Міністерство освіти і науки України
Чорноморський національний університет імені Петра Могили

Фісун М. Т., Дворецький М. Л.,
Ніколенко С. Г., Дворецька С. В.

Організація баз даних. Цикл лабораторних робіт у середовищі СКБД SQLite

Методичні вказівки для підготовки бакалаврів у галузі знань «Інформаційні технології»

Випуск 269

ЧНУ
ВИДАВНИЦТВО
Миколаїв – 2019

УДК 004.65(076.5)
О 64

Рекомендовано до друку вченою радою ЧНУ ім. Петра Могили (протокол № 6 від «28» лютого 2019 р.).

Рецензенти:
Приходько Сергій Борисович – доктор технічних наук, професор, завідувач кафедри програмного забезпечення автоматизованих систем Національного університету кораблебудування імені адмірала Макарова;
Устенко Сергій Анатолійович – доктор технічних наук, професор, завідувач кафедри комп'ютерних систем та мереж Миколаївського національного університету імені В. О. Сухомлинського.

О 64

Фісун М. Т. Організація баз даних. Цикл лабораторних робіт в середовищі СКБД SQLite. Методичні вказівки для підготовки бакалаврів у галузі знань «Інформаційні технології» / М. Т. Фісун, М. Л. Дворецький, С. Г. Ніколенко, С. В. Дворецька. – Миколаїв : Вид-во ЧНУ ім. Петра Могили, 2019. – 64 с. (Методична серія ; вип. 269).

У методичних вказівках для підготовки бакалаврів у галузі знань «Інформаційні технології» розглянуто основні положення щодо баз даних та зокрема реляційної моделі даних, нормалізації відношень та ключових відмінностей щодо файл-серверних та клієнт-серверних систем управління базами даних. Наведено загальний огляд мови SQL, на прикладі СКБД SQLite, приведено етапи створення БД, таблиць та індексів, імпорт-експорт, резервне копіювання та відновлення даних. На детальних прикладах висвітлено основні оператори команди вибірки даних мови SQL. Також приводиться огляд утиліти SQLite Studio для роботи з БД SQLite з використанням графічного інтерфейсу. Наприкінці розглянуто приклад створення клієнтського застосунку із використанням мови PHP.

УДК 004.65(076.5)

ISSN 1811-492X

ЗМІСТ

ВСТУП

Одною з ознак інформаційного суспільства є широке використання інформаційних систем у різних галузях діяльності. Інформаційною системою називають взаємозв'язані дані та засоби утворення, зберігання, редагування, пошуку та обробки для аналізу та прийняття рішень. Ядром інформаційної системи є база даних – сукупність певним чином структурованих даних, між якими встановлені зв'язки. Бази даних формуються і працюють під управлінням спеціальних програмних засобів, що називаються системами керування базами даних (СКБД).

SQLite – це вбудовувана кросплатформна СКБД, що підтримує досить повний набір команд мови SQL, що є стандартною для роботи з реляційними БД. Її програмний код на мові С є у вільному доступі. Тексти програм SQLite знаходяться в public domain, тобто взагалі немає ніяких обмежень на використання. Особливістю SQLite є те, що вона не використовує парадигму клієнт-сервер, тобто ядро SQLite не є окремим процесом, з яким взаємодіє застосунок, а надає бібліотеку, з якої кінцева програма вибирає необхідні модулі при компіляції.

SQLite зберігає всю базу даних (включаючи визначення, таблиці, індекси і дані) в єдиному файлі на тому пристрої, на якому виконується застосунок. Завдяки архітектурі ядра можливо використовувати SQlite як на вбудовуваних (embedded) системах, так і на виділених машинах з гігабайтними масивами даних.

1. ЗАГАЛЬНИЙ ОГЛЯД ТЕОРЕТИЧНИХ ОСНОВ БД

1.1. Поняття та визначення бази даних

Існує декілька визначень поняття бази даних, для початку наведемо та порівняємо їх властивості.

База даних – організована відповідно до певних правил і підтримувана в пам'яті комп'ютера сукупність даних, що характеризує актуальний стан певної предметної області і використовується для задоволення інформаційних потреб користувачів.

База даних – спільно використовуваний набір логічно пов'язаних даних (і опис цих даних), призначений для задоволення інформаційних потреб організації.

База даних – це сукупність відомостей щодо конкретних об'єктів реального миру в якій-небудь предметній області. Під предметною областю маємо на увазі частину реального світу, що підлягає вивченню для організації керування й автоматизації.

У визначеннях найчастіше (явно або неявно) присутні такі відмінні ознаки:

– БД зберігається і обробляється в обчислювальній системі. Таким чином, будь-які накопичені масиви даних (архіви, бібліотеки, картотеки і т. д.) не можна назвати базою даних.

– Дані у БД логічно структуровані (систематизовані) з метою забезпечення можливості їх ефективного пошуку і обробки в обчислювальній системі. Структурованість передбачає явне виділення складових частин (елементів), зв'язків між ними, а також типізацію елементів і зв'язків.

– БД включає схему або метадані, що описують їх логічну структуру у формальному вигляді (відповідно до певної моделі даних).

Наведемо ще одне визначення бази даних. БД – це іменована сукупність взаємопов'язаних даних, що перебувають під керуванням СКБД.

Система керування базами даних (СКБД) – це комплекс програмних і мовних засобів, необхідних для створення баз даних, підтримки їх в актуальному стані, агрегації й пошуку в них необхідної інформації. Нижче наведено основні вимоги до СКБД.

– СКБД має сприймати й обробляти команди користувачів і програм на вибірку, зміну, додавання або видалення даних. Таким чином, у СКБД повинен бути компонент, відповідальний за виконання цих дій, спеціальна мова обробки даних. Найчастіше цією мовою є який-небудь різновид мови SQL.

– СКБД повинна мати можливість приймати дані у вхідній формі з різних за своєю природою джерел і перетворювати їх у форму, що відповідає власним об'єктам. Тобто СКБД має містити компоненти перетворення форматів даних.

– СКБД повинна мати функції забезпечення безпеки, цілісності, а у випадку пошкодження, відновлення інформації, що зберігається у базі даних.

– Бажано, щоб у СКБД були реалізовані механізми оптимізації виконання перерахованих вище операцій. Тобто має бути ряд компонентів, відповідальних за максимальну ефективність виконання СКБД своїх функцій.

1.2. Файл-серверні та клієнт-серверні СКБД

Файловий сервер. БД розташовується на файловому сервері (або декількох файлових серверах), у якості якого може використовуватися найбільш потужний комп'ютер із об'єднаних у мережу. Функції файлового сервера полягають у зберіганні баз даних і забезпеченні доступу до них користувачів, що працюють на різних (клієнтських)

комп'ютерах. Файли бази даних відповідно до користувальницьких запитів передаються на робочі станції, де, в основному, й відбувається обробка. Передані дані обробляються СКБД, що також перебувають на комп'ютерах користувачів. Після того як користувачі виконають необхідні операції з даними, вони копіюють файли назад на файловий сервер, де інші користувачі, у свою чергу, зможуть з ними працювати. Крім того, кожен користувач може створювати на локальному комп'ютері свої власні бази даних, використовувані ним монопольно.

Ця схема підходить для невеликих обсягів даних. При збільшенні числа комп'ютерів у мережі або росту обсягів БД, продуктивність інформаційної системи різко падає. Це пов'язано зі збільшенням обсягу даних, переданих мережею, оскільки вся обробка відбувається на комп'ютері користувача. Недоліком подібного підходу є висока ймовірність втрати змін, виконаних користувачами, під час збереження змінених файлів на центральному сервері. Справа у тому, що користувачі можуть і не підозрювати, що окрім них ще хтось змінював дані, і тільки через деякий час помітять, що дані перебувають не в тому стані, що очікувалося. Прикладами СКБД, призначених безпосередньо для розробки локальних користувальницьких додатків БД, тобто додатків, що працюють на одному локальному комп'ютері, або в комп'ютерній мережі є Microsoft Visual FoxPro, Microsoft Access, Paradox for Windows, dBase for Windows та ін.

Клієнт-сервер. Технологія клієнт-сервер передбачає, що окрім зберігання бази даних центральний комп'ютер (сервер бази даних) має ще й забезпечувати виконання основного обсягу обробки даних. При технології клієнт-сервер запит клієнта (робочої станції) на виконання якої-небудь операції з даними (наприклад, звичайної вибірки) провокує на сервері пошук і добування даних. Витягнуті дані (але не файли) транспортуються мережею від сервера до клієнта.

Системи, що використовують технологію клієнт-сервер, складаються з двох частин:

– клієнтська частина (Front-End) забезпечує інтерфейс користувача (у тому числі графічний) і перебуває на його комп'ютері;

– серверна частина (Back-End) забезпечує керування даними, поділ інформації, адміністрування й безпеку. Серверна частина перебуває на спеціально виділених для цього комп'ютерах-серверах. Прикладами СКБД, що працюють за технологією клієнт-сервер, є Microsoft SQL Server, Oracle, MySQL, Firebird та ін.

1.3. Моделі даних. Реляційна модель даних

Крім наведеної вище класифікації, СКБД можна класифікувати і за використовуваною моделлю (або структурі) даних. За допомогою моделі даних можна наочно представити структуру об'єктів і встановлення між ними зв'язку. Модель даних визначається такими характеристиками:

– структурні (первинні, базові) елементи, вміст яких визначає вміст БД;
– взаємозв'язки між структурними елементами;
– мова (мови) маніпулювання даними;
– засоби забезпечення цілісності БД.

Нині найбільш поширеною моделлю даних є реляційна модель, автором якої є американський вчений Е. Кодд. Будучи математиком, він запропонував використовувати для обробки даних апарат теорії множин (об'єднання, перетинання, різниця, декартів добуток) і предикативної логіки для того, щоб внести в область керування базами даних чіткі принципи й точність математики. Він довів, що будь-який набір даних можна представити у вигляді двовимірних таблиць особливого виду, відомих у математиці як «відношення». Від англійського слова «relation» («відношення») і

відбулася назва «Реляційна модель даних». Тому повернемося до математичної основи реляційної моделі, тобто до алгебраїчного визначення відношення.

Якщо a і b – об'єкти, то через (a,b) позначають упорядковану пару, рівність яких визначається наступним чином:

(a,b) = (c,d), якщо a=c & b=d.

Прямим (декартовим) добутком двох множин A і B називається множина упорядкованих пар, у якій перший елемент кожної пари належить A, а другий – B:

AxB = {(a,b)|a∈A & b∈B}.

Бінарним відношенням R з множини A до множини B називається підмножина прямого добутку A і B:

ARB = {(a,b)|a∈A & b∈B & (aRb = true)} ⊆ AxB,

тобто воно складається з пар (a,b), де a∈A, b∈B, якщо є правдивим заданий тип висловлювання, визначеного на елементах цієї пари. Вище для бінарних відношень використана інфіксна форма запису. За такого запису R означає те висловлювання, що задає тип відношення, наприклад, «дисципліна (курс) m викладається в семестрі n», «клієнт s замовив товар q» та ін.

Приклад. Маємо два базових (унарних) відношень:

D = {Програмування, Вища математика, Фізика} – дисципліни;

S = {1,2,3,4}– номери семестрів навчального плану.

Тоді декартів добуток **D**x**S** = {(Програмування, 1); {(Програмування, 2); …, (Фізика, 4)}, тобто отримаємо 12 пар. Практичного сенсу така інформація не несе окрім того, що вона показує усі можливі варіанти пар (дисципліна, семестр).

Якщо ж ми визначимо відношення **DRS,** де висловлювання dRs означає «дисципліна d викладається в семестрі s», то, відповідно до гіпотетичного навчального плану, отримаємо множину таких пар: **DRS=**{(Програмування, 1); (Програмування, 2); (Вища математика, 1); (Вища математика, 2); (Вища математика, 3); (Фізика, 2); (Фізика, 3)}. Наведене відношення **DRS** як множина, є підмножиною декартового добутку **D**x**S**. Наведені вище базові відношення та їх декартів добуток і сформульоване відношення між ними наведено на рис. 1.3.1 у вигляді двовимірних таблиць, які є основою реляційної моделі даних.

D
Dysc
Програмування
Вища математика
Фізика

S
Sem
1
2
3
4

Dx**S**	
Dysc	**Sem**
Програмування	1
Програмування	2
Програмування	3
Програмування	4
Вища математика	1
Вища математика	2
Вища математика	3
Вища математика	4
Фізика	1
Фізика	2
Фізика	3
Фізика	4

DRS	
Dysc	**Sem**
Програмування	1
Програмування	2
Вища математика	1
Вища математика	2
Вища математика	3
Фізика	2
Фізика	3

Рис. 1.3.1. Базові сутності, їх декартів добуток та відношення

Назви колонок в реляційних таблицях називаються *атрибутами*. Вони відображають властивості інформаційних об'єктів, що зазначені у таблицях. Рядки реляційних таблиць називають кортежами. На відміну від звичайної класичної алгебри в реляційній, як моделі даних, крім самого факту «відношення» між таблицями

для користувачів представляють інтерес властивості цих відношень. Наприклад, відношення «дисципліна» (D) має властивість «обсяг дисципліни, кред» (**Volume**), відношення (**S**) має властивість «кількість тижнів» (**Weeks**), відношення «дисципліна d викладається у семестрі s» (**DRS**, яке нижче позначатиметься як таблиця **D_S**) може мати такі властивості, як «аудиторних годин у семестрі» (**Aud),** «вид контролю» (**Control**) та ін., як показано на фрагментах таблиць (рис. 1.3.2).

D	
Dysc	**Volume**
Вища математика	10.0
Програмування	5.0
Фізика	5.0

S	
Sem	**Weeks**
1	15
2	18
3	15

D_S			
Dysc	**Sem**	**Aud**	**Control**
Вища математика	2	60	Залік
Вища математика	3	90	Іспит
Фізика	2	60	Залік

Рис. 1.3.2. Фрагменти таблиць D, S та D_S

Але не кожна таблиця представляє відношення. Оскільки вона має відповідати певним вимогам, тому перед її формулюванням введемо поняття і визначення ключа відношення.

Будь-яке відношення має один або декілька атрибутів, значення яких однозначно ідентифікують кожен кортеж. Такий атрибут (або комбінація атрибутів) називається *первинним ключем* (primary key). Для базових таблиць потенційні ключі, у тому числі й первинний, є простим, тобто він складається з одного атрибута. Так, для таблиці **D** первинним ключем є атрибут **Dysc**, а для таблиці **S** – атрибут **Sem**. Для таблиці **D_S,** яка представляє бінарне відношення між таблицями **D** і **S** первинним ключем є сукупність атрибутів <**Dysc**, **Sem**>.

Часто вводять додаткове поле для нумерації записів у таблиці, цей номер покликаний забезпечити унікальність кожного запису.

Однак, первинний ключ є окремим випадком з більш загальним поняттям – потенційного (або можливого) ключа. Кожне відношення має, принаймні, один потенційний ключ, оскільки, щонайменше, сукупність усіх його атрибутів задовольняє умові унікальності. Це є наслідком властивості відношень про те, що у відношенні не може бути однакових кортежів.

Один з можливих ключів обирається як первинний. Інші ключі, якщо вони є, приймаються за альтернативні. Наприклад, первинним ключем таблиці обрано ідентифікаційний номер фізичної особи, то номер її паспорта буде альтернативним.

Потенційний ключ повинен мати наступні властивості:

Унікальність. За будь-яких умов два кортежі відношення не повинні мати однакових комбінацій значень атрибутів, що входять до складу потенційного ключа. Наприклад, якщо у таблиці поле з номером квартири визначено як потенційний ключ, то в ній не повинно бути двох рядків, які мають однаковий номер квартири. Якщо ключем були обрані поля, номер квартири та номер будинку, то комбінації значень полів для цієї таблиці мають бути унікальними (допускаються однакові номери квартир, але у різних будинках).

Ненадмірність. Жоден із атрибутів складеного ключа не може бути виключений із нього без порушення унікальності. Це означає, що не варто створювати ключ, який містить номер паспорта, та ідентифікаційний номер. Досить обрати лише один із цих атрибутів, щоб однозначно ідентифікувати запис. Не варто також включати в ключ не унікальний атрибут, тобто комбінації ідентифікаційного номера та імені людини – при виключенні імені людини із ключа однаково можна буде унікально ідентифікувати кожен рядок.

Примітка. На практиці виконання умови унікальності є обов’язковим. Водночас за необхідності умова ненадмірності може бути порушена.

У реляційної моделі з поняттям первинного ключа пов'язане правило цілісності об'єктів. Це свідчення того, що первинний ключ базового відношення не має містити значень NULL.

З урахуванням визначення терміну «ключ» реляційна таблиця має відповідати таким як:

1. Комірки таблиці мають бути атомарними:

2. Кортежі мають бути унікальними, та визначатимуться унікальністю сукупності значень ключових атрибутів. Наприклад, таблиця **D** має такий вміст (рис. 1.3.3):

Dysc	**Sem**	**Aud**
Вища математика	3	90
Фізика	2	45
Вища математика	3	72

Рис. 1.3.3. Таблиця D

Наведена таблиця не відповідає вимозі, хоча усі кортежі унікальні, оскільки значення сукупності ключових атрибутів <Dysc, Sem> першого і третього атрибутів повторюються.

3. Назви атрибутів мають бути унікальними.

В алгебрі розглядається також узагальнене поняття відношення: `n`-арне відношення – це множина упорядкованих наборів (кортежів), що є результатом сполучення кортежів базових відношень і висловленнями над якими певне висловлення (предикат) приймає значення «істина» (true).

Відповідно до вищенаведеного в теорії реляційних баз даних відношення мають наступні властивості.

У відношенні немає однакових кортежів. Ця властивість випливає з того факту, що тіло відношення – це математична множина (множина кортежів). Множини ж за визначенням не містять однакових елементів. Але, як уже зауважувалося вище, унікальність кортежів визначається унікальністю ключа відношення, а не усього кортежу.

Примітка. Вищезазначена властивість може служити підтвердженням того, що відношення й таблиця не зовсім одне і теж, оскільки таблиця може містити однакові рядки, якщо це не заборонено якими-небудь правилами, тоді як відношення однакових кортежів містити не може.

Кортежі невпорядковані. Ця властивість також випливає з того, що тіло відношення – це математична множина. Прості ж множини є невпорядкованими. Тому, якщо в якому-небудь відношенні кортежі розташувати в іншому порядку, відношення залишиться таким самим. Отже, щодо відношення не можна сказати, що існує поняття, наприклад, другого або п'ятого кортежів.

Атрибути невпорядковані. З погляду реляційної теорії відношення, у якому атрибути представлені у такому порядку: <Прізвище, Зріст, Вага, Стать>, і відношення, у якого атрибути представлені в порядку <Стать, Прізвище, Вага, Зріст>, буде тим самим відношенням. Таким чином, не існує поняття, наприклад, першого, третього або останнього атрибута. Атрибут завжди визначається за іменем, а не за розташуванням. До речі, це допомагає уникнути плутанини під час написання програм.

Усі значення атрибутів відношення атомарні. У кожній позиції на перетинанні стовпця й рядка розташовано тільки одне значення, а не декілька. Таблиці, що задовольняють цій умові, перебувають у першій нормальній формі (докладніше про це див. далі).

Ми наголошували на тому, що реляційна теорія забороняє використання неатомарних значень атрибутів. Але в реальному житті бувають і протилежні випадки. Наприклад, іноді в історичних записах зустрічаються такі, як «Дата народження

невідома» або в паспортному столі: «Особа без певного місця проживання». Було запропоновано такий підхід, коли для подання відсутньої або невідомої інформації використовуються спеціальні маркери, названі значеннями NULL. Тобто, за наявності в кортежі значення NULL у якій-небудь позиції атрибута вважається, що в цій позиції значення атрибута відсутній.

1.4. Нормалізація та її необхідність

Під час використання універсальних (унарних) таблиць виникають наступні проблеми.

Проблема надмірності даних. Дані практично всіх стовпців багаторазово повторюються.

Проблема їх оновлення (або проблема потенційної суперечливості даних). Внаслідок надмірності можна оновити, наприклад, адресу постачальника в одному рядку, залишаючи її незмінною в інших. З'являються дві адреси, одна з яких уже недійсна. Отже, під час відновлення даних необхідно переглядати всю таблицю для знаходження й зміни всіх підходящих рядків.

Проблема додавання даних. Додавання нових даних вимагає введення значень для *всіх* стовпців, навіть якщо в таблиці вже існують необхідні дані. Рано чи пізно, при введенні даних, що дублюються, буде допущена помилка, яка приведе до виникнення двох різних значень. Крім того, не можна додати інформацію раніше, ніж вона знадобиться. Стосовно попереднього прикладу, це означає, що не можна ввести ім'я продавця, поки він не продасть який-небудь товар.

Проблема видалення даних. Під час видалення одного рядка може бути загублена потрібна інформація.

Примітка. Останні три проблеми прийнято називати *аномаліями відновлення, додавання й видалення.*

Для запобігання описаних проблем необхідно використовувати так звану нормалізацію.

Нормалізація таблиць – це формальний апарат обмежень на формування таблиць, що описує їх проекцію на дві або більше частин, що дозволяє застосування кращих методів додавання, оновлення й видалення даних.

Можна також сказати, що *нормалізація* – це процес подання даних у вигляді простих двовимірних таблиць, що дозволяє усунути їх дублювання і забезпечує несуперечність збереженої інформації в базі даних.

Таким чином, остаточною метою нормалізації є одержання проекту бази даних, у якому будь-яка частина інформації зберігається лише в одному місці, тобто виключається надмірність інформації. Це робиться не тільки для економії місця (у деяких випадках нормалізовані таблиці займають більше місця, чим ненормалізовані), а й для виключення можливості протиріччя у збережені даних.

Розглянемо на прикладах перші три нормальні форми, оскільки більш глибока нормалізація на практиці застосовується порівняно рідко.

Найважливіші на практиці нормальні форми відношень ґрунтуються на фундаментальному в теорії реляційних БД понятті функціональної залежності. Для цього введемо ряд визначень.

Визначення 1. У відношенні R атрибут Y функціонально залежить від атрибута X (X і Y можуть бути складеними атрибутами, тобто реально складатися з декількох атомарних атрибутів) тільки у тому випадку, якщо кожному значенню X відповідає лише одне значення Y: $R.X \rightarrow R.Y$.

Визначення 2. Функціональну залежність R.X→R.Y називають *повною,* якщо атрибут Y функціонально залежить від атрибута X і не залежить функціонально від будь-яких інших атрибутів.

Визначення 3. Функціональну залежність R.X→R.Y називають *транзитивною,* якщо існує такий атрибут Z, для якого R.X→R.Z і R.Z→R.Y.

Визначення 4. Потенційним (можливим) ключем відношення називають його окремий (атомарний) або складений атрибут (сукупність атрибутів), значення якого повністю функціонально визначають значення всіх інших атрибутів відношення. Це ще одне визначення потенційного ключа,

Визначення 5. Потенційний ключ, що обраний для ідентифікації рядків та операцій з ними і зв'язків з іншими відношеннями, називають *первинним.* Інші потенційні ключі називають *вторинними.*

Визначення 6. Неключовим атрибутом називають будь-який атрибут відношення, що не входить до складу первинного ключа.

Визначення 7. Два або більше атрибутів називають *взаємно незалежними,* якщо жоден з них не є функціонально залежним від інших атрибутів.

Перша нормальна форма. Відношення знаходиться в 1НФ, якщо всі його атрибути є простими (атомарними), усі кортежі і назви усіх атрибутів унікальні. Ці вимоги вже наводилися вище. Третя вимога є тривіальною. Приклад таблиць, що не відповідають другій вимозі, наводилися вище. Приклад таблиць, що не відповідають першій нормальній формі, тобто не відображають відношення, наведено нижче на рис. 1.4.1.

Semestr	Course
1	Історія України
	Вища математика
2	Вища математика
	Фізика
	Фізкультура

Semestr	Course /Контроль
4	Вища математика/Іспит
4	Фізика/Іспит
5	Фізкультура/Залік

Рис. 1.4.1. Приклад таблиць, що не відповідають першій нормальній формі

Нижче наведено ці таблиці після приведення до 1НФ (рис. 1.4.2).

Semestr	Course
1	Історія України
1	Вища математика
2	Вища математика
2	Фізика
2	Фізкультура

Semestr	Course	Control
4	Вища математика	Іспит
4	Фізика	Іспит
5	Фізкультура	Залік

Рис. 1.4.2. Таблиці, приведені до 1НФ

Друга нормальна форма. Відношення знаходиться в 2НФ, якщо воно знаходиться в 1НФ і кожен не ключовий атрибут функціонально залежить від первинного ключа (ПК).

Розглянемо такий приклад схеми відношення «Приміщення-Предмети»: **ROOMS_ITEMS(IdR,SECUR,DEP-R,IdP,QTY)**, де:

IdR – код приміщення;

SECUR – рівень режимності приміщення;

DEP-R – відділ, за яким закріплено приміщення;

IdP – код предмету;

QTY – кількість предметів у приміщенні.

На рисунку 1.4.3 наведено приклад поточного вмісту таблиці, що відповідає заданому відношенню **ROOMS_ITEMS**.

IdR	SECUR	DEP-R	IdP	QTY
R1	2	ІОЦ	P1	30
R1	2	ІОЦ	P2	20
R1	2	ІОЦ	P3	1
R1	2	ІОЦ	P4	2
R1	2	ІОЦ	P5	1
R1	2	ІОЦ	P6	3
R2	1	ККТ	P1	10
R2	1	ККТ	P2	5
R3	1	ККТ	P2	15
R4	2	ІОЦ	P2	40
R4	2	ІОЦ	P4	2
R4	2	ІОЦ	P5	1
R4	2	ІОЦ	P7	4

Рис. 1.4.3. Приклад поточного вмісту таблиці, що відповідає заданому відношенню **ROOMS_ITEMS**

У таблиці значення атрибуту DEP-R мають такі скорочення:
ІОЦ – інформаційно-обчислювальний центр;
ККТ – кафедра комп'ютерних технологій.

Ця таблиця знаходиться в 1НФ. Вона має всього один потенційний складений ключ **<IdR, IdP**>, тому він є первинним.

Функціональні залежності атрибутів :

IdR → SECUR; **IdR → DEP-R**; <**IdR , IdP**> **→ QTY**.

Крім того, в результаті додаткового аналізу таблиці можна побачити, що існує також функціональна залежність **DEP-R→ SECUR**.

На рисунку 1.4.4. такі функціональні залежності показані у вигляді схеми.

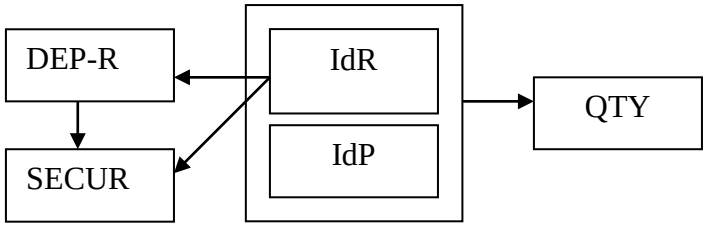

Рис. 1.4.4. Схема функціональних залежностей атрибутів відношення **ROOMS_ITEMS**

Як видно, хоча первинним ключем є складений атрибут **<IdR,IdP**>, атрибути **SECUR** і **DEP-R** функціонально залежать від його частини, тобто атрибута IdR. У результаті, не можна вставити у відношення **ROOMS_ITEMS** кортеж, який описує предмет, котрий ще не встановлений у жодному приміщенні (первинний ключ не може містити невизначене значення). Такі неприємні явища називають *аномаліями схеми відношення.* У цьому випадку це *аномалія додавання*.

Під час вилучення останнього кортежу із таблиці на рис. 1.4.4 не тільки руйнується зв'язок предмету **P7** з даним приміщенням **R4**, але й втрачається інформація, щодо предметів **P7,** які знаходяться у будинку (офісі), але на поточний час ще не встановлені. Це *аномалія видалення*.

У разі перенесення однакових предметів у інше приміщення необхідно модифікувати усі кортежі, які описують цей предмет, інакше отримаємо непогоджений результат. Це *аномалія оновлення*.

Такі неприємні явища (аномалії схеми відношення) усувають шляхом нормалізації.

Визначення 8. Відношення R знаходиться у *другій нормальній формі (2НФ)* лише тоді, коли воно знаходиться в 1НФ і кожний неключовий атрибут повністю залежить від первинного ключа.

Для приведення відношення **ROOMS_ITEMS** до другої нормальної форми можна провести наступну декомпозицію у два відношення: **ROOMS(IdR, SECUR, DEP-R)** та **RM_IT(IdR, IdP,QTY)**. Схему функціональних залежностей наведено на рис. 1.4.5, а вміст таблиць, що відповідають новим відношенням – на рис. 1.4.6.

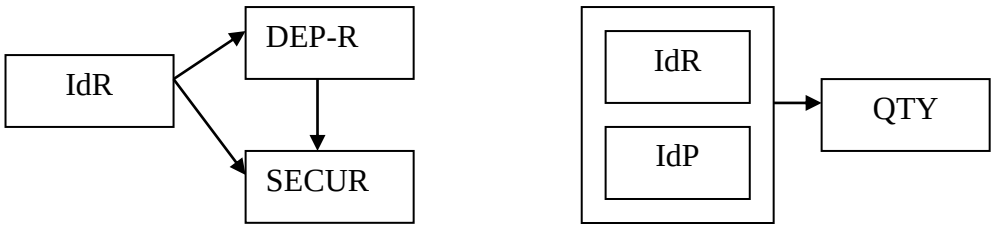

Рис. 1.4.5. Схеми функціональних залежностей відношень **ROOMS** та RM_IT

ROOMS

IdR	SECUR	DEP-R
R1	2	ІОЦ
R2	1	ККТ
R3	1	ККТ
R4	2	ІОЦ

RM_IT

IdR	IdP	QTY
R1	P1	30
R1	P2	20
R1	P3	1
R1	P4	2
R1	P5	1
R1	P6	3
R2	P1	10
R2	P2	5
R3	P2	15
R4	P2	40
R4	P4	2
R4	P5	1
R4	P7	4

Рис. 1.4.6. Вміст таблиць **ROOMS** та **RM_IT**

Первинним ключем відношення **ROOMS** є атрибут **IdR**, а відношення **RM_IT** – сполучення атрибутів **<IdR, IdP>**.

Кожне з двох відношень знаходиться у 2НФ, і в них усунені вищезазначені аномалії. Але існує інша аномалія, яку ми розглянемо нижче.

Третя нормальна форма. Відношення знаходиться у 3НФ, коли знаходиться у 2НФ і кожен не ключовий атрибут нетранзитивно залежить від первинного ключа.

Розглянемо ще раз відношення **ROOMS**, яке знаходиться у 2НФ. Відзначимо, що функціональна залежність **IdR→SECUR** є транзитивною, тобто вона є наслідком (у математичному розумінні) функціональних залежностей **IdR→DEP-R** та **DEP-R→SECUR.** Рівень режимності приміщення насправді є характеристикою **DEP-R** – відділу, за яким закріплено приміщення (це не дуже природне припущення, але достатнє для прикладу).

У результаті, не можна внести інформацію у БД щодо рівня режимності приміщення, доки це приміщення не буде закріплене за якимось підрозділом. У разі вилучення кортежу, що описує останній підрозділ, втрачається інформація щодо рівня режимності приміщення. Для того, щоб за погодженням змінити рівень режимності приміщення, необхідно заздалегідь знайти усі кортежі, що описують підрозділи. Таким чином, у відношенні **ROOMS** досі існують аномалії, які можна усунути шляхом подальшої нормалізації.

Визначення 9. Відношення R знаходиться у третій нормальній формі (3НФ) тільки в тому випадку, якщо воно знаходиться у 2НФ і кожен неключовий атрибут нетранзитивно залежить від первинного ключа.

Еквівалентним альтернативним визначенням третьої нормальної форми є визначення 9а.

Визначення 9а. Відношення R знаходиться в третій нормальній формі (3НФ) тільки у тому випадку, якщо всі неключові атрибути R взаємно незалежні і повністю залежать від первинного ключа.

Виконаємо декомпозицію відношення ROOMS у два відношення: ROOMS_DEP та DEP_SEC – та представимо схеми ФЗ усіх трьох відношень початкової БД (рис. 1.4.7).

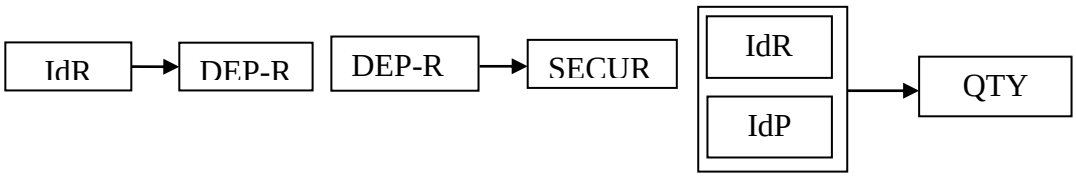

Рис. 1.4.7. Схеми ФЗ відношень **ROOMS_DEP**, **DEP_SEC** та **RM_IT**

Вміст таблиць, що відповідають новим відношенням, представлено на рис. 1.4.8.

Як бачимо, в усіх трьох відношеннях відсутні транзитивні залежності між атрибутами, тому кожне з трьох відношень знаходиться у 3НФ, і в них усунені вищезазначені аномалії.

ROOMS_DEP

IdR	DEP-R
R1	ІОЦ
R2	ККТ
R3	ККТ
R4	ІОЦ

DEP_SEC

DEP-R	SECUR
ККТ	1
ІОЦ	2

RM_IT

IdR	IdP	QTY
R1	P1	30
R1	P2	20
R1	P3	1
R1	P4	2
R1	P5	1
R1	P6	3
R2	P1	10
R2	P2	5
R3	P2	15
R4	P2	40
R4	P4	2
R4	P5	1
R4	P7	4

Рис. 1.4.8. Вміст таблиць, приведених до 3НФ

На практиці третя нормальна форма схем відношень у більшості випадків є достатньою і, як правило, зведенням до третьої нормальної форми закінчується процес проектування реляційної БД. Однак інколи доцільно продовжити процес нормалізації.

1.5. Взаємозв'язок відношень

Взаємозв'язок відношень є найважливішим елементом реляційної моделі даних. Вони підтримуються *зовнішніми ключами (foreign key)*. Так, на рис. 1.5 наведено фрагмент структури бази даних, що зберігає відомості про різні кафедри університету (таблиця departments), а також відомості про викладачів кафедр (таблиця lecturers). Первинним ключем таблиці викладачів є поле code, а поле dep_code (код кафедри) є зовнішнім ключем для зв'язку з таблицею Кафедри (departments).

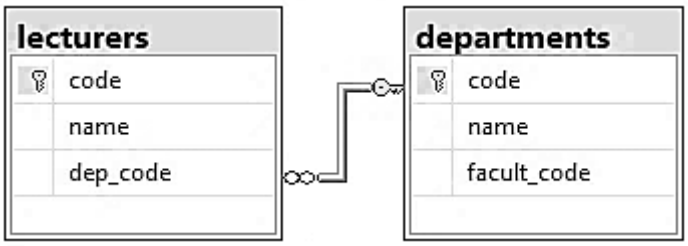

Рис. 1.5. Зв'язок таблиць «Викладачі» та «Кафедри»

Примітка. Реляційна теорія баз даних вимагає, щоб зовнішні ключі були позв'язані саме з первинними ключами, а не з потенційними. На практиці ця вимога, як правило, не дотримується, тобто дозволяється зв'язування зовнішнього ключа з потенційним.

Кожне значення атрибута зовнішнього ключа має бути значенням відповідного потенційного ключа. Причому зворотна ситуація необов'язкова. Тобто потенційний ключ, що відповідає зовнішньому ключеві, може містити значення, що у цей момент не є значенням зовнішнього ключа. Наприклад, у таблиці зі списком філій компанії можуть вказуватися ідентифікаційні номери країн. Інформація зберігається в окремій таблиці, що містить повний перелік країн з різною інформацією для кожної з них. Компанія може мати філії не в усіх країнах. Тобто на деякі з них посилань не буде.

Кількість атрибутів зовнішнього ключа має чітко відповідати кількості атрибутів потенційного ключа. Наслідком цього є те, що зовнішній ключ буде складовим (тобто складатися більш ніж з одного атрибута) тільки тоді, коли відповідний потенційний ключ також є складовим. Відповідно, зовнішній ключ буде простим тільки в тому випадку, якщо відповідний потенційний ключ є простим.

Для зовнішнього ключа не потрібно, щоб він був компонентом первинного або взагалі якого-небудь потенційного ключа у відношенні, до якого він належить. Таким чином, значення в зовнішньому ключі можуть повторюватися, що неможливо для потенційного. Наприклад, в одній і тій же країні може бути більше однієї філії. Відповідно, на ту саму країну може бути більше одного посилання.

Кожен атрибут, що входить у зовнішній ключ, визначатиметься на тому ж домені, що й відповідний атрибут потенційного ключа.

Для атрибутів зовнішнього ключа дозволяється мати значення NULL.

З поняттям зовнішнього ключа в реляційній теорії баз даних пов'язане так зване *правило посилальної цілісності.* Це правило говорить про те, що *база даних не повинна містити значень зовнішніх ключів, для яких не існує відповідних значень потенційних ключів.* Дійсно, коли що-небудь посилається на щось, те це щось повинне існувати.

Будь-який стан бази даних, не задовольняючий правилу посилальної цілісності, вважається некоректним. Яким же чином на практиці можна уникнути таких станів?

– По-перше, можна заборонити будь-які операції, що призводять до некоректного стану.

– По-друге, можна допустити виконання операції, але за умови виконання деяких додаткових дій, для того щоб система залишилася у коректному стані. Така операція називається операцією *каскадного видалення, або каскадної зміни.* І найкраще, забезпечити користувачеві можливість вибору конкретного варіанта залежно від ситуації.

– Можливий ще й третій варіант виходу із ситуації порушення посилальної цілісності. Він прийнятний у випадку, якщо в стовпцях зовнішнього ключа дозволене зберігання значень NULL. Тоді під час спроби видалення рядків з таблиці з первинним ключем відповідні значення зовнішнього ключа приймають значення NULL, після чого й відбувається видалення рядків. При цьому губиться інформація щодо приналежності

відповідних рядків таблиці із зовнішнім ключем. Під час спроби зміни значень первинного ключа відповідні значення зовнішнього ключа приймають значення NULI., а значення первинного обновляються. Інформація про зв'язок цих рядків зі значеннями первинного ключа також губиться.

Далі розглянемо типи зв'язків між відношеннями.

Один до одного (1:1). Зв'язок типу «один до одного» означає, що в кожен момент часу кожному представникові (екземпляру) сутності А відповідає 1 або 0 екземплярів сутності В. Говорячи мовою конкретної реалізації сутностей у вигляді таблиць, якщо конкретний рядок першої таблиці пов'язаний у кожен момент часу з нулем рядків або одним рядком другої таблиці, то між ними встановлене відношення «один до одного».

Як приклад відношення «один до одного» можна навести зв'язок людини й номера паспорта. Кожна людина може мати тільки один паспорт, і відповідно він належить тільки одній людині. У цьому прикладі немає рації створювати окремі таблиці для інформації про людину та її паспорт. Більш логічним буде зберігання відомостей в одній таблиці.

Ще одним прикладом буде зв'язок подружжя в якій-небудь європейській країні. Кожен чоловік може бути одночасно одружений тільки на одній жінці, відповідно і кожна жінка може бути одружена тільки з одним чоловіком. У цьому прикладі деякі рядки можуть бути незв'язаними (коли людина не є одруженою). У попередньому прикладі паспорт не може існувати самостійно, хоча людина може його і не мати.

Один до багатьох (1:M). Зв'язок типу «один до багатьох» означає, що одному представникові сутності А відповідають 0, 1 або декілька представників сутності В. При цьому, сутність А буде називатися *батьківською* сутністю, а сутність В – *дочірною.* Якщо говорити мовою конкретної реалізації сутностей у вигляді таблиць, якщо конкретний рядок першої таблиці може бути пов'язаний з нулем, одним або декількома рядками другої таблиці, але рядок другої таблиці пов'язаний з *єдиним* рядком першої таблиці, то між ними встановлене відношення «один до багатьох».

Прикладом відношення «один до багатьох» може служити зв'язок між службовцями та відділами. Кожен службовець може бути прикріплений у кожний момент часу тільки до одного відділу. Водночас у кожному відділі може працювати *багато* людей. У цьому прикладі відомості щодо відділів становлять батьківську таблицю, з якою пов'язана таблиця службовців. Кожному рядку в таблиці відділів може відповідати *одна* або *декілька* рядків з таблиці службовців, але на кожен рядок у таблиці службовців може посилатися тільки *один* рядок з таблиці відділів.

Ще одним прикладом може служити зв'язок сутності «квартира» і сутності «мешканець». У який-небудь момент часу яка-небудь квартира може пустувати, в іншій квартирі може жити один-єдиний мешканець, у третій їх може бути декілька.

Багато до багатьох (M:N). Зв'язок типу «багато до багатьох» означає, що кожен екземпляр першої сутності може бути пов'язаний з нулем, однієї або декількома екземплярами другої й кожен екземпляр другої сутності може бути пов'язаний з нулем, однієї або декількома екземплярами першої. Або мовою конкретної реалізації сутностей у вигляді таблиць: якщо кожному рядку в першій таблиці відповідає нуль, одна або декілька рядків у другій таблиці, і навпаки, то між ними встановлене відношення «багато до багатьох».

Тип зв'язку «багато до багатьох» є тимчасовим типом, припустимим на ранніх етапах розробки моделі. Надалі цей тип зв'язку повинен замінюватися двома зв'язками типу «один до багатьох» шляхом створення проміжної сутності (таблиці).

Як приклад типу відношення «багато до багатьох» можна розглянути зв'язок службовця й проекту. Кожен службовець може одночасно працювати над *декількома* проектами, і водночас у кожному проекті можуть брати участь *кілька* службовців.

Кожен зв'язок може мати одну із двох *модальностей* – «Може» і «Повинен». Модальність «може» означає, що екземпляр однієї сутності може бути пов'язаний з

одним чи декількома екземплярами іншої, або не пов'язаний з жодним. Модальність «повинен» означає, що екземпляр однієї сутності має бути пов'язаний не менш ніж з одним екземпляром іншої сутності. Та сам зв'язок може мати різну модальність із різних кінців.

1.6. Мова структурованих запитів SQL

Термін SQL означає «Мова Структурованих Запитів». SQL – це мова, спеціально орієнтована на реляційні бази даних. Це декларативна мова програмування для взаємодії користувача з базами даних, що застосовується для формування запитів, оновлення і керування реляційними БД, створення схеми бази даних та її модифікації, системи контролю за доступом до бази даних.

Самостійно SQL не може бути системою керування базами даних, чи окремим програмним продуктом. На відміну від дійсних мов програмування, SQL може формувати інтерактивні запити або, якщо вона вбудована в прикладні програми, виступати як інструкція для керування даними.

Команди в SQL можуть працювати з усіма групами таблиць як з єдиним об'єктом і обробляти будь-яку кількість інформації, витягнутої або отриманої з них, у вигляді єдиного модуля.

Стандарт SQL визначається за допомогою коду ANSI (Американський Національний Інститут Стандартів), фактично SQL винахід IBM. Але інші компанії підхопили SQL одразу ж, принаймні, одна компанія (Oracle), забрала в IBM право на ринковий продаж SQL продуктів.

Після того, як на ринку з'явився ряд конкуруючих програм SQL, ANSI визначив стандарт, до якого вони мають бути доведені.

Усередині продуктів SQL є деякі неузгодженості. SQL з'явився з комерційного світу баз даних як інструмент, і був пізніше перетворений на стандарт ANSI. На жаль, ANSI не завжди визначає найбільшу користь, тому програмісти намагаються відповідати стандарту ANSI, не дозволяючи йому обмежувати їх занадто сильно. ANSI – вид мінімального стандарту – ви можете робити більше, ніж він це дозволяє, але ви повинні бути здатні одержати ті ж результати, що й при виконанні завдання на основі стандарту.

Виділяють декілька напрямків мови SQL:

DDL (Data Definition Language – Мова Визначення Даних) – так звана Мова Опису Схеми в ANSI, складається з команд, які створюють об'єкти (таблиці, індекси, представлення, і так далі) у базі даних.

DML (Data Manipulation Language – Мова Маніпулювання Даними) – це набір команд, що визначають, які значення представлені в таблицях у будь-який момент часу.

DCL (Data Control Language – Мова Керування Даними) складається із засобів, які визначають, чи дозволити користувачеві виконувати певні дії чи ні.

Запити – імовірно, найбільше часто використовуваний аспект SQL. Запит – команда, що ви даєте вашій програмі бази даних, і яка повідомляє її, щоб вона вивела певну інформацію з таблиць у пам'ять.

Запити зазвичай розглядають як частину мови DML. Однак, запит не змінює інформацію в таблицях, а просто показує її користувачеві, ми будемо розглядати його як самостійну категорію серед команд DML, які виконують дію, а не просто показують зміст бази даних.

Усі запити в SQL складаються з одиночної команди. Структура цієї команди проста, але ви маєте розширювати її так, щоб виконати складні оцінки й обробки даних. Ця команда називається – SELECT.

1.7. SQLite. Загальні відомості

SQLite – це програмна бібліотека, що реалізує самодостатню, не потребуючу сервера, транзакційну систему управління реляційними базами даних SQL. SQLite – це одна з найбільш швидко зростаючих СКБД. Вихідний код SQLite є у вільному доступі. Це база даних, яка не потребує конфігурування, та налаштовування під вашу операційну систему. SQLite-рушій не є окремим процесом. Як і з іншими СКБД, ви можете поєднати його статично або динамічно відповідно до вимог з вашим додатком. SQLite отримує прямий доступ безпосередньо до файлів даних.

До переваг СКБД SQLite можна віднести наступні:

– SQLite не вимагає окремого серверного процесу або систему для роботи (serverless);

– SQLite поставляється з нульової конфігурації, яка означає, що не потрібні установки або адміністрування;

– Уся база даних SQLite зберігається в одному кросплатформному файлі;

– СКБД SQLite за обсягом пам’яті дуже мала, менше, ніж 400KiB у повній комплектації або менше, ніж 250KiB при виключенні деяких функції;

– SQLite є самодостатньою, це означає, що немає зовнішніх залежностей;

– SQLite-транзакції повністю задовольняють вимогам ACID, дозволяючи безпечний доступ з декількох процесів або потоків;

– SQLite підтримує більшість функцій мови запитів стандарту SQL92 (SQL2);

– SQLite написано в ANSI-C та забезпечує простий і легкий у використанні API;

– SQLite доступна на UNIX (Linux, Mac OS X, Android, iOS) і Windows (Win32, WinCE, WinRT).

Наведемо основні кроки, які необхідно виконати для інсталяції та запуску SQLite:

Крок 1. Перейти на сторінку завантаження SQLite http://www.sqlite.org/download.html в розділ Windows.

Крок 2. Завантажити sqlite-shell-win32*.zip і sqlite-dll-win32-*.zip архіви.

Крок 3. Створити папку C:\>sqlite і розпакувати завантажені архіви у цю папку (мають бути sqlite3.def, sqlite3.dll та sqlite3.exe файли).

Крок 4. Додавати C:\>sqlite у змінній оточення PATH.

Крок 5. Перейти до командного рядка і виконати команду sqlite3.

Під час виконання роботи у комп’ютерному класі:

– Завантажити з інтернету (або скопіювати в локальній мережі – уточнити у викладача де саме) файли sqlite3.def, sqlite3.dll та sqlite3.exe у папку c:\temp\sqlite (або у будь-яку іншу, де є права на запис даних).

– Запустити командний рядок Windows->run (win+r) ->cmd (рис. 1.7.1).

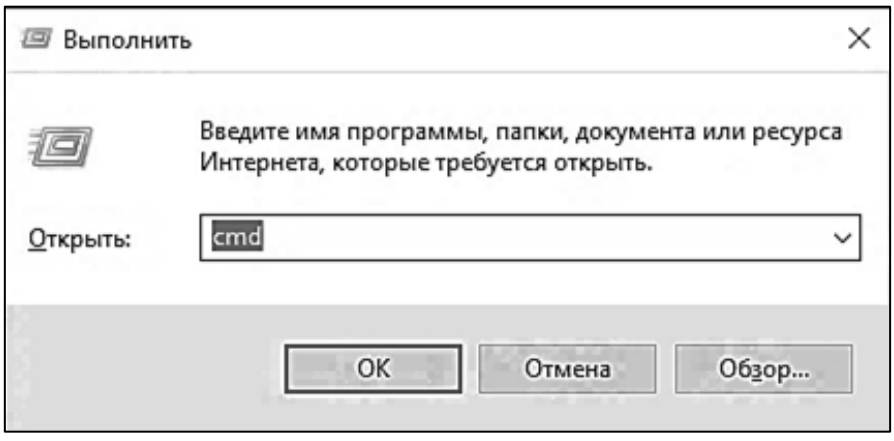

Рис. 1.7.1. Запуск командного рядку

– Перейти до директорії c:\temp\sqlite (рис. 1.7.2).

```
Microsoft Windows [Version 10.0.15063]
(c) Корпорация Майкрософт (Microsoft Corporation), 2017. Все права защищены.

C:\Users\michael>cd c:\temp\sqlite
```

Рис. 1.7.2. Перехід до потрібної директорії у командному рядку

– Запустити sqlite – виконати команду sqlite3 (рис. 1.7.3).

```
c:\temp\sqlite>sqlite3
SQLite version 3.21.0 2017-10-24 18:55:49
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite>
```

Рис. 1.7.3. Запуск sqlite

– Введемо команду .help для отримання відомостей щодо переліку управляючих команд та .exit для завершення першого сеансу роботи із sqlite.

Тип даних SQLite є атрибутом, який визначає тип даних будь-якого об'єкта. Кожен атрибут, змінна і вираз має відповідний тип даних у SQLite. Ви маєте використовувати ці типи даних під час створення таблиці. Але при цьому слід розуміти, що SQLite використовує більш загальну динамічну систему типів. У SQLite тип даних значення асоціюється з самим значенням, а не з її контейнером.

Класи для збереження даних в SQLite. Кожне значення, що зберігається у БД SQLite має один із наступних класів.

– NULL – це значення NULL.

– INTEGER – значення є цілим, зберігається в 1, 2, 3, 4, 6 або 8 байтів залежно від величини значення.

– REAL – значення числа з плаваючою точкою, збережені як IEEE 8-байтове число з рухомою точкою.

– TEXT – це текстовий рядок, збережений із використанням кодової сторінки БД (UTF8, UTF-16BE або UTF-16LE).

– BLOB значення blob, зберігаються у такому вигляді, як були введені.

SQLite клас зберігання даних є трохи більш загальним, ніж тип даних. Клас зберігання, ціле число, наприклад, включає 6 різних цілих типів даних різної довжини.

SQLite підтримує концепцію спорідненості типів для колонок. Будь-яка колонка може зберігати будь який тип даних, але клас, якому надається перевага, має назву спорідненого. Кожна колонка таблиці в SQLite3 БД віднесена до одного із наступних споріднених типів: TEXT, NUMERIC, INTEGER, REAL, NONE.

SQLite не має окремого бульового класу зберігання. Замість цього, логічні значення зберігаються як цілі числа 0 (неправильно) і 1 (правильно).

2. РОБОТА ЗІ СКБД SQLITE

2.1. Створення БД. Створення таблиці. Імпорт-експорт, резервне копіювання та відновлення

Створення бази даних. Для створення БД за допомогою SQLite непотрібно привілеїв або додаткових команд. Усе, що потрібно зробити – під час запуску sqlite3, вказати назву БД. Якщо вона існує, то робота буде продовжена в контексті вказаної БД, в іншому випадку вона буде створена автоматично. У командному рядку виконаємо команду sqlite3 myFirstDB.db (рис. 2.1.1).

```
c:\temp\sqlite>sqlite3 myFirstDB.db
```

Рис. 2.1.1. Створення нової БД

Команда створить файл myFirstDB.db в поточному каталозі. Цей файл буде використовуватися як поточна база даних SQLite. Якщо ви помітили, після успішного створення бази даних, sqlite3 виводить запрошення sqlite > для вводу наступної команди (рис. 2.1.2).

```
c:\temp\sqlite>sqlite3 myFirstDB.db
SQLite version 3.21.0 2017-10-24 18:55:49
Enter ".help" for usage hints.
sqlite> .databases
main: c:\temp\sqlite\myFirstDB.db
sqlite>
```

Рис. 2.1.2. Запрошення sqlite3 для продовження роботи

Після створення бази даних, ви можете перевірити результат у списку баз даних, використовуючи команду .databases (рис. 2.1.3).

```
sqlite> .databases
main: c:\temp\sqlite\myFirstDB.db
```

Рис. 2.1.3. Перевірка списку БД. Команда .databases

Створимо таблицю students, та додамо до неї декілька рядків (рис. 2.1.4).

```
sqlite> create table students(id int, name text, age int);
sqlite> insert into students values(1, 'Ivanov', 20);
sqlite> insert into students values(2, 'Petrov', 19);
sqlite> insert into students values(3, 'Sidorov', 21);
```

Рис. 2.1.4. Створення таблиці «студенти» та заповнення її даними

Перевіримо вміст таблиці, попередньо виконавши налаштування для відображення імен колонок виводу (рис. 2.1.5–2.1.6).

```
sqlite> .mode column
sqlite> .headers on
```

Рис. 2.1.5. Налаштування виведення шапки таблиці

```
sqlite> select * from students;
id          name        age
----------  ----------  ----------
1           Ivanov      20
2           Petrov      19
3           Sidorov     21
```

Рис. 2.1.6. Виведення вмісту таблиці на екран

Для відображення схеми поточної БД можемо виконати команду .schema (рис. 2.1.7).

```
sqlite> .schema
CREATE TABLE students(id int, name text, age int);
```

Рис. 2.1.7. Виведення схеми (структури) таблиці

Також уявлення про перелік об'єктів БД можна отримати звернувшись до системної таблиці sqlite_master (рис. 2.1.8).

```
sqlite> select * from sqlite_master;
type        name        tbl_name    rootpage    sql
----------  ----------  ----------  ----------  ------------------------------------------------
table       students    students    2           CREATE TABLE students(id int, name text, age int)
```

Рис. 2.1.8. Звернення до системної таблиці sqlite_master

За допомогою команди .dump можна виконати резервну копію БД у вигляді скрипту на створення об'єктів БД та заповнення їх даними (рис. 2.1.9).

```
sqlite> .output myDump.sql
sqlite> .dump
sqlite> .output stdout
```

Рис. 2.1.9. Створення резервної копії БД

На рис. 2.1.10 наведемо вміст створеного файлу myDump.sql.

myDump.sql — Блокнот

Файл Правка Формат Вид Справка

```
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE students(id int, name text, age int);
INSERT INTO students VALUES(1,'Ivanov',20);
INSERT INTO students VALUES(2,'Petrov',19);
INSERT INTO students VALUES(3,'Sidorov',21);
COMMIT;
```

Рис. 2.1.10. Вміст файлу резервної копії БД

Дамп БД також може бути створений з командного рядка за допомогою sqlite3 (рис. 2.1.11).

```
c:\temp\sqlite>sqlite3 myFirstDB.db .dump > myDump.sql
```

Рис. 2.1.11. Створення резервної копії БД із командного рядку ОС

Маючи резервну копію, можемо у разі необхідності відновити БД. Видалимо файл myFirstDB.db та виконаємо команду відновлення БД із резервної копії (рис. 2.1.12).

```
c:\temp\sqlite>sqlite3 myFirstDB.db < myDump.sql
```

Рис. 2.1.12. Відновлення БД із резервної копії

Далі завантажимо БД myFirstDB та пересвідчимося у наявності таблиці students та даних у ній (рис. 2.1.13–2.1.15).

```
c:\temp\sqlite>sqlite3 myFirstDB.db
SQLite version 3.21.0 2017-10-24 18:55:49
Enter ".help" for usage hints.
```

Рис. 2.1.13. Підключення до БД

```
sqlite> .tables
students
```

Рис. 2.1.14. Виведення списку таблиць

```
sqlite> select * from students;
id          name        age
----------  ----------  ----------
1           Ivanov      20
2           Petrov      19
3           Sidorov     21
```

Рис. 2.1.15. Вибірка даних із таблиці «студенти»

Виконаємо експорт даних таблиці students до зовнішнього файлу stud.csv (рис. 2.1.16).

```
sqlite> .headers off
sqlite> .mode list
sqlite> .output stud.csv
sqlite> select * from students;
sqlite> .output stdout
```

Рис. 2.1.16. Експорт даних до зовнішнього csv файлу

Видалимо дані з таблиці «students» та пересвідчимося, що таблиця порожня (рис. 2.1.17).

```
sqlite> delete from students;
sqlite> select * from students;
sqlite>
```

Рис. 2.1.17. Видалення даних із таблиці «студенти»

Тепер завантажимо дані з файлу до таблиці «students» та перевіримо її вміст (рис. 2.1.18–2.1.19).

```
sqlite> .import stud.csv students
```

Рис. 2.1.18. Завантаження даних із csv файлу

```
sqlite> select * from students;
1|Ivanov|20
2|Petrov|19
3|Sidorov|21
sqlite>
```

Рис. 2.1.19. Перевірка вмісту таблиці

2.2. Зміна таблиці. Обмеження цілісності первинний, зовнішній ключі та перевірка. Індекси

Запустимо sqlite3 із вказанням поточної БД studentsList.db та створимо таблицю «students» із полями id та name (рис. 2.2.1). (у разі необхідності попередньо виконати копіювання необхідних для роботи sqlite файлів, що описані у попередньому розділі).

```
c:\temp\sqlite>sqlite3 studentsList.db
SQLite version 3.21.0 2017-10-24 18:55:49
Enter ".help" for usage hints.
sqlite>
```

Рис. 2.2.1. Підключення (створення) до БД studentsList.db

```
sqlite> create table students(id int, name text);
sqlite> .schema students
CREATE TABLE students(id int, name text);
sqlite>
```

Рис. 2.2.2. Створення таблиці студентів

Тепер змінимо ім’я таблиці, та виконаємо команду alter table (рис. 2.2.3).

```
sqlite> .tables
students
sqlite> alter table students rename to stud;
sqlite> .tables
stud
```

Рис. 2.2.3. Зміна назви таблиці

Змінимо структуру таблиці, та додамо до неї поле age (рис. 2.2.4). Також перед та після додавання виконаємо команду відображення схеми таблиці.

```
sqlite> .schema stud
CREATE TABLE IF NOT EXISTS "stud"(id int, name text);
sqlite> alter table stud add column age int;
sqlite> .schema stud
CREATE TABLE IF NOT EXISTS "stud"(id int, name text, age int);
```

Рис. 2.2.4. Додавання поля до таблиці

УВАГА! За допомогою команди ALTER TABLE може бути змінена назва таблиці а також додані додаткові стовпці. Ніякі інші операції зміни командою ALTER TABLE в SQLite на жаль не підтримується.

Видалимо створену таблицю, та створимо її знову із вказаним обмеженням цілісності первинного ключа по полю id (рис. 2.2.5).

```
sqlite> create table students(id INTEGER PRIMARY KEY, name TEXT, age INTEGER);
sqlite> .schema students
CREATE TABLE students(id INTEGER PRIMARY KEY, name TEXT, age INTEGER);
sqlite>
```

Рис. 2.2.5. Додавання обмеження цілісності «первинний ключ»

Спробуємо додати до таблиці 2 рядки з однаковим значенням ключа (рис. 2.2.6).

```
sqlite> insert into students(id, name, age) values(1, 'Vasya', 20);
sqlite> insert into students(id, name, age) values(1, 'Petya', 21);
Error: UNIQUE constraint failed: students.id
sqlite>
```

Рис. 2.2.6. Спроба дублювання значення первинного ключа

Створимо таблицю знову із заданням авто-інкрементного значення первинного ключа (рис. 2.2.7).

```
sqlite> drop table students;
sqlite> create table students(id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT, age INTEGER);
```

Рис. 2.2.7. Видалення та створення таблиці «студентів»
із автоінкрементним числовим первинним (сурогатним) ключем

Тепер додаємо до неї 2-х студентів та переглянемо результат (рис. 2.2.8–2.2.9).

```
sqlite> insert into students(name, age) values('Petya', 21);
sqlite> insert into students(name, age) values('Vasya', 20);
```

Рис. 2.2.8. Додавання даних до таблиці «студенти»

```
sqlite> .headers on
sqlite> .mode column
sqlite> select * from students;
id          name        age
----------  ----------  ----------
1           Petya       21
2           Vasya       20
sqlite>
```

Рис. 2.2.9. Перегляд даних таблиці «студенти»

Створимо у нашій БД ще одну таблицю – групи із полями id, номер групи та староста групи (рис. 2.2.10).

```
sqlite> create table groups(id integer primary key autoincrement, number text, starosta text);
```

Рис. 2.2.10. Створення таблиці «студентські групи»

Тепер переглядаючи список маємо вже 2 таблиці (рис. 2.2.11).

```
sqlite> .tables
groups    students
```

Рис. 2.2.11. Список таблиць БД

Переглянемо структуру створеної таблиці за допомогою команди shema (рис. 2.2.12).

```
sqlite> .schema groups
CREATE TABLE groups(id integer primary key autoincrement, number text, starosta text);
```

Рис. 2.2.12. Перегляд структури таблиці «студентські групи»

Додаємо до створеної таблиці декілька рядків даних (рис. 2.2.13).

```
sqlite> insert into groups(number, starosta) values(201, 'Ivanov');
sqlite> insert into groups(number, starosta) values(202, 'Petrova');
```

Рис. 2.2.13. Додавання даних до таблиці «студентські групи»

Перевіримо результат, виконавши вибірку даних (рис. 2.2.14).

```
sqlite> select * from groups;
id          number      starosta
----------  ----------  ----------
1           201         Ivanov
2           202         Petrova
sqlite>
```

Рис. 2.2.14. Перегляд вмісту таблиці «студентські групи»

Бувають ситуації, коли первинний ключ складається з декількох полів. Наприклад сутність «паспорт» може мати складений первинних ключ із окремих полів «серія» та «номер». У такому випадку синтаксис створення таблиці матиме вигляд, наведений на рис. 2.2.15.

```
CREATE TABLE passport(seria TEXT, number TEXT, who_gave TEXT, PRIMARY KEY(seria, numb
er));
```

Рис. 2.2.15. Приклад складеного первинного ключа

Перевіривши роботу, додаємо 3 рядки, 2 без порушення унікальності, 1 із порушенням (рис. 2.2.16).

```
sqlite> insert into passport values('EO', '123432', 'Mykolaiv');
sqlite> insert into passport values('EK', '123432', 'Odesa');
sqlite> insert into passport values('EK', '123432', 'Kiev');
Error: UNIQUE constraint failed: passport.seria, passport.number
sqlite>
```

Рис. 2.2.16. Додавання даних *без* та *із* порушенням унікальності

Отже реалізуємо зв'язок між таблицями «студенти» та «студентські групи» додавши до таблиці students поле group_id із обмеженням цілісності зовнішній ключ (рис. 2.2.17) та перевіримо, як змінилася схема даних.

```
sqlite> alter table students add column group_id references groups(id);
sqlite> .schema students
CREATE TABLE students(id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT, age INTEGER, group_
id references groups(id));
sqlite>
```

Рис. 2.2.17. Додавання обмеження цілісності «зовнішній ключ»

Виконаємо зміну поля group_id для students та переглянемо результат (рис. 2.2.18–2.2.20).

```
sqlite> update students set group_id = 3;
```

Рис. 2.2.18. Зміна поля «код групи» таблиці «студенти»

```
sqlite> select * from students;
id          name        age         group_id
----------  ----------  ----------  ----------
1           Petya       21          3
2           Vasya       20          3
```

Рис. 2.2.19. Перевірка вмісту таблиці «студенти» після виконання змін

```
sqlite> select * from groups;
id          number      starosta
----------  ----------  ----------
1           201         Ivanov
2           202         Petrova
```

Рис. 2.2.20. Перевірка вмісту таблиці «групи»

Як можна переконатися з результатів запитів до таблиць students та groups, маємо порушення посилальної цілісності, оскільки групи із кодом 3 не існує. Це стало можливим тому що за замовчуванням sqlite налаштований таким чином, що не виконує перевірку на порушення області значень зовнішнього ключа. У цьому можна переконатись, виконавши наступну команду (рис. 2.2.21).

```
sqlite> pragma foreign_keys;
foreign_keys
------------
0
```

Рис. 2.2.21. Налаштування щодо контролю посилальної цілісності

Змінимо це (рис. 2.2.22).

```
sqlite> pragma foreign_keys = on;
sqlite> pragma foreign_keys;
foreign_keys
------------
1
```

Рис. 2.2.22. Включення контролю посилальної цілісності

При повторному виконанні запиту на зміну номера групи матимемо помилку (рис. 2.2.23).

```
sqlite> update students set group_id = 3;
Error: FOREIGN KEY constraint failed
```

Рис. 2.2.23. Порушення посилальної цілісності для зовнішнього ключа

Змінимо номери груп для усунення порушення посилальної цілісності (рис. 2.2.24) та переглянемо результат (рис. 2.2.25).

```
sqlite> update students set group_id = 1 where id = 1;
sqlite> update students set group_id = 2 where id = 2;
```

Рис. 2.2.24. Усунення порушення посилальної цілісності для зовнішнього ключа

```
sqlite> select * from students;
id          name        age         group_id
----------  ----------  ----------  ----------
1           Petya       21          1
2           Vasya       20          2
```

Рис. 2.2.25. Перегляд результату після виконання змін

Обмеження цілісності «зовнішній ключ» бажано визначити на етапі створення таблиці. Розберемо на прикладі, як це робиться. Для цього видалимо таблицю students і створимо її знову, одразу із зовнішнім ключем (рис. 2.2.26–2.2.28).

```
sqlite> drop table students;
```

Рис. 2.2.26. Видалення таблиці «студенти»

```
sqlite> CREATE TABLE students(
   ...> id INTEGER PRIMARY KEY AUTOINCREMENT,
   ...> name TEXT,
   ...> group_id INTEGER,
   ...> FOREIGN KEY(group_id) REFERENCES groups(id)
   ...> );
```

Рис. 2.2.27. Створення таблиці «студенти» із зовнішнім ключем

```
sqlite> .schema students
CREATE TABLE students(
id INTEGER PRIMARY KEY AUTOINCREMENT,
name TEXT,
group_id INTEGER,
FOREIGN KEY(group_id) REFERENCES groups(id)
);
```

Рис. 2.2.28. Перегляд структури таблиці із використанням команди schema

Виконуючи повторне створення таблиці «students», не було додане поле age. Додамо його із перевіркою на допустимість значення >0 і <150 (рис. 2.2.30).

```
sqlite> alter table students add column age INTEGER CHECK(age>0 and age<150);
```

Рис. 2.2.30. Створення поля із перевіркою на відповідність значення

```
sqlite> .schema students
CREATE TABLE students(
id INTEGER PRIMARY KEY AUTOINCREMENT,
name TEXT,
group_id INTEGER, age INTEGER CHECK(age>0 and age<150),
FOREIGN KEY(group_id) REFERENCES groups(id)
);
```

Рис. 2.2.31. Перевірка змін у структурі таблиці «студенти»

Перевіримо роботу створеного обмеження цілісності «перевірка» (рис. 2.2.32–2.2.33). Як бачимо, рядки з невідповідним значенням поля age не додаються до таблиці.

```
sqlite> insert into students(name, age, group_id)
   ...> values('Vasya', 20, 1);
sqlite> insert into students(name, age, group_id)
   ...> values('Petya', 220, 1);
Error: CHECK constraint failed: students
sqlite> insert into students(name, age, group_id)
   ...> values('Klya', -2, 2);
Error: CHECK constraint failed: students
sqlite>
```

Рис. 2.2.32. Спроба додавання рядків до таблиці «студенти»

```
sqlite> select * from students;
id          name        group_id    age
----------  ----------  ----------  ----------
1           Vasya       1           20
sqlite>
```

Рис. 2.2.33. Перегляд вмісту таблиці «студенти» після додавання рядків

Тепер у таблиці Students створимо 2 індекси для полів age та name – без та з контролем унікальності відповідно (рис. 2.2.34).

```
sqlite> create index stud_age on students(age);
sqlite> create unique index stud_name on students(name);
```

Рис. 2.2.24. Створення унікального та неунікального індексів

Переглянемо перелік індексів таблиці (рис. 2.2.35).

```
sqlite> .indices students
stud_age    stud_name
```

Рис. 2.2.35. Перегляд переліку створених індексів

Переглянути індекси також можна скориставшись системною таблицею sqlite_master (рис. 2.2.26).

```
sqlite> select * from sqlite_master where type='index';
type        name        tbl_name    rootpage    sql
----------  ----------  ----------  ----------  -------------------------------------
index       stud_age    students    5           CREATE INDEX stud_age on students(age)
index       stud_name   students    6           CREATE UNIQUE INDEX stud_name on stude
sqlite>
```

Рис. 2.2.26. Перегляд переліку індексів через системну таблицю sqlite_master

Перевіримо роботу унікального індексу (рис. 2.2.27).

```
sqlite> insert into students(name, age, group_id)
   ...> values('Petya', 19, 2);
sqlite> insert into students(name, age, group_id)
   ...> values('Petya', 20, 1);
Error: UNIQUE constraint failed: students.name
sqlite>
```

Рис. 2.2.27. Перевірка роботи унікального індексу

Дозволимо додавання студентів з однаковим іменем, але в різні групи. Для цього видалимо індекс stud_name (рис. 2.2.28–2.2.29) та створимо його заново, але цього разу він складатиметься з 2-х полів – name та group_id (рис. 2.2.30–2.2.31).

```
sqlite> drop index stud_name;
```

Рис. 2.2.28. Видалення унікального індексу з іменем студента

```
sqlite> select * from sqlite_master where type='index';
type        name        tbl_name    rootpage    sql
----------  ----------  ----------  ----------  -------------------------------------
index       stud_age    students    5           CREATE INDEX stud_age on students(age)
sqlite>
```

Рис. 2.2.29. Перевірка змін у списку об’єктів БД

```
sqlite> create unique index stud_name on students(group_id, name);
```

Рис. 2.2.30. Створення складеного унікального індексу

```
sqlite> select * from sqlite_master where type='index';
type        name        tbl_name    rootpage    sql
----------  ----------  ----------  ----------  -------------------------------------
index       stud_age    students    5           CREATE INDEX stud_age on students(age)
index       stud_name   students    6           CREATE UNIQUE INDEX stud_name on stude
```

Рис. 2.2.31. Повторна перевірка змін у списку об'єктів БД

Протестуємо роботу створеного індексу щодо перевірки на унікальність (рис. 2.2.32–2.2.33).

```
sqlite> select * from students;
id          name        group_id    age
----------  ----------  ----------  ----------
1           Vasya       1           20
2           Petya       2           19
```

Рис. 2.2.32. Вміст таблиці «студенти»

```
sqlite> insert into students(name, age, group_id)
   ...> values('Petya', 20, 2);
Error: UNIQUE constraint failed: students.group_id, students.name
```

Рис. 2.2.33. Спроба додавання рядка із порушенням унікальності індексу

Отже Petya не додається до групи із кодом 2, в якій від уже є. Але він може бути доданий до іншої групи (рис. 2.2.34).

```
sqlite> insert into students(name, age, group_id)
   ...> values('Petya', 20, 1);
sqlite> select * from students;
id          name        group_id    age
----------  ----------  ----------  ----------
1           Vasya       1           20
2           Petya       2           19
3           Petya       1           20
sqlite>
```

Рис. 2.2.34. Додавання студента із таким самим іменем до іншої групи

2.3. SQLite. Виконання запитів на вибірку даних

Для виконання різноманітних вибірок даних будемо використовувати тестову БД, яку завантажимо із дампу foods.sql (рис. 2.3.1).

```
c:\temp\sqlite>sqlite3 food.db < foods.sql
```

Рис. 2.3.1. Відновлення БД із дампу foods.sql

Далі запустимо sqlite3 із вказанням поточної БД food.db та переглянемо перелік таблиць БД (рис. 2.3.2).

```
c:\temp\sqlite>sqlite3 food.db
SQLite version 3.21.0 2017-10-24 18:55:49
Enter ".help" for usage hints.
sqlite> .tables
episodes        food_types      foods           foods_episodes
```

Рис. 2.3.2. Перегляд переліку таблиць БД food

Як бачимо, БД складається з 4-х таблиць. Наведемо даталогічну модель БД та її загальний огляд.

База даних, що використовується, містить назви страв з кожного епізоду серіалу Seinfeld. (If you've ever watched Seinfeld, you can't help but notice a slight preoccupation with food). Глядач цього серіалу не може не відчути занепокоєність їжею ☺. Більш ніж у ста вісімдесяти епізодах згадується більш ніж чотириста різних страв. На рис. 2.3.3 показано схему бази даних.

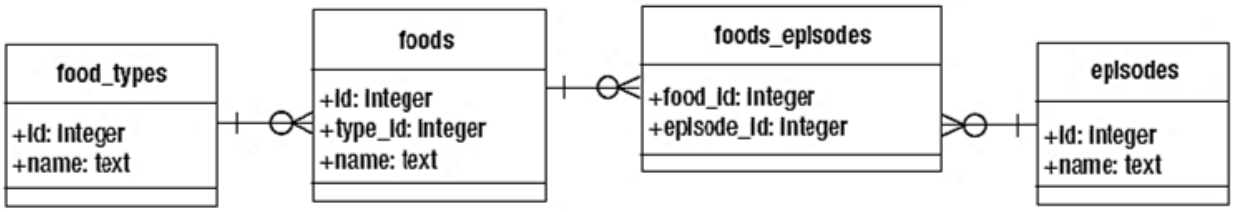

Рис. 2.3.3. Даталогічна модель БД «Food»

Foods є головною таблицею. Кожен запис відповідає окремій страві, її назва реєструється в полі name. Поле Type_id посилається на таблицю food_types, яка містить класифікацію продуктів (тобто, фаст-фуд, напої і так далі). Нарешті, таблиця food_episodes поєднує продукти з великою кількістю епізодів.

Перед розглядом конкретних прикладів щодо вибірки даних із БД, наведемо загальний огляд команди select.

Оператор select включає реляційні операції за допомогою серії фраз (речення – clause). Кожна фраза відповідає своїй реляційній операції. У SQLite майже всі фрази не обов'язкові. Користувач SQLite може використовувати тільки ті, операції яких він потребує. Найбільша загальна форма select в SQLite представлена на рис. 2.3.4.

```
select [distinct] heading
from tables
where predicate
group by columns
having predicate
order by columns
limit count,offset;
```

Рис. 2.3.4. Загальна форма команди вибірки даних

Одна із інтерпретацій оператора select – це уявити його як конвеєр, який обробляє відношення. На конвеєрі є необов'язкові процеси, виконання яких можна пропускати. Незалежно від того, використовуються чи ні конкретні операції (процеси), конвеєр завжди працює однаково. На малюнку нижче можна подивитися порядок виконання. Виконання оператора select починається з фрази from, яка приймає одне або більше відношень і по'єднує їх в одне складне відношення і, потім, передає ланцюжку послідовних операцій (рис. 2.3.5).

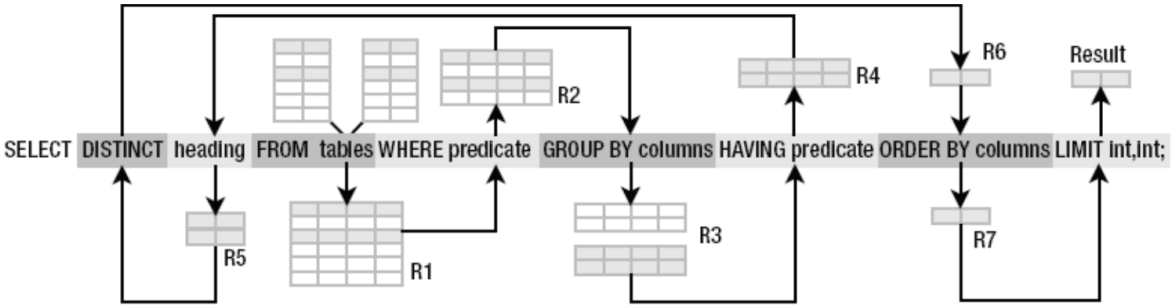

Рис. 2.3.5. Ланцюжок послідовних операцій під час виконання вибірки даних

Фраза from містить список декількох таблиць, представлень і підзапитів (представлених у вигляді змінної tables), розділених комами. Більш ніж одна таблиця (представлення або підзапит) будуть по'єднуватися в одне відношення, яке на тому ж рисунку представляється назвою R1. Різні елементи поєднуються в одне відношення операцією join.

Фраза where відбирає необхідні записи з R1. За ключовим словом where йде предикат (логічний вираз), який визначає критерій відбору записів з R1, які повинні бути включені до наступного відношення. Обрані записи утворюють нове відношення R2.

Як показує рис. 2.3.6, фраза select в SQLite по'єднує усі дані, розглянуті у фразі from, відбирає записи (обмежує) їх у фразі where і відбирає поля (проектує) у фразі select.

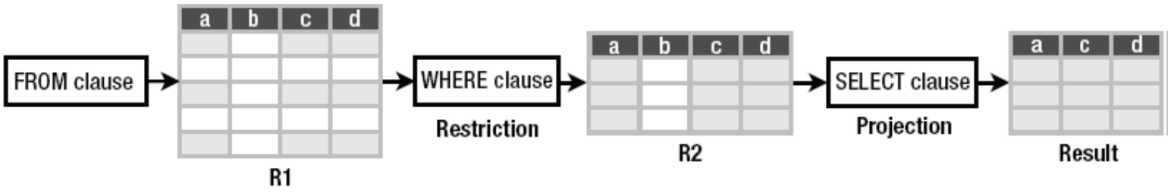

Рис. 2.3.6. Етапи виконання команди select

Далі перейдемо до розгляду простих прикладів. Першим розглянемо просту вибірку зі списку типів страв (рис. 2.3.7).

```
sqlite> select * from food_types;
id          name
----------  ----------
1           Bakery
2           Cereal
3           Chicken/Fo
4           Condiments
5           Dairy
6           Dip
7           Drinks
8           Fruit
9           Junkfood
10          Meat
11          Rice/Pasta
12          Sandwiches
13          Seafood
14          Soup
15          Vegetables
sqlite>
```

Рис. 2.3.7. Вибірка зі списку типів страв

Виконаємо проекцію даних, вибравши лише назви типів страв (рис. 2.3.8).

```
sqlite> select name from food_types;
name
----------
Bakery
Cereal
Chicken/Fo
Condiments
Dairy
Dip
Drinks
Fruit
Junkfood
Meat
```

Рис. 2.3.8. Назви типів страв

Аналогічним чином виберемо назви всіх серій серіалу (рис. 2.3.9).

```
sqlite> select name from episodes;
name
------------------
Good News Bad News
Male Unbonding
The Stake Out
The Robbery
The Stock Tip
The Ex-Girlfriend
The Pony Remark
```

Рис. 2.3.9. Вибірка назв серій серіалу

Кількість рядків у результаті попереднього запиту досить велика, тому за допомогою limit обмежимо його вибравши лише 10 серій із загального списку (рис. 2.3.10).

```
sqlite> select name from episodes LIMIT 5;
name
------------------
Good News Bad News
Male Unbonding
The Stake Out
The Robbery
The Stock Tip
sqlite>
```

Рис. 2.3.10. Обмеження результуючого списку

Аналогічно оберемо перші 5 страв зі списку типів страв (рис. 2.3.11).

```
sqlite> select * from foods limit 5;
id          type_id     name
----------  ----------  ----------
1           1           Bagels
2           1           Bagels, ra
3           1           Bavarian C
4           1           Bear Claws
5           1           Black and
sqlite>
```

Рис. 2.3.11. Перші 5 страв зі списку типів страв

Для зручності виводу можна змінювати ширину стовбців (рис. 2.3.12).

```
sqlite> .width 3 3 30
sqlite> select * from foods limit 5;
id   typ  name
---  ---  ------------------------------
1    1    Bagels
2    1    Bagels, raisin
3    1    Bavarian Cream Pie
4    1    Bear Claws
5    1    Black and White cookies
```

Рис. 2.3.12. Вказання ширини стовбців при виводі даних

Далі перейдемо до фільтрування даних. Виконаємо вибірку зі списку назв серій другого сезону серіалу (рис. 2.3.13).

```
sqlite> select * from episodes where season = 2;
id          season      name
----------  ----------  -----------------
5           2           The Ex-Girlfriend
6           2           The Pony Remark
7           2           The Busboy
8           2           The Baby Shower
9           2           The Jacket
10          2           The Chinese Resta
11          2           The Phone Message
12          2           The Apartment
```

Рис. 2.3.14. Список серій 2-го сезону

Умови можуть поєднуватись із використанням бульових операторів. Оберемо список страв, що відноситься до 2-го та 3-го типу (рис. 2.3.15).

```
sqlite> select * from foods where type_id = 2 or type_id = 3;
id          type_id     name
----------  ----------  ----------
48          2           Bran
49          2           Cheerios
50          2           Corn Flake
51          2           Double Cru
52          2           Fruit Loop
53          2           Grape Nuts
```

Рис. 2.3.15. Список страв, що відноситься до 2-го та 3-го типу

Цю вибірку також можна реалізувати за допомогою оператора in (рис. 2.3.16).

```
sqlite> select * from foods where type_id in (2,3);
id          type_id     name
----------  ----------  ----------
48          2           Bran
49          2           Cheerios
50          2           Corn Flake
51          2           Double Cru
52          2           Fruit Loop
```

Рис. 2.3.16. Використання оператора in

Під час виконання фільтрації за строковим значенням можна вказувати шаблони, яким має відповідати поле. Наприклад, оберемо страви, назва яких пов’язана із морквою (рис. 2.3.17).

```
sqlite> select * from foods where name like '%carrot%';
id          type_id     name
----------  ----------  ----------
3           1           Carrot Cake
363         14          Carrot Soup
387         15          Carrots
sqlite>
```

Рис. 2.3.17. Страви, назва яких пов’язана із морквою

Або, наприклад серії 3-го сезону, в назві яких є слово «бойфренд» (рис. 2.3.18).

```
sqlite> select * from episodes where season = 3 and name like '%boyfriend%';
id   seaso  name
---  -----  --------------------------------------------------
32   3      The Boyfriend 1
33   3      The Boyfriend 2
```

Рис. 2.3.19. Серії 3-го сезону, в назві яких є слово «бойфренд»

Мова SQL дозволяє виконувати групування та розраховувати значення полів із застосуванням функцій агрегації. Так, визначимо загальну кількість серій у серіалі (рис. 2.3.20).

```
sqlite> .width auto
sqlite> select count(*) from episodes;
count(*)
----------
181
```

Рис. 2.3.20. Загальна кількість серій у серіалі

Або, наприклад, кількість серій 2-го сезону (рис. 2.3.21).

```
sqlite> select count(*) from episodes where season = 2;
count(*)
----------
13
sqlite>
```

Рис. 2.3.21. Кількість серій 2-го сезону

Функція агрегації count може бути використана разом із оператором distinct, що усуває дублювання. Наступний запит визначає кількість сезонів у серіалі (рис. 2.3.22).

```
--
sqlite> select count(distinct season) from episodes;
count(distinct season)
----------------------
9
sqlite>
```

Рис. 2.3.22. Кількість сезонів у серіалі

Потім додаємо речення group by для розбиття вибірки на діапазони. Реалізуємо підрахунок кількості серій за сезонами (рис. 2.3.23).

```
sqlite> select season, count(id)
   ...> from episodes
   ...> group by season;
season      count
----------  -----
            2
1           4
2           13
3           22
4           24
5           22
6           24
7           24
8           22
9           24
sqlite>
```

Рис. 2.3.23. Підрахунок кількості серій за сезонами

Результат роботи функцій агрегації може бути відфільтровано із використанням речення having. Так, виберемо сезони, в яких більше 20 серій (рис. 2.3.24).

```
sqlite> select season
   ...> from episodes
   ...> group by season
   ...> having count(id) > 20;
season
----------
3
4
5
6
7
8
9
sqlite>
```

Рис. 2.3.24. Сезони, в яких більше 20 серій

Розглянемо можливості SQL для упорядкування виводу. У першому прикладі виведемо список типів страв, упорядкований за назвою (рис. 2.3.25).

```
sqlite> select name from food_types order by name;
name
----------
Bakery
Cereal
Chicken/Fo
Condiments
Dairy
Dip
Drinks
Fruit
```

Рис. 2.3.25. Список типів страв, упорядкований за назвою

Сортування може бути виконано і у зворотному напрямку. Оберемо список страв 1-го типу, відсортований за назвою у зворотному порядку (рис. 2.3.26).

```
sqlite> select * from foods where type_id = 1 order by name desc;
id          type_id     name
----------  ----------  --------------------
47          1           Wedding Cake (Royal)
46          1           Triscuits
45          1           Poppy Seed Muffins
35          1           Pizza Bagels
37          1           Pie (blueberry)
42          1           Pie (Strawberry)
```

Рис. 2.3.26. Список страв 1-го типу, відсортований за назвою у зворотному порядку

Сортування результуючого набору також працює і за декількома полями. Виберемо список серій, відсортований за сезоном, за зростанням, назвою серії та спаданням (рис. 2.3.27).

```
sqlite> select * from episodes order by season, name desc;
id          season      name
----------  ----------  ----------------------------------
180                     Reunion Special:The Seinfeld Story
9                       Good News Bad News
4           1           The Stock Tip
2           1           The Stake Out
3           1           The Robbery
1           1           Male Unbonding
13          2           The Stranded
14          2           The Statue
16          2           The Revenge
```

Рис. 2.3.27. Список серій, відсортований за сезоном, зростанням, назвою серії та спаданням

Упорядковувати також можна і за результатом роботи функції агрегації. Продемонструємо це, відобразивши список сезонів серіалу, впорядкований за кількістю серій за спаданням.

```
sqlite> select season, count(id)
   ...> from episodes
   ...> group by season
   ...> order by 2 desc;
season      count(id)
----------  ----------
4           24
6           24
7           24
9           24
3           22
5           22
8           22
2           13
1           4
            2
sqlite>
```

Рис. 2.3.28. Список сезонів серіалу, впорядкований за кількістю серій за спаданням

Ми розглянули лише запити до бази даних із використанням однієї таблиці. Але запити на вибірку також можуть використовувати більше однієї таблиці. У розглянутих нижче прикладах для поєднання 2-х таблиць буде використовуватись конструкція join. Так, наприклад, виведемо назву та тип страви, що знаходяться у різних таблицях (рис. 2.3.29).

```
sqlite> select f.name, t.name
   ...> from foods f inner join food_types t
   ...> on f.type_id = t.id
   ...> limit 10;
name        name
----------  ----------
Bagels      Bakery
Bagels, ra  Bakery
Bavarian C  Bakery
Bear Claws  Bakery
Black and   Bakery
Bread (wit  Bakery
Butterfing  Bakery
Carrot Cak  Bakery
Chips Ahoy  Bakery
Chocolate   Bakery
sqlite>
```

Рис. 2.3.29. Список страв із вказанням її типів

У наступному прикладі покажемо реалізацію вибірки салатів із категорії овочі (рис. 2.3.30).

```
sqlite> select f.name
   ...> from foods f inner join food_types t
   ...> on f.type_id = t.id
   ...> where f.name like '%salad%' and t.name = 'Vegetables'
   ...> ;
name
--------------------
Big Salad
Chef Salad
Potato Salad
Salad
sqlite>
```

Рис. 2.3.30. Салати із категорії овочі

Реалізуємо приклад, що використовуватиме як поєднання таблиць за допомогою join, так і групування даних group by. Виберемо перші 5 епізодів за кількістю згадувань про їжу (рис. 2.3.31).

```
sqlite> select e.name, count(f.food_id)
   ...> from episodes e inner join foods_episodes f
   ...> on e.id = f.episode_id
   ...> group by e.name
   ...> order by 2 desc
   ...> limit 5;
name                 count(f.food_id)
-------------------  ----------------
The Soup             23
The Fatigues         14
The Bubble Boy       12
The Finale 1         10
The Dinner Party     9
sqlite>
```

Рис. 2.3.31. Перші 5 епізодів за кількістю згадувань про їжу

Або перші 3 сезони за кількістю згадувань різних страв (рис. 2.3.32).

```
sqlite> select e.season, count(distinct f.food_id)
   ...> from episodes e inner join foods_episodes f
   ...> on e.id = f.episode_id
   ...> group by e.season
   ...> order by 2 desc
   ...> limit 3;
season               count(distinct f.food_id)
-------------------  -------------------------
9                    82
5                    73
6                    69
sqlite>
```

Рис. 2.3.32. Перші 3 сезони за кількістю згадувань різних страв

Ще один цікавий приклад – підрахувати, у скількох серіях кожного із сезонів згадується про морозиво (рис. 2.3.33).

```
sqlite> select e.season, count(distinct e.id)
   ...> from episodes e inner join foods_episodes fe
   ...> on e.id = fe.episode_id
   ...> inner join foods f on fe.food_id = f.id
   ...> where f.name like '%ice Cream%'
   ...> group by e.season;
season                count(distinct e.id)
--------------------  --------------------
2                     2
4                     1
9                     2
sqlite>
```

Рис. 2.3.33. У скількох серіях кожного із сезонів згадується про морозиво

Або, у яких серіях 5-го сезону згадується випічка (рис. 2.3.34).

```
sqlite> select distinct e.*
   ...> from episodes e inner join foods_episodes fe
   ...> on e.id = fe.episode_id
   ...> inner join foods f on fe.food_id = f.id
   ...> inner join food_types t on f.type_id = t.id
   ...> where t.name = 'Bakery' and e.season = 5;
id                    season      name
--------------------  ----------  ----------------------
67                    5           The Sniffing Accountant
82                    5           The Raincoats 2
76                    5           The Dinner Party
66                    5           The Glasses
78                    5           The Pie
75                    5           The Stall
sqlite>
```

Рис. 2.3.34. У яких серіях 5-го сезону згадується випічка

2.4. SQLite. Представлення та тригери

Окрім таблиць та індексів у SQLite є ще 2 типи логічних об’єктів – це представлення та тригери. Почнемо із огляду представлень.

Представлення (view) є віртуальними таблицями. Їх ще називають похідними таблицями, у зв’язку з тим, що їх вміст є результатом виконання запитів до інших таблиць. Насправді, хоча представлення схожі на справжні таблиці, вони ними не є. Вміст справжніх таблиць є справжніми даними, тоді як вміст представлення динамічно генерується під час звернення до нього.

Синтаксис для створення представлення наступний (рис. 2.4.1):

```
create view name as select-stmt;
```

Рис. 2.4.1. Синтаксис команди створення представлення

Назва представлення задається текстом name, а визначається оператором select – stmt. У результаті представлення буде виглядати, як таблиця з назвою name. Уявіть, що існує запит, який потрібно часто виконувати. Представлення – це засіб, який дозволяє уникати постійного набрання такого запиту. Інший приклад – коли запит є частиною деякої кількості складних запитів.

Створимо представлення, що буде відображати сезони, назви епізодів, страв, що в них згадуються та типів страв, до яких вони відносяться (рис. 2.4.2).

```
sqlite> CREATE VIEW episode_food as
   ...> select e.season, e.name ename, f.name fname, t.name tname
   ...> from episodes e inner join foods_episodes fe on e.id = fe.episode_id
   ...> inner join foods f on fe.food_id = f.id
   ...> inner join food_types t on f.type_id = t.id;
sqlite>
```

Рис. 2.4.2. Створення представлення, що буде відображати сезони, назви епізодів, страв, що в них згадуються та типів страв, до яких вони відносяться

Перевіримо результат, обравши перші 10 рядків із створеного представлення (рис. 2.4.3).

```
sqlite> select * from episode_food limit 10;
season      ename       fname       tname
----------  ----------  ----------  ----------
9           The Strike  Bagels      Bakery
9           The Strike  Bagels, ra  Bakery
8           The Muffin  Bagels, ra  Bakery
7           The Soup N  Bavarian C  Bakery
9           The Strong  Bear Claws  Bakery
5           The Sniffi  Bear Claws  Bakery
5           The Rainco  Bear Claws  Bakery
5           The Dinner  Black and   Bakery
6           The Unders  Black and   Bakery
9           The Apolog  Bread (wit  Bakery
sqlite>
```

Рис. 2.4.3. Вибірка даних із використанням представлення

Виконаємо ще декілька запитів із використанням створеного представлення. Виберемо які типи страв зустрічаються у другому сезоні серіалу (рис. 2.4.3).

```
sqlite> select distinct tname
   ...> from episode_food
   ...> where season = 2;
tname
----------
Bakery
Condiments
Dairy
Dip
Drinks
Fruit
Junkfood
Meat
Seafood
Vegetables
sqlite>
```

Рис. 2.4.3. Типи страв, що зустрічаються у другому сезоні серіалу

Або назва серії, у якій згадується найбільше страв (рис. 2.4.4).

```
sqlite> select ename
   ...> from episode_food
   ...> group by ename
   ...> order by count(fname) desc
   ...> limit 1;
ename
----------
The Soup
sqlite>
```

Рис. 2.4.4. Вибірка серії, у якій згадується найбільше страв

SQLite Triggers – функції зворотного виклику бази даних, які виконуються/ викликаються автоматично при виникненні певної події бази даних.

– Тригер SQLite може бути створений для запуску кожного разу, коли спрацьовує команда DELETE, INSERT або UPDATE до конкретної таблиці бази даних, або коли відбувається UPDATE на одному або декількох вказаних стовпцях таблиці.

– Наразі SQLite підтримує тільки тригери FOR EACH ROW, а не FOR EACH STATEMENT. Отже, чітке визначення FOR EACH ROW є необов'язковим.

– Ключові слова BEFORE або AFTER визначають, коли буде виконуватися тригер, відносно вставки, модифікації або видалення відповідного рядка.

– Тригери автоматично видаляються, коли видаляється таблиця, з якою вони пов'язані.

– Таблиця, яку потрібно змінити, має існувати в тій самій базі даних, що і таблиця або представлення, до яких прикріплений тригер, звернення до таблиці має виглядати, як tablename, а не database.tablename.

– Спеціальна функція SQL RAISE() може бути використана в тілі тригерів для ініціювання exception-у.

Загальний синтаксис створення тригера має наступний вигляд (рис. 2.4.5).

```
CREATE  TRIGGER trigger_name [BEFORE|AFTER] event_name
ON table_name
BEGIN
 -- Trigger logic goes here....
END;
```

Рис. 2.4.5. Синтаксис створення тригера

Event_name може приймати значення операцій БД INSERT, DELETE та UPDATE до таблиці table_name. За бажанням можна вказати FOR EACH ROW після назви таблиці. Далі наводиться синтаксис для створення тригера операції оновлення на одному або більше заданих стовпцях таблиці (рис. 2.4.6).

```
CREATE  TRIGGER trigger_name [BEFORE|AFTER] UPDATE OF column_name
ON table_name
BEGIN
 -- Trigger logic goes here....
END;
```

Рис. 2.4.6. Синтаксис створення тригера на одному або більше стовпцях таблиці

Створимо окрему таблицю для фіксації додавання нових серій до серіалу та тригер, що автоматично її заповнюватиме при виконанні команди INSERT у таблицю episodes (рис. 2.4.7–2.4.8).

```
c:\temp\sqlite>sqlite3 foods.db
SQLite version 3.21.0 2017-10-24 18:55:49
Enter ".help" for usage hints.
sqlite> .tables
episode_food    episodes        food_types      foods           foods_episodes
sqlite> create table episodes_ins_log(id integer, added text);
```

Рис. 2.4.7. Створення таблиці-журналу

```
sqlite> create trigger episode_add BEFORE INSERT
   ...> on episodes
   ...> begin
   ...> insert into episodes_ins_log(id, added)
   ...> values(NEW.id, datetime());
   ...> end;
```

Рис. 2.4.8. Створення тригера для ведення таблиці-журналу

Додамо новий рядок до таблиці episodes та перевіримо роботу тригера (рис. 2.4.9–2.4.10).

```
sqlite> insert into episodes(id, season, name)
   ...> select max(id)+1, 20, 'New series about triggers, not food :)'
   ...> from episodes;
```

Рис. 2.4.9. Додавання нової серії

```
sqlite> select * from episodes_ins_log;
id          added
----------  -------------------
181         2018-01-27 08:16:49
sqlite>
```

Рис. 2.4.10. Перегляд таблиці-журналу

Створимо тригер, що забороняє додавання нової серії, якщо в сезоні вже більше 20 серій (рис. 2.4.11).

```
sqlite> create trigger episode_to_season_add BEFORE INSERT
   ...> on episodes
   ...> begin
   ...> SELECT CASE
   ...> WHEN (SELECT COUNT(id) FROM episodes WHERE season=NEW.season) > 20
   ...> THEN RAISE(ABORT, 'Quantity of sesies in one season should be less then 20 !')
   ...> END;
   ...> end;
```

Рис. 2.4.11. Тригер, що забороняє додавання нової серії, якщо в сезоні вже більше 20 серій

Перевіримо поточну кількість серій за сезонами для подальшої перевірки роботи нашого тригера (рис. 2.4.12).

```
sqlite> select season , count(id) from episodes group by season;
season      count(id)
----------  ----------
            2
1           4
2           13
3           22
4           24
5           22
6           24
7           24
8           22
9           24
20          1
sqlite>
```

Рис. 2.4.12. Поточна кількість серій за сезонами

Додамо серію до першого сезону (рис. 2.5.13) та переконаємося, що вона є у таблиці (рис. 2.6.14).

```
sqlite> insert into episodes(id, season, name)
   ...> select max(id)+1, 1, 'One more series about triggers, not food :)'
   ...> from episodes;
```

Рис. 2.6.13. Додавання нової серії до 1-го сезону

```
sqlite> select * from episodes where season = 1 and name like '%triggers%';
id          season      name
----------  ----------  ------------------------------------------
182         1           One more series about triggers, not food :)
sqlite>
```

Рис. 2.6.15. Перевірка результату додавання

Тепер спробуємо додати серію до 3-го сезону, де вже 22 серії (рис. 2.6.16).

```
sqlite> insert into episodes(id, season, name)
   ...> select max(id)+1, 3, 'One more series about triggers, not food :)'
   ...> from episodes;
Error: Quantity of sesies in one season should be less then 20 !
sqlite>
```

Рис. 2.6.16. Спроба додавання серії до 3-го сезону

Виконавши запит на вибірку серії про тригери із 3-го сезону, переконуємося у тому, що рядок не додався (рис. 2.6.17).

```
sqlite> select * from episodes where season = 3 and name like '%triggers%';
sqlite>
```

Рис. 2.6.17. Спроба вибірки серії, що додавалася до 3-го сезону

Переглянути перелік тригерів у БД та у конкретній таблиці можна використовуючи системну таблицю sqlite_master (рис. 2.6.18).

```
sqlite> select * from sqlite_master
   ...> where tbl_name = 'episodes' and type='trigger';
type        name         tbl_name    rootpage    sql

----------  -----------  ----------  ----------  ---------------------------------------------------
-------------------------------------------
trigger     episode_add  episodes    0           CREATE TRIGGER episode_add BEFORE INSERT
on episodes
begin
insert into episodes_ins_log(id, added)
values(NEW.id, datetime());
end
trigger     episode_to_  episodes    0           CREATE TRIGGER episode_to_season_add BEFORE INSERT
on episodes
begin
SELECT CASE
WHEN (SELECT COUNT(id) FROM episodes WHERE season
sqlite>
```

Рис. 2.6.18. Перегляд відомостей щодо тригерів БД
у системній таблиці sqlite_master

За необхідності тригер може бути видалений за допомогою команди drop (рис. 2.6.19).

```
sqlite> drop trigger episode_add;
```

Рис. 2.6.19. Приклад видалення тригера

2.5. Графічний інтерфейс під час роботи з SQLite

Майже будь-яка сучасна СКБД має графічний інтерфейс менеджера баз даних. Менеджери баз даних часто допомагають розробникам і прискорюють процес їх розробки. Окрім командного рядка (sqlite.exe), який правомірно можна назвати менеджером баз даних SQLite, існують й інші менеджери, які можуть працювати з базами даних під управлінням SQLite3. Їх загальний недолік – усі вони від сторонніх розробників. Переваги: під час використання графічних менеджерів ми пишемо менше коду. Далі наведемо загальний огляд трьох менеджерів баз даних SQLite3.

SQLiteStudio – має російський інтерфейс, розповсюджується безкоштовно, його можна завантажити з офіційного сайту SQLiteStudio. Менеджер БД SQLiteStudio є кросплатформним, на сторінці завантажень можна обрати відповідну версію. Цей менеджер БД встановлюється шляхом розпакування архіву в будь-яку папку на комп’ютері.

SQLite Manager – плагін, що представляє собою зручний менеджер баз даних SQLite3. Функціонал менш багатий, ніж у попередньої програми, але його у більшості випадків цілком достатньо. Плагін кросплатформний, працює на будь-якій ОС, де є Firefox. Може бути завантажений на офіційній сторінці.

DBeaver – потужний та безкоштовний засіб для проектувальника БД, підтримує синтаксис багатьох СКБД, у тому числі і SQLite. Перераховувати можливості цього менеджера баз даних немає сенсу, оскільки він вимагає окремого вивчення. Менеджер баз даних DBeaver написаний на Java, а тому він кросплатформний і для його роботи необхідно встановити JRE. Завантажити DBeaver можна з офіційного сайту.

Насправді для бібліотеки SQLite3 існує набагато більше менеджерів: як безкоштовних, так і платних. Але для досвідчених користувачів у 95 випадках зі 100 зручніше та швидше користуватись консоллю.

Розглянемо роботу із SQLiteStudio. Після першого запуску головне вікно виглядає наступним чином (рис. 2.5.1).

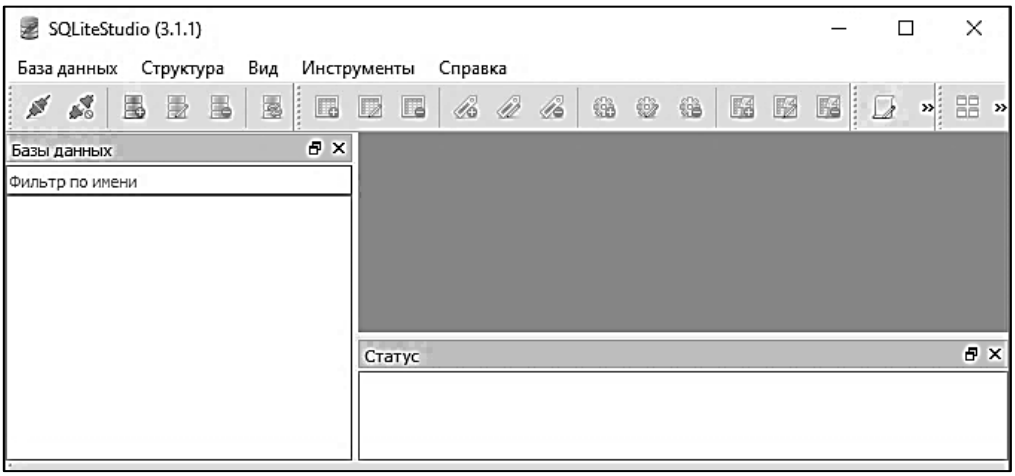

Рис. 2.5.1. Головне вікно SQLiteStudio

Для створення нової БД, або підключення існуючої (тут принцип такий самий, як і в консолі – за відсутності БД вона автоматично створюється) виконуємо «База даних» -> «Додати базу даних» (рис. 2.5.2).

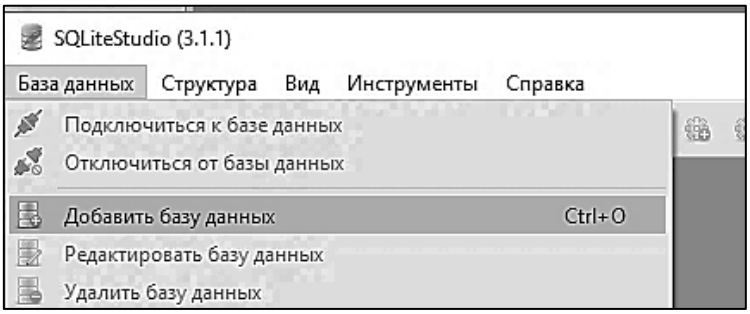

Рис. 2.5.2. Додавання нової БД

Потім обираємо або набираємо шлях до файлу БД, ім’я БД у списку баз даних SQLite Studio та натискаємо «ок» (рис. 2.5.3).

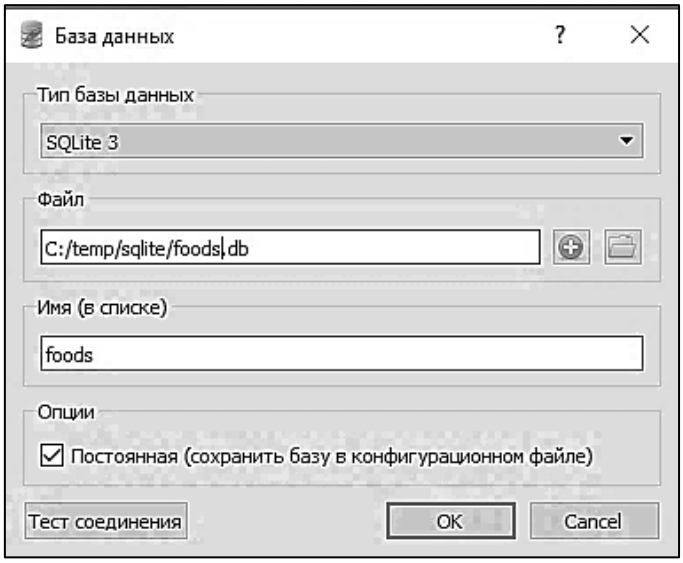

Рис. 2.5.3. Вказання назви БД та шляху до файлу із даними

Підключившись до БД (подвійний клік на БД у списку або кнопка «Підключитись до БД» на панелі інструментів можемо переглянути список таблиць та представлень БД (рис. 2.5.4).

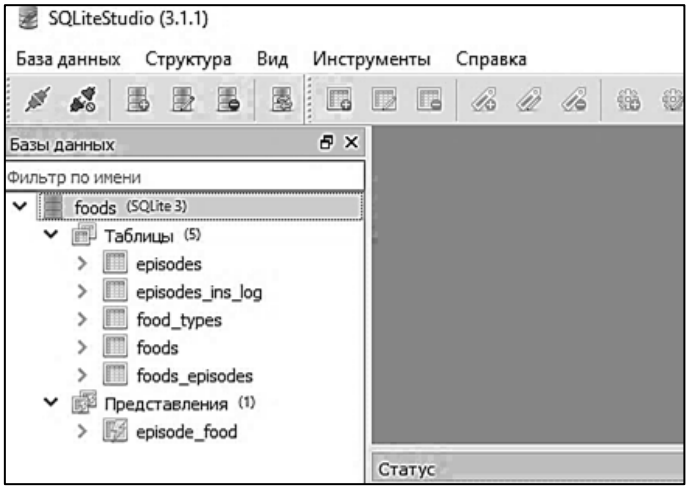

Рис. 2.5.4. Перелік логічних об’єктів БД

Зробимо експорт БД у sql формат. Для цього оберемо «База даних» -> «Експортувати базу даних» (рис. 2.5.5).

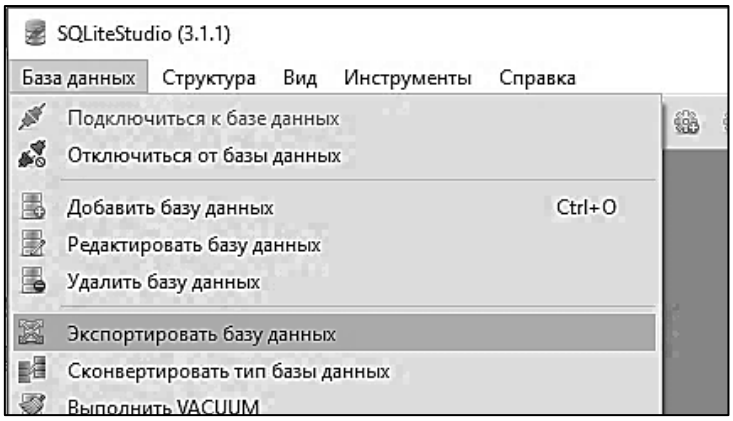

Рис. 2.5.5. Експорт БД

Відмітимо таблиці та представлення, що необхідно експортувати (рис. 2.5.6).

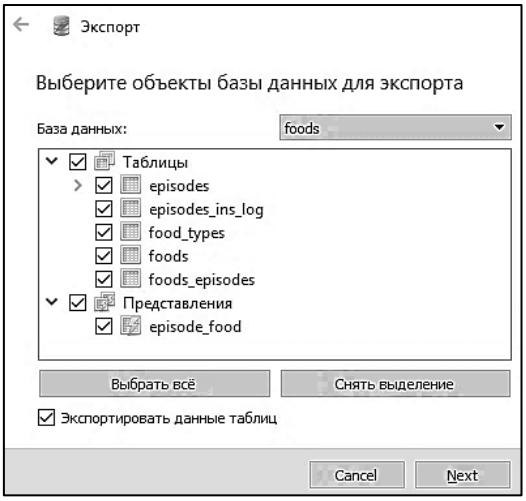

Рис. 2.5.6. Вибір даних для експорту

Обираємо необхідний формат даних та шлях до файлу, у який буде виконано експорт БД (рис. 2.5.7).

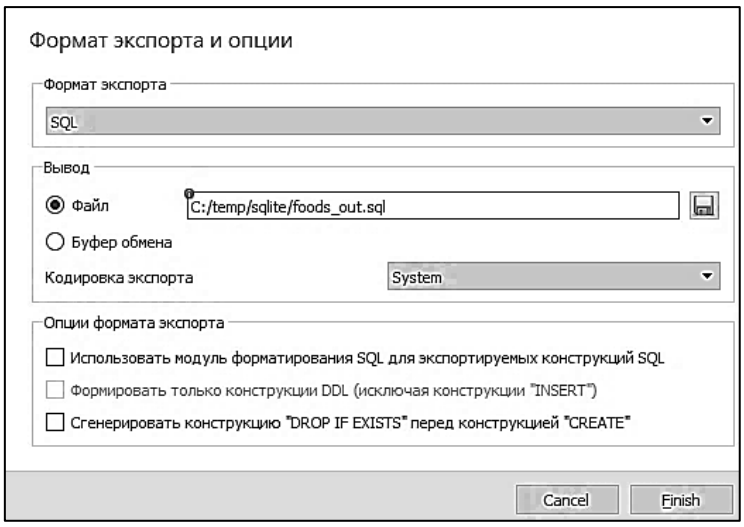

Рис. 2.5.6. Вибір формату даних та файлу

Потім натискаємо finish. Після завершення надійде повідомлення у вікно «статус» (рис. 2.5.7).

Рис. 2.5.7. Статус завершення експорту даних

Натиснувши праву кнопку миші, на таблиці, і обравши у контекстному меню «імпортувати дані у таблицю» або «експортувати таблицю» можна виконати відповідну операцію для окремої таблиці БД.

При деталізації вкладених об’єктів таблиці (або представлення) у дереві метаданих «Бази даних» можемо переглянути окремі стовбці таблиці, їх типи, а також індекси та тригери, що були створені для цієї таблиці (рис. 2.5.8).

Рис. 2.5.8. Перегляд полів та додаткових об’єктів, пов’язаних із таблицею

Користуючись контекстним меню, можна додати новий стовбець, видалити існуючий, чи змінити його назву та тип даних. Те ж саме стосується до індексів та тригерів.

Зверніть увагу, що оскільки обмеження синтаксису команди alter table мови SQLite не дозволяє змінювати або видаляти колонку таблиці, то цьому випадку буде згенеровано послідовність команд. Спочатку дані таблиці будуть збережені в тимчасо-

вій таблиці, потім початкова таблиця буде знищена і створена із новою структурою. Після цього дані будуть перенесені назад. Для запобігання порушення посилальної цілісності перед початком буде відключений режим контролю за зовнішніми ключами, а після завершення знову відновлено.

Користуючись графічним інтерфейсом, створимо нову БД. Від'єднаємося від foods.db (рис. 2.5.9).

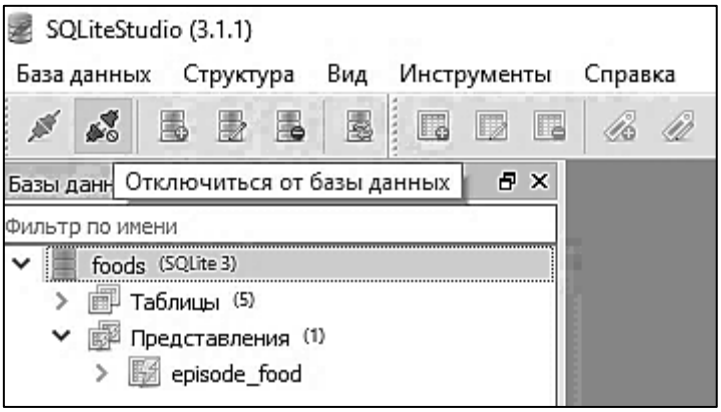

Рис. 2.5.9. Від'єднання від БД

Тепер додамо нову БД students_list.db (рис. 2.5.10–2.5.11).

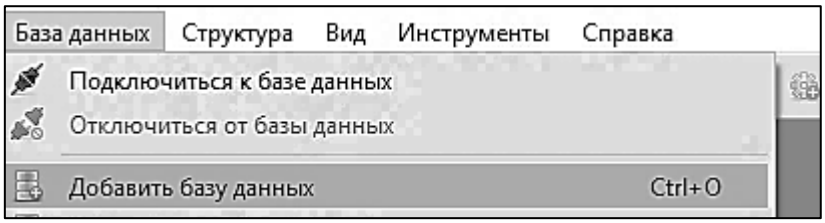

Рис. 2.5.10. Створення нової БД

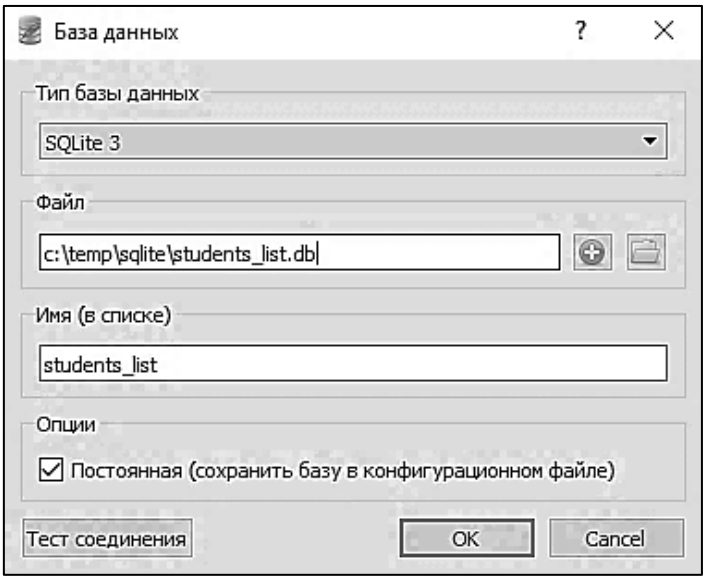

Рис. 2.5.11. Вказання назви та шляху

Далі створимо таблицю групи зі структурою, аналогічною тій, що створювалась у розділі 2.1 із використанням командного рядка (рис. 2.5.12–2.5.16).

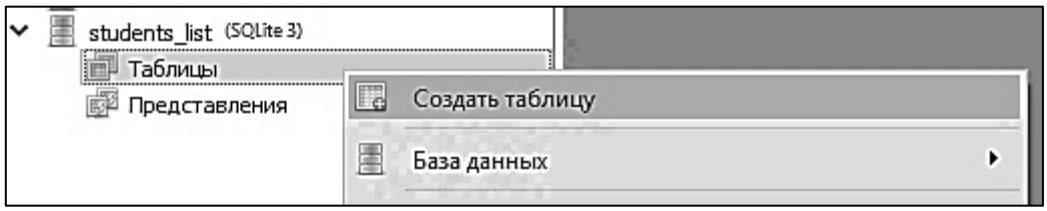

Рис. 2.5.12. Створення нової таблиці

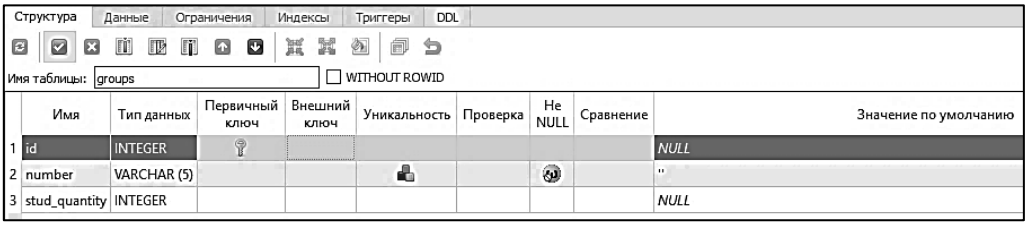

Рис. 2.5.13. Вказання переліку полів із типами даних

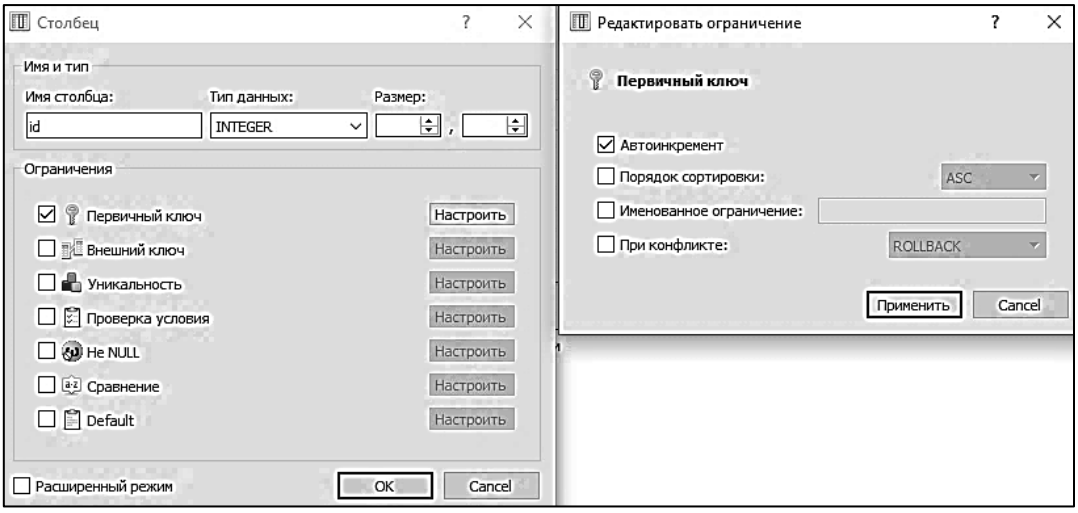

Рис. 2.5.14. Додавання обмеження цілісності: первинний ключ та автоінкремент для поля id

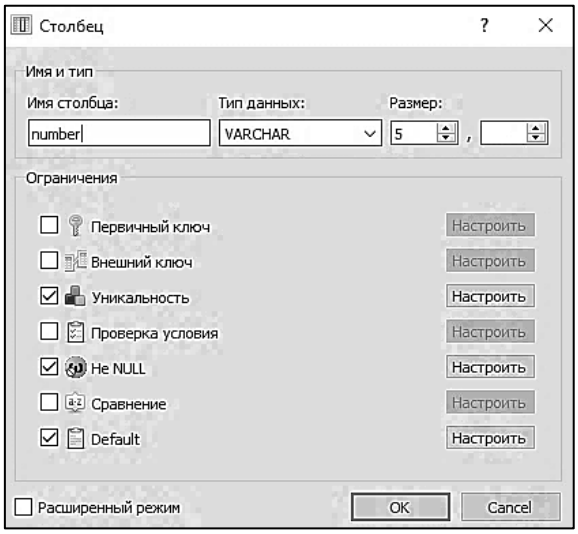

Рис. 2.5.15. Додавання обмеження цілісності унікальність, not null та значення за замовчуванням для поля number

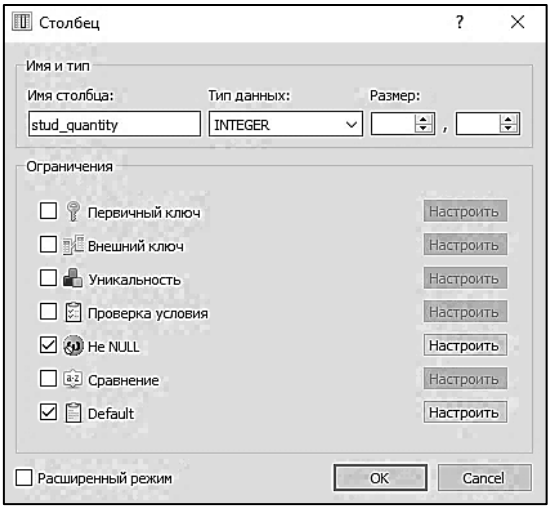

Рис. 2.5.16. Додавання обмеження цілісності not null та значення за замовчуванням для поля stud_quantity

Після вказання необхідних полів та типів даних на панелі інструментів натискаємо «підтвердити зміни структури» (рис. 2.5.17).

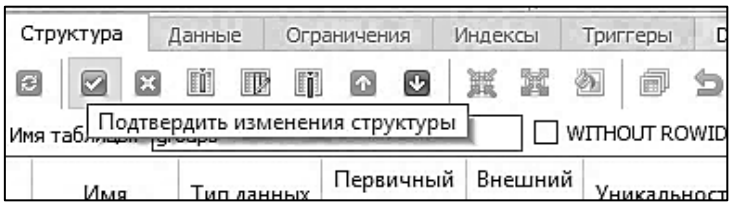

Рис. 2.5.17. Підтвердження зміни структури

Виконуємо спеціально згенерований фрагмент коду (рис. 2.5.18).

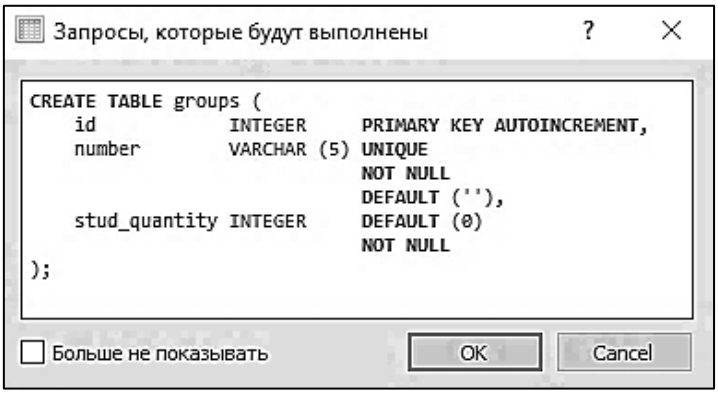

Рис. 2.5.18. Код на створення таблиці, згенерований SQLite Studio

Перейдемо на вкладку «дані» та додамо до таблиці декілька рядків (рис. 2.5.19).

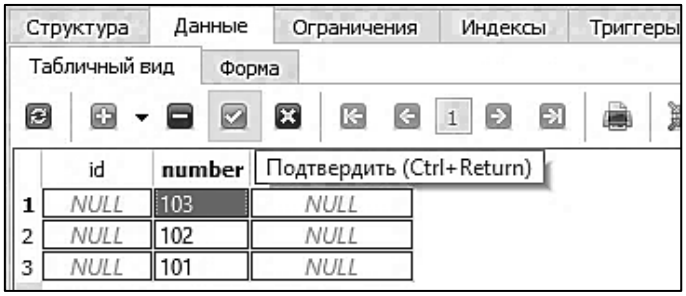

Рис. 2.5.19. Додавання даних до таблиці групи

Після введення даних натиснемо кнопку «підтвердити». Зверніть увагу, що автоінкрементні та дефолтні поля заповнилися автоматично (рис. 2.5.20).

	id	number	stud_quantity
1	1	103	0
2	2	102	0
3	3	101	0

Рис. 2.5.20. Автоматичне заповнення автоінкрементних та дефолтних полів

За аналогією створимо таблицю «студент» із атрибутами код, ім’я, вік, група (рис. 2.5.21–2.5.25).

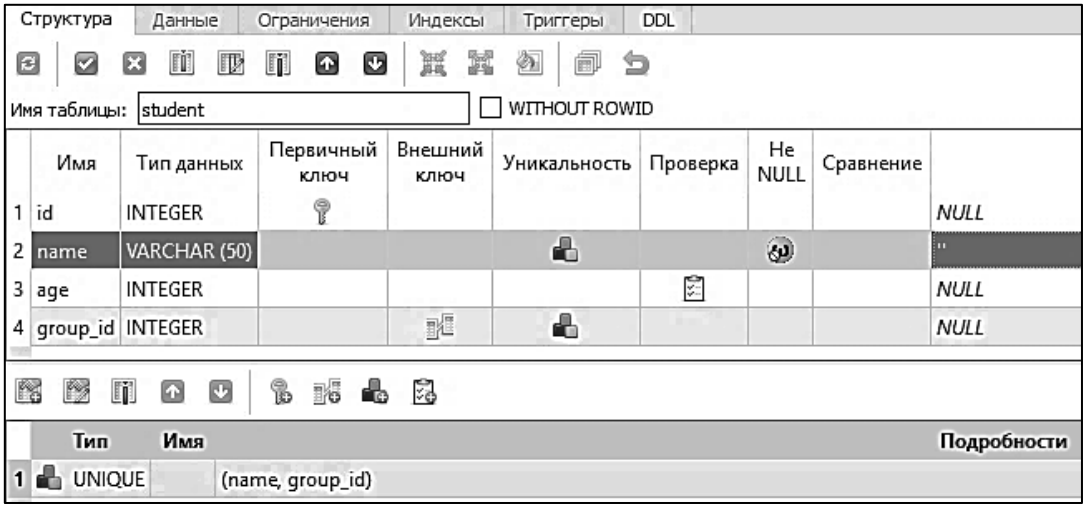

Рис. 2.5.21. Структура таблиці «студенти»

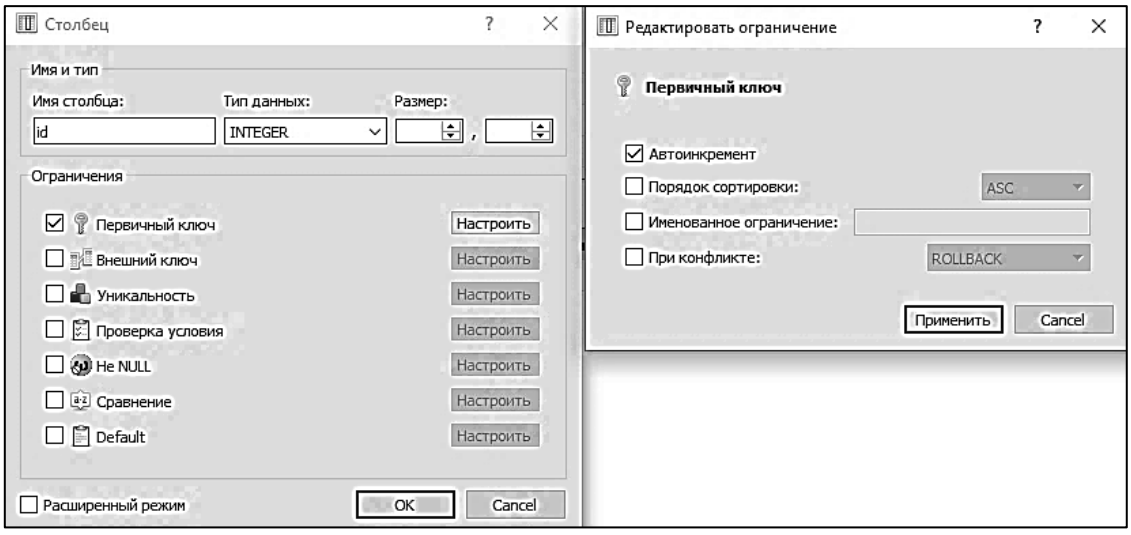

Рис. 2.5.22. Додавання обмеження цілісності первинний ключ та автоінкремент для поля id

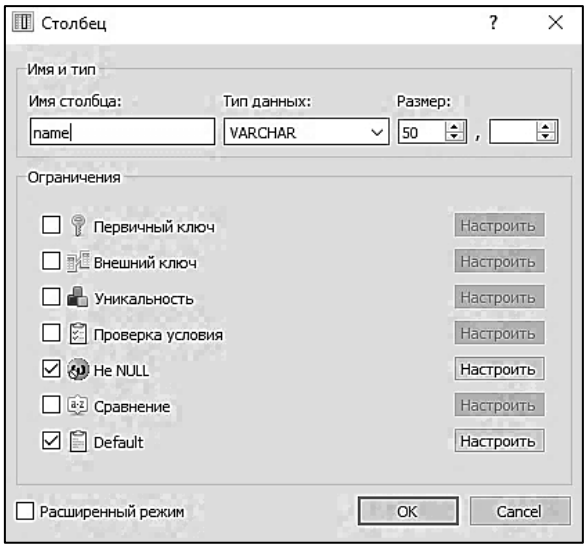

Рис. 2.5.23. Додавання обмеження цілісності not null та значення за замовчуванням для поля name

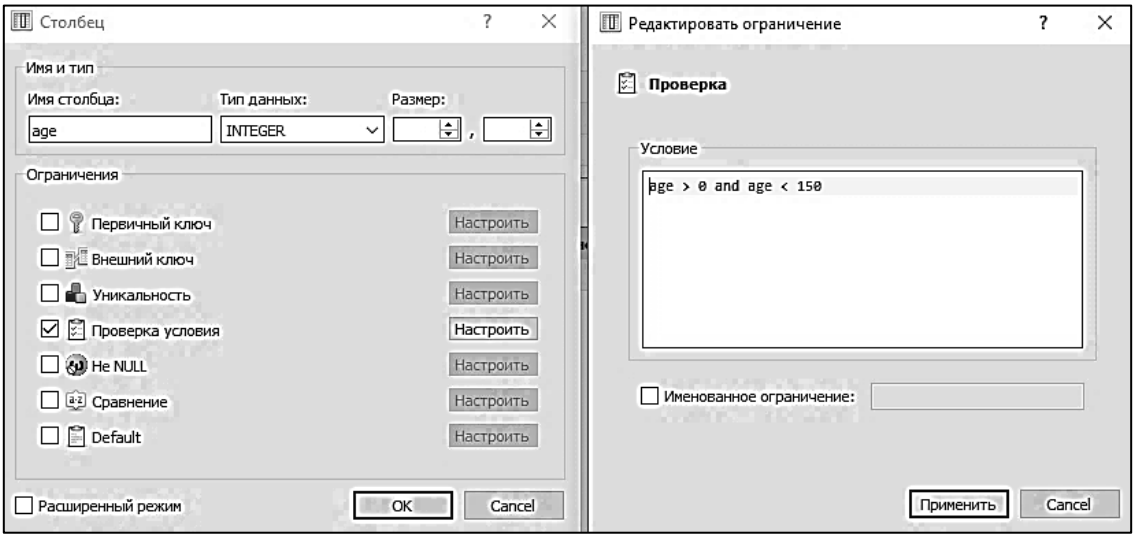

Рис. 2.5.24. Додавання перевірки для поля age

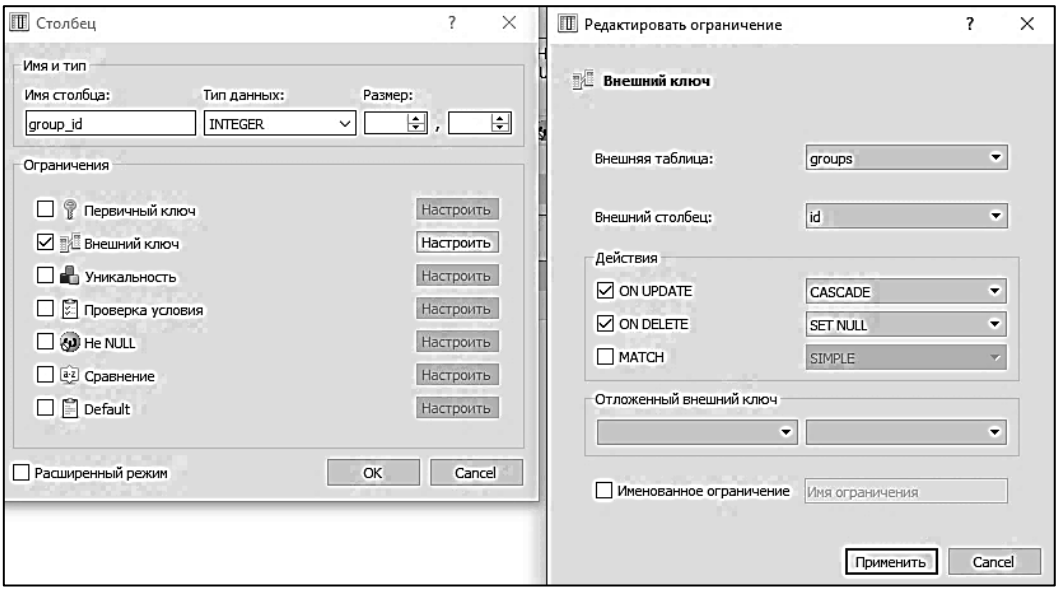

Рис. 2.5.25. Додавання обмеження цілісності зовнішній ключ для поля «group_id»

Виконуємо згенерований скрипт (рис. 2.5.26).

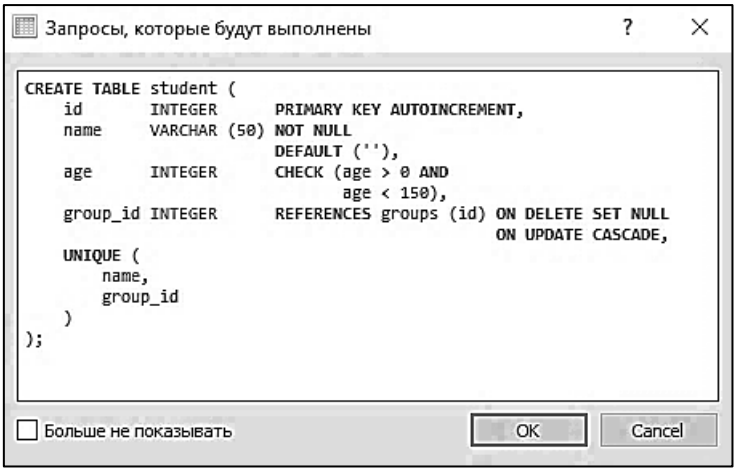

Рис. 2.5.26. Виконання скрипта на створення таблиці «студенти»

Перед заповненням даними таблиці «студент», створимо тригери для підтримки в актуальному стані поля «кількість студентів». При додаванні студента це поле потрібно збільшити для відповідної групи, при видаленні зменшити. Для спрощення задачі додатково створимо тригер, що заборонить зміну поля «група» таблиці «студенти» (рис. 2.5.27–2.5.33).

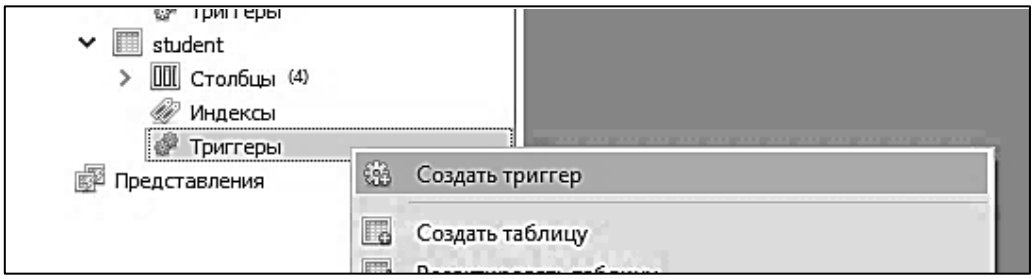

Рис. 2.5.27. Створення тригера для таблиці «студенти»

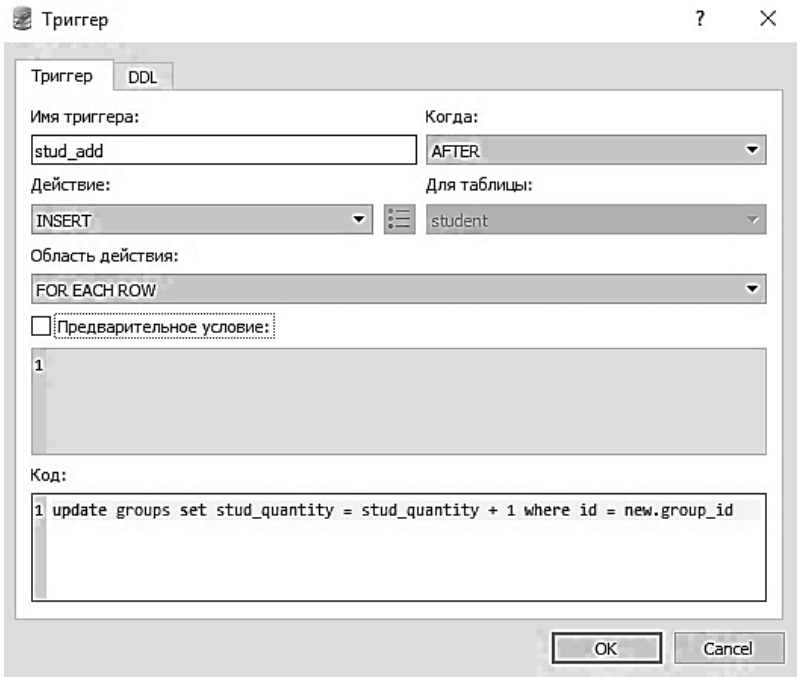

Рис. 2.5.28. Завдання властивостей та тіла тригера на додавання даних

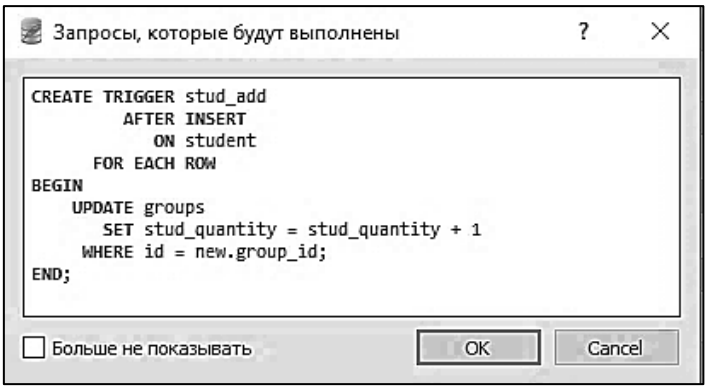

Рис. 2.5.29. Виконання скрипта та створення тригера на додавання даних

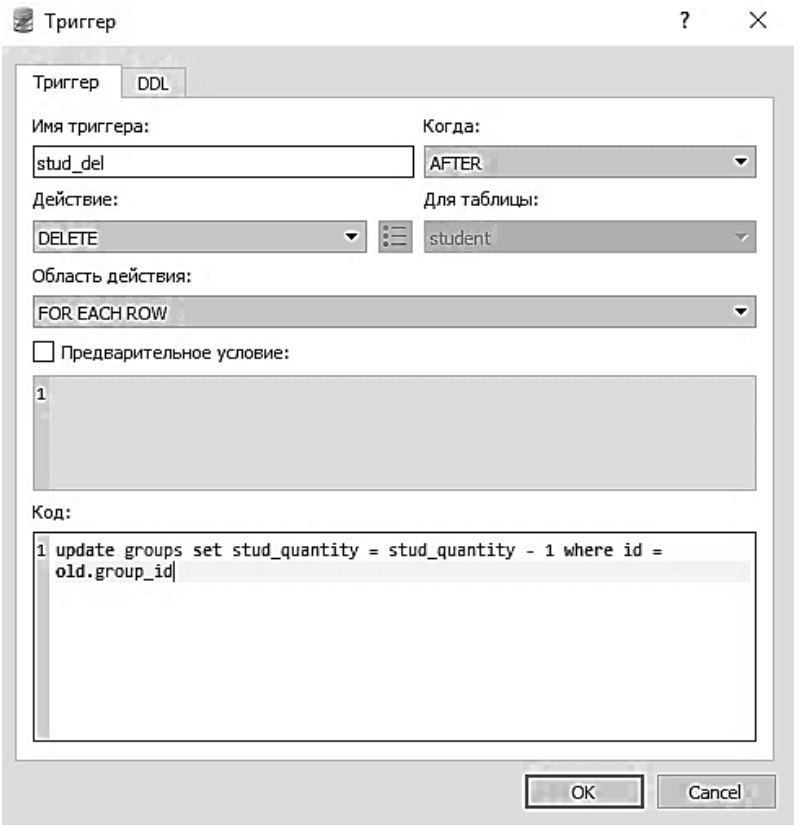

Рис. 2.5.30. Задавання властивостей та тіла тригера на видалення даних

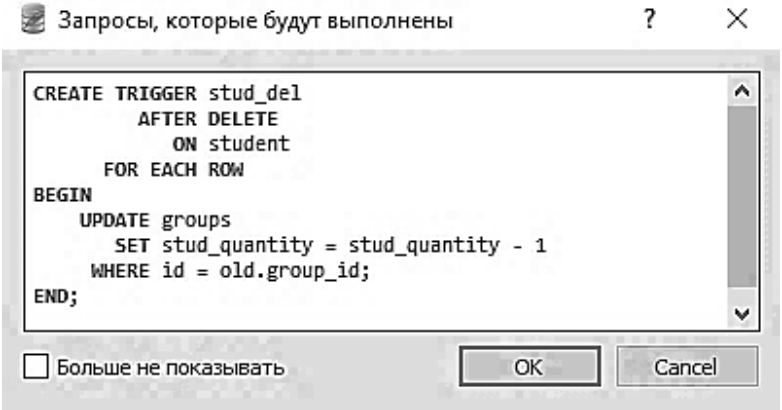

Рис. 2.5.31. Виконання скрипта та створення тригера на видалення даних

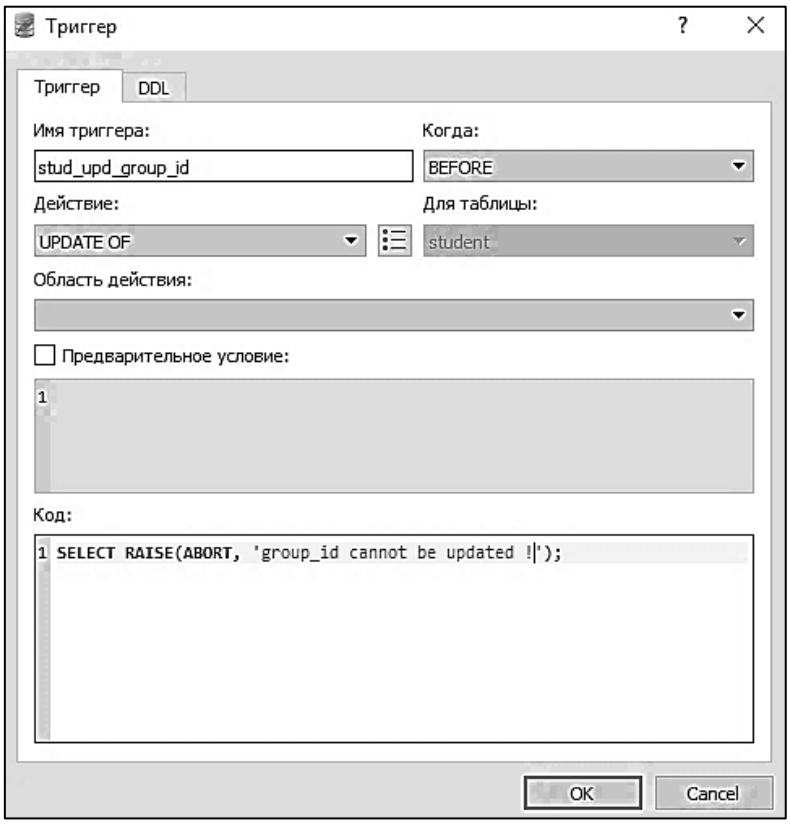

Рис. 2.5.32. Задавання властивостей та тіла тригера, що забороняє зміну коду групи

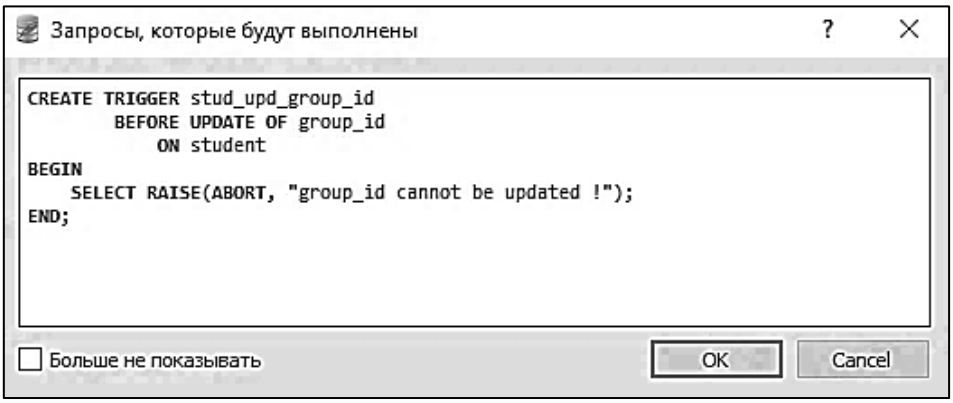

Рис. 2.5.33. Виконання скрипта та створення тригера,
що забороняє зміну коду групи

Додамо до таблиці «студенти» декілька рядків та перевіримо роботу тригерів та правила. Спробуємо додати до таблиці 4-х студентів (рис. 2.5.34).

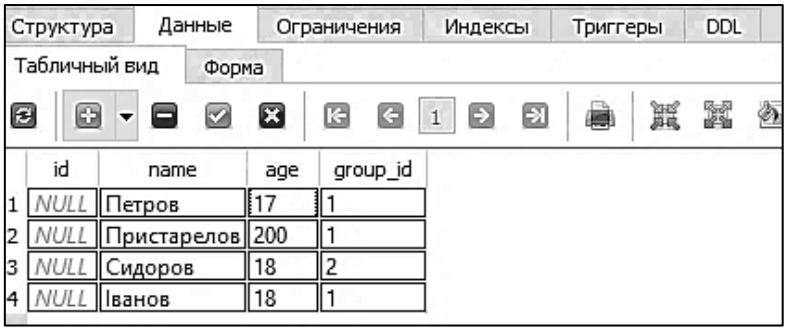

Рис. 2.5.35. Зміни, що суперечать правилу для поля age

При підтвердженні отримуємо порушення правила (рис. 2.5.36).

	id	name	age	group_id
1	1	Петров	17	1
2	NULL	Пристарелов	200	1
3	NULL	Сидоров	18	2
4	NULL	Іванов	18	1

[15:01:52] Error while committing new row: CHECK constraint failed: student

Рис. 2.5.36. Порушення check constraint

Видаляємо помилковий рядок та повторюємо операцію.
Переходимо до перегляду вмісту таблиці «групи» (рис. 2.5.37).

	id	number	stud_quantity
1	1	103	2
2	2	102	1
3	3	101	0

Рис. 2.5.37. Перегляд таблиці «групи»

Як бачимо, вміст поля «кількість студентів» був змінений автоматично при додаванні студентів у групу.

Спробуємо змінити номер групи у студента та переконуємось, що наш тригер не дозволяє цього зробити.

	id	name	age	group_id
1	1	Петров	17	1
2	2	Сидоров	18	2
3	3	Іванов	18	3

[15:06:04] An error occurred while committing the data: group_id cannot be updated !

Рис. 2.5.38. Перевірка заборони зміни номера групи

Видаляємо студента та перевіряємо зміну кількості студентів у групі (рис. 2.5.39 – 2.5.40).

	id	name	age	group_id
1	1	Петров	17	1
2	2	Сидоров	18	2

Рис. 2.5.39. Видалення студента

	id	number	stud_quantity
1	1	103	1
2	2	102	1
3	3	101	0

Рис. 2.5.40. Перевірка зміни кількості студентів у таблиці «групи»

Створимо представлення для відображення списку студентів із номером групи, в якій він навчається (рис. 2.5.41–2.5.43).

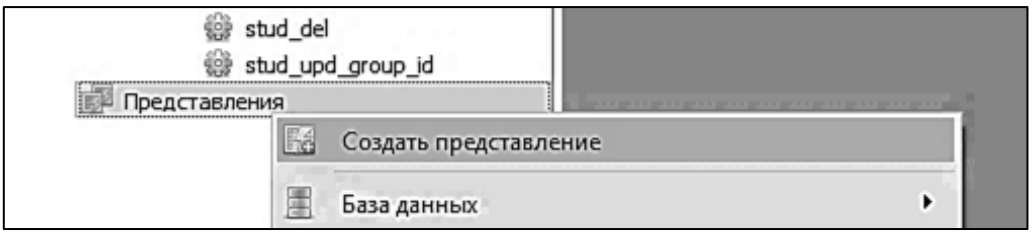

Рис. 2.5.41. Створення нового представлення

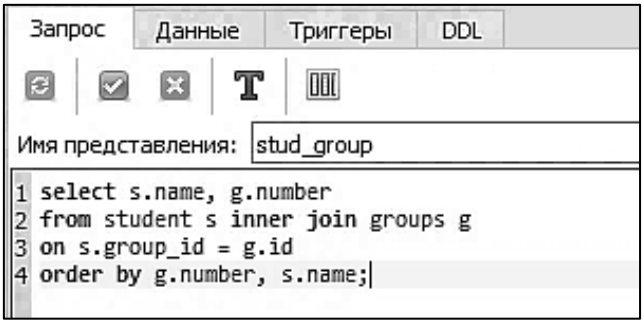

Рис. 2.5.42. Представлення для відображення списку студентів із номером групи

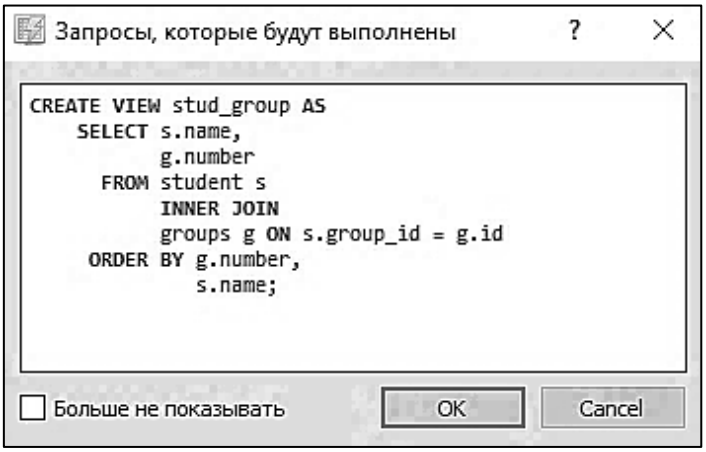

Рис. 2.5.43. Виконання згенерованого коду для створення представлення

На вкладці «дані» переглянемо результат (рис. 2.5.44).

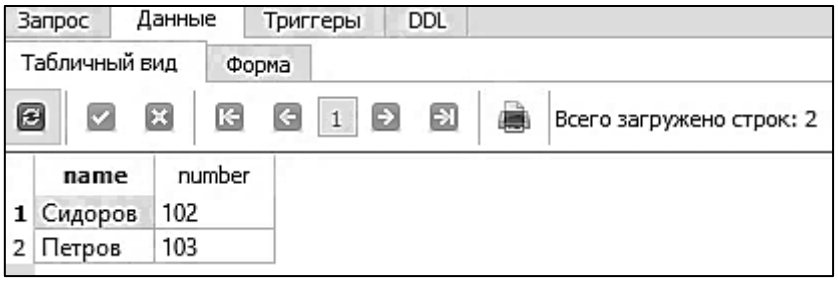

Рис. 2.5.44. Дані створеного представлення

Тепер напишемо довільний SQL-запит. Для цього відкриємо редактор (рис. 2.5.45).

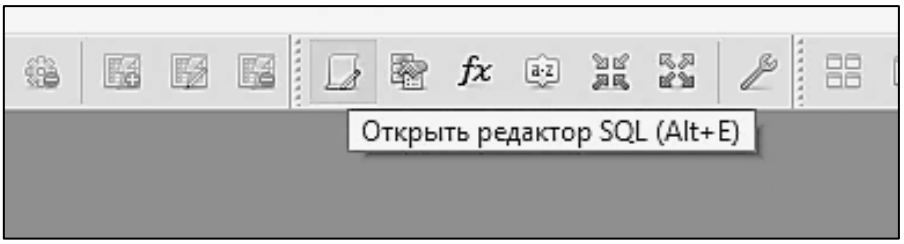

Рис. 2.5.45. Відкриття редактора для написання SQL-запиту

Та наберемо наступний запит, що має повернути список номерів непорожніх груп, тобто тих, де є хоча б 1 студент (рис. 2.5.46–2.5.47).

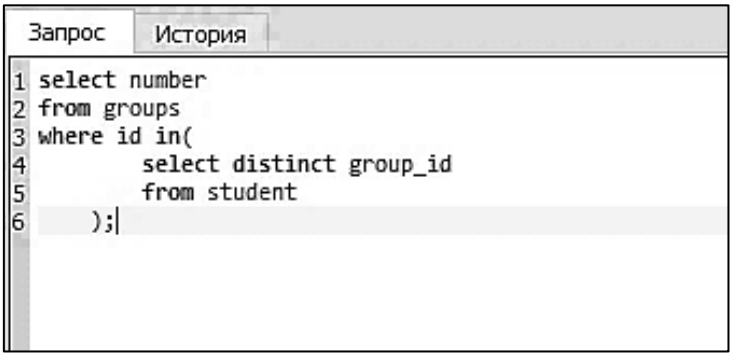

Рис. 2.5.46. Запит на вибірку даних

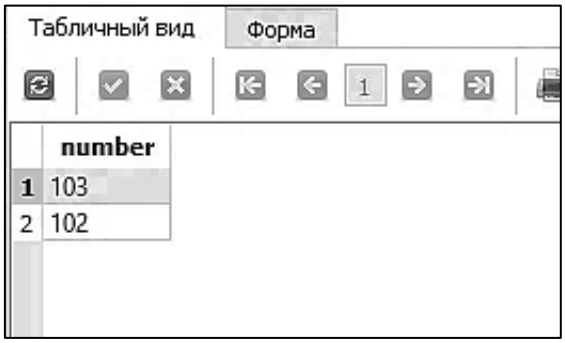

Рис. 2.5.47. Результат виконання запиту на вибірку даних

2.6. Написання клієнтського застосунку на мові PHP

Під час виконання лабораторної роботи будемо використовувати пакет OpenServer, що включає також скриптову мову програмування PHP. OpenServer не вимагає інсталяції, папка з ПО може бути скопійована на диск, після чого потрібно просто запустити Open Server x64.exe або Open Server x86.exe залежно від версії ОС.

У PHP підтримка SQLite починаючи із версії 5 включена автоматично, тому ніяких додаткових розширень встановлювати не доведеться. Переконаємось лише, що відповідний модуль доступу до даних підключено в розділі extensions нашого php.ini.

За умови використання OpenServer у ролі веб-сервера, запускаємо «Додатково» -> «Конфігурація» -> «PHP *» (рис. 2.6.1).

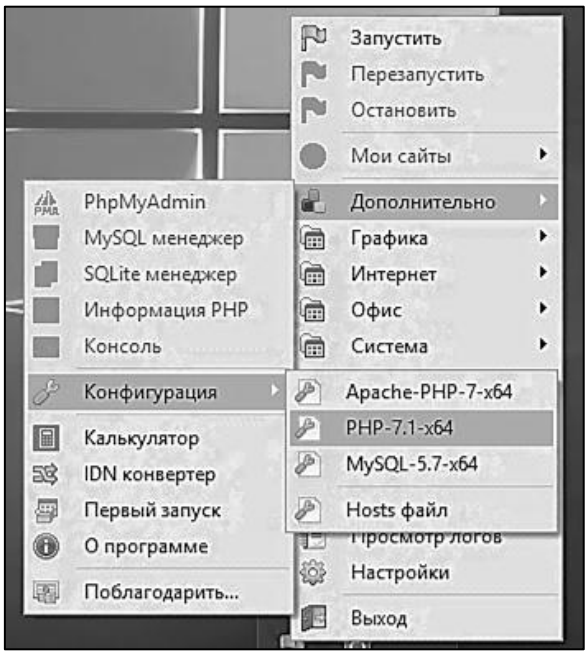

Рис. 2.6.1. Меню OSPanel для зміни php.ini

У вікні текстового редактора, що з'явився, переконуємося, що рядок «extension=php_pdo_sqlite.dll» розкоментовано (у разі необхідності прибираємо перший символ рядка «;» та зберігаємо файл (рис. 2.6.2).

```
182    extension=php_sqlite3.dll
183    ;extension=php_sysvshm.dll
```

Рис. 2.6.2. Зміни в php.ini для підключення драйвера sqlite

Якщо веб-сервер було запущено, він має бути перезавантажений. Далі для роботи з БД будемо використовувати вбудований клас SQLite3. При створенні екземпляра класу відбувається автоматичне підключення БД, отже в конструктор необхідно передати параметр – назви файлу БД.

$myDB = new SQLite3('myDatabase.db');

Крім цього, будемо використовувати ще 2 методи класу SQLite3. Перший – метод exec(), наприклад, $myDB->exec('create table t1(f1 int);'); якому в якості параметра передається SQL запит для виконання. Метод повертає істину у разі успішного виконання, і хибність у випадку виникнення помилки.

Другий – метод query(), він є дещо схожим на попередній, також приймає параметр – SQL запит для виконання але повертає об'єкт SQLite3Result, або false у випадку виникнення помилки.

Клас SQLite3Result представляє доступ до результуючого набору розширення SQLite3. У цьому класі використаємо метод fetchArray, що повертає рядок із результуючого набору. Метод викликається із параметром SQLITE3_ASSOC, що повертає асоціативний масив, у якому індекс відповідає назві стовпчика у результуючому наборі.

Переходимо до реалізації програмного коду, що відповідатиме за відображення, додавання, зміну та видалення даних із таблиці «Типи страв» бази даних «Їжа». У папці з OpenServer у директорії domains/localhost/ папку «sqlite», створимо файл sqlite.php, в якому сформуємо клас Food, що буде відповідати за операції із БД. Підключатися до БД будемо під час сформуємо екземпляра, тобто у конструкторі. Крім того знадобиться ще 5 методів для отримання списку типів їжі, читання одного з типів їжі, додавання, зміни та видалення типів їжі відповідно.

Нижче, на рис. 2.6.3 наведено клас Food:

```php
<?
class Food {
    private $db;

    function __construct() {
        $this->db = new SQLite3('c:\\temp\\sqlite\\foods.db');
    }

    function getFoodTypes() {
        $res = array();
        $sql = 'select id, name from food_types';
        $ret = $this->db->query($sql);
        while($row = $ret->fetchArray(SQLITE3_ASSOC) ) {
            $res[] = $row;
        }
        return $res;
    }

    function addFoodType($name) {
        $sql = "insert into food_types(id, name) select max(id)+1, '" . $name ."' from food_types";
        $this->db->exec($sql);
    }

    function removeFoodType($id) {
        $sql = "delete from food_types where id = " . $id;
        $this->db->exec($sql);
    }

    function editFoodType($id, $name) {
        $sql = "update food_types set name = '" . $name . "' where id = " . $id;
        $this->db->exec($sql);
    }

    function getFoodTypeName($id) {
        $sql = 'select name from food_types where id = ' . $id;
        $ret = $this->db->query($sql);
        if($row = $ret->fetchArray(SQLITE3_ASSOC) ) {
            return $row['name'];
        }
        return '';
    }
}
```

Рис. 2.6.3. Клас «Food», що реалізує роботу із таблицею «їжа»

Далі створимо index.php, що буде відповідати за відображення списку типів їжі у таблиці. Крім того, тут реалізуємо форму додавання нового типу їжі. Спочатку підключаємо sqlite.php та далі у коді створюємо екземпляр Food та використовуємо метод getFoodTypes() для отримання вмісту таблиці «Типи їжі» (рис. 2.6.4).

```php
<?php
    require 'sqlite.php';
?>
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="style.css">
    <title>food types</title>
</head>
<body>
    <form name='add-food-type' action='add.php' method='post'>
        Новий тип їжі <input type="text" name="food-type-name">
        <input type="submit" value="Додати">
    </form>
```

```
15      <table class="food-types-list">
16          <thead>
17              <tr>
18                  <th class='id'>id</th><th class='name'>name</th><th></th>
19              </tr>
20          </thead>
21          <tbody>
22              <?php
23              $food = new Food();
24              $foodTypes = $food->getFoodTypes();
26                  ?>
27                  <tr>
28                      <td class='id'><?php echo $foodType['id']; ?></td>
29                      <td class='name'><?php echo $foodType['name']; ?></td>
30                      <td>
31                          <a href="edit.php?id=<?php echo $foodType['id']; ?>">Змінити</a> | <a
                            href="delete.php?id=<?php echo $foodType['id']; ?>">Видалити</a>
32                      </td>
33                  </tr>
34                  <?php
35              }
36              ?>
37          </tbody>
38      </table>
39
40  </body>
41  </html>
```

Рис. 2.6.4. Вміст файлу головної сторінки index.php

У головному файлі ми посилаємося на add.php, що має додавати новий рядок до таблиці, delete.php для видалення рядку, та edit.php, в якому реалізована форма редагування та безпосередньо сама зміна даних у БД. На рис. 2.6.5–2.6.7 наведемо програмний код їх реалізації.

```
1   <?php
2   require 'sqlite.php';
3   if ($_POST['food-type-name']) {
4       $food = new Food();
5       $food->addFoodType($_POST['food-type-name']);
6   }
7   header("Location: index.php");
```

Рис. 2.6.5. Реалізація Add.php

```
1   <?php
2   require 'sqlite.php';
3   if ($_GET['id']) {
4       $food = new Food();
5       $food->removeFoodType($_GET['id']);
6   }
7   header("Location: index.php");
```

Рис. 2.6.6. Реалізація Delete.php

```
1   <?php
2   require 'sqlite.php';
3   $food = new Food();
4   if ($_GET['id'] && $_POST['food-type-name']) {
5       $food->editFoodType($_GET['id'], $_POST['food-type-name']);
6       header("Location: index.php");
7   }
8   ?>
```

```
<!DOCTYPE html>
<html>
<head>
    <title>food types</title>
</head>
<body>
    <form name='food-types-edit' method='post' action='edit.php?id=<?php echo $_GET['id'];?>'>
        Тип їжі <input type="text" name="food-type-name" value="<?php echo $food->getFoodTypeName($_GET
        ['id']); ?>">
        <input type="submit" value="Змінити">
    </form>
</body>
</html>
```

Рис. 2.6.7. Реалізація Edit.php

Для форматування виводу створимо style.css та підключимо його до index.php (рис. 2.6.8).

```
table.food-types-list td, table.food-types-list th {
    border: solid 1px black;
    padding: 5px;
}
table.food-types-list th {
    background-color: lightgray;
}
table.food-types-list th.id, table.food-types-list td.id {
    width: 50px;
    text-align: center;
}
table.food-types-list th.name, table.food-types-list td.name {
    width: 200px;
}
form[name='add-food-type'] {
    margin-top: 20px;
    margin-bottom: 20px;
}
```

Рис. 2.6.8. Таблиця стилів style.css

Перевіримо роботу додатку, набравши в адресному рядку браузера «http://localhost/sqlite/» (рис. 2.6.9). Спробуємо додати, змінити та видалити дані.

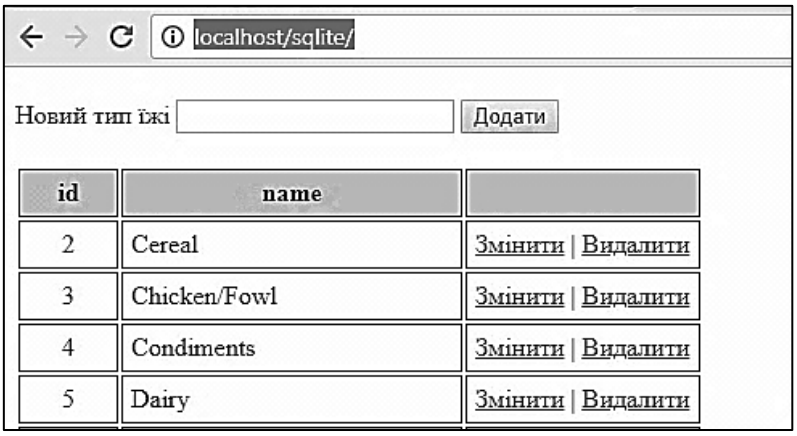

Рис. 2.6.9. Вигляд сторінка для редагування таблиці «їжа»

ЗАВДАННЯ ДЛЯ ІНДИВІДУАЛЬНОГО ВИКОНАННЯ

Завдання № 1.

У окремій БД створити 2 таблиці із вказаними обмеженнями цілісності первинного ключа (в кожній таблиці), зовнішній ключ (в одній із таблиць), хоча б однієї перевірки та 2-х індексів (один з яких унікальний).

Завдання № 2.

Для БД «Страви» реалізувати наступні запити на вибірку даних:

1. Вибрати типи страв, що починаються з літери «D»
2. Вибрати страви із кодом категорії від 3 до 5.
3. Вибрати серії 9 сезону, в назві яких зустрічається «fin»
4. Типи страв із кількістю страв, що входять до них
5. Перші 3 типи страв за кількістю страв
6. Страва, що згадується у найбільшій кількості епізодів.
7. Скільки різних типів страв згадується у кожному із сезонів.

Завдання № 3.

Для запитів 1–3 завдання 2 створити представлення. Переробити запити 4–7 завдання 2 наступним чином: створити представлення що включає всі необхідні для запиту поля і таблиці (зі зв'язками). Наступний запит з реченням group by спрямувати до створеного представлення.

Завдання № 4.

1. Створити таблицю-журнал тригер, на видалення даних із таблиці foods (зареєструвати назву блюда, що видаляється). Додати та потім видалити рядок із таблиці foods для перевірки роботи тригера.

2. Створити тригер, який під час видалення серії перевіряє, чи вона не є останньою у сезоні, але якщо так, відміняє операцію.

3. Створити тригер, що забороняє зміну поля, сезон таблиці епізодів.

4. Створити тригер, що забороняє додавати більше 12 різних сезонів (номер сезону може бути >12, наприклад 1, 2 та 20 сезони – всього їх 3, отже вимога не порушується).

5. Створити 3 тригери (додавання, зміна та видалення) до однієї з таблиць зі свого варіанта індивідуального завдання. Зміст тригера задати за власним бажанням.

Завдання № 5.

Користуючись графічним менеджером або консоллю (на власний розсуд) додати до БД, що була виконана у завданні 1, мінімум 5 таблиць із первинними та зовнішніми ключами та ін. необхідними обмеженнями цілісності даних, попередньо узгодивши структуру БД із викладачем.

Завдання № 6.

На мові програмування на власний вибір реалізувати форму перегляду та редагування (додавання, зміни та видалення) даних однієї з таблиць своєї БД.

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Kreibich Jay. Using SQLite. Small. Fast. Reliable. Choose Any Three. – O'Reilly Media, 2010. – 530 p.
2. Allen Grant, Owens Mike. The Definitive Guide to SQLite. – Apress, 2011. – 368 p.
3. Sibsankar Haldar. SQLite Database System Design and Implementation (Second Edition, Version 2). – O'Reilly, 2016. – 264 p.
4. What Is SQLite? [Електроний ресурс]. Режим доступу : URL : https://www.sqlite. org/index.html. – Загол. з екрану.
5. Кренке Д. Теория и практика построения баз данных. 8-е изд. – СПб. : Питер, 2003. – 800 с.
6. Дейт К. Дж. Введение в системы баз данных, 7-е издание. : пер. с англ. – М. : Издательский дом «Вильямс», 2001. – 1072 с.
7. Грофф Дж., Вайнберг П. SQL : Полное руководство : пер. с англ. – 2-е изд. – К. Издательская группа BHV, 2001. – 816 с.
8. Конноллн Т., Карелии Б. Базы данных. Проектирование, реализация и сопровождение. Теория и практика. 3-е издание. : пер. с англ. – М. : Издательский дом «Вильямс», 2003. – 1440 с.
9. Майкл Дж. Хернандес, Джон Л. Вьескас. SQL-запросы для простых смертных. Практическое руководство по манипулированию данными в SQL. – Москва : Изд-во «Лори», 2003. – 460 с.
10. Райордан Р. Основы реляционных баз даннях / Пер. с англ. – М. : Издательско-торговый дом «Русская Редакция», 2001. – 384 с.
11. Форта Бен. SQL за 10 минут. 4-е изд. : пер.с англ. – М. : ООО «И. Д. Вильямс», 2014. – 288 с.
12. Пасічник В. В., Резниченко В. А. Організація баз даних та знань. К. : Видавн. група BHV, 2006. – 384 с.
13. Фісун М. Т., Ніколенко С. Г. Створення та ведення баз даних засобами мови Jet SQL. Методичні вказівки до виконання блоку лабораторних робіт з дисципліни «Організація баз даних». – Миколаїв : видавн. ЧДУ ім. Петра Могили, 2009. – 83 с.
14. Фісун М. Т. Організація баз даних : методичні вказівки до виконання курсових робіт. – Миколаїв : Вид-во ЧДУ ім. Петра Могили, 2012. – 126 с.

Навчальне видання

Фісун Микола Тихонович
Дворецький Михайло Леонідович
Ніколенко Світлана Григорівна
Дворецька Світлана Володимирівна

Організація баз даних. Цикл лабораторних робіт у середовищі СКБД SQLite

Методичні вказівки для підготовки бакалаврів у галузі знань «Інформаційні технології»

Випуск 269

Редактор *А. Грубкіна.*
Технічний редактор, комп'ютерна верстка *Н. Хасянова.*
Друк, фальцювально-палітурні роботи *С. Волинець.*

Підп. до друку 11.05.2019
Формат 60х84$^{1}/_{16}$. Папір офсет.
Гарнітура «Times New Roman». Друк ризограф.
Ум. друк. арк. 12,56. Обл.-вид. арк. 3,67.
Тираж 5 пр. Зам. № 5708.

Видавець і виготовлювач: ЧНУ ім. Петра Могили.
54003, м. Миколаїв, вул. 68 Десантників, 10.
Тел.: 8 (0512) 50–03–32, 8 (0512) 76–55–81, e-mail: rector@chmnu.edu.ua.
Свідоцтво суб'єкта видавничої справи ДК № 6124 від 05.04.2018.