МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

Л. Ю. Гальчинський, В. О. Капустян

ТЕХНОЛОГІЇ ЕЛЕКТРОННОЇ ОБРОБКИ ДАНИХ В ІНФОРМАЦІЙНИХ СИСТЕМАХ ЕКОНОМІКИ

НАВЧАЛЬНИЙ ПОСІБНИК

Рекомендовано
Міністерством освіти і науки України
для студентів вищих навчальних закладів

Київ
«Центр учбової літератури»
2010

УДК 681.3(075.8)
ББК 22.18я73
Г 17

Гриф надано
Міністерством освіти і науки України
(Лист № 1/11-2081 від 18.03.2010 р.)

Рецензенти*:*
Черняк О. І. – доктор економічних наук, професор (Київський національний університет імені Тараса Шевченка, кафедра економічної кібернетики);
Кузнєцов Г. В. – доктор технічних наук, професор (Національний гірничий університет, Міжгалузевий інститут безперервної освіти);
Глибовець М. М. – доктор фізико-математичних наук, професор (Національний університет «Києво-Могилянська академія», кафедра інформатики).

Відповідальний редактор*:*
Новіков О. М. – доктор технічних наук, директор Фізико-технічного інституту Національного технічного університету України «КПІ».

Гальчинський Л. Ю., Капустян В. О.
Г 17 Технології електронної обробки даних в інформаційних системах економіки. Навч. посіб.– К.: Центр учбової літератури, 2010. – 504 с.

ISBN 978-611-01-0192-9

Метою даного посібника є висвітлення питань використання новітніх технологій обробки даних в інформаційних системах економіки.

Для студентів економічних спеціальностей вищих закладів освіти України, які вивчають сучасні інформаційні технології в рамках дисциплін "Інформаційні системи та технології в економіці", «Системи оброблення економічної інформації» «Економічна інформатика» та інших, так і викладачів, які забезпечують навчальний процес із даних дисциплін, та спеціалістів-практиків, які підвищують свою кваліфікацію в галузі сучасних інформаційних технологій.

Посібник, також, може стати в нагоді фахівцям, котрі займаються практичною розробкою та підтримкою баз даних у сучасних інформаційних системах економіки.

УДК 681.3(075.8)
ББК 22.18я73

ISBN 978-611-01-0192-9

НАВЧАЛЬНЕ ВИДАННЯ

Леонід Юрійович ГАЛЬЧИНСЬКИЙ
Володимир Омелянович КАПУСТЯН

ТЕХНОЛОГІЇ ЕЛЕКТРОННОЇ ОБРОБКИ ДАНИХ В ІНФОРМАЦІЙНИХ СИСТЕМАХ ЕКОНОМІКИ

НАВЧАЛЬНИЙ ПОСІБНИК

Оригінал-макет підготовлено
ТОВ «Центр учбової літератури»

Підписано до друку 14.12.2010. Формат 60x84 1/16
Друк офсетний. Папір офсетний. Гарнітура PetersburgCTT.
Умовн. друк. арк. 28,5. Тираж 300 прим.

Видавництво «Центр учбової літератури»
вул. Електриків, 23 м. Київ 04176
тел./факс 044-425-01-34
тел.: 044-425-20-63; 425-04-47; 451-65-95
800-501-68-00 (безкоштовно в межах України)
e-mail: office@uabook.com
сайт: www.cul.com.ua

Свідоцтво суб'єкта видавничої справи ДК № 2458 від 30.03.2006

Л.Ю.Гальчинський , В.О.Капустян

Технології електронної обробки даних в інформаційних системах економіки
Анотація
(навчальний посібник)
Анотація

Метою даного посібника є висвітлення питань використання новітніх технологій обробки даних в інформаційних системах економіки. Характер розробки модерних підходів обробки даних за останні 10-15 років носили лавиноподібний характер. Відтак зміст та можливості цих новітніх технологій на даний момент ще не до кінця усвідомлені спільнотою, яка займається розробкою, впровадженням та експлуатацією інформаційних систем, де вони мають застосовуватись. Не так легко навіть професіоналу розібратися в заплутаному лабіринті сучасних комп'ютерних технологій, не кажучи вже про новачків. Разом із тим зрозуміти механізми сучасних складних процесів обробки даних можна тільки на основі систематичного вивчення базових принципів, які склалися по мірі вирішення нових задач, що виникали в процесів розвитку економіки. Тому в даному посібнику витримується концепція історичного розвитку технології електронної обробки даних – від табуляторів Голлерита до розподілених інформаційних систем. І хоча нема дефіциту комп'ютерної літератури, існує необхідність у системному розгляді питань суті нових технологій обробки даних в інформаційних системах, в першу чергу для економічних суб'єктів. Найбільшою мірою це стосується як студентів економічних спеціальностей вищих закладів освіти України, які вивчають сучасні інформаційні технології в рамках дисциплін "Інформаційні системи та технології в економіці", «Системи оброблення економічної інформації» «Економічна інформатика» та інших, так і викладачів, які забезпечують навчальний процес із даних дисциплін, та спеціалістів-практиків, які підвищують свою кваліфікацію в галузі сучасних інформаційних технологій. Посібник, також, може стати в нагоді фахівцям, котрі займаються практичною розробкою та підтримкою баз даних у сучасних інформаційних системах економіки.

В посібнику розглянуті сучасні технології, які дозволяють реалізувати найскладніші завдання обробки даних, які виникають у діяльності корпорацій, фінансових та кредитних установах, та інших бізнесових структурах, а також державних органах управління народним господарством.

ЗМІСТ 6

Вступ

Історія інформаційних технологій в економіці нараховує більше п’яти десятиріч. Інформаційна комп’ютерна технологія, зародившись у нетрях військово-промислових комплексах країн, що воювали в II світовій війні, з середини 50-х років 20-го сторіччя почала стукатись у двері величезного світу застосувань у сфері економіки. Найбільш далекоглядним ученим, інженерам, керівникам та економістам була очевидна перспектива використання комп’ютерів не тільки в таких сферах, як закриті військові проекти, пов'язані з розробкою новітніх засобів масового знищення людей, або в чисто наукових розрахунках академічної науки, але й у таких «приземлених» галузях, як управління підприємствами, планування народногосподарськими комплексами, банківської сфери й тому подібне.

Але на шляху до широкого впровадження інформаційних технологій на базі комп’ютерів стояла немала низка нелегких проблем, без розв’язання яких впровадження цих технологій було б черговою утопією. Однією із ключових проблем постала розробка електронної технології обробки даних ЕОД. Незважаючи на всю прозаїчність своєї назви, розвиток цієї технології мало епохальне цивілізаційне значення - аналог тому, що людина спочатку пересовувалась пішки, потім почала використовувати для пересування тварин, потім почала їздити на поїздах та автомобілях, зрештою, полетіла на літаку та ракеті.

Можна провести певні аналогії далі. Так ось, на початку впровадження інформаційних систем у практику діяльності народногосподарчих комплексів у першій половині 50-х років XX століття стан управління обробкою даних приблизно відповідав стану транспорту, де ходять нечисленні паровози по досить обмежених шляхах. У ролі таких паровозів вже доволі довгий час виступали розрахунково-перфораційні комплекси (РПК), основою яких був перфокартковий табулятор Голлерита. Наступні роки свідчать про бурхливий, вибухоподібний прогрес у цій галузі, і власне саме на останні десятиліття припадають всі основні етапи розвитку технології обробки даних для економічних застосувань. Далі

розглянемо етапи розвитку технології обробки даних, базуючись на класифікації Джеймса Грея, лауреата премії Алана Тьюринга.

Перші кроки систем управління даними пов'язані з рішенням традиційних задач автоматизації - обліку транзакцій у бізнесі, науці й виробництві. Ці дані складалися, головним чином, із чисел і символьних рядків. Сьогодні ці системи забезпечують інфраструктуру для більшої частини суспільства, надаючи можливість швидкого, надійного, безпечного й автоматичного доступу до даних, розподілених по всьому світу. Наступні кроки полягають в автоматизації доступу до більш різноманітних і складних форм даних: графічних, звукових і відеообразів, карт та інших середовищ. Іншим важливим напрямком є використання даних просторово віддалених від кінечного користувача. Тепер комп'ютери можуть зберігати всі форми інформації: записи, документи, образи, звуко- і відеозаписи, наукові дані, і багато нових форматів даних. Людство добилися великих успіхів у технології отримання, зберігання, управління, аналізу й візуалізації даних. Сукупно ці задачі називаються управлінням даними. У даному посібнику дається огляд еволюції систем управління даними із приведенням опису шести поколінь менеджменту даними.

Системи управління даними зазвичай зберігають величезні об'єми даних, що представляють історичні записи організації. Розміри цих баз даних бурхливо ростуть. Важливо те, що старі дані й програми продовжують працювати при додаванні нових даних і прикладних програм. Системи постійно змінюються. Дійсно, велика частка крупних систем баз даних була розроблена декілька десятків років тому і змінювалася разом із розвитком технології. Погляд в історію допомагає зрозуміти сучасні системи.

В управлінні даними було шість різних фаз. Спочатку дані оброблялися вручну. На наступному кроці використовувалися устаткування з перфокартами й електромеханічні машини для сортування й табуляції мільйонів записів. На третій фазі дані зберігалися на магнітних стрічках, і програми, що зберігаються, виконували пакетну обробку послідовних файлів. Четверта фаза ввела поняття схеми бази даних і оперативного навігаційного доступу до даних. На п'ятій фазі був забезпечений автоматичний доступ до реляційних баз даних і була впроваджена розподілена та клієнт-серверна обробка. Тепер ми знаходимося на етапі бурхливого розвитку шостого покоління систем, які можуть зберігати, і обробляти широкий спектр типів даних, зокрема, документи, графіку, звук і відеообрази. Ці системи шостого покоління є базовими засобами зберігання для прикладних програм Internet та Intranet, що з'являються.

Нульове покоління: ручне управління записами (4000 р. до н.е. - 1900).

З перших відомих письмових свідченнях, які збереглися, описується облік царської казни й податків в Шумері. Підтримка записів має довгу історію. У наступні шість тисяч,

якщо відраховувати від шумерів, років спостерігалася еволюція від глиняних табличок до папірусу, потім до пергаменту й, нарешті, до паперу. Серед артефактів писемності, таких як літературні та релігійні твори або літописні свідчення, дуже великий відсоток займають дані економічної діяльності: боргові розписки, торгові угоди, складські облікові документи та інше. Це були великі досягнення, але обробка інформації в цю епоху проводилася вручну.

Перше покоління: механізоване управління записами (1900-1955).

Прототип автоматизованої обробки інформації з'явився приблизно в 1800 році, коли Джеквард Лум почав проводити розкрій тканини за зразками, представлених перфокартами. Пізніше аналогічна технологія використовувалася в механічних піаніно. В 1890 р. Голлерит використовував технологію перфокарт для виконання перепису населення Сполучених Штатів. До 1955 року у багатьох компаній були цілі поверхи, призначені для зберігання перфокарт, що дуже нагадувало архіви глиняних таблиць шумерів. На інших поверхах розміщувалися шеренги перфораторів, сортувальників і табуляторів. Ці машини програмувалися шляхом перемонтування управляючих панелей, які управляли певними регістрами-накопичувачами і вибірково відтворювали карти на інших картах або на папері. Великі компанії обробляли і проводили мільйони записів щоночі. Цю роботу було б неможливо виконати ручними методами обробки. Проте було ясно, що настає час, коли нова технологія витіснить перфокарти і електромеханічні комп'ютери.

Друге покоління: програмоване устаткування обробки записів (1955-1970).

Електронні комп'ютери з програмами, що зберігаються були розроблені в 1940-х і початку 1950-х років для виконання наукових і чисельних обчислень. Електронні комп'ютери з програмами, що зберігаються були розроблені в 1940-х і початку 1950-х років для виконання наукових і чисельних обчислень. Ці нові комп'ютери могли обробляти сотні записів в секунду і їх можна було розмістити на набагато меншій площі, ніж попереднє устаткування. Ключовим компонентом цієї нової технології було програмне забезпечення. Стало набагато легше програмувати і використовувати комп'ютери. З’явилися досконалі на той час програмні засоби, як COBOL і RPG, що дозволило писати прикладні програми для сортування, аналізу та обробки даних. Почали з'являтися стандартні пакети для таких загальновживаних бізнес-застосувань як загальна бухгалтерія, розрахунок заробітної платні, ведення інвентарних відомостей, управління підпискою, банківська діяльність і ведення бібліотек документів.

Реакція на появу цих нових технологій була легкопередбачуваною. Великі корпорації накопичували все більше інформації і це вимагало все більше засобів обробки даних. В міру зниження цін не тільки крупні компанії, але і середні стали зберігати дані на картках і використовувати комп'ютер для обробки карток. Великим кроком вперед був винахід та

застосування магнітної стрічки для збереження даних. Програмне забезпечення цього часу підтримувало модель обробки записів на основі файлів. Типові програми послідовно читали декілька вхідних файлів і генерували на виході нові файли. Для полегшення розв'язку цих орієнтованих на записи послідовних задач були створені COBOL і деякі інші мови програмування. Операційні системи забезпечували абстракцію файлу для зберігання цих записів, мову управління завданнями для виконання завдань і планувальник завдань для управління потоком робіт.

Системи пакетної обробки зберігали дані з програмами на картах або стрічках і збирали їх в пакети для подальшої обробки. Раз на день ці пакети сортувалися. Відсортовані транзакції записувалися на стрічку, де приєднувалися до набагато більшої за розмірами базою даних (основним файлом) для утворення нового основного файлу. На основі цього основного файлу також робився звіт про роботу за день. Пакетна обробка дозволяла дуже ефективно використовувати комп'ютери, але мала два серйозні обмеження. Якщо в транзакції була помилка, вона не розпізнавалася до обробки основного файлу, і можна було витратити декілька днів для виправлення помилки. Ще більш суттєвим було те, що керівництво та спеціалісти не знали поточного стану бази даних - оскільки транзакції реально не оброблялися до наступного ранку. Для вирішення цих двох проблем був потрібен наступний крок еволюції - операційні системи. Цей крок також істотно полегшив написання програм.

Третє покоління: оперативні мережеві бази даних (1965-1980).

Для таких застосувань, як ведення операцій на фондовій біржі або резервування квитків, потрібне знання поточної інформації. Ці програми не можуть використовувати вчорашню інформацію, яку забезпечують системи пакетної обробки транзакцій, - їм потрібен негайний доступ до поточних даних. З кінця 1950-х лідируючі компанії з декількох областей індустрії почали вводити у використання системи баз даних з оперативними транзакціями; транзакції над оперативними базами даних оброблялися в інтерактивному режимі. Оперативний доступ до даних забезпечувався декількома ключовими технологіями. Апаратура для підключення до комп'ютера інтерактивних комп'ютерних терміналів пройшла шлях розвитку від телетайпів до алфавітно-цифрових дисплеїв і, нарешті, до сьогоднішніх інтелектуальних терміналів, заснованих на технології персональних комп'ютерів. Монітори телеобробки були спеціалізованим програмним забезпеченням для мультиплексування тисяч терміналів зі скромними серверними комп'ютерами того часу. Ці монітори збирали повідомлення-запити, що надходять з терміналів, швидко призначали програми серверу для обробки кожного повідомлення і потім направляли відповідь на відповідний термінал. Оперативна обробка транзакцій доповнювала можливості пакетної обробки транзакцій, за якою залишалися задачі фонового формування звітів.

Оперативні бази даних зберігалися на магнітних дисках або барабанах, які забезпечували доступ до будь-якого елемента даних за частки секунди. Ці пристрої і програмне забезпечення управління даними давали можливість програмам прочитувати декілька записів, змінювати їх і потім повертати нові значення користувачу прикладноъ програми. Спочатку системи забезпечували простий пошук даних: або прямий пошук по номеру запису, або асоціативний пошук по ключу.

Проста індексно-послідовна організація записів швидко розвинулися до більш потужної моделі записів, орієнтованої на набори. Прикладним програмам часто необхідно зв'язати два або більше записів. На той момент ця проблема була розв’язана за рахунок розробки моделей даних, зокрема ієрархічних та мережевих.

До 1980 року мережні (і ієрархічні) моделі даних, орієнтовані на набори записів, стали дуже популярні. Заснована Ч.Бахманом компанія Cullinet в ті часи була найбільш успішною софтверною компанією в світі.

Четверте покоління: реляційні бази даних і архітектура клієнт-сервер (1980-1995).

Незважаючи на успіх мережної моделі даних, багато розробників програмного забезпечення відчували, що навігаційний програмний інтерфейс був інтерфейсом дуже низького рівня. Було важко проектувати і програмувати ці бази даних. В статті Е.Ф. Кодда 1970 року була описана реляційна модель [], яка пропонувала альтернативу низькорівневому навігаційному інтерфейсу. Ідея реляційної моделі полягає в тому, щоб одноманітно представляти і сутності, і зв'язки. Реляційна модель даних мала уніфіковану мову для визначення даних, навігації за даними і маніпулювання даними, а не окремими мовами для кожної з цих задач. Ще більш важливо те, що реляційна алгебра має справу з множиною записів (відношеннями) як єдиним цілим, застосовуючи операції до множини записів в цілому і як результат теж отримується у вигляді множини записів. Реляційні модель даних і операції дають можливість отримання більш коротких і більш простих програм для вирішення задач управління записами. Замість неявного зберігання зв'язку між сутностями в реляційній системі явно зберігається кожна пара відношень як запис бази даних.

Отримавши такий непроцедурний запит, реляційна система баз даних автоматично знаходить кращий спосіб підбору записів із відповідних таблиць. Запит не залежить від того, які визначені зв'язки. Він продовжуватиме працювати навіть після логічної реорганізації бази даних. Отже, запит суттєво більш незалежний від даних, ніж навігаційний запит, заснований на мережній моделі даних. Крім більшої незалежності даних реляційні програми часто в п'ять - десять разів простіші відповідних навігаційних програм.

Реляційні системи, з'єднані з GUI, наразі дозволяють сотням тисяч людей щодня формулювати складні запити до баз даних. Комбінація GUI і реляційних систем підходить ближче всього в досягненню мети автоматичного програмування. GUI дозволяють просто конструювати складні запити. Для заданого непроцедурного запиту реляційні системи самі знаходять найефективніші способи його виконання.

П'яте покоління: мультимедійні бази даних (1995-...).

Реляційні системи внесли багато удосконалень в полегшення використання, графічні інтерфейси, клієнт-серверні прикладні програми, розподілені бази даних, паралельний пошук даних і добування даних. Проте біля 1985 року дослідницьке співтовариство почало розглядати питання, що виходять за рамки реляційної моделі. Традиційно існувало чітке розділення програм і даних. Цей підхід добре працював, поки йшлося тільки про такі дані як числа, символи, масиви, списки або множини записів. В міру появи нових прикладних програм розділення програм і даних стало проблематичним. Від прикладних програмам вимагалося дати правила поведінки з даними . Наприклад, якщо дані представляли собою складний об'єкт, то методи пошуку, порівняння і маніпулювання даними ставали специфічними для типів даних «документ», «графічний образ», «звук» або «карта» Ця поведінка може бути інкапсульована в бібліотеках класів, які підтримують нові типи. В цій моделі система баз даних зберігає, вибирає дані і забезпечує зв'язки між елементами даних, а бібліотеки класів забезпечують поведінку цих елементів.

Традиційний підхід полягав у вбудовуванні типів даних прямо в систему баз даних. В мові SQL були додані нові типи даних для часу, часових інтервалів і символьних рядків з двобайтовим представленням символів. Кожне з цих розширень було значним досягненням. Коли вони були зроблені, результати задовольнили не всіх. Наприклад, тип даних «час» не дозволяє представляти дати до Різдва Христова, а в символьних рядках не можна застосовувати кодування Unicode (універсальний набір символів майже для всіх мов). Користувачі, які хотіли використовувати Unicode або дати до нашої ери, повинні визначати свої власні типи даних. Ці прості приклади, а також багато інших, переконали співтовариство баз даних в тому, що системи баз даних повинні дозволяти прикладним фахівцям реалізовувати типи даних для своїх прикладних областей. Географам слід мати можливість реалізації карт, фахівцям в області текстів має сенс реалізовувати індексацію і вибірку текстів, фахівцям по графічних образах було варто б реалізувати бібліотеки типів для роботи з образами. Конкретним прикладом може служити поширений об'єктний тип часових рядів. Замість вбудовування цього об'єкту в систему баз даних рекомендується реалізація відповідного типу у вигляді бібліотеки класів з методами для створення, оновлення і видалення часових рядів.

Спеціалісти з об'єктно-орієнтованого програмування чітко розпізнали проблему: для розробки типів даних потрібна добра модель даних і уніфікація процедур і даних. Дійсно, програми інкапсулюють дані і забезпечують всі методи для маніпулювання даними. Дослідники, розробники реляційних баз даних з 1985 року почали роботи по модифікації реляційної моделі в такий спосіб, аби об'єднати об'єктно-орієнтовані і реляційні системи. В кінці 1980-х на ринку з'явилося більше десяти продуктів об'єктно-орієнтованих баз даних, але замовники не поспішали прийняти ці системи. Тим часом традиційні постачальники прагнули розширити мову SQL, щоб включити в нього концепції об'єктно-орієнтованого підходу, зберігаючи переваги реляційної моделі. Суперечки відносяться до ролі SQL, деталей об'єктної моделі і базових бібліотек класів, які слід підтримувати в системах баз даних.

Швидкий розвиток Internet підсилює ці дебати. Клієнти і сервери Internet будуються з використанням апплетів і «хелперів», які зберігають, обробляють і відображають дані того або іншого типу. Користувачі вставляють ці апплети в браузер або сервер. Загальнопоширені апплети керують звуком, графікою, текстами, відео, електронними таблицями. Для кожного з асоційованих з цими апплетами типів даних є бібліотека класів. Настільні комп'ютери і Web-браузери є поширеними джерелами і приймачами більшої частини даних. Тому типи і об'єктні моделі, що використовуються в настільних комп'ютерах, диктуватимуть, які бібліотеки класів повинні підтримуватися на серверах баз даних.

Підводячи підсумок, зауважимо, що бази даних покликані зберігати більше, ніж тільки числа і текстові рядки. Вони використовуються для зберігання багатьох видів об'єктів, які ми бачимо в World Wide Web, і зв'язків між цими об'єктами. Відмінність між базою даних і рештою частини Web стає неясною. Кожний постачальник баз даних обіцяє «універсальний сервер», який зберігатиме і аналізуватиме всі форми даних (всі бібліотеки класів і відповідні об'єкти).

Розділ 1. Соціально-економічні причини виникнення та розвитку технологій електронної обробки даних

Век девятнадцатый, железный,
Воистину жестокий век!
Тобою в мрак ночной, беззвездный
Беспечный брошен человек!..

А.Блок

1.1 Передісторія електронної обробки даних

Другу половину девятнадцятого сторіччя багато хто з істориків та економістів вважають не тільки часом бурхливого розвитку індустріальної революції, але і початком інформаційної

революції, прискорені темпи якої ми переживаємо зараз. Справді, винайдення телеграфу, телефону та радіо революційним чином змінило засоби комунікації між людьми. До цього ряду інновацій належить і розрахунково-перфораційні комплекси Голлерита. Нас цей факт особливо цікавить, бо саме вони стала вперше інструментом машинної обробки економічної інфомації. Обчислювальна машина замислювалася Г.Голлеритом як Census Machine (машина для перепису). На перший погляд нам зараз це здається дивним. Але, якщо розібратися, то Г.Голлерит не випадково гордо називав себе першим «статистичним інженером». Поява і початок виробництва розрахункових машин Г.Голлерита в 80-х рр. демонструє адекватну відповідь суспільства на виклик нових соціально-економічних задач, пов'язаних з обробкою великих обсягів інформації (перш за все в сферах обліку і статистики).

Одним з наслідків Великої промислової революції виникла необхідність в точному обліку населення. Ось чому з другої половини XIX в. перепис населення як одна з найважливіших державних задач проводилася регулярно - через 10 років, і цю вимогу статистики строго дотримували всі розвинуті індустріальні держави. В колах, причетних до статистики гуляла байка про те, чому це так. Суть байки в тому, що на іспиті зі статистики одного студента запитали, чому переписи населення називають десятирічками. «Тому, - відповів він, - що їх результати обробляються протягом 10 років». І в цьому була велика доля правди, оскільки обробка отриманих даних дійсно була страшенно трудомістка, проводилася протягом декількох років, як правило, вручну. Дуже яскраво ці перипетії описував класик світової літератури А.П.Чехов, який брав безпосередню участь у Всеросійському перепису 1897 року. Ситуацію ускладнювало те, що статистиків вже не задовольняли дані тільки про кількість населення. Були необхідні відомості про національність, рідну мову, вік, стать, віросповідання і т.д. Для цього необхідно було класифікувати зібраний матеріал і виконувати розрахунки за різними ознаками. При цьому обсяг роботи настільки збільшувався, що виконати його оперативно і якісно навіть при використанні механічних арифмометрів або підсумовуючих машинках, виявилося неможливим. Це був один з перших серйозних проявів явища, яке пізніше було названо кризою контролю[]. Суть цієї гіпотези, сформульованої Джеймсом Бенігером, полягає в тому, що інформаційні технології, і з ними саме інформаційне суспільство(information society) почали формуватися з появою машинної індустрії та розвинутої мережі транспортних засобів у вигляді залізниць, з середини XIX ст. Як він писав: «повозкам і караванам не потрібні розклади та регулювальники». Натомість машинна промисловість породила величезні, швидкі потоки товарів, і це не може контролюватися і керуватися без належного рівня інформаційної технології (яку Бенігер розглядає дещо розширено, включаючи такі речі, як стандартизацію продукту, бюрократію і рекламування, а також і звичайні механічні пристрої): і якщо цим всім не керувати, це просто не може працювати. Таке індустріальне виробництво - вагома причина поліпшити

інформаційну технологію, а відтак, поліпшити технологію контролю. Вивчаючи економічний розвиток , можна побачити, що кожна якісна зміна способів технології та організації виробництва, або суттєва зміна зовнішнього ринкового оточення породжує кризу контролю(управління) і адекватна відповідь може бути знайдена тільки у революції управління(Control revolution).

Гіпотеза про кризу контролю як джерело розвитку інформаційних технологій нині поділяється спільнотою соціологів, економістів і філософів, лежить в процесах, що почалися ще в XIX ст. і триває до сьогоднішнього дня. Зокрема, на той час ця загальна проблема нагально проявила себе у такій важливій сфері, як перепис населення. Вона і була подолана за рахунок впровадження інновації - створення технології обробки даних за допомогою нового спеціального класу пристроїв, що пізніше отримали назву рахунково-аналітичних машин, а з початку 1960-х рр. – перфораційних, таких собі широкозахватних комбайнів, які могли б перелопатити величезні масиви даних.

Рис. 1.1 Герман Голлерит

Вперше проблемою механізованої обробки статистичної інформації зайнявся талановитий американський винахідник Герман Голлерит. Його трудова діяльність почалася в Бюро цензів США. Це статистичне управління при міністерстві внутрішніх справ займалося проведенням переписів населення і обробкою результатів. Тут в 1880 р. Голлерит за рекомендацією керівника управління Дж. Біллінгса почав займатися дослідженнями в області механізованої обробки статистичних даних з використанням перфорованих карток.

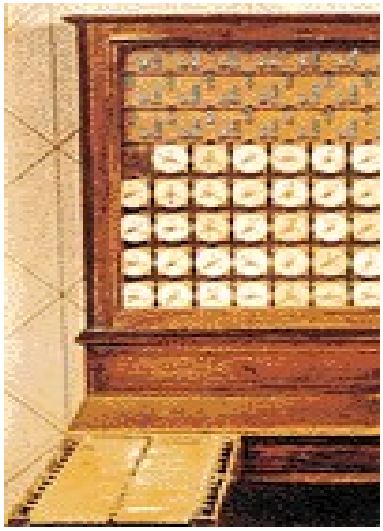


Рис. 1.2 Розрахунково-перфораційний комплекс Г. Голлерита.
Державний Політехнічний музей м.Москва

Задум Г. Голлерита полягав в тому, щоб на кожну людину завести особисту картку і всі дані, що підлягали обробці представити отворами у фіксованих місцях (позиціях). Ця перфокарта на вигляд не була схожа на залізничний квиток або вже відомі карти, і була оригінальною авторською розробкою. Вона була зроблена з щільного картону розміром приблизно з доларовий папірець, але розмір картки міг коливатися залежно від кількості позицій, кожна з яких відповідала за певну ознаку (стать, сімейний стан, віросповідання і т. д.), наприклад в австрійському переписі 1890 р. застосовувалися перфокарти, що мають, 20Х12 позицій; у російському переписі 1897 р. – 22Х12 позицій.

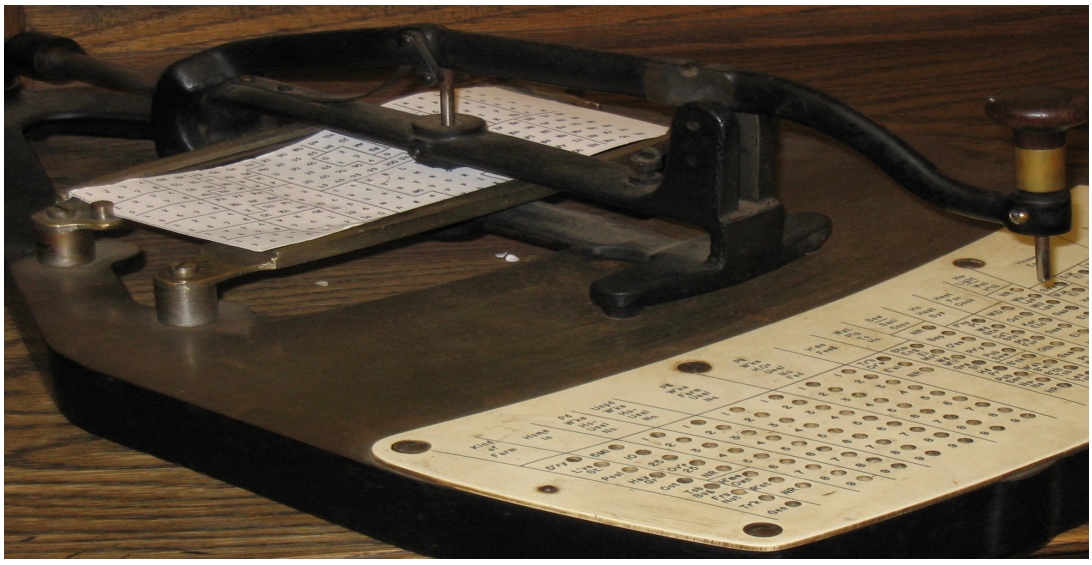


Рис. 1.3 Перфоратор за роботою

Відомості заносилися на перфокарту вручну за допомогою пробивного пристрою - пантографа або перфоратора. На лицевій панелі перфоратора є таблиця ознак у вигляді карти-шаблона з отворами по всій координатній сітці, над якою по радіусу переміщається важіль з штифтом на кінці. Якщо в спеціальну раму для картки покласти чисту перфокарту і опустити штифт в отвір, відповідний якій-небудь ознаці, то спеціальний пристрій в рамі в тій

же позиції перфокарти проб'є ідентичну ознаку. За годину на перфораторі можна заповнити біля 80 карток. Тепер можна було або підрахувати отвори на всіх перфокартах на основній машині - табуляторі, або розподілити їх за тим же принципом на сортуванні. «Елементною базою» для табулятора стали електромагнітні реле. У необхідній кількості (біля 120 штук) реле встановлювали на задній панелі табулятора. Це було революційною інновацією Г.Голлерита, бо з моменту винаходу реле в 1831 р. йому першому прийшла в голову думка застосувати його для обчислювальної техніки. В сортувальній машині вони розташовувалися в кожному із 26 осередків для відсортованих перфокарт і з'єднувалися електропроводами з лічильниками табулятора. Швидкість обробки карток на табуляторі складала 1000 штук в годину.

Рис 1.4 Макет карти перепису 1890 року в США на перфокарті Голлерита

Кожна перфокарта була призначена для зберігання даних про одну людину. Можна виділити наступні зони розміщення інформації:
1. Перші чотири колонки(1), починаючи зліва, містили цифровий код району проведення перепису.
2. Раса: Jp, Ch, In, W, B і т.д.
3. Стать: M, F (рядки 3..4)
4. Вік визначався попаданням в п'ятирічний період: 0,5,10..100
5. Вік у середині п'ятирічного періоду: 0..4 (рядки 3..5)
6. Сімейний стан
7. Закодоване місце народження згідно спеціальної таблиці.

Рис. 1.5 Доларовий папірець (розмір в 1890 року)

Табулятор приймав картки розміром як доларовий папірець. На картках було 240 позицій (12 рядів по 20 позицій). При прочитуванні інформації з перфокарт 240 голок пронизували ці карти. Там, де голка потрапляла в отвір, вона замикала електричний контакт, внаслідок чого збільшувалося на одиницю значення у відповідному лічильнику. Розроблена Голлеритом 80-колонна перфокарта не зазнала істотних змін і як носій інформації використовувалася в перших трьох поколіннях комп'ютерів. Отже, управління механічними лічильниками і сортуванням здійснювалося електричними імпульсами, що виникають при замиканні електричного ланцюга за наявності отвору в перфокарті. Імпульси використовувалися і для введення чисел, і для управління роботою машини. Тому машина Г.Голлерита була визнана першою електромеханічною обчислювальною машиною з програмним управлінням. Вона була комплектом пристроїв (перфоратора, сортувальної машини і табулятора) різного функціонального призначення, але зв'язаних між собою технологічно процесом обробки інформації.

Технологія Г.Голлерита цілком виправдала себе під час перепису населення США у 1890 році. Немалою мірою цьому сприяла і чітка організація роботи по збору та наступної обробки даних. Гучний успіх застосування цих комплексів при перепису населення в США 1890 році сприяв значному поширенню цієї технології як в США, так і в інших країнах(Канада, Австро-Угорщина, Норвегія, Італія, Російська імперія), щоправда не скрізь успішно, зокрема в Російській імперії. Проте окремі невдачі були обумовлені, не стільки недоліками технології Голлерита, скільки недоліками її впровадження. Виник досить стійкий попит як на самі РПК, так і їх застосування, причому не тільки у сфері статистики, але і для розв’язку інших задач обробки даних, переважно економічного змісту. В Україні перші застосування РПК відносяться до середини 1920-х років для ЦСУ УРСР та Наркомату шляхів сполучень. Використання РПК було організовано у формі машинно-лічильних станцій, що проіснували до 70-х років двадцятого століття, які поступово були замінені на обчислювальні центри, оснащені ЕОМ. Природньо, що попит породив компанії, які спеціалізувались у виробництві та впровадженні РПК. Одну з них організував сам Голлерит ще в 90-х роках XIX століття, потім вже в 1911 вона стала частиною Computer-Tabulating-Recording Co., яка з 1924 отримала ім’я IBM(International Business Machines) -тепер всесвітньо відомий бренд.

Рис. 1.6 Логотип IBM

Єдиним серйозним конкурентом IBM на цьому секторі ринку була фірма Remington Rand.

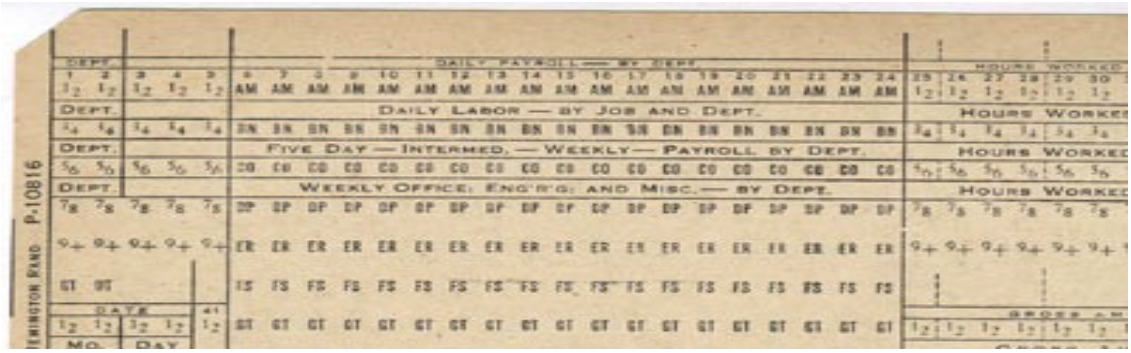


Рис. 1.7 Розрахункова обчислювальна картка зарплати співробітників Grunman Aircraft Engineering Corp за стандарт-перфокартою Remington Rand

Подібно IBM, Remington Rand спочатку виробляла офісну техніку: друкарські машинки, арифмометри та інші засоби оргтехніки. В більш давні часи фірма також намагалася конкурувати з IBM на її традиційному ринку табуляторів і перфокарт. Прикладом цієї боротьби було введення власної стандарт-перфокарти з 90 колонками, замість перфокарти з 80 колонками як у IBM, що давало збільшення місткості перфокарти. Але цей інноваційний маневр Remington Rand провела невдало, табулятори компанії не були сумісні з устаткуванням IBM і мали гірші технічні характеристики, що зрештою, в результаті привело до зменшення ринкової частки компанії, а отже, і її доходів.

Треба було чекати якоїсь нової ситуації для чергової конкурентної атаки. І ось виникнення нового ринку ЕОМ давало компанії новий шанс обігнати свого давнього конкурента. Ось чому в кінці 1940-х рр. чільник компанії Д. Ренд поставив мету одночасного створення комп'ютерів трьох типів: ЕОМ для урядових організацій, наукових досліджень і для бізнесу, які замінили б морально застарілі табулятори.

Поява перших обчислювальних машин в США, наприклад ENIAC, спочатку була сприйнята як інновація, виключно пов’язана з потребами військово-промислового комплексу(ВПК) та академічної науки(як правило, те і інше сполучалось, бо часи були військові). Подібна ситуація була і у Великобританії. Colossus I – перша обчислювальна машина на електронних лампах, створена англійцями при безпосередній участі самого Алана Тьюринга в 1943 р., для розкодування німецьких військових шифрів. У той час секретні повідомлення німецьких військових були зашифровані кодом ENIGMA, який добре знали англійські шифрувальники, але для цілком таємних повідомлень німці використовували інший шифр, який англійці змогли зламати тільки за допомогою Colossus. Цей «колосс» складався з 1800 електронних ламп і був одним з перших програмованих електронних цифрових комп'ютерів. Але оскільки він не призначався для широкопланових обчислень, а мав свою конкретну задачу, його не вважали повноцінним комп'ютером. І довгий час лаври першого електронного цифрового комп'ютера належали ENIAC, який був запущений в 1946 році.

Нещодавно британські спецслужби розсекретили документи, з яких виходить, що першим електронним обчислювальним пристроєм в світі, що працював за програмою, тобто “справжнім комп'ютером», був Colossus II, досить істотно модифікована версія комп'ютера Colossus I. Colossus II почав роботу в червні 1944 року. Основним його призначенням була дешифровка перехоплених повідомлень ворога, проте він міг бути перепрограмований під різноманітні задачі. Всього було випущено 10 комп'ютерів Colossus II, але всі вони відразу після війни були знищені. На думку британських спецслужб, Colossus II був такою передовою розробкою, що його краще було ліквідувати, аби тільки б він не потрапив не у ті руки. Можливо, що у британських спецслужб були підстави так думати, проте це вже тема іншого дослідження.

І ось після закінчення війни з’явилася думка про комерційне застосування комп’ютерів. Зараз нам здається, що ця ідея мала бути очевидною, проте, коли в березні 1946 р. Джон Преспер Еккерт і Джон Уїльям Моучлі - творці ENIAC'a і автори ідеї ЕОМ з програмою, що зберігається, - пішли з Пенсільванського університету, щоб організувати власну компанію і зайнятися розробкою і виробництвом комерційних ЕОМ. Ідея про те, що ЕОМ буде призначена для вільного продажу будь-якому споживачеві для цивільного, тобто економічного застосування сприймалась в кінці 40-х років як зухвалий виклик. Громадська думка, тобто переважна більшість спеціалістів по ЕОМ, керівництва фірм, що потенційно могли реалізувати проект та урядовців не була готова сприйняти цю ідею. Найголовніше ж було те, що топ-менеджери провідних фірм США не розуміли, що це дасть їхньому бізнесу. Тому це був дуже ризиковий проект, і його ініціатори керувалися не стільки економічними розрахунками, скільки своєю візінерською місією.

Рис.1.8 Преспер Еккерт та Джон Моучлі

Вони хотіли створити по-справжньому універсальну ЕОМ, яка на відміну від ENIAC'a і EDVAC'a вирішувала б не тільки наукові і інженерні задачі, але і - головним чином! - виконувала ділові застосування (завдання обліку, планування, статистики, сортування даних,

логістики та інше). Але з іншого боку апріорі було неясно, скільки таких машин буде потрібно, а отже, і чи окупляться витрати розробників і виробників (а тим більше - чи можна буде отримати прибуток від продажів), і ця невизначеність була, мабуть, головною проблемою. За сучасною термінологією це був інноваційний проект з дуже великим ризиком. Тому не дивно, що на перших порах реалізація цього проекту практично ніким не була підтримана.

Рис. 1.9. Джон фон Нейман (1903-1957)

Навіть авторитетні фахівці (Д. фон Нейман, Г. Ейкен і ін.), що входили в Комітет з швидкодіючих обчислювальних машин при Національній раді з досліджень (National Research Council), досить скептично віднеслися до планів Еккерта та Моучлі стосовно створення ділових ЕОМ: вони вважали, що обчислювальні машини будуть потрібні тільки для науково-технічних розрахунків, а для обробки ділових даних як і раніше служитимуть розрахунково-перфораційні комплекси (РПК). Так, Говард Ейкен, який вважав, як правило, гідним уваги лише власні розробки, заявив на засіданні Комітету:

“Ми вводимо в оману не тільки державні установи, які виділяють кошти на розробку машин подібного роду, але і громадську думку, створюючи враження, що ці плани можуть дістати схвалення, хоча ніколи не буде об'єму завдань, достатнього для завантаження роботою більш ніж однієї або двох подібних машин. Необхідно зупинити дурниці, які мають намір скоїти Еккерт і Моучлі“. Це був досить промовистий приклад помилкового прогнозу у сфері прогресу інформаційних технологій, хоча він ґрунтувався на достовірних фактах, оскільки якихось серйозних претензій у користувачів до РПК на той час не було. Але, очевидно більш правильний прогноз має ґрунтуватися на передбаченні майбутнього, а не на реаліях сьогодення. Одначе зазначимо, що через декілька років Ейкен чесно визнав свою помилку відносно ділових ЕОМ. Стосовно висловлювань Дж. фон Неймана на цю тему, свідчень на жаль, не збереглося, крім того, що він в свій час солідаризувався з Ейкеном. Хоча

нам зараз було б дуже цікаво знати думку цього генія, який по суті сам був живим комп'ютером: фон Нейман запам'ятовував все що бачив і чув, та міг робити складні розрахунки «у голові» швидше ніж інші люди на арифмометрі. Крім всього іншого, що встиг зробити фон Нейман - це сформулювати принцип, за яким і зараз функціонують комп'ютери. На жаль, передчасна смерть перервала творчий шлях одного з найвидатніших вчених XX - го століття.

Хоча основна проблема була в інноваційній новизні ділового комп'ютера, успіху справи не сприяли ще й деякі суб'єктивні моменти. А.Ауербах, один з перших співробітників компанії, згодом досить критично відгукнувся про управлінські здібності батьків-засновників: "На мою думку, Еккерт і Моучлі не були компетентними менеджерами і взагалі не розуміли, як вести бізнес. Вони були провидцями, блискучими інженерами, але не дозволяли кому-небудь керувати діяльністю компанії в тій сфері, в якій самі мало що розуміли". Справа йшла так важко, що виникла загроза банкрутства і втрата репутації молодою компанією. Власникам компанії не залишалося нічого іншого, як продати її. Спочатку вони звернулися до президента корпорації IBM Томаса Джона Уотсона-старшого. За свідченням співробітника корпорації Ґерберта Ґроша той вирішив заручитися думкою авторитетів. Один з них, професор Колумбійського університету Джон Еккерт теж був прихильником РПК і прихильником розробки спеціалізованих, а не універсальних обчислювальних машин. Він рекомендував відмовитись від пропозиції, що Т.Уотсон і зробив, що і не дивно, враховуючи те, що справи IBM і так йшли чудово без цього нового клопоту. Основним і найприбутковішим бізнесом було виробництво табуляторів і перфокарт. Саме перфокарти, що замовлялися мільйонами, довели прибуток IBM до величезних розмірів.

"Після цього, - пише Ґрош, - двоє невдах попрямували на яхту Джима Ренда(президента фірми Remington Rand (RemRand) - багаторічного і основного конкурента IBM на ринку РПК) до Флоріди і підписали угоду про продаж/купівлю EMCC". Як стверджують свідки цієї події, рішення про підтримку проекту Еккерта і Моучлі було продиктовано не стільки мотивами отримання прибутку, скільки потенційною можливістю насолити конкуренту. Проте, так чи інакше фінансова і організаційна підримка Remington Rand дозволила реалізувати намічений проект в досить короткий термін.

1.2 Становлення основи ЕОД – універсальних електронних обчислювальних машин

Век шествует путем своим железным;
В сердцах корысть, и общая мечта
Час от часу насущным и полезным
Отчетливей, бесстыдней занята.
(А.Баратынский)

Остаточна витратність розробки і виготовлення першого зразка машини склала 930 тис. дол., але Джеймс Ренд не помилився, купуючи EMCC: згодом було виготовлено ще 45 UNIVAC'ів, виручка від їх продажу державним установам і приватним компаніям з надлишком відшкодувала ці витрати (вартість подальших екземплярів машини лежала в межах 1,25-1,5 млн. дол.).

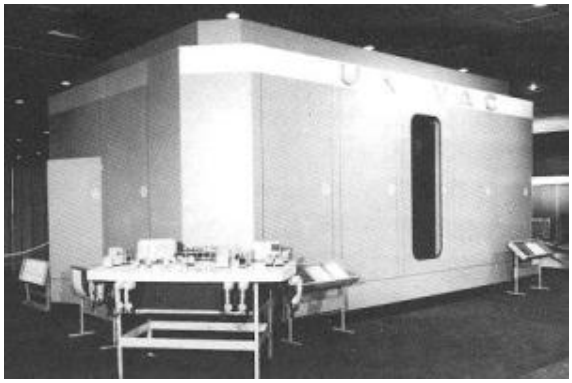


Рис. 1.10. Процесор ЕОМ UNIVAC I

За сучасними мірками UNIVAC (UNIVers Automatic Computer) був справжнім монстром. Його системний блок вагою в 13 тонн включав п'ять із гаком тисяч електронних ламп і займав приміщення розміром з добрячий автомобільний гараж. В центральний обчислювальний вузол, тобто процесор, UNIVAC можна було навіть ввійти через спеціально передбачені двері - інші декілька дверей розміщувалися всередині, розділяючи вузли машини. При тактовій частоті в два з лишком мегагерца машина споживала 125 кіловатів електричної потужності, так що система охолоджування була комбінованою повітряно-водяною. Коштував такий протомейнфрейм 750 тисяч доларів. Швидкісний друкуючий пристрій, яким його почали комплектувати декілька років пізніше, обходився покупцям ще в двісті тисяч.

Вперше UNIVAC виконав тестову програму 29-30 березня 1951 р. і 14 червня того ж року був представлений публіці. Але ця подія залишилася майже непоміченою: лише газета New York Times повідомила, не згадуючи назву машини, “про математичного генія 8-футової висоти, покликаного вирішити проблеми Бюро перепису”. UNIVAC залишалася у виробника приблизно рік, виконуючи роботи за завданнями замовника, потім була демонтована, перевезена в офіс Бюро перепису у Вашингтоні і знову введена в дію, де успішно використовувалася для обробки результатів перепису, перейнявши таким чином естафету від РПК. Машина експлуатувалася цілодобово сім днів на тиждень (не рахуючи щотижневу профілактику, на яку відводилося 8 годин). Всього машина пропрацювала в Бюро перепису близько 73 тис. годин; на початку жовтня 1963 р. вона була демонтована і передана як експонат Музею американської історії при вашингтонському Смітсоновському інституті.

Декілька наступних екземплярів машини було придбано Пентагоном і Комісією з атомної енергії (Atomic Energy Commission), а в 1953 р. корпорація General Electric стала першою приватною компанією, що купила UNIVAC. Саме розробниками цієї фірми був зроблений перший прорив у сферу застосування ЕОМ у бізнесі. General Electric вирішила за допомогою цієї ЕОМ автоматизувати роботу заводів в Луїсвілль (шт. Кентуккі), причому не тільки складський, бухгалтерський облік і виписування рахунків, але і задачі планування виробництва. В результаті в 1954-м група програмістів почала створення першої комп'ютерної системи управління. Розробка велася в машинних кодах і абсолютних адресах! Ніхто у той час навіть приблизно не уявляв, скільки часу буде потрібно на відлагодження більш-менш серйозної програми. Тим не менше героїчні зусилля розробників GE привели до того, що до кінця 1955 р. комп'ютерна система почала функціонувати і роздруковувати правильні рахунки, - правда, цей процес займав більше часу, ніж їх підготовка вручну, і інтерес до корпоративного застосування ЕОМ в США на декілька років підупав, проте не зник – принципова можливість застосування комп'ютерної обробки в бізнесі була доведена. Проте процес пішов і згодом ряди покупців поповнилися страховими компаніями Pacific Mutual Insurance, Metropolitan Life і Franklin Life, промисловими гігантами (DuPont, U.S. Steel, Westinghouse Electric, Alcoa, Pittsburgh Plate Glass) і деякими іншими комерційними організаціями. У другій половині 50-х RemRand безоплатно передала машини Гарвардському і Пенсільванському університетам, а також Кейсовському технічному інституту (Case Institute of Technology) в Клівленді.

Широкій публіці UNIVAC стала відома після 4 листопада 1952 р. Цього дня американці обирали свого президента, і машина, обробивши результати попереднього підрахунку голосів на деяких виборчих ділянках, передбачила перемогу кандидата від республіканської партії Дуайта Ейзенхауера над демократом Едлаєм Стівенсоном. Оскільки проведені раніше опити громадської думки говорили про зворотнє, керівники вечірніх випусків новин телевізійної мережі CBS заборонили видавати в ефір прогнози UNIVAC'а і вважали за краще почекати офіційних результатів. А після півночі, коли стало ясно, що машинний прогноз виправдався, популярний телекоментатор Уолтер Кронкайт повідомив про це на всю країну. Така мимовільна реклама зробила ім'я машини Еккерта-Моучлі настільки популярним, що надалі торгова марка UNIVAC надавалася і обчислювальним машинам іншої компанії, придбаною Рендом, а для широкої публіки протягом деякого часу ця назва стала синонімом "обчислювальної машини".

Успіху UNIVAC'а багато в чому сприяло програмне забезпечення, що поставлялося разом з ним, і в зв'язку з цим не можна не згадати легендарного програміста Ґрейс Мюррей Хоппер (Grace Murray Hopper, 1906-1992), яка прийшла в компанію в 1949 р. У 1951 р. очолювана Хоппер група програмістів розробила принципово новий транслятор мови А-0,

який вона назвала компілятором. Коли в 1957 р. Sperry Rand вирішила продавати компілятор окремо від ЕОМ, йому дали звучнішу, на думку маркетологів, назву - MATH-MATIC. На відміну від інтерпретації тексту по рядках, компілятор спочатку цілком аналізував всю програму, записану в мнемонічних позначеннях, приєднував до неї у разі потреби підпрограми з бібліотеки (що зберігалася в UNIVAC'і на магнітній стрічці) і лише потім здійснював перетворення скомпонованої таким чином програми в машинні коди; при цьому отримана програма могла або виконуватися відразу, або зберігатися для подальшого використання, причому транслююча програма могла бути видалена з пам'яті (слід згадати, що приблизно в цей же час незалежно від групи Хоппер аналогічний транслятор розробив в Британії Р.Брукер). Застосування компілятора значно прискорювало виконання програми (в порівнянні з її інтерпретацією) і надалі стало основним способом трансляції мов програмування високого рівня (МПВР). Наступний крок Грейс Хоппер і її співробітники зробили в 1956-му: вони склали список приблизно з 30 слів, що утворили мову програмування високого рівня під назвою FLOW-MATIC, а потім створили компілятор, який транслював в машинний код програми, написані на цій мові. Заради справедливості треба сказати, що перша МПВР (Фортран) була розроблена за декілька років до Хоппер під керівництвом іншого видатного програміста - Джона Бекуса. Ще через чотири роки Грейс Хоппер стала ініціатором організації групи програмістів з різних компаній, яка отримала назву “Асоціації мов для систем даних” (Conference on Data Systems Languages, CODASYL). У січні 1960 р. ця група розробила “Універсальну мову, призначену для вирішення ділових завдань” (Common Business Oriented Language, COBOL), модифікації якого користуються популярністю і у наш час. До речі, одним з активних учасників CODASYL’а був Роберт У.Бемер, який прославився як автор стандартного коду для обміну інформацією(American Standard Code for Information Interchange, ASCII), що став міжнародним стандартом де-факто.

Таким чином на кінець 50-х - початок 60-х були створені реальні передумови для реального впровадження комп’ютерної технології в бізнесову сферу, в першу чергу завдяки розробці архітектури комп'ютера та спеціально орієнтованих мов програмування високого рівня, в першу чергу COBOL, що знаменувало корінний поворот в обробці даних.

Парадоксом виявилось те, що фірмі Remington Rand не вдалось скористатись своєю перевагою на першому етапі комп’ютерної гонки. В даній роботі важко виявити всі причини цього явища, але факт є факт – фірма IBM зуміла зробити правильні висновки з допущених помилок і за лічені роки зайняла домінуючі позиції в комп'ютерній індустрії не тільки США, але і всього світу

З 60-х років 20-го сторіччя за IBM закріпилася назва Blue Giant(Голубий гігант). IBM домоглася домінуючої позиції на новому ринку з середини 1950-х рр. Успіх фірми був визначений її попереднім розвитком в епоху табуляторів: були підготовлені кваліфіковані

кадри, які спеціалізувалися на продажах пристроїв для обробки інформації, створена клієнтська база, побудовані виробничі потужності, і найголовніше – накопичений потужний творчий потенціал, який довгі роки забезпечував лідерство IBM. Але головним чинником подальших гучних успіхів IBM полягало у правильній інноваційній стратегії та рішучості по її реалізації. Чого не скажеш про її головного на той час конкурента Remington Rand .

Як видається, головна причина невдачі Remington Rand полягала у відсутності прийнятної стратегії для малого і середнього бізнесу. Комп'ютери, що виготовлялися ERA, не отримали широкого розповсюдження через слабку технічну підтримку і відсутність реклами. Ринок же наукових комп'ютерів мав меншу потенційну місткість порівняно з ринком комп'ютерів для бізнесу, тому навіть успішні продажі комп'ютера UNIVAC і його добре відома торгова марка не змогли врятувати ситуацію. Компанії була потрібна вдала модель для комерційного ринку, що розширювався, проте керівництво Remington Rand не змогло, чи не захотіло вирішити цю задачу. В результаті, в 1955 р. компанія Remington Rand була куплена фірмою Sperry Gyroscope, яка займалася виробництвом устаткування за військовими контрактами. А в 1986 р. фірма SperryRand об'єдналася з компанією Burroughs і змінила назву на Unisys, повністю залишила ринок мейнфреймів, і в даний час займається наданням послуг в області інформаційних технологій.

Треба зазначити, що на той час IBM вже мала досвід власної розробки комп'ютера, зокрема під керівництвом вже згадуваного Г.Ейкена і фінансово-технічній підтримці IBM була проведена розробка електромеханічного комп'ютера Mark I. В довжину він досягав 17 м. і у висоту більше 2,5 м, містив близько 750 тис. деталей, сполучених дротами загальною протяжністю близько 800 км. Згодом машина була передана у ВМФ і використовувалася для виконання складних балістичних розрахунків. Про якесь бізнесове застосування не йшлося.

Оскільки керівництво фірми в особі Т.Уотсона-старшого не поспішало з переорієнтацією фірми на виробництво комп'ютерів, інженери компанії настирливо прагнули викликати у адміністрації інтерес до комп'ютерної революції, що відбувається, і, нарешті, їм вдалося знайти союзника в особі сина голови її правління Томаса Дж. Уотсона-молодшого, що вже займав посаду віце-президента IBM. Його вплив згодом втілився в поступовому переході компанії до виробництва комп'ютерів. Для того, щоб не йти на пряму конфронтацію з батьком, Уотсон-молодший придумав хитромудрий хід: він затвердив інструкцію, яка давала перевагу людям з вченим ступенем в математиці, фізиці або електроніці при прийомі на роботу в IBM - це дозволило створити творчий потенціал не тільки екстреної програми по розробці комп'ютера, початій в 1951 р., але і подальших розробках електронної обробки даних. Про розмір інтелектуального потенціалу наукової команди, зібраної Уотсоном-молодшим, свідчать не тільки практичні здобутки IBM, але й більше десятка найпрестижніжих премій в наукових сферах, зокрема премії Нобеля та Т'юринга.

Але для суттєвих змін був потрібен імпульс. Таким імпульсом стала війна в Кореї. Під час Корейської війни, Уотсон-старший підписав ряд контрактів з урядовими структурами. Для IBM співпраця з урядом була звичною справою – під час Другої світової війни корпорація забезпечувала американську армію табуляторами, які прискорювали обробку документів, породжених загальною мобілізацією, а ще раніше, під час Великої кризи забезпечувала облік величезної армії безробітних.

А Т.Уотсон-молодший і його співробітники, використали цю можливість для розробки комп'ютерів для урядових організацій. Були укладені договори на поставку ЕОМ, що отримали назву Defense Calculator. Саме при реалізації цього проекту сталася непомітна для широкого кола, але по суті революційна подія – використання в цих машинах магнітної стрічки для зберігання інформації. Це стало на перший погляд непомітним, але вирішальним фактором переходу IBM в комп'ютерну сферу, а заодно якісну зміну в електронній технології обробки даних, бо магнітний носій дозволяє організувати багаторазовий запис та зчитування. То був перший, ще далеко не остаточний розрив з перфокартами. Згодом комп'ютер був перейменований в Model 701 і під цим ім'ям успішно продавався на комерційному ринку, бо технологічно був для цього придатний. З 1955 р. почалися відвантаження Model 702, а пізніше і її поліпшеної версії Model 705. Про грандіозний успіх серії 700 говорить також той факт, що ця лінійка переросла в нову лінійку 7000, коли новим базовим елементом замість вакуумних ламп стали транзистори. З 1955 р. кількість встановлених комп'ютерів 700-й серії вперше перевищила кількість комп'ютерів, що поставила Remington Rand.

У IBM, також як і у Remington Rand, існувало дві лінійки комп'ютерів - для урядових організацій і для крупного, малого і середнього бізнесу. Лінійка 700 була першим напрямом, а другим була Model 650, призначена для малого і середнього бізнесу, ціна аренди якої складала всього лише 3 250 дол. на місяць. Було продано більше тисячі таких комп'ютерів. Таким чином Model 650 можна назвати першим комп'ютером, що випускався як масовий товар. В середині 50-х рр. IBM домоглася домінуючого положення на обох сегментах комп'ютерного ринку, що зароджувався. Одним з показників успіху було те, що в 1956 р. для прогнозування результатів виборів використовувалася вже технологія IBM. Але справжні успіхи IBM припадають на 60-ті роки, а разом з цим і подальший прогрес в електронній обробці даних, зокрема пов'язаний з новою елементною базою – тразисторами.

Але не можна не згадати і розробки поза територією США. Тут треба віддати належне досягненню колективу розробників у Великобританії, очолюваних Стаффордом Біром, якого зараз вважають батьком економічної кібернетики. Працюючи в найбільшій англійській сталеливарній корпорації United Steel, він створив дочірнє підприємство з символічною назвою CyborHouse. Там колектив учених, яких Стаффорд Бір очолював, у 1956 році

застосував комп'ютер під назвою Ferranti Pegasus для управління виробництвом. Значно пізніше, на початку 70-х років Бір керував розробкою кібернетичної системи управління національною економікою Чілі.

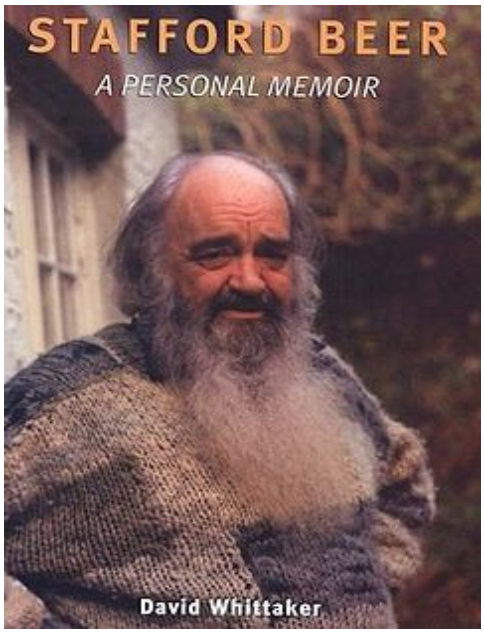


Рис. 1.11. Ентоні Стаффорд Бір

Не можна залишити поза увагою процеси у цій сфері в колишньому СРСР, особливо в Україні. В жовтні 1948 р. під керівництвом академіка С.А.Лєбєдєвва в Києві почалася розробка Малої електронної обчислювальної машини МЕСМ.

Рис. 1.12. С.О.Лєбєдєв

Незалежно і одночасно із західними ученими їм були розроблені основні принципи побудови ЕОМ. І вже в 1951р., коли МЕСМ була введена в експлуатацію в Інституті електротехніки АН УССР, Україна стала другою в Європі країною(або, як любили писати в свій час пропагандисти, першою на континентальній Європі) і третьою у світі, в якій було створено справжній комп’ютер(перший в світі комп’ютер, як відомо, був створений у Великобританії).

Рис. 1.13. Обчислювальна машина МЕСМ

В подальші три десятиріччя вагомий внесок у фундаментальні дослідження в галузі комп'ютерної науки і техніки був зроблений вченим з світовим ім'ям академіком Віктором Михайловичем Глушковим - основоположником інформатики в Україні, і багатьма провідними ученими заснованого Глушковим Інституту кібернетики НАН України. Тут же був виконаний цілий ряд важливих прикладних досліджень, направлених на створення нових ЕОМ і їх використовування в піонерських системах управління технологічними процесами, енергетичними та іншими об'єктами, у тому числі військового призначення, для систем автоматизації наукового експерименту та ін. Більш третини обчислювальної техніки, що серійно випускалась, в Радянському Союзі були розроблені в Інституті кібернетики НАН України. Починаючи з 60-х років проектування і великосерійне виробництво ЕОМ для управління технологічними процесами і енергетичними об'єктами з великим розмахом здійснювалося в Сєвєродонецькому науково-виробничому об'єднанні “Імпульс”. Переважна більшість управляючих систем промислового призначення в СРСР була розроблена з участю “Імпульсу”.

Треба зазначити особливу рису київських розробок засобів обчислювальної техніки тих часів - їх оригінальність, що дуже сильно контрастувало з розробками комп’ютерної техніки пізніших періодів. В ті роки в Україні оформилася і плідно працювала вітчизняна науково-технічна школа в галузі обчислювальної техніки, де в першу чергу треба відзначити академіка Сергія Олексійовича Лєбєдєва, під керівництвом якого в Україні була створена перша ЕОМ - Мала електронна обчислювально-обчислювальна машина (МЕСМ). Дослідження в енергетиці, чим спочатку займався С.О. Лєбєдєв, вимагали великої кількості обчислень, і тому його інтереси перемістилися у бік автоматизації обчислень. До Києва Лєбєдєв був запрошений в 1946 р. на посаду директора Інституту електротехніки (спочатку енергетики) Академії наук УРСР. Після від'їзду Лєбєдєва до Москви, частина його учнів, що залишилася в Києві, розробила ЕОМ «Київ». Машина хоча і поступалася за

характеристиками новій лебедєвській ЕОМ М-20, розробленій в Інституті точної механіки в Москві, але цілком відповідала вимогам того часу.

В 1958 р. колишню лабораторію С.А. Лєбєдєва очолив В.М.Глушков. Під його керівництвом успішно завершилася розробка ЕОМ «Київ», яка довго використовувалася в Обчислювальному центрі АН України, розгорненому на базі лабораторії. Інший її екземпляр був закуплений Об'єднаним інститутом ядерних досліджень, де також довго і успішно експлуатувався. Створений в 1957 р. Обчислювальний центр АН України в 1962 р. був перетворений в Інститут кібернетики, який сьогодні носить ім'я його творця - В.М. Глушкова, що продовжив справу, розпочату С.А. Лєбєдєвим.

Рис. 1.14. В.М.Глушков

Слідом за ЕОМ «Київ» у ОЦ НАН України була розроблена перша в Україні (і в Радянському Союзі) напівпровідникова управляюча машина "Дніпро". Паралельно із створенням «Дніпра», рядом підприємств України була проведена велика підготовча робота по застосуванню машин для управління складними технологічними процесами. Разом із співробітниками металургійного заводу ім. Дзержинського (Дніпродзержинськ) досліджувалися питання управління процесом виплавки сталі в бессемерівських конверторах, із співробітниками содового заводу в Славянську - колони карбонізації та ін. Потім з'ясувалося, що американці дещо раніше почали роботи по створенню універсальної управляючої машини, аналогічній «Дніпру», але запустили її у виробництво в червні 1961 року, практично одночасно. Окремо треба сказати про розвиток програмного забезпечення, зокрема ступінчате мікропрограмне управління було використано в машині для інженерних розрахунків, скорочено - МИР-1, створеної слідом за ЕОМ «Промінь» (1965 р.).

В 1967 році на виставці в Лондоні, де демонструвалася МИР-1, вона була закуплена представниками самої IBM, яка на той час контролювала майже 80% ринку обчислювальної техніки всього світу(крім СРСР і їх союзників). Це була перша (і остання) закупівля радянської електронної машини американською кампанією. Як з'ясувалося пізніше, менеджери IBM купили машину не стільки для того, щоб працювати на ній, скільки для того, щоб довести в суді своїм конкурентам, які запатентували в 1963 р. принцип ступінчатого мікропрограмування і позивалися до IBM, що навіть в СРСР давно про цей принцип знали і реалізували в машині, яка серійно випускається. У 1969 р. була прийнята у виробництво

нова досконаліша ЕОМ МИР-2. Потім була розроблена МИР-3. По швидкості виконання аналітичних перетворень їм не було конкурентів. МИР-2, наприклад, успішно змагалася з універсальними ЕОМ звичайної структури, що перевершують її по номінальній швидкодії і об'єму пам'яті в сотні разів. На цій машині вперше в практиці вітчизняного математичного машинобудування було реалізовано діалоговий режим роботи з використанням дисплею з світловим пером. З сучасного розуміння тенденцій розвитку комп'ютерних технологій розробники МИР стояли на порозі створення персонального комп'ютера. Але, на жаль, так і не створили.

1.3 Революція IBM

Двадцатый век… Ещё бездомней,
Ещё страшнее жизни мгла
(Ещё чернее и огромней
Тень Люциферова крыла).
Сознанье страшное обмана
Всех прежних малых дел и вер,
И первый взлет аэроплана
В пустыню неизвестных сфер...
(А.Блок)

Однак фактом є те, що магістральний шлях комп'ютерної технології був прокладений саме в США. Більше того, цей напрямок визначила в 60-х роках по-суті одна фірма – IBM. Підґрунтям цього великомасштабного бізнесу був процес широкого впровадження інформаційних систем в різних сферах економіки США та інших розвинутих країн. Якщо в 50-х роках тільки лічена кількість компаній пішла на ризик по вкладенню інвестицій в сферу комп'ютерних технологій для застосування в бізнесі, то в 60-х роках цей процес набув масового характеру. Головна причина коріниться в черговій прояві тієї ж кризи контролю.

Винайдення у 15-му столітті Лукою Паччолі подвійної бухгалтерії упорядкувало неорганізований до цього стан речей в комерційній діяльності. Бухгалтерська модель представляє економічну дійсність наборами рахунків, які фактично є продуманим розділенням однієї грошово-кредитної рівності (Активи = Зобов'язання + Акціонерний капітал) і фактично зводить задачу обліку до відомої шкільної задачі про басейн з двома трубами, з однієї вода затікає в басейн, а іншої витікає. Ця схема залишається правильною і сьогодні, але проблема з'являється там, де ускладнюються організаційні структури та підвищується швидкість змін. Система Паччолі досить добре обслуговує бізнес-структури більше 400 років, але на початку минулого сторіччя відчулася потреба у зміні технологій обліку та планування діяльності фірм, тобто черговий раз проявила себе криза контролю.

Впродовж XX в. у більшості країн стали регламентувати бухгалтерський облік і навіть управляти його організацією і методологією, перш за все, для цілей оподаткування, а також в прагненні захистити власників від можливостей бухгалтерськими методами спотворити правду про їх майновий стан, зобов'язання і фінансові результати діяльності комерційних підприємств, що їм належать. Особливу увагу цьому аспекту приділяли держави з централізованою керованою економікою, де головним завданням бухгалтерії було забезпечення збереження державної власності, режиму економії у витрачанні державних коштів, дотримання державної дисципліни у всьому. В СРСР панувала думка, що бухгалтерія соціалістичного підприємства – «государева служба» по обліку і контролю, а бухгалтер – «казны народной часовой». Показники бухгалтерського обліку мало використовувалися для внутрішнього управління організацією. За даними бухгалтерського балансу і інших форм річної звітності основні рішення по управління тим чи іншим підприємством ухвалювалися на рівні главку, міністерства або іншого зовнішнього органу управління.

Ринкова економіка вимагає принципово іншого управління і інших керівників. Ними стають менеджери господарюючого суб'єкта, яким потрібний облік для внутрішньоорганізаційного управління. Становленню сучасного управлінського обліку сприяла розробка ідей методу стандарт-кост і систем управління по відхиленнях. Історично стандарт-кост передував обліку по нормативах і нормативному обліку фактичної собівартості. Термін «стандарт-кост» (standart-cost) вперше застосував Емерсон – автор широко відомої свого часу теорії продуктивності[].

Початок практичному використанню системи управлінського обліку належало промисловим компаніями США (20-х рр. XX ст.). Ділових людей привернула, забезпечувана цим методом, можливість вибору оптимального варіанту витрат, прогнозування витрат на майбутні періоди, контролю їх рівня шляхом зіставлення фактичних значень із заданими, виявлення і аналізу відхилень. При порівняно менших витратах на ведення обліку за фактичною собівартістю вони отримували інформацію для оперативного цілеспрямованого внутрішньофірмового управління по відхиленнях. Пізніше з використанням принципів стандарт-кост була розроблена теорія управління по відхиленнях. Відтак все це вимагало ефективної інформаційної технології.

Але технологія обробка даних, базована на РПК не була здатна вирішувати ці нові проблеми, оскільки за своїм задумом була пристосована до обробки великих масивів однотипних даних, нанесених на перфокарти, які в плановому порядку доставлялися на РПК. Про якісь більш–менш суттєві зміни в структурі вхідних даних не йшлося, як і про якусь швидку реакцію на зовнішні зміни. А темпи змін в економіці все наростали і вже на початок 60-і рр. XX –го століття в найбільш розвинутих країнах, в першу чергу США, стало ясно, що традиційні прийоми і способи роботи з інформацією вже вичерпали себе, і люди все більш і

більш демонстрували свою нездатність справлятися з наростаючою «інформаційною лавиною». В цей час в США паперове діловодство стрімко збільшувалося, темпи його зростання перевершували в три рази темпи приросту валового продукту країни. З'явився навіть термін: «паперове забруднення середовища». Споживання паперу кожні чотири роки подвоювалося.

Технологія IBM на той час найкраще відповідала цим новим вимогам, бо базувалася на принципах обробки економічних даних за допомогою програм, що зберігаються в пам'яті, розвинутій операційній системі, алгоритмічним мовам та надійним пристроям вводу/виводу. Комп'ютери серії 70-х і 14хх принесли IBM широку популярність, а об'єм продаж за шість років подвоївся (збільшився з 1,17 млрд. дол. в 1958 р. до 2,31 млрд. дол. в 1964 р.), темпи зростання продажів склали 30% щорічно. За даними журналу «Datamation», в 1961г. вже 81,2% комп'ютерного ринку належала IBM. Успіху IBM на новоствореному комп'ютерному ринку сприяли наступні три головні фактори:

- Серйозні власні наукові дослідження і дослідно-конструкторські розробки (НДДКР), внаслідок чого фірма стала власником ключових патентів. Витрати на НДДКР були збільшені з 15% чистого доходу в 1940 р. до 35% в 1950 р. і 50% в 60-70-х рр. З 1960 р. бюджет IBM на НДКР в області комп'ютерів перевершив федеральний бюджет в цій сфері.
- Орієнтація компанії на збут. Компанія мала величезний досвід збуту і обслуговування комплексних систем, якого не було у конкурентів.
- Власне виробництво периферії (особливо накопичувачів на магнітній стрічці і високошвидкісних принтерів).

IBM не іґнорувала потенційних клієнтів, як це робили багато фірм, що фокусувалися виключно на потужних наукових комп'ютерах. IBM, розвиваючи наукову сферу, ніколи не забувала про комерційні ринки та про потенційний зв'язок між ними. Кінець кінцем, успіху в комп'ютерному бізнесі допомогла добитися основна традиційна стратегія фірми - входження одночасно на декілька ринків і обмін досвідом між ними.

Кроки, зроблені в цей період, мали своєю кульмінацією System/360, лінійку комп'ютерів, яка була покликана замінити всі комп'ютери IBM, що раніше випускалися. System/360 була сімейством комп'ютерів, що виконували одні й ті ж програми, містили стандартні електронні схеми, повністю виготовлені IBM. Назва «360» означала 360 градусів (повний круг) і символізувала те, що нові комп'ютери вирішуватимуть все коло поставлених задач: обчислення для наукових, військових і комерційних організацій.

Рис. 1.15. Машинний зал обчислювального центру на базі System/360

В своїх ранніх комп'ютерах IBM спочатку використовувала транзистори, отримані за ліцензією від Texas Instruments. Але згодом керівництво IBM ухвалило рішення випускати всі електронні компоненти самостійно, щоб не залежати від зовнішніх постачальників і забезпечити максимально низькі ціни на електронні елементи. При розробці System/360 керівництво IBM вирішило не покладатися на інтегральні схеми, винайдені ще в 1959 р., оскільки це була нова, ще не апробована і дорога технологія, натомість в IBM була розроблена гібридна технологія SLT (Solid Logic Tecnology).

Не менш вражаючим була розробка операційної системи IBM/360. Програмна сумісність вимагала і сумісності операційних систем. Такі операційні системи мали працювати і на великих, і на малих обчислювальних системах, як із великою так і з малою кількістю різноманітної периферії, в комерційній області і в області наукових досліджень. Операційні системи, побудовані з намірами задовольнити всім цим суперечливим вимогам, виявилися надзвичайно складними «монстрами». Вони складалися з багатьох мільйонів асемблерних рядків, написаних тисячами програмістів, і містили тисячі помилок, які породжували нескінченний потік виправлень. В кожній новій версії операційної системи виправлялися одні помилки і вносилися інші. Проте, не дивлячись на неохопні розміри і масу проблем, OS/360 і інші їй подібні операційні системи машин третього покоління дійсно задовольняли більшість вимог споживачів того часу. Найважливіше досягненням ОС даного покоління полягало в реалізації мультипрограмування. Мультипрограмування - це спосіб організації обчислювального процесу, при якому на одному процесорі поперемінно виконуються декілька програм. Поки одна програма виконує операцію введення-виведення, процесор не простоює, як це відбувалося при послідовному виконанні програм

(однопрограмний режим), а виконує іншу програму (багатопрограмний режим). При цьому кожна програма завантажується в свою ділянку оперативної пам'яті, що зветься розділом.

Інше нововведення - спулінг (spooling). Спулінг у той час визначався як спосіб організації обчислювального процесу, відповідно до якого завдання прочитувалися з перфокарт на диск в тому темпі, в якому вони з'являлися в приміщенні обчислювального центру, а потім, коли чергове завдання завершувалося, нове завдання з диска завантажувалося в розділ, що звільнився. Разом з мультипрограмною реалізацією систем пакетної обробки з'явився новий тип ОС - системи розділення часу. Варіант мультипрограмування, вживаний в системах розділення часу, націлений на створення для кожного окремого користувача ілюзії одноосібного використання обчислювальної машини.

Випуск System/360 був ризикованим кроком з боку IBM. Справді, тільки витрати на розробку машини склали 500 млн. дол., проте, це була тільки «верхівка айсберга». В цілому компанія вклала, за оцінками експертів, за 4 роки 5 млрд. дол. в будівництво шести заводів, розташованих в різних країнах світу. Це була безпрецедентна, дійсно найдорожча в історії програма, що фінансувалася однією компанією. Ця програма об'єднала близько 2000 інших, її виконання привело також до збільшення загальної чисельності зайнятих на підприємствах IBM на 50 тис. людей. Проект завершився вдало, що зробило IBM практично недосяжною на ринку комп'ютерів на довгий час. Проте, можливо, тільки зараз можна оцінити всю глибину стратегічного задуму керівництва IBM, а заодно і рівень ризику. Наскільки це був ефективний стратегічний маневр, свідчить історія про спроби деяких компаній протистояти IBM. Наприклад, потужна фірма General Electric(GE), яка була піонером застосування комп’ютерів в управлінні бізнесом і мала серйозні розробки у цій сфері, пробувала реалізувати план виходу на комп’ютерний ринок. Але GE увійшла до комп'ютерного бізнесу надто пізно і їй не судилося зайняти домінуючі позиції. На цій стадії гри наздогнати IBM можна було тільки дуже великою ціною, якщо взагалі можливо. Тому нове керівництво GE прийняло рішення вийти з комп'ютерного бізнесу, що корпорація кінець кінцем і зробила, продавши цей напрям Honeywell.

Початкові інвестиції IBM обсягом біля 5 млрд. дол. швидко окупилися, оскільки замовлення на систему протягом двох років досягли 1000 штук в місяць. System/360 потроїла кількість комп'ютерів IBM, що використовуються, і удвічі збільшила доходи компанії в подальші п'ять років. Попит на нові комп'ютери була настільки високий, що багато компаній купували місця в черзі на поставку нових комп'ютерів IBM.

Причини, які стояли за стрімким зростанням попиту на комп’ютерну техніку та інформаційну технологію не були звичайною модою, хоча і таке траплялось і трапляється неодноразово. Просто стрімке зростання рівеня економіки викликало черговий прояв кризи контролю, яку можна подолати за рахунок застосування інформаційних технологій на базі

комп'ютерів. Білл Ґейтс у своїй книзі « Бізнес зі швидкістю думки» приводить промовистий приклад зі знаменитою фірмою-автогігантом General Motors. Довгий час, з 1923 по 1956 рік, біля керма компанії General Motors перебував А.Слоун. За час правління А.Слоуна GM перетворилася на одне з перших по-справжньому масштабних ділових підприємств США. Слоун розумів, що розробляти оптимальну стратегію розвитку і робити правильні дії неможливо без опори на факти та інтуїцію всіх співробітників компанії. Своє розуміння бізнесу він виробив в ході тісної особистої співпраці з інженерами і менеджерами своєї компанії і регулярних відвідин її виробничих цехів. Проте найбільший вплив на діяльність компанії зробили запропоновані А.Слоуном методи створення ділових зв'язків з ділерами GM в масштабі всієї країни. Ця ділерська мережа дісталася Слоуну від його славетного попередника Джурана. Він постійно збирав інформацію у ділерів і вибудовував з ними тісні, плідні відносини. Його цікавили не тільки процеси постачання продукції GM ділерам, але і те, як у них йде продаж цій продукції покупцям. А.Слоун в кінці 20-х років усвідомив, що автомобільний бізнес зазнає серйозних змін. Базовим транспортним засобом для основної маси населення стали автомобілі, що були у вжитку. Нові, високоякісні машині купували покупці середнього достатку, користуючись при цьому можливостями заліку вартості старого автомобіля і оплати в розстрочку. Слоун зрозумів, що такі зміни вимагають і істотної зміни характеру відношень GM з ділерами, що відображає перехід автомобільного бізнесу від елементарного збуту продукції на складнішу систему торгівлі і сервісу. Виробник і ділери повинні були стати, по суті, партнерами.

Проте отримання точної інформації про продажі, як і раніше, було утруднене. Дані про об'єми збуту GM були суперечливими, застарілими і неповними. «Коли прибутковість торгівлі якого-небудь ділера виявлялася нижче очікуваною, ми не могли визначити причину - проблема могла бути пов'язана з новими або уживаними автомобілями, сервісом, запасними частинами, або ще з чим-небудь. А без таких відомостей неможливо було реалізувати стабільну політику розповсюдження продукції. Як висловлювався Слоун, він готовий заплатити «скажені гроші» і відчувати «повну виправданість таких витрат», якби кожен ділер «знав фактичні дані про свій бізнес і міг осмислено враховувати всі дрібні деталі його ведення». Слоун вважав, що допомога ділерам у вирішенні таких інформаційних проблем була б «найкориснішою» зі всіх інвестицій General Motors.

Для вирішення цієї проблеми Слоун ввів у GM і для всіх її ділерів стандартизовану систему обліку. Ключове слово тут - «стандартизовану». Всі ділери компанії і співробітники всіх її рівнів дістали можливість класифікувати свої дані абсолютно одноманітно. З середини 30-х років ділери GM, автомобільні заводи і штаб-квартира корпорації могли виконувати детальний фінансовий аналіз, спираючись на одні і ті ж цифри. Конкретний ділер, наприклад, міг не тільки оцінювати власну ефективність, але і зіставляти її з середніми

показниками. Тобто така система обліку не просто фіксувала факти, а була спрямована на результат, тобто була системою управлінського обліку.

Завдяки системі управлінського обліку GM стала настільки динамічною організацією, що інші автомобілебудівники десятиліттями не могли навіть наблизитися. Така інфраструктура, яку Б.Ґейтс називає «нервовою системою» компанії, допомогла GM утримувати лідерство на ринку автомобілебудування протягом всієї кар'єри Слоуна в цій компанії. Хоча ця «нервова система» ще не була електронною, а базувалась на стандартизованих картотеках, вона була виключно корисною. GM - як ніхто інший - володіла інформацією про інвентарні запаси ділерів, що і дозволило компанії отримати величезну перевагу в конкурентній боротьбі. Відзначимо, що робота з цією інформацією не обмежувалася рамками власне GM, і, таким чином, впроваджені компанією системи ручного обліку фактично сформували у неї першу з відомих «екстрамереж» – діючу мережу обміну інформацією між GM, її постачальниками і агентами по збуту.

Зрозуміло, в ті часи обсяги даних компаній, що проходять по інформаційних каналах, були відносно невеликі. Але вже в 60-ті роки функціонування серйозної компанії вимагало набагато більших, ніж в часи правління Слоуна, обсягів і швидкостей обробки інформаційних потоків. Занадто багато треба було саджати людей за телефони та картотеки, щоб оперативно відслідковувати всі зміни. Тобто «нервову систему» за термінологією Б.Ґейтса треба було якісно удосконалювати. Управління на основі фактів - один із фундаментальних принципів ведення бізнесу по Слоуну - невідворотно приводить до використання інформаційних технологій. Відповідно GM, як і багато хто з успішних компаній, в 60-х почала впроваджувати інформаційну систему на базі систем IBM. Зерно було посіяне в в благодатний ґрунт.

Рис. 1.16. Динаміка об'єму продаж IBM

Бурхливий розвиток комп’ютерної індустрії не повинен затьмарювати нам головну мету, заради якого вона виникла і розвивалася - для використання в економічних,

промислових, та військових застосуваннях. Треба визнати, що далеко не всі інформаційні системи, побудовані в 60-ті, 70-ті та наступні роки дали належну віддачу. Причин багато. Зокрема не всі компанії, які впроваджували інформаційні системи були до цього готові так само, як General Motors. Проблема виявилася складною і багатогранною. Але центральна проблема без вирішення якої побудова дієвих інформаційних систем немає сенсу, полягала в ефективних технологіях електронної обробки даних. І треба зазначити, що саме в 60-х роках в процесі розробки численних комп'ютерно-інформаційних систем, як успішних , так і невдалих були усвідомлені, а потім сформулювані базові принципи побудови і функціонування систем електронної обробки даних. Саме на цей період припадає формування першої справжньої технології електронної обробки даних, відомої під назвою мейнфрейм(mainframe), серцем якої і був великий у повному розумінні цього слова комп'ютер, бо його габарити вимагали цілого залу, часто немалого. І тому, природньо виникли центри обробки інформації, які довго існували як інформаційно-обчислювальні центри. До певної міри ці інформаційно-обчислювальні центри успадкували принципи організації обробки даних, що існували на РПК. Всі вони працювали в пакетному режимі, для чого треба було завдання у вигляді наборів перфокарт спочатку зібрати, а потім всі разом відправити на пакетну обробку. Проте були принципові відмінності від РПК, зокрема завдання в цьому пакеті могли бути будь-які, аби тільки задовольняли вимогам операційної системи та правилам компіляторів. До того ж складувати стоси перфокарт було необов'язково – вже існували носії на магнітних стрічках та магнітних дисках. З погляду сучасності така технологія була неспроможна остаточно подолати вже згадувану перманентну кризу контролю, але її значення в тому, що вона стала фундаментом подальшого прогресу в сфері електронної обробки даних та проклала дорогу наступним етапам інформаційної революції - в першу чергу саме на цьому етапі кристалізувалися основні принципи електронної обробки даних, які ми розглянемо далі.

1.4 Революція персональних обчислень

Время новое. Новая смена.
Смена сердца?- язык запчастей.
Наша юность еще современна,
Потому что мы помним о ней.
(Светлана Евсеева)

Підвалини технологічної системи обробки даних, в яка функціонує зараз, склалася в 1970-х роках. Власне конкретні траєкторіїї розвитку могли виглядати і інакше, проте, як зазначає М.Кастельс[], ніхто не може заперечити вплив ключових інновацій на стан речей у сучасних інформаційних технологіях. Кастельс відзначає і спільні риси: базуючись головним чином на існуючому знанні і розвиваючись як продовження ключових технологій, вони завдяки доступності і падінню вартості при підвищенні якості є кардинальним проривом в масовому розповсюдженні технології в область комерційних і цивільних застосувань. Так, мікропроцесор, ключовий пристрій для наступних революційних змін був винайдений в 1971 р., коли Тед Хофф, інженер Intel, винайшов мікропроцесор I4004, тобто комп'ютер на чіпі, який почав широко розповсюджуватися в середині 1970-х. Мікрокомп'ютер був винайдений в 1975 р., а перший успішний комерційний варіант був представлений в квітні 1977 р., тобто приблизно в той же час, коли Digital Research намагалася розробляти операційні системи для мікрокомп'ютерів. Перший промисловий електронний комутатор з'явився в 1969 р., цифрове перемикання було розроблене в середині 1970-х років і досягло стадії комерційного розповсюдження в 1977 р. Оптичні волокна були вперше запущені в промислове виробництво Coming Glass на початку 1970-х років. Також до середини 1970-х років Sony почала промислове виробництво відеомагнітофонів на базі відкриттів, зроблених в 1960-х роках в Америці і Англії. Нарешті у 1969 р. Advanced Research Project Agency (ARPA) Міністерства оборони США створило нову, революційну електронну комунікаційну мережу, яка зростатиме протягом 1970-х років, поки не перетворилась на нинішній Інтернет. Цьому дуже сприяв винахід в 1974 р. Серфом і Каном TCP/IP, міжмережевого протоколу, який ввів «шлюзову» технологію, що дозволяє зв'язувати мережі різних типів. Тепер ми можемо зрозуміти, що на початку 70-х років 20 століття була створена критична маса іновацій і революція в комп’ютерній технології – питання часу.

Як ми бачимо у цьому списку відсутні якісь успіхи IBM, хоча її фінансові показники були весь час на високому рівні. Проте, коли в кінці 1970-х стало ясно, що в комп'ютерній індустрії назріла революція, IBM виявилася до неї не готовою. І знову, як і 50-х роках, стратеги IBM проґавили мейнстрім – головний напрямок розвитку комп’ютерної індустрії, який мав яскраво виражений іноваційний характер. Як і на початку 50-х років IBM неадекватно реагувала на появу так званих підривних іноваційних технологій, тобто таких, які загрожували існуванню базових технологій, на яких фірма проводила дуже успішний бізнес. Але на цей раз ситуація виглядала куди серйозніше. В це важко повірити, але Голубий гігант справді міг опинитися на узбіччі прогресу. Керівництво IBM, з одного боку заколисане стабільними прибутками за рахунок практично монопольного становища на ринку комп'ютерних технологій протягом добрих двадцяти років, а з іншого боку, засмикане безкінечними судовими позовами конкурентів і урядових організацій, «проспало»

новий напрямок, який з початку 80-х років і по сьогоднішний день визначає стан комп'ютерної індустрії. Довгий час IBM практично не звертала уваги на появу персональних комп'ютерів і появу абсолютно нових фірм і людей у цій сфері, найпомітнішими з яких були Стів Джобс та Стів Возняк. Ці двоє за лічені роки, почавши з виробництва персональних комп'ютерів у гаражі, довели обсяги продаж комп'ютерів своєї фірми Apple до мільярда доларів США.

Правда, ради справедливості треба сказати, що IBM таки вела власний дослідницький проект, який стосувався персонального комп'ютера, проте пріоритет його був настільки низький, що коли обставини почали вимагати швидко створити конкурентноздатну обчислювальну систему у вигляді персонального комп'ютера, то треба було все починати з нуля. Державної підтримки, як на початку 50-х теж не було. Кажуть, що керівництво IBM примусило взятися вирішувати на цю проблему тільки та обставина, що воно помітило чужі персональні комп'ютери на столах своїх співробітників! До речі, це був не єдиний стратегічний прорахунок. В свій час IBM, зосередившись на мейнфреймах, недостатньо швидко усвідомила, що комп'ютери стають меншими, дозволивши фірмам-початківцям, зокрема DEC, захопити домінуюче положення в секторі міні-комп'ютерів і вирости в компанію з мільярдними річними доходами. В кінці 70-х керівництву IBM стало ясно, що та ж помилка може повторитися, але вже щодо персональних комп'ютерів. Різкий злет Apple не залишився непоміченим найбільшим виробником бізнес-комп'ютерів, і в 1980 р. IBM вирішила стати активним учасником на новому ринку. Проте інтрига зав'язалася набагато раніше - в 1974 р.

Молодий розробник і викладач Гарі Кілделл, знаходячись під глибоким враженням від можливостей мікропроцесора 4040, запропонував Intel свої послуги як програміст. Він був зарахований у штат цієї і на той час впливової електронної компанії, для якої і розробив управляючу програму для першого мікропроцесора. Коли з'явилися мікрокомп'ютери на базі процесора 8080, обладнані такими периферійними пристроями, як монітори, накопичувачі на гнучких дисках і принтери, Кілделл створює для них першу в світі повноцінну операційну систему CP/M (Control Program for Microcomputers), яка отримала широку популярність. CP/M дозволила мікрокомп'ютерам виконувати задачі, які раніше зазвичай призначалися для мейнфреймів. Коли ж Intel вирішила не брати участь у виробництві мікрокомп'ютерів, Кілделл заснував свою власну компанію під «скромною» назвою- Intergalactic Digital Research, більше відому як Digital Research. Виходячи з того, що ринкова ситуація вимагала дуже поспішати та великі ризики зовсім втратити цей ринок, IBM вирішила не витрачати час на власні розробки, крім розробки архітектури самого персонального комп'ютера, а скористатися вже готовими рішеннями. Відтак, вона обрала мікропроцесори Intel 8088 і 8086, монітори Zenith, Microsoft BASIC як мову програмування. Керівництво IBM також збиралась

отримати ліцензію на ОС CP/M, яка була у той час індустріальним стандартом і звернулося з комерційною пропозицією до Digital Research. Але тут коса найшла на камінь.

Після тривалих переговорів, розлючені упертістю Кілделла, представники IBM оголосили йому остаточну пропозицію - 200 тис. долл. Натомість представники IBM зажадали необмежені права на ліцензію без відрахувань від продаж своїх комп'ютерів з встановленою на них CP/M. Але Кілделл знав, що без операційної системи комп'ютери IBM - просто металеві коробки, а CP/M, як він думав, була у той час єдиним «гравцем» на полі ОС для персональних комп'ютерів з 16-розрядними процесорами Intel. Розуміючи, що може диктувати умови, президент DR відхиляє пропозицію IBM. Але він прорахувався. Компанія Digital Research у той час дійсно мала версію CP/M для 16-розрядних процесорів Intel. Проте Кілделл не знав про те, що співробітник невеликої фірми Seattle Computer Тім Патерсон написав на основі CP/M власну операційну систему. «У мене не було часу виконувати доручену роботу так, як належить, - згадував автор нової ОС. - Довелося робити все швидко і поспішно».

За творінням Патерсона, офіційно пойменованим 86-QDOS, закріпилося прізвисько Quick and Dirty Operating System. Коли представники IBM повідомили Білу Ґейтсу про складні відносини IBM з Digital Research, у того вже було готове рішення. Microsoft викупила ліцензіію на QDOS у Seattle Computer, і після деякої модифікації під іменем MS DOS запропонувала її IBM. В IBM захвилювалися. Microsoft продає їм продукт, який виглядає, як CP/M, працює, як CP/M, але коштує в 4 рази дешевше, ніж CP/M. Не роздумуючи, IBM приймає заманливу пропозицію Ґейтса. Проте, підписуючи в поспіху угоду з Microsoft, менеджери IBM пропустили одну деталь, яка зіграла згодом для їх компанії фатальну роль - не обумовила ексклюзивних прав на MS-DOS. В підписаній угоді Ґейтс залишив за Microsoft право продавати операційну систему іншим виробникам комп'ютерів. Операційна система стала називатися Personal Computer DOS (PC-DOS), якщо поставлялася IBM, або MS-DOS, якщо продавалася кимось іншим. На момент виходу на ринок комп'ютерів IBM PC система PC-DOS була лише однією з трьох операційних систем, пропонованих для них IBM. Дві інші (вже згадувана CP/M та UCSD p-System, розроблена Softech) Чому ж було віддано перевагу PC-DOS? Основною її перевагою вважалася ціна. PC-DOS продавалася за 40 долл., CP/M - за 450 долл., а UCSD p-System - за 550 долл.

Озираючись назад, можна з упевненістю стверджувати, що як вчорашній успіх, так і сьогоднішнє положення справ визначили не стільки якість розробленої IBM обчислювальної системи, скільки оригінальне маркетингове рішення, прийняте компанією. Вона перша почала виробляти комп'ютери з відкритою архітектурою, зробивши специфікації доступними для інших виробників. Сотні компаній стали розробляти програмне забезпечення і апаратні компоненти, що базувалися на цих специфікаціях, і з'явилося поняття IBM PC Compatible.

Ну, а зараз пригадаємо (а для більшості читачів це буде, швидше за все, першим знайомством), яким же був IBM PC. Як CPU використовувалася мікросхема Intel 8088. Це був 16-розрядний процесор, що працював на частоті 4,77 MHz, з 8-розрядною шиною для підключення всієї периферії. Базова модель поставлялася з 16 KB RAM, з можливістю розширення до 256 KB, і з одним (опціонально - з двома) дисководами для гнучких дисків об'ємом 160 KB.

І ось настав 1981-й. Персональний комп'ютер, що з'явився на ринку як IBM Personal Computer (IBM PC) буквально викликав шок. Машина була швидша і інтелектуальніша, ніж будь-яка інша, що пропонувалася у той час. І ціна 2 тис. долл. виглядала привабливою: вона була трохи вищою, ніж набагато менш потужні комп'ютери інших виробників. На IBM почав працювати її авторитет в крупних корпораціях, урядових установах і школах. Для того, щоб охопити малий і середній бізнес, IBM проводить широку рекламну кампанію по національному телебаченню. Результати перевершили всі очікування, і компанія не встигає виконувати лавину замовлень, що хлинула. Вже до кінця 1982 р. IBM продала понад мільйон комп'ютерів. Персональний комп'ютер IBM мав такий приголомшуючий успіх, що газета «Time» в січні 1983 р. обрала його «Машиною року», де зазвичай перемагають автомобілі та літаки. Зрештою, якщо глянути на графік прибутків IBM, який був приведений в Розділі , то можна помітити досить значний стрибок, який дещо нагадує знаменитий стрибок 1965 року. Чому ж саме ПК вибився в лідери? Для першої моделі комп'ютер був дуже добре спроектований. За ним стояла потужність знаменитого виробника. Не останню роль зіграла і дотепна реклама. Але IBM не судилося повторити свій успіх 65- го року.

Рис. 1.17. IBM PC став сприйматися как синонім персонального комп'ютера

Вже на першому етапі розробники прагнули того, щоб виробництвом периферійних пристроїв для IBM PC зайнялися інші компанії. Тому комп'ютер спочатку створювався як відкрита, добре документована система. І дійсно, розробники дістали потрібні їм периферійні пристрої, але разом з тим вони отримали і дещо інше - а саме клони IBM PC. В 1982 році молода компанія Compaq випустила «переносимий комп'ютер» розміром зі швейну машинку. Він працював із тим же програмним забезпеченням і з тією ж платою розширення, що і IBM PC. Таке стало можливим не тільки завдяки відкритості архітектури і ПК, але і тому що IBM широко використовувала вже готові компоненти. Будь-яка компанія могла придбати процесор Intel і операційну систему Microsoft. Один головних розробників системи

IBM PC Бредлі вважає, що це рішення змінило вигляд всієї галузі: «Декларуючи перехід до відкритої системи, ви пропонуєте тим самим взяти участь в проекті і всім іншим. Такі компанії, як Lotus, отримали можливість зайнятися розробкою прикладних програм. В результаті були створені умови для ухвалення відкритих апаратних стандартів, і тисячі виробників виявилися залучені в сектор ПК».

До 1984 року вже декілька компаній позиціонувалось на ринку персональних комп'ютерів. В їх числі були як новачки, подібні Compaq (Columbia, Eagle, Leading Edge), так і достатньо відомі виробники (ITT, Tandy). Але лідером як і раніше залишалася IBM. В 1983 році Голубий гігант випустив комп'ютер PC/XT, що мав в своєму складі жорсткий диск. Конфігурація з диском місткістю 10 Мбайт коштувала 4995 долл. А в 1984 році корпорація залишила своїх переслідувачів далеко позаду, почавши продавати PC AT - перший комп'ютер на базі Intel 80286/6 Мгц. IBM поступилася пальмою першості лише в 1986 році, коли Compaq випустила перший ПК на основі 32-розрядного процесора 80386 (або, скорочено, просто 386). В цілому система Compaq була чимось більшим, ніж просто клон AT з поліпшеним процесором і швидким доступом до оперативної пам'яті. В 1987 році ринок відмовився прийняти широко розрекламований комп'ютер PS/2, який повинен був прийти на зміну AT. Навіть сам по собі термін «IBM-сумісний» поступово забувся. Відтоді всі комп'ютери сталі називатися просто ПК.

Персональні комп'ютери сталі настільки звичними, що вже складно уявити наше життя без них. Тим часом, перший IBM PC з'явився на світ 12 серпня 1981 року. За цей час «персоналка» кардинально змінилася, перетворилася з обчислювального пристрою з обмеженими можливостями в потужну мультимедійну машину, здатну замінити мейнфрейми минулого, які займали величезні площі , найскладніші ігрові приставки, майже всю домашню аудіовідеотехніку і багато чого іншого.

Тому навряд чи хтось стане заперечувати те, що випущений в 1981 році перший настільний комп'ютер, що отримав назву IBM Personal Computer або просто IBM PC, - один з найвидатніших винаходів другої половини XX століття, що серйозно змінило світ і спосіб життя людини. В IBM була зроблена ставка не тільки на оптові поставки, але і на роздрібний продаж комп'ютерів, внаслідок чого всього за перші п'ять років IBM продала три мільйони ПК при початковому прогнозі в 250 тисяч. До 1985 року IBM захопила 40 відсотків світового ринку мікрокомп'ютерів. Проте поступово на ринку виросли сильні гравці, що випускали клони IBM PC - ними стали такі компанії, як Compaq, Dell, Gateway, Kaypro і інші, що позиціонували свою продукцію спочатку як «IBM-сумісні комп'ютери». Так IBM з своєю відкритою архітектурою сама того не бажаючи, спочатку мало не монополізувала ринок настільних бізнес-систем, а потім - випустила його з своїх рук. Тим не менше, весь час, доки фірма IBM займалась виробництвом, її персональні комп'ютери були взірцевим брендом,

еталоном якості і надійності. Історію комп'ютерів власне IBM PC можна вважати закінченою з грудня 2004 року, коли Голубий гігант продав підрозділ персональних комп'ютерів, що став збитковим, найбільшому тайванському комп'ютерному концерну Lenovo. Хоча за договором з IBM, Lenovo має право користуватися брендом IBM протягом п'яти років з моменту підписання угоди, і працювати над машинами продовжують ті ж колишні співробітники IBM, але комп'ютери, що випускаються сьогодні моделі - це вже не легендарні IBM PC, а просто одні з численних нащадків першого настільного комп'ютера, що став дійсно персональним вже не для одного покоління користувачів. Проте, творці першого IBM Personal Computer 5150 заслуговують того, щоб ми сьогодні згадали про них і про їх дітище з повагою і щирою подякою.

ПК дуже швидко підтримали незалежні розробники програм. В першу чергу треба відзначити розробку текстових процесорів(компанія MicroPro з програмою WordStar, SSI з текстовим процесором WordPerfect і ряд інших). Важливість цієї подіїї важко переоцінити, бо текстові процесори саме на персональних комп’ютерах кардинально змінили життя людей, які мають справу з написанням текстів, а таких у світі мільярди. Єдина інновація, яка стоїть між текстовим процесором на персональному комп'ютері і олівця з папером – друкарська машинка. Але вона якісно поступається текстовим процесорам в плані обробки тексту. Відомо, що спроби застосувати ЕОМ для обробки текстів були задовго до появи персональних комп'ютерів. На початку 70-х років, коли велике розповсюдження отримали міні-ЕОМ, цілі компанії, наприклад Wang, спеціалізувалися на виробництві комп'ютерів і програм, спеціально призначених для ведення діловодства в крупних офісах. Проте це були вузькопрофесійні системи, не призначені для широкої публіки. Масове розповсюдження системи обробки текстів отримали тільки після появи персональних комп'ютерів.

Пітер Нортон представив першу версію пакету Norton Utilities, який дозволяв відновити випадково видалені файли. А Ендрю Флюгельман почав безкоштовно поширювати PC-Talk -програму, що помітно спрощувала для користувачів модемів процедуру з'єднання зі службами, що цікавлять їх , такими як CompuServe і Source.

Розробка М.Капуром в 1983 році електронних таблиць 1-2-3 ще більше закріпила репутацію IBM PC як машини ділового призначення. Це теж була видатна подія, бо включала в себе новий спосіб електронної обробки даних візуально-табличним способом. Справедливості ради, треба відзначити, що перший процесор електронних таблиць з'явився дещо раніше, а саме у 1979 році. Ідея такого табличного процесора була запропонована студентом Гарварду Даном Брікліном, якому довелося вирішувати складні фінансові задачі, що вимагали великої кількості обчислень. З своїм другом Бобом Франкстоном вони написали для комп'ютера Apple II відповідну програму, яку назвали VisiCalc. Програма виявилася настільки зручною для фінансових обчислень, що багато компаній стали купувати Apple II

для своїх співробітників тільки тому, що там був передвстановлено VisiCalc. Проте успіх Lotus 1-2-3 був набагато масштабнішим. Розробник Lotus 1-2-3 М.Капор дуже грамотно використав переваги 16-розрядного процесора ПК і запропонував цілий ряд революційних рішень. Система Lotus 1-2-3 для IBM PC була зроблена так, що в обхід DOS працювала з відеопам'яттю, і це давало їй великі переваги по швидкодії перед конкурентами. Крім того, Lotus 1-2-3 мала вбудований текстовий редактор u1088 і засоби ділової графіки (цим пояснюється її незвичайна назва - три в одному), а також інтерактивну help-підтримку і інші зручності (зокрема, екранне меню). На рекламу системи Капор витратив мільйон доларів, проте ці витрати окупилися дуже скоро. За рік було продано 107 000 екземплярів по 495 доларів кожний, а в наступний рік доходи фірми склали 156 мільйонів доларів.

Lotus 1-2-3 швидко стала лідером ринку, а її формат - популярним форматом електронних таблиць для обміну, чим і визначили подальший успіх не тільки табличного процесора Lotus 1-2-3 як такого, але і цілого напряму електронної обробки даних, без якої не уявляють свого життя мільйони економістів. Ідея електронних таблиць була геніальна в своїй простоті, вона провела справжній переворот у свідомості рядових користувачів, які отримали можливість, не вивчаючи Фортран і COBOL, виконувати достатньо складні обчислення(згадаємо, що колись Грейс Хоппер справді сподівалась, що бухгалтери будуть програмувати на їх COBOL'і, але ця мрія так і залишилося утопією). Слідом за VisiCalc на ринок були викинуті десятки аналогічних пакетів - SuperCalc фірми Computer Associates, QuattroPro фірми Borland і т.п. За функціональними можливостями всі вони були приблизно однакові, відрізняючись лише деталями інтерфейсу і ціною.

В 1984 році в змагання з іншими виробниками вступила Microsoft з табличним процесором MultiPlan для IBM PC. Згодом він отримав істотно допрацьований графічний інтерфейс і перейменовано в Excel. В 1990 році вийшла версія Excel для Windows. Оскільки Lotus Development не зуміла вчасно створити конкурентноздатну версію свого продукту для Windows, Excel протягом декількох подальших років відвоював ринок у Lotus 1-2-3. Завдяки широкому набору стандартних функцій і вбудованій мові програмування VBA - Visual Basic for Applications, Excel може використовуватися не тільки для найпростіших, але і для складних статистичних і оптимізаційних розрахунків. В даний час він є лідером в секторі електронних таблиць у світі, а на українському ринку - монополістом. Lotus Development довелося залишити цей сегмент ринку. Природньо, що виникло питання про розробку СУБД на базі персональних обчислень, та про це мова в наступних розділах.

Контрольні запитання

1. Назвіть причини появи рахунково-перфораційних комплексів у кінці XIX століття.
2. В чому полягає суть гіпотези Бенігера?
3. Чим відрізнялася UNIVAC від своїх попередників?

4. Завдяки чому IBM виграла гонку на ринку ЕОМ у Remington Rand, незважаючи на запізнілий старт?
5. Сформулюйте досягнення у розвитку електронно-обчислювальної техніки у бувшому СРСР та в Україні зокрема.
6. Які якісні досягнення у програмному забезпеченні відбулися в 50-60 роки XIX століття?
7. Які фактори сприяли величезному успіху IBM у 60-ті роки XX ст.?
8. Поясніть суть терміну спулінг.
9. Які інновації початку 70-х років заклали фундамент революції персональних обчислень.
10. Чому IBM спізнилася на ринок персональних обчислень?
11. В чому суть ідеї мультипрограмування?
12. Як коротко можна виразити суть класичної схеми персональних обчислень на базі IBM PC?
13. В чому полягала роль Microsoft в процесі революції персональних обчислень?
14. Які якісно нові програмні продукти для електронної обробки даних з'явилися на базі технології персональних обчислень?

Вже першим розробниками інформаційних систем для підприємств було ясна специфіка обчислювальних задач, пов'язаних з електронною обробкою даних:

- великий обсяг даних досить складної структури;
- великий обсяг обчислень, але досить простих алгоритмічно;
- проблема інтерфейсу між користувачами системи та обчислювальним комплексом.

Відповідно треба було сформувати адекватну методологію побудови електронної обробки даних, на базі якої можна було б створювати надійні комп'ютеризовані системи управління. В першу чергу треба було сформулювати єдиний підхід до трактування даних, які підлягали електронній обробці. Зрештою такий єдиний підхід був виражений поняттям моделі даних.

2.1 Поняття моделі даних

2.1.1 Типи структур даних

Структуризація даних базується на використанні концепцій «агрегації» і «узагальнення». Перший варіант структуризації даних був запропонований Асоціацією по мовах обробки даних (Conference on Data Systems Languages, CODASYL) (Рис.2.1.).

Рис. 2.1. Композиція структур даних за версією CODASYL

Елемент даних – найменша поіменована одиниця даних, до якої СУБД може звертатися безпосередньо і за допомогою якої виконується побудова всієї решти структур. Для кожного елемента даних повинен бути визначений його тип.

Агрегат даних – поіменована сукупність елементів даних у середині запису, який можна розглядати як єдине ціле. Агрегат може бути простим (що включає тільки елементи даних, рис.2.2,а) і складовим (що включає разом з елементами даних і інші агрегати, рис.2.2,б).

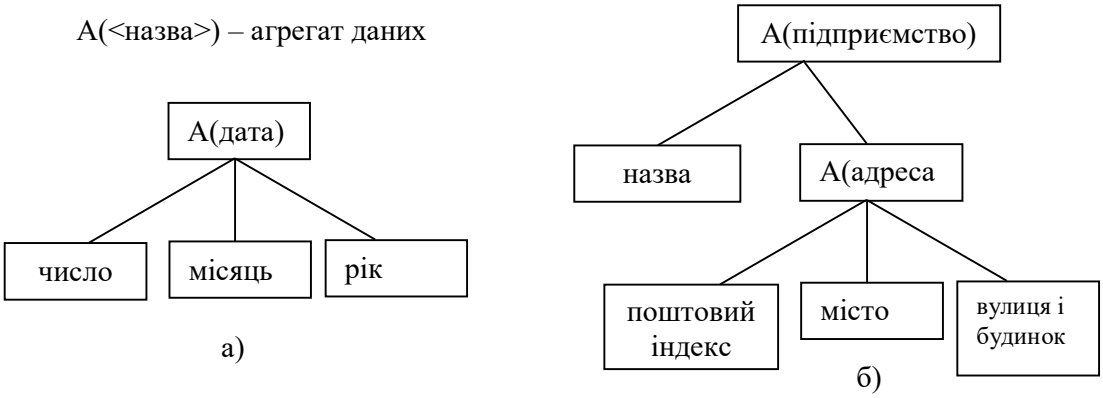


Рис.2.2. Приклади агрегатів: а) простий і б) складовий агрегат

Запис – поіменована сукупність елементів даних або елементів даних і агрегатів. Запис – це агрегат, що не входить до складу жодного іншого агрегату; вона може мати складну ієрархічну структуру, оскільки допускається багатократне застосування агрегації. Розрізняють тип запису (її структуру) і екземпляр запису, тобто запис з конкретними значеннями елементів даних. Один запис описує властивості одного об'єкта ПО (екземпляра).

Серед елементів даних (полів) виділяються одне або декілька ключових полів. Значення ключових полів дозволяють класифікувати об'єкт, до якого відноситься конкретний запис. Ключі з унікальними значеннями називаються потенційними. Кожний ключ може бути агрегатом даних. Один з ключів є первинним, інші – вторинними. Первинний ключ ідентифікує екземпляр запису і його значення повинне бути унікальним в межах записів одного типу. Іноді термін «запис» замінюють терміном «група».

Набір (або групове відношення) – поіменована сукупність записів, які створюють дворівневу ієрархічну структуру. Кожний тип набору є відношенням (зв'язком) між двома або декількома типами записів. Для кожного типу набору один тип запису може бути оголошений власником набору, решта типів запису оголошується членами набору. Кожний екземпляр набору повинен містити тільки один екземпляр запису типу власника і стільки екземплярів записів типу членів набору, скільки їх пов'язано з власником. Для групового відношення також розрізняють тип і екземпляр.

Файл – поіменована сукупність екземплярів записів одного типу. Використовується для представлення однорідного набору сутностей.

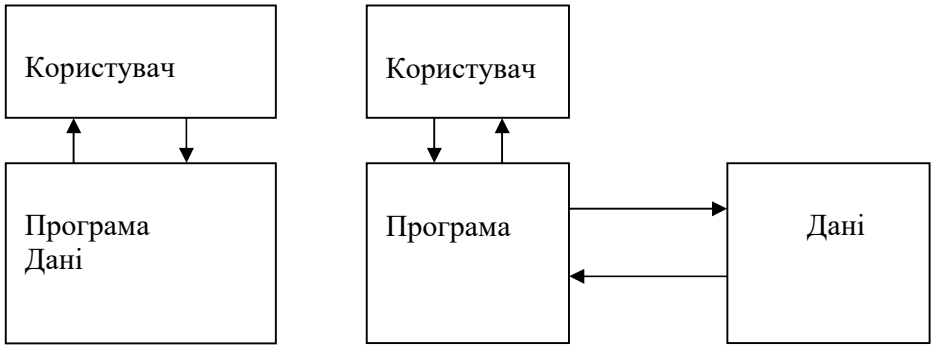
Набір файлів – поіменована сукупність файлів, що обробляються в системі.

У цей період виникло ряд понять, без яких не можна було б оперувати новою сферою професійної діяльності. В першу чергу це стосується поняття бази даних - централізоване сховище даних, що забезпечує зберігання, доступ, первинну обробку і пошук інформації. Це мав бути електронний еквівалент картотеки, якою маніпулювали вручну, або

за допомогою табуляторів. Це потребувало відповідних спеціалістів, яких треба було готувати. Довгий час існувала навіть така спеціальність як «механізована обробка економічної інформації».

2.1.2 Проблема програмного маніпулювання структурованими даними

Якісно цю ситуацію змінило створення спеціального програмного забезпечення для маніпулювання базою даних – систем управління базою даних(СУБД). В свою чергу досвід розробки та експлуатації привів до формулювання принципів обробки даних, на яких повинна базуватися саме електронна технологія обробки даних. В принципі, на той час технологія програмування бурхливо розвивалася і була розроблена маса алгоритмічних мов, серед них вже згадуваний COBOL. Розробники програмних систем дуже добре усвідомлювали принцип незалежності програми від даних, суть якого фактично зводилась до того, що правильно написана програма не повинна змінюватись при зміні вхідних даних. На Рис.2.3 а) зображена схема використання програми, що залежить від даних та на Рис.2.3 б) використання програми, що не залежить від даних. Більшість алгоритмічних мов, створених на той час, зокрема і COBOL, цілком задоволняли принципу незалежності програми від даних. Але практика електронної обробки даних, базованих на роботу з базами даних, показала, що забезпечення тільки цього принципу недостатнє. Дуже скоро спеціалістам по обробці даних, які працювали над розробкою та впровадженням інформаційних систем стало зрозумілим, що для успішної розробки баз даних в сфері бізнесу треба врахувати ще одну суттєву особливість – високу динамічність вхідних даних.



а) б)

Рис.2.3. Схеми використання даних

На відміну від класичної схеми обробки даних, яка склалася ще за часів Голлерита, коли довго і скрупульозно збирали дані, потім їх фактично складували в стосах перфокарт і тільки потім починалась обробка, бази даних для бізнесу мали працювати в такому часовому режимі, який би забезпечував можливість прийняття рішень на основі оброблених даних. А такі рішення треба приймати щоденно, а може й щогодинно. Ще одним вузьким місцем було забезпечення узгодженої роботи великій кількості менеджерів і спеціалістів, кожен з яких мав керуватися своїми власними функціональними обов'язками, при цьому опираючись на дані з однієї бази даних.

Ці особливості обробки даних породжують нові проблеми, які потребували вирішення, в першу чергу – це проблема цілісності даних, тобто, будь які звернення до бази даних не повинні приводити до їх втрат. З іншого боку небажано тримати надлишок даних, бо це приводить до невиправданого збільшення їх обсягу на зовнішніх носіях і збільшення часу на їх обробку. Великий досвід реальної розробки інформаційних систем в економіці, помножений на напружені теоретичні дослідження в 70-х роках зрештою привів до фундаментальних, загальновизнаних положень сучасної електронної обробки даних, які набули характеру міжнародних стандартів розробки інформаційних систем.

Рис.2.4. Схема обробки даних з використанням СУБД.

2.2. Стандарт ANSI/SPARC

В 1975 р. була офіційно визнана відмінність між логічним і фізичним представленням даних. Комітет ANSI/SPARC(Standards Planning And Requiments Committee) запропонував узагальнену структуру БД, яка отримала назву трьохрівневої, тобто існування трьох рівнів абстракції, на яких можна розглядати БД: 1) концептуальний, 2) зовнішній і 3) внутрішній рівні.

2.2 .1. Трьохрівнева архітектура ANSI/SPARC

Трьохрівнева архітектура – стандартна структура БД, що складається з концептуального, зовнішнього і внутрішнього рівнів.

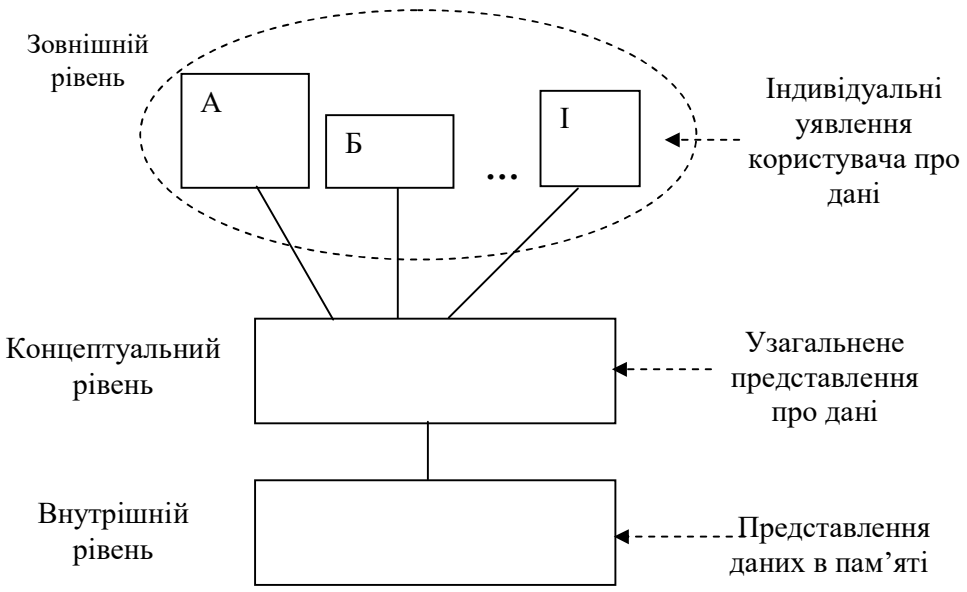


Рис. 2.5. Трьохрівнева архітектура

в цій архітектурі враховується, що з базою даних може працювати декілька користувачів (в нашому випадку А, В, І) з різними інформаційними потребами. При цьому виходять з того, що будь-який *банк даних* повинен підтримувати різноманітні *уявлення* користувачів про *предметну область*. Предметна область– частина реального світу, що представляє інтерес для даного дослідження (використання), для інформаційних систем в економіці –це економічні об'єкти. Докладніше кожний рівень архітектури виглядит таким чином.

Зовнішній рівень. Окремого користувача цікавить, як правило, тільки певна частина всієї бази даних. Уявлення окремого користувача про предметну область називається *зовнішнім представленням*. Таким чином, зовнішній рівень складається із зовнішніх представлень (в англійській термінології називаються Views)

Зовнішні представлення - це зміст бази даних, яким його бачить певний користувач.

Воно складається з множини типів *зовнішніх записів*.

Під *записом* розуміється група взаємозв'язаних елементів даних, що розглядаються як єдине ціле. Наприклад, користувач з бухгалтерії може розглядати базу даних як набір записів з інформацією про платежі і перерахування коштів, і може нічого не знати про записи з інформацією з інформацією про службовців, які ведуться у відділі кадрів.

Для використання комп'ютера при обробці інформації про предметну область цю інформацію потрібно представляти в спеціальному вигляді, строго, формалізовано. Формалізація - невід'ємна частина розробки будь-якої програмної системи. Спосіб формального опису баз даних полягає у використанні схем.

Схема -опис логічної структури БД. Схеми використовуються для строгого, формального опису кожного рівня архітектури. На зовнішньому рівні кожне представлення користувача описується за допомогою *зовнішньої схеми* (іноді її називають *підсхемою*). Зовнішня схема складається з визначень кожного типу записів на зовнішньому представленні. Визначення даються на мові визначення даних (DDL) з призначеного для користувача підмови даних.

2.2.2. Концептуальний рівень

Концептуальне представлення формується на основі інтеграції зовнішніх представлень користувачів. Концептуальне представлення – представлення всього вмісту БД. Як правило, концептуальне представлення істотно відрізняється від зовнішніх представлень окремих користувачів (оскільки підсумовує їх розрізнені представлення в одне узагальнене), і складається з множин типів концептуальних записів. Концептуальне представлення визначається за допомогою концептуальної схеми. Концептуальна схема - опис повної загальної логічної структури бази даних. Концептуальна схема використовує (в загальному випадку) іншу мову опису даних. Визначення концептуальної мови повинне відноситися тільки до змісту даних, не торкаючись фізичних подробиць бази.

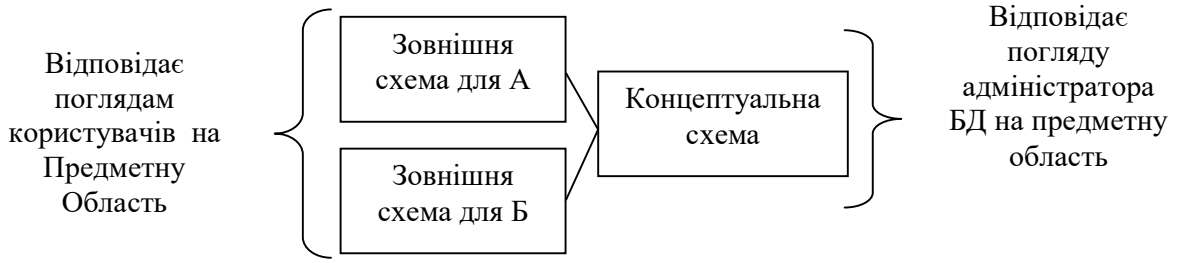
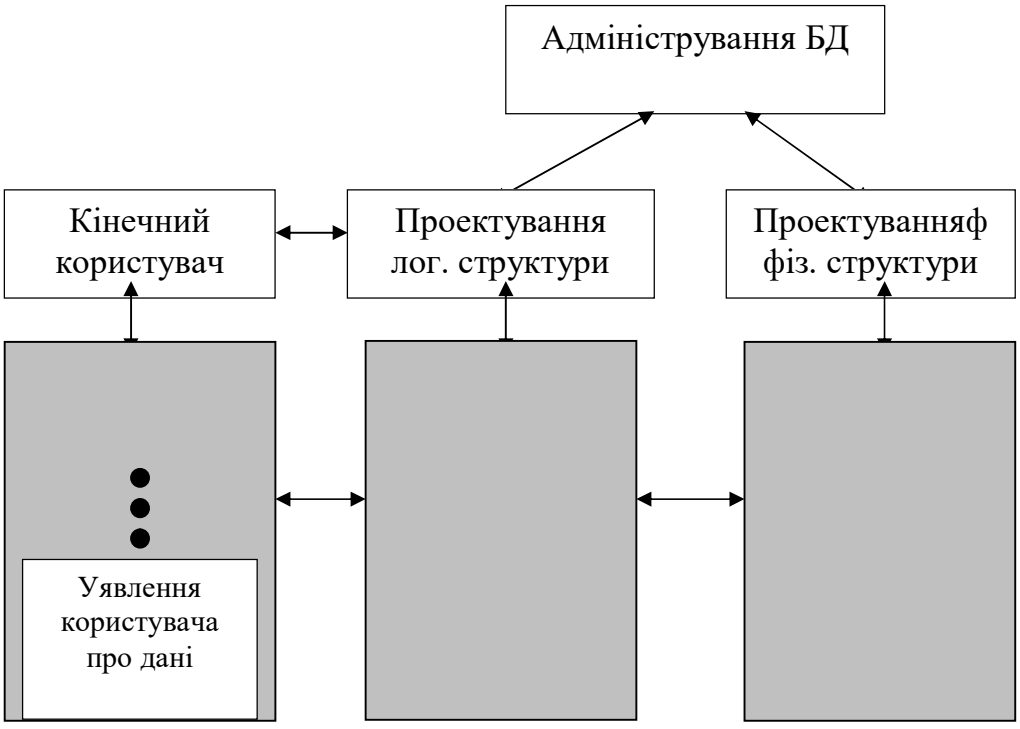


Рис.2.6. Відповідність між поглядами користувачів і поглядом адміністратора БД В концептуальній схемі не розглядаються способи організації зберігання або методи доступу до даних, що зберігаються. Визначення в концептуальній схемі, крім опису типів записів, можуть включати такі засоби, як безпека, правила підтримки цілісності. записи концептуального рівня не зобов'язані співпадати із записами зовнішніх рівнів.

2.2.3. Внутрішній рівень

Внутрішнє представлення БД – представлення структури зберігання записів складається з множини типів внутрішніх записів (інша назва – записів, що зберігаються). Внутрішнє представлення так само не пов'язано з фізичним рівнем, тобто не розглядаються фізичні записи, фізичні пристрої зберігання (наприклад, циліндри і доріжки), способи видалення та доступу до даних. Внутрішнє представлення описується за допомогою внутрішньої схеми, яка визначає не тільки різні типи записів, що зберігаються , але і існуючі індекси, способи представлення полів, що зберігаються, і т.д. Внутрішня схема використовує внутрішню мову визначення даних. Записи внутрішнього рівня частіше за все не співпадають із записами зовнішніх і концептуального рівнів.

Деталізована архітектура системи баз даних показана на Рис. 2.7.



зовнішній рівень	концептуальний рівень	внутрішній рівень

Рис.2.7. Деталізованная архітектура системи баз даних

На рівні 1 виконується *концептуальне проектування* БД. Воно включає аналіз інформаційних потреб користувачів і визначення потрібних їм даних. Результатом є концептуальна схема, єдиний логічний опис всіх елементів даних і відношень між ними.

На рівні 2 складають *призначені для користувача представлення даних* БД. Кожна група, що підлягає визначенню, одержує своє представлення даних в БД. Кожне представлення даних дає орієнтоване на користувача опис елементів даних і відношень між ними. Його можна прямо вивести з концептуальної схеми.

На рівні 3 забезпечує *фізичний погляд* на БД: дисководи, фізичні адреси, індекси, вказівники і т.д. За цей рівень відповідають проектувальники фізичної БД, які вирішують такі питання як:

- різновид фізичних пристроїв, що зберігатимуть дані;
- методи доступу, що використовуватимуться для добування і зберігання даних;
- заходи, які слід прийняти для підтримки або підвищення швидкодії СУБД.

Жоден користувач не має прямого доступу до цього рівня. Для того, щоб представити дані користувачам на 1) або 2) рівнях, система повинна уміти перетворювати адреси і вказівники у відповідні логічні імена і відносини. Таке перетворення може відбуватися і у зворотному напрямі – з логічного рівня на фізичний. Ціна такого перетворення - велика системна затримка. Первагою ж є незалежність логічного і фізичного представлення даних. Отже, подібна архітектура забезпечує незалежність даних.

Інфологічна модель ПО – опис ПО, без орієнтації на використання надалі програм і технічних засобів.

Даталогічна модель – модель даних логічного рівня, що підтримується засобами СУБД. Вона є відображенням логічних зв'язків між елементами даних безвідносно до їх змісту і середовища зберігання, але з урахуванням обмеження конкретної СУБД. Фізична модель використовується для прив'язки даталогічної моделі до середовища зберігання, визначає тип запам’ятовуючого пристрою, спосіб розташування даних в запам’ятовуючому пристрої, спосіб фізичної реалізації логічних відношень між елементами даних, враховує обмеження СУБД і ОС.

2.3 Моделі баз даних

Стандарт ANSI/SPARC – це фактично тільки рамкові правила, яким має відповідати правильно розроблена система електронної обробки даних. Інструментом її реалізації мають

бути апаратні та програмні засоби. Тому кожний розробник, беручи за основу наявні апаратні засоби, або, як зараз прийнято виражатись платформи, повинен використати адекватні програмні засоби, щоб задовольнити вимоги замовника, які не можуть ніяк бути менші ніж вимоги стандарту. Домінучою апаратною платформою завдяки зусиллям IBM та її конкурентів з 70-х років стала модель з централізованою архітектурою. При використанні цієї технології база даних, СУБД і прикладна програма розташовуються на одному комп'ютері - мейнфреймі (зараз таку архітектуру можна реалізувати і на персональному комп'ютері). Для такого способу організації не потрібна підтримка мережі і все зводиться до автономної роботи. Робота побудована таким чином:

- База даних у вигляді набору файлів знаходиться на жорсткому диску комп'ютера (або магнітних стрічках).
- На тому ж комп'ютері встановлена СУБД і прикладна програма для роботи з БД.
- Користувач запускає програму. Використовуючи призначений для користувача інтерфейс, що надається прикладною програмою, він ініціює звернення до БД на вибірку/оновлення інформації.
- Всі звернення до БД йдуть через СУБД, яка інкапсулює усередині себе всі відомості про фізичну структуру БД.
- СУБД ініціює звернення до даних, забезпечуючи виконання запитів користувача (здійснюючи необхідні операції над даними).
- Результат СУБД повертає в програму.
- Програма, використовуючи призначений для користувача інтерфейс, відображає результат виконання запитів.

Саме тоді почала утверджуватись думка про те, що розробка електронної обробки даних буде успішною тільки при використанні певної моделі баз даних. Можна стверджувати дещо більше – саме винахід моделей баз даних зрештою і привів до появи стандарту ANSI/SPARC.

2.3.1 Загальні поняття про моделі баз даних

Пусть опрокинет статуи война.
Мятеж развеет каменщиков труд,
Но врезанные в память письмена
Бегущие столетья не сотрут!
(В.Шекспир)

Можна по різному характеризувати поняття моделі даних. З одного боку, модель даних - це спосіб структуризації даних, причому дані розглядаються як певна абстракція у відриві від предметної області. З іншого, боку модель даних це інструмент представлення концептуальної моделі предметної області і динаміки її зміни у вигляді бази даних.

Враховуючи обидва ці фактори, визначимо основні структури моделей даних, що використовуються для представлення концептуальної моделі предметної області (сутностості, атрибутів, зв'язків).

Як вже зазначалось Асоціацією по мовах обробки даних (Conference on Data Systems Languages, CODASYL) була запропонована наступна структуризація даних.

Група – це поіменована сукупність елементів даних або елементів і інших груп.

Приклад:

Замовлення на закупівлю										
№ замовлення	Дата замовлення			Партія товару						
	число	місяць	рік	Шифр товару	Кількість	Ціна		Шифр товару	Кількість	Ціна

Екземпляр P1 Екземпляр Pn

Рис.2.8. Приклад структуризації економічних даних

Найважливішим поняттям концептуальної моделі є поняття зв'язку між сутностями (наборами сутностей). В моделях даних, відповідне поняття відображається поняттям «групове відношення». Групове відношення – поіменоване бінарне відношення, визначене на двох множинах екземплярів даних груп. По характеру бінарних зв'язків розрізняють групові відносини вигляду 1:1, 1:M, M:1, M:N. Пари чисел називають ступенем групового відношення. В груповому відношенні один член групи призначається власником відношення, а інший – членом. Виходячи з цих понять базу даних можна визначити як поіменовану сукупність екземплярів груп і групових відношень. Для представлення групового відношення використовується дві форми:

а) Графова. Групи зображаються вершинами графа. Зв'язки між групами – дугами, направленими від групи власника до групи члену з вказівкою імені відношення і коефіцієнта. По типу графів розрізняють:

- ієрархічну модель (граф без циклів – дерево);
- мережеву модель (орієнтований граф загального вигляду).

б) Теоретико-множинна(Таблична). Зв'язок між групами зображається таблицею, колонки якої представляють ключі відповідних груп. Для формального опису таблиці використовується математичне (теоретико-множинне) поняття відношення. Відповідна модель даних називається реляційною моделлю.

2.3.2 Ієрархічні системи

Типовим представником ієрархічної СУБД (найбільш відомим і поширеним) є Information Management System (IMS) фірми IBM. Перша версія з'явилася в 1968 р. Але й до цього часу підтримується немало працюючих баз даних, що з одного боку не може не викликати повагу до якості вирішення задачі, з іншого - створює істотні проблеми з переходом як на нову технологію БД, так і на нову апаратну платформу. Ієрархічна БД складається з впорядкованого набору дерев; більш точно, з впорядкованого набору декількох екземплярів одного типу дерева, тобто лісу з дерев. Тип дерева складається з одного «кореневого типу» запису і впорядкованого набору з нуля або більш типів піддерев (кожне з яких є деяким типом дерева). Тип дерева в цілому є ієрархічно організованим набором типів запису. Приклад типу дерева (схеми ієрархічної БД): Для груп ВІДДІЛ, НАЧАЛЬНИК, СПІВРОБІТНИК в будь якій організації може бути промодельовано наступним чином(Рис.2.9.)

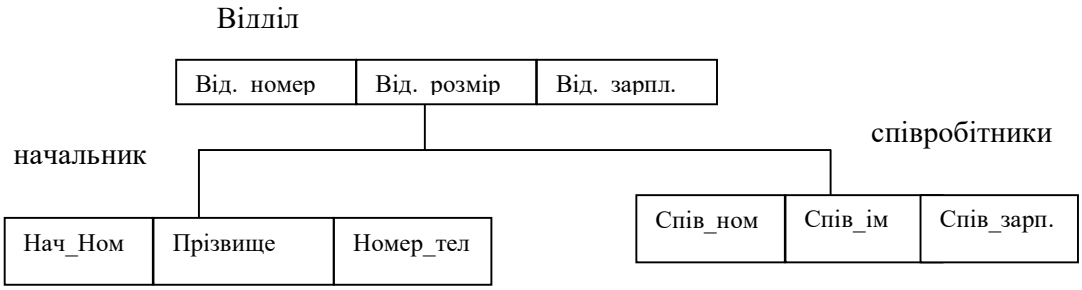


Рис.2.9. Структура зв’язків між записами в ієрархічній моделі

Тут Відділ є предком для Начальник і Співробітники, а Начальник і Співробітники - нащадки ВІДДІЛ. Між типами запису підтримуються зв'язки. Екземпляр дерева в базі даних з такою схемою могла б виглядати таким чином (Рис.2.10.):

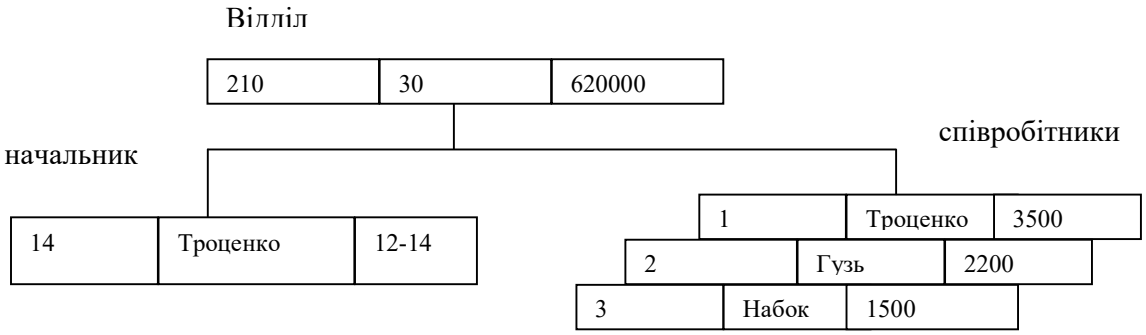


Рис.2.10. Екземпляри записів в для БД ієрархічного типу

Всі екземпляри даного типу нащадка із загальним екземпляром типу предка називаються близнятами. Для БД визначений повний порядок обходу - згори-донизу, зліва-направо. В СУБД IMS використовувалася оригінальна і нестандартна термінологія: «сегмент» замість «запис», а під «записом БД» розумілося все дерево сегментів. Маніпулювання ієрархічно організованими даними реалізується операторами маніпулювання, зокрема:

- знайти вказане дерево БД (наприклад, відділ 310);
- перейти від одного дерева до іншого;
- перейти від одного запису до іншого в середині дерева (наприклад, від відділу - до першого співробітника);
- перейти від одного запису до іншого за встановленим порядком обходу ієрархії;
- вставити новий запис у вказану позицію;
- видалити поточний запис.

Для автоматичної підтримки цілісності посилань між предками і нащадками має бути виконуватись основне правило: жоден нащадок не може існувати без свого батька. Зауважимо, що аналогічна підтримка цілісності за посиланнями між записами, що не входять в одну ієрархію, не підтримується . В ієрархічних системах підтримувалася певна форма представлень БД на основі обмеження ієрархії. Наприклад:

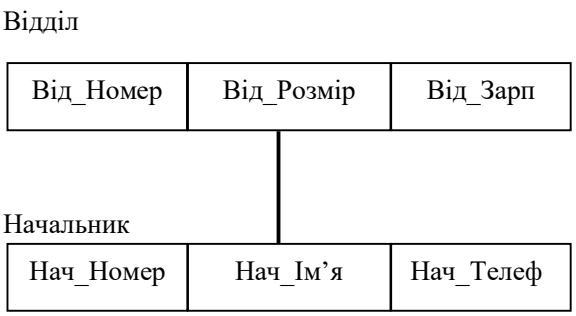


Рис.2.11. Приклад обмеження ієрархій

Найістотнішим недоліком ієрархічної моделі, на наш погляд, є «жорсткість» отримуваної концептуальної схеми. Зв'язки закріплені в записах у вигляді вказівників. При появі нових аспектів використання цих же даних може виникнути необхідність встановлення нових зв'язків між даними. Це вимагає введення в записи нових вказівників, тобто зміни структури БД, і, відповідно переформовування всієї бази даних.

2.3.3 Мережева модель даних

Мережевий підхід до організації даних є розширенням ієрархічного. В ієрархічних структурах запис-нащадок повинен мати одного і тільки одного предка; в мережевій структурі даних нащадок може мати будь-яке число предків. Типова мережева модель даних була запропонована робочою групою по базах даних (Data Base Task Group –DTBG) системного комітету CODASYL (Conference Data System Languages), основними функціями якого були аналіз відомих фірмових систем обробки управлінських даних з єдиних позицій і в єдиній термінології, узагальнення досвіду організації таких систем і розробка рекомендацій по створенню відповідних систем.

Представлення зв'язків 1:1, 1:M, N:1 є окремим випадком зв'язку типу M:N і здійснюється аналогічно розглянутому вище. Зауважимо, що група може бути членом більш ніж одного групового відношення. В цьому випадку вводиться декілька додаткових груп-вказівників, а в групі-власнику відношень вводиться декілька полів-вказівників на додаткові групи Тоді множина записів (груп) і зв'язків між ними утворюють якусь мережну структуру (орієнтований граф загального вигляду). Вершинами графа є групи; дугами графа, направленими від власника до члена групового відношення, є зв'язки між групами. Мережева модель даних підтримує всі необхідні операції над даними, реалізовані як дії із списковими структурами.

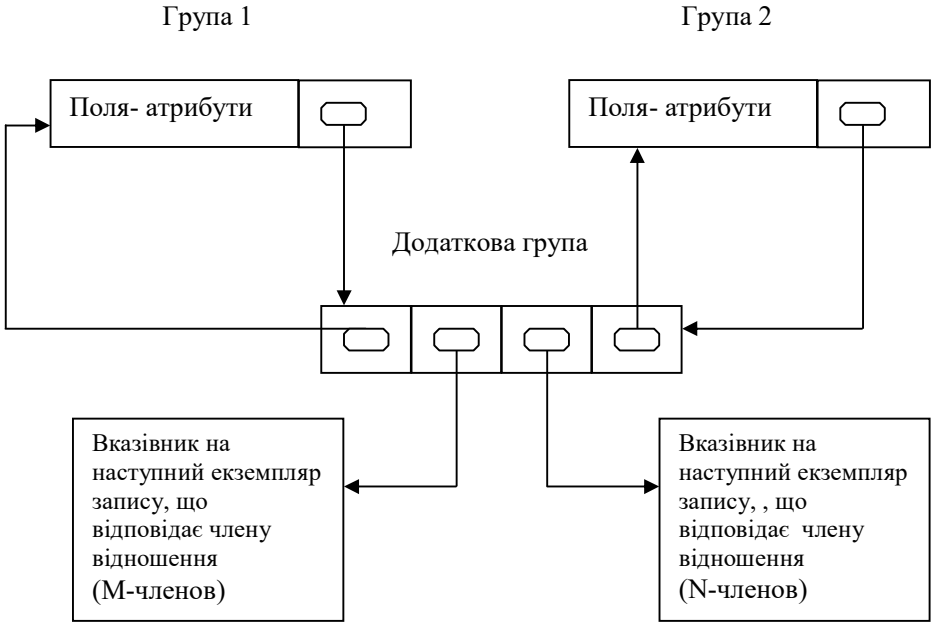


Рис.2.12. Представлення зв'язків типу M:N в мережевій моделі

Мережева БД складається з набору записів і набору зв'язків між цими записами, а якщо говорити більш точно, з набору екземплярів кожного типу із визначеного в схемі БД набору типів записів і набору екземплярів кожного типу із визначеного набору типів зв'язку. Тип зв'язку визначається для двох типів записів: предка і нащадка. Екземпляр типу зв'язку складається з одного екземпляра типу запису предка і впорядкованого набору екземплярів типу запису нащадка. Для даного типу зв'язку L з типом запису предка P і типом запису нащадка K повинні виконуватися наступні дві умови:

- кожний екземпляр типу P є предком тільки одному екземпляру L;
- кожний екземпляр K є нащадком не більш, ніж одного екземпляру L.

На формування типів зв'язку не накладаються особливі обмеження; можливі, наприклад, наступні ситуації:

- тип запису нащадка в одному типі зв'язку L1 може бути типом запису предка в іншому типі зв'язку L2 (як в ієрархії).
- даний тип запису P може бути типом запису предка в будь-якій кількості типів зв'язку.
- даний тип запису P може бути типом запису нащадка в будь-якому числі типів зв'язку.
- може існувати будь-яке число типів зв'язку з одним і тим же типом запису предка і одним і тим же типом запису нащадка; і якщо L1 і L2 - два типи зв'язку з одним і тим же типом запису предка P і одним і тим же типом запису нащадка K, то правила, по яких утворюється спорідненість, в різних зв'язках можуть розрізнятися.
- типи запису X і Y можуть бути предком і нащадком в одному зв'язку і нащадком і предком - в іншій.
- предок і нащадок можуть бути одного типу запису.

Простий приклад мережевої схеми БД:

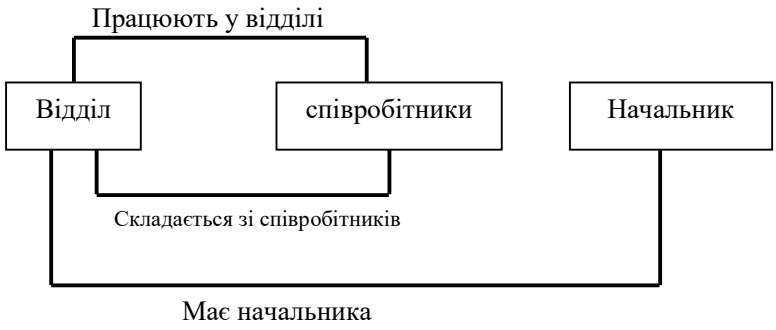


Рис.2.13. Мережева схема даних

Базовий набір операцій в мережевій моделі дещо схожий на на відповідний набір в ієрархічній моделі, але дещо більший за обсягом :

- знайти конкретний запис в наборі однотипних записів (інженера Набока);
- перейти від предка до першого нащадка по деякому зв'язку (до першого співробітника відділу 310);
- перейти до наступного нащадка в деякому зв'язку (від Троценка до Гузя);
- перейти від нащадка до предка по деякому зв'язку (знайти відділ Троценка);
- створити новий запис;
- знищити запис;
- модифікувати запис;
- включити в зв'язок;
- виключити із зв'язку;
- переставити в інший зв'язок і т.д.

Схоже на те, що мережева модель даних є, ймовірно, найбільш загальною моделлю даних по можливостях представлення концептуальної моделі. По суті, будь-яка діаграма сутностей без змін представляється засобами мережної моделі. До недоліків мережної моделі зазвичай відносять складність одержання на її основі концептуальної схеми і велику трудомісткість розуміння відповідної схеми зовнішнім користувачем, крім того їй притаманна така ж «жорскість» як і в ієрархічній моделі. Тим не менше довгий час вона домінувала.

СУБД, що підтримують мережну модель, широко використовувалися на обчислювальних системах серії IBM 360/370 (ЄС ЕОМ). Як приклади таких систем можна вказати системи IDMS, UNIBAD(БАНК), CETOP. На персональних комп'ютерах мережні СУБД не отримали широкого розповсюдження. Таким рідкісним прикладом мережної СУБД для персонального комп'ютера є *db_VISTA III*. Відзначимо, що система *db_VISTA* реалізована мовою С і тому є переносимою. Система може експлуатуватися на ПЕОМ типу IBM PC, SUN, Macintosh.

Управління асоціативним доступом і обробкою, орієнтованою на набори, було настільки поширено, що в співтоваристві COBOL виділилася група Data Base Task Group (DBTG) для визначення стандартного способу специфікації таких даних і доступу до них. Чарльз Бахман в GE (General Electric) побудував прототип системи навігації даних. За керівництво роботами групи DBTG, яка розробила стандартну мову визначення даних і маніпулювання даними, Ч.Бахман у 1973 р. був відзначений премією Тьюринга. Саме в асоціації баз даних CODASYL викристалізувалася концепція схем і незалежності даних. Було узаконене приховування фізичних деталей розташування записів. Програми мають «бачити» тільки логічну організацію записів і зв'язків і не змінюватись при зміні запособів зберігання даних. Записи, поля і зв'язки, які не використовуються програмою, було слід

приховати - як з міркувань безпеки, так і для того, щоб ізолювати програму від неминучих змін схеми бази даних. В цих ранніх базах даних підтримувалися три види схем даних:

1) логічна схема, яка визначає глобальний логічний проект записів бази даних і зв'язків між записами;

2) фізична схема, що описує фізичне розміщення записів бази даних на пристроях пам'яті і у файлах, а також індекси, потрібні для підтримки логічних зв'язків;

3) підсхема, що надається кожній прикладній програмі і розкриває тільки частину логічної схеми, яку використовує програма.

Механізм логічних, фізичних і підсхем забезпечував незалежність даних. І дійсно, багато програм, написаних в ту епоху, все ще працюють сьогодні з використанням тієї ж самої підсхеми, з якою все починалося, хоча логічна і фізична схеми абсолютно змінилися.

Додатково до цього, операційні системи, на базі яких мала бути реалізована електронна обробка даних, повинні забезпечувати одночасне виконання багатьох транзакцій над базою даних, що спільно використовується багатьма термінальними користувачами. До цього підхід, заснований на однопрограмному режимі і періодичній зміні основного файлу, усував проблеми паралельного доступу і відновлення. Ранні операційні системи проклали шлях поняттю транзакцій, які блокували тільки ті записи, до яких проводився доступ. Блокування дозволяють конкуруючим транзакціям діставати доступ до різних записів. Системи також підтримували журнал записів, що змінювалися кожною транзакцією. При збої транзакції журнал використовувався для усунення в базі даних дій цієї транзакції. Журнал транзакцій використовувався також для відновлення бази даних у разі аварії носія. При краху системи для відтворення поточного стану бази даних журнал застосовувався до її архівної копії.

2.3.4. Реляційна модель даних

Враховуючи відзначені в попередніх розділах недоліки мережних і ієрархічних моделей, можна сформулювати бажані вимоги до моделі даних:

- модель має бути зрозуміла користувачу, який не є професіоналом у програмуванні;
- поява нових аспектів використання даних і необхідність введення нових зв'язків не повинна приводити до реструктуризації всієї моделі даних і бази даних в цілому.

Моделлю даних, що задовольняє вищезгаданим вимогам, є реляційна модель, яку іноді називають також табличною моделлю. Докладніше реляційну модель баз даних розглянемо в наступному розділі, а зараз розглянемо питання про обробку даних на фізичному рівні.

2.4. Зберігання та обробка даних в зовнішній пам'яті ЕОМ

В сучасних СУБД домінує реляційна модель. Тому природньо приділити більше уваги структурам зберігання для табличної моделі. Проте, відзначимо, що розглянуті нижче структури зберігання, можуть використовуватися і для представлення мережевих і ієрархічних моделей. Як зовнішню пам'ять ми розглядаємо найбільш поширену в сучасних ЕОМ пам'ять прямого доступу. Пам'ять прямого доступу дає можливість звернення до будь-якого запису, якщо відома її адреса. Для спрощення викладу ми не конкретизуватимемо ряд службових полів, які містить фізичний запис.

2.4.1. Послідовне розміщення фізичних записів

В цій структурі зберігання записи в пам'яті розміщуються послідовно один за одним. Як вже наголошувалося, вважаємо, що всі записи мають однакову довжину. Фізична адреса запису може бути легко обчислена за номером запису (для відповідного обчислення необхідно знати формат відповідної фізичної пам'яті). Фізичний запис з номером *I* містить логічні записи з номерами

*(I – 1)*k+1*

*(I – 1)*k+2* .

*(I – 1)*k+k I=1, 2 ., [N mod k];*

[N mod k]позначимо найближче ціле наближення зверху *N/K* . Розглянемо, як реалізуються основні елементарні операції моделі даних в цій структурі зберігання, і оцінимо число цих операцій. Нагадаємо, що з погляду користувача в табличній моделі даних ці операції є операціями над рядками (колонками) таблиці.

2.4.2. Пошук запису із заданим значенням ключа

Пошук здійснюється дихотомічним методом. Номер фізичного запису, відповідний середині файлу, визначається найближчим цілим числом до числа [N mod k]/2. По цьому номеру визначається номер фізичного запису і запис прочитується в ОП. Перевіряється, чи запис із заданим значенням ключа лежить вище або нижче відповідного запису. Відповідна половина файлу ділиться пополам, процес повторюється доки не буде знайдений потрібний фізичний запис. $TP = C\log_2 [N \bmod k];$ де С - певна константа.

Після цього в ОП знайдений запис розблокується, в ньому відбудеться пошук потрібного логічного запису. Читання запису здійснюється аналогічно операції пошуку. Кількість звернень до ЗП дорівнює ТР.

2.4.3. Розміщення фізичних записів у вигляді спискової структури

Основна проблема у використанні описаних способів організації записів полягає у відображенні додавання логічного запису в довільне місце таблиці. При цьому доводиться

переписувати в пам'яті (зсовувати на одну позицію) фізичні записи, відповідні логічним записам таблиці, розташованим нижче за місце вставки рядка, що додається. Відповідну проблему можна усунути, використовуючи для представлення фізичних записів зв'язний список. Окрім цього списку у ЗП формується список вільних елементів («порожніх фізичних записів»), елементи якого використовуються при введенні нового запису з даними. Розглянемо, як реалізуються основні елементарні операції моделі даних в цій структурі зберігання. У випадку пошуку запису із заданим значенням ключа впорядкування записів по значеннях ключа не дає прискорення процедури пошуку. Це зв'язано з тим, що після ряду додавань нових записів і видалення якихось наявних записів фізична і логічна послідовність записів в списку істотно розрізнятимуться. При цьому буде неможливо за номером запису визначити її адресу, і звертатися до запису, що відповідає середині таблиці, для реалізації дихотомічного методу пошуку. Тому пошук можна вести тільки за допомогою перебору. В ОП читається перший запис списку, далі значення ключових полів логічних записів цього фізичного запису порівнюється із заданим значенням. Якщо значення співпали, потрібний запис знайдений, якщо не співпали, із запису вибирається адреса наступного запису списку, читається цей запис. Далі процедура повторюється. Середнє число звернень до ЗП буде дорівнюватиме *[N mod k]/ 2*. При коригуванні запису обчислюваний запис коригується і заноситься у ЗП на своє місце (за своєю адресою). Число звернень до ЗП - на одиницю більше, ніж при читанні.

Операція видалення логічного запису аналогічна операції коригування. Службове поле відповідного логічного запису позначається як «видалений запис». Сформований фізичний запис заноситься у ВП. Число звернень до ЗП дорівнює TP+1.

При додавання запису здійснюється операція пошуку і читання фізичного запису, в якому розташований запис з ключем РК. Якщо в цьому блоці є логічний запис, помічений як видалений, запис , що додається заноситься на її місце. Блок записується у ЗП. Число звернень до ЗП дорівнює TP+1. Якщо в цьому блоці немає логічних записів, помічених як видалені, необхідно додавати новий фізичний запис, який обрається із списку вільних елементів. З цією метою адреса зв'язку знайденого раніше фізичного запису замінюється на адресу початку списку вільних елементів. Читається перший фізичний запис списку вільних елементів. Адреса зв'язку цього запису замінює адресу початку порожнього списку. В ОП формується новий фізичний запис, який містить логічний запис, що додається. Як її адреса зв'язку заноситься адреса зв'язку з фізичного запису, що додається. Кожний з цих записів заноситься у зовнішню пам’ять. Число звернень до ЗП при додаванні запису приблизно дорівнює TP+3. Розглянутий метод організації структури зберігання достатньо ефективно вирішує проблеми додавання і видалення записів, але не знімає необхідності перебору при пошуку потрібного запису.

2.4.4. Використання індексів (індексація)

Як вже наголошувалося, впорядкування записів дозволяє використовувати дихотомічний метод пошуку потрібного запису і, тим самим, істотно скоротити одну з основних складових часу пошуку – число звернень до ЗП. Проте при цьому виникають проблеми з додаванням записів, пов'язані з необхідністю перезапису частини фізичних записів (зсуву). Для того, щоб використовувати дихотомічний пошук і не переміщати фізичні записи при додаванні нових записів використовується так зване логічне впорядкування фізичних записів (індексація). Основна структура зберігання містить записи початкової таблиці і представлена у вигляді нерегулярної послідовності фізичних записів.

Для можливої реалізації дихотомічного пошуку по певному ключу створюється додаткова структура зберігання (так званий індекс). Число записів в індексі дорівнює числу записів початкової таблиці (числу фізичних записів в основній структурі зберігання). Кожний запис індексу має два поля: ключове поле запису основної структури і вказівник - адреса запису основної структури з відповідним значенням ключа. Записи індексу (індексного файлу) впорядковані по значенню ключа. Адреси зв'язку цих записів визначають логічне впорядкування записів основної структури зберігання. Приклад відповідної структури зберігання наводиться на Рис. 2.13. Дану структуру зберігання називають ще інвертованим списком. Значення цього терміну полягає в наступному. Можна було б упорядкувати записи основної структури зберігання, об'єднавши їх у відповідний впорядкований список.

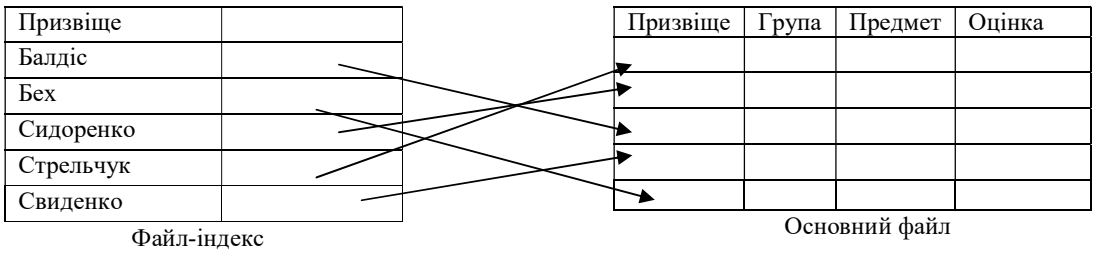


Рис.2.13. Індексація

В нашому випадку адреси зв'язку начебто вилучаються зі списку і включаються до складу файлу-індексу (інвертуються). Тому отримана структура інтерпретується як інвертований список. Пошук потрібного запису по заданому значенню ключа здійснюється в індексному файлі методом половинного ділення. Відзначимо, що оскільки записи індексу містять всього два поля, сумарний об'єм записів індексу невеликий, тому індекс, як правило, цілком прочитується для обробки в ОП за одне звернення до ЗП. Після того, як в індексному файлі знайдений запис, що шукається, за адресою зв'язку зчитується повний відповідний запис основної структури зберігання. Якщо потрібний пошук по іншому ключу, будується

ще один індекс по відповідному ключу. Таким чином, по будь-якому ключу пошук можна здійснювати дихотомічним методом.

Для пошуку запису із заданим значенням ключа із ЗП читається індексний файл (кількість звернень до ЗП для цього залежить від об'єму індексного файлу, як правило, невелика і набагато менше числа записів N). Після знаходження потрібного запису в індексному файлі, читається відповідний запис основного файлу (одне звернення до ЗП). При коригуванні запису потрібний запис коригується і заноситься на своє місце (ще одне звернення до ЗП). При видаленні запису знайдений запис позначається як видалений в основному файлі, відповідний запис в індексному файлі видаляється, змінений індекс записується у ЗП. Число звернень до ЗП в цьому випадку в порівнянні з числом звернень до ЗП при пошуку збільшується на два. При додаванні запису запис, що додається, заноситься в кінець основного файлу. Формується новий запис індексу, відповідний запису, що додається. Записи індексу переупорядковуються за значеннями ключа, і індекс заноситься у ЗП. Число звернень до ЗП в цьому випадку, в основному, визначається читанням – записом індексу. Таким чином, використання індексів дозволяє ціною деякого збільшення об'єму пам'яті (за рахунок індексу), що використовується, істотно скоротити час реалізації основних операцій. У зв'язку з цим індексація використовується в багатьох сучасних СУБД.

2.4.5. Бінарне дерево (B-дерево).

Структура бінарного дерева є подальшого розвитком концепції використання індексів (будується індекс над індексом). Бінарне дерево будується таким чином. Послідовність записів, відповідна записам початкової таблиці, упорядковується по значеннях первинного ключа. Логічні записи об'єднуються в блоки (по до записів в блоках). Значенням ключа блоку є мінімальне значення ключа у записів, що входять в блок. Послідовність блоків є останнім рівнем B-дерева. Будується індекс попереднього рівня. Запис цього рівня містять значення ключа блоку наступного рівня і вказівник-адреса зв'язку відповідного блоку запису цього рівня також об'єднуються в блоки (по K записів). Потім аналогічно будується індекс більш високого рівня до тих пір, доки кількість записів індексу на певному рівні буде не більша *K*. Отримана структура зображена на Рис.2.14. (для спрощення малюнка на рівні 4 представлені тільки ключі логічних записів і не представлені значення інших полів цих записів). Жирними лініями показано пошук потрібного блоку.

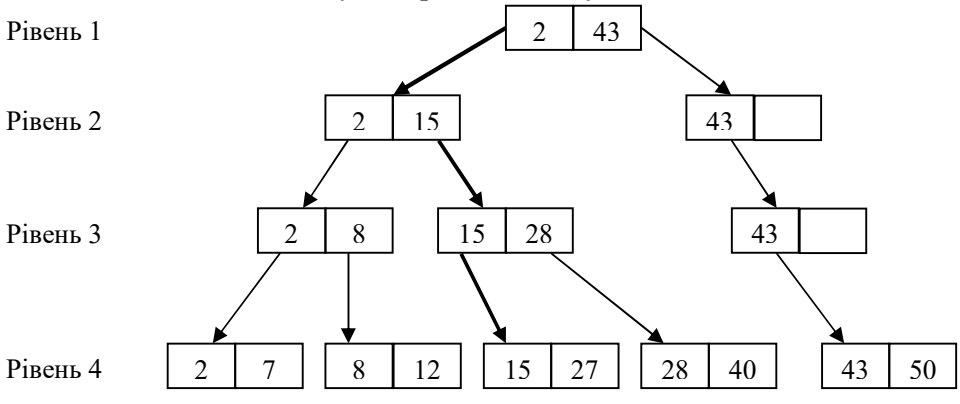


Рис.2.14. Схема пошуку по бінарному дереву

При пошуку і читанні запису із заданим значенням ключа спочатку читається верхній індекс. Порівнюємо задане значення ключа із значенням ключа записів індексу. Якщо задане значення ключа більше або дорівнює значенню ключа чергового запису індексу (якщо такий запис є), то за вказаною адресою зв'язку, читається блок записів індексу наступного рівня. Далі процес повторюється. Вважаємо, що всі блоки розташовані у ЗП. Тоді, число звернень до ЗП при пошуку інформації буде дорівнювати кількості рівнів дерева. Кількість рівнів дерева дорівнює мінімальному значенню l, при якому виконується $k_l \geq N$ (N – кількість логічних записів).

Модифікація (коригування) запису вимагає пошуку і читання запису. Після чого змінюються кориговані поля. Якщо коригується не ключ запису, то змінений запис заноситься на своє місце. Якщо змінено значення ключа, то старий запис видаляється (у відповідному блоці з'являється «порожній запис»), а змінений запис заноситься так само, як запис, що знову додається.

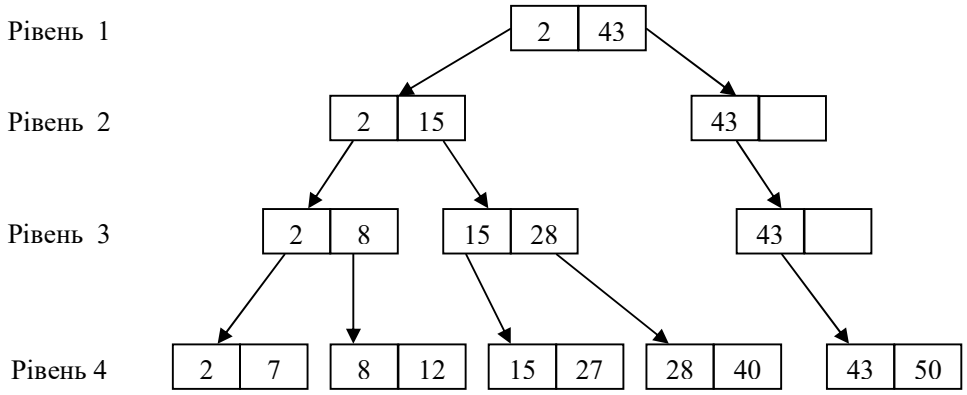
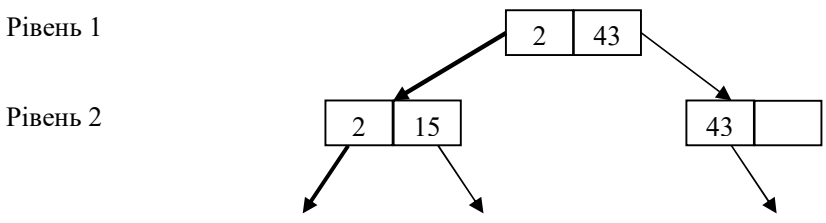


Рис.2.15. Схема модифікації (коригування) запису

Для *видалення запису* після пошуку знайденого запису він видаляється (у відповідний блок на місце цього запису заноситься «порожній запис»). Пунктиром помічено видалений блок і нульвий вказівник



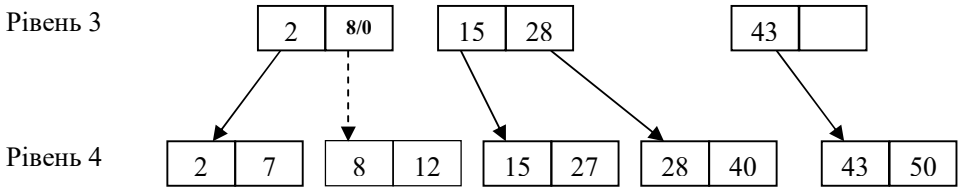


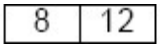
Рис. 2.16. Схема видалення запису

При *додаванні запису* перш за все визначаємось, де має бути розташований запис із заданим значенням ключа, що додається. Відповідна процедура аналогічна вищеописаній процедурі пошуку записів із заданим значенням ключа. Якщо в знайденому блоці низького рівня є «порожній запис», запис, що додається, заноситься в цей блок (з необхідним переупорядковуванням записів усередині блоку). Якщо у відповідному блоці низького рівня немає порожнього місця, блок ділиться на два блоки. В перший з них заноситься [*k/2*] записів, в другий заноситься інші. Значенням ключа кожного з вказаних блоків буде, як і описано раніше, мінімальне значення ключів у записів, що входять в блок. Запис, що додається, заноситься в той блок, значення ключа якого менше значення ключа запису, що додається. Поява нового блоку з новим значенням ключа обумовлює необхідність формування відповідного нового запису в індексі на попередньому рівні. Цей запис містить нове значення ключа нового блоку і вказівник на його місцерозташування. Процедура додавання такого запису аналогічна описаній вище. Знаходиться блок попереднього рівня, куди повинен бути поміщений цей запис. Якщо в блоці є порожнє місце, запис додається в блок, якщо блок повний, він ділиться на два блоки, запис 143 заноситься в один з блоків, формується запис індексу попереднього рівня і т.д. Можливий варіант, коли доведеться ділити блок найвищого рівня і формувати ще один рівень дерева. Розглянемо для прикладу, зображеного на Рис. 38, додавання запису з ключем 10.

1. Порівняння на першому рівні. 2<10<43 Рух по лівій гілці.

2. Порівняння на другому рівні. 2<10<15 Рух по лівій гілці.

3. Порівняння на третьому рівні. 2<8<10 Рух по правій гілці. Знайдено блок



4. Блок заповнений. Він ділиться на 2 блоки з ключами 8 12 Порівняння 8<10<12. Запис з ключем 10 заноситься в перший блок

5. На низькому рівні з'явився новий запис із значенням ключа 12. Необхідне додавання нового запису з ключем 12 і вказівником на запис низького рівня до індексу попереднього рівня.

6. Запис з ключем 12 рівня 3 повинен додаватися в блок 2 | 8. Блок повний, він ділиться на два блоки з ключами 2 і 8 Порівняння 8<12. Запис додається в другий блок 8 | 12

7. На рівні 3 з'явився блок з новим ключем

8. Необхідне додавання нового запису з ключем 8 і вказівником на відповідний блок рівня 3 на рівні 2.

2 | 8

8. Запис з ключем 8 рівня 2 повинен додатися в блок 2 15 . Блок повний, він ділиться на два блоки. 2 15 2<8<15 Запис додається в блок 1 2 8 . 144 2 | 43.

9. На рівні 2 з'явився блок з новим ключем 15, необхідне додавання нового запису з ключем 15 і вказівником на відповідний блок рівня 2 на рівні 1.

10. Запис з ключем 15 рівня 1 повинен додаватися в блок 2 43 . Блок повний, він ділиться на два блоки. 2 43 2<15<43 Запис з ключем 15 додається в перший блок 2 15 43

11. Необхідно сформувати ще один рівень дерева 2 43 .

Отримана структура матиме вигляд, представлений на Рис . 2.17.

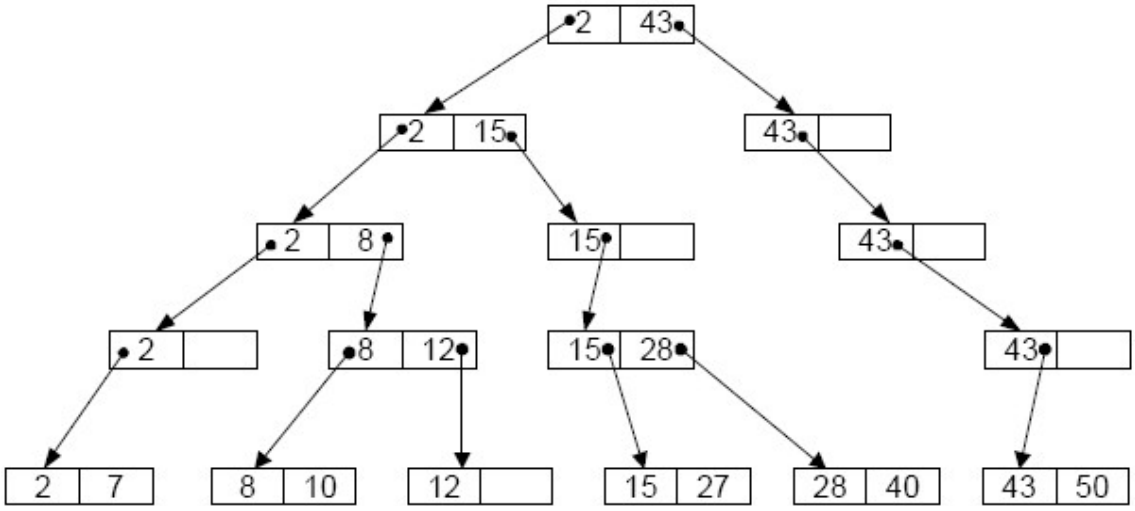


Рис. 2.17. B-дерево після додавання елемента

Необхідно відзначити, що прийом ділення повністю заповненого блоку пополам при додаванні в нього, приведе до того, що блоки будуть заповнені, в середньому, наполовину. Тоді процедура додавання запису буде істотно менш трудомісткою (якщо в потрібному блоці є місце, запис додається в цей блок і більш високі рівні не перебудовуються).

Структура зберігання у вигляді B-дерева дозволяє ефективно проводити операції пошуку, читання, видалення, модифікації з оцінкою числа звернень до зовнішньої пам'яті при кількості рівнів дерева l ($l \approx log_k N$), що істотно менше за число обходів при переборі [N mod k]. Процедура додавання запису теж достатньо ефективна.

2.4.6. Розміщення записів з використанням хешування

На відміну від всіх інших способів організації структур зберігання тут вибраний особливий спосіб групування. Певним чином вибирається так звана хеш-функція *f(x).* Аргументом цієї функції є значення x первинного ключа логічного запису. Тоді *f(x*) вказує адресу розташування блоку, в якому повинен знаходитися логічний запис із значенням ключа x. По можливості функція *f* повинна рівномірно розподіляти значення *x* у фізичних блоках. Обговоренню можливих хеш - функцій присвячено достатньо багато літератури, тому тут ми не торкатимемося цього питання. Можна лише додати, що іноді, виходячи із специфіки множини значень *x* первинного ключа, можна побудувати функцію *f,* що задовольняє всім необхідним умовам. Таким чином, логічний запис таблиці із значенням *x* первинного ключа розміщується в блоці зовнішньої пам'яті за адресою *Y= f(x*). В цьому блоці може знаходитися не більш К записів. Може виявитися, що обрана функція відображає на одну адресу пам'яті (один блок) більше К записів.

Виникає так звана колізія. Можливим способом розв’язку колізій є використання додаткової області. Ідея цього прийому наступна: якщо черговий запис розподіляється за допомогою функції хешування в блок, а він повністю заповнений, то в області переповнювання формується список записів, відповідних цьому блоку, з включенням в нього вказаного запису, а в сам блок заноситься вказівник – адреса зв'язку на перший запис цього списку. Можливі і інші способи розв’язку колізій . Розглянемо реалізацію основних операцій і дамо оцінку числа звернень до ЗП при їх виконанні.

Для пошуку запису із заданим значенням ключа і читання по визначеному значенню ключа *x* підраховується значення функції *f(x)*. Далі прочитується з ЗП блок, що знаходиться за адресою *f(x).* В ОП усередині цього блоку перебором проводиться пошук потрібного запису. Якщо записів в блоці немає, то по вказівнику в блоці (адресі зв'язку) читається перший запис списку переповнювання, що відноситься до цього блоку. Далі необхідний запис шукається за цим списком. Кількість звернень до ЗП при цьому дорівнює:

- одиниці, якщо запис знаходиться в блоці;

- одиниці плюс число записів у відповідному цьому блоку списку області переповнювання .

При *модифікації запису* здійснюється пошук і читання запису, потім в ОП модифікуються поля запису (які не є первинним ключем), запис заноситься на своє місце.

Число звернень до ЗП в цьому випадку на одиницю більше, ніж при читанні запису. Якщо модифікується значення ключа, то занесення запису здійснюється як введення нового запису (додавання). Таким чином, розглянута структура хешування є найбільш ефективною структурою збереження по критерію мінімізації кількості звернень до ЗП при реалізації основних операцій. Проте залишається фундаментальна проблема: пошук ефективної хеш-функції.

Зауважимо, що в СУБД можуть використовуватися як кожну з вище розглянутих структур окремо, так і їх комбінації. Так, наприклад, у ряді промислових систем UNIBAD, БАНК для ЕОМ типу IBM 360/370 (ЄС ЕОМ), PARADOX для персональних ЕОМ використовується наступні комбінації методів:

- розміщення записів по первинному ключу організовано з використанням хешування;
- послідовність проходження записів по вторинному ключу визначається за допомогою спискової структури.

Контрольні запитання і вправи.

1.Проведіть оцінку трудомісткості модифікації файла з даними розміром 5 МБ, розмір блока 5 байт в залежності від способу організації :

1.1 Послідовні записи

1.2 Індексований файл зі щільним записом

1.3 Індексований файл типу B-дерево з трьома рівнями.

2. В чому полягають особливості структуризації даних, запропонованих CODASYL?
3. Які можливі варіанти моделей групових відношень?
4. Сформулюйте визначення моделей даних
5. Визначити основні поняття структуризації даних
6. В чому полягають особливості моделі обробки даних з централізованою архітектурою?
7. Визначити базові поняття трирівневої архітектури даних .
8. В чому полягає спосіб формального опису баз даних?
9. Дати характеристику трьох рівнів стандарту ANSI/SPARC.
10. Сформулювати основні операції маніпулювання ієрархічно організованими даними.
11. Які особливості обмеження цілісності ієрархічно організованими даними?

12. Сформулювати основні операції маніпулювання даними, організованими за мережевим принципом.
13. Які особливості обмеження цілісності даних, організованими за мережевим принципом.
14. Охарактеризуйте ідею реляційної моделі даних.

Розділ 3. Фізичні моделі даних

Розглянемо основні положення обробки даних на фізичному рівні. Почнемо з характеристики оперативної пам’яті, в якій власне і відбувається обробка даних:

- одиницею пам'яті є байт;
- пам'ять прямо адресується (кожний байт має адресу);
- процесор вибирає для обробки потрібні дані, безпосередньо адресуючись до послідовності байтів, що містять ці дані.

Відзначимо основні властивості зовнішньої пам'яті, в якій дані зберігаються, зчитуються та записуються:

- мінімальною одиницею, що адресується, є фізичний запис;
- для подальшої обробки (наприклад роботи з полями) запис повинен бути розміщений в оперативну пам'ять;
- час читання запису в ОП (занесення запису з ОП у ЗП) на декілька порядків вище за час обробки процесором запису з ОП;
- організація обміну здійснюється порціями, оскільки неможливо обробляти відразу всю базу даних.

3.1. Представлення екземпляра логічного запису

Логічний запис представляється в оперативній пам'яті відображено на Рис.:

ЛОГІЧНИЙ			ЗАПИС		ПОСЛІДОВНІСТЬ			БАЙТ
Поле1	Поле2	...	ПолеN	→	B_1	B_2	...	B_N

Рис. 3.1. Представлення логічного запису в оперативній пам’яті

Відзначимо, що вказане представлення не залежить від того, що ці записи призначені для мережевої, ієрархічної чи реляційних моделей. У разі мережевої і ієрархічної моделей деякі поля можуть бути вказівниками, тоді послідовність байтів, що використовується для зберігання цих полів містить адресу початку послідовності байтів, відповідного запису – члена відношення. В більшості сучасних СУБД використовується формат записів фіксованої довжини. В цьому випадку всі записи мають однакову довжину, що визначається сумарною

довжиною полів, які становлять запис. Використання в СУБД інших форматів записів (змінної довжини, невизначеної довжини) зустрічаються значно рідше, тому розглядати їх не будемо. Відзначимо, що поля запису, що приймають значення істотно різної довжини в різних екземплярах записів, в реальних задачах зустрічаються достатньо часто. Прикладом може служити поле резюме в записі СПІВРОБІТНИК. Резюме може складати півсторінки тексту, сторінку і т.д. Виникає проблема, як цю інформацію змінної довжини представити в запису фіксованої довжини. Можливим варіантом є встановлення розміру відповідного поля за максимальним значенням. В цьому випадку у багатьох екземплярів запису вказане поле буде заповнено не повністю і, таким чином, пам'ять ЕОМ буде використовуватися неефективно. Більш ефективний прийом організації таких записів полягає в наступному. Замість поля (полів), що приймає значення істотно різної довжини, в запис включається поле-вказівник на область пам'яті, де розміщуватиметься значення початкового поля. Як правило, ця область пам'яті є областю зовнішньої пам'яті прямого доступу. В процесі введення відповідного значення у виділеній області пам'яті займається стільки пам'яті, скільки треба для розміру цього значення. На Рис.3.2. представлений приклад представлення екземплярів *записів* з K полів, причому поле K – приймає значення відповідно різної довжини у різних екземплярів записів.

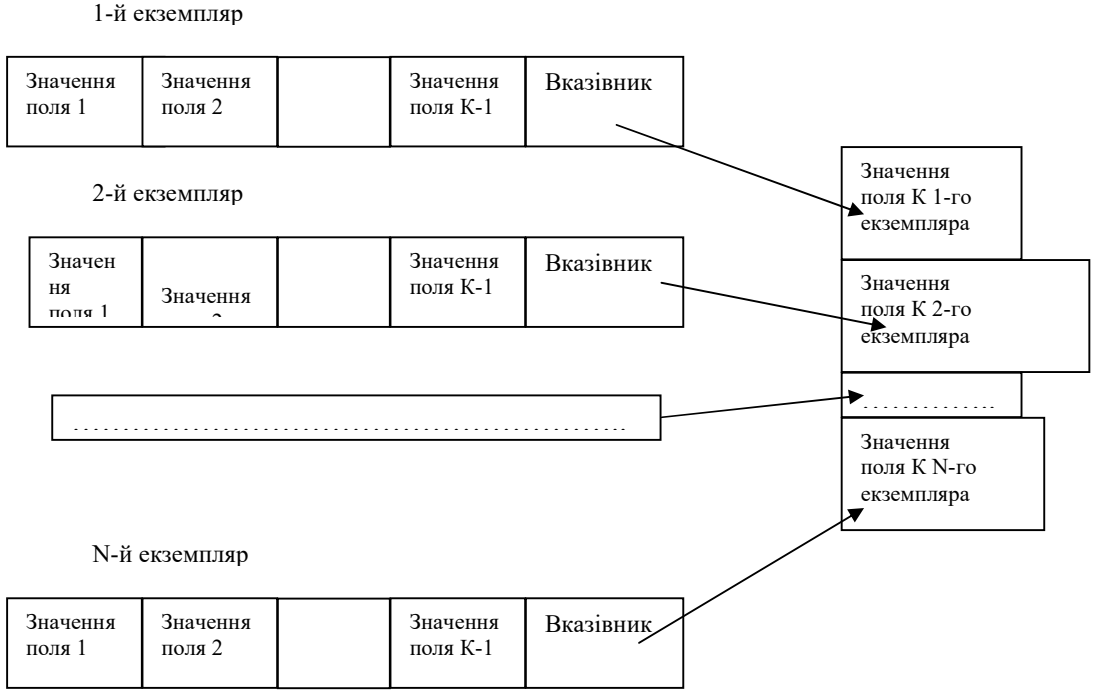


Рис.3.2. Представлення екземплярів записів різної довжини

3.2. Організація обміну між оперативною і зовнішньою пам'яттю

Одиницею обміну даними між оперативною і зовнішньою пам'яттю є фізичний запис. Фізичний запис читається (записується) за одне звернення до зовнішньої пам'яті. Зокрема, фізичний запис може відповідати одному логічному запису. Число звернень до зовнішньої пам'яті при роботі з базою даних визначає час відгуку системи. У зв'язку з цим, для зменшення числа звернень до БД при роботі з нею, збільшують довжину фізичного запису (об'єднують в один фізичний запис декілька екземплярів логічних записів). В цьому випадку фізичний запис називають також блоком, число К екземплярів логічних записів, що становлять фізичний запис, називають коефіцієнтом блокування. Введення початкових даних в БД здійснюється таким чином:

- в ОП послідовно вводяться *К* екземплярів логічних записів (кортежів);
- введені *К* екземплярів об'єднуються у фізичний запис (блок);
- фізичний запис заноситься в зовнішню пам'ять. Введення К екземплярів записів початкової таблиці, складових і- го фізичнго запису, зображено на Рис.3.3.

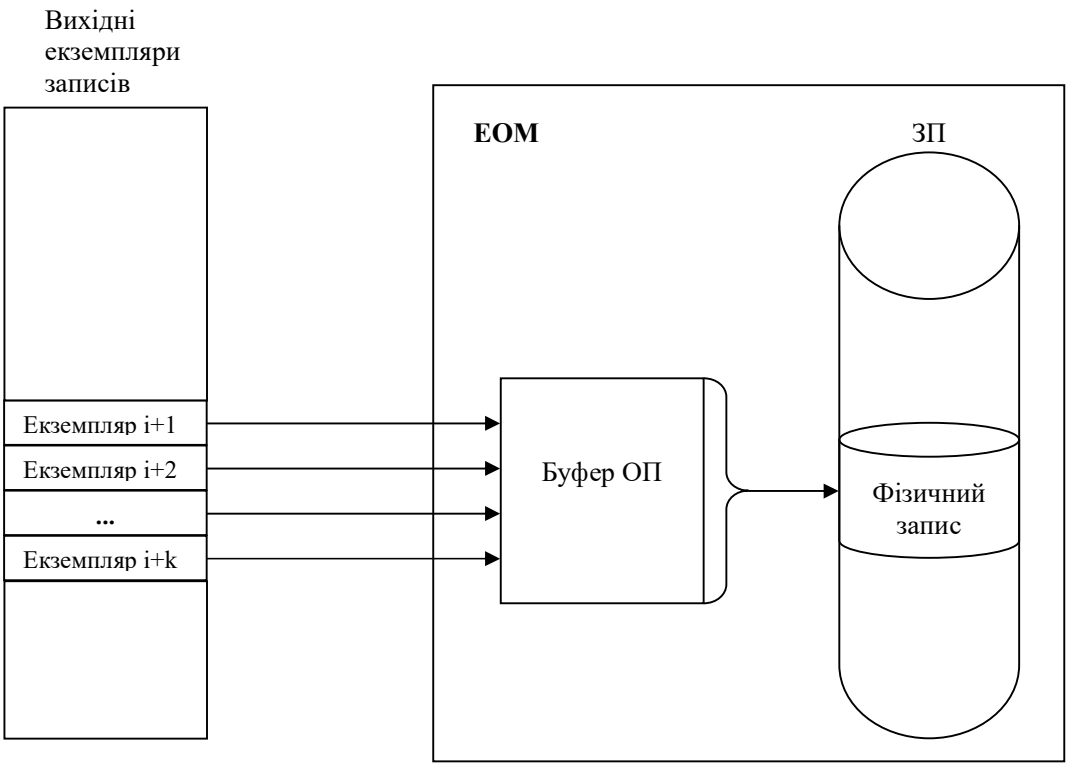


Рис. 3.3. Схема вводу записів у зовнішню пам’ять

Обробка даних, що зберігаються в зовнішній пам'яті, здійснюється таким чином:

- запис (блок) прочитується в оперативну пам'ять;
- екземпляри логічних записів у середині блоку (вибираються потрібні поля, проводиться порівняння ключового поля із заданим значенням, здійснюється коригування полів, виконуються операції видалення і т.п.).

3.3. Механізми реалізації звернень до баз даних для різних моделей баз даних

Ідея розділу обробки даних, що належить БД , на три рівні(в подальшому число рівнів може бути збільшене) дає справжню свободу користувачам СУБД від особливостей апаратного і програмної реалізації конкретної інформаційної системи. Таким способом досягається достатній рівень продуктивної абстракції, яка дає можливість зводити технологію обробки даних на рівень маніпулювання даними мовою, реалізованої для даної моделі баз даних. Але ця відносна легкість і свобода дається дорогою ціною - розробкою алгоритмів і відповідних їм низькорівневих програм для реалізації звернень до даних в СУБД. Розглянемо особливості цих механізмів для різних моделей баз даних. При цьому врахування фізичної структури бази даних є важливим, але не єдиним фактором.

3.3.1 Механізм звернень до фізичної пам’яті для ієрархічної моделі

Це найбільш інтуїтивно зрозуміла модель, бо людям ментально притаманно маніпулювати ієрархічними поняттями. Будь-яка група об'єктів, в якій всі об'єкти можна класифікувати як «батьки» або «діти» («батько» може мати множину «дітей», але не навпаки), організована у вигляді ієрархічного дерева. При роботі з ієрархіями використовується «сімейна термінологія» (батьки, діти, онуки, предки, нащадки), оскільки сім'я є найпоширенішим прикладом об'єктів (в даному випадку - людей), з'єднаних ієрархічними відношеннями. В той же час, місце об'єкту в ієрархічному дереві - не більше ніж умовне позначення зв'язку з іншими об'єктами. Ієрархічна структура лише допомагає зберегти, упорядкувати і знайти об'єкт.

Для детального розгляду механізму обробки даних нам знадобиться приклад. Розглянемо учбовий приклад на базі фірми, що надає послуги аренди нерухомості, яка має відділення в різних регіонах. Даний приклад запозичений з монографії Коннолі та Бегг[]. Відділення(*Branch)* має унікальний номер, адресу, номер телефона і факсу. В штаті кожного відділення(*Staff*) передбачені працівники, кожний з яких має табельний номер, ПІБ, номер телефону, стать, дату народження, посаду, оклад, дату зарахування на роботу. Крім того, компанія веде облік найближчих родичів співробітників(батьки, дружина(чоловік), діти).

Об’єкти нерухомості(*Property_for_Rent*) характеризуються наступним набором: адреса, тип об’єкта, кількість кімнат, помісячна арендна плата. Власники нерухомості ідентифікуються номером, ПІБ, адресою, телефоном. Потенційні орендатори описуються ПІБ, адресою, телефоном, типом нерухомості, максимальною ставкою орендної оплати. Крім цього в практиці діяльності компанії передбачена процедура оглядин(*View*) об‘єкта нерухомості клієнта, при цьому фіксується дата і результат оглядин.

Важливою особливістю ієрархічної структури є те, що в ній дочірні екземпляри не можуть існувати без наявності батьківських екземплярів. Це викликано присутністю в даній структурі такої вбудованої залежності, при якій при видаленні кореневого екземпляра віддаляється все дерево (або піддерево). Подібний підхід має як переваги, так і недоліки. Однією з переваг є те, що цілісність на рівні посилань автоматично забезпечується там, де екземпляри записів повністю залежать від інших екземплярів записів.

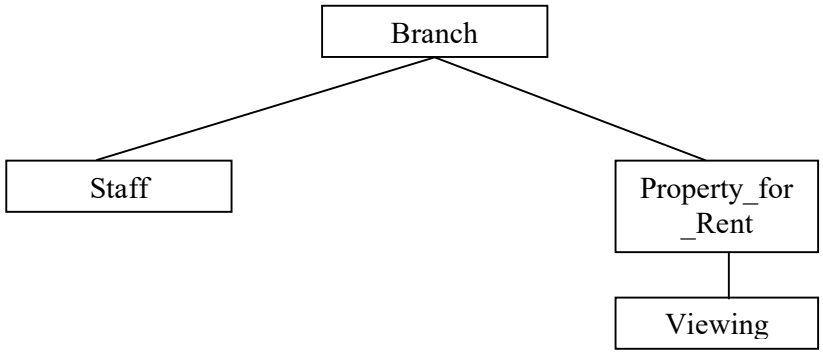


Рис.3.4. Ієрархічна схем для БД орендної фірми

Наприклад, на Рис.*3.4.*, видалення екземпляра типу запису *Property_for_Rent* приводить до видалення пов'язаних з ним екземплярів типу запису *Viewing*. І, навпаки, екземпляр запису типу *Viewing* не можна вставити, якщо відповідного йому екземпляра запису типу *Property_for_Rent* не існує.

Недоліки такої структури пов'язані з неможливістю зберігання екземплярів записів, які не мають жодних батьківських записів. Наприклад, видалення екземпляра запису Branch приведе до видалення всіх пов'язаних з ним екземплярів записів інших типів. В подібній структурі зовсім непросто зберегти в базі запису типу *Staff* або типу *Property_for_Rent,* якщо вони не відносяться до жодного з наявних записів *Branch*. Звичайно, існує декілька методів розв’язку цієї проблеми - наприклад, шляхом введення порожнього підстановлювального запису *Branch*, - але основна проблема полягає в тому, що подібна структура не дозволяє достатньо просто моделювати характеристики екземплярів даних з реального світу.

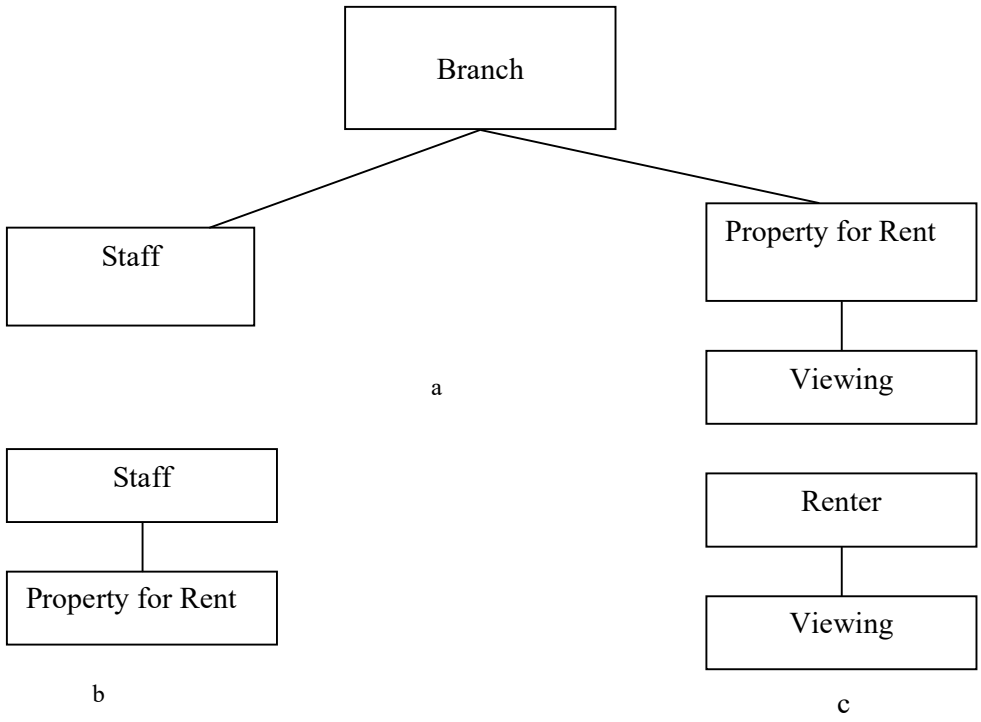


Рис.3.5 . Додаткові ієрархічні структури, необхідні для представлення складних зв'язків (Рис.3.5. b) та Рис.3.5. c))

Далі розглянемо систему IMS як типового представника ієрархічних СУБД. СУБД IMS - велика і складна система, для ефективної роботи з якою необхідно володіти обширними знаннями і навичками. Саме ця особливість часто примушувала розробників відмовлятись від неї. Проте в часи її розквіту конкурентів у неї було мало.

Термінологія, що використовується в системі IMS відрізняється від сучасної. Наприклад, в системі IMS використовуються такі поняття:

- тип сегменту (segment type), який застосуються замість терміну «тип запису»;
- фізична база даних, яка містить декілька записів бази даних;
- опис фізичної бази даних, або ОФБД (Physical Database Description - DBD), яке визначає структуру і характеристики фізичної бази даних і створюється за допомогою мови DL/1.

Кожна фізична база даних визначається на основі її фізичного ОФБД, яке може бути строгим ієрархічним впорядкуванням сегментів. Вона також може включати логічні зв'язки (logical relationships), які використовуються для моделювання складних зв'язків, наприклад зв'язків типу M:N. Оскільки логічні зв'язки визначаються у фізичному ОФБД, то результуюча логічна структура для системи IMS повинна бути визначена ще до її використання. Ця задача виконується за допомогою логічного ОФБД**,** в якому ефективно визначається фактична

структура базових фізичних ОФБД, що використовуються. На зовнішньому рівні кожний користувач має представлення бази даних, яке називається блоком специфікації програми (або БСП-блоком (Program Specification Block - PSB)). Призначений для користувача БСП-блок складається з одного або декількох комунікаційних блоків програми (або КБП-блоків (Program Communication Block - PCB)), кожний з яких визначає зовнішню схему окремої програми.

Фізична база даних визначається структурою зберігання даних бази, яка є ієрархією типів сегментів, причому деякі з них можуть містити логічні вказівники на типи сегментів в інших структурах даних бази (фізичних базах даних). Кожний тип сегменту визначається за допомогою фізичного ОВД, яке задає структуру бази даних. В лістингу приведений опис кореневого (батьківського) і залежного (дочірнього) типів сегменту. В кожному описі сегменту спочатку указуються ім'я сегменту, його розмір в байтах і ім'я батьківського сегменту, а потім слідують описи полів. Кожне поле має ім'я, розмір, початкову позицію у середині сегменту і заданий тип, наприклад визначуваний символом С. Слідом за ім'ям поля може розташовуватися позначення SEQ, яке служить для вказівки того, що дане поле виконує роль унікального ключового поля.

Лістинг 3.1. Приклад опису батьківського і дочірнього сегментів

```
SEGM NAME = Branch, BYTES = 23, PARENT = Про
ім'я сегменту, його розмір і ім'я його батьківського сегменту
FIELD NAME = (Bno, SEQ), BYTES = 3, START = 1, TYPE = 3
ім'я поля, індикатор ключа, розмір і тип поля
FIELD NAME = Street, BYTES = 20, START = 4, TYPE = 3
ім'я поля, його розмір і тип
SEGM NAME = Staff, BYTES = 35, PARENT = Branch
FIELD NAME = (Sno, SEQ), BYTES = 5, START = 1, TYPE = 3
FIELD NAME = Fname, BYTES =15, START = 6, TYPE = 3
FIELD NAME = Lname, BYTES = 15, START = 21, TYPE = 3
```

Сутність ієрархічної бази даних являють ієрархічні структури. Фізична база даних є ієрархічною структурою, яка може включати зв'язки типу 1:М і 1:1. Наприклад, для нашого учбового прикладу про орендну фірму ця структура може виглядати як дуже проста ієрархія (Рис.3.6. а)) , або більш загальний варіант ієрархії (Рис.3.6. б)) - , в якій один тип запису залежить від іншого.

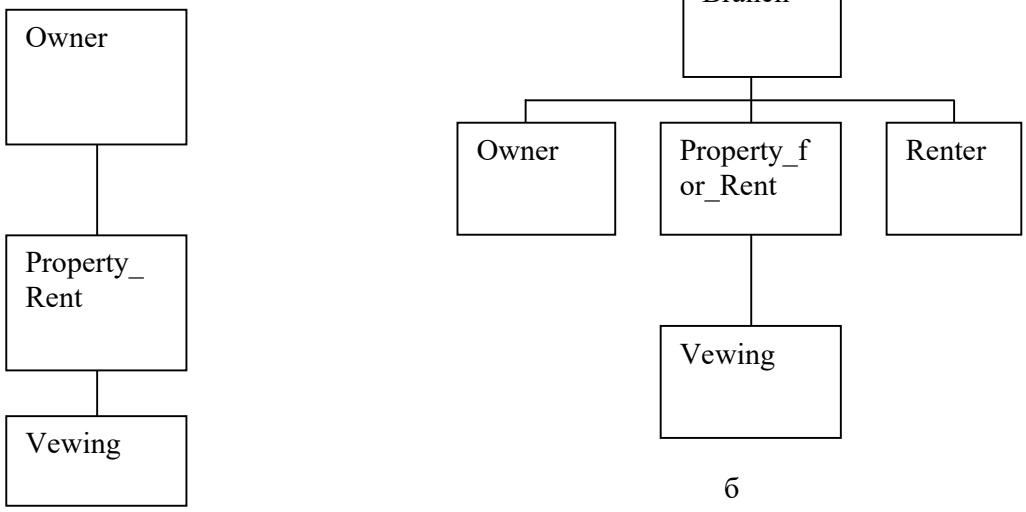


Рис.3.6. Основні варіанти ієрархічних структур в системі IMS:

а) проста ієрархія;

б) узагальнена ієрархія

Як вже згадувалося вище, в ієрархічних структурах достатньо складно організувати роботу із зв'язками типу M:N. В системі IMS один з прийомів полягає в збереженні даних в двох місцях, тобто за рахунок підвищення надмірності даних в базі. Наприклад, якщо об'єкт нерухомості рекламується в декількох газетах, а в одній газеті рекламується відразу декілька об'єктів нерухомості, то зв'язок між записами Newspaper і Property_for Rent має тип M:N. Цей зв'язок можна представити одним із способів, показаних на Рис.3.7. Як показано на Рис.3.7. а, запис з описом одного і того ж об'єкту нерухомості може бути продубльований для кожної газети, в якій рекламується даний об'єкт. На Рис.3.7. , б показаний інший варіант, в якому запис з описом однієї і тієї ж газети може бути продубльований для кожного об'єкта нерухомості, який в ній рекламується.

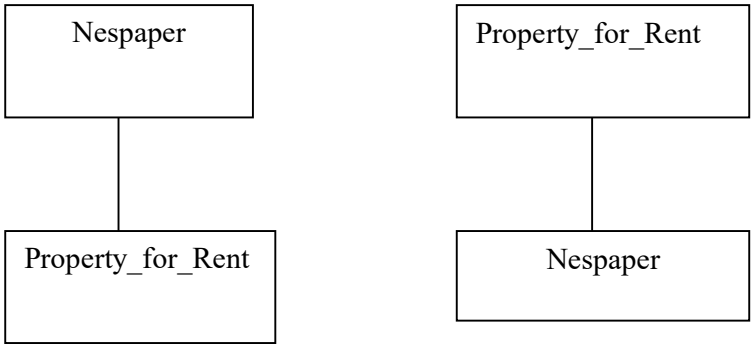


Рис. 3.7. Два варіанти представлення зв'язку типу M:N між записами *Newspaper* і *Property for Rent*

3.3.2. Структури зберігання даних для ієрархічної моделі

Кожний фізичний файл має власну структуру зберігання, яка визначається вимогами до його обробки. Основні структури зберігання діляться на наступні дві групи (хоча є і деякі інші типи):

- структури зберігання з послідовним доступом;
- структури зберігання з прямим доступом.

Нижче перераховані найпоширеніші структури зберігання, що використовуються в рамках вказаних груп.

- Ієрархічний послідовний метод доступу (*Hierarchical Sequential Access Method - HSAM*).
- Ієрархічний індексно-послідовний метод доступу(*Hierarchical Indexed Sequential Access Method - HISAM).*
- Ієрархічний прямий метод доступу (*Hierarchical Direct Access Method - HDAM*).
- Ієрархічний індексно-прямий метод доступу (*Hierarchical Indexed Direct Access Method - HIDAM*).
- Основне сховище бази даних (*Main Storage DataBase - MSDB*).

- База даних для введення інформації (*Data Entry DataBase - DEDB*).
- Узагальнений послідовний метод доступу (*Generalized Sequential Access Method - GSAM).*

В структурах послідовних методів доступу сегменти фізично розташовуються поряд один з одним в ієрархічному порядку, тоді як в структурах методів прямого доступу сегменти містять вказівники на інші сегменти.

Фізичний опис бази даних системи IMS містить ім'я фізичної бази даних, її базову структуру файлів, а також описує структуру бази даних. Ця структура включає кореневий сегмент разом зі всіма залежними від нього типами сегментів. Кожний тип сегменту визначається в тому порядку, в якому він зберігається в цій структурі. Наприклад, в структурі бази даних, показаної в табл. , типи сегментів будуть визначені в наступному порядку: *Branch, Staff, Property_for_Rent і Viewing.*

В лістингу 3.2. приведений фізичний опис БД для певної частини бази даних компанії.

Кожний опис починається з вказівки імені бази даних, за яким слідує назва структури зберігання. Наприклад, база даних *BranchDB* має структуру зберігання *HID AM*. Потім в ієрархічному порядку розташовуються типи сегментів, за якими йдуть завершальні команди генерації бази даних і закінчення цього процесу. Опис типу сегменту містить ім'я його батьківського сегменту (якщо він не є кореневим), довжину і відомості про поля. Звичайно кожний тип сегменту містить деякі поля даних, для кожного з яких указується початковий зсув в байтах усередині сегменту, а також довжина в байтах. Крім того, одне поле може бути визначено як ключове в кореневому сегменті, який може бути унікальним або неунікальним. Якщо деяке ключове поле задано в кореневому сегменті (за винятком баз даних із структурою HSAM), то прикладна програма зможе легко отримати доступ до заданого запису в базі даних. Ключове поле в системі *IMS* називається полем послідовності і позначається символами SEQ. За умовчанням воно є унікальним, якщо спеціально не обумовлене зворотнє. Символ U звичайно використовується для позначення унікальних значень, а символ M - для позначення можливості наявності дублікатів. Кореневий сегмент в структурах *HISAM, HDAM* і *HIDAM* обов'язково повинен містити унікальне поле послідовності.

Лістинг. 3.2. Приклад фізичного ОФБД з вказівкою односторонніх логічних зв'язків для частини бази даних компанії , що містить відомості про арендовані об'єкти нерухомості

DBD NAME = BranchDB, ACCESS = HIDAM
SEGM NAME = Branch, BYTES = 23, PARENT = 0
FIELD NAME = (Brio, SEQ), BYTES = 3, START = 1, TYPE = 3
FIELD NAME = Street, BYTES = 20, START = 4, TYPE = 3
SEGM NAME = Staff, BYTES = 35, PARENT = Branch

FIELD NAME = (Sno, SEQ), BYTES = 5, START = 1, TYPE = 3

FIELD NAME = Fname, BYTES = 15, START = 6, TYPE = 3

FIELD NAME = Lname, BYTES = 15, START = 21, TYPE = 3

SEGM NAME = Property for_Rent, BYTES = 79, ARENT = ((Branch) (Owner, V, OwnerDB)), POINTER = LPARNT

ім'я сегменту, розмір, фізичний батько, логічний батько з вказівкою типу і розташування

```
FIELD   NAME = (Pno, SEQ), BYTES = 5, START = 1, TYPE = C
FIELD   NAME = Street, BYTES =20, START = 6, TYPE = C
SEGM   NAME = Viewing, BYTES =13, PARENT = Property_for Rent
FIELD   NAME = (Rno, SEQ, M), BYTES = 5, START = 1, TYPE = C
FIELD   NAME = Date, BYTES = 8, START = 6, TYPE =C3
DBDGEN
FINISH
END

DBD NAME = OwnerDB, ACCESS = HIDAM

SEGM    NAME = Owner, BYTES =103, PARENT = 0
FIELD   NAME = (Ono, SEQ, U), BYTES = 5, START = 1, TYPE =C
FIELD   NAME = Fname, BYTES =15, START = 6, TYPE = C
        .
        .
        .
FIELD   NAME = Tel-no ....
LCHILD NAME = (Property_for_Rent, BranchDB)
```

ім'я і розташування логічного дочірнього сегменту

DBDGEN

FINISH

END

Всі вказівники визначаються у фізичному ОВД. Вони включають не тільки типи вказівників, допустимих для оголошеної структури файлу (виключаючи HSAM), але також будь-які логічні вказівники**,** які використовуються для визначення логічної структури бази даних серед різних фізичних структур баз даних. В лістингу показаний приклад визначення однонаправленого логічного зв'язку між сегментами *Рroperty for Rent* і *Owner.* Відповідно в описі сегменту *Property_for_Rent* згадується не тільки його фізичний батьківський сегмент (*Branch*), але і логічний батьківський сегмент (*Owner*) разом з назвою фізичної бази даних (*OwnerDB*), в якій знаходиться логічний батьківський сегмент. Крім опису логічного батьківського сегменту, потрібно описати тип вказівника, що використовується. Він визначається за допомогою символу *V* (віртуальний вказівник) і виразу *POINTER=*, який позначає збереження фізичної адреси логічного батька для надання прискореного доступу до

нього. Для встановлення відповідності між описами сегментів *Property_for_Rent* і *Owner* в описі сегменту *Owner* повинна міститися команда (*LCHILD*), яка означає, що сегмент *Owner* має логічний дочірній сегмент, описує його ім'я і ім'я фізичної бази даних, в якій зберігається цей логічний дочірній сегмент.

3. 3. 3. Мова управління даними DML для ієрархічної моделі

Мова DML включає команди мови *DL/I (Data Language with Indexes)*, які вбудовуються в базові мови програмування. В ролы базових мов програмування можуть використовуватися такі мови, як COBOL, PL/1 або мова асемблера. Управління даними здійснюється за допомогою записаних на базовій мові зовнішніх викликів відповідних команд мови DL/I. Команди мови DML послідовно обробляють один сегмент за іншим і не призначені для обробки груп сегментів. В цьому відношенні вони схожі на команди CODASYL-сумісної мови DML, в яких звичайні програмні конструкції повинні використовуватися разом з операторами CALL. Нижче наводяться невеликі приклади використання мови DL/I для створення деяких запитів, заснованих на структурі бази даних, описаної в лістингу 3.3. В цих прикладах оператори CALL показані не повністю - приведена тільки найістотніша їх частина. Проте команди мови DL/I показані повністю, з вказівкою (в дужках) скороченої форми їх запису.

Лістинг 3.3. Використання ієрархічної мови DML Ввести в базу даних інформацію про нове відділення компанії (Branch**)**

<створити запис в області введення-виведення, наприклад, так:>

MOVE 'B2' TO Bno

MOVE '56 Clover Dr' TO Street

MOVE 'EastEnd' TO Area

<і т.д.>

INSERT (ISRT) Branch

2.Вивести відомості про всі об'єкти нерухомості, з якими ведеться робота у відділенні компанії з номером B7.

GET UNIQUE (GU) Branch Bno = 'B7'

WHILE NOT FINISHED

GET NEXT WITHIN PARENT (GNP) Property for Rent

<виведення отриманих відомостей> *ENDLOOP*

3.Відновити коментарі з приводу проглядання об'єкту нерухомості з номером 'PG4', який був організований у відділенні компанії з номером 'B3'.

GET UNIQUE (GU) Branch Bno = 'B7'

GET HOLD UNIQUE (GHU) Property for Rent Pno = 'PG4'

Viewing Rno = 'CR56'

<при цьому на сегмент накладається блокування>

<аж до завершення цього оновления>

MOVE 'Riddled with dryrot' TO Comments

REPLACE(REPL)

3.4. Механізм звернень до фізичної пам’яті для мережевої моделі

Мережна модель більш симетрична, ніж ієрархічна модель. Проте процедури управління даними (наприклад, оновлення) значно складніші. Проблема полягає в наступному: завжди є дві стратегії для визначення місця одного екземпляра запису, перша починається з «власника» і проглядання його ланцюжка для вибору ланки, а інша починається з «підлеглої ланки» і проглядання його ланцюжка для вибору «власника». Як користувач може вирішити, яку стратегію прийняти? Вибір і тут має велике значення. Як в ієрархічних, так і мережних СУБД при описі даних звичайно указуються характеристики записів кожного типу, що сприяють більш ефективному розміщенню даних в зовнішній пам'яті і більш швидкому доступу до них. До таких характеристик відносяться: розміри полів запису (мінімальні, середні, максимальні), склад ключа, допустимий набір символів, інтервали значень і т.д.

Ієрархічні і мережні бази даних часто називають базами даних з навігацією. Ця назва відображає технологію доступу до даних, що використовується при написанні оброблювальних програм мовою маніпулювання даними. При цьому, очевидно, що доступ до даних по шляху, не передбаченому при створенні бази даних, може вимагати занадто багато часу. Підвищуючи ефективність доступу до даних, і скорочуючи таким чином час відповіді на запит, принцип навігації разом з цим підвищує і ступінь залежності програм і даних.

Наскільки заплутаними є схеми представлення ієрархічних і мережних баз даних, настільки і трудомістким є проектування конкретних прикладних систем на їх основі. Як показує досвід, тривалі терміни розробки прикладних систем нерідко призводять до того, що вони постійно знаходяться у стадії розробки і доробки. Причому, розробка нового часто пов'язана з потребою переробки вже зробленого. Багатолітній досвід для багатьох фірм свідчить про феномен замкнутого кола, який постійно існує при проектуванні, розробці і супроводі баз даних графового типу. Саме намагання якоюсь мірою подолати ці проблеми, з

якими постійно стикаються розробники і користувачі ієрархічних і мережних систем послужили стимулом до створення реляційної моделі даних і реляційних СУБД.

Якщо відштовхуватися від поняття «Мережева», то можна було б припустити, що походження цієї моделі засновано на використанні графів. Проте насправді не існує ніякого формального визначення мережної моделі даних від CODASYL, на яке можна було послатися. Першим, як це не дивно, намагався зробити Е.Кодд – автор альтернативної мережевій реляційної моделі в своїй відомій дискусії з Ч.Бахманом – найвідомішим адептом мережевої моделі. Робив це він з необхідності, бо захищаючи своє дітище він змушений був якось окреслити предмет спору. В даній роботі будемо притримуватись визначення наданого в монографії Коннолі і Беггс[]:

«Мережева модель даних - модель, що складається із записів, елементів даних і зв'язків типу «один до багатьох « (1:M), встановлених між записами».

В табл. 3.1 представлена термінологія, прийнята в середовищі CODASYL-систем.

Таблиця 3.1. Термінологія, прийнята в середовищі CODASYL-систем

Термін	Опис
Елемент даних	Іменоване поле (або підлегле поле)
Група	Іменована група підлеглих полів
Тип запису	Іменована група полів або підлеглих полів
Тип набору	Іменований зв'язок типу 1:M між двома типами записів
Схема	Визначення глобальної структури бази даних
Підлегла схема (підсхема)	Визначення зовнішнього/логічного представлення частини бази даних
Схема збереження/внутрішня схема	Визначення фізичного пристрою бази даних
Робоча область користувача (*User Work Area - UWA*)	Область пам'яті, контрольована виконуваною програмою, в якій виділяється пам'ять, необхідна для збереження всіх змінних, що використовуються. Обмін даними відбувається між базою даних і *UWA*-областями
Виконуваний процес	Екземпляр виконуваної програми. Виконуваний процес створюється при кожному запуску деякої прикладної програми
Мова *SDDL (Schema DDL)*	Декларативна мова, що використовується для визначення структури бази даних
Мова *SSDDL (Sub-schema DDL)*	Декларативне розширення мови програмування, на основі якого визначається призначене для користувача представлення бази даних

Відзначимо, що в цій таблиці відсутнє визначення мови, на якій проводиться найголовніше для користувача – маніпулювання даними. Стандарт CODASY DBTG їх явно

не фіксує, але розробники мережних моделей чудово знають ці засоби – це найчастіше *COBOL*, іноді *PL/1*, або навіть *FORTRAN*.

3.4.1.Програмна архітектура СУБД за версією комітету CODASYL DBTG

Комітет CODASYL DBTG запропонував свій варіант структури СУБД, який схемно представлений на Рис.3.8. З цієї схеми видно, що кінцеві користувачі здійснюють доступ до бази даних за допомогою прикладної програми написаній на базовій мові (зазвичай це COBOL). Для того, щоб здійснити доступ до інформації в базі даних, кожна прикладна програма повинна використовувати якусь підсхему, яка є обмеженим представленням структури всієї бази даних. Декілька програм можуть паралельно використовувати одну і ту ж підсхему, проте кожна програма може використовувати тільки одну підсхему. Більш того, підсхема визначається тільки на одній схемі, але вона може перекривати іншу підсхему. CODASYL-сумісна СУБД може підтримувати декілька різних баз даних, структура кожної з яких визначається власною схемою. Схема в термінології CODASYL має визначати глобальну структуру бази даних. Схема і підсхеми визначаються за допомогою різних DDL (мова SSDDL - Schema DDL є декларативним розширенням мови програмування.) Визначивши схему за допомогою мови SDDL, перш ніж вона зможе використовуватися СУБД, її необхідно транслювати з початкової форми в об'єктну. Після цього початкове визначення кожної підсхеми також повинне бути трансльовано з перетворенням в об'єктну форму - тільки після цього воно зможе використовуватися прикладною програмою.

Звичайно усередині прикладної програми має міститися тимчасове сховище, яке призначене або для зберігання даних, добутих з допоміжного джерела зберігання, або для розміщення даних перед їх збереженням. В програмі на мові COBOL це досягається за рахунок вказівки відповідних типів записів в розділі оперативного пристрою (working storage section) та розділу даних (data division). Тимчасове сховище необхідне і в тих випадках, коли програма працює з СУБД, оскільки тут також має місце обмін даними між прикладною програмою і базою даних. Використання підсхеми прикладною програмою викликає неявне оголошення елементів цієї підсхеми в тимчасовому сховищі. Простір, зарезервований для елементів підсхеми, називається робочою областю користувача (User Work Area - UWA), або UWA-областю (в термінології DBTG). В пропозиціях групи DBLTG (Database Language Task Group) цей термін опущений, хоча він як і раніше використовується на практиці.

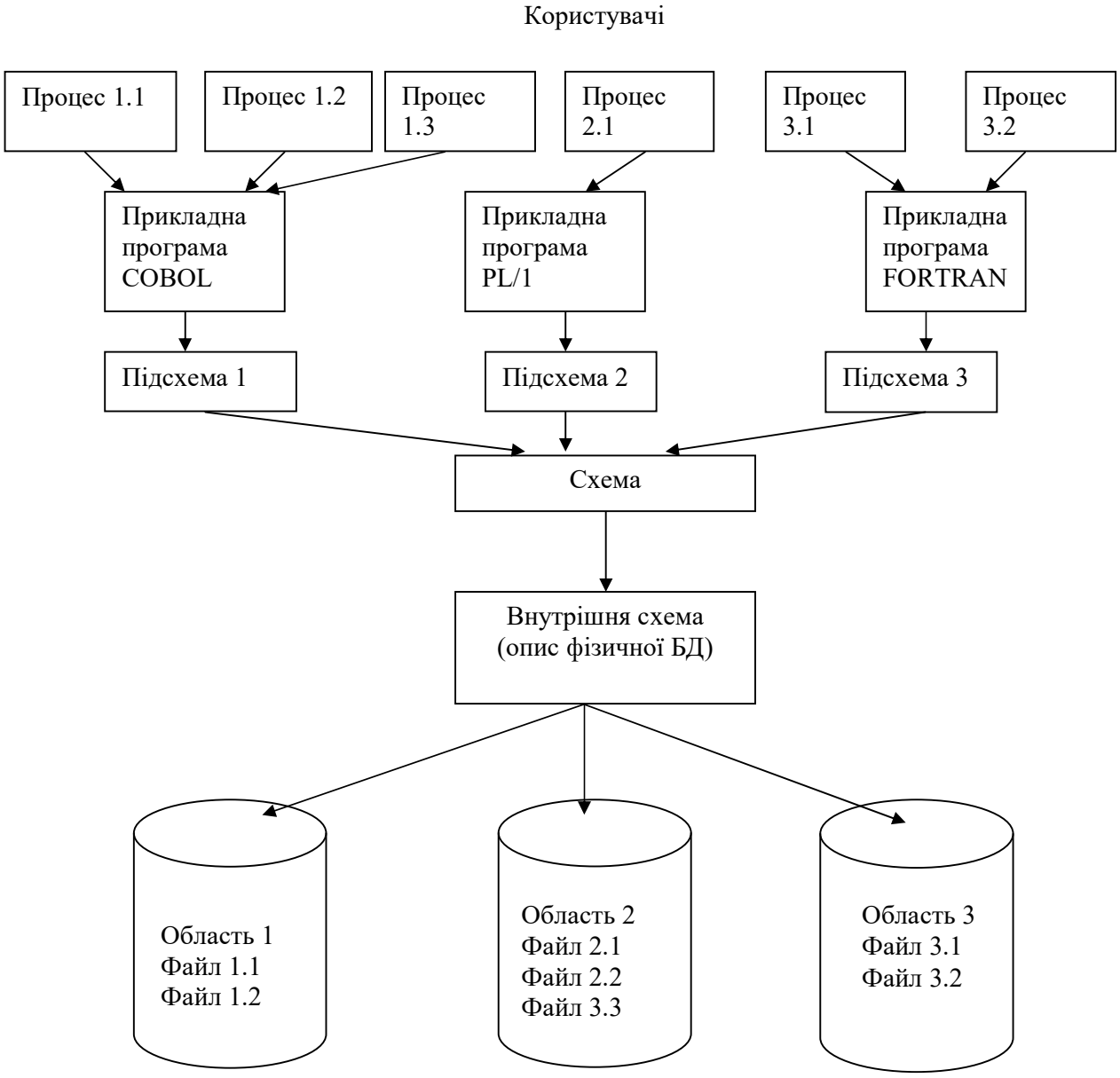


Рис.3.8. Загальна архітектура CODASYL-сумісної СУБД

3.4.2. Мови опису схеми і підсхеми

Існують дві мови опису даних (DDL): один - для схеми та інший для підсхеми (представлення). Вони використовуються для опису фундаментальних структур CODASYL-сумісних СУБД. Тут ми розглянемо тільки мову схеми DDL - Schema DDL (SDDL) на прикладі.

Таблиця 3.2. Екземпляр Schema DDL

	Branch	*Has*	*Staff*
1.	*B3*		*SG5, SG14, SG37*

2.	*B5*		*SL21, SL41*

Таблиця 3.3. Екземпляр Schema DDL

	Branch	*IsAllocated*	*Property_for_Rent*
1.	*B3*		*PG4, PG16, PG21, PG36*

В перекладі з *SDDL* на українську у виразі Таблиці 3.2. стверджується, що орендна фірма має два регіональні відділення *B3* та *B5*, в яких має персонал з табельними номерами *SG5, SG14, SG37* та *SL21, SL41* відповідно.

У Таблиці 3.3.вираз *SDDL* стверджує, що у підпорядкуванні відділення *B3* знаходяться об'єкти нерухомості за кодами *PG4*, *PG16*, *PG21*, *PG36*

Приклад опису взаємозв'язків між орендарем та можливою орендною нерухомістю на мові SDDL показано в Таблиці 3.3.

Таблиця 3.3. Опис взаємозв'язків між орендарем та можливою орендною нерухомістю

	Property_for_Rent	*IsViewedBy*	*Renter*
1	*PG4*		*GR56, GR71*
2	*PG21*		*GR71*
3			*GR41*
4	*PG16, PG36*		

У цьому виразі фіксується, що об'єкт нерухомості з кодом PG4 був оглянутий клієнтами з кодами GR56, GR71; об'єкт нерухомості – клієнтом з кодом GR71; клієнт з кодом GR41 ще не оглянув жодного об'єкта, а об'єкт з кодом GR36 не був оглянутий жодним клієнтом

3.4.3.Мова SDDL

В лістингу 3.4. показаний фрагмент визначення схеми бази даних для учбового проекту , яка складається з типів записів *Staff* і *Property_for Rent,* а також з типу набору *Supervises*, що об'єднує цих два типи записів. Мова визначення схеми SDDL визначає концептуальну або глобальну структуру бази даних (див. Рис.В.1), в якій визначені всі типи записів і типи наборів. В CODASYL-сумісній СУБД може бути декілька визначень схеми. Будь-яке з них включає перераховані нижче розділи.

- *Опис схеми***.** Це початковий розділ, в якому вказана назва схеми.
- *Опис області.* В цьому розділі указуються області фізичного зберігання і може міститися інша інформація про характеристики фізичного пристрою зберігання.
- *Опис записів.* В цьому розділі дається повний опис структури кожного запису разом зі всіма елементами даних, а також можуть включатися детальні відомості про розташування записів і методи їх зберігання**.**
- *Опис набору.* В цьому розділі перераховуються всі набори з вказівкою їх типів

записів-власників і типів записів-членів, а також приводяться інші відомості про набори (наприклад, про впорядкування**).**

Лістинг. 3.4. Фрагмент опису схеми для орендарів об'єктів нерухомості

```
SCHEMA NAME IS Estate-agent
AREA NAME IS Estate-Area
RECORD NAME IS Property_for_Rent
LOCATION MODE IS VIA Manages
WITHIN Estate-Area;
02 Pno;     PICTURE IS XX999.
02 Street;   PICTURE IS A(15).
02 Area;    PICTURE IS X(15).
02 City;    PICTURE IS X(15).
02 Postcode; PICTURE IS A(8).
02       Type;   PICTURE IS X(8).
02       Rooms;PICTURE IS 99.
02       Rent;   PICTURE IS 9999.99.
02       Ono;    PICTURE IS XX999.
RECORD NAME IS Staff
LOCATION MODE IS CALC USING Sno
DUPLICATES ARE NOT ALLOWED;
WITHIN Estate-Area;
02 Sno;     PICTURE IS XX999.
02 Fname;    PICTURE IS X(15).
02 Lname;    PICTURE IS X(15).
02 Address;  PICTURE IS A(50).
02 Tel-no;   PICTURE IS X(13).
02 Position; PICTURE IS A(9).
02 Sex;     PICTURE IS X.
02 STYPENDary;   PICTURE IS 999999.99.
02 NI-No;    PICTURE IS XX9(6)X.
SET NAME IS Supervises;
OWNER IS Staff;
ORDER IS SORTED DEFINED KEYS
DUPLICATES ARE NOT ALLOWED
```

MEMBER IS Property for_Rent

KEY IS ASCENDING PIio

NULL IS NOT ALLOWED

INSERTION IS AUTOMATIC RETENTION IS MANDATORY

SET SELECTION IS THRU Supervises OWNER IDENTIFIED CALC-KEY

Історично склалося так, що *CODASYL*-сумісна мова *DML* вважалася розширенням мови програмування, яка використовувалася для роботи з підсхемою, визначеною на основі її власної мови підсхеми *SSDDL*. Таким чином, ця мова програмування є базовою мовою. В ідеалі, будь-яка прикладна програма, створена за допомогою базової мови і що містить команди *DML*, повинна компілюватися модифікованим компілятором мови, проте на практиці препроцесор мови перетворить початковий код в певний вигляд, зрозумілий компілятору мови. Підсхема, яка визначається і зберігається окремо від прикладної програми, підключається до прикладної програми на мові *COBOL* за допомогою відповідної команди в розділі даних (*Data Division*). Це дозволяє автоматично визначити робочий простір в пам'яті для підсхеми. Команди мови *DML* оперують з кожним записом окремо, а не відразу з цілим набором. Отже, для організації роботи з багатьма записами необхідно використовувати звичайні програмні конструкції: умовні вирази і цикли, а це означає навігаційну природу обробки даних в рамках мережевої моделі.

Розглянемо приклад використання мови управління даними DML, ілюстровані на діаграмі схеми, показаної на Рис.3.9.

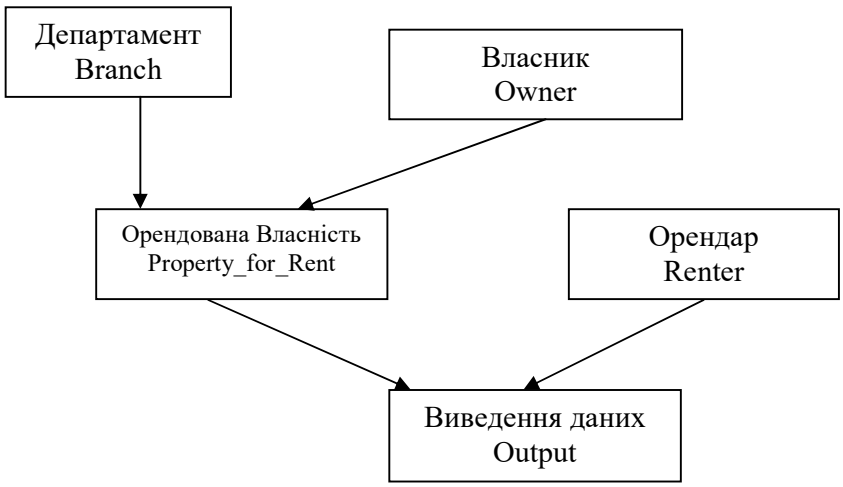


Рис.3.9. Діаграма схеми, що ілюструє відношення між даними в прикладах запитів як мережеву модель

Приклад використання CODASYL-сумісної мови DML

Лістинг 3.4.

1.Ввести в базу даних інформацію про нове відділення компанії (Branch).

MOVE 'B2' TO Bno

MOVE '56 Clover Dr' TO Street

MOVE 'EastEnd' TO Area

<и т.д.>

STORE Branch

<перевірка успішності завершення цієї операції>

2.Вивести відомості про всі арендовані об'єкти нерухомості, з якими ведеться робота у відділенні компанії під номером B7.

MOVE 'B7' TO Bno

FIND ANY Branch USING Bno

<перевірка успішності завершення цієї операції>

GET

<виведення отриманих даних>

FIND FIRST Property_for_Rent WITHIN ISTYPENDlocated

WHILE DB-STATOS=0

GET

<виведення отриманих даних>

FIND NEXT Property_for_Rent WITHIN ISTYPENDlocated

ENDLOOP

3.Вивести відомості про всі об'єкти нерухомості, що належать власнику з номером CO87, разом з відомостями про те, хто і коли їх проглядав.

MOVE 'C087' TO Ono

FIND ANY P-owner USING Ono

<перевірка успішності завершення цієї операції>

GET

<виведення отриманих даних>

FIND FIRST Property for_Rent WITHIN Owns

WHILE DB-STATUS=0

GET

<виведення отриманих даних>

FIND FIRST Viewing WITHIN Takes

WHILE DB-STATUS=0

GET

FIND OWNER WITHIN Attends

GET

<виведення отриманих даних>

FIND NEXT Viewing WITHIN Takes

ENDLOOP

FIND NEXT Property_for_Rent WITHIN Owns

ENDLOOP

3.5. Механізм звернень до фізичної пам’яті для реляційної моделі

Чим же принципово відрізняються реляційні моделі від мережних і ієрархічних? На це питання можна відповісти стисло таким чином: ієрархічні і мережні моделі даних мають зв'язок по структурі даних, а реляційні - мають зв'язок по значенню даних. Проектування баз даних з використанням навігаційних моделей даних традиційно вважається дуже важкою задачею. Реляційна технологія значно спрощує цю справу (на рівні моделей). Але разом з цим виникають нові проблеми:

- потреби в спрощенні призначеного для користувача інтерфейсу і підвищенні його ефективності;
- «безшовна стиковка» з іншими інформаційними системами на рівні даних і інтерфейсів;
- відповідність різним міжнародним стандартам.

Реляційна технологія, завдяки розділенню логічного і фізичного рівнів системи, спрощує процес відображення «рівня реального світу» в структуру даних, яку система може прямо підтримувати. Оскільки реляційна структура сама по собі концептуально проста, вона дозволяє швидко і ефективно реалізовувати невеликі бази даних на персональних комп'ютерах. Але проблема розробки програмних механізмів реалізацій запитів до реляційної бази даних виявилась набагато складнішою. Причина цих складнощів корініться в семантичному розриві між декларативною логікою запитів і реальною організацією розміщення даних на зовнішніх носіях, в основному на дисках. Не дивно, що за ці більше ніж 30 років так і не склалися якісь загальновизнані стандарти, і особливості реалізації цієї частини СУБД є ноу-хау виробника. Тим не менше, розглянемо принципові підходи до реалізації реляційних запитів. Ефективні методи обробки довільних запитів є вирішальною передумовою успіху узагальнених систем управління базами даних. Цей набір методів звівся до оптимізації шляхів доступу і зберігання даних на рівні файлових систем.

3.5.1. Оптимізація, призначена для користувача

Загальна вартість інформаційної системи складається з вартості СУБД і витрат на зусилля користувачів при роботі з системою. Якщо мова запитів має задані функціональні можливості, а мінімізація часу відгуку є метою системи виконання запитів, то оптимізація запитів може вважатися однією з підзадач оптимізації, призначеної для користувача .

Структури файлів. Алгоритм оптимізації запитів має проводити вибір між множиною шляхів доступу для виконання запиту. Після перетворення запит має бути відображений в послідовність операцій, що повертають необхідні дані. Для ряду задач оптимізації запитів потрібне трактування унаслідок більш високої складності запитів.

3.5.2. Проблема оптимізації реляційних запитів

Загальна задача оптимізація процедур обчислення запитів є, взагалі кажучи, обчислювально дуже важкою, яка ще більше ускладнюється відсутністю точної статистичної інформації про базу даних і тому наразі невирішеною. Як визнають дослідники в області обчислення запитів в алгоритмах обчислення запитів доводиться великою мірою покладатися на евристині підходи. Тому термін «оптимізація запитів» тут вживається для позначення стратегій, направлених на підвищення ефективності процедур обчислення запитів. Далі ми формулюємо мету оптимізації запитів і представляємо загальну процедуру, розроблену для структуризації процесу пошуку рішення.

Запит - це вираз на мові, що описує дані, які підлягають вибірці з бази даних. Запити використовуються в різних програмних реалізаціях. Найбільш очевидним є прикладна програма, до якої надходять безпосередні запити кінцевого користувача, що потребує інформацію про структуру або зміст бази даних. Якби потреби користувачів обмежувались набором стандартних запитів, то їх можна було б оптимізувати вручну. Але проблема в тому, що вимагається ефективно реалізувати всі дозволені мовою запити без обмежень. Тому система автоматичної оптимізації запитів стає необхідною.

Далі розглянемо мету оптимізації. Економічний принцип вимагає, щоб процедури оптимізації намагалися або максимізувати пропускну спроможність при заданому числі ресурсів, або мінімізувати споживання ресурсів при даній пропускній спроможності. Оптимізація запитів направлена на мінімізацію часу відгуку для заданого запиту і суміші типів запитів в даному системному середовищі. Ця загальна мета допускає ряд різних цільових функцій. Час відгуку є розумною метою тільки при припущенні, що час користувача є найважливішим критичним ресурсом. В зворотньому випадку можна прагнути безпосередньої мінімізації витратності споживання технічних ресурсів. На щастя, обидва критерії є у великій мірі взаємнодоповнюючими; при виникненні конфліктів вони розв’язуються шляхом призначення обмежень на доступні технічні ресурси (наприклад, на розмір буферного простору в основній пам'яті). Щоб отримати коректне порівняння ефективності, функціональні можливості порівнюваних систем виконання запитів повинні бути схожими. Вимога «реляційної повноти», сформульована Коддом [Codd 1972] фактично стала стандартом. Методи оптимізації запитів розглядаються як інструменти поліпшення

реалізації запитів на реляційно повній мові з мінімальною витратністю виконання або часу відгуку.

Загальна витратність, що підлягає мінімізації, складається з наступних компонентів:

- Витратність комунікацій: Витратність передачі даних з місця, в якому вони зберігаються, в місця, де виконуються обчислення і представляються результати. Ця витратність складається з витратності комунікаційного каналу, яка звичайно зв'язана з часом, протягом якого канал відкритий, і вартістю затримок в обробці, обумовленою передачею. Останній компонент, більш важливий для оптимізації запитів, часто вважається лінійною функцією від об'єму передаваних даних.
- Витратність доступу до вторинної пам'яті: витратність (або час) завантаження сторінок даних з вторинної пам'яті в основну пам'ять. На цю витратність вибираних даних впливає: кластеризація даних на фізичних сторінках, розмір доступного буферного простору, швидкість пристроїв, що використовуються.
- Витратність зберігання: витратність заняття вторинної пам'яті і буферів основної пам'яті. Витратність зберігання враховується тільки у тому випадку, коли пам'ять стає вузьким місцем системи, і розмір необхідної пам'яті може змінюватися від запиту до запиту.
- Витратність обчислень: час використання центрального процесора.

Виходить, що задача в загальному випадку має багатокритеріальний характер. Тому алгоритми оптимізації запитів залежать від співвідношення між цими компонентами витратності. В територіально розподіленій СУБД з відносно повільними комунікаційними каналами переважає витратність комунікаційних затримок, а інші чинники істотні тільки для локальної підоптимізації. В централізованих системах домінує час доступу до вторинної пам'яті. В локально розподілених СУБД всі чинники мають близьку вагу, що приводить до дуже складних функцій оцінки і процедур оптимізації.

Беручи до уваги багатокритеріальність, розглянемо випадок централізованих баз даних, де витратність комунікацій не враховується. Для оптимізації одиночних запитів витратність зберігання також вважається не першорядною. Залишаються витратність доступу до вторинної пам'яті (звичайно, що виміряється кількістю звернень до сторінок). Що стосується витратності використання ЦП, то цей критерій не беруть до уваги через те, що обчислювальні швидкості сучасних процесорів на декілька порядків перевищують можливості периферійних пристроїв. В основі більшості методів, розроблених для скорочення цієї витратності, лежить ряд загальних ідей:

- уникати дублювання зусиль;
- використовувати стандартизовані компоненти;
- заглядати вперед, щоб уникати зайвих операцій;

- вибирати найдешевші способи виконання елементарних операцій;
- вибудовувати їх послідовність оптимальним чином.

Наступний простий приклад показує, що можна чекати від оптимізації запитів.

Розглянемо реляційну схему бази даних, яка описує учбовий центр, що пропонує курси з інформаційних технологій підрозділам різним комерційних фірм.

employees (enr, ename, status, city)

papers (enr, title, year)

departments (dname, city, street address)

courses (cnr, cname, abstract)

lectures (cnr, dname, enr, daytime)

Ключові атрибути підкреслені; задана комбінація значень ключових атрибутів унікально ідентифікує елемент відношення. Нехай учбові корпуси розташовані в різних містах, а користувача цікавлять «назви відділів, розташованих в Києві і пропонують курси по управлінню базами даних».

Є багато можливих стратегій реалізації цього запиту, три з яких порівнюються по відношенню до наступних припущень щодо реальних значень даних. Зауважимо, що детальні дані, що використовуються в що приводяться нижче обчисленнях, звичайно недоступні оптимізатору запитів, а повинні оцінюватися. Розглянемо різні варіанти наповнення нашої бази даних:

- 100 відділів, 5 з яких розміщуються в Києві. У фізичному блоці може поміститися 5 записів про відділи або 50 значень dname4;
- 500 курсів, 20 з яких присвячені управлінню базами даних. У фізичному блоці поміщається 10 записів;
- 2000 лекцій, три сотні з них - про управління базами даних, 100 проходять у відділах в Києві і 20 (з трьох відділів) задовольняють обом умовам. У фізичний блок поміщаються 10 записів.

Нехай час сортування складає $N * log_2 N$, де N - розмір файлу в блоках, і що є буфер з одного блоку для кожного відношення. Нарешті, всі відносини фізично впорядковані за збільшенням значень ключа.

Перша представлена тут стратегія виходить з прямого підходу трансляції виразу реляційного числення в операції алгебри [Codd 1972]. Разом з кожним кроком стратегії приводиться число звернень до вторинної пам'яті, потрібних для читання (r) і запису (w).

Стратегія 1

1. Сформувати декартовий добуток відношень «courses», «lectures» і «departments» (r: 200000)

2. Залишити колонку dname з тих записів «departments», для яких значення колонок cnr в «courses» і «lectures» співпадають, і значення колонок dname з «lectures» і «departments» співпадають, і cname = 'database management' and city = 'Київ'.

(w: 1)

разом: приблизно 200000 звернень.

Виключно висока витратність цієї стратегії походить з того факту, що вона генерує проміжний результат, дуже невелика частина якого дійсно істотна для подальшої обробки. Ефективність може бути істотно підвищена шляхом розгляду тільки тих комбінацій елементів різних відношень, які мають однакові значення спільних атрибутів. Шляхом використання існуючих порядків сортування, такі комбінації можуть формуватися *«злиттям»* відношень, які беруть в цьому участь. Ця операція називається *«з'єднанням».*

Стратегія 2

1. Виконати злиття відношень *«courses»* і «*lectures*» *(r: 50 + 200; w: 400)*
2. Відсортувати результат по *dnames (r + w: 400*1og(2)400)*
3. Виконати злиття результату з відношенням *«departments» (r: 400 + 20; w: 400 + 400)*
4. Вибрати комбінації з *cname = 'database management' and city = 'Київ' (r: 800)*
5. Залишити тільки колонку *dname. (w: 1)*

*Р*азом*:* Приблизно 6000 звернень.

Витратність запиту можна ще скоротити шляхом виконання обмежень на основі значень якомога раніше і зменшення за рахунок цього витрат на сортування і злиття проміжних результатів.

Стратегія 3

1.Виконати злиття відношень *«courses»* і «*lectures*» (r: 50 + 200)

2.Залишити тільки dnames з комбінацій cname = *'database management'*

(w: 2)

3.Відсортувати згенерований список *dname(r + w: 2)*

4.Виконати злиття результату з відношенням «*departments*» *(r: 2 + 20)*

5.Залишити тільки ті dnames, для яких *city = ' Київ' (w: 1)*

Разом: 277 звернень.

Таким чином, було досягнуте скорочення витратності приблизно в 700 разів. Для великих баз даних і більш складних запитів оптимізовані методи можуть привести до ще більших скорочень.

На цьому прикладі можна помітити, що виконання операції декартового добутку або з'єднання таблиць пов'язано з більшими витратами, ніж виконання операції вибірки. Тому основна перевага третього варіанту обробки запиту полягає в тому, що розміри

оброблюваних таблиць були істотно скорочені до виконання операції з'єднання. Звідси можна зробити попередній висновок, що при оптимізації запитів унарні операції вибірки і проекції доцільно виконувати якомога раніше, оскільки це дозволить зменшити розміри наборів даних, що беруть участь в бінарних операціях.

3.5.2.1. Низхідний підхід до оптимізації запитів

Далі розглянемо загальну процедуру обчислення, яка є оптимізацією запитів на базі низхідного підходу.

Крок 1. Знайти внутрішнє представлення запитів, в яке можуть легко відображатися запити користувачів, що залишає системі всі необхідні ступені свободи для оптимізації виконання запитів.

Крок 2. Застосувати логічні перетворення до представлення запиту, які б стандартизували та спрощували запит для уникнення дублювання зусиль та покращення виконання запиту і створення можливості застосування процедур.

Крок 3. Відобразити перетворений запитів в можливу послідовність елементарних операцій, для яких відома добра реалізація і відповідні оцінки витратності. В результаті цього кроку з'являється набір можливих «планів доступу».

Крок 4. Обчислити загальну витратність кожного плану доступу, вибрати найдешевший план і виконати його.

Перші два кроки цієї процедури є у великій мірі незалежними і тому часто можуть бути виконаний під час компіляції. Якість кроків 3 і 4, тобто кількість планів доступу, що генеруються, і оптимальність алгоритму вибору сильно залежить від знання значень в базі даних.

Контрольні запитання та вправи

1. Як принципи організації збереження даних ієрархічній моделі даних?
2. На чому базуються моделювання зв’язків M:N в в ієрархічній моделі даних?
3. Як реалізується мова DML для ієрархічної моделі даних?
4. В чому причина складності розробки програмних механізмів реалізацій запитів до мережевих бази даних ?
5. В чому необхідність оптимізації виконання запитів до мережевих баз даних?
6. Проітерпретуйте лістинг 1з точки зору опису даних CODASYL.
7. Проітерпретуйте лістинг 3.2 з точки зору маніпулювання даними CODASYL.
8. В чому полягає мета оптимізації запитів до реляційних баз даних?
9. На чому базуються еврістичні підходи по оптимізації запитів до реляційних баз даних?

10. В чому полягає стратегія низхідного підходу до оптимізації запитів до реляційних баз даних?

Розділ 4. Загальна характеристика реляційної моделі даних

З огляду на важливість розлянута вище реляційна модель баз даних заслуговує на те, щоб розглянути її значно детальніше, ніж це було зроблено в попередньому розділі. Основи реляційної моделі даних були вперше викладені у статті співробітника IBM Resourch Едгара Кодда [] в 1970 р. «A Relational Model of Data for Large Shared Data Banks».

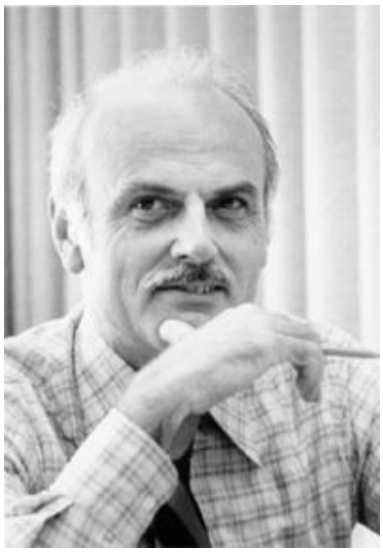


Рис.4.1. Едгар Кодд

Сам Едгар Кодд, маючи за плечима не тільки докторський ступінь, але і бойовий досвід у другій світовій війні як пілот ВПС Великобританії, був з числа тих нових кадрів, які потрапили до IBM завдяки ініціативі Томаса Уотсона – молодшого. Подальша доля його складалася досить нелегко, але зрештою, працюючи в одному з дослідницьких проектів IBM він сформулював свою знамениту модель. Ця робота стала джерелом для незліченної кількості статей і книг, в яких реляційна модель отримала подальший розвиток.

4.1 Сучасне трактування реляційної моделі баз даних

Одним з найбільш популярних трактувань реляційної моделі даних є формулювання К.Дейта [11]. Згідно Дейту, реляційна модель складається з трьох частин:

- структурної частини.
- цілісної частини.
- маніпуляційної частини.

Структурна частина описує, які об'єкти розглядаються реляційною моделлю. Постулюється, що єдиною структурою даних, що використовується в реляційній моделі, є нормалізовані n-арні відношення.

Цілісна частина описує обмеження спеціального вигляду, які повинні виконуватися для будь-яких відношень в будь-яких реляційних базах даних. Це ***цілісність сутностей*** і ***цілісність зовнішніх ключів***.

Маніпуляційна частина описує два еквівалентні способи маніпулювання реляційними даними - ***реляційну алгебру*** і ***реляційне числення***.

В даному розділі розглядається структурна частина реляційної моделі.

4.1.1. Типи даних

Будь-які дані, що використовуються в програмуванні, мають свої типи даних.

Реляційна модель вимагає, щоб типи даних, що використовуються, були *простими*.

Для уточнення цього твердження розглянемо, які взагалі типи даних звичайно розглядаються в програмуванні. Як правило, типи даних діляться на три групи:

- прості типи даних.
- структуровані типи даних.
- посилальні типи даних.

Прості, або атомарні, типи даних не мають внутрішньої структури. Дані такого типу називають ***скалярами***. До простих типів даних відносяться наступні типи:

- логічний.
- символьний.
- чисельний.

Різні мови програмування можуть розширювати і уточнювати цей список, додаючи такі типи як:

- цілий.
- дійсний.
- рядковий.
- дата.
- час.
- грошовий.
- перелічуваний.
- Інтервальний.

Звичайно, поняття атомарності досить відносне. Так, рядковий тип даних можна розглядати як одновимірний масив символів, а цілий тип даних - як набір бітів. Важливо лише те, що при переході на такий низький рівень втрачається семантика (зміст) даних. Якщо рядок, що виражає, наприклад, прізвище співробітника, розкласти в масив символів, то при цьому втрачається значення такого рядка як єдиного цілого.

Що стосується складних типів даних, то реляційна модель їх теж допускає з досить цікавим підтекстом: власне, для реляційної моделі даних тип даних, що використовуються, не важливий. Вимога про те, щоб тип даних був простим, потрібно розуміти так, що в реляційних операціях не повинна враховуватися внутрішня структура даних. Натомість, повинні бути описаний дії, які можна проводити з даними як з єдиним цілим, наприклад, дані числового типу можна складати, для рядків можлива операція конкатенації і т.д.

З цієї точки зору, якщо розглядати масив, наприклад, як єдине ціле і не використовувати поелементних операцій, то масив можна вважати простим типом даних. Більш того, можна створити свій, як завгодно складний тип даних, описати можливі дії з цим типом даних, і, якщо в операціях не потрібне знання внутрішньої структури даних, то такий тип даних також буде простим з погляду реляційної теорії.

Як і графових моделях баз даних, основними поняттями, що використовуються, тут також є поле, запис, файл. Структура запису визначає структуру таблиці, що містить екземпляри відповідного запису. Колонки таблиці є іменами полів запису, рядки таблиці є екземплярами запису. Таким чином, поняття «таблиця» тут відповідає поняттю «файл» моделі даних. Групове відношення може представлятися двома способами. При першому способі в таблиці, відповідні групам - членам відношення, додаються колонки ключових полів (атрибутів) іншого члена відношення (зв'язок описується через ключові атрибути). При другому способі групове відношення визначається як додаткова група (додаткова таблиця). Колонкими цієї додаткової таблиці є ключі груп–членів відношення. Таким чином, при будь-якому способі відповідна модель даних є сукупністю структур таблиць.

Для формального опису таблиці використовується теоретико-множинне поняття відношення. Список назв колонок таблиці (імен полів запису, відповідних атрибутам), називають схемою відношення і позначають R (A_1, A_2, A_n). Сукупність схем відношень, що використовуються для представлення концептуальної моделі, називається схемою реляційної бази даних, а поточні значення відповідних відношень – реляційною базою даних.

Незабаром після появи ідея (і теорія) реляційних баз даних стала популярна серед розробників СУБД. Проте зробити реальну реляційну СУБД виявилося непросто. Склалася неоднозначна ситуація, коли після певних непринципових удосконалень одні розробники почали рекламувати свої розробки як реляційні, а інші - відмовлятися від створення реляційних СУБД через складність задачі. Як те, так і інше дискредитувало ідею реляційної СУБД.

4.1.2. Правила Кодда

Для того, щоб внести ясність в оцінку розробок одних фірм і більш точно сформулювати мету, до якої розробникам потрібно прагнути, Е. Кодд в кінці 70-х рр.

опублікував свої 12 правил відповідності довільної СУБД реляційній моделі [2], [3], доповнивши основні поняття реляційних баз даних визначеннями, важливими для практики. Нижче наводяться ці правила разом з доповнюючим їх загальним положенням, а саме:

0. Основне правило. Система, яка рекламується або проголошується постачальником як реляційна СУБД, повинна управляти базами даних виключно способами, відповідними реляційній моделі.

1. Інформаційне правило. Вся інформація, що зберігається в реляційній базі даних, повинна бути явно, на логічному рівні, представлена єдиним чином - у вигляді значень в R-таблицях.

2. Правило гарантованого логічного доступу. До кожного типу даних в реляційній базі атомарного значення повинен бути гарантований доступ за допомогою вказання імені R-таблиці, значення первинного ключа і імені колонки.

3. Правило наявності значення (missing information). В повністю реляційній СУБД повинні бути спеціальні індикатори (відмінні від порожнього символьного рядка або рядка з одних пропусків і відмінні від нуля або якогось іншого числового значення) для виразу (на логічному рівні, систематично і незалежно від типу даних) того факту, що значення відсутнє щонайменше із двох різних причин: його дійсно немає або воно непридатне на даній позиції. СУБД повинна не тільки відображати цей факт, але і поширювати на такі індикатори свої функції маніпулювання даними незалежно від типу даних.

4. Правило динамічного діалогового реляційного каталога. Опис бази даних виглядає логічно як звичайні дані, так що авторизовані користувачі і прикладні програми можуть вживати для роботи з цим описом ту ж реляційну мову, що і при роботі із звичайними даними.

5. *Правило повноти мови роботи з даними*. Скільки б не підтримувалося мов в СУБД і режимів роботи з даними, має бути принаймні одна мова, що виражається у вигляді командних рядків в якомусь зручному синтаксисі, який би дозволяв формулювати:

- визначення даних ;
- визначення правил цілісності ;
- маніпулювання даними (в діалозі і з програми);
- визначення таблиць (у тому числі можливості їх модифікації), що виводяться ;
- визначення правил авторизації ;
- межі транзакцій.

6. *Правило модифікації таблиць-представлень.* В СУБД повинен існувати коректний алгоритм, який дозволяє автоматично для кожної таблиці-представлення визначати під час її створення, чи може вона використовуватися для вставки і видалення рядків і які із колонок

допускають модифікацію, і що заносить отриману таким чином інформацію в системний каталог.

7. Правило множинності операцій. Можливість оперування базовими таблицями, або таблицями, що виводяться розповсюджується повністю не тільки на видачу інформації з БД, але і на вставку, модифікацію і видалення даних.

8. Правило фізичної незалежності. Діалогові оператори і прикладні програми на логічному рівні не повинні залежати від будь-яких змін у внутрішньому зберіганні даних або в методах доступу СУБД.

9. Правило логічної незалежності. Діалогові оператори і прикладні програми на логічному рівні не повинні залежати від таких змін в базових таблицях, які зберігають інформацію і теоретично допускають незмінність цих операторів і програм.

10. Правило збереження цілісності. Діалогові оператори і прикладні програми не повинні змінюватися при зміні правил цілісності в БД (що визначаються мовою роботи з даними і зберігаються в системному каталозі).

11. Правило незалежності від розподілу. Діалогові оператори і прикладні програми на логічному рівні не повинні залежати від таких змін як фізичний розподіл даних (якщо спочатку СУБД працювала з нерозподіленими даними) або перерозподілу (якщо СУБД дійсно була розподілена).

12. Правило непорушення реляційної мови. Якщо в реляційній СУБД є мова низького рівня (для роботи з окремими рядками), вона не повинна дозволяти порушувати або «обходити» правила, сформульовані на мові високого рівня (множинному) і занесені в системний каталог.

Важливість правил Кодда в тому, що, будучи сформульовані не менш ніж 30 років тому, вони ніким не оскаржені, не доповнювалися і до цього часу є єдиними правилами такого роду. Не зважаючи на те, що не всі вони рівноцінні, ці правила протягом довгого часу визначають точку відліку для розробників і критерій відповідності для розробників і користувачів.

4.2. Особливості реляційної моделі баз даних

На даний момент теорія реляційних баз даних настільки розвинулась, що більш-менш повний її розгляд потребує окремої солідної монографії. Тому давайте розглянемо коротко основні властивості реляційної моделі, виходячи з критеріїв важливості цієї моделі для електронної обробки даних. Для бажаючих поглибити свою ерудицію в цій важливій сфері можна порекомендувати []-[].

Фактично всі особливості реляційної моделі баз даних вже вказані в попередньому параграфі, проте необхідно їх дещо деталізувати. В першу чергу треба відзначити рідкісну

рису реляційної моделі – теоретичні, формальні положення виступають тут як норми прямої дії, а їх порушення завжди веде до погіршення якості реальних баз даних. Тому, розглядаючи предметно реляційну модель баз даних, не можна уникнути розгляду цього аспекту.

4.2.1.Формалізований опис відношень і схеми відношень

Як вже відзначалося, реляційна модель описує представлення даних у вигляді двовимірної таблиці, що називається відношенням. Найменуваннями колонок цієї таблиці служать імена атрибутів. Розглянемо формалізований опис відповідних понять.

Схемою відношень $R(A_1, A_2, ..., A_n)$ називається кінечна множина імен атрибутів $\{A_1, A_2 ., A_n\}$, де n – арність схеми відношення. Поняттю схема відношення відповідає опис структури двовимірної таблиці (імена колонок). Кожному імені атрибута A_i відповідає допустима множина значень, які може приймати атрибут A_i. Ця множина значень D_i називається доменом атрибута A_i, $i = 1, n$.

За визначенням, домени є непорожніми кінечними або перелічуваними множинами. Уточнимо, що в теорії реляційних баз даних домен розглядається як множина значень одного простого типу даних. Поняттю домена D_i відповідає множина значень, що стоять в колонках A_i даної таблиці.

Нехай $D = D_1 \cup D_2 \cup ... \cup D_n$. Відношенням r зі схемою R називається кінечна множина відображень $\{t_1, t_2., t_p\}$ з множини $R: \{A1, A2, ..., An\}$ в множину $D:\{ D_1 \cup D_2 \cup ... \cup D_n\}$, таких, що $t_k(A_i) \in D_i$, $k = 1, p$; i=□1, n. Таким чином, достатньо визначити значення кортежу на множині , щоб однозначно його ідентифікувати. Сукупність схем відношень, що використовуються для представлення концептуальної моделі, називається схемою реляційної бази даних (реляційною моделлю даних). Поточні значення відповідних відношень називаються реляційною базою даних. Відзначимо наступні властивості відношення:

1. Порядок розгляду атрибутів в схемі відношень (відношенні) не має значення, оскільки для посилання на значення атрибута в кортежі відношення завжди використовується ім'я атрибута.

2. Значення всіх атрибутів є атомарними (неподільними). Це витікає з визначення домена як множина значень простого типу даних, тобто серед значень домена не можуть міститися множини.

3. Порядок розгляду кортежів у відношенні не має значення, оскільки відношення є множина кортежів, а елементи множини, за визначенням теорії множин, неупорядковані.

4.2.2.Маніпулювання даними в реляційній моделі

Для маніпулювання даними в реляційній моделі використовуються дві формальні методології:

- реляційна алгебра, що базується на теорії множин;
- реляційне числення, що базується на численні предикатів першого порядку.

Операції, що реалізовувуються за допомогою вказаних методологій, мають важливу властивість: вони замкнуті на множині відношень. Це означає, що вирази реляційної алгебри і формули реляційного числення визначаються над відношеннями реляційних БД і результатом обчислення також є відношення. В результаті будь-який вираз або формула можуть інтерпретуватися як відношення, що дозволяє використовувати їх в інших виразах або формулах. Як ми побачимо, алгебра і числення мають велику потужність, бо дуже складні запити до бази даних можуть бути сформульовані за допомогою одного виразу реляційної алгебри або однієї формули реляційного числення. Саме з цієї причини ці механізми включені в реляційну модель даних. Конкретна мова маніпулювання реляційними БД називається реляційно повною, якщо будь-який запит, виражається за допомогою однієї операції реляційної алгебри або однієї формули реляційного числення, може бути виражений за допомогою одного оператора цієї мови. Механізми реляційної алгебри і реляційного числення еквівалентні, тобто для будь-якого допустимого виразу реляційної алгебри можна побудувати еквівалентну формулу реляційного числення і навпаки. Чому ж в реляційній моделі даних присутні обидва ці механізми? Річ у тім, що вони розрізняються рівнем процедурності.

Вирази реляційної алгебри будуються на основі операцій (високого рівня) алгебри, і подібно тому, як інтерпретуються арифметичні і логічні вирази, вираз реляційної алгебри також має процедурну інтерпретацію. Іншими словами, запит, представлений на мові реляційної алгебри, може бути реалізований на основі обчислення елементарних операцій алгебри з урахуванням їх старшинства і можливої наявності дужок. Для формули реляційного числення однозначна інтерпретація, взагалі кажучи, відсутня. Формула тільки встановлює умови, яким повинні задовольняти кортежі результуючого відношення. Тому мови реляційного числення є непроцедурними або декларативними. Оскільки механізми реляційної алгебри і реляційного числення еквівалентні, то в конкретній ситуації для перевірки ступеня реляційності якоїсь мови БД можна користуватися будь-яким з цих механізмів. Зауважимо, що вкрай рідко алгебра або числення в чистому вигляді приймаються як основа будь-якої мови БД. Зазвичай (як, наприклад, у випадку, наприклад, SQL) мова ґрунтується на певній суміші алгебраїчних і логічних конструкцій.

4.2.3.Операції реляційної алгебри

Операції реляційної алгебри визначені на множині відношень і є замкнутими щодо цієї множини (утворюють алгебру). Виявляється, що будь-який довільний запит до БД можна

представити у вигляді послідовності, складеної з п'яти основних операцій реляційної алгебри. Розглянемо ці операції.

Об'єднання $r \cup s$ Об'єднанням відношень *r* і *s* називається множина кортежів, які належать або *r* або *s* або ним обом. Для операції об'єднання потрібна однакова арність відношень. Для прикладу, нехай :

r

a	b	a
b	a	f
c	b	d

s

b	g	a
d	a	f

Тоді об'єднання $r \cup s$ буде мати вигляд:

r ∪ s

a	b	a
d	a	f
c	b	d
b	g	a

Зауважимо, що за допомогою операції об'єднання може бути реалізовано додавання нового кортежу до наявного відношення. В цьому випадку *r* – початкове відношення, *s* – відношення, що містить один кортеж, що додається.

Різниця $r - s$ Різницею відношень *r* і *s* називається множина кортежів, належать *r*, але не належать *s*. Для цієї операції також потрібен однакова арність відношень.

r – s

a	b	a
c	b	d

Зауважимо, що за допомогою операції різниці може бути реалізовано видалення кортежу з наявного відношення. В цьому випадку *r* – початкове відношення, *s* – відношення, що містить один кортеж, що видаляється.

Декартовий добуток $r \times s$ Нехай *r* і *s* відносини арності $к_1$ і $к_2$ відповідно. Декартовим добутком $r \times s$ називається множина кортежів довжини $к_1+к_2$, перші $к_1$ компонентів яких утворюють кортежі, що належать *r*, а останні $к_2$ кортежів, належать *s*.

r × s

a	b	a	b	g	a
a	b	a	d	a	f
d	a	f	b	g	a
d	a	f	d	a	f
c	b	d	b	g	a
c	b	d	d	a	f

Проекція $\pi_{Ai\ Ai\ Aim}(r)$ Проекція $\pi_{Ai\ Ai\ Aim}(r)$ є множина кортежів, одержаних з кортежів відношення *r* вибором колонок з іменами $A_{i1}, A_{i2},..., A_{im}$. Іншими словами, це операція

побудови «вертикальної підмножини», одержаної шляхом вибору певних атрибутів і виключення інших. Кортежі, що повторюються, виключаються.

$\pi_{1,3}(r)$

a	c
d	f
c	d

Вибір (селекція) $\sigma_F(r)$ Нехай F – формула, що утворена операндами, константами або іменами атрибутів, арифметичними операторами порівняння, логічними операторами (і, або, не), що є, тоді вибором (селекцією) σ_F називається множина кортежів, компоненти якої задовольняють умові, заданій формулою F. Наприклад:

$\sigma(1)_{=(3)}(r)=$

a	в	c

Тут *F:(1)=(3)* – зміст першої колонки дорівнює змісту третьої колонки. Приведемо ряд прикладів представлення запитів в задачі зарахування абітурієнтів за допомогою формальних операцій.

Приклад 4.1. Сформувати список абітурієнтів (прізвище, ім'я, по батькові).

Розглянемо сутність АБІТУРІЄНТ (*entrant*). Атрибут «Прізвище» позначений тут «*surname* «. Для відповіді на запит необхідно взяти проекцію відношення entrant на колонку surname.

π surname (entrant)

Приклад 4.2. Надати список прізвищ і дат народжень абітурієнтів, яким на поточну дату (*pot_date*) більше 35 років. Розглянемо те ж відношення *entrant*. Спочатку обираємо абітурієнтів, яким більше 35 років.

σ birth _ date□35>pot _ date (entrant).

Потім беремо проекцію отриманого відношення на колонки surname і birth_date.

π surname,birth _ date (sbirth _ date□35>pot _ date (entrant)).

Зауважимо, що можна було б виконати ці дві операції в іншій послідовності – спочатку проекція, а потім селекція. Пропонується оцінити, який з цих варіантів краще за оцінкою числа виконуваних елементарних дій і об'єму необхідної пам'яті.

Приклад 4.3. Видати список прізвищ абітурієнтів, що отримали оцінку «незадовільно». Оцінка абітурієнта є атрибутом відношення *mark*. Якби в цьому відношенні був присутній атрибут «прізвище», то задача розв'язувалася б аналогічно прикладу 4.2. Щодо *mark* присутній атрибут «ідентифікатор абітурієнта» – *entrant_id*, а прізвище присутнє щодо *entrant.* Для відповіді на цей запит необхідно зв'язувати по *entrant_id* відношення *mark* і відношення *entrant*. Спочатку виберемо з відношення *mark* кортежі з оцінкою « незадовільно «. Позначимо отримане відношення R_1. (Подальші проміжні відносини позначатимемо послідовно R_2, R_3 і т.д.): $R_1 = \sigma mark=«незадовільно»(mark)$

Далі нас цікавитиме тільки атрибути *entrant_id* і *mark*. Тому візьмемо проекцію на ці колонки: $R_2 = \pi_{entrant\_id, mark}(R_1)$

Далі необхідно зв'язати відносини entrant і R_2 (склеїти таблиці). Для склеювання таблиць використовується операція декартовий добуток: $R_3 = entrant \times R_2$.

Щодо R_3 присутні дві однакові колонки: *entrant_id з* відношення *entrant і* відношення R_2. Вибираючи з відношення R_3 рядка, в яких значення entrant_id у відповідних колонках співпадають, отримаємо відомості про абітурієнтів, що отримали оцінку незадовільно.

$R_4 = \pi_{\sigma entrant.\ entrant\_id\ =\ R2.\ entrant\_id}(R_3)$, де *entrant.entrant_id* і R_2*.entrant_id* – позначають відповідно колонка *entrant_id* відповідної першої і другої складової частини декартова добутку. Тепер залишилося тільки обрати прізвища відповідних студентів

$R_5 = \pi_{surname}(R_4)$.

Одержуємо необхідний результат. Зауважимо, що для економії дій і пам'яті, перш ніж склеювати таблиці, доцільно зробити операцію проекції відношення *entrant* на колонки *entrant_id*, surname (щоб не включати в декартовий добуток зайві колонки). Введені п'ять основних операцій реляційної алгебри дозволяють реалізувати будь-який запит до реляційної бази даних. Проте разом з основними операціями достатньо часто зручно використовувати так звані додаткові операції реляційної алгебри (які можуть бути виражений через основні).

Перетин *r*∩*s*. Перетином відношень *r* і *s* називається множина кортежів, що належать як *r*,такі s. Перетин може бути виражений через операції різниці r∩s= r – (r – s).

Θ-з'єднання r і s по колонках A_i і A_j є множиною таких кортежів в декартовому добутку r і *s*, що *i-ий* компонент *r* знаходиться щодо Θ з *j-им* компонентом *s*, де Θ – арифметичний оператор порівняння. Якщо Θ є оператором рівності, то ця операція називається еквіз'єднання.

Θ-*з'єднання* $r \underset{i\Theta j}{><} s = \sigma_{i\Theta(l+i)}(r \times s)$, де l - арність відношення r.

Приклад 4.4 Задано *r* і *s:*

r

1	2	3
4	5	6
7	8	9

S

3	1
6	2

Тоді:

$r \underset{(2)<(1)}{><} s$

1	2	3	3	1
1	2	3	6	2

Зауважимо, що в цьому прикладі дві операції (декартовий добуток і селекція), які послідовно виконуються, разом представляють операцію з'єднання. Причому, використання декартова добутку для з'єднання таблиць обов'язково обумовлює використання селекції, як наступної операції для встановлення зв'язку між таблицями. Тому доцільно використовувати таку з'єднану операцію і програмно реалізовувати в СУБД саме операцію з'єднання.

Природне з'єднання $r >\!\!< s$ Операція може бути застосована тоді і тільки тоді, коли колонки мають імена (є атрибутами). Операція застосовується до відношень, у яких є однакові атрибути (передбачається, що у колонок є імена).

Нехай $r = (A_1 ..., A_k, B_1..., B_n)$, $s = (A_1 ..., A_k, C_1..., C_m)$, імена $A_1 ..., A_k$ співпадають. Тоді $r >\!\!< s$ визначається таким чином:

$$r >\!\!< s = \Pi_{B1,...,Bn,A_1,..., A_k,C_1,...,Cm}(\sigma_{r.A1=s.A_1\&r.A2=s.A_2\&...r.Ak=s.Ak}(r \times s))$$

.

Щоб підкреслити важливість розглянутих *операцій* реляційної алгебри, а також для уточнення поняття реляційної СУБД, приведемо таке визначення К.Дж.Дейта, видатного фахівця в області реляційних баз даних : «називатимемо систему реляційною, якщо вона підтримує, принаймні, реляційні бази даних, тобто бази даних, які можуть сприйматися користувачем як таблиці і лише як таблиці, операції селекції, проекції і з'єднання реляційної алгебри, не вимагаючи при цьому, щоб якимсь чином були перекриті фізичні шляхи доступу для підтримки цих операцій».

4.2.4. Проблема вибору раціональних схем відношень

При представленні концептуальної схеми у вигляді реляційної моделі можливі різні варіанти вибору схем відношень. Одні варіанти вибору схем відношень розглядалися в попередніх розділах, інші варіанти отримуються об'єднанням (або розбиттям) деяких схем відношень. Від правильного вибору схем відношень, що представляють концептуальну схему, в значній мірі залежатиме ефективність функціонування бази даних. Розглянемо для прикладу конкретну схему відношень і проаналізуємо її недоліки. Нехай дані про абітурієнтів, спеціальності, форми навчання включені в таблицю, що містить дані про атрибути. Схема відношень має наступний вигляд: АБІТУРІЄНТ (код абітурієнта, П.І.П., факультет, спеціальність, форма навчання). Ця схема відношень обумовлює наступні недоліки відповідної бази даних:

- Дублювання інформації (надлишковість). Для абітурієнтів, що вступають на один факультет повторюватиметься назва факультету. Для різних факультетів повторюватимуться однакові форми навчання, спеціальності.
- Потенційна суперечність (аномалії оновлення). Якщо, наприклад, зміниться назва спеціальності, то, змінюючи її в одному кортежі (у одного абітурієнта), необхідно змінювати її і у всіх інших кортежах, де вона присутня.
- Потенційна можливість втрати даних (аномалії видалення). При видаленні всіх абітурієнтів, що поступають на певну спеціальність, ми втрачаємо всі відомості про спеціальність.
- Потенційна можливість не включення інформації в базу даних (аномалії включення). В базі даних будуть відсутні відомості про спеціальність, якщо на неї абітурієнти не подали заяв.

В теорії реляційних баз даних існують формальні методи побудови реляційної моделі бази даних, в якій відсутня надлишковісь і аномалії оновлення, видалення і включення. Побудова раціонального варіанту схем відношень (що мають кращі властивості при операціях включення, модифікації і видалення даних, ніж вся решта наборів схем) здійснюється шляхом так званої нормалізації схем відношень. Нормалізація схем відношень проводиться у декілька етапів. На початковому етапі схема відношень повинна знаходитися в першій нормальній формі (1НФ). Відношення знаходиться в першій нормальній формі, якщо всі атрибути відношення приймають прості значення (атомарні або неподільні), не є множиною або кортежем з більш елементарних складових. Далі відношення, представлене в першій нормальній формі, послідовно перетвориться в другу і третю нормальні форми. Процес побудови другої і третьої нормальних форм буде описаний нижче. При деяких припущеннях про дані третя нормальна форма є достатньо добрим варіантом. Якщо ці припущення не виконуються, то процес нормалізації продовжується і відношення перетвориться в четверту і п'яту нормальну форму. Побудова відповідних форм описана в [], і в даному посібнику не розглядається, виходячи з принципу розумної достатності. Перш ніж перейти до побудови другої нормальної форми, необхідно визначити цілий ряд формальних понять. Найголовнішим є поняття функціональної залежністі , або залежність між атрибутами відношення

Визначення: нехай R ($A_1, A_2, ..., A_n$) – схема відношення, *а* X і У – підмножини $\{A_1, A_2, ..., A_n\}$. Тоді X функціонально визначає $У$, або $У$ функціонально залежить від X (позначається $X \rightarrow Y$) тоді і тільки тоді, коли кожне значення множини X відношення R пов'язано з одним значенням множини $У$ відношення R. Інакше кажучи, якщо два кортежі R співпадають по значенню X, вони співпадають і по значенню $У$.

Зауваження: функціональна залежність характеризує всі відношеня, які можуть бути значеннями схеми відношення R у принципі. Тому єдиний спосіб визначити функціональну залежність – уважно проаналізувати семантику (значення) атрибутів. Функціональна залежність є, зокрема, обмеженнями цілісності, тому доцільно перевіряти їх при кожному оновленні бази даних. Приклад функціональної залежності:

Код Абітурієнта→Прізвище, Ім'я, По батькові.

4.2.5. Повна множина функціональної залежності.

Для кожного відношення існує цілком певна множина функціональних залежностей між атрибутами даного відношення. Причому з однієї, або більше функціональної залежності, властивої даному відношенню, можна вивести іншу функціональну залежність, також властиві цьому відношенню. Визначену множину функціональної залежності для відношення R позначимо F, повну множину функціональної залежності, яку логічно можна отримати з F, називається замиканням F і позначається F+. Якщо множина функціональної залежності співпадає із замиканням даної множини, то така множина функціональної залежності називається повною. Введені поняття дозволяють формально визначити поняття ключа. Нехай існує певна схема R з атрибутами $A_1,A_2,...,A_n$, F – якась множина функціональної залежності і X – деяка підмножина R. Тоді X називається ключем, якщо, по-перше, в $F+$ існує залежність $X \rightarrow A_1,A2,...,A_n$ і, по-друге, для жодної підмножини У, що входить в X, залежність $У \rightarrow A_1A_2,...,A_n$ не належить $F+$. Повною функціональною залежністю називається залежність неключового атрибута від всього складного ключа. Частковою функціональною залежністю називатимемо залежність неключового атрибута від частини складного ключа. Для обчислення замикання множини функціональної залежності використовуються наступні правила виведення (аксіоми Армстронга): Нехай визначена певна схема відношення R з множиною атрибутів $A_1, A_2,..,A_n\}$ і множина функціональних залежностей, визначених на певній множині U.

Аксіома рефлективності. Якщо Y входить в X, а X входить в U, то $X \rightarrow Y$ логічно слідує з F. Це правило дає тривіальну залежність, оскільки в цій залежності права частина міститься в лівій частині.

Аксіома поповнення. Якщо $X \rightarrow Y$ і Z є підмножина U, то $XZ \rightarrow YZ$. В даному випадку функціональна залежність $X \rightarrow Y$ або містилася в початковій множині F, або може бути виведений з F з використанням описаних аксіом.

Аксіома транзитивності. Якщо $X \rightarrow Y$ і $У \rightarrow Z$, то $X \rightarrow Z$. Аксіоми Армстронга є повними і надійними. Це значить, що використовуючи їх, ми виведемо всю можливу

функціональну залежність, що логічно слідує з *F*, і не виведемо жодної зайвої залежності. Існують дещо інші правил висновку, які виходять з аксіом Армстронга.

Правило самовизначення. $X \rightarrow X$

Правило об'єднання. Якщо $X \rightarrow Y$ і $X \rightarrow Z$, то $X \rightarrow Y \rightarrow Z$.

Правило псевдотранзитивності. Якщо $X \rightarrow Y$ і $W \rightarrow Y \rightarrow Z$, то $X \rightarrow W \rightarrow Z$.

Правило композиції. Якщо $X \rightarrow Y$ і $Z \rightarrow W$, то $X \rightarrow W \rightarrow Y \rightarrow W$.

Правило декомпозиції. Якщо $X \rightarrow Y$ і Z входить в $У$, то $X \rightarrow Z$.

Треба відзначити, що обчислення замикання множини функціональної залежності є трудомісткою задачею при достатньо великій кількості атрибутів (за рахунок виписування великої кількості тривіальних залежностей).

4.2.6. Декомпозиція схеми відношення

Послідовний перехід від однієї нормальної форми до іншої при нормалізації схем відношень здійснюється за допомогою декомпозиції. Основною операцією, за допомогою якої здійснюється декомпозиція, є проекція. Декомпозицією схеми відношення R = $\{A_1, A_2, .A_n\}$називається заміна її сукупністю підмножин R, таких, що їх об'єднання дає R.

При цьому допускається, щоб підмножини перетиналися. Декомпозиція повинна забезпечити те, що запити (вибірка даних по умові) до початкового відношення і відношень, отриманих в результаті декомпозиції, дадуть однаковий результат. Відповідна умова виконуватиметься, якщо кожний кортеж відношення R може бути представлений як природне з'єднання його проекцій на кожну з підмножин. В цьому випадку говорять, що декомпозиція має властивість з'єднання без втрат.

Алгоритм декомпозиції базується на наступній теоремі.

Теорема Фейджина. Нехай *R(A, B, C)* відношення, *A, B, C* – атрибути. Якщо R задовольняє залежність $A \rightarrow B$, то *R* дорівнює з'єднанню його проекцій *A, B і A, C*

R(A, B, R(A, B) R(A, C)

При нормалізації необхідно вибирати такі декомпозиції, які мають властивість з'єднання без втрат. Для перевірки того, чи має декомпозиція згадану властивість використовується спеціальний алгоритм перевірки.

4.2.7. Вибір раціонального набору схем відношень шляхом нормалізації

Перша нормальна форма(1НФ) має повний набір недоліків: аномалії включення, аномалії видалення, аномалії оновлення, дублювання інформації. Тому Кодд запропонував цілий каскад поліпшень шляхом декомпозицій. Друга нормальна форма (2НФ) витікає з першої шляхом накладення обмеження на функціональну залежність. Відношення знаходиться в 2НФ, якщо воно знаходиться в 1НФ і кожний неключовий атрибут залежить

від первинного ключа (не залежить від частини ключа). Для переходу відношення в 2НФ необхідно, використовуючи операцію проекції, розкласти його на декілька відношень таким чином:

1) побудувати проекцію без атрибутів, що знаходяться в частковій функціональній залежності від первинного ключа;

2) побудувати проекції на частини складового ключа і атрибути, залежні від цих частин.

Друга нормальна форма значно знижує надлишковість, але по суті не усуває жоден з видів аномалій.

Розвитком 2НФ є третя нормальна форма (3НФ). Відношення знаходиться в 3НФ, якщо воно знаходиться в 2НФ і кожний ключовий атрибут нетранзитивно залежить від первинного ключа. Відношення знаходиться в 3НФ тоді і лише тоді, коли всі неключові атрибути відношення взаємно незалежні і повністю залежать від первинного ключа. Виявляється, що будь-яка схема відношень може бути приведена до 3НФ декомпозицією, що має властивості з'єднання без втрат і зберігає залежності. Третя нормальна форма виключає надлишковість і аномалії включення і видалення. Проте, 3НФ не запобігає всім можливим аномаліям.

Нормальна форма Бойса-Кодда (НФБК). Якщо в R для кожної залежності X ->A, де A не належить X, X включає деякий ключ, то говорять, що дане відношення знаходиться в нормальній формі Бойса-Кодда. Детермінантом функціональної залежності називається мінімальна група атрибутів, від якої залежить якийсь інший атрибут або група атрибутів, причому ця залежність нетривіальна. Відношення знаходиться в НФБК тоді і тільки тоді, коли кожний його детермінант є потенційним ключем. НФБК є більш строгою версією 3НФ. Іншими словами: будь-яке відношення, що знаходиться в НФБК, знаходиться в 3НФ. Але не навпаки.

Приклад 4.5. Розклад консультацій. Кожна група може прийти на консультацію один раз в день. Для проведення консультації або консультацій викладачу на певний день надається аудиторія. Протягом дня дана аудиторія може використовуватися різними викладачами.

РОЗКЛАД КОНСУЛЬТАЦІЙ (Група, Дата, Час, Викладач, Аудиторія)

Як первинний ключ вибрали «Група, Дата».

Потенційні ключі: «Група, Дата», «Викладач, Дата, Час», «Аудиторія, Дата, Час».

Таблиця 4.1 Розклад консультацій

Група	Дата	Час	Викладач	Аудиторія
УК-21	13травня	1030	Пасічніченко	330-7
УК-22	13 травня	12.20	Пасічніченко	330-7

УК-31	13 травня	12.20	Капустян	332-7
УК-22	1 червня	10.25	Пасічніченко	332-7

Всі атрибути залежать від всього ключа. Неповної залежності не існує. Немає транзитивної залежності. Проте існує наступна залежність – атрибути Викладач, Дата,Час визначають атрибут Аудиторія. Порушується визначення форми БК – даний складовий атрибут є детермінантом, але не є потенційним ключем. Дана функціональна залежність не порушує третьої нормальної форми, проте породжує аномалії оновлення. Викладачу Пасічніченко на 13 травня виділена аудиторія 332-7. Якщо із якихось причин адміністрація виділяє викладачу іншу аудиторію, то інформація про це повинна бути внесена двічі, що може бути не зроблено і, відповідно, бути причиною приведення бази даних в суперечливий стан. Вихід – розбиття початкового відношення на два. РОЗКЛАД КОНСУЛЬТАЦІЙ (Група, Дата, Час, Викладач) АУДИТОРІЇ (Викладач, Дата, Аудиторія) Важливо відзначити, що хоча таке розбиття (декомпозиція) приводить до усунення надлишкості даних, воно також приводить до втрати функціональної залежності: Аудиторія, Дата, Час перестають бути потенційним ключем, оскільки тепер ці атрибути знаходяться в різних відношеннях.

В нормальній формі Бойса-Кодда не існує надлишковості і аномалій включення, видалення і модифікації. Виявляється, що будь-яка схема відношення може бути приведена в нормальну форму Бойса-Кодда, так, щоб декомпозиція мала властивість з'єднання без втрат. Проте схема відношення може бути такою, що не приводиться в НФБК із збереженням залежності. В цьому випадку доводиться задовольнятися третьою нормальною формою.

4.2.8. Цілісна частина реляційної моделі.

Нагадаємо, що під цілісністю бази даних розуміється те, що в ній міститься повна, несуперечлива і адекватна інформація, що відображає предметну частину. Підтримка цілісності в реляційних БД заснована на виконанні наступних вимог.

1. Цілісність сутностей. Об'єкту або сутності із реального світу в реляційних БД відповідають кортежі відношень. Конкретно вимога полягає в тому, що будь-який кортеж довільного відношення відрізняється від будь-якого іншого кортежу цього відношення, тобто іншими словами, будь-яке відношення повинне мати певний первинний ключ. Ця вимога автоматично задовольняється, якщо в системі не порушуються базові властивості відношень.

2. Цілісності за посиланнями. Очевидно, що при дотриманні нормалізованості відношень складні сутності реального світу представляються в реляційній БД у вигляді декількох кортежів певної кількості відношень. Зв'язок між відношеннями здійснюється за допомогою міграції ключа. Приклад зовнішнього ключа. СТУДЕНТ (Код студента, ПІП) здає ІСПИТ (Код студента, Предмет, Оцінка) Атрибут Код студента сутностей ІСПИТ називається зовнішнім ключем, оскільки його значення однозначно характеризують сутність,

представлені кортежами деякого іншого відношення – відношення Студент (ми припускаємо, що поле Код студента є ключем відношення Студент). Говорять, що відношення, в якому визначений зовнішній ключ, посилається на відповідне відношення, в якому такий же атрибут є первинним ключем.

Вимога цілісності по посиланнях, або вимога до зовнішнього ключа полягає в тому, що для кожного значення зовнішнього ключа у відношенні, що посилається, у відношенні, на яке веде посилання, повинен знайтися кортеж з таким же значенням первинного ключа, або значення зовнішнього ключа повинне бути невизначеним (тобто ні на що не вказувати). Обмеження цілісності сутностей і цілісності за посиланнями повинні підтримуватися СУБД. Для дотримання цілісності сутностей достатньо гарантувати відсутність в будь-якому відношенні кортежів з одним і тим же значенням первинного ключа. (В MS Access для цього призначена спеціальна реалізація цілочисельного поля – поле типу «Counter»).

З цілісністю за посиланнями справи виглядають дещо складніше. Зрозуміло, що при оновленні відношення (вставці нових кортежів або модифікації значення зовнішнього ключа в існуючих кортежах), що посилається, достатньо стежити за тим, аби не з'являлися некоректні значення зовнішнього ключа. Але як бути при видаленні кортежу з відношення, на яке веде посилання? Тут існують три підходи, кожний з яких підтримує цілісність по посиланнях.

Перший підхід полягає в тому, що забороняється проводити видалення кортежу, на який існують посилання (тобто спочатку потрібно або видалити кортежі, що посилаються, або відповідним чином змінити значення їх зовнішнього ключа).

При другому підході при видаленні кортежу, на який є посилання, у всіх кортежах, що посилаються, значення зовнішнього ключа автоматично стає невизначеним.

Нарешті, третій підхід (каскадне видалення) полягає в тому, що при видаленні кортежу з відношення, на яке ведеться посилання, з відношення, що посилається, автоматично віддаляються всі кортежі, що посилаються. В розвинутих реляційних СУБД звичайно можна вибрати спосіб підтримки цілісності за посиланнями для кожної окремої ситуації визначення зовнішнього ключа. Звичайно, для ухвалення такого рішення необхідно аналізувати вимоги конкретної прикладної області. Зауважимо, що всі сучасні СУБД підтримують і цілісність сутностей і цілісність по посиланнях, але дозволяють користувачам вимикати ці обмеження і, таким чином, будувати бази даних, що не відповідають реляційній моделі. Досвід показує, що відхід від основних положень реляційної моделі приводить до короткострокового виграшу – алгоритми стають простішими, але згодом серйозно ускладнюють задачу, особливо супровід задачі.

4.3 Представлення запитів до реляційних БД

Все, что видим мы, - видимость только одна.
Далеко от поверхности мира до дна.
Полагай несущественным явное в мире,
Ибо тайная сущность вещей – не видна.
(Омар Хайам

Запити можуть бути представлені в різних формах. В контексті оптимізації запитів доцільне представлення запитів повинне задовольняти наступним вимогам: Воно повинне бути достатньо потужним для можливості забезпечення великого класу запитів і повинне забезпечувати правильно побудований базис для перетворення запитів. Далі розглянемо чотири різні форми представлення запитів, кожна з яких використовується у ряді підходів до оптимізації запитів.

4.3.1. Реляційне числення

Реляційне числення, як воно було введено Коддом - це нотація для визначення результату запиту через опис його властивостей. Представлення запиту в реляційному численні складається з двох частин: цільовий список (target list) і вираз вибірки (selection expression).

Вираз вибірки специфікує вміст відношення, що відбувається в результаті виконання запиту, засобами предикатів першого порядку (тобто узагальнених булевих виразів, можливо, що містять квантори існування і загальності).

В цільовому списку визначаються вільні змінні, що зустрічаються в предикаті, і специфікується структура результуючого відношення. В Прикладі 4.6. демонструється представлення реляційного числення з використанням псевдокоду, запропонованого в [] [Schmidt 1977] і необхідного тут як нотація алгоритму запитів.

Приклад 4.6. *Імена професорів, які мають публікацію в 2007 р.*

```
[<em.ename>
EACH em IN employees:
  em.status = professor
    AND
  SOME p IN papers
    (em.enr = p.enr AND p.year = 2007)]
```

В цільовому списку, тобто підвиразі, що передує двокрапці, область визначення (вільної) змінної em обмежується елементами відношення «employees». Тому відношення «employees» називається відношенням області визначення (*range relation*) змінної em. Специфікація атрибута мети «*<em.ename>*« показує, що в результаті запиту залишаться тільки імена службовців.

Вираз вибірки - предикат, наступний за двокрапкою - визначає обмеження на вільну змінну. Перше обмеження є обмежуючим, або одномісним термом, *який* обмежує вільну змінну тими записами «*employees*», значенням колонки *status* яких є значення «*professor*». Це обмеження зв'язується через AND із сполучним, або двомісним термом, що пов'язує *«employees» з «papers*», і ще одним одномісним термом, що ще більш обмежує результат тими співробітниками, які опублікували яку-небудь статтю в 2007 р.

В реляційному численні дозволяються змінні, прив'язані до різних різновидів відношень області визначення; наприклад, змінна em пов'язана з «*employees*», а змінна *p* - з «*papers*». Окрім логічної операції AND, в предикатах можуть використовуватися і операції OR і NOT. Предикати реляційного числення повністю визначаються наступними рекурсивними правилами:

1. Атомарні предикати:

1.1. (Одномісний або двомісний) терм є атомарним предикатом.

1.2. *TRUE* є атомарним предикатом.

1.3. *FALSE* є атомарним предикатом.

2. Атомарний предикат є предикатом. Хай A - предикат, r - змінна елемента, і rel - відношення. Тоді

   2.1. SOME r IN rel(A)

   2.2. *ALL r IN rel(A)* також є предикатами.

3. Хай A і B - предикати. Тоді

   3.1. *NOT (A*) (заперечення)

   3.2. *A AND B* (кон'юнкція)

   3.3. *A OR B* (диз'юнкція) є предикатами.

4. Жодні інші формули предикатами не є.

Форма представлення називається *реляційно повною*, якщо в ній допускається визначення результату будь-якого запиту, що визначається виразом реляційного числення. Ясно, що реляційна повнота повинна розглядатися як мінімальна вимога щодо виразної потужності. Крім того, запити в сьогоднішніх прикладних програмах часто містять агрегації, які неможливо виразити в чистому реляційному численні. Проте реляційне числення досить легко розширюється агрегатними функціями .

4.3.2. Реляційна алгебра

Реляційна алгебра, як вона визначена Коддом в [Codd 1972] є набором операцій над відношеннями. Ці операції розпадаються на два класи: традиційні операції над множинами,

такі як декартовий добуток, об'єднання, перетин і різниця, і спеціальні операції реляційної алгебри, такі як обмеження, проектування, з'єднання і ділення. Нижче спеціальні операції визначаються шляхом їх зв’язування з еквівалентними виразами реляційного числення.

Операція обмеження, застосована до відношення «rel», конструює горизонтальну підмножину відповідно до безквантворного предиката, що містить тільки одномісні терми (порівняння між атрибутами одного елемента відношення):

Rest(rel, pred)= [EACH r IN rel: pred].

Операція проекції служить для конструювання вертикальної підмножини відношення «rel» шляхом вибору набору вказаних атрибутів A і видалення кортежів-дублікатів з цих атрибутів:

Proj(rel, A)= [(r.A) EACH r IN rel: true].

Операція *з'єднання* дозволяє сполучати два відношення *«rel1»* і «*rel2*» в одне відношення, атрибути якого є об'єднанням атрибутів «*rel1*» і «*rel2*»:

Join(rel1, A op B, rel2)= [EACH rl IN rel1, EACH r2 IN rel2: rl.A op r2.B].

Операції порівняння, що допускаються в з'єднаннях «*or*» є такими ж, як в двомісних термах реляційного числення. Якщо «*or*» є операцією порівняння на рівність, в результаті «природного з'єднання» опускається A або B.

Операція ділення є двійником алгебри квантора загальності. Вона визначається таким чином:

```
Divi(rel1, A B, rel2)= [(r.compl(A)) EACH rl IN rel1:
        ALL rel2 IN rel2 SOME r3 IN rel1
       (rl.compl(A)= rel2.compl(A) AND rel2.B = r3.A)]
```

де compl(A) - це доповнення A в множині атрибутів «rel1». Як показує визначення, ділення є досить складною операцією, що може утрудняти розуміння запитів.

В Прикладі 4.7. представляється запит Прикладу 4.6. в термінах реляційної алгебри.

Приклад 4.7. *Імена професорів, що мають публікації в 2007 р.*

```
Proj (Rest(Join(employees
            enr = enr
            Rest(papers, year = 2007))
        status = professor)
    ename)
```

В противагу виразу реляційного числення, яке описує відношення, що виходить із запиту, в термінах його властивостей, вираз реляційної алебри визначає алгоритм конструювання результуючого відношення. Вираз числення здається кращим варіантом для оптимізації запитів, оскільки воно забезпечує оптимізатор тільки базовими властивостями запиту; можливості оптимізації можуть бути приховані в конкретній послідовності операцій

алгебри. Для візуального представлення реляційних запитів зручним засобом представлення є графи. Можна виділити два класи графів: об'єктні графи і графи операцій.

В об'єктних графах вузли представляють об'єкти, такі як змінні (відношення) і константи. Дуги описують предикати, яким ці об'єкти повинні задовольняти [Bernstein and Chiu 2007; Palermo 1972; Youssefi and Wong 1979]. Об'єктні графи містять властивості результатів запитів і тому тісно пов'язані з реляційним численням.

Графи операцій описують керовані операціями потоки даних шляхом представлення операцій як вузлів, зв'язаних дугами, що вказують напрям руху даних. В [Smith and Chang [1975]; Yao [1979] графи операцій використовувалися для представлення виразів алгебри. На Рис. 4.1 і Рис. 4.2 наведені приклади об'єктного графа і графа операцій відповідно.

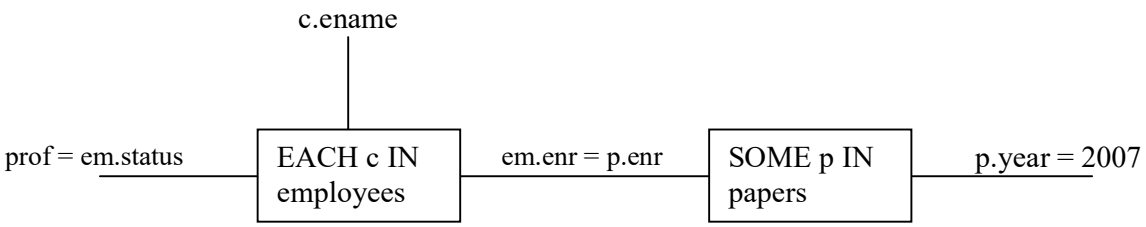


Рис. 4.1. Об'єктний граф, що представляє запит з прикладу 4.7.

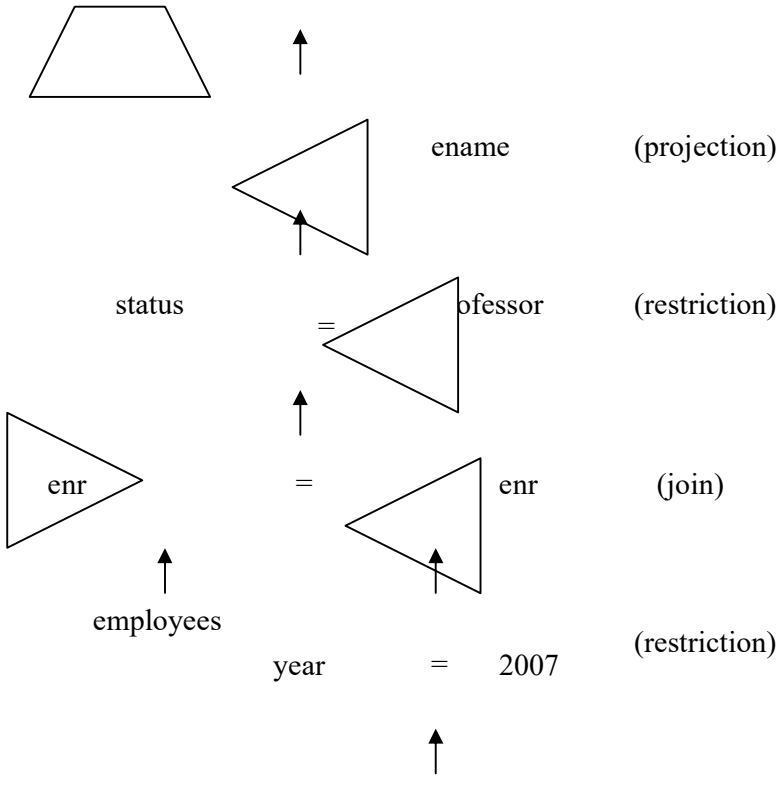


Рис. 4.2. Граф операцій запиту з прикладу

У графів запитів є багато привабливих властивостей. Візуальне представлення запиту сприяє більш простому розумінню його структурних характеристик. Крім того, в теорії графів є багато результатів, корисних для автоматичного аналізу графів, наприклад, виявлення циклів і властивості деревовидності.

4.3.3. Табло

Табло, як вони були визначені Ахо і ін. [Aho et al. 1979a, 1979b, 1979c] є табличною нотацією для підмножини запитів реляційного числення, що характеризуються наявністю тільки зв'язаних через AND термів і відсутністю кванторів загальності. Таким чином табло-запити є частковим різновидом кон'юнктивних запитів [Chandra and Merlin 1977; Rosenkrantz and Hunt 1980]. Табло являють собою спеціалізовані матриці, колонки яких відповідають атрибутам відповідної схеми бази даних. Перший рядок матриці, підсумок(summary), служить тій же меті, що цільовий список виразу реляційного числення. Решта рядків описує предикат. Символи, що з'являються в табло, є виділеними (distinguished) змінними (відповідними вільним змінним), невиділеними (nondistinguished) змінними (відповідними змінним під знаком квантора існування), константами, пропусками і тегами (вказуючими відносини областей визначення).

На Рис. 4.3. ілюструється конструювання табло, що представляє запит з Прикладу 4.1. Процес починається з табло для одиночних відношень і продовжується шляхом комбінування цих табло в нові табло для всіх великих виразів. Виділені змінні наголошуються символами *a*; невиділені - символом *b*.

T(employees) =

status	ename	enr	year	
a1	a2	a3		
a1	a2	a3		employees

T(papers) =

		a3	a4	
		a3	a4	papers

T(Rest(papers,years=1981)) =

		a3	1981	
		a3	1981	papers

T(Join(employees,enr=enr,Rest(papers,years=1981)) =

a1	a2	a3	1981	
a1	a2	a3		employees
		a3	1981	papers

T(Rest(Join(employees,enr=enr,Rest
(papers,years=1981),status=professor)) =

professor	a2	a3	1981	

Рис.4.3. Покрокове конструювання табло T, що представляє запит з Прикладу 4.1

Вирази, що містять диз'юнкцію (об'єднання множин) і заперечення (віднімання множин) можуть представлятися наборами табло .

4.4. Перетворення запитів

Ми бачимо вирази запитів можуть бути різні. Крім того, навіть всередині заданої мови для кожного запиту може існувати ряд семантично еквівалентних виразів. Розглянемо перетворення виразу запитів в еквівалентний вираз на основі певних правил. Мета перетворення запитів полягає в :

3. конструювання стандартизованной форми для оптимізації запиту (стандартизація)
4. усунення надлишковості (спрощення)
5. конструювання вдосконалених виразів по відношенню до ефективності виконання (поліпшення).

4.4.1. Стандартизація

В деяких підходах до оптимізації запитів стандартизована форма запиту визначається через нормалізований варіант [Jarke and Schmidt 1981, Kim 1982; Palermo 1972; Wong and Youssefi 1976]. Розглянемо дві нормальні форми для реляційного числення, а також правила, яким повинна підлягати процедура нормалізації.

Кажуть, що представлення запиту з використанням реляційного числення знаходиться в *передуючій нормальній формі (prenex normal form)*, якщо його вираз вибірки має вигляд

SOME/ALL rel1 IN rel1 . . .

SOME/ALL rn IN reln(M)

де *M* - предикат, що не містить кванторів. *M* називається *матрицею (matrix)* і теж може бути стандартизована.

Кажуть, що матриця, що складається з диз'юнкції кон'юнкцій (термів Aij), таких як

$(A_{11}$ *AND . . .* $A_{1n})$ *OR . .*

OR $(A_{m1}$ *AND . . . AND* $A_{mn})$

знаходиться в *диз'юнктивній нормальній формі*, а матриця, що складається з кон'юнкції диз'юнктів, таких як

$(A_{11}$ *OR . . .* $A_{1n})$ *AND . .*

AND $(A_{m1}$ *OR . . . OR* $A_{mn})$

знаходиться в *кон'юнктивній нормальній формі*

Передуюча нормальна форма з нормальними формами матриці приводить до двох нормальних форм виразів реляційного числення: *диз'юнктивній передуючій нормальній формі (disjunctive prenex normal form*, DPNF) і *кон'юнктивній передуючій нормальній формі (conjunctive prenex normal form*, CPNF).

4.4.2. Спрощення і поліпшення

Ми вже бачили, що може існувати декілька семантично еквівалентних виразів одного і того ж запиту. Одним з найважливіших критеріїв відмінності між двома еквівалентними виразами є рівень надлишковості [Hall 1976; Stroet and Engmann 1979]. Прямолінійне обчислення надлишкового виразу привело б до виконання набору операцій, багато з яких є зайвими. Тому оптимізатор запитів прагне усунення надлишковості шляхом перетворення надлишкового виразу до еквівалентного виразу, що не є надмірним.

Результатом спрощення запиту не обов'язково буде унікальний вираз. Можуть існувати інші ненадлишкові вирази, семантично еквівалентні тому, який згенеровано певним методом спрощення. Обчислення виразів, відповідних одному і тому ж запиту може істотно розрізнятися параметрами ефективності, наприклад, розмірами проміжних результатів і кількістю звернень до елементів відношень.

Метою ряду вказаних перетворень є мінімізація розміру сконструйованих проміжних результатів, що зберігаються і прочитуються. Важливе евристичне міркування полягає в тому, щоб надавати більш високий пріоритет селективним операціям, таким як обмеження і проекція, у порівнянні з конструктивнимими операціями, такими як з'єднання та декартовий добуток, з тим, щоб виконувати селективні операції якомога раніше [Smith and Chang 1975]. В контексті реляційного числення розбір певної послідовності обчислень може бути

представлено вкладеним виразом. Обчислення вкладеного виразу починається з обчислення найбільш віддаленого внутрішньо вкладеного виразу, за яким йде безпосередньо оточуючий його вираз і т.д., поки не буде досягнутий останній зовнішній вираз.

Такий спосіб обчислення селективних операцій представляє окремий випадок від'єднання запиту (query detachment)[Wong and Youssefi 1976]. Підвираз, який перекривається з частиною запиту, що залишилася, по одній змінній, від'єднується і утворює внутрішню вкладеність. Від'єднання виконується рекурсивно на кожному рівні вкладеності доки вираз не перестане скорочуватися. В Прикладі 3.2 демонструється від'єднання підвиразу складного виразу.

Приклад 4.2. Відділи, що пропонують лекції, які проводяться професорами, що проживають в тому ж місті, в якому знаходиться відділ, і кожний з яких опублікував яку-небудь статтю в 2007 р.

Відповідний вираз:

[EACH d IN departments:
SOME l IN lectures
SOME em IN employees
(em.status = professor
AND
d.dname = l.dname
AND 1.enr = em.enr
AND em.city = d.city
AND
SOME p IN papers
(p.year = 2007
AND p.enr = em.enr))]

Ось еквівалентний вираз, проведений шляхом від'єднання запиту:

```
[EACH d IN departments:
  SOME l IN lectures
   SOME e IN [EACH e IN
            [EACH e IN employees:
              em.status = professor]:
            SOME p IN [EACH p IN
             papers: p.year = 2007]
            (em.enr = p.enr)]
    (d.dname = 1.dname AND 1.enr = em.enr
      AND em.city = d.city)]
```

Об'єктний граф, що представляє запит, показаний на Рис. 4.4

Відзначимо, що результуючий вкладений вираз є нероздільним [Goodman and Shmueli 1980]; тобто його не можна розділити на два підвирази, що перекриваються по одній змінній.

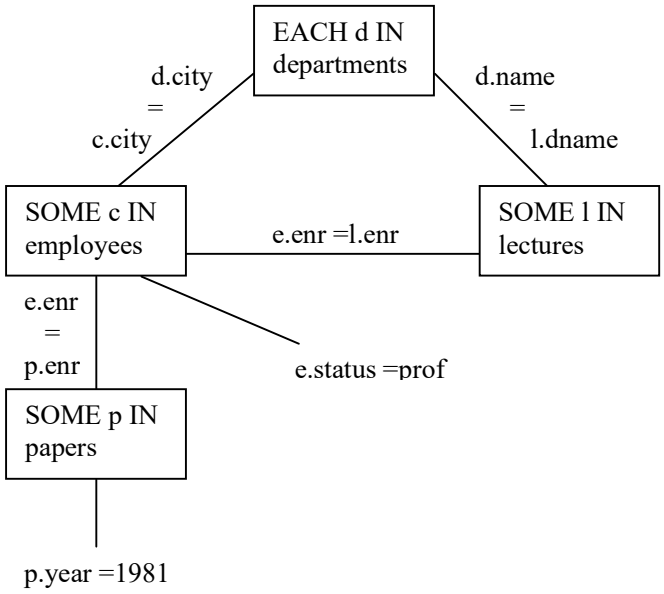


Рис 4.4. Об'єктний граф для Прикладу 4.2.

В представлених перетвореннях, що поліпшують запит, були керувалися такими міркуваннями:

загальні правила перетворення і евристики, правила їх застосування, знання структур реляційних даних і сам запит. Крім цього існують ще деякі фактори: обмеження цілісності, які в багатьох базах даних доповнюють структурне визначення схеми і реальні дані в базі даних.

Обмеження цілісності - це предикати, які повинні бути істинними для кожного елемента певного відношення або кожної комбінації елементів деякої групи відношень. Тому їх можна додавати до виразу вибірки будь-якого запиту без зміни його істинного значення.

На Рис. 4.5. приведена загальна схема реалізації реляційного запиту[]

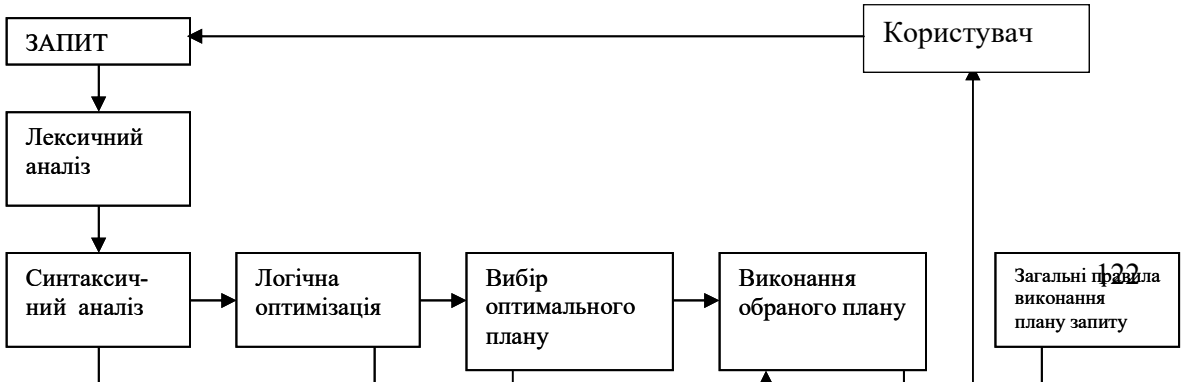


Рис. 4.5. Етапи обробки запиту в реляційній СУБД.

4.5. Декомпозиція запитів

Декомпозиція є першим етапом обробки запитів. Призначення етапу декомпозиції полягає в перетворенні запиту на мові високого рівня виразу реляційної алгебри, з подальшою перевіркою його синтаксичної і семантичної коректності. Звичайно декомпозиція включає стадії аналізу, нормалізації, семантичного аналізу, спрощення і реструктуризації запиту.

4.5.1.Стадія аналізу

На цій стадії виконується лексичний і синтаксичний аналіз запиту з використанням методів, вживаних в компіляторах мов програмування високого рівня [(Aho and Ulman, 1977)]. Додатково уточнюється, чи присутні в системному каталозі визначення для вказаних в запиті відношень і атрибутів. Крім того контролюється, чи можуть бути виконані операції над різними об'єктами бази даних . Наприклад, розглянемо наступний запит:

SELECT staff_no

FROM staff

WHERE position > 10;

Виконання цього запиту буде відхилено з наступних двох причин.

1.В списку вибірки запиту вказаний атрибут *Staff_No*, відсутній у визначенні відношення Staff (насправді відповідний атрибут має ім'я Sno).

2.У виразі *WHERE* операція порівняння «>10» несумісна з типом атрибута Position, який описаний у відношенні як рядок символів.

Після завершення даної стадії декомпозиції запит на мові високого рівня перетвориться до якогось внутрішнього представлення, більш зручного для виконання подальшої обробки.

4.6. Формування внутрішнього представлення запиту

Етап аналізу загалом-то є традиційною задачею трансляції. Всі труднощі реалізації запиту для реляційної СУБД якраз і лежать на етапі генерації коду запиту, критичним фактором тут є оптимізація виконуваного коду. Як вже зазначалось на прикладі, витратність виконання запиту суттєво залежить від порядку виконання послідовності операцій. Далі приведемо евристичні правила оптимізації запиту.

4.6.1 Стратегії евристичної обробки запитів

В багатьох СУБД для вибору стратегії обробки запитів використовуються ті або інші евристичні правила. Далі ми обговоримо декілька подібних правил, які частіше за все застосовуються на практиці для оптимізації обробки запитів.

1.Виконання операцій вибірки на якомога ранніх етапах обробки. Виконання операції вибірки скорочує кардинальність відношення і скорочує час подальших операцій обробки запиту. Отже, доцільно починати оптимізацію для формування каскадної послідовності операцій вибірки, після чого поміняти місцями унарні або бінарні операції операціями вибірки. Мета цих маніпуляцій полягає в тому, щоб перемістити операцію вибірки в дереві реляційної алгебри якнайнижче. Додатково корисно зберігати разом всі предикати, що стосуються одного і того ж відношення.

2.Об'єднання в одну операцію з'єднання операції декартового добутку і наступного за нею операції вибірки, предикат якої представляє умову з'єднання.

Вище вже наголошувалося, що операція вибірки з предикатом типу тета-з'єднання і операція декартового добутку можуть бути представлений однією операцією тета-з'єднання:

yR.a I S.b (R x S) = R >< R.a I S.b S

3. Використання властивості асоціативності бінарних операцій для переупорядковування листових вузлів таким чином, що листові вузли з найбільш обмежуючими операціями вибірки виконуватимуться в першу чергу.

Нагадаємо, що основне правило підвищення ефективності полягає у виконанні якомога більшої кількості операцій обмеження до моменту переходу до виконання бінарних операцій. Мета полягає в переупорядковуванні операцій так, щоб першим виконувалося з'єднання, в результаті якого буде отримано менше за розміром результуюче відношення. Це дозволить виконати друге з'єднання з меншим за розміром початковим відношенням.

4.6.2.Принцип раннього виконання операцій проекції.

Нагадаємо, що операції проекції зменшують кардинальність відношень, що приводить до скорочення об'ємів даних, що підлягають подальшій обробці. Метою даної процедури є переміщення операцій проекції вниз по дереву реляційної алгебри (наскільки це буде можливо). Додатково рекомендується об'єднувати операції проекції, виконувані над одним і тим же відношенням.

4.6.3.Одноразове обчислення загальних виразів.

Якщо один і той же вираз зустрічається в дереві реляційної алгебри кілька разів і результат його обчислення не дуже великий, доцільно зберегти цей результат після його першого обчислення з метою повторного використання у міру необхідності. Проте виграш буде отриманий тільки в тому випадку, якщо розмір результуючої таблиці загального виразу достатньо малий, щоб зберігати його в первинній пам'яті, або витратність його вибірки з вторинної пам'яті буде менше витратності його повторного обчислення. Цей підхід може виявитися особливо ефективним при роботі з представленнями, оскільки при кожному зверненні до деякого представлення повинні обчислюватися одні і ті ж вирази.

4.7.Оцінка витратності операцій реляційної алгебри

СУБД може підтримувати множину різних способів реалізації операцій реляційної алгебри. Призначення всього процесу оптимізації запитів полягає у виборі найефективнішого способу їх обробки. Для цього система використовує формули, що дозволяють оцінити витратність декількох різних варіантів обробки, після чого вибирає той з них, витратність якого мінімальна. Основна витратність процесу обробки запиту звичайно створюється операціями доступу до диска, які виконуються у багато разів повільніше, ніж власне обробка даних в оперативній пам'яті. Тому значення оцінки витратності реляційних операцій визначатиметься як витратність необхідних операцій доступу до дисків. Кожна оцінка відображатиме необхідну кількість операцій доступу до блоків даних на диску, за винятком операцій запису результуючого набору даних.

Дуже часто оцінка витратності операції залежить від кардинальності відношень, що обробляються. Отже, оскільки необхідно уміти визначати кардинальність створюваних

проміжних відношень розглянемо і деякі типові методи оцінки кардинальності виконуваних операцій.

4.7.1.Статистичні показники бази даних

Достовірність оцінки об'єму і витратності виконання проміжних операцій реляційної алгебри в значній мірі визначається повнотою і актуальністю статистичної інформації, накопичуваної службами СУБД. Як правило, типова СУБД зберігає в своєму системному каталозі статистичні дані, перераховані нижче.

Для кожного базового відношення R:

ntuples(R): - кількість кортежів (записів) щодо R (тобто його кардинальність);

bfactor(R): - показник блокування відношення R (тобто кількість кортежів відношення R, розміщуваних в одному дисковому блоці);

nblocks(R): - кількість блоків вторинної пам'яті, необхідних для збереження відношення R. Якщо кортежі відношення R фізично розміщуються в одній ділянці, то справедлива наступна формула: *nblocks(R)= [ntuples(R)/bfactor(R)]*.

Для кожного атрибута A базового відношення R необхідне визначення:

•*ndistinctA(R):* - кількість унікальних значень атрибута *A* щодо *R;*

•*minA(R),maxA(R):* - мінімальна і максимальна кількість можливих значень атрибута *A* щодо *R;*

•*SCA(R):* -кардинальність вибірки атрибута *A* щодо R.

Воно є середньою кількістю кортежів, які задовольняють умові, завдань для атрибута A. Якщо припустити, що значення атрибута A рівномірно розподілені по відношенню R і що існує принаймні одне значення, що задовольняє заданій умові, те значення *SCA*(R) може бути визначено по наступних формулах: 1 - якщо атрибут A є ключовим атрибутом відношення *R [ntuples(R)/ndistinctA(R)]* - в зворотньому випадку.

Крім того, може бути виконаний оцінка кардинальності вибірки для різних умов. В цьому випадку значення SCA(R) може бути визначено по наступних формулах:

[ntuples(R)((maxA(R)-c)/(maxA(R)-minA(R))] - для (A>c);*

[ntuples(R)((c-maxA(R))/(maxA(R)-minA(R))] - для (A<c);*

*[(ntuples(R)/ndistinctA(R))*n] -для (A in { $c_1,c_{2,\ldots},c_n$});*

SCA(R)SC_B(R) - для (A ^ B)*

SCA(R)+SC_B(R)-SCA(R)SC_B(R) - для (A v B).*

Для кожного багаторівневого індексу I множини значень атрибута A:

nlevelsA(I): - кількість рівнів індексу I;

nlfblocksA(I): - кількість листових блоків індексу I.

Підтримка всіх цих статистичних показників в актуальному стані може бути серйозною проблемою. Якщо СУБД виконуватиме оновлення статистичних показників кожного разу при вставці, оновленні або видаленні кортежу, то це істотно вплине на загальну продуктивність системи в періоди її пікового навантаження. В альтернативному варіанті, який найчастіше використовується на практиці, процедури оновлення статистики виконуються або зі встановленим періодом (наприклад, щоночі), або в моменти простою системи.

Зважаючи на складність цієї проблеми, розглянемо оцінку витратності операцій реляційної алгебри для певного набору операцій. Розглянемо на прикладі операції вибірки

4.7.2.Унарна операція вибірки

Як вже раніше наголошувалося, в реляційній алгебрі операція вибірки виконується над одним відношенням, наприклад, таким як *R*. В результаті її виконання створюється нове відношення *S*, що містить тільки ті кортежі відношення *R*, які задовольнять вказаному предикату *p*. Предикат може бути простим, включаючим єдину операцію порівняння атрибута відношення R з деякою константою або значенням іншого атрибута. Предикат може також бути складовим, включаючим декілька умов, сполучених за допомогою логічних операцій ^ *(AND), v (OR)* і ~ *(NOT).* Існує декілька різних способів реалізації операції вибірки, вибір яких залежить від структури файлу, в якому зберігається відношення, а також від того, якими є що використовуються в предикаті атрибути - індексованими або хешированими. Нижче перераховані основні типи стратегій, з якими ми познайомимося в цьому розділі.

- Лінійний пошук (файл не відсортований, індекс відсутній).
- Двійковий пошук (впорядкований файл, індекс відсутній).
- Пошук по рівності ключа перемішування.
- Пошук по рівності первинного ключа.
- Пошук по нерівності первинного ключа.
- Пошук по рівності значення кластерного (вторинного) індексу.
- Пошук по рівності значення некластерного (вторинного) індексу.
- Пошук по нерівності значення вторинного індексу типу B+-Tree.

Формули розрахунку витратності для всіх вказаних типів стратегій приведені в табл. 4.2

Таблиця 4.2. Зведені дані за оцінкою витратності операцій введення-виведення для різних стратегій виконання операцій вибірки

Стратегія	Оцінка витратності
Лінійний пошук (неврегульований файл без індексу)	*[nblocks(R)/2]* - у разі пошуку по рівності ключового атрибута; *nblocks(R)* - в зворотньому випадку
Двійковий пошук (впорядкований файл без індексу)	*[log2(nblocks(R))]* - у разі пошуку по рівності ключового атрибута; *[log2(nblocks(R))]+[SCA(R)/bfactor(R)]-1* - в зворотньому випадку
Пошук по рівності ключу перемішування	1 - за відсутності переповнювання
Пошук по рівності первинному ключу	*nlevelsA(I)+l*
Пошук по нерівності первинному ключу	*nlevelsA(I)+[nblocks(R)/2]*
Пошук по рівності значенню кластерного (вторинного) індексу	*nlevelsA(I)+[SCA(R)/bfactor (R)*]
Пошук по рівності значенню некластерного (вторинного) індексу	*nlevelsA(I)+ [SCA(R)]*
Пошук по нерівності значенню вторинного індексу типу B+-Tree	*nlevelsA(I)+ [nlfblocksA(I)/2+ntuples(R)/2]*

4.7.3.Оцінка кардинальності операції вибірки

Нижче познайомитися з методами оцінки очікуваної кількості кортежів і очікуваної кількості унікальних значень атрибута в результуючому відношенні S, отриманому при виконання операції вибірки щодо *R*. Оцінки робляться при спрощуючому припущенні про те, що значення атрибута рівномірно розподілені в просторі його домена і незалежні один від одного, то можна скористатися наступними формулами:

ntuples(S)= SCA(R) предикат p має вигляд (*A I x)*

Для будь-якого атрибута B A щодо S значення *ndistinctB(S)* буде дорівнювати: *ntuples(S)* якщо ntuples(S)< ndistinctB(R)/2;

*[(ntuples(S)+ ndistinctB(R))/3] якщо ndistinctB(R)/2 ?ntuples(S) ? 2*ndistinctB(R);*

*ndistinctB(R) якщо ntuples(S)> 2*ndistinctB(R).*

Розглянемо декілька варіантів

Варіант 1. Лінійний пошук (невідсортований файл без індексу)

В даному випадку може бути потрібно виконати сканування кожного кортежу в кожному дисковому блоці з метою перевірки його на відповідність заданому предикату. Цей метод ілюструється алгоритмом, представленим в лістингу 4.1. Якщо пошук виконується по ключовому атрибуту відношення і можна припускати, що кортежі рівномірно розподілені по файлу, то, в середньому, до знаходження необхідного кортежу буде потрібно рахувати половину блоків файлу. Тому формула для оцінки витратності матиме такий вигляд:

[nblocks(R)/2]

Лістинг 4.1. Алгоритм лінійного пошуку

```
//
//Лінійний пошук
// predicate  - це ключ пошуку.
// Файл не відсортований. Блоки послідовно пронумеровані, починаючи з 1.
// Повертається результуюча таблиця, що містить ті кортежі відношення R
// які відповідають предикату.
//
for i = 1 to nblocks(R){       // цикл по всіх блоках файлу
block = read_block(R, i);
for j = 1 to ntuples(block){   // цикл по всіх кортежах в блоці і
 if (block.tuple[j]<задовольняє> predicate)
 then <розміщення кортежу в результуючій таблиці>;
}
}
```

Для будь-яких інших умов можна обґрунтовано прийняти, що буде потрібно рахувати весь файл, тому формула для оцінки витратності матиме такий вигляд:

nblocks(R)

Варіант 2. Двійковий пошук (впорядкований файл без індексу)

Якщо предикат має вигляд (A=x) і файл впорядкований по значеннях атрибута A, який одночасно є ключовим атрибутом відношення R, формула для оцінки витратності операції вибірки матиме наступний вигляд:

[log2(nblocks(R))]

Алгоритм для виконання даного типу пошуку представлений в лістингу 4.2. В більш загальному випадку оцінка витратності може бути виконаний по іншій формулі:

[log2(nblocks(R))] + [SCA(R)/bfactor(R)] - 1

Тут перший терм відображає витратність пошуку першого з вибираних кортежів, для чого використовується метод двійкового пошуку. Очікується, що предикату задовольнятиме SCA(R) кортежів, розташованих в *[SCA(R)/bfactor(R)]* блоках файлу, один з яких вже був лічений при виконанні двійкового пошуку першого кортежу.

Лістинг 4.2. Алгоритм двійкового пошуку у впорядкованому файлі

```
// Двійковий пошук
// predicate - це ключ пошуку.
// Файл впорядкований за збільшенням значень ключового поля A.
// Файл складається з nblocks блоків, послідовно пронумерованих, починаючи з 1.
// Повертається булева змінна (found),  вказуюча, чи був знайдений запис
// яка відповідає предикату, а також результуюча таблиця
// якщо така існує.
//
next = 1; last = nblocks; found = FALSE;
keep searching = TRUE;
while (last >= 1 and (not found) and (keep_searching)) {
                i = (next + last)/2; // Розділення обстежуваної області пополам
                block = read_block(R, i);
                        if (predicate < ordering key_field(first_record(block)))
                        then    // запис в нижній половині обстежуваної області
                                last = 1-1;
                        else if (predicate >  ordering_key_field(last_record(block)))
                        then    // запис у верхній половині обстежуваної області
                                next = i + 1;
                        else if (check_block_for_predicate{block, predicate, result))
                        then      // необхідний запис в даному блоці
                        found = TRUE;
                        else      // необхідний запис відсутній
                                keep_searching = FALSE;
}
```

Варіант 3. Пошук по рівності первинному ключу

Якщо предикат включає умову рівності значенню в полі первинного ключа *(A=x)*, то для вибірки єдиного запису, що задовольняє заданій умові, може використовуватися первинний індекс. В цьому випадку буде потрібно рахувати кількість блоків, на одиницю більше

кількості операцій читання індексу, яке буде дорівнювати кількості його рівнів. Тому формула обчислення оцінки витратності матиме наступний вигляд: *nlevelsA(I)+ 1*

4.7.4.Операція з'єднання

Як вже згадувалося, що однієї з основних проблем, зв'язаних з успішним використанням комерційних СУБД, була і є швидкість виконання запитів. В першу чергу це стосується обробки операцій з'єднання, яка є найдорожчою зі всіх операцій реляційної алгебри (не враховуючи операції декартового добутку). Тому слід докладати всі зусилля для забезпечення виконання цієї операції з максимально можливою продуктивністю. В результаті виконання бінарної операції тета-з'єднання створюється результуюче відношення, яке містить кортежі, що задовольняють заданому предикату F, що вживається до декартового добутку двох початкових відношень - наприклад, таких як R і S. Предикат F визначається у форматі R.aΘS.b, де символ Θ позначає одну з операцій логічного порівняння. Якщо предикат містить тільки операції рівності (=), дане з'єднання називається з'єднанням по еквівалентності. Якщо результат з'єднання включає всі загальні атрибути відношень R і S, з'єднання називається природним з'єднанням. Розглянемо основні типи способи реалізації операцій з'єднання.

- З'єднання блоками з використанням вкладених циклів.
- Індексоване з'єднання блоками з використанням вкладених циклів.
- З'єднання за допомогою сортування-злиття.
- З'єднання за допомогою перемішування.

Оцінки витратності різних стратегій виконання операцій з'єднання приведені в табл. 4.3. Ми почнемо наше обговорення з визначення оцінок кардинальності операцій з'єднання

Таблиця 4.3. Зведені дані за оцінкою витратності операцій введення-виведення для різних стратегій виконання операцій з'єднання

Стратегія	Витратність	Умова
З'єднання блоками з використанням вкладених циклів	*nblocks(R)+(nblocks(R)*nblocks(S))*	Буфер містить тільки один блок для *R і S*
	nblocks(R)+ [nblocks(S)(nblocks(R)/(nbuffer-2))]*	Буфер містить *(nbuffer-*2) блоків для *R*
	nblocks(R)+nblocks(S)	У випадку, якщо всі блоки R можуть бути

		розміщені в буфері
Індексоване з'єднання блоками з використанням вкладених циклів	*Залежить від методу індексації, наприклад:*	
	nblocks(R)+ntuples(R)(nlevelsA(I)+1)*	У випадку, якщо атрибут, що сполучається, A є щодо S первинним ключем
	nblocks(R)+ntuples(R)(nlevelsA(I)+[SCA(R)/bfactor(R)])*	Якщо індекс I - кластерний індекс атрибута A
З'єднання за допомогою сортування-злиття	*nblocks(R)*[log2(nblocks(R)]+ nblocks(S)*[log2(nblocks(S)]*	У разі сортування
	nblocks(R)+nblocks(S)	У разі злиття
З'єднання за допомогою перемішування	*3(nblocks(R)+nblocks(S))+ 2*max_partitions*	У випадку, якщо індекс перемішування міститься в пам'яті
	2(nblocks(R)+nblocks(S)) [lognbuffer-1(nblocks(S))-l] + nblocks(R)+nblocks(S)*	В решті випадків

4.7.5.Оцінка кардинальності операції з'єднання

Кардинальність декартового добуткувідношень R і S, RxS, визначається дуже просто:

ntuples(R)* ntuples(S)

На жаль, визначення кардинальності будь-яких операцій з'єднання є значно складнішою задачею, оскільки воно залежить від розподілу значень атрибутів, що сполучаються. В найгіршому випадку нам відомо, що кардинальність будь-якого з'єднання не може перевищувати кардинальності декартового добутку відношень, що сполучаються, тому:

ntuples(T)< ntuples(R) ntuples(S)*

В деяких системах використовується саме ця верхня межа, проте подібну оцінку слід вважати надмірно песимістичною. Якщо знов припустити, що значення атрибутів в обох відношеннях розподілені рівномірно, то можна таким чином поліпшити оцінку кардинальності з'єднання по еквівалентності з використанням предиката *(R.A = S.B)*.

1. Якщо атрибут *A* є ключовим атрибутом відношення *R*, то кожний кортеж відношення S може бути сполучений тільки з одним кортежем відношення R. Отже, кардинальність з'єднання по еквівалентності не може перевищити кардинальності відношення S: *ntuples(T)< ntuples(S)*
2. Аналогічно, якщо атрибут B є ключовим атрибутом відношення S, то: *ntuples(T)< ntuples(R)*
3. Якщо ні атрибут A, ні атрибут B не є ключовими, то кардинальність з'єднання можна оцінити таким чином: *ntuples(T)= SCA(R)*ntuples(S) або ntuples(T)= SCB(S)*ntuples(R)*

Для отримання першої оцінки використовується той факт, що для будь-якого кортежу s щодо *S* очікується, в середньому, *SCA(R)* кортежів із заданим значенням атрибута *A* і саме ця кількість кортежів відношення *R* братиме участь в з'єднанні. Помноживши це значення на кількість кортежів щодо *S*, ми отримаємо першу з приведених вище оцінок. Аналогічним методом отримана і друга формула.

Варіант 1. З'єднання блоками з використанням вкладених циклів

Найпростішим алгоритмом виконання операції з'єднання є метод вкладених циклів, при якому з'єднання двох відношень виконується по одному кортежу за цикл. В зовнішньому циклі виконується послідовний перебір кортежів першого відношення R, а у внутрішньому циклі послідовно є видимими всі кортежі другого відношення S. Проте, оскільки відомо, що основною одиницею обміну при читанні даних на дискових пристроях є блок, можна поліпшити базовий алгоритм, створивши ще два додаткові цикли, призначені для обробки лічених з диска блоків, що і зроблено в алгоритмі, пропонованому вашій увазі в лістингу 4.3.

Лістинг 4.3. Алгоритм з'єднання блоками з використанням вкладених циклів

```
//
    // З'єднання блоками з використанням вкладених циклів
    //Блоки в обох файлах послідовно пронумеровані, починаючи з 1.
    // Повертається результуюча таблиця з'єднання відношень R і S.
    //
    for iblock = 1 to nblocks(R){       // Зовнішній цикл
    Rblock = read_block(R, iblock);
    for jblock = 1 to nblocks(S){   // Внутрішній цикл
    Sblock = read_block(S, jblock);
    for i = 1 to ntuples(Rblock){
    for j = 1 to ntuples(Sblock){
    if (Rblock.tuple[i]/Sblock.tuple[j]
```

<якщо відповідає умові з'єднання>)
then <розміщує його в результуючу таблицю>;
}
}
}

Оскільки кожний блок відношення *R* повинен бути перелічений хоча б один раз, а кожний блок відношення *S* мав бути порахований для кожного блоку відношення *R*, оцінка витратності цього методу може бути обчислена за формулою:

nblocks(R)+ (nblocks(R) nblocks(S))*

Зауважимо, що значення другого терма цієї формули постійне, тоді як значення першого терма залежить від того, яке з відношень вибране для прочитування в зовнішньому циклі. Очевидно, що в зовнішньому циклі повинне прочитуватися те з двох відношень, що сполучаються, яке займає меншу кількість дискових блоків.

Ще одним поліпшенням даної стратегії можна стати прочитування в буфер бази даних максимально можливої кількості блоків меншого з відношень (скажімо, *R*), із збереженням місця тільки для одного блоку внутрішнього відношення і одного блоку результуючого відношення. Якщо в буфер бази даних може бути поміщений *nbuffer* блоків, то слід зразу ж вважати *nbuffer*-2 блоку відношення *R* і один блок відношення *S.* Загальна кількість прочитуваних блоків відношення R як і раніше залишається рівною *nblocks*(R), проте загальна кількість прочитуваних блоків відношення S зменшується приблизно до *[nblocks(S)*(nblocks(R)/(nbuffer-2))]*. При використанні даного підходу формула обчислення оцінки витратності операції з'єднання набуває наступний вигляд:

nblocks(R)+ [nblocks(S)(nblocks(R)/(nbuffer - 2))]* .

Якщо виявиться можливим рахувати всі блоки відношення *R*, то формула матиме такий вигляд: *nblocks(R)+ nblocks(S).*

Якщо атрибути в з'єднанні по еквівалентності (або природному з'єднанні), що сполучаються, утворюють ключ внутрішнього відношення, то внутрішній цикл може завершуватися, як тільки буде знайдена перша відповідність.

Варіант 2. Індексоване з'єднання блоками з використанням вкладених циклів

Якщо для атрибутів внутрішнього відношення, що сполучаються, існує індекс (або функція перемішування), то можна замінити неефективну операцію сканування файлу більш ефективним пошуком по індексу. Для кожного кортежу відношення R вибірка відповідних кортежів відношення S здійснюється за допомогою індексу. Алгоритм індексованого з'єднання блоками з використанням вкладених циклів представлений в Лістингу 4.4. Для поліпшення читабельності тут приведений спрощений алгоритм, в якому в зовнішньому циклі за один прохід обробляється один блок. Як вже указувалося вище, для підвищення

ефективності доцільно відразу прочитувати в буфер бази даних стільки блоків відношення R, скільки виявиться можливим.

Лістинг 4.4. Алгоритм індексованого з'єднання блоками з використанням вкладених циклів

```
//
// З'єднання відношень R і S по індексованому атрибуту A
// виконуване блоками за допомогою алгоритму вкладених циклів
// Нехай існує індекс I по атрибуту A
// для відношення S
// що містить m рядків I[1], I[2] ... , I[m]
// із значенням кортежу R[i].a, що індексується
// Блоки відношення R послідовно пронумеровані, починаючи з 1.
// Повертається результуюча таблиця з'єднання відношень R і S.
//
for iblock = 1 to nblocks(R){
   Rblock = read_block(R, iblock);
   for i = 1 to ntuples(Rblock){
          for j = 1 to m {
       if (Rblock.tuple[i].A = I[j])
       then <розміщення відповідних кортежів
                        в таблицю результату:»;
          }
   }
}
```

Це значно ефективніший алгоритм з'єднання, що дозволяє уникнути створення декартового добутку відношень R і S як проміжного результату . Витратність сканування відношення R складає nblocks(R), що і в попередньому варіанті. Проте витратність вибірки відповідних кортежів відношень S тепер залежить від типу наявного індексу і кількості відповідних умові кортежів. Наприклад, якщо атрибут, що сполучається, A є щодо S первинним ключем, то витратність операції з'єднання може бути обчислений по наступній формулі:

nblocks(R)+ ntuples(R)(nlevelsA(I)+ 1)*

Якщо атрибут, що сполучається, A щодо S має кластерний індекс, то витратність з'єднання можна обчислити за такою формулою:

nblocks(R)+ ntuples(R)(nlevelsA(I)+ [SCA(R)/bfactor(R)])*

Варіант 3. З'єднання за допомогою сортування-злиття

Для з'єднань по еквівалентності найефективніший спосіб з'єднання досягається у тому випадку, коли обидва відношення відсортовано по атрибутах, що сполучаються. В цьому випадку можна відшукати відповідні кортежі відношень R і S за допомогою злиття двох відношень. Якщо початкові відносини неврегульовані, можна заздалегідь виконати їх сортування. Оскільки кортежі відношень розташовуються в порядку сортування, ті з них, які містять однакові значення атрибута, що сполучається, гарантований розміщуватимуться поряд один з одним. Якщо припустити, що з'єднання виконується для зв'язку типу «багато до багато», тобто в обох відношеннях може існувати декілька кортежів з одним і тим же значенням, що сполучається, а також прийняти, що будь-який набір кортежів з одним і тим же значенням, що сполучається, може бути повністю поміщений в буфер бази даних, то кожний блок кожного з відношень, що сполучаються, буде достатньо рахувати тільки один раз. Отже, оцінка ціни з'єднання методом сортування-злиття може бути обчислений за формулою:

QC=nblocks(R)+ nblocks(S).

Якщо одне з відношень (наприклад, R) повинне бути заздалегідь відсортоване, то у формулу слід додати оцінку витратності операції сортування, яка приблизно може бути виконана за наступною формулою:

QC=nblocks (R)[log2(nblocks(R)].*

Схематично алгоритм з'єднання методом сортування-злиття представлений в лістингу 4.5.

Лістинг 4.5. Алгоритм з'єднання сортуванням-злиттям

```
//
// З'єднання відношень R і S по атрибуту A методом сортування-злиття
// Алгоритм припускає з'єднання типу «багато хто до багато кого».
// Для простоти процедури читання даних опущені.
// Відносини R і S заздалегідь сортуються (цього не вимагається
// якщо файли вже відсортовані по атрибуту, що використовується для з'єднання).
sort(R);
sort(S);
// Виконання злиття
nextR = 1; nextS = 1;
while (nextR <= ntuples(R) and nextS <=ntuples(S)) {
join_value = R.tuples[nextR].A;
// Сканування відношення S до знаходження значення
// меншого ніж поточне, що використовується для з'єднання
while (S.tuples[nextS].A < join_value and nextS <= ntuples(S)) {
nextS = nextS + 1;
}
// Можливо, знайдені відповідні кортежі відношень R і S.
// Кожний кортеж щодо S з деяким значенням join_value
// ставиться у відповідність кожному кортежу щодо R з цим же
// значенням join_value.  (Передбачається з'єднання типу M:N).
while (S.tuples[nextS].A = join_value and nextS <= ntuples(S)) {
```

```
m = nextR;
while (R.tuples[m].A = join_value and m <= ntuples(R)) {
<виведення  відповідних кортежів S.tuples[nextS]
 і R.tuples[m] в таблицю результата>
 m = m + 1;
}
nextS = nextS + 1;
}
// Знайдені всі кортежі відношень R і S
// з одним і тим же значенням join_value.
//  Тепер вибираємо з відношення R кортеж з іншим значенням join_value.
while (R.tuples[nextR].A = join value and nextR <= ntuples(R)) {
nextR = nextR + 1;
}
}
```

Контрольні запитання і вправи.

1. Дайте формальне визначення реляційної моделі даних.
2. Що таке арність відношення? розмірність? ключ?
3. Для чого використовуються ключі?
4. Яке обмеження накладається на типи даних, що є множиною доменів в реляційній моделі даних?
5. Які порушення цілісності даних можуть відбутися при виконанні операції ПРОЕКЦІЯ?
6. В чому суть вимог до підтримки цілісності в реляційних БД ?
7. Для чого потрібно нормалізувати схеми відношень?
8. На чому базується можливість оптимізації схем відношень?
9. В таблиці надані дані про поліклініку, де реєстратура призначає хворих на прийом до лікарів

Таблиця 4.1 Дані про прийом хворих у поліклініці

Таб №	Призвіще лікаря	Номер картки хворого	Прізвище хворого	Дата прийому	Час прийому	Номер кабінету
1-17	Пахомов	Н1001	Циганкова	12.09	8.00	3-15
1-17	Пахомов	Н1021	Новикова	12.09	8.30	3-15
1-20	Комісаренко	Н1008	Матвієнко	12.09	10.00	3-10
1-20	Комісаренко	Н1008	Матвієнко	15.09	14.00	3-10
1-31	Гавриш	Н1020	Бондар	15.09	16.30	3-15
1-31	Гавриш	Н1105	Лисенко	16.09	17.00	3-13

9.1 Проаналізувавши таблицю, виявіть втрату цілісності при вставці, видаленні та при зміні даних.

9.2 Проведіть нормалізацію таблиці до НФБК.

10. Чим відрізняється обробка запитів в реляційних схемах від обробки запитів в схемах графових моделей даних?
11. Які основні стадії обробки запиту?
12. Які основні стадії етапу декомпозиції запиту?
13. Охарактеризуйте показники ефективності запитів.
14. Чи можуть бути ситуації, коли операцію вибірки доцільно виконувати шляхом лінійного пошуку?
15. Якими евристичними правилами користуються для підвищення ефективності обробки запитів?
16. Як можна перевірити семантичну коректність запиту?

РОЗДІЛ 5. Реляційні СУБД як основа сучасних ІС

Всё время схватывая нить
Судеб, событий,
Жить, думать, чувствовать, любить,
Свершать открытья…

(Б. Пастернак)

Системи управління базами даних , в основі яких є реляційна модель баз даних з кінця 80-х років 20-го сторіччя і дотепер є основним елементом переважної більшості інформаційних систем в економіці. Як ми тепер бачимо геніальна ідея Кодда матеріалізувалась в гігантську інформаційну індустрію. Проте, якщо ми порівняємо динаміку просування ієрархічної та мережевої моделі баз даних в реальні інформаційні системи, то освоєння реляційної моделі потребувало як мінімум вдвоє більше часу ніж ієрархічна та мережева. В чому ж причина? Головна з них – це складність реляційної моделі взагалі і складність «вмонтування» її в комп'ютерну технологію зокрема. Перші досягнення знову ж таки пов'язано з розробниками IBM. Проте довга і напружена робота по реальному застосуванню реляційних баз даних почала приносити перші успіхи тільки наприкінці 70-х років, напередодні революції персональних обчислень.

5.1 СУБД стають персональними

Революція персональних обчислень, започаткована на початку 70-х років, докотилася і до СУБД. Справді, а чому ні? Проте спеціалістам з електронної обробки даних якось важко

було повірити, що така солідна справа, як база даних може бути реалізована засобами персонального комп'ютера. Всі серйозні люди, як розробники, так і користувачі, опиралися на потужності IBM 360/System або їм подібних, а тут настільна система, мало не іграшка. Але творчий потенціал адептів персональних обчислень виявися невичерпним. Там, де визнані корифеї вбачали нездоланні перепони, ентузіасти сміливо рушили вперед і незабаром перші СУБД на персональних комп'ютерах почали з'являтися.

Одним з таких піонерів стала компанія Ashton-Tate, яка запропонувала СУБД Dbase II. В 1981 році інженер американського космічного відомства NASA Уейн Ретклiфф у вільний від роботи час став робити просту СУБД, як стверджують, для ведення футбольної статистики. Головним в цій СУБД було те, що вона базувалась на реляційній моделі баз даних. її реалізація була з сучасних позицій досить примітивною, потужності персонального комп'ютера вистачало лише на обробку декілька сот записів. Про жодне промислове використання такого виробу не йшлося , але для побутових потреб цього вистачало. Ретклiфф почав продавати програму під назвою «Вулкан», втім, без особливого успіху. Тому Ретклiфф продав права на програму фірмі Ashton-Tate. Завдяки грамотній маркетинговій політиці компанії цей продукт під назвою dBase II набув чималу популярність. Успіху сприяли як простота використання, так і помірні вимоги до ресурсів комп'ютера. З виходом наступних версій цієї СУБД- dBase III і dBase+ (1986 р.), з більшими можливостями по обробці даних, оснащених досить комфортним на ті часи середовищем розробки і засобами маніпуляції даними, швидко зайняла лідируючі позиції серед настільних СУБД і засобів створення прикладних програм, що їх використовують.

Фізичний рівень в dBase реалізовано на принципі «одна таблиця - один файл» (ці файли звичайно мають розширення *.dbf). MEMO-поля і BLOB-поля (доступні в пізніх версіях dBase) зберігаються в окремих файлах (звичайно з розширенням *.dbt). Індекси для таблиць також зберігаються в окремих файлах. Для цього в ранніх версіях цієї СУБД була потрібна спеціальна операція реіндексування для приведення індексів у відповідність з поточним станом таблиці. Оскільки формат даних dBase є відкритим, це дозволило ряду інших розробників запозичити його для створення dBase-подібних СУБД, частково сумісних з dBase за форматами даних. Наприклад, досить в свій час популярна СУБД FoxBase (розроблена Fox Software, Inc. і нині належить Microsoft) використовувала формат даних dBase для таблиць, проте формати для зберігання MEMO-полів і індексів були своїми власними, несумісними з dBase. Дуже популярний на початку 90-х років засіб розробки Clipper компанії Nantucket Corp (придбаної згодом компанією Computer Associates) маніпулював як з даними формату dBase III (включаючи індексні файли і файли для MEMO-полів), так і з індексними файлами власного формату. Таким чином в кінці 80-х років

виникла нова індустрія СУБД на базі персональних обчислень. Крім цього в їх основі були реляційна модель бази даних (логічний рівень) та формат даних (фізичний рівень).

Більш того, dBase є родоначальником ще одної спільної риси - мови програмування, що отримало назву xBase. Всі мови цього сімейства, що використовуються і в FoxBase, Fox Pro, Clipper, і в деяких більш пізніх засобах розробки, таких як CA Visual Objects фірми Computer Associates, містять схожий набір команд для маніпуляції даними і по суті є мовами-інтерпретаторами. В ролі інтерпретатора команд xBase зазвичай виступає або середовище розробки програми на цій мові, або середовище часу виконання, яке можна поставляти разом з програмою. Фундаментальна ідея мов програмування типу xBase базується на реалізації процедурного підходу до реляційної алгебри, в основі якого лежить навігаційний доступ до елементів даних. В основі ідеї мови xBase лежить один цікавий факт: для прикладної програми користувача дуже важлива обробка не реляційного відношення в цілому, як множини неупорядкованих рядків, а саме кожного окремого рядка. Для цього служать спеціальні функції DB_next, DB_prev та інші, що дозволяють переміщуватися по рядках. При цьому можливий доступ до значень полів тільки одного рядка, того, який є "поточним". Такий підхід до маніпулювання даними, при якому прикладна програма явно розрізняє «попередні» і «подальші рядки» і може управляти переміщенням вказівника від одного рядка до іншого, отримав назву навігаційного (нагадаємо, що навігаційний підхід типовий для мережних і ієрархічних СУБД). Це міркування, очевидно, наштовхнуло Ретклiффа і його послідовників на створення мови для невеликих прикладних програм, розрахованих на одного користувача , в СУБД (dBase, FoxPro, Paradox) на основі навігаційного підходу.

Розглянемо стисло мову xBase, яка підтримується такими СУБД як dBase, FoxPro, Clipper і іншими. Крім власне процедурних операторів (цикл, умова і т.п.) вона включає оператори створення призначеного для користувача інтерфейсу (меню, вікна, екранні форми...). Для роботи з даними служить наступний набір команд:

- *USE <ім'я_таблиці>* - відкрити таблицю. При цьому один рядок таблиці вважається «поточним». До полів «поточного рядка» можна звертатися по їх іменах.
- *GO [TOP BOTTOM]* - зробити «поточною» перший / останній запис таблиці.
- *SKIP* - зробити поточним наступний запис.
- *REPLACE <им'я_колонки> WITH <значення> [,<им'я_колонки> WITH <значення> ...]* - змінити значення полів поточного запису

Існують також команди екранного редагування записів *(EDIT, BROWSE),* пошуку даних в таблиці *(FIND)* і інші. Наведемо приклад невеликої програми на мові xBase, яка роздруковує таблицю *WRITER* , а потім додає в неї новий рядок:

USE WRITER / Відкрити таблицю **writer** */*

GO TOP / Перейти на перший запис */*

DO WHILE .NOT.EOF() / Виконувати цикл поки не буде досягнутий кінець таблиці */*

PRINT WRITER / Надрукувати вміст поля* ***writer*** **/*

SKIP / Перейти на наступний запис */*

ENDDO / Кінець циклу */*

APPEND BLANK / Додати порожній запис. Вона автоматично стає поточною */*

REPLACE AU_ID WITH 90, WRITER WITH «W.Sheakspire» / Змінити значення полів поточного запису */*

Дані персональних СУБД зберігаються в звичайних файлах (як правило, кожна таблиця в окремому файлі), засобів опису обмежень цілісності не існує. Зауважимо, що опис даних відокремлений від самих даних (знаходиться в оброблювальній програмі), і тому не гарантується єдиний спосіб інтерпретації даних для всіх користувачів. Звичайно ж, суттєвою особливістю xBase було врахування фізичного рівня БД(формату DBF, індексів і т.п.)

Відзначимо, проте, що для роботи з даними формату dBase (або інших dBase-подібних СУБД) абсолютно необов'язково користуватися діалектами xBase. Доступ до цих даних можливий за допомогою ODBC API (і відповідних драйверів) і деяких інших механізмів доступу до даних (наприклад, Borland Database Engine, деяких бібліотек інших виробників типу CodeBase фірми Sequenter). А це дозволяє створювати прикладні програми, що використовують формат даних dBase, практично за допомогою будь-якого засобу розробки, що підтримує один з цих механізмів доступу до даних. Але докладніше ми поговоримо про це пізніше.

Після придбання dBase компанією Borland цей продукт, що отримав згодом назву Visual dBase, отримав набір додаткових можливостей, характерних для засобів розробки цієї компанії. Серед цих можливостей були спеціальні типи полів для графічних даних, підтримувані індекси, зберігання правил посилальної цілісності усередині самої бази даних, а також можливість маніпулювати даними інших форматів, зокрема серверних СУБД, за рахунок використання BDE API і SQL Links. Проте не всі настільні СУБД орієнтувались на dBase. Так, наприклад, перша версія Paradox була розроблена компанією Ansa Software в 1985 році. Цей продукт був згодом придбаний компанією Borland. З 1996 року він вже належить компанії Corel і є складовою частиною Corel Office Professional. Принцип зберігання даних в Paradox схожий з принципами зберігання даних в dBase - кожна таблиця зберігається в окремому файлі (розширенням *.db), MEMO- і BLOB-поля зберігаються в спеціальному файлі (розширення *.md), як і індекси (розширення *.px). Проте, на відміну від dBase, формат даних Paradox не є відкритим, тому для доступу до даних цього формату потрібні спеціальні бібліотеки. Наприклад, в програмах, написаних на C або Pascal, використовувалася популярна колись бібліотека Paradox Engine, яка стала основою Borland Database Engine. Ця бібліотека використовується нині в програмах, створених за допомогою

засобів розробки Borland (Delphi, C++Builder), в деяких генераторах звітів (наприклад, Crystal Reports) і в самому Paradox. Існують і ODBC-драйвери до баз даних, створених різними версіями цієї СУБД. Відзначимо, що відсутність «відкритості» формату даних має і свої переваги. Оскільки в цій ситуації доступ до даних здійснюється тільки за допомогою спеціальних бібліотек, просте редагування подібних даних в порівнянні з даними відкритих форматів типу dBase істотно ускладнено. В цьому випадку можливі такі недоступні при використанні «відкритих форматів» даних можливості, як захист таблиць і окремих полів паролем, зберігання деяких правил посилальної цілісності в самих таблицях - всі ці сервіси надаються Paradox, починаючи з перших версій цієї СУБД.

В порівнянні з аналогічними версіями dBase ранні версії Paradox звичайно надавали розробникам баз даних істотно більш розширені можливості, такі як використання ділової графіки в DOS-програмах, оновлення даних в програмах при розрахованій на багато користувачів роботі, візуальні засоби побудови запитів, на основі інтерфейсу QBE - Query By Example (запит за зразком), засоби статистичного аналізу даних.

Парадигма персональних обчислень породила ще один надзвичайно важливий напрямок електронної обробки даних - їх інтеграцію. Фактично до початку 90-х років, користувач персонального комп'ютера стикався з типовою проблемою зв'язку між різними програмами. Пояснимо її на прикладі. Нехай готується звіт про витрати коштів на заробітну платню співробітників відділу. Перше, що було слід зробити - завантажити СУБД і вибрати дані про потрібних співробітників. Ці дані роздруковувалися, а програма вивантажувалася, оскільки операційна система DOS була однозадачною. Після цього запускалася електронна таблиця і в неї вручну вносилися дані з роздруківки (буфера обміну в DOS не було). Виконавши розрахунки на таблиці, її потрібно було знову роздрукувати, щоб потім, після того, як буде завантажений текстовий процесор, ввести в потрібне місце остаточного документа. Така тяганина з передачею даних від програми до програми природним чином привела до ідеї створення інтегрованої системи, яку б користувач мав застосовувати з мінімумом проміжних ручних операцій. Елементи інтеграції були закладені ще в Lotus 1-2-3, де можна було, не виходячи з програми, скористатися простим текстовим редактором і подивитися як ведуть себе дані на графіку. Цю ідею Lotus Development постаралася розвинути в подальших розробках. В 1984 році для Macintosh була випущена система Jazz, а для PC - Symphony. Самі назви цих пакетів натякали на ансамбль можливостей: в рамках однієї системи можна було використовувати текстові документи, електронні таблиці, бази даних і ділову графіку. Хоча повторити феноменальний успіх Lotus 1-2-3 новим пакетам не вдалося, вони заклали основу майбутніх наборів офісних програм. Слідом за продуктами Lotus Development на ринку почали з'являтися інтегровані системи інших розробників. При цьому винаходилися різні «коктейлі»: окрім стандартного набору (текст, таблиці, СУБД, ділова графіка), в систему

включалися найрізноманітніші функції. Наприклад, відома нам фірма Ashton-Tate вийшла на ринок в тому ж 1984 році з дуже цікавим продуктом Framework. Ще до появи Windows, під управлінням DOS, Framework моделював графічне віконне середовище. Документи розташовувалися на робочому столі, збільшувалися і зменшувалися в розмірах, згорталися в значки, ховалися в папках і шафах. Але найголовнішою перевагою пакету була потужна, схожа на Lisp, мова програмування Fred, яка дозволяла розробляти в середовищі Framework різноманітні прикладні системи. Серед інших розробок слід відзначити оригінальний пакет Guru, розроблений в 1986 році фірмою Micro DataBase Systems Inc. В нього була вбудована система логічного виведення, за допомогою якої можна було створювати різні експертні системи. Але це був тільки початок, справжня інтеграція була попереду.

5.2 Становлення SQL

Таким чином, на протязі 80-х років фактично виникла нова індустрія обробки даних, яка почавшись з ідеї персонального, домашнього використання стрімко дійшла до рівня професійних бізнесових застосувань. Як ми бачили це стосувалося і центрального питання обробки електронної даних – СУБД. Але для інформаційних систем локальні персональні обчислення малоефективні – вимагалась технологія спільної колективної роботи. Для розробників інформаційних систем це і на той час було азбучною істиною і вони добре знали як це робиться в традиційній технології мейнфреймів. Спеціалістам було зрозуміло, що замінити технологію мейнфреймів персональними комп'ютерами можна тільки об'єднавши їх у мережу. На той час СУБД dBASE компанії Ashton-Tate була інстальована більш ніж на мільйоні PC, що працювали під управлінням MS-DOS; інші продукти, такі як R-BASE, PFS: File і Paradox, також досягли значного успіху. На комп'ютерах сімейства Macintosh такі СУБД, як 4th Dimension, об'єднали в собі управління даними і графічний інтерфейс користувача. Хоча в більшості СУБД для персональних комп'ютерів дані зберігалися в табличній формі, ці СУБД ще не могли повністю використати потужність реляційної бази даних, не в малій мірі із-за того, що не підтримували SQL. До кінця 80-х мало відомо про використання SQL на персональних комп'ютерах, хоча на той час звичайним явищем стали персональні комп'ютери, що підтримували дискові пристрої об'ємом в десятки і сотні мегабайтів. Проте, коли користувачі почали об'єднувати персональні комп'ютери в мережі, тоді і з'явилася необхідність в спільному використанні даних. Звичайно, це потребувало значних зусиль розробників в плані системного і мережевого забезпечення.

Перші СУБД для персональних комп'ютерів були відповідним чином переробленими версіями відомих СУБД для мінікомп'ютерів і ледве вміщувалися на персональних комп'ютерах. Система Professional Oracle, анонсована в 1984 році, вимагала двох мегабайтів

оперативної пам'яті на IBM PC, а Oracle for Macintosh, представлена в 1988 році, мала схожі вимоги. Версія СУБД Ingres для PC, випущена в 1984 році, ледве задовольняла обмеженню MS-DOS на об'єм оперативної пам'яті, що використовується (640 Кб). СУБД Informix-SQL для MS-DOS була випущена в 1986 році і була версією популярної СУБД, що працювала під управлінням UNIX. В тому ж 1986 році компанія Gupta Technologies, заснована колишнім менеджером з Oracle, випустила SQLBase, СУБД для локальних мереж в архітектурі клієнт/сервер і була прототипом нинішніх СУБД для локальних мереж.

З появою в квітні 1987 року операційної системи OS/2, створеної компаніями Microsoft і IBM, почалося зростання популярності SQL на платформі персональних комп'ютерів. Окрім стандартної версії OS/2, компанія IBM випустила розширену редакцію OS/2 (OS/2 Extended Edition - OS/2 II) з вбудованою підтримкою реляційних баз даних. Зробивши SQL частиною операційної системи(дивовижне явище, що і через 20 років продовжує дивувати), компанія IBM тим самим знов підтвердила свою прихильність до цієї мови.

Поява такої версії OS/2 занепокоїла Microsoft. Оскільки вона була розробником стандартної OS/2 і продавала її іншим виробникам персональних комп'ютерів, була потрібна альтернатива розширеній OS/2, якою володіла IBM. Відповіддю Microsoft стала купіля ліцензії на СУБД компанії Sybase, розробленої для системи VAX (чудової серії мінімашин компанії DEC), і перенесення цієї СУБД в систему OS/2, трохи згодом її назвали MS SQL Server. Хоча успіх SQL Server для OS/2 був обмеженим, вона продовжує грати ключову роль в планах компанії Microsoft. Зрештою ця СУБД стала реляційною базою даних для операційних систем типу Windows NT, флагманської операційної системи компанії Microsoft, призначеної для роботи в середовищі клієнт/сервер. З розвитком комп'ютерних мереж персональні СУБД стали переходити в розряд багатокористувацьких. При цьому файли даних розміщувалися на мережному диску, який мав здатність розділятися між запитами прикладних програм багатьох користувачів. Проте, створення достатньо великих прикладних програм (10-20 одночасно працюючих користувачів) показало, що в цьому випадку різко знижується продуктивність і виникають проблеми з підтримкою цілісності. Мова xBASE, як чудовий інструмент для локальної версії PC, де особливо важливі ресурсні обмеження, не був пристосований для мережевої технології.

Реляційна модель даних, що містить набір чітких вимог до базової організації будь-якої реляційної системи управління базами даних (СУБД), дозволяє користувачам працювати в ненавігаційній манері, тобто для вибірки інформації з БД користувач повинен лише вказати список таблиць, що його цікавлять і ті умови, яким повинні задовольняти відібрані дані. СУБД приховує від користувача виконувані нею послідовні перегляди таблиць, виконуючи їх найефективнішим чином. Дуже важлива особливість реляційних систем полягає в тому, що результатом виконання будь-якого запиту до таблиць БД є також таблиця, яку можна

зберегти в БД і по відношенню до якої можна виконувати нові запити. В результаті персональні комп'ютери стали потребувати засобу, який міг би задовольнити реляційні бази даних і адекватну мову обробки даних. Такою мовою і стала мова SQL - Structured Query Language (іноді її сприймають як Standard Query Language).

Що ж вона собою являє? До появи SQL в СУБД (незалежно від того, на якій моделі вони ґрунтувалися) доводилося підтримувати принаймні три мови, у яких зазвичай було мало спільного: мова визначення даних (МВД), необхідна для специфікації структур БД (звичайно загальну структуру БД називають схемою БД); мова маніпулювання даними (ММД), що дозволяє створювати прикладні програми, які взаємодіють з БД; і мова адміністрування БД (МАБД), за допомогою якого можна було виконувати службові дії (наприклад, змінювати структуру БД або проводити її налаштування з метою підвищення ефективності). Крім того, якщо користувачам СУБД треба надати інтерактивний доступ до БД (зараз це просто предмет першої необхідності), доводилося вводити ще одну мову, оператори якої виконуються в діалоговому режимі. Мова SQL дозволяє вирішувати всі ці проблеми системно. Вже в піонерських роботах Едгара Кодда намітились базові підходи до розробки такої мови для реляційних баз даних, яку він назвав Альфа-мовою. Ідея цієї мови базувалась на реляційному численні кортежів. З погляду чистої теорії це була одна з можливих альтернатив. В Розділі IV були розглянуті деякі інші можливі підходи, наприклад табло, які являють собою спеціалізовані матриці, колонки яких відповідають атрибутам відповідної схеми бази даних, або реляційне числення на доменах, що знайшло своє вираження в в мові QBE (Query by Example). Але тільки SQL судилося стати мовою сучасної індустрії електронної обробки даних.

Мова SQL, призначена для взаємодії з базами даних, з'явилася в середині 70-х рр. XX-го століття і була розроблена в компанії IBM в рамках проекту експериментальної реляційної СУБД System R. Треба зазначити, що цей проект, який хоч і не привів до створення якоїсь конкретної СУБД (принаймні це сталося значно пізніше, і не IBM була першою, а Oracle) мав надзвичайно велике значення для подальшого розвитку і впровадження SQL. Заради справедливості треба згадати й той факт, що проект System R був не єдиним дослідницьким проектом в цьому напрямку- зокрема немало досягли в цій сфері і дослідники з університету Берклі. Початкова назва мови SEQUEL *(Structured English Query Language)* тільки частково відображала суть цієї мови. Звичайно, мова була орієнтована головним чином на зручне і зрозуміле користувачам формулювання запитів до реляційних БД. Але, насправді, вона майже із самого початку була повною мовою БД, що забезпечує крім засобів формулювання запитів і маніпулювання БД наступні можливості:

- засоби визначення і маніпулювання схемою БД;

- засоби визначення обмежень цілісності;
- засоби визначення представлення БД;
- засоби визначення структур фізичного рівня, що підтримують ефективне виконання запитів;
- засоби авторизації доступу до відношень і їх полів;
- засоби визначення точок збереження транзакції і виконання фіксації і відкотів транзакцій.

В мові були відсутні засоби явної синхронізації доступу до об'єктів БД з боку паралельно виконуваних транзакцій: із самого початку передбачалося, що необхідну синхронізацію неявно виконує СУБД. В даний час мова SQL реалізована у всіх комерційних реляційних СУБД і майже у всіх СУБД, які спочатку ґрунтувалися не на реляційному підході. Всі компанії-виробники проголошують відповідність своєї реалізації стандарту SQL, і насправді реалізовані діалекти SQL дуже близькі. Цього вдалося досягнути не відразу. Найбільш близькі до System R були дві системи компанії *IBM* – SQL/DS і DB2. Розробники обох систем використовували досвід проекту System R*, а СУБД* SQL/DS прямо ґрунтувалася на програмному коді *System R*. Звідси гранична близькість діалектів SQL, реалізованих в цих системах, до SQL System R.

Інший підхід застосовувався в таких системах, як Oracle*,* Informix *і* Sybase. Незважаючи на відмінність в способах розробки систем, реалізація *SQL* скрізь відбувалася з низу до верху. По-перше, у випущених на ринок версіях цих систем використовувалася обмежена підмножина SQL *System R*. Зокрема, в першій відомій нам реалізації SQL *в СУБД* Oracle в операторах вибірки не допускалося використання вкладених підзапитів і була відсутня можливість формулювання запитів із з'єднаннями декількох відношень.

Проте, не дивлячись на ці обмеження і на дуже слабку, на перших порах, ефективність СУБД, орієнтація компаній на підтримку різних апаратних платформ і зацікавленість користувачів в переході до реляційних систем дозволили компаніям добитися комерційного успіху і перейти до вдосконалення своїх реалізацій. В поточних версіях Oracle, Informix*,* Sybase *і* Microsoft SQL Server підтримуються достатньо потужні діалекти SQL. Особливістю більшості сучасних комерційних СУБД є відсутність одноманітного опису мови, що утрудняє порівняння існуючих діалектів SQL. Зазвичай опис розкиданий по різних документах. Проте можна сказати, що базовий набір операторів SQL, який включає оператори визначення схеми БД, вибірки і маніпулювання даними, авторизації доступу до даних, підтримка вбудовування *SQL* в мови програмування і оператори динамічного SQL, в комерційних реалізаціях більш-менш відповідає стандарту.

Діяльність по стандартизації мови SQL почалася практично одночасно з появою іі перших комерційних реалізацій. Перший документ датований жовтнем 1985 р. і є вже не першим проектом стандарту ANSI. Стандарт був прийнятий ANSI в 1986 р., а в 1987 р. схвалений

Міжнародною організацією по стандартизації (ISO). Цей стандарт прийнято називати SQL/86.

За основу був узятий діалект мови SQL, що склався в IBM до початку 1980-х рр. По-суті, цей діалект був підмножиною *SQL* System R, яка була технічно відпрацьована. До 1989 р. стандарт SQL/86 був дещо розширений і прийнятий наступний стандарт, що отримав назву ANSI/ISO SQL/89. Аналіз доступних документів показує, що процес стандартизації *SQL* відбувався дуже складно з використанням не тільки наукових аргументів. В цьому стандарті повністю відсутні такі важливі розділи, як маніпулювання схемою БД і динамічний SQL. Можливо, найважливішими досягненнями стандарту SQL/89 є чітка стандартизація синтаксису і семантики операторів вибірки даних і маніпулювання даними і фіксація засобів обмеження цілісності БД. Були специфіковані засоби визначення первинного і зовнішніх ключів відношень і перевірочних обмежень цілісності, які є підмножиною обмежень цілісності SQL System R, що негайно перевіряються. Засоби визначення зовнішніх ключів дозволяють легко формулювати вимоги так званої посилальної цілісності БД. Проте розробники розуміли, що стандарт SQL ще далекий від досконалості. Тому фахівці різних компаній продовжили цю роботу під прапором розробки нового стандарту SQL2. Ця робота також тривала декілька років, було випущена немало варіантів стандарту, доки нарешті в березні 1992 р. не був прийнятий остаточний проект стандарту SQL/92. Цей стандарт істотно повніший за стандарт *SQL/89* і охоплює практично всі аспекти, необхідні для реалізації прикладних програм: маніпулювання схемою БД, управління транзакціями (з'явилися точки збереження) і сесіями (сесія – це послідовність транзакцій, в межах якої зберігаються тимчасові відношеньи), підключення до БД, динамічний SQL.

В 1995 р. стандарт був доповнений специфікацією інтерфейсу рівня виклику (Call-Level Interface – SQL/CLI). SQL/CLI є набором специфікацій інтерфейсів процедур, виклики яких дозволяють виконувати оператори SQL, що динамічно визначаються. По суті, SQL/CLI є альтернативою динамічному SQL. Інтерфейси процедур визначені для всіх основних мов програмування: C++, Ada, Pascal і т.д. Слід зауважити, що стандарт SQL/CLI послужив основою для створення скрізь поширених сьогодні інтерфейсів ODBC (Open Database Connectivity) і JDBC (Java Database Connectivity).

В 1996 р. до стандарту SQL/92 був доданий ще один компонент – SQL/PSM (Persistent Stored Modules). Основна мета цієї специфікації полягає в тому, щоб стандартизувати способи визначення і використання збережених процедур, тобто спеціальним чином оформлених програм, які включають оператори SQL, що зберігаються в базі даних і можуть викликатися прикладними програмами для виконання в середині СУБД. Незадовго до завершення робіт за визначенням стандарту SQL2 була почата розробка стандарту SQL3. Роботу над новим

стандартом вдалося частково завершити тільки в 1999 р., і тому стандарт отримав назву SQL:1999.

Перш за все, зауважимо, що кожний новий варіант стандарту мови SQL був істотно більшим попередніх версій. Так, якщо стандарт SQL/89 займав близько 600 сторінок, то об'єм SQL/92 складав на 300 із гаком сторінок більше. Найперші проекти SQL3 займали близько 1500 сторінок. Це цілком природньо, тому що мова ускладнюється, а її специфікації стають більш детальними і точними. Саме складність змусила розробників вдатися до модульного принципу розробки, тобто розбити стандарт на відносно незалежні частини, які можна було б розробляти і підтримувати окремо. В 1999 р. були прийнято п'ять перших частин стандарту SQL:1999. Перша частина (SQL/Framework) присвячена опису концептуальної структури стандарту. В цій частині приводиться розгорнена анотація наступних чотирьох частин і формулюються вимоги до реалізацій, що претендують на відповідність стандарту.

Друга частина SQL:1999 (SQL/Foundation) утворює базис стандарту. Вводиться система типів мови, формулюються правила визначення функціональної залежності і можливих ключів, визначаються синтаксис і семантика основних операторів SQL:

- операторів визначення і маніпулювання схемою бази даних;
- операторів маніпулювання даними;
- операторів управління транзакціями;
- операторів управління підключеннями до бази даних і т.д.

Третю частину займає уточнена в порівнянні з SQL/92 специфікація *SQL/CLI*. В четвертій частині специфікується SQL/PSM – синтаксис і семантика мови визначення збережених процедур. Нарешті, в п'ятій частині – SQL/Bindings – визначаються правила зв’язування *SQL* для стандартних версій мов програмування FORTRAN, COBOL, PL/1, Pascal, Ada, З і MUMPS. В стандарт SQL:1999 повинні були ввійти ще декілька частин. Серед них специфікації наступних засобів:

- управління розподіленими транзакціями (*SQL/Transaction*);
- підтримка темпоральних властивостей даних (SQL/Temporal);
- управління зовнішніми даними (SQL/MED);
- зв’язок з об'єктно-орієнтованими мовами програмування (SQL/OLB);
- підтримка оперативної аналітичної обробки (SQL/OLAP).

В кінці 2003 р. був прийнятий і опублікований новий варіант міжнародного стандарту SQL:2003. Багато фахівців вважали, що у варіанті стандарту, наступному за *SQL:1999*, будуть всього лише виправлені неточності SQL:1999. Але насправді, в *SQL:2003* специфікований ряд нових і важливих властивостей. Зазнала деякі зміни загальна організація стандарту. Стандарт SQL:2003 складається з наступних частин:

- 9075-1, SQL/Framework;
- 9075-2, SQL/Foundation;
- 9075-3, SQL/CLI;
- *9075-4, SQL/PSM;*
- *9075-9, SQL/MED;*
- *9075-10, SQL/OLB;*
- *9075-11, SQL/Schemata;*
- *9075-13, SQL/JRT;*
- *9075-14, SQL/XML.*

Частини 1-4 і 9-10 з необхідними змінами залишилися такими ж, як і в SQL:1999 . Частина 5 (SQL/Bindings) перестала існувати; відповідні специфікації включені в частину 2. Розділ частини 2 SQL*:1999,* присвячений інформаційній схемі, виділений в окрему частину 11. Поточний стан процесу стандартизації мови SQL відображає поточний стан технології SQL-орієнтованих баз даних. Провідні постачальники відповідних СУБД (сьогодні це компанії IBM, Oracle і Microsoft) прагнуть максимально швидко реагувати на потреби і кон'юнктуру ринку і розширюють свої продукти всі новими і новими можливостями. Очевидна потреба в стандартизації відповідних мовних засобів, але процес стандартизації явно не встигає за змінами, що відбуваються.

5.3 Структура мови SQL

Спочатку ми опишемо базові механізми мови SQL. Щоб пояснити, про яку частину мови йтиметься , звернемося до Рис. 5.1.

На даний момент мова *SQL*, що відповідає останнім стандартам SQL:2003*,* SQL:1999 *(і навіть* SQL/92), є дуже багатою і складною мовою. Тому доводиться розбивати мову на рівні, або шари, такі, що кожний рівень мови включає всі конструкції, що входять в більш низькі рівні. В стандарті визначається декілька способів розбиття мови на рівні. В одній з класифікацій мова розбивається на «базовий (entry), «проміжний» (intermediate) і «повний» (full) рівні»[].

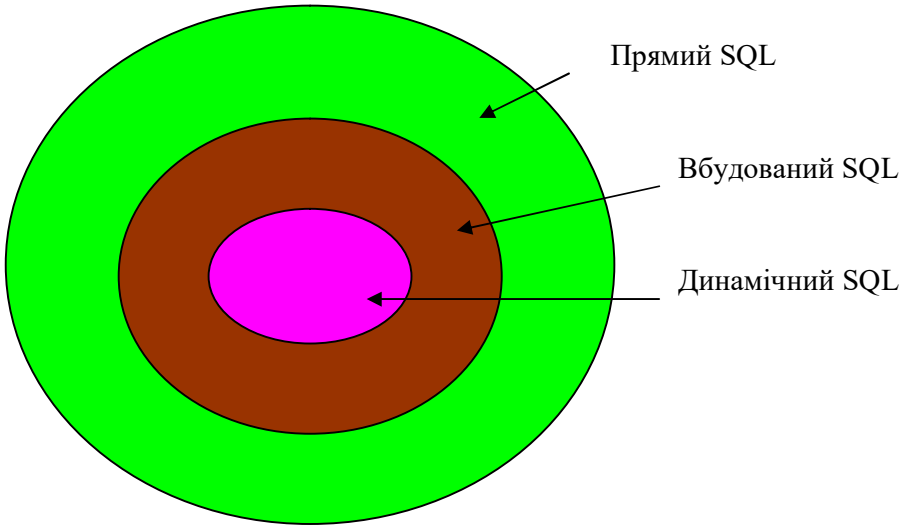


Рис. 5.1. Один із способів поділу мови SQL на рівні

Ця класифікація орієнтована, перш за все, на виробників СУБД, в яких підтримується SQL. Реалізація базового рівня мови є обов'язковою умовою хоча б якоїсь відповідності стандарту. Реалізація проміжного рівня бажана, і звичайно саме такий рівень мови підтримується провідними компаніями-виробниками SQL-орієнтованих СУБД. Нарешті, повний рівень мови є метою, до досягнення якої слід прагнути. В даній класифікації критерієм віднесення тієї або іншої можливості мови до певного рівня є оцінювана розробниками стандарту *SQL* (більшість яких є співробітниками провідних компаній, що проводять SQL-орієнтовані СУБД) технічна складність реалізації цієї можливості. Звичайно, така класифікація важлива і для програмістів прикладних програм баз даних, але тільки для того, щоб оцінити реальні можливості конкретної СУБД. Для розуміння мови *SQL* таке розбиття на рівні неістотне.

Інша класифікація показана на Рис. 5.1. Серед всіх конструкцій мови SQL можна виділити такі конструкції, які можна було б використовувати при «прямій (direct) взаємодії» кінцевого користувача з СУБД (наприклад, в інтерактивному режимі). В певному розумінні цей рівень також є базовим, оскільки відповідні засоби мови найбільшою мірою відображають його орієнтованість на роботу з множинами. На наступному рівні, рівні «вбудовуваного» (embedded) SQL, мова розширюється конструкціями, що дозволяють використовувати можливості прямого SQL в програмах, написаних на традиційних мовах програмування. Нарешті, на рівні «динамічного» (dynamic) SQL у вбудовуваний SQL

додаються конструкції, що дозволяють прикладним програмам звертатися до СУБД з конструкціями прямого SQL, які динамічно утворюються під час виконання програми.

Звернемося до прямого SQL, причому не в повному об'ємі стандартів SQL:2003 і SQL:1999 (цього не дозволяє зробити обсяг посібника). Обговоримо тільки найважливіші аспекти. Зокрема це стосується типів даних.

Дані, що зберігаються в колонках таблиць SQL-орієнтованої бази даних, є значеннями одного з типів даних, визначених в мові SQL або визначених користувачами шляхом застосування відповідних засобів мови. Для цього при визначенні таблиці кожній її колонці призначається певний тип даних (або домен), і надалі СУБД повинна стежити, щоб в довільній колонці кожного рядка будь-якої таблиці були присутні тільки допустимі значення. Всі можливі в SQL типи даних, які можна використовувати при визначенні колонок, поділяються на наступні категорії:

- точні числові типи ;
- наближені числові типи ;
- типи символьних рядків зокрема CHAR або CHAR(n) -символьні рядки фіксованої довжини. Довжина рядка визначається параметром n. CHAR без параметра відповідає CHAR(1). Для зберігання таких даних завжди відводиться n байт незалежно від реальної довжини рядка. VARCHAR(n) - символьний рядок змінної довжини. Для зберігання даних цього типу відводиться число байт, відповідне реальній довжині рядка;
- типи бітових рядків - дозволяють зберігати дані будь-якого об'єму в двійковому коді (оцифровані зображення, виконувані файли і т.п.). Визначення цих типів найбільш розрізняються від СУБД до СУБД , часто використовуються ключові слова:

  BINARY, BYTE, BLOB
- типи дати і часу DATE - тип даних для зберігання дати. TIME - тип даних для зберігання часу. INTERVAL - тип даних для зберігання часового інтервалу. DATETIME - тип даних для зберігання моментів часу (рік + місяць + день + години + хвилини + секунди + частки секунд);
- типи часових інтервалів;
- булевий тип ;
- типи колекцій;
- анонімні рядкові типи;
- типи, визначувані користувачем;
- посилальні типи.

В колонках таблиць, визначених на будь-яких типах даних, разом із значеннями цих типів, допускається збереження невизначеного значення, яке позначається ключовим словом

NULL. Це значення має кожний елемент колонки доти, поки в нього не будуть введені дані. При створенні таблиці можна явно вказати СУБД чи можуть елементи тієї чи іншої колонки мати значення NULL (це неможливо, наприклад, для колнки, яка є первинним ключем).

5.3.1. Мова визначення даних (МВД)

Мова виззначення даних (Data Definition Language скорочено DDL) містить оператори, що дозволяють створювати, змінювати і знищувати бази даних і об'єкти, що в них знаходяться (таблиці, уявлення і ін.). Ці оператори перераховані в табл.

Таблиця 5.1 Оператори мови визначення даних SQL

Оператор	Опис
CREATE TABLE	Застосовується для додавання нової таблиці до бази даних
DROP TABLE	Застосовується для видалення таблиці з бази даних
ALTER TABLE	Застосовується для зміни структури наявної таблиці
CREATE VIEW	Застосовується для додавання нового уявлення до бази даних
DROP VIEW	Застосовується для видалення уявлення із бази даних
CREATE INDEX	Застосовується для створення індексу для даного поля
DROP INDEX	Застосовується для видалення існуючого індексу
CREATE SCHEMA	Застосовується для створення нової схеми в базі даних
DROP SCHEMA	Застосовується для видалення схеми із бази даних
CREATE DOMAIN	Застосовується для створення нового домена
ALTER DOMAIN	Застосовується для перевизначення домена
DROP DOMAIN	Застосовується для видалення домена з бази даних

5.3.2. Створення схеми бази даних.

При описі команд передбачається, що:

- текст, набраний рядковими буквами (наприклад, *CREATE TABLE*) є обов'язковим
- текст, набраний прописними буквами і поміщений в кутові дужки (наприклад <им'я_бази_даних>) позначає змінну, що вводиться користувачем
- в квадратні дужки (наприклад [NOT NULL]) розміщується необов'язкова частина команди
- елементи команди, що взаємовиключаються, розділяються вертикальною рискою (наприклад [UNIQUE PRIMARY KEY]).

Оператори бази даних

Команда	**Опис**
CREATE DATABASE <ім'я_бази_даних>	*Створення бази даних.*
DROP DATABASE <ім'я_бази_даних>	*Видалення бази даних.*

Створення і видалення таблиць

Створення таблиці:

```
CREATE TABLE <ім'я_таблиці>
   (<ім'я_колонки> <тип_колонки>
            [NOT NULL]
            [UNIQUE  PRIMARY KEY]
            [REFERENCES <ім'я_мастер_таблиці> [<ім'я_колонки>]]
          ,  . . . )
```

Користувач зобов'язаний вказати ім'я таблиці і список колонок. Для кожної колонки обов'язково вказують її ім'я і тип , а також опціонально можна вказати параметри

- *NOT NULL* - в цьому випадку елементи колонки завжди повинні мати певне значення (не *NULL*)
- один з параметрів UNIQUE, що взаємовиключають, - значення кожного елемента колонки повинне бути унікальним або PRIMARY KEY - колонка є первинним ключем.
- REFERNECES <ім'я_мастер_таблиці> [<ім'я_колонки>] - ця конструкція визначає, що даний колонка є зовнішнім ключем і указує на ключ якої мастер_таблиці він посилається.

Контроль виконання вказаних умов здійснює СУБД.
Приклад: створення бази даних publications:

```
CREATE DATABASE publications;

CREATE TABLE writers (au_id INT PRIMARY KEY
            author VARCHAR(25) NOT NULL);
CREATE TABLE publishers (pub_id INT PRIMARY KEY
             publisher VARCHAR(255) NOT NULL,url VARCHAR(255));
CREATE TABLE titles (title_id INT PRIMARY KEY
            title VARCHAR(255) NOT NULL
            yearpub INT
            pub_id INT REFERENCES publishers(pub_id));
CREATE TABLE titleautors (au_id INT REFERENCES writers(au_id)
              title_id INT REFERENCES titles(title_id));
CREATE TABLE wwwsites (site_id INT PRIMARY KEY
             site VARCHAR(255) NOT NULL
             url VARCHAR(255));
CREATE TABLE wwwsitewriters (au_id INT REFERENCES writers(au_id)
                site_id INT REFERENCES wwwsites(site_id));
```

Видалення таблиці:

```
DROP TABLE <ім'я_таблиці>
```

Модифікація таблиці:

Додати колонки

```
ALTER TABLE <ім'я_таблиці> ADD
            (<ім'я_колонки> <тип_колонки>
                  [NOT NULL]
            [UNIQUE  PRIMARY KEY]
           [REFERENCES <ім'я_мастер_таблиці>
      [<ім'я_колонки>]]
                  ...)
```

Видалити колонки

```
ALTER TABLE <ім'я_таблиці> DROP (<ім'я_колонки>...)
```

Модифікація типу колонки

```
ALTER TABLE <ім'я_таблиці> MODIFY
            (<ім'я_колонки> <тип_колонки>
```

[NOT NULL]
[UNIQUE PRIMARY KEY]
[REFERENCES
<ім'я_майстер_таблиці> <ім'я_колонки>]]
...)

5.3.3. Оператори створення індексів.

Створення індексу**:**

CREATE [UNIQUE] INDEX <ім'я_індексу> ON <ім'я_таблиці> (<ім'я_колонки>...)

Ця команда створює індекс із заданим ім'ям для таблиці *<ім'я_таблиці>* по колонках, що входять в список, вказаний в дужках. Як вже зазначалося, індекс часто являє собою структуру типу B-дерева , але можуть використовуватися і інші структури. Створення індексів значно прискорює роботу з таблицями. У разі вказівки необов'язкового параметра UNIQUE СУБД перевірятиме кожне значення індексу на унікальність.

Дуже часто постає питання, які поля необхідно індексувати. Обов'язково треба будувати індекси для первинних ключів, оскільки по їх значеннях здійснюється доступ до даних при операціях з'єднання двох і більше таблиць. Також у відповіді на це питання допоможе аналіз найчастіших запитів до бази даних. Наприклад, для *БД publications* можна чекати, що одним з найчастіших запитів буде вибірка всіх публікацій даного письменника. Для мінімізації часу цього запиту необхідно побудувати індекс для таблиці writers за іменами авторів:

CREATE INDEX au_names ON writers (author);

Створення індексів для первинних ключів:

CREATE INDEX au_index ON writers (au_id);
CREATE INDEX title_index ON titles (title_id);
CREATE INDEX pub_index ON publishers (pub_id);
CREATE INDEX site_index ON wwwsites (site_id);

Початкове визначення структури індексів проводиться розробником на стадії проектування інформаційної системи. Надалі вона уточнюється адміністратором системи за наслідками аналізу її роботи, на основі статистики запитів.

5.3.4. Обмеження цілісності колонки

Елемент необов'язкового списку обмежень цілісності колонки визначається наступними синтаксичними правилами:

column_constraint_definition ::=
[CONSTRAINT constraint_name]
NOT NULL
{ PRIMARY KEY UNIQUE }
references_definition
CHECK (conditional_expression)

Обмеження NOT NULL означає, що у визначуваній колонці ніколи не повинні міститися невизначені значення. Якщо визначувана колонка має ім'я A, то це обмеження еквівалентно наступному табличному обмеженню: *CHECK (A IS NOT NULL).*У визначення колонки може входити одне з обмежень: обмеження первинного ключа *(PRIMARY KEY)* або обмеження можливого ключа *(UNIQUE).* Обмеження *PRIMARY KEY*, на додаток до цього, спричиняє за собою обмеження NOT NULL для колонки, що визначається. Ці обмеження колонки еквівалентні наступним табличним обмеженням: *PRIMARY KEY (A) і UNIQUE (A).* Обмеження *references_definition* означає оголошення визначуваної колонки зовнішнім ключем таблиці і має наступний синтаксис:

references_definition ::= REFERENCES base_table_name [(column_commalist)]
[MATCH { SIMPLE FULL PARTIAL }]
[ON DELETE referential_action]
[ON UPDATE referential_action]

Насправді, дана синтаксична конструкція працює і у разі визначення зовнішнього ключа на рівні таблиці (в одному з визначень табличних обмежень цілісності). Тому ми відкладемо обговорення цього питання до розгляду цього загального випадку. Відзначимо тільки, що при використанні конструкції на рівні визначення колонки column_commalist може містити ім'я тільки однієї колонки (тому що зовнішній ключ складається з однієї визначуваної колонки). Обмеження еквівалентно наступному табличному обмеженню: *FOREIGN KEY (A) references_definition*.

Перевірочне обмеження *CHECK (conditional_expression)* призводить до того, що в даній колонці можуть знаходитися тільки ті значення, для яких обчислення conditional_expression не приводить до результату *false*. В умовному виразі перевірочного обмеження колонки дозволяється використовувати ім'я тільки визначеної колонки. Помітимо, що перевірочне обмеження колонки може бути без жодних змін перенесено на рівень визначення табличних обмежень.

Є три різновиди табличних обмежень: обмеження первинного або можливого ключа *(PRIMARY KEY* або *UNIQUE*), обмеження зовнішнього ключа (*FOREGN KEY*) і перевірочне обмеження (*CHECK*).

Елемент визначення табличного обмеження цілісності визначається наступним чином:

base_table_constraint_definition ::=
[CONSTRAINT constraint_name]
{ PRIMARY KEY UNIQUE } (column_commalist)

FOREIGN KEY (column_commalist)
references_definition
CHECK (conditional_expression)

Будь-якому обмеженню може явним чином призначатися ім'я, якщо перед визначенням обмеження розмістити конструкцію *CONSTRAINT constraint_name.*

5.3.5. Табличне обмеження первинного або можливого ключа

Табличне обмеження первинного або можливого ключа *{ PRIMARY_KEY UNIQUE } (column_commalist)* означає вимогу унікальності складових значень вказаної групи колонок (тобто у весь час існування визначуваної таблиці у всіх її рядках складові значення даної групи колонок повинні бути різні. Обмеження *PRIMARY KEY*, на додачу до цього, обумовлює обмеження *NOT NULL* для всіх колонок, згадуваних у визначенні обмеження. У визначенні таблиці дозволяться довільне число визначень можливого ключа (для різних комбінацій колонок), але не більше одного визначення первинного ключа. З цього витікає, що в мові *SQL* дійсно допускається визначення таблиць, у яких відсутні можливі ключі. Ця особливість мови, є характерним відхиленням реальної мови *SQL* від строгої реляційної моделі даних.

5.3.6. Перевірочне табличне обмеження

Визначення табличного обмеження виду *CHECK (conditional_expression)* призводить до того, що вказаний умовний вираз обчислюватиметься при кожній спробі оновлення відповідної таблиці (вставці нового рядка, видаленні або модифікації існуючого рядка). Вважається, що спроба оновлення таблиці порушує перевірочне обмеження цілісності, якщо після виконання операції оновлення обчислення умовного виразу дає результат *false.* Іншими словами, таблиця знаходиться відповідно до даного перевірочного табличного обмеження, якщо для всіх рядків таблиці результатом обчислення відповідного умовного виразу не є false. Розглянемо різновиди умовних виразів пізніше в контексті розгляду оператора SELECT мови SQL.

5.3.7. Табличне обмеження зовнішнього ключа

Синтаксис і семантика визначення зовнішнього ключа в операторі SQL визначення базової таблиці є досить заплутаним і складним. З цієї причини ми присвячуємо цій мовній конструкції окремий підрозділ. Табличне обмеження FOREIGN KEY (*column_commalist*)

references_definition означає оголошення зовнішнім ключем групи колонок, імена яких перераховані в списку *column_commalist*. Обговоримо тепер значення обмеження зовнішнього ключа при різних варіантах формування визначення посилань *(references_definition*). Для зручності повторимо синтаксичне правило.

```
references_definition ::=
   REFERENCES base_table_name [ (column_commalist) ]
      [ MATCH { SIMPLE  FULL  PARTIAL } ]
      [ ON DELETE referential_action ]
      [ ON UPDATE referential_action  ]
```

В цьому визначенні *base_table_*name повинне бути ім'ям деякої базової таблиці (хай, наприклад, ця таблиця має ім'я T). Якщо визначення посилань включає список колонок (column_commalist), то цей список повинен співпадати (з точністю до порядку проходження імен колонок) із списком імен колонок, використаних в деякому визначенні первинного або можливого ключа (*PRIMARY_KEY* або *UNIQUE*) у визначенні таблиці T. Якщо у визначенні посилань список колонок явно не заданий, то вважається, що він співпадає із списком колонок, використаних у визначенні первинного ключа (*PRIMARY_KEY*) таблиці T.

Розглянемо приклади визначення обмежень. Колонки *STUD_NO, STUD_STYPEND, GROUPE_NO, COURS_NO, GROUPE_TOTAL_STYPEND, GROUPE_LECT* і *COURS_LECT* визначаються на раніше визначених доменах (суть цих доменів тут не істотна). Первинними ключами відношень STUD, GROUPE і курсів COURS є колонки STUD_NO, GROUPE_NO і COURS_NO відповідно. В таблиці STUD колонки GROUPE_NO і COURS_NO є зовнішніми ключами, вказуючими на групу, в якій навчається студент, і на курс, на якому він навчається відповідно. В таблиці GROUPE зовнішнім ключем є колонка GROUPE_LECT, що вказує на студента, який представляє відповідну групу, а в таблиці COURS зовнішнім ключем є колонка COURS_LECT, що вказує на викладача, що є лектором відповідної дисципліни. Інші обмеження цілісності ми обговоримо пізніше.

Визначимо таблицю STUD:

```
1.        CREATE TABLE STUD (
2.        STUD_NO STUD_NO PRIMARY KEY
3.        STUD_NAME VARCHAR(20) DEFAULT 'None' NOT NULL
4.        STUD_BDATE DATE DEFAULT NULL CHECK (
      VALUE >= DATE '1970-07-01')
5.        STUD_STYPEND STYPENDIA
6.        GROUPE_NO GROUPE_NO DEFAULT NULL REFERENCES
      GROUPE ON DELETE SET NULL
7.        COURS_NO COURS_NO DEFAULT NULL
8.        FOREIGN KEY COURS_NO REFERENCES COURS (COURS_NO)
      ON DELETE SET NULL
9.        CONSTRAINT COURS_STUD_NO CHECK
     ((SELECT COUNT (*) FROM STUD E
       WHERE E.COURS_NO = COURS_NO) <= 50));
10.   Birth_place VARCHAR(25) DEFAULT 'None' NOT NULL
```

Обговоримо частини цього визначення. В рядку номер 1 вказується, що створюється таблиця з ім'ям *STUD.* В рядку номер 2 визначається колонка *STUD_NO* на домені *STUD_NO.* У цієї колонки не визначено значення за умовчанням, і вона оголошена первинним ключем таблиці (це обмеження цілісності додається через AND до обмежень, успадковуваною колонкою від визначення домена). Крім того, це означає неявну вказівку заборони для даної колонки невизначених значень. В рядку номер 3 визначена колонка STUD_NAME на базовому типі даних символьних рядків змінної довжини з максимальною довжиною 20. Для колонки вказано значення за умовчанням – рядок 'None', і як обмеження цілісності заборонені невизначені значення. В рядку номер 4 визначається колонка *STUD_BDATE* (дата народження). Він має тип даних DATE, значенням за умовчанням є NULL (дати народження деяких службовців невідомі). Крім того, обмеження колонки забороняє зараховувати осіб, про яких відомо, якщо вони народилися до 1970 р. В рядку номер 5 визначена колонка *STUD_STYPEND* на домені *STYPEND*. Значення за умовчанням і обмеження цілісності успадковуються з визначення домена. В рядку номер 6 колонка *GROUPE_NO* визначається на однойменному домені (для нашої мети його значення неістотне), але явно оголошується, що значенням за умовчанням цієї колонки буде *NULL* (деякі студенти(екстернат) не приписані до жодної групи. Крім того, додається обмеження зовнішнього ключа: колонка *GROUPE*_NO посилається на первинний ключ таблиці *GROUPE*. Визначена посилальна дія: при видаленні рядка з таблиці *GROUPE* у всіх рядках таблиці *STUD*, що посилалися на цей рядок, колонці *GROUPE_NO* повинне бути присвоєно невизначене значення. В рядку номер №@: визначається колонка *COURS*_NO. Це визначення аналогічно визначенню колонки *GROUPE_NO*, але обмеження зовнішнього ключа винесено в частину №@;, де воно визначається в повній формі як табличне обмеження. Нарешті, в рядку номер №@< визначається табличне перевірочне обмеження з ім'ям *COURS_STUD_NO,* яке вимагає, щоб на жодному курсі навчалося не більше 50 студентів.

Визначимо таблицю GROUPE:

```
1.CREATE TABLE GROUPE (
2.GROUPE_NO GROUPE_NO PRIMARY KEY
3.GROUPE_STUD_NO INTEGER NOT NULL CHECK (
VALUE BETWEEN 1 AND 30)
4.GROUPE_NAME VARCHAR(200) DEFAULT 'Nameless' NOT NULL
5.GROUPE_TOTAL_STYPEND STYPENDARY DEFAULT 10000.00
 NOT NULL CHECK (VALUE > = 10000.00)
6.GROUPE_MNG STUD_NO DEFAULT NULL
  REFERENCES STUD ON DELETE SET NULL
 CHECK (IF (VALUE IS NOT NULL) THEN
((SELECT COUNT(*) FROM GROUPE
  WHERE GROUPE.GROUPE_MNG = VALUE) = 1)
7.CHECK (GROUPE_STUD_NO =
```

```
(SELECT COUNT(*) FROM STUD
  WHERE GROUPE_NO = STUD.GROUPE_NO))
    8.CHECK (GROUPE_TOTAL_STYPEND >=
(SELECT SUM(STUD_STYPEND) FROM STUD
  WHERE GROUPE_NO = STUD.GROUPE_NO)));
```

В рядку під номером 3 колонка *GROUPE_STUD_NO* (кількість студентів в групі) визначений на базовому типі *INTEGER* без значення за умовчанням, із забороною невизначеного значення і з перевірочним обмеженням, що встановлює допустимий діапазон значень числа студентів у групі. Ще одне перевірочне обмеження цієї колонки – 7– винесено на рівень визначення табличного обмеження. Це обмеження встановлює, що в кожному рядку таблиці *GROUPE* значення колонки *GROUPE_STUD_NO* повинне дорівнювати загальному числу рядків таблиці *STUD*, в яких значення колонки GROUPE_NO дорівнює значенню однойменної колонки даного рядка таблиці *GROUPE*. В рядку 5 для визначення колонки *GROUPE_TOTAL_STYPEND* (об'єм стипендіального фонду групи) використовується домен *STYPENDARY*. Але при цьому явно встановлено значення колонки за умовчанням (відмінне від значення за умовчанням домена), заборонено наявність невизначених значень і введено додаткове перевірочне обмеження, що визначає нижній поріг об'єму стипендіального фонду групи. Ще одне перевірочне обмеження – 8 – винесено на рівень визначення табличного обмеження. Це обмеження встановлює, що в кожному рядку таблиці *GROUPE* значення колонки *GROUPEE_TOTAL_STYPEND* повинне бути не менше суми значень колонки *STUD_STYPEND* у всіх рядках таблиці *STUD,* в яких значення колонки *GROUPE_NO* дорівнює значенню однойменної колонки даного рядка таблиці GROUPE. Звернемо увагу на визначення колонки GROUPE_MNG – частина 6. Ця колонка оголошується зовнішнім ключем таблиці GROUPE. Але ми хочемо сказати більше. У групи може бути тимчасово відсутній староста, тому в колонці допускаються невизначені значення. Але якщо у групи є староста, то він повинен бути старостою тільки цієї групи. На перший погляд можна було б скористатися обмеженням колонки UNIQUE. Але таке обмеження припускало б наявність невизначеної колонки GROUPE_MNG тільки в одному рядку таблиці GROUP, а ми хочемо припускати відсутність старости у декількох груп. Тому було потрібно ввести більш громіздке перевірочне обмеження колонки.

Визначимо таблицю COURS.

```
1.      CREATE TABLE COURS (
2.      COURS_NO COURS_NO PRIMARY KEY
3.      COURS_TITLE VARCHAR(200)DEFAULT 'No title' NOT NULL
4.      COURS_SDATE DATE DEFAULT CURRENT_DATE NOT NULL
5.      COURS_DURAT INTERVAL YEAR DEFAUL INTERVAL '1'
    YEAR NOT NULL
6.      COURS_LECT STUD_NO UNIQUE NOT NULL
```

REFERENCES STUD ON DELETE NO ACTION
7. COURS_DESC CLOB(10M));

Колонка *COURS_SDATE* містить дату початку курсу, а колонка *COURS_DURAT* – тривалість курсу в семестрах. В цьому визначенні має сенс прокоментувати частину 6. Будемо вважати, що якщо група, принаймні тимчасово, може існувати без старости, то у курсу завжди має бути лектор. Тому визначення колонки *COURS_LECT* є набагато більш строгим, ніж визначення колонки *GROUPE_LECT* в таблиці *GROUPE*. Поєднання обмежень *UNIQUE* і *NOT NULL* за відсутності значень за умовчанням приводить до абсолютної унікальності значень колонки *COURS_LECT*. Іншими словами, ця колонка має всі характеристики первинного ключа, хоча оголошена тільки як можливий ключ. Крім того, вона оголошена як зовнішній ключ з дією при видаленні рядка таблиці *STUD* з відповідним значенням первинного ключа *NO ACTION*, що забороняє такі видалення. В сукупності це гарантує, що у будь-якого курсу існуватиме лектор, що входить в штат університету. В рядку номер 5 колонка *COURS_DESC* (опис курсу) визначена як великий символьний об'єкт з максимальним розміром 10 Мбайт

Обмеження цілісності, що входять у визначення таблиці (включаючи явні і успадковувані від визначення доменів обмеження колонок), є обмеженнями бази даних, частиною якої є дана таблиця. Крім того, можуть визначатися додаткові обмеження бази даних. В стандарті SQL такі додаткові обмеження бази даних називаються *ASSERTION*, а ми їх називатимемо загальними обмеженнями цілісності.

Визначимо таблицю Mark

CREATE TABLE Mark(
1. Mark mark NOT NULL
2. Stud_id STUD_NO NOT NULL
REFERENCES stud ON DELETE SET NULL YEAR NOT NULL
3. Exam_id Exam_id NOT NULL
REFERENCES exam ON DELETE SET NULL

Визначимо таблицю Exam:

CREATE TABLE Exam(
Exam_id
Exam_date
Exam_forma_id
Examenator_name
Examenator_position

5.3.8. Визначення загальних обмежень цілісності

Для визначення загального обмеження цілісності служить оператор *CREATE ASSERTION,* що має наступний синтаксис:

CREATE ASSERTION constraint_name
CHECK (conditional_expression)

Зауважимо, що при створенні загального обмеження цілісності його ім'я завжди повинне вказуватися явно. Хоча синтаксис визначення загального обмеження співпадає з синтаксисом визначень обмежень колонки і таблиці, в даному випадку допускаються тільки спеціальні види умовних виразів. Ми не можемо зараз точно сформулювати властивості цих видів умов, оскільки спочатку необхідно розглянути різновиди умовних виразів, які будуть зроблені в наступному розділі. Якщо говорити неформально, то особливі властивості умов зв'язані з тим, що при визначенні загальних обмежень цілісності контекстом, в якому обчислюється умовний вираз, є весь набір таблиць бази даних, а не набір рядків таблиці, як це було при визначенні табличних обмежень. Продемонструємо і прокоментуємо декілька прикладів визначень загальних обмежень цілісності.

У визначенні таблиці *STUD* містилося обмеження колонки *STUD_BDATE*:

CHECK (STUD_BDATE >= '1970-07-01')

(студентами можуть бути особи вік, яких на момент вступу не перевищував 35 років). Ось яким чином можна визначити таке ж обмеження на рівні загальних обмежень цілісності:

CREATE ASSERTION MIN_STUD_BDATE CHECK ((SELECT MIN(EMP_BDATE)) FROM EMP) >= '1973-07-01')

В логічній умові цього загального обмеження вибирається мінімальне значення колонки *STUD_BDATE* (дата народження найстарішого службовця). Значенням умовного виразу буде false в тому і лише в тому випадку, якщо серед студентів є хоча б один, що народився до вказаної дати. Тепер переформуємо обмеження цілісності обмеження таблиці STUD у загальному вигляді *COURS_STUD_NO,* яке визначється таким чином:

CONSTRAINT COURS_STUD_NO CHECK
((SELECT COUNT (*) FROM STUD E
WHERE E. COURS_NO = COURS_NO) <= 50)

(один курс слухають не більше 50 студентів).

Ось формулювання еквівалентного загального обмеження цілісності:

CREATE ASSERTION NEW_ COURS_STUD_NO CHECK
(NOT EXISTS (SELECT COURS_NO FROM STUD GROUPE COURS_NO
HAVING COUNT > 50)).*

Логічний вираз цього обмеження може приймати тільки значення *true і false*. Внутрішній оператор вибірки впорядковує рядки таблиці *STUD* таким чином, що в одну групу потрапляють всі рядки з однаковим значенням колонки *COURS_NO*. Потім ці групи фільтруються за умовою розділу *HAVING*, і залишаються тільки групи, що включають більше 50 рядків. В результуючій таблиці містяться рядки з однієї колонки, яка містить значення *COURS_NO* груп, що залишилися. Предикат *NOT EXISTS* приймає значення *true*

тоді і тільки тоді, коли ця результуюча таблиця не містить жодного рядка, тобто немає жодного курсу, в якому вчаться більше 50 студентів.

5.4.Мова маніпулювання даними

Основою мови маніпулювання даними SQL описані в розділі операції реляційної алгебри. В інтерпретації SQL ці операції виглядають наступним чином:

Операція об'єднання
Засобами мови SQL операція об'єднання представляється таким чином:
*SELECT **
FROM A
UNION
*SELECT **
FROM B
Операція різниця
Засобами мови SQL операція різниця представляється таким чином:
*SELECT **
FROM A
EXCEPT
*SELECT **
FROM B
Операція проекція
SELECT Fieldi1 ., Fieldin
FROM A
Операція вибірка (селекція)
*SELECT **
FROM A
WHERE (<condition>)

Операція перетин

*SELECT **

FROM A

INTERSECT

*SELECT **

FROM B

Операція з'єднання, еквіз'єднання

SELECT A.Field1, ., A.Fieldn, B.Field1, ., B.Fieldm

FROM A, B

WHERE (A.Fieldi Θ B.Field1)

Якщо Θ – операція «=«, то це еквіз'єднання.

Операція природного з'єднання

Нехай є відносини $A\ (X_1 .,\ X_n,\ A_1 .,\ A_m)$ і $B\ (X_1 .,\ X_n\ B_1 .,\ B_r)$.

SELECT $A.X_1, .,\ A.X_n,\ A.A_1, .,\ A.A_m,\ B.B_1, .,\ B.B_r$

FROM A, B

WHERE $(A.X_1 = B.X_1)$ *AND . AND* $(A.X_n = B.X_n)$

Як ми бачимо, основною конструкцією є оператор *SELECT* – тобто оператора вибірки, очевидно, найважливішого інструменту для роботи з даними бази даних. Синтаксис цього оператора досить складний. Розглянемо його далі докладніше.

5.4.1.Загальна структура оператора вибірки мови SQL

Для вибірки даних у прямому SQL використовується оператор *SELECT*, що повертає набір з одного, або декількох рядків одинакової структури і що визначається в наступним синтаксисом:

SELECT [ALL DISTINCT] select_item_commalist
FROM table_reference_commalist
[WHERE conditional_expression]
[GROUP BY column_name_commalist]
[HAVING conditional_expression]
[ORDER BY order_item_commalist]

Спершу опишемо загальну схему виконання *оператора SELECT* відповідно до вимог стандарту. Виконання запиту складається з декількох кроків, що відповідають розділам оператора вибірки. На першому кроці виконується розділ *FROM*. Якщо список посилань на таблиці (*table_reference_commalist*) цього розділу відповідає таблицям A, B, C, то в результаті виконання розділу *FROM* утворюється таблиця (назвемо її T), що є розширеним декартовим добутком таблиць A, B , C. Якщо в розділі *FROM* вказана тільки одна таблиця, то вона ж і є результатом виконання цього розділу.

На другому кроці виконується *розділ WHERE.* Умовний вираз (conditional_expression) цього розділу застосовується до кожного рядка таблиці T, і результатом є таблиця T_1, що містить ті і лише ті рядки таблиці T, для яких результатом обчислення умовного виразу є *true.* (Заголовки таблиць T і T_1 співпадають.) Якщо *розділ WHERE* в операторі вибірки відсутній, то це трактується як наявність розділу *WHERE true*, тобто T_1 містить ті і лише ті рядки, які містяться в таблиці T.

Якщо в операторі вибірки присутній розділ *GROUP BY*, то він виконується на третьому кроці. Кожен елемент списку імен колонок (column_name_commalist), що указується в цьому розділі, повинен бути одним з імен колонок таблиці T_1. В результаті виконання розділу GROUP BY утворюється згрупована таблиця T_2, в якій рядки таблиці T_1 розставлені в мінімальне число груп, таких, що у всіх рядках однієї групи значення колонок, вказаних в списку імен колонок розділу GROUP BY (колонок угрупування), однакові.

Якщо в операторі вибірки присутній розділ *HAVING*, то він виконується на наступному кроці. Умовний вираз цього розділу застосовується до кожної групи рядків таблиці T_2, і результатом є згрупована таблиця T_3, що містить тільки ті групи рядків таблиці T_2, для яких результатом обчислення умовного виразу є *true*. Умовний вираз розділу *HAVING* будується за синтаксичними правилами, загальними для всіх умовних виразів, але має таку особливість, що застосовується до груп рядків, а не до окремих рядків. Тому предикати, з яких будується цей умовний вираз, повинні бути предикатами на групу в цілому. У них можуть використовуватися імена колонок угрупування (інваріанти групи) і так звані агрегатні функції (COUNT, SUM, MIN, MAX, AVG) від інших колонок. Ці агрегатні функції очевидно носять нереляційну природу, але без них ефективність SQL була б значно нижчою.

За наявності в запиті розділу *HAVING*, перед яким немає розділу *GROUP BY*, таблиця T_1 розглядається як згрупована таблиця, що складається з однієї групи рядків, без колонок групування. В цьому випадку логічний вираз розділ*у HAVING* може складатися тільки з предикатів з агрегатними функціями, а результат обчислення цього розділу T_3 або співпадає з таблицею T_1, або є порожнім.

Якщо в операторі вибірки присутній розділ *GROUP BY*, але відсутній розділ *HAVING*, то це трактується як наявність розділу HAVING true, тобто T_3 містить ті і лише ті групи рядків, які містяться в таблиці T_2.

Після виконання розділу *WHERE* (якщо в запиті відсутні *розділи GROUP BY* і *HAVING*, (випадок a)) або явно або неявно заданого розділу *HAVING* (випадок b)) виконується розділ *SELECT*. При виконанні цього розділу на основі таблиці T_1 у випадку (a) або на основі згрупованої таблиці T_3 у випадку (b) будується таблиця T_4, що містить стільки рядків, скільки рядків або груп рядків міститься в таблицях T_1 або T_3 відповідно. Число колонок в таблиці T_4 залежить від числа елементів в списку елементів вибірки *(select_item_commalist)* і від виду елементів.

Розглянемо, яким чином формуються значення колонок в таблиці T_4. Елемент списку вибірки може задаватися одним з двох способів:

*select_item ::= value_expression [[AS] column_name] [correlation_name .] **

Спочатку обговоримо перший варіант. В цьому випадку кожен елемент списку елементів вибірки відповідає колонці таблиці T_4. Колонці може бути явним чином приписано ім'я (коли і навіщо можуть використовуватися імена таблиці T4, ми обговоримо пізніше). Порядок формування значення цього колонки для виділених вище випадків (a) і (b) розрізняється, і ми розглянемо подібні випадки окремо.

У випадку (a) вираз, що міститься в елементі вибірки, може містити літеральні константи і виклики функцій із значеннями відповідних типів . Крім того, у виразі можуть

використовуватися імена колонок таблиці T1. Вираз обчислюється для кожного рядка таблиці T1, і іменам колонок відповідають значення цих колонок в даному рядку таблиці T1.

У випадку (b), як і у випадку (a), вираз, що міститься в елементі вибірки, може містити літеральні константи і виклики функцій. Але, на відміну від випадку (a), у вираз можуть входити безпосередньо імена тільки тих колонок таблиці T3, які входили в список колонок угрупування розділу GROUP BY оператора вибірки. (Якщо згрупована таблиця T3 була утворена за рахунок наявності розділу *HAVING* без присутності *розділу GROUP BY*, то у виразі елементу вибірки взагалі не можна безпосередньо використовувати імена колонок таблиці T3). Імена інших колонок таблиці T3 можуть використовуватися тільки в конструкціях виклику агрегатних функцій *COUNT, SUM, MIN, MAX, AVG*. Вираз обчислюється для кожної групи рядків таблиці T3. Іменам колонок, що входять у вираз безпосередньо, зіставляються значення цих колонок, які відповідають даній групі з рядків таблиці T3.

Отже, ми отримали таблицю T4. Якщо в специфікації *розділу SELECT* відсутнє ключове слово DISTINCT, або присутнє ключове слово ALL, або відсутні і ALL, і DISTINCT, то T4 є результатом виконання *розділу SELECT*. Інакше на завершуючій стадії виконання *розділу SELECT* в таблиці T4 віддаляються рядки-дублікати.

Якщо в операторі вибірки не міститься розділ *ORDER BY*, то таблиця T4 є результуючою таблицею запиту. Інакше на завершуючій стадії виконання запиту проводиться сортування рядків таблиці T4 відповідно до списку елементів сортування (*order_item_commalist*) розділу *ORDER BY.* У стандарті SQL:1999 елемент списку елементів сортування має наступну синтаксичну форму:

order_item ::= value_expression [collate_clause]

[{ ASC DESC }]

Виконання *розділу ORDER BY* проводиться таким чином. Вибирається перший елемент списку сортування, і рядки таблиці T4 розставляються в порядку зростання (якщо в елементі присутня специфікація *ASC*; за відсутності специфікації *ASC/DESC* передбачається наявність *ASC*) або в порядку убування (за наявності специфікації DESC) відповідно до значень виразу, що міститься в даному елементі, які обчислюються для кожного рядка таблиці T4. Далі вибирається другий елемент списку сортування, і відповідно до значень заданого в нім виразу і порядку сортування розставляються рядки, які після першого кроку сортування утворили групи з однаковим значенням виразу першого елементу списку сортування. Операція триває до вичерпання списку елементів сортування. Результуючий відсортований список рядків є остаточним результатом запиту.

5.4.2.Посилання на таблиці розділу FROM

Нагадаємо, що *розділ FROM* оператора вибірки визначається синтаксичним правилом

FROM table_reference_commalist

Розглянемо детальніше, який вигляд можуть мати елементи цього списку. Спершу приведемо повний набір синтаксичних правил SQL:1999, визначальний *table_reference*.

table_reference ::= table_primary joined_table

table_primary ::= table_or_query_name [[AS] correlation_name

[(derived_column_list)]]

derived_table [[AS] correlation_name

[(derived_column_list)]]

lateral_derived_table [[AS] correlation_name

[(derived_column_list)]]

collection_derived_table [[AS] correlation_name

[(derived_column_list)]]

ONLY (table_or_query_name)[[AS] correlation_name

[(derived_column_list)]]

(joined_table)

table_or_query_name ::= { table_name query_name }

derived_table ::= (query_expression)

lateral_derived_table ::= LATERAL (query_expression)

collection_derived_table ::= UNNEST

(collection_value_exression) [WITH ORDINALITY]

Крім цього існують додаткові можливості SQL при маніпулюванні таблицями, наприклад, породжуваних таблицями із горизонтальними зв'язками *(lateral_derived_table)* і «*сполучених таблиць*» *(joined_table*). Крім того, ми не розглядатимемо тут конструкції *collection_derived_table* і *ONLY (table_or_query_name),* оскільки вони відносяться до об'єктних розширень мови SQL, які в даному посібнику не розглядаються. Але навіть при таких самообмеженнях для подальшого просування нам доведеться визначити декілька додаткових синтаксичних конструкцій мови SQL.

5.4.3.Табличний вираз, специфікація запиту і вираз|вираження| запитів

Табличним виразом (table_expression) називається конструкція

table_expression ::= FROM table_reference_commalist

[WHERE conditional_expression]

[GROUP BY column_name_commalist]

[HAVING conditional_expression]

Специфікацією запиту *(query_specification)* називається конструкція

query_specification SELECT [ALL DISTINCT]

select_item_commalist table_expression

Нарешті, виразом запитів (query_expression) називається конструкція

query_expression ::= [with_clause] query_expression_body

query_expression_body ::= { non_join_query_expression

joined_table }

non_join_query_expression ::= non_join_query_term

query_expression_body

{ UNION EXCEPT }[ALL DISTINCT]

[corresponding_spec] query_term

query_term ::= non_join_query_term joined_table

non_join_query_term ::= non_join_query_primary

query_term INTERSECT [ALL DISTINCT]

[corresponding_spec] query_primary

query_primary ::= non_join_query_primary joined_table

non_join_query_primary ::= simple_table

(non_join_query_expression)

simple_table ::= query_specification

table_value_constructor

TABLE table_name

corresponding_spec ::= CORRESPONDING

[BY column_name_comma_list]

Якщо не звертати уваги на те, що не обговорювалися конструкції *joined_table* і *table_value_constructor*, синтаксичні правила показують, що вираз запитів будується з виразів, значеннями яких є таблиці, з використанням «теоретико-множинних» операцій *UNION* (об'єднання), *EXCEPT* (віднімання) і *INTERSECT* (перетин). Операція перетину є «мультиплікативною» і має вищий пріоритет, ніж «адитивні» операції об'єднання і віднімання. Обчислення виразу проводиться зліва направо з урахуванням пріоритетів операцій і круглих дужок. При цьому діють наступні правила.

- Якщо вираз запитів не включає жодної теоретико-множинної операції, то результатом обчислення виразу запитів є результат обчислення простої або *сполученої таблиці*.

- Якщо в термі (*non_join_query_term*) або виразі запитів *(non_join_query_expression)* без з'єднання присутня теоретико-множинна операція, то хай T_1, T_2 і T_R позначають відповідно перший операнд, другий операнд і результат терма або виразу відповідно, а OP - використовувану теоретико-множинну операцію.

Якщо в операції присутня специфікація *CORRESPONDING*, то:

якщо присутня конструкція *BY column_name_comma_list*, то всі імена в цьому списку повинні бути різні, і кожне ім'я повинне бути одночасно ім'ям певної колонки таблиці T_1 і ім'ям певної колонки таблиці T_2, причому типи цих колонок повинні бути сумісними; позначимо даний список імен через *SL*;

якщо список відповідності колонок не заданий, нехай SL позначає список імен , колонок, що є іменами, і в T_1, і в T_2, в тому порядку, в якому ці імена фігурують в T_1;

обчислюваний терм або вираз запитів без з'єднання еквівалентні виразу *(SELECT SL FROM* T_1*) OP (SELECT SL FROM* T_2*)*, що не включає специфікацію *CORRESPONDING*.

За відсутності в операції специфікації *CORRESPONDING* операція виконується таким чином, начебто ця специфікація була присутня і включала конструкцію *BY column_name_comma_list*, в якій були б перераховані всі колонки таблиці T_1.

1. При виконанні операції *OP* два рядки s_1 з іменами колонок $c_1, c_2 ., c_n$ і s_2 з іменами колонок $d_1, d_2 ., d_n$ вважаються рядками-дублікатами, якщо для кожного $I (i = 1, 2 ., n)$ або c_i і d_i не містять *NULL*, і $(c_i = d_i)$ = *true,* або і c_i, і d_i містять *NULL*.
2. Якщо в операції OP не задана специфікація *ALL*, то в *TR* рядки-дублікати видаляються.
3. Якщо специфікація ALL задана, то хай s - рядок, що є дублікатом якогось рядка T_1, або якогось рядка T_2, або обох; хай m - число дублікатів s в T_1, а n – кількість дублікатів s в T_2. Тоді:

А) якщо вказана операція *UNION*, то число дублікатів s в TR дорівнює m + n;

Б)якщо вказана операція *EXCEPT,* то число дублікатів s в *TR* дорівнює max *((m-n),0);*

4. якщо вказана операція *INTERSECT*, то число дублікатів *s* в *TR* дорівнює *m i n (m,n).*

5.4.4.Розділ WITH виразу запитів

Як випливає з синтаксису, в цьому виразі може бути присутній розділ WITH. Він визначається в наступним чином:

with_clause ::= WITH [RECURSIVE] with_element_comma_list

with_element ::= query_name [(column_name_list)]

AS (query_expression) [search_or_cycle_clause]

Поки що обмежимося випадком, коли в розділі WITH відсутні специфікація *RECURSIVE і search_or_cycle_clause.* Тоді конструкція

WITH query_name (c_1, c_2 , ... ,c_n) AS (query_exp_1) query_exp_2

означає, що в будь-якому місці виразу запитів query_exp_2, де допускається поява посилання на таблицю, можна використовувати ім'я *query_name*. Можна вважати, що перед виконанням *query_exp_2* відбувається виконання *query_exp_1*, і результуюча таблиця з іменами колонок c_1, c_2 . c_n зберігається під ім'ям *query_name*. Як ми побачимо пізніше, в цьому випадку розділ *WITH* фактично служить для локального визначення таблиці, що представляється*(VIEW*).

5.4.5.Конструктори значення рядка і таблиці

Щоб завершити обговорення виразів запитів , нам залишилося розглянути конструкції *table_value_constructor* і *TABLE table_name.*

У визначенні конструктора значення-таблиці використовується конструктор значення-рядка, який будує впорядкований набір скалярних значень, що представляє рядок:

row_value_constructor ::= row_value_constructor_element

[ROW] (row_value_constructor_element_comma_list)

row_subquery

row_value_constructor_element ::= value_expression NULL DEFAULT

Відзначимо, що значення елементу за умовчанням можна використовувати лише у тому випадку, коли конструктор значення-рядка застосовується в операторі *INSERT* (тоді цим значенням буде значення за умовчанням відповідного колонки).

Конструктор значення-таблиці проводить таблицю на основі заданого набору конструкторів значень-рядків:

table_value_constructor ::= VALUES

row_value_constructor_comma_list

Звичайно, для того, щоб коректно побудувати таблицю, потрібно, щоб рядки, згенеровані всіма конструкторами рядків, були одного і того ж ступеню і щоб типи (або домени) відповідних колонок були такими, що приводяться.

Нарешті, конструкція *TABLE table_name* є скороченою формою запису виразу *SELECT * FROM table_name.*

5.4.6.Посилання на базові таблиці, що представляються або породжуються

Тепер ми можемо завершити обговорення різновидів посилань на таблицю в *розділі FROM*. Для зручності повторимо синтаксичні правила :

table_reference ::= table_primary

table_primary ::= table_or_query_name [[AS] correlation_name
[(derived_column_list)]]
derived_table [AS] correlation_name
[(derived_column_list)]

table_or_query_name ::= { table_name query_name }

derived_table ::= (query_expression)

Отже, у найпростішому випадку як посилання на таблицю використовується ім'я таблиці (базовою або такою, що представляється) або ім'я запиту, приєднаного до даного запиту за допомогою розділу *WITH*. У іншому випадку (derived_table) породжувана таблиця визначається виразом запиту, взятому у круглі дужки. Явна вказівка імен колонок результату запиту із розділу *WITH* або породжуваної таблиці потрібна у тому випадку, коли ці імена не виводяться явно із відповідного виразу запиту. Зверніть увагу, що у таких випадках у відповідному елементі списку розділу *FROM* повинен указуватися псевдонім (correlation_name), бо інакше таблиця була б взагалі позбавлена імені. Можна вважати, що вираз запиту обчислюється і зберігається в тимчасовій таблиці при обробці розділу *FROM.*

Треба звернути увагу на рекурсивну природу синтаксичних визначень, приведених в цьому розділі. Щоб визначити поняття посилання на таблицю в розділі FROM оператора вибірки, який спирається на специфікацію запиту, нам довелося запровадити більш загальне поняття виразу запитів, у визначенні якого використовується специфікація запиту. Так, дійсно, багато синтаксичних конструкцій SQL визначаються рекурсивно. Але ця рекурсія ніколи не приводить до зациклення. Використовуючи перерахування імен колонок безпосередньо після ключового слова SELECT, ми можемо обмежити результат запиту лише декількома необхідними колонкими. Аналогічні дії необхідно робити й для рядків. Так, дуже часто потрібно вибрати з таблиці дані, що задовольняють певним умовам. Для реалізації цієї можливості в мові SQL існує ключове слово WHERE. Ключове слово WHERE пишеться в команді SELECT після закінчення секції FROM і містить

після себе логічний вираз, що перевіряється послідовно для кожного рядка розглянутої таблиці. Розглянемо кілька нескладних прикладів запитів:

SELECT name, second_name, surname FROM STUD WHERE STUD_BDATE >= '1990-09-01'

Цей запит роздрукує імена, прізвища всіх студентів, що народилися після 1 вересня 1990 р.

SELECT name, second_name, surname FROM STUD WHERE STUD_BDATE <= '01.01.1990'

Цей запит роздрукує імена, прізвища всіх студентів, що народилися до 2 січня 1990 року.

5.4.7.Кваліфікований вибір - вираз WHERE.

Відомо, що сучасні мови програмування високого рівня надають набагато потужніші засоби для складання умов, ніж зазначено вище. Не лишився осторонь і SQL. Так, SQL припускає формування складних умов відбору з використанням реляційних і булевих операторів. Під реляційними операторами в цьому випадку розуміються:

> - більше;

< - менше;

>= - більше або дорівнює;

<= - менше або дорівнює;

= - дорівнює;

<> - не дорівнює.

Приклади використання реляційних операторів ми розглянули вище. Під булевими операторами в цьому випадку розуміють наступні:

AND - і;

OR - або;

NOT - не.

За допомогою булевих операторів можна створювати складені умови, комбінуючи різні обмеження. Розглянемо приклади відповідних запитів:

SELECT name, second_name, surname FROM stud WHERE (birth_place = ' Київ') AND (birth_date <= '01.01.1989')

Цей запит виведе як результат дані всіх студентів, які народилися в Києві раніше 2 січня 1989 року.

SELECT DISTINCT stud_id FROM mark WHERE (COURS_DURAT > 1) AND (mark < 4) Цей запит виведе як результат ідентифікатори всіх студентів, які слухають курси тривалістю більше одного семестру й менше чотирьох.

SELECT name, second_name, surname FROM STUD WHERE NOT (birth_place = ' Київ')

Цей запит виведе як результат дані всіх студентів, які народилися не в Києві.

Поряд з розглянутими операторами для складання умов, мало чим відрізняються від таких же операторів, що існують у мовах програмування високого рівня, у мові SQL (через специфіку області застосування) існують ряд спеціальних операторів, які, як правило, не мають аналогів в інших мовах. Ось ці оператори:

IN - входження в деяку множину значень;

BETWEEN - входження в деякий діапазон значень;

LIKE перевірка на збіг зі зразком;

IS NULL - перевірка на невизначене значень.

Оператор IN використовується для перевірки входження в певну множину значень. Наприклад запит:

SELECT DISTINCT stud_id FROM mark WHERE mark IN (2,3)

виведе всіх студентів, що одержали хоча б одну двійку або трійку на іспитах. Того ж результату можна домогтися, використовуючи оператор BETWEEN:

SELECT DISTINCT stud_id FROM mark WHERE mark BETWEEN 2 AND 3

Оператор LIKE застосовується винятково до символьних полів, і дозволяє встановлювати, чи відповідає значення поля зразку. Зразок може містити спеціальні символи: _ (символ підкреслення) - заміщає будь-який одиночний символ; % (знак відсотка) - заміщає послідовність будь-якого числа символів. Так, наприклад, що випливає запит покаже дані всіх студентів, прізвища яких починаються з букви «А».

SELECT name, second_name, surname FROM stud WHERE name LIKE 'A%'

Оператор IS NULL дозволяє виявляти в таблиці невизначені значення (нагадаємо, що подібні значення називаються NULL). Так, наприклад, виглядає запит щоб показати всіх студентів, які не вказали місце свого народження:

SELECT name, second_name, surname FROM stud WHERE birth_place IS NULL

Дуже часто виникає необхідність зробити обчислення мінімальних, максимальних або середніх значень у колонках. Так, наприклад, може знадобитися обчислити середній бал або щось інше. Для здійснення подібних обчислень у мові програмування високого рівня треба було б почати певні дії, зокрема організацію циклу, підрахунок суми, потім розділили б її на кількість доданків у сумі, одержавши значення середнього бала. SQL рятує користувача від необхідності програмування

всієї послідовності подібних дій. Для цього SQL надає Вам спеціальні агрегатні функції:

MIN - мінімальне значення в колонці;

MAX - максимальне значення в колонці;

SUM - сума значень у колонці;

AVG - середнє значення в колонці;

COUNT - кількість значень у колонці, відмінних від NULL.

Запити, записані нижче означають відповідно, мінімум, максимум, суму, середнє серед всіх балів, отриманими студентами на іспитах. Кількість оцінок обраховує останній запит.

SELECT MIN(mark) FROM mark

SELECT MAX(mark) FROM mark

SELECT SUM(mark) FROM mark

SELECT AVG(mark) FROM mark

SELECT COUNT(mark) FROM mark

Зауважимо, що, змінивши в такий спосіб текст останнього запиту, ми одержимо кількість різних оцінок, отриманих на іспитах:

SELECT COUNT(DISTINCT mark) FROM mark

Крім описаних вище простих випадків, можна використати агрегатні функції разом із виразом *WHERE:*

SELECT AVG(mark) FROM mark WHERE stud_id = 100

Даний запит обчислить середній бал студента з кодом 100 за результатами всіх зданих їм іспитів.

SELECT AVG(mark) FROM mark WHERE exam_id = 10

А цей запит обчислить середній бал студентів за результатами здачі іспиту з кодом 10. Цих нечисленних прикладів достатньо, щоб переконатись у тому, що агрегатні функції, у сполученні з обмеженням області даних за допомогою виразу *WHERE,* являють собою потужний апарат, який дозволяє відповісти на множину реально виникаючих питанн шляхом застосування *SQL* - запитів.

На додаток до розглянутих механізмів мова *SQL* надає потужний апарат для обчислення агрегатних функцій не для всієї таблиці результатів запиту, а для різних значень по групах. Для цього в SQL існує спеціальна конструкція *GROUP BY,* призначена для позначення тієї колонки, за значеннями якої буде сформовано угруповання. Так, наприклад, ми можемо обчислити середній бал по всіх іспитах для кожного студента. Для цього досить виконати наступний запит:

SELECT stud_id, AVG(mark) FROM mark GROUP BY stud_id

Все це, як звичайно, може бути сполучене із виразом *WHERE*. При цьому, не

вдаючись у тонкощі виконання запиту в середині СУБД, можна вважати, що спочатку виконується вибірка тих рядків таблиці, які задовольняють умовам із виразу WHERE, а потім відбувається угруповання й агрегування. Приведемо запит, який обчислює середній бал за оцінками, отриманими на іспиті з кодом 100, для кожного студента. Для цього досить виконати наступний запит:

SELECT stud_id, AVG(mark) FROM WHERE mark exam_id = 100 GROUP BY stud_id

Зауважимомо, що угруповання може відбутися більш, ніж по одному полю. Можна обчислити середній бал, увівши додаткове розмежування по днях, у які вносилася інформація в базу (вважаючи, наприклад, що ця дата збігається з датою проведення іспиту):

SELECT stud_id, mod_date, AVG(mark) FROM WHERE mark exam_id = 100 GROUP BY stud_id, mod_date

Для запитів, що містять секцію GROUP BY існує важливе обмеження:
такі запити можуть включати як результат колонки, за якими відбувається угруповання, і колонки, які містять власне результати агрегування.

У мові SQL є засоби, які дозволяють зробити таке:
нехай, треба одержати як результати запиту даних тільки тих студентів, середній бал яких перевищує 4. Як це зробити? Проблема в тому, що стандарт мови не допускає використання агрегатних функцій у виразі WHERE. Виходить, що спочатку необхідно виконати запит і порахувати середнє, а потім обмежити результат за умовою «Середнє > 4». Для здійснення подібних дій стандартом мови передбачена спеціальна секція HAVING. Так, запит може виглядати в такий спосіб:

SELECT stud_id, mod_date, AVG(mark) FROM WHERE mark.exam_id = 100 GROUP BY stud_id, mod_date HAVING AVG(mark) > 4

Для того, щоб форматувати виведення, можна використати різні можливості SQL. Так, наприклад, припустимим є включення тексту в запит. Розглянемо приклад того, як це робиться:

SELECT 'Середній бал=', AVG(mark) FROM mark WHERE exam_id = 10

У результаті даного запиту буде отримане не просто якесь число, а число, що супроводжується пояснювальним текстом. Поряд зі звичайною вибіркою значень з колонок таблиці, є можливість вносити зміни в ці значення шляхом обчислення виразів. Так, наприклад, можна перейти від 5-бальної до 10-бальної системи, щляхом перетворення значення середнього бала в такий спосіб:

*SELECT 'Середній бал=', AVG(mark)*2 FROM mark WHERE exam_id = 10*

Наступний важливий момент пов'язаний зі здійсненням сортування (упорядкування) даних.

Для організації сортувань у стандарті SQL передбачена спеціальна секція *ORDER BY*. У цій секції Вам необхідно вказати, по якій колонці (по яких колонках) упорядковувати результати запиту. При цьому, при використанні декількох колонок упорядкування буде відбуватися послідовно (за першим критерієм, якщо значення збіглися - за другим критерієм і т.д.). Наступний запит роздруковує список студентів, у якому прізвища виводяться за абеткою:

SELECT name, second_name, surname FROM stud ORDER BY name

За замовчуванням вважається, що сортування відбувається за зростанням. Якщо необхідно впорядковувати за зменшенням, треба використати вказівку «DESC».

SELECT name, second_name, surname FROM stud ORDER BY name DESC

Для впорядкування по зростанню присутня вказівка «ASC», але його звичайно не пишуть явно. При впорядкуванні по декількох колонках можна впорядковувати як по зростанню, так і по убуванню, указуючи відповідну конструкцію після найменування кожної колонки.

SELECT name, second_name, surname FROM stud ORDER BY name DESC, second_name ASC

Поряд з упорядкуванням у звичайних запитах, стандарт SQL дозволяє сортувати й запити, що містять агрегування. У цьому випадку порядок проходження секцій наступний:

SELECT stud_id, mod_date, AVG(mark) FROM mark WHERE exam_id = 100 GROUP BY stud_id, mod_date HAVING AVG(mark) > 4 ORDER BY stud_id

Досі ми розглядали ту частину SQL, яка стосувалася вибору інформації з єдиної таблиці. Природно, можливості мови цим не обмежуються. Так, можна запитувати інформацію із декількох таблиць, реалізовуючи описані у відповідному розділі посібника реляційні операції. Розглянемо деякі приклади того, як це робиться. Варто згадати, що повний розгляд цієї теми виходить за рамки даного посібника. Детально це описано в монографіях, зокрема, [] - []. Ми розглянемо деякі нескладні випадки, що часто зустрічаються на практиці, в яких присутня необхідність вибору інформації з декількох таблиць відразу. Як правило, в тих випадках, коли виникає необхідність вибирати інформацію з різних таблиць, вони якимсь чином пов'язані один з одним. Розглянемо приклад, який базується на таблицях, описаними в попередньому розділі. На Рис. 5.2. зображена схема зв’язку трьох таблиць *stud, mark* і *exam*.

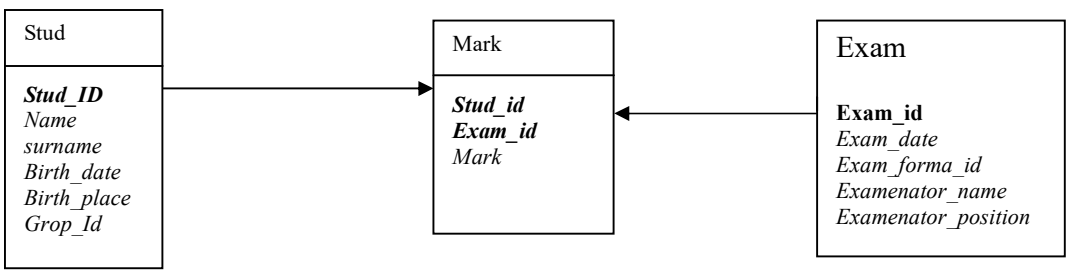


Рис. 5.2. Приклад зв'язаних таблиць

Таблиця mark (оцінки) пов'язана з таблицею exam (іспит) полями id–stud_id. Таблиця mark (оцінка) пов'язана з таблицею stud (студент) полями id– exam_id. Нехай нам потрібно роздрукувати список студентів із оцінками, які вони отримали на іспитах. Для цього необхідно виконати наступний запит:

SELECT stud.name, mark.exam_id, mark.mark FROM stud, mark WHERE stud.id = mark.stud_id

Відзначимо, наступні зміни в порівнянні з тими запитами, які ми писали раніше:

1. У секції FROM вказано 2 таблиці.
2. Позаяк таблиць стало більше однієї, з'явилася певна неоднозначність при використанні полів. Так, у багатьох випадках невідомо, із якої таблиці зі списку FROM брати поле. Для усунення неоднозначності імена полів указуються з префіксом – ім'ям таблиці. Ім'я таблиці від імені поля відокремлюєтья крапкою.
3. У реченні WHERE вказана умова сполучення таблиць. Нехай нас не задовольняє той факт, що ми бачимо код предмету, але не бачимо дату складання іспиту. Для того, щоб вирішити цю проблему необхідно виконати наступний запит:

SELECT stud.name, exam.exam_date, mark.exam_id, mark.mark FROM stud, mark, exam WHERE (stud.id = mark.stud_id) AND (exam.id = mark.exam_id)

Неважко помітити, що використання префіксів-імен таблиць значно утруднює написання запиту. Щоб цього уникнути використовуються псевдоніми. Так, ми можемо переписати попередній запит таким чином:

SELECT E.name, X.exam_date, M.exam_id, M.mark FROM stud E, mark M, exam X WHERE (E.id = M.stud_id) AND (E.id = M.exam_id)

5.5.Оператори модифікації даних

Розглянутий оператор SELECT є чудовим інструментом вибірки даних з бази. Цим оператором можна робити все – крім змін даних в базі. Але великий досвід використання вказує на те, що база даних є суттєво динамічним об’єктом: дані в базі додаються, знищуються та змінюють значення. В практиці використання SQL-подібних СУБД добре відома різниця між запитами на вибірку і запитами на зміну. Для забезпечення цих

базових операцій в SQL реалізовані через відповідні оператори, імена яких англійською мовою і означають імена відповідних операцій INSERT, DELETE, UPDATE.

5.5.1.Оператор INSERT

Природно почати розгляд з оператора *INSERT,* хоча б тому, що для того щось модифікувати в таблицях або видаляти із таблиць, потрібно, щоб таблиці не були пусті.

В загальному вигляді синтаксис оператора INSERT виглядає таким чином:

INSERT INTO table_name

{ [(column_commalist)] query_expression

DEFAULT VALUES

Характерний приклад використання INSERT виглядає так (вставити рядок з колонками column1, column2, column3, надавши відповідні значення (‘data1’, ‘data2’, ‘data3’):

INSERT INTO table_name (column1, column2, column3) VALUES (‘data1’, ‘data2’, ‘data3’),

де columnI – ім’я колонки, а dataI –відповідне значення. Якщо ви збираєтеся вставляти всі значення в порядку, в якому знаходяться колонки таблиці, то можна і не указувати імена колонок, хоча це краще для полегшення читання. Крім того, якщо ви перераховуєте колонки, необов'язково указуватиїх по порядку знаходження в базі даних, треба зберігати відповідність даних цьому порядку. На перший погляд синтаксичні правила видаються дуже простими. Проте і тут криється проблема, пов’язана з синтаксисом виразу query_expression. Навіть якщо обмежитися простою складовою цієї конструкції (simple_table), то ми маємо наступні можливості:

simple_table ::= query_specification

table_value_constructor

TABLE table_name

Тим самим, стандарт допускає вставку у вказану таблицю всіх рядків іншої таблиці (варіант table_name). Ця інша таблиця може бути як базовою, так і такою, що представляється. Природно, що в останньому випадку у визначенні представлення не повинне бути посилатисязаслання на таблицю, в яку проводиться вставка. При використанні даного варіанту оператора вставки число колонок таблиці, що вставляється, повинне співпадати з числом колонок таблиці, в яку проводиться вставка, або із числом колонок, вказаних в списку column_commalist, якщо цей список заданий. Типи даних відповідних колонок таблиці, що вставляється, і таблиці, в яку проводиться вставка, повинні бути сумісними. Якщо в операції заданий список column_commalist і в ній містяться не всі імена колонок таблиці, в яку проводиться вставка, то в колонки, що залишилися, у всіх рядках

заносяться значення колонок за умовчанням. Якщо для якоїсь із колонок, що залишилися, значення за умовчанням не визначене, при виконанні операції вставки фіксується помилка.

Нехай в базі даних STUD-MARK-EXAM наявна ще одна проміжна таблиця STUD_TSTUD, в якій тимчасово зберігаються дані про студентів, які знаходяться в академічній відпустці. Нехай ця таблиця має наступний заголовок:

Таблиця 5.2 Неповні відомості про студентів

STUD_TSTUD**:**
STUD_NO : STUD_NO
STUD_NAME : VARCHAR
STUD_BDATE : DATE

У таблиці Таблиця 5.2 зберігаються не повні відомості про студентів, а саме ті, які потрібні на час академічної відпустки. Якщо виконати операцію

INSERT INTO STUD(STUD_NO, STUD_NAME, STUD_BDATE) TABLE STUD_TSTUD;

то в основній таблиці STUD з'являться рядки, що відповідають студентам, які знаходяться в академічній відпустці. При цьому в колонках STUD_NO, STUD_NAME, STUD_BDATE цих рядків міститимуться дані, узяті із таблиці STUD_TSTUD, а в колонках STUD_STYPEND , GROUPE_NO, PRO_NO перебуватимуть значення, визначені для даних колонок за умовчанням. Звичайно, оскільки колонка STUD_NO є первинним ключем таблиці STUD (як і таблиці STUD_TSTUD), операція вставки буде успішно виконана лише у тому випадку, коли обмеження первинного ключа таблиці STUD не буде порушено (звичайно ж, потрібне виконання і всіх інших обмежень цілісності, визначених для таблиці STUD).

Далі розглянемо варіант оператора INSERT, в якому набір рядків, що вставляються, визначається явно з використанням синтаксичної конструкції table_value_constructor. Нагадаємо синтаксичні правила, що визначають цю конструкцію:

table_value_constructor ::= VALUES row_value_constructor_comma_list

row_value_constructor ::= row_value_constructor_element

[ROW] (row_value_constructor_element_comma_list)

row_subquery

row_value_constructor_element ::= value_expression

NULL DEFAULT

Найпростіший приклад використання цього варіанту оператора вставки полягає в занесенні в таблицю STUD даних, явно задані про нового студента :

INSERT INTO STUD

ROW ('УК-3128', 'Осередько', '1987-04-08', 300.00, 'УК-31', 772,'Київ');

В даному прикладі явно визначені значення всіх колонок рядка, що заноситься (за правилами, ключове слово ROW можна опустити). Можливий і такий варіант

INSERT INTO STUD ROW ('УК-3128', DEFAULT, NULL, DEFAULT, NULL, NULL);

В цьому випадку ми знаємо про нового студента дуже мало, але упевнені в тому, що його ім'я і розмір стипендії повинні бути призначені за умовчанням, а про дату народження, код групи і код дисципліни та місце народження нічого не відомо. Зверніть увагу, що виконання подібної операції не порушує обмеження цілісності таблиці *STUD.*

Якщо мати повну інформацію про визначення таблиці *STUD,* то формулювання операції можна переписати коротше наступним еквівалентним чином :

INSERT INTO STUD (STUD_NO) 'УК-3128';

За правилами в список елементів конструктора рядка можуть входити скалярні запити, тобто запити, результат виконання яких складається з єдиного рядка, що включає єдину колонку. Тому можлива, наприклад, така операція вставки :

```
INSERT INTO STUD VALUES
    ROW (‘УК-3129 ‘ (SELECT STUD_NAME
            FROM STUD
            WHERE STUD_NO = ‘УК-3120’ ),
        '1987-04-08'
        SELECT STUD_STYPEND
            FROM STUD
            WHERE STUD_NO = ‘УК-3120’ ),
        NULL, NULL ),
    ROW (‘УК-3130‘(SELECT STUD_NAME
            FROM STUD
            WHERE STUD_NO = ‘УК-3121’ ),
        '1978-05-09'
        (SELECT STUD_STYPEND
            FROM STUD
            WHERE STUD_NO = ‘УК-3121’ ),
        NULL, NULL );
```

Після виконання цієї операції в таблиці STUD з'являться два нові рядки для студентів з унікальними ідентифікаторами ‘УК-3129’ і ‘УК-3130’, причому першому з них буде привласнено ім'я і розмір стипендії студента з унікальним ідентифікатором ‘УК-3120’, а другому - аналогічні дані про студента з унікальним ідентифікатором ‘УК-3121’.

Нарешті обговоримо варіант оператора вставки, коли набір рядків, що вставляються, визначається через специфікацію запиту. Нехай, наприклад, що потрібно зберегти в окремій таблиці GROUPE_SUMMARY дані про кількість студентів кожної групи, їх максимальні,

мінімальні і сумарні стипендії. Нехай таблиця *GROUPE_SUMMARY* вже створена і має наступний заголовок:

Таблиця 5.3. Дані про стипендії

GROUPE_SUMMARY:
GROUPE_NO : GROUPE_NO
GROUPE_STUD_NO : INTEGER
GROUPE_MAX_STYPEND : STYPENDARY
GROUPE_MIN_STYPEND : STYPENDARY
GROUPE_TOTAL_STYPEND : STYPENDARY

Тоді заповнити таблицю можна за допомогою наступної операції вставки :

```
INSERT INTO GROUPE_SUMMARY
  (SELECT GROUPE_NO, COUNT(*), MAX (STUD_STYPEND )
     MIN (STUD_STYPEND ), SUM (STUD_STYPEND )
   FROM STUD
   GROUP BY GROUPE_NO);
```

5.5.2.Оператор UPDATE для модифікації існуючих рядків в існуючих таблицях

Загальний синтаксис оператора *UPDATE* виглядає таким чином:

```
UPDATE table_name SET update_assignment_commalist
   WHERE conditional_expression
update_assignment ::= column_name = { value_expression  DEFAULT  NULL }
```

Семантика оператора модифікації існуючих рядків визначається таким чином:

1.Для всіх рядків таблиці з ім'ям table_name обчислюється булевий вираз *conditional_expression*. Рядки, для яких значенням цього булевого виразу є *true*, вважаються такими, що підлягають модифікації (позначимо множину таких рядків через T_m);

2.Кожен рядок s (s T_m) піддається модифікації таким чином, що значення кожної колонки цього рядка, вказаного в списку `update_assignment_commalist`, замінюється значенням, вказаним в правій частині відповідного елементу списку модифікації. Значення колонок рядка `s`, не вказані в списку модифікації, залишаються незмінними.

Приведемо приклади операцій модифікації таблиць.

Перевести, всіх студентів, навчаються в групі УК-52 , у групу УК-51 і призначити їм стипендію розміром 300 грн.

```
UPDATE STUD SET GROUPE_NO = 'УК-51', STUD_STYPEND  = STUD_STYPEND  +
10.00
```

WHERE GROUPE_NO = 'УК-52';

При виконанні даної операції на першому кроці в таблиці *STUD* будуть знайдені всі рядки, що відносяться до студентів, які вчаться в групі 'УК-52' . На другому кроці у всіх цих рядках значення колрнки GROUPE_NO буде змінено на 'УК-51' а до значення колонки STUD_STYPEND буде додано 300.00. Для студентів всіх груп, встановити розмір стипендії, на 10 грн. вище середнього розміру стипендії груп:

UPDATE STUD SET STUD_STYPEND = (SELECT AVG (STUD1_STYPEND)
FROM STUD STUD1
WHERE STUD.GROUPE_NO = STUD1.GROUPE_NO)
+ 10.00

Ці приклади дозволяють зрозуміти, наскільки багаті можливості оператора *UPDATE*. У розділі *WHERE* може міститися будь-яка умова, що допускається в операторові вибірки, а в елементах списку розділу *SET* може бути присутнім будь-який вид *value_expression*, зокрема будь-який запит, що виробляє одиночне значення (скалярний підзапит).

5.5.3.Оператор DELETE для видалення рядків в існуючих таблицях

Загальний синтаксис оператора *DELETE* виглядає таким чином:

DELETE FROM table_name
WHERE conditional_expression

В певному розумінні оператор *DELETE* є окремим випадком оператора *UPDATE* (або, навпаки, дія оператора *UPDATE* є комбінацією дій операторів *DELETE* і *INSERT*).

Семантика оператора модифікації існуючих рядків визначається таким чином:

1. для всіх рядків таблиці з ім'ям table_name обчислюється булевий вираз conditional_expression. Рядки, для яких значенням цього булевого виразу є true, вважаються такими, що підлягають видаленню(позначимо множину таких рядків через `Td`);
2. кожен рядок `Td` видаляється зі вказаної таблиці.

З метою ілюстрації приведемо два приклади операції видалення рядків.

DELETE FROM STUD WHERE COURS_NO= 450;

Видалити із таблиці STUD всі рядки, що відносяться до студентів, які здали іспит з дисципліни номер 450.

DELETE FROM STUD (SELECT STUD.NAME, COUNT(MARK.MARK) FROM STUD INNER JOIN EXAM INNER JOIN MARK ON STUD.Stud_id = MARK.Stud_id AND Exam.Exam_id = MARK.Exam_id AND MARK.MARK='FX'
GROUP BY STUD.NAME HAVING COUNT(MARK.MARK) >2)

Видалити із таблиці STUD всі рядки, що відносяться до студентів, у яких кількість оцінок FX, отриманих за сесію перевищує 2. Як і в операторі *UPDATE*, в розділі WHERE оператора *DELETE* можна використовувати будь-який вид булевого виразу, дозволеного в операторі вибірки. Тому можливості оператора видалення рядків обмежені лише фантазією користувача. З цього приводу варто лише додати, що запит *DELETE* видаляє цілі рядки і стандарт SQL не передбачає жодних відкочень типу *UNDO*, як це реалізовано в текстових процесорах(наприклад MS WORD). Тому будьте обережні!

Запит *DELETE* повністю видаляє рядок із бази даних. Якщо треба видалити одне - єдине поле, то потрібно використовувати запит *UPDATE* і встановити для цього поля значення, яке буде аналогом *NULL* у вашій програмі. Будьте уважні, і обмежуйте ваш запит *DELETE* умовою *WHERE*, інакше ви можете втратити весь зміст таблиці.

DELETE FROM table_name WHERE column1 = 'data1';

Як тільки рядок був видалений із вашої бази даних, він не підлягає відновленню, тому бажано мати колонку на ім'я «IsActive», або щось на зразок цього, який ви можете змінити на нуль, що вказуватиме на блокування представлення даних із цього рядка.

Контрольні вправи та запитання

1.Які дані будуть отримані в результаті виконання запиту?

```
SELECT *
FROM STUDENT
WHERE (STIPEND < 100 OR
NOT (BIRTHDAY >= '10/03/1990'
AND STODENT_ID > 1005));
```

2. Які дані будуть отримані в результаті виконання запиту?

```
SELECT *
FROM STUDENT
WHERE NOT ((BIRTHDAY = '10/03/1990' OR STIPEND > 100)
AND STUDENT_ID >= 1005);
```

3. Напишіть запит, що вибирає дані про всі предмети навчання , екзамени з яких здані студентами, що мають ідентифікатори 12 і 32.
4. Напишіть запит на виведення назв предметів навчання, що починаються на букву Б .
5. Напишіть запит, що вибирає відомості про студентів, у яких імена починаються на букви Б або М.
6. Напишіть запит для вибору з таблиці EXAM_MARKS записів у яких відсутні значення оцінок (поле MARK).
7. Напишіть запит, який виконує виведення (для кожного предмета навчання) найменування предмету і максимального значення номеру семестру, в якому цей предмет викладається.
8. Напишіть запит, який виконує виведення даних для кожного конкретного дня складання іспиту про кількість студентів, що складали іспит цього дня.

9. Напишіть запит для отримання середнього балу для кожного курсу по кожному предмету.
10. Напишіть запит для отримання середнього балу для кожного студента.
11. Напишіть запит для отримання середнього балу для кожного іспиту.
12. Напишіть запит, що вибирає дані про імена всіх студентів що мають по предмету з ідентифікатором 101 бал вищий за загальний середній бал.
13. Напишіть запит, який виконує вибірку імен всіх студентів, що мають по предмету з ідентифікатором 102 бал нижчий за загальний середній бал
14. Напишіть запит, що виконує виведення кількості предметів, по яких іспитувався кожен студент, що здавав більше 20 предметів.
15. Напишіть команду SELECT, що використовує зв'язані підзапити і що виконує виведення імен і ідентифікаторів студентів
16. Напишіть запит, який виконує виведення даних про прізвища студентів, що складали іспити, разом з найменуваннями кожного зданого ними предмету навчання.
17. Напишіть запит на видачу для кожного студента назв всіх предметів навчання, по яких цей студент отримав оцінку 4 або 5.
18. Напишіть запит на видачу даних про назви всіх предметів, по яких студенти отримали тільки гарні (4 і 5) оцінки.
19. Напишіть команду, яка вводить в таблицю SUBJECT рядок для нового предмету навчання з наступними значеннями полий:
20. SEMESTER = 4; SUBJ_NAME = 'Інформатика'; HOUR = 72; SUBJ_ID =201.
21. Введіть запис для нового студента, якого звуть Сабадин Михайло, що навчається на першому курсі НТУУ «КПІ», що народився у Вінниці, дані про дату народження і розмір стипендії невідомі.
22. Напишіть команду, що видаляє з таблиці EXAM_MARKS записи про всі оцінки студента, ідентифікатор якого дорівнює 100.

РОЗДІЛ 6. Розширені можливості SQL та способи реалізації

Криши, ламай, трощи стереотипи!
Вони кричать, пручаються ,- ламай!
Хоч давня звичка з профілем Ксантиппи
Благає, плаче, просить: «Не займай!»
(Ліна Костенко)

Мова SQL, як інструмент розробки та експлуатації інформаційних систем високого рівня складності потребувала розробки суттєвого розширення можливостей у порівнянні з базовими. Деякі з них розлянемо нижче.

6.1. Мова адміністрування БД (МАБД)

Розділ мови SQL, який передбачає набір інструментальних засобів для адміністрування даних, змінами в базі даних та правилами виконання інструкцій SQL. Це декілька груп операторів, досить різних за функціональністю, концептуально поєднаних тільки ідеєю адміністрування. Зокрема це:

6.1.1. Група Data Control Language (DCL)

Оператори Data Control Language, застосовуються для здійснення адміністративних функцій, що надають або відміняють право (привілей) використовувати базу даних, таблиці в базі даних, а також виконувати ті або інші операторів SQL. Оператори DCL представлені в табл.

Таблиця 6.1 Оператори DCL

Оператор	Опис
GRANT	Застосовується для надання привілею
REVOKE	Застосовується для відміни привілею

6.1.2. Група Transaction Control Language (TCL)

Оператори Transaction Control Language застосовуються для управління змінами, виконаними групою операторів DML. Оператори TCL представлені в табл.

Таблиця 6.2. Оператори TCL

Оператор	Опис
COMMIT	Застосовується для завершення транзакції і збереження змін в базі даних
ROLLBACK	Застосовується для відкоченняу транзакції і відміни змін в базі даних
SET TRANSACTION	Застосовується для установки параметрів

	доступу до даних в поточній транзакції

6.1.3. Група Cursor Control Language (CCL)

Оператори Cursor Control Language використовуються для визначення курсора, підготовки SQL-виразів для виконання, а також для деяких інших операторів. Оператори CCL представлені в табл.

Таблиця 6.3. Оператори CCL

Оператор	Опис
DECLARE CURSOR	Застосовується для визначення курсора для запиту
OPEN CURSOR	Застосовується для відкриття курсора при отриманні результатів запиту
FETCH	Застосовується для отримання рядка з результатів запиту
CLOSE CURSOR	Застосовується для закриття курсора
PREPARE	Застосовується для підготовки оператора SQL для виконання
EXECUTE	Застосовується для виконання оператора SQL
DESCRIBE	Застосовується для опису підготовленого запиту

Окремі оператори цієї групи можуть з'явитися в конкретних СУБД. Наприклад, оператор EXPLAIN для Microsoft SQL Server 7.0 застосовується для опису плану запиту. Для СУБД Oracle цю функцію виконує оператор *EXPLAIN PLAN.*

6.2.Приклади використання операторів адміністрування

Розглянемо деякі приклади використання операторів адміністрування. Зауважимо спочатку, що ця частина мови SQL виникла під впливом перспектив використання баз даних в мережевому середовищі і тому її конкретна реалізація більше залежить від розробників конкретних СУБД, особливо таких гігантів як IBM, Oracle та Microsoft, ніж інші частини SQL.

6.2.1. Класифікація категорій користувачів СУБД

Спільним для переважної більшості користувачів сучасних СУБД є розуміння, що користувачів треба класифікувати за категоріями, наприклад таку:

- Прикладні програмісти - відповідають за створення програм, що використають базу даних. В розумінні захисту даних програміст може бути як користувачем, що має привілею створення об'єктів даних і маніпулювання ними, так і користувачем, що має привілею тільки маніпулювання даними.
- Кінцеві користувачі бази даних - працюють із БД безпосередньо через термінал або робочу станцію. Як правило, кінцеві користувачі мають строго обмежений набір привілеїв маніпулювання даними. Цей набір може визначатися при конфігуруванні інтерфейсу кінцевого користувача й не змінюватися. Політику безпеки в цьому випадку визначає адміністратор безпеки або адміністратор бази даних (якщо це та сама посадова особа).
- Адміністратори баз даних - утворюють особливу категорію користувачів СУБД. Вони створюють самі бази даних, здійснюють технічний контроль функціонування СУБД, забезпечують необхідну швидкодію системи. В обов'язках адміністратора, крім того, входить забезпечення користувачам доступу до необхідного їм даним, а також написання (або надання допомоги у визначенні) необхідних користувачеві зовнішніх подань даних. Адміністратор визначає правила безпеки й цілісності даних.

В принципі цю класифікацію можна уточнювати та деталізувати для сучасних СУБД. Розглянемо як приклад реалізацію системи адміністрування в MS SQL Server. Для одержання доступу до даних необхідно, щоб обліковий запис користувача відповідав якомусь користувачу бази даних (database user). Користувач бази даних – сукупність дозволів та обмежень на роботу з даними в конкретній базі даних. На другому етапі обліковий запис користувача перетворюється в користувача бази даних, і одержує всі привілеї, що відповідають цьому користувачеві бази даних. Другий етап зачіає систему безпеки конкретної бази даних, а не всього сервера СУБД. У різних базах даних однієї й того ж облікового запису можуть відповідати однакові або різні імена користувача бази даних з різними правами доступу. У тому випадку, коли обліковий запис користувача не відображається в користувача бази даних, клієнт все-таки може одержати доступ до бази даних під гостьовим ім'ям *guest* (якщо воно не заборонено адміністратором БД). Гістьовий вхід дозволяє отримати мінімальний доступ до даних тільки в режимі читання. Користувачі баз даних можуть поєднуватися в ролі (іноді - групи) для спрощення керування системою безпеки.

Ролі бази даних поєднують декількох користувачів в адміністративну одиницю й дозволяють призначати права доступу до об'єктів бази даних для ролі, наділяючи цими правами всіх учасників цієї ролі. Розрізняють користувальницькі й вбудовані ролі. Вбудовані ролі створюються автоматично при встановленні сервера СУБД (і не можуть мінятися).

Розрізняють вбудовані ролі рівня СУБД і рівня конкретної бази даних. Так, наприклад, в SQL Server 2000 є такі ролі сервера:

Таблиця 6.1. Ролі сервера в SQL Server 2000

Вбудована роль сервера	**Призначення**
Sysadmin	Може виконувати будь-які дії в SQL Server
Serveradmin	Виконує конфігурування й вимикання сервера
Setupadmin	Управляє зв'язаними серверами та процедурами, що автоматично запускаються при запуску SQL Server
Securityadmin	Управляє обліковими записами й правами на створення бази даних, також може читати журнал помилок.
Processadmin	Управляє процесами, запущеними в SQL
Server Dbcreator	Може створювати й модифікувати бази даних
Diskadmin	Управляє файлами SQL Server
Bukladmin	Може вставляти дані з використанням засобів масового копіювання, не маючи безпосереднього доступу в таблицям

SQL Server 2000 підтримує такі вбудовані ролі рівня бази даних:

Таблиця 6.2. Ролі рівня бази даних

Вбудована роль користувача	**Призначення**
Db_owner	Має усі права в базі даних
Db_accessadmin	Може додавати або видаляти користувачів
Db_securityadmin	Управляє всіма дозволами, об'єктами, ролями й членами ролей
Db_ddladmin	Може виконувати будь-які команди DDL, крім GRANT, DENY, REVOKE
Db_backupoperator	Може виконувати команди DBCC, CHECKPOINT, BACKUP
Db_datareader	Може переглядати будь-які дані в будь-якій таблиці
Db_datawriter	Може модифікувати будь-які дані в будь-якій

	таблиці
Db_denydatareader	Забороняється переглядати дані в будь-який таблиці
Db_denydatawriter	Забороняється модифікувати дані в будь-якій таблиці

Крім зазначених ролей існує ще одна роль, яку не вилучають - public. Учасниками цієї ролі є всі користувачі, що мають доступ до бази даних. Не можна явно вказати учасників цієї ролі, тому що всі користувачі вже включені в неї.

6.2.2. Створення і ліквідація ролей

Для створення нової ролі використовується оператор *CREATE ROLE*, що визначається наступним синтаксичним правилом:

CREATE ROLE role_name

[WITH ADMIN { CURRENT_USER CURRENT_ROLE }]

Ім'я створюваної ролі повинне відрізнятися від будь-якого ідентифікатора авторизації, уже визначеного й збереженого в базі даних. У разі успішного створення ролі деякий *authID* отримує привілей на виконання даної ролі. Якщо в операторові *CREATE ROLE* не міститься розділ *WITH ADMIN*, то привілей на виконання ролі отримує поточний ідентифікатор користувача SQL-сесії, якщо значення цього ідентифікатора відмінне від NULL; інакше привілей на виконання ролі дається поточному імені ролі сесії.

Якщо до складу оператора включається розділ *WITH ADMIN*, то можна вибрати, чи буде власником *ролі authID*, відповідний поточному ідентифікатору користувача *SQL-сесії*, або *authID*, відповідний поточному імені ролі (за умови, що відповідні поточний ідентифікатор або поточне ім'я не містять NULL). Крім того, включення цього розділу означає, що *authID*-власник ролі отримує право на передачу привілею виконання даної ролі іншим *authID*.

Відповідно до стандарту SQL:1999, привілеї, потрібні для виконання оператора *CREATE ROLE,* визначаються в реалізаціях SQL. Наприклад, у деяких реалізаціях виконання цієї операції вирішується адміністратором бази даних.

Існуючу роль можна ліквідовувати за допомогою оператора:

DROP ROLE role_name.

Для виконання цієї операції потрібно, щоб поточний *authID SQL-сесії* прямо або побічно (через ланцюжок ролей) був власником ліквідовуваної ролі. При ліквідації ролі, перш за все, вилучається привілей на її виконання у всіх *authID*, яким даний привілей був раніше переданий.

6.2.3. Передача привілеїв і ролей

Для передачі привілеї і ролей від одних *authID* іншим підтримується оператор *GRANT*, якого ми обговоримо окремо для випадків передачі привілею і передачі ролей.

У разі передачі привілеїв використовується наступний синтаксис оператора *GRANT*:

GRANT { ALL PRIVILEGES privilege_commalist }
* ON privilege_object*
TO { PUBLIC authID_commalist } [WITH GRANT OPTION]
* [GRANTED BY { CURRENT_USER CURRENT_ROLE }]*
privilege ::= SELECT [column_name_commalist]
DELETE
INSERT [column_name_commalist]
UPDATE [column_name_commalist]
REFERENCES [column_name_commalist]
USAGE
TRIGGER
EXECUTE
privilege_object ::= [TABLE] table_name
DOMAIN domain_name
CHARACTER SET character_set_name
COLLATION collation_name
TRANSLATION translation_name

У списку привілеїв можна використовувати *SELECT, DELETE, INSERT, UPDATE, REFERENCES* і *TRIGGER* тільки у тому випадку, коли як об'єкт привілеїв указується таблиця. Відповідно, список привілеїв може складатися з єдиного привілею *USAGE* тільки у тому випадку, коли об'єктом є домен, набір символів, порядок сортування або трансляції. Якщо в списку привілеїв указується більш ніж один привілей, то вони всі передаються вказаним authID, але для цього поточний authID SQL-сесії повинен володіти привілеєм на передачу привілеїв.

Використання ключового слова *ALL PRIVILEGES* замість явного завдання списку привілеїв означає, що передаються всі привілеї доступу до відповідного об'єкту бази даних, якими володіє поточний *authID SQL-сесії*.

Як показує синтаксис, один оператор *GRANT* дозволяє передавати привілеї доступу до одного об'єкту, але у тому випадку, коли об'єктом єтаблиця, різні привілеї можуть передаватися по відношенню до одного і тому ж набору колонкив або до різних наборів. Якщо при вказівці привілеїв *SELECT, DELETE, UPDATE* і *REFERENCES* список імен колонкив не визначається, передаються привілеї по відношенню до всіх колонкив таблиці. Відзначимо, що ці привілеї стосуються всіх існуючих стовпів даної таблиці, а також всіх колонкив, які коли-небудь будуть до неї додані.

Включення в оператора необов'язкового розділу *WITH GRANT OPTION* означає, що одержувачам передаваних привілеїв дається також привілей на подальшу передачу отриманих привілеї, включаючи привілей передачу привілеї. Включення в оператора розділу *GRANTED BY* дозволяє явно вказати, чи передаються привілеї від імені поточного ідентифікатора користувача або ж поточного імені ролі.

При перевірці можливості виконання операції в *SQL-сесії* враховуються привілеї поточного *authID SQL-сесії*, а також привілеї всіх ролей, які передані даному *authID*. Оскільки цим ролям могли бути передані інші *ролі*, що володіють власними *привілеями*, аналіз можливості виконання операції є рекурсивною процедурою.

Якщо один і той же привілей передається більше одного разу одному і тому ж authID2 від імені одного і того ж authID1, то виникає ситуація, звана надмірним дублюючим *привілеєм*. Ця ситуація не викликає додаткових проблем, оскільки надмірна *передача привілею* ігнорується. *Для* анулювання даного привілею у authID2 від імені authID2 потрібне виконання лише однієї операції *REVOKE* . Якщо привілей був один раз переданий authID2 від імені authID1 разом з привілеєм на передачу цьому привілею (*WITH GRANT OPTION*), а іншим разом - без цієї опції (порядок дій не є істотним), то authID2 володіє даним привілеєм і привілеєм на її передачу.

Якщо робиться спроба передачі декількох привілеїв, але відповідний *authID* не володіє жодного з них, то фіксується помилка. Аналогічно, якщо проводиться спроба передачі декількох привілеїв з передачею привілею на передачу привілеїв, але відповідний *authID* не володіє *привілеєм WITH GRANT OPTION* ні для одного з передаваних привілеїв, то фіксується помилка. Нарешті, якщо проводиться спроба передачі декількох привілеїв з передачею привілею на передачу привілеїв і відповідний *authID* володіє привілеєм на передачу тільки частини цих привілеїв, то в результаті виконання операції виробляється попередження, але відповідна частина привілеїв передається з привілеєм *WITH GRANT OPTION*. Привілею конкретному користувачеві можуть бути призначені адміністратором явно, або неявно, наприклад через роль. Вирази керування привілеями:

- призначення привілею:

GRANT привілей [ON об'єкт] TO суб'єкт [WITH GRANT OPTION]

- скасування привілею:

REVOKE привілей [ON об'єкт] FROM суб'єкт

Якщо суб'єкт=користувач, то привілей призначається йому явно. Якщо суб'єкт=роль, то для керування привілеями використаються відповідно:

GRANT ROLE ім'я_ролі [ON об'єкт] TO суб'єкт [WITH GRANT OPTION]

REVOKE ROLE ім'я_ролі [ON об'єкт] FROM суб'єкт

Призначення привілею всім користувачам системи здійснюється в такий спосіб:

GRANT привілей [ON об'єкт] TO PUBLIC

У цьому випадку кожний новий створений користувач автоматично одержить такий привілей. Скасування привілею здійснюється так:

REVOKE привілей [ON об'єкт] FROM PUBLIC

Необхідно мати на увазі, що деякі реалізації, наприклад IBM DB2, використають групи користувачів, певні в операційній системі. Тому варто звертати увагу на особливості реалізації аналогів ролей у СУБД. Потрібно з'ясувати, чи містить реалізація SQL-вирази:

CREATE ROLE ім'я_ролі

DROP ROLE ім'я_ролі

Приклад 6.1. Використання ролей.

Створимо роль *stud* і включимо в цю роль двох користувачів *user1* і *user2*:

sp_addrole 'student'

sp_addrolemember 'student','user1'

sp_addrolemember 'student','user2'

Надамо права ролі stud і безпосередньо користувачеві user2:

GRANT SELECT, INSERT ON Stud TO student

GRANT SELECT, INSERT ON Stud TO user2

Після виконання цих команд користувачі user1 і user2 можуть виконувати команди вибірки й додавання запису в таблицю Stud. Призупинимо право на виконання вставки в таблицю Stud для ролі stud:

REVOKE INSERT ON Stud TO student

Після виконання попередньої команди користувач user1 втрачає право вставки запису, а user2 зберігає це право, оскільки право вставки надане йому явно. Виконаємо команду

DENY INSERT ON Stud TO student.

Після виконання цієї команди обидва *користувачі* втрачають *права* вставки в таблицю Stud.

Застосування операторів груп *Transaction Control Language (TCL)* та *Cursor Control Language (CCL)* розглянемо дещо далі.

6.3. Уявлення(View)

Однією з таких можливостей є поява так званих уявлень(Views). Їх ідея полягає створенні віртуальних таблиць з можливостю маніпулювання ними як із реальними. Це знижує ризик небажаних змін даних від помилок та спростити виконання запитів.

Визначення:

Уявлення (*view*) - це таблиця, значення якої береться з інших таблиць за допомогою запиту. При цьому нові копії даних не створюються. Коли вміст базових таблиць змінюється, СУБД автоматично перевиконує запити, які створюють View, що приводить до відповідних змін в уявленнях.

Уявлення створюється за допомогою команди:

CREATE VIEW <ім'я_Уявлення> [<ім'я_колонки>...]
AS <запит>

При цьому повинні дотримуватися наступні обмеження:

- уявлення повинне базуватися на єдиному запиті (використання `UNION` не допускається)
- вихідні дані запиту, що формує уявлення, повинні бути не впорядковані (використання `ORDER BY` не допускається)

Створимо уявлення, що в якому міститься інформація про студентів, здані ними екзамени та отримані оцінки :

CREATE VIEW studentHM AS
SELECT stud.name,mark.mark,course.coursetitle,course.annotation
FROM stud,exam,mark,course
WHERE stud.stud_id=mark.stud_id AND exam.exam_id=mark.exam_id AND course.course_id=exam.course_id

Тепер будь-який користувач, чиїх прав на доступ до даного уявлення достатньо, може здійснювати вибірку даних *studentHM.* Наприклад:

SELECT name,annotation FROM studentHM WHERE mark ='A'

SELECT name,count(coursetitle) FROM studentHM GROUP BY name

(Права користувачів на доступ до уявлення призначаються також за допомогою команд *GRANT / REVOKE*.)

Із приведеного вище прикладу досить зрозуміло показують переваги використання уявлень. Якщо запити типу «вибрати всі екзамени даного студента із вказанням назви предмета» виконуються досить часто, то створення таблиці *studentHM,* що представляється, значно

скоротить ресурсні витрати на виконання з'єднання чотирьох базових таблиць *stud, exam*, mark, course. Крім того, в уявленні може бути представлена інформація, що явно не зберігається жодній із базових таблиць. Наприклад, одна із колонок уявлення, може бути обчислювана:

```
CREATE VIEW amount (name, av_mark) AS
SELECT stud.name, avarage(mark.mark)
   FROM stud,mark
   WHERE stud.stud_id=mark.stud_id
   GROUP BY name;
```

Тут використана ще одна, раніше не описана, можливість SQL - присвоєння нових імен колонкам уявлення. У приведеному прикладі середній бал кожного студента буде зберігаєтися в колонці з ім'ям *av_mark.* Зауважимо, що якщо ми хочемо привласнити нові імена колонкам уявлення, потрібно вказувати імена для всіх колонок. Тип даних колонки уявлення та його нульовий статус завжди залежать від того, як він був визначений в базовій таблиці (таблицях). Запит на вибірку даних до уявлення виглядає абсолютно аналогічно запиту до будь-якої іншої таблиці. Однак на зміну даних в уявленні накладаються обмеження. Стисло про них можна сказати таке:

- Якщо уявлення береться з однієї таблиці, зміни даних в ньому допускаються. При цьому змінюються дані в пов'язаній з ним таблиці.
- Якщо уявлення береться з більш ніж однієї таблиці, то зміни даних в ньому не допускаються, оскільки в більшості випадків СУБД не може правильно відновити схему базових таблиць із схеми уявлення(характерний приклад зворотньої задачі).

Видалення уявлення проводиться за допомогою оператора:

```
DROP VIEW <ім'я_представлення>
```

6.4. Поняття збереженої процедури

Є групи зв'язаних між собою операторів SQL, застосування яких робить роботу програміста легшою і гнучкішою, оскільки виконати збережену процедуру , часто виявляється набагато простішим, ніж послідовність окремих операторів SQL. Збережені процедури, є набором команд, що складається з одного або декількох операторів SQL або функцій і зберігається в базі даних у відкомпілюваному вигляді. Виконання в базі даних збережених процедур, замість окремих операторів SQL дає користувачеві наступні переваги:

- необхідні оператори вже містяться в базі даних*;*

•всі вони пройшли етап синтаксичного аналізу і знаходяться у виконуваному форматі; перед виконанням збережену процедуру сервер SQL генерує для неї план виконання, виконує її оптимізацію і компіляцію;

•збережені процедури, підтримують модульне програмування, оскільки дозволяють розбивати великі завдання на самостійні, дрібніші і зручніші в управлінні частини;

•збережені процедури, можуть викликати інші збережені процедури, і функції;

•збережені процедури, можуть бути викликані з прикладних програм інших типів;

•як правило, збережені процедури, виконуються швидше, ніж послідовність окремих операторів;

•збережені процедури, простіше використовувати: вони можуть складатися з десятків і сотень команд, але для їх запуску досить вказати всього лише ім'я потрібної процедури, що зберігається. Це дозволяє зменшити розмір запиту, що посилається від клієнта на сервер, а значить, і навантаження на мережу.

Зберігання процедур в тому ж місці, де вони виконуються, забезпечує зменшення об'єму передаваних по мережі даних і підвищує загальну продуктивність системи. Зазвичай всі обмеження цілісності у вигляді правил і алгоритмів обробки даних реалізуються на сервері баз даних і доступні кінцевому застосуванню у вигляді набору збережених процедур, які і представляють інтерфейс обробки даних. Для забезпечення цілісності даних, а також для безпеки, прикладні програми зазвичай не дістають прямого доступу до даних – вся робота з ними ведеться шляхом виклику тих або інших збережених процедур.

Подібний підхід робить досить простою модифікацію алгоритмів обробки даних, що негайно ж стають доступними для всіх користувачів мережі, і забезпечує можливість розширення системи без внесення змін до прикладної програми: досить змінити процедуру, що зберігається, на сервері баз даних. Розробникові не потрібно перекомпілювати застосування, створювати його копії, а також інструктувати користувачів про необхідність роботи з новою версією. Користувачі взагалі можуть не підозрювати про те, що до системи внесені зміни.

Збережені процедури, існують незалежно від таблиць або яких-небудь інших об'єктів баз даних. Вони викликаються клієнтською програмою, іншою збереженою процедурою, або тригером. Розробник може управляти правами доступу до збереженої процедури, вирішуючи або забороняючи її *виконання*. Змінювати код збереженої процедури, дозволяється тільки її власникові або члену фіксованої ролі бази даних. При необхідності можна передати права володіння нею від одного користувача до іншого.

6.4.1. Збережені процедури в конкретних середовищах

Практичний досвід створення програм обробки даних показує, що є ряд операцій над даними, які реалізують загальну для всіх користувачів логіку доцільно винести на сервер. Однак, для написання процедур, що реалізують ці операції стандартних можливостей SQL не досить, оскільки тут необхідні оператори обробки розгалужень, циклів і т.д. Тому багато постачальників СУБД пропонують власні процедурні розширення SQL (наприклад PL/SQL компанії Oracle і т.д.). Ці розширення містять логічні оператори (IF ... THEN ... ELSE), оператори переходу за умовою (SWITCH ... CASE ...), оператори циклів (FOR, WHILE, UNTIL) і оператори предачі керування в процедури (CALL, RETURN). За допомогою цих засобів створюються функціональні модулі, які зберігаються на сервері разом з базою даних. Звичайно такі модулі називають збереженими процедурами. Вони можуть бути викликані з передачею параметрів будь-яким користувачем, що має на те відповідні права. У деяких системах збережені процедури можуть бути реалізовані й у вигляді зовнішніх стосовно СУБД модулів на мовах загального призначення, таких як C або Pascal. Приклад для СУБД PostgreSQL:

CREATE FUNCTION <ім'я_функції>
([<тип_параметра1>,...<тип_параметра2>])
RETURNS < типи, що повертаються>
AS [<SQL_оператор> <ім'я_об'єктного_модуля>]
LANGUAGE 'SQL' 'C' 'internal'

Збережені процедури, є повноцінними об'єктами бази даних, а тому кожна з них зберігається в конкретній базі даних. Безпосередній виклик процедури, що зберігається, можливий, тільки якщо він здійснюється в контексті тієї бази даних, де знаходиться процедура.

6.4.2. Типи збережених процедур

У SQL Server є декілька типів збережених процедур.

- Системні збережені процедури призначені для виконання різних адміністративних дій. Можна сказати, що системні збережені процедури, є інтерфейсом, що забезпечує роботу з системними таблицями, яка, кінець кінцем, зводиться до зміни, додавання, видалення і вибірки даних з системних таблиць як призначених для користувача, так і системних баз даних. Системні збережені процедури, мають префікс `sp_`, зберігаються в системній базі даних і можуть бути викликані в контексті будь-якої іншої бази даних.

- Призначені для користувача збережені процедури, реалізують ті або інші дії. збережені процедури, – повноцінний об'єкт бази даних. Внаслідок цього кожна процедура, що зберігається, розташовується в конкретній базі даних, де і виконується.
- Тимчасові збережені процедури, існують лише якийсь час, після чого автоматично знищуються сервером. Вони діляться на локальні і глобальні. Локальні тимчасові збережені процедури, можуть бути викликані тільки з того з'єднання, в якому створені. При *створенні* такої процедури їй необхідно дати ім'я, що починається з одного символу #. Як і всі тимчасові об'єкти, процедури цього типу, що зберігаються, автоматично віддаляються при відключенні користувача, перезапуску або зупинці сервера. Глобальні тимчасові збережені процедури, доступні для будь-яких з'єднань сервера, на якому є така ж процедура. Для її визначення досить дати їй ім'я, що починається з символів ##. Видаляються ці процедури при перезапуску або зупинці сервера, а також при закритті з'єднання, в контексті якого вони були створені.

6.4.3. Створення, зміна і видалення збережених процедур

Створення процедури, що зберігається, вимагає вирішення наступних задач:

- визначення типу створюваної процедури, що зберігається: тимчасова або призначена для користувача. Окрім цього, можна створити свою власну системну процедуру, що зберігається, призначивши їй ім'я з префіксом *sp_* і помістивши її в системну базу даних. Така процедура буде доступна в контексті будь-якої бази даних локального сервера;
- планування прав доступу. При створенні процедури, що зберігається, слід враховувати, що вона матиме ті ж має рацію доступу до об'єктів бази даних, що і користувач, що створив її;
- визначення параметрів процедури, що зберігається. Подібно до процедур, що входять до складу більшості мов програмування, збережені процедури, можуть мати *вхідн*і та вихідні параметри;
- розробка коду процедури, що зберігається. Код процедури може містити послідовність будь-яких команд SQL, включаючи виклик інших збережених процедур.

Створення нової і зміна наявної процедури, що зберігається, здійснюється за допомогою наступної команди:

<Визначення_процедури>::=

{CREATE ALTER } PROC[EDURE] ім'я_процедури

[;номер]

[{@ім'я_параметра тип_даних } [VARYING]

[=default][OUTPUT]][,...n]

[WITH { RECOMPILE ENCRYPTION RECOMPILE

ENCRYPTION }]

[FOR REPLICATION]

AS

sql_оператор [...n]

Розглянемо параметри даної команди.

Використовуючи префікси *sp_ #, ##,* створювану процедуру можна визначити як системною або тимчасовою. Як видно з синтаксису команди, не допускається указувати ім'я власника, якому належатиме створювана процедура, а також ім'я бази даних, де вона повинна бути розміщена. Так, щоб розмістити створювану процедуру, що зберігається, в конкретній базі даних, необхідно виконати команду *CREATE PROCEDURE* в контексті цієї бази даних. При зверненні з тіла процедури, що зберігається, до об'єктів тієї ж бази даних можна використовувати укорочені імена, тобто без вказівки імені бази даних. Коли ж потрібно звернутися до об'єктів, розташованих в інших базах даних, вказівка імені бази даних обов'язково.

Номер в імені – це ідентифікаційний номер процедури, що зберігається, що однозначно визначає її в групі процедур. Для зручності управління процедурами логічно однотипні збережені процедури, можна групувати, привласнюючи їм однакові імена, але різні ідентифікаційні номери. Для передачі вхідних і вихідних даних у створюваній збереженій процедурі, можуть використовуватися параметри, імена яких, як і імена локальних змінних, повинні починатися з символу @. У одній збереженій процедури, можна визначити множину параметрів, розділених комами. У тілі процедури не повинні застосовуватися локальні змінні, чиї імена співпадають з іменами параметрів цієї процедури.

Для визначення типу даних, який матиме відповідний параметр процедури, що зберігається, годяться будь-які типи даних SQL, включаючи визначені користувачем. Проте тип даних CURSOR може бути використаний тільки як вихідний параметр процедури, що зберігається, тобто з вказівкою ключового слова OUTPUT. Наявність ключового слова OUTPUT означає, що відповідний параметр призначений для повернення даних з процедури, що зберігається. Проте це зовсім не означає, що параметр не підходить для передачі значень в процедуру, що зберігається. Вказівка ключового слова OUTPUT наказує серверу при виході з процедури, що зберігається, привласнити поточне значення параметра локальної змінної, яка була вказана при виклику процедури як значення параметра. Відзначимо, що при вказівці ключового слова *OUTPUT* значення відповідного параметра при виклику процедури

може бути задане тільки за допомогою локальної змінної. Не вирішується використання будь-яких виразів або констант, допустиме для звичайних параметрів.

Ключове слово *VARYING* застосовується спільно з параметром *OUTPUT,* що має тип *CURSOR*. Воно визначає, що вихідним параметром буде результуюча множина. Ключове слово *DEFAULT* є значення, яке прийматиме відповідний параметр за умовчанням. Таким чином, при виклику процедури можна не указувати явно значення відповідного параметра. Оскільки сервер кешує план виконання запиту і компілює код, при подальшому виклику процедури використовуватимуться вже готові значення. Проте в деяких випадках все ж таки потрібно виконувати перекомпіляцію коду процедури. Вказівка ключового слова *RECOMPILE* наказує системі створювати план виконання процедури, що зберігається, при кожному її виклику. Параметр *FOR REPLICATION* затребуваний при реплікації даних і включенні створюваної процедури, що зберігається, як стаття в публікацію. Ключове слово *ENCRYPTION* наказує серверу виконати шифрування коду процедури, що зберігається, що може забезпечити захист від несанкціонованого використання алгоритмів, які реалізовують дію процедури, що зберігається. Ключове слово AS розміщується на початку власного тіла процедури, що зберігається, тобто набору команд SQL, за допомогою яких і реалізовуватиметься та чи інша дія. У тілі процедури можуть застосовуватися практично всі команди SQL, оголошуватися транзакції, встановлюватися блокування і викликатися інші збережені процедури. Вихід з процедури, що зберігається, можна здійснити за допомогою команди RETURN. Видалення процедури, що зберігається, здійснюється командою:

DROP PROCEDURE { ім'я_процедури} [,...n]

6.4.4. Виконання збереженої процедури

Для виконання збереженої процедури, використовується команда:

[[EXEC [UTE] ім'я_процедури [;номер]
[[@ім'я_параметра=]{ значення @ім'я_переменной}
[OUTPUT][DEFAULT]][,...n]

Якщо виклик збереженої процедури, не є єдиною командою в пакеті, то присутність команди *EXECUTE* обов'язкова. Більш того, ця команда потрібна для виклику процедури з тіла іншої процедури або тригера. Використання ключового слова OUTPUT при виклику процедури вирішується тільки для параметрів, які були оголошені при створенні процедури з ключовим словом *OUTPUT.*Коли ж при виклику процедури для параметра указується ключове слово *DEFAULT*, то буде використано значення за умовчанням. Природно, вказане

слово *DEFAULT* дозволяється тільки для тих параметрів, для яких визначено значення за умовчанням.

З синтаксису команди *EXECUTE* видно, що імена параметрів можуть бути опущені при виклику процедури. Проте в цьому випадку користувач повинен указувати значення для параметрів в тому ж порядку, в якому вони перераховувалися при створенні процедури. Привласнити параметру значення за умовчанням, просто пропустивши його при перерахуванні не можна. Якщо ж потрібно опустити параметри, для яких визначено значення за умовчанням, достатньо явної вказівки імен параметрів при виклику збереженої процедури. Більш того, у такий спосіб можна перераховувати параметри і їх значення в довільному порядку. Відзначимо, що при виклику процедури указуються або імена параметрів із значеннями, або тільки значення без імені параметра. Їх комбінування не допускається.

Приклад 6.2. Процедура без параметрів. Розробити процедуру для отримання назв дисциплін і оцінок на екзаменах, студентом Закревським.

```
CREATE PROC my_proc1
AS
SELECT course.coursetitle       Exam.mark
  AS Mark stud.name
  FROM stud INNER JOIN
  (exam INNER JOIN
  ON stud.stud_id=exam.stud_id)
  ON exam.course_id=course.course_id
  WHERE stud.name=’Закревський’
```

Приклад 6.3. Процедура для отримання назв дисциплін і оцінок , отриманих студентом Закревським.

Для звернення до процедури можна використовувати команди:

EXEC my_proc1 або my_proc1

Процедура повертає набір даних.

Приклад 6.4. Процедура без параметрів. Створити процедуру для збільшення стипендії студентам третього курсу на 10%.

```
CREATE PROC my_proc2
AS
UPDATE STUD SET STUD_STYPEND  =STUD_STYPEND  *1.1
  WHERE StudYear=’3’
```

Приклад 6.5. Процедура збільшення стипендії студентам третього року навчання на 10%.

Для звернення до процедури можна використовувати команди:

EXEC my_proc2 або my_proc2

Процедура не повертає жодних даних.

Приклад 6.6. Процедура з вхідним параметром. Розробити процедуру для отримання назв дисциплін і оцінок на екзаменах, певним студентом

```
CREATE PROC my_proc3
  @k VARCHAR(20)
AS
SELECT course.coursetitle       Exam.mark
   AS Mark stud.name
  FROM stud INNER JOIN
  (exam INNER JOIN exam
  ON stud.stud_id=exam.stud_id)
  ON exam.course_id=course.course_id
  WHERE Stud.Name=@k
```

Приклад 6.7. Процедура для отримання назв дисциплін і оцінок , отриманих заданим студентом.

Для *звернення до процедури* можна використовувати команди:

EXEC my_proc3 'Закревський' або

my_proc3 @k='Закревський'

Приклад 6.8. Процедура з вхідними параметрами. Процедура збільшення стипендії студентам певного року навчання на заданий %.

```
CREATE PROC my_proc4
  @t VARCHAR(2) @p FLOAT
AS
UPDATE STUD SET STUD_STYPEND  =STUD_STYPEND *(1+@p)
  WHERE StudYear=@t
```

Для звернення до процедури можна використовувати команди:

EXEC my_proc4 '3',0.15 або

EXEC my_proc4 @t='3', @p=0.15

Приклад. 6.9. Використання вкладених процедур. Створити процедуру для визначення середній бал, групи, в якій перебуває заданий студент

Спочатку розробимо процедуру для визначення фірми, де працює співробітник.

```
CREATE PROC my_proc7
  @n VARCHAR(20)
  @f VARCHAR(20) OUTPUT
AS
SELECT @f=Group
```

```
  FROM stud
  WHERE Name=@n
```

Потім створимо процедуру, що підраховує загальну кількість товару, який куплений фірмою, що цікавить нас.

```
CREATE PROC my_proc8
  @fam VARCHAR(20)
  @avmark FLOAT OUTPUT
AS
DECLARE @firm VARCHAR(20)
EXEC my_proc7 @fam,@group OUTPUT
SELECT @avmark=Average(Mark.mark)
  FROM stud INNER JOIN exam
  ON stud.stud_id=exam.stud_id
  GROUP BY stud.group
  HAVING stud.group=@firm
```

Приклад. 6.10. Створення процедури для визначення загальної кількості товарів, придбаних фірмою, в якій працює заданий співробітник.

Виклик процедури здійснюється за допомогою команди:

```
DECLARE @k INT
EXEC my_proc8 ‘Закревський’,@k OUTPUT
SELECT @k
```

6.5.Визначення тригера в стандарті мови SQL

Тригери є одним з різновидів збережених процедур. Їх виконання відбувається при виконанні для таблиці якого-небудь оператора мови маніпулювання даними (DML). Тригери використовуються для перевірки цілісності даних, а також для відкочення транзакцій. Тригери – це SQL-процедура, що відкомпілювалася, виконання якої обумовлене настанням певних подій усередині реляційної бази даних. Застосування тригерів як правило досить зручне для користувачів бази даних. Та все ж їх використання часто пов'язане з додатковими витратами ресурсів на операції введення/виведення. У тому випадку, коли таких же результатів (з набагато меншими непродуктивними витратами ресурсів) можна добитися за допомогою збережених процедур, або прикладних програм, застосування *тригерів* недоцільне. Тригери – особливий інструмент SQL-сервера, використовуваний для підтримки цілісності даних в базі даних. За допомогою обмежень цілісності, правил і значень за умовчанням не завжди можна добитися потрібного рівня функціональності. Часто потрібно

реалізувати складні алгоритми перевірки даних, що гарантують їх достовірність і реальність. Крім того, іноді необхідно відстежувати зміни значень таблиці, щоб потрібним чином змінити зв'язані дані.

Тригер є спеціальний тип збережених процедур, які запускаються сервером автоматично при спробі змінити дані в таблицях, з якими *тригери* пов'язані. Кожен *тригер* прив'язується до конкретної таблиці. Всі модифікації даних розглядаються як одна транзакція. У разі виявлення помилки або порушення цілісності даних відбувається відкочення цієї транзакції. Тим самим внесення змін забороняється. Відміняються також всі зміни, вже зроблені тригером. Створює тригер тільки власник бази даних. Це обмеження дозволяє уникнути випадкової зміни структури таблиць, способів зв'язку з ними інших об'єктів і т.п. Тригер є досить корисним і в той же час небезпечним засобом. Так, при неправильній логіці його роботи можна легко знищити всю базу даних, тому тригери необхідно дуже ретельно відлагоджувати.

На відміну від звичайної підпрограми, тригер виконується неявно в кожному випадку виникнення події тригера, до того ж він не має аргументів. Приведення його в дію іноді називають запуском тригера. За допомогою тригерів досягаються наступні цілі:

- додатковий засіб обмежень цілісності даних перевірка коректності введених даних і виконання, які важко або неможливо підтримувати за допомогою обмежень цілісності, встановлених для таблиці;
- видача попереджень, що нагадують про необхідність виконання деяких дій при оновленні таблиці, реалізованому певним чином;
- збір даних про внесені зміни і тих осіб, які їх виконали;

Основний формат команди CREATE TRIGGER показаний нижче:

<Визначення_тригера>::= CREATE TRIGGER ім'я_тригера
BEFORE AFTER <тригерна_подія>
ON <ім'я_таблиці>
[REFERENCING
<список_старих_або_нових_псевдонімів>]
[FOR EACH { ROW STATEMENT}]
[WHEN(умова_тригера)]
<тіло_тригера>.

Події тригерів складаються зі вставки, видалення і оновлення рядків в таблиці. У останньому випадку для події тригера можна вказати конкретні імена колонок таблиці. Час запуску *тригера* визначається за допомогою ключових слів *BEFORE* (тригер запускається до виконання пов'язаних з ним подій) або *AFTER* (після їх виконання). Виконувані тригером дії визначаються для кожного рядка (*FOR EACH ROW*), охопленого даною подією, або тільки

один раз для кожної події (*FOR EACH STATEMENT*). Позначення <список_старих_або_нових_псевдонімів> стосується таких компонентів, як старий або новий рядок (*OLD / NEW*) або стара або нова таблиця (*OLD TABLE / NEW TABLE*). Ясно, що старі значення не застосовні для подій вставки, а нові – для подій видалення.

За умови правильного використання тригери можуть стати дуже потужним механізмом. Основна їх перевага полягає в тому, що стандартні функції зберігаються усередині бази даних і погоджено активізуються при кожному її оновленні. Це може істотно спростити прикладну програму. Проте слід згадати і про властивих тригеру недоліках:

- складність: при переміщенні деяких функцій в базу даних ускладнюються завдання її проектування, реалізації і адміністрування;
- прихована функціональність: перенесення частини функцій в базу даних і збереження їх у вигляді одного або декількох *тригерів* іноді приводить до приховування від користувача деяких функціональних можливостей, що може привести до небажаних і шкідливих побічних ефектів.
- вплив на продуктивність: перед виконанням кожної команди по зміні стану бази даних СУБД повинна перевірити умову тригера з метою з'ясування необхідності запуску *тригера* для цієї команди, що знижує продуктивність, особливо, якщо навантаження пікове.

6.6.Реалізація тригерів в середовищі MS SQL Server

Для реалізації СУБД MS SQL Server використовується наступний оператор створення або зміни *тригера*:

<Визначення_тригера>::=
{CREATE ALTER} TRIGGER ім'я_тригера
ON { ім'я_таблиці ім'я_просмотра }
[WITH ENCRYPTION]
{
{ { FOR AFTER INSTEAD OF }
{ [DELETE] [,] [INSERT] [,] [UPDATE] }
[WITH APPEND]
[NOT FOR REPLICATION]
AS
sql_оператор[...n]
}
{ {FOR AFTER INSTEAD OF } { [INSERT] [,]
[UPDATE] }

[WITH APPEND]

[NOT FOR REPLICATION]

AS

{ IF UPDATE(ім'я_колонки)

[{AND OR} UPDATE(ім'я_колонки)] [...n]

IF (COLUMNS_UPDATES(){ оператор_біт_обробки}

біт_маска_змін)

{ оператор_біт_порівняння }біт_маска [...n]}

sql_оператор [...n]

}

}

Тригер може бути створений тільки в поточній базі даних, але допускається звернення усередині тригера до інших баз даних, у тому числі і розташованих на віддаленому сервері.

Розглянемо призначення аргументів з команди *CREATE ALTER TRIGGER*. Ім'я тригера повинне бути унікальним в межах бази даних. Додатково можна вказати ім'я власника. При вказівці аргументу *WITH ENCRYPTION* сервер виконує шифрування коду *тригера*, щоб ніхто, включаючи адміністратора, не міг дістати до нього доступ і прочитати його. Шифрування часто використовується для утаєння авторських алгоритмів обробки даних, що є інтелектуальною власністю програміста або комерційною таємницею.

6.6.1.Типи тригерів

У SQL Server існує два параметри, що визначають поведінку тригерів:

- `AFTER`. Тригер виконується після успішного виконання команд, що викликали його. Якщо ж команди з якої-небудь причини не можуть бути успішно завершені, *тригер* не виконується. Слід зазначити, що зміни даних в результаті виконання запиту користувача і виконання *тригера* здійснюється в тілі однієї транзакції: якщо відбудеться відкочення *тригера*, то будуть відхилені і призначені для користувача зміни. Можна визначити декілька *AFTER*-тригерів для кожної операції (*INSERT, UPDATE, DELETE*). Якщо для таблиці передбачено виконання декількох *AFTER*-тригерів, то за допомогою системної процедури *sp_settriggerorder*, що зберігається, можна вказати, який з них виконуватиметься першим, а який останнім. За умовчанням в SQL Server всі *тригери* є *AFTER*-тригерами.
- `INSTEAD OF`. *Тригер* викликається замість виконання команд. На відміну від `AFTER`-тригера `INSTEAD OF`-тригер може бути визначений як для таблиці, так і для

перегляду. Для кожної операції *INSERT, UPDATE, DELETE* можна визначити тільки один *INSTEAD OF-тригер.*

Тригери розрізняють за типом команд, на які вони реагують.

Існує три типи *тригерів*:

- *INSERT TRIGGER* – запускаються при спробі вставки даних за допомогою команди *INSERT*.
- *UPDATE TRIGGER* – запускаються при спробі зміни даних за допомогою команди *UPDATE*.
- *DELETE TRIGGER* – запускаються при спробі видалення даних за допомогою команди *DELETE.*

Конструкції *[DELETE] [,] [INSERT] [,] [UPDATE] і FOR AFTER INSTEAD OF } { [INSERT] [,] [UPDATE]* визначають, на яку команду реагуватиме *тригер*. При його створенні повинна бути вказана хоч би одна команда. Допускається створення тригера, що реагує на дві або на всі три команди.

Аргумент *WITH APPEND* дозволяє створювати декілька тригерів кожного типу. При *створенні тригера* з аргументом *NOT FOR REPLICATION* забороняється його запуск під час виконання модифікації таблиць механізмами реплікації. Конструкція AS sql_оператор[...n] визначає набір SQL- операторів і команд, які будуть виконані при запуску тригера. Відзначимо, що усередині тригера не допускається виконання ряду операцій, таких, наприклад, як:

- створення, зміна і видалення бази даних;
- відновлення резервної копії бази даних або журналу транзакцій.

Виконання цих команд не дозволене, оскільки вони не можуть бути відмінені у разі відкоту транзакції, в якій виконується тригер. Ця заборона навряд може якимсь чином позначитися на функціональності створюваних тригерів. Важко знайти таку ситуацію, коли, наприклад, після зміни рядка таблиці потрібно буде виконати відновлення резервної копії журналу транзакцій.

6.6.2.Програмування тригера

При виконанні команд додавання, зміни і видалення записів сервер створює дві спеціальні таблиці: *inserted* і *deleted*. У них містяться списки рядків, які будуть вставлені або видалені після закінчення транзакції. Структура таблиць *inserted* і *deleted* ідентична структурі таблиць, для якої визначається тригер. Для кожного тригера створюється свій комплект таблиць *inserted* і *deleted*, тому ніякий інший тригер не зможе дістати до них доступ. Залежно від типу операції, що викликала виконання тригера, вміст таблиць *inserted* і *deleted* може бути різним:

- команда `INSERT` – в таблиці `inserted` містяться всі рядки, які користувач намагається вставити в таблицю; у таблиці `deleted` не буде жодного рядка; після завершення *тригера* всі рядки з таблиці `inserted` перемістяться в початкову таблицю;
- команда *DELETE* – в таблиці *deleted* міститимуться всі рядки, які користувач спробує видалити; *тригер* може перевірити кожен рядок і визначити, чи дозволено її видалення; у таблиці *inserted* не опиниться жодного рядка;
- команда *UPDATE* – при її виконанні в таблиці *deleted* знаходяться старі значення рядків, які будуть видалені при успішному завершенні *тригера.* Нові значення рядків містяться в таблиці *inserted*. Ці рядки додадуться в початкову таблицю після успішного виконання тригера.

Для отримання інформації про кількість рядків, яка буде змінено при успішному завершенні *тригера*, можна використовувати функцію *@@ROWCOUNT*; вона повертає кількість рядків, оброблених останньою командою. Слід підкреслити, що тригер запускається не при спробі змінити конкретний рядок, а у момент виконання команди зміни. Одна така команда впливає на множину рядків, тому *тригер* повинен обробляти всі ці рядки.

Якщо тригер виявив, що з 100 рядків, що вставляються, змінних або таких, що видаляються, тільки одна не задовольняє тим або іншим умовам, то жодний рядок не буде вставлений, змінений або видалений. Така поведінка обумовлена вимогами транзакції – повинні бути виконані або всі модифікації, або жодна.

Тригер виконується неявно як певна транзакція, тому усередині тригера допускається застосування команд управління транзакціями. Зокрема, при виявленні порушення обмежень цілісності для переривання виконання тригера і відміни всіх змін, які намагався виконати користувач, необхідно використовувати команду *ROLLBACK TRANSACTION*.

Для отримання списку колонок, змінених при виконанні команд *INSERT* або *UPDATE*, що викликали виконання тригера, можна використовувати функцію *COLUMNS_UPDATED().* Вона повертає двійкове число, кожен біт якого, починаючи з молодшого, відповідає одній колонці таблиці (в порядку проходження колонок при створенні таблиці). Якщо біт встановлений в значення «1», то відповідна колонка бувла змінена. Крім того, факт зміни клонки визначає і функція UPDATE (ім'я_колонки). Для видалення тригера використовується команда:

DROP TRIGGER { ім'я_тригера} [,...n]

Приведемо приклади використання тригерів.

Приклад 6.11. Використання тригера для реалізації обмежень на значення. Наприклад інформаційна система відслідковує транзакції, що відбуваються на складі супермаркету. Кожна транзакція фіксується в таблиці Операція. Бізнес-логіка цієї операції

проста: кількість проданого товару не може перевищувати залишок з таблиці Склад. Команда вставки запису в таблицю exam може бути, наприклад, такий:

```
INSERT INTO Операція
  VALUES (3,1-299,'08/01/2009')
```

Створюваний тригер повинен відреагувати на її виконання таким чином: необхідно відмінити команду, якщо в таблиці Склад величина залишку товару виявилася менше кількості товару, що продавалася, з введеним кодом (у прикладі код товара=3). У записі, що вставляється, кількість товару указується із знаком «+», якщо товар поставляється, і із знаком «-», якщо він продається. Представлений тригер налаштований на обробку тільки одного запису, що додається.

```
CREATE TRIGGER Тригер_ins
ON Операція FOR INSERT
AS
IF @@ROWCOUNT=1
BEGIN
  IF NOT EXISTS(SELECT *
     FROM inserted
     WHERE -inserted.Кількість<=ALL(SELECT
       Склад.Залишок
    FROM Склад, Операція
    WHERE Склад.КодТовара=
     Операція.КодТовара))
    BEGIN
     ROLLBACK TRAN
    PRINT
     'Відміна постачання: товару на складі немає'
    END
END
```

Приклад 6.12. Використання тригера для збору статистичних даних.

Створити тригер для обробки операції вставки запису в таблицю Операція, наприклад, такої команди:

```
INSERT INTO Операція
  VALUES (3,1,200,'01/08/2002')
```

Поставляється товар з кодом 3 від клієнта з кодом 1 в кількості 200 одиниць. При продажі або отриманні товару необхідно відповідним чином змінити кількість його складського

запасу. Якщо товару на складі ще немає, необхідно додати відповідний запис в таблицю Склад. Тригер обробляє тільки один рядок, що додається.

```
ALTER TRIGGER Тригер_ins
ON Операція FOR INSERT
AS
DECLARE @x INT @y INT
IF @@ROWCOUNT=1
у таблицю Операція додається запис про постачання товару
BEGIN
кількість проданого товару повинна бути не менше, ніж його залишок з таблиці Склад
IF NOT EXISTS(SELECT *
        FROM inserted
        WHERE -inserted.Кількість<  =ALL(SELECT Склад.Залишок
      FROM Склад, Операція
      WHERE Склад.КодТовара=
         Операція.КодТовара))
   BEGIN
     ROLLBACK TRAN
       PRINT 'відкочування товару нема '
    END
```

Якщо запису про поставлений товар ще немає, додається відповідний запис у таблицю Склад:

```
IF NOT EXISTS ( SELECT *
        FROM Склад C, inserted i
        WHERE C.КодТовара=i.КодТовара )
    INSERT INTO Склад (КодТовара,Залишок)
ELSE
```

Якщо запис про товар вже був в таблиці Склад, то визначається код і кількість товару із доданої в таблицю Операція запису

```
BEGIN
  SELECT @y=i.КодТовара, @x=i.Кількість
  FROM Операція C, inserted i
  WHERE C.КодТовара=i.КодТовара і
проводиться зміни кількості товару в таблиці Склад
      UPDATE Склад
      SET Залишок=Залишок+@x
```

```
        WHERE КодТовара=@y
    END
END
```

Приклад 6.13. Створити тригер для обробки операції видалення запису з таблиці Операція, наприклад, такої команди:

```
DELETE FROM Операція WHERE КодУгоди=14
```

Для товару, код якого вказаний при видаленні запису, необхідно відкоригувати його залишок на складі. Тригер обробляє тільки один запис, що видаляється.

```
CREATE TRIGGER Тригер_del
ON Операція FOR DELETE
AS
IF @@ROWCOUNT=1 - видалений один запис
BEGIN
  DECLARE @y INT,@x INT
визначається код і кількість товару з видаленою з таблиці Склад запису
  SELECT @y=КодТовара, @x=Кількість
  FROM deleted
у таблиці Склад коригується кількість товару
    UPDATE Склад
    SET Залишок=Залишок-@x
    WHERE КодТовару=@y
END
```

Приклад 6.14. Створити тригер для обробки операції зміни запису в таблиці Операція, наприклад, такою командою:

```
UPDATE Операція SET Кількість=Кількість-10
WHERE КодТовару=31
```

у всіх операціях з товаром, що має код, дорівнює 31, зменшити кількість товару на 10 одиниць. Вказана команда може привести до зміни відразу декількох записів в таблиці Операція. Тому покажемо, як створити тригер, що обробляє не один запис. Для кожного зміненого запису необхідно для старого (до зміни) коду товару зменшити залишок товару на складі на величину старої (до зміни) кількості товару і для нового (після зміни) коду товару збільшити його залишок на складі на величину нового (після зміни) значення. Щоб обробити всі змінені записи, введемо курсори, в яких збережемо всі старі (з таблиці *deleted*) і всі нові значення (з таблиці inserted). Ось тут нам знадобляться оператори групи *Cursor Control Language (CCL)*, зокрема *FETCH.*

```
CREATE TRIGGER Тригер_upd
```

```
ON Операція FOR UPDATE
AS
DECLARE @x INT @x_old INT @y INT @y_old INT
курсор з новими значеннями
DECLARE CUR1 CURSOR FOR
   SELECT КодТовара,Кількість
   FROM inserted
 курсор із старими значеннями
DECLARE CUR2 CURSOR FOR
   SELECT КодТовара,Кількість
   FROM deleted
OPEN CUR1
OPEN CUR2
переміщаємося паралельно по обох курсорах
   FETCH NEXT FROM CUR1 INTO @x, @y
   FETCH NEXT FROM CUR2 INTO @x_old, @y_old
   WHILE @@FETCH_STATUS=0
    BEGIN
для старого коду товару зменшується його кількість на складі
     UPDATE Склад
     SET Залишок=Залишок-@y_old
     WHERE КодТовару=@x_old
для нового коду товару, якщо такого товару ще немає на складі, вводиться новий запис
     IF NOT EXISTS (SELECT * FROM Склад
       WHERE КодТовару=@x)
     INSERT INTO Склад(КодТовара,Залишок)
       VALUES (@x,@y)
     ELSE
інакше для нового коду товару збільшується його кількість на складі
       UPDATE Склад
       SET Залишок=Залишок+@y
       WHERE КодТовару=@x
     FETCH NEXT FROM CUR1 INTO @x, @y
   FETCH NEXT FROM CUR2 INTO @x_old, @y_old
END
CLOSE CUR1
```

```
CLOSE CUR2
DEALLOCATE CUR1
DEALLOCATE CUR2
```

У розглянутому тригері відсутнє порівняння кількості товару при зміні запису про операцію з його залишком на складі.

Приклад 6.15. Виправлення помилки. Для генерування повідомлення про помилку використовуємо в тілі тригера команду MS SQL Server RAISERROR, аргументами якої є текст повідомлення, рівень серйозності і статус помилки.

```
ALTER TRIGGER Тригер_upd
ON Операція FOR UPDATE
AS
DECLARE @x INT @x_old INT @y INT
      @y_old INT,@o INT
DECLARE CUR1 CURSOR FOR
   SELECT КодТовару, Кількість
   FROM inserted
DECLARE CUR2 CURSOR FOR
   SELECT КодТовару, Кількість
   FROM deleted
OPEN CUR1
OPEN CUR2
   FETCH NEXT FROM CUR1 INTO @x, @y
   FETCH NEXT FROM CUR2 INTO @x_old, @y_old
   WHILE @@FETCH_STATUS=0
    BEGIN
      SELECT @o=залишок
      FROM Склад
      WHERE кодтовара=@x
      IF @o<-@y
      BEGIN
        RAISERROR('откат',16,10)
        CLOSE CUR1
        CLOSE CUR2
        DEALLOCATE CUR1
        DEALLOCATE CUR22
        ROLLBACK TRAN
```

```
        RETURN
      END
        UPDATE Склад
      SET Залишок=Залишок-@y_old
      WHERE КодТовара=@x_old
      IF NOT EXISTS (SELECT * FROM Склад
        WHERE КодТовара=@x)
      INSERT INTO Склад(КодТовара,Залишок)
        VALUES (@x,@y)
    ELSE
      UPDATE Склад
      SET Залишок=Залишок+@y
      WHERE КодТовара=@x
   FETCH NEXT FROM CUR1 INTO @x, @y
   FETCH NEXT FROM CUR2 INTO @x_old, @y_old
END
CLOSE CUR1
CLOSE CUR2
DEALLOCATE CUR1
DEALLOCATE CUR2
```

6.7. Технології використання SQL

Розглянута нами вище мова реляційних баз даних являє собою надзвичайно потужний інструмент для створення, підтримки та використання реляційних баз даних. Разом з тим, для того щоб справді бути таким інструментом, мова SQL потребує програмної реалізації, бо сама по собі є лише засобом абстрагування для організації електронної обробки даних.

Стандарти мови SQL, що регламентують синтаксис операторів, очевидно відносять SQL до парадигми декларативних мов, тобто відсутня можливість оголошення змінних, немає інструкції IF, немає циклу FOR і т.д. Одним словом, у такому вигляді мова призначається винятково для інтерактивного режиму роботи з базою даних: користувач вводить запит – отримує результат, вводить інший запит – отримує інший результат і т.д. У стандарті SQL відсутні будь-які вказівки на організацію вводу/виводу. Програмний продукт на подібній мові реалізувати вкрай важко. Тому SQL з самого початку потребував підтримки

засобами, які не передбачалися можливостями самої мови. За десятиліття його застосування накопичилася різномантні варіанти реалізації SQL. Наразі викристалізувались наступні різновиди реалізації мови SQL:

- інтерактивний SQL;
- програмний SQL:

  -програмно – вбудований (статичний SQL, динамічний SQL);

  - API – інтерфейси виклику підпрограм.

Розберемо ці різновиди докладніше. На рис. зображені схеми використання SQL. Дві з них ми розлянемо в цьому розділі. Схему використання SQL з API – інтерфейсом розглянемо у наступному розділі.

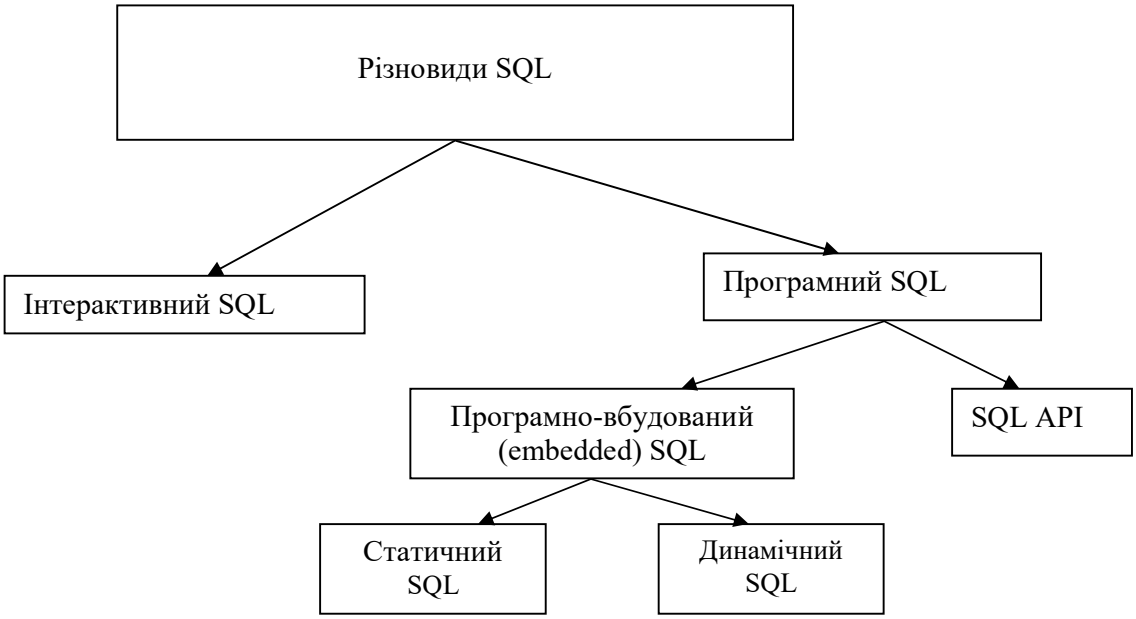


Рис. 6.1. Різновиди SQL

6.7.1. Поняття інтерактивного SQL

Отже, інтерактивний SQL передбачає безпосередню роботу користувача із базою даних за наступним алгоритмом: використовуючи прикладну програму (клієнтське застосування) або стандартну утиліту, що входить в СУБД, користувач:

- встановлює з'єднання з БД (підтверджуючи наявність прав доступу);
- вводить текст SQL-запиту;

- запускає запит на виконання. Текст запиту надходить в СУБД, яка здійснює синтаксичний аналіз запиту (перевіряє, чи є запит коректним); перевіряє, чи має користувач право виконувати подібний запит (наприклад, користувач намагається щось видалити, а права має лише на читання);
- вибирає, яким чином здійснювати виконання запиту – план виконання запиту;
- виконує запит;
- результат виконання посилає користувачеві.

В різних СУБД форми і механізми реалізації інтерактивного SQL дещо відрізняються, проте перелічені принципи не змінюються. Для ілюстрації принципів інтерактивного SQL надамо наступні приклади, причому один з них на прикладі середовища потужної комерційної клієнт-серверної *СУБД Sybase SQL Anywhere***,** а інший з вільно поширюваної (freeware) *СУБД MYSQL :*

6.7.2. Утиліта інтерактивного доступу ISQL СУБД Sybase SQL Anywhere

Утиліта *ISQL* є складовою частиною СУБД Sybase SQL Anywhere. Ця СУБД спеціально розроблена для реалізації технології клієнт-сервер. У цій технології СУБД не призначена для реалізації функцій інтерфейсної частини застосувань. В зв'язку з цим дана утиліта не дозволяє створювати екранні форми, звіти, меню і т.д. Вона дозволяє тільки виконувати SQL-операторів, але і це теж немало. Вона дозволяє виконувати в інтерактивному режимі SQL-оператори, у тому числі і ті, які змінюють схему бази даних. Дана утиліта призначена для оперативного аналізу вмісту бази даних і для відладки програмних об'єктів баз даних. Ця утиліта є файлом *Isql.exe*, який знаходиться в одному з каталогів , вкладених в каталог *SQLANY50*.

На початку роботи даної утиліти, видається запит на введення параметрів з'єднання Залежно від того відкрита необхідна база чи ні, запит представляється у вигляді скороченого (або повного вікна параметрів з'єднання (Рис.).

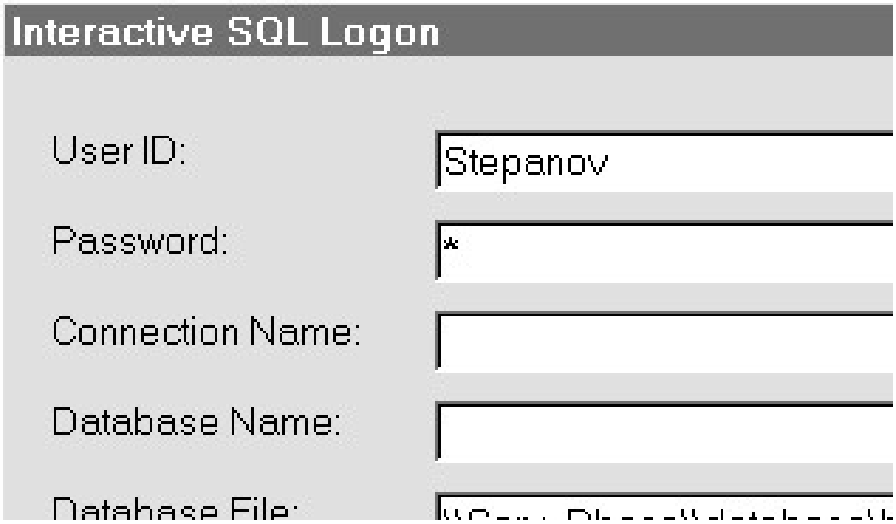


Рис. 6.2. *Повне вікно параметрів з'єднання утиліти ISQL*

За умовчанням як значення параметра *DatabaseName* використовується ім'я файлу, задане в параметрі *DatabaseFile.* Після встановлення з'єднання на екрані монітора з'являється головне вікно ISQL (Рис. 6.2.).

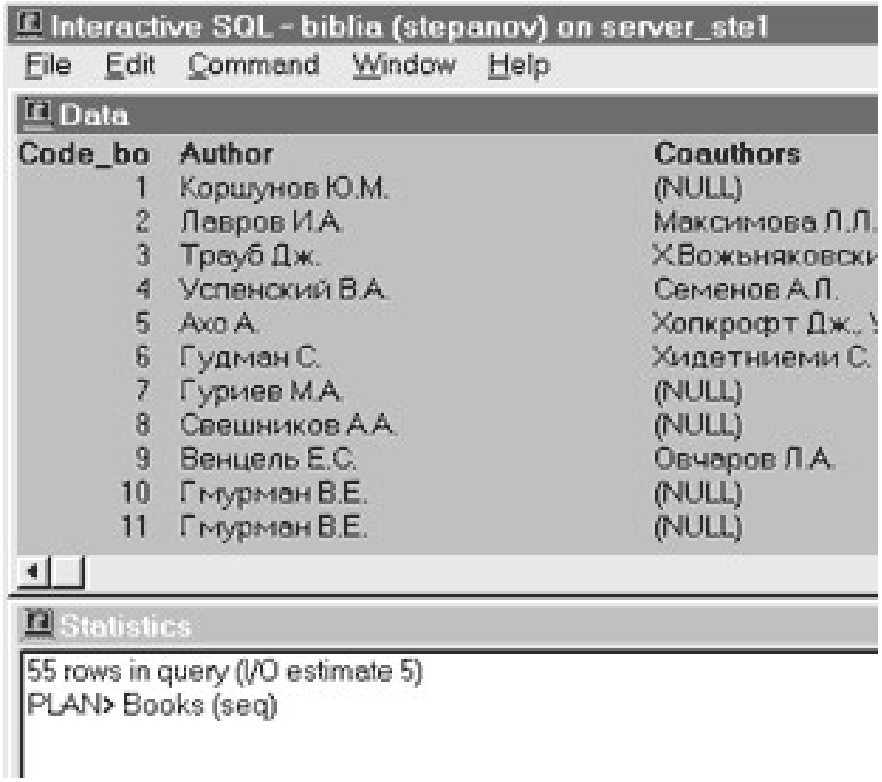


Рис. 6.3. *Головне вікно утиліти ISQL*

Заголовок головного вікна утиліти ISQL є складеним. Він включає найменування утиліти, назву підключеної бази даних, ім'я користувача, що встановив з'єднання (у дужках), і назву сервера, адміністратора підключеної бази даних. Так на рис. видно, що користувач *Stepanov* в середовищі утиліти *ISQL* встановив з'єднання з базою, якою управляє сервер *Server_ste1*.

Головне вікно даної утиліти містить три вкладені вікна:

- вікно *Data* - вікно даних, що формуються в результаті виконання запиту;
- вікно *Statistics* - вікно характеристик відпрацьованого запиту і плану його виконання);
- вікно *Command* - вікно виконуваних SQL-операторів.

На початку функціонування ISQL всі вони порожні. Інформація в них з'являється в процесі роботи з утилітою. При цьому вікно ***Command*** використовується для введення виконуваних SQL-операторів. На Рис. 6.3. це вікно містить оператора вибору всіх полів з таблиці **Stepanov.Books**. У назві таблиці фраза Stepanov позначає ім'я власника таблиці, тобто користувача того, що створив її. Фраза Books позначає безпосереднє ім'я таблиці.

Ще одним прикладом інтерактивного використання SQL-операторів є вікно Query Analyzer у середовищі MS SQL Server.

6.7.3. Приклад використання інтерактивного SQL в СУБД MYSQL

СУБД MySQL користується широкою популярністю в програмістському середовищі, хоча не набула статусу гранду серед СУБД, проте широко розповсюджена й працює під різними платформами. Зокрема, даний приклад ілюструє роботу інтерактивного SQL під операційною системою Linux Debian.

Для початку роботи в командному рядку введемо:

...$ mysql

Командна оболонка шелла видасть запрошення на зразок:

*welcome to the mysql monitor. commands **end** with ; or g. your mysql*

connection id is 46 to server version: 3.22.32-log

*type 'help' **for** help.*

mysql>

то у відповідь на нього наберіть q, залишимо на якийсь час інтерпретатор sql запитів, і займемося адмініструванням сервера mysql. Перш за все необхідно встановити пароль адміністратора сервера БД. Тепер потрібно завести користувачів і дати їм деякі права. Все адміністрування проводиться через звичайні таблиці mysql, а їхня правка також здійснюється стандартними sql командами. Найперша таблиця, яка визначає допуск користувача до сервера, так і називається - user. Щоб з’ясувати хто у нас там є, й що він може робити, наберемо:

...$ mysqldump -u root -p -opt mysql user>mysql-users.sql

Після виконання цієї команди у нас з'явився файл mysql-users.sql. Завантажимо його в текстовий редактор, щоб детальніше вивчити, і, можливо, трохи поправити.

 # mysql dump 1.1

#

```
# host: localhost database: mysql
#--------------------------------------------------------
# server version 3.22.32-log
#
# table structure for table 'user'
#
drop table if exists user;create table user (host char(60) default '' not null,user char(16) default '' not null,password char(16) default '' not null
select_priv enum('n','y') default 'n' not null
insert_priv enum('n','y') default 'n' not null
update_priv enum('n','y') default 'n' not null
delete_priv enum('n','y') default 'n' not null
create_priv enum('n','y') default 'n' not null
drop_priv enum('n','y') default 'n' not null
reload_priv enum('n','y') default 'n' not null
shutdown_priv enum('n','y') default 'n' not null
process_priv enum('n','y') default 'n' not null
file_priv enum('n','y') default 'n' not null
grant_priv enum('n','y') default 'n' not null
references_priv enum('n','y') default 'n' not null
index_priv enum('n','y') default 'n' not null
alter_priv enum('n','y') default 'n' not null
primary key (host,user)
);
#
# dumping data for table 'user'
#
lock tables user write;
insert into user values
('localhost','root','','y','y','y','y','y','y','y','y','y','y','y','y','y','y','y','y'),
('localhost','ophil','','n','n','n','n','n','y','n','y','n','n','n','y','n','n','n','n'),
('localhost','proba','','n','n','n','n','n','n','n','n','n','n','n','n','n','n','n','n');
unlock tables;
```

Це перші sql-речення, хоча ми поки не починали робити якісь запити. У реченні *create table* перераховані всі 14 різних привілеїв, які можуть мати, або бути позбавлені користувачі. Перші 6 - *select, insert, update, delete, create і drop*, стосуються права

користувача працювати із записами таблиць і із самими таблицями. Наступні 4 - *reload, shutdown, process i file* - стосуються сервера в цілому. Привілеї *grant, references, index i alter* дають можливість передавати права, а також змінювати, зв'язувати й індексувати таблиці. Два важливі зауваження.

По-перше, дані в цій таблиці, за умовчанням розповсюджуються на всі БД, що є на сервері. Тому не варто надавати цій таблиці жодних привілеїв, що стосуються таблиць. Краще це робити з точністю до окремих полів і адрес хостів та прав користувачів в інших таблицях, а ця таблиця повинна тільки дозволяти вхід на сервер.

По-друге, привілеї, які стосуються сервера в цілому, налаштовуються тільки в цій таблиці, і хоча б один користувач, у даному випадку *root***,** повинен мати ці привілеї, інакше сервер стане некерованим.

Якщо у ваші плани входить дати доступ до сервера всім користувачам і під будь-яким ім'ям, заведіть користувача з порожнім ім'ям "". Ті ж правила застосовуються до імен і адрес комп'ютерів-хостів. Тепер, поправивши записи в таблиці, але в жодному випадку не її структуру, відправимо команди назад в mysql.

...$ mysql -u root -p < mysql-users.sql

і попросимо сервер перечитати змінені права

...$ mysqladmin -u root -p reload

Ще одне зауваження щодо паролів. У розглянутому нами файлі поля паролів порожні, і це треба негайно виправити. Правити їх у текстовому файлі незручно, тому що *mysql* використовуваний для шифрування пароля окрему програму, і зберігає пароль у зашифрованому вигляді. Щоб установити пароль, наприклад, користувачеві *«user2»*, треба виконати таку команду:

...$ mysql mysql -i 'update user set password=password(«0») where user=«user2»;'

Відмінність від звичайної системи паролів у тому, що імена користувачів БД можуть бути не пов'язані з їхніми реєстраційними іменами в системі, користувачі не можуть міняти свої паролі, і цей пароль відомий адміністраторові БД.

Розібравшись із найпершою адміністративною таблицею *user,* решта таблиць: *db, host, tables_priv, columns_priv, func* - змінюємо аналогічно.

Кожну з команд, що посилається *mysql*, можна задавати або в моніторі запитів *mysql*, або з командного рядка, або створивши файл і відправивши його до інтерпретатора *mysql* через той же монітор. При роботі в інтерпретаторі завжди повідомляється час, витрачений на виконання запиту, а при виведенні у файл (канал) не малюється рамочка навколо таблиці. Це робиться тільки для нашої зручності, і не впливає ні на результат, ні на сам *sql* запит.

Далі вже можна робити запити,якщо, звичайно, початкова база готова. Нехай у нас є телефонна база даних. Тоді до неї можна робити інтерактивні запити, наприклад :

select p.phonum, p.naim, s.nick, b.bldng

from phone p, street s, building b # короткі синоніми таблиць

where

p.bd_id=b.bd_id # таким чином

and b.st_id=s.st_id # зв'язують таблиці

and p.phonum like «%1234%» # власне запит

order by p.naim;

Зауважимо, що в секторі клієнт-серверних застосувань Mysql цілком здатний конкурувати з визнаними комерційними СУБД, навіть із такими як Sybase. Але вся потужність MySql розкривається в поєднанні з технологіями Internet.

6.7.4. Програмно-вбудований(embedded) SQL

Інтерактивна форма роботи з SQL адекватна для адміністратора баз даних, можлива для певних категорій користувачів, проте зовсім неприйнятна для всіх програмістів і переважної більшості користувачів БД. Програмно-вбудований SQL (Embedded SQL) призначений для того, щоб вбудовувати SQL запити в прикладну програму. Але як це зробити?

Як «пояснити» компілятору C++, COBOL чи PL/1, що частина програми це не оператори мови, а оператори якогось іншого коду? Ця проблема дещо нагадує задачу реалізації мережевої моделі по стандарту CODASYL. Там теж застосовується мови високого рівня. Принципова різниця полягає в тому, що на відміну від схем та підсхем стандарту CODASYL, які по-суті є підмножинами процедурних мов, SQL є декларативною. Як з'єднати можливості мови програмування структурного рівня (змінні, розгалуження, цикли) і використати можливості SQL (запити декларативного типу, де вже не на програміста, а на СУБД лягає обов'язок по їхній оптимізації й вибору плану виконання)?

Ці й деякі інші проблеми вирішуються декількома способами. Ці способи частково описані й у стандарті SQL-92. На даний момент використовуються два варіанти програмного SQL: статичний та динамічний. Розглянемо, що являє собою кожний варіант.

Статичний SQL є різновид програмного SQL, призначеного для вбудовування SQL-операторів у текст програми мовою програмування високого рівня. Розглянемо ще раз алгоритм виконання SQL-запитів в інтерактивному режимі роботи. Легко бачити, що користувач змушений очікувати результатів виконання запиту протягом усього часу роботи алгоритму. Якщо через якийсь час користувачеві знову потрібно буде виконати той же самий

запит, СУБД знову проробить ті ж самі дії, що й при попередньому зверненні. У наявності деяка недосконаліть механізму: ті самі етапи виконуються щораз заново для однакових запитів; СУБД не може обробляти інтерактивні запити з випередженням. Рішення подібних проблем очевидне - частина дій по обробці запиту необхідно виконувати один раз, зберігати результат у якомусь вигляді, а потім використати стільки разів, скільки необхідно. Ця ідея є однією з основних ідей програмного SQL. Отже, програмний, а зокрема, і статичний SQL дозволяє:

- використати оператори інтерактивного SQL у тексті програми мовою програмування високого рівня;
- поряд з операторами інтерактивного SQL використати нові спеціальні конструкції, що доповнюють SQL, і збільшують його можливості;
- для передачі параметрів у запит використати в тексті запиту змінні, оголошені в програмі;
- для повернення в програму результатів запиту використати спеціальні конструкції, відсутні в інтерактивному SQL;
- здійснювати компіляцію запитів разом із програмою, забезпечуючи, як наслідок погоджену роботу програми й СУБД.
- Заздалегідь (на етапі компіляції) виконувати дії по аналізу й оптимізації запитів, заощаджуючи час, що витрачаєтья на етапі виконання програми.

Розглянемо два основних етапи, пов'язаних із роботою статичного SQL - компіляцію та виконання програми. Схема компіляції й збирання програми виглядає в такий спосіб: програма, що включає оператори мови програмування високого рівня разом з операторами SQL, подається на вхід спеціального препроцесора, що виділяє з її частини, пов'язані з SQL. Замість інструкцій вбудованого SQL препроцесор підставляє виклики спеціальних функцій СУБД. Бібліотеки таких функцій для зв'язку з мовами програмування існують для всіх розповсюджених серверних СУБД. Варто особливо відзначити, що ці бібліотеки мають «закритий» інтерфейс, тобто розробники можуть міняти його за своїм розсудом, відповідно обновивши препроцесор. Усе це говорить про те, що програміст не повинен втручатися в цей процес. Самі інструкції SQL препроцесор виділяє в окремий файл. Програма надходить на вхід звичайного компілятора мови програмування, після чого виходять об'єктні модулі. Далі ці об'єктні модулі разом із бібліотеками СУБД збираються в один виконує модуль, що, - додаток. Поряд із цими операціями відбувається робота з файлом, що містить SQL-інструкції. У літературі цей модуль часто зветься «модуль запитів до бази даних» (Database Request Module, DBRM) . Обробку цього модуля здійснює спеціальна утиліта, що зазвичай називається BIND. Для кожної інструкції SQL утиліта виконує наступні дії:

- здійснює синтаксичний аналіз запиту (перевіряє, чи є запит коректним);
- перевіряє, чи існують у базі даних ті об'єкти, на які посилається запит;

- вибирає, яким чином здійснювати виконання запиту - план виконання запиту.

Усі плани виконання запитів зберігаються в СУБД для наступного використання.

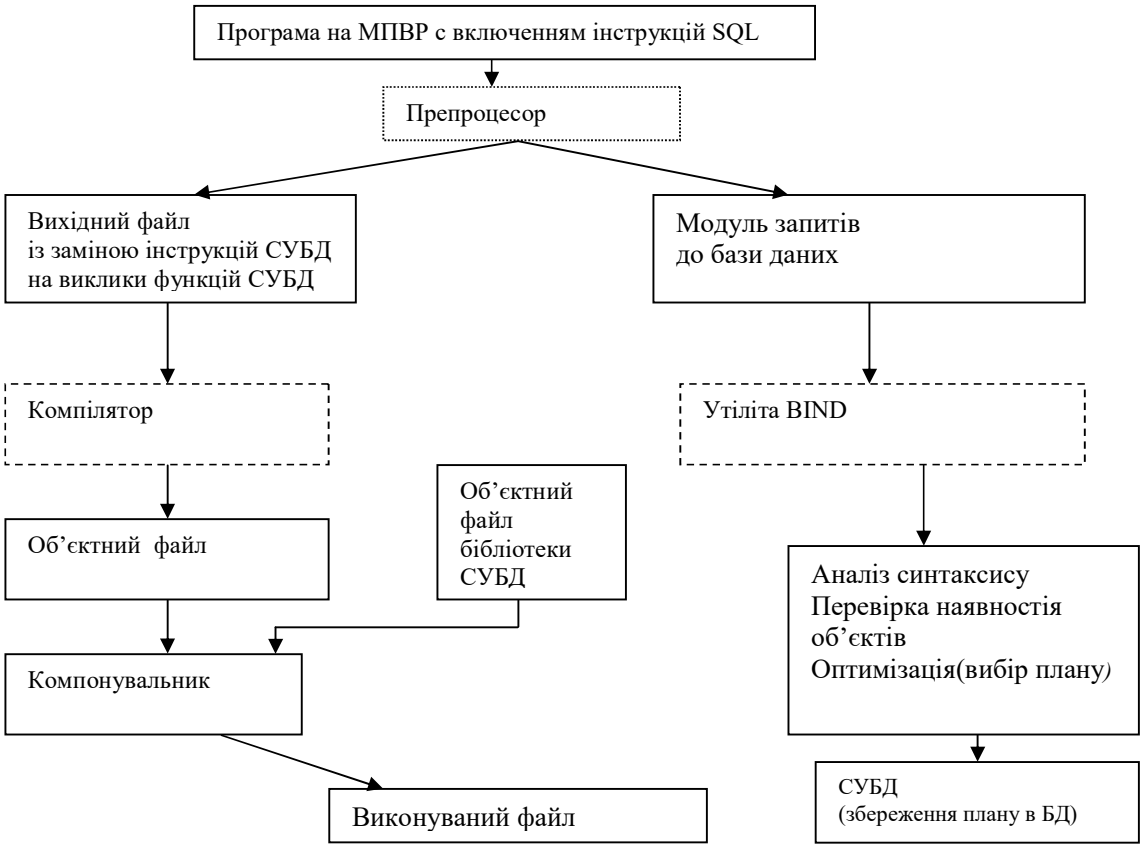


Рис. 6.4. Схема компіляції програми зі вбудованими інструкціями статичного SQL

Схема виконання програми виглядає в такий спосіб:

Програма запускається на виконання звичайним чином.

При необхідності виконати запит програмою здійснюється виклик спеціальної функції СУБД, що відшукує вже сформований раніше план виконання запиту.

СУБД виконує запит відповідно до обраного плану. Результат виконання запиту надходить у додаток. Основними **командами** статичного SQL є:

EXEC SQL	специфікатор, що вказує, що наступна за ним інструкція є інструкцією вбудованого *SQL*
;	у мові С – ознака закінчення інструкції вбудованого *SQL*
DECLARE TABLE	повідомляє таблицю, що потім буде використовуватися в інструкціях вбудованого *SQL*

SQLCODE	змінна для обробки помилок
SQLSTATE	змінна для обробки помилок
GET DIAGNOSTICS	інструкція для обробки помилок
WHENEVER SQLWARNING GOTO SQLERROR NOT FOUND CONTINUE	інструкцій для спрощення набір спільно використовувані обробки помилок
BEGIN DECLARE SECTION END DECLARE SECTION	інструкції для визначення області, у якій будуть оголошені змінні, у наслідку використовувані в запитах *SQL*
INTO	використається в операторі *SELECT* для вказівки змінної, у яку необхідно помістити результат виконання запиту

можуть використатися тільки в тих місцях, де в запитах зазвичай стоять константи. Наприклад, неможливо параметризувати ім'я таблиці, з якої проводиться вибірка, а також назви колонок. У результаті ми приходимо до наступного висновку. При використанні статичного варіанта вбудованого (програмного) SQL необхідно на етапі написання програми точно знати склад запитів, які необхідно буде виконувати в програмі. У багатьох випадках це обмеження є істотним. Для його усунення реалізовано новий різновид програмного SQL – динамічний SQL. Розглянемо коротко основні ідеї динамічного SQL.

6.7.5. Динамічний SQL

Ще один різновид програмного SQL, призначена для вбудовування SQL-операторів у текст програми мовою програмування високого рівня, який допускає динамічне формування й виконання запитів під час роботи програми. Історія виникнення динамічного SQL багато в чому пов'язана з компанією IBM, що впровадила цей потужний інструмент у свою СУБД DB2. Стандарт SQL-89 не підтримував динамічний SQL. Лише в 1992 році в стандарт SQL-2 були включені специфікації динамічного SQL. Для того щоб використати цей інструментарій у своїх програмах, необхідно представляти основні концепції, основні ідеї, закладені в динамічний SQL і принципи його функціонування. Саме це і є предметом розгляду в даному розділі. Теоретики вважають основною концепцією динамічного SQL таке твердження: вбудована інструкція SQL не записується у вихідний текст програми. Замість цього програма формує текст інструкції під час виконання в одній зі своїх областей даних, а потім передає сформовану інструкцію в СУБД для динамічного виконання [].

Згадаємо, як працювала статична SQL. Схема використання передбачала 2 етапи – компіляція програми й виконання програми. На етап компіляції лягало основне

навантаження в тому розумінні, що він вирішував питання перевірки, розбору й оптимізації запитів, оскільки було відомо, що саме розбирати й оптимізувати. Цілком очевидно, що подібну двоетапну схему не можна реалізувати для динамічного SQL, адже на етапі компіляції програми ще невідомо, що саме потрібно розбирати й оптимізувати. Які наслідки цієї зміни? Будь-який механізм має свої переваги й недоліки. Переваги динамічного SQL у тому, що він дозволяє формувати запит до бази даних під час роботи програми, реагуючи на ті або інші події, що відбулися. Така можливість є життєво важливою для клієнт-серверної й триланкової архітектури, у яких структура бази даних і ділові правила мають тенденцію до зміни, що вимагає певної гнучкості при організації процесу обробки даних. Недолік динамічного SQL – низька продуктивність у порівнянні зі статичним. Дійсно, багато з операцій, що виконувалися на етапі компіляції один раз, тепер виконуються безпосередньо під час роботи програми. У зв'язку із цим, вважається правильним рекомендувати використання статичного різновиду SQL, там, де тільки можливо. У свою чергу апарат динамічного SQL треба застосовувати тільки там, де без цього обійтись неможливо. Розглянемо схему функціонування динамічного SQL. Схема передбачає одноетапне й двоетапне виконання інструкцій. Одноетапне виконання інструкцій здійснюється командою *EXECUTE* в такій послідовності:

1. *IMMEDIATE*. Модуль, що виконує, (запуск на виконання).
2. Виконання звичайних інструкцій *МПВУ*.
3. Виклик функцій СУБД для виконання інструкцій вбудованого SQL. Аналіз синтаксису Перевірка наявності об'єктів. Оптимізація (вибір плану)
4. Виконання плану.
5. Команда *EXECUTE IMMEDIATE*.

Схема виконання інструкції передбачає: двоетапне динамічне формування команди SQL у рядковому вигляді під час роботи програми; двоетапна передача рядкового вигляду інструкції в СУБД за допомогою команди EXECUTE IMMEDIATE; двоетапне виконання інструкції системою керування БД, що включає синтаксичний аналіз, перевірку параметрів, оптимізацію (вибір плану) і виконання цього плану. Основні проблеми одноетапної схеми полягають у тому, що вона не дозволяє виконувати інструкції SELECT (тому що немає засобів для повернення в додаток результатів запиту) і приводить до нераціональних витрат обчислювальних ресурсів (тому що при повторному виконанні тієї ж інструкції знову буде витрачений час на всі ті ж дії по її інтерпретації й виконанню).

Двоетапне виконання інструкцій ґрунтується на наступному міркуванні: швидше за все, команда динамічного SQL у такому виді, як вона надходить на виконання, буде виконуватися неодноразово. При цьому можуть мінятися якісь конкретні деталі. А це значить, що інструкцію можна параметризувати. Використання параметризованих інструкцій

дозволяє зробити схему виконання двоетапним, розділивши процес на «підготовку інструкції» і «виконання інструкції».

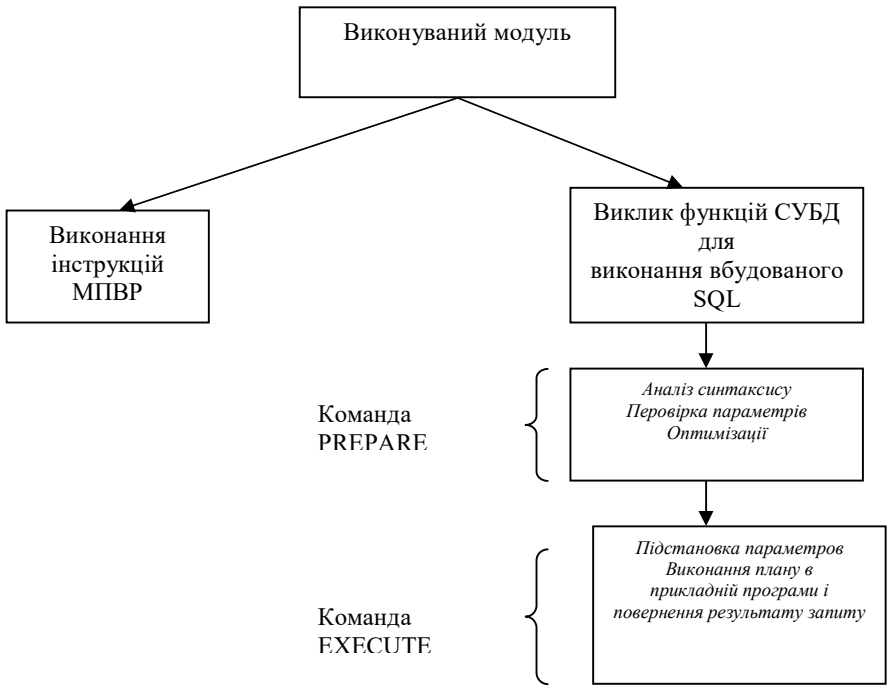


Рис. 6.5. Схема виконання програми із вбудованими інструкціями динамічного SQL із застосуванням двоетапної схеми

На етапі підготовки можна здійснити синтаксичний аналіз інструкції, інтерпретувати її та підготуватися до виконання, вибравши план виконання. На етапі виконання СУБД підставляє значення параметрів (отримані із програми) і використає сформований раніше план виконання для досягнення результату. При цьому реалізується ідея одноразового виконання тих дій, які можна виконати один раз. Так, підготовлена один раз інструкція може бути виконана десятки разів із різними параметрами. Основними **командами** динамічного SQL є:

EXECUTE IMMEDIATE Негайне виконання інструкції

PREPARE Підготовка інструкції до виконання

EXECUTE Виконання підготовленої раніше інструкції

DESCRIBE Спеціальна команда, що бере участь при поверненні результату виконання інструкцій динамічного SQL.

DECLARE CURSOR Різновид інструкції DECLARE CURSOR, що застосовувалася раніше в рамках статичного SQL, що містить замість запиту його ім'я (пов'язане із запитом за допомогою інструкції PREPARE).

OPEN FETCH CLOSE Різновиду інструкцій для роботи з курсором у динамічному SQL.

Як ми бачимо основу цих операторів складають згадувані в розділі оператори , які в мові SQL називаються операторами управлінням курсора(Cursor Control Language).

6.7.6. Реалізація статичного SQL на мові C++, що використовує OCCI-бібліотеку

Приведемо приклад застосування на мові C++, що використовує OCCI-бібліотеку для реалізації операцій вставки, видобування даних, зміни і видалення даних.

Приклад:

```
#include <iostream.h>
#include <occi.h>     // Бібліотека розташовується
                      // у каталозі [ORACLE_HOME]\oci\include
using namespace oracle::occi;     // Для доступу до простору імен occi
using namespace std;
class occi_select
{
 private:
 Environment *env;          // Змінна оточення
 Connection *conn;          // Змінна з'єднання
 Statement *stmt;           // Змінна оператора
 public:
 occi_select (string user, string passwd, string db)        // Конструктор з'єднання з БД
 {
  env = Environment::createEnvironment (Environment::DEFAULT);
  conn = env->createConnection (user, passwd, db);
 }
 ~occi_select ()            // Деструктор
 {
  env->terminateConnection (conn);        // Звільнення з'єднання
Environment::terminateEnvironment (env);
 }
 // Додавання рядка з динамічним зв'язуванням
 void insertRowBind(int c1, string c2)
 {
  string sqlStmt = «INSERT INTO tbl1 VALUES (:x :y)»;
  stmt=conn->createStatement (sqlStmt);
  try{
```

```
  stmt->setInt (1, c1);                // Визначення значення для першої колонки
  stmt->setString (2, c2);
  stmt->executeUpdate ();
  }catch(SQLException ex)
  {  cout<<ex.getMessage() << endl;  }
  conn->terminateStatement (stmt);      // Звільнення оператора
 }
 // Додавання рядка в таблицю
 void insertRow ()
 {
  string sqlStmt = «INSERT INTO tbl1 VALUES (6, 'ABC')»;
  stmt = conn->createStatement (sqlStmt);
  try{
  stmt->executeUpdate ();
  }catch(SQLException ex)
  {  cout<<ex.getMessage() << endl;  }
  conn->terminateStatement (stmt);
 }
// Зміна рядка
 void updateRow (int c1, string c2)
 {
  string sqlStmt =
    «UPDATE tbl1 SET f1 := x WHERE f2: = y»;
  stmt = conn->createStatement (sqlStmt);
  try{
  stmt->setString (1, c2);
  stmt->setInt (2, c1);
  stmt->executeUpdate ();
  }catch(SQLException ex)
  {  cout<<ex.getMessage() << endl;  }
  conn->terminateStatement (stmt);
 }
// Видалення рядка
 void deleteRow (int c1, string c2)
 {
  string sqlStmt =
```

```
    «DELETE FROM tbl1 WHERE f1:= x AND f2 := y»;
   stmt = conn->createStatement (sqlStmt);
   try{
   stmt->setInt (1, c1);
   stmt->setString (2, c2);
   stmt->executeUpdate ();
   }catch(SQLException ex)
   {  cout<<ex.getMessage() << endl;  }
   conn->terminateStatement (stmt);
  }
// Відображення всіх рядків
  void displayResultSet ()
  {
   string sqlStmt = «SELECT f1, f2 FROM tbl1»;
   stmt = conn->createStatement (sqlStmt);
   ResultSet *rs = stmt->executeQuery ();
  try{
  while (rs->next ())
  {
     cout << «f1: « << rs->getInt (1) << « f2: «
       << rs->getString (2) << endl;
   }
   }catch(SQLException ex)
   {
    cout<<«Error number: «<< ex.getErrorCode() << endl;
    cout<<ex.getMessage() << endl;
   }
   stmt->closeResultSet (rs);              // Звільнення результуючого набору
   conn->terminateStatement (stmt);
  }
};              // Кінець коду класу occi_select
int main (void)
{
  string user = «SCOTT»;
  string passwd = «TIGER»;
  string db = ««;
```

```
occi_select *occi1 = new occi_select (user, passwd, db);
occi1-> displayResultSet ();          // Відображення всіх записів
occi1->insertRow ();                  // Вставка рядка
occi1-> displayResultSet ();
occi ->insertRowBind(6, «ABC»);       // Вставка рядка з використанням
                                      // динамічного зв'язування  occi1-> displayResultSet ();
occi ->deleteRow (6, «ABC»);          // Видалення рядка
occi1->updateRow (4, «EEE»);          // Оновлення рядка
occi1-> displayResultSet ();
delete (occi1);                       // Звільнення OCCI-об'єкту ?
```

6.7.7. Реалізація динамічного SQL на мові C++ для СУБД ЛИНТЕР

Розглянемо реалізацію динамічного SQL на прикладі СУБД ЛИНТЕР. Ця СУБД не настільки відома, як Oracle IBM DB2, проте досить добре документована[]. Вбудований SQL дозволяє конструювати й виконувати SQL-запити, у яких текст, кількість і тип змінних невідомі в момент компіляції програми. Текст таких запитів формуються в процесі виконання програми й, залежно від умов виконання програми, щораз може мати різний вигляд. При конструюванні динамічного запиту необхідно описати число, тип і імена обираних колонок і, відповідно, указати змінні, у які обирані значення повинні бути завантажені. Ці описи виконуються за допомогою дескрипторів - спеціальних типів даних для зв'язування змінних у динамічних запитах.

Позаяк в динамічному запиті можуть бути присутні дві групи невідомих параметрів: вхідні й вихідні (обрані колонки й вихідні параметри збережених процедур), для динамічного запиту потрібні дві змінні, у яких ці параметри будуть зберігатися. Ці змінні називаються дескрипторами й мають однакову структуру для вхідних параметрів (BIND-дескриптор) і вихідних параметрів (SELECT-дескриптор).

Дескриптор може бути оголошений такими способами:

1.Як ідентифікатор вбудованої мови при першому використанні у вихідному тексті модуля.

У цьому випадку прекомпілятор автоматично створює змінну типу «дескриптор» (DESCRIPTOR) з даним ім'ям, наприклад,:

EXEC SQL DESCRIBE INPUT my_statement INTO SQL DESCRIPTOR ‘DSI’; /* неявне оголошення дескриптора в операторі DESCRIBE.

Перед виконанням цього оператора повинен бути викликаний оператор ALLOCATE DESCRIPTOR, тому що оператор DESCRIBE лише вводить змінну DSI у простір імен вбудованої мови */

EXEC SQL ALLOCATE DESCRIPTOR tabl_desc WITH MAX 12; /* Неявне завдання імені дескриптора при його ініціалізації */

2.Як змінна основної мови типа «дескриптор» DESCRIPTOR у секції оголошень змінних основної мови:

Лістинг програми, яка реалізує динамічний SQL -запит

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#if defined(VXWORKS)
#include «vxstart.h»
#endif
#ifdef _DOS_
#include <conio.h>
#endif
EXEC LINTER IFDEF SQL;
EXEC SQL INCLUDE SQLCA;
EXEC LINTER ENDIF;
EXEC SQL BEGIN DECLARE SECTION;
 char *user = «SYSTEM»;
 char *pswd = «MANAGER»;
 char szSQL[4096] = {0};
 char buf[65535] = {0};
 int i = 0;
 int colCount = 0;
 int type = 0;
 int length = 0;
 int shift = 0;
 char * p = 0;
 DESCRIPTOR  selectOutDesc;
EXEC SQL END DECLARE SECTION;
char *aszTypes[] = {
 »UNKNOWN»,
«CHAR»,»INT»,»SMALLINT»,»BIGINT»,»DOUBLE»,»REAL»,»DATE»,»DECIMAL»,
```

```
 »BYTE»,»BLOB»,»VARCHAR»,»VARBYTE»,»BOOL»,»NCHAR»,»NVARCHAR»,»EXTFILE»
};
int pcc2index(int pcc)
{
 switch (pcc)
 {
  case 14 : return 9;
  case 15 : return 12;
  case 31 : return 10;
  case 16 : return 13;
  case 1  : return 1;
  case 12 : return 11;
  case 3  : return 8;
  case 9  : return 7;
  case 8  : return 5;
  case 6  : return 6;
  case 7  : return 6;
  case 4  : return 2;
  case 2  : return 8;
  case 5  : return 3;
  default : return 0;
 }
}
int align4(int shift)
{
 return shift % 4 ? shift + 4 - shift % 4 : shift;
}
void printAnswer()
{
 char szMask[200];
 char szLen[20];
 int i = 0;
 float _f = 0;
 int _i = 0;
 L_WORD varcharlen = 0;
 shift = 0;
```

```
 for (i = 1; i <= colCount; i++)
 {
  EXEC SQL GET DESCRIPTOR :selectOutDesc VALUE :i :length = LENGTH;
  EXEC SQL GET DESCRIPTOR :selectOutDesc VALUE :i :type = TYPE;
  switch (type)
  {
   case 1: /*char*/
    printf(«%s «, &(buf[shift]));
   break;
   case 4: /*int*/
    memcpy(&_i, &(buf[shift]), sizeof(_i));
    printf(«%d «, _i);
   break;
   case 6: /*dbl*/
    memcpy(&_f, &(buf[shift]), sizeof(_f));
    printf(«%g «, _f);
   break;
  }
  shift += align4(length);
 }
 printf(«\n»);
}
#if defined(VXWORKS)
MainStart(pcc_sample, 1024*32, UninitLinterClient)
#else
int main()
#endif
{
 char c = 0;
 printf(«Створення з'єднання...»);
 EXEC SQL WHENEVER SQLERROR GOTO not_conn;
 EXEC SQL CONNECT AUTOCOMMIT :user IDENTIFIED BY :pswd;
 printf(«готово.\n»);
 printf(«Виділення пам'яті під дескриптор...»);
 EXEC SQL ALLOCATE DESCRIPTOR :selectOutDesc;
 printf(«готово.\n»);
```

```
printf(«Введіть SELECT-запит: >«);
gets(szSQL);
printf(«Відкриття оператора...», szSQL);
EXEC SQL WHENEVER SQLERROR GOTO not_st_opened;
EXEC SQL PREPARE ST_SEL FROM :szSQL;
printf(«готово.\n»);
printf(«Заповнення дескриптора...»);
EXEC SQL WHENEVER SQLERROR GOTO not_desc_created;
EXEC SQL DESCRIBE OUTPUT ST_SEL INTO SQL DESCRIPTOR :selectOutDesc;
printf(«готово.\n»);
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL GET DESCRIPTOR :selectOutDesc :colCount = COUNT;
printf(«Полів (колонок) у вибірці: = %d\n», colCount);
for (i = 1; i <= colCount; i++)
{
  EXEC SQL GET DESCRIPTOR :selectOutDesc VALUE :i :type = TYPE;
  EXEC SQL GET DESCRIPTOR :selectOutDesc VALUE :i :length = LENGTH;
  printf(«Колонка %i. Тип %d, %s Довжина %d\n», i, type, aszTypes[pcc2index(type)], length);
}
printf(«Прив'язка буфера...»);
shift = 0;
for (i = 1; i <= colCount; i++)
{
  p = &(buf[shift]);
  EXEC SQL GET DESCRIPTOR :selectOutDesc VALUE :i :length = LENGTH;
  EXEC SQL SET DESCRIPTOR :selectOutDesc VALUE :i DATA = :p;
  shift += align4(length);
}
printf(«готово.\n»);
printf(«Відкриття курсору...»);
EXEC SQL WHENEVER SQLERROR GOTO not_cur_opened;
EXEC SQL DECLARE CUR_SEL CURSOR FOR ST_SEL;
EXEC SQL OPEN CUR_SEL;
printf(«готово.\n»);
EXEC SQL WHENEVER SQLERROR GOTO not_fetched;
for (i = 1; ;i++)
```

```
 {
  memset(buf, 0, sizeof(buf));
  EXEC SQL FETCH CUR_SEL ABSOLUTE :i USING DESCRIPTOR :selectOutDesc;
  if (ErrPCI_)
  {
   if (ErrPCI_ == 3000) printf(«No more records\n»);
   else printf(«Error code: %d\n», ErrPCI_);
   break;
  }
  printAnswer();
 }
 printf(«Звільнення пам'яті дескриптора...»);
 EXEC SQL DEALLOCATE DESCRIPTOR :selectOutDesc;
/* EXEC SQL DEALLOCATE :CUR_SEL;*/
 printf(«готово.\n»);
 return 0;
not_conn:
 printf(«Не вдалося створити з'єднання\n»);
 printf(«Код помилки: %d\n», ErrPCI_);
 return 1;
not_selected:
 printf(«Не вдалося створити вибірку\n»);
 printf(«Код помилки: %d\n», ErrPCI_);
 return 1;
not_st_opened:
 printf(«Не вдалося відкрити оператор\n»);
 printf(«Код помилки: %d\n», ErrPCI_);
 return 1;
not_desc_created:
 printf(«Не вдалося створити дескриптор\n»);
 printf(«Код помилки: %d\n», ErrPCI_);
not_cur_opened:
 printf(«Не вдалося відкрити курсор\n»);
 printf(«Код помилки: %d\n», ErrPCI_);
 return 1;
not_fetched:
```

```
 printf(«Не вдалося переміститися по вибірці\n»);
 printf(«Код помилки: %d\n», ErrPCI_);
 return 1;
}
```

Контрольні вправи і запитання

6. Напишіть команду *CREATE TABLE* для створення таблиці *LECTURER*.
7. Напишіть команду *CREATE TABLE* для створення таблиці *SUBJECT*.
8. Напишіть команду *CREATE TABLE* для створення таблиці *UNIVERSIT*Y.
9. Напишіть команду *CREATE TABLE* для створення таблиці *EXAM_MARKS* .
10. Напишіть команду *CREATE TABLE* для створення таблиці *SUBJ_LECT*.
11. Напишіть команду, яка дозволить швидко вибрати дані про студентів по курсах, на яких вони вчаться.
12. Створіть індекс, який дозволить для кожного студента швидко здійснити пошук оцінок, згрупованих по датах.
13. Створіть таблицю EXAMMARKS так, щоб не допускалося введення в таблицю двох записів про оцінки одного студента по конкретному іспиту і предмету навчання і щоб не допускалося проведення двох екзаменів з будь-яких предметів в один день
14. Створіть таблицю предметів навчання SUBJECT так, щоб кількість годин, що відводиться на предмет, за умовчанням була рівна 36, не допускалися записи з відсутньою кількістю годин, поле SUBJ_ID було первинним ключем таблиці і значення семестрів (поле SEMESTER) лежали в діапазоні від 1 до 12.
15. Напишіть збережену процедуру, яка перевіряє чи є в даній групі студенти, що не склали сесію.
16. Напишіть збережену процедуру, яка підраховує кількість відмінників на курсі.
17. Напишіть тригер, який видаляє з бази даних відомості про студентів, які не склали сесію.
18. Які особливості інтерактивного SQL та першочергові задачі вирішуються в цій формі використання?
19. Які етапи має пройти прикладна програма, яка містить вирази SQL в статичному варіанті використання?
20. Які переваги і недоліки одноетапної та двоетапної процедур виконання інструкцій динамічного SQL?

РОЗДІЛ 7. Новітні технології електронної обробки даних для розподілених систем

І буде форма, буде навіть зміст,
шасі, таксі, готелі і мотелі...
Благословен останній альпініст,
що буде вгору дертися по скелі!
(Ліна Костенко)

Якщо притримуватись логіки попереднього розділу, то наступним етапом мав би бути розділ, який прояснював би деталі використання SQL у варіанті використання з API-функціями. Проте попередній аналіз сучасного стану електронної обробки даних та тенденцій її розвитку примушує дещо відозмінити попередню схему. Головною причиною цього є бурхливий розвиток комп'ютерної технології за останнє чверть століття. Згадаємо, що мова SQL з'явилася ще в епоху мейнфреймів і розвивалася вже в епоху персональних обчислень. В 70-80 роках двадцятого століття інтерактивний та програмно-вбудований спосіб вживання SQL здавався цілком природнім.

Схема використання СУБД виглядала наступним чином: дані, СУБД та прикладна програма розміщуються на одному комп'ютері(мейнфремі). Користувачі використовують термінали(консолі) для звернення до БД через прикладну програму, яка в свою чергу активізує СУБД. Задача СУБД проглянути БД. Потім результат звернення у зворотньому порядку повертається користувачу як потік символів і відображується на екрані монітора.

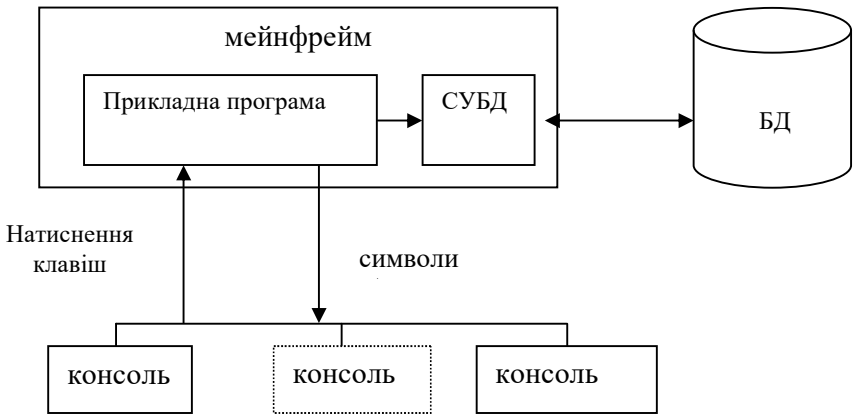


Рис.7.1. СУБД в централізованій архітектурі

Така схема роботи – це якісний стрибок технології обробки даних при колективній роботі з даними у порівнянні з пакетним режимом, проте дуже швидко виявляє свій головний недолік – різке зниження ефективності при одночасній роботі множини користувачів.

З появою персональних комп'ютерів та мереж на їх основі виникла так звана файл/серверна архітектура. Суть її в тому, що на окремій робочій станції зосереджуються спільно використовувані дані, кожен користувач знаходиться не просто за віддаленим терміналом, а за монітором персонального комп'ютера, під'єднаного в мережу. Кожний користувач має право на доступ до спільних даних , з якими потім може на свій розсуд

робити все, що вважає за потрібне. Користувач проводить запити на виконання дискових операцій вводу/виведення, а на робочу станцію користувача надсилаються блоки даних з диска, які він може обробити, використовуючи СУБД. Така архітектура має певні переваги порівняно з централізованою: швидко обробляти прості запити та формувати звіти, використовуючи ту обставину , що на робочій станції знаходиться СУБД. Проте недоліки теж очевидні: загроза цілісності спільно використовуваним даним та перевантажений трафік по мережі із-за постійної необхідності переміщувати екземпляри спільної СУБД.

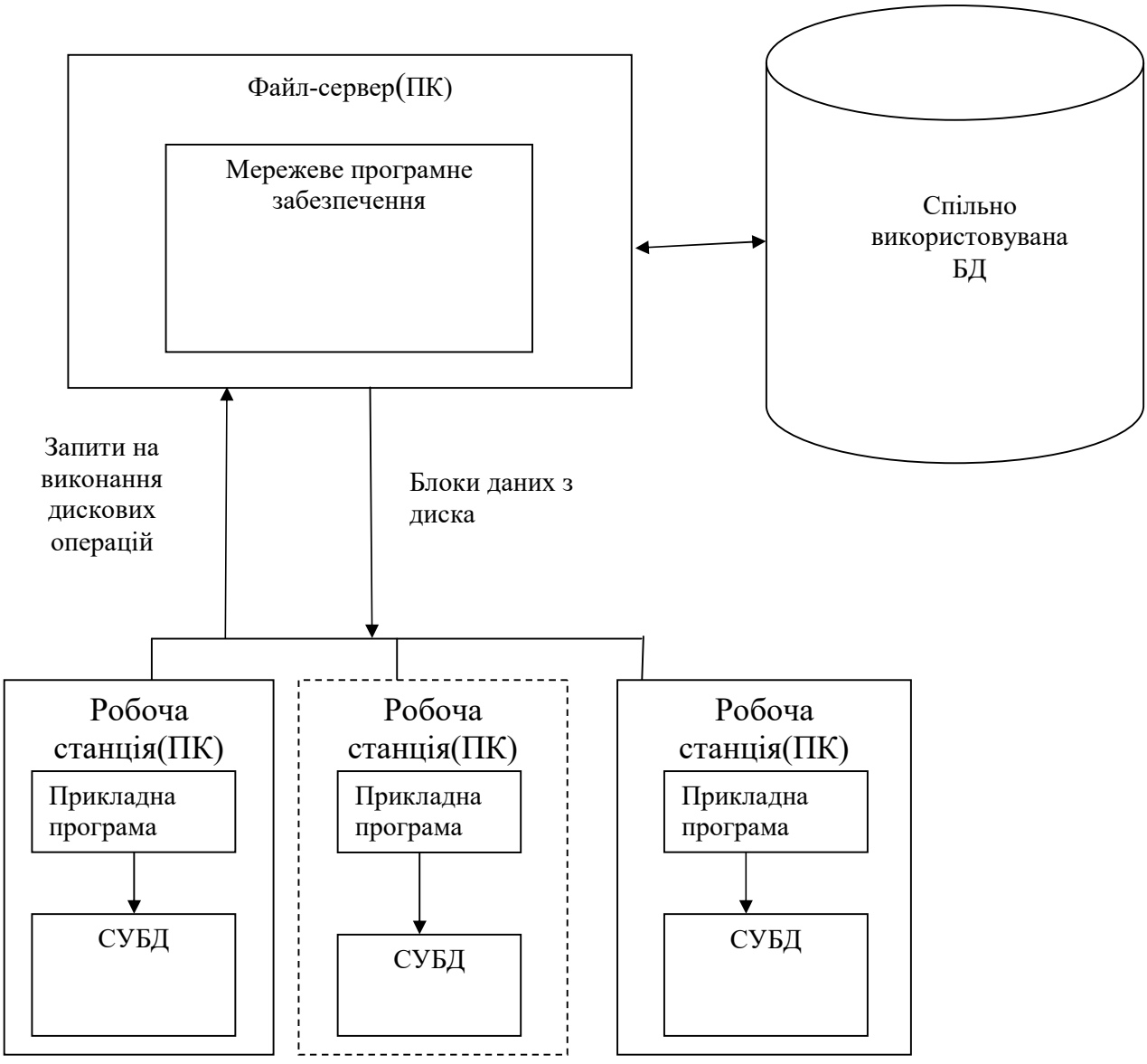


Рис.7.2. Файл-серверна архітектура використання СУБД

Альтернатива цим підходам виникла на початку 90-х років минулого століття під впливом нових можливостей, які надавалася операційними системами(Windows та Unix), створеними для забезпечення роботи мережевих обчислень в різноманітних конфігураціях. В дещо спрощеному варінті узагальнена схема взаємодії користувача бази даних з СУБД виглядає як зображено на Рис. Такий принцип отримав назву клієнт-серверної організації СУБД.

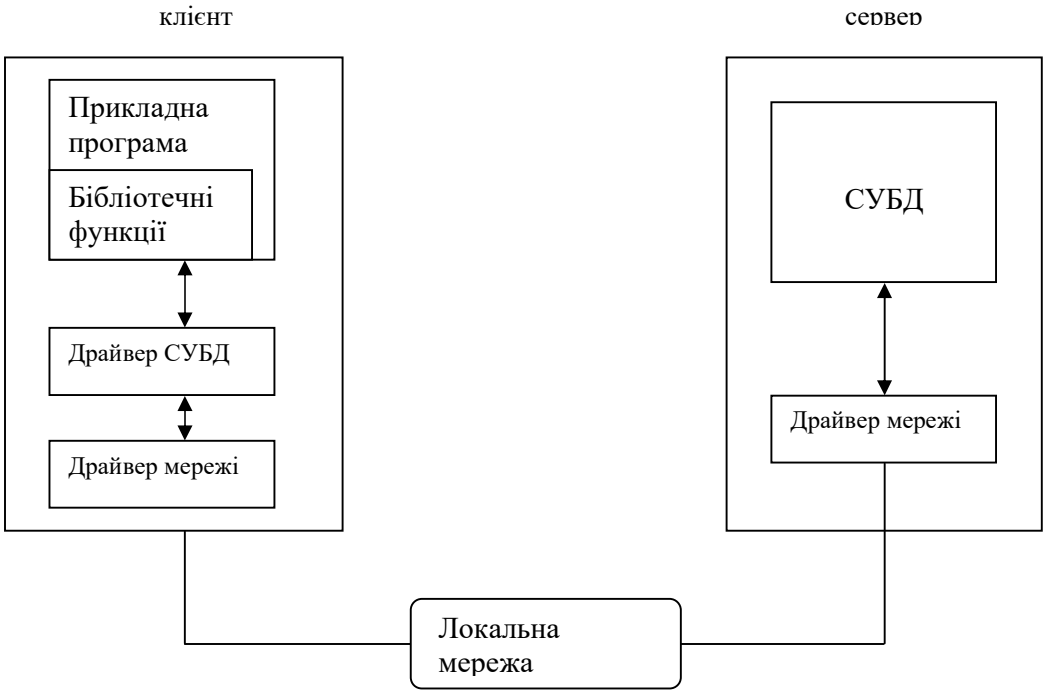


Рис. 7.3. Схема взаємодії користувача з клієнт-серверною СУБД

У чому переваги клієнт-серверних інформаційних систем у порівнянні з їхніми аналогами, створеними на основі мережевих версій настільних СУБД? Одним з найважливіших переваг є зниження мережного трафіка при виконанні запитів. Наприклад, при необхідності вибору п'яти з мільйона записів, які містять в таблиці , клієнтський додаток посилає серверу запит, що сервером компілюється й виконується, після чого результат запиту (ті самі п'ять записів, а зовсім не вся таблиця) передається зворотно на робочу станцію (якщо, звичайно, клієнтський додаток коректно формулює запити до сервера).

Іншою перевагою архітектури клієнт/сервер є можливість зберігання бізнес-правил на сервері, що дозволяє уникнути дублювання коду в різних прикладних програмах, що використовують загальну базу даних. Крім того, у цьому випадку будь-яке редагування даних, у тому числі й редагування позаштатними засобами, може бути зроблено тільки в рамках цих правил.

Крім перерахованих можливостей, сучасні серверні СУБД мають широкі можливості керування користувацькими привілеями й правами доступу до різних об'єктів бази даних, резервного копіювання й архівації даних, а нерідко й оптимізації виконання запитів. Вони також, як правило, надають можливість паралельної обробки даних, особливо у випадку використання багатопроцесорних комп'ютерів як серверів баз даних.

Отже, клієнт-серверна інформаційна система складається в найпростішому випадку із трьох основних компонентів:

- сервер баз даних, що керує зберіганням даних, доступом і захистом, резервним копіюванням, відслідковує цілісність даних відповідно до бізнес-правил і, найголовніше, виконуючи запити клієнта;
- клієнт, який надає інтерфейс користувача, виконує логіку прикладної програми, перевіряє допустимість даних, посилає запити до сервера й одержує відповіді від нього;
- мережа й комунікаційне програмне забезпечення, що здійснює взаємодію між клієнтом і сервером за допомогою мережних протоколів.

Є й більш складні реалізації архітектури клієнт/сервер, наприклад, трирівневі інформаційні системи з використанням серверів прикладних програм, але про це пізніше.

Вперше розглянутий в попередньому розділі програмно-вбудований SQL почав застосовуватись фірмою IBM ще в епоху мейнфреймів, потім вона небезуспішно намагалася перевести цю технологію в середовище мереж з персональних комп’ютерів. Проте конкуренти IBM, такі як Sybase та Microsoft не поспішали доганяти її саме по цьому напрямку. Вони пішли шляхом розробки спеціальних бібліотек, які створювали середовище програмування для прикладних програмістів(Application Programming Interface-API). Прикладна програма викликає функцію SQL API для передачі викликів SQL і отримання результатів SQL-викликів. Сутність Windows , Unix та деяких інших сучасних операційних систем така, що підхід на базі API розглядається як головний спосіб розробки і підтримки програмних систем взагалі: від організації файлового вводу/виводу до використання якихось хитромудрих математичних обчислень. Тому ідея використання SQL API швидко завоювала популярність. З часом з’явилося декілька різновидів бібліотек, що дуже швидко привело до того, що програмно-вбудований SQL став вживатися значно менше. Особливо ця тенденція підсилилася з кінця 90-х років минулого століття. Про причини цього поговоримо дещо пізніше, а зараз спробуємо розібратися в складних аспектах застосування SQL API-бібліотек для розробки інформаційних систем.

Розглянемо як працюють прикладні програми, що використовують різні API. Спочатку відзначимо спільні риси механізмів використання. Аналізуючи реальне функціонування різних SQL API – бібліотек можна відзначити, що принцип роботи при

переході від однієї бібліотеки до іншої не зазнає принципових змін. Схема роботи прикладної програми разом з SQL API виглядає в такий спосіб [8]:

- програма отримує доступ до бази даних шляхом виклику однієї або декількох API-функцій, що підключають програму до СУБД і до конкретної бази даних;
- для пересилання інструкцій SQL у СУБД програма формує інструкцію у вигляді текстового рядка й потім передає цей рядок як параметр при виклику API-функції;
- програма викликає виконання API-функції для перевірки стану переданої в СУБД інструкції й обробки помилок;
- якщо інструкція SQL являє собою запит на вибірку, то, викликаючи API-функції, програма записує результати запиту у свої змінні; звичайно за один виклик повертається один рядок, або колонка даних;
- своє звертання до бази даних програма закінчує викликом API-функції, що відключає її від СУБД [8].

З наявних SQL API на даний момент виділилося кілька бібліотек, «стандартних» у тому розумінні, що вони активно застосовуються спільнотою розробників в усьому світі. Це так звані стандарти де-факто.

7.1. Інтерфейси програмування прикладних програм (API)

Першою із СУБД, для якої була реалізована бібліотека API- функцій була SQL Server, розроблена Microsoft і Sybase і названа бібліотекою баз даних, тобто DB-Library. Дану бібліотеку становлять більше 100 функцій. Основними з них є:

dblogin(); *dbopen*() - підключення до БД;

dbopen();

dbexit() - установка/розрив з'єднання із БД;

dbcmd() - передача інструкції (пакета інструкцій) SQL у СУБД у текстовому вигляді;

dbsqlexec() - вимога до СУБД виконати поточний пакет інструкцій;

dbresults() - одержати результати виконання чергової інструкції SQL у поточному пакеті;

dbcancel() - припинення виконання пакета інструкцій SQL;

dbbind(), *dbdata(), dbnextrow(), dbnumcols(), dbdatlen()...* - для обробки результатів запитів на вибірку даних.

Логіка роботи прикладної програми, що обробляє дані, які зберігаються в базі даних під керуванням Microsoft SQL Server, виглядає в такий спосіб:

•за допомогою зазначених вище функцій (dblogin(), dbopen()) прикладна програма формує відомість про авторизації й намагається встановити з'єднання із СУБД;

•за допомогою СУБД програма відкриває конкретну базу даних, з якої буде відбуватися робота (*dbopen*());

•за допомогою спеціальної функції *(dbcmd())* програма передає в СУБД текст SQL-інструкції, що далі необхідно буде виконати; у бібліотеці DB-Library підтримується так 195 називаний пакетний режим роботи. Даний режим має на увазі можливість створення пакетів інструкцій. Так, викликаючи функцію *dbcmd*() кілька разів, Ви можете передати в СУБД текст декількох команд SQL, які згодом будуть виконані як одна команда;

•використовуючи функцію *dbsqlexec*(), програма викликає виконання інструкцій, переданих раніше за допомогою викликів функцій *dbcmd*();

•викликаючи функцію *dbresults*(), програма може визначити, чи вдалося СУБД виконати чергову інструкцію (як правило, число викликів dbresults() відповідає числу інструкцій у черговому пакеті); у випадку, якщо ми маємо справу із запитом, що повертає набір рядків як результат (запитом на вибірку);

•програма за допомогою викликів функції dbbind() здійснює зв'язування кожного поля результатів запиту з деякою областю оперативної пам'яті.

Далі за допомогою функції dbnextrow() програма виконує перехід до наступного рядка результатів запиту, що приводить до приміщення в буфер нових даних; за допомогою функції dbexit() програма розриває з'єднання з базою даних. У бібліотеці , також, передбачені спеціальні механізми обробки помилок, різні способи передачі результатів виконання запитів у прикладну програму й т.д.

7.1.1. Бібліотека DB-Library

Помітним явищем була поява (1994 р.) у стандарті SQL інтерфейсу рівня виклику - CLI (Call Level Interface), у якому ряд софтверних компаній запропонували узгоджений стандартизований загальний набір робочих процедур, що забезпечують сумісність із усіма основними серверами баз даних. Зауважимо, що на той час Microsoft не входила в цей консорціум. Ключовий елемент CLI - спеціальна бібліотека для комп'ютера-клієнта, у якій зберігаються виклики процедур і більшість часто використовуваних мережних компонентів для організації зв'язку із сервером. Це ПЗ поставляється розробником засобів SQL, не є універсальним і підтримує різноманітні транспортні протоколи.

Використання програмних викликів дозволяє звести до мінімуму операції на комп'ютері-клієнті. У загальному випадку клієнт формує оператор мови SQL у вигляді рядка й пересилає її на сервер за допомогою процедури виконання (execute). Коли ж сервер як відповідь повертає кілька рядків даних, клієнт зчитує результат за допомогою серії викликів процедури вибірки даних. Далі інформація зі колонкив отриманої таблиці може бути пов'язана з відповідними змінними прикладної програми. Виклик спеціальної процедури дозволяє клієнту визначити зчитане число рядків, колонок і типи даних у кожній колонці. Інтерфейс CLI побудований таким чином, що перед передачею запиту серверу клієнт не треба задумуватись про тип оператора SQL, будь то вибірка, оновлення, видалення або вставка.

7.1.2. CLI - інтерфейс рівня викликів.

Помітним явищем була поява (1994 р.) у стандарті SQL інтерфейсу рівня виклику - CLI (Call Level Interface), у якому ряд софтверних компаній запропонували узгоджений стандартизований загальний набір робочих процедур, що забезпечують сумісність із усіма основними серверами баз даних. Зауважимо, що на той час Microsoft не входила в цей консорціум. Ключовий елемент CLI - спеціальна бібліотека для комп'ютера-клієнта, у якій зберігаються виклики процедур і більшість часто використовуваних мережних компонентів для організації зв'язку із сервером. Це ПЗ поставляється розробником засобів SQL, не є універсальним і підтримує різноманітні транспортні протоколи.

Використання програмних викликів дозволяє звести до мінімуму операції на комп'ютері-клієнті. У загальному випадку клієнт формує оператор мови SQL у вигляді рядка й пересилає її на сервер за допомогою процедури виконання (execute). Коли ж сервер як відповідь повертає кілька рядків даних, клієнт зчитує результат за допомогою серії викликів процедури вибірки даних. Далі інформація зі колонкив отриманої таблиці може бути пов'язана з відповідними змінними прикладної програми. Виклик спеціальної процедури дозволяє клієнту визначити зчитане число рядків, колонок і типи даних у кожній колонці.

Інтерфейс CLI побудований таким чином, що перед передачею запиту серверу клієнт не треба задумуватись про тип оператора SQL, будь то вибірка, оновлення, видалення або вставка.

7.1.3. ODBC - відкритий інтерфейс до баз даних на платформі MS Windows

Інтерфейс ODBC (Open Database Connectivity) був розроблений фірмою Microsoft як відкритий інтерфейс доступу до баз даних, що надає уніфіковані засоби взаємодії прикладної програми, яка називається клієнтом із сервером - базою даних. Основна ідея полягала в

розробці універсального інтерфейсу на рівні сімейства операційних систем Windows, що міг би бути підтриманий у різних СУБД. Зрештою Microsoft узгодила протокол ODBC(версія ODBC 3.0) зі стандартом CLI і тому в подальшому його почали називати стандартом ODBC/CLI. В подальшому цей ідейний союз був зафіксований на рівні стандарту SQL3. Проте можливості ODBC ширші ніж ті, що зафіксовані в стандарті – Microsoft ставила задачу не просто спростити розробку баз даних за рахунок уніфікації драйверів, але і забезпечити можливість взаємодії прикладної програми одночасно з декількома базами даних.

Розглянемо коротко структуру програмного забезпечення ODBC [8]:

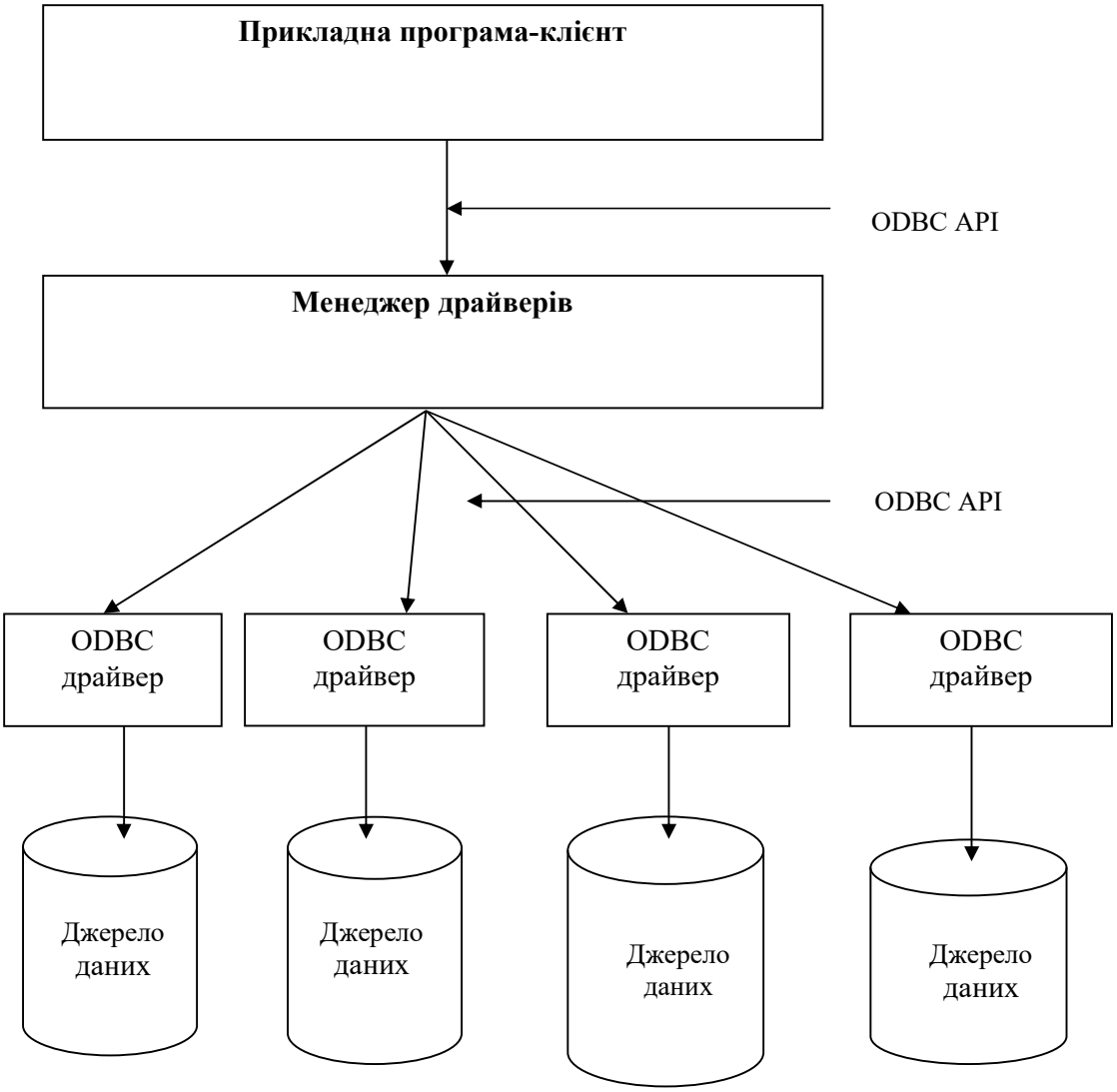


Рис. 7.4. Структура програмного забезпечення ODBC

- інтерфейс викликів функцій ODBC: це так званий верхній рівень ODBC, що містить API і використовується безпосередньо прикладними програмами. Даний API реалізований у вигляді бібліотеки динамічного компонування DLL і входить до складу операційної системи Windows;
- драйвери ODBC: це так званий нижній рівень ODBC, що містить набір драйверів для СУБД, що підтримують протокол ODBC. У рамках технології, для кожної СУБД може бути розроблений відповідний ODBC драйвер, що буде проміжною ланкою між прикладною програмою й API, транслюючи виклики функцій API у виклики внутрішніх спеціалізованих функцій СУБД. У такий спосіб вирішується проблема стандартизації. Для багатьох сучасних СУБД існують спеціалізовані драйвери ODBC, які спеціально встановлюються в операційну систему;

- диспетчер драйверів ODBC**:** даний програмний механізм представляє середній рівень ODBC, управляючи процесом завантаження необхідних драйверів.

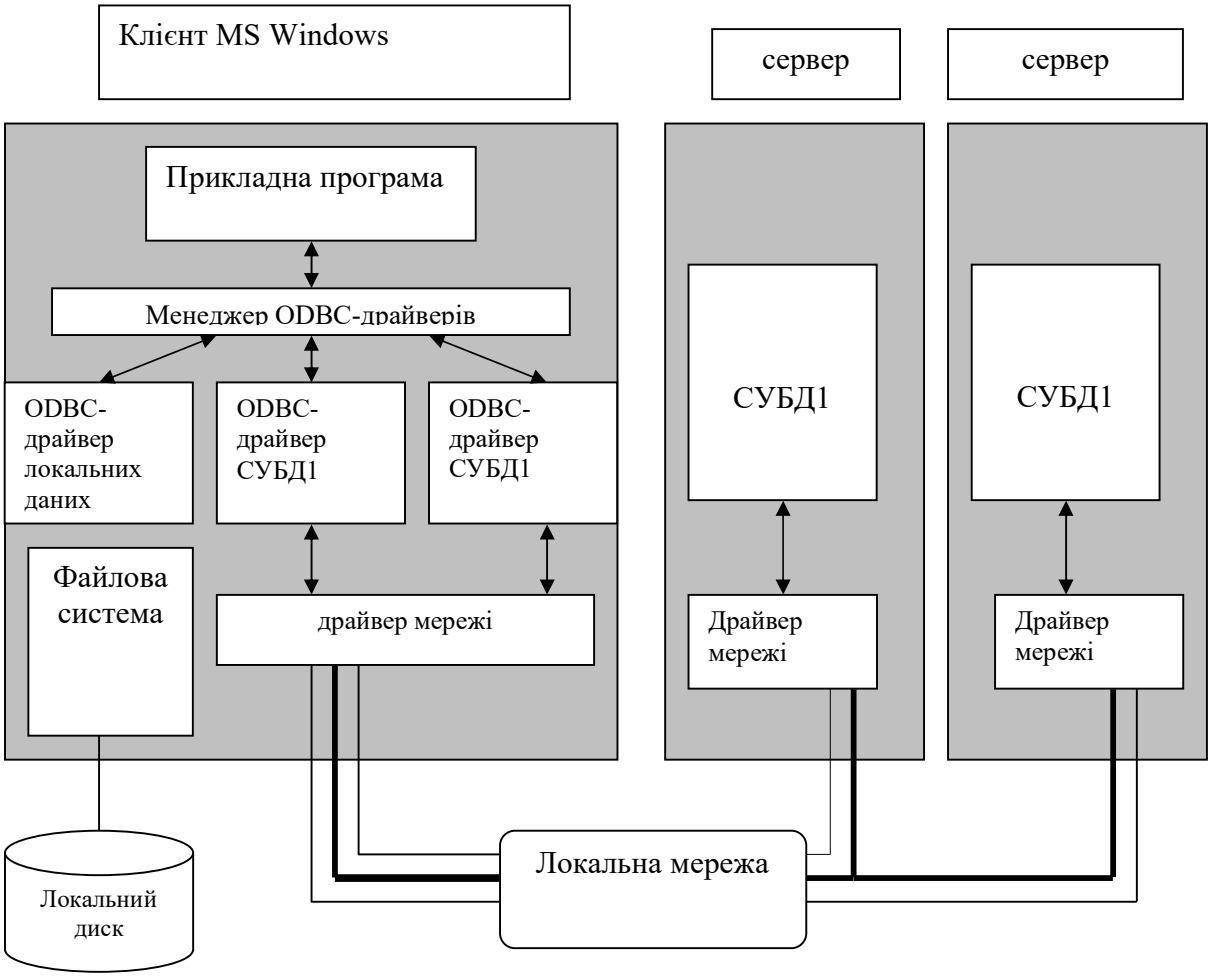


Рис. 7.5. Архітектура ODBC

ODBC інтерпретує оператори SQL під час виконання. Основна програма не потребує прекомпіляції для виконання різних операторів SQL, як і не треба компілювати окремі версії програми для різних джерел даних. Перш ніж реалізувати оператори SQL, прикладна програма ODBC повинна виконати ряд дій, щоб під'єднатись до джерела даних. Типові дії звертання ODBC:

```
SQLAllocEnv();.//Виділяє середовище ODBC
    SQLAllocConnect();//Виділяє пам'ять для підключення
        SQLConnect();//Завантажує драйвердо джерела, що підключається
            SQLAllocStmt(); // Виділяє пам'ять для оператора SQL
                    //Виконання операторів ODBC
                    SQLPrepare();
                    SQLExecute();
                    SQLBindCol();
                    SQLFetch();
             SQLFreeStmt(); //Звільняє пам'ять,  виділену для оператора
        SQLDisconnect();// Призупиняє драйвер і відключає від джерела
```

SQLFreeConnect();// Звільняє пам'ять, виділену для підключення

SQLFreeEnv();// Звільняє середовище, перериває сеанс

Рис. 7.6. Типова структура прикладної програми, що використовує ODBC

Виклик *SQLAllocEnv()* ініціалізує бібліотеку ODBC і повертає дескриптор типу *SQLHENV*. Перед використанням функцій ODBC API прикладна програма-клієнт створює дескриптор (ідентифікатор) оточення, що визначає глобальний контекст для доступу до джерел даних. Для підключення до бази даних необхідно створити дескриптор (ідентифікатор) з'єднання. Для одного дескриптора оточення може бути створено кілька дескрипторів з'єднання. Для виконання SQL-оператора створюється дескриптор (ідентифікатор) оператора. Для одного дескриптора з'єднання може бути створено кілька дескрипторів оператора. За специфікацією ODBC для кожної прикладної програми драйвери можуть підтримувати необмежене число дескрипторів кожного типу. Однак конкретний драйвер може накладати деякі обмеження на кількість дескрипторів.

Дескриптор типу *SQLHDBC*, що повертається функцією *SQLAllocConnect(),* використовується в наступних звертаннях до функцій ODBC для посилання на певне підключення. Одне застосування може підтримувати декілька відкритих підключень. Функція *SQLConnect()* шляхом завантаження драйверу і підключення до джерела даних встановлює з'єднання. Це звертання до функції має варіанти; наприклад, виклик *SQLDriverConnect(),* можна використовувати для підключення до джерел даних, які не встановлюються за допомогою програми початкової установки ODBC. *SQLBrowseConnect()* дозволяє застосуванню циклічно продивлятися джерела даних. Виділяючи пам'ять для оператора SQL за допомогою функції *SQLAllocStmt()* на окремому кроці, ODBC забезпечує механізм, при якому оператори можуть конструюватися і використовуватися не один раз перш, ніж буде виділена пам'ять. Після чотирьох звертань, як правило, програма ODBC створює звертання до бази даних для виконання операторів SQL. Воно може використовувати функцію *SQLPrepare()* для підготовки (компіляції) оператора SQL і функцію *SQLExecute()* для дійсного його виконання. Як альтернативи виклику *SQLPrepare()* та *SQLExecute()* програми можуть використовувати функцію *SQLExecDirect()* для виконання оператора SQL за одну дію. Спочатку колонки зв'язуються із змінними програми за допомогою *SQLBindCol*, потім ці змінні зчитуються після виконання *SQLFetch*() над рядком таблиці. Якщо даних більше нема, *SQLFetch*() повертає SQL*_NO_DATA_FOUND*. Як альтернативу використанню SQLBindCol() для пов'язаних колонок програми можна використовувати функцію SQLGETDATA() для отримання даних з незв'язаних колонок.

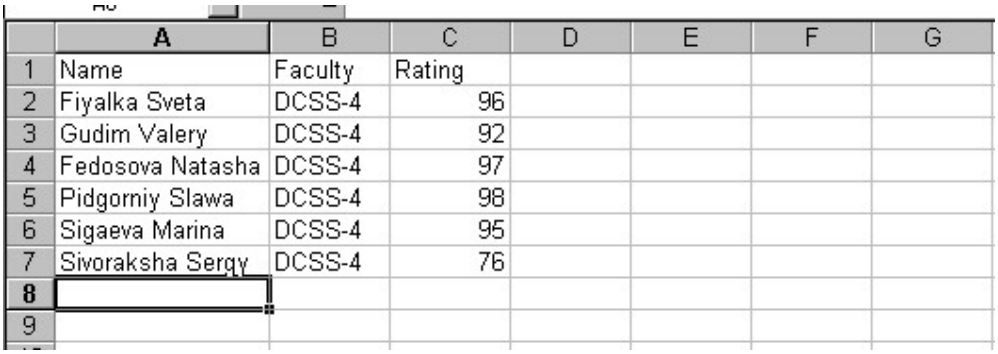
Коли виконання цих дій закінчено, програма має звільнити використані нею ресурси ODBC.

В ODBC 3.0 функція SQLFreeHandle() використовується замість функцій ODBC 2.x SQLFreeEnv(), SQLFreeConnect(), SQLFreeHandle(). Це є приклад узгодження ODBC з CLI.

Для додавання джерела даних треба викликати утіліту встановлення ODBC в Control Panel і, вибравши необхідний тип джерела даних, вибрати драйвер і додати його кнопкою Add. У

діалоговому вікні “Create New Data Source”, що з’явилось, можна вибрати базу даних і встановити необхідні властивості драйвера.

Нижче наведено текст простої програми ODBC, яка зчитує рядки, що містяться в таблиці Excel. Коли доступ до таблиці Excel здійснюється з використанням драйвера Microsoft Excel ODBC, електронні таблиці відіграють роль таблиць бази даних, а рядки в таблиці – роль записів у ній. Таблиця



	A	B	C	D	E	F	G
1	Name	Faculty	Rating				
2	Fiyalka Sveta	DCSS-4	96				
3	Gudim Valery	DCSS-4	92				
4	Fedosova Natasha	DCSS-4	97				
5	Pidgorniy Slawa	DCSS-4	98				
6	Sigaeva Marina	DCSS-4	95				
7	Sivoraksha Sergy	DCSS-4	76				
8							
9							

Рис. 7.7. Вихідна таблиця Excel

Excel складається з 3 колонок, що містять інформацію про прізвище та ім’я студента, група, де він навчається, середній бал за 4 роки навчання. Замість інсталяції цієї таблиці як джерела даних, використовуючи програму початкової установки ODBC, можна скористуватися можливостями функції SQLDriverConnect. Програма може бути скомпільована з командного рядка введенням команди:

cl student.cpp odbc32.lib

Файл student.xls (таблиця Excel) має знаходитись в поточному каталозі.

```
#include <windows.h>
#include <sql.h>
#include <sqlext.h>
#include <string.h>

#define CONNSTR «DBQ=Student.XLS;DRIVER={Microsoft Excel Driver (*.xls)}»
#define CONNLEN (sizeof(CONNSTR)-1)
#define SQLTRY(x,y)
      {
         rc = y;
         if (rc != SQL_SUCCESS)
         {
            char szState[6];
            char szMsg[255];
            SDWORD sdwNative;
            SWORD swMsgLen;
            SQLError(hEnv, hDBC, hStmt, szState, &sdwNative, szMsg, sizeof(szMsg), &swMsgLen);
            printf(«Error %d performing %s\nSQLState = %s\nSQL message = %s\n», rc, x, szState,
szMsg);
            goto Terminate;
         }
      }
```

```
void main(void)
{
   SQLHENV hEnv = 0;
   SQLHDBC hDBC = 0;
   SQLHSTMT hStmt = 0;
   SQLCHAR szConnStr[255];
   SQLCHAR szStmt[255];
   SQLCHAR szName[255];
   SQLCHAR szFaculty[255];
   long nRating;
   SWORD cbConnStr;
   RETCODE rc;
   SDWORD sdwNLen;
   SDWORD sdwFLen;
   SDWORD sdwRLen;
   int i;
   char szResult[1000];

   SQLTRY(«SQLAllocEnv», SQLAllocEnv(&hEnv))
   SQLTRY(«SQLAllocConnect», SQLAllocConnect(hEnv, &hDBC))
   SQLTRY(«SQLDriverConnect», SQLDriverConnect(hDBC, NULL, CONNSTR, CONNLEN,
szConnStr, sizeof(szConnStr), &cbConnStr, SQL_DRIVER_NOPROMPT))
   SQLTRY(«SQLAllocStmt», SQLAllocStmt(hDBC, &hStmt))

   sprintf(szStmt, «SELECT * FROM [Sheet1$]»);
   SQLTRY(«SQLPrepare», SQLPrepare(hStmt, szStmt, strlen(szStmt)))

   SQLTRY(«SQLBindCol», SQLBindCol(hStmt, 1, SQL_C_CHAR, (PTR)szName, sizeof(szName),
&sdwNLen))
   SQLTRY(«SQLBindCol», SQLBindCol(hStmt, 2, SQL_C_CHAR, (PTR)szFaculty,
sizeof(szFaculty), &sdwFLen))
   SQLTRY(«SQLBindCol», SQLBindCol(hStmt, 3, SQL_C_SLONG, (PTR)&nRating,
sizeof(nRating), &sdwRLen))
   SQLTRY(«SQLExecute», SQLExecute(hStmt))

   for (i = 1; (rc = SQLFetch(hStmt)) == SQL_SUCCESS; i++)
   {
      printf(«Record #%d\tName: %s\tFaculty: %s\tRating: %d\n», i, szName, szFaculty, nRating);
   }
   if (rc != SQL_NO_DATA_FOUND)
   {
      SQLTRY(«SQLFetch», rc)
   }
   printf(«Successfully completed.\n»);
Terminate0:
   if (hStmt) SQLFreeStmt(hStmt, SQL_CLOSE);
   if (hDBC) SQLDisconnect(hDBC);
   if (hDBC) SQLFreeConnect(hDBC);
   if (hEnv) SQLFreeEnv(hEnv);

   SQLTRY(«SQLAllocEnv», SQLAllocEnv(&hEnv))
   SQLTRY(«SQLAllocConnect», SQLAllocConnect(hEnv, &hDBC))
```

```
   SQLTRY(«SQLDriverConnect», SQLDriverConnect(hDBC, NULL, CONNSTR, CONNLEN,
szConnStr, sizeof(szConnStr), &cbConnStr, SQL_DRIVER_NOPROMPT))
   SQLTRY(«SQLAllocStmt», SQLAllocStmt(hDBC, &hStmt))

   sprintf(szStmt, «SELECT * FROM [Sheet1$] WHERE Rating>91 ORDER BY Rating DESC»);
   SQLTRY(«SQLPrepare», SQLPrepare(hStmt, szStmt, strlen(szStmt)))

   SQLTRY(«SQLBindCol», SQLBindCol(hStmt, 1, SQL_C_CHAR, (PTR)szName, sizeof(szName),
&sdwNLen))
   SQLTRY(«SQLBindCol», SQLBindCol(hStmt, 2, SQL_C_CHAR, (PTR)szFaculty,
sizeof(szFaculty), &sdwFLen))
   SQLTRY(«SQLBindCol», SQLBindCol(hStmt, 3, SQL_C_SLONG, (PTR)&nRating,
sizeof(nRating), &sdwRLen))
   SQLTRY(«SQLExecute», SQLExecute(hStmt))

   for (i = 1; (rc = SQLFetch(hStmt)) == SQL_SUCCESS; i++)
   {
      printf(«Record #%d\tName: %s\tFaculty: %s\tRating: %d\n», i, szName, szFaculty, nRating);
   }
   if (rc != SQL_NO_DATA_FOUND)
   {
      SQLTRY(«SQLFetch», rc)
   }
   printf(«Successfully completed.\n»);
Terminate:
   if (hStmt) SQLFreeStmt(hStmt, SQL_CLOSE);
   if (hDBC) SQLDisconnect(hDBC);
   if (hDBC) SQLFreeConnect(hDBC);
   if (hEnv) SQLFreeEnv(hEnv);
}
```

Після обов'язкових звертань до функцій SQLAllocEnv та SQLAllocConnect програма викликає SQLDriveConnect. Цей виклик робить можливим відкриття таблиці, яка не встановлювалась з використанням програми початкової установки ODBC і робить це без відображення інтерфейса користувача. Для відкриття таблиці використовуються константи *CONNSTR* та *CONNLEN.* Як тільки підключення до бази даних успішно завершено, виконуються послідовно два оператори SQL:

1) *SELECT * FROM [Sheet$]* - Вибрати всі записи з таблиці Student.
2) *SELECT * FROM [Sheet$] WHERE AVMARK>91 ORDER BY Rating DESC* – Вибрати записи з таблиці Student про тих студентів, що мають середній бал більший 4.5, та впорядкувати виведений список за зменшенням рейтингів студентів.

Ім'я **Sheet$ -** це ім'я, яке надається драйвером для першої таблиці в робочій книзі Excel. Оператор SQL використовується для отримання полів всіх записів. Наступні 4 звертання прив'язують змінні до колонок таблиці. Таке призначення функції *SQLBindCol.* Після послідовного отримання записів, значення полів переміщуються в ці змінні. Самі записи отримуються за допомогою функції *SQLFetch* і

відображаються з використанням printf. Функція *SQLFetch* викликається доти, доки значення, що повертається нею, не відрізняється чим-небудь від *SQL_SUCCESS*. Значення, що повертається, *SQL_NO_DATA_FOUND* показує, що отриманий останній запис, все інше є помилкою і обробляється відповідним чином. Програма завершується обов'язковими звертаннями до функцій *SQLFreeStmt, SQLDisconnect, SQLFreeConnect* та *SQLFreeEnv* для звільнення ресурсів і закінчення зв'язку з джерелом даних.

7.1.4. Протокол OCI

Інтерфейс викликів Oracle [] – Oracle Call Interface – OCI – спеціальний інструмент, що представляє собою програмний інтерфейс в Oracle. Даний інтерфейс з'явився ще в ранніх версіях Oracle і продовжує своє існування донині, незважаючи на успішне сходження ODBC. Більше того, при переході від версії до версії OCI продовжує розвиватися. Так, при переході від версії 7 до версії 8 в OCI були внесені істотні зміни. По суті своєї функції OCI дуже схожі на функції динамічного SQL, особливо в тій частині, що стосується старої версії OCI. У нову версію були внесені зміни, багато з яких ідеологічно запозичені з ODBC. Інтерфейс OCI докладно описаний у літературі і бажаючі можуть розібратися самостійно

7.1.5. Протокол JDBC

JDBC (Java Database Connectivity) – являє собою API для виконання SQL-запитів до баз даних із програм, написаних мовою Java [8]. Розглянемо коротко історію виникнення й основні принципи JDBC. Більш повний опис можна знайти у відповідній літературі, наприклад []. Історично, мова C/C++ довго була домінуючою на ринку розробки комерційного програмного забезпечення. Суперечки в літературі та Internet на тему «Яка мова краща?» продовжуються. Очевидно, що саме поняття «краща» істотно залежить від предметної області, розв'язуваної завдання, наявних ресурсів і т.д. У рамках даного посібника не будемо робити порівняльний аналіз різних мов програмування, але неможливо заперечити той факт, що мова C/C++, як єдина в світі сертифікована за стандартами ISO алгоритмічна мова, все ще домінує на ринку розробки ПЗ. Із цієї причини довгий час всі додаткові засоби програмування (такі як, наприклад, SQL API) орієнтувалися в першу чергу саме на C/C++. Однак з розвитком глобальних мереж, зокрема, Інтернету й всіх супутніх технологій стали з'являтися нові мови, спеціально призначені для роботи в нових умовах. Одна з таких мов є мова програмування Java. Маючи стандартний набір операторів та синтаксис схожий на мову C++, Java швидко завоювала велику популярність на ринку розробки прикладних програм для Інтернет. Спочатку інтерфейс JDBC був розроблений

компанією Sun Microsystems, у даний момент цей API підтримується всіма провідними комерційними СУБД.

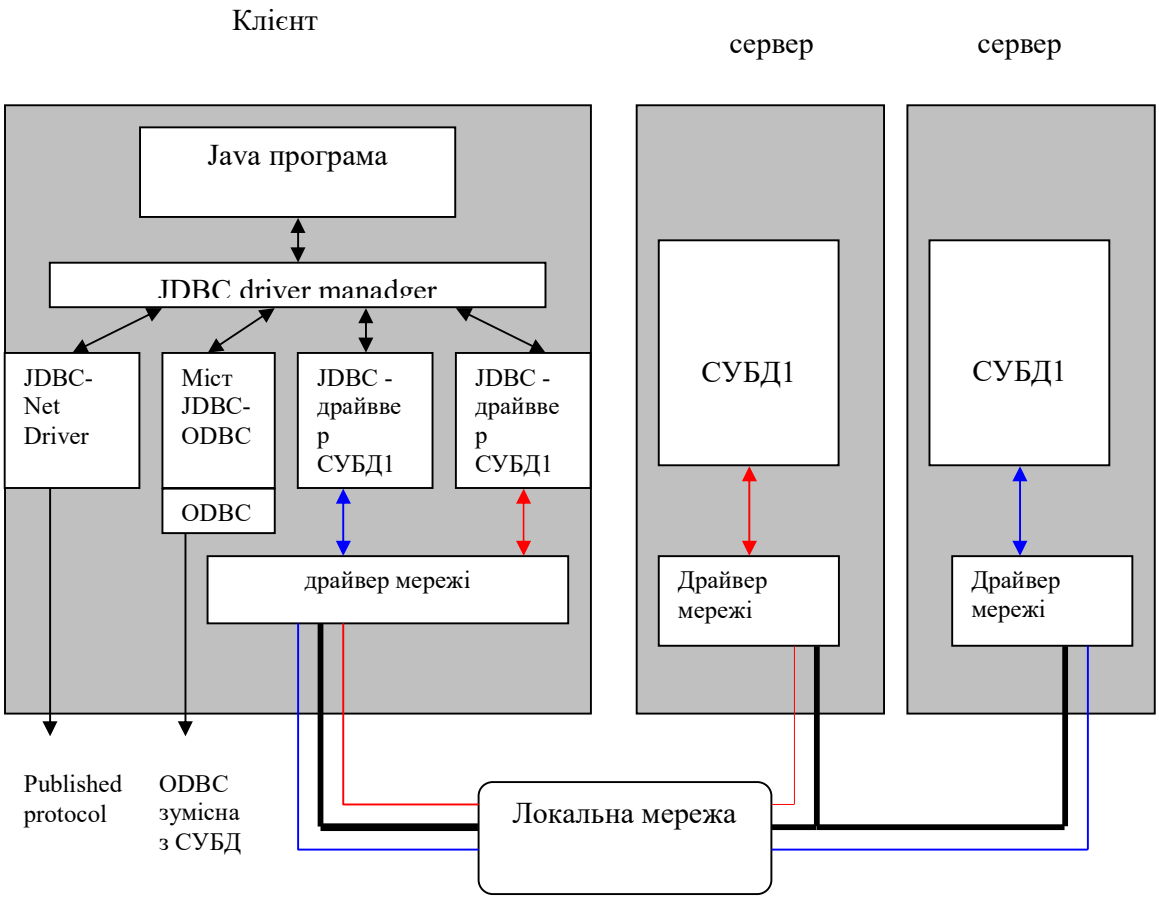


Рис.7.6.Схема виконання програми на Java з використанням протоколу JDBC для доступу до даних

Протокол JDBC багато в чому подібний ODBC (див. Рис.), також побудований на основі специфікації CLI, однак має ряд відмінностей. Це пов’язано специфікою компіляції і використання Java-програм, яка полягає в створенні так званого байт-коду.

Java-програми компілюються в особливий код (так званий байт-код), що виконується на віртуальній машині (Java Virtual Machine). Байт-код є універсальним форматом програми, єдиним для всіх апаратних платформ - і для робочих станцій, і для великих універсальних ЕОМ, і для персональних комп'ютерів.

По-перше, прикладна програма завантажує JDBC-драйвер динамічно, отже адміністрування клієнтів спрощується, більше того, з'являється можливість перемикатися на роботу з іншу СУБД без переналаштування клієнтського робочого місця.

По-друге, JDBC, як і Java у цілому, не прив'язаний до конкретної апаратної платформи, отже проблеми з переносимістю прикладних програм практично знімаються.

По-третє, використання Java-прикладних програм і пов'язаної з ними ідеології «тонких клієнтів» обіцяє знизити вимоги до устаткування клієнтських робочих місць.

Відомо кілька різних версій JDBC. Так, версія 1.0 містила деякі засоби доступу до даних:

диспетчер драйверів (для підключення до різних СУБД);

механізм керування сеансами (для одночасної роботи з декількома СУБД);

механізм передачі інструкцій SQL на виконання в СУБД;

механізм роботи з курсорами (для передачі результатів виконання запитів із СУБД у додаток). Цей перелік певним чином нагадує аналогічний функціональний апарат протоколу ODBC. Версія JDBC 2.0 містить істотні відмінності. Так, внаслідок збільшення можливостей інтерфейсу був проведено його ідеологічний поділ на 2 основні частини: Core API (основні можливості) і Extensions API (так звані розширення). У літературі зазначені наступні основні параметри JDBC [8]:

- Пакетні операції. Програма на Java може здійснити відновлення бази даних у пакетному режимі, тобто одна функція JDBC може додати в базу даних кілька записів, що позитивно позначається на продуктивності програм.
- Курсори з довільним доступом. В JDBC 2.0 існує засіб, що дозволяє переміщатися за результатами запиту довільним чином.
- Оновлювані курсори**.** В JDBC 2.0 курсори, поряд з функцією повернення результату запиту виконують ще й роль відновлення бази даних. Відновлення виробляються при додаванні або зміні одного з рядків у результатах запиту.
- Організація зв'язного пулу. Кілька програм мовою Java можуть користуватися спільним доступом до бази даних, зменшуючи витрати на підключення/відключення до бази данихвід бази даних [8]. Даний перелік можна продовжити (розподілені транзакції, підтримка JNDI і т.д.). У версії JDBC 3.0 з'явилися такі новації, як об’єктно-реляційні розширення SQL і поліпшені механізми обробки транзакцій. Архітектура JDBC бере свій початок від ODBC і в істотній частині повторює її.

7.1.6. Типи драйверів JDBC

На відміну від ODBC, драйвери JDBC підрозділяються на 4 типи. Основні відмінності між цими типами пов'язані з місцезнаходженням API СУБД (на клієнтській або серверній СУБД) і способом доступу до бази даних (через власний API СУБД або через ODBC) :

1. *Міст JDBC-ODBC + драйвер ODBC:* Цей міст, як уже було сказано, використовує інтерфейс JDBC для доступу до драйверів ODBC. Треба відзначити, що виконуваний

код ODBC і, у багатьох випадках, клієнтський код для СУБД повинні бути встановлені на кожній машині, що використовує цей тип драйверів. Отже, цей тип драйверів найбільш типовий для корпоративних мереж, для яких конфігурація клієнтських місць - не проблема, або для серверної прикладної програми, написаного на Java у триланковій архітектурі.

2. *Напів-Java драйвери:* Цей тип драйверів перетворюють виклики JDBC у виклики клієнтського API для Oracle, Sybase, Informix, DB2 і т.д. Як і міст JDBC-ODBC, цей тип драйверів вимагає, щоб на кожній клієнтській станції був установлений певий виконуваний (двійковий) не-java код.
3. *Мережні JDBC-драйвери на чистому Java(Pure Java):* Ці драйвери транслюють програмні виклики JDBC у незалежний від СУБД мережний протокол, що потім транслюється до протоколу СУБД спеціальним проміжним сервером. Цей мережний проміжний серверний додаток з'єднує клієнтів на чистій Java з множиною різних БД. Цей спеціальний мережний протокол залежить від конкретного постачальника, тобто жорстко не стандартизований. Це найбільш гнучкий і захищений в агресивному середовищі Інтернету спосіб.
4. *Драйвери «рідного» протоколу СУБД:* Цей тип драйверів перетворить програмні виклики до мережевого протоколу, що використовується СУБД безпосередньо. Стає можливим пряме підключення клієнтських машин до сервера БД. Цей метод знаходить практичне застосування в Інтранет-середовищі. Позаяк багато таких протоколів патентовані (є власністю постачальників), то за розробку таких драйверів відповідають постачальники СУБД.

Для ефективної і надійної роботи кращим способом доступу до БД є драйвери типу 3 і 4. Типи 1 і 2 драйверів - це вирішення для тих випадків, коли драйвери на чистому Java недоступні. Проте різноманітність джерел даних та СУБД часто примушує шукати засоби інтеграції даних. В ідеології JDBC таким засобом є міст JDBC-ODBC**.**

7.1.7. Драйвер моста JDBC-ODBC

Міст JDBC-ODBC - це драйвер JDBC, що реалізує операції JDBC шляхом трансляції їх в операції ODBC. З погляду ODBC - це звичайна прикладна програма. Міст у такий спосіб надає JDBC-інтерфейс до будь-яких СУБД, для яких доступний ODBC-драйвер. Міст реалізований у вигляді пакета *sun.jdbc.odbc* і містить *native library* для доступу до ODBC. Міст підтримується починаючи з версії ODBC 2.x. Міст реалізований на Java і використовує Java native-методи для виклику ODBC.

Для моста не потрібне спеціальне конфігурування. Міст установлюється автоматично разом з JDK як пакет *sun.jdbc.odbc* і використовується для відкриття JDBC-з'єднання з використанням URL з підпротоколом *odbc*. При завантаженні класу відбувається реєстрація нового драйвера моста в JDBC. Міст розрахований на нереентерабельні драйвери ODBC. Це означає, що міст синхронізує доступ до цих драйверів.

Нижче приводиться приклад Java-програми, яка використовує міст JDBC-ODBC для доступу до даних з використанням *SQL[/]*

Лістинг програми, яка реалізує код Java-програми з використанням мосту JDBC-ODBC

```
// Наступний код може використовуватися як шаблон. Просто треба
// замінити відповідні URL, login, пароль, і SQL-вираз
// на необхідні в кожному конкретному завданні рядка.
//---------------------------------------------------------------------------
//
// Модуль:    SimpleSelect.java
//
// Опис: Тестова програма для інтерфейсу ODBC API.  Ця java-програма
// з'єднується із драйвером JDBC запускає запит «select»
// і відображає всі результати у вигляді таблиці
//---------------------------------------------------------------------------

import java.net.URL;
import java.sql.*;
class SimpleSelect {

    public static void main (String args[]) {
        String url   = «jdbc:odbc:my-dsn»;
        String query = «SELECT * FROM emp»;
        try {
            // Завантажуємо драйвер моста jdbc-odbc
            Class.forName («sun.jdbc.odbc.JdbcOdbcDriver»);
            DriverManager.setLogStream(System.out);
            // Намагаємося з'єднатися із драйвером. Іде пошук
            // хоча б одного із зареєстрованих
            // драйверів, що може обробити цей URL
            Connection con = DriverManager.getConnection (
                url, «my-user», «my-passwd»);
            // Якщо в нас не вийде, то буде
            // згенервано виключення. Тобто, якщо ми до сюди дійшли,
            // ми успішно з'єднались з URL
            // Подивимося, які зауваження були згенервані
            // процедурою з'єднання. Виведемо їх.

            checkForWarning (con.getWarnings ());
            // Отримати об'єкт DatabaseMetaData і відобразити
            // певну інформацію про це підключення
            DatabaseMetaData dma = con.getMetaData ();
            System.out.println(«\nConnected to « + dma.getURL());
            System.out.println(«Driver       « + dma.getDriverName());
            System.out.println(«Version      « + dma.getDriverVersion());
            System.out.println(««);
            // Створюємо об'єкт Statement, щоб можна було
```

```
            // відсилати SQL-запити до драйвера
            Statement stmt = con.createStatement ();
            // Відсилаємо запит, одержуємо об'єкт
            ResultSet rs = stmt.executeQuery (query);
            // Показати всі колонки й рядки з набору даних
            dispResultSet (rs);
            // Закрити набір даних
            rs.close();
            // Закрити оператор
            stmt.close();
            // Закрити з'єднання
            con.close();
        }
        catch (SQLException ex) {

            // Згенерувалося виключення SQLException. Спіймати його
      // і відобразити інформацію про помилку. Помітимо, що може бути декілька
            // об'єктів помилки, з'єднаних разом в один ланцюжок
        System.out.println («\n*** Спіймали SQLException ***\n»);
        while (ex != null) {
            System.out.println («SQLState: « +
                        ex.getSQLState ());
            System.out.println («Повідомлення:  « + ex.getMessage ());
            System.out.println («Vendor:   « +
                        ex.getErrorCode ());
            ex = ex.getNextException ();
            System.out.println («»);
            }
        }
        catch (java.lang.Exception ex) {
            // Отримали помилку іншого типу. Роздрукувати її.
            ex.printStackTrace ();
        }
    }
    //-----------------------------------------------------------------
    // checkForWarning
    // Перевірити, є чи попередження, і відобразити їх. Повертає
    // true, якщо попередження є.
    //-----------------------------------------------------------------
    private static boolean checkForWarning (SQLWarning warn)
            throws SQLException  {
        boolean rc = false;
        // Якщо дано об'єкт SQLWarning, відобразити
        // повідомлення про попередження. Зауважте, що може бути
        // кілька попереджень, зв'язаних у ланцюжок
        if (warn != null) {
            System.out.println («\n *** Warning ***\n»);
            rc = true;
            while (warn != null) {
                System.out.println («SQLState: « + warn.getSQLState ());
                System.out.println («Message:  « + warn.getMessage ());
                System.out.println («Vendor:   « + warn.getErrorCode ());
                System.out.println («»);
                warn = warn.getNextWarning ();
```

```
            }
        }
        return rc;
    }

    //-----------------------------------------------------------------
    // dispResultSet
    // Відображає всі колонки й рядки в даному наборі даних
    //-----------------------------------------------------------------
    private static void dispResultSet (ResultSet rs)
        throws SQLException
    {
        int i;
        // Одержати ResultSetMetaData. Він потрібний для
        // отримання заголовків колонок

        ResultSetMetaData rsmd = rs.getMetaData ();
        // Взяти кількість колонок у наборі даних
        int numCols = rsmd.getColumnCount ();
       // Показати шапку
        for (i=1; i<=numCols; i++) {
            if (i > 1) System.out.print(«,»);
            System.out.print(rsmd.getColumnLabel(i));
        }
        System.out.println(««);

        // Показати всі дані аж до кінця набору даних

        boolean more = rs.next ();
        while (more) {

            // Для кожної колонки в циклі: одержати
            // її значення й показати його

            for (i=1; i<=numCols; i++) {
                if (i > 1) System.out.print(«,»);
                System.out.print(rs.getString(i));
            }
            System.out.println(««);
            // Пересунутися на наступний рядок набору даних
            more = rs.next ();
        }
    }
}
```

Контрольні вправи та запитання

1. Поясніть різницю між схемою використання СУБД в централізованій архітектурі та файл/серверною архітектурою
2. Які переваги та недоліки має клієнт/серверна архітектура порівняно з файл/серверною?

3. Яку роль грає драйвер СУБД? Як він реалізується в сучасних операційних системах?
4. Що собою являє інтерфейс CLI?
5. В чому полягає основна ідея стандарту ODBC?
6. Що собою являють рівні драйверів ODBC?
7. Опишіть основні етапи виконання запиту з використанням ODBC.
8. В чому призначення OLE DB?
9. Які недоліки ODBC можуть бути подолані в OLE DB?
10. Для чого треба було винаходити ADO?
11. Який зв'язок OLE DB, ADO та ODBC?
12. Які мови програмування можна використати для застосування ADO?
13. Назвіть область застосування протоколу OCI.
14. Чим спричинена появу інтерфейсу JDBC?
15. В чому полягають особливості використання інтерфейсу JDBC в мові Java?
16. Вкажіть типи драйверів JDBC та їх особливості .
17. Що являє собою міст JDBC-ODBC ?

РОЗДІЛ 8. Технології доступу до баз даних в середовищах розробки прикладних програм Microsoft

У приверженца точки портрет запятой
Вызывает зевоту.
Как нам быть? На каком языке с немотой
Говорить полиглоту?

(Дмитрий Быков)

Розробка програм для використання – це найбільш поширений спосіб реалізації електронної обробки даних. Описані у попередньому розділі технології широко використовуються розробниками прикладних програм. Але реальна ситуація, обумовлена складною історією розвитку сучасних методів обробки даних не дозволяє обмежитися розглянутим рядом. В першу чергу це стосується продуктів софтверних гігантів - в першу чергу Microsoft, Borland, Sybase та деяких інших, які не обмежували себе розробкою виключно СУБД, а намагалися охопити більш широкий спектр програмних продуктів, зокрема і системи програмування.

Найпотужніші з цих систем програмування за останні 15-20 років є системи програмування Microsoft та Borland. Sybase пішла своїм шляхом, обравши головним дуже своєрідний інструмент для розробки – мову Power Builder, в якійсь мірі відгородившись від конкурентів, з іншого боку певною мірою ізолюваши себе, бо переважна більшість розробників не хотіла відмовлятися від популярних мов – C++, Object Pascal, Visual Basic.

Саме Microsoft та Borland всі ці роки розвивали найбільш потужні системи програмування, ставлячи завдання охопити всі можливі комп'ютерні технології, причому Microsoft опиралася на C++(Visual C++) та Visual Basic, а Borland - C++(C++Builder) та Object Pascal(Delphi). Дещо дивно про це писати, знаючи, що саме Microsoft була розробником ODBC – фактичного стандарту , тим не менше ця софтверна фірма розробила ще й свою власну технологію для доступу до даних, яка має назву Data Access Object (DAO).

8.1. Технологія Data Access Object

Причина цього становища скоріше можна пояснити тим, що на початку 90-х років Microsoft випустила Access – першу настільну СУБД в середовищі Windows та продукт швидкої розробки Visual Basic. ODBC реалізовує низькорівневий інтерфейс, тому програмісти на C и C++ реально використовуть всі переваги технології ODBC, а програмісти Visual Basic - ні . Довгий час Visual Basic (VB) не мав простого доступу до інтерфейсу ODBC. Аж до появи VB 6.0 розробники застосовували високорівневий доступ до даних. Тому, щоб не втратити потенційних клієнтів програмістам на VB була запропонована технологія Data Access Object (DAO) для доступу до даних. DAO (Data Acess Objects) - об'єкти доступу до баз даних, об'єктно-орієнтований інтерфейс для ядра керування базами даних Microsoft Jet. Паралельно Microsoft забезпечила компонентами DAO і бібліотеку MFC, яка підтримує Visual C++. Об'єкти доступу до даних призначені для програмного доступу й керування даними в локальній або віддаленій базі даних, а також для програмного керування самими базами даних, їхніми об'єктами й структурою.

На Рис. 8.1 показано, як DAO використовується для доступу до даних

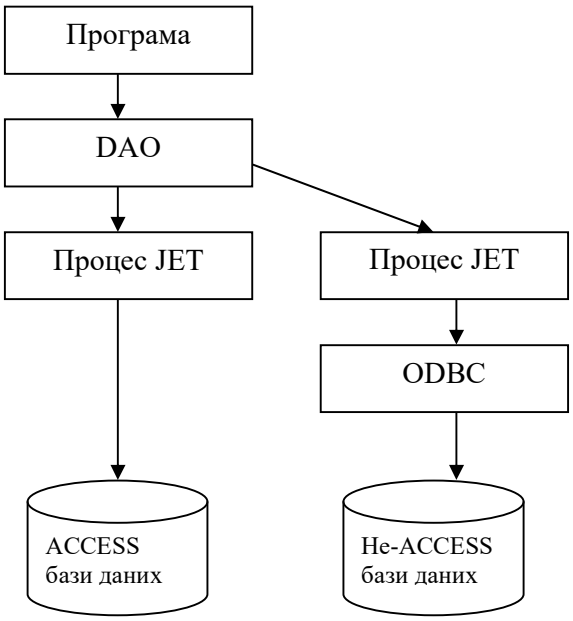


Рис. 8.1. Схема доступу до даних за допомогою DAO

DAO базується на технології баз даних *Microsoft Jet* - процесорі баз даних, призначеному для *Microsoft Access. JET* був першим об'єктно-орієнтованим інтерфейсом для зв'язку з *Access*. Програми, що використовують *Access,* можуть задіяти DAO для прямого доступу до даних. Оскільки DAO створювалася відразу ж слідом за *Access,* застосування цієї технології - найшвидший і найбільш ефективний спосіб доступу до баз даних *Access*. Зараз DAO може працювати й з базами даних, що відрізняються від Access , такими, як *SQL Server і Oracle. DAO* може використовувати ODBC, але, оскільки метод *DAO* спроектований спеціально для взаємодії з *JET, JET* транслює запити між *DAO і ODBC*. Цей додатковий крок трансляції і є причиною вповільнення роботи з не-Access базами даних. Щоб подолати цей недолік, розробники Microsoft, зрештою впровадили *ODBCDirect* - розширення DAO. Microsoft постійно підтримує і розвиває DAO. Наприклад в Microsoft Office 2000 увійшли нові версія *DAO 3.6 і Microsoft Jet 4.0,* у яких реалізована підтримка *Unicode,* тобто в базах даних з'явилася можливість зберігати символи будь-яких національних алфавітів одночасно. І хоча в Microsoft Access 2002 з'явилася нова технологія доступу, за допомогою ADO (Active Data Objects), збереглася можливість роботи з даними і за допомогою DAO (використовуються ті ж версії Jet і DAO). Кожному із цих способів відповідає своя об'єктна модель.

Приведемо приклад програмного створення бази даних по технології DAO, фактично бази даних Access , в середовищі Visual C++, використовуючи бібліотеку MFC.

ЛІСТИНГ 8.1

//Для початку нашу базу необхідно створити. Для створення БД нам буде потрібно тільки

```
//один клас CDaoDatabase.
 //Файл БД буде створюватися в директорії програми. Якщо файл існує, то відбувається
//відкриття БД.
  //Повідомляємо вказівник на об'єкт класу CDaoDatabase:
 CDaoDatabase *pDatabase;
// Створюємо функцію, що буде визначати наявність файлу БД у поточній директорії й
//залежно від результату створювати або відкривати БД:
 CreateOpenDB()
{
 CString strPath;
 //одержуємо ім'я поточної директорії
 int i = ::GetCurrentDirectory((DWORD) (255), strPath.GetBuffer(255));
 strPath.ReleaseBuffer(i);
 strPath = strPath + «\\» + «SDB.mdb»;
//шукаємо файл БД
 WIN32_FIND_DATA FindFileData;
 HANDLE hFind;
hFind = FindFirstFile((LPCTSTR) strPath, &FindFileData);
 if (hFind == INVALID_HANDLE_VALUE)
 {
 //створюємо БД
 CreateDB( );
 }
 //інакше відкриваємо БД
 else OpenDB( );
::FindClose(hFind);
 }
//Для створення БД скористаємося конструктором класу CDaoDatabase:
 CDaoDatabase::CDaoDatabase(CDaoWorkspace* pWorkspace = NULL),
// залишивши як параметр значення за замовчуванням NULL (при цьому буде створений
//тимчасовий об'єкт класу CDaoWorkspace) і функцією:
 virtual void CDaoDatabase::Create(LPCTSTR lpszName, LPCTSTR lpszLocale = dbLangGeneral,
int dwOptions = 0);
 throw(CDaoException, CMemoryException);
//Перший параметр lpszName - це ім'я БД зі шляхом.
//Другий параметр lpszLocale визначає порядок порівняння значень при пошуку в БД. //Повний
```

список його можливих значень можна подивитися в документації MSDN, ми ж //будемо використати значення dbLangCyrillic.

//Третій параметр dwOptions може бути комбінацією можливих значень, які визначають,

//чи буде БД шифруватися (dbEncrypt), і версію ядра Jet бази даних (dbVersion10,dbVersion11,

//dbVersion20, dbVersion30). Але нас не влаштовує жодна із запропонованих версій:
//1.0, 1.1,2.0 і 3.0. В даній версії документованого значення, що визначає версію Jet 4.0 не
//існує
// Тому для створення БД із необхідної нам версією Jet 4.0 необхідно скористатися
//недокументированими в MSDN можливостями бібліотеки MFC для DAO 3.6 і визначити
//цей параметр значенням 0x40.
//Отже, функція, що створює об'єкт класу CDaoDatabase і файл БД на диску виглядає так:
CreateDB()
{
CString strPath;
//одержуємо ім'я поточної директорії й визначаємо шлях до файлу БД
int i = ::GetCurrentDirectory((DWORD) (255), strPath.GetBuffer(255));
strPath.ReleaseBuffer(i);
strPath = strPath + «\\» + «SDB.mdb»;
// створюємо об'єкт класу CDaoDatabase
pDatabase = new CDaoDatabase();
// створюємо БД із версією Jet 4.0
pDatabase->Create(strPath, dbLangCyrillic, 0x40);
}

В результаті на диску в поточній директорії з'явиться файл БД (SDB.mdb), з яким буде зв'язаний вказівник на об'єкт класу *CDaoDatabase - pDatabase.*

8.2. Технологія OLE DB

Механізм доступу до даних ODBC завоював заслужену популярність в середовищі розробників і користувачів баз даних і став фактичним індустріальним стандартом застосування SQL API. Створений набір ODBC-драйверів дозволяє працювати з даними, які навіть не відповідають реляційним таблицям, наприклад, з електронними таблицями Excel. Проте зростаючі вимоги інтеграції даних спонукали Microsoft розробити ще два механізми доступу до даних: OLE DB та на його основі ADO. Саме поєднання цих трьох технологій дозволило стверджувати, що існує універсальний механізм доступу до даних*(Universal Data Access).*

Розглянемо, що собою являє механізм OLE DB. OLE DB - це метод для доступу до всіх наявних даних через стандартний COM-інтерфейс, незалежно від того, де і як дані зберігаються. Дані при цьому можуть бути найрізноманітніші. Це можуть бути реляційні БД, документи, електронні таблиці, файли й повідомлення електронної пошти. В OLE-технології, БД стає компонентом, що називається провайдером (тобто постачальником інформаційних послуг). Провайдер - це частина програмного забезпечення, у якій реалізовані інтерфейси OLE DB. Ідеологія OLE DB передбачає два типи OLE DB-провайдерів- провайдери даних і

сервісні компоненти. Провайдер даних - це компонент програмного забезпечення, що маніпулює даними. Він розташовується між споживачем даних і базою даних. В OLE DB всі провайдери представляють дані в табличному форматі (аналогічному тому, як зберігаються дані в реляційних СУБД і файлах електронних таблиць), у вигляді віртуальних таблиць. Провайдер даних виконує наступні функції:

- одержання від споживача запитів на одержання або модифікацію даних;
- одержання даних з бази даних або їхню модифікацію в базі даних;
- повернення даних споживачеві.

Прикладом провайдерів даних є провайдер *Microsoft Jet 4.0 OLE DB Provider*, що використається для доступу до даних Microsoft Access, а також до даних *I-ISAM (Installable Indexed Sequential Access Method),* файлів робочих книг Excel, сховищ даних Microsoft *Outlook і Microsoft Exchange*, таблиць dBase і Paradox, текстових файлів, файлів у форматі HTML і ін. Ще один приклад OLE DB-провайдера- Microsoft *OLE DB Provider for SQL Server*, що застосовуються для доступу до баз даних *Microsoft SQL Server 6.5 і 7.0*.

Провайдер сервісів (або сервісний компонент) реалізує розширену функціональність, не підтримувану звичайними провайдерами даних, наприклад сортування й фільтрацію даних, обробку транзакцій і SQL-запитів, керування курсором і ін. Сервісний компонент може звертатися до сховища даних безпосередньо або за допомогою відповідного провайдера даних - у цьому випадку провайдер сервісів є одночасно і провайдером, і споживачем. Наприклад, сервісні компоненти, такі як Microsoft Cursor Service for OLE DB і Microsoft Data Shaping Service for OLE DB, можуть використатися разом із провайдерами даних OLE DB для розширення їхньої функціональності. На Рис. 8.2. показано, як компоненти OLE DB взаємодіють між собою. Як можна побачити, споживач може одержувати дані як безпосередньо за допомогою провайдера даних, так і з використанням сервісів, надаваних сервісними компонентами.

Рис. 8.2. Схема обробки OLE DB даних через провайдери

Фактично будь-який компонент, що безпосередньо виставляє свої функціональні можливості через OLE DB- інтерфейс понад локальним форматом даних, є справжнім OLE DB-провайдером. Це може бути SQL-сервер, ISAM- файл, звичайний текстовий файл або навіть потік даних.

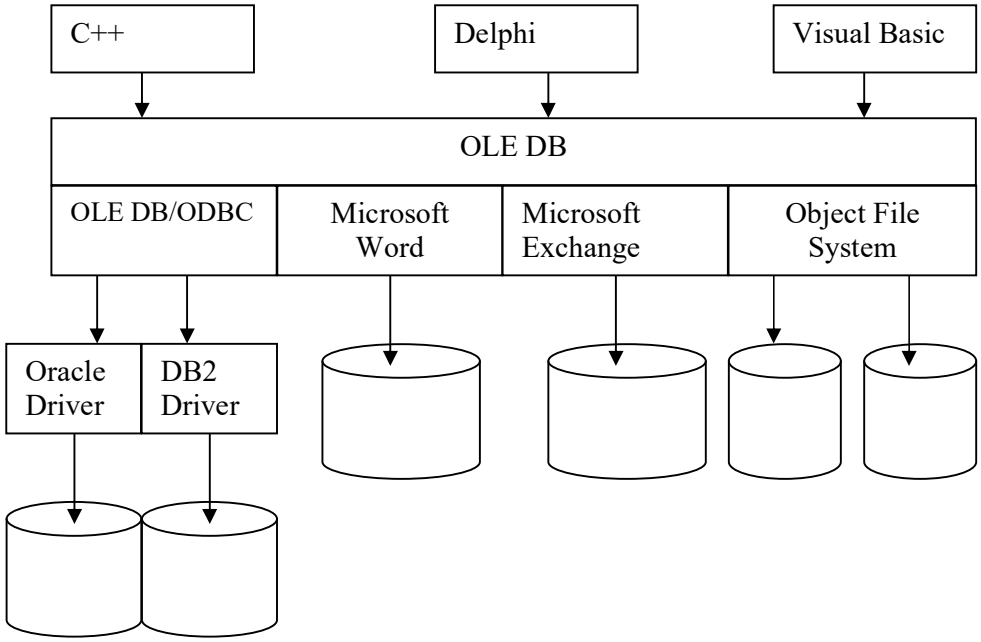


Рис. 8.3. Загальна схема роботи універсального інтерфейсу OLE OB

Як є різні типи провайдерів даних, так є різні типи OLE DB-споживачів даних. Споживачами даних можуть бути програми, написані для роботи з одним провайдером, або

цілі сімейства програм, написаних для роботи з множиною провайдерів. Наприклад, Microsoft Office містить такі програми, як Word, Excel, MS Project, що можуть бути споживачами даних так само, як і їхніми постачальниками, тобто провайдерами. У цьому випадку Microsoft Word безпосередньо звертатається до даних в Microsoft Excel Microsoft Excel, або звертатися до даних у файлах Microsoft Project. Що стосується ODBC, то Microsoft просто доповнила ODBC Driver Manager відповідним провайдером для доступу до ODBC-даних. Цей компонент забезпечує OLE DB-споживачів SQL-даними, розширюючи при цьому клас прикладних програм, що взаємодіють з ODBC-драйверами.

8.2.1. Інтерфейси базового рівня

Інтерфейси базового рівня - це набір мінімуму об'єктів і інтерфейсів, які повинні підтримуватися як провайдерами даних, так і сервісними провайдерами (Рис. 8.4.).

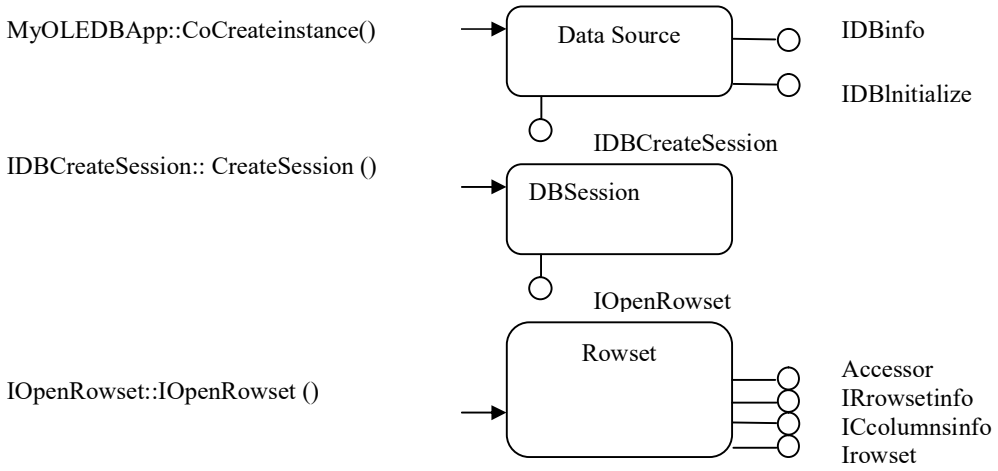


Рис. 8.4. Інтерфейси базового рівня OLE

Так простий провайдер даних, що не підтримує транзакції або запити через об'єкти типу Command, може підтримувати тільки необхідні йому інтерфейси, у той час як більш складні провайдери даних і сервісних провайдерів будуть користуватися ці можливості. Всі провайдери даних і сервісних провайдерів повинні підтримувати об'єкт джерела даних - *Data Source Object (DSO). DSO* являє собою з'єднання із джерелом даних, через який можна впливати на дані. DSO створює багаторазові сеанси зв'язку за допомогою об'єкта *DBSession.* Область зв'язку із джерелом включає інформацію про ім'я й розташування джерела даних і розпізнавальну інформацію, якщо дані перебувають поза захищеним середовищем. Ця інформація забезпечується споживачем даних через функцію IDBInitialize().

Оскільки деякі користувачі, подібні до процесора запитів (який є також сервісним провайдером), можуть потенційно використати провайдер даних без усяких відомостей щодо його реалізації, важливо забезпечити інформацію щодо властивостей вашого провайдера

даних і те, як можна формувати й виконувати запити. Споживач даних може дістати цю інформацію через інтерфейс *IDBInfo*. Функція членства *IDBinfo::GetPropertyInfo* повертає інформацію щодо джерела даних. Така інформація включає ім'я СУБД, версію, підтримуваний синтаксис і т.д.

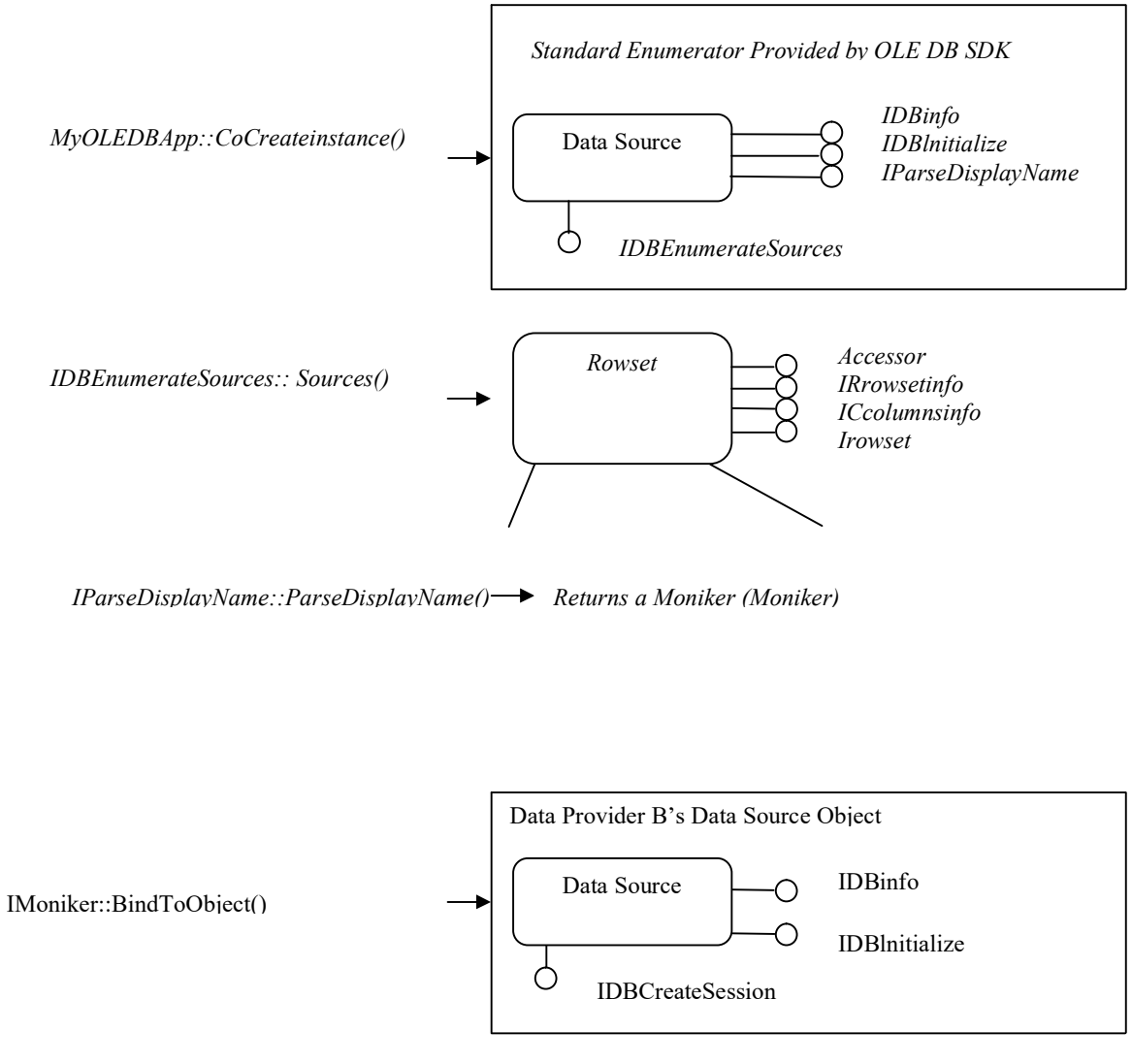


Рис. 8.5. Структура об'єкта джерела даних

IDBInfo::Getkeywords і *IDBInfo::GetLiteralInfo* дозволяють формувати команди, які виймають дані із джерел даних і провайдерів. Ці функції забезпечують ключові слова, які є унікальними для провайдера, і інформацію щодо літералів, що використовуються у текстових командах. Якщо DSO підтримує інтерфейс *IPersistFile*, ви можете зберігати стан з'єднання, контекст і властивості, ви можете додатково зберігати розпізнавальну інформацію, таку, як пароль та інше, хоча це може бути небезпечно. Зверніть, однак, увагу, що, коли

зберігається DSO, жодних активних сеансів не зберігається, як і об'єкти *Command* і об'єкти *Rowset* (про які пізніше); завантаження збереженого файлу відтворює DSO у тому же самому стані, у якому він був створений.

Розглянемо далі важливий для розуміння механізму OLE DB об'єкт *Rowset.* Об'єкт *Rowset* робить дані доступними споживачеві даних. Завдяки об'єкту *Rowset* і його інтерфейсам споживач даних може управляти рядками даних і додавати, змінювати або видаляти дані, прив'язані до джерела даних. Виклик *IDBCreateSession::CreateSession*, що підтримує DSO, створює об'єкт DBSession. Можна звертатися до даних з таблиці, створювати й виконувати запити, управляти транзакціями й створювати таблицю або індекс через цей об'єкт. Як мінімум, всі об'єкти *DBSession* повинні підтримувати інтерфейс *IOpenRowset*. Через *IOpenRowset::OpenRowset* споживач даних генерує *Rowset,* створюючи можливість доступитися до даних з індивідуальної таблиці. *Rowset*, згенерований через інтерфейс *IOpenRowset,* повертає результат SQL-запиту *SELECT* FROM TABLENAME*, де *TABLENAME* є ім'я таблиці, що задає користувачем.

Додатково прості провайдери даних можуть враховувати область визначення таблиць і індексів через два зв'язані інтерфейси, *ITableDefinition* і *IIndexDefintion*, які дозволяють провайдеру створювати таблицю або індекс без використання підтримуваної СУБД мови визначення даних (DDL). *IOpenRowset, ITableDetinition* і *IIndexDefinition* можуть бути використані, щоб створити простого провайдера даних, який віддає неопрацьовані, «брудні» (raw) дані споживачам даних у тому вигляді, у якому вони існують.

Якщо використовується інтерфейс *IDBSchemaRowset,* споживачі даних можуть одержувати інформацію щодо БД без того, щоб точно знати її структуру. Наприклад, ви могли б мати SQL Server, у якому кожна база організована у вигляді набору схем, з таблицями й запитами для кожної окремої схеми, або *БД Microsoft Access* у вигляді контейнера таблиць і контейнера запитів *(queries).*

8.2.2. Послідовне читання таблиці

Існує декілька способів, за допомогою яких споживач даних може звертатися до даних із провайдера даних, - читання таблиці послідовним методом, пряме позиціювання в *Rowset* і скролінг. Підтримка цих різних методів переміщення за даними залежить від інтерфейсів, які провайдер даних підтримує. Всі *Rowset*-об'єкти підтримують мінімальний набір обов'язкових інтерфейсів для звертання до даних. Цей мінімум-набір включає *IAccessor, IColumnsInfo, IRowsetInfo* і *IRowset.*

Давайте подивимося на кроки, які потрібні, щоб прочитати таблицю послідовно. В першу чергу треба створити об'єкт Rowset, використовуючи будь-які функції *IOpenRowset::OpenRowset*, що надається об'єктом *DB Session*, або *ICommand*::*Execute*, яку можна застосовати в об'єкті *Command. IOpenRowset* має справу з більш простим випадком добування всіх даних з таблиці *ICommand* дістає *Rowset*, що зустрічає специфічну область визначення даних або оператор маніпулювання даними. Виконання кожної із цих функцій повертає вказівник на інтерфейс *IRowset*, *pIRowset*, що є найбільш загальним інтерфейсом, використовуваним для читання даних послідовно.

Спочатку треба одержати якусь інформацію про імена колонки та типи колонки. Ця інформація потрібна, щоб створити зв'язування, що є наступним кроком. За допомогою функцій *IRowsetInfo::GetProperties* можна отримати інформацію, яка описує можливості *Rowset*: наприклад, *rowset*-закладки, максимальне число відкритих рядків, які можуть бути активні одночасно, і ще приблизно 60 інших властивостей.

Існує біля 70 Rowset-властивостей, визначених у специфікації OLE DB. Якщо ви хочете дістати деякі дані або запрограмувати сервісний провайдер на виконання певних дій, ви можете додати додаткові властивості, щоб описати спеціальні можливості вашого *Rowsets*. Ви можете також добувати інші об'єкти, пов'язані з *Rowset*, через інтерфейс *IRowsetInfo. IRowsetInfo::GetSpecification* повертає вказівник на об'єкт, що створив *Rowset*. Цей об'єкт звичайно, але не обов'язково, - об'єкт *Command.* Досліджуючи об'єкт *Command*, ви можете отримати текст команди, використовуваний для створення *Rowset*. Якщо повертається об'єкт *DBSession*, ви можете одержати інформацію про схему й про інтерфейси транзакції. Функція *IRowsetInfo::GetSpecification* в основному забезпечує додаткові методи для споживача даних про інформації щодо *Rowset*.

Інтерфейс *ColumnsInfo* забезпечує необхідні методи для споживача даних, щоб визначити метадані або характеристики колонок в Rowset. Із метаданих або зі специфікації команди, що згенерувала *Rowset (IRowsetInfo::GetSpecification),* споживач даних може визначити, які колонки йому потрібні. *ColumnsInfo::GetColumnInfo* повертає звичайно найбільше часто використовувані метадані (ІДЕНТИФІКАТОР - ID, ім'я колонки, порядковий номер колонки, тип даних, і т.д.).

8.2.3. Створення зв'язування (Bindings)

Після знайомства з *Rowset*, треба створити зв'язування. Зв'язування встановлює адресну відповідність між областю пам'яті в буфері користувача - споживача даних зі колонкою даних в *Rowset*. Наприклад, якщо оператор SQL виглядає як *SELECT orderID, customerID, Shipname FROM ORDERS WHERE*, а потім виконується щоб видати *Rowset,* ви б

могли провести операцію зв'язування із іншою колонкою, щоб витягти дані поля CustomerID. Протягом терміну життя *Rowset* колонки завжди мають фіксований порядковий номер, що починається з 1, і нумеруються в тому порядку, у якому колонки з'являються в масиві, що повертається з *IColumnsInfo::GetColumnInfo*.

Структура *DBBINDING,* що визначена в *OLE DB* файлу заголовка *OLEDB.H*, описує зв'язок між полем у структурованому буфері користувача й значенням поля в Rowset (ЛІСТИНГ 8.2.) Кожне зв'язування повинне мати принаймні один з атрибутів: значення, довжину й стан.

ЛІСТИНГ 8.2. DBBINDING Structure

```
typedef struct tagDBBINDING
{
DBCOLUMNPART dwPart; // Значення 1 довжина 1 Стан
DBPARAMIO eParamIO; // Як Accessor використовується (Input/Output parameter)
ULONG iColumn; // Порядковий номер колонки
ULONG dwType; // DBTYPE користувальницької структури
ITypeInfo __RPC_FAR *pTypeInfo; // абстрактний тип даних у специфікації COM
DBNUMERIC __RPC_FAR *pNum; // Значення властивостей об'єкта
ULONG obValue; // Значення зсуву користувальницької структури
ULONG cbMaxLen; // Розмір поля
DBOBJECT *pObject; // Інформація для зв'язування OLE-об'єкта LONG obLength;
// Зсув до довжини  структури користувача
ULONG obStatus; // Зсув до статусу стану  Структури користувача
} DBBINDING;
```

Значення, визначені в *dwType, cbMaxLen* і *obValue*, говорять про структуру, визначеної в області даних користувача, і про її розмір, тобто про те, скільки необхідного простору потрібно зарезервувати. Довжина вказує дійсну довжину даних головним чином для типів даних змінної довжини. При вживанні типів даних фіксованої довжини вона встановлюється з фактичної ширини колонки, прочитаної з *Rowset.* Атрибут стану зв'язування дозволяє споживачеві даних одержувати інформацію стану при читанні або записі в *Rowset*. Стан може вказувати, чи є значення вказівника порожнім, або дійсно відбувся обмін даними. Нижче в приведеному прикладі(ЛІСТИНГ 8.3.) мовою С показано, як створити зв'язування, що відображає дані в кожній колонці *Rowset* відносно їхнього місця розташування в буфері даних користувача. Зверніть увагу на використання параметра *pColumnInfo* у визначенні інформації зв'язування. Інформація, що втримується в *pColumnInfo,* отримана шляхом виклику *IColumnsInfo::GetColumnInfo.*

ЛІСТИНГ 8.3. Створення зв'язування

```
//**************************************************
// SetupBindings
// Purpose:
// Зв'язує дані, що перебувають у колонках Rowset
// з буфером, зазначеним користувачем
// Parameters:
// ULONG ulCol - число стовпчиків в Rowset для зв'язування
// DBCOLUMNINFO* pColumnInfo - вказівник на метадані, що описують колонки
// DBBINDING* rgBind_out - вказівник на масив зв'язаних
// структур, одна структура на кожну колонку
// ULONG* pcBind_out - вказівник, через який можна одержати
// номер зв'язаної колонки (число
// дійсних елементів в rgBind out)
// ULONG* pcMaxRowSize_out - вказівник на розмір буфера, необхідного для
// розміщення найдовшого рядка
//**************************************************
HRESULT SetupBindings (ULONG cCo, DBCOLUMNINFO* pColumnInfo, DBBINDING*
rgBindout, ULONG* pcBind_out, ULONG* pcMaxRowSize out)
{
ULONG dwOffset = ПРО;
for (UNLONG ulCol = ПРО; ulCol < cCol; ulCol++)
{
rgBindout[ulCol].dwPart =
DBCOLUMNPART_VALUEIDBCOLUMNPART_LENGTH  DBCOLUMNPART_STATUS;
rgBindout[ulcol].eParamIO = DBPARAMIO_NOTPARAM;
rgBindout[ulCol].iColumn = pColumnInfo[ulCol].iNumber; rgBindout[ulCol].dwType =
DBTYPE_STR;
rgBindout[ulCo].pTypeInfo = NULL;
rgBindout[ulCol].pNum = NULL; rgBindout[ulcol].obvalue = dwOffset +
offsetof(COLUMNDATA,bData);
rgBindout[ulCol].obLength = dwOffset + offsetof(COLUMNDATA,dwLength);
rgBindout[ulCol].obstatus = dwOffset + offsetof(COLUMNDATA, dwStatus);
rgBindout[ulCol].cbMaxLen = pColumnInfo[ulCol].dwType == DBTYPE_STR?
pColumnInfo[ulCol].cbMaxLength + sizeof(char): DEFAULT_CBMAXLENGTH;
rgBindout[ulCol].pObject = NULL;
dwOffset += rgBindout[ulCol].cbMaxLen + offsetof( COLUMNDATA, bData);
```

dwOffset = ROUND_UP(dwOffset, COLUMN_ALIGNVAL);

}

**pcBind_out = ulCol;*

**pcMaxRowSize_out = dwOffset;*

return NOERROR;

}

8.2.4. Створення аксесорів (Accessors)

Тепер, коли зв'язування встановлено, треба зібрати їх в аксесор, що читає дані із провайдера даних і додатково може записати їх у провайдер. Аксесор містить інформацію або код, щоб упаковувати й розпаковувати рядка, наявні в провайдера даних, або робити над ними якісь заплановані маніпуляції. Ви можете використати їх, подібно програмам обробки рядків, одержуючи й змінюючи зміст рядка й передаючи його в колонки, які ви зв'язали в попередньому кроці. Ви можете створити аксесор у будь-який час протягом виконання Rowset, хоча має бути зроблене перед одержанням даних із провайдера даних або сервісного провайдера. Приведемо приклад аксесора, створеного користувачем - споживачем даних, що застосовує *Iaccessor::CreateAccessor*:

ЛІСТИНГ 8.4.

HRESULT CreateAccessor (DBACCESSORFLAGS dwAccessorFlags,

ULONG cBindings,

const DBBINDING rgBindings[],

ULONG cbRowSize,

ULONG pulErrorBinding,*

*HACCESSOR *phAccessor);*

Параметри пояснюються в Таблиці 8.1.

Таблиця 8.1. Параметри аксесора

DBACCESSORFLAGS dwAccessorFlags [in]	Бітова маска, що описує властивості аксесора і його використання (*DBACCESSOR_READ, DBACCESSOR_READWRITE* і т. д.)
ULONG cBindings [in]	Кількість зв'язаних елементів аксесора
const DBBINDING ґgBindings[] [in]	Масив зв'язаних структур
ULONG cbRowSize [in]	Кількість байт, відведених для одиночної колонки в користувацькому буфері

ULONG pulErrorBinding [out]*	Повернення індексу зв'язаного елемента у випадку виникнення помилки. Рахунок індексів починається з нуля
*HACCESSOR *phAccessor [out]*	Вказівник на область пам'яті, де буде розміщений ідентифікатор *HACCESSOR,* створений аксесором

Для того щоб створити аксесор, який базується на зв'язуванні інформації, описаної в лістингу 8.4. викличте функцію ::CreateAccessor, наприклад:

HACCESSOR hAccessor;

hr = plAccessor->CreateAccessor(DBACCESSOR_READ DBACCESSOR_ROWDATA, cBind, rgBind, 0, NULL, &hAccessor);

Після створення аксесора наступним кроком необхідно завантажити рядки. Завантаження (вибірка) рядків - процес, у якому споживач даних робить запит провайдеру даних, щоб одержати певне число рядків із джерела даних. Рядки вибираються за допомогою методів типу *IRowset::GetNextRows, IRowsetLocate::GetRowsByBookmarks*, *IRowsetLocate::GetRowsAt* і *IRowsetScroll*::*GetRowsAtRatio* (вони визначені в OLE DB-специфікації). Кожний метод повертає ідентифікатори для обробки рядків, передані в програму *HROW*. Приклад показує використання *IRowset*::*GetNextRows* -методу, що повертає масив *HROWs* в останньому параметрі, *rghRows*.

ЛІСТИНГ 8.5. отримання даних з *Rowset*

```
// Read the rows; 160 rows at a time
// Approximately 16KB of data assuming 100 Byte records
while (SUCCEEDED(hr = pIRowset -> GetNextRows (0, NULL, ПРО, 160, &cRowsObtained,
&rghRows)) && cRowsObtained)
{
for (ULONG ulrow = 0; ulrow < cRowsObtained; ulrow++)
{
pIRowset -> GetData (rghRows(ulrow), hAccessor, rgData);
// code to print the data
PrintData (rgData);
}
pIRowset -> ReleaseRows (cRowsObtained, rghRows, NULL, NULL);
CoTaskMemoryFree(rghRows);
}
//release the Accessor and the Rowset
```

pIAccessor -> ReleaseAccessor(hAccessor);

pIRowset -> Release();

При використанні функції *GetNextRows* даних ще одержати не можна. Треба викликати *IRowset::GetData*, щоб дійсно одержати рядки, які будуть повернуті з попередніх викликів *IRowset::GetNextRows;*

IRowset::GetData is prototyped as follows:

*HRESULT GetData (HROW hRow, HACCESSOR hAccessor, void *pData);*

HROW hRow [in] - програма обробки рядка, з якої отримуються дані, і HACCESSOR hAccessor [in] - програма обробки аксесора, що треба використати; void * pData [out] представляє вказівник на буфер, виділений користувачем для одержання даних. Провайдер даних використає зв'язування в аксесор, щоб визначити, як повернути дані. Для кожного зв'язування в аксесорі GetData утримуються дані для певної колонки рядком, позначеним як hRow. Потім ця функція може перетворити дані відповідно до типу даних у зв'язуванні. Потім вона поміщує перетворені дані в структуру користувача. GetData можна викликати будь-яку кількість разів. У кожному виклику можна передавати ті ж самі значення, різні аксесори, різні вказівники на буфери користувача. Це означає, що ви можете отримати стільки копій даних, скільки треба, і їх можна отримувати в різних типах, якщо альтернативні представлення даних забезпечуються провайдером даних

8.2.5. Звільнення рядків

Рядки блокуються провайдером даних доки споживач даних не звільнить їх. Звільняються рядки за допомогою функції *IRowset::ReleaseRows. ReleaseRows* зменшує лічильник посилань рядків, переданих функції; викликається *ReleaseRows* щораз, коли рядок був добутий. Наприклад, якщо рядок був добутий 3 рази, *ReleaseRows* повинен викликатися 3 рази. Коли лічильник посилань на рядок стає нулем, виходить, рядок дійсно звільнений. Нарешті, після того як закінчена робота з *Rowset*, треба звільнити аксесор. Метод *IAccessor*::*ReleaseAccessor* звільняє аксесор і всі ресурси, пов'язані із цим. Споживач даних повинен також звільнити й *Rowset* за допомогою *IRowset::Release,* після звільнення аксесора.

8.2.6. Знаходження й актуалізація DSO

Ми визначили інтерфейси базового рівня й основні функціональні можливості; наступним кроком повинні бути додані додаткові функціональні можливості. Якщо ви

будуєте провайдер даних або сервісний провайдер, ви, імовірно, захочете використати деякі унікальні можливості, щоб сформувати його в розширеному вигляді. Перш ніж почати визначення цих додаткових функціональних можливостей, важливо зрозуміти, як споживач даних використовує DSO. Інформація щодо OLE DB-провайдерів даних і сервісних провайдерів зберігається в реєстрі, який використовує стандарт структури *OLE Component Categories*, що дозволяє OLE-компонентам стати членами довільних категорій. Це також дозволяє OLE-компонентам описувати себе з достатніми подробицями, щоб вони не здійснювали процедуру стандартної прив'язки та реєстрації, що може вимагати з'єднання з віддаленою машиною. При використанні цієї моделі споживач даних може розміщувати й безпосередньо реєструвати сервісні провайдери і провайдери даних або DSO-користувач може читати інформацію з реєстру для визначення правильного провайдера даних. Споживач даних може також отримати список нумераторів(enumerators). *Enumerator* - це COM-сервер, що здійснює *IDBEnumerateSources* (тобто, простіше кажучи, перебір заздалегідь описаних за допомогою стандартного механізму ODBC джерел даних). Не всі провайдери є нумераторами, і не всі нумератори - провайдери. Нарешті, споживач даних може одержувати список сервісних провайдерів, які є COM-серверами, для розширення можливостей провайдера. Звичайно сервісний провайдер експозує підмножину OLE DB-інтерфейсів. Наприклад, процесор запитів міг би робити запити відразу до декількох провайдерів даних.

У більшості випадків прикладна програма, побудована таким чином, щоб мати можливість одержувати дані від специфічного постачальника даних, буде сама знаходити й пускати в хід OLE DB провайдерів даних або сервісних провайдерів. Споживачі даних будуть використати стандарт enumerator, включений як суттєва частина SDK, що буде читати інформацію про нумератори з реєстру. Оскільки кожний провайдер повинен підтримувати *IDBEnumerateSources,* може бути багато нумераторів. Таким чином, стандарт *enumerator* формує корінь ієрархії *IDBEnumerateSources.* Як працюють нумератори? Користувач - споживач даних викликає функцію *CoCreatelnstance*, щоб пустити в хід стандартний нумератор (Рис. 8.6.).

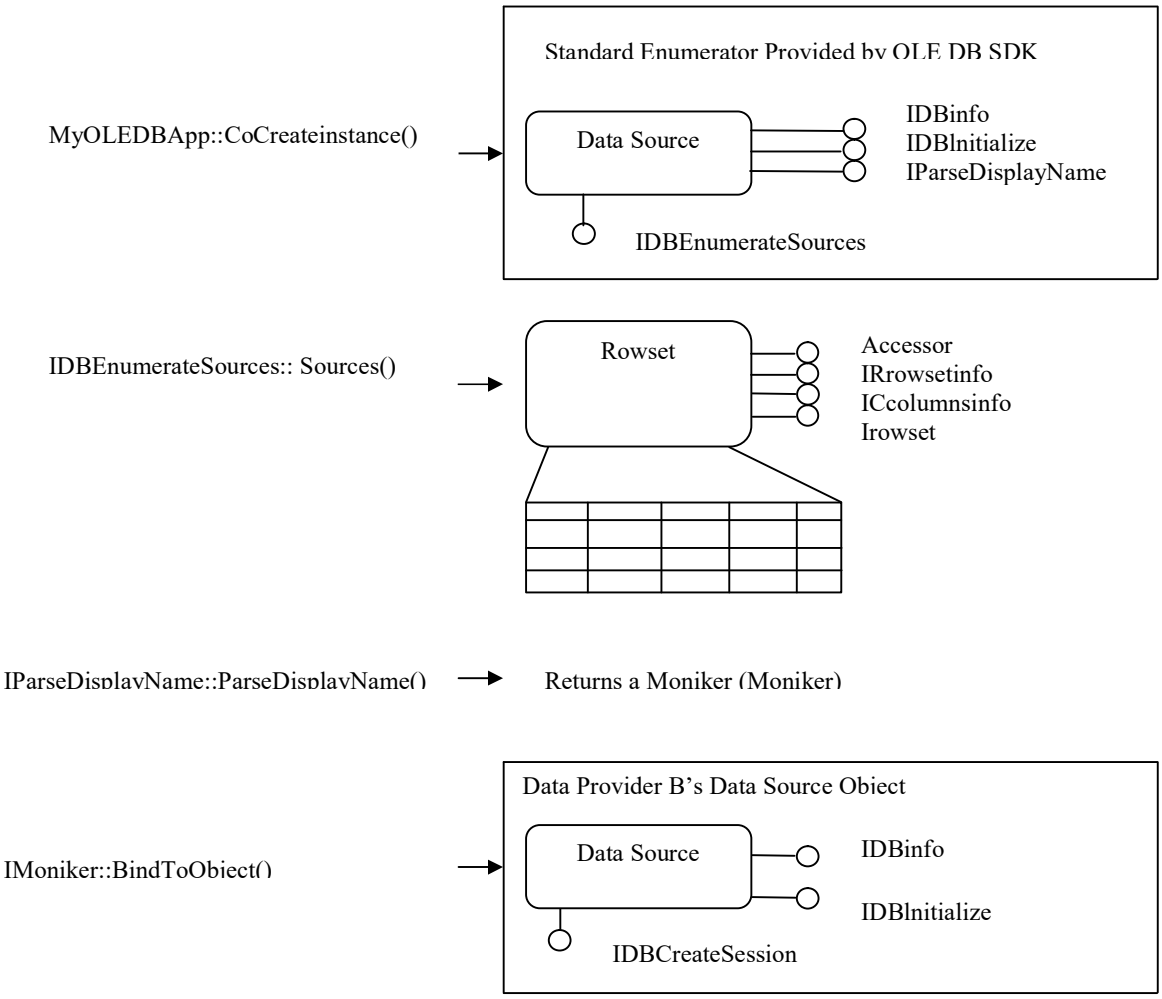


Рис. 8.6. Модель дії нумератора

IDBEnumerateSources::Sources забезпечує *Rowset*, що містить ім'я джерела даних, ім'я синтаксичного аналізатора, опис джерела даних і властивостей джерела даних. Установлене значення властивості *DBSOURCE_ISENUMERATOR* означає, що джерело даних підтримує *IDBEnumerateSources*. Властивість *DBSOURCE_ISPARENT* встановлюється, якщо джерело даних - перебуває в батьківських відношеннях до джерела даних в тільки що викликаному *IDBEnumerateSources::Sources*. Це дозволяє користувачеві застосовувати зворотне перерахування (enumeration). У моделі файлової системи Microsoft це еквівалентно переходу на вищий рівень каталогу "..". Споживач даних може здійснювати навігацію по джерелах даних так само як по файловій системі.

Як тільки споживач даних розміщає джерело даних в *Rowset*, він може використати синтаксичний аналізатор, застосовуючи *IParseDisplayName*, щоб одержати так званий монікер (moniker), тобто ім'я, пов'язане із джерелом даних. Користувач вживає монікер через функцію IMoniker::BindToObiect, щоб зв'язати об'єкт, ідентифікований цим ім'ям .

8.3. Microsoft Active Data Objects(ADO)

Сама Microsoft визначає Active Data Objects(ADO) як Об'єкти Даних ActiveX - незалежна від мови мови програмування об'єктна модель для даних, які оброблені на основі провідника OLE DB. Найчастіше використовується провідник OLE DB - провідник OLE DB для Драйверів ODBC, які виступають як джерела Даних ODBC ADO. ADO - частина компонент Доступу до Даних Microsoft (MDAC). Чим викликана поява ADO? На думку самої компанії Microsoft OLE DB, є інтерфейсом системного рівня для використання системними програмістами. Технологія OLE DB є складною, важкою у використанні і дуже чутливою до помилок. Щоб полегшити роботу з OLE DB для прикладних програмістів, був створений додатковий прикладний рівень, що отримав назву *ADO(Active Data Objects)*. Працювати з ADO істотно простіше, ніж з OLE DB. Технологія ADO призначена для прикладних програмістів. Тобто ADO являє собою високорівневий програмний інтерфейс для доступу до OLE DB-інтерфейсів. Він дозволяє маніпулювати даними за допомогою будь-яких OLE DB-провайдерів, що як входять до складу *Microsoft Data Access Components* деяких інших продуктів *Microsoft*, так і зроблених сторонніми виробниками. ADO містить набір об'єктів, використовуваних для з'єднання із джерелом даних, для читання, додавання, видалення й модифікації даних.

Ця технологія дозволяє маніпулювати з даними незалежно від їхнього формату, типу або розташування. *OLE DB* підтримує SQL запити, транзакції й «курсори» і багато інших потужних технологій для роботи з базами даних. Курсор являє собою вказівник на поточний запис у масиві записів (*record set*). Існують різні способи позиціювання курсору. Наприклад, *forward-only*, коли курсор може рухатися по масиву даних тільки від першого запису до останнього. Це дозволяє збільшити швидкість доступу до даних, коли потрібно швидко прочитати або перебрати всю їх множину.

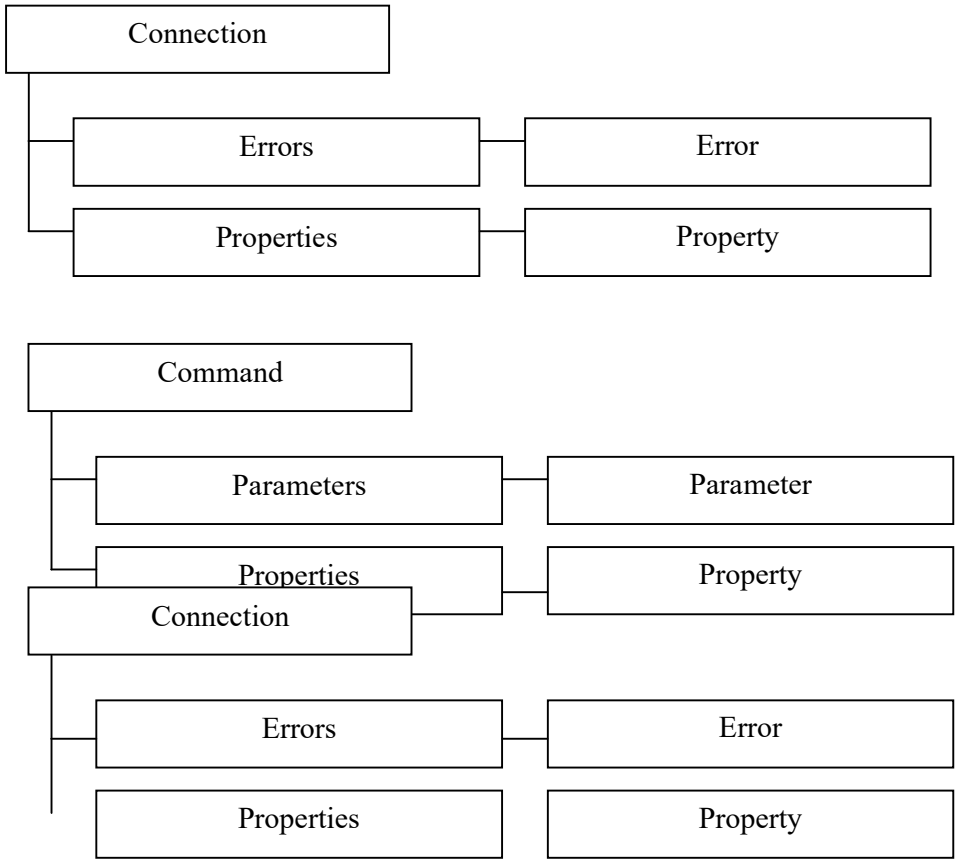
Рис. 8.7. Схема доступу до даних

Роль посередника між програмою й даними виконує OLE DB provider. Він реалізує все техніки, задекларовані OLE DB для роботи зі своїм типом набору даних.

Використовуючи ADO/OLE DB, необхідно лише вказати ім'я провайдера, а потім працювати з даними в абстрагуючись від фізичної структури даних.

Об’єкти Microsoft Active Data Objects організовані у вигляді об’єктної бібліотеки. На Рис.. 8.8. зображена об'єктна модель ADO. Об'єкт ADO Connection застосовується для встановлення зв'язку із джерелом даних. Він представляє єдину сесію. Цей об'єкт дозволяє змінити параметри з'єднання з базою даних, а також почати або завершити транзакцію. Використовуючи об'єкт Connection, ми можемо виконувати команди (наприклад, SQL-запити) за допомогою методу Execute. Якщо команда повертає набір даних, автоматично створюється об'єкт Recordset, що повертається в результаті виконання цього методу. Об'єкт Error використовується для одержання повідомлень про помилки, які виникають у процесі виконання, що засвідчує системність об’єктної моделі ADO, бо помилки при програмуванні не просто можливі, а гарантовані. Об'єкт Command являє собою команду, яку можна виконати в джерелі даних. Команда може містити SQL-пропозицію або виклик збереженої процедури. В останньому випадку для визначення параметрів процедури може бути використана колекція Parameters об'єкта Command. Об'єкт Recordset - це набір записів, отриманих із джерела даних, і може бути використаний для додавання, видалення, зміни, перегляду записів. Даний об'єкт може бути відкритий безпосередньо або створений за допомогою об'єктів Connection або Command.

Об'єкт Field - це колонка у наборі даних, представлених об'єктом Recordset. Він може бути використаний для одержання значень конкретного поля, його модифікації, витягу метаданих, таких як ім'я колонки й тип даних.



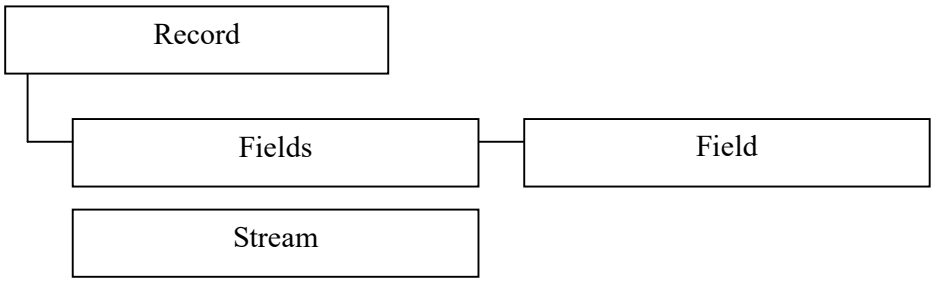


Рис. 8.8. Структура об’єктів ADO

Остання версія бібліотека ADO 2.5, що є складовою частиною операційної системи Windows 2000, містить два нових об'єкти - *Record і Stream*. Об'єкт *Record* становить один запис усередині об'єкта *Recordset* і може бути використаний для роботи з гетерогенними й ієрархічними даними, як ми бачимо значно розширює можливості по обробці даних. Об'єкт Stream становить двійкові дані, пов'язані з об'єктом *Record.* Наприклад, якщо об'єкт *Record* являє собою файл, то його об'єкт *Stream* повинен містити дані у середині цього файлу.

8.4. OLE DB і ADO

Раніше в цьому параграфі вже розглянули деякі основні об'єкти OLE DB. Тепер розглянемо, як об'єкти ADO використовують OLE DB:

- об'єкт *ADO Connection* використовує об'єкти *OLE DB DataSource і Session*. Транзакції підтримуються за допомогою інтерфейсів *ITransaction* і *ITransactionLocal*, метод Execute використовує метод *Execute* інтерфейсу *ICommand* або метод *OpenRowset* інтерфейсу *IOpenRowset*. Більшість властивостей доступно за допомогою *IDBProperties.* Об'єкт *Error* підтримується за допомогою інтерфейсу *IErrorRecords*;
- об'єкт *ADO Command* використовує об'єкт *OLE DB Command* і інтерфейс *ICommand.* Наприклад, його метод *Execute* викликає безпосередньо однойменний метод об'єкта *OLE DB Command*, його властивість *CommandText* доступно за допомогою методів *GetCommandText і SetCommandText* інтерфейсу *ICommandText*. Колекція *Parameters* об'єкта *ADO Command* доступна за допомогою інтерфейсу *ICommandWithParameters;*
- об'єкт *ADO Recordset* використовує об'єкт *OLE DB Rowset*. Він застосовує інтерфейси *IRowset, IRowsetLocate і IRowsetInfo* для реалізації більшості методів, властивостей і колекцій. Об'єкт *Field* використовує інтерфейс *IColumnsInfo.*

ADO (ActiveX Data Objects) – інструментарій, що дає можливість звертатися та маніпулювати даними в базі даних через OLE DB. Серед переваг ADO висока швидкість,

простота використання, невисокі затрати пам'яті та ін. ADO – бібліотека типів COM з дуальним інтерфейсом. Файл бібліотеки *Msado15.dll* . Програмний *ID (ProgID) – ADODB*.

Щоб використовувати ADO з Microsoft Visual Basic або Microsoft Office необхідно встановити посилання на бібліотеку типів ADO. Для цього треба вибрати пункт *References* з меню *Project* і відмітити поле «Microsoft ActiveX Data Objects 1.5 Library», після чого об'єкти ADO, методи та реквізити будуть доступні через VBA Object Browser та IDE Editor.

Для деяких мов програмування необхідно додатково включити *(#include)* файли заголовків :

- Для VC++ - *Adoint.h, Adoid.h.*
- Для VB - *Adovbs.inc.*

Докладнішу інформацію можна знайти в Microsoft OLE DB Programmer's Reference, яка є складовою частиною Platform SDK.

Розглянемо на прикладі механізм ADO .

Приклад. 8.4. Єдиний шлях підтримати ADO у програмі C++ - імпортувати інформацію бібліотеки типу ADO. Бібліотека типу для поточного ADO міститься в межах ADO DLL, *msado15.dll.* Ви можете імпортувати цю бібліотеку типу використовуючи директиву *#import*

Зараз створюється файл-заголовок C/C++ названий Database.h, а далі додають коди тестової програми:

Модуль Database.h

```
#define CATCHERROR(ptr,a)       catch(_com_error &e)\
                                {\
                                        ErrorHandler(e,m_ErrStr);\
                                        ptr=NULL;\
                                        return a;\
                                }

#define CATCHERRGET             catch(_com_error &e)\
                                {\
                                        ErrorHandler(e,m_ErrStr);\
                                        sprintf(m_ErrStr,»%s\n**For Field
Name:%s»,m_ErrStr,FieldName);\
                                        return 0;\

#import «c:\Program Files\Common Files\System\ADO\msado15.dll» \
        rename(«EOF», «EndOfFile»)

typedef ADODB::_RecordsetPtr    RecPtr;
typedef ADODB::_ConnectionPtr   CnnPtr;

class Database;
class Table;
```

```
class Database
{
public:
        CnnPtr m_Cnn;
        char m_ErrStr[500];
        Database();
        bool Open(char* UserName, char* Pwd,char* CnnStr);
        bool OpenTbl(int Mode, char* CmdStr, Table& Tbl);
        bool Execute(char* CmdStr);
        bool Execute(char* CmdStr, Table& Tbl);
        void GetErrorErrStr(char* ErrStr);
};

class Table{
public:
        RecPtr m_Rec;
        char m_ErrStr[500];
        Table();
        void GetErrorErrStr(char* ErrStr);
        int ISEOF();
        HRESULT MoveNext();
        HRESULT MovePrevious();
        HRESULT MoveFirst();
        HRESULT MoveLast();
        int AddNew();
        int Update();
        int Add(char* FieldName, char* FieldValue);
        int Add(char* FieldName,int FieldValue);
        int Add(char* FieldName,float FieldValue);
        int Add(char* FieldName,double FieldValue);
        int Add(char* FieldName,long FieldValue);
        bool Get(char* FieldName, char* FieldValue);
        bool Get(char* FieldName,int& FieldValue);
        bool Get(char* FieldName,float& FieldValue);
        bool Get(char* FieldName,double& FieldValue);
        bool Get(char* FieldName,double& FieldValue,int Scale);
        bool Get(char* FieldName,long& FieldValue);
};
```

Зверніть увагу, що всі *typedef* оголошення, *smart*-вказівники для інтерфейсів, і *enumerators* знаходяться в *msado15.tlh*. Компілятор автоматично генерує цей файл після компіляції згаданої вище *#import* директиви. Вам не потрібно включати цей файл, тому що C++ відповідальний за цей файл. Далі оголошується клас, який формує об'єкт *ADO Connection*. Після цього йде код для класу, який формує об'єкт *ADO Connection*.

Об'єкт *ADOConnection* має багато методів. Але, щоб спростити код, тут оголошені тільки три методи *(Open, Close, and Execute)* , щоб надати доступ до трьох методів об'єкту підключення. Open встановлює фізичний зв'язок до джерела даних, *Execute* виконує команду або запам'ятав процедуру на встановленому підключенні, і *Close* перериває встановлене підключення. Встановлюються *m_Cnn* до порожнього вказівника, тому що у цей момент

немає жодного встановленого підключення. Якщо об'єкт підключення створюється успішно, метод *Open()* намагається встановити фізичний зв'язок з джерелом даних. *Open*() бере три параметри: рядок підключення, призначене для користувача ім'я, і пароль. Якщо фізичне підключення встановлене, ця функція повертає 1. Тому треба використовувати обробник виключної ситуації, щоб перехопити винятковий стан, згенерований методами об'єкту *Connection*. *Execute*() метод бере дійсний запит SQL і повертає *smart* - вказівник об'єкту *Recordset*. Зверніть увагу, що в модулі Test.cpp, приведеному у Лістингу 8.4. треба ініціалізувати COM перед використанням цього класу. Можна зробити це, додаючи наступний код перед використанням цього класу ::*CoInitialize*(*NULL)* на початку програми. Аналогічно правильно деініціалізувати COM використовуючи *::CoUninitialize()* в кінці програми.

Лістинг 8.4.

```
#include <stdio.h>
#include <iostream.h>
#include <comdef.h>
#include <conio.h>
#include «Database.h»

char CnnStr[200]=«Provider=SQLOLEDB.1;Persist Security Info=False;»\
                                «User ID=lab;Initial Catalog=pubs;Data «\
                                «Source=PRASUN@CSE» ;
char ErrStr[200];


int main()
{
        ::CoInitialize(NULL); //Ініціалізація COM
        Database db;
        Table tbl;
        if(!db.Open(«lab»,»cse-98»,CnnStr)) //відкриття бази
        {
                db.GetErrorErrStr(ErrStr); //перехоплення помилки
                cout<<ErrStr<<«\n»;
        }
        if(!db.Execute(«select * from authors order by au_fname,au_id»,tbl))//Вибірка запиту
        {
                db.GetErrorErrStr(ErrStr);
                cout<<ErrStr<<«\n»;
        }
        /*if(!db.OpenTbl(ADODB::adCmdText,»select * from authors order by
au_fname,au_id»,tbl)) //Запис результату у таблицю
        {
                db.GetErrorErrStr(ErrStr);
                cout<<ErrStr<<«\n»;
        }*/
        char id[100];
```

```
if(!tbl.ISEOF())
        tbl.MoveFirst();

while(!tbl.ISEOF())
{
        if(tbl.Get(«au_id»,id))
                cout<<«\nid:»<<id;
        else
        {
                tbl.GetErrorErrStr(ErrStr);
                cout<<«\n»<<ErrStr<<«\n»;
                break;
        }
        if(tbl.Get(«au_fname»,id))
                cout<<« fname:»<<id;
        else
        {
                tbl.GetErrorErrStr(ErrStr);
                cout<<«\n»<<ErrStr<<«\n»;
                break;
        }

        tbl.MoveNext();
}
::CoUninitialize();//Деініціалізація COM
return 0;
```

8.5. Microsoft Universal Data Access

Розглянуті раніше механізми доступу до даних ODBC, OLE DB і ADO сформували компоненти архітектури універсального механізму Microsoft(Universal Data Access) доступу до даних. Універсальний механізм доступу до даних являє собою стратегію надання доступу до будь-якого типу інформації суб'єкта економічної діяльності. Він забезпечує високопродуктивний доступ до різних джерел інформації (включаючи реляційні і нереляційні дані), зокрема до даних, що зберігаються на мейнфреймах, даних електронної пошти і файлової системи, текстових, графічних і географічних даних і ін. Для багатьох сучасних прикладних програм, що використовують дані, характерна подібна різноманітність їх джерел. Більш того, цілком очевидно, що можуть з'являтися нові формати даних і способи їх зберігання, тому розумною вимогою до універсального механізму доступу до даних була б відкритість – тобто можливість підтримки не тільки форматів і джерел даних, що існують в даний час, але і форматів даних, які будуть створені в майбутньому. Призначення універсального механізму доступу до даних фірми Microsoft - надати доступ до джерел даних за допомогою єдиної моделі.

ADO є частиною більш широкомасштабної технології за назвою Microsoft Data Access Components (MDAC). Термін MDAC є спільним позначенням для всіх розроблених компанією Microsoft технологій, пов'язаних із БД. До цього набору відносяться ADO, OLE DB, ODBC і RDS (Remote Data Services). Іноді терміни MDAC і ADO вважають синонімами, однак це неправильно. Насправді ADO є лише однією з частин MDAC. Коли ми говоримо про версії ADO, ми маємо на увазі версії MDAC. До основних версій MDAC відносяться версії 1.5, 2.0, 2.1, 2.5 і 2.6. Компанія Microsoft поширює MDAC у вигляді окремого продукту. Цей продукт може бути завантажений з веб-вузла Microsoft безкоштовно. Мало того, фактично його можна безкоштовно включати до складу власних продуктів. Крім того, MDAC входить у комплект поставки більшості продуктів Microsoft, що мають відношення до баз даних. Компанія Microsoft підтримує лише останню версію MDAC, а також передостанню. Розробник ADO має регулярно звертатися до веб-вузла Microsoft, присвяченому MDAC(www. microsoft.com/data). Тут можна безкоштовно завантажити найсвіжішу версію MDAC. Також рекомендується завантажити MDAC SDK розміром біля 13 Мбайт. На даний час універсальний механізм доступу до даних фірми Microsoft підтримують більшість найбільш популярних настільних і серверних СУБД.

Архітектура універсального механізму доступу до даним Microsoft схематично представлена на Рис. 8.9.

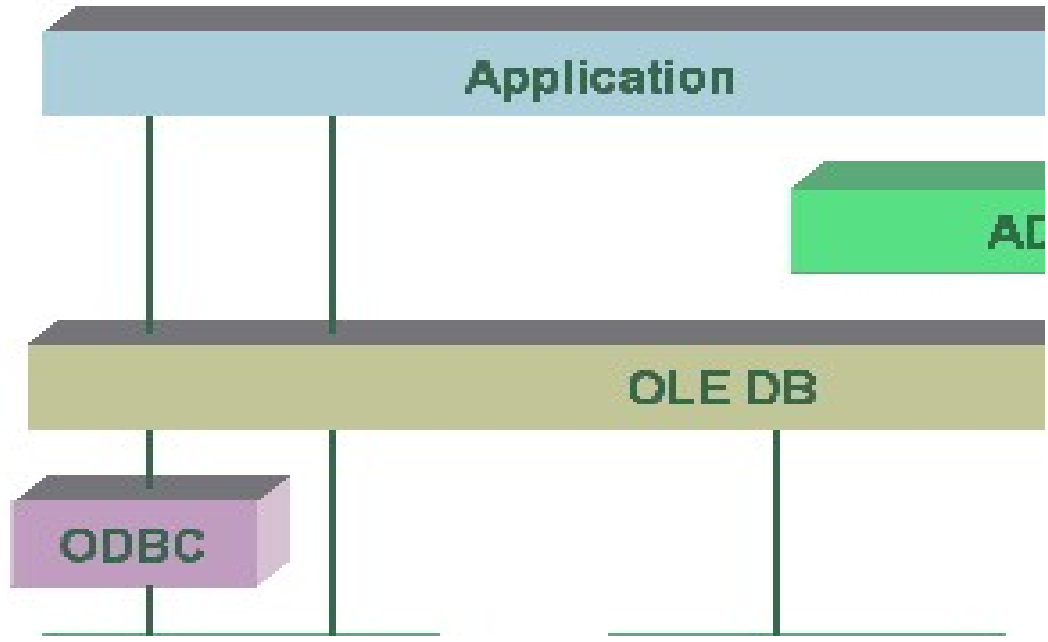


Рис. 8.9. Архітектура ADO

Природньо, що сучасні системи програмування теж надають можливість реалізації як окремих компонент Microsoft Universal Data Access, так і всього механізму в цілому при розробці програм обробки даних. В першу чергу це стосується самої Microsoft.

На Рис. 8.10. приведена схема сучасної архітектури розробки програмних засобів обробки даних з використанням засобів Microsoft. Крім добре відомих програмних засобів: Visual C++ та Visual Basic присутні ще два порівняно нові засоби, обидва нащадки C - C# та Visual J++, які Microsoft реалізувала на противагу Java. Проте Microsoft відмовилась від байт-коду в цих мовах і ,таким чином, в своєму Microsoft Universal Data Access не реалізує інтерфейс JDBC. Навіть не вдаючись в деталізацю, можна зрозуміти, що це надзвичайно складна система, яка потребувала нового системного підходу, що ,зрештою, знайшло своє відображення у технології NET.

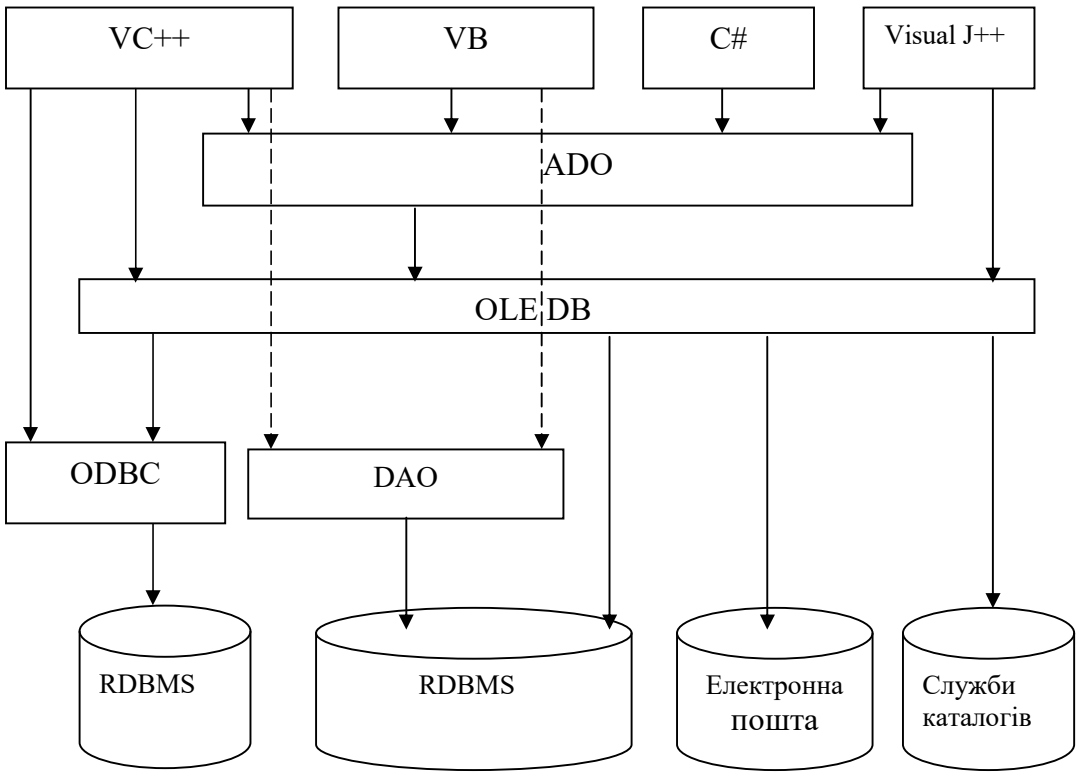


Рис. 8.10.Архітектура сучасних програмних засобів обробки даних Microsoft

Контрольні вправи та запитання

1. Чим обумовлена причина появи технології DAO?
2. Що пов’язує DAO з ядром керування базами даних Microsoft Jet?
3. Як взаємодіють технології DAO та ODBC?
4. Назовіть компоненти архітектури універсального механізму Microsoft доступу до даних?
5. За рахунок чого значно розширені можливості по обробці даних об’єктами бібліотеки ADO 2.5?
6. Що являє собою Active Data Objects(ADO)?
7. Чим викликана поява ADO?

8. Як можна програмно створити базу даних по технології DAO?
9. Що собою являє механізм OLE DB?
10. Що собою являють OLE DB-провайдери?
11. Що виконує OLE DB-провайдер ?
12. Яку розширену функціональність реалізує провайдер сервісів (або сервісний компонент) ?
13. Для чого потрібні інтерфейси базового рівня OLE?
14. Що здійснює Enumerator ?.
15. Опишіть модель дії нумератора.

Розділ 9.Сучасні технології обробки даних фірми Borland

Як вже зазначалося, навіть Microsoft не спроможна охопити весь спектр сучасної технології обробки даних. Тому доцільно розглянути альтернативні технології доступу до даних. По суті тут великого вибору нема – технології Borland та Sybase.

9.1.Бібліотеки BDE фірми Borland

По-суті єдиним інтерфейсом, який довгий час міг успішно конкурувати з технологіями Microsoft довгий час був *BDE.* Розробки почалися ще в з кінця 80-х років 20-го сторіччя. У ті часи на комп'ютерах панували настільні СУБД - dBase, Paradox, FoxPro, Clipper і т.п. Серед форматів настільних СУБД був повний різнобій, і наприклад, хоча Clipper, FoxPro і dBase працювали з форматом DBF, використати таблиці один одного вони фактично не могли через дрібні, але істотні розходження. Обмінюватися даними в ті часи між різними СУБД можна було хіба що за допомогою імпорту-експорту. Багато компаній розуміли, що так далі тривати не може. Та на практиці користувачі працювали тільки з одним форматом даних, а не декількома одночасно.

В 1990-му році Borland придбала компанію Ashton-Tate, а разом з нею й dBase. У такий спосіб Borland стала власником двох приблизно рівноцінних настільних СУБД, але із зовсім різними форматами - dBase і Paradox (ця СУБД, як ми пам’ятаємо, є стала власністю ще з 1985 р.) . Зрозуміло, що для подальшого розвитку цих продуктів зусилля по розвитку форматів даних і роботи з ними фактично подвоювалися. І тому було ухвалене рішення створити якесь універсальне ядро доступу до даних, що могло б працювати з декількома форматами даних одноманітним для користувача чином. Створенню такого ядра також сприяло поява Windows, а отже й поділюваних динамічних бібліотек - DLL. Можна було випускати кілька продуктів, використовуючи ті самі DLL-бібліотеки доступу до даних.

Технологія була названа Open Database Application Programming Interface - ODAPI, і вперше була використана в Quattro Pro 1.0 for Windows у вересні 1992 року. У січні 1993-го ця ж версія ODAPI 1.0 була використана в Paradox 1.0 for Windows, а потім і в dBase 1.0 for Windows. ODAPI спочатку підтримував тільки формати dBase і Paradox, і міг виконувати запити до обох форматів за допомогою механізму Query By Example (QBE). Версія 2.0 була перейменована в IDAPI (слово Open було замінено на Integrated), і роботами з розширення й стандартизації цього інтерфейсу вже займалася не тільки Borland, а цілий комітет IDAPI куди ще входили IBM, Novell та Wordperfect. У цій версії з'явилося Local SQL - ядро для виконання запитів SQL до локальних форматів даних, і IDAPtor - механізм для підключення ODBC-драйверів до IDAPI. Починаючи з версії BDE 3.0 (1996 рік в складі Paradox 5.0 for Windows)стала 32-розрядною. Додавалися нові драйвери для доступу до SQL-серверів DB2, Informix, в BDE 3.5 з'явилися кешовані відновлення (CachedUpdates), драйвер FoxPro і сполучення з DAO, але все це відбувалося протягом досить тривалого часу - з 1996 по 2000. Власне, на цьому розвиток функціональності BDE закінчилося.

На думку багатьох експертів, функціональність BDE перевищує функціональність ODBC , тим не менше ODBC став стандартом, а BDE – ні. Причини цього лежать не в якості програмних продуктів, а в ефективності в конкурентній боротьбі, з якої Microsoft вийшла переможцем. Попри все, BDE активно використовувався і продовжує використовуватися не тільки самим Borland, але й багатьма іншими фірмами. Це Novell (продукт InForms), ReportSmith , CrystalReports (аж до версії 5.0 використовував BDE).

Спочатку механізм Borland Database Engine широко використовувався при створенні застосувань з базами даних за допомогою Borland Pascal 7.0 і Borland C++ 4.5 і 5. Потім засоби розробки Borland були перетворені в засоби швидкої розробки застосувань (Rapid Application Development, RAD) у знаменитому програмному продукті Delphi, причому більшість викликів BDE API було інкапсульовано в компонентах доступу до даних бібліотеки Visual Components Library (VCL). BDE був фактично єдиним механізмом доступу до даних в Delphi і C++Builder, що підтримувався на рівні компонентів класів, а також візуальних компонентів для редагування даних, аж до 5-ої версії обох продуктів - Delphi і C++Builder.

Фізично BDE є набором бібліотек доступу до даних, що реалізовують BDE API - набір функцій для маніпуляції даними, які викликаються із прикладних програм. Ці функції, у свою чергу, можуть звертатися до функцій клієнтського API (у випадку, наприклад, Oracle, Informix, IB Database) або ODBC API (Access 2000, Microsoft SQL Server 7.0, будь-які ODBC-джерела), а також безпосередньо маніпулювати файлами деяких СУБД (dBase, Paradox).

Для доступу до бази даних за допомогою BDE на комп'ютері, що містить клієнтську програму, мають бути встановлені бібліотеки BDE загального призначення, а також *BDE-*

драйвер для даної СУБД. У випадку серверних СУБД такі драйвери носять назву *SQL Links*. Ці драйвери містять BDE API - стандартний набір функцій, створених на основі функцій клієнтських API відповідних СУБД. Серед BDE-драйверів є драйвер, створений для використання ODBC API, - так званий ODBC Link, який застосовується разом з ODBC-драйвером для обраної СУБД.

На відміну від ODBC-драйверів і OLE DB-провайдерів, що випускаються як виробниками СУБД, так і багатьма сторонніми виробниками, BDE-драйвери розробляються винятково самою компанією Borland. Кількість СУБД, для яких є BDE-драйвери, було обмежене п'ятьма найбільш популярними серверними СУБД (Oracle, Sybase, IBM DB2, Informix,Microsoft SQL Server), декількома форматами даних настільних СУБД (в основному ранніх версій СУБД), і сервером IB Database, що входить в комплект поставки засобів розробки Borland. Для доступу до даних решти СУБД за допомогою BDE можна використовувати тільки ODBC-драйвер і *ODBC Link*. Нижче розглянемо це питання детальніше.

Якщо придивитися до файлів BDE, то можна побачити його складові частини.

- IDAPI32.DLL - основний інтерфейс
- BLW32.DLL, BANTAM.DLL- мовні функції
- *.BTL - файли з мовним кодуванням.
- IDBAT32.DLL - операції пакетного копіювання даних
- IDDR32.DLL - модуль роботи з Data Repository
- IDASCI32.DLL - драйвер для роботи з текстовим форматом
- IDDAO32.DLL - драйвер трансляції викликів до DAO
- IDODBC32.DLL - драйвер трансляції викликів до ODBC
- IDPDX32.DLL - драйвер для роботи з форматом Paradox
- IDDBAS32.DLL - драйвер для роботи з форматом dBase і FoxPro
- IDQBE32.DLL - ядро обробки запитів QBE
- IDSQL32.DLL - ядро обробки запитів SQL
- SQLINT32.DLL - SQLLink-драйвер трансляції викликів до Interbase API
- *SQLORA32*.DLL - SQLLink-драйвер трансляції викликів до Oracle Call Level Interface
- *SQL*32.DLL* - інші SQLLink-драйвери

Таким чином, при встановленні BDE «зайві» файли можна не встановлювати.

Також зрозуміло, чому BDE «не працює» з SQL-сервером, якщо не встановлена клієнтська частина цього сервера (те ж саме по відношенню до DAO - без дистрибутива DAO BDE не працюватиме з файлами MS Access). Взагалі клієнтські частини SQL-серверів зовсім

несумісні між собою. Тому неможливо написати універсальний SQL Link. Interbase для Delphi рівноправний з Oracle або будь-яким іншим ODBC-драйвером. На відміну від продуктів Microsoft в BDE немає жодних «обхідних» функцій для роботи з своїми форматами, тобто робота з IB ведеться тільки через SQL Link (без sqlint32.dll BDE взагалі не знає, що таке Interbase). Окреме місце в архітектурі BDE займають Local SQL та QBE Engine.

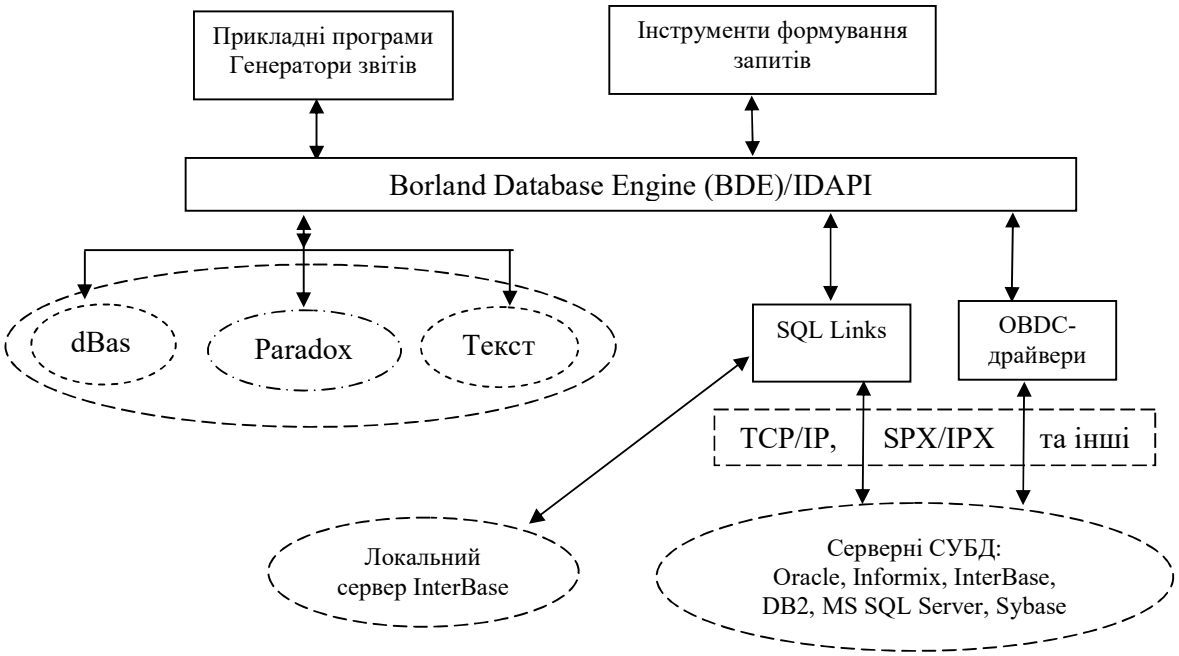


Рис. 9.1. Схема використання BDE прикладними програмами

9.2.Розробка прикладних програм для роботи з базами даних з використанням BDE

Використання *BDE* вимагає адекватного програмного середовища, яким, як вже зазначалося, є системи програмування Delphi або C++ Builder. Обидві вони базуються на одній і тій же об'єктній бібліотеці VCL(Visual Component Library) і хоча останні версії цих програмних продуктів розширили підтримку, зокрема завдяки появі нової бібліотеки CLX, тим не менше основні технологічні моменти використання BDE залишаються незмінними з версії Delphi 2.0. По-перше в об'єктній бібліотеці VCL існує поняття набору даних, рівень абстрагування якого вище ніж просто використання конкретного драйвера баз даних, навіть

такого потужного, як BDE. Клас в об'єктній бібліотеці VCL, який описує набір даних називається TDataSet клас - один з найбільш важливих об'єктів БД. Щоб почати працювати з ним, необхідно глянути на наступну ієрархію(зауважимо, дещо скорочену у порівнянні з фактичною):

TDataSet
|
TDBDataSet
|
-- TTable
-- TQuery

Рис. 9.2. Ієрархія класів компонент для БД

TDataSet містить абстрактні методи для керування даними. *TDBDataSet* «знає», як звертатися з паролями та що потрібно зробити, щоб приєднати програму до певної таблиці. Методи *TTable* «знають» (тобто вже всі абстрактні методи перевизначені), як працювати з таблицею, її індексами й т.д. *TQuery* має певні методи для обробки SQL запитів. *TDataSet* - інструмент, який використати щоб відкрити таблицю, і переміщатися по ній. Практично безпосередньо об'єкт типу *TDataSet* створювати недоцільно. Замість цього, використовується *TTable, TQuery* або інші нащадки *TDataSet* (наприклад, *TQBE*). На найбільш фундаментальному рівні, Dataset це просто набір записів, як зображено на Рис. 9.3.

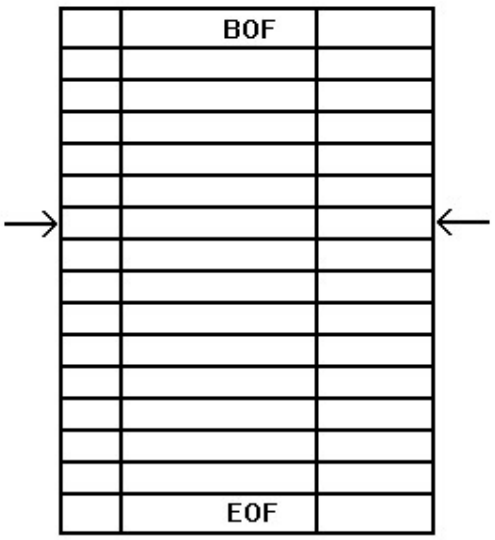


Рис. 9.3. Схема `Dataset`

Будь-який `Dataset` складається з ряду записів (кожен містить N полів) і вказівник на поточний запис. У більшості випадків *dataset* буде мати пряму відповідність із

фізичною таблицею, що існує на диску. Однак, в інших випадках можна виконувати запит або іншу дію, що повертають dataset, що містить або будь-яку підмножину записів однієї таблиці, або об'єднання декількох таблиць. Тому іноді терміни `DataSet` і `TTable` використовуються як синоніми.

Для визначення набору даних необхідно визначити наступні властивості:

- для класу `TTable` - значення властивостей `DatabaseName` і `TableName`;
- для класу `TQuery` - значення властивості SQL і, можливо, властивості `DatabaseName`.

Для того щоб читати дані з таблиць або записувати їх у таблиці, набір даних попередньо повинен бути відкритий.

- Відкрити набір даних можна одним з наступних способів:
- встановити значення властивості `Active` набору даних рівним `True` під час виконання програми (наприклад, `Table1.Active:= True;`) або в режимі проектування в інспекторі об'єктів;
- викликати метод `Open` (наприклад, `Table1. Open;`).
- Аналогічно, закрити набір даних можна викликом методу `Close` або встановивши значення властивості `Active` рівним `False`. Для компонента типу `TQuery` метод `Open` може бути виконаний тільки для закритого набору даних: спроба відкрити вже відкритий набір даних ініціює помилку.

Відкриття набору даних спричиняє:

- ініціацію подій *BeforeOpen* і *AfterOpen*;
- установлення стану набору даних в *dsBrowse*;
- відкриття курсору для набору даних.

Якщо в момент відкриття набору даних трапилася помилка, то стан набору даних встановлюється в *dsInactive*, а курсор закривається. За замовчуванням при переході від одного запису набору даних до іншого відбувається запис всіх зроблених змін у базу даних. Для того щоб можна було скасовувати зроблені зміни, або виконувати відновлення декількох записів, застосовують кешовані відновлення. Вони дозволяють значно знизити мережний трафік за рахунок того, що всі зроблені зміни зберігаються у внутрішньому кеші й при переході від одного запису до інший інформація в базу даних не передається. Щоб включити режим кешованого відновлення, варто встановити значення властивості *CachedUpdates*, що дорівнює True для компонента набору даних. Для присвоєння кешованого відновлення викликається метод *ApplyUpdates*, а для скасування - *CancelUpdates*.

Як і будь яку програму у візуальному середовищі Borland застосування для роботи з БД реалізується через взаємодії візуальних*(TDBGrid, TDBEdit)* і невізуальних*(TTable, TQuery, TDataSet, TField)* . У Палітрі Компонентів невізуальні об'єкти розташовані на сторінці *Data Access*. Візуальні, об'єкти розташовані на сторінці *Data Controls*. Мостом між

цими об'єктами служить *TDataSet*. Для реалізації прикладної прикладної програми, що використовує BDE у візуальному середовищі присутні ряд утиліт.

Одна з них *BDECFG*, створює псевдонім бази даних, який називається аліасом. Це дуже правильна ідея. Аліас зберігається в окремому конфігураційному файлі в довільному місці на диску і дозволяє виключити із програми пряму вказівка на шляхи доступу до бази даних. Такий підхід дає можливість розташовувати дані в будь-якому місці, не перекомпілюючи при цьому програму. Крім шляху доступу, в аліасі вказуються тип бази даних, мовний драйвер та багато іншої інформації. Тому використання аліасів дозволяє легко переходити від локальних баз даних до SQL-серверних баз. Для створення аліаса треба завантажити утиліту конфігурації BDE (програму BDECFG.EXE), що перебуває в директорії, у якому розташовуються динамічні бібліотеки BDE.

Головне вікно утиліти налаштування BDE має вигляд, зображений на Рис.. Для створення аліаса виберіть сторінку «Aliases» і натисніть кнопку «New Alias». У діалоговому вікні, що з'явилося, уведіть ім'я аліаса й виберіть його тип (тип бази даних) зі списку, що випадає. Тип аліаса може бути стандартним (STANDARD) для роботи з локальними базами у форматі dBase або Paradox або відповідати найменуванню SQL-сервера (InterBase, Sybase, Informix, Oracle і т.д.).

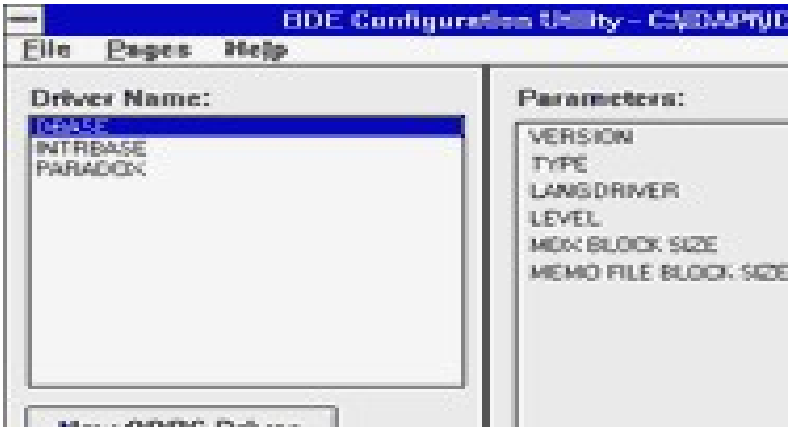


Рис. 9.4. Головне вікно утиліти конфігурації BDE

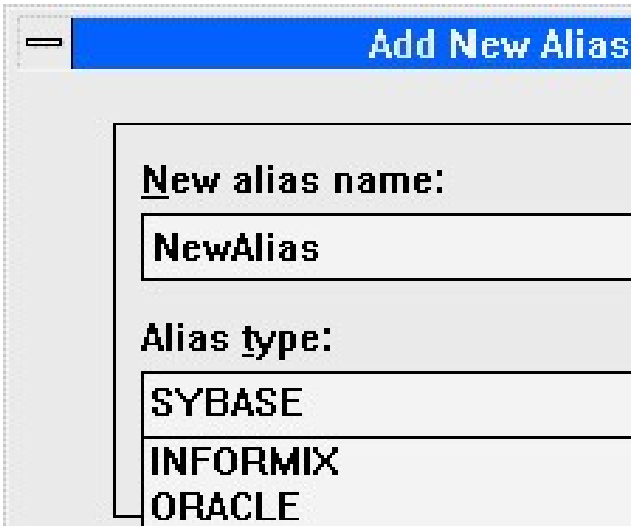


Рис. 9.5. Діалогове вікно утиліти конфігурації BDE

У діалоговому вікні додавання нового аліаса можна вказати ім'я аліаса й тип бази даних Після створення нового аліаса його ім'я з'явиться в списку аліасів на тій же сторінці «Aliases». Однак просто створити аліас не достаньо. Треба ще вказати додаткову інформацію, зміст якої залежить від типу обраної бази даних. Наприклад, для баз даних *Paradox* і *dBase (STANDARD*) потрібно вказати лише шлях доступу до даних:

TYPE *STANDARD*

PATH *c:\users\data*

SQL-сервер InterBase вимагає визначення десяти параметрів наданих в Таблиці 9.1., багато з яких можна залишити встановленими за замовчуванням (крім, звичайно, параметрів *SERVER NAME* і *USER NAME):*

Таблиця 9.1.Параметри InterBase

TYPE	*INTRBASE*
PATH	*C:\USERS*
SERVER NAME	*myserv:g:\users\contacts.gdb*
USER NAME	*SYSDBA*
OPEN MODE	*READ/WRITE*
SCHEMA CACHE SIZE	*8*
LANGDRIVER	*Pdox ANSI Cyrillic*
SQLQRYMODE	
SQLPASSTHRU MODE	*SHARED AUTOCOMMIT*
SCHEMA CACHE TIME	*-1*

У цьому прикладі база даних *CONTACTS.GDB* розміщується в директорії USERS, що перебуває на диску G Windows NT сервера, що називається MYSERV. Ім'я користувача при зв'язку з базою даних по цьому аліасу - SYSDBA. Інші параметри - *LANGDRIVER, SQLQRYMODE, SQLPASSTHRU MODE, SCHEMA CACHE SIZE і SCHEMA CACHE TIME* розглянемо далі.

Параметр *LANGDRIVER* визначає мовний драйвер для доступу до бази даних. Для правильної роботи з кириличними буквами при роботі з базою даних формату dBase потрібно вибрати значення «*dBASE RUS cp866*», при роботі з базами даних формату Paradox і SQL-серверами (у тому числі InterBase) - *«Pdox ANSI Cyrillic».* Крім того, на етапі створення бази даних InterBase необхідно вказати *CHARACTER SET* (набір символів) WIN1251.

Параметр *SQLQRYMODE* з'являється тільки у випадку, якщо встановлено Borland SQL Links для зв'язку з SQL-серверами. Він визначає режим передачі SQL-запитів і може мати три значення:

- *NULL* (порожній рядок - режим за замовчуванням) - запит спочатку посилає на SQL-сервер. Якщо сервер не може виконати запит, останній обробляється локально (це актуально для розподілених баз даних);
- *SERVER* - запит посилає на SQL-сервер. Якщо сервер не може виконати запит, генерується помилка;
- *LOCAL* - запит завжди виконується на робочій станції.

Параметр *SQLPASSTHRU MODE* визначає, чи можуть запити, передані для виконання на сервер (passthrouh SQL, що використовують set-орієнтований підхід), і стандартні виклики BDE (що використовують record-орієнтований навігаційний підхід) оброблятися в тому самому сеансі з'єднання з базою даних (у тому самому «коннекті») - бути «*SHARED*». Він також може мати три значення:

- *SHARED AUTOCOMMIT* (значення за замовчуванням) - для кожної операції по одному рядку таблиці автоматично стартує неявна транзакція, що, у випадку успіху, завершується оператором *COMMIT* (закріпляючим зроблені зміни). Такий підхід щонайкраще підходить для роботи з локальними базами, але неефективний для SQL-серверних баз даних, тому що нові транзакції, що стартують щораз, значно завантажують мережевий трафік.
- *SHARED NOAUTOCOMMIT* - додаток повинне явно стартувати й завершувати транзакцію. Ця установка може привести до конфліктів у багатокористувацькому середовищі, де велика кількість користувачів намагаються обновити той самий рядок таблиці.
- *NOT SHARED* - означає, що запити, передані для виконання на сервер (*passthrouh SQL*), і стандартні виклики BDE (методи Delphi) використають роздільні з'єднання («коннекти»)

з базою даних. Для керування транзакціями через «passthrouh SQL» необхідно встановлювати саме це значення, інакше «passthrouh SQL» і методи Delphi можуть заважати один з однмому, що може привести до непередбачених результатів.

У параметрі *SCHEMA CACHE SIZE* вказується число таблиць бази даних, інформація про структуру яких буде кешуватися, забезпечуючи швидкий доступ до метаданих. Значення цього параметра може бути цілим числом від 0 до 32. За замовчуванням установлене число 8.

Параметр *SCHEMA CACHE TIME* визначає час, протягом якого буде кешуватися інформація з таблиць бази даних. Може мати наступні значення:

- -1 (значення за замовчуванням) - інформація з таблиць кешується до самого закриття бази даних;
- 0 - інформація з таблиць взагалі не кешується;
- 1 - 2,147,483,647 - інформація з таблиць кешується протягом зазначеного часу (у секундах).

Практикою встановлено, що установки за замовчуванням параметрів *SQLQRYMODE, SQLPASSTHRU MODE, SCHEMA CACHE SIZE і SCHEMA CACHE TIME* забезпечують цілком прийнятний режим роботи з базою даних. Експериментувати з ними для оптимізації ефективності роботи з конкретною базою даних рекомендується після нагромадження деякого досвіду роботи з BDE.

Зупинимося докладніше на визначенні такого важливого параметра, як SERVER NAME. У ньому потрібно вказати не тільки ім'я сервера (на якому перебуває база даних) і повний шлях доступу до бази, але й мережний протокол. Розробники утиліти налаштування BDE не вважали потрібним виділяти протокол в окремий параметр, тому необхідно використати наступні вирази:

- для доступу по протоколу TCP/IP

```
IB_SERVER:PATH\DATABASE.GDB.
```

Наприклад, шлях до бази на Windows NT сервері буде виглядати в такий спосіб - mynt:c:\ib\base.gdb, а до бази на UNIX-сервері - myunix:/ib/base.gdb;

- для доступу по протоколу IPX/SPX-

```
IB_SERVER@PATH\DATABASE.GDB.
```

Наприклад: *mynw@sys:ib\base.gdb;*

- для доступу по протоколу NETBEUI-

```
\\IB_SERVER\PATH\DATABASE.GDB.
```

Наприклад: *\\mynt\c:\ib\First_base.gdb.*

У цих прикладах mynt - ім'я сервера Windows NT, myunix - ім'я сервера UNIX-системи, mynw - ім'я сервера Novell NetWare, sys - ім'я тому NetWare, ib - директорій, у якому перебуває база даних, First_base.gdb - ім'я бази даних InterBase. Для того щоб правильно вказати ім'я сервера Oracle, потрібно писати ім'я за правилами Oracle - перед ім'ям поставити @.

Власне для розробки баз даних у візуальному середовищі Borland існують різноманітні засоби. Одним з таких засобів - це утиліта Database Desktop , що поставляється з Delphi та C++Builder для інтерактивної роботи з таблицями різних форматів локальних баз даних - *Paradox* і *dBase*, а також SQL-серверних баз даних InterBase, Oracle, Informix, Sybase (з використанням SQL Links). Файл утиліти, що виконує, називається *DBD32.EXE*. Після старту *Database Desktop* виберіть команду меню *File/New/Table* для створення нової таблиці. Перед Вами з'явиться діалогове вікно вибору типу таблиці, як показано на Рис.9.6. Ви можете вибрати будь-який формат із запропонованого, включаючи різні версії того самого формату.

Після вибору типу таблиці *Database Desktop* надасть діалогове вікно, специфічне для кожного формату, у якому можна визначити поля таблиці і їхній тип.

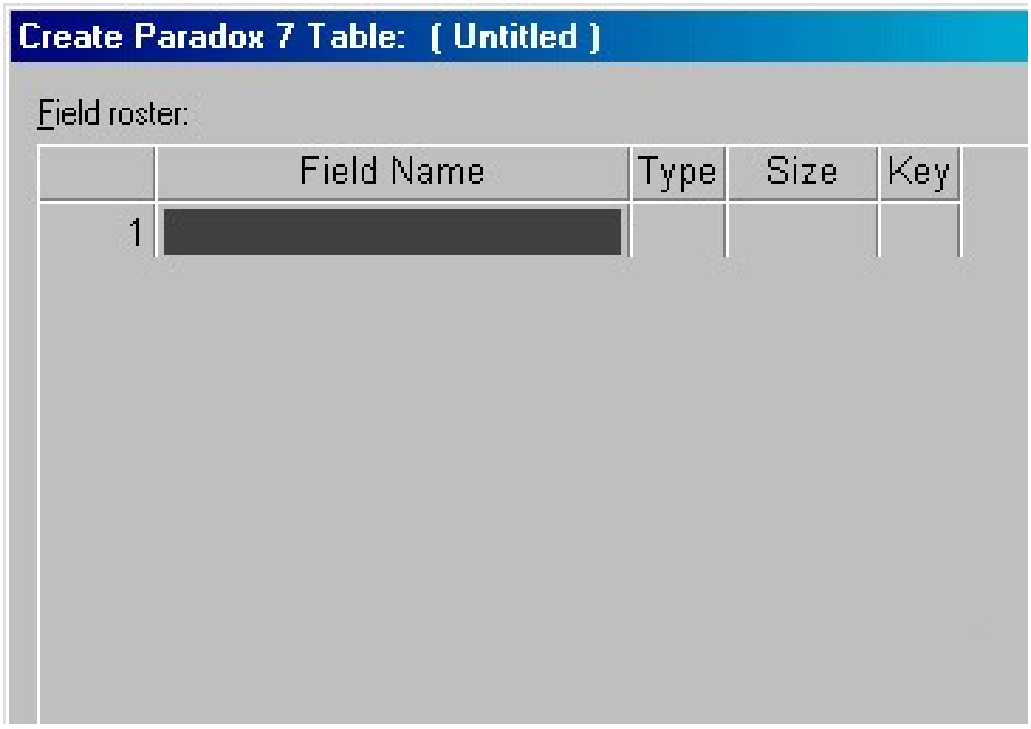


Рис. 9.6. Діалогове вікно вибору типу таблиці

Наприклад, ім'я поля в таблиці формату *Paradox* являє собою рядок, написання якого підлягає наступним правилам:

- Ім'я повинне бути не довше 25 символів.
- Ім'я не повинне починатися із пробілу, однак може містити пробіли. Однак, якщо планується в майбутньому переносити базу даних в інші формати,

розумніше буде уникати включення пробілів у назву поля. Фактично, з метою переносимості краще обмежитися дев'ятьма символами в назві поля, не включаючи в нього пробіли.

- Ім'я не повинне містити квадратні, круглі або фігурні дужки або тире, а також комбінацію символів “тире” і “більше” (->).
- Ім'я не повинне бути тільки символом #, хоча цей символ може бути присутнім в імені серед інших символів. Хоча Paradox підтримує крапку (.) у назві поля, краще її уникати, оскільки крапка зарезервована в Delphi та C++ для визначення операції.

Ім'я поля в таблиці формату *dBase* являє собою рядок, написання якої підлягає правилам, відмінним від Paradox:

- Ім'я повинне бути не довше 10 символів.
- Пропуски в імені неприпустимі.

Таким чином, бачимо, що написання імен полів у форматі dBase суттєво більш обмежені, ніж такі ж у форматі Paradox. Однак, якщо постають питання сумісності, то краще відразу закладати цю сумісність - давати полям імена, що підпорядковуються більш суворим правилам.

Вкажемо правила, яким підпорядковується написання імен полів у форматі ***InterBase***.

- Ім'я повинне бути не довше 31 символу.
- Ім'я повинне починатися з букв A-Z, a-z.
- Ім'я поля може містити букви (A-Z, a-z), цифри, знак $ і символ підкреслення _.
- Пробіли в імені неприпустимі.
- Для імен таблиць забороняється використати зарезервовані слова InterBase.

Наступний (після вибору імені поля) крок складається в завданні типу поля. Типи полів дуже сильно розрізняються друг від друга, залежно від формату таблиці. Для одержання списку типів полів перейдіть до колонки “Type”, а потім натисніть пробіл або клацніть правою кнопкою мишки.

Після цього для таблиць *Paradox* можемо визначити поля, що становлять первинний ключ, причому всі вони повинні бути на початку запису, а перше поле, що входить у ключ, повинне бути першим полем у записі. Для цього досить по ній двічі клацнути мишкою або нажати будь-яку клавішу.

Після створення таблиці, з нею можна зв'язати деякі властивості, перелік яких залежить від формату таблиці. Так, для таблиць формату Paradox можна визначити:

- *Validity Checks* (перевірка правильності) - ставиться до поля запису й визначає мінімальне й максимальне значення, а також значення за замовчуванням. Крім того, дозволяє визначити маску уведення
- *Table Lookup* (таблиця для "підглядання") - дозволяє вводити значення в таблицю, використовуючи вже існуюче значення в іншій таблиці
- *Secondary Indexes* (вторинні індекси) - дозволяють доступатися до даних в іншому порядку, ніж заданого первинним ключем
- *Referential Integrity* (посилальна цілісність) - дозволяє визначити зв'язки між таблицями й підтримувати ці зв'язки на рівні ядра. Звичайно визначається після створення всіх таблиць у базі даних
- *Password Security* (парольний захист) - дозволяє закрити таблицю паролем
- *Table Language* (мова таблиці) - дозволяє визначити для таблиці мовний драйвер.

У таблицях *dBase* не існує первинних ключів. Однак, цю обставину можна подолати шляхом визначення унікальних (*Unique*) і підтримуваних (*Maintained*) індексів (*Indexes*). Крім того, для таблиць dBase можна визначити й мову таблиці (*Table Language*) - мовний драйвер, керуючий сортуванням і відображенням символьних даних. Визначення додаткових властивостей таблиць всіх форматів доступні через кнопку "*Define*" (для таблиць InterBase дана кнопка називається "*Define Index...*" і дозволяє визначати тільки індекс, але не первинний ключ) у правій верхній частині вікна (група *Table Properties*). Причому, всі ці дії можна пробляти не тільки при створенні таблиці, але й тоді, коли вона вже існує. Для цього використовується команда *TableRestructure Table* (для відкритої в цей момент таблиці) або *UtilitiesRestructure* (з можливістю вибору таблиці). Однак, якщо Ви бажаєте змінити структуру або додати нові властивості для таблиці, що у цей момент уже використається іншим прикладною програмою, *Database Desktop* відмовить у цьому, оскільки дана операція вимагає монопольного доступу до таблиці. Натомість всі зроблені в структурі зміни відразу ж починають "працювати" - наприклад, якщо визначити посилальну цілісність для пари таблиць, то при спробі вставити в дочірню таблицю дані, що відсутні в батьківській таблиці, в Delphi генерується помилка.

Необхідно зазначити ще одну часто використовувану дуже корисну можливість *Database Desktop*. Створювати таблицю будь-якого формату можна не тільки "із чистого аркуша", але й шляхом копіювання структури вже існуючої таблиці. Для цього досить скористатися кнопкою "Borrow", наявної в лівому нижньому куті вікна. Діалогове вікно, що з'являється, дозволяє обрати існуючу таблицю й включити/виключити додаткові опції, що збігаються із уже перерахованими властивостями таблиць. Це найлегший спосіб створення таблиць. Однак можливості *Database Desktop* обмежуються розробкою своїх «рідних» СУБД

Paradox та *dBase*. Доступ до даних серверів SQL забезпечує окрема система драйверів - SQL Links. За їх допомогою в Delphi та C++Builder можна без особливих проблем розробляти прикладні програми для серверів Oracle, Informix, Sybase, DB2 і, природно, InterBase. Ці драйвери необхідно встановлювати додатково. При встановленні SQL Links для якогось SQL-сервера не потрібно вручну додавати драйвери для зв'язку з цим сервером - програма установки SQL Links зробить це автоматично. Відзначимо, що при встановленні Delphi *Client-Server* автоматично встановлюються також і *SQL Links*, так що окремо їх інсталяцію проводити не потрібно. Вам залишиться тільки підправити деякі параметри (наприклад, мовний драйвер) для встановлених драйверів. Але окрім «рідних» SQL Links, для доступу до SQL-серверів і деяких локальних баз даних можна використовувати ODBC-драйвери. Delphi та C++Builder використовує *Microsoft 2.0 ODBC Driver Manager*.

Рис. 9.7. Вікно Панелі управління з іконкою ODBC

Установка *BDE*-аліаса, що опирається на *ODBC*-драйвер і складається з трьох кроків. На першому кроці потрібно створити ODBC-аліас, на другому кроці - створити над ним надбудову у вигляді BDE-драйвера, а на третьому кроці - визначити стандартний BDE-аліас.

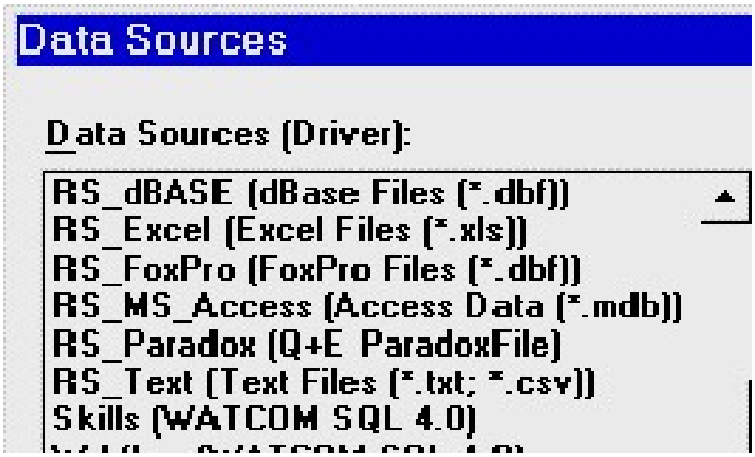


Рис. 9.8. Список ODBC-аліасів з вказівкою ODBC-драйверів

Установка *ODBC*-драйвера починається з перевірки того, які драйвери вже знаходяться в системі. Для цього в Панелі Управління (*Control Panel*) треба знайти ікону «ODBC» (Рис.9.7.), яка завантажує утиліту конфігурації ODBC. У її діалоговому вікні представлений список ODBC-аліасів (*Data Source*) з вказівкою в дужках ODBC-драйверів, на яких вони засновані (Рис.9.8.).

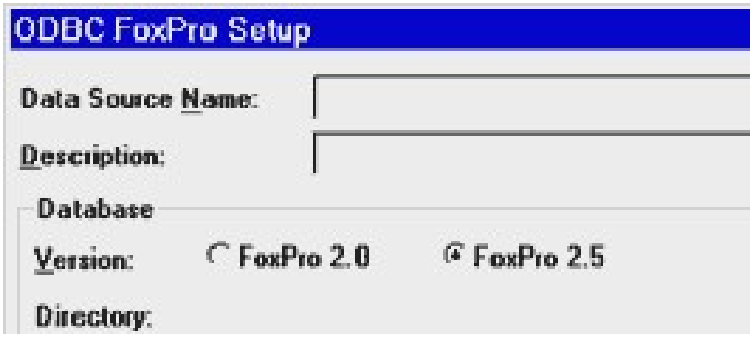


Рис. 9.9. Налаштування *FoxPro ODBC*-аліаса

Наприклад, для налаштування у діалоговому вікні FoxPro ODBC-аліаса потрібно призначити ім'я цього аліаса (*Data Source Name*) і вказати директорію, в якій знаходяться файли бази даних. Для створення нового ODBC-аліаса спочатку потрібно обрати ODBC-драйвер. Для цього налаштування треба натиснути кнопку «*Add*», наявну в правій частині діалогового вікна «*Data Sources*», зображеного на Рис. 9.8. Якщо в списку встановлених ODBC-драйверів не ознайдеться потрібного драйвера, треба повернутися до вікна «Data Sources» і встановити новий драйвер, вказавши його місцезнаходження (за допомогою кнопки «Drivers»). Після вибору ODBC-драйвера з'явиться діалогове вікно, вміст якого залежить від обраного драйвера (Рис.9.10.) і в якому можна провести налаштування ODBC-аліаса, визначивши його ім'я (Data Source Name). Після натиснення кнопки «OK» ODBC-аліас буде створений, і можна повернетися у вікно «Data Sources» (Рис. 9.8.).

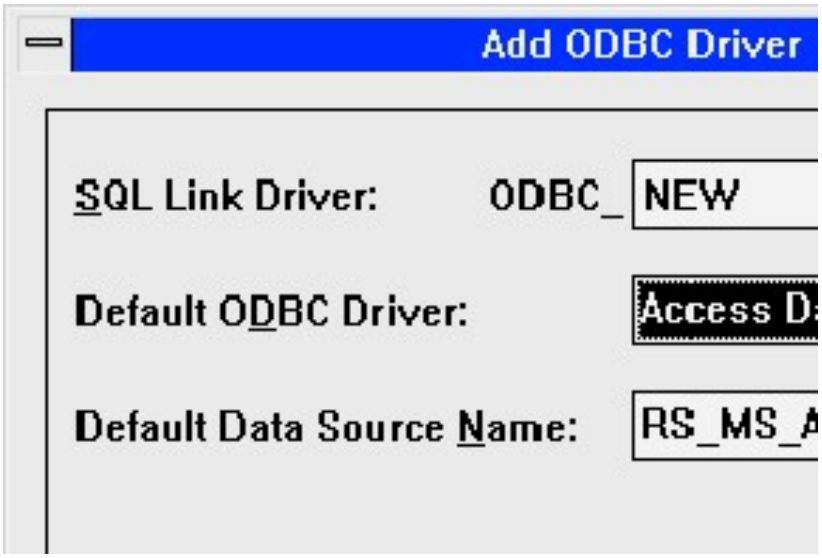


Рис. 9.10. Діалогове вікно «*Add ODBC Driver*» дозволяє встановити драйвери для Access, FoxPro і інших баз даних

Наступний крок полягає в створенні BDE-надбудови над ODBC-аліасом. Для цього треба переконатися, що в директорії *IDAPI* є файл *IDODBC01.DLL* - інакше потрібно наново встановити BDE. Після цього можна завантажити утиліту налаштування BDE. Назва BDE-драйвера, що використовує ODBC-аліас, повинна починатися з букв «ODBC_». Тому такі букви вже винесені перед назвою драйвера, так що їх не потрібно вводити. Введіть будь-яку назву драйвера і виберіть з випадаючих списків спочатку ODBC-драйвер, а потім - створений на його основі ODBC-аліас (Default Data Source Name). Таким чином, можна створити BDE-драйвер, що використовує ODBC-аліас. Після цього BDE-аліас створюється утілітою BDECFG.

Незважаючи на те, що через ODBC-драйвер можна успішно підключити локальні бази даних, зокрема фірми Microsoft і навіть власні *dBase* та *Paradox*, при практичному програмуванні треба враховуати доцільність тієї чи іншої схеми підключення. Надлишковість ODBC-драйверів для *Dbase* та *Paradox* очевидна, проте і для локальних баз Microsoft треба враховувати певну специфіку. Доступ до даних *FoxPro* здійснюється в першу чергу за допомогою BDE-драйвера прямого доступу, що дозволяє проводити запис у файли цієї СУБД. Крім цього можливий також доступ через *ODBC Link* і відповідний ODBC-драйвер. Доступ до даних *Visual FoxPro* здійснюється тільки за допомогою *ODBC Link* і відповідного ODBC-драйвера, бо BDE-драйвер для баз даних Visual FoxPro (*.vfp) наразі відсутній.

Для популярної *СУБД Access BDE*-драйвер прямого доступу розроблений для для версій *Access 95* і *Access 97.* Обидва ці драйвери працюють тільки у тому випадку, коли на комп'ютері, де експлуатується прикладна програма, що використовує їх, встановлена відповідна версія бібліотек *Microsoft Jet Engine* (вона входить в комплект поставки *Microsoft Access і Microsoft Visual FoxPro)*. Проте ці драйвери не можуть працювати з даними *Access* 2000 і вище. Для останніх версій Delphi і C++Builder доцільніше здійснювати доступ до даних Access за допомогою ADO і OLE DB, оскільки ці механізми надають в порівнянні з BDE значно більше функціональних можливостей. Деякі детальні аспекти використання BDE розглянемо дещо пізніше, а зараз прийшов час розібратися з новітніми технологіями обробки даних, реалізованих фірмою Borland.

На початку 2000-х в комп'ютерній пресі стали поширюватись матеріали про те, що в скорому мабутньому Borland перестане розвивати BDE і навіть відмовиться від їх поставок в складі Delphi та C++Builder. Для професіоналів такі повідомлення вже не були чимось несподіваним, проте тут можна поставити питання: як так сталося, що єдина системна технології обробки даних, яка могла на рівні конкурувати з ODBC втратила цінність в очах

самих розробників? Відповідь одна – стрімкий прогрес технології обробки даних, в першу чергу зусиллями Microsoft. З попередніх розділів ми знаємо яку грандіозну системну перебудову технології обробки даних провела ця фірма. Відтак і Borland не могла залишити це поза увагою, щоб не залишитись на узбіччі. Треба сказати, Borland зуміла дати адекватну відповідь на цей виклик.

9.3.Новітні технології обробки даних Borland

Зображена на Рис. 9.11. схема використання BDE прикладними програмами досить довгий час можна було називати схемою роботи з даними фірми Borland. Проте таке твердження втратило чинність з 1999 року після виходу версії Delphi 5.0. Сучасні уявлення Borland про технологію обробки відображує схема на Рис.

В ній можна побачити, що BDE є одним з компонентів загальної технології обробки даних.

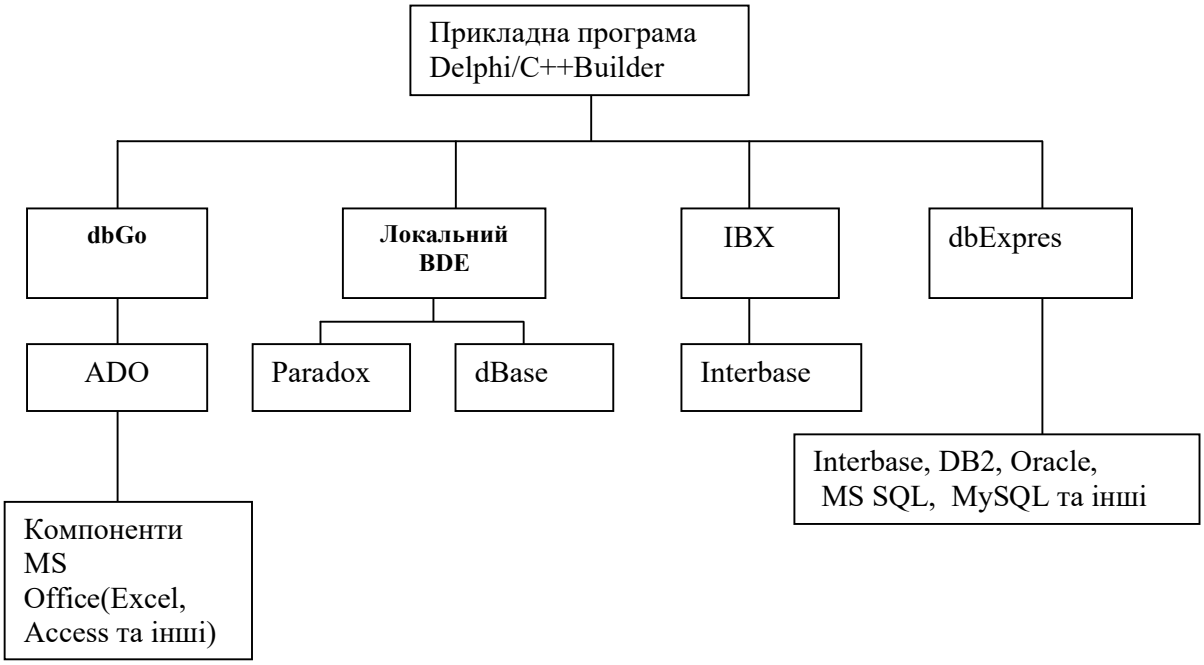


Рис. 9.11.Структура сучасної технології обробки даних Borland

Як можна бачити, машина баз даних BDE все ще виконує свої функції, проте її основна роль – в забезпеченні локальних баз даних, до того ж тільки з форматами *Paradox* та *dBase*. Натомість з'являється новий драйвер баз даних *dbExpress*, *InterBase* отримує можливість зв'язуватись напряму(в локальній версії), або через *dbExpress*, як і всі віддалені

сервери БД (Oracle, DB2, MS SQL та інші). З'явилася ще одна гілка: технологія *dbGo,* в якій Borland адаптує механізми *ADO* та *OLE*/*DB*.

9.3.1.Характеристика об'єктних компонент бібліотеки VCL для обробки даних

Зрозуміло, що для забезпечення розробки уніфікованої технології RAD у візуальному середовищі Delphi/C++Builder треба мати підтримку об'єктних бібліотек. Дійсно у пізніших версіях Delphi/C++Builder Borland значно розширила розділ бібліотеки VCL, пов'язаної з обробкою даних. Класи прикладних програм баз даних VCL-бібліотека представлені наступними групами:

- компоненти для доступу до даних, що реалізують:
- доступ через машину баз даних BDE (Borland Database Engine), що надає доступ через ODBC-драйвери або через внутрішні драйвери машини баз даних BDE (компоненти сторінки BDE-палітри інструментів);
- доступ через ADO-об'єкти (Active Data Objects), в основі якого лежить застосування технології OLE DB (компоненти сторінки ADO);
- доступ до локального або віддаленого SQL-сервера InterBase (компоненти сторінки InterBase);
- доступ за допомогою легковаговових драйверів dbExpress; доступ до БД при багатоланковій архітектурі (компоненти сторінки DataSnap);
- візуальні компоненти, що реалізують інтерфейс користувача;
- компоненти для зв'язку джерел даних з візуальними компонентами, що надають інтерфейс користувача;
- компоненти для візуального проектування звітів.

Основні механізми доступу до даних, підтримувані в Delphi/, є:

- *ODBC* - доступ через ODBC-драйвери БД або BDE-драйвери;
- *OLE DB* - доступ з використанням провайдерів даних (OLE DB - це метод доступу до будь-яких даних через стандартний COM-інтерфейс);
- засоби *dbExpress*, що використають полегшені драйвери БД;
- засоби доступу до розподілених наборів даних у багатоланковій архітектурі.

Найпростіший механізм керування даними, що використають ODBC-драйвери, може бути реалізований за наступною схемою:

У модуль даних (або у форму) додається компонент набору даних (об'єкт класу *TDataSet*) і встановлюється зв'язок із джерелом даних, обумовлене властивістю *DatabaseName*. Зв'язок може бути зазначена одним із трьох способів: по імені бази даних, каталогу або псевдоніму (спосіб вказівки зв'язки може бути обмежений типом джерела даних). Список всіх псевдонімів доступний на етапі проектування.

У модуль даних (або у форму) додається компонент джерела даних (*TDataSourse*), що є центральною сполучною ланкою між набором даних і елементами керування, що відображають ці дані. Властивість *DataSet* компонента типу *TDataSourse* указує набір даних, формована компонентами таких класів як *TTable* або *TQuery*. Якщо компоненти набору даних і джерела даних розташовані в модулі даних, то їх варто додати в проект (команда меню `File / Use unit`). У форму додаються елементи керування для роботи з даними, такі як *TDBGrid*, *TDBEdit*, *TDBCheckbox*. Вони зв'язуються з компонентом джерелом даних, що вказується властивістю *DataSource***.** Ім'я поля набору даних визначається властивістю *DataField***.**

Предком всіх класів наборів даних є клас *TDataSet***.** Він визначає основу структури всіх наборів даних - масив компонентів типу *TField* (кожний елемент масиву відповідає колонці таблиці). Набір даних - це впорядкована послідовність рядків, добутих із джерела даних. Кожний рядок набору даних складається з полів, вказаних у властивостях класу. Залежно від механізму доступу, використовуваного прикладною програмою, базовими класами набору даних можуть бути:

- *TTable, TQuery, TStoredProc* – для одноланкових або дволанкових прикладних програм, що використовують машину баз даних BDE. Клас *TQuery* `додатково дозволяє виконувати параметричні запити:`
- *TClientDataSet* - для реалізації клієнтського набору даних і для багатоланкової архітектури, що використає розподілений доступ;
- `TADODataSet` - для прикладних програм, що використають ADO-об'єкти;
- `TSQLDataSet` - для доступу до бази даних за допомогою `dbExpress`. Цей клас реалізує спрямований набір даних, що функціонує за принципом курсору. Для такого набору даних не створюється кеш пам'яті на клієнті, і серед методів доступу можливі тільки методи `Next` і `First`. Редагування записів у спрямованому наборі даних можливо тільки явним виконанням SQL-оператора `UPDATE` або при встановленні з'єднання із клієнтським набором даних через провайдера;

TSQLTable і *TSQLQuery* - для доступу до бази даних за допомогою *dbExpress*.

Окремо треба сказати про автономний компонент *ClientDataSet*, який відображується на значення локального файла специфічної структури. На відміну від традиційного способу читання з локального файлу по одному запису, компонент відображує в пам’ять всю таблицю, яка записана у файлі та її структуру. Щоб відобразити компонент *ClientDataSet* на локальний файл, треба налаштувати властивість *FileName* цього компонента. Компонет *ClientDataSet* підтримує два формати файлів: власний формат *CDS* та формат *XML*. Обидва формати мають спільні риси, дуже важливі для обробки даних: універсальність та кросплатформенність. Різниця полягає в тому, що формат *CDS* фактично не відрізняється від

формату представлення даних в оперативній пам'яті, який використовується компонентом *ClientDataSet.* Формат *XML* – це текстовий формат, хоча і специфічний.

На наступній схемі приведена ієрархія класів наборів даних бібліотеки VCL:

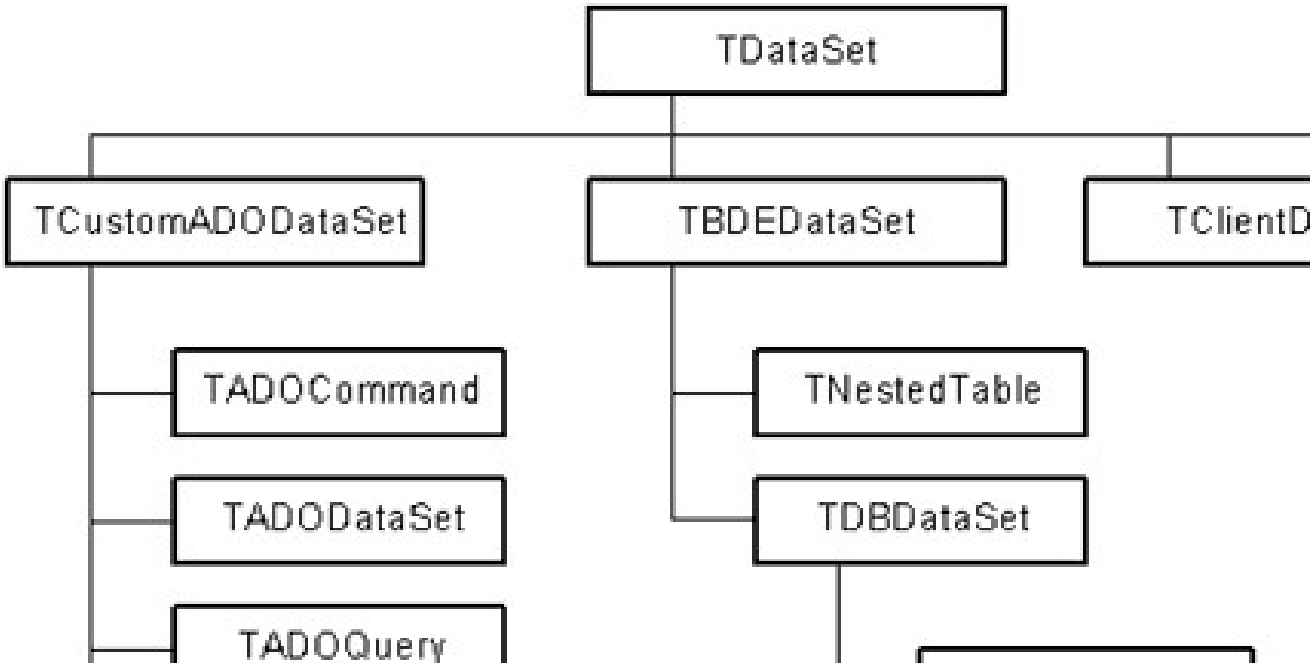


Рис. 9.12.Ієрархія класів VCL для обробки даних

Для визначення набору даних необхідно визначити наступні властивості:

- для класу *TTable* - значення властивостей *DatabaseName* і *TableName*;
- для класу *TQuery* - значення властивості SQL і, можливо, властивості *DatabaseName*.

Для того щоб читати дані з таблиць або записувати їх у таблиці, набір даних попередньо повинен бути відкритий. Відкрити набір даних можна одним з наступних способів:

- установити значення властивості *Active* набору даних рівним *True* під час виконання прикладних програм (наприклад, *Table1.Active:= True;*) або в режимі проектування в інспекторі об'єктів;
- викликати метод *Open* (наприклад, *Table1.Open;*).

Аналогічно, закрити набір даних можна викликом методу *Close* або встановивши значення властивості *Active* рівним *False*. Для компонента типу *TQuery* метод *Open* може бути виконаний тільки для закритого набору даних: спроба відкрити вже відкритий набір даних ініціює помилку.

Відкриття набору даних спричиняє:

- ініціацію подій *BeforeOpen* і *AfterOpen*;
- встановлення стану набору даних в *dsBrowse;*
- відкриття курсору для набору даних.

Якщо в момент відкриття набору даних відбулася помилка, то стан набору даних установлюється в *dsInactive,* а курсор закривається.

При роботі з компонентами наборів даних можна обійтися без явного використання компонентів, що реалізують з'єднання з базою даних. Однак деякі можливості, такі як керування транзакціями або кешовані відновлення, неможливі без компонентів типу *TDatabase* або *TADOConnection*. Компонент «база даних» *TDatabase* застосовується для з'єднання із джерелом даних через драйвери BDE або зовнішні ODBC-драйвери. Компонент *TADOConnection* використається для створення об'єкта «з'єднання» при доступі через OLE DB, що інкапсулюється за допомогою ADO-об'єктів VCL-бібліотеки.

За замовчуванням при переході від одного запису набору даних до іншої відбувається запис всіх зроблених змін у базу даних. Для того щоб можна було скасовувати зроблені зміни або виконувати відновлення декількох записів, застосовують кешовані відновлення. Вони дозволяють значно знизити мережевий трафік за рахунок того, що всі зроблені зміни зберігаються у внутрішньому кеші й при переході від одного запису до інший інформація в базу даних не передається. Щоб включити режим кешованого відновлення, треба встановити значення властивості *CachedUpdates*, що дорівнює *True* для компонента набору даних. Для присвоєння кешованого відновлення викликається метод *ApplyUpdates*, а для скасування - *CancelUpdates*. Давайте дещо детальніше розглянемо новітні технології Borland.

9.3.2. dbExpress

Однією із проблем різних технологій доступу до даних, використовуваним у прикладних програмах Delphi, є труднощі поширення готових прикладних програм. Для BDE потрібна окрема установка, що займає близько 15 Мбайт дискового простору, а також спеціальне налаштування псевдонімів. Драйвери *dbExpress* використовують для одержання даних винятково запити SQL. При цьому на боці клієнта відсутнє кешуваннння даних, внаслідок цього тут застосовуються винятково односпрямовані курсори й відсутня можливість прямого редагування наборів даних.

Для функціонування компонентів *dbExpress* необхідний тільки один драйвер, що взаємодіє прямо із клієнтським програмним забезпеченням для обраного сервера БД. У поставку входять драйвери для перших чотирьох зі списку серверів баз даних:

- DB2;
- InterBase;
- MySQL;
- Oracle;
- Microsoft SQL Server 2000.

Драйвери реалізовані у вигляді динамічних бібліотек, а при необхідності можуть бути прикомпільовані безпосередньо до файлу прикладної програми. Тому проблема поширення разом з прикладною програмою засобів доступу до даних у випадку з *dbExpress* знімається

повністю. Природно, на комп'ютері повинне бути встановлене клієнтське ПЗ відповідного SQL сервера. Крім того, технологія *dbExpress* забезпечує доступ до даних у кросплатформних програмах для Windows і Linux, тому що застосовується як в Delphi та C++Builder, так і в Kylix , а способи її застосування ідентичні.

Таким чином, технологія *dbExpress* є найкращим розв'язанням для програм, для яких є необхідним швидкий і легкий перегляд даних серверів SQL. Але для складних клієнт-серверних або багаторівневих прикладних програм, що забезпечують серйозну роботу з даними, краще застосовувати інші технології.

Технологія *dbExpress* являє собою сукупність драйверів, компонентів, що інкапсулюють з'єднання, транзакції, запити й набори даних, а також інтерфейсів, що забезпечують універсальний доступ до функцій *dbExpress.*

Компоненти *dbExpress* розташовуються в Палітрі компонентів на однойменній сторінці. Технологія *dbExpress* забезпечує доступ до сервера баз даних за допомогою драйвера, реалізованого як динамічна бібліотека. Для кожного сервера є своя динамічна бібліотека.

Таблиця 9.2. Драйвери *dbExpress*

Сервер БД	**Драйвер**	**Клієнтське ПЗ**
DB2	*Dbexpdb2.dll.*	*Db2cli.dll*
InterBase	*Dbexpint.dll*	*GDS32.DLL*
Informix	*Dbexpinf.dll*	*Isqlb09a.dll*
Microsoft SQL Server 2000	*Dbexpmss.dll*	*OLE DB*
MySQL	*Dbexpmys.dll*	*LIBMYSQL.DLL*
Oracle	*Dbexpora.dll*	*OCI.DLL*

Перераховані в табл. файли перебувають у папці \Delphi7\Bin або CBuilder6\Bin

Для доступу до даних сервера драйвер має бути встановлений на комп'ютері клієнта. Для доступу до даних драйвер взаємодіє із клієнтським ПЗ сервера, що також повинне бути інстальоване на клієнтському боці. Стандартні налаштування для кожного драйвера зберігаються у файлі *\Borland Shared\DBExpress\dbxdrivers.ini.*

Архітектура provider/resolver використовує чотири компоненти для надання даних й їхнього редагування. Перший компонент - *SQLConnection* - призначений для встановлення з'єднання між драйвером dbExpress і використовуваним сервером БД. Далі йдуть компоненти, які надають доступ до даних, одержуваним оператором SELECT або викликом збережених процедур. Третій компонент - *DataSetProvider*, і четвертий - *ClientDataSet*. Коли ви відкриваєте *ClientDataSet*, він запитує дані в *DataSetProvider. DataSetProvider* відкриває

компонент, що виконує запит або збережену процедуру, вибирає дані, закриває цей компонент, і поставляє дані (і необхідні метадані) компоненту *ClientDataSet*.

ClientDataSet зберігає дані в пам'яті, поки вони проглядаються й модифікуються. При додаванні, видаленні або відновленні запису, у коді або через користувальницький інтерфейс, компонент *ClientDataSet* запам'ятовує ці операції в пам'яті. Для відновлення бази даних потрібно викликати метод *ClientDataSet.ApplyUpdates*. *ApplyUpdates* передає зміни компоненту *DataSetProvider*. Провайдер починає транзакцію, потім створює й виконує оператори SQL, що відповідають проведеними операціям над даними *ClientDataSet.* Якщо всі оператори SQL були виконані успішно, провайдер завершує транзакцію по *commit*; якщо ні - скасовує транзакцію по *rollback*. Зміни в базі даних можуть не пройти, наприклад, якщо зміни порушують правила контролю даних, або якщо інший користувач уже модифікував ці дані певним чином. При виникненні помилки транзакція скасовується по *rollback*, і викликається подія *ClientDataSet.OnReconcileError*, надаючи вам можливість обробки помилок.

В чому переваги архітектури *provider/resolver*? Вони полягають у наступному.

А)Короткий час життя транзакцій. Довгі транзакції змушують сервер БД утримувати блокування, які знижують можливості багатокористовачевої обробки даних і перевантажують ресурси сервера. При архітектурі *provider/resolver*, транзакція існує тільки в той момент, коли застосовуються відновлення. Це істотно знижує вимоги до ресурсів і зменшує ймовірність блокувань, особливо при великій кількості користувачів сервера БД.

Б)Можливість редагування будь-яких записів. Записи, що повертають багататабличними вибірками, збереженими процедурами або *View* не можуть бути змінені прямо. Можна вказати за допомогою властивості *ProviderFlags* в об'єкті *TField,* які колонки повинні обновлятися, і в події *DataSetProvider.* Властивість *OnGetTableName* визначає яка саме таблиця повинна обновлятися. При цьому більшість даних, що не редагуються стануть редаговані.

В)Швидке сортування й пошук. Оскільки *ClientDataSet* зберігає записи у пам'яті, вони можуть бути швидко відсортовані. Якщо сортування в пам'яті працює повільно, ви можете створити індекси над даними *ClientDataSet* або під час розробки, або під час виконання прикладної програми. Ці індекси в пам'яті допоможуть міняти порядок записів або шукати записи дуже швидко, без використання індексів у базі даних.

Г)Автоматичне агрегування. ClientDataSet може виконувати автоматично обумовлені вами складні обчислення, такі як Sum(Price) -Sum(Cost). Ви можете групувати обчислення сум по полю або комбінації полів. Ви також можете використати агрегати Min, Max, Count й Avg.

Д)Перегляд підмножини даних. Вирази фільтрації можуть використовувати синтаксис WHERE для відображення підмножини записів ClientDataSet, без необхідності конструювати запит на клієнті й відправляти його щораз на сервер.

Є)Множина видів даних. Можливість "клонувати" курсор ClientDataSet дозволяє переглядати ті самі дані різними способами, одночасно. Наприклад, можна переглядати ті самі дані, відсортовані по різних стовпцях.

Ж)Колонки, що обчислюють, на клієнті. Також ви можете додавати обчислюють колонки, що, до ClientDataSet під час розробки, для того щоб обчислюють колонки, що, стали частиною даних у пам'яті. Оскільки обчислення виробляються скомпільованим Delphi або C++ кодом, вони виконуються дуже швидко й можуть бути більше складними, ніж обчислення, зроблені на сервері в *COMPUTED BY* колонках.

З)Легкість поширення та створення драйвера. Прикладні програми, що використають *dbExpress,* для роботи вимагають дві DLL. Перша dll - драйвер *dbExpress*, наприклад *DBEXPINT.DLL* для *Interbase*, і друга - *MIDAS.DLL*, бібліотека підтримки *ClientDataSet*. Разом ці дві DLL займають менш ніж 500 кілобайт. Це мінімізує розмір прикладної програми й спрощує його установку. Якщо ви не хочете поширювати ці DLL, ви можете скомпілювати їх прямо в EXE додатки. Поширення на Linux те ж саме, за винятком того, що бібліотеки мають розширення .so, а не .dll. Драйвери dbExpress повинні реалізовувати п'ять інтерфейсів, які описані в онлайн-довідці. Borland також поставляє вихідний текст драйвера для MySQL як приклад. Це істотно знижує витрати виробників СКБД на створення високопродуктивних драйверів. Ви можете створити свій власний драйвер, якщо ви працюєте з незвичайною або застарілою базою даних, для якої немає доступних комерційних драйверів.

Для створення з'єднання із сервером у рамках технології *dbExpress* програма повинна використати компонент TSQLConnection. Це обов'язковий компонент, всі інші компоненти пов'язані з ним і використовують його для одержання даних. Після переносу цього компонента в модуль даних або на форму необхідно вибрати тип сервера й налаштувати параметри з'єднання. Властивість *property ConnectionName: string;* дозволяє вибрати зі списку, що випадає, конкретне налаштовне з'єднання. Після вибору з'єднання автоматично встановлюються значення властивостей:

- *property DriverName: string;*

визначає використовуваний драйвер;

- *property LibraryName: string*;

задає динамічну бібліотеку драйвера dbExpress;

- *property VendorLib: string;*

визначає динамічну бібліотеку клієнтського ПЗ сервера ;

- *property Params: TStrings;*

список цієї властивості містить налаштування для обраного з'єднання.

При необхідності всі перераховані властивості можна встановити додатково.

Розробник може доповнювати й змінювати список налаштованих з'єднань. Для цього використовується спеціалізований Редактор з'єднань *dbExpress Connections* (Рис.9.13.). Він відкривається після подвійного клацання на компоненті *TSQLConnection* або вибору команди *Edit Connection Properties* зі спливаючого меню компонента.

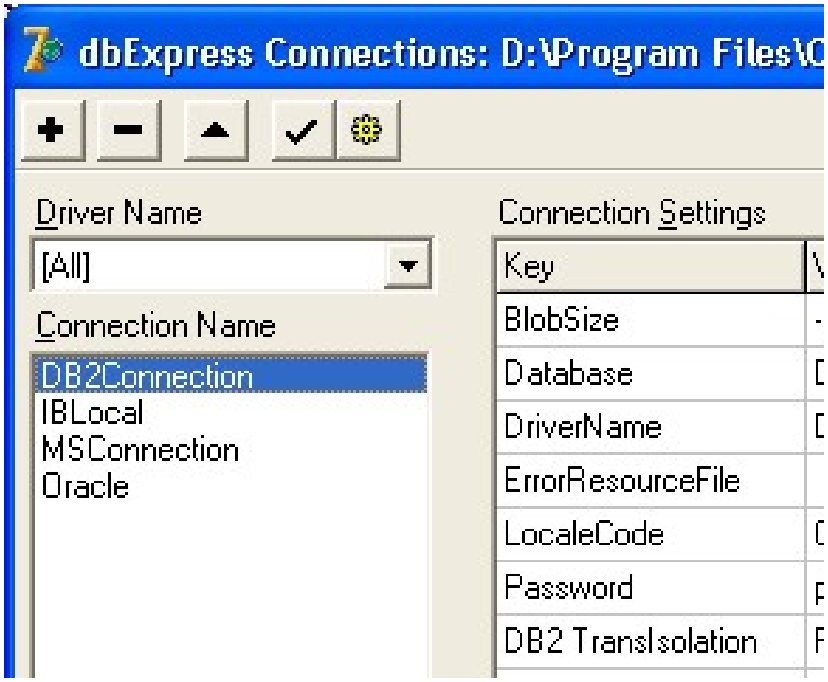


Рис. 9.13. Вікно редактора налагоджених з'єднань компонента TSQLConnection

У списку ліворуч розташовуються існуючі з'єднання. У правій частині для обраного з'єднання відображаються поточні налаштування. За допомогою кнопок на Панелі інструментів можна створювати, перейменовувати й видаляти з'єднання. Налаштування також можна редагувати. Список з'єднань у лівій частині відповідає списку вибору властивості *ConnectionName* в Інспекторі об'єктів. Налаштування з'єднання із правої частини відображуються у властивості *Params*. Налаштовані з'єднання і їхні параметри зберігаються у файлі *\Borland Shared\DBExpress\dbxconnections.ini*. Конкретні значення налаштувань з'єднань залежать від використовуваного сервера БД і вимог прикладної програми (табл. 9.3.).

Таблиця 9.3.Налаштування з'єднання dbExpress

Параметр	**Значення**
Загальні Налаштування	
BlobSize	Визначає обмеження на обсяг пакета даних для даних BLOB
DriverName	Ім'я драйвера
ErrorResourceFile	Файл повідомлень про помилки
LocaleCode	Код локалізації, що визначає вплив національних символів на сортування даних
User Name	Ім'я користувача
Password	Пароль
DB2	
Database	Ім'я клієнтського ядра
DB2 Translsolation	Рівень ізоляції транзакцій
Informix	
HostName	Ім'я комп'ютера, на якому працює сервер Informix
Informix Translsolation	Рівень ізоляції транзакції
Trim Char	Визначає, потрібно чи видаляти зі рядкових значень полів пробіли, що доповнюють значення до повного рядка, заданої розміром поля даних
Interbase	
CommitRetain	Задає поведінку курсора після завершення транзакції. При значенні True курсор обновляється, інакше - видаляється
Database	Ім'я файлу бази даних (файл GDB)
InterBase	Рівень ізоляції транзакції

Translsolation	
RoleName	Роль користувача
SQLDialect	Використовуваний діалект SQL. Для IntreBase 5 можливо тільки одне значення -1
Trim Char	Визначає, чи потрібно видаляти із рядкових значень полів пробіли, що доповнюють значення до повного рядка, заданої розміром поля даних
WaitOnLocks	Дозвіл на очікування зайнятих ресурсів
Microsoft SQL Server 2000	
Database	Ім'я бази даних
HostName	Ім'я комп'ютера, на якому працює сервер MS SQL Server 2000
MSSQL Translsolation	Рівень ізоляції транзакції
OS Autenification	Використання облікового запису поточного користувача операційної системи (домена або Active Directory) при доступі до ресурсів сервера
MySQL	
Database	Ім'я бази даних
HostName	Ім'я комп'ютера, на якому працює сервер MySQL
Oracle	
AutoCommit	Прапор завершення транзакції. Установлюється тільки сервером
BlockingMode	Задає режим завершення запиту. При значенні True з'єднання чекає закінчення запиту (синхронний режим), інакше - починає виконання наступні (асинхронний режим)
Database	Запис бази даних у файлі TNSNames.ora
Multiple Transaction	Підтримка керуванням декількома транзакціями в одній сесії
Oracle Transisolation	Рівень ізоляції транзакцій
OS Autenification	Використання облікового запису поточного користувача операційної системи (домена або Active Directory) при доступі до ресурсів сервера

Trim Char	Визначає, потрібно чи видаляти зі рядкових значень полів пробіли, що доповнюють значення до повного рядка, заданої розміром поля даних

Після вибору налаштовного з'єднання або вибору типу сервера й налаштування параметрів з'єднання компонент *TSQLConnection* готовий до роботи.

Властивість *property Connected: Boolean;* відкриває з'єднання із сервером при значенні *True*. Аналогічну операцію виконує метод *procedure Open;*. Після відкриття з'єднання всі компоненти *dbExpress,* що інкапсулюють набори даних і пов'язані з відкритим компонентом *TSQLConnection,* одержують доступ до бази даних. З'єднання закривається тією ж властивістю *connected* або методом : *procedure Close;*. При відкритті й закритті з'єднання розробник може використати обробникі подій : *property BeforeConnect: TNotifyEvent;property AfterConnect: TNotifyEvent; property BeforeDisconnect: TNotifyEvent; property AfterDisconnect: TNotifyEvent;*

Наприклад, на боці клієнта можна організувати перевірку програми користувача :

```
procedure TForml.MyConnectionBeforeConnect(Sender: TObject);
begin
if MyConnection.Params.Values['User_Name']) <> DefaultUser then
begin
MessageDlg('Wrong user name', mtError, [mbOK], 0);
Abort;
end;
end;
```

Властивість *property LoginPrompt: Boolean;* визначає, потрібно чи відображати діалог авторизації користувача перед відкриттям з'єднання. Про поточний стан з'єднання можна судити за значенням властивості *TConnectionState = (csStateClosed, csStateOpen, csStateConnecting, csStateExecuting, csStateFetching, csStateDisconnecting); property ConnectionState: TConnectionState;*. Параметри з'єднання можна налаштовувати на етапі розробки в Інспекторі об'єктів або Редакторі з'єднань . Також це можна зробити й безпосередньо перед відкриттям з'єднання, використовуючи властивість *Params* або метод *procedure LoadParamsFromlniFile(AFileName : String = '');,* який завантажує заздалегідь підготовлені параметри з INI-файлу. Перевірити успішність цієї операції можна за допомогою властивості *property Params Loaded: Boolean*;, значення *True* якого сигналізує про успіх завантаження.

```
procedure TForml.StartBtnClickfSender: TObject);
 begin
if MyConnection.Params.Values['DriverName'] = « then
```

MyConnection.LoadParamsFromlniFile

('c:\Temp\dbxalarmconnections.ini');

if MyConnection.ParamsLoaded then

try MyConnection.Open;

except MessageDlgt'Database connection error', mtError, [mbOK], 0);

end;

end;

Компонент *TSQLConnection* дозволяє виконувати деякі операції з підключеними наборами даних і стежити за їхнім станом. Властивість *property DataSetCount: Integer;* повертає число підключених через дане з'єднання наборів даних. Але це тільки активні набори даних, передані у зв'язані компоненти. Загальна кількість запитів, що виконуються в даний момент, повертає властивість *property ActiveStatements: LongWord;*. Якщо сервер БД встановив для даного з'єднання максимальну кількість одночасно виконуваних запитів, то воно доступно у властивості *property MaxStmtsPerConn: LongWord;*. Тому перед відкриттям набору даних можна виконувати наступний код, що підвищить надійність прикладної програми:

if MyQuery.SQLConnection.ActiveStatements <= MyQuery.SQLConnection.MaxStmtsPerConn

then MyQuery.Open

else MessageDlg ('Database connection is busy', mtWarning, [mbOK] , 0) ;

У випадку виникнення непередбаченої ситуації всі відкриті через дане з'єднання набори даних можна швидко закрити методом *procedure CloseDataSets;* без розриву з'єднання. Компонент *TSQLConnection* може самостійно виконувати запити SQL, не вдаючись до допомоги компонента *TSQLQuery* або *TSQLDataSet*. Для цього призначена функція :

function Execute(const SQL: string; Params: TParams;

ResultSet:Pointer=nil): Integer;

Якщо запит повинен містити параметри, то необхідно спочатку створити об'єкт - список параметрів TParams і заповнити його. При тому що об'єкт *TParams* ще не пов'язаний з конкретним запитом, важливий порядок проходження параметрів, який повинен збігатися в списку *TParams* і в тексті SQL. Якщо запит повертає результат, метод автоматично створює об'єкт типу *TCustomSQLDataSet* і повертає вказівник на нього в параметр *Resultset*. Функція повертає число оброблених запитом записів. Наступний фрагмент коду ілюструє застосування функції *Execute*.

procedure TForml.SendBtnClick(Sender: TObject);

var FParams: TParams;

FDataSet: TSQLDataSet;

begin

```
FParams := TParams.Create;
 try
FParams.Items[0].Aslnteger := 1234; FParams.Items[1].AsInteger := 6751;
MyConnection.Execute('SELECT * FROM Orders WHERE OrderNo >= :Ord AND
EmpNo = :Emp', FParams, FDataSet);
if Assigned(FDataSet) then
with FDataSet do
begin
Open;
while Not EOF do
begin
{...}
Next ;
end;
Close;
 end;
finally
FParams.Free;
end;
end;
```

Якщо запит не має параметрів і не повертає набір даних, можна використати функцію *function ExecuteDirect(const SQL:string):LongWord;* яка повертає нуль в випадку успішного виконання запиту або код помилки. Метод *procedure GetTableNames(List: TStrings; SystemTables: Boolean = False);*повертає список таблиць бази даних. Параметр *SystemTables* дозволяє включати до формованого списку *List* системні таблиці. Метод *GetTableNames* додатково управляється властивістю *TTableScope = (tsSynonym, tsSysTable, tsTable, tsView);*

TTableScopes = set of TTableScope; property TableScope: TTableScopes;, що дозволяє визначити тип таблиць, імена яких попадають у список. Для кожної таблиці можна одержати список полів, використавши метод *procedure GetFieldNames(const TableName: String; List: TStrings);* і список індексів за допомогою методу *procedure GetlndexNames(const TableName: string; List: TStrings);*. В обох методах список значень, що повертаються, міститься в параметр *List*. Подібним чином метод *procedure GetProcedureNames(List: TStrings);* повертає список доступних збережених процедур, а метод *procedure GetProcedureParams(ProcedureName:String; List:Tlist);* визначає параметри окремої процедури. Подібно своїм аналогам в BDE і ADO компонент *TSQLConnection* підтримує

механізм транзакцій і робить це схожим чином. Початок, фіксацію й відкочування транзакції виконують методи :

procedure StartTransaction(TransDesc: TTransactionDesc);

procedure Commit(TransDesc: TTransactionDesc);

procedure Rollback(TransDesc: TTransactionDesc);

При цьому запис TTransactionDesc повертає параметри транзакції:

TTransIsolationLevel = (xilDIRTYREAD, xilREADCOMMITTED, xilREPEATABLEREAD, xilCUSTOM);

TTransactionDesc = packed record

TransactionID : LongWord;

Globall : LongWord;

IsolationLevel : TTransIsolationLevel;

Customlsolation : LongWord;

end;

Запис містить унікальний у рамках з'єднання ідентифікатор транзакції *TransactionID* та рівень ізоляції Транзакції *IsolationLevel.* При рівні ізоляції *xilCustom* визначається параметр *Customlsolation.* Ідентифікатор GlobalID використається при роботі із сервером *Oracle.* Деякі сервери БД не підтримують транзакції, і для визначення цього факту використовується властивість :

property TransactionsSupported: LongBool;

Якщо з'єднання вже перебуває в транзакції, властивості *property InTransaction: Boolean;* привласнюється значення *True*. Оскільки сервер не підтримує множинні транзакції, завжди корисно переконатися, що з'єднання не обслуговує початку транзакцію:

```
var Translnfo: TTransactionDesc;
(...)
if Not MyConnection.InTransaction then
try
MyConnection.StartTransaction(Translnfo); {...}
MyConnection.Commit(Translnfo);
except
MyConnection.Rollback(Translnfo);
end;
```

Набір компонентів dbExpress, що інкапсулюють набір даних, досить звичайний і порівняний з аналогічними компонентами *BDE, ADO, InterBase Express***.** Це компоненти *TSQLDataSet, TSQLTable, TSQLQuery, TSQLStoredProc.* Хоча загальним предком всіх розглянутих тут компонентів є клас *TDataSet* (див. Рис. 9.14.), що має повний інструментарій

для роботи з набором даних, компоненти dbExpress використають тільки односпрямовані курсори й не дозволяють редагувати дані. Односпрямовані курсори обмежують навігацію по набору даних і забезпечують переміщення тільки на наступний запис і повернення на перший. Також тут недоступні будь-які операції, що вимагають буферизації даних - це пошук, фільтрація, синхронний перегляд.

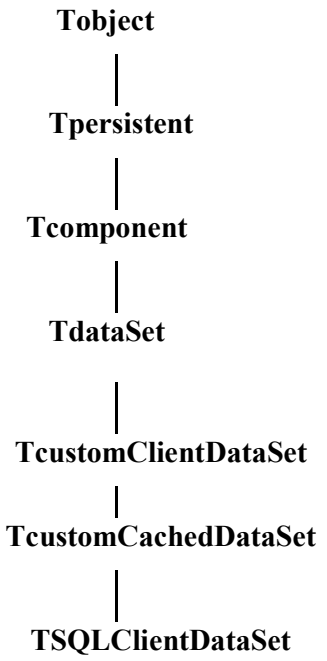


Рис. 9.14.Ієрархія компонент для наборів даних *dbExpress*

Таким чином, для вимикання ряду механізмів класу TDataSet знадобився ще один проміжний клас *TCustomSQLDataSet.*

Відображення даних за допомогою компонентів зі сторінки *Data Controls* також обмежено. Не можна використати компоненти *TDBGrid і TDBCtrlGrid*, а в компоненті *TDBNavigator* не забудьте відключити кнопки повернення на одну позицію назад і переходу на останній запис, а також компоненти синхронного перегляду. Інші компоненти можна використати звичайним способом. Доступ до даних у всіх розглянутих компонентах здійснюється однаково - через з'єднання, інкапсульоване компонентом *TSQLConnection*. Прив'язування до з'єднання виконує властивість *property SQLConnection: TSQLConnection;*

Готова програма, що використовує технологію *dbExpress,* може поставлятися двома способами. Разом з прикладною програмою поставляється динамічна бібліотека для обраного сервера. Вона перебуває в директорії *\Delphi7\Bin*.

Якщо в прикладній програмі використається компонент *TSimpleDataSet,* необхідно включити в поставку динамічну бібліотеку Midas.dll. Програма компілюється разом з наступними DCU-файлами: *dbExpInt.dcu, dbExpOra.dcu, dbExpDb2.dcu, dbExpMy.dcu* (залежно від обраного сервера). Якщо в прикладній програмі використається компонент

TSimpleDataSet, варто додати файли *Crtl.dcu* і *MidasLib.dcu.* У результаті необхідно поставляти файл, що виконує тільки програму. Якщо додаткове налаштування з'єднань не потрібне, то і файл dbxconnections.ini не потрібний.

У таблиці наведене порівняння BDE й dbExpress по ряду параметрів, для того щоб привести список відмінностей. Друга таблиця показує компоненти dbExpress, що відповідають компонентам BDE.

Таблиця 9.4.Порівняння BDE та dbExpress

Функціональність	**BDE**	**dbExpress**
буферизація записів	BDE сам визначає кількість записів, збережених у пам'яті	Ви можете управляти кількістю записів, збережених у пам'яті
контроль мережевого трафіка	BDE сам визначає, скільки записів вибирати із сервера	Ви управляєте як кількістю, так і моментом вибірки записів із сервера
керування транзакціями	Є автоматичний режим і ручне керування транзакціями. Транзакції активні в момент редагування даних.	Є автоматичний режим і ручне керування транзакціями. Транзакції активні в момент передачі змін на сервер.
поширення прикладних програм	Велика кількість файлів (обсягом близько 9 мегабайт). Установка й конфігурування достатнє складні й вимагають редагування registry.	Для поширення потрібно дві dll, що займають менш 500 кілобайт, які можуть бути вкомпильовані в EXE. Конфігурація зберігається в dbxconnections.ini й dbxdrivers.ini
кросс-платформеність	Тільки Windows	Windows й Linux
Драйвери від третіх фірм	Складно розробляти, практично немає	Легко розробляти. Доступно багато різних драйверів. За інформацією із драйверів звертайтеся на bdn.borland.com

Наступна таблиця показує компоненти dbExpress, які відповідають компонентам BDE. Зверніть увагу, що в dbExpress немає аналога компоненту BatchMove з BDE.

Таблиця 9.5. Порівняння компонент BDE та dbExpress.

BDE	**dbExpress**

TDatabase	TSQLConnection
TQuery	TSQLQuery
TStoredProc	TSQLStoredProc
TTable	TSQLTable
немає аналога	TSQLDataSet
TBatchMove	немає аналога
утиліта SQL Monitor	TSQLMonitor
TSession	немає
TUpdateSQL	немає
NestedDataSet	немає
BDEClientDataSet	SimpleDataSet

Механізм TSession відсутній в dbExpress, і як такий в dbExpress не потрібний. TUpdateSQL використається тільки при включенні CachedUpdates компонента TQuery. в dbExpress, можливості ClientDataSet заміняють cached updates. Можливість обробки вкладених наборів даних убудована в DataSetProvider й ClientDataSet.

9.3.3. Використання технології ADO в середовищі Delphi/C++Builder

Фірма Borland застосувала технологію *Microsoft Active Data Objects*, що забезпечує універсальний доступ до джерел даних з прикладних програм БД на базі загальної моделі об'єктів *COM*. В сучасних версіях *Delphi/C++Builder* офіційно ця технологія іменується dbGO, проте компонети візуального середовища зосереджені на сторінці *ADO*.

Технологія ADO і інтерфейси *OLE DB* забезпечують для прикладних програм єдиний спосіб доступу до джерел даних різних типів (Рис. 9.15.). Наприклад, прикладна програма, що використовує *ADO*, може застосовувати однаково складні операції й до даних, що зберігається на корпоративному сервері SQL, і до електронних таблиць, і локальних СУБД. Запит SQL, спрямований будь-якому джерелу даних через *ADO*, буде виконаний.

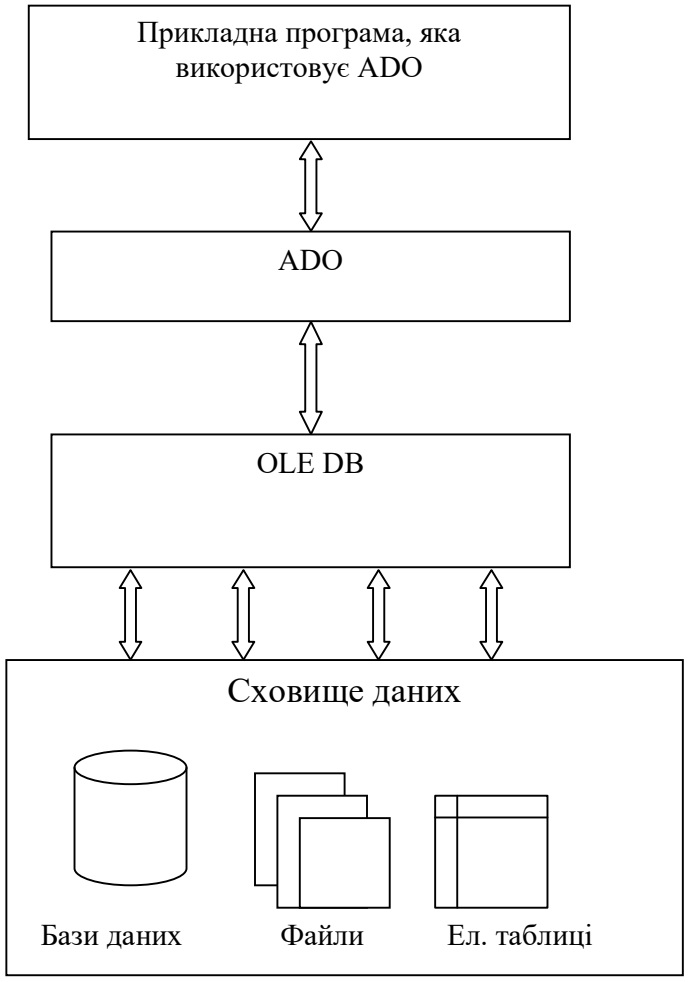


Рис. 9.15.Схема доступу до даних через ADO

Відповідно до термінології ADO, будь-яке джерело даних (база даних, електронна таблиця, файл) називається сховищем даних, з яким за допомогою провайдера даних взаємодіє прикладна програма. Мінімальний набір компонентів прикладної програми може включати об'єкт з'єднання, об'єкт набору даних, об'єкт процесора запитів. Кожному об'єкту відповідає ідентифікатор класу CLSID, що зберігається в системному реєстрі. Для створення об'єкта використовується метод CoCreateinstance і відповідна фабрика класу. Об'єкту відповідає набір інтерфейсів, до методів яких можна звертатися після створення об'єкта. В результаті прикладна програма звертається не прямо до джерела даних, а до об'єкта OLE DB, який «вміє» представити дані (наприклад, з файлу електронної пошти) у вигляді таблиці БД або результату виконання запиту SQL.

Технологія ADO у цілому містить у собі не тільки самі об'єкти OLE DB, але й механізми, що забезпечують взаємодію об'єктів з даними й прикладними програмами. На цьому рівні найважливішу роль грають провайдери ADO, що координують роботу прикладних програм зі сховищами даних різних типів. Така архітектура дозволяє зробити

набір об'єктів і інтерфейсів відкритим і розширюваним. Набір об'єктів і відповідний провайдер може бути створений для будь-якого сховища даних без внесення змін у вихідну структуру ADO. При цьому істотно розширюється саме поняття даних - адже можна розробити набір об'єктів і інтерфейсів і для нетрадиційних табличних даних. Наприклад, це можуть бути графічні дані, деревоподібні структури із системних реєстрів, дані CASE-інструментів і т.д. Позаяк технологія ADO заснована на стандартних інтерфейсах COM, які є системним механізмом Windows, це скорочує загальний обсяг працюючого програмного коду й дозволяє поширювати прикладні програми БД без допоміжних програм і бібліотек.

Специфікація OLE DB розрізняє наступні типи об'єктів, які будуть розглянуті нижче.

- Перелічувальник (Enumerator) виконує пошук джерел даних або інших перелічувальників. Використовується для забезпечення функціонування провайдерів ADO.
- Об'єкт-джерело даних (Data Source Object) представляє сховище даних.
- Сесія (Session) поєднує сукупність об'єктів, що звертаються до одного сховища даних.
- Транзакція (Trasaction) інкапсулює механізм виконання транзакції.
- Команда (Command) містить текст команди й забезпечує її виконання. Командою може бути запит SQL, звертання до таблиці БД і т.д.
- Набір рядів (Rowset) являє собою сукупність рядків даних, що є результатом виконання команди ADO.
- Об'єкт-помилка (Error) містить інформацію про виняткову ситуацію.

Розглянемо функціональні можливості основних об'єктів і інтерфейсів OLE DB. Зокрема, об'єкти-перелічувальники. Об'єкти-перелічувальники забезпечують пошук будь-яких об'єктів ADO, які мають доступ до джерел даних. Первинний пошук джерел даних здійснюється в провайдері ADO. Перелічувальники можуть відбирати тільки джерела даних конкретних типів, тому провайдер забезпечує доступ до конкретного типу сховища даних.

У складі ADO є системний кореневий перелічувальник, що виконує початковий пошук інших перелічувальників і джерел даних. Його можна використати, знаючи його ідентифікатор класу *CLSID_OLEDB_ENUMERATOR*.

В Delphi GUID глобальний перелічувальник міститься у файлі \Delphi7\Source \Vcl\OleDB.pas.

CLSID_OLEDB_ENUMERATOR: TGUID= '{C8B522DO-5CF3-11CE-ADE5-OOAA0044773D}

Функції перелічувальника містяться в інтерфейсі *isourcesRowset*. Метод

function GetSourcesRowset(const punkOuter: lUnknown; const riid: TGUID; cPropertySets: UINT; rgProperties: PDBPropSetArray; out ppSourcesRowset: lUnknown):

HResult; stdcall; повертає посилання на об'єкт набору рядів , що містить відомості про знайдені джерела даних або перелічувальниках.

Внутрішній механізм ADO, що забезпечує з'єднання зі сховищем даних, використовує два типи об'єктів. Це об'єкти-джерела даних і об'єкта-сесії. Об'єкт-джерело даних забезпечує подання інформації про необхідне реальне джерело даних і підключення до нього. Для введення даних про сховище даних використовується інтерфейс *iDBProperties.* Для успішного підключення необхідно визначити обов'язкові відомості. Імовірно, для будь-якого сховища даних буде актуальної інформація про його ім'я, користувача й пароль. Однак кожний тип сховища має власні унікальні налаштування. Для одержання списку всіх обов'язкових параметрів з'єднання з даним сховищем можна скористатися методом

*function GetPropertylnfo(cPropertylDSets: UINT; rgPropertylDSets: PDBPropIDSetArray; var pcPropertylnfoSets: UINT; out prgPropertylnfoSets: PDBPropInfoSet; ppDescBuffer: PPOleStr): HResult; stdcall; ,*який повертає заповнену структуру DBPROPINFO.

```
PDBPropInfo = ^TDBPropInfo;
 DBPROPINFO = packed record
pwszDescription: PWideChar;
dwPropertyl: DBPROPID;
dwFlags: DBPROPFLAGS;
vtType: Word;
vValues: OleVariant;
end;
TDBPropInfo = DBPROPINFO;
```

Для кожного обов'язкового параметра в елементі *dwFlags* встановлюється значення *DBPROPFLAGS_REQUIRED.* Для ініціалізації з'єднання необхідно використати метод function *Initialize: HResult; stdcall;* інтерфейсу *iDBinitiaiize* об'єкта-джерела даних.

З об'єкта-джерела даних можна створювати об'єкти-сесії. Для цього використовується метод :

function CreateSession(const punkOuter: lUnknown; const riid: TGUID; out ppDBSession: lUnknown}: HResult; stdcall; інтерфейсу *iDBCreateSession*. Сесія призначена для забезпечення роботи транзакцій і наборів рядів.

Керування транзакціями в OLE DB реалізовано на двох рівнях. По-перше, об'єкт сесії всіма має всі необхідні методи. Він має інтерфейси *ITransaction, ITransactionJoin, ITransactionLocal, ITransactionObject. В* середині сесії транзакція управляється інтерфейсами *ITransactionLocal, ItransactionSC, ITransaction і їхніми методами StartTransaction, Commit, Rollback.*

По-друге, для об'єкта сесії можна створити об'єкт транзакції за допомогою методу

function GetTransactionObject(ulTransactionLevel: UINT; out ppTransactionObject: ITransaction): HResult; stdcall; інтерфейсу *ITransactionObject*, що повертає посилання на інтерфейс об'єкта-транзакції.

Роботу з даними ADO забезпечує об'єкт *Набір рядків*. Він інкапсулює сукупність рядків із джерела даних, механізми навігації по рядах і підтримки рядків в актуальному стані. Об'єкт сесії має обов'язковий інтерфейс *IOpenRowset* з методом:

function OpenRowset(const punkOuter: IUnknown; pTablel: PDBID; pIndexID: PDBID; const riid: TGUID; cPropertySets: UINT; rgPropertySets: PDBPropSetArray; ppRowset: PlUnknown): HResult; stdcall;, який відкриває необхідний набір рядів. Залежно від можливостей джерела даних набір рядів може підтримувати різні інтерфейси. Але п'ять із них є обов'язковими:

- *IRowset* - забезпечує навігацію по рядах;
- *IAccessor* - забезпечує подання інформації про формат рядків, що містяться в буфері набору рядів;
- *IRowsetinfo* - дозволяє одержати інформацію про набори рядків (наприклад, число рядів або число обновлених рядів);
- *Icoiumnsinfo* - дозволяє одержати інформацію про колонків рядків (найменування, тип даних, можливість відновлення й т.д.);
- *IconvertType* - містить єдиний метод canConvert, що дозволяє визначити можливість перетворення типів даних у наборі рядків.

На відміну від звичної практики розробки інтерфейсів у рамках моделі COM, інтерфейси OLE DB часто мають усього один-два методи. У результаті більша група інтерфейсів реалізує обмаль стандартних функцій.

Додаткові можливості по керуванню набором рядів надають наступні інтерфейси:

- *IRowsetchange* - виконує зміни в наборі рядів (вносить зміни, додає нові ряди, видаляє ряди й т.д.);
- *IRowsetidentity* - дозволяє порівнювати ряди різних рядів;
- *IRowsetindex* - забезпечує використання індексів;
- *IRowsetLocate* - виконує пошук у наборі рядів;
- *IRowsetupdate* - реалізує механізм кешування змін.

Програмні засоби ADO були б неповними, якби не мали можливості використати для роботи з даними мову SQL. Оператори DML і DDL, ряд спеціальних операторів ADO носять спільну назву текстових команд. Об'єкт-команда інкапсулює саму текстову команду й механізм обробки й передачі команди. Об'єкт команди виконує наступні операції:

- розбір тексту команди;

- в'язування команди із джерелом даних;
- оптимізацію команди;
- передачу команди джерелу даних.

Головний інтерфейс об'єкта команди icommand має три методи:

- *function Cancel: HResult; stdcall;*-скасовує виконання команди;
- *function Execute(const punkOuter: lUnknown; const riid: TGUID; var pParams: DBPARAMS; pcRowsAffected: PInteger; ppRowset: PlUnknown): HResult; stdcall;*-виконує команду;
- *function GetDBSession(const riid: TGUID; out ppSession: lUnknown): HResult; stdcall;*-повертає посилання на інтерфейс сесії, що викликав дану команду. Крім основного, об'єкт команди забезпечує доступ до додаткових інтерфейсів:
- *ICommandPrepare* - містить два методи (Prepare та Unprepare) для підготовки команди;
- *iCommandProperties* - задає для команди властивості, які повинні підтримуватися набором даних, що повертають командою;
- *iCommandText* - керує текстом команди (цей інтерфейс обов'язковий для об'єкта команди);
- *iCommandWithParameters* - забезпечує роботу з параметрами команди.

Для з'єднання прикладної програми із джерелом даних (сервером SQL, локальної СУБД, файловою системою й т.д. використовуються провайдери ADO, які забезпечують обробку даних. Для кожного типу сховища даних повинен існувати провайдер ADO. Провайдер «знає» про місце розташування сховища даних і його зміст, уміє звертатися до даних із запитами й інтерпретувати службову інформацію, що повертає, і результати запитів з метою їхньої передачі прикладній програмі. Список установлених у даній операційній системі провайдерів доступний для вибору при встановленні з'єднання через компонент *TADOConnection.* При інсталяції *Microsoft Active Data Objects* в операційній системі Windows встановлюються наступні стандартні провайдери.

- *Microsoft Jet OLE DB Provider* забезпечує з'єднання з даними СУБД Access за допомогою технології BAT.
- *Microsoft OLE DB Provider for Microsoft Indexing Service* забезпечує доступ тільки для читання до файлів і Internet-ресурсів Microsoft Indexing Service.
- *Microsoft OLE DB Provider for Microsoft Active Directory Service* забезпечує доступ до ресурсів служби каталогів (Active Directory Service).

- *Microsoft OLE DB Provider for Internet Publishing* дозволяє використовувати ресурси, надавані Microsoft FrontPage, Microsoft Internet Information Server, HTTP-файли.
- *Microsoft Data Shaping Service for OLE DB* дозволяє використовувати ієрархічні набори даних.
- *Microsoft OLE DB Simple Provider* призначений для організації доступу до джерел даних, підтримуючі тільки базисні можливості *OLE DB*.
- *Microsoft OLE DB Provider for ODBC drivers* забезпечує доступ до даних, які вже «прописані» за допомогою драйверів *ODBC.*
- *Microsoft OLE DB Provider for Oracle* забезпечує з'єднання із сервером Oracle.
- *Microsoft OLE DB Provider for SQL Server* забезпечує з'єднання із сервером *Microsoft SQL Server.*

Механізм доступу до даних через ADO і численні об'єкти й інтерфейси реалізовані в VCL Delphi/C++Builder у вигляді набору компонентів, розташованих на сторінці *ADO.* Всі необхідні інтерфейси, що забезпечують роботу компонентів, оголошені й описані у файлах OleDB.pas і ADODB.pas у папці \Delphi7\Source\Vcl. Наявність компонента *TADOConnection* свідчить про певну надлишковість компонентної моделі Borland. *TADOConnection* увібрав можливості перелічувальника, джерела даних і сесії з можливостями обслуговування транзакцій.

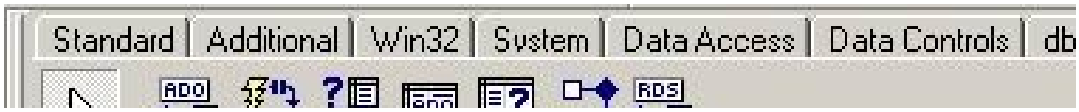


Рис. 9.16. Сторінка ADO візуального середовища Borland

Текстові команди ADO реалізовані в компоненті TADOCommand. Набори рядів (нотація Microsoft) можна одержати за допомогою компонентів *TADOTable, TADOQuery, TADOstoredProc*. Кожний з них реалізує спосіб доступу до конкретного типу представлення даних у сховищі. Набір властивостей і методів компонентів ADO забезпечує реалізацію всіх необхідних прикладній програмі БД функцій. Способи використання компонентів ADO дещо нагадують технологію стандартних компонентів VCL доступу до даних. Однак при необхідності розробник може використати всі можливості інтерфейсів ADO, звертаючись до них через відповідні об'єкти ADO. Компоненти доступу до даних ADO можуть використати два варіанти підключення до сховища даних. Це стандартний метод ADO і стандартний метод Delphi. У першому випадку компонента використовують властивість C*onnectionstring* для прямого звертання до сховища даних. У другому випадку використовується спеціальний компонент *TADOConnection*, що забезпечує розширене керування з'єднанням і дозволяє

звертатися до даного декількох компонентам одночасно. Детальніше технологію використання доцільніше розглянути на практиці, тобто в лабораторному практикумі.

На сторінці *ADO* палітри компонентів Delphi/C++Builder, крім компонентів з'єднання є стандартні компоненти, що інкапсулюють набір даних і адаптовані для роботи зі сховищем даних ADO (Рис. 9.17.). Це компоненти:

- *TADODataSet* - універсальний набір даних;
- *TADOTable* - таблиця БД;
- *TADOQuery* - запит SQL;
- *TADOStoredProc* - процедура, що зберігається.

Як і для всіх компонентів, що інкапсулюють набір даних, їхнім спільним предком є клас TDataSet, що надає базові функції керування набором даних , а прямим абстрактний клас TCustomADODataSet.

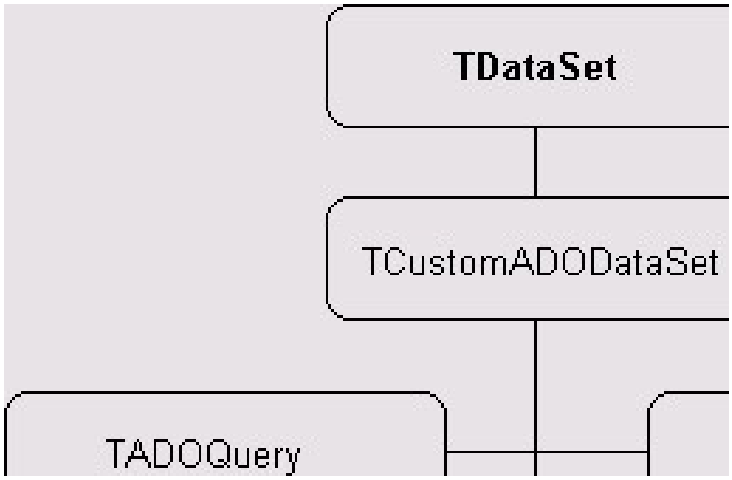


Рис. 9.17. Ієрархія класів наборів даних ADO

Компоненти ADO мають звичайний набір властивостей і методів, а необхідний для доступу до даних через ADO механізм успадковують від свого спільного предка - класу *TCustomADODataSet.*Тому спочатку розглянемо властивості класу *TCustomADODataSet* і тільки потім перейдемо до компонентів ADO. Компонент *TADOconnection* забезпечує доступ до всіх компонентів, які використовують його для доступу до сховища даних ADO. Всі відкриті в такий спосіб набори даних доступні через індексовану властивість *property DataSets[Index: Integer]: TCustomADODataSet;*. Кожний елемент цього списку містить дескриптор компонента доступу до даних ADO (тип CustomADODataSet). Спільна кількість зв'язаних компонентів з наборами даних повертається властивістю *property DataSetCount: Integer;*. Така побудова класу дозволяє для всіх компонентів-нащадків централізовано встановити тип використовуваного курсору за допомогою властивості

type TCursorLocation = (clUseServer, clUseClient); property CursorLocation: TCursorLocation; .

Значення *clUseClient* визначає локальний курсор на боці клієнта, що дозволяє виконувати будь-які операції з даними, у тому числі не підтримувані сервером. Значення *cIUseServer* задає курсор на сервері, що реалізує тільки можливості сервера, але забезпечує швидку обробку більших масивів даних.

Наприклад:

```
for i := 0 to ADOConnection.DataSetCount - 1 do
begin
if ADOConnection.DataSets[i].Active = True then ADOConnection.DataSets[i].Close;
ADOConnection.DataSets[i].CursorLocation := clUseClient; end;
```

Крім наборів даних компонентів *TADOConnection* забезпечує виконання команд ADO. Команду ADO інкапсулює спеціальний компонент *TADOCommand*, що розглядається нижче. Всі команди ADO, що працюють зі сховищем даних через це з'єднання, доступні для керування через індексовану властивість *property Commands[Index: Integer]: TADOCommand.* Кожний елемент цього списку являє собою екземпляр класу *TADOCommand.* Загальна кількість доступних команд повертається властивістю *property CommandCount: Integer.*

Наприклад, відразу після відкриття з'єднання можна виконати всі зв'язані команди ADO, реалізувавши в такий спосіб щось на зразок скрипта:

```
procedure TForml.ADOConnectionConnectComplete(Connection: TADOConnection;
const Error: Error; var EventStatus: TEventStatus);
var i, ErrorCnt: Integer;
begin
if EventStatus = esOK then
for i := 0 to ADOConnection.CommandCount - 1 do
try if ADOConnection.Commands[i].CommandText <>
then ADOConnection.Commands[i].Execute; except
on E: Exception do Inc(ErrorCnt);
 end;
end;
```

Однак компонент TADOConnection може виконувати команди ADO самостійно, без допомоги інших компонентів. Для цього використовується метод перевантаження

```
function Execute(const CommandText: WideString; ExecuteOptions:
TExecuteOptions = []): _RecordSet; overload;
procedure Execute(const CommandText: WideString;
var RecordsAffected:
Integer; ExecuteOptions: TExecuteOptions = [eoExecuteNoRecords]);
overload;
```

Виконання команди здійснюється процедурою *Execute* (якщо команда не повертає набір записів) або однойменною функцією *Execute* (якщо команда повертає набір записів).

Параметр *commandText* повинен містити текст команди. Параметр *RecordsAffected* повертає число оброблених командою записів (якщо вони є). Параметр

type

TExecuteOption = (eoAsyncExecute, eoAsyncFetch, eoAsyncFetchNonBlocking, eoExecuteNoRecords);

TExecuteOptions = set of TExecuteOption;

задає умови виконання команди:

- eoAsyncExecute - команда виконується асинхронно (з'єднання не буде очікувати закінчення виконання команди, а продовжить роботу, обробивши сигнал про завершення команди, коли він надійде);
- *eoAsyncFetch* - команда одержує необхідні записи також асинхронно;
- *eoAsyncFetchNonBlocking* - команда отримує необхідні записи також асинхронно, але при цьому створена нитка не блокується;
- *eoExecuteNoRecords* - команда не повинна повертати запис.

Якщо джерело даних прийняло команду для виконання й сповістило про це з'єднанню, викликається метод-обробник

TWillExecuteEvent = procedure(Connection: TADOConnection;

var CommandText: WideString; var CursorType: TCursorType; var LockType:

TADOLockType; var ExecuteOptions: TExecuteOptions;

var EventStatus:

TEventStatus; const Command: _Command;

const Recordset: _Recordset)

of object;

property OnWillExecute: TWillExecuteEvent;

Після виконання команди викликається метод-обробник

TExecuteCompleteEvent = procedure(Connection: TADOConnection; RecordsAffected: Integer;

const Error: Error; var EventStatus: TEventStatus;

const Command: _Command;

const Recordset: _Recordset) of object;

property OnExecuteComplete: TExecuteCompleteEvent;

Перед відкриттям набору даних необхідно встановити тип використовуваних при редагуванні записів блокування. Для цього застосовується властивість :

type TADOLockType = (ItUnspecified, ItReadOnly, ItPessimistic, ItOptimistic, ItBatchOptimistic); property LockType: TADOLockType;, що визначає режими блокування. Зокрема значення, що входять у згадану властивість визначають:

- ItUnspecified - блокування визначається джерелом даних, а не компонентом;
- *ItReadOnly* - набір даних відкриється в режимі тільки для читання;
- *ItPessimistic* - редагує запис, що, блокується на весь час редагування до моменту збереження в сховище даних;
- *ItOptimistic* - запис блокується тільки на час збереження змін у сховище даних;
- *ItBatchOptimistic* - запис блокується на час збереження в сховище даних при виклику методу updateBatch.

Набір даних відкривається методом Open і закривається методом *close.* Також можна використати властивість *property Active: Boolean;*. Поточний стан набору даних можна визначити властивістю :

type

TObjectState = (stClosed, stOpen, stConnecting, stExecuting, stretching);

TObjectStates = set of TObjectState;

property RecordsetState: TObjectStates;

Набір даних у компонентах ADO базується на використанні об'єкта набору записів ADO. Прямий доступ до цього об'єкта можливий за допомогою властивості :

property Recordset: _Recordset; .

Але оскільки всі основні методи інтерфейсів об'єкта набору записів ADO перекриті методами класу, у звичайних випадках прямий доступ до об'єкта вам не знадобиться. Після відновлення набору даних викликається метод-обробник:

TRecordsetEvent = procedure(DataSet: TCustomADODataSet; const Error: Error;

var EventStatus: TEventStatus) of object; property OnFetchComplete: TrecordsetEvent;,

де *Error* - посилання на об'єкт помилки ADO, якщо вона виникла. Якщо ж набір даних працює в асинхронному режимі, при відновленні викликається метод-обробник:

TFetchProgressEvent = procedure(DataSet: TCustomADODataSet;

Progress, MaxProgress: Integer;

var EventStatus: TEventStatus) of object;

property OnFetchProgress: TFetchProgressEvent; ,

де параметр Progress показує частку виконання операції. Для набору даних ADO залежно від його призначення можна вибрати тип і місце розташування використовуваного курсору. Місце розташування курсору визначається властивістю :

type TCursorLocation = (clUseServer, clUseClient);

property CursorLocation: TCursorLocation;

Курсор може перебувати на сервері (CIUseServer) або на клієнті (CIUseClient).

- Серверний курсор використовується при роботі з більшими наборами даних, які недоцільно пересилати клієнтові цілком. При цьому трохи знижується швидкість роботи клієнтського набору даних.
- Клієнтський курсор забезпечує передачу набору даних клієнтові. Це позитивно позначається на швидкодії, але такий курсор доцільно використовувати тільки для невеликих наборів даних, що не завантажують канал зв'язку із сервером.

При використанні клієнтського курсору необхідно додатково встановити властивість :

TMarshalOption = (moMarshalAll, moMarshalModifiedOnly); property MarshalOptions: TmarshalOption, яка управляє обміном даних клієнта із сервером. Якщо з'єднання із сервером швидке, можна використати значення *moMarshalAll,* що дозволяє повернення серверу всіх записів набору даних. У зворотньому випадку для прискорення роботи компонента можна застосувати властивість *moMarshalModifiedOnly*, що забезпечує повернення тільки модифікованих клієнтом записів.

Тип курсору визначається властивістю

TCursorType = (ctUnspecified, CtOpenForwardOnly, ctKeyset, ctDynamic,

ctStatic);

property CursorType: TCursorType;

Значення властивості наступні:

- *ctunspecified - курсор не заданий, тип курсору визначається можливостями джерела даних;*
- *ctOpenForwardOnly - односпрямований курсор, що допускає переміщення тільки вперед; використовується при необхідності швидкого одиночного проходу по всіх записах набору даних;*
- *ctKeyset* - двонаправлений локальний курсор, що не забезпечує перегляд доданими й вилучених іншими користувачами записів;
- *ctDynamic* - двонаправлений курсор, відображає всі зміни, вимагає найбільших витрат ресурсів;
- *ctStatic* - двонаправлений курсор, повністю іґнорує зміни, внесені іншими користувачами.

Відповідно до й після кожного переміщення курсору в наборі даних викликаються методи - обробники:

TRecordsetReasonEvent = procedure(DataSet: TCustomADODataSet;

const Reason: TEventReason;

var EventStatus: TEventStatus) of object;

property OnWillMove: TRecordsetReasonEvent;

та

TP.ecordsetErrorEvent = procedure(DataSet: TCustomADODataSet;

const Reason: TEventReason;

const Error: Error; var EventStatus: TEventStatus) if object;

property OnMoveComplete: TRecordsetErrorEvent;

де параметр *Reason* дозволяє довідатися, який метод викликав це переміщення. Клас TParameter інкапсулює окремий параметр. Ім'я параметра визначається властивістю *property Name: WideString;*.Тип даних, якому повинне відповідати його значення, визначається властивістю *TDataType = TFieldType; property DataType: TDataType;*.

Із-за того, що параметри взаємодіють із полями таблиць БД, тип даних параметрів співпадає з типами даних полів. Від типу даних залежить розмір параметра *property Size:Integer;,* який може бути змінений для рядкового або символьного типу даних та їм подібних. Саме значення параметра міститься у властивості property *Value: OleVariant;*

А властивість

type

TParameterAttribute = (paSigned, paNullable, paLong);

TParameterAttributes = Set of TParameterAttribute;

property Attributes: TParameterAttributes; контролює значення, що привласнює параметру:

- *paSigned* - значення може бути символьним;
- *paNullable* - значення параметра може бути порожнім;
- *paLong* - значення може містити дані типу BLOB.

Тип параметра визначається властивістю :

type TParameterDirection = (pdUnknown, pdlnput, pdOutput, pdlnputOutput, pdReturnValue);

property Direction: TParameterDirection;

pdUnknown - невідомий тип, джерело даних спробує визначити його самостійно;

pdinput - вхідний параметр, використовується в запитах і збережених процедурах;

pdOutput - вихідний параметр, використовується в збережених процедурах;

pdlnputOutput - вхідний і вихідний параметр одночасно, використовується в збережених процедурах;

pdReturnValue - параметр для передачі значення.

Якщо параметр повинен передавати більші бінарні масиви (наприклад, зображення або файли), то значення параметра можна завантажити, використовуючи методи

procedure LoadFromFile(const FileName: String; DataType: TDataType); та procedure *LoadFromStream(Stream: TStream; DataType: TDataType);*

Компонент *TADODataSet* призначений для подання набору даних зі сховища даних ADO. Він простий у використанні, маючи тільки кілька власних властивостей і методів, і застосовує функції свого предка - класу *TCustomADODataSet*. Це єдиний компонент ADO, що інкапсулює набір даних, для якого опубліковані властивості, що дозволяють управляти командою ADO. Це властивості *property CommandText: WideString;* та *property CommandType: TCommandType;*.У результаті компонентів являє собою гнучкий інструмент, що дозволяє (залежно від типу команди і її тексту) одержувати дані з таблиць, запитів SQL, збережених процедур, файлів і т.д. Наприклад, вибираючи потрібні значення властивості *CommandType = cmdText* і ввести у властивість *CommandText* текст запиту SQL з редактора:

ADODataSet,CommandType = cmdText;

ADODataSet. CommandText := Memol.Lines.Text;

отримаємо запит SQL готовий до виконання. Для запитів SQL можна застосовувати тільки мову *Data Manipulation Language* (використовується тільки *SELECT*). З'єднання з базою даних визначається властивістю *Connectionstring* або *Connection*. Набір даних відкривається й закривається властивістю *Active* або методами *Open* і *Close*. Компонент *TaDOTable* забезпечує використання в прикладній програмі Delphi таблиць БД, підключених через провайдери *OLE DB*. За своїми функціональними можливостями і застосуванню він схожий до стандартного табличного компонента. Як відомо, в основі компонента лежить використання команди ADO, але її властивості налаштовані заздалегідь і зміні не підлягають. Ім'я таблиці БД визначається властивістю *property TableName: WideString;* . Інші властивості й методи компонента забезпечують застосування індексів (цієї можливості позбавлений будь-який компонент запиту). Позаяк не всі провайдери ADO забезпечують пряме використання таблиць БД, то для доступу до них може знадобитися запит SQL. Якщо властивість *property TableDirect: Boolean*; має значення *True*, здійснюється прямий доступ до таблиці. У зворотньому випадку компонент генерує відповідний запит. Властивість *property Readonly: Boolean;* дозволяє включити або відключити для таблиці режим «тільки для читання». Компонент *TADOQuery* забезпечує застосування запитів SQL при роботі з даними через ADO. По своїй функціональності він подібний до стандартного компонента запиту.

Текст запиту визначається властивістю *property SQL: TStrings;* . Параметри запиту визначаються властивістю *property Parameters: TParameters;*. Якщо запит повинен повертати набір даних, для його відкриття використовується властивість :

property Active: Boolean;

або метод

procedure Open;

У зворотньому випадку досить використати метод *function ExecSQL: Integer; ExecSQL*
Число оброблених запитом записів повертає властивість property *RowsAffected: Integer;*. Компонент *TADOStoredProc* дозволяє використати в прикладній програмі Delphi, що звертаються до даних через ADO, збережені процедури. Він схожий на стандартний компонент збереженої процедури. Ім'я збереженої процедури визначається властивістю property *ProcedureName: WideString;*. Для визначення вхідних і вихідних параметрів використовується властивість *property Parameters: TParameters;* Якщо процедура буде застосовуватися без змін багаторазово, має сенс заздалегідь підготувати її виконання на сервері. Для цього властивості *property Prepared: Boolean;* привласнюється значення *True*.

Події ADO призначені для тієї ж мети, що й події VCL. Багато які з них мають аналогічні події VCL і компоненти викликають із подій ADO події VCL. У компонентах доступні як події ADO, так і події BDE.
OnWillConnect - викликається перед установкою з'єднання.
OnConnectComplete - після установки з'єднання.
OnDisconnect - при розриві з'єднання.
Ці події інкапсульовані в компоненті TADOConnection.
OnBeginTransComplete - при виконанні BeginTrans.
OnCommitTransComplete – при виконанні CommitTrans.
OnRollbackTransComplete - при виконанні RollbackTrans.
Ці події інкапсульовані в компоненті TADOConnection, а не в компоненті TADOCommand, як можна було б припустити. Це пов'язано з тим, що в ADO об'єкту команди як такого немає й з цієї причини він не може одержувати повідомлення. В TADOConnection також інкапсульована подія OnInfoMessage, що викликається при надходженні з сервера додаткової інформації. Формат і призначення залежать від сервера, з яким ви працюєте.В ADO так само є події пов'язані з набором даних, а не із з'єднанням, як вищеописані. Вони інкапсульовані в компоненти, які мають набір даних - *TADODataSet, TADOTable, TADOQuery й TADOStoredProc*. Ці події можна розбити на три групи:
*OnFetchProgres*s - багаторазово викликається в процесі вибірки набору даних.
OnFetchComplete - завершення вибірки.
OnWillMove, OnMoveComplete - викликаються до й після зміни положення поточного запису. *OnWillMove* дозволяє скасувати дію.
OnEndOfRecordset - викликається при досягненні кінця набору даних, дозволяє додати новий запис.
OnWillChangeField, OnFieldChangeComplete - до й після зміни поточного запису набору.

OnWillChangeRecord, OnRecordChangeComplete - викликаються до й після зміни, додавання, видалення рядка набору й скасуванню цих дій.

OnWillChangeRecordset, OnRecordsetChangeComplete - викликаються до й після відкриття, закриття, повторного запиту й синхронізації набору даних.

Багато подій допускають переривання дії. Ця можливість буває корисна при асинхронній роботі з сервером. Наприклад, для переривання занадто довгого запиту.

В ADO є можливість, яка не має аналогів ні в BDE ні в InterBase. Це асинхронне виконання операцій з сервером. Можуть асинхронно виконуватися установка з'єднання із сервером (Connection), виконання команди (Execute) і вибірка набору даних (Fetch)

Для включення цього режиму необхідно встановити властивість ConnectOptions компонента *TADOConnection* в *coAsyncConnect*.

При встановленні з'єднання відбувається:

1)Викликається обробник події *OnWillConnect;*

2)Керування передається в програму;

3)Після закінчення з'єднання, як успішного, так і помилкового, викликається обробник події *OnConnectComplete.*

Треба відмітити, що всі компоненти ADO, за винятком компонента *TADOConnection* при активізації або виконанні виконують команду ADO. Встановіть у властивості *ExecuteOptionseoAsyncExecute.*

При виконанні відбувається:

1)Викликається обробник події *OnWillExecute;*

2)Керування передається в програму;

3)Після закінчення виконання команди, як успішного, так і помилкового, викликається обробник події *OnExecuteComplete.*

Асинхронна вибірка даних підтримується в компонентах *TADODataSet, TADOTable, TADOQuery, TADOStoredProc.* Для її включення встановіть у властивості *ExecuteOptionseoAsyncFetch*. Після виконання команди відбувається передача даних. У процесі передачі багаторазово викликається обробник повідомлення *OnFetchProgress*. В його параметрах є як кількість уже переданих записів, так і загальна кількість. Це дуже зручно для створення індикаторів типу *TProgressBar.* Після того, як вибірка з сервера закінчена, викликається обробник події *OnFetchComplete*. Команді ADO відповідає компонент *TADOCormand.* Методи цього компонента багато в чому збігаються з методами класу *TCustomADODataSet*, хоча цей клас не є предком компонента (Рис.9.18.). Він призначений для виконання команд, які не повертають набори даних.

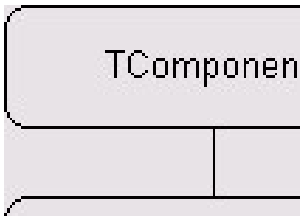


Рис.9.18. Ієрархія класів компонента TADOCommand

Безпосереднім предком є клас *TComponent.* До його функціональності просто доданий механізм з'єднання із БД через ADO і засоби представлення команди. Команда передається в сховище даних ADO через власне з'єднання або через компонент *TADOconnection*, аналогічно іншим компонентам ADO. Текстове представлення виконуваної команди повинне міститися у властивості : *property CommandText: WideString;*

Однак команду можна визначити й іншим способом. Пряме посилання на потрібний об'єкт команди ADO можуть бути задані властивістю *property CommandObject:Command;*.

Тип команди визначається властивістю :

type TCommandType = (cmdUnknown, cmdText, cmdTable, cmdStoredProc, cmdFile, cmdTableDirect);

property CommandType: TCommandType;

Оскільки тип *TCommandType* також використовується в класі *TCustomADODataSet,* де необхідно представляти всі можливі види команд, стосовно компонента TADOcommand цей тип має надмірність. Тут не можна встановити значення *cmdTable, cmdFile*, *cmdTableDirect*, а тип *cmdStoredProc* повинен позначати тільки ті збережені процедури, які не повертають набір даних. Якщо команда повинна містити текст запиту SQL, властивість *CommandType* повинне мати значення *cmdText.* Для виклику збереженої процедури необхідно визначити тип *cmdStoredProc*, а у властивості *CommandText* увести ім'я процедури. Якщо для виконання команди необхідно визначити параметри, використовується властивість :

property Parameters: TParameters;

Виконання команди здійснюється методом Execute:

function Execute: _RecordSet; overload;

function Execute(const Parameters: OleVariant): _Recordset;

overload;

function Execute(var RecordsAffected: Integer; var Parameters: OleVariant; ExecuteOptions: TExecuteOptions = []): _RecordSet; overload;

Розробник може використати кожну із представлених нотацій методів перевантаження:

параметр *RecordsAffected* повертає число оброблених записів;

параметр *Parameters* задає параметри команди;

параметр *ExecuteOptions* визначає умови виконання команди:

TExecuteOption = (eoAsyncExecute, eoAsyncFetch, eoAsyncFetchNonBlocking, eoExecuteNoRecords); TExecuteOptions = set of TExecuteOption;

eoAsyncExecute - асинхронне виконання команди;

eoAsyncFetch - асинхронна передача даних;

eoAsyncFetchNonBlocking - асинхронна передача даних без блокування потоку;

eoExecuteNoRecords - якщо команда повертає набір записів, то вони не передаються в компонент. При роботі з компонентом *TADOConnection* бажано використати опцію *eoExecuteNoRecords*. Для переривання виконання команди використовується метод *procedure Cancel* . Поточний стан команди можна визначити властивістю :

type

TObjectState = (stClosed, stOpen, stConnecting, stExecuting, stFetching);

TObjectStates = set of TObjectState; property States: TObjectStates;

Таким чином технологія ADO у візуальному середовищі Borland забезпечує універсальний спосіб доступу до гетерогенних джерел даних. Завдяки тому, що функції ADO реалізовані на основі інтерфейсів OLE DB і COM, програми для доступу до даних не потрібно додаткових бібліотек, крім інстальованого ADO. Компонент TADOConnection забезпечує з'єднання із джерелами даних через провайдери OLE DB. Компоненти TADODataSet, TADOTable, TADOQuery, TADOStoredProc забезпечують використання наборів записів у прикладній програмі. Властивості й методи компонентів дозволяють створювати повнофункціональні програми. Компонент TADOCommand інкапсулює текстову команду ADO, що надає додаткові можливості роботи з даними - з компонентів можна прямо звертатися до необхідних об'єктів і інтерфейсів ADO.

9.3.4. Використання сервера баз даних InterBase і компонент InterBase Express

На сторінці *InterBase* Палітри компонентів містяться компоненти доступу до даних, адаптовані для роботи із сервером *InterBase* і об'єднані назвою *InterBase Express*. Компоненти з набору *InterBase* Express призначені для роботи із сервером *InterBase* версії не нижче 5.5. Їхня перевага полягає в реалізації всіх функцій за рахунок прямого звертання до API сервера *InterBase*. Завдяки цьому істотно підвищилася швидкість роботи компонентів. Методика використання цих компонентів у прикладних програмах БД не відрізняється від стандартної методики. Будь-який новий компонент, інкапсулює набір даних, зовсім звичайним образом через компонент *TDataSource* можна підключити до будь-якого стандартного компонента відображення даних.

Для компонентів *InterBase Express* з'єднання із сервером БД здійснює компонент *TIBDatabase*, розташований на сторінці *InterBase*. Для створення прикладної клієнт/серверної програми необхідно не тільки мати працюючий сервер, але й інсталювати на клієнтських робочих місцях спеціальне програмне забезпечення, що виконує з'єднання клієнтської прикладної програми із сервером. Механізм доступу до даних *InterBase Express* використовує для звертань до сервера можливості клієнтського ПЗ *InterBase*, яке має бути інстальоване на комп'ютері. Якщо з даного комп'ютера доступні бази даних якого-небудь сервера на платформі *InterBase,* то розглянуті тут компоненти можуть звертатися до цього сервера без BDE. Як результат всі компоненти *InterBase Express*, що інкапсулюють набір даних, повинні звертатися до бази даних тільки через компонент з'єднання *TIBDatabase*.

Із-за того, що для доступу до бази даних компонентам *InterBase Express* не потрібна участь BDE, для створення з'єднання використовується тільки одна властивість *DatabaseName.* У ньому необхідно вказати повний шлях (включаючи ім'я сервера) до обраного файлу БД із розширенням gdb. Для цього можна скористатися стандартним діалогом вибору файлу при клацанні на кнопці властивості в Інспекторі об'єктів. Компонент має власний редактор, що дозволяє визначити значення основних властивостей, що забезпечують з'єднання з базою даних

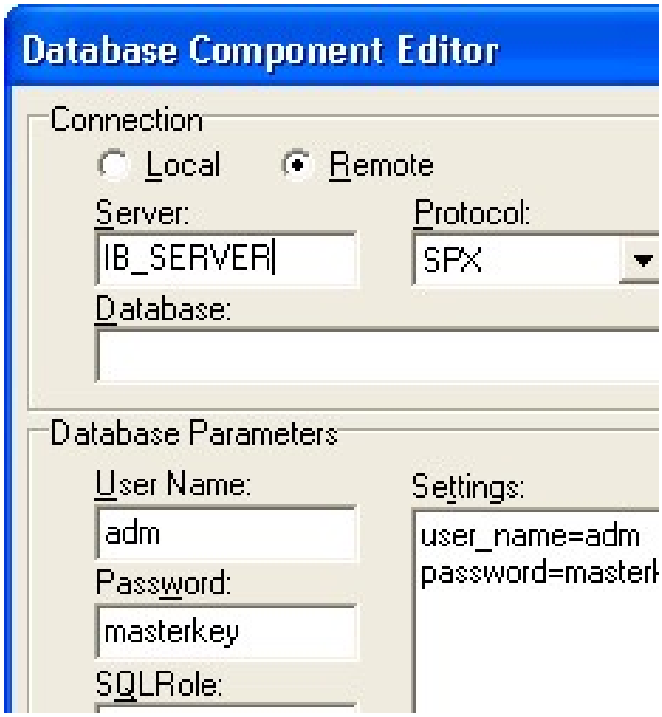


Рис. 9.19. Редактор компонента *TIBDatabase*

Налаштування з'єднання проводиться в такий спосіб. На панелі *Connection* вибирається необхідний сервер *InterBase* (локальний або віддалений), потім у списку *Protocol* визначається використовуваний мережний протокол і за допомогою кнопки *Browse* вибирається файл бази даних. На панелі *Database Parameters* визначаються ім'я користувача,

його пароль і роль. Також можна вибрати й набір шрифтів для мовної адаптації прикладної програми (список *Character Set*).

Для визначення параметрів, що вводять при підключенні, (ім'я користувача, пароль, схема, роль і т.д.) також можна використати властивості *Params* і *LoginPrompt*. Повний список індексів всіх можливих параметрів з'єднання Interbase можна знайти у файлі *\Delphi7\Source\Vcl\IBHeader.pas*.

Компонент *TIBTransaction* інкапсулює засоби управління транзакцією при роботі із сервером *InterBase*. Для цього він повинен бути пов'язаний з компонентом *TiBDatabase* за допомогою своєї властивості *property DefaultDatabase: TiBDatabase;* .Один компонент транзакції може бути пов'язаний з декількома компонентами TiBDatabase. Для цього необхідно визначити один компонент транзакції у властивостях DefaultTransaction всіх необхідних компонентів з'єднань (див. вище). Список всіх зв'язаних компонентів з'єднань міститься у властивості *property Databases[Index: Integer]: TiBDatabase*; а їхню загальну кількість повертає властивість property *DatabaseCount: Integer;* . Під час виконання нове з'єднання може бути пов'язане із транзакцією методом *function AddDatabase(db: TIBDatabase): Integer;*. Або ж, зв'язок може бути скасований: *procedure RemoveDatabase(Idx: Integer);*. А метод procedure *RemoveDatabases;* розриває всі встановлені зв'язки з компонентом *TIBDatabase.* Індекс зв'язаного з'єднання в списку *Databases* транзакції можна одержати за допомогою методу *function FindDatabase (db: TIBDatabase): Integer* .

Наприклад, якщо не відомо нічого, крім імені компонента, можна підійти так:

```
var i, FIndex: Integer;
...
for i := 0 to Forml.ComponentCount - 1 do
 if Forml.Components[i].Name = 'IBDatabasel'
then FIndex :=  IBTransactionl.FindDatabase(TIBDatabase(Forml. Components[i]));
...
```

З'єднання, задане за замовчуванням властивістю *DefaultDatabase,* повертає метод *function FindDefaultDatabase: TIBDatabase;*.Транзакція може мати набір параметрів, визначити які можна за допомогою властивості *property Params: TStrings;* аналогічно компоненту TIBDatabase. Прямий доступ для читання до буфера параметрів транзакції *Transaction Parameters Buffer (TPB)* типу pchar забезпечує властивість *property TPB: PChar;* Довжина буфера втримується у властивості *property TPBLength: Short;*.

Дескриптор транзакції представлений властивістю *property Handle: TISC_TR_HANDLE;*

Після того як транзакція налаштована, її можна почати, зберегти або скасувати.

Транзакція стартує за допомогою методу *procedure StartTransaction;*

При необхідності зберегти всі зроблені в рамках поточної транзакції зміни використається метод *procedure Commit;.Я*кщо виконані дії потрібно скасувати, застосовується метод *procedure Rollback;.* Для відкриття й збереження транзакції можна використати традиційну властивість property *Active: Boolean;* Після початку нової транзакції властивість *property InTransaction: Boolean;* приймає значення True, а після фіксації або відкоту - значення *False.*

При роботі із сервером *InterBase 6.0* можна використати методи *commit-Retaining і RollbackRetaining.* На відміну від стандартних операцій фіксації й відкоту транзакцій, ці методи після передачі або скасування змін залишають поточну транзакцію відкритою. Якщо сервер перевантажений і не відгукується на транзакцію, то після закінчення часу, заданого властивістю *property IdleTimer: Integer*; виконується дія, задана властивістю :
type TTransactionAction = (*taRollback, taCommit, taRollbackRetaining, taCommitRetaining);*
де *property DefaultAction: TTransactionAction;, taRollback* - відкочення транзакції; *taCommit* - фіксація транзакції; *taRollbackRetaining* - скасування змін без завершення транзакції (для сервера InterBase 6.0);, *taCommitRetaining* - фіксація змін без завершення транзакції (для сервера InterBase 6.0).

Для компонента транзакції можна налаштувати її автоматичне завершення при закритті останнього відкритого компонента, що інкапсулює набір даних, пов'язаного з тим же з'єднанням, що й транзакція. Для цього властивість
type TAutoStopAction = (saNone, saRollback, saCoramit, saRollbackRetaining, saCommitRetaining); property AutoStopAction : TAutoStopAction; не повинна мати значення *saNone*. Інші значення властивості виконують наступні дії:

- *saRollback* - відкочування транзакції;
- *saCommit* - фіксація транзакції;
- *saRollbackRetaining* - скасування змін без завершення транзакції (для сервера InterBase 6.0);
- *saCommitRetaining* - фіксація змін без завершення транзакції (для сервера InterBase 6.0).

Метод *procedure CheckAutoStop*; виконує дію, передбачена поточним значенням властивості *AutoStopAction.* Діагностика стану транзакції під час виконання здійснюється групою спеціальних методів. У випадку негативного результату всі вони генерують виключення *EiBClientError.*

Метод *procedure CheckDatabasesInList;*

перевіряє, чи є в списку *Databases* зв'язані з'єднання. Метод procedure *ChecklnTransaction;* перевіряє, чи відкрита в цей момент транзакція. Метод *procedure CheckNotlnTransaction;* перевіряє, чи закрита в цей момент транзакція. Єдиний метод-

обробник транзакції *property OnldleTimer: TNotifyEvent;* викликається після закінчення строку очікування виконання транзакції, заданого властивістю *IdleTimer.*

Оскільки компоненти *InterBase Express* використовують для одержання набору даних власний механізм, ієрархія класів-предків включає тільки обов'язковий для всіх наборів даних *TDataSet* клас *TiBCustomDataSet,* що, власне, і інкапсулює механізм доступу *InterBase Express.* Для зв'язку з базою дані компоненти InterBase Express застосовують компоненти з'єднання *TiBDatabase*. Для цього вони використовують властивість *property Database: TiBDatabase;.* Доступ до зв'язаної транзакції здійснюється через властивість *property Transaction: TIBTransaction;.* Додатково до стандартних властивостей і методів клас *TiBCustomDataSet* має властивість:

type TIBUpdateRecordTypes = set of (cusModified, cuslnserted, cusDeleted,

cusUnmodified, cusUninserted);

property UpdateRecordTypes: TIBUpdateRecordTypes;

cusModified - модифіковані записи;

cuslnserted - додані записи;

cusDeleted - вилучені записи;

cusUnmodified - немодифіковані записи;

cusUninserted - недодані записи.

Дана властивість визначає записи набору даних, на які поширюються операції кешування. Властивість *property BufferChunks: Integer;* визначає число записів, які компонент завантажує у власний локальний буфер для прискорення виконання стандартних операцій. При використанні компонентів у прикладних програмах необхідно враховувати деякі особливості. Відновлення набору даних виконується не при кожному збереженні змін. Така поведінка компонента визначається властивістю :

property ForcedRefresh: Boolean;, яке за замовчуванням має значення False.

Це прискорює роботу компонента. При необхідності виконувати відновлення даних з максимальною частотою властивості *ForcedRefresh* потрібно привласнити значення *True.*

Залежно від налаштувань компонента, з ним можна виконувати різні види операцій редагування, перелік яких міститься у властивості «тільки для читання»:

type

TLiveMode = (Imlnsert, ImModify, ImDelete, ImRefresh);

TLiveModes = set of TLiveMode; property LiveMode: TLiveModes;

Оскільки всі ці компоненти призначені для роботи із сервером, то всі вони підтримують режим кешування змін і мають відповідні властивості, методи й методи-обробникі подій.

Можливості компонентів *TIBTable, TIBQuery, TIBStoredProc, TIBUpdateSQL* мало чим відрізняються від стандартних. Для взаємодії із сервером компонента *InterBase Express*

використають два класи, які інкапсулюють важливі структури *API InterBase*. Ці структури забезпечують передачу серверу параметрів запиту й повернення результату виконання запиту.

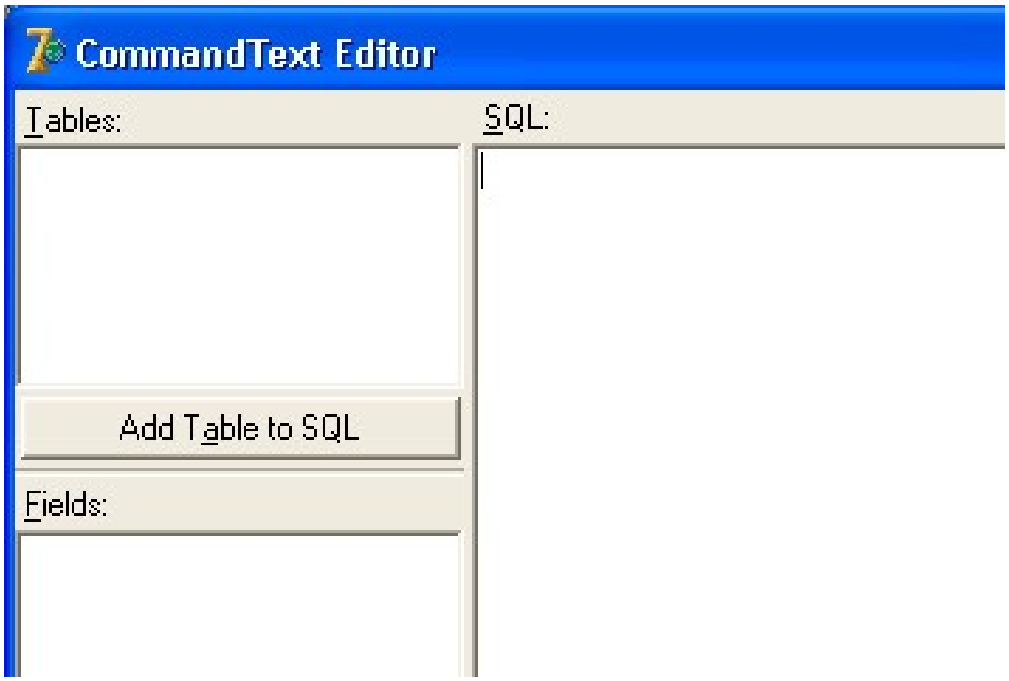


Рис. 9.19. Редактор запиту компонента TIBDataSet

Компонент *TIBSQL* призначений для швидкого виконання запитів SQL, тому не забезпечує зв'язку з компонентами подання даних. Для забезпечення швидкості виконання запиту з компонента вилучені всі додаткові механізми, що обслуговують набір даних. Фактично компонент *TIBSQL* не має відношення до звичайних компонентів доступу до даних, його безпосереднім предком є клас *Icomponent*, а не *TDataSet.* Тому він тільки передає через компонент з'єднання TiBDatabase запит серверу й одержує назад результат виконання запиту. Для підвищення швидкості переміщення по набору даних можливо тільки в прямо(односпрямований курсор). Повернені набором даних поточні значення полів містяться не у звичному наборі об'єктів полів *TField*, а в об'єкті *TIBXSQLDA*. Позаяк структура *XSQLDA* створюється сервером при виконанні запиту, істотно зменшується час відкриття набору даних компонента.

Текст запиту визначається звичайною для всіх компонентів запитів властивістю SQL. Для виконання запиту використовується також знайома властивість *EXGCSQL*. Після цього можна звертатися до створеного компонентом набору даних. Значення полів з поточного запису доступні через властивість *Fields.* Зверніть увагу, що це не об'єкти типу TFields, а структури *XSQLVAR* з області дескрипторів. Чи будуть передані значення полів у компонент, залежить від значення властивості *GoToFirstRecordOnExecute.* Доступ до області дескрипторів здійснюється через властивість Current. Перехід до наступного запису виконується методом Next. При цьому обновляється область дескрипторів запиту. Клієнтська

програма Delphi/C++Builder, що працює із сервером InterBase, має можливість відслідковувати події, що відбуваються в базі даних. Для цього використовується компонент *TIBEvents*. Він дозволяє визначити список необхідних подій і надає розробнику простий механізм відстеження виникаючих на сервері подій. Список подій визначається властивістю *property Events: TStrings*; у якому можна визначити до 15 контрольованих подій. Обрані події необхідно зареєструвати на сервері. Для цього застосовується метод *procedure RegisterEvents;*. Компонент *TIBDatabaseinfo* має більше число властивостей і методів, які містять різноманітні відомості про стан БД. Компонент дуже простий у застосуванні. Для вибору бази даних (компонента *TiBDatabase*) використовується стандартна властивість *property Database: TiBDatabase;*. У процесі роботи з базою даних властивостям компонента *TiBDatabaseinfo* передаються відповідні значення. Розробнику необхідно лише в потрібних місцях використовувати значення необхідних властивостей. Компонент *TIBSQLMonitor* дозволяє отримувати в клієнтській прикладній програмі повідомлення від сервера про виконувані їм операції. Для цього використовується метод-обробник компонента

TSQLEvent = procedure(EventText: String) of object;

property OnSQL: TSQLEvent;

Параметр *EventText* містить текст повідомлення. У компоненті з'єднання із БД можна встановити перелік подій сервера, на які буде реагувати компонент *TIBSQLMonitor*. Це робиться за допомогою властивості *TraceFiags* (див. вище). Ймовірні значення множини означають контроль за наступними операціями:

- *tfQPrepare* - підготовка запиту до виконання (виклик методу Prepare);
- *tfQExecute* - виконання запиту (виклик методу ExecSQL);
- *tfQFetch* - виклик запиту (виклик методів Open, Close);
- *tfError* - виникнення помилки;
- *tfstmt* - всі операції із запитами;
- *tfconnect* - підключення й відключення БД;
- *tfTransact* - виконання транзакцій;
- *tfBlob* - операції з даними BLOB;
- *tfService* - допоміжні операції;
- *tfMisc* - будь-які операції, не враховані перерахованими вище значеннями.

Як підсумок, набір компонентів *InterBase Express* забезпечує швидкий і ефективний доступ до баз даних на серверах *InterBase*. Для доступу до даних цих компонентів не потрібно BDE, вони використовують тільки можливості *API InterBase*. Частина компонентів забезпечує швидкий перехід зі стандартних компонентів, інкапсулюють набір даних, і повторює функціональність компонентів *TTable, TQuery, TStoredProc* і т.д. Компоненти *TIBSQL* і

TiBDataSet повністю засновані на механізмах API *InterBase*, працюють ще ефективніше, але вимагають нестандартних прийомів роботи.

Контрольні запитання

1. Чим викликана розробка технології DAO?
2. В чом сутність та особливість технології DAO?
3. В чому сутність провайдера даних для OLE DB?
4. Які особливості інтерфейсу рівня базового OLE DB?
5. Яку роль грає об'єкт джерела даних - Data Source Object (DSO) в обробці даних?
6. Якими способами споживач даних може звертатися до даних із провайдера даних?
7. Які технологічні кроки потрібні, щоб прочитати таблицю джерела даних послідовно?
8. В чому полягає роль зв'язування (Bindings) для зчитування даних з джерела?
9. Яке місце в механізмі займають OLE DB аксесорів (Accessors**)** ?
10. Для чого потрібні нумератори і як вони використовуються?
11. Як об'єкти ADO використовують OLE DB?
12. В чому призначення універсального механізму доступу до даних фірми Microsoft Data Access Components (MDAC)?
13. Поясніть призначення та компоненти механізму BDE.
14. Чому BDE не працює з SQL-сервером, якщо не встановлена клієнтська частина цього сервера або без дистрибутива DAO BDE не працюватиме з файлами MS Access?
15. Яким чином зв’язана програма написана з використанням бібліотеки VCL з BDE?
16. Опишіть особливості використання BDE для роботи з локальними базами даних та серверами.
17. В чому полягають особливості взаємодії BDE з ODBC?
18. В чому полягає призначення та область застосування технології dbGo?
19. Які характерні особливості компонент ADO технології Borland породжують різні схеми побудови прикладної програми обробки даних?
20. Опишіть особливості призначення та використання драйверів dbExpress.
21. Опишіть використання поняття наборів даних для різних технологій Borland роботи з базами даних .
22. Що таке курсор набору даних і як він реалізується різних технологій Borland з базами даних?

23. Опишіть механізм доступу до даних InterBase Express .

РОЗДІЛ 10. Технології розподіленої обробки даних

Обвуглюйся. З дияволом грай в теніс.
Згори на попіл в думах і літах.
Хай вилітає не той самий фенікс,
А зовсім інший, неймовірний птах!
(Ліна Костенко)

З початку 2000 років суттєво зріс інтерес до так званих розподілених систем, як інструмент для забезпечення ефективного управління складними економічними об’єктами. Під розподіленими системами зазвичай розуміють програмні комплекси, складові частини яких функціонують на різних комп'ютерах в мережі. Ці частини взаємодіють один з одним, використовуючи ту чи іншу технологію різного рівня - від безпосереднього використання сокетів TCP/IP до технологій з високим рівнем абстракції, таких, як RMI або CORBA. Зростання популярності розподілених систем викликане істотним посиленням вимог, які пред'являються замовником до сучасних програмних продуктів. Узагальнюючи думки експертів, можна сформулювати найважливіші з цих вимог:

- Забезпечення масштабованості систем, тобто здатність ефективно обслуговувати як малу, так і дуже велику кількість клієнтів одночасно.
- Надійність створюваних застосувань. Програмний комплекс має бути стійкий не тільки до помилок користувачів (це визначається головним чином якістю клієнтських програм), але і до збоїв в системі комунікацій. Під надійністю розуміється використання транзакцій, тобто гарантованого переходу системи в процесі функціонування з одного стійкого і достовірного стану в інший стійкий і достовірний.
- Можливість безперервної роботи протягом тривалого часу (так званий режим 24x7, тобто режим цілодобової роботи протягом тижнів і місяців).
- Високий рівень безпеки системи, під якою розуміється не тільки контроль доступності тих або інших ресурсів системи і захищеність інформації на всіх етапах функціонування, але і відстежування виконуваних дій з високою мірою достовірності.
- Висока швидкість розробки застосувань і простота їх супроводу і модифікації з використанням програмістів середньої кваліфікації.

Виявилось, що забезпечити відповідність цим вимогам, використовуючи традиційні технології - а саме, дволанкові системи “клієнт-сервер”, де сервери виступають засобом системи управління базами даних, майже неможливо. Треба зазначити , що далеко не для всіх застосувань перелічені вище вимоги є істотними. Існує і буде існувати широке коло економічних суб’єктів, для яких традиційна клієнт-серверна та інші різновиди традійних технологій дають цілком прийнятне рішення. Проте цим технологіям ми уділили увагу, тому далі розглянемо шляхи розв’язку проблем розподіленої обробки даних.

Парадигма розподілених обчислень виникла одночасно з появою комп'ютерних мереж. Прикладні програми були поділені на дві частини: перша, клієнтська, ініціювала розподілені операції, а друга, серверна, забезпечувала їхнє виконання. Така децентралізація знижувала ймовірність виникнення вузьких місць, розподіляючи робоче навантаження між декількома системами. Також забезпечувалася гнучкість дизайну прикладних програм, що була недоступна раніше, в епоху централізованих мережних вузлів (хостів). Але в такий дволанковій архітектурі були й свої обмеження. Для вирішення проблем, пов'язаних з завадостійкістю й масштабованістю, була запропонована триланкова модель, що розділяє прикладні програми на презентаційну частину (проміжну ланку, що містить бізнес-логіку) і третю ланку, безпосередньо пов'язану з даними. Така модель стала найбільш популярним способом розподілу прикладних програм. Вона дозволила зробити прикладні системи масштабованими. Основою для взаємодії між розподіленими частинами прикладної програми в ній є виклики віддалених процедур (RPC), реалізовані в сучасних операційних системах, зокрема у Windows та UNIX. Щоб звільнити розробників від виконання завдань нижнього рівня, наприклад, від перетворення даних або від їх побайтового упорядкування на різних машинах, на ринок був випущене програмне забезпечення нового рівня. Це проміжне програмне забезпечення приховує розходження між типами мережних вузлів. Воно розміщується над операційними системами й мережними службами, встановленими на цих хостах, обслуговуючи прикладні програми, які перебувають на більш високому рівні.

10.1.Різновиди технологій розподіленої обробки даних

Перші проміжні програмні продукти, наприклад DCE, ґрунтувалися на моделі процедурного програмування. Їм на зміну прийшла об’єктно-орієнтова модель, реалізована в проміжних програмних продуктах CORBA, DCOM або RMI, які є найбільш популярним програмним забезпеченням цього класу з середини 90-х років минулого століття. Спочатку обговоримо далі тільки основні можливості *DCOM, CORBA і RMI,* без поглиблення в подробиці цих технологій.

10.1.1. CORBA

CORBA являє собою засноване на відкритих стандартах вирішення для виконання розподілених обчислень. Промисловий консорціум OMG (група по розвитку стандартів керування програмними об'єктами) розробив специфікацію для *CORBA* і визначив протокол *Internet InterORB Protocol (IIOP)*, що є стандартним протоколом комунікації між диспетчерами об'єктних запитів *ORB* (*Object Request Brokers*).

Основна перевага *CORBA* полягає в тому, що й клієнти, і сервери можуть бути написані на будь-якій мові програмування. Це стає можливим, оскільки об'єкти визначаються з високим рівнем абстракції, забезпечуваним мовою визначення інтерфейсу *IDL (Interface Definition Language).* Після того, як визначення буде виконано в IDL-файлі, компілятор забезпечує його перетворення в певну мову програмування. Взаємодії між об'єктами, клієнтами й серверами обробляються за допомогою диспетчерів об'єктних запитів ORB. Якщо від розподіленої прикладної програми потрібна висока продуктивність, тоді *CORBA* є кращим вибором проміжного програмного забезпечення. Однак написання розподілених прикладних програм за допомогою *CORBA* - це дуже складна задача. Для того щоб забезпечити перетворення *IDL* на різні мови програмування, доводиться обмежуватися тими базовими концепціями, які реалізовані в підтримуваних мовах. Таким чином, *IDL* є для них чимось на зразок спільного знаменника. Для охоплення різних областей застосування *CORBA* використовує так звані «сервіси». Це сильно ускладнює специфікацію, й без того складну. Крім того, дотепер постачальниками випущений досить невеликий набір таких сервісів. Щоб забезпечити взаємодію між клієнтськими й серверними об'єктами, диспетчери об'єктних запитів повинні розміщатися по обох боках такого з'єднання.

10.1.2. RMI

Технологія виклику віддалених методів *RMI (Remote Method Invocation)* дозволяє створювати розподілені прикладні програми *Java-to-Java*. У них методи віддалених *Java*-об'єктів можуть викликатися з інших віртуальних машин *Java (JVM)*, що найбільш ймовірно перебувають на різних вузлах мережі. *Java*-програма може виконати виклик віддаленого об'єкта після одержання на нього посилання, або виконавши пошук віддаленого об'єкта в просторі імен, запропонованому *RMI*, або одержавши посилання як аргумент чи значення. Клієнт може викликати віддалений об'єкт на сервері. Цей сервер у свою чергу теж може бути клієнтом для інших віддалених об'єктів. *RMI* використовує серіалізацію об'єктів, щоб упорядкувати й переупорядкувати їхні параметри й запобігти усіканню типів, підтримуючи

справжній об’єктно-орієнтований поліморфізм. Для комунікацій використовується протокол *JRMP (Java Remote Method Protocol).* Програмування з використанням *RMI* не викликає проблем, якщо розробник набув певний досвід створення розподілених прикладних програм і програмування мовою *Java.* При цьому не потрібне використання абстрактних мов (таких як IDL) для опису віддаленого серверного об'єкта. Крім того, *RMI* підтримує розподілений механізм регенерації звільнення памяті. З іншого боку, для використання цієї технології необхідна наявність Java-машин по обидва боки з'єднання. Завдяки відносній простоті цього проміжного ПЗ, його зручно й легко використати. При цьому жодні сервіси не створюється, як це було у випадку з *CORBA.* Всі ці аспекти повинні враховуватися розробниками.

10.1.3. DCOM

Технологія *DCOM (Microsoft Distributed Component Object Model)* базується на вже досить докладно описаній технології *COM*, що дозволяє здійснювати виклики віддалених об'єктів за допомогою ланки надбудованої поверх механізму *DCE RPC*, яка взаємодіє з оперативними COM-сервісами. Сервер *DCOM* публікує свої методи для клієнтів, підтримуючи різні інтерфейси. Вони пишуться мовою *IDL*, схожою з C++. Даний компілятор *IDL* створює модулі доступу , заглушки й скелетні елементи програми так само, як це робить IDL-компілятор *CORBA*, але реєструє їх також у системному реєстрі. Крім цього створюються бібліотеки типів. Вони являють собою файли, у яких описуються віддалені об'єкти й вказується, які з них можуть запитуватися інтерфейсами, що забезпечуються механізмом *COM.* Для цього використовується протокол викликів віддалених об'єктних процедур *ORPC* (*Object Remote Procedure Call*). Специфікація *DCOM* на двійковому рівні дозволяє використовувати різні мови для програмування серверних об'єктів. Як і *RMI*, технологія *DCOM* підтримує розподілений механізм регенерації пам'яті, що звільняється віддаленими серверними об'єктами. Це досягається завдяки використанню механізму опитування, який перевіряє, чи активне підключення клієнта. На серверній стороні для клієнтів підтримується відповідний лічильник посилань. Якщо показуване їм значення скорочується до нуля, відбувається звільнення ресурсів, займаних об'єктом.

Більшість користувачів зв'язують технологію *DCOM* з операційними системами Microsoft. Стосовно реалізації цієї технології в інших операційних системах, то тут перспективи досить неясні, хоча уже анонсовані портовані версії *DCOM* для *Unix, VMS і Apple Macintosh.*

Як підсумок , можна сказати, що всі ці три технології використання проміжного ПЗ працюють за одним схожим сценарієм. Їхні відмінності проявляються, насамперед, у різних підтримуваних можливостях, а також у рівні складності. Всі вони приводять до встановлення надійного з'єднання клієнта із сервером, тому кожне з перерахованих вище проміжних ПО

придатне для використання. Через розходження в протоколах не можна здійснити виклик *DCOM*-сервера з RMI-клієнта. (Одним із кроків для вирішення цієї проблеми є звертання до механізму, що викликає RMI над IIOP, що використається при розробці із застосуванням *Enterprise JavaBeans*). При цьому встановлюється з'єднання за принципом «від точки до точки».

Варто також згадати, що дане проміжне програмне забезпечення зазвичай використовується для інтранет-прикладних програм і організувати його роботу через брандмауер досить проблематично. Для підключення до сервера за брандмауером всі ці технології передбачають механізм *HTTP*-тунелювання.

10.1.4. WEB-сервіси

Технологія інтегрованої електронної розподіленої обробки даних, яку ми далі розлянемо народилась зовсім нещодавно і є породженням епохи Internet. Спочатку *World Wide Web* була мережею документів. *Web*-сервери спілкувалися з клієнтами по протоколу *HTTP (Hypertext Transfer Protocol)* і пересилали інформацію у формі гіпертекстових документів, створених засобами мови *HTML (Hypertext Markup Language*). Такі документи відображалися в браузерах і містили посилання на інші документи, тобто Web з'явився як документо-орієнтована мережа. Проте незабаром стали з'являтися комерційні Web-сайти, що містять документи, які рекламували продукти і сервіси тих або інших компаній. Такі сайти до цього часу складають основне наповнення Web, але зараз все більша кількість комерційних сайтів підтримує транзакції з клієнтами, перетворюючись таким чином на сайти електронної комерції. Багато комерційних сайтів будуються не на основі звичайних Web-серверів, а за допомогою багатоланкової архітектури, що вимагають використання серверів прикладних програм.

Основною відмінністю звичайних Web-серверів від серверів прикладних програм є те, що останні не просто повертають документ, а можуть обробляти призначені для користувача запити і містять код, що реалізує бізнес-логіку. Зазвичай сервери прикладних програм генерують документи динамічно, залежно від вказаних користувачем параметрів. Також слід зазначити, що застосування серверів прикладних програм дозволяє створювати рішення, що масштабуються, здатні одночасно обслуговувати велику кількість транзакцій.

На сайті *WebServices.org* дано визначення :

«Веб-сервіси - це виділені в самостійний елемент, слабкозв'язані стиснуті функції, обмежені стандартними протоколами», де:

- «Виділені в самостійний елемент» означає, що реалізація цієї функції ніколи не помітна ззовні.

- «Слабкозв'язані» означає, що зміна однієї функції не вимагає зміни інших функцій, що її викликають.
- «Обмежені» означає, що існують відкриті для загального доступу описи поведінки функції, способів використання, а також її вхідних і вихідних параметрів.

Поява різноманітних мобільних пристроїв привела до того, що замість традиційних браузерів багато комерційних Web-застосувань тепер крім протоколу *HTTP* підтримують і протокол *WAP (Wireless Access Protocol)* та здатні повертати інформацію не тільки в стандарті *HTML*, але і в стандарті, що задовольняє користувачів, що звертаються до сервісів через мобільні пристрої, - *WML (Wireless Markup Language).* Природно, інформаційні системи не можуть обмежуватися простою обробкою транзакцій - наступним логічних кроком в розвитку Web стала інтеграція бізнес-процесів різних компаній. Таким чином на світ з'являється сервіс-орієнтований Web.

10.2. Сервіс-орієнтований Web

В його основі лежать дві нові технології - *SOAP (Simple Object Access Protocol)* і *XML (Extensible Markup Language).* Згідно цього сценарію Web складається з набору серверів додатків, що обмінюються інформацією у форматі *XML* по протоколу *SOAP*. Для того щоб пояснити технологію Web-сервісів необхідно з’ясувати деякі деталі про формат *XML* та протокол *SOAP*.

10.2.1. XML

XML-документ є звичайним текстовим файлом, в якому за допомогою спеціальних маркерів створюються елементи даних, послідовність і вкладеність яких визначає структуру документа і його зміст. Основною перевагою *XML* документів є те, що при відносно простому способі створення і обробки (звичайний текст може редагуватися будь-яким тестовим процесором і оброблятися стандартними *XML* аналізаторами), вони дозволяють створювати структуровану інформацію, яку добре «розуміють» комп'ютери. Це досягається за рахунок розширеного синтаксису тегів.

XML (Extensible Markup Language) - це мова розмітки, що описує цілий клас об'єктів даних, що називаються *XML*-документами. Ця мова використовується як засіб для опису граматики інших мов і контролю за правильністю складання документів. Тобто сам по собі XML не містить ніяких тегів, призначених для розмітки, він просто визначає порядок їх створення. Таким чином, якщо, наприклад, ми вважаємо, що для позначення елементу Київ в

документі необхідно використовувати тег *<city>*, то *XML* дозволяє вільно використовувати визначений нами тегі ми можемо включати в документ фрагменти, подібні наступному:
<city>Київ</city>.

Набір тегів може бути легко розширений. Нехай, якщо ми хочемо також вказати, що опис міста повинен містити опис країни, до якої він належить, то просто задаємо нові теги і обираємо порядок їх розміщення:

Простий XML- документ може виглядати так, як це показано нижче:

```
<?xml version=«1.0»?>
<list_of_items>
<item id=«1»><first/>Перший</item>
<item id=«2»>Другий <sub_item>підпункт 1</sub_item></item>
<item id=«3»>Третій</item>
<item id=«4»><last/>Останній</item>
</list_of_items>
```

Можна бачити, що цей документ дуже схожий на звичайну HTML-сторінку. Як і в *HTML*, інструкції, взяті в кутові дужки називаються тегами і служать для розмітки основного тексту документа. У *XML* існують відкриваючі, закриваючі та порожні теги (у HTML поняття порожнього тега теж існує, але спеціального його позначення не вимагається). Тіло документа *XML* складається з елементів розмітки (markup) і безпосередньо вмісту документа - даних (*content*). *XML* - теги призначені для визначення елементів документа, їх атрибутів і інших конструкцій мови. Будь-який *XML*- документ повинен завжди починатися з інструкції *<?xml?>,* усередині якої також можна задавати номер версії мови, номер кодової сторінки і інші параметри, необхідні програмі-аналізатору в процесі розбору документа.

В лістингу 10.1 показаний приклад *XML*-документа. Зверніть увагу, що документ має два розділи. У першому розділі визначається структура документа; цей розділ називається визначенням типу документа, або *DTD (Document Type Declaration*). Другий розділ містить власне дані.

Лістинг 10.1 Приклад XML-документа

```
<!DOCTYPE customer [
    <!ELEMENT customer (name, address)>
        <!ELEMENT name (firstname, lastname)>
            <!ELEMENT firstname (#PCDATA)>
            <!ELEMENT lastname (#PCDATA)>
        <!ELEMENT address (street+. city, state. zip)>
            <!ELEMENT street (#PCDATA)>
            <!ELEMENT city (#PCDATA)>
            <!ELEMENT state (#PCDATA)>
            <!ELEMENT zip (#PCDATA)>
]>
    <customer>
```

```
        <name>
            <firstname>Володимир</firstname>
            <lastname>Капустян </lastname>
    </name>
    <address>
            <street>Чорнобильська</street>
            <street>кв.15</street>
            <city>Київ</city>
            <state>К</state>
            <zip>32123-7788</zip>
    </address>
</customer>
```

DTD починається із ключового слова *DOCTYPE*, за яким вказано ім'я типу документа - customer. Далі йде опис змісту документа customer. У ньому є дві групи: *name і address*. Група name складається із двох елементів - *firstname і lastname*. Ці елементи визначені як *#PCDATA*, тобто рядки символьних даних. Нижче описується елемент *address*, що складається із чотирьох елементів: *street, city, state і zip*. Кожний із цих елементів також визначений як символьні дані. Знак + після імені елемента street указує на те, що цей елемент зобов'язаний мати мінімум одне значення, але може мати кілька значень.

Екземпляр даних типу customer, показаний у лістингу 10.2, відповідає *DTD*, тому даний документ називає *XML*-документом, припустимим по типу (*type-valid XML document*). Якби він не відповідав *DTD*, він називався б неприпустимим по типу документом (*not-type valid XML document*). Неприпустимі по типу документи можуть, проте, бути абсолютно правильними з погляду *XML*: вони просто не є припустимими екземплярами свого типу. Наприклад, якби документ у лістингу 1 містив два елементи city, він був би як і раніше правильним з погляду *XML*, але неприпустимим по типу.

Хоча *DTD* майже завжди бажаний, він не є обов'язковим в *XML*-документах. Документ, що не має *DTD*, за визначенням є неприпустимим по типу, оскільки не існує типу, щодо якого можна було б перевірити його допустимість.

Розділ *DTD* не обов'язково повинен міститися в самому документі. У лістингу *10.*2 показаний документ типу customer, у якому *DTD* завантажується з *URL http://www.somewhere.com/dtds/customer.dtd.* Перевага зовнішнього зберігання *DTD* у тім, що можна перевіряти допустимість множини документів того самого *DTD*.

Лістинг 10.2. XML-документ із зовнішнім DTD

```
<!DOCTYPE customer SYSTEM «http://www.somewhere.com/dtds/customer.dtd>

<customer>
    <name>
    <customer>
        <name>
            <firstname>Володимир</firstname>
            <1astname>Копелян </lastname>
    </name>
```

```
    <address>
            <street>Чорнобаївська 5</street>
            <street>кв.15</street>
            <city>Київ</city>
            <state>K</state>
            <zip>32123-7788</zip>
    </address>
</customer>
```

Розробник *DTD* може визначати будь-які елементи за своїм бажанням. Отже, *XML*-документи можуть розширюватися, але розширюватися стандартизованим і контрольованим способом. Відзначимо, що *DTD* є досить зручним засобом опису змінних баз даних. Поява мови і стрімке зростання її популярності в обробці даних *XML* привели до того, що остання частина стандарту SQL:2003 присвячена специфікаціями мовних засобів, що дозволяють працювати з *XML*-документами в середовищі *SQL*. У частині 14 стандарту SQL:2003 специфікується спеціальний "тип XML" (*XML type*), значеннями якого, по суті, є *XML*-документи. Для цього типу визначається ряд операцій, що забезпечують доступ до елементів значення типа XML, перетворення цих значень і т.д. Будемо сподіватися, що для читача, незнайомого зі специфікацією XML приведених тут відомостей про цю мову буде достатньо для подальшого розуміння технології Web-сервісів. Більш детальна інформація про синтаксис та використання XML міститься в численній літературі[]-[].

10.2.2. Стандарти для Web-сервісів

Після винаходу *XML* знадобилося небагато часу на створення технології розподіленої обробки даних, кардинально альтернативної бінарним технологіям *CORBA, DCOM* та *RMI*. Оскільки специфікація *XML* допускає не тільки опис даних, а й можливість трактувати як теги також і модулі програмного коду, з'явилася можливість організації віддаленого виклику, який базується не на бінарному, а на текстовому протоколі. Найважливішим елементом організації віддаленого виклику є протокол SOAP, проте сучасне розуміння цієї технології включає три стандарти:

- *SOAP (Simple Object Access Protocol)* - протокол для посилки повідомлень по протоколу *HTTP* та інших *Internet*-протоколів;
- *WSDL (Web Services Description Language)* - мови для опису програмних інтерфейсів *Web*-сервісів;
- *UDDI (Universal Description, Discovery and Integration)* - стандарт для індексації пошуку каталогу підприємств Web-сервісів.

На Рис. 10.1. показано, як ці три стандарти взаємодіють один із одним.

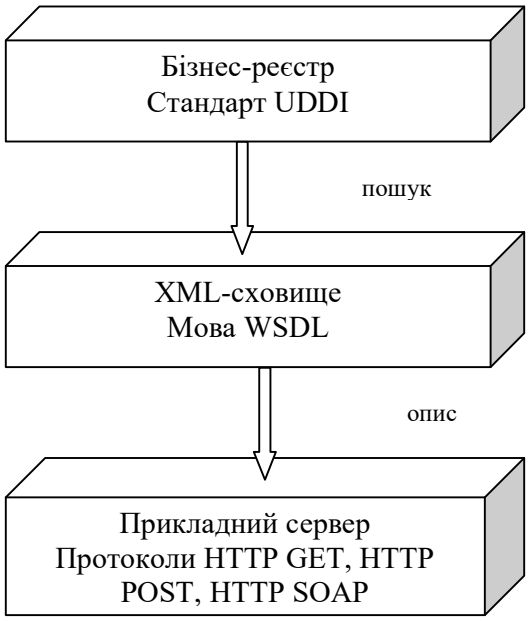


Рис.10.1. Взаємодія стандартів веб-сервісів

Сервери програм-застосувань є сховищами Web-сервісів і роблять їх доступними через протоколи *HTTP GET, HTTP POST і HTTP SOAP*.

Існуючі Web-сервіси описуються в *WSDL*-документах, які розташовуються або на сервері додатків, або в спеціальних *XML*-сховищах. *WSDL*-документ може посилатися на інші *WSDL*-документи і документи *XSD (XML Schema)*, в яких описані типи даних, використовувані *Web*-сервісами. XML-сховища використовуються для управління WSDL-документами. Усередині *WSDL*-документа знаходиться адреса (URL) Web-сервісу. Web-сервіси описані і проіндексовані в бізнес-реєстрі, що містить адреси *(URL) WSDL*-документа.

10.2.3. SOAP

Оскільки веб-сервіси мають виконуватися в гетерогенних середовищах, протоколи, використовувані для перенесення даних між функціями, повинні бути незалежні від середовища виконання. SOAP - це і є протокол, що має такі властивості.

Сам по собі він не визначає ні семантику прикладні програми, таку як модель програмування, ні семантику, що залежить від реалізації, тобто розподілений механізм регенерації звільняє памяти, що. Він скоріше задає найпростіший механізм вираження семантики прикладні програми, забезпечуючи створення моделі модульного компонування й механізмів кодування, для перетворення даних у середині модулів.

У повідомленні *SOAP* як кореневий елемент міститься тег *ENVELOPE* (конверт). Цей конверт містить два елементи:

- допоміжний заголовок;

- основна частина.

Такий заголовок може містити набір так званих рядків заголовка, які можна використати для надання інформації, наприклад, при аутентіфікації або кодуванні даних.

SOAP-повідомлення бувають двох типів: запит (*Request*) і відповідь (*Response*). Запит викликає метод віддаленого об'єкту, відповідь повертає результат виконання даного методу. В Лістингу 10.3. приведені приклади запиту і відповіді у форматі SOAP.

Лістинг 10.3. SOAP-запит

```
< SOAP—ENV: Envelope>
xmlns:SOAP-ENV=”http://(soaporg)/envelope”
SOAP-ENV:endcodingStyle=”http://(soaporg)/endcoding”
< SOAP—ENV: Body>
<m:GetLastStock xmlns:m=”None-URI” >
<symbol>MSF</symbol >
<m:GetLastStock >
< SOAP—ENV: Body>
< /SOAP—ENV: Envelope>
```

SOAP-конверт для повідомлень

Простір імен для повідомлень

SOAP-відповідь

```
< SOAP—ENV: Envelope>
xmlns:SOAP-ENV=”http://(soaporg)/envelope”
SOAP-ENV:endcodingStyle=”http://(soaporg)/endcoding/”>
< SOAP—ENV: Body>
<m:GetLastStockResponce xmlns:m=”None-URI” >
<price>50.6</price>
<value>109</value>
<m:GetLastStockResponce>
 < SOAP—ENV: Body>
< /SOAP—ENV: Envelope>
```

Результат BODY

Специфікація SOAP визначає формат кодування, який, у свою чергу, задає спосіб представлення даних в XML-форматі.

10.2.4. WSDL - Web Services Description Language

Для того, щоб застосування могли використовувати *Web*-сервіси, програмні інтерфейси останніх повинні бути детально описані - з цієї точки зору мова *WSDL* грає ту ж роль, що і мова *Interface Definition Language (IDL)* в розподілених обчисленнях. Опис може включати таку інформацію, як протокол, адресу сервера, номер використовуваного порту, список доступних операцій, формат запиту і відповіді і т.п.

У лістингу 10.4. показаний скелет опису сервісів на мові *WSDL*.

Лістинг 10.4. Опис сервісів на мові *WSDL*

```
<?xml version” 1.0” ?>
<definition name=Stocks” targetNamespace =uri
xmlns:soap =http://(soaporg)/vsdl/soap
xmlns=http://(soaporg)/vsdl/>
```

<types>
<element>…<element>
</types>
<message>…</message>
<portType>…</portType>
<binding>
<operation>
<input>…</input>
<output>…</output>
</operation>
</binding>
<servise>…<servise>
</definitions>

Як ми бачимо, опис сервісів є XML-документом, що складається з декількох елементів, зокрема з опису простору імен (namespace), опису типів і елементів, повідомлень, порту, а також можливих операцій - запитів і відповідей.

Файл, що містить опис сервісів, є достатньо складним документом, тому для його створення краще користуватися автоматичними генераторами, включеними до складу засобів розробки.

10.2.5. UDDI - Universal Description, Discovery and Integration

Завдання UDDI - надати механізм для виявлення Web-сервісів. *UDDI* задає бізнес-реєстр, в якому провайдери Web-сервісів можуть реєструвати сервіси, а розробники - шукати необхідні для них сервіси. Компанії IBM, Microsoft і Ariba створили власні *UDDI*-реєстри (незабаром ці реєстри будуть об'єднані в Web-реєстр), в одному з яких розробники можуть зареєструвати свої Web-сервіси, після чого дані будуть автоматично репліковані в інші реєстри (Рис.10.2.).

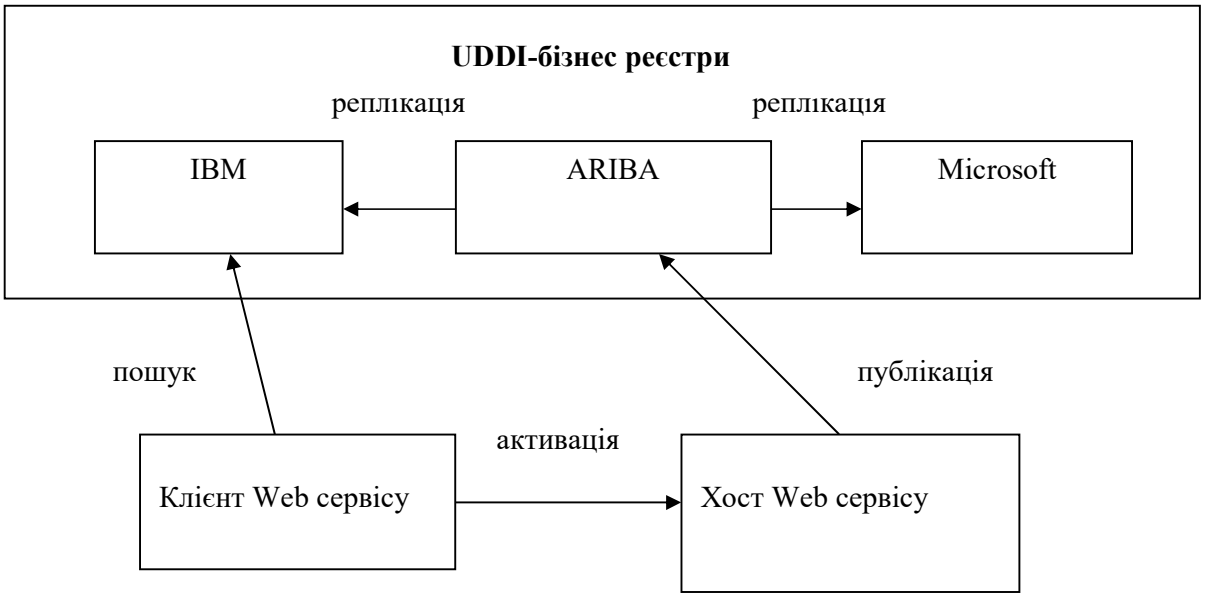


Рис.10.2. Схема взаємозв'язків реєстрів Web-сервісів

UDDI базується на елементах чотирьох типів: *Business Entity, Business Service, Binding Template і Technology Model.* Елемент Business Entity описує індустрію, що надає даний Web-сервіс. Цей елемент може включати описи категорій для даної індустрії, що полегшують детальніший пошук сервісів. *Business Service* - це клас сервісів в рамках певної галузі промисловості або послуг. Кожна галузь належить певному елементу *Business Entity*. Разом *Binding Template* і *Technology Model* визначають Web-сервіс. *Technology Model* містить абстрактний опис, а *BindingTemplate* - конкретну специфікацію сервісу. Кожен елемент *Binding Template* належить певному елементу *Business Service*, але декілька елементів *Binding Template* можуть посилатися на один елемент *Technology Model*. Бізнес-реєстр *UDDI* сам є SOAP Web-сервісом. Він підтримує операції створення, модифікації, видалення і пошуку елементів всіх чотирьох розглянутих вище типів.

10.3. Використання протоколу SOAP

SOAP являє собою протокол, що формалізує метод інтеграції клієнта й браузера Web за допомогою HTTP і XML. Найбільш популярним застосуванням SOAP є віддалений виклик процедур за допомогою мови XML. Підхід SOAP ілюструє Рис. 10.3. Як можна бачити, протокол SOAP сполучає кращі риси протоколів Web і інтеграцію прикладних програм.

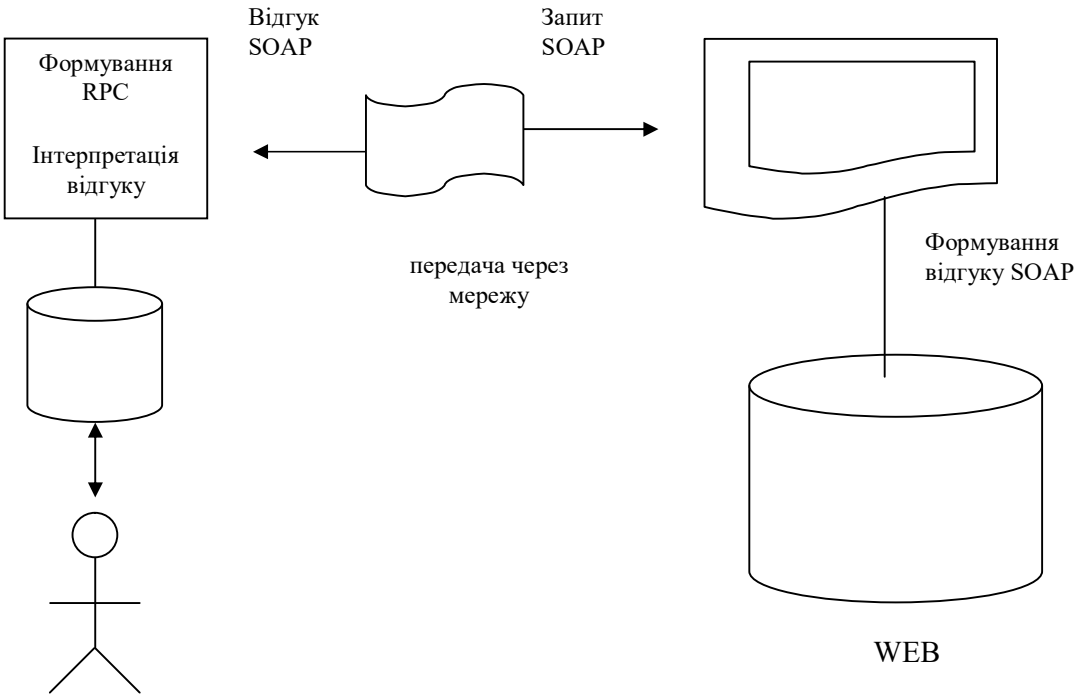


Рис.10.3. Схема взаємодії користувача з віддаленими даними за допомогою протоколу SOAP

Зокрема, протокол SOAP:

• Працює через брандмауери, оскільки підтримує протокол HTTP.

• Встановлення й керування не складніше ніж встановлення й керування Web-вузлом.

• Опирається на добре відомі технології.

• Протокол не обмежений локальними мережами, оскільки він спроектований спеціально для роботи в Інтернеті.

Але за гнучкість цього підходу доводиться платити продуктивністю. Протокол *SOAP* працює значно повільніше, ніж засоби, що використають архітектуру відкритих об'єктів. Причини цього наступні:

• Протокол *SOAP* використовує текстові повідомлення, які повинні піддаватися синтаксичному аналізу приймаючою стороною, і мають більший розмір, ніж повідомлення двійкових протоколів.

• Для встановлення зв'язку протокол *SOAP* опирається на протокол *HTTP* – досить повільний протокол .

По суті, протокол *SOAP* визначає, яким чином клієнт повинен форматувати запит, а сервер - відповідь.

10.3.1. Запит SOAP

У Лістингу 10.5 наведений запит *SOAP*. Як можна бачити, це запит *POST,* використовуваний в HTTP 1.1, з корисним навантаженням *XML*. Специфікація SOAP вимагає використати *HTTP* 1.1, *POST* і *XML*. Крім того, вона ж вимагає присутності в запиті заголовків *Content-Length і SOAPAction.*Заголовок *SOAPAction* є характерною рисою протоколу *SOAP***.** У специфікації цей заголовок визначений як *URI*, що вказує призначення запиту. Причому це необов'язково повинен бути *URL* Web-сервера (дійсно, якщо ви прочитаєте заголовок, запит ставиться до *localhost/stockq*, а не до *http://www.psol.com/xmlns/stockq).*

Для HTTP обов'язкова наявність поля заголовка з назвою *SOAPAction.* Це поле може використатися, щоб показати призначення запиту *SOAP HTTP*. Цим значенням є *URI*, яке показує ціль-адресу. *SOAP* не накладає обмежень на формат або особливості TLocalConnection, а також на можливість його дозволу. HTTP-клієнт винний використати це поле заголовка при відправленні запиту SOAP HTTP. Це поле може оброблятися серверами або брандмауерами з метою фільтрації відповідних повідомлень

Лістинг 10.5 Запит SOAP

```
POST /stockq HTTP/1.1
Host: localhost
Content-Type: text/xml
Content-Length: 422
SOAPAction: «http://www.psol.com/xmlns/stockq»
<?xml version='1.0'?>
<SOAPENV:Envelopexmlns:SOAPENV=«http//schemas.xmlsoap.org/soap/envelope/»>
<SOAP-ENV-Body>
<psol:getStock
xmlns:psol=«http://www.psol.com/xmlns/stockq»
SOAPENV:encodingStyle=«http://schemas.xmlsoap.org/soap/encodig/»>
<manufacturer>Playfield</manufacturer>
<sku>10K/sku>
</psol:getStock>
</SOAP-ENV:Body>
«C/SOAP-ENV: Envelope>
```

Віддалений виклик процедури (RPC) закодований як документ XML. Протокол *SOAP* визначає елементи *SOAP-ENV:Envelope, SOAP-ENV.Body і SOAP-ENV:Header* (у просторі імен http://schemas.xmlsoap.org/soap/envelope/).

Коренем запитів SOAP повинен служити елемент *SOAP-ENV:Envelope*. При необхідності в елемент *SOAP-ENV: Envelope* може входити елемент *SOAP-ENV:Header* і

обов'язково - елемент *SOAP-ENV.Body.* Необов'язковий елемент *SOAP-ENV:Header* у Лістингу 10.5 відсутній. Елемент *SOAP-ENV:Header* указує на розширення протоколу SOAP. Колись заголовки були уведені для підтримки транзакцій, платежів або ідентифікаційної інформації, однак наразі ці можливості використовуються рідко.

Інкапсуляція RPC-викликів є найважливішим аспектом використання *SOAP*. Із цією метою в специфікації визначаються глобальні атрибути, які називаються «стилем кодування» (encodingStyle), які використаються для серіалізації даних і, таким чином, можуть застосовуватися для їх упорядкування при *RPC*-викликах. Вони можуть з'являтися в будь-якому елементі. Дочірні елементи, що не мають свого власного атрибута *encodingStyle*, попадають під дію стилю кодування найближчого до них в ієрархії батьківського елемента. Хоча правил кодування даних, що визначають специфікацією схеми *XML*, було б досить, *SOAP* використовує підмножину цих правил.

Як і в мовах програмування в *SOAP* визначаються прості типи даних (такі як *short, int, float*), а також рядкові, перелічувані, складені типи даних і масиви. Кожний елемент складених типів також містить свій тип даних. Ці типи адаптуються зі специфікації схеми *XML .RPC*-виклики кодуються в основній частині повідомлення. Щоб викликати метод, необхідні наступні дані:

- *URI* об'єкта, що викликається.
- Назва методу.
- Додаткова сигнатура методу.
- Параметри для цього методу.
- Дані допоміжного заголовка.

URI об'єкта, що викликається не вказується в рамках конверта повідомлення. Це виконується при встановленні зв'язку по протоколу. Якщо основним використовується протокол *HTTP*, то запитуваний ним *URI* визначає цільовий об'єкт.

Кожний з методів *SOAP* містить тільки один виклик методу. Виклик, описуваний складеним типом даних, має така ж назву, що й даний метод. Для кожного параметра, що має таке ж ім'я й тип, що й зазначений параметр, існує «засіб доступу». Порядок параметрів повинен бути таким же, як і в зазначеній функції.

Елемент *SOAP-ENV.Body* містить дані *RPC*. У Лістингу 10.5 ім'я *RPC* закодоване як документ *XML: psoligetStock для getStock RPC*.

Параметри процедур *RPC* теж закодовані як елементи. Протокол SOAP пропонує стандартне кодування параметрів, що і використано в Лістингу 10.5. Однак прикладна програма може використати будь-яке кодування, що декларується атрибутом SOAP-ENV:encodingStyle. Наприклад, SOAP для Java корпорації IBM може кодувати параметри, дотримуючись правил *SOAP* або *XML*.

Кодування *SOAP* перераховує параметри як елементи *XML*, імена яких збігаються з іменами параметрів. У Лістингу 10.5. використовуються два параметри: *manufacturer* (виробник) і *sku*, значення яких рівні Playfield і 101.

10.3.2. Відгук SOAP

Сервер, розпізнавши запит SOAP, виконує синтаксичний аналіз даних, що надійшли, і витягає ім'я процедури RPC разом з її параметрами. Потім сервер виконує запит і підготовляє відгук, знов-таки у вигляді документа XML, що відсилається по протоколу HTTP.

У Лістингу10.6. наведений відгук на запит, представлений у Лістингу 10.5. Як можна бачити, це звичайний відгук HTTP з інформацією XML.

Лістинг 10.6. Відгук SOAP

```
HTTP/1.1 200 OK
CONTENT-Type:    text/xml Content-Length:555
<?xml  version=11.0'?>
<SOAP-ENV: Envelope
xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/>
<SOAP-ENV:Body>
<psol:getStockResponse
xmlns:psol='http://www.psol.com/xmlns/stockq1 SOAP-
ENV: encodingstyle='http//schemas.xmlsoap.org/soap/encoding/'>
<stockq>
<manufacturer>Playfield</manufacturer>
 <sku>10K/sku>
<available>true</available>
<level>10</level> </stockq>
</psol:getStockResponse>
 </SOAP-ENV:Body>
</SOAP-ENV: Envelope>
```

Ви вже знайомі з елементами *SOAP-ENV:Envelope і SOAP-ENV:Body*. Тілом відгуку служить елемент *XML*; однак специфікація не уточнює його ім'я. Як ім'я може використатися все що завгодно, хоча специфікація пропонує використати ім'я *RPC* із суфіксом *Response (psohgetStockResponse).*

За правилами, визначеними SOAP для закодованих структур (encode structures) імена елементів мають збігатися з іменами полів. Так, у Лістингу 10.6. використані чотири поля:

manufacturer, sku, available і level. Хоча це не показано в Лістингу 10.6., у відгук можуть входити й параметри. Вони з'являються після елемента відгуку.

10.3.3. Інформація про помилки SOAP

Як ви переконалися, у більшості випадків вміст елемента *SOAP-ENV:Body* визначається прикладною програмою. Однак *SOAP* визначає ще один специфічний елемент, призначений для виведення повідомлень про помилки: *SOAP-ENV:Fault*. Елемент *SOAP-ENV: Fault* являє собою структуру із чотирма полями, причому обов'язкові тільки перші два поля:

• *faultcode* - указує тип помилки. Припустимі значення такі*:* *SOAP-ENVtVersionMismatch, SOAP-ENV:MustUnderstand, SOAP-ENV:Client і SOAP-ENV: Server.*

• *faultstring*-містить текстовий опис помилки.

- *faultfactor* - використовується головним чином для ретрансляції повідомлень (наприклад, при використанні проксі) і являє собою URI, що вказує адресу вузла, на якому відбувся збій запиту при ретрансляції.
- *detail* - призначений для встановлення помилок, специфічних для прикладної програми.

10.4. Приклади реалізації розподіленої обробки

Розглянуті вище загальні механізми розподіленої обробки даних для реальної реалізації потребують підтримки на рівні технологій розробки програмних засобів.

10.4.1 Триланкова прикладна програма в середовищі Delphi/C++Builder.

Фірма Borland розробила технологію розробки прикладних програм розподіленої обробки даних. Архітектура такої багатоланкової технології приводиться на Рис.10.4. Такі програми можна розробляти у візуальному середовищі Delphi/C++Builder. Як варіант розглянемо складові частини триланкової розподіленої прикладної програми Як вже згадувалося, в Delphi доцільно розробляти клієнтську частину триланкової прикладної програми й ПЗ проміжного шару - сервер прикладних програм. Частини триланкових прикладних програм розробляються з використанням компонентів *DataSnap*, а також деяких інших спеціалізованих компонентів, що забезпечують функціонування клієнта. Для доступу до даних застосовується одна із чотирьох технологій, реалізованих в Delphi. Розробку триланкових прикладних програм доцільно вести, використовуючи в середовищі розробки групу проектів замість одиночних проектів. Для цього використовується утиліта *Project Manager (меню View | Project Manager).*

Для передачі даних між сервером прикладних програм і клієнтами використовується інтерфейс *AppServer,* надаваний віддаленим модулем даних сервера прикладних програм. Цей інтерфейс використають компоненти-провайдери *TDataSetProvider* на боці сервера й компоненти *TClientDataSet* на боці клієнта.

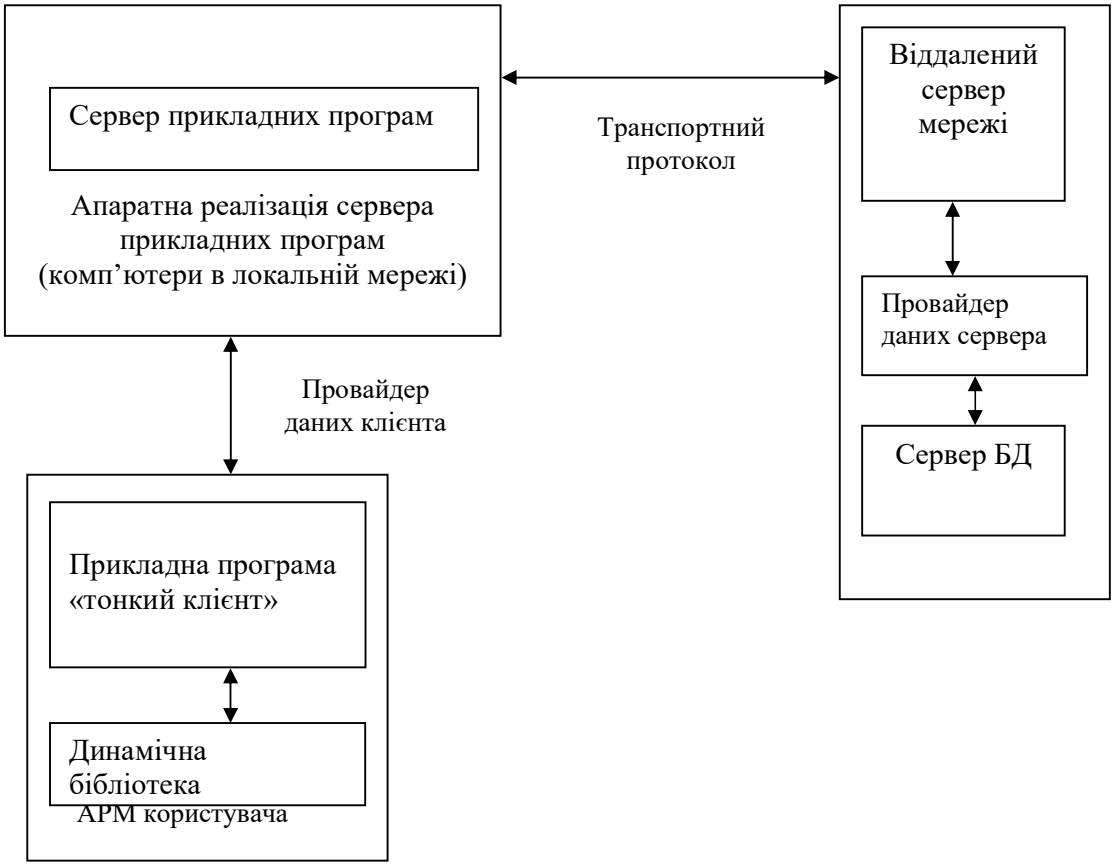


Рис.10.4. Багатоланкова архітектура розподіленої обробки БД

Таким чином, багатоланкова прикладна програма БД складається з наступних компонент:

- »тонких» клієнтських прикладних програм, що забезпечують лише передачу, подання, редагування й найпростішу обробку даних;
- одного або декількох ланок ПЗ проміжного шару (сервер прикладних програм), які можуть функціонувати як на одному комп'ютері, так і разподілено - у локальній мережі;
- сервера БД (Oralce, Sybase, MS SQL, InterBase і т.д.), що підтримує функціонування бази даних і обробляє запитів.

Більш проста триланкова модель містить наступні елементи:

- «тонкі» клієнти;
- сервер прикладних програм;
- сервер БД.

Далі ми будемо розглядати саме триланкову модель. У середовищі розробки Delphi/C++Builder є набір інструментів і компонентів для створення клієнтського ПЗ й ПЗ проміжного шару. Сервер прикладних програм взаємодіє із сервером БД, використовуючи одну з технологій доступу до даних, реалізованих в Delphi/C++Builder. Це технології *ADO, BDE, InterBase Express і dbExpress.* Розробник може обрати найбільш прийнятну технологію, виходячи з поставленої задачі й параметрів сервера БД.

Віддалені клієнтські додатки створюються з використанням спеціального набору компонентів, об'єднаних загальною назвою *DataSnap*. Ці компоненти інкапсулюють стандартні транспорти (*DCOM, HTTP, сокети*) і забезпечують з'єднання віддаленого клієнтського прикладної програми із сервером прикладної програми. Також компоненти DataSnap забезпечують доступ клієнта до функцій сервера прикладних програм за рахунок використання інтерфейсу *AppServer.* Важливу роль при розробці клієнтських прикладних програм грає компонент, що інкапсулює клієнтський набір даних. Його реалізації також залежать від технологій доступу до даних. Поряд з перерахованими вище перевагами, наявність додаткової ланки - сервера прикладних програм - дає деякі додаткові переваги, які можуть бути досить істотною підмогою з погляду підвищення надійності й ефективності системи. Позаяк найчастіше клієнтські комп'ютери - це досить слабкі машини, реалізація складної бізнес-логіки на боці сервера дозволяє істотно підвищити швидкодію системи в цілому. І не тільки за рахунок більш потужної техніки, але й за рахунок оптимізації виконання однорідних запитів користувачів. Наприклад, при надмірному завантаженні сервера, сервер прикладних програм може самостійно обробляти запити користувачів (ставити їх у чергу або скасовувати) без додаткового завантаження сервера БД. Наявність сервера прикладних програм підвищує безпеку системи, тому що можна організувати тут авторизацію користувачів, та й будь-які інші функції безпеки без прямого доступу до даних. Крім того, можна використати захищені канали передачі даних, наприклад HTTPS.

10.4.2. Сервер прикладних програм

Сервер прикладних програм інкапсулює більшу частину бізнес-логіки розподіленого прикладної програми й забезпечує доступ клієнтів до бази даних. Основною частиною сервера прикладних програм є віддалений модуль даних. По-перше, подібно звичайному модулю даних він є платформою для розміщення невізуальних компонентів доступу до даних і компонентів-провайдерів. Розміщені на ньому компоненти з'єднань, транзакцій і компонента, що інкапсулюють набори даних, забезпечують триланкову прикладну програму

зв'язком із сервером БД. Це можуть бути набори компонентів для технологій ADO, BDE, InterBase Express, dbExpress.

По-друге, віддалений модуль даних реалізує основні функції сервера прикладних програм на основі надання клієнтам інтерфейсу IAppServer або його нащадка. Для цього віддалений модуль даних повинен містити необхідну кількість компонентів-провайдерів TDataSetProvider. Ці компоненти передають пакети даних клієнтській прикладній програмі, а точніше компонентам TdientDataSet, а також забезпечують доступ до методів інтерфейсу.

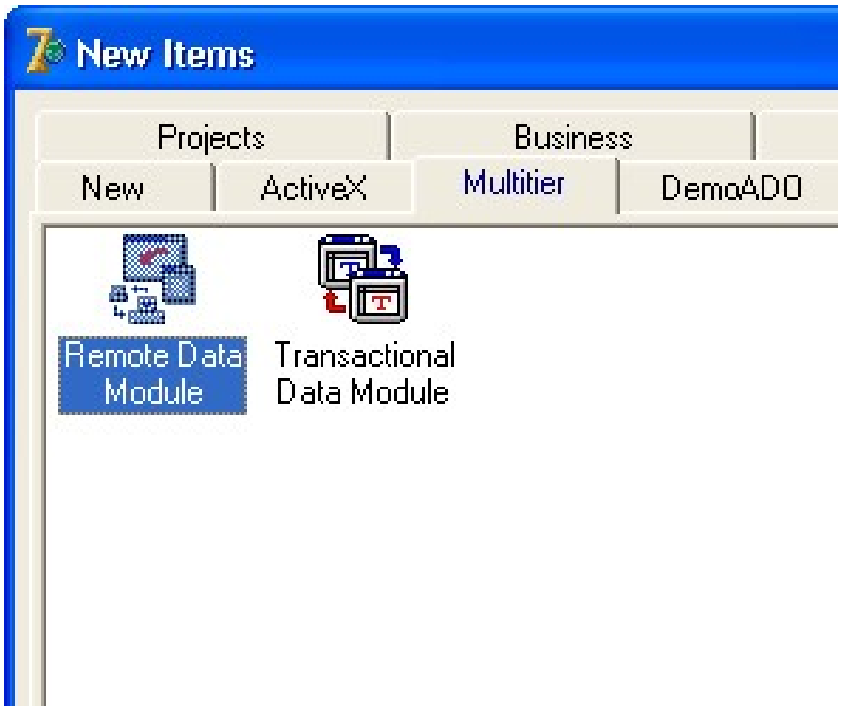


Рис.10.5. Вибір віддалених модулів даних у Репозиторії Delphi

До складу Delphi входять віддалені модулі даних. Для їхнього створення використовують сторінки *Multitier, WebSnap і WebServices* Репозиторія Delphi (Рис.10.5.).

- *Remote Data Module* - віддалений модуль даних, що інкапсулює сервер автоматизації. Використовується для організації з'єднань через *DCOM, HTTP*, сокети.
- *Transactional Data Module* - віддалений модуль даних, інкапсулюючий сервер *MTS (Microsoft Transaction Server).*
- *SOAP Server Data Module* - віддалений модуль даних, інкапсулючий сервер *SOAP (Simple Object Access Protocol).*
- *WebSnap Data Module* - віддалений модуль даних, що використовує Web-служби й Web-браузер як сервер.

Крім віддаленого модуля даних невід'ємною частиною сервера прикладних програм є компоненти-провайдери *TDataSetProvider*. Із кожним компонентом, що інкапсулируює набір даних, призначених для передачі клієнтові, у модулі даних має бути зв'язаний компонент-провайдер

10.4.3. Клієнтська прикладна програма

Клієнтська прикладна програма у триланковій моделі повинн мати лише мінімально необхідний набір функцій, що делегують більшість операцій ПЗ обробці даних серверу прикладних програм. У першу чергу віддалена клієнтська прикладна програма має забезпечити з'єднання із сервером прикладних програм. Для цього використовуються компоненти з'єднань DataSnap:

- *TDCOMConnection* - використовує DCOM;
- *TSocketConnection* - використовує сокети Windows;
- *TWebConnection* - використовує HTTP.
- Компоненти з'єднання DataSnap надають інтерфейс *IAppServer*, використовуваний компонентами-провайдерами на стороні сервера й компонентами *TClientDataSet* на боці клієнта для передачі пакетів даних.
- Для роботи з наборами даних використаються компоненти *TClientDataSet*, що працюють у режимі кешування даних.
- Для подання даних і створення користувацького інтерфейсу в клієнтському ПЗ застосовуються стандартні компоненти зі сторінки *Data Controls* Палітри компонентів.

10.4.4. Механізм віддаленого доступу до даних DataSnap

Для передачі пакетів даних між компонентом-провайдером і клієнтським набором даних (див. Рис. 10.3) повинен існувати якийсь транспортний канал, що забезпечує фізичну передачу даних. Для цього можуть використатися різноманітні транспортні протоколи, підтримувані операційною системою. Різні типи з'єднань, що дозволяють налаштувати транспорт і почати передачу й прийом даних, інкапсулювані в декількох компонентах *DataSnap.* Для створення з'єднання з тим або іншим транспортним протоколом розроблювачеві досить перенести відповідний компонент на форму й правильно налаштувати кілька властивостей. Нижче розглядаються варіанти налаштування транспортних протоколів для компонентів, що використають *DCOM, сокети TCP/IP, http*.

10.4.5. Компонент TDCOMConnection

Компонент *TDCOMConnection* надає транспорт на основі технології *Distributed COM* і застосовується в основному для організації транспорту в рамках локальної мережі. Для налаштування з'єднання *DCOM* у першу чергу необхідно задати ім'я комп'ютера, на якому функціонує сервер прикладних програм. Для компонента *TDCOMConnection* це повинен бути зареєстрований сервер автоматизації. Ім'я комп'ютера визначається властивістю *property ComputerName: string;* **.** Якщо воно задано правильно, у списку властивості *property*

ServerName: string; в Інспекторі об'єктів можна вибрати один з доступних серверів. При виборі сервера також автоматично заповнюється властивість *property ServerGUID: string;* **.** Причому для успішного з'єднання клієнта із сервером прикладних програм обидві властивості мають бути задані в обов'язковому порядку. Тільки ім'я сервера або тільки його *GUID* не забезпечать правильний доступ до віддаленого об'єкта *COM.* Відкриття й закриття з'єднання здійснюється властивістю *property Connected: Boolean;* **,** або методами *procedure Open/procedure Close* відповідно. Для організації передачі даних між клієнтом і сервером компонент *TDCOMConnection* надає інтерфейс *IAppServer property AppServer: Variant;,* який також може бути отриманий методом *function GetServer: lAppServer; override;.* Властивість *property ObjectBroker: TCustomObjectBroker;* дозволяє використати екземпляр компонента TsimpleObjectBroker для одержання списку доступних серверів ПЗ час виконання.

Методи-обробники компонента *TDCOMConnection* представлені в табл. 10.1.

Таблиця 10.1. Методи-обробники подій компонента TDCOMConnection

Оголошення	**Опис**
property Af terConnect: TNotifyEvent;	Викликається після встановлення з'єднання
property AfterDisconnect: TNotifyEvent;	Викликається після розриву з'єднання
property BeforeConnect: TNotifyEvent;	Викликається перед установленням з'єднання
property BeforeDisconnect: TNotifyEvent;	Викликається перед розривом з'єднання
type TGetUsernameEvent = procedure (Sender : TOb j ect ; var Username: string) of object; property OnGetUsername : TGetUsernameEvent ;	Викликається безпосередньо перед появою діалогу віддаленої авторизації користувача. Для цього властивість LoginPrompt повинне мати значення True. Параметр Username може містити ім'я користувача за замовчуванням, що з'явиться в діалозі
type TLoginEvent = procedure (Sender: TOb j ect; Username, Password: string) of object; property OnLogin: TLoginEvent;	Викликається після відкриття з'єднання, якщо властивість LoginPrompt має значення True. Параметри Username і Password містять ім'я користувача й пароль, уведені при авторизації

10.4.6. Компоненти TSocketConnection та TSimpleObjectBroker

Компонент *TSocketConnection* забезпечує з'єднання клієнта із сервером прикладних програм за рахунок використання сокетів *TCP/IP*. Для успішного відкриття з'єднання на боці

сервера повинен працювати сокет-сервер (прикладна програма *ScktSrvr.exe*, Рис. 20.4). Для успішного з'єднання властивість *property Host: String*; повинна містити ім'я комп'ютера сервера. Допоміжні компоненти - брокери з'єднань. До складу компонентів *DataSnap* входить ряд додаткових компонентів, що полегшують роботу із з'єднаннями віддалених клієнтів із сервером прикладних програм. Розглянемо їх.

Компонент *TSimpleObjectBroker* інкапсулює список серверів, доступних для клієнтів даного багатоланкового розподіленого прикладної програми. Список серверів створюється на етапі розробки. При необхідності (відключення сервера, його перевантаження й т.д.) компонент з'єднання клієнтського ПЗ може використати один із запасних серверів зі списку компонента *TsimpleObjectBroker* безпосередньо під час виконання. Для цього необхідно заповнити список серверів компонента *TSimpleobjectBroker* і вказати посилання на нього у властивості *objectBroker* компонента з'єднання (див. вище). І тоді при «перевідкритті» з'єднання ім'я сервера буде запитуватися зі списку компонента *TsimpleobjectBroker.* Список серверів визначається властивістю *property Servers: TServerCollection;* На етапі розробки список серверів заповнюється спеціалізованим редактором (Рис.10.6.), що викликається при клацанні на кнопці властивості в Інспекторі об'єктів.

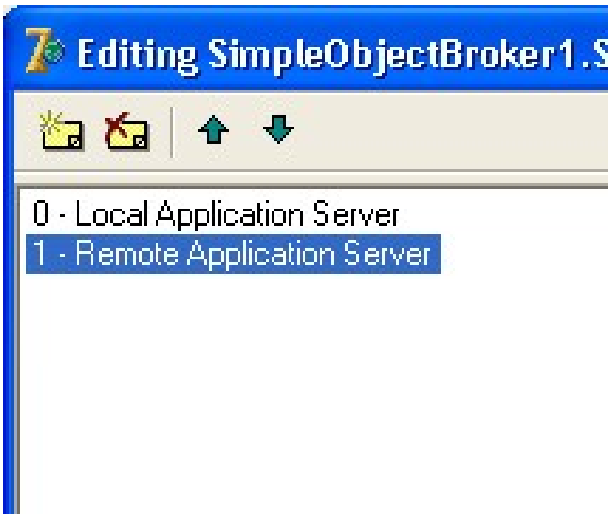


Рис. 10.6. Редактор списку серверів компонента *TSimpleObjectBroker*

Властивість *servers* являє собою колекцію об'єктів класу *TServeritem*. Цей клас має кілька властивостей, що дозволяють описати основні параметри сервера (табл. 10.2). При використанні в з'єднанні значення цих властивостей підставляються у відповідні властивості компонента з'єднання.

Таблиця 10.2. Властивості класу *TServeritem*

Оголошення	**Опис**
property ComputerName: string;	Ім'я комп'ютера, на якому функціонує сервер

property DisplayName: String;	Містить ім'я сервера для подання в списку серверів
property Enabled: Boolean;	Управляє доступністю запису про сервер для вибору при підключенні. При значенні True компоненти з'єднань можуть використати даний запис списку для підключення
property HasFailed: Boolean;	Після невдалої спроби використати даний запис списку при підключенні властивості привласнюється значення True і надалі ця запис не використовується
property Port: Integer;	Містить номер порту, використовуваного при підключенні до сервера

Крім списку серверів компонент має лише кілька допоміжних властивостей і методів. Метод *function GetComputerForGUID(GUID: TGUID): string; override;* повертає ім'я комп'ютера, на якому зареєстрований сервер з GUID, заданим параметром. Метод *function GetComputerForProgID(const ProgID): string; override;* повертає ім'я комп'ютера, на якому зареєстрований сервер з ім'ям, заданим параметром *Progio.* Властивість *property LoadBalanced: Boolean;* управляє вибором сервера зі списку. При значенні *True* запис про сервер вибирається випадковим чином, інакше для з'єднання пропонується перший доступний запис про сервер.

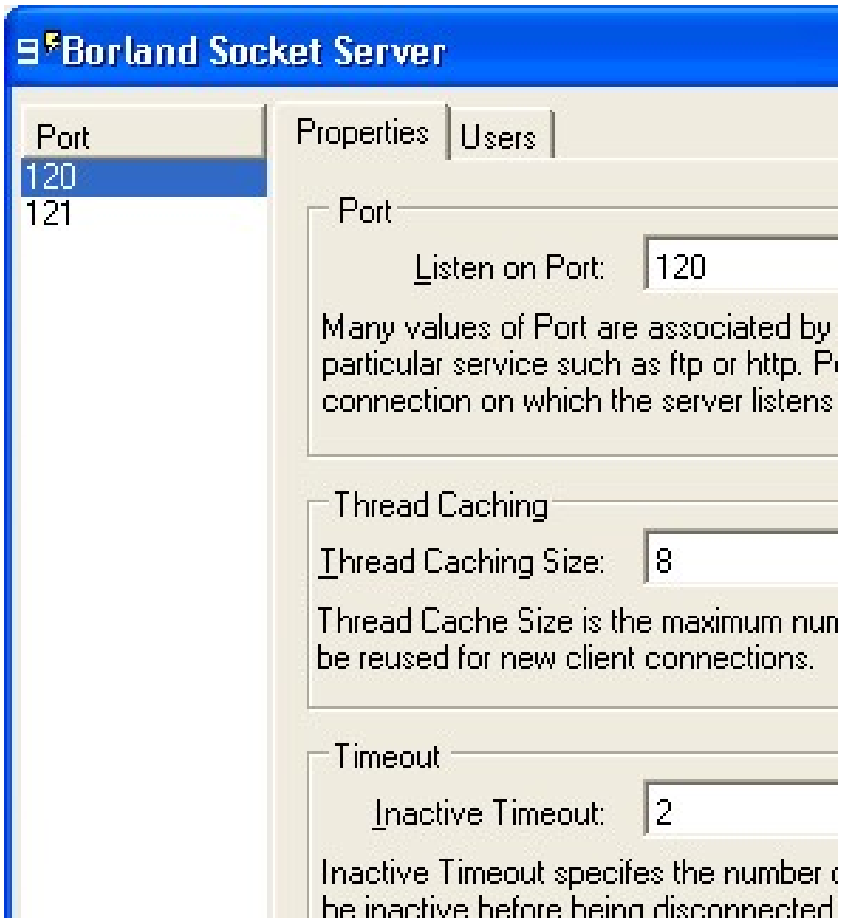


Рис. 10.7. Сокет-сервер *ScktSrvr.exe*

Додатково, властивість *property Address: String;* повинна містити IP-адресу сервера. Для відкриття з'єднання повинні бути задані обоє цих властивості. Властивість *property Port: Integer;* встановлює номер використовуваного порту. За замовчуванням це порт 211, але розроблювач може змінити порт, наприклад, для використання різними категоріями користувачів або для створення захищеного каналу. Після правильного вибору комп'ютера в списку властивості *property ServerName:string;* в Інспекторі об'єктів з'являється перелік доступних серверів автоматизації. І після вибору сервера властивість *property ServerGUID: string;* яке містить ім'я комп'ютера *GUID* зареєстрованого сервера, визначається автоматично, хоча його можна задати й вручну. Метод *function GetServerList: OleVariant; virtual;* повертає список зареєстрованих серверів автоматизації. Відкриття й закриття з'єднання здійснюється властивістю *property Connected: Boolean,* або методами *procedure Open;* та *procedure Close;* відповідно.

Канал сокета *TCP/IP* може бути зашифрований. Для цього використовується властивість *property InterceptName: string;* ,що містить програмний ідентифікатор об'єкта *COM*, який забезпечує шифрування/дешифрування даних у каналі, і властивість *property InterceptGUID: string;,* яка містить ім'я комп'ютера *GUID* цього об'єкта.

Цей об'єкт *COM* перехоплює дані в каналі й здійснює їхню обробку, передбачену власним програмним кодом. Це може бути шифрування, стиснення, обробка шумів і т.д.

Природно, на боці сервера повинен бути зареєстрований об'єкт *COM*, що виконує зворотну операцію. Для цього також використовується сокет-сервер (Рис. **10.8.**). Рядок *Interceptor* на сторінці повинна містити ім'я комп'ютера *GUID* об'єкта-перехоплювача *COM*.

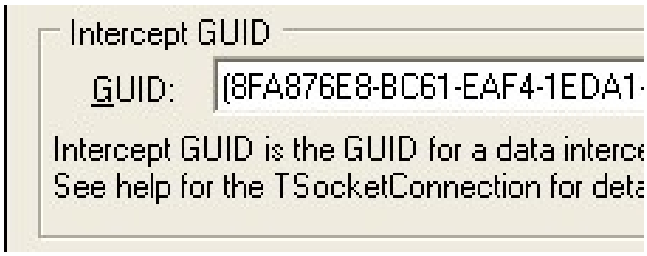


Рис. 10.8. Реєстрація об'єкта-перехоплювача COM у сокет-сервері

Метод *function GetlnterceptorList: OleVariant; virtual;* повертає список зареєстрованих на сервері об'єктів-перехоплювачів. Для організації передачі даних між клієнтом і сервером компонент TSocketConnection надає інтерфейс *IAppServer property AppServer: Variant;* який також може бути отриманий методом *function GetServer: lAppServer; override;*

Властивість *property ObjectBroker: TCustomObjectBroker;* дозволяє використати екземпляр компонента *TSimpieObjectBroker* для одержання списку доступних серверів під час виконання (див. нижче). Методи-обробники подій компонента *TSocketConnection* повністю збігаються з методами-обробникниками компонента *TDCOMCon***n***ection* (див. табл. 10.1).

10.4.7. Компонент TWebConnection

Компонент *TWebConnection* надає клієнтові з'єднання на основі транспорту *HTTP*. Для роботи компонента на клієнтському комп'ютері повинна бути зареєстрована бібліотека *wininet.dll*. Звичайно це не вимагає спеціальних зусиль, тому що цей файл уже є в системній папці *Windows*, якщо на комп'ютері встановлений *Internet Explorer*. На комп'ютері сервера має бути інстальований *Internet Information Server* версії не нижче 4.0 або *Netscape Enterprise версії* не нижче 3.6. Перераховане ПЗ забезпечує доступ компонента *TWebConnection* до динамічної бібліотеки *HTTPsrvr.dll*, що також повинна перебувати на сервері. Наприклад, якщо файл *HTTPsrvr.dll* розташований у папці *Scripts US* 4.0 на Web-сервері *www.someserver.com***,** то властивість *property URL: string;* повинна містити наступне значення*: http://someserver.com/scripts/httpsrvr.dll*

Якщо URL заданий вірно й сервер налаштований правильно, то в списку властивості

property ServerName: string; в Інспекторі об'єктів з'являється перелік зареєстрованих серверів. Ім'я одного з них повинне міститися у властивості *ServerName*. Після вибору імені сервера у властивості *property ServerGUID: string*; автоматично з'являється *GUID* сервера. Властивості *property UserName: string*; та property *Password: string;* при необхідності можуть містити ім'я й пароль користувача, які будуть використані при авторизації. Властивість *property Proxy:* string; містить ім'я використовуваного проксі-сервера. У заголовок повідомлень *HTTP* можна помістити ім'я прикладної програми. Для цього використовується властивість *property Agent: string;*. З'єднання відкривається й закривається за допомогою властивості property *Connected: Boolean;* Аналогічні операції виконують методи *procedure Open; procedure Close*;. Доступ до інтерфейсу *IAppServer* надає властивість *property AppServer*: *Variant*;, або метод *function GetServer: IAppServer; override;* .Список доступних з'єднанню серверів прикладних програм повертає метод *function GetServerList*: *OleVariant; virtual;*. Властивість *property ObjectBroker: TCustomObjectBroker;* дозволяє використати екземпляр компонента *TSimpieObjectBroker* для одержання списку доступних серверів під час виконання. Методи-обробники подій компонента *TWebConnection* повністю збігаються з методами-обробниками компонента *TDCOMConnection* (див. табл. 10.1).

10.4.8. Провайдери даних

Компонент-провайдер *TDataSetProvider* являє собою міст між набором даних сервера прикладних програм і клієнтським набором даних. Він забезпечує формування й передачу пакетів даних клієнтському прикладній програмі й прийом від нього зроблених змін. Всі необхідні операції компонентів виконує автоматично. Розроблювачеві необхідно лише розмістити компонент TDataSetProvider і зв'язати його з набором даних сервера прикладних програм. Для цього призначена властивість *property DataSet: TDataSet; .*Якщо з'єднання в клієнтському прикладній програмі налаштовано правильно (див. вище), то в списку вибору властивості *Provider*r*Name* компонента *TClientDataSet* в Інспекторі об'єктів з'являються імена всіх компонентів-провайдерів сервера прикладних програм. Якщо зв'язати клієнтський набір даних з компонентом-провайдером, а потім відкрити його, у клієнтський набір даних будуть передані записи з набору даних сервера прикладних програм, зазначеного у властивості *DataSet* компонента-провайдера *TDataSetProvider.* Компонент також містить властивості, що допомагають налаштувати процес обміну даними. Властивість *property ResolveToDataSet: Boolean;* управляє передачею даних від клієнта серверу БД. Якщо вона має значення True, всі

зміни передаються в набір даних сервера прикладних програм, визначений властивістю DataSet. Інакше зміни спрямовуються прямо серверу БД. Якщо сервер прикладних програм не повинен відображати зроблені клієнтом зміни, то властивості *ResolveToDataSet* можна привласнити значення False, що прискорить роботу прикладної програми.

Властивість *property Constraints: Boolean;* управляє передачею обмежень серверного набору даних клієнтському. Якщо властивість має значення True, обмеження передаються.

Властивість *property Exported: Boolean;* дозволяє використати в клієнтському наборі даних інтерфейс *IAppServer.* Для цього властивість повинна мати значення *True*.

Параметри компонента-провайдера визначаються властивістю

type

TProviderOption = (poFetchBlobsOnDemand, poFetchDetailsOnDemand, poIncFieldProps, poCascadeDeletes, poCascadeUpdates, poReadOnly, poAllowMultiRecordUpdates, poDisableInserts, poDisableEdits, poDisableDeletes, poNoReset, poAutoRefresh, poPropogateChanges, poAllowCoinmandText, poRetainServerOrder);

TProviderOptions = set of TProviderOption;

Набір параметрів властивості визначається присвоєнням елементам значення *True.*

property Options: TProviderOptions; poFetchBlobsOnDemand - включає передачу в клієнтський набір даних значень полів типу BLOB. По умовчанню ця можливість, відключена для прискорення роботи; Властивісь *poFetchDetailsOnDemand* - включає передачу в клієнтський набір даних підлеглих записів для відношення «один-до-багатьох». За замовчуванням ця можливість відключена для прискорення роботи; , а *poIncFieldProps* - включає передачу в клієнтський набір даних декількох властивостей для об'єктів полів: *Alignment, DisplayLabel, DisplayWidth, Visible, DisplayFormat, EditFormat, MaxValue, MinValue, Currency, EditMask, DisplayValues;*. Властивість *poCascadeDeletes* - включає автоматичне видалення підпорядкованих записів у відношенні «один-до-багатьох» на стороні сервера, якщо головний запис був віддалений у клієнтському наборі даних;. Властивість *poCascadeUpdates* - включає автоматичне відновлення підлеглих записів у відношенні «один-до-багатьох» на стороні сервера, якщо головний запис був змінений у клієнтському наборі даних; .

Властивості перелічені далі впливають наступним чином:

poReadOnly - включає режим «тільки для читання» для набору даних сервера;

*poAllowMultiRecordUpd*ates - включає режим внесення змін відразу в кілька записів одночасно. Інакше всі записи змінюються послідовно, одна за однією;

poDisableInserts - забороняє клієнтові вносити в набір даних сервера нові записи;

poDisableEdits - забороняє клієнтові вносити зміни в набір даних сервера ;

poDisableDeletes - забороняє клієнтові видаляти записи в наборі даних сервера;

poNoReset - забороняє відновлення набору даних сервера перед передачею записів клієнтові (перед викликом методу *AS_GetRecords* інтерфейсу *IAppServer)*;

poAutoRefresh - включає автоматичне відновлення записів клієнтського набору даних. За замовчуванням ця можливість відключена для прискорення роботи;

poPropogateChanges - якщо в методах-обробниках *BeforeUpdateRecord* або *AfterUpdateRecord* клієнтського набору даних були зроблені додаткові зміни, те після їхнього запису в наборі даних сервера, зміни знову спрямовуються клієнтові для відновлення запису. У включеному стані ця можливість дозволяє повністю контролювати збереження змін на сервері;

poAllowCommandText - дозволяє змінювати текст запиту SQL, імена збережених процедур або таблиць у компоненті набору даних на сервері прикладних програм;

poRetainServerOrder - включає заборона на зміну порядку сортування записів клієнтом. Якщо цей параметр відключити, можливі помилки відображення набору даних, що проявляються в появі подвійних записів.

Методи-обробники компонента-провайдера даних представлені в табл. 10.4.

Таблиця 10.4**.** Методи-обробники подій компонента TDataSetProvider

Оголошення	**Опис**
property Af terApplyUpdates: TRemoteEvent;	Викликається після збереження змін, переданих від клієнта, у наборі даних сервера
property AfterExecute: TRemoteEvent;	Викликається після виконання запиту SQL або збереженої процедури на сервері
property AfterGetParams: TRemoteEvent;	Викликається після того, як компонент-провайдер сформував набір параметрів набору даних сервера для їхньої передачі клієнту
property AfterGetRecords: TRemoteEvent;	Викликається після того, як компонент-провайдер сформував пакет даних для передачі набору даних сервера клієнтові
property AfterRowRequest: TRemoteEvent ;	Викликається після відновлення поточного запису клієнта компонентом-провайдером
property AfterUpdateRecord: TAf terUpdateRecordEvent ;	Викликається відразу після відновлення одиничного запису на сервері
property Bef oreApplyUpdates: TRemoteEvent ;	Викликається перед збереженням змін, переданих від клієнта, у наборі даних сервера
property BeforeExecute: TRemoteEvent;	Викликається перед виконанням запиту SQL або збереженої процедури на сервері

property BeforeGetParams: TRemoteEvent ;	Викликається перед тим, як компонент-провайдер сформував набір параметрів набору даних сервера для їхньої передачі клієнтові
property BeforeGetRecords: TRemoteEvent ;	Викликається перед тим, як компонент-провайдер сформував пакет даних для передачі набору даних сервера клієнтові
property BeforeRowRequest: TRemoteEvent ;	Викликається перед відновленням поточного запису клієнта компонентом-провайдером
property BeforeUpdateRecord: TBeforeUpdateRecordEvent;	Викликається безпосередньо перед відновленням одиничного запису на сервері
property OnDataRequest: TDataRequestEvent;	Викликається при обробці запиту на одержання даних клієнтом
property OnGetData: TProviderDataEvent;	Викликається після одержання даних від набору даних сервера, але перед їхнім відправленням клієнтові
property OnGetDataSetProperties: TGetDSProps;	Викликається при створенні структури параметрів набору даних сервера для їхньої передачі клієнтові
property OnGetTableName: TGetTableNameEvent;	Викликається при одержанні компонентом-провайдером імені таблиці, що підлягає відновленню
property OnUpdateData: TProviderDataEvent ;	Викликається при збереженні змін у наборі даних сервера
property OnUpdateError: TResolverErrorEvent;	Викликається при виникненні помилки збереження змін у наборі даних сервера

10.4.9.Компоненти TLocalConnection та TSharedConnection

Компонент *TLocalConnection* використовується локально для одержання доступу до існуючих компонентів-провайдерів. Властивість *property Providers[const ProviderName: string]: TCustomProvider* містить посилання на всі компоненти-провайдери, розміщені з компонентом TLocalConnection на одній формі. Індексація в списку здійснюється ПЗ по імені компонента-провайдера. Загальне кількість компонентів-провайдерів у списку повертає властивість property *ProviderCount: Integer;*. Крім цього, за допомогою компонента TLocalConnection можна одержати доступ до інтерфейсу *IAppServer* локально. Для цього використовується властивість property *AppServer: IAppServer;* або метод *function GetServer: IAppServer; override.*

Якщо інтерфейс *IAppServer* віддаленого модуля даних має метод, що повертає посилання на аналогічний інтерфейс іншого віддаленого модуля даних, то перший модуль називається головним, а інший - дочірнім. Компонент *TSharedConnection* використовується для з'єднання клієнтської прикладної програми з дочірнім віддаленим модулем даних сервера прикладних програм. Властивість *property ParentConnection: TDispatchConnection;* повинна містити посилання на компонент з'єднання з головним віддаленим модулем даних сервера прикладних програм. Дочірній віддалений модуль даних визначається властивістю *property ChildName: string;,* яке повинне містити його ім'я. Якщо інтерфейс головного віддаленого модуля даних налаштований правильно, то в списку вибору властивості в Інспекторі об'єктів з'являються імена всіх дочірніх віддалених модулів даних. Інтерфейс *IAppServer* дочірнього віддаленого модуля даних повертає властивість

property AppServer: Variant; або метод *function GetServer: lAppServer; override;*

Методи-обробників компонента TSharedConnection успадковані від класу предка TCustomConnection (див. табл. 10.4).

10.4.10. Компонент TConnectionBroker

Компонент *TConnectionBroker* забезпечує централізоване керування з'єднанням клієнтських наборів даних із сервером прикладних програм. Для цього властивість *connectionBroker* клієнтських наборів даних повинне посилатися на екземпляр компонента *TConnectionBroker*. Тоді для зміни з'єднання (наприклад, при переході із транспорту HTTP на сокети TCP/IP) немає необхідності змінювати значення властивості *RemoteServer* всіх компонентів *TClientDataSet*, а досить змінити властивість *property Connection*: *TCustomRemoteServer*; компонента *TConnectionBroker.* Доступ до інтерфейсу *IAppServer* забезпечує властивість *property AppServer: Variant;* або метод *function GetServer: IAppServer; override;* .

Методи-обробники компонента TConnectionBroker повністю відповідають табл. 10.4.

Резюме

Багатоланкові розподілені додатки забезпечують ефективну взаємодію великої кількості віддалених «тонких» клієнтів із сервером БД за допомогою ПЗ проміжного шару. Найпоширенішою моделлю є триланкова модель, де ПЗ проміжного шару складається тільки із сервера прикладних програм. В середовищі *Delphi/ C++Builder* для створення триланкових розподілених прикладних програм використовуються компоненти *DataSnap* і віддалені модулі даних. Всі ці інструменти реалізовані для різних типів транспортних протоколів. Також у триланкових прикладних програмах застосовуються компоненти-провайдери

TDataSetProvider і компоненти *TClientDataSet,* що інкапсулюють набори даних на клієнтському боці.

Контрольні запитання та завдання

1. На якому системному механізмі грунтується взаємодія розподілених програмних систем?
2. В чому причина появи програмного забезпечення проміжного рівня?
3. Опишіть особливості реалізації програмного забезпечення проміжного рівня з використанням стандарту CORBA.
4. В чому переваги та недоліки рішення, що базується на CORBA?
5. В чому полягають особливості розробки та використання RMI?
6. На чому базується технологія програмного забезпечення проміжного рівня DCOM?
7. Що спільного і чим відрізняється DCOM від технологій CORBA та RMI?
8. Що являють собою Веб-сервіси ?
9. Коротко охарактеризуйте мову XML.
10. Чи передбачає стандарт SQL взаємодію з XML?
11. Охарактеризуйте структурні елементи технології розподіленої обробки, базованої на текстових протоколах.
12. Як організовані SOAP-повідомлення?
13. Яку роль грає мова WSDL для механізмів розподіленої обробки, базованої на текстових протоколах.
14. На чому базується механізм для виявлення Web-сервісів UDDI?
15. Охарактеризуйте переваги та недоліки розподіленої обробки за допомогою Web-сервісів?
16. Як кодуються RPC-виклики засобами SOAP –запиту?
17. Які компоненти з'єднань використовуються в технології DataSnap?
18. Яку функцію виконує компонент-провайдер TDataSetProvider ?
19. Опишіть призначення компонент TlocalConnection, TsharedConnection та TconnectionBroker.

Лабораторний практикум

Розглянуті нами сучасні методи електронної обробки дають можливість реалізувати задачі будь-якої складності при обробленні економічних даних. Але однієї теорії недостатньо. Справжнє розуміння сучасних технологій обробки даних може прийти тільки в результаті тривалого практичного досвіду застосування розглянутих технологій

Лабораторна робота №1

Назва:Технологія розробки локальної бази даних у середовищі Delphi(випадок зв'язаних таблиць) з використанням BDE

Мета роботи: Відпрацювати методику розробки локальної бази даних у середовищі Delphi для зв'язаних таблиць

1. Теоретичні відомості:

Технологія комп'ютерно-орієнтованих баз даних надзвичайно складна і різноманітна, навіть якщо брати до уваги тільки ідеологію розробки БД із застосуванням персональних комп'ютерів. За два десятиліття були розроблені різноманітні підходи і методології програмної реалізації СУБД для різних платформ і операційних систем від найпростіших одно користувальницьких до корпоративних систем, від однозадачних БД до мереж Intranet. Тому на ринку існує гігантський спектр програмних продуктів для реалізації задач розробки БД. Переважна кількість цих програмних продуктів орієнтується на реляційну модель даних, відрізняючись один від одного способом реалізації і розробки: однокористувальницькі(dBASE, Fox Pro, Clarion), офісні(MS ACCESS, Lotus Approach), корпоративні(ORACLE, Informix, MS SQL).

Технологія розробки БД у системі Borland Delphi істотно відрізняється від перерахованих тим, що сама Delphi не є спеціалізованим продуктом, для розробки БД, а інтегрованою системою програмування. Для такої системи розробка БД є однією з можливих застосувань спектра прикладних програм. Одним із переваг такої ідеології є можливість одержання модуля, що виконується, не прив'язаного до яких-небудь спеціальних програмних систем, крім операційної системи. Але, головна перевага полягає у відкритості системи, що дозволяє нарощувати можливості в міру появи нових задач. Для студентів Delphi є чудовим полігоном по оволодінню сучасною технологією розробки баз даних. Розробка локального прикладної програми для БД є базовою задачею, оволодівши які можна як розробляти реальні прикладні програми, так і освоювати більш складні технології. Для реалізації локальної версії БД фірма Borland давно і успішно використовує машину баз даних(Borland Database Engine) – скорочено BDE.

Фізично BDE є набором бібліотек доступу до даних, що реалізовують BDE API - набір функцій для маніпуляції даними, які викликаються із прикладних програм. Ці функції, у свою чергу, можуть звертатися до функцій клієнтського API (у випадку, наприклад, Oracle,

Informix, IB Database) або ODBC API (Access 2000, Microsoft SQL Server 7.0, будь-які ODBC-джерела), а також безпосередньо маніпулювати файлами деяких СУБД (dBase, Paradox).

Для доступу до бази даних за допомогою BDE на комп'ютері, що містить клієнтську програму, мають бути встановлені бібліотеки BDE загального призначення, а також BDE-драйвер для даної СУБД. У випадку серверних СУБД такі драйвери носять назву SQL Links. Ці драйвери містять BDE API - стандартний набір функцій, створених на основі функцій клієнтських API відповідних СУБД. Проте, в сучасних версіях Delphi/C++Builder BDE рекомендована для створення прикладних програмах локальних баз даних. Відтак, на фізичному рівні розроблювану БД підтримують формати dBase та Paradox.

Тому в цій лабораторній роботі вам пропонується попрактикуватися в створенні простої прикладної програми, який працює з базами, що являють собою набір зв’язаних таблиць даних , пройшовши при цьому всі етапи - від створення псевдоніма БД до запуску розробленого прикладної програми. Слід зазначити, що у порівнянні з випадком бази даних з однієї таблиці з’являються дві проблеми:

- Проблема проектування – треба правильно представити зв’язок таблиць як сутностей;
- Проблема реалізації бази, як зв’язаних таблиць в середовищі Borland Delphi.

2. Виконання роботи

Розглянемо приклад:

Потрібно створити прикладну програму, призначену для обліку товарів, що надходять на склад. База даних складається з двох таблиць: «Товар» і «Надходження товарів». Таблиці будемо створювати й зберігати у форматі СУБД Paradox.

2.1 Проектування

Проблема проектування в даному випадку вирішується досить просто. Позаяк один і той же товар може багаторазово фігурувати при надходженні на склад, то відношення між між сутностями **Товар** і **Надходження товару** має характер *-один-до-багатьох*, тобто одному товару в таблиці «Товари» може відповідати більш одного запису в таблиці «Надходження товарів». Робоча структура таблиць може бути наступною:

Таблиця «Товар»

Назва поля	Зміст	Тип	Довжина
Tovar	Назва товару	Рядок	20
Od_Vym	Одиниця виміру	Рядок	10
Cina	Ціна за	Цілочисельн	

	одиницю виміру	ий	

Таблиця «Надходження товару»

Назва поля	Зміст	Тип	Довжина
N_Prih	Номер приходу	Автоінкремент	
Tovar	Назва товару	Рядок	20
Datprih	Дата приходу	Дата	
Kilkist	Кількість	Цілочисельний	

Для таблиці «Товар» первинний ключ буде побудований по полю Tovar, а для таблиці «Надходження товару» - по полю N_Prih. Для реалізації посилальної цілісності в таблиці «Надходження товару» необхідно побудувати зовнішній ключ по полю Tovar. Для сортування записів при їхньому виведенні прикладній програмі необхідно створити індекс по полях Datprih, Tovar.

Ось таким чином і проведено проектування. Далі почнемо реалізацію цього проекту.

2.2 Реалізація проекту

2.2.1 Утворення бази

При роботі з таблицями локальних БД (у число яких входять таблиці СУБД Paradox і dBase*)* сама база даних розміщується в каталозі на диску і зберігається у виді набору файлів. Для збереження однієї таблиці створюється окремий файл. Окремі файли створюються для збереження індексів таблиці і мемо-полів. Для розміщення нашої БД буде потрібно не більш 10 Кбайт дискової пам'яті: пожертвуйте й створіть каталог SPROBA на диску C:\student\UK**.

Звертання до БД з утиліт і програми здійснюється по *псевдоніму* бази даних. Псевдонім повинен бути зареєстрований у файлі конфігурації конкретного комп'ютера за допомогою утиліти BDE Administrator. Привласнимо псевдонім SPROBA створюваій БД. Для цього запустіть утиліту BDE Administrator (простіше всього це зробити за допомогою головного меню Windows: Пуск\ Програми\Delphi\BDE Administrator*)*. Виберіть в головному меню вікна утиліти елемент Object\New. У вікні, що з'явилося, (рис.2.1) залишіть тип створюваної БД без змін (STANDART*)* і натисніть кнопку *ОК*. .

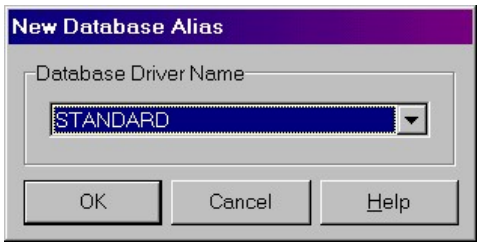


Рис.2.1. вікно вибору типу драйвера бази даних.

У лівому полі вікна адміністратора БД ви побачите рядок з ім'ям STANDART .Змініть це ім'я на SPROBA. У правому полі зазначені параметри БД. Змініть параметр PATH, що вказує маршрут доступу до каталогу, у яких розташовується БД. Можна ввести шлях вручну, але краще скористатися засобами адміністратора, для цього потрібно клацнути по полю PATH і натиснути кнопку, що з'явилася в правому куті поля. Потім варто вибрати каталог **:\Students\UK**** і натиснути кнопку OK(рис.2.2)

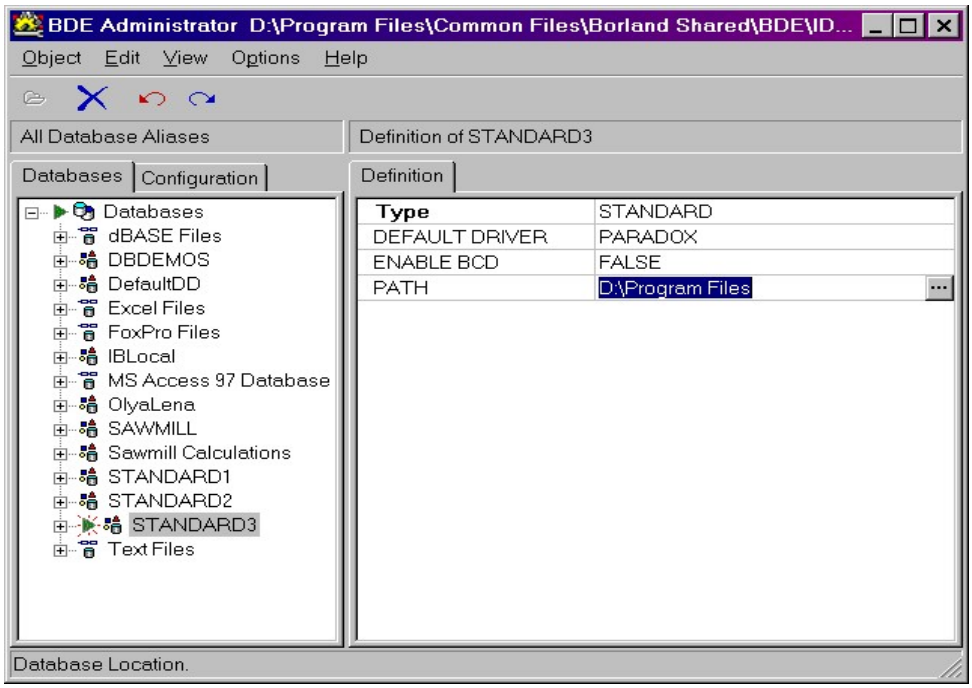


Рис.2.2. Вікно параметрів псевдоніма бази даних.

У додатковому меню вибрати опцію *App1y.* У діалоговому вікні, що з'явилося, (у ньому запитується, чи збираємося ми запам'ятовувати зміни для псевдоніма) натисніть кнопку OK**.** Закрийте вікно утиліти BDE Administrator: створення псевдоніма довершене і до нього можна звертатися з інших утиліт і прикладних програм. Однак каталог, на який посилається псевдонім БД, ще порожній. Необхідно створити в ньому таблиці бази даних.

2.2.2 СТВОРЕННЯ ТАБЛИЦЬ БАЗИ ДАНИХ

2.2.2 .1.Оголошення полів

Для створення таблиць бази даних необхідно запустити утиліту Database Desktop*e* (D&D). Після запуску утиліти установимо псевдонім, що замовчується. Для цього потрібно

вибрати елемент головного меню File\Working Directory і в списку, що випадає, вибрати ім'я псевдоніма SPROBA, після чого натиснути *кнопку ОК.*

Для створення таблиці БД потрібно вибрати елемент головного меню File\New\Table. *У* вікні, що з'явилося, Create Table залишіть без зміни тип створюваної таблиці (Paradox 7) і натисніть *кнопку ОК.* Після цього з'явиться вікно визначення структури таблиці БД(рис.2.3)

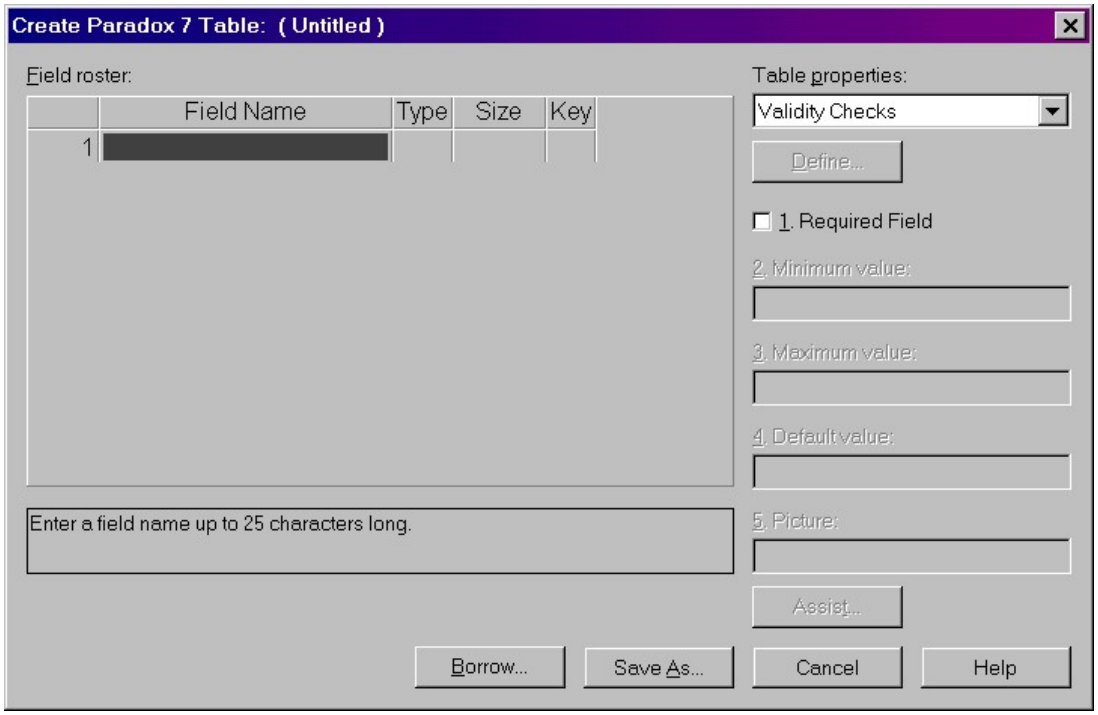


Рис.2.3. Утиліта Database Desktop:вікно визначення структури таблиці БД.

Кожен рядок таблиці відповідає полю. Призначення колонок(

* Fields Name - ім'я поля(
* Type- тип поля;
* Size -розмір поля (для строкових полів , оскільки інші поля мають на увазі розмір, обумовлений типом поля);

Для цього, переходячи від поля до поля, уключіть перемикачі Required Fields. .Інші поля служать для накладення обмежень на значення поля:

- Minimum value- визначає мінімальне значення поля;
- *Maximum value*- визначає максимальне значення поля;
- *Default value*- визначає значення поля за замовчуванням;
- *Picture*- визначає шаблон зображення поля. Відсутність значення в одному з полів означає відсутність обмежень на значення поля.

2.2.2.2.Запам'ятовування таблиці

Щоб запам'ятати збережену таблицю на диску, натисніть кнопку Save As і у вікні, що з'явилося, вкажіть ім'я Tovary. Оскільки ви працюєте в Windows 32, ім'я таблиці, на відміну від імен полів, могло б складатися з українських букв: Товари. При бажанні ви можете саме

так назвати таблицю без найменшого збитку для подальшого виконання нашого приклада. Однак варто пам'ятати про те, що в реальній роботі вам, можливо, буде потрібно перейти до систем клієнт-сервер. Усі промислові сервери не приймають кирилицю в іменах таблиць, тому вам доведеться українські імена заміняти на латинські і вносити додаткові зміни у вже створені вами програми. .

У момент збереження в каталозі .:\ SPROBA буде створений табличний файл TOVARY.DB. і файли індексів із відповідними розширеннями.

2.2.2 .3.Визначення індексів.

Визначіть структуру таблиці «Надходження товарів»(рис.2.6)

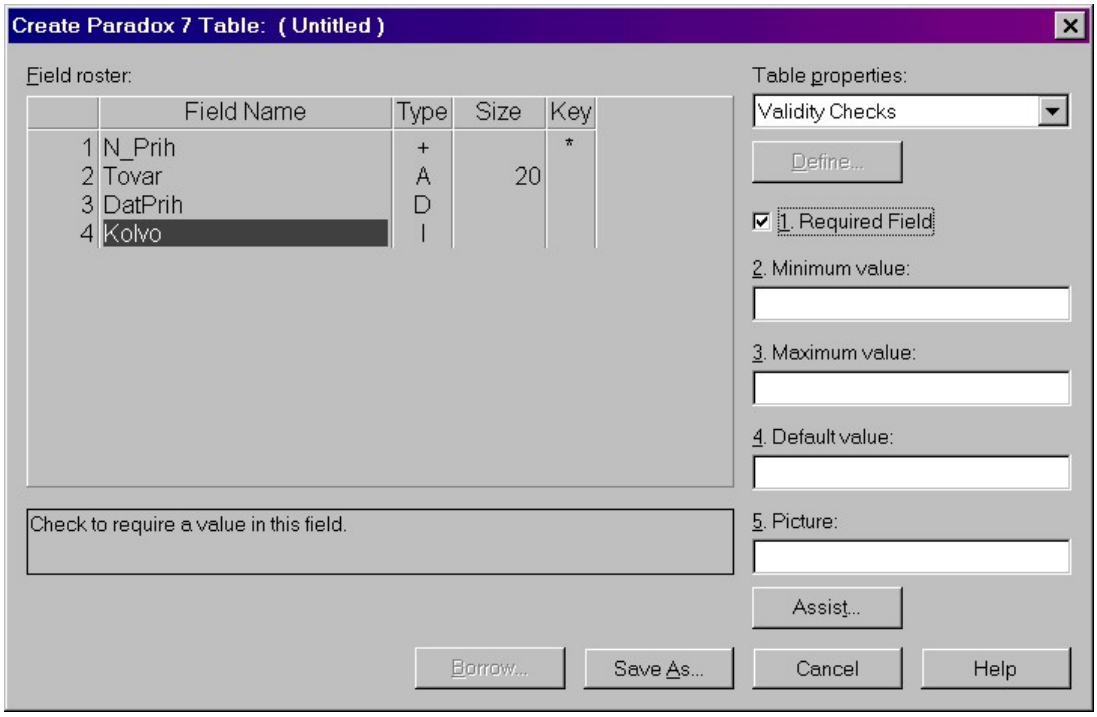


Рис.2.6. Структура таблиці Nadhod.

поле автоінкрементне, заповнення його значенням проводиться автоматично при запам'ятовуванні нового запису. Створимо індекс по полях **DatPrih, Tovar**. Для цього в комбінованому списку **Table** Propertice (у правому верхньому куті вікна) виберіть елемент Secondory Indexes. Щоб визначити новий індекс, натисніть кнопку Define. У діалоговому вікні, що з'явилося, у поле Fields міститься список полів обраної нами таблиці. Поле Index Fields призначене для збереження полів, що входять у створюваний індекс. Щоб скопіювати конкретне поле зі списку Fields у список Index Fields, потрібно натиснути кнопку з зображенням правої стрілки. Послідовність додавання полів у список важлива, вона визначає порядок чергування полів у списку. Після того, як ви помістили потрібні поля в список Index Fields(рис.2.7), натисніть кнопку ОК.

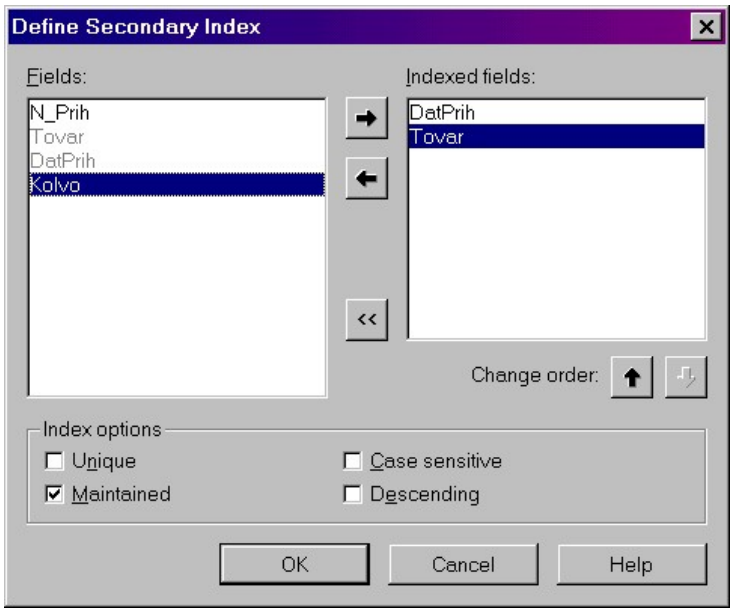


Рис.2.7. Визначення полів , що входять до складу індексу.

У діалоговому вікні, що з'явилося, запитується ім'я індексу. Введіть ім'я Sort_DatPrih_Tovar і натисніть ОК.

Не рекомендується складати назву індексу тільки з імен полів, оскільки такий спосіб іменування індексів використовується автоматично при створенні посилальної цілісності між таблицями (див. наступний розділ). Як видно на рис.2.8, після додавання нового індексу його ім'я з'явилося в списку індексів.

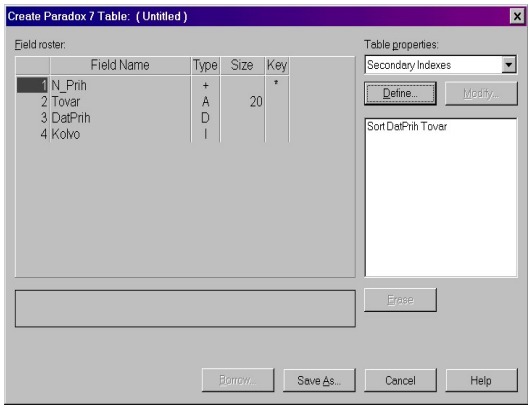


Рис.2.8. Список індексів, визначених для таблиці Nadhod.

Згодом, клацнувши по імені індексу, ми можемо його видалити (кнопка *Erase* чи змінити (кнопка *Modify)).*

Збережіть створену таблицю під ім'ям *Nadhod*.

2.2.2 .4. Визначення посилальної цілісності між таблицями

Таблиці *Tovary* і *Nadhod* знаходяться у відношенні *один-до-багатьох,* тобто з одним записом у таблиці Tovary може бути зв'язано кілька записів по приходу того ж товару в таблиці Nadhod. У якості поля зв'язку виступає поле Tovar, що є присутнім в обох таблицях.

Визначимо посилальну цілісність між даними таблицями. Посилальна цілісність у *Paradox* визначає, по-перше, зв'язок між таблицями, а по-друге, вид каскадних впливі.

Відкрийте таблицю *Nadhod* (елемент меню *File \ Table \ Open)* і потім виберіть режим зміни структури таблиці *(Table | Restructure).* У списку, що *випадає, Table* Properties виберіть *елемент Refrential* Integrity і натисніть *кнопку Define.* У діалоговому вікні, що з'явилося, (рис.2.9) у списку *Fields* показані поля таблиці *Nadhod*, і списку *Tables* - таблиці бази даних *SPROBA.*

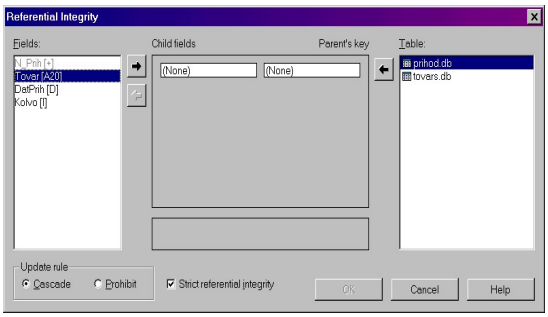


Рис.2.9.Вікно визначення посилальної цілісності.

Назву *Tovar* буде записано в поле *Child Fields* (поле зовнішнього ключа дочірньої таблиці).

Виберіть у списку *Tables* таблицю *Tovary* і натисніть кнопку з зображенням стрілки вліво. У поле *Parents Key* (ключ батьківської таблиці) будуть показані поля з первинного ключа таблиці *Tovary.* У даному випадку це поле *Tovar*.

Перемикачі *Update rules* визначають вид каскадних впливів на таблицю *Nadhod* при зміні значення поля зв'язку в таблиці *Tovary* чи при видаленні в ній запису:

• *Cascade* - дозволені каскадні зміни і видалення підлеглих записів у дочірній таблиці;

• *Prohibit* - заборонені зміни полів чи зв'язки видалення запису в батьківській таблиці, якщо для даного запису є зв'язані записи в дочірній таблиці.

Залишіть вибір за *умовчанням,* Cascade і натисніть *кнопку OK* - DBD запросить ім'я посилальної цілісності (*у* Paradox посилальні цілісності іменуються). Введіть ім'я, наприклад *Tovary_Nadhod_Jntegrity,* і натисніть кнопку *OK.* Тепер ім'я створеної засланої цілісності буде поміщено в список.

Збережіть зміни в таблиці *Nadhod* (кнопка *Save)* і заново ввійдіть у режим реструктуризації *(Table | Restructure).* У списку, що *випадає, Table* properties виберіть *елемент Secondary* Indexes (індекси таблиці, крім індексу, побудованого по визначенню первинного ключа). У списку індексів ви побачите новий індекс з ім'ям *Tovar***.** Цей індекс

побудований автоматично по неявному визначенню зовнішнього ключа при створенні посилальної цілісності. Вийдіть з режиму реструктуризації (кнопка *Cancel)* і закрийте вікно *DBD.*

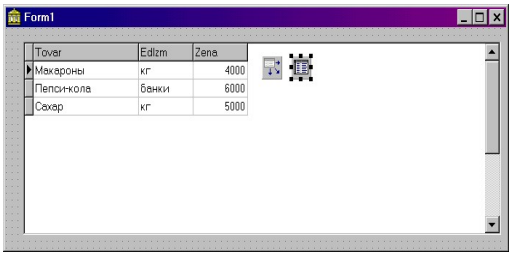
2.4. Створення нйпростішої прикладної програми

Створіть у каталозі Students\Uk** підкаталог *APP.* У ньому будуть зберігатися розроблені вами програми.

Запустіть *Delphi.* За замовчуванням *Delphi* при своєму запуску створює, порожню форму для нового прикладної програми. Скористаємося нею. У палітрі компонентів на сторінці ***BDE*** виберіть мишею компонентів *(Table),* клацніть на ньому мишею і потім клацніть мишею на формі. Після цього зображення компонента залишиться у формі.

У вікні Інспектора Об'єктів Установіть у властивість *DatabaseName* (псевдонім БД) значення SPROBA за допомогою списку, що випадає, чи вручну. У такий же спосіб виберіть значення TOVARY.DB у властивість *TableName* (ім'я таблиці БД). Для властивості *Active* Установіть значення *True.* У цей момент відбудеться реальне зв'язування компонента *Table* (він за замовчуванням має ім'я *Table1)* з таблицею TOVARY.DB.

Розташуйте на формі компонентів *(DataSource).* Він служить як сполучна ланку між невізуальними компонентами (у даному випадку *Table1)* і візуальними компонентами, що ми додамо у форму пізніше. Тому компоненти *DataSource* часто називають *джерелами даних.* Установіть у властивість *DataSet* (ім'я вибору даних) компонента значення *Table1,* вибравши його зі списку, що випадає. Розташуємо у формі компонентів *DBGrid,* узявши його з палітри компонентів на сторінці *Data Controls.* Установіть у властивість *DataSource* компонента значення *Data-Source* (це ім'я, привласнена за замовчуванням створеному нами перед цим компонентом).



Вид розроблювальної форми представлений на рис.2.10.

Рис.2.10. Вид форми на етапі розробки.

Збережіть форму і проект на диску і запустіть програму. Працююча програма відкриває вам безпосередній доступ до даних у таблиці *Tovary.*

Для додавання запису потрібно натиснути на клавіатурі клавішу *Insert* чи, знаходячись в останньому записі набору даних, клавішу зсуву курсору вниз. Таблиця

автоматично перейде в режим додавання нового запису. Після введення значень у поля записів запам'ятати запис у наборі даних можна, перейшовши на інший запис за допомогою клавіш керування курсором. Відмовитися від запам'ятовування запису можна, натиснувши кнопку *Esc.*

Для зміни запису варто перемістити покажчик поточної запису в потрібне місце і змінити значення там, де це необхідно. Набір даних автоматично перейде в режим редагування.

Для видалення запису варто установити на неї покажчик поточної запису і натиснути *Ctrl+Del.*На рис.2.11 показаний вид вікна програми в момент додавання в таблицю *Tovary* нового запису.

2.5. Створення прикладної програми для роботи з двома таблицями

Покажемо, як в одній формі можна зв'язати два набори даних (головний і підлеглий) так, щоб у підлеглому наборі завжди показувалися записи, що відповідають поточному запису в головному наборі.

DataSource (ім'я *DataSource2)* для роботи з таблицею *Nadhod.* За допомогою властивостей *DatabaseName* і *TableName* налаштуйте компонент *Table2* на роботу з таблицею NADHOD.DB і Установіть в його властивість *Active* значення *True,* а за допомогою

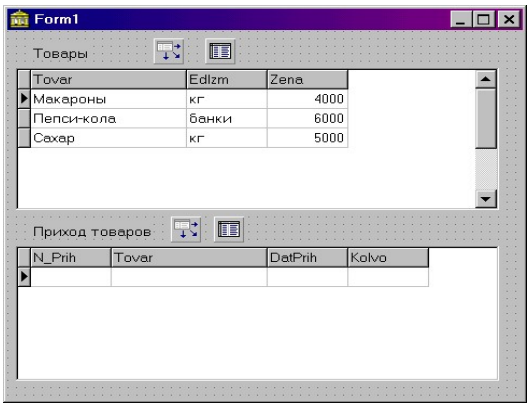


Рис.2.11. Додавання нового запису в таблицю Tovary.

властивості *DataSel* компонента *DataSource2* зв'яжіть його з компонентом *Table2.* Розмістіть на формі

Рис.2.12. Форма з головним і підлеглим наборами даних під час розробки.

компонентів *TDBGrid* (ім'я *DBGrid2)* і Установіть в його властивість *DataSource* значення *DataSource2.* Щоб пояснити, уміст яких таблиць відображають компоненти *DBGrid,* помістіть над кожним з них по компоненті *TLabel* і Установіть в їхніх властивостях *Caption* значення **Товари і Надходження товарів** (рис.2.12).

Якщо зараз запустити програму на виконання, обидві таблиці будуть незалежні одна від одної. Щоб зміст підлеглого набору відповідав вибору запису в головному наборі, підлеглу таблицю потрібно зв'язати з головною. Для цього розкрийте список вибору у властивості *MasterSource* таблиці *Table2* і виберіть єдине наявне в ньому значення *DataSource1.* Потім клацніть по правій частині рядка *Master-Fields* у вікні Інспектора Об'єктів і по кнопці, що з'явився в ній, із трьома крапками, щоб розкрити вікно редактора зв'язків. Розкрийте список *AvailableIndexes* у верхній частині цього вікна і виберіть індекс *Tovar* - у віконці *Detail Fields* з'явиться ім'я поля зв'язку *Tovar.* Виберіть це ж поле у віконці *Master Fields* і натисніть кнопку *Add* -зв'язок установлений (див. рис.2.13).

На інструментальній панелі вікна редактора (рис.2.16,6). Щоб змінити заголовок колонки, потрібно клацнути по ній в списку полів редактора колонок і в *сектор Об'єктів* розкрити список властивості *Title* (для цього потрібно або двічі клацнути по імені властивості мишею, або клацнути по ньому правою кнопкою і вибрати *Expand).* В елементі *Caption* цього списку міститься заголовок колонки. Змініть відповідним чином заголовки і потім закрийте редактор колонок. Таким же чином змініть заголовки колонок компонента *DBGrid1.*

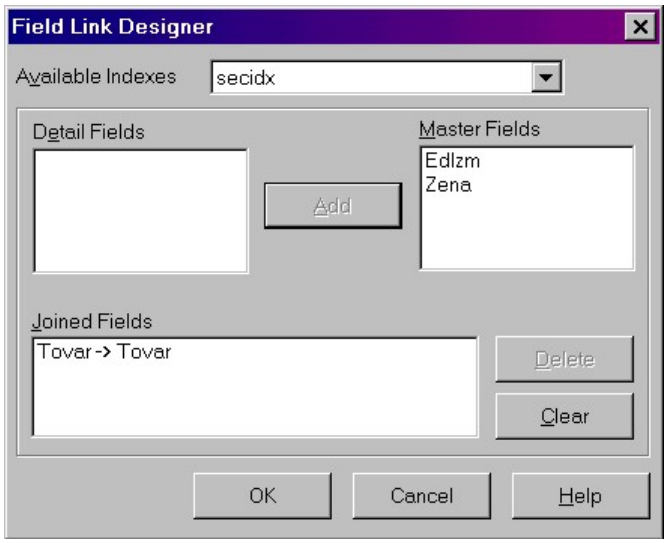


Рис.2.13 Вікно редактора зв'язків.

Закрийте вікно редактора зв'язків і запустіть прикладна програма. Введення і зміна даних у таблиці Nadhod можна робити за допомогою компонента DBGrid2 (мал.2.14). При цьому переміщення покажчика в головній таблиці приводить до автоматичної зміни інформації у відображуваних даних підлеглої таблиці.

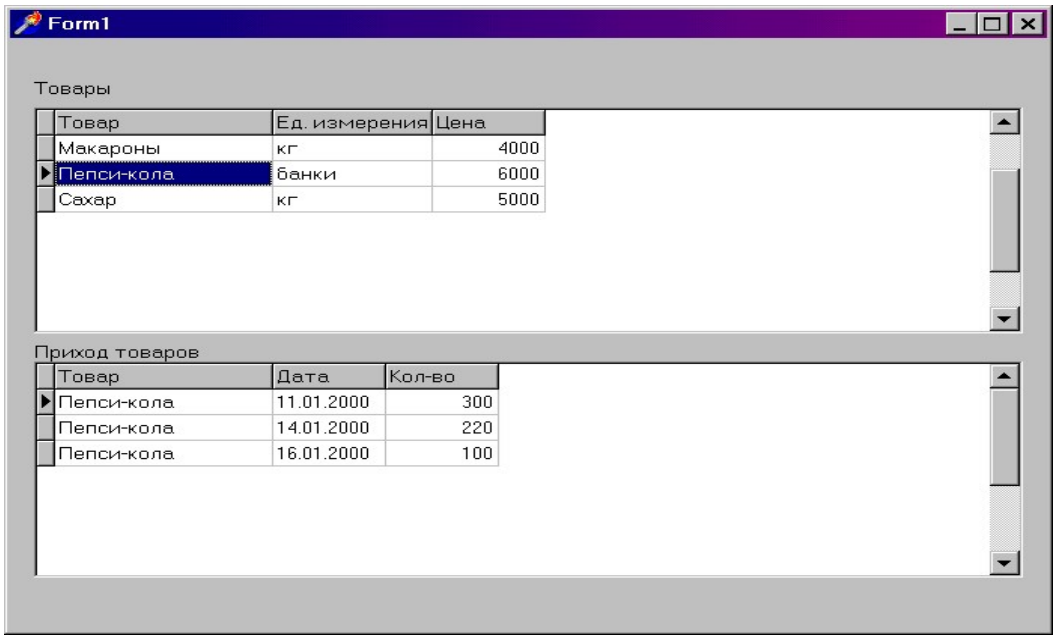
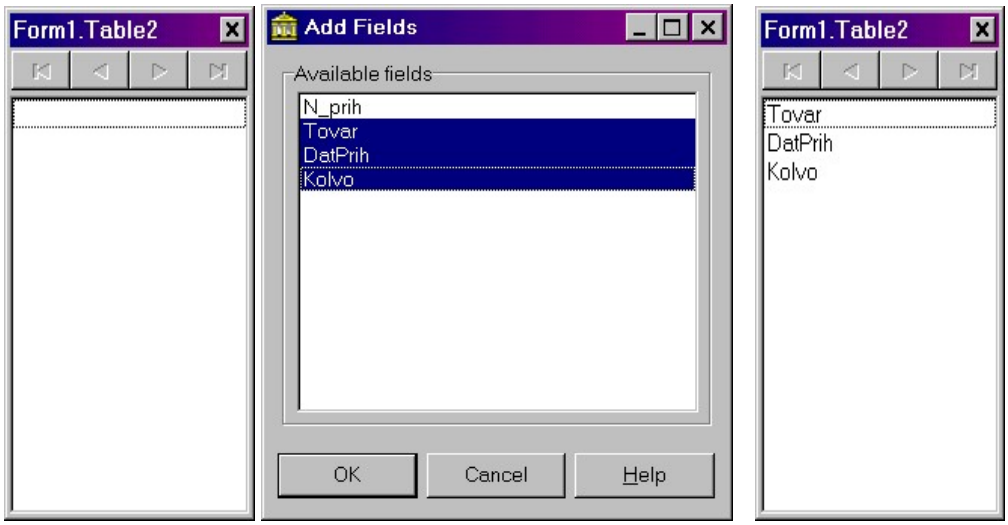


Рис.2.14.Заповнення таблиці Nadhod.

Як можна помітити, значення поля N_Prih необхідно для забезпечення унікальності записів у таблиці Nadhod і не несе ніякого іншого навантаження. Тому дане поле краще не показувати в складі колонок DBGrid2 . Для цієї мети сформуємо список полів таблиці Nadhod .

Двічі клацніть по компоненті *Table2* у вікні чи форми клацніть по ньому правою кнопкою миші й у додатковому меню виберіть елемент Fields Editor. На екрані з'явиться порожнє вікно редактора полів (мал.2.15,а). Клацніть по ньому правою кнопкою миші і виберіть елемент меню Add Fields. Буде показаний список усіх полів Nadhod.DB. Виділіть (за допомогою миші і клавіші SHIFT) усі поля, крім N_Prih(мал.2.15,6) і натисніть кнопку *OK.* Тепер список редактора полів буде включати усі відзначені поля (мал.2.15,в).



а) б) в)

Рис.2.15. Робота з редактором полів:

а) порожній список, б) додавання полів, в) заповнений список.

Для кожного доданого поля Delphi автоматично створює об'єкт класу TFileld(поле набору даних) і призначає йому унікальне ім'я, що виходить зчепленням імені набору даних *(Table2)* і імені поля. Так, компонент TFileld, що відповідає полю Tovar , одержить ім'я *Table 2*Tovar. Імена автоматично створених об'єктів полів ви знайдете в оголошенні класу *TForm1* у вікні коду. Якщо якесь поле **(**у нашому випадку N_Prih *)* не включено в список полів, воно стає невидимим для програми і не з'являється у відповідному наборі даних. Змінимо параметри компонента DBGrid2 так, щоб назви його колонок містили українські найменування. Для цього клацніть по компоненті правою кнопкою мили і виберіть у меню елемент Columns Editor. На екрані з'явиться вікно редактора.

а) б)

Рис.2.16. Вікно редактора колонок: а) порожній список колонок; б) заповнений список.

Відсортуємо запису в наборі даних *Table2* по даті приходу товару. Для цього в Інспекторі Об'єктів розкрийте список у властивості *Table2.IndexFieldNames* і виберіть значення DatPrih;Tovar. Після цього ще раз викличте редактор колонки *DBGrid2* і за допомогою миші «перетягнете» стовпець *DatPrih* так, щоб він передував стовпцю *Tovar.*

2.7. ВИЗНАЧЕННЯ ВІЗУАЛЬНИХ КОМПОНЕНТІВ ДЛЯ РОБОТИ З ПОЛЯМИ

Поліпшимо інтерфейс нашої програми так, щоб до полів запису в наборі даних *Table2* можна було звертатися не тільки із сітки компонента *DBGrid2*, але і за допомогою додаткових візуальних компонентів.

Помістіть на форму два компоненти *TDBEdit* (сторінка *Data Controls* палітри компонентів): компонент *DBEdit* під колонкою «Дата приходу», а компонент *DBEdit2* під колонкою «Кількість» (рис.2.17). Щоб зв'язати нові компоненти з полями, помістіть в їхні властивості *DataSource* значення *DataSource2* і у властивості *DataField* відповідно *DatPrih* і *Kolvo.*

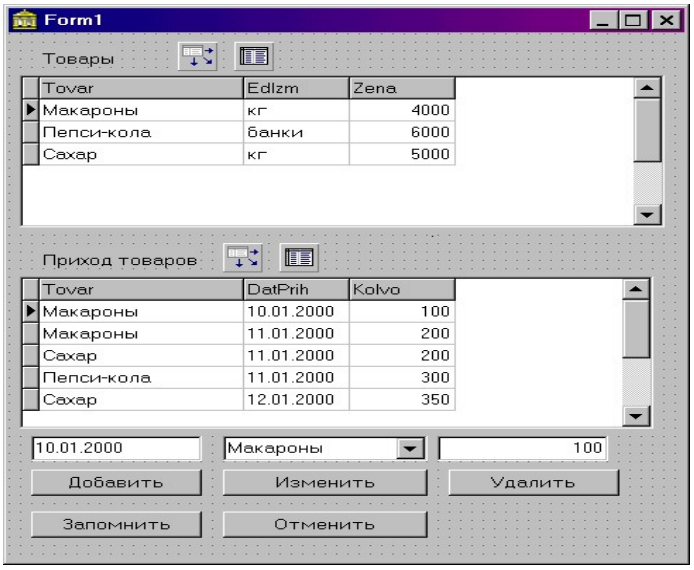


Рис.2.17. Форма з візуальними компонентами для роботи з полями поточного запису.

Для доступу до поля *Tovar* нам потрібний більш складний компонент, що дозволяв би вводити в це поле тільки ті значення, що уже введені в поле *Tovar* таблиці TOVARY.DB і ніякі інші. Для цієї мети під колонкою «Товар» компонента *DBGrid2* розмістіть компонент *TDBLookupComboBox.* Змініть значення наступних властивостей за умовчанням, цього компонента так:

Властивість	Значення	Властивість	Значення
DataSource	DataSource2	ListField	Tovar
DataFiled	Tovar	KeyField	Tovar
ListSource	DataSource1		

Тепер на формі є спеціальні компоненти для доступу до полів запису. Щоб звертання до полів йшло тільки за допомогою цих компонентів, заборонимо доступ до полів з компонента *DBGrid2:* встановіть в його властивості *ReadOnly* значення *True.* Додайте до форми 5 компонентів *TButton* (сторінка *Standard* палітри компонента). Змініть імена цих компонентів (властивість *Name)* відповідно на *InsertButton, EditButton, DeleteButton, PostButton, EditButton.* Змініть напису на кнопках (властивість *Caption)* відповідно на «Додати», «Змінити», «Видалити», «Запам'ятати», «Скасувати».

Напишіть такий обробник події *OnClick* для кнопки *InsertButton:*

```
begin
if Table2.State = dsBrowse then
Table2.Insert;
end;
```

Якщо набір даних *Table2* знаходиться в режимі перегляду *dsBrowse,* метод *Insert* переводить його в стан додавання запису *dslnsert.* Уведення значень полів здійснюється в компонентах *DBEdit, DBLookupComboBox1, DBEdit2.*

Для кнопки *EditButton:*

```
procedure Tform1.EditButtonClick(Sender: TObject);
begin
if Table2.State = dsBrowse then
Table2.Edit;
end;
```

Метод *Edit* переводить набір даних *Table2* у стан додавання запису *dsEdit.* Редагування значень полів здійснюється в компонентах *DBEdit1, DBLookup, ComboBox1, DBEdit2.*

Для кнопки *DeleteButton:*

```
procedure Tform1.DeleteButtonClick(Sender: TObject);
begin
if Table2.State = dsBrowse then
if MessageDlg('Підтвердіть видалення запису',
mtConfirmation, [mbYes, mbNo],0) = mrYes then
Table2.Delete;
end;
```

Якщо набір даних *Table2* знаходиться в режимі перегляду записів *dsBrowse,* викликається діалогове вікно *MessageDIg* із пропозицією підтвердити видалення запису. Якщо користувач натискає кнопку *Yes, що* текст запис у наборі даних *Table2* вилучається методом *Delete.*

Для кнопки *PostButton:*

```
procedure Tform1.PostButtonClick(Sender: TObject);
begin
 if Table2.State in [dslnsert, dsEdit] then
Table2.Cancel;
end;
```

Якщо набір даних знаходиться в режимі додавання нового запису чи редагування, викликається метод *Post* набору даних *Table2,* що запам'ятовує поточний стан запису в таблиці БД. Після запам'ятовування набір даних автоматично переводиться в режим перегляду *dsBrowse.*

Для кнопки *CancelButton:*

```
procedure Tform1.CancelButtonClick(Sender: TObject);
begin
if Table2.State in [dslnsert,dsEdit] then
Table2.Cancel;
end;
```

Запустіть програму. При додаванні нового запису чи при коректуванні існуючої в поля можна заносити значення, використовуючи компоненти *DBEdit, DBEdit2* і шляхом вибору зі списку значень у компоненті *DBLookupComboBox1.* Те ж відбувається при зміні запису. При видаленні запису з'являється діалогове вікно, показане на рис.2.18.

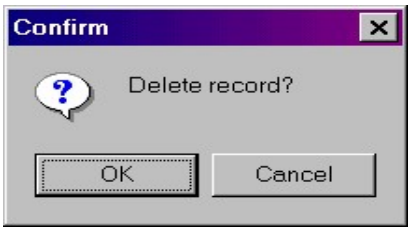


Рис. 2.18. Вікно підтвердження видалення запису.

2.8. Технологія маніпулювання даними бази зв’язаних таблиць засобами DatabaseDesktop.

Розробка баз передбачає її подальше використання для зручного маніпулювання цими даними в подальшому. Для цього Delphi надає надзвичайно потужний набір засобів. Для випадку однієї таблиці це було розглянуто в попередній роботі. Що стосується зв’язаних таблиць , то тут ми розглянемо найпростішу класичну технологію з використанням утіліти **DatabaseDesktop.** Навіть у цьому дійсно найпростішому вигляді можливі два варіанти: варіант **QBE** та варіант ***SQL.***

2.8.1 Використання мови QBE

Запустіть утіліту DatabaseDesktop. Щоб створити новий запит виберіть команду меню File/New/QBE. Це викличе вікно вибору імені файлів, Щоб обрати послідовність файлів

треба тримати натиснутою клавішу Ctrl, а мишою відібрати файли. Для додавання таблиці в запит треба вибрати Edit/Add Table.

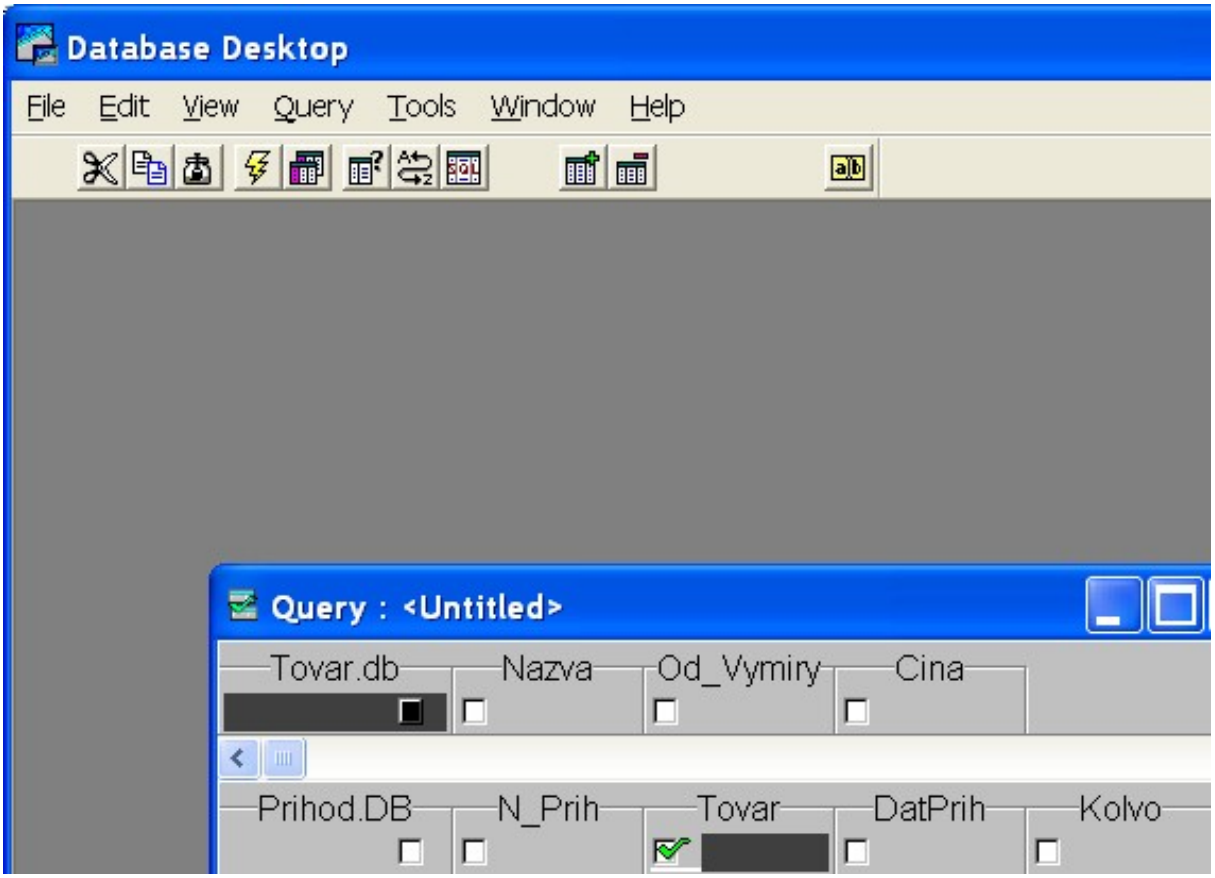


Рис 2.19 Формування QBE-запиту

Типи запитів

Для формування типу запиту треба стати на крайнє ліве поле і правою клавішею миші клацнути по ньому, що викличе список варінтів вибору: запит на вибірку(пустий рядок), запит на додавання(Insert), запит на видалення(Delete). Встановлення змісту запиту формується в кожному полі таблиці запиту справа від виключателя.

2.8.2 Використання мови *SQL*

Ця технологія відома як локальне використання *SQL*. У цьому розділі ілюструється створення набору даних з двох таблиць за допомогою утіліти DatabaseDesktop. Щоб створити новий запит виберіть команду меню File/New/*SQL* після чого з’явиться редактор коду *SQL*

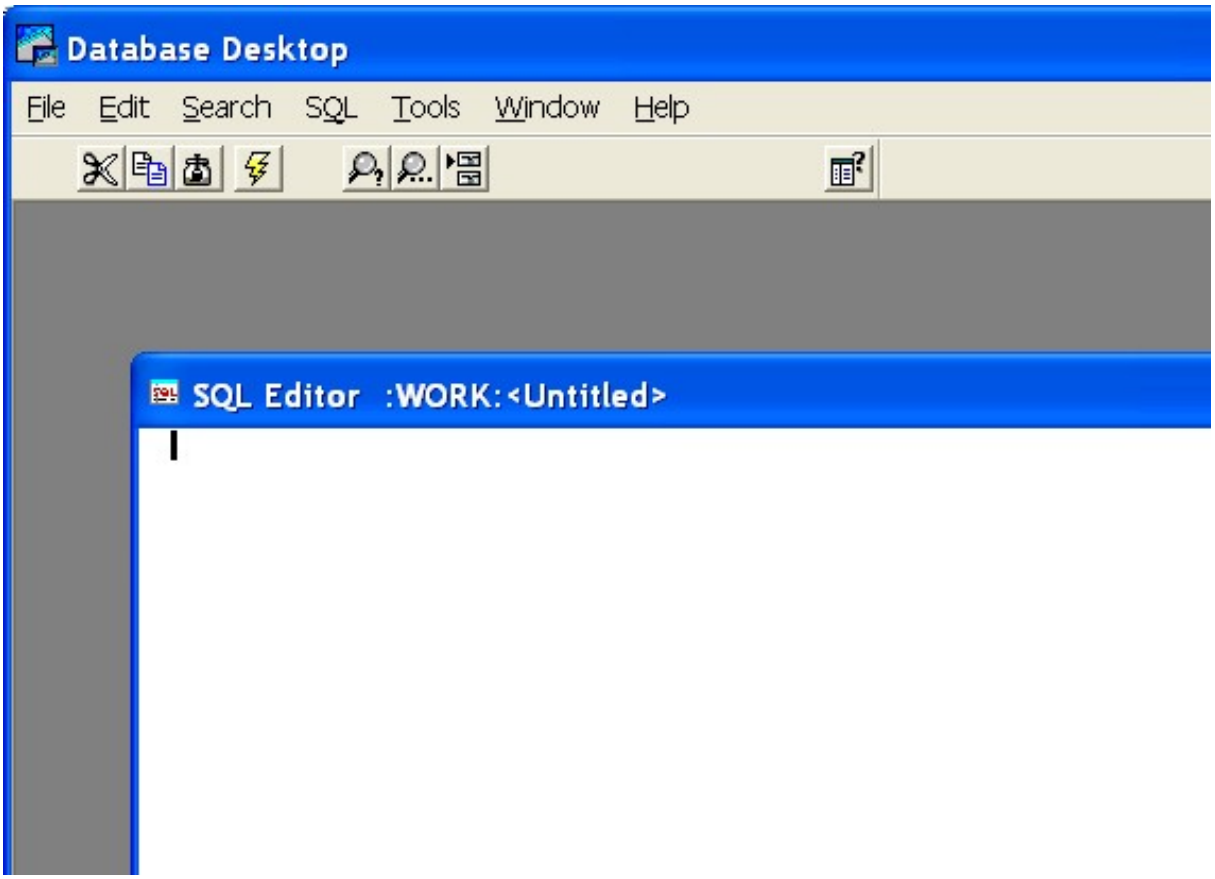


Рис 2.20 Формування *SQL*-запиту

Розкрийте редактор введіть такий текст *SQL-запиту*:

```
SELECT P.DatPrih, P.Tovar, P.Kolvo, T.Cina,
        (P.Kolvo * T.Cina) As Vartist
FROM Tovary T, Nadhod P
WHERE T.Tovar = P.Tovar
```

ORDER BY P.DatPrih, P.Tovar

Закрийте редактор кнопкою *OK*.

Як видно з тексту запиту, набір даних збирається з двох таблиць: TOVARY.DB І NADHOD.DB (FROM Tovary T, Nadhod P). При цьому з'єднуються записи, що мають однакове значення поля TOVAR (*WHERE T.Tovar = P.Tovar*). Частина *SELECT* P.DatPrih, P.Tovar, P.Kolvo, *T.Cina, (P.Kolvo * T.Cina) As Vartist* тексту запиту визначає обрані поля, причому створюється поле, що обчислюється, Vartist зі значенням *P.Kolvo * T.Cina*.

Щоб виконати запит, встановіть у властивості *Query1.Active* значення *True*.

Якщо ви запустите програму, то знайдете, що змінити яким-небудь чином відображувані в *DBGrid1* дані не можна, тому що вони отримані в результаті виконання оператора *SELECT мови SQL,* що об'єднав дані з двох таблиць.

3. Завдання на виконання

Створити локальні бази даних в середовищі Delphi(спроектувати схеми відношення таблиць, інтерфейс для вводу даних, запити до БД, звіти) для згідно варіанту . Узгодьте з викладачем зміст вхідних таблиць(не менше двох), що необхідні для формування набору даних , перелік запитів(QBE,*SQL)* і звітів.

Лабораторна робота №2

1. Назва: Інтерфейс роботи з БД dbExpress та робота з ним

2. Мета: Вивчити механізм дії технології dbExpress та оволодіти навичками розробки прикладних програм для роботи с БД з використанням dbExpress

3.Теоретичні положення

Borland dbExpress

Технологія Borland dbExpress – це варіант полегшеного використання спільного API для різних баз даних, відома як provider/resolver. У даній лабораторній роботі розглядається архітектура dbExpress і механізм provider/resolver, демонструється приклад створення прикладних програм на компонентах dbExpress.

Архітектура dbExpress

dbExpress був розроблений для вирішення наступних 6 завдань:

- мінімізувати обсяг і кількість використовуваних ресурсів
- одержати максимальну швидкість роботи
- забезпечити кросс-платформеність
- забезпечити легкість поширення
- забезпечити легкість розробки драйверів
- дати розробнику більше можливостей керування пам'яттю й мережним трафіком

Драйвери dbExpress невеликі за обсягом й швидкі, тому що вони забезпечують досить невелику функціональність. Кожен драйвер виконаний у вигляді dll (на платформі Windows) або як so (shared library на Linux). Драйвер dbExpress надає п'ять інтерфейсів для вибірки метаданих, виконання операторів SQL і збережених процедур, і можливість читання записів з вибірки в одному напрямку (unidirectional cursor). У цей же час, при використанні з DataSetProvider й ClientDataSet, dbExpress надає повнофункціональну, високопродуктивну, багатокористовачеву систему для роботи з SQL-серверами баз даних.

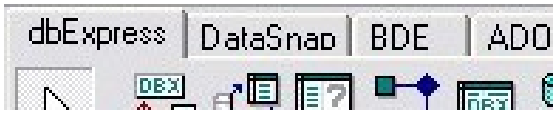
Створення прикладних програм на dbExpress

Перед тим як переносити існуючі BDE-додатки на dbExpress, Ви повинні вміти працювати з компонентами dbExpress. У цьому розділі ми створимо прикладну програму dbExpress, крок

за кроком, з описом кожного використовуваного компонента. Цей приклад створений в Delphi на Windows (або C++Builder), але кроки ідентичні при роботі з Kylix на Linux.

Прикладна програма буде використовувати базу даних прикладів EMPLOYEE.GDB і відображувати відношення один-до-багатьох між таблицями Employee й Salary History. Прикладна програма демонструє наступні можливості dbExpress:

- вкладеність таблиці деталей у поле головної таблиці
- редагування даних через SQLQuery, DataSetProvider і ClientDataSet
- використання SQLQuery як джерела даних тільки для читання без DataSetProvider й ClientDataSet
- застосування змін, накопичених ClientDataSet, до бази даних
- обробка помилок при застосуванні змін до БД



Компонент SQLConnection

Для створення простої прикладної програми dbExpress, почнемо з наступних кроків

1. Створіть новий прикладну програму і додайте data module. Назвемо модуль даних MainDm
2. Використайте вікно Project Options, для того щоб переконатися, що модуль створюється автоматично перед створенням головної форми
3. Помістіть компонент SQLConnection із закладки dbExpress на модуль даних
4. Назвіть компонент SQLConnection як EmployeeConnection й Встановіть його властивість DriverName в Interbase

Відкрийте редактор властивостей Params й установимо параметр Database у шлях до EMPLOYEE.GDB

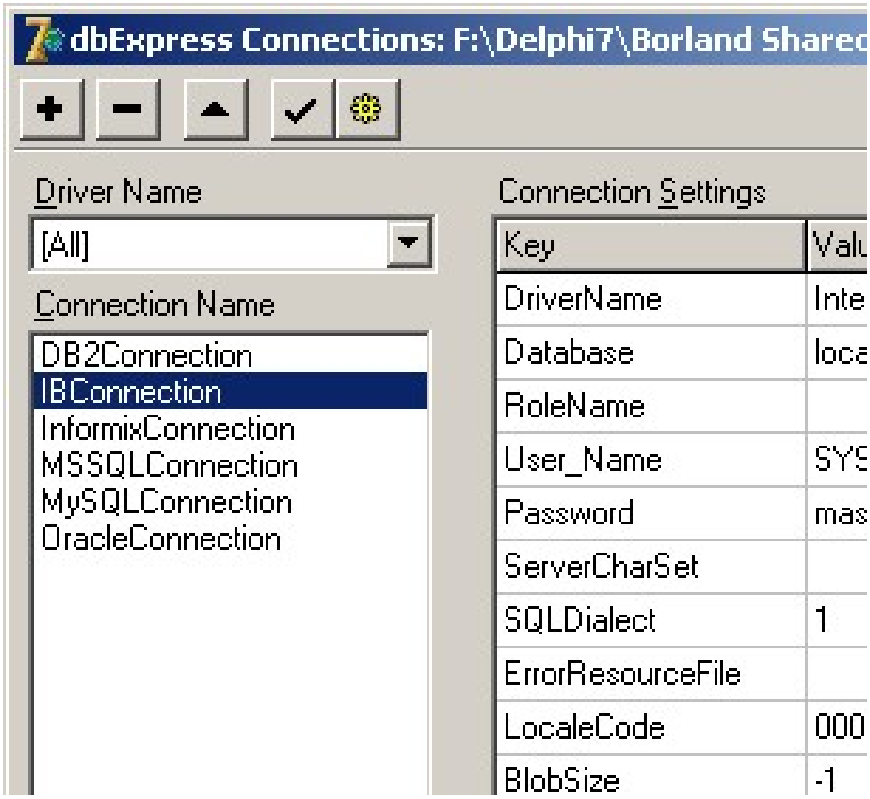
5. Змініте значення властивостей UserName й Password на потрібні
6. Встановіть властивість LoginPrompt в False, щоб не з'являлося вікно запиту імені користувача й пароля при старті програми
7. Встановіть властивість Connected в True для перевірки з'єднання, і назад в False

Компонент SQLConnection забезпечує з'єднання з базою даних для будь-якої кількості компонентів. Ви можете використати трохи SQLConnection для приєднання до декількох баз даних одночасно.

Існує три способи здійснити з'єднання з базою даних: використати існуючий аліас, створити новий "аліас" або записати параметри з'єднання прямо в SQLConnection.Properties.

Для створення нового з'єднання виконайте подвійне клацання мишею на компоненті SQLConnection, при цьому відкриється вікно Connection Editor. З лівого боку покажуться створені раніше аліаси. Список вибору Driver Name дозволяє відфільтровувати потрібні

аліаси по типах серверів. Сітка Connection Settings показує список параметрів для поточного з'єднання (вони можуть відрізнятися залежно від типу драйвера). Всі аліаси, які ви створюєте, зберігаються у файлі **dbxconnections.ini**.



якщо база даних на сервері має кодування WIN1251, то його потрібно вказати у властивості ServerCharSet. Якщо на сервері в базі даних установлена підтримка кодування KOI8R, то для Kylix більше природним буде вказати ServerCharSet=KOI8R (навіть якщо база даних має кодування WIN1251).

Файл аліасів

Якщо ви використаєте іменовані з'єднання (аліаси), то файл dbconnections.ini можна розповсюджувати зі своїм додатком.

Властивість Params компонента SQLConnection

При створенні нового аліаса потрібно вибрати тип сервера в DriverName. Список серверів зберігається у файлі dbxdrivers.ini. Вибір потрібного сервера в DriverName установить властивості LibraryName й VendorLib у значення, що відповідають даному серверу. LibraryName містить ім'я драйвера dbExpress, а VendorLib - ім'я клієнтської бібліотеки сервера.

Після вводу параметрів інформація про з'єднання буде зберігається усередині прикладної програми. Якщо ви хочете дати можливість користувачам прикладних програм змінювати цю інформацію, то параметри можна зберігати у своєму файлі конфігурації, і редагувати його в додатку або окремій утиліті конфігурації. Це зробить прикладну програму самодостатньою.

В SQLConnection є методи StartTransaction, Commit й Rollback для явного керування транзакціями. Якщо потрібно викликати оператори SQL, які не повертають набір даних (наприклад, оператори DDL), то можна скористатися методами Execute або ExecuteDirect. При цьому компоненти для роботи з SQL не потрібні. Якщо потрібний доступ до метаданим, то визвіть методи GetTableNames, GetFieldNames й GetIndexNames компонента SQLConnection.

Компоненти DataSet

dbExpress надає чотири компоненти: SQLDataSet, SQLQuery, SQLStoredProc й SQLTable. SQLDataSet є універсальним для будь-яких прикладних програм. Установкою властивості CommandType можна виконувати оператори SQL, викликати збережені процедури, або вибирати запису із запитів. Інші компоненти DataSet створені для імітації функціональності BDE, наскільки це можливо. Використання цих компонентів значно полегшує перехід з BDE.

Компонент SQLQuery

Властивості й методи компонента SQLQuery дуже схожі на властивості компонента TQuery BDE. Оскільки SQLQuery повертає тільки односпрямований набір записів, властивості й методи, використовувані в BDE для редагування даних, відсутні. Ви можете використати SQLQuery для виконання SQL як операторів DML, так й DDL. Для тих операторів, які повертають набір записів, можна викликати метод Open або встановити властивість Active в True. Для інших операторів використайте метод ExecSQL. Продовжимо створювати нашу прикладну програму:

1. Розмістіть три компоненти SQLQuery на модуль даних
2. Установіть їхню властивість SQLConnection в EmployeeConnection
3. Назвіть перший компонент EmployeeQry, другий HistoryQry, і третій - DeptQry
4. Установіть властивість SQL компонента **EmployeeQry** в

*select * from employee*
where dept_no = :dept_no
order by last_name

5. Установіть властивість SQL компонента **HistoryQry** в

*select * from salary_history*
where emp_no = :emp_no

6. Установіть властивість SQL компонента **DeptQry** в

select dept_no, department from department
order by department

7. Помістіть компонент DataSource на модуль даних, установіть його ім'я в EmpLinkSrc, а властивість DataSet - в EmployeeQry

8. Встановіть властивість компонента HistoryQry.DataSource в EmpLinkSrc

9. Поверніться до EmployeeQry, відкрийте редактор властивості Params, і встановіть значення параметра **dept_no** в XXX. Оскільки це невірне значення, у результаті не буде відображено жодного запису, поки користувач не введе правильний номер відділу.

10. Виконайте подвійний клацання на EmployeeQry, для того щоб відкрити Fields Editor. Додайте всі поля (у меню по правій кнопці add all fields)

11. Виберіть колонка **emp_no**, властивість ProviderFlags, і встановіть pfInKey в True. Установка цього прапора вказує на те, що колонка emp_no є первинним ключем. Компоненту DataSetProvider (який ми додамо пізніше) потрібна ця інформація для генерації операторів SQL, що модифікують дані.

12. Виберіть колонка **full_name**, властивість ProviderFlags, і встановіть **pfInUpdate** й **pfInWhere** в False. Колонка full_name є обчислюваним полем, що, тому не може бути оновлений і не потрібний в умові where SQL операторів, що генерується DataSetProvider.

13. Установіть властивість Active компонента EmployeeQry в True

14. Виконайте подвійне клацання на HistoryQry, і в FieldEditor додайте всі поля (add all fields)

15. Виберіть поле **emp_no** й установіть **pfInKey** в True. Повторіть те ж саме для полів change_date й updater_id, тому що всі ці три поля є первинним ключем.

16. Поле **new_salary** також є обчислюваним, тому для нього теж потрібно встановити pfInUpdate й pfInWhere в False

17. Установіть властивість Active компонента EmployeeQry в False

18. Виконайте подвійне клацання на компоненті DeptQry, додайте всі поля (add all fields)

19. Установіть властивість Connected компонента EmployeeConnection в False

Компонент SQLTable

Цей компонент дуже схожий на TTable з BDE. Як і в BDE, цей компонент повертає всі колонки й всі записи з таблиці, тому він не дуже підходить для прикладних програм клієнт-сервер. Його єдина перевага тільки в тому, що його використання дозволяє швидко перевести на dbExpress прикладні програми, які використовують TTable. Для використання SQLTable потрібно встановити властивість SQLConnection в ім'я компонента SQLConnection, потім указати в TableName ім'я потрібної таблиці, і відкрити компонент методом Open або установкою властивості Active в True.

Компонент SQLStoredProc

Цей компонент призначений для виклику збережених процедур, і може бути використаний, так само як і компонент TStoredProc BDE. Процедури можна викликати й

компонентом SQLDataSet (і SQLQuery). Для використання компонента установіть властивість SQLConnection в ім'я відповідного SQLConnection, і вкажіть в StoredProcName ім'я процедури. Якщо процедура повертає набір записів, то відкрити її можна методом Open або установкою властивості Active в True. Якщо процедура не повертає набір записів, виконати її можна методом ExecProc.

Компонент SQLDataSet

Для використання SQLDataSet приєднаєте його до обраного SQLConnection. Далі, установіть властивість CommandType в один із трьох варіантів - ctQuery, ctStoredProc або ctTable. Найчастіше ви будете використати ctQuery. Значення властивості CommandText залежить від установленого CommandType. Якщо CommandType дорівнює ctQuery, то CommandText повинне містити текст виконуваного запиту SQL. При ctStoredProc CommandText повинен містити ім'я збереженої процедури. При ctTable CommandText повинен містити тільки ім'я таблиці. Властивість Params використається для параметризованих запитів або процедур. Властивість DataSource призначена для зв'язування з іншими SQLDataSet, наприклад, для організації зв'язування майстер-деталь. Якщо запит повертає набір записів, то SQLDataSet відкривається методом Open або установкою Active в True. Якщо немає - запит виконується методом ExecSQL.

Компонент SQLDataSet забезпечує переміщення по записах тільки в одному напрямку. Якщо цього досить, наприклад, для печатки звіту, то можна використовувати SQLDataSet з DataSource. Якщо ж потрібно забезпечити можливість навігації по записах як униз, так і нагору, або редагувати запису, то потрібно додати DataSetProvider або використовувати SimpleDataSet, як це описано далі в цьому документі.

При необхідності одержання більш докладної інформації про метадані, ніж її надає SQLConnection, можна використати метод SQLDataSet.SetSchemaInfo. Цей метод приймає три параметри - SchemaType, SchemaObject й SchemaPattern. SchemaType може бути stNone, stTables, stSysTables, stProcedures, stColumns, stProcedureParams, або stIndexes. Цей параметр визначає тип інформації, що буде виводити SQLDataSet при відкритті. Якщо виконується звичайний запит, то завжди зазначено stNone. Інші типи схем відкривають вибірку з відповідними полями й повертає информацией, що. SchemaObject - ім'я збереженої процедури або таблиці, коли йде запит інформації про цей об'єкт. SchemaPattern - маска для фільтрації результату запиту. Наприклад, якщо SchemaType = stTables й SchemaPattern = 'EMP%', те набір даних буде містити інформацію тільки про ті таблиці, ім'я яких починаються з 'EMP'.

Компонент SimpleDataSet

Для перегляду й редагування записів в BDE досить використати компонент TQuery. В dbExpress необхідно використовувати комбінацію SQLQuery або SQLDataSet,

DataSetProvider й ClientDataSet, з'єднаних разом. Є два способи скоротити час на додавання цих трьох компонентів й установку їхніх властивостей.

Компонент SimpleDataSet уведений в Delphi 7. Він комбінує в собі SQLDataSet, DataSetProvider й ClientDataSet. Якщо вам потрібний компонент для перегляду й редагування даних однієї таблиці, то просто додайте компонент SimpleDataSet на модуль даних, і встановіть властивості CommandType й CommandText. Однак SimpleDataSet має ряд обмежень:

- ви не можете використовувати його в багатоланкових прикладних програмах. Якщо надалі буде потрібно конвертувати прикладну програму у багатоланкову, використовуйте окремі компоненти
- неможливо приєднати дочірній DataSet
- Події вбудованого DataSetProvider не експортовані
- Властивість Options вбудованого DataSetProvider не експортована. Ви не можете встановлювати параметри провайдера під час розробки або виконання
- Ви не можете визначати поля вбудованого DataSet у момент розробки. Це означає, що ви не можете встановити властивість ProviderFlags під час розробки. Вам доведеться робити це в коді прикладної програми
- Якщо ви використаєте Borland MyBase (база даних убудована в ClientDataSet), і не працюєте із сервером баз даних, то краще використати ClientDataSet окремо, для зменшення використовуваних ресурсів
- Властивості й методи вбудованого SQLDataSet не еквівалентні властивостям TQuery. Тому використання SimpleDataSet при переносі проектів з BDE можуть вимагати більше змін у коді

Якщо потрібно більше функціональності, ніж пропонує ClientDataSet, то помістіть компоненти SQLQuery, DataSetProvider й ClientDataSet на форму або модуль даних. Установіть властивість DataSetProvider.DataSet на SQLQuery. Установіть ProviderName на ClientDataSet. Виберіть всі три компоненти, і в головному меню середовища виберіть - Component|Create Component Template. Укажіть ім'я класу, компонент і палітри для нового шаблону. Тепер можна кинути три компоненти на модуль даних так само легко як один компонент.

SQLMonitor

Цей компонент призначений для оптимізації продуктивності ваших прикладних програм. SQLMonitor контролюють всі оператори SQL, передані компонентом SQLConnection серверу баз даних. Інформація може бути збережена у файл, компонент TMemo, або оброблена будь-яким іншим способом.

Компоненти доступу до даних

При необхідності прокручування записів і відновлення даних, потрібно використати компоненти DataSetProvider й ClientDataSet.

DataSetProvider

Цей компонент зв'язується з компонентами DataSet через однойменну властивість. DataSetProvider поставляє дані компоненту ClientDataSet і генерують оператори SQL DML для відновлення бази даних, використовуючи балку змін, що накопичує ClientDataSet.

Повернемося до модуля даних нашого прикладної програми, і проробимо наступні кроки:

1. Додайте *DataSetProvider* з палітри *DataAccess* на модуль даних
2. Його властивість *DataSet* Встановіть в *EmployeeQry*, а ім'я (name) в *EmployeeProv*
3. Властивість *UpdateMode* дозволяє керувати, яким чином провайдер визначає, чи змінений запис іншим користувачем. Коли провайдер генерує оператори SQL для відновлення бази даних, кожен оператор *UPDATE* або *DELETE* включає умову *WHERE* для ідентифікації запису. Якщо *UpdateMode* установлений в *upWhereAll*, оригінальне значення кожного не-блоб поля в записі буде включено в умову WHERE, за винятком тих полів, у яких в *ProviderFlags* установлено *pfInWhere=False*. Це означає, що виконуваний оператор *UPDATE* або *DELETE* видасть повідомлення про помилку, якщо на сервері кожне з полів даного запису було змінено. upWhereChanged включає в WHERE тільки змінені поля. При upWhereKeyOnly генерується WHERE, що включає тільки поля первинного ключа.

Властивість Options містить багато прапорів, що дозволяють управляти процесом provide/resolve. Якщо poCascadeDeletes установлено в True, то провайдер не буде генерувати оператори SQL для видалення записів у таблиці деталей при видаленні записів у майстру-таблиці. Провайдер буде припускати, що сервер баз даних підтримує каскадне видалення, і видалить запису деталей самостійно. poCascadeUpdate забезпечує аналогічну функціональність при зміні первинного ключа майстра-таблиці.

Якщо SQLQuery, що поставляє дані провайдеру, містить запит з ORDER BY, і ви хочете зберегти цей порядок в ClientDataSet, установіть прапор poRetainServerOrder в True. Якщо потрібно мати можливість змінювати текст запиту SQLQuery шляхом зміни властивості ClientDataSet.CommandText - установіть poAllowCommandText в True.

Якщо використається обробник подій BeforeUpdateRecord, і він може змінити значення поля в записі, перед тим як відновлення буде відправлено на сервер, встановіть poPropogateChanges в True. Тепер провайдер буде відсилати зміни назад в ClientDataSet для відновлення записів, збережених у пам'яті.

З множини корисних подій компонента DataSetProvider варто відзначити найбільш важливі - OnGetTableName й BeforeUpdateRecord. У багатотабличних запитах, збережених процедурах або що не редагують view неможливо модифікувати запису прямо. DataSetProvider надає три інструменти для обробки таких ситуацій. Якщо запис включає колонки з однієї таблиці, поверта наприклад збереженою процедурою, єдиною проблемою є те, що DataSetProvider не в змозі визначити, яку таблицю треба обновляти. Рішення криється в створенні обробника OnGetTableName, щоб ця подія повертала ім'я таблиці.

Інша можливість є для відновлення багатотабличних вибірок, де треба обновляти запис тільки у однієї таблиці. Спочатку, Встановіть ProviderFlags конкретних полів для вказівки, що саме ці поля треба обновляти. Потім створіть OnGetTableName, що повертає ім'я потрібної таблиці. Після цього провайдер буде генерувати оператори SQL для відновлення таблиці автоматично. Якщо потрібно обновляти багато таблиць для кожного запису, то додайте подію BeforeUpdateRecord, у якому можна створювати оператори SQL для кожної таблиці й виконувати їх.

Обробник BeforeUpdateRecord також дає можливість перевіряти запис перед її відновленням, і змінювати значення полів. Можна взагалі заблокувати зміна шляхом виклику excepton. Це робить *BeforeUpdateRecord* гарним місцем для застосування бізнесів-правил.

ClientDataSet

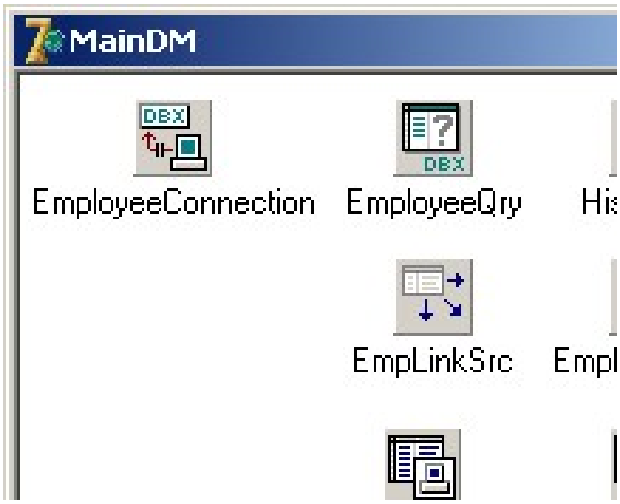
ClientDataSet приєднується до *DataSetProvider* через властивість *ProviderName*. Він одержує дані з *DataSetProvider*, буферизує дані в пам'яті, реєструє всі зміни даних, і відправляє лог змін в *DataSetProvider* при виклику *ClientDataSet.ApplyUpdates*.

Продовжимо роботу над прикладом програми:

1. Помістіть два компоненти *ClientDataSet* на модуль даних
2. Установіть властивість *ProviderName* першого компонента на *EmployeeProv*, і назвіть його *EmployeeCds.*
3. Виконайте подвійне клацання на EmployeeCds, щоб відкрити FieldEditor. Виконаєте Add all fields. Останнє поле має ім'я HistoryQry. Це поле вкладеного DataSet, що містить історію зарплат (salary history) для кожного запису про співробітника (employee). Запису, що повертають HistoryQry видні як вкладені дані усередині даних EmployeeQry. Це відбувається тому, що властивість HistoryQry.DataSource установлено в EmpLinkSrc, що приєднаний до EmployeeQry, і одержує значення параметра запиту із запису EmployeeQry.
4. Клацніть правою кнопкою миші на EmployeeCds і виберіть пункт меню FetchParams, щоб ClientDataSet обновив список параметрів відповідно до компонента EmployeeQry.

5. Назвіть другий ClientDataSet як HistoryCds й установіть його властивість DataSetField в EmploueeCdsHistoryQry. Тепер HistoryCds буде одержувати дані з поля вкладеного набору даних EmployeeCds
6. Виконайте подвійне клацання на HistoryCds, і в Field Editor додайте всі поля (add all fields)
7. Помістіть два компоненти DataSource на модуль даних. Назвіть їх EmployeeSrc й HistorySrc.

Установіть властивість *DataSet* у цих компонентів - в *EmployeeSrc* в *EmployeeCds*, і в *HistorySrc* в *HistoryCds*. Тепер можна встановити властивість *Active* в *True* у компонентів *EmployeeCds* й *HistoryCds*. Модуль даних при цьому буде виглядати в такий спосіб:



Вкладені набори даних - досить потужна річ, оскільки вони виключають нерозуміння порядку відновлень даних майстер- і деталі-таблиці. Наприклад, якщо додані кілька записів у таблиця-майстер і кілька записів у таблицю деталей, то INSERT у таблиця-майстер повинен бути відправлений на сервер БД раніше, ніж INSERT у таблицю деталей. І навпаки, якщо віддаляються записи деталей, а потім запису майстри, то оператор DELETE повинен виконуватися саме в такому порядку. При вкладених наборах даних DataSetProvider забезпечує правильний порядок проходження запитів на відновлення, що виключає помилки на сервері БД.

ClientDataSet має ряд корисних властивостей. Властивість *Aggregates* дозволяє визначити агрегатні колонки, які автоматично будуть уважати суму, мінімум, максимум, кількість або середнє значення будь-якої числової колонки набору даних. Агрегати можуть бути згруповані по будь-якому індексу. Властивість *AggregatesActive* дозволяє управляти станом агрегатів у момент виконання. Для сортування записів у потрібному порядку, укажіть у властивості *IndexFieldNames* імена полів сортування, розділені крапкою з комою. Для сортування по убуванню або для прискорення сортування створіть індекс по потрібних полях, і вкажіть ім'я цього індексу у властивості *IndexName*.

Використайте властивість *CommandText* для зміни запиту SQL у компоненті, що поставляє дані *DataSetProvider*. Закрийте *ClientDataSet,* призначте новий запит в *CommandText*, і відкрийте *ClientDataSet* для одержання результату нового запиту. Також можна призначати нові значення параметрам у джерелі записів, використовуючи властивість *ClientDataSet.Params.* Чому бажано використати *CommandText* й *Params* замість звертання до конкретних властивостей компонента, що поставляє дані (наприклад, SQLQuery)? Справа в тому, що при такому написанні коду його не прийде міняти, якщо згодом буде потрібно переписати прикладну програму на багатоланкову архітектуру (Borland DataSnap).

Властивість *PacketRecords* дозволяє управляти кількістю записівів, що провайдеру вибирає із сервера за один обіг. Значення за замовчуванням -1 указує провайдеру вибирати всі записи, що повертають компонентом, що виконує SQL-запит. Звичайно це нормально, але якщо потрібно обробити велику кількість записів, то може знадобитися вибирати їхніми невеликими групами.

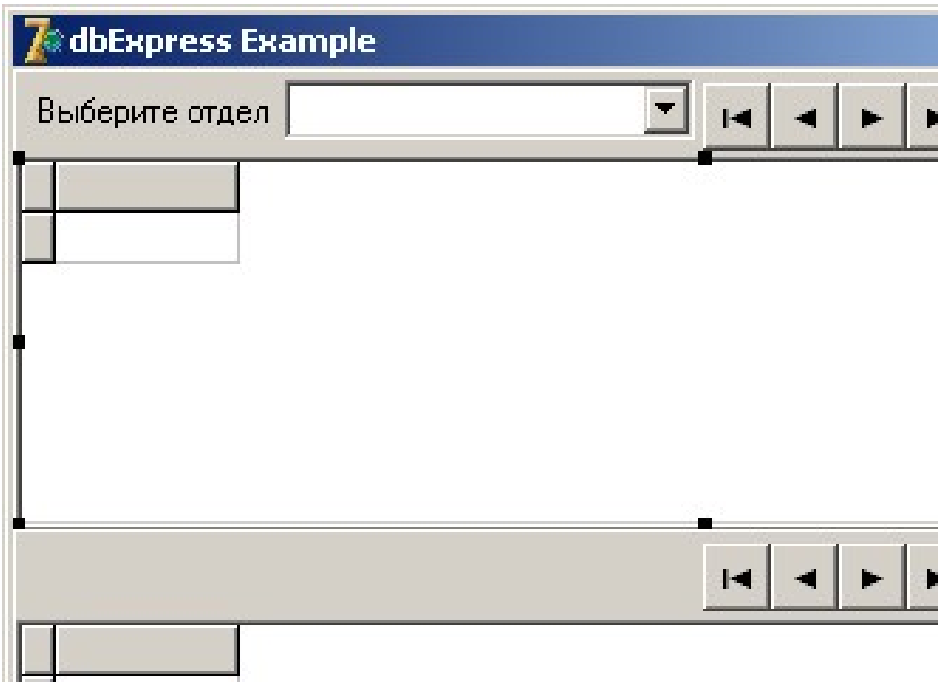
Одне із самих більших відмінностей BDE й архітектури provider/resolver у тому, що ви повинні викликати *ClientDataSet.ApplyUpdates* для застосування змін до бази даних. Використовуйте властивість ChangeCount для перевірки наявності невідправлених змін. *ApplyUpdates* має один параметр, що вказує припустиму кількість помилок, після якого процес відновлення буде перерваний, і транзакція буде скасована по rollback. Звичайно передається 0, тому процес відновлення переривається при першій же помилці. Передача -1 указує на необхідність відправлення всіх змін, незалежно від кількості виникаючих помилок.

Якщо при відновленні даних виникає помилка, відбувається подія *ClientDataSet.OnReconcileError*. Для обробки помилок у нашій програмі-прикладі виконайте наступне:

1. Додайте в проект *Reconcile Error Dialog* із закладки *Dialogs* репозитария об'єктів
2. Додайте ім'я нового модуля в оператор USES модуля даних.
3. У властивостях проекту переконаєтеся, що модуль *Reconcile Error Dialog* не створюється автоматично при запуску прикладної програми
4. Створіть обробник *OnReconcile* для *EmployeeCds*, і додайте туди рядок *Action := HandleReconcileError(DataSet, UpdateKind, E);*

Якщо в момент застосування змін виникне помилка, то буде показаний діалог *Reconcile Error*, у ньому відображені записи, що привели до повідомлення про помилку, і набір кнопок, які дають користувачеві можливість обробити помилку. Щоб завершити прикладну програму, проробіть наступне:

1. Додайте два компоненти Panel, два DBGrid і два DBNavigator на головну форму прикладної програми, тому що це показано на картинці



2. Додайте ім'я модуля даних в USES головної форми

3. Установіть властивість DataSource верхнього DBGrid й DBNavigator в EmployeeSrc

4. Установіть властивість DataSource нижнього DBGrid й DBNavigator в HistorySrc

5. Додайте Label й ComboBox

6. Створіть подію OnChange для ComboBox з наступним текстом

```
procedure TMainForm.DeptComboChange(Sender: TObject);
begin
  with MainDm.EmployeeCds do
    begin
      if Active then CheckBrowseMode;
      Close;
      Params.ParamByName('DEPT_NO').asString :=Copy(DeptCombo.Text, 1, 3);
      Open;
    end; //with
end;
```

7. Додайте кнопку (Button) на верхню панель, і встановіть її напис в "Зберегти".

Додайте наступний код по OnClick для цієї кнопки

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

```
begin
  with MainDm do
    begin
      if EmployeeCds.ChangeCount > 0 then
        begin
          if HistoryCds.Active then
            HistoryCds.CheckBrowseMode;
          if EmployeeCds.Active then
            EmployeeCds.CheckBrowseMode;
          EmployeeCds.ApplyUpdates(0);
          EmployeeCds.Refresh;
        end; //if
    end; //with
end;
```

8. Додайте метод до форми MainForm для заповнення списку вибору номерів відділів

```
procedure TMainForm.LoadDeptCombo;
begin
  with MainDm do
    begin
      DeptQry.Open;
      while not DeptQry.Eof do
        begin
          DeptCombo.Items.Add(DeptQryDept_No.asString +
           ' ' + DeptQryDepartment.asString);
          DeptQry.Next;
        end; // while
      DeptQry.Close;
    end; // with
end;
```

9. Додайте обробник OnCreate для головної форми

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  MainDm.EmployeeCds.Open;
  LoadDeptCombo;
end;
```

прикладна програма готова.

16.

17. IV Виконання роботи

18. Завдання на виконання

19. Розробити прикладну програму Delphi відповідно до варіанта:

1. Підключіться до існуючої БД, використовуючи компоненти dbExpress SQLConnection й SimpleDataSet. Вивести вміст однієї їхній таблиць. Спробувати наступне:
    1) змінити поле
    2) додати поле
    3) видалити поле.
2. Підключіться до існуючої БД, використовуючи компоненти dbExpress. Вивести вміст однієї з таблиць. Створити запит за допомогою SQLQuery, у результаті якого повертаються дані з бази.
3. Підключіться до двох існуючих схожихо за структурою таблиць у БД, використовуючи компоненти dbExpress. Організуйте імпорт певних полів однієї таблиці в іншу.
4. Підключіться до існуючої БД, використовуючи компоненти dbExpress SQLConnection й SimpleDataSet. Вивести вміст однієї з таблиць. Створити запит, у результаті якого повертаються дані з бази.
5. Підключіться до існуючої БД, використовуючи компоненти dbExpress SQLConnection й SimpleDataSet. Організуйте контроль операторів SQL за допомогою SQLMonitor.
6. Підключіться до існуючої БД, використовуючи компоненти dbExpress SQLConnection й SimpleDataSet. Організуйте сортування колонок по натисканню відповідної кнопки.
7. Підключіться до існуючої БД, використовуючи компоненти dbExpress SQLConnection й SimpleDataSet. Організуйте виведення кожної таблиці БД на окремій формі.
8. Підключіться до існуючої БД, використовуючи компоненти dbExpress SQLConnection й SimpleDataSet. Організуйте фільтрацію дані таблиці по натисканню відповідної кнопки.
9. Підключіться до існуючої БД, використовуючи компоненти dbExpress. Організуйте синхронне виведення полів зі зв'язаних таблиць, причому на одній формі повинна відображатися тільки «основна» таблиця, а ланцюжок зв'язаних полів відображається на окремій формі.

10. Підключіться до існуючої БД, використовуючи компоненти dbExpress. Використовуючи компонент SQLquery, створити прикладну програму, в якій реалізуються SQL-запити, причому код SQL-запиту вводиться в окремому компоненті Memo і після натискання кнопки в таблиці виводиться результат обробки запиту або повідомлення про помилку.

3. Завдання на виконання

Створити локальні бази даних в середовищі Delphi(спроектувати схеми відношення таблиць, інтерфейс для вводу даних, запити до БД, звіти) для згідно варіанту . Узгодьте з викладачем зміст вхідних таблиць

Лабораторна робота №3

1. Назва: Основні компоненти ADO та робота з ними

2. Мета: Вивчити механізм дії технології ADO та оволодіти навичками організації обробки даних з використанням ADO

3.Методичні вказівки по виконанню лабораторної роботи

3.1.Огляд компонентів

Для роботи з ADO на вкладці компонентів ADO є шість компонент: TADOConnection, TADOCommand, TADODataSet, TADOTable, TADOQuery, TADOStoredProc.

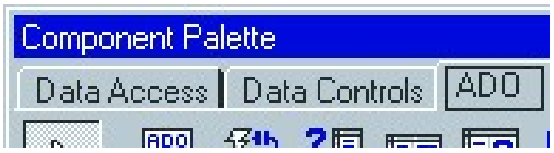


Рис. 1. Палітра компонентів ADO

Як розробляти прикладну програму, що буде використовувати обробку даних в середовищі Delphi/C++Builder, використовуючи технологію ADO/dbGO? Як і у кожній з можливих технологій середовища Delphi/C++Builder треба встановити з'єднання з фізичним рівнем БД, тобто через компоненти зв'язуватися з базою даних. Виходячи з особливостей об'єктної моделі ADO/dbGO це можна зробити двома способами:

- через компонент TADOConnection;
- або прямим визначенням бази даних в інших компонентах.

До TADOConnection інші компоненти прив’язуються за допомогою властивості Connection(як правило автоматично), до бази даних прямо через властивість ConnectionString.

В свою чергу в кожному випадку база даних може бути зазначена теж двома способами: через файл лінку до даних (файл у форматі Microsoft Data Link, розширення UDL), або прямим визначенням параметрів з'єднання.

Значення властивості усіх ConnectionString компонентів можуть бути задані прямо в текстовій формі, але зручніше викликати редактор властивості натиснувши на кнопку “...” наприкінці поля вводу. Вікно цієї властивості виглядає так:

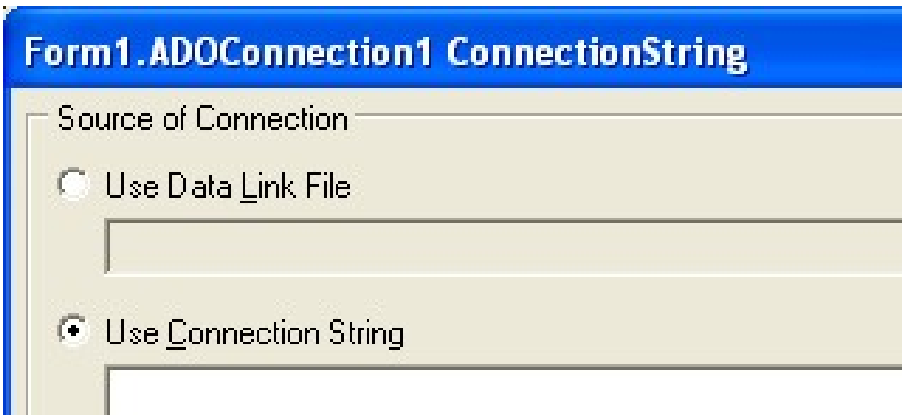


Рис. 2. Вікно з’єднання

При виборі “Use data link file” і натисканні на кнопку “Browse...” з'являється стандартний діалог вибору файлу. Цей файл можна створити в будь-якому вікні explorer-а (у цьому вікні відкриття файлу, у самому explorer, на desktop і т.д.) викликавши контекстне меню й вибравши пункт “New/Microsoft Data Link”. Потім треба викликати локальне меню для створеного файлу й вибрати у ньому пункт “Open”. Після цього з'явиться property sheet описаний трохи нижче. Ці ж вкладки містить й *property sheet*, які викликаються через пункт “Property” локального меню UDL файлу, але в ньому ще є вкладки які стосуються самого файлу.

Використання файлів Microsoft Data Link спрощує підтримку прикладних програм. При виборі в редакторі властивості “Use connection string” і натисканні на кнопку “Build...” з'являється такий самий property sheet, як і при виборі “Open” для Microsoft Data Link файлу.

У цьому вікні вибирається тип бази даних, місце розташування бази й параметри з'єднання.

На першій сторінці вибирається тип бази даних або Provider, у термінах ADO.

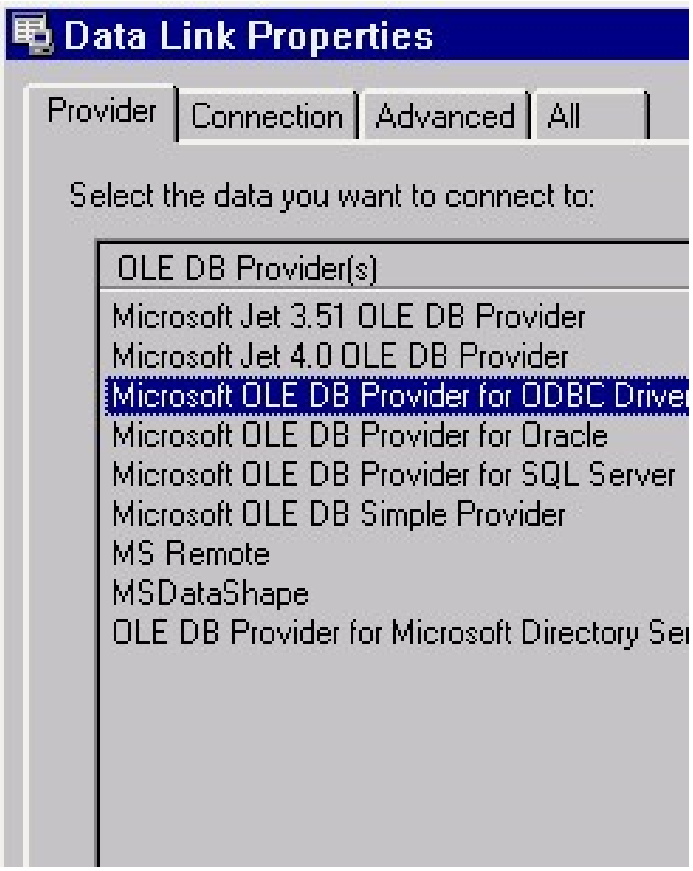


Рис. 3.

Провайдери OLE DB

Провайдери OLE DB забезпечують доступ до джерел даних. У процесі установки MDAC у системі автоматично встановлюються провайдери OLE DB, перераховані в табл. 1.

Таблиця 1 Провайдери OLE DB, що входять до складу MDAC

Драйвер	**Провайдер**	**Опис**
MSDASQL	ODBC Drivers	Драйвери ODBC (за замовчуванням)
Microsoft.Jet.OLEDB.3.5	Jet 3.5 Тільки бази даних MS Access 97	
Microsoft.Jet.OLEDB.4.0	Jet 4.0	Бази даних MS Access та інші БД
SQLOLEDB	SQL Server	Бази даних MS SQL Server
MSDAORA	Oracle	Бази даних Oracle
MSOLAP	OLAP Services	Online Analytical Processing
SampProv	Sample provider	Приклад провайдера OLE DB для файлів CSV
MSDAOSP	Simple provider	Для створення ваших власних провайдерів для простих текстових даних

- ODBC OLE DB використовується для зворотної сумісності з ODBC.
- Jet OLE DB - підтримка MS Access і інших локальних баз даних. Ми повернемося до розгляду цих провайдерів далі.
- SQL Server забезпечує взаємодію з SQL Server 7, SQL Server 2000 і Microsoft Database Engine (MSDE). MSDE - це спрощена версія SQL Server, у якій відсутня більшість інструментів, а крім того, доданий спеціальний код, що навмисно знижує продуктивність у випадку, якщо до бази даних одночасно підключаються більше п'яти користувачів. До переваг MSDE варто віднести те, що цей механізм поширюється безкоштовно й повністю сумісний з SQL Server.
- OLE DB для OLAP може використовуватися прямо, однак частіше звертання до нього здійснюється через ADO Multi-Dimentional (ADOMD). ADOMD - це додаткова технологія ADO, спеціально розроблена для Online Analytical Processing (OLAP). Наприклад, Delphi Decision Cube, Excel Pivot Tables або Access Cross Tabs . Крім уже перерахованих тут провайдерів, компанія Microsoft здійснює підтримку деяких інших провайдерів OLE DB, які входять до складу інших продуктів або до складу SDK.
- Active Directory Services OLE DB входить до складу ADSI SDK; AS/400 OLE DB і VSAM OLE DB входять до складу SNA Server; Exchange OLE DB входить до складу Microsoft Exchange 2000.
- Indexing Service OLE DB входить до складу Microsoft Indexing Service - внутрішній механізм Windows, що прискорює пошук інформації у файлах за допомогою побудови каталогу з файловою інформацією. Служба індексування Indexing Service інтегрована в IIS і часто використовується для індексування веб-вузлів.
- Internet Publishing OLE DB дозволяє розробникам маніпулювати каталогами й файлами з використанням HTTP.
- Існує також категорія провайдерів OLE DB, які називаються провайдерами обслуговування (service providers). Як треба з імені, ці провайдери забезпечують обслуговування інших провайдерів OLE DB і найчастіше активізуються автоматично без участі програміста. Наприклад, служба Cursor Service активізується у випадку, якщо створюється курсор на боції клієнта, а провайдер Persistent Recordset активізується у випадку, якщо ви збираєтеся зберігати дані на локальному диску.

Крім перерахованих, існує також величезна кількість інших провайдерів OLE DB для MDAC. Провайдери OLE DB можна отримати як від Microsoft, так і від незалежних виробників. Список провайдерів OLE DB дуже великий і постійно змінюється, тому його неможливо відтворити повністю. Крім незалежних виробників постачання й підтримку провайдерів OLE DB здійснюють багато виробників систем RDBMS. Наприклад, компанія Oracle підтримує власний провайдер OLE DB за назвою ORAOLEDB.

Ви вже, напевно, звернули увагу на те, що в списку відсутній провайдер OLE DB для InterBase. По-перше, ви можете скористатися драйвером ODBC, по-друге, ви можете використати провайдер IBProvider, розроблений Дмитром Коваленко (www.lipetsk.ru/prog/eng/index.html).

Використання механізму Jet

Зараз важко знайти користувачів, які б не використовували дані хоча б одного з форматів: Access, Paradox, dBase, Excel, Lotus 1-2-3, HTML або дані, що зберігаються в текстових файлах. Як правило, механізм Jet асоціюється з базами даних Microsoft Access. Дійсно, Access є основною системою, з якої взаємодіє Jet. Однак крім Access механізм Jet дозволяє працювати з множиною інших локальних джерел даних. Багато хто не підозрює про це, однак саме в цьому полягає одне з основних переваг Jet. Взаємодія з Access через Jet у стандартному режимі роботи цього механізму виконується відносно просто, тому тут ми не будемо розглядати цей режим використання Jet. Замість цього ми докладно розглянемо взаємодію Jet з іншими форматами.

Існує два провайдери OLE DB для механізму Jet: Jet 3.51 OLE DB і Jet 4.0 OLE DB. Провайдер Jet 3.51 OLE DB використає Jet 3.51 і підтримує роботу тільки з Access 97. Якщо ви будете застосовувати тільки Access 97 і не збираєтеся переходити на Access 2000, то Jet 3.51 у більшості випадків дасть більше високу продуктивність у порівнянні із провайдером Jet 4.0 OLE DB.

Провайдер Jet 4.0 OLE DB підтримує роботу з Access 97, Access 2000 і із драйверами IISAM (Installable Indexed Sequential Access Method). Встановлювані драйвери ISAM спеціально написані для механізму Jet і забезпечують доступ до таких форматів, як Paradox, dBase і текстові файли. Саме можливість використання цих драйверів робить Jet корисним і зручним інструментом. Повний список драйверів ISAM, установлених на вашому комп'ютері, визначається набором встановленого в системі програмного забезпечення. Цей список розташовується в реєстрі за адресою:

HKEY_LOCAL_MACHINE\Software\Microsoft\Jet\4.0\ISAM Formats

До складу комплекту постачання Jet входять драйвери для Paradox, dBase, Excel, текстових файлів і HTML.

Імпорт і експорт

Механізм Jet зручно використати для імпорту й експорту даних. Процес експортування даних однаковий для кожного експортованого формату й складається з виконання виразу SELECT у спеціальному форматі. Розглянемо приклад експортування даних з бази даних Access у прикладі DBDemos у таблицю Paradox. Для цього додамо в програму JetImportExport активне з'єднання ADOConnection з назвою ADOConnection1. Це з'єднання

використовує механізм Jet для того, щоб відкрити базу даних. Наступний код експортує таблицю Customer у файл Customer.db формату Paradox:

*SELECT * INTO Customer IN "C:\tmp" "Paradox 7.x;" FROM CUSTOMER*

Розглянемо складові частини цього SQL-виразу. Після ключового слова INTO вказується нова таблиця, що буде створена в результаті виконання оператора SELECT. До виконання цього коду таблиця із цим ім'ям повинна бути відсутня у базі. Після ключового слова IN вказується база даних, у яку додається нова таблиця. В Paradox це повинен бути каталог, що вже існує на диску. Відразу ж після імені бази даних вказується ім'я драйвера IISAM, що буде використатися для експорту даних. Наприкінці імені драйвера обов'язково потрібно додати символ крапки з комою (;). Ключове слово FROM є стандартним компонентом будь-якого виразу SELECT. У розглянутому прикладі ця операція виконується за допомогою компонента ADOConnection1, замість фіксованого імені каталогу використовується поточний каталог програми:

*ADOConnection1.Execute ("SELECT * INTO Customer IN "' +CurrentFolder + '" "Paradox 7.x;" FROM CUSTOMER');*

Такий же формат використовується в будь-яких операціях експорту, однак залежно від використовуваного драйвера IISAM ім'я бази даних інтерпретується по-різному. От аналогічне виразу, що експортує дані в таблицю Excel:

*ADOConnection1.Execute ('SELECT * INTO Customer IN "' +CurrentFolder + 'dbdemos.xls" "Excel 8.0;" FROM CUSTOMER');*

Новий файл Excel з ім'ям dbdemos.xls створюється в поточному каталозі програми. У цей документ Excel додається робоча книга з ім'ям Customer, у яку заносяться всі дані з таблиці Customer бази даних Access з ім'ям dbdemo. mdb.

От ще однин вираз, що експортує ті ж самі дані в HTMLфайл:

*ADOConnection1.Execute ('SELECT * INTO [customer.htm] IN "' +CurrentFolder + '" "HTML Export;" FROM CUSTOMER');*

У цьому випадку база даних — це каталог (як і в Paradox). Ім'я таблиці містить у собі розширення .htm, тому ім'я таблиці необхідно взяти у квадратні дужки. Зверніть увагу, що драйвер IISAM називається не просто HTML, а HTML Export. Як видно з назви, драйвер дозволяє тільки експортувати дані, але не дозволяє імпортувати їх.

Нарешті, давайте розглянемо вхідний до складу Jet драйвер HTML Import, що є корисним доповненням до HTML Export. Додайте на форму компонентів ADOTable. Налаштуйте рядок підключення *ConnectionString* на використання провайдера Jet 4.0 OLE DB. Привласніть параметру Extended Properties рядка підключення значення HTML Import. Як ім'я бази даних вкажіть ім'я HTML-файлу, що був створений у результаті експорту (трохи раніше), точніше кажучи, Customer.htm. Тепер привласніть властивості TableName значення Customer.

Відкрийте таблицю - ви тільки що імпортували дані з HTML-файлу. Майте на увазі, що якщо ви спробуєте обновити дані, система видасть помилку, тому що драйвер призначений тільки для імпорту. Якщо ви створили власний HTMLфайл, у якому містяться таблиці, і хочете відкрити ці таблиці з використанням даного драйвера, ви повинні пам'ятати, що ім'я таблиці - це значення тегу caption в HTML-розділі table.

Бази MS Access доступні як через “Microsoft Jet OLE DB Provider”, так і через “Microsoft OLE DB Provider for ODBC”.

Наступна сторінка залежить від обраного типу бази, однак для всіх типів є кнопка “Test connection” що дозволяє перевірити правильність і повноту параметрів.

Для “Microsoft Jet OLE DB Provider” вона виглядає так:

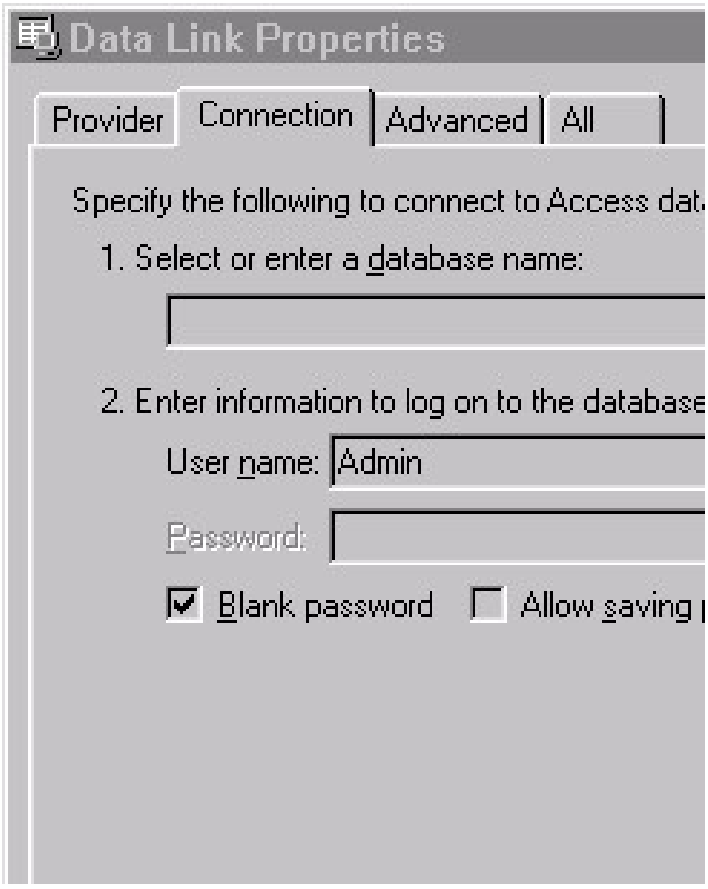


Рис. 4.

Checkbox “Blank password” придушує діалог вводу ідентифікатора й пароля користувача при встановленні з'єднання, якщо поле пароля порожнє.

Checkbox “Allow saving password” зберігає пароль у рядку параметрів з'єднання. Якщо він не позначений, то введений пароль буде використовуватися тільки при виконанні тестового з'єднання.

Для “Microsoft OLE DB Provider for ODBC” ця сторінка виглядає так:

Радіокнопка “Use data source name” дозволяє ввести попередньо встановлений аліас ODBC, а через “Use connection string” вводяться як аліаси так і тип ODBC драйвера й параметри з'єднання.

Параметри ідентифікації користувача аналогічні вище описаним.

На сторінці “Advanced” розташовані додаткові параметри, за допомогою яких встановлюється рівень доступу до файлу бази даних, таймаут мережевого з'єднання (тобто час через який зв'язок буде вважатися загубленим, якщо сервер не відповідає) і рівень глибини перевірки таємності з'єднання.

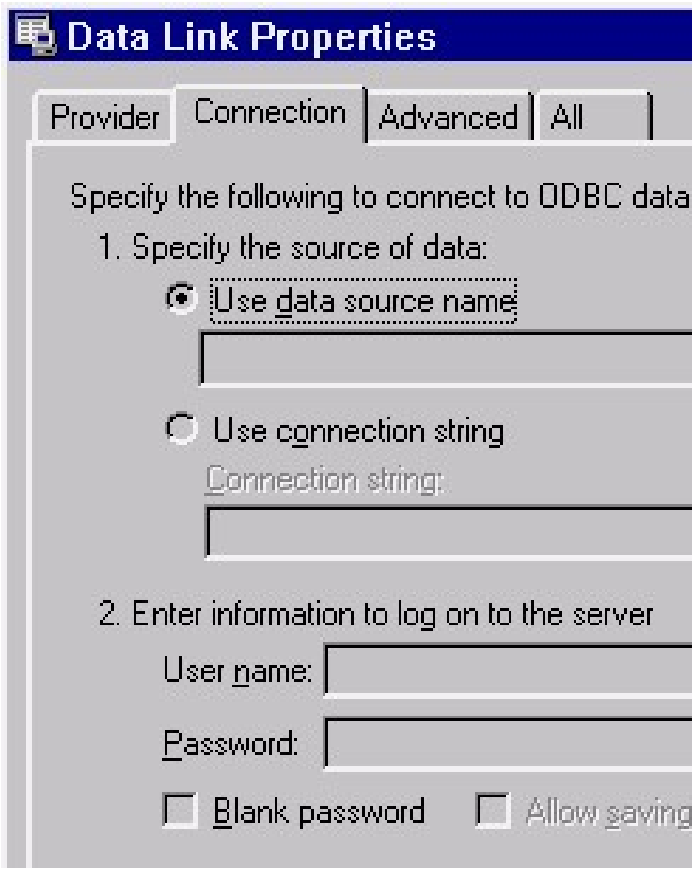


Рис. 4.

На сторінці “All” можна відредагувати всі параметри з попередніх сторінок і параметри які залежать від провайдера, але не ввійшли на сторінку “Connection”. Редагування здійснюється у вигляді параметр - значення, причому в текстовій формі, ніяких діалогів немає. Допомоги теж немає, ці параметри описані тільки в документації на

провайдер. Її можна знайти в MSDN Data Access Services/Microsoft Data Access Components (MDAC) SDK/Microsoft Active Data Objects (ADO)/Microsoft ADO Programmer's Reference/Using Providers with ADO.

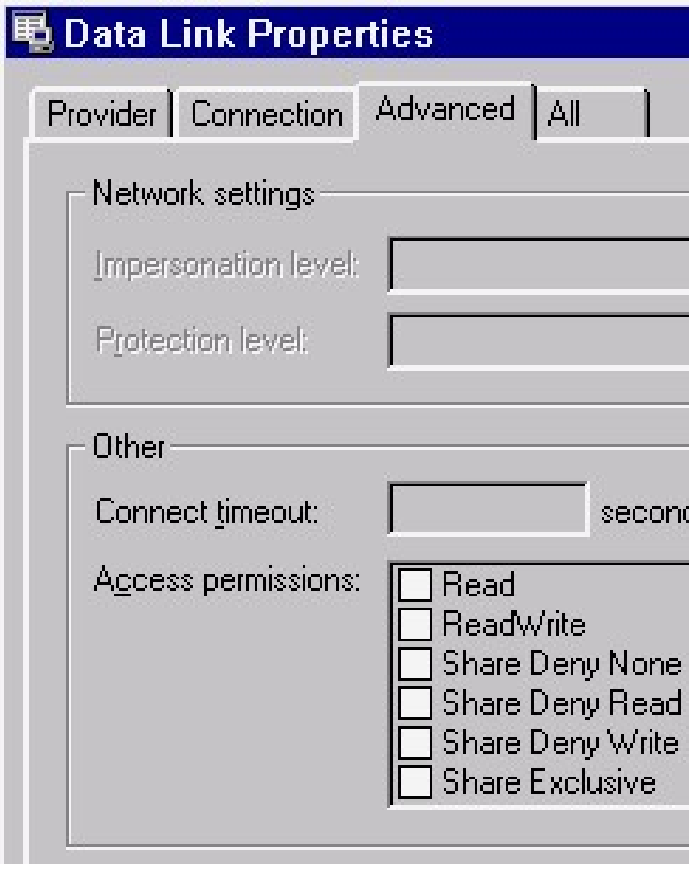


Рис. 5.

3.2 Механізм з'єднання зі сховищем даних ADO

Властивість *Connectionstring* призначена для зберігання інформації про з'єднання з об'єктом ADO. У ньому через крапку з комою перераховуються всі необхідні параметри. Як мінімум, це повинні бути імена провайдера з'єднання або віддаленого сервера:

Connectionstring:='Remote Server=ServerName;Provider=ProviderName';

При необхідності вказуються шлях до віддаленого провайдера:

Connectionstring:='Remote Provider=ProviderName';

і параметри, необхідні провайдеру:

'User Name=User_Name;Password=Password';

Кожний компонент, що звертається до сховища даних ADO самостійно, задаючи параметри з'єднання у властивості *Connectionstring*, відкриває власне з'єднання. Чим більше

прикладна програма містить компонентів ADO, тим більше з'єднань може бути відкрито одночасно.

Тому доцільно реалізувати механізм з'єднання ADO через спеціальний компонент - TADOConnection. Цей компонент відкриває з'єднання, також задане властивістю Connectionstring (див. вище), і надає розробнику додаткові засоби керування з'єднанням.

Компоненти, що працюють зі сховищем даних ADO через дане з'єднання, підключаються до компонента TADOConnection за допомогою властивості *property Connection: TADOConnection;* яку має кожний компонент, що інкапсулює набір даних ADO.

3.2.1Налаштування з'єднання

Перед відкриттям з'єднання необхідно визначити його параметри. Для цього призначена властивість *property ConnectionString: WideString;* . Додамо лише, що набір параметрів змінюється залежно від типу провайдера й може налаштовуватися як вручну, так і за допомогою спеціального редактора параметрів з'єднання, що викликається подвійним клацанням на компоненті TADOConnection, перенесеним на форму, або клацанням на кнопці в полі редагування властивості ConnectionString в Інспекторі об'єктів

Тут можна налаштувати з'єднання через властивість ConnectionString (радіокнопка **Use Connection String)** або завантажити параметри з'єднання з файлу з розширенням udl (радіокнопка **Use Data Link File).**

Файл UDL (лістинг 1) являє собою звичайний текстовий файл, у якому вказується назва параметра й через знак рівності його значення. Параметри розділяються крапкою з комами.

Лістинг .1 Демонстраційний файл DBDEMOS.UDL

[oledb]

Everything after this line is an OLE DB initstring

Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Program Files\Common Files\Borland Shared\Data\DBDEMOS.mdb

Якщо файл параметрів з'єднання відсутній, налаштування доведеться здійснювати вручну. Для цього треба натиснути кнопку **Build.** У результаті з'явиться діалогове вікно **Data Link Properties,** у якому можна налаштувати параметри з'єднання вручну. Воно являє собою чотирьохсторінковий блокнот, що дозволяє поетапно визначи

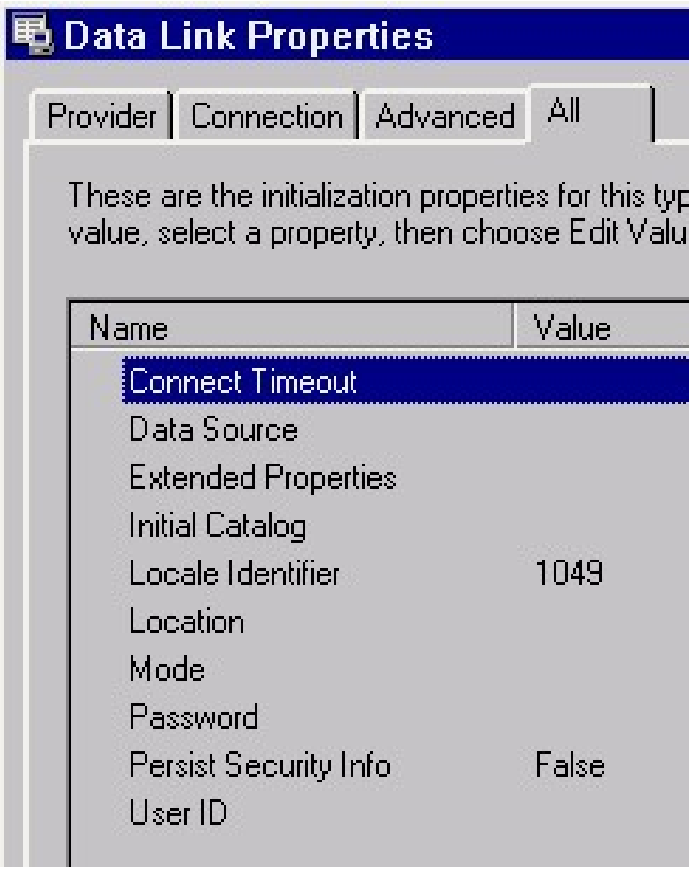


Рис. 6.

У компоненті TADOConnection є властивості Provider, DefaultDatabase й Mode які є альтернативним методом задання частин рядка параметрів з'єднання - провайдера, бази даних (наприклад, шляхи до бази MS Access) і режиму спільного використання файлів бази даних. Значення цих властивостей автоматично включаються в рядок з'єднання, якщо були задані до активізації компонента й автоматично виставляються після з'єднання.

3.3 Приклади створення прикладних ADO програм

3.3.1 Найпростіша прикладна ADOпрограма з використанням TADOTable

Створимо найпростішу програму, що складається з однієї таблиці.

Створюємо форму, що складається із трьох компонентів :

- TADOTable з палітри компонентів ADO,
- TDataSource з палітри компонентів Data Access,
- TDBGrid з палітри компонентів Data Controls.

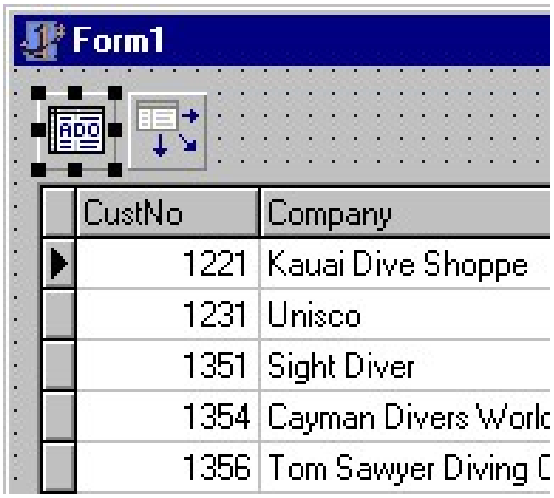


Рис 7

Зв’яжемо компоненти, встановлюючи властивість TDBGridDataSource на компоненті TDataSource, властивість DataSet компонента TDataSource на компоненті TADOTable.

Тепер нам необхідно вказати базу даних. Робиться це у властивості ConnectionString компонента TADOTable. При натисканні на кнопку “...” з'явиться редактор параметрів з'єднання. Позначимо радіокнопку “Use data link file”, натисніть на кнопку “Browse...” і виберіть у вікні, що з'явилося, після файл лінка до бази даних “\Program Files\Common Files\System\ole db\Data Links\DBDEMOS.UDL”. Цей лінк вказує на базу у форматі MS Access, що входить в поставку Delphi.

Після цього у властивості TableName компонента TADOTable виберемо таблицю customer.

Активізуємо компонент TADOTable, встановивши властивість Active в True.

Програму можна запускати на виконання.

3.3.2Приклад використання TADOConnection

У цьому прикладі розглядається робота з компонентом TADOConnection, SQL запитами з параметрами й транзакціями.

Створимо програму з наступних компонентів:

- Connect типу TADOConnection
- MasterSQL й DetailSQL типу TADODataSet
- MasterDS й DetailDS типу TDataSource
- MasterGrid й DetailGrid типу TDBGrid

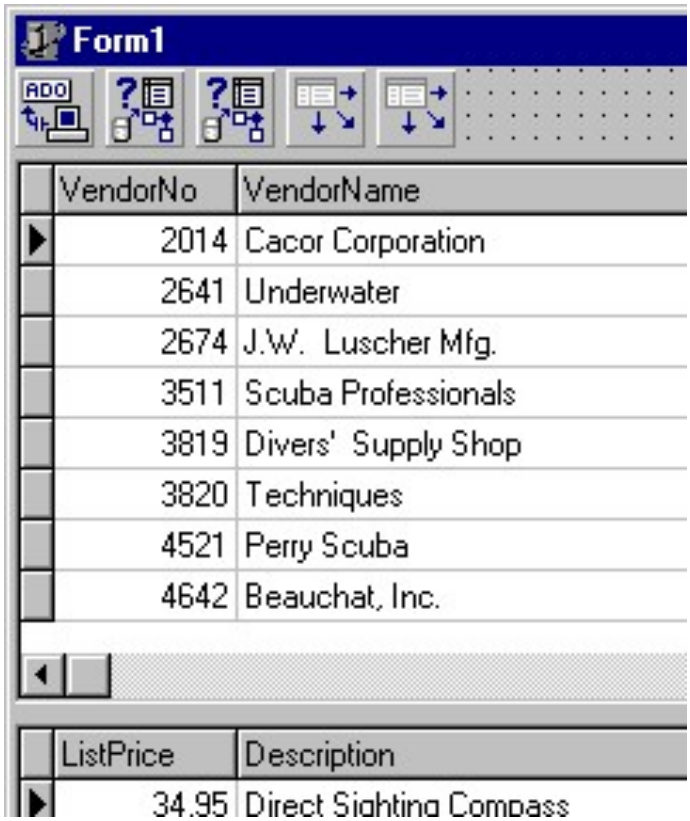


Рис 8. Master-detail форма на етапі дизайну

Зв'язуємо MasterGrid, MasterDS, MasterSQL й DetailGrid, DetailDS, DetailSQL аналогічно попередньому прикладу, за винятком того, що замість типу TADOTable використовується тип TADODataSet.

Прив'язуємо Connect до бази даних. Для цього в редакторі властивості ConnectionString вибираємо ту ж базу даних, що й у попередньому прикладі.

Для вводу SQL запитів необхідно відредагувати властивість CommandText в компонентах MasterSQL й DetailSQL. Після натискання на кнопку "..." з'явиться редактор компонентів, що виглядає так :

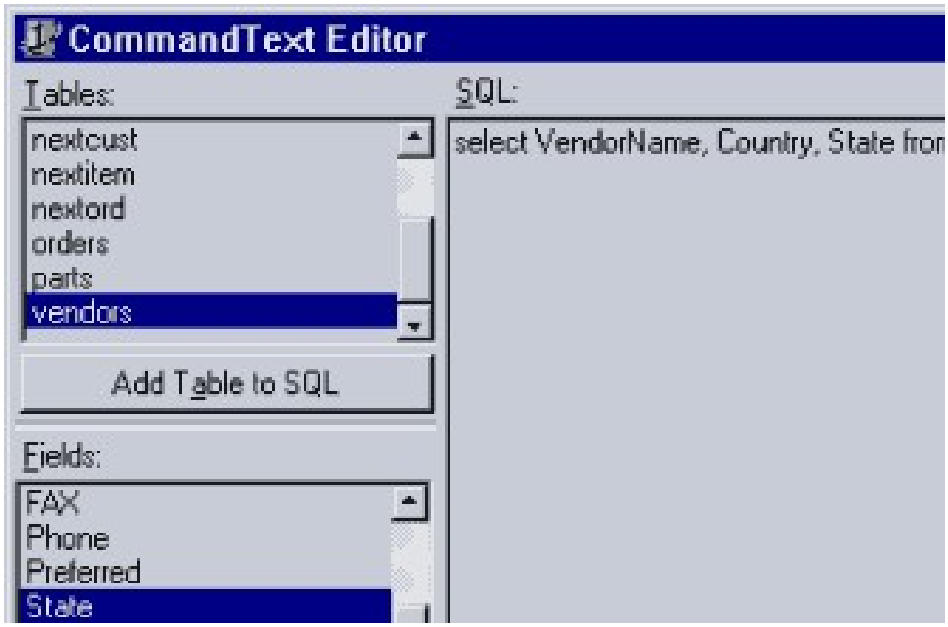


Рис. 9

Кнопка “Add Table to SQL” додає в текст SQL запиту таблицю, обрану в списку “Tables”, а “Add Field to SQL” поле таблиці, обране в списку “Fields”.

Запит для **MasterSQL**

select VendorNo, VendorName, Country, City, State, Preferred
from vendors

Запит в DetailSQL повинен вибирати тільки ті деталі, постачальник яких є поточним в MasterSQL. Для цього встановимо властивість DataSource компонента DetailSQL у значення MasterDS.
Запит для DetailSQL наступний:
select PartNo, OnOrder, OnHand, ListPrice, Description, Cost
from parts
where VendorNo = :VendorNo

:VendorNo у частині where - параметр запиту. Параметри при встановленому DataSource беруться з нього.

Активізуємо MasterSQL й DetailSQL аналогічно попередньому прикладу.

3.3.3 Приклад використання параметрів запиту

Тепер обмежимо вибірку постачальників за значенням поля State. Для цього додамо до форми наступні компоненти StateEdit типу TEdit з вкладки Standard, QueryButton типу TButton з вкладки Standard

Змінимо запит в MasterSQL на

```
select VendorNo, VendorName, Country, City, State, Preferred
from vendors
where State = :StateID
```

:StateID - параметр, замість якого при виконанні підставляється значення.

Додамо так само обробник події OnClick в QueryButton наступного змісту

```
procedure TForm1.QueryButtonClick(Sender: TObject);
begin
 MasterSQL.Active := False;
 DetailSQL.Active := False;
 MasterSQL.Parameters.ParamByName('StateID').Value := StateEdit.Text;
 MasterSQL.Active := True;
 DetailSQL.Active := True;
end;
```

3.4 Робота з транзакціями

В розділі механізмам транзакцій при використанні компонент ADO ,була приділена значна увага. Нижче розглянемо деякі приклади механізмів транзакцій при використанні ADO.

3.4.1.Приклад роботи з транзакціями

За базовий візьмемо приклад використання TADOConnection.

Додамо до форми дві кнопки (StCmButton й RollbackButton) типу TButton, обробники подій OnClick цих кнопок, процедуру fix_controls без параметрів до форми, обробник події OnActivate форми

```
 type TForm1 = class(TForm)
 ...
 private
```

```
procedure fix_controls;
...
procedure TForm1.....fix_controls;
  begin
if Connection.InTransaction then
begin
  StCmButton.Caption := 'Commit';
  RollbackButton.Enabled := True;
end
else
begin
  StCmButton.Caption := 'Begin';
  RollbackButton.Enabled := False;
end;
MasterSQL.Requery;
DetailSQL.Requery;
  end;
procedure TForm1.StCmButtonClick(Sender: TObject);
begin
if not Connection.InTransaction
  then Connection.BeginTrans
  else Connection.CommitTrans;
fix_controls;
end;
procedure TForm1.RollbackButtonClick(Sender: TObject);
begin
Connection.RollbackTrans;
  fix_controls;
end;
procedure TForm1.FormActivate(Sender: TObject);
begin
fix_controls;
  end;
```

Зверніть увагу на те, навіть коли транзакція не запущена, можна змінювати дані й вони записуються одразу.

3.5.Доступ до даних

Технологія ADO підтримує більше налаштувань роботи даних ніж BDE, .

В ADO є поняття набору даних (recordset) і тісно пов'язане з ним поняття курсору (cursor). Що таке курсор у документації на ADO не описано. Фактично місце розташування набору даних називається положенням курсору, можливо курсор і є набір даних.

У всіх компонентах, які мають набір даних (тобто в TADODataSet, TADOTable, TADOQuery, TADOStoredProc) є властивості CursorLocation, CursorType, LockType й MarshalOptions, які встановлюють параметри обміну з сервером. Всі ці властивості повинні бути встановлені до того, як відкривається набір даних . Якщо ви встановите їх пізніше, то ефекту не буде.

CursorLocation - визначає, де виконується робота з набором на клієнті (clUseClient) або на сервері (clUseServer). Якщо набір даних розташований на клієнті, то з сервера дані запитуються однократно (або до виконання повторного запиту), надалі вся вибірка даних і позиціювання йде на клієнті. Однак модифікація даних відбувається негайно.

CursorType - встановлює тип курсору. Значення одне з:

ctUnspecified - бібліотека ADO сама визначає оптимальний тип блокування.

ctStatic - статичний курсор. Статична копія набору записів, що ви можете використати, наприклад, для генерації звіту. Додавання, зміни або видалення записів іншими користувачами не видимі.

ctOpenForwardOnly - ідентичний статичному курсору, за винятком того, що ви можете переходити тільки вперед. Це тип поліпшує ефективність у ситуаціях, коли ви робите тільки один прохід через набір даних.

ctDynamic - динамічний курсор. Додавання, зміни й видалення іншими користувачами видимі й можливі всі типи пересування по набору даних. Закладки (bookmarks) можливі тільки, якщо провайдер даних їх підтримує.

ctKeyset - курсор набору даних. Аналогічний динамічному курсору, за винятком того, що ви не побачите записи додані іншими користувачами, а записи вилучені іншими користувачами недоступні з вашого набору даних. Зміни даних іншими користувачами видимі.

Треба помітити, що TDBGrid й інші компоненти, що одночасно працюють із декількома записами, можуть працювати тільки коли закладки підтримуються. Тому для компонентів з якими ви будете зв'язувати такі компоненти повинні використовуватися типи ctKeyset або ctDynamic.

LockType - визначає тип блокування записів у наборі даних. Воно з:

ltUnspecified - бібліотека ADO сама визначає який тип буде використовуватися.

ltReadOnly - тільки читання, зміна даних неможливо.

ltPessimistic - песимістичне блокування. Запис блокується відразу після початку редагування й до збереження записів.

ltOptimistic - оптимістичне блокування. Запис блокується тільки коли зміни зберігаються.

ltBatchOptimistic - теж саме що й ltOptimistic, але використовується відкладене збереження змін записів. Більш докладно вона розглядається в наступному пункті.

MarshalOptions - це властивість визначає чи будуть відправлені на сервер ті поля, які не були змінені. При значенні moMarshalAll будуть, а при moMarshalModifiedOnly не будуть.

3.6.Приклад роботи з відкладеними змінами.

За основу візьмемо приклад роботи із транзакціями.

Додамо компоненти

BatchCB типу TCheckBox

ApplyButton типу TButton

CancelButton типу TButton

Додамо обробники подій OnClick в усі ці три компоненти.

Змінимо обробник події OnActivate форми.

```
procedure TForm1.FormActivate(Sender: TObject);
begin
   fix_controls;
   ApplyButton.Visible  := BatchCB.State = cbChecked;
   CancelButton.Visible := BatchCB.State = cbChecked;
end;
procedure TForm1.BatchCBClick(Sender: TObject);
begin
   MasterSQL.Close;
   DetailSQL.Close;
   if BatchCB.State = cbChecked then
     begin
       MasterSQL.LockType := ltBatchOptimistic;
       DetailSQL.LockType := ltBatchOptimistic;
     end
   else
     begin
       MasterSQL.LockType := ltOptimistic;
       DetailSQL.LockType := ltOptimistic;
```

```
    end;
  MasterSQL.Open;
  DetailSQL.Open;
  ApplyButton.Visible  := BatchCB.State = cbChecked;
  CancelButton.Visible := BatchCB.State = cbChecked;
end;
procedure TForm1.ApplyButtonClick(Sender: TObject);
begin
  MasterSQL.UpdateBatch;
  DetailSQL.UpdateBatch;
end;
procedure TForm1.CancelButtonClick(Sender: TObject);
begin
  MasterSQL.CancelBatch;
  MasterSQL.CancelBatch;
end;
```

Для підключення до сервера створюють файл.udl, що легко настроїти клацнувши на нього мишкою два рази.

Так підключаються до сервера:

```
ADOConnection1->ConnectionString = "FILE NAME=Server.udl";
try
{
ADOConnection1->Connected = true;
}
catch(...)
{
if (ADOConnection1->Connected == false){
MessageBox(0,"Не вдалося підключитися.",0,MB_OK);
}
}
```

Це функція для SQL запитів. Перший параметр - запит. Другий - прапор, чи повинен запит повернути дані. Повертається статус виконання, якщо все нормально, то повертається true інакше false. Викликати наприклад так SQLEx("select * from mytable",true); або так SQLEx("update mytable set column1=2 where column2=3",false);

```
bool __fastcall TForm1::SQLEx(AnsiString sqlreq,bool rs)
{
if (ADOConnection1->Connected)
{
ADOCommand1->CommandText = sqlreq;
ADOCommand1->Prepared = true;
_di__Recordset Recordset = ADOCommand1->Execute();
if (rs)ADODataSet1->Recordset = Recordset;
ADOCommand1->Prepared = false;
return true;
}
else
{
return false;
}
}
```

Так одержують дані від сервера:

```
ADODataSet1->DisableControls();
try
{
  for (ADODataSet1->First(); !ADODataSet1->Eof; ADODataSet1->Next())
{
ShowMessage(ADODataSet1->FieldByName["column1"]->AsString);
}
}
__finally
{
ADODataSet1->EnableControls();
}
```

Створювати БД для ADO найкраще в Microsoft Access. Для зв'язку із БД використовується компонент ADOConnection. Для формування ConnectionString є вбудований майстер підключення, що викликається при натисканні по доданої до поля кнопці "...".

1. Вибираємо движок(провайдер) БД - для Access використовується Microsoft Jet.

2. Вказуємо ім'я файлу із БД (для Access).

3. Вказуємо логін і пароль (за замовчуванням це "Admin" і "").

4. Натискаємо кнопку "Test Connection". Повинно бути "Ok".

5. При необхідності налаштовуємо правила спільного доступу.

Отриманий рядок можна оголосити константою, замінивши окремі значення символами підстановки, і підставляти в неї ім'я використовуваного файлу й інші вказуванні параметри, які:

```
const AnsiString cConnection_String = "Provider=Microsoft.Jet.OLEDB.4.0;
Data ...
Source=%s";
TForm1::TForm1 () //конструктор вікна
{
AnsiString DB_Name = ExtractPath (Application->ExeName) + "Database.mdb";
if (!FileExits (DB_Name)
throw Exception("Файл бази даних не знайдений");
ADOConn->ConnectionString = Format (cConnection_String, ARRAYOFCONST ((DB_Name));
ADOConn->Connected = true;
}
```

Дана програма при завантаженні підключає базу даних з каталогу програми, де б вона не перебувала.

4.Виконання роботи

1. Ознайомитись з методичними вказівками до лабораторної роботи
2. Виконати всі приведені в лабораторній роботі приклади
3. Створити власну БД засобами BDE, або MS Access
4. Виконати пункти 4.1-4.10 згідно свого варіанту
5. Скласти звіт

Лабораторна робота №4

1. Назва: ОРГАНІЗАЦІЯ РОБОТИ З БАЗАМИ ДАНИХ В МЕРЕЖІ клієнт/серверного типу за допомогою сервера InterBase

2.Мета роботи: Оволодіти технологією розробки розподілених баз даних на основі сервера InterBase в середовищі Delphi/C++Builder

3.Теоретичні положення

3.1Вступ

Для різних задач доцільно використовувати різні моделі баз даних, оскільки, звичайно, базу даних про співробітників якогось невеликого колективу і базу даних про який-небудь банк, що має філії у всіх кінцях країни, треба будувати по-різному. Процес визначення того, яка база даних більш підходить для конкретної програми, називається *масштабуванням*. Найбільші труднощі розробки виникають для інформаційних систем в економіці, де треба обслуговувати множину розподілених територіально користувачів. Один з шляхів вирішення цих проблем для баз даних з множиною користувачів лежить в площині використання архітектури клієнт/сервер. У цьому випадку доступ до бази даних для групи клієнтів виконується спеціальною програмою - сервером SQL, розташованим звичайно на окремому потужному комп'ютері. Клієнт дає завдання серверу виконати ті чи інші операції чи пошуку відновлення бази даних. Потужний сервер, орієнтований на операції з запитами оптимальним способом, виконує їхній запит і повідомляє клієнту результати своєї роботи.

Подібна організація роботи підвищує ефективність виконання прикладних програм за рахунок використання потужності сервера, розвантажує мережу, забезпечує надійний контроль цілісності даних. У базах даних клієнт/сервер виникає додаткова проблема - спроектувати прикладну програму так, щоб вона максимально використовувало можливості сервера і мінімально навантажувало мережу, передаючи через неї тільки мінімум інформації.

Подальший розвиток цього підходу приводить до багатоярусних баз даних. Реалізація платформи клієнт/сервер реалізується такими серверами баз даних, як InterBase, Oracle, Informix, Sybase SQL Server, DB2, MS SQL Server.

Іноді (зокрема, у документації Delphi) цей спосіб організації баз даних називається multi-tier - багатониткові. У цьому терміні під ниткою розуміється один з множини потоків даних, що обмінюються одночасно з базою даних.

Найбільш поширений триярусний варіант:

- На нижньому рівні на комп'ютерах користувача розташовані додатки клієнтів, що забезпечують користувальницький інтерфейс.

- На другому рівні розташований сервер прикладних програм, що забезпечує обмін даними між користувачами і розподіленими базами даних. Сервер прикладних програм розміщається у вузлі мережі, доступному всім клієнтам.
- На третьому рівні розташований віддалений сервер баз даних, що приймає інформацію від серверів прикладних програм і керуючий ними.

Подібну концепцію обробки даних пропагують, зокрема, фірми *Oracle* і *Sun*. Перший, елементарний рівень складається з «тонких клієнтів», тобто нескладних терміналів, призначених, в основному, для введення - виводу. Другий, середній (middleware) рівень - це робочі станції і сервери прикладних програм, тобто значно більш серйозні машини, на яких виконуються програми, критичні до завантаження процесора. Третій і останній рівень - потужні спеціалізовані сервери баз даних.

Це найбільш складна і гнучка організація баз даних. Delphi забезпечує в основному створення прикладних програм для перших двох рівнів цієї системи. При цьому треба відзначити, що на нижньому рівні - на комп'ютерах користувача не потрібно установки Borland Database Engine (BDE) - системи, що є в Delphi посередником між програмими і базами даних. У цьому полягає одне з переваг багатоярусних розподілених баз даних.

.

3.4 Створення демонстраційної бази даних

Бази даних створюються за допомогою програм, що входять до складу тієї чи інший СУБД. Наприклад, бази даних InterBase можна створювати за допомогою програми InterBase Interactive SQL, бази даних Microsoft Access створюються за допомогою відповідної програми, що входить до складу Microsoft Office, і т.д. Крім того до складу BDE, що поставляється разом з Delphi, входить програма Database Desktop, що дозволяє створювати бази даних ряду СУБД. Нарешті, третій шлях - створення баз даних безпосередньо з прикладних програм.

Демонстраційна база даних містить інформацію про якусь організацію. База містять чотири таблиці. Перша з них має ім'я Pers і містить список співробітників. Фрагмент її приведений у табл. 1. Таблиця має поля з іменами Num, Dep, Prizv, Nam, Par, Year_b, Sex, Charact, Photo, що містять відповідно номер запису, назве відділу, прізвище, ім'я, по батькові, рік народження, стать, характеристику співробітника і його фотографію.

Таблиця 1. Фрагмент таблиці даних про співробітників Pers

Номер	Відділ	Прізвище	Ім'я	По батькові	Рік народження	стать	Характеристика	Фотографія
Num	Dep	Prizv	Nam	Par	Year_b	Sex	Charact	Photo
1	Бухгалтерія	Іваненко	Віктор	Гаврилович	1950	ч	...	
2	Цех 1	Капустян	Петро	Петрович	1960	ч	...	
3	Цех 2	Сидоренко	Поліграф	Сидорович	1955	ч		
4	Цех 1	Іваненко	Ірина	Іванівна	1961	ж		...
	...	...	...		...		...	

У базі даних InterBase крім зазначених полів мається поле, що ще обчислюється, Age - вік. Його значення визначається вираженням 200 - **Year_b.** Друга таблиця названа Dep і має два поля: **Dep** - назву підрозділу, **Proisv** - тип підрозділу («у» - керування, «п» - виробництво). Вона приведена в табл 2.

Таблиця 2. Таблиця даних про підрозділи Dep

Відділ	Тип
Dep	Proisv
Бухгалтерія	У
Цех 1	П
Цех 2	П

Третя таблиця названа Storage і має два поля – Num – номер складу, Type – тип складу (таблиця 3.);

Таблиця 3. Таблиця даних про склади Storage

Код складу	Тип складу
Num	Type
1	Холодильний
2	Сортувальний
3	Сортувальний

четверта таблиця названа Invoice (таблиця накладних, по яких відбувається відвантаження товарів) і має наступні поля: Num, Date, Storage, Goods, Q, де, відповідно,

номер накладної, дата відвантаження, склад, з якого здійснювалося відвантаження, товар і його кількість – у кг. (таблиця 4.).

Таблиця 4. Таблиця даних про відвантажувальні накладні Invoice

Номер	Дата	Склад	Товар	Кількість
N	Date	Stora	Good	Q
1	12.12.09	1	Пиво	100
2	12.12.09	1	Риба	10
3	12.12.09	2	М'ясо	10

У базі даних Interbase поля **Sex і Proisv** цих таблиць символьні. У багатокористувацькому режимі, характерному для роботи в мережі, при створенні бази даних вказується ім'я користувача і його пароль. У нашій базі даних зазначений користувач «A» (латинська буква) і пароль «1». В даній лабораторній роботі рекомендується як пароль використовувати цифрові коди. Це звільняє користувача від необхідності стежити за тим, у якому регістрі і якою мовою (українською, чи англійською) вводиться пароль. Щоб одержати надалі доступ до створеного вами ж базі даних, треба зареєструвати себе як користувача. Це робиться за допомогою програми InterBase Server Manager. Після того, як база даних створена, бажано створити її *псевдонім.* Delphi має три інструменти, що дозволяють створювати і змінювати псевдоніми: Database Desktop, BDE Administrator і Database Explorer. Надалі будемо припускати, що псевдонім нашої демонстраційної бази InterBase - ib.

4.Виконання роботи

4.1 Створення, заповнення і реєстрація бази даних InterBase

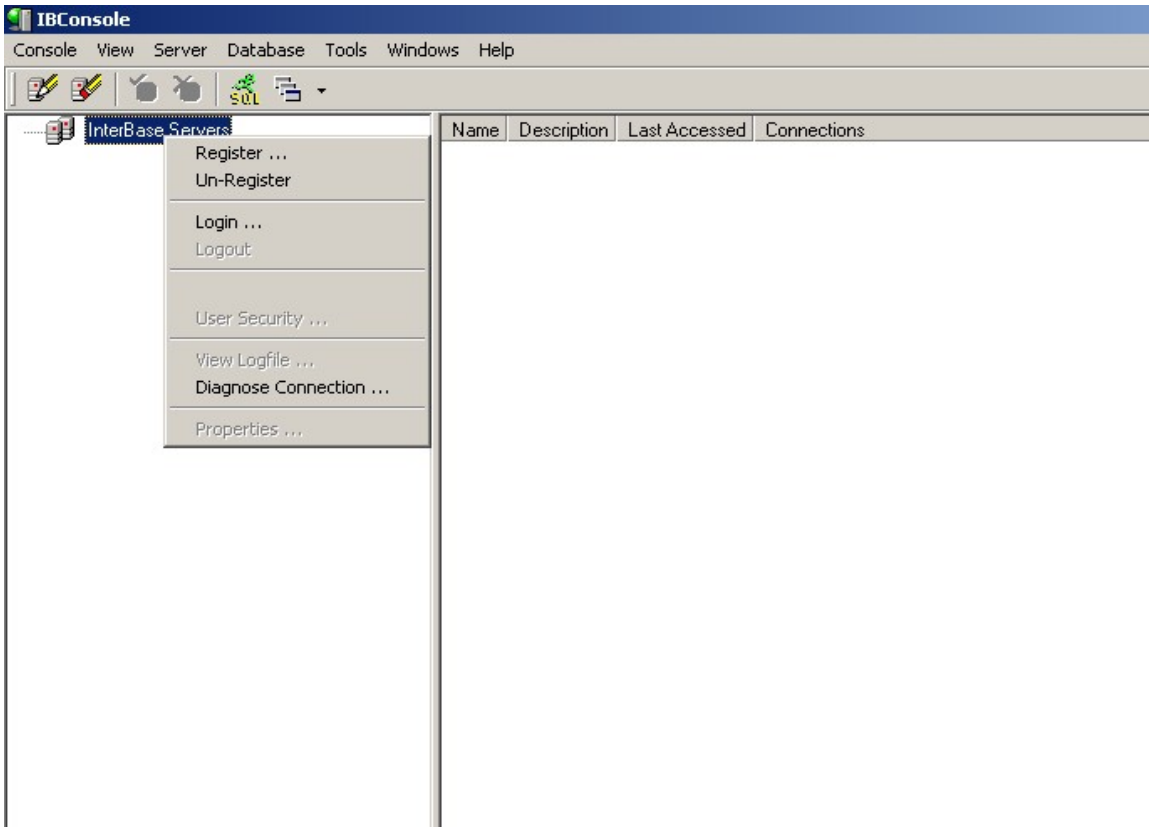
Щоб з'єднатися з базою даних, ви повинні виконати наступні дії у вказаному порядку.

1. Зареєструйте сервер.
2. Зареєструйтесь як клієнт сервера.
3. Зареєструйте базу даних.
4. Зареєструйтесь як клієнт бази даних.

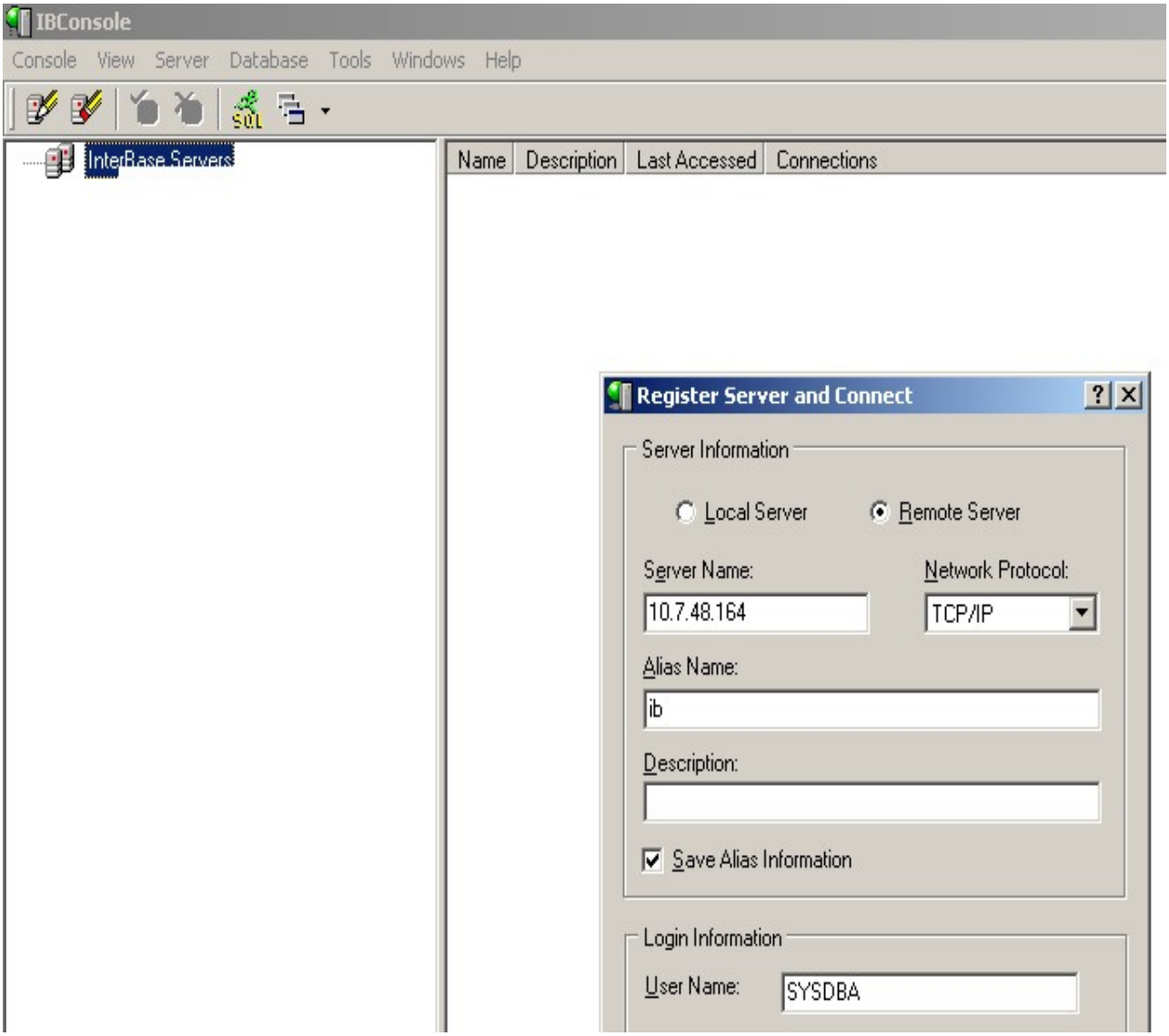
Одного разу реєструвавшись до бази даних, використовуйте таблицю ISQL, або будь-який з інших засобів IBConsole, щоб маніпулювати даними всередині або метаданими

безпосередньо. Нижче покроково реалізовано послідовність дій по створенню бази даних за технологією InterBase

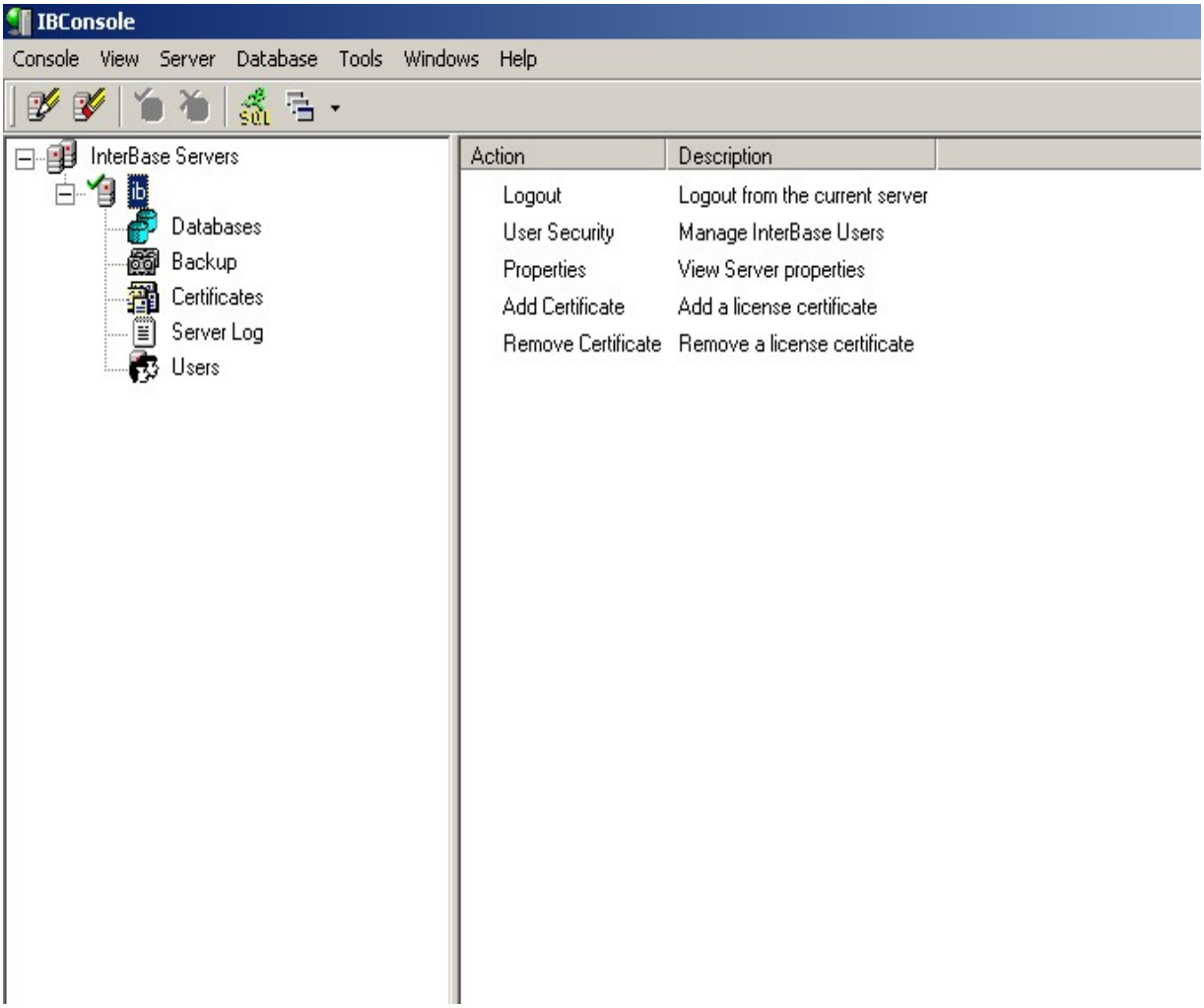
Крок 1)



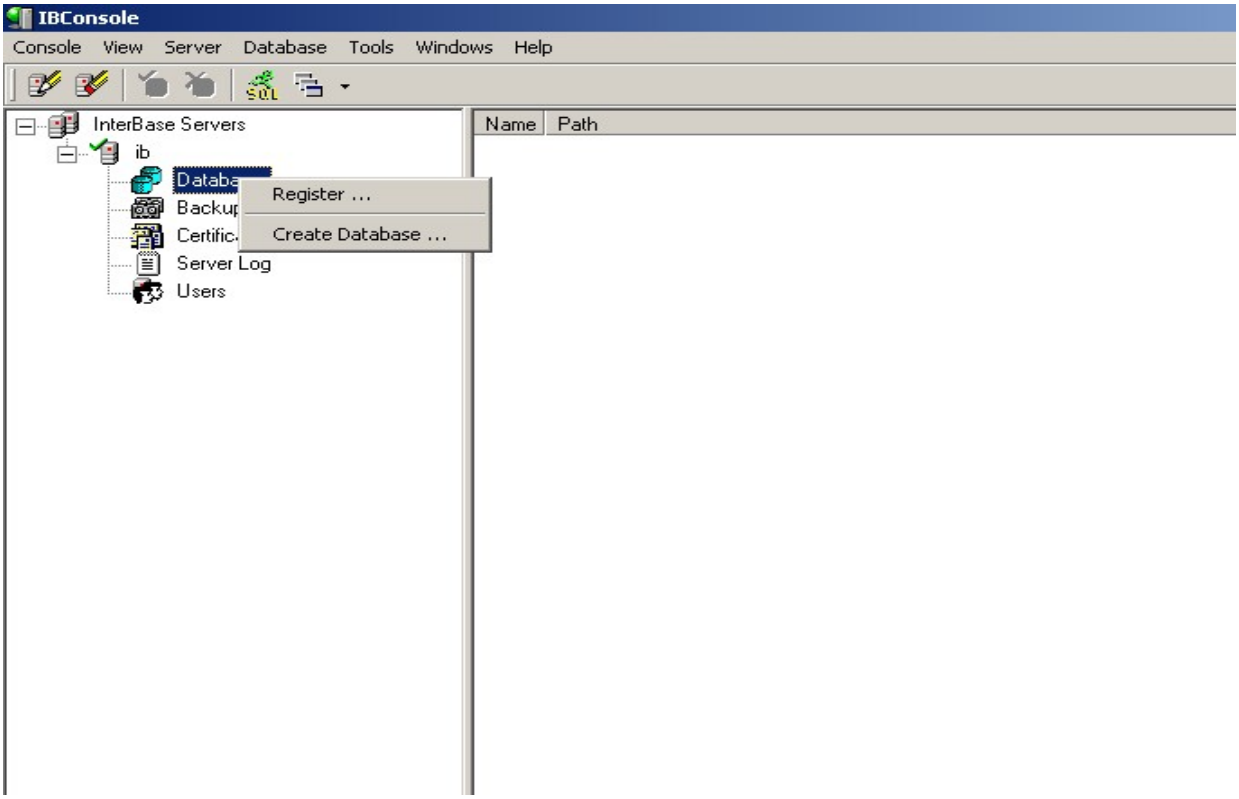
Крок 2)


IBConsole
Console View Server Database Tools Windows Help
InterBase Servers
Name Description Last Accessed Connections
Register Server and Connect
Server Information
Local Server
Remote Server
Server Name:
10.7.48.164
Network Protocol:
TCP/IP
Alias Name:
ib
Description:
Save Alias Information
Login Information
User Name:
SYSDBA

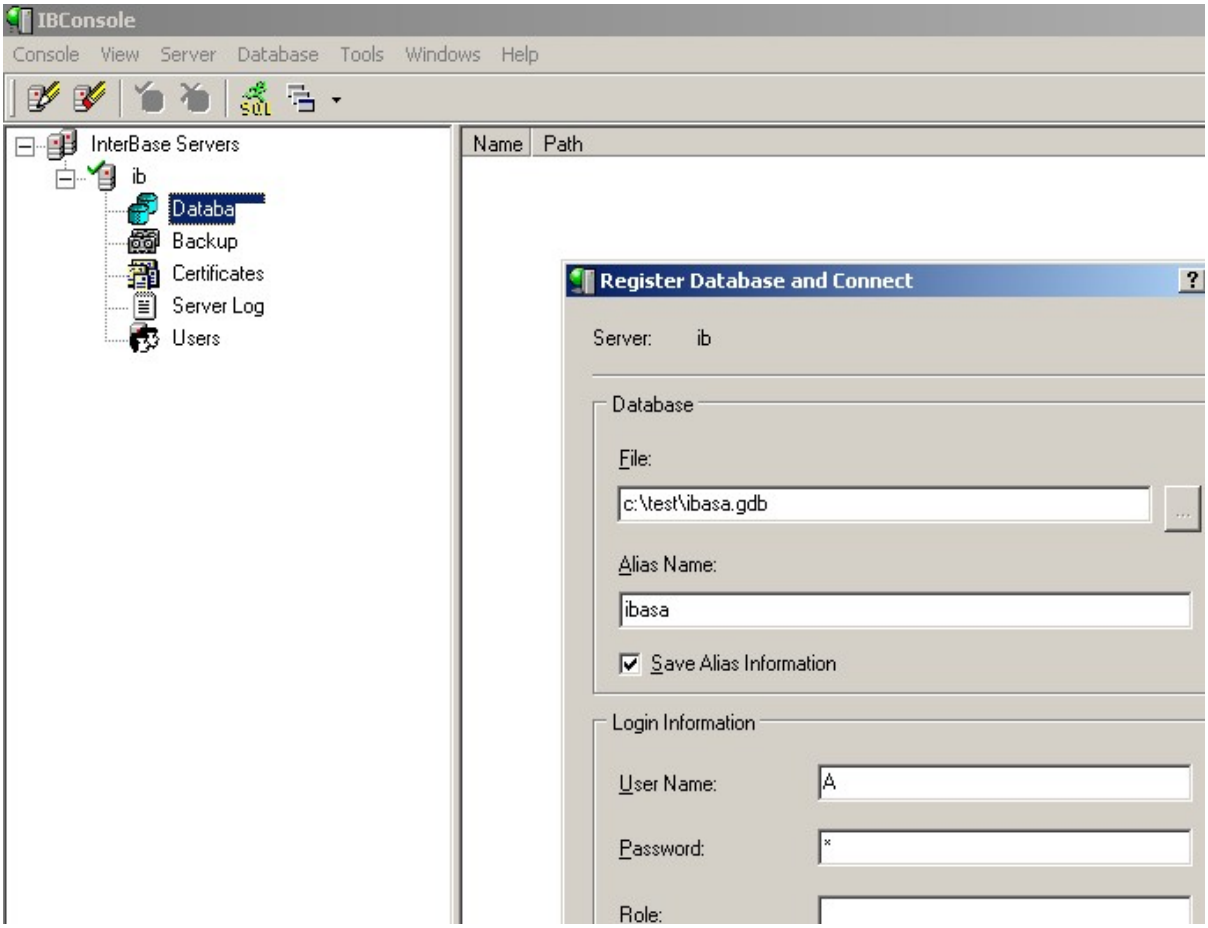
Крок 3)


IBConsole
Console View Server Database Tools Windows Help
InterBase Servers
Databases
Backup
Certificates
Server Log
Users
Action
Description
Logout
Logout from the current server
User Security
Manage InterBase Users
Properties
View Server properties
Add Certificate
Add a license certificate
Remove Certificate
Remove a license certificate

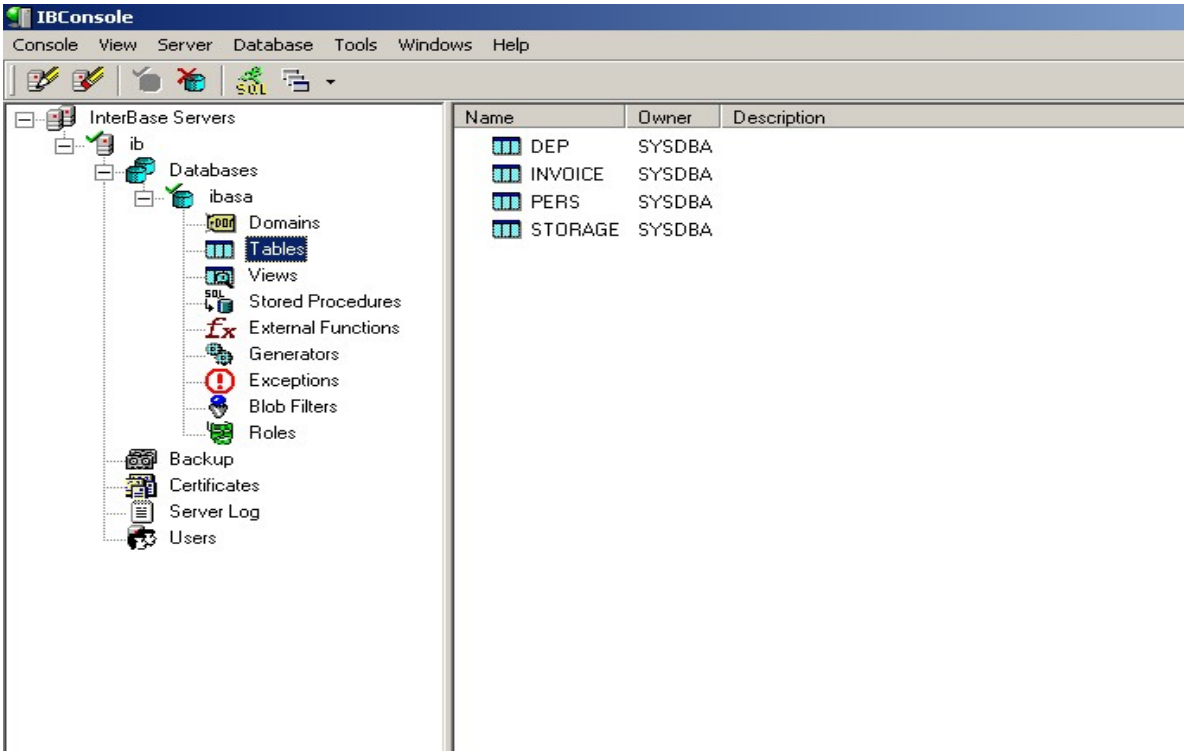
Крок 4)


IBConsole
Console View Server Database Tools Windows Help
InterBase Servers
ib
Databa
Backup
Certific
Server Log
Users
Register ...
Create Database ...
Name Path

Крок 5)


IBConsole
Console View Server Database Tools Windows Help
InterBase Servers
ib
Databa
Backup
Certificates
Server Log
Users
Name Path
Register Database and Connect
Server: ib
Database
File:
c:\test\ibasa.gdb
Alias Name:
ibasa
Save Alias Information
Login Information
User Name:
A
Password:
Role:

Крок 6)



4.2 Створення бази даних

База даних створюється простим скриптом.

CREATE DATABASE '...\Name.GDB'

USER 'A'

PASSWORD '1'

PAGE_SIZE = 8192

DEFAULT CHARACTER SET WIN1251;

CREATE DATABASE - це і є оператор, який створить базу даних. База даних являтиме собою файл, який буде створений в каталозі, вказаному після оператора. Розширення файлу може бути будь-яким, але прийнято, що GDB - розширення для файлу бази даних, а, наприклад, GBP - для резервної копії. *USER і PASSWORD* задають ім'я користувача і пароль. Цей користувач повинен бути зареєстрований на сервері до створення бази даних, інакше InterBase видасть повідомлення про помилку. *PAGE SIZE* задає розмір сторінки даних у файлі за умовчанням. Сторінка викачуватиметься з жорсткого диска тільки цілком. Тому, можна вважати, що це мінімальний розмір буфера роботи з файлом бази даних. Сторінка повинна бути такого розміру, щоб в неї помістився хоч би один запис в будь-якій з таблиць. Тут не потрібно враховувати розмір BLOB поля, оскільки для його зберігання виділяються

додаткові сторінки. Розміри сторінок можуть бути від 1024 до 8192 Kb. Розмір сторінок впливає на швидкодію і ступінь заповнення даними файлу бази даних. Так, якщо наступний запис не поміщається повністю в активну сторінку, то для неї буде виділена нова сторінка. Тому слід прагнути до кратного сторінці розміру запису. Це, звичайно досить проблематично, оскільки у Вас в БД може бути декілька таблиць з різними розмірами запису. Дуже великий розмір сторінки приводить до прочитування з диска записів, які можуть не знадобитися у вихідних даних запиту, що повинне знижувати швидкодію всієї системи в цілому. Очевидно, це відбувається при маленьких розмірах запису в порівнянні з розміром сторінки. Проте, численні досліди показують, що швидкодія може і знижується, але на таку маленьку величину, яку неможливо зафіксувати і зміряти в реальних грамотно побудованих застосуваннях.

DEFAULT CHARACTER SET визначає кодування символів в базі даних. Якщо Ви маєте намір використовувати кирилицю , то Вам слід встановити значення *WIN 1251*. Для інших мов є свої кодові таблиці. Зазвичай базу даних створюють в *IBConsole*. Там потрібно вибрати пункт меню *"Database/Create Database"*. У вікні, що з'явилося, заповнити поля введення параметрів операторів для створення БД. Ви можете створити БД з декількох файлів, які заповнюватимуться даними по черзі. Або створити дзеркало на іншому жорсткому диску для захисту від краху основного жорсткого диска. Далі треба починати роботу по створенню та використанню бази даних з клієнт серверною архітектурою. Пам’ятайте, що ми маємо справу з хрестоматійним *SQL*-сервером і тому використання конструкцій SQL є вирішальним фактором успішної роботи. Як приклад нижче наведено створення і робота з *БД InterBase.* Для того, щоб створити базу даних InterBase треба виконати наступні операції.

1. За допомогою будь-якого текстового редактора наберіть приведений нижче текст. Зокрема, ви можете скористатися для цього редактором Delphi, виконавши в Delphi команду File \ New і на сторінці New вікна Депозитарію обравши піктограму Text. При наборі тексту в перших двох операторах *Create database* і *CONNECT* треба замінити шлях до створюваного файлу бази даних ib.gdb на той каталог, у якому ви збираєтеся розмістити базу даних. База даних буде створена для користувача з ім'ям «A» і з паролем «1». Якщо ви хочете змінити це, то в тих же двох перших операторах замініть ім'я, що випливає після ключового слова USER, і пароль, що випливає після ключового слова PASSWORD.

2. Збережіть набраний текст у файлі. Якщо при наборі тексту ви користалися редактором Word, то файл треба зберігати як «тільки текст». Звичайно редактори автоматично дають текстовим файлам розширення .txt. Правда, для скрипт-файлів (а саме такий файл ви створили) прийняте інше розширення за замовчуванням - .sql. Можете змінити

розширення вашого файлу на .sql засобами Windows чи Norton Commander. А можете це і не робити - робота з файлом .txt ніяких особливих складностей не створить.

3. Викличте програму InterBase Interactive SQL. Це можна - зробити, натиснувши в Windows кнопку Пуск і в розділі Програми/InterBase обравши піктограму InterBase Interactive SQL. Програма реалізована файлом wisql32.exe, що звичайно розташований у каталозі ...\Program Files\InterBase Corp\InterBase\Bin.

4. Виконайте в InterBase Interactive SQL команду File\Run an ISQL Script. У стандартному вікні пошуку, що відкрилося, файлу виберіть створений вами файл. Тільки не забудьте, якщо розширення вашого файлу не .sql, змінити шаблон Тип файлу на All files (*.*). Інакше ви просто не побачите у вікні свого файлу, тому що за замовчуванням шаблон набудований на розширення .sql.

5. Перед вами з'явиться діалогове вікно, у якому вас запитують, чи треба зберігати вихідний файл попередньої роботи (а ви ще нічого не робили). Відповісти NO. Після цього почнеться робота зі створення бази даних, що може зайняти кілька секунд. На закінчення, якщо в підготовленому вами тексті немає принципових помилок, ви побачите діалогове вікно. Воно свідчить про те, що база даних успішно створена. У нижнім вікні InterBase Interactive SQL ви побачите текст файлу, за допомогою якого ви створювали базу даних. Посередині цього тексту після операторів Select * from Pers і Select * from Dep ви побачите уведений вами зміст таблиць Pers і Dep створеної бази даних. Переконайтеся в тім, що усе введено правильно. При цьому врахуйте, що при створенні бази даних заповнюються тільки текстові поля. Поля Charact - характеристика і Photo - фотографія повинні заповнюватися в процесі виконання прикладних програм, що працюють з цією базою даних .

Нижче приведений текст скрипт - файлу, що створює базу даних.

```
/* Створення бази даних */
Create database "d: \database\Ibase\ib.gdb"
USER "A" PASSWORD "1";
/* З'єднання з базою даних */
CONNECT "d:\database\Ibase\ib.gdb" USER "A"
PASSWORD "1";
/* Створення таблиці Dep */
create table Dep(Dep char (15) Not Null Primary Key,Proisv char (1));
/* Створення таблиці Storage */
create table Storage(Num smallint Not Null Primary Key, Type char (20));
/* Створення таблиці Invoice */
create table Invoice(Num smallint Not Null Primary Key,
Date char(15),
```

```
Storage smallint Not Null,
Goods char (20) Not Null,
Q smallint Not Null,
FOREIGN KEY (Num) REFERENCES Storage ON DELETE SET DEFAULT ON UPDATE
CASCADE, CONSTRAINT Storage CHECK(EXISTS(SELECT Num FROM Storage WHERE
Invoice.Storage = Storage.Num)));
/* Створення таблиці Pers */
create table Pers(Num smallint Not Null Primary Key,
Dep char(15) DEFAULT "Невідомий",
Prizv char (20) Not Null,
Nam char (20) Not Null,
Par char(20) Not Null,
Year_b smallint DEFAULT 1950
CHECK((Year_b>1917)and(Year_b<1980)),
Sex char(l) DEFAULT 'м',
Age COMPUTED BY (2000 - Year_b),
Charact blob sub_type 1,
Photo blob,
FOREIGN KEY (Dep) REFERENCES Dep ON DELETE SET DEFAULT ON UPDATE
CASCADE, CONSTRAINT Dep CHECK(EXISTS(SELECT Dep FROM Dep WHERE Pers.Dep =
Dep.Dep)));
/* Заповнення таблиці Dep */
Insert Into DEP( Dep,Proisv)
Values("Бухгалтерія", "у");
Insert Into DEP( Dep,Proisv)Values("Цех 1", "п");
Insert Into DEP( Dep,Proisv)Values("Цех 2", "п");
Insert Into DEP( Dep)Values("Невідомий");
Commit;
/* Заповнення таблиці Pers */
Insert Into PERS(Num, Dep, Prizv, Nam, Par, Year_b, Sex)
Values, "Бухгалтерія", "Іваненко", "Іван", "Іванович",1950, «ч») ;
Insert Into PERS(Num, Dep, Prizv, Nam, Par, Year_b, Sex)
Values(2, "Цех І", "Капустян", "Петро", "Петрович",1960, «ч»);
Insert Into PERS(Num, Dep, Prizv, Nam, Par, Уеаг_b, Sex)
Values(3, "Цех 2", "Сидоренко", "Сидір", "Миколайович",1955, «ч»);
Insert Into PERS(Num, Dep, Prizv, Nam, Par, Year_b, Sex)
```

Values(4, "Цех l", "Іваненко", "Ірина", "Іванівна",1961, "ж");

Insert Into PERS(Num, Dep, Prizv, Nam, Par, Year_b, Sex)

Values(5, "Бухгалтерія", "Миколайчук", "Микола", "Миколайович", 1930, «ч»);

Insert Into PERS(Num, Dep, Prizv, Nam, Par, Year_b, Sex)

Values(6, "Цех 2", "Андрєєв", "Андрій", "Андрійович",1930, «ч»);

Insert Into PERS(Num, Dep, Prizv, Nam, Par, Year_b, Sex)Values(7, "Цех l", "Борисенко", "Борис", Борисович",1957, «ч»);

Insert Into PERS(Num, Dep, Prizv, Nam, Par, Year_b, Sex)Values (8, "Цех l", "Павлов", "Павло", "Павлович",1975, «ч»);

Insert Into PERS(Num, Dep, Prizv, Nam, Par, Year_b, Sex)

Values (9, "Бухгалтерія", "Антонова", "Антоніна","Антонівна", 1955, "ж");

Insert Into PERS(Num, Dep, Prizv, Nam, Par, Year_b, Sex)Values(10, "Цех 2", "Харитонов", "Харитін","Харитонович", 1962, «ч»);

Insert Into PERS(Num. Dep, Prizv, Nam, Par, Year_b, Sex)Values(11, "Цех 2", "Іванников", "Іван", "Іванович",1985, «ч»);

/ Перегляд таблиць */*

*Select * from Pers;*

*Select * from Dep;*

/ Створення індексів */*

create index fio On pers Prizv,Nam,Par;

create index dfio On pers Dep,Prizv,Nam,Par;

create index year On pers Year_b;

/ створення view по підрозділах*/*

Create VIEW dep_l as

select Prizv, Nam, Par, Year_b, Sex from Pers

where Dep="Бухгалтерия";

Create VIEW dep_2 as

select Prizv, Nam, Par, Year b. Sex from Perswhere Dep="Цех l";

Create VIEW dep_3 as

select Prizv, Nam, Par, Year_b, Sex from Perswhere Dep="Цех 2";

/ Процедура Selectd - процедура вибору,повертає співробітників підрозділу */*

SET TERM ^;

CREATE PROCEDURE Selectd (pDep char (15))

RETURNS (pPrizv char (20), pNam char (20),

```
pPar char (20), pYear integer, PSex char(l))
AS
BEGIN
FOR SELECT Prizv, Nam, Par, Year_b, Sex From Pers
Where (Dep=:pDep)
Into pPrizv, pNam, pPar, pYear, pSex
DO
SUSPEND;END;^SET TERM ;^COMMIT;
/* Процедура Setinf – виконувана процедура, змінює дані про співробітника, повертає номер запису чи 0, якщо співробітника немає */
SET TERM ^;
CREATE PROCEDURE Setinf
(pPrizv char(20), pNam char(20), pPar char (20),  pYear integer, pDep char(15),pSex char(l))
RETURN  S
(mess integer)

AS
BEGIN
mess=0;
Select Num From Pers
Where (Prizv=:pPrizv)and(Nam«;pNam)andtPar^ipPar)
Into mess;
if(mess>0)
then
Update Pers Set Year_b=:pYear, Dep=:pDep, Sex=:pSex
Where (Prizv=:pPain)and(Nam=:pNam)and(Par=:pPar) ;
END;^
SET TERM ^;
COMMIT;
/* Процедура Getinf - процедура виконувана, повертає дані про співробітника,
чи рік народження = 0, якщо співробітника немає */

SET TERM ^;
CREATE PROCEDURE Getinf
(pPrizv char (20), pNam char(20), pPar char (20))
```

```
RETURNS
(pYear integer, pDepchar<15),pSex char(l))
AS
BEGIN

pYear=0;
Select Year_b, Dep, Sex From Pers
Where (Prizv=:pPrizv)and(Nam=:pNam)and(Par=:pPar)into pYear, pDep, pSex;

END;^
SET TERM ;^
COMMIT;
```

Якщо ви створили базу даних, задавши ім'я користувача (у приведеному тексті ім'я користувача - «А», пароль - «1»), але це ім'я ще не внесене в список користувачів, то насамперед вам треба зареєструвати себе як користувача. Це можна зробити за допомогою програми InterBase Server Manager. Нижче приведені основні кроки, які потрібно виконати для реєстрації себе як користувача.

1. Викличте програму InterBase Server Manager. Це можна зробити, натиснувши в Windows кнопку Пуск і в розділі Програми \ InterBase обравши піктограму Server Manager. Програма реалізована файлом ibmgr32.exe, що звичайно розташований у каталозі ...\Program Files\InterBase Corp\InterBase\Bin.

2. Виконайте команду File \ Server Login. У вікні, що відкрилося, введіть ім'я користувача (User Name) SYSDBA і пароль (Password) masterkey. Це ім'я і пароль адміністратора, прийняті при установці InterBase. Якщо на вашому комп'ютері этоv було змінено, звернетеся до адміністратора. Пароль masterkey повинний набиратися в нижньому регістрі.

3. Виконайте команду Tasks \ User Security. У вікні інформації, що відкрилося, про користувачів натисніть кнопку Add User - додати користувача. У діалоговому вікні, що з'явилося, введіть у віконце User Name ім'я користувача, у віконце Password уведіть пароль і повторите його у віконці Confirm Password. Натисніть ОК. Після цього ви будете зареєстровані як користувач і можете залишити Server Manager. Для використання вашої бази даних зручно створити її псевдонім. Це може здійснюватися різними шляхами. Зокрема, один зі шляхів наступний:

1.Викличте програму Database Desktop. Це можна зробити, натиснувши в Windows кнопку Пуск і в розділі Програми \ Borland Delphi обравши піктограму Database Desktop.

Програма реалізована файлом dbd32.exe, що звичайно розташований у каталозі ... \Program Files\Borland\Database Desktop.

2. Виконайте команду Tools \ Alias Manager. У діалоговому вікні, що з'явилося, натисніть кнопку New.

3. У віконце Database Alias занесіть псевдонім бази даних (у прикладі використовується псевдонім ib). Включіть індикатор Public Alias. Виберіть у списку, що випадає, Driver Type тип драйвера IntrBase. У віконце Sever Name занесіть ім'я файлу створеної вами бази даних з повним шляхом до нього. У віконце User Name занесіть ім'я користувача.

4. Натисніть кнопку Keep New. Натисніть кнопку ОК. Ви побачите вікно, у якому вам пропонується питання, чи зберегти інформацію про псевдоніми у файлі конфігурації ...\IDAPI.CFG. Відповісти на це питання позитивно. Тепер можете залишити Database Desktop. Створений вами псевдонім зареєстрований у системі.

5.Завдання на виконання

5.1Розбийтесь на бригади по два.

5.2 Зареєструйтесь як користувач на сервері.

5.3 Розробіть базу даних на сервері згідно варіанта та клієнтську прикладну програму.

Варіанти завдань на бази(перелічені сутності):

1. Факультет, кафедра, викладач; розклад;
2. Чемпіонат, клуб, стадіон, результат;
3. Поліклініка, лікар, хвороба, хворий;
4. Залізниця, вокзал, поїзд, пасажир;
5. Універмаг, Покупець, Товар, Постачальник;
6. Туристична фірма, курорт, сезон;
7. Концертна фірма, Концертний зал, гастролер, глядач.
8. Виробнича фірма, Цех, Склад, Магазин
9. Аеропорт, літак, Рейс, пасажир
10. БД адвоката,
11. БД по нерухомості
12. БД "Готель"
13. БД "Спортивный клуб".
14. БД "Відділ кадрів"
15. БД "Ремонтна майстерня"

Лабораторна робота №5

1.Тема: Технологія DataSnap. Механізми віддаленого доступу.

2.Мета: Освоєння процесу розробки трьохшарової клієнт-серверної корпоративної інформаційної системи у середовищі Delphi/C++Builder на основі технології DataSnap.

3.Методичні вказівки

Далі буде розглянута модель розподіленої багатоланкової (multitiered) прикладної програми БД і, зокрема, його найбільш простий варіант - трьохшарова розподілена прикладна програма. Трьома частинами такої прикладної програми є:

- власне сервер БД;
- сервер прикладних програм (серверна частина прикладної програми);
- клієнтська частина прикладної програми.

Всі вони об'єднані в єдине ціле єдиним механізмом взаємодії (транспортний рівень) і обробки даних (рівень бізнес-логіки).

Компоненти та об'єкти Delphi/C++Builder, що забезпечують розробку багатоланкових прикладних програм, об'єднані загальною назвою DataSnap.

Палітра компонентів Delphi/C++Builder містить спеціальну сторінку **DataSnap,** на якій доступні більшість розглянутих далі компонентів.

Структура багатоланкової прикладної програми в Delphi/C++Builder

Багатоланкова архітектура прикладних програм БД викликана необхідністю обробляти на стороні сервера запити від великої кількості віддалених клієнтів. Здавалося б, із цим завданням цілком можуть впоратися й звичайні локальні прикладні програми типу клієнт-сервер.

Однак в цьому випадку при великій кількості клієнтів все обчислювальне навантаження лягає на сервер БД, що має досить незначний набір засобів для реалізації складної бізнес-логіки (збережені процедури, тригери й т.д.). І розробники змушені істотно ускладнювати програмний код клієнтського ПЗ (програмного забезпечення), а це вкрай небажано при наявності великої кількості віддалених клієнтських комп'ютерів. Адже з ускладненням клієнтського ПЗ зростає ймовірність помилок й ускладнюється його обслуговування. Багатоланкова архітектура прикладних програм БД покликана виправити перераховані недоліки. Отже, у рамках цієї архітектури "тонкі" клієнти являють собою найпростіші прикладні програми, що забезпечують лише передачу даних, їх локальне кешування, представлення засобами інтерфейсу користувача, редагування й найпростішу обробку.

Клієнтські прикладні програми звертаються не до сервера БД прямо, а до спеціалізованого ПЗ проміжного шару. Це може бути й одна ланку (найпростіша трьохшарова модель) і більш складна структура. ПЗ проміжного шару називається сервером прикладних програм, приймає запити клієнтів, обробляє їх відповідно до запрограмованих правил бізнес-логіки, при необхідності перетворює у форму, зручну для сервера БД і відправляє серверу. Сервер БД виконує отримані запити й відправляє результати серверу прикладних програм, що адресує дані клієнтам.

Трьохланкові прикладні програми в Delphi/C++Builder

Тепер розглянемо складові частини трьохланкового розподіленої прикладної програми в Delphi/C++Builder (рис. 1). Як згадувалося вище, в Delphi/C++Builder доцільно розробляти клієнтську частину трьохланкового прикладної програми та ПЗ проміжного шару - сервер прикладних програм.

Рис. 1. Схема трьохланкової розподіленої прикладної програми

Частини трьохланковоих прикладних програм розробляються з використанням компонентів DataSnap, а також деяких інших спеціалізованих компонентів, які в основному забезпечують функціонування клієнта. Для доступу до даних застосовується одна із чотирьох технологій, реалізованих в Delphi/C++Builder .

Для передачі даних між сервером прикладних програм і клієнтами використається інтерфейс **AppServer**, надаваний віддаленим модулем даних сервера прикладних програм. Цей інтерфейс використають компоненти-провайдери **TDataSetProvider** на стороні сервера й компоненти **TClientDataSet** на стороні клієнта.

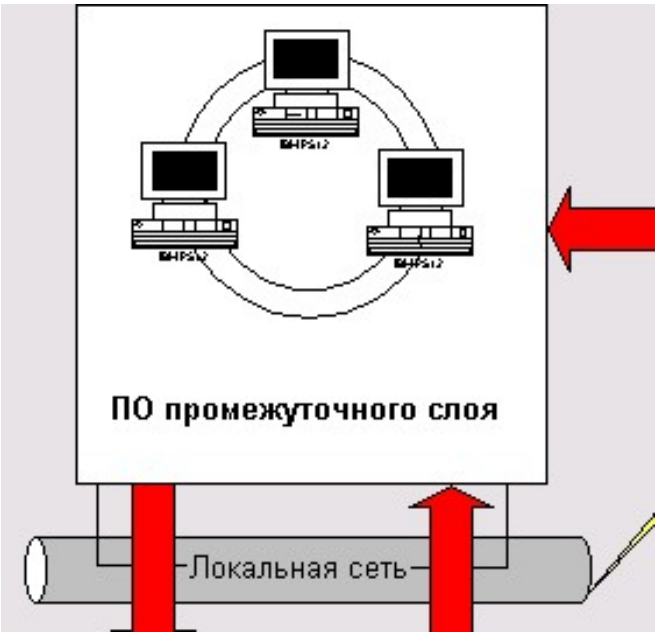


Рис. 2. Багатоланкова архітектура прикладних програм БД

Таким чином, багатоланкова прикладна програма БД складається з (рис. 2):

- "тонких" клієнтських прикладних програм, що забезпечують лише передачу, представлення, редагування й найпростішу обробку даних;
- одного або декількох ланок ПЗ проміжного шару (сервер прикладних програм), які можуть функціонувати як на одному комп'ютері, так і розподілено - у локальній мережі;
- сервера БД (Oralce, Sybase, MS SQL, InterBase і т.д.), що підтримує функціонування бази даних й обробку запитів.

Більш проста трьохланкова модель містить наступні елементи:

- "тонкі" клієнти;
- сервер прикладних програм;
- сервер БД.

Далі ми будемо розглядати саме трьохланкову модель. У середовищі розробки Delphi/C++Builder є набір інструментів і компонентів для створення клієнтського ПЗ й ПЗ проміжного шару. Сервер прикладних програм взаємодіє із сервером БД, використовуючи одну з технологій доступу до даних, реалізованих в Delphi/C++Builder. Це технології ADO, BDE, InterBase Express й dbExpress. Розробник може вибрати найбільш зручну, виходячи з поставленого завдання й параметрів сервера БД.

Віддалені клієнтські прикладні програми створюються з використанням спеціального набору компонентів, об'єднаних загальною назвою DataSnap. Ці компоненти інкапсулюють стандартні транспорти (DCOM, HTTP, сокети) і забезпечують з'єднання віддаленого клієнтської прикладної програми із сервером прикладної програми. Також компоненти

DataSnap забезпечують доступ клієнта до функцій сервера прикладних програм за рахунок використання інтерфейсу AppServer.

Важливу роль при розробці клієнтських прикладних програм грає компонент, що інкапсулює клієнтський набір даних. Його реалізації також залежать від технологій і доступу до даних.

Поряд з перерахованими вище перевагами, наявність додаткової ланки - сервера прикладних програм - дає деякі додаткові бонуси, які можуть бути досить істотною підмогою з погляду підвищення надійності й ефективності системи.

Так як найчастіше клієнтські комп'ютери - це досить слабкі машини, реалізація складної бізнес-логіки на стороні сервера дозволяє істотно підвищити швидкодію системи в цілому. І не тільки за рахунок могутнішої техніки, але й за рахунок оптимізації виконання однорідних запитів користувачів.

Наприклад, при надмірному завантаженні сервера, сервер прикладних програм може самостійно обробляти запити користувачів (ставити їх у чергу або скасовувати) без додаткового завантаження сервера БД.

Наявність сервера прикладних програм підвищує безпеку системи, тому що ви можете організувати тут авторизацію користувачів, та будь-які інші функції безпеки без прямого доступу до даних.

Крім того, ви легко зможете використати захищені канали передачі даних, наприклад HTTPS.

Сервер прикладних програм

Сервер прикладних програм інкапсулює більшу частину бізнес-логіки розподіленого прикладної програми й забезпечує доступ клієнтів до бази даних.

Основною частиною сервера прикладних програм є віддалений модуль даних.

По-перше, подібно звичайному модулю даних він є платформою для розміщення невізуальних компонентів доступу до даних і компонентів-провайдерів. Розміщені на ньому компоненти з'єднань, транзакцій і компоненти, що інкапсулюють набори даних, забезпечують трьохланкова прикладна програма зв'язком із сервером БД. Це можуть бути набори компонентів для технологій ADO, BDE, InterBase Express, dbExpress.

По-друге, відділений модуль даних реалізує основні функції сервера прикладних програм на основі надання клієнтам інтерфейсу IAppServer або його нащадка. Для цього віддалений модуль даних повинен містити необхідне число компонентів-провайдерів TDataSetProvider. Ці компоненти передають пакети даних клієнтській прикладної програми, а точніше компонентам TdientDataSet, а також забезпечують доступ до методів інтерфейсу.

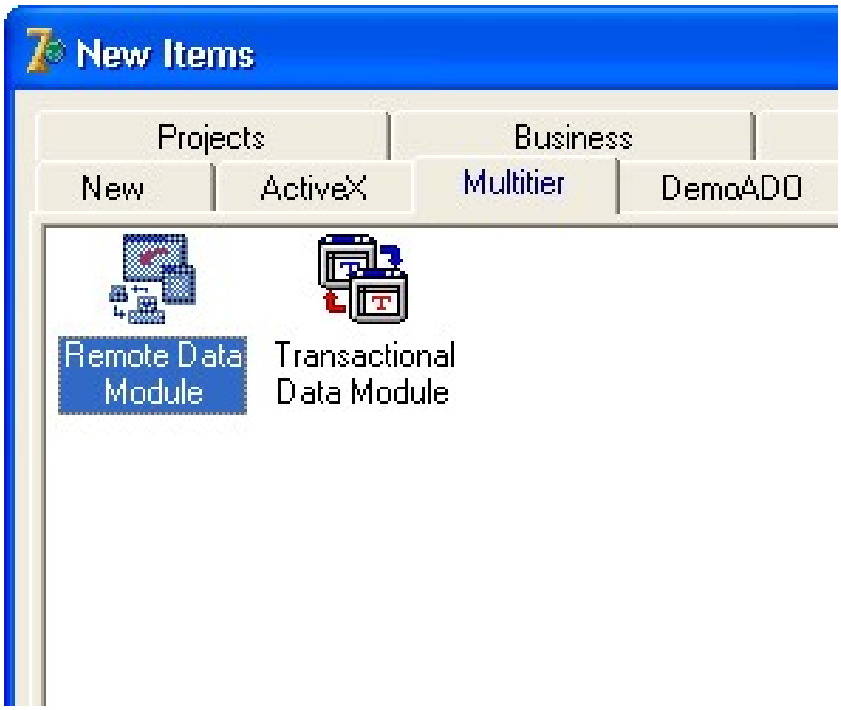


Рис. 3. Вибір віддалених модулів даних

До складу Delphi/C++Builder входять віддалені модулі даних. Для їхнього створення використайте сторінки **Multitier, WebSnap й WebServices (меню File | New | Other)** (рис. 3).

- **Remote Data Module** - віддалений модуль даних, що інкапсулює сервер автоматизації. Використається для організації з'єднань через DCOM, HTTP, сокети.
- **Transactional Data Module** - віддалений модуль даних, який інкапсулює сервер MTS (Microsoft Transaction Server).
- **SOAP Server Data Module** - віддалений модуль даних, який інкапсулює сервер SOAP (Simple Object Access Protocol).
- **WebSnap Data Module** - віддалений модуль даних, що використовує Web-служби й Web-браузер як сервер.

Окрім віддаленого модуля даних невід'ємною частиною серверу прикладних програм є компоненти-провайдери TDataSetProvider. З кожним компонентом, що інкапсулює набір даних, призначеним для передачі клієнтові, у модулі даних повинен бути зв'язаний компонент-провайдер.

Клієнтська прикладна програма

Клієнтська прикладна програма у трьохланковій моделі повинен володіти лише мінімально необхідним набором функцій, делегуючи більшість операцій по обробці даних серверу прикладних програм.

Перш за все віддалений клієнтська прикладна програма повинен забезпечити з'єднання із сервером прикладних програм. Для цього використовуються компоненти з'єднань DataSnap:

- TDCOMConnection - використовує DCOM;
- TSocketconnection - використовує сокети Windows;
- TWebConnection - використовує HTTP.

Компоненти з'єднання DataSnap надають інтерфейс IAppServer, що використовується компонентами-провайдерами на стороні сервера й компонентами TClientDataSet на стороні клієнта для передачі пакетів даних.

Для роботи з наборами даних використовуються компоненти TClientDataSet, що працюють у режимі кешування даних.

Для представлення даних і створення інтерфейсу користувача в клієнтському ПЗ застосовуються стандартні компоненти зі сторінки **Data Controls** палітри компонентів.

Механізм віддаленого доступу до даних DataSnap

Для передачі пакетів даних між компонентом-провайдером і клієнтським набором даних (див. рис. 2) (між клієнтом і сервером) повинен існувати якийсь транспортний канал, що забезпечує фізичну передачу даних. Для цього можуть використовуватися різноманітні транспортні протоколи, підтримувані операційною системою.

Різні типи з'єднань, що дозволяють настроїти транспорт і почати передачу й прийом даних, інкапсульовані в декількох компонентах DataSnap. Для створення з'єднання з тим або іншим транспортним протоколом розробнику досить перенести відповідний компонент на форму й правильно настроїти кілька властивостей. Нижче розглядаються варіанти настройки транспортних протоколів для компонентів, що використовують DCOM, сокети TCP/IP, HTTP.

Компонент TDCOMConnection

Компонент TDCOMConnection надає транспорт на основі технології Distributed COM і застосовується в основному для організації транспорту в рамках локальної мережі.

Для настройки з'єднання DCOM у першу чергу необхідно задати ім'я комп'ютера, на якому функціонує сервер прикладних програм. Для компонента TDCOMConnection це повинен бути зареєстрований сервер автоматизації. Ім'я комп'ютера задається властивістю **ComputerName (string).** Якщо воно задано правильно, у списку властивості **ServerName (string)** в Інспекторі об'єктів можна вибрати один з доступних серверів. При виборі сервера також автоматично заповнюється властивість **ServerGUID (string).** Причому для успішного з'єднання клієнта із сервером прикладних програм обидві властивості повинні бути задані в обов'язковому порядку. Тільки ім'я сервера або тільки його GUID не забезпечать правильний доступ до віддаленого об'єкта COM.

Відкриття й закриття з'єднання здійснюється властивістю **Connected (Boolean)** або методами **Open/Close** відповідно.

Для організації передачі даних між клієнтом і сервером компонент TDCOMConnection надає інтерфейс IAppServer **property AppServer (Variant),** який також може бути отриманий методом **function GetServer (lAppServer; override)**

Властивість **ObjectBroker (TCustomObjectBroker)** дозволяє використати екземпляр компонента TsimpleObjectBroker для одержання списку доступних серверів під час виконання (див. нижче).

Методи-обробники компонента TDCOMConnection представлені в табл. 1.

Таблиця 1. Методи-обробники подій компонента TDCOMConnection

Оголошення	**Опис**
property AfterConnect: TNotifyEvent;	Викликається після встановлення з'єднання
property AfterDisconnect: TNotifyEvent;	Викликається після розриву з'єднання
property BeforeConnect: TNotifyEvent;	Викликається перед установленням з'єднання
property BeforeDisconnect: TNotifyEvent;	Викликається перед розривом з'єднання
type TGetUsernameEvent = procedure (Sender : TOb j ect ; var Username: string) of object; property OnGetUsername : TGetUsernameEvent ;	Викликається безпосередньо перед появою діалогу віддаленої авторизації користувача. Для цього властивість LoginPrompt повинне мати значення True. Параметр Username може містити ім'я користувача за замовчуванням, що з'явиться в діалозі
type TLoginEvent = procedure (Sender: TOb j ect; Username, Password: string) of object; property OnLogin: TLoginEvent;	Викликається після відкриття з'єднання, якщо властивість LoginPrompt має значення True. Параметри Username й Password містять ім'я користувача й пароль, уведені при авторизації

Компонент TSocketConnection

Компонент TSocketConnection забезпечує з'єднання клієнта із сервером прикладних програм за рахунок використання сокетів TCP/IP. Для успішного відкриття з'єднання на стороні сервера повинен працювати сокет-сервер (рис. 4).

Для успішного з'єднання властивість **property Host (String)** повинна містити ім'я комп'ютера сервера.

Допоміжні компоненти - брокери з'єднань.

До складу компонентів **DataSnap** входить ряд додаткових компонентів, що полегшують роботу із з'єднаннями віддалених клієнтів із сервером прикладних програм. Розглянемо їх.

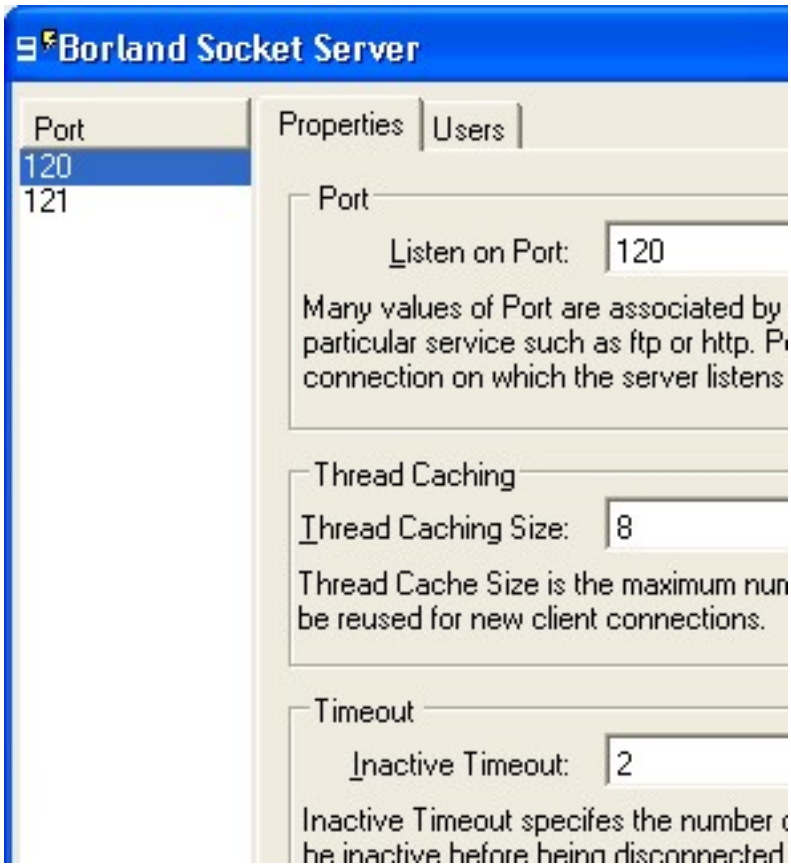


Рис. 4. Сокет-сервер ScktSrvr.exe

Компонент TSimpleObjectBroker

Компонент TSimpleObjectBroker інкапсулює список серверів, доступних для клієнтів даного багатоланкового розподіленої прикладної програми. Список серверів створюється на етапі розробки. При необхідності (відключення сервера, його перевантаження й т.д.) компонент з'єднання клієнтського ПЗ може використати один із запасних серверів зі списку компонента TsimpleobjectBroker безпосередньо під час виконання.

Для цього необхідно заповнити список серверів компонента TSimpleobjectBroker і вказати посилання на нього у властивості objectBroker компонента з'єднання (див. вище). І тоді при "перевідкритті" з'єднання ім'я сервера буде запитуватися зі списку компонента **TsimpleobjectBroker.**

Список серверів задається властивістю **Servers (TServerCollection).** На етапі розробки список серверів заповнюється спеціалізованим редактором (рис. 5), що викликається при клацанні на кнопці властивості в Інспекторі об'єктів.

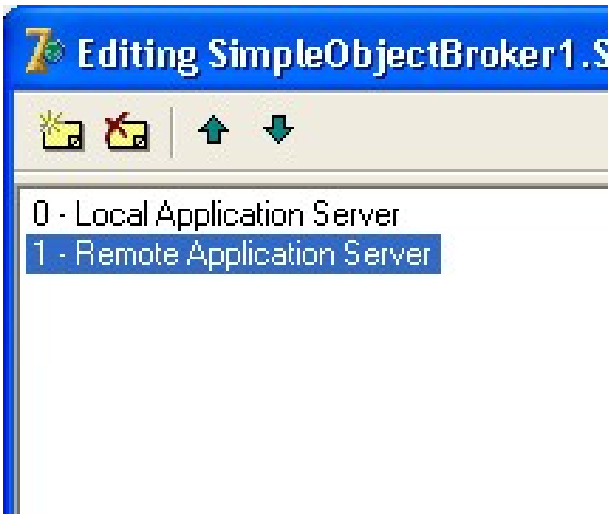


Рис. 5. Редактор списку серверів компонента TSimpleObjectBroker

Властивість **Servers** являє собою колекцію об'єктів класу TServeritem. Цей клас має кілька властивостей, що дозволяють описати основні параметри сервера (табл. 2). При використанні в з'єднанні значення цих властивостей підставляються у відповідні властивості компонента з'єднання.

Таблиця 2. Властивості класу TServeritem

Оголошення	**Опис**
property ComputerName: string;	Ім'я комп'ютера, на якому функціонує сервер
property DisplayName: String;	Містить ім'я сервера для подання в списку серверів
property Enabled: Boolean;	Управляє доступністю запису про сервер для вибору при підключенні. При значенні True компоненти з'єднань можуть використати даний запис списку для підключення
property HasFailed: Boolean;	Після невдалої спроби використати даний запис списку при підключенні властивості привласнюється значення True і надалі ця запис не використається
property Port: Integer;	Містить номер порту, використовуваного при підключенні до сервера

Крім списку серверів компонент має лише кілька допоміжних властивостей і методів.

Метод **function GetComputerForGUID(GUID: TGUID): string; override** повертає ім'я комп'ютера, на якому зареєстрований сервер з GUID, заданим параметром.

Метод **function GetComputerForProgID(const ProgID): string; override** повертає ім'я комп'ютера, на якому зареєстрований сервер з ім'ям, заданим параметром Progio.

Властивість **LoadBalanced (Boolean)** управляє вибором сервера зі списку. При значенні True запис про сервер вибирається випадковим чином, інакше для з'єднання пропонується перший доступний запис про сервер.

Додатково, властивість **Address (String)** повинна містити IP-адреси сервера. Для відкриття з'єднання повинні бути задані обидві ці властивості.

Властивість **Port (Integer)** встановлює номер використовуваного порту. За замовчуванням це порт 211, але розробник вільний змінити порт, наприклад, для використання різними категоріями користувачів або для створення захищеного каналу.

Після правильного вибору комп'ютера в списку властивості **ServerName (string)** в Інспекторі об'єктів з'являється перелік доступних серверів автоматизації. І після вибору сервера властивість **ServerGUID (string),** яка містить ім'я комп'ютера GUID зареєстрованого сервера, задається автоматично, хоча його можна задати й вручну.

Метод **function GetServerList (OleVariant; virtual)** повертає список зареєстрованих серверів автоматизації. Відкриття й закриття з'єднання здійснюється властивістю **Connected (Boolean)** або методами **procedure Open** чи **procedure Close** відповідно.

Канал сокету TCP/IP може бути зашифрований. Для цього використовується властивість **InterceptName (string)**, яка утримує програмний ідентифікатор об'єкта COM, що забезпечує шифрування/дешифрування даних у каналі, і властивість **InterceptGUID (string)**, яка утримує ім'я комп'ютера GUID цього об'єкту. Цей об'єкт COM перехоплює дані в каналі й здійснює їхню обробку, передбачену власним програмним кодом. Це може бути шифрування, стиск, обробка шумів і т.д.

Природно, на стороні сервера має бути зареєстрований об'єкт COM, що виконує зворотну операцію. Для цього також використовується сокет-сервер (рис. 6). Рядок **Interceptor** на сторінці повинен містити ім'я комп'ютера GUID об'єкта-перехоплювача COM.

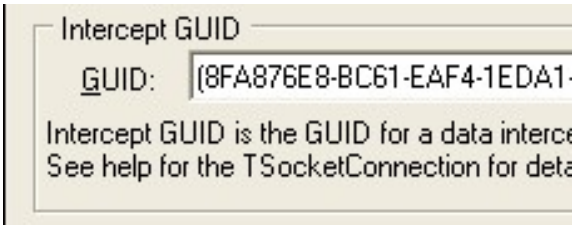


Рис. 6. Реєстрація об'єкта-перехоплювача COM у сокет-сервері

Метод **function GetInterceptorList (OleVariant; virtual)** повертає список зареєстрованих на сервері об'єктів-перехоплювачів.

Для організації передачі даних між клієнтом і сервером компонент TSocketConnection надає інтерфейс **IAppServer property AppServer (Variant),** який також може бути отриманий методом **function GetServer (IAppServer; override).**

Властивість **ObjectBroker (TCustomObjectBroker)** дозволяє використовувати екземпляр компонента **TSimpieObjectBroker** для одержання списку доступних серверів під час виконання (див. нижче).

Методи-обробники подій компонента **TSocketConnection** повністю збігаються з методами-обробниками подій компонента **TDCOMConnection** (див. табл. 1).

Компонент TWebConnection

Компонент TWebConnection надає клієнтові з'єднання на основі транспорту HTTP. Для роботи компонента на клієнтському комп'ютері повинна бути зареєстрована бібліотека wininet.dll. Звичайно це не вимагає спеціальних зусиль, тому що цей файл уже є в системній папці Windows, якщо на комп'ютері встановлений Internet Explorer.

На комп'ютері сервера повинен бути інстальований Internet Information Server версії не нижче 4.0 або Netscape Enterprise версії не нижче 3.6. Перераховане ПЗ забезпечує доступ компонента TWebConnection до динамічної бібліотеки HTTPsrvr.dll, що також повинна перебувати на сервері.

Наприклад, якщо файл HTTPsrvr.dll розташований у папці Scripts US 4.0 на Web-сервері **www.someserver.com,** та властивість **URL (string)** повинна містити наступне значення: **http://someserver.com/scripts/httpsrvr.dll**. Якщо URL заданий вірно й сервер настроєний правильно, то в списку властивості **ServerName (string)** в Інспекторі об'єктів з'являється перелік зареєстрованих серверів Прикладних програм. Ім'я одного з них повинне утримуватися у властивості **ServerName**.

Після вибору імені сервера у властивості **ServerGUID (string)** автоматично з'являється GUID сервера. Властивості **UserName (string)** та **Password (string)** при необхідності можуть містити ім'я й пароль користувача, які будуть використані при авторизації.

Властивість **Proxy (string)** містить ім'я використовуваного проксі-серверу.

У заголовок повідомлень HTTP можна помістити ім'я прикладної програми. Для цього використовується властивість **Agent (string)**. З'єднання відкривається й закривається за допомогою властивості **Connected (Boolean)**.

Аналогічні операції виконують методи **procedure Open** та **procedure Close**.

Доступ до інтерфейсу IAppServer надає властивість **AppServer (Variant)** або метод **function GetServer (IAppServer; override)**.

Список доступних з'єднанню серверів прикладних програм повертає метод **function GetServerList (OleVariant; virtual)**. Властивість **ObjectBroker (TCustomObjectBroker)** дозволяє використати екземпляр компонента TSimpieObjectBroker для одержання списку доступних серверів під час виконання (див. нижче).

Методи-обробники подій компонента TWebConnection повністю збігаються з методами-обробниками компонента TDCOMConnection (див. табл. 1).

Провайдери даних

Компонент-провайдер TDataSetProvider являє собою міст між набором даних сервера прикладних програм і клієнтським набором даних. Він забезпечує формування й передачу пакетів даних клієнтської прикладної програми й прийом від нього зроблених змін (див. рис. 2).

Всі необхідні операції компонентів виконує автоматично. Розробнику необхідно лише розмістити компонент TDataSetProvider і зв'язати його з набором даних сервера прикладних програм. Для цього призначена властивість **DataSet (TDataSet)**. Якщо з'єднання в клієнтської прикладної програми настроєно правильно (див. вище), то в списку вибору властивості **ProviderName** компонента TClientDataSet в Інспекторі об'єктів з'являються імена всіх компонентів-провайдерів сервера прикладних програм. Якщо зв'язати клієнтський набір даних з компонентом-провайдером, а потім відкрити його, у клієнтський набір даних будуть передані записи з набору даних сервера прикладних програм, зазначеного у властивості **DataSet** компонента-провайдера TDataSetProvider.

Компонент також містить властивості, що допомагають настроїти процес обміну даними. Властивість **ResolveToDataSet (Boolean)** керує передачею даних від клієнта до сервера БД. Якщо воно має значення True, всі зміни передаються в набір даних сервера прикладних програм, заданий властивістю **DataSet**. Інакше зміни направляються прямо серверу БД. Якщо сервер прикладних програм не повинен відображати зроблені клієнтом зміни, то властивості **ResolveToDataSet** можна привласнити значення False, що прискорить роботу прикладної програми.

Властивість **Constraints (Boolean)** керує передачею обмежень серверного набору даних клієнтському. Якщо властивість має значення True, обмеження передаються.

Властивість **Exported (Boolean)** дозволяє використати в клієнтському наборі даних інтерфейс IAppServer. Для цього властивість повинне мати значення True.

Параметри компонента-провайдера задаються властивістю **type**

TProviderOption = (poFetchBlobsOnDemand, poFetchDetailsOnDemand, poIncFieldProps, poCascadeDeletes, poCascadeUpdates, poReadOnly, poAllowMultiRecordUpdates, poDisablelnserts, poDisableEdits, poDisableDeletes, poNoReset, poAutoRefresh, poPropogateChanges, poAllowCoinmandText, poRetainServerOrder);

TProviderOptions = set of TProviderOption;

Набір параметрів властивості задається присвоєнням елементам значення True.

property Options: TProviderOptions;

poFetchBlobsOnDemand - включає передачу в клієнтський набір даних значень полів типу BLOB. За замовчуванням ця можливість, відключена для прискорення роботи;

poFetchDetailsOnDemand - включає передачу в клієнтський набір даних підлеглих записів для відношення "один-до-багатьох". За замовчуванням ця можливість відключена для прискорення роботи;

poIncFieldProps - включає передачу в клієнтський набір даних декількох властивостей для об'єктів полів: Alignment, DisplayLabel, DisplayWidth, Visible, DisplayFormat, EditFormat, MaxValue, MinValue, Currency, EditMask, DisplayValues;

poCascadeDeletes - включає автоматичне видалення підлеглих записів у відношенні " один-до-багатьох " на стороні сервера, якщо головний запис був віддалений у клієнтському наборі даних;

poCascadeUpdates - включає автоматичне відновлення підлеглих записів у відношенні " один-до-багатьох " на стороні сервера, якщо головний запис був змінений у клієнтському наборі даних;

poReadOnly - включає режим "тільки для читання" для набору даних сервера;

poAllowMultiRecordUpdates - включає режим внесення змін відразу в кілька записів одночасно. Інакше всі записи змінюються послідовно, одна за однією;

poDisableInserts - забороняє клієнтові вносити в набір даних сервера нові записи;

poDisableEdits - забороняє клієнтові вносити в набір даних сервера зміни;

poDisableDeletes - забороняє клієнтові видаляти записи в наборі даних сервера;

poNoReset - забороняє відновлення набору даних сервера перед передачею записів клієнтові (перед викликом методу AS_GetReccrds інтерфейсу IAppServer);

poAutoRefresh - включає автоматичне відновлення записів клієнтського набору даних. За замовчуванням ця можливість відключена для прискорення роботи;

poPropogateChanges - якщо в методах-обробниках BeforeUpdateRecord або AfterUpdateRecord клієнтського набору даних були зроблені додаткові зміни, то після їхнього запису в наборі даних сервера, зміни знову направляються клієнтові для відновлення запису. У включеному стані ця можливість дозволяє повністю контролювати збереження змін на сервері;

poAllowCommandText - дозволяє змінювати текст запиту SQL, імена збережених процедур або таблиць у компоненті набору даних на сервері прикладних програм;

poRetainServerOrder - включає заборону на зміну порядку сортування записів клієнтом. Якщо цей параметр відключити, можливі помилки відображення набору даних, що проявляються в появі подвійних записів.

Методи-обробники компонента-провайдера даних представлені в табл. 3.

Таблиця 3. Методи-обробники подій компонента TDataSetProvider

Оголошення	Опис
property AfterApplyUpdates: TRemoteEvent;	Викликається після збереження змін, переданих від клієнта, у наборі даних сервера
property AfterExecute: TRemoteEvent;	Викликається після виконання запиту SQL або збереженої процедури на сервері
property AfterGetParams: TRemoteEvent;	Викликається після того, як компонент-провайдер сформував набір параметрів набору даних сервера для їхньої передачі клієнтові
property AfterGetRecords: TRemoteEvent;	Викликається після того, як компонент-провайдер сформував пакет даних для передачі набору даних сервера клієнтові
property AfterRowRequest: TRemoteEvent ;	Викликається після відновлення поточного запису клієнта компонентом-провайдером
property AfterUpdateRecord: TAfterUpdateRecordEvent ;	Викликається відразу після відновлення одиничного запису на сервері
property BeforeApplyUpdates: TRemoteEvent ;	Викликається перед збереженням змін, переданих від клієнта, у наборі даних сервера
property BeforeExecute: TRemoteEvent;	Викликається перед виконанням запиту SQL або збереженої процедури на сервері
property BeforeGetParams: TRemoteEvent ;	Викликається перед тим, як компонент-провайдер сформував набір параметрів набору даних сервера для їхньої передачі клієнтові
property BeforeGetRecords: TRemoteEvent ;	Викликається перед тим, як компонент-провайдер сформував пакет даних для передачі набору даних сервера клієнтові
property BeforeRowRequest: TRemoteEvent ;	Викликається перед відновленням поточного запису клієнта компонентом-провайдером
property BeforeUpdateRecord: TBeforeUpdateRecordEvent;	Викликається безпосередньо перед відновленням одиничного запису на сервері
property OnDataRequest: TDataRequestEvent;	Викликається при обробці запиту на одержання даних клієнтом

property OnGetData: TProviderDataEvent;	Викликається після одержання даних від набору даних сервера, але перед їхнім відправленням клієнтові
property OnGetDataSetProperties: TGetDSProps;	Викликається при створенні структури параметрів набору даних сервера для їхньої передачі клієнтові
property OnGetTableName: TGetTableNameEvent;	Викликається при одержанні компонентом-провайдером імені таблиці, що підлягає відновленню
property OnUpdateData: TProviderDataEvent ;	Викликається при збереженні змін у наборі даних сервера
property OnUpdateError: TResolverErrorEvent;	Викликається при виникненні помилки збереження змін у наборі даних сервера

Компонент TLocalConnection

Компонент TLocalConnection використовується локально для одержання доступу до існуючих компонентів-провайдерів.

Властивість **Providers[const ProviderName: string] (TCustomProvider)** містить посилання на всі компоненти-провайдери, розміщені з компонентом TLocalConnection на одній формі. Індексація в списку здійснюється по імені компонента-провайдера. Загальне число компонентів-провайдерів у списку повертає властивість **ProviderCount (Integer)**.

Крім цього, за допомогою компонента TLocalConnection можна одержати доступ до інтерфейсу IAppServer локально. Для цього використовується властивість **AppServer (AppServer)** або метод **function GetServer (IAppServer; override)**.

Компонент TSharedConnection

Якщо інтерфейс IAppServer віддаленого модуля даних має метод, що повертає посилання на аналогічний інтерфейс іншого віддаленого модуля даних, то перший модуль називається головним, а другий - дочірнім. Компонент TSharedConnection використовується для з'єднання клієнтського прикладної програми з дочірнім віддаленим модулем даних сервера прикладних програм.

Властивість **ParentConnection (TDispatchConnection)** повинна містити посилання на компонент з'єднання з головним віддаленим модулем даних сервера прикладних програм. Дочірній віддалений модуль даних визначається властивістю **ChildName (string)**, яка повинне містити його ім'я. Якщо інтерфейс головного віддаленого модуля даних настроєний правильно, то в списку вибору властивості в Інспекторі об'єктів з'являються імена всіх дочірніх віддалених модулів даних.

Інтерфейс IAppServer дочірнього віддаленого модуля даних повертає властивість **AppServer (Variant)** або метод **function GetServer (lAppServer; override)**.

Методи-обробники компонента TSharedConnection успадковані від класу предка TCustomConnection.

Компонент TConnectionBroker

Компонент TConnectionBroker забезпечує централізоване керування з'єднанням клієнтських наборів даних із сервером прикладних програм. Для цього властивість **connectionBroker** клієнтських наборів даних повинна посилатися на екземпляр компонента TConnectionBroker. Тоді для зміни з'єднання (наприклад, при переході із транспорту HTTP на сокети TCP/IP) немає необхідності змінювати значення властивості **RemoteServer** всіх компонентів TClientDataSet, а досить змінити властивість **Connection (TCustomRemoteServer)** компонента TConnectionBroker.

Доступ до інтерфейсу IAppServer забезпечує властивість **AppServer (Variant)** або метод **function GetServer (lAppServer; override)**.

Методи-оброблювачі компонента TConnectionBroker повністю відповідають табл. 1.

Розробка розподіленої прикладної програми на основі DCOM

Перед розробкою розподіленого необхідно мати створену за допомогою InterBase базу даних (*.gdb), яку ми і будемо під'єднувати до нашої прикладної програми.

1. Розробка серверу

Розробимо прикладну програму - сервер.

1. Створіть нову Delphi-прикладну програму.
2. Змініть значення властивості форми Caption на DataSnap-сервер.
3. Відкрийте Object Repository за допомогою команди File/New/Other.
4. Оберіть майстер Remote Data Module на вкладці Multitier.
5. Визначіть CoClass Name як srvDataSnap та клацніть мишею на кнопці OK.
6. Помістіть з вкладок InterBase та DataAcces в модуль DataSnap компоненти, які зображені на рис. 7.

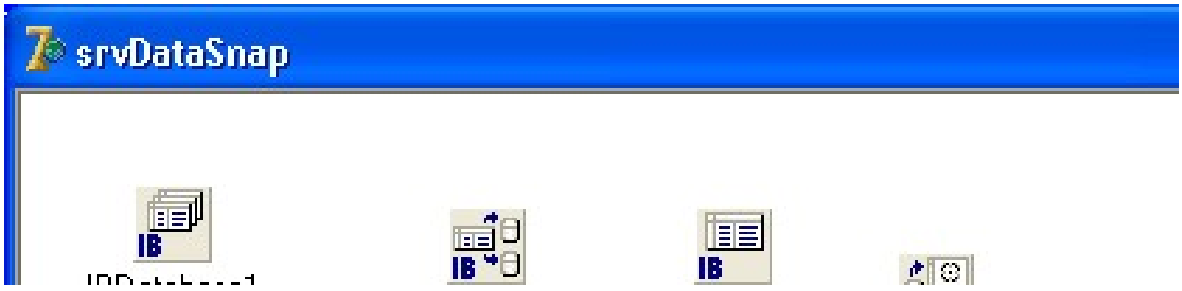


Рис. 1. Модуль даних DataSnap-серверу

7. Визначте їхні властивості наступним чином:

Компонент	Властивість	Значення
TIBDataBase	DataBaseName	Повний шлях до БД (напр. D:\dbApp\clSrv\Sport.gdb)

	Params	Ім'я користувача (USER_NAME) та пароль (PASSWORD), які зазначалися при створенні БД у IBConsole (напр. USER_NAME=SYSDBA PASSWORD=masterkey)
TIBTransaction	DefaultDatabase	IBDatabase1
TIBTable	DataBase Transaction TableName Active	IBDatabase1 IBTransaction1 Ім'я таблиці з обраної раніше БД True
TdataSetProvider	DataSet	IBTable1

8. Збережіть модуль головної форми (Unit1.pas) під ім'ям fmDataSnap.pas, реалізацію DataSnap-серверу (Unit2.pas) під ім'ям dmDataSnap.pas, а сам проект – як DataSnapServer.dpr у папці Server.
9. Відкомпілюйте та запустіть сервер.

Якщо ви все зробили вірно, то результат буде таким, як показано на рис.8.

Рис. 8. Форма DataSnap-серверу

Отже, ви переконалися, що DataSnap-сервер працює. Працює – це не значить, що сервер виконує щось. Просто з'явилося пусте вікно – підтвердження працездатності серверу.

Якщо ж виникає необхідність створення віддаленого, а не локального, серверу, то необхідно змінити деякі властивості компоненту TIBDataBase. Для цього клацніть на компоненті двічі (або правою клавішею миші і виберіть Database Editor…) рис.9,10.

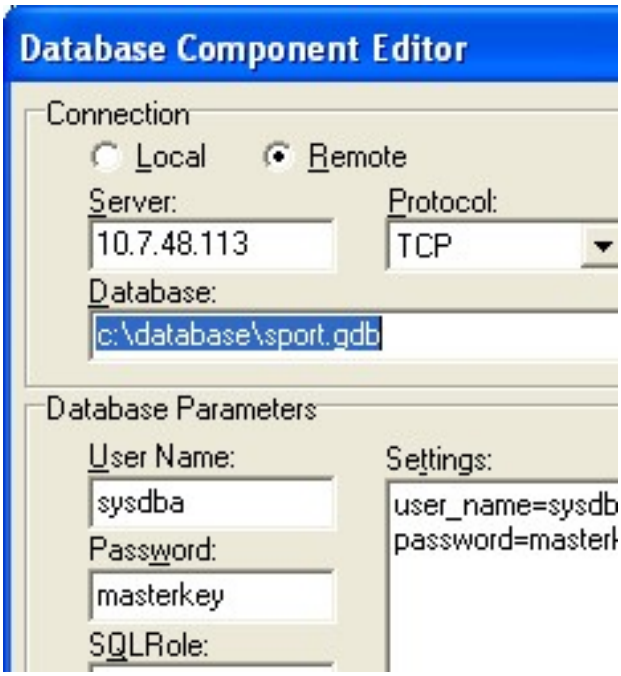


Рис.9. Вигляд діалогового вікна Database Editor…

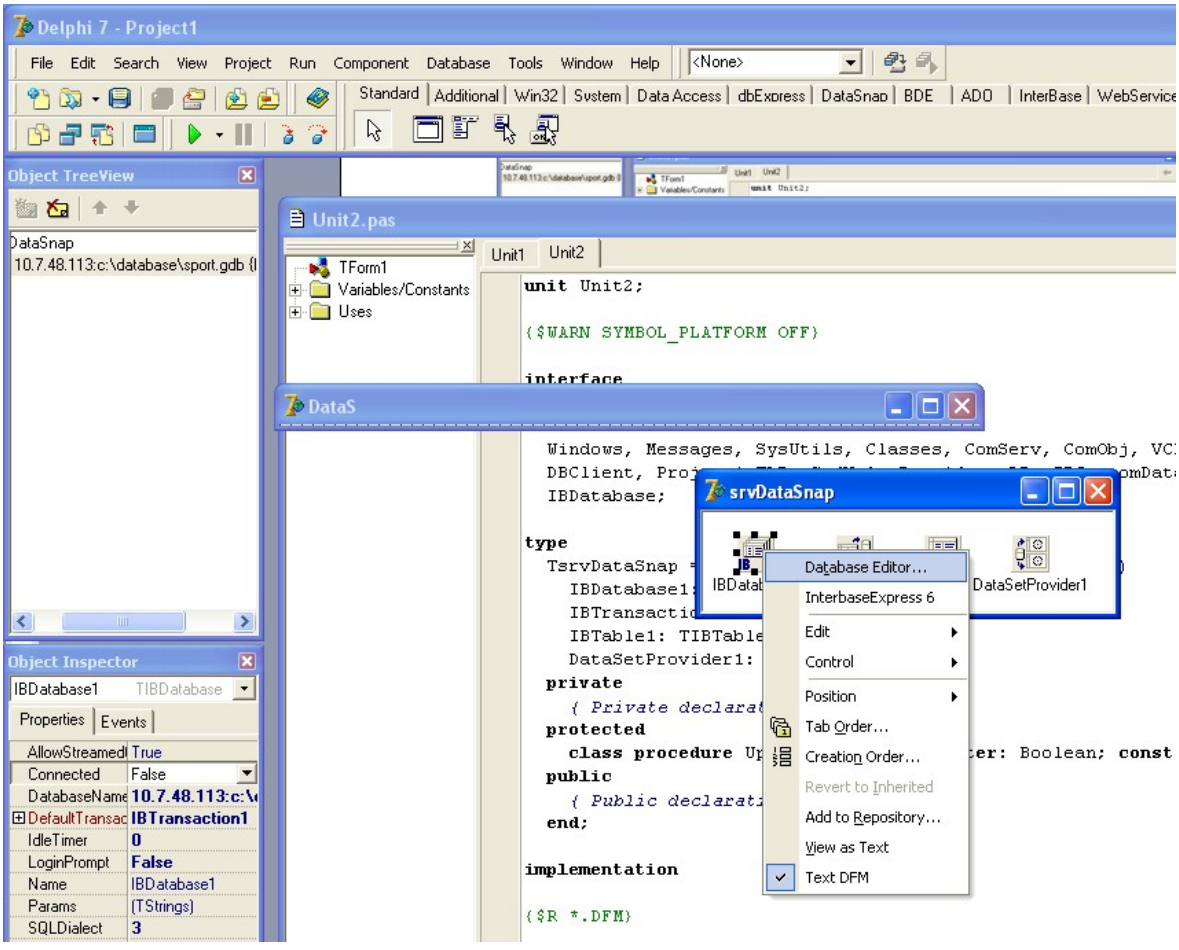


Рис. 10. Меню об’єкту TIBDataBase

У цьому вікні (Рис.9) у частині Connection необхідно вказати вид зв’язку (Remote), IP-адресу сервера, тип протоколу (TCP) та шлях до файлу БД на сервері. У частині Database

Parameters слід зазначити ім’я користувача, пароль та визначити Login Promt для коректної роботи прикладної програми.

Розробка клієнта

1. Створіть нову Delphi-прикладну програму.
2. Помістіть на форму компонент TDCOMConection з вкладки DataSnap.
3. Тепер необхідно вказати сервер. Використовуючи випадаючий список, визначіть властивість ServerName компонента TDCOMConection як DataSnapServer.srvDataSnap.
4. Для того, щоб протестувати зв'язок, встановіть властивість Connected рівним True. Клієнт знайде сервер та запустить його.
5. Помістіть на форму компонент TClientDataSet з вкладки Data Acces.
6. Встановіть значення властивості RemoteServer рівним DCOMConnection1.
7. Оберіть із списку значень властивості ProviderName інтерфейс провайдера DataSetProvider1.
8. Помістіть на форму компонент TDataSource та з’єднайте його з набором даних TClientDataSet.
9. Для завершення формування інтерфейсу клієнта помістіть на форму компонент DBGrid з вкладки Data Controls та з’єднайте його з TDataSource.
10. Встановіть значення властивості Active компонента TClientDataSet рівним True. Якщо все виконано вірно, то при виконанні програми ми зможемо побачити нашу таблицю (ім’я якої ми визначали ще під час створення серверної прикладної програми).

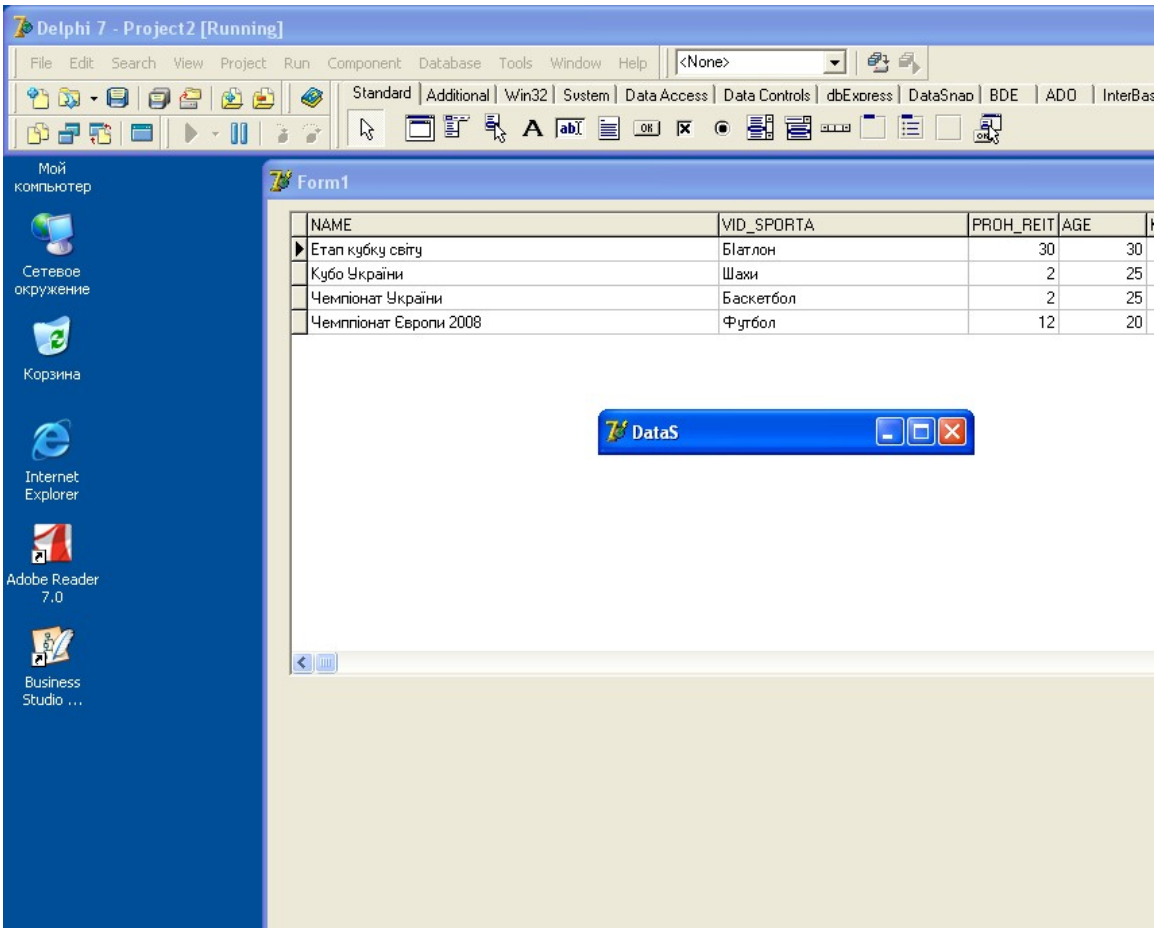


Рис. 11. Реалізація клієнта (вікно сервера на передньому плані)

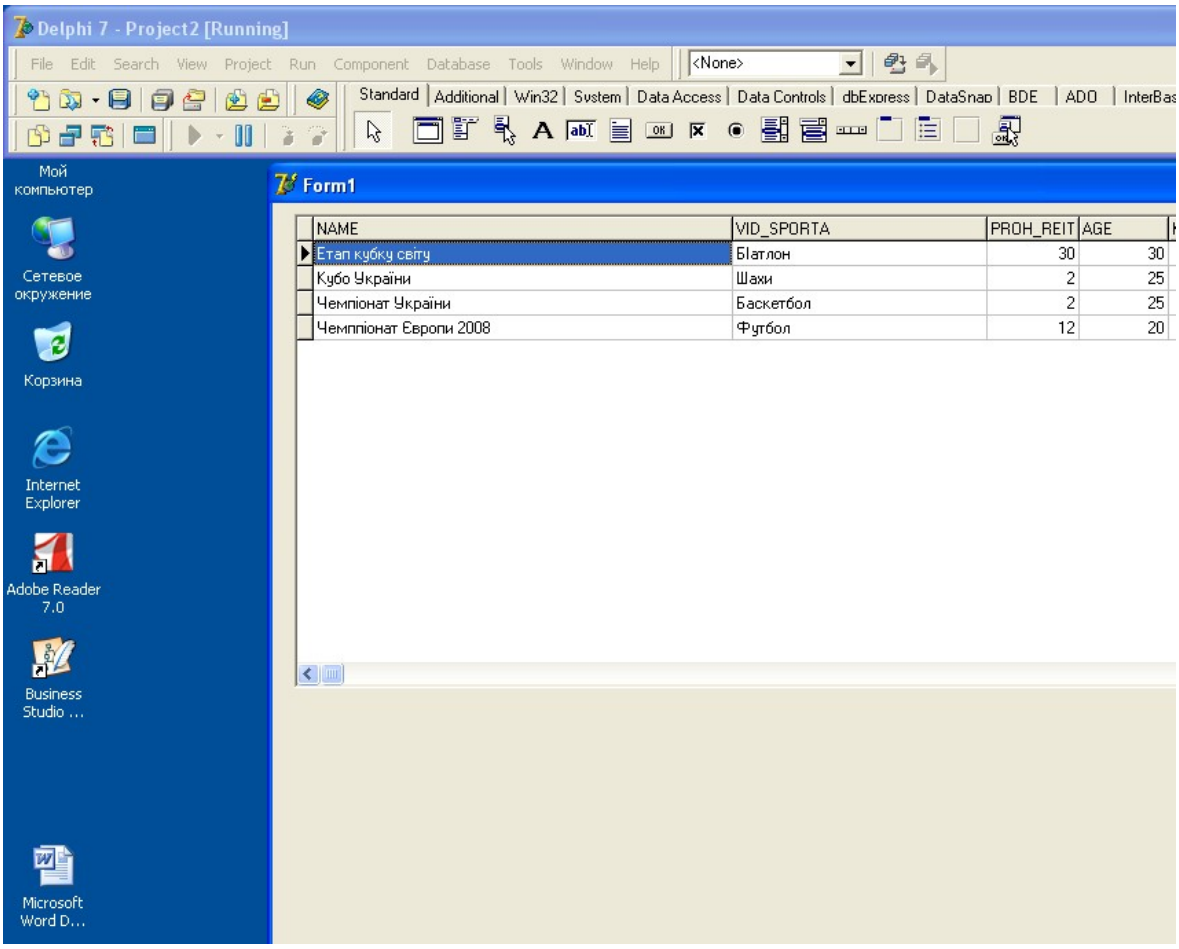


Рис. 12. Реалізація клієнта

4.Завдання на виконання

4.1Розбийтесь на бригади по два.

4.2 Зареєструйтесь як користувач на сервері.

4.3 Розробіть базу даних на сервері згідно варіанта та клієнтську прикладну програму. Для варіанта використайте лабораторну роботу по технології InterBase

Література

1. Грофф Дж., Вайнберг П. Энциклопедия SQL. 3-е изд. СПб.: Питер, 2003.
2. Дейт К. Дж. Введение в системы баз данных.: Пер. с англ. – 6-е изд. – К.: Диалектика, 1998. – 784 с.
3. Джеймс Р. Грофф, Пол Н. Вайнберг, SQL: полное руководство: пер. с англ. – К.: Издательская группа BHV, 2000. – 608с.
4. Крёнке Д. Теория и практика построения баз данных. 8-е изд. –СПб: Питер, 2003. – 800 с.
5. Ульман Дж. Основы систем баз данных: Пер. с англ. / Под ред. М.Р. Когаловского. – М.: Финансы и статистика, 1983. – 334 с.

6. Сайт Oracle http://www.oracle.com.
7. Сайт Sybase http://www.sybase.com.
8. Сайт компанії IBM в Україні http://www.ibm.com/ua/.
9. Сайт «Открытые системы» http://www.osp.ru.
10. Сайт компанії Interface ltd http://www.interface.ru.
11. Сайт http://msdn2.microsoft.com/en-us/library/ms675532.aspx
12. А. Чекалов "Базы данных: от проектирования до разработки приложений";
13. Черняк О.І., Ставицький А.В., Чорноус Г.О. Системи обробки економічної інформації: Підручник.-К.:Знання, 2006.-447 с. –Вища освіта XXI століття).
14. Інформаційні системи і технології в економіці/Під ред.В.С.Пономаренка.-К.:ВЦ Академія, 2002.
15. Ситник В.Ф. та ін. Основи інформаційних систем. –К.: КНЕУ, 2001
16. Цикритизис Д., Лоховски Ф. Модели данных. – М.: Финансы и статистика, 1985. – 344 с.
17. Джексон Г. Проектирование реляционных баз данных для использования с микроЭВМ. - М.: Мир, 1991. – 252 с.
18. Васкевич Д. Стратегии клиент/сервер. - Киев: Диалектика, 1997.
19. Чаудхари С. Методы оптимизации запросов в реляционных системах //СУБД. - 1998. - №3. - С.22-36.
20. Codd E.F. Relation Model of Data for Large Shared Data Banks //Comm. ACM. - 1970. - V.13, №.6. - P.377-383. (Переклад на російську мову: Кодд Е.Ф. Реляционная модель данных для больших совместно используемых банков данных //СУБД. - 1995. - №1. - С.145-160.)
21. Конноли Т., Бегг К., Страчан А. Базы данных: проектирование, реализация и сопровождение. Теория и практика. 2-е издание. – М.: Издательский дом "Вильямс", 2001. – 1120 с.
22. Сергієнко І.В. Інформатика в Україні: становлення, розвиток, проблеми. Київ, Наукова думка, 1999, 354 с.
23. Малиновский Б.Н. Истори вычислительной техники в лицах. //К.: КИТ, 1995, с.380.
24. Когаловский М.Р. Энциклопедия технологий баз данных - М.: Финансы и статистика, 2002.
25. Федоров А., Елманова Н. "Введение в базы данных" // "Компьютер-Пресс", №3, 2000
26. Кузнецов С.Д. Основы современных баз данных http://www.citforum.ru.
27. Анатолий Хомоненко, Владимир Гофман, Евгений Мещеряков, Владимир Никифоров Delphi 7. Наиболее полное руководство BHV СПб

Предметний покажчик