

ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
Київського національного університету
імені Тараса Шевченка

Кафедра програмних систем і технологій

ДУХНОВСЬКА К. К., ЮРЧУК І. А., НІКОЛАЄНКО А. Ю.

СУЧАСНІ БАЗИ ДАНИХ

Підручник

Частина І

Київ – 2025

РЕЦЕНЗЕНТИ:

канд. техн. наук, доцент **Артамонов Є. Б.** – доцент кафедри інтелектуальних кібернетичних систем управління Державного некомерційного підприємства «Державний університет «Київський авіаційний інститут»,

канд. ф.-м. наук, доцент **Перегуда О. В.** – доцент кафедри загальної математики Київського національного університету імені Тараса Шевченка.

канд. техн. наук, доцент **Лещенко О. О.** – доцент кафедри мережевих та інтернет технологій Київського національного університету імені Тараса Шевченка.

Рекомендовано до публікації кафедрою програмних систем і технологій Київського національного університету імені Тараса Шевченка в електронному вигляді, протокол № 14 від 18 березня 2025 р.

Духновська К. К., Юрчук І. А., Ніколаєнко А. Ю.

Сучасні бази даних Частина I [Електронний ресурс]: підручник. Кафедра програмних систем і технологій / **Духновська К. К.** – 2025. – 391 с.

Підручник «Сучасні бази даних» частина I для студентів денної форми навчання зі спеціальності F2 – «Інженерія програмного забезпечення» за освітньо-науковою програмою «Інженерія програмного забезпечення» освітнього рівня «Магістр».

Публікується в авторській редакції.

© Духновська К. К., Юрчук І. А., Ніколаєнко А. Ю., 2025 р.

ЗМІСТ

ВСТУП.....	5
1. ОСНОВИ БАЗ ДАНИХ.....	7
1.1 Базы даних	7
1.2 Реляційні бази даних	10
1.3 SQL і NOSQL бази даних	19
1.4 Документоорієнтовані бази даних.....	20
1.5 Базы даних «ключ-значення» (Key-Value).....	22
2. РЕЛЯЦІЙНІ БАЗИ ДАНИХ	26
2.1 Керування і інструменти реляційних баз даних	26
2.2 Створення, структура та основні елементи баз даних SQL Server	35
2.3 Вибірка, фільтрація та зміна даних	98
2.4 Зв'язування таблиць та отримання зв'язаних даних	125
2.5 Управління безпекою та використання функцій	138
2.6 Програмування та розширені об'єкти бази даних.....	163
2.7 Інтеграція прикладних програм із sql-базами даних	193
3. ДОКУМЕНТООРІЄНТОВАНІ БАЗИ ДАНИХ	229
3.1 Вступ до MongoDB та робота з документами	229
3.3 Оновлення даних. Видалення даних	264
3.4 Інструменти, моделювання та індексація даних у MongoDB.....	274
3.5 Агрегація	291
3.6 C# та MongoDB.....	305
4. БАЗИ ДАНИХ «КЛЮЧ-ЗНАЧЕННЯ».....	329
4.1 Початкова інформація про СУБД Redis.....	329
4.2 Управління базами даних в Redis	335
4.3 Ключі та значення	336
4.4 Типи даних в Redis	341
4.5 Транзакції в Redis	380
4.6 Тривалість життя ключів в Redis	383

4.7 Взаємодія застосунків із СУБД Redis	384
ЛІТЕРАТУРНІ ДЖЕРЕЛА	388

ВСТУП

Навчальна дисципліна «SQL» входить до складу освітньо-професійної програми «Інженерія програмного забезпечення» для студентів освітнього рівня «Магістр» за спеціальністю 121 «Інженерія програмного забезпечення» галузі знань 12 «Інформаційні технології».

Метою дисципліни є формування знань, вмінь і навичок студентів щодо проектування та створення сучасного програмного забезпечення із використанням сучасних баз даних.

Завдання дисципліни полягає у формуванні у студента знань про сучасні системи управління базами даних. У результаті вивчення навчальної дисципліни студент повинен вміти вибирати відповідну систему управління базами даними для вирішення поставленої задачі. Студент повинен засвоїти різні типи запитів до сучасних баз даних, базові типи даних та функції, які визначені на них, принципи оптимізації запитів, вміти писати оператори для додавання та модифікації інформації в базі даних; на основі аналізу задачі, що розробляється, вибирати найбільш раціональні оператори запитів, що забезпечують ефективну реалізацію задачі. Студент має володіти знаннями та вміннями виконувати операції фільтрації структурованих та частково структурованих даних за умовою. У результаті вивчення дисципліни студент повинен вміти працювати зі структурованою і частково структурованою інформацією.

Дисципліна базується на знаннях, вміннях і практичних навичках, отриманих студентами при вивченні дисциплін «Математичні основи програмної інженерії», «Прикладна математика», «Комп'ютерні методи аналізу дискретних об'єктів», «Алгоритми та структури даних», «Об'єктно-орієнтоване конструювання програм», «SQL», «Проектування баз даних», «Структури даних, аналіз та алгоритми комп'ютерної обробки інформації».

Перша глава присвячена розгляду систем управління реляційними базами даних та інструментів для комфортної роботи з ними. У підручнику

розглядаються ключові аспекти сучасних реляційних баз даних, починаючи з основ, таких як моделі даних, таблиці, ключі та зв'язки. Детально вивчається мова SQL, включаючи синтаксис, запити, маніпулювання даними, транзакції, збережені процедури та тригери.

Друга глава детально знайомить з MongoDB, сучасною документо-орієнтованою NoSQL базою даних, яка здобула широку популярність завдяки своїй гнучкості, масштабованості та високій продуктивності. Тут розглядаються основні аспекти роботи з MongoDB, включаючи її архітектуру, моделі даних, колекції та документи. Вивчається мова запитів MongoDB, зокрема операції CRUD, агрегація, індекси. Описується розробка застосунків з використанням MongoDB.

Третя глава присвячена Redis, сучасною базою даних типу ключ-значення, що вирізняється високою продуктивністю та гнучкістю, і набула широкої популярності. Вона охоплює фундаментальні та розширені аспекти Redis, починаючи з основ, таких як архітектура, типи даних, команди та конфігурація. Особливу увагу приділено структурам даних Redis, включаючи рядки, списки, множини, хеші, сортовані множини та потоки. Розглядаються розширені можливості Redis, такі як транзакції, публікація/підписка та геопросторові індекси. Також вивчається використання Redis для кешування, включаючи стратегії кешування, термін дії та витіснення даних, а також для обробки повідомлень, включаючи черги, брокери повідомлень та системи реального часу. Нарешті, наведені практичні приклади використання Redis для розробки веб-додатків, аналітики в реальному часі та систем рекомендацій.

Підручник містить численні приклади коду, вправи та практичні завдання.

1. ОСНОВИ БАЗ ДАНИХ

1.1 Бази даних

Термін дані походить від слова data – факт. Дані – це сукупність відомостей, зафіксованих на певному носіїві у формі, придатній для постійного зберігання, передачі і обробки. Дані обробляються чи то людьми, чи інформаційними системами, і як результат, отримується інформація. Тобто, інформація – це результат перетворення і аналізу даних. Інформація (informatio) означає роз'яснення, виклад, тобто відомості або повідомлення. Відмінність інформації від даних полягає в тому, що дані – це фіксовані відомості про події і явища, які зберігаються на певних носіях, а інформація з'являється в результаті обробки даних при рішенні конкретних задач. Знання – це зафіксована і перевірена практикою оброблена інформація, яка використовувалася і може багато разів використовуватися для прийняття рішень.

Дані є фундаментальним компонентом сучасних інформаційних систем. Від простих веб-застосунків до складних корпоративних рішень, дані слугують основою для зберігання, обробки та аналізу інформації. Будь то коментарі до мультимедійного контенту, записи про користувацьку активність в онлайн-іграх або деталі транзакцій електронної комерції, вся ця інформація систематизується та зберігається в спеціалізованих сховищах даних.

Розділення даних і програм – це фундаментальний принцип в інформатиці. Коли дані та програми зберігаються окремо, їх можна змінювати незалежно один від одного. Це дає змогу оновлювати програму без ризику пошкодження даних і навпаки. Якщо обсяг даних зростає, можна легко збільшити потужність системи зберігання даних, не змінюючи саму програму. Різні програми можуть працювати з одними і тими самими даними, якщо вони збережені в стандартних форматах. Зберігаючи дані окремо, можна впровадити додаткові заходи безпеки для їх захисту від несанкціонованого доступу. Якщо

програма працює неправильно або виникає збій, дані залишаються неушкодженими. Різні програми можуть використовувати одні і ті ж дані, що дає змогу інтегрувати різні системи та автоматизувати процеси. Дані можуть бути оброблені різними аналітичними інструментами для отримання нової інформації. Розробники можуть зосередитися на створенні логіки програми, не турбуючись про деталі зберігання даних. Розділення даних і програми робить код більш зрозумілим і легким для підтримки.

Термін «база даних» має широкий спектр трактувань. В контексті цього курсу під базою даних розумітимемо структуровану сукупність даних, організовану за певними правилами та призначену для ефективного зберігання, пошуку та оновлення інформації. Прикладами таких даних можуть бути персональні дані клієнтів, товарні номенклатури, фінансові показники тощо. Важливо зазначити, що для управління базою даних, тобто для виконання операцій з її даними (вставки, видалення, модифікації, пошуку), використовується спеціалізоване програмне забезпечення – система управління базами даних (СУБД). Таким чином, поняття «база даних» охоплює як самі дані, так і засоби їх обробки.

У процесі еволюції систем управління базами даних сформувався ряд фундаментальних вимог. Виникла необхідність у створенні централізованого сховища даних, яке б служило єдиним авторитетним джерелом інформації для різних програмних застосунків. Такий підхід дає змогу забезпечити узгодженість і актуальність даних, усунути дублювання та підвищити ефективність роботи з інформацією. Зростаюча складність інформаційних систем висунула вимогу до забезпечення високого рівня надійності зберігання даних. Механізми резервного копіювання, відновлення та контролю цілісності стали невід’ємною частиною сучасних СУБД. Крім того, виникла потреба в гарантуванні логічної цілісності даних, тобто забезпечення відповідності даних встановленим обмеженням і правилам. Збільшення масштабу використання баз даних призвело до необхідності впровадження механізмів контролю доступу. Це дає змогу обмежити доступ до даних лише авторизованим користувачам і

забезпечити конфіденційність інформації. Сучасні СУБД підтримують різноманітні моделі доступу, які дозволяють надавати користувачам різні рівні дозволів залежно від їхніх ролей і завдань.

СУБД є складним програмним комплексом, призначеним для ефективного створення, зберігання, оновлення та доступу до структурованих даних. СУБД забезпечує абстракцію даних, відділяючи логічне подання інформації від фізичного способу її зберігання. Це дає змогу прикладним програмам працювати з даними на високому рівні абстракції, не заглиблюючись у деталі їхньої внутрішньої організації.

Основні функціональні можливості СУБД включають:

- надання користувачам мов і засобів для опису структури даних та встановлення між ними семантичних зв'язків;
- організація ефективного зберігання великих обсягів даних на зовнішніх носіях;
- забезпечення механізмів для вибірки, оновлення та видалення даних за допомогою мов запитів;
- гарантування логічної цілісності даних шляхом накладання обмежень на їх значення та взаємозв'язки;
- реалізація механізмів авторизації та аутентифікації для захисту даних від несанкціонованого доступу;
- забезпечення атомарності, узгодженості, ізоляції та стійкості транзакцій для підтримки цілісності даних у багатокористувацькому середовищі;
- ведення журналу змін даних для відновлення після збоїв та аудиту;
- автоматична оптимізація виконання запитів для підвищення продуктивності системи.

Однією з перших і найбільш широко використовуваних мережових СУБД стала Integrated Data Store (IDS), розроблена ще в 60-х роках минулого століття компанією General Electric, IDS відіграла значну роль у розвитку технологій баз даних. IDS була розроблена з акцентом на швидкість обробки великих обсягів

даних, що було особливо важливо в той час, коли обчислювальні ресурси були обмеженими. IDS використовувала мережеву модель даних, яка дозволяла встановлювати складні зв'язки між різними типами даних. IDS стала фундаментом для подальших розробок у сфері систем управління базами даних. Чарльз Бахман, творець IDS, отримав Премію Тюрінга за свій внесок у цю галузь. IDS була першою серйозною спробою створити систему для ефективного управління великими обсягами даних. Багато концепцій і технологій, розроблених для IDS, стали основою для сучасних систем управління базами даних. IDS показала, що навіть на обмеженому засобі того часу можна створити потужну систему для роботи з даними. Ця СУБД була досить складною в освоєнні і використанні, що обмежувало коло її застосування. Також, IDS була тісно інтегрована з операційною системою GE-235, що ускладнювало її портативність. З появою реляційних баз даних, таких як Oracle і SQL Server, мережева модель IDS стала менш популярною. Хоча IDS і відійшла на другий план, її вплив на розвиток технологій баз даних залишається значним. Багато ідей, закладених в IDS, продовжують використовуватися і сьогодні.

Питання для самоперевірки

1. Що таке дані, інформація, знання?
2. Що таке база даних?
3. Які вимоги висуваються до сучасних баз даних?
4. Що таке СУБД?
5. Яка функціональність СУБД?

1.2 Реляційні бази даних

Реляційна модель баз даних є концептуальним базисом для організації та управління даними, де інформація представлена у вигляді таблиць, що містять

стовпці (атрибути) та рядки (кортежі). Кожна таблиця відображає сутність або відношення між сутностями, а операції над даними зводяться до маніпуляцій з цими таблицями за допомогою спеціалізованих запитів.

Основними поняттями реляційних баз даних є: відношення (таблиця), первинний ключ, кортеж, атрибут, домен (рис. 1.1).

Кортеж – це окремий екземпляр сутності, що відповідає рядку таблиці.

Відношення – це множина кортежів з однаковою структурою, що представляє собою таблицю бази даних.

Атрибутом називається властивість сутності, що описується стовпцем таблиці.

Доменом є множина допустимих значень для атрибута певного типу. Як видно із рис. 1.1, допустимими значення для стовпчика, у якому вказується курс студента, буде від 1 до 4, а для стовпчика середній бал – це від 60 до 100.

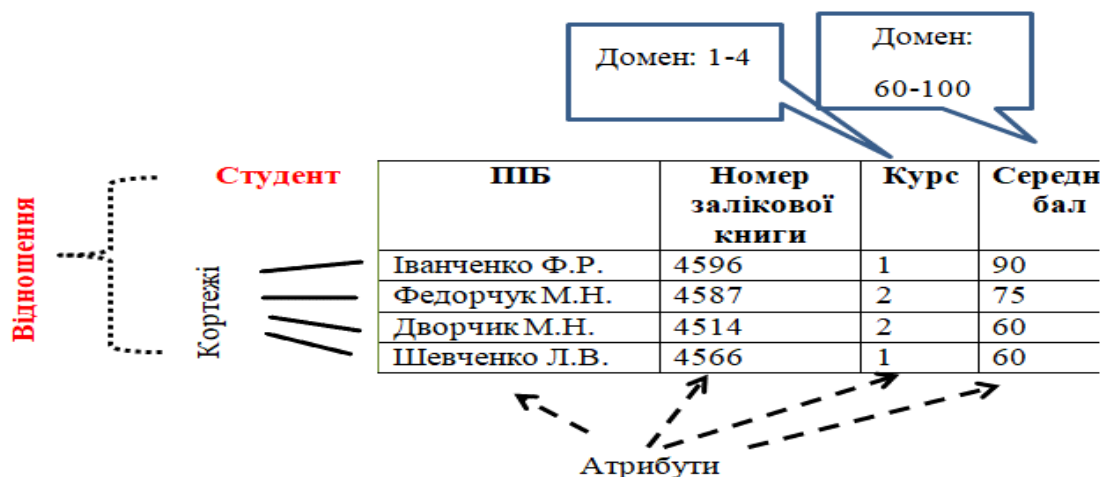


Рисунок 1.1 – Основні поняття реляційних баз даних

Кількість атрибутів у відношенні називається степенем відношення, а кількість кортежів – кардинальністю.

Між цими поняттями існує чітка взаємозалежність, яка визначає структуру та функціональність реляційної моделі даних. Співвідношення між цими поняттями може бути наглядно проілюстровано за допомогою діаграм або схем, що демонструють зв'язки між таблицями та їх атрибутами (рис. 1.2).

На таблиці, які є логічним представленням відношень у теорії реляційних баз даних накладаються такі обмеження:

- відсутність кортежів-дублікатів;
- атомарність значень атрибутів: відсутність повторюваних структур даних;
- довільність порядку кортежів відношення;
- унікальність імен атрибутів відношення.

Для унікальної ідентифікації кожного кортежу відношення розглядають поняття реляційного ключа: це один або кілька атрибутів (стовпців) таблиці, які використовуються для ідентифікації кожного окремого кортежу (рядка) в таблиці. Ключі відіграють важливу роль у забезпеченні цілісності та організації даних. У реляційній БД може існувати декілька видів ключів: потенційні (Candidate Key, СК), первинні (Primary Key, РК), альтернативні (Alternate Key, АК), зовнішні (Foreign Key, FK).

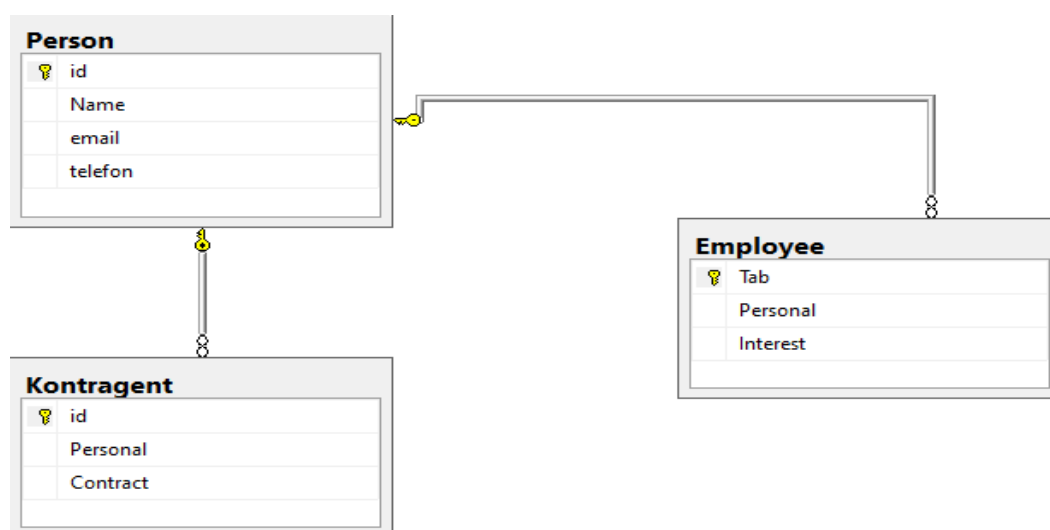


Рисунок 1.2 – Діаграма реляційної бази даних

Потенційний ключ – це атрибут чи набір атрибутів, який можна використати для унікальної ідентифікації кортежів таблиці. Ключ, що складається з декількох атрибутів, називається складним ключем. У таблиці може бути декілька потенційних ключів.

Наприклад, маємо таблицю «Студенти», тут потенційними ключами можуть бути: номер студентського квитка, номер залікової книги, серія і номер

паспорта, електронна скринька. Кожен студент має паспорт, студентський квиток, залікову книжку з унікальними номерами. У даному випадку, кожен з цих атрибутів або їх комбінація може слугувати для унікальної ідентифікації студента, тому вони є потенційними ключами.

Первинний ключ – це особлива форма потенційного ключа. У таблиці може існувати лише один первинний ключ, який вибирається серед потенційних. У реляційних базах даних первинний ключ – це один або кілька атрибутів (стовпців) таблиці, які однозначно ідентифікують кожен окремий запис (кортеж) у цій таблиці.

Значення первинного ключа не можуть повторюватися в межах однієї таблиці. Кожен запис повинен мати унікальне значення первинного ключа, що дає змогу однозначно його ідентифікувати. Первинний ключ не може містити порожні значення. Кожен кортеж повинен мати значення первинного ключа. Первинний ключ повинен складатися з мінімальної кількості атрибутів, необхідних для забезпечення унікальності кожного запису. Значення первинного ключа, як правило, не повинні змінюватися з часом. Зміна первинного ключа може призвести до порушення зв'язків між таблицями та втрати даних.

Первинний ключ служить для однозначної ідентифікації кожного окремого запису в таблиці. Первинний ключ однієї таблиці може використовуватися як зовнішній ключ в іншій таблиці для встановлення зв'язку між ними. Первинний ключ забезпечує цілісність даних, запобігаючи дублюванню записів та забезпечуючи коректність зв'язків між таблицями.

У таблиці «Клієнти» первинним ключем може бути ідентифікаційний номер клієнта (ID клієнта). У таблиці «Товари» первинним ключем може бути артикул товару. У таблиці «Замовлення» первинним ключем може бути номер замовлення. У таблиці «Студент», із прикладу вище – один із потенційних ключів.

Кожна таблиця в реляційній базі даних може мати лише один первинний ключ. Первинний ключ може складатися з одного або кількох атрибутів

(складений ключ). Первинний ключ є важливим елементом проєктування бази даних, який забезпечує ефективну роботу з даними та їх цілісність.

Алгоритм визначення первинного ключа такий: вибираються потенційні ключі, знаходяться такі потенційні ключі, у яких мінімальна кількість атрибутів, необхідних для забезпечення унікальності – це і буде первинним ключем. Всі інші серед потенційних, які не обрані в первинний, називаються альтернативними.

Зовнішній ключ – це атрибут чи набір атрибутів певної таблиці, значення якої (яких) відповідає значенню потенційного ключа деякої іншої таблиці. Зовнішній ключ організовує, як правило, зв'язок між двома таблицями та забезпечує цілісність посилань.

Повнота інформації в базі даних – це характеристика, яка відображає, наскільки вся необхідна та актуальна інформація представлена в базі даних. У базі даних повинні бути присутні всі дані, які потрібні для виконання її функцій та задоволення потреб користувачів. Наприклад, якщо база даних використовується для обліку клієнтів, вона повинна містити інформацію про кожного клієнта, таку як його ім'я, контактні дані, історію покупок тощо. Інформація в базі даних повинна бути актуальною та відповідати поточному стану справ. Застарілі або невірні дані можуть призвести до помилок та прийняття неправильних рішень. У базі даних не повинно бути дублікатів записів. Кожен запис повинен містити унікальну інформацію про конкретний об'єкт або подію. Дані в базі даних повинні бути узгодженими та не суперечити один одному. Наприклад, якщо інформація про клієнта змінюється в одному місці, ці зміни повинні бути відображені і в інших місцях, де ця інформація використовується.

Повнота інформації є критично важливою для забезпечення коректної роботи бази даних та прийняття обґрунтованих рішень на основі даних. Неповна або неактуальна інформація може призвести до: помилок в роботі системи, прийняття неправильних рішень, втрати довіри користувачів. Для забезпечення повноти інформації в базі даних необхідно ретельно спроектувати

структуру бази даних, встановити правила введення даних, регулярно перевіряти та оновлювати дані.

Несуперечність інформації, що міститься в базі даних, є критично важливим аспектом забезпечення її якості та достовірності. Вона означає, що дані в базі даних не містять внутрішніх конфліктів та узгоджуються між собою, відображаючи реальний стан справ. Дані не повинні суперечити один одному на логічному рівні. Наприклад, не може бути ситуації, коли один і той самий клієнт одночасно зареєстрований у двох різних містах. Дані повинні відповідати встановленим форматам та типам даних. Наприклад, дата народження не може бути введена у текстовому форматі, а номер телефону не може містити літери. Дані повинні мати чітке та однозначне значення. Наприклад, якщо в базі даних є поле «вік», воно повинно відображати кількість повних років людини, а не кількість днів або місяців. Дані повинні відображати актуальний стан справ на даний момент часу. Наприклад, інформація про ціну товару повинна бути актуальною на момент здійснення покупки. Якщо дані посилаються на інші об'єкти в базі даних, ці об'єкти повинні існувати. Наприклад, якщо замовлення посилається на конкретного клієнта, цей клієнт повинен бути зареєстрований в базі даних. Несуперечлива інформація є основою для прийняття обґрунтованих рішень та забезпечення коректної роботи інформаційної системи.

Для забезпечення реляційної моделі даних, база даних повинна відповідати критеріям цілісності даних. Цілісність даних визначається як властивість повноти та несуперечності інформації, що міститься в базі даних.

Будь-яка модифікація даних може призвести до порушення їх цілісності. З метою підтримки коректності та несуперечності інформації, реляційна система управління базами даних використовує механізми забезпечення цілісності даних.

Ці механізми включають встановлення одного або кількох правил цілісності, які регламентують допустимі значення, що можуть бути внесені до бази даних під час операцій додавання або оновлення даних. По-перше, деякі

стовпці в базі даних вимагають обов'язкової наявності даних. Це означає, що кожен рядок у таких стовпцях повинен містити значення, і ніщо не може залишатися порожнім. По-друге, для кожного стовпця існує свій власний домен – набір значень, які можна зберігати в ньому. СУБД може бути налаштована таким чином, щоб відхиляти будь-які спроби записати значення, що не входять до цього домену, забезпечуючи тим самим узгодженість даних. По-третє, первинний ключ, який є унікальним ідентифікатором кожного рядка в таблиці, має в кожному рядку мати унікальне значення. Повторення значень у таких стовпцях є неприпустимим, оскільки СУБД не зможе розрізнити один запис від іншого. Нарешті, якщо в таблиці існує зовнішній ключ, його значення повинні або збігатися зі значеннями первинного або унікального ключа деякої базової таблиці, або містити порожні значення (NULL). Це правило забезпечує цілісність посилань між таблицями та запобігає втраті даних.

У реляційних базах даних, де інформація зберігається у вигляді таблиць, зв'язки між ними є основою організації та доступу до даних. Ці зв'язки встановлюються шляхом зіставлення значень у ключових полях. Зазвичай, унікальний ідентифікатор кожного запису в одній таблиці (головній) пов'язується із зовнішнім ключем іншої таблиці (підпорядкованої).

Зовнішній ключ – це поле або кілька полів, що містять посилання на первинний ключ іншої таблиці. Він служить для встановлення зв'язку між таблицями, вказуючи, як саме вони пов'язані між собою. Важливо зазначити, що значення зовнішнього ключа повинні збігатися зі значеннями первинного ключа, хоча імена полів можуть відрізнятися. Наприклад, уявіть собі дві таблиці: «Клієнти» та «Замовлення». У таблиці «Клієнти» є поле «ID клієнта», яке є первинним ключем та унікально ідентифікує кожного клієнта. У таблиці «Замовлення» є поле «ID клієнта», яке є зовнішнім ключем та вказує на те, який саме клієнт зробив замовлення. Завдяки цьому зв'язку ми можемо легко отримати інформацію про всі замовлення конкретного клієнта, звернувшись до таблиці «Замовлення» та знайшовши всі записи, що містять відповідний ID клієнта у полі зовнішнього ключа. Отже, зв'язки між таблицями є критично

важливими для забезпечення цілісності та логічної структури бази даних, дозволяючи ефективно організовувати та отримувати доступ до необхідної інформації.

В реляційних БД використовуються бінарні зв'язки, яких може бути три види:

- один до одного 1:1;
- один до багатьох 1:M;
- багато до багатьох M:M.

При відношенні «один-до-одного» рядок у таблиці А має рівно один зв'язаний рядок у таблиці В і навпаки. Схематично зв'язок можна зобразити як на рис. 1.3.

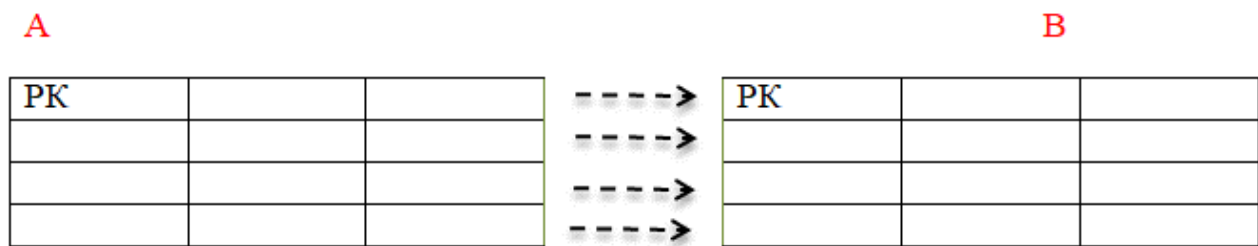


Рисунок 1.3 – Зв'язок між відношеннями 1:1

При відношенні «один-до-багатьох» кожному рядку в таблиці А можуть відповідати кілька рядків у таблиці В (в тому числі і нуль записів), але рядок у таблиці В має рівно один відповідний йому рядок з таблиці А (рис. 1.4).

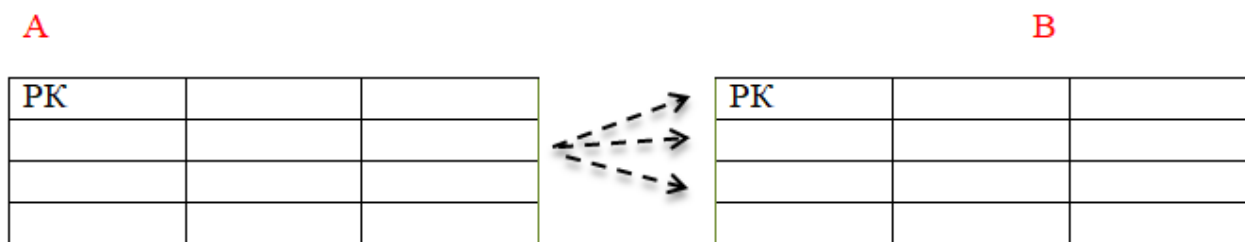


Рисунок 1.4 – Зв'язок між відношеннями 1:M

При відношенні «багато-до-багатьох» одному рядку в таблиці А можуть відповідати кілька рядків у таблиці В (в тому числі і нуль записів), а одному рядку в таблиці В кілька рядків у таблиці А (рис. 1.5).

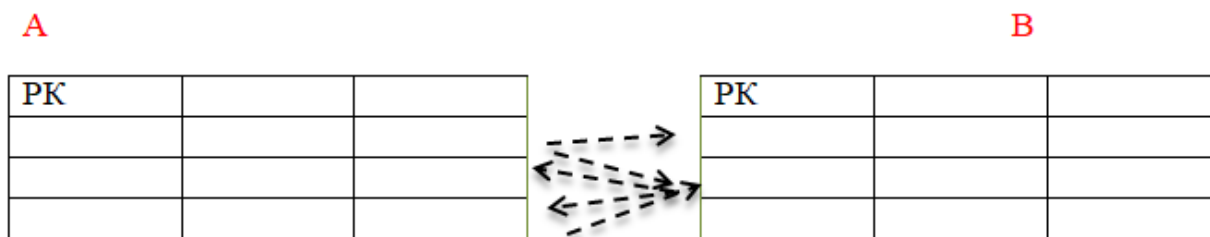


Рисунок 1.5 – Зв’язок між відношеннями М:М

Коли вибрати реляційну СУБД?

- *Одна з основних ознак, яка говорить про те, що потрібно вибрати реляційну СУБД – це висока нормалізація даних. Додатковими ознаками буде необхідність обробки великої кількості коротких транзакцій, з більшою часткою операцій на вставку.*

Коли не вибрати реляційну СУБД?

- *Якщо передбачається зберігати не структуровані дані, або навпаки дуже прості структури типу ключ-значення, краще подивитися у бік документних СУБД і спеціалізованих СУБД типу ключ-значення відповідно. Так само одна з ознак, що має сенс подумати не про реляційні СУБД, це такий факт як необхідність часто оновлювати значення в одних і тих самих рядках. Зазвичай це обходиться «дорого» у реляційних СУБД.*

Серед лідерів реляційних систем управління базами даних виділяють Oracle Database, Microsoft SQL Server, MySQL, PostgreSQL, SQLite, IBM Db2.

Питання для самоперевірки

1. Яке логічне представлення реляційної моделі баз даних?
2. Що таке реляція в термінах реляційних баз даних?
3. Що таке атрибут?
4. Що таке домен атрибута?
5. Що таке кортеж відношення?
6. Яка різниця між потенційним ключем і первинним?
7. Що таке зовнішній ключ?

8. Які бінарні зв'язки між відношеннями ви знаєте?

1.3 SQL і NOSQL бази даних

NoSQL, або нереляційні бази даних, представляють собою парадигму зберігання даних, що відходить від традиційного реляційного підходу. На відміну від реляційних систем, які використовують табличну структуру з фіксованими схемами, NoSQL пропонує більш гнучку модель, часто базуючись на документах (JSON, BSON тощо) або графах. Ця гнучкість дає змогу ефективно зберігати та обробляти різноманітні типи даних, включаючи неструктуровані та напівструктуровані.

Ключові переваги NoSQL полягають у масштабованості, доступності та швидкості обробки запитів, особливо для великих обсягів даних. Розподілена архітектура NoSQL забезпечує високу доступність завдяки реплікації даних та толерантності до відмов. Завдяки відсутності жорстких схем, NoSQL бази даних легко адаптуються до еволюції даних та вимог додатків.

Вибір між реляційною та нереляційною базою даних залежить від конкретних вимог проєкту, таких як тип даних, масштабованість, необхідність складних транзакцій та аналітики. NoSQL особливо добре зарекомендували себе в таких сферах, як веб-додатки, аналітика великих даних та Інтернет речей.

Ієрархічні моделі баз даних, хоча й були першими спробами організованого зберігання даних, мали значні обмеження, пов'язані зі складністю управління та обмеженою гнучкістю. Реляційні бази даних, з їхньою строгою структурою та мовою SQL, виявилися більш ефективним рішенням для багатьох задач. Однак, зростаючі обсяги даних, вимоги до реального часу та необхідність зберігання неструктурованої інформації виявили обмеження реляційної моделі.

NoSQL бази даних, як відповідь на ці виклики, пропонують більш гнучку та масштабовану альтернативу. Відмова від жорстких схем і використання

розподілених архітектур дає змогу NoSQL системам ефективно обробляти великі обсяги даних різноманітних типів, що є критично важливим для сучасних додатків.

Поява NoSQL баз даних не означає повну заміну реляційних систем. Кожна з цих технологій має свої сильні сторони та оптимальні області застосування. Реляційні бази даних, з їхньою структурованою моделлю даних і підтримкою складних запитів, ідеально підходять для задач, що вимагають високого рівня цілісності даних та транзакційності. Однак, їхні обмеження щодо масштабування та гнучкості стають помітними при роботі з великими обсягами неструктурованих або напівструктурованих даних.

NoSQL бази даних пропонують альтернативну парадигму, орієнтовану на масштабованість та гнучкість. Відсутність жорстких схем, розподілена архітектура та підтримка різних моделей даних (документна, ключово-значення, графова) роблять NoSQL ідеальним вибором для таких додатків, як веб-додатки, аналітика великих даних та Інтернет речей.

Сучасні архітектури часто включають як реляційні, так і нереляційні бази даних, дозволяючи оптимально розподілити навантаження та зберігати різні типи даних. Вибір конкретної технології залежить від вимог бізнесу, характеру даних та обчислювальних ресурсів.

Питання для самоперевірки

1. Що таке NoSQL бази даних? Які ключові переваги мають ці СУБД?
2. Які переваги реляційних баз даних?
3. Від чого залежить вибір схеми зберігання даних?

1.4 Документоорієнтовані бази даних

Еволюція систем управління базами даних тісно пов'язана з різноманітністю даних, які необхідно зберігати і обробляти. Традиційні реляційні бази даних, орієнтовані на структуровані дані, виявилися недостатньо

гнучкими для роботи з сучасними типами інформації, такими як текстові документи, мультимедіа та дані соціальних мереж.

Для вирішення цієї проблеми були розроблені NoSQL бази даних, які пропонують більш гнучку модель зберігання даних. Документо-орієнтовані NoSQL бази даних дозволяють зберігати дані у вигляді документів, що мають вільну структуру. Це спрощує завдання зберігання і обробки напівструктурованих та неструктурованих даних, які не піддаються легкій формалізації у вигляді таблиць.

Вибір між реляційною та NoSQL базою даних залежить від конкретних вимог застосування і вимагає аналізу таких факторів, як тип даних, характер робочого навантаження, необхідний рівень транзакційності та масштабованості.

Документо-орієнтовані бази даних NoSQL пропонують альтернативну модель зберігання даних, яка має ряд відмінностей від традиційних реляційних систем. Концептуально, колекції у базах даних документів можна розглядати як аналог таблиць у реляційних базах, а документи – як аналог записів. Кожен документ у колекції ідентифікується унікальним ключем, що дає змогу ефективно здійснювати пошук. Окрім ключового доступу, бази даних документів зазвичай надають потужні можливості для пошуку за вмістом, що робить їх особливо придатними для зберігання і обробки напівструктурованих та неструктурованих даних.

Серед найбільш відомих представників баз даних документів NoSQL можна назвати MongoDB, CouchDB, Couchbase, MarkLogic, OrientDB, IBM Cloudant, CrateDB, Azure Cosmos DB, BaseX, eXist DB та IBM Informix. Кожна з цих систем має свої особливості та оптимально підходить для вирішення різних завдань.

Коли вибирати документо-орієнтовану СУБД?

Документо-орієнтовані СУБД надають ефективне рішення для зберігання даних зі змінною структурою. Вони дозволяють об'єднувати в одній сутності елементи різного типу, такі як скалярні значення, вкладені об'єкти,

масиви та інші структури даних, що часто зустрічаються в форматах, подібних до JSON. Завдяки своїй гнучкості, документо-орієнтовані СУБД знаходять широке застосування як у невеликих мікросервісах, так і в масштабних розподілених системах, де виникає потреба в зберіганні стану складних об'єктів.

Коли не вибирати документо-орієнтовану СУБД?

Ця СУБД не найкраще рішення для реалізації транзакційної моделі.

Питання для самоперевірки

1. У вигляді чого зберігають дані документо-орієнтовані СУБД?
2. Які фактори потрібно проаналізувати для вибору СУБД?
3. Які відмінності від традиційних реляційних баз даних має документо-орієнтована СУБД?
4. Коли вибирати документо-орієнтовану СУБД?

Коли не вибирати документорієнтовану СУБД?

1.5 Базы даних «ключ-значення» (Key-Value)

Базы даних «Ключ-Значення» представляють собою тип NoSQL рішення, їх поява пов'язана з необхідністю швидко записувати, змінювати та отримувати значення за якимось параметром. Нерідко це використовують для некритичних значень, що швидко змінюються без потреби їх запису чи збереження і гарантує реалізацію десятків тисяч операцій за секунду по запису чи зміні даних на малопотужних персональних комп'ютерах. Досягається це завдяки тому, що дані зберігаються в оперативній пам'яті, з якою операції реалізуються швидше. Звідси ж і мінус, якщо її недостатньо, то швидкість буде зменшуватись. Найбільш відомі СУБД такого типу – Redis та Memcached.

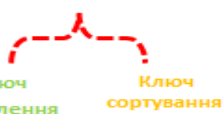
Базы даних «Ключ-Значення» зберігають інформацію у вигляді пар «ключ-значення», де ключ – це унікальний ідентифікатор. Цей тип баз даних

дуже гнучкий і дозволяє зберігати різноманітні типи даних, від простих чисел до складних об'єктів. Вони легко масштабуються і добре підходять для сучасних розподілених систем.

Така гнучкість схем робить бази даних «ключ-значення» особливо ефективними для застосунків, що вимагають високої масштабованості та динамічної зміни структури даних.

У контексті бази даних ключ-значення, ключ розглядається як питання, а значення – як відповідь. На рис. 1.6 наведено приклад зберігання у базі даних ключ-значення українського граматичного словника. Тут первинний ключ складається із двох ключів: ідентифікатора слова і лема слова. Ідентифікатор слова – це ключ розділення або ключ із описом секції, в якій зберігатиметься слово. Багато баз даних типу «ключ-значення» мають вбудовані правила розподілу даних. Ці правила можна налаштувати під свої потреби. Наприклад, числові ключі можна об'єднати в групи по тисячах. Таким чином досягається горизонтальне масштабування. Лема слова – це ключ сортування, який визначає порядок зберігання елементів на жорсткому диску. Комбінація ключа секції та ключа сортування утворює унікальний первинний ключ, який відповідає одному значенню бази даних.

Первинний
ключ



1	Київ	-а	-у	-ом	-і	50.45	30.52	21
2	Стамбул	-а	-у	-ом	-і	41.01224	28.976018	
3	Варшав	-а	-и	-і	-у	-ю	52.2	
4	Софі	-я	-ї	-ю	-єю	42.7	23.33	

Ключ розділення - це спеціальний атрибут або набір атрибутів у даній, який використовується для визначення того, до якого конкретного розділу (partiції) бази даних будуть записані дані. Розділення даних на партиції дозволяє масштабувати систему, покращувати продуктивність запитів і спрощувати управління даними.

Ключ сортування – це поле або комбінація полів, за якими виконується операція сортування.

Рисунок 1.6 – Приклад зберігання даних у базі даних типу «Ключ-Значення»

Бази даних «ключ-значення» мають наступні переваги:

- *можливості масштабування;*
- *зручність використання;*
- *продуктивність;*
- *більшість сховищ «ключ-значення» мають вбудовану функцію реплікації, автоматично дублюючи дані на кількох серверах. Це забезпечує автоматичне відновлення даних у разі виходу з ладу будь-якого з них.*

Коли вибрати СУБД «ключ-значення»:

- *цей тип СУБД ідеально підходить для хешування. Вони дуже швидкі завдяки простому принципу роботи: за кожним записом закріплюється унікальний ключ. Багато таких систем можуть працювати повністю в оперативній пам'яті, що забезпечує блискавичну швидкість доступу. Крім того, вони дозволяють автоматично видаляти старі дані;*
- *проста структура даних і висока пропускна здатність роблять ці СУБД ефективним інструментом для обробки великих обсягів повідомлень у реальному часі.*

Коли не вибрати СУБД «Ключ-Значення»:

- *у випадку, коли база даних передбачає зберігання великої кількості складних сутностей, що включають різноманітні типи даних, і ви очікуєте виконувати складні запити, які повертають множину результатів, традиційні реляційні бази даних є оптимальним вибором. Їхня структура, заснована на таблицях, дозволяє ефективно моделювати складні зв'язки між даними та виконувати складні операції з використанням мови SQL.*

Оскільки бази даних «ключ-значення» не підтримують складних запитів, розробникам доводиться вирішувати цю проблему програмно. В операціях з даними переважно використовуються прості команди, такі як отримання, додавання та видалення. Існують обмеження на фільтрацію та сортування

даних перед доступом до них. Сховище «ключ-значення» спроектовано таким чином, що воно не накладає на розробників жодної схеми. Будь-хто може змінити її безпосередньо в програмі бази даних. Командам розробників необхідно систематично планувати модель даних, щоб уникнути проблем у довгостроковій перспективі. Відсутність жорсткої схеми також означає, що програма відповідає за коректну інтерпретацію отриманих даних. Таку схему часто називають «схемою під час читання».

Системи управління базами даних типу «ключ-значення» віддають перевагу атомарності операцій замість транзакційності. Атомарність гарантує, що кожна операція є неподільною і не може бути частково виконана. Це забезпечує високий рівень цілісності даних, оскільки будь-яка невдала операція не призводить до неконсистентного стану бази. На відміну від цього, транзакційність передбачає виконання декількох операцій як єдиного нерозривного блоку, що дозволяє забезпечувати більш складні сценарії обробки даних.

Питання для самоперевірки

1. Яке призначення СУБД «ключ-значення»?
2. Що таке ключ у контексті СУБД «ключ-значення»?
3. Яким чином можна досягнути горизонтального масштабування в контексті СУБД «ключ-значення»?
4. Коли вибирати СУБД «ключ-значення»?
5. Коли не вибирати СУБД «ключ-значення»? Ч
6. и підтримують СУБД «ключ-значення» складні запити?
7. Чи мають СУБД «ключ-значення» жорстку схему?
8. Що таке «схема під час читання»? Р
9. Які мають переваги СУБД «ключ-значення»?

Рекомендована література до розділу [14, 17, 23-24].

2. РЕЛЯЦІЙНІ БАЗИ ДАНИХ

2.1 Керування і інструменти реляційних баз даних

2.1.1 Мова програмування для управління реляційними базами даних. SQL (Structured Query Language) – це мова програмування, спеціально розроблена для управління реляційними базами даних. SQL та системи управління базами даних, що на цій мові базуються, є однією з ключових технологій в інформатиці. Ця мова, підтримувана сотнями баз даних на різних платформах, від великих серверів до персональних пристроїв, навіть вбудовується в мобільні телефони та автомобілі. Мова SQL, що неодноразово стандартизувалася та розширювалася, лежить в основі майже кожного великого програмного продукту для роботи з даними, включаючи флагманські бази даних таких гігантів, як Microsoft, Oracle та IBM. Крім того, вона є серцем популярних СУБД з відкритим кодом, таких як MySQL та PostgreSQL, сприяючи розвитку Linux та руху відкритого коду. Зі скромного дослідницького проєкту IBM, мова SQL перетворилася на фундаментальну технологію в комп'ютерній сфері.

SQL – це потужна мова, яка дає змогу організовувати, керувати, витягувати та обробляти інформацію, що зберігається в базах даних. Виникнення SQL пов'язано з дослідницькою діяльністю компанії IBM, де прототип цієї мови був відомий як SEQUEL. Через юридичні обмеження, назву було змінено на SQL (Structured Query Language). Сьогодні ця аббревіатура вимовляється двома способами: «сиквел» або «ес-кю-ель». У своїй суті, SQL – це мова програмування, що забезпечує взаємодію користувача з базами даних.

Хоча SQL спочатку розроблявся як мова для запитів до баз даних, її роль значно розширилася. Сьогодні вона є комплексним інструментом, що забезпечує повний спектр функцій СУБД. Процес отримання даних з бази даних, який лежить в основі назви SQL (мова структурованих запитів), залишається важливим, але далеко не єдиним. SQL дає змогу користувачеві

визначити структуру і організацію даних, що зберігаються і взаємини між елементами збережених даних. Вона також дає користувачеві або програмному застосунку можливість витягати з бази дані, що містяться в ній, і користуватися ними. SQL дає змогу користувачу або інформаційній системі змінювати базу даних, тобто додавати в неї нові дані, а також видаляти або оновлювати вже наявні в ній дані. За допомогою SQL можна обмежити можливості користувача за вибіркою, додаванню і зміні даних і захистити їх від несанкціонованого доступу. SQL застосовується для координації спільного використання даних користувачами, що працюють одночасно, з тим щоб зміни, що вносяться одним користувачем, що не приводили до випадкового знищення змін, що вносяться приблизно в той же час іншим користувачем. SQL дає змогу забезпечити цілісність бази даних, захищаючи її від руйнування через неузгоджені зміни або відмови системи.

SQL – це спеціалізований інструмент для керування базами даних, який відрізняється від повноцінних мов програмування, таких як C++, Java або Python. Він є підмовою, що складається з обмеженого набору інструкцій, призначених для вирішення завдань, пов'язаних з базами даних. Ці інструкції можуть бути інтегровані в інші мови програмування, розширюючи їх функціональність, або надсилатися безпосередньо до системи управління базами даних через різні інтерфейси. Ключовою особливістю SQL є її декларативний характер: вона описує, що потрібно отримати від бази даних, а не як це зробити, що робить її унікальною серед процедурних мов програмування.

Перший стандарт SQL, прийнятий Американським національним інститутом стандартів (ANSI) у 1986 році та Міжнародною організацією зі стандартизації (ISO) у 1987 році. У 1989 SQL отримала незначні зміни та доповнення до попереднього стандарту. Значний крок у розвитку SQL відбувся в 1992 році: прийнято новий стандарт, що додав багато нових функцій та можливостей. В 1999 році в SQL було введено підтримку регулярних виразів, рекурсивних запитів, тригерів та об'єктно-орієнтованих концепцій. В 2003 було

додано розширення для роботи з XML-даними. В 2006 – подальше розширення можливостей роботи з XML-даними та інтеграція з XQuery. Покращення і розвиток SQL були в 2008 року. В 2011 року було введено підтримку часових інтервалів, 2016 – JSON, в 2023 – операції над графами.

Важливо зазначити, що хоча ці стандарти існують, багато виробників СУБД додають свої власні розширення та діалекти SQL, що може призвести до деяких відмінностей у реалізації.

2.1.2 Система управління базами даних MS SQL Server. Існують різні випуски MS SQL Server, серед них безкоштовні – це Express і Developer. Для простих застосунків може вистачити випуску Express. У нього є перевага – його можна використовувати як серверний застосунок в реальних задачах, однак він має урізаний функціонал в порівнянні з повною версією.

MS SQL Server Developer Edition – це повнофункціональний випуск, тобто він містить весь функціонал, націлений тільки для потреб розроблення, як і повна версія MS SQL Server Enterprise. У той же час ця версія не може бути використана для розгортання серверного застосунку в реальних проєктах.

Версія Developer Edition є оптимальним варіантом для вивчення всього спектру MS SQL Server, тому саме цю версію буде розглянуто далі. Але, будь-яке використання Developer Edition, яке приносить дохід або безпосередньо обслуговує кінцевих користувачів розробленої системи (наприклад, веб-сайт, який обслуговує клієнтів, або внутрішня бізнес-система), є порушенням ліцензійної угоди. Навіть якщо це тимчасове використання або для збору відгуків чи тестування перед прийняттям, це може вважатися використанням Developer Edition.

Для завантаження MS SQL Server 2022 Developer Edition перейдіть за посиланням <https://www.microsoft.com/en-us/sql-server/sql-server-downloads> (рис. 2.1).



Рисунок 2.1 – Сторінка для встановлення MS SQL Server

Завантажувати інсталятор MS SQL Server потрібно той, який буде працювати саме з вашою операційною системою. Ще можна встановити образ контейнера SQL Server 2022 за допомогою Docker – платформи для управління контейнерами (рис. 2.2).

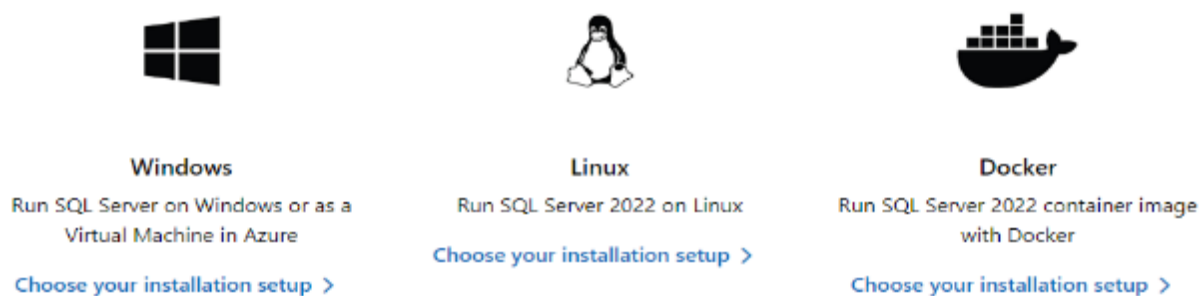


Рисунок 2.2 – Вибір інсталятора MS SQL Server залежно від операційної системи

Також на цьому сайті є інсталятори інструментів для більш комфортної роботи з MS SQL Server. Нам знадобляться SQL Server Management Studio та [Azure Data Studio](#) (рис. 2.3).

SQL Server tools and connectors

Tools

- [Download Azure Data Studio](#)
- [Download SQL Server Management Studio \(SSMS\)](#)
- [Download SQL Server Data Tools \(SSDT\)](#)
- [Download Data Migration Assistant](#)
- [Download SQL Server Migration Assistant for Oracle](#)

Connectors

- [Microsoft ADO.NET for SQL Server](#)
- [Microsoft JDBC Driver for SQL Server](#)
- [Microsoft ODBC Driver for SQL Server](#)
- [Node.js Driver for SQL Server](#)
- [Python Driver for SQL Server](#)
- [Ruby Driver for SQL Server](#)

Рисунок 2.3 – Інструменти і конектори для MS SQL Server

Після встановлення всіх цих програм ви отримаєте на своїй машині систему управління базами даних MS SQL Server та інструменти, які спрощують роботу з ними. MS SQL Server надає діалект мови SQL – T-SQL.

2.1.3 Робота з MS SQL Server в командному рядку. Сервер СУБД не має інтерфейсу користувача. Для виконання операцій з базою даних йому потрібно надсилати команди або за допомогою командного рядка, або за допомогою будь-якої прикладної програми.

Працювати в консолі (командному рядку) дає змогу службова програма sqlcmd. Її область застосування: SQL Server (усі підтримувані версії), база даних SQL Azure, керований екземпляр SQL Azure, Azure Synapse Analytics, Analytics Platform System (PDW).

Sqlcmd дає змогу виконувати запити, вирази T-SQL і сценарії SQL Server за допомогою командного рядка. Для того, щоб його запустити потрібно відкрити консоль (командний рядок, cmd). Консоль операційної системи Windows можна викликати декількома шляхами, наприклад, у меню пуск знайти програму Командний рядок, або «Виконати» (комбінацією клавіш Win + X) і набрати cmd.

Формат запуску службової програми sqlcmd такий:

```
sqlcmd [ /E ] [ /S servername ] .
```

Наприклад, щоб запустити sqlcmd і з'єднатися із сервером на ім'я DESKTOP-BJ9ERKQ\SQLEXPRESS, користувач Users має набрати такий код:

```
Microsoft Windows [Version 10.0.19045.5487]
(c) Корпорація Майкрософт. Усі права захищені.
C:\Users\1>sqlcmd -S DESKTOP-BJ9ERKQ\SQLEXPRESS
1>
```

Цифра 1> означає, що є підключення і програма sqlcmd готова приймати ваші команди для виконання.

Якщо ввімкнена автентифікація SQL Server, то потрібно буде вказати ім'я користувача та пароль користувача:

```
sqlcmd -S DESKTOP-BJ9ERKQ\SQLEXPRESS -U ім'я_користувача,
```

а коли і пароль задаєте, то:

```
sqlcmd -S DESKTOP-BJ9ERKQ\SQLEXPRESS -U ім'я_користувача -P
```

далі повинні ввести пароль.

Для того, щоб подивитися, яка база даних зазначена за замочуванням, потрібно наберіти такий код:

```
1> select DB_NAME()  
2> GO  
  
-----  
master  
  
(1 rows affected)  
1>
```

База даних master – це системна база даних MS SQL Server, яка зберігає всю інформацію сервера, включаючи облікові записи сервера, відомості про всі бази даних та їх користувачів.

Щоб переглянути список баз даних MS SQL Server використовується команда:

```
1> select name from sys.databases  
2> go  
name
```

У результаті буде перераховано всі бази даних в екземплярі SQL:

```
-----  
master  
  
tempdb  
  
model  
  
msdb
```

Для отримання списку баз даних також можна використовувати збережену процедуру sp_databases:

```
1> Sp_databases  
2> Go
```

У цьому випадку буде отримано подібний результат:

Adress	16384 NULL
CheckList	16384 NULL
Education	16384 NULL
FirstDB	16384 NULL
master	7552 NULL
model	

Щоб перевірити, чи SQL Server чутливий до регістру в sqlcmd, потрібно ввести таку команду:

```
1> SELECT CONVERT (varchar, SERVERPROPERTY('collation')) AS 'Server Collation'
2> GO
```

У відповідь можете отримати, наприклад, таку інформацію:

```
Server Collation
-----
Ukrainian_CI_AS
```

Ukraine – визначає сортування символічних рядків за українським алфавітом, CI означає нечутливість до регістру, а CS – чутливість до регістру. AS означає чутливість до акценту, а AI – нечутливість до акценту.

Цю інформацію можливо перевірити за допомогою процедури sp_helpsort:


```

(1 rows affected)
1>
2> sp_helpsort
3> go
Server default collation

-----
Ukrainian, case-insensitive, accent-sensitive, kanatype-insensitive, width-insensitive

```

Випуск SQL Server можливо перевірити, використовуючи такі скрипти T-SQL:

```

1> SELECT SERVERPROPERTY('EDITION')
2> GO

```

Результат може бути такий:

```

-----
Express Edition (64-bit)

```

При встановленні SQL Server було два варіанти автентифікації на SQL Server:

автентифікація Windows, де можна використовувати обліковий запис Active Directory або локальний обліковий запис Windows;

автентифікація Windows і SQL, де ви також можете пройти автентифікацію за допомогою облікового запису, створеного в SQL Server.

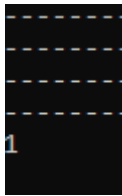
Щоб перевірити автентифікацію, можливо використовувати такі скрипти:

```

(1 rows affected)
1> SELECT SERVERPROPERTY('IsIntegratedSecurityOnly')
2> GO

```

Результат виконання цього скрипта може бути таким:



Якщо результат 0, це означає, що обидві автентифікації ввімкнено. Якщо значення 1, ввімкнено лише автентифікацію Windows.

Змінні середовища SQL Server – це динамічні значення, які впливають на поведінку процесів, запущених у SQL Server. Вони надають інформацію про конфігурацію системи та параметри, які можна використовувати для налаштування роботи сервера. Змінні середовища дозволяють налаштовувати різні аспекти роботи SQL Server, такі як шляхи до файлів, параметри мережі та інші системні налаштування. Вони можуть використовуватися для управління ресурсами, такими як пам'ять та процесорний час, виділені для SQL Server. Змінні середовища можуть надавати корисну інформацію для діагностики проблем та усунення несправностей. Важливо розрізняти системні змінні середовища операційної системи, які впливають на всі процеси, запущені на сервері, включаючи SQL Server, та внутрішні змінні конфігурації SQL Server, які встановлюються безпосередньо в рамках самої інстанції SQL Server.

Хоча термін "змінні середовища SQL Server" може іноді використовуватися для позначення конфігураційних параметрів, які можна змінити за допомогою системних процедур `sp_configure` або через SQL Server Management Studio, важливо розуміти, що на рівні операційної системи SQL Server також може взаємодіяти з традиційними змінними середовища Windows або Linux. Наприклад, змінна `PATH` може впливати на те, як SQL Server знаходить певні виконувані файли.

Щоб отримати список усіх змінних середовища, виконайте таку команду в `sqlcmd`:

```
sqlcmd> :ListVar
```

На екрані відобразиться схожий результат:

```
SQLCMDCOLSEP = " "  
SQLCMDCOLWIDTH = "0"  
SQLCMDDBNAME = ""  
SQLCMDEDITOR = "edit.com"  
SQLCMDERRORLEVEL = "0"  
SQLCMDHEADERS = "0"  
SQLCMDINI = ""  
SQLCMDLOGINTIMEOUT = "8"  
SQLCMDMAXFIXEDTYPEWIDTH = "0"  
SQLCMDMAXVARTYPEWIDTH = "256"  
SQLCMDPACKETSIZE = "4096"  
SQLCMDSERVER = "DESKTOP-BJ9ERKQ\SQLEXPRESS"  
SQLCMDSTATTIMEOUT = "0"  
SQLCMDUSER = "1"  
SQLCMDWORKSTATION = "DESKTOP-BJ9ERKQ"  
1>
```

Питання для самоперевірки

Дайте визначення поняття «база даних» і «таблиця».

Що таке SQL?

Які завдання SQL?

Які на тепер є стандарти SQL?

Що таке система керування базою даних?

Наведіть приклади таких систем.

Що зберігається в базі даних master?

Що таке SQL? Де і коли SQL використовується?

Завдання

Встановити MS SQL Server та інструменти для роботи з ним.

Виконати всі приклади із розділу 2.1.3

Вивчити всі змінні середовища MS SQL Server, надати їх опис, пояснити чому саме такі значення у них.

2.2 Створення, структура та основні елементи баз даних SQL Server

2.2.1 Створення і видалення бази даних.. Самий простий спосіб створити базу даних набрати таку команду в sqlcmd:

```
1> CREATE DATABASE MyFirstDB
2> GO
```

Після виконання команди буде створено екземпляр бази даних з назвою 'MyFirstDB'. Для підтвердження існування об'єкта необхідно виконати:

```
1> select name from sys.databases
2> GO
name
-----
master
tempdb
model
msdb
CheckList
FirstDB
Education
Adress
Sales
MyFirstDB
```

Як видно з останнього рядка із скріна – база даних MyFirstDB на сервері є. У цьому випадку всі параметри створеної бази даних ідуть за замовчуванням. А самі файли з базою даних знаходяться в теці, в яку встановлений MS SQL Server, наприклад:

C:\Program Files\Microsoft SQL Server\MSSQL15.SQLEXPRESS\MSSQL\DATA.

Хоча створення бази даних з параметрами за замовчуванням зручне для швидких тестів або розробки, для виробничих середовищ це майже завжди погана ідея. Завжди варто приділяти час для ретельного планування та налаштування параметрів бази даних, враховуючи її очікуване навантаження, обсяг даних, вимоги до доступності та відновлення. Це допоможе забезпечити стабільну роботу, оптимальну продуктивність та високий рівень безпеки вашої системи.

Для створення бази даних (БД) Sales, яка буде зберігатися у файлі salesdat.mdf в теці D:\DATA\ (перед цим потрібно створити цю теку), наберіть такий код в будь-якому текстовому редакторі:

```

USE master;
GO
CREATE DATABASE Sales
ON
( NAME = Sale,
  FILENAME = 'D:\DATA\salesdat.mdf',
  SIZE = 10,
  MAXSIZE = 50,
  FILEGROWTH = 5 )
LOG ON
( NAME = Sale_log,
  FILENAME = 'D:\DATA\saleslog.ldf',
  SIZE = 5MB,
  MAXSIZE = 25MB,
  FILEGROWTH = 5MB );
GO

```

Шлях до теки можливо вибрати інший, але потрібно врахувати це, коли будете використовувати код, наданий у прикладі.

Скопіюйте цей код в командний рядок sqlcmd:

```

1> USE master;
2> GO
Changed database context to 'master'.
1> CREATE DATABASE Sales
2> ON
3> ( NAME = Sale,
4>   FILENAME = 'D:\DATA\salesdat.mdf',
5>   SIZE = 10,
6>   MAXSIZE = 50,
7>   FILEGROWTH = 5 )
8> LOG ON
9> ( NAME = Sale_log,
10>  FILENAME = 'D:\DATA\saleslog.ldf',
11>  SIZE = 5MB,
12>  MAXSIZE = 25MB,
13>  FILEGROWTH = 5MB );
14> GO

```

У результаті створення БД у теці DATA, з'являться два файли – salesdat.mdf і saleslog.ldf (рис. 2.4). Тобто на одну базу даних припадає як мінімум два файли: *.mdf і *.ldf. В першому зберігаються самі дані, таблиці, індекси тощо, а в другому знаходяться транзакції та інформація, яка необхідна для відновлення бази даних.

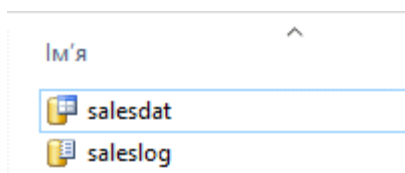


Рисунок 2.4 – Створена база даних Sales – файл з даними та файл з журналом транзакцій

Схематичне зображення скрипту створення БД подано на рис. 2.5.

```
CREATE DATABASE NameDB|
ON
(
*****
*****
)
LOG ON
(
*****
***** );
```

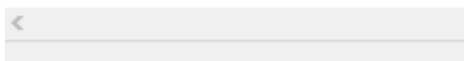


Рисунок 2.5 – Схематичний скрипт створення БД з ім'ям NameDB

CREATE DATABASE Sales – це команда для створення БД з ім'ям Sales. Ця команда містить параметри NAME, FILENAME, SIZE, MAXSIZE та FILEGROWTH, які описують створювані файли. Серед цих параметрів лише параметр FILENAME є обов'язковим.

БД може мати фізичне і логічне ім'я.

NAME = Sale – це параметр для задавання логічного імені, за яким можна звернутися до БД в скриптах. У наведеному вище прикладі логічне ім'я буде Sale.

FILENAME = 'D:\DATA\salesdat.mdf' – це параметр для задавання фізичного імені та повного шляху до файлу операційної системи, де зберігатиметься створювана БД. Зверніть увагу, що його значення є рядком, тобто має бути вказаним в одинарних лапках. Цей параметр вказує ім'я та шлях до файлу. Шлях обов'язково має вказувати на теку на обчислювальному пристрої, на якому інстальовано SQL Server. Не можна використовувати мережеві диски інших пристроїв.

SIZE = 10 – це параметр для задавання початкового розміру кожного створюваного файлу.

Усі створювані в SQL Server бази даних використовують деякий шаблон – база даних model, яка завжди має бути в системі з SQL Server. Вміст бази даних model (включаючи параметри) під час виконання команди CREATE DATABASE повністю копіюється у першу частину створюваної бази даних, а решта нової бази заповнюється порожніми сторінками. Тому розмір, вказаний для основного файлу бази даних, повинен бути не меншим за розмір основного файлу бази даних model. У разі зміни бази даних model усі створені після цього бази даних успадкують ці зміни. Вихідний розмір за замовчуванням файлу журналу транзакцій та файлу даних для бази даних model становить 8 МБ.

У наведеному вище прикладі для файлу даних salesdat.mdf задано розмір 10 МБ, а для файлу транзакцій saleslog.ldf – 5 МБ (рис. 2.6).

Ім'я	Дата змінення	Тип	Розмір
 salesdat	01.02.2023 13:25	SQL Server Databa...	10 240 КБ
 saleslog	01.02.2023 13:25	SQL Server Databa...	5 120 КБ

Рисунок 2.6 – Тека зі створеними файлами заданого розміру

В процесі заповнення БД ці файли будуть збільшуватись. Розмір файлів можна контролювати, встановивши максимальний розмір і крок збільшення файлів.

MAXSIZE – максимальний розмір файлу, ціле число. Якщо не вказано максимальний розмір, то розмір файлу програмно не обмежується, тобто файл може збільшуватись доти, доки є вільне місце у дисковому просторі.

FILEGROWTH – одиниця збільшення файлу, вказується в Мб (за змовчуванням) або у відсотках, значення 0 забороняє збільшення файлу. Це значення використовується, якщо потрібно додати дані у базу, але «зарезервоване» під нові дані місце скінчилось (база заповнена). У цьому випадку її розмір збільшується на вказане значення (але не більше за MAXSIZE). Тобто у базу спочатку додаються порожні сторінки згідно шаблону, які вже потім заповнюються новими даними.

Значення параметрів SIZE, MAXSIZE та FILEGROWTH мають бути обов'язково цілими числами. За замовчуванням ці значення вказуються в

мегабайтах, але це можна змінити за допомогою суфіксів: кілобайт (KB), мегабайт (MB), гігабайт (GB) і терабайт (TB). Значення FILEGROWTH може бути також задано у процентах, для цього використовується суфікс %.

Якщо FILEGROWTH вказано у %, то розмір приросту дорівнює зазначеному відсотку від розміру файлу на момент збільшення. У цьому випадку розмір округлюється до найближчих 64 КБ, а мінімальне значення становить 64 КБ.

Для видалення БД (у нашому випадку MyFirstDB) потрібно в консолі набрати:

```
> drop MyFirstDB
```

Для забезпечення успішного видалення бази даних, необхідно переконатися, що жоден користувач, включаючи адміністратора, не має активних з'єднань з нею. Щоб розірвати всі існуючі з'єднання з базою даних, що підлягає видаленню, виконайте наступну команду:

```
2> use master
```

2.2.2 Створення таблиці бази даних за допомогою командного рядка.

Ви можете не тільки виконувати окремі команди за допомогою командного рядка, а й запускати сценарії в командному режимі.

Сценарії командного рядка – це файли, у яких в текстовому вигляді збережено набори команд. Тобто це такі ж команди, як і для виконання через командний рядок, але збережені у файлі. У разі запуску таких файлів усі команди будуть виконуватись послідовно, тож не забувайте писати кожен окрему команду у файлі з нового рядка. Сценарії можна створювати у будь-якому текстовому редакторі, наприклад, у «Блокноті», тільки під час збереження треба буде вказати розширення – *.sql.

Відкрийте текстовий редактор і наберіть такий скрипт для створення таблиці з ім'ям Goods в БД Sales:

```
CREATE TABLE Goods /* створення таблиці Goods */  
(
```



```

ID_Product Integer Not Null Primary Key,      /* створення стовпця ID_Product типу
Integer, який не може бути порожнім і є первинним ключем */
Name VarChar(100),      /* створення стовпця Name типу VarChar, довжина елементів
якого до 100 символів */
Price Decimal(15, 2),      /* створення стовпця Price типу Decimal, елементи якого
можуть бути довжиною не більше 15 символів до коми і не більше 2 символів після коми */
Count Integer,      /* створення стовпця Count типу Integer */
Date Date,      /* створення стовпця Date типу Date */
Note VarChar(200) /* створення стовпця Note типу VarChar, довжина елементів якого
до 200 символів */

```

Збережіть цей файл з назвою example.sql у теку з БД Sales (у цьому прикладі шлях до бази даних D:\DATA\). Виконайте цей сценарій з командного рядка:

```

C:\Users\1>sqlcmd -S DESKTOP-BJ9ERKQ\SQLEXPRESS -E -i D:\DATA\example.sql
Changed database context to 'Sales'.

```

У результаті буде створена таблиця Goods у БД Sales.

2.2.3 Видалення таблиці бази даних за допомогою командного рядка. Для перегляду інформації щодо всіх таблиць, які входять у склад БД Sales, потрібно створити сценарій такого запиту:

```

use master
select table_name from sales.information_schema.tables
GO

```

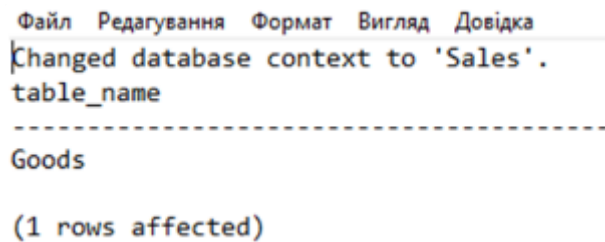
Після збереження сценарію у файлі example1.sql можна виконати його у командному рядку:

```

C:\Users\1>sqlcmd -E -i example1.sql -o D:\output.txt -S DESKTOP-BJ9ERKQ\SQLEXPRESS
C:\Users\1>

```

Параметр -i використовується для визначення вхідних даних, а саме вказується файл сценарію із запитом. Параметр -o використовується для запису результатів виконання сценарію у файл. У результаті виконання цього скрипту у файл output.txt записується інформація про всі таблиці БД Sales (рис. 2.7).

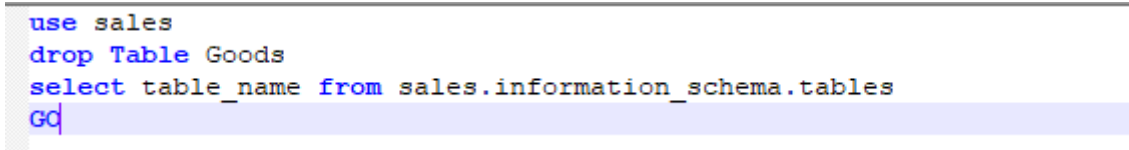


```
Файл  Редагування  Формат  Вигляд  Довідка
Changed database context to 'Sales'.
table_name
-----
Goods

(1 rows affected)
```

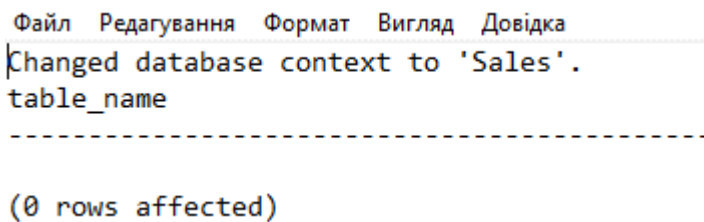
Рисунок 2.7 – Вміст файлу output.txt – інформація про всі таблиці БД Sales

Для видалення таблиці Goods змініть текст у файлі example1.sql таким чином:



```
use sales
drop Table Goods
select table_name from sales.information_schema.tables
GQ
```

Знову виконайте цей сценарій у командному рядку і перегляньте змінений файл output.txt (рис. 2.8).



```
Файл  Редагування  Формат  Вигляд  Довідка
Changed database context to 'Sales'.
table_name
-----

(0 rows affected)
```

Рисунок 2.8 – Вміст файлу output.txt після видалення таблиці

Як видно з файлу output.txt таблиця Goods більше не відображується серед інформації про всі таблиці БД Sales, тобто її було видалено.

2.2.4 Створення і видалення бази даних в середовищі SQL Server Management Studio. Для створення баз даних, крім застосування скриптів на мові SQL, поширеним є використання спеціальних програм, які мають зручні інструменти для створення та адміністрування БД. Однією з найбільш поширених таких програм є SQL Server Management Studio (SSMS), яка містить багато графічних інструментів і багатофункціональних редакторів скриптів для доступу до SQL Server.

Відкрийте SSMS, натисніть правою кнопкою миші на вузол Databases і в контекстному меню виберіть пункт New Database (рис. 2.9).

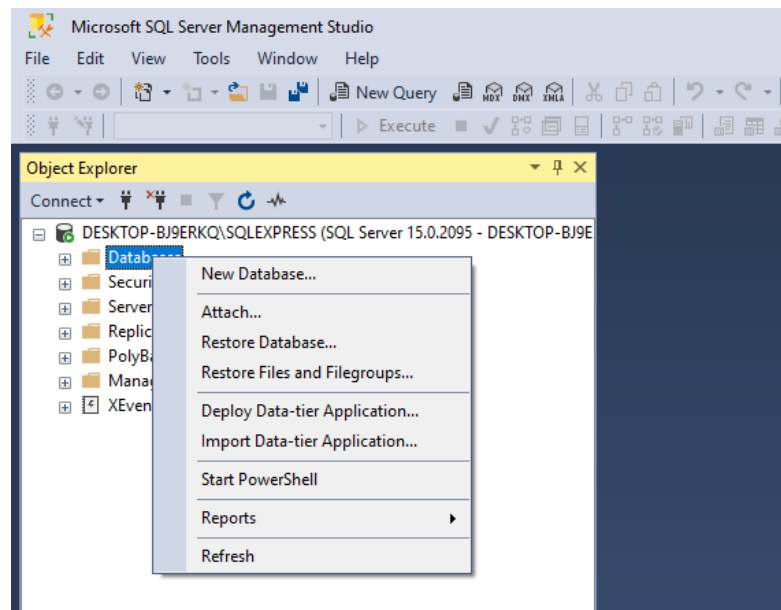


Рисунок 2.9 – Створення нової бази даних в SQL Server Management Studio: пункт New Database

Відкриється вікно для створення бази даних (рис. 2.10). В поле Database name необхідно ввести назву нової бази даних. У розглянутому прикладі нова база даних буде мати назву FIT. Поле Owner задає власника бази даних. За замовчуванням воно має значення <default>, тобто власником буде той, хто створює цю базу даних.

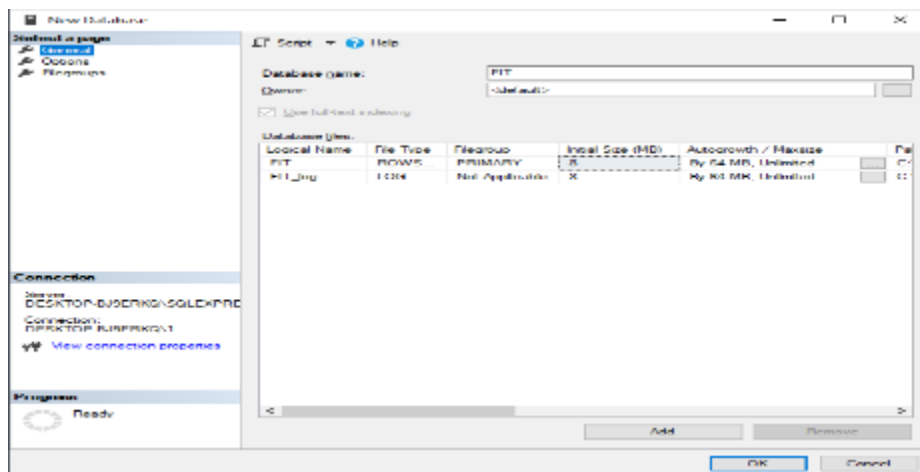


Рисунок 2.10 – Вікно для створення бази даних

Далі йде таблиця для установки загальних налаштувань бази даних. Вона містить два рядки: перший – для установки налаштувань головного файлу, де будуть зберігатися дані, і другий – для конфігурації файлу логування. Зокрема, можливо встановити такі налаштування:

Logical Name – логічне ім'я, яке присвоюється файлу бази даних.

File Type – є кілька типів файлів, але, як правило, основна робота ведеться з файлами даних (ROWS Data) і файлом LOG.

Filegroup – визначає групу файлів, яка може зберігати безліч файлів і може використовуватися для розбиття бази даних на частини для розміщення в різних місцях.

Initial Size (MB) – встановлює початковий розмір файлів, які створюються. Фактичний розмір може відрізнятися від цього значення.

Autogrowth/Maxsize – це значення для кроку збільшення і максимального розміру файлу (аналогічно FILEGROWTH і MAXSIZE, які описані вище).

Path – шлях до теки, де буде зберігатися створювана база даних.

File Name – безпосереднє ім'я фізичної файлу. Якщо воно не вказано, то застосовується логічне ім'я.

Створена база даних FIT з'явиться серед БД сервера.

Якщо згодом ця база даних стане непотрібна, то її можна видалити, натиснувши на неї правою кнопкою миші і вибравши в контекстному меню пункт Delete (рис. 2.11).

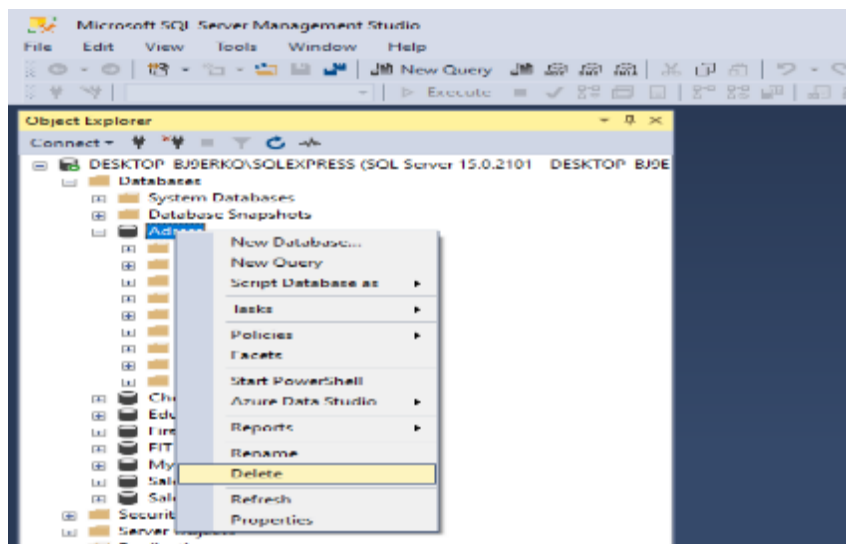


Рисунок 2.11 – Видалення бази даних в середовищі SQL Server Management Studio

В результаті видалення бази даних звільняється весь зарезервований під неї дисковий простір і видаляються всі об'єкти, які в ній зберігались.

Видалену базу даних можна відновити лише за допомогою резервної копії, якщо вона була попередньо створена.

2.2.5 Створення, заповнення і видалення таблиці бази даних в середовищі SQL Server Management Studio. Створення таблиць бази даних в SQL Server Management Studio можливе за допомогою скриптів на мові SQL або графічного дизайнера таблиць. Для того, щоб відкрити графічний дизайнер таблиць, у провіднику об'єктів SSMS розкрийте вузол бази даних натисніть на його підвузол Tables правою кнопкою миші і в контексті меню виберіть New -> Table (рис. 2.12).

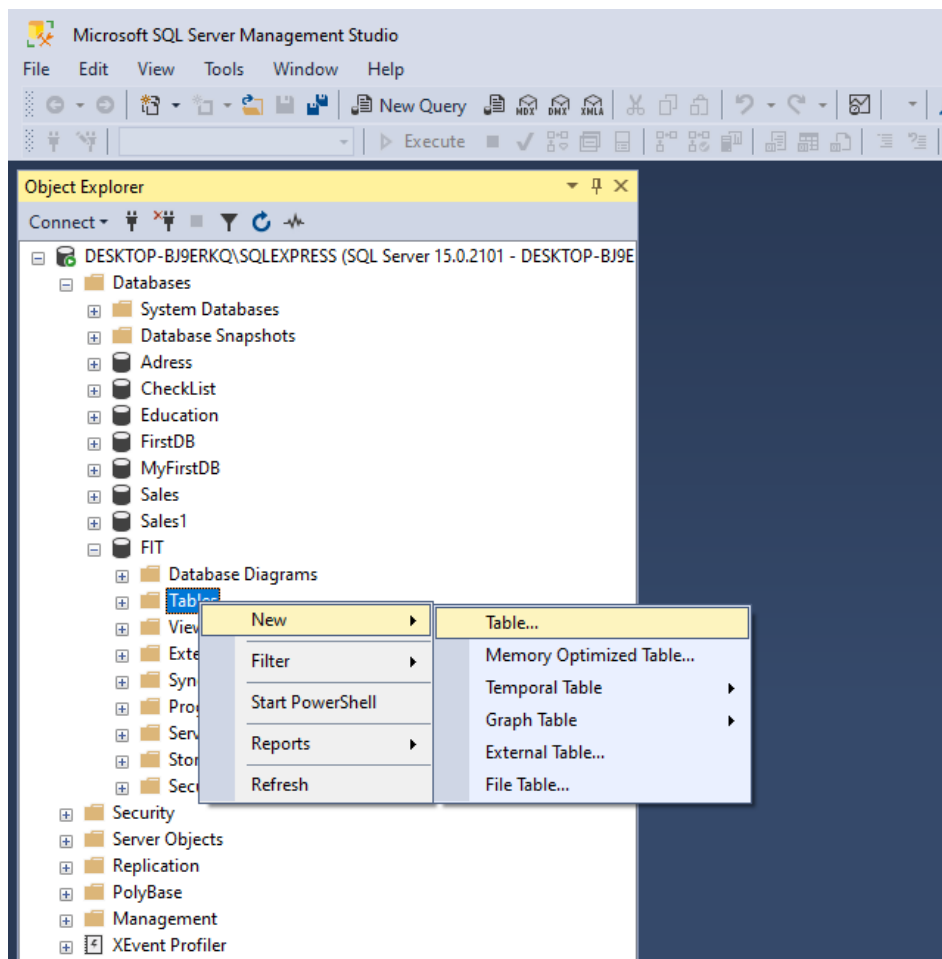


Рисунок 2.12 – Створення нової таблиці в SQL Server Management Studio

Після цього відкриється дизайнер таблиці, який містить три колонки:

Column Name – ім'я стовпця визначає, які будуть стовпці (поля) у створюваній таблиці.

Data Type – тип даних стовпця визначає, які дані можуть зберігатися в кожному стовпці створюваної таблиці.

Allow Nulls – індикатор незаданих значень, який визначає, чи може бути відсутнім значення у стовпці під час додавання нового запису (нового рядка), тобто чи може комірка у цьому стовпці бути порожньою.

Приклад: створення таблиці з даними студентів нашого факультету (рис. 1.6). Для цього в дизайнері таблиці створіть чотири стовпці:

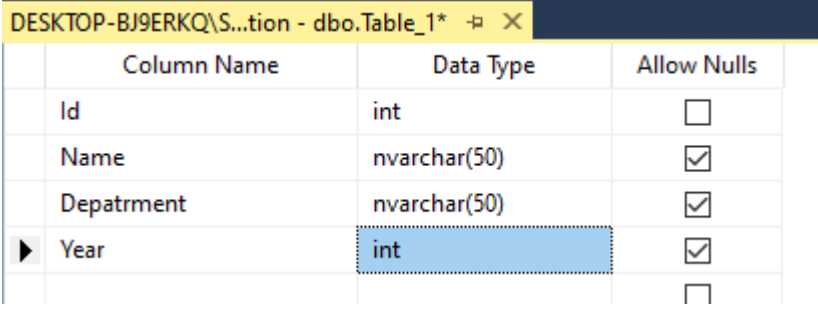
Id – унікальний ідентифікатор студента,

Name – ім'я студента,

Department – кафедра,

Year – рік навчання.

У першого і четвертого стовпця буде тип int, а у стовпців Name, Department – тип nvarchar(50) (рис. 2.13).



DESKTOP-BJ9ERKQ\S...tion - dbo.Table_1* X			
	Column Name	Data Type	Allow Nulls
	Id	int	☐
	Name	nvarchar(50)	☒
	Department	nvarchar(50)	☒
►	Year	int	☒

Рисунок 2.13 – Приклад створення таблиці з даними студентів

У вікні Properties, яка містить властивості таблиці, в поле Name потрібно ввести ім'я таблиці – Students. Ім'я таблиці повинно бути унікальним в рамках бази даних. В поле Identity введіть Id. Таким чином вказується, що стовпець Id буде ідентифікатором (рис. 2.14).

Слід зазначити, що стовпець «Id» буде визначено як первинний ключ. Первинний ключ слугує для забезпечення унікальної ідентифікації кожного запису в таблиці. Первинний ключ може складатися як з одного, так і з декількох стовпців. Складений (Композитний) первинний ключ складається з двох або більше стовпців. Використовується, коли жоден окремий стовпець не може унікально ідентифікувати запис, але їх комбінація може. Це часто

трапляється в таблицях зв'язків ("багато-до-багатьох") або коли природний ключ складається з кількох атрибутів.

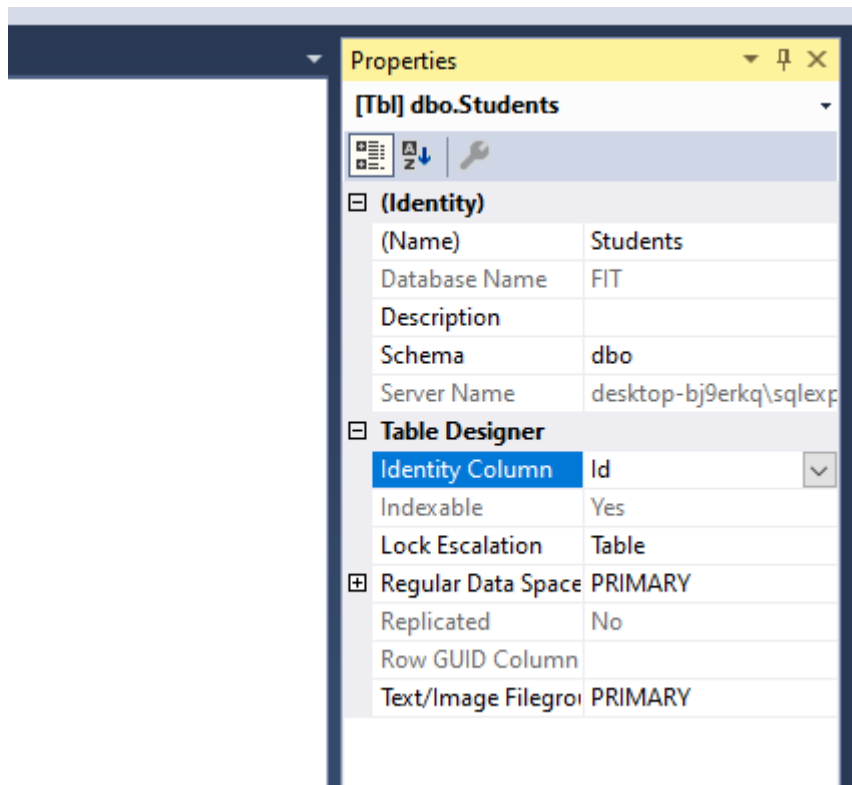


Рисунок 2.14 – Вікно Properties – властивості таблиці Students

Ключовою властивістю первинного ключа є його унікальність: кожне значення первинного ключа в таблиці повинно бути неповторним. Для забезпечення цієї вимоги, SQL Server автоматично створює унікальний індекс для первинного ключа, що унеможливорює вставлення двох рядків з ідентичним значенням ключа. Окрім унікальності, кожен стовпець, що входить до складу первинного ключа, повинен мати непусте значення (NOT NULL). Це означає, що жоден з компонентів первинного ключа не може містити значення NULL, тим самим гарантуючи, що кожний запис має чітко визначений ідентифікатор. Первинний ключ також є основним механізмом для ідентифікації та посилання на конкретний запис у таблиці. Операції пошуку, оновлення або видалення певних записів зазвичай здійснюються із використанням значення їхнього первинного ключа.

Крім того, первинний ключ слугує основою для встановлення взаємозв'язків між таблицями в реляційній моделі. Зовнішній ключ посилюється

на первинний ключ іншої таблиці, встановлюючи зв'язок типу "батьки-нащадки" або "один-до-багатьох". Цей механізм забезпечує реферативну цілісність, унеможливаючи додавання запису до "дочірньої" таблиці за відсутності відповідного запису в "батьківській". Нарешті, слід зазначити, що SQL Server автоматично створює кластерний індекс (якщо інше не визначено) на первинному ключі. Кластерний індекс визначає фізичний порядок зберігання даних у таблиці, що значно прискорює операції пошуку та вибірки даних за цим ключем. Якщо кластерний індекс вже присутній у таблиці, первинний ключ створюється як некластерний індекс.

Для встановлення первинного ключа потрібно натиснути правою кнопкою миші на стовпець Id і в контекстному меню обрати пункт Set Primary Key (рис. 2.15). Після цього напроти поля Id повинен з'явитися золотий ключик. Цей ключик буде вказувати, що стовпець Id виконуватиме роль первинного ключа.

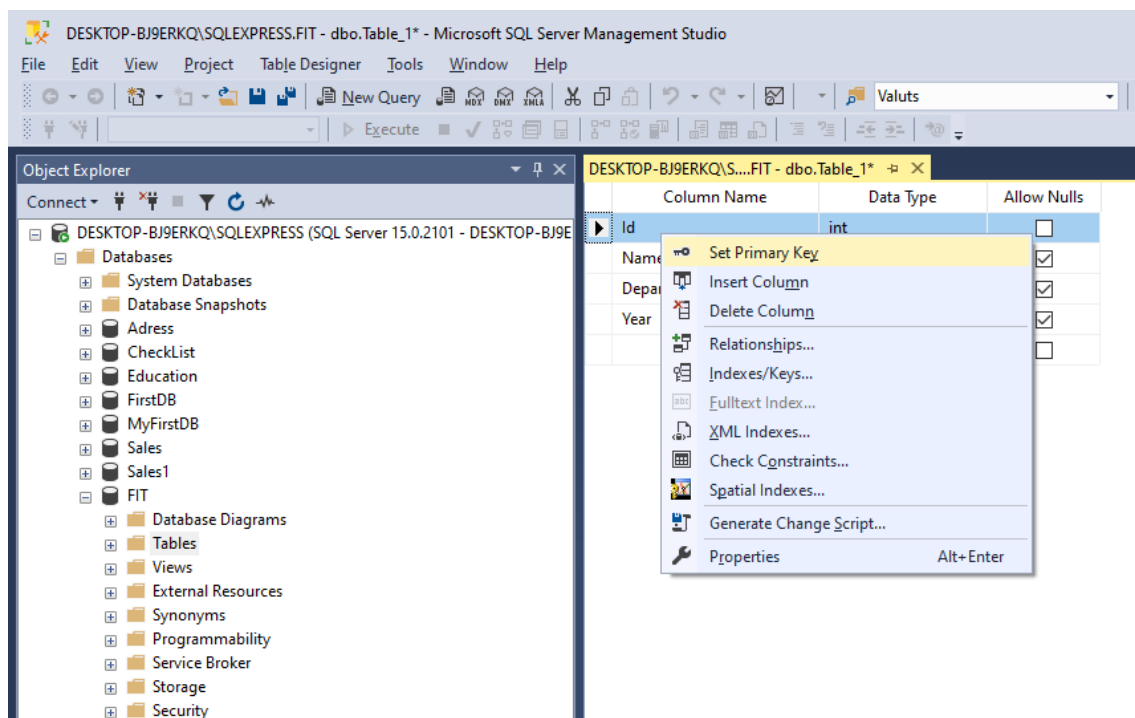


Рисунок 2.15 – Установлення первинного ключа стовпцю Id – Set Primary Key

Для того щоб зберегти таблицю, на шапці вікна з таблицею потрібно натиснути праву кнопку миші і в контекстному меню обрати опцію Save

Students (рис. 2.16). Після цього у базі даних, коли розкриється гілка Таблиці з'явиться нова таблиця, у цьому випадку таблиця з назвою Students. Можливо потрібно буде оновити базу даних, для цього слід на назві бази даних натиснути праву кнопку миші і у контекстному меню обрати опцію Refresh.

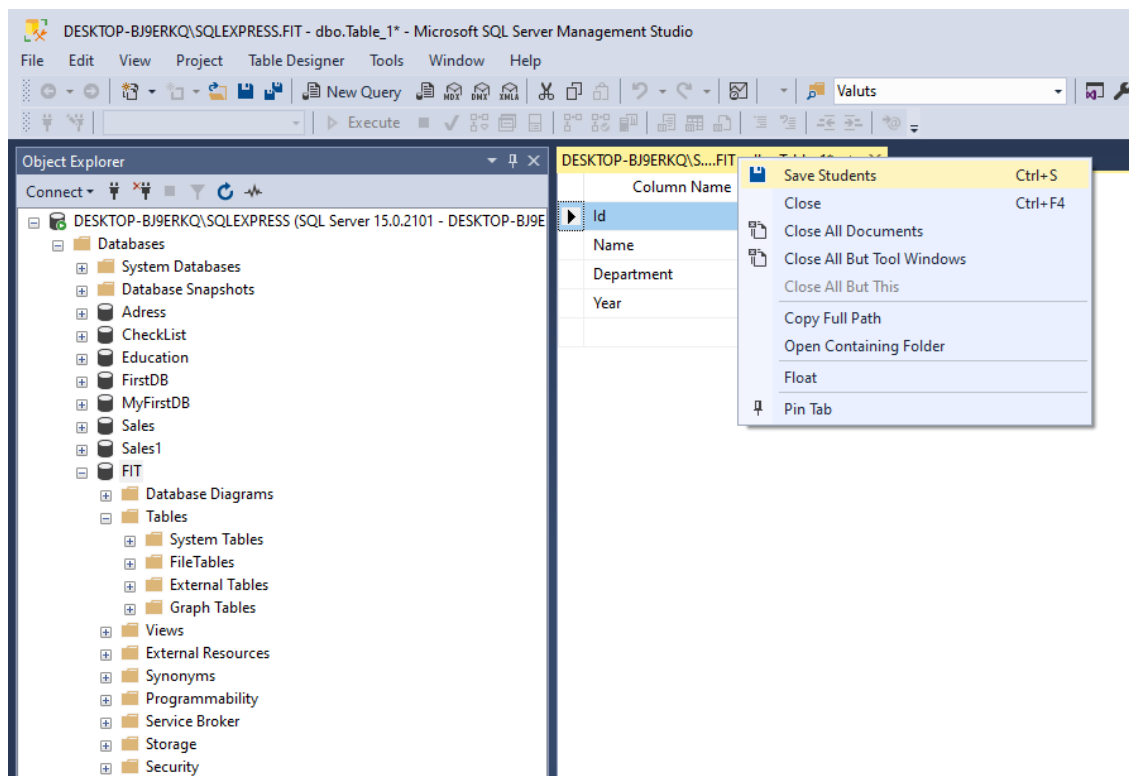


Рисунок 2.16 – Збереження таблиці Students бази даних FIT – Save Students

Для управління таблицею в середовищі SSMS використовується контекстне меню, яке викликається натисканням правої кнопки миші на назві таблиці у провіднику об'єктів (рис. 2.17). Контекстне меню надає доступ до основних функцій роботи з таблицею:

- Опція Delete забезпечує видалення таблиці з бази даних;

- Опція Rename дає змогу змінити назву таблиці;

- Опція Design відкриває вікно дизайнера таблиці, де можна модифікувати структуру таблиці шляхом додавання, видалення або зміни атрибутів полів;

- Опція Edit Top 200 Rows надає можливість додавати, редагувати та видаляти записи (рядки таблиці).

При виборі опції Edit Top 200 Rows відображається табличне представлення для маніпулювання даними, який за замовчуванням відображає до 200 перших записів. У випадку нової таблиці це представлення буде порожнім, оскільки дані ще не внесені.

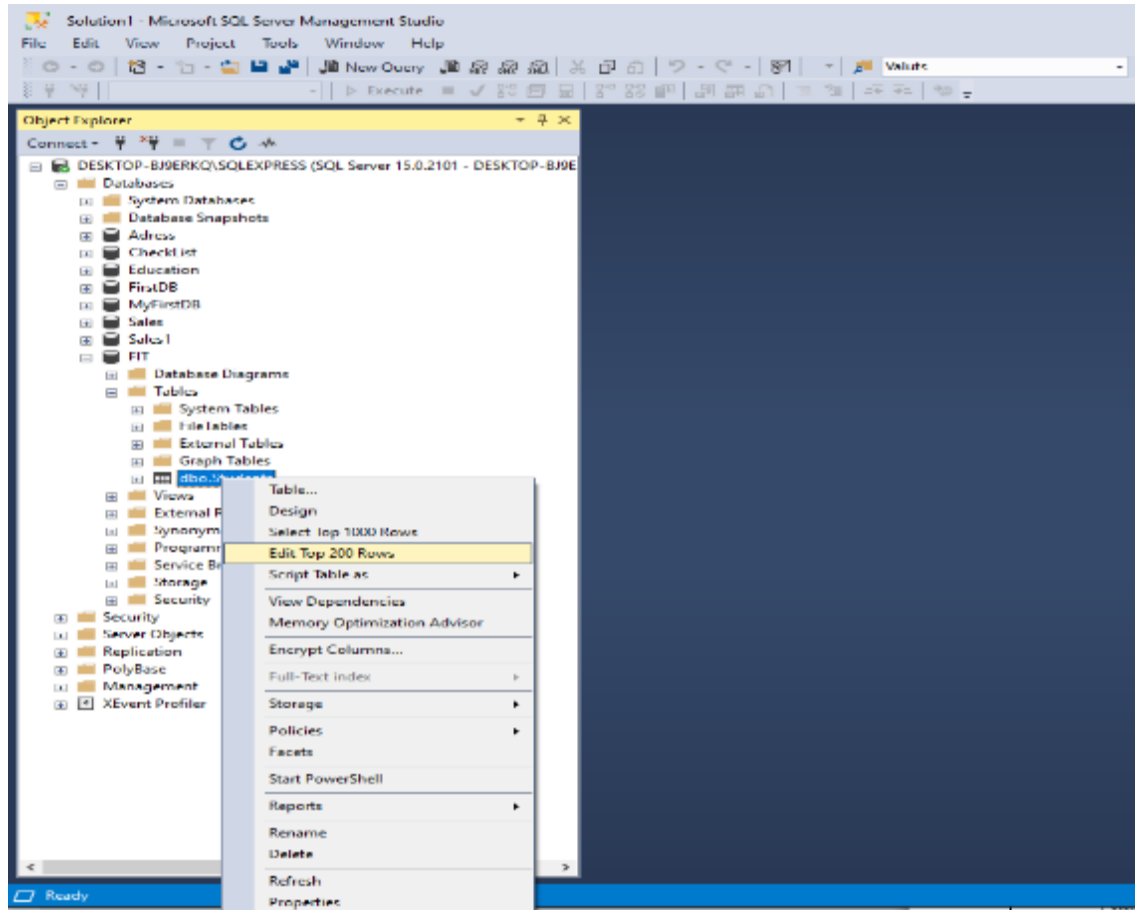


Рисунок 2.17 – Контекстне меню з опціями для управління таблицею.

Опція Edit Top 200 Rows

Для внесення початкових даних у створену таблицю необхідно додати записи, заповнивши значення у відповідних комірках. Приклад заповнення таблиці шляхом введення інформації про студентів наведено на рис. 2.18.

DESKTOP-BJ9ERKQ\S...FIT - dbo.Students				
	Id	Name	Department	Year
	1	Ільченко Віктор	ІПЗ	1
▶*	NULL	NULL	NULL	NULL

Рисунок 2.18 – Заповнення таблиці Students даними про студентів

Одна стандартна таблиця може містити до 1024 стовпців, а кількість записів (рядків) у таблиці обмежена лише обсягом виділеної для неї пам'яті або вільним дисковим простором на сервері, якщо Maxsize не задано.

2.2.6 Логічна структура і фізична реалізація баз даних MS SQL Server. Дані в SQL Server організовані в базах даних, структура яких розглядається на двох рівнях: логічному та фізичному. Логічна структура бази даних визначає організацію даних, включаючи структуру таблиць, зв'язки між ними, перелік користувачів, збережені процедури, правила, значення за замовчуванням та інші об'єкти бази даних. Фізична структура бази даних охоплює опис файлів і груп файлів бази даних, журналу транзакцій, початковий розмір файлів, крок збільшення бази даних, її максимальний розмір, параметри конфігурації та інші фізичні характеристики. Усі логічні компоненти бази даних SQL Server називаються об'єктами, які класифікуються на одинадцять типів. Для об'єктів бази даних SQL Server можна встановити різні правила доступу для користувачів.

Функціональне призначення об'єктів бази даних сервера SQL Server коротко визначено в таблиці 2.1.

Таблиця 2.1. **Об'єкти бази даних сервера SQL Server**

Table	Таблиця являє собою структуровану сукупність даних, організованих у вигляді матриці, де кожен рядок відображає окремий запис, а кожен стовпець – атрибут. Записи складаються зі значень, що характеризують конкретний об'єкт. Стовпці, у свою чергу, містять набір атрибутів, що належать до певної предметної області. Окремі стовпці можуть бути обчислюваними, тобто їх значення визначаються за допомогою заданих формул.
User-defined data type	Користувацький тип даних, який створюється на основі системних. Ім'я нового типу має бути унікальним в межах власника.

View	Подання є віртуальною таблицею, вміст якої визначається запитом. Ця таблиця не містить даних, а тільки їх представляє, можливо з декількох таблиць. Дані з перегляду не зберігаються в базі даних. Фізично перегляд реалізується у вигляді запиту SELECT. Перегляди використовуються в наступних випадках: а) для обмеження доступу користувачів до певних рядків таблиці; б) для обмеження доступу користувачів до певних стовпців таблиці; с) для представлення даних стовпців різних таблиць у вигляді одного об'єкта; д) для перегляду інформації, що є результатом перетворення даних стовпців.
Stored procedure	Збережена процедура являє собою набір команд Transact-SQL, що об'єднані в єдиний модуль. Кожна процедура має унікальне ім'я в межах бази даних, за яким вона викликається. Процедури можуть викликати одна одну. SQL Server містить велику кількість вбудованих процедур, що називаються системними і мають префікс «sp_» в імені.
Trigger	Тригер представляє собою спеціальну збережену процедуру, що автоматично запускається при додаванні, зміні або видаленні даних з таблиці. Дії, що виконуються в одному тригері, можуть викликати інші тригери (вкладені тригери).
Index	Індекс являє собою структуру даних, пов'язану з таблицею або поданням, що призначена для оптимізації пошуку інформації. Індекс створюється на основі одного або декількох стовпців, які називаються індексованими. Він

	містить відсортовані значення індексованих стовпців з посиланнями на відповідні рядки таблиці або подання. Оскільки алгоритми пошуку в відсортованих даних є значно ефективнішими, ніж у невідсортованих, індекси значно прискорюють операції пошуку. Але не варто створювати індекс для стовпців, які мають мало різних значень, наприклад, для стовпця «стать», де є лише «чоловік» і «жінка».
Rule	Правило – це механізм, що використовується для обмеження допустимих значень, які можуть бути збережені у стовпці таблиці або в користувацькому типі даних. Одне правило може бути застосоване до багатьох стовпців різних таблиць і користувацьких типів даних у межах поточної бази даних. Правило створюється за допомогою команди CREATE RULE і пов'язується з об'єктом бази даних через процедуру spbindrule. Важливо зазначити, що правила збережені в Transact-SQL виключно для забезпечення сумісності з попередніми версіями сервера.
Constraint	Обмеження цілісності – це спосіб автоматично перевіряти, чи правильні дані. Вони важливіші, ніж тригери, правила та значення за замовчуванням. Існує п'ять різних видів обмежень цілісності, кожен з яких має свої функції та застосування.
Default	Значення за замовчуванням – це значення, яке автоматично присвоюється елементу стовпця таблиці при додаванні нового рядка, якщо для цього стовпця не було явно вказано інше значення. Значеннями за замовчуванням можуть бути константи, вбудовані функції або математичні вирази, що повертають конкретні значення.

Function	Функція – це програмний модуль, призначений для виконання типових операцій над даними та повернення значення певного типу. Виклик функції здійснюється шляхом зазначення її імені в будь-якому виразі мови Transact-SQL. Існують вбудовані функції, які є невід’ємною частиною середовища програмування сервера та виконують заздалегідь визначені команди (наприклад, COUNT, SUM, MIN, MAX). Вони не можуть бути змінені користувачем. Також існують функції користувача, які створюються користувачем за допомогою мови Transact-SQL для реалізації власних алгоритмів.
Procedure	Процедура (або збережена процедура) – це набір команд SQL, які зберігаються в базі даних.

2.2.7 Обмеження цілісності. Обмеження цілісності – це механізм, що забезпечує автоматичну перевірку даних на відповідність заданим умовам. Вони мають вищий пріоритет порівняно з тригерами, правилами та значеннями за замовчуванням. Існує п’ять типів обмежень цілісності, що відрізняються функціональністю та сферою застосування (таблиця 2.2)

Таблиця 2.2. **Обмеження цілісності**

<i>NULL</i>	Це налаштування, яке визначає, чи може стовпець бази даних або користувацький тип даних містити порожні значення (NULL). Можна дозволити (NULL) або заборонити (NOT NULL) зберігання таких значень.
<i>CHECK</i>	Це обмеження, яке застосовується до стовпця і контролює, які значення можна в ньому зберігати. Воно працює шляхом перевірки логічного виразу для кожного значення, що вводиться. Якщо вираз істинний (TRUE), значення зберігається, інакше – відхиляється, і видається помилка. Для одного стовпця можна задати кілька таких

	перевірок.
UNIQUE	Це обмеження, яке застосовується до стовпця, гарантує, що всі значення в ньому будуть унікальними. На відміну від первинного ключа (PRIMARY KEY), воно дозволяє зберігати порожні значення (NULL).
PRIMARY KEY	Це обмеження, яке діє на рівні стовпця або таблиці, і забезпечує унікальну ідентифікацію кожного рядка. Він може складатися з одного або кількох стовпців, але жоден з них не може містити значення NULL. Кожна таблиця може мати лише один PRIMARY KEY.
FOREIGN KEY	Це обмеження, яке встановлює зв'язок між таблицями, посилаючись на первинний ключ іншої таблиці. Таблиця, що містить зовнішній ключ, називається залежною, а таблиця з первинним ключем – головною. Цілісність даних забезпечується шляхом заборони вставки рядків у залежну таблицю без відповідного значення в головній таблиці та заборони видалення рядків з головної таблиці, якщо вони використовуються в залежній.

2.2.8 Основні складові SQL. SQL підтримує різноманітні типи даних для баз даних і змінних, а також дає змогу використовувати константи, стандартні та обмежені ідентифікатори. За допомогою SQL можна створювати складні арифметичні та логічні вирази, використовуючи константи, змінні, імена стовпців таблиць, функції, підзапити та умовні вирази, а також групувати вирази за допомогою круглих дужок.

SQL надає повний набір SQL-команд для створення, зміни та видалення баз даних та їх об'єктів, а також для визначення запитів на введення, обробку та вилучення даних. Для управління потоком виконання програми SQL включає керуючі програмні структури, які дозволяють визначати умови та порядок виконання команд у заданій послідовності або пакеті команд.

Крім того, SQL підтримує вбудовані (системні) та визначені користувачем функції, а також вбудовані (системні) та визначені користувачем збережені процедури, що дає змогу створювати складні та ефективні програми для роботи з базами даних.

У базах даних, окрім функцій та процедур, можна зберігати набори команд, які називаються скриптами або сценаріями. Сценарії дозволяють легко копіювати структуру бази даних між серверами або переносити структуру таблиць та інших об'єктів між різними базами даних. Скрипти зберігаються у звичайних текстових файлах.

Використання функцій та процедур у базах даних значно зменшує обсяг даних, що передаються між клієнтом та сервером, що покращує продуктивність системи. Наявність вихідного коду для цих компонентів спрощує підтримку та оновлення програм. Як правило, основні завдання бізнесу та алгоритми обробки даних реалізуються на сервері бази даних і доступні користувачам через набір функцій та процедур, які і є інтерфейсом для роботи з даними. Щоб забезпечити безпеку та цілісність даних, програми зазвичай не мають прямого доступу до даних. Вся робота ведеться через цей інтерфейс.

Такий підхід робить зміни в алгоритмах обробки даних дуже простими та дає змогу розширювати систему без необхідності змінювати саму програму. Достатньо оновити збережену процедуру на сервері бази даних, і всі зміни миттєво стануть доступними для всіх користувачів мережі.

2.2.9 Константи. Мова SQL підтримує широкий спектр констант, які використовуються для представлення різних типів даних. До них належать бітові константи, які можуть мати значення 0 або 1, та логічні константи, що представляють істину (TRUE) або хибність (FALSE). Для представлення двійкових даних використовуються шістнадцяткові константи, наприклад, 0x9E70DA.

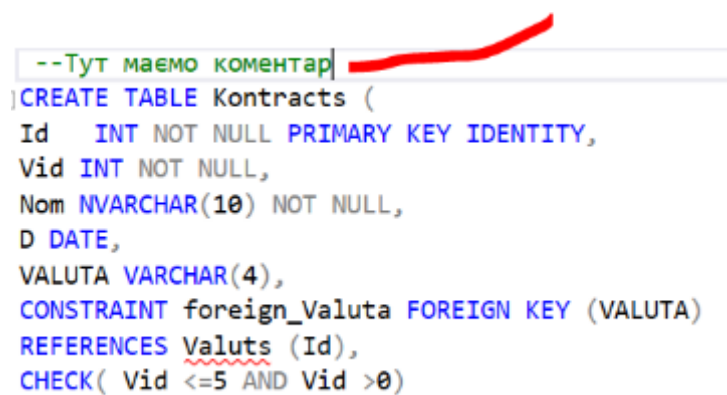
SQL також підтримує символічні константи, які можуть бути укладені в одинарні або подвійні лапки, а також константи Unicode, що починаються з префікса N. Числові константи включають цілі числа, числа з фіксованою та

плаваючою точкою. Для представлення дат використовуються константи в різних форматах, таких як «April 15, 2003», «4/15/2003» або «20031207», а для часу – константи, що вказують години, хвилини, секунди та мілісекунди, або використовують 12-годинний формат з АМ/РМ. Крім того, T-SQL підтримує грошові константи, які представляють грошові суми з використанням символів валют, таких як \$ або ?.

2.2.10 Коментарі. У мові програмування, як і в будь-якій іншій, коментарі відіграють важливу роль у розумінні коду. Вони служать для пояснення того, що робить певна частина коду, і допомагають іншим розробникам (або вам самим у майбутньому) розібратися в його логіці. У мові SQL існують два основних способи додавання коментарів: стічні та блокові.

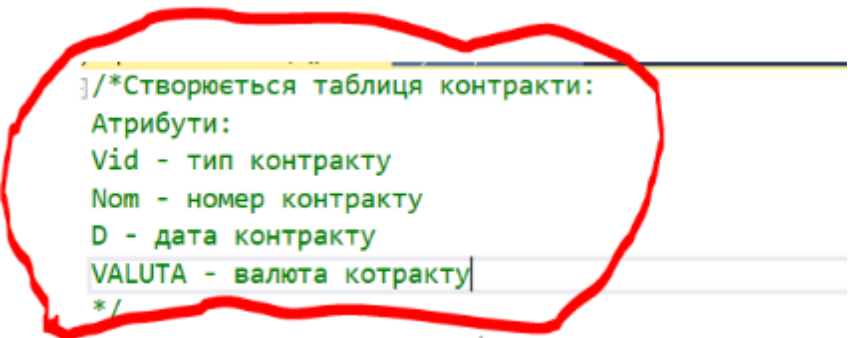
Стічні коментарі використовуються для коментування одного рядка коду. Вони починаються з двох символів мінуса «--», і все, що йде після цих символів до кінця рядка, ігнорується компілятором (рис. 2.19). Це зручно для додавання коротких пояснень або тимчасового виключення частини коду.

Блокові коментарі, з іншого боку, дозволяють коментувати кілька рядків коду одночасно. Вони починаються з символів «/*» і закінчуються символами «*/». Все, що знаходиться між цими символами, вважається коментарем і не виконується (рис. 2.20). Це корисно для додавання довгих пояснень, тимчасового виключення великих блоків коду або для документування складних алгоритмів.



```
--Тут маємо коментар
CREATE TABLE Kontracts (
  Id INT NOT NULL PRIMARY KEY IDENTITY,
  Vid INT NOT NULL,
  Nom NVARCHAR(10) NOT NULL,
  D DATE,
  VALUTA VARCHAR(4),
  CONSTRAINT foreign_Valuta FOREIGN KEY (VALUTA)
  REFERENCES Valuts (Id),
  CHECK( Vid <=5 AND Vid >0)
```

Рисунок 2.19 – Стічний коментар SQL



```

}/*Створюється таблиця контракти:
Атрибути:
Vid - тип контракту
Nom - номер контракту
D - дата контракту
VALUTA - валюта котракту
*/
} CREATE TABLE Kontracts (
  Id INT NOT NULL PRIMARY KEY IDENTITY,
  Vid INT NOT NULL,
  Nom NVARCHAR(10) NOT NULL,
  D DATE,
  VALUTA VARCHAR(4),
  CONSTRAINT foreign_Valuta FOREIGN KEY (VALUTA)
  REFERENCES Valuts (Id),
  CHECK( Vid <=5 AND Vid >0)
);

```

Рисунок 2.20 – Блоковий коментар SQL

Таким чином, використання коментарів у SQL допомагає зробити код більш зрозумілим, читабельним і підтримуваним. Вони є невід’ємною частиною процесу розробки і повинні використовуватися для пояснення складних або неочевидних частин коду.

2.2.11 Імена. Кожен елемент бази даних потребує унікального імені, яке використовується в SQL-запитах для його ідентифікації. Щоб забезпечити однозначність, система вимагає, щоб кожному об’єкту бази даних було присвоєно унікальне ім’я, яке можна використовувати в командах. Окрім імен, які дає користувач, сервер автоматично створює внутрішні, унікальні ідентифікатори для кожного об’єкта. Ці внутрішні ідентифікатори, такі як «PKTableX 014543FA», гарантують, що кожен елемент бази даних може бути однозначно визначений на рівні системи.

У мові SQL імена об’єктів бази даних, такі як таблиці, стовпці або процедури, називаються ідентифікаторами. Існує два основних типи ідентифікаторів: стандартні та обмежені.

Стандартні ідентифікатори, наприклад «TableX» або «KeyCol», мають певні обмеження: їхня довжина повинна бути від 1 до 128 символів, вони не можуть збігатися з ключовими словами мови SQL, а перший символ може бути латинською або українською літерою, знаками «#», «##», «@», «@@» або підкресленням «_». Наступні символи можуть також включати десяткові цифри.

Обмежені ідентифікатори, наприклад «[My Table]» або «Order», дозволяють використовувати символи, які заборонені в стандартних ідентифікаторах, включаючи ключові слова. Вони укладаються в квадратні дужки або подвійні лапки.

В SQL Server кожен об'єкт бази даних пов'язаний з певним користувачем, базою даних і сервером. Це дає змогу створити повне ім'я об'єкта, яке однозначно визначає його розташування. Повне ім'я має таку структуру: «[[[server].[database].[owner_name]].object_name]».

Існує кілька способів посилання на об'єкти бази даних, залежно від того, наскільки детально потрібно вказати їхнє розташування. Наприклад, можна використовувати лише ім'я об'єкта, якщо він знаходиться в поточній базі даних, або повне ім'я, якщо потрібно послатися на об'єкт в іншій базі даних або на іншому сервері.

Щоб послатися на конкретний стовпець таблиці або подання з іншого серверу, потрібно використати повне ім'я цього стовпця таблиці: «server.database.owner_name.table_name.column_name».

2.2.12 Вирази. В мові програмування, як-от Transact-SQL (T-SQL), для виконання різноманітних операцій використовуються оператори виразів. Ці оператори дозволяють виконувати арифметичні обчислення, порівняння значень, логічні операції та інші дії.

Серед операторів виразів можна виділити унарні оператори, такі як «+» та «-», які застосовуються до одного операнда. Також існують бінарні арифметичні оператори, які виконують арифметичні дії з двома операндами, наприклад, «+», «-», «*», «%». Оператор присвоювання «=», як впливає з

назви, присвоює значення змінній. Крім того, існує строкова операція конкатенації «+», яка об'єднує два рядки в один.

Для порівняння значень використовуються оператори порівняння, такі як «=», «>», «<», «<=», «>=», «!=», «<>», «!<», «!>». Логічні оператори, такі як «NOT», «AND», «OR», «ALL», «ANY», «BETWEEN», «EXIST», «IN», «LIKE», «SOME», дозволяють виконувати логічні операції та перевіряти умови. І, нарешті, бітові оператори «&», «|», «^» використовуються для виконання побітових операцій.

Константи, змінні, операнди та вирази є основними будівельними блоками для написання команд, функцій та збережених процедур в SQL Server. Вони визначають потужність та гнучкість системи програмування. Команди дозволяють створювати, змінювати та видаляти бази даних та їх об'єкти, формувати складні запити на введення, обробку та вилучення даних, а також виконувати адміністративні та обслуговуючі функції. Функції та процедури, в свою чергу, реалізують різноманітні алгоритми обробки даних та виконують службові функції сервера.

При складанні SQL-запитів часто використовуються спеціальні логічні оператори, які дозволяють виконувати складні умови та порівняння. Один з таких операторів – це порівняння скалярного виразу з результатами підзапиту. Якщо ми використовуємо «ALL», то вираз вважається істинним лише тоді, коли він відповідає всім значенням, що повертає підзапит. Якщо ж замість «ALL» використовується «SOME» або «ANY», то умова вважається істинною, якщо вираз відповідає хоча б одному значенню з підзапиту.

Оператор «BETWEEN» перевіряє, чи знаходиться значення виразу в заданому діапазоні. «NOT BETWEEN» робить протилежне – повертає істину, якщо значення не знаходиться в діапазоні.

Оператор «EXISTS» перевіряє, чи повертає підзапит хоча б один рядок. Якщо так, то «EXISTS (підзапит)» повертає істину.

Оператор «IN» перевіряє, чи збігається значення виразу з одним із значень у підзапиті або списку значень. «NOT IN» робить протилежне – повертає істину, якщо значення не збігається з жодним із значень.

Оператор «LIKE» використовується для порівняння рядків з шаблоном. Він дає змогу використовувати спеціальні символи: «%» (будь-яка кількість символів), «_» (один будь-який символ), «[символи]» (один із зазначених символів), «[^символи]» (будь-який символ, крім зазначених).

2.2.13 Типи даних. SQL є строго типізованою мовою. Вона підтримує широкий спектр типів даних (таблиця 2.3), які використовуються для зберігання різних видів інформації в базі даних.

Таблиця 2.3. **Типи даних SQL**

binary(n))	Масив байтів фіксованого розміру, що займає в пам'яті n +4 байт. Синтаксис визначення : binary[(n)], де n – число байтів від 1 до 8000. n за умовчанням дорівнює 30.
image	Масив двійкових даних розміром до 2 Гб.
char(n)	Масив 8-розрядних символів фіксованого розміру. Синтаксис визначення : char[(n)], де n – число символів від 1 до 8000. Якщо n не задається у визначенні даних або в інструкції оголошення змінної, довжина за умовчанням дорівнює 1. Якщо при використанні функцій CAST і CONVERT n не задається, довжина за умовчанням дорівнює 30. Синонімом за стандартом ISO для типу char являється character.
varchar(n)	Аналогічний char, але при зберіганні рядків пам'яті відводиться по довжині рядка, а не завжди n байт, але не більше n. Синтаксис визначення : varchar[(n)], де n – число символів від 1 до 8000. Якщо n не задається у визначенні даних або в інструкції оголошення змінної, довжина за умовчанням дорівнює 1. Якщо при використанні функцій

	CAST і CONVERT n не задається, довжина за умовчанням дорівнює 30. Синонімами за стандартом ISO для типу varchar являються типи char varying або character varying.
Nchar(n)	Масив 16-розрядних символів фіксованого розміру. Символи зберігаються в кодуванні UNICODE UCS - 2. Синтаксис визначення : nchar[(n)], де n – число символів від 1 до 4000. Якщо n не задається у визначенні даних або в інструкції оголошення змінної, довжина за умовчанням дорівнює 1. Якщо при використанні функцій CAST і CONVERT n не задається, довжина за умовчанням дорівнює 30. Синонімами за стандартом ISO для типу nchar являються типи national char і national character.
Nvarchar r(n)	Аналогічний nchar за винятком того, що при зберіганні рядків пам'яті відводиться по довжині рядка, а не завжди n байт, але не більше n. Синтаксис визначення : nvarchar[(n)], де n – число символів від 1 до 4000. Якщо n не задається у визначенні даних або в інструкції оголошення змінної, довжина за умовчанням дорівнює 1. Якщо при використанні функцій CAST і CONVERT n не задається, довжина за умовчанням дорівнює 30. Синонімами за стандартом ISO для типу nvarchar являються типи national char varying і national character varying.
Text	Масив 8-розрядних символів розміром до 2 Гб.
Ntext	Масив 16-розрядних символів розміром до 2 Гб. Символи зберігаються в кодуванні UNICODE UCS - 2. Синонімом за стандартом ISO для типу ntext являється national text.
Int	32-х розрядне ціле зі знаком (4 байти) в діапазоні від

	-2147483648 до 2147483647.
Bigint	64-х розрядне ціле зі знаком (8 байт) в діапазоні від -9223372036854775808 до 9223372036854775807.
decimal чи numeric	Дійсне число з фіксованою комою. Назва може бути скорочена до dec. Синтаксис визначення decimal(p, s), де p – точність (кількість десяткових розрядів числа) від 1 до 38, s – масштаб (кількість десяткових розрядів після коми) від 0 до p. У пам'яті значення цього типу займають різне число байт, залежне від точності при точності 1-9 знаків – 5 байт, при точності 10-19 знаків – 9 байт, при точності 20-28 знаків – 13 байт, при точності 29-38 знаків – 17 байт
Float[(n)]	Дійсне число з плаваючою комою. Синтаксис визначення : float[(p)], де p – точність (число знаків основи) від 1 до 53. При точності 1-24 знаків для зберігання числа виділяється 4 байти пам'яті, а при точності 25-53 знаків – 8 байт. За умовчанням p = 53.
Real	Плаваючий тип, який є аналогом float (240).
Datetim e	Зберігає дату і час, займає 8 байт пам'яті, де перші 4 байти зберігають число днів до або після початкової дати (1 січня 1900), а що залишилися 4 байти – число мілісекунд від півночі. Доступний діапазон дат від 1 січня 1753 року до 31 грудня 9999 року з точністю до 1/3 сантисекунди (0,003333 секунд).
Smalldat etime	Зберігає дату в діапазоні від 1 січня 1900 року до 6 червня 2079 року з точністю до хвилини і займає 4 байти пам'яті, де перші 2 байти зберігають число днів після початкової дати, а що залишилися 2 байти – число хвилин від півночі.
Bit	Бітовий (логічний) тип зі значеннями 0 і 1; для

	зберігання виділяється 1 розряд байта пам'яті.
Timestampr	Спеціальний тип даних, призначений для зберігання часу оновлення запису. Зараз тип еквівалентний datetime або binary(чи varbinary(8), якщо допустимий null). Може використовуватися як тип колонки таблиці (атрибуту відношення), причому така колонка має бути єдиною і не мати спеціальної назви. Значення атрибуту оновлюється поточним часом при кожній операції вставки або оновлення для запису (кортежу).
Sysname	тип даних для зберігання імен об'єктів бази даних; аналог nvarchar (128). Sql_variant-варіантний тип даних для зберігання даних будь-якого типу, крім text, ntext, image, timestamp; для отримання інформації про природу збережених даних використовується функція SQL_VARIANT_PROPERTY ().
Table	Тип, що дозволяє в змінній зберігати таблицю (відношення). Призначений для використання в процедурах, що зберігаються, для запису проміжних результатів обробки. Не може бути типом колонки таблиці (атрибуту відношення

2.2.14 Загальний синтаксис створення бази даних. Створення бази даних в SQL – це ключовий крок у роботі з реляційними базами даних. Процес може дещо відрізнятись залежно від конкретної системи управління базами даних (СУБД), яку ви використовуєте (наприклад, MySQL, PostgreSQL, SQL Server тощо), але загальні принципи залишаються схожими.

Для завдання синтаксису команди буде використовуватися метамова, заснована на нормальних формах Бекуса Наура (БНФ). Вона надає точний та однозначний спосіб визначення граматики, що дає змогу описувати, як будувати коректні вирази мови за допомогою набору правил продукції. Ці правила використовують термінальні символи, такі як літери, цифри та знаки

пунктуації, а також нетермінальні символи, що представляють синтаксичні категорії. БНФ підтримує рекурсію, що дає змогу описувати складні структури.

БНФ знаходить широке застосування в описі синтаксису мов програмування, таких як C, Java та Python, а також у визначенні форматів даних, таких як XML та JSON. Вона є незамінним інструментом при створенні компіляторів та інтерпретаторів, а також при аналізі та обробці текстів.

Переваги БНФ полягають у її точності та однозначності опису синтаксису, що дає змогу автоматично обробляти граматику. Завдяки цим властивостям БНФ широко використовується в комп'ютерних науках.

БНФ визначає, що ключові слова без всякої зміни переходять в саму команду; метазмінні, написані курсивом, повинні бути замінені програмними іменами; роздільники (кома, так само, апостроф і т.д.) також переходять в якості роздільників в команду.

Метасинтаксичні знаки мають наступний сенс :

`:: =` – є за визначенням;

`|` – вибір альтернативи;

`[]` – можлива відсутність частини визначення;

`{ }` – об'єднання частин визначення для вибору або повторення;

`[... n]` – повторення попередньої частини 1, 2, ..., n раз з роздільником кома для цієї частини (роздільник може бути будь-який);

`<...>` – метазмінна, яка має своє визначення.

Загальний синтаксис створення бази даних наступний:

`CREATE DATABASE database_name`

`[ CONTAINMENT = { NONE | PARTIAL } ]`

`[ ON`

`[ PRIMARY ] <filespec> [ ,...n ]`

`[ , <filegroup> [ ,...n ] ]`

`[ LOG ON <filespec> [ ,...n ] ]`

`]`

`[ COLLATE collation_name ]`

[WITH <option> [...n]]

[;]

<option> ::=

{

 FILESTREAM (<filestream_option> [...n])

 | DEFAULT_FULLTEXT_LANGUAGE = { lcid | language_name |
language_alias }

 | DEFAULT_LANGUAGE = { lcid | language_name | language_alias }

 | NESTED_TRIGGERS = { OFF | ON }

 | TRANSFORM_NOISE_WORDS = { OFF | ON }

 | TWO_DIGIT_YEAR_CUTOFF = <two_digit_year_cutoff>

 | DB_CHAINING { OFF | ON }

 | TRUSTWORTHY { OFF | ON }

 | PERSISTENT_LOG_BUFFER=ON (DIRECTORY_NAME='path-to-
directory-on-a-DAX-volume')

 | LEDGER = { ON | OFF }

}

<filestream_option> ::=

{

 NON_TRANSACTED_ACCESS = { OFF | READ_ONLY | FULL }

 | DIRECTORY_NAME = 'directory_name'

}

<filespec> ::=

{

(

 NAME = logical_file_name ,

 FILENAME = { 'os_file_name' | 'filestream_path' }

```
[ , SIZE = size [ KB | MB | GB | TB ] ]
[ , MAXSIZE = { max_size [ KB | MB | GB | TB ] | UNLIMITED } ]
[ , FILEGROWTH = growth_increment [ KB | MB | GB | TB | % ] ]
)
}
```

<filegroup> ::=

```
{
FILEGROUP filegroup name [ [ CONTAINS FILESTREAM ] [ DEFAULT ] |
CONTAINS MEMORY_OPTIMIZED_DATA ]
```

```
<filespec> [ ,...n ]
```

```
},
```

де:

database_name – ім'я, що присвоюється новій базі даних, має бути унікальним у межах SQL Server і відповідати правилам іменування ідентифікаторів. Довжина імені бази даних обмежена 128 символами, якщо не задано конкретне логічне ім'я для файлу журналу. У випадках, коли логічне ім'я не задано, SQL Server автоматично створює імена файлів журналу на основі імені бази даних, що, своєю чергою, обмежує довжину імені бази даних до 123 символів, щоб уникнути перевищення 128-символьного обмеження для згенерованого логічного імені файлу;

CONTAINMENT = { NONE | PARTIAL } – вказує стан залежності бази даних від сервера. Параметр CONTAINMENT, що використовується при створенні бази даних в Microsoft SQL Server, визначає ступінь її автономності. Існують два основних значення цього параметра. Якщо встановлено значення NONE, що є стандартним, база даних функціонує як звичайна, залежна від метаданих та об'єктів екземпляра SQL Server. Це означає, що запити можуть використовувати метадані та об'єкти сервера, але гарантувати перенесення такої бази даних між різними екземплярами SQL Server неможливо. З іншого боку, значення PARTIAL вказує на частково автономну базу даних. У цьому

випадку зменшується залежність від екземпляра SQL Server, що покращує можливість перенесення бази даних між серверами. Крім того, встановлюється суворіша перевірка посилань на метадані та об'єкти сервера, використовуються колації бази даних замість колацій сервера, а користувачі можуть проходити автономну автентифікацію безпосередньо в базі даних. Основна мета використання PARTIAL CONTAINMENT полягає у створенні більш незалежних баз даних, які можна легко переносити між серверами, що особливо корисно для хмарних середовищ та сценаріїв, де важлива висока доступність;

ON – вказує, що дискові файли, які використовуються для зберігання розділів бази даних, файли даних, визначені явно. ON є обов'язковим, якщо після нього йде список елементів <filespec>, розділених комами, які визначають файли даних для основної файлової групи. Після списку файлів у основній групі файлів може слідувати необов'язковий список елементів <filegroup>, розділених комами, які визначають групи файлів користувача та їхні файли.

PRIMARY – використовується для визначення місця розташування основного файлу даних, де зберігаються ключові дані. Цей параметр не лише вказує шлях до файлу з розширенням .mdf, але й визначає, до якої файлової групи належить цей файл, зазвичай до головної, де розміщуються системні таблиці та об'єкти користувача. Крім того, при його використанні можна задати початковий розмір файлу та налаштувати автоматичне збільшення його обсягу.

LOG ON – визначає розташування та характеристики файлу журналу транзакцій, який є критично важливим компонентом для запису всіх змін у базі даних. Цей параметр дає змогу вказати фізичний шлях та ім'я файлу журналу транзакцій, зазвичай з розширенням .ldf, а також налаштувати його розмір, включаючи початковий розмір, максимальний розмір (або необмежений розмір) та параметри автоматичного збільшення;

COLLATE *collation_name* – визначає набір правил, що регулюють зберігання, порівняння та впорядкування символічних (текстових) даних. Колація впливає на порядок сортування символів, визначаючи, як вони

розташовуються при сортуванні та порівнянні. Вона також визначає чутливість до регістру та наголосів, тобто, чи вважаються 'а' та 'А' або 'é' та 'е' однаковими. Крім того, колація встановлює кодування символів, визначаючи, який набір символів підтримується, наприклад, Unicode чи ASCII, та визначає мовні правила, що регулюють обробку специфічних для різних мов символів та їх комбінацій;

`WITH NON_TRANSACTED_ACCESS = { OFF | READ_ONLY | FULL }` – визначає режим доступу зовнішніх процесів до даних без використання транзакцій. Це особливо актуально в ситуаціях, коли необхідний швидкий доступ до інформації без гарантій ACID, наприклад, для аналітичних запитів або експорту даних. Значення `OFF`, яке є стандартним, повністю блокує будь-який не транзакційний доступ, вимагаючи виконання всіх операцій у рамках транзакцій. Значення `READ_ONLY` дозволяє зовнішнім процесам лише читати дані без транзакцій, що корисно для швидкого отримання інформації без її зміни. Значення `FULL` надає можливість як читати, так і змінювати дані поза транзакціями, але його використання вимагає особливої обережності, оскільки неконтрольовані зовнішні процеси можуть призвести до пошкодження даних;

`DEFAULT_FULLTEXT_LANGUAGE = <lcid> | <language name> | <language alias>` – визначає мову, яка буде використовуватися за замовчуванням для повнотекстового пошуку в цій базі даних. Це функція SQL Server, яка дає змогу шукати слова та фрази в текстових даних, що зберігаються в базі даних. Для ефективного повнотекстового пошуку SQL Server повинен знати, якою мовою написаний текст. Параметр `DEFAULT_FULLTEXT_LANGUAGE` встановлює мову, яка буде використовуватися для повнотекстового пошуку, якщо не вказано іншу мову для конкретного стовпця або запиту, `<lcid>` – ідентифікатор локалі (LCID), який однозначно визначає мову, `<language name>` – назва мови (наприклад, «English», «Ukrainian»). `<language alias>` – псевдонім мови (наприклад, «en», «uk»).

`DEFAULT_LANGUAGE = <lcid> | <language name> | <language alias>` – оператор визначає мову за замовчуванням для цієї бази даних. Це означає, що

системні повідомлення, такі як повідомлення про помилки, попередження та інформаційні повідомлення, відображатимуться цією мовою. Також якщо користувач не вкаже іншу мову для свого сеансу, використовуватиметься мова, встановлена за замовчуванням для бази даних. Формат дати та часу залежатиме від обраної мови. Правила сортування та порівняння рядків також залежать від мови. При створенні бази даних ви можете вказати мову за допомогою `lcid`: ідентифікатор локалі (наприклад, 1058 для української мови); `language name`: назву мови (наприклад, 'Ukrainian'); `language alias`: псевдонім мови.

`NESTED_TRIGGERS = { OFF | ON }` – визначає, чи можуть тригери викликати інші тригери. `NESTED_TRIGGERS = ON`: Дозволяє тригерам викликати інші тригери. Це означає, що коли спрацьовує один тригер, він може виконати операцію, яка, своєю чергою, спричинить спрацьовування іншого тригера. Такий підхід може бути корисним для реалізації складних каскадних операцій. `NESTED_TRIGGERS = OFF`: Забороняє тригерам викликати інші тригери. Якщо тригер намагається виконати операцію, яка спричинить спрацьовування іншого тригера, ця операція буде проігнорована. Це може бути корисним для запобігання нескінченним циклам тригерів або для спрощення налагодження.

Оператор `TRANSFORM_NOISE_WORDS = { OFF | ON }` при створенні бази даних у SQL Server визначає, чи будуть «шумові слова» (noise words) трансформовані під час повнотекстового пошуку. «Шумові слова» – це слова, які часто зустрічаються в тексті, але не несуть суттєвого змістовного навантаження (наприклад, «і», «а», «в», «на», «про» тощо). Вони можуть сповільнювати повнотекстовий пошук і збільшувати розмір індексу. `TRANSFORM_NOISE_WORDS = ON`: SQL Server буде трансформувати «шумові слова» під час повнотекстового пошуку. Це означає, що вони будуть оброблені спеціальним чином, що може покращити продуктивність пошуку. `TRANSFORM_NOISE_WORDS = OFF`: «шумові слова» не будуть трансформовані. Вони будуть оброблятися як звичайні слова під час повнотекстового пошуку.

`TWO_DIGIT_YEAR_CUTOFF` – визначає поріг для інтерпретації двозначних років. Коли ви вводите двозначний рік (наприклад, '99'), SQL Server повинен визначити, до якого століття він належить. `TWO_DIGIT_YEAR_CUTOFF` вказує, який рік вважається початком 100-річного діапазону, в якому інтерпретуються двозначні роки. Якщо двозначний рік менший або дорівнює значенню `TWO_DIGIT_YEAR_CUTOFF`, він інтерпретується як рік у тому ж столітті, що й `TWO_DIGIT_YEAR_CUTOFF`. Якщо двозначний рік більший за значення `TWO_DIGIT_YEAR_CUTOFF`, він інтерпретується як рік у попередньому столітті. Наприклад, якщо `TWO_DIGIT_YEAR_CUTOFF` встановлено на 2049 (значення за замовчуванням): '49' інтерпретується як 2049; '50' інтерпретується як 1950; '99' інтерпретується як 1999. Ви можете встановити `TWO_DIGIT_YEAR_CUTOFF` на будь-яке значення між 1753 і 9999.

`DB_CHAINING { OFF | ON }` – визначає, чи може база даних брати участь у міжбазових ланцюгах власності. Міжбазові ланцюги власності дозволяють користувачам виконувати запити, які звертаються до об'єктів у кількох базах даних, без необхідності мати явні дозволи на доступ до кожної бази даних. Це працює, якщо власник об'єктів у кожній базі даних є одним і тим же користувачем. `DB_CHAINING = ON`: дозволяє базі даних брати участь у міжбазових ланцюгах власності. Це означає, що якщо користувач має дозволи на доступ до об'єкта в одній базі даних, і цей об'єкт звертається до об'єкта в іншій базі даних, власником якого є той самий користувач, то користувач зможе отримати доступ до об'єкта в другій базі даних без явних дозволів. `DB_CHAINING = OFF`: забороняє базі даних брати участь у міжбазових ланцюгах власності. У цьому випадку користувачі повинні мати явні дозволи на доступ до об'єктів у кожній базі даних, до якої вони звертаються.

`TRUSTWORTHY { OFF | ON }` – визначає, чи довіряє екземпляр SQL Server базі даних та її вмісту. `TRUSTWORTHY = ON`: дозволяє базі даних отримувати доступ до ресурсів за межами бази даних, наприклад, до інших баз даних або до зовнішніх ресурсів. Це може бути необхідно для певних функцій,

таких як CLR-інтеграція або модулі, які виконуються з використанням EXECUTE AS. CLR-інтеграція (Common Language Runtime) в SQL Server – це технологія, яка дає змогу виконувати код, написаний на .NET-сумісних мовах (таких як C#, VB.NET), безпосередньо в середовищі SQL Server. TRUSTWORTHY = OFF: забороняє базі даних отримувати доступ до зовнішніх ресурсів. Це налаштування за замовчуванням і рекомендується для підвищення безпеки.

PERSISTENT_LOG_BUFFER=ON (DIRECTORY_NAME=“ ”) – використовується для покращення продуктивності запису журналу транзакцій.

PERSISTENT_LOG_BUFFER=ON: цей параметр вмикає використання постійного буфера журналу. Постійний буфер журналу дозволяє SQL Server записувати записи журналу транзакцій безпосередньо в постійне сховище (наприклад, NVMe або Intel Optane), що значно зменшує затримку запису. DIRECTORY_NAME вказує шлях до каталогу, де буде розташований постійний буфер журналу. Якщо вказати пустий рядок, SQL сервер буде використовувати місце розташування файлів журналу транзакцій. Для оптимальної продуктивності, цей каталог повинен знаходитися на пристрої з низькою затримкою запису.

LEDGER = { ON | OFF } – вмикає або вимикає функціонал книги (ledger) для цієї бази даних. Книга (ledger) – це функція безпеки, яка надає криптографічно захищений журнал змін даних у базі даних. Вона дає змогу: відстежувати зміни даних, виявляти несанкціоновані зміни, забезпечувати аудит. LEDGER = ON: вмикає функціонал книги для бази даних. Це означає, що всі зміни даних будуть записуватися в криптографічно захищений журнал. LEDGER = OFF: вимикає функціонал книги для бази даних. Це означає, що зміни даних не будуть записуватися в криптографічно захищений журнал.

Колація бази даних — це набір правил, які визначають, як сортуються та порівнюються символи в базі даних. Вона впливає на такі аспекти: сортування, порівняння, чутливість до регістру, чутливість до акцентів.

Вибір колації залежить від вимог вашого застосунку. Якщо вам потрібно, щоб символи сортувалися в певному порядку або порівнювалися певним чином, вам потрібно вибрати колацію, яка відповідає цим вимогам.

Файлова система бази даних — це структурована організація зберігання та управління файлами, що використовуються системою управління базами даних для зберігання даних, індексів, журналів транзакцій та інших системних компонентів.

Основні компоненти файлової системи бази даних:

файли даних — зберігають фактичні дані таблиць та індексів (первинні файли даних (наприклад, .mdf в SQL Server), вторинні файли даних (наприклад, .ndf в SQL Server));

файли журналів транзакцій — зберігають інформацію про всі зміни в базі даних для забезпечення цілісності та можливості відновлення (часто мають розширення .ldf в SQL Server або redo logs в Oracle);

системні файли — зберігають метадані про структуру бази даних, користувачів, права доступу;

файлові групи — логічне групування файлів даних для полегшення адміністрування та балансування навантаження;

тимчасові файли — використовуються для тимчасового зберігання проміжних результатів операцій.

Ключові характеристики файлової системи бази даних:

сторінкова організація — дані зберігаються в блоках або сторінках фіксованого розміру (8 КБ у SQL Server, різні розміри в інших СУБД);

фізична та логічна структура — логічні об'єкти (таблиці, індекси) відображаються на фізичні сховища;

механізми контролю паралелізму — для управління одночасним доступом;

системи кешування — для оптимізації продуктивності доступу до даних

На відміну від звичайних файлових систем ОС, файлові системи баз даних оптимізовані для роботи з структурованими даними, забезпечення цілісності транзакцій та швидкого доступу до даних.

2.2.15 Загальний синтаксис створення таблиці бази даних. Для формування структури таблиці в базі даних використовується команда ‘CREATE TABLE’. Ця команда ініціює створення порожньої таблиці, яка не містить жодних записів. При визначенні таблиці необхідно вказати її назву, детально описати кожен стовпець, включаючи його ім’я, тип даних та розмір, а також встановити обмеження на дані, що зберігаються в таблиці. Назви таблиць повинні бути унікальними в межах однієї бази даних. Кожен стовпець у таблиці також повинен мати унікальне ім’я, а також тип даних, обмеження цілісності або вираз для обчислення значення стовпця.

Загальний синтаксис:

CREATE TABLE

```

{ database_name.schema_name.table_name | schema_name.table_name |
table_name }

```

[AS FileTable]

```
( { <column_definition>
```

| <computed_column_definition>

| <column_set_definition>

| [<table_constraint>] [,... n]

| [<table_index>] }

$$[\dots n]$$

```
[ PERIOD FOR SYSTEM_TIME ( system_start_time_column_name
, system_end_time_column_name ) ]
```

)

```
[ ON { partition_scheme_name ( partition_column_name )
```

| filegroup

| “default” }]

[TEXTIMAGE_ON { filegroup | “default” }]

```

[ FILESTREAM_ON { partition_scheme_name
    | filegroup
    | "default" } ]
[ WITH ( <table_option> [ ,... n ] ) ]
[ ; ]

```

```

<column_definition> ::=
column_name <data_type>
    [ FILESTREAM ]
    [ COLLATE collation_name ]
    [ SPARSE ]
    [ MASKED WITH ( FUNCTION = 'mask_function' ) ]
    [ [ CONSTRAINT constraint_name ] DEFAULT constant_expression ]
    [ IDENTITY [ ( seed , increment ) ] ]
    [ NOT FOR REPLICATION ]
    [ GENERATED ALWAYS AS { ROW | TRANSACTION_ID |
SEQUENCE_NUMBER } { START | END } [ HIDDEN ] ]
    [ [ CONSTRAINT constraint_name ] { NULL | NOT NULL } ]
    [ ROWGUIDCOL ]
    [ ENCRYPTED WITH
        ( COLUMN_ENCRYPTION_KEY = key_name ,
          ENCRYPTION_TYPE = { DETERMINISTIC | RANDOMIZED } ,
          ALGORITHM = 'AEAD_AES_256_CBC_HMAC_SHA_256'
        ) ]
    [ <column_constraint> [ ,... n ] ]
    [ <column_index> ]

```

```

<data_type> ::=
[ type_schema_name. ] type_name
    [ ( precision [ , scale ] | max |

```

[{ CONTENT | DOCUMENT }] xml_schema_collection)]

<column_constraint> ::=

[CONSTRAINT constraint_name]

{

{ PRIMARY KEY | UNIQUE }

[CLUSTERED | NONCLUSTERED]

[(<column_name> [,... n])]

[

WITH FILLFACTOR = fillfactor

| WITH (<index_option> [,... n])

]

[ON { partition_scheme_name (partition_column_name)

| filegroup | “default” }]

| [FOREIGN KEY]

REFERENCES [schema_name.] referenced_table_name [(ref_column)

]

[ON DELETE { NO ACTION | CASCADE | SET NULL | SET
DEFAULT }]

[ON UPDATE { NO ACTION | CASCADE | SET NULL | SET
DEFAULT }]

[NOT FOR REPLICATION]

| CHECK [NOT FOR REPLICATION] (logical_expression)

}

<column_index> ::=

INDEX index_name [CLUSTERED | NONCLUSTERED]

[WITH (<index_option> [,... n])]

```

[ ON { partition_scheme_name ( column_name )
    | filegroup_name
    | default
    }
]
[ FILESTREAM_ON { filestream_filegroup_name | partition_scheme_name
| "NULL" } ]

```

```

<computed_column_definition> ::=
column_name AS computed_column_expression
[ PERSISTED [ NOT NULL ] ]
[
    [ CONSTRAINT constraint_name ]
    { PRIMARY KEY | UNIQUE }
    [ CLUSTERED | NONCLUSTERED ]
    [
        WITH FILLFACTOR = fillfactor
        | WITH ( <index_option> [ ,... n ] )
    ]
    [ ON { partition_scheme_name ( partition_column_name )
        | filegroup | "default" } ]

| [ FOREIGN KEY ]
    REFERENCES referenced_table_name [ ( ref_column ) ]
    [ ON DELETE { NO ACTION | CASCADE } ]
    [ ON UPDATE { NO ACTION } ]
    [ NOT FOR REPLICATION ]

| CHECK [ NOT FOR REPLICATION ] ( logical_expression )
]

```

<column_set_definition> ::=

column_set_name XML COLUMN_SET FOR ALL_SPARSE_COLUMNS

<table_constraint> ::=

[CONSTRAINT constraint_name]

{

{ PRIMARY KEY | UNIQUE }

[CLUSTERED | NONCLUSTERED]

(column_name [ASC | DESC] [,... n])

[

WITH FILLFACTOR = fillfactor

| WITH (<index_option> [,... n])

]

[ON { partition_scheme_name (partition_column_name)

| filegroup | “default” }]

| FOREIGN KEY

(column_name [,... n])

REFERENCES referenced_table_name [(ref_column [,... n])]

[ON DELETE { NO ACTION | CASCADE | SET NULL | SET
DEFAULT }]

[ON UPDATE { NO ACTION | CASCADE | SET NULL | SET
DEFAULT }]

[NOT FOR REPLICATION]

| CHECK [NOT FOR REPLICATION] (logical_expression)

}

<table_index> ::=

{

{

```

INDEX index_name [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ]
    ( column_name [ ASC | DESC ] [ ,... n ] )
| INDEX index_name CLUSTERED COLUMNSTORE [ ORDER
(column_name [ , ...n ] ) ]
| INDEX index_name [ NONCLUSTERED ] COLUMNSTORE (
column_name [ ,... n ] )
}
[ INCLUDE ( column_name [ ,... n ] ) ]
[ WHERE <filter_predicate> ]
[ WITH ( <index_option> [ ,... n ] ) ]
[ ON { partition_scheme_name ( column_name )
    | filegroup_name
    | default
    }
]
[ FILESTREAM_ON { filestream_filegroup_name | partition_scheme_name
| "NULL" } ]
}

```

```

<table_option> ::=
{
    [ DATA_COMPRESSION = { NONE | ROW | PAGE }
    [ ON PARTITIONS ( { <partition_number_expression> | <range> }
    [ ,... n ] ) ] ]
    [ XML_COMPRESSION = { ON | OFF }
    [ ON PARTITIONS ( { <partition_number_expression> | <range> }
    [ ,... n ] ) ] ]
    [ FILETABLE_DIRECTORY = <directory_name> ]
    [ FILETABLE_COLLATE_FILENAME = { <collation_name> |
database_default } ]
}

```

```

[ FILETABLE_PRIMARY_KEY_CONSTRAINT_NAME =
<constraint_name> ]
[ FILETABLE_STREAMID_UNIQUE_CONSTRAINT_NAME =
<constraint_name> ]
[ FILETABLE_FULLPATH_UNIQUE_CONSTRAINT_NAME =
<constraint_name> ]
[ SYSTEM_VERSIONING = ON
  [ ( HISTORY_TABLE = schema_name.history_table_name
    [ , DATA_CONSISTENCY_CHECK = { ON | OFF } ]
  ) ]
]
[ REMOTE_DATA_ARCHIVE =
{
  ON [ ( <table_stretch_options> [ ,... n ] ) ]
  | OFF ( MIGRATION_STATE = PAUSED )
}
]
[ DATA_DELETION = ON
  { (
    FILTER_COLUMN = column_name,
    RETENTION_PERIOD = { INFINITE | number { DAY | DAYS |
WEEK | WEEKS
| MONTH | MONTHS | YEAR | YEARS } }
  ) }
]
[ LEDGER = ON [ ( <ledger_option> [ ,... n ] ) ]
| OFF
]
}

```



```

<ledger_option> ::=
{
    [ LEDGER_VIEW = schema_name.ledger_view_name [ (
<ledger_view_option> [ ,... n ] ) ] ]
    [ APPEND_ONLY = ON | OFF ]
}

```

```

<ledger_view_option> ::=
{
    [ TRANSACTION_ID_COLUMN_NAME = transaction_id_column_name
]
    [ SEQUENCE_NUMBER_COLUMN_NAME =
sequence_number_column_name ]
    [ OPERATION_TYPE_COLUMN_NAME = operation_type_id
column_name ]
    [ OPERATION_TYPE_DESC_COLUMN_NAME =
operation_type_desc_column_name ]
}

```

```

<table_stretch_options> ::=
{
    [ FILTER_PREDICATE = { NULL | table_predicate_function } , ]
    MIGRATION_STATE = { OUTBOUND | INBOUND | PAUSED }
}

```

```

<index_option> ::=
{
    PAD_INDEX = { ON | OFF }
    | FILLFACTOR = fillfactor
    | IGNORE_DUP_KEY = { ON | OFF }
}

```

```

| STATISTICS_NORECOMPUTE = { ON | OFF }
| STATISTICS_INCREMENTAL = { ON | OFF }
| ALLOW_ROW_LOCKS = { ON | OFF }
| ALLOW_PAGE_LOCKS = { ON | OFF }
| OPTIMIZE_FOR_SEQUENTIAL_KEY = { ON | OFF }
| COMPRESSION_DELAY = { 0 | delay [ Minutes ] }
| DATA_COMPRESSION = { NONE | ROW | PAGE | COLUMNSTORE |
COLUMNSTORE_ARCHIVE }
    [ ON PARTITIONS ( { partition_number_expression | <range> }
    [ ,... n ] ) ]
| XML_COMPRESSION = { ON | OFF }
    [ ON PARTITIONS ( { <partition_number_expression> | <range> }
    [ ,... n ] ) ]
}

```

<range> ::=

<partition_number_expression> TO <partition_number_expression>

Аргументи:

DATABASE_NAME – ім'я бази даних, де буде створено таблицю. Воно має відповідати назві вже існуючої бази даних. Якщо ви не вкажете ім'я бази даних, таблиця буде створена в поточній базі даних за замовчуванням. Для того, щоб створити таблицю, ваш логін повинен бути пов'язаний з користувачем, який має дозвіл на створення таблиць (CREATE TABLE) в базі даних, зазначеній як *database_name*;

SCHEMA_NAME – назва схеми, якій належить нова таблиця. Схема таблиці бази даних — це опис структури таблиці, який включає: імена стовпців, типи даних, обмеження, зв'язки. Задання імені схеми при створенні таблиці в базі даних має кілька важливих функцій. Схеми дозволяють групувати таблиці та інші об'єкти бази даних у логічні категорії. Це особливо корисно у великих базах даних з безліччю таблиць, де схеми допомагають підтримувати порядок і

полегшують пошук потрібних об'єктів. Схеми можуть використовуватися для керування дозволами. Можна надати або заборонити доступ до всіх об'єктів у певній схемі, що дає змогу контролювати, хто має доступ до яких даних. У великих базах даних можуть виникати конфлікти імен, коли різні користувачі або програми намагаються створити таблиці з однаковими назвами. Схеми дозволяють уникнути цих конфліктів, оскільки об'єкти з однаковими назвами можуть існувати в різних схемах. Схеми часто використовуються для розмежування різних середовищ, таких як розробка, тестування та виробництво. Наприклад, можна мати схеми «dev», «test» і «prod», які містять таблиці для відповідних середовищ. Так само схеми можуть розділяти функціонал, наприклад, схема «бухгалтерія», «склад» і т.д.;

TABLE_NAME – назва таблиці;

AS FILETABLE дає змогу створити таблицю, яка поєднує можливості бази даних SQL Server і файлової системи Windows. При створенні таблиці в SQL Server використовується для створення спеціального типу таблиці, яка називається FileTable. FileTable – це спеціальний тип таблиці в SQL Server, який дає змогу зберігати файли та каталоги в базі даних. Він надає доступ до файлів через файлову систему Windows, що надає програмам, які працюють з файлами, легкий доступ до даних, які зберігаються в базі даних. FileTable інтегрує простір імен файлової системи Windows та SQL Server, дозволяючи зберігати файли та каталоги в таблицях бази даних. Це дає змогу використовувати функції SQL Server для керування файлами, такі як резервне копіювання, відновлення та пошук. FileTable корисний для програм, які потребують зберігання та керування великою кількістю файлів, таких як зображення, відео або документи. Це також корисно для програм, які потребують доступу до файлів через файлову систему Windows.

COLUMN_NAME AS COMPUTED_COLUMN_EXPRESSION – вираз, який визначає значення обчислюваного стовпця. Обчислюваний стовпець – це віртуальний стовпець, який фізично не зберігається в таблиці, якщо тільки стовпець не позначений як PERSISTED. Стовпець обчислюється з виразу, який

використовує інші стовпці в тій же таблиці. Обчислювані стовпці можна використовувати в запитах з SELECT, умовах WHERE, умовах ORDER BY або в будь-яких інших місцях, де можна використовувати звичайні вирази. Обчислювані стовпці повинні бути позначені як PERSISTED, щоб брати участь у обмеженнях FOREIGN KEY або CHECK. Обчислюваний стовпець може використовуватися як ключовий стовпець в індексі або як частина будь-якого обмеження PRIMARY KEY або UNIQUE, якщо значення обчислюваного стовпця визначено детермінованим виразом і тип даних результату дозволений в стовпцях індексу. Обчислюваний стовпець не може використовуватися з операторами INSERT або UPDATE;

PERSISTED – визначає, що ядро бази даних SQL Server фізично зберігатиме обчислені значення в таблиці та оновлюватиме значення, коли оновлюються будь-які інші стовпці, від яких залежить обчислюваний стовпець;

ON { partition_scheme | filegroup | «default» } – визначає схему розділів або групу файлів, у якій зберігається таблиця. Якщо вказано partition_scheme, таблиця буде розділеною таблицею, розділи якої зберігаються в наборі однієї або кількох груп файлів, визначених у partition_scheme. Якщо вказано filegroup, таблиця зберігається в названій групі файлів. Група файлів повинна існувати в базі даних. Якщо вказано «default» або якщо ON взагалі не вказано, таблиця зберігається в групі файлів за замовчуванням. Механізм зберігання таблиці, визначений у CREATE TABLE, не може бути згодом змінений. Розділені таблиці – це метод організації великих таблиць у базі даних, який дає змогу розділити дані на менші, керовані частини, що називаються розділами. Розділення передбачає розбиття великої таблиці на менші, логічно незалежні частини на основі певного критерію, наприклад, діапазону дат, значень ідентифікаторів або інших відповідних характеристик. Розділення може значно підвищити продуктивність запитів, оскільки запити можуть сканувати лише відповідні розділи, а не всю таблицю. Розділення полегшує керування великими обсягами даних, дозволяючи виконувати операції, такі як резервне копіювання, відновлення та обслуговування, на окремих розділах. Розділи

можна зберігати на різних групах файлів, що дає змогу оптимізувати зберігання та використовувати різні типи сховищ для різних розділів. Запити, які звертаються лише до невеликої частини даних, можуть виконуватися значно швидше. Операції обслуговування, такі як резервне копіювання та відновлення, можуть виконуватися на окремих розділах, а не на всій таблиці. Якщо один розділ виходить з ладу, інші розділи залишаються доступними. Розділення дає змогу легко архівувати або видаляти старі дані, переміщуючи або видаляючи цілі розділи. Розділені таблиці особливо корисні для великих таблиць, які містять мільйони або мільярди рядків. Розділення часто використовується для зберігання історичних даних, де старі дані можуть бути розділені та зберігатися окремо. Розділення добре підходить для даних часових рядів, де дані можуть бути розділені за датою або часом;

`TEXTIMAGE_ON { filegroup | «default» }` визначає, де будуть зберігатися дані для стовпців великих значень. Цей аргумент використовується для вказівки групи файлів, у які будуть зберігатися дані типів `text`, `ntext`, `image`, `xml`, `varchar(max)`, `nvarchar(max)`, `varbinary(max)` та CLR визначених користувачем типів. Ці типи даних можуть зберігати великі обсяги тексту, зображень або інших двійкових даних; `filegroup`: вказує ім'я конкретної групи файлів, у якій будуть зберігатися дані; `«default»`: вказує, що дані будуть зберігатися в групі файлів за замовчуванням. Якщо аргумент `TEXTIMAGE_ON` не вказано, дані також будуть зберігатися в групі файлів за замовчуванням. Аргумент `TEXTIMAGE_ON` не можна використовувати, якщо в таблиці немає стовпців великих значень. Його не можна використовувати, якщо вказано аргумент `partition_scheme`. Місце зберігання даних великих значень, визначене при створенні таблиці, не можна змінити згодом. Використання `TEXTIMAGE_ON` дає змогу розділити зберігання великих обсягів даних від інших даних таблиці, що може підвищити продуктивність та полегшити керування базою даних. Наприклад, можна розмістити великі об'єкти даних на більш швидких дисках;

`FILESTREAM_ON { partition_scheme_name | filegroup | «default» }` визначає групу файлів для даних `FILESTREAM`. `FILESTREAM` — це функція

SQL Server, яка дає змогу зберігати неструктуровані дані, такі як документи, зображення та відео, у файловій системі NTFS, зберігаючи при цьому метадані в базі даних SQL Server. FILESTREAM інтегрує ядро СУБД SQL Server з файловою системою NTFS, дозволяючи зберігати дані великих двійкових об'єктів (BLOB) у вигляді файлів у файловій системі. Це забезпечує баланс між перевагами зберігання даних у базі даних (наприклад, транзакційна цілісність, резервне копіювання та відновлення) та перевагами зберігання даних у файловій системі (наприклад, потокове передавання та доступ до файлів з програм, які не є SQL Server). FILESTREAM особливо корисна для програм, які працюють з великими обсягами неструктурованих даних;

[type_schema_name.] type_name визначає тип даних стовпця; type_name – це ім'я типу даних, який ви хочете призначити стовпцю. Це може бути: системний тип даних (наприклад, int, varchar, datetime), тип псевдоніма, створений на основі системного типу даних, CLR визначений користувачем тип; type_schema_name – це ім'я схеми, до якої належить тип даних;

CONTENT використовується при створенні стовпця типу xml у таблиці SQL Server. Він визначає, як дані XML будуть інтерпретуватися та зберігатися;

DOCUMENT визначає, що кожен екземпляр типу даних xml у column_name може містити лише один елемент верхнього рівня. DOCUMENT застосовується лише до типу даних xml і може бути вказаний лише в тому випадку, якщо також вказано xml_schema_collection. У контексті XML-документів «елемент верхнього рівня» відноситься до елемента, який знаходиться на найвищому рівні ієрархії документа;

xml_schema_collection застосовується лише до типу даних xml для зв'язування колекції XML-схем з типом. Перед тим, як прив'язати стовпець xml до схеми, схему необхідно спочатку створити в базі даних за допомогою CREATE XML SCHEMA COLLECTION;

DEFAULT визначає значення, яке надається стовпцю, коли значення явно не надається під час вставки. Визначення DEFAULT можуть застосовуватися до будь-яких стовпців, крім тих, що визначені як timestamp, або тих, що мають

властивість `IDENTITY`. Якщо для стовпця типу, визначеного користувачем, вказано значення за замовчуванням, тип повинен підтримувати неявне перетворення з `constant_expression` на тип, визначений користувачем;

`IDENTITY` вказує, що новий стовпець є стовпцем ідентифікації. Коли до таблиці додається новий рядок, ядро бази даних надає унікальне, наростаюче значення для стовпця. Стовпці ідентифікації зазвичай використовуються з обмеженнями `PRIMARY KEY` для служіння унікальним ідентифікатором рядка для таблиці. Властивість `IDENTITY` може бути присвоєна стовпцям типів `tinyint`, `smallint`, `int`, `bigint`, `decimal(p, 0)` або `numeric(p, 0)`;

`NOT FOR REPLICATION` може бути вказаний для властивості `IDENTITY`, обмежень `FOREIGN KEY` та обмежень `CHECK`. Якщо цей пункт вказаний для властивості `IDENTITY`, значення не збільшуються в стовпцях ідентифікації, коли агенти реплікації виконують вставки. Якщо цей пункт вказаний для обмеження, обмеження не застосовується, коли агенти реплікації виконують операції вставки, оновлення або видалення. Агенти реплікації в контексті баз даних, зокрема `SQL Server`, — це програми або процеси, які відповідають за копіювання та синхронізацію даних між різними базами даних. Вони є ключовими компонентами механізму реплікації, який дає змогу підтримувати копії даних на кількох серверах.

`GENERATED ALWAYS AS { ROW | TRANSACTION_ID | SEQUENCE_NUMBER } { START | END } [ HIDDEN ] [ NOT NULL ]` використовується для створення стовпців, значення яких генеруються автоматично системою. Ця функція особливо важлива для темпоральних таблиць та таблиць реєстру, де необхідно відстежувати зміни даних з часом. `GENERATED ALWAYS AS`: вказує, що значення стовпця завжди генерується системою і не може бути змінено користувачем. `{ ROW | TRANSACTION_ID | SEQUENCE_NUMBER }`: визначає, який тип значення буде генеруватися — `ROW` вказує, що значення представляє час початку (`START`) або закінчення (`END`) періоду дії рядка. `TRANSACTION_ID` вказує ідентифікатор транзакції, яка створила (`START`) або зробила недійсною (`END`) версію рядка.

SEQUENCE_NUMBER вказує порядковий номер операції. { START | END } визначає, чи представляє згенероване значення час початку або закінчення періоду дії рядка. [HIDDEN] вказує, що стовпець не повинен відображатися в звичайних запитах, якщо явно не вказано. [NOT NULL] вказує, що стовпець не може містити значення NULL. Цей аргумент використовується для створення стовпців, які відстежують період дії рядка в темпоральних таблицях. Темпоральні таблиці дозволяють зберігати історію змін даних. Цей аргумент використовується для створення стовпців, які відстежують ідентифікатори транзакцій та порядкові номери операцій у таблицях реєстру. Таблиці реєстру дозволяють вести аудит змін даних;

INDEX index_name [CLUSTERED | NONCLUSTERED] (column_name [ASC | DESC] [,... n]) визначає створення індексу на таблиці. Це може бути кластеризований або некластеризований індекс. Індекс міститиме перелічені стовпці та сортуватиме дані в порядку зростання або спадання;

INDEX index_name CLUSTERED COLUMNSTORE визначає зберігання всієї таблиці в стовпцевому форматі з кластеризованим стовпцевим індексом. Це завжди включає всі стовпці таблиці. Дані не сортуються в алфавітному або числовому порядку, оскільки рядки організовані для отримання переваг стиснення стовпцевого сховища;

INDEX index_name [NONCLUSTERED] COLUMNSTORE (column_name [,... n]) визначає створення некластеризованого стовпцевого індексу в таблиці. Базова таблиця може бути купою рядкового сховища або кластеризованим індексом, або це може бути кластеризований стовпцевий індекс. У всіх випадках створення некластеризованого стовпцевого індексу в таблиці зберігає другу копію даних для стовпців в індексі. Некластеризований стовпцевий індекс зберігається та керується як кластеризований стовпцевий індекс. Він називається некластеризованим стовпцевим індексом, оскільки стовпці можуть бути обмежені та він існує як вторинний індекс на таблиці;

ON partition_scheme_name (column_name) використовується для визначення схеми розділів, за якою дані таблиці або індексу будуть розподілені

між групами файлів; `partition_scheme_name` вказує на існуючу схему розділів, яка визначає, як дані будуть розподілені між групами файлів. Схема розділів пов'язує функцію розділення з групами файлів; `column_name` вказує стовпець, значення якого використовуються для визначення, в який розділ (і, отже, в яку групу файлів) потрапить кожен рядок даних. Тип даних, довжина та точність цього стовпця повинні відповідати аргументу функції розділення, яка використовується схемою розділів. На основі значень у стовпці розділення та визначеної функції розділення, SQL Server розподіляє рядки даних таблиці між групами файлів, визначеними схемою розділів. Це дає змогу розділити великі таблиці на менші, більш керовані частини, що може покращити продуктивність запитів та обслуговування;

`ON FILEGROUP_NAME` створює вказаний індекс у вказаній групі файлів. Якщо місце розташування не вказано, і таблиця або представлення не розділені, індекс використовує ту ж групу файлів, що й базова таблиця або представлення. Група файлів повинна вже існувати;

`ON «DEFAULT»` використовується для вказівки групи файлів, у якій будуть зберігатися дані. «DEFAULT» позначає, що таблиця або індекс повинні бути створені в стандартній групі файлів бази даних. Стандартна група файлів – це група файлів, яка використовується за замовчуванням для зберігання об'єктів бази даних, якщо не вказано іншу групу файлів;

`[FILESTREAM_ON { filestream_filegroup_name | partition_scheme_name | «NULL» }]` використовується під час створення таблиці для вказівки, де будуть зберігатися дані FILESTREAM; `filestream_filegroup_name` – вказує назву групи файлів FILESTREAM, в якій будуть зберігатися дані FILESTREAM. Група файлів FILESTREAM повинна бути створена до використання цього параметра; `partition_scheme_name` – якщо таблиця розділена, можна використовувати схему розділів, щоб розподілити дані FILESTREAM між кількома групами файлів FILESTREAM. Схема розділів повинна бути створена до використання цього параметра. «NULL» вказує, що дані FILESTREAM будуть зберігатися в

стандартній групі файлів FILESTREAM. Це значення використовується, якщо не потрібно вказувати конкретну групу файлів FILESTREAM.

ROWGUIDCOL використовується при створенні таблиці для позначення стовпця типу uniqueidentifier як глобально унікального ідентифікатора (GUID) для рядків таблиці. UNIQUEIDENTIFIER — це тип даних у SQL Server, який використовується для зберігання глобально унікальних ідентифікаторів. GUID — це 128-бітне число, яке з великою ймовірністю є унікальним у всьому світі. GUID зазвичай відображається у вигляді рядка з 36 символів, розділених дефісами, наприклад: xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx. ROWGUIDCOL вказує, що стовпець, до якого він застосовується, містить GUID, який однозначно ідентифікує кожен рядок у таблиці. Цей стовпець зазвичай використовується для реплікації, злиття та інших сценаріїв, де потрібні глобально унікальні ідентифікатори. Стовпець, позначений як ROWGUIDCOL, повинен мати тип даних uniqueidentifier. У кожній таблиці може бути лише один стовпець, позначений як ROWGUIDCOL. Хоча ROWGUIDCOL позначає стовпець як GUID, він не забезпечує автоматичну генерацію унікальних значень. Для цього потрібно використовувати функцію NEWID() або інші методи. ROWGUIDCOL часто використовується в реплікації SQL Server, щоб забезпечити однозначну ідентифікацію рядків у реплікованих таблицях. Це дає змогу уникнути конфліктів при злитті змін з різних реплік.

ENCRYPTED WITH використовується під час створення таблиці для вказівки, що певні стовпці таблиці повинні бути зашифровані за допомогою функції Always Encrypted.

MASKED WITH (FUNCTION = 'mask_function') використовується під час створення таблиці для маскуванню даних у стовпці. Це дає змогу приховувати конфіденційну інформацію від користувачів, які не мають відповідних прав доступу. MASKED WITH вказує, що дані в зазначеному стовпці будуть замасковані. Це дає змогу захистити конфіденційні дані, такі як номери кредитних карток, адреси електронної пошти або номери телефонів. FUNCTION = 'mask_function' вказує, яку функцію маскуванню використовувати

для маскуванн^я даних. SQL Server надає кілька вбудованих функцій маскуванн^я, таких як: default() – замінює дані значеннями за замовчуванн^{ям} для типу даних стовпця; email() – маскує адреси електронної пошти; partial(prefix, [length], replacement) – частково маскує дані; random([min], max) – замінює дані випадковими числами. Масковані дані відображаються у замаскованому вигляді для користувачів, які не мають дозволу UNMASK. Користувачі з дозволом UNMASK можуть бачити дані у відкритому вигляді. Маскування даних не шифрує дані. Маскування даних може вплинути на продуктивність запитів;

<table_option> дає змогу налаштувати різні опції, які впливають на поведінку та характеристики таблиці.

DATA_COMPRESSION визначає рівень стиснення даних для таблиці або її розділів. Це може допомогти зменшити обсяг зберігання та покращити продуктивність запитів.

MAX_GRANT_PERCENT визначає максимальний відсоток пам'яті, який може бути наданий запиту, що працює з таблицею. Це може допомогти запобігти запитам від споживання надмірної кількості пам'яті.

LOCK_ESCALATION визначає, як SQL Server обробляє ескалацію блокувань для таблиці. Це може вплинути на конкурентність та продуктивність запитів.

MEMORY_OPTIMIZED визначає, чи є таблиця оптимізованою для пам'яті. Оптимізовані для пам'яті таблиці зберігаються в пам'яті, що може значно покращити продуктивність запитів.

DURABILITY визначає, як зміни в оптимізованих для пам'яті таблицях зберігаються на диску.

DELAYED_DURABILITY визначає, як транзакції, що змінюють данні в таблиці, обробляються з точки зору довговічності.

SPARSE використовується для оптимізації зберігання стовпців з великою кількістю значень NULL.

Питання для самоперевірки

Що таке схема таблиці? Що дозволяють робити схеми? Чи є можливість за допомогою схеми налаштувати дозволи?

Для чого служить параметр AS FILETABLE?

Що за тип таблиці FileTable, де використовується?

Що таке обчислювальний стовпець, як він визначається?

Як такий стовпець має бути позначений, щоб мати участь у обмеженнях?

Що визначає параметр PERSISTED?

Що означає partition_scheme у виразі: ON { partition_scheme | filegroup | «default» }?

Що означає filegroup у виразі: ON { partition_scheme | filegroup | «default» }?

Що означає «default» у виразі: ON { partition_scheme | filegroup | «default» }?

Що передбачає розбиття великої таблиці на менші? Яким чином це працює?

Яка зона використання розбиття великої таблиці?

Що визначає TEXTIMAGE_ON { filegroup | «default» }?

Що дозволяє зробити використання TEXTIMAGE_ON?

Що робить функція FILESTREAM?

Де використовується параметр CONTENT?

Що визначає параметр DOCUMENT?

Коли використовується вираз GENERATED ALWAYS AS { ROW | TRANSACTION_ID | SEQUENCE_NUMBER } { START | END } [HIDDEN] [NOT NULL]?

UNIQUEIDENTIFIER – що це за тип даних?

Яка різниця між маскуванням даних і шифруванням даних?

Що таке логічна структура бази даних?

Що являє собою таблиця?

Чи можуть стовпці у таблиці бути обчислюваними?

Як фізично реалізується подання?

Яким чином використовується подання?

Який префікс мають назви збережених процедур?

Що таке тригер?

Чи можуть тригери мати вкладені тригери?

Де застосовуються індекси?

Що таке обмеження цілісності?

Коли використовується значення за замовчуванням?

Які основні концепції метамови, заснованої на нормальних формах Бекуса Наура?

На що вказує значення параметру CONTAINMENT?

Що таке залежність бази даних від серверу і в чому вона проявляється?

Яким чином можливо встановити шлях до файлу журналу транзакцій?

Яке розширення зазвичай він має в SQL Server?

Що таке колація бази даних? Яке значення вона має?

Що таке файлова система бази даних? Назвіть її характеристики.

Яка різниця між параметрами DEFAULT_FULLTEXT_LANGUAGE і DEFAULT_LANGUAGE?

Як дозволити тригерам викликати інші тригери?

Що таке «шумові слова»? Яким чином можливо їх ігнорування під час пошуку текстової інформації?

Як визначається поріг для інтерпретації двозначних років?

Якими типами даних в SQL можна задати символьну інформацію?

Які типи даних визначають числа?

Яка різниця між типом CHAR(N) і VARCHAR(N)?

Яка різниця між типом NCHAR(N) і NVARCHAR(N)?

Яка різниця між типом CHAR(N) і NCHAR(N)?

Якими типами даних в SQL можна задати дату?

Які складові SQL?

Що таке сценарій?

Чим забезпечується безпека та цілісність даних?

Що таке константи?

Які типи констант підтримуються у SQL?

Що називається ідентифікатором?

Які правила найменування підтримуються у SQL?

Які є способи посилання на об'єкти у SQL?

Що таке стандартні ідентифікатори?

Що таке обмежені ідентифікатори?

Які арифметичні оператори підтримує SQL?

Які логічні оператори підтримує SQL?

Що роблять оператори «ALL», «SOME», «ANY»?

Яке назначення операторів «BETWEEN» і «BETWEEN»?

Коли повертає істину оператор «EXISTS»?

Яке назначення операторів «IN» і «NOT IN»?

Для чого використовується оператор «LIKE»?

Скільки є в реляційних базах даних обмежень цілісності? Поясніть кожне з них.

Яка різниця між обмеженням **UNIQUE** і **PRIMARY KEY**?

Яка таблиця називається залежною?

Завдання

Варіант	Опис завдання
1	Створити базу даних, яка відповідає наступним вимогам: часткова залежність від метаданих сервера, порівняння символічних даних відбувається за українським алфавітом, мова повнотекстового пошуку повинна бути українська і продуктивність текстового пошуку повинна бути покращена, мова за замовчуванням українська, поріг для інтерпретації двозначних років починається 2000 року, для бази даних повинен бути увімкнений функціонал книги. Базу даних

	створити за допомогою командного рядка. 2.Створити дві таблиці за допомогою графічного дизайнера SQL Server Management Studio. Заповнити по 3 рядки в першій таблиці і 5 рядків у другій. БД – <i>успішність студентів ВНЗ</i> – складається з таких таблиць: факультети, кафедри. Таблиця факультети має атрибути: ідентифікатор факультету (його аббревіатура, короткий рядок), назва факультету (рядок), ПІБ декана (рядок), номер корпусу (коротке ціле у межах від 1 до 5), номер телефону (рядок, довжина 10). 3.Таблиця кафедра має такі атрибути: ідентифікатор кафедри (рядок, довжиною 5 символів), назва кафедри (рядок), ідентифікатор факультету (короткий рядок), ПІБ завідувача (рядок).
2	1.Створити базу даних, яка відповідає наступним вимогам: залежність від метаданих сервера, порівняння символічних даних відбувається за українським алфавітом, мова повнотекстового пошуку повинна бути українська, мова за замовчуванням англійська, поріг для інтерпретації двозначних років починається 1950 року. Базу даних створити за допомогою командного рядка. 2.Створити дві таблиці за допомогою командного рядка SQL. Заповнити по 3 рядки в першій таблиці і 5 рядків у другій в середовищі SQL Server Management Studio. БД – <i>інформаційна система супермаркету</i> – складається з таких таблиць: відділи, співробітники. Таблиця відділи має такі атрибути: ідентифікатор відділу (коротке ціле, у межах від 1 до 20), назва відділу (рядок, довжиною 20 символів), кількість прилавків (коротке ціле), кількість продавців (коротке ціле), номер залу (коротке ціле, у межах від 1 до 5). 3.Таблиця співробітники має такі атрибути: табельний номер

	співробітника (короткий рядок, довжиною 6, унікальне значення), прізвище, ім'я, по батькові (рядок), відділ (коротке ціле, у межах від 1 до 20), дата народження, дата влаштування на роботу, стаж (коротке ціле, у межах від 0 до 35), посада (рядок), стать (бінарне значення), адреса (рядок, довжиною 250), місто (рядок, довжиною 20 символів), номер телефону (рядок, довжиною 10 символів).
3	1.Створити базу даних, яка відповідає наступним вимогам: часткова залежність від метаданих сервера, порівняння символічних даних відбувається за українським алфавітом, мова повнотекстового пошуку повинна бути українська, мова за замовчуванням українська, поріг для інтерпретації двозначних років починається з 2000 року, для бази даних повинен бути увімкнений функціонал книги. Базу даних створити за допомогою командного рядка. 2.Створити першу таблицю в SQL Server Management Studio. Другу таблицю у командному рядку. Заповнити по 3 рядки у першій таблиці і 5 рядків у другій. БД – <i>інформаційна система військового округу</i> – складається з таких таблиць: місця дислокації, вид військ. Таблиця вид військ має такі атрибути: ідентифікатор (коротке ціле, кожний наступний запис більше на 1), назва (рядок). 3.Таблиця місця дислокації має такі атрибути: (коротке ціле, кожний наступний запис більше на 1), країна (рядок, довжиною 20 символів), місто(рядок, довжиною 20 символів), адреса (рядок, довжиною 250 символів), яку займає площу (раціональне число, з 2 символи після коми).
4	1.Створити базу даних, яка відповідає наступним вимогам: часткова залежність від метаданих сервера, порівняння символічних даних відбувається за українським алфавітом, мова повнотекстового пошуку повинна бути українська і продуктивність текстового

	пошуку повинна бути покращена, мова за замовчуванням українська, поріг для інтерпретації двозначних років починається 2000 року, для бази даних повинен бути увімкнений функціонал книги. Базу даних створити за допомогою командного рядка. 2.Таблиці створити в SQL Server Management Studio. Заповнити по 3 рядки у першій таблиці і 3 рядки у другій. БД – <i>інформаційна система бібліотеки</i> – складається з таких таблиць: бібліотеки, фонд бібліотеки. Таблиця бібліотеки має такі атрибути: номер бібліотеки (коротке ціле, унікальне значення), назва бібліотеки (рядок, довжиною 50 символів), адреса (рядок, довжиною 20 символів), місто (рядок, довжиною 20 символів). 3.Таблиця фонд бібліотеки має такі атрибути: ідентифікатор фонду (коротке ціле, кожне наступне більше на 1), назва фонду (рядок, довжиною 50 символів), номер бібліотеки (коротке ціле), кількість книг (ціле), кількість журналів (ціле), кількість газет (ціле), кількість збірок (ціле), кількість дисертацій (ціле), кількість рефератів (ціле).
5	1.Створити базу даних, яка відповідає наступним вимогам: часткова залежність від метаданих сервера, порівняння символічних даних відбувається за українським алфавітом, мова повнотекстового пошуку повинна бути українська і продуктивність текстового пошуку повинна бути покращена, мова за замовчуванням українська, поріг для інтерпретації двозначних років починається 1989 року. Базу даних створити за допомогою командного рядка. Таблиці створити за допомогою командного рядка. Заповнити по 5 рядків у першій таблиці і 3 рядки у другій. БД – <i>інформаційна система туристичного агентства</i> – складається з таких таблиць: пансіонати, вид житла.

	Таблиця пансіонати має такі атрибути: : ідентифікатор пансіонату (ціле, кожне наступне більше на 1), назва пансіонату (рядок), адреса (рядок, довжиною 250 символів), місто (рядок, довжиною 20 символів), країна (рядок, довжиною 20 символів), номер телефону (рядок, довжиною 10 символів), опис території (текст), кількість кімнат (коротке ціле), наявність басейну (бінарне значення), наявність медичних послуг (бінарне значення), наявність спа-салону (бінарне значення), рівень пансіонату (коротке ціле у межах від 1 до 4), відстань до моря (раціональне число з 3 символами після коми). Таблиця вид житла має такі атрибути: : ідентифікатор виду (коротке ціле, кожне наступне більше на 1), назва (рядок: будинок, бунгало, квартира, 1-я кімната, 2-я кімната тощо), категорія житла (рядок, люкс, напівлюкс, тощо), ідентифікатор пансіонату, опис умов проживання (текст), ціна за номер на добу (грошовий тип).
--	--

2.3 Вибірка, фільтрація та зміна даних

2.3.1 Робота з Azure Data Studio. Azure Data Studio — це середовище для роботи з даними, що використовує сукупність локальних і хмарних платформ даних. Хоча Azure Data Studio не замінює повністю SQL Server Management Studio, він доповнює його, пропонуючи більш легковагове та кросплатформне рішення. SSMS залишається основним інструментом для глибокого адміністрування SQL Server (наприклад, управління реплікацією). Azure Data Studio, навпаки, краще підходить для щоденної розробки запитів, створення сценаріїв, візуалізації та роботи з різними джерелами даних. Одна з найважливіших переваг Azure Data Studio полягає в його кросплатформності. Він доступний для операційних систем Windows, macOS та Linux, що дозволяє фахівцям працювати з базами даних незалежно від обраної ними робочої

станції. Це значно розширює гнучкість та доступність інструменту. Після запуску Azure Data Studio відкриється стартове вікно для підключення до бази даних (рис. 2.21).

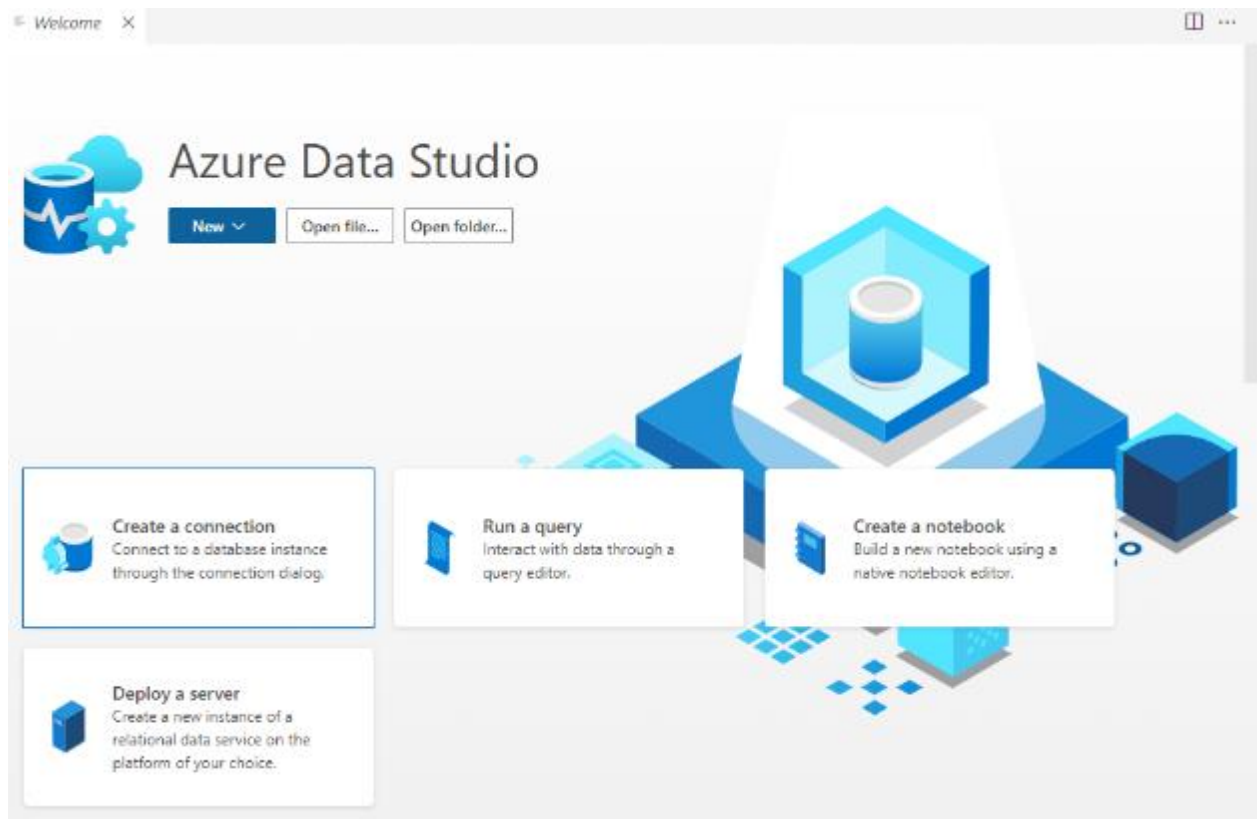


Рисунок 2.21 –Стартове вікно Azure Data Studio

Azure Data Studio, хоча й розпочинався як легкий інструмент спеціально для SQL Server, значно розширив свої можливості, перетворившись на по-справжньому універсальне середовище для роботи з даними. Завдяки системі розширень, його функціонал поширюється на PostgreSQL, MySQL, MongoDB та численні інші джерела даних. Інтерфейс Azure Data Studio, розроблений на базі Visual Studio Code, вирізняється чистотою, сучасністю та інтуїтивністю, що робить його зручним та легким для освоєння розробниками. Він пропонує ряд інтегрованих інструментів, що покращують робочий процес. Серед них — вбудований термінал, який дозволяє запускати команди Bash або PowerShell безпосередньо в середовищі, а також інтегрований контроль версій за допомогою Git, що спрощує управління скриптами та проектами. Функції, такі як фрагменти коду (Snippets) для швидкої вставки типових блоків, та

інтелектуальне завершення коду (IntelliSense) для T-SQL запитів, значно підвищують швидкість і точність написання коду.

Перший крок, який потрібно зробити для роботи з даними в Azure Data Studio – це створити з'єднання із сервером. Для досягнення мети потрібно натиснути на кнопку **Create a connection**, щоб відкрити вікно для налаштування параметрів підключення (рис. 2.22).

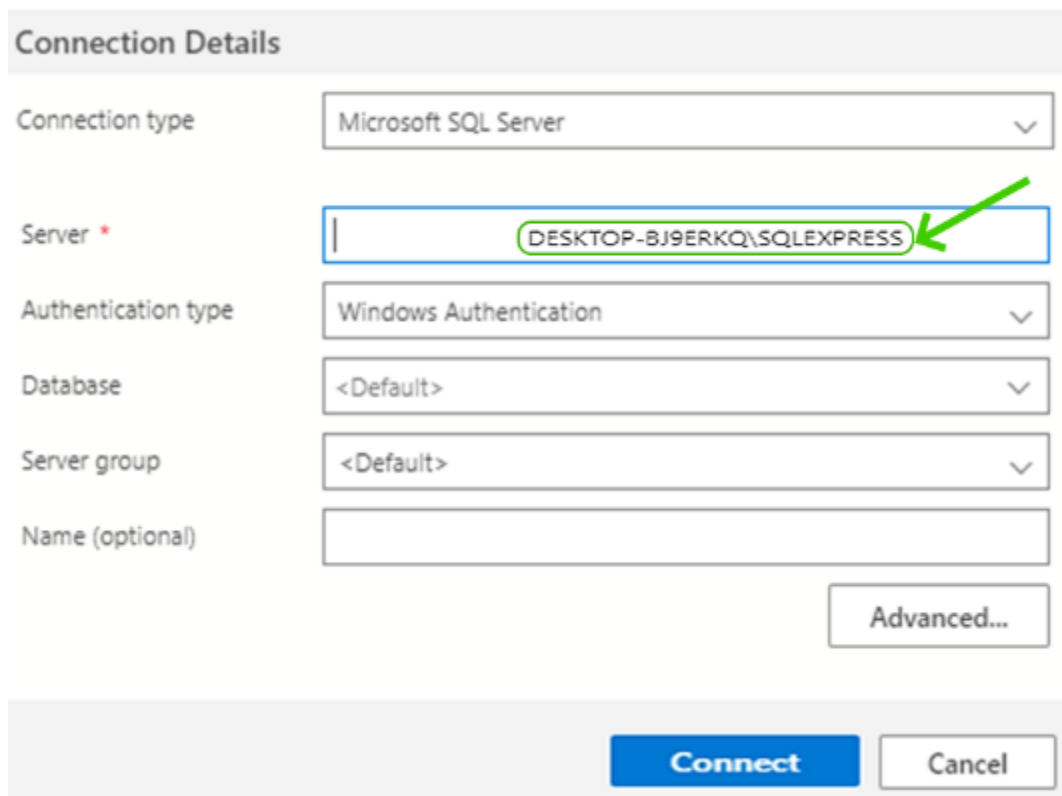


Рисунок 2.22 – Вікно налаштування параметрів підключення до сервера

Параметри підключення:

тип підключення – SQL Server;

ім'я сервера – введіть ім'я комп'ютера, на якому запущено SQL Server. У цьому прикладі – DESKTOP-BJ9ERKQ\SQLEXPRESS – це локальний хост, оскільки підключення йде до локальної БД;

тип автентифікації – рекомендовано «Автентифікація Windows», яка дає змогу використовувати свої облікові дані Windows для автентифікації на SQL Server;

ім'я бази даних – <за замовчуванням>;

група серверів – <за замовчуванням >;

ім'я (за потреби) – цей параметр дає змогу вказати зрозуміле ім'я для вашого сервера.

Якщо натиснути кнопку Advanced, то вводиться така інформація:

тип перевірки довжини: Ім'я входу SQL;

ім'я користувача: ім'я користувача SQL Server;

пароль: пароль для SQL Server;

Потрібно ввести тільки ім'я свого комп'ютера, на якому запущено SQL Server, інші параметри можуть лишитися за замовчуванням. Після натиснення «Connect» відбудеться підключення до SQL Server і у Azure Data Studio відкриється нове вікно, вкладення «Home» (рис. 2.23).

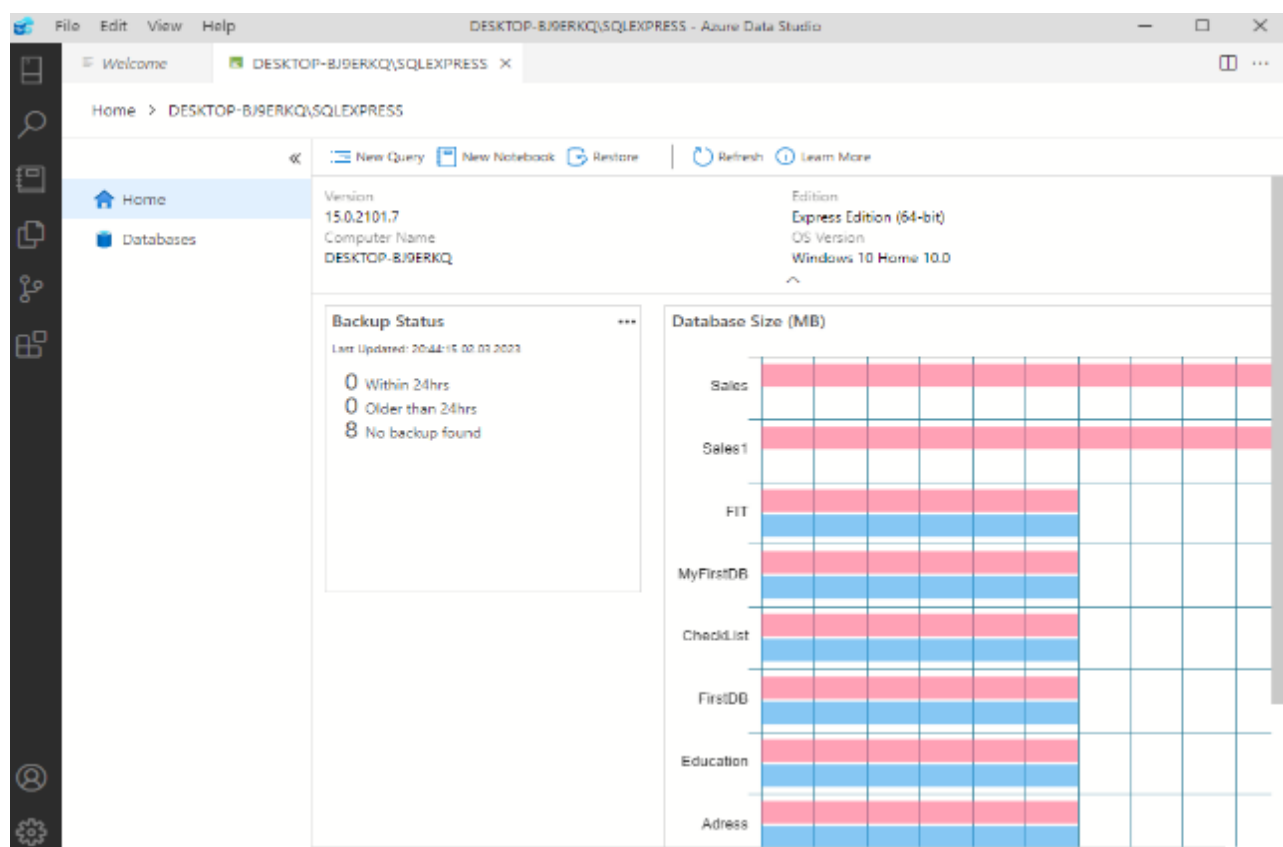


Рисунок 2.23 –Вкладення «Home» в Azure Data Studio після підключення до сервера

На вкладенні «Database» відображаються всі бази даних, що розташовані на сервері (рис. 2.24).

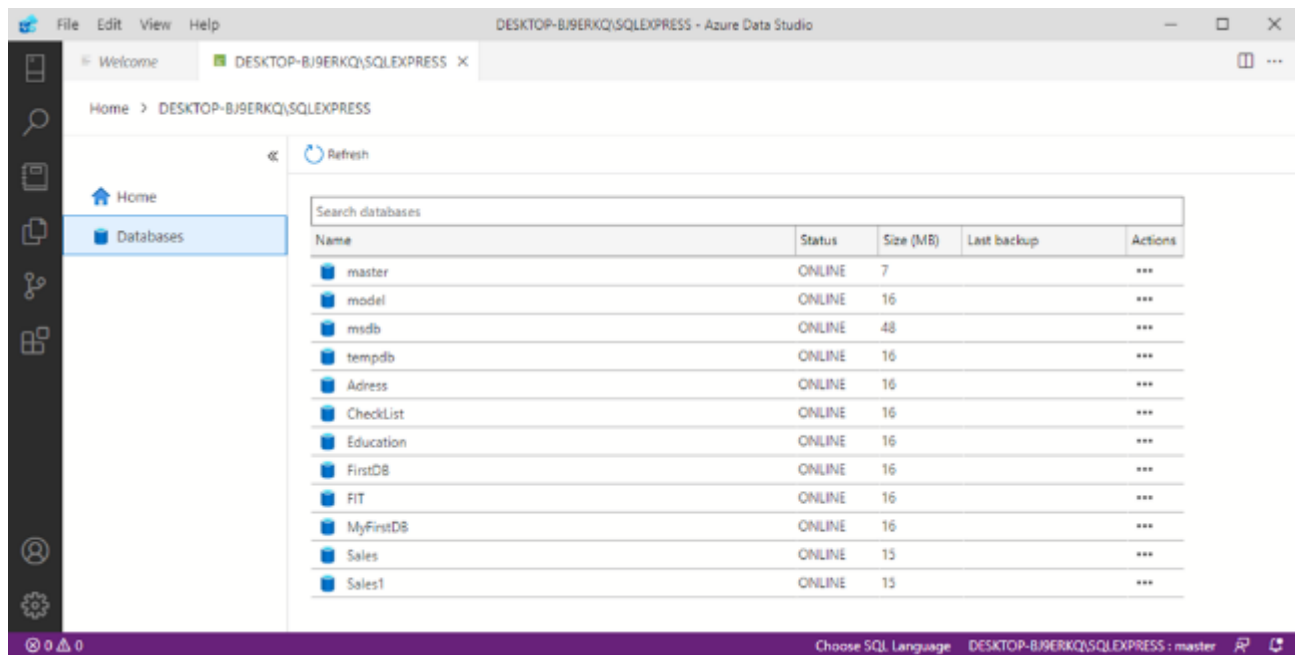


Рисунок 2.24 –Вкладення Database в Azure Data Studio

Щоб переглянути таблиці, які є у базі даних, достатньо натиснути лівою кнопкою миші на назві потрібної бази даних. На рис. 2.25 показано таблиці бази даних FIT.

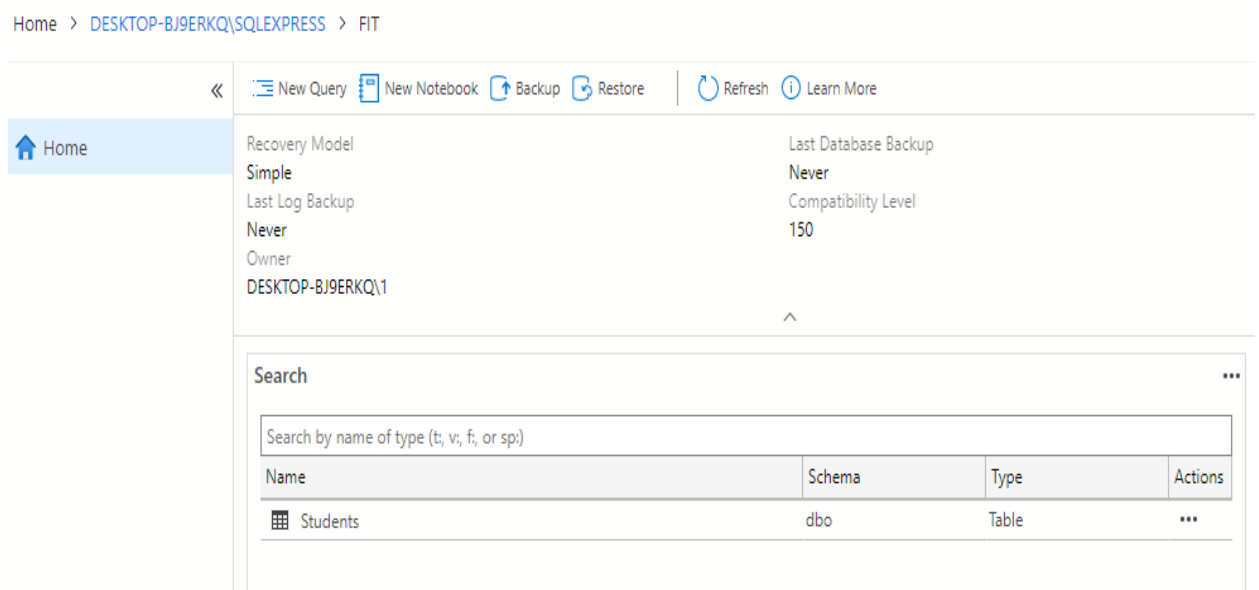


Рисунок 2.25 –Таблиці бази даних FIT

Якщо потрібно повернутися назад, то використовують кнопку «», яка розташована зліва вгорі (над «Home»).

На цьому етапі у базі даних FIT лише одна таблиця – Students. Для того, щоб додати дані в цю таблицю, потрібно створити новий запит (кнопка **New Query** (рис. 2.26)).

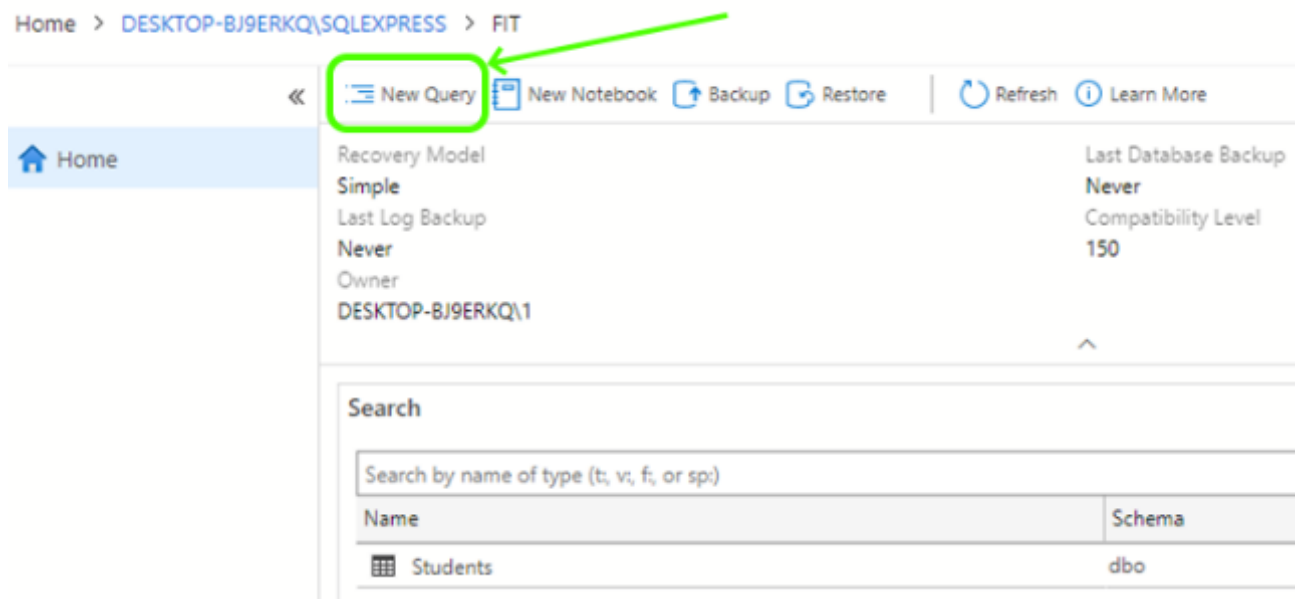


Рисунок 2.26 –Кнопка для створення нового запиту

Після цього відкриється нове вікно з редактором запитів. Для виконання SQL-запиту використовується кнопка Run (рис. 2.27).

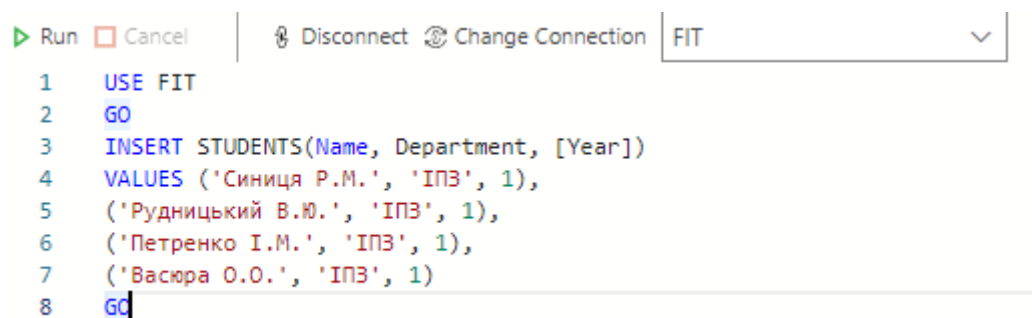


Рисунок 2.27 –Редактор запитів з прикладом SQL-запиту на додавання нових записів до таблиці Students

Для перегляду вмісту таблиці після виконання операцій необхідно перейти до розділу таблиць бази даних. У колонці Actions навпроти відповідної таблиці знаходиться кнопка з трьома крапками, яка відкриває контекстне меню (рис. 2.28).

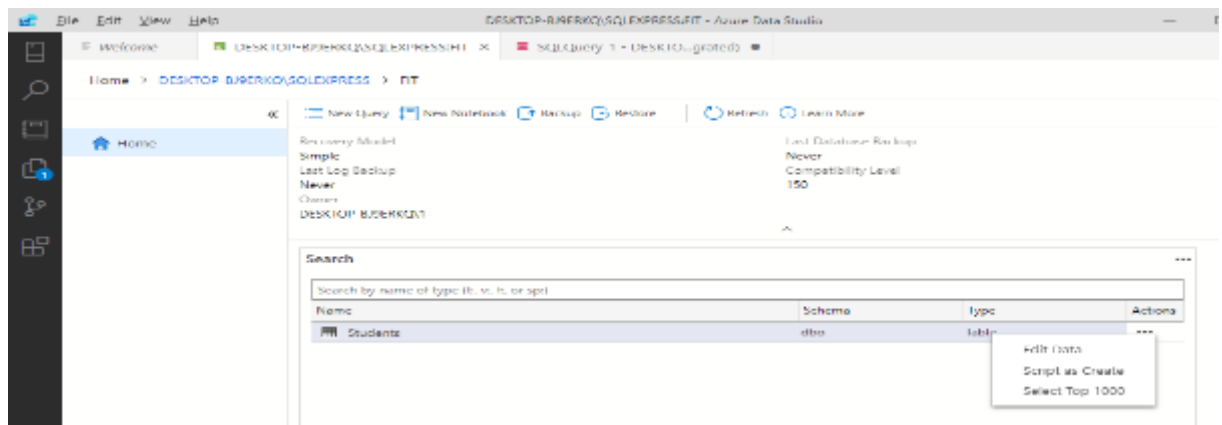


Рисунок 2.28 –Контекстне меню Actions

Для відображення вмісту таблиці потрібно вибрати пункт Select Top 1000, який дає змогу переглянути перші 1000 записів таблиці. Після виконання наведеного вище запиту на додавання нових записів (див. рис. 2.27) можна пересвідчитись, що записи успішно інтегровані до таблиці Students. (рис. 2.29).

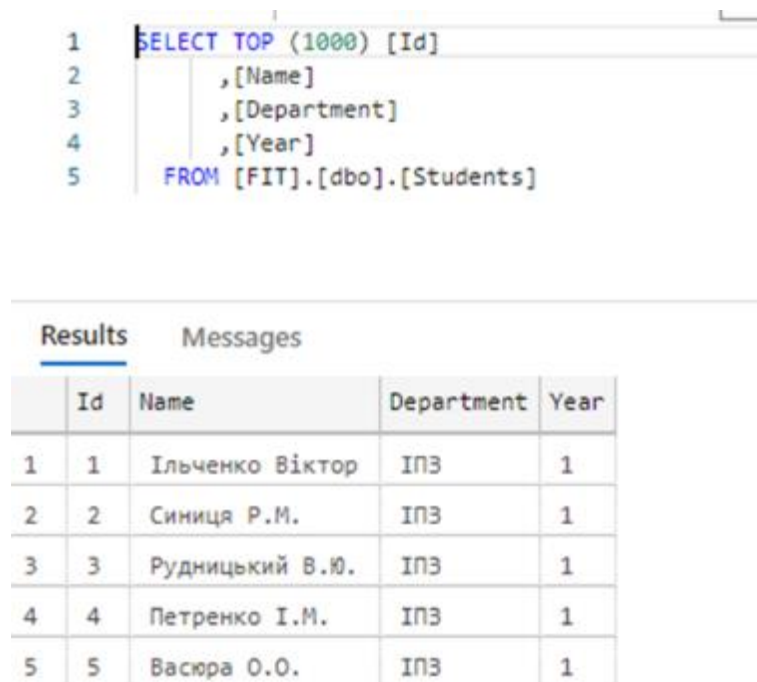


Рисунок 2.29 –Результат перегляду даних таблиці засобами Azure Data Studio

Оскільки таблиця Students містить лише 5 записів (перший рядок було додано вручну, а рядки 2-5 додані за допомогою запиту), у результаті виконання команди Select Top 1000 відображаються всі наявні дані, а не 1000 рядків, як зазначено в назві команди.

2.3.2 Вибірки даних. Оператор SELECT призначений для вибірки даних з бази даних в SQL. Цей оператор відноситься до підмножини DML і має такий синтаксис:

```
SELECT [DISTINCT | ALL]
{* | <величина> [, <величина> ...]}
[INTO: Змінна [, : Змінна ...]]
FROM <tableref> [, <tableref> ...]
[WHERE <умова пошуку>]
[GROUP BY Колонка [, Колонка ...]]
[HAVING <умова пошуку>]
[UNION [ALL] <select_expr>]
[ORDER BY <список сортування>];
```

<Величина> = {Колонка | : Змінна | <Константа>

| <Вираз> | <Функція>

| udf ([<величина> [, <величина> ...]])

| NULL | USER} [AS Псевдонім]

<Константа> = Число | 'Рядок'

<Вираз> = SQL вираз, що повертає одиничне значення

<Функція> =

COUNT (* | [ALL] <величина> | **DISTINCT** <величина>)

| **SUM** ([ALL] <величина> | **DISTINCT** <величина>)

| **AVG** ([ALL] <величина> | **DISTINCT** <величина>)

| **MAX** ([ALL] <величина> | **DISTINCT** <величина>)

| **MIN** ([ALL] <величина> | **DISTINCT** <величина>)

| **CAST** (<величина> **AS** <тип даних>)

| **UPPER** (<величина>)

| **GEN_ID** (Імя_Генератора, <величина>)

<Tableref> = {<joined_table> | table | view

| procedure [(**<величина>** [, **<величина>** ...])]
[Псевдоним]

<Joined_table> = **<tableref>** **<join_type>** **JOIN** **<tableref>**
ON **<умова пошуку>** | (**<Joined_table>**)

<Join_type> = [INNER] | {LEFT | RIGHT | FULL} [OUTER]

<Умова пошуку> =

<Величина> **<оператор порівняння>**

{**<Величина>** | (**<Select_one>**)}

| **<Величина>** [**NOT**] **BETWEEN** **<величина>** **AND** **<величина>**

| **<Величина>** [**NOT**] **LIKE** **<величина>**

| **<Величина>** [**NOT**] **IN**

(**<Величина>** [, **<величина>** ...] | **<select_list>**)

| **<Величина>** **IS** [**NOT**] **NULL**

| **<Величина>** {> = | <=} **<Величина>**

| **<Величина>** [**NOT**] {= | <| >} **<Величина>**

| {**ALL** | **SOME** | **ANY**} (**<select_list>**)

| **EXISTS** (**<select_expr>**)

| **SINGULAR** (**<select_expr>**)

| **<Величина>** [**NOT**] **CONTAINING** **<величина>**

| **<Величина>** [**NOT**] **STARTING [WITH]** **<величина>**

| (**<Умова пошуку>**)

| **NOT** **<умова пошуку>**

| **<Умова пошуку>** **OR** **<умова пошуку>**

| **<Умова пошуку>** **AND** **<умова пошуку>**

<Оператор порівняння> =

{= | <| > | <= | > = | ! <| !> | <> | !=}

<Select_one> = оператор SELECT, що вибирає одну колонку і повертає лише одне значення

<Select_list> = оператор SELECT, що вибирає одну колонку, який повертає нуль або багато значень

<Select_expr> = оператор SELECT, що вибирає кілька величин і повертає нуль або багато значень

<Список сортування> =

{Колонка | Номер} [ASC | DESC]

[, <Список сортування> ...]

Деякі параметри, що входять в оператор SELECT, описані в таблиці 2.4.

Таблиця 2.4. Опис параметрів оператора SELECT

Параметр	Опис
DISTINCT ALL	DISTINCT – запобігає дублюванню даних, які будуть вилучені. ALL (за замовчуванням) – призведе до вилучення всіх
{* <величина> [, <величина> ...]}	Зірочка (*) означає, що треба вивести всі колонки із зазначених таблиць. <Величина> [, <величина> ...] – виводить список зазначених величин
INTO :змінна [, : змінна...]	Використовується тільки в тригерах і збережених процедурах для операторів SELECT , які повертають не більше одного рядка. Вказується список змінних, в які виводяться величини
FROM <tableref> [,<tableref> ...]	Вказує список таблиць, переглядів і збережених процедур, з яких беруться дані. Список може включати з'єднання

Параметр	Опис
table	Ім'я існуючої в базі даних таблиці
view	Ім'я існуючого бази даних перегляду
procedure	Ім'я існуючої збереженої процедури, призначеної для використання в операторі SELECT
Псевдоним	Коротке альтернативне ім'я для таблиці, перегляду або колонки. Після опису в <tableref>, псевдонім може використовуватися для посилань на таблицю або перегляд
join_type	Задає тип з'єднання, яке може бути внутрішнім або зовнішнім.
WHERE <умова_пошуку>	Визначає умову, яка обмежує кількість отриманих рядків
GROUP BY Колонка [, Колонка ...]	Розбиває результат запиту на групи, що включають всі рядки з ідентичними значеннями, зазначеними в списку
HAVING <умова_пошуку>	Використовується спільно з GROUP BY. Задає умову, яка обмежує кількість груп, що виводяться.
UNION [ALL]	Об'єднує результати декількох запитів. Всі запити мають отримувати однакову кількість стовпців, тип даних кожного стовпця першого запиту повинен збігатися з типом даних інших запитів, імена стовпців в різних запитах можуть відрізнятися. Необов'язковий параметр ALL
ORDER BY <список сортування >	Вказує колонки, за якими буде проводитися сортування вказаних рядків. Можна вказувати або імена колонок, або їх порядкові номери у списку видобутих колонок. Якщо вказати ASC , то рядки будуть

Параметр	Опис
	видаватися в порядку зростання значень полів, що відсортовані, якщо DESC – в порядку спадання.

Нехай маємо базу даних **check** яка складається з таких таблиць (рис. 2.30):

Фізичні особи – Person

Контрагенти – Kontragent

Співробітники – Employee

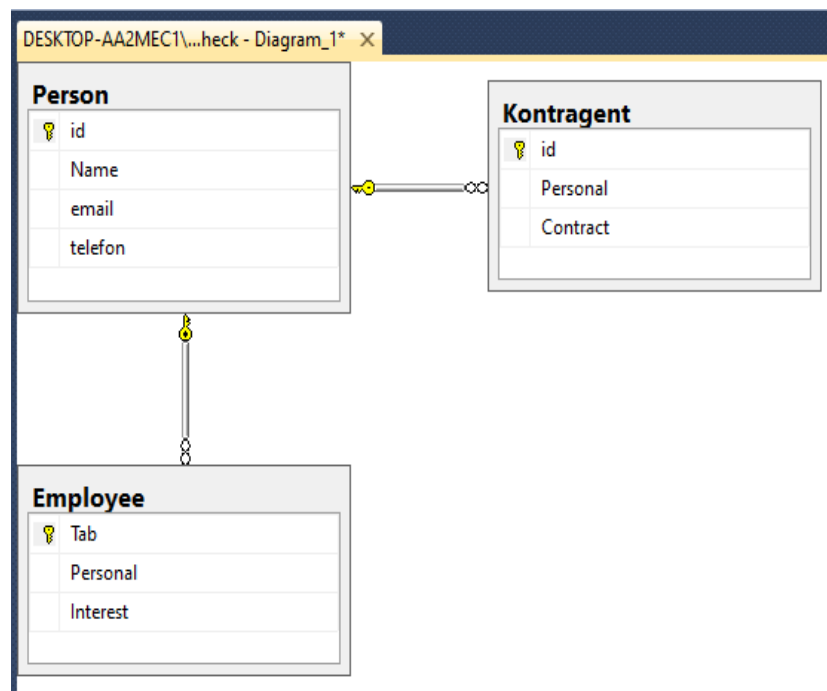


Рисунок 2.30 –Діаграма бази даних **check**

Додаємо нові дані у таблицю Person за допомогою запиту:

```

USE [check]
GO

INSERT Person (Name, email, telefon)
VALUES ('Іванов І.П.', 'abs.gmail.com', '0675144145'),
('Петров О.О.', 'qwerty.gmail.com', '0506137478'),
('Кравченко К.К.', 'travnik.gmail.com', '0506137478'),
('Мартинюк С.Л.', 'rav.ukr.net', '0735137575'),
('Синиця О.В.', 'nirvana.gmail.com', '0666337338'),
('Висоцький М.В.', 'ratusha.ukr.net', '0962141566');
GO

```

В результаті виконання цього запиту у кінці таблиці Person з'явилося 6 нових записів з даними фізичних осіб (рис. 2.31).

	id	Name	email	telefon
	1	Іванов Іван	ivanov@ukr.net	0671947186
	2	Петров Петро	petrov@ukr.net	0733211466
	3	Вагнер Дмитро	NULL	NULL
	4	Ільченко Ната...	NULL	NULL
	10011	Іванов І.П.	abs.gmail.com	0675144145
►	10012	Петров О.О.	qwerty.gmail.c...	0506137478
	10013	Кравченко К.К.	travnik.gmail.c...	0506137478
	10014	Мартинюк С.Л.	rav.ukr.net	0735137575
	10015	Синиця О.В.	nirvana.gmail.c...	0666337338
	10016	Висоцький М.В.	ratusha.ukr.net	0962141566
*	NULL	NULL	NULL	NULL

Рисунок 2.31 –Таблиця Person

Для отримання певної вибірки даних застосовується команда SELECT. У спрощеному вигляді вона має такий синтаксис:

SELECT список_стовпців FROM ім'я_таблиці

Застосуємо команду SELECT для отримання імен фізичних осіб з таблиці Person (рис. 2.32).



Рисунок 2.32 –Вибірка усіх імен фізичних осіб з таблиці Person

Застосуємо команду SELECT для отримання ПІБ осіб та їх електронних скриньок в одному стовпці (рис. 2.33).

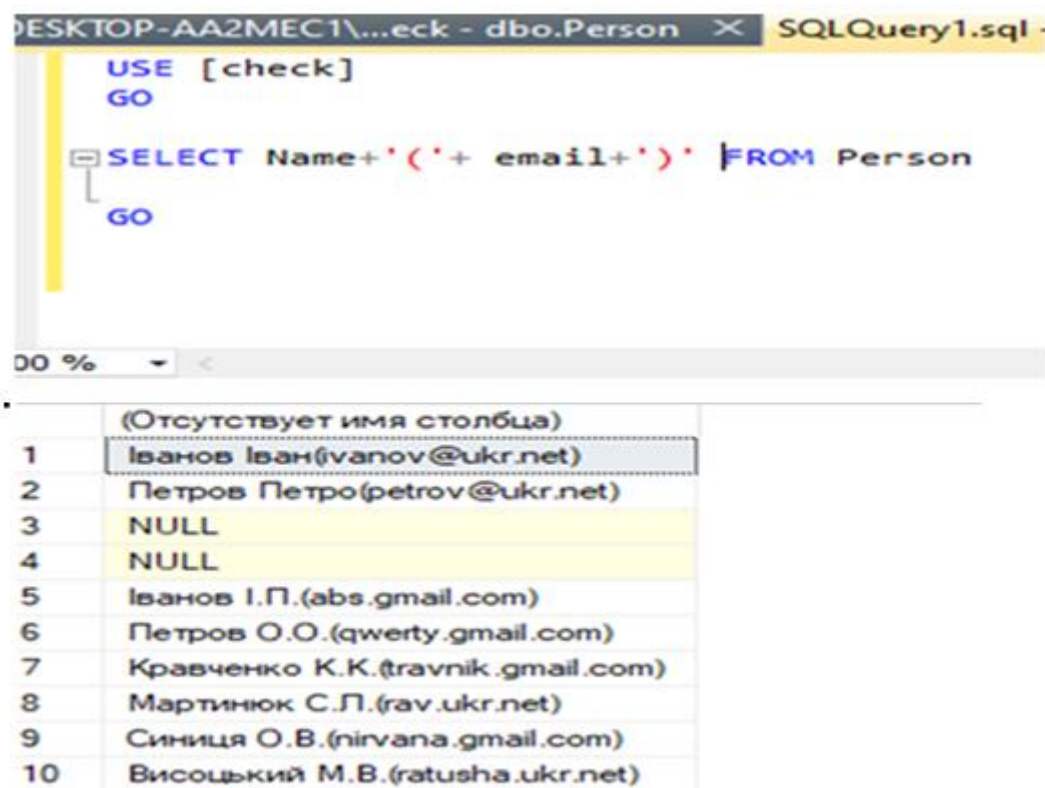


Рисунок 2.33 –Вибірка ПІБ осіб та їх електронних скриньок

Оператор SELECT — це, мабуть, найважливіша команда в мові SQL. Її основне призначення — вибірка даних з однієї або кількох таблиць у базі даних. Вона дозволяє не просто отримати всі дані, а й точно визначити, які саме стовпці, з яких таблиць, за якими критеріями, в якому порядку та в якій формі мають бути представлені.

Знову застосуємо команду SELECT для отримання ПІБ осіб та їх електронних скриньок в одному стовпці, але дамо ім'я цьому стовпцю (рис. 2.34). Так як у деяких людей немає електронної скриньки, замість їх значень вивелось значення NULL.

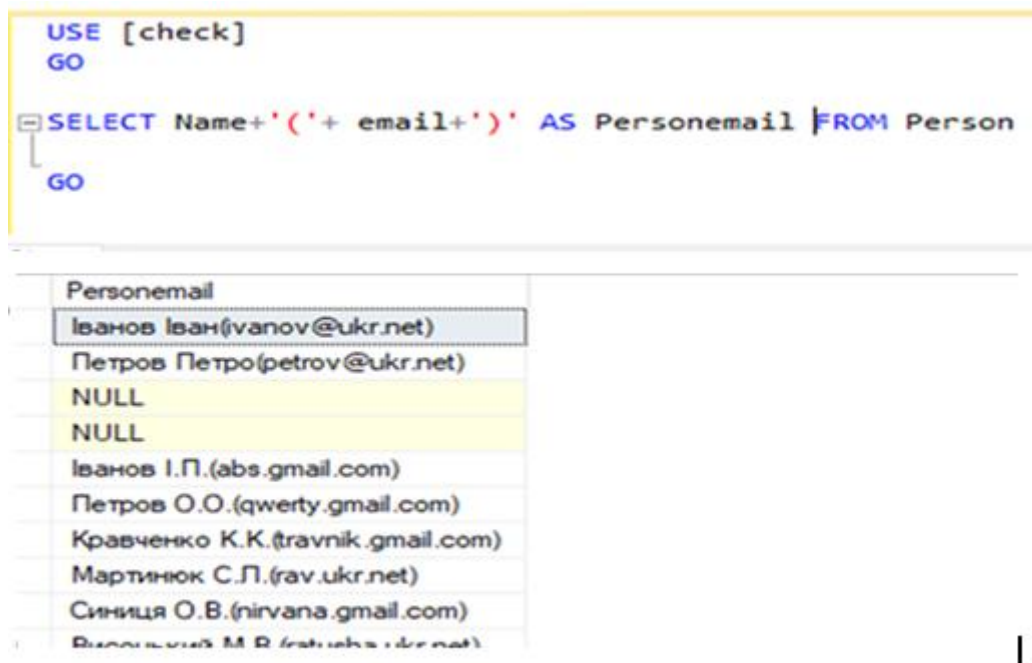


Рисунок 2.34 –Вибірка ПІБ осіб та їх електронних скриньок в одному стовпці з іменем Personemail

Віртуальна таблиця в SQL — це не фізичний об'єкт, який зберігається в базі даних, а динамічний результат запиту. Вона не має власного місця на диску і не містить даних безпосередньо. Замість цього, вона отримує дані з однієї або декількох існуючих фізичних таблиць. Виведемо результат цього запиту у віртуальну таблицю Notes (рис. 2.35).

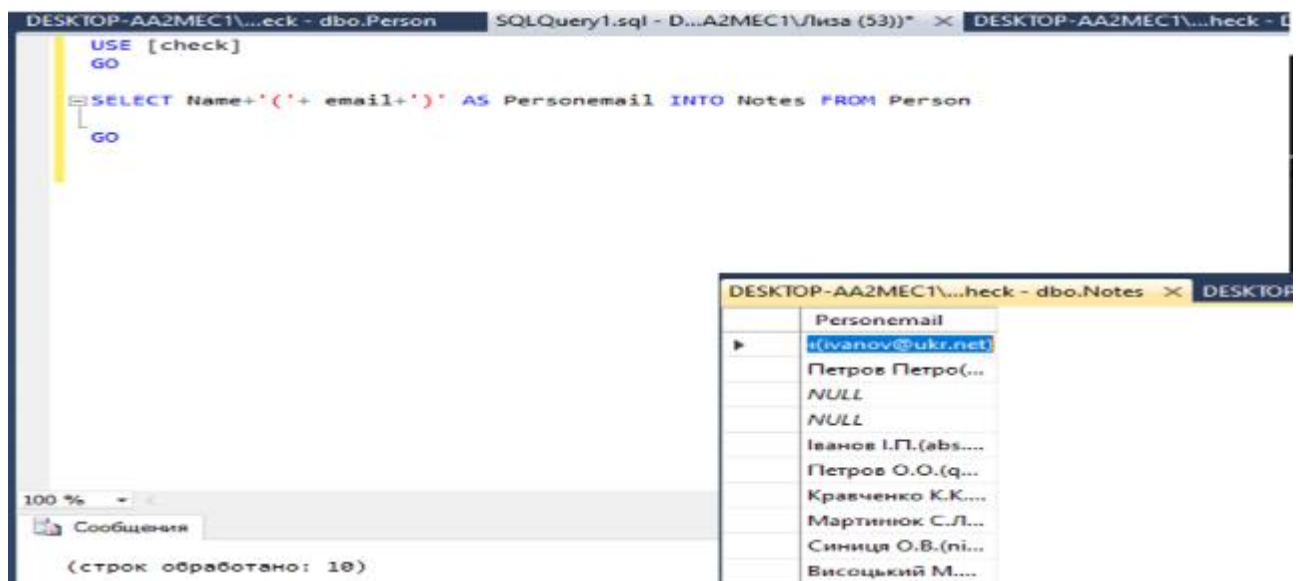


Рисунок 2.35 –Виведення вибірки у віртуальну таблицю Notes

2.3.3 Сортуння. ORDER BY. Оператор ORDER BY призначений для сортування вибірки даних, отриманої за допомогою оператора SELECT з бази даних в SQL.

Застосуємо команду SELECT для отримання усіх фізичних осіб з таблиці Person, але вже за допомогою команди ORDER BY відсортуємо за алфавітом цю вибірку за стовпцем Name:

```
USE [check]
GO

SELECT * FROM Person
ORDER BY Name ;|
GO
```

Результат виконання скрипту показано на рис. 2.36.

Результаты		Сообщения		
	id	Name	email	telefon
1	3	Вагнер Дмитро	NULL	NULL
2	10016	Висоцький М.В.	ratusha.ukr.net	0962141566
3	10011	Іванов І.П.	abs.gmail.com	0675144145
4	1	Іванов Іван	ivanov@ukr.net	0671947186
5	4	Ільченко Наталія	NULL	NULL
6	10013	Кравченко К.К.	travnik.gmail.com	0506137478
7	10014	Мартинюк С.Л.	rav.ukr.net	0735137575
8	10012	Петров О.О.	qwerty.gmail.com	0506137478
9	2	Петров Петро	petrov@ukr.net	0733211466
10	10015	Синиця О.В.	nirvana.gmail.com	0666337338

Рисунок 2.36 –Вибірка фізичних осіб, відсортована за алфавітом за стовпцем Name

Отримаємо вибірку ПІБ осіб та їх електронних скриньок в одному стовпці. Вибірку відсортуємо в зворотному алфавітному порядку (від Я до А), вказавши ключове слово *DESC*:

```
USE [check]
GO

SELECT Name+'('+email+')' AS Personemail FROM Person
ORDER BY Personemail DESC ;
GO
```

Результат виконання скрипту показано на рис. 1.37.

	Personemail
1	Синиця О.В.(nirvana.gmail.com)
2	Петров Петро(petrov@ukr.net)
3	Петров О.О.(qwerty.gmail.com)
4	Мартинюк С.П.(rav.ukr.net)
5	Кравченко К.К.(travnik.gmail.com)
6	Іванов Іван(ivanov@ukr.net)
7	Іванов І.П.(abs.gmail.com)
8	Висоцький М.В.(ratusha.ukr.net)
9	NULL
10	NULL

Рисунок 2.37 –Відсортована в зворотному алфавітному порядку вибірка ПІБ осіб та їх електронних скриньок з таблиці Person в одному стовпці

Вибірку, але вже відсортовану у прямому порядку (від А до Я) можна отримати, якщо вказати параметр ASC:

```
USE [check]
GO

SELECT Name+'('+email+')' AS Personemail FROM Person
ORDER BY Personemail ASC ;
GO
```

За допомогою команди ORDER BY можна сортувати за декількома стовпцями. Отримаємо вибірку лише двох стовпців – імен та поштових скриньок фізичних осіб з таблиці Person. Вибірку відсортуємо за алфавітом за двома стовпцями – спочатку за email, а потім за Name:

```
USE [check]
GO

SELECT Name, email FROM Person
ORDER BY email, Name ;
GO
```

Результат запишемо у два стовпці (рис. 2.38).

	Name	email
1	Вагнер Дмитро	NULL
2	Ільченко Наталія	NULL
3	Іванов І.П.	abs.gmail.com
4	Іванов Іван	ivanov@ukr.net
5	Синиця О.В.	nirvana.gmail.com
6	Петров Петро	petrov@ukr.net
7	Петров О.О.	qwerty.gmail.com
8	Висоцький М.В.	ratusha.ukr.net
9	Мартинюк С.П.	rav.ukr.net
10	Кравченко К.К.	travnik.gmail.com

Рисунок 2.38 –Відсортована спочатку за email, а потім за Name вибірка ПІБ осіб та їх електронних скриньок з таблиці Person в двох стовпцях

Зверніть увагу, що спочатку сортування відбувалося за стовпцем, назва якого йде першою після ключового слова ORDER BY, а потім за стовпцем, назва якого що вказана другою.

3.3.4 TOP – вибірка з вказаної кількості перших рядків таблиці. Для того, щоб виконати вибірку з деякого числа перших рядків таблиці використовують після селекції ключове слово TOP.

Виведемо відсортовану вибірку, яка складається лише з перших чотирьох записів таблиці Person і з одного стовпця Name:

```
USE [check]
GO

SELECT TOP 4 Name FROM Person
ORDER BY Name ;
GO
```

Результат (рис. 2.39):

	Name
1	Вагнер Дмитро
2	Висоцький М.В.
3	Іванов І.П.
4	Іванов Іван

Рисунок 2.39 –Перші чотири записи вибірки

Все те ж саме, що і в минулому прикладі, але цього разу виведемо 75% всіх записів:

```
USE [check]
GO

SELECT TOP 75 Percent Name FROM Person
ORDER BY Name ;
GO
```

Результат (рис. 2.40):

	Name
1	Вагнер Дмитро
2	Висоцький М.В.
3	Іванов І.П.
4	Іванов Іван
5	Ільченко Наталія
6	Кравченко К.К.
7	Мартинюк С.Л.

Рисунок 2.40 –Перші 75% записів вибірки

2.3.5 OFFSET і FETCH – діапазон вибірки. Ключові слова OFFSET і FETCH є параметрами команди ORDER BY. Вони дають змогу обмежити кількість рядків, які повертаються за запитом. Інакше кажучи, ці параметри дають змогу визначити діапазон рядків таблиці, з якого буде зроблена вибірка.

OFFSET визначає кількість рядків, які потрібно пропустити, перш ніж почати повертати рядки за запитом. Після оператора FETCH ставиться ключове слово FIRST або NEXT, а потім вказується кількість рядків, які повертаються за запитом. Ці вибрані рядки (FETCH) вже йдуть після пропущених рядків (OFFSET). У спрощеному вигляді запит має такий синтаксис:

```
SELECT список_стовпців FROM ім'я_таблиці
ORDER BY вираз
OFFSET кількість_пропусків {ROW|ROWS}
FETCH { FIRST | NEXT } кількість_потрібних_рядків { ROW | ROWS }
ONLY
```

Виведемо всі записи стовпця Name із таблиці Person, ігноруючи перших два рядка:

```
USE [check]
GO

SELECT Name FROM Person
ORDER BY id OFFSET 2 ROW
FETCH NEXT 3 ROWS ONLY;
GO
```

Результат (рис. 2.41):

	Name
1	Вагнер Дмитро
2	Ільченко Наталія
3	Іванов І.П.
4	Петров О.О.
5	Кравченко К.К.
6	Мартинюк С.П.
7	Синиця О.В.
8	Висоцький М.В.

Рисунок 2.41 –Пропуск двох перших рядків таблиці

2.3.6 WHERE – фільтрація. Фільтрація – це процес селекції даних за деякою умовою, яка в SQL задається оператором WHERE.

В T-SQL можна застосовувати такі операції порівняння:

- = Порівняння на рівність;
- <> Порівняння на нерівність;
- < Менше ніж;
- > Більше ніж;
- !< Не менш ніж;
- !> Не більше;
- <= Менше ніж або дорівнює;
- >= Більше ніж або дорівнює.

Знайдемо в таблиці Person людину, у якої електронна скринька має назву abc.gmail.com:

```
USE [check]
GO

SELECT * FROM Person
WHERE email= 'abc.gmail.com'
GO
```

Результат:

	id	Name	email	telefon
1	10011	Іванов І.П.	abc.gmail.com	0675144145

Рисунок 2.42 – Результат запиту

2.3.7 Логічні оператори. Для об'єднання декількох умов в одному виразі можуть використовуватися логічні оператори. В T-SQL є такі логічні оператори:

AND: операція логічного І. Вона об'єднує два вирази:

вираз1 AND вираз2

Тільки якщо обидва цих вирази одночасно будуть істинними, то і загальна умова оператора AND також буде істинною.

OR: операція логічного АБО. Вона також об'єднує два вирази:

вираз1 OR вираз2

Якщо хоча б один з цих виразів є істинним, то загальна умова оператора OR також буде істинною.

NOT: операція логічного заперечення. Якщо вираз в цій операції False, то загальна умова True.

NOT вираз

Якщо ці оператори зустрічаються в одному виразі, то спочатку виконується NOT, потім AND і в кінці OR.

Потрібно знайти всіх людей із таблиці Person, у яких електронна скринька буде або abc.gmail.com або qwerty.gmail.com:

```
USE [check]
GO

SELECT * FROM Person
WHERE email= 'abc.gmail.com' OR email= 'qwerty.gmail.com'
GO
```

Результат (рис. 2.43):

	id	Name	email	telefon
1	10011	Іванов І.П.	abc.gmail.com	0675144145
2	10012	Петров О.О.	qwerty.gmail.com	0506137478

Рисунок 2.43 –Приклад роботи оператора OR

2.3.8 IS NULL. Оператор IS NULL перевіряє чи пусте значення у стовпці таблиці.

Виведемо всіх людей із таблиці Person, у яких вказані їх електронні скриньки:

```
USE [check]
GO

SELECT * FROM Person
WHERE email IS NOT NULL;
GO
```

Результат (рис. 2.44):

	id	Name	email	telefon
1	1	Іванов Іван	ivanov@ukr.net	0671947186
2	2	Петров Петро	petrov@ukr.net	0733211466
3	10011	Іванов І.П.	abs.gmail.com	0675144145
4	10012	Петров О.О.	qwerty.gmail.com	0506137478
5	10013	Кравченко К.К.	travnik.gmail.com	0506137478
6	10014	Мартинюк С.П.	rav.ukr.net	0735137575
7	10015	Синиця О.В.	nirvana.gmail.com	0666337338
8	10016	Висоцький М.В.	ratusha.ukr.net	0962141566

Рисунок 2.44 –Результат роботи запиту, який використовує IS NULL

2.3.9 Оператори фільтрації. Оператор фільтрації IN перевіряє чи входить значення стовпця таблиці в заданий масив.

Із таблиці Person виберемо всю інформацію про людей, прізвища яких Іванов Іван, Кравченко К.К. і Синиця О.В.:

```
USE [check]
GO

SELECT * FROM Person
WHERE Name IN ('Іванов Іван', 'Кравченко К.К.', 'Синиця О.В. ');
GO
```

Результат (рис. 2.45):

	id	Name	email	telefon
1	1	Іванов Іван	ivanov@ukr.net	0671947186
2	10013	Кравченко К.К.	travnik.gmail.com	0506137478
3	10015	Синиця О.В.	nirvana.gmail.com	0666337338

Рисунок 2.45 –Результат роботи запиту, який використовує IN

Оператор BETWEEN визначає діапазон значень, яким має відповідати вираз. Після ключового слова BETWEEN вказується початкове і кінцеве значення діапазону таким чином:

WHERE вираз [NOT] BETWEEN початкове_значення AND кінцеве_значення.

Оператор LIKE приймає шаблон рядка, якому має відповідати вираз.

WHERE вираз [NOT] LIKE 'шаблон_рядка'

Шаблон вказується в одинарних лапках. Для визначення шаблону можуть застосовуватися ряд спеціальних символів підстановки:

% Відповідає будь-якому підрядку, наприклад, 'hello w%'. Підрядок може мати будь-яку кількість символів, а може не містити жодного символу;

_ Відповідає будь-якому одиночному символу, наприклад, 'w_rd';

[] Відповідає одному символу, який входить в список символів, вказаних в квадратних дужках, наприклад, [bcdfghklmnpqrstvxz];

[-] Відповідає одному символу з певного діапазону, наприклад, [k-r];

[^] Відповідає одному символу, який не входить в список чи діапазон після символу ^, наприклад, [^aeiouy] чи [^n-q].

Шаблон може містити будь-які символи (зокрема спеціальні) і будь-яку їх кількість, наприклад, 'hell[^a-d] w[aeiouy]rd'.

Із таблиці Person виведемо всіх людей, які використовують скриньку Google.

```
USE [check]
GO

SELECT * FROM Person
WHERE email LIKE '%gmail%';
GO
```

Результат (рис. 2.46):

	id	Name	email	telefon
1	10011	Іванов І.П.	abs.gmail.com	0675144145
2	10012	Петров О.О.	qwerty.gmail.com	0506137478
3	10013	Кравченко К.К.	travnik.gmail.com	0506137478
4	10015	Синиця О.В.	nirvana.gmail.com	0666337338

Рисунок 2.46 –Результат запиту з регулярними виразами

2.3.10 Видалення даних. Команда DELETE. Для видалення застосовується команда DELETE:

```
DELETE [FROM] ім'я_таблиці  
WHERE умова _ видалення
```

Це потужна команда мови маніпулювання даними (DML), яка вимагає обережного використання, оскільки видалені дані, як правило, неможливо відновити без резервної копії.

2.3.11 Оновлення даних. Команда UPDATE. Для зміни вже наявних рядків в таблиці застосовується команда UPDATE. Вона має такий формальний синтаксис:

```
UPDATE ім'я_таблиці  
SET стовпець1 = значення1, стовпець2 = значення2, ... стовпецьN = значенняN  
[FROM вибірка AS псевдонім_вибірки]  
[WHERE умова_оновлення]
```

Збільшимо у всіх товарів ціну на 5000 (рис. 2.17).

SQLQuery3.sql - E:\ENEPC\Eugene (52)*

```
1 USE productsdb;  
2  
3 SELECT * FROM Products  
4  
5 UPDATE Products  
6 SET Price = Price + 5000  
7  
8 SELECT * FROM Products
```

100 %

Results Messages

	Id	ProductName	Manufacturer	ProductCount	Price
1	1	iPhone 6	Apple	3	36000,00
2	2	iPhone 6S	Apple	2	41000,00
3	3	iPhone 7	Apple	5	52000,00
4	4	Galaxy S8	Samsung	2	46000,00
5	5	Galaxy S8 Plus	Samsung	1	56000,00
6	6	Mi6	Xiaomi	5	28000,00
7	7	OnePlus 5	OnePlus	6	38000,00

	Id	ProductName	Manufacturer	ProductCount	Price
1	1	iPhone 6	Apple	3	41000,00
2	2	iPhone 6S	Apple	2	46000,00
3	3	iPhone 7	Apple	5	57000,00
4	4	Galaxy S8	Samsung	2	51000,00
5	5	Galaxy S8 Plus	Samsung	1	61000,00
6	6	Mi6	Xiaomi	5	33000,00
7	7	OnePlus 5	OnePlus	6	43000,00

Рисунок 2.47 –Збільшення ціни в таблиці Product

2.3.12 Мова визначення даних. Мова маніпулювання даними. Мова визначення даних (DDL) в SQL – це підмножина мови SQL, яка використовується для створення та модифікації структури об’єктів бази даних. Вона дає змогу визначати схему бази даних, тобто описувати, як дані організовані та зберігаються. Основні оператори DDL: CREATE, ALTER, DROP, TRUNCATE. Важливо розуміти, що DDL-операції впливають на структуру бази даних, а не на самі дані. Для маніпулювання даними використовуються інші частини SQL, такі як DML (мова маніпулювання даними).

Мова маніпулювання даними (DML) в SQL – це підмножина SQL, яка використовується для додавання, зміни та видалення даних у базі даних. DML зосереджується на маніпулюванні самими даними, а не структурою бази даних. Основні оператори DML: SELECT, INSERT, UPDATE, DELETE.

DML-операції впливають на самі дані в таблицях.

Питання для самоперевірки

Для чого призначена мова визначення даних?

Для чого призначена мова маніпулювання даними?

Яким чином можливо перебрати всі записи із таблиці?

Які є в SQL логічні оператори?

Яким чином відбувається сортування в SQL?

Як в SQL задається умова відбору?

Завдання

Використовуючи командну консоль, напишіть запити на зміну трьох записів у своїх таблицях.

У всіх таблицях баз даних повинен бути первинний ключ.

Використовуючи командну консоль, видаліть один запис із таблиці.

Напишіть запити в Azure Data Studio:

Варіант	Опис завдання
1	1. Змінити всі таблиці бази даних, яка створена була у минулому завданні, так, щоб вони належали до схеми з ім'ям "Dev". 2. За допомогою запиту додати у другу таблицю 5 записів, запит виконати у командному рядку. 3. У командному рядку вивести БД за замовчуванням. 4. Вивести всі факультети, на яких вивчення інформаційних технологій є головним завданням: ФІТ, кібернетика, радіофізика тощо. Вивести всі факультети, які знаходяться у певному корпусі. 5. Вивести всі кафедри, у яких прізвище завідуючого кафедри починається на літеру 'С'.
2	Змінити таблицю Відділи (створену у минулому завданні) таким чином, щоб стовпець Кількість прилавків був зашифрованим. За допомогою запиту додати у таблицю Відділи 5 записів, запит виконати в Azure Data Studio. У командному рядку вивести список БД, не використовуючи збережену процедуру Вивести всі відділи супермаркету для певного залу. Вивести всі відділи супермаркету, кількість прилавків у яких перевищує 3. Вивести всіх співробітників супермаркету, телефон яких використовує зв'язок Lifecell.
3	Змінити таблицю Вид військ, із попереднього завдання, щоб стовпець з назвою був замаскованим. За допомогою запиту додати у таблицю Дислокації 5 записів, запит виконати у командному рядку. Створити таблиці частини і роти. Таблиця частини має поля: номер частини (рядок до 10

	символів), вид військ (коротке ціле), місце дислокації (коротке ціле). Стовпець Місце дислокації повинен бути зашифрованим. Таблиця роти має поля: номер роти (коротке ціле), номер частини (коротке ціле). Вивести всі частини за певним місцем дислокації. Вивести всі роти відсортовані за номером у зворотному до алфавітного порядку. Вивести місця дислокації військ в Україні, площа яких перевищує 10 кв. км.
4	Змінити всі таблиці бази даних, яка створена була у минулому завданні, так, щоб вони належали до схеми з ім'ям "Test". За допомогою запиту додати у таблицю Бібліотеки 2 записи. Запит виконати у командному рядку. У таблицю Фонди бібліотек додати 2 записи, запит виконати в Azure Data Studio. Створити таблицю Співробітники з полями: ПІБ (рядок, довжиною до 100 символів), оклад (грошове), бібліотека (коротке ціле), телефон (рядок, 10 символів). Стовпець Оклад повинен бути зашифрованим. Стовпець Телефон перевіряти на коректність Вивести всіх співробітників бібліотеки у алфавітному порядку, зарплата яких не більша ніж 5000 грн. Вивести всі бібліотеки, розташовані у м. Києві. Зробити вибірку для фондів бібліотеки, в якій вивести сумарну кількість книжок, журналів тощо.
5	Змінити всі таблиці бази даних, яка створена була у минулому завданні, так, щоб вони належали до схеми з ім'ям "Prod". За допомогою запиту додати у таблицю Пансіонати 5 записів, запит виконати в Azure Data Studio. У командному рядку перевірити випуск MS SQL Server. Створити таблицю клієнти з полями: ідентифікатор клієнта

	(ціле, збільшується на 1, з кожним наступним записом), ПІБ (рядок, довжиною 50 символів), вік (коротке ціле), кількість дітей (коротке ціле), телефон (рядок, довжиною 10 символів). Останній стовпчик, повинен бути налаштований на перевірку на коректність. .Створити таблицю тур з полями: ідентифікатор тура (ціле, збільшується на 1, з кожним наступним записом), назва (рядок, довжиною 30 символів), пансіонат (ціле), вартість (грошове). Вивести всіх клієнтів, телефон яких використовує зв'язок KYIVSTAR.
--	---

2.4 Зв'язування таблиць та отримання зв'язаних даних

2.4.1 Обмеження зовнішнього ключа. Зовнішній ключ (Foreign Key) — це стовпець або набір стовпців у таблиці, який встановлює зв'язок з первинним ключем (Primary Key) іншої таблиці. Він служить для забезпечення реферативної цілісності даних, гарантуючи, що посилання між таблицями є коректними. Простіше кажучи, зовнішній ключ в одній таблиці вказує на унікальний рядок в іншій таблиці.

Реферативна цілісність — це головна функція зовнішнього ключа. Він запобігає появі "сирітських" записів, тобто записів у дочірній таблиці, які не мають відповідного запису в батьківській. Це гарантує, що ваша база даних залишається логічно послідовною. Зовнішній ключ може запобігти видаленню або оновленню записів у батьківській таблиці, якщо на них посилаються записи в дочірній таблиці. Це запобігає випадковому видаленню важливих даних. Зовнішні ключі чітко визначають зв'язки між таблицями, що допомагає оптимізатору запитів SQL Server ефективніше виконувати операції JOIN.

Розглянемо таке завдання. Нехай потрібно написати застосунок, який буде виставляти рахунки клієнтам (рис. 2.48).

Зразок заповнення платіжного доручення

Одержувач	Товариство з обмеженою відповідальністю "С.В.С.ГРУП"		
Код	37165039		КРЕДИТ оах. N
Банк одержувача		Код банку	26001313318
АТ "Райффайзен Банк Аваль"		380805	

Рахунок на оплату № 5 від 14 листопада 2018 р.

Постачальник: Товариство з обмеженою відповідальністю "С.В.С.ГРУП"
 П/р 26001313318, Банк АТ "Райффайзен Банк Аваль", МФО 380805
 03131, м. Київ, вул. Любомирська, буд. 3, тел.: (044) 3618907,
 код за ЄДРПОУ 37165039, ІПН 371650326508, № свід. 100308810

Покупець: Київський національний університет імені Тараса Шевченка

Договір: Основний договір

№	Товари (роботи, послуги)	Кіл-сть	Од.	Ціна без ПДВ	Сума без ПДВ
1	Труба водостічна PE RAL 3011 д.150, L=1250мм	50	шт	222,80	11 140,00
2	Нижній елемент труби PE (копію) д.150мм L=590 мм кут. 120	28	шт	318,46	8 916,88
3	Лійки водоприймальні PE д.150мм L=300 мм	5	шт	413,46	2 067,30
4	Кронштейн для кріплення водостічних труб (хомут 20*300мм з пластиною 20*120мм)	72	шт	160,23	11 536,56

Всього: 33 660,74
 Сума ПДВ: 6 732,15
 Всього із ПДВ: 40 392,89

Рисунок 2.48 –Платіжні рахунки клієнтам

Для цього потрібна база даних, яка буде включати такі таблиці:

Фізичні особи

Контрагенти

Співробітники

Товари

Документи

Вже маємо створену базу даних check, яка містить три перші таблиці (рис. 2.49):

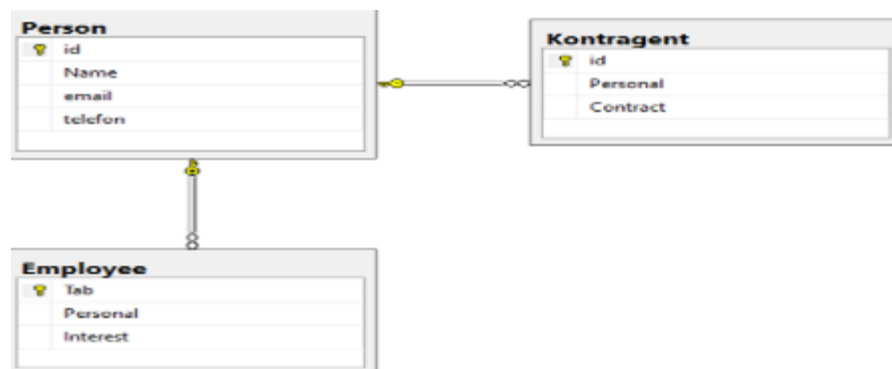


Рисунок 2.49 –Діаграма бази даних check, яка містить 3 таблиці

Потрібно створити ще дві таблиці: Goods і Docs, в яких буде зберігатися інформація про товари і документи відповідно. Створимо таблицю Goods засобами Microsoft SQL Management Studio (рис. 2.50).

	Id	int	☐
	Name	nvarchar(50)	☒
	Price	real	☒
			☐

Рисунок 2.50 – Створення таблиці Goods з трьома полями – ID, назва і ціна товару

Заповнимо цю таблицю даними за допомогою такого запиту:

```
USE [check]
GO

INSERT INTO [dbo].[goods]
    ([Name]
    ,[Price])
VALUES
    ('Пальто жін. сине', 2500.00),
    ('Пальто жін. зимне, демісизонне', 4500.00),
    ('Пальто жін. чорне', 12400.00),
    ('Пальто унісекс. чорне', 5000.00),
    ('Пальто чол. клітинка', 12500.00),
    ('Пальто чол. чорне', 6500.00);

GO
```

У результаті отримаємо таблицю, яка має 6 записів (рис. 2.51).

	Id	Name	Price
	1	Пальто жін. сине	2500
	2	Пальто жін. зимне, демісиз...	4500
	3	Пальто жін. чорне	12400
	4	Пальто унісекс. чорне	5000
	5	Пальто чол. клітинка	12500
	6	Пальто чол. чорне	6500
	NULL	NULL	NULL

Рисунок 2.51 – Таблиця Goods, заповнена даними

Кожен платіжний документ виписує співробітник компанії для клієнта. Тобто таблиця, яка зберігає інформацію про документи, повинна посилатися на співробітника із таблиці Employee і на клієнта із таблиці Kontragent. Водночас в кожному платіжному документі може бути багато товарів, а кожен товар може

бути в багатьох платіжних документах – маємо зв’язок «багато-до-багатьох». Такий зв’язок потребує додаткової таблиці, в якій буде відображатися id документа, що містить певний товар, і id цього товару.

Створимо таблицю, яка зберігає інформацію про документи, за допомогою запиту:

```
CREATE TABLE Docs
(
    Num INT PRIMARY KEY IDENTITY,
    KontragentId Numeric,
    EmployeeId Numeric,
    CreatedAt Date,
    CONSTRAINT FK_Doc_To_Kontragent FOREIGN KEY (KontragentId) REFERENCES Kontragent (Id),
    CONSTRAINT FK_Doc_To_Employee FOREIGN KEY (EmployeeId) REFERENCES Employee (Tab)
);
```

Створимо таблицю зв’язків між документами і товарами за допомогою запиту:

```
USE [check]
GO

CREATE TABLE GoodsDoc
(
    Num INT ,
    GoodsId INT,
    Q INT,
    PRIMARY KEY (Num, GoodsId),
    CONSTRAINT FK_Doc_To_Goods FOREIGN KEY (GoodsId) REFERENCES Goods (Id),
    CONSTRAINT FK_Doc_To_Doc FOREIGN KEY (Num) REFERENCES Docs (Num)
);
```

У цьому випадку ми пов’язали таблиці, вказавши обмеження FOREIGN KEY. Синтаксис цього оператора такий:

CONSTRAINT **назва_обмеження** FOREIGN KEY
(**назва_зовнішнього_ключа**) REFERENCES **таблиця_що_зв’язується**
(**Первинний_ключ_цієї_таблиці**)

У реляційній моделі даних, первинний і зовнішній ключі виконують різні, але взаємодоповнюючі функції для забезпечення цілісності даних. Первинний ключ слугує унікальним ідентифікатором запису в таблиці і, згідно з його

визначенням, не може містити нульових значень. Це гарантує, що кожен запис має чіткий ідентифікатор. На противагу цьому, зовнішній ключ, який встановлює зв'язок з іншою таблицею, може приймати нульові значення, якщо для дочірнього запису відсутнє відповідне посилання.

Фундаментальна відмінність полягає також у їхній кількості в межах однієї таблиці: кожна таблиця може мати лише один первинний ключ, тоді як зовнішніх ключів може бути кілька, що дозволяє встановлювати зв'язки з багатьма іншими таблицями.

У результаті отримаємо базу даних, яка має 5 зв'язаних таблиць (рис. 2.52).

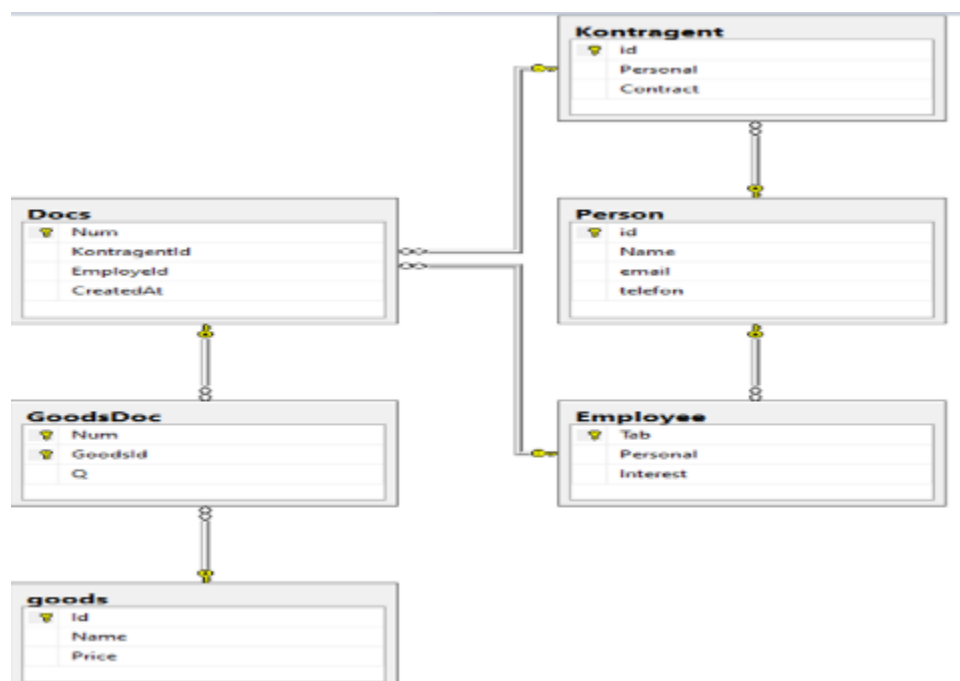


Рисунок 2.52 – Діаграма бази даних check, яка містить 5 таблиць

2.4.2 Додавання даних до зв'язаних таблиць. Для додавання записів до таблиць використовується оператор INSERT INTO.

Спрощений запит на додавання має такий синтаксис:

INSERT INTO **назва_цільової_таблиці** (**поле1**, **поле2**, ...)

VALUES (**значення1**, **значення2**, ...)

Для зв'язування таблиць замість кожного значення, що міститься у круглих дужках після ключового слова VALUES використовується такий запис:

SELECT **поле1**, **поле2**... FROM **назва_таблиці_джерела**

Оператор SELECT може включати ключове слово WHERE для фільтрування даних:

```
SELECT поле1, поле2...
```

```
FROM назва_таблиці_джерела WHERE вираз_для_фільтрування
```

Зверніть увагу на порядок розташування назв полів після операторів SELECT та INSERT INTO, оскільки дані з першого вказаного після SELECT стовпця «поле1» таблиці «назва_таблиці_джерела», будуть незалежно від назви поля заповнені в перший вказаний після оператора INSERT INTO стовпець «поле1» таблиці «назва_цільової_таблиці».

Додамо відфільтровані дані у таблицю Docs, яка пов'язана з таблицями Kontragent та Employee:

```
INSERT INTO Docs ( KontragentId, EmployeeId,
CreatedAt) VALUES (
(SELECT id FROM Kontragent WHERE id = 1),
(SELECT Tab FROM Employee WHERE Tab = 1),
GetDate());
```

Результат (рис. 2.53):



	Num	KontragentId	EmployeeId	CreatedAt
1	1	1	1	2021-09-21
*	NULL	NULL	NULL	NULL

Рисунок 2.53 – Таблиця Docs

Зверніть увагу, що в третій стовпець додано поточну дату.

Додамо дані у таблицю GoodsDoc, яка пов'язана з таблицями Docs і Goods:

```
USE [check]
GO

INSERT INTO GoodsDoc ( Num, GoodsId, Q) VALUES (
(SELECT Num FROM Docs WHERE Num = 1),
(SELECT Id FROM Goods WHERE Id = 1),
5);
```

Зверніть увагу, що в третій стовпець додано просто число 5.

Результат (рис. 2.54):

	Num	GoodsId	Q
▶	1	1	5
•	NULL	NULL	NULL

Рисунок 2.54 – Таблиця GoodsDoc

Ще додамо відфільтровані дані у таблицю Docs:

```
USE [check]
GO

INSERT INTO Docs ( KontragentId, EmployeeId, CreatedAt) VALUES (
(SELECT id FROM Kontragent WHERE id = 2),
(SELECT Tab FROM Employee WHERE Tab = 2),
'2021-05-21');
```

Додамо одразу декілька записів у таблицю GoodsDoc:

```
INSERT INTO GoodsDoc ( Num, GoodsId, Q) VALUES (
(SELECT Num FROM Docs WHERE Num = 3),
(SELECT Id FROM Goods WHERE Id = 4),
2), (
(SELECT Num FROM Docs WHERE Num = 3),
(SELECT Id FROM Goods WHERE Id = 2),
3), (
(SELECT Num FROM Docs WHERE Num = 3),
(SELECT Id FROM Goods WHERE Id = 3),
4) |
```

Загальний результат виконання усіх цих записів подано на рис. 2.55.

PERSON					EMPLOYEE			KONTRAGENT		
id	Name	email	telefon		Tab	Personal	Interent	id	Personal	Contract
1	Іванов Іван	ivanov@ukr.net	0671947186		1	1	5	1	3	105
2	Петров Петро	petrov@ukr.net	0733211498		2	2	10	2	4	106
3	Вогнер Дмитро	NULL	NULL		•	NULL	NULL	•	NULL	NULL
4	Ільченко Наталі	NULL	NULL							
10011	Іванов І.П.	abs@gmail.com	0675144145							
10012	Петров О.О.	qwerty@gmail.c...	0506137478							
10013	Кравченко К.К.	travnik@gmail.c...	0506137478							
10014	Мерлинюк С.Л.	rev.ukr.net	0735137575							
10015	Синица О.В.	nirvana@gmail.c...	0666337338							
10016	Висоцький М.В.	rebusha.ukr.net	0962141566							
•	NULL	NULL	NULL							

Goods			
Id	Name	Price	
1	Пальто жін. сіме	2500	
2	Пальто жін. зимнє, довге...	4500	
3	Пальто жін. чорне	12400	
4	Пальто унісекс. чорне	5000	
5	Пальто чоловіч. кліпінг	12500	
6	Пальто чоловіч. чорне	6500	
•	NULL	NULL	NULL

DOC		
KontragentId	EmployeeId	CreatedAt
1	1	2021-09-21
	2	2021-02-21
	2	2021-05-21

GoodsDoc			
Num	GoodsId	Q	
1	1	5	
2	1	2	
2	2	5	
2	3	4	
3	2	3	

Рисунок 2.55 – Усі таблиці бази даних check

2.4.3 Декартів добуток таблиць. В результаті декартового добутку двох таблиць отримуємо таблицю, яка складається із усіх можливих об'єднань записів однієї таблиці із записами другої таблиці.

Зробимо декартів добуток таблиць Docs і GoodsDoc (рис. 2.56).

```
USE [check]
GO

SELECT * FROM Docs, GoodsDoc;
```

	Num	KontragentId	EmployeId	CreatedAt	Num	GoodsId	Q
1	1	1	1	2021-09-21	1	1	5
2	1	1	1	2021-09-21	2	1	2
3	1	1	1	2021-09-21	2	2	5
4	1	1	1	2021-09-21	2	3	4
5	1	1	1	2021-09-21	3	2	3
6	1	1	1	2021-09-21	3	3	4
7	1	1	1	2021-09-21	3	4	2
8	2	2	2	2021-02-21	1	1	5
9	2	2	2	2021-02-21	2	1	2
10	2	2	2	2021-02-21	2	2	5
11	2	2	2	2021-02-21	2	3	4
12	2	2	2	2021-02-21	3	2	3
13	2	2	2	2021-02-21	3	3	4
14	2	2	2	2021-02-21	3	4	2
15	3	2	2	2021-05-21	1	1	5
16	3	2	2	2021-05-21	2	1	2
17	3	2	2	2021-05-21	2	2	5

Рисунок 2.56 – Декартів добуток таблиць Docs і GoodsDoc

Зверніть увагу, що стовпці йдуть у тому ж порядку, в якому вказані імена таблиць у запиті після ключового слова FROM. Тобто спочатку йдуть усі стовпці таблиці Docs, а потім усі стовпці таблиці GoodsDoc.

Кількість рядків у результаті буде рівна добутку кількості рядків обох таблиць (Docs і GoodsDoc).

Якщо потрібно об'єднувати записи за певним критерієм, то використовується оператор фільтрації (WHERE).

Зробимо декартів добуток таблиць Docs і GoodsDoc, але вже об'єднаємо тільки ті записи, у яких співпадають значення стовпців Num за допомогою ключового слова WHERE. Виведемо лише номер документу, дату документу і id товару (рис. 2.57):

```
USE [check]
GO
```

```
SELECT Docs.Num, Docs.CreatedAt, GoodsDoc.GoodsId FROM Docs, GoodsDoc
WHERE Docs.Num = GoodsDoc.Num;
```

Результат:

	Num	CreatedAt	GoodsId
1	1	2021-09-21	1
2	2	2021-02-21	1
3	2	2021-02-21	2
4	2	2021-02-21	3
5	3	2021-05-21	2
6	3	2021-05-21	3
7	3	2021-05-21	4

Рисунок 2.57 – Декартів добуток таблиць Docs і GoodsDoc з умовою

2.4.4 З'єднання таблиць. З'єднання таблиць може бути вказано за допомогою оператора WHERE або оператора FROM. Можна виділити такі категорії: внутрішні, зовнішні та перехресні з'єднання.

Внутрішні з'єднання використовують оператори порівняння для встановлення відповідності записів з двох сутностей на основі значень загальних стовпців в кожній таблиці. Виведемо телефони всіх співробітників

```
USE [check]
GO
```

```
SELECT Person.Name, Person.telefon, Employee.Tab
FROM Person
JOIN Employee ON Person.Id = Employee.Personal;
```

Результат (рис. 2.58):

	Name	telefon	Tab
1	Іванов Іван	0671947186	1
2	Петров Петро	0733211466	2

Рисунок 2.58 – Внутрішнє з'єднання таблиць

Зовнішні з'єднання (Outer Joins) — це потужний тип операції JOIN в SQL, який використовується для об'єднання записів з двох або більше таблиць, але, на відміну від внутрішніх з'єднань (INNER JOIN), він не вимагає збігу значень у

всіх таблицях. Зовнішні з'єднання повертають не тільки записи, які відповідають умові з'єднання, а й незбіжні записи з однієї або обох таблиць.

Зовнішні з'єднання бувають лівими, правими і повними. Якщо зовнішні з'єднання задаються в FROM, вони вказуються з одним з таких наборів ключових слів:

LEFT JOIN або LEFT OUTER JOIN

RIGHT JOIN або RIGHT OUTER JOIN

FULL JOIN або FULL OUTER JOIN

Результуючий набір лівого зовнішнього з'єднання включає всі рядки з лівої таблиці, заданої в реченні LEFT OUTER, а не тільки ті, в яких сполучаються стовпці, що відповідають один одному. Якщо рядок в лівій таблиці не має збігів з рядком в правій таблиці, результуючий набір рядків містить значення NULL для всіх стовпців списку вибору з правої таблиці.

Виведемо інформацію про співробітників з використанням LEFT JOIN:

```
USE [check]
GO

SELECT *
FROM Person
LEFT JOIN Employee ON Person.Id = Employee.Personal;
```

Результат (рис. 2.59):

	id	Name	email	telefon	Tab	Personal	Interest
1	1	Іванов Іван	ivanov@ukr.net	0671947186	1	1	5
2	2	Петров Петро	petrov@ukr.net	0733211466	2	2	10
3	3	Вагнер Дмитро	NULL	NULL	NULL	NULL	NULL
4	4	Ільченко Наталія	NULL	NULL	NULL	NULL	NULL
5	10011	Іванов І.П.	abs.gmail.com	0675144145	NULL	NULL	NULL
6	10012	Петров О.О.	qwerty.gmail.com	0506137478	NULL	NULL	NULL
7	10013	Кравченко К.К.	travnik.gmail.com	0506137478	NULL	NULL	NULL
8	10014	Мартинюк С.П.	rav.ukr.net	0735137575	NULL	NULL	NULL
9	10015	Синиця О.В.	nirvana.gmail.com	0666337338	NULL	NULL	NULL
10	10016	Висоцький М.В.	ratusha.ukr.net	0962141566	NULL	NULL	NULL

Рисунок 2.59 – Ліве з'єднання таблиць

Праве зовнішнє з'єднання є зворотним для лівого зовнішнього з'єднання. Повертаються всі рядки правої таблиці. Для лівої таблиці повертаються значення NULL кожен раз, коли рядок правої таблиці не має збігів рядків в лівій таблиці.

Виведемо інформацію про співробітників з використанням RIGHT JOIN:

```
USE [check]
GO

SELECT *
FROM Person
RIGHT JOIN Employee ON Person.Id = Employee.Personal;
```

Результат (рис. 2.60):

	id	Name	email	telefon	Tab	Personal	Interest
1	1	Іванов Іван	ivanov@ukr.net	0671947186	1	1	5
2	2	Петров Петро	petrov@ukr.net	0733211466	2	2	10

Рисунок 2.60 – Ліве з'єднання таблиць

Повне зовнішнє з'єднання повертає всі рядки з правої і лівої таблиці. Кожен раз, коли рядок не має відповідних значень в іншій таблиці, стовпці списку вибору іншої таблиці містять значення NULL. Якщо між таблицями є відповідність, весь рядок результуючого набору містить значення даних з базових таблиць.

Перехресне з'єднання повертає всі рядки з лівої таблиці. Кожен рядок з лівої таблиці з'єднується з усіма рядками з правої таблиці. Перехресні з'єднання називаються також декартовим добутком.

Таблиці або подання в реченні FROM можуть зазначатися в будь-якому порядку з внутрішнім з'єднанням або повним зовнішнім з'єднанням. Однак важливий порядок таблиць або подань, заданих з використанням лівого або правого зовнішнього з'єднання:

```
USE [check]
GO

SELECT *
FROM Person
CROSS JOIN Employee ;
```

Результат (рис. 2.61):

Результаты		Сообщения					
	id	Name	email	telefon	Tab	Personal	Interest
1	1	Іванов Іван	ivanov@ukr.net	0671947186	1	1	5
2	2	Петров Петро	petrov@ukr.net	0733211466	1	1	5
3	3	Вагнер Дмитро	NULL	NULL	1	1	5
4	4	Ільченко Наталія	NULL	NULL	1	1	5
5	10011	Іванов І.П.	abs.gmail.com	0675144145	1	1	5
6	10012	Петров О.О.	qwerty.gmail.com	0506137478	1	1	5
7	10013	Кравченко К.К.	travnik.gmail.com	0506137478	1	1	5
8	10014	Мартинюк С.П.	rav.ukr.net	0735137575	1	1	5
9	10015	Синиця О.В.	nirvana.gmail.com	0666337338	1	1	5
10	10016	Висоцький М.В.	ratusha.ukr.net	0962141566	1	1	5
11	1	Іванов Іван	ivanov@ukr.net	0671947186	2	2	10
12	2	Петров Петро	petrov@ukr.net	0733211466	2	2	10
13	3	Вагнер Дмитро	NULL	NULL	2	2	10
14	4	Ільченко Наталія	NULL	NULL	2	2	10
15	10011	Іванов І.П.	abs.gmail.com	0675144145	2	2	10
16	10012	Петров О.О.	qwerty.gmail.com	0506137478	2	2	10
17	10013	Кравченко К.К.	travnik.gmail.com	0506137478	2	2	10
18	10014	Мартинюк С.П.	rav.ukr.net	0735137575	2	2	10

Рисунок 2.61 – Перехрестне з'єднання таблиць

Питання для самоперевірки

Назвіть алгоритм складання багатотабличних запитів.

Що таке з'єднання? Які категорії з'єднань у реляційних БД ви знаєте?

Назвіть різницю між декартовим добутком і перехресним з'єднанням.

Покажіть різницю між лівим зовнішнім з'єднанням і правим?

Завдання

Продовжуйте працювати зі своєю БД. **Таблиці мають бути вже зв'язані зовнішнім ключем. Всі обмеження повинні в БД мати місце.**

Виконайте в консолі:

Декартовий добуток двох таблиць із БД. Таблиці на ваш вибір.

Ліве зовнішнє з'єднання двох таблиць із БД (Таблиці ті самі).

Праве зовнішнє з'єднання двох таблиць із БД (Таблиці ті самі).

Виконайте запити засобами Microsoft SQL Management Studio – індивідуальні завдання за варіантом (подані нижче). **Побудуйте діаграму БД.**

Варіант	Опис завдання
1	1. Створити таблицю групи з полями: номер групи (рядок, довжиною 5 символів) – первинний ключ, кафедра (рядок, довжиною 5 символів). Додати в неї 3 записи. 2. Створити таблицю студенти з полями: номер залікової книги (рядок довжиною 7 символів) – первинний ключ, ПІБ (рядок), група (рядок, довжиною 5 символів). Додати в неї 10 записів. 3. Створити таблицю викладачі з полями: Табельний номер (рядок, довжиною 5 символів) – первинний ключ, ПІБ (рядок), кафедра (рядок, довжиною 5 символів).
2	Створити таблицю посади з полями: ідентифікатор посади (коротке ціле, кожний наступний запис має на 1 більше), назва посади (рядок 50 символів), оклад (грошовий), відділ (коротке ціле). Зв'язати її з таблицями Співробітники і Відділи. Додати в неї 5 записів. Змінити записи в таблиці Співробітники. Створити таблицю товари з полями: артикул товару (рядок, довжиною 10 символів), назва товару (рядок, довжиною 50 символів), одиниця виміру (рядок, довжиною 5 символів), ціна товару (грошовий), виробник (рядок, довжиною 50 символів). 3. Створити таблицю продажу товарів, яка матиме такі атрибути: Співробітник-продавець товару, дата і час продажу, назва і кількість товару. Зв'язати її з іншими таблицями. Додати в неї 5 записів.
3	1. Створити таблицю особового складу військовослужбовців, яка має такі атрибути: ПІБ, рота, посада, рік народження, рік

	надходження на службу, вислуга (кількість років), нагороди, участь у військових заходах. Зв'язати з іншими таблицями. Додати в неї 5 записів. 2. Вивести весь особовий склад з назвою частини, роти, виду військ. 3. Вивести інформацію про військовослужбовців, і про їх дислокацію певної частини на певну дату.
4	1. Створити таблицю поповнення фонду бібліотеки, яка має такі атрибути: фонд, співробітник (який додав літературу у фонд), дата, назва джерела літератури, тип літератури, видавництво, дата видання, кількість примірників. Додати в неї 5 записів. 2. Вивести назви і адреси бібліотек, фонд яких перевищує 1000 екземплярів з типом літератури «Наукова». 3. Вивести інформацію про бібліотеку, в якій працює певний співробітник.
5	1. Створити таблицю продажу путівок, яка має такі атрибути: ПІБ клієнта, назва пансіонату, вид житла, дата заїзду, дата від'їзду, наявність дітей, наявність мед. страховки, к-ть чоловік, ціна за 1 особу, загальна сума. Додати в неї 5 записів. 2. Вивести всіх клієнтів пансіонату, дата заїзду яких не пізніше вказаної дати (наприклад, 01.09.2023 р.) для певного туру по Європі. 3. Вивести всі тури відсортовані в алфавітному порядку за назвою.

2.5 Управління безпекою та використання функцій

2.5.1 Система безпеки SQL сервера. Користувачі БД можуть мати різні права і ролі по відношенню до конкретної БД: одні користувачі можуть

додавати, видаляти, редагувати записи у БД, інші лише переглядати дані з цієї БД задля безпеки і збереження даних. Тому створення користувача і надання йому ролей в SQL має велике значення.

В SQL Server впроваджено два рівні безпеки: на рівні сервера і на рівні баз даних. На рівні сервера система безпеки працює з обліковим записом (login), автентифікацією і з вбудованими ролями сервера. На рівні бази даних система безпеки працює з користувачами (user), роллю в базі даних (рис. 2.62).

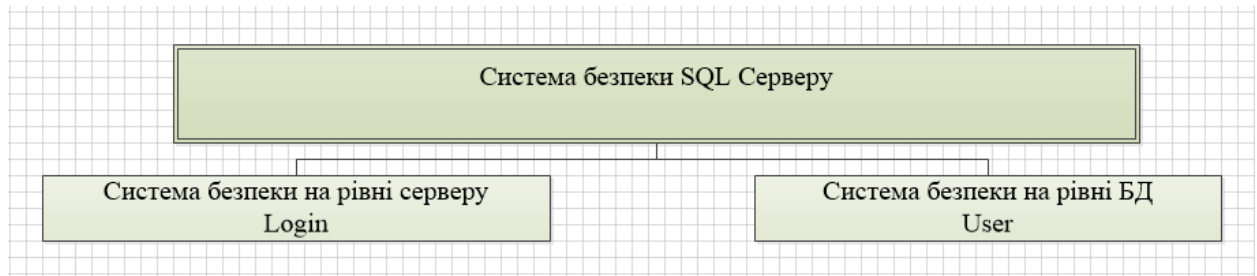


Рисунок 2.62 – Організація системи безпеки в SQL сервері

2.5.2 Створення облікового запису. Створення облікового запису може відбуватися шляхом використання системних процедур або засобами мови SQL.

Взагалі, для створення облікового запису потрібно виконати три етапи:

Створити обліковий запис;

Додати доступ до БД;

Надати користувачу привілеї.

Розглянемо спочатку створення облікового запису сервера шляхом системних процедур. Щоб додати новий обліковий запис використовують процедуру `sp_addlogin`, яка має такий формат:

```
sp_addlogin  
[@login=] 'обліковий_запис'  
[, [@password=] 'пароль']  
[, [@defdb=] 'база_даних_за_замовчуванням']
```

Створимо новий обліковий запис сервера з ім'ям Kseniia і паролем – 2. Для цього створимо новий запит:

```
exec sp_addlogin @loginame = 'Kseniia', @passwd='2'
```

Виконаємо цей запит і перевіримо чи правильно спрацював. Для цього у провіднику об'єктів SSMS відкриємо гілку Security, а потім теку Logins (рис. 2.63).

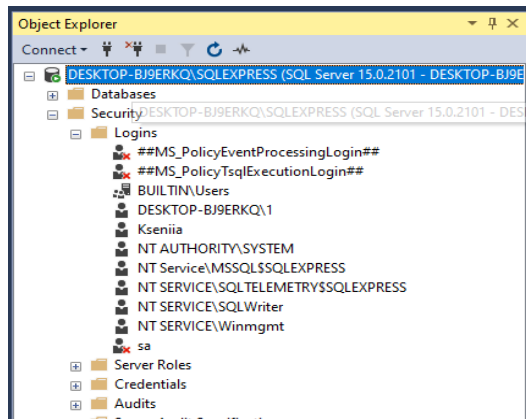


Рисунок 2.63 – В теці Logins з'явився обліковий запис Kseniia

Також створення облікового запису може відбуватися за допомогою команди CREATE. Синтаксис команди для SQL Server у цьому разі такий:

```
CREATE LOGIN login_name { WITH <option_list1> | FROM <sources> }
```

```
<option_list1> ::=
```

```
    PASSWORD = { 'password' | hashed_password HASHED } [
MUST_CHANGE ]
```

```
    [ , <option_list2> [ ,... ] ]
```

```
<option_list2> ::=
```

```
    SID = sid
```

```
    | DEFAULT_DATABASE = database
```

```
    | DEFAULT_LANGUAGE = language
```

```
    | CHECK_EXPIRATION = { ON | OFF }
```

```
    | CHECK_POLICY = { ON | OFF }
```

```
    | CREDENTIAL = credential_name
```

```
<sources> ::=
```

```
    WINDOWS [ WITH <windows_options> [ ,... ] ]
```

```
    | CERTIFICATE certname
```

```
    | ASYMMETRIC KEY asym_key_name
```

```
<windows_options> ::=
```

DEFAULT_DATABASE = database

| DEFAULT_LANGUAGE = language

Повторимо попередній приклад, але ім'я користувача вже буде Ivanov, а пароль – qwerty1111. Створимо новий запит:

```
create login Ivanov with password = 'qwerty'
```

Перевіримо чи правильно спрацював запит (рис. 2.64).

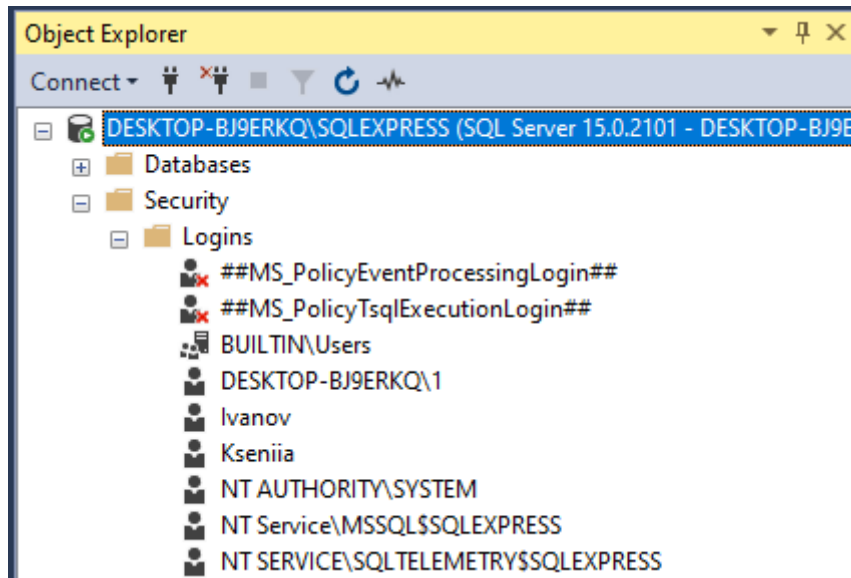


Рисунок 2.64 – В теці Logins з'явився обліковий запис Ivanov

2.5.3 Створення користувача. Для роботи з користувачами в SQL Server використовують або засоби SQL, або збережені процедури.

Обліковий запис користувача асоціюється з користувачем конкретної бази даних, до якої має доступ., Щоб додати користувача за допомогою збереженої процедури, використовують процедуру sp_adduser, яка має такий формат:

sp_adduser

[@loginame=] 'обліковий_запис'

[, [@name_in_db=] 'ім'я_користувача']

[, [@grpname=] 'ім'я_ролі'].

Створимо користувача з ім'ям abc_operator для облікового запису Kseniia.

Для рішення цього завдання створимо і виконаємо новий запит:

```
exec sp_adduser 'Kseniia', 'abc_operator'
```

Для створення користувача БД використовується команда CREATE, яка має такий формат:

```
CREATE USER user_name  
[  
    { FOR | FROM } LOGIN login_name  
]  
[ WITH <limited_options_list> [ ,... ] ]  
[ ; ]
```

-- Users that authenticate at the database

```
CREATE USER  
{  
    windows_principal [ WITH <options_list> [ ,... ] ]  
  
    | user_name WITH PASSWORD = 'password' [ , <options_list> [ ,... ] ]  
    | Azure_Active_Directory_principal FROM EXTERNAL PROVIDER  
}  
  
[ ; ]
```

-- Users based on Windows principals that connect through Windows group

logins

```
CREATE USER  
{  
    windows_principal [ { FOR | FROM } LOGIN windows_principal ]  
    | user_name { FOR | FROM } LOGIN windows_principal  
}  
[ WITH <limited_options_list> [ ,... ] ]  
[ ; ]  
  
-- Users that cannot authenticate
```

```
CREATE USER user_name
```

```

{
    WITHOUT LOGIN [ WITH <limited_options_list> [ ,... ] ]
    | { FOR | FROM } CERTIFICATE cert_name
    | { FOR | FROM } ASYMMETRIC KEY asym_key_name
}
[ ; ]
<options_list> ::=
    DEFAULT_SCHEMA = schema_name
    | DEFAULT_LANGUAGE = { NONE | lcid | language name | language
alias }
    | SID = sid
    | ALLOW_ENCRYPTED_VALUE_MODIFICATIONS = [ ON | OFF ] ]
<limited_options_list> ::=
    DEFAULT_SCHEMA = schema_name ]
    | ALLOW_ENCRYPTED_VALUE_MODIFICATIONS = [ ON | OFF ] ]
-- SQL Database syntax when connected to a federation member
CREATE USER user_name
[ ; ]
-- Syntax for users based on Azure AD logins for Azure SQL Managed
Instance
CREATE USER user_name
    [ { FOR | FROM } LOGIN login_name ]
    | FROM EXTERNAL PROVIDER
    [ WITH <limited_options_list> [ ,... ] ]
[ ; ]
<limited_options_list> ::=
    DEFAULT_SCHEMA = schema_name
    | DEFAULT_LANGUAGE = { NONE | lcid | language name | language
alias }

```

| ALLOW_ENCRYPTED_VALUE_MODIFICATIONS = [ON | OFF]]

Створимо користувача з ім'ям operator для бази даних CheckList для облікового запису Kseniia. Для рішення цього завдання створимо і виконаємо новий запит:

```
SQLQuery3.sql - D...OP-BJ9ERKQ\1 (54))*  SQLQuery2.sql - D...OP-BJ9ERKQ
use CheckList
create user operator for login Kseniia
```

Для того щоб перевірити, як він спрацював, у провіднику об'єктів SSMS розкриємо вузол бази даних CheckList, відкриємо гілку Security, а потім теку Users (рис. 2.65).

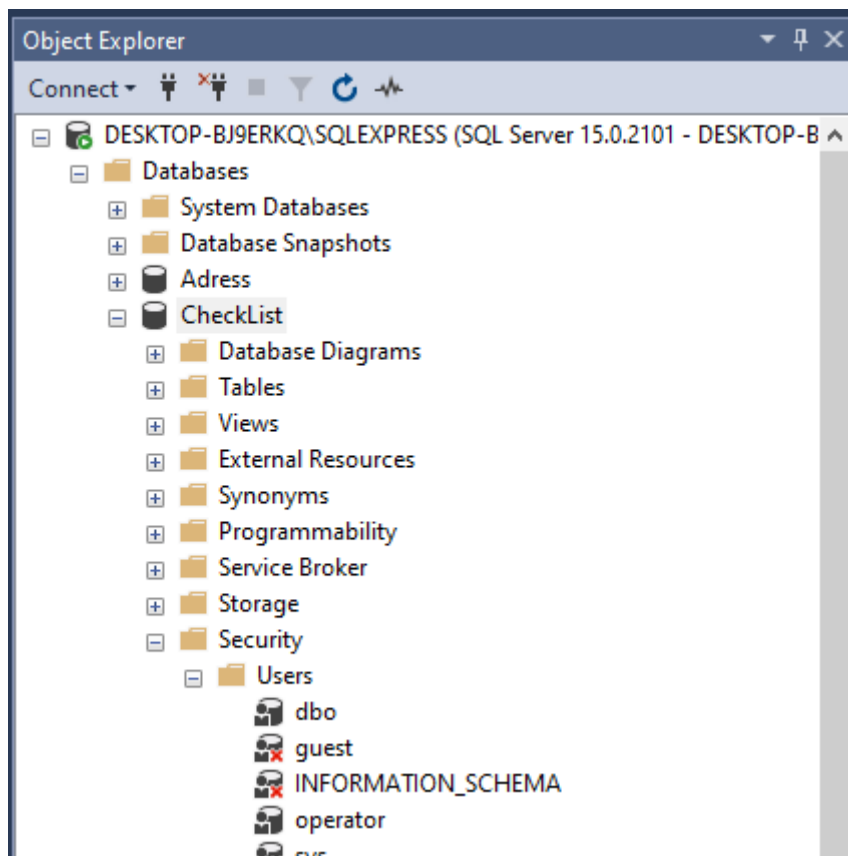


Рисунок 2.65 – В теці Users з'явився користувач operator

2.5.4 Створення ролей. Шляхом створення ролей надаються права і встановлюються обмеження користувачів при роботі з БД. Роль об'єднує користувачів в одну групу. Ролі можна створювати окремо для сервера і окремо для бази даних.

В SQL Server є так звані фіксовані ролі, які не можна видаляти чи редагувати, тобто, не можна змінювати права доступу. Фіксовані ролі також є окремо для сервера (таблиця 2.5) і окремо для бази даних (таблиця 2.6). Для надання користувачу прав, які мають фіксовані ролі, потрібно його включити у ці ролі.

Таблиця 2.5. Фіксовані ролі сервера

Роль	Опис
Sysadmin	Право виконання будь-яких функцій SQL-сервера
Dbcreator	Право на створення бази даних.
Diskadmin	Право на управління файлами на диску.
Processadmin	Право на керування з'єднаннями, запусками і зупинкою екземплярів SQL Server
Securityadmin	Право на створення та управління обліковими записами, право «скинути» пароль облікового запису. Управління дозволами на рівні сервера і на рівні бази даних (при наявності доступу до бази даних).
Serveradmin	Об'єднує можливості ролей diskadmin і processadmin, дає змогу змінювати параметри конфігурації на рівні сервера і вимикати сервер.
Setupadmin	Право на додавання і видалення пов'язаних серверів.
Public	Кожен обліковий запис має цю роль

Таблиця 2.6. Фіксовані ролі баз даних

Роль	Опис
db_owner	Право повного доступу до бази даних
db_accessadmin	Право додавання і видалення користувачів
public	Роль за умовчанням, наявна в кожній базі даних

Роль	Опис
db_securityadmin	Право на управління складом ролей
db_backupoperator	Право на можливість створювати резервні копії бази даних.
db_ddladmin	Право на виконання DDL-інструкцій
db_datareader	Право на читання даних з усіх призначених для користувача таблиць, подань і функцій.
db_denydatareader	Заборона на читання даних з усіх призначених для користувача таблиць, подань і функцій.
db_datawriter	Право додавати, видаляти або змінювати дані в усіх призначених для користувача таблицях.
db_denydatawriter	Заборона додавати, змінювати або видаляти дані в призначених для користувача таблицях.

Різні дії по відношенню до ролі здійснюються за допомогою спеціальних процедур (таблиця 2.7).

Таблиця 2.7. Системні процедури, які опрацьовують ролі

Функція	Параметри	Призначення
sp_addrole	[@rolename=] 'ім'я_ролі' , [@ownername=] 'ім'я_власника']	створення нової ролі
sp_addrolemember	[@rolename=] 'ім'я_ролі' [@membername=] 'ім'я_користувача'	додавання користувача до ролі
sp_droprolemember	[@rolename=] 'ім'я_ролі' [@membername=]	видалення користувача з ролі

Функція	Параметри	Призначення
	‘ім’я_користувача’	
sp_droprole	[@rolename=] ‘ім’я_ролі’	видалення ролі

Додамо роль з ім’ям operator_insert, яка надає право створення таблиці:

```
exec sp_addrole operator_insert
grant create table to operator_insert
```

Ролі можливо створювати також за допомогою команди CREATE.

Формат команди такий:

CREATE ROLE **role_name** [AUTHORIZATION **owner_name**]

Створимо обліковий запис для сервера adminIpz і для нього створимо користувача myAdmin. Далі створимо роль MainIpz і надамо їй права для перегляду записів БД, вставки, редагування та видалення. Роль дамо нашому новому користувачу. Виконаємо це за допомогою п’яти запитів:

```
CREATE LOGIN adminIpz
WITH PASSWORD = 'qwerty';

CREATE USER myAdmin
FOR LOGIN adminIpz;

CREATE ROLE MainIpz;

GRANT SELECT, INSERT, UPDATE, DELETE
TO MainIpz;

ALTER ROLE MainIpz
ADD MEMBER myAdmin;
```

Інструкцією GRANT можливо надати користувачеві або ролі деякі права, а інструкцією REVOKE – відкликати вибрані права.

Інструкцією ALTER можливо змінити обліковий запис, користувача чи роль, а інструкцією DROP – видалити їх.

2.5.5 Вбудовані функції

Категорії, на які поділяються вбудовані функції SQL, подано в таблиці 2.8.

Таблиця 2.8. Категорії вбудованих функцій

Функція	Опис
---------	------

Функція	Опис
Функції, які повертають набір рядків	Повертають об'єкт, який можна використовувати так само, як табличні посилання в SQL-інструкції.
Агрегатні функції	Обробляють колекцію значень і повертають одне результуюче значення.
Ранжуючі функції	Повертають ранжуючі значення для кожного рядка в секції.
Скалярні функції (опис надається далі)	Обробляють і повертають одиночне значення. Можна застосовувати всюди, де допустимим є вираз.
Функції та типи даних дати і часу	Виконують операції над вихідними значеннями дати і часу, повертають символічні та числові значення, а також дату і час.
Логічні функції	Виконання логічних операцій.
Математичні функції	Виконують обчислення, засновані на числових значеннях, переданих функції у вигляді аргументів, і повертають числові значення.
Функції метаданих	Повертають відомості про бази даних та об'єкти баз даних.
Функції безпеки	Повертають дані про користувачів і ролі.
Функції опрацювання символічних даних	Виконують операції над символічним (char або varchar) вхідним значенням і повертають символічне або числове значення.

Функція	Опис
Системні функції	Виконують операції над значеннями, об'єктами і параметрами примірника SQL Server і повертають відомості про них.
Системні статистичні функції	Повертають статистичні відомості про систему.
Функції обробки тексту та зображень	Виконують операції над текстовими або графічними вхідними значеннями або стовпцями.

2.5.6 Агрегатні функції. Аргументами агрегатних функцій можуть бути як стовпці таблиць, так і результати виразів над ними. Агрегатні функції і самі можуть включатися в інші арифметичні вирази.

До агрегатних належать такі функції:

AVG – повертає середнє значення;

MIN – повертає мінімум;

CHECKSUM_AGG – повертає контрольну суму значень в групі. Значення NULL не враховуються;

SUM – повертає суму;

COUNT – повертає кількість;

STDEV – повертає середньоквадратичне відхилення;

COUNT_BIG – повертає кількість елементів у групі;

STDEVP – повертає статистичне стандартне відхилення всіх значень в зазначеному виразі;

GROUPING – перевіряє, чи є вказаний вираз стовпця в списку GROUP BY статистичними, чи ні. В результуючому наборі функція GROUPING повертає 1 (статистичний вираз) або нуль (нестатистичний вираз);

VAR – повертає дисперсію;

GROUPING_ID – повертає рівень групування;

VARP – повертає статистичну дисперсію для заповнення всіх значень в зазначеному виразі;

MAX – повертає максимум.

У стандарті SQL визначені такі види агрегатних функцій: унарні, бінарні, інверсного розподілу, гіпотетичні функції множин.

Загальний формат унарної агрегатної функції такий:

Ім'я_функції ([ALL | DISTINCT] вираз) [FILTER (WHERE умова)]

Ключове слово DISTINCT вказує, що функція повинна розглядати тільки різні значення аргументу, а ALL – всі значення, включаючи повтори (цей варіант використовується за умовчанням). Ключове слово FILTER дає змогу додатково відібрати рядки таблиці, стовпець яких використовується як аргумент функції.

Агрегатні функції застосовуються в конструкціях SELECT і HAVING.

Розглянемо деякі з агрегатних функцій на прикладах.

Функція COUNT має два формати. У першому випадку повертається кількість рядків вхідної таблиці, у другому випадку – кількість значень аргументу у вхідній таблиці:

COUNT (*)

COUNT ([DISTINCT | ALL] вираз)

Визначимо кількість товару у таблиці Goods:

```
USE [check]
GO

SELECT COUNT(*) As 'Кількість товару' FROM goods
GO
```

Результат (рис. 2.66):

	Кількість товару
1	8

Рисунок 2.66 – Результат роботи функції COUNT

Функція SUM підраховує суму значень аргументу для всіх рядків вхідної таблиці. Аргумент повинен мати числовий тип або бути часовим проміжком.

Як аргумент може виступати ім'я стовпця або вираз над стовпцями вхідної таблиці. У цій функції також допускається використання ключових слів DISTINCT і ALL.

Визначимо ціну (загальну суму) товару у таблиці Goods:

```
USE [check]
GO

SELECT Sum(Price) As 'Сумарна ціна товару' FROM goods
WHERE Id > 10;
GO
```

Результат (рис. 2.67):

	Сумарна ціна товару
1	1000

Рисунок 2.67 – Результат роботи функції SUM

Функція AVG. Агрегатна функція AVG підраховує середнє значення аргументу для всіх рядків вхідної таблиці. Аргумент повинен мати числовий тип або бути часовим проміжком. Як аргумент може виступати ім'я стовпця або вираз над стовпцями вхідної таблиці. Допускається використовувати ключові слова DISTINCT і ALL.

Визначимо середню ціну товарів у таблиці Goods:

```
USE [check]
GO

SELECT Avg(Price) As 'Середня ціна товару' FROM goods;
GO
```

Результат (рис. 2.68):

	Середня ціна товару
1	6790.125

Рисунок 2.68 – Результат роботи функції AVG

Функції MIN і MAX дають змогу знаходити відповідно мінімальне (MIN) і максимальне (MAX) значення аргументу для всіх рядків вхідної таблиці. Хоча і в них допускається використання ключових слів DISTINCT і ALL, але вони не

впливають на результат. Аргумент цих функцій може бути будь-якого типу, для якого визначено впорядкування, тобто числового, символьного і часового.

Визначимо максимальну ціну товарів у таблиці Goods:

```
USE [check]
GO

SELECT Max(Price) As 'Максимальна ціна товару' FROM goods;
GO
```

Результат (рис. 2.69):

	Максимальна ціна товару
1	12500

Рисунок 2.69 – Результат роботи функції MAX

2.5.7 Символьні функції. В MS SQL Server є вбудовані функції для роботи із символьними рядками. Розглянемо деякі з них:

LEN – повертає кількість символів у рядку без урахування кінцевих прогалів (пробілів);

UPPER – переводить рядок у верхній регістр;

LOWER – переводить рядок у нижній регістр;

CHARINDEX – виконує пошук в рядку вказаного підрядка і повертає початкову позицію підрядка, якщо такий було знайдено, або 0, якщо рядок не містить такого підрядка;

REPLACE – Замінює всі входження в рядок вказаного підрядка іншим вказаним підрядком:

REPLACE (рядок, шуканий_підрядок, підрядок_на_який_треба_замінити)

Функції LTRIM, RTRIM і TRIM видаляють символи прогалів або інші вказані символи на початку рядка, в кінці рядка та з обох боків рядка відповідно;

Функції LEFT і RIGHT повертають вказану кількість символів з початку або з кінця рядка;

REVERSE – повертає обернений рядок, тобто усі символи початкового рядка будуть переставлені в зворотному порядку.

CONCAT – повертає один рядок, який поєднує усі рядки-аргументи. Аргументів має бути не менше двох;

STRING_AGG – повертає один рядок, який поєднує усі рядки-аргументи через рядок-роздільник (separator). Наприкінці рядка роздільник не додається. Аргументів має бути не менше двох;

STRING_SPLIT – розділяє рядок на підрядки за вказаним рядком-роздільником (separator) і повертає таблицю з одним стовпцем, рядки якого є цими підрядками.

Отримаємо назви у верхньому регістрі всіх товарів з таблиці Goods:

```
USE [check]
GO

SELECT UPPER(Name) FROM goods;
GO
```

Результат (рис. 2.70):

1	ПАЛЬТО ЖІН. СИНЄ
2	ПАЛЬТО ЖІН. ЗИМНЄ, ДЕМІСІЗОННЕ
3	ПАЛЬТО ЖІН. ЧОРНЕ
4	ПАЛЬТО УНІСЕКС. ЧОРНЕ
5	ПАЛЬТО ЧОЛ. КЛІТИНКА
6	ПАЛЬТО ЧОЛ. ЧОРНЕ
7	BALL
8	BAG

Рисунок 2.70 – Результат роботи функції UPPER

2.5.8 Числові функції. Числові (або математичні) функції виконують обчислення з використанням заданих числових аргументів, і повертають числові значення.

Стандарт SQL пропонує ряд числових функцій з очевидною семантикою. Частину функцій перераховано нижче:

ABS – повертає абсолютне значення аргументу;

DEGREES – повертає для значення кута в радіанах відповідне значення в градусах;

RAND – повертає псевдовипадкове значення типу float від 0 до 1;

EXP – повертає експоненту від першого аргументу, тобто число e , піднесене до заданого степеня, який вказано як другий аргумент;

ROUND – повертає числове значення першого аргументу, округлене до зазначеної довжини або точності, яку вказано як другий аргумент;

FLOOR – повертає найбільше ціле число, менше або рівне вказаному числовому значенню аргументу;

LOG – натуральний логарифм аргументу;

LOG10 – десятковий логарифм аргументу;

SIN – синус аргументу;

SQRT – корінь квадратний аргументу;

PI – число 3.14;

SQUARE – повертає квадрат аргументу;

POWER – повертає значення першого аргументу, піднесене до заданого степеня, який вказано як другий аргумент;

TAN – тангенс аргументу;

2.5.9 Часові функції. Ці функції використовують як аргумент типи даних дати, часу, часової позначки або часового проміжку. Тип значення, що повертається не завжди відповідає типу аргументу.

Функції дати і часу в SQL поділяються на:

Функції, які отримують значення системної дати і часу;

Функції, які отримують компоненти дати і часу;

Функції, які отримують значення дати і часу з їх компонентів;

Функції, які отримують різницю дати і часу;

Функції, які змінюють значення дати і часу;

Функції, які встановлюють або отримують формат сеансу;

Функції, що перевіряють дату й час.

Функція GETDATE повертає поточну локальну дату і час на основі системного годинника у вигляді об'єкта datetime:

```
USE [check]
GO

SELECT GETDATE() |
```

Результат (рис. 71):

```
1  2021-10-19 20:20:23.327
```

Рисунок 2.71 – Результат роботи функції GETDATE

Функція DAY повертає день з дати, яка передається як аргумент. Аналогічно функція MONTH повертає місяць дати, а функція YEAR повертає рік з дати.

Виведемо з поточної дати тільки день:

```
USE [check]
GO

SELECT Day(GETDATE() )|
```

Функція DATENAME: повертає частину цієї дати у вигляді рядка. Назва потрібної частини дати передається першим аргументом, а сама дата передається другим аргументом:

```
USE [check]
GO

|
SELECT DATENAME(month, GETDATE())
```

Функція DATEPART повертає частину цієї дати у вигляді числа. Назва потрібної частини дати передається першим аргументом (як і для DATENAME), а сама дата передається другим аргументом.

```
USE [check]
GO

SELECT DATEPART(month, GETDATE())
```

Функція DATEADD повертає дату, яка є результатом складання числа до певного компоненту дати. Першим аргументом передається компонент дати,

другим – кількість, яка додається, третім – сама дата, до якої треба додати значення другого аргументу:

```
SELECT DATEADD(month, 4, Getdate())  
SELECT DATEADD(day, 8, Getdate())  
SELECT DATEADD(day, -3, Getdate())
```

Функція DATEDIFF повертає різницю між двома датами. Першим аргументом передається компонент дати, який вказує, в яких одиницях варто вимірювати різницю. Другим і третім аргументами передаються дати, які потрібно порівняти:

```
USE [check]  
GO  
  
SELECT DATEDIFF(year, Getdate(), '2021-9-1')  
SELECT DATEDIFF(month, Getdate(), '2021-9-1')  
SELECT DATEDIFF(day, Getdate(), '2021-9-1')
```

Функція EOMONTH повертає дату останнього дня того місяця, довільну дату якого передано як аргумент:

```
USE [check]  
GO  
  
SELECT EOMONTH('2021-10-22')
```

Функція DATEFROMPARTS за роком, місяцем і дня створює дату:

```
SELECT DATEFROMPARTS(2021, 10, 22)
```

2.5.10 Запити з угрупованням рядків. Часто під час створення звітів з'являється потреба у формуванні проміжних вибірок з тих записів, які мають якісь спільні ознаки. Саме для цього призначена конструкція GROUP BY. Вона дає змогу всю множину рядків таблиці розділити на групи за ознакою рівності значень одного або декількох стовпців (і виразів над ними).

Фраза GROUP BY повинна розташовуватися після фрази WHERE (якщо вона відсутня, то після фрази FROM). Загальний синтаксис фрази GROUP BY такий:

```
SELECT список_стовпців FROM ім'я_таблиці  
[WHERE <умова_пошуку>]
```

GROUP BY вираз [, вираз ...]

За наявності фрази GROUP BY фраза SELECT застосовується до кожної групи, сформованої фразою угруповання. В цьому випадку і дія агрегатних функцій, зазначених у фразі SELECT, буде поширюватися не на всю результуючу таблицю, а тільки на рядки в межах кожної групи. Кожен вираз в списку фрази SELECT має приймати єдине значення для групи, тобто воно може бути:

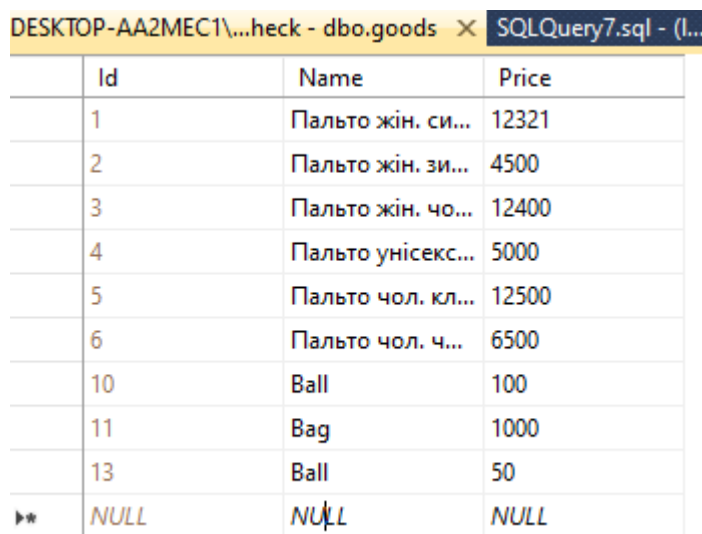
константою;

агрегатною функцією, яка оперує всіма значеннями аргументами в межах групи і поєднує їх в одне значення (наприклад, в суму);

виразом, ідентичним тому, який стоїть у фразі GROUP BY;

виразом, який об'єднує наведені вище варіанти.

Візьмемо таблицю goods бази даних check (рис. 2.72).



	Id	Name	Price
	1	Пальто жін. си...	12321
	2	Пальто жін. зи...	4500
	3	Пальто жін. чо...	12400
	4	Пальто унісекс...	5000
	5	Пальто чол. кл...	12500
	6	Пальто чол. ч...	6500
	10	Ball	100
	11	Bag	1000
	13	Ball	50
»*	NULL	NULL	NULL

Рисунок 2.72 – Таблиця goods бази даних check

Згрупуємо товари за назвою і порахуємо їх кількість:

```
USE [check]
GO

SELECT Name,
COUNT(*)
FROM goods
GROUP BY Name;
```

Результат (рис. 2.73):

	Name	
1	Bag	1
2	Ball	2
3	Пальто жін. зимнє, демісизонне	1
4	Пальто жін. синє	1
5	Пальто жін. чорне	1
6	Пальто унісекс. чорне	1
7	Пальто чол. клітинка	1
8	Пальто чол. чорне	1

Рисунок 2.73 – Результат групування товару

Групування за стовпцями. Візьмемо таблицю Person (рис. 2.74), в якій зберігається інформація про фізичних осіб: id, ім'я, електронна скринька, телефон.

	id	Name	email	telefon
►	1	Іванов Іван	ivanov@ukr.net	0671947186
	2	Петров Петро	petrov@ukr.net	0733211466
	3	Вагнер Дмитро	NULL	NULL
	4	Ільченко Ната...	NULL	NULL
	10011	Іванов І.П.	abs.gmail.com	0675144145
	10012	Петров О.О.	qwerty.gmail.c...	0506137478
	10013	Кравченко К.К.	travnik.gmail.c...	0506137478
	10014	Мартинюк С.Л.	rav.ukr.net	0735137575
	10015	Синиця О.В.	nirvana.gmail.c...	0666337338
	10016	Висоцький М.В.	ratusha.ukr.net	0962141566

Рисунок 2.74 – Таблиця Person бази даних check

І візьмемо таблицю Employee (рис. 2.75), в якій зберігається інформація про співробітників: табельний номер, id із таблиці Person, відсоток від продажу, який отримують ці співробітники.

	Tab	Personal	Interest
	1	1	5
	2	2	10
	3	10016	5
►*	NULL	NULL	NULL

Рисунок 2.75 – Таблиця Employee бази даних check

Визначимо кількість співробітників, які отримують певний відсоток:

```
SELECT count (*) AS "К-сть працівників"
FROM Person p, Employee e
WHERE e.Personal = p.id
GROUP BY e.Interest;
```

Результат (рис. 2.76):

	К-сть працівників
1	2
2	1

Рисунок 2.76 – Результат групування товару

Фраза WHERE перевіряє на відповідність умові кожного рядка вихідних таблиць. Для відбору рядків серед отриманих груп слід застосовувати фразу HAVING. Вона відіграє таку ж роль для груп, що і фраза WHERE для вихідних таблиць, і може використовуватися лише за наявності фрази GROUP BY. У реченні SELECT фрази WHERE, GROUP BY і HAVING обробляються в такому порядку:

фразою WHERE відбираються рядки, які задовольняють вказаній в ній умові;

фраза GROUP BY групує відібрані рядки;

фразою HAVING відбираються групи, що задовольняють вказаній в ній умові.

Виконаємо попередній приклад, але рахувати будемо тільки тих співробітників, у яких відсоток від продажу більше ніж 5.

```
USE [check]
GO
```

```
SELECT count (*) AS "К-сть працівників"
FROM Person p, Employee e
WHERE e.Personal = p.id
GROUP BY e.Interest
having e.Interest > 5;
```

Результат (рис. 2.77):

	К-сть працівників
1	1

Рисунок 2.77 – Результат групування товару з умовою

Питання для самоперевірки

Які рівні захисту SQL Server Ви знаєте?

Що таке обліковий запис SQL Server? Його призначення.

Що таке користувач в SQL Server? Його призначення.

Що таке роль SQL Server? Її призначення.

Як надати права для ролі?

Як надати права для користувача?

Як змінити права облікового запису?

Як видалити обліковий запис?

Як змінити права користувача?

Як видалити користувача?

Як змінити роль? Як видалити роль?

У чому полягає відмінність між користувачами на рівні сервера і на рівні БД?

У чому полягає відмінність між ролями на рівні сервера і на рівні БД?

Які категорії можна виділити вбудованих функцій SQL?

Назвіть приклади агрегатних функцій.

Назвіть функції для роботи з символьною інформацією.

Назвіть функції для роботи з датами.

Що таке угруповання? Назвіть етапи угруповання?

Назвіть особливості застосування оператора HAVING.

Завдання

1. Створити обліковий запис для свого сервера.
2. Створити користувача для цього облікового запису.
3. Створити два користувача для своєї БД.
4. Створити дві ролі: одна дає змогу лише переглядати БД, друга – повну маніпуляцію з даними в БД.

Всі запити виконати в Azure Data Studio:

Варіант	Опис завдання
1	Створити таблицю дисципліни з атрибутами: Назва дисципліни, викладач, кафедра. Зв'язати її з іншими таблицями. Заповнити її 5 записами. Створити таблицю успішність з полями: Дисципліна, Студент, оцінка. Вивести всі факультети, кафедри та ПІБ студентів, успішність яких перевищує 90 балів Розрахувати середню оцінку студентів. Вивести оцінки по всім дисциплінам всіх студентів. Вивести всіх студентів, які за певним предметом, отримали певну оцінку. Вивести всіх студентів максимальна оцінка яких не більша ніж 60 балів.
2	1.Вивести ПІБ та посади всіх співробітників супермаркету, ставка яких перевищує 15000 грн. Сортування в алфавітному порядку за ПІБ. 2.Вивести ПІБ співробітників, які зробили продаж певної дати. 3. Вивести всю інформацію про товари, які були продані певної дати. 4. Вивести інформацію про співробітника, який зробив продаж на максимальну суму за останній місяць. 5. Вивести кількість посад, сума окладу яких більша ніж 5000. 6.Згрупувати всі товари за виробником. 7.Вивести всі товари певного виробника.
3	1.Вивести інформацію про вид військ, якій мав локацію на певній території. 2.Вивести інформацію про військовослужбовців і їх частину.

	3.Вивести інформацію про військовослужбовців, їх вид військ, номер частини і дислокацію. 4.Знайти найменшу площу дислокації частини. 5.Обрахувати кількість частин, площа дислокації яких більша ніж 10 000 м². 6.Згрупувати весь особовий склад за ротами. 7.Вивести особовий склад певної роти.
4	1.Вивести всю інформацію про бібліотеки, співробітники яких мають телефон, що використовує зв'язок Lifecell. 2. Вивести інформацію про співробітників певної бібліотеки, що поповнювали фонд певної дати. 3. Вивести інформацію про поповнення всіх бібліотек певної дати. 4. Знайти середню кількість книжок, журналів тощо для всіх фондів. 5. Порахувати кількість літератури за певним типом. 6. Згрупувати співробітників за бібліотеками. 7. Вивести інформацію про фонд певної бібліотеки.
5	1.Вивести всі пансіонати у Києві, в яких є спа-салон. 2. Вивести всю інформацію про пансіонат і тур, де перебував певний клієнт. 3. Вивести інформацію про клієнтів, вік яких більше ніж 20 років. 4. Знайти різницю між датами заїзду і від'їзду клієнтів певного пансіонату. 5. Вивести кількість пансіонатів, які знаходяться в певному місті. 6. Вивести інформацію про пансіонати, згруповані на стовпчиком Місто.

7. Вивести всі пансіонати, які знаходяться в певному місті.

2.6 Програмування та розширені об'єкти бази даних

2.6.1 Арифметичні і логічні операції. Операнди, які застосовуються в арифметичних і логічних операціях в SQL, подано в таблиці 2.9.

Таблиця 2.9. Операнди SQL

нак	Призначення	нак	Призначення
*	Знак множення	“	В них заключають строкові значення,
-	Знак віднімання	‘	В них заключають строкові значення,
%	Залишок від ділення двох чисел	<	Не дорівнює
+	Знак додавання або конкатенації (об'єднання двох рядків)	[]	Аналог лапок, в них можна розміщувати назви ідентифікаторів, якщо в їх назвах є прогалини (пробіли)
=	Знак рівності	!	Не менше, ніж
<	Менше або дорівнює	<	Не більше, ніж
=	Більше або дорівнює	>	Більше
>	Більше або дорівнює	<	Більше
=	Не дорівнює	>	Менше
()	Ставиться перед іменем змінної	.	Розділяє батьківські і підлеглі об'єкти
()	Вказує на системні	/	Знак ділення

нак	3	Призначення	нак	3	Призначення
@		функції			
-		Однорядковий коментар або коментар з поточної позиції і до кінця рядка	*	/	Багаторядковий коментар

2.6.2 Ідентифікатори та змінні. Ідентифікатори – це імена об’єктів, на які можна посилатися в програмі. Найменування об’єктів відбувається за такими правилами:

перший символ складається з букв англійського алфавіту або «#», «@»;

решта додатково з цифр і «\$»;

ім’я ідентифікатора не повинно збігатися з зарезервованим словом.

Для збереження розрахованих проміжних значень використовуються змінні, ім’я яких повинно починатися з @.

Синтаксис в позначеннях MS SQL Server:

DECLARE @ ім’я_змінної1 тип_змінної (розмір),, @

Ім’я_змінноїN тип_змінної (розмір)

Присвоєння значень змінним відбувається за допомогою оператора SET, синтаксис:

SET @ ім’я_змінної = значення

```

DECLARE @a INT, @b numeric (10,2)
SET @a = 20
SET @b = (@a + @a) / 15
SELECT @b --вивід на екран результату

```

Результат (рис. 2.78):

1	2.00
---	------

Рисунок 2.78 – Результат виведення

Якщо змінній присвоюємо значення результату запиту, то використовуємо оператор SELECT.

Присвоїмо змінній *a* кількість рядків в таблиці *goods*:

```
use [check]
DECLARE @a INT
SELECT @a = COUNT (*) FROM goods
SELECT @a
```

Результат (рис. 2.79):

1	10
---	----

Рисунок 2.79 – Результат виведення

Те саме можна отримати іншим способом:

```
use [check]
DECLARE @a INT
SET @a = (SELECT COUNT (*) FROM goods)
SELECT @a
```

Результат (рис. 2.80):

1	10
---	----

Рисунок 2.80 – Результат виведення

Присвоїмо змінній *str* значення стовпчика *Name* з таблиці *goods*:

```
use [check]
DECLARE @str CHAR (30)
SELECT @str = Name FROM goods
SELECT @str
```

Результат (рис. 2.81):

1	Galaxy S7
---	-----------

Рисунок 2.81 – Результат виведення

У великих базах даних запити часто включають об'єднання (JOIN) багатьох таблиць, складні умови фільтрації або агрегатні функції. Віртуальні або тимчасові таблиці дозволяють інкапсулювати цю складну логіку в єдиний, простий для використання об'єкт. Це значно підвищує читабельність та зручність коду. Тимчасові таблиці є ефективним інструментом для контролю доступу до даних. Адміністратор може надати користувачам доступ лише до

певного подання таблиць бази даних, яке показує лише дозволені стовпці або рядки, приховуючи конфіденційну інформацію в базових таблицях. Наприклад, подання може показувати список співробітників, але приховувати їхні зарплати. Створення тимчасової таблиці відбувається через змінну типу TABLE. Змінна для тимчасової таблиці оголошується за допомогою оператора DECLARE із зазначенням в дужках стовпців таблиці, їх типів, розмірів, значень за замовчуванням, а також індексів типу PRIMARY KEY або UNIQUE.

Створимо таблицю і додаємо в неї два записи:

```
use [check]
DECLARE @mytable TABLE (id INT, myname CHAR (20) DEFAULT 'Іванов Іван')
INSERT INTO @mytable (id) VALUES (1)
INSERT INTO @mytable (id, myname) VALUES (2, 'Ігор Троцький')
SELECT * FROM @mytable
```

Результат (рис. 2.82):

	id	myname
1	1	Іванов Іван
2	2	Ігор Троцький

Рисунок 2.82 – Результат виведення

Створення тимчасової таблиці з двома полями, в яку буде додано рядки, як результат виконання запиту вибірки даних з таблиці goods:

```
DECLARE @mytable TABLE (id INT, myname CHAR (255) DEFAULT 'Введіть назву')
INSERT @mytable SELECT id, Name FROM goods
SELECT * FROM @mytable
```

Результат (рис. 2.83):

	id	myname
1	1	Пальто жін. синє
2	2	Пальто жін. зимнє, демісизонне
3	3	Пальто жін. чорне
4	4	Пальто унісекс. чорне
5	5	Пальто чол. клітинка
6	6	Пальто чол. чорне
7	10	Ball
8	11	Bag
9	13	Ball
10	1013	Galaxy S7

Рисунок 2.83 – Результат виведення

2.6.3 Операторські дужки. Для конструкцій умовних і циклічних розгалужень використовують операторські дужки. Операторські дужки повинні містити хоча б один оператор. Операторські дужки мають такий вид:

BEGIN -- це відкриття операторської дужки

оператор1

оператор2

...

операторN

END -- це закриття операторської дужки

2.6.4 Умовна конструкція IF. Синтаксис умовної конструкції в SQL має вигляд:

IF умова

набір_операторів_1 ELSE

набір_операторів_2

Використаємо операторські дужки і умовні конструкції для пошуку відповіді на питання: «кількість товарів більше 10, так чи ні?». Для цього напишемо і виконаємо такий запит:

```
DECLARE @a INT
DECLARE @str CHAR (30)
SET @a = (SELECT COUNT (*) FROM goods)
IF @a > 10 BEGIN
    SET @str = 'Кількість товарів більше 10'
    SELECT @str
END ELSE
BEGIN
    SET @str = 'Кількість товарів =' + str (@a)
    SELECT @str
END
```

Результат (рис. 2.84):

1	Кількість товарів = 10
---	------------------------

Рисунок 2.84 – Результат виведення

2.6.5 Цикл WHILE. Синтаксис циклу в SQL має вигляд:

WHILE умова

набір_операторів_1

BREAK

набір_операторів_2

CONTINUE

Конструкції **BREAK** і **CONTINUE** є необов'язковою. Цикл можна примусово зупинити, якщо в його тілі виконати команду **BREAK**. Якщо ж потрібно почати цикл заново, не чекаючи виконання всіх команд в тілі, необхідно виконати команду **CONTINUE**.

Оголосимо змінну *a*. Змінна *a* в циклі збільшує своє значення у випадковому порядку. Перевіримо, щоб значення *a* не перевищувало 100. Також напишемо умову примусового виходу з циклу, якщо значення змінної *a* потрапляє в діапазон від 40 до 50.

```
DECLARE @a INT
SET @a = 1
WHILE @a < 100
BEGIN
    PRINT @a --| вивід на екран значення змінної
    IF (@a > 40) AND (@a < 50)
        BREAK --вихід і виконання 1-ї команди за циклом
    ELSE
        SET @a = @a + rand() * 10
    CONTINUE
END
PRINT @a
```

Результат (рис. 2.85):

```
1
2
10
18
18
22
24
26
33
35
41
41
```

Рисунок 2.85 – Результат виведення

З результату видно, що спрацювала умова примусового виходу з циклу, як значення змінної *a* стало рівним 41.

2.6.7 Функції і збережені процедури. Функції і збережені процедури використовуються в SQL Server для реалізації складних поширених алгоритмів обробки даних або різних адміністративних дій створення облікових записів,

отримання інформації про об'єкти бази даних, управління властивостями сервера і баз даних, управління підсистемою реплікації і автоматизації тощо.

Функції і процедури можуть бути викликані клієнтськими програмами, іншими функціями або збереженими процедурами, а також тригерами.

У будь-якому випадку необхідно вказати ім'я функції або процедури, що зберігається, і список аргументів. Аргументи зіставляються параметрам відповідної функції або процедури, яка зберігається. Водночас типи аргументів і параметрів повинні збігатися або допускати автоматичні перетворення типів. Якщо для деякого параметра встановлено значення за замовчуванням і це значення підходить для даного виклику, то відповідний аргумент може бути опущений.

Створення, зміна та видалення функцій і процедур проходить за допомогою таких команд:

```
CREATE FUNCTION,  
CREATE PROCEDURE,  
ALTER FUNCTION,  
ALTER PROCEDURE,  
DROP FUNCTION,  
DROP PROCEDURE.
```

При створенні функції вказується тип значення, а в тілі функції обов'язково задається команда RETURN, за якою слідує вираз для обчислення значення, яке повертається.

У тілі процедури використання команди RETURN (зазвичай, без подальшого виразу) не обов'язково. Коли цієї команди немає, вихід з процедури відбуватиметься після виконання останньої команди процедури.

Тіло, як функції, так і збереженої процедури, починається ключовим словом AS. Так як кожна з них зберігається як окремий об'єкт, то для того, щоб вказати кінець тіла, не потрібно записувати якесь спеціальне ключове слово або знак.

За командами створення функції або процедури перераховуються імена параметрів, що починаються з символу @, і їх типи, а також значення за замовчуванням.

Для *функції* цей список записується в круглих дужках, після яких записується ключове слово RETURNS (повернення) і тип значення, що повертається.

Для *збереженої процедури*, круглі дужки не використовуються, і задавати тип значення, яке повертається, не потрібно.

Для тіла функції часто використовують ключове слово begin після ключового слова as і ключове слово end в кінці тіла.

Додаткові опції функції або процедури, які задаються ключовим словом with до початку тіла.

У SQL Server можна створювати функції трьох класів:

Scalar – повертають звичайне скалярне значення;

Inline – повертають таблицю динамічної структури, яка створюється єдиною командою тіла функції SELECT;

Multi-statement – повертає звичайну таблицю заданої структури, водночас кількість команд в тілі функції не обмежується.

Створення функцій відбувається за такою командою:

CREATE FUNCTION (***),

яка створює визначену користувачем функцію в SQL Server. Обумовлена користувачем функція є підпрограмою SQL або середовища CLR, яка приймає параметри, виконує дії, такі як складні обчислення, а потім повертає результат цих дій у вигляді значення. Значення, яке повертається, може бути скалярним значенням або таблицею. За допомогою цієї інструкції можна створити підпрограму, яку можна повторно використовувати такими способами:

в інструкціях Transact-SQL, наприклад SELECT;

у застосунках, які викликають функцію;

у визначенні іншої користувацької функції;

для параметризації уявлення або поліпшення функціональності індексованого подання;

для визначення стовпця таблиці;

для визначення обмеження CHECK на стовпець;

для заміни збереженої процедури.

Синтаксис створення скалярної функції має такий вид:

```
--Transact-SQL Scalar Function Syntax
CREATE FUNCTION [ schema_name. ] function_name
( [ [ @parameter_name [ AS ] [ type_schema_name. ] parameter_data_type
    [ = default ] [ READONLY ] ]
  [ ,...n ]
]
)
RETURNS return_data_type
    [ WITH <function_option> [ ,...n ] ]
    [ AS ]
BEGIN
    function_body
    RETURN scalar_expression
END
[ ; ]
```

Аргументи:

TABLE – вказує, що значенням функції, яке повертається, є таблиця. Функціям з табличним значенням можуть передаватися тільки константи і @local_variables.

У вбудованих функціях з табличним значенням значення TABLE, яке повертається, визначається за допомогою єдиної інструкції SELECT. Вбудовані функції не мають відповідних змінних, які повертаються.

select_stmt – Одиночна інструкція SELECT, що визначає значення, яке повертається вбудованою функцією з табличним значенням.

У функціях припустимі такі інструкції:

Інструкції присвоювання.

Інструкції управління потоком, за винятком інструкцій TRY ... CATCH.

Інструкції DECLARE, які оголошують локальні змінні і локальні курсори.

Інструкції SELECT, які містять списки вибору з виразами, які привласнюють значення локальних змінних.

Операції над локальними курсорами, які оголошуються, відкриваються, закриваються і звільняються в тілі функції. Допустимі тільки ті інструкції FETCH, які пропозицією INTO привласнюють значення локальним змінним. Інструкції FETCH, які повертають дані клієнта, неприпустимі.

Інструкції INSERT, UPDATE і DELETE, які змінюють локальні табличні змінні.

Інструкції EXECUTE, які викликають розширені процедури.

Обмеження:

Визначені користувачем функції не можуть виконувати дії, які змінюють стан бази даних.

Визначені користувачем функції не можуть містити конструкцію OUTPUT INTO, ціллю якої є таблиця.

Визначені користувачем функції можуть бути вкладеними, тобто з однієї функції може бути викликана інша. Вкладеність визначених користувачем функцій не може перевищувати 32 рівнів.

У наступному прикладі показано створення скалярної визначеної користувачем функції ISOweek, яка отримує в якості аргументу дату і обчислює номер місяця, а потім за номером визначає назву місяця. Для правильної роботи цієї функції перед її викликом повинна бути виконана інструкція SET DATEFIRST 1. У цьому прикладі також показано використання пропозиції EXECUTE AS для вказання контексту безпеки, в якому може бути виконана збережена процедура. Параметр CALLER вказує, що процедура буде виконана в контексті користувача, який її викликає. Перш ніж функцію буде створено, виконується перевірка на наявність в базі даних функції з таким самим ім'ям. Якщо функція з таким ім'ям вже була створена раніше, то вона буде видалена за допомогою команди DROP FUNCTION і створена заново CREATE FUNCTION.

Запит:

```

USE [check];
GO
IF OBJECT_ID (N'dbo.ISOweek', N'FN') IS NOT NULL
    DROP FUNCTION dbo.ISOweek;
GO
CREATE FUNCTION dbo.ISOweek (@DATE date)
    RETURNS CHAR(15)
    WITH EXECUTE AS CALLER
    AS
    BEGIN
        DECLARE @man int;
        DECLARE @ISOweek char(15);
        SET @man= MONTH(@DATE)
        IF (@man=1) SET @ISOweek='Січень';
        IF (@man=2) SET @ISOweek='Лютий';
        IF (@man=3) SET @ISOweek='Жовтень';
        IF (@man=4) SET @ISOweek='Квітень';
        IF (@man=5) SET @ISOweek='Травень';
        IF (@man=6) SET @ISOweek='Червень';
        IF (@man=7) SET @ISOweek='Липень';
        IF (@man=8) SET @ISOweek='Серпень';
        IF (@man=9) SET @ISOweek='Вересень';
        IF (@man=10) SET @ISOweek='Жовтень';
        IF (@man=11) SET @ISOweek='Листопад';
        IF (@man=12) SET @ISOweek='Грудень';
        RETURN(@ISOweek);
    END
GO
SET DATEFIRST 1;
SELECT dbo.ISOweek('12.04.2004') AS 'Місяць';

```

Результат (рис. 2.86):

	Місяць
1	Квітень

Рисунок 2.86 – Результат виведення

Створимо функцію з табличним значенням, яка буде повертати назви та ціни товарів, ціна яких більша за вказану. Застосуємо цю функцію, щоб з таблиці товарів отримати усі товари, ціна яких більша за 5000:

```

USE [check];
GO

IF OBJECT_ID (N'dbo.PriceByStore', N'FN') IS NOT NULL
DROP FUNCTION dbo.PriceByStore;
GO

CREATE FUNCTION PriceByStore(@storeid int)
RETURNS TABLE
AS
RETURN
(
SELECT Name AS "Good", Price From goods
WHERE Price>@storeid
);
GO

SET DATEFIRST 1;
SELECT * FROM dbo.PriceByStore(5000) ;

```

Результат (рис. 2.87):

	Good	Price
1	Пальто жін. чорне	12400
2	Пальто чол. клітинка	12500
3	Пальто чол. чорне	6500

Рисунок 2.87 – Результат виведення

Збережені процедури – це набір команд, які складаються з одного або декількох операторів SQL або функцій і зберігаються в базі даних у відкомпільованому вигляді.

Типи збережених процедур бувають такі:

системні збережені процедури призначені для виконання різних адміністративних дій. Практично всі дії з адміністрування сервера виконуються з їх допомогою;

користувацькі збережені процедури реалізують ті чи інші дії. Збережені процедури – повноцінний об’єкт бази даних. Внаслідок цього кожна збережена процедура розташовується в конкретній базі даних, де і виконується;

тимчасові збережені процедури існують лише деякий час, після чого автоматично знищуються сервером. Вони діляться на локальні і глобальні.

Синтаксис оператора створення нової або зміни існуючої збереженої процедури в MS SQL Server:

```
--Transact-SQL Stored Procedure Syntax
CREATE { PROC | PROCEDURE } [schema_name.] procedure_name [ ; number ]
    [ { @parameter [ type_schema_name. ] data_type }
      [ VARYING ] [ = default ] [ OUT | OUTPUT | [READONLY]
    ] [ ,...n ]
    [ WITH <procedure_option> [ ,...n ] ]
    [ FOR REPLICATION ]
    AS { [ BEGIN ] sql_statement [;] [ ...n ] [ END ] }
[;]

<procedure_option> ::=
    [ ENCRYPTION ]
    [ RECOMPILE ]
    [ EXECUTE AS Clause ]
```

Для виконання збереженої процедури використовується команда:

```
[[EXEC [UTE] ім'я_процедури [; номер]
[[@ ім'я_параметра =] {значення | @ Ім'я_змінної}
[OUTPUT] | [DEFAULT]] [, ... n]
```

Приклад процедури без параметрів:

```
USE [check];
GO

CREATE PROCEDURE Count_Goods
AS
Select count (Name)
from goods
where Price > 5000
Go

GO
EXECUTE Count_Goods
```

Результат (рис. 2.88):

1	3
---	---

Рисунок 2.88 – Результат виведення

Приклад процедури з параметром:

```

USE [check];
GO

CREATE PROCEDURE Count_Goods1
@Sum_price as Int
AS
Select count (Name)
from goods
where Price > @Sum_price
Go

GO

EXECUTE Count_Goods1 10000 |

```

Результат (рис. 2.89):

	(Отсутствует имя столбца)
1	2

Рисунок 2.89 – Результат виведення

2.6.8 Подання. Подання (View) для користувачів баз даних виглядає як таблиця, проте воно не містить даних, а лише надає дані, розташовані в одній або декількох таблицях. Таким чином, подання – це віртуальні таблиці, які визначаються запитом на мові Transact-SQL. Подібно реальним таблицям, подання містять іменовані стовпці і рядки з даними, які динамічно вибираються з таблиць і надаються користувачу для перегляду. Подання часто застосовуються для обмеження доступу до конфіденційних даних в таблицях баз даних.

Коли в подання не включається стовпець вихідної таблиці, то вважають, що на таблицю накладений вертикальний фільтр. Якщо в SQL-запиті встановлено одну або декілька умов для вибірки рядків, то вважають, що на таблицю накладений горизонтальний фільтр.

Подання може вибирати дані з іншого подання, які, в свою чергу, можуть також ґрунтуватися на поданнях або таблицях. Вкладеність подання (глибина) не повинна перевищувати 32 рівень. Подання можна створювати, використовуючи бази даних одного сервера (поточного). Максимальна кількість стовпців в одному поданні 1024. Подання не можуть посилатися на тимчасові таблиці. Крім того, не можна створювати тимчасове подання.

Для подання не можна визначати обмеження цілісності, тригери, правила, або замовчування, а також створювати звичайний або повнотекстовий індекс. В основному подання використовуються для вибірки даних.

Однак за допомогою подання можна змінювати дані в таблицях, на основі яких побудовано ці подання, при цьому необхідно дотримуватися ряду правил: подання має містити, як мінімум, одну таблицю в параметрі FROM команди SELECT, не дозволяється використання функцій агрегування та інших.

Як і для таблиць, для подання можна визначити такі права доступу:

SELECT – перегляд даних;

INSERT – додавання даних через подання;

UPDATE – зміна даних у вихідних таблицях;

DELETE – видалення даних у вихідних таблицях.

Щоб мати можливість створювати подання, треба мати права власника баз даних і мати відповідні дозволи для будь-яких таблиць або подання, згаданих «в запиті на створення цього подання».

Для створення подання використовується така команда Transact-SQL:

```
CREATE VIEW [Ім'я_бази_даних] [Ім'я_власника]
```

```
Ім'я_подання
```

```
[(Ім'я_колонки [... n])]
```

```
[WITH{ENCRYPTION|SCHEMABINDING|
```

```
VIEW_METADATA}]
```

```
AS Команда SELECT
```

```
[WITH CHECK OPTION]
```

Якщо в команді не задані імена колонок подання, то вони визначаються по іменах вибраних колонок в команді SELECT. Параметр ENCRYPTION приховує код створення цього подання, а параметр SCHEMABINDING забезпечує контроль структури вихідних об'єктів, до яких звертається оператор SELECT. Опція WITH CHECK OPTION не дає змогу змінювати рядки таким чином, щоб вони зникли під час вибирання командою SELECT.

Для того, щоб створити подання засобами MS SQL Server Management Studio потрібно у провіднику об'єктів розкрити вузол бази даних, натиснути на його підвузол Подання (View) правою кнопкою миші і в контексті меню вибрати пункт «Створити подання» (рис. 2.90).

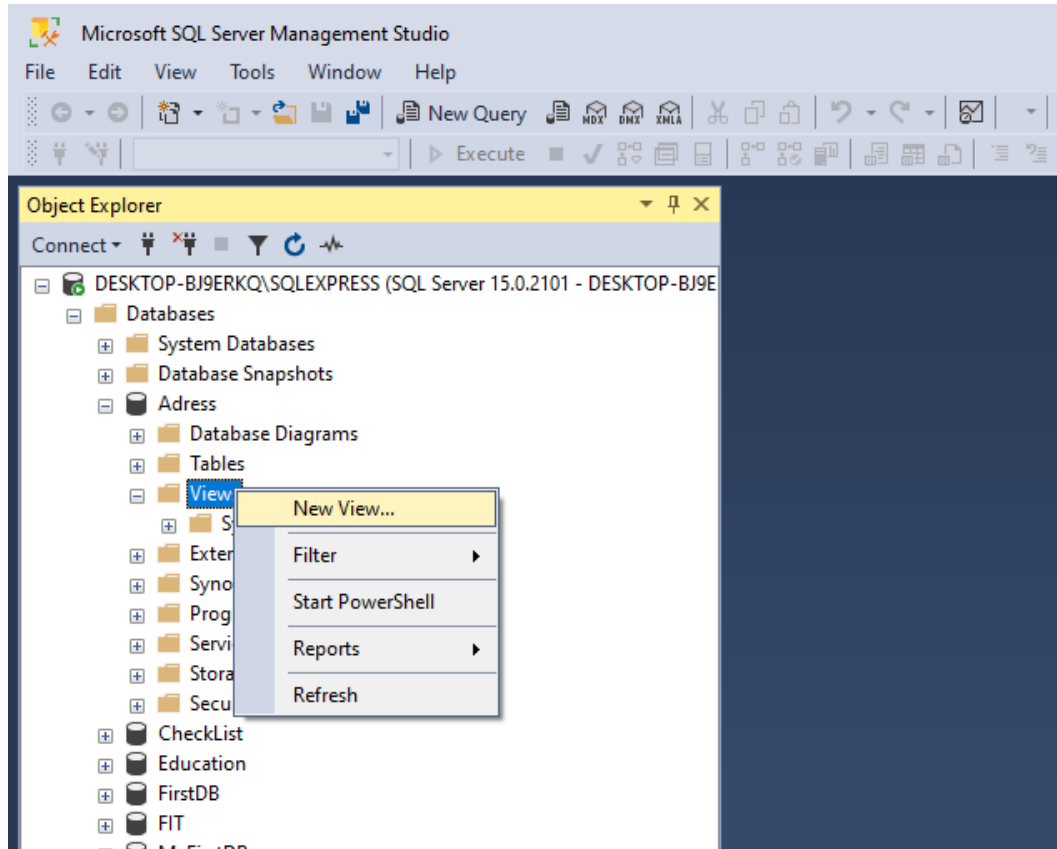


Рисунок 2.90 – Пункт контекстного меню «Створення подання» в SQL Server Management Studio

Відкриється вікно додавання таблиці, в якому виберемо зі списку таблиць ті, які потрібні для створення подання – Employee і Person (рис. 2.91).

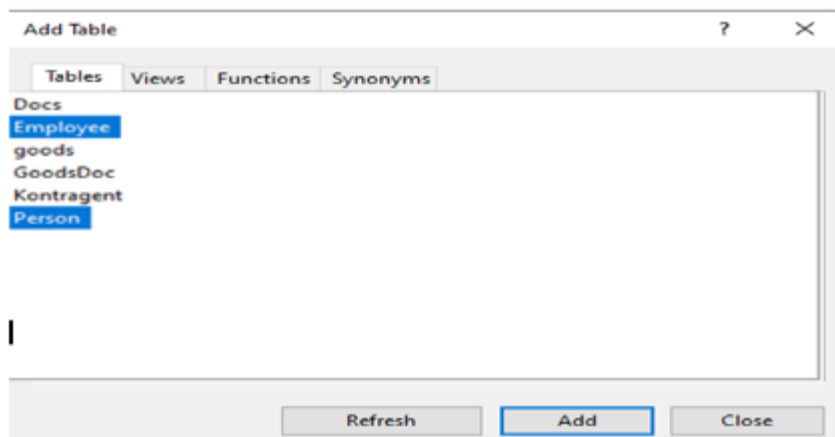


Рисунок 2.91 – Вибір таблиць БД для створення подання

Відкриється вікно, в якому виберемо ті стовпці таблиця, які потрібні для створення подання. Виберемо в таблиці Employee стовпці Tab та Interest, а в таблиці Person – стовпці Name, email і telefon (рис. 2.92). Задамо псевдонім Expr1 стовпцю Tab.

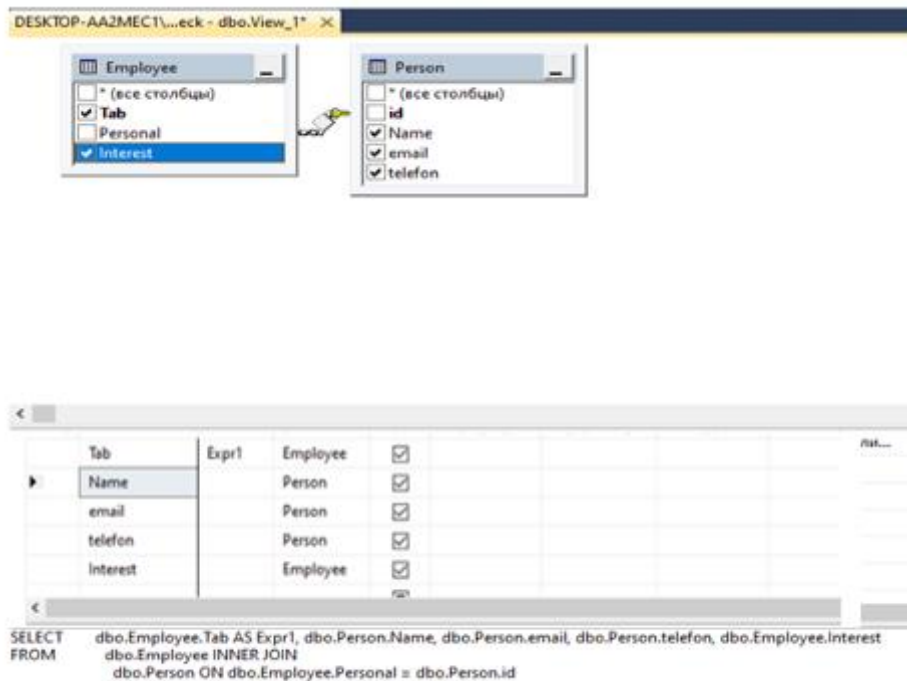


Рисунок 2.92 – Вибір стовпців таблиць для створення подання
Зберігаємо подання (Файл→Зберегти) з іменем View_1 (рис. 2.93).

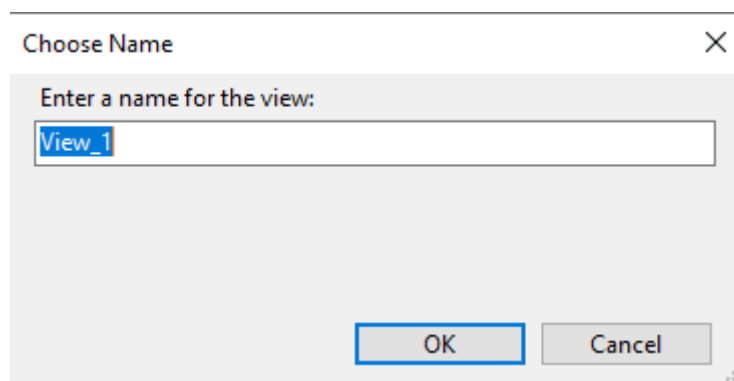


Рисунок 2.93 – Збереження подання

В результаті у теці Подання в провіднику об'єктів SSMS з'являється наше подання (рис. 2.94).

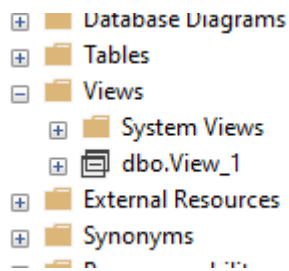


Рисунок 2.94 – Створене подання View_1 в провіднику об'єктів SSMS

Для перегляду створеного подання натисніть на ньому правою клав'яшею миші та виберіть в контекстному меню пункт «Вибрати перші 1000 рядків» або «Змінити перші 200 рядків». Подання має табличний вигляд та містить усі вибрані стовпці з обох таблиць Employee і Person. Зверніть увагу, що ім'я першого стовпця – заданий псевдонім Expr1 (рис. 2.95).

	Expr1	Name	email	telefon	Interest
1	1	Іванов Іван	ivanov@ukr.net	0671947186	5
2	2	Петров Петро	petrov@ukr.net	0733211466	10
3	3	Висоцький М.В.	ratusha.ukr.net	0962141566	5

Рисунок 2.95 – Перегляд подання View_1

Для того щоб створити подання SQL-запитом, створимо і виконаємо такий скрипт:

```
CREATE VIEW EmployView AS
SELECT Employee.Tab AS Tab,
       Person.Name AS Persona,
       Person.telefon As Telefon
FROM Employee INNER JOIN Person ON Employee.Tab = Person.id;
```

В результаті у теці Подання в провіднику об'єктів SSMS з'являється нове подання – EmployView (рис. 2.96).

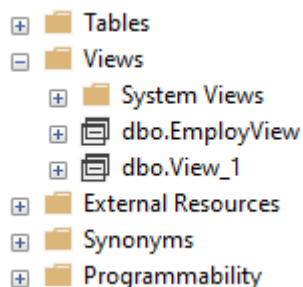


Рисунок 2.96 – Створене подання EmployView в провіднику об'єктів SSMS

Використаємо створене подання EmployView для отримання даних:

```
USE [check]
SELECT * FROM EmployView;
```

Результат (рис. 2.97):

	Tab	Persona	Telefon
1	1	Іванов Іван	0671947186
2	2	Петров Петро	0733211466
3	3	Вагнер Дмитро	NULL

Рисунок 2.97 – Результат виведення

Для зміни подання використовується команда ALTER VIEW. Ця команда має практично той же синтаксис, що і CREATE VIEW:

```
ALTER VIEW EmployView AS
SELECT Employee.Tab AS Tab,
       Person.Name AS Persona,
       Person.telefon As Telefon,
       Person.email AS Email
FROM Employee INNER JOIN Person ON Employee.Tab = Person.id;
```

В результаті виконання цього запиту, отримуємо змінене подання. Переглядаємо його (рис. 1.98):

	Tab	Persona	Telefon	Email
1	1	Іванов Іван	0671947186	ivanov@ukr.net
2	2	Петров Петро	0733211466	petrov@ukr.net
3	3	Вагнер Дмитро	NULL	NULL

Рисунок 2.98 – Результат виведення

Для видалення подання викликається команда DROP VIEW:

```
DROP VIEW View_1;
```

2.6.9 Створення, програмування і управління тригерами. Тригер – це спеціальний тип збережених процедур, які запускаються сервером автоматично при виконанні тих чи інших дій з даними таблиці. Тригери поділяють по типу команд, на які вони реагують: INSERT TRIGGER – запускаються у разі спроби вставити дані за допомогою команди INSERT; UPDATE TRIGGER – запускаються у разі спроби змінити дані за допомогою команди UPDATE; AUTHSMALL TRIGGER – запускаються у разі спроби видалити дані за допомогою команди DELETE.

Формат створення тригера:

```

CREATE TRIGGER Ім'я тригера
ON {Ім'я таблиці \ Ім'я подання}
[WITH ENCRYPTION] -- шифрування коду тригера;
{{{FOR \ AFTER \ INSTEAD OF}}
{[DELETE] \ [INSERT] \ [UPDATE]}}
[WITE APPEND] -- тільки для версій 6.5 і нижче;
[NOT FOR REPLICATION] -- не для реплікації;
AS sql_statement [... n] -- тіло тригера;
}

| { {FOR \ AFTER \ INSTEAD OF}
{[INSERT] \ [UPDATE]}}
[WITE APPEND] -- тільки для версій 6.5 і нижче;
[NOT FOR REPLICATION] -- НЕ для реплікації;
AS {IF UPDATE (column) -- при зміні стовпчика;
{{AND \ OR} UPDATE (column) [... n]}} -- теж;
|
IF (COLUMNS_UPDATED() {bitwise_operator}
Update_bitmask)
{Comparison_operator} column_bitmask [... n]
}
sql_statement [... n] -- тіло
}
|
| { {FOR \ AFTER \ INSTEAD OF}
{[INSERT] \ [UPDATE]}}
[WITE APPEND] -- тільки для версій 6.5 і нижче;
[NOT FOR REPLICATION] -- НЕ для реплікації;
AS {IF UPDATE (column) -- при зміні стовпчика;
{{AND \ OR} UPDATE (column) [... n]}} -- теж;
IF (COLUMNS_UPDATED() {bitwise_operator}
Update_bitmask)
{Comparison_operator} column_bitmask [... n]
}
sql_statement [... n] -- тіло тригера.
}

```

Параметри FOR, AFTER і INSTEAD OF, які вказуються під час створення тригера, визначають його поведінку таким чином:

FOR – запуск тригера під час виконання заданого списку команд;

AFTER – запуск тригера після успішного виконання команд списку;

INSTEAD OF – тригер викликається замість виконання команд списку.

Можна визначити кілька AFTER-тригерів для кожної операції INSERT, UPDATE і DELETE. За замовчуванням всі тригери є AFTER-тригерами.

Тригери можна створювати для тимчасових або системних таблиць. Команда створення тригера повинна бути першою в пакеті і застосовуватися тільки до однієї таблиці.

Створимо тригер, який у разі спроби додавання або редагування товарів у таблицю goods, збільшував ціну товару на 20%.

Запишемо і виконаємо такий запит:

```
USE [check];
GO
CREATE TRIGGER Goods_INSERT_UPDATE
ON goods
AFTER INSERT, UPDATE
AS
UPDATE goods
SET Price = Price + Price * 0.20
WHERE Id = (SELECT Id FROM inserted)
Go

USE [check];
GO

INSERT INTO goods
VALUES ('plate', 50)
```

Результат (рис. 2.99):

	Id	Name	Price
►	1	Пальто жін. си...	3000
	2	Пальто жін. зи...	4500
	3	Пальто жін. чо...	12400
	4	Пальто унісекс...	5000
	5	Пальто чол. кл...	12500
	6	Пальто чол. ч...	6500
	10	Ball	100
	11	Bag	1000
	13	Ball	50
	1013	Galaxy S7	5000
	2013	plate	60
*	NULL	NULL	NULL

Рисунок 2.99 – Результат роботи тригера

Зверніть увагу на нижній рядок результату – ціна товару «plate» записано 60, а не 50 як у запиті, оскільки спрацював тригер.

Для видалення тригера використовується команда `DROP TRIGGER`. Буває, що потрібно лише призупинити дію тригера, а не видаляти. У цьому випадку його можна тимчасово вимкнути за допомогою команди `DISABLE TRIGGER`. А коли тригер знадобиться, його можна увімкнути за допомогою команди `ENABLE TRIGGER`.

У разі додавання даних (під час команди `INSERT`) в тригері можливо отримати додані дані з віртуальної таблиці `INSERTED`. У разі видалення всі видалені дані розміщуються у віртуальній таблиці `DELETED`. Тригер оновлення даних спрацьовує під час операції `UPDATE`. І в такому тригері можливо використати дві віртуальні таблиці. Таблиця `INSERTED` зберігає значення рядків після оновлення, а таблиця `DELETED` зберігає самі рядки, але до оновлення.

2.6.10 Транзакції. Мова запитів Transact-SQL взяла свою назву від слова транзакція. Адже транзакції справді є дуже потужним засобом управління базою даних.

Розглянемо класичне завдання – банківське проведення. Припустимо, що у нас є таблиця із двох полів – номер рахунку у банку та сума грошей на цьому рахунку. Нам необхідно переказати гроші з одного рахунку на інший. Для цього потрібно виконати запит на `UPDATE`, щоб зменшити суму першого рахунку на потрібну суму. Після цього потрібно виконати запит на `UPDATE`, щоб збільшити значення другого рахунку. Все начебто нормально. Але може так статися, що після зменшення першого рахунку різко вимкнули світло і сервер не встиг поповнити інший рахунок? Або сталась ще якась помилка, яка не дала завершити дію. Гроші вже зняті, але нікуди не записані, отже – вони зникли.

Якщо спочатку поповнювати рахунок, а потім знімати гроші, то може статись аналогічна ситуація, але вже навпаки: поповнення відбулось, а зняття з якихось причин ні. Тут вже банк ризикує збанкрутувати, адже з'явилась зайва сума на рахунку одного клієнта, а зняття з рахунку іншого не відбулося.

Так, раптове знеструмлення або збій у роботі сервера відбуваються рідко, але також бувають блокування записів. Один користувач може заблокувати запис для зміни і це не дасть запиту змінити його та зняття або збільшення рахунку не відбудеться.

Проблему вирішує транзакція. Перед виконанням операцій оновлення необхідно явно розпочати транзакцію. Після цього виконуємо дві операції UPDATE і після закінчення завершуємо транзакцію. Якщо на момент виконання одного із запитів відбувається збій, то всі зміни, що відбулися після початку транзакції, скасовуються.

Існує два різновиди транзакцій у SQL Server:

приховані транзакції – кожен оператор, такий як INSERT, UPDATE або DELETE, виконується в транзакції;

явні транзакції оголошені користувачем – оператори, згруповані у BEGIN TRANSACTION та COMMIT TRANSACTION.

Дуже важливо розуміти, що транзакції необхідні лише за модифікації даних, тобто під час використання операторів INSERT, UPDATE чи DELETE. Проста вибірка SELECT не змінює даних, і запам'ятовувати чи відкочувати нічого. Ні, виконувати операції вибірки в транзакції можна, але якщо транзакція не змінює дані, то нема чого її взагалі починати.

У транзакції можна вмикати далеко не всі оператори. Усередині транзакції не може бути таких операторів:

ALTER DATABASE

BUCKUP LOG

CREATE DATABASE

DROP DATABASE

RECONFIGURE

RESTORE DATABASE

RESTORE LOG

UPDATE STATISTICS

Робота транзакцій забезпечується за допомогою журналу транзакцій бази даних. Оскільки журнал записує всю активність і зміни, які відбуваються, SQL Server може відновлювати дані автоматично, які були втрачені в момент знеструмлення, через системні помилки, у зв'язку з проблемами з боку клієнта, через запит скасування журналу тощо.

Сервер SQL автоматично гарантує, що всі завершені транзакції відображаються в базі даних у момент та після системної помилки. Для цього використовується журнал транзакцій, за допомогою якого завершуються успішно закінчені транзакції та скасовуються не завершені.

За допомогою прихованих транзакцій сервер гарантує, що дані будуть збережені в таблиці, якщо оператор додавання, зміни або видалення даних виконано вдало. Якщо під час змін сталася помилка, всі зміни відкочуються. Припустимо, що оператор UPDATE змінює 1000 рядків. Якщо на 500-му рядку сталася помилка, то сервер відкочує всі вже зроблені зміни, як вони відбувалися в явній транзакції.

Користувач може встановлювати точку збереження або маркер усередині транзакції. Точка збереження визначає розташування, до якого може бути повернена транзакція. Транзакція має або продовжити виконання, або бути повністю відкоченою.

Початок транзакції в MS SQL Server має такий синтаксис:

```
BEGIN TRANSACTION  
[ { transaction_name | @transaction_name_variable }  
[WITH MARK['description']]]
```

Опція transaction_name – ім'я транзакції, визначене користувачем. Опція WITH MARK вказує, що транзакція маркована у журналі транзакцій.

Після завершення транзакції зміни необхідно запам'ятати в базі даних. Для цього використовується команда COMMIT

```
COMMIT [ TRANSACTION  
[ transaction_name | @tran_name_variable ] ]
```

Скасування транзакції та всіх змін здійснюватиметься командою ROLLBACK, яка у загальному вигляді виглядає таким чином:

```
ROLLBACK [ TRANSACTION  
[ transaction_name | @tran_name_variable  
| savepoint_name | @savepoint_variable ] ]
```

Дуже важливо розуміти, якщо розпочато транзакцію та змінено якісь записи, то ці записи блокуються, доки транзакція не буде завершена.

Розробимо транзакцію, яка очищує одну із таблиць і виводить дані з таблиці до відкату та після відкату:

```
USE [check]  
-- Починаємо транзакцію  
BEGIN TRANSACTION  
  
-- Очищаємо таблицю  
DELETE GoodsDoc  
  
SELECT * FROM GoodsDoc  
  
ROLLBACK TRANSACTION  
  
SELECT * FROM GoodsDoc
```

Результат (рис. 2.100):

	Num	GoodsId	Q
--	-----	---------	---

	Num	GoodsId	Q
1	1	1	5
2	2	1	2
3	2	2	5
4	2	3	4
5	3	2	3
6	3	3	4
7	3	4	2

Рисунок 2.100 – Результат відкату транзакції

Як видно з результату, до відкату таблицю було дійсно очищено (вона порожня), а після відкату дані відновились з транзакції.

Візьмемо таблицю goods бази даних check (рис. 2.101).

DESKTOP-AA2MEC1\...heck - dbo.goods		SQLQuery1.sql - (L	
	Id	Name	Price
	1	Пальто жін. си...	3000
	2	Пальто жін. зи...	4500
	3	Пальто жін. чо...	12400
	4	Пальто унісекс...	5000
	5	Пальто чол. кл...	12500
	6	Пальто чол. ч...	6500
	10	Ball	100
	11	Bag	1000
	13	Ball	50
	1013	Galaxy S7	5000
	2013	plate	60
	2014	DRESS	20,4
	2016	DRESS1	18
	2017	DRESS2	15
	2018	DRESS2	12
*	NULL	NULL	NULL

Рисунок 2.101 – Таблиця goods бази даних check

Створимо дві послідовні транзакції. Перша транзакція додає товар DRESS3 з ціною 12.5 в таблицю goods, потім змінює його ціну на 15 і зберігає зміни. Друга – змінює ціну товару на 17 і робить відкат до попереднього стану. Виконаємо запит:

```

USE [check]
-- Починаємо транзакцію

BEGIN TRANSACTION

-- Вставляємо рядок даних у
-- таблицю товарів
INSERT INTO goods( Name, Price)
VALUES ( 'DRESS3', 12.50)

-- Оновити дані в останньому рядку
UPDATE goods
SET Price = 15
WHERE Name LIKE 'DRESS3'

COMMIT TRANSACTION

BEGIN TRANSACTION

-- Оновити дані в останньому рядку
UPDATE goods
SET Price = 17
WHERE Name LIKE 'DRESS3'

-- Відкотити транзакцію
ROLLBACK TRANSACTION

-- Вибрати всі дані з таблиці
SELECT * FROM goods

```

Результат (рис. 2.102):

	Id	Name	Price
9	13	Ball	50
10	1...	Galaxy S7	5000
11	2...	plate	60
12	2...	DRESS	20,4
13	2...	DRESS1	18
14	2...	DRESS2	15
15	2...	DRESS2	12
16	2...	DRESS3	18

Рисунок 2.102 – Результат роботи послідовних транзакцій

Розглянемо вкладені транзакції:

```
BEGIN TRANSACTION T1

-- Вставляємо рядок даних у таблицю товарів
INSERT INTO goods
  (Name, Price)
VALUES ('T-shirt', 11.30)

-- Починаємо транзакцію 2
BEGIN TRANSACTION T2

-- Оновити дані в останньому рядку
UPDATE goods
SET Price = 14
WHERE Name LIKE 'T-shirt'

COMMIT TRANSACTION T2

-- Запам'ятовуємо зміни
COMMIT TRANSACTION T1

-- Вибрати всі дані з таблиці
SELECT * FROM goods
```

Результат (рис. 2.103):

	Id	Name	Price
10	1...	Galaxy S7	5000
11	2...	plate	60
12	2...	DRESS	20,4
13	2...	DRESS1	18
14	2...	DRESS2	15
15	2...	DRESS2	12
16	2...	DRESS3	18
17	2...	T-shirt	16,8

Рисунок 2.103 – Результат роботи вкладених транзакцій

Зверніть увагу, що ціна в результаті на T-shirt не 14, а 16,8, оскільки спрацював тригер на вставку (додає 20% від ціни).

Виконаємо відкат до визначеної точки:

```
USE [check]
-- Починаємо транзакцію

BEGIN TRANSACTION T1

-- Вставляємо рядок даних у таблицю товарів
INSERT INTO goods
    (Name, Price)
VALUES ('T-shirt1', 14)

-- Зберігаємо транзакцію
SAVE TRAN ins_complete

-- Оновити дані в останньому рядку
UPDATE goods
SET Price = 20
WHERE Name LIKE 'T-shirt1'

-- Відкочуємо транзакцію
ROLLBACK TRANSACTION ins_complete

-- Запам'ятовуємо зміни
COMMIT TRANSACTION T1

-- Вибрати всі дані з таблиці
SELECT * FROM goods
```

Результат (рис. 1.104):

	Id	Name	Price
11	2...	plate	60
12	2...	DRESS	20,4
13	2...	DRESS1	18
14	2...	DRESS2	15
15	2...	DRESS2	12
16	2...	DRESS3	18
17	2...	T-shirt	16,8
18	2...	T-shirt1	16,8

Рисунок 2.104 – Транзакція з точкою відкату

Питання для самоперевірки

Що таке вбудовані функції SQL?

Що таке збережені функції SQL?

Де зберігаються збережені функції на SQL Server?

Де зберігаються збережені процедури на SQL Server?

Що таке збережена процедура? Яка різниця між функцією і процедурою?

Для чого призначені подання?

Що таке тригер? Які тригери ви знаєте?

Звідки бере інформацію тригер для зміни?

Що таке транзакція? Які операції дозволені в транзакції?

Що таке точка відкату?

Завдання

Варіант	Опис завдання
1	Створити подання, в якому для студентів певної групи будуть виводиться оцінки (як в журналі). Написати скалярну функцію, яка повертає загальну кількість викладачів всіх факультетів. Написати функцію, яка повертає таблицю всіх кафедр університету, згруповані по факультетам. Написати процедуру, яка отримує назву факультету і визначає кількість кафедр на ньому. 5. Створити тригер, який працює при додаванні нової кафедри виводилася кількість кафедр на цьому факультеті. 6. Написати транзакцію, яка працює з додаванням факультету, при цьому ще додається кафедра з назвою «НОВА КАФЕДРА», а на цій кафедрі створюється група: «НОВА».
2	1. Створити подання, в якому за датою продажу виводиться вся інформація про продані товари. 2. Написати скалярну функцію, яка за датою продажу розраховує суму продажу товарів за день. 3. Написати функцію, яка повертає таблицю з інформацією про товари, які продані за певний день. 4. Написати процедуру, яка розраховує суму продажу товарів

	за місяць. 5.Створити тригер, який при додаванні товару, автоматично збільшує його ціну на 30%. 6. Написати транзакцію, яка при додаванні товару в таблицю продажів, автоматично записує інформацію в таблицю продажів за день (дата, сума).
3	. Створити подання, в якому за видом військ буде подаватися інформація про роти та їх особистий склад. Написати скалярну функцію, яка за поданими в неї А, В, С буде знаходити рішення квадратного рівняння. Написати функцію, яка повертає таблицю, що за роком буде повертати всю інформацію про особистий склад. Написати процедуру, яка знаходить суму всіх площ місць дислокації частин. 5.Створити тригер, який при додаванні нового військового в особистий склад буде виводити дату закінчення його служби. 6. Написати транзакцію при додаванні нової роти, додавала нове місце дислокації.
4	1.Створити подання, в якому виводиться інформація про співробітників певної бібліотеки. 2.Написати скалярну функцію, яка за заданою датою визначає назву місяця. 3.Написати функцію, яка повертає таблицю, в якій знаходиться інформація про всі фонди для певної бібліотеки. 4. Написати процедуру, яка переводить всю інформацію, подану в таблиці Співробітники, у верхній регістр 5.Створити тригер, який при додаванні нового співробітника, збільшує його оклад удвічі. 6. Написати транзакцію, яка при додаванні нового фонду,

	автоматично додавалась інформація про поповнення цього фонду.
5	1. Створити подання, в якому виводиться інформація про пансіонати з певним видом житла. 2. Написати скалярну функцію <i>arithmetic</i>, приймаючи 3 аргументи: перші 2 – це числа, третій – операція, яка повинна бути проведена над ними. Якщо третій аргумент +, скласти їх; відповідно – відняти; * помножити; / розділити (перше на друге). В інших випадках повернути рядок «Невідома операція». 3. Написати функцію, яка поверне інформацію про клієнтів, які купили путівки на 10 днів. 4. Написати процедуру, яка переводить всю інформацію, подану в таблиці Клієнти, у нижній регістр. 5. Створити тригер, який працює при зміні назви пансіонату і додає до нової назви дату зміни назви. 6.. Написати транзакцію, яка додає інформацію в таблицю продажів, додає інформацію і про клієнтів.

2.7 Інтеграція прикладних програм із sql-базами даних

2.7.1 Введення в ADO.NET. ADO.NET – технологія роботи з даними, яка заснована на платформі .NET Framework. Ця технологія надає набір класів для відправлення запитів до баз даних, встановлення підключення, отримання відповіді від бази даних і ряду інших операцій.

Причому важливо зазначити, що систем управління базами даних (СУБД) може бути безліч. У своїй сутності вони можуть відрізнятися. MS SQL Server, наприклад, для створення запитів використовує мову T-SQL, а MySQL і Oracle застосовують мову PL-SQL. Різні СУБД можуть мати різні типи даних. Однак функціонал ADO.NET побудований таким чином, щоб надати розробникам уніфікований інтерфейс для роботи з найрізноманітнішими СУБД.

Оснoву інтерфейсу взаємодії з базами даних в ADO.NET складає обмежене коло об'єктів: Connection, Command, DataReader, DataSet і DataAdapter. За допомогою об'єкта Connection встановлюється підключення до джерела даних. Об'єкт Command дає змогу виконувати операції з даними з БД. Об'єкт DataReader зчитує отримані у результаті запиту дані. Об'єкт DataSet призначений для зберігання даних з БД і надає можливість працювати з ними незалежно від БД. Об'єкт DataAdapter є посередником між DataSet і джерелом даних. Загалом через ці об'єкти і буде здійснюватися робота з базою даних.

Однак, щоб використовувати один і той же набір об'єктів для різних джерел даних, потрібен відповідний провайдер даних. Власне через провайдера даних в ADO.NET і здійснюється взаємодія з базою даних. Причому для кожного джерела даних в ADO.NET може бути свій провайдер, який власне і визначає конкретну реалізацію вищевказаних класів.

За замовчуванням в ADO.NET є такі вбудовані провайдери:

провайдер для MS SQL Server;

провайдер для OLE DB – надає доступ до деяких старих версій MS SQL Server, а також до БД Access, DB2, MySQL і Oracle;

провайдер для ODBC – використовується для тих джерел даних, для яких немає своїх провайдерів;

провайдер для Oracle;

провайдер EntityClient – використовується для технології ORM Entity Framework;

провайдер для сервера SQL Server Compact 4.0.

Крім цих провайдерів, які є вбудованими, існує також безліч інших, призначених для різних СУБД, наприклад, для MySQL.

Основні простори імен, які використовуються в ADO.NET:

System.Data визначає класи, інтерфейси, делегати, які реалізують архітектуру ADO.NET;

System.Data.Common містить класи, спільні для всіх провайдерів ADO.NET;

`System.Data.Design` визначає класи, які використовуються для створення своїх власних наборів даних;

`System.Data.Odbc` визначає функціональність провайдера даних для ODBC;

`System.Data.OleDb` визначає функціональність провайдера даних для OLE DB;

`System.Data.Sql` зберігає класи, які підтримують специфічну для SQL Server функціональність;

`System.Data.OracleClient` визначає функціональність провайдера для баз даних Oracle;

`System.Data.SqlClient` визначає функціональність провайдера для баз даних MS SQL Server;

`System.Data.SqlServerCe` визначає функціональність провайдера для SQL Server Compact 4.0;

`System.Data.SqlTypes` містить класи для типів даних MS SQL Server

`Microsoft.SqlServer.Server`: зберігає компоненти для взаємодії SQL Server і середовища CLR.

Функціонально класи ADO.NET можна розбити на два рівня: підключень і відключень. Кожен провайдер даних .NET реалізує свої версії об'єктів `Connection`, `Command`, `DataReader`, `DataAdapter` і ряду інших, які складають рівень підключень. Тобто за допомогою них встановлюється підключення до БД і виконується взаємодія з нею. Як правило, реалізації цих об'єктів для кожного конкретного провайдера в своїй назві мають префікс, який вказує на назву провайдера.

Інші класи, такі як `DataSet`, `DataTable`, `DataRow`, `DataColumn` і ряд інших, складають рівень відключень, оскільки після отримання даних в `DataSet` можливо працювати з цими даними незалежно від того, чи встановлено підключення, чи ні. Тобто після отримання даних з БД застосунок може бути відключено від джерела даних.

2.7.2 C# і MS SQL Server

Взаємодія між програмним застосунком на C# та SQL Server є фундаментальною складовою розробки більшості корпоративних і вебдодатків. Ця взаємодія відбувається за принципом клієнт-сервер, де C# застосунок є клієнтом, а SQL Server — сервером бази даних.

Для реалізації цієї взаємодії в екосистемі .NET існують два основні підходи: класичний ADO.NET та сучасніший Entity Framework.

Створення простого консольного застосунку на C# з підключенням до MS SQL Server — це чудовий спосіб освоїти базові принципи взаємодії програм з базами даних. Цей процес починається з визначення рядка підключення.

Рядок підключення — це важливий елемент конфігурації, який містить усі необхідні дані для встановлення зв'язку між вашим застосунком та базою даних. Він інформує програму про те, де знаходиться сервер, яку базу даних використовувати, і як аутентифікуватися.

```
using System.Text;
using System.Threading.Tasks;
using System.Data.SqlClient;

namespace ConsoleApp1
{
    class Program
    {
        static void Main(string[] args)
        {
            string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=check; User Id =sa; Password =140317";
            // Створюємо підключення
            SqlConnection connection = new SqlConnection(connectionString);
            try
            {
                // Відкриваємо підключення
                connection.Open();
                Console.WriteLine("Підключення відкрите");
            }
            catch (SqlException ex)
            {
                Console.WriteLine(ex.Message);
            }
            finally
            {
                // закриваємо підключення
                connection.Close();
                Console.WriteLine("Підключення закрито...");
            }

            Console.Read();
        }
    }
}
```

В результаті запуску цієї програми у консоль буде виведено:

```
Підключення відкрите  
Підключення закрито...
```

У разі використання різних систем управління базами даних, різних провайдерів даних .NET рядок підключення може відрізнятися. Навіть для підключення однієї і тієї ж бази даних рядок підключення може змінюватися залежно від обставин.

Рядок підключення – це набір параметрів у вигляді пар «ключ = значення». В цьому разі для підключення до раніше створеної бази даних check треба визначити рядок підключення з чотирьох параметрів:

Data Source – назва сервера. За замовчуванням це «.\SQLEXPRESS». Оскільки в рядку використовується слеш, то на початку рядка ставиться символ @. Якщо ім'я сервера бази даних відрізняється, то відповідно його і треба використовувати.

Initial Catalog – назва бази даних на сервері.

User Id – Id користувача SQL Server.

Password – пароль цього користувача.

Жорстке кодування рядка підключення (тобто його визначення в коді програми), як правило, рідко використовується. Набагато більш гнучкий шлях – визначення його в спеціальних конфігураційних файлах програми. У проєктах десктопних застосунків – це файл App.config, а в веб-застосунках – це в основному файл Web.config.

Оскільки у прикладі створено проєкт консольного застосування, то він містить файл App.config (рис. 2.105).

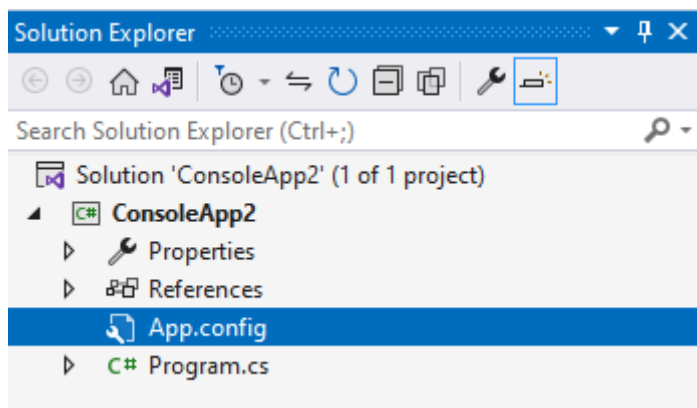
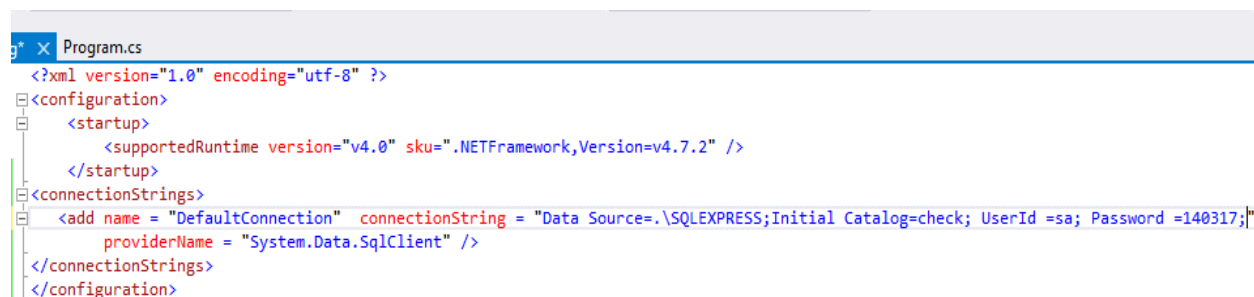


Рисунок 2.105 – Файл App.config в оглядачі рішень

На цьому етапі файл App.config має такий зміст:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.7.2" />
  </startup>
</configuration>
```

Додамо у цей файл визначення рядка підключення:



```
Program.cs
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.7.2" />
  </startup>
  <connectionStrings>
    <add name = "DefaultConnection" connectionString = "Data Source=.\\SQLEXPRESS;Initial Catalog=check; UserId =sa; Password =140317;"
      providerName = "System.Data.SqlClient" />
  </connectionStrings>
</configuration>
```

Для визначення всіх підключень в програмі в межах вузла `<configuration>` додається новий вузол `<connectionStrings>`. У цьому вузлі визначаються рядки підключення за допомогою елемента `<add>`. У застосунку можливо використовувати безліч рядків підключення. У файлі також можна визначити безліч елементів `<add>`.

Кожен рядок підключення має назву, яка визначається за допомогою атрибута `name`. В цьому разі рядок підключення називається «DefaultConnection». Назва може бути довільною. Атрибут `connectionString` зберігає рядок підключення, тобто весь той текст, який було визначено вище в методі `Main`. Третій атрибут – `providerName` – задає простір імен провайдера даних. Для підключення до бази даних MS SQL Server варто використовувати провайдер для SQL Server, функціональність якого полягає в просторі імен `System.Data.SqlClient`.

Тепер отримаємо цей рядок підключення у застосунку:

```

using System;
using System.Configuration;

namespace ConsoleApp1
{
    class Program
    {
        static void Main(string[] args)
        {
            string connectionString = ConfigurationManager.ConnectionStrings["DefaultConnection"].ConnectionString;
            Console.WriteLine(connectionString);

            Console.Read();
        }
    }
}

```

В результаті запуску цієї програми у консоль буде виведено інформацію про SQL Server:

```
Data Source=.\SQLEXPRESS;Initial Catalog=check; UserId =sa; Password =140317;
```

Для роботи з конфігурацією застосунку потрібно додати в проєкт бібліотеку System.Configuration.dll. Для цього в оглядачі рішень натискаємо правою кнопкою миші на вузлі Посилання і вибираємо у контекстному меню Додати посилання (рис. 2.106).

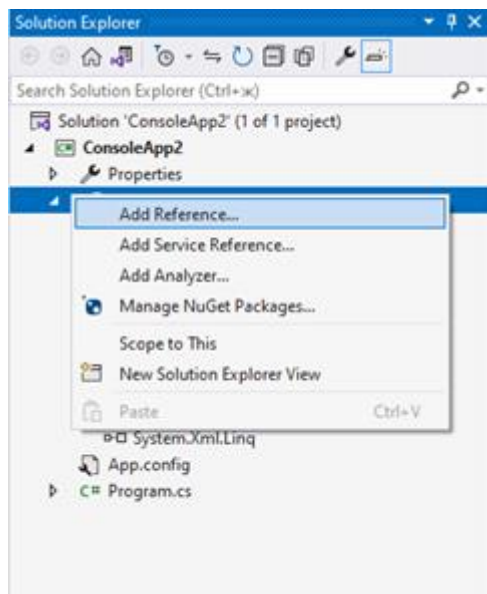


Рисунок 2.106 – Пункт «Додати посилання» контекстного меню в оглядачі рішень

Відкриється вікно управління посиланнями, ставимо галочку напроти System.Configuration (рис. 2.107).

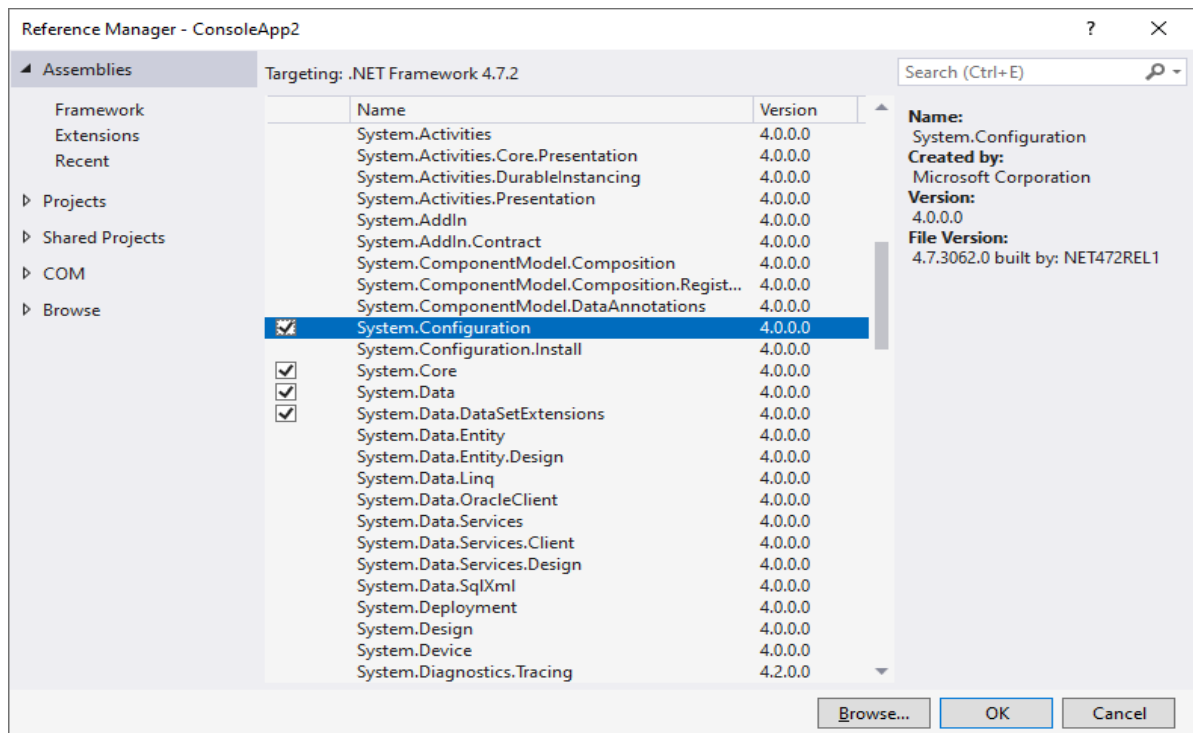


Рисунок 2.107 – Вікно управління посиланнями

За допомогою об'єкта `ConfigurationManager.ConnectionStrings` [«назва_рядка_підключення»] можливо отримати рядок підключення і використовувати його в застосунку.

Параметри рядка підключення:

Application Name – назва програми. Може приймати в якості значення будь-який рядок. Значення за замовчуванням: «.Net SqlConnection Data Provide»;

AttachDBFileName – зберігає повний шлях до бази даних;

Connect Timeout – часовий період в секундах, через який очікується установка підключення. Приймає одне зі значень з інтервалу 0-32767. За замовчуванням дорівнює 15. В якості альтернативної назви параметра може використовуватися `Connection Timeout`;

Data Source – назва примірника SQL Server, з яким буде йти взаємодія;

Integrated Security – встановлює перевірку пароля.

Для підключення до бази даних необхідно створити і використовувати об'єкт `SqlConnection`:


```

using System.Text;
using System.Threading.Tasks;
using System.Data.SqlClient;

namespace ConsoleApp1
{
    class Program
    {
        static void Main(string[] args)
        {
            string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=check; User Id =sa; Password =140317";
            // Створюємо підключення
            SqlConnection connection = new SqlConnection(connectionString);
            try
            {
                // Відкриваємо підключення
                connection.Open();
                Console.WriteLine("Підключення відкрите");
            }
            catch (SqlException ex)
            {
                Console.WriteLine(ex.Message);
            }
            finally
            {
                // закриваємо підключення
                connection.Close();
                Console.WriteLine("Підключення закрито...");
            }

            Console.Read();
        }
    }
}

```

У конструктор об'єкту `SqlConnection` передається рядок підключення, який ініціалізує об'єкт. Щоб використовувати цей об'єкт і підключатися до бази даних, потрібно виконати його метод `Open()`, а після завершення роботи з базою даних треба викликати метод `Close()` для закриття підключення. У разі виникнення помилок закриття підключення відбувається в блоці `finally`.

В якості альтернативного методу можна використовувати конструкцію `using`, яка автоматично закриває підключення:

```

using System;

using System.Data.SqlClient;

namespace ConsoleApp1
{
    Ссылка: 0
    class Program
    {
        Ссылка: 0
        static void Main(string[] args)
        {
            string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=check; User Id =sa; Password =140317;";

            // Створення підключення
            using (SqlConnection connection = new SqlConnection(connectionString))
            {
                // Відкриваємо підключення
                connection.Open();
                Console.WriteLine("Підключення відкрито");
            }

            Console.WriteLine("Підключення закрито ...");

            Console.Read();
        }
    }
}

```

В результаті запуску цієї програми у консоль буде виведено:

```

Підключення відкрито
Підключення закрито ...

```

Пул з'єднань — це механізм оптимізації продуктивності, який автоматично керується .NET Framework і використовується за замовчуванням. Встановлення нового фізичного з'єднання з базою даних є ресурсоємним і повільним процесом. Замість того, щоб щоразу відкривати і закривати фізичне з'єднання, пул переробляє існуючі, вже відкриті з'єднання. Це значно покращує продуктивність і масштабованість додатків, які часто відкривають і закривають з'єднання.

Об'єкт SqlConnection має низку властивостей, які дають змогу отримати інформацію про підключення:

```
using System;

using System.Data.SqlClient;

namespace ConsoleApp1
{
    Ссылка: 0
    class Program
    {
        Ссылка: 0
        static void Main(string[] args)
        {
            string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=check; User Id =sa; Password =140317;";

            // Створення підключення
            using (SqlConnection connection = new SqlConnection(connectionString))
            {
                // Відкриваємо підключення
                connection.Open();
                Console.WriteLine("Підключення відкрито");
                // Виведення інформації про підключення
                Console.WriteLine("Властивості підключення:");
                Console.WriteLine("\tРядок підключення: {0}", connection.ConnectionString);
                Console.WriteLine("\tБаза даних: {0}", connection.Database);
                Console.WriteLine("\tСервер: {0}", connection.DataSource);
                Console.WriteLine("\tВерсія сервера: {0}", connection.ServerVersion);
                Console.WriteLine("\tСтан: {0}", connection.State);
                Console.WriteLine("\tWorkstationId: {0}", connection.WorkstationId);
            }

            Console.WriteLine("Підключення закрито ...");
        }
    }
}
```

Результат виконання програми:

```
Підключення відкрито
Властивості підключення:
    Рядок підключення: Data Source=.\SQLEXPRESS;Initial Catalog=check; User Id =sa;
    База даних: check
    Сервер: .\SQLEXPRESS
    Версія сервера: 12.00.2269
    Стан: Open
    WorkstationId: DESKTOP-AA2MEC1
Підключення закрито ...
```

При підключенні до бази даних можуть виникнути помилки. Найбільш поширеною помилкою є така:

Unhandled Exception: System.ArgumentException: Keyword not supported ...

Така помилка викликана через неправильне задання параметрів рядка підключення. До того ж неважливо, як було визначено рядок підключення – в

кодi або в файлі конфігурації. У повідомленні про помилку вказується, який параметр задано неправильно, і в цьому випадку його треба виправити.

Ще одна поширена помилка така:

Can not open database «назва бази даних» requested by the login. The login failed.

Login failed for user 'користувач'

Вона означає, що даної бази немає або принаймні немає подібної бази даних. В цьому випадку треба переконатися, що на сервері є база даних з такою назвою, а якщо є, то перевірити, чи є доступ для даного користувача до цієї БД.

Також часто зустрічається помилка:

A network-related or instance-specific error occurred while establishing a connection to SQL Server.

The server was not found or was not accessible ...

У цьому випадку, як правило, SQL Server не запущений. І його треба запустити або перезапустити через панель служб. Також може бути, що в рядку підключення вказано неправильне ім'я сервера, або MS SQL Server взагалі не встановлено.

Після встановлення підключення можливо виконувати команди з базою даних, наприклад, додати в базу даних об'єкт, видалити, змінити його або просто витягти. Команди подані об'єктом інтерфейсу System.Data.IDbCommand. Провайдер для MS SQL надає його реалізацію у вигляді класу SqlCommand. Цей клас інкапсулює sql-вираз, який потрібно виконати.

Для виконання команди нам буде потрібно sql-вираз і об'єкт підключення:

```

class Program
{
    Ссылка: 0
    static void Main(string[] args)
    {
        string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=check; User Id =sa; Password =140317;";

        // Створення підключення
        using (SqlConnection connection = new SqlConnection(connectionString))
        {
            // Відкриваємо підключення
            connection.Open();
            Console.WriteLine("Підключення відкрито");
            SqlCommand command = new SqlCommand();
            command.CommandText = "SELECT * FROM Users";
            command.Connection = connection;

        }

        Console.WriteLine("Підключення закрито ...");

        Console.Read();
    }
}

```

За допомогою властивості CommandText встановлюється SQL-вираз, який буде виконуватися. В цьому випадку – це запит на отримання всіх об’єктів з таблиці Users. А за допомогою властивості Connection можна встановити об’єкт підключення SqlConnection.

В якості альтернативи можна було б використовувати одну з версій конструктора класу:

```

class Program
{
    Ссылка: 0
    static void Main(string[] args)
    {
        string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=check; User Id =sa; Password =140317;";

        // Створення підключення
        using (SqlConnection connection = new SqlConnection(connectionString))
        {
            // Відкриваємо підключення
            connection.Open();
            Console.WriteLine("Підключення відкрито");
            SqlCommand command = new SqlCommand();
            command.CommandText = "SELECT * FROM Employee";
            command.Connection = connection;

        }

        Console.WriteLine("Підключення закрито ...");

        Console.Read();
    }
}

```

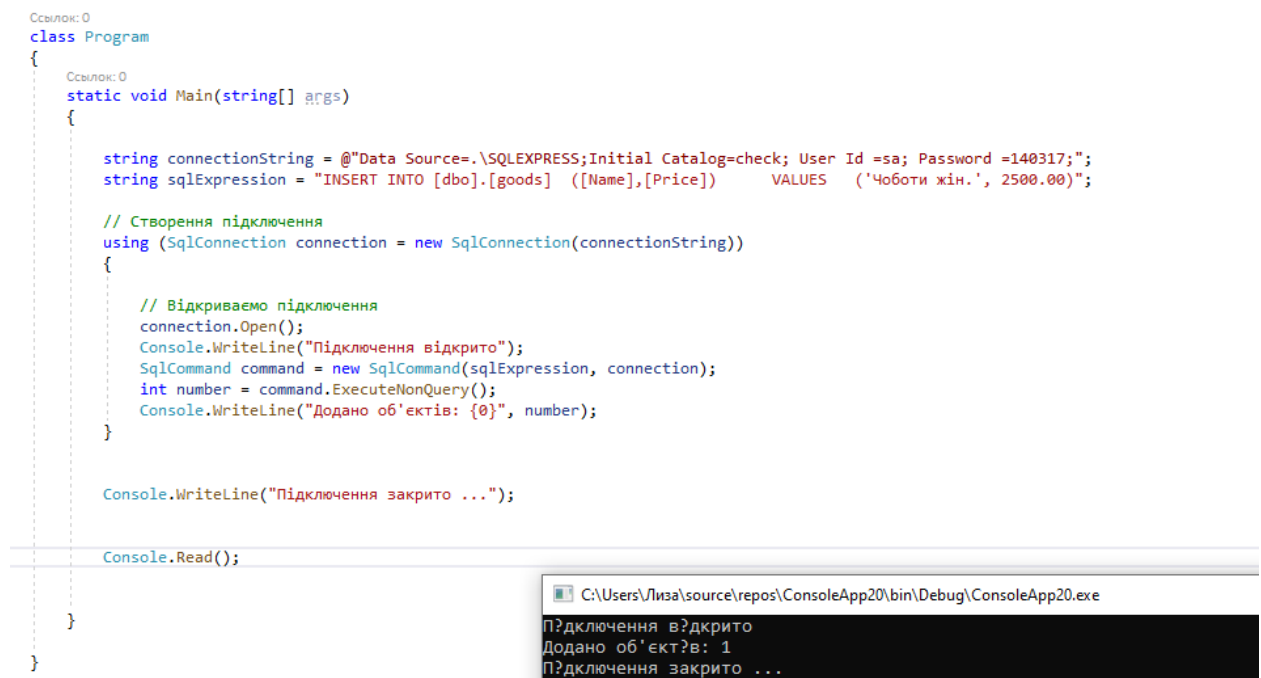
Щоб виконати команду, необхідно застосувати один з методів SqlCommand:

ExecuteNonQuery – просто виконує sql-вираз і повертає кількість змінених записів. Підходить для sql-виразів INSERT, UPDATE, DELETE.

ExecuteReader – виконує sql-вираз і повертає рядки з таблиці. Підходить для sql-вирази SELECT.

ExecuteScalar – виконує sql-вираз і повертає одне скалярне значення, наприклад, число. Підходить для sql-виразу SELECT в парі з однією із вбудованих функцій SQL, як наприклад, Min, Max, Sum, Count.

Виконаємо команду додавання одного об'єкта в таблицю goods бази даних check (рис. 2.108).



```
Ссылка: 0
class Program
{
    Ссылка: 0
    static void Main(string[] args)
    {
        string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=check; User Id=sa; Password=140317;";
        string sqlExpression = "INSERT INTO [dbo].[goods] ([Name],[Price]) VALUES ('Чоботи жін.', 2500.00)";

        // Створення підключення
        using (SqlConnection connection = new SqlConnection(connectionString))
        {
            // Відкриваємо підключення
            connection.Open();
            Console.WriteLine("Підключення відкрито");
            SqlCommand command = new SqlCommand(sqlExpression, connection);
            int number = command.ExecuteNonQuery();
            Console.WriteLine("Додано об'єктів: {0}", number);
        }

        Console.WriteLine("Підключення закрито ...");

        Console.Read();
    }
}
```

C:\Users\Лиза\source\repos\ConsoleApp20\bin\Debug\ConsoleApp20.exe

Підключення відкрито
Додано об'єктів: 1
Підключення закрито ...

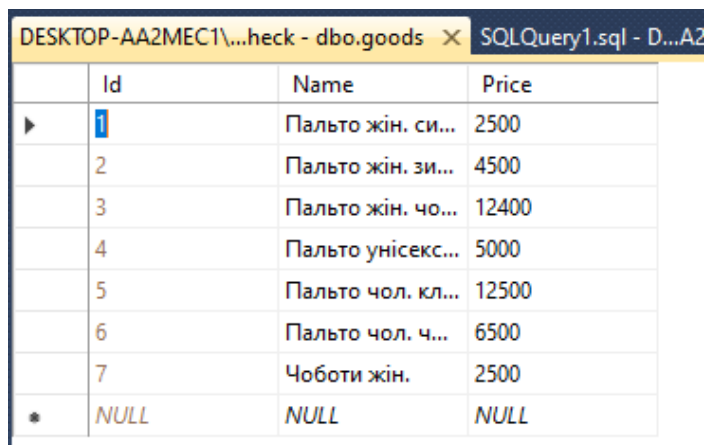
Рисунок 2.108 – Додавання одного об'єкта в таблицю goods бази даних check

Для вставки об'єкта використовується sql-вираз INSERT, що має такий синтаксис:

INSERT INTO назва_таблиці (стовпчик1, стовпчик2, стовпчикN)
VALUES (значення1, значення2, значенняN)

В таблиці goods бази даних check є три стовпці – Id і Price, які зберігають цілі числа і дійсні числа, та Name, який зберігає рядок. Тому відповідно ми

додаємо для стовпця Name значення ‘Чоботи жін.’, а для стовпця Price число 2500.00 (рис. 2.109).



	Id	Name	Price
▶	1	Пальто жін. си...	2500
	2	Пальто жін. зи...	4500
	3	Пальто жін. чо...	12400
	4	Пальто унісекс...	5000
	5	Пальто чол. кл...	12500
	6	Пальто чол. ч...	6500
	7	Чоботи жін.	2500
*	NULL	NULL	NULL

Рисунок 2.109 – Результат додавання запису в таблицю goods бази даних check

Метод ExecuteNonQuery() повертає число змінених рядків (в цьому випадку доданих в таблицю об’єктів). Не обов’язково повертати результат методу, але він може використовуватися задля перевірки, що операція, зокрема, додавання, пройшла успішно.

Оновлення буде відбуватися аналогічно, тільки за допомогою sql-виразу UPDATE, який має такий синтаксис:

UPDATE назва_таблиці

SET стовпчик1 = значення1, стовпчик 2 = значення2, стовпчик N = значенняN

WHERE деякий_стовпчик = деяке_значення

Застосуємо цей вираз для зміни ціни «Чоботи жін.» з 2500.00 на 6500.50 (рис. 2.110)



Рисунок 2.110 – Оновлення рядків в таблиці goods

Тут оновлюється рядок, в якому Name = 'Чоботи жін.', тобто вище доданий об'єкт (рис. 2.111). Якщо в таблиці буде кілька рядків, у яких Name = 'Чоботи жін.', то оновляться всі ці рядки.

DESKTOP-AA2MEC1\...heck - dbo.goods X SQLQuery1.sql - D...			
	Id	Name	Price
5 ▶	1	Пальто жін. си...	2500
	2	Пальто жін. зи...	4500
	3	Пальто жін. чо...	12400
	4	Пальто унісекс...	5000
	5	Пальто чол. кл...	12500
	6	Пальто чол. ч...	6500
	7	Чоботи жін.	6500,5
*	NULL	NULL	NULL

Рисунок 2.111 – Результат оновлення рядків «Чоботи жін.» в таблиці goods

Видалення проводиться за допомогою sql-виразу DELETE, який має такий синтаксис:

DELETE FROM таблиця

WHERE стовпець = значення

Видалимо, наприклад, всі товари з назвою 'Чоботи жін.' (рис. 2.112).

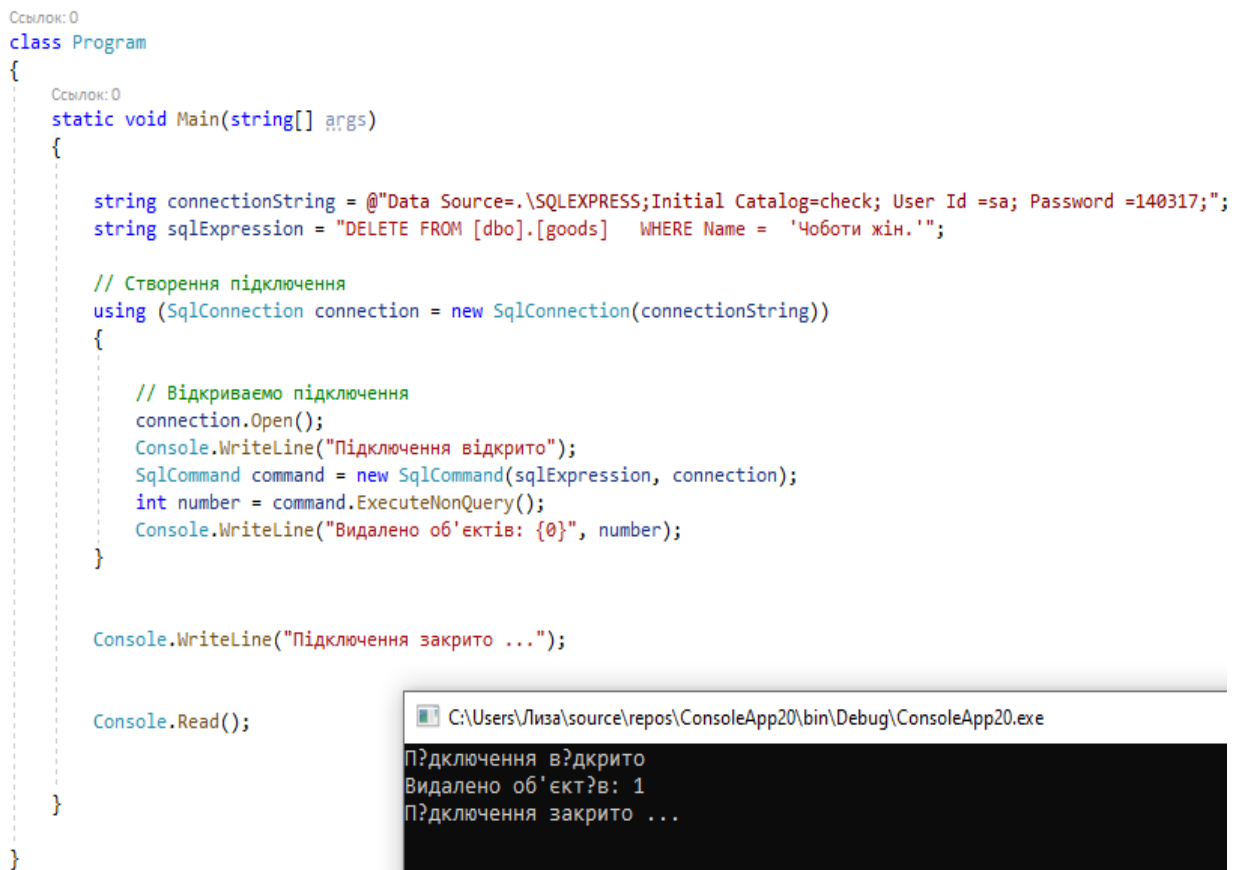


Рисунок 2.112 – Видалення рядків в таблиці goods

У всіх трьох випадках фактично змінюється тільки sql-вираз, а інша логіка залишається незмінною. Sql-вираз у програмному коді на мові C# — це фрагмент тексту, який є синтаксично коректною Sql-командою, що передається для виконання до системи керування базами даних, такою як SQL Server. По суті, це рядок, що містить інструкції для маніпуляції або запиту даних. Sql-вирази в C# зазвичай зберігаються у змінних типу string.

Основна ідея полягає в тому, що значна частина коду, яка відповідає за взаємодію з базою даних, залишається сталою, незалежно від конкретної дії, яку ви виконуєте (наприклад, читання, додавання чи оновлення даних). Ця "незмінна логіка" включає: встановлення та управління з'єднанням, створення об'єкта команди, прив'язка з'єднання, обробка помилок.

У всіх розглянутих випадках змінюється лише завдання, яке ви ставите базі даних, а алгоритм звернення до бази даних залишається однаковим, тобто SQL-вираз.

Ми також можемо виконувати відразу декілька операцій:

```
См. код: 0
static void Main(string[] args)
{
    string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=check; User Id =sa; Password =140317;";

    // Створення підключення
    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        // Відкриваємо підключення
        connection.Open();
        Console.WriteLine("Підключення відкрито");
        Console.WriteLine("Введіть назву товару, який потрібно додати:");
        string name1 = Console.ReadLine();

        Console.WriteLine("Введіть його ціну:");
        double price = Convert.ToDouble(Console.ReadLine());

        string sqlExpression = String.Format("INSERT INTO [dbo].[goods](Name, Price) VALUES ({0}, {1})", ""+name1+ "", price );

        SqlCommand command = new SqlCommand(sqlExpression, connection);
        int number = command.ExecuteNonQuery();
        Console.WriteLine("Додано об'єктів: {0}", number);
        Console.WriteLine("Введіть назву товару, ціну якого потрібно змінити:");
        name1 = Console.ReadLine();

        Console.WriteLine("Введіть його нову ціну:");
        price = Convert.ToDouble(Console.ReadLine());

        sqlExpression = String.Format("UPDATE [dbo].[goods] SET Price = {0} WHERE Name = {1}", price, "" + name1 + "");

        command = new SqlCommand(sqlExpression, connection);
        number = command.ExecuteNonQuery();
        Console.WriteLine("Змінено об'єктів: {0}", number);

        Console.WriteLine("Введіть назву товару, який потрібно видалити:");
        name1 = Console.ReadLine();

        sqlExpression = String.Format("DELETE FROM [dbo].[goods] WHERE Name = {0}", "" + name1 + "");

        command = new SqlCommand(sqlExpression, connection);
        number = command.ExecuteNonQuery();
        Console.WriteLine("Видалено об'єктів: {0}", number);
    }

    Console.WriteLine("Підключення закрито ...");

    Console.Read();
}
```

Результат виконання програми:

```

Підключення відкрито
Введіть назву товару, який потрібно додати:
ball
Введіть його ціну:
100
Додано об'єктів: 1
Введіть назву товару, ціну якого потрібно змінити:
ball
Введіть його нову ціну:
150
Змінено об'єктів: 2
Введіть назву товару, який потрібно видалити:
ball
Видалено об'єктів: 2
Підключення закрито ...

```

У попередньому прикладі було розглянуто, як виконувати команди за допомогою методу `ExecuteNonQuery()`. Якщо ж потрібно зчитувати дані, які зберігаються в таблиці, то нам буде потрібно інший метод – `ExecuteReader()`. Цей метод повертає об'єкт `SqlDataReader`, який використовується для читання даних. Отримаємо всі дані з таблиці `goods` і виведемо їх на консоль (рис. 2.113).

```

using System;
using System.Data.SqlClient;

namespace ConsoleApp1
{
    class Program
    {
        static void Main(string[] args)
        {
            string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=check; User Id =sa; Password =140317;";
            string sqlExpression = "SELECT * FROM [dbo].[goods]";
            using (SqlConnection connection = new SqlConnection(connectionString))
            {
                connection.Open();
                SqlCommand command = new SqlCommand(sqlExpression, connection);
                SqlDataReader reader = command.ExecuteReader();

                if (reader.HasRows) // якщо є дані
                {
                    // виводимо назви стовпців
                    Console.WriteLine("{0}\t {1}\t {2}", reader.GetName(0), reader.GetName(1), reader.GetName(2));

                    while (reader.Read())
                    {
                        object id = reader.GetValue(0);
                        object name = reader.GetValue(1);
                        object price = reader.GetValue(2);

                        Console.WriteLine("{0}\t {1} \t {2}", id, name, price);
                    }
                }

                reader.Close();
            }

            Console.Read();
        }
    }
}

```

Значение из последнего сеанса от ConfigurationManager.Connect

Id	Name	Price
1	Пальто ж?н. синє	2500
2	Пальто ж?н. зимнє, дем?сизонне	4500
3	Пальто ж?н. чорне	12400
4	Пальто ун?секс. чорне	5000
5	Пальто чол. кл?тинка	12500
6	Пальто чол. чорне	6500

Рисунок 2.113 – Виведення усіх даних з таблиці `goods` в консоль

Для вибірки даних з БД використовується sql-вираз SELECT. У цьому прикладі вибираються всі стовпці усіх рядків таблиці. Результатом виконання запиту є об'єкт SqlDataReader, який надає потрібні дані.

Але спочатку виконується перевірка за допомогою властивості HasRows, а чи є взагалі дані. Якщо дані є, то виводяться заголовки таблиці за допомогою методів reader.GetName(). Варто зазначити, що отримані стовпці розташовані в вибірці саме в тому порядку, в якому вони визначені в таблиці. Тобто, якщо другим в таблиці йде стовпець «Name», то, щоб отримати його, застосовується метод GetName(1). Тут індекс у дужках 1, оскільки нумерація стовпців починається з нуля.

Далі зчитуємо самі дані. За допомогою методу reader.Read() відбувається перехід до наступного рядка і повертається логічне значення, яке вказує, чи є дані для зчитування.

У циклі while (reader.Read ()) в порядку проходження стовпців отримуємо дані за допомогою методу GetValue(), який повертає дані у вигляді об'єкти типу object. Наприклад, стовпець Id є першим, тому для його отримання застосовується метод reader.GetValue(0). А стовпець Name є другим, тому його значення отримуємо за допомогою reader.GetValue (1).

Після завершення роботи з SqlDataReader його потрібно закрити методом Close(). Поки один SqlDataReader не закритий, інший об'єкт SqlDataReader для одного і того ж підключення використовувати не вдасться.

В якості альтернативи можливо звертатися до даних через назву параметра:

```
while (reader.Read())
{
    object id = reader["Id"];
    object name = reader["Name"];
    object age = reader["Price"];
}
```

В цьому випадку результат буде аналогічним.

2.7.3 Визначення контексту даних і моделей

LINQ to SQL – технологія доступу і управління реляційними даними. Дана технологія дає змогу складати запити до БД у зручній формі у за допомогою операторів LINQ, які потім трансформуються в sql-вирази. Ключовими об’єктами тут є сутності, які зберігаються в базі даних, контекст даних і запит LINQ.

Щоб задіяти LINQ to SQL в проєкті, нам треба додати бібліотеку System.Data.Linq.dll. Для цього нам потрібно в оглядачі рішень натиснути правою кнопкою миші на гілці Посилання і вибрати з контекстного меню Додати посилання на System.Data.Linq (рис. 2.114).

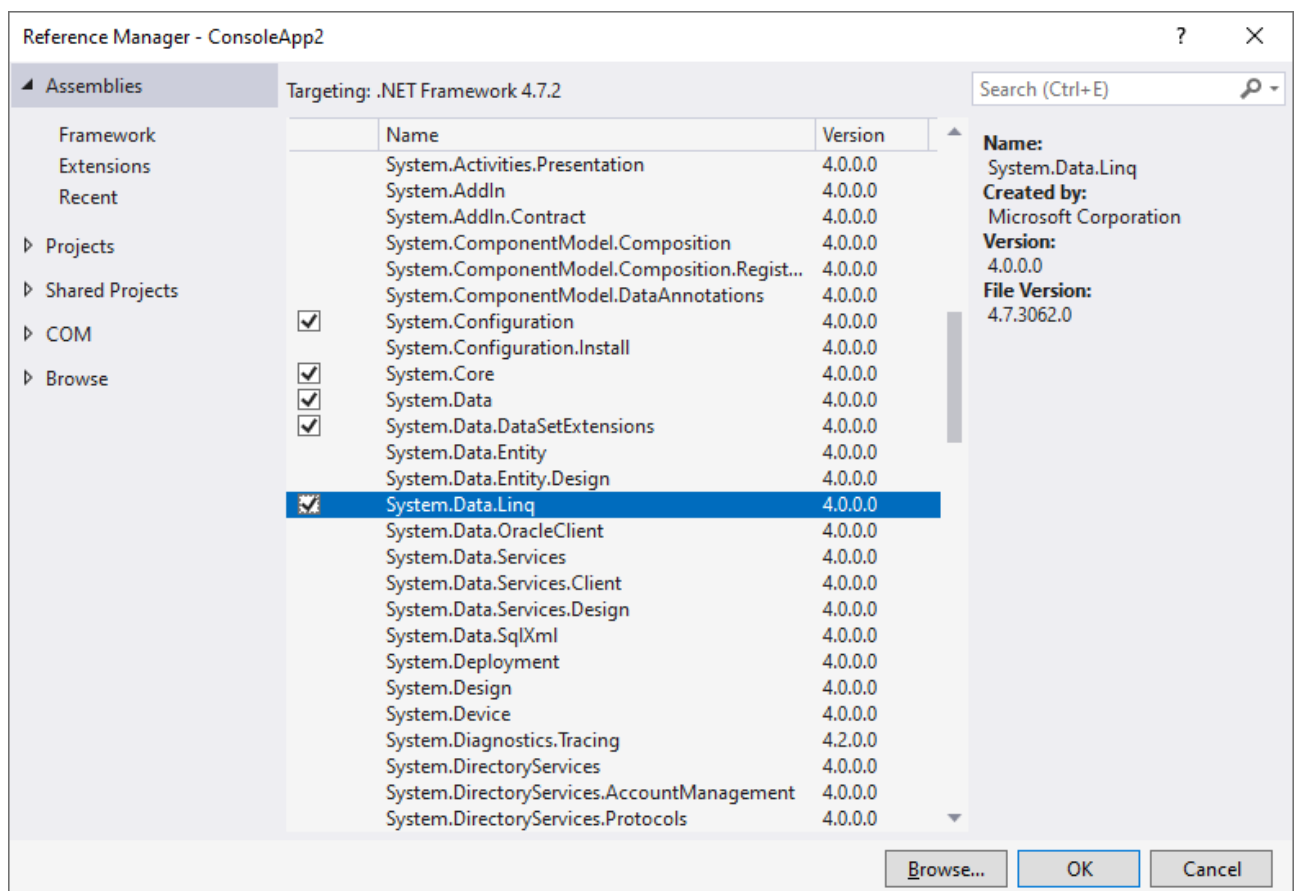


Рисунок 2.114 – Посилання на System.Data.Linq

Для взаємодії з базою даних в LINQ to SQL використовуються моделі і контекст даних. Моделі – це звичайні класи, які зіставляються з однією з таблиць в базі даних.

Нехай ми маємо базу check, в якій визначена таблиця goods (рис. 2.115)

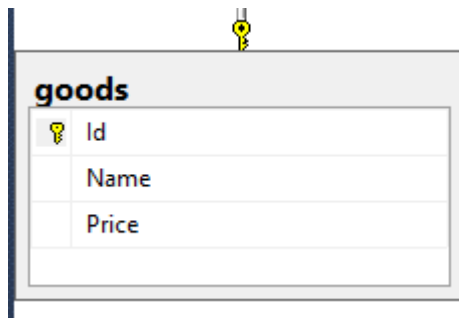


Рисунок 2.115 – Таблиця goods на діаграмі бази даних check

Спочатку визначимо сутність для роботи з даними цієї таблиці:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Linq.Mapping;

namespace ConsoleApp20
{
    [Table(Name = "goods")]
    Ссылка: 2
    public class goods
    {
        [Column(IsPrimaryKey = true, IsDbGenerated = true)]
        ссылка: 1
        public int Id { get; set; }
        [Column(Name = "Name")]
        ссылка: 1
        public string Name { get; set; }

        [Column]
        ссылка: 1
        public float Price { get; set; }
    }
}
```

Атрибут [Table] дає змогу виконати зіставлення таблиці з базою даних з цією моделлю. Однак за замовчуванням модель повинна називатися так само, як і таблиця, тобто goods. Тому, щоб зіставлення пройшло успішно, в атрибуті Table вказується властивість Name з ім'ям таблиці.

Кожна властивість моделі відповідає стовпцю за типом даних. Щоб відбувалося автоматичне співставлення, над властивостями застосовується атрибут [Column]. Без цього атрибута система не зможе зв'язати стовпчик з таблицею з властивістю.

Даний атрибут має ряд властивостей, за допомогою якого можна налаштувати зіставлення стовпців:

AutoSync – вказує, як треба витягувати значення стовпця після вставки або поновлення;

CanBeNull – вказує, чи може стовпець приймати значення null.

DbType – визначає тип стовпчика. Вказується, якщо треба створити нову базу даних;

Expression – зберігає вираз, який буде використовуватися для обчислення значення властивості;

IsPrimaryKey – зберігає логічне значення і вказує, чи виконує стовпець роль первинного ключа (як в даному випадку Id);

IsDbGenerated – зберігає логічне значення, яке вказує, чи буде значення стовпця генеруватися самою БД;

IsDiscriminator – вказує, чи буде стовпець розмежувачем в системі успадкування класів;

IsVersion – вказує, чи буде стовпець зберігати номер версії рядка або значення timestamp, яке вказує на час останньої зміни рядка;

Name – задає ім'я стовпця, з яким буде зіставлятися ця властивість;

Storage – вказує на ім'я приватної змінної, яка буде зберігати значення даного стовпця;

UpdateCheck – визначає, як LINQ to SQL буде вирішувати проблему паралелізму. Якщо в моделі немає властивостей зі значення IsVersion = true, то операції з даними БД буде порівнювати значення рядків з таблиці з новими значеннями.

У класі програми визначимо такий код:

```

ppzv ConsoleAppzv.Program main(string[] args)
using System;
using System.Collections.Generic;

using System.Data.SqlClient;
using System.Data.Linq;

namespace ConsoleApp20
{
    class Program
    {
        static string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=check; User Id =sa; Password =140317;";
        static void Main(string[] args)
        {
            DataContext db = new DataContext(connectionString);

            // Отримуюємо таблицю товари
            Table<goods> goods = db.GetTable<goods>();

            foreach (var good in goods)
            {
                Console.WriteLine("{0} \t{1} \t{2}", good.Id, good.Name, good.Price);
            }

            Console.Read();
        }
    }
}

```

В результаті запуску цієї програми у консоль буде виведено:

```

1      Пальто ж?н. синє      2500
2      Пальто ж?н. зимнє, дем?сизонне  4500
3      Пальто ж?н. чорне      12400
4      Пальто ун?секс. чорне  5000
5      Пальто чол. кл?тинка    12500
6      Пальто чол. чорне      6500
10     Ball      100

```

2.7.4 Фільтрація і сортування. Щоб застосувати фільтрацію і сортування в LINQ to SQL, скористаємося стандартними операторами і виразами LINQ. Наприклад, виведемо всі товари, у яких ціна більша ніж 100 грн. і впорядкуємо їх за назвою:


```

using System;
using System.Collections.Generic;

using System.Data.SqlClient;
using System.Data.Linq;
using System.Linq;

namespace ConsoleApp20
{
    class Program
    {
        static string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=check; User Id =sa; Password =140317;";
        static void Main(string[] args)
        {
            DataContext db = new DataContext(connectionString);

            var query = db.GetTable<goods>().Where(u => u.Price > 100).OrderBy(u => u.Name);
            foreach (var good in query)
            {
                Console.WriteLine("{0} \t{1} \t{2}", good.Id, good.Name, good.Price);
            }

            Console.Read();
        }
    }
}

```

Результат виконання програми:

```

2      Пальто ж?н. зимнє, дем?сизонне  4500
1      Пальто ж?н. синє                2500
3      Пальто ж?н. чорне               12400
4      Пальто ун?секс. чорне           5000
5      Пальто чол. кл?тинка            12500
6      Пальто чол. чорне               6500

```

2.7.5 Зміна об'єктів в LINQ to SQL. Під час отримання об'єктів з бази даних в контекст DataContext ці об'єкти кешуються і у них встановлюється стан Unchanged. Якщо змінимо значення властивостей якого-небудь об'єкта з отриманого набору, то DataContext для цього об'єкта створює копію зі зміненими значеннями і встановлює у неї статус ToBeUpdated.

При виклику методу SubmitChanges () контекст даних звіряє значення оригінального об'єкта і його зміненої копії. І якщо два об'єкти відрізняються, то створюється sql-вираз UPDATE, за допомогою якого відбувається оновлення об'єкта в базі даних.

Для прикладу отримаємо перший об'єкт з бази даних, змінимо його і зробимо збереження:

```
using System;
using System.Collections.Generic;

using System.Data.SqlClient;
using System.Data.Linq;
using System.Linq;

namespace ConsoleApp20
{
    class Program
    {
        static string connectionString = @"Data Source=.\\SQLEXPRESS;Initial Catalog=check; User Id =sa; Password =140317;";
        static void Main(string[] args)
        {
            DataContext db = new DataContext(connectionString);

            Console.WriteLine("До оновлення");
            foreach (var good in db.GetTable<goods>().Take(5))
            {
                Console.WriteLine("{0} \t{1} \t{2}", good.Id, good.Name, good.Price);
            }
            Console.WriteLine();

            // беремо перший товар
            goods good1 = db.GetTable<goods>().FirstOrDefault();
            // і змінимо його ціну
            good1.Price = 12321;
            // збережемо зміни
            db.SubmitChanges();

            Console.WriteLine();
            Console.WriteLine("Після оновлення");
            foreach (var good in db.GetTable<goods>().Take(5))
            {
                Console.WriteLine("{0} \t{1} \t{2}", good.Id, good.Name, good.Price);
            }

            Console.Read();
        }
    }
}
```

Результат виконання програми:

```
До оновлення
1      Пальто ж?н. синє          2500
2      Пальто ж?н. зимнє, дем?сизонне  4500
3      Пальто ж?н. чорне        12400
4      Пальто ун?секс. чорне     5000
5      Пальто чол. кл?тинка     12500

Після оновлення
1      Пальто ж?н. синє          12321
2      Пальто ж?н. зимнє, дем?сизонне  4500
3      Пальто ж?н. чорне        12400
4      Пальто ун?секс. чорне     5000
5      Пальто чол. кл?тинка     12500
```

Щоб додати новий об'єкт в базу даних, необхідно викликати у таблиці в контексті даних метод `InsertOnSubmit ()` або `InsertAllOnSubmit ()` і, за потреби, додати список об'єктів:

```

using System;
using System.Collections.Generic;
using System.Data.SqlClient;
using System.Data.Linq;
using System.Linq;

namespace ConsoleApp20
{
    class Program
    {
        static string connectionString = @"Data Source=.\SQLEXPRESS;Initial Catalog=check; User Id =sa; Password =140317;";
        static void Main(string[] args)
        {
            DataContext db = new DataContext(connectionString);

            Console.WriteLine("До додавання");
            foreach (var good in db.GetTable<goods>().Take(5))
            {
                Console.WriteLine("{0} \t{1} \t{2}", good.Id, good.Name, good.Price);
            }
            Console.WriteLine();

            // Створюємо новий товар
            goods good1 = new goods { Name = "Bag", Price = 1000 };
            // додаємо його в таблицю товарів
            db.GetTable<goods>().InsertOnSubmit(good1);
            db.SubmitChanges();

            Console.WriteLine();
            Console.WriteLine("Після додавання");
            foreach (var good in db.GetTable<goods>().OrderByDescending(u => u.Id).Take(6))
            {
                Console.WriteLine("{0} \t{1} \t{2}", good.Id, good.Name, good.Price);
            }

            Console.Read();
        }
    }
}

```

Результат виконання програми:

```

До додавання
1      Пальто ж?н. синє      12321
2      Пальто ж?н. зимнє, дем?сизонне  4500
3      Пальто ж?н. чорне      12400
4      Пальто ун?секс. чорне    5000
5      Пальто хол. кл?тинка    12500

П?сля додавання
12     Bag      1000
11     Bag      1000
10     Ball     100
6      Пальто хол. чорне      6500
5      Пальто хол. кл?тинка    12500
4      Пальто ун?секс. чорне    5000

```

Для видалення об'єкта з бази даних застосовується метод `DeleteOnSubmit()` або `DeleteAllOnSubmit()`, якщо видаляється список об'єктів (рис. 2.116)

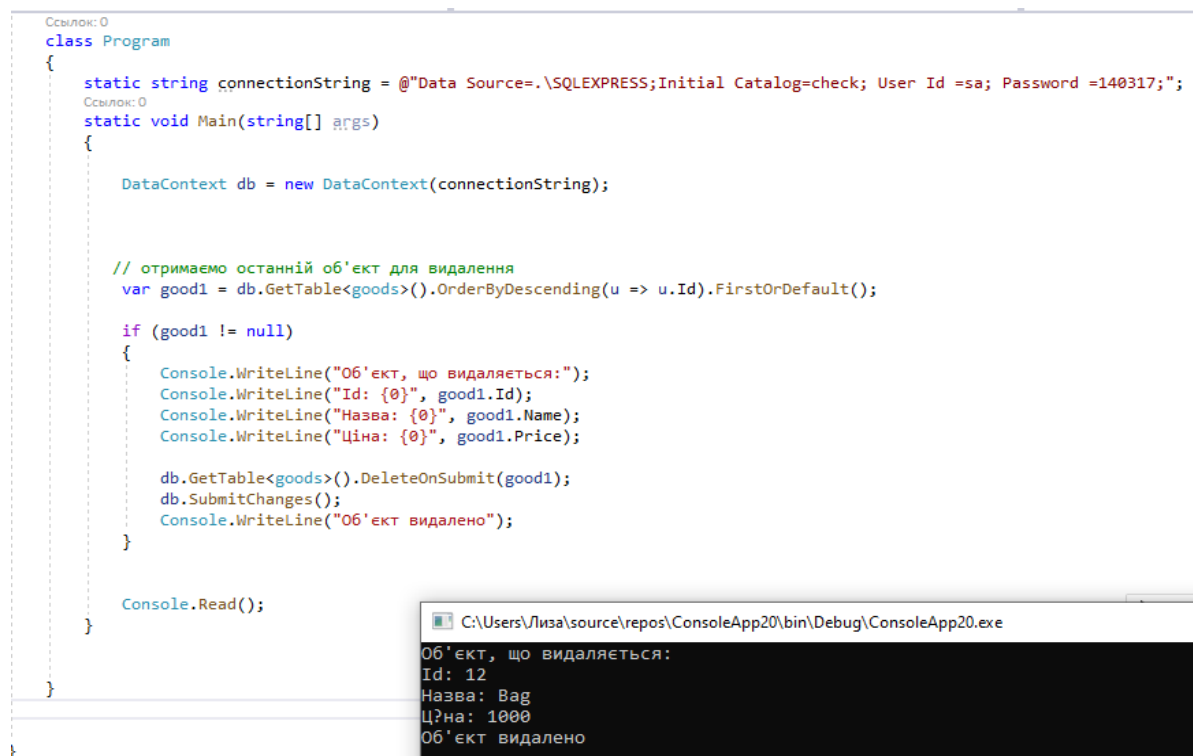


Рисунок 2.116 – Видалення об'єктів з таблиці goods

2.7.6 Entity Framework. Платформа Entity Framework.

Entity Framework – спеціальна об'єктно-орієнтована технологія на базі фреймворка .NET для роботи з даними. Якщо традиційні засоби ADO.NET дають змогу створювати підключення, команди та інші об'єкти для взаємодії з базами даних, то Entity Framework має більш високий рівень абстракції, який дає змогу абстрагуватися від самої бази даних і працювати з даними незалежно від типу сховища. Якщо на фізичному рівні оперують таблицями, індексами, первинними і зовнішніми ключами, то на концептуальному рівні, який пропонує Entity Framework, вже працюють з об'єктами.

Центральною концепцією Entity Framework є поняття сутності або entity. Сутність – це набір даних, асоційованих з певним об'єктом. Тому дана технологія передбачає роботу не з таблицями, а з об'єктами і їх наборами.

Будь-яка сутність, як і будь-який об'єкт з реального світу, має низку властивостей. Наприклад, якщо сутність описує людину, то можливо виділити такі властивості, як ім'я, прізвище, зріст, вік, вага. Властивості необов'язково є простими даними типу int, а й можуть представляти більш комплексні структури даних. І у кожній сутності може бути одна або декілька

властивостей, які будуть відрізняти цю сутність від інших і будуть унікально визначати цю сутність. Подібні властивості називають ключами.

Сутності можуть бути пов'язані асоціативним зв'язком один-до-багатьох, один-до-одного і багато-до-багатьох, подібно до того, як в реальній базі даних відбувається зв'язок через зовнішні ключі.

Відмінною рисою Entity Framework є використання запитів LINQ для вибірки даних з БД. За допомогою LINQ ми можемо не тільки отримувати певні рядки, що зберігають об'єкти, з БД, а й отримувати об'єкти, пов'язані різними асоціативними зв'язками.

Іншим ключовим поняттям є Entity Data Model. Ця модель зіставляє класи сутностей з реальними таблицями в БД.

Entity Data Model складається з трьох рівнів: концептуального, рівень сховища і рівень зіставлення (мапінга). На концептуальному рівні відбувається визначення класів сутностей, які використовуються в застосунку. Рівень сховища визначає таблиці, стовпці, відносини між таблицями і типи даних, з якими порівнюється використовувана база даних. Рівень зіставлення (мапінга) служить посередником між попередніми двома поняттями, визначаючи зіставлення між властивостями класу сутності і стовпцями таблиць.

Таким чином, через класи, визначені у застосунку, можливо взаємодіяти з таблицями бази даних.

Entity Framework передбачає три можливі способи взаємодії з БД:

Database first – Entity Framework створює набір класів, які відображають модель конкретної бази даних;

Model first – спочатку розробник створює модель бази даних, за якою потім Entity Framework створює саму базу даних на сервері;

Code first – розробник створює клас моделі даних, які будуть зберігатися в БД, а потім Entity Framework за цією моделлю генерує базу даних і її таблиці.

Щоб додати пакет EF6, натиснемо на проєкт в Explorer Solution правою кнопкою миші і виберемо в контекстному меню Manage NuGet Packages ... (рис. 2.117):

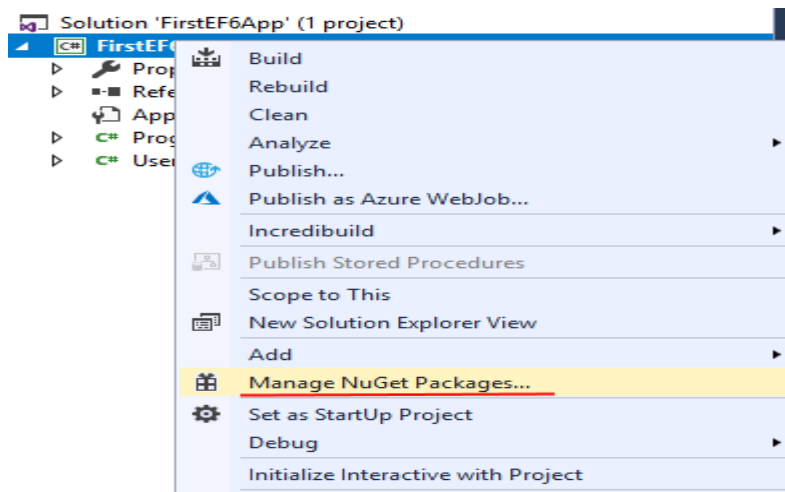


Рисунок 2.117 – Manage NuGet Packages

Потім у вікні управління NuGet-пакетами в вікні пошуку введемо слово «Entity» і виберемо пакет Entity Framework і встановимо його (рис. 2.118).

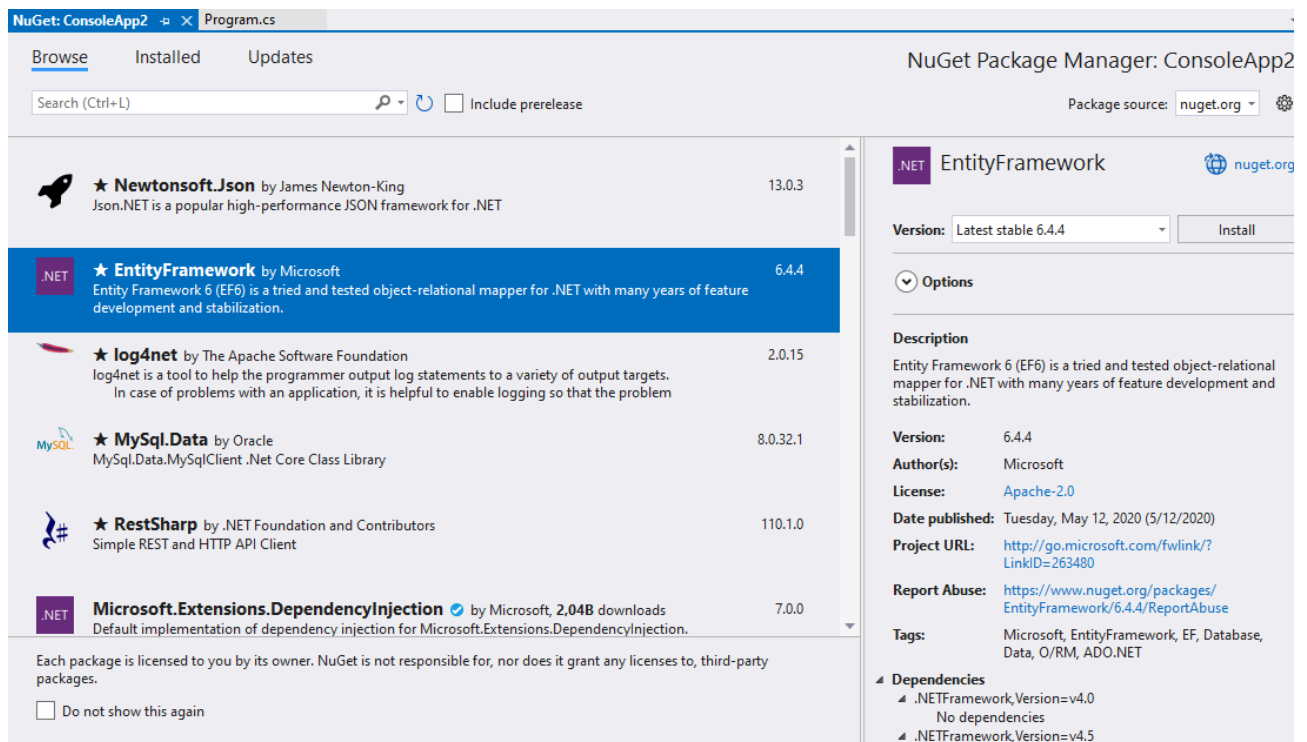


Рисунок 2.118 – Вікно управління NuGet-пакетами – встановлення Entity Framework

Основу функціональності Entity Framework складають класи, які знаходяться в просторі імен System.Data.Entity. Серед усього набору класів цього простору імен слід виділити такі:

DbContext – визначає контекст даних, який використовується для взаємодії з базою даних.

DbModelBuilder – зіставляє класи на мові C # з сутностями в базі даних.

DbSet / DbSet <TEntity> – представляє набір сутностей, які зберігаються в базі даних

У будь-якому застосунку, який працює з БД через Entity Framework, потрібен контекст (клас похідний від DbContext) і набір даних DbSet, через який можливо взаємодіяти з таблицями БД. В цьому разі таким контекстом буде клас goodsContext.

У конструкторі цього класу викликається конструктор базового класу, в який передається рядок «DbConnection» – це ім'я майбутнього рядка підключення до бази даних.

У Solution Explorer знайдемо файл App.config та відкриємо його. Перед закриттям тегом </configuration> додамо нову секцію connectionStrings:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit http://go.microsoft.com/fwlink/?LinkID=237468 -->
    <section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework, Version=6.0.0.0, Culture=neutral, Publi
  </configSections>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.7.2" />
  </startup>
  <entityFramework>
    <providers>
      <provider invariantName="System.Data.SqlClient" type="System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer" />
    </providers>
  </entityFramework>
  <connectionStrings>
    <add name="DefaultConnection" connectionString="Data Source=.\\SQLEXPRESS;Initial Catalog=check; User Id=sa; Password=140317;"
      providerName="System.Data.SqlClient" />
  </connectionStrings>
</configuration>
```

Визначимо класи моделі даних та контексту. Додамо клас моделі goods:

```
Ссылка: 2
public class goods
{
  Ссылка: 2
  public int Id { get; set; }
  Ссылка: 1
  public string Name { get; set; }
  Ссылка: 1
  public float Price { get; set; }
}
```

І також додамо клас контексту даних:

```

Ссылка: 3
public class goodsContext : DbContext
{
    ссылка: 1
    public goodsContext() :
        base("DefaultConnection")
    { }

    ссылка: 1
    public DbSet<goods> Goods { get; set; }
}

```

У конструкторі контексту даних передаємо конструктор базового класу ім'я рядка підключення з файлу конфігурації App.config.

Для отримання даних визначимо такий код:

```

Ссылка: 4
static void Main(string[] args)
{
    using (goodsContext db = new goodsContext())
    {
        var Goods1 = db.Goods;
        foreach (goods u in Goods1)
        {
            Console.WriteLine("{0}.{1} - {2}", u.Id, u.Name, u.Price);
        }
    }

    Console.Read();
}

```

В результаті запуску цієї програми у консоль буде виведено:

```

1.Пальто ж?н. синє - 12321
2.Пальто ж?н. зимнє, дем?сизонне - 4500
3.Пальто ж?н. чорне - 12400
4.Пальто ун?секс. чорне - 5000
5.Пальто чол. кл?тинка - 12500
6.Пальто чол. чорне - 6500
10.Ball - 100
11.Bag - 1000
13.Ball - 50

```

Для додавання застосовується метод Add() об'єкта DbSet:


```

Ссылка: 0
static void Main(string[] args)
{
    using (goodsContext db = new goodsContext())
    {
        goods Good1= new goods { Name = "Galaxy S7", Price = 5000 };
        goods Good2 = new goods { Name = "Honor 7A", Price = 2000 };
        // додавання
        db.Goods.Add(Good1);
        db.Goods.Add(Good2);
        db.SaveChanges(); //Збереження змін
        var Goods1 = db.Goods;
        foreach (goods u in Goods1)
        {
            Console.WriteLine("{0}.{1} - {2}", u.Id, u.Name, u.Price);
        }
    }

    Console.Read();
}

```

Результат виконання програми:

```

1.Пальто ж?н. синє - 12321
2.Пальто ж?н. зимнє, дем?сизонне - 4500
3.Пальто ж?н. чорне - 12400
4.Пальто ун?секс. чорне - 5000
5.Пальто чол. кл?тинка - 12500
6.Пальто чол. чорне - 6500
10.Ball - 100
11.Bag - 1000
13.Ball - 50
1013.Galaxy S7 - 5000
1014.Honor 7A - 2000

```

Контекст даних здатний відстежувати зміни об'єктів, тому для редагування об'єкта достатньо змінити його властивості і після цього викликати метод `SaveChanges()`:

```

Ссылка: 0
static void Main(string[] args)
{
    using (goodsContext db = new goodsContext())
    {
        // отримуємо перший об'єкт
        goods F = db.Goods.Find(1);

        F.Price = 3000;
        db.SaveChanges(); // Зберігаємо зміни

        var Goods1 = db.Goods;
        foreach (goods u in Goods1)
        {
            Console.WriteLine("{0}.{1} - {2}", u.Id, u.Name, u.Price);
        }
    }

    Console.Read();
}

```

Результат виконання консольної програми:

```

1.Пальто ж?н. синє - 3000
2.Пальто ж?н. зимнє, дем?сизонне - 4500
3.Пальто ж?н. чорне - 12400
4.Пальто ун?секс. чорне - 5000
5.Пальто чол. кл?тинка - 12500
6.Пальто чол. чорне - 6500
10.Ball - 100
11.Bag - 1000
13.Ball - 50
1013.Galaxy S7 - 5000
1014.Honor 7A - 2000

```

Для видалення об'єкта застосовується метод Remove() об'єкта DbSet.

Видалимо запис, який містить значення «1014» з таблиці goods (рис. 2.119):

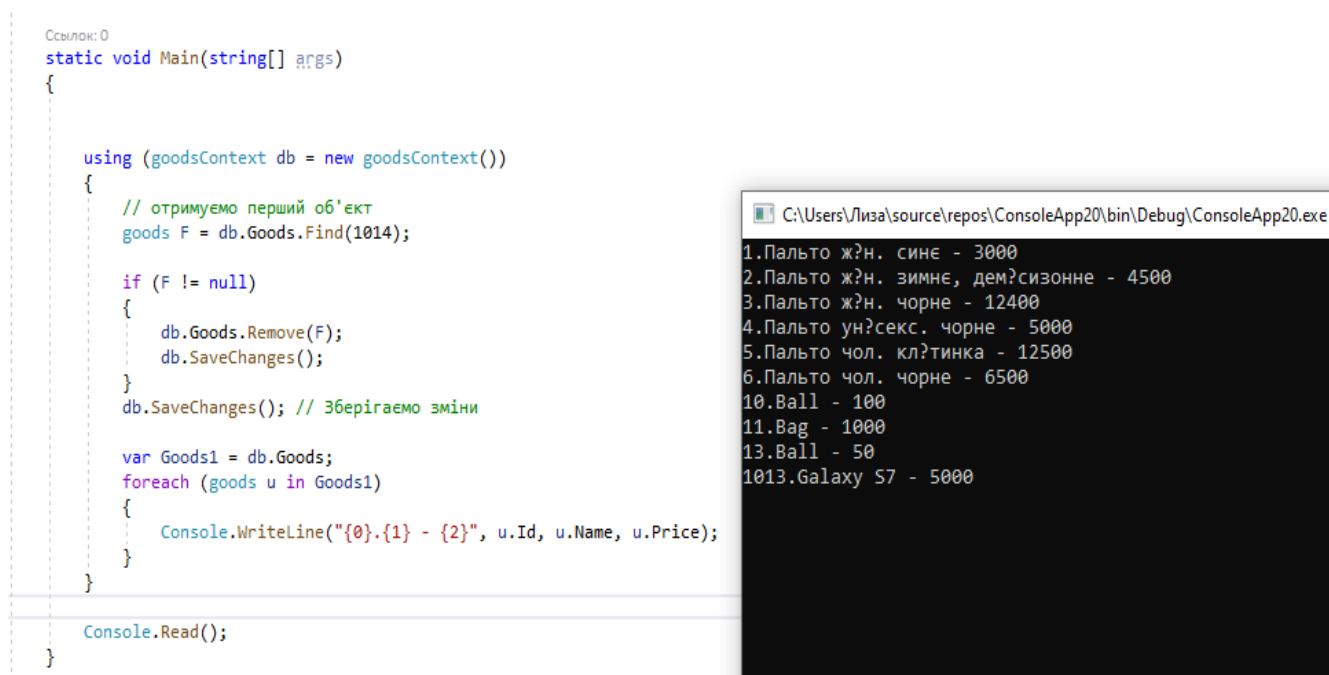


Рисунок 2.119 – Видалення запису з таблиці goods

Питання для самоперевірки

Технологія ADO.NET, недоліки і переваги.

Технологія LINQ, недоліки і переваги.

Технологія Entity Framework, недоліки і переваги.

Завдання

Варіант	Опис завдання
---------	---------------

1	Розробити програмний застосунок для своєї БД, використавши ADO.NET. Розробити зручний інтерфейс. Функціонал застосунку: Додавання/редагування/перегляд факультетів. Додавання/редагування/перегляд кафедр по факультетам. Додавання/редагування/перегляд груп по кафедрам. Додавання/редагування/перегляд викладачів по кафедрам. Додавання/редагування/перегляд студентів по групам. Додавання/редагування/перегляд дисциплін по кафедрам. Перегляд журналу успішності по дисципліні ля студентів із певних груп (має бути вибір: дисципліни і групи).
2	Розробити програмний застосунок для своєї БД для ОС Windows. Розробити зручний інтерфейс. Функціонал застосунку: Додавання/редагування/перегляд всіх таблиць. Перегляд таблиці продажу товару за певний період з виведенням загальної суми продажу (для всіх виробників і по обраному виробнику).
3	Розробити програмний застосунок для своєї БД, використавши LINQ. Розробити зручний інтерфейс. Функціонал застосунку: Додавання/редагування/видалення/перегляд всіх таблиць. Перегляд інформації про місце розташування певного військового, роти, частини.
4	Розробити програмний застосунок для своєї БД, використавши EF. Розробити зручний інтерфейс. Функціонал застосунку: Додавання/редагування/перегляд всіх таблиць. Перегляд інформації по бібліотекам та їх фондам. Виведення сумарної інформації. Перегляд інформації по бібліотекам та їх співробітникам.
5	Розробити програмний застосунок для своєї БД, використавши EF. Розробити зручний інтерфейс. Функціонал застосунку:

	Додавання/редагування/перегляд всіх таблиць. Виведення інформації про пансіонати по містах та їх прибутки. Виведення інформації про найбільш популярні місця для туризму
--	---

Рекомендована література до розділу [1-22].

3. ДОКУМЕНТООРІЄНТОВАНІ БАЗИ ДАНИХ

3.1 Вступ до MongoDB та робота з документами

MongoDB є документо-орієнтованою системою управління базами даних, яка використовує формат BSON для зберігання даних. Вона пропонує гнучку модель даних, що дає змогу зберігати документи з різною структурою в одній колекції. Відсутність жорстких схем робить MongoDB ідеальним вибором для зберігання даних, що швидко змінюються і мають складну ієрархічну структуру. Висока продуктивність і масштабованість дозволяють використовувати MongoDB для створення великих розподілених систем. Завдяки своїм можливостям, MongoDB широко застосовується у веб-розробці, аналітиці даних та інших сферах, де потрібна швидка і гнучка система управління базами даних.

MongoDB, розроблена на мові C++. Розробка MongoDB почалася у 2007 році в компанії 10gen (пізніше перейменованій на MongoDB, Inc.). Засновники компанії, натхненні обмеженнями традиційних реляційних СУБД, поставили собі за мету створити систему, яка б дозволяла зберігати дані у більш природному для них форматі – у вигляді документів.

На відміну від реляційних баз даних, MongoDB не вимагає строгої, заздалегідь визначеної схеми. Дані зберігаються у вигляді документів, що дає змогу зберігати інформацію різної структури в одній колекції. MongoDB забезпечує високу швидкість запитів завдяки використанню власного формату зберігання даних BSON, який оптимізований для ефективного пошуку та оновлення даних. Система підтримує горизонтальне масштабування, що надає можливість збільшувати обчислювальні ресурси за рахунок додавання нових вузлів.

MongoDB спроможна ефективно обробляти великі обсяги даних за умов високих навантажень, забезпечуючи при цьому високу доступність інформації завдяки вбудованому механізму реплікації. Ця технологія дає змогу створювати

репліки даних на різних серверах, що суттєво підвищує надійність системи та мінімізує ризики втрати інформації у випадку виходу з ладу окремих вузлів кластера. MongoDB підтримує широкий спектр операцій пошуку, включаючи пошук за повним текстом, геопросторові запити та агрегацію даних. Вона розроблена з урахуванням вимог розподілених систем, що дає можливість розподіляти дані по множинним серверам, забезпечуючи високу доступність і масштабованість. Відмітимо, що вона є кросплатформною системою, яка підтримує всі основні операційні системи, та доступною як хмарна служба, що спрощує її розгортання та управління.

3.1.1 Встановлення та початок роботи з MongoDB. Для встановлення MongoDB потрібно завантажити пакет, що розповсюджується з офіційного сайту <https://www.mongodb.com/try/download/community> (рис. 3.1).

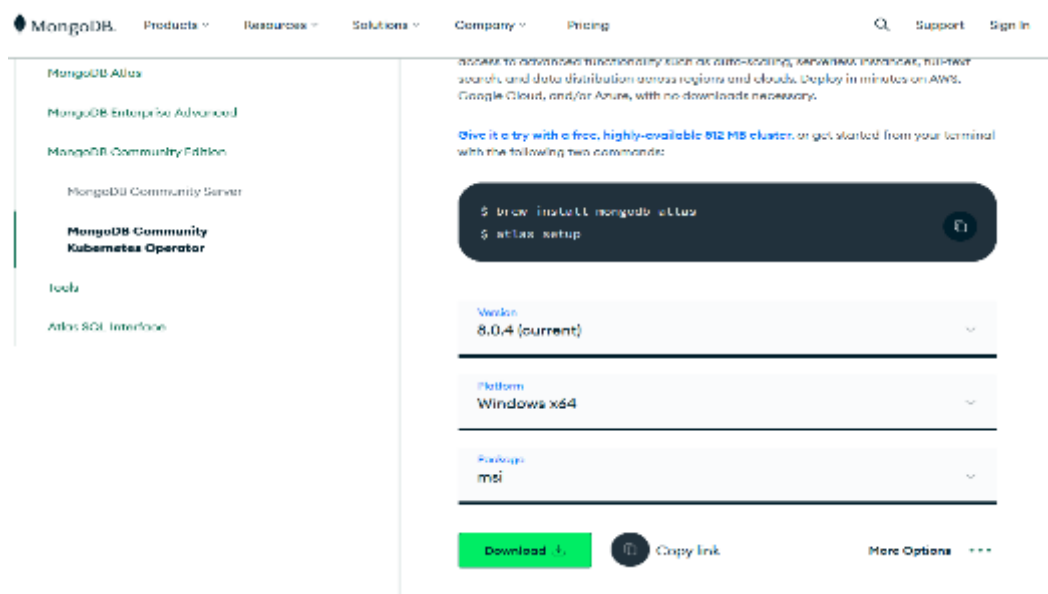


Рисунок 3.1 – Офіційний сайт MongoDB

Офіційний сайт надає пакети дистрибутивів для різних платформ: Windows, Linux, MacOS, Solaris. Для кожної платформи є кілька дистрибутивів. Причому є два види серверів – Community та Enterprise. У цьому випадку необхідно встановити версію Community. Хоча Enterprise-версія має дещо

більші можливості, але вона доступна тільки в тріальному режимі або за підпискою.

На момент написання даного матеріалу останньою версією платформи була версія **8.4**, що побачила світ у грудні 2024 року і для якої постійно виходять підверсії. Використання конкретної версії може дещо відрізнятися від інших версій платформи MongoDB.

Для завантаження необхідного функціоналу потрібно обрати потрібну операційну систему та відповідний тип пакета.

Середовище MongoDB пропонує користувачам декілька опцій інсталяції. Для платформи Windows доступні як самовстановлювальні пакети MSI, так і архіви ZIP. З метою більшої гнучкості та контролю над процесом інсталяції, доцільно обрати варіант із ZIP-архівом. Після завантаження архів розпаковується у вказану директорію, що дає змогу користувачу налаштувати середовище відповідно до своїх потреб.

Після завершення інсталяції MongoDB шляхом розпакування ZIP-архіву у вказану директорію (наприклад, C:\mongodb\bin), користувач отримує доступ до набору утиліт, необхідних для взаємодії з базою даних (рис. 3.2). Ця директорія містить різноманітні консольні інструменти, призначені для запуску сервера MongoDB, управління базами даних, виконання запитів та інших операцій. Важливо зазначити, що більшість сучасних дистрибутивів MongoDB оптимізовані для 64-бітних операційних систем. У випадку використання 32-бітної системи може знадобитися додаткова конфігурація або пошук альтернативних рішень.

windows-x86_64-8.0.4 > mongodb-win32-x86_64-windows-8.0.4 > bin			
Ім'я	Дата змінення	Тип	Розмір
 Install-Compass.ps1	27.11.2024 21:28	Windows PowerS...	2 КБ
 mongod.exe	27.11.2024 22:54	Застосунок	73 732 КБ
 mongod.pdb	27.11.2024 22:54	Program Debug D...	1 202 884 КБ
 mongos.exe	27.11.2024 22:53	Застосунок	46 953 КБ
 mongos.pdb	27.11.2024 22:53	Program Debug D...	783 500 КБ
 vc_redist.x64.exe	03.07.2024 18:18	Застосунок	24 722 КБ

Рисунок 3.2- Вміст робочого каталогу MongoDB

До основних виконуючих файлів відносяться `mongosh`, `mongod`, `mongos`. `Mongosh` – це клієнтська програма (CLI), яка надає користувачеві інтерфейс для взаємодії з сервером MongoDB. Вона дає змогу виконувати адміністративні завдання, такі як створення користувачів, управління правами доступу, а також виконувати операції з даними. Ця програма для деяких версій потребує окреме встановлення (<https://www.mongodb.com/docs/mongodb-shell/install/>). `Mongod` – це сервер, який відповідає за зберігання даних на диску, обробку запитів від клієнтів та забезпечення високої доступності даних. `Mongod` є основним компонентом будь-якого екземпляра MongoDB. `Mongos` – це маршрутизатор запитів, який використовується в розподілених кластерах MongoDB для забезпечення масштабованості та високої доступності, а також приймає запити від клієнтів і перенаправляє їх на відповідні шарди, що дає змогу розподілити навантаження на кілька серверів.

Шарди (shards) в MongoDB становлять фундаментальну концепцію горизонтального масштабування баз даних. За умов зростання обсягів даних до рівня, що перевищує можливості одного сервера, MongoDB забезпечує механізм розподілу інформаційних масивів між кількома серверами (шардами). Така архітектура суттєво збільшує пропускну здатність системи, оптимізує продуктивність операцій з даними та підвищує загальну доступність і надійність інформаційної інфраструктури.

Ініціалізація сервера MongoDB передбачає попередню конфігурацію директорії для зберігання даних. За замовчуванням система використовує стандартні шляхи: `C:\data\db` для Windows та `/data/db` для Linux/macOS. Однак, користувач має можливість вказати власний шлях за допомогою параметра – `dbpath`. Після створення директорії та налаштування параметрів здійснюється запуск демона `mongod`, який відповідає за обробку запитів до бази даних. Для ОС Windows цей процес виглядатиме як показано на рис. 3.3:


```

Microsoft Windows [Version 10.0.19043.1415]
(c) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.

C:\WINDOWS\system32>cd..

C:\Windows>cd..

C:\>cd C:\mongodb-win32-i386-2.6.0\bin

C:\mongodb-win32-i386-2.6.0\bin>mongod
mongod --help for help and startup options
021-12-30T19:26:45.390+0200
021-12-30T19:26:45.390+0200 warning: 32-bit servers don't have journaling enabled by default. Please use --journal if
you want durability.
021-12-30T19:26:45.390+0200
021-12-30T19:26:45.402+0200 [initandlisten] MongoDB starting : pid=768 port=27017 dbpath=\data\db\ 32-bit host=DESKT
A2MEC1
021-12-30T19:26:45.402+0200 [initandlisten]
021-12-30T19:26:45.402+0200 [initandlisten] ** NOTE: This is a 32 bit MongoDB binary.
021-12-30T19:26:45.402+0200 [initandlisten] **      32 bit builds are limited to less than 2GB of data (or less with
journal).
021-12-30T19:26:45.402+0200 [initandlisten] **      Note that journaling defaults to off for 32 bit and is currentl
f.
021-12-30T19:26:45.402+0200 [initandlisten] **      See http://dochub.mongodb.org/core/32bit
021-12-30T19:26:45.402+0200 [initandlisten]
021-12-30T19:26:45.402+0200 [initandlisten] targetMinOS: Windows XP SP3
021-12-30T19:26:45.402+0200 [initandlisten] db version v2.6.0
021-12-30T19:26:45.402+0200 [initandlisten] git version: 1c1c76aeca21c5983dc178920f5052c298db616c
021-12-30T19:26:45.402+0200 [initandlisten] build info: windows sys.getwindowsversion(major=6, minor=0, build=6002,
form=7, service_pack='Service Pack 2') BOOST_LIB_VERSION=1_49

```

Рисунок 3.3 – Запуск сервера mongod

Командний рядок відобразить ряд службової інформації, наприклад, що сервер запускається на localhost через порт 27017.

Наступним кроком після запуску сервера MongoDB є підключення до нього за допомогою інтерактивного інтерфейсу mongosh. Цей інтерфейс надається у вигляді консольної програми mongo.exe. Запуск mongo дає змогу користувачеві безпосередньо взаємодіяти з базою даних, виконувати різноманітні операції та аналізувати дані. Процес запуску цього файлу подано на рис. 3.4

```

Please enter a MongoDB connection string (Default: mongodb://localhost/): mongodb://localhost/
mongodb://localhost/
Current Mongosh Log ID: 6788eb4756fe6f10899dc84b
Connecting to:      mongodb://localhost/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+1.10.6
Using MongoDB:      8.0.4
Using Mongosh:       1.10.6
Mongosh 2.3.8 is available for download: https://www.mongodb.com/try/download/shell
For mongosh info see: https://docs.mongodb.com/mongodb-shell/

-----
The server generated these startup warnings when booting
2025-01-15T23:24:21.250+02:00: Access control is not enabled for the database. Read and write access to data and conf
figuration is unrestricted
-----
test>

```

Рисунок 3.4 – Запуск клієнта mongoDB

Тепер зробимо найпростіші дії. Введемо в mongosh послідовно наступні команди і після кожної команди натиснемо на Enter:

```
test> use testDB
switched to db testDB
testDB> db.Authors.insertOne({Name:"Сергій Жадан"})
{
  acknowledged: true,
  insertedId: ObjectId("6788eea356fe6f10899dc84d")
}
testDB> db.Authors.find()
[
  { _id: ObjectId("6788ede956fe6f10899dc84c"), Name: 'Сергій Жадан' },
  { _id: ObjectId("6788eea356fe6f10899dc84d"), Name: 'Сергій Жадан' }
]
testDB>
```

Перша команда `use testDB` встановлює як використовувану базу даних `testDB`. Навіть якщо такої бази даних немає, вона створюється автоматично. І далі `db` представлятиме поточну базу даних – тобто базу даних `testDB`. Після `db` йде `Authors`, це колекція, в яку потім ми додаємо новий об'єкт. Якщо в SQL нам треба створювати таблиці заздалегідь, то колекції MongoDB створює самостійно за їх відсутності.

За допомогою методу `db.Authors.insertOne()` у колекцію `Authors` бази даних `testDB` додається об'єкт `{Name: «Сергій Жадан»}`. Опис об'єкта, що додається, визначається у форматі, з яким ви можливо знайомі, якщо мали справу з форматом JSON. Якщо об'єкт був успішно доданий, консоль виведе результат операції, зокрема, ідентифікатор доданого об'єкта. А третя команда `db.Authors.find()` виводить на екран усі об'єкти із бази даних `testDB`.

3.1.2 Організація бази даних. Якщо у реляційних базах даних вміст складають таблиці, то `mongodb` база даних складається з колекцій. Кожна колекція має своє унікальне ім'я – довільний ідентифікатор, що складається з не більш ніж 128 різних алфавітно-цифрових символів та підкреслення. На відміну від реляційних баз даних, MongoDB не використовує табличне впорядкування із чітко заданою кількістю стовпців та типів даних. MongoDB є документо-орієнтованою системою, в якій центральним поняттям є документ.

Документ можна подати як об'єкт, який зберігає певну інформацію. У певному сенсі він подібний до рядків у реляційних СУБД, де рядки зберігають інформацію про окремий елемент. Наприклад, типовий документ:

```
{
  "ім'я": "Петро"
  "прізвище": "Петренко"
  "вік": "20"
  "освіта": "Ліцей Ерудит"
}
```

Документ представляє набір пар ключ-значення. Наприклад, у виразі «ім'я»: «Петро» «ім'я» представляє ключ, а «Петро» – значення.

Ключі представляють рядки. Значення можуть відрізнятися за типом даних. У цьому випадку майже всі значення також представляють рядковий тип. Усього є 15 типів значень (таблиця 3.1).

Таблиця 3.1. Типи даних в MongoDB

String	рядковий тип даних, як у наведеному вище прикладі (для рядків використовується кодування UTF-8)
Array (масив)	тип даних для зберігання масивів елементів
Binary data (двійкові дані)	тип зберігання даних у бінарному форматі
Boolean	булевий тип даних, що зберігає логічні значення TRUE або FALSE
Date	зберігає дату у форматі часу Unix
Double	числовий тип даних для зберігання чисел з плаваючою точкою
Integer	використовується для зберігання цілих значень розміром 32 біти
Long	використовується для зберігання цілих значень розміром 64 біти
JavaScript	тип даних для зберігання коду javascript

Min key/Max key	використовуються для порівняння значень із найменшим/найбільшим елементів BSON
Null	тип даних для зберігання Null
Object	рядковий тип даних
ObjectId	тип даних для зберігання id документа
Regular expression	застосовується для зберігання регулярних виразів
Decimal128	тип даних для зберігання десяткових дробових чисел розміром 128 біт, які дозволяють вирішити проблеми з проблемою точності обчислень при використанні дробових чисел, які представляють тип Double
Timestamp	застосовується для зберігання часу

На відміну від рядків в таблицях реляційних баз даних, документи можуть містити інформацію різного змісту. Так, поруч із документом, описаним вище, в одній колекції може бути інший об'єкт, наприклад:

```
{
  "ім'я": "Іван"
  "прізвище": "Іванченко"
  "вік": "40"
  "освіта": [ "Ліцей Розумник",
              "КНУ імені Тараса Шевченка" ]
}
```

В MongoDB запити мають реєстрозалежність і строгу типізацію. Тобто наступні два документи не будуть ідентичними:

```
"вік": "40"  
"вік": 40
```

Якщо в першому випадку для ключа “вік” визначено значенням рядка, то в другому випадку значенням є число.

3.1.3 Ідентифікатор документа. Для кожного документа MongoDB визначено унікальний ідентифікатор, який називається `_id`. При додаванні документа до колекції цей ідентифікатор створюється автоматично. Однак розробник може сам явно задати ідентифікатор, а не покладатися на автоматично генеровані, вказавши відповідний ключ і його значення в документі.

Це поле має мати унікальне значення в межах колекції. І якщо ми спробуємо додати до колекції два документи з однаковим ідентифікатором, то буде додано лише один із них, а при додаванні другого ми отримаємо помилку.

Якщо ідентифікатор не заданий явно, MongoDB створює спеціальне бінарне значення розміром 12 байт. Це значення складається з декількох сегментів: значення типу timestamp розміром 4 байта (що представляє кількість секунд з моменту початку епохи Unix), випадкове число з 5 байт і лічильник з 3 байт, який ініціалізований випадковим числом. Така модель побудови ідентифікатора гарантує з високою ймовірністю, що він матиме унікальне значення.

3.1.4 Адміністрування mongodb. Для початку роботи з системою MongoDB необхідно запустити консольну оболонку `mongosh.exe` та ввести команду `use test`. Ця дія активує базу даних з іменем «test». Перелік доступних баз даних можна отримати за допомогою команди `show dbs`:

```
testDB> show dbs  
admin    40.00 KiB  
config   48.00 KiB  
local    40.00 KiB  
testDB   72.00 KiB  
testDB>
```

Імена баз даних мають певні обмеження: довжина не повинна перевищувати 64 байти, а також заборонено використовувати ряд спеціальних

символів. Наприклад, в імені не повинно бути символів /, \, ., «, *, <, >, :, |, ?, \$. Існують зарезервовані бази даних (local, admin, config), які використовуються системою для внутрішніх потреб.

Щоб перевірити, яка база даних активна в даний момент, слід виконати команду db:

```
> use test
switched to db test
> db
test
>
```

Окрім баз даних, ми можемо переглянути список усіх колекцій у поточній бд за допомогою команди show collections:

```
testDB> show collections
Authors
testDB>
```

Команда db.stats() надає можливість отримати статистику поточної бази даних. Наприклад, у нас як поточна встановлена база даних testDB:

```
testDB> db.stats()
{
  db: 'testDB',
  collections: Long("1"),
  views: Long("0"),
  objects: Long("2"),
  avgObjSize: 56,
  dataSize: 112,
  storageSize: 36864,
  indexes: Long("1"),
  indexSize: 36864,
  totalSize: 73728,
  scaleFactor: Long("1"),
  fsUsedSize: 73192173568,
  fsTotalSize: 895346536448,
  ok: 1
}
```

3.1.5 JSON-файл та BSON. Нотація об'єктів JavaScript (JSON) була запропонована Дугласом Крокфордом як спрощений підмножина мови JavaScript на початку 2000-х років. Офіційна специфікація JSON була стандартизована лише у 2013 році. Об'єкти JavaScript є універсальним механізмом для представлення даних у вигляді пар «ключ-значення», де ключами виступають рядки, а значеннями можуть бути різноманітні типи даних, включаючи інші об'єкти, що дає змогу створювати складні ієрархічні структури. Простота синтаксису JSON та незалежність від мови програмування

сприяли його широкому використанню для обміну даними між різними системами:

```
{
  "lemma": "велорикш",
  "end": [
    "a",
    "i",
    "i",
    "i",
    "ам",
    "у",
    "ею",
    "ами",
    "і",
    "ах",
    "е",
    "і"
  ]
},
{
  "lemma": "вельмож",
  "end": [
    "a",
    "i",
    "i",
    "i",
    "ам",
    "у",
    "ею",
    "ами",
    "і",
    "ах",
    "е",
    "і"
  ]
},
{
```

Поширення JavaScript у сфері веб-розробки безпосередньо вплинуло на зростання популярності формату JSON. Його легкість читання як для людини, так і для комп'ютерних систем, а також відносна простота реалізації підтримки

в різних мовах програмування сприяли його виходу за рамки веб-розробки і перетворенню на універсальний формат обміну даними. JSON є універсальним форматом обміну даними, який широко використовується в розробці програмного забезпечення. Його застосування охоплює такі області, як створення інтерфейсів програмування додатків (API), конфігурування налаштувань програм, зберігання логів подій та організацію структурованих даних у базах даних.

MongoDB була розроблена з самого початку як ідеальна платформа даних для розробки сучасних додатків. Широке використання JSON зробила його очевидним вибором для представлення структур даних в інноваційній моделі даних документів MongoDB. Однак є кілька проблем, які роблять JSON менш ідеальним для використання всередині бази даних.

Одним з недоліків формату JSON є його текстова природа, що призводить до відносно високої обчислювальної складності при парсингу. Крім того, обмежений набір підтримуваних типів даних може бути недостатнім для представлення складних структур даних, що виникають у деяких предметних областях.

Для подолання обмежень, пов'язаних з текстовим представленням даних у форматі JSON, був розроблений бінарний формат BSON, який зберігає сумісність зі структурою JSON, але використовує двійкове кодування, що значно прискорює процеси серіалізації та десеріалізації даних, особливо при роботі з великими обсягами інформації. Двійкова структура BSON кодує інформацію про тип і довжину, що дає змогу аналізувати її набагато швидше. З моменту свого початкового формулювання BSON було розширено, щоб додати деякі додаткові типи даних, які не належать до JSON, як-от дати та двійкові дані, без яких MongoDB не мав би цінної підтримки.

MongoDB зберігає дані у форматі BSON як всередині, так і через мережу, але це не означає, що ви не можете вважати MongoDB базою даних JSON. Все, що ви можете представити в JSON, можна зберігати в MongoDB і так само легко отримати в JSON.

Нижче наведено деякі приклади документів (у синтаксисі стилю JavaScript/Python) та відповідні їм представлення BSON:

```

{"hello": "world"} → \x16\x00\x00\x00 // total document size
                    \x02 // 0x02 = type String
                    hello\x00 // field name
                    \x06\x00\x00\x00world\x00 // field value
                    \x00 // 0x00 = type E00 ('end of object')

{"BSON": ["awesome", 5.05, 1986]} → \x31\x00\x00\x00
                                   \x04BSON\x00
                                   \x26\x00\x00\x00
                                   \x02\x30\x00\x08\x00\x00\x00awesome\x00
                                   \x01\x31\x00\x33\x33\x33\x33\x33\x14\x40
                                   \x10\x32\x00\xc2\x07\x00\x00
                                   \x00
                                   \x00
```

3.1.6 Створення колекцій. Процес створення колекцій не обмежується автоматичною ініціалізацією. Метод `db.createCollection(name, options)` надає більшу гнучкість, дозволяючи явним чином визначити структуру та налаштування нової колекції за допомогою параметрів `name` та `options`. Наприклад:

```
test> db.createCollection("account")
{ ok: 1 }
```

У процесі роботи, можливо, потрібно змінити назву колекції. Наприклад, якщо при першому додаванні даних у назві була помилка. І щоб не видаляти та потім перетворювати колекцію, слід використовувати функцію `renameCollection`:

```
test> db.account.renameCollection("user")
{ ok: 1 }
```

Коли ми надсилаємо запит до бази даних на вибірку, MongoDB повертає нам документи в тому порядку, як правило, в якому вони були додані. Однак такий порядок не завжди гарантується, оскільки дані можуть бути вилучені, переміщені, змінені. Тому MongoDB існує поняття обмеженої колекції (*sapped collection*). Подібна колекція гарантує, що документи розташовуватимуться в тому ж порядку, в якому вони додавалися до колекції. Обмежені колекції мають

фіксований розмір. І коли в колекції вже немає місця, найстаріші документи видаляються, і наприкінці додаються нові дані.

На відміну від звичайних колекцій, обмежені ми можемо створити тільки явно. Наприклад, створимо обмежену колекцію з назвою Work і задаймо для неї розмір 9500 байт:

```
test> db.createCollection("Work",{capped:true, size:9500})
```

Також можна обмежити кількість документів у колекції, вказавши його у параметрі max:

```
test> db.createCollection("Work",{capped:true, size:9500, max:1000})  
{ ok: 1 }
```

Однак при такому способі створення колекції слід враховувати, що якщо все місце під колекцією заповнено (наприклад, виділені нами 9500 байтів), а кількість документів ще не досягла максимуму, в даному випадку 1000, то при додаванні нового документа найстаріший документ буде видалятися, а на його місце вставлятиметься новий документ.

При оновленні документів у таких колекціях слід пам'ятати, що документи не повинні зростати у розмірах, інакше оновлення не вдасться зробити.

Також не можна видаляти документи з таких колекцій, можна лише видалити всю колекцію.

Для спрощення організації даних у колекціях ми можемо використовувати підколекції. Наприклад, дані в колекції users треба розмежувати на профілі та облікові дані. І ми можемо використовувати для створення колекції db.users.profiles і db.users.accounts. При цьому вони не будуть пов'язані з колекцією users. Тобто в результаті будуть три різні колекції, однак у плані логічної організації зберігання даних, можливо, для когось так буде простіше.

3.1.7 Додавання даних. Як було вказано вище, ім'я колекції – довільний ідентифікатор, що складається з не більше ніж 128 різних алфавітно-цифрових символів та знака підкреслення.

Для поповнення колекції документами передбачено три методи: `insertOne()` для вставки єдиного документа, `insertMany()` для одночасної вставки множини документів та застарілий метод `insert()`, який є універсальним і може виконувати як одинарну, так і множинну вставку. Отже, додамо один документ:

```
testDB> db.Authors.insertOne({"Name": "Іван", "Surname": "Івнченко", "Age": 18, "Education": "Ліцей Ерудит"})
{
  acknowledged: true,
  insertedId: ObjectId("6789131556fe6f10899dc84e")
}
```

Документ є набором пар ключ-значення. У цьому випадку документ, що додається, має чотири ключі: `Name`, `Surname`, `Age`, `Education`.

При створенні JSON-документів слід дотримуватися встановлених синтаксичних правил. Зокрема, імена ключів не повинні починатися з символу долара (\$) та містити символ крапки (.). Це обмеження пов'язане з тим, що ці символи мають спеціальне призначення в синтаксисі JSON і їх використання в іменах ключів може призвести до некоректного тлумачення даних.

При додаванні даних, якщо значення для поля «`_id`» (тобто унікального ідентифікатора документа) не вказано, то воно генерується автоматично. Але, в принципі, ми можемо самі встановити цей ідентифікатор при додаванні даних:

```
testDB> db.Authors.insertOne({"_id": 1, "Name": "Петро", "Surname": "Петренко", "Age": 29})
{ acknowledged: true, insertedId: 1 }
```

Варто відзначити, що назви ключів можуть використовуватися в лапках, а можуть і без лапок. У разі успішного додавання, на консоль буде виведено ідентифікатор доданого документа. І щоб переконатися, що документ у базі даних, ми його виводимо функцією `find`.

```
testDB> db.Authors.find()
[
  { _id: ObjectId("6788ede956fe6f10899dc84c"), Name: 'Сергій Жадан' },
  { _id: ObjectId("6788eea356fe6f10899dc84d"), Name: 'Сергій Жадан' },
  {
    _id: ObjectId("6789131556fe6f10899dc84e"),
    Name: 'Іван',
    Surname: 'Івнченко',
    Age: 18,
    Education: 'Ліцей Ерудит'
  },
  { _id: 1, Name: 'Петро', Surname: 'Петренко', Age: 29 }
]
```

Є ще один спосіб додавання до бази даних документа, який включає два етапи: визначення документа (`document = ( { ... } )`) і власне його додавання:

```
testDB> document = ({ "Name": "Галина", "Surname": "Куприченко", "Age": 17 })
{ Name: 'Галина', Surname: 'Куприченко', Age: 17 }
testDB> db.Athors.insertOne(document)
{
  acknowledged: true,
  insertedId: ObjectId("678917e156fe6f10899dc84f")
}
```

Питання для самоперевірки

Які особливості СУБД MongoDB?

Які відмінності MongoDB від реляційних баз даних?

Які переваги MongoDB?

Які види серверів має MongoDB?

Які опції інсталяції пропонує MongoDB?

За що відповідає програма `mongod`?

Для чого призначена програма `mongosh`?

Яке призначення програми `mongos`?

Що таке шарди в MongoDB?

Який порт використовує MongoDB по замовчуванню?

Що таке колекція в межах MongoDB?

Які правила найменування колекцій?

Що таке документ в межах MongoDB?

Чи є база даних MongoDB реєстрозалежною і строго типізованою?

Що собою представляє документ в MongoDB?

Які типи даних є в MongoDB?

Яким чином задається унікальний ідентифікатор для документа MongoDB?

Які складові автоматично створеного унікального ідентифікатора документа MongoDB?

Які правила найменування баз даних?

Як переключатися між базами даних в MongoDB?

Як переглянути усі бази даних?

Як вивести всі колекції бази даних?

Яким чином отримати статистику поточної бази даних?

Що таке json-файл? Які переваги цього формату? Які недоліки?

Що таке bson-файл? Які недоліки json-файлу він вирішує?

Що таке колекція?

Що таке обмежена колекція? Яка функціональність у обмеженої колекції?

Що буде з обмеженою колекцією, якщо буде зайнятий нею виділений для неї простір?

Що буде з обмеженою колекцією, якщо кількість документів в ній досягне максимально допустимої кількості?

Що буде з обмеженою колекцією, якщо буде зайнятий нею виділений для неї простір, але кількість документів в ній не досягне максимально допустимої кількості?

Чи допускається видалення документів у обмеженій колекції?

Що таке підколекція? Яке її функціональне навантаження?

Які правила найменування колекцій в mongoDB?

Які правила найменування ключів в mongoDB?

Чи можливо в mongoDB встановити унікальний ідентифікатор мануально?

Завдання

В текстовому редакторі написати у форматі JSON множини структур за варіантами (по три множини, щоб в одній був присутній масив об'єктів). Варіанти подані нижче.

Створити БД TEST. Додати до неї множини із завдання 1.

Організувати перегляд вмісту колекції.

Варіант	Опис завдання
1	Структура: назва торгової організації, адреса, ПІБ директора, податковий номер, постачальники (може бути декілька)
2	Структура: ПІБ, посада (може бути декілька), номер відділу (може бути декілька), в якому працює, стать, адреса, дата народження.
3	Структура: ПІБ, паспортні дані, стать, вік, діти (може бути декілька)
4	Структура: назва туру, країна (може бути декілька), міста (може бути декілька), тип пересування (може бути декілька), тип харчування, ціна туру, тип проживання
5	Структура: прізвище, ім'я, по батькові, адреса, місто, телефон (може бути декілька).

3.2 Запити та фільтрація даних у MongoDB

Якщо нам треба отримати не всі документи, а лише ті, які задовольняють певній вимозі, то використовуємо функцію `find()`. Виведемо всі документи, у яких ім'я Іван:

```
testDB> db.Authors.find({Name:"Іван"})
[
  {
    _id: ObjectId("6789131556fe6f10899dc84e"),
    Name: 'Іван',
    Surname: 'Івнченко',
    Age: 18,
    Education: 'Ліцей Ерудит'
  }
]
testDB>
```

Тепер складніший запит: нам потрібно вивести ті об'єкти, які мають ім'я Іван і одночасно вік 18 років:

```
testDB> db.Authors.find({Name:"Іван", Age:18})
[
  {
    _id: ObjectId("6789131556fe6f10899dc84e"),
    Name: 'Іван',
    Surname: 'Івнченко',
    Age: 18,
    Education: 'Ліцей Ерудит'
  }
]
```

Зрозуміло, що документи можуть мати деяку властивість, а можуть і не мати. Що якщо ми хочемо отримати документи, у яких немає певної властивості? Для цього передається значення null. Наприклад, знайдемо всі документи, де відсутня властивість освіта:

```
testDB> db.Authors.find({Education:null})
[
  { _id: ObjectId("6788ede956fe6f10899dc84c"), Name: 'Сергій Жадан' },
  { _id: ObjectId("6788eea356fe6f10899dc84d"), Name: 'Сергій Жадан' },
  { _id: 1, Name: 'Петро', Surname: 'Петренко', Age: 29 }
]
```

Або знайдемо всі документи, де ім'я Петро, але властивість освіта не вказано.

```
testDB> db.Authors.find({Name:"Петро", Education:null})
[ { _id: 1, Name: 'Петро', Surname: 'Петренко', Age: 29 } ]
testDB>
```

Додаймо документ з масивом:

```
db.Authors.insertOne({Name: "Людмила", Surname:"Штраус", Age:45, Languages:["українська", "англійська", "німецька"]})
```

Наступний запит виводить усі документи, у яких у масиві мови є українська:

```
testDB> db.Authors.find({Languages:"українська"})
[
  {
    _id: ObjectId("678a8002e3c957dd45eb864d"),
    Name: 'Людмила',
    Surname: 'Штраус',
    Age: 45,
    Languages: [ 'українська', 'англійська', 'німецька' ]
  }
]
```

Тепер виведемо всі документи, в яких «українська» у масиві мови знаходиться на першому місці:

```
testDB> db.Authors.find({"Languages.0":"українська"})
[
  {
    _id: ObjectId("678a8002e3c957dd45eb864d"),
    Name: 'Людмила',
    Surname: 'Штраус',
    Age: 45,
    Languages: [ 'українська', 'англійська', 'німецька' ]
  }
]
```

Зверніть увагу, що «Languages.0» надає складну властивість і тому береться в лапки. Відповідно, якщо нам треба вивести документи, де українська мова стоїть на другому місці (наприклад, [«англійська», «українська»]), то замість нуля ставимо одиницю: «Languages.1».

3.2.1 Проекція. Документ може мати безліч полів, але не всі ці поля нам можуть бути потрібними і важливими при запиті. І в цьому випадку ми можемо включити у вибірку лише потрібні поля, використовуючи проекцію. Наприклад, виведемо тільки порцію інформації, наприклад, значення полів «Age» для всіх документів, у яких Name=»Іван»:

```
testDB> db.Authors.find({Name:"Людмила"}, { Age:1})
[ { _id: ObjectId("678a8002e3c957dd45eb864d"), Age: 45 } ]
testDB> _
```

І зворотна ситуація: ми хочемо знайти всі поля документа, крім властивості Age. В цьому випадку як параметр вказуємо 0:

```
testDB> db.Authors.find({Name:"Людмила"}, { Age:0})
[
  {
    _id: ObjectId("678a8002e3c957dd45eb864d"),
    Name: 'Людмила',
    Surname: 'Штраус',
    Languages: [ 'українська', 'англійська', 'німецька' ]
  }
]
```

При цьому треба враховувати, що навіть якщо ми зауважимо, що ми хочемо отримати тільки поле name, поле _id також буде включено до результуючої вибірки. Тому, якщо ми не хочемо бачити це поле у вибірці, то треба явно вказати: {«_id»:0}. Альтернативно замість 1 та 0 можна використовувати true та false:


```
testDB> db.Authors.find({Name:"Людмила"}, { _id: false})
[
  {
    Name: 'Людмила',
    Surname: 'Штраус',
    Age: 45,
    Languages: [ 'українська', 'англійська', 'німецька' ]
  }
]
```

Якщо ми не хочемо при цьому конкретизувати вибірку, а хочемо вивести всі документи, можна залишити перші фігурні дужки порожніми:

```
testDB> db.Authors.find({Name:"Людмила"}, { _id: false})
[
  {
    Name: 'Людмила',
    Surname: 'Штраус',
    Age: 45,
    Languages: [ 'українська', 'англійська', 'німецька' ]
  }
]
testDB> db.Authors.find({}, { _id: false})
[
  { Name: 'Сергій Жадан' },
  { Name: 'Сергій Жадан' },
  {
    Name: 'Іван',
    Surname: 'Івнченко',
    Age: 18,
    Education: 'Ліцей Ерудит'
  },
  { Name: 'Петро', Surname: 'Петренко', Age: 29 },
  {
    Name: 'Людмила',
    Surname: 'Штраус',
    Age: 45,
    Languages: [ 'українська', 'англійська', 'німецька' ]
  }
]
```

3.2.2 Складні запити. *Запит до вкладених об'єктів.* Попередні запити застосовувалися до найпростіших об'єктів. Але документи можуть бути дуже складними за структурою. Наприклад, додаймо до колекції Authors наступний документ:

```
testDB> db.Authors.insertOne({Name: "Олексій", Surname:"Штраусенко", Company:{ Name: "КНУ", Country:"Україна"}})
{
  acknowledged: true,
  insertedId: ObjectId("678a8b7ce3c957dd45eb864e")
}
```

Тут визначається вкладений об'єкт із ключем Company. І щоб знайти всі документи, у яких у ключі Company вкладена властивість Name="КНУ", нам потрібно використовувати оператор точку:

```
testDB> db.Authors.find({"Company.Name":"КНУ"})
[
  {
    _id: ObjectId("678a8b7ce3c957dd45eb864e"),
    Name: 'Олексій',
    Surname: 'Штраусенко',
    Company: { Name: 'КНУ', Country: 'Україна' }
  }
]
```

Регулярні вирази. Вони є можливістю будувати спеціального типу запити. Наприклад, знайдемо всі документи, у яких значення ключа Name починається з літери Л:

```
testDB> db.Authors.find({Name:/^Л/})
[
  {
    _id: ObjectId("678a8002e3c957dd45eb864d"),
    Name: 'Людмила',
    Surname: 'Штраус',
    Age: 45,
    Languages: [ 'українська', 'англійська', 'німецька' ]
  }
]
```

MongoDB представляє ряд функцій, які допомагають керувати вибіркою із бази даних. Одна з них – функція limit. Вона визначає максимально допустиму кількість одержуваних документів. Кількість передається як числового параметра. Наприклад, обмежимо вибірку трьома документами:

```
testDB> db.Authors.find().limit(3)
[
  { _id: ObjectId("6788ede956fe6f10899dc84c"), Name: 'Сергій Жадан' },
  { _id: ObjectId("6788eea356fe6f10899dc84d"), Name: 'Сергій Жадан' },
  {
    _id: ObjectId("6789131556fe6f10899dc84e"),
    Name: 'Іван',
    Surname: 'Івнченко',
    Age: 18,
    Education: 'Ліцей Ерудит'
  }
]
```

В даному випадку ми отримаємо перші три документи (якщо в колекції 3 та більше документів). Але що, якщо ми хочемо зробити вибірку не спочатку, а пропустивши якусь кількість документів? У цьому допоможе функція skip. Наприклад, пропустимо перші три записи:

```
testDB> db.Authors.find().skip(3)
[
  { _id: 1, Name: 'Петро', Surname: 'Петренко', Age: 29 },
  {
    _id: ObjectId("678a8002e3c957dd45eb864d"),
    Name: 'Людмила',
    Surname: 'Штраус',
    Age: 45,
    Languages: [ 'українська', 'англійська', 'німецька' ]
  },
  {
    _id: ObjectId("678a8b7ce3c957dd45eb864e"),
    Name: 'Олексій',
    Surname: 'Штраусенко',
    Company: { Name: 'КНУ', Country: 'Україна' }
  }
]
```

Сортування. MongoDB дає змогу відсортувати отриманий з бази набір даних за допомогою функції `sort`. Передаючи в цю функцію значення 1 або -1, ми можемо вказати, в якому порядку сортувати: за зростанням (1) або за спаданням (-1). Багато в чому ця функція аналогічна виразу `ORDER BY` в SQL. Наприклад, сортування за зростанням по полю `Name`:

```
testDB> db.Authors.find().sort({Name:1})
[
  {
    _id: ObjectId("6789131556fe6f10899dc84e"),
    Name: 'Іван',
    Surname: 'Івнченко',
    Age: 18,
    Education: 'Ліцей Ерудит'
  },
  {
    _id: ObjectId("678a8002e3c957dd45eb864d"),
    Name: 'Людмила',
    Surname: 'Штраус',
    Age: 45,
    Languages: [ 'українська', 'англійська', 'німецька' ]
  },
  {
    _id: ObjectId("678a8b7ce3c957dd45eb864e"),
    Name: 'Олексій',
    Surname: 'Штраусенко',
    Company: { Name: 'КНУ', Country: 'Україна' }
  },
  { _id: 1, Name: 'Петро', Surname: 'Петренко', Age: 29 },
  { _id: ObjectId("6788ede956fe6f10899dc84c"), Name: 'Сергій Жадан' },
  { _id: ObjectId("6788eea356fe6f10899dc84d"), Name: 'Сергій Жадан' }
]
```

Ну і в кінці треба відзначити, що ми можемо поєднувати всі ці функції в одному ланцюжку:

```
testDB> db.Authors.find().sort({Name:1}).skip(3).limit(3)
[
  { _id: 1, Name: 'Петро', Surname: 'Петренко', Age: 29 },
  { _id: ObjectId("6788ede956fe6f10899dc84c"), Name: 'Сергій Жадан' },
  { _id: ObjectId("6788eea356fe6f10899dc84d"), Name: 'Сергій Жадан' }
]
```

Якщо раптом нам потрібно відсортувати обмежену колекцію, ми можемо скористатися параметром `$natural`. Цей параметр дає змогу встановити сортування: документи передаються в тому порядку, в якому вони були додані в колекцію, або у зворотному порядку.

Наприклад, відберемо останні п'ять документів:

```
testDB> db.Authors.find().sort({$natural:-1}).limit(5)
[
  {
    _id: ObjectId("678a8b7ce3c957dd45eb864e"),
    Name: 'Олексій',
    Surname: 'Штраусенко',
    Company: { Name: 'КНУ', Country: 'Україна' }
  },
  {
    _id: ObjectId("678a8002e3c957dd45eb864d"),
    Name: 'Людмила',
    Surname: 'Штраус',
    Age: 45,
    Languages: [ 'українська', 'англійська', 'німецька' ]
  },
  { _id: 1, Name: 'Петро', Surname: 'Петренко', Age: 29 },
  {
    _id: ObjectId("6789131556fe6f10899dc84e"),
    Name: 'Іван',
    Surname: 'Івнченко',
    Age: 18,
    Education: 'Ліцей Ерудит'
  },
  { _id: ObjectId("6788eea356fe6f10899dc84d"), Name: 'Сергій Жадан' }
]
```

Комбінування функцій limit і skip. `$slice` є певною мірою комбінацією функцій `limit` і `skip`. Але, на відміну від них, `$slice` може працювати з масивами.

Оператор `$slice` приймає два параметри. Перший параметр вказує на загальну кількість документів, що повертаються. Другий параметр необов'язковий, але якщо він використовується, тоді перший параметр вказує на зсув відносно початку (як функція `skip`), а другий – на обмеження кількості документів, що витягуються.

Наприклад, у кожному документі визначено масив мов для зберігання мов, на яких говорить людина. Їх може бути і одна, і дві, і три та більше. І ми

хочемо при виведенні документів зробити так, щоб у вибірку попадала тільки одна мова з масиву Languages, а не весь масив:

```
testDB> db.Authors.find({Name:"Людмила"},{Languages:{$slice:1}})
[
  {
    _id: ObjectId("678a8002e3c957dd45eb864d"),
    Name: 'Людмила',
    Surname: 'Штраус',
    Age: 45,
    Languages: [ 'українська' ]
  }
]
```

Даний запит під час видалення документа залишить у результаті лише першу мову з масиву Languages, тобто у цьому випадку – українська.

Навпаки: нам треба залишити в масиві також один елемент, але не з початку, а з кінця. У цьому випадку необхідно передати до параметра негативне значення:

```
testDB> db.Authors.find({Name:"Людмила"},{Languages:{$slice:-1}})
[
  {
    _id: ObjectId("678a8002e3c957dd45eb864d"),
    Name: 'Людмила',
    Surname: 'Штраус',
    Age: 45,
    Languages: [ 'німецька' ]
  }
]
```

Пошук єдиного документу. Якщо всі документи виймаються функцією find, то один документ витягується функцією findOne:

```
testDB> db.Authors.findOne()
{ _id: ObjectId("6788ede956fe6f10899dc84c"), Name: 'Сергій Жадан' }
testDB>
```

Її дія аналогічна тому, якби ми використовували функцію limit(1), яка також отримує перший документ колекції. А комбінація функцій skip та limit витягне документ за потрібним місцезнаходженням.

3.2.3 Число елементів у колекції. За допомогою функції countDocuments() можна отримати кількість елементів у колекції:

```
testDB> db.Authors.countDocuments()
6
```

Можна групувати параметри пошуку та іншу функцію count(), щоб підрахувати скільки певних документів, наприклад, у яких Name= «Людмила»:

```
testDB> db.Authors.find({Name:"Людмила"}).count()  
1
```

Більш того, ми можемо створювати ланцюжки функцій, щоб конкретизувати умови підрахунку:

```
testDB> db.Authors.find({Name:"Людмила"}).skip(1).count(true)  
0
```

Тут слід зазначити, що за замовчуванням функція count не використовується з функціями limit та skip. Щоб їх використати, як у прикладі вище, у функцію count треба передати булеве значення true

Колекція може мати документи, які містять однакові значення для одного або декількох полів. Наприклад, у кількох документах визначено Name: «Людмила». І нам треба знайти тільки унікальні значення для одного з полів документа. Для цього ми можемо скористатися функцією distinct:

```
testDB> db.Authors.distinct("Name")  
[ 'Іван', 'Людмила', 'Олексій', 'Петро', 'Сергій Жадан' ]
```

3.2.4 Умовні оператори. Умовні оператори задають умову, якій має відповідати значення поля документа (таблиця 3.2).

Таблиця 3.2. Умовні оператори MongoDB

\$eq	рівно
\$ne	не рівно
\$gt	більше ніж
\$lt	менше ніж
\$gte	більше чи рівно
\$lte	менше або рівно
\$in	визначає масив значень, одне з яких повинно мати поле документа
\$nin	визначає масив значень, які не повинно мати поле документа

Наприклад, знайдемо всі документи, у яких значення ключа Age менше 35:

```
testDB> db.Authors.find({Age:{$lt:30}})
[
  {
    _id: ObjectId("6789131556fe6f10899dc84e"),
    Name: 'Іван',
    Surname: 'Івнченко',
    Age: 18,
    Education: 'Ліцей Ерудит'
  },
  { _id: 1, Name: 'Петро', Surname: 'Петренко', Age: 29 }
]
```

Аналогічно використовуємо інші оператори порівняння. Наприклад, той самий ключ, тільки більше 30:

```
testDB> db.Authors.find({Age:{$gt:30}})
[
  {
    _id: ObjectId("678a8002e3c957dd45eb864d"),
    Name: 'Людмила',
    Surname: 'Штраус',
    Age: 45,
    Languages: [ 'українська', 'англійська', 'німецька' ]
  }
]
```

Зверніть увагу, що порівняння тут проводиться над цілими типами, а не рядками. Якщо ключ Age є строковими значеннями, то відповідно треба проводити порівняння над рядками: `db.Acthors.find ({Age: {$gt : «30»}})`, однак результат буде тим самим.

Уявімо ситуацію, що необхідно знайти всі об'єкти зі значенням поля Age більше 30, але менше 50. У цьому випадку ми можемо комбінувати два оператори:

```
testDB> db.Authors.find({Age:{$gt:30, $lt:50}})
[
  {
    _id: ObjectId("678a8002e3c957dd45eb864d"),
    Name: 'Людмила',
    Surname: 'Штраус',
    Age: 45,
    Languages: [ 'українська', 'англійська', 'німецька' ]
  }
]
```

Знайдемо авторів, вік яких дорівнює 18:

```
testDB> db.Authors.find({Age:{$eq:18}})
[
  {
    _id: ObjectId("6789131556fe6f10899dc84e"),
    Name: 'Іван',
    Surname: 'Івнченко',
    Age: 18,
    Education: 'Ліцей Ерудит'
  }
]
```

Зворотня операція – знайдемо користувачів, вік яких не дорівнює 18:

```
testDB> db.Authors.find({Age:{$ne:18}})
[
  { _id: ObjectId("6788ede956fe6f10899dc84c"), Name: 'Сергій Жадан' },
  { _id: ObjectId("6788eea356fe6f10899dc84d"), Name: 'Сергій Жадан' },
  { _id: 1, Name: 'Петро', Surname: 'Петренко', Age: 29 },
  {
    _id: ObjectId("678a8002e3c957dd45eb864d"),
    Name: 'Людмила',
    Surname: 'Штраус',
    Age: 45,
    Languages: [ 'українська', 'англійська', 'німецька' ]
  },
  {
    _id: ObjectId("678a8b7ce3c957dd45eb864e"),
    Name: 'Олексій',
    Surname: 'Штраусенко',
    Company: { Name: 'КНУ', Country: 'Україна' }
  }
]
```

Оператор \$in визначає масив можливих виразів і шукає ключі, значення яких є в масиві:

```
testDB> db.Authors.find({Age:{$in:[18,32]}})
[
  {
    _id: ObjectId("6789131556fe6f10899dc84e"),
    Name: 'Іван',
    Surname: 'Івнченко',
    Age: 18,
    Education: 'Ліцей Ерудит'
  }
]
```

Протилежним чином діє оператор \$nin – він визначає масив можливих виразів і шукає ключі, значення яких відсутнє в цьому масиві:

```
testDB> db.Authors.find({Age:{$nin:[18,32]}})
[
  {
    _id: ObjectId("6789131556fe6f10899dc84e"),
    Name: 'Іван',
    Surname: 'Івнченко',
    Age: 18,
    Education: 'Ліцей Ерудит'
  }
]
```



```
testDB> db.Authors.find({Age:{$nin:[18,32]}})
[
  { _id: ObjectId("6788ede956fe6f10899dc84c"), Name: 'Сергій Жадан' },
  { _id: ObjectId("6788eea356fe6f10899dc84d"), Name: 'Сергій Жадан' },
  { _id: 1, Name: 'Петро', Surname: 'Петренко', Age: 29 },
  {
    _id: ObjectId("678a8002e3c957dd45eb864d"),
    Name: 'Людмила',
    Surname: 'Штраус',
    Age: 45,
    Languages: [ 'українська', 'англійська', 'німецька' ]
  },
  {
    _id: ObjectId("678a8b7ce3c957dd45eb864e"),
    Name: 'Олексій',
    Surname: 'Штраусенко',
    Company: { Name: 'КНУ', Country: 'Україна' }
  }
]
```

3.2.5 Логічні оператори. Логічні оператори виконуються над умовами вибірки (таблиця 3.3).

Таблиця 3.3.Логічні оператори mongoDB

or	\$	з'єднує дві умови, і документ повинен відповідати одній з цих умов
and	\$	з'єднує дві умови, і документ повинен відповідати обом умовам
not	\$	документ повинен НЕ відповідати умові
nor	\$	з'єднує дві умови, і документ повинен НЕ відповідати обом умовам

Оператор \$or представляє логічну операцію АБО і визначає набір пар ключ-значення, які мають бути в документі. І якщо документ має хоч одну таку пару ключ-значення, він відповідає даному запиту і витягується з колекції:

```
testDB> db.Authors.find({$or:[{Name: "Петро"}, {Age:29}]})
[ { _id: 1, Name: 'Петро', Surname: 'Петренко', Age: 29 } ]
```

Цей вираз поверне нам всі документи, у яких або Name=Петро, або Age=29.

У підвиразах or можна застосовувати умовні оператори.

Оператор `$and` представляє логічну операцію І (логічне множення) і визначає набір критеріїв, яким обов'язково має відповідати документ. На відміну від оператора `$or`, документ повинен відповідати всім зазначеним критеріям. Наприклад:

```
testDB> db.Authors.find({$and:[{Name: "Петро"}, {Age:29}]})
[ { _id: 1, Name: 'Петро', Surname: 'Петренко', Age: 29 } ]
```

Тут документи, що вибираються, обов'язково повинні мати ім'я Петро і вік 29 – обидві ці ознаки.

3.2.6 Пошук по масивам. Ряд операторів призначений для роботи з масивами (таблиця 3.4).

Таблиця 3.4. Оператори для роботи з масивами в MongoDB

\$all	визначає набір значень, які мають бути в масиві
\$size	визначає кількість елементів, які мають бути в масиві
\$elemMatch	визначає умову, якій повинні відповідати елементи в масиві

Оператор `$all` визначає масив можливих виразів і вимагає, щоб документи мали весь набір виразів, що визначається. Відповідно він застосовується для пошуку за масивом. Наприклад, у документах є масив `Languages`, що зберігає мови, якими володіє автор. І щоб знайти всіх авторів, які розмовляють одночасно і англійською, і українською, ми можемо використовувати наступний вираз:

```
testDB> db.Authors.find({Languages:{$all:["українська","англійська"]}})
[
  {
    _id: ObjectId("678a8002e3c957dd45eb864d"),
    Name: 'Людмила',
    Surname: 'Штраус',
    Age: 45,
    Languages: [ 'українська', 'англійська', 'німецька' ]
  }
]
```

Оператор `$elemMatch` в MongoDB використовується для вибору документів, де хоча б один елемент у масиві відповідає всім заданим критеріям. Він дозволяє шукати елементи масиву, які є об'єктами, і перевіряти їх властивості одночасно. Окрім використання `$elemMatch` у запитах для пошуку

документів, ви також можете використовувати його в проєкції (оператор `project`) для того, щоб повернути лише той елемент масиву, який відповідає вашим критеріям. Це дуже корисно, коли масив містить багато елементів, але вам потрібен лише один або декілька з них. Оператор `$elemMatch` дає змогу вибрати документи, у яких масиви містять елементи, які під певні умови. Наприклад, нехай у базі даних буде колекція, яка містить рейтинги авторів за певними жанрами літератури. Додамо кілька документів:

```
testDB> var myAuthors=
...
[
...
  {
...
    _id : 1,
    "AuthorsName" : "Сергій Жадан",
    "AuthorsRating": [{ "Name": "poems", "R": 10 }, { "Name": "Novel", "R": 8 }, { "Name": "Detective", "R": 1 } ]
...
  },
  {
...
    _id : 2,
    "AuthorsName" : "Андрій Любка",
    "AuthorsRating": [{ "Name": "poems", "R": 1 }, { "Name": "Novel", "R": 10 }, { "Name": "Detective", "R": 1 } ]
...
  },
  {
...
    _id : 3,
    "AuthorsName" : "Оксана Забужко",
    "AuthorsRating": [{ "Name": "poems", "R": 5 }, { "Name": "Novel", "R": 7 }, { "Name": "Detective", "R": 1 } ]
...
  }
...
];

testDB> db.RatingCall.insertMany(myAuthors);
{ acknowledged: true, insertedIds: { '0': 1, '1': 2, '2': 3 } }
```

Кожен документ має масив рейтингів, який у свою чергу складається з вкладених документів.

Тепер знайдемо авторів, які для жанру поезія мають оцінку вище 5:

```
testDB> db.RatingCall.find({AuthorsRating:{ $elemMatch:{Name:"poems",R:{ $gt:5}}}})
[
  {
    _id: 1,
    AuthorsName: 'Сергій Жадан',
    AuthorsRating: [
      { Name: 'poems', R: 10 },
      { Name: 'Novel', R: 8 },
      { Name: 'Detective', R: 1 }
    ]
  }
]
```

Оператор `$size` використовується для знаходження документів, у яких масиви мають число елементів, що дорівнює значенню `$size`. Наприклад, витягнемо всі документи, в яких у масиві `Languages` три елементи:

```
testDB> db.Authors.find({Languages:{$size:3}})
[
  {
    _id: ObjectId("678a8002e3c957dd45eb864d"),
    Name: 'Людмила',
    Surname: 'Штраус',
    Age: 45,
    Languages: [ 'українська', 'англійська', 'німецька' ]
  }
]
```

Оператор `$exists` дозволяє вилучити лише ті документи, в яких певний ключ присутній або відсутній. Наприклад, повернемо всі документи, в які є ключ компанії:

```
testDB> db.Authors.find({Company:{$exists:true}})
[
  {
    _id: ObjectId("678a8b7ce3c957dd45eb864e"),
    Name: 'Олексій',
    Surname: 'Штраусенко',
    Company: { Name: 'КНУ', Country: 'Україна' }
  }
]
```

Оператор `$regex` визначає регулярний вираз, якому має відповідати значення поля. Наприклад, нехай поле `Name` обов'язково має букву «Л»:

```
testDB> db.Authors.find({Name:{$regex:"Л"}})
[
  {
    _id: ObjectId("678a8002e3c957dd45eb864d"),
    Name: 'Людмила',
    Surname: 'Штраус',
    Age: 45,
    Languages: [ 'українська', 'англійська', 'німецька' ]
  }
]
```

Важливо розуміти, що `$regex` приймає не просто рядки, а саме регулярні вирази, наприклад: `name: {$regex:»ко$»}` – значення `Name` має закінчуватися на «ко».

Питання для самоперевірки

У якому випадку передаємо значення `null` властивості документу?

Чи має значення порядок у якому вказуємо властивості документу у запиті про пошук?

Чи можна обмежити кількість документів, яка відповідатиме запиту?

Скільки є різних способів сортування документів?

Як відсортувати обмежену колекцію документів?

Чи є можливість вплинути на кількість, які відображаються у запиті?

Чи існує спосіб вивести всі документи за допомогою значення поля?

За допомогою якої функції можна отримати кількість документів у колекції?

Як знайти унікальні значення поля документів колекції?

Вкажіть умовні оператори, які визначають масив значень?

Які умовні оператори мають протилежний ефект від їх застосування?

Який оператор відповідає за вивід документу, що відповідає одній з вказаних умов?

Який оператор відповідає за вивід документу, який не вказаний умові?

Який оператор відповідає за вивід документу, який не відповідає одночасно двом вказаним умовам?

Який оператор дозволяє вилучити лише ті документи, в яких певний ключ присутній або відсутній?

Який оператор визначає масив можливих виразів і вимагає, щоб документи мали весь набір виразів, що визначається?

Завдання

Написати запити:

Варіант	Опис завдання
1	Структура: назва торгової організації, адреса, ПІБ директора, податковий номер, постачальники (може бути декілька) 1.Вивести всі назви (лише) торгових організацій, які зберігаються у створеній у попередньому завданні базі даних. 2.Вивести всю інформацію із колекції, відсортовану за ПІБ директора.

	3.Для кожної торгової організації вивести лише першого постачальника. 4.Порахувати торгові організації, які зберігаються у створеній у попередньому завданні базі даних. 5.Вивести торгові організації у яких лише два постачальника. 6.Додати ще три організації, у яких буде зазначено, скільки торгових точок, вони мають. Вивести всі організації, у яких більше ніж одна торгова точка.
2	Структура: ПІБ, посада (може бути декілька), номер відділу (може бути декілька), в якому працює, стать, адреса, дата народження. 1.Вивести всіх співробітників організації, відсортованих за алфавітом. 2.Вивести співробітника, прізвище якого починається на літеру «Ю», та його посади. 3.Вивести всіх співробітників та їх останню посаду. 4.Вивести всіх робітників, дата народження яких не пізніше ніж початок цього століття. 5.Вивести співробітників, які займають дві визначені посади. 6.Додати в колекцію ще п'ять документів, до існуючих ключів потрібно додати масив вкладених структур – діти, з полями стать і вік. Вивести співробітників. Які мають дочок старших за 10 років.
3	Структура: ПІБ, паспортні дані, стать, вік, діти (може бути декілька) 1.Вивести всіх громадян, вік яких від 26 років до 60. 2.Вивести всіх дітей, громадянина, прізвище якого закінчується на літери «ко». 3.Вивести інформацію із колекції, відсортовану у порядку оберненому, як колекція заповнювалася.

	4.Вивести із колекції ПІБ та паспортні дані громадян. 5.Вивести громадян, ім'я синів яких Іван і старших ніж 10 років. 6.Порахувати громадян з ім'ям Михайло.
4	Структура: назва туру, країна (може бути декілька), міста (може бути декілька), тип пересування (може бути декілька), тип харчування, ціна туру, тип проживання. 1.Із колекції вивести всі тури, ціна яких не перевищує 15 тис.грн. 2.Із колекції вивести всі тури з одним типом пересування. 3.Порахувати кількість турів. 4.Із колекції вивести всі тури, ціна яких не перевищує 15 тис.грн. 5.Із колекції вивести всі тури з одним типом пересування. 6.Порахувати кількість турів.
5	Структура: прізвище, ім'я, по батькові, адреса, місто, телефон (може бути декілька). 1.Вивести всю інформацію із колекції, при цьому пропустивши перші два записи. 2.Вивести прізвище, ім'я, по батькові громадян, які мають телефон Lifecell. 3.Вивести прізвище, ім'я, по батькові та їх адреси. 4.Вивести всіх громадян, які мають два оператора зв'язку. 5.В колекцію додати чотири документа, до існуючих ключів додати ще масив з вкладеними структурами, які відображають інформацію – посада та досвід роботи. Вивести всіх громадян, які працювали на визначеній посаді більше 3 років. 6.Порахувати кількість документів в колекції.

3.3 Оновлення даних. Видалення даних

Як і інші системи управління базами даних, MongoDB надає можливість оновлення даних.

Якщо нам потрібно повністю замінити один документ іншим, може використовуватись функція `replaceOne`, яка має наступний синтаксис:

`db.collection.replaceOne(filter, update, options),`

де `filter` – приймає запит на вибірку документа, який потрібно оновити;

`update` – представляє новий документ, який замістить старий під час оновлення;

`options` – визначає додаткові параметри під час оновлення документів, основним з яких є параметр `upsert`. Якщо параметр `upsert` має значення `true`, `mongodb` буде оновлювати документ, якщо він знайдений, і створювати новий, якщо такого документа немає. Якщо він має значення `false`, то `mongodb` не буде створювати новий документ у випадку, коли запит на вибірку не знайде жодного документа.

Наприклад, в колекції є документ:

```
testDB> db.Authors.find({Name:"Олексій"})
[
  {
    _id: ObjectId("678a8b7ce3c957dd45eb864e"),
    Name: 'Олексій',
    Surname: 'Штраусенко',
    Company: { Name: 'КНУ', Country: 'Україна' }
  }
]
```

замінімо його на інший:

```
testDB> db.users.replaceOne({Name: "Олексій"}, {Name: "Олексій", Age: 25})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 0,
  modifiedCount: 0,
  upsertedCount: 0
}
```

У результаті, після фільтрування, отримаємо:


```
testDB> db.Authors.find({Name:"Олексій"})
[
  {
    _id: ObjectId("678a8b7ce3c957dd45eb864e"),
    Name: 'Олексій',
    Age: 25
  }
]
```

Часто не потрібно оновлювати весь документ, а лише значення однієї або кількох його властивостей. Для цього використовуються функції `updateOne()` – вона оновлює лише один документ і `updateMany()` – дає змогу оновити множину документів. Для оновлення окремих полів у цих функціях використовується оператор `$set`:

```
testDB> db.Authors.updateOne({Name : "Олексій", Age: 25}, {$set: {Age : 28}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

Тут ми шукаємо документ з `Name=«Олексій»` та `Age=25` та встановлюємо для його властивості `Age` значення 28

Якщо оновлюваного поля в документі немає, воно додається:

```
testDB> db.Authors.updateOne({Name : "Олексій", Age: 25}, {$set: {Age : 28}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
testDB> db.Authors.updateOne({Name : "Олексій", Age: 28}, {$set: {Salary : 300}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
testDB> db.Authors.find({Name:"Олексій"})
[
  {
    _id: ObjectId("678a8b7ce3c957dd45eb864e"),
    Name: 'Олексій',
    Age: 28,
    Salary: 300
  }
]
```

Якщо потрібно оновити значення кількох полів, то вони передаються оператору \$set через кому:

```
testDB> db.Authors.updateOne({Name : "Олексій"}, {$set: {Name: "Олекса", Age : 25}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

```
testDB> db.Authors.find({Name:"Олекса"})
[
  {
    _id: ObjectId("678a8b7ce3c957dd45eb864e"),
    Name: 'Олекса',
    Age: 25,
    Salary: 300
  }
]
```

Для простого збільшення значення числового поля на певну кількість одиниць застосовується оператор \$inc. Якщо документ не містить оновлюване поле, воно створюється. Цей оператор застосовується лише до числових значень:

```
testDB> db.Authors.updateOne({Name : "Олекса"}, {$inc: {Age:2}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
testDB> db.Authors.find({Name:"Олекса"})
[
  {
    _id: ObjectId("678a8b7ce3c957dd45eb864e"),
    Name: 'Олекса',
    Age: 27,
    Salary: 300
  }
]
```

Якщо необхідно оновити всі документи, що відповідають певному критерію, застосовується функція updateMany():

```
testDB> db.Authors.updateMany({Name : "Олекса"}, {$set: {Salary : 560}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

Для видалення окремого ключа використовується оператор \$unset:

```
testDB> db.Authors.updateOne({Name : "Олекса"}, {$unset: {Salary: 1}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
testDB>
[
  {
    _id: ObjectId("678a8b7ce3c957dd45eb864e"),
    Name: 'Олекса',
    Age: 27
  }
]
```

Якщо раптом такого ключа в документі не існує, то оператор не має жодного впливу. Також можна видаляти одразу кілька полів:

```
testDB> db.Authors.update({Name : "Олекса"}, {$unset: {Salary: 1, Age: 1}})
DeprecationWarning: Collection.update() is deprecated. Use updateOne, updateMany, or bulkWrite.
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
testDB> db.Authors.find({Name:"Олекса"})
[ { _id: ObjectId("678a8b7ce3c957dd45eb864e"), Name: 'Олекса' } ]
```

Оператор \$push дає змогу додати ще одне значення до існуючого. Наприклад, якщо ключ як значення зберігає масив:

```
testDB> db.Authors.updateOne({Name : "Олекса"}, {$push: {Languages: "українська"}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
testDB> db.Authors.find({Name:"Олекса"})
[
  {
    _id: ObjectId("678a8b7ce3c957dd45eb864e"),
    Name: 'Олекса',
    Languages: [ 'українська ' ]
  }
]
```

Вище використовується функція updateOne, але цей оператор також застосовується і у функції updateMany:

```
testDB> db.Authors.updateMany({Name : "Олекса"}, {$push: {Languages: "англійська"}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
testDB> db.Authors.find({Name:"Олекса"})
[
  {
    _id: ObjectId("678a8b7ce3c957dd45eb864e"),
    Name: 'Олекса',
    Languages: [ 'українська ', 'англійська' ]
  }
]
```

Якщо ключ, для якого ми хочемо додати значення, не є масивом, ми отримаємо помилку: «Cannot apply \$push/\$pushAll modifier to non-array».

Використовуючи оператор \$each, можна додати відразу кілька значень:

```
testDB> db.Authors.updateOne({Name : "Олекса"}, {$push: {Languages: {$each: [ "арабська", "китайська" ]}}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
testDB> db.Authors.find({Name:"Олекса"})
[
  {
    _id: ObjectId("678a8b7ce3c957dd45eb864e"),
    Name: 'Олекса',
    Languages: [ 'українська ', 'англійська', 'арабська', 'китайська' ]
  }
]
```

Ще пара операторів дає змогу настроїти вставку. Оператор `$position` задає позицію у масиві для вставки елементів, а оператор `$slice` вказує, скільки елементів залишити у масиві після вставки.

```
testDB> db.Authors.updateOne({Name : "Олекса"}, {$push: {Languages: {$each: ["німецька", "польська"], $position:1, $slice:4} }})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}

testDB> db.Authors.find({Name:"Олекса"})
[
  {
    _id: ObjectId("678a8b7ce3c957dd45eb864e"),
    Name: 'Олекса',
    Languages: [ 'українська ', 'німецька', 'польська', 'англійська' ]
  }
]
```

У даному випадку, елементи [«німецька», «польська»] будуть вставлятися в масив `Languages` з 1-го індексу, і після вставки в масиві залишаться тільки 4 перших елементи.

Оператор `$addToSet` подібно до оператора `$push` додає об'єкти в масив. Відмінність полягає в тому, що `$addToSet` додає дані, якщо їх ще немає в масиві (при додаванні через `$push` дані дублюються, якщо додаються елементи, які є в масиві):

```
testDB> db.Authors.updateOne({Name : "Олекса"}, {$addToSet: {Languages: "румунська"}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}

testDB> db.Authors.find({Name:"Олекса"})
[
  {
    _id: ObjectId("678a8b7ce3c957dd45eb864e"),
    Name: 'Олекса',
    Languages: [
      'українська ',
      'німецька',
      'польська',
      'англійська',
      'румунська'
    ]
  }
]
```

Оператор `$pop` дає змогу видаляти елемент із масиву:

```
testDB> db.Authors.updateOne({Name : "Олекса"}, {$pop: {Languages: 1}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

Вказуючи на ключ Languages значення 1, ми видаляємо перший елемент з кінця:

```
testDB> db.Authors.find({Name:"Олекса"})
[
  {
    _id: ObjectId("678a8b7ce3c957dd45eb864e"),
    Name: 'Олекса',
    Languages: [ 'українська ', 'німецька', 'польська', 'англійська' ]
  }
]
```

Щоб видалити перший елемент спочатку масиву, треба передати негативне значення:

```
testDB> db.Authors.updateOne({Name : "Олекса"}, {$pop: {Languages: -1}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
testDB> db.Authors.find({Name:"Олекса"})
[
  {
    _id: ObjectId("678a8b7ce3c957dd45eb864e"),
    Name: 'Олекса',
    Languages: [ 'німецька', 'польська', 'англійська' ]
  }
]
```

Дещо іншу дію передбачає оператор \$pull. Він видаляє кожне входження елемента до масиву. Наприклад, через оператор \$push ми можемо додати те саме значення в масив кілька разів. І тепер за допомогою \$pull видалимо його:

```
testDB> db.Authors.updateOne({Name : "Олекса"}, {$pull: {Languages: "англійська"}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
testDB> db.Authors.find({Name:"Олекса"})
[
  {
    _id: ObjectId("678a8b7ce3c957dd45eb864e"),
    Name: 'Олекса',
    Languages: [ 'німецька', 'польська' ]
  }
]
```

Якщо ми хочемо видалити не одне значення, а відразу кілька, тоді ми можемо застосувати оператор \$pullAll:

```
testDB> db.Authors.updateOne({Name : "Олекса"}, {$pullAll: {Languages: ["німецька", "польська"]}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
testDB> db.Authors.find({Name:"Олекса"})
[
  {
    _id: ObjectId("678a8b7ce3c957dd45eb864e"),
    Name: 'Олекса',
    Languages: []
  }
]
```

Для видалення документів у MongoDB передбачені функції deleteOne() – видаляє один документ та deleteMany() – дає змогу видалити кілька документів. Як параметр у ці функції передається фільтр документів, що видаляються.

Наприклад, видалимо документ, у якому Name=«Олекса»:

```
testDB> db.Authors.deleteOne({Name : "Олекса"})
{ acknowledged: true, deletedCount: 1 }
```

У результаті перший знайдений документ із name=Том буде видалено. Для видалення всіх документів, які відповідають фільтру, застосовується функція deleteMany():

```
testDB> db.Authors.deleteMany({Name : "Олекса"})
```

Причому, як і у випадку з find, ми можемо задавати умови вибірки для видалення у різний спосіб (у вигляді регулярних виразів, у вигляді умовних конструкцій тощо):

```
testDB> db.Authors.deleteOne({Name : /^T\w+/i}),
testDB> db.Authors.deleteOne({Age: {$lt : 30}}).
```

Щоб видалити разом всі документи з колекції, треба залишити порожнім параметр запиту:

```
testDB> db.Authors.deleteMany({}).
```

Ми можемо видаляти не лише документи, а й колекції та бази даних. Для видалення колекцій використовується функція `drop`:

```
testDB> db.Authors.drop().
```

І якщо видалення колекції пройде успішно, то консоль виведе `true`.

Щоб видалити всю базу даних, треба скористатися функцією `dropDatabase()`:

```
testDB> db.dropDatabase().
```

Питання для самоперевірки

Яка функція дає змогу повністю замінити один документ іншим?

Яка функція дає змогу оновити документ частково?

Яка функція використовується для оновлення всіх документів?

Які оператори дозволяють реалізовувати вставку?

Яка функція реалізовує видалення документів?

Завдання

Написати запити:

Варіант	Опис завдання
1	Структура: назва торгової організації, адреса, ПІБ директора, податковий номер, постачальники (може бути декілька), кількість торгових точок. 1.Для трьох документів оновити значення полів: назва торгової операції, ПІБ директора.

	2.Для трьох документів додати по чотири постачальника. 3.Для трьох документів збільшити кількість торгових точок на три. 4. Для одного документа видалити останнього постачальника.
2	Структура: ПІБ, посада (може бути декілька), номер відділу (може бути декілька), в якому працює, стать, адреса, дата народження. 1.Для чотирьох документів змінити ПІБ а номер відділу. 2.Для двох документів додати на позицію 1 нову посаду. 3.Для двох документів видалити кожне входження певної посади. 4.Видалити всіх всіх робітників з певного відділу.
3	Структура: ПІБ, паспортні дані, стать, вік, діти (може бути декілька) 1.Для трьох документів оновити паспортні дані. 2.Для трьох громадян додати по ще одній дитині.. 3.Для трьох громадян збільшити вік на один рік. 4.Видалити одного певного громадянина.
4	Структура: назва туру, країна (може бути декілька), міста (може бути декілька), тип пересування (може бути декілька), тип харчування, ціна туру, тип проживання 1.Змінити назви чотирьох турів. 2.Для чотирьох турів змінити ціну на три тис. грн.. 3.Для одного туру додати ще одне місто на першу позицію і обмежити кількість міст трьома. 4. Для одного туру видалити перше місто.
5	Структура: прізвище, ім'я, по батькові, адреса, місто, телефон (може бути декілька). 1.Для трьох громадян змінити ім'я та прізвище.

	2.Для трьох громадян додайте ще по одному номеру телефону на позицію 0 і обмежте кількість номерів телефонів двома. 3.Для одного громадянина видаліть перший номер телефону. 4 Із колекції видаліть громадянина, який проживає за певним адресом.
--	--

3.4 Інструменти, моделювання та індексація даних у MongoDB

3.4.1 MongoDB Compass. Існує декілька підходів до управління системами управління базами даних. Серед них виділяються два основних інструменти: командний інтерфейс (CLI) на базі mongosh та графічний інтерфейс користувача (GUI) у вигляді модуля Compass. Останній особливо корисний для візуалізації даних.

Робота з CLI вимагає знання специфічних команд та принципів функціонування командного інтерфейсу. Для користувачів, які не мають досвіду роботи з CLI, компанія MongoDB розробила інтуїтивно зрозумілий графічний інструмент Compass. Цей інструмент дає змогу ефективно взаємодіяти з СУБД без необхідності заглиблюватися у складні синтаксичні конструкції командного рядка.

Після запуску Compass, передбачені два сценарії підключення: до локального сервера (localhost) або до віддаленого сервера. В останньому випадку, користувачу необхідно ввести відповідну IP-адресу та номер порту. Після заповнення всіх необхідних полів, підключення здійснюється шляхом натискання на кнопку «Підключити» (рис. 3.5)

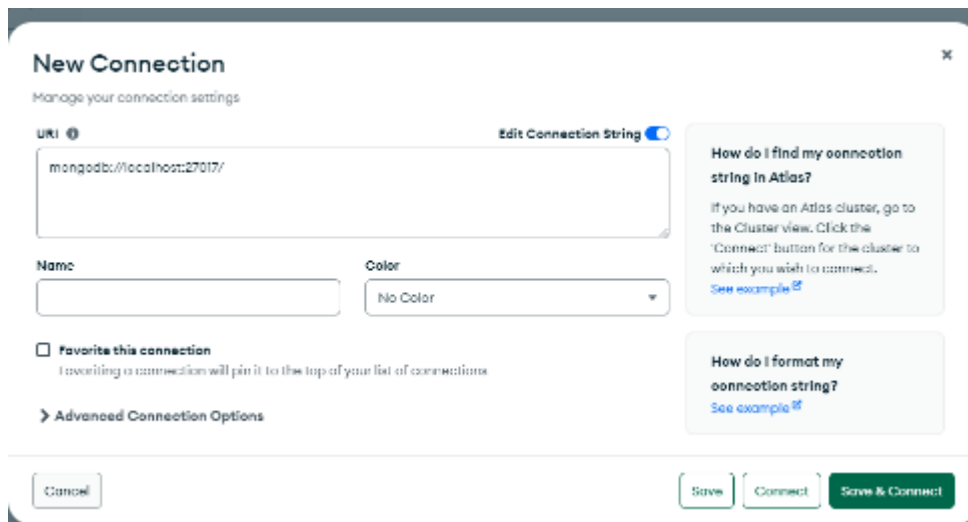


Рисунок 3.5 – Вікно підключення до СУБД

Головне вікно СУБД забезпечує користувачеві інтуїтивно зрозумілий інтерфейс, що складається з різноманітних графічних компонентів та навігаційних меню (рис. 3.6). Окрім стандартних методів взаємодії з системою, передбачена можливість використання вбудованого інтерпретатора команд MongoDB, що дає змогу виконувати більш складні операції з базою даних на низькому рівні (рис. 3.7).

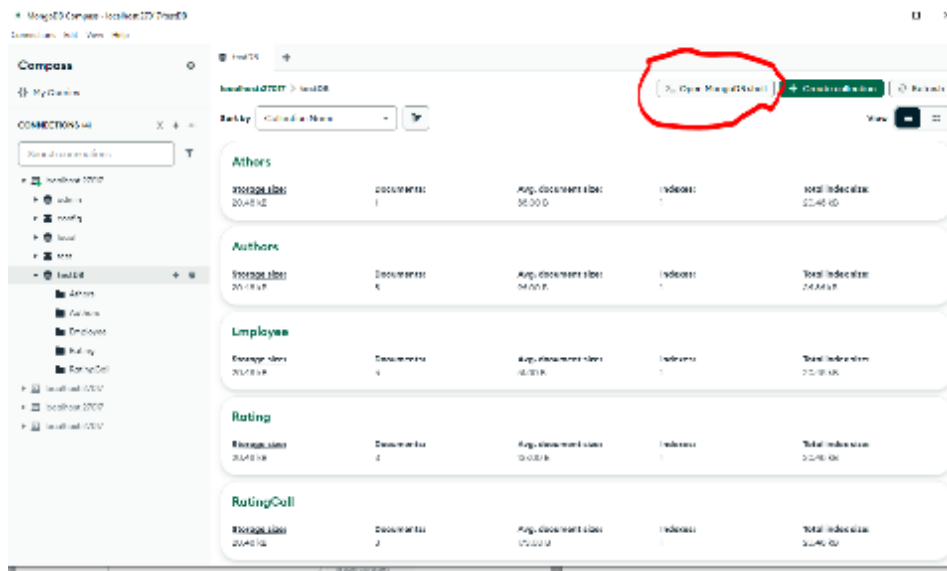


Рисунок 3.6 – Інтерфейс головного екрану GUI Compass

```
>_MONGOSH  
  
> use testDB  
< switched to db testDB  
testDB> |
```

Рисунок 3.7 – Командний рядок

З метою демонстрації функціоналу системи, виконаємо наступні дії: створимо нову базу даних та колекцію. Далі проаналізуємо зміни, що стануться в лівому панелі навігації, де відображається список усіх створених баз даних:

```
>_MONGOSH  
  
> use testDB  
< switched to db testDB  
> db.createCollection("Poetry")  
< { ok: 1 }
```

Оновлення списку баз даних за допомогою кнопки «Refresh» дає змогу миттєво відобразити будь-які зміни в структурі даних, включаючи нові бази даних та колекції (рис. 3.8).

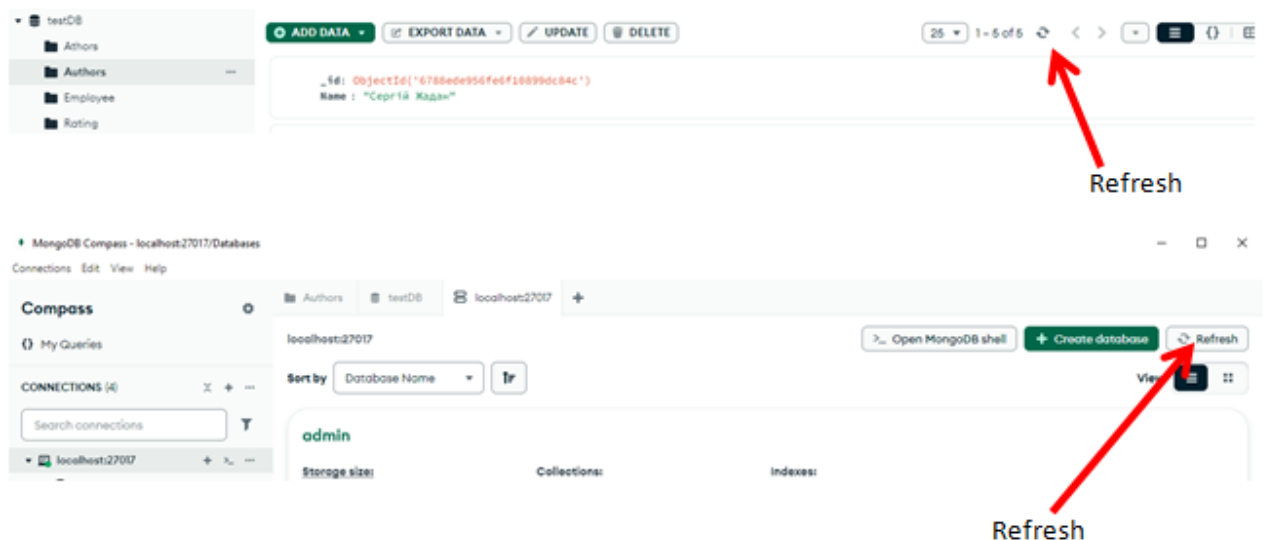


Рисунок 3.8 – Розташування кнопки «Refresh»

Для того щоб створити базу даних без командного рядка, потрібно натиснути кнопку зі знаком «+» біля назви сервера (рис. 3.9).

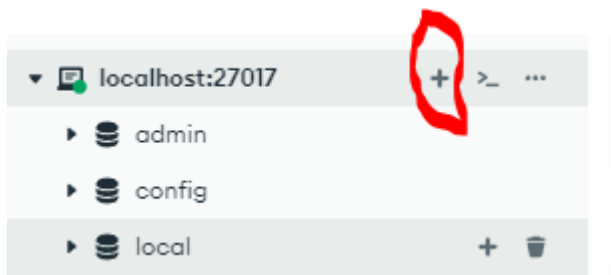


Рисунок 3.9 – Додавання бази даних за допомогою графічного інтерфейсу

Після цього відкриється вікно для додавання бази даних, в якому потрібно ввести назву бази даних і назву колекції і натиснути кнопку “Create Database” (рис. 3.10).

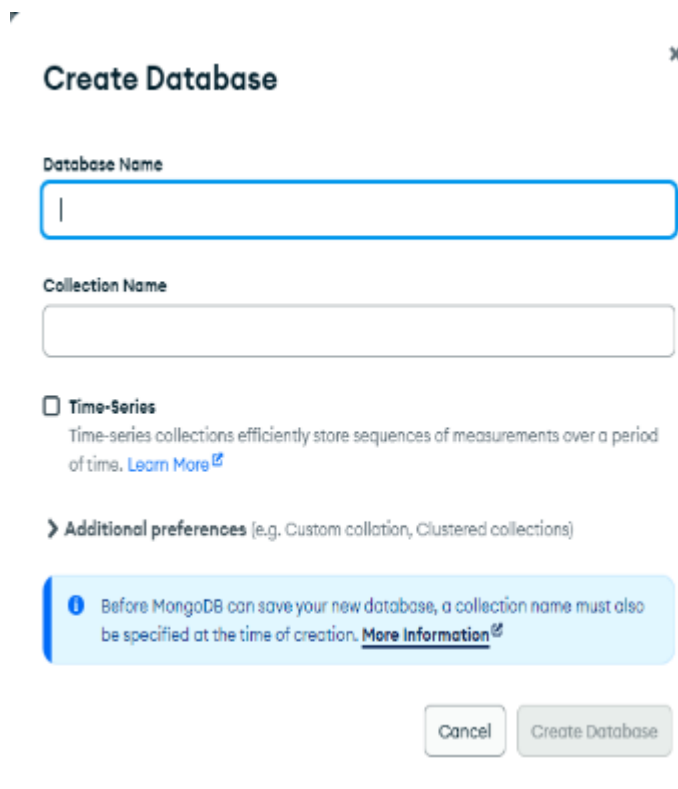


Рисунок 3.10 – Вікно створення бази даних

Перейдемо до детального розгляду створеної нами колекції. Інтерфейс користувача надає широкий спектр інструментів для роботи з даними, представлених у вигляді вкладок. Кожна вкладка відповідає за певний тип операцій над даними: перегляд документів, агрегація, побудова схем, індексація та інші. Зокрема, вкладка «Documents» дає змогу виконувати основні операції з документами, такі як імпорт та експорт. Центральна частина інтерфейсу призначена для виконання запитів до бази даних.

Розробники передбачили інтуїтивно зрозумілий інтерфейс для виконання операцій імпорту та експорту даних. Зокрема, в лівій частині вікна розташована кнопка, що активує контекстне меню, яке надає користувачеві два основних варіанти імпорту даних: з файлів формату JSON або CSV, а також шляхом безпосереднього введення даних у документ. Аналогічно, функція експорту даних підтримує різноманітні формати збереження (рис. 3.11).

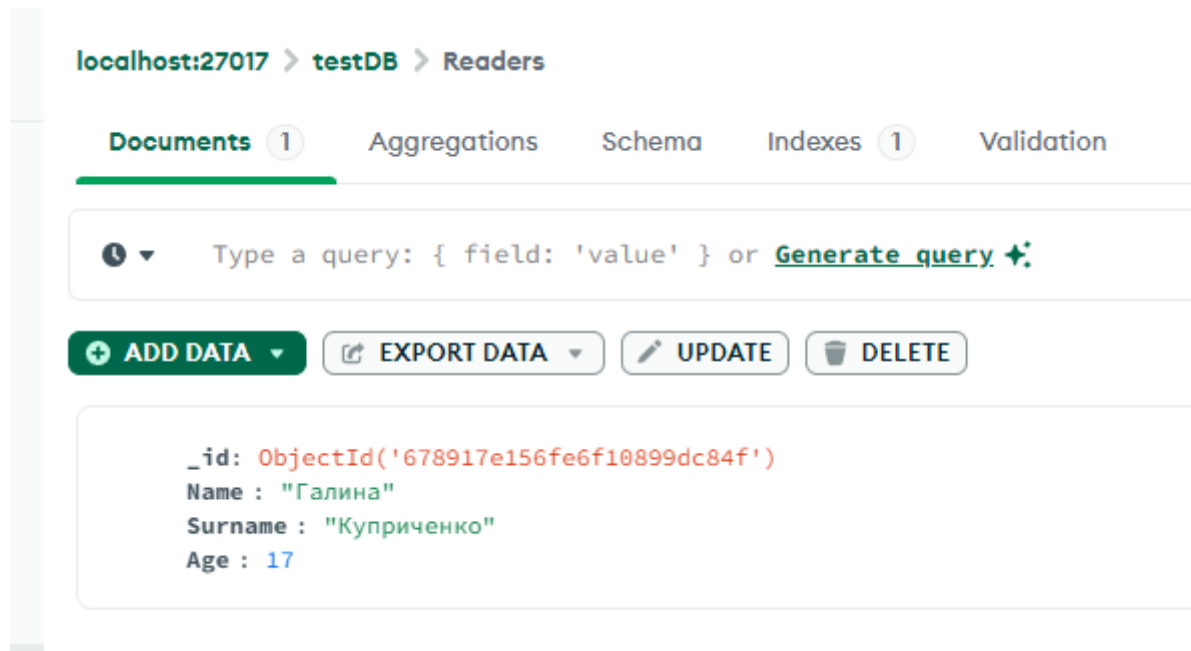


Рисунок 3.11 – Кнопка імпорту файлів

Автоматизація процесу формування запитів за допомогою AI-асистентів значно спрощує взаємодію з базою даних (рис. 3.12).



Рисунок 3.12 – Виклик AI-асистента для створення запиту

Функція пошуку документів за заданими критеріями, реалізована за допомогою оператора `find`, дає змогу ефективно витягувати необхідну інформацію з бази даних. Результати пошуку можуть бути представлені в різних форматах, що забезпечує гнучкість у візуалізації даних. Перелік доступних баз даних відображається в лівому меню, що надає можливість користувачеві швидко перемикатися між різними проєктами.

Для модифікації параметрів системи необхідно перейти до розділу налаштувань. Це можна здійснити шляхом активації контекстного меню, доступного у верхній частині інтерфейсу користувача. Вибравши пункт «Редагувати» та підпункт «Налаштування», користувач отримує доступ до діалогового вікна налаштувань (рис. 3.13), які дозволяють налаштувати різноманітні параметри системи відповідно до вимог користувача.

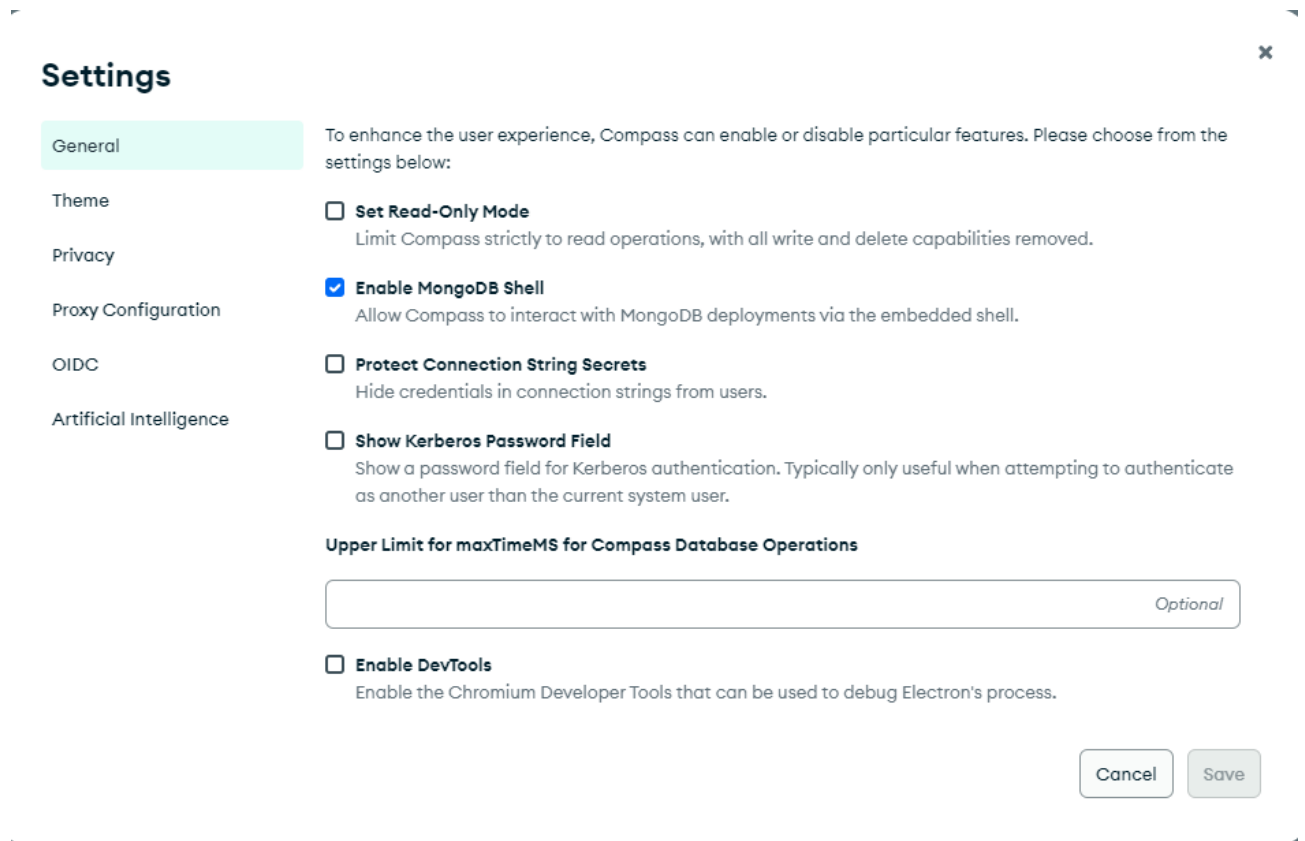


Рисунок 3.13 – Вікно налаштувань GUI Compass

Розділ налаштувань надає широкі можливості для конфігурування системи. Зокрема, користувач може активувати або деактивувати інтерактивну оболонку MongoDB (Mongo Shell), що дозволяє виконувати адміністративні завдання та взаємодіяти з базою даних безпосередньо. Крім того, доступні опції для встановлення різних рівнів доступу, таких як режим лише для читання, що обмежує можливості користувача до перегляду даних. Також передбачені засоби для забезпечення безпеки з'єднання, що дає змогу захистити дані від несанкціонованого доступу.

3.4.2 Посилання. У реляційних базах даних для встановлення зв'язку між даними з різних таблиць використовуються зовнішні ключі. Аналогічний механізм існує і в NoSQL базі даних MongoDB, де також можна встановлювати посилання між колекціями.

У даному випадку буде розглянуто приклад встановлення зв'язку між колекцією Gadgets, що містить інформацію про різні гаджети, та колекцією Instructions, яка містить перелік інструкцій до цих гаджетів.

Для демонстрації цього прикладу створимо нову базу даних GadgetDB в MongoDB Compass (рис. 3.14).



Рисунок 3.14 – База даних GadgetDB з колекцією Gadgets

Далі відкриємо будь-який текстовий редактор (у цьому випадку Notepad++) і створимо json-файл з інформацією про гаджети (рис. 3.15).

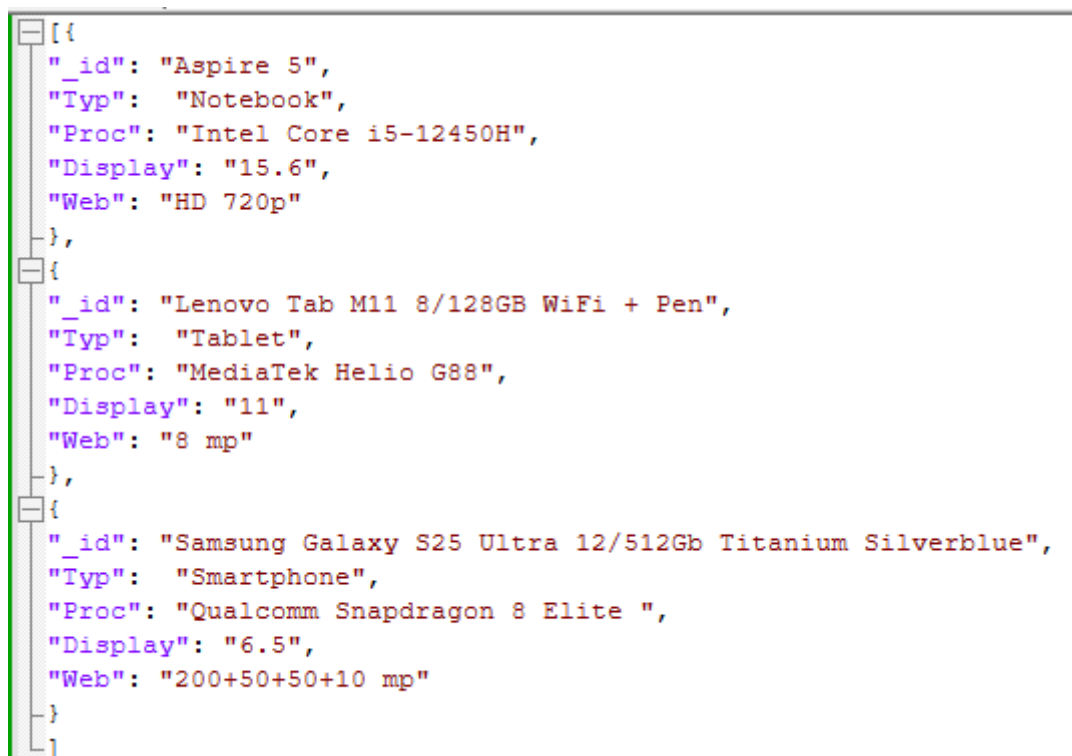


Рисунок 3.15 – Зміст json-файлу з інформацією про гаджети

Імпортуємо цей файл в колекцію Gadgets. Для цього в лівій частині MongoDB Compass курсор повинен стояти на назві колекції і натиснути кнопку “Add Data”, обрати варіант з імпортом json-файлу і вибрати потрібний файл.

Іншу колекцію створимо вручну. Для цього потрібно поставити курсор на назву бази даних і натиснути кнопку з «+» (рис. 3.16).

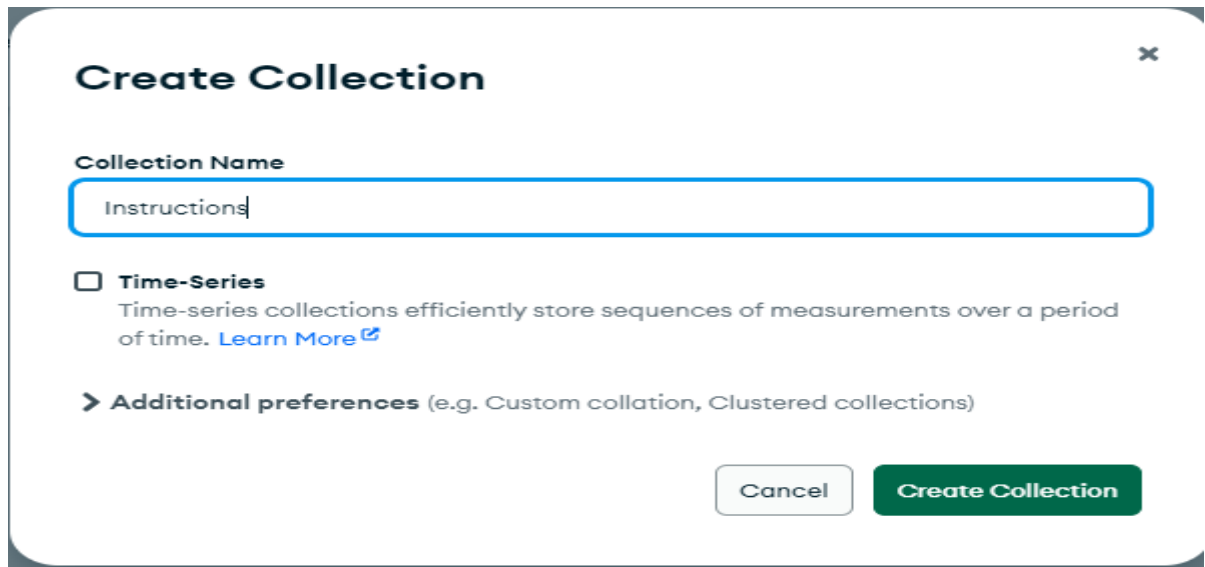


Рисунок 3.16 – Створення нової колекції в MongoDB Compass

Тепер потрібно додати документи. Для цього знов використаємо кнопку “Add Data”, але використаємо варіант «додати документ». На рис. 3.17 показано створення інструкції до гаджета з `_id`=«Aspire 5».



Рисунок 3.17 – Приклад додавання документа у MongoDB

Тепер перейдемо у консоль MongoDB Compass і зробимо два запити:

```

> instruction = db.Instructions.findOne()
< {
  _id: ObjectId('679ba9761139006c0611544a'),
  Name: 'Instruction for Aspire 5',
  Gadget: 'Aspire 5',
  Content: 'Dfgdtykuyoyfoif...'
}

> db.Gadgets.findOne({_id: instruction.Gadget})
< {
  _id: 'Aspire 5',
  Typ: 'Notebook',
  Proc: 'Intel Core i5-12450H',
  Display: '15.6',
  Web: 'HD 720p'
}

```

Таким чином ми вручну зв'язали два документа із різних колекцій.

Розглянемо інший підхід. Використовуючи функціональність DBRef, можна встановити автоматичне зв'язування між документами. Подивимося на прикладі застосування цієї функціональності. Спочатку вставимо новий документ до колекції Gadgets:

```

> Apple = db.Gadgets.insertOne({Name : "Apple iPad 10.9", "Typ": "Tablet"})
< {
  acknowledged: true,
  insertedId: ObjectId('679bb239a74fca56d82cb422')
}

```

Зверніть увагу, що ми отримуємо результат додавання до змінної Apple. При додаванні нового документа генерує `_id`, який ми можемо отримати за допомогою властивості `insertedId`, введених у результати функції.

Далі виконаємо:

```

instruction2 = ({Name: "Instruction for Apple iPad 10.9", Gadget: { "$ref" : "Gadgets", "$id" : Apple.insertedId}})
db.Instructions.insertOne(instruction2)

```

Формальний синтаксис DBRef наступний:

{ «\$ref»: назва_колекції, «\$id»: значення [, «\$db»: назва_бд]},

перший параметр \$ref вказує на колекцію, де зберігається пов'язаний документ. Другий параметр вказує на значення, яке буде представляти щось типу зовнішнього ключа. Третій необов'язковий параметр вказує базу даних.

Цей підхід дає змогу ефективно організувати дані та забезпечити їх цілісність, оскільки зміни в одній колекції можуть автоматично відображатися на пов'язаних даних в іншій колекції.

3.4.3 Проєктування структури даних в MongoDB для ефективного виконання запитів. Розглянемо приклад проєктування студентської соціальної мережі, вимоги до якої наведені нижче:

студент може мати один або декілька облікових записів;

студент може бути у будь-якій кількості груп цієї мережі;

студент може вибрати будь-яку назву для будь-якої групи, в яку він входить.

Для проєктування такого завдання може бути декілька підходів. Ознайомимося спочатку із стратегією без вбудовування. Розглянемо структуру нашої моделі даних за умови відсутності інтеграції даних. У нас є студент **Student** із прізвищем, іменем і номером залікової книги:

```
Student
{
  _id: Student_ID,
  name: "Богдан",
  surname: "Хмельницький",
  grade_book: "CR-000"
}
```

У нас є колекція нікнеймів з паролями Nicknames, в якій кожен документ містить нікнайм, пароль та прив'язку до конкретного студента:

```
Nicknames
{
  _id: Nickname_ID,
  Nickname: "Hmelia",
  Password: "qwrty",
  Student_ID: "001"
}
```

У нас є групи Groups, будь-яка з яких містить лише ідентифікатор групи (вона, звичайно, може містити й інші дані, але ми спеціально опустимо цей момент, щоб сконцентруватися на підписках):

```
GROUPS

{
  _id: GROUP_ID
}
```

Окрім того, система містить сутність «Підписка» (Membership), що репрезентує собою об'єднання користувачів у групи. Кожна підписка характеризується унікальною назвою, визначеною користувачем, та містить посилання на ідентифікатор (нікнейм), що використовується для отримання інформаційної розсилки в контексті відповідної групи:

```
Memberships

{
  _id: MEMBERSHIP_ID,
  student: Student_ID,
  Group: GROUP_ID,
  nickname: Nickname_ID_ID,
  group_name: "Database"
}
```

Хоча розроблена модель даних має переваги у простоті та зручності використання в реляційній базі даних, вона не враховує специфіку документо-орієнтованих баз даних, таких як MongoDB. Це призводить до ускладнення процедури отримання даних, особливо у випадку, коли необхідно отримати інформацію про членів групи за відомим нікнаймом та назвою групи.

Для отримання необхідних даних необхідно виконати наступні кроки:

Пошук Student_ID у колекції «Nicknames» за відомим нікнеймом.

Пошук GROUP_ID у колекції «Memberships» за отриманим Student_ID та назвою групи.

Отримання списку підписок за GROUP_ID у колекції «Memberships».

Отримання списку нікнеймів з колекції «Nicknames» за Nickname_ID кожної підписки.

Очевидно, що така процедура є неефективною та потребує оптимізації.

Тепер розглянемо випадок, коли всі дані вбудовані в один документ, що є вказівкою на стратегію із вбудовуванням. Для цього візьмемо всі Підписки Групи та вбудуємо їх у модель Групи. Також у кожен Підписку вбудуємо дані про студента та його нікнейм:

```
{
  _id: GROUP_ID,
  memberships: [{
    Nickname: "Hmelia",
    name: "Богдан",
    surname: "Хмельницький",
    grade_book: "CR-000",
    student_Nicknames: ["Bogdan", "Bodia", ...],
    group_name: "Database"
  }, ...]
}
```

Перевага інтеграції пов'язаних даних в один документ полягає у спрощенні окремих запитів. Зокрема, запит, описаний у попередній частині статті (отримання нікнеймів інших членів групи за відомим нікнеймом учасника та назвою групи), зводиться до наступних кроків:

Пошук групи у колекції «Groups» за умовою збігу group_name з назвою групи та наявності відомого нікнейму в масиві student_Nicknames:.

Вилучення інших нікнеймів з отриманого документа.

Очевидно, що такий підхід є значно ефективнішим. Проте, слід враховувати, що зміна імені або прізвища користувача потребуватиме оновлення відповідних даних у кожній вбудованій підписці кожної групи, де цей користувач є учасником. Аналогічно, додавання або видалення нікнейму з масиву student_Nicknames також потребуватиме значних зусиль.

Зазначені аспекти свідчать про специфічний характер даної моделі: вона є оптимальною для виконання конкретних запитів (завдяки попередній інтеграції даних), але може призвести до значних труднощів у довгостроковій перспективі через складність підтримки

Рекомендований підхід до проектування моделі даних полягає у первинному розгляді варіанту без вбудовування. Після формування

попередньої моделі слід ідентифікувати випадки, де вбудовування є доцільним. Як правило, це стосується відносин типу «один-до-багатьох».

Наприклад, кілька нінеймів з колекції «Nicknames» належать одному передплатнику (який також є учасником моделі «Memberships») та, зазвичай, не змінюються часто. Тому їх доцільно об'єднати в масив та інтегрувати до моделі «Memberships», що частково відповідає ментальній моделі.

Кожна підписка пов'язана з конкретним передплатником та групою, що дозволяє інтегрувати підписки як до моделі «Memberships», так і до моделі «GROUPS». У таких ситуаціях необхідно враховувати модель доступу до даних та розмір вбудованих даних.

Очікується, що кількість груп, на які студент може бути підписаним, навряд чи перевищить 1000, так само як і кількість передплатників у групі. У цьому випадку кількісні показники не є визначальними. Однак, модель доступу до даних вказує на необхідність відображення всіх передплат конкретної особи. Для спрощення запиту підписки інтегруються до моделі «Memberships».

Перевага даного підходу полягає в тому, що список нікнеймів передплатника знаходиться в моделі передплатника, а в передплаті використовується один з нікнеймів цього списку. У разі необхідності зміни або видалення нікнейму, це можна зробити в одному місці:

```
{
  _id: Student_ID,
  name: "Богдан",
  surname: "Хмельницький",
  grade_book: "CR-000"
  Nickname: ["Hmelia", "Bogdan", "Bodia", ...],
  memberships: [{
    Nickname: "Hmelia",
    group_name: "Database",
    group: GROUP_ID
  }, ...]
}
```

Окрім моделі «Передплатник», існує також модель «Група», ідентична тій, що була описана в контексті стратегії «без вбудовування».

Запит, який ми розглядали раніше, тепер має наступний вигляд:

У колекції «Student» знаходимо передплатника за нікнеймом, серед підписок якого є підписка з шуканою назвою.

Використовуючи ідентифікатор групи (GROUP_ID) знайденої підписки, знаходимо в колекції «Student» інших передплатників цієї групи та отримуємо їхні нікнейми безпосередньо з підписки.

Описаний підхід є майже таким же простим, як і у випадку з вбудованими даними, але тепер модель даних є більш комплексною та зручною в обслуговуванні.

3.4.4 Індеси. У контексті роботи з невеликими колекціями документів пошук необхідної інформації, як правило, не викликає значних труднощів. Проте, зі збільшенням обсягу колекцій до мільйонів документів, вибірка даних за заданим полем може стати ресурсомісткою операцією, що критично впливає на час виконання завдання. У таких ситуаціях ефективним рішенням є використання індесів.

Індеси забезпечують впорядкування даних за певним полем, що сприяє прискоренню пошуку. Наприклад, якщо у вашій задачі або додатку пошук за полем «name» є пріоритетним, індексація колекції за цим полем значно оптимізує процес отримання необхідних даних:

```
testDB> db.Authors.createIndex({"name":1})
```

Отже, індекс для поля «name» встановлюється за допомогою методу createIndex. MongoDB підтримує до 64 індесів на одну колекцію.

Слід зазначити, що просте визначення індексу для колекції, наприклад, db.users.createIndex({«name» : 1}), не гарантує унікальності значень ключа «name». До колекції і надалі можна буде додавати документи з однаковими значеннями цього поля.

Для забезпечення унікальності значень ключа та запобігання додаванню документів з однаковими значеннями, необхідно встановити прапор unique:

```
testDB> db.Authours.createIndex({"name":1}, {"unique":true}).
```

Тепер, якщо ми спробуємо додати в колекцію два документи з тим самим значенням name, то ми отримаємо помилку.

У той же час тут є свої тонкощі. Так, документ може не мати ключа `name`. У цьому випадку для документа, що додається, автоматично створюється ключ `name` зі значенням `null`. Тому при додаванні другого документа, в якому не визначено ключ `name`, буде викинуто виняток, оскільки ключ `name` зі значенням `null` вже є у колекції.

Також можна задати унікальний індекс одразу для двох полів:

```
testDB> db.Authors.createIndex({"name":1, "age": 1},{ "unique":true})
```

Однак у цьому випадку всі документи, що додаються, повинні мати унікальні значення для обох полів.

Для того, щоб проіндексувати поле в MongoDB Compass потрібно обрати потрібну колекцію в базі даних і перейти на закладку **Indexes** (рис. 3.18).

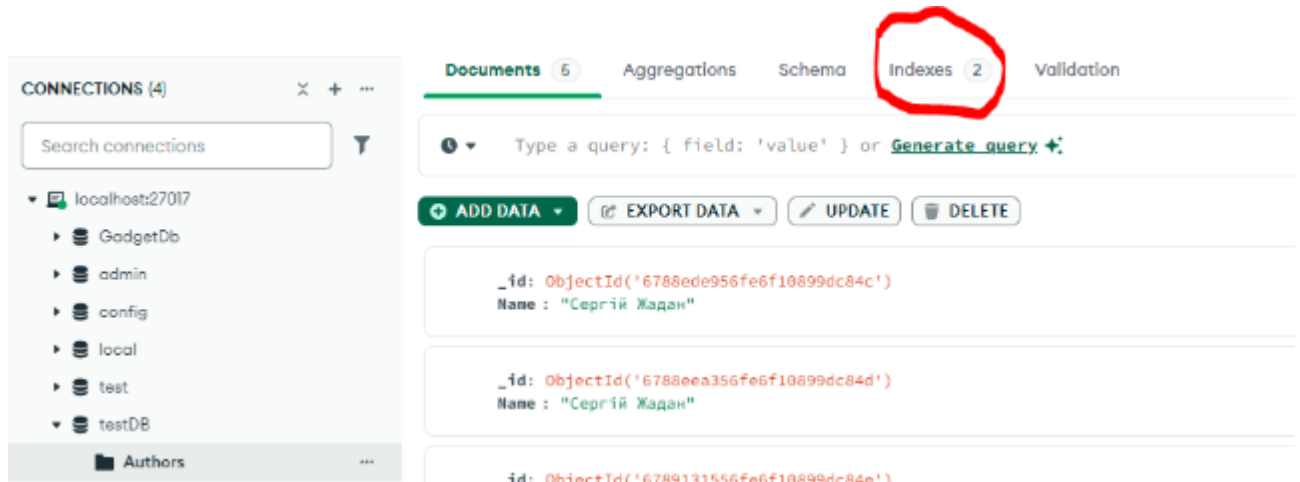


Рисунок 3.18 – Закладка Indexes в MongoDB Compass

Натиснути кнопку **Create Index** (рис. 2.19).

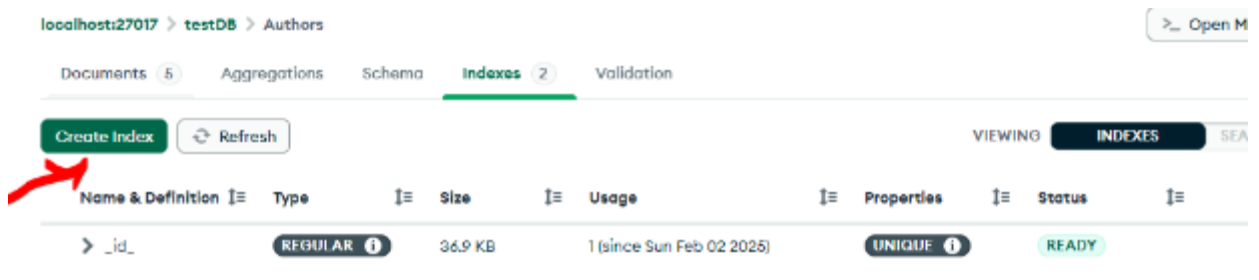


Рисунок 3.19 – Кнопка Create index

У вікні, яке відкриється (рис. 3.20), обрати потрібне поле і натиснути **Create Index**.

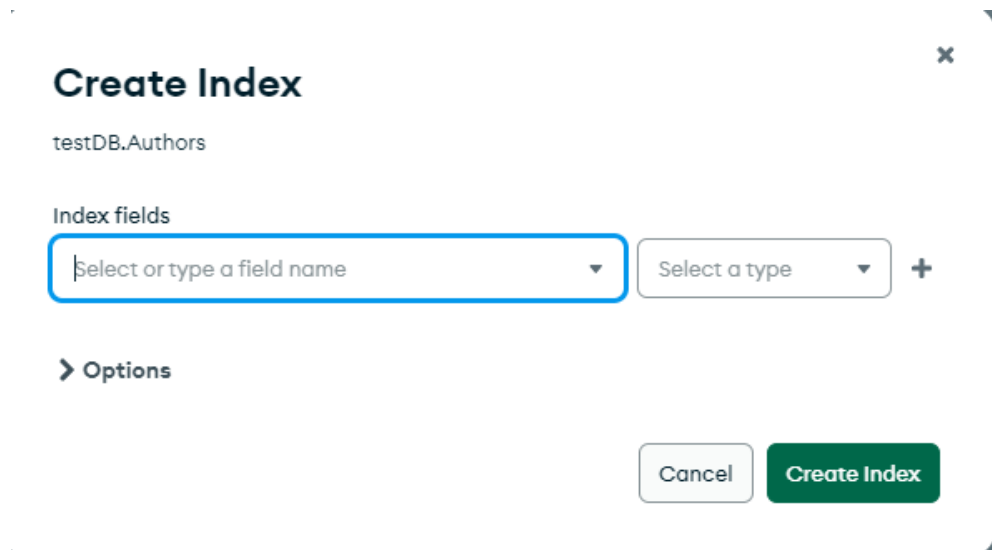


Рисунок 3.20 – Вікно створення індексів в MongoDB Compass

Також тут можна вказати, яким чином буде сортуватися інформація (рис. 3.21).

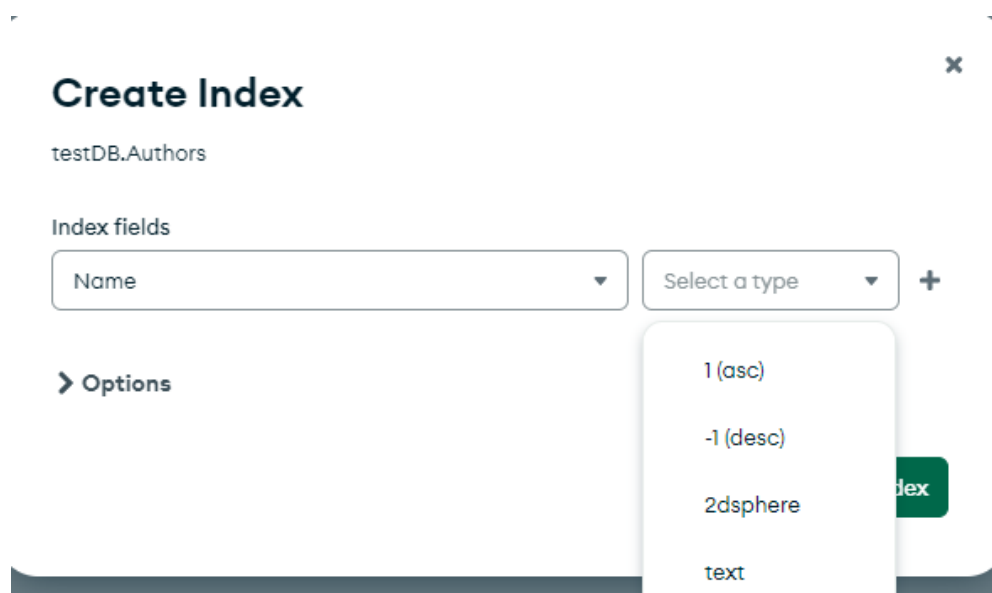


Рисунок 3.21 – Встановлення правил сортування

Коли активувати альтернативу Options, то відкриється вікно для встановлення додаткових параметрів індексування (рис. 3.22).

Create Index

testDB.Authors

Index fields

Name 1 (asc) +

▼ Options

☐ **Create unique index**
A unique index ensures that the indexed fields do not store duplicate values; i.e. enforces uniqueness for the indexed fields.

☐ **Index name**
Enter the name of the index to create, or leave blank to have MongoDB create a default name for the index.

☐ **Create TTL**
TTL indexes are special single-field indexes that MongoDB can use to automatically remove documents from a collection after a certain amount of time or at a specific clock time.

☐ **Partial Filter Expression**

Cancel Create Index

Рисунок 3.22 – Встановлення різних параметрів для індексів

Усі індекси бази даних зберігаються у системній колекції `indexes`. Для звернення до неї ми можемо використовувати функцію `getIndexes`, наприклад, щоб вивести всю інформацію про індекси для конкретної колекції:

```
testDB> db.Authors.getIndexes()
[
  { v: 2, key: { _id: 1 }, name: '_id_' },
  { v: 2, key: { name: 1 }, name: 'name_1', hidden: true }
]
```

Питання для самоперевірки

Що таке індекс? Який індекс в `mongoDB` по замочуванню?

Які параметри для індексів можливо встановити в `mongoDB`?

Які нюанси роботи з унікальними індексами?

Завдання

Сайт Kaggle (<https://www.kaggle.com/>) надає доступ до великої кількості відкритих наборів даних, які можна використовувати для моделей машинного навчання, досліджень та експериментів. Завдання полягає у наступному:

- скачати вказаний у варіантах набір даних;
- розібратися що це за дані;
- завантажити їх у колекцію mongoDB;
- проіндексовати два поля з різними параметрами;
- обґрунтувати вибір поля і параметрів;
- написати по одному запиту до цих полів.

Варіант	Опис завдання
1	https://www.kaggle.com/datasets/anandshaw2001/netflix-movies-and-tv-shows
2	https://www.kaggle.com/datasets/himelsarder/road-accident-survival-dataset
3	https://www.kaggle.com/datasets/ankushpanday1/alzheimers-prediction-dataset-global
4	https://www.kaggle.com/datasets/atifaliak/youtube-comments-dataset
5	https://www.kaggle.com/datasets/iabhishekoofficial/mobile-price-classification

3.5 Агрегація

Агрегація в MongoDB – це процес обробки множини документів для отримання результату у вигляді одного документа. Цей результат є обчисленням, виконаним над даними з усіх вхідних документів. Іншими словами, агрегація дозволяє нам групувати дані, виконувати різноманітні операції над ними та отримувати підсумкові результати.

Агрегація дозволяє нам проводити глибокий аналіз даних, обчислюючи такі показники, як суму, середнє значення, мінімум, максимум, кількість елементів тощо. За допомогою агрегації ми можемо створювати різноманітні звіти, наприклад, підраховувати кількість замовлень за певний період, визначати найпопулярніші продукти тощо. Агрегація дозволяє об'єднати декілька операцій в один запит, що підвищує ефективність роботи з базою даних.

Агрегація в MongoDB виконується за допомогою конвеєрів (pipelines). Конвеєр – це послідовність стадій, кожна з яких виконує певну операцію над даними. Результат однієї стадії передається на вхід наступної.

Основні стадії агрегації:

- `$match`: Фільтрує документи за заданими критеріями.
- `$project`: Вибирає певні поля з документів або створює нові поля.
- `$group`: Групує документи за певним полем і виконує обчислення для кожної групи.
- `$sort`: Сортує документи за вказаним полем.
- `$limit`: Обмежує кількість документів у результаті.
- `$skip`: Пропускає певну кількість документів у результаті.

Розглянемо приклади. Припустимо, колекція `Student` використовується для зберігання даних про студентів (рис. 3.23).

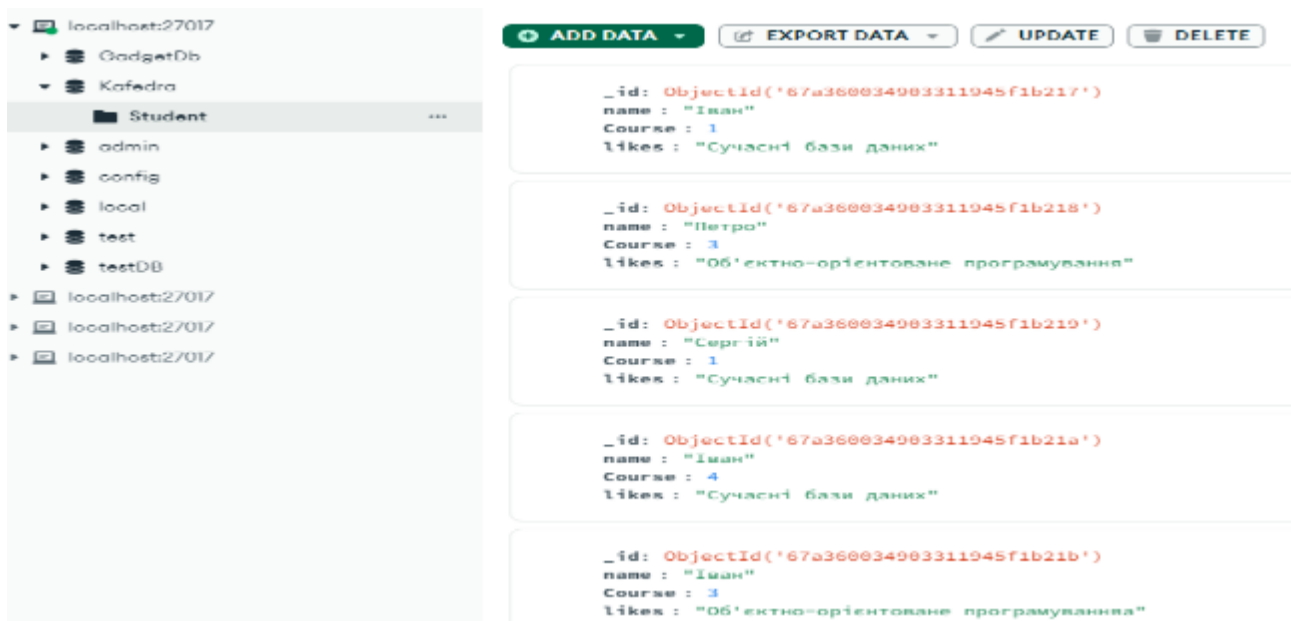


Рисунок 1.7 – Вміст колекції Student

Стандартна вибірка:

```
> db.Student.find({name: "Іван"})
< {
  _id: ObjectId('67a360034903311945f1b217'),
  name: 'Іван',
  Course: 1,
  likes: 'Сучасні бази даних'
}
{
  _id: ObjectId('67a360034903311945f1b21a'),
  name: 'Іван',
  Course: 4,
  likes: 'Сучасні бази даних'
}
{
  _id: ObjectId('67a360034903311945f1b21b'),
  name: 'Іван',
  Course: 3,
  likes: "Об'єктно-орієнтоване програмування"
}
```

```
{
  _id: ObjectId('67a360034903311945f1b21e'),
  name: 'Іван',
  Course: 2,
  likes: "Об'єктно-орієнтоване програмування"
}
```

буде еквівалентна:

```
> db.Student.aggregate({$match: {name: "Іван"}})
< {
  _id: ObjectId('67a360034903311945f1b217'),
  name: 'Іван',
  Course: 1,
  likes: 'Сучасні бази даних'
}
{
  _id: ObjectId('67a360034903311945f1b21a'),
  name: 'Іван',
  Course: 4,
  likes: 'Сучасні бази даних'
}
{
  _id: ObjectId('67a360034903311945f1b21b'),
  name: 'Іван',
  Course: 3,
  likes: "Об'єктно-орієнтоване програмування"
}
{
  _id: ObjectId('67a360034903311945f1b21e'),
  name: 'Іван',
  Course: 2,
  likes: "Об'єктно-орієнтоване програмування"
}
```

Проте, другий приклад дає більше можливостей, адже ми можемо будувати ланцюжок обробки даних, перераховуючи оператори через кому. Оператор `$sort` використовується для сортування, наприклад:

```
> db.Student.aggregate({$match: {name: "Іван"}}, {$sort: {Course: 1}})
< {
  _id: ObjectId('67a360034903311945f1b217'),
  name: 'Іван',
  Course: 1,
  likes: 'Сучасні бази даних'
}
{
  _id: ObjectId('67a360034903311945f1b21e'),
  name: 'Іван',
  Course: 2,
  likes: "Об'єктно-орієнтоване програмування"
}
{
  _id: ObjectId('67a360034903311945f1b21b'),
  name: 'Іван',
  Course: 3,
  likes: "Об'єктно-орієнтоване програмування"
}
{
  _id: ObjectId('67a360034903311945f1b21a'),
  name: 'Іван',
  Course: 4,
  likes: 'Сучасні бази даних'
}
```

Цей код вибере всіх людей з ім'ям «Іван» і відсортує їх за курсом навчання. Щоб вибрати Івана з найстаршого курсу, нам потрібно обмежити вибірку одним елементом:

```
> db.Student.aggregate({$match: {name: "Іван"}}, {$sort: {Course: -1}}, {$limit: 1} )
< {
  _id: ObjectId('67a360034903311945f1b21a'),
  name: 'Іван',
  Course: 4,
  likes: 'Сучасні бази даних'
}
```

Оператор `$project` використовується для маніпулювання з полями: він може додавати, видаляти та перейменовувати їх у документах, що надходять на

його вхід. Наступна конструкція включатиме в потік документів (відфільтрує) лише імена та вік користувачів:

```
> db.Student.aggregate({$match: {name: "Іван"}},{ $project: {name: 1, Course: 1}})
< {
  _id: ObjectId('67a360034903311945f1b217'),
  name: 'Іван',
  Course: 1
}
{
  _id: ObjectId('67a360034903311945f1b21a'),
  name: 'Іван',
  Course: 4
}
{
  _id: ObjectId('67a360034903311945f1b21b'),
  name: 'Іван',
  Course: 3
}
{
  _id: ObjectId('67a360034903311945f1b21e'),
  name: 'Іван',
  Course: 2
}
```

На вхід наступного оператора потраплять всі документи лише з двома полями, інших полів у потоці не буде (за винятком поля `_id`, щоб його виключити, потрібно спеціально вказати `_id: 0`). Цифра 1 включає, а цифра 0 виключає передачу поля.

Для демонстрації наступного оператора змінимо колекцію, так щоб поле `likes` зберігало масиви. Масив в MongoDB – це впорядкована колекція значень, які можуть бути будь-якого типу: рядки, числа, булеві значення, об’єкти або навіть інші масиви (створюючи таким чином багатовимірні структури). Це дуже гнучкий інструмент, який дозволяє зберігати різноманітні типи даних в одному документі. Нехай поле `likes` означає, які дисципліни подобаються тому чи іншому студенту:


```

] [
] [
  {
    "name" : "Іван",
    "Course":1,
    "likes" : ["Сучасні бази даних", "Алгоритми і структури даних"]
  },
  {
    "name" : "Петро",
    "Course":3,
    "likes" : ["Об'єктно-орієнтоване програмування", "Сучасні бази даних"]
  },
  {
    "name" : "Сергій",
    "Course":1,
    "likes" : ["Сучасні бази даних", "Алгоритми і структури даних"]
  },
  {
    "name" : "Іван",
    "Course":4,
    "likes" : ["Сучасні бази даних", "Алгоритми і структури даних"]
  },
]

```

Оператор \$unwind дозволяє «розгортати» вкладені масиви на кожен елемент вибірки документів. Застосуємо оператор \$unwind:

```

> use Kafedra
< switched to db Kafedra
> db.Student.aggregate({$unwind: "$likes"})
< {
  _id: ObjectId('67a3780d4903311945f1b222'),
  name: 'Іван',
  Course: 1,
  likes: 'Сучасні бази даних'
}
{
  _id: ObjectId('67a3780d4903311945f1b222'),
  name: 'Іван',
  Course: 1,
  likes: 'Алгоритми і структури даних'
}
{
  _id: ObjectId('67a3780d4903311945f1b223'),
  name: 'Петро',
  Course: 3,
  likes: 'Об'єктно-орієнтоване програмування'
}

```

Якщо ми хочемо знайти найпопулярнішу дівчину, нам достатньо згрупувати вибірку за полем `likes`. Для цього використовується наступний оператор `$group`. Для зручності додамо до вибірки ще одне поле, заповнене цифрою 1 (так буде простіше сумувати):

```
> db.Student.aggregate({$unwind: "$likes"}, {$project: {name: 1, likes: 1, count: {$add: [1]}}})
< {
  _id: ObjectId('67a3780d4903311945f1b222'),
  name: 'Іван',
  likes: 'Сучасні бази даних',
  count: 1
}
{
  _id: ObjectId('67a3780d4903311945f1b222'),
  name: 'Іван',
  likes: 'Алгоритми і структури даних',
  count: 1
}
{
  _id: ObjectId('67a3780d4903311945f1b223'),
  name: 'Петро',
  likes: "Об'єктно-орієнтоване програмування",
  count: 1
}
{
  _id: ObjectId('67a3780d4903311945f1b223'),
  name: 'Петро',
  likes: 'Сучасні бази даних',
  count: 1
}
```

Це дозволить нам використовувати оператор агрегування `$sum`. Тобто тепер ми просто додаємо в поле `number` значення поля `count` кожен раз і групуємо всю вибірку по полю `likes`, що містить назву дисципліни:

```
> db.Student.aggregate({$unwind: "$likes"}, {$project:
{name: 1, likes: 1, count: {$add: [1]}}},
{$group: {_id: "$likes", number: {$sum: "$count"}}})
```

Результат буде наступний:

```
< {
  _id: 'Database',
  number: 1
}
{
  _id: "Об'єктно-орієнтоване програмування",
  number: 2
}
{
  _id: 'Сучасні бази даних',
  number: 9
}
{
  _id: 'Алгоритми і структури даних',
  number: 3
}
{
  _id: "Об'єктно-орієнтоване програмування",
  number: 3
}
```

Залишилося відсортувати і обмежити вивід лише одним документом:

```
> db.Student.aggregate({$unwind: "$likes"},
{$project: {name: 1, likes: 1, count: {$add: [1]}}},
{$group: {_id: "$likes", number: {$sum: "$count"}}},
{$sort: {number: -1}}, {$limit: 1})
```

Цей код знаходить найпопулярнішу дисципліну серед студентів:

```
< {
  _id: 'Сучасні бази даних',
  number: 9
}
```

Для демонстрації наступних прикладів створимо колекцію Group і завантажимо до неї наступні дані:

```

{
    "name" : "Іванов Іван",
    "Course":1,
    "Grade" :[{ "Сучасні бази даних": 60}, {"Алгоритми і структури даних": 85}, {"Об'єктно-орієнтоване програмування": 90}],
    "holidays": [1,4],
    "staff": { "likes" : ["Сучасні бази даних","Алгоритми і структури даних"], "dislikes":["Об'єктно-орієнтоване програмування"]}
},

{
    "name" : "Петренко Костянтин",
    "Course":1,
    "Grade" :[{ "Сучасні бази даних": 100}, {"Алгоритми і структури даних": 95}, {"Об'єктно-орієнтоване програмування": 99}],
    "holidays": [1,4],
    "staff": { "likes" : ["Алгоритми і структури даних"], "dislikes":["Об'єктно-орієнтоване програмування", "Сучасні бази даних"]}
},

{
    "name" : "Зеленська Марина",
    "Course":1,
    "Grade" :[{ "Сучасні бази даних": 75}, {"Алгоритми і структури даних": 60}, {"Об'єктно-орієнтоване програмування": 60}],
    "holidays": [3,4],
    "staff": { "likes" : ["Сучасні бази даних"], "dislikes":["Об'єктно-орієнтоване програмування", "Алгоритми і структури даних"]}
},

{
    "name" : "Трамп Василь",
    "Course":1,
    "Grade" :[{ "Сучасні бази даних": 75}, {"Алгоритми і структури даних": 75}, {"Об'єктно-орієнтоване програмування": 75}],
    "holidays": [5,4],
    "staff": { "likes" : ["Сучасні бази даних","Алгоритми і структури даних", "Об'єктно-орієнтоване програмування"], "dislikes":[]}
},

{
    "name" : "Путін Вовка",
    "Course":1,
    "Grade" :[{ "Сучасні бази даних": 20}, {"Алгоритми і структури даних": 20}, {"Об'єктно-орієнтоване програмування": 20}],
    "holidays": [5,4],
    "staff": { "likes" : [], "dislikes":["Об'єктно-орієнтоване програмування", "Сучасні бази даних","Алгоритми і структури даних"]}
}

```

Поле name зберігає прізвище та ім'я студента, поле Grade — це масив об'єктів, що зароблені бали з дисциплін, holidays — це дні, в які студенти не були на парах, staff.like — дисципліни, які подобаються студентам, staff.dislike — не подобаються.

Тепер із колекції виведемо лише імена студентів:

```

> db.Group.aggregate({$project: {name: 1}})
< {
  _id: ObjectId('67a4d7196eab347e3b9ac5aa'),
  name: 'Іванов Іван'
}
{
  _id: ObjectId('67a4d7196eab347e3b9ac5ab'),
  name: 'Петренко Костянтин'
}
{
  _id: ObjectId('67a4d7196eab347e3b9ac5ac'),
  name: 'Зеленська Марина'
}

```

Для початку відфільтруємо потік даних і виділимо лише два поля в документах – ім'я та оцінки:

```

> db.Group.aggregate({$project: {name: 1}})
< {
  _id: ObjectId('67a4d7196eab347e3b9ac5aa'),
  name: 'Іванов Іван'
}
{
  _id: ObjectId('67a4d7196eab347e3b9ac5ab'),
  name: 'Петренко Костянтин'
}
{
  _id: ObjectId('67a4d7196eab347e3b9ac5ac'),
  name: 'Зеленська Марина'
}
{
  _id: ObjectId('67a4d7196eab347e3b9ac5ad'),
  name: 'Трамн Василь'
}

```

Далі розгорнемо масив оцінок на елементи основного масиву:

```
> db.Group.aggregate({$project: {name: 1, _id: 0, Grade: 1}}, {$unwind: "$Grade"})
< {
  name: 'Іванов Іван',
  Grade: {
    'Сучасні бази даних': 60
  }
}
{
  name: 'Іванов Іван',
  Grade: {
    'Алгоритми і структури даних': 85
  }
}
```

Основні оператори агрегації представлені в таблиці 3.5.

Таблиця 3.5. **Основні оператори агрегації**

\$project	Вибирає конкретні поля з документів або створює нові поля на основі існуючих. Це дозволяє формувати структуру вихідних даних.
\$match	Фільтрує вхідні документи за заданими критеріями, аналогічно оператору пошуку find().
\$group	Групує документи за певним полем або виразом, дозволяючи обчислювати агреговані значення (суми, середні значення, мінімуми, максимуми тощо) для кожної групи.
\$sort	Сортує документи за вказаним полем або виразом у зростаючому або спадаючому порядку.
\$limit	Обмежує кількість документів у результаті агрегації.
\$skip	Пропускає задану кількість документів у результаті агрегації.
\$unwind	Розгортає масиви, перетворюючи кожен елемент масиву в окремий документ.
\$lookup	Виконує з'єднання колекцій, подібно до з'єднання таблиць у реляційних базах даних.
\$addFields	Додає нові поля до документів.

\$replaceRoot	Замінює кореневий документ результатом виразу.
\$count	Підраховує кількість документів, що відповідають критеріям.
\$avg	Обчислює середнє значення числового поля.
\$sum	Обчислює суму числового поля.
\$min	Знаходить мінімальне значення числового поля.
\$max	Знаходить максимальне значення числового поля.
\$first	Вибирає перший документ в кожній групі.
\$last	Вибирає останній документ в кожній групі.
\$push	Додає елементи до масиву.
\$addToSet	Додає унікальні елементи до масиву.

Питання для самоперевірки

У чому полягає процес агрегації?

Які є основні стадії агрегації?

Який оператор дозволяє «розгортати» вкладені масиви на кожен елемент вибірки документів? Я

кий оператор обмежує кількість документів у результаті агрегації?

Який оператор виконує з'єднання колекцій?

Завдання

Написати запити:

Варіант	Опис завдання
1	Структура: назва торгової організації, адреса, ПІБ директора, податковий номер, постачальники (може бути декілька) 1.Порахувати всіх постачальників. 2.Вивести найбільш активного постачальника (який співпрацює з найбільшою кількістю торговельних організацій)

	3.Згрупувати торгівельні організації за постачальником, вивести лише назви торгівельних організацій.
2	Структура: ПІБ, посада (може бути декілька), номер відділу (може бути декілька), в якому працює, стать, адреса, дата народження. 1.Згрупувати співробітників за посадами. Вивести лише їх ПІБ. 2.Порахувати співробітників для конкретної посади. 3.Порахувати всіх дітей у всіх співробітників.
3	Структура: ПІБ, паспортні дані, стать, вік, діти (може бути декілька) 1.Змінити колекцію, додавши вкладену структуру із назвою дитячого закладу і ціною, куди дитина ходить. Додати п'ять документів. 2.Згрупувати громадян, за назвою дитячих закладів. 3.Порахувати цих громадян для кожного закладу.
4	Структура: назва туру, країна (може бути декілька), міста (може бути декілька), тип пересування (може бути декілька), тип харчування, ціна туру, тип проживання 1.Згрупувати тури по містам та порахувати їх середню ціну. 2.Вивести найбільш популярне місто серед туристів.. 3.Порахувати всі міста, де є тури.
5	Структура: прізвище, ім'я, по батькові, адреса, місто, телефон (може бути декілька). 1.Змінити колекцію, додавши вкладену структуру: назва тарифу мобільного оператора та ціну. Додати 5 документів. 2.Розгорнути документи по операторам зв'язку. 3.Порахувати порахувати середню ціну тарифа для кожного оператора.

3.6 C# та MongoDB

Фреймворк .NET і, зокрема, мова C# завдяки наявності драйвера також можуть працювати з MongoDB. Для встановлення всіх необхідних файлів та бібліотек драйвера у проєкт простіше використовувати пакетний менеджер NuGet (рис. 3.24).

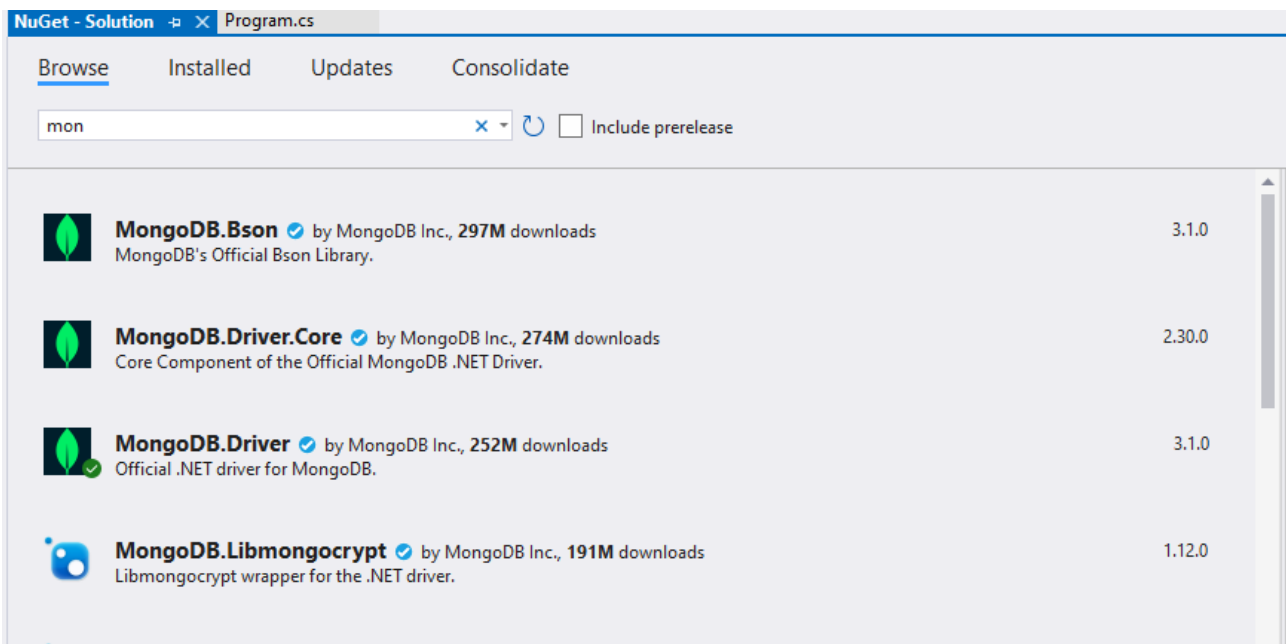


Рисунок 3.24 – Встановлення драйверу для MongoDB

Бібліотека MongoDB.BSON.dll є ключовим компонентом драйвера MongoDB для платформи .NET. Її основне призначення – забезпечити можливість роботи з типом даних BSON (Binary JSON), який використовується для зберігання та обміну даними у базі даних MongoDB. MongoDB.BSON.dll відповідає за перетворення об'єктів .NET у формат BSON для зберігання в MongoDB та навпаки (перетворення даних з формату BSON в об'єкти .NET для використання в програмі). Бібліотека надає класи та методи для створення, читання та маніпулювання документами BSON, а також для роботи з різноманітними типами даних, які підтримуються BSON (наприклад, ObjectId, DateTime, Int32, String тощо). MongoDB.BSON.dll оптимізована для забезпечення швидкого та ефективного обміну даними між програмою на .NET та сервером MongoDB.

Бібліотека MongoDB.Driver.dll надає функціональність для взаємодії з базою даних MongoDB, включаючи встановлення з'єднання, виконання запитів та операцій, а також обробку результатів. Вона дозволяє встановлювати з'єднання з сервером MongoDB, вказуючи параметри підключення, такі як адреса сервера, порт та інші опції, а також надає методи для виконання різноманітних операцій з базою даних MongoDB, такі як створення колекцій, додавання документів, пошук документів за різними критеріями, оновлення та видалення документів. MongoDB.Driver.dll дозволяє будувати складні запити за допомогою спеціальних класів та методів, що полегшує роботу з даними та забезпечує гнучкість у вибірці необхідної інформації. Бібліотека надає засоби для отримання та обробки результатів запитів, дозволяючи перетворювати дані з формату BSON (Binary JSON), який використовується в MongoDB, в об'єкти .NET для подальшої обробки в програмі. Вона підтримує роботу з сесіями та транзакціями, що дозволяє забезпечити цілісність даних та узгодженість операцій при роботі з MongoDB.

Для ініціалізації з'єднання з базою даних MongoDB первинним інструментом є клас MongoClient. У конструктор цього класу передається рядок підключення, який містить необхідну інформацію для встановлення зв'язку з сервером бази даних:

```
string connectionString = "mongodb://localhost:27017";  
MongoClient client = new MongoClient(connectionString);  
IMongoDatabase database = client.GetDatabase("testDB");
```

Рядок підключення має такий вигляд:

mongodb://[username:password@]hostname[:port]/[database][?options].

Наприклад, при вказівці всіх параметрів рядок підключення міг би виглядати так: *mongodb://user:pass@localhost/db1?authSource=userDb* – у цьому випадку користувач з логіном user та паролем pass підключається до бази даних db1 на хості localhost. Також задається додатковий параметр authSource зі значенням userDb. Якщо порт не вказано, використовується порт 27017 за промовчанням. Після отримання клієнта, з його допомогою ми можемо безпосередньо звертатися до конкретних баз даних. Для отримання бази даних застосовується

метод `GetDatabase`, у який передається назва бази. Цей метод повертає об'єкт `IMongoDatabase`. У даному випадку йде отримання посилання на базу даних `testDB`. Отримавши об'єкт бази даних, ми можемо звертатися до конкретних колекцій та проводити різні операції з даними. Після виконання всіх необхідних операцій нам необов'язково закривати підключення, як, наприклад, у разі підключення до інших баз даних, оскільки MongoDB сама виконає всю роботу.

Отримаємо всі колекції всіх баз даних, які є на сервері:

```
using MongoDB.Driver;

namespace MongoApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string connectionString = "mongodb://localhost:27017";
            MongoClient client = new MongoClient(connectionString);
            GetCollectionsNames(client).GetAwaiter();
            Console.ReadLine();
        }
        private static async Task GetCollectionsNames(MongoClient client)
        {
            using (var cursor = await client.ListDatabasesAsync())
            {
                var dbs = await cursor.ToListAsync();
                foreach (var db in dbs)
                {
                    Console.WriteLine("У базі даних {0} є наступні колекції:",
db["name"]);

                    IMongoDatabase database =
client.GetDatabase(db["name"].ToString());

                    using (var collCursor = await database.ListCollectionsAsync())
                    {
                        var colls = await collCursor.ToListAsync();
                        foreach (var col in colls)
                        {
                            Console.WriteLine(col["name"]);
                        }
                    }
                    Console.WriteLine();
                }
            }
        }
    }
}
```

Результат виконання програми:

```

У баз? даних GadgetDb є наступн? колекц?ї:
Instructions
Gadgets

У баз? даних Kafedra є наступн? колекц?ї:
Group
Student

У баз? даних admin є наступн? колекц?ї:
system.version

У баз? даних config є наступн? колекц?ї:
system.sessions

У баз? даних local є наступн? колекц?ї:
startup_log

У баз? даних test є наступн? колекц?ї:
Employee
Work
user

У баз? даних testDB є наступн? колекц?ї:
Readers
RatingCall
Poetry
Rating
Employee
Authors

```

Крім жорсткого кодування рядків підключення в кодї можна також визначити їх у конфігураційних файлах: app.config (для десктопних додатків) і web.config (для веб-додатків) в секції connectionString:

```

<connectionStrings>
  <add name=«MongoDb» connectionString=«mongodb://localhost/testDB» />
</connectionStrings>

```

Атрибут connectionString встановлює параметри підключення, зокрема сервер – той самий localhost і назва бази даних – testDB.

Тоді на стороні сервера отримання рядка підключення з файлу виглядатиме так:

```

using MongoDB.Driver;
using System.Configuration;

namespace ConsoleApp4
{
    internal class Program
    {
        static void Main(string[] args)
        {
            string con
ConfigurationManager.ConnectionStrings[«MongoDb»].ConnectionString;
            var client = new MongoClient(con);

        }
    }
}

```

Для використання класу ConfigurationManager необхідно додати до проекту бібліотеку System.Configuration.dll (рис. 3.25).

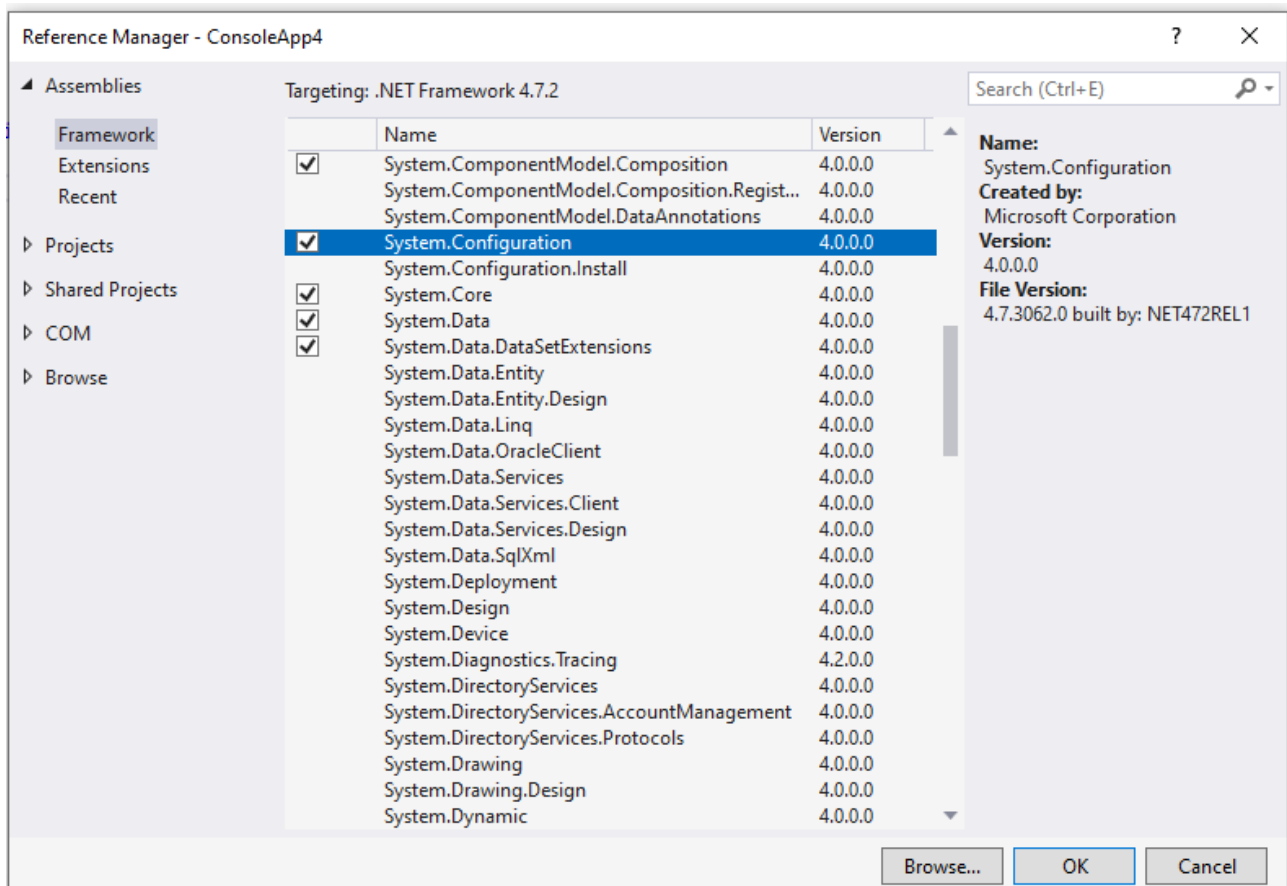


Рисунок 1.8 – Встановлення бібліотеки System.Configuration.dll

Створимо невелику консольну програму, яка виводитиме на екран усі бази даних сервера:

```
using System;
using System.Threading.Tasks;
using MongoDB.Driver;
using System.Configuration;

namespace MongoDBApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string con =
            ConfigurationManager.ConnectionStrings[«MongoDb»].ConnectionString;
            MongoClient client = new MongoClient(con);
            GetDatabaseNames(client).GetAwaiter();
            Console.ReadLine();
        }

        private static async Task GetDatabaseNames(MongoClient client)
        {
            using (var cursor = await client.ListDatabasesAsync())
```

```

    {
        var databaseDocuments = await cursor.ToListAsync();
        foreach (var databaseDocument in databaseDocuments)
        {
            Console.WriteLine(databaseDocument[«name»]);
        }
    }
}
}

```

Результатом роботи цього коду буде:

```

GadgetDb
Kafedra
admin
config
local
test
testDB

```

Хоча створення документів можна здійснювати за допомогою класу BsonDocument, але набагато простіше працювати безпосередньо через класи, які представляють дані. І драйвер MongoDB для C# надає цю можливість.

Створимо модель даних:

```

using System;
using System.Collections.Generic;
using MongoDB.Bson;

namespace MongoApp
{
    class Authors
    {
        public ObjectId Id { get; set; }
        public string Name { get; set; }
        public string Surname { get; set; }
        public int Age { get; set; }
        public Company Company { get; set; }
        public List<string> Languages { get; set; }
    }
    class Company
    {
        public string Name { get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Authors p = new Authors { Name = «Сергій», Surname = «Жадан», Age = 40 };
            p.Company = new Company { Name = «Львівська книгарня» };

            Console.WriteLine(p.ToJson());
            Console.ReadLine();
        }
    }
}

```

Тут визначено клас Authors, який представляє авторів, та клас Company, який представляє компанію, яка продає книгу автора.

Варто звернути увагу, що як ідентифікатор Id в класі Authors використовується об'єкт класу ObjectId, який визначений в бібліотеці MongoDB.Bson.dll.

Простір імен MongoDB.Bson додає ряд функціональностей до класів C#, які дозволяють використовувати об'єкти цих класів як документи.

Бібліотека MongoDB.Bson.dll додає ряд методів до класів, зокрема метод toJson, який перетворює об'єкт у формат JSON, а сам об'єкт представляє документ.

При створенні документа ми можемо скористатися як стандартним класом C#, так і класом BsonDocument, та за необхідності перейти від одного до іншого:

```
using System;
using System.Collections.Generic;
using MongoDB.Bson;
using MongoDB.Bson.Serialization;

namespace MongoApp
{
    class Authors
    {
        public ObjectId Id { get; set; }
        public string Name { get; set; }
        public string Surname { get; set; }
        public int Age { get; set; }
        public Company Company { get; set; }
        public List<string> Languages { get; set; }
    }
    class Company
    {
        public string Name { get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            BsonDocument doc = new BsonDocument
            {
                {«Name», «Сергій»},
                {«Surname», «Жадан»},
                {«Age», new BsonInt32(40)},
                { «Company»,
                    new BsonDocument{
                        {«Name», «Львівська книгарня»}
                    }
                }
            };
            Authors autor = BsonSerializer.Deserialize<Authors>(doc);
```

```

        Console.WriteLine(autor.ToJson());
        Console.ReadLine();
    }
}

```

За допомогою методу `Deserialize()` класу `BsonSerializer` з простору імен `MongoDB.Bson.Serialization` ми можемо виконати десеріалізацію з документа в об'єкт моделі `Authors`. При цьому важливо, щоб імена властивостей моделі збігалися з іменами елементів у документі (у тому числі й по регістру), інакше програмі не вдасться порівняти елементи та властивості.

Також можна виконати зворотну операцію перетворення об'єкта в `BsonDocument`:

```

using System;
using System.Collections.Generic;
using MongoDB.Bson;

namespace MongoApp
{
    class Authors
    {
        public ObjectId Id { get; set; }
        public string Name { get; set; }
        public string Surname { get; set; }
        public int Age { get; set; }
        public Company Company { get; set; }
        public List<string> Languages { get; set; }
    }
    class Company
    {
        public string Name { get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Authors autor = new Authors {Name = «Сергій», Surname = «Жадан», Age = 40};
            autor.Company = new Company { Name = «Львівська книгарня» };
            BsonDocument doc = autor.ToBsonDocument();
            Console.WriteLine(doc);
            Console.ReadKey();
        }
    }
}

```

Кожна сутність у базі даних обладнана полем `_id`, що слугує однозначною дескрипторною ознакою для ідентифікації цієї сутності. Застосовуючи атрибут `BsonId`, ми явно визначаємо властивість, яка буде виконувати функцію первинного ключа, забезпечуючи таким чином гарантовану унікальність та цілісність даних:

```

using MongoDB.Bson.Serialization.Attributes;

```



```
namespace MongoApp
{
    class Authors
    {
        [BsonId]
        public int AuthorsId { get; set; }
        public string Name { get; set; }
    }
}
```

Застосовуючи атрибути, ми маємо можливість модифікувати поведінку класів моделей під час процесу мапування на структуру документів MongoDB:

```
using System;
using System.Collections.Generic;
using MongoDB.Bson;
using MongoDB.Bson.Serialization.Attributes;

namespace MongoApp
{
    class Authors
    {
        [BsonId]
        public int AuthorsId { get; set; }
        [BsonElement(«NickName»)]
        public string Name { get; set; }
        public string Surname { get; set; }
        public int Age { get; set; }
        [BsonIgnore]
        public Company Company { get; set; }
        public List<string> Languages { get; set; }
    }
}
```

Атрибут BsonIgnore дозволяє виключити властивість Company з операції серіалізації, таким чином запобігаючи її включенню до результуючого документа. Атрибут BsonElement надає механізм для налаштування параметрів, пов'язаних з представленням певної властивості в документі. Зокрема, в наведеному прикладі атрибут BsonElement використовується для зміни імені поля з Name на NickName. Щоб уникнути додавання до документа елементів, які не мають значення, можна використовувати атрибут BsonIgnoreIfNull. Параметр BsonIgnoreIfDefault в MongoDB використовується для оптимізації процесу серіалізації об'єктів у документи MongoDB. Його основна функція полягає в тому, щоб ігнорувати властивості об'єкта під час серіалізації, якщо їх значення збігається зі значенням за замовчуванням для цього типу даних.

Атрибут BsonRepresentation забезпечує механізм для явного задання способу зберігання властивості в базі даних, забезпечуючи таким чином гнучкість та контроль над процесом серіалізації. Наприклад:

```
using System;
using System.Collections.Generic;
using MongoDB.Bson;
```

```

using MongoDB.Bson.Serialization.Attributes;

namespace MongoApp
{
    class Authors
    {
        [BsonRepresentation(BsonType.ObjectId)]
        public string Id { get; set; }

        public int AuthorsId { get; set; }
        [BsonElement(«NickName»)]
        public string Name { get; set; }
        public string Surname { get; set; }
        [BsonRepresentation(BsonType.String)]
        public int Age { get; set; }
    }
}

```

Для властивості Id явно вказано, що вона буде виконувати роль первинного ключа в базі даних, при цьому тип даних відповідного поля буде ObjectId. Властивість Age, яка має цілочисельний тип, буде зберігатися в базі даних як рядок завдяки застосуванню атрибута BsonRepresentation(BsonType.String), що дозволяє здійснювати додаткову обробку або зберігати додаткову інформацію в цьому полі.

Для здійснення детального налаштування відповідності між класами C# та колекціями MongoDB передбачений клас BsonClassMap. Цей клас дозволяє явно вказати правила мапування, такі як відповідність властивостей класу полям документа, типи даних та інші аспекти. Наприклад, розглянемо клас Authors:

```

using System;
using System.Collections.Generic;
using MongoDB.Bson;
using MongoDB.Bson.Serialization.Attributes;
using MongoDB.Bson.Serialization;

namespace MongoApp
{
    class Authors
    {
        [BsonRepresentation(BsonType.ObjectId)]
        public string Id { get; set; }

        public int AuthorsId { get; set; }
        [BsonElement(«NickName»)]
        public string Name { get; set; }
        public string Surname { get; set; }
        [BsonRepresentation(BsonType.String)]
        public int Age { get; set; }
        [BsonIgnore]
        public Company Company { get; set; }
        public List<string> Languages { get; set; }
    }
    class Company
    {

```

```

        public string Name { get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            BsonClassMap.RegisterClassMap<Authors>(cm =>
            {
                cm.AutoMap();
                cm.MapMember(author => author.Name).SetElementName(«name»);
            });
            Authors a = new Authors { Name = «Сергій», Age = 40 };
            BsonDocument doc = a.ToBsonDocument();
            Console.WriteLine(doc);

            Console.ReadLine();
        }
    }
}

```

За допомогою методу RegisterClassMap() визначається карта зіставлення об'єктів Authors та BsonDocument. Зокрема, у разі для властивість Name буде зіставлятися з полем name.

Механізм конвенцій забезпечує гнуккий і розширюваний підхід до визначення правил мапування, доповнюючи атрибути та BsonClassMap. Наприклад, для стандартизації імен полів в документах можна застосувати конвенцію, що переводить всі ключі в нижній регістр:

```

using System;
using System.Collections.Generic;
using MongoDB.Bson;
using MongoDB.Bson.Serialization.Attributes;
using MongoDB.Bson.Serialization.Conventions;

namespace MongoApp
{
    class Authors
    {
        [BsonRepresentation(BsonType.ObjectId)]
        public string Id { get; set; }

        public int AuthorsId { get; set; }
        [BsonElement(«NickName»)]
        public string Name { get; set; }
        public string Surname { get; set; }
        [BsonRepresentation(BsonType.String)]
        public int Age { get; set; }
        [BsonIgnore]
        public Company Company { get; set; }
        public List<string> Languages { get; set; }
    }
    class Company
    {
        public string Name { get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {

```

```

        var conventionPack = new ConventionPack();
        conventionPack.Add(new CamelCaseElementNameConvention());
        ConventionRegistry.Register(«camelCase», conventionPack, t => true);
        Authors a = new Authors { Name = «Сергій», Age = 40 };

        BsonDocument doc = a.ToBsonDocument();
        Console.WriteLine(doc);

        Console.ReadLine();
    }
}

```

У даному випадку застосовується конвенція `CamelCaseElementNameConvention`, яка автоматично переводить назви властивостей класів у формат `CamelCase` під час мапування на документи MongoDB. Ця конвенція реєструється за допомогою методу `Add()` класу `ConventionPack` та активується викликом методу `ConventionRegistry.Register()`. Останній метод приймає три аргументи: ім'я конвенції, об'єкт `ConventionPack` та умову застосування. Встановлення умови як `true` означає, що конвенція буде застосовуватися до всіх властивостей всіх класів. Таким чином, при виведенні на консоль назв полів документа, вони будуть представлені у форматі `CamelCase`.

Для додавання документів до колекції використовується метод `InsertOneAsync`, що є членом інтерфейсу `IMongoCollection`. Розглянемо приклад додавання одного документа до колекції:

```

using MongoDB.Bson;
using MongoDB.Driver;

namespace MongoDBApp
{
    class Program
    {
        static void Main(string[] args)
        {
            SaveDocs().GetAwaiter().GetResult();

            Console.ReadLine();
        }

        private static async Task SaveDocs()
        {
            string connectionString = «mongodb://localhost»;
            var client = new MongoClient(connectionString);
            var database = client.GetDatabase(«testDB»);
            var collection = database.GetCollection<BsonDocument>(«Authors»);
            BsonDocument a1 = new BsonDocument
            {
                {«Name», «Юрій»},
            }
        }
    }
}

```

```

        {"Surname", "Андрухович"},
        {"Age", 50},
        {"Languages", new BsonArray{"українська", "німецька"}}
    };
    await collection.InsertOneAsync(a1);
}
}
}

```

Результат відпрацювання коду буде наступним:

```

testDB> db.Authors.find()
[
  { _id: ObjectId("6788ede956fe6f10899dc84c"), Name: 'Сергій Жадан' },
  { _id: ObjectId("6788eea356fe6f10899dc84d"), Name: 'Сергій Жадан' },
  {
    _id: ObjectId("6789131556fe6f10899dc84e"),
    Name: 'Іван',
    Surname: 'Івченко',
    Age: 18,
    Education: 'Ліцей Ерудит'
  },
  { _id: 1, Name: 'Петро', Surname: 'Петренко', Age: 29 },
  {
    _id: ObjectId("678a8002e3c957dd45eb864d"),
    Name: 'Людмила',
    Surname: 'Штраус',
    Age: 45,
    Languages: [ 'українська', 'англійська', 'німецька' ]
  },
  {
    _id: ObjectId("67ac8d9d8546fcc115855a28"),
    Name: 'Юрій',
    Surname: 'Андрухович',
    Age: 50,
    Languages: [ 'українська', 'німецька' ]
  }
]

```

Для підвищення ефективності операцій запису, коли необхідно зберегти декілька документів одночасно, метод `InsertManyAsync()` надає можливість передати масив об'єктів, мінімізуючи кількість звернень до бази даних:

```

using MongoDB.Bson;
using MongoDB.Driver;
namespace MongoDBApp
{
    class Program
    {
        static void Main(string[] args)
        {
            SaveDocs().GetAwaiter().GetResult();

            Console.ReadLine();
        }

        private static async Task SaveDocs()
        {
            string connectionString = "mongodb://localhost";
            var client = new MongoClient(connectionString);

```

```

var database = client.GetDatabase("testDB");
var collection = database.GetCollection<BsonDocument>("Authors");
BsonDocument a1 = new BsonDocument
{
    {"Name", "Богдан"},
    {"Surname", "Шевченко"},
    {"Age", 32},
    {"Languages", new BsonArray{"китайська", "німецька"}}
};
BsonDocument a2 = new BsonDocument
{
    {"Name", "Марічка"},
    {"Surname", "Бджілка"},
    {"Age", 50},
    {"Languages", new BsonArray{"українська", "іспанська"}}
};
await collection.InsertManyAsync(new[] { a1, a2 });
}
}
}

```

Однак ми можемо працювати не лише з об'єктами BsonDocument, але й зі стандартними класами C#. Допустимо, нам треба зберегти об'єкти наступних класів:

```

using MongoDB.Bson;
using MongoDB.Driver;

namespace MongoDBApp
{
    class Program
    {
        class Authors
        {
            public ObjectId Id { get; set; }
            public string Name { get; set; }
            public string Surname { get; set; }
            public int Age { get; set; }
            public List<string> Languages { get; set; }
        }

        static void Main(string[] args)
        {
            SaveDocs().GetAwaiter().GetResult();

            Console.ReadLine();
        }

        private static async Task SaveDocs()
        {
            string connectionString = "mongodb://localhost";
            var client = new MongoClient(connectionString);
            var database = client.GetDatabase("testDB");
            var collection = database.GetCollection<Authors>("author");
            Authors a1 = new Authors
            {
                Name = "Леонід",
                Surname = "Сайко",
                Age = 29,
                Languages = new List<string> { "англійська", "німецька"},
            };
        }
    }
}

```

```

        await collection.InsertOneAsync(a1);
        Console.WriteLine(a1.Id);
    }
}

```

З метою забезпечення строгої типізації даних, колекція, призначена для зберігання об'єктів класу Authors, декларується як `MongoCollection<Authors>`. Таким чином, кожен документ в цій колекції відповідає екземпляру класу Authors. При відсутності явно заданого значення поля `_id` під час операції вставки, система автоматично генерує унікальний ідентифікатор для нового документа.

Інтерфейс `IMongoCollection` в MongoDB надає методи `Find()` та `FindAsync` для реалізації операцій пошуку документів. Ці методи приймають об'єкт `BsonDocument` як фільтр, який визначає умови відбору.

Метод `FindAsync()` повертає асинхронний курсор, що забезпечує ефективну обробку великих результатів шляхом поступової завантаження даних.

Метод `Find()` повертає об'єкт, який дозволяє налаштувати додаткові параметри запиту, такі як сортування, проєктування та обмеження кількості результатів. Для отримання результатів у вигляді списку документів використовується метод `ToListAsync()`.

Розглянемо практичний приклад, використаємо метод `FindAsync()` для отримання записів із колекції:

```

using MongoDB.Bson;
using MongoDB.Driver;

namespace MongoDBApp
{
    class Program
    {
        static void Main(string[] args)
        {
            FindDocs().GetAwaiter().GetResult();

            Console.ReadLine();
        }
        private static async Task FindDocs()
        {
            string connectionString = "mongodb://localhost";
            var client = new MongoClient(connectionString);
            var database = client.GetDatabase("testDB");
            var collection1 = database.GetCollection<BsonDocument>("Authors");

```

```

var filter = new BsonDocument();
using (var cur = await collection1.FindAsync(filter))
{
    while (await cur.MoveNextAsync())
    {
        var a1 = cur.Current;
        foreach (var auth in a1)
        {
            Console.WriteLine(auth);
        }
    }
}
}
}
}
}

```

В результаті запуску цієї програми у консоль буде виведено:

```

{ "_id" : { "$oid" : "6788ede956fe6f10899dc84c" }, "Name" : "Серг?й Жадан" }
{ "_id" : { "$oid" : "6788eea356fe6f10899dc84d" }, "Name" : "Серг?й Жадан" }
{ "_id" : { "$oid" : "6789131556fe6f10899dc84e" }, "Name" : "?ван", "Surname" : "Ерудит" }
{ "_id" : 1, "Name" : "Петро", "Surname" : "Петренко", "Age" : 29 }
{ "_id" : { "$oid" : "678a8002e3c957dd45eb864d" }, "Name" : "Людмила", "Surname" : "українська", "англ?йська", "н?мецька" }
{ "_id" : { "$oid" : "67ac8d9d8546fcc115855a28" }, "Name" : "Юр?й", "Surname" : "українська", "н?мецька" }
{ "_id" : { "$oid" : "67ac915cd8240e2d7316c20b" }, "Name" : "Богдан", "Surname" : "китайська", "н?мецька" }
{ "_id" : { "$oid" : "67ac915cd8240e2d7316c20c" }, "Name" : "Мар?чка", "Surname" : "українська", "?спанська" }

```

Для вибірки всіх документів з колекції MongoDB в С# ми можемо задати порожній фільтр у вигляді об'єкта BsonDocument. Це означає, що ми не накладаємо жодних обмежень на вибірку і отримуємо всі документи з колекції:

```
var filter = new BsonDocument();
```

Драйвер MongoDB для С# використовує асинхронний API для забезпечення ефективної роботи з базою даних. Метод FindAsync повертає об'єкт типу Task<IAsyncCursor>, що представляє асинхронну операцію отримання курсору. Для отримання результату цієї операції необхідно використати оператор await:

```
var cursor = await collection.FindAsync(filter);
```

Асинхронний курсор IAsyncCursor дозволяє ітеративно обробляти результати запиту. Метод MoveNextAsync переміщує курсор до наступного набору документів і повертає true, якщо такі документи існують. Для отримання поточного набору документів використовується властивість Current, яка повертає колекцію об'єктів типу IEnumerable<Computer>.

Використання методу Find дозволяє спростити код, оскільки він повертає синхронний результат:

```
using MongoDB.Bson;
using MongoDB.Driver;
using MongoDB.Driver.Linq;

namespace MongoDBApp
{
    class Program
    {
        static void Main(string[] args)
        {
            FindDocs().GetAwaiter().GetResult();

            Console.ReadLine();
        }

        private static async Task FindDocs()
        {
            string connectionString = "mongodb://localhost";
            var cli = new MongoClient(connectionString);
            var database = cli.GetDatabase("test");
            var col = database.GetCollection<BsonDocument>("Authors");
            var filter = new BsonDocument();
            var a1 = await col.Find(filter).ToListAsync();
            foreach (var auth in a1)
            {
                Console.WriteLine(auth);
            }
        }
    }
}
```

Однак, для більшої ефективності та інтеграції з асинхронним кодом, рекомендується використовувати метод FindAsync.

Методи Find та FindAsync приймають об'єкт типу BsonDocument як параметр для визначення критеріїв фільтрації. Цей об'єкт представляє собою документ MongoDB, що містить набір ключ-значення, які визначають умови відбору. Пустий BsonDocument відповідає універсальному набору, тобто повертає всі документи колекції. Однак, для більш точного відбору даних можна використовувати складніші структури запитів:

```
using MongoDB.Bson;
using MongoDB.Driver;

namespace MongoDBApp
{
    class Program
    {
        static void Main(string[] args)
        {
            FindAuthors().GetAwaiter().GetResult();

            Console.ReadLine();
        }
    }
}
```

```

private static async Task FindAuthors()
{
    string connectionString = "mongodb://localhost";
    var client = new MongoClient(connectionString);
    var database = client.GetDatabase("testDB");
    var col = database.GetCollection<BsonDocument>("Authors");
    var fil = new BsonDocument("Name", "Сергій");
    var authors = await col.Find(fil).ToListAsync();
    foreach (var a in authors)
    {
        Console.WriteLine(a);
    }
}
}
}

```

В якості фільтра використовується об'єкт `BsonDocument(«Name», «Сергій»)`, що відповідає запиту на вибір усіх документів, де поле `Name` має значення «Bill». Мова запитів MongoDB надає широкий спектр операторів порівняння (наприклад, `$gt`, `$lt`) для формулювання складніших умов відбору. Для ілюстрації, запит `BsonDocument(«Age», new BsonDocument(«$gt», 25))` дозволяє вибрати всі документи, де значення поля `Age` перевищує 25.

Або, наприклад, поставимо фільтр для вибору документів, у яких поле «Age» має значення від 33 і вище, або поле «Name» має значення «Сергій»:

```

var fil = new BsonDocument("$or", new BsonArray{
    new BsonDocument("Age", new BsonDocument("$gte", 33)),
    new BsonDocument("Name", "Сергій")
});
var authors = await col.Find(fil).ToListAsync();

```

Оператор `$and` дозволяє об'єднати декілька умов в одну, вибираючи лише ті документи, які задовольняють усім вказаним умовам. Тобто, для отримання множини документів, де `Name` має значення «Василь» і `Age` більше 31, використовується наступна конструкція: `BsonDocument(«$and», new BsonArray {BsonDocument(«Name», «Василь»), BsonDocument(«Age», new BsonDocument(«$gt», 31)) })`.

Іншим способом для створення фільтрації є застосування класу `FilterDefinitionBuilder`, методи якого зазначені в таблиці 3.6.

Таблиця 3.6. **Методи класу `FilterDefinitionBuilder`**

Операції порівняння	
Eq	Вибирає ті документи, у яких значення певної властивості дорівнює деякому значенню. Наприклад,

	Builders<Authors>.Filter.Eq(Name», «Сергій»).
Gt	Вибирає лише ті документи, у яких значення певної властивості більше певного значення. Наприклад, Builders< Authors >.Filter.Gt(«Age», 25).
Gte	Вибирає лише ті документи, у яких значення певної властивості більше або дорівнює деякому значенню.
Lt	Вибирає лише ті документи, у яких значення певної властивості менше певного значення.
Lte	Вибирає тільки ті документи, у яких значення певної властивості менше або дорівнює деякому значенню.
Ne	Вибирає ті документи, у яких значення певної властивості не дорівнює деякому значенню Builders< Authors >.Filter.Ne(«Age», 23).
In	Отримує всі документи, у яких значення властивості може приймати одне із зазначених значень. Наприклад, знайдемо всі документи, у яких властивість Age одно або 23, або 25, або 27: Builders< Authors >.Filter.In(«Age»,new List<BsonInt32>() { 23, 25, 27}).
Nin	Протилежність оператору In – вибирає всі документи, у яких значення властивості не набуває ні одного із зазначених значень.
Логічні операції	
Not	Повертає документи, які не підпадають під певну умову Builders<Person>.Filter.Not(Builders< Authors >.Filter.Eq(«Name», «Сергій»)).
Or	Визначає набір окремих фільтрів, які використовуються для вибірки.
And	Так само, як і Or, визначає набір окремих фільтрів, які

	використовуються для вибірки, але тепер повертаються тільки ті документи, які відповідають усім цим об'єктам FilterDefinition.
Операції з елементами	
Exists	Вибирає з бази даних документи, у яких присутній певний ключ, наприклад, Builders< Authors >.Filter.Exists(«Name»);
NotExists	Вибирає з бази даних документи, у яких відсутній певний ключ, наприклад, Builders< Authors >.Filter.NotExists(«Age»);
Обчислення значень	
Regex	Вибирає всі документи, у яких значення ключа відповідає регулярним значенням. Наприклад, виберемо всі документи, в іменах яких є буква «о»: Builders< Authors >.Filter.Regex(«Name», new BsonRegularExpression(«o»)). Об'єкт BsonRegularExpression дозволяє встановити регулярне значення.
Where	Повертає документи, які відповідають певним умовам. Умова визначається за допомогою лямбда-виразу: Builders< Authors >.Filter.Where(e => e.Name == «Ольга»);
Операції з масивами	
All	Вибирає всі документи, які містять всі елементи масиву.
Size	Вибирає всі документи, які містять певну кількість елементів.

Повторимо минулий приклад, але вже через клас FilterDefinitionBuilder:

```
using MongoDB.Bson;
using MongoDB.Driver;

namespace MongoDBApp
{
    class Program
    {
        static void Main(string[] args)
        {
            FindPeople().GetAwaiter().GetResult();

            Console.ReadLine();
        }
        private static async Task FindPeople()
        {
            MongoClient client = new MongoClient(«mongodb://localhost:27017»);
```

```

var database = client.GetDatabase(«testDB»);
var collection = database.GetCollection<BsonDocument>(«Authors»);
// Визначаємо будівельника фільтрації
var builder = Builders<BsonDocument>.Filter;
// Здається фільтр
var fil = builder.Eq(«Name», «Сергій»);
var authors = await collection.Find(fil).ToListAsync();
foreach (var a in authors)
{
    Console.WriteLine(a);
}
}
}
}

```

За допомогою стандартних операцій програмування кон'юнкції, диз'юнкції та логічного заперечення, ми можемо поєднувати запити. Наприклад, фільтр всіх документів у яких Name="Сергій" чи Name="Антон":

```

var builder = Builders<BsonDocument>.Filter;
var filter = builder.Eq(«Name», «Сергій») | builder.Eq(«Name», «Антон»);
var authors = await collection.Find(filter).ToListAsync();

```

Запит на отримання об'єктів, у яких Name = "Сергій", але при цьому Age не дорівнює 30 може мати такий код:

```

var builder = Builders<BsonDocument>.Filter;
var filter = builder.Eq(«Name», «Сергій») & !builder.Eq(«Age», 30);
var authors = await collection.Find(filter).ToListAsync();

```

Застосування стандартних класів C#, замість BsonDocument надає додаткові можливості для формулювання запитів. Зокрема, використання лямбда-виразів дозволяє визначати критерії відбору більш інтуїтивно та компактно. Так, попередній приклад можна переписати у вигляді лямбда-виразу, що суттєво спрощує його читання та написання:

```

var collection = database.GetCollection<Person>(«Authors»);
var authors = await collection.Find(x => x.Name == «Сергій» && x.Age != 30).ToListAsync();
foreach (var a in authors)
{
    Console.WriteLine(«{0} - {1}», a.Name, a.Age);
}

```

Код, наведений нижче демонструє роботу методів And і Or:

```

// метод Or
var filter1 = Builders<Authors>.Filter.Eq(«Name», «Сергій»);
var filter2 = Builders<Authors>.Filter.Eq(«Age», 20);
var filterOr = Builders<Authors>.Filter.Or(new List<FilterDefinition<Authors>> { filter1, filter2 });
// метод And
var filter3 = Builders<Authors>.Filter.Eq(«Name», «Антон»);
var filter4 = Builders<Authors>.Filter.Eq(«Age», 27);
var filterAnd = Builders<Authors>.Filter.And(new List<FilterDefinition<Authors>> { filter3, filter4 });

```

Розглянемо роботу з масивами. Наприклад, у нас є клас Authors, який містить список мов:

```
class Authors
{
    //.....
    public List<string> Languages { get; set; }
}
```

Тепер знайдемо авторів, у яких є деякі фіксовані дві мови, а також знайдемо авторів, у яких в цьому масиві є дві будь-які мови:

```
// отримаємо всі об'єкти, у яких визначено англійську та французьку мови
var filter = Builders<Person>.Filter.All(«Languages», new List<string>() {
«англійська», «французька» });
// отримаємо всі об'єкти, у яких в масиві languages лише 2 елементи
var filter = Builders<Person>.Filter.Size(«Languages», 2);
```

Як видно із прикладів, драйвер MongoDB для C# забезпечує всі необхідні інструменти для ефективної роботи з цією базою даних.

Питання для самоперевірки

1. Яке призначення має бібліотека MongoDB.BSON.dll?
2. Яку функціональність надає бібліотека MongoDB.Driver.dll?
3. Чи обов'язково закривати підключення з базою даних MongoDB?
4. Що робить метод Deserialize() і якому класу він належить?
5. Яка різниця між методами Find та FindAsync?
6. Що таке конвенція у програмуванні?

Завдання

Написати програмний застосунок, який відповідає наступним вимогам:

Варіант	Опис завдання
1	Структура: назва торгової організації, адреса, ПІБ директора, податковий номер, постачальники (може бути декілька) 1. Закодувати жорстке підключення до бази даних у застосунку. 2. Застосунок виводить всі колекції бази даних. 3. Застосунок має бути розробленим за допомогою класу BsonDocument.

	4.Застосунок повинен додавати документи в колекцію. 5. Застосунок повинен уміти вибирати документи, з назвами всіх торгових організацій, назви яких починаються з літери «О» і які мають двох постачальників.
2	Структура: ПІБ, посада (може бути декілька), номер відділу (може бути декілька), в якому працює, стать, адреса, дата народження. 1.Рядок підключення до бази даних має бути в конфігураційному файлі. 2. Вивести всі бази даних, що є на сервері. 3. Застосунок має бути розробленим за допомогою стандартних класів с#. 4.Застосунок повинен додавати документи в колекцію. 5. Застосунок повинен уміти вибирати документи співробітників, які працюють у відділі №2 і які мають дві посади.
3	Структура: ПІБ, паспортні дані, стать, вік, діти (може бути декілька) 1.Закодувати жорстке підключення до бази даних у застосунку. 2.Застосунок має бути розробленим за допомогою стандартних класів с#. 3. Перевести об'єкти стандартних класів у об'єкти класу BsonDocument із застосуванням атрибутів BsonId,BsonElement, BsonIgnore. 4.Застосунок повинен додавати документи в колекцію. 5. Застосунок повинен уміти вибирати документи громадян, які двох дітей і молодші 30 років.
4	Структура: назва туру, країна (може бути декілька), міста (може бути декілька), тип пересування (може бути декілька), тип харчування, ціна туру, тип проживання 1. Рядок підключення до бази даних має бути в конфігураційному файлі.

	2.Застосунок має бути розробленим за допомогою стандартних класів с#. 3. Перевести об'єкти стандартних класів у об'єкти класу BsonDocument із застосуванням атрибуту BsonRepresentation. 4.Застосунок повинен додавати документи в колекцію. 5. Застосунок повинен уміти вибирати документи з назвами турів в Німеччину і ціною нижче 20 тис.грн.
5	Структура: прізвище, ім'я, по батькові, адреса, місто, телефон (може бути декілька). 1. Закодувати жорстке підключення до бази даних у застосунку. 2.Застосунок має бути розробленим за допомогою стандартних класів с#. 3. Перевести об'єкти стандартних класів у об'єкти класу BsonDocument із застосуванням механізму конвенцій. 4.Застосунок повинен додавати документи в колекцію. 5. Застосунок повинен уміти вибирати документи з прізвищами громадян, які живуть у місті Києві і мають оператор мобільного зв'язку Vodafone.

Рекомендована література до розділу [23-29].

4. БАЗИ ДАНИХ «КЛЮЧ-ЗНАЧЕННЯ»

4.1 Початкова інформація про СУБД Redis

Redis (Remote Dictionary Server) – це потужна та відкрита база даних, що оперує в оперативній пам'яті, яка застосовується для хешування, управління сесіями та інших завдань. Вона є високопродуктивною системою управління базами даних з відкритим кодом, що базується на моделі «ключ-значення». Завдяки своїй ін-меморійній архітектурі та підтримці широкого спектру типів даних, Redis знайшла застосування в розробці програмного забезпечення.

Типові сценарії використання Redis включають: реалізацію механізмів обміну повідомленнями типу «видавець-підписник», обробку поточкових даних у реальному часі, керування чергами завдань та створення хешів для підвищення швидкодії додатків.

Як ін-меморійний хеш, Redis помітно зменшує навантаження на основну базу даних (наприклад, MySQL або T-SQL), забезпечуючи швидкий доступ до часто використовуваних даних, таких як сесійні дані, хешовані результати запитів та динамічно згенерований вміст.

Незважаючи на первинну роль хешу, Redis пропонує достатній рівень стійкості даних, що дозволяє використовувати його як основну базу даних у деяких випадках. Гнучка система плагінів та конфігурацій високої доступності роблять Redis універсальним інструментом для широкого кола завдань.

Однією з ключових переваг Redis є здатність значно зменшувати латентність операцій з даними завдяки використанню архітектури в оперативній пам'яті. На відміну від традиційних систем управління базами даних, що оперують з даними, розміщеними на повільніших носіях, таких як жорсткі диски чи твердотільні накопичувачі, Redis забезпечує на порядки швидший доступ до даних, що зберігаються в оперативній пам'яті.

Найпростіша конфігурація Redis передбачає використання одного екземпляру (рис. 4.1), що забезпечує базову функціональність сховища даних. Такий підхід є доцільним для невеликих проєктів, оскільки дозволяє швидко

розгорнути систему та отримати перші результати. Однак, одноточкова архітектура робить систему вразливою до відмов: збій єдиного екземпляру призводить до повного припинення доступу до даних та, як наслідок, до деградації продуктивності системи в цілому.

Єдиний екземпляр

Redis



Рисунок 4.1 – Єдиний екземпляр

Для забезпечення надійності та довготривалості зберігання даних у Redis використовуються механізми персистенції: RDB (Redis Database) та AOF (Append-Only File). RDB дозволяє створювати періодичні знімки стану бази даних у вигляді компактних файлів, тоді як AOF зберігає послідовність команд, що змінюють стан бази. Ці механізми забезпечують відновлення даних у разі збоїв та дозволяють реалізувати різноманітні стратегії реплікації та відновлення після аварій.

Однією з поширених архітектур розгортання Redis є конфігурація «майстер-слейв» (Redis HA, High Availability) (рис. 4.2), що передбачає реплікацію даних з основного вузла (майстра) на один або декілька вторинних вузлів (слейвів). Реплікація здійснюється асинхронно шляхом передачі команд з майстра на слейви. Така конфігурація дозволяє масштабувати операції читання та підвищити доступність системи за рахунок дублювання даних.

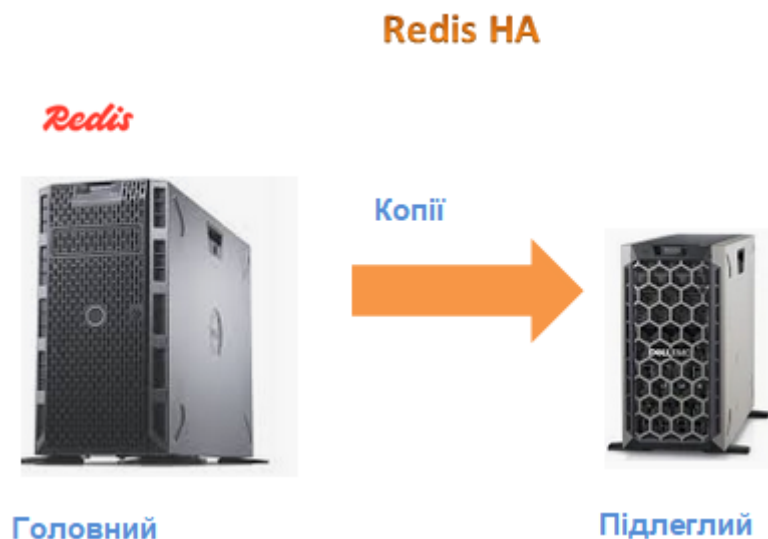


Рисунок 4.2 – Конфігурація Redis HA

Для забезпечення високої доступності (HA) системи, що використовують Redis, важливо мінімізувати кількість точок відмови. Реплікація даних на декілька слейвів є одним із способів досягнення цієї мети. Крім того, для підвищення надійності системи необхідне використання високошвидкісних та надійних каналів зв'язку між майстром та слейвами, а також впровадження механізмів автоматичного виявлення та усунення збоїв.

Redis Sentinel – це розподілена система моніторингу та управління, призначена для забезпечення високої доступності кластерів Redis (рис. 4.3). Sentinel формує кластер незалежних процесів, які спільно моніторять стан основних вузлів Redis (майстрів) та їхніх реплік (слейвів).



Рисунок 4.3 – Конфігурація Sentinel Redis

Ключові функції Sentinel:

- моніторинг: безперервне відстеження стану вузлів Redis, включаючи доступність, затримку, конфігурацію та об'єм даних;
- виявлення збоїв: швидке визначення несправностей вузлів та ініціація процедур відновлення;
- автоматичне перемикання: при виявленні збою майстра, Sentinel автоматично вибирає нового майстра з числа доступних слейвів на основі алгоритму кворуму;
- управління конфігурацією: динамічне оновлення конфігурації кластера Redis, включаючи додавання/видалення вузлів та зміну параметрів реплікації;
- нотифікації: надсилання сповіщень адміністраторам про зміну стану системи, що дозволяє оперативно реагувати на проблеми.

Для прийняття рішень про перемикання майстра та інші критичні операції Sentinel використовує механізм кворуму. Кворум – це мінімальна кількість вузлів Sentinel, які повинні підтвердити необхідність виконання операції. Це дозволяє уникнути помилкових тригерів та забезпечує узгодженість стану кластера.

Переваги використання Sentinel:

- висока доступність: забезпечує безперервну роботу кластера Redis навіть при виході з ладу окремих вузлів;
- автоматизація: знімає з адміністратора рутинні завдання з моніторингу та управління кластером;
- гнучкість: підтримує різні конфігурації кластерів та сценарії використання;
- надійність: завдяки механізму кворуму забезпечує стійкість до помилок.

Redis Sentinel є незамінним інструментом для забезпечення високої доступності та надійності кластерів Redis. Завдяки своїм функціональним

можливостям, Sentinel дозволяє автоматизувати рутинні операції та підвищити загальну стабільність системи.

Система керування базами даних Redis не має офіційної підтримки для операційної системи Windows. Проте, для потреб розробки, існує можливість інсталяції Redis на Windows за допомогою певної конфігурації.

Необхідною умовою для успішної інсталяції є активація підсистеми Windows для Linux (WSL). Ця технологія дозволяє виконувати бінарні файли Linux безпосередньо в середовищі Windows. Для використання даного методу операційна система повинна відповідати вимогам: Windows 10 версії 2004 або новішої, або Windows 11. Для того щоб встановити підсистему WSL, потрібно перейти в командний рядок і виконати команду:

```
C:\Users\1>wsl --install
```

Після встановлення WSL потрібно інсталювати операційну систему, наприклад Ubuntu:

```
wsl --install -d Ubuntu
```

Потім потрібно оновити інформацію про доступні пакети в операційній системі Ubuntu:

```
sudo apt update -y
```

і оновити встановлені пакети до останніх версій:

```
sudo apt upgrade -y
```

Після цього переходимо до встановлення СУБД Redis. Для цього потрібно виконати команду:

```
sudo apt install redis-server
```

Після успішної інсталяції системи необхідно внести одну важливу зміну до конфігураційного файлу `redis.conf`, що був автоматично згенерований під час встановлення. Для цього потрібно відкрити файл в текстовому редакторі (рис. 4.4). Всередині файлу потрібно знайти директиву `supervised`. За замовчуванням для директиви `supervised` встановлено значення `no`. Оскільки ми

застосовуємо операційну систему Ubuntu, яка використовує систему ініціалізації systemd, змініть значення для директиви supervised на systemd.

```
# Redis configuration file example.
#
# Note that in order to read the configuration file, Redis must be
# started with the file path as first argument:
#
# ./redis-server /path/to/redis.conf
#
# Note on units: when memory size is needed, it is possible to specify
# it in the usual form of 1k 5GB 4M and so forth:
#
# 1k => 1000 bytes
# 1kb => 1024 bytes
# 1m => 1000000 bytes
# 1mb => 1024*1024 bytes
# 1g => 1000000000 bytes
# 1gb => 1024*1024*1024 bytes
#
```

Рисунок 2.1 – Конфігураційний файл Redis.

Потім перезапустіть службу Redis, щоб відобразити зміни, які були внесені:

```
sudo systemctl restart redis.service
```

Далі потрібно перевірити правильність встановлення серверу Redis. Для цього потрібно запустити клієнт серверу redis-cli і набрати команду:

```
127.0.0.1:6379> ping
PONG
127.0.0.1:6379>
```

відповіддю клієнта на яку буде Pong. Сервер встановлено правильно.

Питання для самоперевірки

1. Які типові сценарії використання СУБД Redis?
2. Яка основна відмінність СУБД Redis від традиційних СУБД?
3. Яка найпростіша конфігурація Redis?
4. Які механізми використовуються для забезпечення надійності та довготривалості даних у Redis?
5. Що таке RDB і для чого призначене?
6. Що таке AOF і для чого призначене?

7. Яка архітектура конфігурації «майстер-слейв»?
8. Що досягається реплікацією даних на декілька слейвів?
9. Яку архітектуру має конфігурація Redis Sentinel?
10. Назвіть основні переваги цієї конфігурації та ключові функції.
11. Що таке конфігураційний файл Redis, його призначення, яка інформація в ньому зберігається?

Завдання

Встановіть СУБД Redis. Перевірте правильність встановлення.

4.2 Управління базами даних в Redis

Екземпляр Redis за замовчуванням підтримує 16 логічно розділених баз даних. Дані у кожній з цих баз даних є ізольованими від даних інших баз даних у межах того ж екземпляра. Таким чином, виконання операцій над даними в одній базі даних не спричиняє жодних змін у даних, що зберігаються в інших базах даних цього ж екземпляра.

Логічні бази даних Redis індексуються числовими значеннями від 0 до 15. При встановленні з'єднання з екземпляром Redis, за замовчуванням встановлюється з'єднання з базою даних під індексом 0. Переключення між базами даних здійснюється за допомогою команди SELECT після встановлення з'єднання.

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

У випадку вибору бази даних, індекс якої відмінний від 0, відповідна інформація відображається у командному інтерфейсі redis-cli.

```
127.0.0.1:6379[1]>
```



Для здійснення обміну всіма даними між двома базами даних передбачено команду `swapdb`. Наприклад, виконання команди `swapdb` з

аргументами 0 та 1 призведе до обміну вмістом бази даних з індексом 0 з вмістом бази даних з індексом 1. При цьому, будь-які клієнтські з'єднання, встановлені з однією з цих баз даних, негайно відобразять внесені зміни. У разі успішного виконання операції, команда `swapdb` повертає відповідь OK.

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

Питання для самоперевірки

1. Скільки за замовчуванням баз даних підтримує екземпляр Redis?
2. Як переключатися з однієї бази даних на іншу?
3. Як індексуються бази даних в Redis?
4. Як скопіювати дані з однієї бази даних в іншу?

4.3 Ключі та значення

Ключ у контексті Redis слугує унікальним ідентифікатором для зберігання та доступу до даних. Він являє собою рядок символів, що асоціюється з певною інформацією. Структура ключа може бути довільною, проте часто використовується ієрархічний принцип, як у прикладі `product:<id>`, де `product` вказує на те, які це дані, а `<id>` – на конкретний екземпляр. Розділювачі, такі як двокрапка, підвищують читабельність ключа, хоча їх синтаксичне значення може варіюватися залежно від конкретної реалізації.

Значення – це дані, пов'язані з відповідним ключем. Redis підтримує широкий спектр типів даних, включаючи рядки, числа, а також складні структури, такі як списки, множини, хеші. На низькому рівні всі значення зберігаються як послідовності байтів, що забезпечує гнучкість у представленні різних типів даних.

Системою управління базами даних Redis передбачено низку операцій для маніпулювання ключами, незалежно від типу асоційованих з ними даних. Розглянемо деякі з найпоширеніших.

Операція `rename` призначена для зміни імені ключа. У разі успішного виконання повертається статус `OK`:

```
127.0.0.1:6379[1]> rename product:1 product:4
OK
```

Для випадкового вибору ключа з поточної бази даних використовується команда `randomkey`:

```
127.0.0.1:6379[1]> randomkey
"product:2"
```

Результатом виконання цієї команди буде повернення випадково обраного ключа.

Команда `type` повертає тип даних, асоційований з вказаним ключем. У разі успішного виконання команди для існуючого ключа результатом буде рядок, що вказує на тип даних (наприклад, `string`, `list`, `hash`, тощо). Якщо ж зазначений ключ відсутній у базі даних, команда поверне значення `none`.

```
127.0.0.1:6379[1]> type product:2
string
```

Для зміни бази даних, в якій зберігається ключ, використовується команда `move`. Синтаксис команди наступний:

`move` ключ номер_бази_даних,

де:

- ключ: ім'я ключа, який необхідно перемістити.
- номер_бази_даних: номер цільової бази даних.

```
127.0.0.1:6379[1]> move product:2 0
(integer) 1
```

У разі успішного виконання команди ключ буде переміщено до вказаної бази даних, а команда поверне статус `OK`.

Одним з фундаментальних аспектів управління системою хешування Redis є видалення даних, що більше не потрібні. Для цієї мети система надає дві основні команди: `del` та `unlink`.

Команда `del` забезпечує синхронне видалення одного або кількох ключів. Синтаксис команди простий:

```
127.0.0.1:6379[1]> del product:3
(integer) 1
```

Після виконання команди `del` відбувається блокування клієнта до моменту повного звільнення ресурсів, зайнятих видаленими ключами. Хоча для невеликих об'єктів цей процес займає незначний час, при роботі з великими даними, особливо хешами, синхронне видалення може призвести до значних затримок у роботі системи.

Команда `unlink`, на відміну від `del`, пропонує більш ефективний підхід до видалення даних. Вона ініціює асинхронний процес звільнення ресурсів, що дозволяє продовжити обробку інших запитів клієнта без затримок. Алгоритм роботи `unlink` наступний:

1. оцінка вартості операції: перед початком видалення команда оцінює обсяг ресурсів, зайнятих ключем;
2. вибір стратегії: якщо вартість видалення невелика, `unlink` працює аналогічно `del`, виконуючи видалення синхронно. В іншому випадку запускається окремий потік, який поступово звільняє зайняту пам'ять у фоновому режимі.

Загальна рекомендація полягає у використанні команди `unlink` для видалення ключів, особливо якщо вони пов'язані з великими об'єктами даних. Це дозволяє зберегти високу продуктивність системи та уникнути блокування клієнтів.

Для повного очищення бази даних або окремої бази даних передбачені команди `flushdb` та `flushall` відповідно. Важливо зазначити, що виконання цих команд призводить до безповоротної втрати всіх даних. Тому використовувати їх слід з особливою обережністю.

Для асинхронного виконання масового видалення передбачена опція `async`. Наприклад, для асинхронного очищення всіх баз даних використовується команда:

```
127.0.0.1:6379[1]> flushall async
```

Розуміння особливостей команд `del`, `unlink`, `flushdb` та `flushall` є необхідним для ефективного управління системою Redis. Правильний вибір

команди залежить від конкретних вимог до продуктивності та надійності застосування.

Ефективне використання системи керування базами даних Redis значною мірою залежить від раціонального проєктування ключів. Добре продумана структура ключів забезпечує не лише зручність доступу до даних, а й оптимізує роботу системи в цілому.

Ключі мають бути інтуїтивно зрозумілими та відображати свою функціональну роль у системі. Використання дефісів або підкреслень для розділення сегментів ключа підвищує його читабельність. Введення префіксів дозволяє класифікувати ключі за типами даних або функціональними областями (наприклад, `product:`, `cache:`, `session:`). Кожен ключ повинен бути унікальним у межах бази даних, щоб уникнути невизначеності та помилок при отриманні даних. Довжина ключа повинна бути достатньою для забезпечення його унікальності, але не надто великою, щоб уникнути негативного впливу на продуктивність системи. Потрібно уникати включення в ключ надлишкової інформації, яка може бути отримана з інших джерел. Тип даних, що зберігається за ключем, повинен відповідати характеру даних та операціям, які будуть над ними виконуватися. Вибір оптимального типу даних дозволяє ефективно використовувати ресурси системи та прискорити виконання запитів. Використання різних просторів імен для різних категорій даних сприяє організації та пошуку інформації. Розділення просторів імен мінімізує ризик випадкового перезапису даних. Для даних, що мають обмежений термін актуальності, встановлюється час життя (TTL), що автоматично видаляє застарілу інформацію.

Дотримання вищезазначених принципів дозволяє створити ефективну та масштабовану систему керування ключами в Redis, що є основою для побудови високопродуктивних додатків.

Питання для самоперевірки

1. Що таке ключ у контексті СУБД Redis? Яка структура ключа?

2. Що таке ключ у контексті СУБД Redis?
3. Яким чином можна перейменувати ключ?
4. Яка команда використовується для випадкового вибору ключа?
5. Яким чином дізнатися якого типу даних значення зберігається в ключі?
6. Як перенести ключі із однієї бази даних в іншу?
7. Які команди використовуються для видалення ключів, у чому їх різниця?
8. Які команди використовуються для повного очищення баз даних?
9. Яким чином організувати асинхронне очищення баз даних?
10. Як іменувати ключі?
11. Якої довжини повинен бути ключ?
12. Що таке TTL?

4.4 Типи даних в Redis

В СУБД Redis імплементовані типи даних подані в таблиця 4.1

Таблиця 4.1. Типи даних, які підтримує Redis

Тип даних	Опис
String	Базовий тип даних Redis. Рядки в Redis бінарно-безпечні, можуть використовуватися як числа, обмежені розміром 512 Мб. Бінарно-безпечні рядки в контексті Redis означають, що у базі даних можна зберігати абсолютно будь-які дані у форматі рядка, не турбуючись про спеціальні символи, нульові байти чи інші обмеження. Redis використовує рядки як основний тип даних для зберігання значень, що асоціюються з ключами. Рядок в Redis є впорядкованою послідовністю байтів, що дозволяє зберігати як текстові, так і бінарні дані. Над рядками можна виконувати широкий спектр операцій, включаючи інкрементування, декрементування, побітові операції та конкатенацію. Ці можливості роблять рядки потужним інструментом для реалізації різних структур даних та алгоритмів, таких як хеші, лічильники та бітові масиви.
Lists	Списки Redis — це списки рядків, відсортованих за порядком вставки. Максимальна кількість елементів списків— $2^{32} - 1$.
Sets	На відміну від списків, множини не зберігають порядок елементів і не дозволяють мати дублікати. Основні операції над множинами — це додавання, видалення елементів та перевірка на наявність. Максимальна кількість елементів списків— $2^{32} - 1$.

Hashes	Класичні хеш-таблиці. Хеші надають гнучкий спосіб зберігання структурованої інформації в Redis. Кожен хеш може містити довільну кількість пар «поле-значення», що робить їх ідеальними для представлення різних типів об'єктів.
Sorted sets	Впорядковані множини в Redis представляють собою структури даних, що містять набір унікальних рядків, які впорядковуються за числовим значенням (оцінкою), пов'язаним з кожним рядком.
Streams	Це спеціальна структура даних, яка функціонує як журнал, до якого можна лише додавати нові записи.
Bitfields	Це особливий тип даних у Redis, який дозволяє зберігати масиви бітів (нулів та одиниць) у межах одного рядкового значення. Кожен біт у цьому масиві може представляти окреме логічне значення або флаг. Ця структура даних особливо корисна для зберігання невеликих цілих чисел, бітових масок або будь-якої іншої інформації, яку можна представити у вигляді бітових послідовностей.
Bitmaps	Бітмапи в Redis не є окремим типом даних, а скоріше набором бітових операцій, які виконуються над рядковими даними.
Geospatial indexes	Структура, яка зберігає географічні координати (широту та довготу).

Redis – це база даних ключ-значення, де значення зберігаються як прості рядки. Вона спроектована для максимальної гнучкості та швидкості. Статична типізація передбачає жорсткий контроль над типами даних на етапі компіляції, що суперечить динамічному характеру Redis.

Одна з ключових переваг Redis – це швидкість цієї СУБД. Статична типізація може додати накладні витрати на перевірку типів, що сповільнить роботу. У Redis можна зберігати будь-які дані у вигляді рядків: від простих чисел до складних об'єктів, серіалізованих у будь-який формат. Тут працює динамічна типізація, тобто тип даних визначається під час виконання запиту.

Питання для самоперевірки

1. Що таке бінарно-безпечні рядки?
2. Який основний тип даних для зберігання інформації в Redis?
3. Що таке списки в Redis?
4. Яка максимальна кількість елементів в списку Redis?
5. Чим відрізняється множина від списку в Redis?
6. Що таке хеші в Redis?
7. Чим відрізняється множина від упорядкованої множини в Redis?
8. Як функціонує структура Streams в Redis?

4.4.1 Тип даних «string». Структура даних «string» у Redis призначена для зберігання єдиного значення. Це означає, що кожному ключу відповідає саме один рядок даних. Однак, формат даних, що зберігається у рядку, не обмежений текстовими символами. Redis підтримує зберігання бінарних даних, включаючи зображення, звукові файли тощо, за умови, що загальний розмір не перевищує 512 мегабайт. Для запису нового рядкового значення використовується команда `SET`. Наприклад, команда:

```
127.0.0.1:6379[1]> set product:1 "notebook HP"
OK
```

створює пару «ключ-значення», де ключ `product:1` пов'язаний зі значенням «notebook HP».

В СУБД Redis є можливість відразу додати декілька рядків у базу даних за допомогою команди `mset`:

```
127.0.0.1:6379[1]> mset product:2 "notebook HP i7" product:3 "notebook Lenovo"
OK
```

Для створення нового рядкового значення або для зміни контексту рядку в Redis також можна використовувати команду `append`:

```
127.0.0.1:6379[1]> append product:1 "Noteook HP"
(integer) 10
```

Якщо рядок вдало додався, то у клієнті виводиться кількість символів у доданому рядку:

```
(integer) 10
```

Щоб отримати значення рядка по ключу, використовується команда `get`:

```
127.0.0.1:6379[1]> get product:1
"Noteook HP"
```

Для того, щоб отримати рядкові значення по декількох рядках зазвичай використовують команду `mget`:

```
127.0.0.1:6379[1]> mget product:1 product:2 product:3
1) "Noteook HP"
2) (nil)
3) (nil)
```

Якщо рядок містить ціле число, ви можете використовувати команду `incr` для його збільшення на одиницю:

```
127.0.0.1:6379[1]> set value:1 "100"
OK
127.0.0.1:6379[1]> get value:1
"100"
127.0.0.1:6379[1]> INCR value:1
(integer) 101
```

Аналогічно, командою `incrby` можна збільшити значення числового рядка на певну величину:

```
127.0.0.1:6379[1]> INCRBY value:1 19
(integer) 120
```

Команди `decr` та `decrby` працюють аналогічно, але вони зменшують ціле число, що зберігається в числовому рядку.

Для видалення ключа з цим типом використовується команда `del`.

Питання для самоперевірки

Скільки значень типу “string” зберігається в одному ключі?

Яка команда використовується для створення нового рядка?

Яким чином можливо додати кілька рядків у базу даних?

Як прочитати один або декілька рядків з бази даних?

Як видалити ключ, який зберігає рядок?

Завдання

Варіант	Опис завдання
1	Для мікросервісів «Розумного поля» потрібне спільне збереження даних в хеші. В текстовому файлі написати сценарій, який реалізує наступний функціонал: База даних знаходиться під індексом 1. У базі зберігаються назви полів (наприклад, «огіркове поле», «полуничне поле» тощо). Потрібно продумати назви ключів. Додати п'ять значень в базу даних. Переглянути всі записи. Останній видалити. Вивести випадковий ключ із поточної бази даних.
2	Для інформаційної системи потрібно збереження даних про користувачів в хеші. У текстовому файлі написати сценарій, який реалізує наступний функціонал: 1. У базі зберігаються ніки користувачів. Потрібно продумати назви ключів. 2. Додати п'ять значень в базу даних. 3. Переглянути всі записи. 4. Останній видалити. 5. Скопіювати поточну базу даних в базу даних під індексом 1. 6. Поточну базу очистити
3	Для мікросервісів «Розумна освіта» потрібне спільне збереження даних в хеші. В текстовому файлі написати сценарій,

	який реалізує наступний функціонал: 1.У базі зберігаються назви дисциплін. Потрібно продумати назви ключів. 2.Додати п'ять значень в базу даних. 3.Переглянути всі записи. 4.Останній видалити. 4.Вивести випадковий ключ із поточної бази даних. 5.Скопіювати інформацію з поточної бази в базу даних під індексом 2. 6.Очистити поточну базу даних.
4	Для мікросервісів «Розумного місто» потрібне спільне збереження даних в хеші. В текстовому файлі написати сценарій, який реалізує наступний функціонал: 1.База даних знаходиться під індексом 1. 2.У базі зберігаються назви ресторанів. Потрібно продумати назви ключів. 3. Додати п'ять значень в базу даних. 4. Переглянути всі записи. 5.Останній видалити. 6.Вивести випадковий ключ із поточної бази даних.
5	Для пошукової системи потрібно зберігати останні слова із запиту користувача в хеші. В текстовому файлі написати сценарій, який реалізує наступний функціонал: 1.У базі зберігаються слова із запиту користувача. Потрібно продумати назви ключів. 2.Додати п'ять значень в базу даних. 3.Переглянути всі записи. 4.Останній видалити. 5. Скопіювати поточну базу даних в базу даних під індексом 1.

4.4.2 Тип даних «List». Структури даних типу «список» забезпечують ефективне зберігання та обробку впорядкованих колекцій елементів, індексованих за допомогою цілих чисел. До списку можна динамічно додавати нові елементи, отримувати доступ до першого та останнього елементів, а також здійснювати модифікацію елементів за заданим індексом. Порядок слідування елементів у списку строго фіксується, що дозволяє виконувати операції доступу та модифікації за лінійним часом.

Для ілюстрації, розглянемо створення списку `visitors`, призначеного для зберігання інформації про користувачів, які нещодавно зареєструвалися на нашому торговельному майданчику:

```
127.0.0.1:6379> lpush visitors "Ivan"
```

Ключ у системі Redis асоціюється з єдиним списком, який може містити до 4,294,967,295 елементів. Операції додавання елементів до списку здійснюються за допомогою команд `lpush` (додавання в початок) та `rpush` (додавання в кінець). Кількість елементів у списку повертається як результат виконання цих команд.

СУБД Redis дозволяє додати кілька елементів списку за допомогою одного `lpush` або `rpush`.

Команди `lpushx` та `rpushx` призначені для додавання елементів до існуючого списку. У разі відсутності списку операція не виконується, а повертається нульове значення:

```
127.0.0.1:6379> lpushx visitors "Pavlo" "Iren" "Mariia"  
(integer) 4
```

Команда `lset` забезпечує механізм оновлення елементів списку. Для її використання необхідно вказати ключ, індекс елемента, що підлягає зміні, та нове значення, яким слід замінити поточне.

```
127.0.0.1:6379> lset visitors 3 "Kseniia"  
OK
```

Команда `lrange` дозволяє отримати діапазон елементів зі списку, задаючи початковий та кінцевий індекси. Індксація елементів у списку здійснюється, починаючи з нуля, тобто перший елемент має індекс 0, другий – 1 і так далі.

```
127.0.0.1:6379> lrange visitors 0 3
1) "Kseniia"
2) "Iren"
3) "Pavlo"
4) "Ivan"
```

Синтаксис команди `lindex` передбачає обов'язкову наявність аргументу – індексу. Цей індекс визначає позицію елемента в списку, який необхідно отримати. Таким чином, команда дозволяє здійснювати доступ до окремих елементів лінійної структури даних:

```
127.0.0.1:6379> lindex visitors 1
"Iren"
```

Для визначення кількості елементів, що містяться у лінійному списку, застосовується функція `llen`. Ця команда повертає цілочисельне значення, яке вказує на довжину списку:

```
127.0.0.1:6379> llen visitors
(integer) 4
```

Команда `lrem` здійснює видалення першого елемента зі списку, значення якого збігається із заданим шаблоном. Кількість видалених елементів обмежується вказаним аргументом:

```
127.0.0.1:6379> lrem visitors 1 "Iren"
(integer) 1
127.0.0.1:6379> lrange visitors 0 2
1) "Kseniia"
2) "Pavlo"
3) "Ivan"
```

Команда `lpop` здійснює операцію видалення та повернення першого елемента зі списку. Іншими словами, вона вилучає елемент, розташований на початку списку, та повертає його значення користувачу:

```
127.0.0.1:6379> lpop visitors
"Kseniia"
```

База даних Redis надає команду `poplpush`, яка реалізує атомарну операцію переміщення елемента між двома списками. Конкретно, вона видаляє останній елемент з одного списку та додає його на початок іншого.

Команда ltrim скорочує список до вказаного числа:

```
127.0.0.1:6379> ltrim visitors 0 1  
OK
```

Питання для самоперевірки

Що таке в Redis типу «список»?

Яким чином в Redis можливо створити список?

Чим відрізняються команди lpush і rpush?

Чим відрізняються команди lpush, rpush і lpushx, rpushx?

Як забезпечується механізм оновлення списку?

Як отримати діапазон елементів списку?

Для чого призначена команда lindex?

Як визначити кількість елементів у списку?

Як видалити елемент списку за шаблоном?

Як видалити перший елемент списку?

Як скоротити список до вказаного числа?

Завдання

Варіант	Опис завдання
1	Для віртуального торговельного майданчика потрібно зберігати списки товарів у хеші. В текстовому файлі написати сценарій, який реалізує наступний функціонал: 1. У базі зберігаються тип товару і його назва через список. 2. Додати п'ять елементів списку з права і один елемент зліва. 3. Змінити третій елемент списку. 4. Перший елемент видалити. 5. Вивести перших три елемента списку. 6. Скопіювати зміст поточної бази даних в базу даних під номером 5.

2	Для мікросервісів «Розумного поля» потрібне спільне збереження даних в хеші. В текстовому файлі написати сценарій, який реалізує наступний функціонал: 1.База даних знаходиться під індексом 1. 2.У базі зберігаються назви полів (наприклад, «огіркове поле», «полуничне поле» тощо). Потрібно продумати назви ключів. 3.Додати п'ять елементів до списку зліва і один справа. 4.Переглянути перший елемент списку. 5. Видалити елемент за шаблоном. 6.Вивести випадковий ключ із поточної бази даних.
3	Для інформаційної системи потрібно збереження даних про користувачів в хеші. В текстовому файлі написати сценарій, який реалізує наступний функціонал: 1.У базі зберігаються ніки користувачів. Потрібно продумати назви ключів. 2.Додати три елемента списку зліва і три справа. 3.Визначити довжину списку. 4.Перший елемент видалити. 5. Видалити елемент за шаблоном. 6.Поточну базу очистити
4	Для мікросервісів «Розумна освіта» потрібне спільне збереження даних в хеші. В текстовому файлі написати сценарій, який реалізує наступний функціонал: 1.У базі зберігаються назви дисциплін. Потрібно продумати назви ключів. Створити список з дисциплінами. 2.Додати п'ять елементів до існуючого списку. 3.Визначити довжину списку. 4.Переглянути всі елементи списку. 5.Вивести випадковий ключ із поточної бази даних.

	6.Скопіювати інформацію з поточної бази в базу даних під індексом 1.
5	Для мікросервісів «Розумного місто» потрібне спільне збереження даних в хеші. В текстовому файлі написати сценарій, який реалізує наступний функціонал: 1.База даних знаходиться під індексом 1. 2.У базі зберігаються назви ресторанів. Потрібно продумати назви ключів. Створити список з назвами ресторанів. 3. Додати п'ять елементів списку з права. 4. Визначити довжину списку. 5.Перший елемент видалити. 6.Вивести випадковий ключ із поточної бази даних.

4.4.3 Тип даних «Sets». У базі даних Redis тип Sets представляє собою неупорядковані множини унікальних рядкових значень, що асоціюються з певним ключем. Кожен елемент набору називається членом. Що можна зберігати під цим типом даних? Наприклад, список онлайн-користувачів, які зараз активні на сайті або список сторінок, які відвідав користувач. Для соціальних мереж – це може бути список друзів користувача. Для пошукових систем можна зберігати список ключових слів для пошуку документа.

Redis забезпечує швидкий доступ до елементів Sets. Тут автоматично видаляються дублікати. Redis підтримує широкий спектр операцій над наборами об'єктами цього типу, таких як об'єднання, перетин, різниця. Sets ефективно використовують пам'ять, оскільки зберігають лише унікальні значення.

Команда sadd забезпечує функціональність створення множини типу Sets та додавання до неї одного або кількох елементів. У наступному прикладі буде створено набір список товарів у кошику за ключем під назвою basketOfgoods з елементами «Vivobook 17» і «Notebook HP 255»:

```
127.0.0.1:6379> sadd basketOfgoods "Vivobook 17" "Notebook HP 255"  
(integer) 2
```

Щоб переглянути елементи множини (набір, тип Sets), потрібно виконати команду `smembers`:

```
127.0.0.1:6379> smembers basketOfgoods  
1) "Notebook HP 255"  
2) "Vivobook 17"
```

Щоб дізнатися, чи є в наборі елемент, потрібно виконати команду `sismember`:

```
127.0.0.1:6379> sismember basketOfgood "Vivobook 15"  
(integer) 0
```

Як видно із прикладу, команда повернула значення 0. Це тому, що у множині немає елемента “Vivobook 15”.

Перевірити кількість елементів у наборі надає можливість команда `scard`:

```
127.0.0.1:6379> scard basketOfgoods  
(integer) 2
```

Redis містить три команди, які використовуються для видалення елементів із набору: `spop`, `srem` і `smove`. Перша команда видаляє елементи випадковим чином, команда `srem` дозволяє видаляти конкретні елементи множини, команда `smove` використовується, щоб перемістити елемент з одного набору до іншого. Також Redis дозволяє об'єднувати два набори, або знаходити їх перетин.

Питання для самоперевірки

Що таке множина у Redis?

Чи мають місце у множині дублікати?

Як створити множину у Redis?

Як переглянути елементи множини?

Як перевірити кількість елементів у множині?

Як дізнатися, чи є в множині елемент?

Завдання

Варіант	Опис завдання
1	Для мікросервісів «Розумного місто» потрібне спільне збереження даних в хеші. В текстовому файлі написати сценарій, який реалізує наступний функціонал: 1. У базі зберігаються назви ресторанів у форматі «string». Додати п'ять рядків. 2. Створити множину з адресами ресторанів. 3. Додати п'ять елементів множини. 4. Визначити довжину множини. 5. Переглянути всі елементи множини. 6. Переглянути рядок по ключу.
2	Для віртуального торговельного майданчика потрібно зберігати списки товарів у хеші. В текстовому файлі написати сценарій, який реалізує наступний функціонал: 1. У базі зберігаються тип товару і його назва через множину. 2. Додати п'ять елементів множини. 3. Змінити третій елемент множини. 4. Дізнатися кількість елементів у множині. 5. Видалити випадковий елемент множини. 6. Скопіювати зміст поточної бази даних в базу даних під номером 1.
3	Для мікросервісів «Розумного поля» потрібне спільне збереження даних в хеші. В текстовому файлі написати сценарій, який реалізує наступний функціонал: 1. База даних знаходиться під індексом 1. 2. У множині зберігаються назви полів (наприклад, «огіркове поле», «полуничне поле» тощо). 3. Додати п'ять елементів до множини.

	4.Переглянути елементи множини. 5. Видалити перший елемент множини. 6.Вивести випадковий ключ із поточної бази даних.
4	Для інформаційної системи потрібно збереження даних про користувачів в хеші. В текстовому файлі написати сценарій, який реалізує наступний функціонал: 1.У множині зберігаються ніки користувачів. Потрібно продумати назви ключів. 2.Додати три елемента до множини. 3.Визначити довжину множини. 4.Останній елемент видалити. 5. Випадковий елемент множини видалити. 6.Поточну базу очистити
5	Для мікросервісів «Розумна освіта» потрібне спільне збереження даних в хеші. В текстовому файлі написати сценарій, який реалізує наступний функціонал: 1.У базі зберігаються назви дисциплін. у форматі «string». Додати п'ять рядків. 2.Створити множину, яка зберігає прізвища студентів Додати п'ять елементів до множини. 3.Визначити довжину множини. 4.Переглянути всі елементи множини. 5.Вивести випадковий ключ із поточної бази даних. 6.Скопіювати інформацію з поточної бази в базу даних під індексом 1.

4.4.4 Тип даних «Hashes». Хешування – це процес перетворення даних довільної довжини в бітову послідовність фіксованої довжини, або хеш. У контексті індексації даних, хеш-функції використовуються для створення хеш-таблиць, які забезпечують швидкий доступ до даних за ключем. Фіксована

довжина хеш-значення дозволяє ефективно організувати дані в пам'яті та оптимізувати пошукові операції. Крім того, хешування використовується для забезпечення цілісності даних, оскільки будь-яка зміна вхідних даних призводить до значних змін у хеш-значенні.

Хеш-таблиця – це динамічна структура даних, що базується на принципі хешування. Кожному елементу, що зберігається в таблиці, відповідає унікальний ключ. Застосування хеш-функції до ключа дозволяє визначити індекс комірки, в якій буде зберігатися відповідне значення. Цей підхід забезпечує швидкий доступ до даних за ключем.

Оцінка ефективності типових операцій (вставка, видалення, пошук) над стандартними контейнерами за допомогою асимптотичного аналізу надана в таблиці 4.2.

Таблиця 4.2. Оцінка ефективності типових операцій над стандартними контейнерами

Контейнер \ Операція	Insert	Delete	Find
Set	$O(N)$	$O(N)$	$O(N)$
List	$O(1)$	$O(1)$	$O(N)$
Sorted Set	$O(N)$	$O(N)$	$O(\log N)$
Hashes	$O(1)$	$O(1)$	$O(1)$

Дані під типом «Hashes» варто використовувати, коли маються об'єктні дані. Тобто, якщо ви зберігаєте дані, які логічно представляють об'єкти з різними атрибутами (наприклад, користувач з ім'ям, прізвищем, електронною поштою тощо). Також під цим типом доречно зберігати словники та асоціативні масиви, коли потрібно зберегти множину пар ключ-значення, де ключі є унікальними всередині одного хешу. З типом «Hashes» зберігаються дані з результатів складних обчислень або запитів до бази даних, щоб уникнути повторних обчислень.

Операції над хешами, такі як отримання, встановлення та видалення полів, виконуються дуже швидко. Можна зберігати різні типи даних у полях хешу (числа, рядки, списки). Операції над хешами є атомарними, що гарантує їх цілісність. Хеші зазвичай займають менше місця в пам'яті, ніж окремі ключі для кожного поля. Хеші в Redis дуже швидко оновлюються, тому вони ідеально підходять для даних, які часто змінюються. Для дуже великих наборів даних або часто потрібно читати весь хеш повністю можуть бути більш ефективними інші структури даних, такі як списки або множини.

Процес створення або оновлення хешу ініціюється командою `hset`:

```
127.0.0.1:6379> hset user1215:cart good_1 10
(integer) 1
127.0.0.1:6379> hset user1215:cart good_2 15
(integer) 1
127.0.0.1:6379>
```

У цьому прикладі в хеші зберігається купівельна корзина користувача торговельного майданчика (`user1215:cart` – це ключ хешу, який представляє кошик користувача з ID `user1215`, `good_1` та `good_2` – це поля хешу, які відповідають ID товарів, 10 та 15 – це значення полів, які вказують кількість кожного товару в кошику).

Команда `HSETNX` призначена для унікального встановлення значень полів у хешах. Якщо ви спробуєте використати її для поля, яке вже існує, нічого не зміниться.

Вміст кошика користувача торговельного майданчика можна переглянути за допомогою команди `hgetall`:

```
127.0.0.1:6379> hgetall user1215:cart
1) "good_1"
2) "10"
3) "good_2"
4) "15"
```

За допомогою команди `hmget` можна отримати інформацію по декільком полям хеша:

```
127.0.0.1:6379> hmget user1215:cart good_1 good_2
1) "10"
2) "15"
```

Для отримання вичерпного переліку всіх полів, що входять до складу конкретного хешу, необхідно виконати команду `hkeys`. Ця операція повертає масив, елементами якого є назви всіх полів, визначених у вказаному хеші:

```
127.0.0.1:6379> hkeys user1215:cart
1) "good_1"
2) "good_2"
```

Для отримання повного переліку значень, асоційованих з усіма полями конкретного хешу, необхідно виконати команду `hvals`. Результатом виконання цієї команди є масив, елементами якого є значення всіх полів, визначених у вказаному хеші.

Для видалення хешу виконується команда `hdel`:

```
127.0.0.1:6379> hdel user1215:cart good_2
(integer) 1
```

Питання для самоперевірки

Що таке хешування? Яка структура хеш-таблиці?

Коли варто використовувати дані з типом «Hashes»?

Чи можна зберігати різні типи даних у полях хешу?

Які переваги має цей тип?

Як створити, оновити, видалити хеш?

Як встановити унікальне значення полів?

Як отримати значення хеша по ключу?

Як отримати перелік всіх полів хешу?

Як отримати всі значення хешу?

Завдання

Варіант	Опис завдання
1	Для мікросервісів «Розумного місто» потрібне спільне збереження даних в хеші. В текстовому файлі написати сценарій, який реалізує наступний функціонал:

	1. У базі зберігаються назви ресторанів та їх адреса. Створити хеш 2. Додати п'ять елементів. 3. Переглянути вміст. 4. Отримати інформацію по декільком полям хешу. 5. Вивести всі назви полів хешу. 6. Видалити хеш.
2	Для віртуального торговельного майданчика потрібно зберігати списки товарів у хеші. В текстовому файлі написати сценарій, який реалізує наступний функціонал: 1. У базі зберігаються назви товарів, їх ціна і назва постачальника. Створити хеш 2. Додати п'ять елементів. 3. Переглянути вміст. 4. Отримати інформацію по декільком полям хешу. 5. Вивести всі назви полів хешу. 6. Видалити хеш.
3	Для мікросервісів «Розумного поля» потрібне спільне збереження даних в хеші. В текстовому файлі написати сценарій, який реалізує наступний функціонал: 1. У базі зберігаються назви полів, що росте на полі і прізвище відповідального агронома. Створити хеш 2. Додати п'ять елементів. 3. Переглянути вміст. 4. Отримати інформацію по декільком полям хешу. 5. Вивести всі назви полів хешу. 6. Видалити хеш.
4	Для інформаційної системи потрібно збереження даних про користувачів в хеші. В текстовому файлі написати сценарій, який

	реалізує наступний функціонал: 1. У базі зберігаються ніки та паролі користувачів. Створити хеш 2. Додати п'ять елементів. 3. Переглянути вміст. 4. Отримати інформацію по декільком полям хешу. 5. Вивести всі назви полів хешу. 6. Видалити хеш.
5	Для мікросервісів «Розумна освіта» потрібне спільне збереження даних в хеші. В текстовому файлі написати сценарій, який реалізує наступний функціонал: 1. У базі зберігаються назви дисциплін, кількість годин, прізвища лекторів. Створити хеш 2. Додати п'ять елементів. 3. Переглянути вміст. 4. Отримати інформацію по декільком полям хешу. 5. Вивести всі назви полів хешу. 6. Видалити хеш.

4.4.5 Тип даних «Sorted sets». У системі Redis упорядковані множини представляють собою структуру даних, схожу на множини, обидві з яких є колекціями унікальних рядків. Однак, відмінною рисою упорядкованих множин є наявність асоційованого з кожним елементом числового значення, яке називається «балом». Цей бал дозволяє впорядковувати елементи множини за зростанням або спаданням. Незважаючи на вимогу унікальності елементів, кілька елементів можуть мати однаковий бал.

Упорядковані множини (sorted sets) в Redis – це надзвичайно потужна структура даних, яка дозволяє зберігати унікальні елементи, кожен з яких пов'язаний з числовим значенням (балом). Ця особливість робить їх ідеальними для широкого спектру завдань, де потрібна як унікальність елементів, так і їх

впорядкування. Наприклад, якщо потрібно відстеження найактивніших користувачів за кількістю дій, найпопулярніших продуктів за кількістю продажів тощо, то найкраще зберігати їх у форматі упорядкованих множин. Іншим прикладом слугують рейтинги, тобто зберігання рейтингів гравців у грі, популярності постів у соціальних мережах тощо.

Для конструювання упорядкованих множин в Redis передбачена команда `zadd`. Синтаксис команди передбачає вказання імені множини (ключа) та послідовності пар «оцінка-елемент». Кожен елемент упорядкованої множини асоціюється з унікальною оцінкою, яка визначає його позицію в множині. Оцінки можуть бути як цілими, так і дробовими числами. Наприклад:

```
127.0.0.1:6379> zadd products:discounts 10 "Notebook Lenovo"
(integer) 1
```

команда створює нову відсортовану множину «Товари із знижками» та вставляє в неї елемент «Notebook Lenovo» із знижкою 10%.

За допомогою команди `zadd` можна поповнювати упорядковану множину новими елементами. Оцінки, які привласнюються елементам при додаванні, можуть бути довільними дійсними числами. Не існує вимоги до суміжності оцінок, тобто між ними можуть бути значні проміжки. Важливою особливістю упорядкованих множин є те, що різні елементи можуть мати однакові оцінки:

```
127.0.0.1:6379> zadd products:discounts 25 "Notebook HP" 5 "Vivobook 17" 50 "Vivobook 15"
(integer) 3
```

Команда `zadd` в Redis підтримує ряд опцій, які дозволяють контролювати поведінку додавання елементів до упорядкованих множин. Серед них: `nx` (додавання лише нових елементів), `xx` (оновлення лише існуючих елементів), `ch` (повернення кількості змін) та `incr` (інкремент оцінки елемента). Команда `zincrby` є спеціалізованою версією `zadd` для інкрементування оцінок.

Основним методом отримання елементів, що зберігаються у упорядкованій множині Redis, є використання команди `zrange`. Ця команда приймає як аргументи ключ, що ідентифікує множину, так і діапазон індексів. Індеси в Redis починаються з нуля, тобто перший елемент множини має індекс

0. Таким чином, вказавши діапазон індексів, можна отримати підмножину елементів, розташованих між заданими межами:

```
127.0.0.1:6379> zrange products:discounts 0 5
1) "Vivobook 17"
2) "Notebook Lenovo"
3) "Notebook HP"
4) "Vivobook 15"
```

Варто зазначити, що у випадку, коли упорядкована множина містить елементи з однаковими оцінками, команда `zrange` виконує додаткове лексикографічне сортування цих елементів. Тобто, елементи з однаковими оцінками будуть розташовані у відсортованому результаті в алфавітному порядку.

Індекси, що використовуються для визначення діапазону елементів, можуть бути як додатними, так і від'ємними. Позитивні індекси починаються з нуля і вказують на позицію елемента з початку множини. Від'ємні індекси відраховуються з кінця множини, причому індекс `-1` відповідає останньому елементу, `-2` – передостанньому і так далі:

```
127.0.0.1:6379> zrange products:discounts -5 -1
1) "Vivobook 17"
2) "Notebook Lenovo"
3) "Notebook HP"
4) "Vivobook 15"
127.0.0.1:6379> zrange products:discounts -4 -2
1) "Vivobook 17"
2) "Notebook Lenovo"
3) "Notebook HP"
```

Команда `zrange` в Redis за замовчуванням повертає елементи упорядкованої множини у неспадному порядку за їхніми балами. Для отримання елементів у спадному порядку використовується команда `zrevrange`. Слід зазначити, що команда `zrevrange` не змінює порядок елементів у самій множині, а лише тимчасово інвертує порядок при виведенні результату. Таким чином, в команді `zrevrange` індекс `0` відповідає останньому елементу множини, індекс `1` – передостанньому і так далі:

```
127.0.0.1:6379> zrange products:discounts -4 -2
1) "Vivobook 17"
2) "Notebook Lenovo"
3) "Notebook HP"
127.0.0.1:6379> zrevrange products:discounts 0 5
1) "Vivobook 15"
2) "Notebook HP"
3) "Notebook Lenovo"
4) "Vivobook 17"
```

Команда `zrevrange` дозволяє додатково вказати параметр `WITHSCORES`, що забезпечує повернення не лише самих елементів, але й їхніх відповідних балів. Для отримання діапазону елементів за заданим діапазоном балів слід використовувати команду `zrangebyscore`:

```
127.0.0.1:6379> zrangebyscore products:discounts 0 5
1) "Vivobook 17"
```

Команда `zrangebyscore` підтримує опції `WITHSCORES` та `LIMIT` для більш тонкого контролю над результатом. Параметр `LIMIT` дозволяє обмежити кількість повернутих елементів, задаючи зсув та розмір підмножини:

```
127.0.0.1:6379> zrangebyscore products:discounts 0 5 WITHSCORES
1) "Vivobook 17"
2) "5"
```

Команда `zrangebylex` забезпечує лексикографічне сортування елементів відсортованої множини в тих випадках, коли звичайне сортування за балами не дозволяє однозначно визначити порядок.

Команда `zcard` служить для визначення розміру упорядкованої множини, тобто кількості елементів, які вона містить:

```
127.0.0.1:6379> zcard products:discounts
(integer) 4
```

Команда `zcount` дозволяє підрахувати елементи відсортованої множини, бали яких лежать в заданому діапазоні. Це дозволяє оцінити розподіл елементів за балами:

```
127.0.0.1:6379> zcount products:discounts 10 50
(integer) 3
```

Команда `zscore` виводить оцінку вказаного члена відсортованого набору:

```
127.0.0.1:6379> zscore products:discounts "Notebook Lenovo"
"10"
```

Команда `zrank` подібна до команди `zscore`, але замість того, щоб повертати заданий бал елемента, замість цього повертається його ранг.

Команда `zrem` може видалити одного або кількох членів із відсортованого набору:

```
127.0.0.1:6379> zrem products:discounts "Notebook Lenovo"  
(integer) 1
```

Redis надає команди `zremrangebyrank`, `zremrangebyscore` та `zremrangebylex` для видалення елементів з відсортованої множини за заданим діапазоном.

Питання для самоперевірки

Яку структуру представляють упорядковані множини у Redis?

Чим відрізняється упорядкована множина від просто множини?

Коли слід використовувати упорядковані множини?

Яким чином створити упорядковану множину? Як додати до неї нові елементи?

Чи можуть мати елементи упорядкованих множин однакові оцінки?

Які опції має команда `zadd`?

Яка команда є спеціалізованою версією `zadd` для інкрементування оцінок?

Яким чином можливо отримати елементи упорядкованої множини? При цьому яким чином будуть розташовані елементи з однаковими оцінками?

Як отримати елементи упорядкованої множини у спадному порядку?

Що дозволяють опції `WITHSCORES` та `LIMIT`?

Як визначить розмір упорядкованої множини?

Як підрахувати елементи відсортованої множини, бали яких лежать в заданому діапазоні?

Як вивести оцінку вказаного елемента упорядкованої множини?

Як видалити елементи упорядкованої множини?

Завдання

Ва	Опис завдання
----	---------------

ріант	
1	Для мікросервісів «Розумна освіта» потрібне спільне збереження даних в хеші. В текстовому файлі написати сценарій, який реалізує наступний функціонал: 1. У базі зберігається інформація про студентів і їх рейтинг. Створити упорядковану множину. 2.Додати п'ять елементів. 3.Визначити розмір множини. 4. Переглянути всі елементи множини. 5. Вивести оцінку вказаного елемента множини. 6.Вивести ранг елемента множини. 7.Видалити елемент множини.
2	Для мікросервісів «Розумного місто» потрібне спільне збереження даних в хеші. В текстовому файлі написати сценарій, який реалізує наступний функціонал: 1.У базі зберігаються назви ресторанів та їх рейтинги. Створити упорядковану множину. 2.Додати п'ять елементів. 3.Визначити розмір множини. 4. Переглянути всі елементи множини. 5. Вивести оцінку вказаного елемента множини. 6.Вивести ранг елемента множини. 7.Видалити елемент множини.
3	Для віртуального торговельного майданчика потрібно зберігати списки товарів у хеші. В текстовому файлі написати сценарій, який реалізує наступний функціонал: 1.У базі зберігаються назви товарів і скидки на них. Створити упорядковану множину. 2.Додати п'ять елементів.

	3.Визначити розмір множини. 4. Переглянути всі елементи множини. 5. Вивести оцінку вказаного елемента множини. 6.Вивести ранг елемента множини. 7.Видалити елемент множини.
4	Для мікросервісів «Розумного поля» потрібне спільне збереження даних в хеші. В текстовому файлі написати сценарій, який реалізує наступний функціонал: 1.У базі зберігаються назви полів і рейтинги їхнього ґрунту. Створити упорядковану множину. 2.Додати п'ять елементів. 3.Визначити розмір множини. 4. Переглянути всі елементи множини. 5. Вивести оцінку вказаного елемента множини. 6.Вивести ранг елемента множини. 7.Видалити елемент множини.
5	Для інформаційної системи потрібно збереження даних про користувачів в хеші. В текстовому файлі написати сценарій, який реалізує наступний функціонал: 1. У базі зберігаються ніки та рейтинги користувачів. Створити упорядковану множину. 2.Додати п'ять елементів. 3.Визначити розмір множини. 4. Переглянути всі елементи множини. 5. Вивести оцінку вказаного елемента множини. 6.Вивести ранг елемента множини. 7.Видалити елемент множини.

4.4.6 Тип даних «Streams». Спочатку система управління базами даних Redis розроблялась як інструмент для хешування та тимчасового зберігання

даних в оперативній пам'яті. З виходом версії 2.0 було впроваджено механізм публікації-підписки (PUBLISH/SUBSCRIBE), що перетворило Redis на багатофункціональну платформу для обміну повідомленнями, тобто у якості брокера повідомлень. Сьогодні Redis широко використовується в реалізації систем реального часу, таких як чати та сервіси обміну повідомленнями, де модель публікації-підписки знаходить своє найбільш ефективне застосування.

Брокер повідомлень – це програмний компонент, який діє як посередник у взаємодії між різними частинами системи. Він приймає повідомлення від одних програм і доставляє їх іншим. Це дозволяє роз'єднати компоненти системи, зробити їх більш незалежними та масштабованими.

До основних функцій брокера повідомлень, як правило, відносять :

- приймання повідомлень;
- зберігання повідомлень;
- маршрутизація повідомлень;
- доставка повідомлень;
- безпека доставки.

Брокери повідомлень відіграють ключову роль у сучасній розподіленій обчислювальній архітектурі, забезпечуючи ефективну декомпозицію систем на слабо пов'язані компоненти. Такий підхід не лише спрощує розробку та тестування окремих модулів, але й значно підвищує масштабованість та надійність системи в цілому. Асинхронна модель обробки повідомлень, реалізована у брокерах, дозволяє оптимізувати використання ресурсів та забезпечити високу пропускну здатність. Крім того, брокери повідомлень пропонують гнучкі механізми маршрутизації та доставки, що дозволяє адаптувати систему до змінних умов та інтегрувати різноманітні технології.

Брокери повідомлень знайшли широке застосування в сучасних розподілених системах, охоплюючи широкий спектр доменів. Від мікросервісної архітектури, де вони забезпечують гнучку та масштабовану комунікацію між службами, до систем реального часу, які вимагають швидкої доставки критичних повідомлень, брокери повідомлень доводять свою

універсальність. Крім того, вони є незамінним інструментом для обробки транзакцій, гарантуючи їхню цілісність, та для побудови складних систем аналізу даних, де збираються та обробляються потоки даних з різноманітних джерел.

Модель публікації-підписки (Pub/Sub) є ефективним механізмом асинхронної передачі повідомлень у розподілених системах, що забезпечує слабку зв'язність між компонентами. Цей підхід нагадує механізм розповсюдження новин, де видавець створює контент, а підписники динамічно отримують оновлення, не маючи прямого зв'язку між собою. В основі Pub/Sub лежить концепція топіків – тематичних каналів, через які розповсюджуються повідомлення. Видавці публікують повідомлення в конкретні топіки, а підписники підписуються на ті топіки, які їх цікавлять. Такий підхід забезпечує високу масштабованість та гнучкість системи, дозволяючи легко додавати нових видавців та підписників без впливу на існуючу інфраструктуру.

Для того щоб СУБД Redis працювала у ролі брокера повідомлень, у файл конфігурації повинен бути внесеним рядок:

```
loadmodule /usr/lib/redis/modules/redisearch.so
```

Система управління базами даних Redis пропонує абстракцію списків, яка виявляється корисною для реалізації простих черг. Завдяки набору базових операцій та можливості блокування, списки Redis можуть ефективно використовуватися для побудови високопродуктивних систем реального часу, таких як чати, стрічки новин та системи взаємодії серверів. Цей підхід дозволяє забезпечити швидку обробку запитів та масштабованість систем.

Для забезпечення надійної доставки повідомлень у строго визначеному порядку часто використовуються механізми черг. Команда BLMOVE в Redis дозволяє реалізувати такі черги шляхом перенесення елементів між списками з гарантією збереження їхньої послідовності. Крім того, ця команда забезпечує блокування операції доти, доки в списку з'явиться новий елемент, що гарантує, що жоден елемент не буде пропущений.

Модель Pub/Sub в Redis передбачає асинхронну передачу повідомлень (подій) між клієнтами. Клієнт, бажаючи отримувати оновлення щодо певної теми, підписується на відповідний канал за допомогою команди SUBSCRIBE:

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

Видавці ж використовують команду PUBLISH для розповсюдження подій серед усіх підписаних клієнтів:

```
127.0.0.1:6379> PUBLISH myChanel "My news"
(integer) 0
```

Так само можна відписатися від каналу, використовуючи команди UNSUBSCRIBE або PUNSUBSCRIBE.

Потокові черги пропонують значно ширший спектр функціональних можливостей порівняно зі стандартними чергами. До них належать механізми групування повідомлень, асинхронної обробки (через зворотні виклики) та послідовного споживання за допомогою ітераторів. Для ініціалізації потоку повідомлень використовується команда:

```
127.0.0.1:6379> xadd myStream * wehreField:"Kyiv" whenField:"2025"
"1735927578326-0"
```

де команда XADD створює нове повідомлення у потоці та додає його до кінця потоку.

Команда XREAD використовується для читання повідомлень з одного або кількох потоків:

```
127.0.0.1:6379> Xread count 5 streams myStream 1
1) 1) "myStream"
   2) 1) 1) "1735927578326-0"
      2) 1) "wehreField:Kyiv"
      2) "whenField:2025"
127.0.0.1:6379>
```

COUNT 5 означає кількість повідомлень, які необхідно прочитати з потоку, а 1 вказує на те, з якого повідомлення необхідно почати читання.

Різні корисні команди для управління поточковими чергами в Redis та обробки повідомлень у цих чергах подані в таблиці 4.3.

Таблиця 4.3. Команди для змінних типу “Streams”

Команда	Функціонал
XGROUP	створення нової групи потоків або оновлення наявної
XREADGROUP	читання повідомлення із зазначеної групи потоків
XDEL	видалення повідомлення з потоку
XPENDING	отримання інформації про необроблені повідомлення у групі потоків

Питання для самоперевірки

З виходом якої версії Redis було впроваджено механізм публікації-підписки?

Що таке брокер повідомлень? Які основні функції брокера повідомлень?

Які переваги отримуються при застосуванні брокера повідомлень?

Де знайшли своє застосування брокери повідомлень? Як працює така модель? Як включити цей механізм в СУБД Redis?

Як підписатися на канал повідомлень у Redis?

Як опублікувати повідомлення?

Що таке механізм групування повідомлень?

Як ініціювати потік повідомлень?

Як прочитати повідомлення із потоку повідомлень?

Завдання

Варіант	Опис завдання
1	Для інформаційної системи потрібно зберегти дані про користувачів в хеші. У текстовому файлі написати сценарій, який реалізує наступний функціонал:

	1. Потрібно організувати спілкування користувачів. Кожен користувач підписаний на якусь групу потоків. Створити групу для потоків. 2.Створити п'ять потоків. 3.Прочитати групу потоків.
2	Для мікросервісів «Розумна освіта» потрібне спільне збереження даних в хеші. У текстовому файлі написати сценарій, який реалізує наступний функціонал: 1. Потрібно організувати спілкування студентів. Кожен користувач підписаний на якусь групу потоків. Створити групу для потоків. 2.Створити п'ять потоків. 3.Прочитати групу потоків.
3	Для мікросервісів «Розумного місто» потрібне спільне збереження даних в хеші. У текстовому файлі написати сценарій, який реалізує наступний функціонал: 1. Деякі мікросервіси публікують новини міста, деякі користувачі читають їх. Кожен користувач підписаний на якусь групу новин. Створити групу для потоків. 2.Створити п'ять потоків. 3.Прочитати групу потоків.
4	Для віртуального торгівельного майданчика організувати обмін відгуками про товар. В текстовому файлі написати сценарій, який реалізує наступний функціонал: 1.Деякі користувачі пишуть відгуки про товар, деякі їх читають. Відгуки згруповані по товару. Створити групу для потоків. 2.Створити п'ять потоків. 3.Прочитати групу потоків.
5	Для мікросервісів «Розумного поля» потрібне спільне

	збереження даних в хеші. В текстовому файлі написати сценарій, який реалізує наступний функціонал: 1.Деякі мікросервіси публікують інформацію про стан рослин, деякі мікросервіси читають їх читають їх. Кожен мікросервіс налаштований на свою групу рослин. Створити групу для рослин. 2.Створити п'ять потоків. 3.Прочитати групу потоків.
--	--

4.3.7 Тип даних «Bitmaps». Bitmaps в Redis представляють собою абстракцію, яка надає можливість здійснювати бітові операції над рядковими даними. Рядок у Redis інтерпретується як послідовність бітів, що дозволяє зберігати великі обсяги даних у компактному бінарному форматі. Такий підхід забезпечує високу ефективність використання пам'яті та швидкість виконання операцій над даними.

Завдяки компактному представленню даних, Bitmaps дозволяють ефективно використовувати пам'ять. Наприклад, для зберігання інформації про онлайн-користувачів, де кожен біт відповідає одному користувачу, Bitmaps вимагають значно менше місця, ніж традиційні структури даних. Redis забезпечує високу швидкість виконання операцій над Bitmaps, таких як встановлення, отримання та підрахунок бітів. Bitmaps знаходять широке застосування в різних сферах. Наприклад у статистиці для підрахунку кількості відвідувань веб-сторінок за днями, відстеження активності користувачів тощо.

Операція `setbit` встановлює або скидає значення біта в заданій позиції (зсуві) бітового масиву, який асоційований з вказаним ключем. Якщо ключ відсутній, створюється новий бітовий масив, ініціалізований нулями. Для забезпечення достатньої довжини масиву для розміщення біта в заданій позиції, масив динамічно розширюється, причому додані біти встановлюються в нульовий стан. Зсув біта обчислюється від нуля і обмежений 32-бітним беззнаковим цілим числом, що відповідає максимальному розміру бітового масиву 512 мегабайт:

```
127.0.0.1:6379> setbit user:product 0 1
(integer) 0
127.0.0.1:6379> setbit user:product 1 1
(integer) 0
```

Синтаксис команди наступний:

```
SETBIT key offset value
```

Команда GETBIT повертає значання біта ключа по заданому зсуву.

Синтаксис команди наступний:

```
GETBIT key offset
```

Приклад:

```
127.0.0.1:6379> setbit user:product 30 1
(integer) 0
127.0.0.1:6379> getbit user:product 30
(integer) 1
```

Команди, які можуть виконуватися з типом Bitmaps подані в таблиця 4.4.

Таблиця 4.4. Команди для типу bitmaps

Команда	Функціонал
BITCOUNT key [start] [end]	Підраховує кількість встановлених бітів у вказаному діапазоні.
BITOP operation destkey key1 [key2 ...]	Виконує логічні операції (AND, OR, NOT, XOR) над кількома бітовими масивами і зберігає результат в новому ключі.
BITPOS key bit [start] [end]	Знаходить позицію першого біта зі значенням bit (0 або 1) в заданому діапазоні.

Питання для самоперевірки

Що собою представляють структури bitmaps в Redis? Для чого вони призначені?

Як встановити або скинути значення біті? Як прочитати значення у біті?
Які команди корисні для типу bitmaps?

Завдання

Варіант	Опис завдання
1	Для мікросервісів «Розумного поля» потрібне спільне збереження даних в хеші. В текстовому файлі написати сценарій, який реалізує наступний функціонал: 1. Нехай для кожного поля в структурі bitmaps чи мають рослини хворобу (1), чи ні (0). Таких хвороб декілька (визначають кількість структур). Створити ці структури. 2. Порахувати кількість полів, на яких рослини з хворобою.
2	Для інформаційної системи потрібно збереження даних про користувачів в хеші. В текстовому файлі написати сценарій, який реалізує наступний функціонал: 1. Нехай для кожного дня в структурі bitmaps, зберігається інформація чи заходив користувач в інформаційну систему (1) чи ні (0). Створити структури на декілька днів. 2. Отримати користувачів, які користувалися інформаційною системою два дні поспіль.
3	Для мікросервісів «Розумна освіта» потрібне спільне збереження даних в хеші. В текстовому файлі написати сценарій, який реалізує наступний функціонал: 1. Нехай для кожного дня в структурі bitmaps, зберігається інформація чи користувався студент освітньою платформою (1) чи ні (0). Створити структури на декілька днів. 2. Отримати студентів, які користувалися освітньою платформою два дні поспіль хоча б один раз.

4	Для мікросервісів «Розумного місто» потрібне спільне збереження даних в хеші. В текстовому файлі написати сценарій, який реалізує наступний функціонал: 1. Нехай в структурі bitmaps зберігається інформація чи відкрите кафе (1) чи ні (0). Створити структуру. 2.Визначити кафе, яке зараз відкрите.
5	Для віртуального торговельного майданчика в текстовому файлі написати сценарій, який реалізує наступний функціонал: 1. Нехай для кожного дня в структурі bitmaps, зберігається інформація чи заходив користувач на торговельний майданчик (1) чи ні (0). Створити структури на декілька днів. 2.Отримати користувачів, які заходили на торговельний майданчик два дні поспіль.

4.4.8 Тип даних «Bitfields». Механізм бітових полів (Bitfields) в Redis надає можливість здійснювати маніпуляції над окремими цілочисельними значеннями, які зберігаються у вигляді бітових послідовностей всередині більшого бітового масиву. На відміну від Bitmaps, де весь масив розглядається як єдине ціле, Bitfields дозволяють структурувати дані шляхом визначення полів різної довжини, що забезпечує більш гнучку організацію інформації та спрощує виконання арифметичних операцій над окремими частинами даних.

Атомарність операцій та висока швидкість обробки даних є одними з ключових переваг використання бітових полів в Redis. Можливість упакувати декілька чисел в один бітовий масив дозволяє значно зменшити витрати пам'яті, а атомарність операцій гарантує цілісність даних навіть при одночасній модифікації з різних потоків. Це робить бітові поля ідеальним вибором для реалізації високопродуктивних систем, які вимагають гарантій цілісності даних.

Бітові поля в Redis знаходять широке застосування в тих випадках, коли необхідно ефективно зберігати і обробляти невеликі числові значення, такі як лічильники, гістограми, або дані про стан системи. Вони особливо корисні при

реалізації гейм-механік, систем керування чергами та інших сценаріях, що вимагають високої продуктивності та компактності зберігання даних.

Припустимо, що ми хочемо зберігати інформацію про користувача торговельного майданчика: його вік (5 бітів), суму останньої покупки (10 бітів) і зароблені бонуси (1 біт). Створимо бітовий масив і визначимо у ньому три поля:

```
127.0.0.1:6379> BITFIELD user:1 SET u5 0 25 SET u10 5 1023 SET u1 15 1
1) (integer) 0
2) (integer) 0
3) (integer) 0
127.0.0.1:6379>
```

Ця команда створить бітовий масив з ключем `user:1` і встановить в ньому значення: вік – 25, сума покупки – 1023, бонуси – 1.

Команди, які можуть виконуватися з типом Bitfields подані в таблиці 4.5.

Таблиця 4.5. Команди для типу Bitfields

Команда	Функціонал
GET	Зчитування значення
INCRBY	Інкремент/декремент
OVERFLOW	Перевірка на переповнення

Питання для самоперевірки

Як працює механізм бітових полів? Які переваги цієї структури?

Де знаходять застосування бітові поля?

Як створити бітовий масив?

Які команди працюють з бітовим полем в Redis?

Завдання

Варіант	Опис завдання
1	Для віртуального торговельного майданчика в текстовому

	файлі написати сценарій, який зберігає інформацію про користувачів: ім'я, вік, частотність користування майданчиком, сума останньої покупки, бонуси.
2	Для мікросервісів «Розумного поля» в текстовому файлі написати сценарій, який зберігає дані: ім'я поля, площа поля, кислотність поля, вологість ґрунту поля, температура ґрунту поля.
3	Для інформаційної системи в текстовому файлі написати сценарій, який зберігає інформацію про користувачів: ім'я, вага, зріст, тиск, температура.
4	Для мікросервісів «Розумна освіта» в текстовому файлі написати сценарій, який зберігає інформацію про студентів: ім'я, рік народження, курс, рейтинг.
5	Для мікросервісів «Розумного місто» в текстовому файлі написати сценарій, який зберігає інформацію про аптеки міста: назва, площа, рейтинг серед містян, відстань до центру міста.

4.3.9 Тип даних «Geospatial indexes». Геопросторове індексування – це високоспеціалізована методика, що застосовується в системах управління базами даних для оптимізації процесів зберігання та доступу до географічних даних. Суть методу полягає у використанні спеціалізованих структур даних, які дозволяють ефективно визначати просторове положення об'єктів в базі даних відносно заданого географічного регіону.

Ця технологія набула широкого застосування у галузях, що оперують великими обсягами геопросторових даних, зокрема в геоінформаційних системах, сервісах геолокації та системах просторового аналізу. Завдяки геопросторовому індексуванню, вказані системи здатні здійснювати швидкий пошук та аналіз даних на основі їхнього географічного розташування, уникнувши необхідності послідовного перегляду великих обсягів інформації.

Геопросторові індекси забезпечують суттєве підвищення швидкості виконання запитів, що базуються на географічних критеріях, порівняно з

традиційними методами доступу до даних. Завдяки ефективній організації даних в індексі, значно зменшується обсяг даних, які необхідно переглянути для відповіді на запит, що призводить до зниження навантаження на систему та підвищення її продуктивності. Геопросторові індекси дозволяють виконувати складні просторові операції, такі як визначення найближчих сусідів, перетинів об'єктів, буферних зон та інші, що є необхідними для проведення різноманітних геопросторових аналізів.

Використання геопросторових індексів у геоінформаційних системах дозволяє забезпечити швидке відображення великих обсягів географічних даних на електронних мапах, а також виконувати різноманітні просторові аналізи. Геопросторові індекси є основою для побудови ефективних навігаційних систем, які дозволяють визначати оптимальні маршрути, надавати інформацію про місцезнаходження об'єктів та орієнтуватися на місцевості. Вони використовуються для аналізу просторових розподілів різних екологічних явищ, таких як забруднення, зміни клімату, природні катаклізми, що дозволяє оцінювати стан навколишнього середовища та прогнозувати його зміни. Їх застосування є ефективним при аналізі географічної розподіленості клієнтів, визначення ефективних маркетингових стратегій, а також для сегментації ринку за географічним принципом.

Система управління базами даних Redis пропонує набір команд (GEO) для роботи з геопросторовими даними. Однак, на відміну від інших типів даних, що підтримуються Redis, геопросторові дані не мають власного, спеціалізованого типу. Замість цього, географічні координати (широта та довгота) кодуються за допомогою алгоритму Geohash та зберігаються у вигляді балів відсортованого множини (sorted set). Такий підхід дозволяє ефективно здійснювати пошук та аналіз геопросторових даних за допомогою стандартних операцій над відсортованими множинами.

Процес додавання нових елементів до геопросторового індексу в Redis є інтуїтивно зрозумілим. Розглянемо типовий сценарій: відстеження інфраструктури міста. Нехай ми маємо відсортовану множину з ім'ям «city»,

яка містить інформацію про ті об'єкти міста, що відстежуються. Кожен з них ідентифікується унікальним іменем (наприклад, «hospital»). Для того, щоб додати новий автомобіль до множини, необхідно виконати команду GEOADD:

```
127.0.0.1:6379> GEOADD city -100.87 26.13062 hospital  
(integer) 1
```

Команда GEOADD в Redis для додавання елементів до геопросторового індексу приймає наступні аргументи: ключ відсортованої множини, довгота, широта, ім'я елемента. Унікальний ідентифікатор, що асоціюється з кожною точкою на карті. Таким чином, команда GEOADD дозволяє одночасно вказати географічне розташування елемента та його унікальний ідентифікатор.

Для оновлення географічного розташування елемента достатньо повторно виконати команду GEOADD з новими координатами та тим самим іменем елемента. Оскільки відсортована множина не допускає дублікатів, повторне додавання елемента з тим самим іменем призведе до перезапису його попереднього розташування. Це забезпечує простий та ефективний механізм оновлення геопросторових даних в Redis.

Розглянемо сценарій, коли до вже існуючого геопросторового індексу «city» необхідно додати новий елемент. Припустимо, це буде парковка. Для цього виконаємо команду:

```
127.0.0.1:6379> GEOADD city -100.171971 36.120609 parking  
(integer) 1
```

Після додавання нового елемента до індексу виникає необхідність оцінити відстань між різними об'єктами. Для розрахунку відстані між двома точками, представленими в геопросторовому індексі Redis, використовується команда GEODIST. Ця команда дозволяє визначити відстань між двома елементами в кілометрах або милях:

```
127.0.0.1:6379> GEODIST city hospital parking km  
"1113.1250"
```

Щоб переглянути всі об'єкти в радіусі 2000 км від заданої точки, потрібно використати команду:

```
127.0.0.1:6379> GEORADIUS city -100.87 26.11996 2000 km
1) "hospital"
2) "parking"
```

Також можна переглянути всі об'єкти відносно іншого об'єкта в радіусі:

```
127.0.0.1:6379> GEORADIUSBYMEMBER city parking 2000 km
1) "hospital"
2) "parking"
```

Якщо до двох останніх команд додати параметр WITHDIST, то можна ще дізнатися відстань між об'єктами.

```
127.0.0.1:6379> GEORADIUSBYMEMBER city parking 2000 km WITHDIST
1) 1) "hospital"
   2) "1113.1250"
2) 1) "parking"
   2) "0.0000"
```

З цими командами працює параметр WITHCOORD для виведення інформації про географічні координати об'єкта.

Для видалення об'єкта цього типу із набору використовується команда ZREM.

Питання для самоперевірки

Що таке геопросторове індексування? Де застосовується така технологія?

Що дозволяють виконувати геопросторові індекси?

За допомогою якого алгоритму кодуються координати в Redis?

За допомогою якої команди можна створити геопросторові координати в Redis?

Як оцінити відстань між об'єктами в Redis?

Як переглянути всі об'єкти в радіусі від заданої точки?

Як переглянути всі об'єкти відносно іншого об'єкта?

Завдання

Варіант	Опис завдання
1	За допомогою Google Maps визначити координати вашого

	дому і п'яти найближчих кафе. Занести цю інформацію до бази даних. Визначити відстань від вашого дому до кафе. Вивести всі кафе, що знаходяться в радіусі одного кілометра від вашого дому.
2	За допомогою Google Maps визначити координати вашого дому і чотири найближчих лікувальних закладів. Занести цю інформацію до бази даних. Визначити відстань від вашого дому до лікувального закладу. Вивести всі лікувальні заклади, що знаходяться в радіусі п'ятсот метрів від вашого дому.
3	За допомогою Google Maps визначити координати вашого дому і шість найближчих відділень банку. Занести цю інформацію до бази даних. Визначити відстань між відділеннями банку. Вивести всі відділення банку, що знаходяться в радіусі п'ятсот метрів від вашого дому.
4	За допомогою Google Maps визначити координати вашого дому і п'ять найближчих супермаркетів. Занести цю інформацію до бази даних. Визначити відстань від вашого дому до супермаркету. Вивести всі супермаркети, що знаходяться в радіусі п'яти кілометрів від вашого дому.
5	За допомогою Google Maps визначити координати вашого дому і чотирьох найближчих освітніх закладів. Занести цю інформацію до бази даних. Визначити відстані між освітніми закладами. Вивести всі освітні заклади, що знаходяться в радіусі п'яти кілометрів від вашого дому.

4.5 Транзакції в Redis

Механізм транзакцій у Redis забезпечує виконання групи команд як єдиної атомарної операції. Це означає, що всі команди в межах транзакції або виконуються успішно, або жодна з них не виконується, гарантуючи таким

чином цілісність даних, особливо при послідовному виконанні декількох операцій.

Транзакція ініціюється командою MULTI та завершується командою EXEC. Між цими командами в буфер поміщаються всі команди, які мають бути виконані.

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379(TX)> set myKey "Василь"
QUEUED
127.0.0.1:6379(TX)> set keyMy "Костянтин"
QUEUED
127.0.0.1:6379(TX)> exec
1) OK
2) OK
```

Якщо під час виконання транзакції виникає помилка, жодна з команд не застосовується до бази даних, і всі зміни відкочуються. Для примусового переривання транзакції використовується команда DISCARD.

Однією з ключових особливостей транзакцій є їхня атомарність. Це означає, що транзакція виконується як єдина нероздільна операція, що забезпечує послідовність виконання команд та ізоляцію від інших операцій, які виконуються одночасно. Однак, важливо зазначити, що транзакції в Redis не забезпечують повної ізоляції від збоїв системи. У разі збою сервера, результати невиконаної транзакції можуть бути непередбачуваними.

Транзакції в Redis знаходять широке застосування в сценаріях, де необхідно гарантувати цілісність даних при виконанні декількох пов'язаних операцій. Наприклад, при переведенні коштів з одного рахунку на інший, необхідно одночасно зменшити баланс на одному рахунку та збільшити баланс на іншому. Використання транзакції гарантує, що обидві операції будуть виконані або не будуть виконані зовсім, що запобігає втраті коштів.

Незважаючи на свої переваги, транзакції в Redis мають певні обмеження. Так, використання транзакцій може знизити продуктивність системи, оскільки під час виконання транзакції інші клієнти не можуть виконувати команди. Крім того, транзакції не забезпечують повної ізоляції від збоїв системи.

Для більш складних сценаріїв, що вимагають більшої гнучкості та потужності, можуть бути використані альтернативні механізми, такі як Lua-скрипти або команда WATCH. Lua-скрипти дозволяють виконувати складні алгоритми всередині Redis, тоді як команда WATCH дозволяє стежити за змінами даних і скасовувати транзакцію, якщо дані були змінені іншими клієнтами. **Lua-скрипти** – це невеликі програми, написані на мові програмування Lua, які виконуються безпосередньо на сервері Redis. Вони надають розробникам значно більше можливостей для взаємодії з даними, ніж стандартні команди Redis.

Питання для самоперевірки

1. Що забезпечує механізм транзакції?
2. Як почати транзакцію в Redis?
3. Як виконати транзакцію?
4. Як перервати транзакцію?
5. Яка ключова особливість транзакції?
6. Що означає атомарність операцій?
7. Де застосовуються транзакції?
8. Які обмеження транзакцій?
9. Що таке **Lua-скрипти**?

Завдання

Варіант	Опис завдання
1	Написати і виконати транзакцію: створюється два ключа із рядковим вмістом, два із списком. Скопіювати інформацію з поточної бази в базу даних під індексом 2.
2	Написати і виконати транзакцію: створюється два ключа із рядковим вмістом, два – із множиною. Переглядається вміст

	множин.
3	Написати і виконати транзакцію: створюється два ключа із списками, два із упорядкованими множинами. Скопіювати інформацію з поточної бази в базу даних під індексом 1.
4	Написати і виконати транзакцію: створюється два ключа із хешем та із упорядкованою множиною. Вивести випадковий ключ.
5	Написати і виконати транзакцію: створюється три ключа із хешем, геопросторовим індексом, із списком.

4.6 Тривалість життя ключів в Redis

Тривалість життя ключа (TTL – Time To Live) в Redis – це проміжок часу, після якого ключ автоматично видаляється з бази даних. Ця функція є дуже корисною для керування розміром бази даних та автоматичного видалення застарілих даних.

Існує кілька команд для встановлення TTL. Команда EXPIRE встановлює TTL для ключа в секундах:

```
127.0.0.1:6379> EXPIRE keyMy 60
(integer) 1
```

Команда PEXPIRE встановлює TTL для ключа в мілісекундах:

```
127.0.0.1:6379> PEXPIRE myKey 60
(integer) 1
```

Команда SETEX Встановлює значення для ключа і одночасно встановлює для нього TTL:

```
127.0.0.1:6379> SET myExample "Chacking!" EX 3600
OK
```

Щоб перевірити TTL ключа, можна скористатися командою TTL:

```
127.0.0.1:6379> TTL myExample
(integer) 3454
```

яка повертає залишковий час життя ключа в секундах. Якщо ключ не має TTL, повертається -1. Команда Якщо вказати від’ємне значення для TTL, ключ буде

видалений негайно. PTTL повертає залишковий час життя ключа в мілісекундах:

```
127.0.0.1:6379> PTTL myExample  
(integer) 3317902
```

Одним із ключових сценаріїв використання TTL є оптимізація хешів. Завдяки TTL застарілі дані автоматично видаляються з хешу, що забезпечує актуальність інформації та ефективне використання пам'яті.

Ще одним важливим застосуванням є управління сесіями користувачів. Встановлюючи TTL для сесійних даних, можна автоматично завершувати сесії після певного періоду бездіяльності користувача, підвищуючи тим самим рівень безпеки системи.

Крім того, TTL може бути використаний для реалізації простих систем планування завдань. Шляхом встановлення відповідного TTL для ключа, можна ініціювати виконання певної дії через заданий проміжок часу.

Нарешті, TTL є ефективним інструментом для контролю розміру бази даних. Автоматичне видалення застарілих даних дозволяє уникнути переповнення бази даних та оптимізувати використання ресурсів системи.

Загалом, механізм TTL є невід'ємною частиною арсеналу розробника, який працює з Redis. Його правильне застосування дозволяє створювати більш ефективні, надійні та масштабовані системи.

Питання для самоперевірки

1. Що таке тривалість життя ключа?
2. Як встановити тривалість життя ключа?
3. Які є сценарії використання тривалості життя ключа?

4.7 Взаємодія застосунків із СУБД Redis

Для ефективної взаємодії з системою керування базами даних Redis в середовищі програмування C# рекомендується застосовувати бібліотеку StackExchange.Redis. Цей високопродуктивний клієнт надає розширений набір

інструментів для встановлення з'єднання з сервером Redis та виконання широкого спектру операцій над даними, що зберігаються в базі. Для встановлення цієї бібліотеки після створення проєкту потрібно скористатися функціоналом NuGet (рис. 4.5).

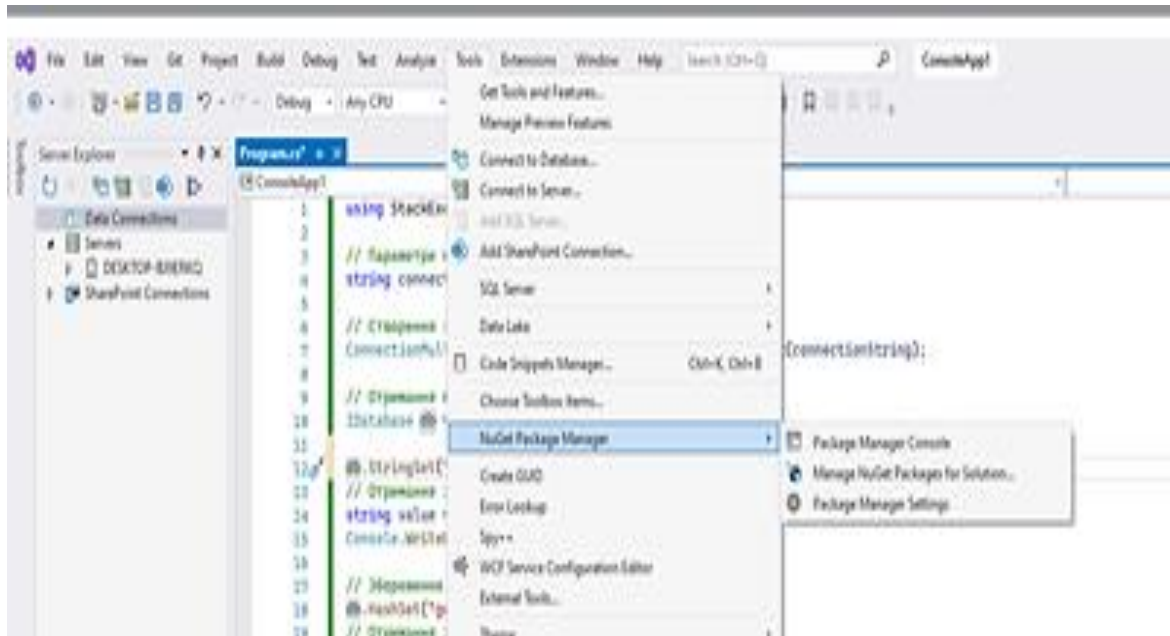


Рисунок 4.5 – Функціонал NuGet

Для того, щоб підключитися до Redis, потрібно створити рядок з'єднання

```
// Рядок з'єднання  
string connectionString = "localhost:6379";
```

В цьому випадку буде прослуховуватися порт 6379, який Redis використовує по замовчуванню. Але бувають випадки, що налаштування комп'ютера або операційної системи не дозволяють використовувати саме цей порт, в такому разі потрібно вказати потрібний порт.

Далі створюється з'єднання за допомогою класу ConnectionMultiplexer:

```
// Створення з'єднання  
ConnectionMultiplexer redis = ConnectionMultiplexer.Connect(connectionString);
```

В кінці програми завжди потрібно закрити з'єднання:

```
redis.Close();
```

Після створення з'єднання потрібно отримати базу даних:

```
// Отримання бази даних
IDatabase db = redis.GetDatabase();
```

Для того, щоб була можливість отримати базу даних, потрібно щоб працював сервер Redis. Якщо сервер був запущений щойно, то потрібно заповнити базу даними:

```
127.0.0.1:6379> set product:1 "Notebook HP"
OK
```

Щоб прочитати ці дані із бази даних Redis, потрібно користатися командою:

```
// Отримання значення по ключу
string value = db.StringGet("product:1");
//Виведення отриманого значення
Console.WriteLine(value);
```

В результаті виконання цього коду у консоль буде виведено:

```
Notebook HP
D:\Project\Redis\ConsoleApp1\ConsoleApp1\bin\Debug\net6.0\ConsoleApp1.exe (process 13364) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Для додавання рядка в Redis використовують команду `db.StringSet(«key», «value»)`.

Для роботи з хешем призначені функції:

```
// Збереження хеш-значення
db.HashSet("goods:it", "proc", "i7");
// Отримання значення з хеша
string hashValue = db.HashGet("goods:it", "proc");
Console.WriteLine(hashValue); // "value1"
```

Для роботи з елементами списку використовуються `db.ListRightPush(«myList», «Item1»)` – для додавання елемента до списку, `db.ListRange` – для читання списку.

Для видалення ключа використовують – `db.KeyDelete(«key»)`.

Питання для самоперевірки

1. Що таке рядок з'єднання?
2. Який порт по замочуванню призначений для Redis?

Завдання

Варіант	Опис завдання
1	Написати два застосунки: перший записує у базу даних Redis інформацію про геопросторові координати руху машини (наприклад, змінює один раз в хвилину). Другий зчитує цю інформацію і виводить повідомлення про вміст інформації.
2	Написати два застосунки: перший записує у базу даних Redis випадкові оголошення, другий їх зчитує і виводить на екран монітора.
3	Написати два застосунки: перший записує у базу даних Redis інформацію про вміст купівельної корзини. Другий зчитує цю інформацію і виводить повідомлення про її вміст.
4	Написати два застосунки: перший записує в базу даних Redis інформацію про успішність студентів. Другий застосунок зчитує цю інформацію і виводить її на екран монітора.
5	Написати два застосунки: перший записує в базу даних Redis інформацію про знижки на товари. Другий застосунок зчитує цю інформацію і виводить повідомлення про вміст інформації.

Рекомендована література до розділу [30-33].

ЛІТЕРАТУРНІ ДЖЕРЕЛА

1. Educational SQL resources. URL: <https://learn.microsoft.com/en-us/sql/sql-server/educational-sql-resources?view=sql-server-ver16>.
2. Електронний ресурс. URL: https://learn.ztu.edu.ua/pluginfile.php/204794/mod_resource/content/0/lr7.%202sem_Adm.pdf
3. Електронний ресурс. URL: <https://www.sqlservertutorial.net/sql-server-administration/sql-server-create-role/>
4. Електронний ресурс. URL: <https://learn.microsoft.com/en-us/sql/t-sql/statements/create-role-transact-sql?view=sql-server-ver16>
5. Погромська Г.С., Махровська Н.А. Базы даних: проектування та реалізація. – Миколаїв: МНУ ім. В.О. Сухомлинського, 2019. – 183 с.
6. Лосєв М. Ю. Базы даних : навчально-практичний посібник для самостійної роботи студентів [Електронний ресурс] / М. Ю. Лосєв, В. В. Федько. – Харків : ХНЕУ ім. С. Кузнеця, 2018. – 233 с. Доступ: http://repository.hneu.edu.ua/bitstream/123456789/21468/1/2018-Лосєв_М_Ю%2С_Федько_В_В.pdf
7. Васильєв Д. О., Горлова Т. В. Методичні вказівки щодо виконання лабораторних робіт з навчальної дисципліни «Організація баз даних та знань» для студентів денної форми навчання зі спеціальності 122 – «комп'ютерні науки» за освітньо-професійною програмою «комп'ютерні науки» освітнього ступеня «бакалавр». Кременчук: МОН України Кременчуць. нац. ун-т ім. Михайла Острогр., 2020. 58 с.
8. Методичні вказівки до лабораторної роботи «Створення баз даних за допомогою мови Transact SQL» з курсу «Проектування баз даних» для студентів спеціальності 122 Комп'ютерні науки та інформаційні технології, з курсу «Базы даних» спеціальності 186 Видавництво та поліграфічна справа та з курсу «Проектування інформаційних систем»

- спеціальності 124 Системний аналіз / уклад.: Л. Б. Кащев, С. В. Коваленко, О. С. Куценко – Харків : НТУ «ХПІ», 2018. – 16 с.
9. Рябцев В. В. Методичні рекомендації щодо забезпечення самостійної роботи студентів з дисципліни «Бази даних» (для бакалаврів спеціальностей 121 Інженерія програмного забезпечення та 122 Комп'ютерні науки та інформаційні технології). – К. : МАУП, 2018. – 24 с.
10. Програмування баз даних: конспект лекцій [Електронний ресурс] : навч. посіб. Для студ. спеціальності 152 «Метрологія та інформаційно-вимірювальна техніка» / М. В. Добролюбова ; КПІ ім. Ігоря Сікорського. – Електронні текстові дані (1 файл: 22,7 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2021. – 275 с. Доступ: <https://ela.kpi.ua/handle/123456789/43918>
11. Методичні рекомендації до виконання лабораторних робіт з навчальної дисципліни «Принципи проектування баз даних та баз знань» для студентів напряму підготовки 6.051501 «Видавничо-поліграфічна справа» всіх форм навчання : [Електронне видання] / уклад. О. К. Пандорін. – Х. : ХНЕУ ім. С. Кузнеця, 2015. – 44 с.
12. Мулеса О.Ю. Основи мови запитів SQL. – Ужгород, 2015. – 48 с.
13. Буяк Л.М., Мушак А.Я., Хома Н.Г. Працюємо з базами даних в середовищі Microsoft Office: теоретичні аспекти та приклади розв'язування задач Навчальний посібник з курсу „Сучасні інформаційні технології” для студентів денної та заочно-дистанційної форм навчання галузей знань 07 „Управління та адміністрування”, 23 „Соціальна робота” ступеня вищої освіти „бакалавр”. – Тернопіль: ТНЕУ, 2019. – 80 с.
14. Бардус І. О. Бази даних у схемах (на основі фундаменталізованого підходу) : навч. посіб. /І. О. Бардус, М. І. Лазарєв, А. О. Ніценко. – Харків: «Діса плюс», 2017. – 133 с.
15. Мулеса О.Ю. Інформаційні системи та реляційні бази даних. Навч.посібник. – Електронне видання, 2018. – 118 с.

- 16.Лосєв М. Ю. Організація баз даних та знань (ADO.NET): конспект лекцій / М. Ю. Лосєв, О. В. Тарасов, В. В. Федько. – Х. : Вид. ХНЕУ, 2011. – 108 с.
- 17.Федько В. В. Організація баз даних та знань : навч.-прак. посібн. / В. В. Федько, О. В. Тарасов, М. Ю. Лосєв. – Х. : Вид. ХНЕУ, 2013. – 200 с.
- 18.Johnson G. Exam 70-516: TS: Accessing Data with Microsoft .NET Framework 4 / G. Johnson. – Microsoft Press, 2011. – 671 p.
- 19.Lerman J. Programming Entity Framework, Second Edition. / J. Lerman. – O'Reilly, 2010. – 914 p.
- 20.Lerman J. Programming Entity Framework: Code First / J. Lerman, R. Miller. – O'Reilly Media, 2011. – 194 p.
- 21.Lerman J. Programming Entity Framework: DbContext / J. Lerman, R. Miller. – O'Reilly Media, 2012. – 258 p.
- 22.Rattz J. Pro LINQ: Language Integrated Query in C# 2010 / J.Rattz, A. Freeman. – Apress, 2010. – 656 p.
- 23.Технології проєктування No Sql баз даних [Електронний ресурс] : навч. посіб. для здобувачів ступеня магістра за освіт. програмою «Інформаційне забезпечення робототехнічних систем» спец. 126 Інформаційні системи та технології / Є. В. Крилов, В. К. Анікін, ; КПІ ім. Ігоря Сікорського. – Електрон. текст. дані (1 файл). – Київ : КПІ ім. Ігоря Сікорського, 2023. – 117 с.
- 24.Бази даних NoSQL. Конспект лекцій [Електронний ресурс]: навч. посіб. для студ. спеціальності 121 «Інженерія програмного забезпечення», освітньої програми «Інженерія програмного забезпечення мультимедійних та інформаційно-пошукових систем», які вивчають навчальну дисципліну «Програмне забезпечення інформаційно-пошукових систем 1. Бази даних NoSQL» / К. О. Радченко, А. В. Петрашенко; КПІ ім. Ігоря Сікорського. – Електронне мережне навчальне видання. – Київ: КПІ ім. Ігоря Сікорського, 2023. – 270 с.

25. Borucki A. MongoDB in Action, Third Edition: Building on the Atlas Data Platform 3rd Edition. Manning, 2025. 576 p.
26. Електронний ресурс: <https://www.mongodb.com/products/self-managed/community-edition>
27. Електронний ресурс: <https://learn.mongodb.com>
28. Електронний ресурс: <https://www.mongodb.com/docs/manual/data-modeling/>
29. Електронний ресурс: <https://www.codecademy.com/article/mongo-db-data-modeling-basics>
30. Redis: The Complete Developer's Guide. Master Redis v7.0 with hands-on exercises. Includes Modules, Scripting, Concurrency, and Streams! URL: <https://www.udemy.com/topic/redis>
31. Електронний ресурс: <https://redis.io/docs/latest/>
32. Бази даних NoSQL. Конспект лекцій [Електронний ресурс]: навч. посіб. для студ. спеціальності 121 «Інженерія програмного забезпечення», освітньої програми «Інженерія програмного забезпечення мультимедійних та інформаційно-пошукових систем», які вивчають навчальну дисципліну «Програмне забезпечення інформаційно-пошукових систем 1. Бази даних NoSQL» / К. О. Радченко, А. В. Петрашенко; КПІ ім. Ігоря Сікорського. – Електронне мережне навчальне видання (1 файл: 3,45 Мбайт). – Київ: КПІ ім. Ігоря Сікорського, 2023. – 270 с.
33. Redis: The Complete Developer's Guide. Master Redis v7.0 with hands-on exercises. Includes Modules, Scripting, Concurrency, and Streams! URL: <https://www.udemy.com/topic/redis>.