Навчально-методична література

Міністерство освіти і науки України
Тернопільський національний технічний університет
ім. Івана Пулюя
Кафедра комп'ютерно-інтегрованих технологій

Методичні вказівки до виконання лабораторних робіт

з дисципліни

«Системи управління базами даних»

напряму підготовки 151«Автоматизація та комп'ютерно-інтегровані технології»

Тернопіль – 2022

УДК 681.3
ББК 32.973

Укладачі:
Чихіра І.В., канд. техн. наук, доцент,
Дідич І.С., Ph.D, асистент

Рецензент
Коноваленко І.В., канд. техн. наук, доцент

Схвалено та рекомендовано до друку на засіданні кафедри протокол №1 від 29.08.2022р.

Засідання факультету протокол №1 від 30.08.2022р.

Відповідальний за випуск *Чихіра І.В.*, канд. техн. наук, доцент,

ЛАБОРАТОРНА РОБОТА №1

Тема: Побудова таблиць у СУБД Microsoft Access

1. Запустити ***СУБД Ms Access*** у вікні, що відкрилося, вибрати перемикач ***"Новая база данных"***, після натиснення кнопки **"ОК"** відкривається вікно ***"Файл новой базы данных"*** (рис. 1), у якому потрібно задати ім'я файла бази даних. База даних повинна мати ім'я ***sam_№ комп'ютера*** (наприклад, sam_05 - ім'я файла бази даних, створеного за комп'ютером з номером 5).

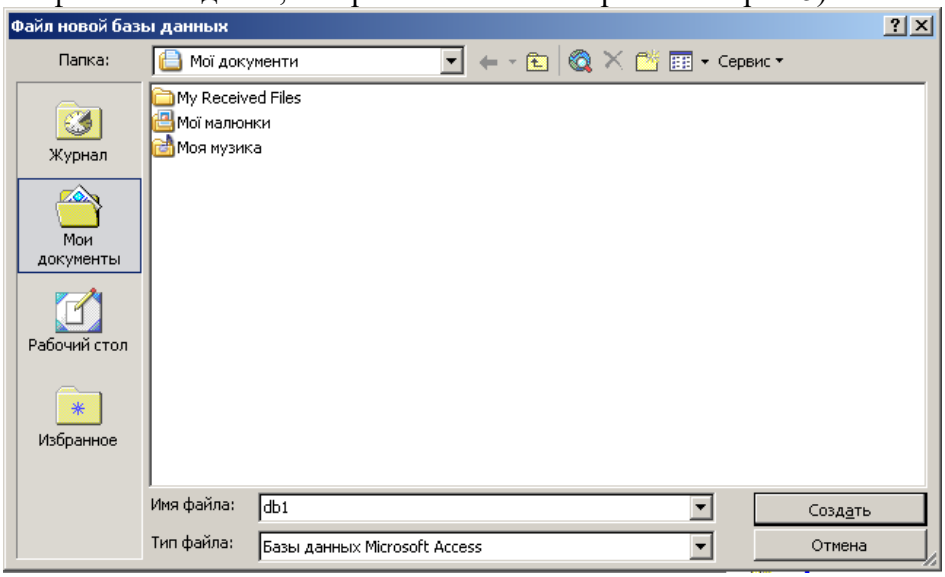


Рис. 1. Вікно "Файл новой базы данных"

Після натискання кнопки ***"Создать"*** на екрані з'явиться основне вікно бази даних (рис. 2).

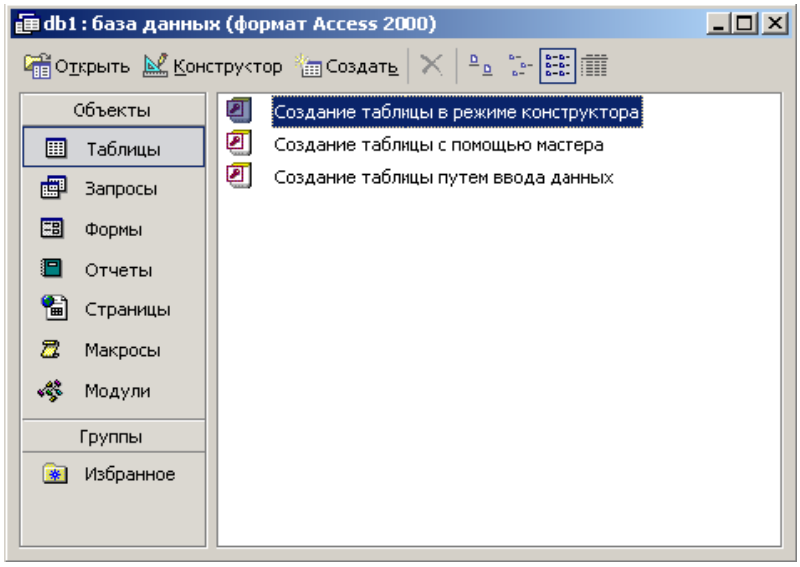


Рис. 2. Основне вікно бази даних

2. Для створення таблиці на вкладці ***"Таблицы"*** потрібно натиснути кнопку ***"Создать"*** (рис. 3) і обрати режим конструктора створення бази даних.

3. У режимі конструктора потрібно задати імена полів таблиці і зі списку вибирати типи даних поля. Для полів, що містять дробові значення, потрібно вибрати значення *"с плавающей точкой"* властивості *"размер поля"* (рис. 4). Для полів, що набувають значень "-t-" або ", чи "так або "ні" вибирається тип даних *"логічний". Для* поля ***"Стать"*** потрібно задати значення "ч" або "ж" властивості *"условие на значение".* Ця властивість означає, що у дане поле можуть бути введені значення лише "ч" або "ж" і ніякі інші.

4. За умовою завдання поле ***"Освіта"*** потрібно заповнити з використанням майстра підстановок. Для цього у списку типів поля вибирають *"Мастер подстановок".* На першому кроці

створення підстановки потрібно вибрати перемикач *"будет введен фиксированный набор значений"* (рис. 5). Для переходу до наступного кроку натискаємо кнопку *"Далее "*.

5. Далі вводиться фіксований набір даних. Для цього потрібно поставити "мишу" у полі ***"Столбец"*** і ввести значення списку підстановок (рис. 6).

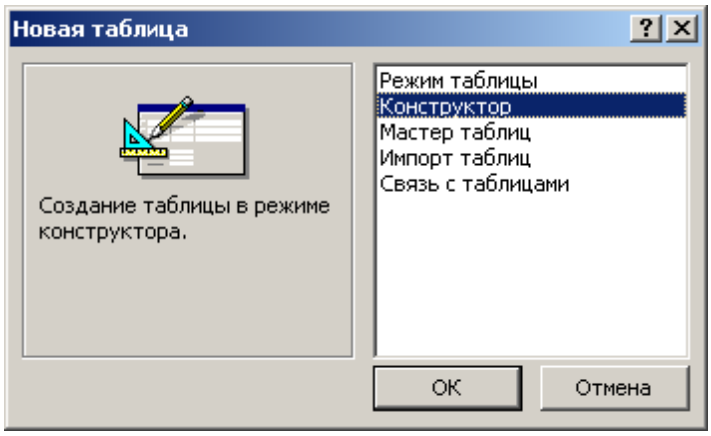


Рис. 3. Вибір режиму створення нової таблиці

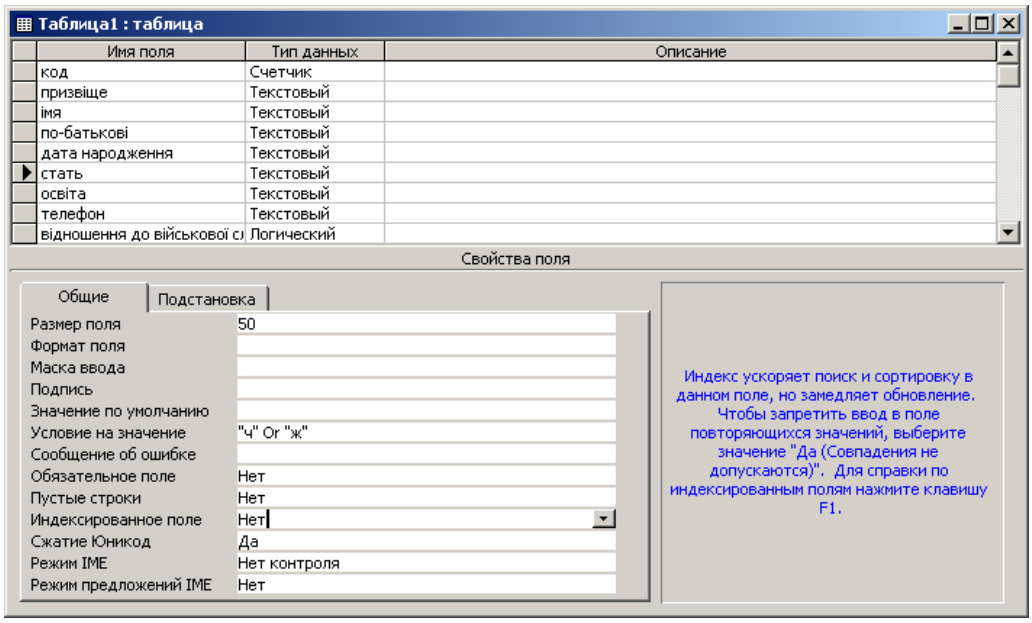


Рис. 4. Вікно конструктора таблиць

Після створення списку підстановок потрібно натиснути кнопку ***"Далее"***.

6. На останньому кроці роботи майстра підстановок вводиться назва поля підстановок. ***Access*** автоматично пропонує ім'я поля як назву поля підстановок (рис. 7). Завершення роботи майстра підстановок здійснюється натисненням кнопки ***"Готово"***.

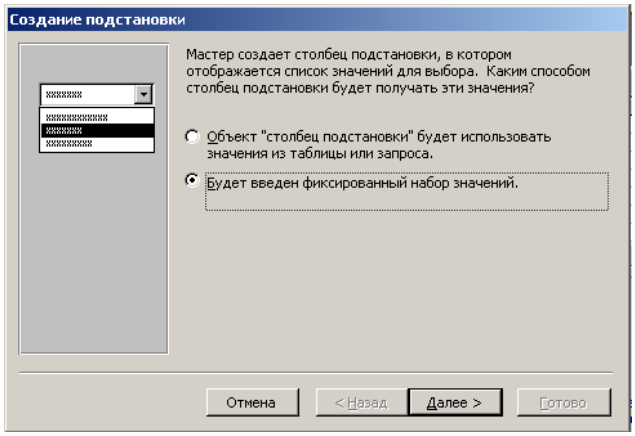


Рис. 5. Перший крок роботи Майстра підстановок

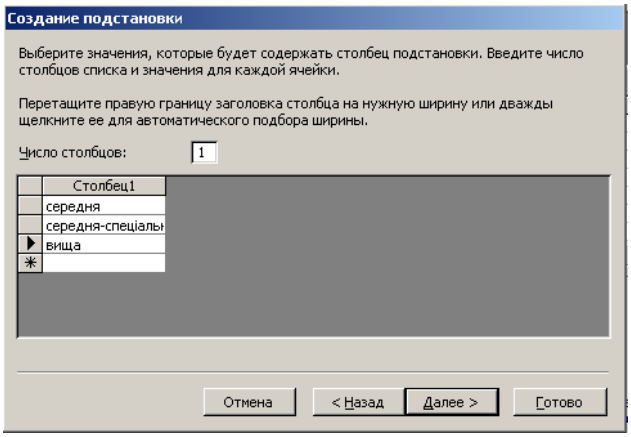


Рис. 6. Створення списку підстановок

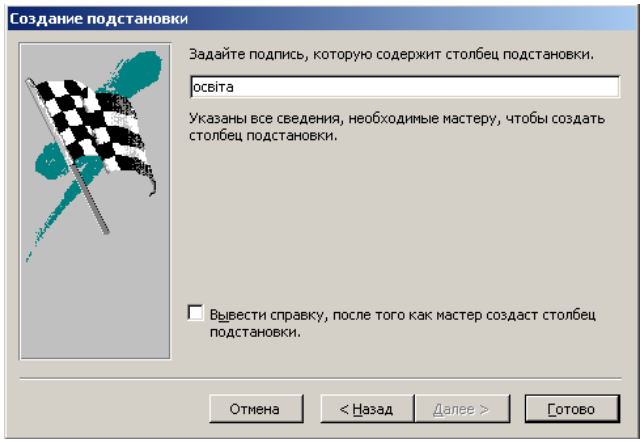


Рис. 7. Завдання назви поля підстановок

7. Після завершення створення структури таблиці потрібно закрити вікно конструктора таблиці і дати таблиці ім'я ***"Кадри".*** Після чого відкрити таблицю і ввести в неї початкові дані. При заповненні бази даних інформацією слід пам'ятати, що дані типу *"счетчик"* заповнюються автоматично. При заповненні інших полів таблиці потрібно вводити значення, які відповідають типам даних полів таблиці. Оскільки поле ***"Дата народження"*** має короткий формат дати, то вводити її потрібно у вигляді ДД.ММ.РР, де ДД - номер дня, ММ - номер місяця, РР - номер року (рис. 8).

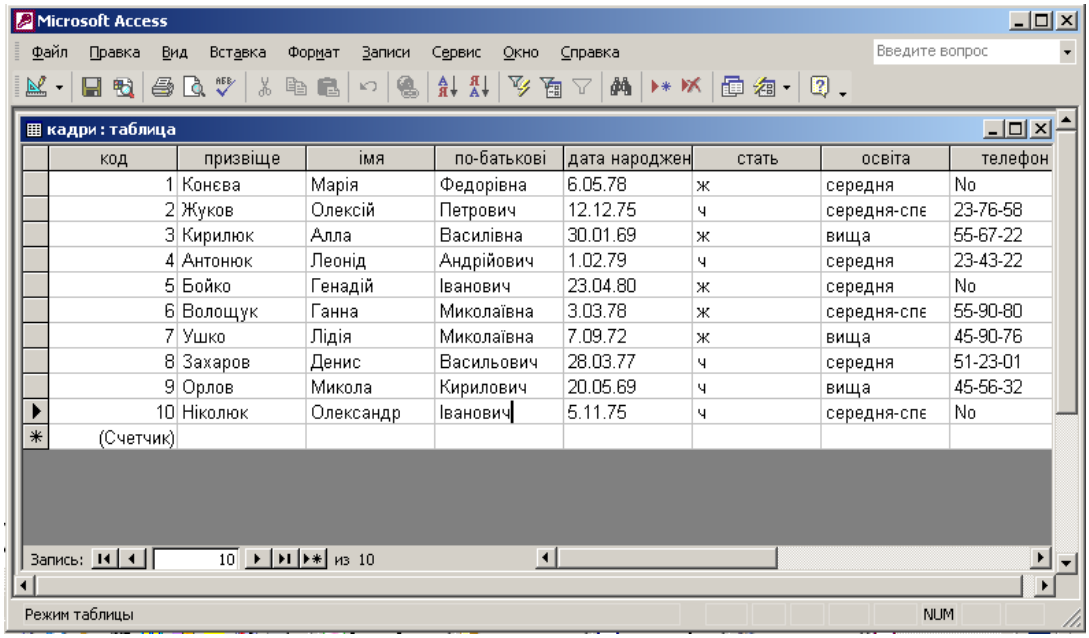


Рис. 8. Заповнення таблиці інформацією

Після цього треба закрити вікно конструктора, зберігши введені дані. Відкрити таблицю для введення даних і ввести інформацію у нові додані поля.

9. Закріплення стовпчиків використовується для введення даних і перегляду таблиць з великою кількістю стовпців. Під час прокрутки полів таблиці закріплені стовпці залишаються на екрані. Для закріплення стовпців потрібно виділити їх і вибрати *"Формат"* -*"Закрепить столбцы"*. Закріплення стовпчиків відображається весь час у лівій частині екрана (рис. 10).

8. Для зміни структури таблиці, тобто для того, щоб додати або видалити поле таблиці, потрібно перейти в режим конструктора таблиць, натиснувши кнопку *"Конструктор"* в головному вікні бази даних. На екрані з'явиться вікно конструктора таблиць (рис. 9). Нижче у вікні конструктора потрібно дописати імена нових полів таблиці і вибрати відповідні типи даних.

Для звільнення закріплених стовпців потрібно вибрати *"Формат"* - *"Освободить все столбцы"*. Але навіть після цього стовпці залишаться крайніми лівими стовпцями таблиці. Для переміщення їх у потрібне місце потрібно виділити стовпчик, натиснувши на його назві, після чого перемістити його в потрібне місце, утримуючи натисненою ліву клавішу "мишки".

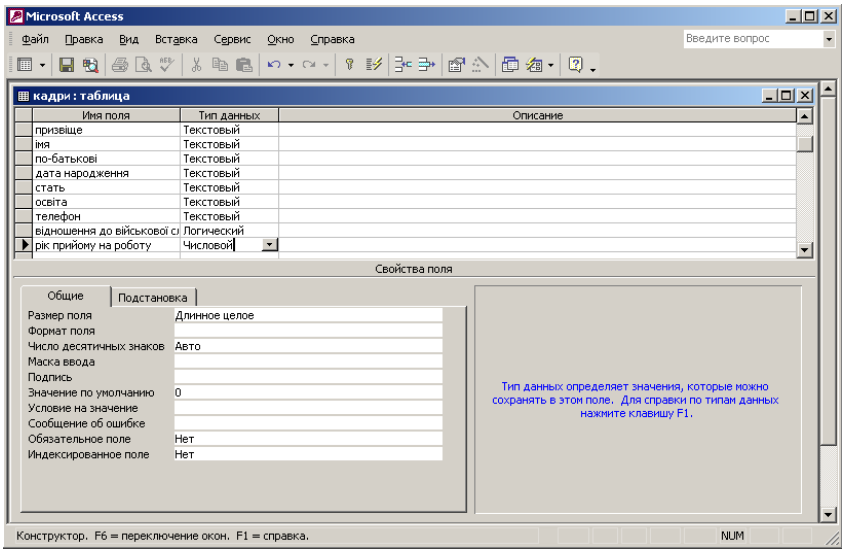


Рис. 9. Зміна структури таблиці в режимі конструктора

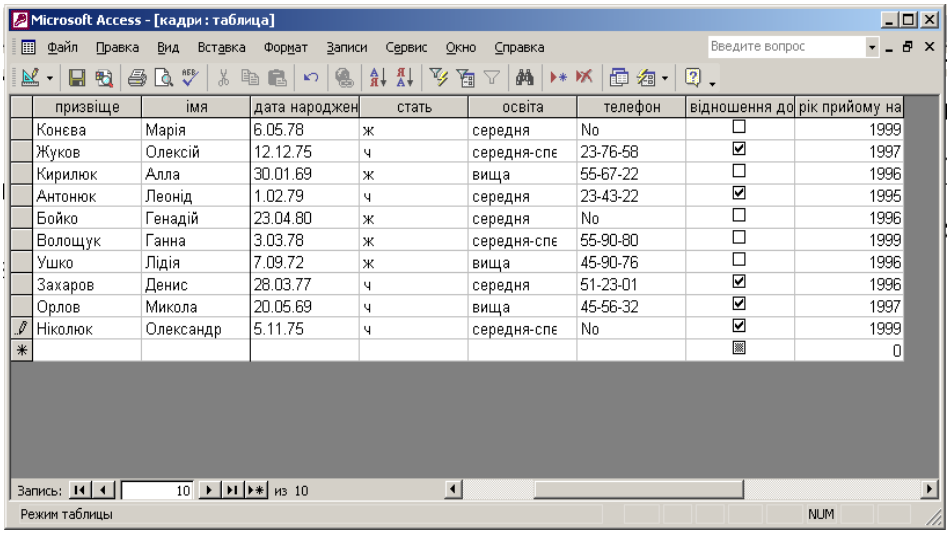


Рис. 10. Закріплення стовпчиків

10. Для впорядкування записів по деякому полю потрібно виділити необхідний стовпчик і натиснути одну з кнопок - £|впорядкування за зростанням або XI - впорядкування за спаданням.

11. Для встановлення альбомної орієнтації сторінки потрібно відкрити таблицю і вибрати *"Файл"* - *"Параметры страницы"*, на вкладці *"Страница"* активізувати перемикач *"Альбомная"* *(рис.* 11).

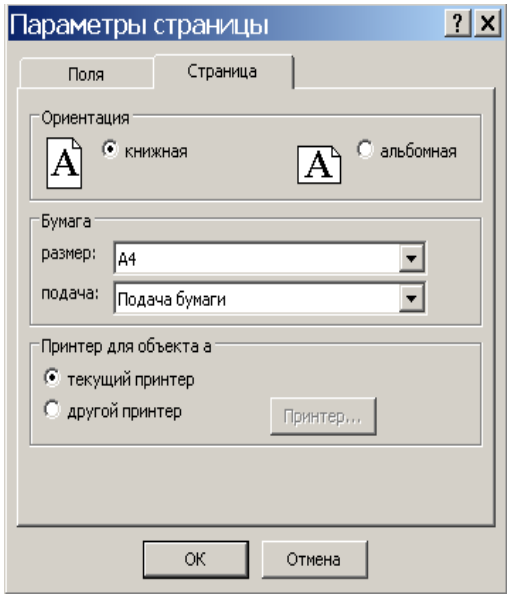


Рис. 11. Зміна параметрів сторінки

Завдання: 1. Створити таблицю з такими полями:

Назва поля	**Тип даних**
Код	Счетчик
Прізвище	Текстовый
Ім’я	Текстовый
По батькові	Текстовый
Дата народження	Дата/время
Стать	Текстовый
Освіта	Мастер подстановок, список значень майстра підстановок: • *середня* • *середня спеціальна* • *вища*
Телефон	Текстовый
Відношення до військової служби	Логический

Для поля ***"Стать"*** встановити властивість *"Условие на значение"* значення "ч"або "ж".

2. Запам'ятати таблицю під іменем ***"Кадри".*** Заповнити таблицю даними відповідно до свого варіанта. Заповнюючи даними поля ***"Освіта"*** здійснювати вибір значення поля зі списку, утвореного за допомогою майстра підстановок.

3. Додати в таблицю ***"Кадри"*** поле ***"Рік прийому на роботу",*** яке заповнити інформацією відповідно до свого варіанта. Заповнюючи інформацією нових доданих полів закріпити поля ***"Прізвище" ^ "Ім'я".***

4. Приховати відображення стовпчиків ***"Ім'я"*** та ***"По батькові"*** за допомогою команди **"Скрыть столбцы"** меню **"Формат".**

5. Показати всі стовпчики таблиці, включаючи приховані стовпчики за допомогою команди **"Показать столбцы"** меню **"Формат".**

6. Впорядкувати записи у алфавітному порядку за полем ***"Прізвище".***

7. Впорядкувати записи за зростанням значень поля ***"Код"***.

8. Поміняти поля ***"Стать"*** та ***"День народження"*** місцями.

9. Встановити альбомний макет сторінки для друку таблиці.

10. Переглянути таблицю.

11.Надрукувати таблицю.

ЛАБОРАТОРНА РОБОТА №2

ТЕМА:
УПРАВЛІННЯ ДАНИМИ У ТАБЛИЦЯХ ДАНИХ ACCESS

1. Для побудови звичайного *фільтра* потрібно вибрати *„Записи”, „Фильтр”, „Изменить фильтр”*. На екрані залишиться порожній запис. У потрібному полі вводиться критерій фільтрації. Для отримання списку осіб, які є військовозобов’язаними, потрібно в полі „Відношення до військової служби” встановити значення – *ƒ* (рис.12)

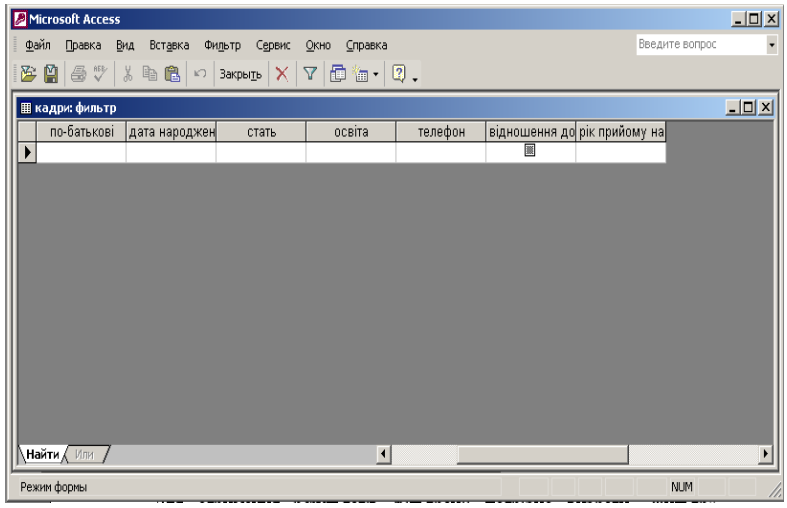


Рис.12.Завдання умов фільтрації у звичайному фільтрі

Для отримання результатів фільтрації потрібно вибрати „Фильтр” – „Применить фильтр” (рис.13).

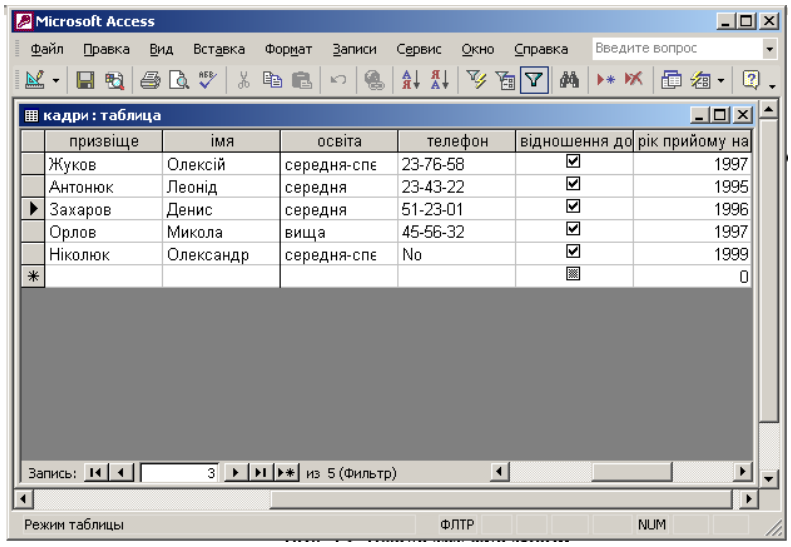


Рис. 13. Результат фільтрації

Для відключення фільтра потрібно вибрати *"Фильтр" "Удалить фильтр"*.

2. Для побудови розширеного фільтра потрібно вибрати *"Записи"* - *"Фильтр"* - *"Расширенный фильтр"*. На екрані з'явиться бланк побудови розширеного фільтра. Для відбору даних відповідно до значення деякого поля потрібно в рядку *"поле"* вибрати зі списку ім'я поля- ***"Стать" & "Відношення до військової служби"***, в рядку *"условие от бора"* задається умова відбору записів. Для поля ***"Стать"*** умовою відбору є значення "ч", для поля ***"Відношення до військової служби”*** - *"Истина"* або *"Да"* (рис. 14).

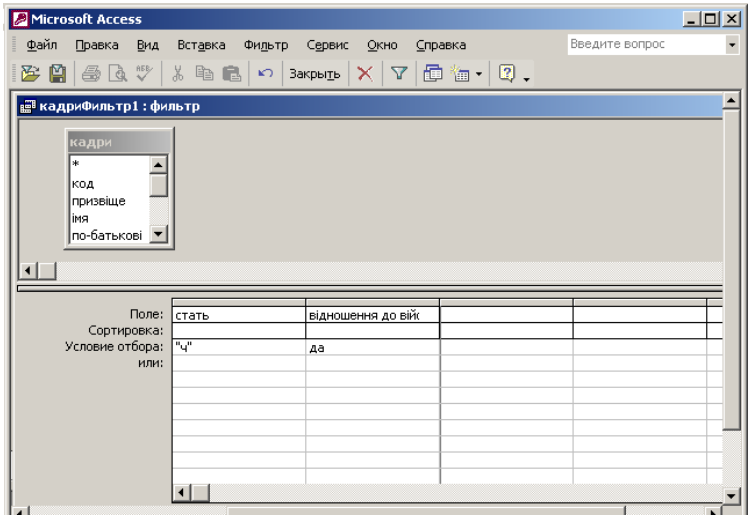


Рис. 14. Вікно розширеного фільтра

3. Для побудови *фільтру* по виділеному потрібно в таблиці виділити значення поля, яке є критерієм фільтрації, в нашому випадку потрібно в полі ***"Освіта"*** виділити значення **"вища".** Після цього вибираємо *"Записи "Фильтр по виделеному" (рис.* 15).

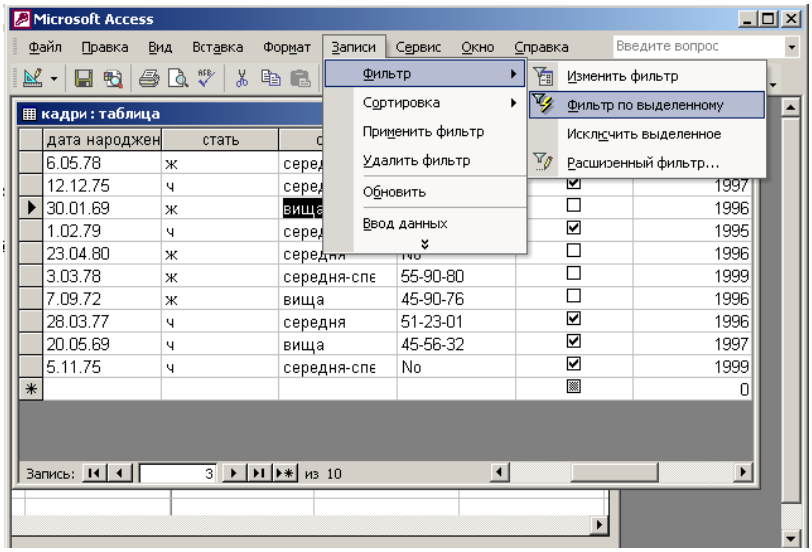


Рис. 15. Створення фільтру по виділеному

4. Для створення в базі даних таблиці в режимі таблиці потрібно в головному вікні бази даних натиснути кнопку *"Создать"* і вибрати*"Режим таблицы "*.

5. Для копіювання полів з даними з однієї таблиці в іншу потрібно виділити поле в таблиці ***"Кадри",*** натиснувши на його назві,скопіювати його в буфер обміну, вибравши *"Правка"- "Копировать".*Після цього потрібно виділити порожнє поле в новій створенійтаблиці і вставити інформацію з буфера, вибравши *"Правка" -"Вставить".* Для переключення між вікнами з таблицями використовується меню *"Окно"* (рис. 16).

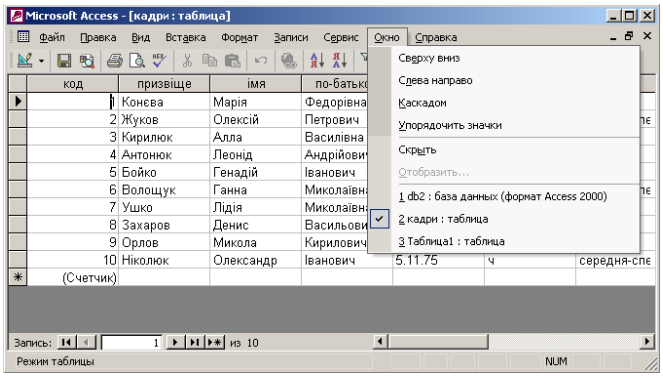


Рис. 16. Переключення між відкритими вікнами

6. Для того, щоб додати в таблицю нове поле потрібно перейти в режим Конструктора таблиць, вказати ім'я поля та тип даних.

7. Для зміни порядку розміщення полів у базі даних потрібно виділити поле, яке потрібно переставити, натиснути "мишку" на імені поля і, утримуючи натисненою ліву клавішу "мишки", перемістити поле в потрібне місце.

Завдання:

1. Відкрити таблицю ***"Кадри"*** своєї бази даних.
2. За допомогою функції звичайного фільтру *(Записи - Фильтр - Изменить фильтр)* вивести на екран список осіб, які є військовозобов'язаними.
3. Відключити фільтр.
4. За допомогою розширеного фільтру створити список осіб чоловічої статі, які мають середню спеціальну освіту.
5. Надрукувати отриманий список.
6. За допомогою розширеного фільтру отримати список осіб, прийнятих на роботу після 1995 року.
7. За допомогою фільтру по виділеному отримати список осіб з вищою освітою.
8. У поточній базі даних у режимі таблиці створити нову таблицю.
9. Скопіювати в нову таблицю поля з даними таблиці ***"Кадри"*** у такій послідовності: ***Прізвище, Телефон, Дата народження, Освіта.*** Зберегти таблицю під іменем **"Список".**
10. У таблицю ***"Список"*** додати поля ***" Рік прийому на роботу"*** та ***„Ім'я"*** з таблиці ***„Кадри"*** (див.дод.1).
11. Розташувати поля таблиці ***„Список"*** у такому порядку: ***Призвіще, Ім'я, Дата народження, Рік прийому на роботу, Освіта***.
12. У таблиці ***"Список"*** за допомогою фільтру по виділеному отримати список працівників фірми, прийнятих на роботу в 1998 році.
13. Впорядкувати записи в алфавітному порядку за полем ***"Прізвище".***
14. Впорядкувати записи за зростанням поля ***„Код".***
15. Встановити альбомну орієнтацію сторінки.
16. Надрукувати таблицю ***„Список".***

ЛАБОРАТОРНА РОБОТА №3
ТЕМА: ПОБУДОВА ЗАПИТІВ У СЕРЕДОВИЩІ ACCESS

1.Для видалення поля з таблиці потрібно виділити поле, натиснувши на його назві, і натиснути клавішу **delete.** Решта дій щодо створення таблиць, вводу інформації в таблиці описана в методичних вказівках до лабораторних робіт №1 та №2.

2. Для створення запиту потрібно перейти на вкладку *"Запросы "* вікна бази даних, натиснути кнопку *"Создать "*. Для створення запиту на вибірку потрібно вибрати режим конструктора.

3. У наступному вікні потрібно вибрати таблицю, на основі якої буде будуватись запит, і натиснути кнопку *"Добавить"*(рис. 17), після чого закрити вікно *"Добавление таблицы "*.

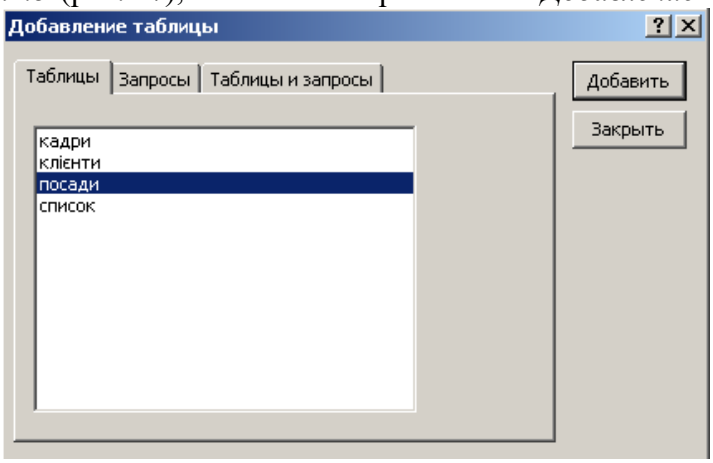


Рис. 17. Додавання таблиці в запит

4. Для відображення в запиті поля таблиці потрібно в рядку *"Поле"* на бланку запитів із списку вибрати потрібне поле таблиці (рис. 18).

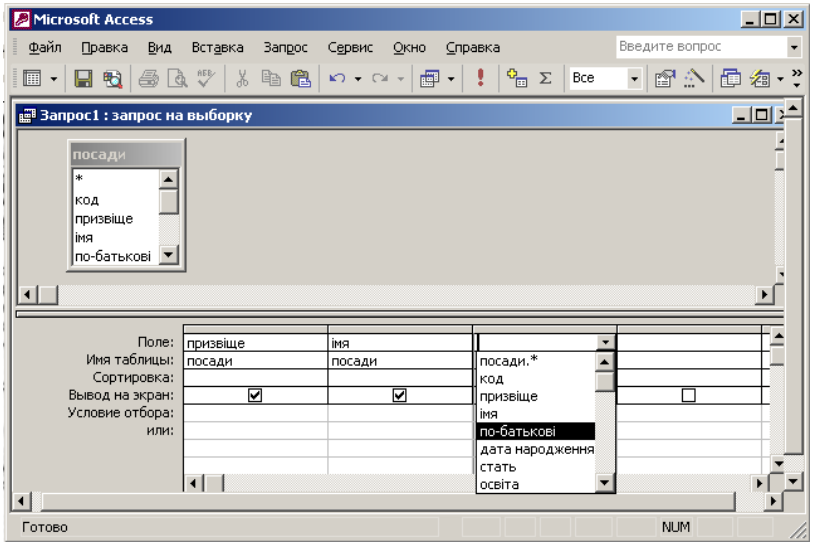
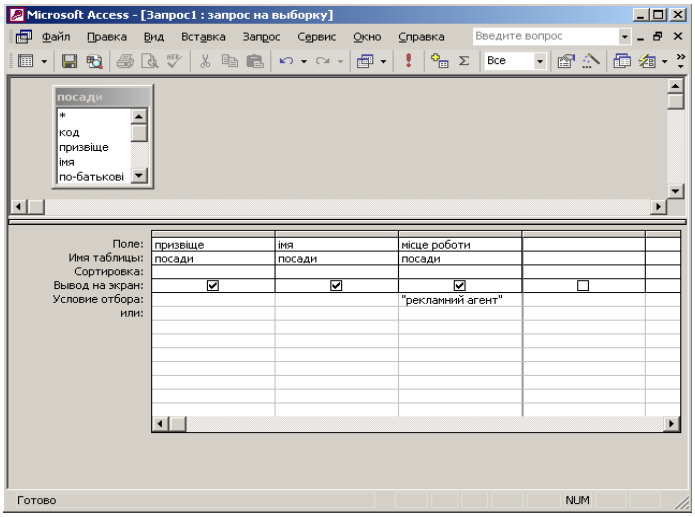


Рис. 18. Вибір полів для запиту

У результаті виконання запиту потрібно отримати список працівників, які є рекламними агентами. Для задания умови відбору записів до запиту потрібно в рядку *"условие отбора"* для поля ***"Місце роботи"*** задати умову *"рекламний агент"* (рис. 19).



19. Задания критеріїв відбору записів до запиту

Для запуску запиту і отримання його результатів на екрані потрібно вибрати *"Запрос"*- *"Запуск"*.

5. Для отримання списку працівників, що отримують надбавку, потрібно на бланку запитів вибрати відповідні поля, і для поля ***"Сума надбавки"*** встановити умову відбору *">0"*.

6. Для відображення списку фірм, що внесли попередню оплату, потрібно для поля ***"Попередня оплата"*** встановити умову відбору *"истина "*.

*7.*Для створення підсумкового запиту потрібно у вікні конструктора запитів вибрати *"Вид" "Групповые операции"*. На бланку запитів з'явиться поле ***"Групповая операция",*** в якому автоматично буде записано значення *"группировка"*, яка означає, що записи з однаковим значенням полів будуть відображатись у запиті лише один раз. Для використання інших підсумкових функцій у підсумковому запиті потрібно зі списку вибрати назву іншої підсумкової функції. Для обчислення середнього значення суми угоди потрібно використати підсумкову функцію avg (рис. 20).

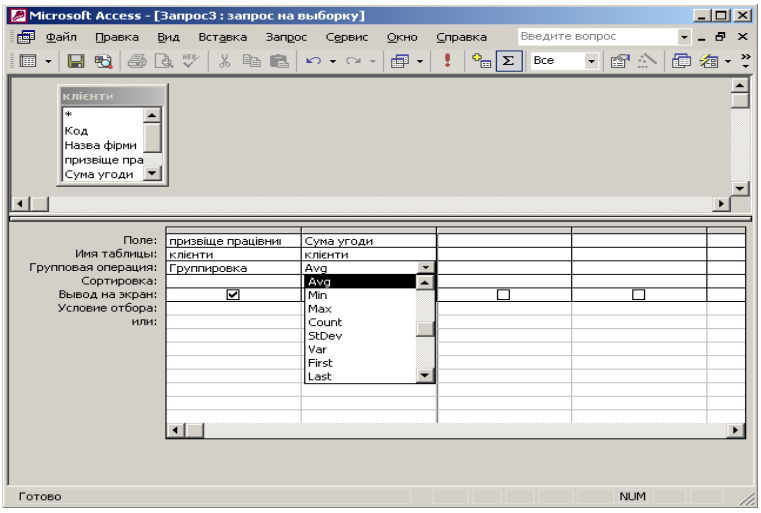


Рис. 20. Використання підсумкових функцій у підсумковому запиті

8. Для отримання суми знижки в запиті потрібно в запит додати обчислювальне поле. Для цього потрібно в першому порожньому стовпчику на бланку запитів в рядку *"Поле"* натиснути праву клавішу "мишки" і вибрати ***"Построить".*** На екрані з'явиться вікно *"Построителя выражений"* (рис. 21).

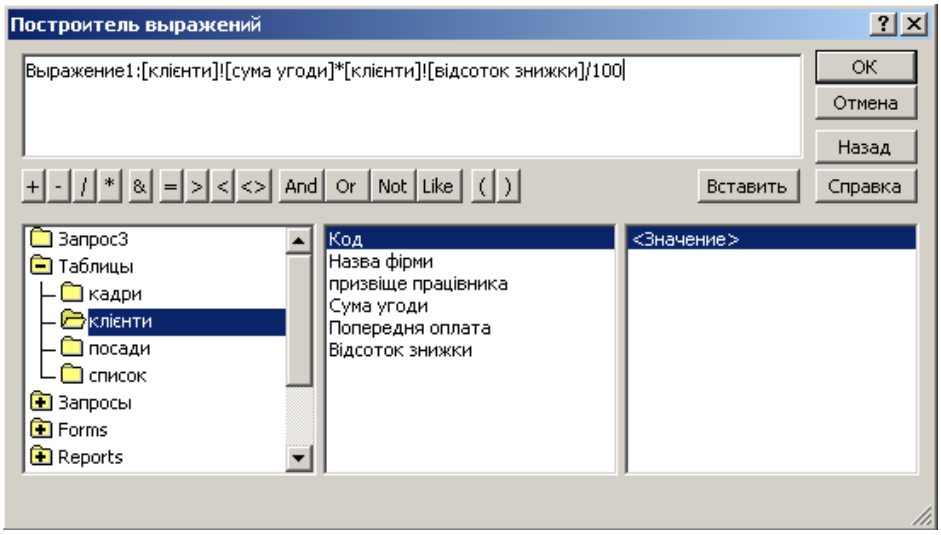


Рис. 21. Вікно "Построителя выражений"
з прикладом побудованого виразу

9. Для використання в запиті значень полів таблиці, потрібно в крайньому лівому стовпці вікна *Построителя выражений* отримати список таблиць бази даних, двічі натиснувши "мишку" на піктограмі поряд зі словом *"Таблицы"*. Нижче виводиться список таблиць бази даних. Для використання одного з полів таблиці потрібно в центральному стовпчику двічі натиснути на імені поля. Ім'я поля з'явиться в полі для побудови обчислювальних виразів. Після завершення побудови обчислювального виразу потрібно закрити вікно *Построителя выражений.* Ім'я поля, яке є обчислювальним виразом, присвоюється Access автоматично.

Для зміни імені поля в запиті потрібно натиснути праву клавішу "мишки" в рядку *"Имя поля",* вибрати пункт *"Свойства"* в рядку *"Подпись",* ввести нове ім'я поля і закрити вікно властивостей поля (рис. 22).

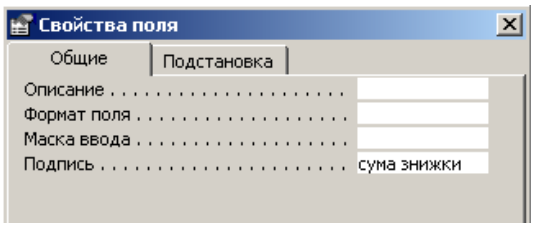


Рис. 22. Зміна імені поля в запиті

8. Для створення перехресного запиту потрібно у вкладці *"Запросы"* натиснути кнопку *"Создать",* вибрати *"Перекрестный запрос"*. Запуститься майстер перехресних запитів. На першому кроці вибирається таблиця, на основі якої буде створюватись перехресний запит. Для переходу до наступного кроку майстра перехресних запитів потрібно натиснути кнопку *"Далее"* (рис. 23).

На другому кроці роботи майстра перехресних запитів вибирається поле, яке буде використано як заголовки рядків - поле ***"Прізвище працівника"*** (рис. 24). Для вибору поля потрібно натиснути двічі лівою клавішею "мишки" на його імені в списку *"Доступные поля "*.

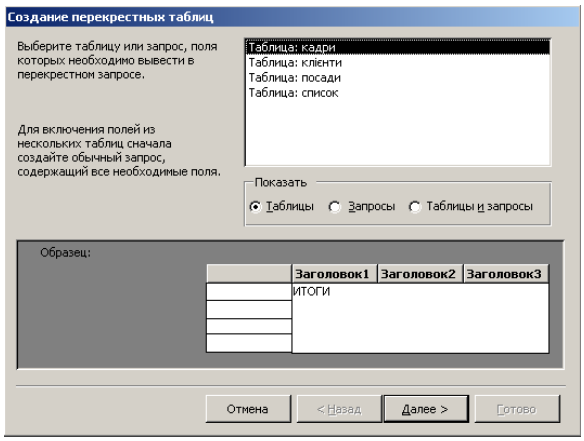


Рис. 23. Вибір таблиці для перехресного запиту

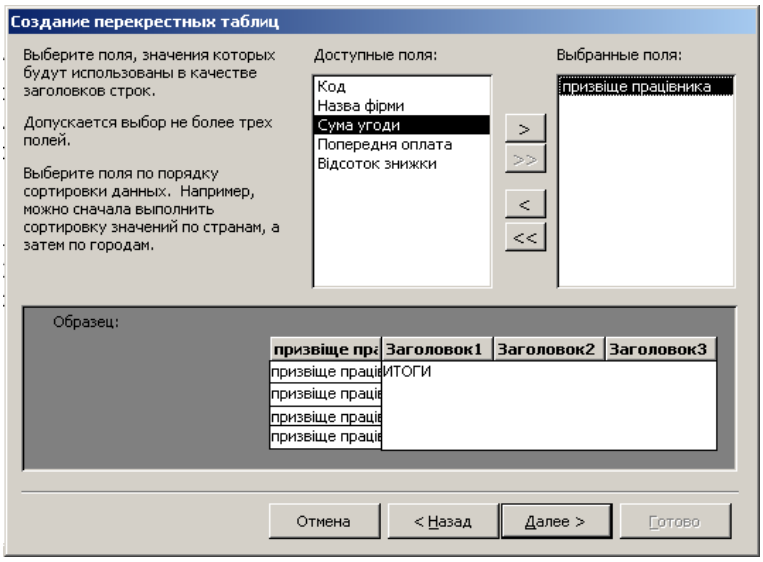


Рис. 24. Вибір поля для заголовків рядків перехресного запиту

На третьому кроці вибираються поля, що будуть використані як заголовки стовпців - поле ***"Назва фірми"***(рис. 25).

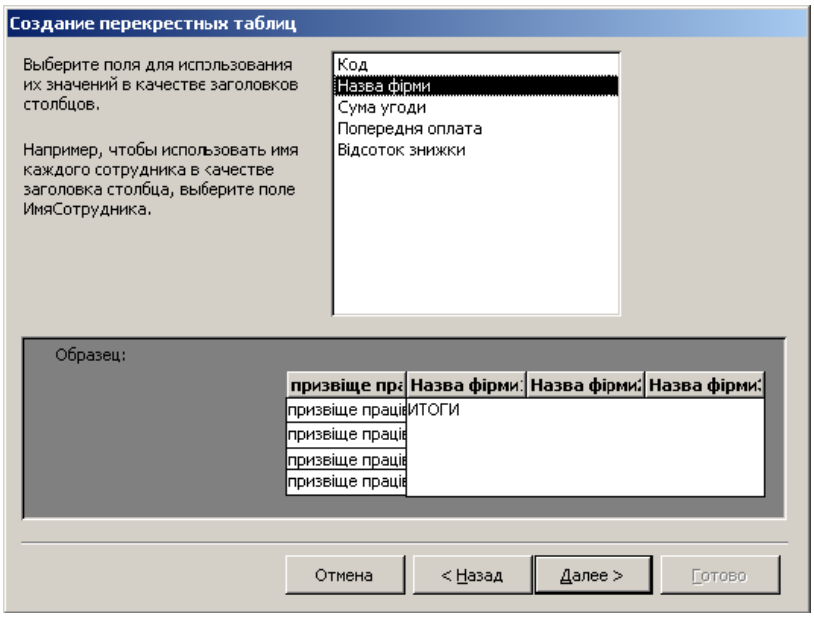


Рис. 25. Вибір полів для заголовків стовпців перехресного запиту

На четвертому кроці вибирається поле, значення якого буде розташовано на перетині рядків та стовпчиків перехресної таблиці -це поле ***"Сума угоди".***

Для обчислення додаткового стовпчика з підсумковими значеннями (сумарне значення для кожного працівника) потрібно у вікні четвертого кроку майстра перехресних запитів активізувати перемикач *"Вычислить итоговое значение для каждой строки" "Да"* і вибрати значення підсумкової функції **Sum** (рис. 26).

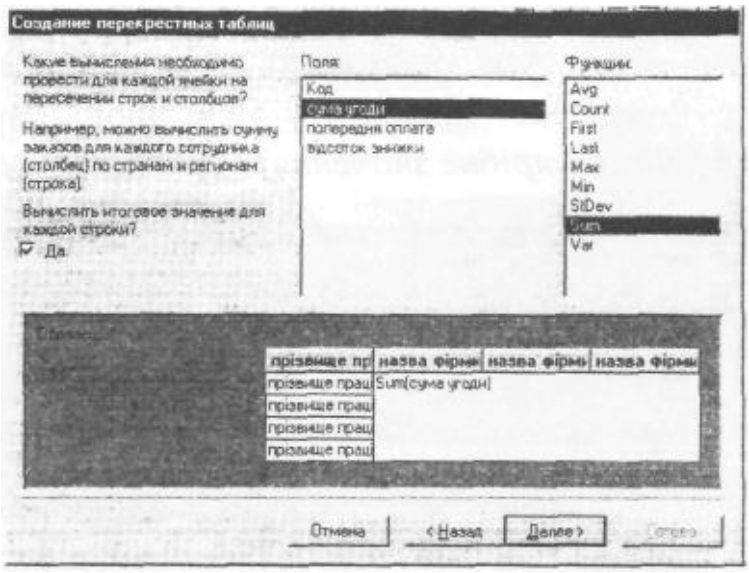


Рис. 26. Вибір поля для виводу на перетині
рядків та стовпчиків і вибір підсумкової функції

На останньому кроці задається ім'я перехресного запиту **(зв'язки).** Після натиснення кнопки **"Готово"** на екрані з'являється результат роботи перехресного запиту - таблиця із вибраних рядків та стовпчиків, на перетині яких розміщено значення вибраних полів (рис. 27).

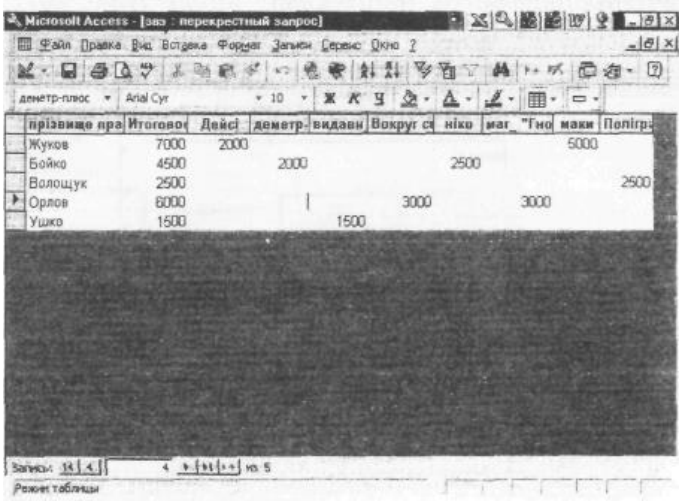


Рис. 27. Результат роботи перехресного запиту

11. Робота параметричного запиту залежить від значень деякої величини - параметра запиту. Результат запиту буде визначатись значенням параметра, яке вводиться в спеціальному вікні перед відбором записів. Для створення параметричного запиту потрібно в рядку *"Условие отбора"* в квадратних дужках вказати текст, який буде використано як текст підказки у вікні вводу параметра запиту, наприклад *"[введіть потрібне значення]"* (рис. 28).

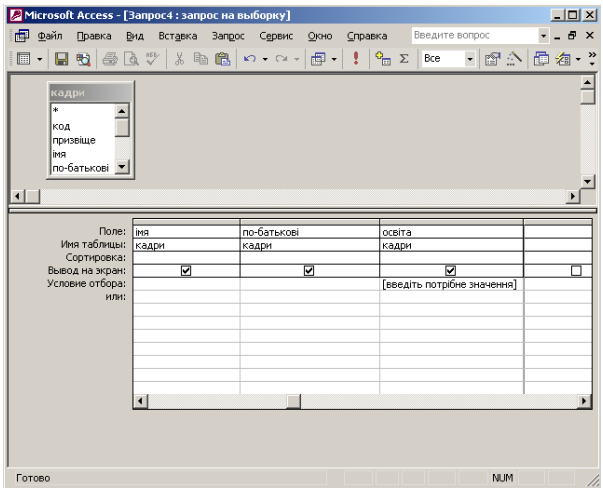


Рис. 28. Ввід тексту-підказки при створенні параметричного запиту

Після запуску запиту у вікні для вводу параметра потрібно ввести шукане значення параметра, наприклад, *"вища"* (рис. 29).

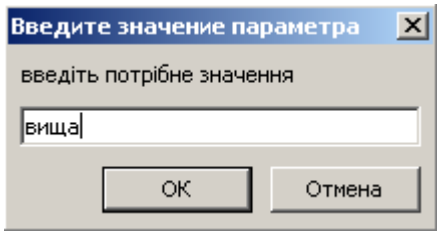


Рис. 29. Вікно для вводу значення параметра

Після натиснення *"ОК"* на екран виведуться записи таблиці, в яких значення поля ***"Освіта"*** відповідає введеному значенню параметра (рис. 30).

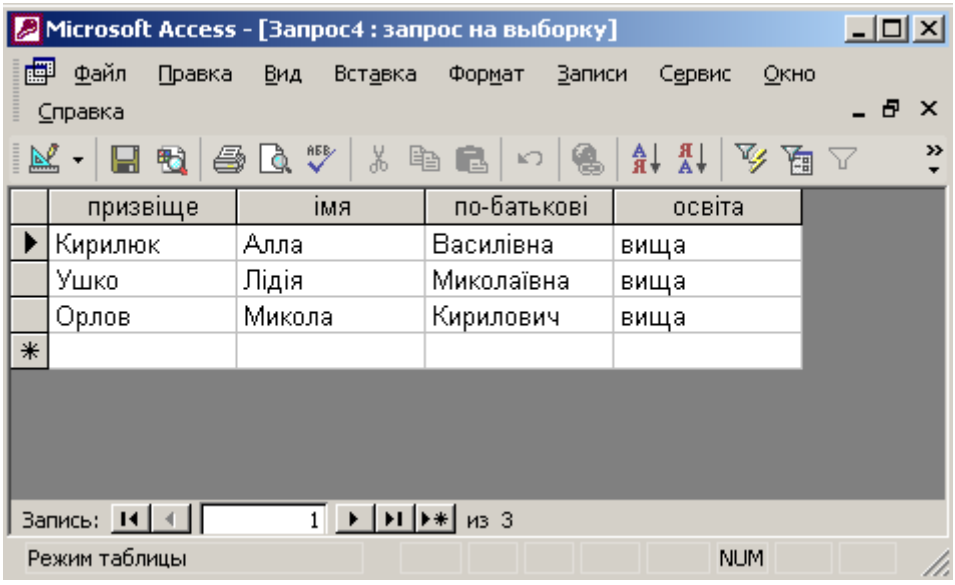


Рис. 30. Результат роботи параметричного запиту із значенням параметра "вища» для поля **"Освіта"**

12. Для того, щоб зробити деяке поле ключовим, потрібно у вікні конструктора таблиць виділити поле і вибрати, наприклад, *"Правка"* -- *"Ключевое поле",* або натиснути відповідну піктограму на панелі інструментів (рис. 31).

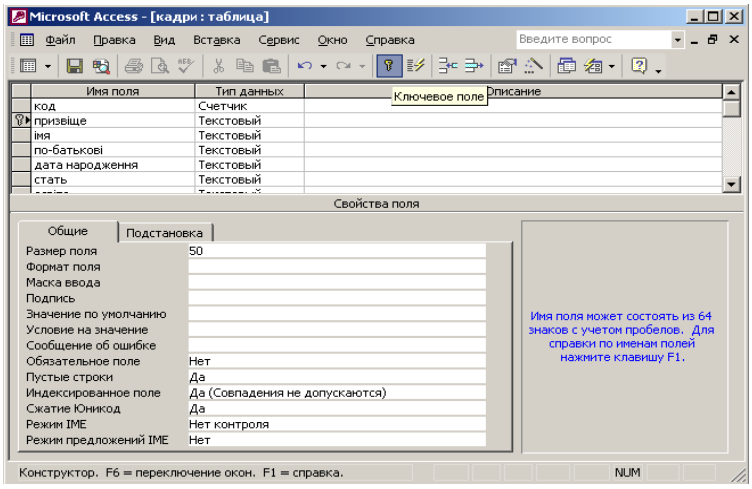


Рис. 31. Вікно конструктора таблиць, в якому ключовим полем зроблено поле **"Прізвище"**

13. Для зв'язування таблиць потрібно вибрати *"Сервис" -"Схема данных"*. У вікні схеми даних потрібно вибрати таблиці, які будуть зв'язані, виділяючи кожну з них, і натискаючи кнопку *"Добавить"* **(рис. 32)**.

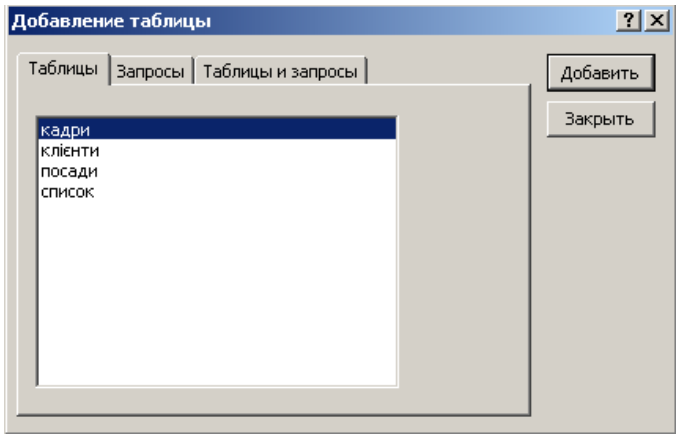


Рис. 32. Додавання таблиць до схеми даних

Для зв'язування таблиць по деякому полю потрібно у вікні *"Схема данных"* перетягнути поле, по якому відбувається зв'язок з однієї таблиці на те саме поле іншої таблиці, утримуючи натисненою ліву клавішу "мишки". Після того, як клавіша "мишки" буде відпущена, на екрані з'явиться вікно *"Связи"* для створення зв'язку. У полі *"Тип отношения"* Access автоматично визначить тип зв'язку між таблицями (може бути один-к-одному або один-ко-многим). Для успішної спільної роботи кількох таблиць у вікні *"Связи"* потрібно активізувати перемикачі *"Обеспечение целостности данных"*, *"Каскадное обновление связанных полей"* та *"Каскадное удаление связанных записей"* (рис. 33).

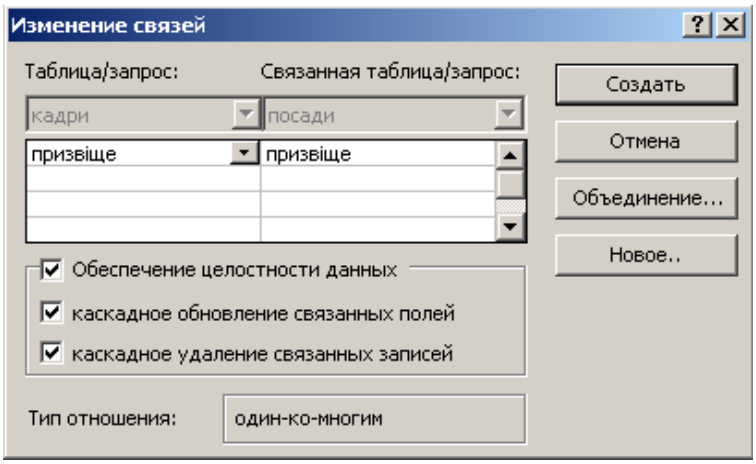


Рис. 33. Вікно "Связи" для типу зв'язку "один-к-одному"

Після активізації потрібних перемикачів натискаємо *"Создать"*, і створений зв'язок відображається в схемі даних (рис. 34).

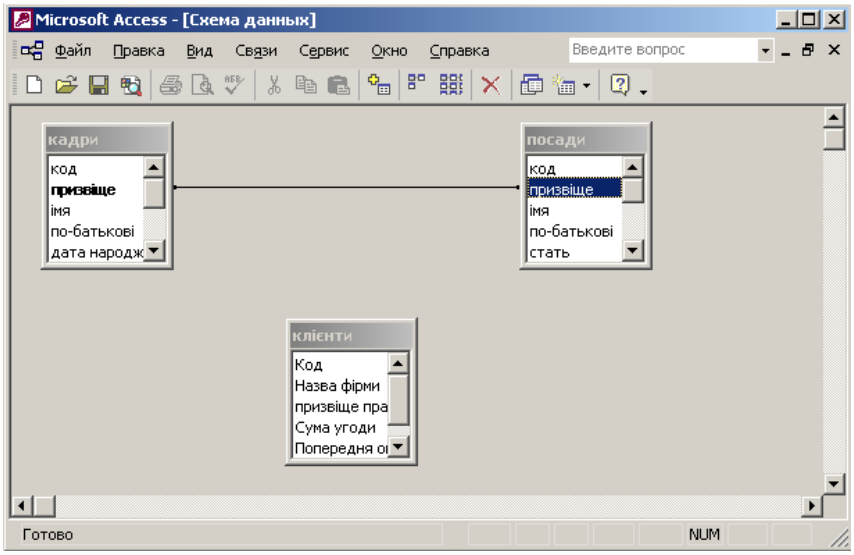


Рис. 34. Створений зв'язок у вікні "Схема данных"

Таку дію потрібно зробити для кожної пари таблиць, вказаних в умові.

14. Запит на основі кількох таблиць створюється так само, як і звичайний запит на вибірку на основі однієї таблиці, тільки при додаванні таблиць, на основі яких будуватиметься запит, потрібно вибирати таблицю і для кожної з них натискати кнопку *"Добавить "*.

15. Для побудови модифікаційного запиту на створення таблиці потрібно спочатку створити звичайний запит на вибірку, в який включити всі необхідні поля всіх таблиць (рис. 35)

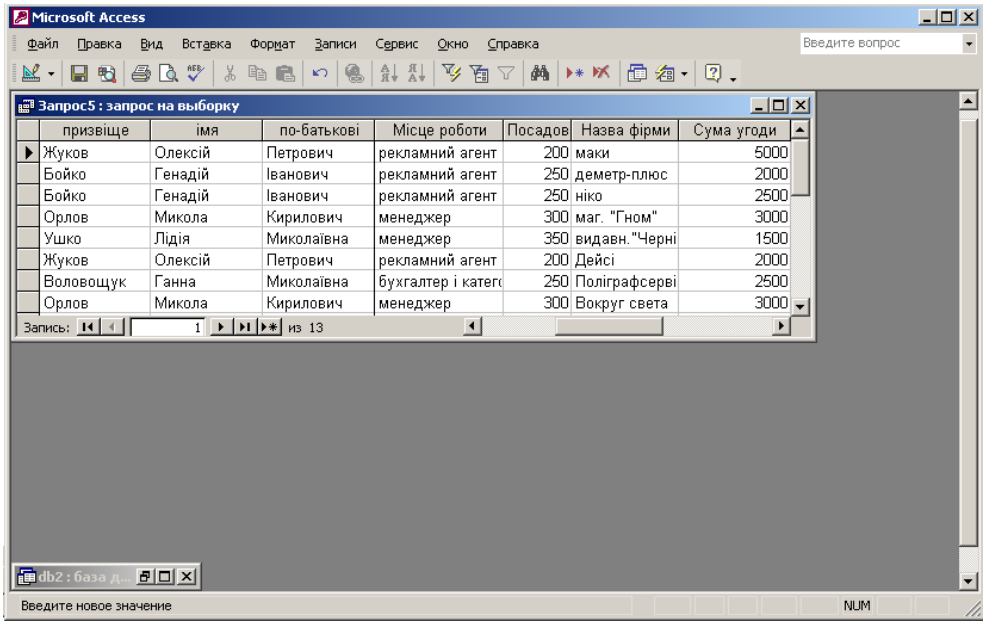


Рис. 35. Запит на вибірку, на основі якого буде будуватись запит на створення таблиці

Після цього потрібно перейти у вікно конструктора запитів і вибрати *"Запрос"* - *"Создание таблицы"*, і у вікні, що з'явиться, задати ім'я нової створеної таблиці (рис. 36).

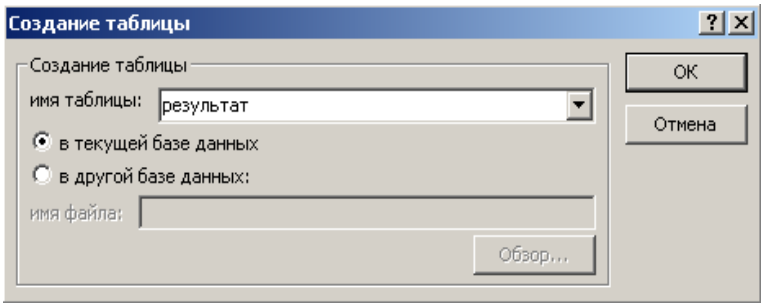


Рис. 36. Вікно для завдання імені нової створеної таблиці

Після запуску запиту на створення таблиці на вкладці *"Таблицы"* з'явиться нова таблиця - *"Результат"* (рис. 37), яка буде містити відібрані поля з таблиць ***"Клієнти", "Посади"*** та ***„Кадри".***

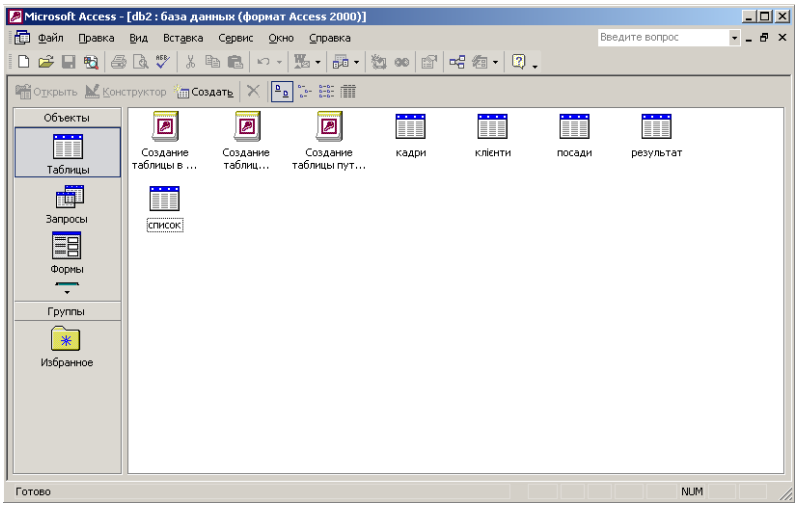
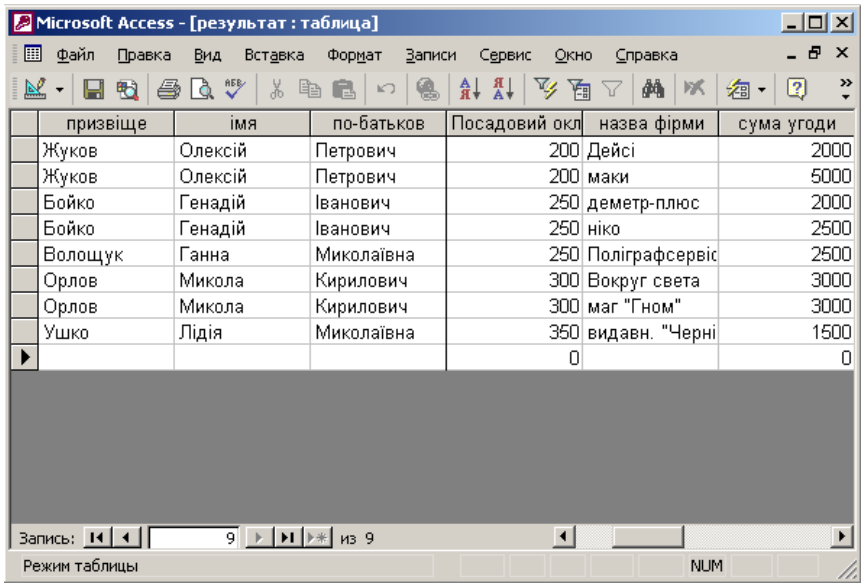


Рис. 37. Результат роботи запиту на створення таблиці — нова таблиця "Результат"



призвіще	ім'я	по-батьков	Посадовий окл	назва фірми	сума угоди
Жуков	Олексій	Петрович	200	Дейсі	2000
Жуков	Олексій	Петрович	200	маки	5000
Бойко	Генадій	Іванович	250	деметр-плюс	2000
Бойко	Генадій	Іванович	250	ніко	2500
Волощук	Ганна	Миколаївна	250	Поліграфсервіс	2500
Орлов	Микола	Кирилович	300	Вокруг света	3000
Орлов	Микола	Кирилович	300	маг "Гном"	3000
Ушко	Лідія	Миколаївна	350	видавн. "Черні	1500
			0		0

Рис. 38. Структура нової створеної таблиці

Завдання:

1. Скопіювати таблицю ***"Кадри"*** цієї бази даних у таблицю, яку назвати ***"Посади".*** Відкрити таблицю посади і видалити з цієї таблиці поля *"Дата народження", "Телефон", "Відношення до військової служби".*

2. Перейти в режим конструктора таблиці ***"Посади"*** і додати в цю таблицю наступні поля:

Місто	Текстове
Місце роботи	Текстове
Стаж роботи	Числове
Посадовий оклад	Числове, дійсне
Сума надбавок	Числове, дійсне

3. Заповнити нові додані поля таблиці *"Посади"* інформацією відповідно до свого варіанта.

4. Створити таблицю ***"Клієнти"*** з такими полям

Назва фірми	Текстове
Прізвище працівника	Текстове
Сума у годинах	Числове
Попередня оплата	Логічне
Відсоток знижки	Числове

5. Заповнити таблицю ***"Клієнти"*** інформацією щодо до свого варіанту.

6. На основі таблиці ***"Посади"*** створити запит, в якому відобразити прізвище, ім'я, по батькові працівників, які є рекламними агентами. Зберегти запит під іменем **"Агенти".** Роздрукувати запит.

7. На основі таблиці ***"Посади"*** створити запит, в якому відобразити прізвище, ім'я, місце роботи та суму надбавки тих працівників, які отримують надбавку до зарплати. Зберегти запит під іменем **"Надбавка ".** Роздрукувати запит.

9. На основі таблиці ***"Клієнти"*** побудувати запит, в якому відобразити назву фірми, прізвище працівника, суму угоди тих фірм, які внесли попередню оплату. Зберегти запит під іменем ***"Попередня оплата".*** Роздрукувати запит.

10.За допомогою Конструктора запитів створити підсумковий запит на основі таблиці ***"Клієнти",*** в якому відобразити прізвище працівника, суму угоди та підрахувати середнє значення суми угоди для кожного з працівників. Для зазначення, що запит буде підсумковим, потрібно вибрати ***"Вид"*** - ***"Групповые операции".*** Використати групову операцію **Avg.** Зберегти запит під іменем **"Середня сума** ". Роздрукувати запит.

10.На основі таблиці ***"Клієнти"*** побудувати запит, в якому відобразити прізвище працівника, назву фірми, суму угоди, відсоток знижки та суму знижки. Поле ***"Сума знижки"*** отримати як обчислювальне поле, в якому потрібну суму угоди помножити на відсоток знижки та розділити на 100. Використовуючи вікно властивостей поля, назвати додане поле ***"Сума знижки".*** Назвати запит **"Знижка ".** Роздрукувати запит.

11.На основі таблиці ***"Клієнти"*** створити перехресний запит, у рядках якого відобразити прізвище працівника, в стовпчиках -назву фірми, а на перетині - суму угоди. Додати поле, в якому обчислити сумарне значення для кожного працівника. Зберегти запит під іменем ***"Зв'язки".*** Роздрукувати запит.

12.На основі таблиці ***"Кадри"*** створити параметричний запит, в якому відібрати з таблиці працівників із заданим значенням поля ***"Освіта".*** Зберегти запит під іменем **"Освіта".** Роздрукувати запит.

13.У таблиці ***"Кадри"*** зробити ключовим полем поле ***"Прізвище",*** в таблиці ***"Посади"*** - ***"Прізвище",*** в таблиці ***"Клієнти"*** *– назву фірми.*

14.Зв'язати між собою таблиці ***"Кадри"*** **і** ***"Посади"*** по полях ***"Прізвище",*** таблицю ***"Кадри"*** **та** ***"Клієнти"*** по полях ***"Прізвище"*** **і** ***"Прізвище працівника",*** аналогічно зв'язати таблицю ***"Посади"*** і ***"Клієнти".***

15. На основі таблиць ***"Посади"*** **і** ***"Клієнти"*** створити запит, в якому відобразити поля ***"Прізвище", "Ім'я", "По батькові", "Місце роботи"*** таблиці ***"Посади"*** і поле ***"Назва фірми"*** таблиці ***"Клієнти".*** Зберегти запит під іменем **"Зв'язки".** Роздрукувати запит.

16.На основі таблиць ***"Кадри", "Клієнти", "Посади"*** створити модифікаційний запит на створення таблиці ***"Результат",*** в якому відобразити поля ***"Прізвище", "Ім'я", "По батькові", "Освіта"*** таблиці ***"Кадри",*** поля ***"Місце роботи", "Посадовий оклад"*** таблиці **"Посади"** і поля ***"Назва фірми", "Сума угоди"*** таблиці ***"Клієнти".*** Застосувати операцію групування записів. Зберегти запит під іменем **"Створення результату".** Роздрукувати запит.

ЛАБОРАТОРНА РОБОТА №4

Тема: Побудова форм у СУБД Microsoft Access

Методичні вказівки до виконання лабораторної роботи №4

1. Побудову запиту на створення таблиці описано в методичних вказівках до виконання лабораторної роботи №3. Як додавати до таблиці нове поле описано в методичних вказівках до виконання лабораторних робіт №1,2.

2. Для побудови автоформи на основі одного із стандартних шаблонів потрібно на вкладці *"Форми"* натиснути кнопку *"Создать"*. У нижній частині вікна створення нової форми вибирається таблиця, на основі якої будується нова форма, в нашому випадку це таблиця *"Зв'язки"* (рис. 39).

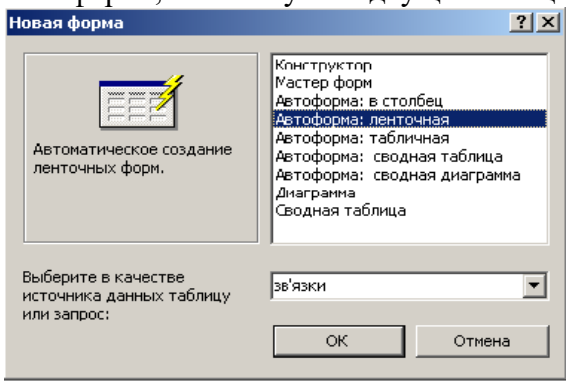


Рис. 39. Вікно для вибору таблиці та вибору типу автоформи

У верхній частині вікна вибирається тип автоформи *(автоформа - ленточная),* після чого потрібно натиснути *"ОК"*. На екрані з'явиться створена автоформа (рис. 40).

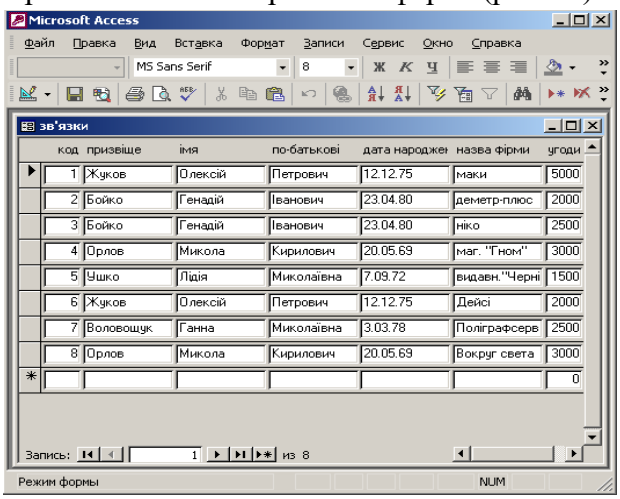


Рис. 40. Автоформа типу "ленточная" для таблиці "зв'язки"

3. Для побудови модифікаційного запиту на поновлення таблиці потрібно побудувати звичайний запит на вибірку записів, вибрати *"Запрос" "Обновление"*. У рядку *"Обновление",* який з'явився на бланку запитів, потрібно задати значення, яким будуть замінюватись існуючі в таблиці значення. У нашому випадку це значення потрібно побудувати у вікні *"Построителя выражений",* в якому побудувати вираз, що додає до значення поля *"сума угоди"* число 100 (рис. 41).

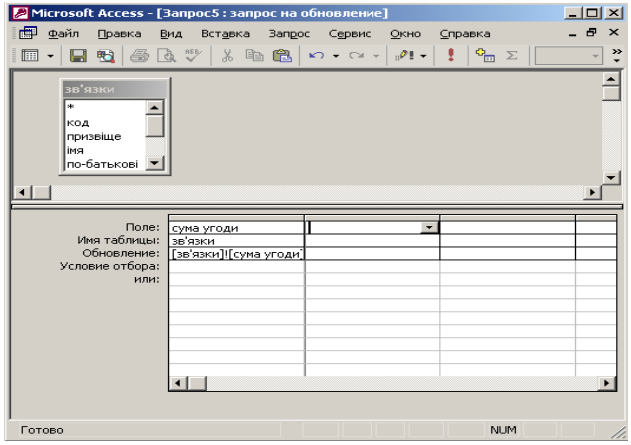


Рис. 41. Вікно конструктора запитів при побудові запиту на поновлення

4. Модифікаційний запит на видалення будується аналогічно: спочатку за допомогою звичайного запиту на вибірку потрібно відібрати ті записи, які потрібно видалити із таблиці, після цього вибрати *"Запрос"- Удаление"*. Після запуску такого запиту із таблиці будуть видалені вказані записи.

5. Для створення форми за допомогою майстра форм потрібно у вікні створення нової форми після вибору таблиці вибрати *"Мастер форм"* і натиснути "ОК". У вікні першого кроку майстра форм відбираються поля, які відобразяться у формі. Для цього потрібно в списку *"Доступные поля"* вибрати необхідне поле і натиснути кнопку “>”. Для відображення у формі всіх записів таблиці потрібно у вікні першого кроку роботи майстра форм натиснути кнопку “>>” . Перехід до наступного кроку роботи майстра здійснюється натисненням кнопки *"Далее"* (рис. 42).

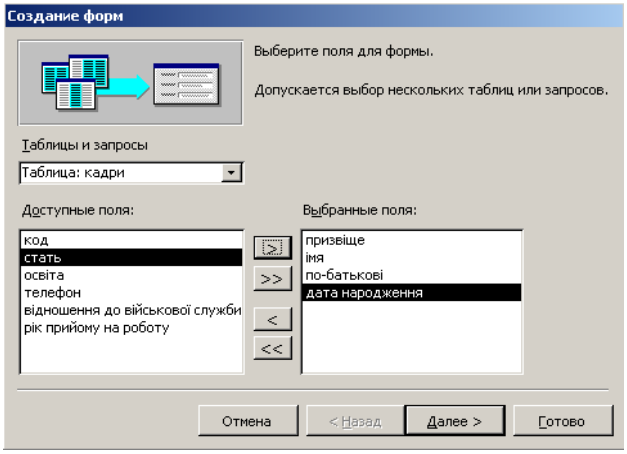


Рис. 42. Вікно першого кроку роботи майстра форм

На наступних кроках вибирається розташування полів у формі, стиль оформлення. На останньому кроці задається ім'я форми (рис. 43).

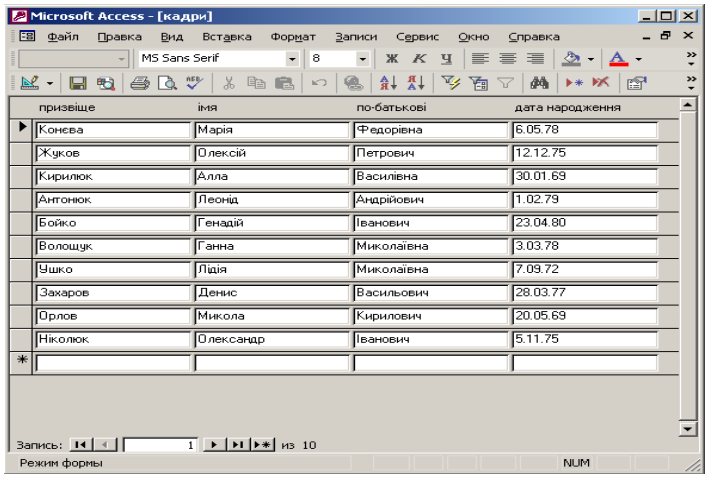


Рис. 43. Результат роботи майстра форм - створена форма

6. Для того, щоб в заголовок форми вставити надпис, потрібно перейти в режим конструктора форм, вибравши *"Вид" "Конструктор",* збільшити область заголовка форми, на панелі елементів управління вибрати об'єкт *"надпись",* натиснувши кнопку ***Aa,*** відмітити місце розташування напису і всередині надрукувати запропонований текст - ***"Дані про клієнтів"*** (рис. 44).

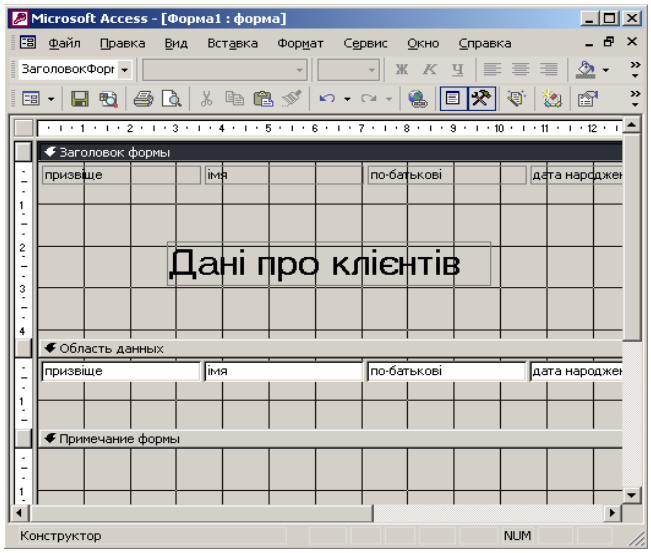


Рис. 44. Вікно конструктора форм із вставленим написом "Дані про клієнтів "

Якщо панелі елементів управління немає у вікні Access, то відобразити її можна, вибравши *"Вид "* - *"Панель элементов "*.

1. Для побудови форм на основі двох таблиць потрібно при запуску майстра форм вибрати одну з таблиць, яка є основною (таблицю ***"Кадри*** "), на першому кроці роботи майстра форм вибрати поля цієї таблиці, після чого в частині *"Таблицы и запросы"* вибрати другу таблицю, яка буде підлеглою (таблицю ***"Клієнти")*** (рис. 45), і нижче вибрати її поля.

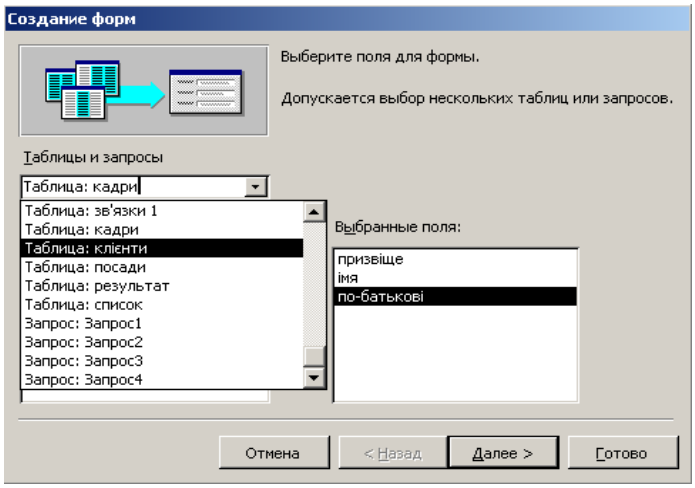


Рис. 45. Вибір другої таблиці при побудові форм на основі двох таблиць

У вікні другого кроку майстра форм вибирається спосіб представлення форм *"подчиненные"* або *"связанные"* (рис. 46)

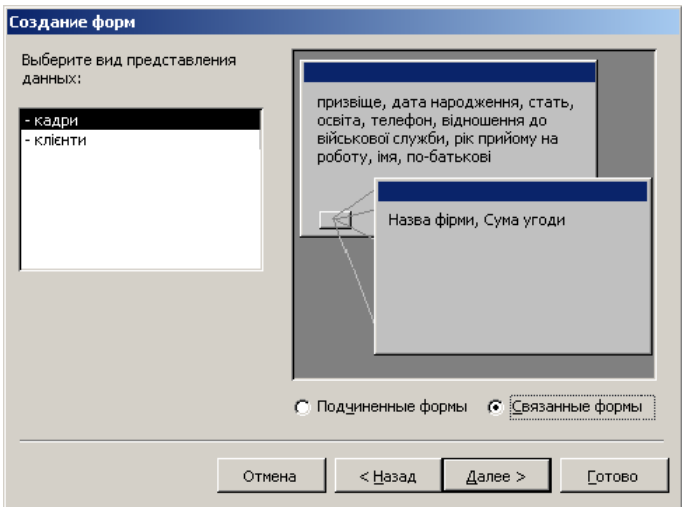


Рис. 46. Вибір способу представлення двох таблиць при побудові форм

На останньому кроці задаються імена форм. При відкритті таких форм на екрані відкривається основна форма, на якій знаходиться кнопка з іменем зв'язаної з нею форми, натиснення на яку відкриває вікно зв'язаної форми (рис. 47).

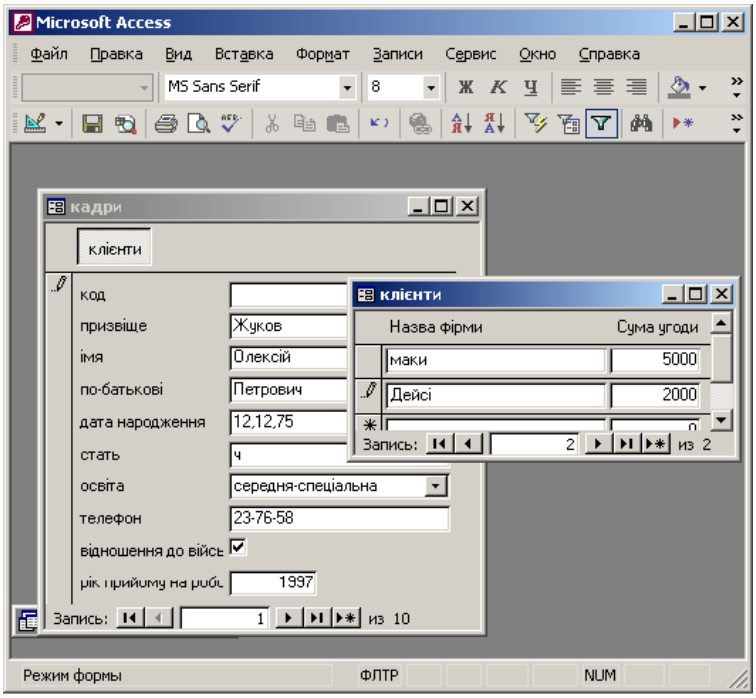


Рис. 47. Результат створення зв'язаних форм

.

ЛАБОРАТОРНА РОБОТА №5

Тема: Побудова звітів СУБД Microsoft Access

Методичні вказівки до виконання лабораторної роботи №5

І. Для створення звіту потрібно на вкладці *"Отчеты "* натиснути кнопку *"Создать"*. У вікні *"Новый отчет" у* нижній його частині вибирається таблиця, на основі якої будується звіт, у верхній частині спосіб створення звіту *"Мастер отчетов",* після чого натискається кнопка *"ОК"* (рис. 48).

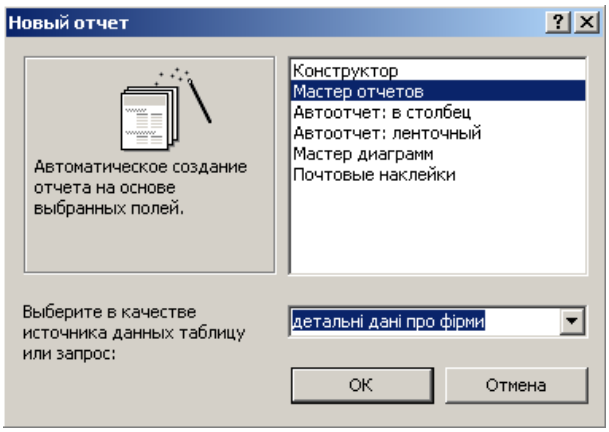


Рис. 48. Вікно першого кроку майстра звітів, в якому вибирається базова таблиця та спосіб створення звіту

На першому кроці роботи майстра звітів вибираються поля, які повинні бути представлені у звіті (аналогічно роботі другого кроку майстра форм). Перехід до наступного кроку здійснюється натисненням кнопки *"Далее"*. На другому кроці роботи майстра звітів задаються рівні групування даних. Для групування даних по деякому полю потрібно двічі натиснути на імені цього поля в списку полів звіту (рис. 49).

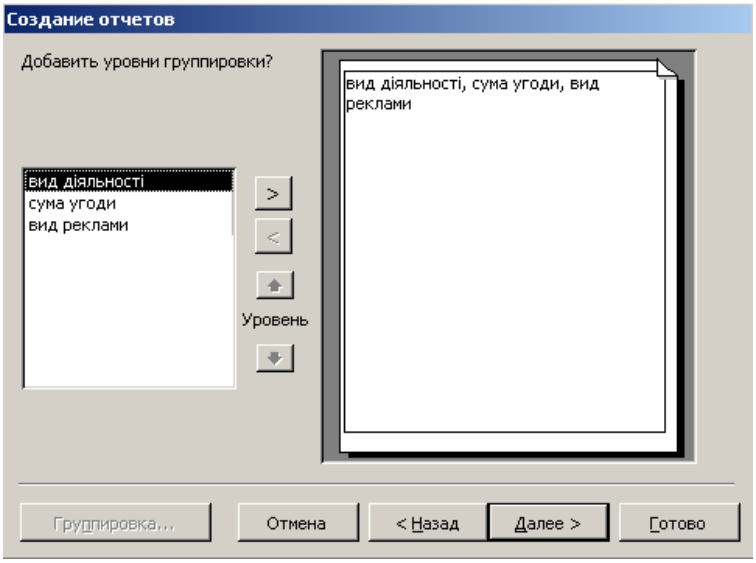


Рис. 49. Вибір поля для групування даних у звіті

На наступному кроці майстра звітів натиснення на кнопку *"Итоги"* дозволяє задати обчислення підсумкових значень у звіті. На наступному кроці вибирається вигляд макета для звіту та стиль оформлення звіту. На останньому кроці задається ім'я звіту. Після натиснення на кнопку *"Готово"* на екрані з'являється готовий звіт.

2. Для зміни підпису до поля у звіті потрібно перейти в режим конструктора звітів, вибравши *"Вид"- "Конструктор"*. Для вставки у звіт обчислювального поля потрібно скористатись кнопкою ab| панелі елементів управління, відмітити місцерозташування елемента управління на звіті. Після цього потрібно вивести вікно властивостей поля, натиснувши праву клавішу "мишки" на елементі

управління і вибравши пункт *"Свойства"*. У вікні властивостей поля на вкладці *"Данные"* натиснути кнопку поряд з полем ***"Данные"*** (рис. 51) для відображення вікна *"Построителя выражении"*.

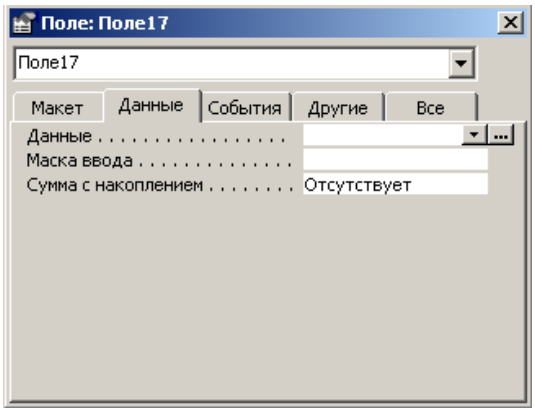


Рис. 51. Вікно властивостей поля

Завдання:

1. У базу даних **"Кадри"** додати таблицю ***"Детальні дані про фірми"*** і заповнити її інформацією відповідно до свого варіанта. Поле ***"Вид діяльності"*** визначити з використанням майстра підстановок із значень *торгівля, виробництво* та *інші послуги;* поле ***"Вид реклами"*** - майстер підстановок із значень *газети, радіо, телебачення, біг-борди.*
2. Створити звіт на основі таблиці ***"Детальні дані про фірми",*** в якому відобразити такі поля таблиці: ***"Назва фірми", "Вид діяльності", "Сума угоди", "Вид реклами".*** Згрупувати звітні дані по полю ***"Назва фірми".*** Обчислити підсумкове значення суми угоди по кожній групі записів та по всіх записах звіту. Настроїти поля для розміщення їх на одній сторінці. У вікні конструктора звітів видалити підпис підсумкового значення "Итоги", залишивши лише значення поля. Як підпис до поля задати текст **"Sum".** Назвати звіт **"підсумок по фірмах".** Роздрукувати звіт.
3. Створити звіт на основі таблиці ***"Детальні дані про фірми",*** в якому відобразити такі поля таблиці: ***"Вид реклами", "Назва фірми", "Вид діяльності", "Наявність філіалів", "Кількість філіалів".*** Згрупувати звітні дані по полю ***"Видреклами".*** Задати макет звіту **"структура".** Настроїти поля для розміщення їх на одній сторінці. Назвати звіт **"дані по видах реклами".** Роздрукувати звіт.
4. Створити звіт на основі таблиці ***"Детальні дані про фірми",*** в якому відобразити такі поля таблиці: ***"Вид реклами", "Назва фірми", "Сума угоди", "Річний прибуток (тис. грн)".*** Згрупувати звітні дані спочатку по полю ***"Назва фірми",*** потім - по полю ***"Сума угоди",*** після чого по полю ***"Річний прибуток (тис. грн)".*** Настроїти поля для розміщення їх на одній сторінці. Назвати звіт **"дані у відсотках прибутку".** Додати у звіт обчислювальне поле ***"Відсоток прибутку",*** яке обчислити поділивши суму угоди на річний прибуток і помножити на 10. Роздрукувати звіт.
5. Створити звіт на основі таблиці ***"Детальні дані про фірми",*** в якому відобразити такі поля таблиці: ***"Вид реклами", "Назва фірми", "Кількість філіалів", "Сума угоди".*** Згрупувати звітні дані спочатку по полю ***"Вид реклами",*** потім - по полю ***"Назва фірми".*** Настроїти поля для розміщення їх на одній сторінці. Назвати звіт **"сума по видах реклами".** Додати у звіт у розділ **"примечание отчета"** обчислювальне поле ***"Загальна сума",*** в якому обчислити загальну суму угод по всіх фірмах. Роздрукувати звіт.

Лабораторна робота №6

Тема: Встановлення сервера баз даних Borland InterBase. Адміністрування сервера. Створення баз даних та таблиць.

1. Основи використання СУБД InterBase/FireBird

1.1. Встановлення InterBase на платформі Windows

InterBase встановлюють як на сервер, так і на робочу станцію, на якій має працювати прикладна програма. Завдяки своїй невимогливості до ресурсів робота **InterBase** на комп'ютері не приводить до помітного зниження швидкодії. Коли **InterBase** не обслуговує під'єднань до баз даних, очікуючи запитів, то займає пам'яті менше, ніж такі популярні програми, як **ICQ** або **WinAmp**.

Перш ніж встановлювати **InterBase**, слід вирішити, який з його клонів (варіантів версій) буде вибрано для роботи. Є три основних клони цього сервера – **FireBird**, **InterBase** та **Yaffil**. Всі вони виникли на основі відкритого коду **InterBase 6.0**.

Клон **Firebird** є безплатним продуктом з відкритим кодом (**open source**). У його створенні брали участь колишні співробітники **InterBase Software Corporation**. Навіть версія **Firebird 1.0** підійде для повнофункціональної роботи з сервером . Компанія **Borland** наприкінці 2001 року випустила версію **InterBase 6.5**. Проте ця версія є платною (як і наступні), тому вільно можна користуватися лише обмеженою в часі (протягом 90 днів) так званою тріал-версією. Її можна скачати із сайту компанії **Borland** . Крім **Firebird** та **InterBase**, існує ще й російська вітка **InterBase 6.x** – **Yaffil**, що виник на основі **Firebird** в 2001 році. **Yaffil** також має всі головні параметри функціональності сервера **InterBase**.

Слід зазначити, що всі клони **InterBase 6.x** сумісні між собою і базу даних з одного сервера можна легко перевести під іншу за допомогою процесу **backup/restore**.

Різні варіанти дистрибутивів представлені, зокрема, на сайті http://www.ibase.ru. Там же можна отримати посилання на найновіший дистрибутив всіх клонів **InterBase**, а також іншу корисну інформацію з використання **InterBase** і розробки прикладних програм для баз даних.

Безпосередньо з першоджерела дистрибутиви й вихідні коди **InterBase** можна взяти на сайті http://sourceforge.net/projects/InterBase, а дистрибутиви **Firebird** можна знайти за наступним посиланням: http://sourceforge.net/projects/Firebird.

Інсталятор (програма, що займається встановленням продукту) будь-якого клону **InterBase** займає не більше 10 Мбайт. Для встановлення сервера слід просто запустити файл установки.

Якщо на комп'ютері, куди встановлюється **InterBase**, вже є якась його версія, то її спочатку необхідно зупинити. Для цього можна скористатися або "Панеллю керування" **Windows**, апплетом "Служби", якщо встановлення йде на **Windows NT/2000/XP**, або іконкою **InterBase**-сервера на панелі задач, якщо робота проводиться на **Windows 95/98/Me**.

Після запуску інсталятора з'явиться повідомлення, що міститиме дані , зокрема, про версію продукту. Для переходу до наступного кроку слід натиснути кнопку **Next**. На екрані з'явиться текст умов поширення продукту – **InterBase Public License**. Щоб перейти до наступного етапу встановлення, слід вибрати пункт "**I agree**" (що означає згоду з приведеними умовами поширення). З'явиться вікно, у якому пропонується вибрати шлях для встановлення **Firebird**. Це важлива опція. Звичайно за замовчуванням інсталятор пропонує шлях **C:\Program Files\InterBase** (або **C:\Program Files\FireBird**). Проте часто є сенс перевизначити цей шлях.

Вибравши шлях для встановлення, слід натиснути **Next** для переходу до наступного кроку інсталяції. При цьому на екрані з'явиться вікно, що містить компоненти, необхідні для роботи. Після цього починається встановлення вибраних компонентів на комп'ютер.

На ОС **Windows 95/98/Me** працюючий **InterBase** відобразиться на панелі задач у вигляді значка серед системних іконок, а на ОС **Windows NT/2000/XP** сервер запуститься як служба (**service**) і на екрані ніяк його функціонування не відобразиться – спостерігати за його статусом та управляти роботою можна лише через апплет "**InterBase Manager**" (або "**FireBird Server Manager**") панелі керування або через вікно "Служби" ("**Services**").

Для лабораторних робіт даного курсу використовується сервер **FireBird 1.0.x.**

InterBase завжди поставляються засоби адміністрування командного рядка. Це потужні засоби, які необхідні для професійного керування роботою сервера. Проте багато користувачів вважають зручнішим використовувати інструменти з графічним інтерфейсом. Зокрема, разом з **InterBase** поширюється така утиліта адміністрування, як **IBConsole.** Крім нього, можна користуватися більш зручними програмами сторонніх розробників. В подальшому для роботи з сервером буде описуватися стандартна програма адміністрування **IBConsole.**

1.2. Реєстрація сервера

Для створення бази даних використаємо програму **IBConsole**. З її допомогою можна створити базу даних як на локальному комп'ютері , так і на віддаленому комп'ютері у мережі, на якому працює сервер **FireBird**. Тому спочатку слід зареєструвати для програми сервер, з яким вестиметься робота. Якщо він вже зареєстрований, його просто можна вибрати із списку в лівій частині вікна (рис. 1.1). Операція по реєстрації сервера здійснюється один раз. В подальшому програма пам'ятатиме всі введені параметри, необхідні для з'єднання з сервером.

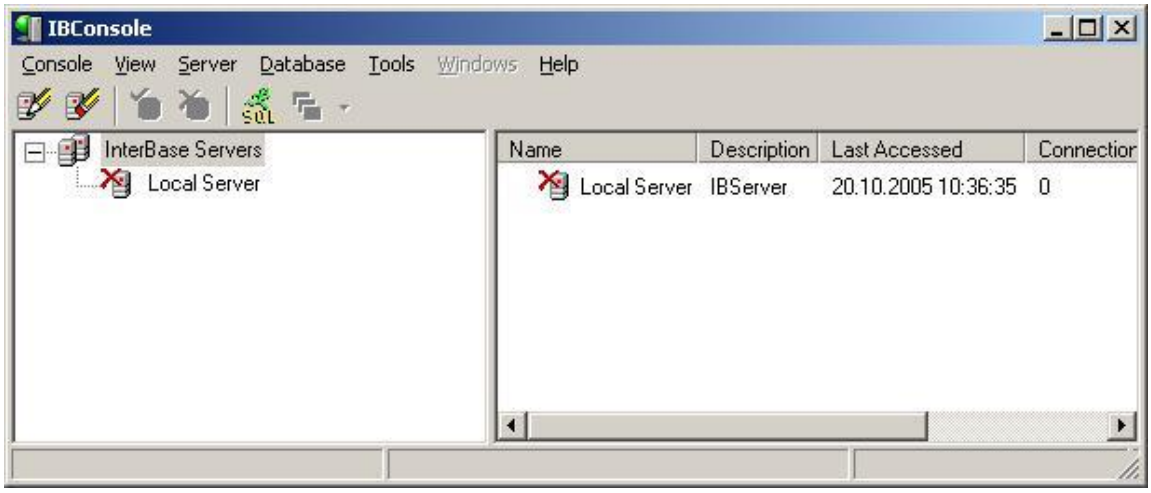


Рис. 1.1. Головне вікно утиліти **IBConsole.**

Для реєстрації нового сервера слід вибрати команду меню
реєстрації (рис. 1.2) слід вказати наступні параметри:

Вид під'єднання до сервера. Під'єднання може бути локальним (**Local Server**) або віддаленим (**Remote Server**).

1. Ім'я сервера (**Server Name**) – це назва комп'ютера в мережі, на якому встановлений сервер **FireBird** (якщо під'єднання віддалене).
2. Мережевий протокол, що використовується для з'єднання з сервером (**Network Protocol**). За замовчуванням використовується **TCP/IP**, тому як назву сервера можна вказувати пряму **IP**-адресу комп'ютера із встановленим **FireBird**. Якщо робиться локальне під'єднання до сервера, то протокол можна не вказувати. Слід зауважити, що на комп'ютері повинні бути інстальовані програмні компоненти, що підтримують вказаний протокол.
3. Псевдонім зареєстрованого сервера (**Alias Name**) – його назва, що відображуватиметься в даній програмі.
4. Опис сервера (**Description**) – опис сервера (поле не обов'язкове для заповнення).

5. Назва користувача, який під'єднується до сервера (**User Name**), та його пароль (**Password**). Так як для повнофункціональної роботи з **СУБД** слід під'єднуватися як адміністратор, то варто тут

вказати назву користувача-адміністратора бази даних. При встановленні **FireBird** завжди створюється користувач **SYSDBA**, і це ім'я не можна змінювати. Даний користувач має повний контроль над сервером. За замовчуванням його пароль – "**masterkey**" [1]. Проте при професійній роботі для обмеження несанкціонованого доступу до сервера необхідно змінити пароль для адміністратора **SYSDBA** на інший. Назва користувача нечутлива до регістру символів, пароль – чутливий.

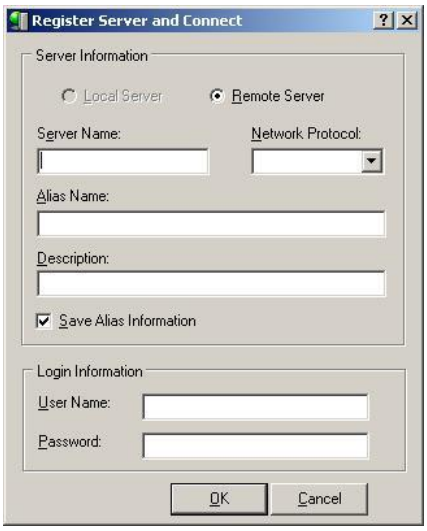


Рис. 1.2. Вікно реєстрації сервера.

Якщо сервер був зареєстрований раніше, то для під'єднання до нього слід двічі клацнути мишкою на його піктограмі. При цьому програма перепитає назву користувача і пароль того, хто робить спробу під'єднання до сервера.

Після з'єднання з сервером його вітка списку в головному вікні (рис. 1.1) розкриється. Якщо зв'язок із сервером встановлено, з ним можна працювати, зокрема, створити базу даних.

1.3. Створення бази даних

Для створення нової бази даних слід вибрати команду меню "**Database\Create Database**". З'явиться діалогове вікно створення бази даних (мал. 1.3). У ньому необхідно задати наступні параметри:

з Назву файла, у якому міститиметься база даних (поле **File(s)**). База даних **FireBird** розміщується у єдиному файлі, при цьому в нього поміщуються всі таблиці. Часто цьому файлу дають розширення ***.gdb** (для **InterBase**) або ***.fdb** (для **FireBird**), хоча його можна задати будь-яким. Базу можна розбити на кілька частин, розмістивши їх в окремих файлах. Тоді назви та розміщення цих фалів слід вказати у полі **File(s)** (стовпчик **Filename(s)**). У випадку багатофайлової бази даних для всіх файлів, крім останнього, слід задати максимальний розмір у сторінках (стовпчик **Size (Pages)**). Для однофайлової БД розмір не вказують. Керувати видом інформації, яка поміщуватиметься у тому чи іншому файлі, може лише сервер. Файл БД, що створюється, може міститися тільки на тому комп'ютері, на якому працює сервер.

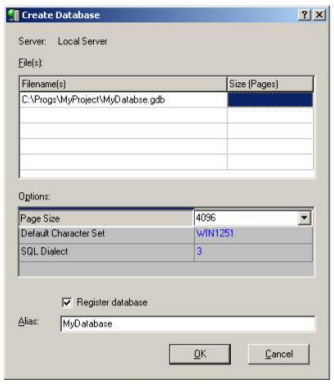


Рис. 1.3. Вікно створення бази даних.

- Розмір сторінки бази даних. Вибір цього параметра дуже важливий для забезпечення ефективної роботи сервера **FireBird** з базою даних. Файл бази даних розбивається на сторінки фіксованого розміру, і всі звертання до диска, які виконує **FireBird**, зчитують і записують інформацію посторінково. Вибирати розмір сторінки слід не менше **4096** байт. При малому розмірі сторінки записи великої довжини у базі займатимуть кілька сторінок, і для читання такого запису **FireBird** буде змушений здійснити кілька звертань до диска, що спричинить зменшення швидкодії. Змінити заданий при створенні розмір сторінки можна пізніше, але для цього слід буде створити резервну копію бази, а потім відновити базу з цієї резервної копії. Рекомендації з вибору розміру сторінки наступні:

  Для дискових накопичувачів з файловою системою **NTFS** вибирають розмір сторінки, рівним **4096** байтам. Але варто переконатися, що розмір кластера в **NTFS**-диска також встановлений в **4096** байт.

  Для дисків з **FAT32** встановлюють розмір сторінки **8192** або **16384** байта (слід відмітити, що розмір сторінки **16384** байт є не у всіх клонах **InterBase**).
- Кодування символів у рядках. Цей параметр також дуже важливий. Кодування визначає, символи якого національного алфавіту будуть використовуватися в базі даних за замовчуванням. Якщо в процесі роботи планується працювати лише з українською (російською) та англійською мовою, то найкраще встановити кодування **WIN1251**. Якщо ж необхідно зберігати інформацію в базі на різних мовах, то можна залишити значення кодування як **None**, а вже в налаштуваннях драйверів для бібліотеки доступу визначати необхідні опції для підтримки тієї або іншої мови. Задане кодування визначає набір використовуваних символів тільки за замовчуванням. Під час створення таблиць чи інших об'єктів бази даних можна встановлювати інший набір символів, вказавши його явно. Для більшості прикладних програм баз даних, що створюються для україномовних (російськомовних) користувачів, оптимальним є кодування **WIN1251** для всієї бази даних.
- Діалект бази даних. Ця властивість може приймати тільки два можливих значення: **1** або **3.** Діалект **1** і **3** відрізняються один від одного наступним:
  - У Діалект **3** дозволяє використовувати розширений набір типів даних, таких, як типи для роботи з великими цілими числами, типи для роботи з датою й часом – **DATE** і **TIME.**
  - У Діалект **3** розрізняє регістр ідентифікаторів, якщо ідентифікатор оточений подвійними лапками. Так, записи **Table1** й **TABLE1** в обох діалектах будуть рівнозначні, а записи "**Table1**" і "**TABLE1**" будуть інтерпретовані як різні.
  - У Діалект **3** не підтримує неявного приведення типів даних (як це було в Діалекті **1**). Так, у Діалекті **1** вираз **'25'+5** буде коректним і його результатом буде **30**. У Діалекті **3** він викличе помилку невідповідності типів.

  Крім цього, є ще ряд відмінностей. Наприклад, **Borland Database Engine (BDE)** аж до версії **5.2** погано працює з **3-** м діалектом – виникають проблеми з підтримкою нових типів даних. Вибір **SQL Dialect**, у якому буде створюватися база даних, важливий також з тієї причини, що перехід між різними діалектами – заняття досить нетривіальне й трудомістке, якого по можливості варто уникати. В загальному випадку треба керуватися наступними правилами:

  - У вибирають Діалект **3**, якщо проектується база даних для програми, що використовуватиме тільки сучасні бібліотеки прямого доступу до **FireBird**, які повністю підтримують Діалект **3**;

У вибирають Діалект **1**, якщо важливою є сумісність з більш ранніми бібліотеками доступу до

Крім діалектів **1** та **3** існує ще діалект **2**, але він використовується як проміжний етап при міграції з Діалекту **1** на Діалект **3.**

5) Псевдонім для бази даних (**Alias**) – назва, що використовуватиметься для ідентифікації бази даних. За цією назвою БД буде присутня в списку у вікні рис. 1.1.
6) Якщо відмітити перемикач **Register database**, то БД буде автоматично зареєстрована у програмі і відразу доступна для роботи.

Якщо база даних була створена раніше і не зареєстрована, то для роботи з нею її слід зареєструвати. Для цього слід вибрати команду меню **Database\Register**. У вікні реєстрації (рис. 1.4) слід вказати такі параметри:

У Шлях до файла бази даних та його назву (поле "**File**" [2]). Його можна вибрати за допомогою стандартного діалогового вікна вибору файла, якщо натиснути кнопку "...".
У Псевдонім БД (поле **Alias Name**).
У Назву користувача, який під'єднується до БД. Він буде власником створюваної бази даних (**OWNER**). Це дає йому право повністю управляти нею – створювати й видаляти різні об'єкти, виконувати запити на вибірку й зміну даних тощо. Як правило, для створення бази використовують права адміністратора **FireBird** – **SYSDBA**, так як цей користувач реалізує функції системного адміністратора бази даних. За замовчуванням **SYSDBA** може змінювати всі об'єкти бази даних, незалежно від того, ким вони створені – самим **SYSDBA**, чи іншим користувачем.
У Пароль користувача (поле **Password**).
У Роль, під якою з'єднується з базою користувач (можна не вказувати). Ролі використовуються для того, щоб на рівні бази даних розмежувати для різних груп користувачів доступ до інформації, яка зберігається в базі.
У Множина символів, яка використовуватиметься для інтерпретації рядкових значень.

Після реєстрації база даних буде доступна для роботи в утиліті **IBConsole.**

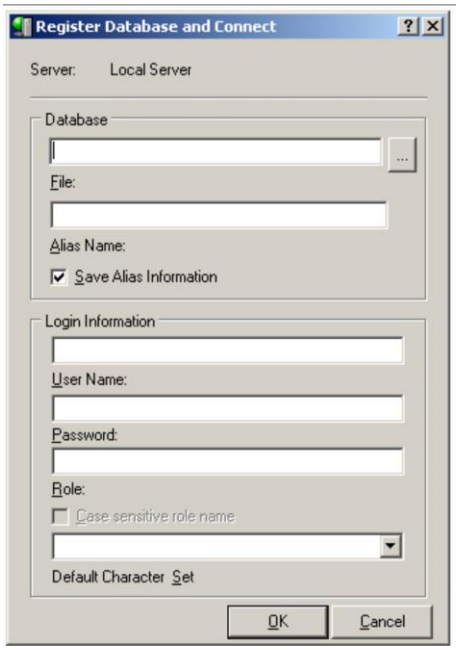


Рис. 1.4. Вікно для реєстрації бази даних.

Утиліта **IBConsole** дає зручний інтерфейс для створення бази даних, проте здійснення цієї операції "вручну" не набагато складніше. Для цього слід створити два файли. Перший з них – це файл із командами **SQL** (його називають файлом скрипта **SQL**), який створить базу даних, другий – командний файл **WINDOWS**, що передасть створений **SQL**-файл на виконання стандартній утиліті **isql.exe**, яка його виконає. Наприклад, вид цих двох файлів може бути наступний.

Файл скрипта **crebase.sql:**

```
SET SQL DIALECT 3; SET NAMES WIN1251;
CREATE DATABASE 'Localhost:C:\Database\my.gdb'
USER 'SYSDBA' PASSWORD 'masterkey'
PAGE_SIZE 16384
DEFAULT CHARACTER SET WIN1251;
```

Командний файл **runscr.bat**:

```
"C:\FireBird\Bin\isql.exe" -i "C:\temp\crebase.sql"
```

Звичайно, у цих файлах повинні бути прописані реальні шляхи.

2. Синтаксис мови SQL для визначення структури БД

2.1. Типи даних в SQL

Типи даних – це базові елементи будь-якої мови програмування та будь-якого сервера **СУБД**, в тому числі й **FireBird**. Кожен тип даних має набір операцій, які можна виконувати над значеннями цього типу, тому при проектуванні бази необхідно правильно вибрати тип даних, що допоможе уникнути багатьох проблем при розробці клієнтських програм.

В **FireBird** існує 12 типів даних, які здатні задовольнити практично будь-які потреби розроблювача. Ці типи умовно розділяються на 6 наступних груп:

- для зберігання цілих чисел – **INTEGER** та **SMALLINT**;
- для зберігання дійсних чисел (чисел з плаваючою комою) – **FLOAT** та **DOUBLE PRECISION**;
- для чисел з фіксованою точністю – **NUMERIC** та **DECIMAL**;
- для зберігання дати, часу й дати/часу – **DATE, TIME** та **TIMESTAMP**;
- для зберігання символьних рядків – **CHARACTER** (скорочено **CHAR**) і **VARYING CHARACTER** (**VARCHAR**);
- Для зберігання великих масивів даних змінної довжини – **BLOB.**

Також можливо визначати масиви значень елементарних типів, тобто всіх перерахованих типів, крім **BLOB.**

Більшість типів даних **FireBird** відповідають типам, наявним у стандарті **SQL92**, проте є й відмінні типи, зокрема, масиви та тип **BLOB.**

Масиви в **FireBird** можуть містити множину даних певного типу в одному полі, наприклад можна визначити масив значень типу **INTEGER**. Масиви можуть бути багатовимірними. Тип даних **BLOB** може динамічно змінювати свою довжину. Його назва є абревіатурою від фрази **Binary Large OBject** – "великий двійковий об'єкт". Треба сказати, що **BLOB** – це винахід розроблювачів **InterBase**, який пізніше поширився й прижився у всіх сучасних **SQL**-серверах.

Цілі типи

До цілих типів відносяться **SMALLINT** та **INTEGER**. Тип **SMALLINT** є вкороченою версію **INTEGER** і має довжину 2 байти, на відміну від 4 байт типу **INTEGER**. Так як використання типу **SMALLINT** суттєвого заощадження дискового простору не дасть, то загальною рекомендацією є використовувати для зберігання цілих значень тип **INTEGER.**

Область застосування цілих типів очевидна: вони потрібні для полів, що містять тільки цілі числа –зберігання лічильників, кількості неподільних предметів і т.д. Як правило, тип **INTEGER** мають також поля, що містять первинні ключі.

Дійсні типи даних

До дійсних типів (їх ще називають типами чисел із плаваючою комою) належать **FLOAT** і **DOUBLE PRECISION.** Варто зазначити, що найчастіше використовувати тип **FLOAT** недоцільно – його точність недостатня для зберігання більшості значень. Особливо не рекомендується зберігати в ньому грошові величини – у змінних типу **FLOAT** дуже швидко наростають помилки округлення, що при значній кількості даних та операцій над ними приводить до помітних похибок. Тому, якщо в базі даних передбачається зберігати числа із плаваючою комою (наприклад, у бухгалтерських системах або в системах для наукових розрахунків), то кращим вибором буде тип **DOUBLE PRECISION.**

В 3-му діалекті **SQL** для зберігання грошових величин існує механізм зберігання типів з фіксованою крапкою довжиною 64 біти. Використання цих типів забезпечує найкращу точність.

Типи даних з фіксованою комою

До цих типів даних належать **NUMERIC** та **DECIMAL**. Обидва типи мають однакову розрядність – від 1 до 18 знаків, та однакову точність – від нуля до розрядності [3]. В сучасних версіях **FireBird (InterBase)** ці типи є синонімами.

Типи для зберігання дати й часу

В **FireBird** існує 3 типи для зберігання дати й часу – це **DATE, TIME** і **TIMESTAMP.**

Тип **DATE** зберігає дати з точністю до дня. Діапазон можливих значень – від 1 січня 100 року н.е. до 29 лютого 32768 року. Тип **TIME** зберігає дані про час з точністю до десятитисячної долі секунди. Діапазон можливих значень – від 00:00 AM до 23:59.9999 PM. Тип **TIMESTAMP** є комбінацією типів **DATE** й **TIME.**

Типи даних для зберігання тексту

В **FireBird** існує два типи, призначених для зберігання текстової інформації – **CHAR** та **VARCHAR.** Повні їхні назви – **CHARACTER** та **CHARACTER VARYING**, проте немає ніякої причини користуватися довгими іменами.

Щоб визначити поле або змінну символьного типу, необхідно в дужках після імені типу або вказати число символів, що буде використовуватися в стовпчику, або опустити число символів – при цьому буде створене поле з довжиною 1 символ. Наприклад:

```
Field1 CHAR(200),
Field2 CHAR
```

Поле **Field1** буде мати довжину 200 символів, а **Field2** – 1 символ.

Типи **CHAR** і **VARCHAR** багато в чому схожі – вони можуть містити до 32768 символів, однак є
й відмінності. Хоча зберігаються ці два типи в базі даних однаково, але працює з ними **FireBird** по-різному. Значення у полі типу **CHAR** завжди має однакову довжину, задану при створенні таблиці. Якщо довжина рядка, що зберігається в такому полі, менша від заданої, то він доповнюється пробілами. Рядок типу **VARCHAR** завжди має той вид, який він мав при записі. Так, якщо у якійсь таблиці є два поля: **NAME1** типу **CHAR(15)** та **NAME2** типу **VARCHAR(15)**, і в них записати значення слова **"Святослав"**, то при звертанні до цих полів сервер поверне відповідно значення **"Святослав"** та **"Святослав"**.

Поведінка з рядками, притаманна типу **CHAR**, може бути корисною дуже рідко (а недоліки такого підходу видно відразу, зокрема, – значне збільшення мережевого трафіка), тому загальним правилом є використання типу **VARCHAR.**

Однією з найважливіших характеристик символьного типу є його набір символів – **CHARACTER SET.** Набір символів визначається для всієї бази даних і використовується за замовчуванням для всіх символьних полів, якщо не перевизначається явно при створенні поля.

Щоб створити символьне поле з явною вказівкою набору символів, необхідно в описі стовпця помістити опис набору символів. Для підтримки кирилиці, як правило, використовується набір символів **WIN1251**. Ось приклад опису символьного поля з явно заданим набором символів **WIN1251**:

```
Field1 VARCHAR(200),
Field2 VARCHAR(200) CHARACTER SET WIN1251
```

Тут **Field1** – поле без явної вказівки набору символів, тому для нього буде використовуватися той набір символів, що був зазначений при створенні бази даних. Для поля **Field2** явно визначено, що в ньому будуть зберігатися символи в кодуванні **WIN1251**.

Тип даних BLOB

Тип даних **BLOB** призначений для зберігання великої кількості даних змінного розміру. Він дозволяє зберігати дані, які не можуть бути поміщені в полях інших типів, – наприклад, малюнки, аудіофайли, відеофрагменти і т.д.

Реалізація полів таких типів всередині бази даних значно відрізняється від реалізації раніше розглянутих полів. Не **BLOB**-поля розташовані на сторінці даних поруч один з одним, а у випадку **BLOB** на сторінці даних зберігається тільки ідентифікатор **BLOB**, а самі дані розташовуються на спеціальній сторінці. Саме така організація даних дозволяє зберігати дані нефіксованого розміру.

У типу **BLOB** є можливість визначати набір декількох підтипів і спеціальних процедур, що називаються фільтрами (**BLOB filters**) для роботи із цими підтипами. Існує кілька визначених підтипів **BLOB**, які вмонтовані в **FireBird**. Всі ці підтипи мають невід'ємні номери, наприклад **subtype 0** – це дані невизначеного типу, **subtype 1** – текст, **subtype 2** – **BLR** (**Binary Language Representation**) і т.д.

Користувач також може визначати свої підтипи **BLOB**, які можуть мати від'ємні значення. Кожному типу може бути поставлений у відповідність фільтр, що перетворить поле цього підтипу в інший підтип.

Використання **BLOB**-полів служить альтернативою зберіганню зовнішніх щодо бази даних файлів. Що стосується фільтрів **BLOB**, то вони використовуються досить рідко через свою орієнтацію на вузький клас завдань.

Масиви

СУБД InterBase була однією з перших, у якій з'явилися масиви. Підтримка масивів у базі даних
є розширенням традиційної реляційної моделі. Наявність масивів дозволяє спростити роботу з множинами даних одного типу. Масив – це сукупність значень одного типу, що має загальне ім'я й дозволяє звернутися до будь-якого елемента масиву за його номером. Масиви в **FireBird** можуть бути одномірними й багатомірними.

Так, поле типу "масив чисел **INTEGER**", описується так:

```
myOneDimArray INTEGER[12],
myTwoDimArray INTEGER[5,4],
myThreeDimArray INTEGER[2,10,8]);
```

Поле **myOneDimArray** містить одномірний масив довжиною 12 чисел, поле **myTwoDimArray** містить двовимірний масив (матрицю) 5x4 цілих чисел, і поле **myThreeDimArray** – тривимірний масив 2x10x8. При такому описі елементи масиву нумеруються, починаючи з одиниці, тобто перший елемент має номер 1, другий – номер 2 і т.д. Якщо необхідно вказати границі масиву самостійно, наприклад з 0 до 5, то слід використати такий синтаксис:

```
myArray INTEGER[0:5]
```

Масиви реалізовані на базі полів типу **BLOB**, тому багатомірний масив буде розміщений на окремих сторінках, щоб оптимізувати операції вводу-виводу в цих полях.

Хоча масиви і надають зручний механізм для зберігання однотипних об'єктів, але в більшості випадків множинні дані зберігають в підлеглих таблицях.

2.2. Таблиці

Як і в усіх реляційних **СУБД**, дані в **InterBase** зберігаються у вигляді таблиць. Таблиця обов'язково має ім'я, унікальне в межах однієї бази даних. Таблиці є основним вмістилищем інформації в базі даних, тому завдання по створенню таблиць є важливою складовою у проектуванні програмних засобів по роботі з **БД**.

Розглянемо синтаксис **SQL** щодо створення таблиці:

```
CREATE TABLE table [EXTERNAL [FILE] 'filespec']
  (<col_def> [, <col_def> | <tconstraint> …]);
```

Тут **table** – ім'я створюваної таблиці, **<col_def>** – опис стовпців (полів) створюваної таблиці. Опція **[EXTERNAL [FILE] "<filespec>"]** означає, що буде створена так звана зовнішня таблиця, що зберігається не в загальному файлі бази даних, а в окремому файлі з ім'ям **<filespec>**. Зовнішні таблиці слід створювати лише в таких випадках:

- Якщо вони повинні містити дані із зовнішнього джерела, наприклад, дані з файлів, що управляються іншими операційними системами або іншими, не зв'язаними з **СУБД** прикладними програмами.
- Якщо є потреба перетворити дані із зовнішнього файлу до існуючої **FireBird**-таблиці.

Синтаксис створення стовпця описується наступним синтаксисом:

```
<col_def> = col {<datatype> | COMPUTED [BY] (<expr>) |
 domain} [DEFAULT {literal | NULL | USER}]
        [NOT NULL]
        [<col_constraint>]
        [COLLATE collation]
```

Стандарт **InterBase** допускає різноманітні опції при визначенні даних таблиці, проте лише невелика частина наведених у визначенні стовпця опцій є обов'язковою. Кожен стовпчик у таблиці повинен мати ім'я, унікальне в межах таблиці, а також або тип даних, або вираз для обчислення значення стовпця (для обчислюваних стовпців), або домен.

Для створення таблиці за допомогою мови визначення даних **FireBird** використаємо раніше розглянутий інструмент – утиліту **IBConsole**. Для цього слід під'єднатися до потрібної бази даних і вибрати команду меню **Tools\Interactive SQL** (або натиснути відповідну кнопку на панелі інструментів). При цьому з' явиться вікно, за допомогою якого в інтерактивному режимі можна виконувати **SQL**-запити (рис. 1.5).

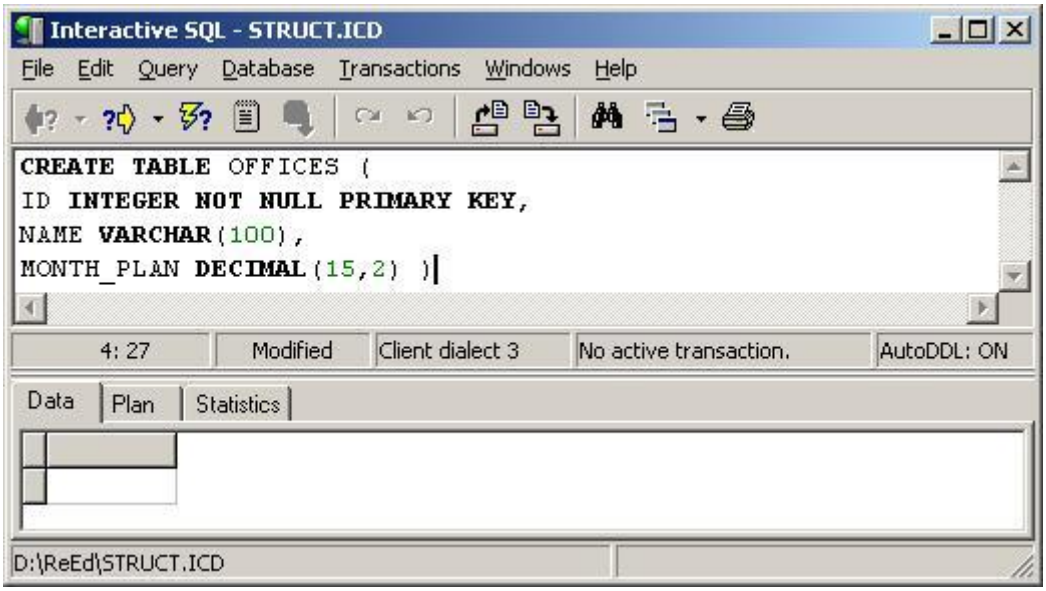


Рис. 1.5. Інтерактивний **SQL**.

У верхній частині цього вікна слід ввести потрібний **SQL**-запит, після чого натиснути комбінацію клавіш **Ctrl+Enter** (або кнопку "**Execute Query**" на панелі інструментів).

Наприклад, для створення таблиці **OFFICES** можна використати такий синтаксис:

```
CREATE TABLE OFFICES (
ID INTEGER NOT NULL PRIMARY KEY,
NAME VARCHAR(100),
MONTH_PLAN DECIMAL(15,2) )
```

Можливі й інші способи визначення полів. Так, можна задати тип поля, використовуючи домен. Домен – це тип, заданий користувачем (на основі існуючих типів) для зручності застосування певних сполучень параметрів. Наприклад, можна визначити домен **D_ID** для задання полів ідентифікаторів. Визначивши домен, можна скористатися ним для задання типу поля:

```
CREATE DOMAIN D_ID AS INTEGER CHECK (VALUE > 0);
CREATE TABLE Table_example (
ID D_ID,
NAME VARCHAR(80) ,
PRICE_1 DOUBLE PRECISION);
```

При цьому поле **ID** буде мати тип, обумовлений доменом **D_ID**. Таким чином, визначивши в домені тип поля, необхідні перевірки й обмеження, можна багаторазово застосовувати цей домен для створення полів однакового призначення.

Третій спосіб задати стовпчик у таблиці – це визначити його обчислюваним (**COMPUTED BY**) і задати умову, відповідно до якої буде обчислюватися його значення. Наприклад, інколи виникає потреба обчислити суму товару на основі ціни та кількості, що містяться в таблиці. Для цього можна використати наступний синтаксис:

```
CREATE TABLE GOODS (
ID INTEGER NOT NULL PRIMARY KEY,
NAME VARCHAR(80),
PRICE_IN DECIMAL(15, 2),
CNT DOUBLE PRECISION,
SUMM_IN COMPUTED BY (PRICE_IN*CNT)) ;
```

Слід зауважити, що в обчислюваному полі дані не зберігаються, але при кожному звертанні до нього обчислюється вираз, пов'язаний з полем, і результат видається у відповідь на запит.

Розглянемо опції, які можна задавати при створенні поля таблиці.

Опція **[DEFAULT {literal | NULL | USER}]** дозволяє задати значення стовпця за замовчуванням. Це дуже зручна можливість для автоматичного заповнення даних. Існує три можливості задавати значення за замовчуванням. Перша позначена як **literal** і дозволяє задавати значення за замовчуванням у вигляді текстових констант, чисел або дати. Наприклад, можна сформувати наступний вираз для створення стовпця з числовим значенням за замовчуванням:

```
MONTH_PLAN DECIMAL(15, 2) DEFAULT 100000
```

При цьому всі поля, що заносять у таблицю, будуть приймати значення за замовчуванням (в даному випадку **100000**), якщо при вставці нового запису не було задано іншого значення для цього поля.

Друга можливість задавати значення за замовчуванням – це вказати **DEFAULT NULL** у визначенні стовпця. При цьому в новостворюваних записах значення цього стовпця буде **NULL**. Нариклад:

```
PRICE_IN DOUBLE PRECISION DEFAULT NULL
```

Третій спосіб задати значення за замовчуванням – вказати **DEFAULT USER** у визначенні стовпця. При цьому в новостворених записах у це поле буде заноситися ім'я поточного користувача, тобто користувача, що встановив з'єднання з **InterBase** і виконав цю вставку.

Для деяких полів необхідно, щоб у них обов'язково містилося якесь значення (тобто ніколи не містилося значення **NULL**). Щоб на рівні бази даних задати таке обмеження, потрібно внести наступний модифікатор в опис стовпця:

```
NAME VARCHAR(50) NOT NULL
```

Як правило, обмеження **NOT NULL** поєднується з опцією **DEFAULT**, що гарантовано присвоює якесь значення цьому полю.

Але часто обмеження **NOT NULL** буває недостатньо. Наприклад, у випадку зберігання в базі даних цін очевидно, що вони не можуть приймати негативні значення. Щоб реалізувати таку перевірку на рівні сервера, слід визначити стовпчик таким чином:

```
PRICE_IN DOUBLE PRECISION CHECK (PRICE_IN>0)
```

Різні несуперечливі опції для перевірки можуть сполучатися, наприклад можна задати непусте значення й перевірку на позитивність:

```
PRICE_IN DOUBLE PRECISION NOT NULL CHECK (PRICE_IN>=0)
```

Інколи бувають випадки, коли необхідно змінити вже існуючу таблицю. Звичайно, можна перестворити таблицю в цілому. Для цього можна виконати команду видалення таблиці й знову створити її, наприклад так:

```
DROP TABLE OFFICES;

CREATE TABLE OFFICES (ID INTEGER...
```

Можна також використовувати команду **RECREATE TABLE**, синтаксис якої збігається з синтаксисом команди **CREATE TABLE.**

Але такі способи зміни таблиць мають значні недоліки. Зокрема, при цьому всі дані, які існували в таблиці, знищуються. Тому для легкої зміни структури таблиць існує команда **ALTER TABLE**, що дозволяє додавати нові поля, видаляти існуючі, а також додавати/видаляти обмеження посилальної цілісності.

Щоб додати в таблицю ще один стовпчик, використовується така команда:

```
ALTER TABLE OFFICES ADD Patronimic VARCHAR(50);
```

Після виконання цієї команди в таблиці **OFFICES** з'явиться новий стовпчик з назвою **Patronimic** і типом **VARCHAR(50)**. Щоб видалити з таблиці стовпчик з ім'ям **NameCol**, слід виконати

```
ALTER TABLE OFFICES DROP NameCol;
```

Щоб змінити стовпчик, використовується команда **ALTER TABLE <table> ALTER COLUMN.**

Зокрема, для зміни назви стовпця використовується така конструкція:

```
ALTER TABLE OFFICES ALTER COLUMN NAME TO FIRSTNAME;
```

2.2.1. Первинні ключі в таблицях

Первинний ключ – це одне або кілька полів у таблиці, за допомогою яких можна однозначно ідентифікувати записи в межах цієї таблиці. Значення первинного ключа є для кожного рядка унікальним в межах однієї таблиці. Таблиці можна створювати і без первинного ключа. Але правильно побудована реляційна таблиця завжди повинна його містити. Створити первинний ключ можна як при створенні таблиці, так і пізніше. Якщо, наприклад, на момент створення таблиці було вирішено, що первинним ключем буде поле **ID**, то додати первинний ключ можна так:

```
CREATE TABLE OFFICES (

ID INTEGER NOT NULL PRIMARY KEY,

NAME VARCHAR(80),
```

```
MONTH_PLAN DOUBLE PRECISION);
```

Всі поля, які входять до складу первинного ключа, повинні бути визначені з модифікатором **NOT NULL**. Це важливо, оскільки первинний ключ повинен бути унікальним і не допускати невизначених значень.

Щоб створити первинний ключ, можна також в кінці визначення таблиці дописати команду

виду **CONSTRAINT <ім'я_ключа> <тип_ключа> (<поля_що_входять_в_ключ>)**:

```
CONSTRAINT pkID PRIMARY KEY (ID)
```

Тут **pkID** – ім'я первинного ключа, а **ID** – стовпчик, на основі якого він створюється.

Якщо слід додати чи видалити первинний ключ у вже існуючу таблицю, то можна скористатися ще одним розширенням команди **ALTER TABLE**. Приклад додавання первинного ключа в нашу таблицю:

```
ALTER TABLE OFFICES ADD CONSTRAINT PK_ID PRIMARY KEY (ID);
```

При цьому в таблицю **OFFICES** додасться такий же первинний ключ, як і в попередньому прикладі, коли він створювався разом з таблицею. Щоб видалити первинний ключ, необхідно ввести наступну команду:

```
ALTER TABLE OFFICES DROP CONSTRAINT pkID;
```

При цьому ключ із ім'ям **pkID** буде вилучений з бази даних.

2.2.2. Генератори

Оскільки первинний ключ призначений для забезпечення унікальності значень у таблиці, то ніякі два записи в цій таблиці не можуть мати однакових значень цього ключа. Для того щоб задовольнити цю умову, при занесенні нового запису в таблицю **FireBird** повинен переглянути всі наявні записи й з'ясувати, чи немає у ній вже таких значень. Для швидкого пошуку в **FireBird** існує механізм індексів – спеціальних об'єктів **FireBird**, які дозволяють швидко знайти запис у таблиці. Тому при створенні й видаленні первинного ключа створюється або видаляється індекс на те поле (або поля), що входить у первинний ключ.

Первинний ключ може містити кілька полів. При цьому буде відслідковуватися унікальність сполучення значень цих полів. Проте на практиці найпоширенішим видом ключа є лічильник – ціле поле, що містить значення, яке автоматично збільшуються на 1 при вставці кожного наступного запису.

Деякі **СУБД**, такі як **MS SQL**, **MySQL** мають спеціальний тип – лічильник (**auto increment**). При додаванні в таблицю нового запису значення поля із цим типом автоматично збільшується на певну величину (як правило, на одиницю). В **FireBird** немає поля типу лічильник, однак є можливість реалізувати подібну поведінку. Для створення поля, яке б заповнювалося автоматично при додаванні запису в таблицю, використовується сукупність двох засобів: генератор та тригер.

Генератор – це іменований лічильник. Всередині бази даних можна створити такий лічильник, дати йому унікальне ім'я в межах цієї бази й управляти його значеннями. Генератор можна створити за допомогою команди **CREATE GENERATOR**:

```
CREATE GENERATOR GID;
SET GENERATOR GID TO 15;
```

У даному прикладі в першому рядку в базі даних створюється генератор з іменем **GID**, а в другому – цьому генератору за допомогою команди **SET GENERATOR** присвоюється значення **15**. Щоб одержувати й змінювати значення генераторів, існує вмонтована в **FireBird** функція **GEN_ID**. Ця функція приймає як параметри ім'я генератора й величину збільшення, яку потрібно застосувати до даного генератора, а повертає ціле значення, що відповідає значенню генератора, отриманому в

результаті додавання до нього заданого числа. Ось приклад виклику функції **GEN_ID** у тригері або збережуваній процедурі:

```
Current_value = GEN_ID (GID, 1)
```

Тут **Current_value** – змінна, **GID** – генератор, **1** – приріст значення для генератора. У цьому прикладі в змінну **Current_value** потрапить значення генератора **GID** після додавання до нього **1,** тобто наступне значення генератора. Приріст для генератора може приймати будь-яке ціле значення, в тому числі нульове або від'ємне. Нульове значення приросту можна використовувати в тому випадку, коли потрібно лише прочитати стан генератора, без його зміни.

Щоб отримати значення генератора в прикладній програмі, можна скористатися таким запитом:

```
SELECT GEN_ID(GID, 1) FROM RDB$DATABASE
```

В таблиці **RDB$DATABASE** завжди міститься тільки один запис, тому в результаті даного запиту буде отримане значення генератора **GID.**

Якщо одночасно кілька користувачів спробують змінити і прочитати значення генератора, то всі вони отримають різні значення. Це гарантується самою " конструкцією" генераторів: вони працюють на найнижчому рівні сервера й ніякі процеси запису й вставки не впливають на них – інколи говорять, що генератори працюють "поза контекстом транзакцій".

Для того щоб помістити отримане від генератора значення в поле первинного ключа, існують два способи – вставка первинного ключа на стороні клієнта і на стороні сервера.

У випадку формування первинного ключа на стороні клієнта відбувається наступне. Коли сформовано запис, що буде вставлений у базу даних, виконується виклик функції **GEN_ID(<ім'я генератора>,1)** і отримане значення підставляється в сформований запис. Відбувається вставка в таблицю, при цьому ми отримаємо гарантовано унікальний первинний ключ.

Другий спосіб – формування первинного ключа на стороні сервера – взагалі виключає всяку турботу зі сторони клієнта про те, яке буде значення первинного ключа. В цьому випадку при вставці запису спрацьовує тригер – спеціальний об 'єкт бази даних, що може здійснювати якісь дії при вставці/видаленні/зміні записів у таблицях. І в цьому тригері відбувається виклик функції **GEN_ID**, отримання потрібного значення генератора і вставка його в таблицю. Проте при цьому прикладна програма-клієнт не бачить новоствореного запису, і не може оперувати значенням отриманого первинного ключа, доки не прочитає його.

2.3. Індекси

Концепція, покладена в основу індексів, є однією з найважливіших основ проектування баз даних. На основі індексів базуються багато головних об'єктів бази даних, до того ж правильне використання індексів є ключем до покращення продуктивності баз даних.

Індекс – це сукупність впорядкованих вказівників на записи в таблиці. Тобто, він містить значення одного або кількох полів у таблиці й адреси сторінок даних, на яких розташовуються ці значення. Таким чином, за значенням поля (або полів), що входить в індекс, за допомогою індексу можна швидко знайти те місце в таблиці , де розташовується запис, що містить це значення. Індекси сприяють прискоренню пошуку запису за індексованим полем. Основна функція індексів – забезпечувати швидкий пошук запису в таблиці.

Індекс не є частиною таблиці – це окремий об'єкт, пов'язаний з таблицею й іншими об'єктами бази даних. Це дозволяє відокремити зберігання інформації від її подання.

Індекси використовуються в трьох основних випадках:

- Для прискорення виконання запитів. Індекси створюються для полів, які використовуються в умовах пошуку **SQL**-запитів.
- Для забезпечення унікальності значень у полях. Так, обмеження первинного ключа вимагає, щоб у всій таблиці не було двох однакових значень полів, що входять у первинний ключ. Щоб виконати цю умову, необхідно при кожній вставці нового запису робити пошук такого ж

значення, яке буде вставлено. Для пошуку запису використовується особливий різновид індексу – унікальний індекс.

- Для забезпечення посилальної цілісності. Обмеження зовнішніх ключів **Foreign key** використовуються для перевірки того, щоб значення, що вставляють у таблицю, обов'язково існували в іншій таблиці. При створенні зовнішнього ключа автоматично створюється індекс, що застосовується як для прискорення запитів, які використовують з'єднання таблиць, так і для перевірки умов зовнішнього ключа.

Правильне застосування індексів може значно прискорити виконання запитів, але існують і певні застереження щодо їх використання. Є два моменти, що перешкоджають загальній індексації, – це дисковий простір і витрати при модифікації даних у таблиці. Кожен створюваний індекс має обсяг, що дорівнює обсягу даних в індексованому полі, плюс обсяг даних про розташування записів. Якщо створити індекси на кожне поле в таблиці, то їхній сумарний обсяг буде більшим, ніж обсяг даних у таблиці. Тому створення великої кількості індексів приводить до великої витрати дискового простору. Другий момент ще більше важливий – це часові витрати при модифікації даних у таблиці. У реляційній **СУБД** додавання/видалення записів відбуваються без значних витрат ресурсів сервера. Навіть якщо видаляється запис із середини бази даних, то не відбувається переміщення обсягів даних для того, щоб закрити "діру", – сервер просто позначить місце, що звільнилося, і при нагоді запише туди щось. Додавання нового запису майже завжди відбувається в кінець таблиці. Проте, хоча основні дані в таблиці не переписуються сервером при модифікації, але дані, що зберігаються в індексах, повинні модифікуватися при кожному додаванні/видаленні записів. Звичайно, реалізація індексу певним чином розрахована на часті перебудови, але ці дії все-таки забирають час і ресурси процесора й при надто великій кількості індексів у таблиці модифікація даних у ній може бути в десятки разів повільнішою, ніж у такої ж таблиці без індексів.

Крім цих двох головних причин, є ще кілька зауважень щодо обмеження застосування індексу . Перше – так зване правило 20 %. Воно говорить, що коли запит на вибірку повертає більше 20 % записів з таблиці, то використання індексу може сповільнити вибірку даних. Звичайно, ситуація залежить від конкретного запиту й умов, накладених на вибірку , але потрібно пам'ятати, що 20 % записів є межею, за якою ефективність використання індексів ставиться під питання.

Друге зауваження пов'язане з роботою оптимізатора **FireBird**. Оптимізатор – це сукупність механізмів, які розробляють план виконання запиту. Коли користувач передає **FireBird** якийсь **SQL**-запит, він вказує, що повинен повернути сервер у результаті виконання запиту, але не визначає, як сервер повинен виконувати запит. Оптимізатор на основі переданого запиту формує план його виконання, тобто звідки й у якому порядку будуть братися дані для виконання запиту, які індекси будуть при цьому використовуватися і т.д. Коли сервер аналізує умови на вибірку, то для кожного поля, що входить в умову, сервер намагається використати індекс. На жаль, алгоритм створення плану недосконалий й оптимізатор часто використовує індекси, які не дуже ефективні для конкретного запиту, через що може істотно сповільнитися час виконання. Тому створення зайвих індексів може привести до створення неоптимальних планів.

Треба відзначити, що у клоні **Yaffil** ця проблема ефективно вирішена за рахунок використання сучасних алгоритмів побудови планів.

Третім випадком, коли індекс не потрібний, є поля з обмеженим набором значень – наприклад, поле, що зберігає інформацію про стать людини й містить тільки два можливих значення – "ч" та "ж". Індексувати це поле не має сенсу.

Тепер розглянемо, коли слід використовувати індекси, щоб домогтися покращення продуктивності. Існує три основних випадки, коли необхідно індексувати поле:

- Коли це поле використовується в умовах пошуку в запитах.
- Коли це поле використовується при з'єднанні таблиць.
- Коли це поле використовується при сортуванні.

Синтаксис для створення індексів наступний:

```
CREATE [UNIQUE] [ASC[ENDING] | DESC[ENDING]] INDEX index_name ON table (col [, col ...]);
```

Мінімальним виразом для створення індексу, є наступний:

```
CREATE INDEX my_index ON My_Table(ID)
```

У цьому прикладі створюється індекс з назвою **my_index** для таблиці **My_Table**, причому індексованим полем є поле **ID**. Індекс є зростаючий, тобто значення в ньому впорядковані по зростанню, а також неунікальним, тобто поле **ID** може мати кілька однакових значень.

Як видно з опису синтаксису, індекс може містити не одне, а кілька полів. Такий індекс використовується часто у запитах, які містять в умовах пошуку або сортування сполучення індексованих полів. Наприклад, якщо в нас є таблиця, що містить поля "**Прізвище**", "**Ім'я**", "**По батькові**", то при запиті, що використовує сортування по цим полям, буде застосовано такий індекс.

У документації для оптимізації виконання запиту, що містить у блоці **WHERE** поєднання полів з умовою **OR** ("або") рекомендується використовувати не складений індекс, а кілька одинарних індексів на всі поля, що входять в умову **OR**.

Для зміни індексу використовується команда **ALTER INDEX**:

```
ALTER INDEX name {ACTIVE | INACTIVE};
```

Тут **name** – ім'я індексу, а **ACTIVE** та **INACTIVE** – два стани індексу, в які його можна перевести за допомогою команди **ALTER INDEX**. Параметр **ACTIVE** означає, що індекс активний і може застосовуватися у всіх запитах і процедурах. Встановлення індексу в **INACTIVE** (неактивний) приводить до відключення його використання. Для перебудови дерева індекса треба послідовно виконати дві команди:

```
ALTER INDEX name INACTIVE;
ALTER INDEX name ACTIVE;
```

Для видалення індексу з бази даних використовується команда **DROP INDEX**:

```
DROP INDEX назва_індексу;
```

2.4. Обмеження бази даних

Обмеження бази даних, – це правила, які визначають взаємозв'язки між таблицями і можуть перевіряти й змінювати дані в базі. Реалізовані ці правила у вигляді особливих об'єктів бази. Головна перевага використання обмежень полягає в можливості здійснити перевірку даних на сервері, внаслідок чого частина бізнеси-логіки прикладної програми на рівні обробки даних переноситься на сервер, що дозволяє централізувати і спростити її.

Існують наступні види обмежень у базі даних **InterBase**:

- первинний ключ – **PRIMARY KEY**;
- унікальний ключ – **UNIQUE KEY**;
- зовнішній ключ – **FOREIGN KEY**;
- перевірки – **CHECK**.

Обмеження бази даних можуть бути на основі одного поля і на основі кількох полів таблиці.

Так, первинний ключ із використанням синтаксису обмеження на основі одного поля описується так:

```
CREATE TABLE testl(
ID_PK INTEGER CONSTRAINT pktest NOT NULL PRIMARY KEY);
```

Для тієї ж самої мети можна скористатися синтаксисом обмежень на основі кількох полів:

```
CREATE TABLE test2(
ID_PK INTEGER NOT NULL,
CONSTRAINT pktst PRIMARY KEY (ID_PK));
```

Первинні ключі є одним з основних видів обмежень у базі даних. Вони застосовуються для однозначної ідентифікації записів у таблиці. Унікальні ключі мають аналогічне призначення – вони також служать для однозначної ідентифікації записів у таблиці . Відмінність первинних ключів від унікальних полягає в тому, що первинний ключ може бути в таблиці тільки один, а унікальних ключів – кілька.

Приклади первинних і унікальних ключів:

```
CREATE TABLE pkuk(
pk NUMERIC(15,0) NOT NULL PRIMARY KEY, /* первинний ключ */
        VARC        NOT NULL                    унікальний
kl  HAR(SO)     UNIQUE,                  * ключ */
        INTE                                    ще один            кл
k2  GER NOT             NULL UNIQUE       * унікальний         юч */);
```

Ще одним обмеженням, що часто використається в базах даних **InterBase**, є обмеження зовнішнього ключа. Це потужний засіб для підтримки посилальної цілісності в базі даних, що дозволяє не тільки контролювати наявність правильних посилань у БД, але й автоматично управляти цими посиланнями.

Ідея використання зовнішнього ключа наступна: якщо дві таблиці служать для зберігання взаємозалежної інформації, то необхідно гарантувати, щоб цей взаємозв'язок був завжди правильним . Приклад – документ "накладна", що містить загальний заголовок (дата, номер накладної і т.д.) і множина елементарних записів про товар (назва товару, кількість тощо). Для зберігання такого документа в базі даних створюється дві таблиці – одна для зберігання заголовків накладних, а друга – для зберігання вмісту накладної – записів про товари і їхню кількість. Такі таблиці називаються головною і підлеглою.

Для реалізації подібної поведінки головна таблиця зв’язується з підлеглою таблицею за допомогою зовнішнього ключа.

Нехай є наступна таблиця для зберігання заголовків накладних:

```
CREATE TABLE TITLE(
ID_TITLE INTEGER NOT NULL Primary Key,
DateNakl DATE,
NumNakl INTEGER,
NoteNakl VARCHAR(200));
```

Таблиця для зберігання інформації про товари, що входять у накладну:

```
CREATE TABLE INVENTORY(
ID_INVENTORY INTEGER NOT NULL PRIMARY KEY,
FK_TITLE INTEGER NOT NULL,
ProductName VARCHAR (255),
Kilkist DOUBLE PRECISION,
Positio INTEGER);
```

Якщо потрібно отримати інформацію про товари певної накладної, то слід використати такий запит:

```
SELECT * FROM INVENTORY WHERE     FK_TITLE=id_title
```

У цьому запиті **id_title** – ідентифікатор відповідної накладної.

По суті , в описуваній ситуації ніщо не заважає заповнити таблицю **INVENTORY** записами, що посилаються на неіснуючі записи в таблиці **TITLE**. Також нічого не перешкоджає видаленню заголовка вже існуючої накладної, у результаті чого записи про товари можуть стати "висячими".

Таким чином, контроль за цілісністю даних у базі повністю покладено на прикладну програму, що може привести до різної інтерпретації даних і до помилок.

Тому необхідно накласти обмеження про те, що в таблиці **INVENTORY** можуть міститися лише такі записи про товари, які мають правильні посилання на існуючий заголовок накладної. Саме це

і є обмеження зовнішнього ключа, яке дозволяє вставляти в поля, що входять в обмеження, тільки ті значення, які є в іншій таблиці.

Для даного прикладу необхідно накласти обмеження зовнішнього ключа на поле **FK_TITLE** і зв'язати його з первинним ключем **ID_TITLE** в **TITLE**. Додати зовнішній ключ у вже існуючу таблицю можна наступною командою:

```
ALTER TABLE INVENTORY
ADD CONSTRAINT fktitle FOREIGN KEY(FK_TITLE) REFERENCES TITLE (ID_TITLE)
```

Тут **INVENTORY** – ім'я таблиці, на яку накладається обмеження зовнішнього ключа; **fktitle** – ім'я зовнішнього ключа; **FK_TITLE** – поле, що входить до зовнішнього ключа; **TITLE** – ім'я таблиці , що надає значення (посилальну основу) для зовнішнього ключа; **ID_TITLE** – поля первинного або унікального ключа в таблиці **TITLE**, які є посилальною основою для зовнішнього ключа.

Найчастіше зовнішній ключ створюється при створенні таблиці, наприклад:

```
CREATE TABLE Inventory2(
...
FK_TABLE INTEGER NOT NULL CONSTRAINT fkinv FOREIGN KEY (FKJTABLE) REFERENCES
TITLE (ID_TITLE)
...) ;
```

У полях, на основі яких створюється зовнішній ключ, допускається застосування значення **NULL**. Ця можливість додана для дозволу взаємних посилань. Коли є дві таблиці, що посилаються одна на одну за допомогою зовнішніх ключів, то якщо не дозволити посилання на **NULL** у цих зовнішніх ключах, то в зв'язані таблиці неможливо буде додати ні одного запису: щоб додати запис до першої таблиці, треба буде мати запис у другій таблиці, і навпаки. Використання **NULL** дозволяє організувати взаємні зв’язки двох таблиць з перехресним посиланням, а також зберігати ієрархічні структури в реляційних таблицях – при цьому кореневі вузли посилаються на "порожні" записи.

Одним з найбільш часто використовуваних обмежень у базі даних є обмеження перевірки. Призначення його дуже просте – перевіряти значення, що вставляються в таблицю, на певну умову і, залежно від її виконання, вставляти або не вставляти дані. Ось найпростіший приклад перевірки:

```
CREATE TABLE CHECKTST(
 ID INTEGER CHECK(ID>0));
```

Дане обмеження перевірки встановлює, чи нове значення поля **ID** більше від нуля, і залежно від результату дозволяє вставити або змінити його. При використанні **CHEK** слід мати на увазі наступне:

- Дані в **CHECK** беруться тільки з поточного запису. Не слід брати дані для виразу перевірки з інших записів цієї ж таблиці – вони можуть бути змінені іншими користувачами.
- Поле може мати тільки одне обмеження **CHECK**.
- Якщо для опису поля використовувався домен, що має доменне обмеження **CHECK**, то його не можна перевизначити на рівні конкретного поля в таблиці.

Слід також зазначити, що обмеження **CHECK** реалізовано за допомогою системних тригерів, тому варто бути обережним у використанні великих умов – вони можуть сильно сповільнити процеси вставки й відновлення записів.

Щоб видалити обмеження, необхідно скористатися конструкцією **ALTER TABLE** наступного

виду:

```
ALTER TABLE tablename DROP CONSTRAINT constraintname
```

де **constraintname –** назва обмеження, яке треба видалити. Якщо при створенні обмеження було задано назву, то слід використати її, а якщо ні, то потрібно відкрити якийсь засіб адміністрування **InterBase**, знайти потрібне обмеження й з'ясувати, яке системне ім'я для нього згенерував сервер. Видаляти обмеження може тільки власник таблиці або системний адміністратор **SYSDBA**.

3. Завдання на лабораторну роботу

Створити базу даних, яка містить чотири таблиці з логічно пов'язаними між собою даними. Кожна таблиця повинна мати щонайменше п'ять полів з даними різного типу (в тому числі текстові дані, що міститимуть кирилицю), одне (або кілька) з яких повинно бути первинним ключем. Для двох таблиць створити по одному індексу. Два поля повинні мати значення за замовчуванням, і ще два поля повинні бути обчислюваними. База даних повинна містити генератор, який слід встановити у значення згідно завдання за варіантом (табл. №1.1).

Вид даних, які мають зберігатися у таблицях, слід вибрати самостійно згідно завдання за варіантом (табл. №1.1).

Таблиця №1.1.

Варіант	Вид бази даних	Значення генератора
1	Дані про комплектуючі до комп'ютерів	10
2	Дані про бібліотечну літературу	70
3	Дані про абонентів бібліотеки	11
4	Дані про абонентів телефонної станції	71
5	Дані про географічні об'єкти області	12
6	Дані про студентів, що навчаються в університеті	73
7	Дані про працівників підприємства	13
8	Дані про структуру підприємства	75
9	Дані про автомобілі	15
10	Дані про фармацевтичну продукцію	77
11	Дані про меблеву продукцію	17
12	Дані про будівельні матеріали	79
13	Дані по складському обліку на підприємстві	19
14	Дані по прихідних та розхідних платіжних документах	81
15	Дані про відвідувачів WEB-сайту	21
16	Дані про кабельно-провідникову продукцію	83
17	Дані про радіоелектронну апаратуру	23
18	Дані про побутову техніку	85
19	Дані про продуктові товари	25
20	Дані про астрономічні об'єкти	87
21	Дані про напівпровідникові пристрої	27
22	Дані про рослинний світ країни	89
23	Дані про тваринний світ країни	29
24	Дані про підприємства області	91
25	Дані про історичні пам'ятки області	30
26	Дані про успішність студентів групи	93
27	Дані про CD та DVD- диски	31
28	Дані про продукцію легкої промисловості	95
29	Дані про міста світу	33
30	Дані про морських тварин та рослин	97

Лабораторна робота №7

Тема: Збережені процедури. Види процедур. Тригери.

1. Збережені процедури

Клієнти, з'єднані з **SQL**-сервером, можуть звертатися до нього для вирішення відносно складних завдань. Часто це більш ефективно, ніж вирішувати такі завдання на комп'ютері клієнта. Зокрема, це дозволяє мінімізувати обсяг інформації, що пересилається по мережі. Наприклад, якщо користувачеві потрібно знати лише суму значень у якомусь стовпці таблиці, то доцільно всю процедуру пошуку та додавання виконати на сервері, а користувачу переслати лише кінцеве значення. Ідеальним є випадок, коли клієнт отримує лише необхідний обсяг даних.

Багато серверів, у тому числі й **InterBase**, дозволяють створення спеціальних процедур, які зберігаються та виконуються на сервері. Такі процедури називаються збереженими процедурами (ЗП) [4].

InterBase підтримує два види збережених процедур: виконавчі процедури та процедури вибірки. Результати, що повертають збережені процедури, представляють собою віртуальні таблиці з даними, які можна читати. Це дозволяє використовувати їх у звичайних **SQL**-запитах.

Збережені процедури дозволяють реалізувати значну частину логіки програми на рівні бази даних і, таким чином, підвищити продуктивність програми, централізувати обробку даних і зменшити кількість коду, необхідного для виконання поставлених завдань.

Збережена процедура – це частина метаданих бази, що представляє собою відкомпільовану у внутрішнє подання **InterBase** підпрограму, написану спеціальною мовою, компілятор якої вмонтований у ядро сервера.

Збережену процедуру можна викликати із клієнтської програми, з тригера або іншої процедури. Процедура виконується всередині серверного процесу й може маніпулювати даними в базі, а також повертати клієнтові, що її викликав (тобто тригеру, іншій процедурі, прикладній програмі тощо), результати свого виконання.

1.1. Створення збережених процедур

Основою потужних можливостей, закладених у ЗП, є процедурна мова програмування, що має у своєму складі як модифіковані оператори звичайного **SQL** (такі як **INSERT**, **UPDATE** й **SELECT**), так і засоби організації розгалужень і циклів (**IF… THEN**, **WHILE… DO**), а також засоби обробки виняткових ситуацій. Мова збережених процедур дозволяє реалізувати складні алгоритми роботи з даними, а завдяки орієнтованості на роботу з реляційними даними ЗП виходять значно компактнішими від аналогічних процедур на традиційних мовах програмування.

Синтаксис збереженої процедури описується так:

```
CREATE PROCEDURE name [(param <datatype> [, param
<datatype> …])] [RETURNS <datatype> [, param <datatype> …])]
AS
<procedure_body> [terminator]

<procedure_body> =
[<variable_declaration_list>]
<block>

<variable_declaration_list> =
DECLARE VARIABLE var <datatype>;
[DECLARE VARIABLE var <datatype>; …]

<block> =
BEGIN
 <compound_statement>
 [<compound_statement> …]
END
```

Розглянемо приклад простої збереженої процедури, яка отримує на вході два числа, додає їх і повертає отриманий результат:

```
CREATE PROCEDURE SP_ADD (FIRST_ARG DOUBLE PRECISION, SECOND_ARG DOUBLE PRECISION)
RETURNS (RES DOUBLE PRECISION)
AS
BEGIN
 RES=FIRST_ARG+SECOND_ARG;
 SUSPEND;
END
```

Після команди **CREATE PROCEDURE** вказується назва процедури (яка повинна бути унікальною у межах бази даних) – в даному випадку **SP_ADD**, потім у дужках перераховуються вхідні параметри із вказуванням їхніх типів – у приведеному прикладі **FIRST_ARG** та **SECOND_ARG**. Список вхідних параметрів є необов'язковою частиною оператора **CREATE PROCEDURE** – бувають випадки, коли всі необхідні дані для своєї роботи процедура отримує за допомогою запитів до таблиць всередині тіла процедури.

У збережених процедурах використовуються будь-які скалярні типи даних. Не передбачено застосування масивів і доменів. Після ключового слова **RETURNS** в дужках перелічуються параметри, що повертаються процедурою, із вказуванням їхніх типів.

Якщо процедура не повинна повертати параметри, то слово **RETURNS** і список вихідних параметрів відсутні.

Після **RETURNS** вказується ключове слово **AS**. До нього розташовується заголовок, після нього – тіло процедури.

Тіло збереженої процедури може містити список внутрішніх (локальних) змінних (якщо вони потрібні), і блок операторів, що розміщується між **BEGIN** та **END**. В даному прикладі тіло ЗП дуже просте – вона додає два вхідних аргументи й присвоює результат вихідному, після чого викликається команда **SUSPEND**, яка потрібна для передачі вихідних параметрів туди, звідки була викликана процедура.

Оператори всередині процедури закінчуються крапкою з комою (;). Проте цей символ є водночас і стандартним роздільником команд в **SQL** – він вказує інтерпретаторові **SQL**, що текст команди введено і треба починати його обробляти. Для усунення конфлікту між стандартним розділювачем команд **SQL** та розділювачем операторів процедури, необхідно перед командою створення ЗП міняти роздільник команд **SQL** на інший символ, відмінний від крапки з комою, а після тексту ЗП відновлювати його назад. Команда **ISQL** [5], що змінює роздільник **SQL**, виглядає так:

```
SET TERM <NEW_TERM> <OLD_TERM>
```

де **NEW_TERM** – новий розділювач, **OLD_TERM** – діючий розділювач.

Для типового випадку створення збереженої процедури це виглядає так:

```
SET TERM ^ ;
CREATE PROCEDURE SOME_PROCEDURE
 .....
END ^
SET TERM ; ^
```

1.2. Виклик збереженої процедури

Щоб отримати результати роботи процедури, можна використати такий **SQL**-запит:

```
SELECT * FROM SP_ADD (133, 644)
```

Цей запит поверне нам один рядок, що містить одне поле **RESULT**, у якому буде перебувати сума чисел **133** і **644**, тобто **777.**

Таким чином, процедуру можна використовувати в звичайних **SQL**-запитах, що виконуються як у клієнтських програмах, так і в інших ЗП або тригерах. Таке використання процедури стало можливим через застосування команди **SUSPEND** в кінці процедури.

Як вже було зазначено, в **InterBase** (і у всіх його клонах) існує два типи збережених процедур: процедури вибірки (**selectable procedures**) і виконавчі процедури (**executable procedures**). Відмінність у роботі цих двох видів ЗП полягає в тому, що процедура вибірки повертає множину вихідних параметрів, згрупованих порядково, які мають вигляд набору даних, а виконавчі процедури або взагалі не повертають параметрів, або повертають тільки один набір параметрів, перерахованих в **RETURNS**, тобто один рядок параметрів. Процедури вибірки викликаються в запитах **SELECT**, а виконавчі процедури, – за допомогою команди **EXECUTE PROCEDURE.**

Обидва види збережених процедур мають однаковий синтаксис створення й формально нічим не відрізняються, тому будь-яка процедура може бути викликана як у **SELECT**-запиті , так і за допомогою оператора **EXECUTE PROCEDURE**. Різниця полягає в проектуванні процедури для певного типу виклику. Тобто процедура вибірки створюється для виклику із запиту **SELECT**, а виконавча процедура, – для виклику з використанням **EXECUTE PROCEDURE.**

Розглянемо приклад процедури-вибірки. Для цього створимо збережену процедуру, що

працює так само, як запит виду **SELECT ID, NAME FROM TABLE_EXAMPLE:**

```
CREATE PROCEDURE SIMPLE_SELECT_SP
RETURNS ( PROCID INTEGER, PROCNAME VARCHAR(80) )
AS
BEGIN
FOR SELECT ID, NAME FROM TABLE_EXAMPLE INTO :PROCID, :PROCNAME DO
 BEGIN SUSPEND;
  END
END
```

Ця процедура не має вхідних параметрів і має два вихідних параметри – **PROCID** та **PROCNAME.**

Для отримання даних з таблиці використано конструкцію **FOR SELECT:**

```
FOR SELECT ID, NAME FROM TABLE_EXAMPLE INTO :PROCID,
:PROCNAME DO BEGIN
 /* ЩОСЬ РОБИМО ЗІ ЗМІННИМИ PROCID Й PROCNAME
*/ END
```

Для кожного рядка, вибраного з таблиці **TABLE_EXAMPLE**, значення полів **ID** та **NAME** поміщуються у параметри **PROCID** й **PROCNAME**, а потім передаються запиту, що викликав процедуру. У даному прикладі при звертанні до параметрів використовується символ двокрапки (":"). Це робиться для того, щоб відрізняти поля таблиць від змінних (параметрів). Адже змінні можуть мати таку ж назву, як і поля в таблицях.

Щоб оголосити локальну змінну в збереженій процедурі, необхідно помістити її опис після ключового слова **AS** і до першого слова **BEGIN**. Опис локальної змінної виглядає так:

```
DECLARE VARIABLE <VARIABLE_NAME> <VARIABLE_TYPE>;
```

Тут **VARIABLE_NAME** – назва змінної, а **VARIABLE_TYPE** – її тип. Наприклад, щоб оголосити локальну змінну **MYINT** цілого типу, потрібно вставити між **AS** та **BEGIN** наступний опис:

```
DECLARE VARIABLE MYINT INTEGER;
```

Але двокрапку перед ім'ям змінної необхідно використовувати тільки всередині **SQL**-запитів.

В інших випадках звертання до змінної робиться без двокрапки, наприклад:

```
PROCNAME='SOME NAME';
```

Повернемося до прикладу наведеної вище процедури. Оператор **FOR SELECT** повертає дані не

у вигляді таблиці – набору даних, а по одному рядку. Під час кожної ітерації циклу повертаються поля, що поміщаються у відповідну змінну: **ID => PROCID, NAME => PROCNAME**. У частині **DO** ці змінні за допомогою команди **SUSPEND** надсилаються до клієнта, що викликав процедуру. Таким чином, команда **FOR SELECT... DO** організує цикл по записах, що вибираються у частині **SELECT** цієї команди.

1.3. Цикли й оператори розгалуження

Крім команди **FOR SELECT... DO**, що організує цикл по записах якоїсь вибірки, існує інший вид циклу – **WHILE...DO**, що дозволяє організувати цикл на основі перевірки певних умов. Розглянемо приклад процедури, що використовує цикл **WHILE... DO**. Ця процедура повертає квадрати цілих чисел від **0** до **99**:

```
CREATE PROCEDURE QUAD
RETURNS (QUADRAT INTEGER)
AS
DECLARE VARIABLE I INTEGER;
BEGIN
 I = 1;
 WHILE (I<100) DO BEGIN
  QUADRAT=I*I;
  I=I+1;
  SUSPEND;
 END
END
```

У результаті виконання запиту **SELECT * FROM QUAD** ми отримаємо таблицю, що містить один стовпчик **QUADRAT**, у якому будуть квадрати цілих чисел від **1** до **99.**

Крім перебору результатів **SQL**-вибірки й класичного циклу, у мові збережених процедур використовується оператор **IF...THEN...ELSE**, що дозволяє організувати розгалуження залежно від виконання якихось умов. Його синтаксис схожий на більшість операторів розгалуження в мовах програмування високого рівня, таких як Паскаль чи Сі.

Розглянемо більш складний приклад збереженої процедури, що робить такі дії:

1. Обчислює середню ціну в таблиці **TABLE_EXAMPLE.**
2. Для кожного запису в таблиці робить перевірку: якщо існуюча ціна (**PRICE**) більша від середньої ціни, то встановлює ціну, рівну величині середньої ціни, плюс фіксований відсоток.
3. Якщо існуюча ціна менша або рівна середній ціні, то встановлює ціну, що дорівнює колишній ціні, плюс половину різниці між колишньою й середньою ціною.
4. Повертає всі змінені рядки в таблиці.

Для початку задамо ім'я процедури, а також вхідні й вихідні параметри. Все це прописується в заголовку збереженої процедури

```
CREATE PROCEDURE INCREASEPRICES (
PERCENT2INCREASE DOUBLE PRECISION)
RETURNS (ID INTEGER, NAME VARCHAR(50), NEW_PRICE DOUBLE PRECISION) AS
```

Процедура буде називатися **INCREASEPRICES**, у неї один вхідний параметр **PERCENT2INCREASE**, що має тип **DOUBLE PRECISION**, і три вихідних параметри – **ID, NAME** й **NEW_PRICE**. Зверніть увагу, що перші два вихідних параметри мають такі ж імена, як і поля в таблиці **TABLE_EXAMPLE**, з якою ми збираємося працювати. Це допускається правилами мови збережених процедур.

Тепер ми повинні оголосити локальну змінну, яка буде використовуватися для зберігання середнього значення:

```
DECLARE VARIABLE AVG_PRICE DOUBLE PRECISION;
```

Тепер перейдемо до тіла збереженої процедури. Спочатку нам необхідно виконати перший крок нашого алгоритму – обчислити середню ціну. Для цього скористаємося наступним запитом:

```
SELECT AVG(PRICE) FROM TABLE_EXAMPLE INTO :AVG_PRICE;
```

Цей запит використовує агрегатну функцію **AVG**, що повертає середнє значення поля **PRICE** серед відібраних рядків запиту – в нашому випадку середнє значення **PRICE** по всій таблиці **TABLE_EXAMPLE.** Значення, що повертається запитом, записується у змінну **AVG_PRICE.** Зверніть увагу, що змінна **AVG_PRICE** записується із двокрапкою – для того, щоб відрізнити її від полів запиту.

Особливістю даного запиту є те, що він завжди повертає лише один запис. Такі запити називаються **singleton**-запитами. Тільки такі вибірки можна використовувати в збережених процедурах. Якщо запит повертає більше одного рядка, то його необхідно оформити у вигляді конструкції **FOR SELECT...DO**, що організує цикл для обробки кожного отриманого рядка.

Таким чином, ми отримали середнє значення ціни. Тепер необхідно пройтися по всій таблиці, порівняти значення ціни в кожному записі із середньою ціною й виконати відповідні дії:

```
      FOR SELECT ID, NAME, PRICE FROM TABLE_EXAMPLE INTO :ID, :NAME,
:NEW_PRICE DO BEGIN
       /* ТУТ ОБРОБЛЯЄМО КОЖЕН ЗАПИС*/
      END
```

При виконанні цієї конструкції з таблиці **TABLE_EXAMPLE** порядково вибираються дані й значення полів у кожному рядку привласнюються змінним **ID**, **NAME** й **NEW_PRICE**. Ці змінні оголошено як вихідні параметри. (Передача параметрів клієнту здійснюється тільки при виконанні команди **SUSPEND**, а до цього вони використовуються в ролі звичайних змінних.)

Отже, всередині тіла циклу **BEGIN...END** ми можемо обробити значення кожного рядка.

Процедуру порівняння реалізуємо за допомогою оператора **IF**:

```
      IF (NEW_PRICE > AVG_PRICE) THEN /*ЯКЩО ІСНУЮЧА ЦІНА БІЛЬША ВІД СЕРЕДНЬОЇ
ЦІНИ*/ BEGIN
       /* ВСТАНОВЛЮЄМО НОВУ ЦІНУ, РІВНУ ВЕЛИЧИНІ СЕРЕДНЬОЇ ЦІНИ, ПЛЮС ФІКСОВАНИЙ
 ВІДСОТОК */ NEW_PRICE = (AVG_PRICE + AVG_PRICE*(PERCENT2INCREASE/100));
 UPDATE TABLE_EXAMPLE SET PRICE_1 = :NEW_PRICE WHERE ID = :ID;
END ELSE BEGIN
 /* ЯКЩО ІСНУЮЧА ЦІНА МЕНША АБО ДОРІВНЮЄ СЕРЕДНІЙ ЦІНІ, ТО ВСТАНОВЛЮЄМО ЦІНУ, РІВНУ
    КОЛИШНІЙ ЦІНІ, ПЛЮС ПОЛОВИНУ РІЗНИЦІ МІЖ КОЛИШНЬОЮ Й СЕРЕДНЬОЮ ЦІНОЮ */
       NEW_PRICE = (NEW_PNCE + ( (AVG_PNCE NEW_PRICE)/2) ) ;
 UPDATE TABLE_EXAMPLE SET PRICE_1 = :NEW_PRICE WHERE ID = :ID;
END
```

Для того щоб змінити ціну відповідно до обчисленої різниці, використовується оператор **UPDATE**, що дозволяє модифікувати існуючі записи. Щоб однозначно вказати, у якому записі потрібно змінювати ціну, в умові **WHERE** використовується поле первинного ключа (воно порівнюється із значенням змінної, у якій зберігається значення **ID** для поточного запису (**ID=:ID**)). Зверніть увагу, що змінна **ID** записана із двокрапкою.

Після виконання конструкції **IF...THEN...ELSE** у змінних **ID**, **NAME** і **NEW_PRICE** перебувають дані, які потрібно повернути клієнту , що викликав процедуру. Для цього після **IF** необхідно вставити команду **SUSPEND**, що перешле дані туди, звідки викликали процедуру. На час пересилання дію процедури буде припинено, а коли від ЗП буде потрібно новий запис, то вона буде знову продовжена, – і так буде тривати доти, поки **FOR SELECT...DO** не перебере всі записи свого запиту.

Треба відзначити, що крім команди **SUSPEND**, що тільки призупиняє дію збереженої процедури, існує команда **EXIT**, що припиняє збережену процедуру після передачі рядка. Однак командою **EXIT** користуються досить рідко, оскільки вона потрібна в основному для того, щоб перервати цикл при досягненні якоїсь умови.

При цьому у випадку, коли процедура викликалася оператором **SELECT** і завершена по **EXIT**, останній вибраний рядок не буде повернуто. Тобто, якщо потрібно перервати процедуру й все-таки отримати цей рядок, слід скористатися послідовністю команд:

```
      SUSPEND;
      EXIT;
```

Основне призначення оператора **EXIT** – отримання **singleton**-наборів даних, що повертають параметри шляхом виклику через **EXECUTE PROCEDURE**. У цьому випадку встановлюються значення вихідних параметрів, але з них не формується набір даних **SQL**, і виконання процедури завершується.

Таким чином, процедура має наступний вид:

```
      CREATE PROCEDURE INCREASEPRICES ( PERCENT2INCREASE DOUBLE
PRECISION) RETURNS (ID INTEGER, NAME VARCHAR(50), NEW_PRICE DOUBLE
PRECISION) AS DECLARE VARIABLE AVG_PRICE DOUBLE PRECISION;
```

```
      BEGIN
       SELECT AVG(PRICE) FROM TABLE_EXAMPLE INTO :AVG_PRICE;
FOR SELECT ID, NAME, PRICE FROM TABLE_EXAMPLE INTO :ID, :NAME, :NEW_PRICE DO
 BEGIN /*ТУТ ОБРОБЛЯЄМО КОЖЕН ЗАПИС*/
       IF (NEW_PNCE > AVG_PRICE) THEN /*ЯКЩО ІСНУЮЧА ЦІНА БІЛЬШЕ СЕРЕДНЬОЇ
 ЦІНИ*/ BEGIN
        /*ВСТАНОВЛЮЄМО НОВУ ЦІНУ, РІВНУ ВЕЛИЧИНІ СЕРЕДНЬОЇ ЦІНИ, ПЛЮС ФІКСОВАНИЙ ВІДСОТОК */


        NEW_PRICE = (AVG_PRICE + AVG_PRICE*(PERCENT2LNCREASE/100));
  UPDATE TABLE_EXAMPLE SET PRICE_1 = :NEW_PRICE WHERE ID =
 :ID; END ELSE BEGIN
  /* ЯКЩО ІСНУЮЧА ЦІНА МЕНША АБО ДОРІВНЮЄ СЕРЕДНІЙ ЦІНІ, ТО ВСТАНОВЛЮЄМО ЦІНУ,
     РІВНУ КОЛИШНІЙ ЦІНІ, ПЛЮС ПОЛОВИНА РІЗНИЦІ МІЖ КОЛИШНЬОЮ Й СЕРЕДНЬОЮ ЦІНОЮ */
        NEW_PRICE = (NEW_PRICE + ((AVG_PRICE - NEW_PRICE)/2));
  UPDATE TABLE_EXAMPLE SET PRICE_1 = :NEW_PRICE WHERE ID =
 :ID; END
       SUSPEND;
      END
     END
```

Даний приклад ілюструє застосування основних конструкцій мови збережених процедур і тригерів.

1.4. Рекурсивні збережені процедури

Збережені процедури **InterBase** можуть бути рекурсивними. Це означає, що з процедури можна викликати саму себе. Допускається до 1000 рівнів вкладеності збережених процедур, однак треба пам'ятати про те, що вільні ресурси на сервері можуть закінчитися раніше, ніж буде досягнута максимальна вкладеність ЗП.

Одне з розповсюджених застосувань збережених процедур – це обробка деревоподібних структур, що зберігаються в базі даних. Дерева часто використовуються в складських, кадрових й в інших розповсюджених програмах.

Розглянемо приклад збереженої процедури, що вибирає всі товари певного типу, починаючи з певного рівня вкладеності. Нехай є наступна постановка завдання: маємо довідник товарів з ієрархічною структурою такого виду:

- Побутова техніка
- Холодильники
  - Трикамерні
  - Двокамерні
  - Однокамерні
- Пральні машини
  - Вертикальні
  - Фронтальні
    - Класичні
    - Вузькі
- Комп'ютерна техніка

Ця структура довідника категорій товарів може мати вітки різної глибини. Завдання – забезпечити вибірку всіх кінцевих елементів з довідника із "розгортанням повного імені", починаючи з будь- якого вузла. Наприклад, якщо ми вибираємо вузол "Пральні машини", то треба отримати наступні категорії:

Пральні машини • Вертикальні
Пральні машини • Фронтальні • Класичні
Пральні машини • Фронтальні • Вузькі

Спочатку визначимо структуру таблиць для зберігання інформації довідника товарів.

Використаємо спрощену схему для організації дерева в одній таблиці:

```
CREATE TABLE GOODSTREE (ID_GOOD INTEGER NOT NULL, ID_PARENT_GOOD INTEGER,
 GOOD_NAME VARCHAR(80), CONSTRAINT PKGOOCI PRIMARY KEY (ID_GOOD) );
```

Так створюється одна таблиця **GOODSTREE**, у якій всього три поля: **ID_GOOD** – унікальний ідентифікатор категорії, **ID_PARENT_GOOD** – ідентифікатор батьківської категорії для даної категорії, і **GOOD_NAME** – назва категорії. Щоб забезпечити цілісність даних у таблиці, накладемо на неї обмеження зовнішнього ключа:

```
ALTER TABLE GOODSTREE ADD CONSTRAINT FK_GOODSTREE FOREIGN KEY (ID_PARENT_GOOD)
REFERENCES GOODSTPEE (ID_GOOD)
```

Таблиця посилається сама на себе і даний зовнішній ключ стежить за тим, щоб у таблиці не було посилань на неіснуючі батьківські записи, а також перешкоджає спробам видалити категорію товарів, у якій є нащадки.

У збережених процедурах, що обробляють деревоподібні структури, склалась своя термінологія. Кожен елемент дерева називаються вузлом; а відносини між вузлами, що посилаються один на одного, називаються відносинами "предок-нащадок". Вузли, що перебувають на самому кінці дерева й не мають нащадків, називаються "листками".

У збереженої процедури вхідним параметром буде ідентифікатор категорії, починаючи з якого слід прочитати дані. Процедура матиме такий вигляд:

```
CREATE PROCEDURE GETFULLNAME (ID_GOOD2SHOW INTEGER)
RETURNS (FULL_GOODS_NAME VARCHAR(1000), ID_CHILD_GOOD INTEGER)
AS
DECLARE VARIABLE CURR_CHILD_NAME VARCHAR(80);
BEGIN
 /* ОРГАНІЗОВУЄМО ЗОВНІШНІЙ ЦИКЛ ПО БЕЗПОСЕРЕДНІХ НАЩАДКАХ ТОВАРУ З
ID_GOOD=ID_GOOD2SHOW */ FOR SELECT GTL.ID_GOOD, GTL.GOOD_NAME FROM GOODSTREE GTL
 WHERE GTL.ID_PARENT_GOOD=:ID_GOOD2SHOW INTO:ID_CHILD_GOOD, :FULL_GOODS_NAME DO
BEGIN
 /* ПЕРЕВІРКА ЗА ДОПОМОГОЮ ФУНКЦІЇ EXISTS, ЩО ПОВЕРТАЄ TRUE, ЯКЩО ЗАПИТ У ДУЖКАХ
    ПОВЕРНЕ ХОЧА Б ОДИН РЯДОК. ЯКЩО В ЗНАЙДЕНОГО ВУЗЛА З ID_PARENT_GOOD = ID_CHILD_GOOD
          НЕМАЄ НАЩАДКІВ, ТО ВІН Є "ЛИСТКОМ" ДЕРЕВА Й ПОПАДАЄ В РЕЗУЛЬТАТИ */
       IF (NOT EXISTS(SELECT * FROM GOODSTREE WHERE
 GOODSTREE.ID_PARENT_GOOD=:ID_CHILD_GOOD)) THEN BEGIN
        /* ПЕРЕДАЄМО "ЛИСТОК" ДЕРЕВА В РЕЗУЛЬТАТИ */
        SUSPEND;
       END ELSE /* ДЛЯ ВУЗЛІВ, У ЯКИХ Є НАЩАДКИ */
       BEGIN
        /* ЗБЕРІГАЄМО ІМ'Я ВУЗЛА-БАТЬКА В ТИМЧАСОВІЙ
  ЗМІННІЙ */ CURR_CHILD_NAME=FULL_GOODS_NAME;
        /* РЕКУРСИВНО ЗАПУСКАЄМО ЦЮ ПРОЦЕДУРУ */
        FOR SELECT ID_CHILD_GOOD, FULL_GOODS_NAME FROM GETFULLNAME (:ID_CHILD_GOOD)
         INTO:ID_CHILD_GOOD, :FULL_GOODS_NAME DO
         BEGIN
   /* ДОДАЄМО ІМ'Я ВУЗЛА-БАТЬКА ДО ЗНАЙДЕНОГО ІМЕНІ НАЩАДКА ЗА ДОПОМОГОЮ
      ОПЕРАЦІЇ КОНКАТЕНАЦІЇ РЯДКІВ || */
          FULL_GOODS_NAME=CURR_CHILD_NAME || FULL_GOODS_NAME;
          SUSPEND; /* ПОВЕРТАЄМО ПОВНЕ ІМ'Я ТОВАРУ */
         END
       END
      END
     END
```

2. Тригери

Тригер в **InterBase** – це особливий вид збереженої процедури, що виконується автоматично при вставці, видаленні або модифікації запису таблиці або представлення (**view**). Тригери можуть "спрацьовувати" безпосередньо до або відразу ж після зазначеної події. Тригери дозволяють надати "активності" даним, що зберігаються в базі, централізувати їхню обробку й спростити логіку клієнтських програм.

SQL дає можливість вставляти, видаляти й модифікувати дані в таблицях бази даних за допомогою відповідних команд – **INSERT**, **DELETE** й **UPDATE**. Часто є корисною можливість перехоплення таких команд, що дозволятиме зробити з даними якусь дію. Наприклад, записати ці

дані в спеціальну таблицю, а заодно записати, хто й коли здійснив модифікацію даних. Або ж перевірити дані на правильність чи відповідність певним умовам, і залежно від результатів перевірки прийняти проведені зміни або відкинути їх.

Для того, щоб виконувати якісь дії, пов'язані зі зміною даних у базі даних, і використовуються тригери.

Фактично тригер є набором команд процедурної мови **InterBase**, що виконується при виконанні операцій **INSERT/ DELETE/ UPDATE**. На відміну від збережених процедур, тригер ніколи не має вхідних та вихідних параметрів, проте містить контекстні змінні **NEW** та **OLD**. Ці змінні дозволяють отримати доступ до полів таблиці, з якою зв'язаний тригер.

Тригер завжди прив'язаний до якоїсь певної таблиці й може "перехоплювати" дані тільки цієї таблиці. Існує три основні **SQL**-операції по відношенню до даних, – **INSERT/ DELETE/ UPDATE.**

Відповідно перший поділ тригерів – за цими операціями. Кожен конкретний тригер прив'язаний до однієї з них, тобто він спрацьовує, коли в "його" таблиці відбувається дана операція.

Спрацювання тригера може відбуватися "до" і "після" операції. Таким чином, ми отримуємо шість можливих видів тригерів на таблицю – до і після кожної із трьох можливих **SQL**-операцій.

2.1. Приклад тригера

Розглянемо простий приклад тригера, що спрацьовує до вставки в таблицю даних і заповнює поле первинного ключа. Нехай існує така таблиця:

```
CREATE TABLE TABLE_EXAMPLE (
ID INTEGER NOT NULL,
NAME VARCHAR(80),
PRICE DOUBLE PRECISION,
CONSTRAINT PKTABLE PRIMARY KEY (ID));
```

Тут поле **ID** є первинним ключем і значення цього поля повинні бути унікальними в межах таблиці. Щоб забезпечити виконання цієї вимоги, створимо генератор і тригер, що отримуватиме значення генератора й підставлятиме його в таблицю автоматично при вставці нового запису. Таким чином, у полі **ID** завжди будуть унікальні значення, тому що значення генератора автоматично збільшуватиметься щораз при звертанні до тригера. Отже, створюємо генератор:

```
CREATE GENERATOR GEN_TABLE_EXAMPLE_ID;
```

Встановлюємо його початкове значення в одиницю:

```
SET GENERATOR GEN_TABLE_EXAMPLE_ID TO 1;
```

Тепер створимо тригер. Треба сказати, що тригер, як і збережена процедура, може містити у своєму тілі кілька операторів, розділених крапкою з комою. Тому при роботі з **ISQL** необхідно скористатися командою зміни розділювача команд **SET TERM**, як це було описано вище, в пункті "Збережені процедури".

Розглянемо текст тригера:

```
CREATE TRIGGER TABLE_EXAMPLE_BI FOR TABLE_EXAMPLE
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
 IF (NEW.ID IS NULL) THEN NEW.ID=GEN_ID(GEN_TABLE_EXAMPLE_ID, 1);
END
```

Опис команди створення тригера починається із ключових слів **CREATE TRIGGER**, після яких треба вказати ім'я тригера – **TABLE_EXAMPLE_BI**. Потім слідує ключове слово **FOR**, після якого зазначено ім'я таблиці, для якої створюється тригер – **TABLE_EXAMPLE.**

На другому рядку команди приводиться опис стану тригера – ключове слово **ACTIVE** указує, що тригер є "активним". Тригер також може бути переведений у стан **INACTIVE**. Це означає, що він буде зберігатися в базі даних, але не спрацьовуватиме. Сполучення ключових слів **BEFORE INSERT** визначає, що тригер спрацьовує до вставки; а ключове слово **POSITION** і число **0** вказують почерговість (позицію) створюваного тригера серед тригерів того ж типу для даної таблиці. Позиція тригера потрібна тому, що в **InterBase** можна створити до 32000 тригерів кожного виду (наприклад, **BEFORE INSERT** або **AFTER UPDATE**), і серверу потрібно вказати, у якому порядку ці тригери будуть виконуватися. Тригери з меншою позицією виконуються першими. Якщо є кілька тригерів з одинаковою позицією, то вони будуть виконуватися за алфавітом.

Все вищерозглянуте до ключового слова **AS** утворює заголовок тригера. Після **AS** слідує його тіло. Тут і здійснюється вставка значення в поле первинного ключа. Спочатку за допомогою оператора **IF...THEN** перевіряється, чи не було заповнено це поле клієнтом. Вираз перевірки, що повертає булеве значення (**TRUE** (істина) або **FALSE** (неправда)), виглядає так: **IF (NEW.ID IS NULL)...** .

2.2. Контекстні змінні

В момент спрацьовування розглянутого вище тригера дані, прислані на вставку, ще не занесені в таблицю. Вони перебувають у проміжному буфері. І в тригера є можливість звертатися до цього буфера, щоб перевірити і/або змінити значення цих даних. Така можливість реалізована за допомогою контекстної змінної **NEW**. Можна розглядати цю змінну як структуру (щось подібне **struct** у **Ci** або **record** в **Pascal**), елементи якої є значеннями, присланими для здійснення операції (**INSERT** у нашому прикладі). Тобто всередині тригера можна звернутися до всіх полів ще не вставленого запису, використовуючи для цього синтаксис типу: **NEW.ID**, **NEW.NAME** тощо.

Для порівняння значення поля **ID** на присутність використовується наступна ділянка коду:

```
IF (NEW.ID IS NULL) THEN NEW.ID = GEN_ID(GEN_TABLE_EXAMPLE_ID, 1);
```

Спочатку в операторі **IF...THEN** перевіряється ідентифікатор **ID** на наявність якогось значення, адже воно може бути задане клієнтом. Якщо значенням **NEW.ID** є **NULL** (воно не задане), то викликається функція **GEN_ID**, що збільшує значення генератора **GEN_TABLE_EXAMPLE_ID** на одиницю й повертає отримане число, яке присвоюється полю **NEW.ID**. Таким чином задається унікальне значення первинного ключа **ID.**

Крім контекстної змінної **NEW**, існує її аналог – змінна **OLD**. На відміну від **NEW** , **OLD** містить старі значення запису, які видаляються або змінюються. Наприклад, ми можемо використати змінну **OLD** для одержання значень записів, які видаляються з таблиці:

```
CREATE TRIGGER TABLE_EXAMPLE_AD FOR TABLE_EXAMPLE
ACTIVE AFTER DELETE POSITION 0
AS
BEGIN
 IF (OLD.ID>1000) THEN BEGIN
  /*..ЩОСЬ РОБИМО..*/
 END
END
```

Тут створюється тригер, що спрацьовує після видалення даних (**AFTER DELETE**). Як видно з прикладу, можна отримати доступ до вже видалених даних. Звичайно, присвоєння виду **OLD.ID=10;** не має ніякого змісту – присвоєне значення втратиться на виході із тригера.

В різних видах тригерів контекстні змінні **NEW** та **OLD** використовуються по-різному, а в деяких випадках їх взагалі неможливо використати. Так, контекстна змінна **OLD** не може бути використана в тригерах **BEFORE/ AFTER INSERT** (якщо вставляється новий запис, то ніяких "старих" даних не існує). А змінна **NEW** не може бути використана в **BEFORE/ AFTER DELETE**. Обидві ці змінні одночасно можуть бути використані в тригерах **BEFORE/ AFTER UPDATE**, причому змінювати що-небудь можна, тільки застосовуючи змінну **NEW**, і тільки в тригерах **BEFORE INSERT/ UPDATE.**

2.3. Керування станом тригера

За замовчуванням тригер створюється активним, тобто він буде спрацьовувати при здійсненні відповідної операції. Станом тригера управляє ключове слово **ACTIVE** у заголовку. Якщо ж тригер зробити неактивним, то він не буде виконуватися при здійсненні операції. Це буває корисним при проведенні якихось позапланових дій над даними, наприклад, при масовому занесенню даних або їх ручному виправленні. Щоб відключити тригер, необхідно виконати команду:

```
ALTER TRIGGER <TRIGGER_NAME> INACTIVE;
```

Зверніть увагу, що ця команда відноситься до **Data Definition Language** (**DDL**), і її не можна викликати зі збережених процедур або інших тригерів. Хоча й існує спосіб управляти станом тригерів за допомогою модифікації системних таблиць, проте зміна системних таблиць є недокументованим способом роботи із тригерами.

3. Завдання на лабораторну роботу

Для бази даних, створеної протягом лабораторної роботи №1, виконати наступні завдання:

1. Спроектувати для кожної з таблиць тригер, що автоматично заповнюватиме унікальними значеннями поле первинного ключа.
2. Створити три збережені процедури, які розраховуватимуть якісь значення на основі даних у базі (наприклад, повну кількість товару, загальну суму і т. д.). Хоча б одна процедура повинна мати як вхідний, так і вихідний параметр (-и).

Лабораторна робота №8

Тема: Додаткові засоби InterBase для роботи з даними: представлення, виняткові ситуації, події, функції користувача (UDF).

1. Представлення

1.1. Використання представлень

Представлення (**VIEW**) – це віртуальна таблиця, створена на основі запиту до таблиць бази даних. Представлення реалізовується як запит, що зберігається на сервері й виконується завжди, коли відбувається звертання до представлення.

Розглянемо різні варіанти використання представлень. Вони дають можливість створити рівні організації даних, що дозволяє відокремити реалізацію зберігання даних від їхнього виду для користувачів. Наприклад, можна створити представлення, що вибирає дані з кількох таблиць. Якщо клієнти будуть використовувати це представлення, а не прямо звертатися до таблиць, то в розроблювача бази даних з'являється можливість міняти запит, що лежить в основі представлення, змінювати його (наприклад, з метою оптимізації), а клієнт цього не помітить – для нього представлення буде незмінним.

Крім того, що представлення ізолюють реалізацію зберігання даних від користувача, вони дозволяють організувати дані в більш зручному й простому виді . Проблема "спрощення" організації даних виникає, коли число таблиць у базі стає значним, а взаємозв'язки між ними – складними. Тоді представлення дозволяє виключити (або, навпаки, додати) частину даних, непотрібних (або навпаки – необхідних) конкретному клієнту бази.

Представлення також дозволяють спростити організацію безпеки доступу до даних в базі **InterBase.** Певні користувачі можуть мати права тільки на читання/зміну даних у представленні , але не мати ніяких прав (і навіть ніякого поняття) про таблиці, що лежать в основі представлення.

Для створення й видалення представлень існують команди мови **DDL** (**Data Definition Language** – підмножина **SQL**). Щоб створити представлення в **InterBase**, необхідно використати запит наступного синтаксису:

```
CREATE VIEW viewname [ (view_column [,
view_column...] ) ] AS <SELECT> [WITH CHECK OPTION];
```

Тут **viewname** – ім'я представлення, що повинно бути унікальним в межах бази даних, далі йде група не завжди обов'язкових назв полів, що входять у представлення: **[(view_column [, view_column...])]**. Обов'язково слід задати блок **<SELECT>**, який вибирає дані, що включаються у представлення.

Щоб змінити представлення, треба його перестворити, тобто видалити й створити наново. При видаленні представлення необхідно також знищити всі залежні від нього об'єкти – тригери, збережені процедури та інші представлення. У цьому полягає одна з головних незручностей роботи з представленнями – необхідність перестворювати дерево об'єктів, що використовують представлення (проте існують утиліти, які дозволяють зробити це простіше, ніж стандартні засоби **InterBase**). Щоб видалити представлення, необхідно скористатися наступною командою **DDL**:

```
DROP VIEW viewname;
```

Розглянемо приклад простого представлення:

```
CREATE VIEW MYVIEW AS SELECT NAME, PRICE FROM TABLE_EXAMPLE;
```

У цьому прикладі створюється представлення на основі запиту до таблиці **TABLE_EXAMPLE.** Представлення буде складатися із двох полів – **NAME** та **PRICE**, які будуть вибиратися з таблиці **TABLE_EXAMPLE** без всяких умов, тобто число записів у представленні **MYVIEW** дорівнюватиме числу записів в таблиці **TABLE_EXAMPLE.**

Однак представлення не завжди є такими простими. Вони можуть отримувати дані з кількох таблиць та інших представлень. Крім цього, вони можуть містити дані, що обчислюються на основі різних виразів – у тому числі на основі агрегатних функцій.

Щоб докладніше розглянути використання таких представлень, створимо дві зв'язані таблиці:

```
/* TABLE: WISEMEN */
CREATE TABLE WISEMEN (
ID_WISEMAN INTEGER NOT NULL,
WISEMAN_NAME VARCHAR(80));
/* PRIMARY KEYS DEFINITION */
ALTER TABLE WISEMEN ADD CONSTRAINT PK_WISEMEN PRIMARY KEY (ID_WISEMAN);
/* TABLE: WISEBOOK */
CREATE TABLE WISEBOOK (
ID_BOOK INTEGER NOT NULL,
ID_WISEMAN INTEGER,
BOOK VARCHAR (80) ) ;
/* PRIMARY KEYS DEFINITION */
ALTER TABLE WISEBOOK ADD CONSTRAINT PK_WISEBOOK PRIMARY KEY (ID_BOOK);
/* FOREIGN KEYS DEFINITION */
ALTER TABLE WISEBOOK
ADD CONSTRAINT FK_WISEBOOK FOREIGN KEY (ID_WISEMAN) REFERENCES WISEMEN
(ID_WISEMAN);
```

Таблиці **WISEMEN** та **WISEBOOK** зв'язані між собою відношенням **master-detail** за допомогою зовнішнього ключа – **FOREIGN KEY**. Припустимо, що ці таблиці будуть зберігати інформацію про вчених (**WISEMEN**) та їхні книги (**WISEBOOK**). Тоді, наприклад, представлення, що показує, скільки книг є в кожного вченого, може бути таким:

```
CREATE VIEW WISEBOOKCOUNT (WISEMAN, HOW_WISEBOOKS)
AS SELECT M.WISEMAN_NAME, COUNT(B.BOOK) FROM WISEMEN M, WISEBOOK B
WHERE (M.ID_WISEMAN = B.ID_WISEMAN) GROUP BY M.WISEMAN_NAME
```

Зверніть увагу, що при використанні у представленні обчислюваних виразів (наприклад, агрегатних функцій **COUNT**, **SUM**, **MAX**), необхідно використовувати явне іменування полів представлення, тобто давати імена всім полям, що повертаються запитом. Як видно з цього прикладу , вони не обов'язково повинні збігатися з іменами полів запиту, але їх кількість має збігатися з кількістю полів, що повертаються запитом. При цьому перше поле запиту відобразиться в перше поле представлення, друге – у друге і т.д.

Читання даних із представлення здійснюється так, як у звичайній таблиці. Наприклад, щоб прочитати список наявних авторів з вищенаведеного представлення, можна використати такий **SQL**-запит:

```
SELECT WISEMAN FROM WISEBOOKCOUNT
```

Слід зауважити, що сортування за допомогою оператора **ORDER BY** у самих представленнях не допускається – при спробі створити представлення із запитом, що містить **ORDER BY**, виникне помилка . Якщо треба відсортувати результати, що повертаються із представлення, то це слід зробити на стороні клієнта:

```
SELECT * FROM WISEBOOKCOUNT ORDER BY HOW_WISEBOOKS
```

Крім обмеження на застосування виразу **ORDER BY**, у представленнях не можна використовувати в ролі джерела даних набір даних, що отримується в результаті виконання збережених процедур.

1.2. Представлення, що модифікуються

Є два способи надати представленню можливість зміни даних у ньому. Перший спосіб – створити представлення на основі єдиної таблиці (або іншого представлення, що має можливість модифікації), причому всі стовпці даної таблиці повинні дозволяти наявність **NULL**. При цьому запит, на якому базується представлення, не може містити підзапитів, агрегатних функцій, **UDF**, збережених процедур, пропозицій **DISTINCT** та **HAVING**. Якщо виконуються всі ці умови, то представлення автоматично набуває здатності до модифікації, тобто для нього можна виконувати запити **DELETE, INSERT** та **UPDATE**, які будуть змінювати дані в таблиці-джерелі.

Як бачимо, список умов досить значний і сильно обмежує застосування таких представлень, тому вони використовуються відносно рідко.

Щоб зробити модифіковане представлення, що порушує перераховані вище умови, застосовується механізм тригерів. Так, для реалізації обновлюваного представлення за допомогою тригерів необхідно зробити наступне. Створити три тригери для даного представлення на події: **BEFORE DELETE**, **BEFORE UPDATE** та **BEFORE INSERT**. У цих тригерах описати, що має відбуватися з даними при видаленні, зміні й вставці рядка таблиці.

Коли **InterBase** виконуватиме запит на модифікацію даних у представленні , то перевірить, чи існують для нього відповідні тригери, тобто **BEFORE DELETE/ INSERT/ UPDATE**. Якщо тригер для виконуваної дії існує, то **InterBase** викличе його для модифікації реальних даних у таблицях, що лежать в основі представлення (хоча це можуть бути й інші дані), а потім перечитає змінений рядок.

Таким чином, існує можливість реалізувати складні ланцюжки відновлень даних у складних представленнях.

При створенні представлення може використовуватися опція **WITH CHECK OPTION**. Якщо для модифікованого представлення її буде зазначено, то кожен рядок даних, що вставляється або змінюється в ньому, буде перевірятися на умову " попадання" у представлення. Зокрема, якщо новий запис не задовольняє умовам базового для **VIEW** запиту, то вставка цього запису буде скасована й виникне помилка.

2. Обробка виняткових ситуацій

2.1. Виняткові ситуації

Виняткові ситуації (винятки) **InterBase** багато в чому схожі на виняткові ситуації інших мов високого рівня, однак мають свої особливості. Фактично виняткова ситуація в **InterBase** – це повідомлення про помилку, що має власне, задане програмістом ім'я й текст повідомлення про помилку. Створюється виняткова ситуація так:

```
CREATE EXCEPTION <ІМ'Я_ВИНЯТКОВОЇ_СИТУАЦІЇ> <ТЕКСТ_ВИНЯТКОВОЇ_СИТУАЦІЇ>;
```

Наприклад, можна створити виняткову ситуацію такого виду:

```
CREATE EXCEPTION TEST_EXCEPT 'MESSAGE ABOUT TEST EXCEPTION';
```

Виняткові ситуації легко видалити або змінити – видалення відбувається командою

```
DROP EXCEPTION <ІМ'Я_ВИНЯТКОВОЇ_СИТУАЦІЇ>
```

а зміна:

```
ALTER EXCEPTION <ІМ'Я_ВИНЯТКОВОЇ_СИТУАЦІЇ> <ТЕКСТ_ВИНЯТКОВІ СИТУАЦІЇ>
```

Щоб використовувати виняткові ситуації в збереженій процедурі або тригері, необхідно скористатися командою наступного виду:

```
EXCEPTION <ІМ'Я_ВИНЯТКОВОЇ_СИТУАЦІЇ>;
```

Розглянемо застосування винятків на прикладі збереженої процедури, що виконує ділення одного числа на інше й повертає результат. При цьому необхідно відстежити випадок ділення на

нуль і згенерувати виняткову ситуацію, якщо дільник дорівнює нулю. Для нашого прикладу створимо таку виняткову ситуацію:

```
CREATE EXCEPTION ZERO_DIVIDE 'Cannot divide by zero!';
```

Створимо збережену процедуру, що використовує цю виняткову ситуацію:

```
CREATE PROCEDURE SP_DIVIDE (DILENE DOUBLE PRECISION, DILNIK DOUBLE PRECISION)
RETURNS (RESULT DOUBLE PRECISION)
AS
BEGIN
 IF (DILNIK<0.0000001) THEN BEGIN
  EXCEPTION ZERO_DIVIDE;
  RESULT=0;
 END ELSE BEGIN
  RESULT=DILENE/DILNIK;
 END
 SUSPEND;
END
```

На вході ЗП отримує ділене (**DILENE**) і дільник (**DILNIK**), потім дільник порівнюється з числом **0.0000001**, (тобто – практично з нулем, в межах вибраної похибки) . Це робиться тому, що дійсні числа неможливо безпосередньо порівнювати через похибки у їх представленні в дробовій частині. Якщо **DILNIK** близький до нуля, то генерується виняткова ситуація **ZERO_DIVIDE**. Якщо викликати процедуру **SP_DIVIDE** з нульовим дільником в **ISQL**, те отримаємо відповідне повідомлення (рис. 3.1).

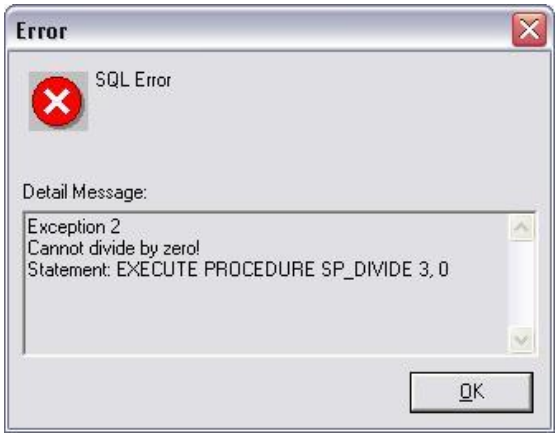


Рис. 3.1. Повідомлення про виняткову ситуацію в **ISQL**.

Якщо ж викликати цю процедуру з нульовим дільником у клієнтській прикладній програмі, то з’явиться стандартне повідомлення про помилку (рис. 3.2) .

Рис. 3.2. Повідомлення про виняткову ситуацію в клієнтській програмі.

Таким чином, повідомлення про помилку – це результат обробки сервером **InterBase** відповідної виняткової ситуації. Коли **InterBase** виявляє в процедурі або тригері виняткову

ситуацію, він перериває роботу цієї процедури та відміняє всі зміни, зроблені в поточному блоці **BEGIN... END**, причому якщо ЗП є процедурою вибірки, то скасовуються дії лише по відношенню до останнього оператора **SUSPEND.**

Щоб розроблювач зміг обробити виниклу виняткову ситуацію на рівні бази даних, використовується наступна конструкція мови ЗП і тригерів:

```
WHEN EXCEPTION <ІМ'Я_ВИНЯТКОВОЇ_СИТУАЦІЇ> DO
BEGIN
 /*ОБРОБКА ВИНЯТКОВОЇ СИТУАЦІЇ*/
END
```

Використання цієї конструкції допомагає уникнути появи стандартної помилки з текстом виняткової ситуації й зробити власні дії по її обробці.

Коли генеруються виняткові ситуації, відбувається наступне: виконання збереженої процедури (або тригера) переривається. **InterBase** починає шукати конструкцію **WHEN EXCEPTION... DO** для обробки виниклої виняткової ситуації в поточному блоці **BEGIN...END**. Якщо вона відсутня, то піднімається на рівень вище (якщо є вкладені блоки **BEGIN...END** або коли одна ЗП викликана з іншої) і шукає оброблювач виняткові ситуації там і т.д., доки не знайде підходящий оброблювач виняткової ситуації, або не закінчиться вкладеність рівнів збереженої процедури. Якщо оброблювач виняткові ситуації так і не був знайдений, то повертається стандартне повідомлення про помилку, що включає текст виняткової ситуації. Якщо оброблювач знайдено, то виконуються дії в його блоці **BEGIN...END** і керування передається на перший оператор, що міститься за **END** оброблювача.

Розглянемо приклад обробки виняткової ситуації, що генерується в розглянутій вище процедурі **SP_DIVIDE**. Припустимо, що є деяка зовнішня процедура, що викликає **SP_DIVIDE** для того, щоб поділити два числа. Звичайно, даний приклад не має практичної цінності, але він дозволяє пояснити спосіб використання винятків.

Розглянемо текст збереженої процедури:

```
CREATE PROCEDURE SP_TEST_EXCEPT(DILNIK DOUBLE PRECISION)
RETURNS (RSLT DOUBLE PRECISION, STATUS VARCHAR(SO))
AS
BEGIN
 STATUS='Everything is Ok';
 SELECT RESULT FROM SP_DIVIDE(12,:DILNIK) INTO :RSLT;
 SUSPEND;
 WHEN EXCEPTION ZERO_DIVIDE DO BEGIN
  STATUS='Zero value found!';
  RSLT=-1;
  SUSPEND;
 END
END
```

Ця процедура викликає процедуру **SP_DIVIDE**. Якщо параметр **DILNIK** не дорівнює нулю, то процедура **SP_DIVIDE** виконується й повертає значення – результат ділення, а статусна змінна **status** приймає значення **'Everything is Ok'**. Якщо ж виникла виняткова ситуація, то результуюча змінна **rslt** буде дорівнювати **-1**, а змінній **status** буде присвоєне повідомлення про помилку – **'Zero value found!'**. Звичайно, в оброблювачі виняткової ситуації можна зробити й більш складну обробку, наприклад записати некоректні дані в особливу таблицю, або спробувати змінити дані і повторити виконання операції тощо.

2.2. Обробка помилок SQL та InterBase

Помилка **InterBase** – це також виняткова ситуація, тільки генерується вона самим сервером. Принцип обробки помилок такий самий, як і винятків: якщо виникає помилка, то сервер шукає її

оброблювач, послідовно переглядаючи всі рівні вкладеності (якщо вони є) збереженої процедури, починаючи з того рівня, на якому виникла помилка.

Якщо оброблювач знайдений, то виконується його код, а потім керування передається на перший оператор за оброблювачем.

Конструкція, за допомогою якої відбувається обробка помилок, така ж, як і для обробки винятків, тільки замість оператора **EXCEPTION** застосовується оператор **GDSCODE** або **SQLCODE**:

```
WHEN GDSCODE [SQLCODE] <КОД_ПОМИЛКИ> DO
BEGIN
 /*ОБРОБЛЯЄМО ПОМИЛКУ*/
END
```

Залежно від того, чи в конструкції обробки помилок стоїть **GDSCODE**, чи **SQLCODE**, обробляються різні помилки. Оператор **SQLCODE** означає, що обробляються помилки **SQL**, а **GDSCODE** – помилки **InterBase.** Прикладом помилки **SQL** є помилка з **SQLCODE=-802 "Arithmetic exception, numeric overflow, or string truncation"** або **SQLCODE=-817 "Attempted update during read-only transaction".** Прикладом помилки **InterBase** є помилка **isc_bad_dbkey 3355443221 "invalid database key".**

Таким чином, всередині збереженої процедури можна "перехопити" практично будь-яку помилку й коректно на неї відреагувати. Можна перераховувати оброблювачі помилок один за одним, щоб визначити реакцію на різні помилки.

Для того щоб написати безумовний оброблювач, що реагує на будь-яку помилку **SQL**, **InterBase** або виняткові ситуації, слід використати конструкцією **WHEN... DO** з ключовим словом **ANY**:

```
WHEN ANY DO
BEGIN
 /*ДІЇ ПРИ БУДЬ-ЯКІЙ НЕСТАНДАРТНІЙ ОПЕРАЦІЇ, ПОМИЛЦІ АБО ВИНЯТКУ */
END
```

За допомогою використання описаних механізмів можна передбачити розвинені засоби обробки помилок, які зроблять прикладну програму для роботи з базою даних значно стійкішою в роботі.

3. Події InterBase

Однією з важливих можливостей **InterBase**, що часто використовується у тригерах, є події (**events**). Події – це рядкові повідомлення, які можуть бути послані з тригера або збереженої процедури. Отримають ці повідомлення ті клієнти **InterBase**, які зареєстровані як зацікавлені в даних подіях. Таким чином, можна сповіщати клієнта про якісь зміни всередині бази даних.

Події не є постійним об'єктом бази даних – вони ніде в базі не зберігаються, не створюються й не модифікуються, а генеруються під час виконання. Щоб послати якусь подію, використовується наступна конструкція:

```
POST_EVENT 'ТЕКСТ_ПОВІДОМЛЕННЯ';
```

Вміст текстового повідомлення може братися із змінної і, таким чином, можна породжувати події динамічно, наприклад так:

```
IF (<ЯКАСЬ УМОВА>) THEN BEGIN
 EVENT_TEXT ='IT IS TRUE!';
END ELSE BEGIN
 EVENT_TEXT ='IT IS FALSE!';
END
POST_EVENT :EVENT_TEXT;
```

Проте, якщо жодна клієнтська програма, з'єднана з базою даних, не зареєстрована на отримання цих подій, то всі вони "втратяться".

Для реєстрації (підписки) на отримання потрібних подій використовують спеціальні функції **InterBase API**, які реалізовані, наприклад, у бібліотеці **IBExpres** (для **Delphi** та **C++ Builder**) – у компоненті **IBEvents.**

Як тільки прикладна програма реєструється для отримання події, запис про це заноситься у спеціальну таблицю **InterBase**, що є єдиною для всіх користувачів, і сервер починає переглядати всі генеровані події на предмет появи подій для даного клієнта. Якщо така подія з'являється, то клієнтська програма отримує відповідний сигнал, на який може якось відреагувати. Зокрема, події в тригерах є зручним механізмом для організації протоколу змін у певних таблицях.

4. Функції користувача (User Defined Functions)

Функції користувача є потужним засобом, який дозволяє значно розширити стандартний набір функцій **InterBase** практично будь-якими доповненнями, що потрібні в конкретній базі даних. Таким чином розроблювач може реалізувати для своїх програм навіть такі функції, які ніколи не входять у поставку серверів баз даних.

4.1. Механізм підключення функцій

Спеціально для розширення функціональності **SQL InterBase** пропонує механізм функцій, визначених користувачем (**user defined functions**). Для цього за допомогою будь-якої системи розробки створюється динамічна бібліотека (**Dynamic Link Library**) . Як середовище розробки можна використати **Borland Delphi, Borland C++ Builder, Microsoft Visual C++** тощо. Далі, необхідно помістити отриману **DLL** у каталог, з якого **InterBase** зможе її викликати, і оголосити потрібні функції з **DLL** у своїй базі даних за допомогою команди **DECLARE EXTERNAL FUNCTION**. Після цього такі функції можна викликати так, ніби вони є звичайними вмонтованими функціями **InterBase.**

InterBase до версії **6.0** вимагав, щоб **DLL** перебувала в одному з каталогів, зазначених у системній змінній **PATH. InterBase 6** і вище (включаючи клони **Firebird** та **Yaffil**) вимагає, щоб **DLL** була поміщена в спеціальний каталог **UDF**, що перебуває в загальному каталозі установки **InterBase.**

4.2. Створення власних функцій

Продемонструємо написання й підключення однієї функції на прикладі. Як засіб розробки використаємо **Borland Delphi** . Створимо функцію, що буде перетворювати літери рядка до верхнього регістра. Текст відповідної динамічної бібліотеки для середовища **Delphi** наступний:

```
library TestUDF;
uses SysUtils;
function malloc(Size: Integer): Pointer; cdecl; external 'msvcrt.dll';
function StrUpperCase(sz: PChar): PChar; cdecl; export;
var Tmp: string;
begin
 Tmp := AnsiUpperCase(sz);
 Result := malloc(length(Tmp) + 1);
 StrPCopy(Result, Tmp);
end;

exports StrUpperCase;

begin
end.
```

Динамічна бібліотека експортує тільки одну функцію: **StrUpperCase**. Для передачі рядкових параметрів, так само як і результату функції, використовується тип **PChar**, тобто динамічний рядок, обмежений символами **#0**. Вміст коду функції очевидний: рядок **sz** приводиться до верхнього регістра за допомогою стандартної функції **AnsiUpperCase**. Дана функція коректно працює з літерами кирилиці, якщо в системі встановлена відповідна кодова сторінка. Після цього виділяється пам'ять для результуючої змінної за допомогою **malloc** – стандартної функції **Windows.** Далі значення тимчасової змінної **Tmp** копіюється у змінну **Result**. Дану бібліотеку слід

скомпілювати і помістити отриманий файл **TestUDF.dll** у потрібний каталог. Для **InterBase 6.x** або його клонів це каталог **\Udf**, що перебуває в каталозі установки сервера.

Після цього необхідно зареєструвати функцію в базі даних. Для реєстрації слід виконати команду **DECLARE EXTERNAL FUNCTION**, що має наступний синтаксис:

```
DECLARE EXTERNAL FUNCTION name [datatype | CSTRING (int)
[, datatype | CSTRING (int) ...] ]
RETURNS (datatype [BY VALUE] | CSTRING (int)} [FREE_IT]
ENTRY_POINT 'entryname'
MODULE_NAME ' modulename';
```

Параметр **name** – це ім 'я функції всередині бази даних. Воно не обов'язково повинно збігатися з реальною назвою функції в **DLL**.

Параметр **datatype** визначає тип параметрів. На параметри накладаються наступні обмеження:

• всі параметри передаються за посиланням (через вказівники);

• вихідний параметр (значення функції) може повертатися за значенням;

• параметри не можуть бути масивами.

Якщо необхідно застосовувати рядкові параметри, то слід використати тип **CSTRING**. У дужках при цьому необхідно вказати максимальну довжину рядка. Якщо рядок є результатом функції, вона завжди передається за посиланням, а не за значенням.

Необов’язковий параметр **FREE_IT** вказує **InterBase**, що після виконання функції необхідно автоматично звільнити пам'ять, виділену для параметрів. Дана опція потрібна тільки в тому випадку, якщо бібліотека сама виділяла пам'ять під якісь параметри функції.

У параметрі **entryname** необхідно вказати назву функції в **DLL**.

У параметрі **modulename** необхідно вказати назву файлу **DLL**, у якому перебуває оголошувана функція користувача. Параметри **entryname** й **modulename** чутливі до регістру літер.

Таким чином, щоб додати спроектовану вище функцію в базу даних, необхідно виконати наступну команду:

```
DECLARE EXTERNAL FUNCTION USTRUPPERCASE
cstring(254)
RETURNS cstnng(254) FREE_IT
ENTRY_POINT 'StrUpperCase' MODULE_NAME 'TestUDF.dll'
```

Після цього нову функцію **USTRUPPERCASE** можна використовувати у будь-якому **SQL**-запиті, наприклад:

```
SELECT USTRUPPERCASE(DEPARTMENT) FROM DEPARTMENT
```

Простий, але дуже потужний механізм **UDF** дозволяє реалізувати систему бізнес-правил у базах даних набагато більш ефективно й зручно. Механізм функцій користувача в **InterBase** має тільки одне суттєве обмеження – він не дозволяє обробляти параметри **NULL**. В іншому функціональність **UDF** для **SQL** залежить тільки від конкретних потреб розроблювача.

В Інтернеті на відповідних сайтах можна знайти широкий вибір бібліотек з функціями користувача для різноманітних потреб, що можуть виникати про розробці конкретних прикладних програм для баз даних.

5. Завдання на лабораторну роботу

Для бази даних, створеної протягом лабораторних робіт №1-2, виконати наступні завдання:

1. Створити два представлення: одне просте і одне складне (на основі кількох таблиць).
2. Спроектувати для однієї з таблиць тригер, який перевірятиме нові дані, що вносяться у таблицю, на відповідність певним умовам, і у випадку їх неправильності генеруватиме виняткову ситуацію.
3. Створити функцію користувача, яка виконуватиме якісь дії над даними у базі.

Лабораторна робота №9

Тема: Архітектура прикладної програми для роботи з базою даних.

1. Загальна структура програми, що працює з БД

Прикладні програми для роботи з базами даних, як правило, складаються з інтерфейсу користувача, компонентів, що дають доступ до БД, та компонентів, що з'єднують їх один з одним і з джерелом даних. Поєднуючи ці компоненти в певній послідовності, можна досить легко розробити програму, що взаємодіє з базою даних. Типова схема такої програми приведена на рис. 4.1.

Рис. 4.1. Типова схема прикладної програми для роботи з БД.

Як видно з малюнка, така програма містить поєднання інтерфейсу користувача і модуля даних. Модуль даних призначений для зберігання компонентів, що працюватимуть з БД. Серед таких компонентів найважливіші: набір даних, який містить у собі дані з бази, та джерело даних, що надає їх іншим частинам програми. При цьому програма може містити довільне число форм і використовувати будь-яку парадигму роботи із документами (**MDI** або **SDI**).

Візуальні компоненти для відображення даних розташовані на вкладці **Data Controls**. Вони, як правило, є стандартними елементами керування з невеликими змінами й доповненнями, призначеними для полегшення роботи з БД.

В основі будь-якої програми для БД лежать набори даних, що представляють собою масиви записів, отримані з бази. Набори даних зберігаються в спеціальних компонентах, побудованих на базі класу **TDataSet**. Для забезпечення зв'язку набору даних з візуальними компонентами відображення даних призначений спеціальний компонент – джерело даних, реалізований класом **TDataSource**. Компонент **TDataSource** забезпечує передачу даних у візуальні компоненти, відслідковує синхронну зміну їхнього стану, якщо змінюється стан пов'язаного з ними набору даних, та передачу змінених даних назад у набір. Цей компонент розміщено на вкладці **Data Access**.

Базовий механізм доступу до даних реалізовано за допомогою таких компонентів (див. схему на рис. 4.2):

- компоненти-нащадки класу **TDataSet** (набори даних);
- компоненти **TDataSource** (джерело даних);
- візуальні компоненти відображення даних, що формують інтерфейс користувача (**DBGrid**, **DBComboBox**, **DBLabel** та інші).

Для зв’язку з БД можна використовувати також компонент **TDataBase**, який забезпечує ряд переваг порівняно з безпосереднім зв’язком через набір даних.

При відкритті набору даних у нього автоматично передаються записи з таблиці бази даних, і курсор (вказівник) встановлюється на перший запис. При переміщенні по наборі значення записів автоматично оновлюються (або не оновлюються, залежно від розташування й типу курсора бази даних).

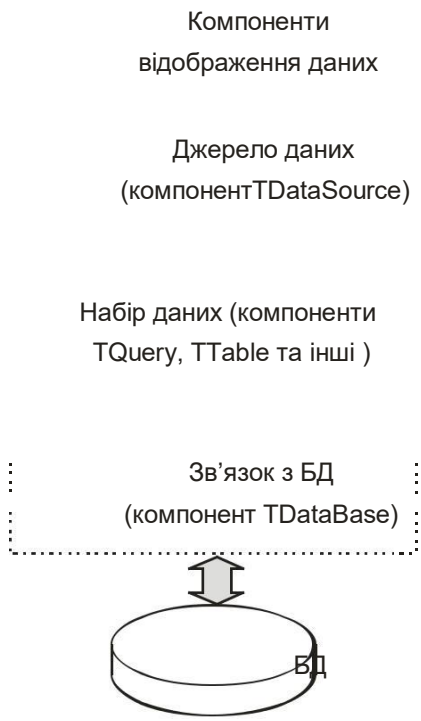


Рис. 4.2. Механізм доступу до даних.

1.1. Модуль даних

Модуль даних (**Data Module**) – це невізуальний контейнер, у якому розміщуються компоненти доступу до даних. У ньому можна розміщати тільки невізуальні компоненти. Модуль даних доступний програмістові на етапі розробки у вигляді форми, в яку можна покласти компоненти й налаштувати їхні властивості (рис. 4.3) . Модуль даних відрізняється від звичайної форми тим, що бере свій початок від класу **TComponent.**

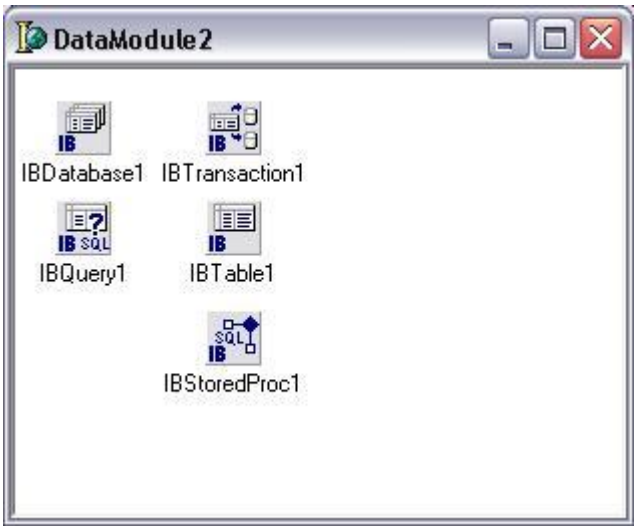


Рис. 4.3. Модуль даних.

Для створення модуля даних можна скористатися командою меню **New/ Data Module.**

Вкладка **Diagram** редактора коду відкриває вікно формування моделей даних (діаграм, схем). Моделі даних дозволяють наочно відобразити зв'язки, що існують між наборами даних. Це особливо зручно, якщо доводитися працювати з великою кількістю таблиць.

Для роботи з діаграмами даних використовується "дерево об'єктів", вікно якого можна викликати командою меню **View/ Object TreeView**. Воно дозволяє відображати в списку об'єкти, розміщені в модулі даних. Перетягуючи об'єкти з "дерева об'єктів" у робочу область **Diagram**, можна легко налаштувати їх властивості, з'єднуючи лініями зв'язку. На рис. 4.4 показано приклад

моделі бази даних. За допомогою кнопки **Property Connector** легко зв'язати таблиці з компонентом-провайдером.

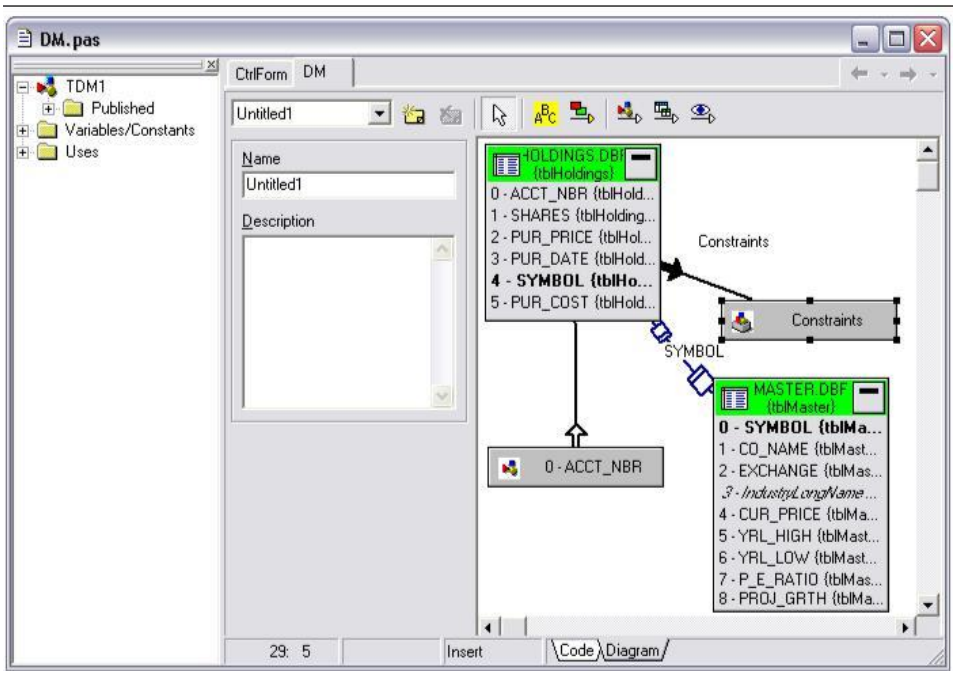


Рис. 4.4. Діаграма моделі даних БД.

Використання модуля даних при розробці програми дає програмістові ряд переваг. Наприклад , при зміні якоїсь властивості об'єкта в модулі зміни автоматично відобразяться у всіх пов'язаних з ним компонентах. Також важливою перевагою є те, що всі компоненти, які забезпечують доступ до бази даних, зібрані в одному місці. Це полегшує сприйняття бази в цілому. Хоча слід відмітити, що використання модуля даних необов’язкове, і відповідні компоненти можна розмістити на одній з форм програми.

Компонент набору даних є основою програми, що взаємодіє з БД. Він містить дані з вибраних таблиць і дозволяє ефективно управляти ними. У процесі роботи даний компонент використовує функції відповідної технології через сукупність інтерфейсів (методів, наданих даною технологією). Розглянемо основні кроки, потрібні для підключення набору даних і одержання інформації з нього:

1. Помістити компонент у модуль даних і зв'язати його з базою даних.
2. Підключити до компонента таблицю, використовуючи властивість **TableName.**
3. Активізувати зв'язок, встановивши для компонента значення властивості **Active** у **True.**
4. Розмістити на формі компонент **TDataSource.** Він забезпечує взаємодію візуальних компонентів відображення даних з набором даних.
5. Зв'язати набір даних і компонент **TDataSource.**
6. Зв'язати візуальні компоненти відображення даних з компонентом **TDataSource.**

Джерело даних – компонент **TDataSource** – зв'язується з набором даних. Цей зв'язок здійснюється через властивість **DataSet**, що містить інформацію про поточний набір даних. Компонент **DataSource** має набір властивостей і методів, які призначені для керування його роботою.

Властивість **AutoEdit** автоматично переводить набір даних у стан редагування, якщо має значення **True**, коли зв'язаний елемент вводу отримує фокус.

Метод **Edit** переводить пов'язаний набір даних у стан редагування.

Метод-оброблювач **OnDataChange** викликається при редагуванні даних у зв’язаному візуальному компоненті.

Метод -оброблювач події **OnUpdateData** викликається перед тим, як змінені дані будуть збережені в наборі даних. Оброблювач викликається перед виконанням методу **Post.**

Метод-оброблювач події **OnStateChange** викликається, коли змінюється стан зв'язаного набору даних.

1.2. Набір даних

Масив записів, отриманий програмою за власним запитом, називається набором даних. Набір даних як об'єкт веде свій початок від класу **TDataSet** й успадковує його властивості.

Клас **TDataSet** є базовим класом ієрархії, він інкапсулює абстрактний набір даних і реалізує загальні методи роботи з ним . На основі базового класу реалізовані спеціальні компоненти, що надають розроблювачу доступ до тієї або іншої технології.

Для читання або запису в набір даних його необхідно відкрити. Це можна зробити двома способами: присвоїти властивості **Active** значення **True** або викликати метод **Open.**

Завершення роботи з набором даних проводитися подібним способом: властивості **Active** присвоюється значення **False**, або викликається метод **Close**. Перед відкриттям набору даних автоматично ініціюється подія **BeforeOpen**, а після відкриття, коли в набір будуть передані дані, активується подія **AfterOpen.**

При закритті набору даних викликаються методи **BeforeClose** й **AfterClose.**

Використовуючи ці методи-оброблювачі, можна виконати в програмі деякі дії, – наприклад, при закритті набору даних перепитати користувача, чи слід прийняти зроблені зміни:

```
procedure TForml.CustTableVerifyBeforeClose(DataSet: TDataSet);
begin
 if (CustTable.State in [dsEdit, dsInsert]) then begin
  case MessageDlg(Зберегти зміни?', mtConfirmation, mbYesNoCancel , 0) of
   mrYes                  : CustTable.Post;   { Збереження змін }
   mrNo                   : CustTable.Cancel; { Скасування змін }
   mrCancel : Abort;      { Припинення закриття набору даних }
  end;
 end;
end;
```

У приведеному прикладі здійснюється обробка події, що виникає перед тим, як набір даних буде закритий.

1.3. Навігація по набору даних

Запис, вибраний у даний момент в наборі даних, називається курсором. Після відкриття набору даних курсор автоматично встановлюється на перший запис. Для переміщення по набору даних використовуються методи **Next, Prior**, **First** і **Last.**

Методи **Next** і **Prior** переміщають курсор на наступний та попередній запис відповідно, методи **First** і **Last** – на перший та останній.

Властивість **Eof** логічного типу показує, чи досяг курсор набору даних останнього запису. Номер поточного запису і переміщення на певний запис за його номером можна виконати за

допомогою властивості **RecNo** цілого типу, присвоюючи або отримуючи її значення (проте, ця дія недоступна для БД з архітектурою "клієнт-сервер").

Для переміщення по набору даних використовується метод **MoveBy** (для табличних БД). Параметр методу **Distance** вказує число записів, на яке буде здійснений перехід. Якщо параметр має негативне значення, курсор зміщується назад.

Щоб з'ясувати розмір запису в байтах, можна звернутися до властивості **RecordSize.**

У тих випадках, коли потрібне переміщення по великому наборі даних, для прискорення можна відключати відображення інформації з набору у зв'язаних елементах відображення. Цю можливість використовують для прискорення роботи програми під час масової обробки даних. Протягом цього часу відображати зміни у вікні не потрібно, тому оновлення вмісту відповідних

елементів керування тимчасово відключається. Для цього використовуються методи **DisableControls**

і **EnableControls.** Перший з них вимикає відображення інформації в елементах керування вікна, а другий – повертає їм цю здатність.

Для одержання числа записів, що утримуються в наборі, можна використати властивість

RecordCount.

1.4. Редагування набору даних

Перед тим як змінювати набір даних, варто довідатися, чи можна його міняти, за допомогою властивості **CanModify**, яка приймає значення **True**, якщо набір даних може бути змінений. Метод **Edit** переводить набір даних у стан редагування. У деяких випадках набір даних переводиться в стан редагування автоматично, наприклад при його зміні через зв'язані елементи редагування або при використанні деяких методів, таких як **Insert** або **Append.**

Для збереження змінених даних викликається метод **Post**. Цей метод може викликатися як розроблювачем, так і самим набором даних при переході на інший запис. Для різних типів баз даних дія методу **Post** трохи розрізняється:

- Для наборів даних, зв'язаних з базою даних прямо, зміни зберігаються відразу на диск.
- При використанні клієнтських наборів зміни зберігаються в локальному кеші бази даних. Для їхнього збереження на сервері необхідно викликати метод **ApplyUpdates.**

У деяких ситуаціях буває необхідно скасувати зроблені дії. В цьому випадку викликається метод **Cancel**. Метод повертає набір даних у стан, що був при останньому виклику методу **Post.**

Для додавання нового запису по місцю розташування курсора використовується метод **Insert.** А якщо необхідно додати запис у кінець набору даних, потрібно викликати метод **Append.**

Вибраний запис видаляється методом **Delete**. А метод **ClearFields** очищає поля активного запису. При цьому набір даних повинен перебувати в режимі введення нового запису або в режимі редагування.

1.5. Пошук записів і фільтрація в наборах даних

Метод **Locate** використовується для пошуку інформації в базі даних. Він шукає перший запис, що задовольняє критерію пошуку, і якщо такий запис знайдений, робить його поточним. У випадку вдалого пошуку метод повертає **True**, у противному випадку – **False.**

До складу параметрів цього методу входить список **KeyFields**. У цьому параметрі вказується список полів (мінімум одне), за якими буде проводитися пошук. Якщо пошук буде здійснюватися за кількома полями, їхні назви перелічуються через крапку з комою. Значення полів, за якими проводиться пошук, задаються у варіантному масиві **KeyValues.**

У властивості **TLocateOption** задаються необов'язкові додаткові опції пошуку:

- Значення **loCaseInsensitive** вказує, що пошук ведеться без врахування регістру символів.
- Значення **loPartialKey** використовується, якщо значення полів пошуку можуть містити тільки частину введеного значення. Наприклад, якщо в ключі пошуку задане значення 'Com', а поле містить значення 'Computer', те метод зупинить пошук на цьому записі, оскільки він містить у своєму полі частину ключового значення.

Метод **Locate** дозволяє робити пошук по будь-якому полю. Поля, за якими буде проводитися

пошук, можуть не перебувати в первинному ключі й можуть не індексуватися. Але якщо поле, за яким здійснюється пошук, включено в той чи інший індекс, метод використає його.

Приклад використання методу **Locate**:

```
var LocateSuccess: Boolean;
    SearchOptions: TLocateOptions:
begin
 SearchOptions:=[loPartialKey];
 LocateSuccess:=CustTable.Locate('Company;Vendor',VarArrayOf(['Sight','Point']),
```

```
      SearchOptions);
    end;
```

У цьому прикладі проводиться пошук у наборі даних по полях **Company** й **Vendor.**

Проте, пошук можна проводити з більш жорсткими умовами. Метод **Lookup** знаходить запис, що точно задовольняє умовам пошуку. Цей метод не переводить курсор на знайдений запис, але повертає значення його деяких полів у вигляді варіантного масиву.

У списку **KeyFields** вказується список полів, по яких буде здійснюватися пошук. Якщо пошук ведеться по кількох полях, то вони розділяються крапкою з комою. У масиві **KeyValues** вказуються ключові значення, по яких буде проводитися пошук. В списку **ResultFields** вказуються поля, які будуть повернуті у варіантному масиві, якщо пошук виявиться успішним.

У даному прикладі проводиться пошук в полі **Company** значення "Запорізька Січ". У результаті отримується варіантний масив, що містить значення полів **Company, Contact** і **Phone.**

```
    var LookupResults : Variant;
    begin
LookupResults := CustTable.Lookup('Company', 'Запорізька Січ',
'Company;Contact;Phone'); end;
```

У випадку, якщо метод нічого не знайшов, він повертає значення **null**. Перевірити дану умову можна за допомогою оператора

```
    if VarType(LookupResults) = varNull then ...
```

Властивість **Filter** набору даних дозволяє задати критерії фільтрації. Набір даних буде відфільтровано, як тільки його властивість **Filtered** прийме значення **True**. В умову фільтрації можна включати логічні оператори and, or, not, **<>, <, >, <=, >=**. Наприклад:

```
([size] > 20) and ([size] < 40)
([size] > 20) and ([weight] < 30)
([name] <> 'Petross') and ([size] < 10)
```

Як видно з цих прикладів, порівняння заданої умови проводиться із значенням поля, яке вказується у квадратних дужках.

За допомогою властивості **FilterOptions** можна задати необов'язкові параметри фільтрації, як і для методу **Locate.**

Подія **OnFilterRecord** виникає, коли у властивості **Filtered** встановлюється значення **True.** Метод-оброблювач події **FilterRecordEvent** має два параметри. У параметрі **DataSet** передається фільтрований набір даних, а в параметрі **Accept** – змінна, у яку поміщується значення **True**, якщо поточний запис відповідає умовам фільтрації (або **False** – у протилежному випадку). Наприклад:

```
    procedure TForm1.Table1FilterRecord(DataSet: TDataSet; var Accept: Boolean):
    begin
Accept :=(DataSet.FieldByName('Size').AsInteger<10) and
 ((DataSet.FieidByName('Weight' ).AsFloat/1000)<2);
    end;
```

Зауважимо, що здійснення фільтрації описаним методом часто виявляє дуже низьку швидкодію, тому його варто використовувати для відносно малих наборів даних. В сучасних умовах практично всі операції вибірки, сортування, пошуку та фільтрування здійснюються засобами **SQL.**

1.6. Стани набору даних

Набір даних може знаходитися у кількох станах, в які він переходить під час свого функціонування. Стани міняються в результаті переміщення по даних, редагування, введення нових записів і т.д. Інколи, відслідковуючи поточний стан набору даних, можна проводити якісь дії. Поточний стан набору даних міститься у властивості **State**, значення якої належить перелічуваному типу **TDataSetState**. Можливі стани зазначені в таблиці №4.1.

Таблиця №4.1

Значення	Стан набору даних
dsInactive	Набір даних закритий
dsBrowse	Набір даних знаходиться у режимі перегляду. Редагування неможливе. Даний стан є найбільш "звичним", встановлюється за замовчуванням
dsEdit	ктивний запис можна редагувати
dsInsert	Новий запис поміщено у набір даних, але не затверджено. Запис можна прийняти, змінити чи відхилити
dsSetKey	Використовується механізм пошуку
dsCalcFields	Встановлюється при виникненні події **OnCalcField** (при розрахунку обчислюваного поля)
dsFilter	Встановлюється при виникненні події **OnFilterRecord** (при здійсненні фільтрації)
dsNewValue	Встановлюється при звертанні до значення **NewValue** поля набору даних
dsOldValue	Встановлюється при звертанні до значення **OldValue** поля набору даних
dsCurValue	Встановлюється при звертанні до значення **CurValue** поля набору даних
dsBlockRead	Встановлюється при блочному переміщенні по наборі даних (режим стає активним після виклику методу **DisableControls**).
dsOpening	Встановлюється при відкриванні набору даних

Набір даних може переходити в стани **dsNewValue**, **dsOldValue**, **dsCurValue** тільки в тому випадку, коли доступ до бази даних здійснюється через компонент **TClientDataSet** або використовується кеш даних.

Коли набір відкривається, він перебуває в стані **dsOpening**. Після того як він відкрився, активується стан перегляду **dsBrowse**. Цей стан є основним для набору даних.

Для перевірки, чи перебуває набір даних у стані редагування або вставки нового запису можна використати такий код:

```
with DataSet do begin
 if State in [dsInsert, dsEdit] then Post;
end;
```

2. Робота з полями

Будь-який запис набору даних є сукупністю полів. Поля – це об'єкти, похідні від типу **TField**. Кожне поле має певний тип даних і відповідний йому об'єкт. Властивість **Fields** містить множину полів набору даних.

2.1. Використання об'єктів-полів

До поля набору даних можна звернутися по його імені, використовуючи метод **FieldByName**, або через властивість **Fields**. У першому випадку використовується конструкція виду

DataSet.FieldByName("Назва поля").AsInteger, а в другому – **DataSet.Fields[i].AsInteger** (звичайно, даний приклад передбачає, що у полі містяться дані цілого типу. Якщо тип даних інший, то слід використовувати інший метод приведення типу – **AsFloat**, **AsString** тощо).

Кожен об'єкт поля має ряд властивостей, що визначають його структуру: назва, тип, вид, розмір, та інші атрибути, характерні полю конкретного типу.

Властивість **FieldDefs** містить список параметрів кожного поля з набору даних. За допомогою методу **AddFieldDef** можна додати у набір даних інші поля з певними властивостями:

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
begin
 with ClientDataSet1 do begin
  with FieldDefs.AddFieldDef do begin
   DataType:=ftInteger;
   Name:='Field1':
  end;
  with FieldDefs.AddFieldDef do begin
   DataType:=ftString;
   Size:=10;
   Name:='Field2';
  end;
 end;
end;
```

Як видно із прикладу , кожне поле має певний тип даних (**DataType**), розмір (**DataSize**) і назву (**FieldName**). Також, кожне поле має якийсь вид (**FieldKind**), що визначає його функціональне призначення.

Властивість **FieldCount** повертає число полів, що утримуються в наборі даних. Часто потрібно отримати також додаткову інформацію про поля. Для цієї мети використовується метод **GetFieldNames**. Як параметр цьому методу передається список. Після виконання методу в переданому списку будуть розміщені назви всіх полів даного набору даних.

Заголовок стовпця міститься у властивості **DisplayLabel**. Вона відображається, зокрема, в компоненті таблиці (**DBGrid**) як заголовок стовпця. Заголовок стовпця відображуваної таблиці може не збігатися з ім'ям відповідного поля.

2.2. Статичні й динамічні поля

Існує два способи задання полів у наборі даних. За замовчуванням використовується динамічний спосіб. Проте можна примусово використати статичний спосіб формування списку полів.

Динамічні поля створюються автоматично при кожному відкритті набору даних, якщо раніше не були створені статичні об'єкти полів. Будь-який об 'єкт поля є спадкоємцем класу **TField**, а його конкретний тип залежить від типу даних, що містяться в таблиці.

Статичні поля створюються на етапі розробки, їхні властивості доступні в інспекторі об'єктів. Також на етапі розробки статичному полю можна присвоїти ім'я.

Компонент набору даних після підключення до таблиці бази даних без додаткових налаштувань використовує тільки динамічні поля. Якщо на етапі розробки задано хоч одне статичне поле, динамічні поля створюватися вже не будуть.

Додати до списку статичних полів нове поле дуже просто, для цього необхідно вибрати команду **Fields Editor** контекстного меню набору даних або клацнути на ньому два рази.

На рис. 4.5 показано вигляд редактора полів набору даних. У наборі визначено кілька статичних полів.

Рис. 4.5. Редактор полів

2.3. Типи й види полів

Функціональне призначення поля визначається властивістю **FieldKind**. У загальному випадку його призначення визначається автоматично на етапі створення поля. Вид поля можна змінити у процесі роботи програми. Проте , якщо новий тип даних поля не сумісний із значенням поля, то буде виведене повідомлення про помилку.

Значення властивості **TFieldKind** належить до перелічуваного типу. Відповідно, існує обмежена кількість значень цієї властивості:

fkData – поле даних, що має той же тип, що й зв'язане поле бази даних;

fkCalculated – обчислюване поле;

fkLookup – поле синхронного перегляду;

fkInternalCalc – внутрішнє обчислюване поле;

fkAggregate – агрегатне поле.

Тип даних однозначно пов'язаний з конкретним полем таблиці бази даних. Без цього поля саме поняття типу даних не має практичного змісту. В **Delphi** властивості абстрактного поля інкапсулює клас **TField**, що не має наперед визначеного типу. Від цього класу походить множина класів для типизованих полів, кожний з яких може працювати із своїм типом даних.

Тип поля задається у властивості **DataType**. На основі типів полів і класу **TField** створюються об'єкти полів певного класу. Список цих класів приведено у таблиці №4.2.

Таблиця №4.2.

Клас	Опис поля
TADTField	Поле абстрактного типу даних. Може містити масиви, набори даних та будь-які інші типи даних
TAggregateField	Агрегатне поле
TArrayField	Поле містить масив
TDateField	Поле дати
TDateTimeField	Поле дати і часу
TSmallIntField	Ціле поле (розмір – два байти)
TFloatField	Поле дійсного числа
TSQLTimeStampField	Поле дати і часу набору **DBExpress**
TAutoIncField	Автоінкрементне поле
TFMTBCDField	Форматоване двійково-десяткове поле
TStringField	Рядкове поле
TBCDField	Двійково-десяткове поле
TGraphicField	Графічне поле
TTimeField	Поле часу
TBinaryField	Двійкове поле
TGuidField	Поле, що зберігає унікальний ідентифікатор **GUID**
TVarBytesField	Байтове поле змінної довжини
TBlobField	**BLOB**-поле
TIDispatchField	Поле, що містить вказівник на інтерфейс диспетчеризації
TVariantField	Варіантне поле
TBooleanField	Логічне поле
TIntegerField	Поле для зберігання цілих чисел довжиною 32 біти
TWideStringField	Велике рядкове поле

TBytesField	Байтове поле фіксованої довжини
TinterfaceField	Поле, що містить вказівник на інтерфейс
TWord Field	Поле для 16-розрядних цілих беззнакових чисел
TCurrencyField	Грошове поле
TLargeintField	Поле для зберігання цілих чисел довжиною 64 біти
TDataSetField	Поле, що містить вкладений набір даних
TMemoField	Memo- поле

2.4. Стандартні компоненти для відображення даних

Найпоширенішим компонентом, без якого не обходиться практично жодна програма для БД, є компонент **DBGrid**, що реалізує табличне подання даних (рис. 4.6).

Компонент **DBGrid** використовується для відображення вмісту набору даних у табличному виді, коли рядки відповідають записам набору даних, а стовпці – полям запису.

	Таб. ном.	Прізвище, ім'я, по-батькові	Відділ	Посада
	1	Вікторенко Ю.Ю.	Головний офіс	Директор
	2	Старченко В.Л.	Гараж	Майстер
	3	Маринін П.Д.	Головний офіс	Зам. директора по госп. частин
	4	Хапченко Р.І.	Бухгалтерія	Головний бухгалтер
	5	Несенко У.К.	Бухгалтерія	Бухгалтер
	6	Колісник А.А.	Гараж	Шофер
	7	Лопатенко Д.В.	Гуртожиток №1	Зав. гуртожитком №1
▶	8	Кардаш Т.Б.	Відділ проектування	Ст. інженер
	9	Климов В.Ж.	Мебельний цех №1	Столяр
	10	Куберман В.Ж.	Бухгалтерія	Бухгалтер

Рис. 4.6. Компонент **DBGrid.**

Об'єкт **DBGrid** зв'язується із джерелом даних через властивість **DataSource**, що вказує на компонент джерела даних, який, в свою чергу, посилається на набір даних. У процесі виконання програми можна міняти джерела даних, використовуючи одну таблицю (**DBGrid**) для відображення даних від декількох джерел.

Для визначення складу стовпців можна використати редактор стовпців, реалізований діалоговим вікном **Column Editor** (рис. 4.7).

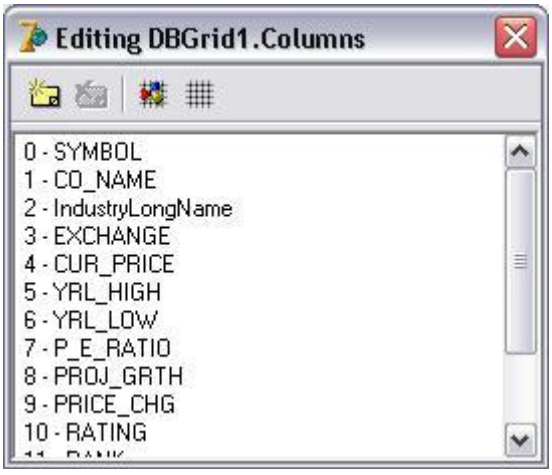


Рис. 4.7. Редактор стовпців.

Якщо стовпчики таблиці за допомогою редактора стовпців не створювалися, то за замовчуванням будуть створені поля, оголошені за допомогою редактора полів набору даних (рис. 4.5). При цьому всі властивості стовпців і порядок їхнього проходження успадковуються з редактора набору даних. Якщо проводиться динамічне формування об 'єктів **TField**, то порядок проходження полів і їх характеристики відповідають тим, що були задані при визначенні структури запису таблиці бази даних при створенні.

Редактор стовпців викликається подвійним клацанням на компоненті. У діалоговому вікні редактора можна встановити різні властивості об'єктів-стовпців, такі як колір фону, шрифт заголовка, основного поля таблиці й вибраного стовпця.

Звернутися до конкретного стовпця компонента **DBGrid** можна як до елемента індексованого масиву: **DBGrid1.Columns.Items[I].**

Для відображення й редагування даних часто використовуються також елементи **DBEdit**, які є модифікованими компонентами **Edit**. Зв'язок із джерелом даних здійснюється за допомогою властивості **DataSource**. Поле, з яким зв'язується даний компонент, вказується у властивості **DataField.**

Компонент **DBMemo** є багаторядковим редактором. Як правило, він зв'язується з **memo**-полями.

З його допомогою часто відображають великі текстові масиви.

Компонент **DBComboBox** – це список, зв’язаний з певним полем набору даних. Значення списку зберігаються у властивості **Items**. Елементи списку можна додавати, видаляти, зберігати й завантажувати за допомогою методів **Delete, Insert, LoadFromFile** та **SaveToFile**. Параметр **Index** вказує на позицію рядка у списку, а параметр **FileName** містить повний шлях до файлу, звідки завантажувалися значення списку. Даний компонент зручно використовувати при організації довідників користувача.

Компонент **DBImage** призначений для відображення графічної інформації, що зберігається в пов'язаному з ним **BLOB**-полі. Зв'язується цей об'єкт з набором даних через ті ж властивості, що й інші компоненти відображення даних. Зображення , що отримується з набору даних, зберігається у властивості **Picture.**

Компонент **DBRichEdit** є багаторядковим редактором з можливістю відображати форматований текст. Даний компонент підходить для відображення різних приміток або зберігання звітів.

Компонент **DBChart** служить для побудови графіків на основі набору даних. Іноді за допомогою кількох маніпуляцій мишею можна отримати гарні об'ємні графіки, що показують різні залежності.

Щоб побудувати графік, треба вибрати для даного компонента команду контекстного меню **Edit Chart.** У результаті буде активовано діалогове вікно **Editing DBChart.** Для створення нової серії даних необхідно натиснути кнопку **Add**. У результаті з’явиться наступне діалогове вікно **TeeChart Gallery,** у якому можна буде вибрати тип діаграми чи графіка (рис. 4.8). Перейшовши на вкладку **Series,** можна отримати доступ до налаштувань діаграми. На цій сторінці розташовано вкладки **Format, General, Mark** й **Data Source.** На вкладці **Format** можна задати різні параметри відображення графіка. Вкладка **General** дозволяє задати кілька властивостей загального характеру. На вкладці **Mark** можна задати підпис – легенду, розміщення пояснень до нього та інші параметри. Вкладка **Data Source** дозволяє задати джерело даних і визначити поля, по яких буде будуватися графік.

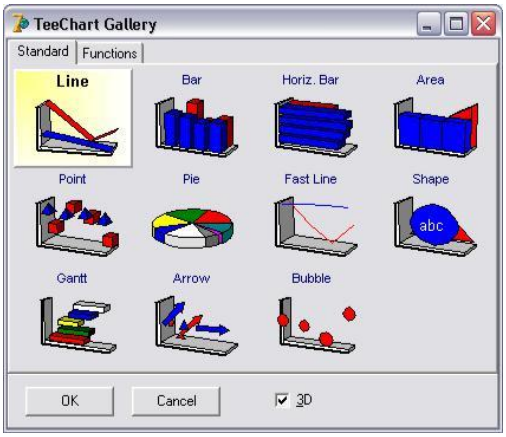


Рис. 4.8. Вибір виду графіка

З випадаючого списку можна вибрати тип джерела даних для графіка. У даному прикладі варто зупинитися на значенні **Dataset**. Далі зі списку **Dataset** потрібно вибрати конкретний набір даних. У списку **Labels** необхідно вибрати поле, значення якого будуть підписуватися під значеннями графіка. Поле, на підставі якого буде побудований графік, вибирається в списку **Pie** з доступних полів. Приклад отриманої діаграми показано на рис. 4.9.

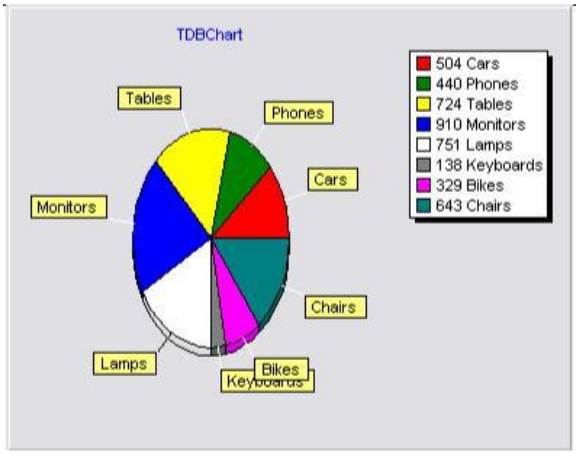


Рис. 4.9. Графік, побудований за допомогою компонента **DBChart**

3. Розробка прикладної програми для роботи з БД

Як приклад використання можливостей **Delphi** для роботи з **БД** розглянемо технологію створення простої прикладної програми. Цю програму можна розробити без написання коду, а виконати всі необхідні операції за допомогою утиліти **Database Desktop**, конструктора форм та інспектора об'єктів. Основні етапи створення прикладної програми такі:

- Створення таблиць БД
- Створення програми, що працюватиме з розробленою БД

В найпростішому випадку БД містить тільки одну таблицю. Якщо таблиці БД вже є, то створення програми починається відразу з другого етапу.

У даному прикладі розглянемо програму, яка працюватиме з табличною БД за технологією "файл-сервер".

3.1. Створення бази даних

Для роботи з таблицями **БД** при проектуванні програми зручно використовувати утиліту **Database Desktop**, яка дозволяє створювати таблиці, змінювати їх структуру та редагувати записи у них.

Процес створення нової таблиці починається командою **New/Table** (**Нова/Таблиця**) і відбувається в інтерактивному режимі, при цьому розробник повинен:

- Вибрати тип таблиці

- Задати структуру таблиці
- Вказати ключові поля
- Визначити індекси
- Визначити обмеження на значення полів
- Визначити пароль
- Задати посилальну цілісність (зв'язки) між таблицями

Обов'язковими є дві перші дії, перераховані в цьому списку . Частина дій, наприклад, задання ключових полів, застосовується тільки для таблиць певних типів, наприклад, **Paradox.**

Рис. 4.10. Вибір формату таблиці.

Спочатку у вікні **Create Table** (рис. 4.10) вибирається формат таблиці. За замовчуванням пропонується формат баз даних **Paradox** версії **7**. Для таблиць інших типів (наприклад, **dBase IV**) дії користувача при створенні таблиці принципово не відрізняються.

Після вибору типу таблиці з'являється вікно визначення її структури (рис. 4.11), в якому виконується решта дій, при цьому для таблиці потрібно задати хоча б одне поле.

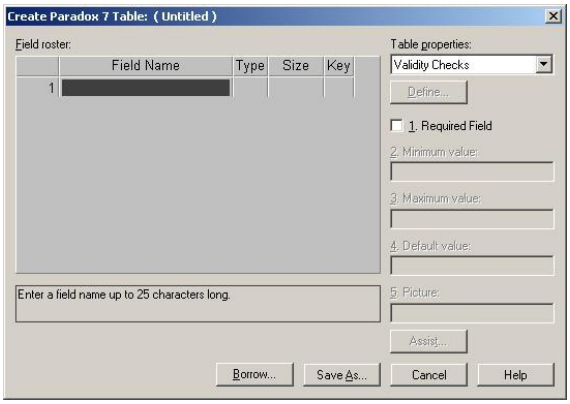


Рис . 4.11. Вікно для визначення структури таблиці.

Для кожного поля задається ім'я (в стовпці **Field Name**), тип (в стовпці **Type**) і при необхідності розмірність (стовпці **Size**). Можна задати тип поля, безпосередньо вказавши відповідний символ, або вибрати із списку (рис. 4.12), що розкривається після натиснення клавіші **<Пробіл>** або клацання правої кнопки миші на стовпці типів. Список містить всі типи полів, допустимі для заданого формату таблиці.

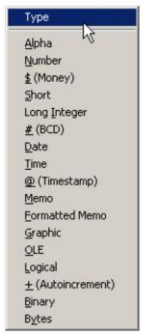
Для задання ключових полів в стовпці ключа **Key** потрібно ввести символ "*****" наступним чином: встановити курсор в цю позицію і натиснути будь-яку алфавітно-цифрову клавішу. При повторному натисненні клавіші знімається відмітка про приналежність поля ключу. Ключові поля повинні бути в списку полів першими, тобто розташовуватися у вікні визначення структури таблиці вгорі.

Для виконання додаткових дій щодо визначення структури таблиці використовується комбінований список **Table properties** (Властивості таблиці), який містить наступні пункти:

☑ **Secondary Indexes** – задання вторинних індексів
☑ **Validity Checks** – обмеження на введення значень полів
☑ **Password Security** – визначення пароля
☑ **Referential Integrety** – визначення посилальної цілісності між таблицями
☑ **Table Language** – задавання мови
☑ **Table Lookop** – задання полів перегляду

Після вибору пункту цього списку у вікні визначення структури таблиці з'являються відповідні елементи, за допомогою яких виконуються подальші дії. Як часто виконувану дію розглянемо задання індексу.

При виборі пункту **Secondary Indexes** комбінованого списку стає доступною кнопка **Define** (Визначити). Після її натиснення з'являється вікно **Define Secondary Index** (Задання вторинного індексу) (рис. 4.13).



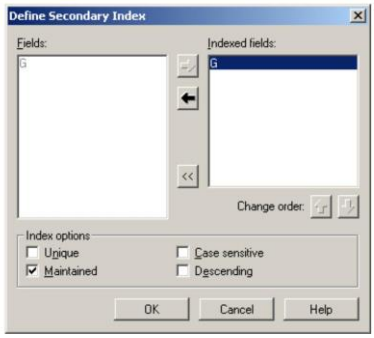


Рис. 4.12. Список типів полів Рис. 4.13. Задання вторинного індексу таблиці **Paradox 7**

В цьому вікні задаються індексні поля, що відображаються в правому списку, і параметри індексу, основні з яких змінюються за допомогою наступних прапорців:

☑ **Unique** – індекс допускає лише унікальні значення для складових полів

☑ **Case sensitive** – для полів рядкового типу враховується регістр символів

☑ **Descending** – сортування виконується в порядку спадання значень

Оскільки для таблиць **dBase** немає ключів, то для таких БД використовування прапорця **Unique** є єдиною можливістю забезпечити унікальність записів на фізичному рівні (рівні організації таблиці), не вдаючись до програмування.

Якщо для індексу таблиці **Paradox** встановлений прапорець **Descending**, то при спробі зробити цей індекс поточним (наприклад, через інспектор об'єктів або в процесі виконання програми) виникає помилка.

Після задання складу індексних полів і натиснення кнопки **OK** з'являється вікно **Save Index As**,

в якому слід вказати ім'я індексу. Для зручності звернення до індексу в його ім'я можна включити імена полів, вказавши при цьому якийсь префікс, наприклад **ind**. Небажано утворювати ім'я індексу тільки з імен полів, оскільки для таблиць **Paradox** подібна система іменування використовується при автоматичному утворенні імен для позначення посилальної цілісності між таблицями. Після натиснення кнопки **OK** сформований індекс додається до таблиці.

Аналогічним чином виконуються й інші дії, наприклад, визначення пароля, який перепитується при спробі відкриття таблиці.

Після визначення структури таблиці її необхідно зберегти, натиснувши кнопку **Save as...**

(Зберегти як) і вказавши розташування таблиці на диску і її ім'я. В результаті на диск записується нова таблиця, спочатку порожня, і автоматично створюються всі необхідні файли.

В подальшому структуру таблиці можна змінити, викликавши команду **Table/Restructure...**

(Таблиця/Змінити структуру), яка доступна тільки для відкритої таблиці. В результаті з'являється вікно визначення структури таблиці, далі виконуються дії, що аналогічні діям при створенні таблиці.

Під час зміни структури таблиці з нею не повинні працювати інші програми, у тому числі **Delphi.** Тому заздалегідь необхідно закрити **Delphi** або програму, в якій компоненти **Table** пов'язані з таблицею, що реструктурується. Інший варіант – деактивізувати компоненти **Table**, пов'язані з робочою таблицею. З цією метою властивості **Active** цих компонентів через інспектор об'єктів встановлюється значення **False.**

Перейменовування таблиці слід виконувати із середовища програми **Database Desktop**, а не з середовища **Windows** (наприклад, з допомогою провідника). При цьому стара таблиця також

зберігається. Інформація про назву таблиці використовується всередині файлів, тому просте перейменування всіх файлів таблиці призведе до помилки.

Якщо необхідно ознайомитися із структурою таблиці, то можна виконати команду **Table/Into Structure...** (Таблиця/Перегляд структури) утиліти **Database Desktop.** В результаті з'явиться вікно визначення структури таблиці, в якому заблоковано елементи, що дозволяють вносити в таблицю зміни. Переглядати можна навіть структуру тієї таблиці, з якою працюють інші програми.

3.2. Створення прикладної програми

Як приклад розглянемо форму програми, за допомогою якої можна переміщатися по записах таблиці БД, переглядати і редагувати поля записів, вставляти в таблицю нові записи, а також видаляти їх з таблиці. Вид форми головного вікна програми показаний на рис. 4.14. На формі розташовано наступні компоненти: **Table1**, **DataSource1**, **DBGrid1** і **DBNavigator1.**

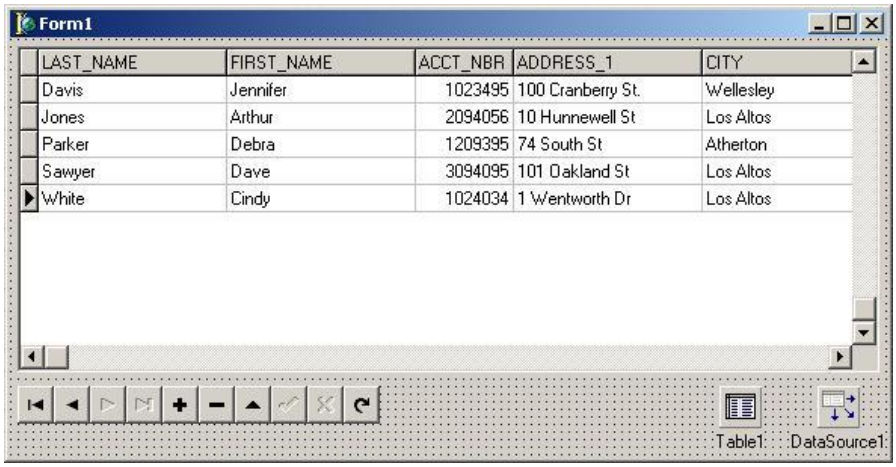


Рис. 4.14. Головна форма програми.

Компонент **Table1** забезпечує взаємодію з таблицею БД. Для зв'язку з необхідною таблицею необхідно встановити відповідні значення властивостей **DataBaseName**, яка вказує шлях до БД, і **TableName**, яка задає ім'я таблиці. Після задання таблиці БД властивості **Active** повинно бути встановлено значення **True.**

При зміні налаштувань таблиці (наприклад, перед зміною значень властивостей **DataBaseName** і **TableName**) потрібно встановити значення **False** властивості **Active.**

Ім'я таблиці краще вибирати з випадаючого списку в полі властивості **TableName** . Якщо шлях до БД (властивість **DataBaseName**) заданий правильно, в цьому списку відображаються всі доступні файли (таблиці).

В даній програмі використано таблицю з файлу **clients.dbf**, що поставляється як приклад разом з **Delphi.** Файл **clients.dbf** і файли інших таблиць при стандартній інсталяції середовища знаходяться в підкаталозі **DEMOS\DATA** головного каталога **Delphi**. Шлях до БД вказує псевдонім **dbdemos**, який слід вказати у властивості **DataBaseName.**

Компонент **DataSource1** є проміжною ланкою між компонентом **Table1**, який сполучений з реальною таблицею БД, і компонентами **DBGrid1** й **DBNavigator1**, за допомогою яких користувач взаємодіє з цією таблицею. На компонент **Table1**, що пов'язаний з компонентом **DataSource1**, вказує властивість **DataSet** останнього.

Таблиця №4.3. Значення властивостей компонентів

Компонент	Властивості	Значення
Table1	**Active** **DataBaseName** **TableName**	**true** **dbdemos** **clients.dbf**
DataSource1	**DataSet**	**Table1**
DBGrid1	**DataSource**	**DataSource1**
DBNavigator1	**DataSource**	**DataSource1**

Компонент **DBGrid1** відображає вміст таблиці БД у вигляді сітки, в якій стовпці відповідають полям, а рядки – записам. За замовчуванням користувач може переглядати і редагувати дані. Компонент **DBNavigator1** надає можливість переміщатися по таблиці, редагувати, вставляти і видаляти записи. Компоненти **DBGrid1** і **DBNavigator1** зв'язуються з джерелом даних – компонентом **DataSource1** через свої властивості **DataSource.**

При розробці програми значення всіх властивостей компонентів можна задати за допомогою інспектора об'єктів. При цьому необхідні значення можна набрати в полі значень або вибрати з випадаючих списків. В табл. №4.3 приведено компоненти, що використовуються для роботи з таблицею БД, а також основні властивості і їх значення.

4. Завдання на лабораторну роботу

Створити базу даних, яка містить одну таблицю формату **Paradox**. Таблиця повинна мати щонайменше сім полів з даними різного типу (в тому числі текстові дані, що міститимуть кирилицю), одне (або кілька) з яких повинно бути первинним індексом. Для таблиці створити вторинний індекс. Вид даних, які мають зберігатися у БД, слід вибрати самостійно згідно завдання за варіантом (табл. №4.4).

Розробити прикладну програму, яка працюватиме з вказаною базою і дозволятиме вводити нові записи, а також змінювати та витирати існуючі записи.

Таблиця №4.4.

Варіант	Вид бази даних
1	Дані про комплектуючі до комп’ютерів
2	Дані про бібліотечну літературу
3	Дані про абонентів бібліотеки
4	Дані про абонентів телефонної станції
5	Дані про географічні об’єкти області
6	Дані про студентів, що навчаються в університеті
7	Дані про працівників підприємства
8	Дані про структуру підприємства
9	Дані про автомобілі
10	Дані про фармацевтичну продукцію
11	Дані про меблеву продукцію
12	Дані про будівельні матеріали
13	Дані по складському обліку на підприємстві
14	Дані по прихідних та розхідних платіжних документах
15	Дані про відвідувачів WEB-сайту
16	Дані про кабельно-провідникову продукцію
17	Дані про радіоелектронну апаратуру
18	Дані про побутову техніку
19	Дані про продуктові товари
20	Дані про астрономічні об’єкти
21	Дані про напівпровідникові пристрої
22	Дані про рослинний світ країни
23	Дані про тваринний світ країни
24	Дані про підприємства області
25	Дані про історичні пам’ятки області
26	Дані про успішність студентів групи
27	Дані про CD та DVD- диски
28	Дані про продукцію легкої промисловості
29	Дані про міста світу
30	Дані про морських тварин та рослин

Лабораторна робота №10

Тема: Особливості роботи Delphi-програм з сервером InterBase.

1. Компоненти InterBase eXpress

Компоненти **InterBase eXpress (IBX)** призначені для роботи із сервером **InterBase** за допомогою **InterBase API** . Застосовуючи їх, можна читати дані, вносити в них зміни, керувати транзакціями, отримувати відомості про базу даних, відслідковувати стан виконання запитів і здійснювати інші дії. При цьому всі зазначені операції виконуватимуться помітно швидше, ніж при використанні аналогічних компонентів **BDE**. Основою коду **IBX** є бібліотека **FreeIBComponents**, розроблена Грегорі Ділтцом в 1998 році. Компанія **Borland** продовжила розробку цього набору компонентів, і вони присутні у всіх наступних версіях **Delphi** та **C++ Builder**.

1.1. Компонент TIBDataBase

Механізм доступу до даних **InterBase eXpress** використовує для звертань до сервера можливості клієнтської частини **InterBase/ FireBird** , встановленої на машині. Компонент **TIBDataBase** призначений для встановлення з'єднання з базою даних, розташованою на сервері **InterBase/ FireBird.** Для встановлення зв'язку із сервером необхідно вказати ім'я сервера і повний шлях до БД. Якщо сервер є локальним , то можна вказати шлях до файлу бази даних. Для цього слід присвоїти відповідне значення властивості **DatabaseName** або скористатися редактором, вікно якого показане на рис. 5.1. Для виклику редактора потрібно двічі клацнути на компоненті або натиснути на кнопку, розташовану біля правого краю властивості **DatabaseName** у інспекторі об'єктів.

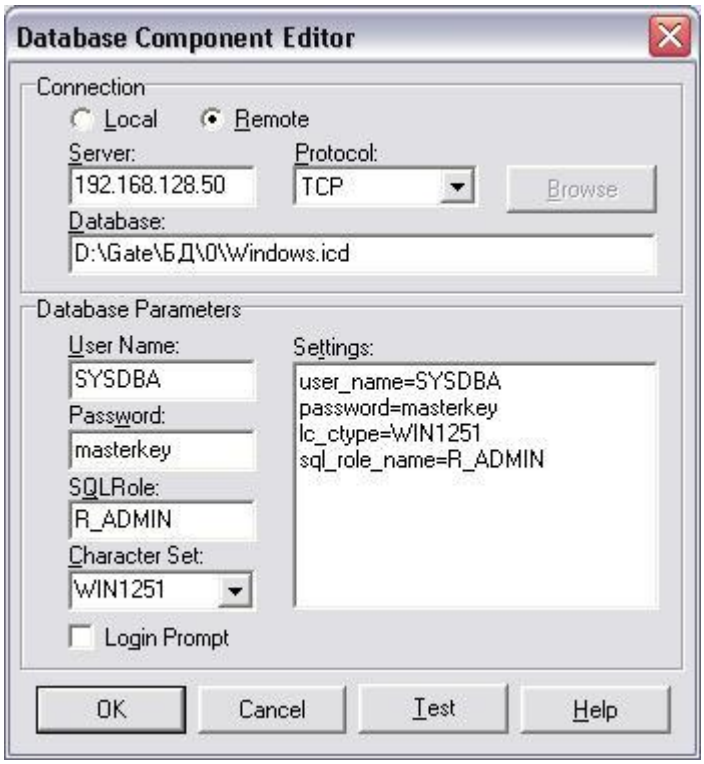


Рис. 5.1. Редактор з'єднання з БД компонента **IBDataBase.**

На панелі **Connection** вибирається тип з'єднання (локальне чи віддалене), потім вказується адреса сервера (або його мережеву назву) та протокол зв'язку. У поле **Database** прописується шлях до файлу бази даних. Можна також встановити з'єднання з локальним сервером як з віддаленим за допомогою протоколу **TCP/IP** (або іншого, але він повинен бути встановлений у системі). Для цього

в поле **Server** треба ввести адресу **127.0.0.1** (ця адреса зарезервована для локальної машини) або мережеву назву свого комп'ютера, і вибрати зі списку **Protocol** значення **TCP**. У поле **Database** необхідно вказати шлях до бази даних. Активувати з'єднання можна за допомогою властивості **Connected.** Властивість **AllowStreamedConnected** дозволяє заборонити з'єднання з базою даних під час запуску програми, навіть якщо на етапі розробки властивості **Connected** було присвоєне значення **True.** За замовчуванням властивість приймає значення **True,** і з'єднання з базою даних може здійснюватися автоматично під час запуску. В іншому випадку для встановлення зв'язку з БД у програмі необхідно присвоїти властивості **Connected** значення **True** або викликати метод **Open.**

За допомогою властивості **Params** можна задати параметри з'єднання, такі як ім'я користувача, пароль, роль і кодування символів для рядкових даних. Властивість **DBParamByDPB** дозволяє звернутися до окремих параметрів з'єднання за їх індексом. Наприклад, для одержання імені користувача можна використати конструкцію **DBParamByDPB[isc_dpb_user_name].** Визначити діалект бази даних можна за допомогою властивості **DBSQLDialect.** Всі дії з базою проводяться в межах транзакцій. Властивість **DefaultTransaction** містить посилання на компонент транзакції **IBTransaction,** що використовується за замовчуванням. Всі компоненти, пов'язані з **IBDataBase,** також будуть використовувати даний компонент транзакції. З одним компонентом **IBDataBase** може бути з'єднано кілька компонентів транзакцій. З'ясувати, скільки насправді їх використовується, можна за допомогою властивості **TransactionCount**, а звернутися до компонента транзакції можна за його індексом, використовуючи властивість **Transactions.**

Помістити компонент транзакції в список використовуваних можна за допомогою методу

AddTransaction, а для видалення слід використати метод **RemoveTransaction.** Метод **RemoveTransactions**

застосовується для видалення всіх зв'язаних транзакцій.

Для отримання індексу транзакції використовується метод **FindTransaction** , що отримує як параметр вказівник на компонент транзакції. З компонентом може бути зв'язано кілька компонентів **TIBEvents**, що автоматично відслідковують події, які виникають у базі даних. Для додавання такого об'єкта використовується метод **AddEventNotifier**. В свою чергу, для видалення об'єкта зі списку використовується метод **RemoveEventNotifier**. Ціла властивість **IdleTimer** дозволяє визначити часовий інтервал, після закінчення якого неактивне з'єднання буде розірване. В момент закінчення часу очікування ініціюється подія **OnIdleTimer.**

Група методів **CheckActive, CheckInactive** й **CheckDatabaseName** дозволяє з'ясувати, чи активна база даних і чи має властивість **DatabaseName** значення. Якщо перевірка виявляється невдалою, викликається виняткова ситуація. Під час з'єднання з базою даних сервер запитує назву користувача і пароль. Це ініціює подію **OnLogin.**

Для створення бази даних можна використати метод **CreateDatabase.** Ім'я файлу бази даних задається у властивості **DatabaseName**, а інші параметри – у властивості **Params**. Ось приклад використання цього методу:

```
      procedure TForml.CreateDBBtnClick(Sender:
TObject); begin
       with IBDatabase2 do begin
        Params.Clear;
        DatabaseName:='D:\MyIntDBName.gdb';
        SQLDialect:=3:
        Params.Add('USER "SYSDBA"');
        Params.Add('PASSWORD "masterkey"');
        Params.Add('PAGE_SIZE 8192');
        CreateDatabase;
       end;
      end;
```

1.2. Компонент TIBTransaction

Транзакцією називають логічно пов'язаний з базою даних блок операцій, що виконується як єдине ціле або не виконується зовсім. Компонент **TIBTransaction** надає властивості й методи, призначені для керування транзакціями однієї або кількох баз даних. Список компонентів **TIBDatabase**, з якими зв'язаний даний компонент транзакції **TIBTransaction**, міститься у властивості **Databases**. Кількість поєднаних з компонентом баз даних можна з'ясувати за допомогою властивості **DatabaseCount**.

Властивість **DefaultDatabase** дозволяє вказати компонент бази даних **TIBDatabase**, що буде застосовуватися в межах даної транзакції. Для того щоб додати новий компонент БД під час виконання програми, використовується метод **AddDatabase**. Для розриву зв'язку використовується метод **RemoveDatabase**. Метод **CheckDatabasesInList** дозволяє з'ясувати, чи існують зв'язані з даним компонентом транзакції компоненти **TIBDatabase**. Якщо вони існують, метод повертає значення **True**.

Для запуску нової транзакції на сервері використовується метод **StartTransaction**. До виклику цього методу програма повинна зробити перевірку на виконання іншої транзакції (пов'язаної з даним компонентом) в поточний момент. Для цього використовується властивість **InTransaction**. Якщо властивість має значення **True**, (воно автоматично встановлюється при старті транзакції), то транзакція вже виконується. Спроба запуску транзакції методом **StartTransaction** без попереднього її завершення за допомогою методів **Commit** або **Rollback** приведе до виникнення винятку.

Для завершення активної транзакції й збереження змін, зроблених у її межах, використовується метод **Commit**. Для завершення транзакції і відміни всіх змін, зроблених у базі даних у рамках даної транзакції, використовується метод **Rollback**.

Приклад запуску і завершення транзакції:

```
procedure TForml.ApplyButtonClick(Sender:
TObject); begin
 IBDatabasel.Open;
 IBTransactionl.StartTransaction;
 Tablel. Insert;
 Tablel.FieldByName('CNT').AsInteger := StrToInt(Editl.Text);
 Tablel.Post;
 IBTransactionl.Commit;
end;
```

Кожен запит до БД виконується в контексті транзакції. Властивість **AllowAutoStart** вказує компоненту на необхідність автоматичного запуску транзакції, коли це необхідно. А властивість **AutoStopAction** визначає, яке дія буде виконана в момент автоматичного завершення транзакції. Ця властивість може приймати одне з наступних значень:

- Значення **saNone** вказує, що транзакція не може завершитися неявно.
- Значення **saRollback** вказує, що транзакція відкочується й закривається при неявному завершенні.
- Значення **saCommit** вказує, що транзакція підтверджується й закривається при неявному завершенні.
- Значення **saRollbackRetaining** вказує, що транзакція не завершується при закритті останнього зв'язаного набору даних, але всі зроблені зміни відкочуються.
- Значення **saCommitRetaining** вказує, що фіксація змін відбудеться, але транзакція не буде завершена.

Властивість **IdleTimer** визначає проміжок часу, після закінчення якого транзакція буде автоматично зафіксована або відкочена. Для визначення того, яку дію слід почати після закінчення зазначеного часу, використовується властивість **DefaultAction**. Вона може приймати одне із значень:

- Значення **TARollback** вказує, що буде зроблено відкат транзакції.

- Значення **TACommit** задає, що транзакція буде зафіксована.
- Значення **TARollbackRetaining** вказує, що буде зроблений відкат транзакції, але вона не буде завершена.
- Значення **TACommitRetaining** вказує, що буде зроблена фіксація транзакції, але вона не буде завершена.

Транзакція може мати набір параметрів. До цих параметрів відносяться: рівень ізоляції транзакції, можливість перегляду або модифікації таблиць, можливість одночасного доступу до таблиці (сторінки даних) з іншими транзакціями. Для зміни цих параметрів використовується властивість **Params**. Компонент транзакції має редактор **Transaction Editor**, що містить найчастіше застосовувані параметри. Для доступу до редактора слід двічі клацнути мишею на компоненті **TIBTransaction**. В результаті буде відображене діалогове вікно, показане на рис. 5.2. Параметри транзакцій детально будуть розглянуті під час наступної лабораторної роботи.

Рис. 5.2. Редактор параметрів транзакцій.

1.3. Компонент TIBDataSet

Компонент **TIBDataSet** призначений для формування в програмах наборів даних, отриманих у ході виконання **SQL**-запитів. Оскільки клас **TIBDataSet** є предком класу **TDataSet**, він може працювати з компонентами відображення даних. Компонент має ряд об'єктів, призначених для виконання запитів на вибірку, видалення, модифікацію й додавання записів. Основний запит на вибірку міститься у властивості **SelectSQL.**

Для створення запиту можна використати вмонтований редактор, що істотно спрощує цей процес. Щоб запустити редактор, досить вибрати команду контекстного меню **Edit SQL**. В результаті буде відображене вікно, показане на рис. 5.3.

Допоміжні запити містяться у властивостях **InsertSQL, ModifySQL , DeleteSQL** і **RefreshSQL**. Ці запити призначені, відповідно, для вставки, зміни, видалення та оновлення записів. Дані властивості надають прямий доступ до пов'язаних з ними об'єктів **SQL**. Кожному запиту відповідає власний об'єкт **IBSQL** , призначений для його виконання. Вказівники на ці об'єкти **SQL** містяться у властивостях **QSelect, QInsert, QModify, QDelete** і **QRefresh.**

Для виклику редактора допоміжних запитів слід вибрати пункт контекстного меню **DataSet Editor.** Зовнішній вигляд вікна редактора запитів показані на рис. 5.4. Формування запитів відбувається дуже просто. Потрібно вибрати зі списку **Key Fields** ключові поля, за якими буде проводитися пошук, і вибрати редаговані поля зі списку **Update Fields**. Після цього для автоматичного складання запитів слід натиснути кнопку **Generate SQL**. Текст сформованих запитів можна подивитися на вкладці **SQL.**

Сервер **InterBase**, як і будь-який інший сервер БД, має аналог автоінкрементного поля. Таке поле реалізується за допомогою генератора. Як правило, значення генератора збільшується спеціально написаним тригером. Для того щоб задати генератор для даної таблиці, можна скористатися властивістю **GeneratorField** , що забезпечує доступ до об'єкта **TIBGeneratorField,**

використовуючи який, можна вказати поле, значення якого буде збільшуватися, і метод обчислення нового значення поля.

У властивості **Field** вказується ім'я поля, для якого буде генеруватися значення. А використовуючи властивість **IncrementBy**, можна вказати значення, на яке буде збільшено лічильник. Як правило, ця властивість має одиничне значення. У властивості **Generator** вказується ім'я генератора, що викликається об'єктом **TIBGeneratorField** при генеруванні нового значення.

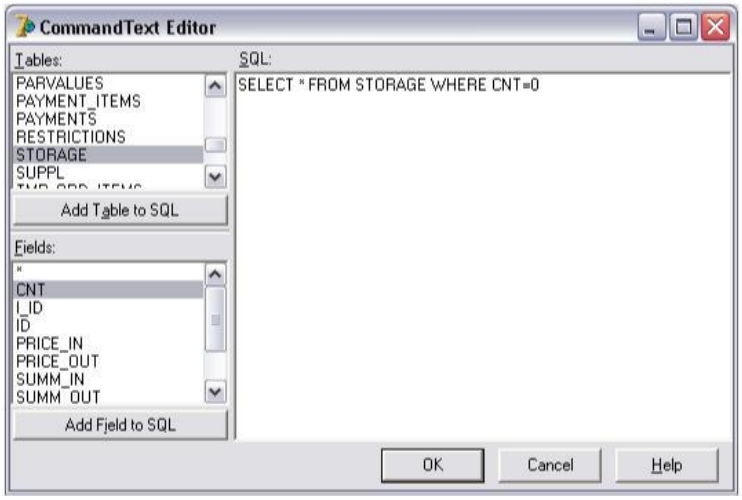


Рис. 5.3. Редактор запитів **SQL**.

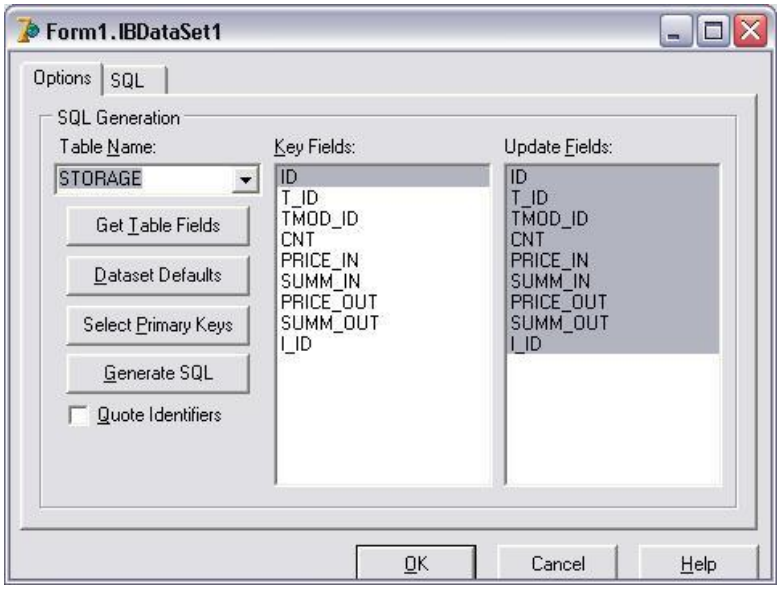


Рис. 5.4. Редактор допоміжних запитів **SQL**.

За допомогою властивості **ApplyEvent** можна вказати, коли серверу буде передаватися запит на генерування нового значення поля. Вона може приймати одне з наступних значень:

- Значення **gamOnNewRecord** вказує, що набір даних викликає метод генерування нового значення поля відразу після вставки запису, але до ініціювання події **OnNewRecord**.
- Значення **gamOnPost** вказує, що набір даних викликає метод сервера для генерування нового значення поля запису після виклику метода **Post**, але до виклику методу **BeforePost**.
- Значення **gamOnServer** вказує, що набір даних не робить виклик методу для генерації нового значення поля запису, а сервер генерує значення автоматично за допомогою відповідного тригера.

Для виклику генератора на сервері і генерування нового значення поля можна викликати метод **Apply**, що поміщає згенероване значення в інкрементоване поле.

Найзручніше визначати параметри об'єкта **TIBGeneratorField** за допомогою спеціального редактора, доступ до якого можна отримати за допомогою кнопки, розташованої в лівій частині властивості **GeneratorField**. Вікно редактора показане на рис. 5.5.

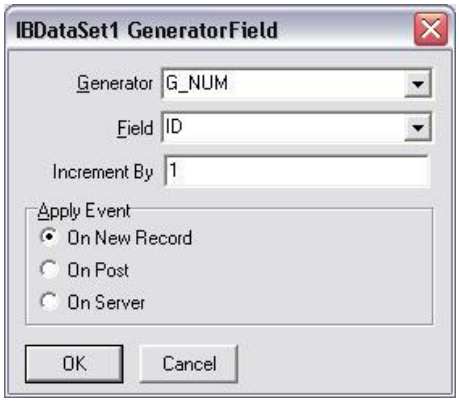


Рис. 5.5. Вікно редактора **GeneratorField.**

У полі **Generator** слід вибрати генератор зі списку, у полі **Field** – поле, для якого буде генеруватися значення, а в полі **Increment By** можна вказати приріст, на який буде збільшуватися поточне значення генератора при створенні нового запису. На панелі **Apply Event** вказується момент, коли нове значення буде генеруватися й відсилатися на сервер.

Повернемося до розгляду компонента **TIBDataSet**. Визначити, чи підготовлений **SQL**- запит до виконання, можна за допомогою властивості **Prepared**. Якщо властивість має значення **True**, то запит підготовлений. Як правило, використання даної властивості рекомендується у випадку використання однотипних параметризованих запитів, у яких міняється не структура запиту, а тільки його параметри. Це пов'язане з тим, що сервер розбирає й аналізує запит перед виконанням тільки один раз. Всі наступні виклики даного запиту обробляються прямо, використовуючи результати попереднього аналізу . Для підготовки запиту до виконання потрібно викликати метод **Prepare**, а для його повернення до вихідного стану – метод **UnPrepare.**

Для виконання **SQL** -запиту, що не повертає набір даних, застосовується метод **ExecSQL**, а для запиту на читання даних – метод **Open.**

1.4. Методика роботи з сервером

Розглянемо створення тестової програми, що ілюструє методику роботи з сервером **InterBase/ Firebird** за допомогою компонентів **IBX**. До нового проекту потрібно додати модуль даних і помістити в нього компоненти **TIBDatabase, TIBTransaction, TIBDataSet** і **TDataSource.**

Для початку необхідно налаштувати компонент **TIBDatabase**. Після подвійного клацання мишею на компоненті з'явиться вікно **Database Component Editor**. Для прикладу , під'єднаємося до сервера через протокол **TCP/IP**, імітуючи роботу через мережу. Для цього слід задати віддалений сервер і вказати **IP**-адресу **127.0.0.1** (вона використовується для адресації локальної машини). Замість адреси можна вказати мережеву назву комп'ютера (дійсну чи **localhost** – ця назва використовується для позначення локального комп'ютера). Також потрібно вибрати протокол **TCP**, а в полі **Database** вказати шлях до бази даних (в ролі бази можна використати тестову БД, яка поставляється разом з **Delphi** – **employee.gdb**. Вона за замовчуванням інсталюється у папку **X:\Program Files\Common Files\Borland Shared\Data**). У полях **User Name** і **Password** слід вказати значення імені користувача **InterBase/ FireBird** і його пароль (можна вказати **SYSDBA** та **masterkey** відповідно). Після натискання кнопки **OK** властивості **LoginPrompt** краще присвоїти значення **False** (тоді при з'єднанні з БД назва користувача і пароль не будуть перепитуватися).

Після цього слід встановити з'єднання з базою даних, задавши для властивості **Connected** значення **True**. Компонент бази даних **TIBDatabase** за допомогою властивості **DefaultTransaction** слід зв'язати з компонентом транзакції **IBTransaction**. Рівень ізоляції транзакції встановлюється за допомогою значення **Read Commited** у редакторі **Transaction Editor** компонента **IBTransaction.**

Активується компонент за допомогою присвоювання властивості **Active** значення **True.**

Компонент **IBDataSet** з'єднується з компонентами бази даних і транзакції за допомогою властивостей **Database** і **Transaction**. У властивості **SQL** треба вказати запит **select * from COUNTRY**, що отримує всі записи з таблиці **COUNTRY.**

Тепер потрібно задати допоміжні запити. Для цього слід запустити редактор **DataSet Editor**, вибрати в списку **Key Fields** ключові поля, зі списку **Update Fields** – поля для змін і згенерувати запити, натиснувши кнопку **Generate SQL**. Потрібно також включити механізм кешування змін, задавши властивості **CachedUpdates** значення **True**. Після цього слід активувати компонент, присвоївши властивості **Active** значення **True.**

Компонент **TDataSource** за допомогою властивості **DataSet** потрібно зв'язати з компонентом **TIBDataSet.** Модуль даних необхідно оголосити в секції **uses** модуля головної форми програми. На формі слід розмістити компоненти **DBGrid**, два компоненти **DBEdit** і чотири кнопки. Компоненти **DBGrid** і **DBEdit** слід зв'язати з компонентом **DataSource.** На рис. 5.6 показане вікно програми.

Рис. 5.6. Головне вікно програми.

Кнопка "Добавити" називається **btnAdd**, кнопка "Прийняти" – **btnPost**, кнопка "Витерти" – **btnDelete**, кнопка "Зберегти" – **btnSave.** Код оброблювачів натискання на відповідні кнопки наступний:

```
      procedure TForm1.btnAddClick(Sender: TObject);
      begin
       IBDataSet1.Insert;
      end;

      procedure TForm1.btnPostClick(Sender: TObject);
      begin
       if IBDataSet1.Modified then IBDataSet1.Post;
      end;

      procedure TForm1.btnDeleteClick(Sender:
TObject); begin
       IBDataSet1.Delete;
      end;

      procedure TForm1.btnSaveClick(Sender: TObject);
      b
 egin
      t
   ry
```

```
    IBDataSet1.ApplyUpdates;
    IBTransaction1.CommitRetaining;
   except
    IBTransaction1.Rollback ;
   end;
 end;
```

Зверніть увагу на те, що в процедурі, яка зберігає зміни даних, після виклику методу **ApplyUpdates** слід викликати метод **CommitRetaining** або **Commit.** Це необхідно робити через те, що метод **ApplyUpdates** не викликає метод **CommitRetaining** автоматично.

1.5. Компонент TIBSQL

Компонент **TIBSQL** призначений для виконання **SQL**-запитів з мінімальними втратами часу. Компонент не підтримує роботу з механізмами відображення даних і надає розроблювачу тільки односпрямований курсор. Безпосереднім предком компонента є клас **TComponent**, тому він тільки передає запит на сервер через компонент бази даних **TIBDatabase** і повертає результат виконання запиту.

Для зв'язку з базою даних необхідно вибрати відповідний компонент у властивості **Database.**

За допомогою властивості **Transaction** слід також задати робочий компонент транзакції.

Компонент у своїй роботі не використовує об'єкти полів **TField**, але повертає дані полів в об'єкті **TIBXSQLDA**, що містить об'єкти **TIBXSQLVAR**. Кожен об 'єкт **TIBXSQLVAR** містить у собі значення якогось поля набору даних. Для звертання до поля за його індексом використовується властивість **Fields**. А для визначення індексу поля по його імені можна використати властивість **FieldIndex**. Щоб отримати вказівник на потрібний об'єкт **TIBXSQLVAR** через його назву, можна скористатися методом **FieldByName.**

Визначити кількість записів, що містяться в наборі даних, можна за допомогою цілої властивості **RecordCount**. Для переміщення до наступного запису використовується метод **Next.** При досягненні курсором початку набору даних властивість **Bof** приймає значення **True**. Аналогічно, при досягненні кінця набору даних значення **True** приймає властивість **Eof**. Щоб набір даних у момент відкриття встановлював курсор на перший запис, слід встановити властивості **GoToFirstRecordOnExecute** значення **True.** При використанні параметризованих запитів можна змінювати їхні параметри під час виконання програми. Для того щоб компонент міг автоматично оновлювати список параметрів, необхідно присвоїти властивості **ParamCheck** значення **True**. Якщо використання параметризованих запитів не передбачається, краще присвоїти властивості значення **False.**

Для формування списку імен параметрів запиту використовується властивість **GenerateParamNames.** Для формування списку досить присвоїти йому значення **True.**

У властивості **SQL** вказується текст запиту. Запит можна створювати в спеціальному редакторі, (для його виклику потрібно натиснути на кнопку в інспекторі об’єктів, розташовану в правій частині властивості). Зробити перевірку коректності запиту допоможе метод **CheckValidStatement.** Якщо запит не є коректним, метод згенерує виняткову ситуацію. Визначити, чи підготовлений запит до виконання, можна за допомогою властивості **Prepared**. А для підготовки запиту викликається метод **Prepare.**

Після того як запит був підготовлений, можна подивитися план його виконання, що зберігається у властивості **Plan**. Для виконання запиту використовується метод **ExecQuery**. За допомогою властивості **SQLType** можна з'ясувати тип виконаного запиту. Властивість може повертати кілька значень:

- Значення **SQLCommit** вказує, що було зроблене підтвердження активної транзакції.
- Значення **SQLDelete** означає, що був вилучений запис.
- Значення **SQLPutSegment** вказує, що був зроблений запис сегмента даних типу **BLOB.**

- Значення **SQLRollback** сигналізує про те, що був зроблений відкат транзакції.
- Значення **SQLSetForUpdate** вказує, що була виконана збережена процедура, що змінила набір даних.
- Значення **SQLSetGenerator** означає, що було здійснено збільшення значення генератора автоінкрементного поля.
- Значення **SQLSelect** вказує, що був виконаний запит, результатом якого є набор даних.
- Значення **SQLStartTransaction** сигналізує про запуск нової транзакції.
- Значення **SQLUnknown** застосовується для позначення невідомого типу запиту.
- Значення **SQLUpdate** вказує, що був виконаний запит для оновлення даних.

У властивості **RowsAffected** вказується кількість записів, оброблених останнім запитом. Кожному виконуваному запиту присвоюється унікальний внутрішній ідентифікатор, що зберігається у властивості **UniqueRelationName.**

Для визначення активності набору даних можна скористатися властивістю **Open**, що містить значення логічного типу. Метод **CheckOpen** згенерує виняток, якщо набір даних неактивний. Щоб закрити набір даних, слід використати метод **Close**. Метод **CheckClosed** створить виняткову ситуацію, якщо набір даних відкритий. Метод **Call** повертає повідомлення про помилку на основі її коду.

Розглянемо принцип використання компонента на прикладі. Як було відзначено раніше, компонент **TIBSQL** не може передавати дані в стандартні компоненти відображення даних, тому отримані при запиті дані треба виводити в інший табличний компонент, наприклад **TStringGrid.**

Після створення нового проекту до нього слід додати модуль даних. У ньому потрібно розмістити компоненти **TIBDataBase, TIBTransaction** і **TIBSQL**. За допомогою компонента **TIBDataBase** налаштовується з'єднання з базою **employee.gdb**. Після базового встановлення параметрів компонента транзакції **TIBTransaction** потрібно вказати рівень ізоляції транзакції **Snapshot.** Компонент **TIBSQL** зв'язується з базою даних за допомогою властивості **Database**, а з компонентом транзакції – за допомогою **Transaction**. У властивості **SQL** потрібно вказати запит **select * from EMPLOYEE**, що дозволяє отримати всі записи з таблиці **EMPLOYEE.**

На головній формі програми потрібно розташувати компонент **TStringGrid** і одну кнопку (її назва **btnExecSQL**). Метод, що викликається після натискання кнопки, буде заповнювати таблицю значеннями й формувати заголовок. Вікно програми показане на рис. 5.7. Код, що отримує дані з БД та виводить їх у вікно, наступний:

```
procedure TForm1.btnExecSQLClick(Sender: TObject);
var I, J: integer;
    FieldCount : integer;
begin
 FieldCount:=11;
 with IBSQL1 do begin
  ExecQuery;
  StringGrid1.ColCount:=FieldCount;
  for I:=0 to FieldCount-1 do
StringGrid1.Cells[I,0]:=Fields[I].Name; J:=0;
  while not Eof do begin
   Inc(J);
   StringGrid1.RowCount:=StringGrid1.RowCount+1;
   for I:=0 to FieldCount-1 do begin
    StringGrid1.Cells[I,J]:=Fields[I].AsString;
   end;
   Next;
  end;
 end;
end;
```

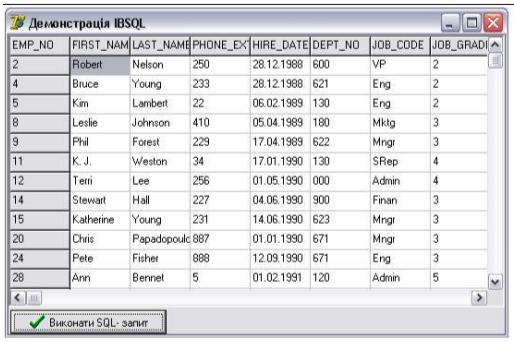


Рис. 5.7. Використання компонента **TIBSQL.**

1.6. Компонент TIBTable

Компонент **TIBTable**, що є прямим нащадком класу **TIBCustomDataSet**, інкапсулює функціональність таблиці бази даних сервера **InterBase**. Компонент може використовуватися для отримання прямого доступу до даних таблиці або представлення. Також компонент має можливість працювати з множинами записів усередині таблиці бази даних, використовуючи фільтри.

Після з'єднання з базою даних у властивості **TableNames** вказується список доступних таблиць. Використовуючи властивість **TableTypes**, можна визначити, чи будуть доступні для перегляду системні таблиці та представлення. Властивість має вигляд множини, в яку можуть входити такі елементи:

ttSystem – вказує, що для перегляду будуть доступні системні таблиці й подання.

ttView – вказує, що для перегляду доступні також і представлення.

Властивість **Filtered** дозволяє керувати режимом фільтрації записів. Якщо властивість має значення **True**, то фільтрація ввімкнена. Критерій, по якому проводиться фільтрація записів, вказується в текстовій властивості **Filter**. Ось приклад використання цієї властивості:

```
IBTabie1.Filtered:=False;
IBTable1.Filter:='Country = Ukraine';
IBTable1.Filtered:=True;
```

Для задання параметрів фільтрації використовується властивість **FilterOptions.**

У момент відкриття набору даних проводиться спроба впорядкування записів за первинним ключем. Впорядкування проводиться, якщо властивість **DefaultIndex** має значення **True**. Вручну вибрати індекс можна за допомогою властивості **IndexName**. Поля індексу задаються у довільному порядку шляхом перелічування їх у властивості **IndexFieldNames.**

Визначити, чи існує в базі даних таблиця, ім'я якої зазначене у властивості **TableNames**, можна за допомогою властивості **Exists**, що поверне значення **True** у випадку успішного пошуку. Часто ця властивість використовується, коли необхідно створити таблицю бази даних. Тоді спочатку можна провести перевірку на існування таблиці, і якщо таблиця не існує, вона створюється методом

CreateTable.

Для видалення таблиці можна використати метод **DeleteTable**, а для її очищення – метод

EmptyTable.

У деяких випадках виникає необхідність синхронізації положення курсорів різних наборів даних. Для цього використовується метод **GoToCurrent**, якому передається вказівник на набір даних, курсор якого буде синхронізований з положенням поточного курсору.

1.7. Компонент TIBQuery

Компонент **TIBQuery** призначений для виконання **SQL**-запитів, що повертають набори даних. Безпосереднім предком компонента є клас **TIBCustomDataSet**. Компонент **TIBQuery** надає розроблювачеві нередагований набір даних. Для зміни цих даних необхідно використати, наприклад, компонент **TIBUpdateSQL.**

Текст **SQL**-запиту міститься у властивості **SQL**, з якою зв'язаний редактор запитів. Якщо необхідно розглянути код запиту, можна використати властивість **Text**. Параметри **SQL**-запиту містяться у властивості **Params**. Загальна кількість параметрів запиту зберігається у властивості **ParamCount.**

Щоб виконати запит, що міститься у властивості **SQL**, можна присвоїти властивості **Active** значення **True** або викликати метод **Open** . Для виконання запиту, що не повертає даних, потрібно використати метод **ExecSQL**. Ось приклад використання компонента **IBQuery**:

```
IBQuery1.Active:=false;
IBQuery1.SQL.Clear;
IBQuery1.SQL.Add('SELECT * FROM PEOPLES WHERE TabNum=:TABNUM');
IBQuery1.ParamByName('TABNUM').AsInteger:=StrToInt(eTabNum.Text);
IBQuery1.Active:=true;
```

1.8. Компонент TIBUpdateSQL

Компонент **TIBUpdateSQL** реалізує об 'єкти, призначені для редагування наборів даних, доступних тільки для читання. Компонент, як правило, використовується разом з компонентом **TIBQuery**, коли в нього ввімкнено механізм кешування даних. Вся робота механізму є прозорою для користувача й практично не вимагає написання коду.

Зв'язок компонента з набором даних (**TIBQuery**) здійснюється за допомогою властивості **UpdateObject** компонента, що представляє набір. Запити зручно створювати в спеціальному редакторі, доступ до якого можна отримати за допомогою команди контекстного меню **UpdateSQL Editor.**

1.9. Компонент TIBStroredProc

Компонент **TIBStoredProc** призначений для виконання збережених процедур, що зберігаються на сервері, і повернення результатів їхньої роботи. При з'єднанні з базою даних у властивість **StoredProcedureNames** завантажується список збережених процедур бази даних. Збережена процедура, яку необхідно виконати, вказується у властивості **StoredProcName.**

Як правило, збережені процедури мають набори вхідних і вихідних параметрів, що містяться у властивості **Params**. Звернутися до параметра можна за його індексом або назвою, використовуючи метод **ParamByName.**

Кількість параметрів, що має дана збережена процедура, зберігається у властивості **ParamCount.** Використовуючи метод **CopyParams**, можна скопіювати параметри збереженої процедури в інший список, що вказується в параметрі **Value.**

Визначити чи готова збережена процедура до виконання дозволяє властивість **Prepared**. Якщо властивість має значення **True**, то процедура підготовлена до виконання. Для підготовки збереженої процедури до виконання використовується метод **Prepare**. Якщо при виклику процедури вона ще не підготовлена, то викликається метод **Prepare**. Після виконання збереженої процедури автоматично викликається метод **UnPrepare** для повернення її у вихідний стан.

Для виконання збереженої процедури на сервері потрібно викликати метод **ExecProc**:

```
StoredProcedure1.ParamByName('CNT').AsFloat
:=12.7; StoredProcedure1.ExecProc;
```

У цьому прикладі процедура має один вхідний параметр **CNT** дійсного типу, якому перед викликом присвоюється значення **12.7.**

1.10. Компонент TIBDataBaselnfo

Компонент **TIBDatabaseInfo** призначений для отримання різної системної інформації про базу даних. Властивість **DBFileName** повертає ім'я файлу бази даних. Розмір сторінки БД повертає властивість **PageSize**, а число виділених базі сторінок можна з'ясувати за допомогою властивості **Allocation.** Діалект бази даних вказується у властивості **DBSQLDialect.**

Версію БД повертає властивість **BaseLevel** . У ньому зберігається число із двох розрядів. У першому розряді міститься номер бази даних, а в другому – її версія. Інформацію про версію можна також прочитати через властивість **Version.**

Залежно від версії дискової структури БД (**ODS** – **On Disk Structure**) розрізняється формат зберігання даних і міняються можливості роботи з базою. Кожна версія сервера має свою структуру. У процесі роботи з БД значення версії **ODS** може змінюватися. Властивість **ODSMinorVersion** повертає початкове значення версії **ODS.** При зміні структури БД збільшується значення версії. Властивість **ODSMajorVersion** повертає останню версію структури **ODS.**

Властивість **DBImplementationNo** повертає номер опису бази даних, а **DBImplementationClass** повертає номер класу опису. За допомогою властивості **CurrentMemory** можна з'ясувати обсяг оперативної пам'яті, яку займає сервер на момент запиту. Використовуючи властивість **MaxMemory,** можна з'ясувати максимальне значення завантаження оперативної пам'яті сервером з моменту першого підключеного процесу. Властивість **NumBuffers** повертає кількість буферів пам'яті, виділених серверу на даний момент.

Визначити, доступна база даних для запису, чи вона надає можливість тільки читати дані, дозволяє властивість логічного типу **ReadOnly.**

Властивість **DeleteCount** повертає кількість записів, видалених з моменту останнього під'єднання до БД. А властивість **PurgeCount** повертає кількість повністю видалених записів, тобто записів, робота з якими в рамках транзакцій завершена. Довідатися число видалених версій записів можна за допомогою властивості **BackoutCount**. Кількість записів, доданих у базу даних з моменту останнього запуску сервера, повертає властивість **InsertCount**. А властивість **Writes** дозволяє визначити кількість операцій запису сторінок.

Властивість **SweepInterval** повертає кількість транзакцій, які можуть бути підтверджені в інтервалі між чищеннями. Властивість **UserNames** повертає список користувачів, з'єднаних з базою даних у поточний момент. Метод **Call** повертає повідомлення про помилку, приймаючи як параметр її код.

1.11. Компонент TIBSQLMonitor

Компонент **TIBSQLMonitor** відслідковує динамічні **SQL**-запити й дані, що пересилаються між сервером і клієнтською програмою. Тип операцій, що відслідковуються компонентом, вказується у властивості **TraceFlags**. Властивість може приймати кілька значень:

- Значення **tfQPrepare** вказує, що будуть відслідковуватися повідомлення про підготовку **SQL**-запиту або збереженої процедури до виконання.
- Значення **tfQExecute** вказує, що будуть відслідковуватися повідомлення про виконання **SQL**-запиту або збереженої процедури.
- Значення **tfError** вказує, що будуть відслідковуватися повідомлення про помилки, що виникають під час виконання запиту. У тексті повідомлення буде повернуто код помилки.
- Значення **tfStmt** вказує, що будуть відслідковуватися всі операції із запитами.
- Значення **tfConnect** вказує, що будуть відслідковуватися моменти з'єднання й розриву з'єднання з базою даних.
- Значення **tfTransact** вказує, що будуть відслідковуватися операції із транзакціями.

- Значення **tfBlob** вказує, що будуть відслідковуватися операції з полями типу **BLOB**.
- Значення **tfService** вказує, що будуть відслідковуватися різні сервісні операції.
- Значення **tfMisc** вказує, що будуть відслідковуватися всі інші операції.

У момент отримання від сервера повідомлення ініціюється подія **OnSQL**. Метод, що обробляє його, містить у параметрі **EventText** текст повідомлення, а в параметрі **EventTime** – час реєстрації повідомлення. Використовуючи даний метод, можна організувати роботу журналу подій в прикладній програмі.

1.12. Компонент TIBEvents

InterBase має спеціальний оператор, що дозволяє генерувати події при настанні певних умов. Сигнал про подію відсилається всім клієнтським програмам, підключеним до сервера. Використовуючи даний компонент, програма може реєструвати події БД, реалізувавши відстеження цих операцій у реальному часі.

Для відстеження виникнення подій у програмі призначений компонент **TIBEvents**. У властивості **Events** вказується список подій, що відслідковуються. В момент реєстрації такої події викликається метод, що обробляє подію – **OnEventAlert**. У його параметрі **EventName** вказується ім'я зареєстрованої події, а в параметрі **EventCount** – кількість подій, що надійшли з моменту початку реєстрації.

Для того щоб припинити відстеження подій, слід присвоїти параметру **CancelAlerts** значення **True.** Якщо властивість має значення **False**, то реєстрація подій буде тривати.

Для того щоб сервер сповіщав програму про настання якоїсь події, її необхідно зареєструвати на сервері. Для цього використовується метод **RegisterEvents**. Метод **RegisterEvents** викликається автоматично, якщо властивість **AutoRegister** має значення **True**. Метод **SetAutoRegister** дозволяє задати значення властивості **AutoRegister**. А метод **GetAutoRegister** повертає значення властивості **AutoRegister**. Визначити, чи була зроблена реєстрація події, дозволяє властивість **Registered.**

Для того щоб скасувати реєстрацію події на сервері, використовується метод **UnRegisterEvents.** Для задання подій зручно користуватися редактором **EventAlerter Events**, запустити який можна за допомогою кнопки, розташованої в правій частині властивості **Events** у інспекторі об’єктів. Вікно редактора показано на рис. 5.8.

Рис. 5.8. Редактор подій компонента **TIBEvents.**

Розглянемо приклад використання цього компонента. Для початку необхідно створити в базі даних тригер. Припустимо, що у БД є таблиця **ORDERS**. Тоді тригер для таблиці **ORDERS**, що викликається після видалення запису, створюється таким **SQL**-запитом:

```
CREATE TRIGGER DELETE_ORDER FOR ORDERS
ACTIVE AFTER DELETE POSITION 0
AS
```

```
BEGIN
 POST_EVENT 'DELETE';
END
```

Текст тригера, що викликається після вставки нового запису, описується іншим запитом:

```
CREATE TRIGGER NEW_ORDER FOR ORDERS
ACTIVE AFTER INSERT POSITION 0
AS
BEGIN
 POST_EVENT 'NEW_ORDER';
END
```

Як видно з тексту запитів, оператор **POST_EVENT** буде викликаний в момент спрацювання відповідного тригера – при видаленні або вставці запису.

Після створення нового проекту до нього треба додати модуль даних. У модулі даних варто розмістити компоненти **TIBDataBase**, **TIBTransaction** і **TIBEvents**. Після цього потрібно налаштувати з'єднання з базою даних та присвоїти властивості **Connected** компонента **TIBDataBase** значення **True.**

Компонент **TIBEvents** за допомогою властивості **Database** потрібно зв'язати з компонентом бази даних **TIBDataBase**, а потім встановити режим автоматичної реєстрації подій на сервері, присвоївши властивості **AutoRegister** значення **True.**

У метод, що обробляє подію **OnEventAlert** компонента **TIBEvents**, потрібно помістити код, що буде виводити повідомлення про подію в момент її виникнення:

```
procedure TForm1.IBEvents1EventAlert(Sender: TObject; EventName: String;
  EventCount: Integer; var CancelAlerts: Boolean);
begin
if EventName='DELETE' then ShowMessage('З таблиці видалено запис') else
  if EventName='NEW_ORDER' then ShowMessage('Вставлено новий
запис'); end;
```

Якщо тепер добавити до таблиці **ORDERS** новий запис чи витерти існуючий (наприклад, за допомогою утиліти **IBConsole** або іншої програми), то на екран буде виведено відповідне текстове повідомлення для користувача. При цьому повідомлення про подію отримають всі екземпляри програми (якщо їх запустити кілька).

Як правило, код оброблювача події **OnEventAlert** складніший – у ньому можна, наприклад, оновити вміст табличних даних, щоб користувач при роботі з програмою мав найновішу інформацію з БД.

2. Створення програми, що працює з БД сервера InterBase

2.1. Зв’язок з БД

Припустимо, що база даних вже створена раніше. Для прикладу використаємо демонстраційну БД **EMPLOYEE.GDB**, яка за замовчуванням встановлюється у папку

X:\Program Files\Common Files\Borland Shared\Data при інсталяції **Delphi**. У вказаній БД є таблиця

EMPLOYEE з інформацією про працівників певної організації. Вона, зокрема, містить такі поля:

EMP_NO – ідентифікатор працівника

FIRST_NAME – ім’я працівника

LAST_NAME – прізвище працівника

База містить також таблицю **DEPARTMENT**, що представляє собою список відділів, які належать організації. Ця таблиця містить, зокрема, такі поля:

DEPT_NO – ідентифікатор відділу, первинний ключ

DEPARTMENT – назва відділу

HEAD_DEPT – ідентифікатор відділу, якому підпорядковується даний

MNGR_NO – ідентифікатор менеджера (з поля **EMP_NO** таблиці **EMPLOYEE**)

BUDGET – бюджет відділу

LOCATION – розташування відділу

PHONE_NO – телефонний номер

Програма матиме обмежену функціональність і дозволятиме:

1. Вставляти у базу запис про новий відділ.
2. Відслідковувати подію вставки нового запису і оновлювати список відділів у вікні в реальному масштабі часу.
3. Проводити пошук відділів за їх розташуванням.
4. Сортувати список відділів за назвою та розташуванням.

Створимо у **Delphi** новий проект і збережемо його в окрему папку. Після цього скопіюємо до цього ж каталогу файл бази даних **EMPLOYEE.GDB**. Назвемо головну форму **fMain**. Добавимо до проекту новий модуль даних (команда меню **File/ New/ Data Module**), назвавши його **DM**. Далі розмістимо на модулі даних компоненти **IBDatabase, IBTransaction** та **IBQuery**. Після цього присвоїмо їм такі назви (властивість **Name**):

Компонент	Назва
IBDatabase	DBase
IBTransaction	Transaction
IBQuery	qrDeps

У список модулів **uses** для модуля головної форми слід добавити модуль даних (це можна зробити також за допомогою команди меню **File/ Use Unit**).

Після цього встановимо властивість **DefaultTransaction** компонента БД у значення **Transaction**, зв'язавши його з наявним компонентом транзакції. Після цього у полі **DatabaseName** слід задати назву файла БД, тобто **EMPLOYEE.GDB**. Далі за допомогою редактора властивостей компонента БД (він викликається подвійним клацанням мишки на компоненті) слід задати тип сервера – локальний, назву користувача **InterBase**, від імені якого буде відбуватися з'єднання з базою – **SYSDBA**, пароль цього користувача – **masterkey**, та множину символів для рядкових даних – **WIN1251** (цей набір символів підтримує кирилицю).

Властивість **LoginPrompt** компонента **IBDatabase** слід встановити у значення **False** (цим усувається операція перепитування користувача і пароля, оскільки вони задані явно за допомогою редактора). Властивості **AllowStreamedConnected** також слід присвоїти значення **False**.

Після цього для перевірки правильності налаштувань компонента БД задамо властивості **Connected** значення **True**. Якщо це вдалось зробити, то з'єднання з базою встановлюється і всі параметри задані вірно.

Далі для компонента транзакції **IBTransaction** встановимо рівень ізоляції транзакції. Для цього за допомогою подвійного клацання мишкою слід викликати редактор параметрів транзакції і вибрати рівень **Read Commited**.

Після цього для компонента **qrDeps** задамо базу даних (встановимо значення властивості **Database** у **DBase**). Даний компонент **IBQuery** формуватиме список всіх відділів та їх менеджерів на основі запиту до таблиць **DEPARTMENT** та **EMPLOYEE**. Текст відповідного **SQL**-запиту задається у властивості **SQL**:

```
      SELECT DEPT_NO, DEPARTMENT, LOCATION,
      (SELECT LAST_NAME || ' ' || FIRST_NAME FROM EMPLOYEE WHERE EMP_NO=DEPARTMENT.MNGR_NO)
AS NAME, BUDGET, PHONE_NO
      FROM DEPARTMENT
      ORDER BY 2
```

Оператор **| |** в тексті запиту означає поєднання рядків. Між лапками стоїть пробіл, що розділяє прізвище та ім ’я. Для перевірки правильності **SQL**-запиту можна встановити властивості **Active** цього компонента значення **True.** Якщо це вдалося зробити, то запит успішно виконано.

Далі на модуль даних слід розмістити компонент джерела даних **DataSource**, назвати його **dsDeps** і задати для нього джерело даних (властивість **DataSet**) – **qrDeps.**

На форму головного вікна слід покласти наступні компоненти: **MainMenu, StatusBar** та **DBGrid.**

Для головного меню слід задати таку структуру команд:

Дані	**Пошук**	**Сортування**
Вставити запис	Пошук відділу	За назвою
-		За розташуванням
Вихід		

Для компонента **DBGrid** потрібно задати такі властивості:

Властивість	**Значення**
Name	**dbg**
DataSource	**DM.dsDeps**
Align	**alClient**

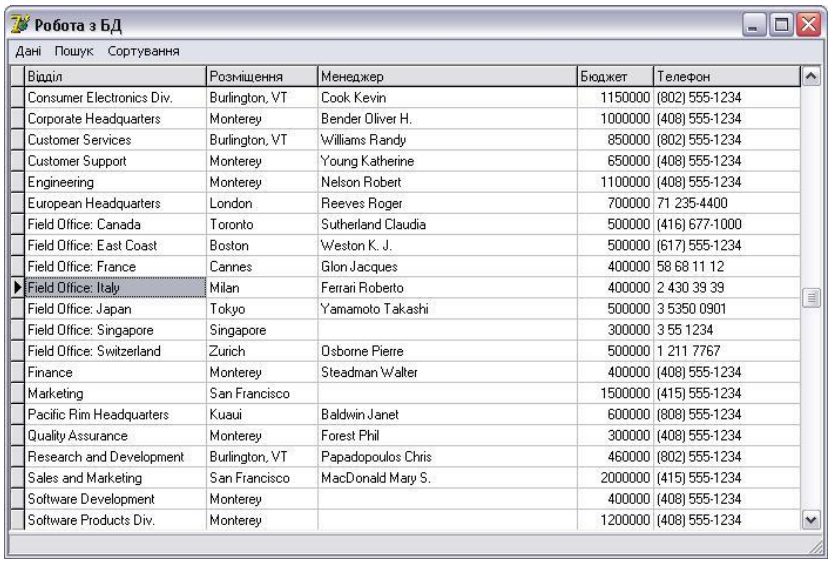


Рис. 5.9. Головне вікно програми із списком відділів

Тепер створимо множину колонок, які відображуватимуться у таблиці головного вікна. Для цього слід за допомогою команди контекстного меню **Columns Editor** викликати редактор стовпців компонента **DBGrid**. Далі у редакторі стовпців слід натиснути кнопку **Add All Fields** (вона розміщує для редагування всі наявні у зв’язаній множині даних поля). Оскільки ідентифікатор відділу для користувача не потрібен, то видалимо з редактора стовпців значення **DEPT_NO** (для цього слід натиснути кнопку **Delete Selected**). Щоб задати заголовок відповідного стовпчика, його слід виділити і в інспекторі об’єктів встановити значення властивості **Title:Caption.**

Тепер слід вибрати команду меню **Project/ Options** і у списку автоматично створюваних форм (**Auto-create forms**) першою поставити форму модуля даних. Для під’єднання програми до БД при її запуску для оброблювача події **OnCreate** головної форми потрібно добавити такі оператори:

```
DM.DBase.Open;
DM.qrDeps.Open;
```

Перший оператор з'єднується з базою даних, а другий виконує **SQL**-запит та отримує з таблиці відділів потрібну для користувача інформацію.

Загальний вигляд головного вікна програми під час роботи показано на рис. 5.9.

2.2. Програмування операцій по роботі з БД

2.2.1. Вставка нового запису

Дії щодо вставки у БД нового запису про відділ організації виконуватимуться при вибиранні команди меню **Дані/Вставити запис**. Для того, щоб користувач програми міг внести необхідні дані про новий відділ, створимо окрему форму (команда **File/ New/ Form** головного меню **Delphi**). Задамо для цієї форми такі властивості:

Властивість	Значення
Name	fDep
Caption	Дані про відділ
Position	poOwnerFormCenter
BorderIcons▸biMaximize	False

Остання властивість означає, що центр вікна, яке представляє форма, співпадатиме з центром вікна-власника. Зробимо також дану форму не автоматично створюваною. Для цього слід викликати команду меню **Project/ Option** і на сторінці **Forms** забрати форму **fDep** із списку автоматично створюваних форм (**Auto-create forms**) . Це варто зробити, оскільки операція по вставці нового відділу виконуватиметься відносно рідко, і немає необхідності постійно тримати у пам'яті структуру форми та її дані.

Розмістимо на формі компоненти, необхідні для внесення даних про відділ. Для задання назви, розміщення, бюджету і телефону використаємо компоненти текстового вводу **Edit** і назвемо їх відповідно **eName**, **eLocation**, **eBudget** та **ePhone**. Після цього очистимо для всіх цих компонентів властивості **Text.**

Для вибору менеджера нового відділу використаємо комбінацію компонентів **DBEdit** та **DBGrid** (назвавши їх, відповідно **eManager** та **dbgWorkers**). Перший з них просто показуватиме активного менеджера, а другий – дозволятиме із списку працівників вибрати когось в ролі менеджера. Щоб організувати список працівників у компоненті **dbgWorkers**, добавимо до цієї форми компоненти **IBQuery** та **DataSource** (присвоївши їм назви **qrWorkers** та **dsWorkers**). Так як множина даних **qrWorkers** повинна мати доступ до компонента бази даних, необхідно добавити модуль даних до списку uses модуля форми **fDep** (це можна зробити через команду **File/ Use unit**). Після цього для компонента **qrWorkers** слід задати компонент БД, встановивши властивість Database у значення **DM.DBase. SQL**-запит (тобто властивість **SQL** компонента **qrWorkers**), що формуватиме список всіх працівників у алфавітному порядку, наступний:

```
SELECT EMP_NO, LAST_NAME || ' ' || FIRST_NAME FROM EMPLOYEE ORDER BY 1
```

Для перевірки правильності введення запиту можна присвоїти властивості **Active** компонента **qrWorkers** значення **True.** Якщо це вдалося зробити, то текст запиту введено вірно.

Для джерела даних (компонента **dsWorkers**) потрібно задати набір даних, встановивши властивість **DataSet** у значення **qrWorkers**. Після цього для компонента **dbgWorkers** слід задати джерело даних, присвоївши властивості **DataSource** значення **dsWorkers**. Таким чином, компоненти **qrWorkers** (набір даних), **dsWorkers** (джерело даних) та **dbgWorkers** (відображення даних) послідовно з'єднані між собою. У компоненті **dbgWorkers** за допомогою редактора стовпців варто створити один стовпчик, що міститиме дані про працівників, і задати йому заголовок (наприклад, **"Працівники"**). Щоб вибраний працівник автоматично показувався у компоненті **eManager**, слід

задати для нього властивості **DataSource** (їх слід присвоїти значення **dsWorkers**) та **DataField** (встановивши для неї поле **F_1**).

Щоб користувач міг підтвердити або відхилити своє бажання вставити новий запис, добавимо на форму ще дві кнопки: "**ОК**" та "**Відмінити**". Для цього застосуємо компоненти **BitBtn**, давши їм назви **btnOK** та **btnCancel**. Для кнопки "**ОК**" задамо властивість **Kind** (вона відповідає за вид кнопки), присвоївши їй значення **bkOK**, а для кнопки "**Відмінити**" встановимо вид **bkCancel**. Після цього встановимо для кнопки **btnOK** властивість **ModalResult** у **mrNone.**

В кінці присвоїмо властивості **AutoSize** форми **fDep** значення True, чим забезпечимо автоматичний підбір її розмірів під розміри наявних на ній елементів керування. Щоб при створенні даної форми автоматично формувався список працівників, слід відкрити набір даних **qrWorkers.** Для цього потрібно створити оброблювач події **OnCreate** для форми:

```
procedure TfDep.FormCreate(Sender: TObject);
begin
 qrWorkers.Open;
end;
```

При внесенні інформації в БД слід перевірити задані користувачем дані на їх відповідність структурі БД. Так, для текстових полів **DEPARTMENT**, **LOCATION** та **PHONE_NO** існує обмеження по довжині – відповідно **25**, **15** і **20** символів[6]. Якщо користувач введе рядок більшої довжини, то при спробі його вставки виникне виняткова ситуація, і дані добавлені у базу не будуть. Тому потрібно для компонентів **eName**, **eLocation** та **ePhone** задати значення властивості **MaxLength** (відповідно – **25**, **15** і **20**), якою встановлюється найбільша можлива довжина тексту (довший рядок користувач ввести не зможе).

Поле **BUDGET** таблиці відділів може містити дійсне число, більше від **10000** та менше або рівне **2000000.** Для здійснення перевірки введеного значення на вірність слід створити оброблювач натискання кнопки **btnOK**:

```
procedure TfDep.btnOKClick(Sender: TObject);
var Cnt : real;
begin
 try
  Cnt:=StrToFloat(eBudget.Text);
 if (Cnt<=10000) or (Cnt>2000000) then raise Exception.Create('');
except
  MessageDlg('Недобре задано бюджет!', mtWarning,
 [mbCancel], 0); btnOK.ModalResult:=mrNone;
  Exit;
 end;
 ModalResult:=mrOK;
end;
```

Вікно для введення даних про відділ показане на рис. 5.10. У ньому для пояснення призначення відповідних елементів керування використано компоненти **Label**, а для оформлення списку працівників – компонент **GroupBox** та **Image** (зелений кружок із стрілкою).

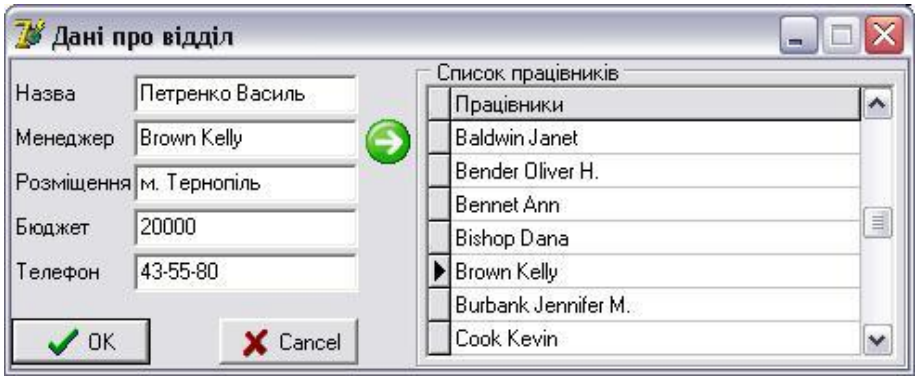


Рис. 5.10. Вікно для введення даних про відділ.

Для вставки нового запису про відділ використовується команда меню **Дані/ Вставити запис.** Тому оператори, які виконуватимуть ці дії, слід записати в оброблювачі події **OnClick** цього елемента меню. Щоб відпрацьовувати поточні **SQL**-запити по вставці нових записів, використаємо ще один компонент **IBQuery** (його слід розмістити у модулі даних, з'єднати з компонентом БД та назвати **qrTemp**). Код для вставки нового запису наступний:

```
procedure TfMain.miInsertClick(Sender: TObject);
var NO : word;
begin
 fDep:=TfDep.Create(Self);
 if fDep.ShowModal=mrOK then begin
  with DM.qrTemp do begin
   Close;
   SQL.Text:='SELECT MAX(DEPT_NO) FROM DEPARTMENT';
   Open;
   NO:=FieldByName('MAX').AsInteger;
   Inc(NO);

   Close;
SQL.Text:='INSERT INTO DEPARTMENT (DEPT_NO, DEPARTMENT, HEAD_DEPT, MNGR_NO, '+
 'BUDGET, LOCATION, PHONE_NO) '+
       'VALUES (:DEPT_NO, :DEPARTMENT, :HEAD_DEPT, :MNGR_NO, :BUDGET, :LOCATION,
 :PHONE_NO)';
   ParamByName('DEPT_NO').AsString:=IntToStr(NO);
   ParamByName('DEPARTMENT').AsString:=fDep.eName.Text;
   ParamByName('HEAD_DEPT').AsString:='000';
   ParamByName('MNGR_NO').AsInteger:=fDep.qrWorkers.FieldByName('EMP_NO').AsInteger;
   ParamByName('BUDGET').AsFloat:=StrToFloat(fDep.eBudget.Text);
   ParamByName('LOCATION').AsString:=fDep.eLocation.Text;
   ParamByName('PHONE_NO').AsString:=fDep.ePhone.Text;
   ExecSQL;
  end;
  with DM.qrDeps do begin
   Close;
   Open;
   Locate('DEPT_NO', IntToStr(NO), []);
  end;
 end;
 fDep.Free;
end;
```

У цьому коді перед вставкою нового запису здійснюється обчислення значення для первинного ключа. Як правило, первинний ключ у БД має тип цілого числа і його значення заповнюється автоматично шляхом використання тригера та генератора. Але у демонстраційній базі, з якою працює наша програма, такого механізму не передбачено, а саме поле ключа (**DEPT_NO**) має рядковий тип довжиною 3 символи, у якому зберігається числовий код відділу. Для обчислення значення первинного ключа застосовано наступний простий механізм: знаходиться найбільше значення у полі **DEPT_NO**, воно збільшується на 1 і записується як первинний ключ для нового запису (зауважимо, що при багатокористувацькому режимі роботи такий підхід неприпустимий).

Текст **SQL**-запиту для вставки нового запису наступний:

```
INSERT INTO DEPARTMENT (DEPT_NO, DEPARTMENT, HEAD_DEPT, MNGR_NO, BUDGET, LOCATION,
PHONE_NO) VALUES (:DEPT_NO, :DEPARTMENT, :HEAD_DEPT, :MNGR_NO, :BUDGET, :LOCATION,
:PHONE_NO).
```

Після зміни вмісту таблиці відділів здійснюється оновлення відповідного набору даних (він закривається, відкривається і переходить до нового запису). Так як демонстраційна БД

EMPLOYEE.GDB містить рядки, які не підтримують символів кирилиці, то текстову інформацію для нового відділу слід вводити латинськими літерами.

2.2.2. Вихід з програми

Для закінчення роботи програми застосуємо наступний оператор:

```
Application.Terminate;
```

Його слід розмістити в оброблювачі події **OnClick** елемента меню, який здійснює вихід з програми.

2.2.3. Пошук відділу

Для пошуку відділу використовується команда меню **Пошук/ Пошук відділу**. Ось код оброблювача відповідного елемента меню:

```
procedure TfMain.miFindClick(Sender: TObject);
var S : string;
begin
 S:=InputBox('Пошук відділу за назвою', 'Назва відділу', '');
if S<>'' then DM.qrDeps.Locate('DEPARTMENT', S, [loPartialKey, loCaseInsensitive])
end;
```

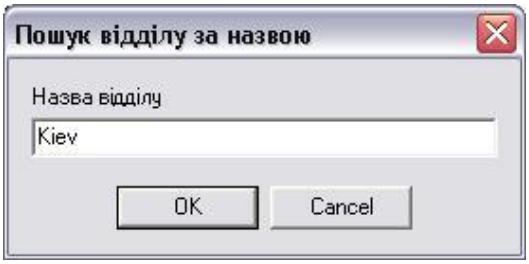


Рис. 5.11. Вікно пошуку.

Назва відділу, запис про якого потрібно знайти, отримується від користувача за допомогою функції **InputBox**. Діалогове вікно, яке формується нею, показане на рис. 5.11. Пошук здійснюється без врахування регістру літер (параметр **loCaseInsensitive**) та за частковим значенням (параметр **loPartialKey**).

2.2.4. Сортування

Щоб елементи меню, які відповідають за сортування даних ("**За назвою**", "**За розташуванням**"), відображували поточний вид сортування, задамо для них наступні параметри: властивості **GroupIndex** присвоїмо значення 1, а властивості **RadioItem** – значення **True.** Після цього для елемента меню **miSName** встановимо властивість **Checked** у **True** (за замовчуванням сортування здійснюється по назві відділу).

Код для сортування даних за назвою наступний:

```
(Sender as TMenuItem).Checked:=true;
with DM.qrDeps do begin
 Close;
 SQL.Text:='SELECT DEPT_NO, DEPARTMENT, LOCATION, '+
  '(SELECT LAST_NAME || '' '' || FIRST_NAME FROM EMPLOYEE
WHERE '+ 'EMP_NO=DEPARTMENT.MNGR_NO) AS NAME, '+
  'BUDGET, PHONE_NO '+
  'FROM DEPARTMENT '+
  'ORDER BY 2';
 Open;
```

```
      end;
```

Сортування за розміщенням робиться аналогічно:

```
      (Sender as TMenuItem).Checked:=true;
      with DM.qrDeps do begin
       Close;
SQL.Text:='SELECT DEPT_NO, DEPARTMENT, LOCATION, '+
 '(SELECT LAST_NAME || '' '' || FIRST_NAME FROM EMPLOYEE
 '+ 'WHERE EMP_NO=DEPARTMENT.MNGR_NO) AS NAME, '+
        'BUDGET, PHONE_NO '+
        'FROM DEPARTMENT '+
        'ORDER BY 3';
       Open;
      end;
```

Різниця між цими двома методами полягає лише у секції **ORDER BY** оператора **SELECT**: у першому випадку сортування проводиться за 2 стовпчиком набору даних, а в другому – за 3 стовпчиком.

3. Завдання на лабораторну роботу

Для бази даних, створеної протягом лабораторних робіт №1-3, розробити програму, яка дозволятиме:

- вставляти нові записи, редагувати та видаляти існуючі записи для всіх таблиць БД
- проводити пошук в головній таблиці за головним параметром
- сортувати дані з головної таблиці за трьома головними параметрами

Лабораторна робота №11

Тема: Транзакції в InterBase. Ізольованість транзакцій. Використання транзакцій.

1. Транзакції. Параметри транзакцій

1.1. Поняття транзакції

Транзакція – це механізм, що дозволяє поєднувати різні дії в логічні блоки й забезпечує можливість приймати рішення про підтвердження або скасування результату роботи всіх операцій у блоці.

Можливість пакетного скасування або прийняття змін у базі даних – тільки одна із властивостей транзакції. Транзакція має властивості атомарності, погодженості, ізоляції й тривалості (по-англійському **ACID** – **Atomicity, Consistency, Isolation, Durability**).

Розглянемо більш детально це визначення. Операції, про які йде мова – це **INSERT/ UPDATE/ DELETE** та **SELECT** . Якщо транзакція поєднує якісь операції в єдиний блок, то говорять, що ці дії виконуються в контексті даної транзакції.

Для ілюстрації роботи транзакції розглянемо приклад банківського переказу грошей. Коли клієнт ініціює переказ, то починається транзакція. Гроші знімаються з рахунку-джерела й переводяться на рахунок-приймач. Коли приходить підтвердження, що гроші переведені, транзакція завершується, тобто саме в цей момент відбувається "узаконювання" грошового переказу. Якщо ж хоча б один етап операції не відбувся, то транзакція відкочується і всі проведені в її межах зміни скасовуються. Тільки після підтвердження транзакції гроші, що прийшли на рахунок, стануть "реальними" (а зняті з рахунку – реально зникнуть).

Поняття транзакції як логічного блоку операцій, яким можна оперувати як єдиним цілим при підтвердженні/ скасуванні результатів, дуже популярне , так як воно дозволяє пояснити більшість феноменів, що відбуваються в програмах баз даних і в той же час досить просте для інтуїтивного розуміння.

Механізм транзакцій обов'язково використовується для всіх операцій у базі даних (про деякі особливі випадки буде розказано нижче). Транзакції реалізовуються як механізм керування бізнес-логікою в базах даних. Тому об'єднанням послідовності операцій у транзакцію управляє клієнтська прикладна програма.

Підтвердження або скасування результатів операцій, об'єднаних однією транзакцією, не означає, що всі ці операції виконалися успішно (або закінчилися помилкою). Підтвердження транзакції – це рішення про те, що в базі даних слід залишити результати роботи всіх операцій, які входять у транзакцію, незалежно від того, як вони закінчилися. Якщо клієнтська програма підтвердить результати транзакції, то при цьому підтвердяться результати всіх успішних дій (а в неуспішних дій просто не буде результатів, тому відбудеться лише формальне підтвердження). Якщо програма "вирішила" відкотити транзакцію, то всі результати – і успішні, і неуспішні, будуть анульовані.

1.2. Ізольованість транзакцій

Розглянемо детальніше, що відбувається з даними під час транзакції. Нехає у базі даних є дві таблиці, які відповідають за зберігання інформації про товари на двох різних відділах підприємства. Тоді переведення певної кількості товарів з одного відділу на інший відбувається, наприклад, так: спочатку з таблиці, що відповідає відділу-джерелу, віднімається відповідна кількість товару. Зауважимо, що після цього моменту товар вже "зник" з відділу-джерела і ще відсутній на відділі-приймачі. Після цього у таблицю, що відповідає відділу-приймачеві, вносяться дані про прихід товару. З цього прикладу видно, що всередині транзакції база даних у якісь

моменти перебуває в неправильному з погляду бізнеси-логіки стані (товар відсутній на обох відділах). Такий неправильний стан називається "нецілісним", а правильний – "цілісним".

Цілісний стан бази даних – це стан, в якому база містить коректну інформацію, яка відповідає правилам бізнес- логіки, що застосовується у програмі. Таким чином , транзакцію можна розглядати як механізм, що дозволяє переводити базу даних з одного цілісного стану в інший.

Під час виконання транзакції результати операцій, що виконуються в її контексті, невидимі для інших користувачів, аж до моменту підтвердження. Зважаючи на те, що всі дії виконуються в контексті транзакцій, то можна стверджувати наступне: результати операцій, що виконуються в контексті однієї транзакції, невидимі для операцій, що здійснюються в контексті інших транзакцій.

Звичайно, це справедливо в тому випадку, коли транзакції повністю ізольовані . В реальності домогтися такого важко через скорочення числа одночасно виконуваних транзакцій, які працюють із загальними даними. Тому для кожної транзакції передбачається рівень ізоляції, що обирається як компроміс між зниженням продуктивності та припустимим порушенням ізольованості.

Розглянемо ще такий приклад. Припустимо, у нас є документ, різні частини якого зберігаються в кількох таблицях – таблиці заголовків і таблицях з деталями. Очевидно, що документ повинен існувати тільки цілісним, при заповнених параметрах заголовка та коректно заданих даних в таблицях з подробицями. При цьому процес створення такого документа може бути відносно тривалим – спочатку користувач введе заголовок, потім заповнить деталі і т.д. В той же час, інші користувачі повинні побачити документ лише у цілісному виді, тобто не може бути документа без заголовка або з частковими чи некоректними даними.

У даному прикладі транзакція починається в момент початку створення/редагування документа й закінчується після закінчення цього редагування, тобто тоді, коли користувач, який редагує документ, підтвердить, що документ перебуває в цілісному стані.

Варто розрізняти поняття ізольованості, коли одна транзакція не бачить зміни, зроблені в контексті іншої (причому це настроюється рівнями ізоляції), і цілісності, коли стан всієї бази після завершення транзакції відповідає всім правилам бізнес-логіки. Протягом виконання транзакції в принципі можливі порушення цілісності. Механізм транзакцій служить для забезпечення ізоляції змін, спричинених операціями в контексті однієї транзакції, від операцій в інших транзакціях.

1.3. Механізм транзакцій в InterBase

Реалізація транзакцій в **InterBase** відрізняється від реалізації транзакції в багатьох інших **СУБД**. Це пов'язано з особливістю архітектури баз даних **InterBase**, яку називають **Multi Generation Architecture (MGA)** – багатоверсійною архітектурою.

InterBase – перша **СУБД**, у якій реалізовано багатоверсійну архітектуру. Саме багатоверсійна архітектура дозволяє організувати взаємодію користувачів таким чином, що читаючі користувачі не блокують пишучих, а також дає можливість дуже швидко відновлюватися після збоїв у базі даних і відмовитися від ведення протоколу транзакцій, а також надає інші переваги.

Суть багатоверсійної архітектури досить проста . Головна ідея полягає в тому, що всі зміни, проведені над конкретними записами, виробляються не над самим записом, а над його версією. Версія запису – це копія, що створюється при спробі його зміни. Для кожного запису можливе існування кількох версій, при цьому кожна транзакція бачить тільки одну з них.

Транзакція може або підтвердитися, або скасуватися. При підтвердженні транзакції **InterBase** спробує позначити попередню (вихідну) версію запису як вилучену й зробити поточну (змінену в рамках транзакції , що завершується) версію основною. Якщо тільки один користувач міняв запис, то саме так все й відбудеться – змінена версія запису стане головною і саме її побачать всі інші операції в транзакціях, які запустяться пізніше.

Якщо після запуску першої транзакції, але до підтвердження її результатів запуститься друга транзакція, у якій користувач бажає прочитати запис, що змінюється, то із-за того, що непідтверджені в першій транзакції зміни не можна побачити (у тому числі й у транзакції №2), то транзакція № 2 побачить попередню версію запису.

Проте пишучий користувач завжди може бути тільки один. Якщо спробувати редагувати той же запис одночасно в різних транзакціях, то, залежно від параметрів транзакції, виникне конфлікт відновлення записів. У випадку одночасної роботи багатьох користувачів можуть виникати складні комбінації версій, і це може привести до конфліктів.

1.4. Реалізація багатоверсійності

Кожна транзакція в **InterBase** має свій унікальний ідентифікатор – **transaction ID** або **TID** (фактично це номер транзакції з моменту створення бази даних або останнього **restore**). Кожна транзакція, запускаючись, одержує свій номер, що послідовно збільшується – тобто старші транзакції мають менші номери від нових.

Для обліку транзакцій використовують сторінки обліку транзакцій (**Transaction Inventory Page, TIP**). Коли в **InterBase** починається транзакція, то на сторінці обліку транзакцій з'являється помітка про те, що транзакція з певним ідентифікатором перебуває в стані виконання (є активною).

Всього є 4 можливі стани транзакції: активна (**active**), підтверджена (**Commited**), скасована (**Rolled back**) і лімбо (**limbo**). Статус активної мають транзакції, що виконуються в даний момент.

Підтверджені транзакції – це ті транзакції, що завершилися командою **Commit.** Скасовані – ті транзакції, що завершилися командою **Rollback.**

Транзакції **Лімбо** – це транзакції з невизначеним статусом, які виникають при використанні механізму двофазного підтвердження транзакцій, що застосовується при проведенні транзакцій відразу в кількох базах даних.

Коли операція в контексті транзакції з ідентифікатором **TID** змінює якісь записи, то для цієї зміни створюються версії записів. Кожна версія позначається ідентифікатором **TID** – на фізичному рівні це виглядає як номер транзакції в заголовку версії запису.

Якщо транзакція з номером **TID** підтверджується, тобто переходить у стан **Commited** (а процес підтвердження часто називають **commit**), то на сторінці обліку транзакцій робиться про це запис. При цьому ніяких дій над зміненими записами не відбувається, і в базі можуть міститися різні копії одного і того ж запису. Тоді при необхідності прочитати дані читаюча транзакція зчитує всі версії зміненого запису, бере номери транзакцій із заголовків записів і дійсним вважається той запис, чий номер найбільший.

При читанні версій записів відбувається додаткова перевірка на підтвердженість транзакції, що створила версію. Для цього в момент читання версій не просто порівнюються номери транзакцій, але й перевіряється на сторінці обліку транзакцій, в якому стані перебуває транзакція з певним **TID.**

Якщо транзакція, що створила версію запису, відхилена (тобто перебуває в стані **Rollbacked**), то така версія запису ніколи не буде актуальною і називається "сміттям" (**garbage**). Якщо ж транзакція, що створила версію запису, підтверджена (перебуває в стані **Committed**), то запис можна вважати претендентом на актуальність. Проте підтверджених транзакцій може бути кілька. Тому актуальною буде та версія запису, у якої **TID** найбільший.

При використанні багатоверсійної архітектури постійно накопичуються застарілі версії, що називаються "сміттям". Ці версії не є актуальними й підлягають видаленню. Процес видалення непотрібних версій записів називається збиранням "сміття".

Збирання "сміття" відбувається тоді, коли якась транзакція побажає прочитати даний запис.

Ця транзакція зчитує всі існуючі версії запису, з'ясовує, що певні версії застаріли, й видаляє їх.

1.5. Взаємодія транзакцій

Для опису процесу взаємодії транзакцій введемо кілька визначень.

Поточною називається транзакція, поведінка якої досліджується.

Зацікавлена транзакція – це транзакція, що конкурує з поточною.

Найстарша зацікавлена транзакція (**oldest interesting transaction**) – це найстарша транзакція, що конкурує з поточною транзакцією.

Кожна транзакція (і поточна також) має "маску транзакції", що є знімком сторінки обліку транзакцій, починаючи від найстаршої зацікавленої транзакції до поточної.

Найстарша активна транзакція (**oldest active transaction**, **OAT**) – це найстарша транзакція, що була активна в той момент, коли запускалася поточна.

Саме найстарша активна транзакція й займається збиранням "сміття". Найстарша транзакція збирає " сміття" тільки від транзакцій, що завершилися, і старші за неї. Тобто процес збирання "сміття" є динамічним – обов'язки по збиранню "сміття" постійно передаються від однієї транзакції до іншої.

1.6. Рівні ізоляції транзакцій

Транзакції забезпечують ізолювання змін, що проводяться в їхньому контексті, так що ці зміни невидимі користувачам аж до підтвердження транзакції. Але при

цьому може виникнути така ситуація. Припустимо, є таблиця з даними, до якої звертаються два користувачі одночасно – один з них змінює дані, а інший читає. Тоді виникає питання, чи може користувач, що читає таблицю, бачити підтверджені зміни, зроблені іншим користувачем.

Це й визначається рівнем ізоляції транзакції.

Рівень ізольованості транзакції визначає, які зміни, зроблені в інших транзакціях, будуть видимі в даній транзакції. Кожна транзакція має свій рівень ізоляції, що встановлюється при її запуску й залишається незмінним протягом всього її існування.

Транзакції в **InterBase** можуть мати три основних рівні ізоляції: **READ COMMITTED**, **SNAPSHOT** та **SNAPSHOT TABLE STABILITY**. Кожен з цих трьох рівнів ізоляції визначає правила видимості дій, які виконуються іншими транзакціями. Розглянемо рівні ізоляції більш докладно.

READ COMMITTED. Буквально переводиться як "читати підтверджені дані", проте це не завжди так. Рівень ізоляції **READ COMMITTED** використовується, коли треба бачити всі підтверджені результати паралельно виконуваних (в рамках інших транзакцій) дій. Цей рівень ізоляції гарантує, що непідтверджені дані, змінені в інших транзакціях, прочитатись не можуть, а підтверджені дані – можуть.

SNAPSHOT. Даний рівень ізоляції використовується для створення "моментального" знімка бази даних. Всі операції читання даних, виконувані в рамках транзакції з рівнем ізоляції **SNAPSHOT**, будуть бачити тільки стан бази даних на момент початку запуску транзакції. Всі зміни, зроблені в паралельних транзакціях (підтверджених і непідтверджених), невидимі в цій транзакції. У той же час **SNAPSHOT** не блокує дані, які він не змінює.

SNAPSHOT TABLE STABILITY. Це рівень ізоляції також створює "моментальний" знімок бази даних, але одночасно блокує на запис дані, задіяні в операціях даної транзакції. Це означає, що якщо транзакція **SNAPSHOT TABLE STABILITY** змінила дані в якійсь таблиці, то після цього дані в таблиці вже не можуть бути змінені в інших паралельних транзакціях. Крім того, транзакції з рівнем ізоляції **SNAPSHOT TABLE STABILITY** не можуть отримати доступ до таблиці, якщо дані в них вже змінюються в контексті інших транзакцій.

1.7. Параметри транзакцій

Програмісти, що використовують такі сучасні бібліотеки для доступу до баз даних **InterBase**, як **FIBPlus, IBProvider, IBX** та **IBObjects**, мають можливість гнучко управляти параметрами транзакцій для отримання найкращих результатів. Тому розглядатимемо параметри транзакцій саме в інтерпретації для цих бібліотек.

Налаштування параметрів транзакції здійснюється за допомогою перелічування набору констант, що визначають поведінку транзакції, наприклад, рівень ізоляції. Ці константи містяться

в **InterBase API** і мають такий вигляд: **isc_tpb_read, isc_tpb_write, isc_tpb_ read_committed** і т.д.

Звичайно префікс **isc_tpb_** опускається й константи для визначення параметрів транзакції пишуться без нього.

Розглянемо значення й синтаксис застосування кожної константи.

Всі параметри транзакції можна розділити на групи, кожна з яких відповідає за певний аспект поведінки транзакцій. Ці групи приведено в таблиці №6.1.

Таблиця № 6.1.

Групи параметрів	Константа	Короткий опис константи
Режим доступу	**read**	Дозволяє тільки операції читання
	write	Дозволяє операції запису
Режим блокування	**wait**	Встановлює режим відстроченого вирішення конфліктів
	nowait	При виникненні конфлікту негайно виникає помилка
Рівень	**read_committed rec_version**	Можливість читати підтверджені дані інших транзакцій. Додатковий параметр **rec_version** дозволяє читати записи, що мають непідтверджені версії
	read_committed no_rec_version	Можливість читати підтверджені дані інших транзакцій. Додатковий параметр **no_rec_version** не дозволяє читати запису, що мають непідтверджені версії
	concurrency	При запуску транзакції створюється миттєвий "знімок" стану бази даних (точніше, копіюється "маска транзакцій" на цей момент), тому зміни, зроблені в інших транзакціях, не видимі в цій транзакції
	consistency	Аналогічний рівню **concurrency**, але додатково ще блокує таблицю на запис

Режим доступу визначає, які операції можуть здійснюватися в контексті транзакції. За замовчуванням (тобто якщо ніяких додаткових параметрів не вказувати) ставиться режим читання-запису, тобто можуть здійснюватися будь-які операції. Слід зазначити, що транзакції з режимом доступу тільки для читання менше навантажують сервер, тому що не створюють зайвих версій записів.

Режим блокування визначає, як будуть вирішуватися конфлікти. Якщо виникає конфлікт, то в транзакції, що виявила його, є два виходи – або негайно згенерувати виняткову ситуацію, або почекати якийсь час, після чого знову спробувати розв'язати конфлікт. Відповідно є два варіанти режиму блокування – **wait** та **nowait**. За замовчуванням використовується режим **wait.**

Розглянемо випадок, коли транзакція **А** вставляє запис, але ще не підтвердила його. Потім у межах іншої транзакції **Б** робиться спроба вставити запис з тим же первинним ключем, що вже вставлено в транзакції **А**. При цьому транзакції з різними режимами блокування вестимуть себе по-різному:

- коли транзакція **Б** запущена в режимі **wait**, то вона буде очікувати завершення транзакції **А**, і якщо **А** завершиться підтвердженням (**commit**), то вставлений в **Б** запис буде визнано неактуальним і виникне помилка **Deadlock**, а якщо вона відкотиться (**rollback**), то зміни в транзакції **Б** будуть прийняті й вона зможе підтвердити їх (тобто зробити **commit**);

- якщо транзакція **Б** запущена в **nowait**, то негайно виникне помилка **'lock conflict on no wait transaction'.**

Розглянемо інший випадок: транзакція **А** змінила запис, але ще не підтвердила зміни.

Транзакція **Б** намагається видалити або змінювати цей же запис. Знову впливає режим блокування:

- якщо **Б** у режимі **wait**, то вона буде чекати, поки **А** не підтвердиться або не скасується; якщо **А** підтвердиться, то в **Б** виникне помилка **'Deadlock - update conflict with concurrent update'**, – тому що коли **А** підтвердить свої зміни, зміни в **Б** стануть неактуальними; якщо ж транзакція **А** відкотиться, **Б** одержить можливість підтвердитися;
- якщо **Б** у режимі **nowait**, то негайно виникне помилка **'lock conflict on no wait transaction'.**

Взаємне блокування записів різними транзакціями – це класична проблема при синхронізації доступу до ресурсу, при якій принципово неможлива подальша робота конкуруючих транзакцій. Для ілюстрації розглянемо дві транзакції **T1** і **T2** й два ресурси – **A** та **B**. У контексті розмови про бази даних ресурсами можуть бути, наприклад, записи в таблиці. Припустимо, виконується така послідовність дій:

1. Транзакція **T1** блокує ресурс **A**, після чого благополучно працює з ним.
2. Транзакція **T2** блокує ресурс **B**, після чого також з ним працює.
3. Транзакція **T1** бажає працювати з ресурсом **B**, для чого вона намагається встановити на нього блокування. Оскільки в цей час ресурс **B** вже зайнятий транзакцією **T2**, транзакція **T1** входить у стан очікування.
4. Транзакція **T2** бажає працювати з ресурсом **A**, намагається виконати його блокування і також переходить у стан очікування.

У результаті ми маємо ситуацію: транзакції не мають ніякого шансу продовжити своє виконання через те, що блокують одна другу. Такі взаємні блокування називають ще "мертвим" блокуванням (**deadlock**). Для сервера проблема вирішується вибором однієї із транзакцій як "жертви" та її відкат, при цьому інша транзакція отримує можливість виконатися до кінця.

Алгоритм розпізнавання ситуації взаємоблокування в **InterBase** не запускається відразу при виникненні конфлікту, – це зроблено з міркувань продуктивності. Замість цього очікується певний інтервал часу, що задається параметром **DEADLOCK_TIMEOUT** у конфігураційному файлі **InterBase ibconfig**, тільки після цього відбувається сканування таблиці блокувань на предмет взаємного блокування.

В реальній практиці програмування баз даних взаємоблокування виникають досить рідко.

1.8. Встановлення рівнів ізоляції

Як вже зазначалося, в **InterBase** є три основних рівні ізоляції. Теоретично існує також четвертий рівень ізоляції, – так зване **DIRTY READ** – "брудне читання". Транзакції з рівнем ізоляції **DIRTY READ** можуть читати непідтверджені дані в інших транзакціях. В

InterBase користувачеві не можна запускати транзакції з таким рівнем ізоляції, хоча теоретично багатоверсійна архітектура могла б забезпечити це.

Розглянемо рівень ізоляції **Read Committed**, що задається константою **read_committed.** Транзакція, запущена з таким рівнем ізоляції, може читати зміни, зроблені іншими паралельними транзакціями. Цей рівень ізоляції використовується для отримання найновішого стану бази даних.

Як видно з таблиці №1, існують два різновиди цього рівня ізоляції: **read_committed rec_version** та **read_committed no_rec_version**. За замовчуванням використовується варіант із параметром **rec_version**. Це означає, що при читанні запису просто зчитується остання підтверджена версія запису.

Варіант із **no_rec_version** більш складний. Використання рівня ізоляції **read_committed** з опцією **no_rec_version** зводиться до того, що транзакція буде не тільки намагатися отримувати найновішу підтверджену версію запису, але й вимагати, щоб не було ще новішої непідтвердженої версії.

При читанні запису в такій транзакції відбувається перевірка, чи не існує в нього непідтвердженої версії. Якщо існує, то відбувається наступне (в залежності від того, який режим блокування обрано):

- якщо **wait**, то транзакція чекає, поки не завершиться та транзакція, у якій створено непідтверджений запис. І якщо вона підтвердилася або скасувалася, то зчитується остання підтверджена версія.
- якщо блокування **nowait**, то негайно виникає помилка "**Deadlock**".

Очевидно, що рівень ізоляції **read_committed no_rec_version** може привести до різних конфліктів, і використовувати його треба обережно.

Рівень ізоляції **SNAPSHOT** задається параметром **concurrency**. Відзначимо, що **SNAPSHOT** – найбільш "рідний" режим **InterBase**, при якому переваги багатоверсійності проявляються найповніше. При його використанні транзакція робить "знімок" маски транзакцій у базі даних на момент запуску, і поки вона триває, бачить ті ж дані, які існували на момент її запуску. Ніякі зміни, що робляться паралельно, їй невидимі. При спробі в цій транзакції змінити дані, модифіковані іншими транзакціями вже після її запуску (маються на увазі як уже підтверджені, так і ще непідтверджені дані), виникає конфлікт.

Поки виконується транзакція з рівнем ізоляції **concurrency**, утримуються всі версії записів на момент її запуску, так як конкуруючі транзакції "бачать", що активна **SNAPSHOT i** не мають права зібрати версії записів, оскільки вони можуть (гіпотетично) знадобитися активній **SNAPSHOT**-транзакції.

Як правило, **SNAPSHOT** застосовується або для тривалих за часом запитів (звітів), або для організації блокування записів, щоб запобігти їх одночасне редагування/видалення іншими транзакціями.

Рівень ізоляції **SNAPSHOT TABLE STABILITY** задається параметром **consistency**. Цей рівень ізоляції аналогічний рівню **SNAPSHOT**, але додатково блокує таблицю на запис. Суть цієї ідеї проста – якщо транзакція з рівнем ізоляції **consistency** проводить зміни у якійсь таблиці, то транзакції з рівнями ізоляції **read_committed** й **concurrency** можуть тільки читати цю таблицю, а транзакції з таким же рівнем ізоляції (тобто **consistency**) не зможуть навіть читати.

Очевидно, що використання цього рівня ізоляції дозволяє організувати послідовні відновлення таблиці. Як правило, такий рівень ізоляції використовується тільки для коротких оновлюючих транзакцій. Транзакція запускається, проводить дуже короткі за часом зміни й відразу ж завершується. Інші транзакції, залежно від режиму блокування **wait** чи **nowait**, або чекають своєї черги, або генерують виняткові ситуації.

1.9. Рекомендації з використання параметрів транзакцій

Звичайно, для кожного конкретного завдання потрібно вирішувати питання параметрів транзакцій індивідуально. Проте, як правило, всі запити до бази даних діляться на групи – запити на читання найновішого стану бази даних, запити на поточні зміни, запити на читання довідкових таблиць, запити на читання даних для побудови звіту і т.д. Для кожної групи запитів можна встановити свою транзакцію (або групу транзакцій) з набором параметрів, потрібних для виконання завдання.

Розглянемо типову програму для БД, за допомогою якої користувач читає та змінює дані. У програмі є таблиця (**dbGrid** в **Delphi/C++Builder**), у якій користувач переглядає поточні дані. Таблиця містить **lookup**-поля, які заповнюються значеннями з довідників. Коли користувач знаходить запис, який потрібно змінити (або бажає додати запис у таблицю), то він натискає кнопку додавання/редагування і в діалозі, що з'явився, заповнює/змінює поля запису, після чого зберігає/скасовує редагування.

Для запиту **SELECT**, що читає дані в таблицю, варто використати транзакцію з доступом "тільки для читання" з рівнем ізоляції **READ COMMITED**, щоб мати можливість отримати найновіші дані з таблиці (вважатимемо, що програма багатокористувацька і одночасно можуть працювати кілька екземплярів програм у мережі). Стандартний набір параметрів може бути такий:

read
read_committed
rec_version
nowait

При цьому забезпечується читання всіх підтверджених іншими транзакціями записів, причому без конфліктів з паралельно працюючими пишучими й читаючими транзакціями. Таку транзакцію можна тривалий час тримати відкритою – сервер не навантажується версіями записів.

Для запиту на зміну/додавання даних можна використати транзакцію з рівнем ізоляції concurrency. Запит на оновлення даних в цьому випадку повинен бути дуже коротким : користувач заповнює необхідні поля, запускається транзакція, робиться спроба виконати запит, і потім, якщо не виник конфлікт на запис із іншою транзакцією, слідує підтвердження нашої транзакції або відкат, якщо був конфлікт (на рівні клієнтської програми конфлікти проявляються у вигляді виняткових ситуацій, які зручно відслідковувати за допомогою конструкцій **try...except** (**Delphi**) або

try.. .catch (**C++ Builder**)).

Параметри такої транзакції будуть наступними:

write
concurrency
nowait

Такий набір параметрів дозволить відразу (**nowait**) виявити те , що запис редагується/змінюється іншим користувачем (виникне помилка), а також запобігти спробам інших користувачів почати зміну записів, що міняються нашою транзакцією (у претендента виникне помилка "**update conflict**"). Треба відзначити, що перед редагуванням потрібно перечитати запис, тому що він міг бути змінений, а в кеші таблиці може усе ще перебувати стара версія.

Запити, які застосовуються для побудови звітів, повинні використовувати транзакцію з режимом доступу "тільки для читання" і з рівнем ізоляції **concurrency**:

read
concurrency
nowait

Така транзакція буде повертати строго ті дані, що існували на момент її запуску, – це дуже важлива особливість для тих звітів, які формуються за кілька проходів по базі даних.

Для запитів на читання довідкових даних можна використати транзакцію, аналогічну запиту **SELECT** для вибірки даних.

1.10. Дії за межами транзакцій

Раніше було сказано, що всі дії в **InterBase** виконуються в контексті транзакцій.

Однак існують об'єкти, про які говорять, що вони перебувають поза контекстом транзакцій.

Це генератори й зовнішні таблиці.

Генератор є лічильником, що зберігає певне ціле значення. Проте по своїй реалізації генератор є об 'єктом унікальним. На відміну від інших даних у базі значення генераторів зберігаються на найнижчому фізичному рівні – на особливих сторінках генераторів. Це дозволяє одночасно всім транзакціям одночасно бачити значення генераторів у будь-який момент часу. Ця риса дозволяє організовувати безконфліктні конкурентні вставки у паралельних транзакціях.

Зовнішні таблиці – це файли, що перебувають за межами основного файлу бази даних. Над зовнішніми таблицями виконуються лише операції вставки і вибірки (**INSERT/SELECT**). Відсутність відновлень у зовнішніх таблицях дозволяє відмовитися від зберігання у них версій записів, тому там завжди перебувають актуальні дані, і використання механізму транзакцій є непотрібним.

1.11. Двофазне підтвердження транзакцій

InterBase забезпечує унікальну можливість організовувати розподілені транзакції між різними базами даних і навіть різними серверами.

Суть двофазного підтвердження полягає в тому, що у прикладній програмі можна запустити транзакцію відразу на двох серверах. Простіше всього це зробити, прив'язавши один компонент транзакції відразу до двох компонентів баз даних.

Така транзакція запуститься відразу на двох серверах. Щоб синхронізувати процес завершення цієї транзакції, вводиться особливий стан, що називається **Prepared** (підготовлений). Цей стан означає, що транзакція завершилася на одному сервері і

готова перейти в стан **Committed**, як тільки транзакції на інших серверах також перейдуть у стан **Prepared**. Якщо ж транзакція хоча б

на одному із серверів, що беруть участь у процесі, завершиться **Rollback**, то всі транзакції зі стану **Prepared** теж відкотяться.

Якщо між серверами розірветься з'єднання в той момент, коли одна транзакція перейшла в стан **Prepared** і готова підтвердитися, то сервер не зможе вирішити, підтвердити або видалити зміни, зроблені цією транзакцією. Такі транзакції ще називають лімбо-транзакціями. Не варто використовувати двофазне підтвердження транзакцій на серверах, з'єднаних повільними каналами зв'язку (модемами, наприклад).

2. Завдання на лабораторну роботу

Модифікувати програму, створену протягом лабораторної роботи №5, таким чином, щоб операції читання даних з таблиць та модифікації даних у таблицях виконувалися різними транзакціями з необхідними для них рівнями ізоляції.

Лабораторна робота №12

Тема: Технології доступу до даних BDE та dbExpress.

Технології доступу до даних

Технологією доступу до даних називається система інтерфейсів, що забезпечує взаємодію між програмою і базою даних. У багатьох системах керування базами даних є бібліотеки, що містять інтерфейси прикладного програмування (**Application Programming Interface – API**), які представляють собою набір функцій, за допомогою яких можна виконувати з даними певні дії.

Щоб найповніше використовувати можливості сервера баз даних, необхідно працювати з ним прямо, через **API**. Проте це означає повну залежність програми від виду сервера й складність переходу на іншу платформу, так як це викликатиме необхідність переписування значного обсягу коду.

Питання незалежності (повної або відносної) програмного забезпечення від платформи покликані вирішити різні технології доступу до даних. Вони займають проміжне місце між **API** конкретного сервера та програмою користувача, даючи програмістові простий уніфікований механізм роботи з даними. Є багато різних технологій доступу до даних, серед них – **BDE**, **OLE**, **ODBC**, **ADO**, і на даний час розробляються нові, надійніші, зручніші в роботі та швидші технології.

1. BDE

Фірма **Borland** розробила власну технологію доступу до даних **SQL Links**, що має можливість взаємодіяти з **ODBC** через спеціальні "інтерфейси-мости". Технологія **BDE** є набором динамічних бібліотек, які надають інтерфейси, що дозволяють передавати запити на отримування або модифікацію даних з програми в базу даних і одержувати результат обробки. У процесі роботи бібліотеки використовують допоміжні файли мовної підтримки й інформацію про налаштування середовища.

Відзначимо, що фірма **Borland** перестала розвивати технологію **BDE** з 2002 року. Розробляти проекти на основі **BDE** можна, але вже не рекомендується, особливо для великих БД з тривалим строком підтримки.

Для розроблювача **BDE** надає кілька переваг:

- безпосередній доступ до локальних баз даних (**dBase, Paradox**, текстові файли);
- доступ до **SQL**-серверів (**Oracle, Sybase, MS SQL Server, InterBase, Informix, DB2**) за допомогою набору драйверів **Borland SQL Links**;
- доступ до будь-яких джерел даних, що мають драйвер **ODBC** (**Open DataBase Connectivity**), наприклад до файлів електронних таблиць (**Excel, Lotus 1-2-3**), і до серверів баз даних, що не мають драйверів **SQL Links**;
- створення прикладних програм "клієнт-сервер", що використовують різнорідні дані;
- використання **SQL**, у тому числі й при роботі з локальними даними;
- ізоляцію програми від засобів мовної підтримки.

На рис. 7.1 представлено схему, на якій показано зв'язок програми з базою даних через **BDE**. Для з’єднання з базами даних за допомогою механізму **BDE** використовуються компоненти,

розташовані на вкладці **BDE** палітри компонентів **Delphi.**

1.1. Створення псевдоніма бази даних

При роботі з таблицями локальних БД чи **СУБД** сама база розміщується або на локальному диску, або на віддаленому сервері. Звертання до бази даних відбувається за її псевдонімом (**Database Alias**). Псевдонім повинен бути зареєстрований на конкретній машині, з якої буде проводитися доступ до БД. Псевдоніми баз даних та інші налаштування **BDE** зберігаються у файлі **idapi32.cfg,** розташованому у тому ж каталозі, що й файли **BDE** (як правило – **X:\Program Files\Common Files\Borland Shared\BDE\).**

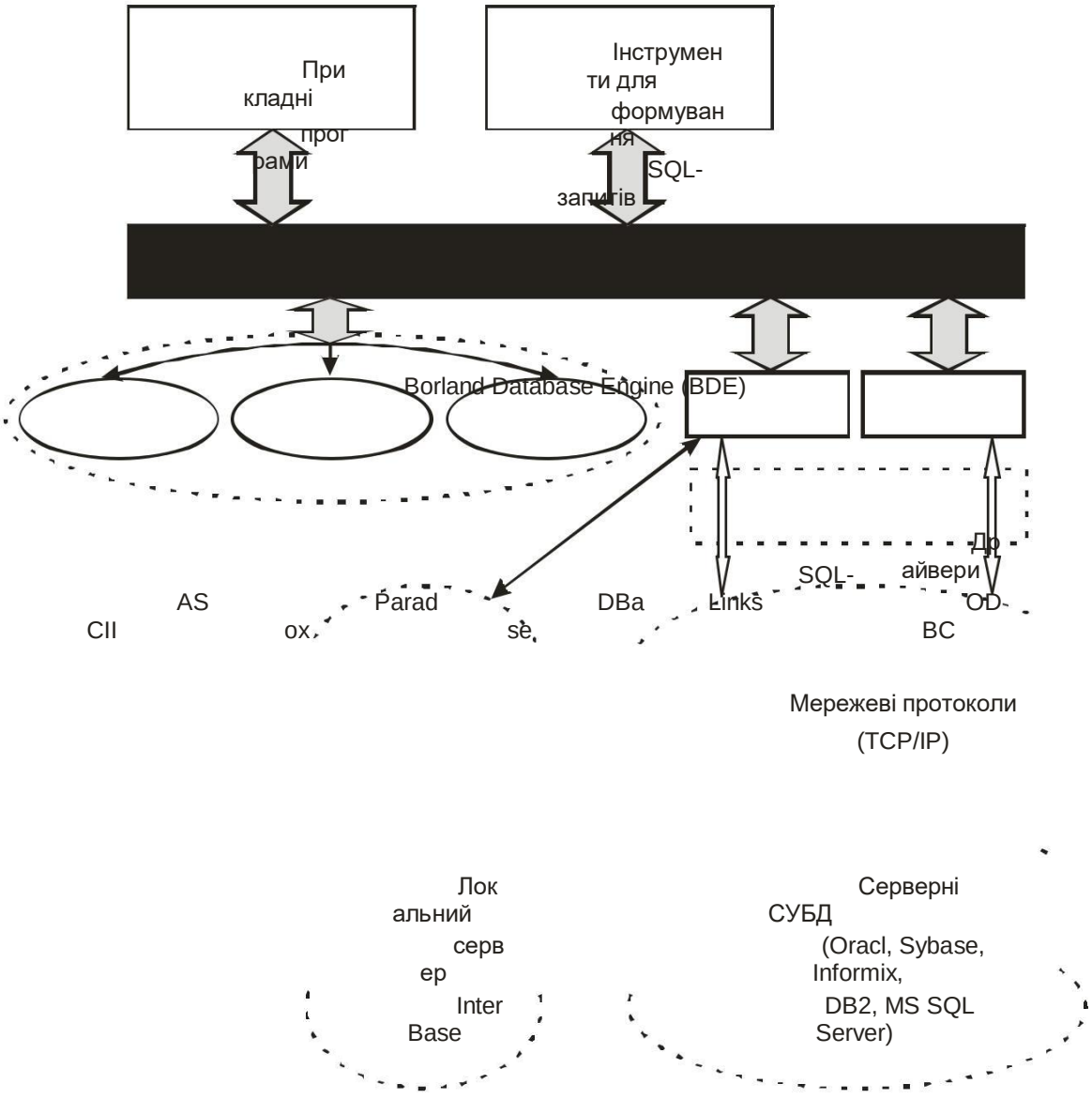


Рис. 7.1. Зв’язок прикладної програми з БД через **BDE.**

Рис. 7.2. Вікно вибору драйвера.

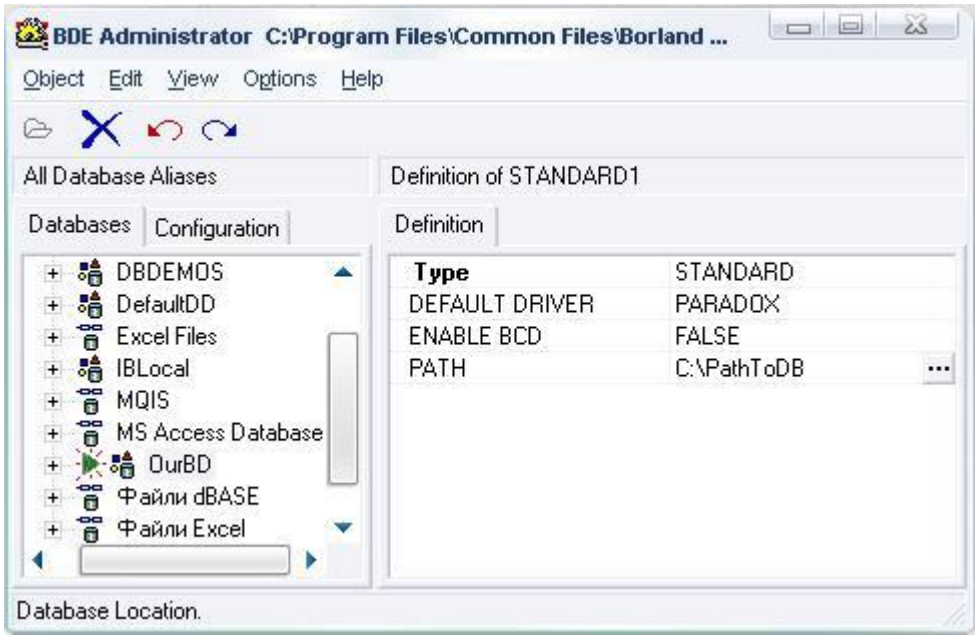


Рис. 7.3. Створення псевдоніму.

Створення і робота з псевдонімами баз даних відбувається за допомогою утиліти **BDE Administrator.** Для того щоб створити псевдонім, у головному меню утиліти потрібно вибрати пункт **Object/ New**. У вікні, що з'явиться після цього (рис. 7.2), слід вибрати драйвер для доступу до БД.

Створений псевдонім БД можна перейменувати (на рис. 7.3 він називається **OurBD**). У правій частині вікна утиліти як значення властивості **Path** вказується шлях до бази даних (у нашому прикладі шлях – **C:\PathToDB** (рис. 7.3). Шлях до каталогу можна вказати за допомогою діалогового вікна, що активується відповідною кнопкою в полі **Path**. Після виконання цих дій необхідно зберегти дані про створений псевдонім. Для цього використовується команда меню **Object/ Apply.**

Для створення таблиць бази даних можна скористатися утилітою **Database Desktop** або іншим подібним інструментом.

1.2. Налаштування BDE

Для доступу до параметрів драйвера в лівій частині вікна Адміністратора потрібно вибрати вкладку **Configuration** і необхідний драйвер. Система **BDE** взаємодіє з двома типами драйверів. Це драйвери **Native**, що є "рідними" драйверами **SQL Links**, і драйвери **ODBC.**

При виборі потрібного драйвера в правій частині вікна з'являється вікно з інформацією про його властивості:

- Властивість **TYPE** вказує тип сервера баз даних. Значення **FILE** означає, що використовується локальний сервер **Interbase** або таблиці **Paradox** чи **DBase**. Значення **SERVER** свідчить про використання віддаленого **SQL**-сервера.
- Властивість **LANGDRIVER** визначає драйвер мови, що використовується для кодування символів.

- Властивість **ENABLE BCD** вказує на необхідність переведення чисел у двійково-десятковий формат.
- Властивість **OPEN MODE** визначає режим доступу до даних.
- Параметр **MAX ROWS** визначає кількість записів, які входять у пакет даних. Використовується при роботі з віддаленими серверами.
- Властивість **SERVER NAME** задає ім'я віддаленої бази даних і містить шлях до неї.
- Параметр **SQLPASSTHRU MODE** регламентує режим взаємодії **BDE** із **СУБД** на рівні транзакцій:
  - ✓ значення **SHARED AUTOCOMMIT** означає, що транзакції виконуються автоматично, без відповідної команди від клієнтської програми;
  - ✓ значення **SHARED NOAUTOCOMMIT** вказує, що транзакції не виконуються автоматично. За їхнім виконанням стежить клієнтська програма;
  - ✓ значення **NOT SHARED** застосовується, коли для кожного компонента, пов'язаного із БД, створюється своє з'єднання з базою даних.
- Параметр **SQLQRYMODE** визначає політику виконання запитів:
  - ✓ значення **LOCAL** означає, що запит виконується локально;
  - ✓ значення **NULL** вказує, що буде застосовуватися змішаний механізм виконання запитів. Спочатку **BDE** відправляє запит на сервер; якщо результат не отриманий, то запит повторюється в режимі **LOCAL**;
  - ✓ значення **SERVER** вказує, що запит виконується на сервері.
- Властивість **DLL32** містить ім'я динамічної бібліотеки драйвера.

Системні налаштування поділяються на два типи. До першого типу належать параметри за замовчуванням, а до другого – різноманітні формати даних. Налаштування за замовчуванням доступні в розділі **Configuration► System► Init:**

- Властивість **DEFAULT DRIVER** містить назву драйвера, встановленого за замовчуванням для локальних баз даних.
- Властивість **LANGDRIVER** задає драйвер мови за замовчуванням.
- Параметр **LOCAL SHARE** визначає можливість розподілу доступу до локальних баз даних між **BDE** та іншими програмами при одночасному звертанні.

1.3. Компонент TDatabase

Як правило, при розробці програм, що використовують бази даних, за допомогою утиліт конфігурації **BDE** створюються псевдоніми, що вказують на тип і розташування даних. Компоненти **TTable**, **TQuery**, **TStoredProc** мають властивість **DatabaseName**, при налаштуванні якої на етапі проектування можна вибрати необхідний псевдонім зі списку, або явно вказати каталог, у якому розташовуються таблиці локальних БД. Однак іноді буває необхідно створити псевдонім динамічно або перевизначити параметри драйвера бази. У цьому випадку використовується компонент **TDatabase**, який поміщають на форму або в модуль даних. Якщо визначити властивість **DatabaseName** цього компонента, вона з'явиться в списку псевдонімів при установці однойменної властивості **DatabaseName** компонентів **TTable**, **TQuery** і **TStoredProc**. Якщо компонент не був розміщений на формі, то він буде створений динамічно, з налаштуваннями за замовчуванням. Так як компонент відповідає за взаємодію з **BDE**, його створення буде ініціюватися компонентами

TTable, TQuery та **TStoredProc.**

Змінити параметри псевдоніма бази даних можна за допомогою інспектора об'єктів. У властивості **TransIsolation** можна визначити рівень ізоляції транзакції.

Також параметри псевдоніма БД можна змінювати в редакторі властивостей компонента **TDatabase.** Вікно редактора показане на рис. 7.4. Викликається редактор подвійним клацанням мишки на компоненті.

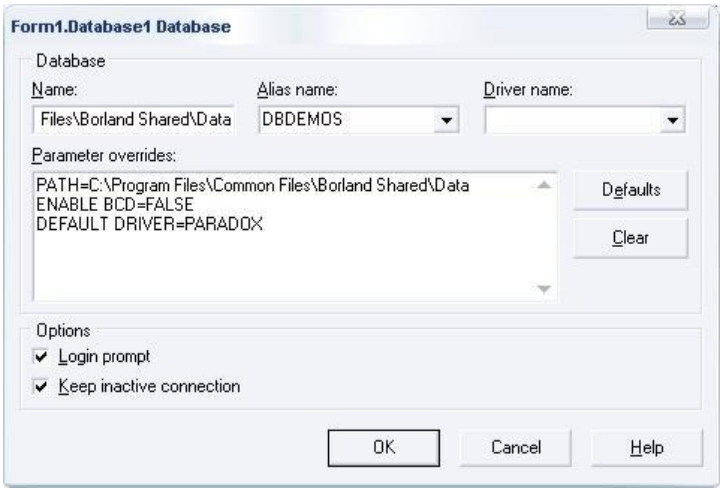


Рис. 7.4. Редактор властивостей компонента **TDatabase.**

Натиснувши кнопку **Defaults**, можна встановити стандартні налаштування псевдоніма. Звернутися до параметрів псевдоніма можна через властивість **Params.**

Серед іншого, компонент **TDatabase** використовується для скорочення числа звертань до бази даних. Наприклад, якщо на формі розташовано кілька компонентів, що взаємодіють з БД, то кожний з них звертається до сервера й передає йому параметри, що ідентифікують користувача. На сервері відбувається перевірка і приймається рішення про допуск користувача до роботи. Щоб мінімізувати число звертань до сервера, всі компоненти, призначені для роботи з БД, зв'язуються через властивість **Database** з компонентом **TDatabase**, а той, у свою чергу, зв'язується з базою даних. Якщо програма повинна працювати з кількома БД, то в модулі даних розміщується кілька компонентів **TDatabase.**

Вплинути на виконання серверних транзакцій можна шляхом кешування внесених користувачем змін замість спроби негайного збереження їх у базі даних. У цьому випадку зміни, зроблені користувачем, зберігаються у кеші даних методом **Post** і відсилаються на сервер методом **ApplyUpdates.** Для ввімкнення механізму кешування змінених даних властивості **CachedUpdates** компонентів **TTable** і **TQuery** встановлюється значення **True.**

Використовувати кешування змін зручно з різних причин. По-перше, значно знижується навантаження на мережу, оскільки пересилаються дані не кількох транзакцій, а тільки однієї. По-друге, якщо з якихось причин транзакція не була завершена, то метод поверне значення **False**, але при цьому незбережені зміни залишаться у кеші бази даних. Через деякий час можна спробувати зберегти їх ще раз.

При виконанні методу **ApplyUpdates** автоматично запускається транзакція, що завершується після внесення всіх змін на сервері. У випадку успішного завершення транзакції кеш очищується.

1.4. Компонент TSession

Компонент **TSession** реалізує поточний сеанс роботи з базою даних і призначений для керування всіма з'єднаннями з БД. Компонент створюється автоматично при запуску програми. Явне використання компонента може бути викликане необхідністю створення багатопоточних програм. Компонент може бути також застосований для керування налаштуваннями псевдонімів і драйверів.

Властивість **SessionName** визначає ім'я сеансу. Компоненти **TTable**, **TQuery** і **TDatabase** зв'язуються з компонентом **TSession** через властивість **Session**, у якій вказується назва сесії, задана програмістом. Властивості сесії, визначені компонентом **TSession**, діють на всі пов'язані з ним об'єкти **TDatabase.**

Властивість **Databases** містить масив зв'язаних компонентів **TDatabase**. За допомогою цієї властивості можна отримати доступ до активних баз даних, з якими встановлено з'єднання. А властивість **DatabaseCount** показує кількість пов'язаних баз даних.

Методи **GetDatabaseNames** і **GetAliasNames** повертають списки драйверів і псевдонімів баз даних. Метод **GetAliasParams** дозволяє розроблювачеві отримати параметри даного псевдоніма. А метод **GetTableNames** повертає імена таблиць, що містяться в базі даних.

Також компонент має методи, що дозволяють видаляти (додавати) псевдоніми баз даних та їх драйвери і робити інші дії.

1.5. Компонент TStoredProc

Компонент **TStoredProc** застосовується для виконання з програм збережених процедур. Збережені процедури можуть приймати вхідні параметри, передані їм з програми. Ім'я бази даних,

з якою зв'язується збережена процедура, міститься у властивості **DatabaseName**. А ім'я збереженої процедури міститься у властивості **StoredProcName**. При розробці в інспекторі об'єктів значення властивості вибирається зі списку збережених процедур, що формується при з'єднанні з базою даних.

У властивості **Params** міститься масив вхідних і вихідних параметрів процедури. Метод **ParamByName** повертає значення параметру процедури, якщо йому передано назву цього параметра. А метод **ExecProc** ініціює виконання збереженої процедури на сервері.

Для підготовки збереженої процедури до виконання слід викликати метод **Prepare.** Якщо необхідно звільнити ресурси, зайняті підготовленою до виконання процедурою, то використовується метод **UnPrepare.**

1.6. Компонент TUpdateSQL

Компонент **TUpdateSQL** може бути використаний для модифікації (додавання, зміни, видалення) даних на сервері за допомогою операторів **SQL**. Компонент містить методи-оброблювачі, у яких можна визначити набір команд, що виконуються при виклику методів **Insert,**

Delete, Update компонентів **TQuery** і **TTable.**

Компонент **UpdateSQL** може бути пов'язаний з компонентами **TQuery** і **TTable** через їхню властивість **UpdateObject**, у якій вказується ім'я компонента.

Компонент містить старі значення полів, які мало поле до внесення в нього змін, у полях із префіксом **"OLD_"**. Наприклад:

```
UPDATE "Country.db" SET Name=:Name WHERE :OLD_Name = "Vlad"
```

Параметр **OLD_Name** містить старе значення поля, по якому запис буде оновлено.

Для керування кожним окремим оновленням даних всередині транзакції, що переносить дані

з кеша на сервер, можна використати подію **OnUpdateRecord** відповідного компонента за допомогою

параметрів **UpdateKind** й **UpdateAction.**

1.7. Компонент TBatchMove

Цей компонент забезпечує копіювання даних з однієї таблиці в іншу. Таблиця-джерело вказується у властивості **Source**. Таблиця, у яку копіюються дані, вказується у властивості **Destination.** Копіювання здійснюється за допомогою методу **Execute.** Властивість **Mode** компонента визначає тип переміщення даних:

- Значення **batAppend** вказує, що нові рядки просто додаються в результуючу таблицю.
- Значення **batUpdate** змушує метод заміняти рядки таблиці-приймача відповідними рядками таблиці-джерела. Відповідність рядків визначається індексним полем.
- Використання значення **batAppendUpdate** приводить до того, що метод заміняє рядки приймача відповідними рядками таблиці-джерела. Відповідність рядків визначається по індексному полю. Якщо відповідні рядки не виявлені, то відбувається їхнє додавання в таблицю.
- Значення **batCopy** указує, що створюється таблиця з такою ж структурою, що й вихідна. Якщо таблиця існує, то метод видаляє її й створює нову.
- Значення **batDelete** змушує метод видаляти рядки приймача, що відповідають рядкам вихідної таблиці. Відповідність рядків визначається по індексному полю.

Властивість **Mappings** визначає відповідність полів вихідної таблиці полям приймаючої

таблиці. Для таблиць формату **Paradox** можна використати властивість **KeyViolTableName**. В цій властивості зберігається назва таблиці, у яку помістяться записи, які не вдалося скопіювати через обмеження посилальної цілісності або порушення ключа. А у властивості **ProblemTableName** вказується таблиця **Paradox**, у яку буде поміщено записи, не скопійовані через невідповідності типів полів або з іншої причини.

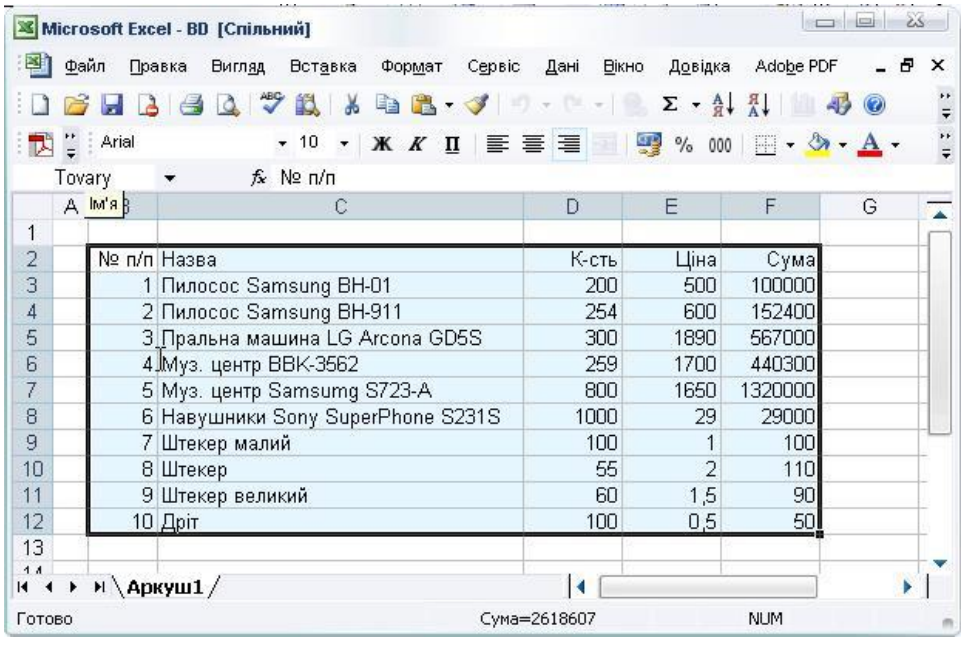


Рис. 7.5. Таблиця **Excel** з даними.

Для початку копіювання даних потрібно викликати метод **Execute**. Властивості **AbortOnKeyViol** та **AbortOnProblem** у випадку встановлення їх значень у **True** перервуть операцію копіювання при виникненні помилки.

1.8. Приклад зв'язку з Excel за допомогою BDE через драйвер ODBC

Розглянемо приклад програми, що працює з таблицею **Excel** через **BDE.**

Спочатку необхідно підготувати файл **Excel** і настроїти доступ до нього через драйвер **ODBC**. На рис. 7.5 показано вміст електронної таблиці у форматі **Excel**, що буде використовуватися в прикладі. В середовищі **Excel** потрібно виділити діапазон комірок і у полі **"Ім'я"** (зліва над робочим листом) задати йому назву, наприклад **Tovary** (рис. 7.5).

Створений файл потрібно зберегти (дамо йому назву **BD.xls**). Далі необхідно дозволити багатокористувацький доступ до файлу. Для цього слід виконати команду меню **Сервіс/ Доступ до книги.** У діалоговому вікні, що відкриється, потрібно помітити поле **"Дозволити спільне редагування"**, а потім знову зберегти файл.

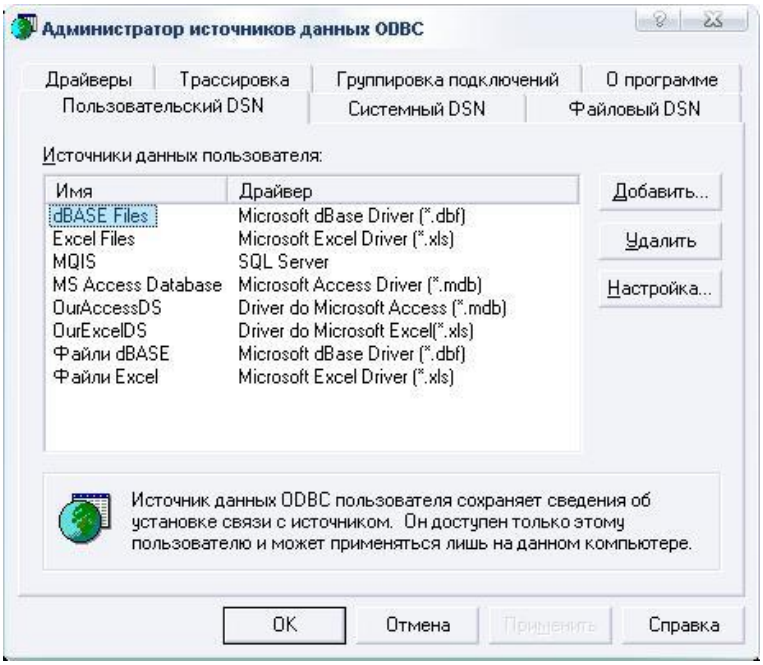


Рис. 7.6. Адміністратор **ODBC.**

Тепер можна створити **ODBC**-з'єднання з робочою книгою. Для цього потрібно запустити утиліту **ODBC Administrator**. У **Windows XP** вона називається **Data Sources** (Джерела даних) і перебуває

в розділі **Administrative Tools** (Адміністрування). В головному вікні утиліти (рис. 7.6) треба натиснути кнопку **Add** (Добавити) і у діалоговому вікні, що з'явилося, вибрати значення **Driver do Microsoft Excel.** Після цього натиснути кнопку **Finish** (Готово). У вікні, що з'явиться після цього, потрібно дати ім'я (наприклад, **OurExcelDS**) джерелу даних у полі **Data Source Name** (Ім'я джерела даних). Після цього потрібно за допомогою кнопки **Select WorkBook** задати робочу книгу **Excel**, з якої отримуватимуться дані. Для цього

вкажемо раніше створений файл – **BD.xls**. Потім потрібно за допомогою кнопки **Options** відкрити діалогове вікно налаштувань і зняти прапорець **Read Only** (Тільки для читання). Створене з'єднання з'явиться в списку джерел даних **ODBC** і автоматично з'явиться в списку доступних псевдонімів баз даних **BDE.**

Тепер можна перейти до розробки програми у **Delphi**. На формі слід розмістити компоненти **Table, DataSource, DBGrid** і кнопку. У властивості **DatabaseName** компонента **Table** треба вибрати зі списку псевдонім **OurExcelDS**. Властивості **TableName** слід присвоїти значення **Tovary**, яке визначає заданий раніше діапазон комірок в робочій книзі **Excel.**

Далі необхідно зв'язати компоненти **Table** і **DataSource**, а потім **DataSource** й **DBGrid.** Для кнопки **"Прийняти"** слід викликати метод **Post** та перечитати вміст таблиці:

```
Table1.Post;
Table1.Close;
Table1.Open;
```

Після цього можна скомпілювати програму, закрити **Delphi** (щоб не було конфлікту доступу до файлу) і запустити скомпільований модуль автономно. На рис. 7.7 показано вікно програми. Вона дозволяє змінювати дані і вводити нові (видаляти дані не дозволяє драйвер, що використовується в прикладі). При введенні нових даних границі іменованої області комірок в таблиці **Excel** будуть автоматично збільшуватися.

Рис. 7.7. Приклад роботи з **Excel** через **BDE.**

1.9. Приклад зв'язку з Access через BDE

Аналогічно до розглянутого прикладу, можна за допомогою **BDE** реалізувати доступ і до таблиць у форматі **Microsoft Access**. Спочатку в середовищі **Access** потрібно створити таблицю, у якій міститимуться дані. Приклад такої таблиці з назвою **Goods** показано на рис. 7.8. У ній поля **No** (Номер) та **Cnt** (Кількість) мають числовий (цілий) тип, поле **Name** (Назва) – текстовий, а поле **Price** (Ціна) – грошовий тип. Збережемо файл БД з назвою **accessdb.mdb.**

Робота з **Access** через **BDE** може відбуватися лише шляхом використання драйверів ODBC. Тому, аналогічно до випадку з **Excel**, потрібно зв'язати драйвер **ODBC** із джерелом даних. Для цього слід запустити утиліту **ODBC Administrator** (з Панелі

керування **Windows**). У список джерел даних користувача потрібно додати драйвер **Driver do Microsoft Access**. Створений зв'язок із джерелом даних назвемо **OurAccessDS**. Потім слід натиснути кнопку **Select** (Вибрати) і в діалоговому вікні вибору бази даних вказати ім'я раніше створеного файлу **accessdb.mdb**. Після натискання кнопки **OK** в **BDE** автоматично з'явиться псевдонім **OurAccessDS**, налаштований на з'єднання з вибраною базою даних.

Тепер потрібно створити новий проект у **Delphi**. На формі розмістимо компоненти **Database**, **Table**, **DataSource**, і **DBGrid**. Компонент **Database** слід зв'язати з базою даних за допомогою псевдоніма **OurAccessDS**. Властивості **DatabaseName** потрібно присвоїти значення **AccessDB** (назва БД), властивості **LoginPrompt** слід задати значення **False**, а властивості **Connected** – **True**.

Після цього властивості **DatabaseName** компонента **Table** потрібно присвоїти те ж значення, що і для компонента **Database** – **AccessDB**. Для компонента **Table** потрібно також задати назву таблиці у БД. Для цього слід властивості **TableName** присвоїти значення **Goods** (так ми назвали таблицю у базі).

На цій стадії можна активувати набір даних (властивості **Active** таблиці **Table1** присовїти **True**). Компонент **DataSource** потрібно зв'язати з таблицею **Table1**, а **DBGrid** – з джерелом даних **DataSource**. На формі потрібно також розмістити дві кнопки. Одну з них (кнопку **"Прийняти"**) назвемо **btnPost**, а іншу (**"Витерти"**) – **btnDelete**. В оброблювачах натискання кнопок потрібно вказати наступний код:

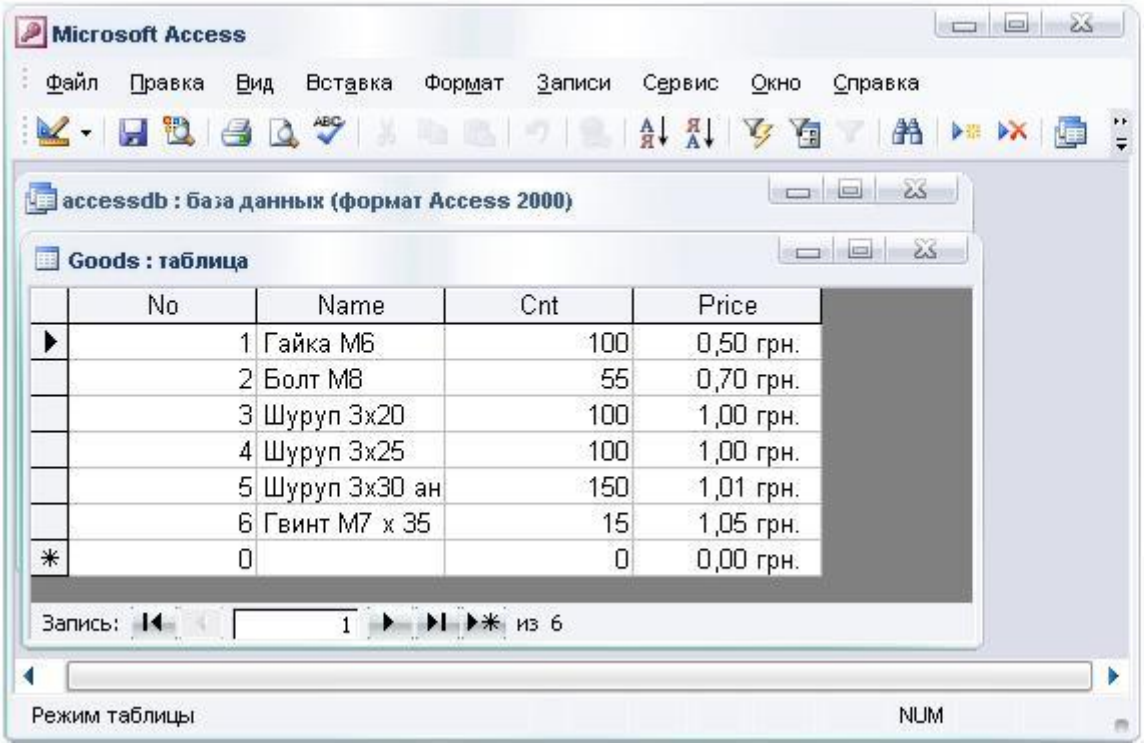


Рис. 7.8. Таблиця з даними в середовищі **MS Access.**

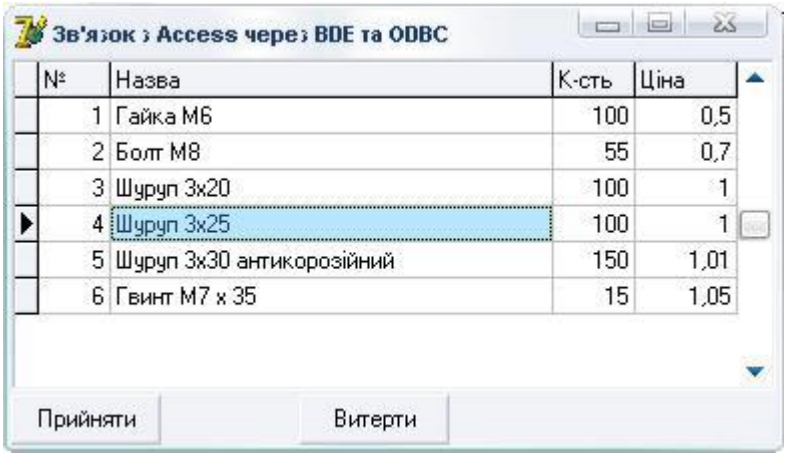


Рис. 7.9. Головне вікно програми.

```
       procedure TForm1.btnPostClick(Sender: TObject);
       begin
        if Table1.State In [dsInsert, dsEdit]
 then Table1.Post; Table1.Close;
        Table1.Open;
       end;

       procedure
TForm1.btnDeleteClick(Sender:
TObject); begin
        if Table1.State=dsBrowse then Table1.Delete;
       end;
```

Головне вікно програми показане на рис. 7.9.

Відзначимо, що для роботи програми, яка працює з даними за допомогою **BDE**, на комп'ютері повинна бути встановлена **BDE**. Тому при поширенні таких програм необхідно в інсталяційні пакети включати і **BDE**.

2. Технологія dbExpress

Технологія **dbExpress** була розроблена **Borland** для того, щоб замінити трохи застарілу технологію **BDE**. Ця технологія доступу до даних забезпечує взаємодію програм із **СУБД**. Вона не залежить від конкретної **СУБД** і має відкриті інтерфейси, що надають методи для виконання динамічно формованих запитів. **dbExpress** повертає тільки односпрямовані курсори, не підтримує кешування і, отже, не дозволяє прямо редагувати набори даних.

Технологія **dbExpress** є сукупністю драйверів та компонентів, які інкапсулюють з'єднання, транзакції, запити й набори даних, а також інтерфейсів, що забезпечують універсальний доступ до функцій **dbExpress**. Драйвери доступу до **СУБД** реалізовані у вигляді динамічних бібліотек. Ця особливість дозволяє легко поширювати програми, що використовують цю технологію доступу до даних. На машині кінцевого користувача обов'язково повинна бути встановлена клієнтська частина **СУБД**.

Драйвери для доступу до **СУБД** поставляються як **Borland**, так і сторонніми розроблювачами.

2.1. Компонент TSQLConnection

Даний компонент використовується для встановлення з'єднання із **СУБД** . З одним компонентом **TSQLConnection** може бути зв'язано кілька компонентів набору даних через їхню властивість **Connection**. Компонент взаємодіє із драйвером і двома файлами. Файл

dbxdrivers.ini містить список встановлених драйверів і набір налаштувань для кожного драйвера. Файл **dbxconnections.ini** містить базовий набір налаштувань.

Для встановлення з'єднання з базою даних треба властивості **Connected** присвоїти значення **True.** Відкрити або закрити з'єднання можна також за допомогою методів **Open** і **Close.** При відкритті й закритті з'єднання з БД виникають події **AfterConnect, AfterDisconnect, BeforeConnect** і **BeforeDisconnect.** За допомогою відповідних методів можна реалізувати різні перевірки даних.

Властивість **ConnectionName** містить ім'я з'єднання. Його можна вибрати зі списку. А властивість **DriverName** визначає драйвер **dbExpress**, що використовується для взаємодії із **СУБД**. Якщо необхідно отримати ім'я динамічної бібліотеки драйвера **dbExpress**, то слід прочитати значення властивості **LibraryName**. У властивості **Params** міститься список параметрів з'єднання. У параметрах можна вказати ім'я користувача й пароль, які будуть підставлятися автоматично при встановленні з'єднання.

Ці властивості отримують свої значення автоматично при виборі конкретного з'єднання з базою даних у властивості **ConnectionName.**

Властивість **ConnectionState** вказує поточний стан з'єднання.

Звернутися до конкретного набору даних можна через його індекс , використовуючи властивість **DataSets**. А метод **GetTableNames** дозволяє одержати список таблиць бази даних. Якщо його параметру **SystemTables** привласнити значення **True**, то метод також поверне імена системних таблиць.

Для отримання списку полів таблиці використовується метод **GetFieldNames**. У його параметрі **TableName** вказується ім'я таблиці, імена полів якої необхідно отримати.

Метод **GetProcedureNames** повертає список збережених процедур. Для одержання списку індексів використовується метод **GetIndexNames**. Як і раніше, у параметрі **TableName** вказується таблиця, для якої повертаються індекси.

Початок, фіксацію і відкат транзакції виконують методи **StartTransaction, Commit** і **Rollback** відповідно. Параметри транзакції містяться в записі **TTransactionDesc.**

Властивість **InTransaction** дозволяє з'ясувати, чи виконується в даний момент транзакція. Якщо транзакція запущена, властивість має значення **True.**

2.2. З'єднання із сервером баз даних

Для з'єднання із сервером використовується компонент **TSQLConnection**. З ним зв'язуються всі інші компоненти, яким потрібен доступ до даних у рамках технології **dbExpress.**

Властивість **ConnectionName** дозволяє вибрати з отриманого списку раніше налаштоване з'єднання із **СУБД**. Для того щоб налаштувати таке з'єднання, необхідно викликати редактор з'єднань. Для цього слід двічі клацнути на компоненті **TSQLConnection**. З'явиться вікно редактора,

показане на рис. 7.10. У списку **Connection Name** містяться існуючі з'єднання. У правій частині вікна, у редакторі властивостей **Connection Settings** вказуються параметри вибраного з'єднання. У верхній частині вікна розташована панель з кнопками, що дозволяють додати нове з'єднання в список, видалити його зі списку, перейменувати й перевірити його.

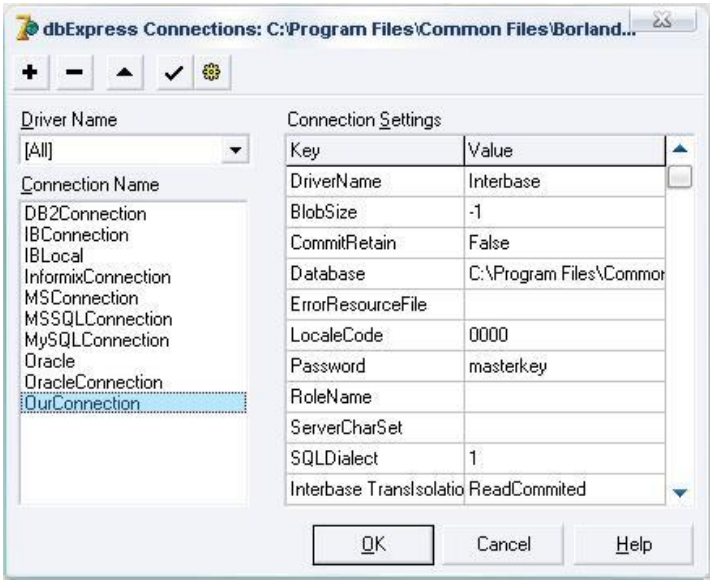


Рис. 7.10. Редактор з’єднань.

Для створення нового з'єднання потрібно натиснути на кнопку **"+"**. У результаті буде відображене діалогове вікно, показане на рис. 7.11, у якому потрібно вибрати тип драйвера зі списку **Driver Name** і вказати назву з'єднання в полі **Connection Name**. На цьому малюнку вибрано тип драйвера **Interbase**, а з'єднання названо **OurConnection.**

Рис. 7.11. Нове з’єднання.

У діалоговому вікні редактора новостворене з'єднання потрібно вибрати в списку **Connection Name.** Для нього в полі **DataBase** слід вказати шлях до БД (наприклад, **C:\Program Files\Common Files\Borland Shared\Data\EMPLOYEE.GDB**). Крім розташування бази даних, можна задати ім'я користувача й пароль у полях **User_Name** та **Password**. Також можна налаштувати інші параметри з'єднання.

2.3. Компонент TSQLDataSet

Компонент **TSQLDataSet** є односпрямованим набором даних. Він дозволяє виконувати запити

й збережені процедури, а також отримувати дані з таблиць. У властивості **CommandType** вказується тип виконуваної команди, а у властивості **CommandText** – її текст, ім'я таблиці чи збереженої процедури.

Для відкриття або закриття набору даних застосовуються методи **Open** та **Close.** Проте, можна оперувати й властивістю **Active** . Якщо запит або збережена процедура не

повертають набір даних, то потрібно використовувати метод **ExecSQL** . Його параметр **ExecDirect** вказує на необхідність підготовки команди перед виконанням, якщо вона містить параметри. Якщо команда не містить параметрів, параметру слід присвоїти значення **True.**

Через властивість **TransactionLevel** можна визначити рівень ізоляції транзакції.

З'єднання з компонентом **TSQLConnection** встановлюється через властивість **SQLConnection**. А властивість **IndexDefs** містить опис всіх індексів, що використовуються у результуючому наборі даних.

2.4. Компонент TSQLTable

Компонент **TSQLTable** інкапсулює односпрямований набір даних і використовується для подання інформації з бази даних у табличному виді. Компонент генерує запити на отримування всіх записів з усіма полями для вибраної таблиці.

Властивість **IndexName** містить ім'я поточного індексу, за яким відбувається сортування записів. Через властивість **IndexFields** можна звернутися до полів, що входять у даний індекс. У властивості **IndexFieldNames** перераховані поля, що входять в поточний індекс. Поля перелічуються через крапку з комою.

Метод **PrepareStatement** генерує **SQL**-запит для отримання даних від **СУБД**. А метод **DeleteRecords** застосовується для видалення з таблиці всіх записів.

2.5. Компонент TSQLQuery

Даний компонент використовується для виконання **SQL**-запитів на сервері. Він може повертати результати запиту у вигляді односпрямованого набору даних. У властивості **SQL** вказується текст **SQL**-запиту. А властивість **Text** представляє **SQL**-запит у вигляді текстового рядка.

Метод **PrepareStatement** викликається для підготовки запиту до виконання на сервері. Після того як запит буде підготовлений, його можна виконати за допомогою методу **ExecSQL.**

2.6. Компонент TSQLStoredProc

Компонент **TSQLStoredProc** використовується для виконання збережених процедур на сервері і отримання результатів їх роботи. Як і для інших компонентів, з'єднання з базою даних вказується
у властивості **SQLConnection.**

Після з'єднання з БД можна вибрати процедуру у властивості **StoredProcName**. При розробці в інспекторі об'єктів для цієї властивості відображається список збережених процедур, розташованих на сервері у вибраній базі. Після задання збереженої процедури компонент **TSQLStoredProc** запитує список її параметрів і використовує отриману інформацію для ініціалізації властивості **Params**. Деякі **СУБД** не дозволяють повернути інформацію про параметри збереженої процедури. У цьому випадку параметри необхідно визначати самостійно.

У властивості **Params** задаються параметри збереженої процедури. З її допомогою збереженій процедурі передаються аргументи й обробляються результати виконання процедури.

Якщо збережена процедура не повертає набір даних, то потрібно використати метод **ExecProc**. Для виконання процедури, що повертає набір даних, застосовується метод **Open.**

2.7. Компонент TSQLMonitor

Компонент **TSQLMonitor** перехоплює повідомлення, що пересилаються між компонентом **TSQLConnection** і **СУБД** , і зберігає їх у списку. Його зручно використовувати для відстеження взаємодій програми із сервером.

З'єднання з компонентом **TSQLConnection** встановлюється за допомогою властивості **SQLConnection** . У властивості **FileName** указується ім'я файлу, у який зберігається журнал повідомлень. Властивість **TraceList** містить список повідомлень, переданих від програми до сервера

й назад.

У властивості **AutoSave** можна вказати режим автоматичного збереження повідомлень у файл зі списку **TraceList**, в іншому випадку дану можливість потрібно реалізовувати вручну.

У властивості **TraceCount** указується число повідомлень, зареєстрованих на даний момент. А властивість **MaxTraceCount** дозволяє вказати максимальне число повідомлень, яке буде зареєстровано. Якщо вона приймає значення -1, то обмеження на кількість повідомлень немає. Якщо зазначено нульове значення, то **TSQLMonitor** не веде журнал повідомлень.

Метод **SaveToFile** зберігає вміст властивості **TraceList** у файл. А метод **LoadFromFile** дозволяє завантажити у властивість **TraceList** інформацію з файлу.

Подія **OnTrace** викликається, коли повідомлення зафіксоване, але ще не збережено в списку. Відразу після того як повідомлення було додано в список, ініціюється подія **OnLogTrace**. Метод-оброблювач цієї події можна використати для того, щоб періодично зберігати вміст списку у файл.

2.8. Приклад роботи з Interbase за допомогою dbExpress

Розглянемо приклад роботи із сервером **InterBase** за допомогою технології **dbExpress**. Для цього потрібно створити з'єднання із **СУБД InterBase**, як було описано в п.2.3 і продемонстровано на рис. 7.10 та 7.11. З'єднання назвемо **OurConnection.**

Створимо новий проект у **Delphi**. До на форму добавимо компоненти **SQLConnection**, **SQLQuery**, **SQLStoredProc**, **SQLMonitor** і **DataSource**. Для встановлення з'єднання у властивості **ConnectionName** компонента **SQLConnection** потрібно вибрати значення **OurConnection**. Властивості **LoginPrompt** слід присвоїти значення **False**. Компоненти **TSQLQuery**, **TSQLStoredProc** і **TSQLMonitor** слід зв'язати з компонентом **TSQLConnection** за допомогою властивості **SQLConnection.**

У властивості **StoredProcName** компонента **TSQLStoredProc** вкажемо ім'я збереженої процедури **MAIL_LABEL**. Ця процедура повертає кілька значень із таблиці **Customer**, приймаючи як аргумент номер покупця. У властивість **Params** компонента буде автоматично завантажений список параметрів.

Потрібно також вказати текст **SQL**-запиту. Для цього у властивість **SQL** компонента **SQLQuery** потрібно ввести рядок **SELECT * FROM Customer** і активувати компонент. Після цього можна зв'язати

компонент **DataSource** з **SQLQuery.**

Тепер перейдемо до компонента **SQLMonitor**. У властивості **FileName** потрібно ввести ім'я файлу, наприклад, – **Log.txt**. У ньому буде зберігатися журнал з'єднань із сервером. Властивість **AutoSave** встановимо у **True.** Після цього можна активувати компонент, для чого властивості **Active** слід присвоїти значення **True.**

На головній формі треба розмістити компоненти **Edit**, **Memo**, **StringGrid** і дві кнопки (рис. 7.12). Кнопку **"Виконати"** назвемо **btnExecute**, а кнопку **"Прочитати значення"** – **btnRead**. У компонент **StringGrid** шляхом послідовного переміщення по наборі даних, отриманого від компонента **SQLQuery**, будуть завантажені записи таблиці **Customer.** Оскільки **SQLQuery** може повертати тільки односпрямовані набори, використати компонент **DBGrid** не можна.

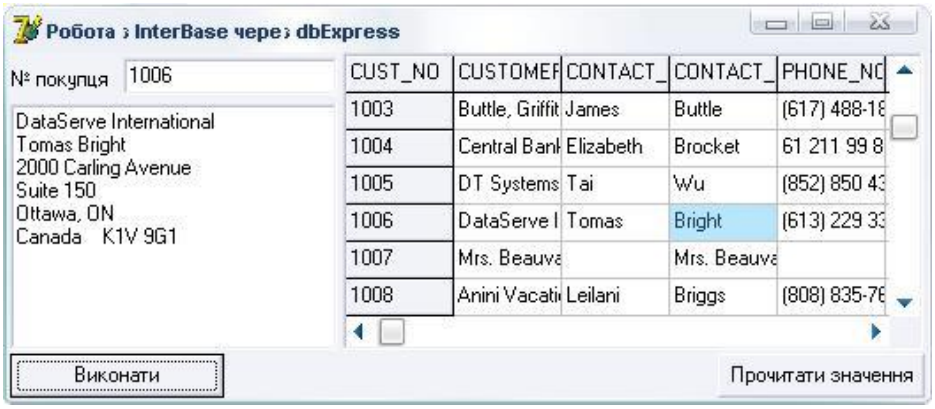


Рис. 7.12. Головне вікно програми.

У компоненті **Edit** вказуватиметься номер покупця, що передається як вхідний параметр збереженої процедури компоненту **SQLStoredProc**. При натисканні на кнопку **"Виконати"** процедурі передається цей номер, а результати, що повертає процедура, відображаються в компоненті **Memo.**

Кнопка **"Прочитати значення"** призначена для заповнення даними таблиці **StringGrid**. Вікно програми показане на рис. 7.12.

Виконавчий код програми наступний:

```
      procedure TForm1.btnExecuteClick(Sender: TObject);
      var I : integer;
      begin
       Memo1.Lines.Clear;
       SQLStoredProc1.ParamByName('CUST_NO').AsInteger:=Str
ToInt(Edit1.Text); SQLStoredProc1.ExecProc;
for I:=1 to SQLStoredProc1.Params.Count-1
 do
 Memo1.Lines.Add(SQLStoredProc1.Params[I].A
 sString);
      end;

      procedure TForm1.btnReadClick(Sender: TObject);
      var I, J : integer;
      begin
```

```
 // Заголовок таблиці
 StringGrid1.ColCount:=SQLQuery1.FieldCount;
 for I:=0 to StringGrid1.ColCount-1 do
  StringGrid1.Cells[I, 0]:=SQLQuery1.Fields[I].DisplayName;

 // Значення комірок таблиці
 while not SQLQuery1.Eof do begin
  Inc(J);
  StringGrid1.RowCount:=StringGrid1.RowCount+1;
  for I:=0 to StringGrid1.ColCount-1 do
   StringGrid1.Cells[I, J]:=SQLQuery1.Fields[I].AsString;
  SQLQuery1.Next;
 end;
end;

procedure TForm1.StringGrid1SelectCell(Sender: TObject; ACol,
  ARow: Integer; var CanSelect: Boolean);
begin
 Edit1.Text:=StringGrid1.Cells[0, ARow];
end;
```

Зауважимо, що при вибиранні певної комірки у таблиці з даними в компонент **Edit1** автоматично заноситься номер вибраного покупця. Це реалізується оброблювачем події

OnSelectCell компонента **StringGrid.**

Під час роботи програми дії користувача фіксуватимуться у файлі **Log.txt**. Ось приклад вмісту такого файлу-журналу:

```
INTERBASE - isc_attach_database
INTERBASE - isc_dsql_allocate_statement
INTERBASE - isc_start_transaction
SELECT * FROM Customer

INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - SQLDialect = 1
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_fetch
```

3. Завдання на лабораторну роботу

1. Створити базу даних, яка містить одну таблицю формату **Microsoft Excel.** Таблиця повинна мати щонайменше сім полів з даними різного типу. Вид даних, які мають зберігатися у БД, слід вибрати самостійно згідно завдання за варіантом (табл. №7.1). Розробити прикладну програму, яка працюватиме з вказаною таблицею і дозволятиме вводити нові записи, а також змінювати існуючі записи.

2. Створити прикладну програму, яка відображатиме дані з головної таблиці бази даних, створеної протягом Лабораторних робіт №1-3 і працюватиме за технологією **dbExpress.**

Таблиця №7.1.

Варіант	Вид бази даних
1	Дані про комплектуючі до комп'ютерів
2	Дані про бібліотечну літературу
3	Дані про абонентів бібліотеки
4	Дані про абонентів телефонної станції
5	Дані про географічні об'єкти області
6	Дані про студентів, що навчаються в університеті
7	Дані про працівників підприємства
8	Дані про структуру підприємства
9	Дані про автомобілі
10	Дані про фармацевтичну продукцію
11	Дані про меблеву продукцію
12	Дані про будівельні матеріали
13	Дані по складському обліку на підприємстві
14	Дані по прихідних та розхідних платіжних документах
15	Дані про відвідувачів WEB-сайту
16	Дані про кабельно-провідникову продукцію
17	Дані про радіоелектронну апаратуру
18	Дані про побутову техніку
19	Дані про продуктові товари
20	Дані про астрономічні об'єкти
21	Дані про напівпровідникові пристрої
22	Дані про рослинний світ країни
23	Дані про тваринний світ країни
24	Дані про підприємства області
25	Дані про історичні пам'ятки області
26	Дані про успішність студентів групи
27	Дані про CD та DVD- диски
28	Дані про продукцію легкої промисловості
29	Дані про міста світу
30	Дані про морських тварин та рослин

Лабораторна робота №13

Тема: Встановлення сервера баз даних Microsoft SQL Server. Адміністрування сервера.

1.1. Теоретичні відомості

1.1.1. Загальні відомості про СУБД Microsoft SQL Server 2016

Microsoft SQL Server – система управління базами даних (СУБД), розроблена корпорацією Microsoft. Microsoft SQL Server застосовується для керування реляційними базами даних середніх та великих розмірів. Входить до трійки (Oracle, IBM DB/2, Microsoft SQL Server) найпоширеніших СУБД у світі. В даний час найпопулярнішою версією цього програмного продукту є 10.5 або Microsoft SQL Server 2016 (MSS). З історією випуску версій MSS можна ознайомитись у [2].

MSS має широку лінійку варіантів випуску. Вибір варіанту залежить від призначення, розміру бази даних, обладнання, на якому встановлюватиметься програмне забезпечення, а також від фінансових можливостей покупця.

1.1.2. Інсталяція Microsoft SQL Server 2016

Microsoft Server 2016 - програмний засіб, розроблений корпорацією Microsoft і призначений для підтримки електронної документації Microsoft Server 2016. Файл, який буде пересланий, має розширення msi, з чого випливає, що це пакет стандартного інсталятора Windows. Подвійний клік мишею на цьому файлі запустить інсталятор Windows, який за кілька кроків встановить SQL. Загальний час установки не перевищує 5 хвилин і не вимагає спеціальних знань або навичок.

Після завершення інсталяції запустити можна за допомогою меню Пуск/Всі програми/Microsoft SQL Server 2016. Запуск спричинить появу на моніторі комп'ютера вікна приблизно такого вигляду, як на рис. 1.1.

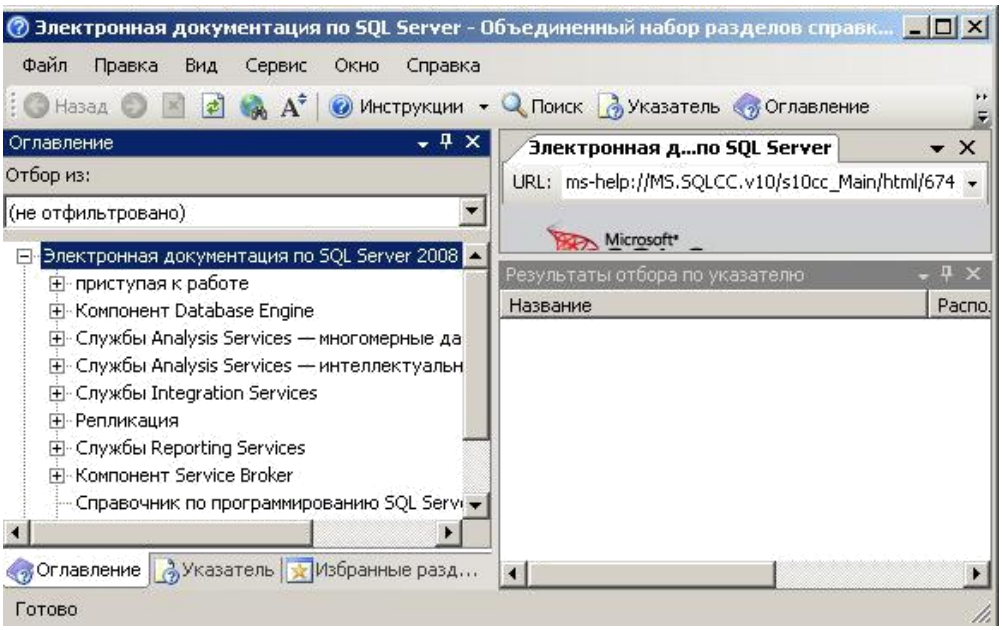


Рис. 1.1. Стартове вікно

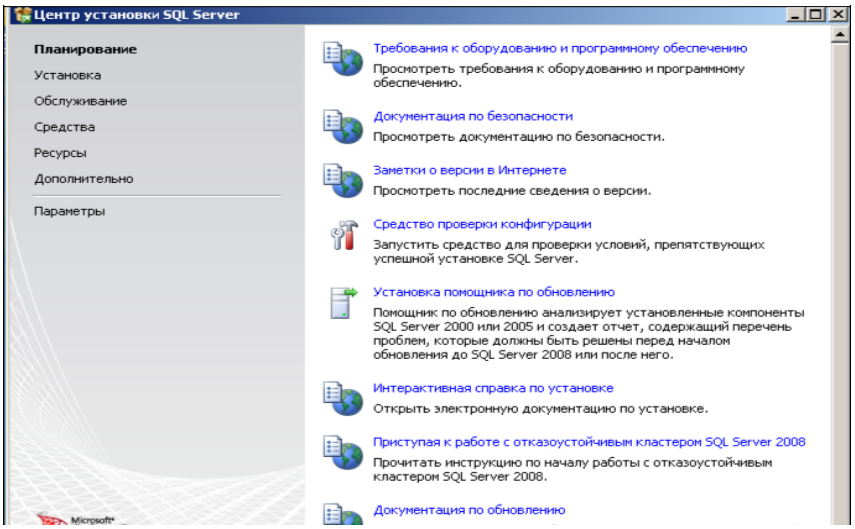


Рис. 1.2. Стартове вікно пакету встановлення

Як правило, при встановленні Microsoft SQL Server 2016 електронна документація (BOL) встановлюється автоматично.

1.1.3. Інсталяція Microsoft SQL Server 2016

Запуск пакета установки MSS здійснюється за допомогою файлу setup.exe, який розташований у кореневій папці дистрибутиву. Робота пакета установки починається з вікна, представленого на рис. 1.2. Зверніть увагу на пункт меню *Засіб перевірки конфігурації*. Виклик його дозволяє перевірити конфігурацію комп'ютера, призначеного для встановлення MSS. На рис. 1.3 представлено вікно, яке отримано після виконання такої перевірки. При успішному виконанні навпроти кожного правила, що перевіряється (всього їх 13) повинна стояти одна з відміток: Виконано або Незастосовно.

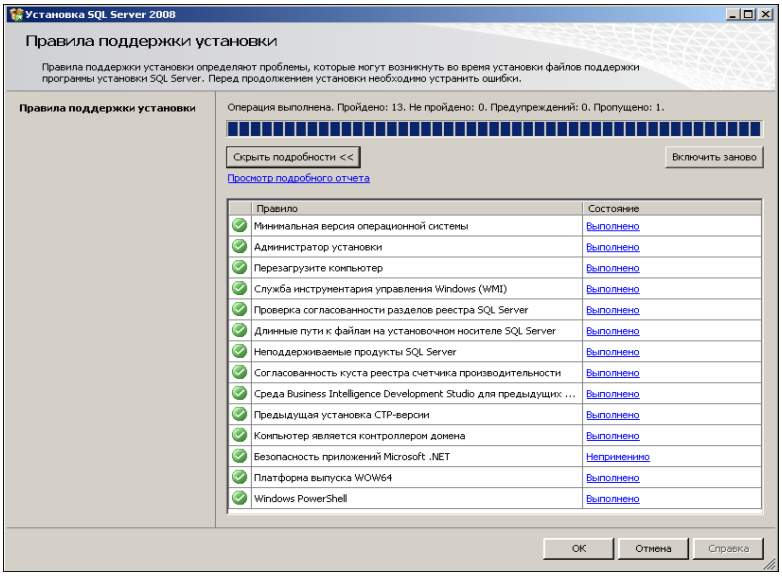


Рис. 1.3. Перевірка конфігурації комп'ютера

Пункт меню Параметри дозволяє встановити тип процесора (x86, x64) комп'ютера, призначеного для установки MSS, а також кореневу папку, в якій будуть розміщуватися файли MSS, що встановлюються.

Безпосередньо установка MSS виконується в меню *Установка*, в якому пропонується кілька варіантів. У нашому випадку найпридатнішим є перший варіант - *Нова установка ізольованого сервера*. Процес установки здійснюється покроково. На деяких кроках задаються питання, на які необхідно відповісти. Процес встановлення може вимагати доступність Інтернету. Загалом установка триває не більше 20 хвилин.

Детально процес установки MSS описаний у [3, 4].

1.1.4. Налаштування сервера та клієнта

Для налаштування MSS використовується спеціальний програмний засіб – SQL Server Configuration Manager (SCM). SCM встановлюється автоматично в процесі установки MSS. Запустити SCM можна після встановлення MSS за допомогою меню Пуск/Всі програми/Microsoft SQL Server 2016/Налаштування/Диспетчер конфігурації SQL Server. На рис. 1.4 представлено стартове вікно SCM.

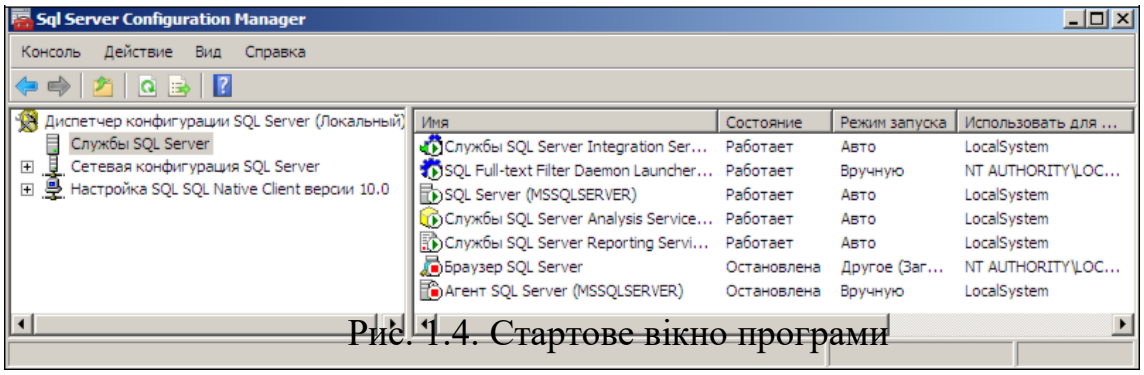


Рис. 1.4. Стартове вікно програми

SQL Server Configuration Manager

SCM дозволяє конфігурувати, зупиняти та запускати служби (компоненти) MSS, налаштовувати мережевий доступ до сервера, встановлювати параметри мережного доступу клієнтів, створювати псевдоніми сервера та ін.

1.2. Завдання

1.2.1. Інсталяція СУБД Microsoft SQL Server 2016

1. Отримайте у викладача або web-сайті виробника для встановлення MSS та створіть його копію.
2. Досліджуйте вміст кореневої папки дистрибутивного диска, знайдіть файл setup.exe.
3. Відкрийте файл setup.exe.
4. За допомогою інсталяційного пакета здійсніть попередню перевірку конфігурації комп'ютера; якщо виявлено помилки конфігурації, усуньте їх і повторіть перевірку.

5. Отримайте скріншот протоколу успішної перевірки конфігурації та помістіть його у звіт про контрольну роботу.

6. Встановіть MSS; всі параметри, що вводяться під час процесу інсталяції, фіксуйте за допомогою скріншотів і відображайте у звіті про контрольну роботу.

7. Запустіть SCM і переконайтеся, що SQL Server знаходиться в стані «працює».

1.2.2. Дослідження конфігурації сервера

1. За допомогою SCM з'ясуйте список усіх служб MSS.
2. За допомогою SCM перейдіть до списку зупинених служб MSS.
3. За допомогою SCM з'ясуйте розташування системної бази даних master та її журналу транзакцій.
4. З'ясуйте: як забезпечити автоматичний старт служб MSS під час завантаження операційної системи.
5. З'ясуйте: як запустити та зупинити служби MSS за допомогою
   a. SCM.
6. Результати дослідження відобразіть у звіті.

1.2.3. Дослідження конфігурації клієнта

1. За допомогою SCM з'ясуйте перелік мережевих протоколів доступу до MSS; визначте, які з них вимкнено.
2. За допомогою SCM з'ясуйте номер порту TCP за промовчанням для доступу до MSS.
3. Призначте TCP/IP-з'єднання псевдонім SRV-SNF, де S – перша буква вашого прізвища, N – імені, F – по-батькові.
4. Результати дослідження відобразіть у звіті.

Контрольні питання

1. Перерахуйте параметри MSS, які ви вводили при інсталяції MSS і поясніть їхній зміст.

2. Поясніть поняття «архітектура клієнт-сервер».

3. Поясніть призначення SCM.

4. Поясніть поняття "сервіс MSS".

5. Перерахуйте мережеві протоколи, за допомогою яких можна отримати доступ до MSS.

6. Назвіть стандартний номер порту TCP для доступу до MSS.

Навіщо використовується псевдонім сервера?

2. Створення та видалення бази даних за допомогою Microsoft SQL Server Management Studio.

2.1. Теоретичні відомості

2.1.1. Створення та видалення бази даних за допомогою Microsoft SQL Server Management Studio

Один екземпляр MSS може керувати кількома базами даних. Вся інформація про бази даних, керованих MSS, зберігається в системній базі даних master.

Для створення бази даних SMS слід скористатися контекстним меню Створити базу даних, що дозволяє відобразити наступне вікно (рис. 2.1).

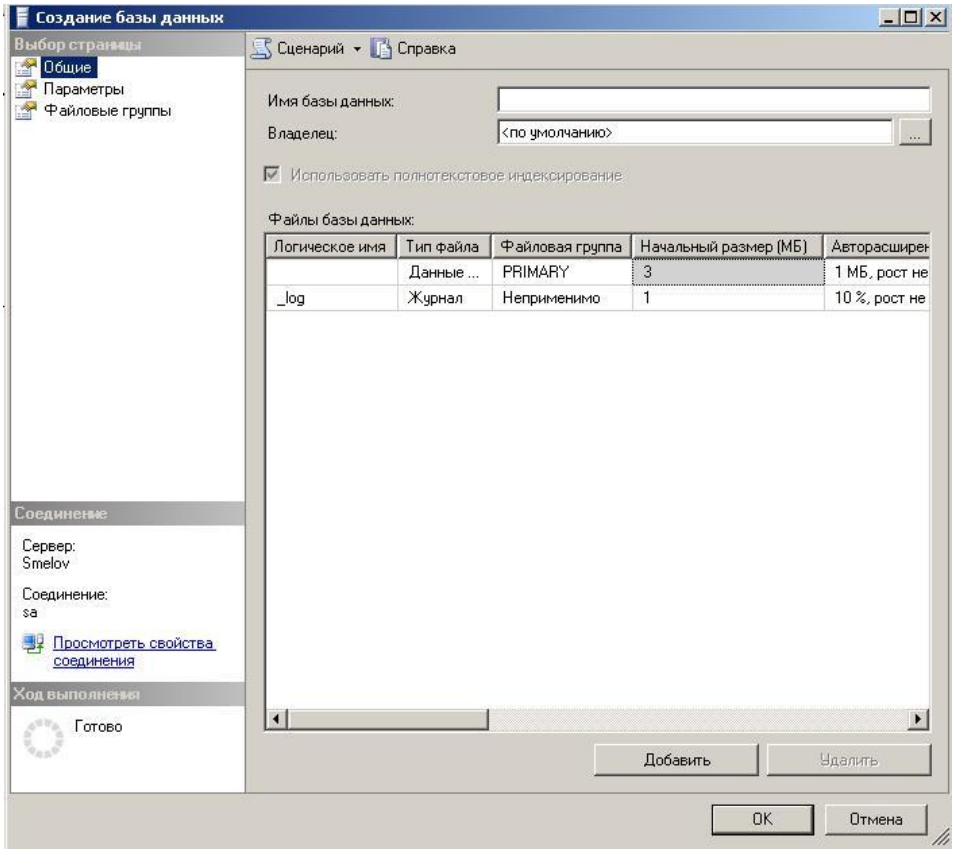


Рис. 2.1. Створення бази даних

Створення бази даних зводиться до заповнення трьох сторінок, які вибираються за допомогою меню в лівому верхньому куті вікна. Слід звернути увагу на пункт меню Сценарій, який дозволяє автоматично сформувати SQL-сценарій створення бази даних.

Видалити базу даних можна за допомогою контекстного меню, вибравши пункт *Видалити*.

2.1.2. Створення та видалення бази даних за допомогою SQL

Найпростіший спосіб сформувати SQL-сценарій – скористатися меню Сценарій вікна Створення бази даних SMS (рис. 2.1).

Крім того, для створення SQL-сценарій створення бази даних можна скористатися контекстним меню SMS Створити сценарій для бази даних/використовуючи CREATE (рис. 2.2), за допомогою якого можна отримати сценарій створення вказаної бази даних.

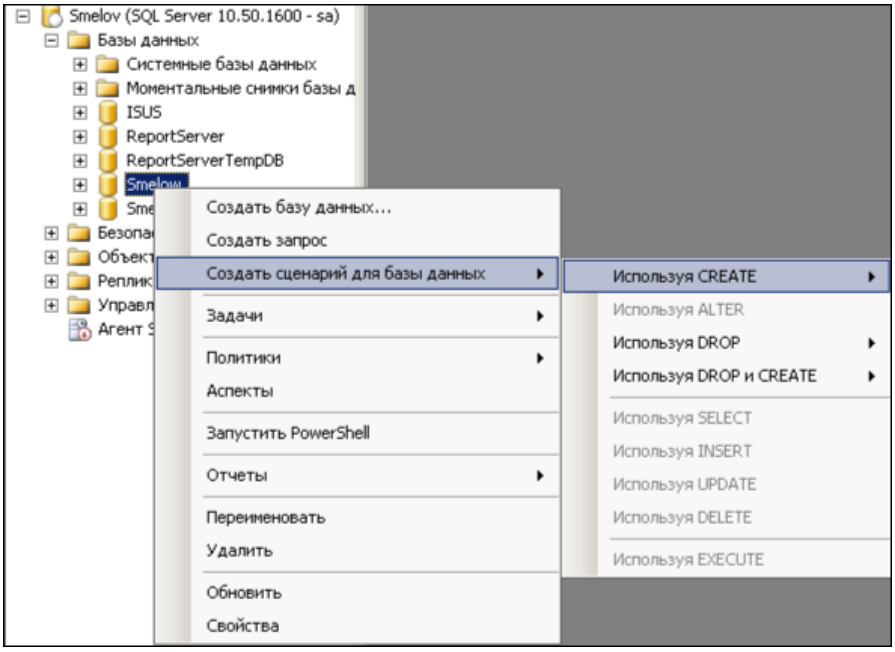


Рис. 2.2. Створення сценарію для створення бази даних

Основним оператором, з допомогою якого створюється база даних, є CREATE DATABASE. Крім того, установка деяких параметрів бази даних може бути виконана оператором

ALTER DATABASE

Видалення бази даних виконується оператором DROP DATABASE. Для знайомства з можливостями та синтаксисом цих операторів можна скористатися бібліотекою MSDN.

2.1.3. Від'єднання та приєднання бази даних

Від'єднання та приєднання бази даних використовується для перенесення бази даних між різними екземплярами MSS. Наприклад, розроблена під управлінням сервера S1 база даних може бути від'єднана від цього сервера, скопійована на який-небудь носій, відновлена на диск іншого комп'ютера і приєднана до сервера S2, що функціонує на другому комп'ютері.

Найбільший спосіб від'єднати та приєднати базу даних – це скористатися контекстними меню SMS Завдання/Від'єднати… (рис. 2.3, 2.4).

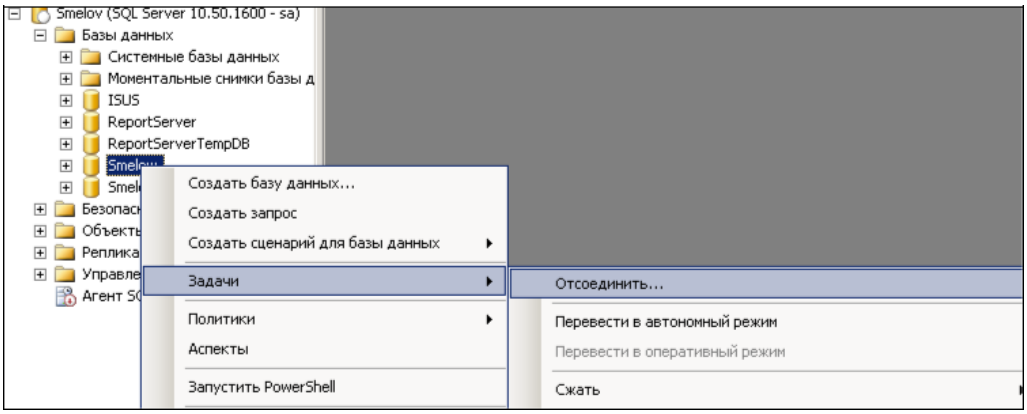


Рис. 2.3. Від'єднання бази даних

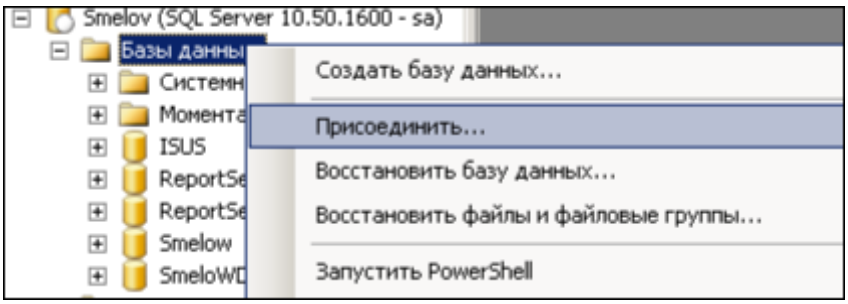


Рис. 2.4. Приєднання бази даних

Крім того, від'єднання та приєднання бази даних може бути виконано за допомогою SQL. Від'єднання бази даних

может бути виконано за допомогою системної процедури *sp_detach_db*,

а приєднання – двома способами: за допомогою системної процедури;

sp_attach_db і за допомогою оператора CREATE DATABASE з опцією FOR ATTACH.

2.2. Завдання

2.2.1. Створення бази даних за допомогою SMS

1. Створіть базу даних з ім'ям DBxxxxxx (де xxxxxx – номер залікової книжки студента), що містить крім первинної (PRIMARY) ще три файлові групи з іменами G1, G2 та G3, причому файлова група G1 містить 1 файл, G2 – 2 файли та G3 - 3 файли; сформуйте або збережіть сценарій для створення бази даних.

2. Встановіть значення параметра модель відновлення бази даних *Повна*.

3. Встановіть значення параметра автоматичне стиснення бази даних *true*.

4. Встановіть значення параметра обмеження доступу до бази даних *MULTI_USER*.

5. Кожен оператор у сформованому вище сценарії створення бази даних забезпечте коментарем, що пояснює призначення оператора.

6. Помістіть у звіт про контрольну роботу сценарій з коментарями.

7. Видаліть базу даних DBxxxxxx.

2.2.2. Створення бази даних за допомогою SQL

1. Скористайтеся сценарієм створення бази даних, сформованим у попередньому завданні.

2. Змініть значення параметра модель відновлення бази даних на неповним протоколюванням.

3. Змініть параметр автоматичного стиснення бази даних на *false*.

4. Збережіть новий сценарій і помістіть його у звіт про роботу.

5. Виконайте скоригований сценарій створення бази даних.

6. За допомогою SMS переконайтеся, що базу даних DBxxxxxx створено.

2.2.3. Від'єднання та приєднання бази даних за допомогою SMS

1. Відобразіть у звіті про контрольну роботу скріншот вікна SMS Властивості бази даних, що відображає розташування файлів бази даних DBxxxxxx, створеної в попередньому завданні.

2. Від'єднайте базу даних DBxxxxxx.

3. Перемістіть файли від'єднаної бази даних на інше місце (в інші папки жорсткого диска).

4. Приєднайте базу даних DBxxxxxx з урахуванням переміщення файлів бази даних.

5. Відобразіть у звіті про контрольну роботу скріншот вікна SMS Властивості бази даних, що відображає розташування файлів приєднаної бази даних DBxxxxxx.

Контрольні питання

1. Перерахуйте типи файлів, які використовуються базою даних, і поясніть їхнє призначення.

2. Що таке файлова група та для чого вони використовуються?

3. Навіщо використовується первинна (PRIMARY) файлова група?

4. Поясніть значення параметра *Модель* відновлення.

5. Поясніть значення параметра *ANSI NULL* за промовчанням.

6. Поясніть значення параметра Увімкнено переривання при розподілі на нуль.

7. Поясніть значення параметра *База даних доступна лише для читання.*

8. Поясніть значення *Обмеження доступу.*

9. За допомогою якого оператора SQL створюється база даних?

10. За допомогою якого оператора SQL можуть бути змінені параметри бази даних?

11. За допомогою якого оператора SQL може бути вилучена база даних?

12. Для чого застосовується від'єднання та приєднання бази даних?

13. Перерахуйте способи від'єднання бази даних.

Лабораторна робота №14

Тема: Microsoft SQL Server. Створення і видалення таблиць

1.1. Теоретичні відомості

1.1.1. Таблиці та обмеження цілісності бази даних

Основними об'єктами реляційної бази даних є таблиці. При проектуванні бази даних приймається рішення про перелік таблиць, їх структури (склад стовпців, їх типи та найменування), а також про обмеження, які накладаються на дані, що зберігаються в стовпцях таблиці. Сукупність структур таблиць та обмежень бази даних називається логічною схемою.

Сукупність обмежень на дані, які у стовпцях таблиць, називаються обмеженнями цілісності бази даних. Ім'я обмеження задається автоматично, за умовчанням, або вказується спеціально при створенні або модифікації таблиці. Назване обмеження називають констрейнтом.

MSS підтримує такі види обмежень: PRIMARY KEY, FOREIGN KEY, UNIQUE, CHECK, NULL, NOT NULL, DEFAULT. Всі обмеження задаються при створенні таблиці і в деяких випадках можуть бути скасовані або змінені при модифікації таблиць.

1.1.2. Створення та видалення таблиць за допомогою SMS

Для створення таблиці можна скористатися контекстним меню SMS (рис. 3.1), що дозволяє створити вкладку (рис. 3.2), призначену для опису властивостей стовпців таблиці та обмежень.

Для видалення існуючої таблиці також можна використовувати контекстне меню, встановивши попередньо курсор на таблицю, що видаляється. Слід пам'ятати, що якщо між таблицями встановлено відношення PRIMARY KEY / FOREIGN KEY, то видалення таблиці з PRIMARY KEY не можливо до тих пір, поки існує таблиця, що посилається на неї, з FOREIGN KEY.

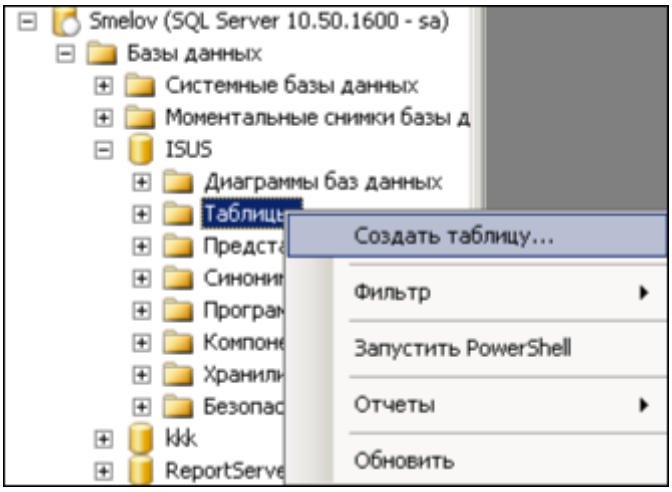


Рис. 3.1. Створення таблиць

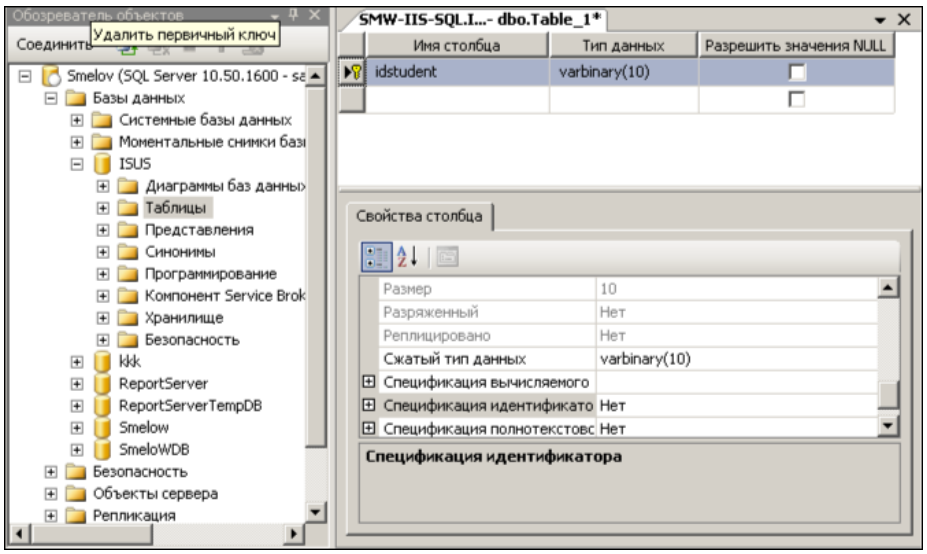


Рис. 3.2. Опис властивостей стовпців таблиці

3.1.3. Створення та видалення таблиць за допомогою SQL

Створити таблицю бази даних можна за допомогою SQL-оператора ***CREATE TABLE***.

Якщо таблиця вже існує, то сценарій для створення таблиці може бути отриманий автоматично за допомогою контекстного меню в SMS (рис. 3.3).

Аналогічно може бути побудований сценарій для видалення таблиці за допомогою оператора DROP.

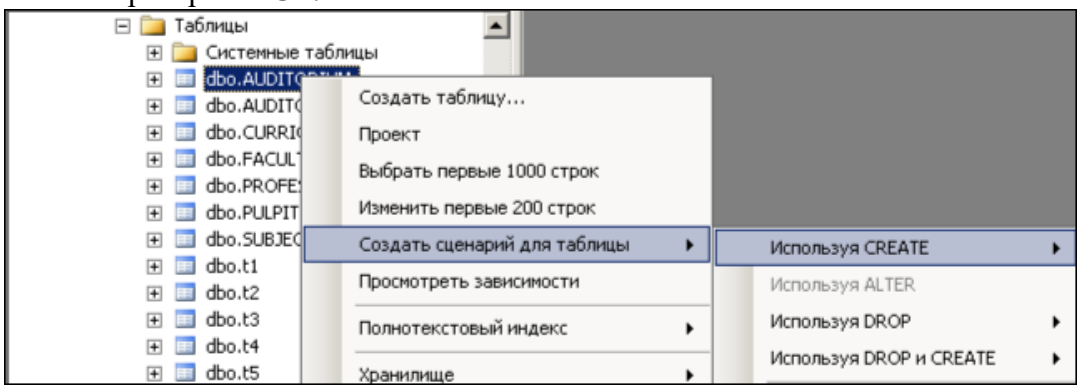


Рис. 3.3. Формування сценарію для створення таблиці

Модифікацію структури таблиці та її обмежень можна виконати за допомогою оператора ALTER TABLE. З повним синтаксисом операторів CREATE, DROP та ALTER TABLE можна ознайомитися в [5].

1.2. Завдання

1.2.1. Створення таблиць за допомогою SMS

1. Ознайомтеся з діаграмою логічної схеми бази даних, наведеної на рис. 3.4; база даних описує структуру вузу та містить інформацію про факультети (таблиця FACULTY), спеціальності, за якими проводиться навчання (PROFESSION) на факультетах, кафедрах (PULPIT), викладачах (TEACHER), що працюють на цих кафедрах, та дисциплінах (SUBJECT), закріплених за кафедрами; крім того, в базі даних міститься інформація про навчальні аудиторії вузу (AUDITORIUM), які класифікуються за типами (AUDITORIUM_TYPE).

2. Ознайомтеся з табл. 3.1, що містить опис таблиць, зображених на діаграмі (рис. 3.4); у таблиці використовуються скорочені позначення для обмежень: PK - PRIMARY KEY, FK -

FOREIGN KEY.

3. Створіть у базі даних DBxxxxxx за допомогою SMS таблиці

AUDITORIUM_TYPE та AUDITORIUM відповідно до опису в табл. 3.1; таблиці повинні бути у файловій групі G1.

1.2.2. Створення таблиць за допомогою SQL

1. Розробте сценарій для створення таблиць FACULTY, PRO-FESSION, PULPIT, TEACHER та SUBJECT відповідно до опису-

у табл. 3.1 у базі даних DBxxxxxx; таблиці FACULTY, PROFESSION, PULPIT повинні розташовуватися у файловій групі G2, таблиці TEACHER та SUBJECT – у файловій групі G3.

2. За допомогою SMS створіть діаграму бази даних DBxxxxxx та помістіть її у звіт про роботу.

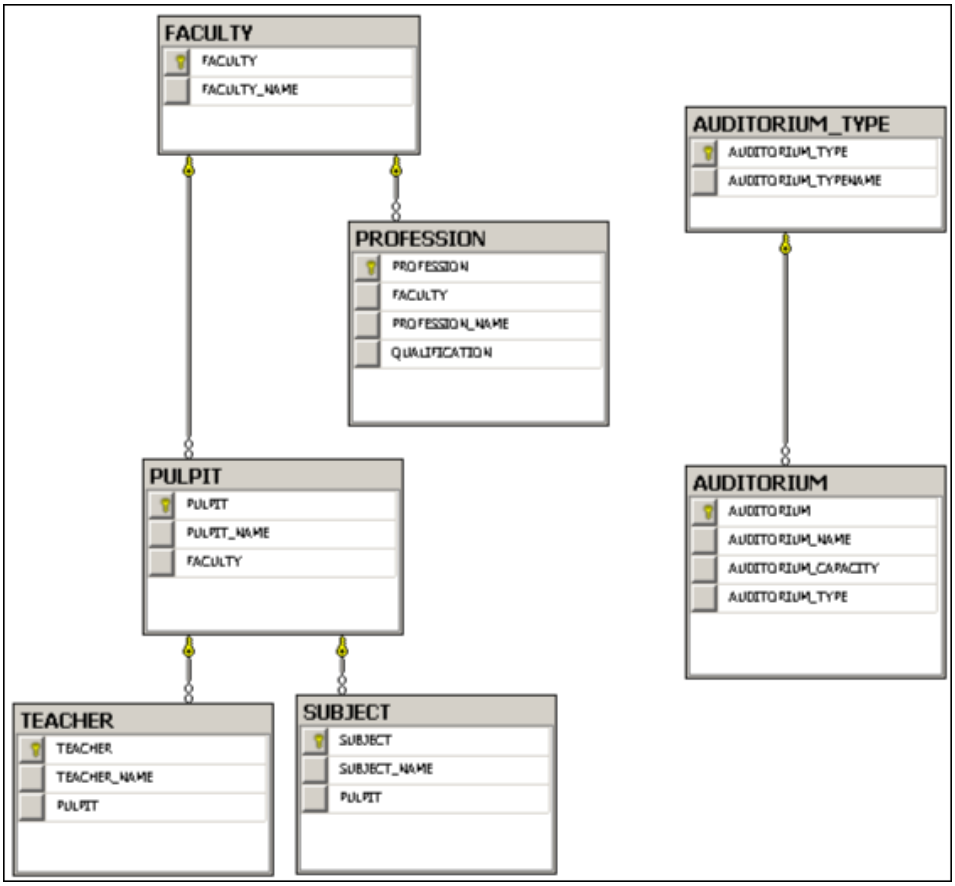


Рис. 3.4. Діаграма логічної схеми бази даних

Таблиця 3.1

Перелік таблиць та їх структура

Ім'я таблиці	Стовпці таблиці	Найменування та властивості стовпців
FACULTY	Факультети	
	FACULTY	код факультету, PK, char(10), not null
	FACULTY_NAME	найменування факультету, var-char(50), default

Ім'я таблиці	Стовпці таблиці	Найменування та властивості стовпців
PROFESSION	Спеціальності	
	PROFESSION	код спеціальності, PK, char(20), not null
	FACULTY	код факультету, FK(FACULTY), char(10), not null
	PROFESSION_NAME	найменування спеціальності, varchar(100), null
	QUALIFICATION	кваліфікація, varchar(50), null
PULPIT	Кафедри	
	PULPIT	код кафеди, PK, char(20), not null
	PULPIT_NAME	найменування кафедри, var-char(100), null
	FACULTY	код факультету, FK(FACULTY), char(10). not null
TEACHER	Викладачі	
	TEACHER	код викладача, PK,

		char(10), not null
	TEACHER_NAME	прізвище, ім'я, по-батькові препо-давача, varchar(100), null
	PULPIT	код кафедри, FK(PULPIT), char(10), not null
SUBJECT	Дисциплін и	
	SUBJECT	код дисципліни, PK, Char (10). not null
	SUBJECT_NAME	найменування дисциплі ни, varchar(100), null, unique
	PULPIT	код кафеди, FK(PULPIT), char(20), not null
AUDITORIUM_TYPE	Типи навчальних аудиторій	
	AUDITORIUM_TYPE	код типу аудиторії, PK, char(10), not null
	AUDITORIUM_TYPE-NAME	найменування типу аудиторії, varchar(30), null
AUDITORIUM	Навчальні аудиторії	
	AUDITORIUM	код аудиторії, PK, char(20), not null
	AUDITORIUM_TYPE	код типу аудиторії, FK(AUDI-TORIUM_TYPE), char(10), not null
	AUDITORIUM_CAPA-CITY	місткість аудиторії, int, de-fault 1, check between 1 and 300
	AUDITORIUM_NAME	найменування аудиторії, var-char(50), null

Контрольні питання

1. Що таке логічна схема бази даних?

2. Що таке обмеження цілісності бази даних?

3. Що таке констрейнт?

4. Перерахуйте види обмежень для стовпців та поясніть їх зміст.

5. Перерахуйте типи даних для зберігання символьної інформації, що підтримуються MSS, і поясніть принцип їх застосування.

6. Перерахуйте типи даних для зберігання числової інформації, що підтримуються MSS, та поясніть принцип їх застосування.

7. Перерахуйте типи даних для зберігання дати і часу, які підтримуються MSS, і поясніть принцип їх застосування.

8. Поясніть призначення та принцип застосування властивості INDENTITY стовпця таблиці.

9. Що таке стовпці, що обчислюються?

10. Перерахуйте оператори SQL, за допомогою яких можна створити, видалити та модифікувати таблицю бази даних.

Лабораторна робота №15

Тема: Microsoft SQL Server. Застосування SQL DML (Data Manipulation Language, Мова маніпулювання даними).

1.1. Теоретичні відомості

1.1.1. Оператор INSERT

Оператор INSERT призначений для додавання нових рядків до таблиці. На рис. 4.1 наведено приклад застосування оператора INSERT для додавання одного нового рядка в таблицю TEACHER.

```
use ISUS        -- контекст базы данных ISUS


insert into TEACHER(TEACHER, TEACHER_NAME, PULPIT)
       values('ГРБЮВ', 'Горбунова Юлия Владимировна','ИСиТ');
```

Рис. 4.1. Приклад застосування оператора INSERT

Існує інші форми запису оператора INSERT, що дозволяють одним оператором додавати до таблиці більше одного рядка. Приклад, поданий на рис. 4.2, додає таблицю TTT підмножину рядків таблиці TEACHER.

```
use ISUS        -- контекст базы данных ISUS

insert into TTT(T, P)  select TEACHER, PULPIT from  TEACHER where PULPIT = 'ИСиТ';
```

Рис. 4.2. Приклад застосування оператора INSERT
для введення більше одного рядка

1.1.2. Оператор SELECT

Оператор SELECT призначений для пошуку та вибору рядків із таблиці. Це найпотужніший і найчастіше використовуваний DML-оператор. Оператор дозволять створювати складні конструкції для пошуку даних з однієї або декількох таблиць, гнучкого вибору

безлічі рядків, групування та сортування даних, а також для виконання різних обчислень.

На рис. 4.3 наведено декілька прикладів застосування оператора SELECT.

```
use ISUS       -- контекст базы данных ISUS

select * from FACULTY                                                    --1

select PROFESSION, PROFESSION_NAME from PROFESSION order by PROFESSION   --2

select TEACHER, PULPIT from TEACHER where PULPIT = 'ИСиТ'                --3

select p.FACULTY, t.TEACHER_NAME from TEACHER t join PULPIT p            --4
                                  on t.PULPIT = p.PULPIT

select  AUDITORIUM_TYPE, sum(AUDITORIUM_CAPACITY) ss  from AUDITORIUM    --5
        group by  AUDITORIUM_TYPE

select  at.AUDITORIUM_TYPE  from AUDITORIUM_TYPE at                      --6
      where exists (select * from  AUDITORIUM aa
                          where aa.AUDITORIUM_CAPACITY > 50
                            AND aa.AUDITORIUM_TYPE = at.AUDITORIUM_TYPE)

select  AUDITORIUM_TYPE, sum(AUDITORIUM_CAPACITY)  from AUDITORIUM       --7
        group by  AUDITORIUM_TYPE
        compute  sum(sum(AUDITORIUM_CAPACITY))
```

Рис. 4.3. Приклади застосування оператора SELECT

В тих випадках, коли результати двох SELECT-запитів (два множини рядків) мають однакову структуру (однотипний набір стовпців), можна використовувати команди UNION, INTERSECT, EXCEPT для виконання операцій об'єднання, перетину та різниці між множинами рядків (рис. 4.4).

```
use ISUS

select TEACHER_NAME from TEACHER where TEACHER like 'А%' -- 1
union
select TEACHER_NAME from TEACHER where TEACHER like 'Б%'

select  PULPIT from PULPIT                                -- 2
except
select distinct PULPIT from TEACHER

select  FACULTY from PULPIT                               -- 3
intersect
select  FACULTY from PROFESSION
```

Рис. 4.4. Приклади застосування команд UNION, INTERSECT, EXCEPT

1.1.3. Оператор DELETE

Оператор DELETE призначений видалення рядків із заданої таблиці. Для вибору підмножини рядків, що видаляються, використовуються практично такі ж конструкції, як і в операторі SELECT. Слід звернути увагу: видалення рядків здійснюється тільки з однієї таблиці.

На рис. 4.5 наведено кілька прикладів застосування оператора DELETE.

```
delete TEACHER                                                          --1

delete TEACHER where  TEACHER = 'СМЛВ'                                  --2

delete  TEACHER from TEACHER t join PULPIT p on t.PULPIT  = p.PULPIT    --3
                where not exists (
                                    select * from PROFESSION f
                                           where  f.FACULTY = p.FACULTY and
                                                  f.PROFESSION = '1-40 01 02'
                                  )
```

Рис. 4.5. Приклади застосування оператора DELETE

Для видалення всіх рядків таблиці часто використовують спеціальний оператор TRUNCATE.

1.1.4. Оператор UPDATE

Оператор UPDATE призначений для зміни значень стовпців у рядках заданої таблиці. Для вибору підмножини змінюваних рядків використовуються практично такі ж конструкції, як і в операторах SELECT і DELETE.

Слід звернути увагу: зміна рядків здійснюється лише у одній таблиці.

На рис. 4.6 наведено декілька прикладів застосування оператора UPDATE.

```
update  TEACHER set  TEACHER_NAME = TEACHER +'-'+TEACHER_NAME;     --1

update TEACHER set TEACHER_NAME = 'Иванов Сергей Борисович',       --2
                   TEACHER  = 'ИВН'
       where  TEACHER = 'СМЛВ'

update TEACHER set TEACHER_NAME = TEACHER_NAME + ' +'               --3
       from TEACHER t join PULPIT p on t.PULPIT  = p.PULPIT
       where exists (
                       select * from PROFESSION f
                              where f.FACULTY = p.FACULTY and
                                    f.PROFESSION = '1-40 01 02'
                     )
```

Рис. 4.6. Приклади застосування оператора UPDATE

1.2. Завдання

1.2.1. Застосування оператора INSERT

1. Використовуючи інформацію розділу «Факультети та кафедри» сайту ТНТУ, ознайомтеся з переліком таблиць у табл. 3.1.

2. Ознайомтеся із сайтом ТНТУ.

3. Використовуючи дані сайту, заповніть за допомогою операторів INSERT таблиці FACULTY та PULPIT.

4. Використовуючи розділ «Абітурієнтам» сайту, заповніть за допомогою операторів INSERT таблицю PROFESSION.

5. Використовуючи розділи «Професорсько-викладацький склад кафедри» сайту, заповніть операторами INSERT таблиці TEACHER.

6. Використовуючи розділи «Учбова робота» сайту, заповніть за допомогою операторів INSERT таблицю SUBJECT.

7. Заповніть за допомогою операторів INSERT таблиці AUDITORIUM_TYPE та AUDITORIUM за допомогою вмісту табл. 4.1 та 4.2.

8. Наведіть приклад застосування конструкції INSERT … SE-LECT (див. рис. 4.2) для введення інформації про перелік всіх лекційних аудиторій з таблиці AUDITORIUM в іншу, попередньо створену, таблицю LK, що складається з одного стовпця з ім'ям AUD.

9. За допомогою оператора SELECT виведіть вміст усіх заповнених таблиць і помістіть у звіт про роботу.

Список типів аудиторій Таблиця 4.1

Тип аудиторії	Найменування типу аудиторії
ЛБ-Х	Хімічна лабораторія
ЛБ-СК	Спеціалізований комп'ютерний клас
ЛК	Лекційна аудиторія
ЛК-К	Лекційна аудиторія з комп'ютерами
ЛБ-К	Комп'ютерний клас

Список аудиторій Таблиця 4.2

Код	Найменування	Місткість	Тип аудиторії
103-4	103-4	80	ЛК
105-4	105-4	80	ЛК
107-4	107-4	80	ЛК
110-4	110-4	80	ЛК
111-4	111-4	50	ЛК
114-4	114-4	70	ЛК-К
131-4	131-4	20	ЛБ-К
132-4	132-4	80	ЛК
137-4	137-4	50	ЛК
200-3а	200-3а	140	ЛК
229-4	229-4	80	ЛК
236-1	236-1	80	ЛК-К
2Б-4	02Б-4	100	ЛК
301-1	301-1	20	ЛБ-СК
304-4	304-4	20	ЛБ-К

305-4	305-4	110	ЛК-К
309-1	309-1	22	ЛБ-СК
310а-1	310а-1	20	ЛБ-К
313-1	313-1	85	ЛК
314-4	314-4	80	ЛК
320-4	320-4	80	ЛК
324-1	324-1	90	ЛК
408-2	408-2	80	ЛК
410-3а	410-3а	20	ЛБ-Х
413-1	413-1	20	ЛБ-К
415-1	322-3	22	ЛБ-Х
423-1	423-1	20	ЛБ-К
429-4	429-4	80	ЛК
5-1	5-1	60	ЛК

1.2.2. Застосування оператора SELECT

1. Для виконання завдань використовуйте заповнені у попередньому завданні таблиці; Усі завдання виконуйте за допомогою оператора SELECT.

2. Отримайте список усіх факультетів університету.

3. Отримайте список всіх лекційних аудиторій з місткістю понад 80.

4. Отримайте перелік усіх спеціальностей, у назві кваліфікації яких міститься слово інженер.

5. Отримайте перелік найменувань спеціальностей, відповідних кваліфікацій, а також найменування факультету, на якому навчаються студенти даної спеціальності; перелік має бути відсортований за кодом факультету.

6. Отримайте перелік усіх дисциплін, що викладаються на факультеті ФПТ.

7. Отримайте перелік викладачів із зазначенням найменування факультету, на якому працюють ці викладачі; перелік має бути відсортований в алфавітному порядку прізвищ.

8. Отримайте перелік найменувань всіх факультетів із зазначенням кількості кафедр на кожному факультеті; перелік має бути відсортований у порядку зменшення кількості кафедр (перший рядок переліку повинен містити найменування факультету з найбільшою кількістю кафедр).

9. Отримайте перелік найменувань всіх факультетів із зазначенням кількості викладачів на кожному факультеті; перелік повинен бути відсортований у порядку зростання кількості викладачів (перший рядок переліку повинен містити найменування факультету з найменшою кількістю викладачів).

10. Обчисліть сумарну, середню, максимальну та мінімальну місткості всіх аудиторій.

11. Обчисліть сумарну, середню, максимальну та мінімальну місткості аудиторій для кожного типу аудиторій.

12. Наведіть приклад конструкції SELECT … HAVING і поясніть результат.

13. Наведіть приклад конструкції SELECT … COMPUTE та поясніть результат.

1.2.3. Застосування оператора UPDATE

1. Для виконання завдань використовуйте заповнені у попередньому завданні таблиці; Виконуйте всі завдання завдання за допомогою оператора UPDATE.

2. Змініть найменування аудиторії 415-1 так, щоб воно збігалося з кодом.

3. Збільште вміст всіх аудиторій на 10%.

4. Встановіть місткість 85 для аудиторії 324-1.

5. Встановіть найменування амфітеатру для аудиторії 200-3а.

6. Зменшіть місткість на 5 усіх лекційних аудиторій.

7. Збільште місткість на 3 лекційні аудиторії з комп'ютерами, код яких закінчується символами -4.

1.2.4. Застосування оператора DELETE

1. Для виконання завдань використовуйте заповнені у попередньому завданні таблиці; всі пункти завдання (якщо спеціально не зазначено інший спосіб) виконуйте за допомогою оператора DELETE; після виконання завдання відновіть вміст таблиць AUDITO-RIUM_TYPE та AUDITORIUM для відповідності їх таблиці. 4.1 та 4.2.

2. Видаліть інформацію про всіх аудиторіях, місткість яких перевищує 100.

3. Видаліть інформацію про всі аудиторії, місткість яких перевищує 20, але менше 60 (у секції WHERE використовуйте опе-

рацію BETWEEN).

4. Видаліть інформацію про всіх аудиторіях, місткість яких перевищує на 10% середню місткістю всіх аудиторій.

5. Видаліть за допомогою оператора TRUNCATE усі рядки таблиці LK, створеної в завданні 13; за допомогою оператора DROP видаліть таблицю LK.

Контрольні питання

1. Перерахуйте всі групи операторів SQL.

2. Розшифруйте абревіатуру DML англійською мовою та переведіть її російською.

3. Перерахуйте всі оператори, які входять до групи DML.

4. Поясніть призначення оператора INSERT.

5. Наведіть приклади всіх відомих форматів оператора

INSERT.

6. Поясніть призначення SELECT.

7. Поясніть призначення таких ключових слів.

мих в операторі SELECT: DISTINCT, TOP, FROM, WHERE, BE-

TWEEN, NOT, IS NULL, IN, ALL, ANY, AND, OR, LIKE, EXISTS, GROUP BY, CUBE, ROLLUP, HAVING, ORDER BY, COMPUTE.

8. Поясніть призначення та спосіб застосування наступних агре-

функцій: COUNT, SUM, MAX, MIN, AVG.

9. Поясніть призначення та принцип застосування конструкції

SELECT … INTO ….

10. Поясніть призначення та принцип застосування команд UNION, UNION ALL, INTERSECT та EXCEPT.

11. Поясніть призначення оператора DELETE.

12. Наведіть приклади всіх відомих форматів оператора

DELETE.

13. Поясніть призначення та принцип застосування оператора TRUNCATE та поясніть його відмінність від оператора DELETE.

Лабораторна робота №16

Тема: Microsoft SQL Server. Створення і видалення представлнень.

1.1. Теоретичні відомості

1.1.1. Загальні відомості про представлення

Представлення - це об'єкти бази даних, що представляють іменований результат виконання фіксованого SELECT-запиту. У секції FROM оператора SELECT подання можуть використовуватися так само як таблиці, тому часто представлення називають віртуальними таблицями.

В деяких випадках при роботі з представленнями допускається застосування операторів INSERT, DELETE та UPDATE, які проектуються на одну відповідну подання таблицю.

Створити, видалити та змінити подання можна за допомогою SMS або, як і будь-який інший об'єкт бази даних, за допомогою операторів CREATE, DROP, ALTER мови SQL.

1.1.2. Створення та видалення представлень за допомогою SMS

Для створення уявлення можна скористатися контекстним меню SMS (рис. 5.1), що дозволяє створити вкладку (рис. 5.2), призначену для формування SELECT-запиту.

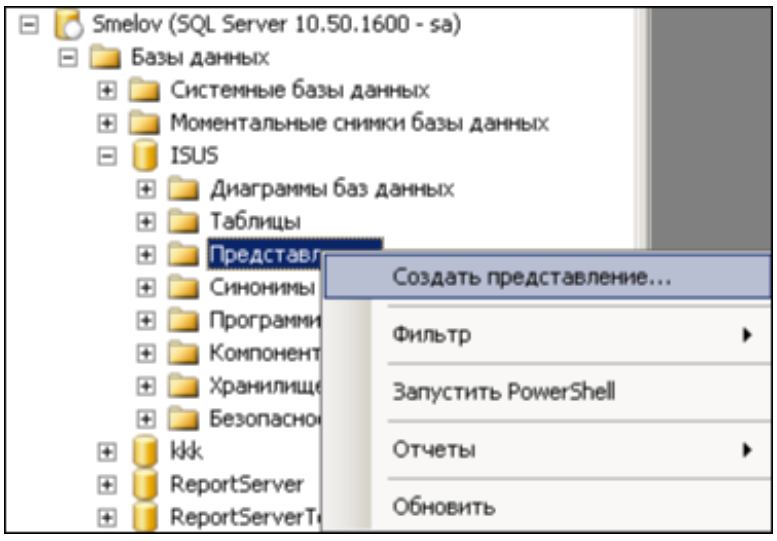


Рис. 5.1. Створення предтавлень

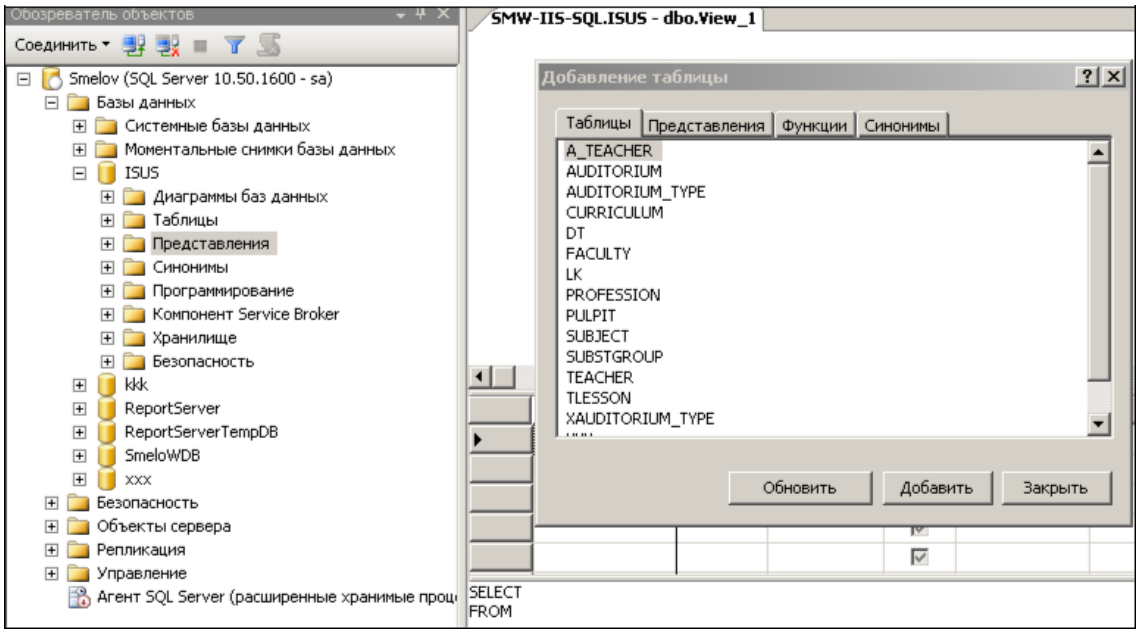


Рис. 5.2. Формування SELECT-запиту

Для видалення існуючого уявлення теж можна використовувати контекстне меню, встановивши попередньо курсор на ділянку, що видаляється.

1.1.3. Створення та видалення представлень за допомогою SQL

Створити представлення можна за допомогою SQL-оператора CREATE VIEW. Якщо уявлення вже існує, то сценарій для його створення може бути отриманий автоматично за допомогою контекстного меню в SMS (рис. 5.3).

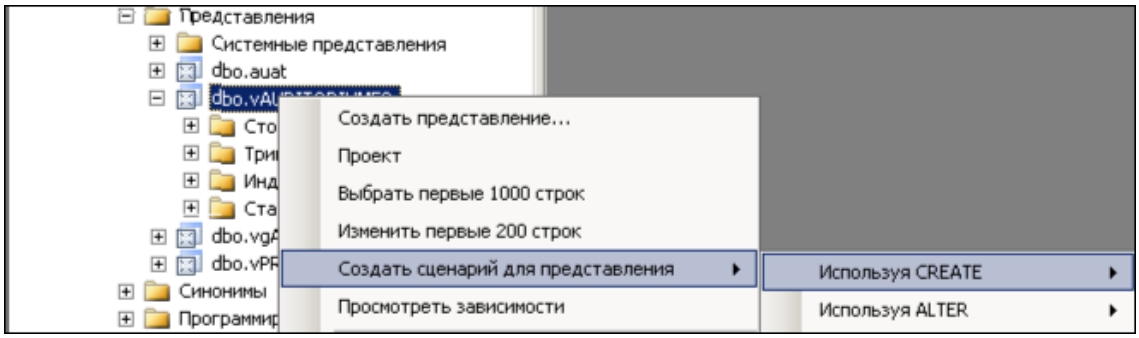


Рис. 5.3. Формування сценарію для створення уявлення

Аналогічно може бути побудований сценарій для видалення представлення за допомогою оператора *DROP* або для зміни за допомогою *ALTER*.

При створенні за допомогою конструкції WITH CHECK OPTION для представлення може бути зазначено його спеціальну властивість, яке дозволяє розглядати логічний вираз у WHERE як додаткове обмеження, що виявляється при виконанні операторів INSERT і UPDATE, що застосовуються до цього представлення.

1.2. Завдання

1.2.1. Створення представлення за допомогою SMS

1. Створіть представлення за допомогою SMS, яке відображає прізвище, ім'я та по-батькові всіх викладачів тільки кафедри АВ; перевірте працездатність подання за допомогою оператора

SELECT.

2. Створіть представлення за допомогою SMS, яке відображає всі найменування факультетів та прізвище, ім'я та по батькові викладачів, які працюють на цих факультетах.

3. Створіть представлення за допомогою SMS, яке відображає кількість викладачів, які працюють на кожному факультеті.

4. Сформуйте за допомогою SMS сценарії для створення представлень, створених у попередніх пунктах завдання.

1.2.2. Створення представлень за допомогою SQL

1. Створіть представлення за допомогою SQL, яке відображає всі лекційні аудиторії та їх місткість.

2. Створіть представлення за допомогою SQL, яке відображає середню місткість кожного типу аудиторій.

3. Розробіть самостійно представлення з властивістю WITH CHECK OPTION та допускає виконання операторів INSERT та UPDATE, продемонструйте його працездатність та дію властивості WITH CHECK OPTION.

4. Розробіть самостійно представлення, SELECT-запит якого використовує інше представлення.

1.2.3. Застосування представлень

1. Продемонструйте найпростіший *SELECT*-запит до представлення, розробленого в попередніх завданнях.

2. Розробіть *SELECT*-запит до представлення.

3. Розробіть *SELECT*-запит, що містить секції *WHERE*

та ORDER BY, до розробленого в попередніх завданнях представленнях.

4. Розробіть *SELECT*-запит, що містить секцію GROUP BY,

до розробленого у попередніх завданнях представленні.

5. Розробіть *DELETE*-запит, що містить підзапит до розробленого в попередніх завданнях представленнях.

6. Розробіть *UDATE*-запит, що містить підзапит до розробленого в попередніх завданнях представленнях.

Контрольні питання

1. Дайте визначення представленню.

2. Поясніть, у яких випадках доцільно використовувати представлення.

3. Перерахуйте оператори SQL, за допомогою яких представлення створюються, видаляються і змінюються.

4. Перерахуйте способи створення представлень.

5. Перерахуйте існуючі обмеження при створенні представлень.

6. Поясніть призначення та принцип дії секції WITH CHECK OPTION.

7. Перерахуйте правила створення представлень, що допускають виконання операцій INSERT, DELETE, UPDATE.

Лабораторна робота №17

Тема: Microsoft SQL Server. Створення та видалення індексів.

1.1. Теоретичні відомості

1.1.1. Загальні відомості про індекси

Індекс - це об'єкт бази даних, призначений для прискорення запитів до даних у таблиці бази даних.

Створити, видалити або змінити індекс можна за допомогою SMS або операторами CREATE, DROP, ALTER мови SQL.

При створенні індексу вказується один або кілька стовпців таблиці, за значеннями яких буде побудовано та підтримувати індекс. Індекс є структурою пам'яті, організовану у вигляді збалансованого дерева. У вузлах дерева містяться сторінки зі значеннями вибраних стовпців. SELECT-запити, що використовують у секції WHERE стовпці таблиці, для яких побудований індекс, не вимагають сканування всієї таблиці, так як індекс дозволяє отримати покажчики на всі рядки, що запитуються, за невелику кількість операцій читання. Крім того, індекси, як правило, мають значно менші розміри, ніж таблиці, що в більшості випадків дозволяє їх розміщувати в оперативній пам'яті сервера. Використання індексів є прозорим для програміста, який пише запити. Він не має можливості вказати серверу, який індекс слід використовувати у запиті. Питання застосування індексу для вибірки даних вирішується спеціальною компонентою сервера, що називається оптимізатор запитів. Програміст може лише з'ясувати порядок вибірки даних, отримавши план виконання запиту.

Під час створення таблиці, що містить стовпці, які мають властивості PRIMARY KEY або UNIQUE, індекси створюються автоматично. Крім того, індекси можуть бути унікальними та не унікальними. Унікальний індекс діє для стовпця або групи стовпців таблиці як обмеження цілісності UNIQUE.

Слід зазначити, що, крім «класичних» типів індексів MSS, підтримує ще ряд специфічних типів: XML-індекси, SPATIAL-індекси, FULLTEXT-індекси.

1.1.2. Кластеризовані та некластеризовані індекси

MSS підтримує два типи індексів: кластеризовані та не-кластеризовані індекси.

При створенні кластеризованого індексу дані таблиці, що індексується, розташовуються у фізичному порядку, відповідному індексу, і стають частиною кластеризованого індексу. Тому класифікований індекс для таблиці може бути створений лише один.

Некластеризований індекс - це окремий об'єкт, що має покажчики на рядки таблиці. Максимальна кількість некластерованих індексів для однієї таблиці не повинна перевищувати 1000.

1.1.3. Створення та видалення індексів за допомогою SSMS

Для створення індексів за допомогою SSMS слід скористатися контекстним меню, як це показано на рис. 6.1.

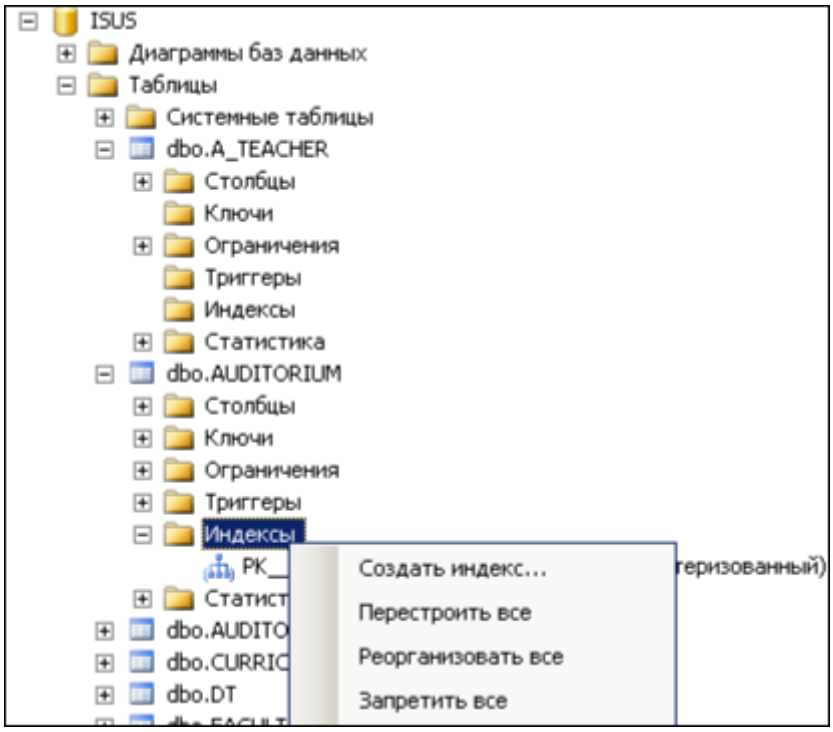


Рис. 6.1. Створення індексу

Видалення індексу здійснюється так само.

1.1.4. Створення та видалення індексів за допомогою SQL

Як і будь-який об'єкт бази даних індекс може бути створений за допомогою оператора CREATE, а видалений за допомогою оператора DROP.

Для існуючого індексу за допомогою SSMS досить легко можна отримати готові скрипти SQL для створення та видалення цього індексу. На рис. 6.2 представлений приклад застосування контекстного меню SSMS для формування скрипту, що створює зазначений індекс.

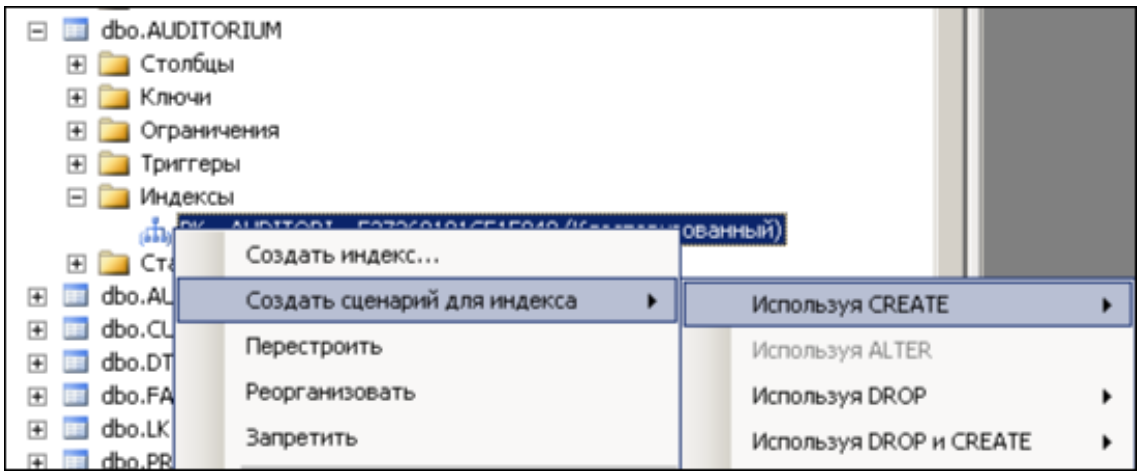


Рис. 6.2. Формування скрипта для створення індексу

1.1.5. Перебудова та реорганізація індексів

Якщо в проіндексованій таблиці здійснюється велика кількість змін (INSERT, UPDATE, DELETE), то на сторінках індексу залишаються невикористані фрагменти при

повному заповненні самої сторінки. Це називається фрагментацією індексів. Справа в тому, що пам'ять, виділена в сторінці індексу, не звільняється в оперативному режимі, так як це для сервера занадто витратна операція, яка може нівелювати весь ефект від застосування індексу. Фрагментація призводить до того, що індекс збільшується в обсязі і це в кінцевому підсумку призводить до зниження продуктивності сервера.

Така організація індексів призводить до того, що з таблиць, інформація у яких часто змінюється, потрібна періодична перебудова індексів.

Для перебудови індексів можна скористатися SSMS, виконавши пункти «Перебудувати» або «Реорганізувати» в контекстному меню (рис. 6.2) або виконавши оператори.

Аналогічного ефекту можна досягти за допомогою операторів ALTER INDEX … REBUILD та ALTER INDEX … REORGANIZE.

Основна відмінність перебудови індексу від його реорганізації полягає в тому, що в першому випадку індекс перестворюється повністю. Такого ж ефекту можна досягти видаленням (DROP) і створювати (CREATE) цього індексу. Перебудова повністю знищує фрагментацію в індексі. У разі реорганізації здійснюється перебудова тільки нижнього рівня індексу, що дозволяє здебільшого істотно знизити фрагментацію. При цьому час витрачається на реорганізацію індексу значно менше, ніж його перебудову.

Перебудова індексу може виконуватись у оперативному режимі. За умовчанням на час виконання цієї операції сервер заблокує таблицю, асоційовану з індексом, що призведе до того, що таблиця перестане бути доступною користувачам. Щоб таблиця була доступна, слід оператору ALTER INDEX … REBUILD вказати ключове слово ONLINE.

Реорганізація індексу завжди виконується в оперативному режимі і не викликає довгострокового блокування таблиці.

1.2. Завдання

1.2.1. Робота з індексами за допомогою SSMS

1. За допомогою SSMS визначте, випишіть та вкажіть у звіті всі індекси у базі даних ISUS, створені автоматично.

2. Створіть за допомогою SSMS унікальний некластеризований індекс з ім'ям IX_UNIQUE_TEACHER_NAME для стовпця

TEACHER_NAME таблиці TEACHER.

3. Продемонструйте, що створення індексу IX_UNIQUE_ TEACHER_NAME аналогічне до створення обмеження цілісності

UNIQUEдля колонки TEACHER_NAME таблиці TEACHER.

4. Створіть за допомогою SSMS унікальний некластеризований індекс з ім'ям IX_SUBJECT_NAME для стовпця SUBJET_NAME таблиці SUBJECT.

5. За допомогою SSMS виконайте реорганізацію індексу IX_UNI-

QUE_TEACHER_NAME.

6. За допомогою SSMS перебудуйте індекс IX_SUB-JECT_NAME.

7. За допомогою SSMS видаліть індекс IX_SUBJECT_NAME.

1.2.2. Робота з індексами за допомогою SQL

1. За допомогою SSMS створіть скрипт для видалення індексу IX_UNIQUE_TEACHER_NAME.
2. За допомогою SSMS створіть скрипт для створення індексу IX_UNIQUE_TEACHER_NAME.
3. Послідовно виконайте скрипти на видалення та створення індексу IX_UNIQUE_TEACHER_NAME, переконайтеся, що він створився.
4. Розробіть скрипт на SQL, який створює індекс IX_SUB-JECT_NAME, аналогічний створюваному в попередньому завданні.
5. Розробіть скрипт на SQL, що реорганізує індекс IX_UNIQUE_TEACHER_NAME та перебудовуючий індекс IX_UNIQUE_TEACHER_NAME.

Контрольні питання

1. Дайте визначення індексу та поясніть його призначення.
2. Назвіть усі типи індексів, які підтримуються MSS.
3. Чим відрізняється кластеризований індекс від некластеризованого?
4. У яких випадках індекси автоматично створюються?
5. Що таке фрагментація індексів, у яких випадках вона виникає?
6. Чим відрізняються процедури перебудови та реорганізації індексів?
7. Навіщо застосовується ключове слово ONLINE в операторі перебудови індексу?

Лабораторна робота №18

Тема: Microsoft SQL Server. Застосування процедур, функцій та тригерів.

1.1. Теоретичні відомості

1.1.1. Збережені процедури, їх створення, видалення та застосування

Програми на мові SQL можуть бути об'ємними. Крім того, часто одні й ті ж фрагменти коду зручно використовувати одній або різних програмах. Для структуризації коду, а також для зберігання повторно використовуваного коду в мові SQL застосовуються процедури та функції.

Процедура – це об'єкт бази даних, що є іменованим фрагментом коду SQL, який можна виконувати в програмі SQL за допомогою команди EXECUTE. Процедура може мати параметри, значення яких встановлює програма, що викликає. Параметри можуть бути вхідні та вихідні. При виклику процедури параметри можуть задаватися в позиційному або параметричному вигляді. Процедура може повертати числове значення типу int, у разі процедура повинна закінчуватися командою RETURN. Крім того, процедура може формувати результуючий набір, що обробляється в коді, що викликає. Процедура створюється за допомогою оператора CREATE, видаляється за допомогою оператора DROP, а також може бути змінена оператором ALTER. Слід зазначити, що під час створення процедури можуть бути зазначені додаткові властивості,

На рис. 9.1 продемонстровано створення найпростішої процедури, що має два параметри, а на рис. 9.2 – виклик цієї процедури двома способами: з позиційною та параметричною передачею параметрів.

```
create procedure CFAC @s varchar(20), @c int output
as begin
 set @c = (select COUNT(*) from  FACULTY where upper (FACULTY_NAME) like  upper('%'+@s+'%'));
 if @c > 0 select FACULTY  from  FACULTY where upper (FACULTY_NAME) like  upper('%'+@s+'%');
 return 1;
end;
```

Рис. 9.1. Приклад створення процедури

```
declare @rc int, @cc int;
execute @rc = CFAC 'лес', @cc output
print @rc;
print @cc;
execute @rc = CFAC @c = @cc out, @s = 'лес'
print @rc;
print @cc;
```

Рис. 9.2. Виклики процедури

1.1.2. Функції, їх створення, видалення та застосування

Функція – це об'єкт бази даних, що є іменованим фрагментом коду SQL, який можна використовувати у виразах мови SQL.

Головна відмінність функцій від процедур полягає в тому, що функція може повертати будь-які за малим винятком типи значень. У функції заборонено виконання будь-яких змін у базі даних, параметри можуть бути тільки вхідні, а при виклике використовується тільки позиційний формат передачі даних; крім того, функція не може формувати результуючий набір, але може повертати результат SELECT-запиту або сформовану таблицю.

Розрізняють три види функцій користувача: скалярні (повертають єдине значення), табличні (повертають результат SELECT-запиту) і табличні багатооператорні (повертають сформовану в пам'яті таблицю). Функція створюється за допомогою оператора CREATE, видаляється за допомогою оператора DROP, також може бути змінена оператором ALTER.

На рис. 9.3 продемонстровано створення найпростішої табличної функції, має один параметр, але в рис. 9.4 – виклик цієї функції.

```
create function FFAC(@s varchar(20)) returns table
as  return select FACULTY
          from  FACULTY
          where upper (FACULTY_NAME) like  upper('%'+@s+'%');
```

Рис. 9.3. Приклад створення табличної функції

```
select * from FFAC('лес')
```

Рис. 9.4. Виклик табличної функції

1.2. Завдання

1.2.1. Створення та видалення збережених процедур

1. Розробіть процедуру, що зберігається, що приймає два параметри (один вхідний і один вихідний), формує код повернення, значення вихідного параметра, а також результуючий набір.

2. Продемонструйте виклик розробленої процедури з позиціонною передачею параметрів.

3. Продемонструйте виклик процедури з параметричної передачі параметрів.

4. Видаліть процедуру.

1.2.2. Створення та видалення функцій

1. Розробіть скалярну функцію без настройок.

2. Розробіть табличну функцію з одним параметром.

3. Розробіть багатооператорну табличну функцію із двома параметрами.

4. Розробіть програму на SQL, що викликає розроблені функції.

Контрольні питання

1. Дайте визначення процедурі, що зберігається.

2. Який тип значень може повертати процедуру?

3. Які види параметрів можна використовувати в процедурі?

4. Які способи передачі параметрів допускаються під час виклику процедури?

5. Дайте визначення функції користувача.

6. Перерахуйте відмінності функцій від процедур, що зберігаються.

7. Перерахуйте види функцій та поясніть їх особливості.

2 ТРИГЕРИ ТА ЇХ ЗАСТОСУВАННЯ

2.1. Теоретичні відомості

2.1.1. Поняття тригера, типи тригерів

Тригер – це особливий вид збережених процедур, викликаних сервером у разі певних подій у базі даних. Розрізняють два види тригерів: системні та DML-тригери.

Системні тригери призначені для реагування на події сервера.

Кожен DML-тригер закріплюється за певною таблицею або поданням і викликається на події, пов'язані з додаванням (INSERT), видаленням (DELETE) або зміною (UPDATE) рядків таблиць. MSS підтримує два види тригерів: AFTER та INSTEAD OF.

До однієї таблиці допускається прикріплення кількох тригерів.

Тригери AFTER спрацьовують після настання події в асоційованій таблиці або поданні.

Тригери INSTEAD OF спрацьовують замість події, а сама подія ніколи не настає.

При виконанні тригера MSS забезпечує створення та заповнення двох таблиць INSERTED і DELETED, доступних у коді триггера. Таблиці дозволяють отримати віддалені, змінені (до зміни та після), а також додані рядки у відповідній події.

2.1.2. Створення та видалення тригерів

Створення тригерів здійснюється за допомогою оператора CREATE, видалення за допомогою оператора DROP.

На рис. 10.1 наведено приклад створення AFTER-тригера, який за наявності рядків у таблицях INSERTED і DELETED визначає тип події, що обробляється.

На рис. 10.2 наведено приклад створення INSTEAD OF-тригера, аналогічний за призначенням тригеру, представленому на рис. 10.1.

```
create trigger ITR_SUBJECT on dbo.SUBJECT
instead of  insert, update, delete
as
 if exists (select * from inserted)and
    exists (select * from deleted)
   print 'Update';
 else if  exists (select * from inserted)and
        not exists (select * from deleted)
   print 'Insert';
 else if  not exists (select * from inserted)and
         exists (select * from deleted)
    print 'Delete';
 else   print 'Unknowm';
```

Рис. 10.1. Приклад створення AFTER-тригера

```
create trigger TR_SUBJECT on dbo.SUBJECT
for insert, update, delete
as
 if exists (select * from inserted)and
    exists (select * from deleted)
   print 'Update';
 else if  exists (select * from inserted)and
        not exists (select * from deleted)
   print 'Insert';
 else if  not exists (select * from inserted)and
         exists (select * from deleted)
    print 'Delete';
```

Рис. 10.2. Приклад створення INSTEAD OF-тригера

2.2. Завдання

2.2.1. Застосування AFTER-тригерів

1. Повторіть приклад тригера на рис. 10.1.
2. Додати, змініть та видаліть рядок у таблиці SUBJECT та поясніть результат.
3. Змініть тригер так, щоб у ньому виводився вміст непорожніх таблиць INSERTED і DELETED.
4. Додати, змініть та видаліть рядок у таблиці SUBJECT та поясніть результат.

2.2.2. Застосування INSTEAD OF-тригерів

1. Повторіть приклад тригера на рис. 10.2.
2. Додати, змініть та видаліть рядок у таблиці SUBJECT та поясніть результат.
3. Змініть тригер так, щоб у ньому виводився вміст непорожніх таблиць INSERTED і DELETED.
4. Додати, змініть та видаліть рядок у таблиці SUBJECT та поясніть результат.

Контрольні питання

1. Дайте визначення тригеру.

2. Перелічіть типи тригерів, які підтримуються MSS.

3. Перерахуйте види DLL-тригерів, які підтримуються MSS.

4. Поясніть різницю між типами DLL-тригерів.

5. Поясніть призначення та принцип заповнення таблиць INSERTED та DELETED.

Список літературних джерел.

1. Александр Шпортько, Леся Шпортько. Розробка баз даних в СУБД Microsoft Access / Кондор, 2018.- 184с.

2. Балик Н.Р. Мандзюк В.І. Бази даних MySQL: Навчальний посібник. — Тернопіль: Навчальна книга – Богдан, 2010.— 160 с.

3. Куліків С. Реляційні бази даних у прикладах. Практичний посібник для програмістів і тестувальників. / Кондор, 2019. – 175с.

4. Гайдаржи В.І., Ізварін І.В. Бази даних в інформаційних системах / В-во: Університет "Україна", 2018 – 418с.

5. Булатецька Л В. Мова запитів SQL : текст лекцій нормативної навчальної дисципліни "Бази даних та розподілені інформаційно-аналітичні системи" / Булатецька Леся Віталіївна, Булатецький Віталій Вікторович. – Луцьк : СНУ імені Лесі Українки, 2018. – 92

6. Рамський Ю.С., Цибко Г.Ю.Проектування й опрацювання баз даних: Посібник для вчителів. / Богдан, 2010.-116 с.

7. https://www.embarcadero.com/products/delphi

Зміст