МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ ЕКОНОМІЧНИЙ УНІВЕРСИТЕТ ІМЕНІ СЕМЕНА КУЗНЕЦЯ

ПРОЄКТУВАННЯ БАЗ ДАНИХ ТА БАЗ ЗНАНЬ

Методичні рекомендації до лабораторних робіт для студентів спеціальності 186 "Видавництво та поліграфія" першого (бакалаврського) рівня

Харків
ХНЕУ ім. С. Кузнеця
2022

УДК 004.65(07.034)
П79

Укладач А. С. Гордєєв

Затверджено на засіданні кафедри комп'ютерних систем і технологій.
Протокол № 5 від 15.11.2021 р.

Самостійне електронне текстове мережеве видання

Проєктування баз даних та баз знань [Електронний ресурс] :
П79 методичні рекомендації до лабораторних робіт для студентів спеціальності 186 "Видавництво та поліграфія" першого (бакалаврського) рівня / уклад. А. С. Гордєєв. – Харків : ХНЕУ ім. С. Кузнеця, 2022. – 73 с.

Подано основні положення щодо організації та виконання самостійної роботи. Уміщено загальні положення щодо виконання самостійної роботи з навчальної дисципліни та програму виконання самостійної роботи. Наведено детальний опис завдань для самостійної роботи та перелік необхідної для виконання завдань літератури.

Рекомендовано для студентів спеціальності 186 "Видавництво та поліграфія" першого (бакалаврського) рівня.

УДК 004.65(07.034)

Вступ

Навчальна дисципліна "Проєктування баз даних та баз знань" належить до групи обов'язкових навчальних дисциплін циклу професійно-орієнтованих дисциплін та вивчається згідно з навчальним планом підготовки за спеціальністю 186 "Видавництво та поліграфія".

Мета навчальної дисципліни: формування у студентів фундаментальних теоретичних знань з архітектури баз даних та баз знань, принципів їх організації та особливостей використання; здобуття навичок проєктування технологічних систем, що базуються на використанні баз даних та баз знань.

Завданням навчальної дисципліни є вивчення теоретичних основ і технологій створення баз даних, особливостей роботи над спеціальними інформаційними джерелами та інструментальними засобами, що дозволяють більш ефективно застосовувати сучасні автоматизовані системи управління базами даних.

Об'єктом вивчення навчальної дисципліни є технології роботи з базами даних на платформі .NET Framework.

Предметом вивчення навчальної дисципліни бази даних та знань у інформаційні системи стадій розроблення та експлуатації електронних мультимедійних видань.

Інструментальною базою вивчення дисципліни є сучасне програмне забезпечення для створення і управління базами даних *Microsoft SQL Server Management Studio 2018* та *Visual Studio 2019.*

Необхідність здобуття розгорнутих знань із навчальної дисципліни "Проєктування баз даних та баз знань" для успішного виконання подальшої професійної діяльності й обмеженість навчального (зокрема аудиторного) навантаження студентів спеціальності 186 "Видавництво та поліграфія" зумовлює характер лабораторних робіт студентів у межах цієї навчальної дисципліни. Виконання лабораторних завдань має забезпечувати набуття студентами нових компетентностей у межах тієї тематики, що розглядається на лекційних заняттях.

Отже, основною метою лабораторних робіт з навчальної дисципліни "Проєктування баз даних та баз знань" є надання можливості закріплення студентами спеціальності 186 "Видавництво та поліграфія" таких компетентностей.

- знання та розуміння предметної області та розуміння професійної діяльності;
- здатність застосовувати знання у практичних ситуаціях;
- здатність приймати обґрунтовані рішення;
- здатність спілкуватися з представниками інших професійних груп різного рівня (з експертами з інших галузей знань/видів економічної діяльності);
- здатність здійснення безпечної діяльності;
- здатність зберігати та примножувати моральні, культурні, наукові цінності і досягнення суспільства на основі розуміння історії та закономірностей розвитку предметної області, її місця у загальній системі знань про природу і суспільство та у розвитку суспільства, техніки і технологій, використовувати різні види та форми рухової активності для активного відпочинку та ведення здорового способу життя.

Лабораторна робота 1
Створення і управління базою даних у СУБД Microsoft SQL Server

Мета роботи: створити бази даних і таблиці в середовищі *Visual Studio* з використанням мови структурованих запитів SQL.

Завдання

1. Створити базу даних за допомогою SQL-запитів.
2. Заповнити таблиці комп'ютерної фірми.
3. Виконати завдання.

Зміст звіту

Найменування і мета роботи.

База даних комп'ютерної фірми.

Порядок виконання

1. Створення бази даних за допомогою SQL-запитів.

Необхідно створити базу даних, яку назвати *comp_firm*. Для цього в SQL існує оператор *create database*. Створення бази даних має такий синтаксис:

```
CREATE DATABASE НАЗВА_БАЗИ_ДАНИХ;
```

Максимальна довжина назви БД складає 64 знаки і може містити букви, цифри, символ "_" і символ "$". Назва може починатися з цифри, але не повинна повністю складатися з цифр. Будь-який запит до БД закінчують крапкою з комою (цей символ називається роздільником – *delimiter*). Отримавши запит, сервер виконує його і в разі успіху видає повідомлення "Query OK ...".

Тепер у цій базі даних треба створити 4 таблиці (у дужках вказані найменування стовпців таблиці):

```
PRODUCT (maker, model, type);
PC (code, model, speed, ram, hd, cd, price);
```

LAPTOP (code, model, speed, ram, hd, prise, screen);

PRINTER (code, model, color, type, price).

Таблиця *Product* представляє виробника (*maker*), номер моделі (*model*), і тип (*type*) ('*PC*' – ПК, *'Laptop'* – ноутбук, *'Printer'* – принтер). Передбачається, що номери моделей у таблиці *Product* унікальні для всіх виробників і типів продуктів.

У таблиці *PC* для кожного ПК, однозначно визначається унікальним кодом (*code*), вказана модель (*model*), швидкість процесора в мегагерцах (*speed*), обсяг пам'яті в мегабайтах (*ram*), розмір диска в гігабайтах (*hd*), швидкість пристрою, що зчитує (*cd*) і ціна (*price*).

Таблиця *Laptop* аналогічна таблиці PC за винятком того, що замість швидкості cd містить розмір екрана в дюймах (*screen*).

У таблиці *Printer* для кожної моделі вказується, чи є він кольоровим – *color* ('y', якщо кольоровий), тип принтера – *type* (лазерний – *Laser*, струменевий – *Jet* або матричний – *Matrix*) і ціна – *price*.

Для створення таблиць в SQL існує оператор *create table*. Створення таблиць має такий синтаксис:

```
CREATE TABLE назва_таблиці
(назва_першого_стовпчика тип,
назва_другого_стовпчика тип,
...,
назва_останнього_стовпчика тип);
```

Вимоги до назв таблиць і стовпців такі ж, як і для назв БД. До кожної колонки прив'язаний певний тип даних, який обмежує характер інформації, яку можна зберігати в стовпці (наприклад, запобігати введення букв у числове поле). SQL підтримує кілька типів даних: числові, строкові, календарні і спеціальний тип NULL, що означає відсутність інформації.

Типи даних та схема для нашої бази наведені на рис. 1.1.

Не варто забувати вказувати первинні і зовнішні ключі.

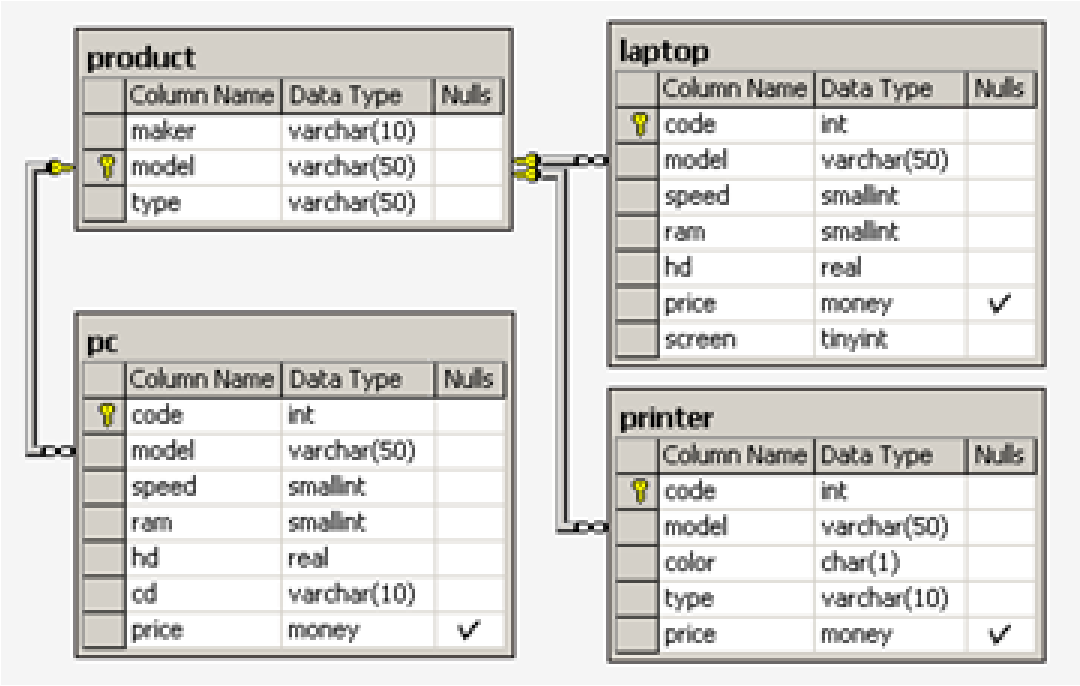


Рис. 1.1. **Типи даних та схема для бази *comp_firm***

2. Внесення даних у таблиці.

Після створення таблиць необхідно ввести дані в наші таблиці. Для цього використовується оператор *INSERT*:

```
INSERT назва_таблиці
VALUES
('значення_першого_стовпчика',
'значення_другого_стовпчика',
...,
'значення_останнього_стовпчика');
```

Щоб додати відразу кілька рядків, треба просто перераховувати дужки зі значеннями через кому:

Слід заповнити таблицю *Product* даними, що подані на рис. 1.2.

maker	model	type
A	1232	PC
A	1233	PC
A	1276	Printer
A	1298	Laptop
A	1401	Printer
A	1408	Printer
A	1752	Laptop
B	1121	PC
B	1750	Laptop
C	1321	Laptop
D	1288	Printer
D	1433	Printer
E	1260	PC
E	1434	Printer
E	2112	PC
E	2113	PC

Рис. 1.2. **Дані для заповнення таблиці *Product***

У таблицях БД всі поля обов'язкові для заповнення, але перші поля таблиць *PC*, *Laptop, Printer* має ключове слово – *IDENTITY* (тобто воно заповнюється автоматично), тому можна пропустити цей стовпець.

Якби були наявні поля з типом NULL, тобто необов'язкові для заповнення, можна було в теж їх проігнорувати. А ось якщо спробувати залишити порожнім поле зі значенням NOT NULL, то сервер видасть повідомлення про помилку і не виконає запит. Крім того, під час внесення даних сервер перевіряє зв'язки між таблицями. Тому не вдасться внести в поле, що є зовнішнім ключем, значення, відсутнє в пов'язаній таблиці. У цьому буде можливість переконатися, вносячи дані в таблиці, що залишилися.

Заповнити таблицю *PC* згідно з рис. 1.3, таблицю *Laptop* (рис. 1.4) і *Printer* (рис. 1.5).

code	model	speed	ram	hd	cd	price
1	1232	500	64	5.0	12x	600.0000
10	1260	500	32	10.0	12x	350.0000
11	1233	900	128	40.0	40x	980.0000
12	1233	800	128	20.0	50x	970.0000
2	1121	750	128	14.0	40x	850.0000
3	1233	500	64	5.0	12x	600.0000
4	1121	600	128	14.0	40x	850.0000
5	1121	600	128	8.0	40x	850.0000
6	1233	750	128	20.0	50x	950.0000
7	1232	500	32	10.0	12x	400.0000
8	1232	450	64	8.0	24x	350.0000
9	1232	450	32	10.0	24x	350.0000

Рис. 1.3. **Дані для заповнення таблиці *PC***

code	model	speed	ram	hd	price	screen
1	1298	350	32	4.0	700.0000	11
2	1321	500	64	8.0	970.0000	12
3	1750	750	128	12.0	1200.0000	14
4	1298	600	64	10.0	1050.0000	15
5	1752	750	128	10.0	1150.0000	14
6	1298	450	64	10.0	950.0000	12

Рис. 1.4. **Дані для заповнення таблиці *Laptop***

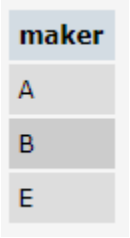
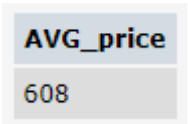
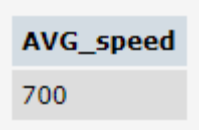
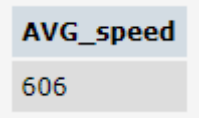
code	model	color	type	price
1	1276	n	Laser	400.0000
2	1433	y	Jet	270.0000
3	1434	y	Jet	290.0000
4	1401	n	Matrix	150.0000
5	1408	n	Matrix	270.0000
6	1288	n	Laser	400.0000

Рис. 1.5. **Дані для заповнення таблиці *Printer***

Перевірити правильність заповнення всіх таблиць.

3. Після створення таблиць виконайте наведені завдання.

<table>
<tr><th>№ з/п</th><th>Завдання</th><th>Відповідь</th></tr>
<tr><td>1</td><td>2</td><td>3</td></tr>
<tr><td>1</td><td>Знайдіть номер моделі, швидкість та розмір жорсткого диска для всіх ПК вартістю менше 500 дол. США. Вивести: model, speed та hd</td><td><table><tr><th>model</th><th>speed</th><th>hd</th></tr><tr><td>1232</td><td>500</td><td>10.0</td></tr><tr><td>1232</td><td>450</td><td>8.0</td></tr><tr><td>1232</td><td>450</td><td>10.0</td></tr><tr><td>1260</td><td>500</td><td>10.0</td></tr></table></td></tr>
<tr><td>2</td><td>Знайдіть виробників принтерів. Вивести: maker</td><td><table><tr><th>maker</th></tr><tr><td>A</td></tr><tr><td>D</td></tr><tr><td>E</td></tr></table></td></tr>
<tr><td>3</td><td>Знайдіть номер моделі, обсяг пам'яті та розміри екранів ПК-блокнотів, ціна яких перевищує 1 000 дол. США</td><td><table><tr><th>model</th><th>ram</th><th>screen</th></tr><tr><td>1750</td><td>128</td><td>14</td></tr><tr><td>1298</td><td>64</td><td>15</td></tr><tr><td>1752</td><td>128</td><td>14</td></tr></table></td></tr>
<tr><td>4</td><td>Знайдіть усі записи таблиці Printer для кольорових принтерів</td><td><table><tr><th>code</th><th>model</th><th>color</th><th>type</th><th>price</th></tr><tr><td>3</td><td>1434</td><td>y</td><td>Jet</td><td>290.0000</td></tr><tr><td>2</td><td>1433</td><td>y</td><td>Jet</td><td>270.0000</td></tr></table></td></tr>
<tr><td>5</td><td>Знайдіть номер моделі, швидкість та розмір жорсткого диска ПК, що мають 12x або 24x CD та ціну менше 600 дол. США</td><td><table><tr><th>model</th><th>speed</th><th>hd</th></tr><tr><td>1232</td><td>500</td><td>10.0</td></tr><tr><td>1232</td><td>450</td><td>8.0</td></tr><tr><td>1232</td><td>450</td><td>10.0</td></tr><tr><td>1260</td><td>500</td><td>10.0</td></tr></table></td></tr>
<tr><td>6</td><td>Для кожного виробника, що випускає ноутбуки з об'ємом жорсткого диска щонайменше 10 Гбайт, знайти швидкості таких ПК-блокнотів. Висновок: виробник, швидкість</td><td><table><tr><th>maker</th><th>speed</th></tr><tr><td>A</td><td>450</td></tr><tr><td>A</td><td>600</td></tr><tr><td>A</td><td>750</td></tr><tr><td>B</td><td>750</td></tr></table></td></tr>
</table>

1	2	3
7	Знайдіть виробників ПК із процесором не менше 450 МГц. Вивести: *maker*	
8	Знайдіть моделі принтерів, які мають найвищу ціну. Вивести: *model, price*	
9	Знайдіть середню швидкість ПК	
10	Знайдіть середню швидкість ПК-блокнотів, ціна яких перевищує 1 000 дол. США	
11	Знайдіть середню швидкість ПК, випущених виробником А	

Контрольні запитання

1. Які операції можна програмувати за допомогою мови SQL?
2. Які ключові слова використовуються під час організації запитів?
3. Які мови програмування підтримує SQL?
4. Який символ дає повідомлення про те, що команда сформована і готова до виконання?

Лабораторна робота 2
Об'єднання таблиць. Угруповання записів і функція Count ()

Мета роботи: навчитися виконувати об'єднання таблиць і угруповання записів.

Завдання

1. Запустити програму *MS SQL Management Studio.*
2. Виконати завдання, перераховані в "Порядку виконання".

Зміст звіту

Найменування і мета роботи.

Завдання.

Теоретична частина

Об'єднання

Для об'єднання запитів використовується службове слово *Union*:

<запит 1>
UNION [ALL]
<запит 2>

Пропозиція *Union* приводить до появи в результівному наборі всіх рядків кожного із запитів. При цьому, якщо визначено параметр *ALL*, то зберігаються всі дублікати вихідних рядків, а в іншому випадку в результівному наборі присутні тільки унікальні рядки. Слід зауважити, що можна пов'язувати разом будь-яку кількість запитів. Крім того, за допомогою дужок можна задавати порядок об'єднання.

Операція об'єднання може бути виконана тільки у разі виконання таких умов:

- кількість вихідних стовпців кожного із запитів повинна бути однаковим;
- вихідні стовпчики кожного із запитів повинні бути сумісні між собою (у порядку їх слідування) за типами даних;
- у результівному наборі використовують назви стовпців, задані в першому запиті;
- пропозицію *ORDER BY* застосовують до результату з'єднання, тому воно може бути зазначено лише в кінці всього складеного запиту.

Знайти номери моделей і ціни ПК і портативних комп'ютерів. Сортування ціни за спадальною ознакою:

```
SELECT model, price
FROM PC
UNION
SELECT model, price
FROM Laptop
ORDER BY price DESC;
```

Знайти тип продукції, номер моделі і ціну ПК і портативних **комп'ютерів**. Сортування ціни по спадаючій:

```
SELECT Product.type, PC.model, price
FROM PC INNER JOIN Product
ON PC.model = Product.model
UNION
SELECT Product.type, Laptop.model, price
FROM Laptop INNER JOIN Product
ON Laptop.model = Product.model
ORDER BY price DESC;
```

Нехай наявна база даних "форум" і необхідно дізнатися, які теми і якими авторами були створені. Для цього найпростіше звернутися до таблиці "Теми" (*topics*).

Але, що якщо необхідно, щоб у відповіді на запит були не ідентифікатори авторів, а їх імена, то вкладені запити не допоможуть, оскільки в кінцевому підсумку вони видають дані з однієї таблиці. А необхідно отримати дані з двох таблиць ("Теми" і "Користувачі") й об'єднати їх в одну. Запити, які дозволяють це зробити, у SQL називають *Об'єднаннями*.

Синтаксис найпростішого об'єднання такий:

```
SELECT назва_стовпців_таблиці_1, назва_стовпців_таблиці_2
FROM назва_таблиці_1, назва_таблиці_2;
```

Створимо просте об'єднання:

```
SELECT topic_name, name FROM topics, users;
```

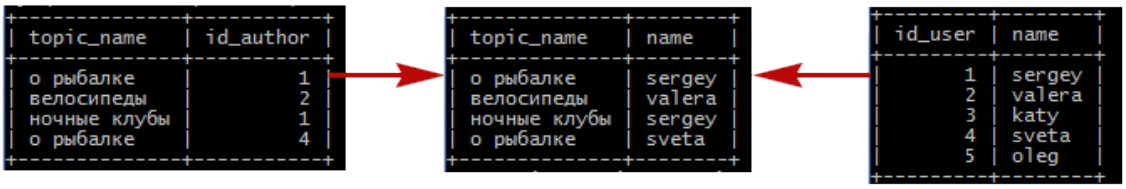
Таке об'єднання науково називають декартовим, коли кожному рядку першої таблиці ставиться у відповідність кожен рядок другої таблиці. Можливо, бувають випадки, коли таке об'єднання корисно, але це явно не цей випадок.

Щоб результівна таблиця виглядала так, як було задумано, необхідно вказати умову об'єднання. Пов'язуємо таблиці за ідентифікатором автора, це і буде умовою. Тобто варто вказати в запиті, що необхідно виводити тільки ті рядки, в яких значення поля *id_author* таблиці *topics* збігаються зі значеннями поля *id_user* таблиці *users*:

```
SELECT topic_name, name
```

```
FROM topics, users
WHERE topics.id_author = users.id_user;
```

На схемі буде зрозуміліше:



Тобто в запиті зробили таку умову: якщо в обох таблицях є однакові ідентифікатори, то рядки з цим ідентифікатором необхідно об'єднати в один результівний рядок.

Варто звернути увагу на дві речі:

- Якщо в одній з поєднуваних таблиць є рядок з ідентифікатором, якого немає в іншій об'єднаній таблиці, то в результівній таблиці рядки з таким ідентифікатором не буде. У цьому прикладі є користувач *Oleg* (id = 5), але він не створював теми, тому в результаті запиту його немає.
- Під час вказівки умови назву стовпця пишеться після назви таблиці, в якій цей стовпець знаходиться (через точку). Це зроблено, щоб уникнути плутанини, адже стовпці в різних таблицях можуть мати однакові назви, і MS SQL може не зрозуміти, про які конкретно стовпці йдеться.

Коректний синтаксис об'єднання з умовою виглядає так:

```
SELECT назва_таблиці_1.назва_стовпця 1_таблиці_1,
  назва_таблиці_1.назва_стовпця 2_таблиці_1,
  назва_таблиці_2.назва_стовпця 1_таблиці_2,
  назва_таблиці_2.назва_стовпця 2_таблиці_2
FROM
  назва_таблиці_1, назва_таблиці_2
WHERE
  назва_таблиці_1.назва_стовпчика_за_яким_об'єднуєм =
  назва_таблиці_2.назва_стовпчика_за_яким_об'єднуєм;
```

Якщо назва стовпця унікальна, то назву таблиці можна опустити (як це зробили в прикладі).

Як зрозуміло, об'єднання дають можливість обирати будь-яку інформацію з будь-яких таблиць, причому об'єднання таблиць може бути і три, і чотири, та й умова для об'єднання може бути не одна.

Для прикладу варто створити запит, який покаже всі повідомлення, до яких тем вони належать, авторів цих повідомлень. Зазвичай вся ця інформація зберігається в таблиці "Повідомлення" (*posts*).

Але щоб замість ідентифікаторів відображалися імена авторів і назви тем, доведеться зробити об'єднання трьох таблиць (рис. 2.1):

```
SELECT posts.message, topics.topic_name, users.name
FROM posts, topics, users
WHERE posts.id_author = users.id_user
AND posts.id_topic = topics.id_topic;
```

```
mysql> SELECT posts.message, topics.topic_name, users.name
    -> FROM posts, topics, users
    -> WHERE posts.id_author=users.id_user AND posts.id_topic=topics.id_topic;
+--------------------------+--------------+--------+
| message                  | topic_name   | name   |
+--------------------------+--------------+--------+
| согласен с автором       | о рыбалке    | sergey |
| можно сделать по другому | ночные клубы | valera |
| всем привет              | о рыбалке    | katy   |
| у меня другое мнение     | о рыбалке    | sergey |
+--------------------------+--------------+--------+
4 rows in set (0.09 sec)

mysql>
```

Рис. 2.1. **Результат об'єднання трьох таблиць**

Тобто було об'єднано таблиці "Повідомлення" і "Користувачі" умовою posts.id_author = users.id_user, а таблиці "Повідомлення" і "Теми" – умовою posts.id_topic = topics.id_topic (рис. 2.2).

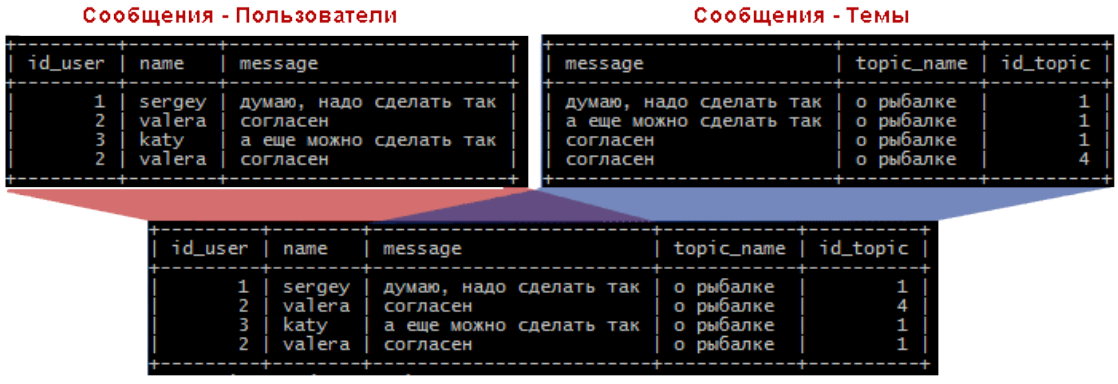


Рис. 2.2. **Об'єднання таблиці "Повідомлення" і "Користувачі"**

Об'єднання, які було розглянуто, називають *Внутрішніми об'єднаннями*. Такі об'єднання пов'язують рядки однієї таблиці з рядками другої таблиці (а може ще й третьої таблиці). Але бувають ситуації, коли необхідно, щоб в результат були включені рядки, які не мають пов'язаних. Наприклад, коли створювали запит, які теми і якими авторами були створені, користувач *Oleg* у результівну таблицю не потрапив, тому що сам не створював, а тому і пов'язаного рядку в об'єднаній таблиці не мав.

Тому, якщо буде потрібно скласти дещо інший запит – вивести всіх користувачів і теми, які вони створювали, якщо такі є – то доведеться скористатися *Зовнішнім об'єднанням*, що дозволяє виводити всі рядки однієї таблиці і наявні пов'язані з ними рядки з іншої таблиці. Для цього варто трохи змінити запит:

```
SELECT users.name, topics.topic_name
FROM users LEFT JOIN topics
ON users.id_user = topics.id_author;
```

Буде отримано бажаний результат – усі користувачі і теми, ними створені. Якщо користувач не створював тему, то в відповідному стовпці стоїть значення NULL.

Отже, було додано в запит ключове слово – *LEFT JOIN*, вказавши тим самим, що з таблиці зліва треба взяти все рядки, і поміняли ключове слово *WHERE* на *ON*. Крім ключового слова *LEFT JOIN* може бути використано ключове слово *RIGHT JOIN*. Тоді будуть вибиратися всі рядки з правої таблиці і наявні пов'язані з ними з лівої таблиці. І нарешті, можливо повне зовнішнє об'єднання, яке отримає всі рядки з обох таблиць і зв'яже між собою ті, які можуть бути пов'язані. Ключове слово для повного зовнішнього об'єднання – *FULL JOIN*.

Варто змінити у запиті лівостороннє об'єднання на правостороннє:

```
SELECT users.name, topics.topic_name
FROM users RIGHT JOIN topics
ON users.id_user = topics.id_author;
```

Тепер є всі теми (всі рядки з правої таблиці), а ось користувачі тільки ті, які теми створювали (тобто з лівої таблиці вибираються тільки ті рядки, які пов'язані з правою таблицею).

Підведемо підсумок. Синтаксис для зовнішнього об'єднання такий:

SELECT назва_таблиці_1.назва_стовпця, назва_таблиці_2.назва_стовпця
FROM назва_таблиці_1 ТИП ОБ'ЄДНАННЯ назва_таблиці_2
ON умова_об'єднання;

Перетин і різниця

У стандарті мови SQL є пропозиції оператора *SELECT* для виконання операцій перетину і різниці результатів запитів-операндів. Цими пропозиціями є *INTERSECT [ALL]* (перетин) і *EXCEPT [ALL]* (різниця), які працюють аналогічно пропозицією *UNION*. У результівний набір потрапляють тільки ті рядки, які присутні в обох запитах (*INTERSECT*) або тільки ті рядки першого запиту, які відсутні в другому (*EXCEPT*). При цьому обидва запити, які беруть участь в операції, повинні мати однакову кількість стовпців, і відповідні стовпці повинні мати однакові типи даних. Назва стовпців результівного набору формують із заголовків першого запиту.

Якщо не використовують ключове слово *ALL* (за замовчуванням мається на увазі *DISTINCT*), то під час виконання операції автоматично усуваються дублікати рядків. Якщо вказано ALL, то кількість дубльованих рядків підпорядковується таким правилам (n1 – кількість дублікатів рядків першого запиту, n2 – кількість дублікатів рядків другого запиту):

- INTERSECT ALL: min (n1, n2).
- EXCEPT ALL: n1 - n2, якщо n1> n2.

Знайти кораблі, які присутні як у таблиці Ships, так і в таблиці *Outcomes*.

SELECT name FROM Ships
INTERSECT
SELECT ship FROM Outcomes;

У реляційній алгебрі операція перетину є комутативність, оскільки вона може бути застосована до відносин з однаковими заголовками. У SQL теж можна поміняти запити місцями. Наведене рішення дасть той же результат, що і таке:

SELECT ship FROM Outcomes

```
INTERSECT
SELECT name FROM Ships;
```

За винятком заголовка. У першому випадку єдиний стовпець буде мати заголовок *name*, а в другому – *ship*.

Знайти кораблі з таблиці *Outcomes*, які відсутні в таблиці *Ships*.

Завдання легко вирішується за допомогою оператора *EXCEPT*:

```
SELECT ship FROM Outcomes
EXCEPT
SELECT name FROM Ships;
```

Операція різниці не є комутативність, тому якщо переставити місцями запити, то буде отримано рішення зовсім іншого завдання: знайти кораблі з таблиці *Ships*, які відсутні в таблиці *Outcomes*.

Слід зазначити, що не всі СУБД підтримують ці пропозиції в операторі *SELECT*. Немає підтримки *INTERSECT / EXCEPT*, наприклад, у MySQL, а в MS SQL Server вона з'явилася, лише починаючи з версії 2005 р., і то без ключового слова *ALL*. *ALL* з *INTERSECT / EXCEPT* також ще не реалізована в *Oracle*. Слід зазначити, що замість стандартного *EXCEPT* в *Oracle* використовується ключове слово *MINUS*.

Тому для виконання операцій перетину і різниці можуть бути використані інші засоби. Тут доречно зауважити, що один і той же результат можна отримати за допомогою різних формулювань оператора *SELECT*. У разі перетину і різниці можна скористатися предикатом існування *EXISTS*.

На закінчення варто розглянути приклад використання операції *INTERSECT ALL.*

Знайти виробників, які випускають не менше двох моделей ПК і не менше двох моделей принтерів.

```
SELECT maker FROM (
SELECT maker FROM Product WHERE type = 'PC'
INTERSECT ALL
SELECT maker FROM Product WHERE type = 'Printer')
GROUP BY maker
HAVING COUNT(*) > 1;
```

INTERSECT ALL у підзапиті цього рішення залишить мінімальну кількість дублікатів, тобто якщо виробник випускає дві моделі ПК і одну модель принтера (або навпаки), то він буде присутній в результівному наборі один раз. Далі виконують угруповання за виробником, залишаючи тільки тих з них, хто присутній у результатах підзапиту більше одного разу.

Угруповання записів і функція COUNT ()

Варто згадати, які повідомлення, і в яких темах наявні. Для цього можна скористатися звичним запитом:

```
SELECT * FROM posts;
```

Якщо треба лише дізнатися, скільки повідомлень на форумі є, то для цього можна скористатися вбудованою функцією *COUNT()*. Ця функція підраховує кількість рядків. Причому, якщо в якості аргумента цієї функції виступає *, то підраховують всі рядки таблиці. А якщо в якості аргумента вказують назву стовпця, то підраховують тільки ті рядки, які мають значення в зазначеному стовпці.

У даному прикладі обидва аргументи дадуть однаковий результат, тому що всі стовпці таблиці мають тип NOT NULL. Слід написати запит, використовуючи як аргумент стовпець id_topic:

```
SELECT COUNT (id_topic) FROM posts;
```

Результат виконання запиту представлено на рис. 2.3.

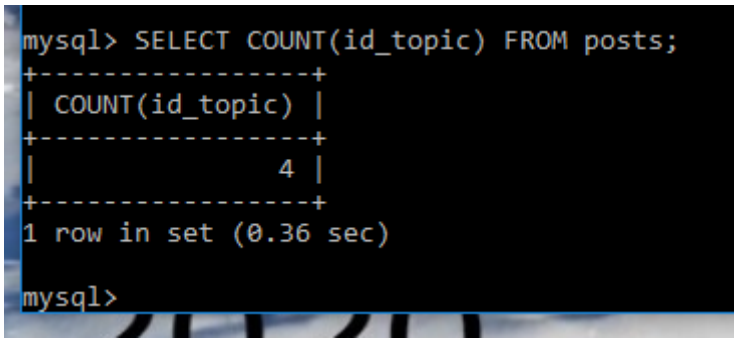


Рис. 2.3. **Результат виконання функції *COUNT()***

Отже, у темах є 4 повідомлення. Якщо є необхідність дізнатися, скільки повідомлень є в кожній темі, то для цього знадобиться згрупувати повідомлення за темами і обчислити для кожної групи кількість повідомлень.

Для угруповання у SQL використовують оператор *GROUP BY*. Запит тепер буде виглядати так:

```
SELECT id_topic, COUNT (id_topic) FROM posts
GROUP BY id_topic;
```

Оператор *GROUP BY* вказує СУБД групувати дані за стовпцем *id_topic* (тобто кожна тема – окрема група) і для кожної групи підрахувати кількість рядків. Результат виконання запиту представлено на рис. 2.4.

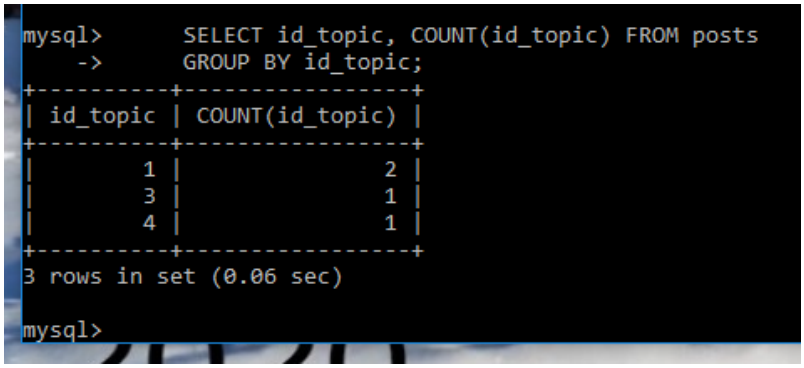


Рис. 2.4. **Результат виконання функції *GROUP BY***

У темі з id = 1 два повідомлення, з id = 3 одне повідомлення і з id = 4 – одне. До речі, якби в полі *id_topic* були можливі відсутності значень, то такі рядки були б об'єднані в окрему групу зі значенням NULL.

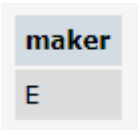
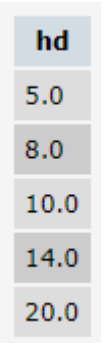
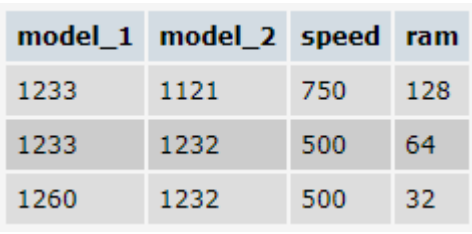
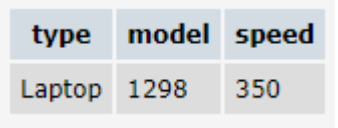
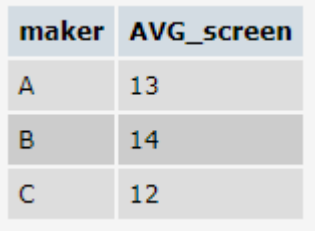
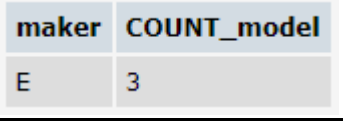
Нехай інтерес становлять тільки ті групи, в яких більше двох повідомлень. У звичайному запиті вказали б умову за допомогою оператора *WHERE*, але цей оператор вміє працювати тільки з рядками, а для груп ті ж функції виконує оператор *HAVING*:

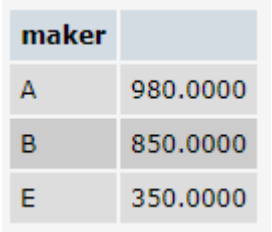
```
SELECT id_topic, COUNT (id_topic) FROM posts
GROUP BY id_topic
HAVING COUNT (id_topic)> 2;
```

На попередньому занятті було розглянуто, які умови можна задавати оператором *WHERE*, ті ж умови можна задавати й оператором *HAVING*, тільки **треба запам'ятати, що *WHERE* фільтрує рядки, а *HAVING* – групи**.

Порядок виконання

Виконайте наведені завдання.

№ з/п	Завдання	Відповідь
1	2	3
1	Знайдіть номери моделей і ціни всіх наявних у продажу продуктів (будь-якого типу) виробника B (латинська буква)	
2	Знайдіть виробника, що випускає ПК, але не ноутбуки	
3	Знайдіть розміри жорстких дисків, які збігаються у двох і більше PC. Вивести: *hd*	
4	Знайдіть пари моделей PC, що мають однакові швидкість і ram. У результаті кожну пару вказують тільки один раз, тобто (i, j), але не (j, i), Порядок виведення: модель з великим номером, модель з меншим номером, швидкість і ram	
5	Знайдіть моделі ПК-блокнотів (*Laptop*), швидкість яких менше швидкості будь-якого з ПК. Вивести: *type, model, speed*	
6	Знайдіть виробників найдешевших кольорових принтерів. Вивести: *maker, price*	
7	Для кожного виробника, що має моделі в таблиці *Laptop*, знайдіть середній розмір екрана ПК-блокнотів, які випускаються ним. Вивести: *maker*, середній розмір екрана	
8	Знайдіть виробників, що випускають щонайменше три різних моделі ПК. Вивести: *maker*, кількість моделей ПК	

1	2	3
9	Знайдіть максимальну ціну ПК, що випускаються кожним виробником, у якого є моделі в таблиці PC. Вивести: *maker*, максимальна ціна	 maker | A | 980.0000 B | 850.0000 E | 350.0000
10	Для кожного значення швидкості ПК, що перевищує 600 МГц, визначте середню ціну ПК з такою ж швидкістю. Вивести: *speed*, середня ціна	 speed | AVG_price 750 | 900.0000 800 | 970.0000 900 | 980.0000

Лабораторна робота 3
Загальні табличні вирази (CTE)

Мета роботи: навчитися виконувати SQL запити з використанням загальних табличних виразів.

Завдання

1. Запустити програму *MS SQL Management Studio.*
2. Виконати завдання.

Зміст звіту

Найменування і мета роботи.

Завдання.

Теоретична частина

Загальні табличні вирази (CTE)

Щоб з'ясувати призначення загальних табличних виразів, слід почати з прикладу.

Знайти максимальну суму доходів/витрат серед усіх чотирьох таблиць бази даних "Вторсировина", а також тип операції, дату і пункт прийому, коли і де вона була зафіксована.

Завдання можна вирішити, наприклад, у такий спосіб.

```
SELECT inc AS max_sum, type, date, point
FROM (SELECT inc, 'inc' type, date, point
FROM Income UNION ALL SELECT inc, 'inc' type, date, point
FROM Income_o
UNION ALL
```

```
SELECT out, 'out' type, date, point
FROM Outcome_o
UNION ALL
SELECT out, 'out' type, date, point FROM Outcome
WHERE inc > = ALL (SELECT inc FROM Income
UNION ALL
SELECT inc FROM Income_o
UNION ALL SELECT out FROM Outcome_o
UNION ALL SELECT out FROM Outcome);
```

Тут варто спочатку об'єднати всю наявну інформацію, а потім обрати тільки ті рядки, у яких сума не менше, ніж кожна з сум тієї ж вибірки з чотирьох таблиць.

Фактично, двічі було написано код об'єднань чотирьох таблиць. Як уникнути цього? Можна створити подання, а потім адресувати запит вже до нього:

```
CREATE VIEW Inc_Out AS
SELECT inc, 'inc' type, date, point
FROM Income
UNION ALL
SELECT inc, 'inc' type, date, point
FROM Income_o
UNION ALL
SELECT out, 'out' type, date, point
FROM Outcome_o
UNION ALL
SELECT out, 'out' type, date, point
FROM Outcome;
GO
SELECT inc AS 'max_sum', type, date, point
FROM Inc_Out WHERE inc > = ALL (SELECT inc FROM Inc_Out);
```

Отже, *CTE* відіграє роль подання, яке створюється в рамках одного запиту і, не зберігається як об'єкт схеми. Попередній варіант рішення можна переписати за допомогою *CTE* таким чином:

```
WITH Inc_Out AS (
SELECT inc, 'inc' type, date, point
```

```
FROM Income
UNION ALL
SELECT inc, 'inc' type, date, point
FROM Income_o
UNION ALL
SELECT out, 'out' type, date, point
FROM Outcome_o
UNION ALL
SELECT out, 'out' type, date, point FROM Outcome)
SELECT inc AS 'max_sum', type, date, point
FROM Inc_Out WHERE inc > = ALL (SELECT inc FROM Inc_Out);
```

Як можна бачити, все аналогічно використанню уявлення за винятком обов'язкових дужок, обмежують запит; формально, достатній але лише замінити *CREATE VIEW* на *WITH*. Як і для подання, у дужках після назви *CTE* може бути вказаний список стовпців, якщо нам буде потрібно включити їх не всі з підмета запиту і/або перейменувати. Наприклад, додано додатково визначення мінімальної суми в попередній запит.

```
WITH Inc_Out (m_sum, type, date, point) AS (
SELECTinc, 'inc' type, date, point
FROM Income
UNION ALL
SELECT inc, 'inc' type, date, point
FROM Income_o
UNION ALL
SELECT out, 'out' type, date, point
FROM Outcome_o
UNION ALL
SELECT out, 'out' type, date, point FROM Outcome)
SELECT 'max' min_max, * FROM Inc_Out
WHERE m_sum > = ALL (SELECT m_sum FROM Inc_Out )
UNION ALL
SELECT 'min', * FROM Inc_Out
WHERE m_sum <= ALL (SELECT m_sum FROM Inc_Out);
```

Загальні табличні вирази дозволяють істотно зменшити обсяг коду, якщо багаторазово доводиться звертатися до одних і тих же похідних таблиць.

Слід зауважити, що *CTE* можуть використовуватися не тільки з оператором *SELECT*, а й з іншими операторами мови *DML*. Варто вирішити таке завдання:

Пасажирів рейсу 7772 від 11 листопада 2005 року потрібно надіслати іншим найближчим рейсом, що вилітають пізніше в той же день в той же пункт призначення.

Тобто це завдання на оновлення записів у таблиці *Pass_in_trip*.

Слід почати з таблиці, яку потрібно буде оновити:

```
WITH Trip_for_replace AS (
SELECT * FROM Pass_in_trip
WHERE trip_no = 7772 AND date = '20051129')
SELECT * FROM Trip_for_replace;
```

Оскільки *CTE* грають роль уявлень, то їх можна в принципі використовувати для поновлення даних.

Тепер нам потрібна інформація про рейс 7772 для того, щоб знайти найближчий до нього відповідний рейс. Варто додати ще одне *CTE* у визначення:

```
WITH Trip_for_replace AS(
SELECT * FROM Pass_in_trip
WHERE trip_no = 7772 AND date = '20051129'),
Trip_7772 AS (SELECT * FROM Trip WHERE trip_no = 7772)
SELECT * FROM Trip_7772;
```

Зверніть увагу, що в одному запиті можна визначити будь-яку кількість загальних табличних виразів. І що особливо важливо, *CTE* може включати посилання на інше *CTE* (зверніть увагу на посилання *Trip_7772* у визначенні *Trip_candidates*).

```
WITH Trip_for_replace AS (
SELECT * FROM Pass_in_trip
WHERE trip_no = 7772 AND date = '20051129'),
Trip_7772 AS ( SELECT * FROM Trip WHERE trip_no = 7772),
Trip_candidates AS (SELECT Trip. *
FROM Trip, Trip_7772
```

```
WHERE Trip.town_from + Trip.town_to = Trip_7772.town_from +
Trip_7772.town_to AND Trip.time_out > Trip_7772.time_out)
SELECT * FROM Trip_candidates;
```

Trip_candidates – це табличне вираження, яке визначає кандидатів на заміну, а саме, рейси, що вилітають пізніше, ніж 7772, і які відбуваються між тими ж містами. Використовуємо конкатенацію рядків *town_from* + *town_to*, щоб не писати окремі критерії для пункту відправлення і місця призначення.

Знайдемо тепер серед рядків-кандидатів найбільш близький за часом рейс:

```
WITH Trip_for_replace AS (
SELECT * FROM Pass_in_trip
WHERE trip_no = 7772 AND date = '20051129'),
Trip_7772 AS (SELECT * FROM Trip WHERE trip_no = 7772),
Trip_candidates AS (SELECT Trip. * FROM Trip, Trip_7772
WHERE Trip.town_from + Trip.town_to = Trip_7772.town_from +
Trip_7772.town_to AND Trip.time_out > Trip_7772.time_out),
Trip_replace AS (
SELECT * FROM Trip_candidates
WHERE time_out <= ALL (SELECT time_out FROM Trip_candidates))
SELECT * FROM Trip_replace;
```

Тепер залишилося останній оператор *SELECT* замінити на *UPDATE*, щоб вирішити задачу:

```
WITH Trip_for_replace AS (
SELECT * FROM Pass_in_trip
WHERE trip_no = 7772 AND date ='20051129'),
Trip_7772 AS (SELECT * FROM Trip WHERE trip_no = 7772),
Trip_candidates AS (
SELECT Trip. * FROM Trip, Trip_7772
WHERE Trip.town_from + Trip.town_to = Trip_7772.town_from +
Trip_7772.town_to AND Trip.time_out > Trip_7772.time_out),
Trip_replace AS (SELECT * FROM Trip_candidates
WHERE time_out <= ALL (SELECT time_out FROM Trip_candidates))
```

UPDATE Trip_for_replace SET trip_no = (SELECT trip_no FROM Trip_replace);

Тут варто виходити з досить природного припущення про те, що між заданими містами немає двох рейсів, які б відправлялися в один і той же час в одному напрямі. В іншому випадку, знадобився б додатковий критерій для відбору єдиного рейсу, тому що наша мета – оновлення даних, а не подання всіх можливих кандидатів на заміну.

З використанням *CTE* з оператором *DELETE* можна познайомитися на прикладі видалення дублікатів рядків з таблиці.

Запит, який був використаний для видалення дублікатів у SQL Server

```
WITH CTE AS (
SELECT name, ROW_NUMBER() OVER (PARTITION BY name ORDER BY name) rnk
FROM T
)
DELETE FROM CTE
WHERE rnk > 1;
```

У SQL завершиться помилкою:

ПОМИЛКА: ставлення "cte" не існує

Ця помилка означає, що можна видаляти рядки з базових таблиць, але не з *CTE*. Проте, можливо виконати видалення дублікатів одним запитом, використовуючи *CTE*.

Варто вчинити таким чином:

1. Видалити всі рядки з базової таблиці, повертаючи їх у табличне вираження (перше *CTE*).
2. Використовуючи результат першого етапу, формуємо унікальні рядки, які повинні залишитися у таблиці (друге *CTE*).
3. Вставити рядки, отримані на другому етапі в базову таблицю.

Скористаємося таблицею з цитованого прикладу, щоб написати запит:

```
CREATE TABLE T (name varchar (10));
INSERT INTO T VALUES
('John'),
('Smith'),
```

```
('John'),
('Smith'),
('Smith'),
('Tom');
```

Ось і сам запит

```
WITH t_deleted AS
(DELETE FROM T returning *), – перший етап
t_inserted AS
(SELECT name, ROW_NUMBER() OVER (PARTITION BY name ORDER BY name) rnk
FROM t_deleted) – другий етап
INSERT INTO T SELECT name FROM t_inserted
WHERE rnk = 1; – третій етап (сюди переносимо умову відбору з другого етапу для скорочення коду)
```

Відповідно до стандарту можуть використовуватися імена з обмежувачами (*delimited identifier*), при цьому в якості обмежувача застосовується символ подвійної лапки ("). Такий прийом допускає присутність в іменах спеціальних символів і зарезервованих слів. Наприклад, запит:

```
SELECT 'SELECT' "SELECT";
```

виведе значення вираження (у даному випадку символьну константу *'SELECT'*) у стовпці з назвою *SELECT*. Тобто варто використовувати зарезервоване слово в якості імені стовпця. Без цього компілятор (SQL Server) не зможе коректно виконати розбір подібного запиту:

```
SELECT 'SELECT' SELECT;
```

і видасть таку помилку:
Incorrect syntax near 'SELECT'.
(Некоректний синтаксис біля 'SELECT')

Крім стандартного обмежувача, різні СУБД допускають використання своїх власних. Наприклад, у SQL Server запит можна написати так:

```
SELECT 'SELECT' [SELECT];
```

У той же час, стандартний обмежувач використовується паралельно, але не скрізь він прийнятий налаштуванням за замовчуванням. У MS SQL налаштування, що відповідає за імена з обмежувачами, можна змінити за допомогою оператора:

SET QUOTED_IDENTIFIER {ON | OFF}

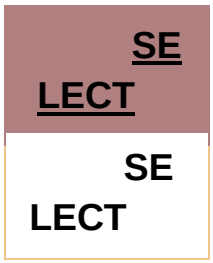
При цьому стандартну поведінку (*ON*) прийнято за замовчуванням.

Щоб заборонити використання подвійної лапки в якості одинарної, можна поміняти налаштування на стандартні. Наступний оператор змінить налаштування, про яку йдеться:

SET GLOBAL sql_mode = 'ANSI_QUOTES';
а цей всі налаштування зробить стандартними:
SET GLOBAL sql_mode = 'ANSI';

Після цього запит

SELECT 'SELECT' "SELECT" ;
дасть



Перетворення типів і оператор CAST

У реалізаціях мови *SQL* може бути виконано неявне переобрання типів. Так, наприклад, у *SQL Server* і *Sybase ASE.Transact-SQL* під час порівняння або комбінування значень типів *smallint* і *int*, дані типу *smallint* неявно перетворюються до типу *int*.

Вивести середню ціну портативних комп'ютерів з попереднім текстом "середня ціна =".

Спроба виконати запит:

SELECT N'Середня ціна =' + AVG(price)
FROM Laptop;

приведе до повідомлення про помилку:

Implicit conversion from data type varchar to money is not allowed. Use the CONVERT function to run this query.

(*Не допускається неявне перетворення типу varchar до типу money. Варто використовувати для виконання цього запиту функцію CONVERT.)*

Це повідомлення означає, що система не може виконати неявне перетворення типу *varchar* до типу *money*. У подібних ситуаціях може допомогти явне перетворення типів. При цьому, як зазначено в повідомленні про помилку, можна скористатися функцією *CONVERT*. Ця функція не стандартизована, тому з метою перенесення рекомендується використовувати стандартний вираз *CAST*. З нього і слід почати.

Отже, якщо переписати запит у вигляді:

```
SELECT N'Середня ціна =' + CAST(AVG(price) AS CHAR(15))
FROM Laptop;
```

у результаті отримаємо те, що було потрібно:

Середня ціна = 1 003.33

Було використано вираз явного перетворення типів *CAST* для приведення середнього значення ціни до строкового подання.

Синтаксис виразу *CAST* дуже простий:

CAST (<вираз> AS <тип даних>)

Увага:

Слід мати на увазі, по-перше, що не будь-які перетворення типів можливі (стандарт містить таблицю допустимих перетворень типів даних). По-друге, результат функції CAST для значення виразу, рівного NULL, теж буде NULL.

Слід розглянути ще один приклад.

Визначити середній рік спуску на воду кораблів з таблиці *Ships*.
Запит:

```
SELECT AVG(launched)
FROM Ships;
```

Дасть результат – 1926 р. У принципі все правильно, оскільки було отримано в результаті те, на що було створено запит – рік. Однак середнє арифметичне становитиме приблизно 1926.23. Тут слід нагадати, що агрегатні функції (за винятком функції *COUNT*, яка завжди повертає ціле кількість) успадковують тип даних оброблюваних значень. Оскільки поле *launched* – цілочисельне, отримано середнє значення з відкинутою дробовою частиною (наголошено – не округлене).

А якщо цікавить результат з заданою точністю, наприклад, до двох десяткових знаків, то застосування виразу *CAST* до середнього значення нічого не дасть за вказаною причини. Дійсно,

```
SELECT CAST (AVG(launched) AS NUMERIC(6, 2))
FROM Ships;
```

поверне значення 1926.00. Отже, *CAST* потрібно застосувати до аргументу агрегатної функції:

```
SELECT AVG (CAST (launched AS NUMERIC (6, 2)))
FROM Ships;
```

Результат – 1926.90909. Знову не те. Причина полягає в тому, що під час обчислення середнього значення було виконано неявне перетворення типу. Слід пройти ще один етап:

```
SELECT CAST (AVG (CAST (launched AS NUMERIC (6, 2))) AS NUMERIC (6, 2))
FROM Ships;
```

У результаті отримано те, що потрібно – 1926.91. Однак це рішення виглядає дуже громіздко. Змусимо неявне перетворення типу попрацювати на нас:

```
SELECT CAST (AVG (launched * 1.0) AS NUMERIC (6, 2))
FROM Ships;
```

Тепер було використано неявне перетворення цілочисельного аргумента до точного числового типу (*EXACT NUMERIC*), помноживши його на речову одиницю, після чого застосували явне приведення типу результату агрегатної функції.

Аналогічні перетворення типу можливо виконати за допомогою функції SQL Server *CONVERT*:

```
SELECT CONVERT (NUMERIC (6, 2), AVG (launched * 1.0))
FROM Ships;
```

Функція *CONVERT* має такий синтаксис:

```
CONVERT (<тип_даних [(<довжина>)]>, <вираз>[<стиль>])
```

Основна відмінність функції *CONVERT* від функції *CAST* полягає в тому, що перша дозволяє форматувати дані (наприклад, темпоральні дані типу *datetime*) під час перетворення їх до символьного типу і вказувати формат у процесі зворотного перетворення. Різні цілочисельні значення необов'язкового аргумента стиль відповідають різним типам форматів. Далі розглянути такий приклад:

```
SELECT CONVERT (char(25), CONVERT (datetime, '20030722'));
```

Тут перетворити строкове подання дати до типу *datetime*, після чого виконати зворотне перетворення, щоб продемонструвати результат форматування. Оскільки значення аргумента стиль не задано використовується значення за замовчуванням (0 або 100). У результаті буде отримано:

```
Jul 22 2003 12:00 AM
```

Далі наведено деякі інші значення аргумента стиль і результат, отриманий на наведеному прикладі. Слід зауважити, що збільшення значення стиль на 100 приводить до чотиризначного відображення року.

1	07/22/03
11	03/07/22
3	22/07/03
121	2003-07-22 00: 00: 00.000

Перетворення типу *money*

Грошовий тип даних не є стандартним. У SQL Server є два грошових типи:

money: діапазон значень від -922,337,203,685,477.5808 до 922,337,203,685,477.5807;

smallmoney: діапазон значень від -214 748,3648 до 214 748,3647.

Точність обох типів одна десятитисячна.

Константу типу *money* можна задати за допомогою префікса $, або ж використовувати перетворення типів, наприклад:

```
SELECT 1.2 num, $ 1.2 mn1, CAST (1.2 AS MONEY) mn2;
```

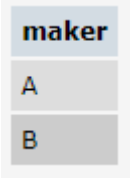
Зверніть увагу на кому як на роздільник "рублів" і "копійок" – не крапка!

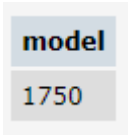
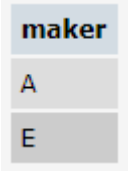
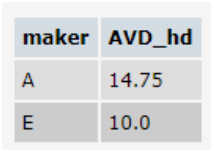
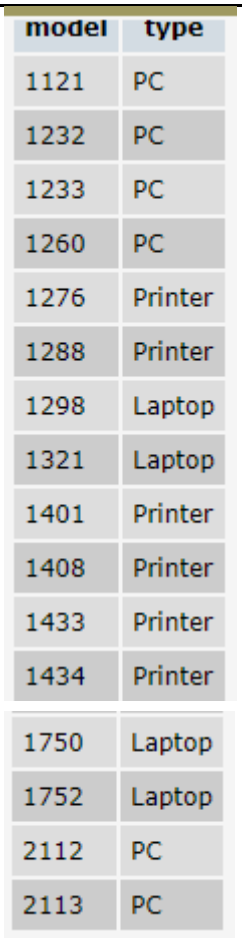
Перетворення до цілого типу для чисел і грошей виконують по різному: в першому випадку дрібну частину відкидають, у другому відбувається округлення. Гроші просто так втрачати не можна!

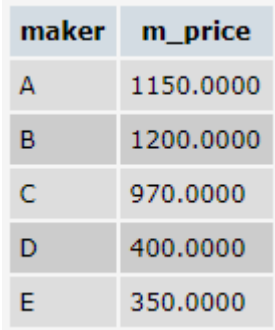
```
SELECT CAST (1.75 AS INT) int_num, CAST ($ 1.75 AS INT) int_mon;
```

Порядок виконання

Виконайте наведені завдання.

№ з/п	Завдання	Відповідь
1	2	3
1	Знайдіть виробників, які виробляли б як ПК зі швидкістю не менше 750 МГц, так і ПК-блокноти зі швидкістю не менше 750 МГц. Вивести: *maker*	

1	2	3
2	Перерахуйте номери моделей будь-яких типів, що мають найвищу ціну за всієї наявної в базі даних продукції	
3	Знайдіть виробників принтерів, які виробляють ПК з найменшим обсягом *ram* і з найшвидшим процесором серед усіх ПК, що мають найменший обсяг *ram*. Вивести: *maker*	
4	Знайдіть середню ціну ПК і ПК-блокнотів, випущених виробником А (латинська буква). Вивести: одна загальна середня ціна	
5	Знайдіть середній розмір диска ПК кожного з тих виробників, які випускають і принтери. Вивести: *maker*, середній розмір *hd*	
6	Використовуючи таблицю *product*, визначити кількість виробників, що випускають по одній моделі	
7	У таблиці *Product* знайти моделі, які складаються тільки з цифр або тільки з латинських букв (AZ, без урахування регістра). Вивести: номер моделі, тип моделі	
8	Знайти виробників, які випускають більше однієї моделі, при цьому всі випускаються виробником моделі є продуктами одного типу. Вивести: *maker, type*	

1	2	3
9	Для кожного виробника, у якого присутні моделі хоча б в одній з таблиць *PC, Laptop* або *Printer*, визначити максимальну ціну на його продукцію. Вивести: назва виробника, якщо серед цін на продукцію даного виробника присутній *NULL*, то виводити для цього виробника *NULL*, інакше максимальну ціну	

maker	m_price
A	1150.0000
B	1200.0000
C	970.0000
D	400.0000
E	350.0000

Лабораторна робота 4
Транзакції SQL

Метою лабораторної роботи є освоєння основ програмування транзакцій з використанням *MS SQL Server*, а також розбір характерних проблем і можливості налаштування транзакцій.

Завдання

1. Виконати завдання.
2. Дати відповіді на питання.
3. Сформувати звіт.

Зміст звіту

1. Скріншоти виконаних завдань.
2. Відповіді на питання.

Порядок виконання

Надійний доступ до даних повинен бути заснований на розроблених належним чином транзакціях SQL з урахуванням нульовий толерантності (абсолютну неприпустимість) неправильних даних у базі даних. Програміст, що не має уявлення про необхідні технології транзакцій, може легко порушити цілісність даних у базі даних, а також погіршити або повністю звести до нуля продуктивність.

У першому блоці завдань необхідно провести тести явних і неявних транзакцій SQL, команд *COMMIT* і *ROLLBACK*, а також логіку транзакцій з використанням *MS SQL Server*.

У другому блоці розбираються проблеми паралелізму й аномалії читання даних під час проведенні транзакцій.

У даній лабораторній роботі буде використано програму *SQL Server Management Studio* (*SSMS*). Для початку створити нову базу даних "*TestDB*" як це показано далі і буде використано установки за замовчуванням для всіх параметрів конфігурації, а за допомогою команди *USE* буде використано її в якості поточної бази даних у SQL-сесії.

1. Створіть нову базу даних з назвам "*TestDB*"

CREATE DATABASE TestDB;

2. Виконайте вправи і дайте відповідь на питання.

Частина 1. Експерименти з поодинокими транзакції

За замовчуванням, у *SQL Server* сесії працюють в режимі *AUTOCOMMIT* (автоматичної фіксації) і використання явних транзакцій. Можна побудувати транзакції декількох SQL-команд. Проте, весь сервер може бути налаштований на використання і неявних транзакцій. Але можливе використання неявних транзакцій і тільки в одній сесії SQL, що можна налаштувати за допомогою такої команди SQL:

```
SET IMPLICIT_TRANSACTIONS ON;
```

яка буде в силі до кінця сесії, але у разі необхідності її можна буде вимкнути за допомогою такої команди:

```
SET IMPLICIT_TRANSACTIONS OFF;
```

Крім цього буде встановлено перерахування результатів у текстовому режимі за допомогою таких виборів меню: Query> Results To> Results to Text.

Вправа 1.1

Створіть нову таблицю з назвою "T", що має три стовпці: *id* (ціле кількість, первинний ключ), *s* (рядок з літерними символами має довжину від 1 до 40 знаків), і *si* (у вигляді малого цілого числа):

```
CREATE TABLE T (
Id INT NOT NULL PRIMARY KEY,
```

```
S VARCHAR (40),
Si SMALLINT
);
```

Вставити кілька рядків у щойно створену таблицю:

```
INSERT INTO T (id, s) VALUES
(1, 'first'),
(2, 'second'),
(3, 'third');
SELECT * FROM T;
```

Команда "SELECT * FROM T" підтверджує, що три нових рядки були додані в таблицю (зверніть увагу на значення *NULL*, зареєстровані в колонці "*s*").

Виконайте таку команду:

```
BEGIN TRANSACTION;
INSERT INTO T (id, s) VALUES (4, 'fourth');
SELECT * FROM T;
ROLLBACK;
SELECT * FROM T;
```

Питання: порівняйте результати, отримані шляхом виконання зазначених двох команд "SELECT * FROM T".

Вправа 1.2

Далі слід виконати такі команди:

```
INSERT INTO T (id, s) VALUES (5, 'fifth');
ROLLBACK;
SELECT * FROM T;
```

Питання: який результат, отриманий після виконання наведеної команди "SELECT * FROM T"?

Який висновок можна зробити про наявність можливих обмежень у використанні команди "*BEGIN TRANSACTION*" в MS SQL Server?

Вправа 1.3

Деякі команди SQL належать до категорії *Data Definition Language* (*DDL* – мова опису даних). Інші належать до категорії *Data Manipulation Language* (*DML* – мова маніпулювання даними). Прикладами команд *DDL* є команди *CREATE TABLE, CREATE INDEX і DROP TABLE*, а прикладами *DML* є команди *SELECT, INSERT* та *DELETE*. Знаючи про це, далі варто досліджувати "глибину охоплення" команди *ROLLBACK*.

```
SET IMPLICIT_TRANSACTIONS ON;
INSERT INTO T (id, s) VALUES (9, 'will this be committed?');
CREATE TABLE T2 (id INT);
INSERT INTO T2 (id) VALUES (1);
SELECT * FROM T2;
ROLLBACK;
SELECT * FROM T;
SELECT * FROM T 3;
```

Питання:

Що сталося з T?

Що сталося з T2?

Що сталося з відсутньою таблицею T3?

Вправа 1.4

```
BEGIN TRANSACTION;
DELETE FROM T WHERE id> 1;
COMMIT;
SELECT * FROM T;
COMMIT;
```

Питання: прокоментуйте дії першого і другого *COMMIT*.

Вправа 1.5

Тепер настав час перевірити, чи викликає виникнення помилки автоматичний відкат (*ROLLBACK*) у MS SQL. Для цього слід виконати такі команди SQL:

```
BEGIN TRANSACTION;
INSERT INTO T (id, s) VALUES (2, 'Error test starts here');
```

```
SELECT (1/0) AS dummy FROM DUAL;
UPDATE T SET s = 'foo' WHERE id = 9999;
DELETE FROM T WHERE id = 7777;
INSERT INTO T (id, s) VALUES (2, 'Hi, I am a duplicate');
INSERT INTO T (id, s) VALUES (3, 'How about inserting too long of a string value?');
INSERT INTO T (id, s, si) VALUES (4, 'Smallint overflow for 32769?', 32769);
INSERT INTO T (id, s) VALUES (5, 'Is the transaction still active?');
SELECT * FROM T;
COMMIT;
DELETE FROM T WHERE id> 1;
SELECT * FROM T;
COMMIT;
```

Питання: прокоментуйте отримані результати.

Вправа 1.6

```
CREATE TABLE Accounts (
acctID INTEGER NOT NULL PRIMARY KEY,
balance INTEGER NOT NULL
CONSTRAINT unloanable_account CHECK (balance> = 0)
);
```

Наступна транзакція SQL призначена для перевірки перекладу суми в 500 євро з банківського рахунку номер 101 на неіснуючий номер банківського рахунку, що має, наприклад, acctID = 777:

```
BEGIN TRANSACTION;
UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 500 WHERE acctID = 777;
SELECT * FROM Accounts;
ROLLBACK;
```

Питання: чи виконуються дві команди UPDATE, незважаючи на те, що друга відповідає запиту поновлення неіснуючого рахунку в таблиці рахунків?

Якби команда *ROLLBACK* у прикладі транзакції B або C була замінена на COMMIT, стала б відповідна транзакція успішною, зробивши свій вплив на базу даних постійним?

Частина 2. Експерименти з паралельними транзакціями

Для експерименту з паралельними транзакціями відкрити два вікна з паралельними запитами SQL і сесіями "Клієнт A" і "Клієнт B", що мають доступ до тієї ж бази даних *TestDB*. Для обох сесій обрано перерахування результатів у текстовому режимі за допомогою таких виборів меню: Query> Results To> Results to Text.

В обох сесіях встановити використання неявних транзакцій:

```
SET IMPLICIT_TRANSACTIONS ON;
```

Для максимальної зручності відображення паралельних запитів SQL у *Management Studio*, треба розташувати одне вікно поруч з іншим вертикально. Для цього треба натисканням правої кнопки мишки або в самому вікні, або на назві вікна *SQLQuery* викликати контекстне меню і вибрати там *New Vertical Tab Group* (рис. 4.1). Нехай сесія у лівому вікні терміналу відповідають сесії A, сесія в правому вікні терміналу відповідають сесії B.

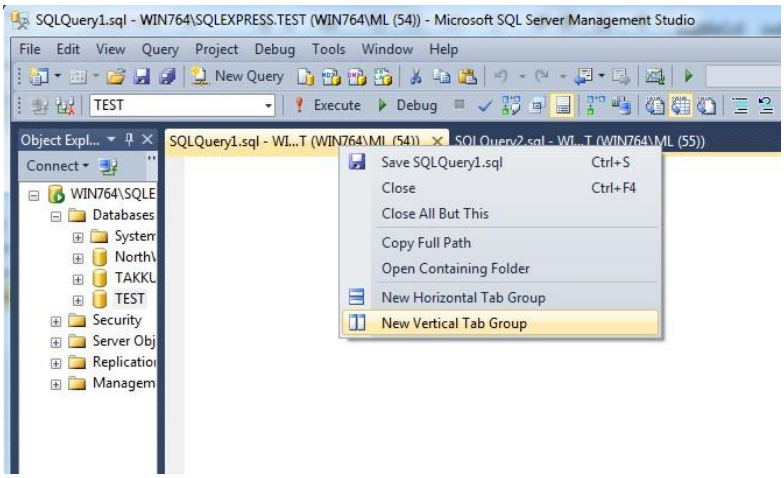


Рис. 4.1. **Розташування двох вікон SQL-запитів поруч один з одним**

Як можна бачити, паралельні транзакції, які отримують доступ до тих же самих даних, можуть блокувати інші. Таким чином, транзакції повинні бути розроблені так, щоб бути якомога більш короткими, тільки

для виконання необхідної роботи. Включення діалогу з кінцевим користувачем в логіці транзакції SQL призведе до катастрофічного часу очікування у виробничому середовищі. Тому необхідно, щоб ніякі транзакції SQL не передавали управління призначеному для користувача рівнем інтерфейсу перш, ніж транзакція закінчиться.

Для забезпечення безпеки таблиця акантів видаляється і створюється заново (ініціалізується) для реєстрації двох рядків даних:

```
DROP TABLE Accounts;
CREATE TABLE Accounts (
acctID INTEGER NOT NULL PRIMARY KEY,
balance INTEGER NOT NULL,
CONSTRAINT remains_nonnegative CHECK (balance> = 0)
);
BEGIN TRANSACTION;
INSERT INTO Accounts (acctID, balance) VALUES (101,1000);
INSERT INTO Accounts (acctID, balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;
```

Як було зазначено вище, проблема втрачених оновлень на увазі перезапис тільки що оновленого значення рядка одній (паралельній) транзакції до закінчення даної транзакції. Будь-який сучасний продукт СУБД з сервісами управління паралелізмом запобігає цю проблему, тому її відтворення у тестах неможливо. Однак після фіксації транзакції будь-яка необережна паралельна транзакція може переписати результати без першого зчитування поточного зафіксованого стану. У цьому випадку маємо справу з записом "брудних даних" (*Dirty Write*), відтворення яких може виявитися навіть занадто простим.

Ситуація з записом "брудних даних" виникає, наприклад, коли додаток зчитує значення в базі даних, оновлює значення в пам'яті, а потім записує оновлене значення в базу даних. У наступній таблиці змоделювати частину програми, використовуючи місцеві змінні в MS SQL. Місцева змінна – це просто іменний заповнювач у робочій пам'яті поточної сесії. Він ідентифікований символом @, доданим до початку його імені,

і він може бути включений у синтаксис операторів SQL, за умови, що він завжди приймає єдине скалярне значення.

Вправа 2.1

Табл. 4.1 пропонує проекспериментувати з паралельними сесіями A і B (кожна в окремому стовпці). Порядок операторів SQL для виконання цієї вправи відзначений у першому стовпці. Мета полягає в тому, щоб змоделювати ситуацію втрачених оновлень.

Транзакцією в сесії A є списання € 200 з рахунку 101 і транзакцією в сесії B є списання € 500 з того ж рахунку, що призведе до перезапису значення балансу рахунку (втрата інформації транзакцією A про зняття € 200). Таким чином, результат з записом "брудних даних" буде такою ж, як якщо б це був результат втрачених оновлень.

Таблиця 4.1

Проблема записи "брудних даних", моделювання додатка проведено за допомогою місцевих змінних

№ з/п	Session A	Session B
1	2	3
1	BEGIN TRANSACTION ; SET TRANSACTION ISOLATION LEVEL READ COMMITTED ; DECLARE @ amountA INT , @ balanceA INT ; SET @ amountA = 200 ; SELECT @ balanceA = balance FROM Accounts WHERE acctID = 101 ; SET @ balanceA = @ balanceA - @ amountA ; SELECT @ balanceA ;	
2		BEGIN TRANSACTION ; SET TRANSACTION ISOLATION LEVEL READ COMMITTED ; DECLARE @ amountB INT , @ balanceB INT ; SET @ amountB = 500 ;

Закінчення табл. 4.1

1	2	3
		SELECT @ balanceB = balance FROM Accounts WHERE acctID = 101 ; SET @ balanceB = @ balanceB - @ amountB ; SELECT @ balanceB ;
3	UPDATE Accounts SET balance = @ balanceA WHERE acctID = 101 ;	
4	WAITFOR DELAY '00: 00: 10 ' ;	UPDATE Accounts SET balance = @ balanceB WHERE acctID = 101 ;
5	SELECT acctID , balance FROM Accounts WHERE acctID = 101 ; COMMIT ;	
6		SELECT acctID , balance FROM Accounts WHERE acctID = 101 ; COMMIT ;

На четвертому етапі тривалість блокування у MS SQL дорівнює 10 секундам. Таким чином, транзакція (клієнт) А повинні перейти до п'ятого етапу без затримки, відразу після четвертого етапу транзакції (клієнта) В.

З огляду на паралельне виконання транзакцій А і В у табл. 4.1, клієнт А готується вивести € 200 з банківського рахунку на першому етапі (без оновлення бази даних). Клієнт В готується вивести € 500 на другому етапі. А обновляє базу даних на третьому кроці. А перевіряє баланс банківського рахунку з метою переконатися в його відповідності очікуваному значенню перед фіксацією на п'ятому етапі. Зробить те ж саме на шостому етапі.

Питання: чи було поведінку системи очікуваним?
Чи є в цьому випадку докази присутності втрачених даних?

Примітка: Усі продукти СУБД здійснюють управління паралелізмом, тому ні в якому рівні ізольованості аномалія втрачених оновлень

ніколи не проявиться. Однак, завжди є ймовірність, що недбало написаний код додатка стане причиною появи "сліпий перезапису", наслідки якої будуть мати такі ж катастрофічні наслідки з аномалією поновлення. Це схоже на аномалію втраченого поновлення, що переходила в сценарій одночасно виконуваних операцій через чорний хід (*back door*).

Вправа 2.2

Повторіть вправу 2.1, але вже з використанням рівня ізольованості *SERIALIZABLE*.

Не варто забувати відновлювати зміст таблиці рахунків.

Питання: що станеться, якщо в обох транзакціях "*SERIALIZABLE*" буде замінений на "*REPEATABLE READ*"?

Вправа 2.3

Повторення вправи 2.2, на цей раз з використанням вразливих оновлень (*sensitive updates*) у сценарії *SELECT – UPDATE* без локальних змінних (табл. 4.2).

Спочатку відновити зміст таблиці рахунків.

Таблиця 4.2

Конкуренція *SELECT – UPDATE* з використанням вразливих оновлень

№ з/п	Session A	Session B
1	2	3
1	BEGIN TRANSACTION ; SET TRANSACTION ISOLATION LEVEL SERIALIZABLE ; SELECT balance FROM Accounts WHERE acctID = 101 ;	
2		BEGIN TRANSACTION ; SET TRANSACTION ISOLATION LEVEL SERIALIZABLE ;

Закінчення табл. 4.2

1	2	3
		SELECT balance FROM Accounts WHERE acctID = 101 ;
3	UPDATE Accounts SET balance = balance – 200 WHERE acctID = 101 ;	
4	WAITFOR DELAY '00: 00: 10 ' ;	UPDATE Accounts SET balance = balance – 500 WHERE acctID = 101 ;
5	SELECT acctID , balance FROM Accounts WHERE acctID = 101 ; COMMIT ;	
6		SELECT acctID , balance FROM Accounts WHERE acctID = 101 ; COMMIT ;

Вправа 2.4

Конкуренція *UPDATE – UPDATE* на двох ресурсах у різному порядку. Спочатку відновити зміст таблиці рахунків (табл. 4.3).

Таблиця 4.3

Сценарій *UPDATE – UPDATE*

№ з/п	Session A	Session B
1	2	3
1	BEGIN TRANSACTION ; SET TRANSACTION ISOLATION LEVEL READ COMMITTED ; UPDATE Accounts SET balance = balance – 100 WHERE acctID = 202 ;	
2		BEGIN TRANSACTION ; SET TRANSACTION ISOLATION

Закінчення табл. 4.3

1	2	3
		LEVEL READ COMMITTED ; UPDATE Accounts SET balance = balance – 200 WHERE acctID = 202 ;
3	UPDATE Accounts SET balance = balance + 100 WHERE acctID = 202 ;	
4	WAITFOR DELAY '00: 00: 10 ' ;	UPDATE Accounts SET balance = balance + 200 WHERE acctID = 202 ;
5	SELECT acctID , balance FROM Accounts WHERE acctID = 202 ; COMMIT ;	
6		SELECT acctID , balance FROM Accounts WHERE acctID = 202 ; COMMIT ;

Примітка: рівень ізольованості транзакції не відіграє ніякої ролі в цьому сценарії, але це хороша практика, щоб завжди визначати рівень ізоляції на початку кожної транзакції! Там можуть бути деякі приховані обробки, наприклад, перевірки за допомогою зовнішнього ключа і тригери, що використовують доступ зчитування. Насправді за конструкцію тригерів повинні нести відповідальність адміністратори баз даних.

Вправа 2.5

У продовження експериментів з аномаліями спробувати змоделювати виникнення ситуації зчитування "брудних даних". Транзакція А виконується в *REPEATABLE READ*, тоді як транзакція В налаштована для роботи в режимі рівня ізольованості *READ UNCOMMITTED*:

Спочатку відновити зміст таблиці рахунків (табл. 4.4).

Таблиця 4.4

Проблема зчитування "брудних" даних

№ з/п	Session A	Session B
1	BEGIN TRANSACTION ; SET TRANSACTION ISOLATION LEVEL REPEATABLE READ ; UPDATE Accounts SET balance = balance – 100 WHERE acctID = 101 ;	
2		BEGIN TRANSACTION ; SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED ;
3	UPDATE Accounts SET balance = balance + 100 WHERE acctID = 202 ;	
4	WAITFOR DELAY '00: 00: 10 ' ;	
5	ROLLBACK ; SELECT * FROM Accounts ;	
6		SELECT * FROM Accounts ; COMMIT ;

Питання: можна зазначити про надійність транзакції B?
Як можна вирішити цю проблему?

Вправа 2.6

Далі слід аномалія неповторюваного читання.
Спочатку відновимо зміст таблиці рахунків (табл. 4.5).

Таблиця 4.5

Проблема неповторюваного читання

№ з/п	Session A	Session B
1	2	3
1	BEGIN TRANSACTION ; SET TRANSACTION ISOLATION LEVEL READ COMMITTED ;	

Закінчення табл. 4.5

1	2	3
	SELECT * FROM Accounts WHERE balance > 500;	
2	WAITFOR DELAY '00: 00: 10 ' ;	BEGIN TRANSACTION ; UPDATE Accounts SET balance = balance – 500 WHERE acctID = 101 ;
		UPDATE Accounts SET balance = balance + 500 WHERE acctID = 202 ; SELECT * FROM Accounts ; COMMIT ;
3	SELECT * FROM Accounts WHERE balance > 500; COMMIT ;	

Питання: зчитує чи транзакція А в третьому кроці той же результат, що вона вважала в першому кроці?

Як щодо установки транзакції в рівень ізольованості *REPEATABLE READ*?

Вправа 2.7

Тепер зробити спробу відтворення класичної "вставки фантома".

Спочатку відновити початковий зміст таблиці рахунків (табл. 4.6).

Таблиця 4.6

Проблема "вставки фантома"

№ з/п	Session A	Session B
1	2	3
1	BEGIN TRANSACTION ; SET TRANSACTION ISOLATION LEVEL REPEATABLE READ ;	
2		BEGIN TRANSACTION ; INSERT INTO Accounts (acctID , balance) VALUES (301 , 3000);

Закінчення табл. 4.6

1	2	3
3	SELECT * FROM Accounts WHERE balance > 1000 ;	
4	WAITFOR DELAY '00: 00: 10 ' ;	INSERT INTO Accounts (acctID , balance) VALUES (302 , 3000); COMMIT ;
5	SELECT * FROM Accounts WHERE balance > 1000 ; COMMIT ;	

Питання: чи повинна транзакція B очікувати транзакцію A в кожному етапі?

Чи є недавно вставлені номери акаунтів 301 і 302 видимими в середовищі транзакції A?

Чи впливає це на результати четвертого етапу, якщо змінити порядок етапів 2 і 3?

Вивчити питання про попередження появи фантомів шляхом заміни рівня ізольованості *REPEATABLE READ* на *SERIALIZABLE*.

Вправа 2.8

Проблема втрачених оновлень виникає, коли деякі вставлені або оновлені першої транзакцією рядки будуть оновлені або видалені будь-якими іншими паралельними транзакції до завершення першої транзакції. Це можливо в файлових рішеннях "NoSQL", але сучасні СУБД не допустять виникнення цього явища. Проте, після фіксації першої транзакції, будь-яка транзакція, що конкурує, може перезаписати рядки зафіксованих транзакцій. Далі варто змоделювати сценарій втрати поновлення за допомогою рівня ізольованості READ COMMITTED, який не тримає S-блокування. Спочатку клієнт зчитує значення балансу, знімаючи S-блокування.

```
- 1. Client A starts
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT acctID , balance FROM Accounts WHERE acctID = 101;
```

- 2. Client B starts

```
SET NOCOUNT ON;
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT acctID , balance FROM Accounts WHERE acctID = 101;
```

- 3. Client A continues

```
UPDATE Accounts SET balance = 1000 - 200 WHERE acctID = 101;
```

- 4. Client B continues

```
UPDATE Accounts SET balance = 1000 - 500 WHERE acctID = 101;
```

- 5. Client A continues

```
SELECT acctID , balance FROM Accounts WHERE acctID = 101;
COMMIT;
```

- 6. Client B continues

```
SELECT acctID , balance FROM Accounts WHERE acctID = 101;
COMMIT;
```

Таким чином, кінцевий результат (рис. 4.2) є помилковим!

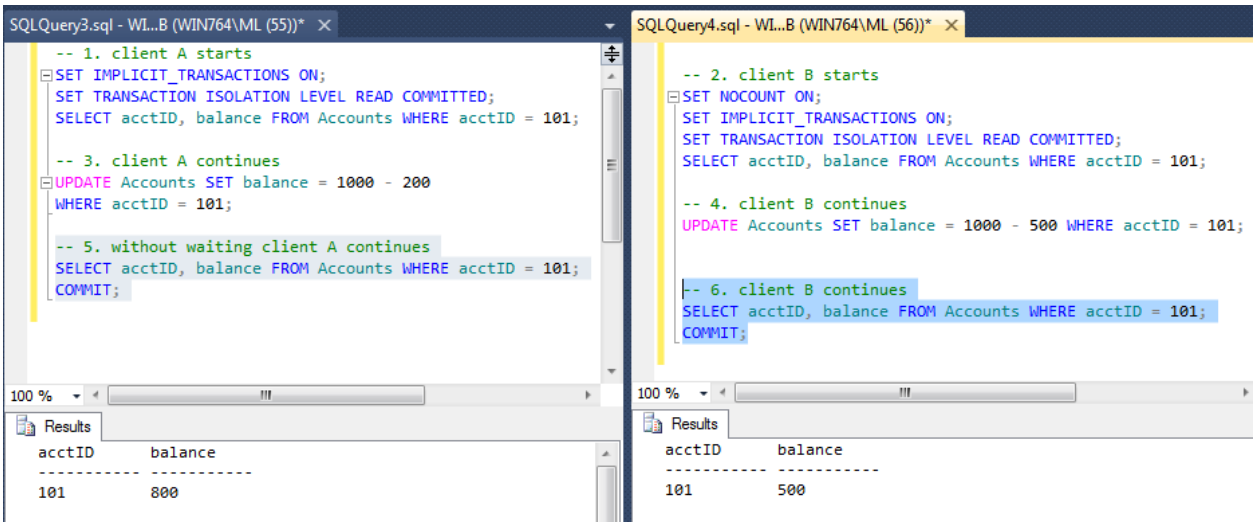


Рис. 4.2. **Результат виконання сценарію втрати поновлення**

Примітка: в цьому експерименті не було реальної проблеми "втраченого поновлення", але після того, як A фіксує свою транзакцію, B може

продовжити, в результаті чого переписує оновлення, виконане A. Називають цю проблему надійності "сліпий перезапис" (*Blind Overwriting*) або записом "брудних даних" (*Dirty Write*). Це може бути усунуто, якщо команди *UPDATE* використовують чутливі поновлення:

SET balance = balance – 500.

Вправа 2.9
"Проблема з загубленим оновленням" усунена блокуваннями

```
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts ( acctID , balance ) VALUES (101,1000);
INSERT INTO Accounts ( acctID , balance ) VALUES (202,2000);
COMMIT;

- 1. Client A starts
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT acctID , balance FROM Accounts WHERE acctID = 101;

- 2. Client B starts
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT acctID , balance FROM Accounts WHERE acctID = 101;

- 3. Client A continues
UPDATE Accounts SET balance = balance - 200 WHERE acctID = 101;

- 4. Client B continues without waiting for A
UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;

- 5. Client A continues
SELECT acctID , balance FROM Accounts WHERE acctID = 101;
COMMIT;
```

Вправа 2.10

Конкурування на двох ресурсах у різному порядку з використанням сценаріїв *UPDATE – UPDATE*.

Клієнт A переводить 100 євро з рахунку 101 на рахунок 202.

Клієнт B переводить 200 євро з рахунку 202 на рахунок 101.

Відновлення справжнього вмісту.

```
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts ( acctID , balance ) VALUES (101,1000);
INSERT INTO Accounts ( acctID , balance ) VALUES (202,2000);
COMMIT;
```

```
- 1. Client A starts
UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101;
```

```
- 2. Client B starts
SET IMPLICIT_TRANSACTIONS ON;
UPDATE Accounts SET balance = balance - 200 WHERE acctID = 202;
```

```
- 3. Client A continues
UPDATE Accounts SET balance = balance + 100 WHERE acctID =
= 202;
```

```
- 4. Client B continues
UPDATE Accounts SET balance = balance + 200 WHERE acctID =
= 101;
```

```
- 5. Client A continues if it can ...
COMMIT;
```

Далі у вправах 2.11 – 2.14 будемо експериментувати з аномаліями паралелізму, тобто ризиками надійності даних, які відомі за стандартом ISO SQL. Чи буде можливим ідентифікувати їх? І як можна усунути аномалії?

Вправа 2.11

Брудне читання?

Відновлення справжнього вмісту.

```
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts ( acctID , balance ) VALUES (101,1000);
INSERT INTO Accounts ( acctID , balance ) VALUES (202,2000);
COMMIT;
```

- 1. Client A starts

```
SET IMPLICIT_TRANSACTIONS ON;

UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 100 WHERE acctID = = 202;
```

- 2. Client B starts

```
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM Accounts; acctID balance
COMMIT;
```

- 3. Client A continues

```
ROLLBACK;
SELECT * FROM Accounts;
COMMIT;
```

Зробіть висновки за отриманими результатами.

Вправа 2.12

Є повторювання читання?

Під час аномалії повторювання читання деякі рядки, які читаються в поточній транзакції, можуть не відображатись в наборі результатів, якщо операція читання повторюватиметься до завершення угоди.

Відновлення справжнього вмісту.

```
SET IMPLICIT _ TRANSACTIONS ON;
DELETE FROM Accounts;
```

INSERT INTO Accounts (acctID , balance) VALUES (101,1000);
INSERT INTO Accounts (acctID , balance) VALUES (202,2000);
COMMIT;

- 1. Client A starts
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT * FROM Accounts WHERE balance> 500;

- 2. Client B starts
SET IMPLICIT_TRANSACTIONS ON;
UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 500 WHERE acctID = = 202;
COMMIT;

- 3. Client A continues
SELECT * FROM Accounts WHERE balance> 500;
COMMIT;

Питання: можна бачити як і раніше ті ж облікові записи, що і на етапі 1? Чому?

Вправа 2.13

Вставити фантом?

Вставити фантоми – це рядки, вставлені паралельними транзакціями і які можна побачити до завершення транзакції.

Відновлення справжнього вмісту.
SET IMPLICIT _ TRANSACTIONS ON ;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID , balance) VALUES (101,1000);
INSERT INTO Accounts (acctID , balance) VALUES (202,2000);
COMMIT;

- 1. Client A starts
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT * FROM Accounts WHERE balance> = 1 000;

- 2. Client B starts
SET IMPLICIT_TRANSACTIONS ON;
INSERT INTO Accounts (acctID , balance) VALUES (303,3000);
COMMIT;

- 3. Client A continues
SELECT * FROM Accounts WHERE balance> = 1 000;
COMMIT;

Питання: як можна запобігти появі фантомів?

Лабораторна робота 5
Створення додатка "Комп'ютерна фірма" для роботи з даними за допомогою ADO.NET

У процесі створення програми, яка працює з базами даних, необхідно виконати такі основні завдання, як визначення рядків підключення, вставка даних і виконання збережених процедур. У лабораторній роботі розглядаються питання взаємодії з базою даних з простого *Windows Forms* додатка за допомогою *Visual C #, Visual Basic* і ADO.NET.

Для створення програми будуть потрібні такі компоненти.

- Visual Studio.
- SQL Server Express LocalDB.

Якщо у вас немає *SQL Server Express LocalDB*, його можна встановити на сторінці завантаження *SQL Server Express*.

У процесі виконання лабораторної роботи ви познайомитеся з базовою функціональністю інтегрованого середовища розробки *Visual Studio* і зможете створювати *Windows Forms* додаток, додавати форми в проєкт, поміщати кнопки та інші елементи управління в форми, задавати властивості елементів управління і створювати код для простих подій.

Опис проєкту

Проєкт буде містити три форми (рис. 5.1):

1. Навігація (*Navigation*).
2. Продажі (*Sales*).
3. Склад (*Warehouse*).

Під час відкриття проєкту відкривається форма *Навігація*. З якої можна перейти в форму *Продажі* або *Склад*. Форми *Продажі* і *Склад* мають з'єднання з базою даних. Проєкт необхідно налаштувати так, щоб під час продажу товару в формі *Продажі* він автоматично списувався в базі даних.

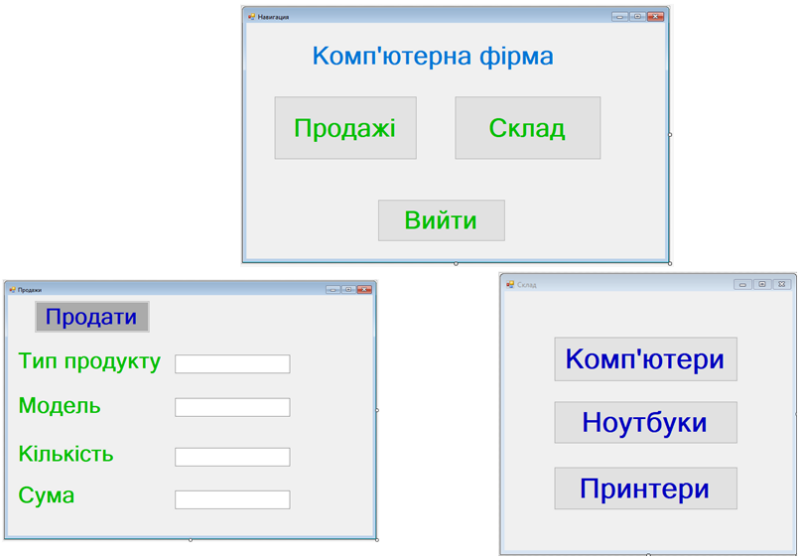


Рис. 5.1. **Форми проєкту "Комп'ютерна фірма"**

Створення проєкту

Запустіть *Visual Studio* 2019.

На початковому екрані обрати *Створити проєкт* (рис. 5.2).

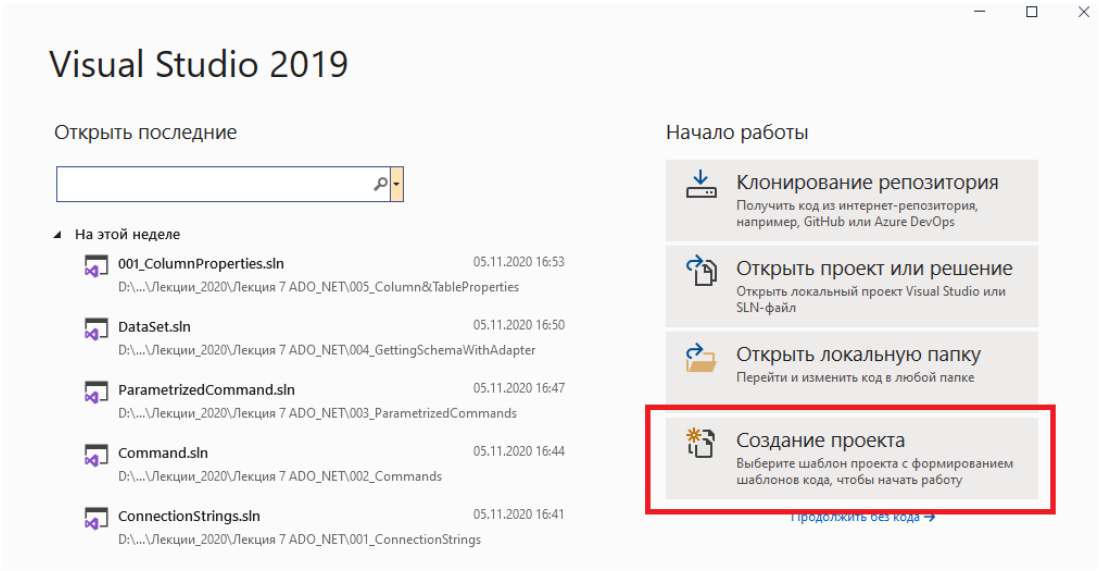


Рис. 5.2. **Створення проєкту у *Visual Studio***

У вікні *Створити проєкт* обрати шаблон *Додаток Windows Forms (.NET Framework) для C #* (рис. 5.3).

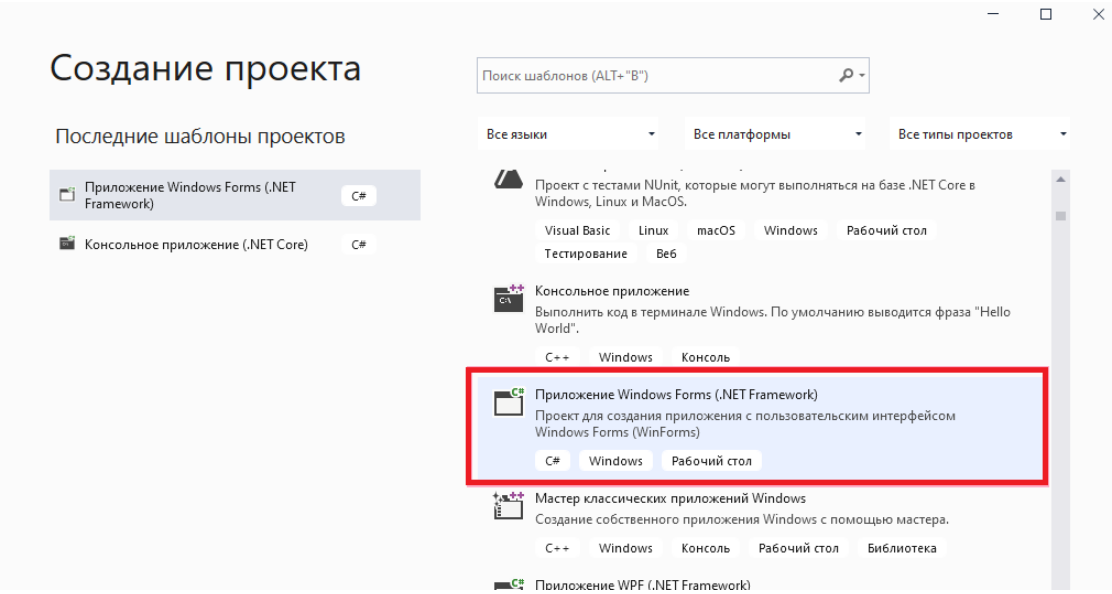


Рис. 5.3. **Виберіть шаблон *Додаток Windows Forms (.NET Framework) для C #***

У поле *Назва проєкту* вікна *Налаштувати новий проєкт* введіть *Comp_firm*. Потім натисніть *Створити* (рис. 5.4).

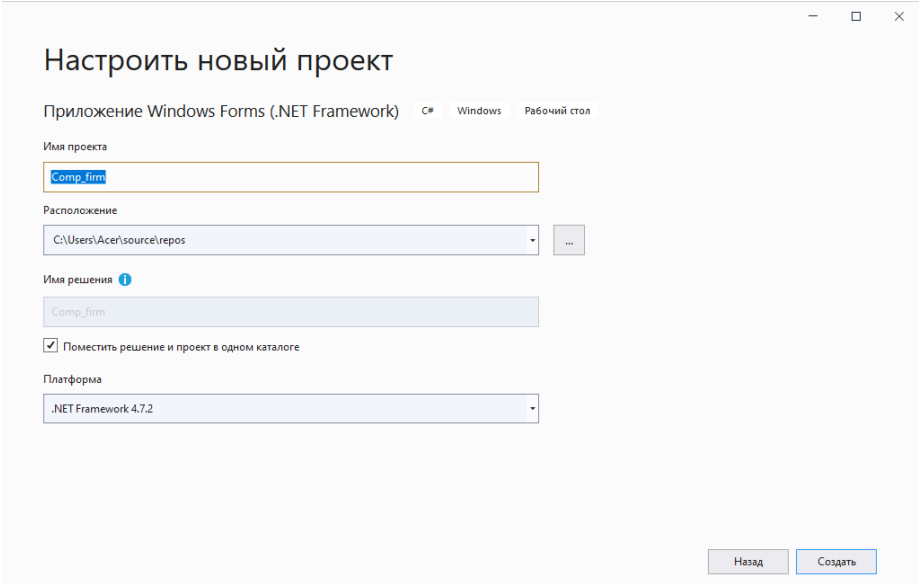


Рис. 5.4. **Налаштування нового проєкту**

Новий проєкт відкривається у *Visual Studio*.

Створення форми "Навігація"

Після вибору шаблону і задавання назви файла, *Visual Studio* відкриває форму. Форма є призначеним для користувача інтерфейсом *Windows*.

Для кожної форми необхідно додати текстові поля, кнопки та інші елементи управління, які відображаються на рисунках далі. Для кожного елемента управління задайте властивості, зазначені в таблицях.

Створимо першу форму "Навігація", додамо елементи управління на форму, а потім запустимо його (рис. 5.5).

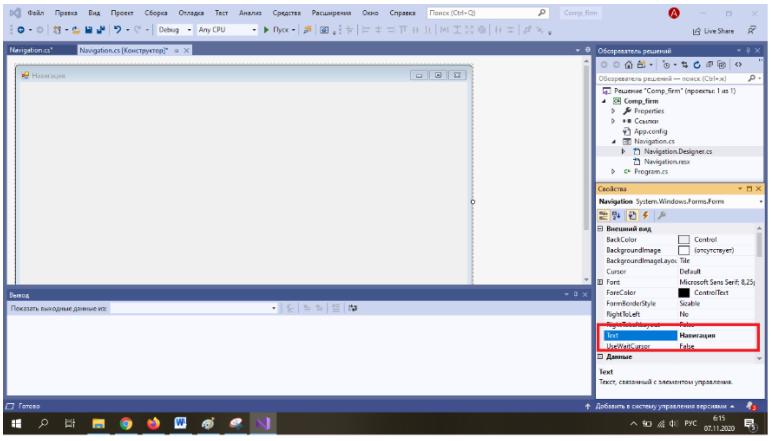


Рис. 5.5. **Створення першої форми "Навігація"**

Перейменуйте файл Form1.cs на Navigation.cs.

У вкладці властивості перейменуйте форму з "Form1" на "Навігація".

Додавання кнопки на форму (Рис. 5.6). Відкрийте *Елементи управління.* Для цього виберіть *Вид* > *Панель елементів* > *Стандартні елементи управління.*

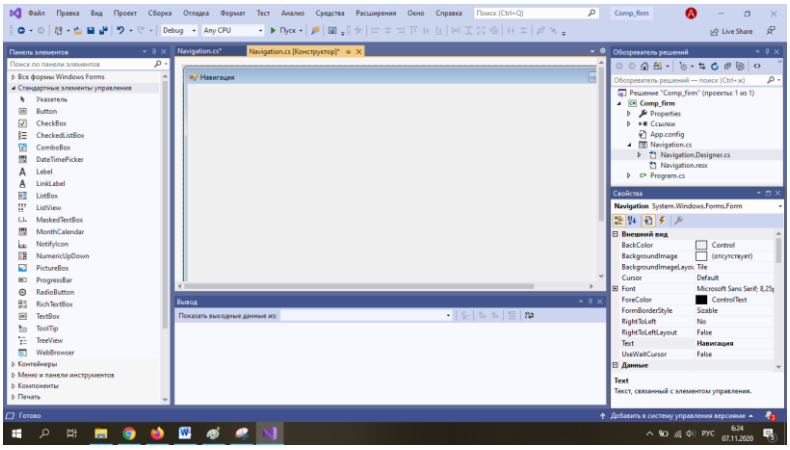


Рис. 5.6. **Додавання кнопки на форму**

Для створення напису виберіть елемент управління *Label* і перетягніть його на форму. У вікні *Властивості* змініть назва з *Label1* на *Комп'ютерна фірма*, а потім натисніть клавішу *ВВЕДЕННЯ*. Змініть колір і розмір шрифту на свій розсуд.

Обрати елемент управління *Button* і створіть кнопку *Продажі*. Також створіть кнопки: *Склад* і *Вийти*. Готова форма подана на рис. 5.7.

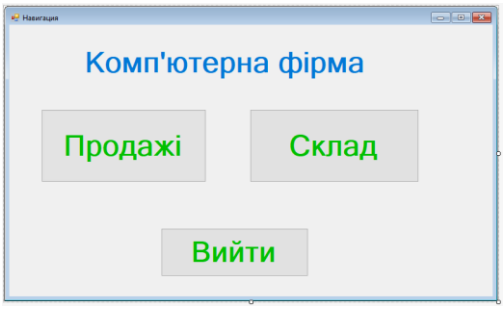


Рис. 5.7. **Зовнішній вигляд форми "Навігація"**

Для кожного елемента управління задати властивості, зазначені в таблиці.

Елементи управління	Властивості
кнопка *"Продажі"*	Name = btnCales
кнопка *"Склад"*	Name = btnWarenhouse
кнопка *"Вийти"*	Name = btnClose

Створення форми "Продажі" (Sales)

Для створення нової форми "Продажі" обрати *Проєкт* > *Додати форму* (рис. 5.8). У вікні, в графі *Назва* введіть назву форми Sales.cs.

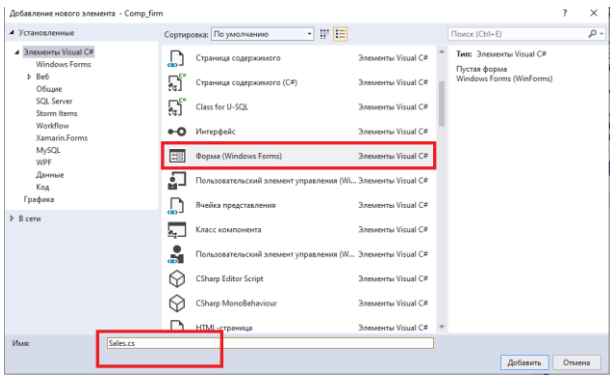


Рис. 5.8. **Створення форми "Продажі" (Sales)**

Створіть форму, зображену на рис. 5.9. Форма містить:

Кнопку – "Продати" (елемент управління *Button*);

4 написи – "Тип продукту", "Модель", "Кількість" і "Сума" (елемент управління *Label*);

4 віконця для введення тексту (елемент управління *TextBox*).

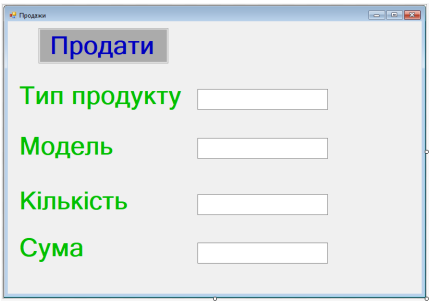


Рис. 5.9. **Форма "Продажі"**

Для кожного елемента управління задайте властивості, зазначені в таблиці.

Елементи управління	Властивості
Кнопка *"Продати"*	Name = btnCales_Cales
Віконце *"Тип продукту"*	Name = txtProduct
Віконце *"Модель"*	Name = txtModel
Віконце *"Кількість"*	Name = txtCount
Віконце *"Сума"*	Name = txtPrice

Створення форми "Склад" (Warehouse)

Таким же чином створіть третю форму "Склад" (рис. 5.10).

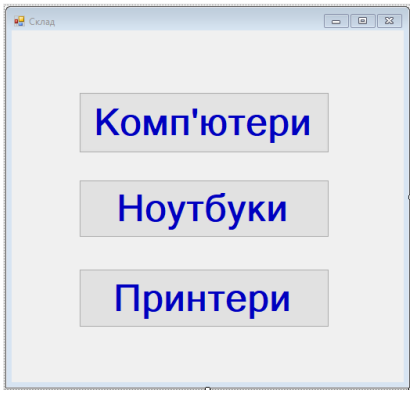


Рис. 5.10. **Форма "Склад"**

Для кожного елемента управління задайте властивості, зазначені в таблиці.

Елементи управління	Властивості
Кнопка *"Комп'ютери"*	Name = btnPC
Кнопка *"Ноутбуки"*	Name = btnLaptop
Кнопка *"Принтери"*	Name = btnPrinter

Додавання коду на форму "Навігація"

Форма "Навігація" відкривається під час запуску додатка. Кнопка *Продажі* відкриває форму *Sales*. Кнопка *Склад* відкриває форму *Warehouse*. Кнопка *Вихід* закриває додаток.

Перетворення форми навігації в початкову форму

Відкрийте файл **Program.cs** і змініть рядок Application.Run на таку: Application.Run(new Navigation()).

Створення обробників подій

Двічі клацніть три кнопки в формі навігації, щоб створити порожні методи обробника подій. Під час подвійного натискання кнопки також додається автоматично створений код у файл коду конструктора, який дозволяє натиснути кнопку для виклику події.

Додавання коду для логіки форми "Навігація"

На сторінці коду для форми "Навігація" заповните основні тексти методів для трьох обробників подій натискання кнопки, як показано в наступному коді.

```
using System;
using System.Windows.Forms;

namespace Comp_firm
{
public partial class Navigation : Form
{
public Navigation ()
{
```

```
InitializeComponent ( );
}

private void Navigation_Load ( object sender, EventArgs e)
{

}

// Кнопка ПРОДАЖУ
private void btnCales_Click (object sender, EventArgs e)
{
Form frm = new Sales ( );
frm.Show ( );
}

// Кнопка СКЛАД
private void btnWarenhouse_Click (object sender, EventArgs e)
{
Form frm = new Warehouse ( );
frm.Show ( );
}

// Кнопка ВИЙТИ
private void btnClose_Click (object sender, EventArgs e)
{
this.Close ();
}
}
}
```

Запуск додатка

Відкрити програму, перевірити, що з форми "Навігація" можна відкрити дві інші форми, а також вийти з програми.

Форми "Продажі" і "Склад" необхідно підключити до бази даних. Ці питання будуть розглянуті в наступній лабораторній роботі.

Лабораторна робота 6
Підключення додатка "Комп'ютерна фірма" до бази даних за допомогою ADO.NET

На попередній лабораторній роботі були створені форми: "Навігація" (*Navigation*), "Продажі" (*Sales*) і "Склад" (*Warehouse*). Тепер необхідно підключити створені форми до нашої бази даних "Комп'ютерна фірма" (Komp_firm).

Починати слід з форми *Склад* (*Warehouse*) (рис. 6.1).

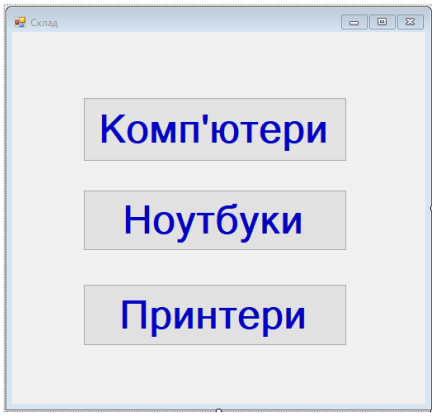


Рис. 6.1. **Форма "Склад"**

Двічі натиснути мишкою на кожній кнопці і переходимо до файла Warehouse.cs. Підключити обробник подій, як зазначено в наведеному далі коді:

```
using System;
using System.Windows.Forms;

namespace Comp_firm
{
public partial class Warehouse : Form
{
public Warehouse()
{
InitializeComponent();
}
```

```
// Кнопка "Комп'ютери"
private void btnPC_Click (object sender, EventArgs e)
{
Form frm = new PC ();
frm.Show ();
}

// Кнопка "Ноутбуки"
private void btnLaptop_Click (object sender, EventArgs e)
{
Form frm = new Laptop ();
frm.Show();
}
// Кнопка "Принтери"
private void btnPrinter_Click (object sender, EventArgs e)
{
Form frm = new Printer ();
frm.Show();
} } }
```

Проведемо попередні налаштування. Перш за все, створити новий клас з назвою TbaleExtensoinClass (рис. 6.2).

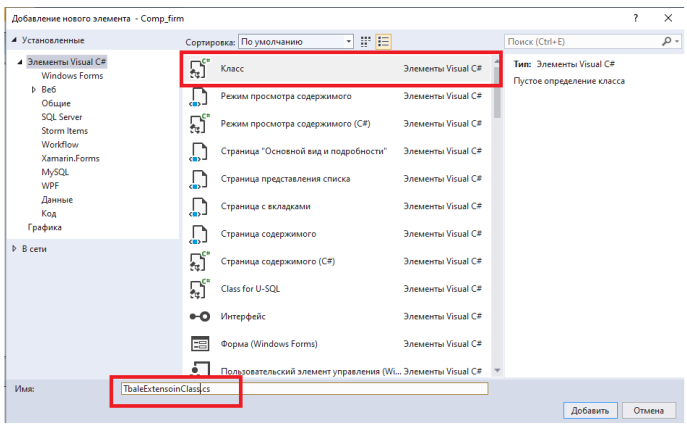


Рис. 6.2. **Створення нового класа TbaleExtensoinClass**

У цьому класі варто розмістити метод, який буде записувати дані в таблицю на основі *DataReader*. Для завантаження в таблицю *DataTable* рядків з джерела даних буде використовуватися метод *Load*.

Метод *CreateSchemaFromReader* створює такі обмеження на стовпці таблиці на основі отриманого об'єкта *DataReader*:

1. *Readonly* – повертає або задає значення, яке вказує на допустимість зміни стовпчика після додавання рядка в таблицю.

2. *AllowDBNull* – повертає або задає значення, яке вказує на допустимість нульових значень в цьому стовпці для рядків, що належать таблиці.

3. *MaxLength* – повертає або задає максимальну довжину текстового стовпчика.

4. *Unique* – повертає або задає значення, що показує, чи повинні значення в кожному рядку стовпця бути унікальними.

Заповніть сторінку коду як зазначено далі:

```
using System;
using System.Data.SqlClient;
using System.Data;

namespace Comp_firm
{
static class TbaleExtensoinClass
{
//цей метод записує дані в таблицю на основі DataReader
public static void LoadWithSchema (this DataTable table, SqlDataReader reader)
{
table.CreateSchemaFromReader (reader);

table.Load (reader); //Метод Load використовується для завантаження в таблицю DataTable рядків з джерела даних.
}

// цей метод створює обмеження на стовпці таблиці на основі отриманого об'єкта DataReader :
private static void CreateSchemaFromReader (this DataTable table, SqlDataReader reader)
{
```

DataTable schemaTable = reader.GetSchemaTable (); // Метод повертає таблицю яка описує метадані стовпці об'єкта *SqlDataReader*.

foreach (DataRow schemaRow in schemaTable.Rows)
{
DataColumn column = new DataColumn ((string) schemaRow ["ColumnName"]); //створення стовпця з назвам стовпця в джерелі даних
column.AllowDBNull = (bool) schemaRow ["AllowDbNull"]; //отримання значення властивості *AllowDBNull*
column.DataType = (Type) schemaRow ["DataType"]; //отримання значення властивості *DataType*
column.Unique = (bool) schemaRow ["IsUnique"]; //отримання значення властивості *Unique*
column.ReadOnly = (bool) schemaRow ["IsReadOnly"]; //отримання значення властивості *Readonly*
column.AutoIncrement = (bool) schemaRow ["IsIdentity"]; //отримання значення властивості *AutoIncrement*

if (column.DataType == typeof (string)) //якщо поле типу *string*
column.MaxLength = (int) schemaRow ["ColumnSize"]; //отримати значення властивості *MaxLength*

if (column.AutoIncrement == true) // Якщо поле з автоінкрементом
{Column.AutoIncrementStep = -1; column.AutoIncrementSeed = 0;} //задати властивості AutoIncrementStep і AutoIncrementSeed

table.Columns.Add (column);
} } }}

Створити форму "Комп'ютери".

Таким же чином, як було створено форми на попередній лабораторній роботі, створюємо нову форму *"Комп'ютери"* (PC.cs).

Відкрити панель елементів і розміщувати на формі елемент *DataGridView* (рис. 6.3).

У *Microsoft Visual Studio* елемент управління розроблений для використання в додатках, створених за шаблоном *Windows Forms Application*.

Даний елемент управління дозволяє організовувати дані у вигляді таблиці. Дані можуть бути отримані з бази даних, колекції, внутрішніх змінних – масивів або інших об'єктів програми.

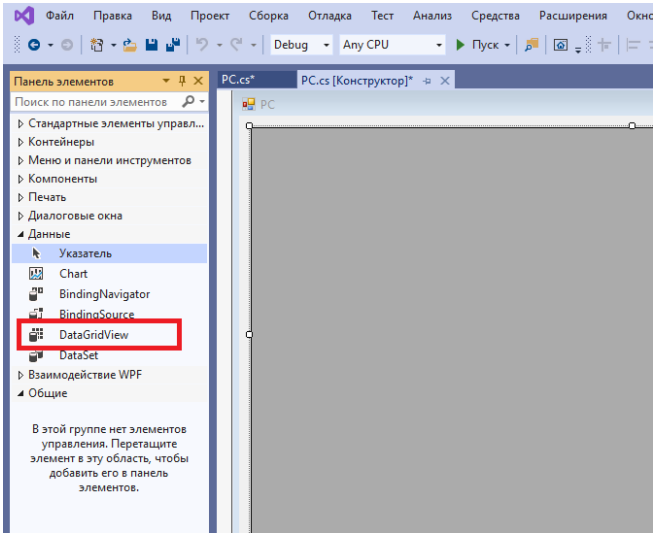


Рис. 6.3. **Розміщення на формі елемента *DataGridView***

Після вставки елемента *DataGridView* його необхідно підключити до бази даних. Для цього натиснути на трикутник у верхньому правому куточку елемента, щоб відкрити вікно *DataGridView "Завдання"* (рис. 6.4).

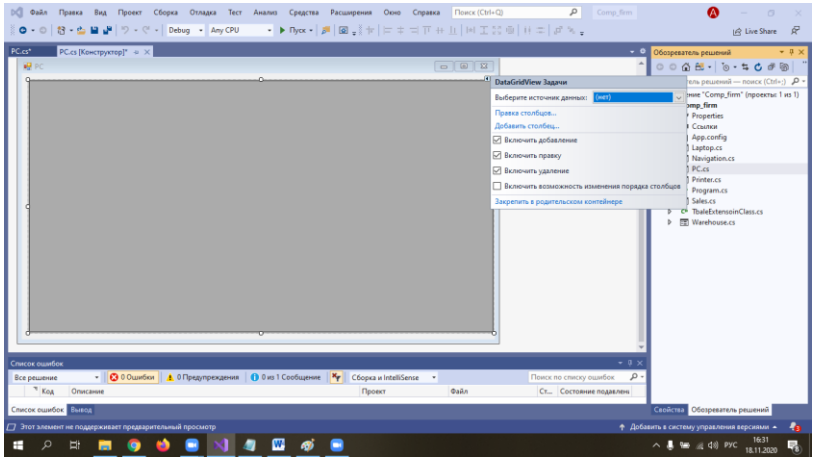


Рис. 6.4. **Підключення до бази даних елемента *DataGridView***

У рядку "Оберіть джерело даних" обрати пункт *"Додати джерело даних проєкту"* (рис. 6.5).

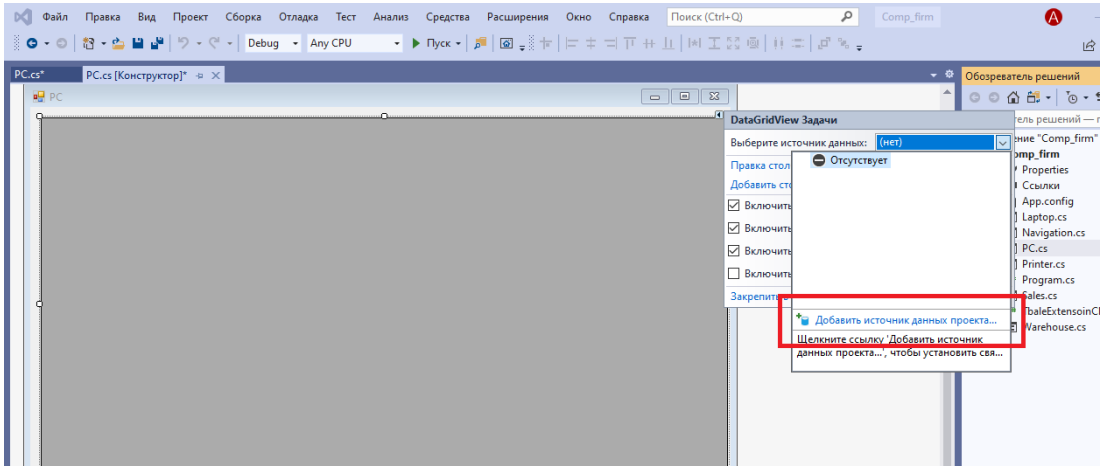


Рис. 6.5. **Вибір пункту *"Додати джерело даних проєкту"***

Подальший шлях: *База даних -> Набір даних -> Створити підключення.* В огляді обрати свою базу даних.

У вікні *"Вибір об'єктів бази даних"* встановити галочку біля таблиці PC (рис. 6.6).

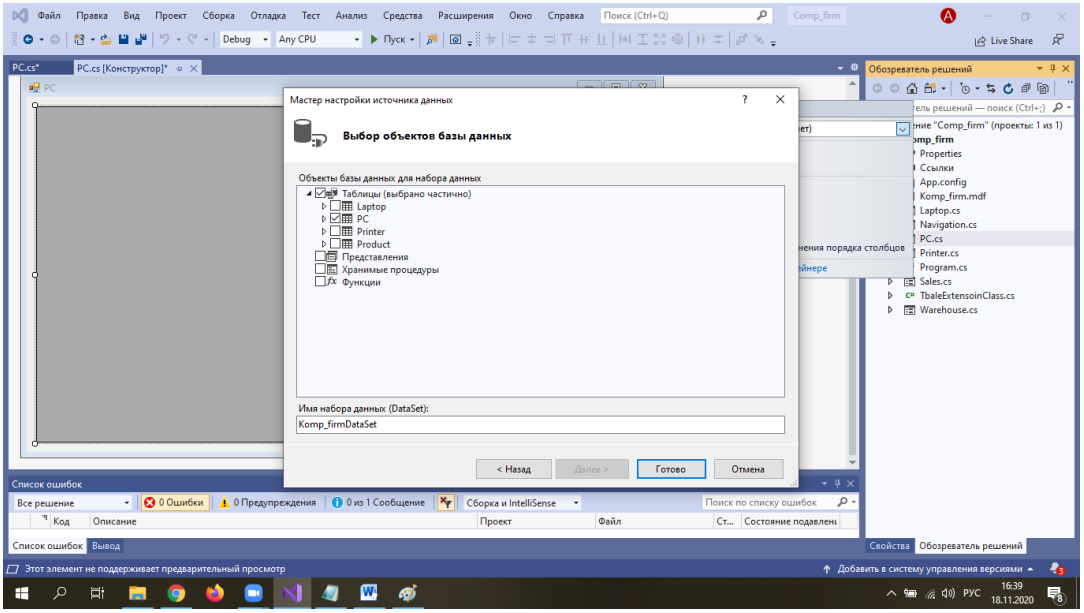


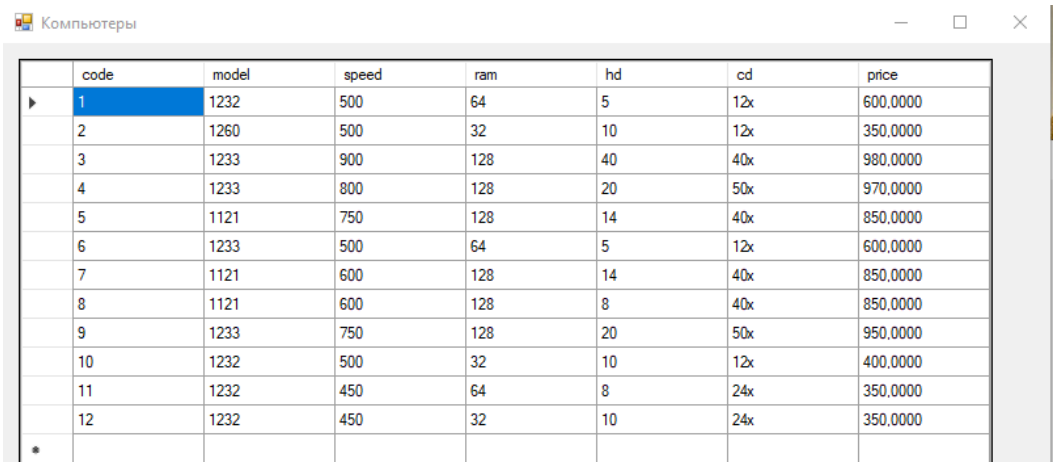
Рис. 6.6. **Налаштування вибору *об'єктів бази даних***

Аналогічним чином створити форми *"Ноутбуки"* та *"Принтери"*.

Тестування додатка

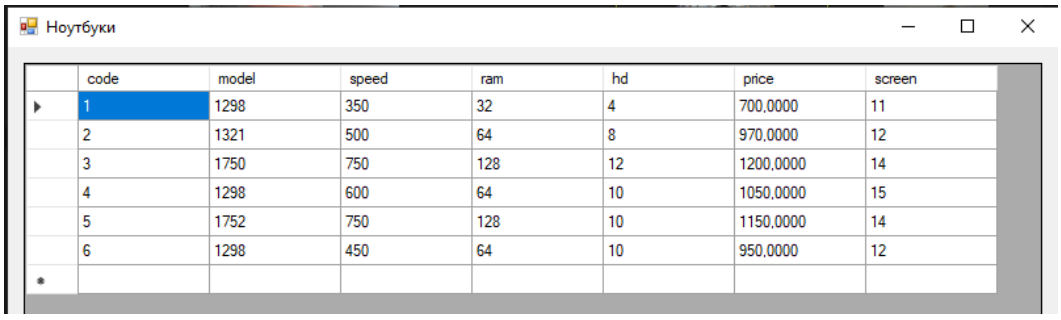
Натисніть **F5** для збірки і тестування програми після написання коду для кожного обробника події натискання кнопки і загального коду

програми. Зовнішній вигляд вікон, що відкриваються, наведено далі (рис. 6.7 – 6.9).



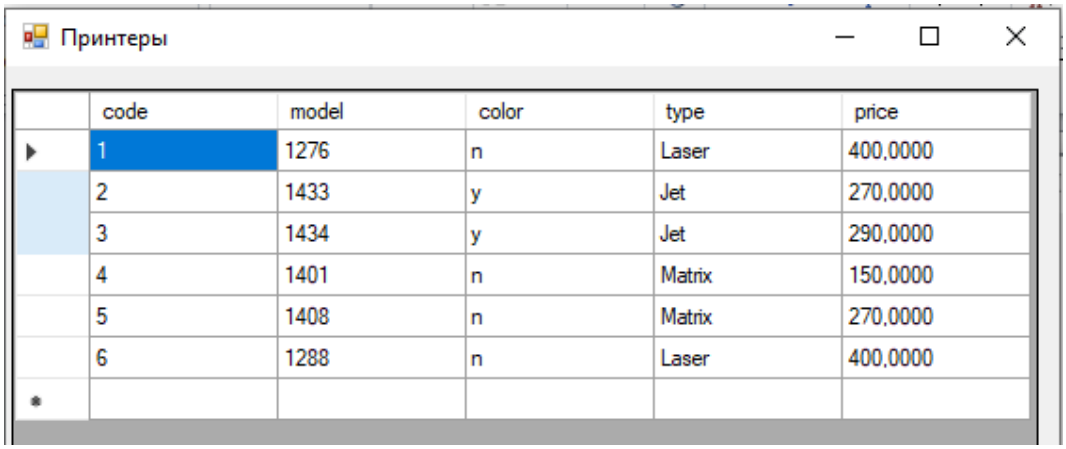
code	model	speed	ram	hd	cd	price
1	1232	500	64	5	12x	600,0000
2	1260	500	32	10	12x	350,0000
3	1233	900	128	40	40x	980,0000
4	1233	800	128	20	50x	970,0000
5	1121	750	128	14	40x	850,0000
6	1233	500	64	5	12x	600,0000
7	1121	600	128	14	40x	850,0000
8	1121	600	128	8	40x	850,0000
9	1233	750	128	20	50x	950,0000
10	1232	500	32	10	12x	400,0000
11	1232	450	64	8	24x	350,0000
12	1232	450	32	10	24x	350,0000

Рис. 6.7. **Вікно "Комп'ютери"**



code	model	speed	ram	hd	price	screen
1	1298	350	32	4	700,0000	11
2	1321	500	64	8	970,0000	12
3	1750	750	128	12	1200,0000	14
4	1298	600	64	10	1050,0000	15
5	1752	750	128	10	1150,0000	14
6	1298	450	64	10	950,0000	12

Рис. 6.8. **Вікно "Ноутбуки"**



code	model	color	type	price
1	1276	n	Laser	400,0000
2	1433	y	Jet	270,0000
3	1434	y	Jet	290,0000
4	1401	n	Matrix	150,0000
5	1408	n	Matrix	270,0000
6	1288	n	Laser	400,0000

Рис. 6.9. **Вікно "Принтери"**

Форму *"Продажі"* розробити самостійно (рис. 6.10).

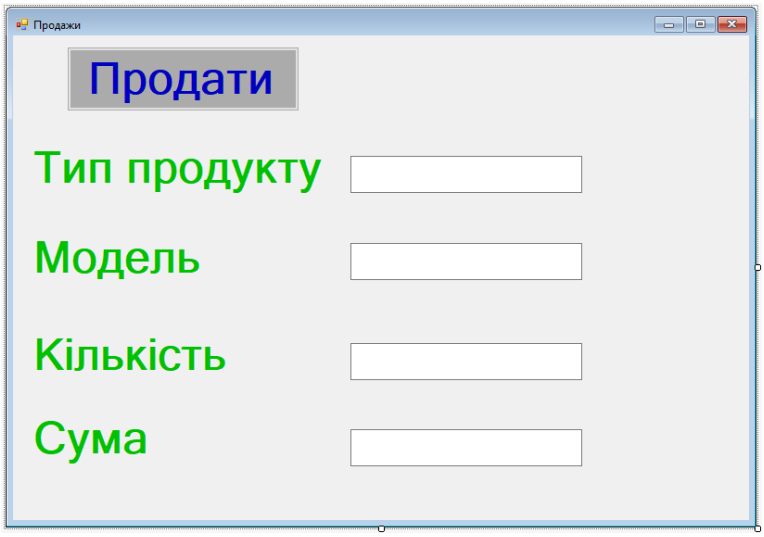


Рис. 6.10. **Вікно "Продажі"**

У процесі натискання на кнопку *"Продати"* кількість товарів у базі даних має змінитися.

Рекомендована література

Основна

1. Дейт К. Дж. Введение в системы баз данных / К. Дж. Дейт. – 8-е изд. – Киев : Диалектика, 2020. – 1328 с.

2. Бакаев А. А. Методы организации и обработки баз знаний / А. А. Бакаев. – Київ : Наукова думка, 2018. – 148 с.

3. ДСТУ 2874-94. Бази даних. Терміни та визначення. – Київ : Держстандарт України, 1995. – 32 с.

4. ДСТУ 2940-94. Системи оброблення інформації. Керування процесами оброблення даних. Терміни та визначення. – Київ : Держстандарт України, 1995. – 28 с.

5. Оньон Ф. Основы ASP.NET с примерами на C# /Ф. Оньон. – Киев : Диалектика, 2019. – 304 с.

Додаткова

6. Microsoft Corporation. Разработка Web-приложений на Microsoft Visual Basic .NET и Microsoft Visual C# .NET : Учебный курс MCAD/ MCSD. – Москва : Издательско-торговый дом "Русская Редакция", 2018. – 704 с.

7. Microsoft Corporation. Разработка Web-сервисов XML и серверных компонентов на Microsoft Visual Basic .NET и Microsoft Visual C# .NET : Учебный курс MCAD/MCSD. – Москва : Издательско-торговый дом "Русская Редакция", 2018. – 576 с.

Інформаційні ресурси

8. Вендров А. М. CASE-технологии. Современные методы и средства проектирования информационных систем [Электронный ресурс] / А. М. Вендров. – Режим доступа : http://www.citforum.ru/database/case/index.shtml.

Зміст

НАВЧАЛЬНЕ ВИДАННЯ

ПРОЄКТУВАННЯ БАЗ ДАНИХ ТА БАЗ ЗНАНЬ

Методичні рекомендації до лабораторних робіт для студентів спеціальності 186 "Видавництво та поліграфія" першого (бакалаврського) рівня

Самостійне електронне текстове мережеве видання

Укладач **Гордєєв** Андрій Сергійович

Відповідальний за видання *О. І. Пушкар*

Редактор *В. О. Дмитрієва*

Коректор *В. Ю. Труш*

План 2022 р. Поз. № 107 ЕВ. Обсяг 73 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.