МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
Факультет комп’ютерних інформаційних технологій

Кафедра кібербезпеки

МЕТОДИЧНІ ВКАЗІВКИ
ДО ВИКОНАННЯ ЛАБОРАТОРНИХ РОБІТ
З ДИСЦИПЛІНИ

"ПРОЕКТУВАННЯ ТА ЗАХИСТ БАЗ ДАНИХ"

для студентів денної форми навчання
галузі знань 12 Інформаційні технології
спеціальності 125 Кібербезпека
(освітньо-кваліфікаційний рівень «бакалавр»)

Тернопіль 2022

Методичні рекомендації до виконання лабораторних робіт з дисципліни «Проектування та захист баз даних» для студентів денної форми навчання галузі знань галузі знань 12 Інформаційні технології 125 Кібербезпека (освітньо-кваліфікаційний рівень «бакалавр»), Укладач: Івасьєв С.В. Тернопіль, ФОП «Шпак». 2022, 70 с.

Методичні рекомендації до виконання лабораторних робіт з дисципліни «Проектування та захист баз даних» та навчальних планів для студентів вищих навчальних закладів. Методичні рекомендації містять мету та завдання дисципліни, методичні рекомендації до виконання лабораторних робіт, рекомендовану літературу та інші методичні матеріали.

Укладачі: Івасьєв Степан Володимирович, к. т. н., доцент, доцент кафедри кібербезпеки ЗУНУ

Рецензенти:

Трембач Р.Б.. к.т.н., доцент, кафедри автоматизації технологічних процесів та виробництв Тернопільського національного технічного університету імені Івана Пулюя.

Батько Ю.М. к.т.н., доцент кафедри комп'ютерної інженерії Західноукраїнського національного університету

Відповідальний за випуск: Яцків В. В. д.т.н., професор, завідувач кафедри кібербезпеки ЗУНУ.

Рекомендовано на засіданні кафедри кібербезпеки (Протокол № 8 від 20 квітня 2022 р.)

Рекомендовано групою забезпечення спеціальності «Кібербезпека» (Протокол № 4 від від 20 квітня 2022 р.)

ВСТУП

Методичні вказівки до виконання лабораторних робіт призначені для підготовки та виконання лабораторних робіт з дисципліни “ Проектування та захист баз даних”, викладання якої забезпечує кафедра Кібербезпеки.

У методичних вказівках описано 5 лабораторних робіт, які входять до складу програми із вказаної дисципліни. У ньому наведено теоретичні відомості про моделювання предметної області, основні принципи організації та проектування баз даних, структури та моделі даних, системи керування базами даних. Наявність теоретичного матеріалу обумовлена тим, що практично складно забезпечити фронтальний метод виконання лабораторних робіт, крім того, не завжди можливо синхронізувати в часі лекційні й лабораторні заняття. Наведено послідовність досліджень, які проводяться під час лабораторної роботи, вимоги щодо опрацювання результатів, а також контрольні запитання, які орієнтують студента в конкретному напрямі досліджень. Вміщено також перелік рекомендованої літератури для підготовки кожної лабораторної роботи.

ЛАБОРАТОРНА РОБОТА 1.

ТЕМА: ПРОЕКТУВАННЯ БАЗИ ДАНИХ

МЕТА: НАВЧИТИСЬ ВИКОРИСТОВУВАТИ ОСНОВНІ ПРИЙОМИ ПРОЕКТУВАННЯ БД.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Дані - це безліч інформаційних об'єктів, кожен з яких має свої властивості та поведінку, а також зв'язків між цими об'єктами.

База даних - це набір взаємозалежних даних, що відображають інформацію про певну предмети)' область. Бази даних призначені для зберігання, накопичення, оновлення та пошуку необхідної інформації.

Інформаційна область, для якої створюється база даних, називається ***предметною областю***.(Наприклад, шкільна база даних: школа - це предметна область).

Об'єкти бази даних - це дані, що в ній використовуються. СУБД - це комп'ютерна програма, ідо дає змогу описувати дані у вигляді об'єктів і зв'язків, маніпулювати ними і має зручний інтерфейс. Є такі моделі даних для опису предметної області:

Інфологічна модель даних - це опис предметної області, виконаний природною мовою, за допомогою математичних формул, графіків, таблиць.

Датологічна модель даних - це опис предметної області, виконаний мовою обраної системи управління базами даних.

Створення інфологічної моделі полягає:

- у визначенні числа і структури таблиць
- формуванні запитів до бази даних
- визначенні типів звітних документів
- розробці алгоритмів обробки інформації
- розробці форм для введення і редагування даних у базі даних

Є такі етапи розробки бази даних:

1. Концептуальне проектування бази даних - це розуміння того, які дані мають бути взяті з предметної області для представлення їх в базі даних і як вони взаємопов’язані. Результатом цього етапу є побудова концептуальної моделі БД, тобто подання предметної області у вигляді інформаційних об’єктів і їх зв’язків.

При концептуальному проектуванні БД використовується модель «сутність - зв’язок».

Сутність - це об’єкт предметної області, що є множиною елементів. Наприклад у БД «Школа» прикладами сутності є учні, предмети. Сутності подаються у базі даних як таблиці (для зберігання інформації). Ім’я сутності - це назва таблиці, характеристики - назви стовпців таблиці, а екземпляри - рядки таблиці.

Зв’язки відображають важливі для проектування бази даних відносини між сутностями. Зв'язок між сутностями можна відобразити у вигляді ліній між окремими екземплярами.

2. Логічне проектування бази даних. На цьому етапі сутності і зв’язки перетворюються на логічну модель даних, побудовану за законами логіки. Існує кілька логічних моделей даних; реляційна, ієрархічна, мережна.

Реляційна модель даних базується на створенні відношень і зв’язків. Відношення подається у вигляді таблиці, що складається з рядків і стовпців.

Кожен стовбець відношення називається полем, а кожен рядок - кортежем (записом). Назви полів - атрибути. Основна властивість відношення полягає в тому, що в ньому не повинно бути одинакових записів.

Основні елементи реляційної моделі даних:

Атрибут - заголовок стовпця таблиці
Відношення - таблиця
Домен - стовбець таблиці
Кортеж - рядок таблиці
Первинний ключ - один або кілька атрибутів
Схема відношення - рядок заголовків таблиці
Тип даних - тип значень елементів таблиці

На концептуальному рівні здійснюється інтегрований опис предметної області, для якої розробляється БД, незалежно від її сприйняття окремими користувачами та способів реалізації в комп'ютерній системі. Дамо означення основних понять, що використовуються на концептуальному рівні.

Предметна область (ПО) — частина реального світу, для якої здійснюється концептуальне моделювання.

Концептуальна модель ПО — формальне зображення сукупності думок, які характеризують можливі стани ПО, а також переходи з одного стану в інший (включно з класифікацією наявних у ПО сутностей, чинних правил, законів, обмежень тощо).

Концептуальне моделювання ПО — процес побудови концептуальної моделі ПО, яка б відображувала ПО з урахуванням вимог, висунутих до цього процесу.

Концептуальна схема — фіксація концептуальної моделі ПО засобами конкретних мов моделей даних. У СКБД концептуальна модель подається у вигляді концептуальної схеми.

Опишемо властивості концептуальної моделі (схеми) й характерні особливості концептуального моделювання. Спільне та однозначне тлумачення предметної області всіма зацікавленими особами. До розробки складної бази даних залучається великий колектив: експерти, системні аналітики, проектувальники, розробники, ті, хто займається впровадженням і супроводом. Усі вони повинні однозначно розуміти, чим є ПО, в чому зміст використаних понять, як вони взаємопов'язані між собою, які обмеження висуваються до моделі ПО тощо. Спільність понять має забезпечувати концептуальна модель.

- Концептуальна схема відображує лише концептуально важливі аспекти ПО, виключаючи будь-які аспекти зовнішнього або внутрішнього відображення даних. Ця модель не повинна відображувати конкретні потреби окремих користувачів або застосувань. Вона має фіксувати, чим є ПО в цілому, а не з точки зору інтересів або потреб користувачів. Для отримання цілісного уявлення про ПО її модель має інтегрувати думки, погляди та інтереси окремих користувачів, але саме інтегрувати, а не виражати їхні конкретні побажання.

- Визначення допустимих меж еволюції бази даних. У процесі експлуатації база даних може розвиватися, проте цей розвиток може відбуватися тільки в межах, допустимих для концептуальної схеми.
- Відображення зовнішніх схем на внутрішню. Саме через концептуальну схему зовнішні дані відображуються на внутрішні, й навпаки. У такий спосіб створюється єдина основа для опису даних і підтримки цих відображень.
- Забезпечення незалежності даних. Наявність відображень концептуальний-зовнішній і концептуальний-внутрішній дає змогу вирішувати проблему логічної та фізичної незалежності даних. Будь-які зміни в тій чи іншій зовнішній моделі не повинні спричиняти зміни в концептуальній або внутрішній моделях. У цьому випадку має змінитися тільки відповідне відображення «кон-цептуальний-зовнішній». Аналогічно, будь-які зміни у внутрішній моделі не зачіпають концептуальну модель і моделі зовнішнього рівня, а тільки приводять до змін відображення «концептуальний-внутрішній».
- Централізоване адміністрування. Саме через концептуальну схему здійснюється адміністрування баз даних.
- Стійкість. Концептуальна схема не має підладжуватися до вимог тих чи інших користувачів (зовнішній рівень) або до вимог зберігання даних (внутрішній рівень). Будучи моделлю ПО, вона має змінюватися тільки тоді, коли входить у суперечність із нею.

Існує багато мов, які претендують на роль мов концептуального моделювання ПО. Найпопулярнішими і широковживаними є мови, що належать до класу так званих графічних мов, які оперують поняттями «сутність-атрибут-зв'язок» (Entity-Relationship language).

Порядок виконання роботи

1. Обрати предметну область, в якій виникає необхідність створення бази даних.
2. Ознайомитись з концептуальним рівнем моделювання предметної області.
3. Визначити основні сутності предметної області.
4. Визначити атрибути сутностей.
5. Структурувати інформацію про сутності та відповідні їм атрибути.
6. Зробити висновки, що до отриманих результатів, визначити основні етапи і принципи отримання інформації, охарактеризувати предметну область в термінах сутностей та атрибутів.

4\. Дати назви сутностям та їхнім атрибутам (ідентифікатори іменників).
3\. Встановити зв’язки, якими зв’язані сутності (дієслова).
5\. Розробити діаграми ER-екземплярів для першого встановленого зв’язку (типу «автор пише книгу»).
6\. Розробити відповідні ER-діаграми для всіх зв’язків (названих дієсловами).

7. Встановити клас приналежності одної сутності бінарного зв'язку (тобто, встановити значення «1», або «n»).
8. Встановити клас приналежності другої сутності бінарного зв'язку (тобто, встановити значення «1», або «n»).
9. Розробити загальну ER-діаграму для всіх зв'язків і встановити кратності їх відношень.

Контрольні питання:

1. Що таке атрибут
2. Що таке таблиця
3. Що таке відношення
4. Що таке домен
5. Що таке кортеж
6. Що таке первинний ключ
7. Схема відношення –
8. Тип даних
9. Інфологічне проектування
10. Даталогічне проектування
11. Етапи проектування БД
12. Види відношень

Література

1. Тарасов, О. В. Організація баз даних та знань. Проектування баз даних [Текст] : навч.-практ. посіб. Ч. 1 / О. В. Тарасов, В. В. Федько, М. Ю. Лосєв. – Х. : ХНЕУ, 2011. – 200 с. – Режим доступу : http://library.tneu.edu.ua/images/stories/zmist/2012/літо/організація баз даних та знань тарасов 2011.pdf.
2. Барміна, В. А. Економічна інформатика. СУБД Access [Текст] : опорн. консп. лекцій / В. А. Барміна, М. О. Цензура. – К. : КНТЕУ, 2010. – 156 с. – Режим доступу : http://library.tneu.edu.ua/images/stories/zmist/2012/літе/економічна інформатика барміна 2010.pdf.
3. Мінухін, С. В. Методи і моделі проектування на основі сучасних CASE-засобів [Текст] : навч. посіб. / С. В. Мінухін, О. М. Беседовський, С. В. Знахур. – Х. : ХНЕУ, 2008. – 272 с. – Режим доступу : http://library.tneu.edu.ua/images/stories/zmist/2012/літм/методи і моделі проектування мінухін 2008.pdf.
4. Степанов, В. П. Публикация баз данных в Интернете [Текст] : учеб. пособ. / В. П. Степанов, И. В. Купрейчик, О. В. Черкашина. – Х. : Изд. ХНЭУ, 2008. – 300 с . – Режим доступу : http://library.tneu.edu.ua/images/stories/zmist/2012/літп/публикация баз даних степанов 2008.pdf.
5. Перевозчикова, О. Л. Інформаційні системи і структури даних [Текст] / О. Л. Перевозчикова. – К. : Києво-Могилянська академія, 2007. – 287 с. – Режим доступу :

http://library.tneu.edu.ua/images/stories/zmist/2012/літі/інформаційні системи перевозчикова 2007.pdf.

6. Karrano, M. Data abstraction and problem solvins with C++ [Текст] / M. Karrano, I. Prichard. – Boston : INS, 2001. – 118 p.

ЛАБОРАТОРНА РОБОТА 2.
ТЕМА: СТВОРЕННЯ ЕЛЕМЕНТАРНИХ ЗАПИТІВ, А ТАКОЖ ЗАПИТІВ ДЛЯ ЗАПИСУ, РЕДАГУВАННЯ ДАНИХ.
МЕТА: НАБУТИ ПРАКТИЧНИХ НАВИКІВ ВИКОРИСТАННЯ ЗАПИТІВ МАНІПУЛЯЦІЇ ДАНИМИ МОВИ SQL

ТЕОРЕТИЧНІ ВІДОМОСТІ

SQL (англ. Structured query language — мова структурованих запитів) — декларативна мова програмування для взаємодії користувача з базами даних, що застосовується для формування запитів, оновлення і керування реляційними БД, створення схеми бази даних і її модифікації, системи контролю за доступом до бази даних. Сам по собі SQL не є ні системою керування базами даних, ні окремим програмним продуктом. Не будучи мовою програмування в тому розумінні, як C або Pascal, SQL може формувати інтерактивні запити або, будучи вбудованою в прикладні програми, виступати в якості інструкцій для керування даними. Стандарт SQL, крім того, вміщує функції для визначення зміни, перевірки і захисту даних.

SQL — це діалогова мова програмування для здійснення запиту і внесення змін до бази даних, а також управління базами даних. Багато баз даних підтримує SQL з розширеннями до стандартної мови. Ядро SQL формує командна мова, яка дозволяє здійснювати пошук, вставку, обновлення, і вилучення даних, використовуючи систему управління і адміністративні функції. SQL також включає CLI (Call Level Interface) для доступу і управління базами даних дистанційно.

Перша версія SQL була розроблена на початку 1970-х років у IBM. Ця версія носила назву SEQUEL і була призначена для обробки і пошуку даних, що містилися в реляційній базі даних IBM, System R . Мова SQL пізніше була стандартизована Американськими Держстандартами (ANSI) в 1986. Спочатку SQL розроблялась як мова запитів і управління даними, пізніші модифікації SQL створено продавцями системи управління базами даних, які додали процедурні конструкції, control-of-flow команд і розширення мов. З випуском стандарту SQL:1999 такі розширення були формально запозичені як частина мови SQL через Persistent Stored Modules (SQL/PSM).

Критики SQL включає відсутність крос-платформенності , невідповідною обробкою відсутніх даних (дивіться Null (SQL)), і іноді неоднозначна граматика і семантика мови.

Питання сумісності

Як і з багатьма стандартами, що мають місце в IT-індустрії, з мовою SQL виникла проблема, що у минулому багато виробників ПЗ з використанням SQL вирішили, що функціональність в поточній (на той момент часу) версії стандарту недостатня (що, в принципі, для ранніх версій

SQL було певною мірою справедливо) і його бажано розширити. Що і призводить в цей час до того, що у різних виробників СУБД в ході різні діалекти SQL, в загальному випадку між собою несумісні.

До 1996 року питаннями відповідності комерційних реалізацій SQL стандарту займався в основному інститут NIST, який і встановлював рівень відповідності стандарту. Але пізній підрозділ, що займався СУБД, був розформований, і в цей час всі зусилля з перевірки СУБД на відповідність стандарту лягають на її виробника.

Вперше поняття «Рівня відповідності» було запропоноване в стандарті SQL-92. А саме, ANSI і NIST визначали чотири рівні відповідності реалізації цьому стандарту:

Entry (базовий)

Transitional (перехідний) — перевірку на відповідність цьому рівню проводив тільки інститут NIST

Intermediate (проміжний)

Full (повний)

Легко можна зрозуміти, що кожен подальший рівень відповідності свідомо мав на увазі відповідність попередньому рівню. Далі, згідно з цими рівнями стандартів будь-яка СУБД, яка відповідала рівню Entry, могла заявляти себе як «SQL-92 відповідна», хоча насправді переносимість і відповідність стандарту обмежувалося набором можливостей, що входять у цей рівень.

"Положення" змінилося з введенням стандарту SQL:1999. Відтепер стандарт придбав модульну структуру — основна частина стандарту була винесена в розділ «SQL/Foundation», всі інші були виведені в окремі модулі. Відповідно, залишився тільки один рівень сумісності — Core, що означало підтримку цієї основної частини. Підтримка решти можливостей залишена на розсуд виробників СУБД. Аналогічне положення мало місце і з подальшими версіями стандарту.

Переваги

Незалежність від конкретної СУБД

Незважаючи на наявність діалектів і відмінностей в синтаксисі, в більшості своїй тексти SQL-запитів, що містять, DDL і DML, можуть бути досить легко перенесені з однієї СУБД в іншу. Існують системи, розробники яких спочатку закладалися на застосування щонайменше кількох СУБД (наприклад: система електронного документообігу Documentum може працювати як з Oracle, так і з Microsoft SQL Server та IBM DB2). Природно,

що при застосуванні деяких специфічних для реалізації можливостей такої переносимості добитися вже дуже важко.

Наявність стандартів

Наявність стандартів і набору тестів для виявлення сумісності і відповідності конкретній реалізації SQL загальноприйнятому стандарту тільки сприяє «стабілізації» мови. Правда, варто звернути увагу, що сам по собі стандарт місцями занадто формалізований і роздутий в розмірах, наприклад, Core-частину стандарту SQL:2003 включає понад 1300 сторінок тексту.

Декларативність

За допомогою SQL програміст описує тільки те, які дані потрібно витягнути або модифікувати. Те, яким чином це зробити, вирішує СУБД безпосередньо при обробці SQL-запиту. Проте не варто думати, що це повністю універсальний принцип — програміст описує набір даних для вибірки або модифікації, проте йому при цьому корисно уявляти, як СУБД розбиратиме текст його запиту. Особливо критичні такі моменти стають при роботі з великими базами даних і зі складними запитами — чим складніше сконструйований запит, тим більше він допускає варіантів написання, різних за швидкістю виконання, але тих самих за набором даних.

Недоліки

Невідповідність реляційної моделі даних

Творець реляційної моделі даних Едгар Кодд, Крістофер Дейт та їхні прихильники указують на те, що SQL не є істинно реляційною мовою. Зокрема вони указують на такі проблеми SQL[3]:

Рядки, що повторюються
Невизначені значення (null)
Явна вказівка порядку стовпчиків зліва направо
Стовпці без імені та імена стовпчиків, що дублюються
Відсутність підтримки властивості «=»
Використання покажчиків
Висока надлишковість

У опублікованому Крістофером Дейтом і Г'ю Дарвеном Третьому Маніфесті[4] вони висловлюють принципи СУБД наступного покоління і пропонують мову Tutorial D, яка є достовірно реляційною.

Складність

Хоча SQL і замислювався, як засіб роботи кінцевого користувача, врешті-решт він став настільки складним, що перетворився на інструмент програміста.

Відступи від стандартів

Незважаючи на наявність міжнародного стандарту ANSI SQL-92, багато компаній, СУБД (наприклад, Oracle, Sybase, Microsoft, MySQL), що займаються розробкою, вносять зміни до мови SQL, вживаної в розроблених ними СУБД, тим самим відступаючи від стандарту. Таким чином з'являються специфічні для кожної конкретної СУБД діалекти мови SQL.

Складність роботи з ієрархічними структурами

Раніше SQL не пропонував стандартного способу маніпуляції деревовидними структурами. Деякі постачальники СУБД пропонували свої рішення. Наприклад, Oracle використовує вираз CONNECT BY. В наш час[Коли?] як стандарт прийнята рекурсивна конструкція WITH.

Процедурні розширення

Оскільки SQL не є мовою програмування (тобто не надає засобів для автоматизації операцій з даними), введені різними виробниками розширення стосувалися в першу чергу процедурних розширень. Це збережені процедури (англ. stored procedures) і процедурні мови-«надбудови». Практично в кожній СУБД застосовується своя процедурна мова. Подібні мови для найпопулярніших СУБД приведені в такій таблиці:СУБД Коротка назва
Розшифровка
Borland InterBase/
Firebird PSQL Procedural SQL
IBM DB2 SQL PL SQL Procedural Language (розширює SQL/PSM)
Microsoft SQL Server/
Sybase ASE Transact-SQL Transact-SQL
MySQL SQL/PSM SQL/Persistent Stored Module
Oracle PL/SQL Procedural Language/SQL (заснований на мові Ada)
PostgreSQL PL/pgSQL Procedural Language/PostgreSQL Structured Query Language (схожий на Oracle PL/SQL)

Простий приклад

Простий запит для виведення списку із атрибутами Name, Address, Class із таблиці School у певній базі данних має такий вигляд:
SELECT Name, Address, Class

```
FROM School;
```

1 Синтаксис оператора SELECT

Оператор SELECT має наступну структуру:

```
SELECT [STRAIGHT_JOIN]
    [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
    [SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS] [HIGH_PRIORITY]
    [DISTINCT | DISTINCTROW | ALL]
  select_expression, ...
  [INTO {OUTFILE | DUMPFILE} 'file_name' export_options]
  [FROM table_references
    [WHERE where_definition]
    [GROUP BY {unsigned_integer | col_name | formula} [ASC | DESC], ...]
    [HAVING where_definition]
    [ORDER BY {unsigned_integer | col_name | formula} [ASC | DESC], ...]
    [LIMIT [offset,] rows]
    [PROCEDURE procedure_name]
    [FOR UPDATE | LOCK IN SHARE MODE]]
```

SELECT застосовується для вилучення рядків, вибраних з однієї або декількох таблиць. Вираз select_expression задає стовпці, в яких необхідно проводити вибірку. Крім того, оператор SELECT можна використовувати для витягання рядків, обчислених без посилання якусь таблицю. Наприклад:

```
mysql> SELECT 1 + 1;
    -> 2
```

При вказівці ключових слів слід точно дотримуватися порядок, зазначений вище. Наприклад, вираз HAVING повинне розташовуватися після всіх виразів GROUP BY і перед всіма виразами ORDER BY .

Використовуючи ключове слово AS , вираженню в SELECT можна призначити назву. Псевдонім використовується в якості імені стовпця в даному виразі і може застосовуватися в ORDER BY або HAVING . Наприклад:

```
mysql> SELECT CONCAT (last_name, ',', first_name) AS full_name
    FROM mytable ORDER BY full_name;
```

Псевдоніми стовпців можна використовувати у виразі WHERE , оскільки знаходяться в стовпцях величини на момент виконання WHERE можуть бути ще не визначені. See Section A.5.4 Проблеми з alias .

Вираз FROM table_references задає таблиці, з яких належить витягувати рядки. Якщо вказано ім'я більш ніж однієї таблиці, слід виконати об'єднання. Інформацію про синтаксис об'єднання можна знайти в розділі Section 6.4.1.1 Синтаксис оператора JOIN . Для кожної заданої таблиці за бажанням можна вказати псевдонім.

table_name [[AS] alias] [USE INDEX (key_list)] [IGNORE INDEX (key_list)]

У версії MySQL 3.23.12 можна вказувати, які саме індекси (ключі) MySQL повинен застосовувати для добування інформації з таблиці. Це корисно, якщо оператор EXPLAIN (виводить інформацію про структуру та порядку виконання запиту SELECT), показує, що MySQL використовує хибний індекс. Якщо потрібно. щоб для пошуку запису в таблиці застосовувався тільки один із зазначених індексів, слід задати значення цього індексу в USE INDEX (key_list). Альтернативне вираз IGNORE INDEX (key_list) забороняє використання в MySQL даного конкретного індексу. Вирази USE/IGNORE KEY є синонімами для USE/IGNORE INDEX .

Посилання на стовпці можуть задаватися у вигляді col_name , tbl_name.col_name або db_name.tbl_name.col_name . У виразах tbl_name або db_name.tbl_name немає необхідності вказувати префікс для посилань на стовпці в команді SELECT , якщо ці посилання не можна витлумачити неоднозначно. See section 6.1.2 Імена баз даних, таблиць, стовпців, індекси псевдоніми , де наведені приклади неоднозначних випадків, для яких потрібні більш чіткі визначення посилань на стовпці.

Посилання на таблицю можна замінити псевдонімом, використовуючи tbl_name [AS] alias_name :

```
mysql> SELECT t1.name, t2.salary FROM employee AS t1, info AS t2
       WHERE t1.name = t2.name;

mysql> SELECT t1.name, t2.salary FROM employee t1, info t2
       WHERE t1.name = t2.name;
```

У виразах ORDER BY і GROUP BY для посилань на стовпці, вибрані для виведення інформації, можна використовувати або імена стовпців, або їх псевдоніми, або їх позиції (місця розташування). Нумерація позицій стовпців починається з 1 :

```
mysql> SELECT college, region, seed FROM tournament
       ORDER BY region, seed;

mysql> SELECT college, region AS r, seed AS s FROM tournament
       ORDER BY r, s;

mysql> SELECT college, region, seed FROM tournament
       ORDER BY 2, 3;
```

Для того щоб сортування проводилася в зворотному порядку, в утвердженні ORDER BY до імені заданого стовпця, в якому виробляється сортування, слід додати ключове слово DESC (спадаючий). За

замовчуванням прийнята сортування в зростаючому порядку, який можна задати явно за допомогою ключового слова ASC .

У виразі WHERE можна використовувати будь-яку з функцій, яка підтримується в MySQL. See section 6.3 Функції, використовувані в операторах SELECT і WHERE . Вираз HAVING може посилатися на будь-який стовпець або псевдонім, згаданий у вираженні select_expression . HAVING відпрацьовується останнім, безпосередньо перед відсиланням даних клієнтові, і без якої б то не було оптимізації. Не використовуйте цей вираз для визначення того, що повинно бути визначено в WHERE . Наприклад, не можна задати наступний оператор:

```
mysql> SELECT col_name FROM tbl_name HAVING col_name> 0;
```

Замість цього слід задавати:

```
mysql> SELECT col_name FROM tbl_name WHERE col_name> 0;
```

У версії MySQL 3.22.5 або пізнішої можна також писати запити, як показано нижче:

```
mysql> SELECT user, MAX (salary) FROM users
    GROUP BY user HAVING MAX (salary)> 10;
```

У більш старих версіях MySQL замість цього можна вказувати:

```
mysql> SELECT user, MAX (salary) AS sum FROM users
    GROUP BY user HAVING sum> 10;
```

Параметри (опції) DISTINCT , DISTINCTROW і ALL вказують, чи повинні повертатися дублюються записи. За умовчанням встановлений параметр (ALL), тобто повертаються всі зустрічаються рядки. DISTINCT і DISTINCTROW є синонімами і вказують, що дублюються рядки в результуючому наборі даних мають бути видалені.

Всі параметри, що починаються з SQL_ , STRAIGHT_JOIN і HIGH_PRIORITY , являють собою розширення MySQL для ANSI SQL.

При вказівці параметра HIGH_PRIORITY містить його оператор SELECT матиме вищий пріоритет, ніж команда поновлення таблиці. Потрібно тільки використовувати цей параметр із запитами, які повинні виконуватися дуже швидко і відразу. Якщо таблиця заблокована для читання, то запит SELECT HIGH_PRIORITY буде виконуватися навіть за наявності команди оновлення, що очікує, поки таблиця звільниться.

Параметр SQL_BIG_RESULT можна використовувати з GROUP BY або DISTINCT , щоб повідомити оптимізатору, що результат буде містити велику кількість рядків. Якщо вказаний цей параметр, MySQL при необхідності буде безпосередньо використовувати тимчасові таблиці на диску, проте перевага буде віддаватися не створення тимчасової таблиці з ключем за елементами GROUP BY , а сортуванні даних.

При вказівці параметра SQL_BUFFER_RESULT MySQL буде заносити результат в тимчасову таблицю. Таким чином MySQL отримує можливість раніше зняти блокування таблиці; це корисно також для випадків, коли для посилки результату клієнтові потрібен значний час.

Параметр SQL_SMALL_RESULT є опцією, специфічної для MySQL. Даний параметр можна використовувати з GROUP BY або DISTINCT , щоб повідомити оптимізатору, що результуючий набір даних буде невеликим. У цьому випадку MySQL для зберігання результуючої таблиці замість сортування буде використовувати швидкі тимчасові таблиці. У версії MySQL 3.23 вказувати даний параметр зазвичай немає необхідності.

Параметр SQL_CALC_FOUND_ROWS повертає кількість рядків, які повернув би оператор SELECT , якби не був вказаний LIMIT . Шукана кількість рядків можна отримати за допомогою SELECT FOUND_ROWS() . See Section 6.3.6.2 Різні функції .

Параметр SQL_CACHE наказує MySQL зберігати результат запиту в кеші запитів при використанні SQL_QUERY_CACHE_TYPE=2 (DEMAND). See section 6.9 Кеш запитів в MySQL .

Параметр SQL_NO_CACHE забороняє MySQL зберігати результат запиту в кеші запитів. See section 6.9 Кеш запитів в MySQL .

При використанні виразу GROUP BY рядки виводу будуть сортуватися у відповідності з порядком, заданим в GROUP BY , - так, як якщо б застосовувалося вираз ORDER BY для всіх полів, зазначених в GROUP BY . В MySQL вираз GROUP BY розширене таким чином, що для нього можна також вказувати параметри ASC і DESC :

SELECT a, COUNT (b) FROM test_table GROUP BY a DESC

Розширений оператор GROUP BY в MySQL забезпечує, зокрема, можливість вибору полів, не згаданих у вираженні GROUP BY . Якщо ваш запит не приносить очікуваних результатів, прочитайте, будь ласка, опис GROUP BY . See section 6.3.7 Функції, використовувані в операторах GROUP BY .

При вказівці параметра STRAIGHT_JOIN оптимізатор буде об'єднувати таблиці в тому порядку, в якому вони перераховані у виразі FROM . Застосування даного параметра дозволяє збільшити швидкість виконання запиту, якщо оптимізатор виробляє об'єднання таблиць неоптимальним чином. See section 5.2.1 Синтаксис оператора EXPLAIN (одержання інформації про SELECT) .

Вираз LIMIT може використовуватися для обмеження кількості рядків, повернутих командою SELECT . LIMIT приймає один або два числових аргументу. Ці аргументи повинні бути цілочисельними константами. Якщо задані два аргументи, то перший вказує на початок першої повертаної рядки, а другий задає максимальну кількість повертаних рядків. При цьому зміщення початкового рядка одно 0 (не 1):

mysql> SELECT * FROM table LIMIT 5,10; # повертає рядки 6-15

Якщо заданий один аргумент, то він показує максимальну кількість повертаних рядків:

mysql> SELECT * FROM table LIMIT 5; # повертає перших 5 рядків

Іншими словами, LIMIT n еквівалентно LIMIT 0,n .

Оператор SELECT може бути представлений у формі SELECT ... INTO OUTFILE 'file_name' SELECT ... INTO OUTFILE 'file_name' . Цей різновид

команди здійснює запис вибраних рядків у файл, вказаний в file_name . Даний файл створюється на сервері і до цього не повинен існувати (таким чином, крім іншого, запобігається руйнування таблиць і файлів, таких як `/ etc / passwd '). Для використання цієї форми команди SELECT необхідні привілеї FILE . Форма SELECT ... INTO OUTFILE SELECT ... INTO OUTFILE головним чином призначена для виконання дуже швидкого дампа таблиці на серверному комп'ютері. Команду SELECT ... INTO OUTFILE SELECT ... INTO OUTFILE не можна застосовувати, якщо необхідно створити результуючий файл на іншому хості, відмінному від серверного. У такому випадку для генерації потрібного файлу замість цієї команди слід використовувати деяку клієнтську програму на зразок mysqldump --tab або mysql -e "SELECT ..." > outfile mysql -e "SELECT ..." > outfile . Команда SELECT ... INTO OUTFIL SELECT ... INTO OUTFIL E є додатковою по відношенню до LOAD DATA INFIL E; синтаксис частини export_options цієї команди містить ті ж вирази FIELDS і LINES , які використовуються в команді LOAD DATA INFIL E. See section 6.4.9 Синтаксис оператора LOAD DATA INFILE . Слід враховувати, що в результуючому текстовому файлі оператор ESCAPED BY екранує тільки наступні символи:

Символ оператора ESCAPED BY

Перший символ оператора FIELDS TERMINATED BY

Перший символ оператора LINES TERMINATED BY

Крім цього ASCII-символ 0 конвертується в ESCAPED BY , за яким слідує символ `0 '(ASCII 48). Це робиться тому, що необхідно екранувати будь-які символи операторів FIELDS TERMINATED BY , ESCAPED BY або LINES TERMINATED BY , щоб мати надійну можливість повторити читання цього файлу. ASCII 0 екранується, щоб полегшити перегляд файлу за допомогою програм виведення типу pager. Оскільки результуючий файл не повинен задовольняти синтаксису SQL, немає необхідності екранувати небудь ще. Нижче наведено приклад того, як отримати файл у форматі, який використовується багатьма старими програмами.

```
SELECT a, b, a + b INTO OUTFILE "/ tmp / result.text"
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
LINES TERMINATED BY "\ n"
FROM test_table;
```

Якщо замість INTO OUTFILE використовувати INTO DUMPFILE , то MySQL запише в файл тільки один рядок без символів завершення стовпців або рядків і без якого б то не було екранування. Це корисно для зберігання даних типу BLOB в файлі.

Слід враховувати, що будь-який файл, створений за допомогою INTO OUTFILE і INTO DUMPFILE , буде доступний для читання всім користувачам! Причина цього полягає в наступному: сервер MySQL не може створювати файл, що належить тільки якого-небудь поточному користувачеві (ви ніколи не можете запустити mysqld від користувача root), відповідно, файл повинен бути доступний для читання всім користувачам.

При використанні FOR UPDATE з обробником таблиць, що підтримує блокування сторінок / рядків, обрані рядки будуть заблоковані для запису.

2. Синтаксис оператора INSERT

```
INSERT [LOW_PRIORITY | DELAYED] [IGNORE]
    [INTO] tbl_name [(col_name, ...)]
    VALUES (expression, ...), (...), ...
або INSERT [LOW_PRIORITY | DELAYED] [IGNORE]
    [INTO] tbl_name [(col_name, ...)]
    SELECT ...
або INSERT [LOW_PRIORITY | DELAYED] [IGNORE]
    [INTO] tbl_name
    SET col_name = expression, col_name = expression, ...
```

Оператор INSERT вставляє нові рядки в існуючу таблицю. Форма даної команди INSERT ... VALUES INSERT ... VALUES вставляє рядки відповідно до точно зазначеними в команді значеннями. Форма INSERT ... SELECT INSERT ... SELECT вставляє рядки, обрані з іншої таблиці або таблиць. Форма INSERT ... VALUES INSERT ... VALUES зі списком з декількох значень підтримується у версії MySQL 3.22.5 і більш пізніх. Синтаксис вираження col_name=expression підтримується у версії MySQL 3.22.10 і більш пізніх.

tbl_name задає таблицю, в яку повинні бути внесені рядки. Стовпці, для яких задані величини в команді, вказуються в списку імен стовпців або в частині SET :

Якщо не зазначений список стовпців для INSERT ... VALUES INSERT ... VALUES або INSERT ... SELECT INSERT ... SELECT , то величини для всіх стовпців повинні бути визначені в списку VALUES() або в результаті роботи SELECT . Якщо порядок стовпців у таблиці невідомий, для його отримання можна використовувати DESCRIBE tbl_name .

Будь стовпець, для якого явно не вказано значення, буде встановлений у своє значення за замовчуванням. Наприклад, якщо в заданому списку стовпців не зазначені всі стовпці в даній таблиці, то не згадані стовпці встановлюються у свої значення за замовчуванням. Установка значень за замовчуванням описується в розділі section 6.5.3 Синтаксис оператора CREATE TABLE . В MySQL завжди передбачено значення за замовчуванням для кожного поля. Ця вимога `` нав'язано'' MySQL, щоб забезпечити можливість роботи як з таблицями, що підтримують транзакції, так і з таблицями, не підтримують їх. Наша точка зору (розробників) полягає в тому, що перевірка вмісту полів повинна проводитися додатком, а не сервером баз даних.

Вираз expression може відноситься до будь-якому стовпцю, який раніше був внесений до списку значень. Наприклад, можна вказати наступне:

mysql> INSERT INTO tbl_name (col1, col2) VALUES (15, col1 * 2);

Але не можна вказати:

mysql> INSERT INTO tbl_name (col1, col2) VALUES (col2 * 2,15);

Якщо вказується ключове слово LOW_PRIORITY , то виконання даної команди INSERT буде затримано до тих пір, поки інші клієнти не завершать читання цієї таблиці. У цьому випадку даний клієнт повинен чекати, поки дана команда вставки не буде завершена, що в разі інтенсивного використання таблиці може зажадати значного часу. На противагу цьому команда INSERT DELAYED дозволяє даному клієнтові продовжувати операцію відразу ж. See section 6.4.4 Синтаксис оператора INSERT DELAYED . Слід зазначити, що покажчик LOW_PRIORITY зазвичай не використовується з таблицями MyISAM , оскільки при його вказівці стають неможливими паралельні вставки. See section 7.1 Таблиці MyISAM .

Якщо в команді INSERT з рядками, які мають багато значень, вказується ключове слово IGNORE , то всі рядки, що мають дублюються ключі PRIMARY або UNIQUE в цій таблиці, будуть проігноровані і не будуть внесені. Якщо не вказувати IGNORE , то дана операція вставки припиняється при виявленні рядка, що має дублюється значення існуючого ключа. Кількість рядків, внесених в дану таблицю, можна визначити за допомогою функції C API mysql_info() .

Якщо MySQL був налаштований з використанням опції DONT_USE_DEFAULT_FIELDS , то команда INSERT буде генерувати помилку, якщо явно не вказати величини для всіх стовпців, які вимагають значень не- NULL . See section 2.3.3 Типові опції configure .

За допомогою функції mysql_insert_id можна знайти величину, використану для стовпця AUTO_INCREMENT . See Section 8.4.3.126 mysql_insert_id() .

Якщо задається команда INSERT ... SELECT INSERT ... SELECT або INSERT ... VALUES INSERT ... VALUES зі списками з декількох значень, то для отримання інформації про даному запиті можна використовувати функцію C API mysql_info() . Формат цієї інформаційної рядки наведено нижче:

Records: 100 Duplicates: 0 Warnings: 0

Duplicates показує число рядків, які не могли бути внесені, оскільки вони дублювали б значення деяких існуючих унікальних індексів. Покажчик Warnings показує число спроб внести величину в стовпець, який з якоїсь причини опинився проблематичним. Попередження виникають при виконанні будь-якого з таких умов:

Внесення NULL в стовпець, який був оголошений, як NOT NULL . Даний стовпець встановлюється в значення, задане за замовчуванням.

Установка числового стовпця в значення, що лежить за межами його допустимого діапазону. Дана величина усікається до відповідної кінцевої точки цього діапазону.

Занесення в числовий стовпець такої величини, як '10.34 a' . Кінцеві дані видаляються і вноситься тільки залишилася числова частина. Якщо величина зовсім не має сенсу як число, то стовпець встановлюється в 0 .

Внесення в стовпці типу CHAR , VARCHAR , TEXT або BLOB рядки, що перевершує максимальну довжину стовпця. Дана величина усікається до максимальної довжини стовпця.

Внесення в стовпець дати або часу рядки, неприпустимій для даного типу стовпця. Цей стовпець встановлюється в нульову величину, відповідну даному типу.

3 Синтаксис оператора UPDATE

```
UPDATE [LOW_PRIORITY] [IGNORE] tbl_name
    SET col_name1 = expr1 [, col_name2 = expr2, ...]
    [WHERE where_definition]
    [LIMIT #]
```

Оператор UPDATE оновлює стовпці відповідно до їх новими значеннями в рядках існуючої таблиці. У виразі SET вказується, які саме стовпці слід модифікувати і які величини повинні бути в них встановлені. У виразі WHERE , якщо воно присутнє, задається, які рядки підлягають оновленню. В інших випадках оновлюються всі рядки. Якщо задано вираз ORDER BY , то рядки будуть оновлюватися у зазначеному в ньому порядку.

Якщо вказується ключове слово LOW_PRIORITY , то виконання даної команди UPDATE затримується до тих пір, поки інші клієнти не завершать читання цієї таблиці.

Якщо вказується ключове слово IGNORE , то команда оновлення не буде перервана, навіть якщо при оновленні виникне помилка дублювання ключів. Рядки, через які виникають конфліктні ситуації, оновлені не будуть.

Якщо доступ до стовпцю із зазначеного виразу здійснюється по аргументу tbl_name , то команда UPDATE використовує для цього стовпця його поточне значення. Наприклад, наступна команда встановлює стовпець age в значення, на одиницю більше його поточної величини:

```
mysql> UPDATE persondata SET age = age +1;
```

Значення команда UPDATE присвоює зліва направо. Наприклад, наступна команда дублює стовпець age , потім інкрементіруется його:

```
mysql> UPDATE persondata SET age = age * 2, age = age +1;
```

Якщо стовпець встановлюється в його поточне значення, то MySQL помічає це і не оновлює його.

Команда UPDATE повертає кількість фактично змінених рядків. У версії MySQL 3.22 і більш пізніх функція C API mysql_info() повертає кількість рядків, які були знайдені і оновлені, і кількість попереджень, що мали місце при виконанні UPDATE .

У версії MySQL 3.23 можна використовувати LIMIT # , щоб переконатися, що було змінено лише задану кількість рядків.

4 Синтаксис оператора DELETE

```
DELETE [LOW_PRIORITY | QUICK] FROM table_name
      [WHERE where_definition]
      [ORDER BY ...]
      [LIMIT rows]
```

або

```
DELETE [LOW_PRIORITY | QUICK] table_name [. *] [, Table_name [. *] ...]
      FROM table-references
      [WHERE where_definition]
```

оілі

```
DELETE [LOW_PRIORITY | QUICK]
      FROM table_name [. *], [Table_name [. *] ...]
      USING table-references
      [WHERE where_definition]
```

Оператор DELETE видаляє з таблиці table_name рядки, що задовольняють заданим в where_definition умовам, і повертає число віддалених записів.

Якщо оператор DELETE запускається без визначення WHERE , то видаляються всі рядки. При роботі в режимі AUTOCOMMIT це буде аналогічно використанню оператора TRUNCATE . See section 6.4.7 Синтаксис оператора TRUNCATE . В MySQL 3.23 оператор DELETE без визначення WHERE поверне нуль як число віддалених записів.

Якщо дійсно необхідно знати число видалених записів при видаленні всіх рядків, і якщо припустимі втрати в швидкості, то можна використовувати команду DELETE в наступній формі:

```
mysql> DELETE FROM table_name WHERE 1> 0;
```

Слід враховувати, що ця форма працює набагато повільніше, ніж DELETE FROM table_name без виразу WHERE , оскільки рядки видаляються по черзі по одній.

Якщо вказано ключове слово LOW_PRIORITY , виконання даної команди DELETE буде затримано до тих пір, поки інші клієнти не завершать читання цієї таблиці.

Якщо задано параметр QUICK , то вказівник таблиці при виконанні видалення не буде об'єднувати індекси - в деяких випадках це може прискорити дану операцію.

У таблицях MyISAM видалені записи зберігаються у зв'язаному списку, а наступні операції INSERT повторно використовують місця, де розташовувалися видалені записи. Щоб повернути невикористовуване простір і зменшити розмір файлів, можна застосувати команду OPTIMIZE TABLE або утиліту myisamchk для реорганізації таблиць. Команда OPTIMIZE TABLE простіше, але утиліта myisamchk працює швидше. See section 4.5.1 Синтаксис команди OPTIMIZE TABLE . See Section 4.4.6.10 Оптимізація таблиць .

Перший з числа наведених на початку даного розділу багатотабличних формат команди DELETE підтримується, починаючи з MySQL 4.0.0. Другий багатотабличних формат підтримується, починаючи з MySQL 4.0.2.

Ідея полягає в тому, що видаляються тільки співпадаючі рядки з таблиць, перелічених перед виразами FROM або USING . Це дозволяє видаляти одноразово рядки з декількох таблиць, а також використовувати для пошуку додаткові таблиці.

Символи .* після імен таблиць потрібні лише для сумісності з Access:
DELETE t1, t2 FROM t1, t2, t3 WHERE t1.id = t2.id AND t2.id = t3.id

або
DELETE FROM t1, t2 USING t1, t2, t3 WHERE t1.id = t2.id AND t2.id = t3.id

У попередньому випадку просто видалені співпадаючі рядки з таблиць t1 і t2 .

Вираз ORDER BY і використання декількох таблиць в команді DELETE підтримується в MySQL 4.0.

Якщо застосовується вираз ORDER BY , то рядки будуть видалені в зазначеному порядку. В дійсності це вираз корисно тільки в поєднанні з LIMIT . Наприклад:

```
DELETE FROM somelog
    WHERE user = 'jcole'
    ORDER BY timestamp
    LIMIT 1
```

Даний оператор видалить саму стару запис (по timestamp), в якій рядок відповідає вказаній у вираженні WHERE .

Специфічна для MySQL опція LIMIT для команди DELETE вказує сервера максимальну кількість рядків, які слід видалити до повернення управління клієнтові. Ця опція може використовуватися для гарантії того, що дана команда DELETE не зажадає занадто багато часу для виконання. Можна просто повторювати команду DELETE до тих пір, поки кількість віддалених рядків менше, ніж величина LIMIT .

Порядок виконання роботи

1. Виконати простий запит на вибірку
2. Виконати запит на створення запису
3. Виконати запит на редагування запису
4. Виконати запит на видалення запису

ЛАБОРАТОРНА РОБОТА 3.
ТЕМА: СКЛАДНІ ЗАПИТИ НА ВИБІРКУ І ГРУПУВАННЯ.
ПОБУДОВА ЗАПИТІВ НА ОСНОВІ КІЛЬКОХ ТАБЛИЦЬ.
МЕТА: НАБУТИ ПРАКТИЧНИХ НАВИКІВ ВИКОРИСТАННЯ
РІЗНИХ ТИПІВ ОБЄДНАНЬ ТА ПІДЗАПИТІВ

ТЕОРЕТИЧНІ ВІДОМОСТІ

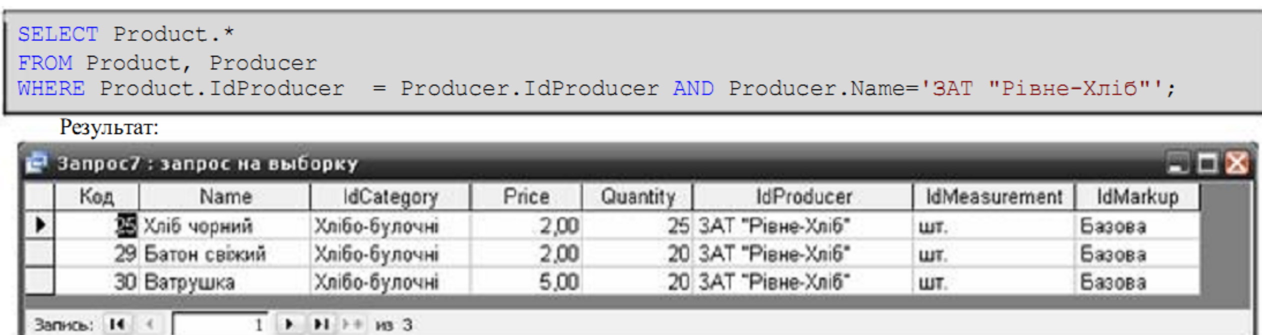
1. Підзапити

1.1. Види та застосування вкладених підзапитів

Стандартом SQL передбачена можливість вкладати запити один в одний, що має велике практичне застосування. В результаті такого вкладення, одні запити можуть управляти іншими. При цьому вводиться поняття підзапиту. Підзапит - це запит, який міститься в іншому запиті SQL. Він являє собою повноцінний SELECT вираз, результат виконання якого використовується в іншому запиті. Розміщуватись вони можуть майже в довільному місці SQL виразу, наприклад, замість одного з імен в списку SELECT, в операторі FROM, при вказанні умови в операторах WHERE або HAVING. Найчастіше зустрічається використання підзапитів при вказанні умови. Слід відмітити, що самі по собі підзапити не додають жодних функціональних можливостей, але іноді з ними запити стають більш читабельнішими, ніж при складній вибірці.

При використанні підзапитів часто використовують поняття зовнішнього та внутрішнього запиту. Спочатку виконується підзапит, тобто внутрішній запит, який розміщується, наприклад, в інструкції WHERE, а потім основний, тобто зовнішній запит, який може бути інструкцією SELECT, INSERT, DELETE або UPDATE. При цьому вкладений підзапит завжди обмежується дужками.

Щоб краще зрозуміти як користуватись підзапитами і як вони працюють, розглянемо все на прикладі. Для початку, напишемо запит без використання підзапитів, який виводить всю інформацію про товари лише одного постачальника. За звичайних умов ми напишемо:



```
SELECT Product.*
FROM Product, Producer
WHERE Product.IdProducer  = Producer.IdProducer AND Producer.Name='ЗАТ "Рівне-Хліб"';
```

Результат:

Запрос7 : запрос на выборку

Код	Name	IdCategory	Price	Quantity	IdProducer	IdMeasurement	IdMarkup
25	Хліб чорний	Хлібо-булочні	2,00	25	ЗАТ "Рівне-Хліб"	шт.	Базова
29	Батон свіжий	Хлібо-булочні	2,00	20	ЗАТ "Рівне-Хліб"	шт.	Базова
30	Ватрушка	Хлібо-булочні	5,00	20	ЗАТ "Рівне-Хліб"	шт.	Базова

Запись: 1 из 3

З використанням підзапитів, наш запит перепишеться наступним чином:

```
SELECT  *
FROM Product
WHERE IdProducer  =
                (SELECT IdProducer
                FROM Producer
                 WHERE Name='ЗАТ "Рівне-Хліб"');
```

Результат буде аналогічний. Що ж відбувається при використанні підзапиту?

Спочатку цілісно виконується підзапит, який розміщується в інструкції WHERE. Результатом його виконання буде ідентифікатор, - значення поля Name рівне ЗАТ “Рівне-Хліб”.

Producer : таблица

IdProducer	Name	IdAddress
27	ПП "Ряба курка"	Рівне
28	ЗАТ "Яблуко"	Омськ
29	ЗАТ "Картошка"	Мінськ
30	ВАТ "Карась"	Петрозаводск
31	ЗАТ "ДПС"	Рівне
32	ЗАТ "Рівне-Хліб"	Рівне
33	ЗАТ "Румянець"	Рівне
34	ВАТ "Росинка"	Київ
35	Banana Republica	Вашингтон
(Счетчик)		

Запрос8...

IdProducer
32
(Счетчик)

Запись:

Отриманий результат повертається в основний запит і використовується при його виконанні. Тобто зовнішні ключі, які використовуються для зв’язку з таблицею Producer співставляються з первинним ключем, який є результатом виконання внутрішнього підзапиту. В результаті будуть обрані лише ті товари, які були виготовлені на ЗАТ «Рівне-Хліб».

Перш, ніж ми перейдемо до розгляду інших прикладів по використанню підзапитів, ознайомимось з обмеженнями при їх використанні. Вони наступні:

1)Результатом підзапиту повинно бути лише одне значення, причому його тип даних повинен відповідати типу

2)значення, яке використовується в зовнішньому запиті. Наприклад, нам потрібно вивести всю інформацію про товари двох виробників: ЗАТ «Рівне-Хліб» та ВАТ «Росинка».

```
SELECT  *
FROM Product
WHERE IdProducer  =
                (SELECT IdProducer
                FROM Producer
                 WHERE Name='ЗАТ "Рівне-Хліб"' OR    Name='ВАТ "Росинка"');
```

Даний запит призведе до помилки, оскільки результатом даного підзапиту є кілька значень, тобто два ідентифікатори, значення полів Name яких рівне ЗАТ “Рівне-Хліб” та ВАТ “Росинка”.

Отже, при використанні запитів, основаних на операторах порівняння або логічних операторах, слід бути дуже уважним, щоб кінцевим результатом підзапиту був лише один запис.

Виходом з цієї ситуації є використання оператора IN, який застосовується для перебору значень результату роботи внутрішнього підзапиту:

```
SELECT  *
FROM Product
WHERE IdProducer IN
                 (SELECT IdProducer
                 FROM Producer
                  WHERE Name='ЗАТ "Рівне-Хліб"' OR    Name='ВАТ "Росинка"');
```

2) Результатом підзапиту може бути NULL-запис. В такому випадку результат роботи підзапиту буде оцінений як UNKNOWN, що рівносильне FALSE. В результаті попередній запит є вірним навіть без використання оператора IN, але за умови, якщо товарів одного з вказаних виробників або обох в базі даних не існує. В такому випадку підзапит на виході поверне один або жодного записів.

В де-яких випадках, для підстраховки, існує також можливість використання оператора DISTINCT для гарантованого отримання одного запису на виході підзапиту.

3) Згідно стандарту ANSI SQL підзапити є непереміщувані, тобто наступний запит являється вірним:

```
SELECT *
FROM Product
WHERE IdProducer =
(SELECT IdProducer FROM Producer
WHERE Name='ЗАТ "Рівне-Хліб"' OR   Name='ВАТ "Росинка"');
```

Але, якщо поміняти тіло підзапиту з порівнюваним значенням місцями, повинна згенеруватись помилка:

```
SELECT *
FROM Product
WHERE (SELECT IdProducer        FROM Producer
WHERE Name='ЗАТ "Рівне-Хліб"' OR       Name='ВАТ "Росинка"') =
IdProducer;
```

ПРИМІТКА! Дане правило не стосується середовища СУБД MS Access, оскільки вона розглядає і перший і другий варіант як однакові.

4) Вкладений запит не можна використовувати в інструкції ORDER BY.

5) При використанні підзапитів для перевірки результату не можна використовувати оператори BETWEEN, LIKE, IS NULL.

6) Якщо підзапит використовується з немодифікованим оператором порівняння, тобто звичаним знаком рівності (=), без використання ключових

слів SOME, ANY або ALL, то він не може містити оператори групування GROUP BY та HAVING.

В підзапитах допускається використовувати функції агрегування, оскільки їх результатом є єдине значення. Але і тут слід бути уважним (див. п. 6 зауважень). Розглянемо кілька прикладів:

1. Запит на отримання інформації про найдорожчий товар, тобто товар з найбільшою ціною продажу:

```
SELECT DISTINCT Product.Name as [Товар], Sale.Price &' грн.' as [Ціна]
FROM Product, Sale
WHERE Product.IdProduct = Sale.IdProduct
AND Sale.Price = (SELECT max(Sale.Price)
FROM Sale);
```

Результат:

Sale : таблица

IdSale	IdProduct	Price	Quantity	DateSale
1	Фарш "315км/год"	50,10 грн.	1	12.02.2008
16	Ковбаса "Ласунка"	50,10 грн.	2	20.07.2009
3	Фарш "315км/год"	50,10 грн.	10	12.05.2009
2	Цукерки "Радощі у козлика"	50,00 грн.	2	
12	Цукерки "Крабіки"	15,50 грн.	0	
5	Горілка "Чіполіно"	15,00 грн.	5	
4	Горілка "Чіполіно"	13,00 грн.	7	
15	Апельсини "Наколоті"	7,00 грн.	1	

Запрос7 : запрос на выборку

Товар	Ціна
Ковбаса "Ласунка"	50,1 грн.
Фарш "315км/год"	50,1 грн.

Запись: 1 из 2

2. Запит, який виводить на екран імена та прізвища всіх постачальників, які поставляли товар в проміжку між 01/06/2009 та поточною датою:

```
SELECT Name as [Постачальник] FROM Supplier
WHERE IdSupplier IN (SELECT IdSupplier
FROM Delivery
WHERE DateDelivery BETWEEN #01/06/2009# AND Date());
```

Результат:

Delivery : таблица

IdDelivery	IdProduct	IdSupplier	Price	Quantity	DateDelivery
5	Горілка "Чіполіно"	ТзОВ "Бистринка"	10,00р.	10	01.07.2009
4	Молоко "Успевайка"	"International Road" Co.	3,50р.	70	25.06.2009
2	Фарш "315км/год"	ПП "Швидкий вітер"	35,60р.	15	25.06.2009
7	Сухарікі "Дубовые дровишки	ПП "Швидкий вітер"	4,00р.	15	05.06.2009
1	Фарш "315км/год"	ТзОВ "Вмерти, але доставити"	40,50р.	20	01.05.2009
6	Банани	"International Road" Co.	7,00р.	20	03.02.2009
9	Цукерки "Крабіки"	ЗАТ "Хвилинка"	10,50р.		
8	Ковбаса "Роспізнай"	ТзОВ "Бистринка"	50,00р.		

Запрос7 : запрос на выборку

Постачальник
ПП "Швидкий вітер"
"International Road" Co.
ТзОВ "Бистринка"

3. Отримати інформацію про всіх постачальників, які поставляли товар більше, ніж 2 рази. Вибірку відсортувати по назві виробника:

```
SELECT *
FROM Supplier s
WHERE 2 > ( SELECT COUNT(Delivery.IdSupplier)
```

```
FROM Delivery
WHERE s.IdSupplier = Delivery.IdSupplier)
ORDER BY 2;
```

4. Визначити, які поставляємі товари були вироблені в м. Київ:

```
SELECT Name as [Товар], Format(Price, '## ###.00 грн.') as [Ціна] FROM
Product
WHERE IdProduct IN ( SELECT DISTINCT IdProduct
FROM Delivery
WHERE IdSupplier IN (SELECT s.IdSupplier
FROM Supplier s, Address addr
WHERE s.IdAddress=addr.IdAddress AND addr.Town =
'Київ'));
```

Як вже було сказано, підзапит можна використовувати в виразі FROM зовнішнього запиту. Це дозволяє створити тимчасову таблицю і додати її в запит. Наприклад, розглянемо наступний простий запит:

```
SELECT IdProduct, Name, IdCategory
FROM Product
WHERE Price BETWEEN 20 AND 50;
```

Даний запит виведе на екран список товарів та ідентифікатори їх категорій, ціна яких знаходиться в межах 20-50 грн. Цей запит можна використати в рамках іншого, щоб отримати додаткову інформацію. Наприклад, використаємо його, щоб вивести список товарів, категорій «Мясні» та «Ковбасні», ціни яких яких знаходяться у вищевказаному діапазоні.

```
SELECT pr.Name as [Товар]
FROM (SELECT IdProduct, Name, IdCategory FROM Product
WHERE Price BETWEEN 20 AND 50) as pr, Category as c
WHERE pr.IdCategory = c.IdCategory AND c.Name IN ('Мясні',
'Ковбасні');
```

Отже, ми використовуємо підзапит для створення таблиці, яка містить лише три поля: IdProduct, Name та IdCategory. Цій таблиці ми присвоюємо псевдонім pr. Після цього до створеної таблиці можна звернутись з запитом, як до будь-якої іншої таблиці. В даному випадку, ми використовуємо її, щоб отримати інформацію про товари необхідних категорій.

1.2. Оператори EXIST, ANY, SOME, ALL

В попередньому підрозділі ми з вами розглянули в основному однорядкові запити, тобто ті, які повертають один запис. Для того, щоб обробити кілька записів, які поверне підзапит (багаторядковий підзапит), ми використовували оператор IN. Але він не єдиний. Крім оператора IN, існує ще 4 логічних оператора: EXISTS, ALL, ANY та SOME. Розглянемо їх і почнемо з оператора EXISTS.

Оператор EXISTS використовується, коли необхідно визначити наявність значень, які відповідають умові в підзапиті. Якщо дані на виході підзапиту існують, тоді даний оператор поверне значення true, інакше - false. При використанні оператора EXISTS ми насправді використовуємо в підзапиті дані зовнішнього запиту. Такий запит іноді називають зв'язаним або корельованим підзапитом.

Розглянемо наступний запит: вивести інформацію про всіх постачальників, які коли-небудь поставляли товари в магазин:

```
SELECT *
FROM    Supplier
WHERE   EXISTS  (     SELECT *
FROM  Delivery
WHERE Supplier.IdSupplier  = Delivery.IdSupplier)
ORDER BY  3;
```

Проаналізуємо, що вийшло в результаті. В підзапиті ми шукаємо записи таблиці Delivery, в яких значення IdSupplier співпадає з значенням Supplier.IdSupplier. Кожен запис таблиці Supplier співставляється з результатом підзапиту і У ВИПАДКУ ІСНУВАННЯ (WHERE EXISTS) інформація про постачальника додається в результуючу множину.

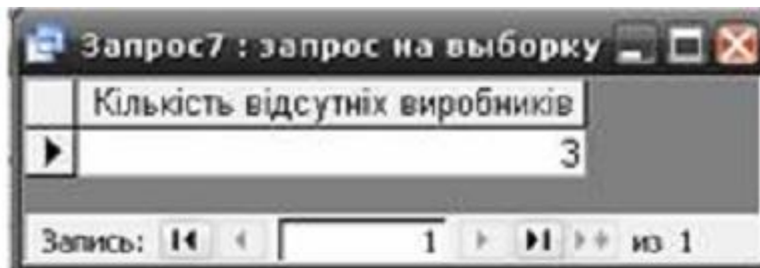
Доречі, те, що повертається підзапитом в підзапит EXISTS або NOT EXISTS абсолютно немає значення. В зв'язку з цим, інколи повертають довільне константне значення. Все, що необхідно знати, - це сам факт наявності або відсутності записів, які відповідають критерію підзапита. Наприклад:

```
SELECT *
FROM Supplier
WHERE EXISTS (   SELECT 'X'
FROM Delivery
WHERE Supplier.IdSupplier  = Delivery.IdSupplier)
ORDER BY  3;
```

Наведемо ще один приклад на використання даного оператора. Необхідно вивести інформацію про кількість виробників, інформація про яких є в базі даних, але товарів яких в магазині немає.

```
SELECT COUNT(*) as  [Кількість відсутніх виробників] FROM Producer
WHERE NOT EXISTS (   SELECT  *
FROM Product
WHERE Product.IdProducer  = Producer.IdProducer);
```

Результат:



Оператори ALL, ANY та SOME використовуються для порівняння одного значення з множиною даних, які повертаються підзапитом. Їх можна комбінувати з усіма операторами порівняння і можуть включати інструкції GROUP BY та HAVING. Варто відмітити, що оператори ANY та SOME взагалі є ідентичними і в стандарті ISO вказується, що вони еквівалентні.

Розглянемо для початку дію оператора ANY.

Оператор Значення

= ANY Рівне довільному значенню, яке повертається під запитом. Прирівнюється до дії оператора IN і може
бути ним замінений

< ANY Меньше найбільшого значення, яке повертається підзапитом. Це можна трактувати як «меньше будь-
якого значення».

> ANY Більше найменьшого значення, яке повертається підзапитом. Це можна трактувати як «більше будь-
якого значення».

Для демонстрації роботи даного оператора перепишемо запит на отримання інформації про постачальників, які поставляли товари в магазин з використанням оператора ANY:

```
SELECT IdSupplier, Name
FROM Supplier
WHERE IdSupplier = ANY ( SELECT IdSupplier
FROM Delivery)
ORDER BY 2;
```

В даному випадку оператор ANY співставляє всі значення поля IdSupplier з таблиці Delivery та повертає результат true, якщо БУДЬ_ЯКЕ (ANY) значення співпаде з значенням поля IdSupplier.

Щодо оператора SOME (який-небудь), то він дасть аналогічний результат. SELECT IdSupplier, Name

```
FROM Supplier
WHERE IdSupplier = SOME ( SELECT IdSupplier
FROM Delivery)
```

```
ORDER BY 2;
```

Різниця між операторами ANY та SOME полягає лише в термінології та заключається в тому, щоб дозволити людям використовувати той термін, який найбільш підходить в даній ситуації.

Використання оператора ALL також нескладне. Даний оператор повертає значення true, якщо кожне значення, яке
буде отримане в результаті роботи підзапиту, відповідає умові зовнішнього запиту. Принцип його дії відображений в
наступній таблиці:

Оператор	Значення
> ALL	Більше найбільшого значення, яке повертається підзапитом. Це можна трактувати як «більше всіх значень».
< ALL	Меньше найменьшого значення, яке повертається підзапитом. Це можна трактувати як «менше всій значень».

В якості прикладу ви конаємо запит, в якому поставимо ціллю довести, що товари категорії "Фрукти" є найбільш
продаваємими.

```
SELECT Product.Name FROM Product, Sale
WHERE Product.IdProduct = Sale.IdProduct
AND Sale.Quantity > ALL ( SELECT Sale.Quantity
FROM Sale, Product, Category
WHERE Sale.IdProduct=Product.IdProduct
AND Product.IdCategory=Category.IdCategory AND
Category.Name='Фрукти');
```

Підзапит поверне список значень кількості продаж (Sale.Quantity) товарів категорії "Фрукти". Потім зовнішній запит шукає кількість продаж товарів, які були більшими, ніж дані, використовуючи для цього вираз Sale.Qauntity>ALL(кількість продаж). Якщо даний запит не поверне жодного запису - це буде підтвердженням того, що това ри вказаної категорії є дійсно найбільш продаваємими, інакше виведеться перелік товарів, які продаються більше, ніш вказаної в підзапиті категорії.

Ще один приклад. Знайдемо та виведемо назви і ціни кондитерських виробів, вартість яких перевищує вартість товарів
категорії «Молочні вироби».

```
SELECT DISTINCT Product.Name as [Товар],
Format(Product.Price, '## ###.00 грн.') as [Ціна] FROM Product, Category
WHERE Product.Price > ALL (SELECT Product.Price
FROM Product, Category
WHERE Product.IdCategory=Category.IdCategory AND
```

Category.Name='Молочні')
AND Product.IdCategory=Category.IdCategory AND Category.Name='Кондитерські';

Результат:

Product : таблиця

Код	Name	IdCategory	Price	Quantity
24	Моршинська 0,5л	Бакалія	2,00	5
23	Моршинська 2л	Бакалія	4,35	5
31	Ковбаса "Ласунка"	Ковбасні	50,00	20
21	Ковбаса "Рослізнай"	Ковбасні	54,00	50
14	Цукерки "Крабіки"	Кондитерські	30,00	56
13	Цукерки "Радощі у козлика"	Кондитерські	49,00	23
18	Сухаріки "Дубовые дровишки"	Кондитерські	5,00	15
16	Лікер "Щасливий випадок"	Лікеро-горілчані	121,00	12
15	Горілка "Чіполіно"	Лікеро-горілчані	12,00	56
17	Молоко "Успевайка"	Молочні	4,00	75
32	Йогурт "Живинка"	Молочні	7,50	10
28	Фарш "Байкер"	Мясні	42,00	50
12	Фарш "315км/год"	Мясні		

Запрос9 : запрос на выборку

Товар	Ціна
Цукерки "Крабіки"	30,00 грн.
Цукерки "Радощі у козлика"	49,00 грн.

Запись: 1 из 2

2. Об’єднання запитів. Оператори UNION та UNION ALL

Об’єднання - це зв’язування даних, що містяться в двох таблицях або запитах в один результуючий набір. Оскільки об’єднання запитів та таблиць відрізняється, розглянемо окремо один і другий спосіб об’єднання. Розпочнемо з простішого, а саме з об’єднання результатів двох або більше запитів.

Об’єднання запитів здійснюється за допомогою оператора SQL UNION. З його допомогою можна об’єднати результати від 2 до 255 результатів запитів в один результуючий набір. В результаті такого об’єднання однакові записи по

замовчуванню знищуються, але при наявності ключового слова ALL (тобто при використанні оператора UNION ALL)

повертаються всі записи, в тому ч ислі і однакові. Синтаксис оператора UNION наступний:

SELECT <список_полів>
[FROM <список_таблиць>] [WHERE <умова>]
[GROUP BY <список_полів_для_групування>] [HAVING <умова_на_групу>]
UNION [ALL]
SELECT <список_полів>
[FROM <список_таблиць>] [WHERE <умова>]
[GROUP BY <список_полів_для_групування>] [HAVING <умова_на_групу>];
[ORDER BY <умова_сортування>];

При використанні оператора UNION притримуються наступних правил:

кількість, послідовність і типи даних полів в обох списках SELECT повинні співпадати;

оператори GROUP BY та HAVING використовуються лише в одному запиті;

жодна з таблиць не може бути відсортована окремо; можна сортувати лише результуючий запит.

Тому оператор ORDER BY можна використовувати лише в кінці оператора UNION;

імена полів результуючої вибірки визначаються списком полів першого оператора SELECT.

Для початку виберемо всі товари, ціна яких більше 20 грн., АБО код категорії товару рівний 1.

```
SELECT Name FROM Product
WHERE Price > 20 UNION
SELECT Name FROM Product
WHERE    IdCategory=1;
```

Результат:

Запрос1 : запрос на объединение

Товар	Ціна	Категорія
назва відсутня		Хлібо-булочні
Лікер "Щасливий випадок"	121,00 грн.	Лікеро-горілчані
Цукерки "Крабіки"	30,00 грн.	Кондитерські
Фарш "Байкер"	42,00 грн.	Мясні
Цукерки "Радощі у козлика"	49,00 грн.	Кондитерські
Фарш "315км/год"	50,00 грн.	Мясні
Ковбаса "Ласунка"	50,00 грн.	Ковбасні
Ковбаса "Роспізнай"	54,00 грн.	Ковбасні

Запись: 1 из 8

товари категорії "Хлібо-булочні"

категорії, ціна на товари яких > 20 грн.

А тепер виберемо всі товари, ціна як их більше 20 грн., АБО які відносяться до категорії "Хлібо-бул очні".

```
SELECT pr.Name as  [Товар], Format(pr.Price, '##  ###.00 грн.') as  [Ціна],
c.Name as  [Категорія]
FROM Product pr, Category c
WHERE    pr.IdCategory  = c.IdCategory AND pr.Price  >  20 UNION
SELECT 'назва відсутня', NULL, Name
FRO M Category
WHERE Name  ='Хлібо-булочні';
```

Результат:

Запрос1 : запрос на объединение

Товар	Ціна	Категорія
назва відсутня		Хлібо-булочні
Лікер "Щасливий випадок"	121,00 грн.	Лікеро-горілчані
Цукерки "Крабіки"	30,00 грн.	Кондитерські
Фарш "Байкер"	42,00 грн.	Мясні
Цукерки "Радощі у козлика"	49,00 грн.	Кондитерські
Фарш "315км/год"	50,00 грн.	Мясні
Ковбаса "Ласунка"	50,00 грн.	Ковбасні
Ковбаса "Роспізнай"	54,00 грн.	Ковбасні

Запись: 1 из 8

товари категорії "Хлібо-булочні"

категорії, ціна на товари яких > 20 грн.

3. Об'єднання таблиць. Стандарт SQL2.

3.1. Види об’єднань

Однією з найсильніших сторін SQL є можливість зв’язувати дані, що розміщуються в окремих таблицях. Це зв’язування здійснюється за рахунок об’єднання таблиць, які вказуються в списку FROM. Разом з цим задається спосіб або тип об’єднання.

СУБД MS Access підтримує лише три типи об’єднань:

1. внутрішнє, яке іноди називають простим;
2. зовнішнє;
3. самооб’єднання.

Інші СУБД, крім вищеперелічених, підтримують і інші типи об’єднань. Причому, деякі специфічні лише для них. Але ці три типи об’єднань являються основними і підтримуються всіма.

Щоб задати спосіб об’єднання, слід скористатись одним з синтаксисів стандарту ANSI: старого стандарту SQL’86 або стандартів SQL2 та вище.

Вони виглядають наступним чином:

```
-- SQL’86
SELECT [таблиця.]поле [,... n] FROM таблиця [,таблиця] [, ...] WHERE умова_об’єднання
-- SQL2 і вище
SELECT [таблиця.]поле [,... n]
FROM таблиця [тип_об’єднання] JOIN таблиця ON умова_обєднання
```

Як видно з опису, синтаксис стандартів відрізняється. Причому, бачимо, що до цього часу ми користувались старим стандартом ANSI SQL, в якому немає можливості вказувати тип об’єднання.

Згідно нового стандарту тип об’єднання вказується ключовим словом при зв’язку двох таблиць. Умова об’єднання, яка тепер вказується після оператора ON являє собою вираз, аналогічний умові відбору, який використовувався в старому стандарті у виразі WHERE. Вона задає як будуть відноситись між собою записи в двох таблицях. Більшість операцій зв’язування виконуються на основі виразів еквівалентності, таких як ПолеА = ПолеВ. Однак умова об’єднання може бути і більша, при цьому всі вирази, які входять в умову об’єднуються за допомогою логічних операторів AND або OR. В цьому плані нічого не змінилось.

3.2. Внутрішні об’єднання. Оператор INNER JOIN

Перший вид об’єднання, який ми розглянемо, буде внутрішнє об’єднання, яке являє собою звичайне об’єднання двох або більше таблиць. Насправді, ви його використовували раніше. Старий стандарт ANSI SQL використовує внутрішній тип об’єднання для зв’язку таблиць.

В стандартах SQL2 і вище, внутрішнє об’єднання здійснюється за допомогою оператора INNER JOIN. Причому використання даного оператора без ключового слова INNER також допускається (СУБД MS Access виключення). В результаті такого об’єднання отримується нова таблиця, записи якої задовольняють відповідним умовам. Як ви вже

зрозуміли, внутрішні об'єднання повертають дані, якщо знаходять спільну інформацію в обох таблицях.

Для прикладу, відобразимо черговий раз інформацію про товари та їх категорії.

```
SELECT pr.Name as [Товар], c.Name as [Категорія] FROM Product pr, Category c
WHERE pr.IdCategory = c.IdCategory;
```

Але даний запис використовує старий стандарт ANSI SQL. Перепишемо його згідно вимог стандартів ANSI SQL2 та вище, тобто з використанням оператора INNER JOIN.

```
SELECT pr.Name as [Товар], c.Name as [Категорія] FROM Product pr
INNER JOIN Category c ON pr.IdCategory = c.IdCategory;
```

Результат буде однаковий. Фактично, відмінність полягає лише в тому, що зв'язки між таблицями вказуються за допомогою оператора INNER JOIN, а зв'язки по ключовим полям описуються після оператора ON. Всі інші оператори діють так само, як і раніше.

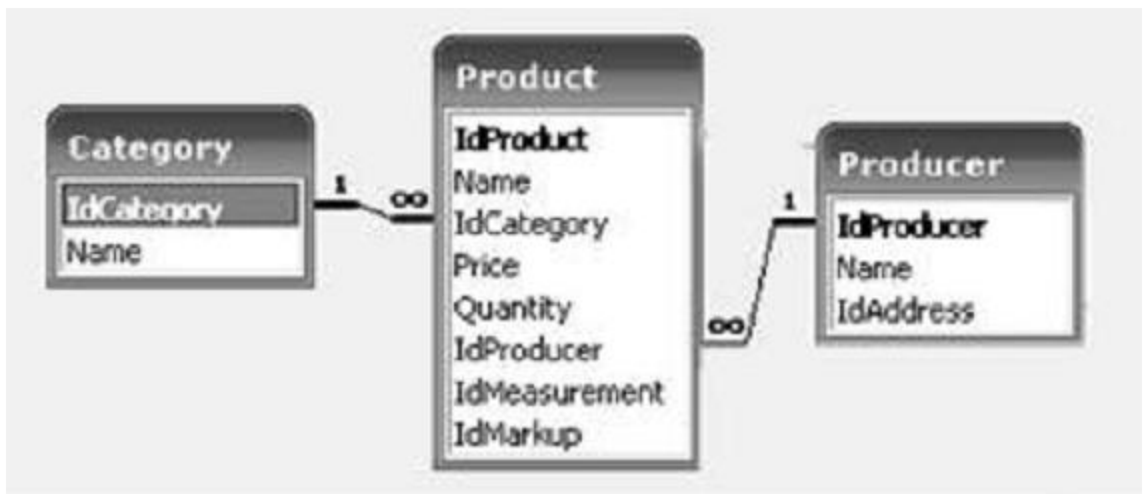
Розглянемо ще один приклад, в якому в зв'язку буде приймати участь більше двох таблиць. Додамо до нашого запиту інформацію про виробника товару. Згідно старого стандарту ANSI SQL такий запит буде виглядати так:

```
SELECT Product.Name as [Товар], Category.Name as [Категорія],
        Producer.Name as [Виробник]
FROM Product, Category, Producer
WHERE Product.IdProducer=Producer.IdProducer
AND Product.IdCategory = Category.IdCategory;
```

Згідно стандарту SQL2 і вище:

```
SELECT Product.Name as [Товар], Category.Name as [Категорія],
        Producer.Name as [Виробник]
FROM Category INNER JOIN (Product INNER JOIN
Producer ON Product.IdProducer=Producer.IdProducer) ON
Product.IdCategory=Category.IdCategory;
```

Отже, при використанні нового стандарту ANSI SQL2 слід лише просто запам'ятати один принцип: при зв'язку таблиць, вони повинні утворювати суцільний послідовний ланцюг. В нашому випадку він наступний:



Щодо накладання умови, то тут нічого не змінилось, умова повинна розміщуватись в інструкції WHERE. Отже, накладемо умову на виведення товарів лише категорії "Бакалія":

```
SELECT Product.Name as [Товар], Category.Name as [Категорія],
    Producer.Name as [Виробник]
FROM Category INNER JOIN (Product INNER JOIN
Producer ON Product.IdProducer=Producer.IdProducer) ON
Product.IdCategory=Category.IdCategory
WHERE Category.Name='Бакалія';
```

3.3. Зовнішні об'єднання (OUTER JOIN) та його типи: ліве, праве та повне

При використанні оператора INNER JOIN СУБД шукає та виводить записи, які задовольняють критеріям пошуку для двох або кількох таблиць. Але що робити у випадках, коли необхідно знайти записи однієї таблиці, відповідностей яких немає в іншій?

Наприклад, потрібно відобразити інформацію про всі товари та постачальників, що їх поставляли.

```
SELECT Supplier.Name AS Постачальник, Product.Name
FROM Supplier INNER JOIN (Product INNER JOIN
Delivery ON Product.IdProduct = Delivery.IdProduct)
ON Supplier.IdSupplier = Delivery.IdSupplier;
```

Результат:

Запрос10 : запрос на выборку

Постачальник	Name
ТзОВ "Вмерти, але доставити"	Фарш "315км/год"
ПП "Швидкий вітер"	Фарш "315км/год"
ПП "Швидкий вітер"	Горілка "Чіполіно"
"International Road" Co.	Молоко "Успевайка"
ТзОВ "Бистринка"	Горілка "Чіполіно"
"International Road" Co.	Банани
ПП "Швидкий вітер"	Сухаріки "Дубовые дровишки"
ТзОВ "Бистринка"	Ковбаса "Роспізнай"
ЗАТ "Хвилинка"	Цукерки "Крабіки"
ТзОВ "Хрещатик"	Картопля "Зелене Чудо"
ТзОВ "Хрещатик"	Цукерки "Крабіки"

Як видно з результатів запиту, на екрані відобразились лише ті товари, які поставлялись і про яких існує інформація в базіданих. Щоб відобразити постачальників, інформація про яких присутня в базі даних, не залежно від того, поставляв він вже якийсь товар в магазин чи ні, неопотрібно скористатись зовнішнім об’єднанням таблиць.

Зовнішні об’єднання здійснюються за допомогою оператора OUTER JOIN та використовуються в випадку, коли потрібно, щоб запит повертав всі записи з однієї або більше таблиць, незалежно від того, чи мають вони відповідні записи в іншій таблиці. Фактично, вони дозволяють обмежити кількість повертаємих полів однієї таблиці, не обмежуючи при цьому їх для іншої таблиці.

Стандарт ANSI виділяє наступні типи зовнішніх об’єднань та оператори, що їх здійснюють:

1. LEFT OUTER JOIN. Ліве об’єднання - записи першої таблиці (зліва) включаються в результуючу таблицю повністю, а з другої таблиці (справа) в результат включаються лише ті, що мають пару в першій таблиці. В якості пари для записів першої таблиці, які не мають пари в іншій використовують пусті (NULL) поля.
2. RIGHT OUTER JOIN. Праве об’єднання - навпаки.
3. FULL OUTER JOIN. Повне об’єднання - включає всі співставляємі і неспівставляємі записи з обох таблиць.

Але СУБД MS Access підтримує лише два перших види зовнішніх об’єднань: ліве і праве.

Для демонстрації роботи зовнішніх об’єднань перепишемо вищерозглянутий запит таким чином, щоб він виводив список постачальників, інформація про яких присутня в базі даних, не залежно від того, поставляв він вже якийсь товар в магазин чи ні:

```
SELECT Supplier.Name, Product.Name
FROM Supplier LEFT OUTER JOIN  (Delivery   LEFT OUTER JOIN
Product ON Product.IdProduct  = Delivery.IdProduct)
ON Supplier.IdSupplier  = Delivery.IdSupplier;
```

Результат:

Запрос9 : запрос на выборку

Supplier.Name	Product.Name
ПП Вася	
ЗАТ "Хвилинка"	Цукерки "Крабіки"
ТзОВ "Хрещатик"	Картопля "Зелене Чудо"
ТзОВ "Хрещатик"	Цукерки "Крабіки"
ПП Кулаков В.В.	
ТзОВ "Вмерти, але доставити"	Фарш "315км/год"
ПП "Швидкий вітер"	Фарш "315км/год"

В результаті запиту, якщо постачальник не постачав ще товар в магазин, йому відповідає NULL-значення в полі назви поставляємого товару.

І навпаки. Якщо необхідно відобразити повний список товарів, з інформацією про їх постачальників, незалежно від того відома про них інформація чи ні:

```
SELECT Supplier.Name AS Постачальник, Product.Name AS Товар
FROM Supplier RIGHT OUTER JOIN  (Product RIGHT OUTER JOIN
Delivery ON Product.IdProduct  = Delivery.IdProduct)
ON Supplier.IdSupplier  = Delivery.IdSupplier;
```

Такий запит дозволить одразу виявити товари, інформацію про постачальників яких не заповнили. В нашій базі даних передбачено, щоб такої ситуації не трапилось, адже наявність такої інформації є важливим.

3.3. Самооб’єднання таблиць

Аналогічно об’єднанню кількох таблиць, таблицю можна об’єднати саму з собою. Такий вид об’єднання носить назву самооб’єднання. Це може знадобитись, коли Вам будуть потрібні зв’язки між рядками однієї і тієї ж таблиці. Наприклад, наступний запит виведе інформацію про виробників, назви яких починаються з «ЗАТ», тобто форма відповідальності яких Закрите Акціонерне Товариство:

```
SELECT p1.Name
FROM Producer p1, Producer p2
WHERE p1.IdProducer=p2.IdProducer AND p1.Name LIKE 'ЗАТ*';
```

Або:

```
SELECT p1.Name
FROM Producer p1 INNER JOIN Producer p2    ON        p1.IdProducer
=p2.IdProducer WHERE p1.Name LIKE 'ЗАТ*';
```

Результат:

Producer : таблица

IdProducer	Name	IdAddress
26	"MicroChips" Ltd.	Вашингтон
35	Banana Republica	Вашингтон
21	АТ "Русская водка"	Москва
30	ВАТ "Карась"	Петрозаводськ
24	ВАТ "Конфеті"	Мінськ
34	ВАТ "Росинка"	Київ
22	ВАТ "Сальце України"	Рівне
31	ЗАТ "ДПС"	Рівне
29	ЗАТ "Картошка"	Мінськ
32	ЗАТ "Рівне-Хліб"	Рівне
33	ЗАТ "Румянець"	Рівне
28	ЗАТ "Яблуко"	Омськ
23	ПП "Корівка"	Варшава

Запрос7 : запрос н...

Name
ЗАТ "Яблуко"
ЗАТ "Картошка"
ЗАТ "ДПС"
ЗАТ "Рівне-Хліб"
ЗАТ "Румянець"

Запись: 5

В такому запиті для таблиці Producer ми визначили два різних псевдоніми, тобто ми повідомляємо саму СУБД, що ми хочемо мати дві різні таблиці, які повинні містити однакові дані. Після цього ми їх об'єднуємо так само, як і будь-які інші таблиці. А далі отримуємо записи, що задовольняють умову.

Спочатку це може здатись незвичним, але при роботі з кількома таблицями в запитах ідея самооб'єднання таблиць не повинна викликати великих труднощів.

Порядок виконання роботи

1. Для обраної предметної області навести приклади використання всіх описаних вище типів обєднань(Внутрішнього, зовнішнього лівостороннього та зовнішнього правостороннього).
2. Для обраної предметної області навести приклади використання EXIST, ANY, SOME, ALL
3. Для обраної предметної області навести приклади використання UNION та UNION ALL
4. Приклади використання підзапитів

ЛАБОРАТОРНА РОБОТА 4.
ТЕМА: СТВОРЕННЯ ДОДАТКУ ДОСТУПУ ДО БД НА МОВІ ПРОГРАМУВАННЯ ВИСОКОГО РІВНЯ.
МЕТА: НАБУТИ ПРАКТИЧНИХ НАВИКІВ СТВОРЕННЯ ДОДАТКУ ДОСТУПУ ДО БД НА МОВІ ПРОГРАМУВАННЯ ВИСОКОГО РІВНЯ.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Зазвичай для доступу і маніпулювання даними використовується відповідна СУБД. Проте часто виникає необхідність дістати доступ до інформації, яка знаходиться в базі даних, з прикладної програми. Вирішити цю задачу можна за допомогою компонентів доступу до даних. C++ Builder надає в розпорядження програміста компоненти, використовуючи які можна побудувати додаток, що забезпечує роботу практично з будь-якою базою даних.

Компоненти доступу до даних знаходяться у вкладках BDE, Data Access, ADO і Interbase. Компоненти вкладок BDE і Data Access для доступу до даних використовують процесор баз даних Borland Database Engine (BDE), реалізований у вигляді набору динамічних бібліотек і драйверів. Компоненти вкладки ADO для доступу до даних використовують розроблену технологію Microsoft ADO (ACTIVEX Data Object — ADO). Компоненти вкладки Interbase забезпечують безпосередній доступ до даних Interbase. Найбільш універсальним механізмом доступу до баз даних є механізм, реалізований на основі BDE. Драйвери, що входять до складу BDE, забезпечують доступ як до локальних баз даних (Paradox, Access, dbase), так і до видалених серверів баз даних (Microsoft SQL Sewer, Oracle, Infomix). Набор драйверів, включених в BDE визначається варіантом C++ Builder.

У перших версіях C++ Builder основою роботи з базами даних являвся Database Engine (BDE) — процесор баз даних фірми Borland. BDE служить посередником між додатком і базами даних. Він пропонує користувачу єдиний інтерфейс для роботи, що розв'язує користувача від конкретної реалізації бази даних. Завдяки цьому не треба міняти додатки при зміні реалізації бази даних. Додаток C++ Builder звертається до бази даних через BDE. Додаток C++ Builder, коли йому потрібно зв'язатися з базою даних, звертається до BDE і повідомляє звичайно псевдонім бази даних і необхідну таблицю у ній. BDE реалізований у вигляді динамічно приєднуваних бібліотек DLL. Вони, як і будь-які бібліотеки, забезпечені API (Application Program Interface — інтерфейсом прикладних програм), названим IDAPI (Integrated Database Application Program Interface). Це список процедур і функцій для роботи з базами даних, яким і користуються додатки. BDE по псевдоніму знаходить відповідний для вказаної бази даних драйвери. Драйвер — це допоміжна програма, яка розуміє, як спілкуватися з базами даних певного типу. Якщо в BDE є власний драйвер відповідної СУБД, то BDE зв'язується через нього з базою даних і з потрібною таблицею в ній, обробляє запит користувача і повертає в додаток результати обробки. BDE

підтримує природний доступ до таких баз даних, як Microsoft Access, FoxPro, Paradox, dBase. Якщо власного драйвера потрібної СУБД в BDE немає, то використовується драйвер ODBC. ODBC (Open Database Connectivity) — це DLL, аналогічна по функціях BDE, але розроблена фірмою Microsoft.

BDE підтримує SQL — стандартизовану мову запитів, що дозволяє обмінюватися даними з SQL-серверами, такими, як Sybase, Microsoft SQL, Oracle, Interbase. Ця можливість використовується особливо широко при роботі на платформі клієнт/сервер і в розподілених базах даних. Починаючи з C++ Builder 5 введена інша альтернативна можливість роботи з базами даних, минувши BDE. Це розроблена в Microsoft технологія ActiveX Data Objects (ADO). ADO — це призначений для користувача інтерфейс до будь-яких типів даних, включаючи реляційні і не реляційні бази даних, електронну пошту, системні, текстові і графічні файли. Зв'язок з даними здійснюється по засобом так званої технології OLE DB.

Ще один альтернативний доступ до баз даних Interbase був введений у C++ Builder 5 на основі технології InterBase Express (IBX). У бібліотеці компонентів C++ Builder 5 з'явилася сторінка InterBase, що містить компоненти для роботи з InterBase, минувши BDE. Ці компоненти забезпечують підвищену продуктивність і дозволяють використовувати нові можливості серверу InterBase, недоступні звичним компонентам BDE.

Сьогодні ми розпочинаємо знайомство з базами даних у C++ Builder. Ми розглянемо як улаштована архітектура доступу до БД у C++ Builder; як створювати форми для роботи з БД, які компоненти є необхідними для побудови додатків, працюючих з БД.

Поняття бази даних. Структура бази даних

База даних – це набір однорідної та, як правило, упорядкованої за деяким критерієм інформації. БД може бути представлена в „паперовому” чи в „комп’ютерному” вигляді.

Типовим прикладом „паперової” БД є каталог бібліотеки – набір паперових карток, які містять інформацію про книжки. Інформація в цій базі однорідна (містить лише відомості про книги) й впорядкована (картки розміщено, наприклад, в алфавітному порядку прізвищ авторів). Іншим прикладом паперової БД є телефонний довідник чи розклад руху поїздів.

„Комп’ютерна” БД, з точки зору користувача, це програма, що забезпечує роботу з великою кількістю однотипної інформації.

З точки ж зору програміста, БД — це набір зв’язаних файлів, у яких знаходиться інформація (*файли даних*). Створюючи БД для користувача, програміст створює програму, яка забезпечує роботу з файлами даних. Така програма (додаток) називається системою управління базою даних (*СУБД*).

Файл даних складається з *записів*. Кожний запис містить інформацію про один екземпляр БД. В свою чергу, записи складаються з *полів*. Кожне поле містить інформацію про одну характеристику екземпляру БД.

Слід зауважити, що кожний запис складається з однакових полів. Деякі поля можуть бути не заповнені, але вони все одно повинні бути присутні в запису.

Інформацію „комп’ютерних” БД зручно представляти у вигляді таблиці. Кожен рядок таблиці відповідає запису, а кожна комірка таблиці – полю. При цьому заголовок стовпця таблиці – це *ім’я поля*, а номер рядка таблиці – це *номер запису*. Тому часто замість терміну „файл даних” використовують термін „таблиця даних” чи просто „таблиця”.

Локальні і віддалені бази даних

У залежності від розташування самих даних й програми, що використовує ці дані, а також від способу розділення даних між кількома користувачами розрізняють *локальні* та *віддалені* бази даних.

Якщо додаток роботи з БД (СУБД) і саму БД розміщено на одному комп’ютері, то в цьому випадку маємо локальну БД. Очевидною перевагою локальної БД є висока швидкість доступу до інформації. Проте, локальні БД не забезпечують одночасного доступу до інформації кількома користувачами. Т. б., поки дані використовуються одним користувачем, інший користувач не може працювати з цими даними, дані для нього закриті, заблоковані. Бази даних dBase, Paradox, FoxPro, Microsoft Access це локальні бази даних.

Віддалені бази даних будуються за технологією “клієнт-сервер”. Механізм роботи такої БД забезпечує можливість доступу до даних багатьом користувачам. Oracle, Infomix, Microsoft SQL Server і InterBase — це приклади віддалених баз даних.

Ми говоритимемо лише про локальні БД.

Механізм доступ до бази даних. Псевдоніми DBE

Механізм доступу до баз даних реалізований на основі *процесору баз даних* Borland Database Engine (BDE). До складу C++ Builder включено компоненти для створення програм роботи з файлами даних у форматах dBase, Paradox, Microsoft Access, Infomix, Oracle тощо.

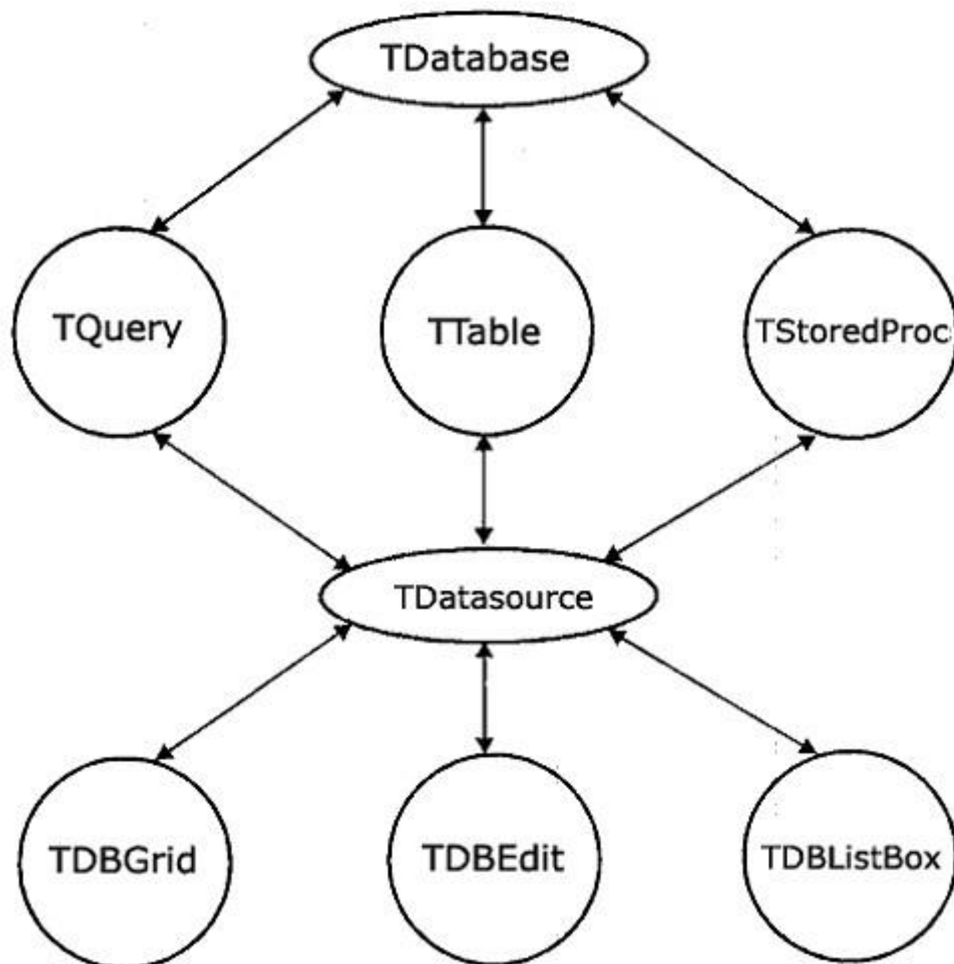
Розробляючи програму роботи з БД, програміст не може знати, на якому диску й у якому каталозі будуть знаходитися файли БД під час її використання. Тому виникає проблема передачі в програму інформації про місце знаходження файлів БД.

У C++ Builder ця проблема вирішується шляхом використання псевдонімів BDE. Псевдонім — це слово для позначення несправжнього ім’я. Псевдонім (Alias) BDE визначає шлях до джерела даних, т. б. до реального, повного імені каталогу, в якому знаходяться файли БД. І програма роботи з БД для доступу до даних з C++ Builder використовує не реальне ім’я каталогу, а псевдонім. Наприклад, псевдонімом каталогу E:\Student\DS\DS01 може бути ім’я Gruppa.

Псевдонім БД можна створити за допомогою утиліти BDE Administrator. Ім’я псевдоніма повинне містити не більше 8 символів – букв латинського алфавіту та цифр. Інформація про всі зареєстровані в системі псевдоніми зберігається в спеціальному конфігураційному файлі.

Компоненти доступу до баз даних

Архітектуру доступу до БД з точки зору додатку C++ Builder можна зобразити схемою:



Ці компоненти знаходяться на укладках Data Access, BDE та Data Controls палітри компонентів.

Компонент `TDatabase` (укладка BDE) представляє БД як єдине ціле, т. б. як сукупність таблиць. Компонент не візуальний. Використання в додатках компонента `TDatabase` не є обов'язковим, але це найкращий спосіб керувати доступом до БД з однієї ключової точки. Якщо ви явно не включили компонент TDatabase в форму, C++ Builder автоматично створить тимчасовий екземпляр.

Компонент `TTable` (укладка BDE) використовується для встановлення зв'язку додатку (форми) з конкретною таблицею БД, т. б. забезпечує доступ до таблиць БД. Компонент не візуальний. Зазвичай використовується для додавання, редагування та видалення даних з бази. Він є центральною частиною схеми доступу до БД у C++ Builder.

Компонент `TDataSource` (укладка Data Access) забезпечує зв'язок між джерелом даних (компонент `TTable`) та компонентом відображення даних (інтерфейсним компонентом). Компонент не візуальний.

Компоненти, що забезпечують відображення й редагування полів БД, знаходяться на укладці Data Controls:

Компонент `TDBGrid` забезпечує перегляд БД в режимі таблиці. Компонент візуальний.

Компоненти TDBEdit (однорядковий елемент редагування), TDBMemo (багаторядковий елемент редагування), TDBText (позначка на формі), TDBImage (ілюстрація) обслуговують конкретний стовпчик з таблиці бази даних. Вони відображають дані, що містяться у стовпчику, й передають назад до таблиці всі зміни, що були внесені користувачем. Компоненти візуальні. Вони подібні до відповідних елементів управління, які ви вже використовували в своїх додатках, але розроблені спеціально для обміну інформацією з базами даних.

Отже, висновки: для створення простої форми вам потрібні лише три компоненти: TTable, TDataSource та один з інтерфейсних (TDBGrid, TDBEdit, TDBListBox, TDBMemo…). Компонент TTable забезпечує базовий зв'язок між формою та таблицею бази даних. Функцією компонента TDataSource є передача даних між TTable й відповідним інтерфейсним компонентом. Інтерфейсні компоненти відповідають за візуальне відображення та редагування даних БД.

Створення форми для роботи з базами даних (вручну)

Сьогодні ми розглянемо, як використовувати компоненти C++ Builder для створення форм, що працюють з БД.

Існує два основних способи побудови форми для роботи з БД. Це можна зробити вручну, по черзі розміщуючи компоненти в форму й зв'язуючи їх один з одним. Чи можна передати цю справу майстру форм БД.

Давайте ми зробимо це вручну, щоб побачити деталі цього процесу.

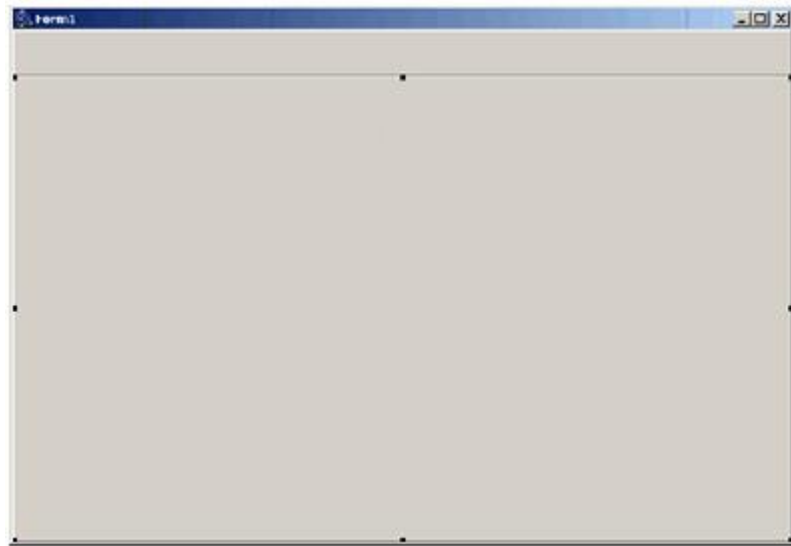
1. Завантажте C++ Builder.

2. Розмістіть в форму два компоненти TPanel (укладка Standard).

Один з них розмістіть поряд із верхнім краєм форми й встановіть для його властивості Align значення alTop. (властивість Align визначає чи залишається компонент незмінним при зміні розмірів форми, чи змінюється, займаючи певну область). В результаті панель займе верхню частину форми.

Другий компонент розмістіть біля нижнього краю форми й встановіть для його властивості Align значення alClient. В результаті панель займе всю вільну область форми.

3. Видалимо текст, що відображується в кожній панелі. Для цього знищить вміст властивостей Caption компонентів.



4. Помістіть на нижню панель (TPanel2) компонент TScrollBox (укладка Additional), який дозволить відображати більше інтерфейсних компонентів, ніж вміщує форма. Якщо таблиця має багато полів, то відповідні інтерфейсні компоненти можуть не поміститися у вікні. TscrollBox забезпечить доступ до них шляхом прокручування вмісту вікна під час виконання програми. Встановіть для його властивості Align значення alClient. В результаті TscrollBox займе всю робочу область панелі TPanel2.

Тепер можна переходити до розміщення компонентів, які власне й забезпечують доступ до бази даних.

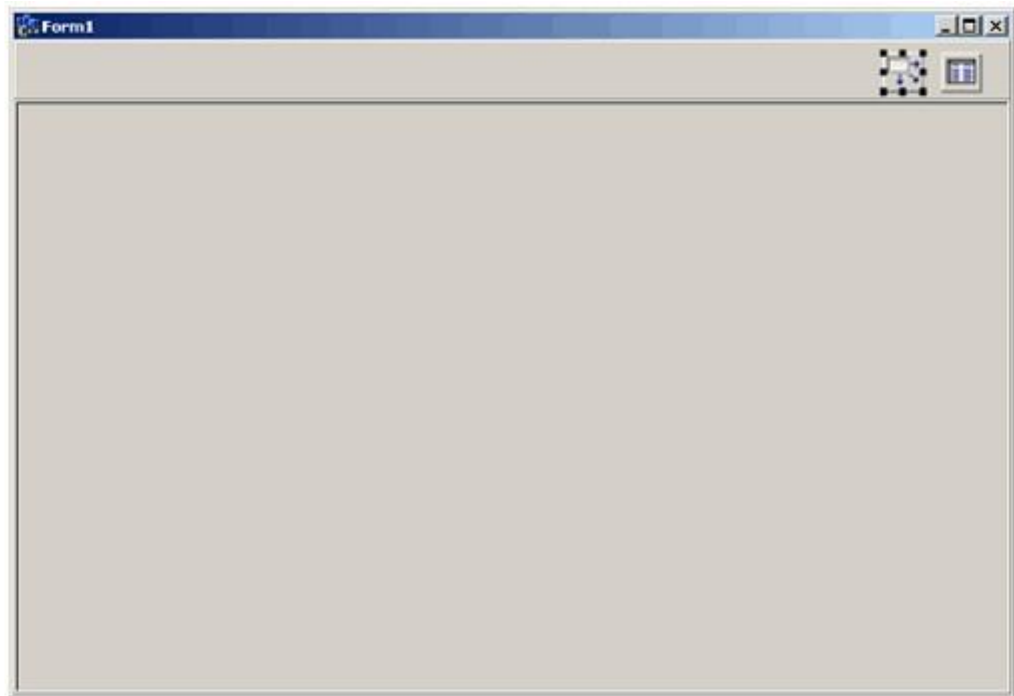
5. Помістіть в довільному місті форми компоненти TTable (укладка BDE) та TDataSource (укладка Data Access). Положення не відіграє ніякої ролі, бо компоненти не візуальні.

6. Встановіть для властивості DatabaseName компонента TTable псевдонім BCDEMOS. Ця властивість надає процесору баз даних (BDE) інформацію, необхідну для пошуку таблиці, що вказана в іншій важливій властивості TableName.

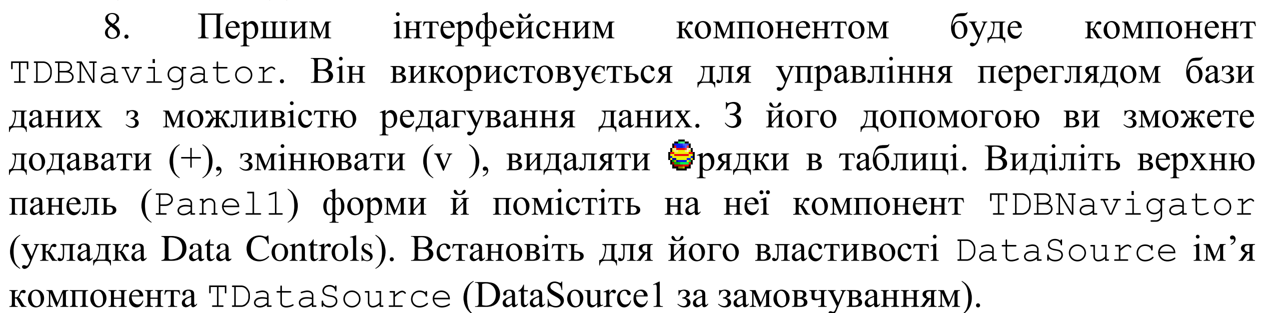
Властивості TableName надайте ім’я таблиці biolife.db, розміщення якої визначається псевдонімом BCDEMOS.

7. Встановіть для властивості DataSet компонента TDataSource ім’я компонента TTable (Table1 за замовчуванням). Це встановить зв’язок між TTable та інтерфейсними компонентами, які обслуговує TDataSource. Т. б., що при обміні даними з інтерфейсними компонентами TDataSource буде посилати чи приймати ці дані від таблиці, що зв’язана з Table1.

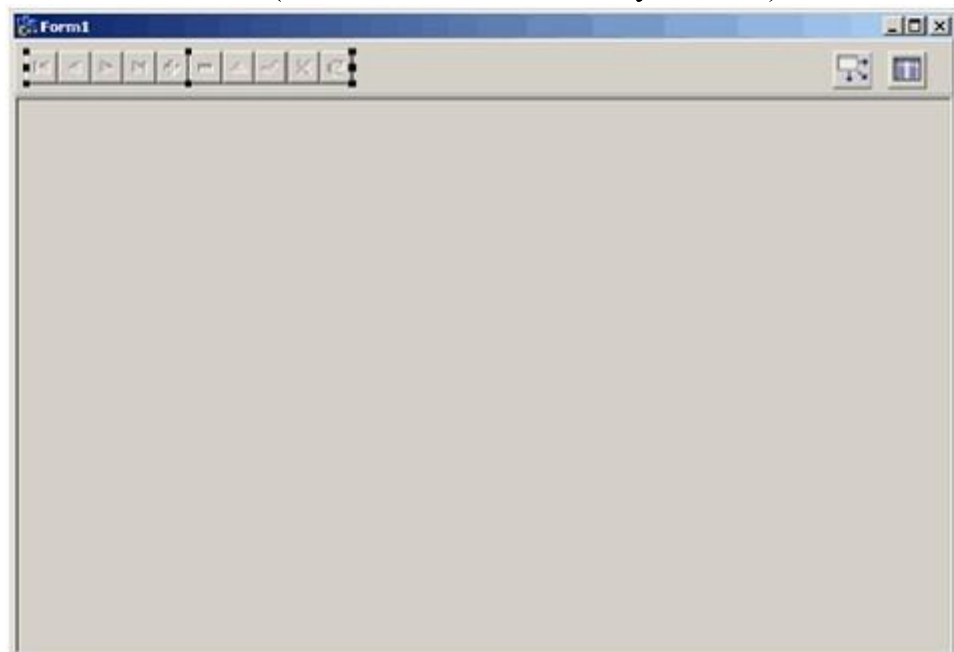
Ще зверніть увагу на властивість AutoEdit. Для неї встановлене значення true (за замовчуванням). Це призводить до автоматичного переключення TDataSource в режим редагування при зміні даних у відповідному інтерфейсному компоненті.



Тепер, коли невізуальні компоненти розміщено, можна переходити до інтерфейсних компонентів, які забезпечують взаємодію користувача з таблицями бази даних.



8. Першим інтерфейсним компонентом буде компонент `TDBNavigator`. Він використовується для управління переглядом бази даних з можливістю редагування даних. З його допомогою ви зможете додавати (+), змінювати (v), видаляти рядки в таблиці. Виділіть верхню панель (`Panel1`) форми й помістіть на неї компонент `TDBNavigator` (укладка Data Controls). Встановіть для його властивості `DataSource` ім'я компонента `TDataSource` (DataSource1 за замовчуванням).

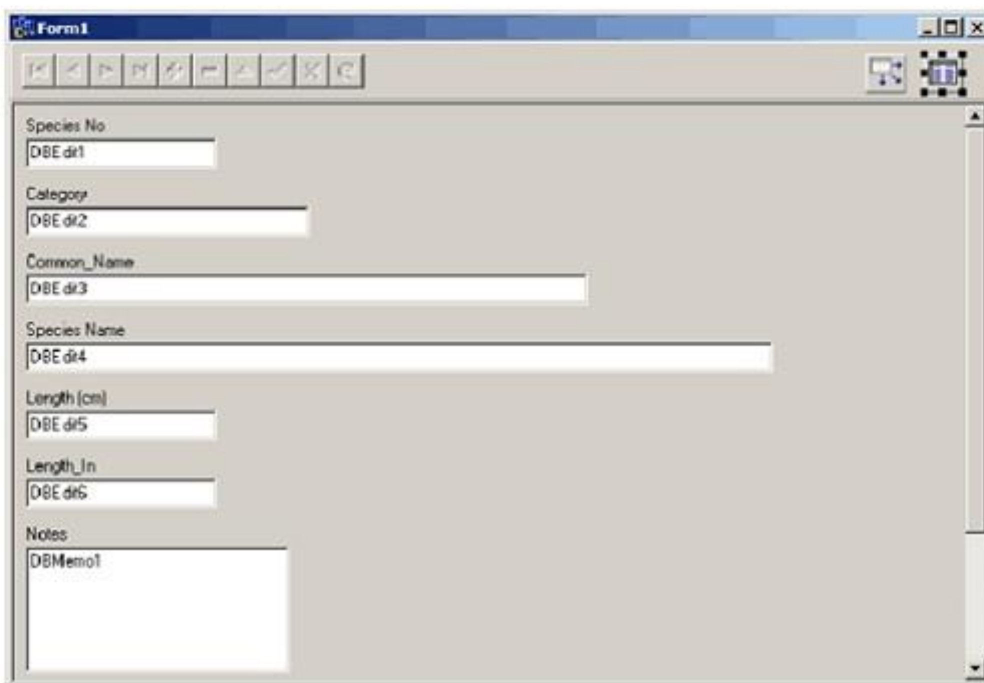


Тепер можна розміщувати інші інтерфейсні компоненти. Для цього існує два основних способи. Перший більш трудомісткий, оскільки полягає в послідовному розміщенні компонентів (`TDBEdit`, `TDBMemo` тощо) в форму

й встановленні властивостей DataSource та DataField для кожного з них окремо. Після цього залишиться лише розмістити підписи до них (компоненти TLabel чи TDBText). Цей спосіб ви опануєте при виконанні практичної роботи №9.

Ми ж з вами зараз виберемо другий, дещо скоріший спосіб, який передбачає використання редактору полів C++ Builder. Ви зможете перемістити компоненти полів в форму прямо з редактору. C++ Builder самостійно створить відповідні інтерфейсні компоненти й підписи до них.

1. Клікніть правою кнопкою миші на компоненті TTable для виклику контекстного меню.
2. Виберіть в контекстному меню пункт Fields Editor…
3. Клікніть правою кнопкою миші на редакторі полів та оберіть пункт Add fields… (Додати поля).
4. Після відображення списку полів натисніть кнопку OK, щоб додати усі поля.
5. Клікніть правою кнопкою миші на редакторі полів та оберіть пункт Select all, щоб знову виділити всі поля, й перемістіть їх в форму як групу. Після цього ви повинні побачити для кожного поля відповідний інтерфейсний компонент з компонентом TLabel.



Можна сказати, що ваша форма для роботи з базою даних в основному закінчена. Єдине, що залишилося з'ясувати – це те, як і коли відкривати таблицю TTable. Для цього, знов таки, існує два простих способи.

По-перше, ви можете включити властивість Active компонента TTable на етапі розробки. Встановлення значення true для цієї властивості у вікні Object Inspector призведе до відкриття TTable вже під час роботи з ним.

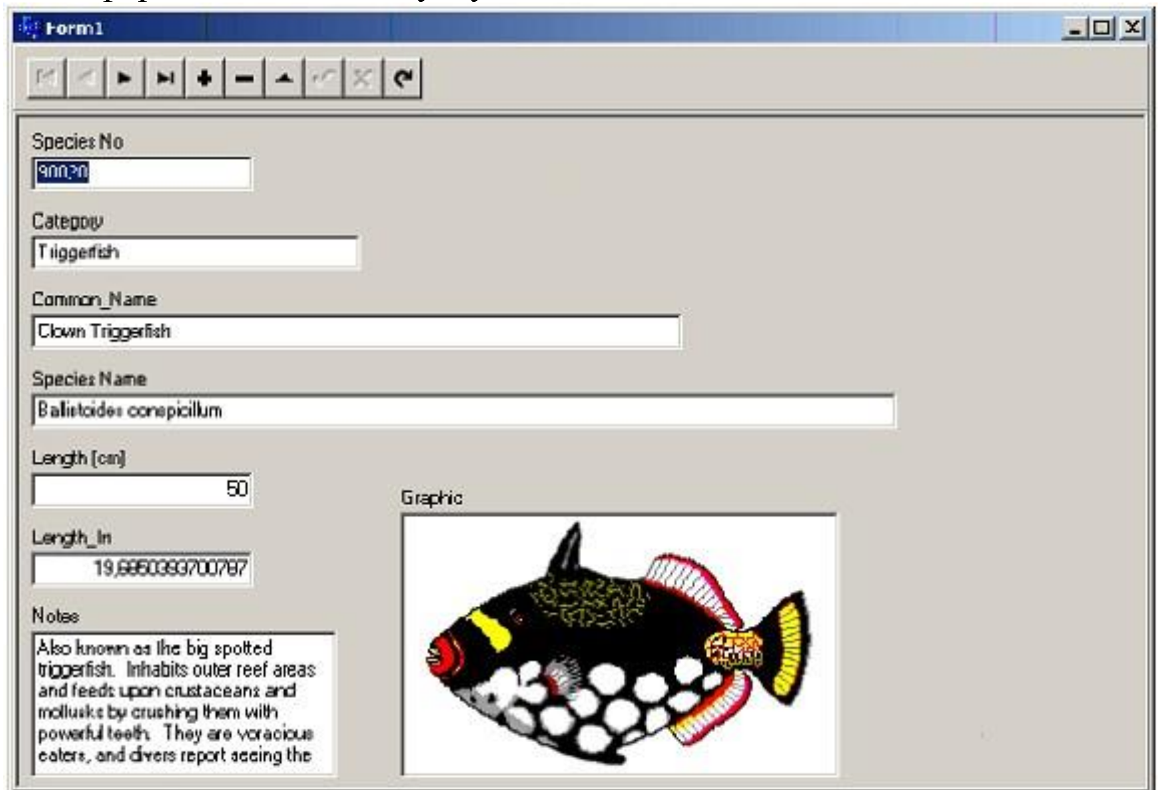
По-друге, ви можете встановити обробник події форми OnCreate чи OnShow, який буде відкривати TTable під час роботи додатку. Саме так

поступає майстер форм – для відкриття таблиці під час виконання програми він генерує обробник події `OnCreate`.

Тому ми також давайте пропишемо обробник події, який буде відкривати таблицю при створенні форми:

```
Table1-
>Open();
```

Ця інструкція призведе до відкриття таблиці при створенні форми. Ось тепер ваша форма готова до запуску.



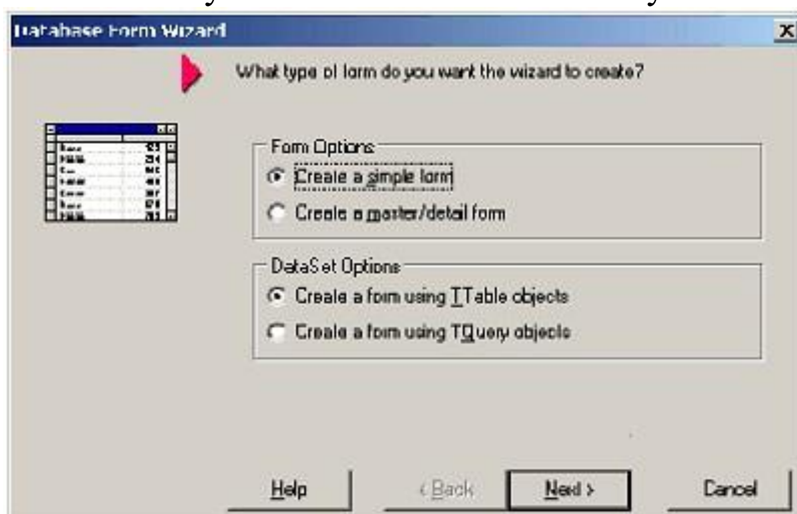
Створення форми для роботи з базами даних (майстром)

Майстер форм передбачений для конструювання форми на основі ваших відповідей на прості питання. Для запуску майстра:

1. Виберіть на панелі інструментів кнопку New
2. Клікніть на укладці Business у діалоговому вікні New Items.

3. Двічі клікніть на значку `Database Form Wizard`.

4. В першому діалоговому вікні майстра запитується, чи хочете ви будувати просту форму чи форму типу головна/підлегла та на яких компонентах повинна бути заснована ваша форма – `TTable` чи `TQuery`. Залиште установки за замовчуванням та натисніть кнопку Next.



5. Далі майстер запитає, яку таблицю ви хотіли б використати в формі. Почніть з вибору псевдоніму `DBDEMOS` у списку `Drive` or `Alias` name (Ім’я диску чи псевдоніма).Цим самим ви вкажете сервер БД чи місце на диску, де розміщені таблиці, до яких передбачається доступ. Після вибору `DBDEMOS` у списку імен таблиць будуть перераховані всі таблиці, які розміщені в тому каталозі, на який посилається псевдонім `DBDEMOS`. Двічі клікніть на таблиці biolife у верхній частині списку. Для продовження натисніть кнопку Next.

6. Далі ви побачите список полів таблиці biolife. Ви можете вибирати по одному за допомогою кнопки > , чи всі одразу, використавши кнопку >>. Для продовження натисніть кнопку Next.

7. Далі майстер попросить вибрати орієнтацію полів у новій формі. Вам пропонується три варіанти: `Horizontal` (Горизонтально), `Vertical`

(Вертикально) та `In a grid` (Сітка). Оберіть та натисніть кнопку Next для продовження.

8. В останньому вікні майстер форм запитує, чи хочете ви зробити форму головною формою додатку. (Головна форма додатку – це та форма, яка відображається на екрані відразу після запуску додатка). Залишимо установку за замовчуванням та дозволимо новій формі стати головною. Крім того, в цьому вікні запитується, чи хочете ви створити лише форму чи форму разом із відповідним модулем даних. Залишимо установку за замовчуванням Form Only (Тільки форма).

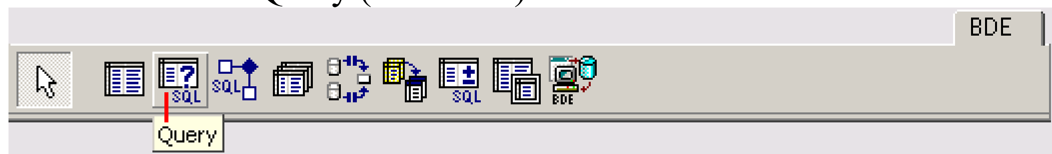
9. Клікніть на кнопці Finish. Ви маєте побачити нову форму, завантажену в редактор форм C++ Builder.

Вибір інформації з бази даних

При роботі з базою даних користувача, як правило, цікавить не весь її вміст, а деяка конкретна інформація. Знайти потрібні відомості можна послідовним переглядом записів. Проте такий спосіб пошуку незручний і малоефективний.

Більшість систем управління базами даних дозволяють виконувати вибірку потрібної інформації шляхом виконання запитів. Користувач формулює запит, указуючи критерій, якому повинна задовольняти інформація, що цікавить його, а система виводить записи, що задовольняють запиту.

Для вибірки з бази даних записів, що задовольняють деякому критерію, призначений компонент Query (мал. 5.17).



Компонент Query

Компонент Query як і компонент Table є записами бази даних, але на відміну від останнього він представляє не всю базу даних (всі записи), а тільки її частина — записи, що задовольняють критерію запиту.

У таблиці перераховані деякі властивості компоненту Query.

Властивості компоненту Query

Властивість	Визначає
Name	Ім'я компоненту. Використовується компонентом Datasource для зв'язку результату виконання запиту (набору записів) з компонентом, що забезпечує проглядання записів, наприклад Dbgrid
SQL	Записаний на мові SQL запит до бази даних (до таблиці)
Active	При привласненні властивості значення true активізується процес виконання запиту
Recor	Кількість записів, що задовольняють критерію запиту

dcount	

Для того, щоб під час розробки програми задати, яка інформація має бути виділена з бази даних, у властивість SQL треба записати запит — команду на мові SQL (Structured Query Language, мова структурованих запитів).

У загальному вигляді SQL-запрос на вибірку даних з бази даних (таблиці) виглядає так:

```
SELECT  Спісокполей
 FROM Таблиця
 WHERE
(Критерій)
 ORDER BY  Спісокполей
```

де:

- SELECT — команда "вибрати з таблиці записи і вивести вміст полів, імена яких вказані в списку";
- FROM — параметр команди, який визначає ім'я таблиці, з якої потрібно зробити вибірку;
- WHERE — параметр, який задає критерій вибору. У простому випадку критерій — це інструкція перевірки вмісту поля;
- ORDER BY — параметр, який задає умову, відповідно до якої будуть впорядковані записи, що задовольняють критерію запиту.

Наприклад, запит

```
SELECT Date_f, Task_f
FROM ':organizer:org.db'
WHERE ( Date_f = '09.02.2003')
 ORDER BY Date_f
```

забезпечує вибірку записів з бази даних organizer (з таблиці org.db), у яких в полі Date_f знаходиться текст 09.02.2003, тобто формує список заходів, призначених на 9 лютого 2003 року.

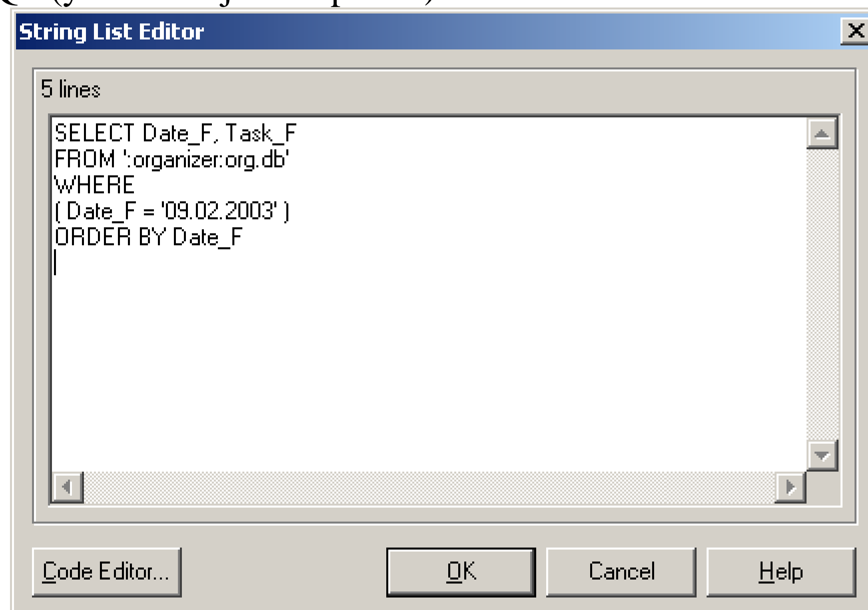
Інший приклад. Запит

```
SELECT Date_f, Task_f
 FROM ':organizer:org.db1
WHERE
( Date_f >= '10.02.2003') AND ( Date_f <= 46.02.2003')
ORDER BY Date_f
```

формує список справ, призначених на тиждень (з 10 по 16 лютого 2003 року).

Запит може бути сформований і записаний у властивість SQL компоненту Query під час розробки форми або під час роботи програми.

Для запису запиту у властивість SQL під час розробки форми використовується редактор списку рядків (мал. 5.18), вікно якого відкривається в результаті клацання на кнопці з трьома крапками в рядку властивості SQL (у вікні Object Inspector).



Введення SQL-запроса під час розробки форми додатку

Сформувати запит під час роботи програми можна за допомогою методу Addзастосувавши його до властивості SQL компоненту Query.

Нижче приведений фрагмент коди, яка формує запит (тобто записує текст запиту у властивість SQL компоненту Query) на вибір інформації з таблиці org бази даних organizer. Передбачається, що строкова змінна today (тип Ansistring) містить дату у форматі dd/mm/yyyy.

```
Form1->query1->sql->add("SELECT Date_f, Task_f");
Form1->query1->sql->add("FROM ':organizer:org.db'");
Form1->query1->sql->add("WHERE (Date_f = '" + today + "')");
Form1->query1->sql->add("ORDER BY Date_f");
```

Якщо запит записаний у властивість SQL компоненту Query під час розробки форми додатку, то під час роботи програми критерій запиту можна змінити простою заміною відповідного рядка тексту запиту.

Наприклад, для запиту:

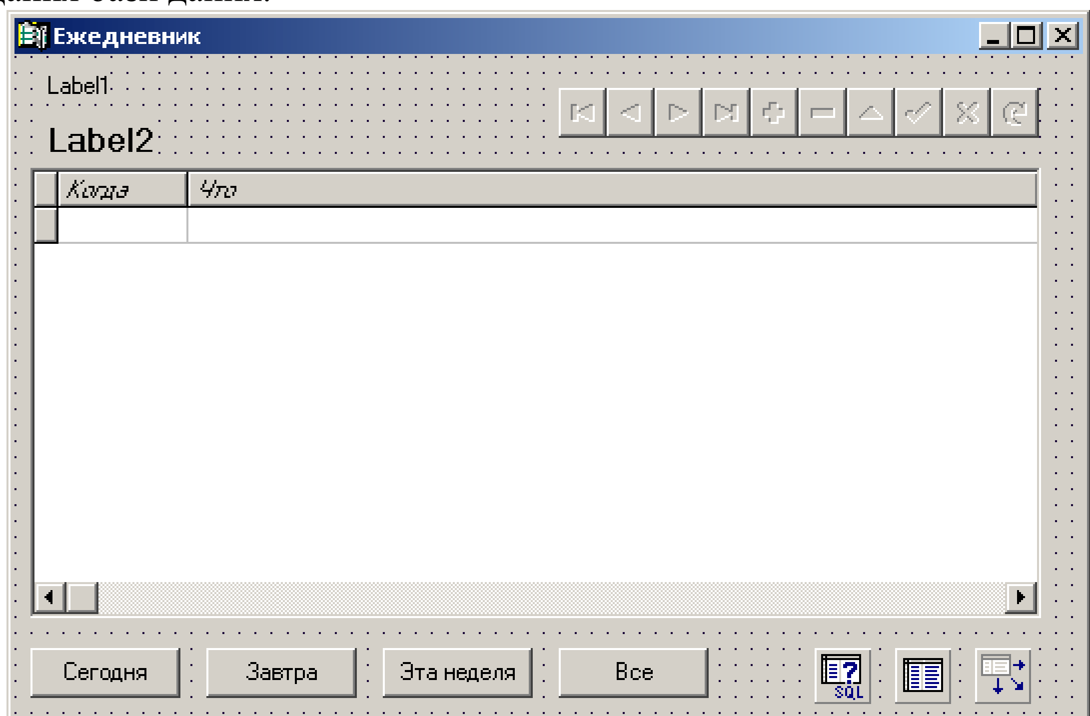
```
SELECT Date_f, Task_f
FROM ':organ!zer:org.db'
 WHERE
```

(Date_f = '09.02.2003')
ORDER BY Date_f

інструкція заміни критерію виглядає так:
Queryl->sql->strings[3] = "(Date_f = '" + tomorrow + "')";

Слід звернути увагу на те, що властивість SQL є структурою типу Tstrings у якого рядка нумеруються з нуля.

Для того, щоб користувач міг вибирати інформацію з бази даних, у форму застосування, що розробляється, треба додати кнопки Сьогодні, Завтра, Цей тиждень і Все (мал). Призначення цих кнопок очевидне. Також у форму додано два компоненти Label. Поле Label1 використовується для відображення поточної дати. У полі Label2 відображається режим проглядання бази даних.



Остаточний вид форми

Функції обробки події click на кнопках Сьогодні, Завтра і Цей тиждень приведені в лістингу. Кожна з цих функцій змінює відповідним чином сформований під час розробки форми SQL-запит. Для отримання поточної дати функції звертаються до стандартної функції NOW яка повертає поточну дату і час. Перетворення дати в рядок символів виконує стандартна функція Formatdatetime.

Лістинг Обробка події Click на кнопках Сьогодні, Завтра і Цей тиждень

```
// Клацання на кнопці Сьогодні
void __fastcall Tforml::buttonlclick(Tobject *Sender)
{
Ansistring today = Formatdatetime("dd/mm/yyyy",now());
```

```
Forml->label2->caption = "Сьогодні";
// змінити критерій запиту
Queryl->sql->strings[3] = "(Date_f = '" + today + "')";
// виконати запит Forml->queryl->open();
Forml->datasourcel->dataset = Forml->queryl;
if ( ! Forml->queryl->recordcount)
{
Showmessage("На сьогодні ніяких справ не заплановано!");
} }
// клацання на кнопці Завтра
void __fastcall Tforml::button2click(Tobject *Sender)
{ <
Ansistring tomorrow = Formatdatetime("dd/mm/yyyy", Now() +1);
Forml->label2->caption = "Завтра";
// змінити критерій запиту
Queryl->sql->strings[3] = "(Date_f = '" + tomorrow + "')";
// виконати запит Forml->queryl->open();
Forml->datasourcel->dataset = Forml->queryl;
if ( ! Forml->queryl->recordcount)
{
Showmessage("На завтра ніяких справ не заплановано!");
} }
// клацання на кнопці Цей тиждень
void __fastcall Tforml::button3click(Tobject *Sender)
{
// від поточного дня до кінця тижня (до воскресіння)
Tdatetime Present
Endofweek;
Label2->caption = "На цьому тижні";
Present= Now(); // Now — повертає поточну дату
// для доступу до Startofweek, Endofweek, Yearof і Weekof
// треба підключити Dateutils.hpp (див. директиви tfinclude)
// ************************
Endofweek = Startofaweek(Yearof(Present),weekof(Present)+1);
Queryl->sql->strings[3] =
"(Date_f >= '"+
Formatdatetime("dd/mm/yyyy",present)+"')
AND " + "(Date_f<  '"+ Formatdatetime("dd/mm/yyyy",endofweek)+"'}";
 Queryl->0pen();
if ( Queryl->recordcount) {
Datasourcel->dataset = Forml->queryl;
 } else
Showmessage("На цей тиждень ніяких справ не заплановано.");
 }
```

В результаті клацання на кнопці Все в діалоговому вікні програми повинно бути виведено весь вміст бази даних. Базу даних представляє компонент Table1. Тому функція обробки події click на кнопці Все просто "перемикає" джерело даних на таблицю.

```
Обробка події на кнопці Все
// Клацання на кнопці Все
void__fastcall Tforml::button4click(Tobject *Sender)
{
// встановити: джерело даних — таблиця
// таким чином, отобажаєтся вся БД
Forml->datasourcel->dataset = Forml->tablel;
Label2->caption = "Все, що намічене зробити"; }
```

Програма "Ежедневник" спроектована таким чином, що при кожному її запуску в діалоговому вікні виводиться поточна дата і список справ, запланованих на цей і найближчі дні. Виведення дати і назви дня тижня в полі Label виконує функція обробки події onactivate (її текст приведений в лістингу). Ця ж функція формує критерій запиту до бази даних, що забезпечує виведення списку завдань, вирішення яких заплановане на сьогодні (в день запуску програми) і на завтра. Якщо програма запускається в п'ятницю, суботу або воскресіння, то завтрашнім днем вважається понеділок. Такий підхід дозволяє зробити попереджуюче нагадування, адже, можливо, що користувач не включить комп'ютер у вихідні дні.

```
Функція обробки події Onactivate
Ansistring stday[7] = ("воськресенье","понедельник"
"вівторок", "середовище"
"четверг","пятніца","суббота"};
Ansistring stmonth[12] = {"января","февраля","марта"
"апреля","мая","іюня","іюля"
 "августа","сентября"
"жовтня", "ноября","декабря"};
// активізація форми
void _fastcall Tforml::formactivate(Tobject *Sender)
{
Tdatetime Today,   // сьогодні
Nextday;  // наступний день Сні обов'язково завтра)
Word Year, Month, Day; // рік, місяць, день
Today = Now ();
Decodedate(Today, Year, Month, Day);
Labell->caption =  "Сьогодні " + Inttostr(Day)+
  " " + stmonth[Month-l]+ " " +
  Inttostr(Year)+ " роки " + stday[Dayofweek(Today)-1];
Label2->caption = "Сьогодні і найближчі дні";
// обчислимо наступний день
```

```
// якщо сьогодні п'ятниця, то, щоб не забути
// що заплановане на понеділок, вважаємо, що наступний *
// день — понеділок
switch ( Dayofweek(Today)) {
case 6 : Nextday = Today + 3; break; // сьогодні п'ятниця
case 7 : Nextday = Today + 2; break; // сьогодні субота
default : Nextday = Today + 1; break;
}
// запит до бази даних: чи є справи, заплановані
 // на сьогодні і наступного дня
Queryl->sql->strings[3] =
"(Date_f >= '"+ Formatdatetime("dd/mm/yyyy",today)+"')
AND " + "(Date_f<= '"+
 Formatdatetime("dd/mm/yyyy",nextday)+"')";
Queryl->open();
Datasourcel->dataset = Forml->queryl;
if ( ! Queryl->recordcount)
{
Showmessage("На сьогодні і найближчі дні ніяких справ не заплановано.");
}
}
```

Використання псевдоніма для доступу до бази даних забезпечує незалежність програми від розміщення даних в системі, дозволяє розміщувати програму роботи з даними і базу даних на різних дисках комп'ютера, у тому числі і на мережевому. Разом з тим для локальних баз даних типовим рішенням є розміщення бази даних в окремому підкаталозі того каталога, в якому знаходиться програма роботи з базою даних. Таким чином, програма роботи з базою даних "знає", де знаходяться дані. При такому підході можна відмовитися від створення псевдоніма при допомозі Database Desktop і покласти завдання створення псевдоніма на програму роботи з базою даних. Очевидно, що такий підхід полегшує адміністрування бази даних.

Як ілюстрація сказаного в лістингу приведений варіант реалізації функції Onactivate яка створює псевдонім для бази даних organizer. Передбачається, що база даних знаходиться в підкаталозі DATA того каталога, в якому знаходиться виконуваний файл програми. Безпосереднє створення псевдоніма виконує функція Addstandardalias якою як параметр передається псевдонім і відповідний йому каталог. Оскільки під час розробки програми не можна знати, в якому каталозі буде розміщена програма роботи з базою даних і, отже, підкаталог бази даних, ім'я каталога визначається під час роботи програми шляхом звернення до функцій Paramstrto) і Extractfilepatch. Значення першої — повне ім'я виконуваного файлу програми, другої, — шлях до цього файлу. Таким чином, процедурі Addstandardalias передається повне ім'я каталога бази даних.

Створення псевдоніма під час роботи програми

```
void__fastcall Tforml::formactivate(Tobject *Sender)
{
Tdatetime Today,   // сьогодні
Nextday;  // наступний день (не обов'язково завтра)
Word Year, Month, Day; // рік, місяць, день
Today = Now (};
Decodedate(Today, Year, Month, Day);
Labell->caption =  "Сьогодні " + Inttostr(Day)+
  " " + stmonth[Month-l]+ " " +
   Inttostr(Year)+ " роки " + stday[Dayofweek(Today)-1];
Label2->caption = "Сьогодні і найближчі дні";
// обчислимо наступний день
// якщо сьогодні п'ятниця, то, щоб не забути
// що заплановане на понеділок, вважаємо, що наступний
// день — понеділок
switch ( Dayofweek(Today)) {
case 6 : Nextday = Today + 3; break; // сьогодні п'ятниця
case 7 : Nextday = Today + 2; break; // сьогодні субота
default : Nextday = Today + 1; break;
}
#define Din_alias
// псевдонім доступу до БД створюється динамічно
// якщо псевдонім створений за допомогою Database Desktop
// або BDE Administrator, директиву tfdefine Din_alias
//треба видалити ("закоментувати")
#ifdef Din_alias // псевдонім створюється динамічно
// створимо псевдонім для доступу до БД
Session->configmode = cmsession;
Session->addstandardalias("organizer"
Extractfilepath(Paramstr(0))+"DATA\\", "PARADOX");
 // база даних "Ежедневник"
// у форматі Paradox
#endif
Forml->tablel->active = true;
 // відкрити таблицю
// запит до бази даних: чи є справи, заплановані
 // на сьогодні і завтра
Queryl->sql->strings[3] =
"(Date_f >= '"+
Formatdatetime("dd/mm/yyyy",today)+"')
 AND " + "(Date_f <= '"+
Formatdatetime("dd/mm/yyyy",nextday)+"')";
Queryl->0pen();
```

```
Datasourcel->dataset = Forml->queryl;
if ( ! Queryl->recordcount)
(
Showmessage("На сьогодні і найближчі дні
ніяких справ не заплановано.");
```

Порядок виконання роботи

1. Створити додаток доступу та маніпулювання даними в двох таблицях попередньо створеної БД.
2. Забезпечити виконання таких операцій, як :
   2.1. створення запису;
   2.2. знищення
   2.3. редагування
   2.4. відображення всієї таблиці.
3. Завдання виконати для однієї таблиці з використанням компоненнту TTable, для іншої з використанням TSQLQuery. В звіті навести скріншоти і лістинг додатків.

ЛАБОРАТОРНА РОБОТА 5.
ТЕМА: МЕТОДИ ПОШУКУ ТА ФІЛЬТРАЦІЇ.
МЕТА: НАБУТИ ПРАКТИЧНИХ НАВИКІВ СТВОРЕННЯ ДОДАТКУ ДЛЯ ПОШУКУ ТА ФІЛЬТРАЦІЇ НА МОВІ ПРОГРАМУВАННЯ ВИСОКОГО РІВНЯ.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Пошук даних

Як приклад для реалізації методів пошуку та фільтрації використаємо мову високого рівня Delphi.

Універсальні методи класу TDataSet

Клас TDataSet є базовим класом компонента, що реалізує функції доступу до БД. Більшість з його методів є абстрактними і реалізуються в нащадках. Він має два методи для пошуку даних: Locate і Lookup. Дані методи шукають запис, що задовольняє заданим умовам.

function Locate(const KeyFields: string; const KeyValues: Variant; Options: TLocateOptions): Boolean; virtual;

function Lookup(const KeyFields: string; const KeyValues: Variant; const ResultFields: string): Variant; virtual;

Різниця між ними в тім, що функція Lookup при пошуку запису переводить курсор на знайдений запис, а Lookup не робить цього. Якщо поля, що указані для пошуку, є індексованими, то пошук провадиться з використанням індексу, що значно прискорює пошук.

Як приклад розглянемо використання даних методів на прикладі TTable у дворівневому додатку. Розробку будемо вести із використанням Delphi5 для кращої сумісності з різними умовами використання цього прикладу, інші версії Delphi мають лише незначні відмінності у бібліотеці візуальних компонент. Для Delphi7 треба в uses для unit Unit1 додати Variants.

Вихідний код даного прикладу записаний у файлі.

На головну форму помістимо компонент Table із закладки Data Access (за замовчуванням буде мати ім'я Table1) та налаштуємо його на взаємодію з таблицею country з демонстраційної бази даних DBDEMOS. Для цього необхідно встановити наступні властивості Table1 у наступній послідовності:

1. DatabaseName ==> DBDEMOS
2. TableName ==> country.db

Тепер можна активізувати компонент Table1 (властивість Active ==> true).

Далі розмістимо на формі компонентів DataSource, що дозволяє зробити дані Table1 доступними для візуальних компонентів відображення даних. Зв'яжемо DataSource1 з Table1, установивши властивість DataSet компонента DataSource1 на Table1.

Розмістимо візуальні компоненти відображення даних зі сторінки Data Controls: DBGrid та DBNavigator. Установимо властивість DataSource обох на DataSource1.

У результаті ми одержимо форму, зображену на рис. 2. Після компіляції і запуску проекту ми зможемо переглядати і редагувати дані таблиці country. Для організації пошуку в таблиці розмістимо на формі компонент Edit1 і дві кнопки. Властивість Caption кнопки Button1 установимо на "Locate", а Button2 "Lookup". Далі в обробнику події OnClick кнопки Button1 організуємо виклик методу Locate за текстом, що наведений нижче:

Пошук даних

N	Capital	Continent	Area	Population
B	La Paz	South America	1098575	7300000
Brazil	Brasilia	South America	8511196	150400000
Canada	Ottawa	North America	9976147	26500000
Chile	Santiago	South America	756943	13200000
Colombia	Bagota	South America	1138907	33000000
Cuba	Havana	North America	114524	10600000
Ecuador	Quito	South America	455502	10600000
El Salvador	San Salvador	North America	20865	5300000
Guyana	Georgetown	South America	214969	800000
Jamaica	Kingston	North America	11424	2500000

Пошук даних

Name	Capital	Continent	Area	Population
Brazil	Brasilia	South America	8511196	150400000
Canada	Ottawa	North America	9976147	26500000
Chile	Santiago	South America	756943	13200000
Colombia	Bagota		1138907	33000000
Cuba	Havana	North America	114524	10600000
Ecuador	Quito	South America	455502	10600000
El Salvador	San Salvador	North America	20865	5300000
Guyana	Georgetown	South America	214969	800000
Jamaica	Kingston	North America	11424	2500000
Mexico	Mexico City	North America	1967180	88600000

Країна Cuba Locate Інші методи

Континент Edit2 Lookup Вихід

```
TForm1.Button1Click(Sender: TObject);
begin
 if not Table1.Locate('Name',Edit1.Text,[]) then   ShowMessage ('Запис не знайдений ...');
end;
```

Виклик функції Locate організує пошук запису в таблиці Country. Перший параметр цієї функції - поля, значення яких потрібно перевіряти. У нашому випадку ми шукаємо запис за одним полем Name. Другий параметр -

це шаблон пошуку і третій опції пошуку. Функція повертає значення типу boolean, що вказує на успішність пошуку.

Для тестування прикладу запустимо програму на виконання, при цьому рядку вводу пишемо Cuba і натискаємо кнопку Locate. Курсор у DBGrid1 повинен переміститися на запис, що має в полі Name уведене значення.

Однак наш приклад має поки один недолік, у рядок редагування необхідно вводити повне ім'я із врахуванням регістру, тому якщо замість Cuba ввести, наприклад Cu чи cuba, то наш пошук не буде результатним. Оскільки це не може нас влаштовувати розглянемо більш докладно опції пошуку. Цей параметр має тип TlocateOptions і дозволяє задавати набір із двох параметрів пошуку: loCaseInsensitive і loPartialKey. Перший з них скасовує чутливість до регістра в текстових полях, а другий дозволяє шукати запис, що частково відповідає заданій умові. Після цих змін обробник подій буде мати такий вид:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if not Table1.Locate('Name',Edit1.Text,[loCaseInsensitive, loPartialKey])then
    ShowMessage('Запис не знайдений ...');
end;
```

Наступною проблемою є пошук запису за кількома полями. Для організації пошуку за іменем країни і назвою континенту додамо на форму ще один компонент Edit2. Код обробника події натискання на кнопку Locate змінимо в такий спосіб:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if not Table1.Locate(
    'Continent;Name',
    VarArrayOf([Edit2.Text,Edit1.Text]), [loCaseInsensitive, loPartialKey])
  then
    ShowMessage('Запис не знайдений ...');
end;
```

Зазначимо, що при пошуку за кількома полями, усі вони перелічуються в параметрі функції Locate

Запустивши додаток, у рядку пошуку континенту пишемо South America, а в рядку "країна" - C. Натискаємо кнопку Lookup - результат пошуку - встановлення курсору в DBGrid на запис Chile. Частковий пошук при пошуку за кількома полями працює лише для останнього поля, зазначеного в списку.

Модифікуємо наш приклад для використання функції Lookup. Нижче наведений код обробника події натискання на кнопку Lookup:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Res:Variant;
begin
```

```
Res:=Table1.Lookup('Continent;Name',VarArrayOf([Edit2.Text,Edit1.Text]),'Area
');
      if Res <> Null then
        ShowMessage('Площа '+String(Res));
    end;
```

Як видно з наведеного тексту, ми проводимо пошук за полями Continent, Name. Якщо запис знайдено, видаємо повідомлення про площу країни, при цьому курсор на знайдений запис не переміщається. На жаль серед параметрів цієї функції відсутній LocateOptions.

У підсумку зазначимо, що функції Locate та Lookup призначені для пошуку в базі даних одного запису, що задовольняє заданим умовам. Дані методи визначені в класі TDataSet як віртуальні і можуть бути перевизначені в класах нащадках. Метод Locate установлює курсор на виявлений запис, Lookup цього не робить. Метод Locate, хоча і з деякими застереженнями, може шукати записи при частково заданому ключі..

Необхідно відзначити, що дуже часто необхідно знайти відразу кілька записів. Для розв'язання таких роду задач потрібно застосовувати інші методи, що надаються класом Ttable, Tquery ..., або проводити фільтрацію.

Методи класу TTable

Компонент TTable призначений для роботи з таблицею база даних. Даний компонент використовується в дворівневих додатках баз даних, або в сервері додатків у трьохрівневій БД. Як правило, використання TTable для роботи з клієнт-серверною БД виявляється менш ефективним, ніж TQuery, оскільки TTable добуває відразу всі записи з таблиці, а в TQuery лише ті, що задовольняють умовам запиту.

Компонент має кілька специфічних методів для пошуку даних. Дані методи використовуються для пошуку тільки за індексованими полями (для dBase і Paradox як мінімум). Усі вони поділяються на дві групи:

- Методи пошуку одного запису
- Методи пошуку діапазону записів.

Розглянемо спочатку першу групу методів. До них відносяться GotoKey, FindKey, GotoNearest, FindNearest. Перші два методи використовуються для пошуку строгої відповідності, інші шукають часткову відповідність.

Як приклад створимо нову форму, на якій розмістимо компоненти Table, DataSource, DBGrid, DBNavigator. Установимо властивості даних компонентів відповідно до таблиці 1. Відзначимо, що таблиця country.db має індексоване поле Name, за якими ми далі організуємо пошук.

Таблиця 1

Table1	
DatabaseName	BCDEMOS
TableName	Country.db
Active	true
DataSource1	
DataSet	Table1
DBNavigator1, DBGrid1	
DataSource	DataSource1

Далі розмістимо на формі компоненти Edit і чотири кнопки, установивши їх властивості Caption у GotoKey, GotoNearest, FindKey та FindNearest відповідно. Нижче наведений код обробників подій натискання на ці кнопки.

```
//Використання GotoKey
procedure TForm1.Button1Click(Sender: TObject);
begin
 with Table1 do
 begin
  EditKey;
  FieldByName('Name').AsString := Edit1.Text;
  if not GotoKey then
   ShowMessage('Запис не знайдений ...');
 end;
end;

// Використання GotoNearest
procedure TForm1.Button2Click(Sender: TObject);
begin
 with Table1 do
 begin
  EditKey;
  FieldByName('Name').AsString := Edit1.Text;
  GotoNearest;
 end;
end;

// Використання FindKey
procedure TForm1.Button3Click(Sender: TObject);
begin
 with Table1 do
 begin
  if not FindKey([Edit1.Text]) then
```

```
    ShowMessage('Запис не знайдений ...');
  end;
end;

// Використання FindNearest
procedure TForm1.Button4Click(Sender: TObject);
begin
  with Table1 do
  begin
    FindNearest([Edit1.Text]);
  end;
end;
```

Методи GotoKey, FindKey ведуть пошук на точну відповідність заданому ключу. Вони повертають значення типу boolean при успіху у пошуку.
Методи GotoNearest, FindNearest проводять пошук першого запису, що хоча б частково відповідає ключу. Вони не повертають значень, оскільки такий вид пошуку завжди буде успішним.

Перед викликом методів GotoKey, GotoNearest необхідно викликати метод EditKey або SetKey, щоб перевести компонент Table у режим редагування ключа пошуку.

Методи пошуку діапазону записів дозволяють відобразити записи таблиці, що лежать у зазначеному діапазоні значень поля. Для таблиць Paradox і dBase дані методи працюють тільки для індексованих полів. До даних методів відносяться SetRangeStart, SetRangeEnd, EditRangeStart, EditRangeEnd, ApplyRange, CancelRange.

Спочатку необхідно установити початок і кінець діапазону викликом функцій SetRangeStart, SetRangeEnd, EditRangeStart, EditRangeEnd, указуючи при цьому значення полів формування діапазону. Потім викликом ApplyRange застосувати зазначений діапазон. Скасування встановленого діапазону дозволяє знову відобразити всі записи таблиці. Воно виконується викликом функції CancelRange.

Для ілюстрації створимо нову форму у нашому проекті, на якій помістимо компоненти Table, DataSource, DBGrid, DBNavigator. Установимо їх властивості як зазначено в таблиці 1. Далі розмістимо два рядки вводу і дві кнопки SetRange і CancelRange. Обробники подій натискання цих кнопок наведені нижче.

```
procedure TForm1.SetRangeClick(Sender: TObject);
begin
  with Table1 do
  begin
    SetRangeStart; {Входимо в режим встановлення початку діапазону}
    FieldByName('Name').AsString := Edit1.Text; {значення початку діапазону}
    SetRangeEnd; { Входимо в режим встановлення кінця діапазону }
```

```
    FieldByName('Name').AsString := Edit2.Text; {значення кінця діапазону}
    ApplyRange; { Застосовуємо зазначений діапазон}
  end;
end;

procedure TForm1.CancelRangeClick(Sender: TObject);
begin
  Table1.CancelRange; {Очищення діапазону}
end;
```

Використання TQuery

Компонент TQuery призначений для добування даних за допомогою операторів мови SQL (Structured query language). Компонент використовується в дворівневих додатках роботи з базами даних і в сервері додатків у трьохрівневих. Він, як правило, застосовується для роботи з клієнт-серверними базами даних.

Сам компонент TQuery не має спеціальних методів пошуку записів, однак, можливості пошуку закладені в мові SQL.Обмеження на прості дані, одержувані пропозицією select, реалізуються за допомогою пропозиції where.

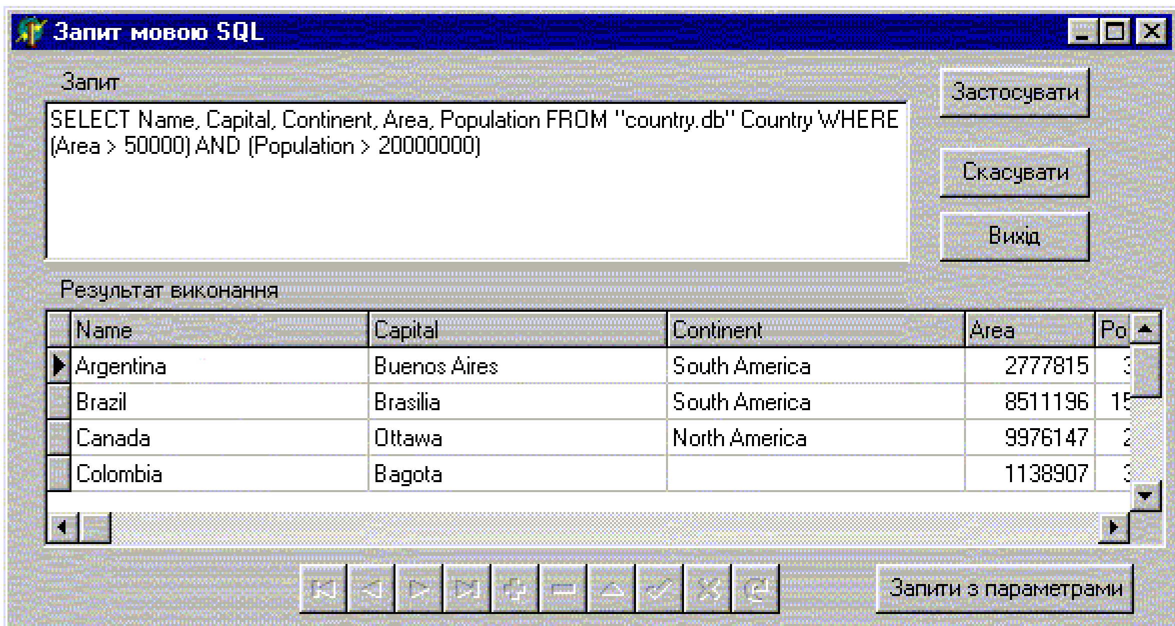
Як ілюстрацію наведемо наступний приклад. Створимо новий додаток, розмістимо на його головній формі компонента: Query, DataSource, DBGrid, DBNavigator, Memo, дві кнопки Button1 і Button2.

Установимо властивості компонентів, як це зазначено в таблиці 2.

Таблиця 2

Query1	
DatabaseName	BCDEMOS
SQL	SELECT Name, Capital, Continent, Area, Population FROM "country.db" Country WHERE (Area > 50000) AND (Population > 20000000)
Active	true
DataSource1	
DataSet	Query1
DBNavigator1, DBGrid1	
DataSource	DataSource1
Button1	
Caption	Застосувати
Button2	
Caption	Скасувати

Вигляд головної форми на етапі розробки наведений на рис. 4.



В обробнику події OnCreate форми реалізуємо копіювання тексту запиту з Query1, код наведений нижче.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Memo1.Text:=Query1.SQL.Text;
end;
```

Кнопка Button1 при натисканні буде переносити текст запиту, виправлений користувачем, з Memo1 назад у Query1. Обробник натискання Button1 буде виглядати так:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Query1.Active:=false;
  Query1.SQL.Text:=Memo1.Text;
  Query1.Active:=true;
end;
```

Обробник події натискання Button2 установимо так, щоб він вказував на FormCreate.

Відкомпілюємо і запустимо нашу програму. Переконаємося в тім, що при зміні тексту запиту (площі і населення) відбувається зміна записів у DBGrid1.

Даний підхід має ряд істотних недоліків, що утрудняють його практичне використання.

1. Користувач додатка зобов'язаний розбиратися в мові SQL;
2. Відсутні засоби автоматичного контролю синтаксису запиту;
3. При зміні тексту запиту, він відправляється на сервер, де щораз аналізується і компілюється, що знижує швидкодію пошуку.
4. Для генерації запиту користувачу необхідно знати імена таблиць і полів бази даних.

Зазначені причини призводять до того, що даний підхід рідко застосовується. Як правило, застосування даного підходу є виправданим при написанні додатків для адміністраторів БД або розробників звітів.

Більш прийнятним є застосування параметризованих запитів. У цьому випадку в тексті запиту містяться параметри для подальшої підстановки конкретних значень на етапі виконання. Параметри в тексті запиту починаються з двокрапки.

Створимо нову форму, розмістивши на ній формі компоненти Query, DataSource, DBGrid, DBNavigator, кнопку Button1. Установимо їх властивості за аналогєю до попереднього прикладу.

Текст властивості SQL компонента зробимо таким:

```
SELECT Name, Capital, Continent, Area, Population
FROM "country.db" Country
WHERE (Name = :param1)
```

В останньому рядку ми визначили параметр запиту param1. Тепер відкривши в інспекторі об'єктів редактор властивості Params компонента Query1, ми побачимо елемент, що відповідає цьому параметру. Вибравши даний елемент відредагуємо його властивості.

DataType - тип значення параметра (за замовчуванням ftUnknown) установимо в ftString.

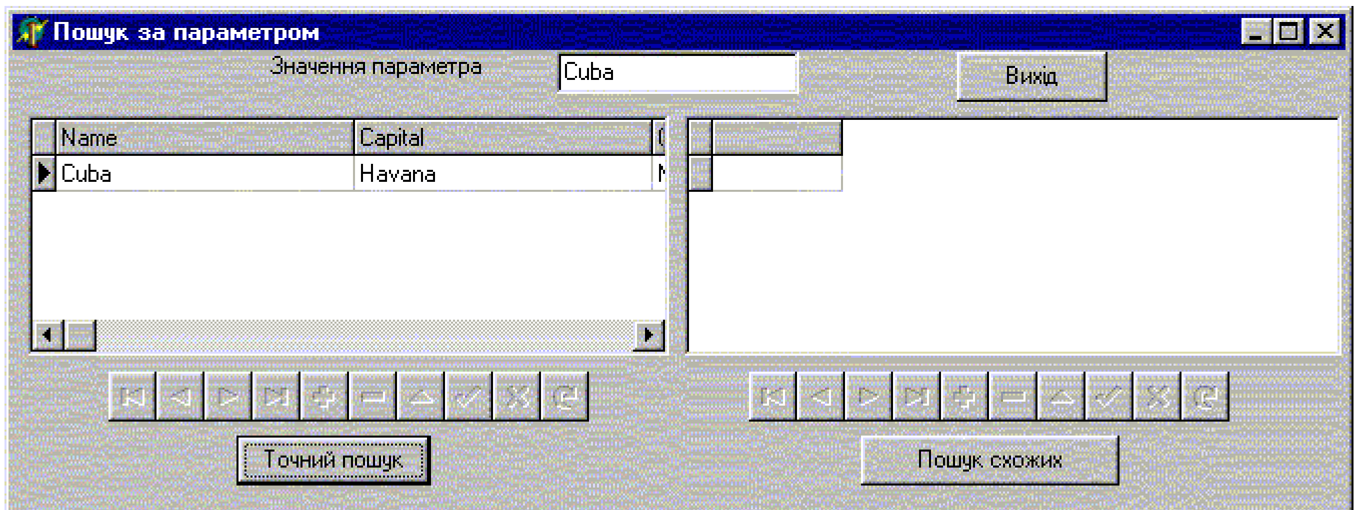
ParamType - тип параметра (за замовчуванням ptUnknown) установимо в ptInput.

Обробник події натискання на кнопку буде виглядати в такий спосіб:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Query1.Close;
  if not Query1.Prepared then
    Query1.Prepare;
  Query1.ParamByName('param1').Value:=Edit1.Text;
  Query1.Open;
end;
```

Для збільшення швидкості пошуку здійснюється перевірка підготовки запиту. Якщо запит не підготовлений (що трапляється при початковому запуску програми) викликається метод Prepare. При цьому сервер аналізує і компілює запит, зберігаючи його на сервері. Далі серверу необхідно відсилати лише значення параметрів запиту.

Виконаємо компіляцію та запустимо створений проект. У рядку пошуку введемо значення Cuba і натиснемо кнопку "Застосувати". Результатом пошуку буде вивід у рядків у DBGrid1, як це показано на рис. 5.



Для пошуку записів, що частково відповідають введеному критерію змінимо текст запиту компонента Query2 у такий спосіб:

SELECT Name, Capital, Continent, Area, Population
FROM "country.db" Country
WHERE (Name Like :param1)

Як видно, ми замінили строгу рівність (=) на LIKE. Це дозволить шукати текстові значення, що не строго відповідають заданому критерію. При цьому в критерії пошуку можна використовувати знаки підстановки. % - будь-які символи, _ - один символ. Наприклад, критерій пошуку C% у нашому випадку дасть нам вивід чотирьох записів (Canada, Chile, Colombia, Cuba).

В одному запиті можна комбінувати кілька умов пошуку за допомогою операторів OR, AND та ін. Звичайно, у прикладі ми освітили лише незначну частину можливостей SQL.

Фільтрація

На відміну від методів пошуку, що припускають добування даних зі сховища даних фільтрація припускає добір уже відібраних даних у клієнтському додатку. Для реалізації даного підходу в Delphi у компонентах доступу до даних уведені дві властивості Filter і Filtered. Установка властивості Filtered типу boolean у true переводить компонент у режим фільтрації. У властивості Filter при цьому можна визначити значення фільтру для відбору записів. Побудова фільтру багато в чому схоже на побудову умови where у SQL запиті. Основна відмінність у тім, що слово where не пишеться, використовуються інші знаки підстановки, у тексті фільтру не можна після знаків порівняння вставляти імена полів для локальних таблиць. Якщо ім'я поля містить пробіли, то його записують в квадратних дужках, наприклад [Home directory] Властивість FilterOptions дозволяє встановити додаткові параметри фільтрації, а саме

foCaseInsensitive - нечутливість до регістру в текстових полях;

foNoPartialCompare - відсутність пошуку по частковій умові, при установці даної опції знак * сприймається як літера, а не як знак підстановки будь-яких символів.

Приклад фільтрації можна знайти в прикладах, що поставляються з Delphi.

Реалізуємо фільтрацію даних за досить простих умов. У випадку, якщо необхідно реалізувати більш складний нестандартний фільтр можна написати обробник події OnFilterRecord. Тип події визначений як

type TFilterRecordEvent = procedure(DataSet: TDataSet; var Accept: Boolean) of object;

Іншими словами в обробнику події можна змінювати змінну Accept, указуючи чи буде відображатися кожен конкретний запис. Наприклад обробник може виглядати так:

Accept := DataSet['DateOfPayment'] > DataSet['DateOfPurchase'] + 30;

Зауважимо, що обробник події доповнює, а не заміщає властивість Filter, тож якщо включена фільтрація (Filtered := true) і ця властивість містить значення фільтра, то умова в обробнику події і фільтр пов'язані логічним відношенням "AND".

Порядок виконання роботи

1. Реалізувати пошук в БД по декількох полях створивши відповідний додаток на мові програмування високого рівня.