли/взаємозв'язки; оператор *SET* оновлює значення властивостей наявних вузлів/взаємозв'язків графу.

Створимо вузол з типом «кінофільм» і властивостями: «назва» – «*Game of Thrones*»; «жанр» – «*Serial*»; «рік» – «2011».

CREATE (f:Film {name:"Game of Thrones", genre:"Serial", year:2011})

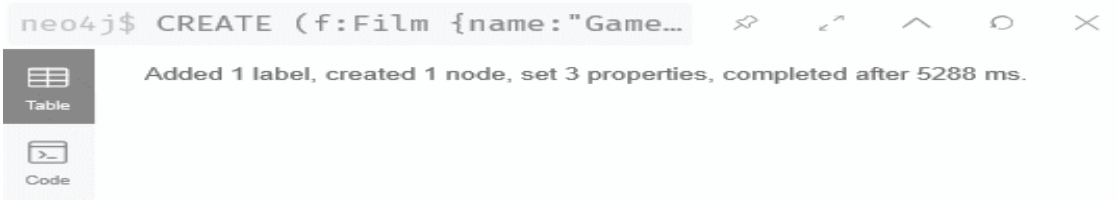


Рисунок 3.8. Створено вузол з новим типом

В інтерфейсі виведено повідомлення СБД, в верхній частині розміщується запит Cypher, вкладка *Table* відображає вузли та/або взаємозв'язки, створені запитом Cypher (див. рис. 3.9), вкладка *Code* відображає системну інформацію щодо запиту (див. рис. 3.10).

Оператор *MATCH* зчитує вузли графу за умовою запиту.

MATCH (n) RETURN n ;

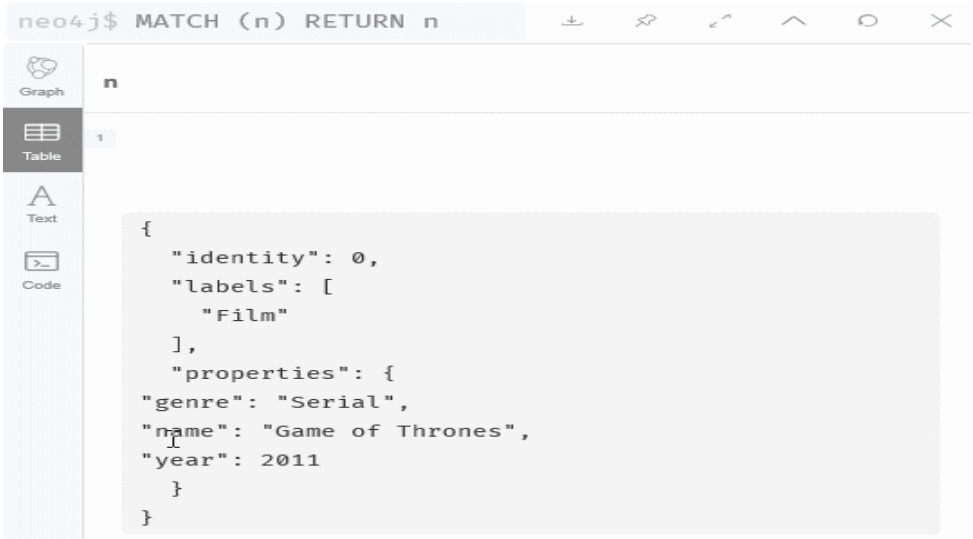


Рисунок 3.9. Зчитано всі вузли графу

Створюємо додатковий вузол з типом «новини».

CREATE (p:Publication {name:"Film Expert Monthly"});

Рисунок 3.10. Створено новий вузол

Виконаний запит містить іменування *Film* і *Publication* – типи вузлів; скорочення *f* і *p* – назви змінних. Список властивостей в межах певного типу може варіюватись, подібно до полів в документних СБД.

Створимо зв'язок з типом «*reported_on*» для вузлів з типами «*Publication*» і «*Film*».

MATCH (p:Publication {name:"Film Expert Monthly"}),
(f:Film {name:"Game of Thrones", year:2011})
CREATE (p)-[r:reported_on]->(f);

Рисунок 3.11. Створено новий зв'язок

Оператор *CREATE* створює зв'язок з типом «*reported_on*» для вузлів за умови відповідності *MATCH* значень властивості «*name*».

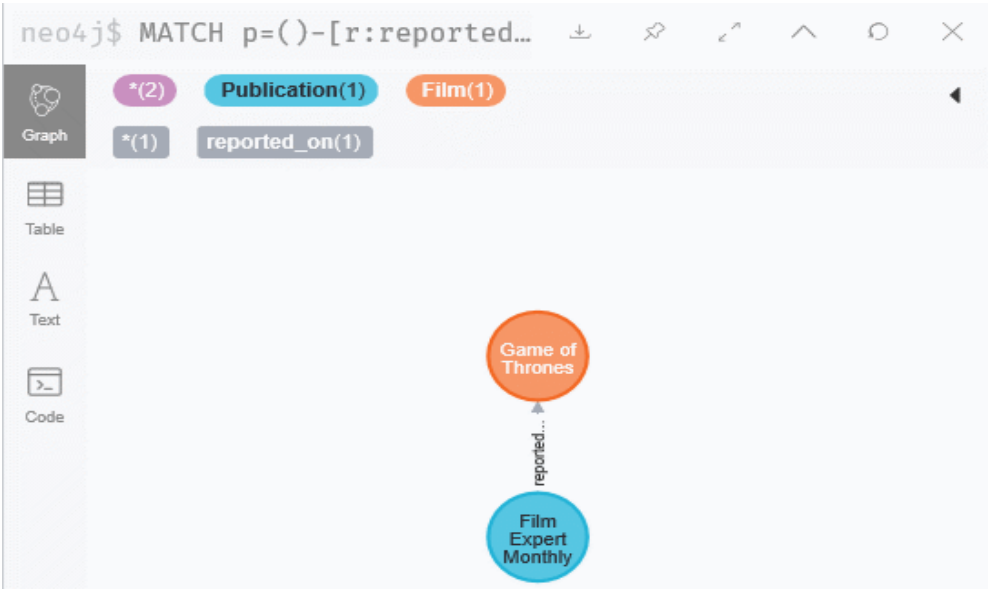


Рисунок 3.12. Створено зв'язок між вузлами

Взаємозв'язки, як і вузли, можуть містити властивості. Доповнимо зв'язок № 0 властивістю «рейтинг».

MATCH ()-[r]-() WHERE id(r) = 0 SET r.rating = 97 RETURN r ;

```
{
  "identity": 0,
  "start": 1,
  "end": 0,
  "type": "reported_on",
  "properties": {
"rating": 97
  }
}
```

Рисунок 3.13. Доповнено зв'язок властивістю

Зазначення властивостей допускається і в запиті створення взаємозв'язку.

MATCH (p:Publication {name:"Film Expert Monthly"}),
(f:Film {name:"Game of Thrones"})
CREATE (p)-[r:reported_on {rating:97}]->(f) ;

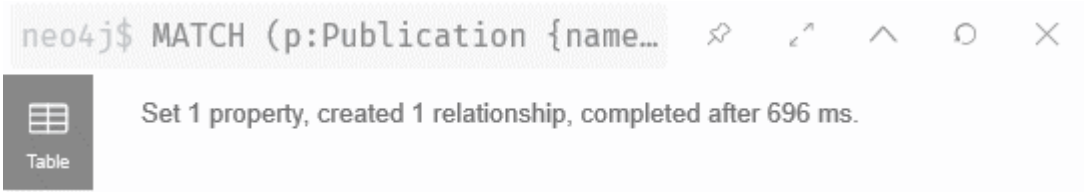


Рисунок 3.14. Доповнено зв'язок властивістю

Якщо відобразити весь граф запитом « *MATCH (n) RETURN n*; » і курсором переміститись за взаємозв'язком між вузлами, відобразиться значення 97, котре є властивістю взаємозв'язку «*reported_on*». Так як кінофільми кінокомпанії «*HBO*» створені в різноманітних жанрах, створимо новий вузол «тип жанру» («*genre_type*») зі значенням «*Serial*» властивості «*name*».

CREATE (g:GenreType {name:"Serial"}) ;

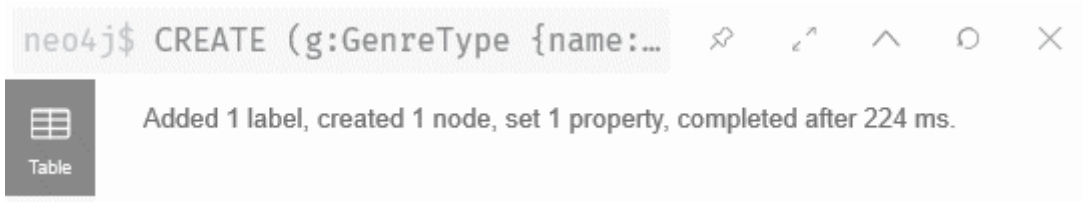


Рисунок 3.15. Створено тип вузла з властивостями

Створюємо зв'язок між вузлами з типами «жанр» і «кінофільм».

```
MATCH (f:Film {name:"Game of Thrones"}),
    (g:GenreType {name:"Serial"})
    CREATE (f)-[r:genre_type]->(g) ;
```

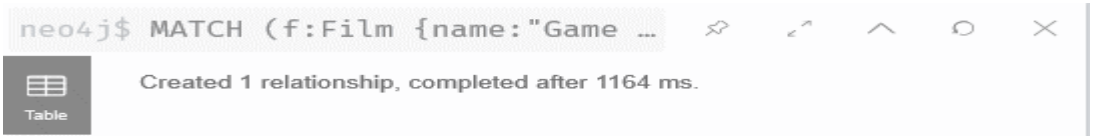


Рисунок 3.16. Створено зв'язок без зазначення властивостей

Наявий граф містить вузли трьох типів: кінофільм, жанр і новини. Створимо тестові вузол і зв'язок, для подальшого видалення з БД.

Створюємо вузол.

```
CREATE (e:EphemeralNode {name:"short lived"}) ;
```

Рисунок 3.17. Створено тестовий вузол

Створюємо зв'язок між тестовим вузлом і вузлом «кінофільм».

```
MATCH (f:Film {name:"Game of Thrones"}),
(e:EphemeralNode {name:"short lived"})
CREATE (f)-[r:short_lived_relationship]->(e) ;
```

Рисунок 3.18. Створено тестовий зв'язок

Переглядаємо наявні в графі вузли і взаємозв'язки.

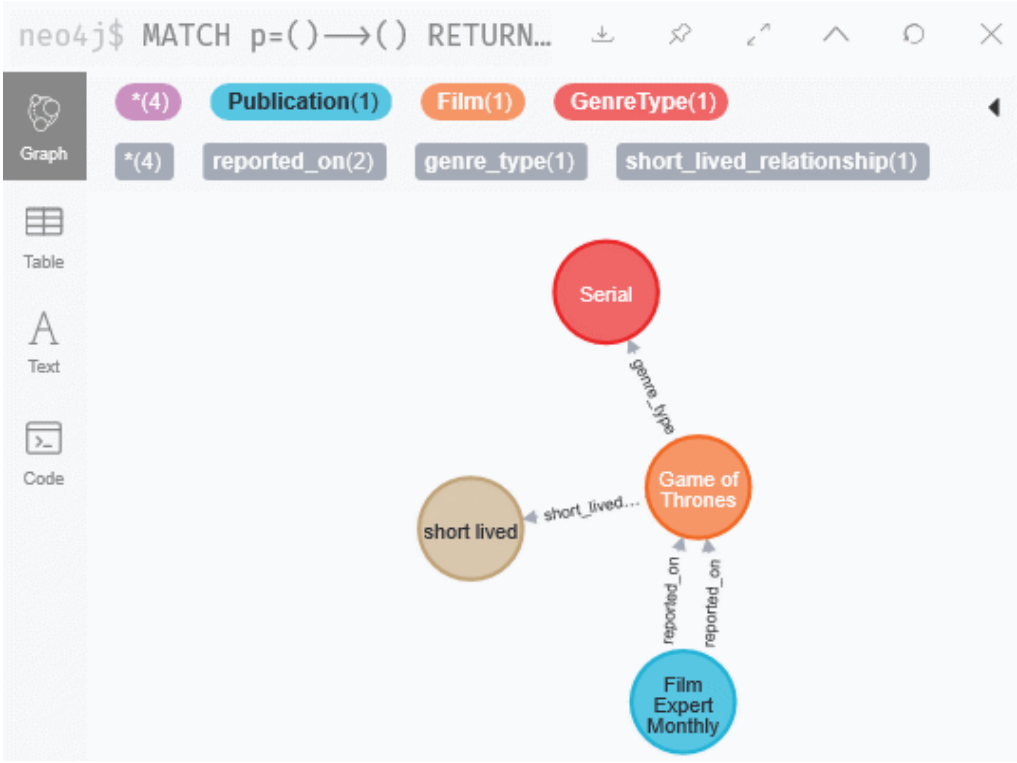


Рисунок 3.19. Відображено граф до видалення тестового взаємозв'язку

Видаляємо тестовий зв'язок з вказаним типом.

MATCH ()-[r:short_lived_relationship]-() DELETE r ;

Переглядаємо структуру графу після видалення тестового взаємозв'язку.

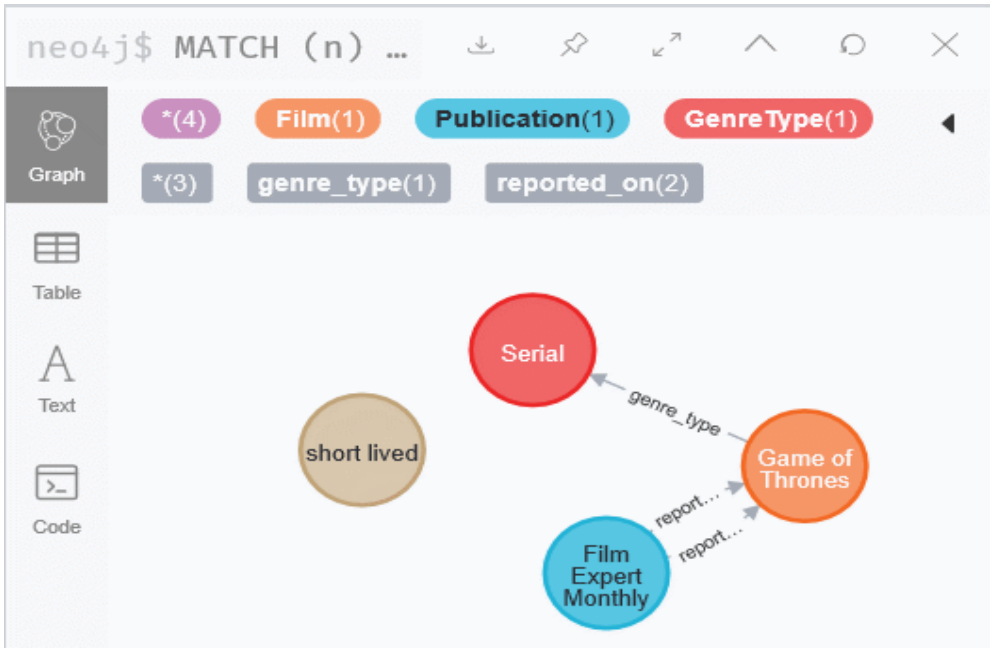


Рисунок 3.20. Відображено граф після видалення тестового взаємозв'язку

Видаляємо тестовий вузол з вказаним типом.

MATCH (e:EphemeralNode) DELETE e ;

Граф кінофільмів повернувся до стану перед створенням вузла «*short lived*». Існує також шаблон запиту Cypher для видалення всіх вузлів і взаємозв'язків, тобто всього графу. Залишимо запит без виконання!

// MATCH (n) OPTIONAL MATCH (n)-[r]-() DELETE n, r ;

Продовжуємо будувати граф кінофільмів. Кінофільми продукують різноманітні кінокомпанії. Створимо вузол з типом «кінокомпанія» і дозапишемо в граф вузол з іменуванням «*HBO*».

CREATE (fc:Company {name:"HBO"})

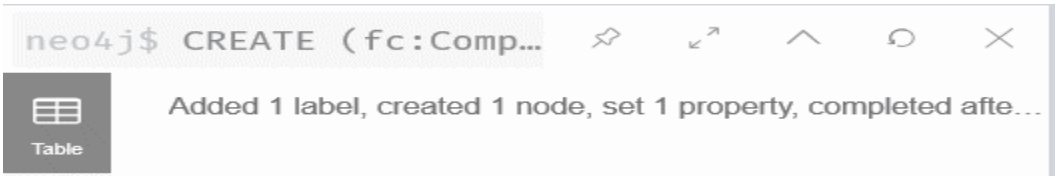


Рисунок 3.21. Доповнено вузол новим типом

Так як кінокомпанії створюють кінофільми різноманітних жанрів, додатково створимо вузли з типом «жанр» і значеннями *Fiction* і *Drama*.

CREATE (f:Film {name:"Game of Thrones", genre:"Fantasy", year:2012})

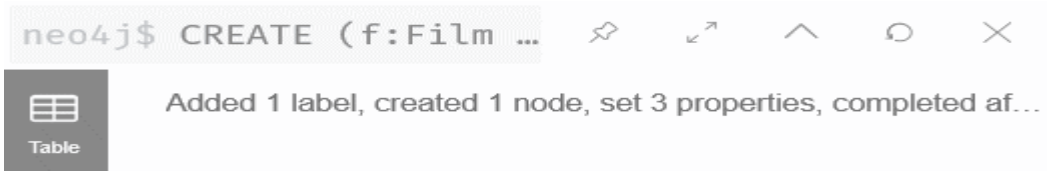


Рисунок 3.22. Створено вузол з наявним типом в графі

CREATE (f:Film {name:"Game of Thrones", genre:"Drama", year:2013})

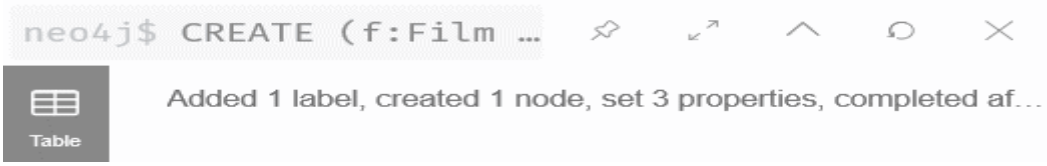


Рисунок 3.23. Створено вузол з наявним типом в графі

Створюємо зв'язок «вироблено» (*produced*) між вузлами «*Game of Thrones*» і «*HBO*».

MATCH (fc:Company {name:"HBO"}),
(f:Film {name:"Game of Thrones"})

CREATE (fc)-[r:produced]->(f)

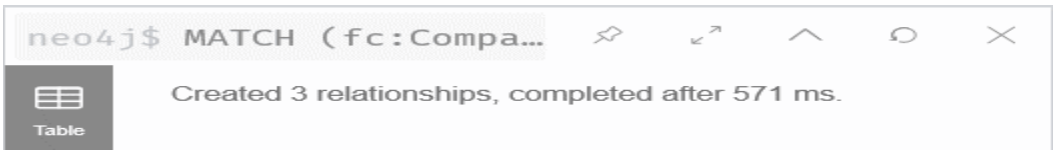


Рисунок 3.24. Створено три взаємозв'язки між чотирма вузлами

Створюємо зв'язок з типом «*genre_type*» між вузлами з типами «*Serial*» і «*Film*».

MATCH (f:Film),
(g:GenreType {name:"Serial"})
CREATE (f)-[r:genre_type]->(g)

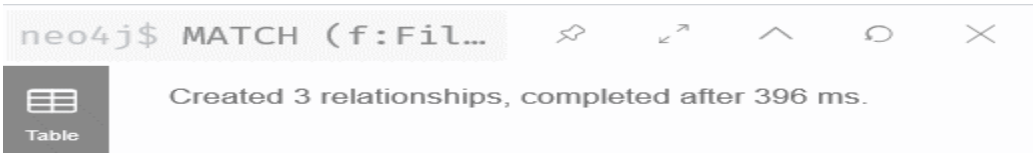


Рисунок 3.25. Створено взаємозв'язки між вузлами

Переглядаємо структуру графу (див. рис. 3.26).

Приклад соціальної мережі

Крім ознайомлення користувача з кінофільмами і новинами, може знадобитись соціальна складова для формування рекомендацій. Тестова соціальна мережа має містити дані щодо перегляду кінофільмів і взаємозв'язки між користувачами. Створимо тип «користувач». Для прикладу створимо трьох користувачів, двох знайомих і певного незнайомого, всі з уподобаннями щодо кінофільмів і жанрів.

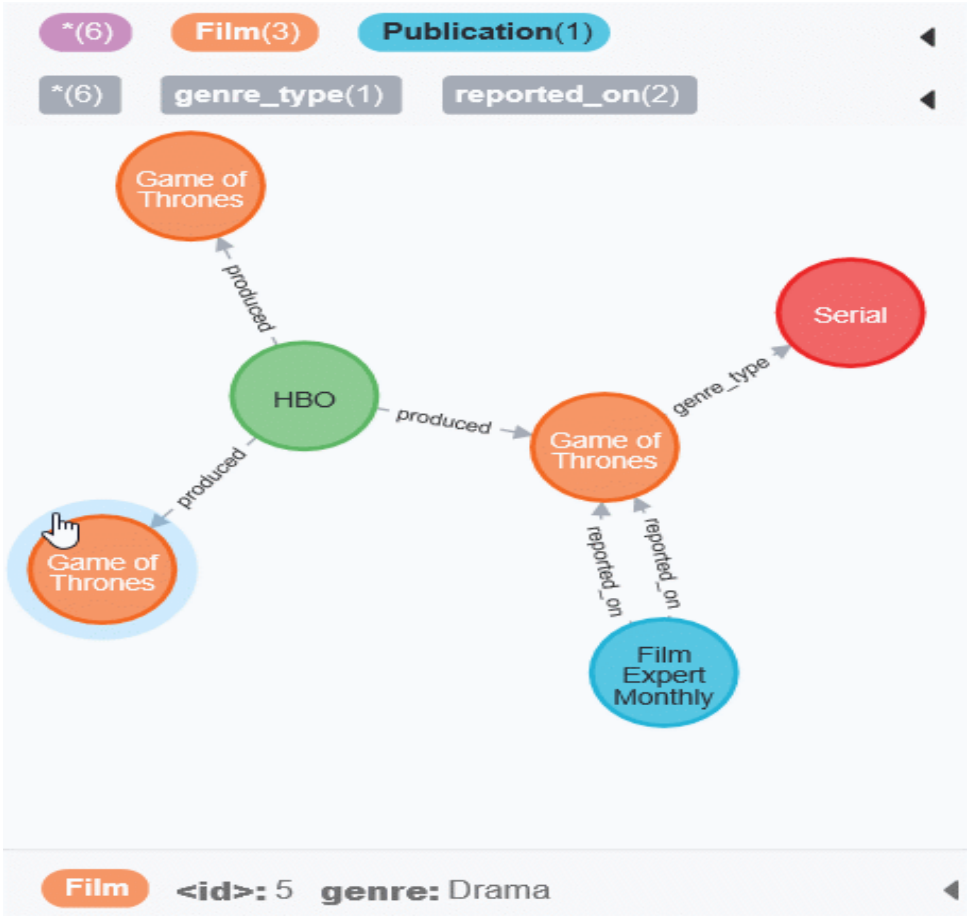


Рисунок 3.26. Наявні взаємозв'язки в графі

Створюємо вузол з типом «користувач» і властивістю «*name*».

CREATE (p:Person {name:"Jane"})

Рисунок 3.27. Створено вузол соціальної мережі

Створюємо зв'язок з типом «*likes*» між вузлами з типами «*Game of Thrones*» і «*Jane*».

MATCH (p:Person {name:"Jane"}),
(f:Film {name:"Game of Thrones", genre:"Drama"})
CREATE (p)-[r:likes]->(f)

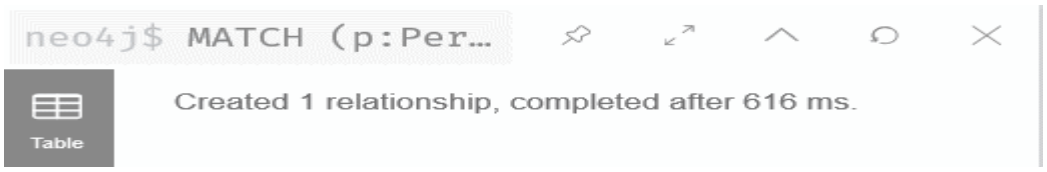


Рисунок 3.28. Створено зв'язок між вузлами соціальної мережі

Створюємо вузол користувача *Tom*, котрий також дивиться кінофільм, але вважає «жанр» – *Drama*.

CREATE (p:Person {name:"Tom"})

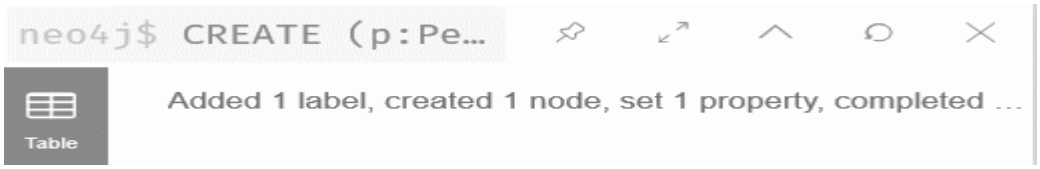


Рисунок 3.29. Створено вузол з наявним типом

MATCH (p:Person {name:"Tom"}),
(f:Film {name:"Game of Thrones", genre:"Drama"})
CREATE (p)-[r:likes]->(f)

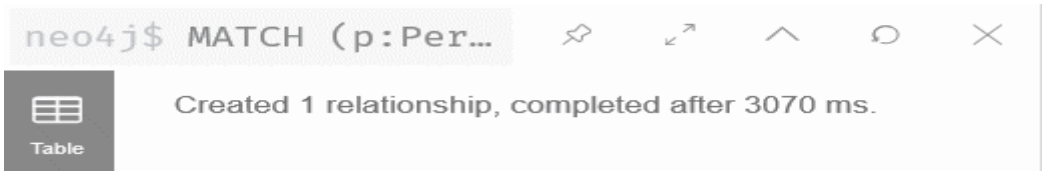


Рисунок 3.30. Створено зв'язок між вузлами соціальної мережі

Створюємо зв'язок між вузлами «користувач» і «новини».

MATCH (p:Person {name:"Tom"}),

(pub:Publication {name:"Film Expert Monthly"})

CREATE (p)-[r:trusts]->(pub)

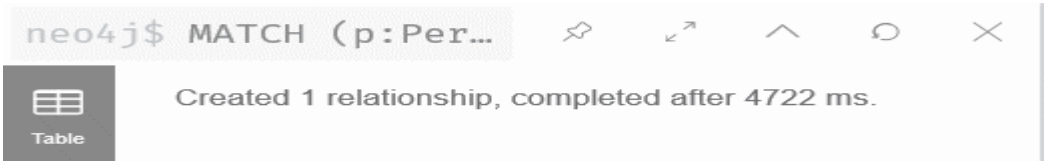


Рисунок 3.31. Доповнено соціальну мережу вузлом і взаємозв'язком

Переглядаємо наявні в графі вузли й взаємозв'язки.

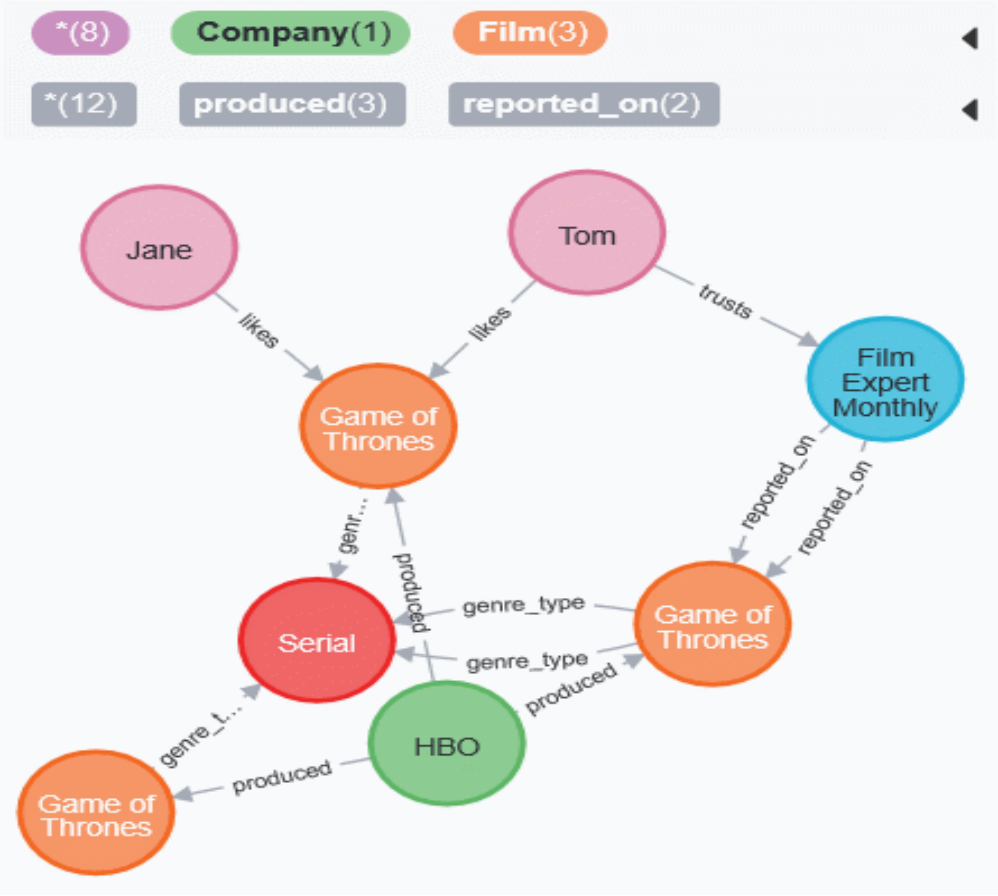


Рисунок 3.32. Відображено структуру графу соціальної мережі

Створимо вузол користувача *Patty*, котрий дружить з *Tom* і *Jane*, але поки не переглянув жодного кінофільму.

```
CREATE (p:Person {name:"Patty"})
```

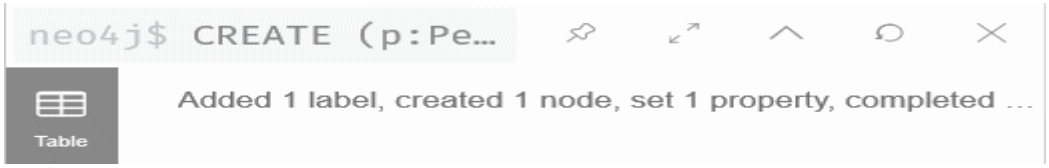


Рисунок 3.33. Доповнено граф соціальної мережі вузлом

```
MATCH (p1:Person {name:"Patty"}),
(p2:Person {name:"Tom"})
CREATE (p1)-[r:friends]->(p2);
```

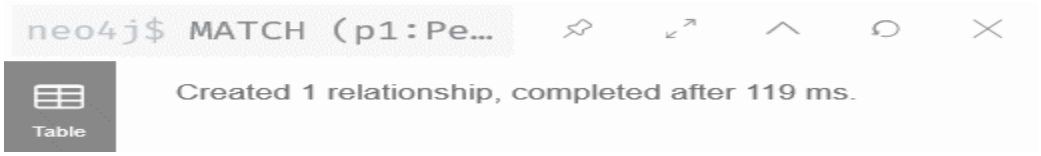


Рисунок 3.34. Створено зв'язок між вузлами

```
MATCH (p1:Person {name:"Patty"}),
(p2:Person {name:"Jane"})
CREATE (p1)-[r:friends]->(p2);
```

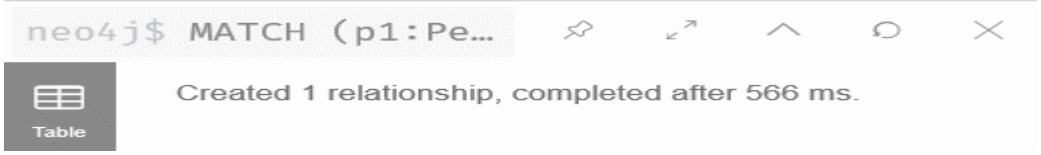


Рисунок 3.35. Створено зв'язок між вузлами

Пошук взаємозв'язків вузла

Для пошуку всіх взаємозв'язків певного вузла існує оператор « --> ». Зчитаємо всі вузли, взаємозв'язані з Jane.

MATCH (p:Person {name:"Jane"})-->(n) RETURN n;

Рисунок 3.36. Відображено всі взаємозв'язані вузли

Тепер зчитаємо вузли *Person*, за виключенням імені *Patty*. В Cypher, подібно до SQL, послідовність «<>» позначає умову «*не дорівнює*».

MATCH (p:Person) WHERE p.name <> 'Patty' RETURN p;

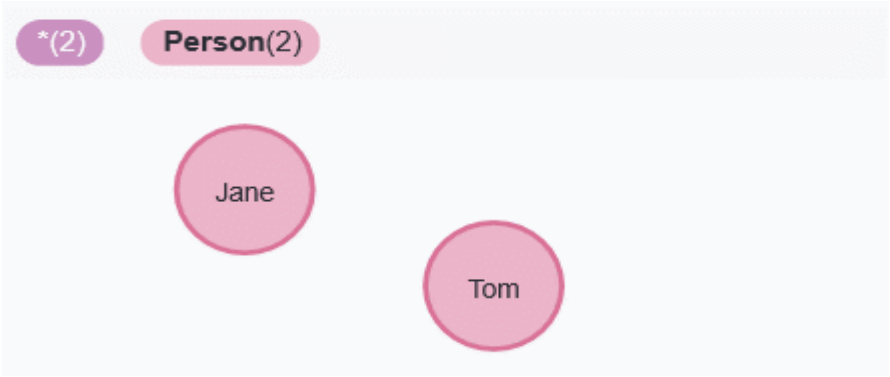


Рисунок 3.37. Зчитано вузли з умовою виключення

Дозапишемо деякі вузли, котрі незв'язані безпосередньо з Patty – Jane з Janine, Tom з Kate.

CREATE (p1:Person {name:"Janine"}), (p2:Person {name:"Kate"});

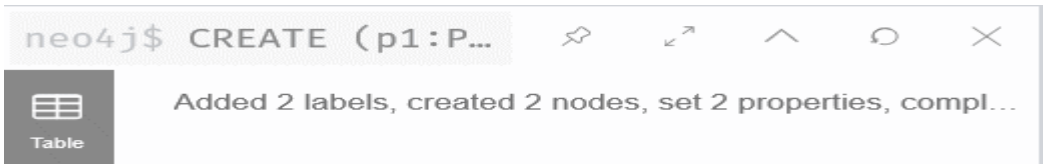


Рисунок 3.38. Створено незв'язані вузли з єдиним типом

MATCH (p1:Person {name:"Janine"}),
(p2:Person {name:"Jane"})
CREATE (p1)-[r:friends]->(p2);

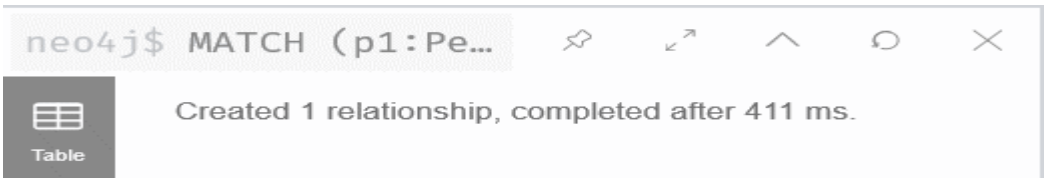


Рисунок 3.39. Створено зв'язок між вузлами з єдиним типом

MATCH (p1:Person {name:"Kate"}),
(p2:Person {name:"Tom"})
CREATE (p1)-[r:friends]->(p2);

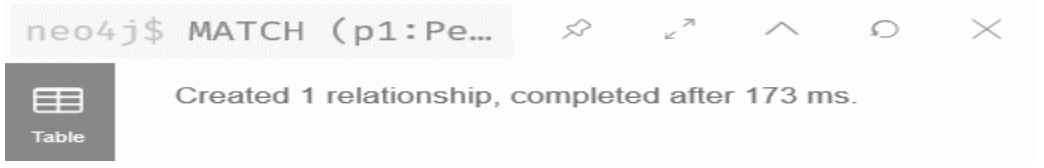


Рисунок 3.40. Створено зв'язок між вузлами з єдиним типом

Наявні вузли й взаємозв'язки в графі.

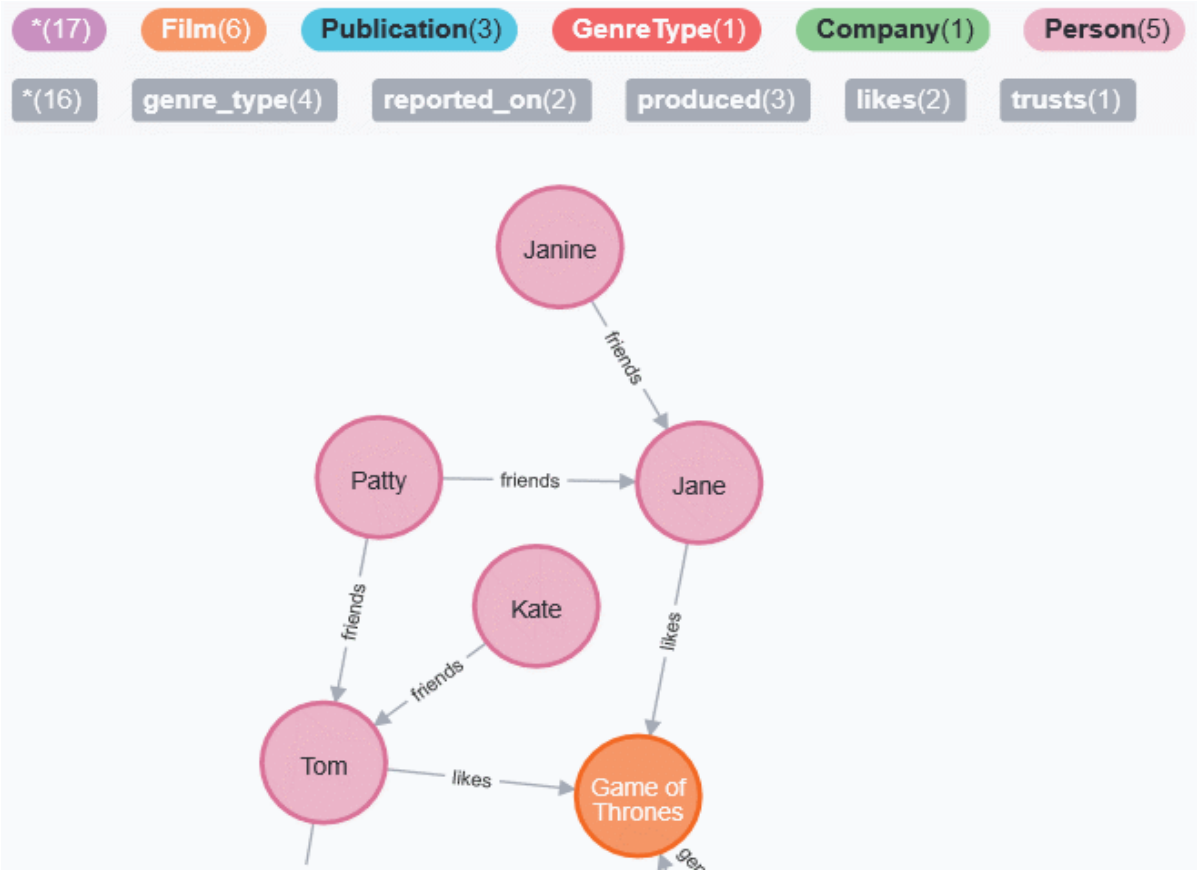


Рисунок 3.41. Зчитано взаємозв'язки в оновленому графі

Знайдемо вузли, взаємозв'язані з вузлом графу.

MATCH (fof:Person)-[:friends]-(p:Person {name:"Jane"})
RETURN fof.name;

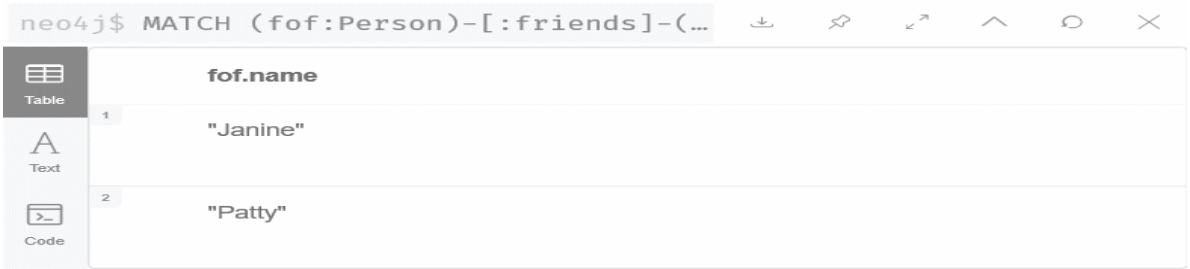


Рисунок 3.42. Зчитано взаємозв'язані вузли

Система відображає два значення – Janine і Patty.

3.3.6. Пошук вкладених взаємозв'язків

Вкладені взаємозв'язки між двома вузлами можна знайти за запитом, котрий містить типи вузлів початку і завершення пошуку, властивості вузлів, іменування алгоритму пошуку.

Зчитаємо вкладені взаємозв'язки між вузлами з типом «*Person*», застосувавши алгоритм пошуку «*shortestPath*».

MATCH (start:Person {name:"Janine"}), (end:Person {name:"Patty"}), p = shortestPath((start)-[:friends]-(end)) RETURN p;*

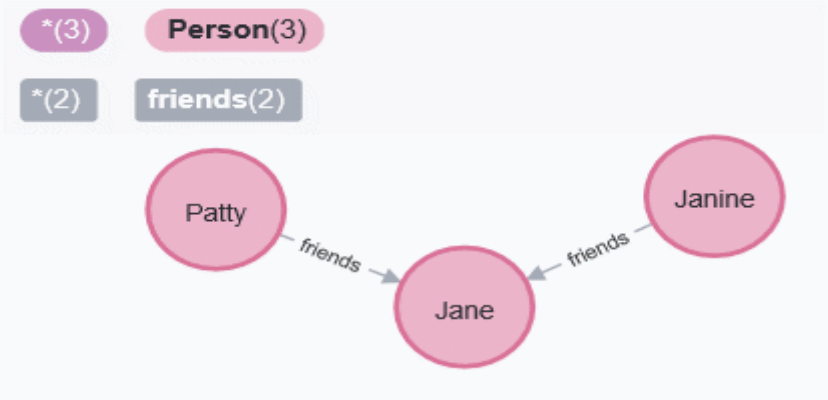


Рисунок 3.43. Зчитано вкладені взаємозв'язки між вузлами

Додаткові алгоритми пошуку взаємозв'язків – *allPaths*, *dijkstra тощо*.

3.3.7. Агрегатні функції

Агрегатні функції зміщують реалізацію обробки даних з програмного забезпечення до системи баз даних, що іноді є необхідним рішенням. Мова Cypher містить агрегатні функції: обчислення кількості вузлів/взаємозв'язків в графі – count(), виконання фільтрації – filter() тощо.

Список функцій міститься в документації Cypher за посиланням *neo4j.com/docs/developer-manual/current/cypher/functions/*.

Зчитаємо кількість наявних в графі взаємозв'язків з типом «*friends*».

MATCH (fof:Person)-[:friends]-(p:Person {name:"Jane"})
RETURN count(fof.name);

Рисунок 3.44. Зчитано значення агрегатної функції

Допускається створення агрегатних функцій й підключення їх в Cypher.

3.3.8. Створення індексів

Як і в випадку інших моделей баз даних, завдяки створенню індексів забезпечується пришвидшення виконання запитів щодо типів і властивостей екземплярів даних БД.

Створюємо індекс за типом вузла і властивістю.

CREATE INDEX ON:Film (name) ;

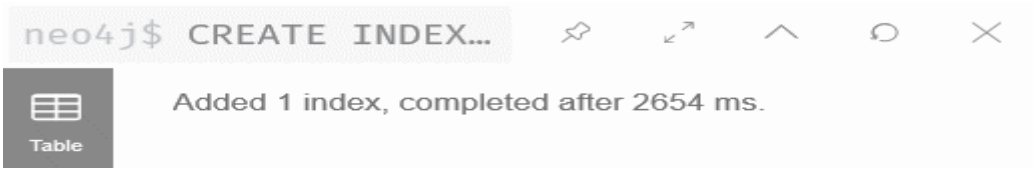


Рисунок 3.45. Створено індекс за типом вузла і властивістю

Допускається видалення індексів – « *DROP INDEX ON:Film (name) ;* ».

Застосування індексів прискорює обробку запитів, але умови унікальності зменшують об'єм БД. Створимо «умову унікальності» для властивості *name* вузла «*Site*».

CREATE CONSTRAINT ON (s:Site) ASSERT s.name IS UNIQUE;

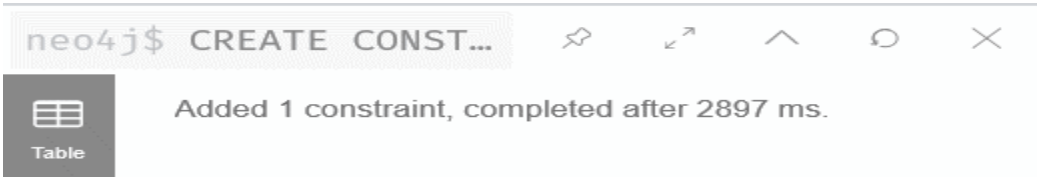


Рисунок 3.46. Створено умову унікальності значення властивості

Якщо сформувати запит створення вузла *Site* з ідентичною назвою, СБД поверне помилку.

CREATE (s:Site {name:"New Films", url:"www.newfilms.info"});

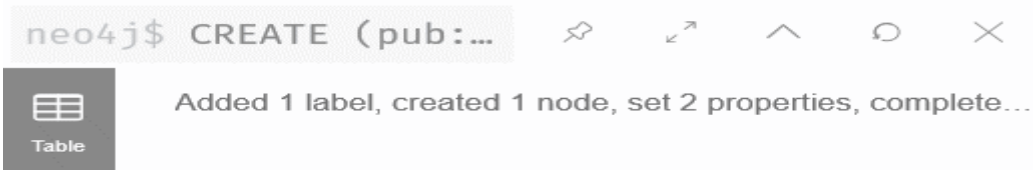


Рисунок 3.47. Створено вузол з властивостями

CREATE (s:Site {name:"New Films", url:"www.bestfilms.info"});

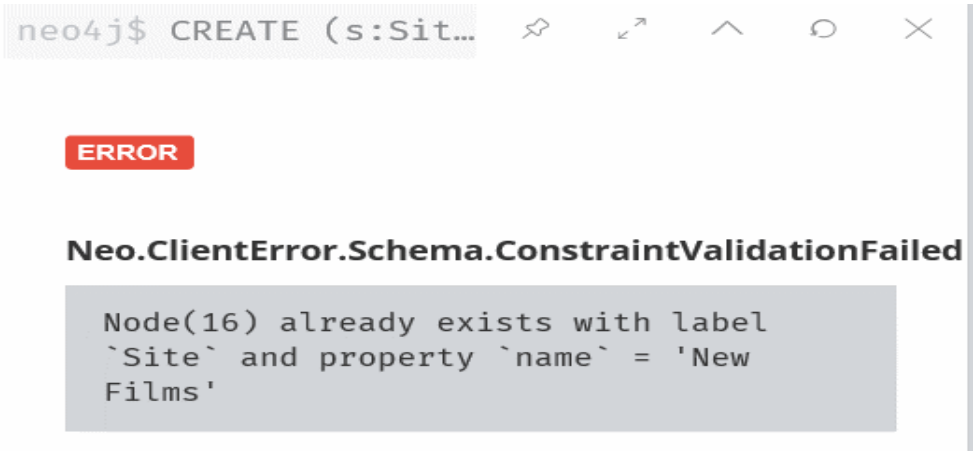


Рисунок 3.48. Відображено помилку повторного створення ідентичного вузла

При зазначенні умови щодо властивості, СБД створює індекс.

Оператор *DROP* видаляє умову унікальності і створений індекс.

DROP CONSTRAINT ON (s:Site) ASSERT s.name IS UNIQUE;

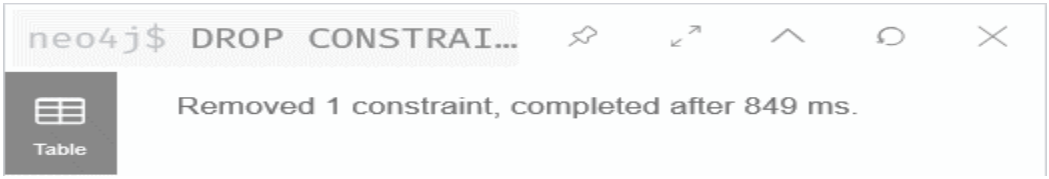


Рисунок 3.49. Видалено умову щодо значень властивості за типом в графі

Вимикаємо контейнер.

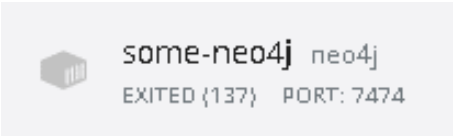


Рисунок 3.50. Вимкнено контейнер

3.4. Висновки

Розібрано переваги і вади СБД Neo4j, оператори мови Cypher, застосовуючи котрі, зокрема, можна відобразити ER-модель реляційної БД. Побудовано граф за тестовими даними.

На прикладі графу соціальної мережі виконано пошук вкладених взаємозв'язків між вузлами. За умовою знайдено вузли, до котрих застосовано агрегатну функцію. Перевірено врахування умов щодо значень властивостей окремих екземплярів даних графу.

3.4.1. Теоретичне завдання

1. За посиланням *neo4j.com/docs* прочитайте документацію щодо синтаксису мови Cypher. Знайдіть всі функції Cypher.

2. В консолі браузера за посиланням *http://localhost:7474/browser*, виконайте запит :play з назвою тестової бази даних.

3. Прочитайте публікацію за посиланням *neo4j.com/blog/graph-search-algorithm-basics*.

4. Прочитайте документацію за посиланням *neo4j.com/docs/driver-manual*.

3.4.2. Практичне завдання

1. Встановіть драйвер мови програмування за вашим вибором і програмно створіть граф – сім вузлів з типом «актор», котрі взаємозв'язані з п'ятьма вузлами з типом «кінофільм». Зчитайте всі вкладені взаємозв'язки за всіма вузлами «актор»-«актор», і, окремо, «кінофільм»-«кінофільм».

2. На базі драйвера Neo4j реалізуйте: функція № 1 обчислює відстань за вкладеними взаємозв'язками з п. 1; функція № 2 формує списки («актор»-«актор», «кінофільм»-«кінофільм») за зменшенням відстані.

3.4.3. Питання для закріплення знань

1. Випадки, коли рекомендовано застосовувати графові СБД.
2. Випадки, коли не рекомендовано застосовувати графові СБД.
3. Синтаксис мови Cypher. Які оператори застосовують в запитах?
4. Сутність агрегатних функцій СБД Neo4j, приклади запиту з параметрами.
5. Які основні структурні елементи графових баз даних, поняття теорії графів? Назначення властивостей в описі графу.
6. Назначення типів вузлів, умов унікальності в СБД Neo4j.
7. Способи взаємодії з СБД, створення індексів в Neo4j.

Розділ 4. Системи баз даних «ключ/значення»

Система баз даних «ключ/значення» (СБД «ключ/значення») співставляє ключам keys – значення values, за прикладом словника map в мовах програмування [19; 21]. СБД «ключ/значення» підтримують включення в кортежі різноманітних структур даних, наприклад, «хешів» або «списків».

Існує значна кількість реалізацій СБД «ключ/значення» з відкритим вихідним кодом. З розповсюджених відзначимо Memcached (подібні Memcachedb, Membase), Redis і Riak [19].

4.1. Випадки, коли рекомендовано застосовувати

Основною характеристикою СБД «ключ/значення» є горизонтальна масштабованість й значна швидкодія. СБД «ключ/значення» призначені для завдань, де між даними немає взаємозв'язків, немає необхідності підтримання індексів, наприклад, подібним критеріям відповідають значення ідентифікатора і час автентифікації, права доступу до Веб-сервісу, повторювані запити до інформаційної системи тощо.

4.2. Випадки, коли не рекомендовано застосовувати

Значення в кортежах мають єдиний строковий тип, порівняно з документними СБД, в котрих допускається комбінування списків і, наприклад, строкових/чисельних значень. Оперативна пам'ять характеризується тимчасовістю зберігання даних; якщо зупинити СБД не виконавши дублювання БД в «зовнішню» пам'ять (зокрема, HDD- чи SSD-накопичувач), тоді дані будуть втрачені. Виконання сервера однопотокове.

4.3. СБД Redis

Очевидна перевага Redis – швидкодія, порівнюючи з подібними системами баз даних «ключ/значення». Redis забезпечує збалансованість

надійності і швидкості зберігання, завдяки встановленню тривалості існування даних. Розподілене дублювання в зовнішню пам'ять за моделлю об'єднання вузлів комп'ютерної мережі «керуючий-службовий» (англ. master-slave) – спосіб забезпечення надійності зберігання даних, без сповільнення обробки запитів, зокрема зчитування БД [13; 14; 19; 21].

Швидкодія СБД Redis пояснюється, перш за все, розміщенням всіх даних у «внутрішній» щодо материнської плати пам'яті, зокрема оперативній, тому база даних «ключ/значення» функціонує швидше, порівнюючи з іншими типами СБД, так як не звертається до «зовнішньої» пам'яті.

4.3.1. Передісторія

REDIS – абревіатура від REmote DIctionary Server, перекладається як «дистанційний сервер словників», розроблена в 2009-у році. СБД дуже швидко зчитує дані, розміщені в оперативній пам'яті, але й дозаписування теж виконує порівняно швидко. Розробник Salvatore Sanfilippo зауважує щодо реалізованого зберігання різноманітних структур даних.

Швидке зчитування даних за хеш-таблицями є перевагою СБД «ключ/значення». Звісно перевага для конкретних завдань може виявитись вадою, якщо, наприклад, екземпляри даних БД містять взаємозв'язані значення. В СБД Redis реалізовано засоби комунікації на базі блокованих черг і моделі «публікація/підписка» (publish-subscribe) [13; 14; 19; 21; 22].

4.3.2. Встановлення системи баз даних

Встановлюємо СБД в вигляді контейнера платформи Docker.

docker run --name some-redis -p 6379:6379 -d redis

```
PS C:\Users\dbs> docker run --name some-redis -p 6379:6379 -d redis
Unable to find image 'redis:latest' locally
latest: Pulling from library/redis
d121f8d1c412: Already exists
2f9874741855: Pull complete
d92da09ebfd4: Pull complete
bdfa64b72752: Pull complete
e748e6f663b9: Pull complete
eb1c8b66e2a1: Pull complete
Digest: sha256:1cfb205a988a9dae5f025c57b92e9643ec0e7ccff6e66bc639d8a5f95bba928c
Status: Downloaded newer image for redis:latest
72a0671ld4lab82ff9362lc926a3b819772482951e64e39b37f08e97f522dfae
PS C:\Users\dbs>
```

Рисунок 4.1. Встановлено СБД Redis в вигляді контейнера

Виконуємо запуск контейнеру.

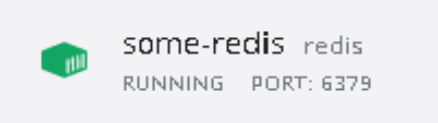


Рисунок 4.2. Виконано запуск контейнеру

Виконуємо запуск консолі контейнеру.

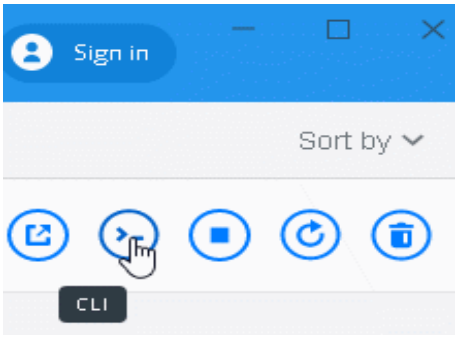


Рисунок 4.3. Виконано запуск консолі контейнеру

4.3.3. Створення бази даних

СБД Redis дедалі частіше застосовується як програмний інструментарій, котрий містить алгоритми для роботи з різноманітними

структурами даних, ніж базою даних певного типу. Значна кількість клієнтських бібліотек полегшує застосування СБД в різноманітних мовах програмування [19].

Виконуємо запуск консолі СБД, котра автоматично підключиться до сервера, розміщеного в локальній мережі, за портом 6379. В відповідь на команду *help*, СБД поверне інструкції щодо застосування довідки. Зазначивши перші літери і переміщуючись на клавіатурі клавішею *Tab*, СБД відображає написання наявних команд.

```
# redis-cli
127.0.0.1:6379> help
redis-cli 6.0.8
To get help about Redis commands type:
      "help @<group>" to get a list of commands in <group>
      "help <command>" for help on <command>
      "help <tab>" to get a list of possible help topics
      "quit" to exit

To set redis-cli preferences:
      ":set hints" enable online hints
      ":set nohints" disable online hints
Set your preferences in ~/.redisclirc
127.0.0.1:6379>
```

Рисунок 4.4. Консоль СБД Redis

Опції резервного дублювання бази даних

Redis підтримує декілька опцій дублювання бази даних, стандартним є зберігання тільки у внутрішній пам'яті, наприклад, оперативній. Команда *LASTSAVE* (або *INFO*) зчитує в форматі Unix значення змінної часу, котра відповідає часовій позначці завершального дублювання вмісту БД в зовнішню пам'ять [19; 21].

```
127.0.0.1:6379> LASTSAVE
(integer) 1600358284
127.0.0.1:6379>
```

Рисунок 4.5. Зчитано часову позначку завершального дублювання вмісту бази даних в зовнішню пам'ять

Команда *SAVE* (або *BGSAVE*) забезпечує включення асинхронного фонового дублювання даних в зовнішню пам'ять.

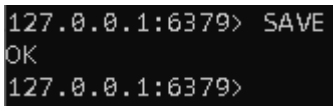


Рисунок 4.6. Включено дублювання даних в зовнішню пам'ять

Упереджувальне протоколювання

Redis підтримує файл *appendonly.aof* (AOF, Append Only File) упереджувального протоколювання, в котрому реєструються всі запити до СБД. Подібне функціонування забезпечує захищеність БД від раптових «вимкнень» серверу, коли СБД не встигає продублювати частину оновлених даних в зовнішню пам'ять, при перезапуску СБД виконує запротокольовані команди, відновивши стан БД, наявний до вимкнення. Для виконання упереджувального протоколювання, необхідно вказати значення *yes* шодо параметру *appendonly* в файлі конфігурації *redis.conf*.

appendonly yes

Далі потрібно вирішити, як часто вміст файлу *appendonly.aof* необхідно синхронізувати з зовнішньою пам'яттю.

 # appendfsync always
appendfsync everysec
 # appendfsync no

Опція «*always*» забезпечує максимальну надійність, відразу синхронізуючи кожну команду, але тоді обробка запитів є занадто повільною, що перекреслює основні переваги Redis. Стандартним встановлюється «*everysec*», коли команди синхронізуються раз в секунду, тоді при вимкненні сервера будуть втрачені тільки команди, виконані протягом завершальної секунди. За «*no*» синхронізацією з зовнішньою пам'яттю керує ОС.

Тимчасове зберігання екземплярів даних

Redis і подібні СБД «ключ/значення» часто застосовують для тимчасового зберігання значень. Встановлення інтервалу зберігання необхідне для врахування об'єму оперативної пам'яті при розростанні бази даних. Redis автоматично видаляє екземпляри даних з БД після завершення інтервалу часу.

Команда *EXPIRE* встановлює інтервал зберігання, параметрами є ключ екземпляру даних і цілочисельне значення інтервалу в секундах. Нижче встановимо інтервал зберігання 10 секунд. Якщо протягом даного часу перевірити існування ключа командою *EXISTS*, СБД поверне значення 1 (true). Після завершення інтервалу часу, команда *EXISTS* поверне значення 0 (false).

```
time
SET ice "I'm melting ..."
EXPIRE ice 10
time
EXISTS ice
time
time
EXISTS ice
```

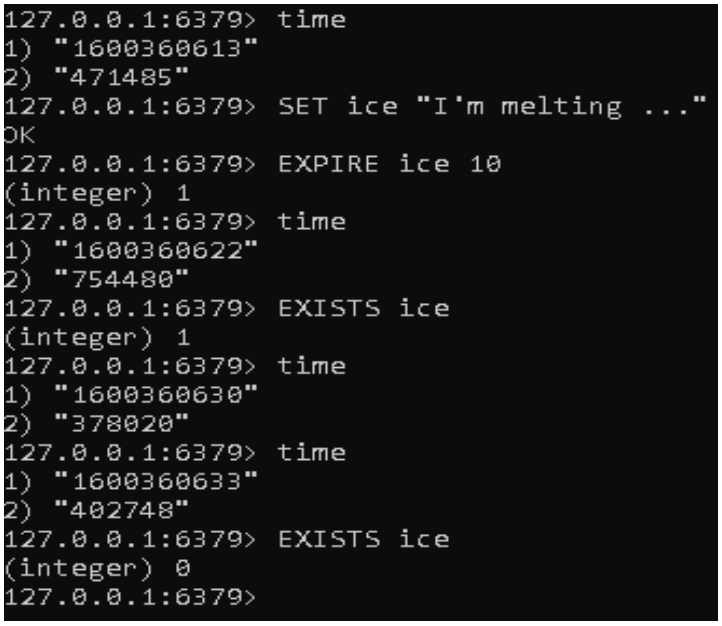


Рисунок 4.7. Встановлено інтервал зберігання ключа

В Redis реалізовано команду *SETEX* для об'єднання створення екземпляра і встановлення інтервалу зберігання. Команда *TTL* (time to live) зчитує різницю часу до видалення екземпляру даних.

SETEX ice 10 "I'm melting ..."

TTL ice

TTL ice

TTL ice

EXISTS ice

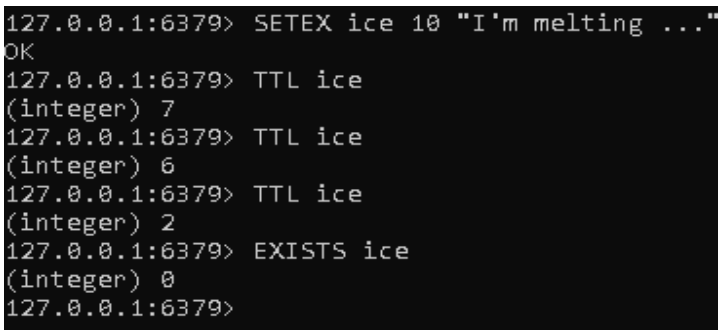


Рисунок 4.8. Зчитано різницю часу до видалення екземпляра даних

В будь-який момент до видалення, тимчасовість зберігання можна скасувати, виконавши команду *PERSIST*.

PERSIST ice

Команда *EXPIREAT* встановлює фіксовану часову позначку видалення в Unix-форматі – кількість секунд, обчислених з *1970-01-01T00:00:00*.

4.3.4. CRUD-запити до бази даних

Реалізуємо на базі Redis скорочення URL-посилань – за прикладом Веб-сервісів *tinyurl* чи *bitly*. Коли користувач переміщується за скороченим написанням посилання, інформаційна система звертається до БД і направляє користувача на Веб-сторінку згідно повного написання URL.

Команда *SET* співставляє скорочений ключ, наприклад *7wks*, значенню повного написання sevenweeks.org. Для зчитування значення за ключем виконуємо команду *GET*.

SET 7wks http://www.sevenweeks.org/
GET 7wks

```
127.0.0.1:6379> SET 7wks http://www.sevenweeks.org/
OK
127.0.0.1:6379> GET 7wks
"http://www.sevenweeks.org/"
127.0.0.1:6379>
```

Рисунок 4.9. Зчитано значення скороченого посилання

Для зменшення кількості запитів реалізовано команду *MSET*, котра створює декілька екземплярів «ключ/значення». Співставимо Google.com – значення gog, а Yahoo.com – yah. Далі зчитаємо командою *MGET* значення за списком ключів.

MSET gog http://www.google.com yah http://www.yahoo.com

MGET gog yah 7wks

```
127.0.0.1:6379> MSET gog http://www.google.com yah http://www.yahoo.com
OK
127.0.0.1:6379> MGET gog yah 7wks
1) "http://www.google.com"
2) "http://www.yahoo.com"
3) "http://www.sevenweeks.org/"
127.0.0.1:6379>
```

Рисунок 4.10. Зчитано значення за списком ключів

Окрім строкових, СБД розрізняє і цілочисельні значення. В Redis реалізовано команди обчислення статистики за екземплярами даних, наприклад, якщо потрібно відображати кількість наявних ключів, необхідно створити лічильник і збільшувати значення на одиницю командою *INCR* при дозаписуванні в БД нових екземплярів даних (див. підрозділ «Транзакції»).

SET count 2

INCR count

GET count

```
127.0.0.1:6379> SET count 2
OK
127.0.0.1:6379> INCR count
(integer) 3
127.0.0.1:6379> GET count
"3"
127.0.0.1:6379>
```

Рисунок 4.11. Збільшено лічильник на одиницю

Команда *INCR* визначає, що значення екземпляру цілочисельне і збільшує значення на одиницю, зменшення виконується командою *DECR*. При збільшенні строкового значення на одиницю, СБД поверне помилку.

SET bad_count "a"
INCR bad_count

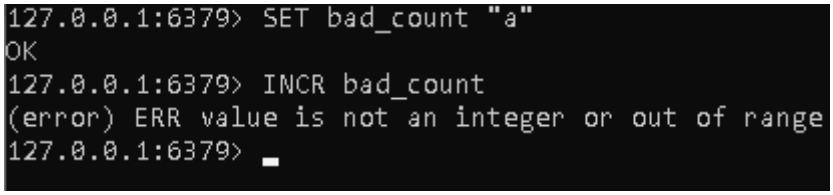


Рисунок 4.12. Відхилено збільшення текстового значення на одиницю

Реалізовані також команди збільшення (*INCRBY*) і зменшення (*DECRBY*) на будь-яке цілочисельне значення.

Простір іменувань

В СБД Redis «простір іменувань» – база даних, котрій співставляється ключ як ідентифікатор. Стандартно всі команди виконуються в просторі іменувань (базі даних) № 0.

Створимо кортеж *greeting* щодо значення «*Hello*».

SET greeting Hello

Рисунок 4.13. Дозаписано значення в простір № 0

Зчитування значення з поточного простору іменувань.

GET greeting

```
127.0.0.1:6379> GET greeting
"Hello"
```

Рисунок 4.14. Зчитано значення з простору № 0

Якщо командою *SELECT* переключитись в простір № 1, екземпляр даних *greeting* виявиться нестворeним.

SELECT 1

```
127.0.0.1:6379> SELECT 1
OK
```

Рисунок 4.15. Включено простір № 1

Умова прив'язки екземплярів даних до простору іменувань.

GET greeting

```
127.0.0.1:6379[1]> GET greeting
(nil)
```

Рисунок 4.16. Відхилено зчитування нестворeного екземпляру даних

Дозаписування екземпляру даних в новий простір іменувань.

SET greeting "Guten Tag"

```
127.0.0.1:6379[1]> SET greeting "Guten Tag"
OK
```

Рисунок 4.17. Дозаписано значення в простір № 1

SELECT 0

```
127.0.0.1:6379[1]> SELECT 0
OK
```

Рисунок 4.18. Включено простір № 0

GET greeting

```
127.0.0.1:6379> GET greeting
"Hello"
```

Рисунок 4.19. Зчитано значення екземпляру даних з простору № 0

Для переміщення кортежів між просторами іменувань реалізовано команду *MOVE*. Перемістимо екземпляр даних *greeting* в простір № 2.

MOVE greeting 2

```
127.0.0.1:6379> MOVE greeting 2
(integer) 1
```

Рисунок 4.20. Переміщено екземпляр даних в простір № 2

SELECT 2

```
127.0.0.1:6379> SELECT 2
OK
```

Рисунок 4.21. Включено простір № 2

GET greeting

```
127.0.0.1:6379[2]> GET greeting
"Hello"
```

Рисунок 4.22. Зчитано значення з простору № 2

SELECT 0

```
127.0.0.1:6379[1]> SELECT 0
OK
```

Рисунок 4.23. Включено простір № 0

4.3.5. Структури даних Redis

СБД Redis підтримує зберігання до 2^{32} значень в вигляді списку з єдиним ключем. Іменування команд в Redis характеризуються структурами даних.

Команди для множин починаються з літери S (set), для хешів – з H (hash), для відсортованих множин – з Z. Команди для списків починаються з літер L (left, зліва) або R (right, справа), відповідно до направленості виконання запитів.

Хеш

Хеші виконують в Redis функцію об'єктів-контейнерів, здатних зберігати довільну кількість кортежів «ключ/значення». Застосуємо хеш для статистики щодо користувачів, які зареєструвались в сервісі скорочення URL.

Дозапишемо в хеш ієрархічну структуру іменувань ключів і значень, сегментованих символом розмежування « : » за їх вкладеністю.

MSET user:eric:name "Eric Redmond" user:eric:password s3cret

MGET user:eric:name user:eric:password

```
127.0.0.1:6379> MSET user:eric:name "Eric Redmond" user:eric:password s3cret
OK
127.0.0.1:6379> MGET user:eric:name user:eric:password
1) "Eric Redmond"
2) "s3cret"
127.0.0.1:6379>
```

Рисунок 4.24. Зчитано значення хешу за ключами

Команда *HVALS* зчитує значення кортежів хешу за строковим ідентифікатором.

HMSET user:eric name "Eric Redmond" password s3cret
HVALS user:eric

```
127.0.0.1:6379> HMSET user:eric name "Eric Redmond" password s3cret
OK
127.0.0.1:6379> HVALS user:eric
1) "Eric Redmond"
2) "s3cret"
```

Рисунок 4.25. Зчитано значення кортежів хешу

Команда *HKEYS* зчитує ключі кортежів хешу.

HKEYS user:eric

```
127.0.0.1:6379> HKEYS user:eric
1) "name"
2) "password"
```

Рисунок 4.26. Зчитано ключі кортежів хешу

Для зчитування значення за ключем кортежу, зазначається ідентифікатор хешу (ідентифікатор множини кортежів) і ключ конкретного кортежу. Наприклад, зчитаємо значення кортежу за ключем *password*.

HGET user:eric password

```
127.0.0.1:6379> HGET user:eric password
"s3cret"
```

Рисунок 4.27. Зчитано значення за ідентифікатором хешу і ключем кортежу

Команда *HDEL* – видалення значення з хешу; *HINCRBY* – збільшення на одиницю; *HLEN* – розрахунок загальної кількості значень в хеші.

Список

Список містить послідовність значень і застосовується як черга (першим прийшов – першим оброблений), так і як стек (останнім прийшов – першим оброблений). Підтримується дозаписування в середину списку і переміщення значень між списками.

Реалізуємо зберігання списку URL-посилань, які збираються відвідати користувачі. Дозапишемо в хеш <*USERNAME*>:*wishlist* список скорочень посилань, які певний користувач планує відвідати.

RPUSH eric:wishlist 7wks gog stor

```
127.0.0.1:6379> RPUSH eric:wishlist 7wks gog store
(integer) 3
127.0.0.1:6379>
```

Рисунок 4.28. Дозаписано значення в список

Як і в більшості команд дозаписування значень, Redis повертає кількість створених значень. Якщо записати три значення, тоді СБД поверне в відповідь значення 3. Команда *LLEN* зчитує кількість значень, наявних в списку.

Команда *LRANGE* зчитує частину списку за вказаним інтервалом номерів послідовності. Перше значення має індекс 0. Від'ємне значення позначає початок відліку з кінця списку.

LRANGE eric:wishlist 0 -1

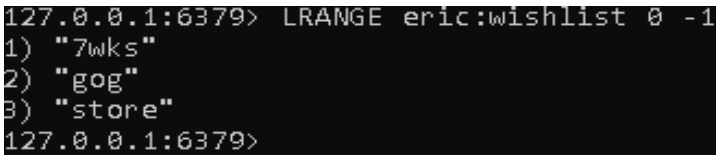


Рисунок 4.29. Зчитано значення за розміщенням в списку

Якщо кількість дорівнює 0, тоді команда *LREM* видаляє всі знайдені кортежі з вказаним значенням ключа. Якщо лічильник > 0, тоді буде видалено вказану кількість значень.

LREM eric:wishlist 0 gog

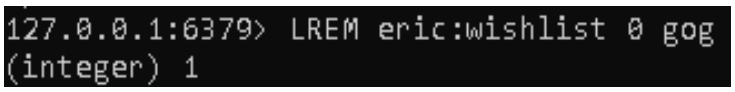


Рисунок 4.30. Видалено кортежи зі списку

Для застосування списку як стеку, потрібно дозаписувати значення командою *RPUSH* і зчитувати – *RPOP*.

Для переміщення значень між списками в Redis реалізовано команду *RPOPLPUSH* – об'єднання команд видалення з кінця поточного списку (pop справа) і дозаписування в початок цільового списку (push зліва).

RPOPLPUSH eric:wishlist eric:visited

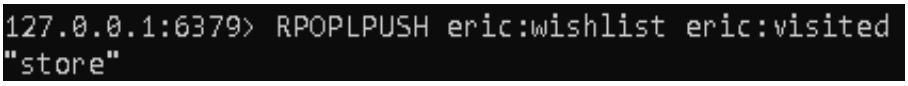


Рисунок 4.31. Видалено з поточного списку і дозаписано в цільовий список

Множина

Доповнимо сервіс опцією об'єднання URL-посилань за категоріями, наприклад, соціальні новини, новини технологій тощо.

Множиною називається список, котрий не містить повторювані значення і призначений для обчислення об'єднання, перетину або різниці. Для об'єднання URL-посилань за загальним ключем, створюють множину і дозаписують декілька значень командою *SADD*.

SADD news nytimes.com store.com

Рисунок 4.32. Дозаписано значення множини

Redis повертає кількість дозаписаних значень.

Команда *SMEMBERS* зчитує всі значення множини, але без сортування.

SMEMBERS news

Рисунок 4.33. Зчитано значення з множини

Створимо категорію *tech* для Веб-ресурсів з новинами технологій.

SADD tech store.com apple.com

Рисунок 4.34. Дозаписано значення створеної множини

Для з'ясування, які Веб-ресурси публікують соціальні і технічні новини, виконуємо перетин множин командою *SINTER*.

SINTER news tech

```
127.0.0.1:6379> SINTER news tech
1) "store.com"
```

Рисунок 4.35. Обчислено перетин множин

Для розрахунку різниці множин – видалення значень, котрі містяться в порівнюваній множині, наприклад, знаходження тільки соціальних новин, виконуємо команду *SDIFF*.

SDIFF news tech

```
127.0.0.1:6379> SDIFF news tech
1) "nytimes.com"
```

Рисунок 4.36. Обчислено різницю значень множин

За особливістю множин, об'єднання видаляє ідентичні значення.

SUNION news tech

```
127.0.0.1:6379> SUNION news tech
1) "nytimes.com"
2) "store.com"
3) "apple.com"
```

Рисунок 4.37. Об'єднано множини

Команда *SUNIONSTORE* за результатом об'єднання створює множину.

SUNIONSTORE websites news tech

```
127.0.0.1:6379> SUNIONSTORE websites news tech
(integer) 3
```

Рисунок 4.38. Створено множину за результатом об'єднання

В Redis для зберігання перетину в новій множині реалізовано команду *SINTERSTORE*, для зберігання різниці – *SDIFFSTORE*.

SMEMBERS websites

```
127.0.0.1:6379> SMEMBERS websites
1) "nytimes.com"
2) "store.com"
3) "apple.com"
```

Рисунок 4.39. Зчитано значення множини

SMOVE переміщує значення між множинами, *SREM key [value ...]* видаляє конкретне значення, *SCARD* обчислює загальну кількість значень.

4.3.6. Відсортовані множини

Відсортовані множини структуровані як списки і не містять повторюваних значень як множини. Для дозаписування значення в відсортовану множину необхідно вказати назву множини і масив кортежів «ключ/значення», де ключ – цілочисельний лічильник, значення – строкове.

ZADD visits 500 7wks 9 gog 9999 store

```
127.0.0.1:6379> ZADD visits 500 7wks 9 gog 9999 store
(integer) 3
```

Рисунок 4.40. Дозаписано значення відсортованої множини

ZINCRBY збільшує лічильник на фіксоване значення (якщо значення від'ємне, тоді зменшує).

ZINCRBY visits 1 store

Рисунок 4.41. Збільшено лічильник на одиницю

Відсортовані множини підтримують створення нових множин за результатом об'єднання або перетину ключів. Шаблон запиту об'єднання:

ZUNIONSTORE destination numkeys key [key ...]
[WEIGHTS weight [weight ...]] [AGGREGATE SUM | MIN | MAX]

Ключ *destination* – іменування нової множини; numkeys – кількість ключів; *key* – список назв ключів; *weight* – список вагових коефіцієнтів, на котрі будуть помножені значення відсортованої множини, наприклад, за наявності двох ключів має бути зазначено два вагових коефіцієнти; *AGGREGATE* – функція розрахунку статистичних показників.

Сервіс скорочення URL виконує і соціальну функцію, наприклад, коли користувачі залишають відгуки щодо вмісту відвіданих посилань. Створимо новий ключ *votes* з відгуками користувачів щодо вмісту Веб-ресурів.

ZADD votes 2 7wks 0 gog 9001 store

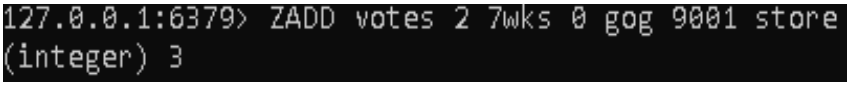


Рисунок 4.42. Дозаписано значення відсортованої множини

Об'єднаємо значення множин відгуків і відвідувань, розрахуємо на базі них значення «важливості» посилань. Збільшимо *votes* вдвічі.

ZUNIONSTORE importance 2 visits votes WEIGHTS 1 2 AGGREGATE SUM

```
127.0.0.1:6379> ZUNIONSTORE importance 2 visits votes WEIGHTS 1 2 AGGREGATE S
UM
(integer) 3
```

Рисунок 4.43. Об'єднано значення відсортованих множин

Зчитаємо відсортовану множину в інтервалі значень.

ZRANGEBYSCORE importance -inf inf WITHSCORES

```
127.0.0.1:6379> ZRANGEBYSCORE importance -inf inf WITHSCORES
1) "gog"
2) "9"
3) "7wks"
4) "504"
5) "store"
6) "28002"
```

Рисунок 4.44. Зчитано відсортовані значення

Подвоїмо значення відсортованої множини *votes*.

ZUNIONSTORE votes 1 votes WEIGHTS 2

```
127.0.0.1:6379> ZUNIONSTORE votes 1 votes WEIGHTS 2
(integer) 3
```

Рисунок 4.45. Оновлено значення відсортованої множини

ZRANGE votes 0 -1 WITHSCORES

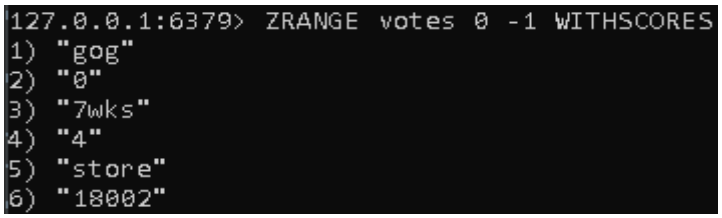


Рисунок 4.46. Зчитано кортежі відсортованої множини

Команда *ZINTERSTORE* обчислює перетин відсортованих множин.

4.3.7. Зчитування за інтервалом значень

Для зчитування значень відсортованої множини за вказаним інтервалом існує команда *ZRANGE*, подібно до *LRANGE*, застосованої до списку.

Зчитаємо декілька посилань з початку списку *visits*.

ZRANGE visits 0 1

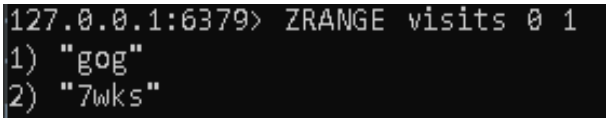


Рисунок 4.47. Зчитано значення відсортованої множини за інтервалом

Для зчитування за зменшенням значень лічильника, запит доповнюють префіксом *REV*, наприклад *ZREVRANGE*. Параметр *WITHSCORES* позначає необхідність зчитування значень лічильника.

ZREVRANGE visits 0 -1 WITHSCORES

```
127.0.0.1:6379> ZREVRANGE visits 0 -1 WITHSCORES
1) "store"
2) "10000"
3) "7wks"
4) "500"
5) "gog"
6) "9"
```

Рисунок 4.48. Зчитано значення відсортованої множини
за зменшенням значень лічильника

Стандартно нижня і верхня межі враховуються для вказаного інтервалу.

ZRANGEBYSCORE visits 9 9999

```
127.0.0.1:6379> ZRANGEBYSCORE visits 9 9999
1) "gog"
2) "7wks"
```

Рисунок 4.49. Зчитано значення відсортованої множини за інтервалом

Для виключення меж, на початку інтервалу допускається зазначення відкриваючої дужки «(». Для прикладу зчитаємо всі лічильники за інтервалом значень (9, 9999].

ZRANGEBYSCORE visits (9 9999

```
127.0.0.1:6379> ZRANGEBYSCORE visits (9 9999
1) "7wks"
```

Рисунок 4.50. Зчитано значення за інтервалом з виключенням меж

Допускається зчитування значень за інтервалом з нескінченними межами.

ZRANGEBYSCORE visits -inf inf

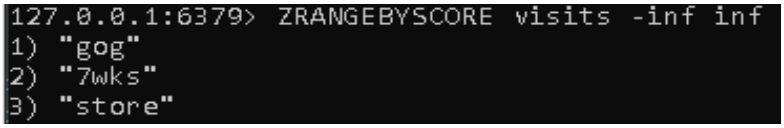


Рисунок 4.51. Зчитано значення за інтервалом з нескінченними межами

Команда *ZREVRANGEBYSCORE* зчитує вміст відсортованої множини за зменшенням значення лічильника.

ZREVRANGEBYSCORE visits inf -inf

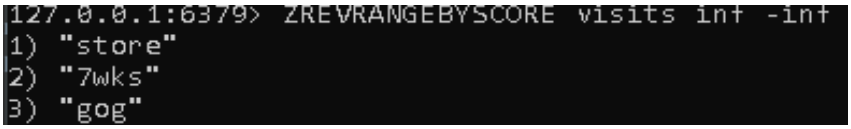


Рисунок 4.52. Зчитано значення за інтервалом за зменшенням значень ключів

Для видалення екземплярів даних за інтервалом також реалізовано команди *ZREMRANGEBYRANK* і *ZREMRANGEBYSCORE*.

4.3.8. Транзакції

Команда *MULTI* в Redis позначає початок атомарного блоку команд. Якщо розмістити в єдиному блоці декілька команд, наприклад, *SET* і *INCR*, тоді або всі виконуються коректно, або всі не виконуються – частковий результат неможливий.

Транзакція починається командою *MULTI* і завершується командою *EXEC* – виконання списку команд транзакції. Виконуємо скорочення URL і збільшення лічильника в межах єдиної транзакції.

MULTI
SET store http://store.com
INCR count
EXEC

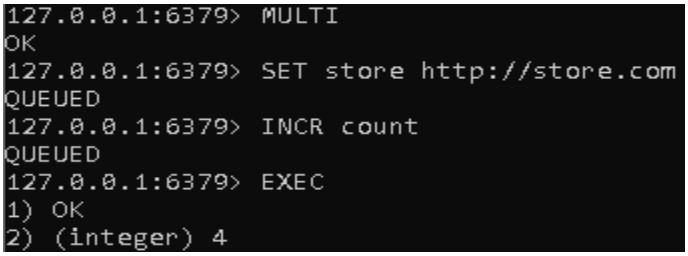


Рисунок 4.53. Виконано послідовність команд транзакції

В блоці *MULTI* команди, в момент зазначення в консолі, відразу не виконуються, подібно до транзакцій в реляційних СБД. Так як виконання сервера однопотокове, команди спочатку розміщуються в черзі і після появи команди *EXEC* виконуються послідовно.

Конвеєр команд

Конвеєр виконує декілька команд, розмежованих символами « \r\n », в межах єдиного запиту.

echo "PING \r\n PING \r\n PING \r\n" | redis-cli

Рисунок 4.54. Конвеєр команд

4.3.9. Моделі комунікації

Блокована черга

Створимо підсистему обміну повідомленнями, коли різноманітні джерела дозаписують повідомлення в чергу, а читач їх зчитує. Для вирішення даного завдання в Redis реалізовано блоковані черги.

Виконуємо запуск додаткової консолі клієнтської утиліти redis-cli. Команда *BRPOP* очікує протягом вказаного інтервалу часу, зазначеного в секундах, створення екземплярів даних блокованої черги (див. рис. 4.56).

Перемістимось до першої консолі і командою *LPUSH* дозапишемо повідомлення в блоковану чергу *comments*.

LPUSH comments"store is great! I buy all my books there."

```
# redis-cli
127.0.0.1:6379> LPUSH comments "store is great! I buy all my books there."
(integer) 1
```

Рисунок 4.55. Дозаписано значення в блоковану чергу

Перемістившись до другої консолі побачимо, що зчитано значення ключа, значення повідомлення, час очікування.

BRPOP comments 600

```
127.0.0.1:6379>  BRPOP comments 600
1) "comments"
2) "store is great! I buy all my books there."
(2.29s)
```

Рисунок 4.56. Зчитано значення з блокованої черги

Існують блоковані версії команд *LPOP – BLPOP*; *RPOPLPUSH – BRPOPLPUSH*

Модель «публікація/підписка»

Вдосконалимо коментування, вже розроблене на базі блокованих черг, забезпечивши зчитування коментарів читачами. Встановлення з'єднання читачі виконують за ключем, іменованим «каналом» в контексті моделі «публікація/підписка».

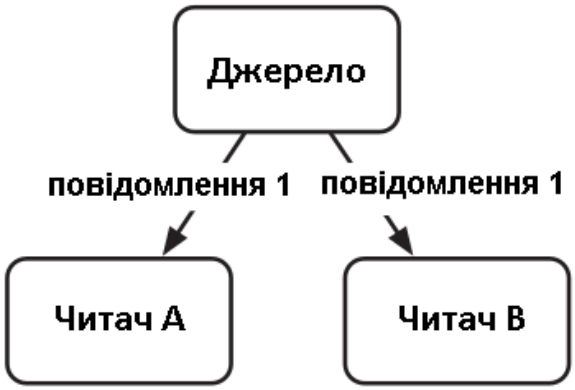


Рисунок 4.57. Направлено повідомлення до читачів

Виконуємо запуск додаткових двох консолей Redis. Під'єднаємо читачів до каналу коментарів. Після встановлення з'єднання з джерелом, консоль перебуває в стані зчитування.

SUBSCRIBE comments

```
127.0.0.1:6379> SUBSCRIBE comments
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "comments"
3) (integer) 1
```

Рисунок 4.58. Встановлено з'єднання з джерелом

Опублікуємо по каналу *comments* строкове повідомлення. Команда *PUBLISH* поверне значення 2, котре відповідає кількості читачів.

PUBLISH comments "Check out this shortcoded site! 7Wks"

```
# redis-cli
127.0.0.1:6379> PUBLISH comments "Check out this shortcoded site! 7Wks"
(integer) 2
```

Рисунок 4.59. Опубліковано повідомлення по каналу від джерела

Обом читачам надіслано «громіздку відповідь» (англ. multibulk reply) – список з трьох значень: ключ «*message*», назва каналу, повідомлення.

```
127.0.0.1:6379> SUBSCRIBE comments
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "comments"
3) (integer) 1
1) "message"
2) "comments"
3) "Check out this shortcoded site! 7Wks"
```

Рисунок 4.60. Зчитано повідомлення консолями читачів

За необхідності читач може від'єднатись командою *UNSUBSCRIBE comments*, або командою *UNSUBSCRIBE* без параметрів – для від'єднання від всіх каналів. Для припинення виконання консолі сервера, в консолі необхідно переключитись клавішами CTRL+C.

В Redis реалізовано команди, наприклад, *RENAME* – перейменування ключа екземпляру, *TYPE* – зчитування типу значення ключа, *DEL* – видалення екземпляру даних. Зокрема є *FLUSHDB* для видалення всіх екземплярів даних БД, і різновид *FLUSHALL* – видаляє всі екземпляри даних з всіх БД.

Вимикаємо контейнер.

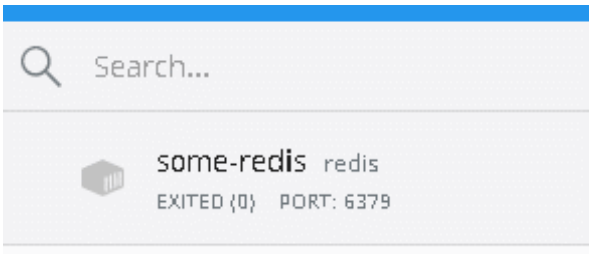


Рисунок 4.60. Вимкнено контейнер

4.4. Висновки

Redis застосовують для зберігання списків (черга або стек), множин (допускається обчислення об'єднання, перетину, різниці), хешів, блокованих черг. Розмежування послідовностей екземплярів даних, за прикладом баз даних, реалізовано на базі просторів іменувань.

СБД підтримує об'єднання послідовностей команд в транзакції, конвеєрне виконання команд, а також інтервали зберігання даних подібно до тимчасової пам'яті. В Redis реалізовані засоби комунікації на базі блокованих черг і за моделлю «публікація/підписка».

4.4.1. Теоретичне завдання

1. Знайдіть документацію щодо команд Redis зі значеннями складності їх виконання в нотації «О-велике» – O(x).

2. Ознайомтесь з різноманітними шаблонами обміну повідомленнями та з'ясуйте, які реалізовані в Redis.

4.4.2. Практичне завдання

1. Встановіть бібліотеку мови програмування за вашим вибором і програмно реалізуйте підключення до сервера Redis. Створіть і збільште на одиницю в межах транзакції цілочисельний екземпляр даних.

2. Застосовуючи API програмно реалізуйте: функція № 1 – дозаписує дані в блоковану чергу; функція № 2 – зчитує екземпляри даних блокованої черги і друкує їх в текстовий файл.

4.4.3. Питання для закріплення знань

1. Випадки, коли рекомендовано застосовувати СБД «ключ/значення».
2. Випадки, коли не рекомендовано застосовувати СБД «ключ/ значення».
3. Відмінність типів даних «черга» і «стек». Приклад створення черги і стеку в СБД Redis.
4. Назначення просторів іменувань в СБД Redis. Приклад переміщення екземпляра даних між просторами іменувань.
5. Що називається каналом в контексті моделі «публікація/підписка»? Навести приклад створення і дозаписування даних.
6. Відмінність типів даних «список» і «множина». Приклад створення списку і множини в СБД Redis.
7. Транзакції в СБД Redis, приклад виконання.

Розділ 5. Системи стовпцевих баз даних

За архітектурою, стовпцеві бази даних подібні до реляційних СБД і СБД «ключ/значення». Розповсюджені приклади – HBase, Cassandra і Hypertable.

Порівнюючи з реляційними базами даних, котрі зберігають значення рядка таблиці послідовно в зовнішній пам'яті, в стовпцевих БД послідовно зберігаються значення стовпця [13; 14; 19-21]. Так як кількість стовпців таблиці є нефіксованою, рядки можуть взагалі не містити значень, і тоді таблиця буде розрідженою без витрат на зберігання значень *NULL*.

5.1. Випадки, коли рекомендовано застосовувати

Стовпцеві бази даних були розроблені з метою забезпечення горизонтальної масштабованості для завдань обробки значних об'ємів даних, наприклад, Веб-сторінок, вміст котрих періодично оновлюється. Подібні системи підтримують створення кластерів з десятків, сотень або тисяч вузлів [19-21].

5.2. Випадки, коли не рекомендовано застосовувати

Стовпцеві СБД мають різноманітні особливості реалізації, але всі потребують проектування схем баз даних з врахуванням ймовірних запитів до БД. Тобто щодо стовпцевих СБД, необхідно апріорно розуміти вміст стовпців, і які конкретно запити будуть сформовані до значень екземплярів даних БД.

5.3. СБД HBase

Поняття, котрі наявні в HBase, подібні до реляційних СБД, але тільки за їх написанням. Наприклад, HBase зберігає дані в контейнерах, які називаються «таблиці». Таблиці складаються з комірок, котрі знаходяться на перетині «рядків» і «стовпців» [21; 22].

Таблиці в HBase відрізняються від реляційних відношень, рядки відрізняються від реляційних кортежів і варіюється кількість значень в стовпцях.

HBase характеризується певними функціями, котрі нереалізовані або частково реалізовані в інших СБД, наприклад, стиснення даних, протоколювання версій екземплярів даних (СБД CouchDB), тимчасове зберігання даних (СБД Redis).

5.3.1. Передісторія

СБД HBase побудована на базі Hadoop – платформи розподілених обчислень, об'єднує розподілену файлову підсистему HDFS і засоби розподіленої обробки даних «map-reduce». Розроблена за зразком BigTable – високопродуктивної комерційної СБД, створеної компанією Google.

З архітектурної точки зору, завдяки журналюванню запитів, подібно до СБД Redis, і їх розподіленню між вузлами кластеру, забезпечується доступність даних навіть при відключенні окремих серверів [20-22].

5.3.2. Встановлення системи баз даних

Встановлюємо СБД в вигляді контейнера платформи Docker.

docker run --name some-hbase -ti harisekhon/hbase

```
PS C:\Users\dbs> docker run --name some-hbase -ti harisekhon/hbase
Unable to find image 'harisekhon/hbase:latest' locally
latest: Pulling from harisekhon/hbase
cd784148e348: Pull complete
9375f15adfac: Pull complete
bd1652f47081: Pull complete
51732cf9b3b1: Pull complete
1560972d8dcf: Pull complete
13bf4aef219a: Pull complete
a48e72e9439e: Pull complete
2b49c23b973d: Pull complete
9e4c2bb46a65: Pull complete
Digest: sha256:c65ce56799f59fab86eb8995a628c878c28c03189c0fd27285c6ebef4cb016fa
Status: Downloaded newer image for harisekhon/hbase:latest
```

Рисунок 5.1. Встановлено СБД HBase в вигляді контейнера

Запуск складових кластеру – керуючий сервер, службовий сервер, координатор Zookeeper.

```
+ echo 'Starting Zookeeper...'
Starting Zookeeper...
+ echo

+ start_master
+ echo 'Starting HBase Master...'
Starting HBase Master...
+ /hbase/bin/hbase-daemon.sh start master
+ /hbase/bin/hbase zookeeper
running master, logging to /hbase/logs/hbase--master-a90022aa9764.out
+ echo

+ start_regionserver
+ echo 'Starting HBase RegionServer...'
Starting HBase RegionServer...
++ cut -c 1
++ sed s,/hbase-,,
++ echo /hbase-2.1.3 /hbase-data
+ '[' 2 = 0 ']'
+ /hbase/bin/hbase-daemon.sh start regionserver
running regionserver, logging to /hbase/logs/hbase--regionserver-a90022aa9764.out
+ echo
```

Рисунок 5.2. Виконано запуск програмних компонентів кластеру

Після запуску компонентів кластеру, слідує запуск консолі СБД.

```
Now starting HBase Shell...

+ /hbase/bin/hbase shell


2020-09-17 20:14:19,687 WARN  [main] util.NativeCodeLoader: Unable to load native-hadoop lib
rary for your platform... using builtin-java classes where applicable
HBase Shell
Use "help" to get list of supported commands.
Use "exit" to quit this interactive shell.
For Reference, please visit: http://hbase.apache.org/2.0/book.html#shell
Version 2.1.3, rda5ec9e4c06c537213883cca8f3cc9a7c19daf67, Mon Feb 11 15:45:33 CST 2019
Took 0.0278 seconds
hbase(main):001:0>
```

Рисунок 5.3. Виконано запуск консолі СБД

Виконуємо запуск контейнера.

Рисунок 5.4. Виконано запуск контейнера

Виконуємо запуск консолі контейнеру.

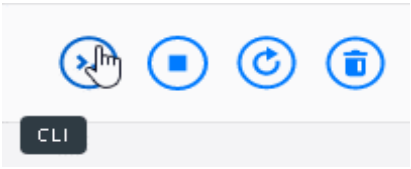


Рисунок 5.5. Виконано запуск консолі контейнеру

Встановлюємо змінну розміщення каталогу СБД в ОС контейнеру.

export HBase_HOME=/hbase-2.1.3/

cd $HBase_HOME

```
/ # export HBase_HOME=/hbase-2.1.3/
/ # cd $HBase_HOME
/hbase-2.1.3 #
```

Рисунок 5.6. Створено змінну розміщення СБД

Дивимось вміст каталогу *bin*.

cd $HBase_HOME/bin/ ; ls

```
/hbase-2.1.3/conf # cd $HBase_HOME/bin/ ; ls
considerAsDead.sh         hbase-config.sh           master-backup.sh          start-hbase.sh
draining_servers.rb       hbase-daemon.sh           region_mover.rb           stop-hbase.cmd
get-active-master.rb      hbase-daemons.sh          region_status.rb          stop-hbase.sh
graceful_stop.sh          hbase-jruby               regionservers.sh          test
hbase                     hbase.cmd                 replication               zookeepers.sh
hbase-cleanup.sh          hirb.rb                   rolling-restart.sh
hbase-common.sh           local-master-backup.sh    shutdown_regionserver.rb
hbase-config.cmd          local-regionservers.sh    start-hbase.cmd
/hbase-2.1.3/bin #
```

Рисунок 5.7. Вміст каталогу СБД

Виконуємо запуск серверу.

$HBase_HOME/bin/start-hbase.sh

```
/hbase-2.1.3/bin # start-hbase.sh
localhost: /hbase/bin/zookeepers.sh: line 52: ssh: command not found
running master, logging to /hbase/bin/../logs/hbase--master-bec5e8df4061.out
: running regionserver, logging to /hbase/bin/../logs/hbase--regionserver-bec5e8df4061.out
/hbase-2.1.3/bin #
```

Рисунок 5.8. Виконано запуск сервера

Виконуємо запуск консолі СБД.

$HBase_HOME/bin/hbase shell

```
/hbase-2.1.3/bin # $HBase_HOME/bin/hbase shell
2020-09-17 21:06:43,087 WARN  [main] util.NativeCodeLoader: Unable to load native-hadoop
ing builtin-java classes where applicable
HBase Shell
Use "help" to get list of supported commands.
Use "exit" to quit this interactive shell.
For Reference, please visit: http://hbase.apache.org/2.0/book.html#shell
Version 2.1.3, rda5ec9e4c06c537213883cca8f3cc9a7c19daf67, Mon Feb 11 15:45:33 CST 2019
Took 0.0119 seconds
hbase(main):001:0>
```

Рисунок 5.9. Виконано запуск консолі СБД

Консоль HBase підтримує команди: *help* – зчитування довідки за наявними операторами; *status* – зчитування інформацію щодо поточного стану кластеру.

Перевіряємо стан кластеру.

status

```
hbase(main):003:0> status
1 active master, 0 backup masters, 1 servers, 0 dead, 2.0000 average load
Took 2.2066 seconds
hbase(main):004:0>
```

Рисунок 5.10. Перевірено стан кластеру

5.3.3. Архітектура

Спочатку розберемо деякі поняття, характерні для HBase [13; 14; 19; 20; 22]:

– **Таблиця (Table)** – зміст подібний до таблиць реляційних СБД.

– **Простір іменувань (Namespace)** – множина таблиць подібна до множини баз даних реляційної СБД.

– **Регіон (Region)** – екземпляри даних таблиці HBase, сегментовані за діапазоном ключів.

– **Службовий сервер (RegionServer)** – вузол, містить регіони.

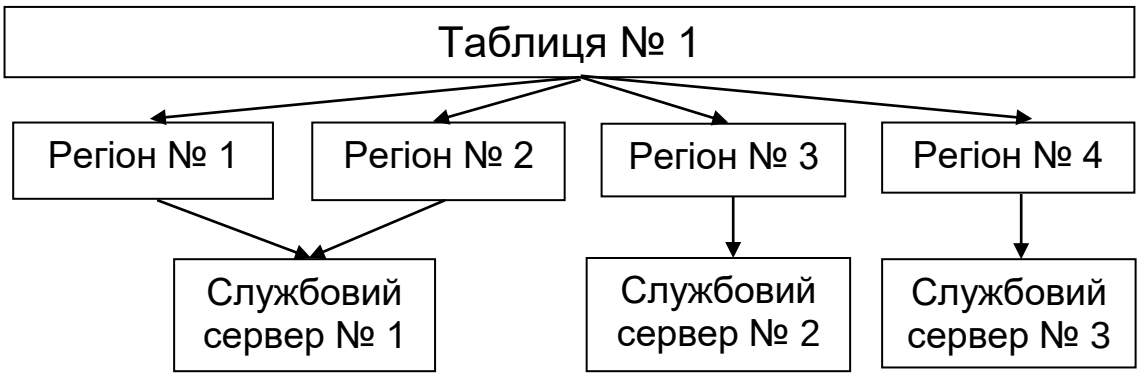


Рисунок 5.11. Приклад таблиці HBase, розподіленої за регіонами

Архітектура кластеру HBase об'єднує програмні компоненти: СБД, координатор Zookeeper, файлова підсистема HDFS [22].

Zookeeper

Apache Zookeeper - координатор розподілених обчислень. Zookeeper забезпечує перевірку стану всіх серверів регіонів як діючих, так і видалених з мережі; метаданих, взаємозв'язаних з поділом кластера на регіони і реплікуванням даних між серверами.

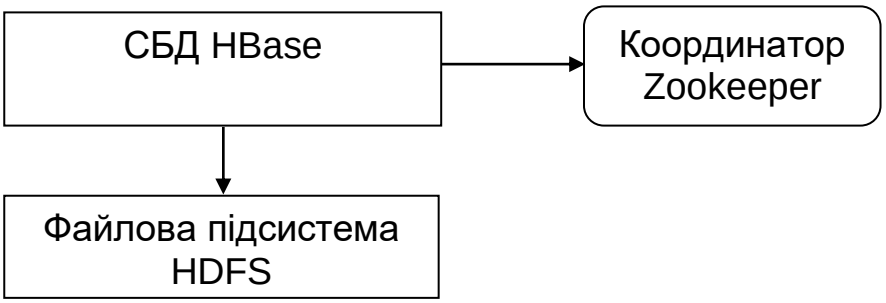


Рисунок 5.12. Програмні компоненти кластеру HBase

HDFS

HDFS (Hadoop Distributed File System) – розподілена файлова підсистема, створена за моделлю Google File System (GFS) в складі проекту розподілених обчислень Hadoop для зберігання і обробки значних об'ємів даних. HDFS підтримує файли об'ємом > 100 Тбайт даних, котрі зберігаються в вигляді множини блоків, розподілених за вузлами кластеру [20; 22].

Метадані блоків містяться в NameNode. Підсистема NameNode виконує адміністрування файлової підсистеми HDFS, зокрема створення каталогів, їх облік, переміщення файлів тощо. Утиліта HDFS реалізує інтерфейс розподіленої файлової підсистеми, забезпечує клієнтські програми абстракцією взаємодії з єдиним файлом. Підсистема DataNode виконує запити дозаписування/зчитування щодо конкретних блоків даних.

HDFS забезпечує надійність роботи HBase, завдяки реплікуванню даних між DataNode, при вимкнені певного DataNode, RegionServer зчитує дані з репліки DataNode. HBase RegionServer і HDFS DataNode розміщуються на

всіх службових серверах кластеру; HBase Master і HDFS NameNode – на керуючому сервері [22].

Керуючий сервер

Керуючий сервер (HBase master) забезпечує адміністрування кластера HBase, координування запитів до вузлів регіонів, створення таблиць, сімейств стовпців, зчитування реплік DataNode [22].

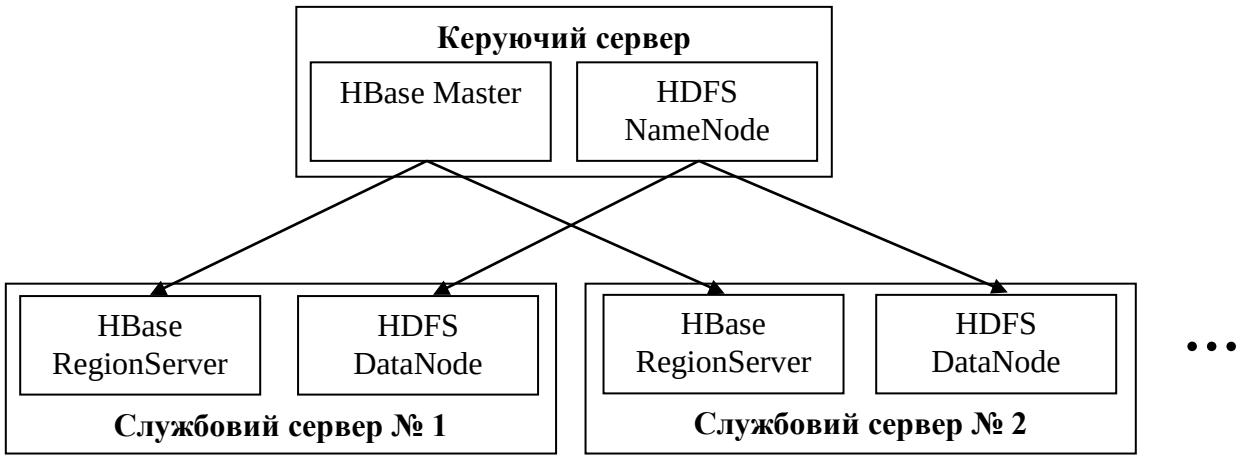


Рисунок 5.13. Приклад розподіленої бази даних

Керуючий сервер періодично перевіряє поточний розподіл регіонів, і якщо розподіл нерівномірний, переміщує регіони між вузлами кластеру, виконує координацію роботи службових вузлів, зупинення і запуск регіонів. Коли службовий сервер HBase припиняє обробку запитів, координатор Zookeeper виявляє дану ситуацію і повідомляє керуючий сервер щодо необхідності направлення запитів до нового службового сервера.

В конфігурації кластеру зазначається декілька екземплярів керуючого серверу. Якщо основний екземпляр керуючого серверу припиняє направляти запити до Zookeeper, тоді резервний переймає керування.

Службовий сервер

На фізичних вузлах кластеру стандартно виконуються підсистеми RegionServer і DataNode [22]. RegionServer містить регіони, в відповідь на запити клієнтських програм зчитує дані, сегментовані за діапазонами ключів.

Регіони розміщуються в єдиному або декількох файлах HDFS. Запити дозаписування/зчитування RegionServer виконує за DataNode, розміщеному на тому ж вузлі. RegionServer підтримує тимчасове зберігання за алгоритмом Least Recently Used (LRU), подібно до EXPIRE в СБД Redis.

5.3.4. Модель зберігання екземплярів даних

«Кластер» HBase містить «простори іменувань» (англ. «namespace») [22]. «Простір іменувань» – множина «таблиць». «Таблиця» в HBase складається з рядків і стовпців. Рядки таблиці групуються в «регіон» за відповідним інтервалом значень стовпця «ROW key» (див. табл. 5.1).

В доповнення до рядків і стовпців HBase має конструкцію «сімейств стовпців». Комірка містить ключ рядка, версію (часова позначка оновлення БД), назву сімейства стовпців (наприклад, *metrics*), ідентифікатор стовпця (наприклад, *temperature*).

Таблиця 5.1. Структура таблиці HBase

Ключ рядка (ROW key)	**Часова позначка (timestamp)**	**Сімейство стовпців: ідентифікатор стовпця « *metrics:humidity* »**	**Сімейство стовпців: ідентифікатор стовпця *« metrics:temperature* »**
k1	t1	val1	val6
k1	t2	val2	val7
k2	t1	val3	val8
k3	t1	val4	val9
k3	t3	val5	val10

Рядок, версія, сімейство стовпців і ідентифікатора стовпця формують комірку HBase – логічну одиницю зберігання даних СБД HBase. Всі запити

створення/оновлення рядків супроводжуються встановленням відповідних часових позначок, застосовуваних при фільтрації екземплярів даних таблиці.

Якщо таблиця містить два сімейства стовпців (column families, CF), тоді регіони містять по 2 HFile з частинами рядків CF (див. рис. 5.13 й рис. 5.14). Стиснення таблиць виконується кодеками snappy, gzip, lzo.

В HBase немає типів даних, байти можуть містити будь-яке значення – строкове, чисельне, дату тощо. Розробнику програмного забезпечення необхідно зберігати типи значень комірок для коректного зчитування.

5.3.5. CRUD-запити до бази даних

Спочатку виконуємо запуск локального екземпляру серверу HBase в автономному стані, далі розберемо як в консолі HBase створювати і редагувати таблиці, дозаписувати і оновлювати значення екземплярів даних.

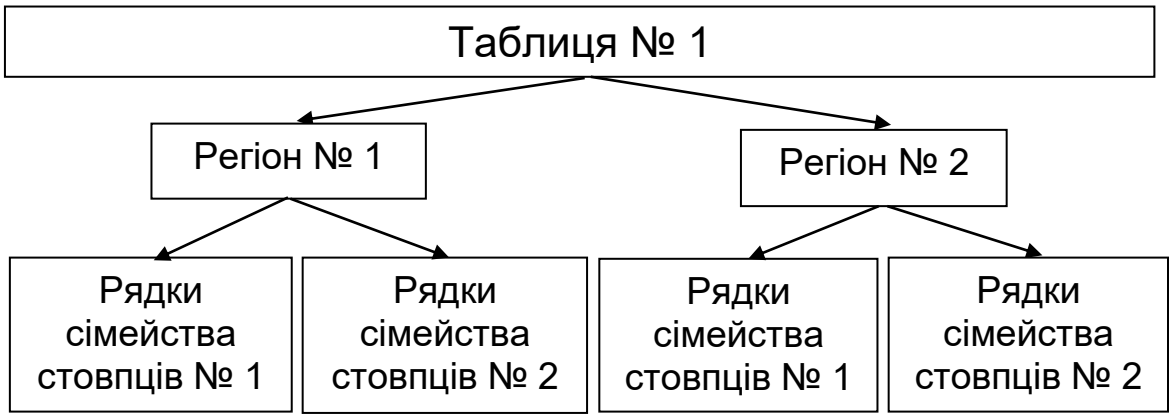


Рисунок 5.14. Приклад розподілу сімейств стовпців за регіонами

Прийнято вважати, що кластер HBase має складатись щонайменше з п'яти серверів.

Кластер HBase підтримує три варіанти виконання:

- автономний стан з єдиним вузлом;

- псевдорозподіленний стан з єдиним вузлом, емулює кластер;

- повністю розподілений стан, об'єднує вузли в кластер.

Назва стовпця складається з двох частин – іменування сімейства стовпців і ідентифікатора стовпця, котрі об'єднуються в єдине строкове значення (наприклад, « family:qualifier »).

На рис. 5.15 зображено приклад вмісту тестової таблиці з двома сімействами стовпців: *color* і *shape*. В таблиці є два рядки позначені пунктирними прямокутниками – *first* і *second*. Рядок *first* містить три стовпці з сімейства *color* з ідентифікаторами *red*, *blue* й *yellow* і стовпець з сімейства *shape* з ідентифікаторами *square* й *triangle*. Наприклад, кортеж «*first/color:red*» вказує на комірку зі значенням «*#F00*».

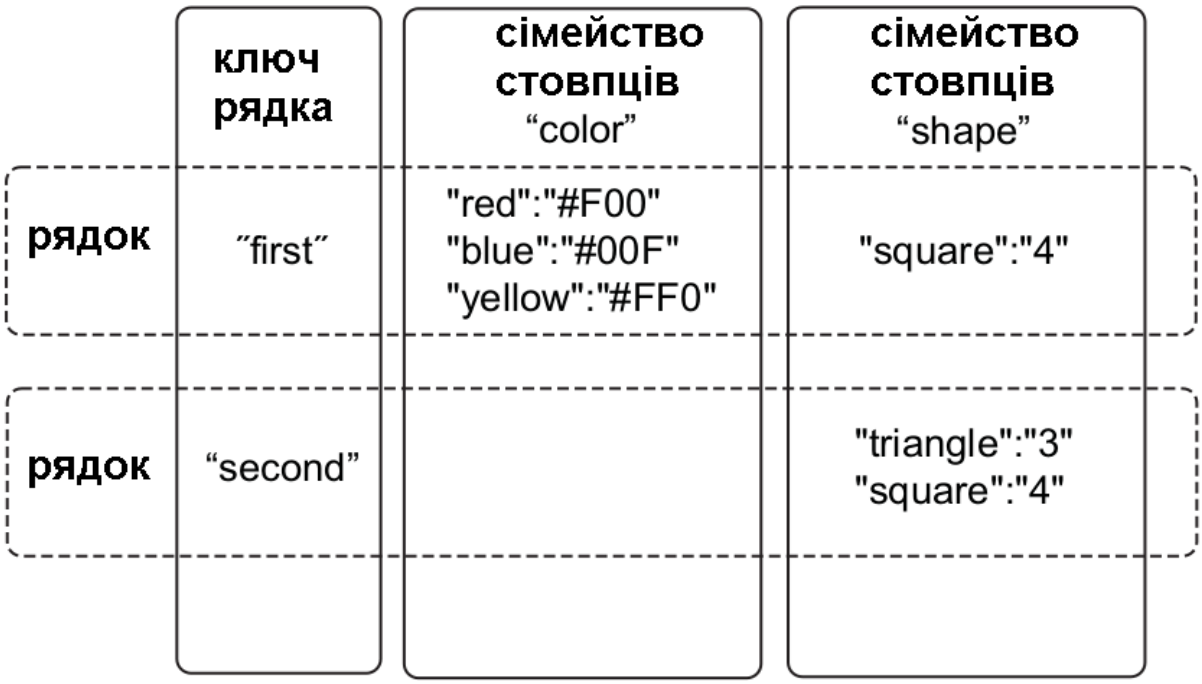


Рисунок 5.15. Структура таблиці

Створимо таблицю «*sensor_telemetry*» і доповнимо сімейством стовпців «*metrics*». Оператор *create* створює таблицю.

create 'sensor_telemetry', 'metrics'

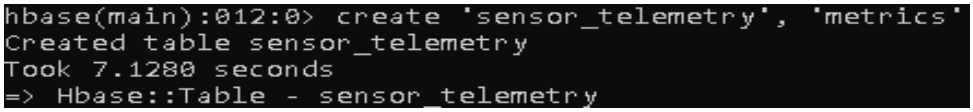


Рисунок 5.16. Створено таблицю

За таблицею створюється індекс, котрий міститься в файлі HFile службового серверу (див. рис. 5.13 й рис. 5.14). Зчитування значень індексу реалізовано на базі метода обробки даних, розробленого Бартоном Блумом.

Метод Блума

Обробка даних виконується на базі бітової послідовності, в котрій всі елементи спочатку дорівнюють 0. Кожен раз, коли метод застосовується до нового блоку даних, значення деяких бітів встановлюються в значення «1», яких конкретно, залежить від обчисленого хешу даних, котрому співставляються номери послідовності бітів.

Якщо пізніше знадобиться дізнатись, чи продивлялись конкретний блок даних, метод виконує розрахунок номерів в послідовності бітів даних і перевіряє рівність значенню «1»? Якщо єдиний з цих бітів дорівнює значенню «0», тоді метод повертає заперечення. Якщо всі елементи бітової послідовності дорівнюють значенню «1», буде повернуто повідомлення «ймовірно даний блок вже продивлялись».

При збільшенні кількості зчитаних блоків, підвищується ймовірність хибнопозитивної відповіді. Хибнопозитивна відповідь залежить від «рівня насиченості», котрий можна розрахувати.

Зчитаємо структуру таблиці sensor_telemetry.

describe 'sensor_telemetry'

```
hbase(main):013:0> describe 'sensor_telemetry'
Table sensor_telemetry is ENABLED
sensor_telemetry
COLUMN FAMILIES DESCRIPTION
{NAME => 'metrics', VERSIONS => '1', EVICT_BLOCKS_ON_CLOSE => 'false', NEW_VE
RSION_BEHAVIOR => 'false', KEEP_DELETED_CELLS => 'FALSE', CACHE_DATA_ON_WRITE
 => 'false', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', MIN_VERSIONS =>
 '0', REPLICATION_SCOPE => '0', BLOOMFILTER => 'ROW', CACHE_INDEX_ON_WRITE =>
 'false', IN_MEMORY => 'false', CACHE_BLOOMS_ON_WRITE => 'false', PREFETCH_BL
OCKS_ON_OPEN => 'false', COMPRESSION => 'NONE', BLOCKCACHE => 'true', BLOCKSI
ZE => '65536'}
1 row(s)
Took 1.0329 seconds
hbase(main):014:0>
```

Рисунок 5.17. Зчитано інформацію щодо таблиці

Параметр *BLOOMFILTER* допускає значення: *NONE*; *ROW* – зчитування наявності вказаного *ROW*-ключа; *ROWCOL* – зчитування наявності *ROW*-ключа і *COL*-ідентифікатора.

Оператор *put* дозаписує дані в таблицю. Наприклад, дозапишемо значення '65' за ключем '/94555/20170308/18:30' і стовпецем *temperature*.

put 'sensor_telemetry', '/94555/20170308/18:30', 'temperature', '65'

```
hbase(main):002:0> put 'sensor_telemetry', '/94555/20170308/18:30', 'temperat
ure', '65'

ERROR: org.apache.hadoop.hbase.regionserver.NoSuchColumnFamilyException: Colu
mn family temperature does not exist in region sensor_telemetry,,160037759834
1.d1f542230fceff06c0198c54ce71f123. in table 'sensor_telemetry', {NAME => 'me
trics', VERSIONS => '1', EVICT_BLOCKS_ON_CLOSE => 'false', NEW_VERSION_BEHAVI
OR => 'false', KEEP_DELETED_CELLS => 'FALSE', CACHE_DATA_ON_WRITE => 'false',
 DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', MIN_VERSIONS => '0', REPLIC
ATION_SCOPE => '0', BLOOMFILTER => 'ROW', CACHE_INDEX_ON_WRITE => 'false', IN
_MEMORY => 'false', CACHE_BLOOMS_ON_WRITE => 'false', PREFETCH_BLOCKS_ON_OPEN
 => 'false', COMPRESSION => 'NONE', BLOCKCACHE => 'true', BLOCKSIZE => '65536
'}
```

Рисунок 5.18. Відхиллено в створенні екземпляру даних
без зазначення сімейства стовпців

Запит дозаписування/зчитування має містити іменування таблиці, ключ рядка, сімейство стовпців і ідентифікатор стовпця.

put 'sensor_telemetry', '/94555/20170308/18:30', 'metrics:temperature', '65'

```
hbase(main):005:0> put 'sensor_telemetry', '/94555/20170308/18:30', 'metrics:
temperature', '65'
Took 1.5680 seconds
hbase(main):006:0>
```

Рисунок 5.19. Дозаписано дані в таблицю

Зчитаємо кількість екземплярів даних таблиці.

count 'sensor_telemetry'

```
hbase(main):006:0> count 'sensor_telemetry'
1 row(s)
Took 1.6782 seconds
=> 1
hbase(main):007:0>
```

Рисунок 5.20. Зчитано кількість екземплярів даних таблиці

Зчитаємо вміст таблиці.

scan 'sensor_telemetry'

```
hbase(main):007:0> scan 'sensor_telemetry'
ROW                 COLUMN+CELL
 /94555/20170308/18: column=metrics:temperature, timestamp=1600377966898, val
 30                 ue=65
1 row(s)
Took 0.4612 seconds
hbase(main):008:0>
```

Рисунок 5.21. Зчитано вміст таблиці

Відповідь СБД містить значення ключа рядка, іменування сімейства стовпців, ідентифікатор стовпця, часову позначку оновлення БД в мілісекундах і значення комірки.

В доповнення до команди *scan* – зчитування всіх екземплярів даних, в HBase реалізовано оператор зчитування екземпляра за ключем рядка.

get 'sensor_telemetry', '/94555/20170308/18:30'

```
hbase(main):008:0> get 'sensor_telemetry', '/94555/20170308/18:30'
COLUMN                CELL
 metrics:temperature timestamp=1600377966898, value=65
1 row(s)
Took 0.9613 seconds
hbase(main):009:0>
```

Рисунок 5.22. Зчитано за ключем вміст рядка

Дозапишемо в базу даних значення 55 для того ж ключа і ідентифікатора стовпця.

put 'sensor_telemetry', '/94555/20170308/18:30', 'metrics:temperature', '55'

```
hbase(main):009:0> put 'sensor_telemetry', '/94555/20170308/18:30', 'metrics:
temperature', '55'
Took 0.4533 seconds
hbase(main):010:0>
```

Рисунок 5.23. Оновлено значення комірки таблиці

Замість перезаписування, HBase не відображає попередні версії рядка. Зчитаємо значення рядка за ключем.

get 'sensor_telemetry', '/94555/20170308/18:30'

```
hbase(main):010:0> get 'sensor_telemetry', '/94555/20170308/18:30'
COLUMN                CELL
 metrics:temperature timestamp=1600378092201, value=55
1 row(s)
Took 0.5337 seconds
hbase(main):011:0>
```

Рисунок 5.24. Зчитано оновлене значення комірки

Оператор *put* в HBase еквівалентний до запиту *upsert* реляційних СБД.

Для зчитування конкретної версії комірки, в запиті *get* зазначаються параметри – іменування сімейства стовпців, ключ рядка, ідентифікатор стовпця, і, що важливо, коректне значення часової позначки.

get 'sensor_telemetry', '/94555/20170308/18:30', {COLUMN =>'metrics:temperature', TIMESTAMP => 1501810397402}

```
hbase(main):011:0> get 'sensor_telemetry', '/94555/20170308/18:30', {COLUMN =
>'metrics:temperature', TIMESTAMP => 1501810397402}
COLUMN                CELL
0 row(s)
Took 0.3801 seconds
hbase(main):012:0>
```

Рисунок 5.25. Не зчитано комірку за помилковим значенням часової позначки

Дозапишемо в таблицю декілька нових значень комірки.

put 'sensor_telemetry', '/94555/20170307/18:30', 'metrics:temperature', '43'

```
hbase(main):012:0> put 'sensor_telemetry', '/94555/20170307/18:30', 'metrics:
temperature', '43'
Took 0.4558 seconds
hbase(main):013:0>
```

Рисунок 5.26. Дозаписано в таблицю значення наявної комірки

put 'sensor_telemetry', '/94555/20170306/18:30', 'metrics:temperature', '33'

```
hbase(main):013:0> put 'sensor_telemetry', '/94555/20170306/18:30', 'metrics:
temperature', '33'
Took 0.2264 seconds
hbase(main):014:0>
```

Рисунок 5.27. Оновлено значення комірки

Зчитаємо вміст таблиці.

scan 'sensor_telemetry'

```
hbase(main):014:0> scan 'sensor_telemetry'
ROW                   COLUMN+CELL
 /94555/20170306/18:  column=metrics:temperature, timestamp=1600378279983, val
 30                   ue=33
 /94555/20170307/18:  column=metrics:temperature, timestamp=1600378258275, val
 30                   ue=43
 /94555/20170308/18:  column=metrics:temperature, timestamp=1600378092201, val
 30                   ue=55
3 row(s)
Took 0.3682 seconds
hbase(main):015:0>
```

Рисунок 5.28. Зчитано вміст таблиці

Для зчитування вмісту рядків таблиці за зменшенням значень часової позначки, в запиті зазначається параметр *REVERSED*.

scan 'sensor_telemetry', {REVERSED => true}

```
hbase(main):015:0> scan 'sensor_telemetry', {REVERSED => true}
ROW                   COLUMN+CELL
 /94555/20170308/18:  column=metrics:temperature, timestamp=1600378092201, val
 30                   ue=55
 /94555/20170307/18:  column=metrics:temperature, timestamp=1600378258275, val
 30                   ue=43
 /94555/20170306/18:  column=metrics:temperature, timestamp=1600378279983, val
 30                   ue=33
3 row(s)
Took 0.6605 seconds
hbase(main):016:0>
```

Рисунок 5.29. Зчитано вміст таблиці за зменшенням часової позначки

Для зчитування версій рядків зазначається параметр *VERSIONS*.

scan 'sensor_telemetry', {RAW => true, VERSIONS => 10}

```
hbase(main):016:0> scan 'sensor_telemetry', {RAW => true, VERSIONS => 10}
ROW                   COLUMN+CELL
 /94555/20170306/18: column=metrics:temperature, timestamp=1600378279983, val
 30                  ue=33
 /94555/20170307/18: column=metrics:temperature, timestamp=1600378258275, val
 30                  ue=43
 /94555/20170308/18: column=metrics:temperature, timestamp=1600378092201, val
 30                  ue=55
 /94555/20170308/18: column=metrics:temperature, timestamp=1600377966898, val
 30                  ue=65
3 row(s)
Took 0.4090 seconds
hbase(main):017:0>
```

Рисунок 5.30. Зчитано фіксовану кількість версій рядків

Для зчитування екземплярів даних за певним значенням ключа рядка, в запиті зазначається параметр *STARTROW*.

scan 'sensor_telemetry', {STARTROW => '/94555/20170307'}

```
hbase(main):018:0> scan 'sensor_telemetry', {STARTROW => '/94555/20170307'}
ROW                   COLUMN+CELL
 /94555/20170307/18: column=metrics:temperature, timestamp=1600378258275, val
 30                  ue=43
 /94555/20170308/18: column=metrics:temperature, timestamp=1600378092201, val
 30                  ue=55
2 row(s)
Took 0.4657 seconds
hbase(main):019:0>
```

Рисунок 5.31. Зчитано рядки за значенням ключа рядка

В HBase реалізовано фільтрацію рядків за значенням ключа.

scan 'sensor_telemetry', {ROWPREFIXFILTER => '/94555/20170307'}

```
hbase(main):020:0> scan 'sensor_telemetry', {ROWPREFIXFILTER => '/94555/20170
307'}
ROW                   COLUMN+CELL
 /94555/20170307/18: column=metrics:temperature, timestamp=1600378258275, val
 30                  ue=43
1 row(s)
Took 0.7042 seconds
hbase(main):021:0>
```

Рисунок 5.32. Виконано фільтрацію рядків за значенням ключа

Для застосування сторонніх методів фільтрації, необхідно підключити бібліотеки програм:

import org.apache.hadoop.hbase.util.Bytes;

import org.apache.hadoop.hbase.filter.SingleColumnValueFilter;

import org.apache.hadoop.hbase.filter.BinaryComparator;

import org.apache.hadoop.hbase.filter.CompareFilter;

import org.apache.hadoop.hbase.filter.Filter;

```
hbase(main):044:0> import org.apache.hadoop.hbase.util.Bytes;
hbase(main):045:0* import org.apache.hadoop.hbase.filter.SingleColumnValueFil
ter;
hbase(main):046:0* import org.apache.hadoop.hbase.filter.BinaryComparator;
hbase(main):047:0* import org.apache.hadoop.hbase.filter.CompareFilter;
hbase(main):048:0* import org.apache.hadoop.hbase.filter. Filter;
hbase(main):049:0*
```

Рисунок 5.33. Підключено бібліотеки програм фільтрації

Програма фільтрації SingleColumnValueFilter зчитує рядок за ідентифікатором стовпця й значенням рядка.

scan 'sensor_telemetry', {FILTER =>
SingleColumnValueFilter.new(Bytes.toBytes('metrics'),

Bytes.toBytes('temperature'),

CompareFilter::CompareOp.valueOf('EQUAL'),

BinaryComparator.new(Bytes.toBytes('55')))}

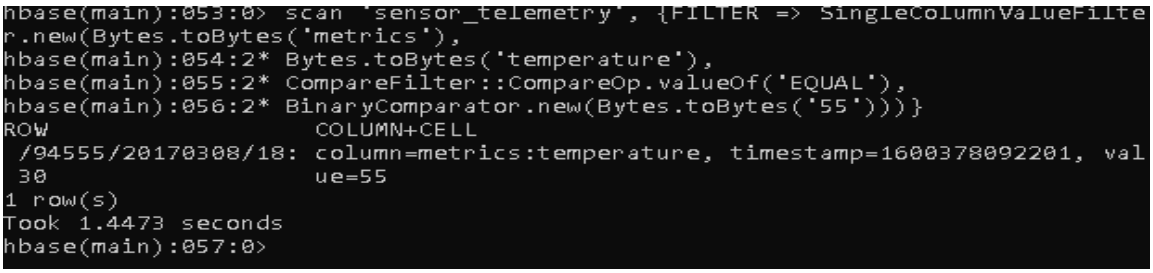


Рисунок 5.34. Виконано фільтрацію екземпляра даних за значенням рядка

Оновлення параметрів сімейства стовпців

Оновлення параметрів сімейства стовпців потребує тривалого часу, так як HBase створить нове сімейство, зупинить обробку запитів користувачів, перемістить дані. Тому, вже при проектуванні бази даних, необхідно коректно описати сімейство стовпців.

Оператор *alter* оновлює сімейство стовпців (див. рис. 5.36).

alter 'sensor_telemetry', {NAME => 'metrics', BLOCKSIZE => '16384', COMPRESSION => 'GZ'}

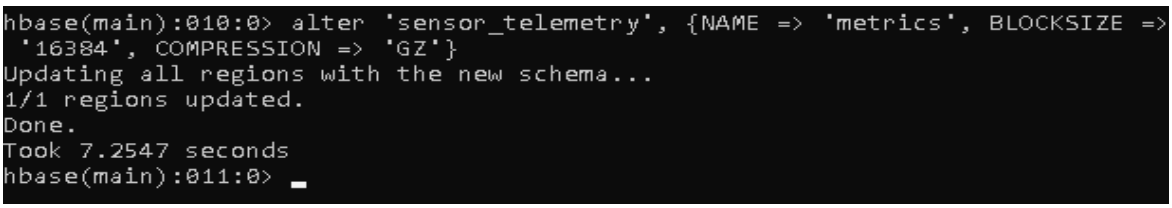


Рисунок 5.35. Оновлено параметри сімейства стовпців

Виконуємо закриття консолі і зупиняємо сервер СБД.

exit

$HBase_HOME/bin/stop-hbase.sh

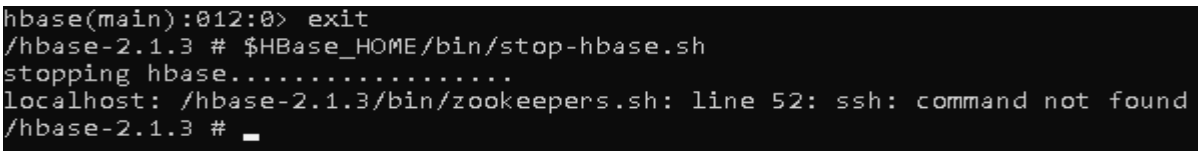


Рисунок 5.36. Зупинено виконання сервера

Мережеві налаштування

Якщо СБД не забезпечує відповідь на запит, тоді, ймовірно, справа в мережевому підключенні. HBase підтримує автоматичне конфігурування мережевих налаштувань.

Конфігураційні параметри зберігаються в файлі hbase-site.xml в каталозі *$HBase_HOME/conf/*, де *$HBase_HOME* – змінна ОС. Спочатку файл містить тег <configuration>, котрий далі доповнюється властивостями за шаблоном:

<property>
<name>some.property.name</name>
<value>A property value</value>
</property>

Для підключень до кластеру з локальної мережі, має сенс зазначити в файлі HBase-site.xml перераховані нижче параметри і значення.

Таблиця 5.2. Мережеві налаштування кластеру

Параметр	**Значення**
HBase.master.dns.interface	lo
HBase.master.info.bindAddress	127.0.0.1
HBase.RegionServer.info.bindAddress	127.0.0.1
HBase.RegionServer.dns.interface	lo
HBase.Zookeeper.dns.interface	lo

Мережеві налаштування вказують HBase як встановлювати з'єднання з керуючим сервером, серверами регіонів і координатором Zookeeper. Значення

«lo» позначає «зворотній» інтерфейс, котрий реалізовано програмно для перевірки з'єднання.

Вимикаємо контейнер.

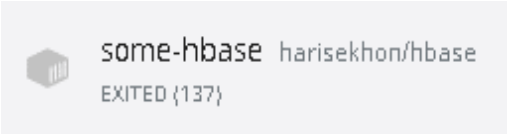


Рисунок 5.37. Вимкнено контейнер

5.4. Висновки

Виконано запуск кластеру HBase. Розібрано приклад моделювання даних, реєстрованих датчиком температури. Створено тестову таблицю і оновлено параметри сімейства стовпців.

5.4.1. Теоретичне завдання

1. Як в консолі HBase виконати:
   - видалення з таблиці комірок, котрі містять тестове значення;
   - видалення з таблиці рядків, котрі містять тестове значення.
2. Встановіть в браузері закладку документації API HBase.

5.4.2. Практичне завдання

Встановіть бібліотеку мови програмування за вашим вибором і реалізуйте функції:

- створення тестових двох таблиць і двох сімейств стовпців;
- дозаписування даних в дві таблиці;
- зчитування даних з двох таблиць;
- оновлення даних в двох таблицях;
- видалення даних з двох таблиць;
- видалення таблиць.

5.4.3. Питання для закріплення знань

1. Випадки, коли рекомендовано застосовувати стовпцеві СБД.
2. Випадки, коли не рекомендовано застосовувати стовпцеві СБД.
3. Групування стовпців в СБД HBase. Розрахунок кількості HFile за кількістю сімейств стовпців БД.
4. Назначення серверу регіону в кластері HBase?
5. Складові архітектури кластеру HBase. Яку функцію виконують керуючий сервер, сервери регіонів?
6. Відмінність DataNode і NameNode СБД HBase. Яку функцію виконує HDFS, клієнтська утиліта HDFS?
7. Яку функцію виконує координатор Zookeeper в кластері HBase?

Список використаних джерел

[1] Guy Harrison. Next Generation Databases. NoSQL, NewSQL, and Big Data. – New York: Apress, 2015. – 235 p.

[2] Edgar F. Codd. A relational model of data for large shared data banks / Communications of the ACM. – vol. 13, no. 6. – New York: ACM, 1970. – p. 377-387.

[3] Mike Stonebraker et al. The implementation and design of INGRES / ACM Transactions on Database Systems. – vol. 1, no. 3. – New York: ACM, 1976. – p. 189-222.

[4] Edgar F. Codd. The Relational Model for Database Management. Ver. 2. – Boston: Addison-Wesley, 1990. – 538 p.

[5] Chris J. Date. A Guide to the SQL Standard, 2nd ed. – Boston: Addison-Wesley, 1989. – 228 p.

[6] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung. The Google file system / Proceedings of the nineteenth ACM symposium on Operating systems principles. – New York: ACM, 2003. – p. 29-43.

[7] Dean Jeffrey, Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. – Berkeley: USENIX, 2004. – p. 137-149.

[8] Chang Fay et al. Bigtable: A Distributed Storage System for Structured Data. – Berkley: USENIX, 2006. – p. 205-218.

[9] Andrzej Bialecki, Mike Cafarella, Doug Cutting, Owen O`Malley. Hadoop: a framework for running applications on large clusters built of commodity hardware. – vol. 11. – Wakefield: Apache Lucene, 2005. – p. 1-5.

[10] Ankur Khetrapal, Vinay Ganesh. HBase and Hypertable for large scale distributed storage systems / Dept. of Computer Science. – vol. 10. – Indianapolis: Purdue University, 2006. – p. 1-8.

[11] Salvatore Sanfilippo. Redis key-value store. – WWW: redis.io, 2009.

[12] Swaminathan Sivasubramanian. Amazon DynamoDB a seamlessly scalable non-relational database service / ACM SIGMOD International Conference on Management of Data. – Scottsdale: ACM, 2012. – pp. 729-730.

[13] Олександр Євгенович Стрижак, Лариса Сергіївна Глоба, Віталій Юрійович Величко, Катерина Яківна Климова, Олена Борисівна Комова та ін. Програмно-інформаційні засоби формування систем знань навчального призначення: посібник / за ред. О. Є. Стрижака. – Київ: Інститут обдарованої дитини, 2014. – 144 с.

[14] Лариса Сергіївна Глоба, Максим Юрійович Терновой, Ріна Леонідівна Новогрудська, Олена Сергіївна Штогріна. Створення та обробка баз даних: навч. посібник для студ. техн. спец. вищ. навч. закл. – Київ: НТУУ «Київський політехнічний інститут», 2013. – 477 с.

[15] Андрій Юліанович Берко, Олег Михайлович Верес, Володимир Володимирович Пасічник. Системи баз даних та знань. Кн. 1. Організація баз даних та знань. – Львів: Магнолія, 2016. – 438 с.

[16] Андрій Юліанович Берко, Олег Михайлович Верес, Володимир Володимирович Пасічник. Системи баз даних та знань. Кн. 2. Системи управління базами даних та знань. – Львів: Магнолія, 2016. – 583 с.

[17] Володимир Володимирович Пасічник, Валерій Анатолійович Резніченко. Організація баз даних та знань. – Київ: BHV, 2006. – 383 с.

[18] Олександр Никифорович Романюк, Тамара Олександрівна Савчук. Організація баз даних і знань. – Вінниця: Універсум, 2003. – 217 с.

[19] Эрик Редмонд, Джим. Р. Уилсон. Семь баз данных за семь недель. Введение в современные базы данных и идеологию NoSQL / Пер. с англ. Слинкин А. А. – Москва: ДМК Пресс, 2013. – 384 с.

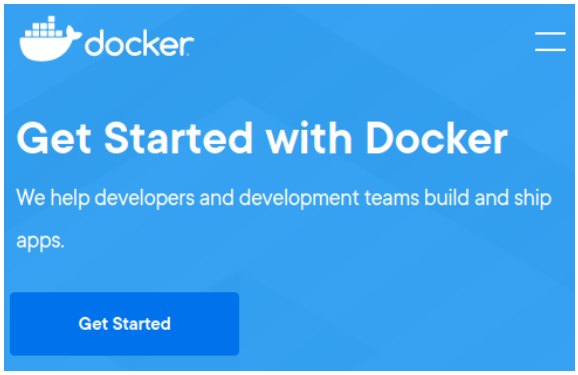
[20] Юрий Павлович Парфенов. Постреляционные хранилища данных. – Екатеринбург: Уральский университет, 2016. – 120 с.

[21] Luc Perkins, Eric Redmond, Jim R. Wilson. Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement, 2nd ed. – Raleigh: Pragmatic Bookshelf, 2018. – 360 p.

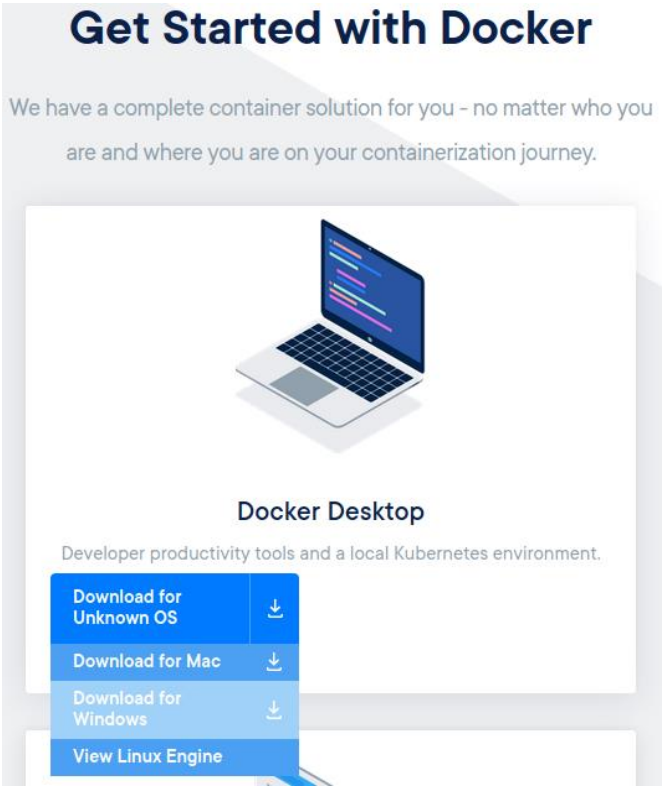
[22] Aaron Ploetz, Devram Kandhare, Sudarshan Kadambi, Xun Wu. Seven NoSQL Databases in a Week: Get up and running with the fundamentals and functionalities of seven of the most popular NoSQL databases. – Birmingham: Packt Publishing, 2018. – 308 p.

Додаток А. Встановлення платформи Docker

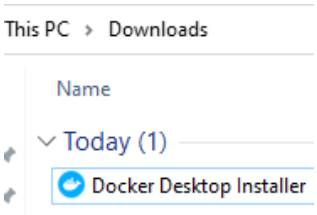
1. Дивимось в Веб-браузері офіційну сторінку розробників платформи за посиланням – *www.docker.com* .



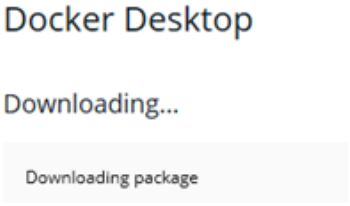
2. Курсором переміщуємось за посиланням *Get Started* і виконуємо завантаження програми-інсталятора для ОС Windows.



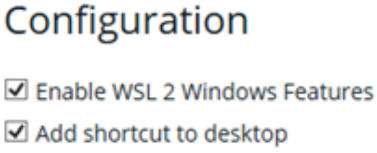
3. Виконуємо запуск файлу встановлення платформи.



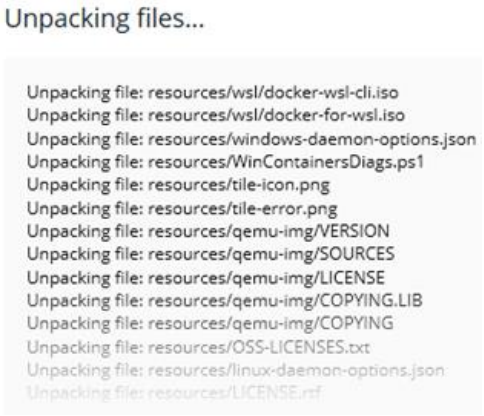
4. В процесі встановлення виконується аналіз необхідних програмних пакетів.



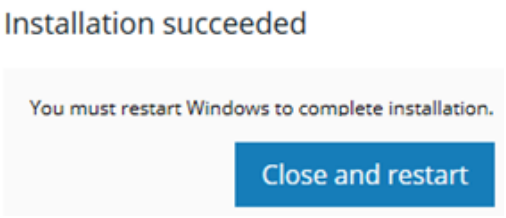
5. Пропонується обрати параметри конфігурації.



6. Виконується процес встановлення.

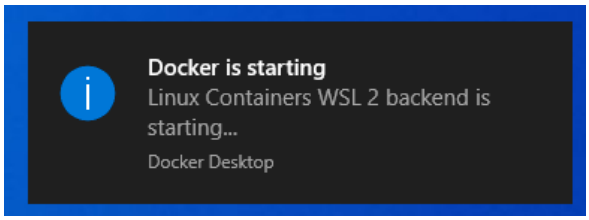


7. Встановлення завершено.

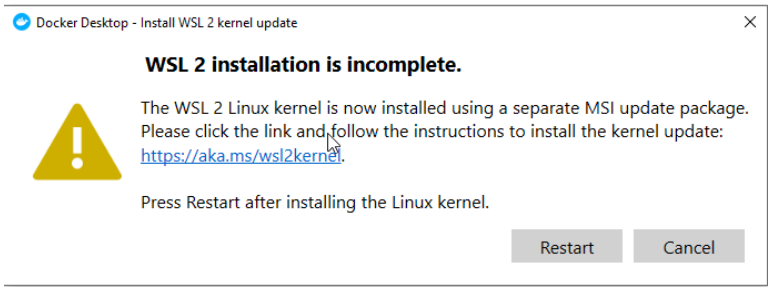


8. Виконуємо запуск Docker з «Робочого стола» ОС Windows.

9. Windows інформує щодо запуску платформи.

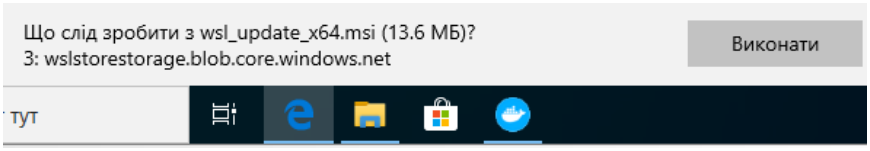


10. Ймовірно з'явиться повідомлення необхідності додаткового встановлення компоненту ОС – «Windows Subsystem for Linux» (WSL 2).

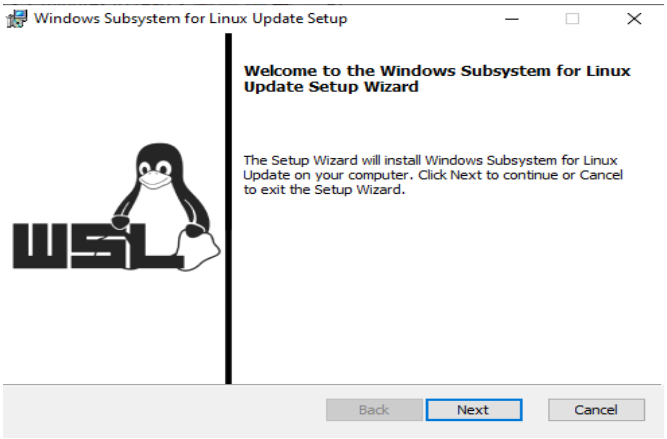


11. Виконуємо завантаження програми інсталятора за посиланням – *https://aka.ms/wsl2kernel* .

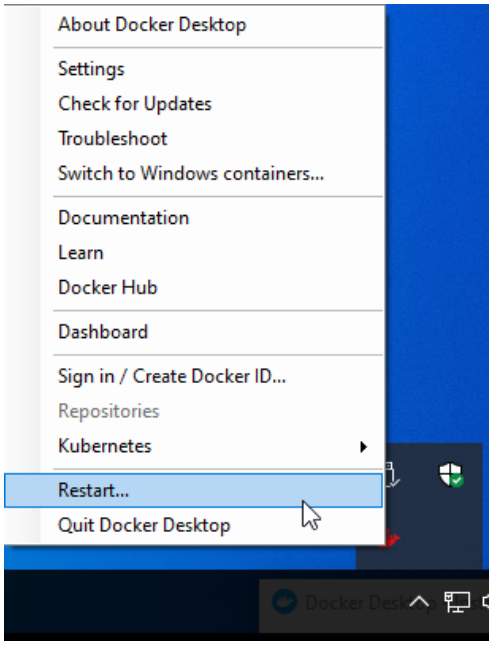
12. Виконуємо запуск програми інсталятора.



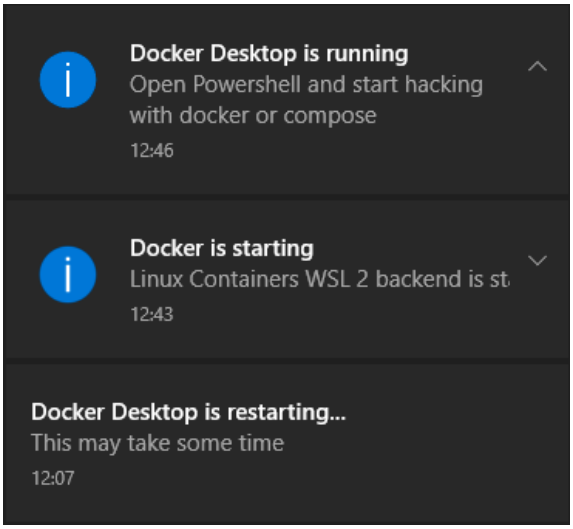
13. Встановлюємо компонент WSL.



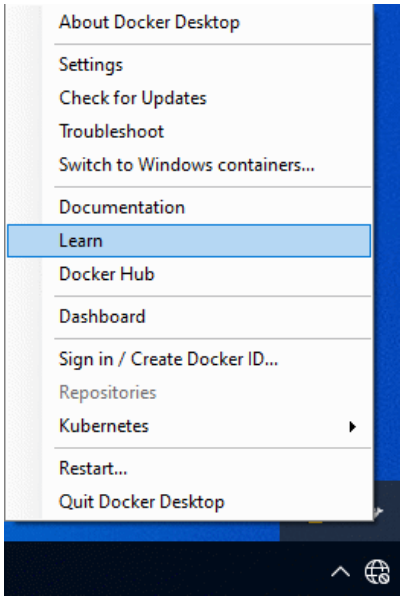
14. Виконуємо перезавантаження Docker з Windows System Tray.



15. Windows відобразить інформацію щодо перезапуску, запуску і виконання Docker.



16. Виконуємо запуск інтерфейсу Learn з Windows System Tray.



17. В підрозділі Start інтерфейсу Learn встановлюємо СБД за розділами посібника.