МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ ЕКОНОМІЧНИЙ УНІВЕРСИТЕТ ІМЕНІ СЕМЕНА КУЗНЕЦЯ

М. Ю. Лосєв
В. В. Федько

БАЗИ ДАНИХ

Навчально-практичний посібник для самостійної роботи студентів

Харків
ХНЕУ ім. С. Кузнеця
2018

УДК 004.65(076.034)
Л79

Авторський колектив: канд. техн. наук, доцент М. Ю. Лосєв – розділи 2, 4; канд. фіз.-мат. наук, доцент В. В. Федько – розділи 1, 3.

Рецензенти: професор кафедри обчислювальної техніки та програмування Національного технічного університету "ХПІ", канд. техн. наук, доцент *О. М. Рисований*; канд. техн. наук, доцент кафедри інформатики і прикладної математики Харківського національного автомобільно-дорожнього університету *Д. П. Лабенко.*

Рекомендовано до видання рішенням ученої ради Харківського національного економічного університету імені Семена Кузнеця.
Протокол № 5 від 29.01.2018 р.

Самостійне електронне текстове мережеве видання

Лосєв М. Ю.

Л79 Бази даних : навчально-практичний посібник для самостійної роботи студентів [Електронний ресурс] / М. Ю. Лосєв, В. В. Федько. – Харків : ХНЕУ ім. С. Кузнеця, 2018. – 233 с.

ISBN 978-966-676-731-1

Наведено навчальний матеріал для самостійної підготовки студентів у процесі вивчення дисципліни. Розглянуто стандартні підходи до створення простих і складних запитів для введення, модифікації, вилучення і відображення даних у таблицях бази даних. Запропоновано велику кількість прикладів робочого проектування за допомогою технічних і програмних засобів. Приклади базуються на одній предметній області й охоплюють широкий спектр використання команд для системи управління базами даних MS SQL Server.

Рекомендовано для студентів галузі знань 12 "Інформаційні технології" першого (бакалаврського) рівня всіх форм навчання.

УДК 004.65(076.034)

ISBN 978-966-676-731-1

Розділ 1. Вступ. Системи баз даних. Основні поняття та архітектура. SQL і реляційні бази даних

1.1. Мета та завдання дисципліни, її місце у навчальному процесі

Мета вивчення дисципліни:

сформувати системне базове уявлення, первинні знання, вміння і навички студентів з основ організації баз даних і знань як наукової і прикладної дисципліни, достатні для подальшого продовження освіти та самоосвіти в галузі обчислювальної техніки, інформаційних систем різного призначення;

дати уявлення про роль і місце баз даних в автоматизованих інформаційних системах, про призначення і основні характеристики різних систем управління базами даних, їх функціональні можливості;

отримання базового рівня щодо роботи та програмування в середовищі СУБД *SQL Server*;

теоретична підготовка в сфері проектування баз даних і використання СУБД;

використання мови SQL для роботи з базами даних.

Предметом навчальної дисципліни є безпосередньо сама база даних як ядро банка даних; базові моделі, що закладені в основу сучасних баз даних; мовні засоби спілкування проектувальників і користувачів з СУБД; засоби проектування баз даних.

Об'єктом навчальної дисципліни є:

бази даних як система, що складається з частин; роль, місце та зв'язок її частин як елементів системи;

основні розділи систем баз даних, їх структури, вирішувані в них завдання, методи дослідження, зв'язок з іншими розділами та іншими дисциплінами, базові поняття;

рівні подання даних, структури даних, основні операції, що виконуються над даними;

ієрархічні, мережеві та реляційні моделі даних, їх сутність, відмінність один від одного та можливі додатки;

реляційна алгебра;

нормальні форми схем відносин; проектування реляційних баз даних; мова маніпулювання даними для реляційної моделі;

мова запитів SQL, методи оптимізації запитів; їх схожість, відмінність; коло розв'язуваних з їх допомогою задач;

програмування в середовищі СУБД Access, Oracle, SQL Server: можливості створення і модифікації баз даних; методи організації пошуку, сортування, індексування баз даних; створення форм і звітів; основи технології програмування і методи розв'язування обчислювальних задач;

захист баз даних, методи забезпечення цілісності та збережуваності даних у базі, їх порівняння, сфери застосування.

Завданням з вивчення навчальної дисципліни "Бази даних" є теоретична та практична підготовка майбутніх фахівців із таких питань:

системи баз даних, основні поняття і архітектура;

моделі даних, реляційна модель даних;

семантичне моделювання предметної області;

візуалізація моделей даних сучасними засобами;

нормалізація реляційної моделі даних;

загальна характеристика мовних засобів спілкування зі СУБД, мови DDL SQL і DML SQL;

особливості реалізації SQL у СУБД MS SQL Server;

проектування баз даних;

застосування CASE-засобів до проектування баз даних;

забезпечення цілісності даних;

захист даних у СУБД;

транзакції та паралельна обробка у базах даних;

системи аналітичної обробки даних;

сховища даних та OLAP-системи;

сучасні підходи до створення баз даних (об'єктно-орієнтовані та об'єктно-реляційні бази даних; просторові бази даних; розподілені бази даних; бази даних у мережі Інтернет);

бази даних NoSQL;

перспективи розвитку баз даних.

Освоєння цих важливих питань дозволяє студентам оволодіти вмінням в рамках компетентностей, визначених стандартами вищої освіти України [7].

У результаті вивчення навчальної дисципліни студент повинен:

знати:

концепції та принципи організації банків і баз даних;

місце банків даних в інформаційних системах (ІС);

рівні абстракції даних у проектуванні баз даних;

вимоги до баз даних і методи їхнього забезпечення;

загальні принципи побудови системи управління базами даних, структуру та принципи її функціонування;

основні моделі та мовні засоби СУБД;

принципи організації структур збереження даних і методи доступу до даних;

відмінності між транзакційною та аналітичною базами даних;

етапи проектування баз даних;

методи розроблення логічної і фізичної моделей і фізичної схеми даних у середовищі сучасних CASE-інструментів та засобами СУБД;

принципи взаємодії з реляційними системами управління базами даних і базами даних NoSQL;

уміти:

формулювати вимоги до бази даних і забезпечувати її властивості;

проектувати концептуальну модель даних конкретної предметної області;

застосовувати принципи побудови баз даних – ненадлишковості, несуперечності та незалежності даних;

вибирати СУБД у процесі технічного проектування на основі оцінних варіантів баз даних, вимог користувачів, аналізу технічних, економічних, функціональних, сервісних характеристик СУБД, використовуючи науково-технічну, довідкову інформацію;

розробляти логічну структуру бази даних у процесі технічного проектування за допомогою методу нормалізації відношень, використовуючи методи реляційної алгебри, рівні абстракції даних, вимоги вибраної СУБД;

розробляти таблиці баз даних і встановлювати зв'язок між ними в умовах технічного проектування за допомогою відповідного технічного та програмного забезпечення, використовуючи конструктори таблиць;

розробляти фізичну структуру бази даних у процесі робочого проектування за допомогою вибраної СУБД, використовуючи сучасні технічні та програмні засоби розробника баз даних;

виконувати операції реляційної алгебри;

будувати ER-діаграми CASE-засобами;

створювати таблиці баз даних за допомогою програмних і технічних засобів проектування баз даних, використовуючи візуальні інструменти інтегрованих оболонок розробника програмного забезпечення та засоби мови DDL;

розробляти введення, модифікацію, вилучення, відображення даних у таблиці бази даних у процесі робочого проектування за допомогою технічних і програмних засобів, використовуючи форми введення та модифікації даних і табличні режими та засоби мови DML;

розробляти навігації щодо набору даних в умовах доступу до потрібного запису набору даних за допомогою програмних засобів СКБД, використовуючи навігаційні методи об'єктів;

розробляти методи сортування, фільтрації, пошуку даних у процесі відбору потрібних даних, що відповідають будь-яким критеріям, за допомогою програмних засобів СУБД;

установлювати обмеження цілісності засобами мови SQL;

застосовувати заходи захисту даних засобами мови SQL;

здійснювати вибір засобів для побудови систем аналітичної обробки даних;

застосовувати аналітичні функції SQL для розв'язання конкретних економічних задач;

виконувати базові операції з базою даних MongoDB;

вибирати СУБД у процесі технічного проектування на основі оцінних варіантів баз даних, вимог користувачів, аналізу технічних, економічних, функціональних, сервісних характеристик СУБД.

Вивчення дисципліни "Бази даних" ґрунтується на знаннях і вміннях, які студенти отримали під час вивчення таких дисциплін: "Вступ до комп'ютерних наук", "Математичний аналіз", "Дискретна математика", "Програмування", "Алгоритми та структури даних", "Основи об`єктно-орієнтованого програмування" (рис. 1.1).

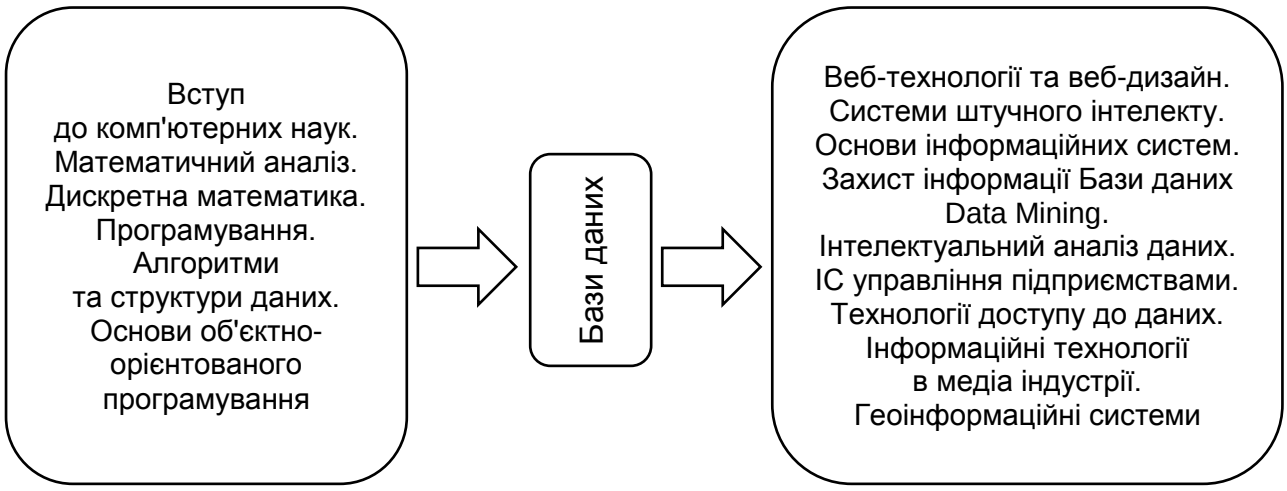


Рис. 1.1. **Місце дисципліни у навчальному процесі**

Дисципліна "Бази даних" забезпечує такі базові та вибіркові дисципліни: "Системний аналіз та проектування інформаційних систем", "Веб-технології та веб-дизайн", "Системи штучного інтелекту", "Основи інформаційних систем", "Захист інформації", "Бази даних Data Mining", "Інтелектуальний аналіз даних", "ІС управління підприємствами", "Технології доступу до даних", "Інформаційні технології в медіаіндустрії", "Геоінформаційні системи", "Системи обробки еколого-економічної інформації".

1.2. Основні поняття баз даних

Стрижневі ідеї сучасних інформаційних технологій базуються на концепції баз даних. Відповідно до цієї концепції основою інформаційних технологій є дані, які повинні бути організовані в бази даних з метою адекватного відображення мінливого реального світу.

Попри різноманіття визначень поняття бази даних завжди передбачається, що воно відповідає таким *ознакам*:

база даних містить деяку безліч даних, необхідних (і, бажано, достатніх) для вирішення конкретних завдань багатьох користувачів (у тому числі як реальних, так і потенційних) або (що, в принципі, майже те саме) для задоволення відповідних інформаційних потреб;

дані або інформаційні елементи в базі даних певним чином структуровані та пов'язані між собою (організовані). Водночас структура, склад даних і їх змістовність у базі даних не залежать від особливостей прикладних програм, що використовуються для управління базою даних;

дані (інформаційні елементи) подані на машинозчитуваних носіях у формі придатній для їх оперативного використання із застосуванням засобів обчислювальної техніки, включаючи і системи управління базами даних (СУБД).

Сукупність даних або інформаційних елементів, які не відповідають будь-якій з цих трьох ознак, не може називатися базою даних.

Існує безліч визначень поняття бази даних (БД), наведемо лише деякі з них.

База даних – спільно використовуваний набір логічно пов'язаних даних (і опис цих даних), призначений для задоволення інформаційних потреб організації [15].

База даних – сукупність взаємозв'язаних даних, організованих згідно схеми бази даних так, щоб з ними міг працювати користувач (ДСТУ 2874-94) [10].

База даних – сукупність інформаційних елементів з деякої предметної області, яка постійно зберігається в пам'яті обчислювальної системи (ВС) і використовується різними прикладними програмами [1].

Предметна область (ПО) – частина реального світу, дані про яку відображаються в базі даних. Предметна область описується в термінах класів об'єктів і їх взаємозв'язків, які є зрозумілими кінцевим користувачам і людям, що працюють у предметній області, проте не обізнані з принципами організації баз даних.

Модель предметної області – наші знання про предметну область.

У процесі дослідження предметної області будують **три рівня** подання інформації: інфологічний; даталогічний; фізичний.

На *інфологічному рівні* визначається, яка інформація про предметну область буде зберігатися і оброблятися в комп'ютері. Інформація в інфологічній моделі подається незалежно від того, які програмні та технічні засоби будуть використані в подальшому для її зберігання і обробки.

На *даталогічному рівні* інформація може надаватися у вигляді даних і логічних зв'язків між ними незалежно від того, чим є дані та які технічні засоби будуть використані для їх зберігання, але з урахуванням програмних засобів (СУБД). Існує кілька видів даталогічних моделей даних: мережева, ієрархічна, реляційна, об'єктна, документо-орієнтована та інші.

На *фізичному рівні* визначається, як і де на фізичному носії будуть зберігатися дані. Фізична модель даних описує дані засобами конкретної СУБД.

Для подання інформації про предметну область з точки зору одного конкретного користувача або групи користувачів можуть бути побудовані зовнішні моделі першого та другого рівнів – інфологічні та даталогічні.

Базу даних можна уявити як пойменовану сукупність даних, організовану за правилом "Три НЕ":

НЕнадлишковість;

Цілісність = достовірність + НЕсуперечливість + повнота;

НЕзалежність даних від додатків.

База даних *ненадлишкова*, якщо видалення будь-якого елемента даних або зв'язку між даними призводить до втрати інформації про ПО.

База даних *несуперечлива*, якщо всі збережувані в ній дані відповідають певним умовам (обмеженням цілісності).

Логічна незалежність передбачає можливість зміни однієї програми без коригування інших додатків, що працюють з тією ж БД.

Фізична незалежність передбачає можливість перенесення даних з одних зовнішніх носіїв на інші без зміни додатків.

Приклад 1.1. Нехай база даних "Робітники заводу" складається з однієї таблиці (табл. 1.1).

У наведених даних можуть бути встановлені обмеження цілісності:

рік народження менше року надходження на завод;

кожен робітник має унікальний табельний номер;

номер цеху однозначно визначає назву цеху;

номер цеху – ціле число і т. п.

Таблиця 1.1

Таблиця бази даних "Робітники заводу"

№ п/п	П. І. Б.	Рік народження	Дата	Посада	Назва цеху	Цех
25	Сопко О. І.	1990	2012	Слюсар	Механічний	2
26	Іванов П. М.	1987	2011	Коваль	Ливарний	4
27	Сєчин А. О.	1989	2013	Токар	Механічний	2
28	Жуков Г. А.	1992	2016	Слюсар	Складальний	1
29	Лиско М. П.	1980	2008	Токар	Механічний	3

База даних надлишкова: має місце надлишкове дублювання даних в останньому стовпчику таблиці.

База даних суперечлива: порушена відповідність між номером і назвою цеху в першому, третьому і останньому рядках таблиці.

База даних не буде надлишкова та несуперечлива, якщо її подати у вигляді табл. 1.2, 1.3.

За такої організації бази даних дублювання даних не є надмірним.

Таблиця 1.2

Таблиця "Робітники" бази даних "Робітники заводу"

№ п/п	П. І. Б.	Рік народження	Рік початку роботи	Посада
25	Сопко О. І.	1990	2012	Слюсар
26	Іванов П. М.	1987	2011	Коваль
27	Сєчин А. О.	1989	2013	Токар
28	Жуков Г. А.	1992	2016	Слюсар
29	Лиско М. П.	1980	2008	Токар

Таблиця 1.3

Таблиця "Цехи" бази даних "Робітники заводу"

№ цеху	Назва цеху
1	Складальний
2	Механічний
4	Ливарний

Основні завдання, що виникають у роботі з базою даних:

зберігання таблиць і записів;

управління записами та таблицями.

Отже, базу даних (або систему баз даних) можна уявити як набір програмно-апаратних засобів, що вирішують завдання централізованого зберігання і обробки даних користувача.

Інформаційна система (ІС) – це система, що включає інформаційні, програмні, технічні, мовні, організаційні функції, призначені для збору, накопичення, передачі, пошуку, обробки даних.

За значущістю функцій інформаційні системи класифікуються на інформаційно-пошукові системи і системи обробки даних.

Основні функції *інформаційно-пошукових систем* – пошук даних і висновок необхідних даних.

Основні функції *систем обробки даних* це оновлення даних і виведення необхідних даних.

Між власне фізичної базою даних і користувачами розташовується рівень програмного забезпечення, тобто система управління базою даних (СУБД).

СУБД – це сукупність мовних і програмних засобів, призначених для створення, ведення і сумісного використання бази даних багатьма користувачами.

Види СУБД:

промислові універсального призначення;

промислові спеціального призначення;

розроблені для конкретного замовника.

Функції СУБД:

безпосереднє управління даними у зовнішній пам'яті;

управління розділами оперативної пам'яті з метою прискорення роботи бази даних;

управління транзакціями. *Транзакція* – послідовність операцій над базою даних, що розглядаються СУБД як єдине ціле. Поняття транзакції необхідне для підтримки логічної цілісності бази даних. Механізм транзакцій забезпечує захист бази даних від апаратних збоїв, можливість багатокористувацького доступу до даних у віддалених базах;

журналізація – ведення журналу змін інформації в цілях підтримки надійності зберігання даних, а також підтримка мов бази даних, мовні засоби сучасних СУБД.

Переваги використання СУБД:

контроль за надмірністю даних;

контроль за несуперечливістю даних;

більше корисної інформації із тим ж обсягом збережених даних;

спільне використання даних користувачами;

підтримка цілісності даних;

підвищена безпека даних;

підвищення ефективності зі зростанням масштабів системи;

можливість знаходження компромісу за суперечливих вимог;

підвищення доступності даних і їх готовності до роботи;

поліпшення показників продуктивності систем;

спрощення супроводу системи за рахунок незалежності від даних;

розвинені служби резервного копіювання та відновлення.

Недоліки використання СУБД:

складність програмного забезпечення;

висока вартість СКБД;

додаткові витрати на апаратне забезпечення;

витрати на перетворення даних;

зниження продуктивності;

більш серйозні наслідки від виходу системи з ладу.

Банк даних – це інформаційна система, побудована на основі концепції бази даних і СУБД.

Поняття *інформаційної системи* ширше за поняття банку даних, тому що банк даних – це різновид ІС.

Поняття "банк даних" і "база даних" не є синонімами, оскільки база даних – інформаційна складова банку даних.

1.3. Архітектура баз даних

Рівень зовнішніх моделей – кожна модель має своє "бачення" даних. Цей рівень визначає точку зору на бази даних окремих додатків. Кожна програма бачить і обробляє тільки ті дані, які необхідні саме цьому додатку.

Концептуальний рівень – база даних подана в найбільш загальному вигляді, який об'єднує дані, використовувані всіма додатками, що працюють з даною базою даних. Фактично концептуальний рівень відображає узагальнену модель предметної області (об'єктів реального світу), для якої створювалася база даних.

Фізичний (внутрішній) рівень – власне дані, розташовані в файлах або в сторінкових структурах, які знаходяться на зовнішніх носіях інформації.

За характером використання СУБД розподіляють на *персональні* та *багатокористувальницькі*. Залежно від місця розташування окремих частин СУБД розрізняють *локальні* бази даних (автономні) та *віддалені* (розраховані на багато користувачів або мережеві). Перелік і особливості архітектур баз даних наведені на рис. 1.2.

Рис. 1.2. **Перелік і особливості архітектур баз даних**

На рис. 1.3. показані три види архітектури з'єднання клієнта з даними. Перший вид має одноланкову архітектуру, яка крім даних містить єдину ланку – клієнт (блоки, об'єднані зв'язком № 1 на рис. 1.3). Дволанкова архітектура містить ще сервер бази даних, що забезпечує значну частину логіки управління даними, тоді як клієнт в основному зайнятий відображенням даних в зручному для користувача вигляді (блоки, об'єднані зв'язком № 2 на рис. 1.3).

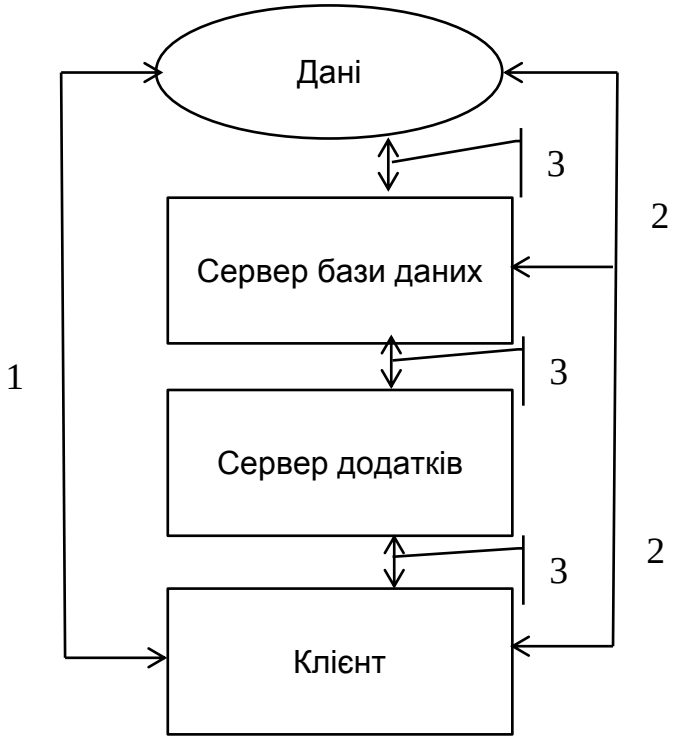


Рис. 1.3. **Архітектури з'єднання клієнта з даними: одноланкова, дволанкова, триланкова**

У триланкових СУБД додано ще одну проміжну ланку – сервер додатків (блоки, об'єднані зв'язком № 3 на рис. 1.3). *Сервер додатків* призначений для забезпечення зв'язку клієнта з сервером бази даних і повністю позбавляє клієнта від будь-яких турбот з управлінням даними.

1.4. SQL та реляційні бази даних

1.4.1. Система баз даних

Система баз даних – є не що інше, як комп'ютеризована система зберігання однотипних записів. Саму ж базу даних можна розглядати

як деяку електронну картотеку, тобто сховище або контейнер для набору файлів даних, занесених у комп'ютер [19].

Система баз даних складається з таких компонент: дані; апаратне забезпечення; програмне забезпечення; користувачі.

1.4.2. Моделі даних

Для формалізації подання даних використовуються моделі даних. **Модель даних** – це абстрактне, самодостатнє, логічне визначення об'єктів, операторів та інших елементів, що в сукупності складають абстрактну машину доступу до даних, з якою взаємодіє користувач [4]. Згадані об'єкти дозволяють моделювати структуру даних, а оператори – поведінку даних. Використовуючи це визначення, можна ефективно розмежувати поняття моделі даних від її реалізації.

Реалізація (implementation) **заданої моделі даних** є фізичним втіленням на реальній машині компонентів абстрактної машини, які в сукупності складають цю модель. **Модель** – це те, про що користувачі повинні знати, а **реалізація** – те, чого користувачі не повинні знати [5].

Аспекти моделі даних:

аспект структури визначає, чим логічно є база даних;

аспект маніпуляції визначає способи переходу між станами бази даних (тобто способи модифікації даних) і способи отримання даних з бази даних;

аспект цілісності визначає засоби описів коректних станів бази даних (обмеження).

Бази даних містять інформацію, яка використовується деякою прикладною системою (**додатком**). Під моделлю даних можна розуміти інтегрований набір елементів для опису даних, зв'язків між ними й обмежень, що накладаються на дані в деякій їх організації.

Дані структуруються так, щоб була забезпечена можливість подальшого нарощування додатків. Розвиток теорії і практики проектування та експлуатації баз даних супроводжувалися інтенсивним розвитком моделей даних.

У загальному сенсі модель є формою відображення дійсності (реальних об'єктів, предметів та явищ) за допомогою інших об'єктів, процесів, явищ або через абстрактний опис у вигляді зображення, плану, карти, сукупності рівнянь, алгоритмів, програм тощо.

Іншими словами, **модель даних** – це деяка абстракція, яка призначена для визначення правил структурування даних, процесів динамічної зміни даних і допустимих станів взаємопов'язаних даних.

У галузі побудови баз даних до класичних відносять ієрархічну, мережеву та реляційну моделі (рис. 1.4). Історично ці моделі з'явилися раніше, ніж інші. З них зараз найширше використовується тільки реляційна модель.

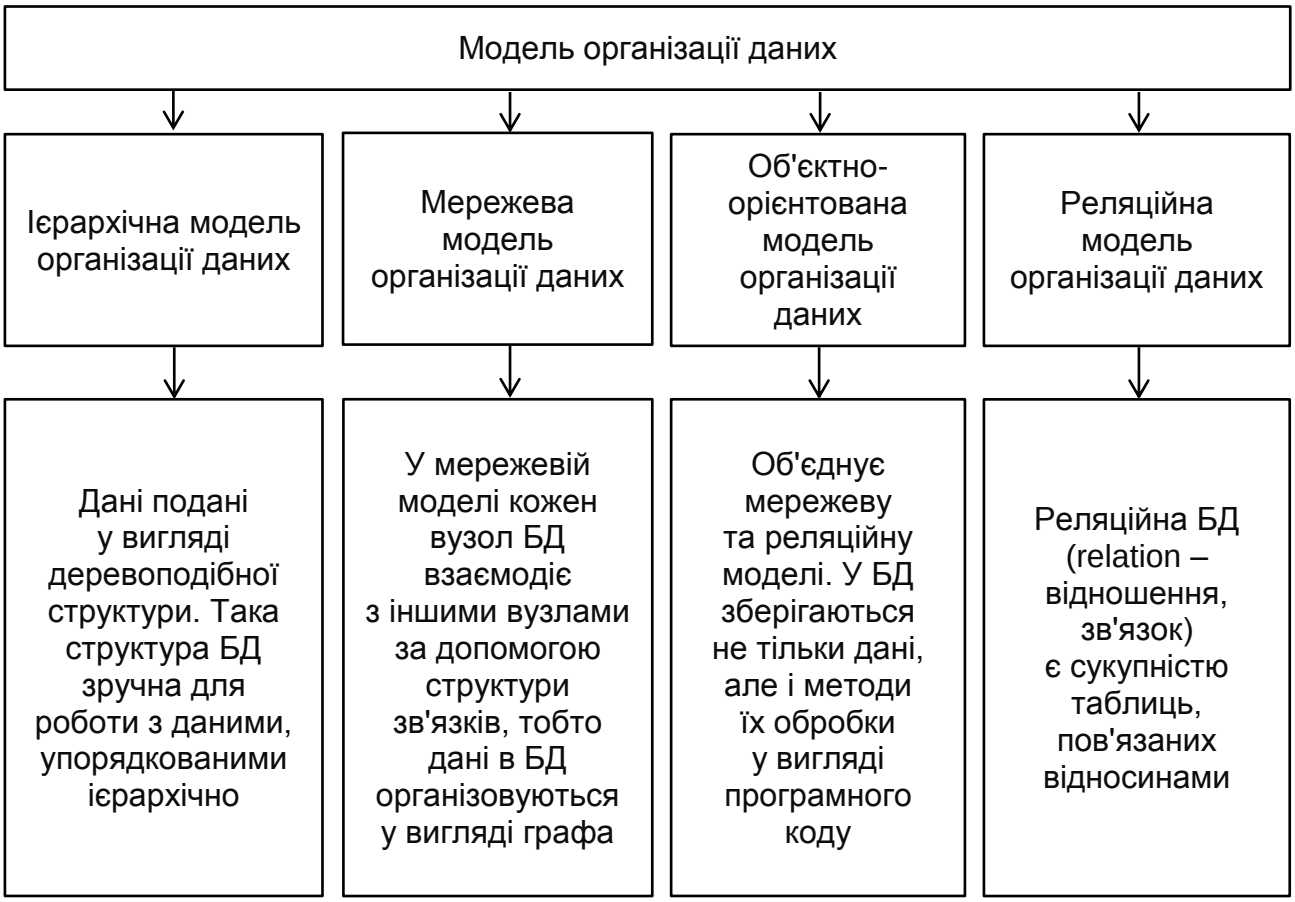


Рис. 1.4. **Моделі організації даних**

Гідністю реляційної моделі є її простота, гнучкість структури, зручність реалізації на комп'ютері. Ці характеристики забезпечують високу ступінь незалежності даних.

1.4.3. Теорія відношень як основа реляційної моделі

Теоретичною основою реляційної моделі стала теорія відношень, основу якої заклали два логіка – американець Ч. Пірс і німець Е. Шредер.

Пізніше американський математик Е. Кодд, скористувавшись працями попередників, сформулював основні поняття і обмеження реляційної моделі, створивши набір операцій з сімома основними й однією додатковою. Пропозиції Е. Кодда були настільки ефективні для систем баз даних, що він був удостоєний премії А. Тьюринга в галузі теоретичних основ обчислювальної техніки.

Реляційна модель даних найбільш проста, в її основі розвинений математичний апарат (реляційна алгебра або реляційне числення), тому вона фактично стала стандартною моделлю подання даних у СУБД.

Основними поняттями реляційних баз даних є тип даних, домен, атрибут, кортеж, первинний ключ і відношення.

1.4.4. Реляційні об'єкти даних

Існує спеціальна термінологія, прийнята в теорії реляційних баз даних [9; 26] (табл. 1.4).

Таблиця 1.4

Основні компоненти реляційного відношення

Відношення	Ціле	Рядок		Ціле		**Типи даних**
	Номер	Ім'я	Посада	Гроші		**Домени**
	№ п/п	Ім'я	Посада	Оклад	Премія	**Атрибути**
	1	Акіншин	Доцент	500	100	**Кортежі**
	2	Володько	Викладач	400	90	
	3	Сєчин	Професор	600	100	

Відношення – це таблиця, що відповідає певним властивостям. Відношення характеризується такими поняттями:

атрибут – це стовпець цієї таблиці. Зазвичай характеризує властивості об'єктів, відомості про які зберігаються в таблиці, у конкретних СУБД атрибути часто називають *полями*;

первинний ключ – це атрибут (чи множина атрибутів), значення якого унікально ідентифікує кортежі (рядки) таблиці;

кортеж відповідає заповненому рядку таблиці, у конкретних СУБД кортежі називають записами;

ступінь відношення – кількість його атрибутів;

кардинальне число (потужність відношення) – кількість кортежів у відношенні у поточний момент часу;

домен – це загальна сукупність значень, з якої беруться конкретні значення для конкретного атрибуту.

Властивості відношень:

немає однакових кортежів. Це випливає з того, що тіло відношення визначене як математична множина кортежів. Множина, за визначенням, не містить однакових елементів. Як наслідок цієї властивості: у відношенні завжди існує первинний ключ;

кортежі не впорядковані. Це виходить також з того, що тіло відношення визначене як математична множина кортежів. Математична множина, за визначенням, не впорядкована. Саме тому у відношенні не існує таких понять, як "наступний", "попередній", "другий кортеж" і т. п.;

атрибути не впорядковані. Це випливає з того, що заголовок відношення визначений як математична множина атрибутів. Множина не впорядкована за визначенням, тобто знову немає понять "перший атрибут", "наступний атрибут" і т. п.;

усі значення атрибутів неподільні. Це наслідок того, що кожен атрибут визначений на своєму домені, а домен – множина неподільних скалярів.

1.4.5. Концепція непроцедурних мов програмування

Спочатку мови доступу до інформації створювались як прості мови запитів до реляційної бази даних [8], наближені до природної мови. Для цього часто використовувався термін very high level language (VHL-language) – мова високого рівня. Це надавало можливість формулювати нескладними мовними конструкціями намір виконати якісь операції з базою даних.

Ефективність СУБД визначається наявністю і зручністю використання таких *операцій*:

традиційні операції над множинами (об'єднання, перетин, віднімання, декартовій твір, поділ);

спеціальні реляційні операції проекції, з'єднання і вибору.

Для виконання операцій над відносинами в реляційних СУБД можуть використовуватися такі мови:

мови реляційної алгебри (записуючи послідовності операцій над відносинами у відповідному порядку, можна отримати бажаний результат, тому мови реляційної алгебри є процедурними);

мови реляційного числення (непроцедурного) засновані на класичному численні предикатів і надають користувачу набір правил для запису запитів до бази даних, в яких міститься інформація про бажаний результат.

За своєю сутністю SQL є непроцедурною мовою, що використовується для керування даними реляційних СУБД. Термін "непроцедурна" означає, що цією мовою можна сформулювати завдання на обробку даних, але не можна проінструктувати, як саме це слід зробити. Іншими словами, в цій мові відсутні алгоритмічні конструкції – такі, як: мітки, оператори циклу, умовні переходи та ін. Запити, синтаксис яких досить простий, вводилися прямо з консолі і послідовно виконувалися.

Незважаючи на свою непроцедурність, мова SQL так і не стала інструментом для роботи з базами даних для різних категорій користувачів, далеких від інформаційних технологій. На практиці з базою даних зазвичай працюють за допомогою застосувань, написаних програмістами процедурними мовами (наприклад C++, C#, Visual Basic, Pascal, Java та ін). Вони мають зручний інтерфейс і не змушують користувача безпосередньо вводити запити мовою SQL. Ці запити зазвичай формуються усередині програми, здійснюється обмін з базою даних, і отримана інформація у зручному вигляді подається на екрані. Як правило у результаті виконання запиту мовою SQL СКБД повертає таблицю, що містить потрібні дані. Якщо у базі даних не виявилося даних, що відповідають запиту, то повертається порожня таблиця.

Кожний запит у мові SQL подається у вигляді спеціальної команди, яка може складатися з декількох речень. Поступово, у міру вивчення матеріалу, кожна з таких команд буде детально розглянута та пояснена.

Заради справедливості необхідно зазначити, що останні версії SQL підтримують оператори керування обчисленнями, властиві процедурним мовам, оператори умовного переходу та циклу. Практично кожна СУБД має у своєму складі такі "процедурні" розширення, як мова PL/SQL у СУБД Oracle [31]. Тому сучасну SQL важко назвати суто непроцедурною або декларативною мовою.

Висновки

Бази даних використовують в усіх сферах діяльності людини, що пов'язані з комп'ютерною обробкою інформації. Тому у сучасній IT-індустрії поняття бази даних є одним з найважливіших понять.

Більшість сучасних СУБД ґрунтуються на реляційній моделі даних, заснованій на спеціальному математичному апараті – теорії відношень. Спеціальна мова спілкування з базами даних на основі цього математичного апарату дістала назву SQL. Ця мова у своєму розвитку пройшла декілька етапів, є стандартизованою та підтримується усіма комерційними СУБД. Унаслідок цього практично неможливо працювати з базами даних не знаючи мови SQL. Однак слід відмітити той факт, що, незважаючи на стандарт, майже у кожній комерційній СУБД є певні відмінності від нього, і це треба мати на увазі у процесі роботи з конкретною системою.

Запитання для самодіагностування

1. Що таке база даних, чим вона відрізняється від банку даних?
2. Які існують рівні подання інформації в базах даних?
3. Що таке інформаційна система?
4. Що таке СКБД і які існують види та функції СУБД?
5. Що таке система баз даних і які існують її компоненти?
6. Що таке модель даних і які існують моделі організації даних?
7. Які існують реляційні об'єкти даних?
8. Хто є автором реляційної моделі даних?
9. Що таке SQL?
10. Де використовують мову SQL?

Розділ 2. Загальна характеристика мовних засобів спілкування зі СУБД. Мова DDL SQL

2.1. Мовні засоби спілкування зі СУБД

2.1.1. Характеристика мовних засобів спілкування зі СУБД

Мовні засоби призначені для спілкування користувачів різних класів із банками даних. Мовні засоби слугують для опису різних компонентів банку даних, а іноді – і зовнішніх відносно до банку елементів, що знаходяться з ним у безпосередній взаємодії, а також для звернення до потрібних частин банку даних.

Функціональні можливості підтримуваної засобами СУБД моделі даних стають доступними для кінцевих користувачів, розробників додатків, системного персоналу адміністрування даними завдяки втіленню її у вигляді комплексу мовних засобів, які підтримує дана СУБД. В існуючих СУБД застосовуються різні способи реалізації таких мов.

Для кінцевих користувачів функції зазначених мов найчастіше доступні в неявній формі завдяки їх реалізації за принципом так званих мов четвертого покоління (4GL). Це призначений для користувача інтерфейс, що включає: різного роду меню, діалогові сценарії, які заповнюються користувачем, екранні форми та інші дружні засоби подання запитів.

Найбільш відомими з мов цього типу є:

Smalltalking – малий розмовний;

QBE (Query By Example) – програмування на прикладі;

Fort, який знаходить застосування у вирішенні складних завдань імітаційного моделювання, в графічних системах і т. п.

Мовні засоби СУБД призначені насамперед для розроблення прикладних програм розв'язання задач економічного управління, інформація для яких зберігається і підтримується за допомогою баз даних.

Мовні засоби СУБД необхідні для описування даних, організації спілкування, виконання процедур пошуку й різних перетворень з даними. Класифікація мовних засобів баз даних розроблена американським комітетом КОДАС1Л [17] з проектування і створення бази даних (рис. 2.1).

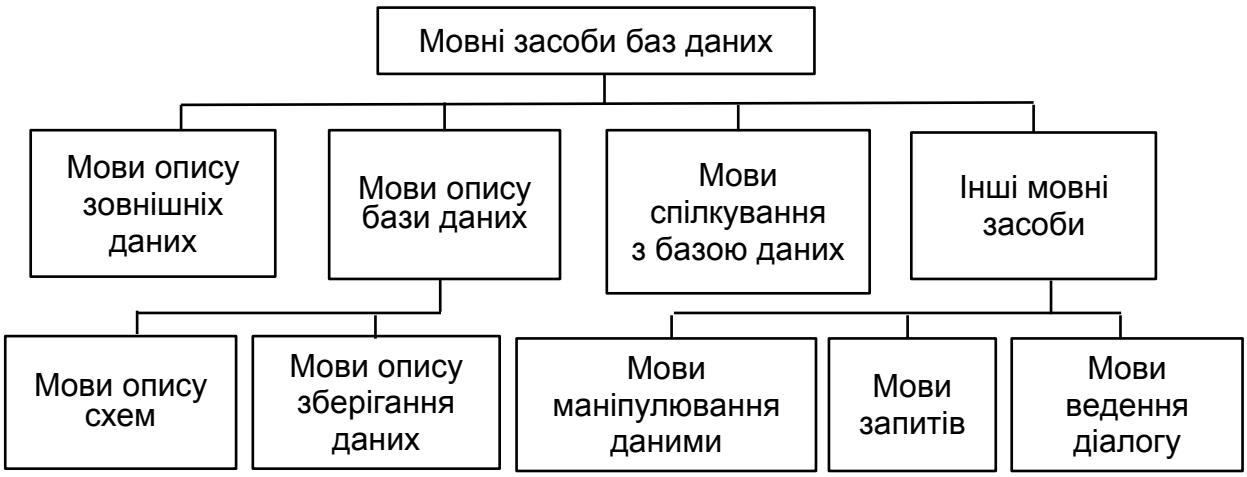


Рис. 2.1. **Класифікація мовних засобів баз даних**

За функціональними можливостями виділяють такі **категорії мов**:

мови, що мають тільки можливості запитів. Вони забезпечують виведення необхідних даних на екран або друкування в потрібному форматі. Сьогодні використовуються рідко;

комплексні мови запитів/оновлень. Це більш розвинені мови, вони дозволяють формулювати складні запити, які належать до окремих взаємопов'язаних записів, і оновлювати дані так само легко, як і формулювати запити. Використовуючи їх, користувачі можуть створювати свої власні файли;

генератори звітів, дають можливість вибирати потрібні дані з файлів або баз даних і форматувати їх у вигляді необхідних форм документів;

графічні мови. Використання графічних засобів постійно розширюється. З їх допомогою можна виводити дані у вигляді різних графіків і діаграм, а також використовувати інші образотворчі можливості. Як і генератори звітів, графічні мови дозволяють здійснювати відбір інформації з файлів або баз даних за різними критеріями, а також виконувати арифметичні та логічні маніпуляції з даними;

інструментальні засоби підтримки рішень. Мови цього типу призначені для створення систем прийняття рішень. Це можуть бути системи типу "що – якщо", системи, що виконують часовий або трендовий аналіз тощо. Можливе використання як універсальних, так і проблемно-орієнтованих засобів;

генератори додатків забезпечують можливість опису непроцедурного додатка шляхом обробки інформації і подальшої автоматичної генерації програм;

машинно-орієнтовані мови специфікацій фактично є генераторами додатків з подальшим їх розвитком. На відміну від генераторів додатків мови специфікацій більш універсальні та дозволяють специфікувати додатки різних типів;

мови дуже високого рівня. У більшості випадків додатки будуються за допомогою непроцедурних мов, однак деякі мови є процедурними (наприклад, NOMAD), але програмування на них значно коротше, ніж, наприклад, на Cobol;

параметризовані пакети прикладних програм (ППП). Ця категорія програмних засобів відома давно. "IV покоління" належить до таких ППП, які допускають легку модифікацію самого пакета, дозволяють користувачам генерувати власні звіти, запити до бази даних тощо;

мови додатків. Більшість мов четвертого покоління є універсальними, інші – спроектовані для специфічних додатків. Прикладами таких мов є мови для управління фінансами, управління роботою верстатів з програмним управлінням.

Мовні засоби більшості сучасних СУБД належать до мов четвертого покоління. До першого покоління відносять машинні мови, до другого – символічні мови асемблера, до третього – алгоритмічні мови типу PL, Cobol і т. п., які в 1960-х рр. називалися мовами високого рівня, але рівень, яких набагато нижче, ніж у мов четвертого покоління. Є ще мови п'ятого покоління, до яких відносять мови систем штучного інтелекту (наприклад, Prolog).

Мови четвертого покоління створювалися за принципом: "Люди коштують дорожче, ніж машини". При їх проектуванні використовувалися такі **принципи**:

принцип мінімуму роботи: мова повинна забезпечити мінімум зусиль, щоб "примусити" машину працювати;

принцип мінімуму майстерності: робота повинна бути настільки проста, наскільки це можливо; вона не повинна бути долею обраних і бути зрозумілою лише присвяченим;

принцип природності мови: скасування "стороннього" синтаксису та мнемоніки. Мова не повинна вимагати від користувачів значних зусиль у вивченні синтаксису або містити багато мнемонічних або інших позначень, які швидко забуваються;

принцип мінімуму часу: мова повинна дозволяти без суттєвої затримки реалізовувати виниклі потреби в доступі до інформації і її обробці;

принцип мінімуму помилок: технологія повинна бути спроектована таким чином, щоб мінімізувати помилки людини; а, якщо вони виникли, то, за можливістю, "виловити" їх автоматично;

принцип мінімуму підтримки: механізм мови повинен дозволяти легко вносити зміни в існуючі програми;

принцип максимуму результату: мова надає користувачам потужний інструмент для вирішення різноманітних завдань.

На рис. 2.2 наведені компоненти мови четвертого покоління. Як бачимо, тут подані всі основні "генератори", наявність яких вже є традиційною для СУБД різних класів.

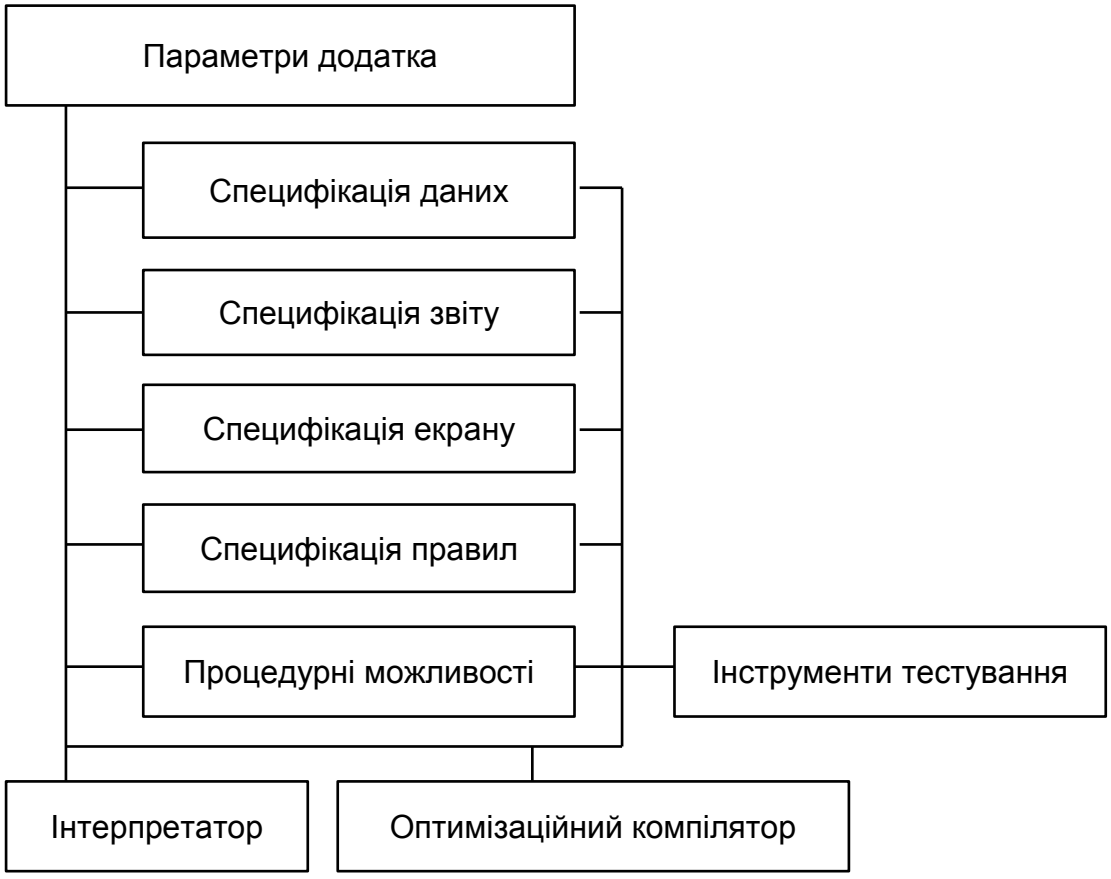


Рис. 2.2. **Компоненти мови IV покоління**

Часто трапляється, що в рамках однієї СУБД для тих самих цілей можуть використовуватися мови різних типів. Так, наприклад, у багатьох СУБД (dBase, FoxPro та ін.) для маніпулювання даними можуть використовуватися:

таблична мова запитів типу QBE;

мова SQL – аналітична мова запитів, що належить до класу мов обчислення кортежів;

процедурна мова програмування (для систем dBase, FoxPro). Це мова xBase, частина операторів якого реалізує операції реляційної алгебри, а інша частина, більш значна за кількістю операторів і функцій, виконує нереляційні операції.

2.1.2. Історія створення та особливості мови Query-By-Example (QBE)

Мова Query-By-Example – "Запит за зразком" (QBE) заснована на обчисленні предикатів на доменах. Мова була розроблена М. Злуффом у 1974 – 1975 рр. (фірма IBM) [19] на базі основної роботи Е. Ф. Кодда з реляційної алгебри (1970 р.).

З метою підвищення ефективності обробки даних, що зберігаються в БД, застосовуються запити, які дозволяють виконувати множинну обробку даних (одночасне введення, редагування і видалення безлічі записів, а також вибір даних з таблиць).

Запит – вимога, описана спеціальним чином, яка визначає склад операцій вибірки, видалення або зміни збережуваних даних, що виконуються над БД. Для виконання запитів засобами різних СУБД найчастіше використовують дві основні мови опису запитів: QBE (Query By Example) – *мова запитів за зразком*; SQL (Structured Query Language) – *структурована мова запитів*. Головною відмінністю між цими мовами є спосіб формування запитів: мова QBE дозволяє формувати запит вручну або візуально, а SQL програмує запит.

Теоретичну основу мови QBE становить реляційне числення доменів. За допомогою мови QBE можна створювати складні запити до бази даних, заповнюючи пропоновану СУБД запитальну форму. Даний спосіб створення запитів дозволяє отримати високу наочність і не вимагає вказувати алгоритм виконання операції. Кожна сучасна реляційна СУБД містить свій варіант QBE.

Мова QBE дозволяє створювати одно- та багатотабличні запити, які вибирають або обробляють дані з більш ніж однієї пов'язаної таблиці. Запити мовою QBE дозволяють виконувати основні операції: вибірки даних; обчислення над даними; вставки нових записів; видалення записів;

зміни даних. У результаті виконання запиту створюється нова таблиця – відповідна або оновлена вихідна таблиця. Усі операції, крім обчислень, можна здійснювати безумовно або за умовами, які задаються логічними виразами. Обчислення над даними виконуються за допомогою арифметичних виразів і створюють в відповідних таблицях нові поля, які називають обчислюваними.

Запитальна форма створюється у вигляді таблиці з ім'ям і назвами полів, які збігаються, відповідно, з ім'ям і назвами полів використовуваної вихідної таблиці. Сучасні СУБД (наприклад, Visual FoxPro, Access) багато дій зі створення запитів мовою QBE дозволяють виконувати візуально за допомогою миші.

Основними відмінностями мов QBE сучасних СУБД від мови, запропонованої М. Злуффом, є незначні зміни в інтерпретації окремих реляційних операцій, введення додаткових операцій і зміна форми подання мови. Наприклад, у системі Paradox операція друку замінена методом вибору в запитальній формі полів за допомогою прапорців, які розташовані на початку кожного її поля.

2.2. Стандарти мови SQL

2.2.1. Мова SQL, її історія та стандарти

На початку 1970-х рр. в одній з дослідницьких лабораторій компанії IBM була розроблена експериментальна реляційна система управління базами даних (СУБД) IBM System R. Згодом для неї була створена спеціальна мова SEQUEL, яка дозволяє відносно просто управляти даними в цій СУБД. Абревіатура SEQUEL розшифровувалася як Structured English QUEry Language – "Структурована Англійська Мова Запитів" [1]. Пізніше з юридичних міркувань мова SEQUEL була перейменована в SQL. У 1986 р., інститутом ANSI (American National Standards Institute) був прийнятий перший стандарт мови SQL. Метою розроблення було створення простої непроцедурної мови, якою міг скористатися будь-який користувач, який навіть не має навиків програмування.

SEQUEL була не єдиною мовою подібного призначення. У Каліфорнійському університеті Берклі була розроблена некомерційна СУБД Ingres (що була, між іншим, пращуром популярної зараз некомерційної

СУБД PostgreSQL). Це була реляційна СУБД, яка використовувала власну мову QUEL, проте не витримала конкуренцію з мовою SQL за кількістю підтримуючих його СУБД [2].

Першими СУБД, що підтримують нову мову, в 1979 р., стали Oracle V2 для машин VAX від компанії Relational Software Inc. (компанія Oracle, що згодом стала Oracle_Corporation) і System/38 від IBM, заснована на System/R.

До початку 80-х рр. XX ст. існувало декілька варіантів СУБД від різних виробників, причому кожний з них володів власною реалізацією мови запитів. Тому було ухвалене рішення розробити стандарт мови, яка гарантуватиме перенесення програмного забезпечення з однієї СУБД на іншу.

У 1983 році Міжнародна організація зі стандартизації (ISO) і Американський національний інститут стандартів (ANSI) приступили до розроблення стандарту мови SQL. Після безлічі консультацій і відхилення декількох попередніх варіантів у 1986 р. ANSI представив свою першу версію стандарту, який викладений в документі ANSI X3.135-1986 під назвою "Database Language SQL" ("Мова баз даних SQL"). Неофіційно цей стандарт SQL–86 отримав назву SQL1.

Відразу після завершення роботи над стандартом SQL1 у 1987 р. була розпочата робота над новою версією стандарту, який мав замінити стандарт SQL89. Йому була надана назва SQL2, оскільки дата ухвалення документа була невідома. Таким чином, SQL89 і SQL2 розроблялися паралельно. Нова версія стандарту була прийнята в 1992 р. на заміну стандарту SQL89. Новий стандарт SQL92 був розширенням стандарту SQL1. Він включив безліч доповнень до інструкцій з попередніх версій. SQL92 був розділений на кілька рівнів: початковий (entry), середній (intermediate), повний (full).

Мова SQL відповідає останнім стандартам SQL:2008, SQL:2003, SQL:1999 (і навіть SQL/92). Це дуже багата та складна мова, всі можливості якої важко відразу усвідомити і тим більше зрозуміти. Тому доводиться розбивати мову на рівні (або шари) – такі, що кожен рівень мови включає всі конструкції, що входять у нижчі рівні. У табл. 2.1 приведена хронологія історії версій стандарту SQL [19].

Сьогодні діє стандарт, прийнятий в 2003 р. (SQL:2003) з невеликими модифікаціями, внесеними в 2008 р.

Таблиця 2.1

Стандарти SQL

Роки	Назва	Інша назва	Зміни
1986	SQL-86	SQL-87	Перший варіант стандарту, який прийнятий інститутом ANSI і схвалений ISO в 1987 р.
1989	SQL-89	FIPS 127-1	Допрацьований варіант попереднього стандарту
1992	SQL-92	SQL2, FIPS 127-2	Значні зміни (ISO 9075). Рівень Entry Level стандарту SQL-92 був прийнятий як стандарт FIPS 127-2
1999	SQL:1999	SQL3	Додана підтримка регулярних виразів, рекурсивних запитів, підтримка тригерів, базові процедурні розширення, нескалярні типи даних і деякі об'єктно-орієнтовані можливості
2003	SQL:2003		Введені розширення для роботи з XML-даними, віконні функції (вживані для роботи з OLAP-базами даних), генератори послідовностей і засновані на них типи даних
2006	SQL:2006		Функціональність роботи з XML-даними значно розширена. З'явилася можливість спільно використовувати в запитах SQL і XQuery

2.2.2. Особливості та призначення команд SQL

Можна виділити три основні форми SQL: інтерактивну (Interactive), статичну (Static) і динамічну (Dynamic). Функціонують вони однаково, але використовуються по-різному.

Інтерактивний SQL застосовується для безпосередньої роботи з БД. Користувач вводить SQL-команду; вона відразу ж виконується і користувач бачить результат виконання (чи код помилки).

Статичний SQL містить SQL-команди, жорстко закодовані в тілі застосування, що виконується. Найбільш поширений вбудований SQL (Embedded SQL), де SQL-код включений в початковий текст програми, написаної іншою мовою (наприклад C++ або Pascal). Під час використання вбудованого SQL результати виконання команд SQL перенаправляються у змінні, якими оперує базова програма. SQL вбудований у більшість сучасних мов програмування: Ada, Cobol, Fortran, C++, Pascal, Java, Mumps тощо.

Динамічний SQL також є частиною застосування, але конкретний SQL-код генерується під час виконання (Run Time), а не вводиться заздалегідь. Зазвичай це зустрічається у роботі інтерактивних застосувань, коли оператор чи користувач програми вводить у діалоговому режимі деякі дані, значення яких у подальшому програмно обробляються, формуючи запит до бази даних.

Окрім вибірки та модифікації даних у базі, SQL дозволяє виконувати всі необхідні дії зі створення, модифікації та забезпечення безпеки баз даних. У зв'язку з цим усі команди SQL підрозділяють на **основні групи**:

Data Definition Language (DDL) – мова визначення даних. Призначена для створення, модифікації та видалення таблиць і всієї бази даних;

Data Manipulation Language (DML) – мова маніпулювання даними. Призначена для виконання основних операцій під час роботи з даними;

Data Control Language (DCL) – мова управління даними. Призначена для забезпечення захисту бази даних; застосовується для здійснення адміністративних функцій, що надають або скасовують право (привілей) використовувати базу даних, таблиці у базі даних, а також виконувати ті або інші оператори SQL.

Крім того, часто виділяють **додаткові групи команд**:

Transaction Control Language (TCL) – спеціальні оператори. Застосовуються для управління модифікацією бази даних, що здійснюють оператори DML (команди управління транзакціями);

Cursor Control Language (CCL) – оператори CCL. Використовують для визначення курсора, підготовки SQL-виразу для виконання, а також для деяких інших операторів [24]. Курсором зазвичай є результівний набір даних, отриманий під час виконання запиту, та пов'язаний з ним покажчик поточного запису. Ці оператори найчастіше використовують у процедурних розширеннях мови SQL.

Кожна *команда* (також зустрічаються терміни – оператор або інструкція) SQL розпочинається зі спеціального **ключового слова, яке визначає, що саме робить ця команда** (СТВОРЮЄ ТАБЛИЦЮ, ВИБИРАЄ ДАНІ, НАДАЄ ПРАВА ДОСТУПУ і т. п.). У команді містяться також *конструкції*, що вказують, над якими даними виконуються ці операції. Кожна конструкція (речення) також розпочинається з *ключових слів* – таких, як FROM, WHERE та ін.

Структура речення залежить від його типу – деякі речення містять імена таблиць або полів; деякі можуть містити додаткові ключові слова, константи або вирази.

Усі інструкції SQL мають схожу структуру, яку схематизовано на рис. 2.3.

Рис. 2.3. **Загальна структура команди SQL**

З переліком зарезервованих слів у мові SQL, згідно із стандарту SQL2, можна ознайомитися у роботі [32].

2.2.3. Імена об'єктів у базі даних

У кожного об'єкта в базі даних є **унікальне ім'я** [8]. Імена використовуються в командах SQL і вказують, над яким об'єктом бази даних команда повинна виконати дію. Основними іменованими об'єктами в реляційній базі є таблиці, стовпці та користувачі. Правила їхнього іменування були визначені ще в стандарті SQL1.

У стандарті SQL2 цей перелік був значно розширений і став включати схеми (колекції таблиць), обмеження (обмежувальні умови, що накладаються на вміст таблиць та їхні відношення), домени (допустимі набори значень, які можуть бути занесені в стовпець) і ряд інших об'єктів.

У багатьох СУБД існують додаткові види іменованих об'єктів, наприклад: збережені процедури, відношення "первинний ключ" – "зовнішній ключ" та ін.

Відповідно до стандарту ANSI/ISO імена в SQL повинні містити від одного до вісімнадцяти символів, розпочинатися з букви та не містити пробілів або спеціальних символів пунктуації. У стандарті SQL2 максимальне число символів в імені збільшене до 128 [9]. Але на практиці підтримка імен у різних СУБД реалізована по-різному. Тому для підви-

щення мобільності доцільно робити імена порівняно короткими й уникати вживання в них спеціальних символів.

Якщо в інструкції вказано ім'я таблиці, СУБД вважає, що відбувається звернення до однієї з власних таблиць користувача, який зараз працює з базою даних.

Більшість СУБД дозволяють різним користувачам створювати таблиці з однаковими іменами. Маючи відповідний дозвіл, можна звертатися до таблиць, власниками яких є інші користувачі, за допомогою повного імені таблиці. Воно складається з імені власника таблиці та її імені, розділених крапкою. Наприклад, повне ім'я таблиці "ТОВАР", власником якої є користувач на ім'я ВИРОБНИК, має такий вигляд: ВИРОБНИК.ТОВАР. Повне ім'я таблиці можна використати замість короткого імені в усіх інструкціях SQL.

Стандарт SQL2 повніше узагальнює поняття повного імені таблиці. Він дозволяє створювати іменовану множину таблиць, що називається **схемою** [8]. Для доступу до таблиці в схемі також застосовується повне ім'я. Наприклад, звернення до таблиці "СТУДЕНТИ", що розміщується у схемі УНІВЕРСИТЕТ, формується так:

УНІВЕРСИТЕТ.СТУДЕНТИ

Якщо в інструкції задається ім'я стовпця (атрибуту), СУБД сама визначає, в якій із вказаних в інструкції таблиць міститься стовпець.

Якщо в інструкції є два стовпці з різних таблиць, але з однаковими іменами, необхідно вказати повні імена стовпців, які однозначно визначають їх місцезнаходження.

Повне ім'я стовпця складається з імені таблиці, що містить стовпець, та імені стовпця (короткого імені), розділених крапкою. Наприклад, повне ім'я стовпця "ОПТОВА_ЦІНА" з таблиці "ТОВАР" має такий вигляд:

ТОВАР.ОПТОВА_ЦІНА

Якщо стовпець знаходиться в таблиці, власником якої є інший користувач, то в повному імені стовпця слід указати повне ім'я таблиці. Наприклад, повне ім'я стовпця "ОПТОВА_ЦІНА" у таблиці "ТОВАР", власником якої є користувач ВИРОБНИК, має такий вигляд:

ВИРОБНИК.ТОВАР.ОПТОВА_ЦІНА

Іменування об'єктів в базі даних слід виконувати відповідно до назв, що використовуються в предметній області.

2.2.4. Типи даних

Одним із найважливіших понять, яким доводиться оперувати в усіх мовах програмування (і в мові SQL зокрема), є поняття типу даних [8]. Тип даних є фундаментальним поняттям у програмуванні.

Тип даних – це множина значень і набір операцій, які можна застосовувати до таких значень. Від типу даних залежить спосіб зберігання значень і виконання операцій над ними. Будь-які дані, якими оперують програми, належать до певних типів.

У стандарті SQL1 описано лише мінімальний набір типів даних, які можна використати для подання інформації в реляційній базі даних. Вони практично підтримуються в усіх комерційних СУБД. Стандарт SQL2 додав у цей перелік рядки змінної довжини, значення дати та часу та ін.

Сучасні СУБД дозволяють обробляти дані найрізноманітніших типів, серед яких найбільш поширеними є такі: цілі числа, десяткові числа (числа, що мають дробову частину для точних обчислень), числа з плаваючою комою, рядки символів постійної і змінної довжини, грошові величини, дата і час, логічні величини, довгий текст (зазвичай рядки завдовжки до 32 000 або 65 000 символів), неструктуровані потоки байтів (для зберігання графіки, відеозображень, виконуваних файлів тощо).

До базових типів даних, що використовуються у SQL, відносять такі [3; 8]:

CHARACTER(n), або CHAR(n), – символьні рядки постійної довжини з n символів. Для задавання цього типу під кожне значення завжди відводиться n символів. Якщо реальне значення містить менше n символів, то СУБД автоматично доповнює відсутні символи пробілами;

NUMERIC[(n, m)] – точні числа, де n – загальна кількість цифр у числі, m – кількість цифр зліва від коми;

DECIMAL[(n, m)] – точні числа (аналог NUMERIC), де n – загальна кількість цифр у числі, m – кількість цифр зліва від коми;

DEC[(n, m)] – те ж саме, що DECIMAL[(n, m)];

INTEGER, або INT, – цілі числа;

SMALLINT – цілі числа меншого діапазону;

FLOAT[(n)] – числа великої точності, що зберігаються у формі з плаваючою комою, де n – число байтів, що резервується для зберігання одного числа. Діапазон чисел визначається конкретною реалізацією;

REAL – дійсний тип чисел, який відповідає числам з плаваючою комою меншої точності, ніж FLOAT;

DOUBLE PRECISION – дійсний тип чисел з визначеною в реалізації точністю, яка більша, ніж для REAL.

У стандарті SQL92 додані такі типи даних [3; 8]:

VARCHAR(n) – рядки символів змінної довжини до n символів включно;

NCHAR(N) – рядки локалізованих символів постійної довжини з n символів;

NCHAR VARYING(n) – рядки локалізованих символів змінної довжини;

BIT(n) – рядок бітів постійної довжини;

BIT VARYING(n) – рядок бітів змінної довжини;

DATE – календарна дата;

TIMESTAMP (точність) – дата та час;

INTERVAL – часовий інтервал.

Багато сучасних СУБД також підтримують спеціальні поля для зберігання дуже великих за розміром об'єктів:

BINARY LARGE OBJECT(n) (BLOB) – тип даних, що дозволяє зберігати великі неструктуровані бінарні об'єкти, наприклад звук і відео;

CHARACTER LARGE OBJECT(n) (CLOB) – тип даних, що дозволяє зберігати великі символьні об'єкти.

Незважаючи на наявність стандартів, відмінності в підтримці типів даних у різних СКБД істотно перешкоджають мобільності застосувань, в яких використовується SQL. Причини подібних відмінностей слід шукати в самому шляху, яким розвивалися реляційні бази даних.

2.2.5. Константи

У багатьох командах SQL необхідно явно вказувати значення чисел, рядків або дат.

Явні значення величин називають **константами**. Константи практично завжди використовуються в командах для введення нових записів

у бази даних або задають умови відбору записів у пошукових операціях [3; 34]. Наприклад, у командах для визначення назви товару, його оптової ціни, а також дати реалізації товару в таблиці "ТОВАР" три константи можна записати в такому вигляді:

```
..... VALUES(ХЛІБ, 12,  '01/12/2017')
```

У фрагменті команди:

```
…… WHERE ОПТОВА_ЦІНА = 12
```

константа використовується для задавання умови відбору товарів, які мають оптову ціну тільки 12 грн.

Для числових типів даних визначені константи у вигляді послідовності цифр з необов'язковим зазначенням знака числа та десятковою комою. Тобто правильними будуть константи:

1213.23 12.2716 15451.7

Константи з плаваючою комою задаються, як і у більшості мов програмування, шляхом задання мантиси та порядку, розділених символом E, наприклад:

12.9E-2 – 21.12E7 0.123E09

Рядкові константи потрібно взяти в одинарні лапки:

'Булка_Городська' 'Харків'

У деяких СУБД, наприклад Access, можна використати подвійні лапки в рядкових константах:

"Хліб_Бородинський" "Тарасов О. В."

Проте використання подвійних лапок може викликати певні складнощі під час перенесення застосувань на іншу платформу, тому рекомендується уникати такого подання символьних констант.

Константи дати, часу та часового інтервалу в реляційних СУБД також подають у вигляді рядкових констант. Їхні формати часто відрізняються у різних СУБД. Крім того, формат подання дати різний у різних країнах. Тому в більшості СУБД реалізовані способи налаштування форматів подання дат або спеціальні функції перетворення форматів дат.

Окрім констант користувача, в СУБД можуть існувати і спеціальні системні константи. Стандарт SQL1 визначає тільки одну системну константу USER, яка відповідає імені користувача, під яким він приєднався до бази даних [13; 34].

В операторах SQL можна використовувати вирази, які побудовані за стандартними правилами застосування знаків арифметичних операцій додавання (+), віднімання (−), множення (*) і ділення (/).

До стандарту SQL2 включено можливість виконання операцій додавання та віднімання над датами. У більшості СУБД також визначена операція конкатенації над рядковими даними, хоча позначається вона по-різному. Так, наприклад, для Oracle та DB2 операція конкатенації позначається подвійною вертикальною рискою (||), а в MS SQL Server – знаком плюс (+).

2.3. Мова DDL SQL

2.3.1. Особливості команд мови визначення даних SQL (Data Definition Language (DDL) SQL)

Вирази мови визначення даних – Data Definition Language (DDL), в ANSI називають мовою опису схем – Shema Definition Language. Мова складається з команд, що створюють об'єкти в базі даних (таблиці, подання, індекси). DDL-вирази використовують також для визначення характеристик бази даних, їх зміни та видалення.

DDL базується на трьох **командах SQL**:

CREATE (створити) – дозволяє визначати та створювати об'єкти бази даних;

DROP (видалити) – дозволяє видаляти існуючий об'єкт бази даних;

ALTER (змінити) – дозволяє змінювати визначення об'єкта бази даних.

Основні команди DDL мови SQL указані в табл. 2.2.

Таблиця 2.2

Основні команди DDL мови SQL

Оператори	Опис
CREATE DATABASE	Створює нову базу даних
DROP DATABASE	Видаляє базу даних
CREATE TABLE	Додає нову таблицю до бази даних
DROP TABLE	Видаляє таблицю з бази даних
ALTER TABLE	Модифікує (змінює) структуру наявної таблиці
CREATE VIEW	Додає нове подання до бази даних
DROP VIEW	Видаляє подання з бази даних
CREATE INDEX	Створює індекс для таблиці
DROP INDEX	Видаляє існуючий індекс
CREATE SCHEMA	Створює нову схему у базі даних
DROP SCHEMA	Видаляє схему з бази даних
CREATE DOMAIN	Створює новий домен у базі даних
ALTER DOMAIN	Перевизначає домен
DROP DOMAIN	Видаляє домен з бази даних

Таким чином, за допомогою операторів DDL можна проводити такі операції: створити нову базу даних; сформувати структуру нової таблиці;

змінити визначення існуючої таблиці; видалити існуючу таблицю; визначити подання даних; забезпечити умови безпеки бази даних; створити індекси для доступу до таблиць; управляти розміщенням даних на пристроях зберігання.

2.3.2. Команди створення та видалення бази даних CREATE/DROP DATABASE

Для створення нової бази даних засобами мови SQL слід використовувати команду:

```
CREATE DATABASE Ім'я_БД [Параметри]
```

Перелік і можливі значення параметрів залежать від конкретної СУБД і найчастіше описують фізичні параметри бази даних, засоби управління безпекою, мовні параметри і т. п. Зокрема, в якості параметрів може бути вказано ім'я бази даних і шлях до файла (типу .mdf для СУБД SQL Server):

```
CREATE DATABASE ім'я_бази_даних ON (NAME = логічне_ім'я_файла,
                FILENAME = шлях_і_фізичне_ім'я_файла)
```

Наприклад:

```
 CREATE DATABASE Товари On (NAME = Товари,
                      FILENAME = 'F: \ fvv \ DB \ Товари.mdf')
```

Аналогічно записується команда видалення бази даних. У наступних прикладах спочатку видаляється база даних з ім'ям DEPARTMENT, а потім дві бази з іменами AVTORYNOK, SHOP відповідно:

```
 DROP DATABASE DEPARTMENT
 DROP DATABASE AVTORYNOK, SHOP
```

Рекомендується створювати базу даних і налаштовувати її параметри засобами спеціальних утиліт, які є в конкретній промисловій СУБД.

2.3.3. Команди створення, модифікації та видалення таблиць у базі даних CREATE/ALTER/DROP TABLE

У загальному випадку (наприклад в MS SQL Server) існує **три способи створення об'єктів бази даних**:

за допомогою засобів мови SQL;

із застосуванням інструментальних засобів інтегрованого середовища (в Management Studio сервер буде генерувати SQL-скрипт);

з використанням першого та другого способів.

Мова DDL, як видно з назви, застосовується для визначення таблиць, полів, індексів та інших об'єктів, які розташовуються в СУБД і, як правило, зберігають дані або полегшують їх обробку.

Таблиці бази даних створюються за допомогою команди CREATE TABLE.

Ця команда:

створює порожню таблицю (яка не має рядків);

визначає ім'я таблиці та безліч названих стовпців у зазначеному порядку;

для кожного стовпця визначає тип даних і розмір;

для кожного стовпця визначає різні обмеження (PRIMARY KEY, FOREIGN KEY, Not Null, Unique, Default, Check, Constraint).

Синтаксис оператора CREATE TABLE має такий вигляд:

```
CREATE TABLE ім'я_таблиці (
поле1 тип1 [NOT NULL] [UNIQUE] [PRIMARY KEY],
поле2 тип2 [NOT NULL] [UNIQUE],
. . .
полеN типN [NOT NULL] [UNIQUE],
[FOREIGN KEY (поле) REFERENCES таблиця (поле)
[ON DELETE {No Action | CASCADE | SET NULL | Set Default}]
[ON UPDATE {No Action | CASCADE | SET NULL | Set Default}]]
)
```

Оскільки проміжки використовують для поділу частин команди SQL, вони не можуть бути частиною імені стовпця або таблиці (або будь-якого іншого об'єкта бази даних). В операторі CREATE TABLE визначення полів перераховуються через "кому".

Приклад 2.1. Створення таблиці "Товари" ("Код_товару" – номер товару, "Товар" – назва товару, "Ціна" – ціна товару, "Виробник" – виробник товару, "Дата" – дата реалізації товару):

```
CREATE TABLE Товари
(Код_товару SMALLINT IDENTITY NOT NULL PRIMARY KEY,
Товар NVARCHAR (18),
Ціна NUMERIC (6, 2),
Виробник  NVARCHAR(15),
Дата SMALLDATETIME)
```

Коли створюється таблиця або змінюється її структура, можна встановлювати обмеження на значення, які можуть бути введені в поля. Якщо це виконано, то СУБД буде відкидати будь-які значення, що порушують обумовлені критерії.

Є такі **типи обмежень цілісності бази даних**:

обмеження поля (атрибута) (застосовується тільки до певного поля);

обмеження таблиці (для груп з одного та більше полів);

обмеження посилальної цілісності (для визначення зовнішнього ключа).

Обмеження на рівні поля:

невизначене значення NOT NULL – заборона невизначених значень (NULL – це спеціальний маркер, що позначає, що поле порожнє або невизначене. Значення NULL вводиться за замовчуванням);

унікальність стовпчика UNIQUE (застосовується тільки разом з обмеженням NOT NULL);

первинний ключ PRIMARY KEY, що еквівалентно NOT NULL UNIQUE (якщо ключ складається з кількох атрибутів, використовується інше визначення);

зовнішній ключ (обмеження стовпця за посиланням) FOREIGN KEY;

перевірка на допустимість значення CHECK;

задане за замовчуванням значення DEFAULT.

Ці обмеження іноді називають обмеженнями семантичної цілісності – для можливості смислового опису даних.

Приклад 2.2. Невизначене значення не може бути встановлено для поля "Товар":

```
CREATE TABLE Товар
(...
Товар NVARCHAR (18) NOT NULL,
...)
```

Порядок розташування полів у таблиці визначається послідовністю, в якій вони вказані в команді створення таблиці.

Приклад 2.3. Необхідно гарантувати, що всі значення, введені в поле NAZVA, відрізняються один від одного. СУБД відхилить будь-яку спробу введення в це поле значення, яке вже є в іншому рядку. Це обмеження застосовується до полів, де раніше було оголошено обмеження NOT NULL. Якщо таке поле не є первинним ключем, то воно буде альтернативним ключем:

```
CREATE TABLE TOVAR
(…
NAZVA NVARCHAR (18) NOT NULL UNIQUE,
…)
```

Приклад 2.4. Необхідно оголосити комбінацію з двох полів унікальною (обмеження таблиці) з присвоєнням цього обмеження таблиці унікального імені:

```
CREATE TABLE Товари
(…
Товар NVARCHAR (18) NOT NULL,
Виробник NVARCHAR (15) NOT NULL,
UNIQUE (Товар, Виробник))
```

Або так:

```
CONSTRAINT UN_Товар_Виробник UNIQUE (Товар, Виробник)
```

Шляхом обмеження таблиць можна привласнювати унікальні імена. Перевага явного задання імені обмеження полягає в тому, що після видачі системою повідомлення про порушення встановленого обмеження буде вказано його ім'я, що спрощує виявлення помилок. Для присвоєння імені обмеження використовується дещо змінений синтаксис команд CREATE TABLE й ALTER TABLE. У цьому запиті UN_Товар_Виробник – ім'я, що присвоєне обмеженню таблиці.

Приклад 2.5. Оголошення складного первинного ключа (обмеження таблиці). Будь-яке поле, яке використовується в обмеженні PRIMARY KEY, повинно вже бути оголошено NOT NULL. Первинний ключ складається з двох полів:

```
CREATE TABLE TOVAR
(…
NAZVA NVARCHAR(18) NOT NULL,
VYROBNYK NVARCHAR(15) NOT NULL,
…
PRIMARY KEY (NAZVA,  VYROBNYK))
```

Або так:

```
CONSTRAINT PK_TOVAR PRIMARY KEY(NAZVA, VYROBNYK)
```

Умова перевірки на допустимість значення: CHECK (<логічне вираження>).

Особливості роботи з обмеженням цілісності CHECK:

основні конструкції, які використовуються в CHECK (<,>, =,! =, And, Or, IN, Between, Like);

обмеження може стосуватися одного стовпця;

обмеження може бути поширене на всю таблицю. Тому перевірка значень в одному стовпці здійснюється на підставі значень іншого стовпця (за умови, що стовпці належать до однієї таблиці);

на підставі CHECK можна перевірити відповідність деякого поєднання значень стовпців заданому критерію;

правила складання CHECK аналогічні правилам складання конструкції WHERE;

перевірка CHECK дозволяє досягти більш високої продуктивності порівняно з перевірками за допомогою правил і тригерів.

Приклад 2.6. Введення обмежень на поле "Ціна":

```
CREATE TABLE Товар
      (…
            Ціна INT CHECK (Ціна <=75),
       …)
```

Або так:

```
Ціна INT CHECK IN (25, 35, 45, 75)    .
```

Або так:

```
Ціна INT CHECK (Ціна BETWEEN 25 AND 75) .
```

Або так:

```
CONSTRAINT CH_Ціна CHECK (Ціна <=75)...  .
```

Приклад 2.7. Введення обмежень на поле "Ціна" та "Дата" (обмеження таблиці):

```
CREATE TABLE Товар
(…
Ціна INT,
Дата SMALLDATETIME,
    CHECK (Ціна <75 AND Дата > 31/08/2017))
```

Або так:

```
CONSTRAINT CH_TOV CHECK (Ціна <5 AND Дата > 31/08/2017)
```

Значення за замовчуванням [DEFAULT {<значення> | USER | NULL}]. Тут ключове слово USER означає, що у ході заповнення колонки йому буде присвоєний символьний рядок, що містить ім'я поточного користувача. У таку таблицю можна ввести тільки ціни нижче 75 для дат пізніше 31/08/2017.

Значення DEFAULT – за замовчуванням визначає, що станеться, якщо не ввести будь-яке з явних допустимих значень. Використовувати значення за замовчуванням – це альтернатива для NULL.

Приклад 2.8. Введення обмежень на поле "Ціна" і завдання значення 75 за замовчуванням:

```
CREATE TABLE Товар
(... Ціна INT CHECK IN (25, 35, 45, 75) DEFAULT 75, ...)
```

Обмеження стовпця за посиланням означає оголошення поля зовнішнім ключем:

```
FOREIGN KEY (<поле>) REFERENCES <ім'я_ основної _таблиці>
 (<ім'я_первинного_ ключа_основної таблиці >)
[ON DELETE { No Action | CASCADE | SET NULL | Set Default }]
[ON UPDATE { No Action | CASCADE | SET NULL | Set Default }]
```

Приклад оголошення поля зовнішнього ключа:

```
CREATE TABLE Продажі (  …
Код_товару INT NOT NULL,
…
CONSTRAINT FK_ PROD FOREIGN KEY (Код_товару) REFERENCES
        Товари(Код_товару) ON DELETE CASCADE ON UPDATE CASCADE)
```

Особливості задання обмеження стовпця за посиланням:

основна таблиця повинна бути описана до підлеглої таблиці;

усі поля, використовувані в якості зовнішніх ключів, у батьківській таблиці повинні мати обмеження PRIMARY KEY або NOT NULL і UNIQUE;

для задання поля, яке є зовнішнім ключем, спочатку його потрібно визначити – описати найменування, тип даних та інші атрибути;

після задання всіх полів може слідувати одне або кілька визначень FOREIGN KEY, які задають поле зовнішнього ключа та визначають таблицю і поле, на яке посилається зовнішній ключ;

кілька визначень FOREIGN KEY перераховуються через кому.

Правило посилальної цілісності: зовнішній ключ не може посилатися на неіснуючий первинний ключ. Посилальна цілісність означає підтримку зовнішніх ключів з можливістю вибору одного з принципів видалення пов'язаних кортежів. Кортежі підлеглого відношення знищуються у ході видалення пов'язаних з ними кортежів основного відношення

(якщо потрібно видалити первинний ключ, то попередньо слід явно видалити всі записи підлеглих таблиць, зовнішні ключі яких посилаються на цей первинний ключ). Визначення ON DELETE CASCADE містить механізм каскадного видалення.

За наявності інструкції ON DELETE CASCADE СУБД автоматично видаляє усі записи, які посилаються на цей первинний ключ.

Якщо указана інструкція ON DELETE SET NULL – кортежі підлеглого відношення модифікуються в NULL з видаленням пов'язаних з ними кортежів основного відношення.

Якщо указана інструкція ON DELETE No Action – забороняється видалення кортежу основного відношення при наявності пов'язаних з ним кортежів підлеглого відношення.

Дуже важливим є розуміння того, що правило посилальної цілісності реалізовано не тільки в операціях роботи з даними, а й в операціях створення об'єктів, які ці дані зберігають.

Таким чином, поки не створена таблиця з первинним ключем, можна створити таблицю із зовнішнім ключем, який на нього посилається.

2.3.4. SQL-оператор зміни існуючої таблиці ALTER TABLE

Для модифікації структури та параметрів існуючої таблиці використовують команду ALTER TABLE. Синтаксис команди ALTER TABLE для додавання стовпців у таблицю має вигляд:

```
ALTER TABLE <ім'я таблиці>
ADD (<ім'я стовпця> <тип даних> <розмір>)
```

За цією командою для існуючих у таблиці рядків додають новий стовпець, в який заносять NULL-значення. Цей стовпець – останній в таблиці. Можна додавати декілька стовпців, тоді їх визначення в команді ALTER TABLE розділяють комою.

Оператор ALTER TABLE не діє, якщо таблиця повинна бути перевизначена, однак при розробленні бази даних не варто виключати необхідність цієї дії.

Приклад 2.9. Додавання до таблиці "Товари" поля для зберігання інформації про кількість товару:

```
ALTER TABLE Товари ADD Кількість INT
```

Приклад 2.10. Видалення з таблиці "Товари" поля "Кількість":

```
ALTER TABLE Товари DROP COLUMN Кількість
```

Спрощена версія синтаксису оператора ALTER TABLE:

```
ALTER TABLE <ім'я> {
[ALTER COLUMN <визначення_стовпця>] |
[ADD <визначення_стовпця>] |
[DROP COLUMN <ім'я_стовпця>] |
[ADD [WITH NOCHECK] CONSTRAINT < обмеження _для_таблиці>]}
```

Ключові слова CHECK (мається на увазі і NOCHECK) перед обмеженням таблиці наказують SQL Server провести тестування якщо не пройшли перевірку наявні в таблиці дані з урахуванням нового обмеження. WITH NOCHECK використовують лише в кількох випадках.

Доцільно розглянути кілька обмежень для фрази ALTER COLUMN. Стовпець не може бути змінений, якщо він:

має тип даних text, image, ntext або timestamp;

є обчислюваним стовпцем або використовується в обчислюваному стовпці;

є реплікований;

використовується в індексі за умови, що стовпець не має типу даних VARCHAR, NVARCHAR або VARBINARY. Тип даних не змінюється, і розмір стовпчика не зменшується;

застосовується в статистиці, що генерується оператором CREATE STATISTIC;

використовується в обмеженнях PRIMARY KEY, FOREIGN KEY, CHECK, UNIQUE;

указується як DEFAULT.

Можлива зміна опису стовпців. Часто це пов'язано зі зміною розмірів стовпців, додаванням або видаленням обмежень, що накладаються на їх значення. Синтаксис команди в цьому випадку має такий вигляд:

```
ALTER TABLE <ім'я_таблиці> ALTER <ім'я_стовпця>
<тип даних> <Розмір / точність>
```

Приклад 2.11. Модифікація типу даних поля "Ціна" в таблиці "Товар":

```
ALTER TABLE Товар ALTER Ціна INT
```

Слід мати на увазі, що характеристики стовпця можуть змінитися тільки з урахуванням таких обмежень:

зміна типу даних можлива тільки в тому випадку, якщо стовпець порожній;

для незаповненого стовпця можна змінювати розмір/точність;

для заповненого стовпця розмір/точність можна збільшити, але не можна зменшити;

обмеження NOT NULL може бути встановлено, якщо жодне значення в стовпці не містить NULL, опцію NOT NULL завжди можна скасувати;

дозволяється змінювати значення, встановлені за замовчуванням.

Змінювання структури таблиці в той момент, коли вона знаходиться в роботі, може призвести до втрати інформації. Наприклад, запит може потерпіти невдачу з тієї причини, що деякого поля в таблиці вже не існує. Таким чином, краще розробляти базу даних так, щоб ALTER TABLE застосовувати тільки в крайньому випадку.

2.3.5. SQL-оператор видалення таблиць DROP TABLE

SQL-оператор DROP TABLE призначений для видалення таблиць. Загальний вигляд синтаксису оператора DROP TABLE:

```
DROP TABLE <ім'я_таблиці1>, <ім'я_таблиці2>, ...
```

Особливості оператора DROP TABLE:

для того щоб мати можливість видалити таблицю, користувач повинен бути її власником, тобто творцем;

перед видаленням бажано очистити таблицю від даних, що дозволить уникнути випадкової і непоправної втрати інформації;

операція видалення виконується незалежно від того, існують в таблиці дані чи ні;

після видалення таблиці її відновлення неможливе;

у результаті виконання цієї команди ім'я таблиці більше не розпізнається;

щоб уникнути багатьох труднощів (наприклад, видалення таблиці було помилкою), рекомендується створити резервну копію бази даних перед видаленням з неї об'єктів; згодом за необхідності можна відновити потрібні об'єкти з резервної копії;

правило посилальної цілісності реалізовано і тут. Не можна видалити таблицю, яка містить первинний ключ, поки не буде видалена таблиця (або її обмеження цілісності), яка містить зовнішній ключ, що посилається на поле таблиці, що видаляється (послідовність видалення таблиць зворотна послідовності їх створення).

Розглянемо **команди видалення таблиць**:

```
DROP TABLE Товар, Продажі      ;
DROP TABLE Установи            ;
DROP TABLE Студенти            ;
DROP TABLE Університети        .
```

Для розроблення програмного продукту іноді потрібно повне видалення бази разом з подальшим її відновленням. Саме тому пишуть команди створення і видалення таблиць бази даних.

2.3.6. Створення, модифікація та видалення інших об'єктів у базі даних

Індексування стовпців необхідне для прискорення процесу вибірки даних. Однак додавання, зміна або видалення в індексованих стовпцях вимагає трохи більше часу, ніж в неіндексованих шпальтах. Цей недолік компенсується за рахунок збільшення продуктивності процесу вибірки даних. Для індексування стовпців використовується команда:

```
CREATE INDEX ім'я індексу ON ім'я таблиці (ім'я стовпця)
```

Наприклад:

```
CREATE INDEX Ind ON Товари (Товар)
```

Краще індексувати стовпці у ході створення таблиць, однак можна створювати індекси і після заповнення таблиць даними.

Для видалення індексу використовується команда:

```
DROP Ім'я_таблиці. Ім'я_індексу
```

Наприклад:

```
DROP Товар.Ind
```

З виконанням цієї команди система видаляє з бази даних заданий індекс.

Для деяких атрибутів необхідно задати потенційні значення, які можуть йому бути присвоєні. Такі дії можна виконувати з використанням домену. Домен створюється за допомогою команди CREATE DOMAIN, що з'явилася в SQL2 і має такий синтаксис:

```
CREATE DOMAIN ім'я_домену [AS]  тип_даних
[ DEFAULT значення]
[ CHECK (допустимі_значення)]
```

У рядку DEFAULT записується значення, яке встановлюється за замовчуванням, а в CHECK – усі допустимі значення атрибуту.

Приклад 2.12. Створити домен для позначення ціни деякого об'єкта, яка може знаходитися у діапазоні від 1 до 1000 гривень із значенням за замовчуванням, рівним 1 гривні:

```
CREATE DOMAIN CENA AS INT
DEFAULT  1.0
CHECK (VALUE Between 0.1  AND  1000)
```

Припустимо, що таблиця "Результати_змагань" має стовпець "Призові_місця", значення якого знаходяться в діапазоні від 1 до 3.

Приклад 2.13. Необхідно створити домен для урахування призових місць результатів змагань:

```
CREATE DOMAIN Приз_місце AS INTEGER
DEFAULT 3
CHECK (VALUE IN (1,2,3))
```

Використання команди CREATE DOMAIN у сучасних СУБД, з точки зору перенесення застосувань на різні платформи не завжди виправдано. Спроба перенести подібну команду з другої платформи (наприклад, у MS SQL SERVER) дасть повідомлення про помилку:

```
Msg 343,Unknown object type 'DOMAIN' used in a  CREATE,  DROP,
OR  ALTER statement
```

Аналогічна команда для ORACLE дасть повідомлення про помилку:

```
ORA-00901: invalid CREATE command
```

Видалення домену виконується командою DROP DOMAIN, яка має такий синтаксис:

```
DROP DOMAIN [ IF EXISTS ] ім'я [, ...]
                  [ CASCADE | RESTRICT ]
```

IF EXISTS – не вважати помилкою, якщо домен не існує;

CASCADE – автоматично видаляти об'єкти, залежні від даного домену (наприклад, стовпці таблиць), і, в свою чергу, всі залежні від них об'єкти;

RESTRICT – відмовити у видаленні домену, якщо від нього залежать будь-які об'єкти. Ця поведінка існує за умовчанням.

Приклад 2.14. Необхідно видалити домен для урахування призових місць:

```
DROP DOMAIN Приз_місце
```

Команда видаляє домен "Приз_місце", видалити домен спроможний тільки його "власник".

Запитання для самодіагностики

1. Опишіть склад і призначення команд SQL.

2. У яких випадках використовують команди групи DDL? Наведіть приклади використання операторів групи DDL.

3. За якими правилами вказують імена об'єктів таблиці в командах SQL?

4. У яких випадках використовують повне ім'я стовпця?

5. Які типи даних використовуються для створення бази даних? Наведіть приклади.

6. Який синтаксис запису команди CREATE TABLE? Наведіть приклади створення таблиць.

7. Який синтаксис запису команди ALTER TABLE?

8. Який синтаксис запису команди DROP TABLE?

9. Який синтаксис створення обмежень в таблицях? Наведіть приклади створення обмежень.

Лабораторна робота 1
Створення баз даних і таблиць у SQL Server

Цілі роботи:

1. Набуття практичних навичок роботи з базами даних у середовищі Visual Studio або Management Studio.
2. Набуття практичних навичок із створення баз даних і таблиць візуальними засобами.
3. Набуття практичних навичок зі створення баз даних і таблиць засобами мови DDL.

Перед виконанням роботи студент повинен знати:

1. Основи використання Visual Studio.
2. Основні об'єкти бази даних MS SQL Server.
3. Основні команди мови SQL.

Після виконання лабораторної роботи студент повинен уміти:

самостійно створювати бази даних, таблиці в них і заповнювати таблиці даними.

Хід роботи

1. Створення бази даних у вигляді mdf-файла у середовищі Visual Studio.
2. Створення таблиць візуальними засобами Visual Studio та встановлення зв'язків між ними.
3. Заповнення таблиць даними з використанням візуальних засобів.
4. Створення бази даних засобами мови DDL.
5. Створення таблиць засобами мови DDL.
6. Заповнення таблиць засобами мови SQL.
7. Створення індивідуальної бази даних, таблиць і заповнення таблиць засобами мови SQL.

Форма звітності

За результатами виконання роботи оформити електронний звіт засобами Word.

Критерії оцінювання

У 12-бальній системі оцінка результату захисту лабораторної роботи формується за такими правилами:

- за кожне завдання з діапазону 1 – 6 може бути виставлено від 0 до 1 бала;
- за завдання 7 може бути виставлено від 0 до 3 балів;
- за самостійне формулювання схеми бази даних під час виконання завдання 7 додається 1 бал;
- за кожне завдання із п. "Завдання для самостійного виконання" може бути виставлено від 0 до 1 бала;
- за кілька варіантів вирішення одного із завдань додається 1 бал;
- за вибір варіанта, який з кількох варіантів розв'язання є оптимальним, та обґрунтування вибору додається 1 бал;
- за побудову скрипта мовою SQL, в якому описані всі операції з базою даних (створення бази даних, таблиць і їхнє заповнення), додається 1 бал за кожний скрипт;
- за створення бази даних, таблиць і їхнє заповнення в іншій СУБД (Oracle, DB2 тощо) додається 1 бал за кожну базу даних;
- за запізнення із захистом лабораторної роботи знімається 2 бала.

Отримана кількість балів за відповіді на кожне завдання лабораторної роботи підсумовується. У результаті такого підрахунку студентом може бути отримано від 0 до 12 балів.

Рекомендована література

1. Андон Ф. Язык запросов SQL : учебный курс / Ф. Андон, В. Резниченко. – Санкт-Петербург : Питер ; Киев : BHV, 2006. – 416 с.

2. Астахова И. Ф. SQL в примерах и задачах / И. Ф. Астахова, А. П. Толстобров, И. М. Мельников.– Минск : Новое знание, 2002. – 176 с.

3. Боуман Д. Практическое руководство по SQL / Д. Боуман, С. Эмерсон, М. Дарновски. – Москва : Вильямс, 2002. – 352 с.

4. Дейт Дж. Введение в системы баз данных / Дж. Дейт. – 8-е изд. – Москва : Вильямс, 2005. – 1328 с.

5. Тарасов О. В., Лосєв М. Ю., Федько В. В. Використання мови SQL для роботи з сучасними системами керування базами даних / О. В. Тарасов, М. Ю. Лосєв, В. В. Федько. – Харків : Вид. ХНЕУ, 2013. – 348 с.

6. Тарасов О. В. Клієнт-серверні технології СКБД Oracle. Мова SQL Oracle : навч. посіб. для самостійної підготовки студентів з навч. дисц.

"Організація баз даних та знань" / О. В. Тарасов, В. В. Федько. – Харків : Вид. ХНЕУ ім. С. Кузнеця, 2015. – 384 с.

7. Форта Б. Освой самостоятельно SQL / Б. Форта. – Москва : Вильямс, 2005. – 288 с.

Інструкції до виконання
1. Створення бази даних у вигляді mdf-файла у середовищі Visual Studio

Завдання 1

Створити базу даних ***ХлібПрізвище*** в СКБД MS SQL Server у вигляді mdf-файла.

Примітки:

1) замість слова ***Прізвище*** в значенні імені бази даних потрібно вставити своє прізвище. Тоді база даних матиме відповідне ім'я, наприклад, ***ХлібПетренко***;

2) в описах подальших завдань база даних буде мати узагальнене ім'я ***Хліб***.

Виконання

1. Відкрийте Visual Studio.

2. Створіть проект Windows Forms мовою C# з ім'ям ***appПрізвище***.

Примітка: замість слова ***Прізвище*** в значенні імені проекту вставте своє прізвище. Тоді проект матиме відповідне ім'я, наприклад, ***app Петренко***.

3. Додайте до проекту mdf-файл бази даних. Для цього:

3.1. Клацніть правою клавішею миші (ПКМ) на значку проекту у вікні **Solution Explorer** і з контекстового меню виберіть команду **Add – New Item**;

3.2. Виберіть елемент **Data** у лівій панелі вікна **Add New Item**, потім елемент **Service-base Database** і в полі **Name** введіть ім'я бази даних, наприклад: ***ХлібПетренко.mdf***. Після цього клацніть кнопку **Add** (рис. 1).

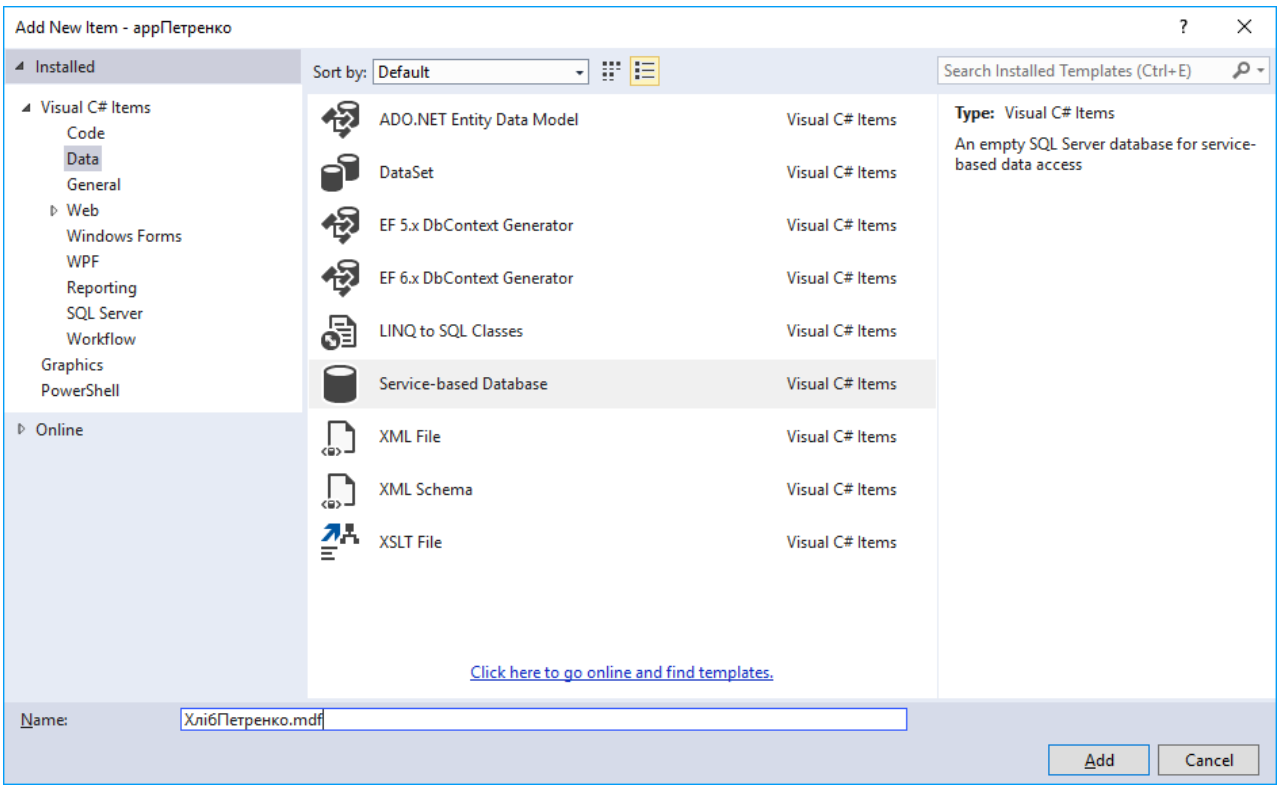


Рис. 1. **Вікно Add New Item**

4. Зачекайте деякий час, поки у вікнах **Solution Explorer** і **Server Explorer** з'являться позначки бази даних (рис. 2, 3).

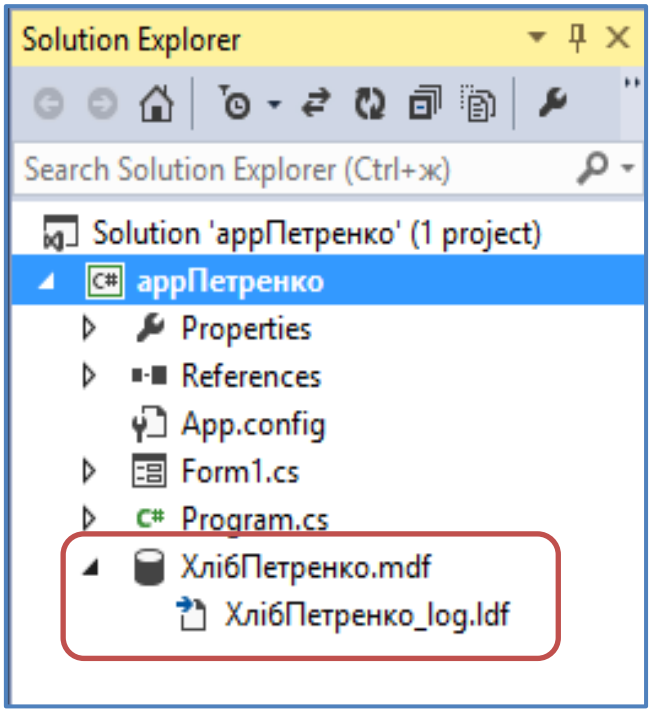


Рис. 2. **Позначка бази даних у вікні Solution Explorer**

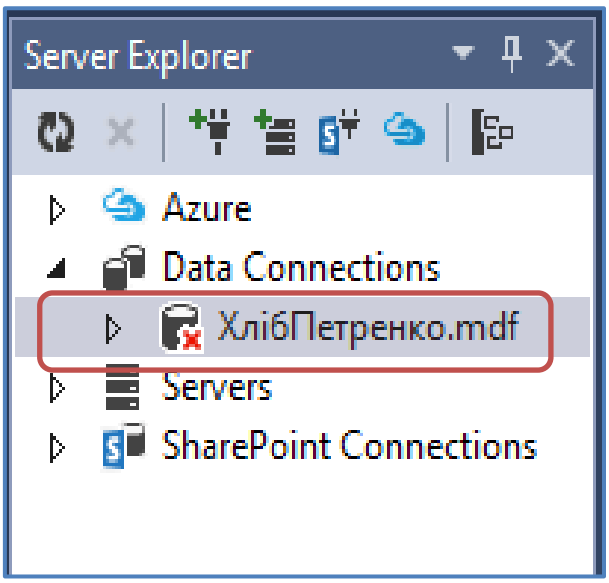


Рис. 3. **Позначка бази даних у вікні Server Explorer**

Примітка. Якщо не з'явиться позначка бази даних у вікні **Server Explorer**, клацніть позначку бази даних у вікні **Solution Explorer**.

5. У звіті з лабораторної роботи помістіть текст завдання і скриншоти, аналогічні рис. 2, 3.

6. Відкрийте вузол позначки бази даних у вікні **Server Explorer**, а в ньому підвузол **Tables**. Ця папка виявиться порожньою. Таблиці в ній створюються у наступному завданні.

2. Створення таблиць візуальними засобами Visual Studio та встановлення зв'язків між ними

Завдання 2

База даних ***Хліб*** містить такі таблиці:
Товари (Код_товару, Товар, Ціна, Ціна_закупівлі);
Виробники (Код_виробника, Виробник, Адреса, Телефон);
Продажі (Код_продажу, Дата, Код_виробника, Код_товару, Кількість).
Потрібно створити таблиці у базі даних ***Хліб*** і з'єднати їх (рис. 4).

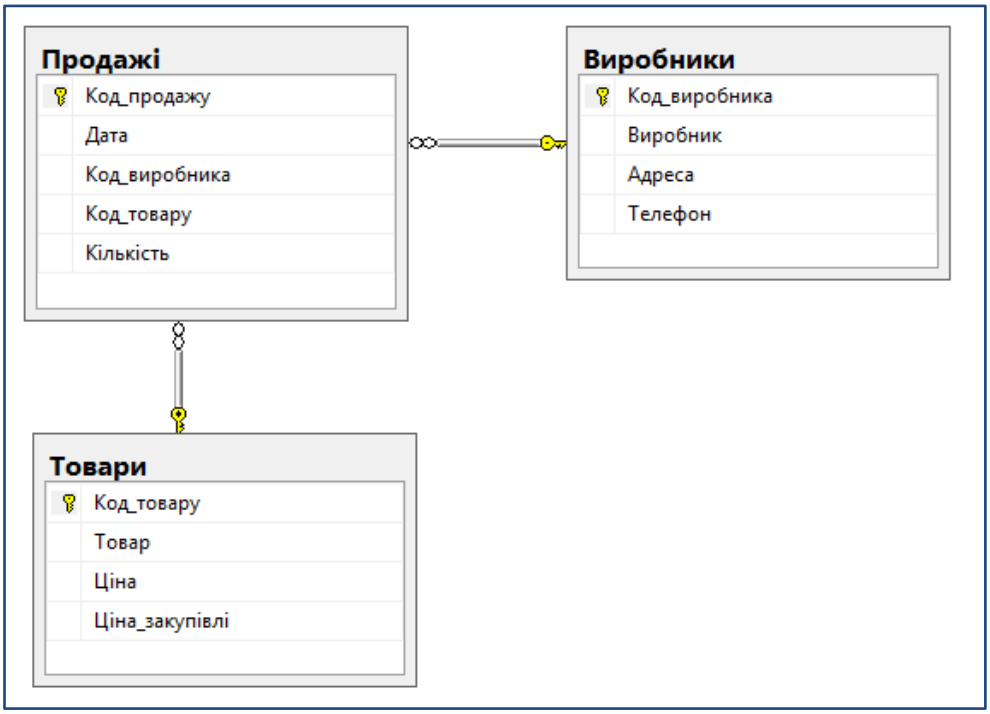


Рис. 4. **Зв'язки між таблицями в базі даних *Хліб***

Завдання 3

Створити таблицю ***Товари*** у базі даних ***Хліб*** з використанням візуальних засобів.

Виконання

1. Розкрийте вузол бази даних у вікні **Server Explorer**.

2. Клацніть ПКМ на позначці **Tables** бази даних і з контекстового меню виберіть команду **Add – New Table**.

3. Задайте імена полів та їхні властивості (табл. 1).

Таблиця 1

Поля таблиці *Товари*

№ п/п	Поле (Name)	Властивість	Значення
1	Код_товару	Data Type	int
		Identity Specification (Is Identity)	True
		Primary Key	True
2	Товар	Data Type	nvarchar
		Length	25
3	Ціна	Data Type	money
4	Ціна_закупівлі	Data Type	money

Для Visual Studio 2010:

4. Клацніть кнопку **Save** і введіть ім'я таблиці ***Товари***.

Для Visual Studio 2012 і вище:

5. Введіть ім'я таблиці ***Товари*** у рядку вкладки **T-SQL** (рис. 5).

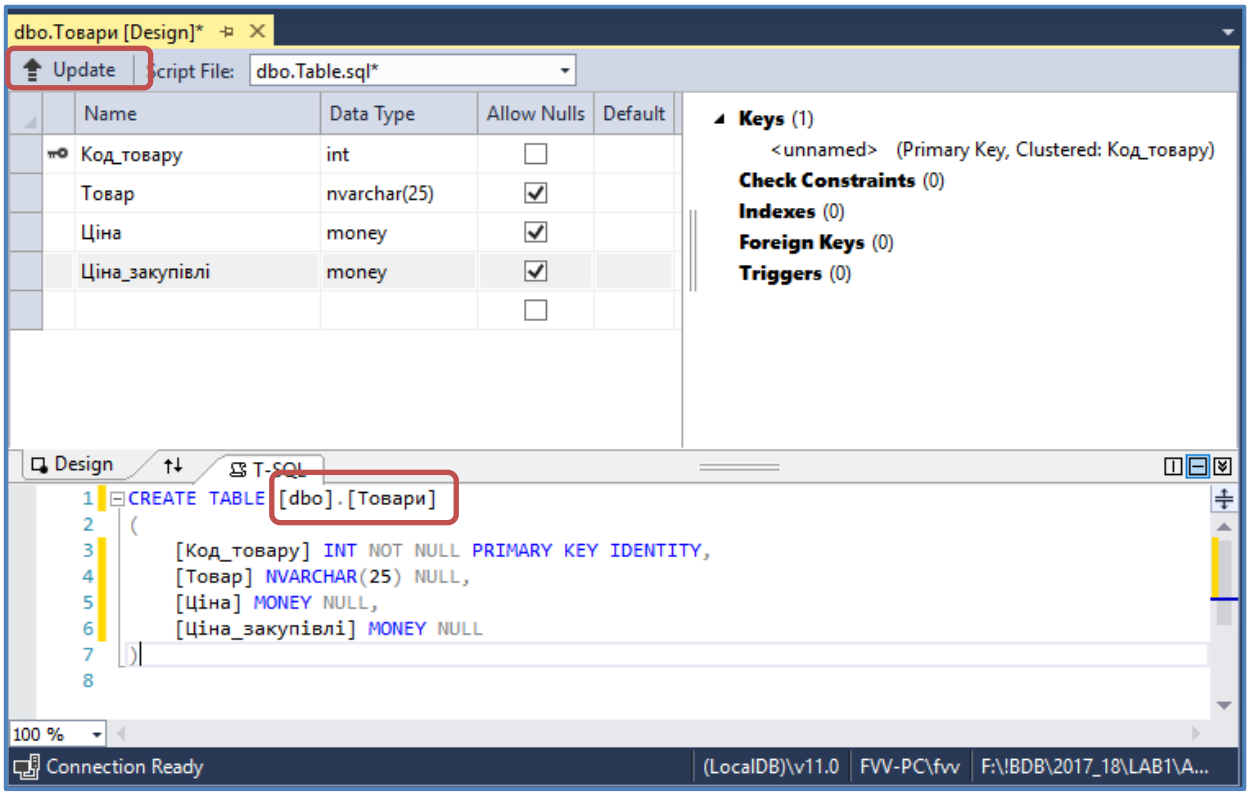


Рис. 5. **Вікно конструктора таблиці**

6. Клацніть кнопку **Update** у верхній частині вікна конструктора таблиці (див. рис. 5), а потім – кнопку **Update Database** у вікні, що з'явилося.

7. Оновіть вузол **Tables** у вікні **Server Explorer**, а потім розкрийте вузол таблиці ***Товари***. Перегляньте схему таблиці ***Товари*** у вікні **Server Explorer** (рис. 6).

8. У звіті з лабораторної роботи помістіть текст завдання і скриншоти, аналогічні рис. 5, 6.

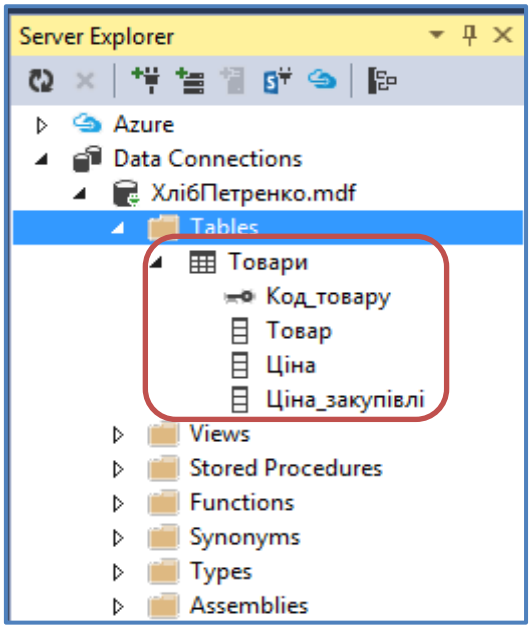


Рис. 6. **Схема таблиці *Товари* у вікні Server Explorer**

Завдання 4

Створити таблицю ***Виробники*** у базі даних ***Хліб*** з використанням візуальних засобів.

Виконання

Побудова таблиці ***Виробники*** здійснюється аналогічно описаному у завданні 1. Імена полів та їхніх властивостей подано в табл. 2.

Таблиця 2

Поля таблиці Виробники

№ п/п	Поле (Name)	Властивість	Значення
1	Код_виробника	Data Type	int
		Identity Specification (Is Identity)	True
		Primary Key	True
2	Виробник	Data Type	nvarchar
		Length	20
3	Адреса	Data Type	nvarchar
		Length	30
4	Телефон	Data Type	nvarchar
		Length	15

Додайте відповідні дані до звіту з лабораторної роботи.

Завдання 5

Створити таблицю ***Продажі*** у базі даних ***Хліб*** з використанням візуальних засобів.

Виконання

Побудова таблиці ***Продажі*** здійснюється аналогічно описаному у завданні 1. Імена полів та їхніх властивостей подано в табл. 3.

Таблиця 3

Поля таблиці *Продажі*

№ п/п	Поле (Name)	Властивість	Значення
1	Код_продажу	Data Type	int
		Identity Specification (Is Identity)	True
		Primary Key	True
2	Дата	Data Type	date
3	Код_виробника	Data Type	int
4	Код_товару	Data Type	int
5	Кількість	Data Type	smallint

Додайте відповідні дані до звіту з лабораторної роботи.

Завдання 6

Установити зв'язки між таблицями в базі даних ***Хліб***.

Ідея розв'язку

У таблиці ***Продажі*** може бути кілька записів про продаж того самого товару від різних виробників протягом одного дня. Ще більше буде записів у таблиці ***Продажі***, у яких згадується той самий товар протягом кількох днів. Тому таблиці ***Товари*** та ***Продажі*** пов'язані відношенням один-до-багатьох.

З аналогічних причин таблиці ***Виробники*** та ***Продажі*** пов'язані відношенням один-до-багатьох.

Отже у базі даних ***Хліб*** таблиці ***Товари*** та ***Виробники*** є батьківськими, а таблиця ***Продажі*** – дочірньою.

Зв'язок між батьківськими та дочірньою таблицями встановлюється за схожістю значень первинного ключа у батьківській таблиці та відповідного зовнішнього ключа у дочірній. Наприклад, таблиці ***Товари***

та ***Продажі*** пов'язані первинним ключем ***Код_товару*** в таблиці ***Товари*** та однойменним зовнішнім ключем у таблиці ***Продажі***.

Установлення зв'язків між таблицями у різних версіях Visual Studio здійснюється різними засобами. У Visual Studio 2010 є діаграми баз даних, а в подальших версіях вони відсутні. Тому у Visual Studio 2010 для цього можна застосувати візуальні засоби, а в подальших версіях – засоби мови SQL.

Примітка. У всіх версіях Management Studio є діаграми баз даних.

Виконання

Для Visual Studio 2010:

1. Додайте діаграму до бази даних за допомогою контекстового меню значка **Diagrams**.
2. Додайте всі три таблиці до діаграми у вікні, що відкрилося.
3. Перетягніть первинний ключ ***Код_товару*** з батьківської таблиці ***Товари*** на однойменний зовнішній ключ дочірньої таблиці ***Продажі***.
4. Повторіть п. 3 для таблиць ***Виробники*** та ***Продажі***.
5. На рис. 7 подано схему бази даних ***Хліб***. У звіті з лабораторної роботи помістіть текст завдання і скриншот, аналогічний рис. 7.

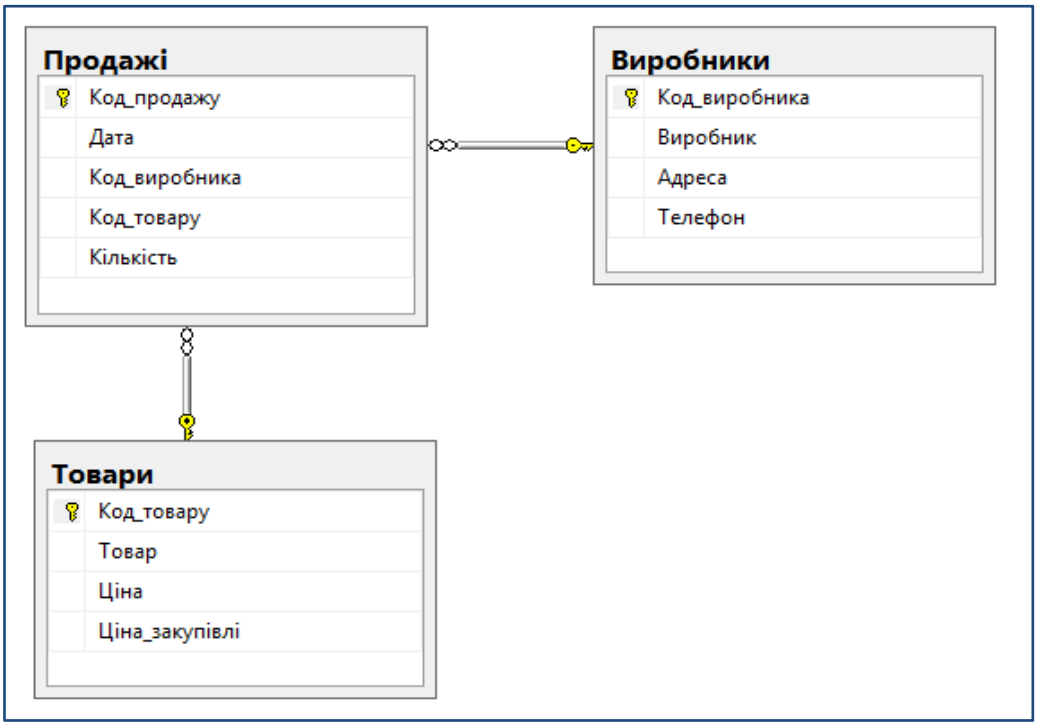


Рис. 7. **Схема бази даних *Хліб***

Для Visual Studio 2012 і вище:

1. Клацніть ПКМ у конструкторі дочірньої таблиці ***Продажі*** на тексті **Foreign Keys**, що знаходиться у правій панелі вікна конструктора, та виберіть команду **Add New Foreign Key**.

2. Замініть у кінці вкладки **T-SQL** текст.

```
CONSTRAINT [FK_Продажі_ToTable] FOREIGN KEY ([Column]) REFERENCES
[ToTable]([ToTableColumn])
```

на такий:

```
CONSTRAINT [FK_ТовариПродажі] FOREIGN KEY ([Код_товару]) REFERENCES
[dbo].[Товари] ([Код_товару]) ON DELETE CASCADE,
CONSTRAINT [FK_ВиробникиПродажі] FOREIGN KEY ([Код_виробника])
REFERENCES [dbo].[Виробники] ([Код_виробника]) ON DELETE CASCADE
```

3. Клацніть кнопку **Update** у верхній частині вікна конструктора таблиці, а потім – кнопку **Update Database** у вікні, що з'явилося.

4. На рис. 8 подано схему дочірньої таблиці ***Продажі*** зі встановленими зв'язками. У звіті з лабораторної роботи помістіть текст завдання і скриншот, аналогічний рис. 8.

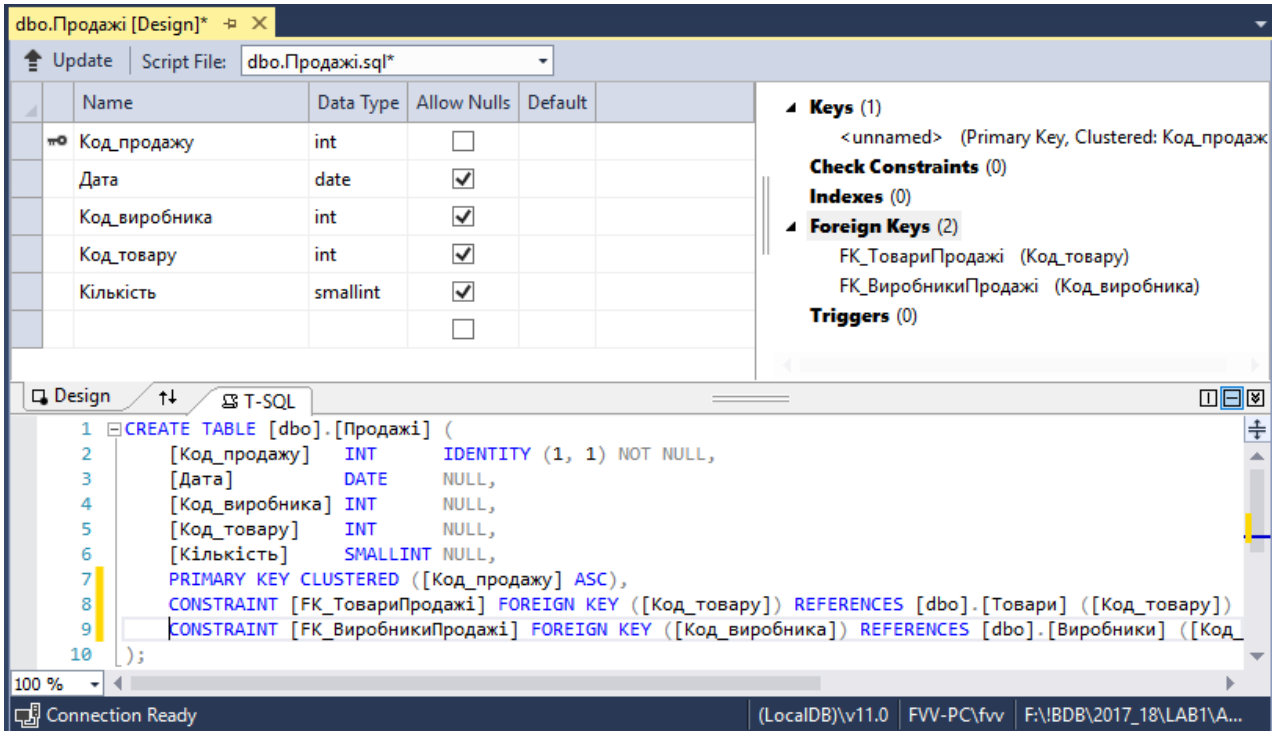


Рис. 8. **Схема таблиці *Продажі* зі встановленими зв'язками**

3. Заповнення таблиць даними з використанням візуальних засобів

Завдання 7

Заповнити даними таблиці бази даних ***Хліб***.

Ідеї розв'язку

Спочатку вводять дані до батьківських таблиць, а потім – до дочірніх, щоб не порушити посилальну цілісність даних. Вона передбачає, що значення зовнішніх ключів у дочірній таблиці можуть вибиратися тільки з множини відповідної батьківської таблиці. Наприклад, можна продати тільки той товар, що є в таблиці ***Товари***, і від того виробника, який є в таблиці ***Виробники***.

Значення первинних ключів не вводять, оскільки вони мають властивість Identity, тобто автоматично задаються сервером під час збереження даних.

Введений запис автоматично зберігається в базі даних у разі переходу на інший рядок.

Значення первинних ключів відображаються після клацання кнопки **Refresh** у вікні даних таблиці.

Виконання

1. Заповніть даними таблицю ***Товари***. Для цього:

1.1. Розкрийте вузол бази даних у вікні **Server Explorer**.

1.2. Клацніть ПКМ на значку таблиці ***Товари*** бази даних ***Хліб***, що знаходиться у вікні **Server Explorer**, і з контекстового меню виберіть команду **Show Table Data**.

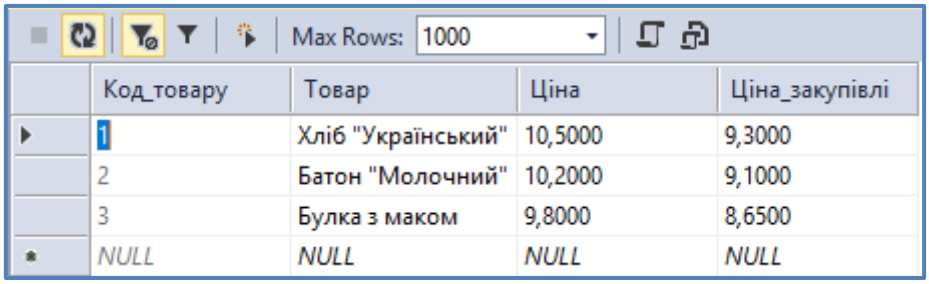
1.3. Введіть дані, що подані в табл. 4.

Таблиця 4

Дані таблиці *Товари*

Товар	Ціна	Ціна_закупівлі
Хліб "Український"	10,50	9,30
Батон "Молочний"	10,20	9,10
Булка з маком	9,80	8,65

1.4. Оновіть зображення даних у таблиці, клацнувши кнопку **Refresh** у вікні даних таблиці. Таблицю подано на рис. 9.



Код_товару	Товар	Ціна	Ціна_закупівлі
1	Хліб "Український"	10,5000	9,3000
2	Батон "Молочний"	10,2000	9,1000
3	Булка з маком	9,8000	8,6500
NULL	NULL	NULL	NULL

Рис. 9. **Дані таблиці *Товари***

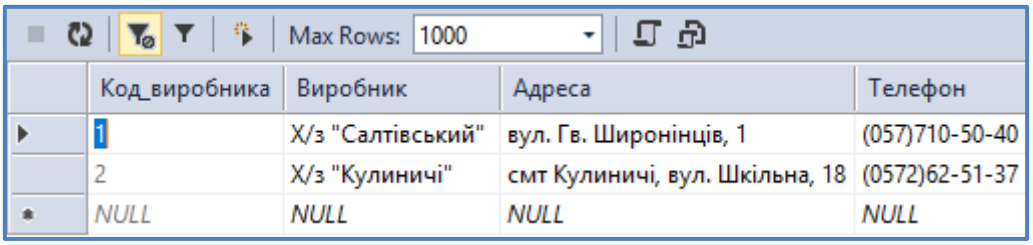
2. Заповніть даними таблицю ***Виробники*** аналогічно п. 1. Дані таблиці подано в табл. 5.

Таблиця 5

Дані таблиці *Виробники*

Виробник	Адреса	Телефон
Х/з "Салтівський"	вул. Гв. Широнінців, 1	(057)710-50-40
Х/з "Кулиничі"	смт Кулиничі, вул. Шкільна, 18	(0572)62-51-37

Після оновлення дані таблиці ***Виробники*** подано на рис. 10.



Код_виробника	Виробник	Адреса	Телефон
1	Х/з "Салтівський"	вул. Гв. Широнінців, 1	(057)710-50-40
2	Х/з "Кулиничі"	смт Кулиничі, вул. Шкільна, 18	(0572)62-51-37
NULL	NULL	NULL	NULL

Рис. 10. **Дані таблиці *Виробники***

3. Заповніть даними таблицю ***Продажі*** аналогічно п. 1. Дані таблиці подано в табл. 6.

Таблиця 6

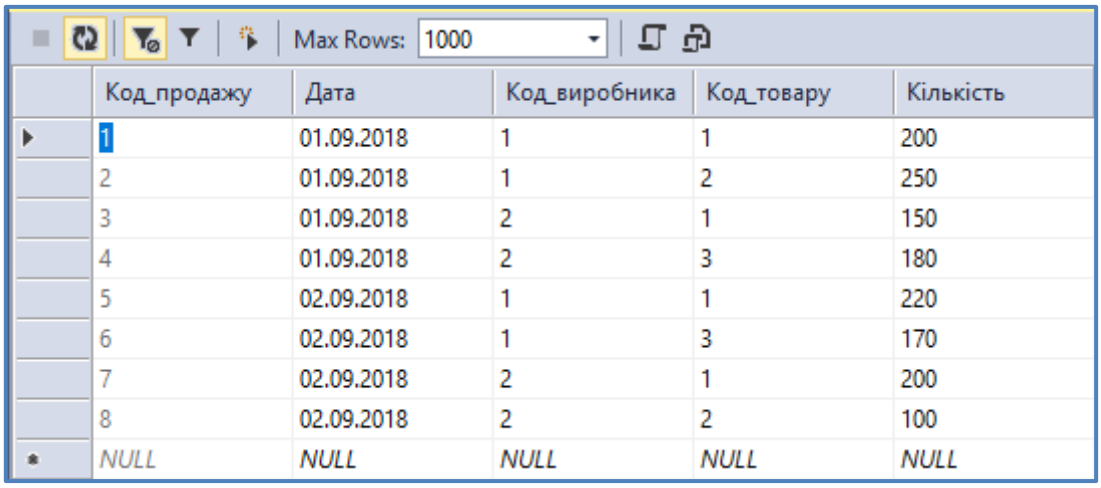
Дані таблиці *Продажі*

Дата	Код_виробника	Код_товару	Кількість
1	2	3	4
01.09.2018	1	1	200

Закінчення табл. 6

1	2	3	4
01.09.2018	1	2	250
01.09.2018	2	1	150
01.09.2018	2	3	180
02.09.2018	1	1	220
02.09.2018	1	3	170
02.09.2018	2	1	200
02.09.2018	2	2	100

Оновлені дані таблиці ***Продажі*** подано на рис. 11.



Код_продажу	Дата	Код_виробника	Код_товару	Кількість
1	01.09.2018	1	1	200
2	01.09.2018	1	2	250
3	01.09.2018	2	1	150
4	01.09.2018	2	3	180
5	02.09.2018	1	1	220
6	02.09.2018	1	3	170
7	02.09.2018	2	1	200
8	02.09.2018	2	2	100
NULL	NULL	NULL	NULL	NULL

Рис. 11. **Дані таблиці *Продажі***

5. У звіті з лабораторної роботи помістіть текст завдання і скриншоти, аналогічні рис. 9 – 11.

4. Створення бази даних засобами мови DDL

Завдання 8

Створити базу даних ***ТоргівляПрізвище*** в СУБД MS SQL Server у вигляді mdf-файла, використовуючи засоби мови DDL.

Примітка. Замість слова ***Прізвище*** в значенні імені бази даних потрібно вставити своє прізвище. Тоді база даних матиме відповідне ім'я, наприклад, ***ТоргівляПетренко***.

Ідея розв'язку

Запити мови DDL задають у вікні редактора T-SQL. Воно у різних версіях Visual Studio викликається по-різному. У Visual Studio 2010 для цього використовують команду **Data – Transact-SQL Editor**, а в подальших версіях Visual Studio достатньо з контекстового меню позначки бази даних вибрати команду **New Query**.

Виконання

1. Відкрийте вікно редактора T-SQL і введіть запит на створення бази даних у форматі

```
CREATE DATABASE ім'я_бази_даних
ON (NAME = логічне_ім'я_файла,FILENAME = шлях_і_фізичне_ім'я_файла)
```

наприклад,

```
CREATE DATABASE ТоргівляПетренко On (NAME = ТоргівляПетренко,
FILENAME='F:\fvv\DB\ТоргівляПетренко.mdf')
```

Примітка. У параметрі **FILENAME** вкажіть повний шлях до mdf-файла, в якому зберігається база даних.

2. Запустіть запит на виконання, клацнувши кнопку **Execute** у вікні редактора.

3. Відкрийте вікно папки, що вказана у параметрі **FILENAME;** переконайтеся, що в ній з'явилося два файли – з розширенням mdf' (файл даних) і з розширенням ldf (файл журналу).

4. Для спрощення подальших операцій скопіюйте нові файли в проект. Для цього:

4.1. Клацніть ПКМ на позначці проекту у вікні **Solution Explorer** і з контекстового меню виберіть команду **Add – Existing Item**.

4.2. Перейдіть у вікно папки з файлами нової бази даних, установіть у фільтрі значення **All files(*.*)**, виділіть mdf-файл нової бази даних і клацніть кнопку **Add**.

4.3. Зачекайте деякий час поки у вікнах **Solution Explorer** і **Server Explorer** з'являться позначки нової бази даних ***Торгівля***.

5. Відкрийте вузол позначки бази даних ***Торгівля*** у вікні **Server Explorer**.

6. У звіті з лабораторної роботи помістіть текст завдання, запит на створення бази даних ***Торгівля*** і скриншоти вікон **Solution Explorer** і **Server Explorer** із позначками нової бази даних.

5. Створення таблиць засобами мови DDL

Завдання 9

База даних ***Торгівля*** містить такі таблиці:

Замовники (Код_замовника, Прізвище_замовника, Місто, Рейтинг);

Продавці (Код_продавця, Прізвище_продавця, Місто, Комісійні);

Замовлення (Код_замовлення, Дата_замовлення, Сума, Код_продавця, Код_замовника).

Ідеї розв'язку

Після того як створено базу даних, для виконання подальших запитів з нею потрібно встановити з'єднання у вікні редактора T-SQL.

У Visual Studio 2010 для цього виконують такі дії.

1. Клацають на позначці бази даних у вікні **Server Explorer**.

2. Копіюють у буфер обміну значення властивості **Connection String** з вікна **Properties** клавішами **Ctrl-C**.

3. Виконують команду **Data – Transact-SQL Editor – New Query Connection**.

4. Клацають кнопку **Options** у вікні **Connect to Server**, а потім переходять на вкладку **Additional Connection Parameters**, вставляють з буфера рядок з'єднання клавішами **Ctrl-V** і клацають кнопку **Connect**.

5. Установлюють ім'я бази даних у полі зі списком **Database** вікна редактора.

У подальших версіях Visual Studio для того, щоб встановити з'єднання з базою даних, достатньо з контекстового меню позначки бази даних вибрати команду **New Query**.

Завдання 10

Створити таблицю ***Замовники*** у базі даних ***Торгівля*** з використанням мови DDL.

Виконання

1. Відкрийте вікно редактора T-SQL, що з'єднаний з базою даних ***Торгівля***, і введіть запит на створення таблиці ***Замовники***

```
CREATE TABLE Замовники
    (Код_замовника int IDENTITY (1, 1) Primary Key,
    Прізвище_замовника NVARCHAR(30),
    Місто NVARCHAR(30),
    Рейтинг INT
     )
```

2. Запустіть запит на виконання, клацнувши у вікні редактора кнопку **Execute**.

3. Обновіть вузол **Tables** у вікні **Server Explorer**, а потім розкрийте вузол таблиці ***Замовники***. Перегляньте схему таблиці ***Замовники*** у вікні **Server Explorer** (рис. 12).

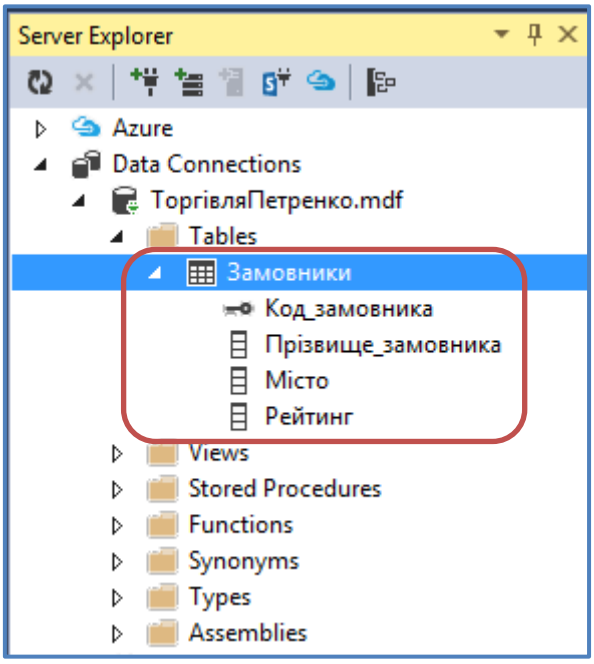


Рис. 12. **Схема таблиці *Замовники* у вікні Server Explorer**

4. У звіті з лабораторної роботи помістіть текст завдання, запит на створення таблиці ***Замовники*** та скриншот, аналогічний рис. 12.

Завдання 11

Створити таблицю ***Продавці*** у базі даних ***Торгівля*** з використанням мови DDL.

Виконання

Побудова таблиці ***Продавці*** здійснюється аналогічно описаному в завданні 1. Запит на створення таблиці ***Продавці*** має такий вигляд:

```
CREATE TABLE Продавці
    (
    Код_продавця int IDENTITY (1, 1) Primary Key,
    Прізвище_продавця NVARCHAR(30),
    Місто NVARCHAR(30),
    Комісійні DECIMAL(5,2)
     )
```

Додайте відповідні дані до звіту з лабораторної роботи.

Завдання 12

Створити таблицю ***Замовлення*** у базі даних ***Торгівля*** з використанням візуальних засобів. Під час створення таблиці встановити зв'язки з батьківськими таблицями ***Замовники*** та ***Продавці***, використавши зовнішні ключі.

Виконання

Побудова таблиці ***Замовлення*** здійснюється аналогічно описаному у завданні 1.

Запит на створення таблиці ***Замовлення*** має такий вигляд:

```
CREATE TABLE Замовлення
    (Код_замовлення INT IDENTITY (1, 1) Primary Key,
     Дата_замовлення DATE,
     Сума MONEY,
     Код_продавця INT FOREIGN KEY REFERENCES Продавці(Код_продавця),
     Код_замовника INT FOREIGN KEY REFERENCES Замовники(Код_замовника)
     )
```

Додайте відповідні дані до звіту з лабораторної роботи.

6. Заповнення таблиць засобами мови SQL

Завдання 13

Заповнити даними таблиці бази даних ***Торгівля*** з використанням команд SQL. У результаті таблиці повинні зберігати такі кількості записів:

таблиця ***Замовники*** – не менше 10 записів;
таблиця ***Продавці*** – не менше 10 записів;
таблиця ***Замовлення*** – не менше 30 записів.

Ідеї розв'язку

Спочатку вводять дані до батьківських таблиць, а потім – до дочірніх, щоб не порушити посилальну цілісність даних. Для цього використовують оператор INSERT. У найпростішому випадку для введення одного рядка таблиці він має такий формат:

```
INSERT INTO ім'я_таблиці VALUES (значення1, значення2,…)
```

Цей формат застосовують, якщо стовпці таблиці вказані в повному складі та в тому порядку, в якому вони перераховані під час створення таблиці. Значення первинних ключів не вводять, оскільки вони мають властивість Identity, тобто автоматично задаються сервером під час збереження даних.

Якщо у разі наявності властивості Identity у первинного ключа потрібно самостійно задавати його значення, слід виконувати дві умови:

1. указувати імена стовпців, значення яких вводять;
2. перед введенням таких даних виконувати для таблиці оператор

```
SET IDENTITY_INSERT ім'я_таблиці ON
```

Тобто оператор INSERT в цьому разі має такий формат:

```
INSERT INTO ім'я_таблиці (стовпець1, стовпець2, …)
        VALUES (значення1, значення2,…)
```

Щоб текстові значення вводилися як символи Юнікоду, перед кожним з них потрібно ставити літеру N. У протилежному разі під час перегляду даних можуть виводитися символи ***?***.

Наприклад, введення даних про першого замовника реалізується так:

```
SET IDENTITY_INSERT Замовники ON;
INSERT INTO Замовники (Код_замовника, Прізвище_замовника, Місто, Рейтинг)
VALUES(1, N'Іваненко', N'Київ', 95);
SET IDENTITY_INSERT Замовники OFF;
```

Щоб дата оброблялася в вигляді 'ДД.ММ.РРРР' для SQL Server доцільно відразу ввести команду

```
SET LANGUAGE RUSSIAN
```

Наприклад, введення даних про замовлення від першого замовника у першого продавця реалізується так:

```
SET LANGUAGE RUSSIAN;
SET IDENTITY_INSERT Замовлення ON;
INSERT INTO Замовлення (Код_замовлення, Дата_замовлення, Сума,
Код_продавця, Код_замовника) VALUES(1, '01.09.2018', 2000, 1, 1);
SET IDENTITY_INSERT Замовлення OFF;
```

Виконання

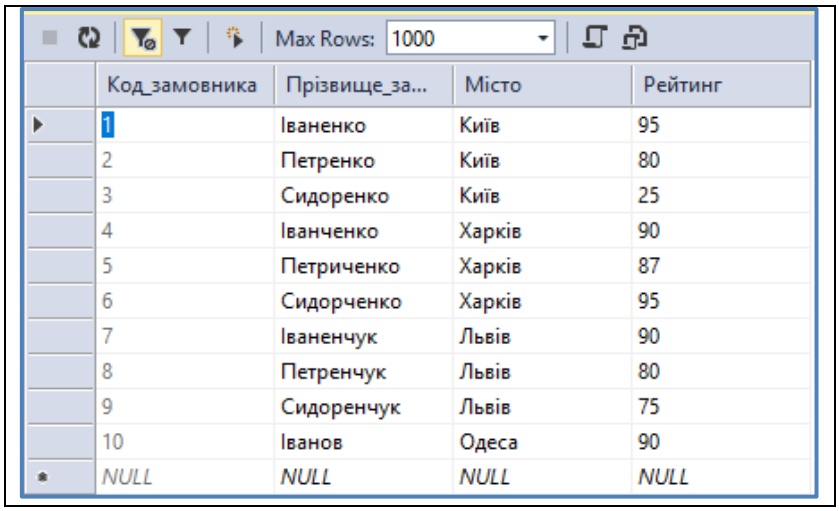
1. Заповніть даними таблицю ***Замовники***. Для цього:

1.1. Введіть у вікні редактора T-SQL текст такого скрипту:

```
SET IDENTITY_INSERT Замовники ON
INSERT INTO Замовники (Код_замовника,Прізвище_замовника,Місто,Рейтинг)
VALUES(1, N'Іваненко', N'Київ', 95);
INSERT INTO Замовники (Код_замовника,Прізвище_замовника,Місто,Рейтинг)
VALUES(2, N'Петренко', N'Київ', 80);
INSERT INTO Замовники (Код_замовника,Прізвище_замовника,Місто,Рейтинг)
VALUES(3, N'Сидоренко',N'Київ', 25);
INSERT INTO Замовники (Код_замовника,Прізвище_замовника,Місто,Рейтинг)
VALUES(4, N'Іванченко',N'Харків', 90);
INSERT INTO Замовники (Код_замовника,Прізвище_замовника,Місто,Рейтинг)
VALUES(5, N'Петриченко',N'Харків', 87);
INSERT INTO Замовники (Код_замовника,Прізвище_замовника,Місто,Рейтинг)
VALUES(6, N'Сидорченко',N'Харків', 95);
INSERT INTO Замовники (Код_замовника,Прізвище_замовника,Місто,Рейтинг)
VALUES(7, N'Іваненчук',N'Львів', 90);
INSERT INTO Замовники (Код_замовника,Прізвище_замовника,Місто,Рейтинг)
VALUES(8, N'Петренчук',N'Львів', 80);
INSERT INTO Замовники (Код_замовника,Прізвище_замовника,Місто,Рейтинг)
VALUES(9, N'Сидоренчук',N'Львів', 75);
INSERT INTO Замовники (Код_замовника,Прізвище_замовника,Місто,Рейтинг)
VALUES(10, N'Іванов',N'Одеса', 90);
SET IDENTITY_INSERT Замовники OFF
```

1.2. Запустіть запит на виконання, клацнувши у вікні редактора кнопку **Execute**.

1.3. Обновіть вузол **Tables** у вікні **Server Explorer**, а потім з контекстового меню таблиці ***Замовники*** виберіть команду **Show Table Data**. Перегляньте дані таблиці ***Замовники*** у вікні даних (рис. 13).



Код_замовника	Прізвище_за...	Місто	Рейтинг
1	Іваненко	Київ	95
2	Петренко	Київ	80
3	Сидоренко	Київ	25
4	Іванченко	Харків	90
5	Петриченко	Харків	87
6	Сидорченко	Харків	95
7	Іваненчук	Львів	90
8	Петренчук	Львів	80
9	Сидоренчук	Львів	75
10	Іванов	Одеса	90
NULL	NULL	NULL	NULL

Рис. 13. **Дані таблиці *Замовники***

1.4. У звіті з лабораторної роботи помістіть текст завдання, скрипт на заповнення таблиці ***Замовники*** та скриншот, аналогічний рис. 13.

2. Заповніть таблицю ***Продавці*** у базі даних ***Торгівля*** з використанням мови DDL (не менше 10 записів), аналогічно описаному в завданні 1. Додайте відповідні дані до звіту з лабораторної роботи.

3. Заповніть таблицю ***Замовлення*** у базі даних ***Торгівля*** з використанням мови DDL (не менше 30 записів). Під час заповнення таблиці використовуйте значення зовнішніх ключів, які співпадають зі значеннями первинних ключів батьківських таблиць ***Замовники*** та ***Продавці***. Додайте відповідні дані до звіту з лабораторної роботи.

7. Створення індивідуальної бази даних, таблиць і заповнення таблиць засобами мови SQL

Завдання 14

За вибраним варіантом створити базу даних і таблиці в ній, встановити зв'язки між таблицями та заповнити даними таблиці, використавши засоби мови SQL.

Варіанти баз даних.

1. База даних **Готель** містить такі таблиці:

Клієнти (Код клієнта, Прізвище, Ім'я, По батькові, Паспортні дані, Коментар);

Номери (Код номера, Номер, Кількість місць, Комфортність, Ціна);

Поселення (Код поселення, Код клієнта, Код номера, Дата поселення, Дата звільнення, Примітка).

2. База даних **Ломбард** містить такі таблиці:

Клієнти (Код клієнта, Прізвище, Ім'я, По батькові, Номер паспорта, Серія паспорта, Дата видачі паспорта);

Категорії товарів (Код категорії товарів, Назва, Примітка);

Здача в ломбард (Код, Код категорії товарів, Код клієнта, Опис товару, Дата здачі, Дата повернення, Сума, Комісійні).

3. База даних **Оптово-роздрібна компанія** містить такі таблиці:

Товари (Код товару, Найменування, Оптова ціна, Роздрібна ціна, Опис);

Покупці (Код покупця, Телефон, Контактна особа, Адреса);

Угоди (Код угоди, Дата угоди, Код товару, Кількість, Код покупця, Ознака оптового).

4. База даних **Оптова компанія** містить такі таблиці:

Товари (Код товару, Ціна, Доставка, Опис);

Замовники (Код замовника, Найменування, Адреса, Телефон, Контактна особа);

Замовлення (Код замовлення, Код замовника, Код товару, Кількість, Дата).

5. База даних **Бюро з працевлаштування** містить такі таблиці:

Роботодавці (Код роботодавця, Назва, Вид діяльності, Адреса, Телефон);

Здобувачі (Код здобувача, Прізвище, Ім'я, По батькові, Кваліфікація, Вид діяльності, Інші дані, Передбачуваний розмір заробітної плати);

Угоди (Код здобувача, Код роботодавця, Посада, Комісійні).

6. База даних **Нотаріальна контора** містить такі таблиці:

Клієнти (Код клієнта, Назва, Вид діяльності, Адреса, Телефон);

Угоди (Код угоди, Код клієнта, Код послуги, Сума, Комісійні, Опис);

Послуги (Код послуги, Назва, Опис).

7. База даних **Страхова компанія** містить такі таблиці:

Договори (Номер договору, Дата висновку, Страхова сума, Тарифна ставка, Код філії, Код виду страхування);

Вид страхування (Код виду страхування, Найменування);

Філія (Код філії, Найменування філії, Адреса, Телефон).

8. База даних **Запасні частини** містить такі таблиці:

Постачальники (Код постачальника, Назва, Адреса, Телефон);

Деталі (Код деталі, Назва, Артикул, Ціна, Примітка);

Поставки (Код постачальника, Код деталі, Кількість, Дата).

9. База даних **Навчальний заклад** містить такі таблиці:

Групи (Номер групи, Спеціальність, Відділення, Кількість студентів);

Викладачі (Код викладача, Прізвище, Ім'я, По батькові, Телефон, Стаж);

Навантаження (Код викладача, Номер групи, Кількість годин, Предмет, Тип заняття, Оплата).

10. База даних **Університет** містить такі таблиці:

Студенти (Код студента, Прізвище, Ім'я, По батькові, Адреса, Телефон);

Предмети (Код предмета, Назва, Обсяг лекцій, Обсяг практик, Обсяг лабораторних робіт);

Екзамени (Код студента, Код предмета, Оцінка).

11. База даних **Академія** містить такі таблиці:

Викладачі (Код викладача, Прізвище, Ім'я, По батькові, Науковий ступінь, Посада, Стаж);

Предмети (Код предмета, Назва, Кількість годин);

Навантаження (Код викладача, Код предмета, Номер групи).

12. База даних **Комерційна компанія** містить такі таблиці:

Співробітники (Код співробітника, Прізвище, Ім'я, По батькові, Оклад);

Види робіт (Код виду, Опис, Оплата за день);

Роботи (Код співробітника, Код виду, Дата початку, Дата закінчення).

13. База даних **Ремонт верстатів** містить такі таблиці:

Види верстатів (Код виду верстата, Країна, Рік випуску, Марка);

Види ремонту (Код ремонту, Назва, Тривалість, Вартість, Примітки).

Ремонт (Код виду верстата, Код ремонту, Дата початку, Примітки).

14. База даних **Туристична компанія** містить такі таблиці:

Маршрути (Код маршруту, Країна, Клімат, Тривалість, Готель, Вартість);

Путівки (Код маршруту, Код клієнта, Дата відправлення, Кількість, Знижка);

Клієнти (Код клієнта, Прізвище, Ім'я, По батькові, Адреса, Телефон).

15. База даних **Перевезення вантажів** містить такі таблиці:

Маршрути (Код маршруту, Назва, Довжина, Кількість днів у дорозі, Оплата);

Водії (Код водія, Прізвище, Ім'я, По батькові, Стаж);

Виконана робота (Код маршруту, Код водія, Дата відправлення, Дата повернення, Премія).

16. База даних **Телефонна компанія** містить такі таблиці:

Абоненти (Код абонента, Номер телефону, Адреса);

Міста (Код міста, Назва, Тариф денний, Тариф нічний);

Переговори (Код переговорів, Код абонента, Код міста, Дата, Кількість хвилин, Час доби).

17. База даних **Банк** містить такі таблиці:

Види кредитів (Код виду, Назва, Умови отримання, Ставка, Термін);

Клієнти (Код клієнта, Назва, Вид власності, Адреса, Телефон, Контактна особа);

Кредити (Код виду, Код клієнта, Сума, Дата видачі).

18. База даних **Бухгалтерія** містить такі таблиці:

Відділи (Код відділу, Назва, Кількість співробітників);

Види витрат (Код виду, Назва, Опис, Гранична норма);

Витрати (Код витрати, Код виду, Код відділу, Сума, Дата).

19. База даних **Прокат книг** містить такі таблиці:

Книги (Код книги, Назва, Автор, Станова вартість, Вартість прокату, Жанр);

Читачі (Код читача, Прізвище, Ім'я, По батькові, Адреса, Телефон);

Видані книги (Код книги, Код читача, Дата видачі, Дата повернення).

20. База даних **Прокат автомобілів** містить такі таблиці:

Автомобілі (Код автомобіля, Марка, Вартість, Вартість прокату, Тип).

Клієнти (Код клієнта, Прізвище, Ім'я, По батькові, Адреса, Телефон);

Видані автомобілі (Код автомобіля, Код клієнта, Дата видачі, Дата повернення).

21. База даних **Зустрічі з товаришами** містить такі таблиці:

Товариші (Код товариша, Ім'я, Вік);

Теми (Код теми, Назва, Опис);

Зустрічі (Код зустрічі, Дата, Код товариша, Код теми, Оцінка).

Виконання

Повторіть дії, аналогічні описаним у п. п. 4 – 6.

У звіті з лабораторної роботи помістіть формулювання вибраного варіанту, тексти завдань, запитів на їхнє виконання і скриншоти з результатами.

Завдання для самостійного виконання

1. Додати до бази даних ***Хліб*** ще дві таблиці:

Накладні (Код_накладної, Номер_накладної, Код_виробника, Дата);

Товари_накладних (Код_товару_накладної, Код_накладної, Код_товару, Кількість).

Установити зв'язки між цими таблицями та іншими таблицями бази даних.

У нові таблиці записують дані про товари, що надходять до хлібного кіоску.

2. Для кожного з наступних завдань зробіть копію з папкою проекту та реалізуйте її.

2.1. Змінити схему бази даних, якщо магазин фірмовий, тобто продає товари тільки свого виробника (хлібозаводу).

2.2. Передбачити, щоб у базу даних записувалася інформація про продажі, яка надходить з декількох магазинів, які утворюють мережу.

2.3. Змінити схему, щоб передбачити продаж декількох видів хліба ("Український", "Родзинка", "Бородинський"), батонів ("Молочний", "Київський", "Нарізний") тощо; тобто товари розподіляються за групами, наприклад: хліб, батони, булки.

2.4. Для визначення в подальшому прибутку кіоску зберігати в базі даних закупівельну ціну товарів кожного виду. У різних виробників вона різна.

Примітка. У звіті за кожним завданням подайте запити на його виконання та скриншот результату їхнього виконання.

Висновки

Мовні засоби призначені для спілкування користувача різних класів з банками даних. Вони слугують для опису різних компонентів банку даних, а іноді – зовнішніх щодо банку елементів, що знаходяться з ним у безпосередній взаємодії, та для звернення до потрібних частин банку даних.

Функціональні можливості підтримуваної засобами СУБД моделі даних стають доступними для кінцевих користувачів, розробників додатків, системного персоналу адміністрування даних завдяки втіленню її у вигляді комплексу мовних засобів, які підтримує СУБД. В існуючих СУБД застосовуються різні способи реалізації таких мов.

Існують три основні форми SQL: інтерактивна (Interactive), статична (Static) і динамічна (Dynamic). Функціонують вони однаково, але використовуються по-різному.

У кожного об'єкта в базі даних є унікальне ім'я. Імена використовують у командах SQL і вказують, над яким об'єктом бази даних команда повинна виконати дію. Основними іменованими об'єктами в реляційній базі є таблиці, стовпці та користувачі. Правила їхнього іменування були визначені ще в стандарті SQL1.

Від типу даних залежить спосіб зберігання значень і виконання операцій над ними. Будь-які дані, якими оперують програми, належать до певних типів.

У стандарті SQL1 описано лише мінімальний набір типів даних, які можна використати для подання інформації в реляційній базі даних. Вони практично підтримуються в усіх комерційних СУБД. Стандарт SQL2 додав у цей перелік рядки змінної довжини, значення дати та часу тощо.

Розділ 3. Мова DML SQL та огляд її можливостей

5.1. Мова DML SQL. Основні команди та їх призначення

Мова маніпулювання даними SQL – Data Manipulation Language (DML) SQL включає оператори, що управляють змістом таблиць бази даних. Основні команди DML мови SQL наведені в табл. 3.1.

Таблиця 3.1

Основні команди DML мови SQL

Оператори	Опис
SELECT	Вибирає дані з бази
INSERT	Додає нові рядки в таблицю
DELETE	Видаляє рядки з таблиці
UPDATE	Модифікує інформацію у базі даних

Команди DML є найважливішим для користувача та найскладнішими в SQL. Вони призначені для вибірки даних з таблиць, додавання, зміни та видалення інформації. Тобто вони реалізують два з основних призначень бази даних – надавати інформацію користувачеві та змінювати її в базі даних. Оператори DML завжди виконуються над деякими таблицями, що входять в базу даних.

SQL-команда (оператор) SELECT – одна з найважливіших у базах даних. Основне завдання оператора – вилучення даних з таблиць згідно з умовою, яке поставив користувач. Сучасні інформаційні системи оперують настільки великим числом записів, що продуктивність SQL-оператора SELECT – є однією з найважливіших характеристик СУБД.

Результатом виконання оператора SELECT завжди є таблиця. Таким чином, за результатами дій оператор SELECT схожий на оператори реляційної алгебри. Будь-який оператор реляційної алгебри може бути виражений відповідним чином сформульованим оператором SELECT. Складність оператора SELECT у тому, що він містить усі можливості реляційної алгебри, а також додаткові можливості, яких в реляційній алгебрі немає.

Якщо цей оператор був введений в інтерактивному режимі взаємодії з базою даних, то результат відображається у вигляді таблиці в поточному діалоговому вікні.

Якщо оператор SELECT виконується з програми на іншій мові програмування, то формується результівний набір, що розміщується в пам'яті програми або сервера БД, а потім додаток отримує дані з результівного набору в свої змінні.

За допомогою кожного оператора модифікації інформації в базі даних (INSERT, DELETE, UPDATE) за одноразове виконання можна змінити дані тільки в одній таблиці, проте в деяких системах ці зміни можуть стосуватися інформації з інших таблиць або інших баз даних. Виконати оператори модифікації даних зазвичай дозволено не всім користувачам. Власник бази даних або власники окремих її об'єктів можуть дозволити або заборонити користувачам виконувати певні команди її модифікації.

5.2. Засоби пошуку даних

3.2.1. Основні конструкції мови, призначені для вибирання даних

Оператор SELECT складається з декількох пропозицій, кожна з яких виконує певні дії. Порядок проходження пропозицій і їх формат чітко визначені, але тільки кілька з них є обов'язкові, решту можна використовувати в разі потреби.

Загальний вигляд SQL-оператора SELECT:

```
SELECT [ALL | DISTINCT] визначення_полів
   FROM визначення таблиць
     [WHERE умова]
          [GROUP BY умова _групування [HAVING умова]]
                    [ORDER BY умова _сортування]
```

Необов'язкова частина виділяється квадратними дужками.

Оператор SELECT може містити *шість ключових слів* (або речень):

SELECT – визначає набір полів, які потрібно отримати в результаті запиту (ALL – повторювані рядки присутні, DISTINCT – відсутні);

FROM – установлює набір таблиць, звідки витягується результат;

WHERE – визначає умову отримання даних;

GROUP BY – задає умову угруповання даних;

HAVING – визначає умову всередині угруповання;

ORDER BY – задає поля і порядок сортування.

Ключове слово SELECT оператора SELECT визначає набір полів, які будуть формувати результівну таблицю.

Як поля можуть бути задані:

поля таблиць;

найпростіше вираження або SQL-вираз;

агрегатна функція, де в якості аргументу використовується поле або SQL-вираз.

Установити поля можна використати такі *нотації*:

"*" – усі поля таблиць, які беруть участь у запиті;

ім'я_поля – позначає поле з ім'ям "ім'я_поля";

ім'я_таблиці.ім'я_поля – позначає поле з ім'ям "ім'я_поля" таблиці "ім'я_таблиці" (або псевдонім.імя_поля);

SQL-вираз – задає вираз, куди входять поля таблиць і константи.

3.2.2. Вирази, умови та оператори

Кожен виробник СУБД визначає термін "SQL-вираз" по-своєму. Відповідно до стандарту ANSI SQL поняття SQL-вираз визначено так:

вирази з арифметичними операціями "плюс", "мінус", "помножити", "розділити", де в якості аргументів використовуються поля таблиць БД або константи;

SQL-функція – як аргумент застосовуються поле таблиці, константа або їх комбінація з допомогою арифметичних операцій.

Використовуються також комбінації з арифметичними операціями полів таблиць і викликів SQL-функцій.

Будь-яка сучасна СУБД містить велику кількість стандартних функцій, які можуть застосовуватися в операторі SELECT з отриманням даних з БД. Використання таких функцій в деяких випадках дуже ефективне, оскільки економить час для розроблення ПЗ і значно прискорює процес отримання кінцевого результату внаслідок того, що SQL-оператор SELECT виконується на сервері зі значно більшою потужністю, ніж робоча станція користувача.

Припустимо, що необхідний результат формується шляхом обчислення синуса над збережуваними значеннями. Можна витягти дані з таблиці, а потім виконати на клієнтському комп'ютері операцію синус. Однак немає необхідності це робити, тому що в SQL-операторі SELECT замість імені поля можна поставити SQL-вираз "SIN (ім'я_поля)" і відразу отримати необхідні значення.

Усі SQL-функції класифікують таким чином:

скалярні – для обробки одного даного;

агрегатні – для обробки безлічі даних.

Скалярні функції розподіляють так:

робота з числами;

робота з символами;

робота з датою і часом;

явне перетворення типів даних;

шифрування/дешифрування даних;

решта.

У табл. 3.2 наведені основні математичні функції MS SQL Server.

Таблиця 3.2

Математичні функції

Функція	Призначення
1	2
ABS (n)	Повертає абсолютне значення n
ACOS (n)	Повертає арккосинус n
ATAN (n)	Арктангенс n
ATN2	Арктангенс кута, описаного двома кутами
CEILING (n)	Найближче ціле, що перевищує n
COS (n)	Косинус n
COT (n)	Котангенс кута n
DEGREES	Перетворює значення кута із радіанів у градуси
FLOOR (n)	Найближче ціле, менше n
LOG	Логарифм на підставі 2
LOG10	Логарифм на підставі 10
POWER(m,n)	Повертає m в ступені n
RADIANS	Перетворення градусів на радіани
RAND	Генератор випадкових чисел
ROUND	Округлення чисел
COT (n)	Котангенс кута n
SIGN	Повертає знак вираження
SIN (n)	Синус n

Закінчення табл. 3.2

1	2
SQRT (n)	Корінь квадратний від n
SQUARE	Зведення в квадрат
TAN (n)	Тангенс n

Приклади SQL-виразів:

x + y − z; SIN(x + y);

SIN(x) + COT (y);

ABS(x); TAN(x);

SIN(y) + COS(x) + SQRT(y).

У розглянутих прикладах x, y і z – поля деякої таблиці бази даних, SIN і COS – функції отримання синуса та косинуса.

У табл. 3.3 наведені основні рядкові функції.

Таблиця 3.3

Рядкові функції

Функції	Призначення
LEFT	Відбирає символи з лівого кінця рядка
LEN	Повертає довжину символьного рядка
LOWER	Перетворює рядок на нижній регістр
LTRIM	Видаляє початкові пробіли з рядка
REPLACE	Замінює задані фрагменти рядка іншим рядком, функція REPLACE ('qwt', 'w', 'e') поверне рядок qet
RIHGT	Відбирає символи з правого кінця рядка
RTRIM	Видаляє кінцеві пробіли з рядка
UPPER	Перетворює рядок у верхній регістр
+	Операція конкатенації рядків

Найпростіший SQL-оператор SELECT завжди містить в своєму складі ключове слово FROM, яке задає таблиці, що беруть участь у запиті.

У SQL-операторі SELECT може бути задано стільки таблиць, скільки необхідно для виконання поточного завдання вилучення даних. Це дозволяє створювати ефективні оператори SELECT, які спрощують роботу з обробки даних.

Форма задання однієї таблиці:

```
FROM ім'я_таблиці [псевдонім]
```

Якщо необхідно отримати дані з декількох таблиць, то їх подають через кому:

```
FROM ім'я_таблиці1 [псевдонім1], ім'я_ таблиці 2[псевдонім2], ...
```

Для кожної таблиці можна визначити її псевдонім, який діє в рамках усього SQL-оператора SELECT. Ім'я користувача дозволяє скоротити визначення полів, оскільки для проектування бази правильним вважається задання імені таблиці, що відображає сутність збережених у ній даних.

Розглянемо приклади виконання операторів SQL на основі навчальної бази даних "Хліб" (рис. 3.1).

Припустимо, що необхідно зберігати інформацію про продукти або виробників продуктів у базі даних "Хліб". Логічно назвати таблиці іменами "Товари" або "Виробники".

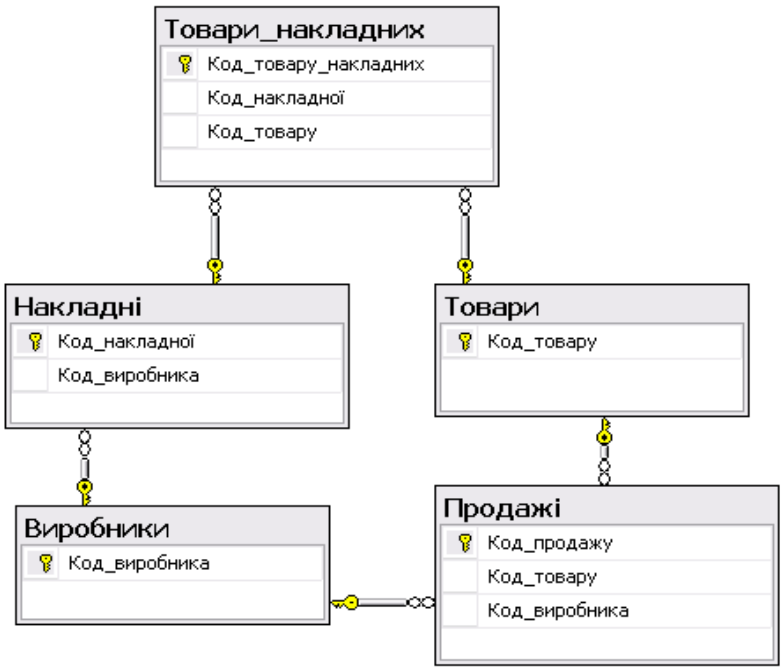


Рис. 3.1. **Діаграма навчальної бази даних "Хліб"**

З подальшим використанням імен SQL-оператор SELECT значно збільшується в розмірах. Це призводить до труднощів у його прочитанні і, як наслідок – до помилок. Тому для такої таблиці в рамках цього оператора вводиться псевдонім, наприклад "P":

```
SELECT P.Товар FROM Товари P
```

Оператор витягує з таблиці "Товари " назви товарів, не сортуючи їх. SQL-оператор SELECT завжди обробляє дані в тому порядку, в якому вони були збережені.

Розглянутий оператор містить деяку кількість надлишкових конструкцій. В одній таблиці не може існувати кілька полів з однаковими іменами. Отже, явна вказівка імені таблиці у заданні поля в реченні SELECT не потрібна так само, як і використання псевдоніму імені таблиці. Таким чином, наведений оператор можна записати так:

```
SELECT Товар FROM Товари
```

Обидві версії правильні.

Розглянемо використання спеціальної нотації "*" для задання полів таблиць. Символ "*", який вказаний в операторі SELECT, позначає всі поля таблиць, зазначені в реченні FROM. Таким чином, щоб отримати вміст всієї таблиці " Продукти ", достатньо записати:

```
SELECT * FROM Товари
```

Виконається перегляд усіх полів таблиці "Товари".

Така форма оператора застосовується тільки для розроблення програмного забезпечення. Під час написання програми правильно вказувати всі поля, що формують результівну таблицю SQL-оператора SELECT. Таке правило пов'язане з можливим розширенням набору полів таблиці в майбутньому. Якщо таке станеться, то неправильне задання SQL-операторів може призвести до великих труднощів у доопрацюванні програмного забезпечення. Порядок проходження полів у такому операторі визначається порядком проходження полів у SQL-операторі CREATE TABLE. Оператор SELECT дозволяє задавати кілька полів, перераховуючи їх через кому.

Приклад 3.1. Необхідно отримати інформацію про те, які є ціни на товари (поле Ціна):

```
SELECT Товар, Ціна FROM Товари
```

Приклад 3.2. Перестановка виведених стовпців:

```
SELECT Ціна, Товар FROM Товари
```

Приклад 3.3. Приклад ілюструє переваги завдання псевдонімів для таблиць запиту. Припустимо: в різних таблицях є поля з однаковими іменами (Tabl1.Name, Tabl2.Name), і запит потрібно виконати до цих таблиць. Такий запит буде помилковий:

```
SELECT Name, Name FROM Tabl1, Tabl2
```

Сервер бази даних не зможе зрозуміти, поля "name" яких таблиць необхідно отримати в результаті. У цьому випадку використовують форму явного задання імені таблиці для заданні поля в операторі SELECT:

```
SELECT Tabl1.Name,Tabl2.Name FROM Tabl1, Tabl2
```

Недолік даного оператора – довгий запис. Тут наочна перевага псевдонімів у заданні імен таблиць. Цей же оператор можна записати таким чином:

```
SELECT s.Name, g.Name FROM Tabl1 g, Tabl2 s
```

Такий оператор виглядає коротше, не втрачаючи читабельності. Не має значення те, що визначення псевдонімів для таблиць йде після речення SELECT, в якому ці псевдоніми вже використовуються. Розбір SQL-оператора SELECT СУБД виконує таким чином, що псевдоніми вже визначені на етапі перевірки речення SELECT. Тому приведений запис правильний.

Який псевдонім вибирати та для якої таблиці – справа особистого смаку та переваг. Ім'я користувача може складатися з будь-якої кількості символів. На практиці у роботі з великою базою застосовують дволітерні позначення, щоб псевдоніми відображали таблиці, а оператор не втратив читаності. У простих операторах досить і одного символу, щоб покрити всі імена таблиць запиту.

Розглянутий запит демонструє ще одну важливу властивість оператора SELECT – можливість задання і застосування декількох таблиць. У цьому випадку використані дві таблиці, але можна задати їх більше. Усе залежить від конкретної операції над даними, оскільки немає вимоги, щоб таблиці "Товари" та "Продажі" містили однакову кількість даних. Тоді яким буде результат попереднього запиту?

На рис. 3.2 – рис. 3.4 наведена інформація, яка зберігається в таблицях "Виробники", "Товари" та "Продажі".

Код_виробника	Виробник	Адреса	Телефон	Контакт
2	Х/з "Кулиничи"	м.Харків, вул. Салтівська, 123	368-62-95	Петров Петр Петрович
3	Х/з "Салтівський"	м.Харків, вул. Салтівська, 8	362-58-55	Сидоров Іван Іллічь

Рис. 3.2. **Інформація, що зберігається в таблиці "Виробники"**

Код_товару	Товар	Ціна	Ціна_закупівлі
1	Хліб "Український"	3,00	2,50
2	Батон "Молочний"	2,80	2,50
3	Булка з маком	2,00	1,80

Рис. 3.3. **Інформація, що зберігається в таблиці "Товари"**

Код_продажу	Дата	Код_товара	Код_виробника	Кількість
1	01.09.2012 0:00:00	1	2	200
2	01.09.2012 0:00:00	2	2	150
3	01.09.2012 0:00:00	1	3	180
4	11.11.2013 0:00:00	2	3	100
6	11.11.2011 0:00:00	2	2	777
7	11.11.2011 0:00:00	2	2	777
8	12.11.2011 0:00:00	2	2	777
9	12.11.2012 0:00:00	2	2	777
10	11.11.2012 0:00:00	2	2	777
11	11.11.2011 0:00:00	2	2	777
12	12.11.2012 0:00:00	2	2	777
13	13.11.2013 0:00:00	2	2	777
14	11.11.2011 0:00:00	1	2	777
15	12.11.2012 0:00:00	1	2	777
16	11.11.2013 0:00:00	1	2	555
18	12.12.2012 0:00:00	1	2	888
19	13.12.2013 0:00:00	1	2	999
20	14.12.2014 0:00:00	1	2	333
131	11.11.2011 0:00:00	1	2	444
132	11.11.2011 0:00:00	1	3	11

Рис. 3.4. **Інформація, що зберігається в таблиці "Продажі"**

Під час виконання SQL-оператора

```
SELECT p.Код_товару, p.Код_продажу,
          t.Код_товару Код_товар, t.Товар
                    From Товари t, Продажі p
```

база даних "склеїть" дані таблиць так, щоб в результаті створилась таблиця в першій нормальній формі. Така "склейка" виконується після багаторазового повторення даних таблиці "Товари" (3 рядки), в якій записів менше, з таблицею "Продажі" (20 рядків), в якій записів більше. Таким чином, знайдений результат виконання даного оператора Select містить 60 рядків (рис. 3.5).

Проте результат не відповідає вимозі щодо отримання інформації про те, який відбувся продаж і якого товару. Насправді результат роботи цього оператора може бути зовсім інший. Для виведення інформації про те, продаж якого товару здійснений, потрібно явним чином за допомогою речення WHERE задати умову зв'язку таблиць "Товари" та "Продажі". Тоді оператор дійсно виведе ту інформацію, яка необхідна.

Код_товару	Код_продажу	Код_товар	Товар
1	14	1	Хліб "Український"
1	15	1	Хліб "Український"
1	16	1	Хліб "Український"
1	18	1	Хліб "Український"
1	19	1	Хліб "Український"
1	20	1	Хліб "Український"
1	131	1	Хліб "Український"
1	132	1	Хліб "Український"
1	1	2	Батон "Молочний"
2	2	2	Батон "Молочний"
1	3	2	Батон "Молочний"
2	4	2	Батон "Молочний"
2	6	2	Батон "Молочний"
2	7	2	Батон "Молочний"
2	8	2	Батон "Молочний"

16 of 60 Cell is Read Only.

Рис. 3.5. **Результат виконання запиту**

Оператор SELECT допускає застосування в цільовому списку обчислюваних полів. Якщо потрібно знайти значення будь-якого вираження, навіть не пов'язаного з БД, то можна використовувати оператор SELECT.

Приклад 3.4. SELECT SIN (1) – повертає значення 0,8414709848078965....

Розглянемо ще одну опцію задання полів в операторі SELECT.

З цією метою розв'яжемо задачу обчислення суми кожного продажу товарів. Для цього кожний елемент стовпця "Кількість" таблиці "Продажі" необхідно помножити на елемент стовпця "Ціна" таблиці "Товари". Тоді сформуємо запит до бази даних:

```
SELECT k.Код_товару, k.Код_продажу, t.Код_товару AS Код_товар,
          t.Товар, t.Ціна * k.Кількість AS Сума
                                FROM  Товари t, Продажі k
```

Виникає таке питання: яке ім'я буде мати поле, що подає результати обчислень? Для того щоб задати ім'я нового стовпчика, потрібно вказати в реченні SELECT ім'я для поля явно за допомогою виразу:

```
t.Ціна * k.Кількість AS Сума
```

Тепер у стовпчику "Сума" будуть виведені результати обчислень (рис. 3.6).

Код_товару	Код_продажу	Код_товар	Товар	Сума
1	14	1	Хліб "Український"	2331,00
1	15	1	Хліб "Український"	2331,00
1	16	1	Хліб "Український"	1665,00
1	18	1	Хліб "Український"	2664,00
1	19	1	Хліб "Український"	2997,00
1	20	1	Хліб "Український"	999,00
1	131	1	Хліб "Український"	1332,00
1	132	1	Хліб "Український"	33,00
1	1	2	Батон "Молочний"	560,00
2	2	2	Батон "Молочний"	420,00
1	3	2	Батон "Молочний"	504,00
2	4	2	Батон "Молочний"	280,00
2	6	2	Батон "Молочний"	2175,60
2	7	2	Батон "Молочний"	2175,60
2	8	2	Батон "Молочний"	2175,60

1 of 60 Cell is Read Only.

Рис. 3.6. **Результат виконання запиту**

Для задання імені поля за допомогою AS використовують будь-які символи, які можуть утворювати не тільки слова, а і словосполучення.

Якщо в найменуванні поля міститься символ "Пробіл", то його можна взяти в одинарні лапки. Це дозволяє записати оператор SELECT:

```
SELECT p.Код_товару, p.Код_продажу, t.Код_товару AS Код_товар,
            t.Товар, t.Ціна * p.Кількість AS 'Сума продажу'
                                    FROM   Товари t Продажі p
```

Результат виконання запиту наведений на рис. 3.7.

Код_товару	Код_продажу	Код_товар	Товар	Сума продажу
1	131	1	Хліб "Український"	1332,00
1	132	1	Хліб "Український"	33,00
1	1	2	Батон "Молочний"	560,00
2	2	2	Батон "Молочний"	420,00

1 of 60 Cell is Read Only.

Рис. 3.7. **Результат виконання запиту**

Оператор демонструє ще одну техніку запису SQL-оператора SELECT. Імовірність того, що SQL-оператор SELECT не поміститься в один рядок, досить висока. Згідно з визначенням SQL-оператор може бути багато-рядковим. Тому, якщо оператор довгий і не поміщається в один рядок, його записують у кілька рядків, використовуючи відступи для підвищення читаності самого оператора, як це роблять у звичайних програмах. Для цього речення записують один під одним, а їхні аргументи – з деяким зрушенням вліво, щоб відокремити речення від аргументів.

Розглянемо останні параметри оператора SELECT – DISTINCT або ALL. Параметр DISTINCT видаляє з результівної таблиці повторювані записи. Параметр ALL, який заданий за замовчуванням, залишає результівну таблицю в тому вигляді, в якому вона створена після склейки значень полів:

```
SELECT DISTINCT Дата
                    FROM Продажі
```

DISTINCT відстежує, які значення з'явилися в списку вихідних даних, і виключає з нього дубльовані значення. У результаті виконання запиту отримуємо дати, в які здійснювався продаж товарів (рис. 3.8).

Параметр DISTINCT можна задати тільки один раз для даного оператора SELECT. Якщо SELECT витягує безліч полів, то він виключає рядки, в яких усі вибрані поля ідентичні.

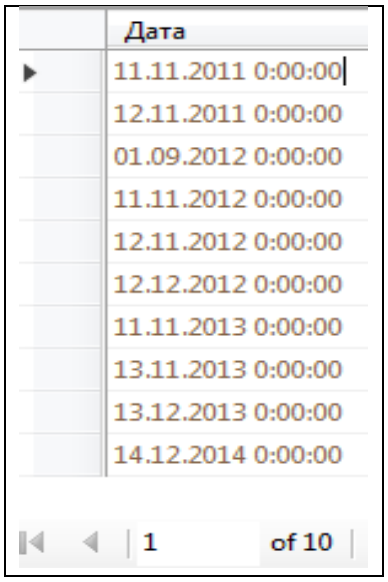


Рис. 3.8. **Результат виконання запиту**

Рядки, де деякі значення однакові, а інші – різні, включаються в результат. DISTINCT фактично діє на весь вихідний рядок, а не на окреме поле (виняток становить його застосування всередині агрегатних функцій), виключаючи можливість їх повторення. Таким чином можна зробити висновок, що будь-який SQL-оператор SELECT:

повинен мати в своєму складі речення SELECT, що задає результівні поля, і речення FROM, що визначає набір таблиць, які беруть участь у запиті;

для задання поля можна використовувати SQL-вирази та визначати для кожного поля його нове ім'я за допомогою виразу AS;

для таблиць з метою скорочення запису оператора та підвищення його читаності можна задати короткі псевдоніми, діючі в рамках усього SQL-оператора SELECT.

Розглянемо, як створювати умови в операторі SELECT. Такі дії можна реалізувати з допомогою речення WHERE.

Особливості речення WHERE:

задає умову отримання даних з таблиць. Умова може бути або істинною, або хибною для кожного рядка таблиці;

витягує тільки ті рядки з таблиці, де предикат має значення "істина";

задається зв'язок між таблицями бази (умови зв'язку), а також додаткові умови (умови вибірки), що вимагаються для успішної реалізації оператора SELECT.

Розглянемо попередній приклад з таблицями "Товари" та "Продажі". Таблиця "Продажі" має в своєму складі зовнішній ключ, який посилається на первинний ключ таблиці "Товари". Зі створенням таблиць бази даних це посилання уможливлює конкретну реалізацію оператора CREATE TABLE. СУБД не виконує автоматично зв'язку між таблицями, зв'язок потрібно задавати явно під час виконання кожного SQL-оператора SELECT. Тільки це може гарантувати правильність результату. Таким чином, розглянутий раніше оператор для визначення того, які товари продано, потрібно записати так:

```
SELECT p.Дата AS 'Дата',
       t.Товар AS 'Товар'
             FROM Товари t, Продажі p
                   WHERE t.Код_товару = p.Код_товару
```

Тут в оператор SELECT додано речення WHERE, в ньому чітко визначено, що ідентифікатор групи в таблиці "Продажі" повинен збігатися з ідентифікатором групи в таблиці "Товари". З першого погляду здається, що ця умова не потрібна. Але її наявність гарантує правильний результат, адже відсутність цієї умови дає право СУБД склеювати таблиці в порядку створення записів (рис. 3.9).

	Дата	Товар
	01.09.2012...	Хліб "Український"
▶	01.09.2012...	Батон "Молочний"
	01.09.2012...	Хліб "Український"
	11.11.2013...	Батон "Молочний"
	11.11.2011...	Батон "Молочний"
	11.11.2011...	Батон "Молочний"
	12.11.2011...	Батон "Молочний"
	12.11.2012...	Батон "Молочний"
	11.11.2012...	Батон "Молочний"
	11.11.2011...	Батон "Молочний"
	12.11.2012...	Батон "Молочний"
	13.11.2013...	Батон "Молочний"
	11.11.2011...	Хліб "Український"
	12.11.2012...	Хліб "Український"

Рис. 3.9. **Результат виконання запиту**

Наступний важливий момент цього оператора – використання полів "Код_товару" та "Код_продажу" в умовах WHERE за їх відсутності в реченні SELECT. Право звертатися до цих полів дає не перерахування їх в SELECT, а перерахування таблиць, куди вони входять, у реченні FROM. Завжди потрібно пам'ятати про те, що SELECT задає поля, які будуть в результівній таблиці, але не визначає набір полів, який можна використовувати у виразах інших реченнях SQL-оператора SELECT.

Для задання умов застосовують такі *операції відношень* (реляційні операції порівняння):

= – " дорівнює "; <> – "не дорівнює" (або ! =);

> – "більше"; < – "менше";

> = – "більше або дорівнює"; <= – "менше або дорівнює";

! > – "не менше"; ! < – "не більше".

Приклад 3.5. Запит на перегляд з бази даних інформації про "Батон "Молочний"", результат виконання якого наведений на рис. 3.10.

```
SELECT * FROM Товари
          WHERE Товар = 'Батон "Молочний"'
```

	Код_товару	Товар	Ціна	Ціна_закупівлі
▶	2	Батон "Молочний"	2,80	2,50
*	NULL	NULL	NULL	NULL

Рис. 3.10. **Результат виконання запиту**

Приклад 3.6. Запит на перегляд з бази даних інформації про товари, ціна яких менше 3 грн, результат виконання якого наведений на рис. 3.11.

```
SELECT * FROM Товари
              WHERE Ціна < 3
```

	Код_товару	Товар	Ціна	Ціна_закупівлі
▶	2	Батон "Молочний"	2,80	2,50
	3	Булка з маком	2,00	1,80

Рис. 3.11. **Результат виконання запиту**

Розглянуті умови належать до типу умов, які часто називають умовами вибірки. Стандартні булеві оператори SQL – це AND, OR, NOT. Існують інші, більш складні булеві оператори (як, наприклад, "виключне АБО"), але їх можна побудувати за допомогою трьох зазначених булевих операторів. Булева логіка "істина/неправда" є повний базис для роботи комп'ютера. Тому весь SQL можна звести до булевої логіки.

Розглянемо булеві оператори та основні принципи їх дії:

AND (операція "логічне І") – використовує два булевих параметри

(у вигляді A AND B) в якості аргументів і дає в результаті істину, якщо вони обидва правильні;

OR (операція "логічне АБО") – застосовує два булевих параметри

(у вигляді A OR B) в якості аргументів і оцінює результат як істину, якщо хоча б один з них правильний;

NOT – використовує єдиний логічний вираз (у вигляді NOT A) в якості аргументу і змінює його значення з істинного на помилкове або з помилкового на істинне.

Операції AND і OR – бінарні:

умова AND умова2;

умова OR умова2;

умова1 AND умова2 OR умова3.

Операції AND, OR і NOT можна комбінувати в одну послідовність, створюючи необхідну умову вибірки даних.

Приклад 3.7. Необхідно вибрати з таблиці "Товари" інформацію про товари, у яких ціна нижче 3 і вище 2 грн:

```
SELECT * FROM Товари
          WHERE Ціна > 2 And Ціна < 3
```

Результат виконання запиту наведений на рис. 3.12.

	Код_товару	Товар	Ціна	Ціна_закупівлі
▶	2	Батон "Молочний"	2,80	2,50
*	NULL	NULL	NULL	NULL

Рис. 3.12. **Результат виконання запиту**

Приклад 3.8. Необхідно вибрати з таблиці "Товари" інформацію про товари, у яких ціна не вище 2 грн і не нижче 3 грн. Таке завдання можна виконати декількома способами:

```
SELECT * FROM Товари
       WHERE Ціна <= 2 Or Ціна >= 3      .
Або:   SELECT * FROM Товари
       WHERE Not(Ціна > 2 And Ціна < 3)
```

В обох прикладах результат виконання запитів буде однаковим (рис. 3.13).

	Код_товару	Товар	Ціна	Ціна_закупівлі
▶	1	Хліб "Український"	3,00	2,50
	3	Булка з маком	2,00	1,80
*	NULL	NULL	NULL	NULL

Рис. 3.13. **Результат виконання запиту**

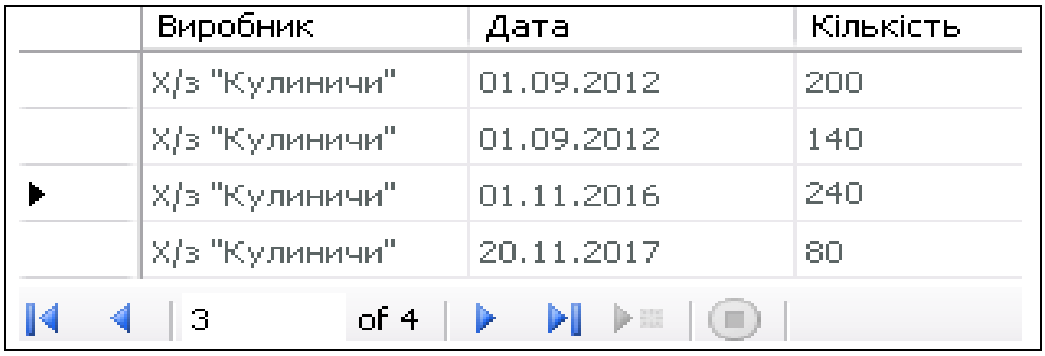
Логічну функцію Not поміщають перед реляційним оператором, на який він діє, а за необхідності розширення дії використовують дужки. У цьому випадку SQL сприймає круглі дужки. Вони означають, що все всередині них буде оцінюватися першим і оброблятися як єдине вираження за допомогою оператора, розташованого зовні.

У навчальній базі даних існують три таблиці "Виробники", "Накладні", "Товари_накладних", які пов'язані з допомогою первинних і зовнішніх ключів (див. рис. 3.1).

Складемо запит, який виводить списки кількості товару, дату накладної і виробника з ім'ям "Х\з "Кулиничі"". Товари накладної мають зв'язок з виробником побічно, через накладні. Тому SQL-оператор SELECT матиме вигляд:

```
SELECT f.Виробник AS 'Виробник', g.Дата AS 'Дата', s.Кількість
     FROM Виробники f, Накладні g, Товари_накладних s
     WHERE f.Код_виробника = g. Код_виробника AND g.Код_накладної =
                    s.Код_накладної AND f.Виробник = 'Х/з "Кулиничі"'
```

Даний оператор містить у реченні WHERE дві умови зв'язку й одну умову вибірки. Усі три умови об'єднує оператор AND (рис. 3.14).



	Виробник	Дата	Кількість
	Х/з "Кулиничи"	01.09.2012	200
	Х/з "Кулиничи"	01.09.2012	140
▶	Х/з "Кулиничи"	01.11.2016	240
	Х/з "Кулиничи"	20.11.2017	80

Рис. 3.14. **Результат виконання запиту**

Якщо кілька таблиць зв'язуються між собою, то завжди існують умови зв'язку, яких на одну менше, ніж кількість таблиць, що беруть участь у запиті. Знання цього правила дозволяє розглядати складні оператори по-іншому. У ході створення таких операторів необхідно пам'ятати, що потрібно задати умови вибірки та деяку кількість умов зв'язку. Причому умови зв'язку формуються автоматично: підраховують кількість таблиць (можна також ураховувати кількість ліній зв'язку на діаграмі) та створюють відповідні умови, об'єднавши їх шляхом операції "І". Потім припускають умови вибірки, хоча можна зробити навпаки – зосередитися на умовах вибірки, а потім автоматично додати до них умови зв'язку для групування умов. З метою зміни порядку операцій можна використовувати круглі дужки:

```
SELECT v.Виробник AS 'Виробник', n.Дата AS 'Дата', tn.Кількість
      FROM Виробники v, Накладні n, Товари_накладних tn
            WHERE (v.Код_виробника = n. Код_виробника
                  AND g.Код_накладної = tn.Код_накладної)
                                    AND v.Виробник = 'Х/з "Кулиничі"'
```

Існує ще кілька способів вирішення цієї задачі. Виходячи з визначення зовнішнього ключа та правила посилальної цілісності в поле Код_виробника таблиці "Накладні" поміщають унікальний ідентифікатор виробника, до якого належить накладна.

Отже, якщо ідентифікатор відомий, то можна вирішити поставлене завдання так:

```
SELECT v.Виробник AS 'Виробник', n.Дата AS 'Дата', tn.Кількість
      FROM Виробники v, Накладні n, Товари_накладних tn
        WHERE n.Код_накладної=tn.Код_накладної AND n.Код_виробника= 2
```

Цей оператор пов'язує лише дві таблиці, але при цьому також вирішує поставлене завдання. Очевидно, що даний запит буде виконуватися швидше, ніж попередній.

Порядок пріоритету операцій – спочатку виконується операція AND, а потім OR. У цьому прикладі застосування дужок – надмірність; але його основна мета – продемонструвати синтаксичну конструкцію використання круглих дужок.

У більшості випадків оператор SELECT ідеологічно неправильний. Користувач завжди працює з інформаційною системою через інтерфейс. Завдання інтерфейсу – спростити роботу користувача та максимально наблизити її до понять предметної області. Тому інтерфейс надає список назв товарів або виробників, а не їх первинних ключів. Ні в якому разі не можна вводити в алгоритм роботи системи правило, що первинний ключ пов'язаний з іншими полями таблиці. Це є поганою практикою у створенні запитів. Навпаки, інші поля залежать тільки від первинного ключа. Тому користувач вибирає в списку виробника "Кулиничі" й автоматично отримує оператор, який був розглянутий першим.

Обмеження на роботу речення WHERE: у реченні WHERE не можна використовувати агрегатні функції (якщо тільки не застосовується підзапит), оскільки предикати оцінюються в термінах єдиного рядка, тоді як агрегатні функції – в термінах груп рядків.

Для створення предикатів оператора WHERE використовують:

оператори порівняння (=,>, <,> =, <=, <>,! =,!>,! <);

булеві оператори AND, OR, NOT;

спеціальні оператори IN, BETWEEN, IS NULL, LIKE, ALL, ANY, SOME, EXISTS;

оператори повнотекстового пошуку:

CONTAINS і CONTAINSTABLE – для точного та неточного вилучення слів або фраз з текстових стовпців;

FREETEXT і FREETEXTTABLE – повертають смисловий контекст фраз, заданих у рядку пошуку.

Зазначені оператори застосовують для отримання виразних і потужних предикатів.

Перевірку умов можна реалізовувати з допомогою оператора IN (NOT IN). Оператор IN повністю визначає множинність якій дане значення може належати або не належати.

Приклад 3.9. Необхідно визначити назву товару з таблиці – "Товари", який має кількість 200 і 150 у таблиці "Товари_накладних":

```
SELECT T.Товар AS 'Товар', Tn.Кількість AS 'Кількість'
      FROM Товари_накладних Tn, Товари T
                WHERE T.Код_товару = Tn. Код_товару
                       AND Tn.Кількість IN (200, 150)
```

Результат виконання запиту наведений на рис. 3.15.

	Товар	Кількість
▶	Хліб "Український"	200
	Хліб "Український"	150

Рис. 3.15. **Результат виконання запиту**

Такий запит можна виконати із застосуванням операції OR:

```
SELECT T.Товар AS 'Товар', Tn.Кількість AS 'Кількість'
       FROM Товари_накладних Tn, Товари T
            WHERE (T.Код_товару=Tn.Код_товару) AND
                      (Tn.Кількість =200 OR Tn.Кількість = 150)
```

Як видно з прикладу, IN є множинність, елементи якої точно перераховуються в круглих дужках і розділяються комами. Якщо значення поля, ім'я якого зазначено зліва від IN, є одне з перерахованих у списку значень (потрібний точний збіг), то предикат вважається дійсним. Якщо елементи множинності мають символьний тип, то значення в дужках застосовуються в одиничних лапках.

Приклад 3.10. Необхідно визначити з таблиці "Виробники" інформацію про виробників, які не є 'Рошен'.

```
SELECT * FROM Виробники
     WHERE Виробник Not IN ('Рошен')
```

Результат виконання запиту наведений на рис. 3.16.

	Код_виробника	Виробник	Адреса	Телефон	Контакт
▶	2	Х/з "Кулиничи"	Тракторобудів...	368-62-95	Петров Петр П..
	3	Х/з "Салтівський"	Салтівська,12	762-58-55	Сидоров Іван І..
*	NULL	NULL	NULL	NULL	NULL

Рис. 3.16. **Результат виконання запиту**

Для перевірки умов часто використовують оператор BETWEEN (NOT BETWEEN). Оператор BETWEEN задає межі, в які повинно потрапляти значення, щоб предикат був істинним.

Синтаксис оператора BETWEEN:

```
<Ім'я_поля> [NOT] BETWEEN <початкове_значення>
                        AND <останнє_значення>
```

В якості початкового та кінцевого значень можуть використовуватися вираження і константи.

Синтаксис предиката BETWEEN у більш загальному випадку:

```
<Value_expression> [NOT] BETWEEN <low_value_expression>
                              AND <High_value_expression>
```

Фактично даний предикат – це скорочений спосіб запису виразу:

```
((<Low_value_expression> <= <value_expression>) AND
                    (<Value_expression> <= <high_value_expression>))
```

Особливості оператора BETWEEN: на відміну від IN оператор BETWEEN чутливий до порядку запису операндів – нижнє значення у реченні повинне бути першим відповідно до алфавітного або числового порядку.

Приклад 3.11. Необхідно витягти з таблиці "Товари" інформацію про товари, які коштують в діапазоні між 2 і 2,8 грн.

```
SELECT *  FROM Товари
        WHERE Ціна BETWEEN 2 AND 2.8
```

Оператор BETWEEN включає граничні значення (в даному прикладі це 2 і 2.8), які роблять предикат істинним (рис. 3.17).

	Код_товару	Товар	Ціна	Ціна_закупівлі
▶	2	Батон "Молочний"	2,80	2,50
	3	Булка з маком	2,00	1,80
*	NULL	NULL	NULL	NULL

Рис. 3.17. **Результат виконання запиту**

Іноді необхідно сформулювати граничні значення так, щоб за необхідності виключити їх. Це можна зробити, наприклад, за допомогою запиту:

```
SELECT * FROM Товари
        WHERE (Ціна BETWEEN 2 AND 2.8) AND Not Ціна IN (2)
```

Результат виконання запиту наведений на рис. 3.18.

	Код_товару	Товар	Ціна	Ціна_закупівлі
▶	2	Батон "Молочний"	2,80	2,50
*	NULL	NULL	NULL	NULL

Рис. 3.18. **Результат виконання запиту**

Аналогічно операторам порівняння – BETWEEN діє на символьних полях, представлених у двійковому (ASCII) коді, тобто для вибірки можна скористатися алфавітним порядком.

Наступний запит вибирає виробників, значення поля "Виробник" яких потрапляє в заданий алфавітний діапазон:

```
SELECT *  FROM Виробники
          WHERE Контакт BETWEEN 'С' AND 'У'
```

Результат виконання запиту наведений на рис. 3.19.

	Код_виробника	Виробник	Адреса	Телефон	Контакт
▶	3	Х/з "Салтівський"	м.Харків, вул. Салтівська, 8	362-58-55	Сидоров Іван Ілліч
*	NULL	NULL	NULL	NULL	NULL

Рис. 3.19. **Результат виконання запиту**

Для перевірки символьних або рядкових даних користуються оператором LIKE (NOT LIKE).

Синтаксис оператора LIKE:

```
<Ім'я_стовпця> [NOT] LIKE <зразок> (<рядок>
                          [NOT] LIKE <підрядок>)
```

Оператор LIKE застосовується тільки до рядкових полів типу CHAR (NCHAR) або VARCHAR (NVARCHAR), оскільки він використовується для пошуку підрядків у реченні WHERE. Отже, він здійснює перегляд рядка для перевірки умови – чи входить заданий підрядок у зазначений рядок. Для гнучкої реалізації цієї мети використовують шаблони (маски) – спеціальні символи для конструювання зразка рядка.

Існують такі типи шаблонів, які використовуються з LIKE:

символ "підкреслення" (_) – замінює будь-який одиночний символ, наприклад:

LIKE 'б_т' – зразку 'б_т' відповідає 'біт' або 'бут';

LIKE '___ст' – будь-яке слово з п'яти символів, що має в кінці слова дві букви 'ст'. Цьому шаблону відповідає слово 'крест' або 'траст';

LIKE '__ав___' – будь-яке слово з семи символів, в середині якого є пара символів 'ав'. Цьому шаблону відповідає слово 'правило';

символ "відсоток" (%) – замінює послідовність символів довільної довжини, в тому числі і нульовий, наприклад:

шаблону LIKE 'R%' відповідає будь-який рядок, що починається з букви R ('Rost');

шаблону LIKE '%T%' відповідає будь-який рядок, що містить букву T ('INTEGER');

[ ] – допускається застосування в рядку будь-якого окремого символу із зазначених у дужках (допустиме перерахування без пробілів та інших роздільників, указівка інтервалу, наприклад:

шаблону LIKE '[ae]%' відповідає будь-який рядок, що починається на a або e;

шаблону LIKE '[a-e]% int' відповідає будь-який рядок, що починається на літери a, b, c, d, e і закінчується на int;

шаблону LIKE '[JT] ed' відповідає будь-який рядок з трьох букв, що починається на J або T і закінчується на 'ed';

шаблону LIKE 'A [^ f]%' відповідає будь-який рядок, що починається на A, за якої не слідує символ f.

Приклад 3.12. Необхідно витягти з таблиці "Виробники" інформацію про виробників, в яких у стовпчику "Контакт" прізвища починаються з літери 'С'. Сформуємо такий запит:

```
SELECT * FROM Виробники
         WHERE Контакт LIKE 'С%'
```

З виконанням оператора SELECT отримуємо результат, який відображений на рис. 3.19;

Приклад 3.13. Необхідно витягти з таблиці "Виробники" інформацію про виробників, в яких у стовпчику "Контакт" прізвища не починаються з літери 'П':

```
SELECT * FROM Виробники
         WHERE Контакт NOT LIKE 'П%'
```

LIKE може бути корисним для пошуку імені або іншого значення, повне написання якого невідомо. Припустимо, що не зовсім зрозуміло, як правильно записується прізвище – 'Петров', 'Петренко' або 'Піотровський'. Можна використовувати відому частину та символи шаблону для знаходження всіх можливих варіантів:

```
SELECT * FROM Student WHERE name LIKE П%тр%'
```

Символ шаблону (%) у кінці рядка необхідний у тих реалізаціях SQL, коли довжина поля "Контакт" перевищує кількість букв в імені 'Піотровський'. У такому випадку значення поля "Контакт" реально зберігається як 'Піотровський', а за ним іде ряд пробілів.

Пошук у рядку символів підкреслення (_) або відсотка (%) виконується за допомогою Escape-символу, який використовується в предикаті

безпосередньо перед символом (%) або (_). Це означає, що наступний за ним символ інтерпретується як звичайний, а не як символ шаблону.

У SQL рядки можуть бути однакові, але не збігатися у процесі виконання оператора LIKE. Для перевірки рівнозначності рядків треба коротший рядок праворуч заповнити пробілами до довжини довшого рядка, а потім посимвольно вирівняти їх. Таким чином, рядки 'Іванов' і 'Іванов ' (з трьома пробілами) фактично збігатимуться, але предикат LIKE пробілів не додає, тому вираз 'Іванов' LIKE 'Іванов ' дасть значення false.

Щоб обійти цю проблему, можна скористатися функціями для обробки рядків LTRIM () і RTRIM () (функції TRIM () у MS SQL Server немає), вони видалять непотрібні пробіли зліва та справа від одного або обох аргументів

```
... WHERE RTRIM (Контакт) LIKE 'Іванов'.
```

Приклад 3.14. Необхідно скласти запит, який виводить списки кількості товару, дату накладної і виробника, розташованого на вулиці "Салтівська":

```
SELECT f.Виробник AS 'Виробник', g.Дата AS 'Дата', s.Кількість
          FROM Виробники f, Накладні g, Товари_накладних s
               WHERE f.Код_виробника = g. Код_виробника
                        AND g.Код_накладної = s.Код_накладної
                        AND f.Адреса LIKE 'Салтівська%'
```

Результат виконання запиту наведений на рис. 3.20.

	Виробник	Дата	Кількість
▶	Х/з "Салтівський"	01.09.2012	150
	Х/з "Салтівський"	17.11.2017	48

Рис. 3.20. **Результат виконання запиту**

Наступний запит неправильний, тому що не зазначена умова зв'язку полів v.Код_виробника, n.Код_виробника, n.Код_накладної і tn.Код_накладної в таблицях:

```
SELECT v.Виробник AS 'Виробник', n.Дата AS 'Дата', tn.Кількість
         FROM Виробники v, Накладні n, Товари_накладних tn
                  WHERE n.Код_накладної = tn.Код_накладної
                                    AND v.Адреса LIKE 'Салтівська%'
```

Виконанням цього запиту виводиться інформація про всі записи таблиці "Товари_накладних" і додається стовпчик "Виробник", в якому друкується адреса підприємства (рис. 3.21).

	Виробник	Дата	Кількість
▶	Х/з "Салтівський"	01.09.2012	200
	Х/з "Салтівський"	01.09.2012	140
	Х/з "Салтівський"	01.09.2012	150
	Х/з "Салтівський"	01.11.2016	240
	Х/з "Салтівський"	17.11.2017	48
	Х/з "Салтівський"	20.11.2017	80

Рис. 3.21. **Результат виконання запиту**

Дуже важлива особливість предикату LIKE. Предикат, який записується таким чином:

```
... name LIKE '% нов'
```

буде працювати довше, ніж предикат виду:

```
... WHERE (name LIKE '_нов') OR (name LIKE '__нов')
                              OR (name LIKE '___ новий')
```

Часто в таблиці зустрічаються записи з незаданими значеннями будь-якого з полів, тому що значення поля невідомо чи його просто немає. У таких випадках SQL дозволяє вказати в полі NULL-значення:

коли значення поля є NULL, це означає що програма бази даних спеціальним чином позначає поле, що може не мати будь-якого значення для цього рядка;

оскільки NULL не є значенням як таким, він не має типу даних;

NULL може розміщуватися в поле будь-якого типу.

Для визначення невідомого значення застосовується оператор IS з ключовим словом NULL.

Приклад 3.15. Необхідно визначити всі рядки таблиці "Товари_ накладних", де відсутнє значення в стовбці "Кількість":

```
SELECT * FROM Товари_накладних
         WHERE Кількість is not NULL
```

Разом з операторами IS NULL можуть бути використані булеві оператори AND, OR і NOT.

3.2.3. Вибирання з кількох таблиць

Сучасні бази даних зазвичай нормалізовані, оскільки дані розподіляються по численним невеликим таблицям з метою усунення сповторюваних даних, скорочення обсягу займаної пам'яті, підвищення продуктивності та забезпечення цілісності даних. Однак практично завжди потрібно витягувати дані з декількох таблиць. Об'єднання (з'єднання багатотабличного запиту) повністю призначені для забезпечення вибірки даних з декількох таблиць і включення цих даних в один результівний набір.

Одна з найбільш важливих рис запитів SQL полягає в їх здатності визначати зв'язки між безліччю таблиць і відображати в них інформацію в термінах цих зв'язків. Таку операцію називають **об'єднанням** (**з'єднанням**), це одна з найбільш значущих операцій для реляційних баз даних. За допомогою з'єднань безпосередньо пов'язується інформація, що міститься в декількох таблицях або між окремими частинами однієї таблиці. У багатотабличному запиті через операцію об'єднання в реченні FROM імена таблиць подані через кому.

Об'єднання таблиць можна виконати *трьома способами*:

за допомогою встановлення зв'язків між таблицями в предикаті запиту (в умові, організованій реченням WHERE);

за допомогою оператора об'єднання JOIN;

з використанням речень UNION, INTERSECT і EXCEPT.

Перший спосіб найбільш простий і широко використовується для з'єднання даних кількох таблиць.

За допомогою конструкції Join реалізується внутрішнє і зовнішнє об'єднання. Другий спосіб називають внутрішнім об'єднанням (з'єднанням), що повертає тільки ті рядки, для яких умова об'єднання приймає значення True. Таблиці, які беруть участь у внутрішньому об'єднанні, рівноправні.

Розглянемо приклад з'єднання двох таблиць за допомогою встановлення зв'язків. У ньому з таблиць "Товари" та "Товари_накладних" необхідно вибрати об'єднану інформацію про товари та їх кількість:

```
SELECT t.Товар, tn.Кількість
       FROM Товари t, Товари_накладних tn
            WHERE t.Код_товару = tn.Код_товару
```

У ході виконання SQL-оператора SELECT будуть склеєні поля "Товар" таблиці "Товари" та "Кількість" таблиці "Товари_накладних" і результат, який відображений на рис. 3.20. Таке об'єднання таблиць називають об'єднанням за допомогою відносини предок – нащадок. У цьому випадку між таблицями створюється відношення "один-до-багатьох". SQL не вимагає, щоб асоційовані стовпці двох (або більше) таблиць обов'язково були парою "первинний ключ – зовнішній ключ". Будь-які два стовпці з двох таблиць можуть бути пов'язаними стовпцями, тільки якщо вони мають порівнянні типи даних. Такі стовпчики створюють між таблицями відношення "багато-до-багатьох". Шляхом виконання багатотабличного запиту SQL досліджують кожну комбінацію рядків двох або більше можливих таблиць і перевіряють ці комбінації за їх предикатами. Якщо комбінація значень рядків дає таке значення, яке робить предикат істинним, то з комбінації цих рядків будуть обрані значення для виведення.

Об'єднання, що використовує предикати, засновані на рівності, називають **еквіоб'єднанням** (об'єднанням за рівністю). Еквіоб'єднання – найбільш поширений тип об'єднання. Фактично в об'єднанні можна використовувати будь-який оператор порівняння.

Об'єднання дозволяє зробити складні порівняння між будь-якими полями будь-якого числа таблиць і використовувати отримані результати для того, щоб вирішувати, яка інформація потрібна.

SQL дозволяє об'єднання двох (або більше) копій окремої таблиці (самооб'єднання, тета-з'єднання або рефлексивне з'єднання). Така дія означає, що будь-який рядок таблиці (один в кожен момент часу) можна комбінувати з її копією та з будь-яким іншим рядком цієї ж таблиці. Кожна така комбінація оцінюється в термінах предиката, як і в разі з'єднання декількох різних таблиць. Це дозволяє легко створювати певні види зв'язків між різними записами всередині одної таблиці, наприклад, здійснювати пошук пар рядків із загальним значенням поля.

Об'єднання таблиці зі своєю копією можна подати таким чином: реально копія таблиці не створюється, але SQL виконує команду як реальну. Такий тип об'єднання не відрізняється від звичайного з'єднання двох таблиць, за винятком того, що в даному випадку таблиці ідентичні.

У разі об'єднання таблиці зі своєю копією з метою посилання до стовпців необхідно мати два різних імені для тієї самої таблиці. Це виконується за допомогою тимчасових імен – псевдонімів, визначених у реченні FROM.

Приклад 3.16. Необхідно визначити товари, які мають однакову ціну закупівлі:

```
SELECT  T.Товар, S.Товар AS Товар2, S.Ціна_закупівлі
        FROM Товари F, Товари S
                  WHERE T.Ціна_закупівлі = S. Ціна_закупівлі
```

Результат цього запиту зображений на рис. 3.22.

	Товар	Товар2	Ціна_закупівлі
▶	Хліб "Українськ...	Хліб "Український"	2,50
	Батон "Молочний"	Хліб "Український"	2,50
	Хліб "Українськ...	Батон "Молочний"	2,50
	Батон "Молочний"	Батон "Молочний"	2,50
	Булка з маком	Булка з маком	1,80

Рис. 3.22. **Результат виконання запиту**

У розглянутому прикладі SQL поводиться так, ніби з'єднує дві різні таблиці з іменами F і S. Тобто псевдоніми дозволяють обробляти ту саму

таблицю незалежно. Логіка цього запиту така: з таблиці "Продажі" вибирається черговий рядок і запам'ятовується під першим псевдонімом. Після цього SQL почне перевіряти її в комбінації з кожним рядком таблиці "Продажі" під іншим псевдонімом. Якщо комбінація рядків задовольняє предикату, то відповідні поля з неї вибираються для виведення.

Недолік даного запиту – виведення має два значення для кожної комбінації прізвищ у різному порядку. Це пов'язано з тим, що поточне значення в першому псевдонімі спочатку вибирається в комбінації зі значенням в другому псевдонімі, а потім – навпаки, крім того кожен рядок порівнюється сам з собою.

Спосіб усунення недоліку – накладення умови порядку на два значення так, що одне значення може стати менше другого або передує йому в алфавітному порядку. Це зробить предикат асиметричним щодо зв'язку, тому ті ж самі значення в зворотному порядку не будуть обрані знову. Тому попередній запит потрібно модифікувати таким чином:

```
SELECT F.Товар, S.Товар AS Товар2, S.Ціна_закупівлі
          FROM Товари F, Товари S
            WHERE F.Ціна_закупівлі = S.
                      AND F.Товар < S.Товар
```

Результат такого запиту зображений на рис. 3.23.

	Товар	Товар2	Ціна_закупівлі
▶	Батон "Молочний"	Хліб "Український"	2,50

Рис. 3.23. **Результат виконання запиту**

Особливості роботи з псевдонімами:

можуть використовуватися в операторі SELECT до їх оголошення в реченні FROM;

псевдоніми таблиць можуть збігатися з іменами таблиць;

допустиме використання будь-якого числа псевдонімів для однієї таблиці в запиті (хоча більше двох псевдонімів у запиті – надмірність);

SQL відхилить запит, якщо псевдоніми далі не будуть визначені в реченні FROM;

в операторі SELECT можна не використовувати кожен псевдонім або таблицю, які згадувалися в реченні FROM запиту;

в одному запиті можна змішувати написання імен таблиць і псевдонімів;

псевдонім існує тільки під час виконання команди, а після завершення запиту псевдоніми запиту більше не мають значення;

псевдоніми застосовують для спрощення запису довгих і складних імен таблиць і для мінімізації синтаксичних помилок.

Таким чином, операція об'єднання з'єднує інформацію з двох (або більше) таблиць (однакових або різних), формуючи з них пари пов'язаних рядків. Об'єднану таблицю утворюють пари тих рядків з різних таблиць, де в пов'язаних стовпцях містять однакові значення.

Часто трапляються ситуації, коли необхідно, щоб результат містив усі або деякі рядки з однієї таблиці, які не задовольняють умові з'єднання. У цьому разі застосовують так звані зовнішні з'єднання, які розподіляють на ліві та праві [19]. Зовнішні з'єднання розглянемо з використанням речення JOIN.

За допомогою конструкції Join реалізується внутрішнє об'єднання (Inner Join або просто Join) і зовнішнє об'єднання (Left | Right | Full Outer | Cross Join). Зовнішнє об'єднання повертає всі рядки з однієї головної таблиці та тільки ті рядки з іншої підпорядкованої таблиці, для яких умова об'єднання приймає значення True. Рядки другої таблиці, що не задовольняють умові об'єднання (тобто мають значення False), отримують значення Null і в результівному наборі не виводяться.

У стандарті ANSI SQL-92 умови об'єднання записуються в реченні From відповідно до такого синтаксису:

```
From <таблиця> [Inner | Left | Right | Full Outer | Cross]
               Join <таблиця> On <умова пошуку>
```

В 'умовах пошуку' можуть бути задані будь-які критерії порівняння рядків двох об'єднуваних таблиць, у тому числі з використанням булевих операторів.

Усі різновиди конструкції Join мають одну загальну відмінну рису: в них один рядок узгоджується з одним або декількома іншими рядками для отримання результівного рядка, що є множинністю, створеною шляхом з'єднання полів з декількох рядків.

У з'єднанні розрізняються два боки – лівий і правий. Лівим вважається таблиця, зазначена в реченні From першою, а правим – таблиця, зазначена другою.

Конструкція Inner Join реалізує внутрішнє об'єднання і повертає рядки, узгоджені за всіма полями, позначеними як використовувані для з'єднання. Внутрішнє з'єднання є винятковим, тобто будь-який рядок, для якого немає відповідності в обох таблицях, неминуче виключається з остаточного варіанту результівного набору.

Існує чотири **види зовнішнього об'єднання**:

ліве (Left Join) – запит повертає всі рядки з лівої таблиці, а з правої – тільки рядки, що задовольняють умові з'єднання (рядки правої таблиці, що не задовольняють умові об'єднання, отримують значення Null і в результівному наборі не виводяться);

праве (Right Join) – зворотні процеси щодо лівого зовнішнього об'єднання;

повне (Full Outer Join) – запит повертає всі рядки лівої і правої таблиць, але в особливому порядку;

перехресне (Cross Join) – запит повертає всі рядки лівої таблиці та перший рядок правої таблиці, всі рядки лівої таблиці та другий рядок правої таблиці тощо.

У разі лівого з'єднання у результат запиту включаються усі рядки з лівої таблиці (таблиця є провідною), тобто з таблиці, яка знаходиться ліворуч в умові з'єднання. Ці рядки з'єднуються з рядками правої таблиці – у разі виконання умови з'єднання. В іншому випадку стовпці правої таблиці будуть мати значення NULL.

У правому з'єднанні, навпаки, провідною таблицею є та, ім'я якої вказано в конструкції JOIN праворуч.

Синтаксис команд, що реалізують зовнішні з'єднання, може відрізнятися в різних СУБД.

Приклад 3.17. Необхідно отримати відомості про те, які виробники за якими накладними продавали товари. Для виконання цього запиту спочатку додамо в таблицю "Виробники" один рядок – ще одного виробника (рис. 3.24). Слід зауважити, що в таблиці "Накладні" даних про цього виробника немає.

	Код_виробника	Виробник	Адреса	Телефон	Контакт
	2	Х/з "Кулиничи"	Тракторобудів...	368-62-95	Петров Петр П...
	3	Х/з "Салтівський"	Салтівська,12	762-58-55	Сидоров Іван І...
	4	Рошен	Науки, 9А	702-18-31	Костін А.В.
▶*	NULL	NULL	NULL	NULL	NULL

Рис. 3.24. **Інформація, що зберігається в таблиці "Виробники"**

Виконання запиту (варіант 1) – внутрішнє з'єднання (рис. 3.25).

```
SELECT DISTINCT Виробники.Виробник, Код_накладної
            FROM Виробники INNER JOIN Накладні ON
                 Виробники.Код_виробника =Накладні.Код_виробника
```

	Виробник	Код_накладної
▶	Х/з "Кулиничи"	1
	Х/з "Салтівський"	2
	Х/з "Кулиничи"	3
	Х/з "Салтівський"	4
	Х/з "Кулиничи"	5

Рис. 3.25. **Результат виконання запиту**

Виконання запиту (варіант 2) – зовнішнє з'єднання (рис. 3.26).

```
SELECT DISTINCT Виробники.Виробник, Код_накладної
      FROM Виробники LEFT JOIN Накладні ON
           Виробники.Код_виробника =Накладні.Код_виробника
```

	Виробник	Код_накладної
	Рошен	*NULL*
▶	Х/з "Кулиничи"	1
	Х/з "Кулиничи"	3
	Х/з "Кулиничи"	5
	Х/з "Салтівський"	2
	Х/з "Салтівський"	4

Рис. 3.26**. Результат виконання запиту**

У другому варіанті з'явився один додатковий рядок для виробника "Рошен". Згідно з даними таблиці "Накладні" продавались товари тільки двох виробників (крім "Рошен"). Тому в першому випадку вибираються тільки ті рядки з таблиці "Виробники", для яких значення табельного номеру співпадає хоча б з одним значенням табельного номера у таблиці "Накладні".

У другому випадку виконується зовнішнє ліве з'єднання. Оскільки ліворуч в умові з'єднання записано ім'я таблиці "Виробники", то з неї вибираються усі рядки. Через те, що "Рошен" відсутній в таблиці "Накладні", для нього у полі Код_накладної буде значення NULL.

Зазначимо, що результат правого з'єднання (RIGHT JOIN) для цього прикладу співпадає з внутрішнім (INNER JOIN), оскільки будуть вибрані всі накладні та товари за кожною накладною.

Формування багатотабличних запитів можна виконувати з допомогою речень UNION, INTERSECT і EXCEPT.

Речення UNION використовують для:

для розміщення декількох запитів разом;

об'єднання їх виведення.

Речення UNION об'єднує виведення двох або більше запитів у єдиний набір рядків і стовпців.

Приклад 3.18. Необхідно отримати список усіх товарів і виробників, назви яких укладені між буквами 'У' і 'Ц':

```
SELECT Виробник, 'Виробник' AS 'Таблиця'
       FROM  Виробники
             WHERE (Виробник BETWEEN 'У' AND 'Ц')
UNION
       SELECT Товар, 'Товар' AS 'Таблиця'
             FROM  Товари
                   WHERE     (Товар BETWEEN 'У' AND 'Ц')
```

Результат такого запиту зображений на рис. 3.27.

	Виробник	Таблиця
▶	Х/з "Кулиничи"	Виробник
	Х/з "Салтівський"	Виробник
	Хліб "Український"	Товар

Рис. 3.27. **Результат виконання запиту**

У наведеному прикладі якщо не буде псевдоніма 'Таблиця', то SQL Server присвоїть цьому стовпцю ім'я No column name.

Особливості виведення:

у MS SQL Server заголовок стовпців формується першим запитом;

під час об'єднання стовпці повинні бути сумісні для об'єднання. Тобто необхідно використовувати в запиті однакове число стовпців у однаковому порядку;

обов'язкова сумісність типів. Тобто якщо порожні значення NULL заборонені в будь-якому стовпці об'єднання, то вони повинні бути заборонені і в усіх інших стовпцях;

речення UNION автоматично виключає дублікати рядків з виведення (за виключенням UNION ALL);

ключове слово DISTINCT не дозволяється;

можна вставляти константи та вирази в речення UNION (повинна бути сумісність констант і виразів);

речення ORDER BY розміщується тільки в кінці запиту;

речення ORDER BY застосовується і для упорядкування вихідних даних об'єднання. Задання упорядкування здійснюється за номером стовпця:

```
SELECT Виробник, 'Виробник' AS 'Таблиця'
       FROM  Виробники
             WHERE (Виробник BETWEEN 'У' AND 'Ц')
UNION
       SELECT Товар, 'Товар' AS 'Таблиця'
              FROM  Товари
                    WHERE     (Товар BETWEEN 'У' AND 'Ц')
 ORDER BY 2 DESC
```

Результат такого запиту зображений на рис. 3.28.

	Виробник	Таблиця
▶	Хліб "Український"	Товар
	Х/з "Кулиничи"	Виробник
	Х/з "Салтівський"	Виробник

Рис. 3.28. **Результат виконання запиту**

У розглянутому прикладі виконане впорядкування за убуванням за значеннями поля "Таблиця".

Можна впорядкувати вихідні дані відповідно до значень одного або декількох полів: для кожного з полів незалежно від зростання або

спадання (ASC або DESC) (так само, як і для вихідних даних одного запиту). Число 2 в реченні ORDER BY задає номер стовпця в упорядкованому списку оператора SELECT.

Якщо об'єднання виконується більше ніж для двох запитів, то для впорядкування обчислень потрібно використовувати круглі дужки (круглі дужки можуть визначати порядок усунення дубліката). Інакше, замість того щоб поставити query X UNION query Y UNION query Z, необхідно конкретизувати або (query X UNION query Y) UNION query Z, або query X UNION (query Y UNION query Z).

Це потрібно тому, що UNION і UNION ALL можна комбінувати для виключення одних дублікатів без усунення інших. Речення (query X UNION ALL query Y) UNION query Z не обов'язково генерує ті ж вихідні дані, що і речення query X UNION ALL (query Y UNION query Z), якщо є дублікати рядків, що підлягають виключенню з вихідних даних.

Оператор INTERSECT (перетин) – це оператор SQL, який виводить однакові рядки з першого, другого та наступних наборів даних. Тобто він виведе тільки ті рядки, які є як в першому результівному наборі, так і в другому (третьому і так далі). Тобто відбувається перетин цих рядків.

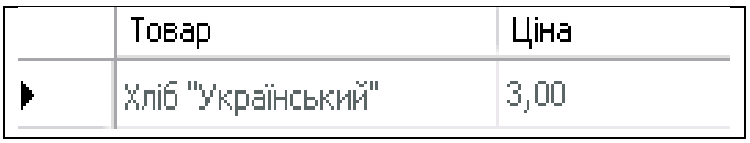
Такий оператор дуже корисний, якщо необхідно дізнатися, які рядки є і в першій таблиці, і в другій (наприклад, для визначення повторення даних).

Також як і у операторів UNION у INTERSECT є правила, наприклад те, що кількість полів у всіх результівних наборах повинно бути однаковим, як і їх тип даних.

Приклад 3.19. Необхідно отримати назву товару та його ціну, яка вище 2,5 грн і який продавався пізніше липня 2017р. (перетин двох підзапитів – ключове слово INTERSECT):

```
SELECT Товари.Товар
       FROM Товари
            WHERE Товари.Ціна >= 2.5
INTERSECT
            SELECT Товари.Товар
                   FROM Товари, Продажі
                        WHERE Товари.Код_товару = Продажі.Код_товару
                                      AND  Продажі.Дата > '2017/07/31'
```

Результат такого запиту зображений на рис. 3.29.



	Товар	Ціна
▶	Хліб "Український"	3,00

Рис. 3.29. **Результат виконання запиту**

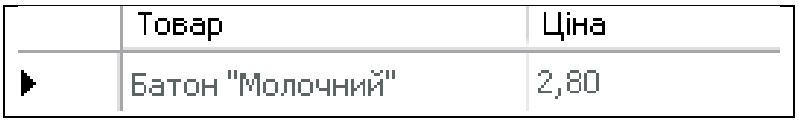
Оператор EXCEPT (різниця) – це оператор SQL, який виводить тільки ті дані з першого набору рядків, яких немає в другому наборі.

Тут діють правила, як у оператора INTERSECT, тобто кількість стовпців (і їх тип) має бути однаковою. EXCEPT корисний, якщо необхідно порівняти дві таблиці та вивести тільки ті рядки першої таблиці, яких немає в іншій таблиці.

Приклад 3.20. Необхідно отримати назву товару та його ціну, яка вище 2,5 грн, за виключенням тих товарів, що продавалися пізніше липня 2017р. (різниця двох підзапитів – ключове слово EXCEPT):

```
SELECT Товари.Товар, Товари.Ціна
         FROM Товари
              WHERE Товари.Ціна >= 2.5
EXCEPT
         SELECT Товари.Товар, Товари.Ціна
                FROM Товари, Продажі
                     WHERE Товари.Код_товару = Продажі.Код_товару
                                  AND Продажі.Дата > '2017/07/31'
```

Результат такого запиту зображений на рис. 3.30.



	Товар	Ціна
▶	Батон "Молочний"	2,80

Рис. 3.30. **Результат виконання запиту**

Таким чином, отримуємо рядок з першого набору рядків, якого немає в другому наборі.

3.2.4. Використання агрегатних функцій. Фраза GROUP BY, HAVING, ORDER BY

Агрегатні функції SQL (табл. 3.4) призначені для виконання дій над безліччю значень, які групуються GROUP BY SQL-оператора SELECT.

Результат роботи функції – значення, отримане шляхом обробки множинності. Усі функції, за винятком COUNT, не враховують у своїй роботі поля зі значенням NULL і мають загальну форму подання:

```
ім'я_функції] ([DISTINCT | ALL] вираз)
```

За замовчуванням перед виразом розміщується ALL, яка стверджує, що в обробку потрапляють усі значення множини.

Таблиця 3.4

Функції агрегування (агрегатні функції)

Функція	Призначення
AVG AVG ([ALL | DISTINCT] <вираз>)	Функція AVG повертає середнє арифметичне значень, поданих у параметрі <вираз>. Параметр expression повинен містити числові значення. NULL-значення ігноруються
COUNT COUNT ([ALL | DISTINCT] < вираз > | *)	Функція COUNT повертає дані типу int про кількість елементів, поданих в параметрі <вираз>. Параметр не може належати до типу даних uniqueidentifier, text, ntext або image. З використанням значення параметра * відбувається повернення даних про кількість рядків у таблиці; дубльовані значення або NULL-значення не виключаються
COUNT_BIG COUNT_BIG ([ALL | DISTINCT] < вираз > | *)	Повертає дані про кількість елементів в групі. Аналогічна функції COUNT, але має тип даних biqint
GROUPING GROUPING(<column_ name>)	Функція GROUPING додає додатковий стовпець до виводу оператора SELECT
MAX MAX ([ALL | DISTINCT] < вираз >)	Функція MAX повертає максимальне зі значень, поданих у параметрі <вираз>. З обчисленням функції MAX усі NULL-значення ігноруються
MIN MIN ([ALL | DISTINCT] < вираз >)	Функція MIN повертає мінімальне зі значень, поданих у параметрі <вираз>. З обчисленням функції MIN усі NULL-значення ігноруються
STDEV STDEV (<вираз >)	Функція STDEV повертає результат обчислення середньоквадратичного відхилення за всіма значеннями, поданими в параметрі <вираз>. При обчисленні функції STDEV усі NULL-значення ігноруються
SUM SUM ([ALL | DISTINCT] < вираз >)	Функція SUM повертає суму всіх значень, поданих у параметрі <вираз>. При обчисленні функції SUM все NULL-значення ігноруються
VAR VAR (<вираз >)	Функція VAR повертає результат обчислення дисперсії за всіма значеннями, поданими у параметрі <вираз>

Особливості запитів з агрегатними функціями:

з функціями SUM і AVG використовуються тільки числові поля, а з COUNT, MAX і MIN – числові або символьні;

функція COUNT (*) вважає кількість значень в стовпці або число рядків в таблиці;

допускається застосування агрегатних функцій з аргументами, які складаються з виразів, що включають в себе одне або більше полів, при цьому команда DISTINCT забороняється;

не можна використовувати агрегатну функцію від агрегатної функції;

у списку стовпців що повертаються не можна одночасно вказувати агрегатні функції і прості імена стовпців (так як стовпець створює таблицю, а агрегатна функція повертає тільки одне значення, то це викличе помилку);

речення GROUP BY дозволяє визначати підмножину значень в полі в термінах іншого поля і застосовувати функцію агрегату до такої підмножини. Це дає можливість об'єднувати поля і агрегатні функції в єдиному реченні SELECT.

Приклад 3.21. Необхідно підрахувати кількість видів товарів, що були продані:

```
SELECT COUNT (DISTINCT Код_товару) FROM Продажі
```

Приклад 3.22. Необхідно підрахувати загальну кількість рядків в таблиці "Товари":

```
SELECT COUNT (*) FROM Товари
```

COUNT (*) включає записи з NULL-значеннями, а також дублікати, тому DISTINCT в даному випадку не може бути використаний.

Приклад 3.23. Необхідно знайти максимальну кількість проданих товарів:

```
SELECT MAX (Кількість) FROM Продажі
```

Розглянемо особливості роботи агрегатних функцій з полями зі значенням NULL:

якщо будь-які зі значень, що містяться в стовпці, дорівнюють NULL, то з обчисленням результату функції вони виключаються;

якщо всі значення в стовпці дорівнюють NULL, то функції AVG (), SUM (), MIN (), MAX () повертають значення NULL; функція COUNT () повертає нуль;

якщо в стовпці немає значень (тобто стовпець порожній), то функції AVG(), SUM(), MIN(), MAX() повертають значення NULL; функція COUNT () повертає нуль;

функція COUNT(*) підраховує кількість рядків і не залежить від наявності або відсутності в стовпці значень NULL. Якщо рядків у стовпці немає, то ця функція повертає нуль.

Особливості роботи з агрегатними функціями та DISTINCT:

якщо використовується DISTINCT і агрегатна функція, то її аргументом може бути тільки ім'я стовпця, а не вираження;

у функціях MIN(), MAX() також немає сенсу використовувати DISTINCT. У функції COUNT() можна застосовувати DISTINCT, але це зустрічається рідко;

до функції COUNT(*) взагалі не може бути застосовано DISTINCT, оскільки вона просто підраховує число рядків;

в одному запиті речення DISTINCT можна застосовувати тільки один раз. Якщо воно застосовується з аргументом агрегатної функції, то його вже не можна використовувати з жодним іншим аргументом.

Припустимо, що необхідно розв'язати таку задачу: в таблиці "Товари_накладних" міститься кількість проданих товарів (поле "Кількість"), а назва товарів (поле "Товар") – у таблиці "Товари". Як отримати середню кількість кожного проданого товару? Оператор SELECT, який розпочинає розв'язувати цю задачу, має такий вигляд:

```
SELECT g.Товар, s.Кількість
        FROM Товари g, Товари_накладних s
             WHERE g.Код_товару = s.Код_товару
```

З виконанням SQL-оператора SELECT будуть склеєні поля "Товар" таблиці "Товари" та "Кількість" таблиці "Товари_накладних". Отриманий результат розміщений на рис. 3.31.

	Товар	Кількість
	Хліб "Український"	200
	Батон "Молочний"	140
	Хліб "Український"	150
	Батон "Молочний"	240
	Хліб "Український"	48
▶	Батон "Молочний"	80

Рис. 3.31. **Результат виконання запиту**

Розглянемо послідовно роботу цього оператора. Зауважимо, що ті самі значення поля "Товар" таблиці "Товари" багаторазово повторюються, надаючи можливість для групування даних. Усередині проміжної таблиці (див. рис. 3.20) з'являються групи даних, об'єднані ім'ям групи. Це дозволяє зробити над значеннями поля "Кількість", які не є умовою угруповання, певні арифметичні дії, результатом яких є один рядок. До таких арифметичних дій належать операції обчислення кількості записів, суми значень, середнього арифметичного, мінімуму, максимуму тощо. Функція AVG розраховує середнє арифметичне. Оператор, який вирішує цю задачу, має вигляд:

```
SELECT t.Товар AS 'Групи товарів',
   AVG (tn.Кількість) AS 'Середня кількість'
        FROM Товари t, Товари_накладних tn
             WHERE t.Код_товару = tn.Код_товару
                                  GROUP BY t.Товар
```

З виконанням цього запиту отримаємо результат, який відображений на рис. 3.32.

	Групи товарів	Середня кількість
▶	Батон "Молочний"	153
	Хліб "Український"	132

Рис. 3.32. **Результат виконання запиту**

Речення GROUP BY SQL-оператора SELECT є механізмом угруповання полів за певною умовою. GROUP BY дозволяє знаходити підмножини значень окремого поля (s.Кількість) у термінах іншого поля (g.Товар) і застосовувати функції агрегування. Це дає можливість комбінувати поля і агрегатні функції в одному реченні SELECT.

Приклад 3.24. Необхідно визначити найбільшу кількість кожного проданого за один день товару. Кількість проданого товару за один день записана в таблиці "Продажі", а запит буде виконуватись шляхом групування у стовпчику "Товар" таблиці "Товари":

```
SELECT t.Товар, MAX(p.Кількість) AS 'Максимальна кількість'
          FROM Товари t, Продажі p
               WHERE t.Код_товару = p.Код_товару
                                        GROUP BY t.Товар
```

Результат такого запиту зображений на рис. 3.33.

	Товар	Максимальна кількість
	Батон "Молочний"	777
	Булка з маком	486
▶	Хліб "Український"	942

Рис. 3.33. **Результат виконання запиту**

Якщо потрібно задати кілька полів, за якими виконується угруповування, **умови** мають бути такими:

вони повинні бути обов'язково задані в реченні SELECT;

подані через кому в реченні GROUP BY;

інші поля речення SELECT, які не беруть участі в угруповуванні, обов'язково повинні бути задані в якості параметрів агрегатних функцій.

Якщо задане в реченні SELECT поле не є частиною умови угруповування і параметром агрегатної функції, то такий оператор вважається помилковим і не виконується.

У роботі з реченням GROUP BY можна стикнутися з певними труднощами. Основна помилка – неправильне виділення поля або полів угруповання. Щоб правильно їх виділити, потрібно подумки виконати

перші дві фази оператора SELECT – склеювання полів і обробку умови WHERE.

Якщо речення GROUP BY не задане, то автоматично проводиться групування за всіма полями, поданими в реченні SELECT. Це дає можливість виконати агрегатні функції над усім вмістом оператора SELECT так, щоб кожен запис складав єдину групу. Це необхідно в тих випадках, коли потрібно, наприклад, підрахувати загальну кількість записів у таблиці:

```
SELECT COUNT (*) FROM Товари
```

COUNT (*) включає записи з NULL-значеннями, а також дублікати, тому DISTINCT у даному випадку не може бути використаний.

Розглянемо способи використання речення HAVING.

Приклад 3.25. Необхідно визначити найменшу кількість кожного товару, проданого за накладними, якщо вона не перевищує 100:

```
SELECT g.Товар, MIN (s.Кількість) AS 'Мінімальна кількість'
        FROM Товари g, Товари_накладних s
             WHERE g.Код_товару = s.Код_товару
                    GROUP BY g.Товар
                           HAVING MIN(Кількість)<=100
```

Результат такого запиту зображений на рис. 3.34.

	Товар	Мінімальна кількість
	Батон "Молочний"	80
▶	Хліб "Український"	48

Рис. 3.34. **Результат виконання запиту**

HAVING можна використовувати з предикатом IN, наприклад:

```
….. HAVING MIN(Ціна) IN (3, 2))
```

Використовуючи речення HAVING, необхідно враховувати таке:

HAVING задає умову всередині групи, визначеної за допомогою речення GROUP BY;

має посилатися тільки на агрегатні функції і поля, обрані в GROUP BY;

HAVING може мати тільки аргументи, що мають єдине значення для групи вихідних даних, тому буде заборонена така команда:

```
SELECT Код_товару,  MIN (Ціна)   FROM Продажі
            GROUP BY Код_товару HAVING DATE= 01/09/2012
```

Наведемо правильний спосіб виконання розглянутого запиту:

```
SELECT Код_товару, MIN (Ціна)
       FROM Продажі
                WHERE DATE = 01/09/2012 GROUP BY Код_товару
```

Для завдання умов використовують ті ж операції, що і в реченні WHERE. Відмінність полягає в тому, що речення WHERE виконується перед угрупованням. Отже, з його допомогою неможливо поставити умову, що використовує результати угруповування. В іншому спосіб завдання і робота пропозиції HAVING нічим не відрізняються від речення WHERE.

Розглянемо способи використання речення ORDER BY. У SQL-операторі SELECT дані не сортуються автоматично. Для завдання набору полів і порядку їх сортування існує речення ORDER BY.

Синтаксис запису запиту з використанням ORDER BY:

```
ORDER BY ім'я_поля1 [ASC | DESC] [, ім'я_поля2 [ASC | DESC], ...]
(To ascend – підніматися, to descend – спускатися)
```

Особливості речення ORDER BY:

параметр "ім'я_поля" задає поле оператора SELECT, за яким виконується сортування;

поля, де відбувається упорядкування, треба вказувати в операторі SELECT;

ключові слова ASC (за зростанням) і DESC (за убуванням) визначають порядок (напрям) сортування. За замовчуванням сортування виконується за зростанням (ASC);

якщо потрібне сортування не в одному, а в декількох полях, то вони перераховуються через кому, причому для кожного поля можна задати напрям сортування;

якщо задано кілька полів, то поле, розташоване першим, має перевагу перед полем, заданим після нього;

під час сортування імена полів можна задавати іменами стовпців або номерами стовпців.

Приклад 3.26. Необхідно вивести список товарів, відсортований за зростанням імен:

```
SELECT Товар FROM Товари
        ORDER BY Товар ASC
```

У розглянутому випадку ASC можна не вказувати. Результат запиту зображений на рис. 3.35.

	Товар
	Батон "Молочний"
	Булка з маком
▶	Хліб "Український"

Рис. 3.35. **Результат виконання запиту**

Приклад 3.27. Необхідно впорядкувати Інформацію з таблиці з даними про накладні за зменшенням дати накладної, а для накладних, які мають однакову дату, – в порядку убування їх номерів:

```
SELECT * FROM Накладні
        ORDER BY Дата DESC Номер_накладної DESC
```

Результат такого запиту зображений на рис. 3.36.

	Код_накладної	Номер_накладної	Дата	Код_виробника
	5	313	20.11.2017	2
	4	212	17.11.2017	3
	3	211	01.11.2016	2
	2	201	01.09.2012	3
▶	1	101	01.09.2012	2

Рис. 3.36. **Результат виконання запиту**

Для вказівки полів, за якими впорядковуються вихідні дані, можна використовувати їх порядкові номери. Це номери стовпців у визначенні вихідних даних оператора SELECT (а не стовпців у базовій таблиці). Тобто перше поле в операторі SELECT для речення ORDER BY визначене полем з номером 1 незалежно від його розташування в базовій таблиці.

Якщо в поле для упорядкування вихідних даних існують NULL-значення, то всі вони розташовані в кінці або передують всім іншим значенням цього поля. Конкретний варіант не обумовлений стандартом ANSI. ORDER BY можна застосовувати з GROUP BY для впорядкування груп. Пропозиція ORDER BY завжди виконується останньою.

Приклад 3.28. Необхідно знайти найменшу кількість проданого виробником товару, впорядкувати його за назвою виробника:

```
SELECT Виробник, MIN (Кількість)
          FROM Продажі P, Виробники V
          WHERE P.Код_виробника = V.Код_виробника
                    GROUP BY Виробник
                              ORDER BY Виробник
```

Результати виконання цього запиту наведені на рис. 3.37:

	Виробник	Мінімальна кількість продаж
	Х/з "Кулиничи"	150
▶	Х/з "Салтівський"	11

Рис. 3.37. **Результат виконання запиту**

До цього вихідні дані були згруповані, але порядок груп довільний; тепер групи вибудовані в певній послідовності. Оскільки в команді ORDER BY не вказано спосіб упорядкування, то за замовчуванням застосовується висхідний порядок.

5.3. Підзапити та їх види

3.3.1. Основні мовні конструкції, що зустрічаються у процесі використання підзапитів

Вкладений запит (підзапит) – це запит, який міститься в вираженні речення WHERE або HAVING іншого запиту для визначення додаткових

обмежень на дані, що виводяться. *Підзапит* у запиті використовують для накладання умов на виведені дані. Підзапити можуть застосовуватися з операторами SELECT, INSERT, UPDATE або DELETE.

Підзапити необхідні для розміщення в запитах умов, точні дані для яких невідомі, тим самим розширюючи можливості та гнучкість SQL. Щоб оцінити зовнішній запит, спочатку SQL повинен оцінити вкладений запит усередині речення WHERE.

Приклад 3.29. Необхідно скласти запит для визначення дат і кількості продажів товару "Булка з маком", якщо відома тільки назва товару, але невідомо значення поля ""Код_товару" (зовнішній ключ таблиці "Продажі") для нього. Щоб витягти дані про всі дати продажів цього товару, можна записати такий запит:

```
SELECT Дата, Кількість FROM Продажі
       WHERE Код_товару =
                    (SELECT Код_товару FROM Товари
                           WHERE Товар = 'Булка з маком')
```

Результат такого запиту зображений на рис. 3.38.

	Дата	Кількість
▶	11.11.2012 0:00:00	486
	14.12.2014 0:00:00	333

Рис. 3.38. **Результат виконання запиту**

Принципи роботи запиту SQL із зв'язаним підзапитом:

вибирається рядок з таблиці "Продажі", ім'я якої зазначене в зовнішньому запиті;

виконується підзапит; отримане значення застосовується для аналізу цього рядка в умові пропозиції WHERE зовнішнього запиту;

за результатом оцінювання цієї умови ухвалюється рішення про включення або невключення рядка до складу вихідних даних;

процедура повторюється для наступного рядка таблиці зовнішнього запиту.

Наведений запит коректний тільки в тому випадку, якщо в результаті виконання зазначеного в дужках підзапиту повертається єдине значення. Якщо в результаті виконання підзапиту буде повернуто кілька

значень, то підзапит помилковий. У даному прикладі це станеться, якщо в таблиці STUDENT буде кілька записів зі значеннями поля Товар = 'Булка з маком'.

Таким чином, необхідно, щоб для операцій відношень (>, <,> =, <=, <>) підзапит повертав одне і тільки одне значення. Якщо в результаті виконання підзапиту створюється кілька значень, то оцінити предикат основного запиту на предмет істинності чи хибності можна. Проте це призведе до оцінювання запиту як помилкового.

У деяких випадках для гарантії отримання одного значення в результаті виконання підзапиту застосовується DISTINCT.

Правила для складання підзапитів:

підзапити необхідно укласти в круглі дужки;

підзапит може посилатися тільки на один стовпець у вираженні ключового слова SELECT (за винятком випадків, коли в головному запиті використовується порівняння з кількома стовпцями з підзапиту);

у підзапиті не можна застосовувати речення ORDER BY, а тільки GROUP BY;

підзапит, який повертає кілька рядків даних, можна використовувати тільки в операторах, що допускають безліч значень (наприклад, у IN);

у списку ключового слова SELECT не допускаються посилання на значення типу BLOB, ARRAY, CLOB або NCLOB;

підзапит не можна безпосередньо застосовувати як аргумент, що допускає безліч значень функції;

операцію BETWEEN до підзапитів застосовувати не можна, але її можна використовувати в самому підзапиті;

предикати в підзапитах – незворотні. Тобто умови, що містять підзапити, використовують конструкцію <вираз> <оператор> <підзапит> і в жодному разі не "підзапит> <оператор> <вираз> або <підзапит> <оператор> <підзапит>;

оператор IN (на відміну від операторів BETWEEN, LIKE, IS NULL) широко застосовується в підзапитах. IN визначає набір значень, які тестуються на збіг з іншими значеннями для визначення істинності предиката. Коли використовується IN з підзапитом, SQL просто формує цей набір з виведення підзапиту;

у будь-якій ситуації, коли застосовується оператор рівності (=), краще використовувати IN. На відміну від запиту з оператором рівності (=), IN не може змусити запит потерпіти невдачу, якщо підзапитом вибрано більше, ніж одне значення;

команда SELECT * ... не може використовуватися в підзапиті;

один з видів функцій, які автоматично видають у результаті єдине значення для будь-якої кількості рядків, – агрегатні функції;

підзапити можна застосовувати для використання речення GROUP BY всередині пропозиції HAVING.

Приклад 3.30. Необхідно вибрати дати про всі продажі товарів, кількість яких вище середньої кількості продажів (це запит із зв'язаним підзапитом):

```
SELECT Дата FROM Продажі
        WHERE Кількість >
                        (SELECT AVG (Кількість) FROM Продажі)
```

Результат запиту зображений на рис. 3.39.

	Дата
	11.11.2011
	11.11.2011
	12.11.2011
	12.11.2012
	11.11.2011
	12.11.2012
	13.11.2013
	11.11.2011
	12.11.2012
	12.12.2012
	13.12.2013

Рис. 3.39. **Результат виконання запиту**

Приклад 3.31. Потрібно визначити кількість записів у таблиці "Продажі", у яких кількість проданих товарів з номером зовнішнього ключа 2 перевищує середні продажі цього товару (запит із зв'язаним підзапитом):

```
SELECT COUNT(DISTINCT Код_продажу) 'Кількість_записів',
       Кількість Кількість_продажу   FROM   Продажі
       GROUP BY Кількість
               HAVING (Кількість >
                        (SELECT AVG(Кількість)
                              FROM Продажі
                                   WHERE(Код_товару = 2)))
```

Результат такого запиту зображений на рис. 3.40.

	Кількість_записів	Кількість_продажу
	9	777
	1	888
▶	1	942

Рис. 3.40. **Результат виконання запиту**

Інший варіант розв'язання задачі:

```
SELECT COUNT(DISTINCT Код_продажу) 'Кількість_записів',
             Кількість Кількість_продажу
             FROM   Продажі
                 WHERE Кількість > (SELECT AVG (Кількість)
                       FROM Продажі
                             WHERE Код_товару = 2)
                                 GROUP BY Кількість
```

Приклад 3.32. Необхідно отримати список виробників, які продавали товар "Булка з маком". Запит виконується з використанням предиката IN:

```
SELECT Виробник, 'Булка з маком' AS Товар
     FROM  Виробники
     WHERE (Код_виробника IN
           (SELECT DISTINCT Код_виробника
                 FROM  Продажі
                 WHERE (Код_товару IN
                       (SELECT DISTINCT Код_товару
                               FROM  Товари
                               WHERE (Товар = 'Булка з маком')))))
```

У цьому випадку вкладені підзапити можуть повертати таблицю, що містить кілька рядків. Результат виконання складного вкладеного запиту буде еквівалентний результату такої **послідовності дій**:

виконати один раз вкладений підзапит і отримати код товару 'Булка з маком';

просканувати таблицю "Продажі" для отримання списку кодів виробників;

за отриманими кодами визначаються виробники товару 'Булка з маком'.

Результат такого запиту зображений на рис. 3.41.

	Виробник	Товар
▶	Х/з "Салтівський"	Булка з маком

Рис. 3.41. **Результат виконання запиту**

У створенні вкладених запитів можуть виникати такі **помилки**:

коли підзапит повертає більше одного рядка з використанням оператора порівняння одиничних значень;

внутрішній запит може не містити жодного рядка.

Приклад 3.33. Необхідно скласти запит для визначення дат і кількості продажів товарів, ціна яких перевищує 2 грн:

```
SELECT Дата, Кількість FROM Продажі
       WHERE Код_товару =
                  (SELECT Код_товару FROM Товари
                                     WHERE Ціна > 2)
```

Для виконання запиту формується таке повідомлення про помилку:

```
Error Source: .Net SqlClient Data Provider
Error Message: Subquery returned more then 1 value.
```

Причина помилки полягає в тому, що підзапит повертає більше одного рядка, а потрібно визначити тільки одне значення поля "Код_товару".

Приклад 3.34. Необхідно скласти запит для визначення дат і кількості продажів товарів, відсутніх у таблиці "Товари":

```
SELECT Дата, Кількість FROM Продажі
       WHERE Код_товару =
                  (SELECT Код_товару FROM Товари
                            WHERE Товар = 'Калач')
```

Результат запиту зображений на рис. 3.42.

	Дата	Кількість
▶*	NULL	NULL

Рис. 3.42. **Результат виконання запиту**

У цьому випадку важко визначитись, у якій частині складного запиту виникла помилка.

3.3.2. Кореляційні підзапити

Кореляційний (співвіднесений, пов'язаний) **підзапит** – це підзапит, що залежить від інформації, що надається головним запитом. Коли в SQL використовуються підзапити, у внутрішньому запиті можна.

посилатися на таблицю, ім'я якої зазначено в пропозиції FROM зовнішнього запиту, тим самим формуючи пов'язаний підзапит (або співвіднесений підзапит – correlated subquery). У цьому випадку підзапит виконується повторно, по одному разу для кожного рядка таблиці з основного запиту. Кореляційні підзапити належать (через складність їх оцінок) до найбільш тонких понять у SQL.

Однак кореляційні підзапити – це засоби, що виконують дуже складні функції за досить компактних команд. Процедура, яка виконується кореляційним підзапитом, повинна відповідати таким **правилам**:

вибір поточного рядка з таблиці в зовнішньому запиті;

збереження значення поточного рядка в псевдонімі, який визначений у FROM зовнішнього запиту;

виконання підзапиту. Для цього всюди, де знайдений псевдонім із зовнішнього запиту, використовується значення з поточного рядка (його називають зовнішнім посиланням);

оцінювання предиката зовнішнього запиту на підставі результатів підзапиту;

описана послідовність повторюється для наступного рядка з таблиці зовнішнього запиту, поки всі рядки з таблиці не будуть перевірені.

Цю задачу можна вирішити за допомогою операції об'єднання таблиць. Можна використовувати підзапити, що зв'язують таблицю з її копією.

Приклад 3.35. Необхідно знайти всі оцінки з дисциплін зі значеннями вище середніх за цими дисциплінами:

```
SELECT * FROM Продажі P1
       WHERE Кількість > (SELECT AVG(Кількість)
                  FROM Продажі P2
                       WHERE P2. Код_виробника = P1.Код_виробника)
```

Результат такого запиту зображений на рис. 3.43. Слід мати на увазі, що реальний час виконання запиту значно залежить від оптимізатора запитів конкретної СУБД.

	Код_продажу	Дата	Код_товару	Код_виробника	Кількість
	10	11.11.2012 ...	3	2	486
▶	16	11.11.2013 ...	1	2	555
	20	14.12.2014 ...	3	2	333
	131	11.11.2011 ...	1	2	444
	1	01.09.2012 ...	1	2	200
	2	01.09.2012 ...	2	2	150
	132	11.11.2011 ...	1	3	11

Рис. 3.43. **Результат виконання запиту**

Речення GROUP BY дозволяє групувати записи, що виводяться запитом за значенням деякого поля. Використання речення HAVING дозволяє через виведення здійснювати фільтрацію таких груп. Предикат речення HAVING оцінюється не для кожного рядка результату, а для кожної групи вихідних записів, сформованої реченням GROUP BY зовнішнього запиту.

Приклад 3.36. Необхідно за даними з таблиці "Продажі" визначити загальну кількість проданих товарів, згрупувавши значення за датами та виключивши ті дні, коли число продажів, було менше 2:

```
SELECT Дата, SUM (Кількість) 'Загальна кількість' FROM Продажі P1
        GROUP BY Дата
                HAVING 1<(SELECT COUNT (Кількість)
                          FROM Продажі P2
                               WHERE P1.Дата = P2.Дата)
```

Результат такого запиту зображений на рис. 3.44.

	Дата	Загальна кількість
	11.11.2011 0:...	3563
▶	01.09.2012 0:...	530
	12.11.2012 0:...	2331
	11.11.2013 0:...	655

Рис. 3.44. **Результат виконання запиту**

Підзапит обчислює кількість рядків з тією самою датою, що збігається з датою, для якої сформована чергова група основного запиту. Пов'язані підзапити за своєю сутністю близькі до з'єднань (JOIN): обидві конструкції включають перевірку кожного запису однієї таблиці з кожним записом іншої таблиці або з псевдонімом з тієї ж таблиці, при цьому більшість операцій схожі.

Хоча кореляційні підзапити виконуються циклічно, по разу на кожен рядок головного запиту, немає підстави вважати, що корельовані підзапити менш ефективні, ніж некорельовані.

Корельований підзапит – це один із способів "читання" всіх рядків таблиці та порівняння значень їх полів зі зв'язаними з ними даними. Такий підзапит використовується, коли він повинен повертати особливий результат або набір результатів для кожного рядка-кандидата з зовнішнього запиту. Тобто, корельований підзапит застосовується тоді, коли результат виконання підзапиту залежить від значення, яке повідомляється йому кожним окремим рядком зовнішнього запиту.

Внутрішній запит виконується один раз для кожного рядка-кандидата з зовнішнього запиту.

5.4. Засоби маніпулювання даними

3.4.1. Додавання рядків до таблиці. Оператор INSERT

SQL-оператор INSERT використовують для вставки даних у таблицю БД. Оператор INSERT має такий синтаксис:

```
INSERT INTO <ім'я_таблиці | уявлення>
     [(Ім'я_поля1, ім'я_поля2, ...)]
         VALUES <(значення1, значення2, ...)>|<Підзапит>
```

Існує дві форми оператора INSERT:

перша дозволяє вставляти в таблицю нові рядки шляхом безпосереднього задання значення кожного поля;

друга базується на даних, які передаються в оператор INSERT за допомогою запиту на вибірку даних SELECT.

Перша форма оператора INSERT має такий синтаксис:

```
INSERT INTO <ім'я_таблиці | подання>
          [(Ім'я_поля1, ім'я_поля2, ...)]
              VALUES <(значення1, значення2, ...)>
```

Особливості роботи оператора INSERT:

вставляють тільки один запис у таблицю з ім'ям "ім'я_таблиці", яка попередньо повинна бути визначена оператором CREATE TABLE;

якщо поля для вставки взагалі не вказані, то їх послідовність задається оператором CREATE TABLE. Значення в списку полів для вставки вводяться в таблицю в тому порядку, в якому вони записані в команді;

якщо в операторі задані не всі поля для вставки, то у відсутні поля поміщається значення "NULL". Водночас потрібно враховувати можливість наявності умови NOT NULL;

щоб привласнити значення не всім полям запису або змінити порядок введення значень, слід явно задати список полів після імені таблиці.

Приклад 3.37. Необхідно вставити в таблицю "Виробники" (Код_виробника, Виробник, Адреса, Телефон, Контакт) новий запис – 12, Рошен, Науки 9А.

```
INSERT INTO Виробники (Виробник, Адреса, Телефон, Контакт)
       VALUES ('Рошен', 'Науки 9А', '702-18-31', 'Костін А.В.')
```

Повідомлення про виконання запиту наведено на рис. 3.45.

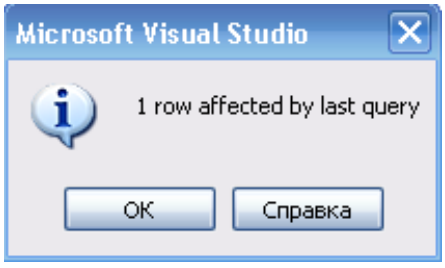


Рис. 3.45. **Результат виконання запиту**

У таблиці "Виробник" поле "Код_виробника" є автоінкрементним (рис. 3.46), тому в запиті відсутнє його значення (воно додається автоматично). Якщо тип даних поля "Код_виробника" буде VARCHAR, то в поле Код_виробника буде автоматично вставлено значення NULL. Тому рекомендується вказувати поля таблиці що вставляються явно. Це пов'язано з тим, що послідовність полів, яку задає оператор CREATE TABLE, можна просто забути. Існує також можливість додавання нових полів у таблицю після створення бази даних.

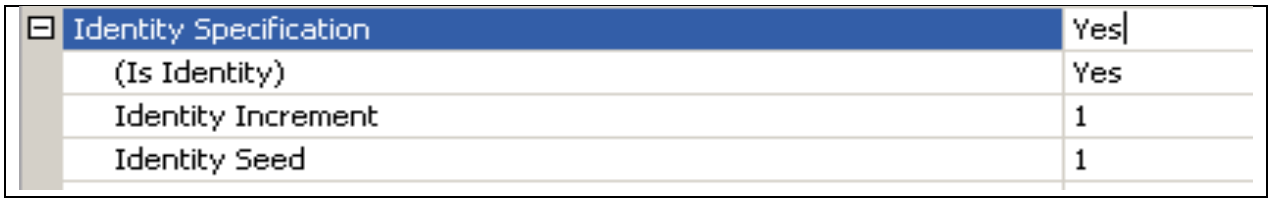


Рис. 3.46. **Специфікація поля "Код_виробника"**

Розглянемо вставку дати / часу в таблицю бази даних.

Перетворення рядка в дату для вставки може виконуватися двома способами:

неявно – привласнюючи символьний рядок значенням змінної дати – часу;

явно – за допомогою вбудованих функцій Cast і CONVERT (кращий варіант).

Приклад 3.38. Вставка в таблицю "Продажі" інформації з датою продажу (типу datetime).

```
INSERT INTO Продажі(Дата, Код_товару, Код_виробника, Кількість)
                VALUES('2017/08/21', 1, 3, 543)
```

Якщо дата додається неявно, то додатково це можна виконати одним із таких способів:

```
VALUES(… , '2017-08-21') -- (формат 'YYYY-MM-DD').
VALUES(… , '2017.08.21') -- (формат 'YYYY.MM.DD').
VALUES(… , '08.21.2017') -- (формат 'MM.DD.YYYY').
VALUES(… , '08.21.17') -- (формат 'MM.DD.YY').
```

Вставка дати в форматі VALUES (..., '05.08.21 ') буде виконана, але дасть результат 2021-05-08 00:00:00, тобто '08.05.21' – це формат 'месяц.день.год '.

Помилка перетворення (Conversion failed when converting character string to smalldatetime data type.) виникне в тому випадку, якщо буде записано таке:

```
VALUES(… , '05 19 2011 ')
```

Речення VALUES задає в круглих дужках значення, які будуть поміщені в таблицю, або значення, які формуються за допомогою підзапиту. Обов'язкова вимога сумісності типів даних полів і типів даних значень, які в них містяться.

Значення, які задають у VALUES, – скалярні. Вони не можуть бути обчислені виразами. Команда INSERT не передбачає вивід даних користувачу.

Друга форма оператора вставки INSERT має такий синтаксис:

```
INSERT INTO <ім'я_таблиці | подання >
            [(ім'я_поля1, ім'я_поля2, ...)]
                                  <підзапит>
```

Застосування оператора INSERT з підзапитом дозволяє завантажувати відразу декілька рядків у одну таблицю, використовуючи інформацію, обрану SQL-оператором SELECT з іншої таблиці цієї ж або іншої бази даних. INSERT з підзапитом додає в таблицю стільки рядків, скільки отримує підзапит з іншої таблиці.

Виконуючи команду INSERT, необхідно мати на увазі такі особливості [19]:

таблиця може мати автоінкрементний стовпець, а унікальні значення у кожному рядку СУБД вводити автоматично. У цьому разі в списку стовпців команди INSERT не слід указувати такий стовпець;

якщо список стовпців, вказаний у команді INSERT, містіть не всі стовпці таблиці, то в них (за винятком автоінкрементного стовпця) будуть введені значення NULL або DEFAULT;

вводячи значення у стовпці нового запису, необхідно слідкувати, щоб типи значень відповідали типам стовпців таблиці. У разі необхідності можна використовувати функцію перетворення типів CAST() або їй подібну в конкретній СУБД;

таблиця може мати обмеження на значення у стовпцях. Якщо дані, що вводяться, не відповідають цим умовам, то запит на додавання даних не буде виконаний. Наприклад, якщо для якогось стовпця вказано, що він не може мати значення NULL, то спроба додати таке значення призведе до помилки;

дати зазвичай вводяться як символьні значення, розташовані в апострофах. Оскільки формати подання дати у різних країнах відрізняються, то необхідно або використовувати функції перетворення, або встановлювати для сеансу роботи режим, який би "інструктував" СУБД про формат таких значень.

Наприклад, для виконання команд у середовищі MS SQL SERVER необхідно перед командою INSERT ввести команду SET LANGUAGE RUSSIAN, яка і налаштує необхідним чином систему.

3.4.2. Модифікація даних у таблицях. Оператор UPDATE

SQL-оператор UPDATE змінює вже існуючі дані в таблиці. Синтаксис оператора модифікації UPDATE:

```
UPDATE <ім'я_таблиці>
    SET<ім'я_поля1>=<значення1 | вираз>
            [, < ім'я_поля2>=<значення2 | вираз>, ...]
                    [FROM < ім'я_таблиці >]
                            [WHERE <умова_обмеження>]
```

За ключовим словом SET слідує перелік стовпців, що підлягають оновленню (вони відокремлюються комами), а також їх нові значення. Форма запису така: ім'я_стовпця = нове_ значення.

Нове значення може бути константою, виразом (який також може посилатися на сам стовпець), DEFAULT або NULL. Наприклад, вираз Ціна = Ціна + 1 буде збільшувати значення в стовпці Ціна на 1.

Фраза WHERE необов'язкова. Якщо вона є, то повинна задавати рядки, що підлягають оновленню. Якщо фраза WHERE в операторі UPDATE відсутня, то будуть модифікуватися всі рядки в таблиці.

Особливості оператора UPDATE:

1) працює тільки з однією таблицею;

2) речення SET визначає набір полів і значень, які будуть їм присвоєні;

3) якщо умова WHERE відсутня, то зміни вносяться в усі рядки таблиці. Наприклад, у таблиці "Продажі" необхідно змінити ціну всіх товарів на 25:

```
UPDATE Товар SET Ціна = 25
```

4) речення WHERE задає умову, що визначає безліч записів, де в зазначені поля будуть поміщені відповідні значення. Наприклад, необхідно змінити в таблиці "Товар" ціну на 25 грн для товару з кодом 263:

```
UPDATE Товар SET Ціна = 25 WHERE Код_товару = 263
```

5) у реченні можна використовувати вирази, розташовуючи їх у списку для поля, яке необхідно змінити. Наприклад, у таблиці "Товар" треба зменшити ціну на 20 % для тих товарів, у яких вона перевищує 300:

```
UPDATE TOVAR SET Ціна = Ціна * 0.8 WHERE Ціна >= 300
```

6) оскільки це речення не є предикатом, то можна звичайним чином вводити NULL значення. Наприклад, у таблиці "Товар" необхідно змінити ціну на NULL щодо товару з кодом 263:

```
UPDATE Товар SET Ціна = NULL WHERE Код_товару = 263
```

7) у реченні WHERE оператора UPDATE допустимо використовувати підзапити;

8) в операторі UPDATE заборонені посилання у вкладених запитах до таблиці, яка модифікується командою UPDATE.

Розглянемо особливості використання підзапитів у операторі UPDATE.

Приклад 3.39. У таблиці "Товари" необхідно збільшити ціну на 10 % для тих товарів, за якими не менше 5 продажів:

```
UPDATE Товари
    SET Ціна = Ціна * 1.1

       WHERE 5 <=
              (SELECT COUNT (Код_товару)
                 FROM Продажі
                WHERE Товарі.Код_товару = Продажі.Код_товару)
```

У прикладі внутрішній запит підраховує кількість записів про продажі в таблиці "Продажі" для кожного товару. Якщо кількість записів не менше 5, то предикат основної функції стає справжнім, а ціна змінюється. У результаті виконання запиту отримаємо повідомлення, показане на рис. 3.47:

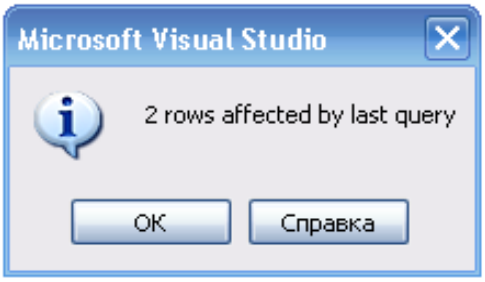


Рис. 3.47. **Результат виконання запиту**

Приклад 3.40. Необхідно зменшити ціну товарів на 10 %, у яких ціна нижча за середню. Для цього треба спочатку обчислити середню оцінку:

```
UPDATE Товари
        SET Ціна = Ціна * 0.9
                WHERE Ціна <=
              (SELECT AVG (Ціна) FROM Товари)
```

У результаті виконання запиту отримаємо повідомлення, зображене на рис. 3.45.

Іноді оператор UPDATE містить потенційну небезпеку, пов'язану з тим, що в базі може зберігатися інформація, що відрізняється незначно (наприклад, про адреси однофамільців або родичів), а користувач не має точних даних про змінювані записи.

Приклад 3.41. Потрібно змінити інформацію в полі "Адреса" таблиці "Виробники", причому користувачеві відома тільки вулиця зі старої адреси підприємства:

```
UPDATE Виробники
      SET Адреса = 'Науки, 12'
          WHERE Код_виробника =
                (SELECT Код_виробника
                        FROM Виробники
                             WHERE Адреси LIKE 'Салтівська%')
```

Виконанням запиту оператор може змінити вміст в декількох записах, що відповідають підприємствам, розташованим на вулиці "Салтівська". Саме тому більш правильна послідовність складається з декількох запитів.

Перший запит:

```
SELECT Код_виробника, Виробник
         FROM Виробники
             WHERE Адреси LIKE 'Салтівська%
```

Після виконання запиту слід провести аналіз отриманих даних і визначити проміжний ідентифікатор id = Код_виробника.

Другий запит:

```
UPDATE Виробники
       SET Адреси = 'Науки, 12'
                 WHERE Код_виробника = id
```

Як і у випадку з SQL-оператором, DELETE – послідовність рознесена в часі. Перший запит виконується для забезпечення користувача необхідною інформацією через інтерфейс, а другий – тільки тоді, коли змінює інформацію. Завдання програміста – витяг первинного ключа разом з інформацією, яка відображається користувачу на екран, і збереження його значення для подальшого використання.

3.4.3. Видалення рядків таблиці. Оператор DELETE

Кожна СУБД надає користувачам свої доповнення до стандарту SQL. Наприклад, SQL Server надає два оператора для видалення рядків з таблиці або подання – DELETE і TRUNCATE TABLE.

Оператор TRUNCATE TABLE без всяких умов видаляє всі рядки в таблиці. Оператор DELETE забезпечує більшу гнучкість і дає можливість видаляти тільки обрані рядки за допомогою фрази WHERE, яка може містити додаткові таблиці та подання.

Оператор DELETE повинен виглядати так:

```
DELETE [FROM] таблиця_або_подання
              [FROM таблиці]
                       [WHERE умова_ відбору]
```

Особливості оператора DELETE:

перша фраза FROM – необов'язкова; використовується для зручності читання;

список стовпців у операторі DELETE не вказується, оскільки з видаленням рядка видаляються всі стовпці;

необов'язкова фраза WHERE дає можливість указувати, які рядки слід видалити;

якщо фраза WHERE відсутня, видаляються всі рядки в зазначеній таблиці або поданні;

необов'язкова друга фраза FROM дозволяє задавати додаткові джерела (таблиці або подання), які будуть використовуватися в умові відбору в реченні WHERE;

подібний синтаксис оператора може ввести в оману, оскільки рядки не будуть вилучатися з таблиць і уявлень, зазначених у другій фразі FROM;

якщо в другій фразі FROM указується більше однієї таблиці або подання, то їх імена слід відокремлювати комами.

Приклад 3.42. Видалення даних з таблиці "Товар":

```
DELETE FROM Товар -- видалення всього вмісту таблиці "Товар".
DELETE FROM Товар -- видалення записів з таблиці "Товар",
WHERE Ціна <25         -- що відповідають умові Ціна <25.
DELETE FROM Продажі --видалення записів з таблиці "Товар",
WHERE Дата<= 18/09/2017 -- в яких Дата<= 18/09/2017.
```

Результат виконання оператора TRUNCATE TABLE ідентичний результату виконання оператора DELETE, для якого не зазначена умова WHERE, тобто з таблиці видаляються всі рядки.

Оператор TRUNCATE TABLE працює більш ефективно, ніж аналогічний оператор DELETE, оскільки останній видаляє по одному рядку за один прохід і вносить у журнал транзакцій окремі записи для кожної з них. Оператор TRUNCATE TABLE видаляє всі рядки шляхом очищення сторінок, призначених для таблиці, і в журнал транзакцій записуються тільки ці звільнення.

Особливості оператора TRUNCATE TABLE:

не можна застосовувати для таблиці, на яку є посилання зовнішнього ключа з інших таблиць. У цьому випадку треба використовувати DELETE;

видалення, які виконані за допомогою TRUNCATE TABLE, не активізують тригери видалення;

якщо таблиця є частиною індексованого подання, то в ній не вдасться використовувати інструкцію TRUNCATE TABLE.

5.5. Подання даних

3.5.1. Поняття подання (VIEW) і його призначення

Подання (VIEW) – це об'єкт СУБД, що є іменованим запитом на вибірку, тобто SQL-оператором SELECT з ім'ям. Подання не зберігає дані, а тільки подає їх. Користувач сприймає подання як частину віртуального відношення (віртуальну таблицю). Подання – це табличний підзапит; міститься в таблицях схеми та викликається щоразу під час його застосування.

Існує **два типи подання**:

модифіковані (оновлювані) – команди модифікації можуть виконуватися в поданні;

призначені тільки для читання у запиті.

Синтаксис подання має такий вигляд:

```
<визначення_подання> :: = {CREATE | ALTER} VIEW
     ім'я_подання [(ім'я_стовпця [, ... n])]
               [WITH ENCRYPTION] AS SELECT_запит [WITH CHECK OPTION]
```

Параметр С ENCRYPTION пропонує серверу шифрувати SQL-код запиту. Це унеможливлює його несанкціонований перегляд і використання. Якщо під час визначення подання необхідно приховати імена початкових таблиць і стовпців, а також алгоритм з'єднання даних, необхідно застосувати цей аргумент.

Параметр WITH CHECK OPTION пропонує серверу проводити перевірку змін, що виконуються через подання, на відповідність критеріям в операторі SELECT цього ж подання.

Для видалення подання слід вживати команду:

```
DROP <ім'я_подання>
```

Створення подання означає його визначення у термінах базових таблиць або вже існуючих подань. Це визначення зберігається у словнику бази даних.

Запит до подання на вибірку повністю аналогічний запиту SELECT на вибірку зі звичайної таблиці. Проте на модифікацію вмісту бази даних через подання накладається декілька умов.

Приклад 3.43. Необхідно створити подання, яке формує дані про товари та приховує інформацію про ціну його закупівлі, а також змінює імена деяких атрибутів:

```
  CREATE VIEW Товар_ціна(Код_товару, Товар, Ціна1)
AS SELECT Код_товару, Товар, Ціна
                FROM Товари
```

Повідомлення про результат запиту зображене на рис. 3.48.

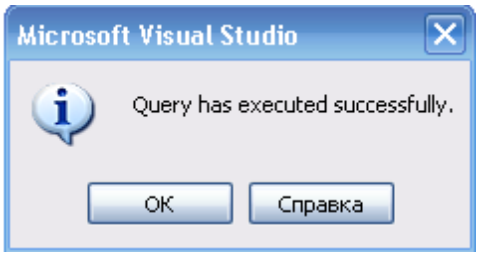


Рис. 3.48. **Повідомлення про виконання запиту**

Тепер створене подання з ім'ям "Товар_ціна", яке можна використовувати в командах, як і будь-яку іншу таблицю бази даних (рис. 3.49).

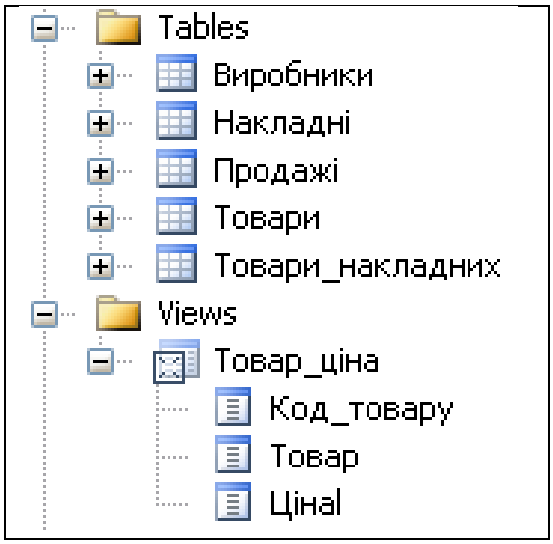


Рис. 3.49. **Подання, створене шляхом виконання запиту**

Ця таблиця може бути запрошена, модифікована, з'єднана з іншими таблицями та поданнями. Запит такого подання має вигляд:

```
SELECT * FROM Товар_ціна
```

Результат виконання подання зображений на рис. 3.50.

	Код_товару	Товар	Ціна!
▶	1	Хліб "Українськ...	3,00
	2	Батон "Молочний"	2,80
	3	Булка з маком	2,00

Рис. 3.50. **Результат виконання запиту**

Механізм подання дозволяє:

приховати несуттєві або небажані деталі бази даних для різних користувачів;

забезпечити підвищену продуктивність БД у разі частого використання одного і того ж оператора SELECT;

модифікувати реальні структури даних у зручному для додатка вигляді, наприклад, зі зростанням або реструктуризацією бази даних (фактично команди модифікації перенаправляються до базової таблиці);

підтримувати стару структуру бази даних (структура бази даних змінена, а старе програмне забезпечення не змінено – функціональність старої залишається, а структура бази даних розширюється новими модулями);

розмежовувати права доступу до даних;

переглядати користувачам ті самі дані в різних варіантах в той самий час;

обмежувати обсяги даних для зручності роботи з БД;

після створення подання використовувати його в запитах нарівні з таблицями;

на основі одного подання створювати нові подання;

якщо список стовпчиків у описі подання не вказаний, то в подання увійдуть усі стовпці з запиту, на основі яких вона створена, з відповідними іменами.

За допомогою опції ENCRYPTHION виконується шифрування подання, стають недоступні можливості перегляду та редагування подання, але його можна виконувати та видаляти. За допомогою опції SCHEMABINDING виконується зв'язування подання зі схемою БД. Цей зв'язок визначає, від яких об'єктів (таблиць або іншого подання) залежить розглядуване подання.

Наявність опції SCHEMABINDING дозволяє перешкодити внесенню змін (за допомогою операторів CREATE, ALTER або DROP) в об'єкти, від яких залежить подання. Створення індексованого подання можливе тільки за наявності опції SCHEMABINDING.

Конструкція WITH CHECK OPTION визначає, як здійснюється модифікація або вставка даних за допомогою такого подання, якщо вставлений рядок відповідає критеріям конструкції WHERE оператора SELECT.

Оператор AS у стандарті SQL-92 дозволяє задавати імена результатам виконання виразів підзапитів і використовувати їх. Таким чином, подання можна вважати постійним табличним підзапитом, що зберігається для схеми БД і викликається за ім'ям.

Особливості роботи з поданнями:

запит, на якому засновано подання, виконується щоразу, коли подання бере участь в якій-небудь команді. Подання щоразу буде модифіковане автоматично, якщо таблиця, закладена в його основу, змінюється;

у поданні можна використовувати обчислювані поля, угруповання, підзапити. Водночас слід враховувати обмеження, що відображають природу подання;

за допомогою подання ніколи неможлива така ж швидкодія, як у безпосередньому виклику оператора SELECT, закладеного в основу цього подання. Дані в поданні впорядковуються за допомогою речення ORDER BY запиту, на якому засноване подання.

подання може бути засноване на декількох базових таблицях;

обчислення в поданні виконується щоразу під час його використання. Будь-які зміни в даних таблиць БД адекватно відобразяться в поданні – в цьому його відмінність від запиту до БД;

матеріалізація подання здійснюється у ході реалізації відповідних дій користувача. До цього моменту матеріалізовані дані подання в БД не зберігаються. Матеріалізація означає, що через посилання на ім'я подання СУБД знаходить його визначення в таблицях схеми та створює

робочу таблицю. Потім нова таблиця заповнюється результатами роботи оператора SELECT, що знаходиться в тілі визначення подання;

для отримання інформації про різні об'єкти БД використовують подання інформаційної схеми та подання каталогів.

Приклад 3.44. Необхідно створити подання для визначення дат і кількості продажів товару "Булка з маком", якщо відома тільки назва товару, але невідомо значення поля "Код_товару" для нього:

```
CREATE VIEW Дата_продажі_товару(Дата, Кількість)
AS
SELECT Дата, Кількість FROM Продажі
    WHERE Код_товару =
                (SELECT Код_товару FROM Товари
                                WHERE Товар = 'Булка з маком')
```

У результаті виконання запиту (який містить підзапит) буде створено подання з ім'ям Дата_продажі_товару (рис. 3.51).

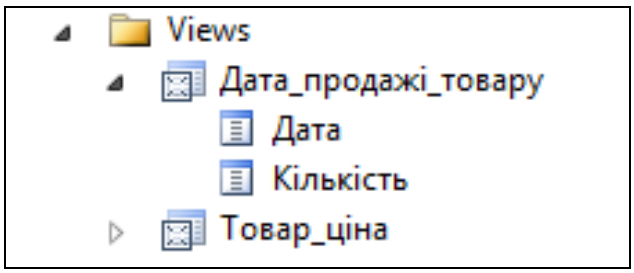


Рис. 3.51. **Результат виконання запиту**

Бажано, щоб поля, які беруть участь у поданні, були включені до складу виведених полів, навіть якщо вони містять однакові значення. З виконанням цього подання отримаємо результат, відображений на рис. 3.39.

Способи внесення змін до визначення подання:

за допомогою оператора ALTER VIEW. Передбачається, що розглядається подання яке вже існує (виконується повна заміна подання, зберігаються всі права на його використання);

знищення подання (DROP VIEW <ім'я_подання>) і його повторне створення за допомогою оператора CREATE VIEW.

У видаленні подання немає необхідності видалення всіх даних з подання, тому що реальні дані в ньому не містяться. Для видалення подання користувач повинен бути його власником.

3.5.2. Модифіковані та немодифіковані подання

Подання (за допомогою його імені) можна використовувати в командах: запитувати (SELECT); змінювати (UPDATE); вставляти в нього записи (INSERT); видаляти з БД (DELETE); з'єднувати з іншими таблицями та поданнями. Існують обмеження на операції модифікації (UPDATE, INSERT, DELETE), які залежать від виду подання. Фактично команди модифікації подання перенаправляються до базової таблиці. Внесена зміна в модифіковане подання обов'язково вноситися і в базову таблицю.

Критерії, за якими визначають модифіковане подання:

необхідно, щоб подання ґрунтувалося тільки на одній базовій таблиці;

подання має містити первинний ключ базової таблиці;

подання не повинно мати жодних полів, які б були агрегатними функціями;

подання не повинно містити запитів з DISTINCT у своєму визначенні;

подання не повинно використовувати GROUP BY або HAVING у своєму визначенні;

бажано, щоб подання не використовувало в своєму рішенні підзапити;

подання може застосовуватися в іншому поданні, але останнє повинно також модифікуватись;

подання не повинно використовувати константи, рядки або вирази значень серед обраних полів виведення;

для команди INSERT подання мають містити будь-які поля основної таблиці з обмеженнями NOT NULL, якщо інше обмеження за замовчуванням не визначено.

У базі даних створимо подання, що дозволяє модифікувати базову таблицю "Товари". Для цього необхідно виконати всі розглянуті умови.

Приклад 3.45. З допомогою подання необхідно визначити інформацію про товари, ціна яких перевищує 2 грн.

```
CREATE VIEW Товар_2(Код_товару, Товар, Ціна, Ціна_закупівлі)
        AS SELECT Код_товару, Товар, Ціна, Ціна_закупівлі
                  FROM Товари
                  WHERE Ціна > 2
```

Результат такого запиту зображений на рис. 3.52.

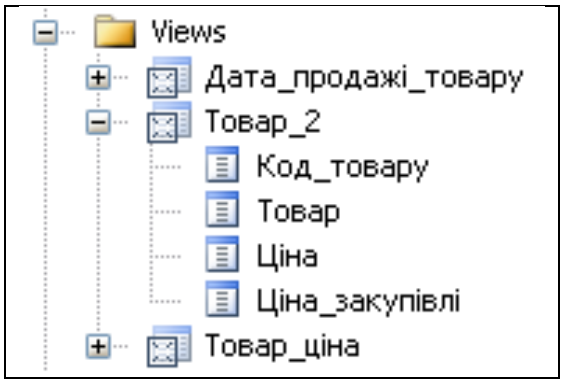


Рис. 3.52. **Результат виконання запиту**

Приклад 3.46. Необхідно додати у базову таблицю "Товари" відомості про нову продукцію "Хліб "Бородинський".

```
INSERT INTO Товар_2
          (Товар, Ціна, Ціна_закупівлі)
               VALUES     ('Хліб "Бородинський"', 6, 5)
```

Щоб пересвідчитись у результаті, отримаємо з таблиці "Товари" відомості про товари:

```
SELECT * FROM Товари
```

Отримаємо результівну таблицю (рис. 3.53):

	Код_товару	Товар	Ціна	Ціна_закупівлі
▶	1	Хліб "Український"	3,00	2,50
	2	Батон "Молочний"	2,80	2,50
	3	Булка з маком	2,00	1,80
	4	Хліб "Бородинський"	6,00	5,00
*	NULL	NULL	NULL	NULL

Рис. 3.53. **Результат виконання запиту**

Щоб забезпечити додавання даних за допомогою подання, потрібно, щоб цим поданням були охоплені всі стовпці без обмежень.

Таким чином, модифіковані подання фактично подібні фрагментам базових таблиць і відображають певну частину їх вмісту.

Подання "тільки для читання" дозволяють отримувати та форматувати дані більш раціонально – уникати складних предикатів і знижувати ймовірність помилок.

Запитання для самодіагностики

1. Які компоненти мови SQL?
2. Які основні речення компонент мови SQL?
3. Який синтаксис оператора SELECT?
4. Які основні особливості оператора SELECT?
5. Які основні особливості речення FROM?
6. Які основні особливості речення WHERE?
7. Які особливості оператора BETWEEN?
8. Які особливості оператора IN?
9. Які особливості оператора IS NULL?
10. Які особливості оператора LIKE?
11. Які особливості простого виразу CASE?
12. Які особливості пошукового виразу CASE?
13. Які особливості речення GROUP BY?
14. Які особливості речення HAVING?
15. Які особливості речення ORDER BY?
16. Який порядок виконання речень SQL-оператора SELECT?
17. Дайте визначення підзапиту.
18. Охарактеризуйте основні правила складання підзапитів.
19. Які особливості роботи з кореляційним підзапитом?
20. Які засоби формування багатотабличних запитів?
21. Які особливості внутрішнього об'єднання таблиць?
22. Які засоби використовують для додавання даних у таблицю?
23. Які засоби використовують для видалення даних з таблиць?
24. Які засоби використовують для модифікації даних UPDATE в таблиці?

Лабораторна робота 2
Побудова DML-запитів

Цілі роботи:

1. Набуття практичних навичок побудови запитів на отримання даних таких типів: детальні, групування, підзапити, модифікування даних.
2. Створення і використання подань.

Перед виконанням роботи студент повинен знати:

1. Основи використання Visual Studio.
2. Основні об'єкти бази даних MS SQL Server.
3. Основні команди мови SQL.

Після виконання лабораторної роботи студент повинен уміти:

1. Самостійно будувати: детальні запити; запити, що містять групування даних за співпаданням значень указаних полів; запити, до складу яких входять підзапити; запити на модифікування даних.
2. Самостійно створювати подання як об'єкта бази даних.

Хід роботи

1. Побудова базових запитів.
2. Створення запитів з групуванням даних.
3. Побудова запитів з підзапитами та запитів на зміну даних.
4. Створення і використання подань.
5. Створення запитів до індивідуальної бази даних.

Форма звітності

За результатами виконання роботи оформити електронний звіт засобами Word.

В електронному звіті за кожним завданням подати текст завдання, запит на його виконання та скриншот результату.

Усі скриншоти з виконаними запитами для SQL Server повинні мати коментар в першому рядку з прізвищем студента.

Коментар задається рядком з двома початковими мінусами.

Приклад.

```
-- Запит 2.1 Іванов В.І. 3 курс 1 група
SELECT * FROM Замовлення.
```

Критерії оцінювання

У 12-бальній системі оцінка результату захисту лабораторної роботи формується за такими правилами:

- за кожне завдання з діапазону 1 – 4 може бути виставлено від 0 до 1,5 балів;
- за кожне завдання з діапазону 1.1 – 1.5 із п. "завдання для самостійного виконання" може бути виставлено від 0 до 1 балу, а за аналогічні завдання з індивідуальною базою даних від 0 до 1,5 балів;
- за кілька варіантів вирішення одного із завдань додається 1 бал;
- за вибір варіанту (який з кількох варіантів вирішення є оптимальним) та обґрунтування вибору додається 1 бал;
- за побудову запитів в іншій СУБД (oracle, db2 тощо) додається 1 бал за кожний запит;
- за запізнення із захистом лабораторної роботи знімається 2 бали за кожний тиждень.

Отримана кількість балів з відповідей на кожне завдання лабораторної роботи підсумовується. У результаті такого підрахунку студентом може бути отримано від 0 до 12 балів.

Рекомендована література

1. Андон Ф. Язык запросов SQL: учебный курс / Ф. Андон, В. Резниченко. – СПб. : Питер ; К. : BHV, 2006. – 416 с.
2. Астахова И. Ф. SQL в примерах и задачах / И. Ф. Астахова, А. П. Толстобров, И. М. Мельников. – Минск : Новое знание, 2002. – 176 с.
3. Боуман Д. Практическое руководство по SQL / Д. Боуман, С. Эмерсон, М. Дарновски. – Москва : Вильямс, 2002. – 352 с.
4. Дейт Дж. Введение в системы баз данных / Дж. Дейт.– 8-е изд. – Москва : Вильямс, 2005. – 1328 с.
5. Тарасов О. В., Лосєв М. Ю., Федько В. В. Використання мови SQL для роботи з сучасними системами керування базами даних. Харків : Вид. ХНЕУ, 2013. – 348 с.
6. Тарасов О. В., Федько В. В. Клієнт-серверні технології СУБД Oracle. Мова SQL Oracle. Навчальний посібник для самостійної підготовки студентів з навчальної дисципліни "Організація баз даних та знань". Харків : Вид. ХНЕУ ім. С. Кузнеця, 2015. – 384 с.
7. Форта Б. Освой самостоятельно SQL / Б. Форта. – Москва : Вильямс, 2005. – 288 с.

Інструкції до виконання

Попередні зауваження

Завдання лабораторної роботи виконуються з базою даних ***Торгівля***, яка містить таблиці ***Замовники***, ***Продавці*** та ***Замовлення*** в середовищі Visual Studio. Для цього слід скопіювати папку з проектом, що створений у попередній лабораторній роботі.

Запити виконують у вікні редактора T-SQL. Подання додають до папки **Views** відповідної бази даних контекстною командою **Add New View**.

Якщо для виконання запитів будуть потрібні додаткові таблиці або нові поля в існуючих таблицях, їх слід створити. Дані підібрати так, щоб у результаті виконання запиту, який може містити кілька рядків, їх було не менше двох.

Завдання виконуються за варіантами відповідно з табл. 1.

Таблиця 1

Варіанти завдань

Варіант	Завдання		
	1	2	3
1	2	3	4
1	1;4;7;10;13	1;3;6;10;12;15	1;3;7;10;14
2	2;5;8;10;14	2;4;7;11;13;16	2;4;8;11;15
3	3;6;9;11;13	1;5;8;10;14;17	1;5;9;12;14
4	1;4;8;11;14	2;3;9;11;12;18	2;4;6;13;15
5	2;5;7;12;13	1;4;6;10;13;17	1;3;7;12;15
6	3;6;8;12;14	2;5;7;11;14;16	2;4;8;11;14
7	1;5;9;10;13	1;4;8;10;12;15	1;5;9;10;15
8	2;6;9;11;14	2;5;9;11;13;18	2;5;6;10;14
9	3;4;8;12;13	1;3;6;10;14;17	1;4;7;11;14
10	1;5;7;12;14	2;4;7;11;13;16	2;3;8;11;15
11	2;6;7;11;13	1;5;8;10;12;15	1;3;9;12;15
12	3;4;9;10;14	2;3;9;11;14;16	2;4;8;12;14
13	1;6;8;11;13	1;5;8;10;13;17	2;5;7;13;14
14	2;4;7;12;14	2;3;7;11;12;18	1;4;6;13;15
15	3;5;8;10;13	1;4;6;10;13;15	2;3;7;10;15
16	1;6;9;11;14	2;5;9;11;14;16	1;3;9;11;14
17	2;4;7;12;13	1;3;8;10;12;17	2;4;8;12;14

Закінчення табл. 1

1	2	3	4
18	3;5;7;11;14	2;4;7;11;13;18	1;4;6;13;15
19	1;4;8;10;13	1;4;6;10;13;17	2;5;9;11;15
20	2;5;8;10;14	2;5;9;11;12;16	1;5;8;12;15
21	3;6;9;11;13	1;3;8;10;14;15	2;4;7;11;14
22	1;5;9;12;14	2;4;7;11;13;16	1;3;6;10;14
23	2;6;7;12;13	1;5;6;10;12;17	2;5;8;10;15
24	3;4;8;11;14	2;3;7;11;14;18	1;4;7;11;14
25	1;6;9;10;13	1;4;8;10;13;15	2;5;9;11;15
26	2;4;8;11;14	2;5;9;11;12;161	1;3;6;12;14
27	3;5;7;12;13	1;4;8;10;14;17	2;4;8;12;14
28	1;5;9;11;14	2;3;7;11;12;18	1;3;9;13;15
29	2;4;8;10;13	1;4;6;10;13;17	2;5;7;13;15
30	3;6;7;12;14	2;5;9;11;14;16	1;4;6;10;14

Завдання 1. Побудова базових запитів

1. Вивести номери замовлень, суму та дату для всіх рядків з таблиці ***Замовлення***.

2. Вивести всі рядки з таблиці ***Замовлення***, у яких код продавця дорівнює 4.

3. Вивести таблицю зі стовпцями ***Місто***, ***Прізвище_продавця***, ***Код_продавця*** і ***Комісійні***.

4. Вивести рейтинг і прізвище кожного замовника з міста ***Київ***.

5. Вивести всі коди продавців з таблиці ***Замовлення*** без повторень.

6. Вивести прізвища продавців з міста ***Харків***, у яких комісійні перевищують 0.1.

7. Вивести список усіх замовників з рейтингом не вище 80, крім тих, які мешкають у Києві.

8. Вивести всі пари продавців, які живуть в одному місті. Виключити комбінації продавців між собою, а також дублікати рядків, що виводяться в зворотному порядку.

9. Вивести прізвища та міста всіх замовників з таким самим рейтингом, як у Іваненка. Використовувати поле Код_замовника для Іваненка, а не його конкретний рейтинг.

10. Визначити загальну суму замовлень на 3 вересня поточного року.

11. Підрахувати кількість різноманітних непорожніх значень поля ***Місто*** в таблиці ***Замовники***.

12. Вибрати прізвище замовника з найвищим рейтингом з тих замовників, у яких прізвище починається на букву 'П'.

13. Обчислити загальну суму комісійних, отриманих продавцями на замовлення, що зроблені замовниками з рейтингом понад 90.

Завдання 2. Створення запитів з групуванням даних

1. Нехай кожен продавець має комісійні в розмірі 8 %. Вивести код замовлення, код продавця і суму комісійних для кожного замовлення.

2. Визначити максимальний рейтинг для кожного міста. Результат повинен бути здійснений у вигляді:

Для поля ***Місто*** максимальний рейтинг становить: ***Рейтинг***.

3. Вивести список замовників у порядку убування рейтингу. Прізвища замовників супроводжувати їхнім кодом і рейтингом.

4. Вивести загальні суми замовлень за кожен день у порядку спадання витрат на пальне.

5. Вивести список замовників, чиї прізвища починаються на букву "В", в алфавітному порядку.

6. Визначити мінімальну суму замовлення для кожного замовника.

7. Написати запит для виведення всіх замовлень, що зроблені 1 або 3 вересня поточного року.

8. Вибрати всіх замовників, що обслуговуються продавцями Іваненком і Петренком.

9. Вивести всіх замовників, чиї прізвища починаються з літер, які потрапляють в діапазон від А до К.

10. Вибрати всі замовлення, що мають нульові або невизначені значення у полі ***Сума***.

11. Вибрати всіх продавців, які обслуговують замовників з рейтингом 30:

а) за допомогою предиката EXISTS;

б) використовуючи операцію з'єднання.

12. Вибрати з таблиці ***Замовники*** тих замовників, яких обслуговує продавець, який вже має хоча б одного замовника з оформленими в таблиці ***Замовлення*** замовленнями (замовленням).

13. Вибрати всіх замовників, рейтинг яких вищий за рейтинг будь-якого замовника з міста Київ.

14. Знайти всіх продавців, які в своєму місті не мають ніякого замовника (використовувати предикат **ANY** або **ALL**).

15. Вибрати всі замовлення з сумою, що перевищує будь-яку для замовників міста ***Одеса***:

а) використовуючи предикат;

б) використовуючи агрегатну функцію **MAX**.

16. Створити об'єднання з двох запитів, що відображають прізвища, міста та рейтинги всіх замовників. Замовники, які мають рейтинг 50 і більше повинні супроводжуватися словами "Високий рейтинг", а інші словами "Низький рейтинг".

17. Вивести прізвища та номери кожного продавця і замовника, який має більше одного поточного замовлення. Результат подати в алфавітному порядку.

18. Сформувати об'єднання з двох запитів. Перший запит вибирає номери всіх продавців міста Київ, а другий – номери всіх замовників з міста Київ.

Завдання 3. Побудова запитів з підзапитами та запитів на зміну даних

1. Використовуючи підзапит, вибрати всі замовлення замовника Іваненка.

2. Вивести прізвища та рейтинги всіх замовників, що мають середнє значення рейтингу.

3. Знайти загальну суму всіх замовлень для кожного продавця, якщо ця загальна сума перевищує суму замовлення, що має максимальну суму замовлення.

4. Використовуючи підзапит, вибрати прізвища та номери всіх замовників з максимальним для їхнього міста рейтингом.

5. Вибрати всіх продавців (із зазначенням прізвища та коду), що мають у своїх містах замовників, які не обслуговуються ними:

а) з використанням з'єднання;

б) з використанням підзапиту.

6. Помістити такі значення в таблицю ***Продавці*** в зазначеному порядку: у полі **Місто** – Київ, у полі **Прізвище** – Петренко, у полі **Комісійні** – NULL, у полі Код_продавця – 11.

7. Видалити всі замовлення замовника Сидоренка з таблиці ***Замовлення***.

8. Збільшити рейтинг усіх замовників міста Одеса на 10.

9. Продавець Іваненко звільнився з компанії. Передати його замовників продавцеві Петренку.

10. Нехай є таблиця **Багато_Замовників** з такою ж структурою, як таблиця **Продавці**, і з такими ж іменами стовпців. Вставити в таблицю **Багато_Замовників** інформацію про тих продавців, які мають більше п'яти замовників.

11. Видалити всіх замовників, які не мають поточних замовлень.

12. Збільшити на 20 % комісійні тих продавців, які мають поточні замовлення на загальну суму понад 3 000 гривень.

13. Визначити дати, коли загальна сума замовлень була мінімальною.

14. Визначити які замовники не оформляли замовлення в своєму місті.

15. Визначити суму всіх замовлень за кожним містом.

Завдання 4. Створення і використання подань

1. Створити подання для виведення всіх замовників з найбільшим рейтингом.

2. Створити подання для виведення кодів продавців у кожному місті.

3. Створити подання, яке показує середню і загальну суму замовлень для кожного продавця після його прізвища (нехай усі прізвища унікальні).

4. Створити подання для показу продавців, які обслуговують більше одного замовника.

5. Визначити які з наведених подань є модифікованими:

5.1)
```
CREATE VIEW Просто_замовлення
    AS SELECT DISTINCT Код_замовника, Код_продавця,
    Код_замовлення, Дата_замовлення
    FROM Замовлення;
```

5.2) CREATE VIEW Загальна_сума_замовлення
AS SELECT Прізвище_замовника, SUM (Сума)
FROM Замовлення, Замовники
WHERE Замовлення. Код_замовника =Замовники. Код_замовника
GROUP BY Прізвище_замовника;

5.3) CREATE VIEW Замовлення_на_третє
AS SELECT * FROM Просто_замовлення
WHERE Дата_замовлення = '03/09/2018;

5.4) CREATE VIEW Невизначені_міста
AS SELECT Код_продавця, Прізвище_продавця, Місто
FROM Продавці
WHERE Місто IS NULL OR
Прізвище_продавця BETWEEN 'A' AND 'Z';

6. Створити подання для таблиці **Продавці** з ім'ям **Комісійні**. Подання має включати тільки поля **Комісійні** та **Код_продавця**. За допомогою цього подання можна буде вводити або змінювати комісійні, але тільки ті значення, які перебувають в діапазоні від 0.10 до 0.20.

Завдання 5. Створення запитів до індивідуальної бази даних

Для індивідуальної бази даних самостійно сформулювати такі запити та реалізувати їх мовою SQL:

1. Створити запит зі з'єднанням кількох таблиць, з умовою фільтрування та впорядкуванням.

2. Створити запит з групуванням та використанням агрегатних функцій.

3. Створити запит з підзапитом.

4. Створити запит на зміну даних (додавання, видалення, модифікування даних).

5. Створити подання для вводу та зміни даних.

Примітка. У звіті за кожним завданням подайте текст завдання, запит на його виконання та скриншот результату.

Висновки

Мова маніпулювання даними SQL – Data Manipulation Language (DML) дозволяє керувати змістом таблиць бази даних. З допомогою функціональних можливостей DML SQL, які підтримує ця СУБД, стає доступною інформація з бази даних для кінцевих користувачів і розробників додатків. DML SQL дозволяє виконувати такі дії:

додавання даних у таблицю;

модифікація даних в таблицях;

видалення даних з таблиць;

отримування інформації засобами багатотабличних запитів з об'єднанням даних таблиць;

отримування інформації з використанням різноманітних функцій.

Оператор SELECT фактично є серцем мови DML SQL. Він дозволяє виконувати пошук і отримувати данні дуже широким набором засобів. З його допомогою можливо відповісти на запитання про тип, кількість, розташування даних тощо.

Розуміння порядку обробки даних у SQL-операторі SELECT значно спрощує його використання. Речення завжди виконуються в тому порядку, в якому вони були задані при описі оператора SELECT. Саме тому в користувача є можливість пропустити деякі речення, якщо вони не потрібні; але він не має права змінювати порядок їх визначення.

Першим виконується речення SELECT, що вибирає з таблиці значення всіх полів таблиць і формує з них проміжну таблицю в 1НФ. Існування DISTINCT впливає на вміст цієї таблиці, видаляючи повторювані записи. Після цього проміжна таблиця надходить до речення WHERE, де відбираються її записи. Далі проміжна таблиця потрапляє до речення GROUP BY, де знаходяться групи та виконуються задані над ними дії. Після цього результат надходить до речення HAVING, яке може своєю умовою відсікти ще якісь дані. Після цього результат сортується згідно з визначеннями речення ORDER BY і надходить до користувача.

Відсутність одного з речень WHERE, HAVING або ORDER BY призводить до пропуску даного етапу, що прискорює обробку запиту користувача. У SQL-операторі SELECT завжди виконується угруповування даних. Якщо пропозиція GROUP BY не задана, то угруповування автоматично проводиться на всіх полях.

Розділ 4. Особливості реалізації SQL у СУБД *MS SQL SERVER*

5.6. Мова Transact-SQL

Transact-SQL (T-SQL) – процедурне розширення мови SQL компанією *Microsoft* (для Microsoft SQL Server). SQL був розширений такими додатковими можливостями, як:

керівні оператори;

локальні та глобальні змінні;

різні додаткові функції для обробки рядків, дат, математики і т. п.;

підтримка аутентифікації *Microsoft Windows*.

Мова Transact-SQL є ключем до використання *MS SQL Server*. Усі додатки, які взаємодіють з екземпляром *MS SQL Server*, незалежно від їх реалізації і призначеного для користувача інтерфейсу відправляють серверу інструкції Transact-SQL.

Аналогічно стандартному SQL, у T-SQL використовують ідентифікатори, змінні, операції і оператори.

Ідентифікатори – це спеціальні символи, які використовуються зі змінними для визначення їх типу або для угруповання слів у змінну.

Типи ідентифікаторів:

@ – ідентифікатор локальної змінної (користувальницької);

@@ – ідентифікатор глобальної змінної (вбудованої);

– ідентифікатор локальної таблиці або процедури;

– ідентифікатор глобальної таблиці або процедури;

[] – ідентифікатор угруповання слів у змінну.

Змінні використовують у сценаріях і для зберігання тимчасових даних. Щоб працювати зі змінною, її потрібно оголосити, притому оголошення повинно бути здійснено в тій транзакції, в якій виконується команда, що використовує цю змінну. Інакше кажучи, після завершення транзакції, тобто після команди GO, змінна знищується.

SQL Server підтримує два види змінних: локальні та глобальні. *Локальні змінні* існують тільки в межах сеансу, під час якого вони були створені. *Глобальні* слугують для отримання інформації про сервер або бази даних в цілому; належить до конкретного сеансу. Оголошення локальної змінної виконується командою DECLARE. Задання значення змінної здійснюється командою SET або SELECT:

```
-- Об'ява змінних
DECLARE @EmpID int, @EmpName varchar(40)
-- Завдання значення змінної @EmpID
SET @EmpID = 1
-- Завдання значення змінної @EmpName
SELECT @EmpName = UserName FROM Users WHERE UserID = @EmpID
SELECT @EmpName AS [Employee Name]
```

Transact-SQL дозволяє використовувати тимчасові таблиці, тобто таблиці, які створюються в пам'яті сервера на час роботи користувача з базою даних. Тимчасові таблиці можуть мати будь-яке ім'я, але починатися обов'язково з символу #.

Наприклад, треба створити тимчасову таблицю #TempTable, в яку скопіювати вміст колонки UserName таблиці Users:

```
SELECT UserName INTO #TempTable
FROM Users
```

Вибрати всі записи тимчасової таблиці #TempTable:

```
SELECT * FROM #TempTable
```

Оператори – це спеціальні команди, призначені для виконання простих операцій над змінними:

арифметичні оператори: "*" – помножити, "/" – ділити, "%" – залишок від ділення, "+" – скласти, "–" – відняти, "()" – дужки;

оператори порівняння: "=" – одно, ">" – більше, "<" – менше, "> =" – більше або дорівнює, "<=" – менше або дорівнює, "<>" ("! =") – не одно;

оператори з'єднання: "+" – з'єднання (конкатенація) рядків;

логічні оператори: "AND" – і, "OR" – або, "NOT" – ні;

оператори з множинами: "IN".

В T-SQL визначені такі **типи даних**:

числа – для зберігання числових змінних (bit, int, tinyint, smallint, bigint, numeric, decimal, money, smallmoney, float, real);

дати – для зберігання дати та часу (date, datetime, smalldatetime);

символи – для зберігання символьних даних (char, nchar, varchar, nvarchar);

двійкові дані – для зберігання бінарних даних (binary, varbinary);

великого обсягу – типи даних для зберігання великих бінарних даних (text, ntext, image);

спеціальні – покажчики (cursor), 16-байтове шістнадцяткове число, яке використовується для GUID (uniqueidentifier), штамп зміни рядка (timestamp), версія рядка (rowversion), таблиці (table);

типи даних користувача – визначаються на розсуд користувача.

Для використання символів кирилиці (НЕ ASCII кодування) застосовують типи даних з префіксом "n" (nchar, nvarchar, ntext), які кодують символи двома байтами. Інакше кажучи, для роботи з Unicode використовують типи даних з "n" (від слова national).

Для даних змінної довжини використовують типи даних з префіксом "var". Типи даних без префікса "var" мають фіксовану довжину області пам'яті, невикористана частина якої заповнюється пробілами або нулями. Існують особливості запису констант у T-SQL. Наприклад, записати константу типу "Дата" можна в один із таких способів:

```
VALUES(… , '2017/08/21') -- (формат 'YYYY/MM/DD')
або так:
VALUES(… , '2017-08-21') -- (формат 'YYYY-MM-DD').
VALUES(… , '2017.08.21) -- (формат 'YYYY.MM.DD').
VALUES(… , '08.21.'2017) -- (формат 'MM.DD.YYYY').
VALUES(… , '08.21.17') -- (формат 'MM.DD.YY').
```

Типи даних дата/час наведені в табл. 4.1.

Таблиця 4.1

Типи даних дата/час в SQL Server

Типи даних	Розмір в байтах	Опис
Date Time	8	Дані про дату або час, що належать до періоду з 1.01.1753 р. до 31.12.9999 р., які визначаються з точністю до 0,03 секунди
Small DateTime	4	Дані про дату або час, що належать до періоду з 1.01.1900 р. до 6.06.2079 р., які визначаються з точністю до хвилини

В табл. 4.2 показані основні функції дати та часу *MS SQL Server*.

Таблиця 4.2

Функції дати і часу MS SQL Server

Функції	Призначення
DATEADD	Додає до дати деяку величину
DATEIFF	Виводить кількість одиниць часу, що задані в аргументі datepart, між двома датами
DATENAME	Повертає текстові імена, що відповідають заданій даті
DATEPART	Витягує певний фрагмент з заданої дати
DAY	Витягує день з дати
GETDATE	Повертає поточний час і дату
GETUTCDATE	Повертає поточну дату та час за Гринвічем, перетворене в формат універсального синхронізованого часу
MONTH	Витягує місяць з дати
YEAR	Витягує рік з дати

Деякі з функцій дати та часу використовують аргумент datepart, що визначає фрагмент, до якого застосовується операція.

Приклад 4.1. Функція DATEADD приймає в якості аргументів маркер datepart, величину приросту та вихідну дату. Вона повертає результат додавання зазначеної величини в одиницях, зазначених в аргументі datepart, до поточної дати:

```
SELECT GETDATE () AS 'Дата', DATEADD (d, 3,
                        GETDATE ()) AS 'Нова дата'
```

У результаті виконання запиту до поточної дати буде додано три дні (рис. 4.1).

	Дата	Нова дата
▶	20.12.2017 10:13:34	23.12.2017 10:13:34

Рис. 4.1. **Результат виконання запиту**

У T-SQL можливе перетворення типів даних. Для цього використовують такі **функції перетворення типів**:

STR – використовується для перетворення чисел на символьні дані;

CONVERT і CAST – функції, які дозволяють перетворити значення одного типу даних в інший (якщо такі зміни можливі). Наведемо узагальнений запис цих функцій:

```
CAST ( expression AS data_type [ ( length ) ] )
CONVERT ( data_type [ ( length ) ] , expression [ , style ] )
```

Розглянемо **параметри** цих **функцій**:

expression – будь-який припустимий вираз;

data_type – цільовий тип даних;

length – додатково указується ціле число, що позначає довжину цільового типу даних. Значення за замовчуванням дорівнює 30;

style – цілочисельний вираз, що визначає як функція CONVERT перетворює параметр expression. Якщо вираз style має значення NULL, повертається NULL. Діапазон значень виразу style визначається параметром data_type.

Приклад 4.2. Перетворимо тип дійсного числа за допомогою розглянутих функцій. Спочатку використовуємо функцію CAST (рис. 4.2).

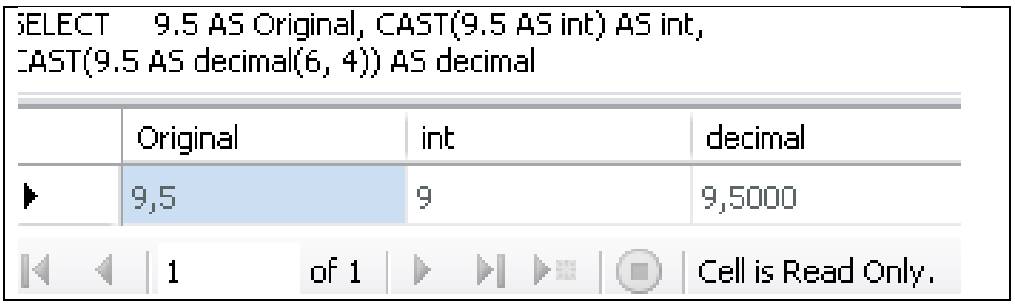


Рис. 4.2. **Результати перетворення чисел за допомогою функції CAST**

Виконаємо ті ж дії з допомогою функції CONVERT (рис. 4.3).

Рис. 4.3. **Результати перетворення чисел з допомогою функції CONVERT**

Результати обох перетворень однакові.

Приклад 4.3. Необхідно створити запит, у якому визначити відсоток ціни закупівлі товару від ціни його продажу, результат перетворити в ціле з допомогою функції CAST (рис. 4.4):

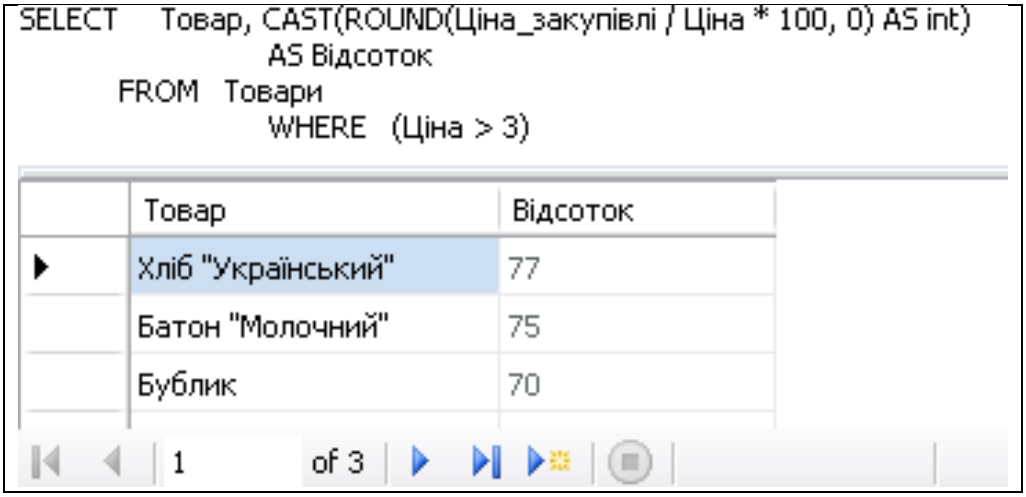


Рис. 4.4. **Результати виконання запиту**

SQL Server гарантує перетворення (перетворення з вихідного типу даних в цільовий, та навпаки).

Приклад 4.4. Необхідно виконати перетворення змінного типу decimal в varbinary і назад (рис. 4.5):

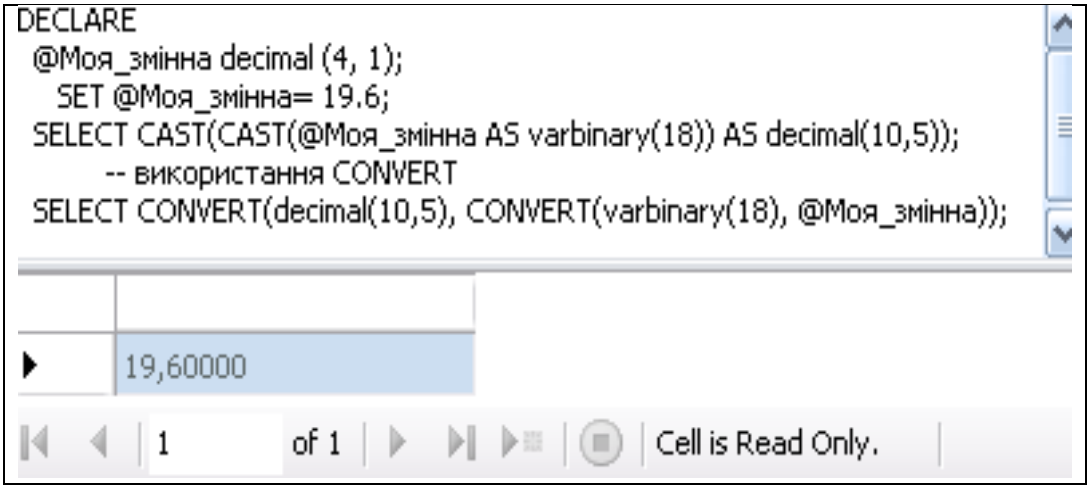


Рис. 4.5. **Результати виконання запиту**

У T-SQL можливе створення типів даних користувача. SQL Server створює в поточній базі даних псевдонім нового типу даних або визначений користувачем тип. Реалізація отриманого псевдоніма ґрунтується на власному системному типі SQL Server.

Приклад 4.5. Необхідно створити псевдонім на основі типу даних varchar.

У цьому прикладі (рис. 4.6) створюється псевдонім на базі певного в системі типу даних varchar (10).

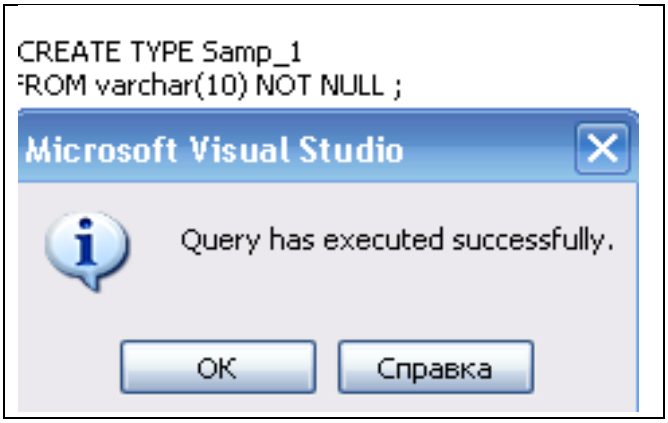


Рис. 4.6. **Результати створення нового типу даних Samp_1**

Широкі можливості T-SQL дозволяють створити табличний тип користувача.

Приклад 4.6. Необхідно створити табличний тип даних, який має два стовпці (рис. 4.7) та визначається користувачем.

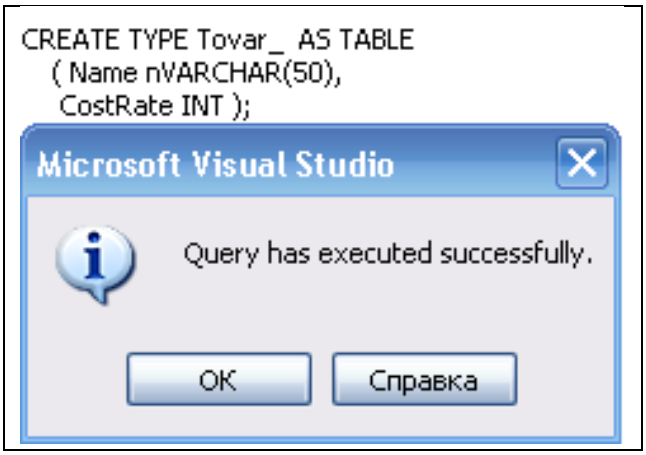


Рис. 4.7. **Результати створення нового табличного типу даних Tovar_**

У результаті виконання запиту створений табличний тип даних, який можна заповнити інформацією (рис. 4.8).

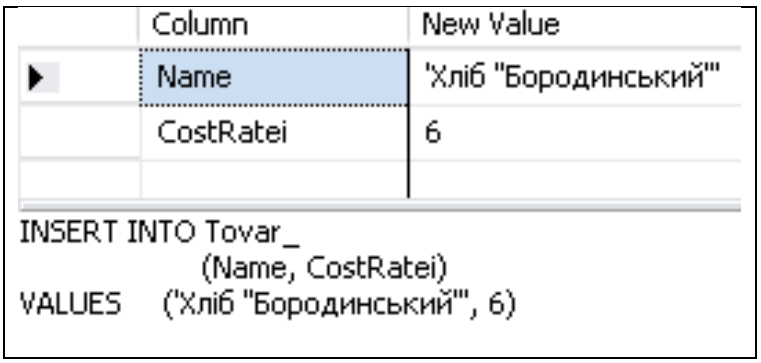


Рис. 4.8. **Результати заповнення об'єкта нового табличного типу даних**

Таким чином, у T-SQL можливе створення різних типів даних користувача.

4.2. Функції в Transact-SQL

4.2.1. Поняття функції. Системні функції

Функції є конструкціями мови SQL, що складаються з декількох інструкцій і мають від нуля до декількох вхідних параметрів, які завжди повертають хоча б одне значення.

Функції можуть бути вбудовані або створювані користувачем. SQL Server містіть велику кількість вбудований функцій, типи яких указані в табл. 4.3.

Таблиця 4.3

Типи функцій

Функції	Опис
Функції, які повертають набори рядків	Повертають об'єкт, який можна використовувати так само, як табличні посилання в SQL-інструкції
Агрегатні функції	Обробляють колекцію значень і повертають одне результівне значення
Ранжувальні функції	Повертають ранжовані значення для кожного рядка в секції
Скалярні функції	Обробляють і повертають одиночне значення. Скалярні функції можна застосовувати всюди, де вираз допустимий

Основні агрегатні функції були розглянуті раніше. Користувачі найбільш частіше використовують скалярні вбудовані функції (табл. 4.4).

Таблиця 4.4

Скалярні вбудовані функції

Категорії функцій	Опис
Функції конфігурації	Повертають дані про конфігурацію
Функції перетворення	Підтримка приведення і перетворення типів даних
Функції роботи з курсором	Повертають дані про курсори
Функції дати та часу	Виконують операції над датою та часом, повертають рядкові та числові значення, а також значення дати та часу
Логічні функції	Виконують логічні операції
Математичні функції	Виконують обчислення, які базуються на числових значеннях, що передаються функції в якості аргументів, і повертають числові значення
Функції метаданих	Повертають дані про бази даних та об'єкти баз даних
Функції безпеки	Повертають дані про користувачів і ролі
Строкові функції	Виконують операції з строковими (char або varchar) даними і повертають строкові або числові значення
Системні функції	Виконують операції над значеннями, об'єктами та параметрами SQL Server і повертають дані про них
Системні статистичні функції	Повертають статистичні данні про систему
Функції обробки тексту та зображень	Виконують операції над текстовими або графічними даними або/і повертають інформацію про значення

Функції бувають детерміновані та недетерміновані. **Детерміновані функції** кожного разу повертають той самий результат, якщо надавати їм той самий набір вхідних значень і використовувати той самий стан бази даних. **Недетерміновані функції** можуть повертати щоразу різні результати, навіть якщо надавати їм той самий набір вхідних значень і використовувати той самий стан бази даних. Наприклад, функція AVG завжди повертає один результат за зазначених умов, а функція GETDATE, що повертає поточну дату та час, завжди повертає різний результат. На детермінізм убудованих функцій вплинути не можна.

Кожна з них детермінована або недетермінована залежно від реалізації в SQL Server.

Для отримання доступу до системних базових таблиць часто використовують спеціалізований механізм компонента Database Engine – системні функції.

Якщо друга інструкція SELECT повертає ідентифікатор таблиці "Товари", використовуючи системну функцію object_id, запис буде таким:

```
USE Func_1;
SELECT object_id
    FROM sys.objects
    WHERE name = 'Товари'
SELECT object_id('Товари')
```

Отриману інформацію можна зберегти у змінній і застосувати в якості параметра для виклику команди або системної збереженої процедури.

Далі наведемо список деяких системних функцій для доступу до системних базових таблиць. Призначення частини з них можна зрозуміти з їх назв:

```
OBJECT_ID (object_name)
OBJECT_NAME (object_id)
USER_ID ([user_name])
USER_NAME ([user_id])
DB_ID ([ db_name])
DB_NAME ([db_id])
@@ERROR
@@IDENTITY
@@ROWCOUNT
```

Розглянемо приклади використання деяких з них.

Наприклад, функція @@ ERROR повертає номер помилки для останньої виконаної інструкції Transact-SQL. Вона повертає 0 (якщо в попередній інструкції Transact-SQL не виникли помилки) або номер помилки (якщо в попередній інструкції виникла помилка). Якщо помилка стосується подання каталогу sys.messages, @@ ERROR містить значення стовпця sys.messages.message_id для цієї помилки. Ви можете переглянути текст, пов'язаний з номером помилки @@ ERROR у sys.messages.

Оскільки функція @@ ERROR очищається і скидається для кожної виконаної інструкції, необхідно відразу після інструкції зберігати її значення в локальну змінну для подальшої перевірки.

Приклад 4.7. Необхідно використати функцію @@ ERROR для виявлення помилки. У цьому прикладі функція @@ ERROR використовується для перевірки обмеження (помилка 547) в інструкції UPDATE.

```
UPDATE  Товари
    SET Товари.Ціна = 4
    WHERE Товари.Код_товару = 2;
IF @@ERROR = 547
    PRINT 'Порушене контрольне обмеження.'
```

Приклад 4.8. Необхідно використати функцію @@ ROWCOUNT для контролю кількості оброблених рядків:

```
UPDATE    Товари
SET              Товари.Ціна = 4
WHERE     Товари.Код_товару = 8;
IF @@ROWCOUNT = 0
 PRINT N'Товар з таким ідентифікатором відсутний.'
```

Приклад 4.9. Необхідно додати рядок у таблицю, яка містить стовпець ідентифікаторів (Код_товару) і застосовує функцію @@ IDENTITY для відображення значення ідентифікатора, що використовується в новому рядку:

```
SELECT MAX(Код_товару) FROM Товари
INSERT INTO Товари (Товар, Ціна, Ціна_закупівлі)
            VALUES ('Бублик', 4.3, 3)
SELECT @@IDENTITY
SELECT MAX(Код_товару) FROM Товари
```

На рис. 4.9. наведено код цього запиту та результати його виконання.

4.2.2. Функції користувача

SQL Server підтримує створення обумовлених користувачем функцій. У реалізації мовою SQL складних алгоритмів, використовуваних більше одного разу, постає питання про збереження розробленого коду

для подальшого застосування. Це завдання можна було б реалізувати за допомогою збережених процедур, однак їх архітектура не дозволяє використовувати процедури безпосередньо в виразах.

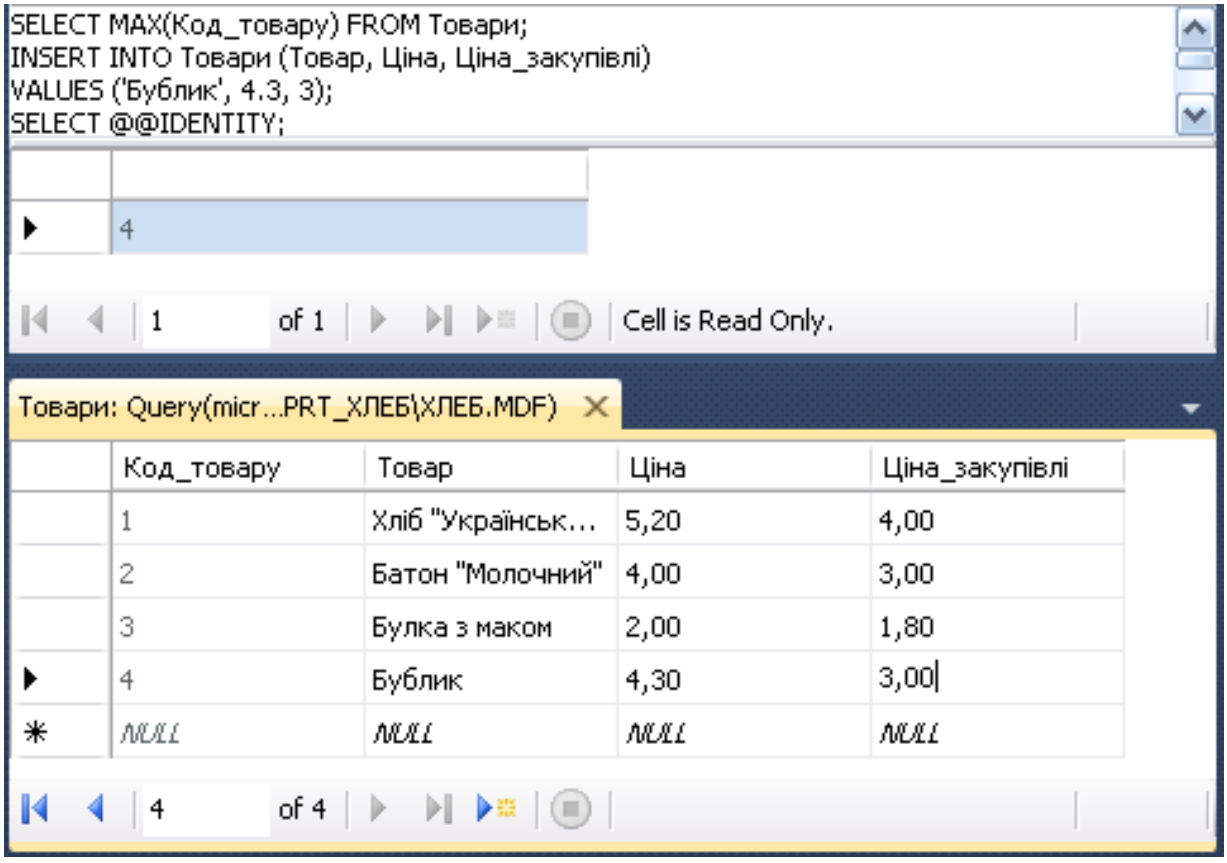


Рис. 4.9. **Запит, що використовує функцію @@IDENTITY**

Функції користувача є самостійні об'єкти бази даних – такі, наприклад, як збережені процедури або тригери. Функція користувача розташовується в певній базі даних і доступна тільки в її контексті.

У *SQL Server* є такі **класи функцій користувача**:

Scalar – функції повертають звичайне скалярне значення. Кожна може включати безліч команд, що об'єднуються в один блок за допомогою конструкції BEGIN ... END;

Inline – функції містять лише одну команду SELECT і повертають користувачеві набір даних у вигляді значення типу даних TABLE;

Multi-statement – функції також повертають користувачеві значення типу даних TABLE, що містить набір даних. Проте в тілі функції знаходиться безліч команд SQL (INSERT, UPDATE тощо); саме з їх допомогою формується набір даних, який треба повернути після виконання функції.

Функції схожі зі збереженими процедурами, але, на відміну від них, можуть застосовуватися в запитах так само, як і системні вбудовані функції. Призначені для користувача функції, які повертають таблиці, можуть стати альтернативою результатам пошуку. Перегляди обмежені одним виразом SELECT, а призначені для користувача функції здатні включати додаткові вирази, що дозволяє створювати більш складні та потужні конструкції.

Визначені користувачем функції створюються за допомогою інструкції CREATE FUNCTION, яка має такий синтаксис:

```
CREATE FUNCTION [schema_name.]function_name
    [( {@param } type [= default]) {,...}
     RETURNS {scalar_type | [@variable] TABLE}
    [WITH {ENCRYPTION | SCHEMABINDING}
    [AS] {block | RETURN (select_statement)}
```

Параметр schema_name визначає ім'я схеми, яка призначається власником створюваної функції, а параметр function_name визначає ім'я цієї функції.

Параметр @param є вхідним параметром функції (формальним аргументом), чий тип даних визначається параметром type.

Параметр default визначає значення за замовчуванням для відповідного параметра функції (також може бути NULL.).

Речення RETURNS визначає тип даних значення, що повертається функцією. Це може бути майже будь-який стандартний тип даних, підтримуваний системою баз даних, включаючи тип даних TABLE. Єдиним типом даних, який не можна вказувати, є тип даних timestamp.

Визначені користувачем функції можуть бути або скалярними, або табличними. Скалярні функції повертають атомарне (скалярне) значення. Це означає, що в реченні RETURNS скалярної функції вказується один зі стандартних типів даних. Функція є табличною, якщо речення RETURNS повертає набір рядків.

Параметр WITH ENCRYPTION у системному каталозі кодує інформацію, що є частиною текстової інструкції CREATE FUNCTION. Таким чином, запобігає несанкціонованому перегляду тексту, який був використаний для створення функції. Дана опція дозволяє підвищити безпеку системи баз даних.

Речення WITH SCHEMABINDING прив'язує функцію до об'єктів бази даних, до яких ця функція звертається. Після цього будь-яка спроба модифікувати об'єкт бази даних, до якого звертається функція, зазнає невдачі. Прив'язка функції до об'єктів бази даних, до яких вона звертається, видаляється лише шляхом зміни функції.

Для того щоб під час створення функції використовувати речення SCHEMABINDING, об'єкти бази даних, до яких звертається функція, повинні відповідати таким **вимогам**:

усі функції, до яких звертається визначувана функція, повинні бути прив'язані до схеми;

усі об'єкти бази даних (таблиці, подання і функції) повинні бути в тій же самій базі даних, що і функція.

Параметр block визначає блок BEGIN/END, що містить реалізацію функції. Останньою інструкцією блоку повинна бути інструкція RETURN з аргументом, значення якого повертається функцією. Усередині блоку BEGIN/END дозволяються тільки такі **інструкції**:

інструкції присвоєння (такі як SET);

інструкції для управління ходом виконання (такі як WHILE і IF);

інструкції DECLARE, що оголошують локальні змінні;

інструкції SELECT, що містять списки стовпців вибірки з виразами, значення яких присвоюються змінним, які є локальними для даної функції;

інструкції INSERT, UPDATE і DELETE, які змінюють змінні з типом даних TABLE, які є локальними для даної функції.

Приклад 4.10. Необхідно створити функцію, яка визначає сумарну кількість проданого товару в задану дату:

```
CREATE FUNCTION
    Продано(@data DATETIME)
RETURNS INT
AS
BEGIN
DECLARE @c INT
SET @c=(SELECT SUM(Кількість)
        FROM Продажі
        WHERE Дата=@data)
RETURN (@c)
END
```

Створимо функцію за допомогою *Microsoft Visual Studio* таким чином.

1. Перебуваючи в Visual Studio з відкритою вкладкою Server Explorer, лівою кнопкою миші виділимо папку Stored Procedures.

2. Натиснемо правою кнопкою миші на виділеній папці Functions і отримаємо меню, в якому необхідно вибрати пункт Add New (рис. 4.10).

Рис. 4.10. **Вибір пункту меню Add New для створення нової функції**

3. Відкривається вікно, в яке поміщається код створюваної функції (рис. 4.11).

```
CREATE FUNCTION dbo.Function3
    (
    /*
    @parameter1 int = 5,
    @parameter2 datatype
    */
    )
RETURNS /* datatype */
AS
    BEGIN
    RETURN /* value */
    END
```

Рис. 4.11. **Вікно для розміщення кода функції**

4. Розмістивши код функції у вікні, закриємо його. Якщо помилки відсутні, то в папці Functions буде створена функція з ім'ям "Продано" (рис. 4.12).

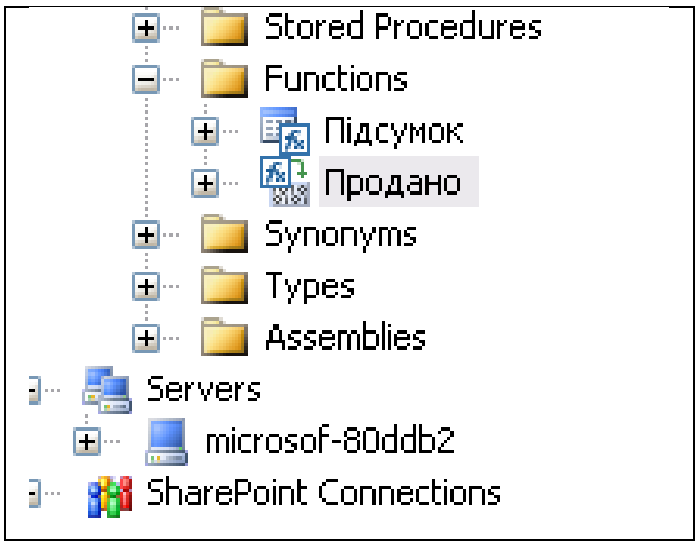


Рис. 4.12. **Місце, де розміщується функція**

5. Для того щоб виконати функцію необхідно натиснути правою кнопкою миші на виділеному її імені в папці Functions і вибрати пункт меню Execute (рис. 4.13).

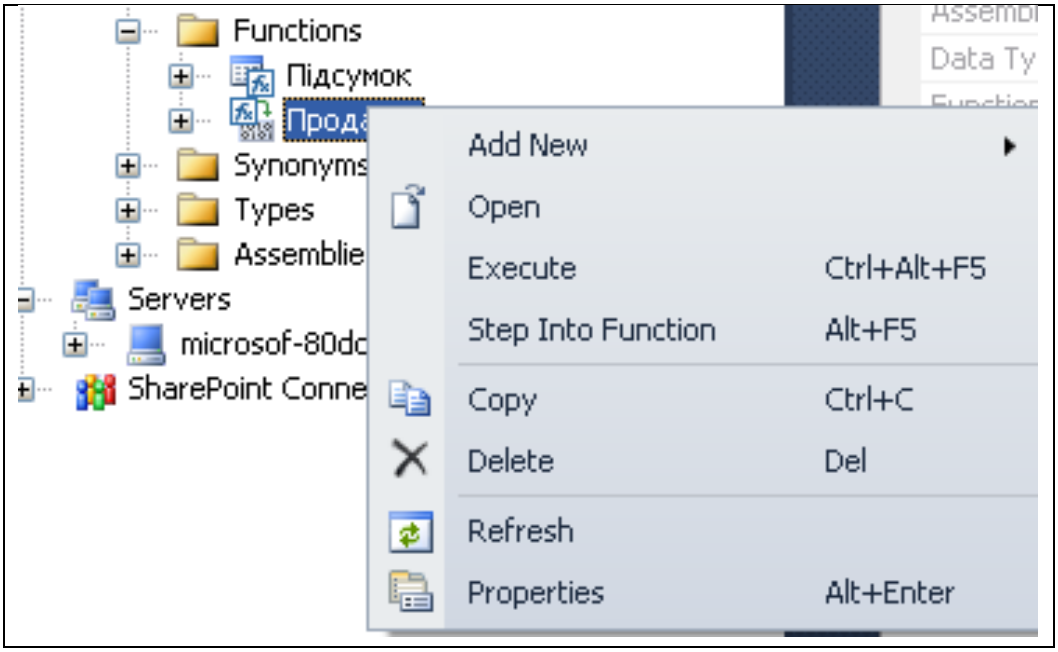


Рис. 4.13. **Вибір пункту меню Execute для виконання функції**

6. У результаті виконання функції відкриється вікно Database Output, в якому будуть відображені дані (рис. 4.14).

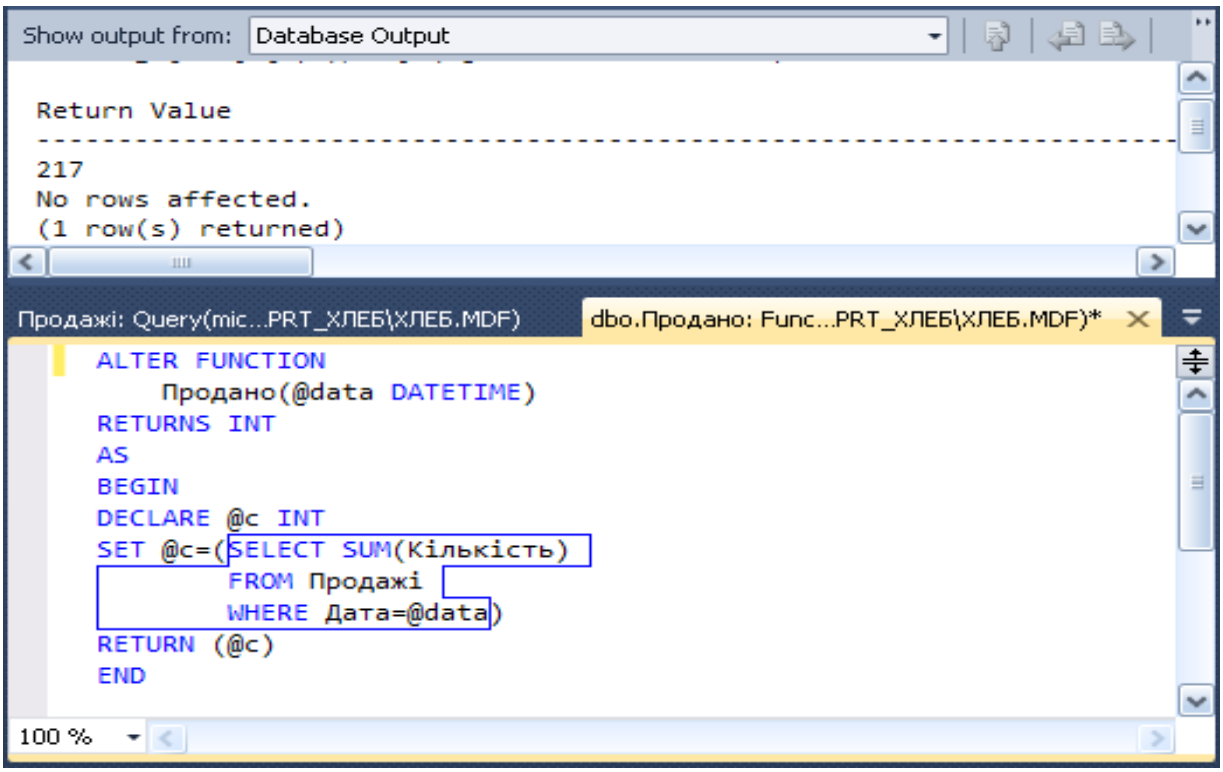


Рис. 4.14. **Результати виконання функції**

Процес створення табличної функції аналогічний. Різниця полягає лише в тому, що для визначення її типу вибирається пункт меню Table-valued Function (рис. 4.15).

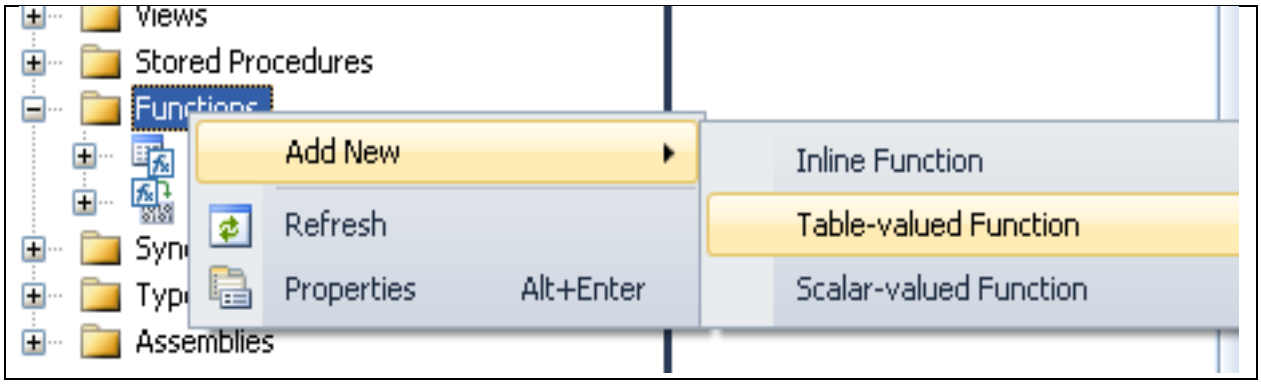


Рис. 4.15. **Вибір пункту меню створення табличної функції**

Приклад 4.11. Необхідно створити та виконати функцію, яка повертає два товари з найбільшою кількістю продажів. Результати виконання функції наведені на рис. 4.16.

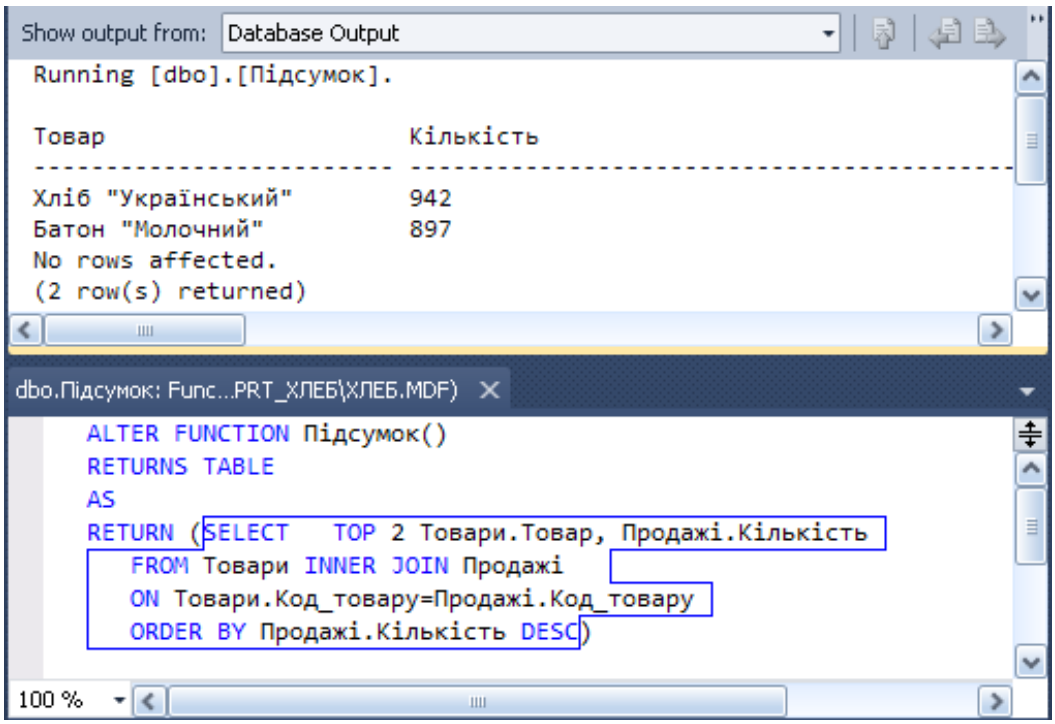


Рис. 4.16. **Результати виконання функції**

Використання функцій дозволяє структурувати систему запитів до бази даних, що суттєво спрощує роботу користувача.

4.3. Конструкції мови T-SQL

4.3.1. Коментарі, пакети, табличні вирази

У процесі створення і супроводу програм дуже корисно використовувати **коментарі**, які суттєво спрощують читання та зміну коду. У Transact-SQL також застосовуються коментарі двох видів: однорядкові та багаторядкові. Для позначення однорядкового коментаря використовують сукупність двох символів – "-". Складний коментар обмежує початок тексту символами – "/*", а кінець тексту символами – "*/". Наприклад:

```
-- це однорядковий коментар;
/ * Це багаторядковий
                коментар * /
```

У процесі створення запитів часто виникає необхідність об'єднання їх в один пакет. **Пакет** є групою з однієї або декількох інструкцій мови

T-SQL, що одноразово надсилаються в *SQL Server* для виконання. *SQL Server* компілює інструкції пакета в єдиний модуль, названий *планом виконання*. Інструкції в цьому плані потім послідовно виконуються.

Кожна *інструкція* Transact-SQL повинна закінчуватися крапкою з комою. Зараз це не обов'язково, але проти інструкцій без крапки з комою є серйозні заперечення, і в наступних версіях *Microsoft SQL Server* таку можливість можуть скасувати.

Для того щоб визначити, коли передається пакет використовується команда GO. Наведемо приклад пакета, який складається з двох команд SELECT:

```
SELECT * FROM Товари;
SELECT * FROM Виробники;
GO
SELECT * FROM Накладні;
```

Спочатку обидві команди передаються серверу та виконуються ним, а їх результати повертаються користувачу. Команди цього пакета компілюються і виконуються як єдина група. Якщо виникають помилки, то не виконується весь пакет. Остання команда цього прикладу знаходиться в окремому пакеті.

Результати виконання пакета запитів наведені на рис. 4.17.

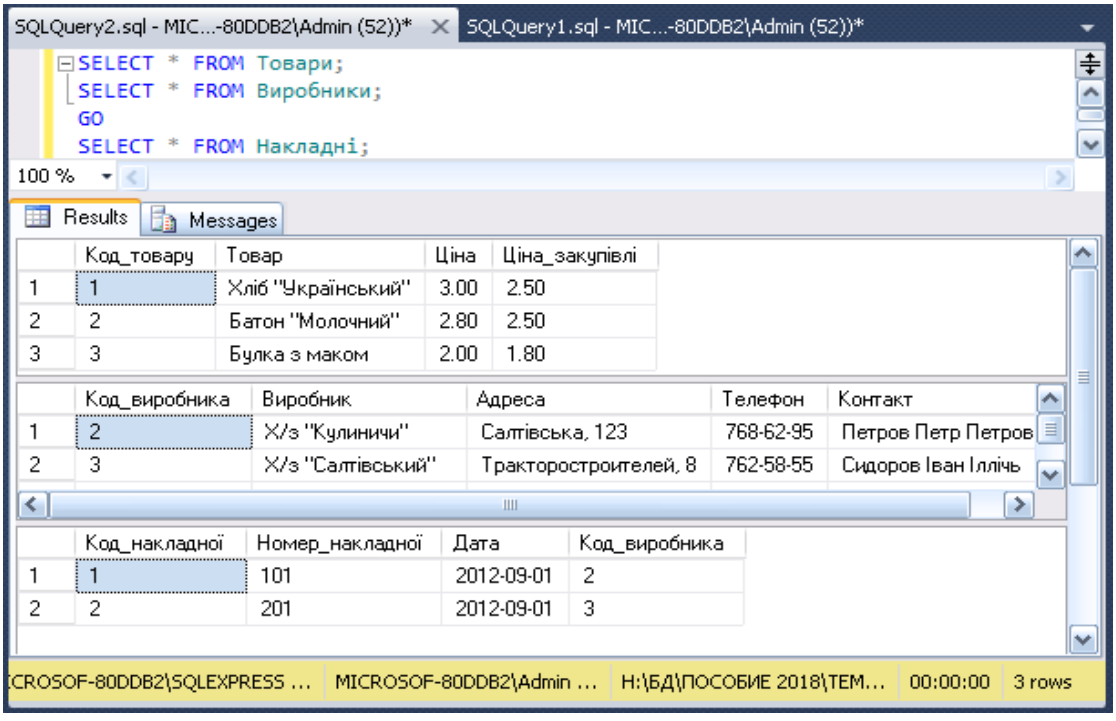


Рис. 4.17. **Результати виконання пакета запитів**

Помилка компіляції (наприклад, синтаксична) припиняє компіляцію. Тому ніякі інструкції в пакеті не виконуються.

Помилка виконання (наприклад арифметичне переповнення або порушення обмеження) призводить до одного з двох результатів:

більшість помилок часу виконання зупиняють поточну інструкцію та подальші за нею в пакеті інструкції;

деякі помилки часу виконання (наприклад порушення обмежень) зупиняють тільки поточну інструкцію, всі інші інструкції в пакеті будуть виконані.

Єдиний виняток – якщо пакет знаходиться в транзакції, і через помилки буде виконаний відкат транзакції. У цьому випадку буде відкат будь-яких незафіксованих змін даних, зроблених перед помилкою виконання.

Припустимо, що в пакеті є п'ять інструкцій. Якщо п'ята інструкція містить синтаксичну помилку, інструкції в пакеті не виконаються. Якщо ж пакет скомпільовано, а потім третя інструкція виконається невдало, це не вплине на результати двох перших інструкції, тому що вони вже виконані.

SQL Server забезпечує повторну компіляцію на рівні інструкцій. Це означає, що, якщо інструкція викликає повторну компіляцію, повторно компілюється тільки ця інструкція, а не весь пакет. Розглянемо приклад, в якому в одному пакеті містяться одна інструкція CREATE TABLE та три інструкції INSERT.

```
      CREATE TABLE Tdob (a int) ;
      INSERT INTO Tdob VALUES (1) ;
      INSERT INTO Tdob VALUES (1,1) ;
      INSERT INTO Tdob VALUES (3) ;
GO
          SELECT * FROM Tdob;
```

Спочатку виконується процес компіляції. Інструкція CREATE TABLE компілюється. Проте оскільки таблиця Tdob ще не створена, інструкції INSERT НЕ компілюються.

Потім пакет починає виконуватися. Створюється таблиця. Перша інструкція INSERT компілюється і негайно виконується. Таблиця тепер містить один рядок. Потім компілюється друга інструкція INSERT. Компіляція закінчується помилкою, і виконання пакету завершується. Інструкція SELECT повертає один рядок (рис. 4.18).

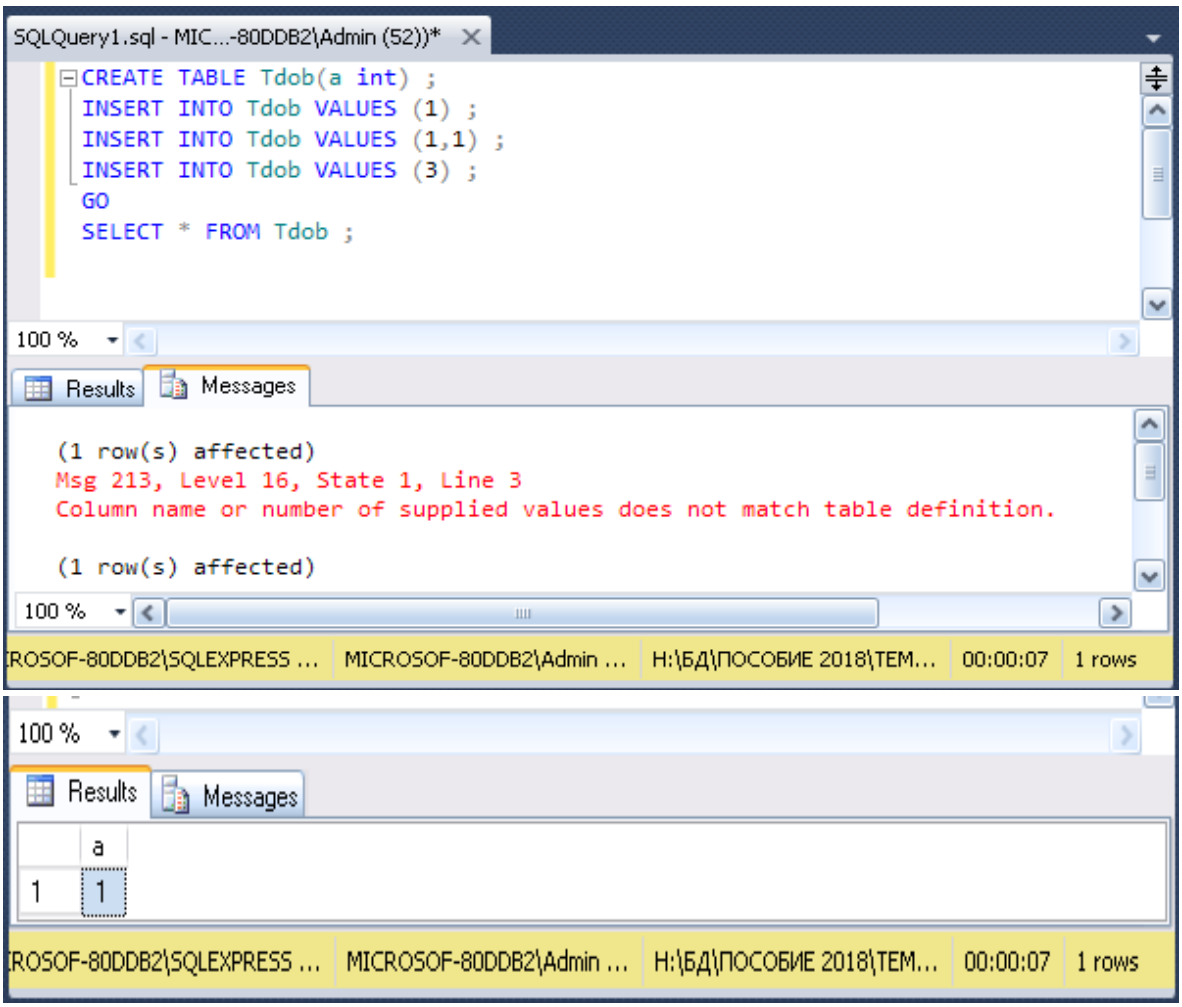


Рис. 4.18. **Результати виконання пакету запитів**

У використанні пакетів застосовуються такі **правила**:

інструкції CREATE DEFAULT, CREATE FUNCTION, CREATE PROCEDURE, CREATE RULE, CREATE SCHEMA, CREATE TRIGGER і CREATE VIEW не можуть бути об'єднані в пакеті з іншими інструкціями. Інструкція CREATE повинна бути в пакеті першою; всі наступні інструкції в цьому пакеті розглядаються як частина визначення інструкції CREATE;

в тому самому пакеті можна спочатку змінити таблицю і потім звернутися до нових стовпців;

ключове слово EXECUTE потрібно, якщо інструкція EXECUTE не є першою в пакеті. Якщо EXECUTE – перша інструкція в пакеті, то ключове слово EXECUTE не потрібно.

Важливим компонентом, який використовується для створення складних і багатоступеневих запитів можуть слугувати табличні вирази.

Табличні вирази можна уявити як тимчасові результівні набори, що існують в області виконання одиничних інструкцій SELECT, INSERT, UPDATE, DELETE або CREATE VIEW.

Табличні вирази, як і похідні таблиці, не зберігаються в базі даних у вигляді об'єктів; час їх життя обмежений тривалістю запиту. Але, на відміну від похідних таблиць, табличні вирази можуть посилатися самі на себе; той самий запит може посилатися на них кілька разів.

Табличні вирази призначені для забезпечення таких дій [38]:

створення рекурсивних запитів;

заміна подання в тих випадках, коли використання подання не виправдано, тобто коли немає необхідності зберігати в метаданих бази його визначення:

групування за стовпцем, який є похідним від скалярного підзапиту вибірки або функції, або функції, що має зовнішній доступ;

багаторазові посилання на результуючу таблицю з тієї самої інструкції.

Застосування табличних виразів дозволяє значно підвищити читаність і спростити роботу зі складними запитами, розбивши їх на окремі логічні будівельні блоки. З них можна створювати більш складні проміжні табличні вирази з метою формування кінцевого результівного набору.

Табличні вирази можуть бути визначені в призначених для користувача підпрограмах (функціях, збережених процедурах, тригерах, поданнях).

Табличний вираз складається з імені виразу, необов'язкового списку стовпців, що визначає його запит. Після визначення табличного виразу на нього можна посилатися з інструкцій SELECT, INSERT, UPDATE і DELETE як на таблицю або подання. Табличний вираз також можна вказати в реченні CREATE VIEW у визначальної інструкції SELECT.

Базова структура синтаксису табличного виразу:

```
WITH expression_name [ ( column_name [,...n] ) ]
AS
( CTE_query_definition )
```

Список імен стовпців необов'язковий тільки в тому випадку, якщо всім результівним стовпцям у визначенні запиту присвоєні унікальні імена.

Інструкція для звернення до табличного виразу має вигляд:

```
SELECT <column_list>
FROM expression_name;
```

Приклад 4.12. У цьому прикладі показані компоненти структури табличного виразу: ім'я виразу, список стовпців і запит. Результат виконання табличного виразу наведений на рис. 4.19.

```
      WITH Продажі_С (Дата, Код_товару, Кількість)
      AS
      -- Визначення табличного виразу
      (
SELECT YEAR(Дата) AS Дата, Код_товару, Кількість
          FROM Продажі
      )
      -- Запит-посилання на табличний вираз
SELECT Дата, COUNT(Код_товару) AS Код_товару, Кількість
      FROM Продажі_С
      GROUP BY Дата, Кількість
```

Results | Messages

	Дата	Код_товару	Кількість
1	2011	1	11
2	2013	1	100
3	2012	1	150
4	2012	1	180
5	2012	1	200
6	2014	1	333
7	2011	1	444
8	2013	1	555
9	2011	5	777
10	2012	4	777
11	2013	1	777
12	2012	1	888
13	2013	1	999

ESS ... | MICROSOF-80DDB2\Admin ... | H:\БД\ПОСОБИЕ 2018\ТЕМ... | 00:00:00 | 13 rows

Рис. 4.19. **Результати використання табличного виразу**

Табличний вираз Продажі_С містить три стовпці (Дата, Код_товару, Кількість) і визначає загальне число продажів за роками для всіх видів товарів.

4.3.2. Процедурні конструкції мови T-SQL

У Transact-SQL існують спеціальні команди, які дозволяють управляти потоком виконання сценарію, перериваючи його або скеровуючи в потрібну логіку.

Конструкція (IF ... ELSE) визначає групу команд (блок команд) або одиничну команду, які виконуються залежно від деякої умови. Конструкція має такий синтаксис:

```
IF логічний_вираз    {блок_команд або команда}
[ ELSE               {блок_команд або команда}]
```

Логічний_вираз – вираз, який повертає константи TRUE або FALSE. Якщо вираз містить інструкцію SELECT, тоді вона обмежується дужками.

Наведемо приклад використання простої конструкції (IF … ELSE), в якій виводиться інформація залежно від кількості рядків у таблиці "Товари" (рис. 4.20).

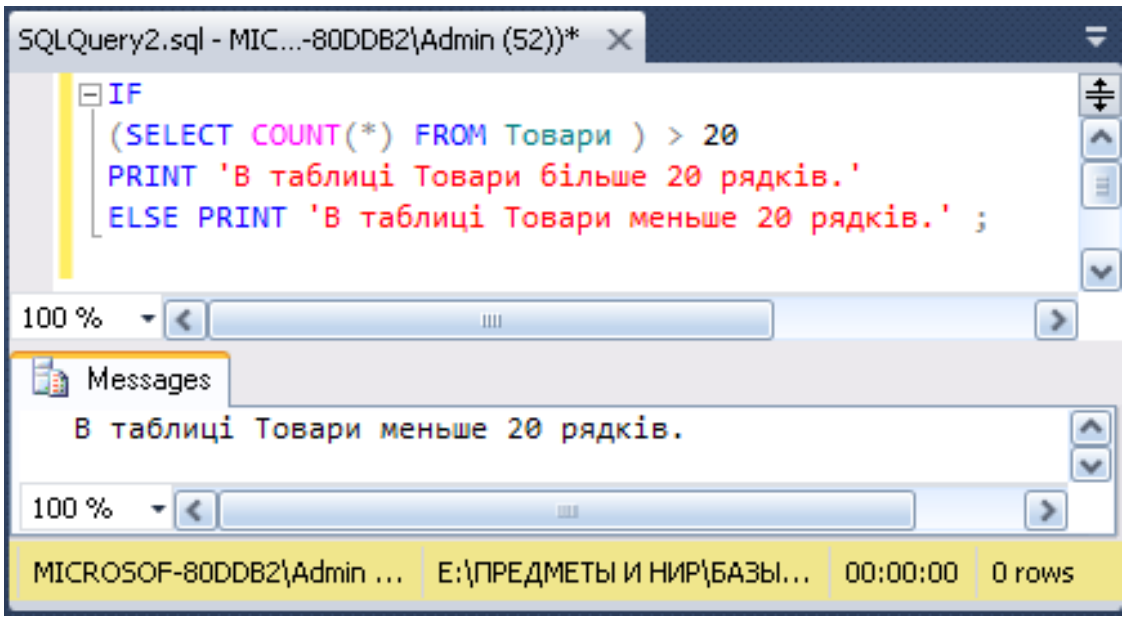


Рис. 4.20. **Результати виконання запиту з використанням конструкції IF**

Для того щоб застосовувати декілька команд в розділі IF або ELSE, необхідно використати блок BEGIN…END. Усі конструкції, що знаходяться між BEGIN та END, є частиною цього блока.

Приклад 4.13. Необхідно створити запит, який перевіряє кількість записів у таблиці "Товари". Якщо їх кількість менше 20, то додати до таблиці один запис і вивести повідомлення про це. Якщо записів більше 20, то видалити записи, які задовольняють такій умові:

```
(Код_товару > 21) AND  (Ціна<2)
```

Результати виконання запиту наведені на рис. 4.21.

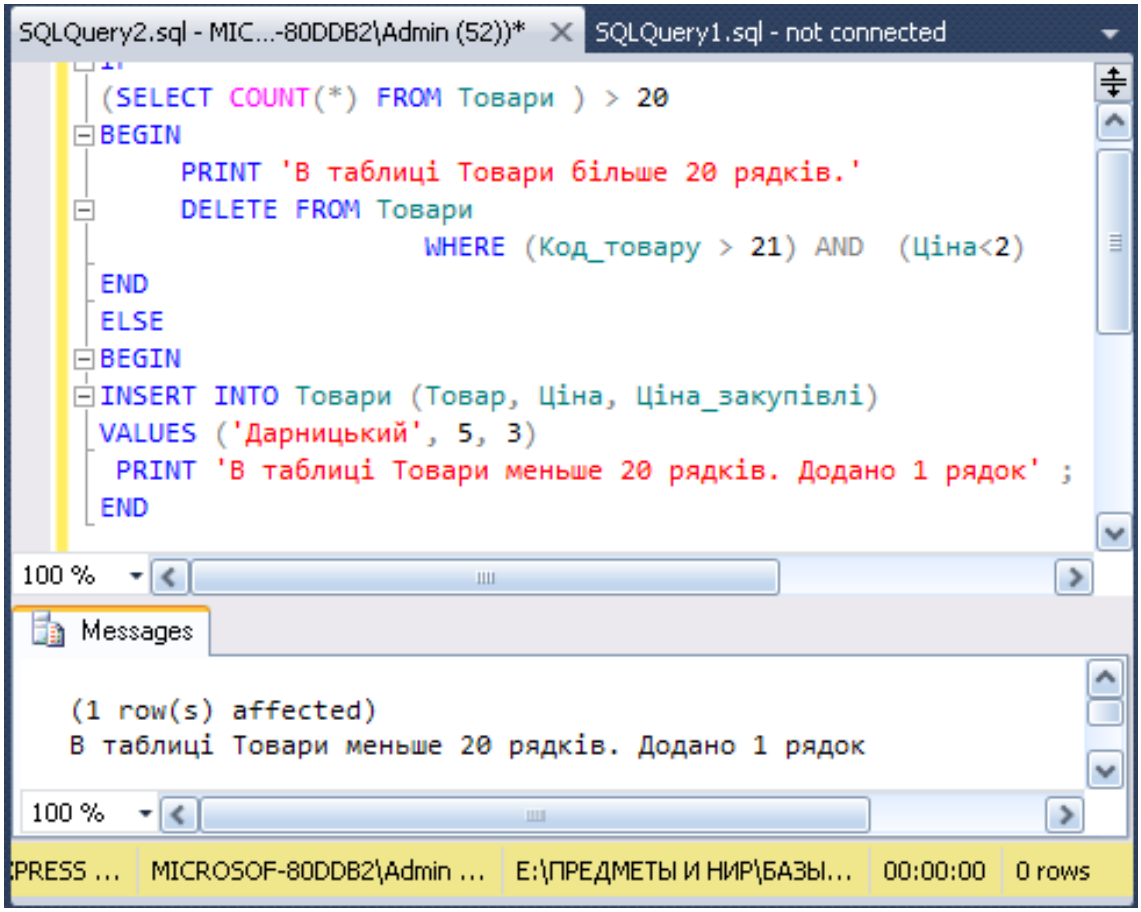


Рис. 4.21. **Результати виконання запиту з використанням конструкції BEGIN…END**

Для перевірки списку умов і отримання одного з декількох можливих результатів використовується інструкція CASE. Функція CASE має два **формати виклику**:

простий CASE – для порівняння заданого виразу з безліччю простих виразів з метою визначення результату;

пошуковий CASE – для безлічі булевих (логічних) виразів.

Простий CASE має такий синтаксис:

```
CASE вхідний_вираз
         WHEN вираз_збігу THEN вираз_результату [... n]
         [ELSE інакший_ вираз_результату]
END
```

Пошуковий CASE має такий синтаксис:

```
CASE
     WHEN логічний_вираз THEN вираз_результату [... n]
     [ELSE інакший_ вираз_результату]
END
```

Розглянемо *аргументи* цих *конструкцій*:

вхідний_вираз – будь-який припустимий вираз;

WHEN вираз_збігу – простий вираз, з яким порівнюється аргумент. Типи даних аргументу вхідний_вираз і вираз_збігу повинні бути однаковими або неявно приведені один до одного;

THEN вираз_результату – аналізується вираз_результату, якщо порівняння вхідний_вираз і вираз_збігу дає в результаті TRUE або логічний_вираз приймає значення TRUE, аргумент вираз_результату може бути будь-якого припустимого типу;

ELSE інакший_вираз_результату – аналізується інакший_вираз_ результату, якщо жодна з операцій порівняння не дає в результаті TRUE;

Інструкцію CASE можна використовувати в таких операторах, як SELECT, UPDATE, DELETE і SET, а також реченнях IN, WHERE, ORDER BY і HAVING.

Приклад 4.14. Оператор SELECT з простим виразом CASE:

```
SELECT Товар 'Товари', 'Оцінка коштовності' =
      CASE Ціна
         WHEN 3 THEN 'ціна задовільна'
         WHEN 2 THEN 'ціна низька'
         ELSE 'невідомо'
      END
FROM Товари
```

Приклад 4.15. Оператор SELECT з пошуковим виразом CASE:

```
SELECT Товар 'Товари', 'Оцінка коштовності' =
      CASE
         WHEN Ціна = 3 THEN 'ціна задовільна'
         WHEN Ціна = 2 THEN 'ціна низька'
         ELSE 'невідомо'
      END
FROM Товари
```

Для забезпечення циклічних повторень обчислень застосовується інструкція WHILE, яка повторює оператор або блок операторів до тих пір, поки зазначена умова виконується.

Зазвичай з інструкцією WHILE використовуються дві інструкції мови Transact-SQL: BREAK або CONTINUE. Інструкція BREAK забезпечує вихід з самого внутрішнього циклу WHILE, а інструкція CONTINUE заново починає цикл WHILE. Програма може виконати інструкцію BREAK у разі, якщо більше немає рядків для обробки. Інструкцією CONTINUE можна користуватись у тому випадку, якщо виконання коду слід продовжити.

Приклад 4.16. Необхідно створити запит, що задовольняє таким умовам. У разі, якщо середня ціна товарів у таблиці "Товари" менша 5,7, цикл WHILE збільшує ціни на 0,5, а потім вибирає максимальну. У тому випадку, якщо максимальна ціна менша або дорівнює 6,5, цикл WHILE повторюється і знову змінює ціни. Цей цикл продовжує змінювати ціни до тих пір, поки максимальна ціна не буде більше 6,5; потім виконання циклу WHILE припиняється.

Вміст таблиці "Товари" приведено на рис. 4.22.

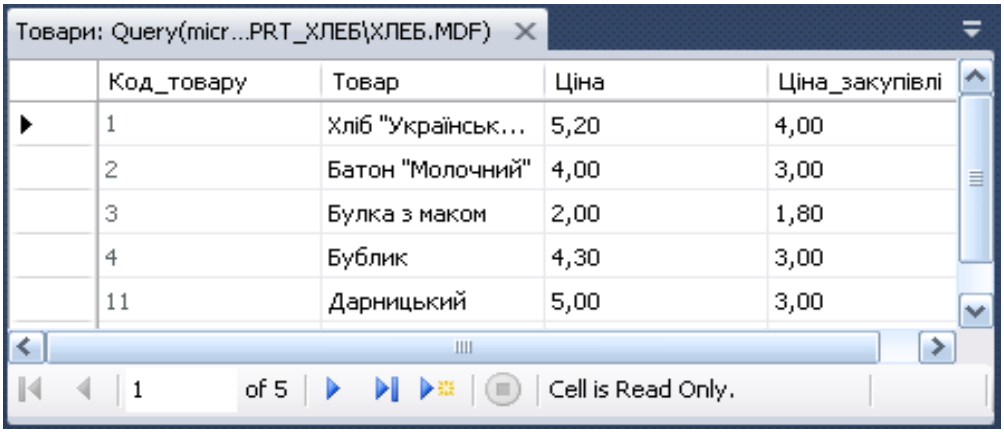


Рис. 4.22. **Інформація, що зберігається в таблиці "Товари" до виконання запиту**

На рис. 4.23 зображений код запиту та повідомлення про його виконання.

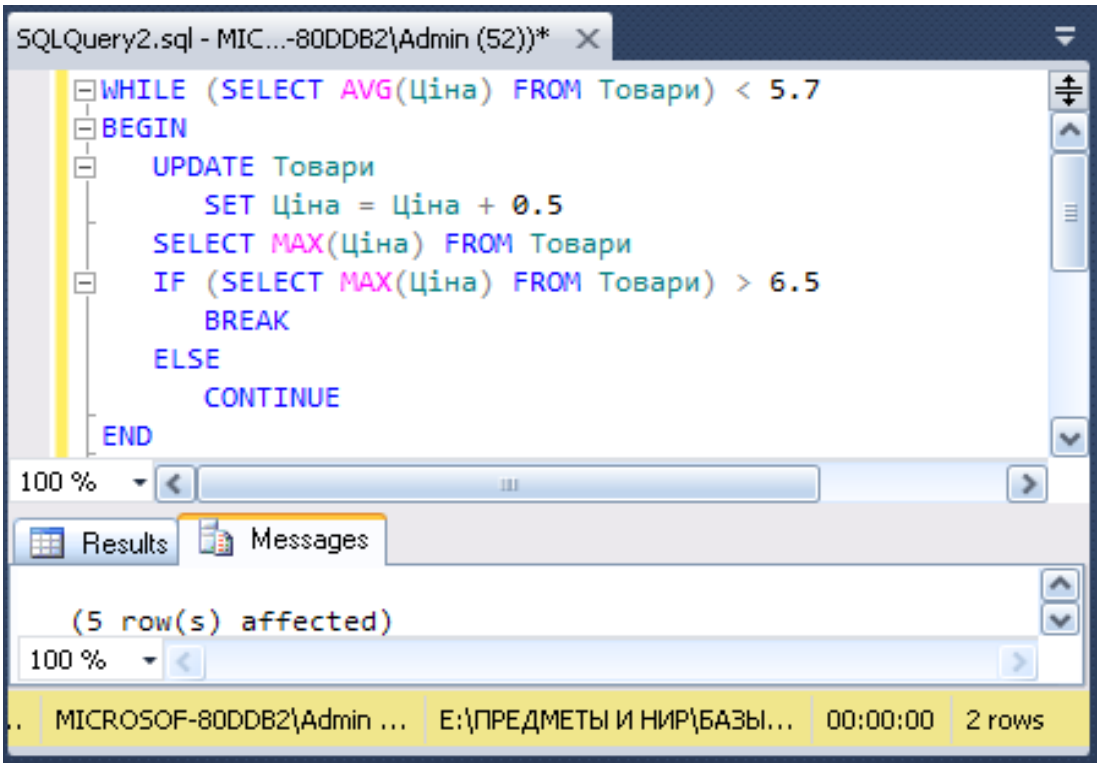


Рис. 4.23. **Код запиту та повідомлення про його виконання**

На рис. 4.24 показана інформація, що зберігається в таблиці "Товари" після виконання запиту.

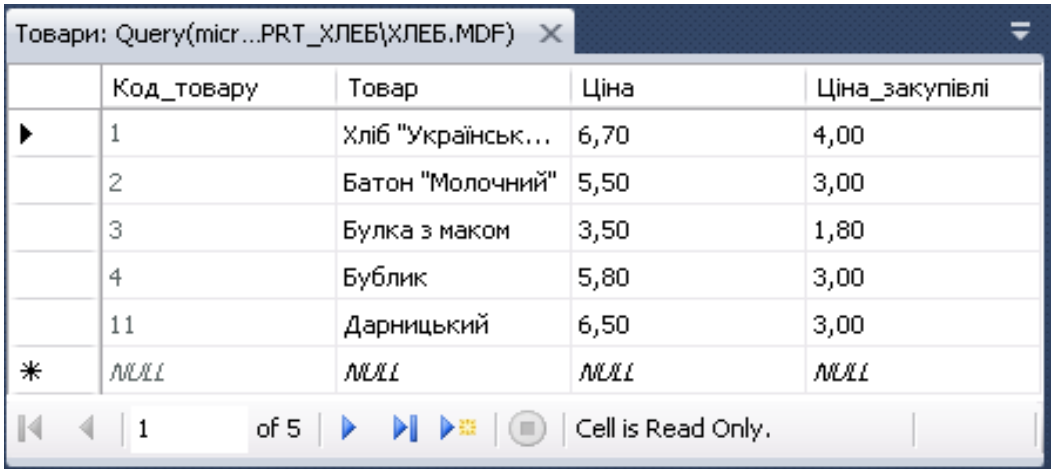


Рис. 4. 24. **Інформація, що зберігається в таблиці "Товари" після виконання запиту**

Таким чином, процедурні конструкції T-SQL містять інструменти, які є у звичайних мовах програмування і суттєво розширюють можливості мови SQL.

4.3.3. Обробка помилок у T-SQL

У ході виконання запитів виникають помилки, що потребують виявлення та обробки. Інструкція RAISERROR передає стан, коди та повідомлення про помилки. Ця інструкція дозволяє використовувати стандартне повідомлення про помилку або сформувати нове.

Простий RAISERROR має такий синтаксис:

```
RAISERROR ( {код_помилки  або  символьний_рядок },
            код_серйозності, стан [  ,  список_аргументів  ] )
      [ WITH параметр ]
```

Інструкція RAISERROR може:

повертати призначене для користувача повідомлення про помилку, створене за допомогою системної збереженої процедури sp_addmessage. Це повідомлення з номерами, що перевищують 50 000, які можна переглянути в поданні каталогу sys.messages;

повертати рядок повідомлення, зазначений в інструкції RAISERROR;

призначати повідомленням номер, серйозність і стан помилки;

запросити реєстрацію помилки в журналі помилок компонента Database Engine та журналі додатків *Microsoft Windows*;

замінити значення аргументів у тексті повідомлення, аналогічно функції printf_s мови C.

Приклад 4.17. Необхідно створити запит, який перевіряє кількість записів у таблиці, сформувати повідомлення користувача про наявність помилок.

У наступному запиті (рис. 4.25) повідомлення про помилку формується з допомогою інструкції RAISERROR, а також функції @@ERROR.

Помилка може бути визначена в блоці TRY…CATCH, який дозволяє продовжити виконання команд (TRY) за відсутності помилок. У протилежному випадку необхідна обробка помилки (CATCH), для чого використовуються інструкції визначення помилок.

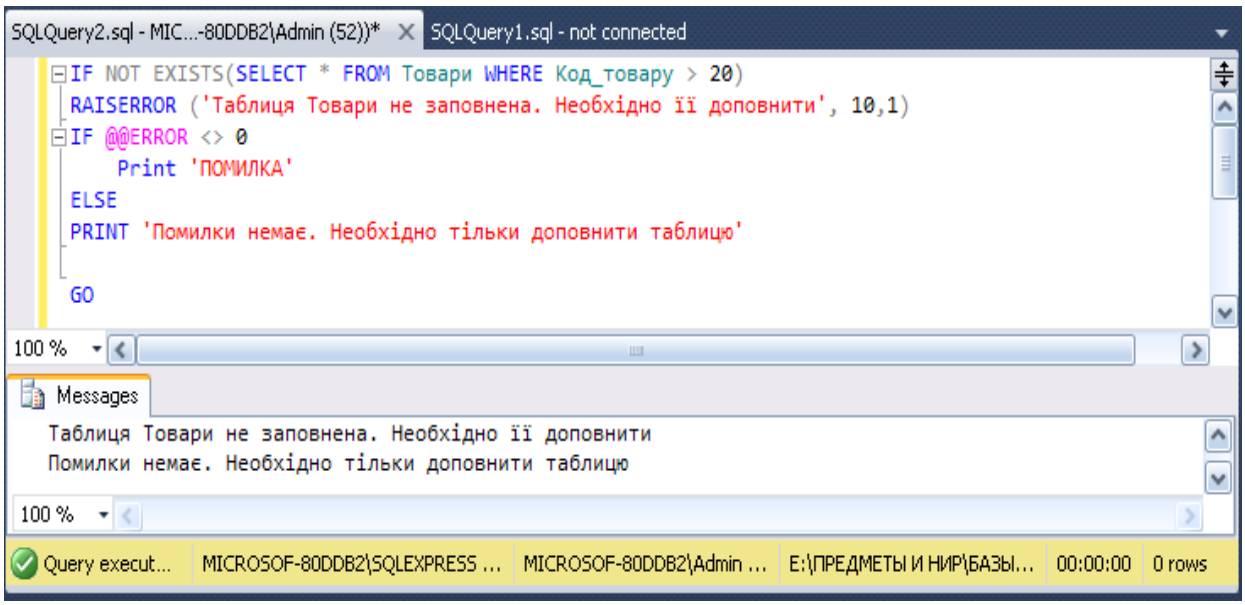


Рис. 4.25. **Код запиту та повідомлення про його виконання**

Приклад 4.18. Необхідно створити запит для контролю оновлення даних з формуванням інформації про можливі помилки.

```
BEGIN TRY
UPDATE Товари SET Ціна = 8
WHERE Товар = 'Хліб'
END TRY
BEGIN CATCH
SELECT    ERROR_NUMBER() AS Er_Number,
          ERROR_MESSEGE() AS Er_Number;
END CATCH
```

Таким чином, мова Transact-SQL має дуже широкі можливості для контролю і визначення помилок у створенні запитів.

4.4. Робота з курсорами

Запит до реляційної бази даних зазвичай повертає кілька записів даних, але додаток за один раз обробляє лише один з них. Навіть якщо він має справу одночасно з декількома рядками (наприклад, виводить дані в формі електронних таблиць), їх кількість, як і раніше, обмежена. Крім того, при модифікації, видаленні або додаванні даних робочою одиницею можливий ряд дій. У цій ситуації на перший план виступає концепція курсора, і в такому контексті курсор – покажчик на ряд записів.

Курсор у SQL – це область у пам'яті бази даних, яка призначена для зберігання останнього оператора SQL. Якщо поточний оператор – запит до бази даних, у пам'яті зберігається і рядок даних запиту, який називають *поточним значенням* або *поточним рядком курсора*. Зазначена область в пам'яті пойменована та доступна для прикладних програм.

Зазвичай курсори використовують для вибору з бази даних деякої підмножини рядків. У кожен момент часу прикладною програмою може бути перевірений один рядок курсора. Курсори часто застосовують в операторах SQL, що вбудовані в написані мовами процедурного типу прикладні програми. Деякі з них неявно створюються сервером бази даних, тоді як інші визначають програмісти.

SQL Server підтримує три **види курсорів**:

курсори SQL, які застосовують переважно всередині тригерів, збережених процедур і сценаріїв;

курсори сервера, які діють на сервері та реалізують програмний інтерфейс додатків для ODBC, OLE DB, DB_Library;

курсори клієнта, які реалізуються для самого клієнта. Вони вибирають весь результівний набір рядків із сервера і зберігають його локально. Це дозволяє прискорити операції обробки даних за рахунок зниження втрат часу на виконання мережевих операцій.

Різні типи багатокористувацьких додатків вимагають і різних типів організації паралельного доступу до даних. Деякі програми потребують негайного доступу до інформації про зміни в базі даних. Це характерно для систем резервування квитків. В інших випадках (наприклад, у системах статистичної звітності) важлива стабільність даних, адже, якщо вони постійно модифікуються, програми не зможуть ефективно відображати інформацію. Різним програмам потрібні різні реалізації курсорів.

У середовищі SQL Server типи курсорів розрізняють за можливостями. Тип курсора визначається на стадії його створення, він не може бути змінений. Деякі типи курсорів можуть виявляти зміни, зроблені іншими користувачами в рядках, включених в результівний набір. Однак *SQL Server* відстежує зміни таких рядків тільки на стадії звернення до рядка, не дозволяючи відстежувати зміни, коли рядок вже зчитаний.

SQL Server підтримує курсори статичні, динамічні, послідовні і керовані набором ключів.

У схемі зі *статичним курсором* інформація читається з бази даних один раз і зберігається у вигляді моментального знімка (станом на де-

який момент часу), тому зміни, внесені в базу даних іншим користувачем, не помітні. На час відкриття курсора сервер установлює блокування на всі рядки, включені в його повний результівний набір. Статичний курсор не змінюється після створення і завжди відображає той набір даних, який існував на момент його відкриття.

Якщо інші користувачі змінять у вихідній таблиці включені в курсор дані, це ніяк не вплине на статичний курсор.

У статичний курсор неможливо вносити зміни, тому він завжди відкривається в режимі "тільки для читання".

Динамічний курсор підтримує дані в "живому" стані, але це вимагає витрат мережевих і програмних ресурсів. З використанням динамічних курсорів повна копія вихідних даних не створюється. Динамічна вибірка з вихідних таблиць виконується тільки за зверненням користувача до тих або інших даних. На час вибірки сервер блокує рядки, а всі зміни, що вносяться користувачем в повний результівний набір курсору, будуть видні в курсорі. Однак якщо інший користувач вніс зміни вже після вибірки даних курсором, то вони не позначаться в курсорі.

Курсор, керований набором ключів, знаходиться посередині між цими крайнощами. Записи ідентифікуються на момент вибірки, і тим самим відслідковуються зміни. Такий тип курсора корисний для реалізації прокрутки назад – тоді додавання і видалення рядків не буде видно, поки інформація не оновиться, а драйвер вибирає нову версію запису, якщо в неї були внесені зміни.

Послідовні курсори не дозволяють виконувати вибірку даних у зворотному напрямі. Користувач може вибирати рядки тільки від початку до кінця курсора. Послідовний курсор не зберігає набір усіх рядків. Вони зчитуються з бази даних, як тільки вибираються в курсорі, що дозволяє динамічно відображати всі зміни, що вносяться користувачами в базу даних за допомогою команд INSERT, UPDATE, DELETE. У курсорі видно найостанніший стан даних.

Статичні курсори забезпечують стабільний погляд на дані. Вони придатні для систем "складування" інформації: додатків для систем звітності або для статистичних та аналітичних цілей. Крім того, статичний курсор краще за інші справляється з вибіркою великої кількості даних. Навпаки, в системах електронних купівель або резервування квитків необхідне динамічне сприйняття нової інформації в міру внесення змін. У таких випадках використовується динамічний курсор. У цих додатках

обсяг переданих даних, як правило, невеликий, а доступ до них здійснюється на рівні окремих записів.

Відповідно до стандарту SQL у роботі з курсором можна виділити такі **основні дії**:

створення або оголошення курсора (DECLARE);

відкриття курсора, тобто наповнення його даними, які зберігаються в багаторівневій пам'яті (OPEN);

вибірка з курсора та зміна з його допомогою рядків даних (FETCH);

закриття курсора, після чого він стає недоступним для користувача програм (CLOSE);

звільнення курсора, тобто видалення його як об'єкта, оскільки його закриття необов'язково звільняє асоційовану з ним пам'ять (DEALLOCATE).

У різних реалізаціях визначення курсора може мати деякі відмінності. Так, наприклад, іноді розробник повинен явним чином звільнити пам'ять, що виділяється для курсора. Після звільнення курсора асоційована з ним пам'ять також звільняється. Тоді стає можливим повторне використання його імені. В інших реалізаціях із закриттям курсора звільнення пам'яті відбувається неявним чином. Відразу після відновлення вона стає доступною для інших операцій: відкриття іншого курсора і т. д.

У середовищі *MS SQL Server* прийнятий такий синтаксис команди створення курсора:

```
DECLARE ім'я_курсора CURSOR [LOCAL | GLOBAL]
[FORWARD_ONLY | SCROLL]
[STATIC | KEYSET | DYNAMIC | FAST_FORWARD]
[READ_ONLY | SCROLL_LOCKS | OPTIMISTIC]
[TYPE_WARNING]
FOR SELECT_оператор
[FOR UPDATE [OF ім'я_стовпця[,...n]]]
```

Розглянемо **параметри оператора створення курсора**:

з використанням ключового слова LOCAL буде створений локальний курсор, який видно тільки в межах пакета, що його створив. Після закінчення роботи пакета курсор неявно знищується. Щоб передати вміст курсора за межі конструкції, що його створила, необхідно привласнити його параметру аргумент OUTPUT;

якщо вказане ключове слово GLOBAL, створюється глобальний курсор, який існує до закриття поточного з'єднання;

зі вказівкою FORWARD_ONLY вибірку даних можна здійснювати тільки в напрямі від першого рядка до останнього;

зі вказівкою SCROLL створюється курсор, який дозволяє звертатися до даних в будь-якому порядку і в будь-якому напрямі;

зі вказівкою STATIC створюється статичний курсор;

зі вказівкою KEYSET створюється ключовий курсор;

зі вказівкою DYNAMIC створюється динамічний курсор;

якщо для курсора READ_ONLY вказати аргумент FAST_FORWARD, то створений курсор буде оптимізований для швидкого доступу до даних. Цей аргумент не можна використовувати разом з аргументами FORWARD_ONLY і OPTIMISTIC;

у курсорі, створеному із зазначенням аргументу OPTIMISTIC, забороняються зміна та видалення рядків, які були змінені після відкриття курсора;

зі вказівкою аргументу TYPE_WARNING сервер буде інформувати користувача про неявну зміну типу курсору, якщо він несумісний із запитом SELECT.

Для відкриття курсора та наповнення його даними використовується така команда:

```
OPEN {{[GLOBAL]ім'я_курсора }
                  |@ім'я_змінної_курсора}
```

Після відкриття курсора відбувається виконання пов'язаного з ним оператора SELECT, вихідні дані якого зберігаються в пам'яті.

Відразу після відкриття курсора можна вибрати його вміст (результат виконання відповідного запиту) за допомогою команди:

```
FETCH [[NEXT | PRIOR | FIRST | LAST
  | ABSOLUTE {номер_рядка
  | @змінна_номера_рядка}
  | RELATIVE {номер_рядка |
    @змінна_номера_рядка}]
  FROM ]{{[GLOBAL ] ім'я_курсора }|
    @ ім'я_змінної_курсора }
  [INTO @ ім'я_ змінної [,...n]]
```

Розглянемо **аргументи оператора FETCH**:

зі вказівкою FIRST буде повернутий найперший рядок повного результівного набору курсора, який стає поточним рядком;

зі вказівкою LAST повертається найостанніший рядок курсору, він стає поточним рядком;

зі вказівкою NEXT повертається рядок, що знаходиться в повному результівному наборі відразу ж після поточного. Тепер він стає поточним. За замовчуванням команда FETCH використовує саме цей спосіб вибірки рядків;

ключове слово PRIOR повертає рядок, що знаходиться перед поточним; він і стає поточним,

аргумент ABSOLUTE повертає рядок за його абсолютним порядковим номером у повному результівному наборі курсора. Номер рядка можна задати за допомогою константи або як ім'я змінної, в якій зберігається номер рядка. Змінна повинна мати цілочисельний тип даних; указуються як позитивні, так і негативні значення (зі вказівкою позитивного значення рядок відраховується від початку набору, негативного – від кінця). Обраний рядок стає поточним; якщо вказане нульове значення, рядок не повертається;

аргумент RELATIVE повертає рядок, що знаходиться через вказану кількість рядків після поточного. Якщо вказати від'ємне значення числа рядків, то буде повернутий рядок, що знаходиться за вказану кількість рядків перед поточним. За вказівкою нульового значення повернеться поточний рядок;

щоб відкрити глобальний курсор, перед його ім'ям потрібно вказати ключове слово GLOBAL. Ім'я курсора також може бути зазначене за допомогою змінної;

у конструкції INTO задається список змінних, в яких будуть збережені відповідні значення стовпців повернутого рядка. Порядок указівки змінних повинен відповідати порядку стовпців у курсорі, а тип даних змінної – типу даних у стовпці курсора. Якщо конструкція INTO не вказана, то поведінка команди FETCH буде нагадувати поведінку команди SELECT.

Приклад 4.19. Необхідно оголосити та відкрити курсор (рис. 4.26)

```
DECLARE ABC CURSOR SCROLL FOR
SELECT * FROM Товари
OPEN ABC
```

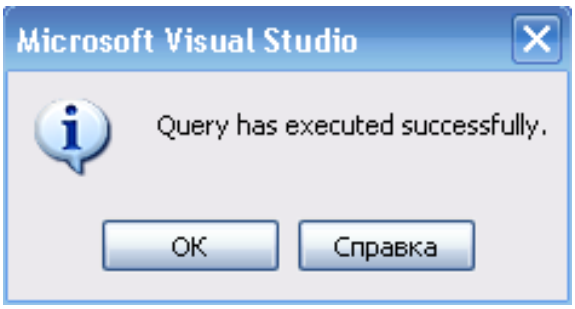


Рис. 4.26. **Результат виконання курсора**

Приклад 4.20. Необхідно використати змінну для оголошення курсора:

```
DECLARE @ Мій_курсор CURSOR
SET @ Мій_курсор =CURSOR LOCAL SCROLL FOR
SELECT * FROM Товари
```

Приклад 4.21. Необхідно використати змінну для присвоювання одного курсора іншому:

```
DECLARE @ Мій_курсор CURSOR
SET @ Мій_курсор = ABC
```

Закрити курсор можна з допомогою такої команди:

```
CLOSE {ім'я_курсора | @ім'я_змінної_курсора}
```

Після закриття курсор стає недоступним для користувачів програми. Шляхом закриття знімаються всі блокування, встановлені в процесі його роботи. Закриття може застосовуватися тільки до відкритого курсора. Закритий, але не звільнений курсор може бути повторно відкритий. Не допускається закривати невідкритий курсор.

Закриття курсора необов'язково звільняє асоційовану з ним пам'ять. У деяких реалізаціях потрібно явно звільнити її за допомогою оператора DEALLOCATE. Після звільнення курсора звільняється і пам'ять, при цьому стає можливим повторне використання імені курсора:

```
DEALLOCATE { ім'я_курсора | @ ім'я_змінної_курсора }
```

Для контролю досягнення кінця курсора рекомендується застосовувати функцію: @@ FETCH_STATUS. Ця функція повертає:

0 – якщо вибірка завершилася успішно;

1 – якщо вибірка завершилася невдало внаслідок вибірки рядка, що знаходиться за межами курсора;

2 – якщо вибірка завершилася невдало внаслідок спроби звернення до видаленого або зміненого рядка.

Приклад 4.22. Необхідно розробити курсор для виведення товару і його ціни, якщо ціна закупівлі товару менше 2 грн:

```
DECLARE  @Товар NVARCHAR(25),
         @Ціна  decimal(5, 2)

DECLARE Мій_курсор CURSOR STATIC FOR

    SELECT Товар, Ціна
       FROM Товари
             WHERE Ціна_закупівлі < 2

OPEN Мій_курсор
FETCH NEXT FROM Мій_курсор
     INTO @Товар, @Ціна
           WHILE @@FETCH_STATUS=0
            BEGIN
                  SELECT @Товар,@Ціна
            END

CLOSE Мій_курсор
DEALLOCATE Мій_курсор
```

Результат виконання курсора зображений на рис. 4.27.

Рис. 4.27. **Курсор виводу даних і результат його виконання**

Приклад 4.23. Необхідно створити курсор для виведення другого запису зі списку записів товарів і цін, якщо ціна закупівлі товару менше 2 грн (рис. 4.28).

```
DECLARE  @Товар   NVARCHAR(25),
      @Ціна    decimal(5, 2)

DECLARE Мій_курсор CURSOR STATIC FOR
   SELECT Товар, Ціна
   FROM Товари
   WHERE Ціна_закупівлі > 2
   ORDER BY Товар
OPEN Мій_курсор
FETCH ABSOLUTE 2 FROM Мій_курсор
 INTO @Товар, @Ціна
WHILE @@FETCH_STATUS=0
BEGIN
   SELECT  @Товар,   @Ціна

 END
CLOSE Мій_курсор
DEALLOCATE Мій_курсор
```

▶	Хліб "Українськ…	3,00

Рис. 4.28. **Курсор виводу даних і результат його виконання**

У результаті читання даних за допомогою курсора вибирається другий запис зі списку, що задовольняє умові "Ціна_закупівлі"> 2 і відсортованих рядків у стовпчику "Товар".

Для виконання змін за допомогою курсора необхідно виконати команду UPDATE в такому форматі:

```
UPDATE ім'я_таблиці SET { ім'я_стовпця={
  DEFAULT | NULL | вираз}}[,...n]
  WHERE CURRENT OF {{[GLOBAL] ім'я_курсора}
  |@ ім'я_змінної_курсора}
```

Приклад 4.24. Необхідно розробити курсор для зміни ціни другого запису зі списку записів товарів і їх цін, якщо ціна закупівлі товару менше 2 грн. Нова ціна товару визначається за формулою: Ціна = Ціна_закупівлі +1.

```
DECLARE  @Товар    NVARCHAR(25),
         @Ціна     decimal(5, 2)
DECLARE Мій_новий_курсор CURSOR
 KEYSET FOR
    SELECT Товар, Ціна
    FROM Товари
    WHERE Ціна_закупівлі > 2
FOR UPDATE

OPEN Мій_новий_курсор
FETCH ABSOLUTE 2 FROM Мій_новий_курсор
      INTO @Товар, @Ціна
 WHILE @@FETCH_STATUS=0
 BEGIN

SELECT @Товар,@Ціна
          UPDATE Товари SET Ціна = Ціна_закупівлі + 1
       WHERE CURRENT OF Мій_новий_курсор
 END
```

Для перевірки результатів виконання курсора скористаємося курсором для виведення змінених даних (рис. 4.29).

Рис. 4.29. **Курсор виводу даних і результат його виконання**

За одну операцію можуть бути змінені декілька стовпців поточного рядка курсора, але всі вони повинні належати одній таблиці.

Для видалення даних за допомогою курсора використовується команда DELETE у такому форматі:

```
DELETE ім'я_таблиці
  WHERE CURRENT OF {{[GLOBAL] ім'я_курсора}
                   |@ім'я_змінної_курсора}
```

У результаті буде видалений рядок, що встановлений в поточному курсорі.

У деяких випадках застосування курсора неминуче. Однак за можливості цього слід уникати та працювати зі стандартними командами обробки даних – SELECT, UPDATE, INSERT, DELETE. Оскільки курсори не дозволяють проводити операції зміни над усім обсягом даних, швидкість виконання операцій обробки даних за допомогою курсора помітно нижча, ніж у стандартних засобів SQL.

4.5. Збережені процедури та тригери

4.5.1. Використання мови T-SQL для створення збережених процедур

Збережені процедури – це набір команд, що складається з одного або декількох операторів SQL або функцій і зберігається в базі даних у відкомпільованому вигляді. Виконання в базі даних збережених процедур замість окремих операторів SQL дає користувачеві такі *переваги*:

необхідні оператори вже містяться в базі даних;

усі вони пройшли етап синтаксичного аналізу і знаходяться в виконуваному форматі;

перед виконанням збереженої процедури SQL Server здійснює її оптимізацію та компіляцію;

збережені процедури підтримують модульне програмування і дозволяють розбивати великі завдання на самостійні, більш дрібні та зручні в управлінні частини;

збережені процедури можуть викликати інші процедури і функції;

збережені процедури можна викликати з прикладних програм інших типів;

як правило, збережені процедури виконуються швидше, ніж послідовність окремих операторів;

збережені процедури можуть складатися з десятків і сотень команд, але для їх запуску достатньо вказати лише ім'я потрібної збереженої процедури. Це дозволяє зменшити розмір запиту, що посилається від клієнта на сервер, а отже, і навантаження на мережу.

Зберігання процедур у тому ж місці, де вони виконуються, забезпечує зменшення обсягу переданих мережею даних і підвищує загальну продуктивність системи. Застосування збережених процедур спрощує супровід програмних комплексів і внесення змін до них. Зазвичай усі обмеження цілісності у вигляді правил і алгоритмів обробки даних реалізуються на сервері баз даних. Вони доступні кінцевому додатку у вигляді набору процедур, які є в інтерфейсі обробки даних. Для забезпечення цілісності даних, а також з метою безпеки додаток зазвичай не отримує прямого доступу до даних; уся робота з ними здійснюється шляхом виклику тих чи інших процедур.

Подібний підхід створює вельми просту модифікацію алгоритмів обробки даних. Вони негайно стають доступними для всіх користувачів мережі, забезпечуючи можливість розширення системи без внесення змін до самого додатку. Для цього досить змінити збережену процедуру на сервері баз даних. Розробнику не потрібно перекомпільовувати додаток, створювати його копії, а також інструктувати користувачів про необхідність роботи з новою версією. Користувачі взагалі можуть не підозрювати про те, що в систему внесені зміни.

Збережені процедури існують незалежно від таблиць або будь-яких інших об'єктів баз даних. Вони викликаються клієнтською програмою, іншою збереженою процедурою або тригером. Розробник може управляти правами доступу до інформації, що зберігається процедурою, дозволяючи або забороняючи її виконання. Змінювати код процедури дозволяється тільки її власнику. За необхідності можна передати права володіння нею від одного користувача до іншого.

У роботі з *SQL Server* користувачі можуть створювати власні процедури, що реалізують ті чи інші дії. Збережені процедури є повноцінними об'єктами бази даних, а тому кожна з них зберігається в конкретній базі даних. Безпосередній виклик збереженої процедури можливий,

тільки якщо він здійснюється в контексті тієї бази даних, де знаходиться процедура.

У *SQL Server* є кілька **типів збережених процедур**:

системні збережені процедури призначені для виконання різних адміністративних дій. Практично всі дії з адміністрування сервера виконуються з їх допомогою. Можна сказати, що системні збережені процедури є інтерфейсом, що забезпечує роботу з системними таблицями. Ця робота зводиться до зміни, додавання, видалення і вибірці даних з системних таблиць як для користувача, так і системних баз даних. Системні збережені процедури мають префікс sp_, зберігаються в системній базі даних і можуть бути викликані в контексті будь-якої іншої бази даних. Крім цього, можна створити свою власну системну збережену процедуру, призначивши їй ім'я з префіксом sp_ і помістивши її в системну базу даних. Така процедура буде доступна в контексті будь-якої бази даних локального сервера;

тимчасові збережені процедури існують лише деякий час, після чого автоматично знищуються сервером. Їх розподіляють на локальні та глобальні. *Локальні тимчасові процедури* можуть бути викликані тільки з того з'єднання, в якому створені. Для створення такої процедури їй необхідно дати ім'я, що починається з одного символу #. Як і всі тимчасові об'єкти, збережені процедури цього типу автоматично видаляються після відключення користувача, перезапуску або зупинки сервера. *Глобальні тимчасові процедури* доступні для будь-яких з'єднань сервера, на якому є така ж процедура. Для її визначення достатньо дати їй ім'я, яке починається з символів ##. Видаляються ці процедури шляхом перезапуску або зупинки сервера, а також закриттям з'єднання, в контексті якого вони були створені.

Створення збереженої процедури здійснюється в такій послідовності:

визначення типу створюваної процедури; тимчасової або призначеної для користувача;

планування прав доступу. У ході створення процедури слід враховувати, що вона буде мати ті ж права доступу до об'єктів бази даних, що і користувач, який її створив;

визначення параметрів процедури. Подібно до процедур, які входять до складу більшості мов програмування, збережені процедури можуть мати вхідні та вихідними параметри;

розроблення коду збереженої процедури. Код процедури може містити послідовність будь-яких команд SQL, включаючи виклик інших процедур.

Створення нової та зміна наявної процедури здійснюється за допомогою такої команди:

```
{CREATE | ALTER } [PROCEDURE] ім'я_процедури
    [;номер]
[{@ім'я_параметра тип_даних } [VARYING ]
     [=default][OUTPUT] ][,...n]
[WITH { RECOMPILE | ENCRYPTION | RECOMPILE,
     ENCRYPTION }]
[FOR REPLICATION]
AS
     sql_оператори [...n]
```

Розглянемо параметри цієї команди. Використовуючи префікси sp_, #, ##, створювану процедуру можна визначити як системну або тимчасову. Як видно з синтаксису команди, не допускається вказувати ім'я власника, якому належатиме створювана процедура, а також ім'я бази даних, де вона повинна бути розміщена. Таким чином, щоб розмістити створювану збережену процедуру в конкретній базі даних, необхідно виконати команду CREATE PROCEDURE в контексті цієї бази даних. Для звернення з тіла збереженої процедури до об'єктів тієї ж бази даних можна використовувати укорочені імена, тобто без указівки імені бази даних. Коли ж потрібно звернутися до об'єктів, розташованих в інших базах даних, указівка імені бази даних обов'язкова.

Номер в імені – це ідентифікаційний номер збереженої процедури, який однозначно визначає її в групі процедур. Для зручності управління процедурами логічно однотипні процедури групувати, привласнюючи їм однакові імена, але різні ідентифікаційні номери.

Для передавання вхідних і вихідних даних у створюваній збереженій процедурі можна використовувати параметри, імена яких (як імена локальних змінних) повинні починатися з символу @. В одній збереженій процедурі можна задати безліч параметрів, розділених комами. У тілі процедури не повинні застосовуватися локальні змінні, чиї імена збігаються з іменами параметрів цієї процедури.

Для визначення типу даних, який буде мати відповідний параметр збереженої процедури, придатні будь-які типи даних SQL, включаючи визначені користувачем. Однак тип даних CURSOR може бути використаний тільки як вихідний параметр збереженої процедури, тобто із зазначенням ключового слова OUTPUT. Наявність *ключового слова OUTPUT* означає, що відповідний параметр призначений для повернення даних зі збереженої процедури. Однак це зовсім не означає, що параметр не доцільний для передавання значень у збережену процедуру. Зазначимо, що зі вказівкою ключового слова OUTPUT значення відповідного параметра для виклику процедури може бути поставлено лише за допомогою локальної змінної. Не дозволяється використання будь-яких виразів або констант, допустимих для звичайних параметрів.

Ключове слово VARYING застосовується спільно з параметром OUTPUT, що має тип CURSOR. Воно визначає, що вихідним параметром буде результівна безліч.

Ключове слово DEFAULT є значенням, яке буде приймати відповідний параметр за замовчуванням. Таким чином, з викликом процедури можна не вказувати явно значення відповідного параметра.

Оскільки сервер кеширує план виконання запиту та компілює код, у наступному виклику процедури будуть використовуватися вже готові значення. Однак у деяких випадках потрібно виконувати перекомпіляцію коду процедури. Указівка ключового слова RECOMPILE наказує системі створювати план виконання процедури з кожним її викликом.

Параметр FOR REPLICATION потрібен для реплікації даних і включення створюваної процедури як статті в публікацію.

Ключове слово ENCRYPTION наказує серверу виконати шифрування коду збереженої процедури. Це може забезпечити захист від використання авторських алгоритмів, що реалізують роботу збереженої процедури.

Ключове слово AS розміщується на початку тіла збереженої процедури, тобто набору команд SQL, за допомогою яких і буде реалізована та чи інша дія. У тілі процедури можуть застосовуватися практично всі команди SQL, оголошуватися транзакції, встановлюватися блокування і викликатися інші збережені процедури.

Вихід з процедури можна здійснити за допомогою команди RETURN.

Видалення збереженої процедури здійснюється командою:

```
DROP PROCEDURE {ім'я_процедури} [,...n]
```

Для виконання збереженої процедури використовується команда:

```
[[ EXECUTE] ім'я_процедури [;номер]
[[@ім'я_параметра=]{значення | @ім'я _змінної}
     [OUTPUT ]|[DEFAULT ]][,...n]
```

Якщо виклик збереженої процедури не є єдиною командою в пакеті, то наявність команди EXECUTE обов'язкова. Більш того, ця команда потрібна для виклику процедури з тіла іншої процедури або тригера.

Використання *ключового слова OUTPUT* до виклику процедури дозволяється тільки для параметрів, які були оголошені під час створення процедури з ключовим словом OUTPUT. Коли ж із викликом процедури для параметра вказується *ключове слово DEFAULT*, то буде використане значення за замовчуванням. DEFAULT дозволяється тільки для тих параметрів, для яких визначено значення за замовчуванням.

З синтаксису команди EXECUTE видно, що імена параметрів можуть бути опущені у виклику процедури. Однак у цьому випадку користувач повинен указувати значення для параметрів у тому ж порядку, в якому вони подавались під час створення процедури.

Присвоїти параметру значення за замовчуванням можна пропустивши його під час перерахування параметрів. Якщо ж потрібно опустити параметри, для яких визначено значення за замовчуванням, досить явної вказівки імен параметрів під час виклику збереженої процедури. Більш того, таким способом можна перераховувати параметри і їх значення в довільному порядку.

Зазначимо, що з викликом процедури вказуються або імена параметрів зі значеннями, або тільки значення без імені параметра. Їх комбінування не допускається.

Приклад 4.25. Необхідно розробити процедуру без параметрів для отримання назв товарів; показати продажі і сумарну вартість продажів, якщо кількість проданих товарів перевищує 800.

```
      CREATE PROC Моя_процедура_1
AS
      SELECT Товари.Товар, Продажі.Дата,
             Товари.Ціна*Продажі.Кількість
      AS Сума
        FROM Товари INNER JOIN Продажі
        ON Товари.Код_товару= Продажі.Код_товару
        WHERE Продажі.Кількість > 800
            RETURN
```

Для звернення до процедури можна використовувати такі команди:

```
EXEC Моя_процедура_1
або
Моя_процедура_1
```

Створимо збережену процедуру за допомогою Microsoft Visual Studio у такій послідовності.

1. Перебуваючи в Visual Studio, у відкритій вкладці Server Explorer лівою кнопкою миші виділимо папку Stored Procedures.

2. Натиснемо правою кнопкою миші на виділеній папці Stored Procedure й отримаємо меню, в якому необхідно вибрати пункт Add New Stored Procedure (рис. 4.30).

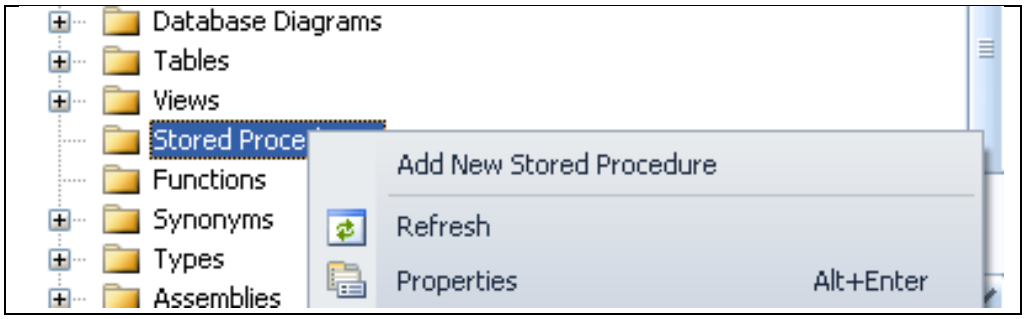


Рис. 4.30. **Вибір пункту меню Add New Stored Procedure**

3. Відкривається вікно, в яке поміщаємо код створюваної процедури (рис. 4.31).

```
CREATE PROC Моя_процедура_1
AS
SELECT Товари.Товар, Продажі.Дата,
       Товари.Ціна*Продажі.Кількість AS Сума
  FROM Товари INNER JOIN Продажі
  ON Товари.Код_товару= Продажі.Код_товару
  WHERE Продажі.Кількість > 800
    RETURN
```

Рис. 4.31. **Вікно для розміщення коду збереженої процедури**

4. Закривши вікно, отримаємо пропозицію зберегти процедуру (рис. 4.32).

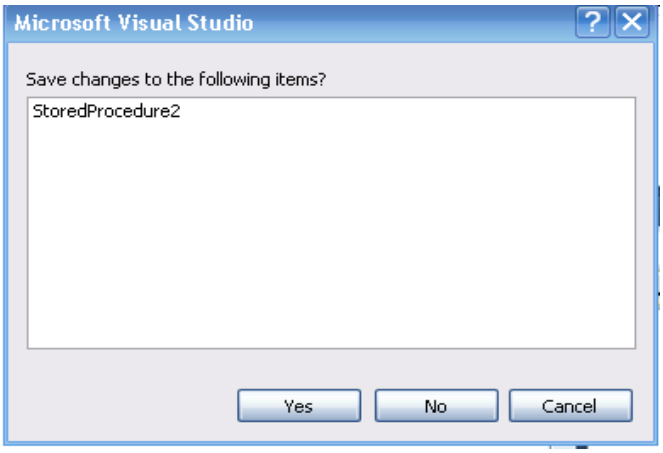


Рис. 4.32. **Вікно для збереження процедури**

5. Після збереження процедури її можна знайти в папці Stored Procedures (рис. 4.33).

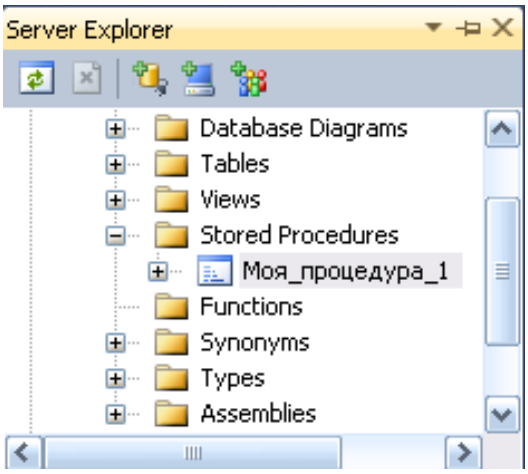


Рис. 4.33. **Місце, де розміщується збережена процедура**

6. Для того щоб виконати збережену процедуру, необхідно натиснути правою кнопкою миші на виділеному імені процедури в папці Stored Procedures і вибрати пункт меню Execute (рис. 4.34).

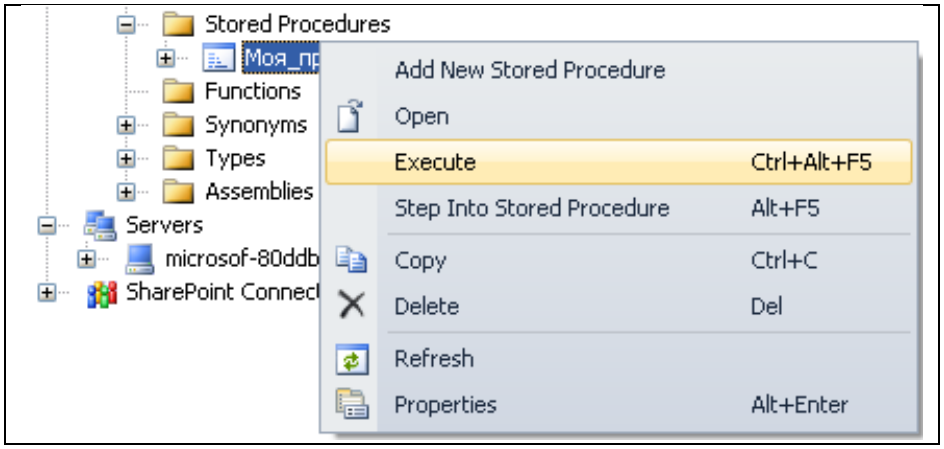


Рис. 4.34. **Вибір пункту меню Execute для виконання збереженої процедури**

Після виконання процедура повертає набір даних (рис. 4.35).

```
Товар                    Дата                          Сума
------------------------ ----------------------------- ---------
Батон "Молочний"         2011-11-11 00:00:00.0000000   3139,50
Хліб "Український"       2012-12-12 00:00:00.0000000   2664,00
Хліб "Український"       2013-12-13 00:00:00.0000000   2826,00
No rows affected.
(3 row(s) returned)
@RETURN_VALUE = 0
Finished running [dbo].[Моя_процедура_1].
```

Рис. 4.35. **Результати виконання збереженої процедури**

Приклад 4.26. Необхідно розробити процедуру для отримання виробника, товару, дати продажу та кількості продажів у день за заданими параметрами коду виробника та коду товару:

```
CREATE PROC Продаж_товарів_виробника
@Cod_tov int,
@Cod_virob int
AS
    Select Товари.Товар, Продажі.Дата, Продажі.Кількість
           From Товари INNER JOIN Продажі
```

```
                ON Товари.Код_товару= Продажі.Код_товару
                      Where (Продажі.Код_товару = @Cod_tov)
                     AND(Продажі.Код_виробника = @Cod_virob)
    RETURN
```

Для звернення до процедури можна використовувати такі команди:

```
  EXEC Продаж_товарів_виробника 3, 2
або
  Продаж_товарів_виробника 3, 2
```

Після виконання процедура повертає набор даних (рис. 4.36).

```
Виробник            Товар                    Дата                          Кількість
------------------- ------------------------ ----------------------------- ----------
Х/з "Кулиничи"      Булка з маком            2012-11-11 00:00:00.0000000   486
Х/з "Кулиничи"      Булка з маком            2014-12-14 00:00:00.0000000   333
No rows affected.
(2 row(s) returned)
@RETURN_VALUE = 0
Finished running [dbo].[Продаж_товарів_виробника].
```

Рис. 4.36. **Результати виконання збереженої процедури**

Приклад 4.27. Необхідно створити процедуру для зменшення ціни заданого товару відповідно до зазначеного відсотка:

```
CREATE PROCEDURE Зниження_цін
  @tovar VARCHAR(25),
  @proc decimal(5, 2)
AS
UPDATE Товари SET Товари.Ціна=Товари.Ціна *(1-@proc/100)
  WHERE Товар=@tovar
  RETURN
```

Для звернення до процедури можна використовувати такі команди:

```
EXEC Зниження_цін 'Батон "Молочний"',0.1
або
EXEC Зниження_цін @ tovar ='Батон "Молочний"', @proc=10
```

Процедура не повертає даних.

Приклад 4.28. Необхідно створити процедуру для отримання інформації про товар і ціну його закупівлі. Процедура повинна мати вхідні параметри зі значеннями за замовчуванням.

```
CREATE PROCEDURE Про_товар
@Назва_товару nvarchar(25)='Булка з маком'
AS
    Select Товар, Ціна_закупівлі
           From Товари
                  Where (Товари.Товар = @Назва_товару)
RETURN
```

Для звернення до процедури можна використовувати команду:

```
EXEC Про_товар
```

У цьому випадку для виклику процедури параметр не вказано і його значення береться за замовчуванням (рис. 4.37).

```
Running [dbo].[Про_товар] ( @Назва_товару = <DEFAULT> ).

Товар                     Ціна_закупівлі
------------------------- -----------------------------------
Булка з маком             1,80
No rows affected.
(1 row(s) returned)
@RETURN_VALUE = 0
Finished running [dbo].[Про_товар].
```

Рис. 4.37. **Результати виконання збереженої процедури**

До тієї ж процедури можна звернутися, задавши значення параметра:

```
EXEC Про_товар 'Батон "Молочний"'
```

У цьому випадку отримаємо дані, які відповідають значенням параметра, що вводиться (рис. 4.38).

```
Running [dbo].[Про_товар] ( @Назва_товару = Батон "Молочний" )

Товар                     Ціна_закупівлі
------------------------- -----------------------------------
Батон "Молочний"          2,50
No rows affected.
(1 row(s) returned)
@RETURN_VALUE = 0
Finished running [dbo].[Про_товар].
```

Рис. 4.38. **Результати виконання збереженої процедури**

Приклад 4.29. Необхідно створити процедуру з вхідними та вихідними параметрами для визначення загальної вартості товарів, що продані за конкретний рік:

```
CREATE PROCEDURE Річні_продажі
  @Рік INT,
  @summa = 0 FLOAT OUTPUT
AS
SELECT @summa=SUM(Товари.Ціна * Продажі.Кількість)
       FROM Товари INNER JOIN
           Продажі ON Товари.Код_товару = Продажі.Код_товару
   GROUP BY YEAR(Продажі.Дата)
   HAVING  (YEAR(Продажі.Дата) = @Рік)
RETURN
```

Для звернення до процедури можна використовувати команди:

```
DECLARE @summa FLOAT =0
EXEC Річні_продажі 2011,@ summa OUTPUT
SELECT @summa
```

Цей блок команд дозволяє визначити сумарну вартість товарів, проданих у 2011 р. Результат виконання процедури відображається в п'ятому рядку повідомлення (рис. 4.39).

```
Running [dbo].[Річні_продажі] ( @Рік = 2011, @summa = 0 ).

No rows affected.
(0 row(s) returned)
@summa = 14994
@RETURN_VALUE = 0
Finished running [dbo].[Річні_продажі].
```

Рис. 4.39. **Результати виконання збереженої процедури**

Приклад 4.30. Необхідно створити процедуру для визначення загальної кількості товарів, проданих виробником, в якому працює заданий співробітник.

Спочатку розробимо процедуру для визначення виробника, де працює співробітник:

```
CREATE PROCEDURE Робітник_виробника
  @робітник VARCHAR(20),
  @вироб INT OUTPUT
AS
SELECT @вироб = Код_виробника
  FROM Виробники
  WHERE Контакт = @робітник
RETURN
```

Тепер створимо процедуру, що підраховує загальну кількість товару, який проданий виробником, що нас цікавить:

```
CREATE PROC Продажі_робітника
  @робітник VARCHAR(20),
  @kol INT OUTPUT
AS
DECLARE @код_вироб INT

EXEC Робітник_виробника @робітник,@код_вироб OUTPUT

SELECT @kol=Sum(Продажі.Кількість)
            FROM Виробники INNER JOIN Продажі
            ON Виробники.Код_виробника = Продажі.Код_виробника
                 WHERE Виробники.Код_виробника = @код_вироб
```

Виклик процедури здійснюється за допомогою команд:

```
DECLARE @kol INT
EXEC Продажі_робітника 'Петров Петр Петрович', @kol OUTPUT
SELECT @kol
```

Результат виконання процедури відображається в п'ятому рядку повідомлення (рис. 4.40).

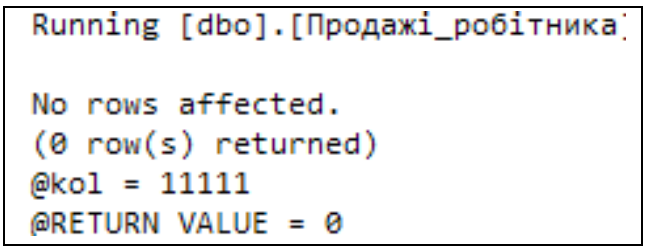


Рис. 4.40. **Результати виконання збереженої процедури**

Збережені процедури дозволяють ефективно виконувати любі складні запити до таблиць бази даних. Вони забезпечують простоту модифікації алгоритмів обробки даних і дають можливість розширення програмного продукту без внесення зміни до самого додатку.

4.5.2. Використання мови T-SQL для створення тригерів бази даних

Тригери є однією з різновидів процедур. Їх ініціалізація відбувається під час виконання будь-якого оператора мови маніпулювання даними (DML). Тригери використовують для перевірки цілісності даних, а також для відкату транзакцій. **Тригер** – це відкомпільована SQL-процедура, виконання якої обумовлене настанням певних подій всередині реляційної бази даних. Застосування тригерів здебільшого вельми зручно для користувачів бази даних. І все ж їх використання часто пов'язане з додатковими витратами ресурсів на операції введення/виведення. У тому випадку, коли тих же результатів (з набагато меншими непродуктивними витратами ресурсів) можна домогтися за допомогою збережених процедур або прикладних програм, застосування тригерів недоцільне.

Тригери – особливий інструмент SQL-сервера, який використовується для підтримки цілісності даних у базі даних. За допомогою обмежень цілісності, правил і значень за замовчуванням не завжди можна домогтися потрібного рівня функціональності. Часто потрібно реалізувати складні алгоритми перевірки даних, що гарантують їх достовірність і реальність. Крім того, іноді необхідно відстежувати зміни значень таблиці, щоб потрібним чином змінити пов'язані дані. Тригери можна розглядати як свого роду фільтри, котрі вступають у дію після виконання всіх операцій відповідно до правил, стандартних значень тощо.

Тригер – це спеціальний тип збережених процедур, що запускаються сервером автоматично зі спробою зміни даних у таблицях, з якими

тригери пов'язані. Кожен тригер прив'язується до конкретної таблиці. Усі вироблені ним модифікації даних розглядаються як одна операція (транзакція). У разі виявлення помилки або порушення цілісності даних відбувається відкат цієї транзакції. Тим самим внесення змін забороняється. Скасовуються також усі зміни, вже зроблені тригером. Тригер створює тільки власник бази даних. Це обмеження дозволяє уникнути випадкової зміни структури таблиць, способів зв'язку з ними інших об'єктів і т. п. Тригер дуже корисний, проте небезпечний засіб. Так, якщо логіка його роботи неправильна, легко знищити цілу базу даних. Тому тригери необхідно дуже ретельно налагоджувати.

На відміну від звичайної підпрограми, тригер виконується неявно в кожному випадку виникнення події, до того ж він не має аргументів. Приведення його в дію іноді називають *запуском тригера*.

За допомогою тригерів досягаються такі **цілі**:

перевірка коректності введених даних;

виконання складних обмежень цілісності даних, які важко (якщо взагалі можливо) підтримувати за допомогою обмежень цілісності, встановлених для таблиці;

видача попереджень, що нагадують про необхідність виконання деяких дій для оновлення таблиці;

накопичення аудиторської інформації за допомогою фіксації відомостей про внесені зміни і тих осіб, які їх виконали;

підтримка реплікації.

Наведемо формат команди створення тригера CREATE TRIGGER:

```
CREATE TRIGGER ім'я_тригера
  BEFORE | AFTER <тригерна_подія>
  ON < ім'я_таблиці>
  [REFERENCING  <список_старих_або_нових_псевдонімів>]
  [FOR EACH { ROW | STATEMENT}]
  [WHEN(умова_тригера)]
      <тіло_тригера>
```

Тригерні події складаються зі вставки, видалення і відновлення рядків у таблиці. В останньому випадку для тригерної події можна вказати конкретні імена стовпців таблиці. Час запуску тригера визначається за допомогою ключових слів BEFORE (тригер запускається до виконання пов'язаних з ним подій) або AFTER (після їх виконання).

Дії, що виконуються тригером, задаються для кожного рядка (FOR EACH ROW), охопленого цією подією, або тільки один раз для кожної події (FOR EACH STATEMENT). Рядок <список_старих_або_нових_псевдонімов> належить до таких компонентів, як старий або новий рядок (OLD/NEW) або стара чи нова таблиця (OLD TABLE/NEW TABLE). Очевидно, що старі значення не застосовні для подій вставки, а нові – для подій видалення. За умови правильного використання тригери можуть стати дуже потужним механізмом. Основна їхня *перевага* полягає в тому, що стандартні функції зберігаються всередині бази даних і узгоджено активізуються з кожним її оновленням. Це може істотно спростити додатки. Проте слід згадати і про властиві тригеру *недоліки*:

при переміщенні деяких функцій в базу даних ускладнюються завдання її проектування, реалізації та адміністрування;

перенесення частини функцій в базу даних і збереження їх у вигляді одного або декількох тригерів іноді призводить до приховування від користувача деяких функціональних можливостей (прихована функціональність). Хоча це дещо спрощує його роботу, але, на жаль, може стати причиною незапланованих, потенційно небажаних і шкідливих побічних ефектів, оскільки в цьому випадку користувач не в змозі контролювати всі процеси, що відбуваються в базі даних;

вплив на продуктивність: перед виконанням кожної команди зі зміни стану бази даних СУБД повинна перевірити тригерні умови з метою з'ясування необхідності запуску тригера для цієї команди. Виконання подібних обчислень позначається на загальній продуктивності СУБД, а в моменти пікового навантаження її зниження може стати особливо помітним. Зі зростанням кількості тригерів збільшуються і накладні витрати, пов'язані з такими операціями.

Неправильно написані тригери можуть призвести до серйозних проблем – таких, наприклад, як поява блокувань. Тригери здатні тривалий час блокувати безліч ресурсів, тому слід звернути особливу увагу на зведення до мінімуму конфліктів доступу.

У реалізації СУБД *MS SQL Server* використовується такий оператор створення або зміни тригера:

```
{CREATE | ALTER} TRIGGER ім'я_тригера
ON {ім'я_таблиці | ім'я_подання }
[WITH ENCRYPTION ]
```

```
{
{ { FOR | AFTER | INSTEAD OF }
{ [ DELETE] [,] [ INSERT] [,] [ UPDATE] }
[ WITH APPEND ]
[ NOT FOR REPLICATION ]
AS
   sql_оператор[...n]
} |
{ {FOR | AFTER | INSTEAD OF } { [INSERT] [,]
  [UPDATE] }
[ WITH APPEND]
[ NOT FOR REPLICATION]
AS
{ IF UPDATE(ім'я_стовпця
[ {AND | OR} UPDATE(ім'я_стовпця)] [...n]
|
IF (COLUMNS_UPDATES(){оператор_біт_обробки}
  біт_маска_змінення)
{оператор_біт_порівняння }біт_маска [...n]}
sql_оператор [...n]
}
}
```

Тригер може бути створений тільки в поточній базі даних, але всередині тригера допускається звернення до інших баз даних, у тому числі розташованим на віддаленому сервері.

Розглянемо призначення аргументів з команди CREATE | ALTER TRIGGER.

Ім'я тригера має бути унікальним у межах бази даних. Додатково можна вказати ім'я власника.

Зі вказівкою аргументу WITH ENCRIPTION сервер виконує шифрування коду тригера, щоб ніхто, включаючи адміністратора, не міг отримати до нього доступ і прочитати його. Шифрування часто використовують для приховування авторських алгоритмів обробки даних, які є інтелектуальною власністю програміста або комерційною таємницею.

У SQL Server існує **два параметри**, що визначають поведінку тригерів:

з використанням параметра AFTER – тригер запускається після успішного виконання виклику його команд. Якщо ж команди з якої-небудь причини не можуть бути успішно завершені, тригер не виконується. Слід

зазначити, що зміни даних у результаті виконання запиту користувача та виконання тригера здійснюються в тілі однієї транзакції: якщо станеться відкат тригера, то будуть відхилені призначені для користувача зміни. Можна визначити кілька AFTER-тригерів для кожної операції (INSERT, UPDATE, DELETE). Якщо для таблиці передбачено виконання кількох AFTER-тригерів, то за допомогою системної збереженої процедури sp_settriggerorder можна вказати, який з них буде виконуватися першим, а який останнім. У SQL Server всі тригери є AFTER-тригерами;

з використанням параметра INSTEAD OF тригер викликається замість виконання команд. На відміну від AFTER-тригера INSTEAD OF-тригер може бути визначений як для таблиці, так і для подання. Для кожної операції INSERT, UPDATE, DELETE можна визначити тільки один INSTEAD OF-тригер.

Тригери розрізняють за типом команд, на які вони реагують. Існує три **типи тригерів**:

INSERT TRIGGER – запускаються спробою вставки даних за допомогою команди INSERT;

UPDATE TRIGGER – запускаються спробою зміни даних за допомогою команди UPDATE;

DELETE TRIGGER – запускаються спробою видалення даних за допомогою команди DELETE.

Конструкції [DELETE] [] [INSERT] [] [UPDATE] і FOR | AFTER | INSTEAD OF} {[INSERT] [] [UPDATE] визначають, на яку команду буде реагувати тригер. Під час його створення треба вказати хоча б одну команду. Допускається створення тригера, що реагує на дві або на всі три команди.

Аргумент WITH APPEND дозволяє створювати кілька тригерів кожного типу.

Зі створеним тригером з аргументом NOT FOR REOLICATION забороняється його запуск під час виконання модифікації таблиць механізмами реплікації.

Конструкція AS sql_оператор [... n] визначає набір операторів SQL і команд, які будуть виконані для запуску тригера.

Зазначимо, що всередині тригера не допускається виконання ряду операцій, таких, наприклад, як:

створення, зміна та видалення бази даних;

відновлення резервної копії бази даних або журналу транзакцій.

Виконання цих команд не дозволено, оскільки їх не можна скасувати у разі відкату транзакції, в якій виконується тригер. Ця заборона навряд чи може якимось чином позначитися на функціональності створюваних тригерів. Важко знайти таку ситуацію, коли, наприклад, після зміни рядка таблиці потрібно виконати відновлення резервної копії журналу транзакцій.

З виконанням команд додавання, зміни та видалення записів сервер створює дві **спеціальні таблиці**: inserted і deleted. В них містяться списки рядків, які будуть вставлені або видалені після закінчення транзакції. Структура таблиць inserted і deleted ідентична структурі таблиць, для якої визначається тригер. Для кожного тригера створюється свій комплект таблиць inserted і deleted, тому інший тригер не зможе отримати до них доступ. Залежно від типу операції, що отримала виконання тригера, *вміст таблиць* inserted і deleted може бути різним:

команда INSERT – у таблиці inserted містяться всі рядки, які користувач намагається вставити в таблицю; в таблиці deleted не буде жодного рядка;

після завершення тригера всі рядки з таблиці inserted перемістяться в вихідну таблицю;

команда DELETE – в таблиці deleted будуть міститися всі рядки, які користувач спробує видалити;

тригер може перевірити кожен рядок і визначити, чи дозволене його видалення;

у таблиці inserted не буде жодного рядка;

команда UPDATE – з її виконанням у таблиці deleted містяться старі значення рядків, які будуть видалені шляхом успішного завершення тригера. Нові значення рядків містяться в таблиці inserted, ці рядки додадуться в вихідну таблицю після успішного виконання тригера.

Для отримання інформації про кількість рядків, яку буде змінено після успішного завершення тригера, можна використовувати функцію @@ROWCOUNT. Вона повертає кількість рядків, оброблених останньою командою. Слід підкреслити, що тригер запускається не зі спробою змінити конкретний рядок, а в момент виконання команди зміни. Одна така команда впливає на безліч рядків, тому тригер повинен обробляти всі ці рядки.

Якщо тригер виявив, що зі ста вставлених, змінених чи видалених рядків хоча б один не задовольняє тій чи іншій умові, то жоден рядок

не буде вставлений, змінений або видалений. Така ситуація обумовлена вимогами транзакції – повинні бути виконані або всі модифікації, або жодної.

Тригер виконується як неявно певна транзакція, тому всередині тригера допускається застосування команд управління транзакціями. Зокрема, для виявлення порушення обмежень цілісності з метою переривання виконання тригера та скасування всіх змін, які намагався виконати користувач, необхідно використовувати команду ROLLBACK TRANSACTION.

Для отримання списку стовпців, змінених проведенням команд INSERT або UPDATE, що викликали виконання тригера, можна використовувати функцію COLUMNS_UPDATED (). Вона повертає двійкове число, кожний біт якого, починаючи з молодшого, відповідає одному стовпцю таблиці (в порядку проходження стовпців під час створення таблиці). Якщо біт встановлено у значення "1", то відповідний стовпець був змінений. Крім того, факт зміни стовпця визначає і функція UPDATE (ім'я_стовпця).

Для видалення тригера використовують таку команду:

```
DROP TRIGGER {ім'я_тригера} [,...n]
```

Наведемо приклади використання тригерів.

Приклад 4.31. Необхідно створити тригер для обробки операції додавання запису в таблицю "Товари_накладних", наприклад, за допомогою такого запиту:

```
INSERT INTO Товари_накладних
           (Код_накладної, Код_товару, Кількість)
            VALUES (3, 1, 100)
```

У запиті в таблицю "Товари_накладних" додається рядок, в якій код накладної дорівнює 3, код товару – 1, кількість товару – 100 одиниць. З додаванням нового запису необхідно перевірити, чи не перевищує кількість товару значення, записані в стовпець "Кількість" у таблиці "Продажі".

```
CREATE TRIGGER Товари_накладних_insert
ON dbo.Товари_накладних

FOR INSERT
AS
IF @@ROWCOUNT=1
--в таблицю Товари_накладних додається
-- запис про товар

BEGIN
--кількість товару в таблиці Товари_накладних
-- не може бути більше,
-- ніж його кількість в таблиці Продажі

IF NOT EXISTS(SELECT *
 FROM inserted
WHERE inserted.Кількість<=ALL(SELECT Продажі.Кількість
            FROM Продажі, inserted
            WHERE Продажі.Код_продажу=
                 inserted.Код_накладної))

    BEGIN
       ROLLBACK TRAN
     END
END
```

Створимо тригер за допомогою *Microsoft Visual Studio* у такому порядку.

1. Перебуваючи в Visual Studio, у відкритій вкладці Server Explorer, а також папці Tables, лівою кнопкою миші виділимо таблицю "Товари_ накладних".

2. Клацнемо правою кнопкою миші на виділеній таблиці "Товари_ накладних" і отримаємо меню, в якому необхідно вибрати пункт Add New Trigger (рис. 4.41).

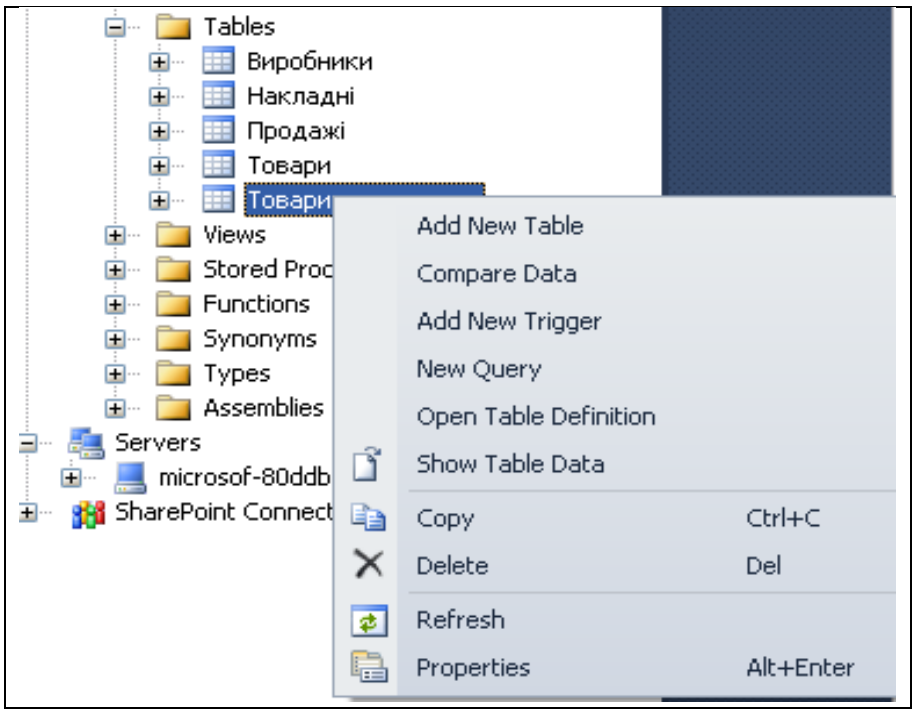


Рис. 4.41. **Вибір пункту меню Add New Trigger**

3. Відкривається вікно, в яке поміщається код створюваного тригера (рис. 4.42).

```
CREATE TRIGGER Товари_накладних_insert
ON dbo.Товари_накладних
FOR INSERT
AS
IF @@ROWCOUNT=1
--в таблицю Товари_накладних додається
-- запис про товар
BEGIN
--кількість товару в таблиці Товари_накладних
-- не може бути більше,
-- ніж його кількість в таблиці Продажі
IF NOT EXISTS(SELECT *
           FROM inserted
           WHERE inserted.Кількість<
  =ALL(SELECT Продажі.Кількість
         FROM Продажі, inserted
         WHERE Продажі.Код_продажу=
               inserted.Код_накладної))
    BEGIN
       ROLLBACK TRAN
     END
```

Рис. 4.42. **Вікно для розміщення коду тригера**

4. Повернемося в вікно Server Explorer і розкриємо таблицю "Товар" у папці Tables, клацнувши лівою кнопкою миші на знак "+" зліва від таблиці. Отримаємо список стовпців таблиці "Товари_накладних", серед яких знаходиться тригер "Товар_накладних_insert" (рис. 4.43).

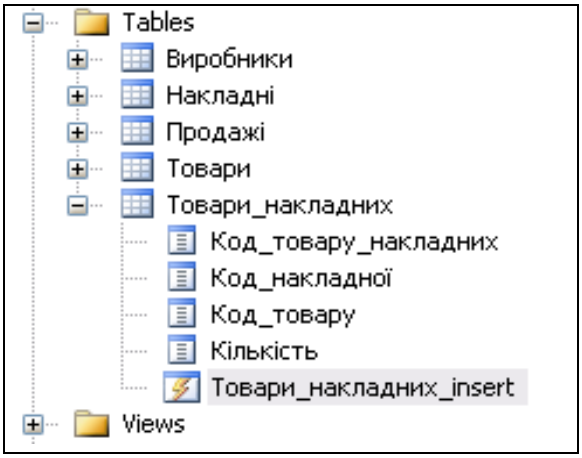


Рис. 4.43. **Розміщення тригера "Товари_накладних_insert" у базі даних**

5. Для того щоб тригер виконував свої функції, досить розпочати додавання даних за допомогою запиту. З виконанням запиту тригер обробляє тільки один рядок, формуючи повідомлення (рис. 4.44).

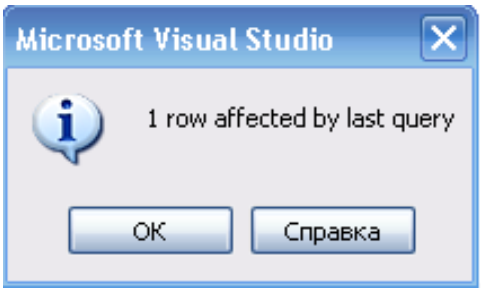


Рис. 4.44. **Результат виконання запиту**

Виконаємо запит з новими даними, в яких кількість доданого товару в таблиці "Товари_накладних" перевищує кількість цього товару в таблиці "Продажі".

```
INSERT INTO Товари_накладних
(Код_накладної, Код_товару, Кількість)
VALUES (3,1,500)
```

З виконанням запиту тригер обробляє тільки один рядок, що додається, і формує повідомлення про переривання транзакції в тригері (рис. 4.45).

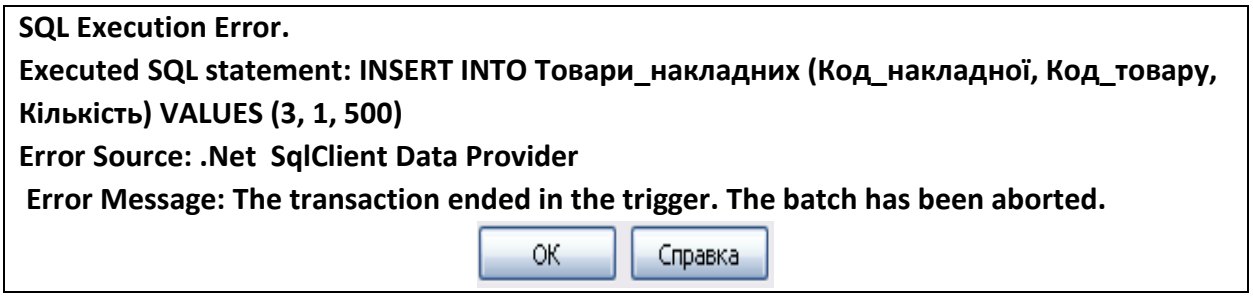


Рис. 4.45. **Результат виконання запиту**

Приклад 4.32. Необхідно створити тригер для обробки операції видалення запису з таблиці "Товари_накладних", наприклад, за допомогою такої команди:

```
DELETE FROM Товари_накладних
WHERE     (Код_товару_накладних = 13)
```

З видаленням рядка таблиці "Товари_накладних", в якій є стовпець, що зберігає кількість товару ("Кількість"), необхідно змінити значення стовпця "Кількість" у таблиці "Продажі".

```
CREATE TRIGGER Товари_накладних_delete
ON Товари_накладних FOR DELETE
AS
IF @@ROWCOUNT=1
-- видалений один запис

BEGIN
  DECLARE @y INT,@x INT
--визначається код накладної та
--кількість товару, який видалений
-- з таблиці Товари_накладних
  SELECT @y=Код_накладної, @x=Кількість
  FROM deleted
--в таблиці Продажі коректується кількість
--товару
     UPDATE Продажі
     SET Кількість= Кількість + @x
     WHERE Продажі.Код_продажу=@y
END
```

Під час виконання запиту тригер обробляє один рядок у таблиці "Товари_накладних" і змінює значення стовпця "Кількість" в одному рядку відповідно до коду накладної. Для відстеження цього процесу формується повідомлення про обробку двох рядків (рис. 4.46).

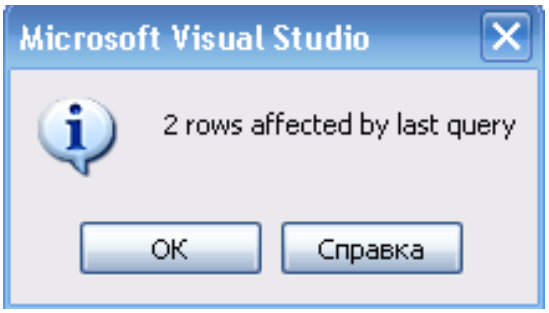


Рис. 4.46. **Результат виконання запиту**

Приклад 4.33. Необхідно створити тригер для обробки операції зміни запису в таблиці "Товари". Наприклад, змінимо значення стовпця "Ціна_закупівлі" за допомогою збереженої процедури:

```
CREATE PROCEDURE Товари_зміна_цін
@Нов_ціна_закупівлі decimal(5, 2),
@Код_тов int
AS
UPDATE    Товари
     SET  Ціна = @Нов_ціна_закупівлі + 1.2,
Ціна_закупівлі=@Нов_ціна_закупівлі
     WHERE   Товари.Код_товару = @Код_тов
RETURN
```

Створимо тригер, який контролює процес зміни полів "Ціна_закупівлі" та "Ціна" одного запису та визначається кодом товару. Нехай зі зміною ціни закупівлі товару тригер повинен пропорційно змінити кількість товарів, що продаються в саму останню дату продажу (в таблиці "Продажі").

У тригері використовуються параметри @x, @y, @z, @k для організації обчислень пропорційно зміни кількості товарів, що продаються, а у підзапиті визначається остання дата продажу заданого в збереженій процедурі товару.

```
CREATE TRIGGER Товари_update
ON Товари FOR UPDATE
AS
 IF @@ROWCOUNT=1 -- оновлений один рядок
BEGIN
DECLARE   @y decimal(5,2), @k int,
@z decimal(5,2),
@x decimal(5,2)

    SELECT @k = deleted.Код_товару,
           @y = deleted.Ціна_закупівлі
         FROM deleted
    SELECT @z = inserted.Ціна_закупівлі
         FROM inserted
        SELECT @x = (@y - @z)/@y

UPDATE  Продажі
SET Продажі.Кількість = Продажі.Кількість * (1 + @x)

     WHERE     Продажі.Дата =
                 (SELECT MAX(Продажі.Дата)
                         FROM Продажі, Товари
     WHERE @k = Продажі. Код_товару)
END
```

З виконанням процедури "Товари_зміна_цін" обробляється один рядок у таблиці "Товари", а тригер змінює значення стовпця "Кількість" в одному рядку, що відповідає коду товару в останню дату продажу. Для простеження цього формується повідомлення про обробку двох рядків (рис. 4.47).

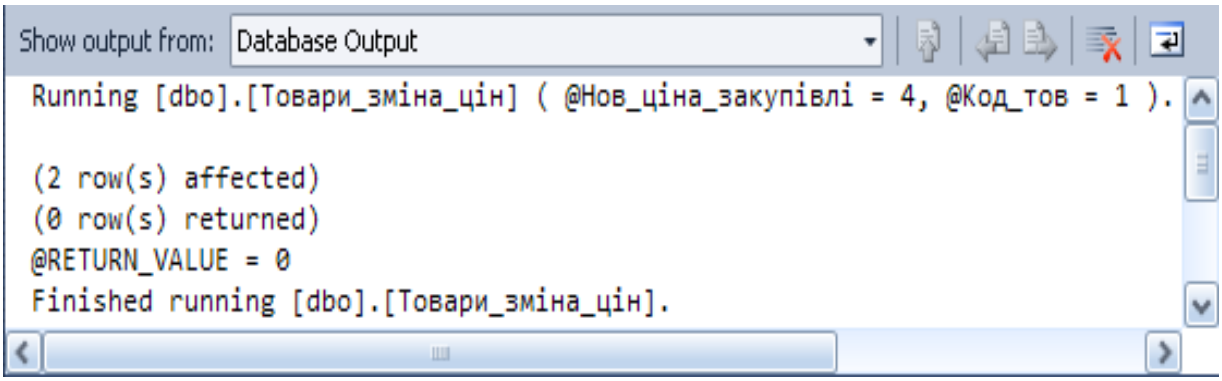


Рис. 4.47. **Результат виконання збереженої процедури та тригера**

Таким чином, тригер запускається сервером автоматично при спробі зміни даних у таблицях. Всі модифікації даних, в яких він "бере участь", розглядаються як одна транзакція. В разі виявлення помилки або порушення цілісності даних, відбувається відкат цієї транзакції.

Запитання для самодіагностики

1. Які види змінних існують у мові T-SQL?
2. Які типи даних існують у мові T-SQL?
3. Наведіть приклади запису дати на мові T-SQL.
4. Які функції перетворення типів даних існують в мові T-SQL?
5. Як мовою T-SQL створити типи даних користувача?
6. Які основні особливості використання системних функцій в T-SQL?
7. Які основні особливості використання функцій користувача у T-SQL?
8. Які основні особливості використання табличних функцій користувача у T-SQL?
9. Які основні особливості використання пакетів у T-SQL?
10. Які основні особливості використання табличних виразів у T-SQL?
11. Які процедурні конструкції використовуються у T-SQL?
12. Які основні особливості обробки помилок у T-SQL?
13. Які основні особливості використання курсорів у T-SQL?
14. Яке призначення збережених процедур?
15. Які основні особливості використання збережених процедур у T-SQL?
16. Які основні особливості використання тригерів у T-SQL?

Лабораторна робота 3
Дослідження особливостей проектування SQL-запитів засобами СУБД SQL Server

Цілі роботи:

1. Набуття практичних навичок побудови збережених процедур і функцій користувача.

2. Створення і використання тригерів.

Перед виконанням роботи студент повинен знати:

1. Основи використання Visual Studio.
2. Основні об'єкти бази даних *MS SQL Server*.
3. Основні команди мови T-SQL.

Після виконання лабораторної роботи студент повинен уміти:

1. Самостійно будувати збережені процедури та функції користувача.
2. Самостійно створювати тригери як об'єкти бази даних.
3. Уміти тестувати збережені процедури та функції користувача, а також тригери.

Хід роботи

1. Збережені процедури.
2. Функції користувача.
3. Тригери.

Форма звітності

За результатами виконання роботи оформити електронний звіт засобами Word.

В електронному звіті за кожним завданням подати текст завдання, скрипт на створення відповідного об'єкта бази даних, скрипт на його тестування та скриншот результату.

Усі скриншоти з виконаними запитами для *SQL Server* повинні мати коментар. Коментар задається рядком з двома початковими мінусами.

Критерії оцінювання

У 12-бальній системі оцінка результату захисту лабораторної роботи формується за такими правилами:

- За кожне завдання з діапазону 1 – 3 може бути виставлено від 0 до 0,7 бала.
- За кожне завдання з діапазону 1.1 – 1.3 із п. "Завдання для самостійного виконання" може бути виставлено від 0 до 0,5 бала, а за аналогічні завдання з індивідуальною базою даних від 0 до 0,7 бала.
- За використання курсора під час виконання завдання – 0,2 бала до кожного завдання.
- За кілька варіантів вирішення одного із завдань додається 0,3 бала.
- За вибір варіанта, який з кількох варіантів розв'язання є оптимальним, та обґрунтування вибору додається 0,5 бала.

- За побудову запитів в іншій СУБД (Oracle, DB2 тощо) додається 1 бал за кожний запит.
- За кожний тиждень запізнення із захистом лабораторної роботи знімається 2 бала.

Отримана кількість балів з відповідей на кожне завдання лабораторної роботи підсумовується. У результаті такого підрахунку студент може отримати від 0 до 12 балів.

Інструкції до виконання

Попередні зауваження

Завдання лабораторної роботи виконуються з базою даних ***Торгівля***, яка містить таблиці ***"Замовники"***, ***"Продавці"*** та ***"Замовлення"*** у середовищі Visual Studio.

Збережені процедури та функції додають у вікні **Server Explorer** до папок **Stored Procedures** і **Functions** відповідної бази даних контекстовими командами **Add New Stored Procedure** та **Add New Scalar-valued Functions**. Тригери додають до таблиць контекстовою командою **Add New Trigger**.

Перед тим як створити збережену процедуру чи функцію користувача, зазвичай попередньо будують скрипт без процедури, щоб відпрацювати алгоритм. Потім готові конструкції перетворюють у збережену процедуру чи функцію користувача.

Наприклад, для обчислення загальної суми замовлень за вказаний день спочатку записують такий скрипт:

```
Declare @Date date,
          @Sum money;
SET LANGUAGE RUSSIAN;
Set @Date='11.10.2012';
SELECT @Sum=Sum(Сума)  From Замовлення
      Where ДатаЗамовлення=@Date;
Select @Sum;
```

Щоб перевірити його правильність, скрипт запускають на виконання. Якщо результат співпадає з попередньо обчисленим, текст перетворюють на збережену процедуру чи функцію користувача.

Збережена процедура має такий вигляд:

```
CREATE PROCEDURE [dbo].[SumOrderOnDate]
      @Date date,
      @Sum money OUTPUT
AS
      SELECT @Sum=Sum(Сумма)
               From Заказ
                        Where ДатаЗамовлення=@Date;
```

Перевіряють описану віще збережену процедуру за допомогою такого скрипта:

```
Declare @S1 money;
SET LANGUAGE RUSSIAN;

EXEC SumOrderOnDate '11.10.2012', @S1 output;
SELECT @S1;
```

Примітки:

1. виклик збереженої процедури виконують оператором EXEC (скорочена форма), або EXECUTE (повна форма);

2. під час викликання збереженої процедури після вихідного параметра записують службове слово output;

3. для збереження значення вихідного параметра описують змінну (у нашому випадку @S1).

Функцію користувача можна викликати як частину виразу в запиті.

Наприклад, для знаходження мінімального рейтингу замовника в указаному місті, можна використати функцію користувача, яка описується таким скриптом:

```
CREATE FUNCTION [dbo].[MinRatingCity]
(
      @City nchar(20)
)
RETURNS smallint
AS
BEGIN
      DECLARE @Rating smallint;

      SELECT @Rating=MIN(Рейтинг)
            FROM Замовники
            WHERE Місто=@City;
      RETURN @Rating
END
```

Перевірку розглядуваної функції користувача можна здійснити таким скриптом:

```
SELECT dbo.MinRatingCity(N'Київ');
```

Робота тригера перевіряється під час виконання завдання, для обробки якого він призначений.

Якщо для виконання завдань будуть потрібні додаткові таблиці або нові поля в існуючих таблицях, їх слід створити. Дані слід підібрати так, щоб у результаті виконання запиту, який може містити кілька рядків, їх було не менше двох.

Завдання виконуються за варіантами відповідно до табл. 1.

Таблиця 1

Варіанти завдань

Варіант	Завдання		
	1	2	3
1	2	3	4
1	1;4;7;10;13	1;3;6;10;12;15	1;3;7;10;14
2	2;5;8;10;14	2;4;7;11;13;16	2;4;8;11;15
3	3;6;9;11;13	1;5;8;10;14;17	1;5;9;12;14
4	1;4;8;11;14	2;3;9;11;12;18	2;4;6;13;15
5	2;5;7;12;13	1;4;6;10;13;17	1;3;7;12;15
6	3;6;8;12;14	2;5;7;11;14;16	2;4;8;11;14
7	1;5;9;10;13	1;4;8;10;12;15	1;5;9;10;15
8	2;6;9;11;14	2;5;9;11;13;18	2;5;6;10;14
9	3;4;8;12;13	1;3;6;10;14;17	1;4;7;11;14
10	1;5;7;12;14	2;4;7;11;13;16	2;3;8;11;15
11	2;6;7;11;13	1;5;8;10;12;15	1;3;9;12;15
12	3;4;9;10;14	2;3;9;11;14;16	2;4;8;12;14
13	1;6;8;11;13	1;5;8;10;13;17	2;5;7;13;14
14	2;4;7;12;14	2;3;7;11;12;18	1;4;6;13;15
15	3;5;8;10;13	1;4;6;10;13;15	2;3;7;10;15
16	1;6;9;11;14	2;5;9;11;14;16	1;3;9;11;14
17	2;4;7;12;13	1;3;8;10;12;17	2;4;8;12;14
18	3;5;7;11;14	2;4;7;11;13;18	1;4;6;13;15
19	1;4;8;10;13	1;4;6;10;13;17	2;5;9;11;15
20	2;5;8;10;14	2;5;9;11;12;16	1;5;8;12;15

Закінчення табл. 1

1	2	3	4
21	3;6;9;11;13	1;3;8;10;14;15	2;4;7;11;14
22	1;5;9;12;14	2;4;7;11;13;16	1;3;6;10;14
23	2;6;7;12;13	1;5;6;10;12;17	2;5;8;10;15
24	3;4;8;11;14	2;3;7;11;14;18	1;4;7;11;14
25	1;6;9;10;13	1;4;8;10;13;15	2;5;9;11;15
26	2;4;8;11;14	2;5;9;11;12;16	1;3;6;12;14
27	3;5;7;12;13	1;4;8;10;14;17	2;4;8;12;14
28	1;5;9;11;14	2;3;7;11;12;18	1;3;9;13;15
29	2;4;8;10;13	1;4;6;10;13;17	2;5;7;13;15
30	3;6;7;12;14	2;5;9;11;14;16	1;4;6;10;14

Завдання 1. Збережені процедури

Побудувати та перевірити збережені процедури для виконання таких операцій:

1. Вивести номери замовлень, суму та дату для всіх рядків з таблиці "Замовлення" вказаного замовника.

2. Вивести всі рядки з таблиці "Замовлення", у яких брав участь вказаний продавець.

3. Вивести таблицю зі стовпцями "Прізвище_продавця" і "Комісійні", що працюють у вказаному місті.

4. Вивести рейтинг і прізвище кожного замовника із вказаного міста.

5. Вивести всі коди продавців з таблиці "Замовлення" без повторень за вказаний проміжок часу.

6. Вивести прізвища продавців із вказаного міста, в яких комісійні перевищують 0.1.

7. Вивести список усіх замовників з рейтингом не вище вказаного, крім тих, які живуть в зазначеному місті.

8. Вивести всі дані про замовлення з сумою, що не перевищує вказану, за зазначений період.

9. Вивести всі пари продавців, які живуть в зазначеному місті за вказаний період. Виключити комбінації продавців з самими собою, а також дублікати рядків, що виводяться в зворотному порядку.

10. Вивести прізвища та міста всіх замовників, що зробили замовлення у зазначений період, з таким самим рейтингом, як у вказаного.

11. Визначити загальну суму замовлень на вказану дату у зазначеному місті.

12. Підрахувати кількість замовлень, що зроблені у зазначеному місті на вказаний діапазон сум.

13. Вивести прізвища замовників, які починаються із заданої букви та які зробили замовлення у зазначеному місті.

14. Обчислити загальну суму комісійних, отриманих продавцями на замовлення у вказаний період, що зроблені замовниками, рейтинг яких перевищує зазначену величину.

Завдання 2. Функції користувача

Побудувати та перевірити функції користувача для виконання таких операцій:

1. Визначити загальну суму комісійних для вказаного продавця.

2. Визначити максимальний рейтинг замовників для вказаного міста.

3. Визначити кількість замовників для вказаного міста.

4. Визначити максимальні комісійні для вказаного міста.

5. Визначити найдовше прізвище замовника, яке починається на вказану букву.

6. Визначити мінімальну суму замовлення для вказаного замовника.

7. Визначити прізвище продавця, який оформив замовлення у вказаний день на найбільшу суму.

8. Визначити кількість замовників, які обслуговувалися вказаним продавцем.

9. Визначити кількість замовників, чиї прізвища починаються з літери в діапазоні вказаних літер від Літера1 до Літера2.

10. Визначити кількість замовлень, що мають нульові або невизначені значення у полі **Сума**.

11. Визначити кількість продавців, які обслуговують замовників із вказаним рейтингом.

12. Визначити кількість замовників із вказаним рейтингом, яких обслуговує продавець із зазначеного міста.

13. Визначити кількість замовників, рейтинг яких вищий за рейтинг будь-якого замовника із вказаного міста.

14. Знайти кількість продавців, які в своєму місті не мають жодного замовника.

15. Визначити кількість замовлень з сумою, що перевищує будь-яку для замовників із вказаного міста.

16. Визначити місце зазначеного продавця у списку, де продавці розташовані за спаданням розміру комісійних.

17. Визначити прізвища продавця, який оформив замовлення на найбільшу суму для зазначеного замовника.

18. Визначити місце у рейтингу зазначеного замовника для вказаного міста.

Завдання 3. Тригери

Створити та перевірити тригери для виконання таких операцій:

1. З додаванням нового замовлення збільшити на 1 % рейтинг замовника. Він не може бути більше 100.

2. З вилученням замовлення зменшити на 3 % рейтинг замовника. Він не може бути менше нуля.

3. З додаванням нового замовлення зменшити на 2 % комісійні продавця. Він не може бути менше нуля.

4. З додаванням нового замовника збільшити на 1 % кількість замовників у його місті (таблиця "Міста").

5. З додаванням нового продавця збільшити на 1 % кількість продавців у його місті (таблиця "Міста").

6. З вилученням замовника зменшити на 1 % кількість замовників у його місті (таблиця "Міста").

7. З вилученням продавця зменшити на 1 % кількість продавців у його місті (таблиця "Міста").

8. З додаванням нового замовлення збільшити на 1 % кількість замовників, яких обслуговує даний продавець, якщо раніше він їх не обслуговував.

9. З додаванням нового замовлення збільшити загальну суму вартостей замовлень, які оформляв продавець.

10. З вилученням замовлення зменшити у подвійному розмірі загальну суму вартості замовлень, які оформляв продавець. Вона не може бути менше нуля.

11. З вилученням замовлення зменшити загальну суму вартості замовлень, які здійснював замовник. Вона не може бути менше нуля.

12. З вилученням продавця з таблиці "Продавці", передати всі його замовлення продавцю з того самого міста, у якого найменше замовників.

13. З зміною суми замовлення у таблицю "Корегування" занести всі дані про замовлення із зазначенням початкової та кінцевої сум.

14. З зміною міста замовника зменшити кількість замовників у попередньому місті, а також збільшити кількість замовників у новому (таблиця "Міста").

15. З додаванням нового замовлення збільшити загальну суму вартості замовлень за поточний день, які оформляв продавець (таблиця "ПродажіПродавців").

Завдання для самостійного виконання

1. Для бази даних ***Хліб*** самостійно сформулювати такі завдання та реалізувати їх мовою SQL:

1.1. З використанням збережених процедур.

1.2. З використанням функцій користувача.

1.3. З використанням тригерів.

2. Повторити п. 1 для індивідуальної бази даних.

Примітка. У звіті за кожним завданням подайте текст завдання, скрипт на його створення, запит на виконання та скриншот результату.

Висновки

Мова Transact-SQL є ключем до використання *MS SQL Server*. Усі додатки, які взаємодіють з екземпляром *MS SQL Server*, незалежно від їх реалізації і призначеного для користувача інтерфейсу, відправляють серверу інструкції Transact-SQL.

Аналогічно стандартному SQL, у T-SQL використовують ідентифікатори, змінні, операції і оператори. *SQL Server підтримує два види змінних: локальні та глобальні.*

Transact-SQL дозволяє використовувати тимчасові таблиці, які створюються в пам'яті сервера на час роботи користувача з базою даних.

У T-SQL можливе створення типів даних користувача. Для цього SQL Server створює в поточній базі даних псевдонім нового типу даних

або визначений користувачем тип. Реалізація отриманого псевдоніма ґрунтується на власному системному типі SQL Server.

Для реалізації на мові SQL складних алгоритмів, які можуть знадобитися більше одного разу необхідно збереження розробленого коду для подальшого застосування. SQL Server підтримує створення обумовлених користувачем функцій. Визначені користувачем функції можуть бути або скалярними, або табличними.

У Transact-SQL існують спеціальні команди, які дозволяють управляти потоком виконання сценарію, перериваючи його або направляючи в потрібну логіку.

Під час виконання запитів виникають помилки, що потребують виявлення та обробки. Мова Transact-SQL має дуже широкі можливості для контролю і визначення помилок у створенні запитів.

SQL Server підтримує три види курсорів:

курсори SQL – застосовуються в основному всередині тригерів, збережених процедур і сценаріїв;

курсори сервера, які діють на сервері;

курсори клієнта – реалізуються для самого клієнта. Вони вибирають весь результівний набір рядків з сервера та зберігають його локально.

Під час роботи з *SQL Server* користувачі можуть створювати власні процедури. *У SQL Server є кілька типів збережених процедур:*

системні збережені процедури, які призначені для виконання різних адміністративних дій;

тимчасові збережені процедури, які існують лише деякий час, після чого автоматично знищуються сервером; їх розподіляють на локальні та глобальні;

стаціонарні збережені процедури користувача.

Тригер – це спеціальний тип збережених процедур, що запускаються сервером автоматично зі спробою зміни даних у таблицях, з якими тригери пов'язані. Кожен тригер прив'язується до конкретної таблиці. Усі вироблені ним модифікації даних розглядаються як одна операція (транзакція).

Рекомендована література

1. Андон Ф. Язык запросов SQL: учебный курс / Ф. Андон, В. Резниченко. – Санкт-Петербург : Питер; Киев : BHV, 2006. – 416 с.
2. Астахова И. Ф. SQL в примерах и задачах / И. Ф. Астахова, А. П. Толстобров, И. М. Мельников. – Минск: Новое знание, 2002. – 176 с.
3. Боуман Д. Практическое руководство по SQL / Д. Боуман, С. Эмерсон, М. Дарновски. – Москва : Вильямс, 2002. – 352 с.
4. Бураков П. В. Введение в системы баз данных : учеб. пособ. / П. В. Бураков, В. Ю. Петров. – Санкт-Петербург : ГУИТМО, 2010. – 129 с.
5. Виейра Р. Программирование баз данных. *Microsoft SQL Server 2005.* Базовый курс / Р. Виейра. – Москва : Вильямс, 2007. – 832 с.
6. Вишневский А. В. *Microsoft SQL Server*. Эффективная работа / А. В. Вишневский. – Санкт-Петербург : Питер, 2009, – 541 с.
7. Галузевий стандарт вищої освіти України з напряму підготовки 6.050101 "Комп'ютерні науки" // Збірник нормативних документів вищої освіти. – Київ : Видавнича група BHV, 2011. – 85 с.
8. Грофф Д. SQL: Полное руководство / Д. Грофф, П. Вайнберг. – Київ : BHV, 2001. – 816 с.
9. Дейт Дж. Введение в системы баз данных / Дж. Дейт. – 8-е изд. – Москва : Вильямс, 2005. – 1328 с.
10. Дунаев В. В. Базы данных. Язык SQL / В. В. Дунаев. – Санкт-Петербург : БХВ-Петербург, 2006. – 288 с.
11. Казакова И. А. Основы языка Transact SQL / И. А. Казакова. – Пенза : Изд. ПГУ, 2010. – 164 с.
12. Карпова Т. С. Базы данных. Модели, разработка, реализация : учебник / Т. С Карпова. – Санкт-Петербург : Питер, 2001. – 302 с.
13. Клайн К. SQL: справочник / К. Клайн. – Москва : Кудиц-образ, 2006. – 832 с.
14. Когаловский М. Р. Энциклопедия технологий баз данных (Эволюция технологий. Технологии и стандарты. Инфраструктура. Терминология) / М. Р. Когаловский. – Москва : Финансы и статистика. 2002. – 836 с.
15. Конноли Т. Базы данных: проектирование, реализация и сопровождение : учеб. пособ. / Т. Конноли. – Москва : Вильямс, 2000. – 1120 с.
16. Нильсен П. Microsoft SQL Server 2005. Библия пользователя / П. Нильсен. – Москва : Вильямс, 2008. – 1232 с.

17. Пушников А. Ю. Введение в системы управления базами данных. Реляционная модель данных : учеб. пособ. / А. Ю. Пушников. – Уфа : Изд. Башкирского ун-та, 1999. – 108 с.

18. Тарасов О. В. Використання мови SQL для роботи з сучасними системами керування базами даних / О. В. Тарасов, В. В. Федько, М. Ю. Лосєв. – Харків : Вид. ХНЕУ, 2013. – 348 с.

19. Тарасов О. В. Проектування баз даних : навч. посіб. / О. В. Тарасов, В. В. Федько, М. Ю. Лосєв. – Харків : Вид. ХНЕУ, 2011. – 200 с.

20. Федько В. В. Лабораторний практикум з модуля "Основи баз даних та знань" навчальної дисципліни "Організація баз даних та знань" / В. В. Федько, О. В. Тарасов, М. Ю. Лосев. – Харків : Вид. ХНЕУ, 2011. – 192 с.

21. Федько В. В. Організація баз даних та знань : навч.-прак. посіб. / В. В. Федько, О. В. Тарасов, М. Ю. Лосєв. – Харків : Вид. ХНЕУ, 2013. – 200 с.

22. Фленов М. Е. Transact-SQL / М. Е. Фленов. – Санкт-Петербург : БХВ-Петербург, 2006. – 576 с.

23. Форта Б. Освой самостоятельно SQL / Б. Форта. – Москва : Вильямс, 2005. – 288 с.

24. Харитонова И. Программирование в Access 2002 : учеб. курс / И. Харитонова, Н. Вольман. – Санкт-Петербург : Питер, 2003. – 480 с.

25. Хендерсон К. Профессиональное руководство по Transact-SQL / К. Хендерсон. – Санкт-Петербург : Питер, 2005. – 558 с.

26. Хернандес М. Д. SQL запросы для простых смертных. Практическое руководство по манипулированию данными в SQL / М. Д. Хернандес, Д. Л. Вьескас. – Москва : Лори, 2003. – 460 с.

27. Codd E. F. (1985a). Is your DBMS really relational? / E. F. Codd // Computerworld. – 14 October. – 1985. – p.1–9.

28. Codd E. F. (1985b). Does your DBMS run by the rules? / E. F. Codd // Computerworld. – 21 October. – 1985. – p. 49–64.

29. Автоинкрементные первичные ключи (суррогатные ключи) = зло? [Электронный ресурс]. – Режим доступа : http://habrahabr.ru/post/107834.

30. Деревянко А. С. Язык SQL в диалектах Oracle и IBM DB2 [Электронный ресурс] / А. С. Деревянко. – Режим доступа : http://khpi-iip.mipk.kharkiv.edu/library/extent/dbms/sql/index.html.

31. Карпова Т. С. Базы данных: модели, разработка, реализация [Электронный ресурс] / Т. С. Карпова. – Режим доступа : http://www.intuit.ru/department/database/dbmdi.

32. Кузнецов С. Д. Стандарты языка реляционных баз данных SQL : краткий обзор [Электронный ресурс] / С. Д. Кузнецов. – Режим доступа : http://citforum.ru/database/articles/art_2.shtml.

33. Скачко В. В. Лекции по дисциплине "Теория баз данных" [Электронный ресурс] / В. В. Скачко. – Режим доступа : http://its.lnpu.edu.ua/~Skachko.

34. Соответствие стандартам SQL [Электронный ресурс]. – Режим доступа : http://postgresql.ru.net/manual/features.html.

35. Сопоставление типов данных для издателей Oracle SQL [Электронный ресурс]. – Режим доступа : http://msdn.microsoft.com/ru-ru/library/ms151817.aspx.

36. Суррогатный_ключ [Электронный ресурс]. – Режим доступа : http://ru.wikipedia.org/wiki/ Суррогатный_ключ.

37. Федоров А. Введение в базы данных. Введение в язык SQL [Электронный ресурс] / А. Федоров, Н. Елманова. – Режим доступа: http://www.compress.ru/article.aspx?id=11944&iid=463#06/.

38. Эквивалентные типы данных ANSI SQL [Электронный ресурс]. – Режим доступа : http://office.microsoft.com/ru-ru/access-help/HP001032 229.aspx.

39. IDENTITY (свойство) (Transact-SQL) [Электронный ресурс]. – Режим доступа : http://msdn.microsoft.com/ru-ru/library/ms186775.aspx.

Зміст

НАВЧАЛЬНЕ ВИДАННЯ

Лосєв Михайло Юрійович
Федько Віктор Васильович

БАЗИ ДАНИХ

Навчально-практичний посібник для самостійної роботи студентів

Самостійне електронне текстове мережеве видання

Відповідальний за видання *О. Г. Руденко*

Відповідальний редактор *М. М. Оленич*

Редактор *Н. І. Ганцевич*

Коректор *Т. А. Маркова*

План 2018 р. Поз. № 23-ЕНП. Обсяг 233 с.

Видавець і виготовлювач – ХНЕУ ім. С. Кузнеця, 61166, м. Харків, просп. Науки, 9-А

Свідоцтво про внесення суб'єкта видавничої справи до Державного реєстру
ДК № 4853 від 20.02.2015 р.