Зміст

С.

# 1 Основи проектування реляційних баз даних

## 1.1 Інформаційні системи з базами даних

### 1.1.1 Інформація й дані

Перш ніж перейти до обговорення поняття інформаційної системи (ІС), спробуємо з'ясувати, що ж розуміється під словом інформація. Відповісти на це питання й просто, і складно: слово "інформація" пов'язане із широким колом понять.

Змістовна сторона поняття "інформація" дуже багатогранна й немає чітких семантичних меж. Однак завжди можна сказати, що можна з нею робити. Саме відповідь на це питання найчастіше й цікавить як системних аналітиків і розроблювачів ІС, так і користувачів інформації (її основних споживачів).

З погляду як користувачів, так і розроблювачів ІС, в інформації є одна важлива властивість - вона є одиницею даних, яка підлягає обробці. Звичайно інформація надходить споживачеві саме у вигляді даних: таблиць, графіків, малюнків, фільмів, усних повідомлень, які фіксують у собі інформацію певної структури й типу. Таким чином, дані виступають як засіб подання інформації у певній, фіксованій формі, придатній для обробки, зберігання й передачі. Хоча дуже часто терміни "інформація" й "дані" виступають як синоніми, варто пам'ятати про цю їхню істотну відмінність. Саме в даних інформація знаходить інтерпретацію у конкретній ІС.

При згадуванні про "форму" подання інформації варто сказати ще про одну, "людську" властивість інформації - її сприйняття різними категоріями людей. Дані можуть бути згруповані спільно у документ. Документ може мати або не мати певну внутрішню структуру. Дані можуть бути відображені на екрані дисплея комп'ютера. Документи можуть мати аудіо- або відеоформу. Розробляючи ІС, ніколи не потрібно забувати, для кого вони (системи) створюються й хто буде їх використовувати. Форма подання інформації в ІС визначає також і категорії користувачів. ІС створюються для конкретних груп користувачів, тобто вони, як правило, проблемно-орієнтовані.

*Інформація* є дані, яким надається деякий зміст (інтерпретація) у конкретній ситуації у рамках деякої системи понять. Інформація представляється за допомогою кодування даних і витягається шляхом їхнього декодування й інтерпретації.

У цьому визначенні фіксується три основних перетворення інформації й даних у процесі їхньої обробки в ІС: інформація – дані, дані – дані, дані – інформація.

На рис. 1.1 подані дві сторони визначення поняття інформації: функціональна й представницька. Перша загалом визначає коло дій над інформацією, а друга – результат виконання цих дій.

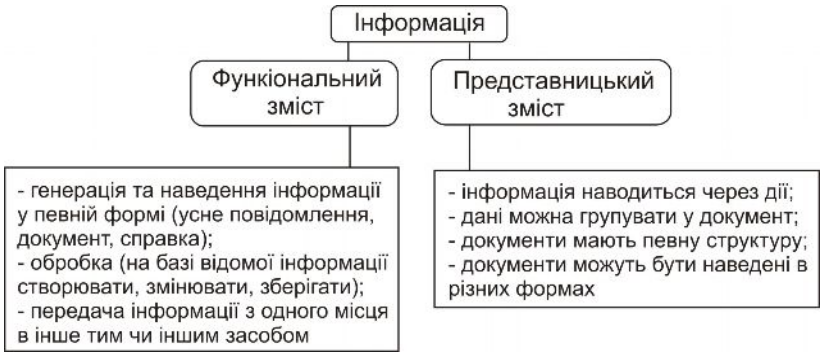



Рисунок 1.1 – Зміст поняття "інформація"

### 1.1.2 Інформаційні системи

Основною метою створення ІС є задоволення інформаційних потреб користувачів шляхом надання необхідної їм інформації на основі збережених даних. Потреба в інформації як такій не вичерпує поняття інформаційних потреб. Звичайно в поняття інформаційних потреб включають певні вимоги до якості інформаційного обслуговування й поводження системи в цілому (продуктивність, актуальність і надійність даних, орієнтація на користувача та ін.).

Під *інформаційною системою* розуміється організаційна сукупність технічних засобів, технологічних процесів і кадрів, що реалізують функції збору, обробки, зберігання, пошуку, видачі й передачі інформації.

Необхідність підвищення продуктивності праці у сфері інформаційної діяльності призводить до того, що як зовнішні засоби зберігання й швидкий доступ до інформації найчастіше використовуються засоби обчислювальної техніки (цифровий й аналоговий) на основі комп'ютерів. Сучасні ІС - складні комплекси апаратних і програмних засобів, технології й персоналу, які ще називають автоматизованими інформаційними системами. Структурно ІС містять у собі апаратне (hardware), програмне (software), комунікаційне (netware), проміжного шару (middleware), лінгвістичне й організаційно-технологічне забезпечення.

*Апаратне забезпечення* ІС містить у собі широкий набір засобів обчислювальної техніки, передачі даних, а також цілий ряд спеціальних технічних пристроїв (пристрою графічного відображення інформації, аудио- і відеопристрою, засобу мовного уведення й т.д.). Апаратне забезпечення є основою будь-якої ІС.

*Комунікаційне (мережне) забезпечення* містить у собі комплекс апаратних мережних комунікацій і програмних засобів підтримки комунікацій в ІС. Воно має істотне значення при створенні розподілених ІС й ІС на основі Інтернету.

*Програмне забезпечення* ІС забезпечує реалізацію функцій введення даних, їх розміщення на носіях, модифікації даних, доступ до даних, підтримку функціонування устаткування. Програмне забезпечення можна розділити на системне (яке вінчає процес вибору апаратно-програмного рішення, або платформи) і користувальницьке (яке застосовується для рішення завдань задоволення потреб користувача у комп'ютерному середовищі).

*Лінгвістичне забезпечення* ІС призначене для рішення завдань формалізації змісту повнотекстової й спеціальної інформації для створення пошукового образу даних (профілю). У класичному змісті звичайно воно включає процедури індексування текстів, їхню класифікацію й тематичну рубрикацію. Найчастіше ІС, що містять складно-структуровану інформацію, містять у собі тезауруси термінів і понять. Сюди можна віднести й створення процесорів спеціалізованих формальних мов кінцевих користувачів, наприклад мов для маніпулювання бухгалтерською інформацією й т.д. Найчастіше роботам по розробці лінгвістичного забезпечення не надається належного значення. Подібні недогляди найчастіше ведуть до несприйняття користувачами самої. Це відноситься в першу чергу до вузько спеціалізованих ІС.

У міру зростання складності й масштабів ІС важливу роль починає грати *організаційно-технологічне забезпечення*, що з'єднує різнорідні компоненти (апаратури, програми й персонал) у єдину систему й забезпечує процедури її керування й функціонування. Недооцінка цієї складової ІС найчастіше призводить до зриву строків впровадження системи й виводу її на виробничі потужності.

На рис. 1.2 наведені функції ІС через її основні структурні компоненти.

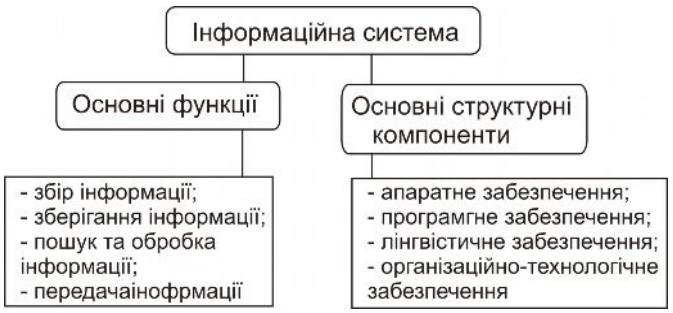



Рисунок 1.2 – Визначення інформаційної системи

### 1.1.3 Ітераційна процедура побудови ІС

Традиційно й повсюди, особливо на початкових етапах розвитку інформаційної інфраструктури організації, використовується так званий позадачний метод рішення завдань автоматизації, спрямований на рішення досить простих і зрозумілих керівництву завдань. Наприклад, виписка рахунків, підготовка документів. Кон'юнктурна перевага такого методу очевидна: досить швидко може бути отриманий результат, існування модної нині ІТ-служби виправдано. Даний метод дозволяє, з одного боку, начебто б не відставати від життя (наявність ІС в організації найчастіше є одним з визначальних факторів її конкурентноздатності), а з іншого боку - заощаджувати кошти на автоматизації. Вищевказаний підхід дозволяє використовувати службовців невисокої кваліфікації. Рано або пізно це стає гальмом у розвитку інформаційної інфраструктури організації.

Зміна напрямів бізнесу організації призводить до питання перегляду відносин до ІС в організації, тобто до питання - переробити або почати спочатку. Почати спочатку завжди вигідніше. Можна застосовувати вже добре відпрацьовані в інформатиці методики проектування "донизу" або "знизу-догори". Однак рано або пізно знову підніматимиться питання про відповідність вимогам сьогоднішнього дня.

Розроблювачі ІС фактично завжди користуються методикою "зсередини" (midlle of design): є деяка створена основа і навколо неї варто розвиватися у різних напрямах, не сильно нехтуючи сформованими традиціями. Таким чином, постулюється ітераційний підхід у розробці й створенні ІС, що визначається життєвою необхідністю.

Основна особливість реалізації концепції розробки ІС, орієнтованої на інтегровані процеси, - це наявність або відсутність складального конвеєра, оскільки необхідно збирати докупи багато технологічних процесів обробки інформації. При об'єднанні технологічних процесів обробки інформації збільшується швидкість проходження інформації у системі, прийняття рішень на основі інформаційних потоків стає частиною процесу обробки інформації, зменшується ієрархія управлінських структур.

Для того, щоб ІС жила довго і її експлуатація приносила відчутну вигоду, необхідно ретельно проектувати і її архітектуру, і її складові компоненти, зокрема БД, про які піде мова нижче.

### 1.1.4 Основні підходи до обробки інформації в автоматизованих ІС

Одним із головних питань розробки програмного забезпечення ІС є питання про співвідношення програм і даних, тому що вирішення цього питання, в остаточному підсумку, визначає вибір алгоритмів обробки інформації, апаратних засобів і технологічної платформи. Фундаментальним принципом у вирішенні питання про співвідношення програм і даних є концепція незалежності прикладних програм від даних, і неважливо, яка обробка даних передбачається: централізована або розподілена. Суть цієї концепції полягає не стільки у відділенні програм від даних, скільки у розгляді їх як самостійних взаємодіючих об'єктів.

Однією з останніх модифікацій цього принципу є концепція незалежності прикладних програм від даних разом із процедурами їхньої обробки (об’єктно-орієнтований підхід у програмуванні), що дозволяє вирішити ряд питань обробки даних, пов'язаних з інтерпретацією семантичного змісту даних.

Формування концепції БД (БД) і створення на її основі методу баз даних для вирішення завдань обробки інформації відбулося у 1962 році. До середини 60-х років минулого століття основною концепцією побудови програмного забезпечення була концепція файлової системи й так званий позадачний метод. Наприкінці 80-х років минулого століття була запропонована концепція об’єктно-орієнтованих баз даних й об’єктно-орієнтований підхід розроблення програм на основі обробки подій. На рис. 1.3 наведені основні риси для кожної з зазначених вище концепцій. На рис. 1.4 проведене зіставлення основних методів обробки даних.

Основний зміст позадачного методу зводиться до декомпозиції програми зі своїми окремими блоками даних та алгоритмами; методу баз даних – до наявних окремих описів логічної структури даних та єдиної точки зору щодо процедури обробки даних; об'єктно-орієнтованого методу, який полягає в тому, що програми розглядаються як сукупність об'єктів, між якими відбувається обмін інформацією.

Об'єкту притаманні такі властивості:

інкапсуляція – об'єкти наділяються структурою й мають певне поводження (набором операцій). Операції над об'єктами становлять його методи. Структура об'єкта схована від користувача, що маніпулює об'єктом через його операції. Об'єкт розглядається як абстракція реального світу. Для того, щоб об'єкт виконав деяку дію, йому потрібно послати повідомлення. Об'єкт взаємодіє з іншими об'єктами через події.


Рисунок 1.3 – Основні концепції обробки інформації


Рисунок 1.4 – Основні проблеми методів обробки інформації

- спадкування – являє собою механізм, що дозволяє робити одні об'єкти з інших, при цьому властивості батьківського об'єкта зберігаються у нащадка;
- поліморфізм – різні об'єкти можуть одержувати однакові повідомлення, але реагувати на них по-різному, відповідно до реалізації своїх однойменних методів.

### 1.1.5 Концепція баз даних

*Базу даних* у загальному випадку можна визначити як уніфіковану сукупність збережених і відтворених даних, що використовуються у рамках організації (Engles R.A., 1972 р.). Однак поняття БД не ґрунтується в цей час на єдиній концепції, скоріше це ціле сімейство пов'язаних між собою понять з ПО, програмного й апаратного забезпечення, аналізу й моделювання даних і додатків. Існує кілька визначень БД.

*База даних* (за Дж. Мартіном) є сукупність взаємозалежних даних, які спільно використовуються декількома додатками й зберігаються з мінімальною регульованою надлишковістю. Дані запам'ятовуються таким чином, щоб вони, у міру можливості, не залежали від програм. Для обробки даних застосовується загальний керуючий метод доступу. Якщо БД не перетинаються за структурою, то говорять про систему баз даних.

*База даних* (відповідно до матеріалів комітету КОДАСІЛ) складається зі всіх екземплярів записів, екземплярів наборів записів й областей, які контролюються конкретною схемою. Під схемою можна розуміти карту всієї логічної структури БД.

Для розроблювача ІС істотним моментом при використанні концепції баз даних (БД) є та обставина, що дані стають певним чином організовані, здобувають якусь упорядкованість і внутрішню структуру, а також те, що є деякий набір уніфікованих операцій обробки даних і декларативних засобів подання даних. До таких операцій варто віднести операції "Вставити" (Insert), "Додати" (Add), "Видалити" (Delete) і ряд інших. До декларативних засобів подання даних варто віднести мови визначення даних. Тобто використання даної концепції при створенні ІС припускає наявність мови визначення даних і мови маніпулювання даними, а також правил побудови інтерфейсів програм (додатків)із БД і користувачем.

Такий розподіл засобів маніпулювання даними і їхнього подання є деякою мірою умовним. Мова визначення даних служить для опису логічної структури (схеми) БД, а в деяких випадках і способів зберігання й доступу до даних. Мова маніпулювання даними надає алгоритмічні засоби побудови додатків для обробки елементів даних, які зберігаються у БД.

### 1.1.6 Системи керування базами даних

У випадку застосування концепції БД для створення ІС природно виникає запитання - а хто або що повинен все це підтримувати? Таким чином, постає питання про систему керування базою даних (СКБД). СКБД є складними програмними системами, що працюють на різних операційних платформах. Саме СКБД повинна надати засоби визначення й маніпулювання даними, зробивши дані незалежними від прикладних програм, що їх використовують.

До основних функцій СКБД необхідно віднести:

- забезпечення мовних засобів опису та маніпуляції даними;
- забезпечення підтримки логічної моделі даних;
- забезпечення взаємодії логічної та фізичної структур даних;
- забезпечення захисту та цілісності даних;
- забезпечення підтримки БД в актуальному стані.

*Системою керування базами даних* (Data-base Management System) називається сукупність програмних засобів, необхідних для використання БД і подання розробникам і користувачам безліч різних подань даних.

### 1.1.7 Поняття про моделі даних

Подання інформації за допомогою даних вимагає уніфікованого підходу до поняття даних як незалежного об'єкта моделювання. Тому для розробника ІС вибір відповідної моделі даних є однією з найважливіших проблем. Вибір моделі даних спричиняє вибір засобів аналізу ПО, як області реального світу, що підлягає вивченню й обробці. Модель даних обмежує можливість вибору СКБД, тому що звичайно окремо взята модель підтримує певну модель даних. Таким чином, поняття моделі даних є одним із фундаментальних понять інформатики, від якого багато в чому залежать механізми реалізації ІС як програмно-апаратного комплексу.

*Модель даних* (Data Model) є логічна структура даних, що представляє притаманні цим даним властивості, незалежні від апаратного й програмного забезпечення й не пов'язані з функціонуванням комп'ютера.

Можна розглянути кілька аспектів моделювання в обробці даних:

- інформаційне моделювання;
- концептуальне моделювання (моделювання семантики ПО);
- логічне моделювання даних;
- фізичне моделювання:
- створення моделей доступу до даних;
- оптимізація фізичної організації даних в апаратному середовищі.

На рис. 1.5 ілюструється загальний зміст поняття моделі даних на теперішній час.

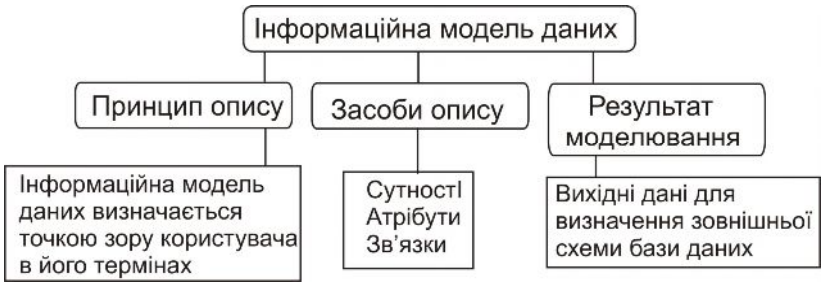



Рисунок 1.5 – Подання про інформаційну модель даних

***Основні типи моделей й їхня еквівалентність***

Наявність у СКБД певної структури даних призводить до поняття баз структурованих даних, тобто дані в таких БД повинні бути представлені як сукупність взаємозалежних елементів. Варто мати на увазі, що для кожного типу БД використовуються відповідні моделі даних.

У цей час для баз структурованих даних розрізняють три основних типи логічних моделей даних залежно від характеру підтримуваних ними зв'язків між елементами даних - мережну, ієрархічну й реляційну. Ознаками класифікації у цих моделях є: ступінь твердості (фіксації) зв'язку, математичне подання структури моделі й припустимих типів даних (див. таблицю 1.1).

Рис. 1.6 ілюструє особливості кожної моделі даних. При зіставленні моделей варто пам'ятати, що всі вони теоретично еквівалентні. Еквівалентність моделей полягає в тому, що вони можуть бути зведені одна до іншої шляхом формальних перетворень.

Таблиця 1.1 – Загальні характеристики моделей даних

| Модель даних | Характер зв'язків між об'єктами | Формальне подання |
|---|---|---|
| Мережна | Напівтверді зв'язки | Довільний граф |
| Ієрархічна | Тверді зв'язки | Деревоподібна структура |
| Реляційна | Мінливі зв'язки | Плоский файл |

## 1.2 Предметна область БД та її моделі

### 1.2.1 Поняття предметної області

Основним призначенням ІС є оперативне забезпечення користувача інформацією про зовнішній світ шляхом реалізації питально-відповідного відношення. Питально-відповідні відношення дозволяють виділити для

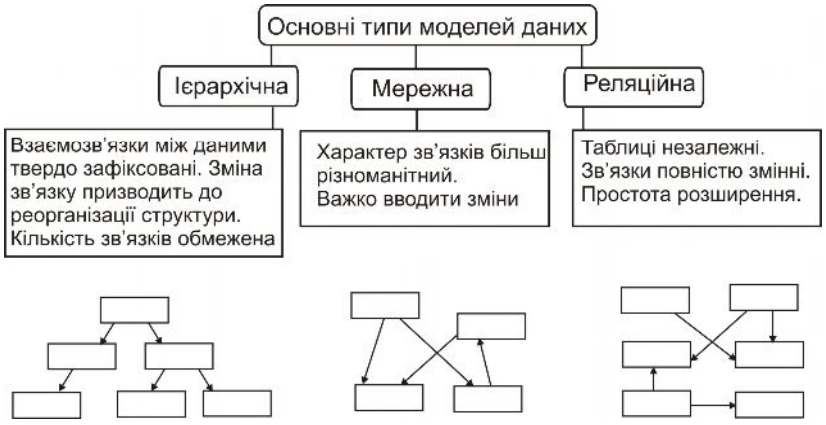



Рисунок 1.6 – Основні типи моделей даних

ІС певний її фрагмент - предметну область (ПО), - який буде втілений в автоматизованій ІС. Інформація про зовнішній світ подається в ІС у формі даних, що обмежує можливості змістовної інтерпретації інформації й конкретизує семантику її подання в ІС. Сукупність цих виділених для ІС даних, зв'язків між ними й операцій над ними утворить інформаційну й функціональну моделі ПО, що описують її стан із певною точністю. Інформаційна й функціональна моделі ПО є вхідними даними для процесу проектування БД.

Сукупність реалій (об'єктів) зовнішнього світу - об'єктів, про які можна задавати питання, - утворює об'єктне *ядро ПО*, яке має онтологічний статус. Не можна одержати в ІС відповідь на питання про те, що їй невідомо. Термін "об'єкт" є первинним поняттям. Синонімами терміна "об'єкт" є "реалія, сутність, річ". *Сутність ПО* є результатом абстрагування реального об'єкта шляхом виділення й фіксації набору його властивостей. На рис. 1.7 наведений один із підходів до класифікації об'єктів ПО.


Рисунок 1.7 – Класифікації об'єктів ПО

Прикладами сутностей (з погляду ІС) або об'єктів (з погляду зовнішнього світу) є окремий студент, група студентів, аудиторія, час занять, слова, числа, символи. Звичайно вважається, що бути об'єктом - це значить бути дискретним і помітним.

З об'єктами пов'язано дві проблеми: ідентифікація й адекватний опис. Для ідентифікації використовують ім'я. Використовується тільки вказівна функція імені. *Ім'я* - це

прямий спосіб ідентифікації об'єкта. До непрямих способів ідентифікації об'єкта відносять визначення об'єкта через його властивості (характеристики або ознаки).

Об'єкти взаємодіють між собою через свої властивості, що породжує ситуації. *Ситуації* - це взаємозв'язки, які виражають взаємини між об'єктами. Ситуації у предметній області (ПО) описуються за допомогою висловлювань про ПО з використанням виразами і обчисленнями предикатів, тобто формальної, математичної логіки.

Методи математичної логіки дозволяють формалізувати ці твердження й представити їх у вигляді, придатному для аналізу.

*Приклад. Розглянемо висловлювання: Студент Іванов А.А, народився у 1982 році. Воно виражає такі властивості об'єкта "Іванов А.А.":*

*у явному вигляді - рік народження;*

*у неявному - приналежність до студентів.*

*Перша властивість встановлює зв'язок між об'єктами "Іванов А.А." й "Рік народження", а друге - між об'єктами "Іванов А.А." й "Безліч студентів". Формалізація цього висловлювання подається як результат присвоювання значень змінним, які входять у предикати:*

*НАРОДИВСЯ (Іванов А.А., 1982)*

*Є СТУДЕНТОМ (Іванов А.А.)*

На рис. 1.8 наведений один із підходів до класифікації ситуацій у рамках ПО.

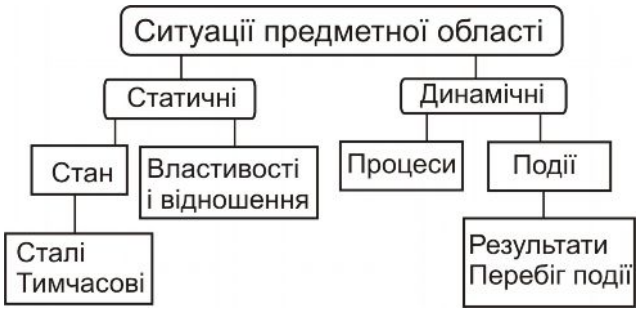



Рисунок 1.8 – Класифікація ситуацій ПО

Розрізняють статичні й динамічні ситуації. Прикладами статичних ситуацій є такі ситуації, як мати кольори, вік. Прикладами динамічних ситуацій є такі ситуації, як випекти хліб.

Наведена класифікація вводить у ПО два важливі аспекти - простір і час, до того ж час як і момент, і як інтервал. ПО існує у просторі і часі, тобто їй притаманні часові та просторові відношення і зв'язки. Необхідно розрізняти реальний час зовнішнього світу та його відображення у БД та у джерелах інформації. У БД взаємозв'язки залежні від часу і фіксуються тільки після реєстрації у БД. Таким чином, ПО у кожний певний момент часу являє собою відокремлену сукупність визначених об'єктів і ситуацій, яку називають станом ПО.

*Предметна область* - це цілеспрямована первинна трансформація картини зовнішнього світу у деяку картину, певна частина якої фіксується в ІС як алгоритмічна модель фрагмента дійсності.

### 1.2.2 Інформаційна модель ПО БД

Інформаційна модель даних призначена для подання семантики ПО у термінах суб'єктивних засобів опису - сутностей, атрибутів, ідентифікаторів сутностей, супертипів, підтипів і т.д.

Інформаційна модель ПО БД містить такі основні конструкції:

- діаграми "сутність-зв'язок" (Entity - Relationship Diagrams);

- визначення сутностей;
- унікальні ідентифікатори сутностей;
- визначення атрибутів сутностей;
- відношення між сутностями;
- супертипи й підтипи.

Елементи інформаційної моделі даних ПО є вхідними даними для вирішення завдання проектування БД - створення логічної моделі даних.

Предметом інформаційної моделі є абстрагування об'єктів або явищ реального світу у рамках ПО, у результаті якого виявляються сутності (entity) ПО. Як правило, вони позначаються іменником природної мови.

Сутність описується за допомогою даних, іменованих властивостями або атрибутами (attributes) сутності. Як правило, *атрибути* є визначеннями у висловленні про сутності й позначаються іменниками природної мови. Сутності вступають у зв'язки один з одним через свої атрибути. Кожна група атрибутів, що описує один реальний прояв сутності, являє собою екземпляр (instance) сутності. Іншими словами, екземпляри сутності - це реалізації сутності, що відрізняються один від одного й допускають однозначну ідентифікацію.

Одним із основних комп'ютерних засобів розпізнавання сутностей у базі даних є присвоєння сутностям ідентифікаторів (Entity identifier). Часто ідентифікатор сутності називають ключем. Завдання вибору ідентифікатора сутності є суб'єктивним завданням. Оскільки сутність визначається набором своїх атрибутів, то для кожної сутності доцільно виділити таку підмножину атрибутів, що однозначно ідентифікує дану сутність.

Завдання розробника БД - забезпечити при збереженні екземплярів сутності у БД наявність у кожного її нового екземпляра унікального ідентифікатора. *Унікальний ідентифікатор сутності* - це атрибут сутності, що дозволяє відрізняти одну сутність від іншої. Якщо сутність має кілька унікальних ідентифікаторів, так званих можливих ключів, то розробник повинен обрати первинний ключ сутності.

Розрізняють однозначні й багатозначні атрибути. Однозначними є атрибути, які в межах конкретного екземпляра сутності мають тільки одне значення. У протилежному випадку вони вважаються багатозначними.

Кожен атрибут сутності має домен (domain). *Домен* це вираз, який визначає значення, дозволені для даного атрибута. Іншими словами, домен - це область значень атрибута. Розробник БД повинен проконтролювати, щоб в інформаційній моделі ПО для кожного атрибута сутностей був визначений домен.

Сутності не існують окремо один від одного. Між ними є реальні відношення (Relationship), і вони повинні бути відбиті в інформаційній моделі ПО. При виділенні відношень акцент робиться на фіксацію зв'язків та їх характеристик. *Відношення* (зв'язок) являє собою з'єднання (взаємовідношення) між двома або більше сутностями. Кожен зв'язок реалізується через значення атрибутів сутностей. Звичайно зв'язок позначається дієсловом. Кожен зв'язок також повинен мати свій унікальний ідентифікатор зв'язку.

Розробник БД повинен проконтролювати, щоб зв'язок між сутностями здійснювався через точно зазначені атрибути, які будуть визначати унікальний ключ зв'язку. Вибір ключів сутностей - одне з найважливіших проектних рішень, що повинен бути зробити розробник при переході від інформаційної моделі ПО до логічної моделі БД.

Зв'язки характеризуються ступенем зв'язку й класом приналежності сутності до зв'язку. Ступінь (потужність) зв'язку - це відношення числа сутностей, що беруть участь в утворенні зв'язку. Існують такі типи: "один-до-одного", "один-до-множини", "множина-до-множини".

Типовою формою документування інформаційної моделі ПО є діаграми "сутність-зв'язок" (ER-діаграми). ER-діаграма дозволяє графічно подати всі елементи інформаційної моделі згідно простим, інтуїтивно зрозумілим, але чітко визначеним правилам - нотаціям. Далі ми будемо користуватися умовними позначками, прийнятими в методології інформаційного проектування.

Сутність на ER-діаграмі наводиться прямокутником з ім'ям у верхній частині. Будемо використовувати англійські слова для іменування елементів моделі.


Рисунок 1.9 – Подання сутності Person (персонал) на ER-діаграмі з атрибутами й унікальним ідентифікатором сутності

У прямокутнику перераховуються атрибути сутності, при цьому атрибути, що становлять унікальний ідентифікатор сутності, підкреслюються.

Домени призначаються аналітиками й фіксуються в спеціальному документі - словнику даних (Data Dictionary). На стадіях розроблення логічної й фізичної моделей реляційної БД домени уточнюються у сутностях на ER-діаграмі.

Розробник БД повинен ретельним образом вивчити домени кожного атрибута з погляду на можливість їх реалізації у СКБД.

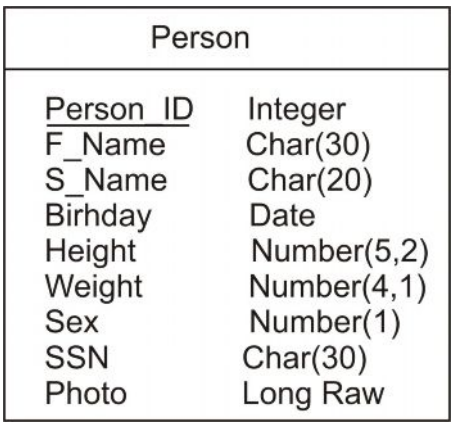



Рисунок 1.10 – Візуалізація визначення доменів атрибутів на ER-діаграмі при створенні фізичної моделі реляційної БД

Відношення (зв'язок) сутностей на ER-діаграмі зображується лінією, що з'єднує ці сутності. Ступінь зв'язку зображується за допомогою символу "пташина лапка", що вказує на те, що у зв'язку бере участь багато (N) екземплярів сутності, і одинарною горизонтальною рисою, що вказує на те, що у зв'язку бере участь один екземпляр сутності.

Відношення читається вздовж лінії або зліва направо, або справа наліво. На рис. 1.11 наведене таке відношення: кожна спеціальність зі створення повинна бути зареєстрована за певною фізичною особою (персоною), фізична особа може мати одну або більше спеціальностей зі створення.

### 1.2.3 Функціональна модель ПО БД

Другим ключовим моментом створення ІС з метою автоматизації інформаційних процесів організації є аналіз функціональної взаємодії об'єктів автоматизації. Аналітики наводять результати у вигляді функціональної моделі ПО БД. Склад функціональної моделі істотно залежить від

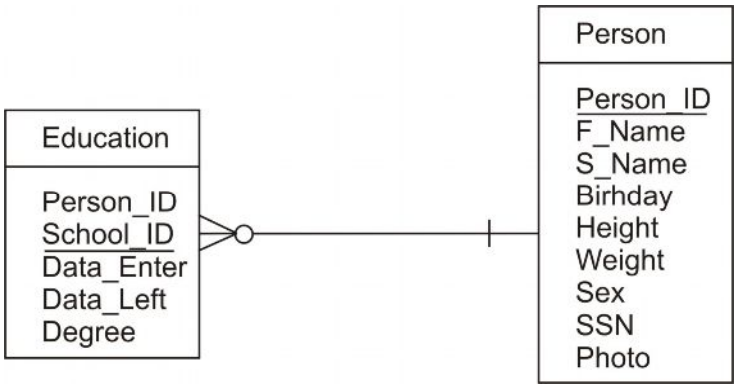



Рисунок 1.11 – Подання відношення між двома сутностями на ER-діаграмі

контексту конкретного ІТ-проекту і може бути представлений за допомогою досить широкого спектра документів у вигляді текстової й графічної інформації.

Функціональна модель призначена для опису процесів обробки даних у рамках виділеної ПО з різних точок зору.

Визначимо *функціональну модель* ПО БД як сукупність деяких моделей, призначених для опису процесів обробки інформації. Будемо називати ці моделі конструкціями функціональної моделі. Нижче наведений перелік основних конструкцій функціональної моделі, які необхідні для виконання проектування реляційних БД.

Моделі процесів:

- бізнес-модель процесів (ієрархія функцій системи);
- модель потоку даних.

Моделі станів:

- модель життєвого циклу сутності;
- набір специфікацій функцій системи (вимоги);
- опис функцій системи через сутності й атрибути;
- бізнес-правила, які реалізують функції.

Елементи інформаційної моделі ПО є вхідними даними для завдання створення логічної моделі даних. Елементи функціональної моделі ПО є вхідними даними для завдання проектування додатків БД і частково для завдання створення фізичної моделі БД.

### 1.2.4 Процес проектування БД

Значна частина проектів в області інформаційних технологій спрямована на розроблення й створення ІС, у рамках яких здійснюється обробка даних різної складності. Практично у всіх таких проектах вирішується завдання проектування БД певного типу.

В експлуатації БД повинна задовольняти набору вимог за рядом інтегрованих параметрів, таких як:

- функціональність й адаптованість;
- продуктивність обробки транзакцій;
- пропускна здатність;
- час реакції;

- безпека.

Такі параметри іноді перебувають у протиріччі один до одного. Так, високі вимоги до функціональності на даному конкретному устаткуванні можуть вступати у конфлікт із високими вимогами до продуктивності. Наприклад, звіти можуть генеруватися протягом декількох годин і знизити в цей час реакції користувачів, що працюють із системою в діалоговому режимі.

Таким чином, процес проектування БД полягає у досягненні компромісів між функціональними, інформаційними, апаратними, архітектурними й технологічними вимогами до БД і будується на інформованому прийнятті рішень за структурою БД.

*Проектування БД* - це пошук засобів задоволення функціональних вимог засобами наявної комп'ютерної технології з урахуванням заданих обмежень.

Як правило, IT-проекти зі створення БД містять у собі такі етапи:

1. Визначення стратегії побудови системи.
2. Аналіз вимог до БД.
3. Проектування БД.
4. Реалізація БД.
5. Тестування
6. Впровадження БД.

Етап проектування БД вважається одним із самих складних етапів створення БД, який не має явно вираженого початку й закінчення. Порівняно з аналізом вимог до БД або розробкою додатків, проектування БД, на думку багатьох провідних фахівців, є невдало структурованим завданням. Якщо всі етапи створення БД перекриваються один з одним у своїй послідовності, то етап проектування перекривається з усіма іншими етапами. Проектування починається з моменту прийняття стратегічних рішень і триває на етапах реалізації й тестування.

Процес проектування БД охоплює кілька основних сфер:

- проектування об'єктів БД (таблиці, подання, індекси, тригери, збережені процедури, функції, пакети) для подання даних ПО в БД;
- проектування інтерфейсу взаємодії з БД (форми, звіти й т.д.), тобто проектування додатків, які будуть супроводжувати дані в БД і реалізовувати питально-відповідні відношення на цих даних;
- проектування БД під конкретне обчислювальне середовище або інформаційну технологію (архітектура "клієнт-сервер", паралельні архітектури, розподілене обчислювальне середовище);
- проектування БД під призначення системи (інтелектуальний аналіз даних, OLAP, OLTP і т.д.).

***Типова бізнес-модель процесу проектування БД***

Процес проектування БД може бути поданий у вигляді моделі бізнес-процесів. Бізнес-модель процесу проектування дозволяє:

- відобразити суб'єктивну думку розробника БД на процес проектування конкретної БД;
- врахувати особливості IT-проекту, у рамках якого проектується БД;
- досить швидко скласти план проектування конкретної БД;
- прорахувати тривалість проектних робіт (створити тимчасову модель проектування).

Розглянемо типову бізнес-модель процесу проектування БД. На рис. 1.12 наведена контекстна діаграма процесу проектування БД.

Як бачимо з рисунка, на вхід процесу проектування БД подаються:

- інформаційна модель ПО БД: діаграми "сутність-зв'язок" (ER-діаграми);

- функціональна модель ПО БД: бізнес-модель процесів, діаграми потоку даних (DF-діаграми), діаграми станів, - діаграми життєвих циклів сутностей, специфікації на системи (вимоги), бізнес-правила;
- загальносистемні вимоги й обмеження;
- завдання зворотного впливу.

На виході процесу проектування БД формуються такі результати:

- фізична модель БД, що може бути перетворена у скрипт для створення БД;
- фізична БД;
- специфікація модулів додатків БД;
- план тестування БД.

Продовжуючи функціональну декомпозицію процесу проектування БД, приходимо до діаграми декомпозиції процесу проектування БД першого рівня, яка


Рисунок 1.12 – Контекстна діаграма процесу проектування БД

відбиває основні найбільш великі професійні завдання (етапи) проектування БД (рис. 1.13).

Такими завданнями (етапами) є:

- збір й аналіз вхідних даних – це початковий етап проектування, на якому здійснюється збір і контроль якості результатів аналізу ПО БД, готується план проектування БД;
- створення логічної моделі БД – це етап, на якому на підставі інформаційної моделі ПО БД створюється логічна структура БД, незалежна від її реалізації;
- створення фізичної моделі БД: внутрішня схема – це етап, на якому на підставі логічної моделі БД створюється фізична структура БД, залежна від її реалізації. На цьому етапі виконується перетворення відношення логічної моделі реляційної БД у команди

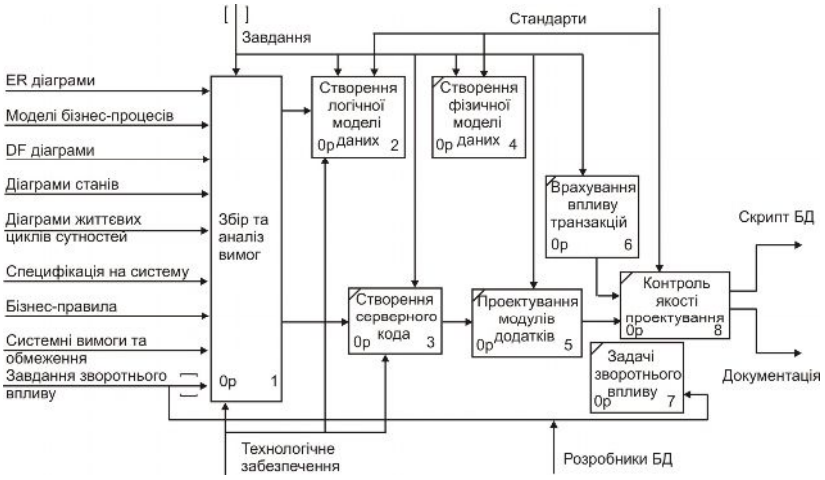



Рисунок 1.13 – Діаграма декомпозиція процесу проектування БД: перший рівень

створення об'єктів фізичної БД, у результаті чого створюється так звана внутрішня схема БД. Додатково може бути створена так звана зовнішня схема БД, останнє відбиває точку зору користувачів на дані в БД;

- створення фізичної моделі БД: облік впливу транзакцій – це етап, на якому аналізуються можливі транзакції системи, виконується при потребі денормалізація відношення для забезпечення більш високої продуктивності БД;
- створення серверного коду – це етап, на якому на підставі функціональної моделі ПО БД створюється серверний код БД у вигляді тригерів, збережених процедур і пакетів. Ці модулі створюються розробником БД і виконуються сервером;
- проектування модулів додатків БД – це етап, на якому створюються специфікації модулів додатків, розробляються стратегії тестування БД і додатків, створюється план тестування додатків БД і готуються тестові дані;
- контроль якості проектування БД полягає в перевірці якості результатів проектування на кожному його етапі;
- облік завдань зворотного впливу полягає у настройці деяких транзакцій до БД і локальному перепроектуванні БД відповідно до вимог, що надходять з інших етапів створення БД.

Коротко розглянемо бізнес-моделі другого рівня.

***Бізнес-модель процесу проектування БД: збір й аналіз вхідних даних***

На рис. 1.14 наведена діаграма декомпозиції процесу проектування БД другого рівня, що відбиває основні завдання етапу збору й аналізу вхідних даних.

Такими завданнями є:

- збір документації з результатами аналізу ПО БД у вигляді діаграм, специфікацій і вимог;
- контроль якості результатів аналізу ПО БД;
- систематизація вимог і специфікацій замовника до БД;
- підготовка плану проектування БД.


Рисунок. 1.14 – Діаграма декомпозиції процесу проектування БД: збір й аналіз вхідних даних

У ході контролю якості основними моментами діяльності є контроль ER-діаграм і контроль діаграм функціональної моделі ПО. На підставі ER-діаграм створюється логічна модель реляційної БД; на підставі діаграм функціональної моделі розробляється серверний код і проектуються модулі додатків БД.

Систематизація вимог замовника до БД виконується з метою їх адекватного розподілу по етапах проектування БД. Важливим результатом систематизації є висновок про достатність вимог і можливість реалізації БД. Аналіз вимог на можливість реалізації БД у рамках конкретного IT-проекту є основою для ухвалення рішення менеджером проекту про можливості реалізації в цілому.

Дійсна бізнес-модель процесу проектування БД являє собою досить простий типовий приклад бізнесу-моделі проектування. У загальному випадку зміст бізнес-моделі проектування залежить від багатьох факторів: особистості менеджера й складу команди проекту, обсягу проекту, проектних ризиків і т.д.

***Бізнес-модель процесу проектування реляційної БД***: створення логічної моделі БД (рис. 1.15). Основною метою етапу створення логічної моделі БД є перетворення інформаційної моделі ПО БД у логічну модель реляційної БД. Створення логічної моделі БД припускає рішення таких основних завдань і виконання операцій у рамках таких завдань:

- нормалізація сутностей ПО: одержати список атрибутів сутності; визначити функціональні залежності (ФЗ) у сутності; визначити детермінанти сутності; визначити можливі ключі відношення, зокрема, розглянувши унікальний ідентифікатор сутності; виконати нормалізацію сутності (перетворити сутність у відношення); для отриманого відношення призначити первинні ключі; сформувати список кандидатів на зовнішні ключі, якщо необхідно; сформувати бізнес-правила підтримки цілісності сутності, якщо необхідно;
- нормалізація відношення логічної моделі БД;
- визначити ступінь зв'язку сутностей;
- визначити клас приналежності сутності до зв'язку: нормалізувати відношення (дозволити зв'язку);
- призначити первинні ключі єднальних відношень, виходячи з унікального ідентифікатора зв'язку й процедури міграції ключів при нормалізації; визначити атрибути єднальних відношень, якщо необхідно; сформувати бізнес-правила підтримки цілісності зв'язків;

- перевірка правильності логічної моделі реляційної БД: перевірка відношень на відповідність нормальній формі Бойса-Кодда; перевірка відношень на властивості з'єднання без втрат і збереження функціональних залежностей; запобігання втрати даних;
- шляхом міграції первинних ключів відношення й призначення зовнішніх ключів; перевірка на відсутність незамкнутих зв'язків; перевірка на відсутність одиночних відношень;
- формулювання частини вихідних даних для вирішення завдання керування посилальною цілісністю;
- документування логічної моделі реляційної БД;
- ухвалення рішення про можливість реалізації побудованої логічної моделі реляційної БД;

ухвалення рішення про розроблення фізичної моделі реляційної БД.

Результатом проектування логічної моделі БД є нормалізована схема відношень БД. Відзначимо, що в ході виконання етапу створення логічної моделі БД можуть бути створені нові об'єкти БД, не передбачені в інформаційній моделі ПО, наприклад, єднальна сутність при нормалізації відношень зі ступенем зв'язку "множина-до-множини".

Подані завдання становлять мінімально необхідний набір завдань, що дозволяють спроектувати логічну модель БД, і можуть розглядатися як один з можливих способів організації робіт у цій області.

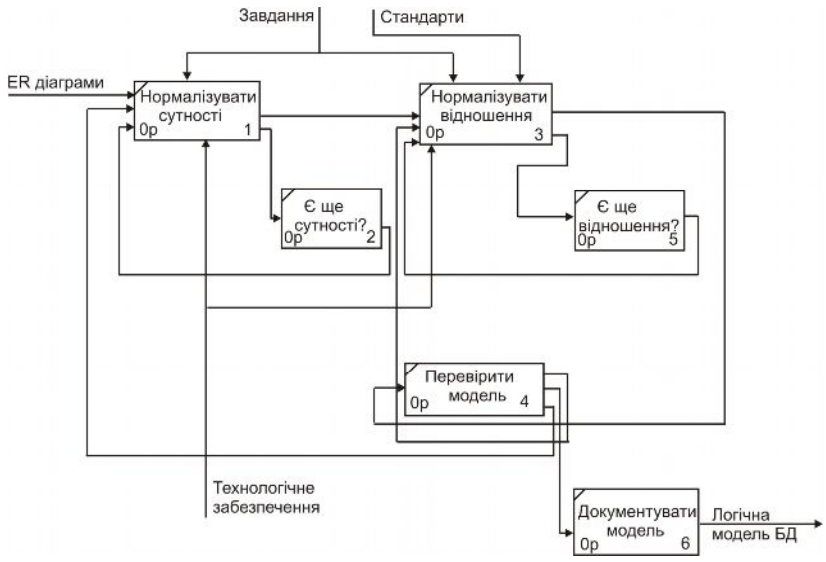



Рисунок 1.15 – Бізнес-модель процесу створення логічної моделі БД

***Бізнес-модель етапу проектування - створення фізичної моделі реляційної БД***

Основна мета вирішення цього завдання: перетворити логічну модель реляційної БД у послідовність команд SQL для створення об'єктів реляційної БД. Таким чином, розробник БД відображає відношення логічної моделі реляційної БД (сутності ПО, подані в нормалізованій формі на ER-діаграмах) у таблиці й індекси реляційної БД.

Це завдання включає виконання ряду обов'язкових послідовних процедур:

- створення базових таблиць. Вони представляють основні блоки зберігання даних і виводяться із сутностей логічної моделі даних. При створенні кожної таблиці розробник повинен розглянути й урахувати ряд факторів: визначити список колонок у таблиці (колонки виводяться з атрибутів сутності логічної моделі даних); визначити типи даних для кожної колонки (типи даних колонок або задані специфікацією домену атрибута логічної моделі, або визначаються розробником самостійно); визначити ім'я таблиці (воно може бути виведене з імені сутності логічної моделі БД або задано розробником самостійно. Бажано в цей момент визначити власника таблиці - користувача, що буде мати усі права доступу на таблицю, а

також потенційних користувачів таблиці); визначити ряд параметрів, пов'язаних із характером зберігання таблиці у фізичній БД; визначити обмеження на значення колонок, виходячи з ряду бізнес-правил;

- створення єднальних таблиць, необхідних для дозволу відношення "множина-до-множини", якщо вони мають місце в логічній моделі БД. У рамках ER-діаграм це відношення може бути вже дозволено. Тоді мова йтиме тільки про його реалізації в командах SQL;
- ухвалити рішення щодо засобів підтримки посилальної цілісності в БД. Якщо буде вирішено підтримувати посилальну цілісність на рівні команд SQL, то розробити специфікацію обмеження посилальної цілісності. Це завдання вирішується в чотири етапи: ідентифікувати первинні ключі кожної таблиці; побудувати індекси первинного ключа; визначити зовнішні ключі в дочірніх таблицях, якщо необхідно; побудувати команди SQL, які ідентифікують зовнішні ключі в дочірніх таблицях і правила підтримки посилальної цілісності; якщо необхідно, побудувати подання зовнішньої схеми БД.

У результаті вирішення даного завдання робиться важливий вивід про правильність отриманої першої ітерації фізичної моделі БД, здійснюється документування фізичної моделі даних у вигляді скріпту, береться рішення про характер подальшої розробки фізичної моделі даних. Зі вказаної в попередніх розділах лекції зрозумілий такий алгоритм дій:

Створення об'єктів для зберігання даних:

Створення таблиць

Ідентифікування таблиці

Визначення типів даних колонок

Визначення первинного ключа

Додавання обмежень

Створення таблиць для взаємозв'язку "множина-до-множини"

Створення індексів

Створення подань

Створення інших об'єктів БД

Перевірка коректності створеної фізичної моделі

На рис. 1.16 нижче подана модель бізнес-процесу першої ітерації фізичної моделі БД.

Головна мета етапу - створити послідовність команд SQL для створення об'єктів зберігання даних. Також можна створювати інші об'єкти, такі як синоніми, подання й індекси. Можна ухвалити рішення щодо підтримки посилальної цілісності БД програмними механізмами СКБД і створити відповідний набір команд SQL.

***Бізнес-модель етапу проектування - створення фізичної моделі реляційної БД: облік впливу транзакцій***. Вирішуючи професійне завдання створення фізичної моделі даних - облік впливу транзакцій, - розробник реляційної прагне створити таку фізичну модель даних, яка б, на його думку, давала найбільшу продуктивність обробки запитів БД. На практиці, особливо при створенні й розробці нових БД, таке завдання навряд чи може бути вирішене повністю. Ясно, що для його вирішення необхідно мати перелік всіх запитів до БД, їхній частоті й обсязі вибірок по кожному, що в принципі неможливо. Тому розробники БД на основі аналізу вихідної документації й опитувань потенційних користувачів

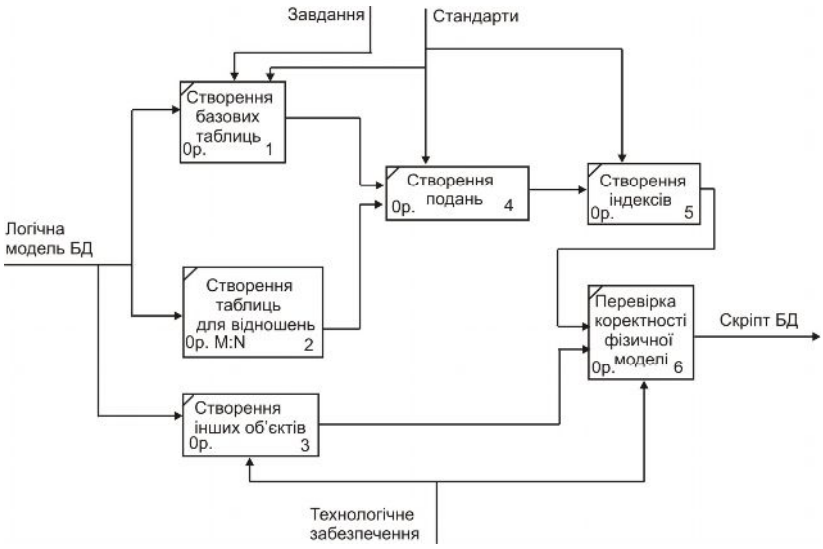



Рисунок 1.16 – Декомпозиція етапу проектування - створення першої ітерації фізичної моделі БД : внутрішня схема

намагаються систематизувати транзакції до БД, оцінити кардинальність таблиць у цілому й окремих колонках зокрема. На основі таких оцінок розробник БД намагається визначити критичні транзакції й налаштувати структури таблиць, задіяних у таких транзакціях, на досягнення максимальної продуктивності. При цьому він висуває гіпотези про застосовність того або іншого засобу підвищення продуктивності обробки запитів й перевіряє їх. Далі ухвалюється рішення щодо застосування найбільш підходящого.

Слід розуміти, що завдання забезпечення високої продуктивності БД - це завдання, яке постійно вирішує адміністратор БД у процесі її експлуатації. На цьому етапі проектування БД розробник, у міру можливості, готовить успішне вирішення цього завдання. Цей етап є дуже відповідальним у фізичному проектуванні БД, тому варто дотримувати при вирішенні цього завдання розумного прагматизму і документувати свої рішення. Повинне діяти емпіричне правило: якщо розробник БД не має досить даних для надійного вирішення завдання підвищення продуктивності БД, то рішення цього завдання повинне бути передане адміністраторові БД.

На цьому етапі проектування фізичної моделі розробник реляційної БД:

- виходячи з вимог до характеру обробки даних, визначає тип додатка БД;
- за наявними вимогами й описами виконує систематизацію й опис за можливістю всіх транзакцій;
- відштовхуючись від вихідної документації, визначає можливі розміри таблиць, а якщо це неможливо, робить припущення про їхній можливий розмір;
- виходячи з фактичних розмірів таблиць і вимог до продуктивності виконання транзакцій, визначає критичні транзакції;
- для кожної критичної транзакції необхідно оцінити кардинальність кожної колонки, задіяної у транзакції й, за можливістю, кардинальність вибірки;
- далі, розглядаючи в першу чергу критичні транзакції й таблиці, які в них беруть участь, розробник БД приймає суб'єктивні рішення по зміні структури таблиць внутрішньої схеми БД, виходячи з тих механізмів, які йому надає конкретна СКБД;
- по завершенні зміни структур таблиць розробник БД документує ці зміни, приводячи обґрунтування своїх рішень для адміністратора БД.

У результаті розробник БД створює фізичну модель БД, що враховує характер обробки даних у БД, виражений через облік впливу транзакцій.

Побудова бізнес-моделі етапу проектування фізичної моделі реляційної БД: облік впливу транзакцій проходить у кілька таких етапів (рис. 1.17):

Визначення основного типу додатка БД
Документування й опис транзакцій
Визначення критичних транзакцій
Для кожної критичної транзакції:
Визначення таблиць транзакції
Визначення способу підвищення продуктивності
Денормализація таблиці?
Розбиття таблиці?
Секціонування таблиці?
Кластерізація таблиці?
Побудова додаткових індексів?
Зміна структури внутрішньої схеми БД
Документування змін
Для кожної таблиці БД
Вибір індексів
Визначення транзакцій таблиці
Визначення кардинальності таблиць
Визначення кардинальності колонок
Визначення індексів
Зміна внутрішньої схеми

***Короткий розгляд завдань створення серверного коду й підготовки скріпту***

Професійне завдання проектування БД - розроблення серверного коду БД - виникають, як правило, в обчислювальному середовищі з багатьма користувачами. У цих системах користувачі спільно використовують обчислювальні ресурси, зокрема ресурси дискової пам'яті й оперативної пам'яті процесора. Обчислювальні ресурси можуть бути сконцентровані в одному місці (централізовані обчислення) або бути розосередженими в різних вузлах, об'єднаних у комп'ютерну мережу (розподілені обчислення). СКБД у кожному разі покликана координувати й здійснювати доступ користувачів до баз даних та їхніх об'єктів.

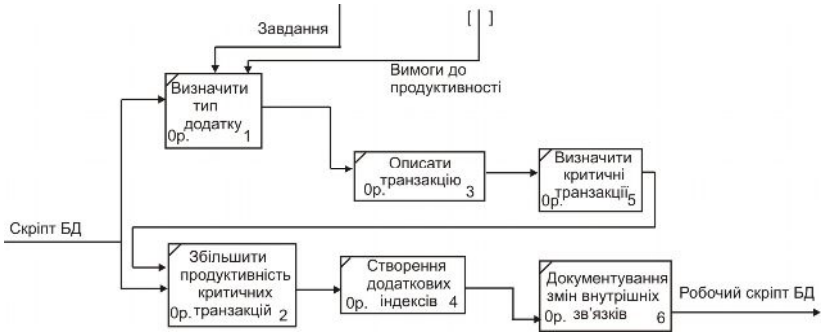



Рисунок 1.17 – Декомпозиція етапу проектування - створення першої ітерації фізичної моделі БД: внутрішня схема

Більшість сучасних СКБД підтримують концепцію клієнт-серверної технології для розподілених обчислень. Це означає, що існують концентратори обчислень (названі серверами), на яких виконується найбільший обсяг обчислень із даними (сервери БД), і машини користувачів (клієнти), на яких виконуються додатки користувачів. Додатки формують запити у формі команд SQL до БД, відправляють їхнім серверам БД, одержують запитувані дані й обробляють їх.

У клієнт-серверному обчислювальному середовищі додаток може взаємодіяти із сервером БД за іншою схемою: коли додаток відправляє запит, цей запит обробляється на

сервері, а додатку вертається готовий результат. Робота додатка за другою схемою ґрунтується на використанні так званого серверного коду (server-side code) - будь-якого коду, який виконується комп'ютером, на якому встановлена СКБД. Ядро СКБД виконує цей код у БД і повертає додатку тільки результат. Наприклад, це може бути трішки колонок рядка або обчислене значення.

Використання серверного коду може значно скоротити обсяг мережного трафіку й тим самим збільшити продуктивність БД в цілому. Однак СКБД повинна мати вбудовані засоби для розпізнавання й обробки такого коду. Багато фірм-виробників промислових СКБД пропонують процедурні розширення SQL, за допомогою яких можна виконувати порядкову обробку даних, використовувати цикли, складні обчислення й операції керування даними.

Таким чином, розробка серверного коду зводиться до рішення таких підзадач:

- ухвалення рішення й створення збережених процедур;
- ухвалення рішення й створення функцій;
- ухвалення рішення й створення пакетів;
- ухвалення рішення й створення тригерів.

Завдання проектування БД - підготовка інсталяційного скріпту для створення БД - деякою мірою завершальна для самостійної роботи розробника БД. Такий скріпт - це один із головних результатів його роботи. Розробник БД, виконавши попередні завдання, фактично виконав свою основну роботу над створенням скріпту для БД.

Завдання створення скріпту БД складається з вирішення великих підзадач:

- створення користувачів, їх ідентифікація й призначення їм привілеїв;
- прив'язка розроблених об'єктів реляційної БД до параметрів фізичного зберігання БД за допомогою створення спеціальних об'єктів БД;
- створення інсталяційного скріпту;
- документування БД.

## 1.3 Реляційна модель даних

### 1.3.1 Поняття відношення

Широкому поширенню і популярністю реляційна модель даних завдячує двом істотних перевагам:

1) однорідністю подання даних у моделі, що обумовлює простоту сприйняття її конструкцій користувачами БД;

2) наявністю розвиненої математичної теорії реляційних БД, що обумовлює коректність її застосування.

В основі реляційної моделі даних лежить поняття відношення, яке задається переліком своїх елементів і перерахуванням їх значень. Розглянемо приклад на рис. 1.18. На ньому наведений розклад руху автобусів по маршруту "Москва - Черноголовка - Москва". Бачимо певну структуру. Кожен включений у розклад рейс має свій номер, час відправлення й час у дорозі. Розклад може бути подано таблицею. Заголовки колонок таблиці звуться атрибутами. Перелік їх імен носить назви схеми відношення. Кожен атрибут визначає тип даних, що разом з областю його значень називається доменом. Вся таблиця цілком називається відношенням, а кожен рядок таблиці зветься кортежем відношення. Таким чином, відношення можна подати у вигляді двовимірної таблиці.

Підходи до визначення поняття відношення можуть бути різними. Математично відношення може бути визначене як безліч кортежів, що є підмножиною декартового добутку фіксованого числа областей (доменів). У результаті одержуємо, що у кожному кортежі повинне бути однакове число компонентів (атрибутів) і значення кожного з них вибирається з деякого певного домену.

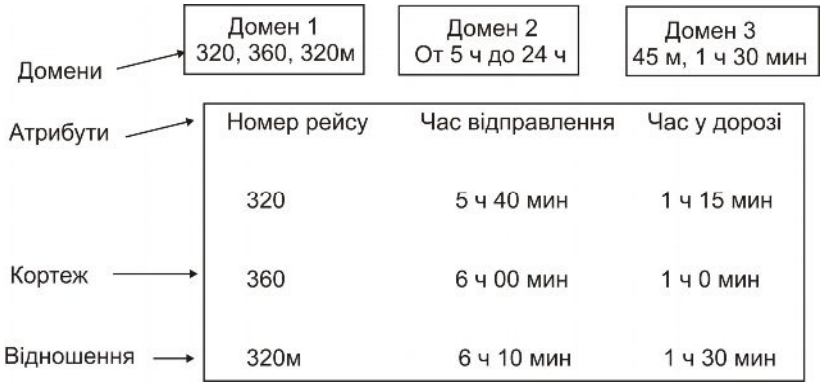



Рисунок 1.18 – Розклад руху автобусів як відношення

Таблична форма подання відношення була введена з метою популяризації моделі серед непідготовлених користувачів БД. Трактування реляційної теорії на рівні таблиць приховують ряд визначень, важливих для розуміння як теорії реляційних БД, так і мови маніпулювання даними.

По-перше, атрибути різних відношень можуть бути визначені на одному домені, так само як й атрибути одного відношення. Це дуже важлива обставина, що дозволяє встановлювати зв'язки за значенням між відношеннями. По-друге, множина математично по своєму визначенню не може мати співпадаючих елементів, і, отже, кортежі у відношенні можна розрізнити лише за значенням їх компонентів. Це теж є дуже важливим для моделі: ніякі два кортежі не можуть мати повністю співпадаючих компонентів. *Таким чином, у реляційній моделі повністю виключається дублювання даних про сутності реального світу*. По-третє, відзначимо, що схема відношення також є множина, що дозволяє працювати з ними за допомогою теоретико-множинних операцій. Це є важливим моментом для побудови теорії проектування реляційних схем БД.

Існує певне розходження між математичним визначенням відношення й дійсне збереженим відношенням у пам'яті комп'ютера. За визначенням, відношення не може мати два ідентичних кортежі. Однак СКБД, що підтримують реляційну модель даних, зберігають відношення у файлах операційної системи комп'ютера. Розміщення відношень у файлах операційної системи допускає зберігання ідентичних кортежів. Якщо не використовується спеціальна техніка (контроль цілісності за первинним ключем), то звичайно більшість промислових СКБД допускають зберігання двох ідентичних кортежів у БД.

*Ключем* або *ключовим полем* називається унікальне значення, що дозволяє тим чи іншим способом ідентифікувати сутність або частину сутності ПО, тобто ключ - це значення деякого атрибута або атрибутів у кортежі відношення, що представляє екземпляр сутності у реляційній моделі даних.

Прийнято розрізняти первинні ключі й часткові ключі. Математично *первинним ключем* відношення є підмножина звуження декартового добутку, що дозволяє однозначно ідентифікувати кортеж. Якщо первинний ключ містить кілька атрибутів, то він називається складеним ключем, у протилежному випадку - атомарним. *Частковим_ключем* називається атрибут складеного ключа, якщо він однозначно визначає сукупність неключових атрибутів відношення. Атрибут кортежу, що є первинним ключем іншого відношення, називається *зовнішнім* (іноді стороннім) *ключем*. Із визначення відношення випливає таки важлива властивість реляційної моделі даних: кожне відношення повинне мати первинний ключ. Зазначимо, що ключ у контексті моделі ПО БД завжди відображає той або інший ступінь зв'язку між атрибутами сутностей ПО, тобто семантично ключ є засіб моделювання зв'язків у моделі.

*Приклад – Розглянемо речення "Громадянин Іванов проживав у місті Москві 10 років". Можливими атрибутами у відношенні Місце_проживання є прізвище громадянина,*

*назва міста проживання й час проживання. Прізвище громадянина може виступати як первинний ключ цього відношення, тому що особистість однозначно визначає час її проживання в конкретному місті. Таким чином, щодо цього моделюється зв'язок "проживав" між атрибутами "прізвище" й "місто".*

Відношення у реляційній моделі даних, як правило, представляються за допомогою функціональної форми запису (тому що ми записуємо функції декількох змінних у математичному аналізі), при цьому атрибути первинного ключа підкреслюються:

ІМ'Я_ВІДНОШЕННЯ (Атрибути первинного ключа, неключові атрибути).

*Приклад. Подання зв'язку відношенням. Представимо зв'язок між особистістю й місцем її проживання через відношення*

*ПРОЖИВАЄ (Кл. особистість, Кл. населений_пункт, час)*

*Опис особистості:*

*ОСОБИСТІСТЬ (Кл. особистість, П.І.П/б, вік, стать)*

*Опис населеного пункту:*

*НАСЕЛЕНИЙ_ПУНКТ (Кл.населений_пункт, географія, населення)*

Однак найбільшого поширення одержало подання відношень у вигляді графічних діаграм, наприклад, ER-діаграм, про які ми говорили раніше. Перевагами такого подання є наочність діаграм і можливість їх побудови у ряді CASE-засобів проектування БД.

У підсумку сформулюємо основні властивості реляційної моделі даних, які випливають із поняття відношення як множини:

- всі кортежі одного відношення повинні мати ту саму кількість атрибутів;
- значення кожного з атрибутів повинне належати деякому певному домену;
- кожне відношення повинне мати первинний ключ;
- ніякі два кортежі не можуть мати повністю співпадаючих наборів значень;
- кожне значення атрибутів повинне бути атомарним, тобто не повинне мати внутрішньої структури й містити як компонент інше відношення;
- реляційна модель даних повинна бути несуперечливою, зокрема повинно виконуватися: 1) принцип посилальної цілісності - зв'язки між відношеннями повинні бути замкнутими, 2) значення колонок повинні належати тому самому визначеному для них домену;
- порядок проходження кортежів у відношенні не має значення. Порядок є більшою мірою властивістю зберігання даних, ніж властивістю безпосередньо самої реляційної моделі даних.

### 1.3.2 Поняття функціональної залежності в даних

На стадії логічного проектування реляційної БД розробник визначає й вибудовує схеми відношення у рамках деякої ПО, а саме - представляє сутності, групує їх атрибути, виявляє основні зв'язки між сутностями. Так, у самому загальному змісті проектування реляційної БД полягає в обґрунтованому виборі конкретних схем відношення з безлічі різних альтернативних варіантів схем.

На практиці побудова логічної моделі БД, незалежно від моделі даних, виконується з урахуванням двох основних вимог: виключити надмірність і максимально підвищити надійність даних. Ці вимоги випливають із вимоги колективного використання даних групою користувачів.

Тому будь-яке апріорне знання про обмеження ПО, що накладають на взаємозв'язки між даними й значення даних, і знання про їх властивості і взаємини між ними може зіграти певну роль у дотриманні зазначених вище вимог. Формалізація таких апріорних знань про властивості даних ПО БД знайшла своє відображення у концепції функціональної залежності даних, тобто обмежень на можливі взаємозв'язки між даними, які можуть бути поточними значеннями схеми відношень.

Кортежі відношення можуть представляти екземпляри сутності ПО або фіксувати їх взаємозв'язок. Але навіть якщо ці кортежі відповідають схемі відношень й обрані з припустимих доменів, не кожен з них може бути поточним значенням деякого відношення. Наприклад, вік людини рідко буває більше 120 років, або той самий пілот не може одночасно виконувати два різних рейси. Такі обмеження семантики домену практично не впливають на вибір тієї або іншої схеми відношень. Вони являють собою обмеження на типи даних.

Оскільки функціональну залежність можна задати у вигляді таблиці, а таблиця є форма подання відношень, то стає очевидний зв'язок між функціональною залежністю і відношенням. Відношення може задавати функціональну залежність. Це твердження є першою конструктивною ідеєю, яка покладена в основу теорії проектування реляційних БД.

*Приклад. Поняття функціональної залежності*

*Проілюструємо поняття функціональної залежності на прикладі графіка польотів аеропорту.*

*ГРАФІК_ПОЛЬОТІВ (Пілот, Рейс, Дата_вильоту, Час_вильоту)*

| | | | |
|---|---|---|---|
| *Іванов* | *100* | *8.07* | *10:20* |
| *Іванов* | *102* | *9.07* | *13:30* |
| *Ісаєв* | *90* | *7.07* | *6:00* |
| *Ісаєв* | *103* | *10.07* | *19:30* |
| *Петров* | *100* | *12.07* | *10:20* |
| *Петров* | *102* | *11.07* | *13:30* |
| *Фролов* | *90* | *8.07* | *6:00* |
| *Фролов* | *90* | *12.07* | *6:00* |

*Відомо: кожному рейсу відповідає певний час вильоту; для кожного пілота, дати й часу вильоту можливий тільки один рейс; на певний день і рейс призначається певний пілот.*

*Отже: "Час_вильоту" функціонально залежить від {"Рейс"}; "Рейс" функціонально залежить від {"Пілот", "Дата_вильоту", "Час_вильоту"}; "Пілот" функціонально залежний від {"Рейс", "Дата_вильоту"}.*

Важливим завданням при виявленні функціональних залежностей на атрибутах відношень, що за визначенням є множиною, необхідно з'ясувати, який з атрибутів виступає як аргумент, а який - як значення функціональна залежність. Найбільш підходящими кандидатами в аргументи функціональної залежності є можливі ключі, тому що кортежі представляють екземпляри сутності, які ідентифікуються значеннями атрибутів свого ключа.

### 1.3.3 Нормальні форми відношень. Створення логічної моделі реляційної БД

Під реляційною БД прийнято розуміти сукупність екземплярів кінцевих відношень. Сукупність схем відношень утворює схему реляційної БД.

Схема реляційної БД є логічною моделлю реляційної БД. На основі інформаційної моделі у процесі проектування створюються логічна й фізична моделі даних. Інформаційна модель даних відбиває потреби системи в даних і зв'язку між даними з погляду їх споживачів - користувачів; логічна модель даних є незалежним логічним поданням даних; фізична модель даних містить визначення всіх реалізованих об'єктів у конкретній БД для конкретної СКБД.

Установлення функціональної залежності й одержання найкращого з погляду мінімальності подання множини функціональних залежностей дозволять побудувати найбільш оптимальний варіант БД, що забезпечує надійність зберігання й обробки даних на основі методів еквівалентних перетворень схем відношень реляційної БД. Процес вирішення такого завдання називається нормалізацією відношень інформаційної моделі ПО й полягає у перетворенні її об'єктів у логічні таблиці БД. Основні вимоги наведені нижче:

- первинні ключі відношень повинні бути мінімальними;
- число відношень БД повинне по можливості давати найменшу надмірність даних – вимога надійності даних;

- число відношень БД не повинне приводити до втрати продуктивності системи;
- дані не повинні бути суперечливими, тобто при виконанні операцій включення, видалення й відновлення даних їх потенційна суперечливість повинна бути зведена до мінімуму;
- схема відношень БД повинна бути стійкою, здатною адаптуватися до змін при її розширенні додатковими атрибутами – вимога гнучкості структури БД;
- розкид часу реакції на різні запити до БД не повинен бути великим;
- дані повинні правильно відбивати стан ПО БД у кожен конкретний момент часу – вимога актуальності даних;

Створення системи, що одночасно задовольняє всім вищезгаданим вимогам, являє собою складну оптимізаційну задачу, що часом не має однозначного вирішення.

Теорія функціональних залежностей дозволяє встановити певні вимоги до схем відношень у реляційній БД. Ці вимоги формулюються у термінах властивостей відношень і називаються *нормальними формами схем відношень*. Кожна нормальна форма відношень пов'язана з певним класом функціональної залежності, які представлені у відношеннях. Одним з очевидних засобів усунення потенційної суперечливості даних у відношеннях логічної моделі реляційної БД є їх розбиття на двоє або більше відношень, у кожному з яких є присутньою тільки одна функціональна залежність.

Процес усунення потенційної суперечливості й надмірності даних у відношеннях реляційної БД називається *нормалізацією вихідних схем відношень*. Нормалізація відношень полягає у виконанні декомпозиції відношень, що перебувають у попередній нормальній формі, на двоє або більше відношень, які задовольняють вимогам наступної нормальної форми.

У теорії реляційних БД звичайно виділяється така послідовність нормальних форм: перша нормальна форма (1NF); друга нормальна форма (2NF); третя нормальна форма (3NF); нормальна форма Бойса-Кодда (BCNF); четверта нормальна форма (4NF); п'ята нормальна форма, або нормальна форма проекції-з'єднання (5NF або PJ/NF).

Основні властивості нормальних форм полягають у такому: кожна наступна нормальна форма у деякому змісті краще попередньої нормальної форми; при переході до наступної нормальної форми властивості попередніх нормальних форм зберігаються.

***Перша нормальна форма*** – відношення перебуває у 1NF, якщо всі атрибути відношення є простими (вимогу атомарності атрибутів), тобто не мають компонентів. Іншими словами, домен атрибута повинен складатися з неподільних значень і не може містити в собі безліч значень із більше елементарних доменів.

Нехай є змінне відношення: Employer_Project_Task {Em_Number, Em_Degrees, Em_Pay, Pr_Number, Em_Task}. Атрибути містять відповідно дані про номер справи, розряд та заробітну платню службовця, номер проекту й про завдання, що виконує службовець у даному проекті. Припустимо, що розряд службовця визначає розмір його заробітної плати й що кожен службовець може брати участь у декількох проектах за умови виконання тільки одного завдання. Тоді очевидно, що єдино можливим ключем відношення є складений атрибут {Em_Number, Pr_Number }. Діаграма мінімальної множини ER показана на рис. 1.19. У наведеному відношенні деякі функціональні залежності атрибутів від можливого ключа не є мінімальними. Це призводить до так званих аномалій відновлення. Під *аномаліями відновлення* розуміються труднощі, з якими зустрічаються при виконанні операцій додавання кортежів у відношення (INSERT), видалення кортежів (DELETE) і модифікації кортежів (UPDATE).

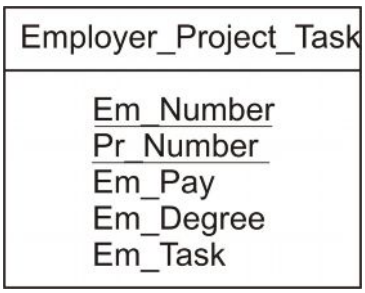



Рисунок 1.19 – ER-діаграма відношення Employer_Project_Task

Стосовно нашого прикладу:

- додавання кортежів – ми не можемо доповнити відношення Employer_Project_Task даними про службовця, який ще не бере участь у жодному проекті (Em_Number є частиною первинного ключа й не може містити невизначених значень). Тим часом часто буває, що спочатку службовця беруть на роботу, встановлюють його розряд і розмір заробітної плати, а лише потім призначають для нього проект;
- видалення кортежів – ми не можемо зберегти у відношенні Employer_Project_Task дані про службовця, який завершив участь у своєму останньому проекті (з тієї причини, що значення атрибута Pr_Number для цього службовця стає невизначеним);
- модифікація кортежів – щоб змінити розряд службовця, ми будемо змушені модифікувати всі кортежі з відповідним значенням атрибута Em_Number. У іншому випадку буде порушений природний зв'язок Em_Number → Em_Degrees (в одного службовця є тільки один розряд).

Для подолання цих труднощів можна зробити декомпозицію змінного відношення Employer_Project_Task на два змінні відношення – Employer {Em_Number, Em_Degrees, Em_Pay} і Employer_Project_Task {Em_Number, Pr_Number, Em_Task}. На рис. 1.20 показані діаграми ER цих відношень. Тепер ми можемо легко впоратися з операціями відновлення.

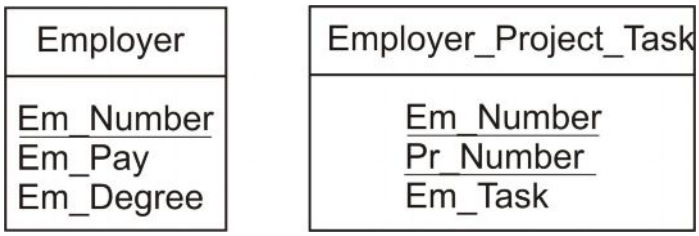



Рисунок 1.20 – ER-діаграми у змінних відношеннях Employer і Employer_Project_Task

***Друга нормальна форма***

Будемо вважати атрибут відношення ключовим, якщо він є елементом якого-небудь ключа відношення. В іншому випадку атрибут буде вважатися неключовим. Відношення перебуває у 2NF, якщо воно перебуває у 1NF, і всі неключові атрибути відношення функціонально мінімально залежать від первинного ключа. Іншими словами, 2NF вимагає, щоб відношення не містило часткових функціональних залежностей.

Стосовно нашого прикладу: відношення Employer знаходиться у 2NF, а відношення Employer_Project_Task – ні, оскільки атрибут Em_Task функціонально залежить від двох ключових атрибутів: Em_Number та Pr_Number. Будь-яке змінне відношення, що перебуває у 1NF, але не перебуває у 2NF, може бути зведене до набору змінних відношень, що перебувають у 2NF. У результаті декомпозиції ми одержуємо набір проекцій вихідного змінного відношення, природне з'єднання значень яких відтворює значення вихідного змінного відношення (тобто це декомпозиція без втрат).

***Третя нормальна форма***

Відношення перебуває у 3NF, якщо воно перебуває в 2NF, і всі неключові атрибути відношення залежать тільки від первинного ключа. Іншими словами, 3NF вимагає, щоб відношення не містило транзитивних функціонального зв’язку неключових атрибутів від ключа.

Функціональні залежності відношення Employer як і раніше породжують деякі аномалії відновлення. Вони викликаються наявністю транзитивного зв’язку Em_Number → Em_Pay (через зв'язок Em_Number → Em_Degrees і Em_Degrees → Em_Pay). Ці аномалії пов'язані з надмірністю зберігання значення атрибута Em_Pay у кожному кортежі, що характеризує службовців із тим самим розрядом:

- додавання кортежів – неможливо зберегти дані про новий розряд (і відповідному йому розміру зарплати), поки не з'явиться службовець із новим розрядом. Первинний ключ не може містити невизначені значення;
- видалення кортежів – при звільненні останнього службовця з даним розрядом ми втратимо інформацію про наявність такого розряду й відповідному розміру зарплати;
- модифікація кортежів – при зміні розміру зарплати, що відповідає деякому розряду, ми будемо змушені змінити значення атрибута Em_Pay у кортежах всіх службовців, яким призначений цей розряд (інакше не буде виконуватися зв'язок Em_Degrees → Em_Pay).

Можлива декомпозиція: для подолання цих труднощів зробимо декомпозицію змінного відношення Employer на два змінні відношення – Employer1 {Em_Number, Em_Degrees} й Degrees {Em_Degrees, Em_Pay}. На рис. 1.21 показані ER-діаграми цих змінних відношень.

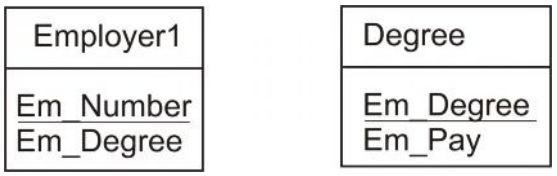



Рисунок 1.21 – ER-діаграми у змінних відношеннях Employer1 і Degrees

Таким чином, процедура зведення відношення до 3NF складається у виконанні двох проекцій: по правій і по лівій частині транзитивного функціонального зв’язку.

Зрозуміло, що в процесі нормалізації декомпозиція відношення на незалежні проекції є кращою. Необхідні й достатні умови незалежності проекцій відношення забезпечує *теорема Риссанена*: проекції r1 і r2 відношення r є незалежними тоді й тільки тоді, коли кожний зв'язок у відношенні r логічно виходить зі зв'язку у r1 і r2; загальні атрибути r1 і r2 утворять можливий ключ хоча б для одного з цих відношень.

Проілюструємо вірність цієї теореми на прикладі декомпозиції відношення Employer. У декомпозиції на проекції Employer1 і Degrees загальний атрибут Em_Degrees є можливим (і первинним) ключем відношення Degrees, а єдиний додатковий зв’язок відношення Employer (Em_Number→Em_Pay) логічно виходить зі зв'язку Em_Number→Em_Degrees і Em_Degrees→Em_Pay, які виконуються для відношення Employer1 й Degrees відповідно.

Атомарним відношенням називається відношення, яке неможливо декомпозувати на незалежні проекції. Далеко не завжди для неатомарних відношень потрібна декомпозиція на атомарні проекції. При виборі способу декомпозиції необхідно прагнути до одержання незалежних проекцій, але не обов'язково атомарних.

***Нормальна форма Бойса-Кодда***

Наприклад, нехай є змінне відношення Employer_Project_Task1 { Em_Number Em_Name, Pr_Number, Em_Task} з множиною зв'язків, зображених на рис. 1.22.

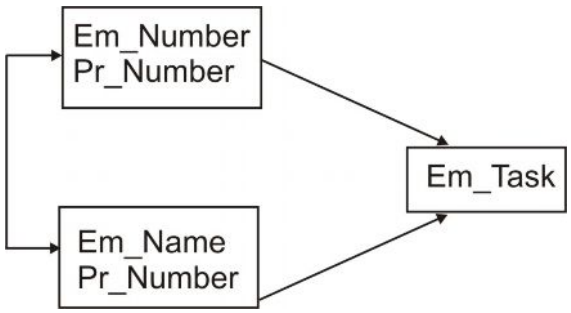



Рисунок 1.22 – Діаграма функціонального зв’язку відношення Employer_Project_Task1

У відношенні Employer_Project_Task1 службовці унікально ідентифікуються як за номерами справ, так і за іменами. Отже, існують зв’язки Em_Number→Em_Name й Em_Name→Em_Number. Але один службовець може брати участь у декількох проектах, тому можливими ключами є {Em_Number, Pr_Number} і {Em_Name, Pr_Number}.

Очевидно, що, хоча у відношенні Employer_Project_Task1 всі зв’язки неключових атрибутів від можливих ключів є мінімальними й транзитивні зв’язки відсутні, цьому відношенню властиві аномалії відновлення. Наприклад, у випадку зміни імені службовця необхідно обновити атрибут Em_Name у всіх кортежах відношення, що відповідають даному службовцеві. Інакше буде порушений зв'язок Em_Number→Em_Name, і БД виявиться у неузгодженому стані.

Причиною відзначених аномалій є те, що у вимогах 2NF і 3NF не була потрібна мінімальна функціональна залежність від первинного ключа атрибутів, що є компонентами інших можливих ключів. Проблему вирішує нормальна форма, що історично прийнята називати нормальною формою Бойса-Кодда і яка є уточненням 3NF у випадку наявності декількох можливих ключів, що перекриваються.

Змінна відношення перебуває в нормальній формі Бойса-Кодда (BCNF) у тому і тільки в тому випадку, коли будь-який виконуваний для цього змінного відношення нетривіальний і мінімальний функціональний зв'язок має як детермінант деякий можливий ключ даного відношення.

Відношення Employer_Project_Task1 може бути наведене до BCNF шляхом однієї з двох декомпозицій: Employer_Number_Name {Em_Number, Em_Name} і Employer_Number_Project_Task {Em_Number, Pr_Number, Pr_Task} з множиною зв’язків, показаними на рис. 7.5, і Employer_Number_Name {Em_Number, Em_Name} і Employer_Name_Project_Task {Em_Name, Pr_Number, Pr_Task} (зв’язки і значення результуючих змінних відношень виглядають аналогічно).

***Четверта нормальна форма***

Розглянемо ще одну можливу інтерпретацію змінної відношення Employer_Project_Task. Припустимо, що кожен службовець може брати участь у декількох проектах, але в кожному проекті ним повинні виконуватися ті самі завдання. Можливе значення змінної відношення Employer_Project_Task показано на рис. 1.24.

Додавання кортежу – якщо службовець, який вже бере участь у проектах, приєднується до нового проекту, то до тіла значення змінної відношення Employer_Project_Task необхідно додати стільки кортежів, скільки завдань виконує цей службовець.

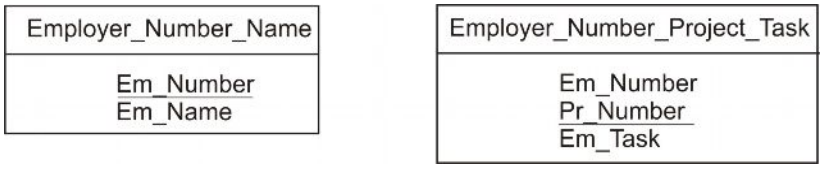



Рисунок 1.23 – ER-діаграми відношень Employer_Number_Name і Employer_Number_Project_Task

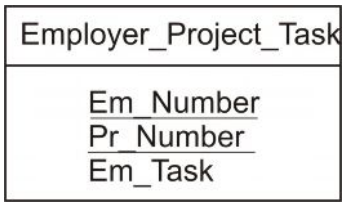



Рисунок 1.24 – Можливе значення змінної відношення Employer_Project_Task

Видалення кортежів – якщо службовець припиняє участь у проектах, то відсутня можливість зберегти дані про завдання, які він може виконувати.

Модифікація кортежів – при зміні одного з завдань службовця необхідно змінити значення атрибута Em_Task у скількох кортежах, у скількох проектах бере участь службовець.

Труднощі, пов'язані з відновленням змінної відношення Employer_Project_Task, вирішуються шляхом його декомпозиції на два змінні відношення: Employer_Project_Number {Em_Number, Pr_Number} і Employer_Task {Em_Number, Em_Task}. Значення цих змінних відношень, що відповідають значенню змінної відношення Employer_Project_Task показані на рис. 1.25.

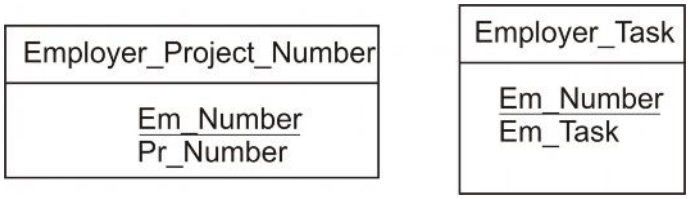



Рисунок 1.25 – Діаграми відношень Employer_Project_Number і Employer_Task

Зверніть увагу, що останній варіант змінної відношення Employer_Project_Task в BCNF, оскільки всі атрибути заголовка відношення входять до складу єдино можливого ключа. Раніше обговорені принципи нормалізації тут не застосовані, але ми одержали корисну декомпозицію. Справа в тому, що у випадку останнього варіанта відношення, ми маємо справу з новим видом залежності, уперше виявленим Роном Фейджином у 1971 р. Фейджин назвав залежності цього виду багатозначними (multi-valued dependency - MVD).

У відношенні Employer_Project_Task виконуються дві MVD: Em_Number→Pr_Number і Em_Number→Em_Task. Перша MVD означає, що кожному значенню атрибута Em_Number відповідає обумовлена тільки цим значенням множина значень атрибута Pr_Number. Аналогічно трактується друга MVD.

У змінній відношення r з атрибутами A, B, C (у загальному випадку складовими) є багатозначна залежність B від A (AB) у тому і тільки в тому випадку, коли множина значень атрибута B, що відповідає парі значень атрибутів A й C, залежить від значення A і не залежить від значення C. Багатозначні залежності мають цікаву властивість "подвійності", що демонструє така лема Фейджина:

У відношенні r {A, B, C} виконується MVD A→→B у тому і тільки в тому випадку, коли виконується MVD A→→C.

Функціональний зв'язок є частковим випадком MVD, коли множина значень залежного атрибута обов'язково складається з одного елемента. Таким чином, якщо виконується зв'язок A→B, то виконується й MVD A→→B.

*Теорема Фейджина*. Нехай є змінна відношення r з атрибутами A, B, C (у загальному випадку, складовими). Відношення r декомпозується без втрат на проекції {A, B} й {A, C} тоді й тільки тоді, коли для нього виконується MVD A→→B | C.

Відношення перебуває у 4NF, якщо воно перебуває в 3NF або BCNF і всі незалежні багатозначні функціональні зв'язки рознесені в окремі відношення з тим самим ключем. Іншими словами, 4NF застосовується при наявності у відношенні більш ніж однієї MVD і вимагає, щоб відношення не містило незалежних багатозначних MVD.

***П'ята нормальна форма***

Приведення відношення до 4NF припускає його декомпозицію без втрат на дві проекції (як й у випадку 2NF, 3NF й BCNF). Однак бувають (хоча й нечасто) випадки, коли декомпозиція без втрат на дві проекції неможлива, але можна зробити декомпозицію без втрат на більше число проекцій. Будемо називати n-декомпозованими відношеннями відношення, що може бути декомпозовано без втрат на n проекцій.

У змінній відношення r з атрибутами (можливо, складовими) A й B MVD A→→B називається тривіальною, якщо або AЄB, або A UNION B збігається із заголовком відношення r.

Тривіальна MVD завжди задовольняється. При AЄB вона вироджується у тривіальний зв'язок. У випадку A UNION B = Hr вимоги багатозначної залежності дотримуються очевидним способом.

Для прикладу n-декомпозованого відношення при n > 2 розглянемо п'ятий варіант змінної відношення Employer_Project_Task, у якій є єдино можливий ключ {Em_Number, Pr_Number, Em_Task} і відсутні нетривіальні MVD (рис.1.26).

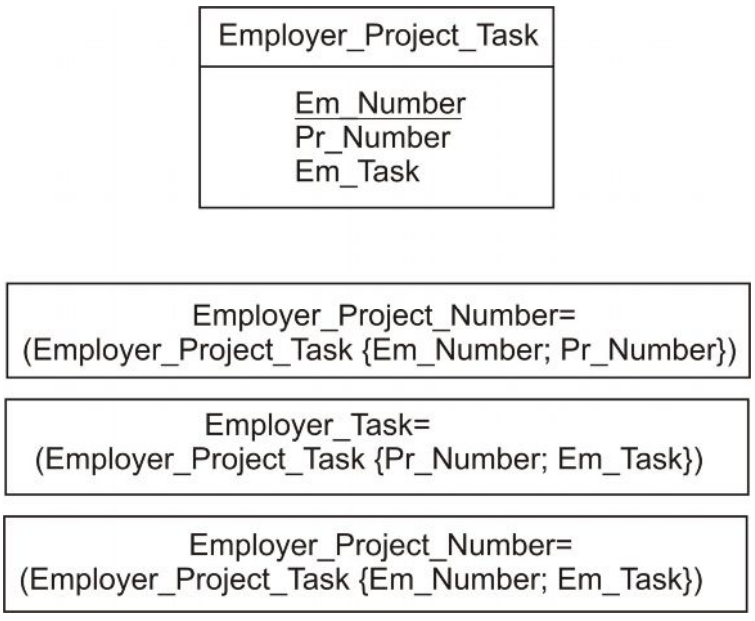



Рисунок 1.26 – Можливі значення змінної відношення Employer_Project_Task та результати проекцій

Залежність проекції/з'єднання. Твердження про те, що тіло відношення Employer_Project_Task відновлюється без втрат шляхом природного з'єднання його проекцій Employer_Project_Number, Project_Number_Task і Employer_Task еквівалентно наступному

звичайному обмеженню реального світу, що для відношення Employer_Project_Task може бути сформульовано природною мовою в такий спосіб (BEPT, BEPN, BPNT і BET позначають тіла значень змінних відношення Employer_Project_Task, Employer_Project_Number, Project_Number_Task і Employer_Task відповідно): якщо службовець із номером en бере участь у проекті pn, і в проекті pn виконується завдання et, і службовець із номером en виконує завдання et, то службовець із номером en виконує завдання et у проекті pn.

У загальному вигляді таке обмеження називається залежністю проекції/з'єднання і має таке формальне визначення.

Нехай задана змінна відношення r, і A, B, ..., Z є довільними підмножинами заголовка r (складовими, що перекриваються атрибутами). У змінної відношення r задовольняється залежність проекції/з'єднання (Project-Join Dependency - PJD) *( A, B, ..., Z) тоді й тільки тоді, коли будь-яке припустиме значення r можна одержати шляхом природного з'єднання проекцій цього значення на атрибути A, B, ..., Z.

У змінної відношення Employer_Project_Task виконується PJD* ({Em_Number, Pr_Number}, {Pr_Number, Em_Task}, {Em_Number, Em_Task}). Наявність такого PJD забезпечує можливість декомпозиції відношення на три для уникнення аномалії відновлення.

У змінної відношення r PJD *( A, B, ..., Z) називається можливими ключами, що маються на увазі, в тому і тільки в тому випадку, коли кожен складений атрибут A, B, ..., Z є суперключем r, тобто включає хоча б один можливий ключ r.

У змінної відношення r залежність проекції/з'єднання *(A, B, ..., Z) називається тривіальною, якщо хоча б один зі складених атрибутів A, B, ..., Z збігається із заголовком r.

Загальноприйняте визначення 5NF має такий вигляд: змінна відношення r перебуває в 5NF, або в нормальній формі проекції/з'єднання (5NF, або PJ/NF - Project-Join Normal Form) у тому і тільки в тому випадку, коли кожна нетривіальна PJD в r розуміється можливими ключами r.

Таким чином, щоб розпізнати, що дана змінна відношення r перебуває в 5NF, необхідно знати всі можливі ключі r і всі PJD цієї змінної відношення. Виявлення всіх залежностей з'єднання є нетривіальним завданням, і для його вирішення немає загальних методів. Тому на практиці проектування реляйціних БД методом нормалізації звичайно завершується після досягнення 4NF, і відношення, що перебувають в 4NF, як правило, перебувають і в 5NF. 5NF є "остаточною" нормальною формою, яку можна досягти в процесі нормалізації на основі проекцій.

Нормальні форми характеризуються такими властивостями:

1NF - всі атрибути відношення прості;

2NF - відношення перебуває в 1NF і не містить часткових залежностей;

3NF - відношення перебуває в 2NF і не містить транзитивних залежностей від ключа;

BKNF - відношення перебуває в 3NF і не містить залежностей ключів від неключових атрибутів;

4NF - застосовується при наявності більш ніж одної багатозначної залежності - відношення перебуває в BKNF або 3NF і не містить незалежних багатозначних функціональних залежностей;

5NF - відношення перебуває в 4NF і не містить залежностей по з'єднанню.

### 1.3.4 Алгоритм методу декомпозиції відношення

Тепер нам відомо, із чого почати нормалізацію - з універсального відношення; що перевірити - знаходження вихідного відношення в BKNF; що почати - декомпозицію вихідного відношення на два інших відношення; і коли зупинитися - всі відношення БД у BKNF. Таким чином, можна сформулювати загальний алгоритм проектування логічної моделі реляційної БД методом декомпозиції:

***Алгоритм методу декомпозиції відношення***

1. Розроблення універсального відношення для БД.

2. Визначення всіх функціональних залежностей між атрибутами відношення.

3. Визначення, чи перебуває відношення в BKNF. Якщо так, то завершити проектування; в іншому випадку відношення повинне бути розбите на два інших відношення.

4. Повторення пунктів 2 й 3 для кожного нового відношення, отриманого у результаті декомпозиції.

Правило ланцюжка полягає в такому:

Якщо A>B>C, то в якості функціональної залежності для здійснення проекції використовується крайня права залежність або "кінець ланцюжка" B>C.

### 1.3.5 Методи проектування на основі синтезу відношення

Виключити надмірність у вихідному наборі функціональних залежностей можна шляхом застосування правил виводу. Як відомо, для класу F-залежностей досить використовувати шість таких правил. При цьому критерієм завершення процедури виключення може служити одержання мінімального покриття вихідного набору функціональних залежностей.

Неодиничність мінімальних покриттів указує на те, що порядок виключення надлишкових функціональних залежностей може вплинути на отримані результати. Звідси маємо емпіричне правило: надлишкові функціональні залежності необхідно видаляти по одній, щоразу проводячи аналіз отриманого набору функціональних залежностей на надмірність.

***Алгоритм методу синтезу відносин***

Причиною втрати функціональної залежності є деяка функціональна залежность , що не може бути виключена з множини F-залежностей, пов'язаних з одержуваними відношеннями r1 або r2. Таким чином, суть проблеми зводиться до порушення замкнутості реляційних операцій над функціональними залежностями на отриманій схемі БД. Для вирішення цієї проблеми необхідно поповнити мінімальне покриття функціональних залежностей, або, як говорять, підсилити мінімальне покриття.

На шляху вирішення цієї проблеми було б непогано підсилити всі функціональні залежності, зв'язавши їх з унікальними ключами, скажемо, описуючи для них унікальні індекси. Тоді можна контролювати цілісність БД. Для цього необхідно підсилити мінімальне покриття. Просто кажучи, підсилювання мінімального покриття означає, що виділено множину первинних ключів, і всі функціональні залежності з мінімального покриття переглянуті через призму цієї множини з погляду виводності функціональних залежностей розглянутої БД.

Реляційна БД називається повною, якщо:

- всі функціональні залежності посилені ключами;
- всі відношення перебувають в 3NF;
- не існує варіанта БД з меншим числом схем, що задовольняють перерахованим вище властивостям.

## 2 Введення в структуровану мову запитів - SQL

### 2.1 Елементи мови SQL

У даній темі розглянемо елементи мови SQL (Structured Query Language). Мова SQL стала фактично стандартною мовою доступу до БД. Всі СКБД, що претендують на назву "реляційні", реалізують той або інший діалект SQL. Багато які нереляційні системи також мають у цей час засоби доступу до реляційних даних. Метою стандартизації є переносимість додатків між різними СКБД.

Треба відмітити, що в цей час, жодна система не реалізує стандарт SQL у повному обсязі. Крім того, у всіх діалектах мови є можливості, що не є стандартними. Таким чином, можна сказати, що кожен діалект - це надмножина деякої підмножини стандарту SQL. Це ускладнює переносимість додатків, розроблених для одних СКБД в інші СКБД.

Мова SQL оперує термінами, що трохи відрізняються від термінів реляційної теорії, наприклад, замість "відношення" використовуються "таблиці", замість "кортежів" - "рядки", замість "атрибутів" - "колонки" або "стовпці".

Стандарт мови SQL, хоча й заснований на реляційній теорії, але у багатьох місцях відходить він неї. Наприклад, відношення у реляційній моделі даних не допускає наявності однакових кортежів, а таблиці у термінології SQL можуть мати однакові рядки. Є й інші відмінності.

Мова SQL є реляційно повною. Це означає, що будь-який оператор реляційної алгебри може бути виражений відповідними оператором SQL.

### 2.2 Припустимі типи даних

Будь-який діалект SQL підтримують три загальних типи даних: строковий, числовий й тип для подання дати й часу. Завдання типу даних визначає значення й довжину даних, а також формат їхнього подання при візуалізації.

Для всіх типів даних визначено так зване нуль-значення, що вказує на відсутність даних у колонку зазначеного типу, тобто та обставина, що значення даних у сучасний момент часу невідомо.

Дані строкового типу являють собою послідовність рядків символів. Строкові дані можуть бути задані як з визначеною довжиною (ключові слова char або varchar (довжина рядка)), так і без вказівки довжини (ключове слово long varchar) для подання рядків довільної довжини. Тип даних varchar2 визначає рядок символів змінної довжини, що має максимальний розмір size. На відміну від строкового типу з визначеною довжиною, з рядками long varchar не допускаються операції порівняння, і вони не можуть бути використані у виразах і як аргументи більшості убудованих функцій. Рядки останнього типу можуть застосовуватися для збереження бітових образів. Стандарт SQL-92 не має типу long varchar й varchar.

Зверніть увагу на тип даних varchar2. Він, так само як і тип даних char, призначений для подання алфавітно-цифрових даних. Але він має формат змінної довжини. Останнє означає, що довжина колонки такого типу дорівнює числу символів у ній, у той час як колонка типу char використовує весь виділений для неї простір.

Існує два типи числових даних. Цілі й речовинні значення (наприклад, сальдо банківського рахунку або ставка відсотка). Вони є об'єктом математичної обробки. Строкові числові дані, у яких єдино припустимими символами є цифри (наприклад, номери банківських рахунків).

Числові типи даних призначені для подання цілих чисел, чисел з десятковою крапкою й чисел із плаваючою крапкою. Будь-яке подання чисел задається своєю точністю й масштабом. Точність визначає припустиме поданням кількість значущих цифр числа, а масштаб - кількість значущих цифр після десяткової крапки.

Для подання цілих чисел використовуються типи interger (точність 10 значущих цифр) і smallint (точність 5 значущих цифр).

Для подання чисел з фіксованою десятковою крапкою використовуються типи number (точність, масштаб) (для чисел з точністю до 15 значущих цифр) і decimal (точність, масштаб) (для чисел заданої точності до 15 значущих цифр). Якщо вказати для колонки тип number без завдання масштабу, максимальне число значущих цифр для Oracle буде 105. Замість завдання точності й масштабу може бути зазначений символ *. Це буде еквівалентно завданню простого типу number. Розходження між цими типами даних полягає в тому, що для типу number немає необхідності стежити за точністю при виконанні операцій.

Для подання чисел із плаваючою крапкою в SQL передбачені такі типи даних:

Double Precision - для чисел з точністю від 22 до 53 значущих цифр;

Float (точність) - для подання чисел з точністю від 1 до 21 значущої цифри;

Real - для чисел із точністю за замовчуванням (залежить від конкретної реалізації).

Тип даних для подання дати й часу відсутній у стандарті SQL. Звичайно в конкретних діалектах SQL використовуються три типи для подання таких даних:

datestamp (timestamp) - для подання дати й часу;

date - для подання дати;

time - для подання часу.

Константи, вираження, системні змінні.

Константи звичайно визначають єдине значення й, відповідно до типу даних, що представляють, можуть бути строковими, числовими й представляти дату/час. Строкові константи повинні бути укладені в одинарні лапки.

У SQL існує ряд визначених системних змінних, які можна використовувати у виразах замість імен колонок і констант. До них відносять такі:

NULL - для подання невизначених значень;

ROWID - (в SQLBase) внутрішній системний номер рядка в таблиці;

USER - ім'я користувача, активного в цей момент;

SYSDATETIME - системний поточний час і дата;

SYSDATE - системна поточна дата;

SYSTIME - системний поточний час;

SYSTIMEZONE - часовий пояс, установлений у системі.

Виразом у SQL є ітем або комбінація ітемів з припустимими для них операціями, що дає єдине значення. У якості ітемів можуть виступати імена колонок, константи, пов'язанні змінні, результати обчислень функцій, системні змінні й інші вирази. При цьому, якщо один з ітемів має нуль-значення, то результат виразу також має нуль-значення.

### 2.3 Оператори SQL

Основу мови SQL становлять оператори, умовно розбиті на кілька груп за виконуваними функціями.

Можна виділити такі групи операторів (перераховані не всі оператори SQL):

*Оператори DDL (Data Definition Language) - оператори визначення об'єктів БД:*

- CREATE SCHEMA - створити схему БД;
- DROP SHEMA - видалити схему БД;
- CREATE TABLE - створити таблицю;
- ALTER TABLE - змінити таблицю;
- DROP TABLE - видалити таблицю;
- CREATE DOMAIN - створити домен;
- ALTER DOMAIN - змінити домен;
- DROP DOMAIN - видалити домен;
- CREATE COLLATION - створити послідовність;
- DROP COLLATION - видалити послідовність;

- CREATE VIEW - створити подання;
- DROP VIEW - видалити подання.

*Оператори DML (Data Manipulation Language) - оператори маніпулювання даними:*

- SELECT - відібрати рядок із таблиць;
- INSERT - додати рядок у таблицю;
- UPDATE - змінити рядок у таблиці;
- DELETE - видалити рядок у таблиці;
- COMMIT - зафіксувати внесені зміни;
- ROLLBACK - відкотити внесені зміни.

*Оператори захисту й керування даними:*

- CREATE ASSERTION - створити обмеження;
- DROP ASSERTION - видалити обмеження;
- GRANT - надати привілею користувачеві або додатку на маніпулювання об'єктами;
- REVOKE - скасувати привілею користувача або додатка.

Крім того, є групи операторів установки параметрів сеансу, одержання інформації про БД, оператори статичного SQL, оператори динамічного SQL.

Найбільш важливими для користувача є оператори маніпулювання даними (DML).

***Приклади використання операторів маніпулювання даними***

*INSERT - вставка рядків у таблицю*

*Приклад*. Вставка одного рядка в таблицю:

```
INSERT INTO
  P (PNUM, PNAME)
  VALUES (4, "Іванов");
```

*Приклад*. Вставка в таблицю декількох рядків, обраних з іншої таблиці (у таблицю TMP_TABLE уставляються дані про постачальників з таблиці P, що мають номери, більші 2):

```
INSERT INTO
  TMP_TABLE (PNUM, PNAME)
  SELECT PNUM, PNAME
    FROM P
    WHERE P.PNUM>2;
```

*UPDATE - відновлення рядків у таблиці*

*Приклад*. Відновлення декількох рядків у таблиці:

```
UPDATE P
  SET PNAME = "Пушников"
  WHERE P.PNUM = 1;
```

*DELETE - видалення рядків у таблиці*

*Приклад*. Видалення декількох рядків у таблиці:

```
DELETE FROM P
  WHERE P.PNUM = 1;
```

*Приклад*. Видалення всіх рядків у таблиці:

```
DELETE FROM P;
```

*Приклади використання оператора SELECT*

Оператор SELECT є фактично найважливішим для користувача й самим складним оператором SQL. Він призначений для вибірки даних із таблиць, тобто він, властиво, і реалізує одне з їхніх основних призначень БД - надавати інформацію користувачеві.

Оператор SELECT завжди виконується над деякими таблицями, що входять у БД.

Насправді в БД можуть бути не тільки постійно збережені таблиці, а також тимчасові таблиці й так звані подання. Подання - це SELECT-вираз, що зберігається в БД. З погляду користувачів подання - це таблиця, яка не зберігається постійно в БД, а "виникає" у момент звертання до неї. З погляду оператора SELECT і постійно збережені таблиці, і тимчасові таблиці, й подання виглядають зовсім однаково. Звичайно при реальному виконанні

оператора SELECT системою враховуються розходження між збереженими таблицями й поданнями, але ці розходження сховані від користувача.

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

***Відбір даних з однієї таблиці***

*Приклад*. Вибрати всі дані з таблиці постачальників (ключові слова ***SELECT…FROM***…):

```
SELECT*
FROM P;
```

У результаті одержимо нову таблицю, що містить повну копію даних із вихідної таблиці P.

*Приклад*. Вибрати всі рядки з таблиці постачальників, що задовольняють деякій умові (ключове слово ***WHERE***…):

```
SELECT*
FROM P
WHERE P...PNUM>2;
```

Як умову в розділі WHERE можна використовувати складні логічні вирази, що використовують поля таблиць, константи, порівняння (>, <, = і т.д.), дужки, союзи AND й OR, заперечення NOT.

*Приклад*. Вибрати деякі колонки з вихідної таблиці (вказівка списку колонок, що відбирають):

```
SELECT P.NAME
  FROM P;
```

У результаті одержимо таблицю з однією колонкою, що містить всі найменування постачальників. Якщо у вихідній таблиці були присутні кілька постачальників з різними номерами, але однаковими найменуваннями, то в результуючій таблиці будуть рядки з повтореннями - дублікати рядків автоматично не відкидаються.

*Приклад*. Вибрати деякі колонки з вихідної таблиці, видаливши з результату повторювані рядки (ключове слово ***DISTINCT***):

```
SELECT DISTINCT P.NAME
  FROM P;
```

Використання ключового слова DISTINCT призводить до того, що у результуючій таблиці будуть вилучені всі повторювані рядки.

*Приклад*. Використання скалярних виразів і перейменувань колонок у запитах (ключове слово ***AS***…):

```
SELECT TOVAR...TNAME, TOVAR.KOL,
            TOVAR.PRICE, "="AS EQU,
            TOVAR.KOL*TOVAR.PRICE AS SUMMA
FROM TOVAR;
```

У результаті одержимо таблицю з колонками, яких не було у вихідній таблиці TOVAR:

| TNAME | KOL | PRICE | EQU | SUMMA |
|---|---|---|---|---|
| Болт | 10 | 100 | = | 1000 |
| Гайка | 20 | 200 | = | 4000 |
| Гвинт | 30 | 300 | = | 9000 |

*Приклад*. Упорядкування результатів запиту (ключове слово ***ORDER BY***…):

```
SELECT PD...PNUM, PD.DNUM, PD.VOLUME
FROM PD
ORDER BY DNUM;
```

В результаті одержимо наступну таблицю, впорядковану по полю DNUM:

| PNUM | DNUM | VOLUME |
|---|---|---|
| 1 | 1 | 100 |
| 2 | 1 | 150 |
| 3 | 1 | 1000 |
| 1 | 2 | 200 |
| 2 | 2 | 250 |
| 1 | 3 | 300 |

*Приклад*. Упорядкування результатів запиту по декількох полях зі зростанням або убуванням (ключові слова ***ASC***, ***DESC***):

```
SELECT   PD.PNUM, PD.DNUM, PD.VOLUME
FROM PD
ORDER BY  DNUM ASC, VOLUME DESC;
```

У результаті одержимо таблицю, у якій рядки йдуть у порядку зростання значення поля DNUM, а рядки з однаковим значенням DNUM ідуть у порядку убування значення поля VOLUME:

| PNUM | DNUM | VOLUME |
|---|---|---|
| 3 | 1 | 1000 |
| 2 | 1 | 150 |
| 1 | 1 | 100 |
| 2 | 2 | 250 |
| 1 | 2 | 200 |
| 1 | 3 | 300 |

Якщо явно не зазначені ключові слова ASC або DESC, то за замовчуванням береться впорядкування по зростанню (ASC).

***Відбір даних із декількох таблиць***

*Приклад*. Природне з'єднання таблиць (спосіб 1 - явна вказівка умов з'єднання):

```
SELECT   P.PNUM, P.PNAME, PD.DNUM,
            PD.VOLUME
FROM P, PD
WHERE P.PNUM = PD.PNUM;
```

У результаті одержимо нову таблицю, у якій рядки з даними про постачальників з'єднані з рядками з даними про поставки деталей:

| PNUM | PNAME | DNUM | VOLUME |
|---|---|---|---|
| 1 | Іванов | 1 | 100 |
| 1 | Іванов | 2 | 200 |
| 1 | Іванов | 3 | 300 |
| 2 | Петров | 1 | 150 |
| 2 | Петров | 2 | 250 |
| 3 | Сидоров | 1 | 1000 |

Таблиці, що з'єднуються, перелічені у розділі FROM оператора, умова з'єднання наведена у розділі WHERE. Розділ WHERE, крім умови з'єднання таблиць, може також містити й умови відбору рядків.

*Приклад*. Природне з'єднання таблиць (спосіб 2 - ключові слова ***JOIN…USING…***):

```
SELECT PNUM, P.PNAME, PD.DNUM,
       PD.VOLUME
FROM P JOIN PD USING PNUM;
```

Ключове слово USING дозволяє явно вказати, за якими із загальних колонок таблиць буде вироблятися з'єднання.

*Приклад*. Природне з'єднання таблиць (спосіб 3 - ключове слово ***NATURAL JOIN***):

```
SELECT   P.PNUM, P.PNAME, PD.DNUM,
         PD.VOLUME
FROM P NATURAL JOIN PD;
```

У розділі FROM не зазначено, по яких полях виробляється з'єднання. NATURAL JOIN автоматично з'єднує по всіх однакових полях у таблицях.

*Приклад*. Природне з'єднання трьох таблиць:

```
SELECT   P.PNAME, D.DNAME, PD.VOLUME
FROM   P NATURAL JOIN PD NATURAL JOIN D;
```

У результаті одержимо таку таблицю:

| PNAME | DNAME | VOLUME |
|---|---|---|
| Іванов | Болт | 100 |
| Іванов | Гайка | 200 |
| Іванов | Гвинт | 300 |
| Петров | Болт | 150 |
| Петров | Гайка | 250 |
| Сидоров | Болт | 1000 |

*Приклад*. Прямий добуток таблиць:

```
SELECT   P.PNUM, P.PNAME, D.DNUM,
         D.DNAME
FROM P, D;
```

У результаті одержимо таку таблицю:

| PNUM | PNAME | DNUM | DNAME |
|---|---|---|---|
| 1 | Іванов | 1 | Болт |
| 1 | Іванов | 2 | Гайка |
| 1 | Іванов | 3 | Гвинт |
| 2 | Петров | 1 | Болт |
| 2 | Петров | 2 | Гайка |
| 2 | Петров | 3 | Гвинт |
| 3 | Сидоров | 1 | Болт |
| 3 | Сидоров | 2 | Гайка |
| 3 | Сидоров | 3 | Гвинт |

Оскільки не зазначена умова з'єднання таблиць, то кожен рядок першої таблиці з'єднується з кожним рядком другої таблиці.

*Приклад*. З'єднання таблиць за довільною умовою. Розглянемо таблиці постачальників і деталей, яким присвоєний деякий статус.

Таблиця 2.1 – Відношення P (Постачальники)

| PNUM | PNAME | PSTATUS |
|---|---|---|
| 1 | Іванов | 4 |
| 2 | Петров | 1 |
| 3 | Сидоров | 2 |

Таблиця 2.2 – Відношення D (Деталі)

| DNUM | DNAME | DSTATUS |
|---|---|---|
| 1 | Болт | 3 |
| 2 | Гайка | 2 |
| 3 | Гвинт | 1 |

Відповідь на запитання "які постачальники мають право поставляти певні деталі?" дає такий запит:

```
SELECT  P.PNUM, P.PNAME,
        P.PSTATUS,
        D.DNUM, D.DNAME, D.DSTATUS
FROM P, D
WHERE P.PSTATUS >= D.DSTATUS;
```

У результаті одержимо таку таблицю:

| PNUM | PNAME | PSTATUS | DNUM | DNAME | DSTATUS |
|---|---|---|---|---|---|
| 1 | Іванов | 4 | 1 | Болт | 3 |
| 1 | Іванов | 4 | 2 | Гайка | 2 |
| 1 | Іванов | 4 | 3 | Гвинт | 1 |
| 2 | Петров | 1 | 3 | Гвинт | 1 |
| 3 | Сидоров | 2 | 2 | Гайка | 2 |
| 3 | Сидоров | 2 | 3 | Гвинт | 1 |

### 2.4 Використання імен кореляції (аліасів, псевдонімів)

Іноді доводиться виконувати запити, у яких таблиця з'єднується сама із собою, або одна таблиця з'єднується двічі з іншою таблицею. При цьому використовуються імена кореляції (аліаси, псевдоніми), які дозволяють розрізняти копії та таблиці-оригінали. Імена кореляції вводяться у розділі FROM і йдуть через пробіл після імені таблиці. Імена кореляції повинні використовуватися як префікс перед ім'ям стовпця й відокремлюються від імені стовпця крапкою. Якщо у запиті вказуються ті самі поля з різних екземплярів однієї таблиці, вони повинні бути перейменовані для усунення неоднозначності в іменуваннях колонок результуючої таблиці. Визначення імені кореляції діє тільки під час виконання запиту.

*Приклад*. Відібрати всі пари постачальників таким чином, щоб перший постачальник у парі мав статус, більший од статусу другого постачальника:

```
SELECT  P1.PNAME AS PNAME1,
        P1.PSTATUS AS PSTATUS1,
        P2.PNAME AS PNAME2,
        P2.PSTATUS AS PSTATUS2
FROM    P P1, P P2
WHERE P1.PSTATUS1 > P2.PSTATUS2;
```

В результаті одержимо наступну таблицю:

| PNAME1 | PSTATUS1 | PNAME2 | PSTATUS2 |
|---|---|---|---|
| Іванов | 4 | Петров | 1 |

| | | | |
|---|---|---|---|
| Іванов | 4 | Сидоров | 2 |
| Сидоров | 2 | Петров | 1 |

*Приклад*. Розглянемо ситуацію, коли деякі постачальники (назвемо їх контрагенти) можуть виступати як постачальники деталей, так як і одержувачі. Таблиці, що зберігають дані можуть мати такий вигляд:

Таблиця 2.3 – Відношення CONTRAGENTS

| **Номер контрагента NUM** | **Ім'я контрагента NAME** |
|---|---|
| 1 | Іванов |
| 2 | Петров |
| 3 | Сидоров |

Таблиця 2.4 – Відношення DETAILS

| **Номер деталі DNUM** | **Найменування деталі DNAME** |
|---|---|
| 1 | Болт |
| 2 | Гайка |
| 3 | Гвинт |

Таблиця 2.5 – Відношення CD (Поставки)

| **Номер постачальника PNUM** | **Номер одержувача CNUM** | **Номер деталі DNUM** | **Кількість, що поставляється, VOLUME** |
|---|---|---|---|
| 1 | 2 | 1 | 100 |
| 1 | 3 | 2 | 200 |
| 1 | 3 | 3 | 300 |
| 2 | 3 | 1 | 150 |
| 2 | 3 | 2 | 250 |
| 3 | 1 | 1 | 1000 |

У таблиці CD поля PNUM й CNUM є зовнішніми ключами, що посилаються на потенційний ключ NUM у таблиці CONTRAGENTS.

Відповідь на запитання "хто, кому, що, в якій кількості поставляє" дається таким запитом:

```
SELECT  P.NAME AS PNAME,
          C.NAME AS CNAME,
          DETAILS.DNAME, CD.VOLUME
FROM    CONTRAGENTS P, CONTRAGENTS C,
          DETAILS, CD
WHERE  P.NUM = CD.PNUM AND
          C.NUM = CD.CNUM AND
          D.DNUM = CD.DNUM;
```

У результаті одержимо таку таблицю:

| **PNAME** | **CNAME** | **DNAME** | **VOLUME** |
|---|---|---|---|
| Іванов | Петров | Болт | 100 |

| | | | |
|---|---|---|---|
| Іванов | Сидоров | Гайка | 200 |
| Іванов | Сидоров | Гвинт | 300 |
| Петров | Сидоров | Болт | 150 |
| Петров | Сидоров | Гайка | 250 |
| Сидоров | Іванов | Болт | 1000 |

Цей самий запит може бути виражений дуже великою кількістю способів, наприклад, так:

```
SELECT  P.NAME AS PNAME,
          C.NAME AS CNAME,
           DETAILS.DNAME, CD.VOLUME
FROM    CONTRAGENTS P, CONTRAGENTS C,
          DETAILS NATURAL JOIN CD
WHERE  P.NUM = CD.PNUM AND
          C.NUM = CD.CNUM;
```

## 2.5 Вбудовані функції

***Арифметичні функції***

SQL підтримує повний набір арифметичних операцій і математичних функцій для побудови арифметичних виражень над колонками БД (+, -, *, /, ABS, LN, SQRT і т.д.). Перелік основних вбудованих математичних функцій поданий нижче:

ABS(X) Повертає абсолютне значення числа X;
ACOS(X) Повертає арккосинус числа X;
ASIN(X) Повертає арксинус числа X;
ATAN(X) Повертає арктангенс числа X;
COS(X) Повертає косинус числа X;
EXP(X) Повертає експоненту числа X;
SIGN(X) Повертає -1, якщо X<0,0, якщо X=0, +1, якщо X>0;
LN(X) Повертає натуральний логарифм числа X;
MOD(X,Y) Повертає залишок від розподілу X на Y;
CEIL(X) Повертає найменше ціле, більше або рівне X;
ROUND(X, n) Округляє число X до числа з n знаками після крапки;
SIN(X) Повертає синус числа X;
SQRT(X) Повертає квадратний корінь числа X;
TAN(X) Повертає тангенс числа X;
FLOOR(X) Повертає найбільше ціле менше або рівне X;
LOG(A, X) Повертає логарифм числа X по основі A;
SINH(X) Повертає гіперболічний синус числа X;
COSH(X) Повертає гіперболічний косинус числа X;
TANH(X) Повертає гіперболічний тангенс числа X;
TRANC(X,n) Скорочує число X до числа з n знаками після десяткової крапки;
POWER(A,X) Повертає значення A, зведене в ступінь X.

Арифметичні вираження необхідні для одержання даних, які безпосередньо не зберігаються в колонках таблиць БД, але значення яких необхідні користувачеві. Припустимо, що вам необхідний список службовців, що показує виплату, що одержав кожен службовець із урахуванням премій і штрафів

***Функції для обробки дати***

У діалекті SQL є невеликий набір функцій для маніпулювання колонками з типом date. Список основних функцій обробки дати й часу наведений нижче:

SYSDATE Повертає поточну дату й час;
ROUND(D[,F]) - Округлює значення дати D відповідно до заданого шаблона;
TRANC(D[,F]) - Скорочує значення дати D відповідно до заданого шаблона;

NEXT_DAY(D,S) - Повертає дату дня, що є першим днем, більше пізнім, ніж поточна дата з назвою S.

Якщо вам потрібен був список нових службовців, що надійшли за останній квартал в організацію, то ви можете написати запит у такому вигляді:

```
SELECT ENAME, HIREDATE,
          HIREDATE + 92  DAYS
FROM    EMPLOYEE
WHERE HIREDATE + 92 DAYS > SYSDATE
         AND DEPNO=30;
```

Ключове слово SYSDATE завжди повертає поточну дату. У цьому прикладі також показано, як використовуються арифметичний оператор додавання зі змінними типу "дата". До змінного типу "дата" можна додавати й віднімати з нього ціле число днів, місяців, років, годин, хвилин, секунд, мікросекунд. Для цього використаються відповідні ключові слова (DAY, MONTH і т.д.), що випливають за цілою константою (дробова частина ігнорується, якщо ви вказуєте число з десятковою крапкою). Є обмеження на використання дужок у таких вираженнях (так, висновок у дужки вираження 1 DAYS + 1 YEARS приведе до помилки).

***Використання агрегатних функцій у запитах***

У мові SQL передбачені такі оператори агрегатних функцій:

AVG(X) = AVG(ALL X) AVG(DISTINCT X) Обчислює середнє значення аргументу, що може бути виразом будь-якого типу. Нуль-значення ігноруються, ключове слово DISTINCT придушує дублікати.

COUNT(*) COUNT(X) = COUNT(ALL X) COUNT(DISTINCT X) Обчислює числа ітемів. При вказівці * завжди повертається число рядків у таблиці. Вказівка DISTINCT придушує дублікати.

MAX(X) = MAX(ALL X) MAX (DISTINCT X) Обчислює максимальне значення аргументу, що може бути виразом будь-якого типу. Нуль-значення ігноруються, ключове слово DISTINCT придушує дублікати.

MIN(X) = MIN(ALL X) MIN (DISTINCT X) Обчислює мінімальне значення аргументу, що може бути виразом будь-якого типу. Нуль-значення ігноруються, ключове слово DISTINCT придушує дублікати.

SUM(X) = SUM(ALL X) SUM (DISTINCT X) Обчислює суму значення аргументу, що може бути виразом будь-якого типу. Нуль-значення ігноруються, ключове слово DISTINCT придушує дублікати.

STDDEV([DISTINCT|ALL]X) Обчислює стандартне відхилення на безлічі значень аргументу, що може бути виразом будь-якого типу. Нуль-значення ігноруються, ключове слово DISTINCT придушує дублікати.

VARIANCE([DISTINCT|ALL]) Обчислює квадрат дисперсії.

*Приклад*. Одержати загальну кількість постачальників (ключове слово ***COUNT***):

```
SELECT COUNT(*) AS N
FROM P;
```

У результаті одержимо таблицю з одним стовпцем й одним рядком, що містить кількість рядків з таблиці P.

*Приклад*. Одержати загальну, максимальну, мінімальну й середню кількості деталей, що поставляють, (ключові слова SUM**,** MAX**,** MIN**,** AVG):

```
SELECT   SUM(PD.VOLUME) AS SM,
            MAX(PD.VOLUME) AS MX,
            MIN(PD.VOLUME) AS MN,
            AVG(PD.VOLUME) AS AV
FROM PD;
```

В результаті одержимо таку таблицю з одним рядком:

| **SM** | **MX** | **MN** | **AV** |
|---|---|---|---|
| 2000 | 1000 | 100 | 333. 33333333 |

***Використання агрегатних функцій з угрупованнями***

*Приклад*. Для кожної деталі одержати сумарну кількість, що поставляється (ключове слово ***GROUP BY***…):

```
SELECT..DNUM, SUM(PD.VOLUME) AS SM
GROUP BY PD.DNUM;
```

Цей запит буде виконуватися в такий спосіб. Спочатку рядки вихідної таблиці будуть згруповані так, щоб у кожну групу потрапили рядки з однаковими значеннями DNUM. Потім усередині кожної групи буде просумовано поле VOLUME. Від кожної групи до результуючої таблиці буде включений один рядок.

У переліку полів оператора SELECT, який містить розділ GROUP BY можна включати тільки агрегатні функції й поля, які входять в умову групування. Наступний запит видасть синтаксичну помилку:

```
SELECT  PD.PNUM, PD.DNUM,
            SUM(PD.VOLUME) AS SM
GROUP BY PD.DNUM;
```

Причина помилки у тому, що у перелік полів, які відбираються, включене поле PNUM, що не входить у розділ GROUP BY. І дійсно у кожну отриману групу рядків може входити кілька рядків із різними значеннями поля PNUM. З кожної групи рядків буде сформовано по одному підсумковому рядку. При цьому немає однозначної відповіді на питання, яке значення вибрати для поля PNUM у підсумковому рядку.

Деякі діалекти SQL не вважають це за помилку. Запит буде виконаний, але передбачити, які значення будуть внесені у поле PNUM у результуючій таблиці, неможливо.

*Приклад*. Одержати номери деталей, сумарна кількість поставки яких перевершує 400 (ключове слово ***HAVING***…).

Умова, що сумарна кількість поставки повинна бути більше 400, не може бути сформульована у розділі WHERE, тому що в цьому розділі не можна використовувати агрегатні функції. Умови, що використовують агрегатні функції повинні бути розміщені у спеціальному розділі HAVING:

```
SELECT   PD.DNUM, SUM(PD.VOLUME) AS SM
GROUP BY PD.DNUM
HAVING SUM(PD.VOLUME) > 400;
```

В одному запиті можуть зустрітися як умови відбору рядків у розділі WHERE, так й умови відбору груп у розділі HAVING. Умови відбору груп не можна перенести з розділу HAVING у розділ WHERE. Аналогічно й умови відбору рядків не можна перенести з розділу WHERE у розділ HAVING, за винятком умов, що включають поля зі списку угруповання GROUP BY.

## 2.6 Використання підзапитів

Дуже зручним засобом, що дозволяє формулювати запити більш зрозумілим чином, є можливість використання підзапитів, вкладених в основний запит.

*Приклад*. Одержати список постачальників, статус яких менше максимального статусу у таблиці постачальників (порівняння з підзапитом):

```
SELECT *
FROM P
WHERE P.STATYS <  (SELECT MAX(P.STATUS)
                   FROM P;
```

Тоді як поле P.STATUS рівняється з результатом підзапиту, то підзапит повинен бути сформульований так, щоб повертати таблицю, що складається рівно з одного рядка й однієї колонки.

Результат виконання запиту буде еквівалентний результату такої послідовності дій:

1. Виконати один раз вкладений підзапит й одержати максимальне значення статусу.

2. Просканувати таблицю постачальників P, щоразу порівнюючи значення статусу постачальника з результатом підзапиту, і відібрати тільки ті рядки, у яких статус менше максимального.

*Приклад*. Використання предиката ***IN***. Одержати перелік постачальників, що поставляють деталь номер 2:

```
SELECT *
FROM P
WHERE P.PNUM IN (SELECT DISTINCT PD.PNUM
                 FROM PD WHERE PD.DNUM = 2);
```

У цьому випадку вкладений підзапит може повертати таблицю, що містить кілька рядків.

Результат виконання запиту буде еквівалентний результату такої послідовності дій:

1. Виконати один раз вкладений підзапит й одержати список номерів постачальників, що поставляють деталь номер 2.

2. Просканувати таблицю постачальників P, щораз перевіряючи, чи втримується номер постачальника в результаті підзапиту.

*Приклад*. Використання предиката ***EXIST***. Одержати перелік постачальників, що поставляють деталь номер 2:

```
SELECT *
FROM P
WHERE EXIST (SELECT *
  FROM PD
  WHERE PD.PNUM = P.PNUM AND PD.DNUM = 2);
```

Результат виконання запиту буде еквівалентний результату такої послідовності дій:

1. Просканувати таблицю постачальників P, щоразу виконуючи підзапит із новим значенням номера постачальника, узятим із таблиці P.

2. У результат запиту включити тільки ті рядки з таблиці постачальників, для яких вкладений підзапит повернув непусту множину рядків.

На відміну від двох попередніх прикладів, вкладений підзапит містить параметр (зовнішнє посилання), переданий з основного запиту - номер постачальника P.PNUM. Такі підзапити називаються ***корельованими*** (***correlated***). Зовнішнє посилання може приймати різні значення для кожного рядка-кандидата, оцінюваного за допомогою підзапиту, тому підзапит повинен виконуватися заново для кожного рядка, який відбирається в основному запиті. Такі підзапити характерні для предиката EXIST, але можуть бути використані й в інших підзапитах.

Може здатися, що запити, які містять корельовані підзапити будуть виконуватися повільніше, ніж запити з некорельованими підзапитами. Насправді це не так, тому що те, як користувач сформулював запит, не визначає, як цей запит буде виконуватися. Мова SQL є не процедурною, а декларативною. Це значить, що користувач, який формулює запит, просто описує, яким повинен бути результат запиту, а як цей результат буде отриманий - за це відповідає сама СКБД.

*Приклад*. Використання предиката ***NOT EXIST***. Одержати перелік постачальників, що не поставляють деталь номер 2:

```
SELECT *
FROM P
WHERE NOT EXIST (SELECT *
  FROM PD
  WHERE PD.PNUM = P.PNUM AND PD.DNUM = 2);
```

Також як й у попередньому прикладі, тут використовується корельований підзапит. Відмінність у тому, що в основному запиті будуть відібрані ті рядки з таблиці постачальників, для яких вкладений підзапит не видасть ні одного рядка.

*Приклад*. Одержати імена постачальників, що поставляють всі деталі:

```
SELECT DISTINCT PNAME  FROM P
WHERE NOT EXIST (SELECT *
  FROM D
  WHERE NOT EXIST  (SELECT *
      FROM PD
 WHERE  PD.DNUM = D.DNUM AND
            PD.PNUM = P.PNUM));
```

Даний запит містить два вкладених підзапитів й реалізує реляційну операцію розподілу відношень.

Самий внутрішній підзапит параметризований двома параметрами (D.DNUM, P.PNUM) і має такий зміст: відібрати всі рядки, що містять дані про поставки постачальника з номером PNUM деталі з номером DNUM. Заперечення NOT EXIST говорить про те, що даний постачальник не поставляє дану деталь. Зовнішній до нього підзапит, сам є вкладеним і параметризованим параметром P.PNUM, має зміст: відібрати перелік деталей, які не поставляються постачальником PNUM. Заперечення NOT EXIST говорить про те, що для постачальника з номером PNUM не повинно бути деталей, які не поставлялися б цим постачальником. Це в точності означає, що в зовнішньому запиті відбираються тільки постачальники, що поставляють всі деталі.

### 2.7 Використання об'єднання, перетинання й різниці

*Приклад*. Одержати імена постачальників, що мають статус, більший 3 або, що поставляють хоча б одну деталь номер 2 (об'єднання двох підзапитів - ключове слово ***UNION***):

```
SELECT P.PNAME
FROM P
WHERE P.STATUS > 3  UNION SELECT P.PNAME
  FROM P, PD
  WHERE P.PNUM = PD.PNUM AND
    PD.DNUM = 2;
```

Результуючі таблиці поєднуваних запитів повинні бути сумісні, тобто мати однакову кількість стовпців й однакові типи стовпців у порядку їхнього перерахування. Не потрібно, щоб поєднувані таблиці мали б однакові імена колонок. Це відрізняє операцію об'єднання запитів у SQL від операції об'єднання у реляційній алгебрі. Найменування колонок у результуючому запиті будуть автоматично взяті з результату першого запиту в об'єднанні.

*Приклад*. Одержати імена постачальників, що мають статус, більший 3 й одночасно поставляють хоча б одну деталь номер 2 (перетинання двох підзапитів - ключове слово ***INTERSECT***):

```
SELECT P.PNAME
FROM P
WHERE P.STATUS > 3 INTERSECT
  SELECT P.PNAME
  FROM P, PD
  WHERE P.PNUM = PD.PNUM AND
            PD.DNUM = 2;
```

*Приклад*. Одержати імена постачальників, що мають статус, більший 3, за винятком тих, хто поставляє хоча б одну деталь номер 2 (різниця двох підзапитів - ключове слово ***EXCEPT***):

```
SELECT P.PNAME
FROM P
WHERE P.STATUS > 3 EXCEPT
  SELECT P.PNAME
   ROM P, PD
```

```
WHERE P.PNUM = PD.PNUM AND
      PD.DNUM = 2;
```

### 2.8 Синтаксис оператора вибірки даних (SELECT). BNF-нотація

Опишемо синтаксис оператора вибірки даних (оператора SELECT) більш точно. При описі синтаксису операторів звичайно використовуються умовні позначки, відомі як *стандартні форми Бекуса-Наура* (*BNF*).

У BNF позначеннях використовуються такі елементи:

- Символ "::=" означає рівність по визначенню. Ліворуч від знака стоїть обумовлене поняття, праворуч - властиво визначенне поняття.
- Ключові слова записуються прописними буквами. Вони зарезервовані й становлять частину оператора.
- Заповнювачі конкретних значень елементів і змінних записуються курсивом.
- Необов'язкові елементи оператора укладені у квадратні дужки [].
- Вертикальна риса | вказує на те, що всі попередні їй елементи списку є необов'язковими й можуть бути замінені будь-яким іншим елементом списку після цієї риси.
- Фігурні дужки {} вказують на те, що все, що міститься усередині них, є єдиним цілим.
- Три крапки "..." означає, що попередня частина оператора може бути повторена будь-яка кількість разів.
- Багато крапок, усередині якого перебуває кома ".,.." вказує, що попередня частина оператора, яка складається з декількох елементів, розділених комами, може мати довільне число повторень. Кому не можна ставити після останнього елемента. Зауваження: дана угода не входить у стандарт BNF, але дозволяє більш точно описати синтаксис операторів SQL.
- Круглі дужки є елементом оператора.

***Синтаксис оператора вибірки***

У досить сильно спрощеному вигляді оператор вибірки даних має такий синтаксис (для деяких елементів ми дамо не BNF-визначення, а словесний опис):

*Оператор вибірки* ::=
*Табличний вираз* [**ORDER BY**
{{*Ім'я стовпця-результату* [**ASC** | **DESC**]} | {*Позитивне ціле* [**ASC** | **DESC**]}.,..];
*Табличне вираз* ::=
*Вираз-Select- вираз* {**UNION** | **INTERSECT** | **EXCEPT**} [**ALL**]
{*Вираз-Select- вираз* | **TABLE** *Ім'я таблиці* | *Конструктор значень таблиці*} ]
*Вираз-Select-вираз* ::=
**SELECT** [**ALL** | **DISTINCT**] {{{*Скалярний вираз* | *Функція агрегування* | *Вираз-Select-вираз*} [**AS** *Ім'я стовпця*]}.,..}
| {{*Ім'я таблиці* | *І'мя кореляції*}.*}
| * **FROM** {
{*Ім'я таблиці* [**AS**] [*Ім'я кореляції*] [(*Ім'я стовпця*.,..)]}
| {*Вираз-Select- вираз* [**AS**] *Ім'я кореляції* [(*Ім'я стовпця*.,..)]}
| *З'єднана таблиця* }.,..
[**WHERE** *Умовний вираз*]
[**GROUP BY** {[{*Ім'я таблиці* | І'*мя кореляції*}.] *Ім'я стовпця*}.,..]
[**HAVING** *Умовний вираз*]

Вираз-Select-вираз у розділі SELECT, який використовується як значення для стовпця, який відбирається, повинен повертати таблицю, що складається з одного рядка й одного стовпця, тобто скалярний вираз.

Умовний вираз у розділі WHERE повинен обчислюватися для кожного рядка, що є кандидатом у результуючу множину рядків. У цьому умовному виразі можна використовувати підзапити. Синтаксис умовних виразів, припустимих у розділі WHERE розглядається нижче.

Розділ HAVING містить умовний вираз, що обчислюється для кожної групи, обумовленої переліком угруповання в розділі GROUP BY. Цей умовний вираз може містити функції агрегування, що обчислюються для кожної групи. Умовний вираз, сформульований у розділі WHERE, може бути перенесений до розділу HAVING. Перенос умов з розділу HAVING у розділ WHERE неможливий, якщо умовний вираз містить агрегатні функції.

Якщо в розділі SELECT присутні агрегатні функції, то вони обчислюються по-різному залежно від наявності розділу GROUP BY. Якщо розділ GROUP BY відсутній, то результат запиту повертає не більше одного рядка. Агрегатні функції обчислюються по всіх рядках, які задовольняють умовному виразу в розділі WHERE. Якщо розділ GROUP BY є присутнім, то агрегатні функції обчислюються окремо для кожної групи, зазначеної у розділі GROUP BY.

Скалярний вираз - як скалярні вирази у розділі SELECT можуть виступати або імена стовпців таблиць, що входять у розділ FROM або прості функції, що повертають скалярні значення.

*Функція агрегування* ::=

**COUNT (*)** | { {**COUNT** | **MAX** | **MIN** | **SUM** | **AVG**} ([**ALL** | **DISTINCT**] *Скалярний вираз*) }

*Конструктор значень таблиці* ::=

**VALUES** *Конструктор значень рядка*.,..

*Конструктор значень рядка* ::=

*Елемент конструктора* | (*Елемент конструктора*.,..) | *Вираз-Select- вираз*

Вираз-Select-вираз, який використовується у конструкторі значень рядка, зобов'язаний повертати рівно один рядок.

*Елемент конструктора* ::=

*Вираз для обчислення значення* | **NULL** | **DEFAULT**

***Синтаксис з'єднаних таблиць***

У розділі FROM оператора SELECT можна використовувати з'єднані таблиці. Нехай у результаті деяких операцій ми одержуємо таблиці A й B. Такими операціями можуть бути, наприклад, оператор SELECT або інша з'єднана таблиця. Тоді синтаксис з'єднаної таблиці має такий вигляд:

*З'єднана таблиця* ::=

*Перехресне з'єднання* | *Природне з'єднання* | *З'єднання за допомогою предиката* | *З'єднання за допомогою імен стовпців* | *З'єднання об'єднання*

*Тип з'єднання* ::=

**INNER** | **LEFT** [**OUTER**] | **RIGTH** [**OUTER**] | **FULL** [**OUTER**]

*Перехресне з'єднання* ::=

*Таблиця A* **CROSS JOIN** *Таблиця B*

*Природне з'єднання* ::=

*Таблиця A* [**NATURAL**] [*Тип з'єднання*] **JOIN** *Таблиця B*

*З'єднання за допомогою предиката* ::=

*Таблиця A* [*Тип з'єднання*] **JOIN** *Таблиця B* **ON** Предикат

*З'єднання за допомогою імен стовпців* ::=

*Таблиця A* [*Тип з'єднання*] **JOIN** *Таблиця B* **USING** (Ім'я стовпця.,..)

*З'єднання об'єднання* ::=

*Таблиця A* **UNION JOIN** *Таблиця B*

Опишемо використані терміни.

CROSS JOIN - перехресне з'єднання повертає просто декартовий добуток таблиць. Таке з'єднання в розділі FROM може бути замінено списком таблиць через кому.

NATURAL JOIN - природне з'єднання відбувається по всіх стовпцях таблиць A і B, що мають однакові імена. У результуючу таблицю однакові стовпці уставляються тільки один раз.

JOIN ... ON - з'єднання за допомогою предиката з'єднує рядки таблиць A і B за допомогою зазначеного предиката.

JOIN ... USING - з'єднання за допомогою імен стовпців з'єднує відношення подібно природному з'єднанню по тим загальним стовпцям таблиць А і Б, які зазначені в списку USING.

OUTER - ключове слово OUTER (зовнішній) не є обов'язковими, воно не використовується ні в яких операціях з даними.

INNER - тип з'єднання "внутрішнє". Внутрішній тип з'єднання використовується за замовчуванням, коли тип явно не заданий. У таблицях А і В з'єднуються тільки ті рядки, для яких знайден збіг.

LEFT (OUTER) - тип з'єднання "ліве (зовнішнє)". Ліве з'єднання таблиць А і В містить у собі всі рядки з лівої таблиці А і ті рядки із правої таблиці В, для яких виявлений збіг. Для рядків з таблиці А, для яких не знайдено відповідності у таблиці В, у стовпці, що витягають із таблиці В, заносяться значення NULL.

RIGHT (OUTER) - тип з'єднання "праве (зовнішнє)". Праве з'єднання таблиць А і В містить у собі всі рядки із правої таблиці В і ті рядки з лівої таблиці А, для яких виявлений збіг. Для рядків з таблиці В, для яких не знайдено відповідності в таблиці А, у стовпці, що витягають із таблиці А, заносяться значення NULL.

FULL (OUTER) - тип з'єднання "повне (зовнішнє)". Це комбінація лівого й правого з'єднань. У повне з'єднання включаються всі рядки з обох таблиць. Для співпадаючих рядків поля заповнюються реальними значеннями, для незбіжних рядків поля заповнюються відповідно до правил лівого й правого з'єднань.

UNION JOIN - з'єднання об'єднання є зворотним стосовно внутрішнього з'єднання. Воно включає тільки ті рядки з таблиць А і В, для яких не знайдено збігів. У них використаються значення NULL для стовпців, отриманих з іншої таблиці. Якщо взяти повне зовнішнє з'єднання й видалити з нього рядки, отримані в результаті внутрішнього з'єднання, то вийде з'єднання об'єднання.

Використання з'єднаних таблиць часто полегшує сприйняття оператора SELECT, особливо, коли використовується природне з'єднання. Якщо не використовувати з'єднані таблиці, то при виборі даних з декількох таблиць необхідно явно вказувати умови з'єднання в розділі WHERE. Якщо при цьому користувач указує складні критерії відбору рядків, то в розділі WHERE змішуються семантично різні поняття - як умови зв'язку таблиць, так й умови відбору рядків.

***Синтаксис умовних виразів розділу WHERE***

Умовний вираз, використовуваний в розділі WHERE оператора SELECT повинен обчислюватися для кожного рядка-кандидата, який відбирається оператором SELECT. Умовний вираз може повертати одне із трьох значень істинності: TRUE, FALSE або UNKNOUN. Рядок-кандидат відбирається до результуючої множини рядків тільки у тому випадку, якщо для неї умовний вираз повернуло значення TRUE.

Умовні вирази мають такий синтаксис (з метою спрощення викладки наведені не всі можливі предикати):

*Умовний вираз* ::=

[ ( ] [**NOT**] {*Предикат порівняння* | *Предикат between* | *Предикат in* | *Предикат like* | *Предикат null* | *Предикат кількісного порівняння* | *Предикат exist* | *Предикат unique* | *Предикат match* | *Предикат overlaps*} [{**AND** | **OR**} *Умовний вираз*] [ ) ] [**IS** [**NOT**] {**TRUE** | **FALSE** | **UNKNOWN**}]

*Предикат порівняння* ::=

*Конструктор значень рядка* {= | < | > | <= | >= | <>} *Конструктор значень рядка*

*Приклад*. Порівняння поля таблиці й скалярного значення:

POSTAV.VOLUME > 100

*Приклад*. Порівняння двох сконструйованих рядків:

(PD.PNUM, PD.DNUM) = (1, 25)

Цей приклад еквівалентний умовному виразу

PD.PNUM = 1 AND PD.DNUM = 25

*Предикат between* ::=

*Конструктор значень рядка* [**NOT**] **BETWEEN** *Конструктор значень рядка* **AND** *Конструктор значень рядка*

*Приклад*. PD.VOLUME BETWEEN 10 AND 100

*Предикат in* ::=

*Конструктор значень рядка* [**NOT**] **IN** {(*Select-вираз*) | (*Вираз для обчислення значення*.,..)}

*Приклад*.

P.PNUM IN (SELECT PD.PNUM FROM PD WHERE PD.DNUM=2)

*Приклад*.

P.PNUM IN (1, 2, 3, 5)

*Предикат like* ::=

*Вираз для обчислення значення рядка-пошуку* [**NOT**] **LIKE**

*Вираз для обчислення значення рядка-шаблона* [**ESCAPE** *Символ*]

Предикат LIKE робить пошук рядка-пошуку в рядку-шаблоні. У рядку-шаблоні дозволяється використовувати два трафаретних символи:

- Символ підкреслення "_" може використовуватися замість будь-якого одиничного символу в рядку-пошуку,
- Символ відсотка "%" може заміняти набір будь-яких символів у рядку-пошуку (число символів у наборі може бути від 0 і більше).

*Предикат null* ::=

*Конструктор значень рядка* **IS** [**NOT**] **NULL**

Зауваження. Предикат NULL застосовується спеціально для перевірки, чи не дорівнює вираз null-значенню.

*Предикат кількісного порівняння* ::=

*Конструктор значень рядка* {= | < | > | <= | >= | <>} {**ANY** | **SOME** | **ALL**} (*Select-вираз*)

Квантори ANY й SOME є синонімами й повністю взаємозамінні. Якщо зазначено один із кванторів ANY й SOME, то предикат кількісного порівняння повертає TRUE, якщо порівнюване значення збігається хоча б з одним значенням, що повертається у підзапиті (select-виразі). Якщо зазначено квантор ALL, то предикат кількісного порівняння повертає TRUE, якщо порівнюване значення збігається з кожним значенням, що повертається у підзапиті (select-вираз).

*Приклад*.

P.PNUM = SOME (SELECT PD.PNUM FROM PD WHERE PD.DNUM=2)

*Предикат exist* ::=

**EXIST** (*Select-вираз*)

Предикат EXIST повертає значення TRUE, якщо результат підзапиту (select-вираз) не порожній.

*Предикат unique* ::=

**UNIQUE** (*Select-вираз*)

Предикат UNIQUE повертає TRUE, якщо в результаті підзапиту (select-вираз) немає співпадаючих рядків.

*Предикат match* ::=

*Конструктор значень рядка* **MATCH** [**UNIQUE**] [**PARTIAL** | **FULL**] (*Select-вираз*)

Предикат MATCH перевіряє, чи буде значення, зазначене в конструкторі рядка збігатися зі значенням будь-якого рядка, отриманого у результаті підзапиту.

*Предикат overlaps* ::=

*Конструктор значень рядка* **OVERLAPS** *Конструктор значень рядка*

Предикат OVERLAPS, є спеціалізованим предикатом, що дозволяє визначити, чи буде зазначений період часу перекривати інший період часу.

## 2.9 Порядок виконання оператора SELECT

Для того щоб зрозуміти, як виходить результат виконання оператора SELECT, розглянемо концептуальну схему його виконання. Ця схема є саме концептуальною, тому що гарантується, що результат буде таким, якби він виконувався крок за кроком відповідно до цієї схеми.

*Стадія 1*. Виконання одиночного оператора SELECT

Якщо в операторі присутні ключові слова UNION, EXCEPT й INTERSECT, то запит розбивається на кілька незалежних запитів, кожний з яких виконується окремо:

*Крок 1 (FROM)*. Обчислюється прямий декартовий добуток всіх таблиць, зазначених в обов'язковому розділі FROM. У результаті кроку 1 одержуємо таблицю A.

*Крок 2 (WHERE)*. Якщо в операторі SELECT є присутнім розділ WHERE, то сканується таблиця A, отримана при виконанні кроку 1. При цьому для кожного рядка з таблиці A обчислюється умовний вираз, наведений у розділі WHERE. Тільки ті рядки, для яких умовний вираз повертає значення TRUE, включаються в результат. Якщо розділ WHERE опущений, то відразу переходимо до кроку 3. Якщо в умовному виразі беруть участь вкладені підзапити, то вони обчислюються відповідно до даної концептуальної схеми. У результаті кроку 2 одержуємо таблицю B.

*Крок 3 (GROUP BY)*. Якщо в операторі SELECT є присутнім розділ GROUP BY, то рядки таблиці B, отриманої на другому кроці, групуються у відповідності зі списком групування, наведеним у розділі GROUP BY. Якщо розділ GROUP BY опущений, то відразу переходимо до кроку 4. У результаті кроку 3 одержуємо таблицю C.

*Крок 4 (HAVING)*. Якщо в операторі SELECT є присутнім розділ HAVING, то групи, що не задовольняють умовному виразу, наведеному в розділі HAVING, виключаються. Якщо розділ HAVING опущений, то відразу переходимо до кроку 5. У результаті кроку 4 одержуємо таблицю D.

*Крок 5 (SELECT)*. Кожна група, отримана на кроці 4, генерує один рядок результату в такий спосіб. Обчислюються всі скалярні вирази, зазначені у розділі SELECT. За правилами використання розділу GROUP BY, такі скалярні вирази повинні бути однаковими для всіх рядків усередині кожної групи. Для кожної групи обчислюються значення агрегатних функцій, наведених у розділі SELECT. Якщо розділ GROUP BY був відсутній, але в розділі SELECT є агрегатні функції, то вважається, що є всього одна група. Якщо немає ні GROUP BY, ні агрегатних функцій, то вважається, що є стільки груп, скільки рядків відібрано до даного моменту. У результаті кроку 5 одержуємо таблицю E, що містить стільки колонок, скільки елементів наведено в розділі SELECT і стільки рядків, скільки відібрано груп.

*Стадія 2*. Виконання операцій UNION, EXCEPT, INTERSECT

Якщо в операторі SELECT були присутні ключові слова UNION, EXCEPT й INTERSECT, то таблиці, отримані в результаті виконання 1-ої стадії, поєднуються, віднімаються або перетинаються.

*Стадія 3*. Впорядкування результату

Якщо в операторі SELECT є присутнім розділ ORDER BY, то рядки, отриманої на попередніх кроках таблиці, впорядковуються у відповідності зі списком упорядкування, наведеному у розділі ORDER BY.

Якщо уважно розглянути наведений вище концептуальний алгоритм обчислення результату оператора SELECT, то відразу зрозуміло, що виконувати його безпосередньо в такому вигляді надзвичайно складно. Навіть на найпершому кроці, коли обчислюється декартовий добуток таблиць, наведених у розділі FROM, може вийти таблиця величезних розмірів, причому практично більшість рядків і колонок з неї буде відкинуто на наступних кроках.

Насправді в РСКБД є *оптимізатор*, функцією якого є знаходження такого оптимального алгоритму виконання запиту, що гарантує одержання правильного результату. Схематично роботу оптимізатора можна подати у вигляді послідовності декількох кроків:

Крок 1 (Синтаксичний аналіз). Запит, який надійшов, піддається синтаксичному аналізу. На цьому кроці визначається, чи правильно взагалі (з погляду синтаксису SQL) сформульований запит. У ході синтаксичного аналізу виробляється деяке внутрішньє подання запиту, використовуване на наступних кроках.

Крок 2 (Перетворення в канонічну форму). Запит у внутрішньому поданні піддається перетворенню у деяку канонічну форму. При перетворенні до канонічної форми використовуються як синтаксичні, так і семантичні перетворення. Синтаксичні перетворення (наприклад, приведення логічних виразів до кон'юнктивної або диз'юнктивної нормальної форми, заміна виразів "x AND NOT x" на "FALSE", і т.п.) дозволяють одержати нове внутрішнє подання запиту, синтаксично еквівалентне вихідному, але стандартне у деякому змісті. Семантичні перетворення використовують додаткові знання, якими володіє система, наприклад, обмеження цілісності. У результаті семантичних перетворень виходить запит, синтаксично не еквівалентний вихідному, але результат, що дає, той самий.

Крок 3 (Генерація планів виконання запиту й вибір оптимального плану). На цьому кроці оптимізатор генерує безліч можливих планів виконання запиту. Кожен план будується як комбінація низькорівневих процедур доступу до даних з таблиць, методам з'єднання таблиць. Зі всіх згенерованих планів вибирається план, який має мінімальні затрати. При цьому аналізуються дані про наявність індексів у таблиць, статистичних даних про розподіл значень у таблицях, і т.п. Вартість плану це, як правило, сума вартостей виконання окремих низькорівневих процедур, які використовуються для його виконання. У вартість виконання окремої процедури можуть входити оцінки кількості звертань до дисків, ступінь завантаженості процесора й інші параметри.

Крок 4. (Виконання плану запиту). На цьому кроці план, обраний на попередньому кроці, передається на реальне виконання.

Багато в чому якість конкретної СКБД визначається якістю її оптимізатора. Гарний оптимізатор може підвищити швидкість виконання запиту на кілька порядків. Якість оптимізатора визначається тим, які методи перетворень він може використовувати, яку статистичну й іншу інформацію про таблиці він має, які методи для оцінки вартості виконання плану він знає.

### 2.10 Реалізація реляційної алгебри засобами оператора SELECT (Реляційна повнота SQL)

Основні об'єкти реляційної БД

Кластери, каталоги й схеми не є обов'язковими елементами стандарту й, отже, програмного середовища реляційних БД.

Під *кластером* розуміється група каталогів, до яких можна звертатися через одне з'єднання із сервером БД (програмний компонент СКБД).

На практиці процедура створення каталогу визначається реалізацією СКБД на конкретній операційній платформі. Під *каталогом* розуміється група схем. На практиці каталог часто асоціюється з фізичною базою даних як набором фізичних файлів операційної системи, які ідентифікуються її ім'ям.

Для розробника БД *схема* - це загальне логічне подання відношень закінченої БД. З погляду SQL, *схема* - це контейнер для таблиць, подань та інших структурних елементів реляційної БД. Принцип розміщення елементів БД у кожній схемі повністю визначається розробником БД.

Для створення таблиць і подань наявність схеми не обов'язкова. Якщо у вас планується інсталяція тільки однієї логічної БД, то ясно, що можна обійтися й без схеми. Але якщо планується, що та сама СКБД буде використовуватися для підтримки декількох БД, то належна організація об'єктів БД у схеми може значно полегшити супровід цих баз даних..

Термін схема (Schema) використовується для опису всіх об'єктів БД, які створені деяким користувачем. Для кожного нового користувача автоматично створюється нова схема.

До числа основних об'єктів реляційних БД відносяться таблиця, подання й користувач.

*Таблиця* (Table) є базовою структурою БД. Вона являє собою одиницю зберігання даних - відношення. Таблиця ідентифікується в БД своїм унікальним ім'ям, що містить у собі ідентифікацію користувача. Таблиця може бути порожньою або складатися з набору рядків.

*Подання* (View) - це пойменована динамічно підтримувана СКБД вибірка з однієї або декількох таблиць БД. Оператор вибірки обмежує видимі користувачем дані. Звичайно СКБД гарантує актуальність подання - його формування виробляється щораз, коли подання використовується. Іноді подання називають віртуальними таблицями.

*Користувач* (User) - це об'єкт, що володіє можливістю створювати або використовувати інші об'єкти БД і запитувати виконання функцій СКБД, таких як організація сеансу роботи, зміна стану БД і т.д.

Для спрощення ідентифікації й іменування об'єктів у БД підтримуються такі об'єкти, як синонім, послідовність і певні користувачем типи даних.

*Синонім* (Synonym) - це альтернативне ім'я об'єкта (псевдонім) БД, що дозволяє мати доступ до даного об'єкта. Синонім може бути загальним і часткою. Загальний синонім дозволяє всім користувачам БД звертатися до відповідного об'єкта по його псевдоніму. Синонім дозволяє сховати від кінцевих користувачів повну кваліфікацію об'єкта в БД.

*Послідовність* (Sequence) - це об'єкт БД, що дозволяє генерувати послідовність унікальних чисел (номерів) в умовах асинхронного доступу багатьох користувачів. Звичайно елементи послідовності використовуються для унікальної нумерації елементів таблиць (рядків) в операціях модифікації даних.

Певні користувачем *типи даних* (User-defined data types) являють собою певні користувачем типи атрибутів (домени), які відрізняються від підтримуваних (убудованих) СКБД типів. Вони визначаються на основі убудованих типів. Певні користувачем типи даних утворять ту частину середовища СКБД, що організована відповідно до об'єктно-орієнтованої парадигми.

Для забезпечення ефективного доступу до даних у реляційних СКБД підтримуються ряд інших об'єктів: індекс, таблична область, кластер, секція.

*Індекс* (Index) - це об'єкт БД, який створений для підвищення продуктивності вибірки даних і контролю унікальності первинного ключа (якщо він заданий для таблиці). Повністю індексні таблиці (index-organized tables) виконують роль таблиці й індексу одночасно.

*Табличний простір* або *область* (Tablespace) - це іменована частина БД, яка використовується для розподілу пам'яті для таблиць й індексів. Всі об'єкти БД, у яких зберігаються дані, відповідають деяким табличним просторам. Більшість об'єктів БД, у яких дані не зберігаються, перебувають у словнику даних, розміщеному в табличному просторі SYSTEM.

*Кластер* (Cluster) - це об'єкт, що задає спосіб спільного зберігання даних у декількох або одній таблиці. Одним із критеріїв використання кластера є наявність загальних ключових полів у декількох таблицях, які використовуються в одній і тій самій команді SQL.

*Секція* (Partition) - це об'єкт БД, що дозволяє подати об'єкт із даними у вигляді сукупності під'обєктів, віднесених до різних табличних просторів. Таким чином, секціонування дозволяє розподіляти дуже великі таблиці на декількох жорстких дисках.

Для обробки даних спеціальним чином або для реалізації підтримки посилальної цілісності БД використовуються об'єкти: збережена процедура, функція, команда, тригер, таймер і пакет (Oracle). За допомогою цих об'єктів БД можна виконувати так звану порядкову обробку (record processing) даних. З погляду додатків баз даних порядкова обробка - це послідовна вибірка даних по одному рядку, її обробка й перехід до обробки наступного рядка.

Дані об'єкти реляційної БД являють собою програми, тобто виконує код.

*Збережена процедура* (Stored procedure) - це об'єкт БД, що представляє пойменований набір команд SQL й/або операторів спеціалізованих мов обробки програмування БД.

*Функція* (Function) - це об'єкт БД, що представляє пойменований набір команд SQL й/або операторів спеціалізованих мов обробки програмування БД, що при виконанні повертає значення - результат обчислень.

*Команда* (Command) - це пойменований оператор SQL, що заздалегідь відкомпільований і зберігається в БД. Швидкість обробки команди вища, ніж у відповідні йому оператора SQL, тому що при цьому не виконуються фази синтаксичного розбору й компіляції.

*Тригер* (Trigger) - це об'єкт БД, що являє собою спеціальну збережену процедуру. Ця процедура запускається автоматично, коли відбувається пов'язана із тригером подія (наприклад, до вставки рядка в таблицю).

*Таймер* (Timer) відрізняється від тригера тим, що подія, що запускається, для збереженої процедури є подія таймера.

*Пакет* (Package) - це об'єкт БД, що складається з пойменованого структурованого набору змінних, процедур і функцій.

***Оператор декартового добутку***

Реляційна алгебра: A Times B

Оператор SQL: SELECT A.Поле1, A.Поле2, ..., B.Поле1, B.Поле2, ...FROM A, B;

або SELECT A.Поле1, A.Поле2, ..., B.Поле1, B.Поле2, ...FROM A CROSS JOIN B;

*Оператор проекції*

Реляційна алгебра: A [X,Y,..,Z]

```
Оператор SQL:    SELECT DISTINCT X, Y, ..., Z
                 FROM A;
```

*Оператор вибірки*

```
Реляційна алгебра: A Where C,
Оператор SQL:    SELECT *
                 FROM A
                 WHERE C;
```

*Оператор об'єднання*

```
Реляційна алгебра: A Union B
Оператор SQL:    SELECT *
                 FROM A
                 UNION SELECT *
                 FROM B;
```

*Оператор віднімання*

```
Реляційна алгебра:A Minus B
Оператор SQL:    SELECT *
                 FROM A
                 EXCEPT SELECT * FROM B
```

Реляційний оператор перейменування RENAME виражається за допомогою ключового слова AS у переліку полів оператора, які відбираються, SELECT. Таким чином, мова SQL є реляційно повною.

Інші оператори реляційної алгебри (з'єднання, перетинання, розподіл) виражаються через примітивні, отже, можуть бути виражені операторами SQL. Проте, для практичних цілей наведемо їх.

***Оператор з'єднання***

Реляційна алгебра: (A Times B) Where C

Оператор SQL: SELECT A.Поле1, A.Поле2, ..., B.Поле1, B.Поле2, ...FROM A, B WHERE C;

або SELECT A.Поле1, A.Поле2, ..., B.Поле1, B.Поле2, ... FROM A CROSS JOIN B WHERE C;

*Оператор перетинання*

Реляційна алгебра: A Intersect B

Оператор SQL:
```
SELECT *
FROM A
INTERSECT SELECT * FROM B;
```

*Оператор розподілу*

Реляційна алгебра: A (X,Y) Devid By B(Y)

Оператор SQL:
```
SELECT DISTINCT A.X
FROM A
WHERE NOT EXIST (SELECT * FROM B WHERE NOT EXIST
(SELECT * FROM A A1
WHERE A1.X = A.X AND
A1.Y = B.Y));
```

Оператор SQL, що реалізує розподіл відношень важко запам'ятати, тому дамо приклад еквівалентного перетворення виразів, які представляють зміст запиту.

Нехай відношення A містить дані про поставки деталей, відношення B містить перелік всіх деталей, які можуть поставлятися. Атрибут X є номером постачальника, атрибут Y є номером деталі.

Розділити відношення A на відношення B означає в даному прикладі "відібрати номера постачальників, які поставляють всі деталі".

Перетворимо текст виразу: "Відібрати номера постачальників, які поставляють всі деталі" еквівалентно "Відібрати ті номери постачальників з таблиці A, для яких не існує деталей, що не поставляють, у таблиці B" еквівалентно "Відібрати ті номери постачальників з таблиці A, для яких не існує тих номерів деталей з таблиці B, які не поставляються цим постачальником" еквівалентно "Відібрати ті номери постачальників з таблиці A, для яких не існує тих номерів деталей з таблиці B, для яких не існує записів про поставки в таблиці A для цього постачальника й цієї деталі".

Останній вираз дослівно переводиться на мову SQL. При перекладі вираз на мову SQL необхідно врахувати, що у внутрішньому підзапиті таблиця A повинна бути перейменована, для того щоб відрізняти її від екземпляра цієї ж таблиці, яка використовується у зовнішньому запиті.

## 3 Основи проектування додатків баз даних

### 3.1 Функціональна модель ODBC

#### 3.1.1 Основа ODBC

Інтерфейс ODBC (Open Database Connectivity) був розроблений фірмою Microsoft як відкритий інтерфейс доступу до БД. Він надає уніфіковані засоби взаємодії прикладної програми, названої клієнтом (або додатком-клієнтом), із сервером - БД.

В основу інтерфейсу ODBC були покладені специфікація CLI-інтерфейсу (Call-Level Interface), розроблена X/Open, і ISO/IEC для API БД, а також мова SQL (Structured Query Language) як стандарт мови доступу до БД.

Інтерфейс ODBC проектувався для забезпечення уніфікованого доступу будь-якого додатка, який використовує ODBC, до різних джерел даних. Так, якщо додаток, який відповідає стандарту ODBC й SQL, спочатку розроблявся для роботи з БД Microsoft Access, а потім таблиці цієї бази були перенесені в БД Microsoft SQL Server або БД Oracle, то додаток зможе й далі обробляти ці дані без внесення додаткових змін.

Для взаємодії з БД додаток-клієнт викликає функції інтерфейсу ODBC, які реалізовані в спеціальних модулях, названих ODBC-драйверами. Як правило, ODBC-драйвери - це DLL-бібліотеки, при цьому одна DLL-бібліотека може підтримувати кілька ODBC-драйверів. При установці на комп'ютер будь-якого SQL-сервера автоматично виконується реєстрація в реєстрі Windows і відповідного ODBC-драйвера.

#### 3.1.2 Архітектура ODBC

Архітектура ODBC подана чотирма компонентами (рис. 3.1):

1. Додаток-клієнт, що виконує виклик функцій ODBC.
2. Менеджер драйверів, що завантажує й звільняє ODBC-драйвери, які потрібні для додатків-клієнтів. Менеджер драйверів обробляє виклики ODBC-функцій або передає їхньому драйверу.
3. ODBC-драйвер, що обробляє виклики SQL-функцій, передаючи SQL-серверу виконуваний SQL-оператор, а додатку-клієнтові - результат виконання викликаної функції.
4. Джерело даних, обумовлений як конкретна локальна або вилучена БД.

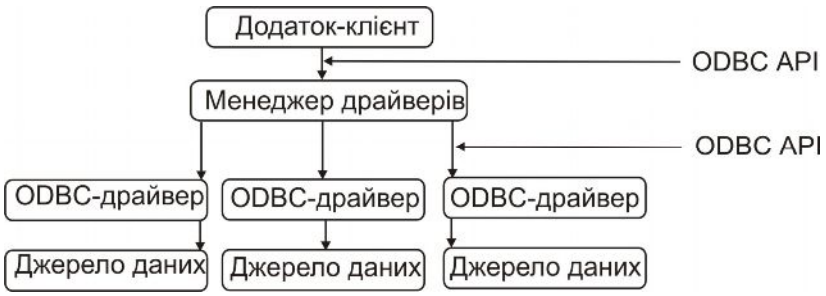



Рисунок 3.1 – Архітектура ODBC

Основне призначення менеджера драйверів - завантаження драйвера, відповідному джерелу даних, що підключається, і інкапсуляція взаємодії з різними типами джерел даних за допомогою застосування різних ODBC-драйверів.

ODBC-драйвери, приймаючи виклики функцій, взаємодіють із додатком-клієнтом, виконуючи такі завдання:

- керування комунікаційними протоколами між додатком-клієнтом і джерелом даних;
- керування запитами до СКБД;
- виконання передачі даних від додатка-клієнта в СКБД і з БД у додаток-клієнт;
- повернення додатку-клієнтові стандартної інформації про виконаний виклик ODBC-функції у вигляді коду повернення;
- підтримує роботу з курсорами й управляє транзакціями.

Додаток-клієнт одночасно може встановлювати з'єднання з декількома різними джерелами даних, використовуючи різні ODBC-драйвери, а також кілька з'єднань із тим самим джерелом даних, використовуючи той самий ODBC-драйвер.

### 1.3.3 Співвідношення стандарту ODBC і стандарту інтерфейсу рівня викликів (CLI)

Як ми вже відзначали вище, відкритий інтерфейс доступу до БД фірми Microsoft заснований на таких стандартах:

специфікація X/Open CAE (Specification "Data Management: SQL Call-Level Interface (CLI)");

специфікація ISO /IEC 9075-3:1995 (E) (Call-Level Interface (SQL/CLI)).

У цей час фірма Microsoft підтримує версію 3.x ODBC API. Додатки, написані на основі специфікації X/Open й ISO CLI, будуть правильно працювати з ODBC-драйверами версії 3.x або драйверами "погодженого стандарту" у тому випадку, якщо вони компілюються із заголовними файлами ODBC версії 3.x і лінкуються з ODBC 3.x бібліотеками, а доступ до ODBC-драйвера одержують через менеджер драйверів ODBC 3.x. Аналогічно, що й самі драйвери 3.x, написані на основі специфікації X/Open й ISO CLI, будуть правильно працювати з додатками при дотриманні цих самих умов.

Драйвер ODBC 3.x завжди підтримує всі можливості, які використовуються додатком "погодженого стандарту", а додаток ODBC 3, що використовує тільки можливості, надані ISO CLI, і обов'язкові засоби, описувані X/Open CLI, завжди буде працювати із драйвером "погодженого стандарту".

На додаток до інтерфейсу, специфікованому у стандартах ISO/IEC й X/Open CLI, ODBC реалізує такі можливості:

- добування декількох рядків (блокова вибірка) за один виклик функції;
- зв'язування з масивом параметрів;
- підтримка закладок, включаючи вибірку за допомогою закладки, закладки змінної довжини, блокове відновлення й видалення за допомогою відзначених операцій над непослідовними рядками;
- порядкове зв'язування (row-wise binding);
- зв'язування зі зсувом (binding offsets);
- підтримка пакетів SQL-операторів як у збережених процедурах, так й у вигляді послідовності окремих SQL-операторів, виконуваних при виклику функцій SQLExecute й SQLExecDirect;
- визначення точного або приблизного числа рядків курсору;
- застосування операції позиціонованого відновлення й видалення й пакетні видалення й відновлень із використанням функції SQLSetPos;
- підтримка функцій каталогу, що дозволяють одержувати інформацію зі схеми БД (системних таблиць);
- бібліотеки перетворення для кодових сторінок;
- асинхронне виконання;
- підтримка збережених процедур, включаючи escape-послідовності, механізм зв'язування вихідних параметрів, функції каталогу;
- більше просунуті можливості з'єднання, що включають підтримку атрибутів з'єднання й перегляду атрибутів.

### 1.3.4 Функції ODBC API

Всі функції ODBC API умовно можна поділити на чотири групи:

- основні функції ODBC, що забезпечують взаємодію з джерелом даних;
- функції установки (setup DLL);
- функції інсталяції (installer DLL) ODBC і джерел даних;
- функції перетворення даних (translation DLL), які викликаються при передачі даних від драйвера до джерела даних або назад.

Оголошення всіх функцій і використовуваних ними типів даних утримуються в заголовних файлах. Група основних функцій ODBC API розбита на три рівні: функції ядра ODBC; функції 1 рівня; функції 2 рівня.

Кожен ODBC-драйвер специфікується як драйвер, що підтримує певний рівень функцій ODBC API.

У таблиці 3.1 поданий список основних функцій ODBC API.

Група функцій установки (setup DLL) поєднує функції, призначені для конфігурування драйверів і джерел даних:

- ConfigDriver виконує установку або видалення драйвера;
- ConfigDSN виконує додавання, зміну або видалення джерела даних;
- ConfigTranslator повертає використовувані за замовчуванням опції перетворення.

Таблиця 3.1 –Основні функції ODBC API

| Функція | Опис |
|---|---|
| 1 | 2 |
| З'єднання із джерелом даних | |
| SQLAllocHandle | Одержує ідентифікатор (дескриптор) середовища, з'єднання або оператора, або дескриптор додатка |
| SQLConnect | З'єднання із джерелом даних по DSN, імені й паролю користувача |
| SQLDriverConnect | З'єднання із джерелом даних по зазначеному рядку з'єднання або за допомогою відображуваного діалогу для інтерактивного уведення параметрів з'єднання |
| SQLBrowseConnect | Послідовно запитує атрибути з'єднання й установлює допустимі значення атрибута. Після специфікації значення для кожного необхідного атрибута з'єднання функція виконує з'єднання із джерелом даних |
| Одержання інформації про драйвери й про джерела даних | |
| SQLDataSources | Повертає список доступних джерел даних |
| SQLDrivers | Повертає список установлених драйверів та їхні атрибути |
| SQLGetInfo | Повертає інформацію про зазначений драйвер й джерело даних |

Продовження таблиці 3.1

| 1 | 2 |
|---|---|
| SQLGetFunctions | Повертає функції, які підтримуються використовуваним драйвером |
| SQLGetTypeInfo | Повертає інформацію про підтримувані типи даних |
| Зміна атрибутів драйверів й одержання інформації про атрибути драйверів | |
| SQLSetConnectAttr | Установлює атрибути з'єднання |
| SQLGetConnectAttr | Повертає значення атрибута з'єднання |
| SQLSetEnvAttr | Установлює атрибути середовища |
| SQLGetEnvAttr | Повертає значення атрибута середовища |
| SQLSetStmtAttr | Установлює атрибути оператора |
| SQLGetStmtAttr | Повертає значення атрибута оператора |

Група функцій інсталяції (installer DLL) поєднує функції, призначені для установки ODBC і конфігурування джерел даних.

Установка ODBC

- SQLConfigDriver завантажує setup DLL для конкретного драйвера;
- SQLGetInstalledDrivers повертає список установлених драйверів;
- SQLInstallDriverEx додає до реєстру дані про драйвер;
- SQLInstallDriverManager повертає каталог, призначений для менеджера драйверів;
- SQLInstallerError повертає інформацію про помилку виконання функції інсталяції;
- SQLInstallTranslatorEx додає до реєстру дані про транслятор;
- SQLRemoveDriver видаляє з реєстру дані про драйвер;
- SQLRemoveDriverManager змінює або видаляє дані про базові компоненти ODBC з реєстру;
- SQLRemoveTranslator видаляє з реєстру дані про транслятор.

Конфігурування джерел даних:

- SQLConfigDataSource викликає setup DLL для конкретного драйвера;
- SQLCreateDataSource відображає діалог для додавання джерела даних;
- SQLGetConfigMode запитує режим конфігурації, що дозволяє визначити, де в реєстрі Windows шукати секцію ODBC.INI. ;
- SQLGetPrivateProfileString записує значення до реєстру;
- SQLGetTranslator показує діалог для вибору транслятора;
- SQLManageDataSources відображає діалог для конфігурування драйверів і джерел даних;
- SQLReadFileDSN читає інформацію про DSN з файлу;
- SQLRemoveDefaultDataSource видаляє джерело даних за замовчуванням;
- SQLRemoveDSNFromIni видаляє джерело даних;
- SQLSetConfigMode установлює режим конфігурації, що вказує, де в реєстрі буде використатися вхід ODBC.INI;
- SQLValidDSN перевіряє правильність і'мя джерела даних;
- SQLWriteDSNToIni додає джерело даних;
- SQLWriteFileDSN записує інформацію про DSN у файл;
- SQLWritePrivateProfileString запитує значення з реєстру Windows.

### 3.1.4 Схема доступу до джерела даних з використанням ODBC API

Першим кроком при реалізації доступу до джерела даних за допомогою ODBC API без застосування пула з'єднань є створення дескриптора (ідентифікатора) оточення. Після виділення пам’яті під дескриптор оточення додаток повинен викликати функцію SQLSetEnvAttr для завдання значення атрибуту дескриптора оточення SQL_ATTR_ODBC_VERSION..

Після створення дескриптора оточення створюються дескриптори з'єднань. Кожен дескриптор з'єднання формується викликом функції SQLAllocHandle з типом дескриптора, рівним SQL_HANDLE_DBC. Драйвер виділяє пам'ять для зберігання інформації про з'єднання й повертає значення дескриптора з'єднання. Далі для реального з'єднання із джерелом даних викликається функція SQLConnect або функція SQLDriverConnect.

Для виконання SQL-операторів створюються дескриптори операторів. Вони дозволяють одержати доступ до інформації про виконаного оператора, ім'я курсору й атрибути. Дескриптор оператора формується викликом функції SQLAllocHandle зі значенням типу дескриптора, таким що дорівнює SQL_HANDLE_STMT.

При створенні дескриптора оператора драйвер автоматично створює ще чотири дескриптори й записує покажчики на них в атрибути дескриптора оператора SQL_ATTR_APP_ROW_DESC, SQL_ATTR_APP_PARAM_DESC, SQL_ATTR_IMP_ROW_DESC й SQL_ATTR_IMP_PARAM_DESC. Ці чотири дескриптори називаються неявно розміщеними дескрипторами.

Для явного розміщення дескриптора додатка варто викликати функцію SQLAllocHandle зі значенням типу дескриптора, таким що дорівнює SQL_HANDLE_DESC. При цьому формується дескриптор, названий явно розміщеним дескриптором.

Явно розміщені дескриптори також асоціюються з дескриптором з'єднання: вони залишаються доступними доти, поки додаток має з'єднання з БД. Оскільки явно розміщені дескриптори асоціюються з дескриптором з'єднання, то додаток може асоціювати такі дескриптори з декількома дескрипторами операторів для даного з'єднання. Неявно розміщений дескриптор додатка не може бути асоційований більш ніж з одним дескриптором оператора.

### 3.1.5 Послідовність дій додатка-клієнта для реалізації доступу до джерела даних

***З'єднання із джерелом даних***

Для безпосереднього підключення до БД ODBC API надає такі три функції: SQLConnect - з'єднання з джерелом даних по DSN, імені й паролю користувача; SQLDriverConnect - з'єднання з джерелом даних по зазначеному рядку з'єднання або за допомогою відображуваного діалогу для інтерактивного уведення параметрів з'єднання; SQLBrowseConnect - з'єднання з джерелом даних із попереднім послідовним запитом атрибутів з'єднання.

Функція SQLConnect має такий формальний опис:

```
SQLRETURN SQLConnect(
SQLHDBC ConnectionHandle, - вказує дескриптор
        з'єднання;
SQLCHAR *     ServerName, - ім'я джерела даних;
SQLSMALLINT   NameLength1, - визначають довжину
        параметрів;
SQLCHAR *     UserName,  - описують ім'я користувача;
SQLSMALLINT   NameLength2, - визначають довжину
        параметрів;
SQLCHAR *     Authentication, - описують пароль;
SQLSMALLINT NameLength3); - визначають довжину
        параметрів.
```

Для виконання з'єднання з джерелом даних, що вимагають для підключення додаткової інформації, або відображення перед підключенням діалогу з уточненням значення параметрів використовується функція SQLDriverConnect, що має такий формальний опис:

SQLRETURN SQLDriverConnect(;
SQLHDBC ConnectionHandle, - вказує дескриптор з'єднання;
SQLHWND WindowHandle, - це покажчик батьківського вікна або NULL;
SQLCHAR * InConnectionString, - задає повністю або частково рядок з'єднання або порожній рядок;
SQLSMALLINT StringLength1, - задає довжину в байтах;
SQLCHAR * OutConnectionString, - це покажчик на буфер, у якому після успішного підключення до джерела даних повертається повний рядок з'єднання;
SQLSMALLINT BufferLength, - задається розмір буфера;
SQLSMALLINT *StringLength2Ptr, - повертає покажчик на буфер, в якому розміщується загальне число символів повного рядка з'єднання;
SQLUSMALLINT DriverCompletion); - це прапорець, що вказує, чи буде менеджер драйверів і драйвер пропонувати діалоги для формування завершеного рядка з'єднання.

Останній параметр визначається такими значеннями:

SQL_DRIVER_PROMPT - підказка пропонується й ураховується навіть у тому випадку, якщо значення атрибута вже задано в рядку з'єднання. Спочатку відображається вікно з доступними джерелами даних (для атрибута DSN);

SQL_DRIVER_COMPLETE - підказка пропонується тільки в тому випадку, якщо необхідний для підключення атрибут не заданий у рядку з'єднання;

SQL_DRIVER_COMPLETE_REQUIRED - підказка пропонується тільки в тому випадку, якщо необхідний для підключення атрибут не заданий у рядку з'єднання й при цьому запитуються тільки необхідні значення атрибутів;

SQL_DRIVER_NOPROMPT - підказки не пропонуються.

Набір ключових слів, що вказують у рядку з'єднання, частково залежить від використаного драйвера. До загальноприйнятих ключових слів відносяться такі:

DSN - ім'я джерела даних (функція SQLDataSources повертає список доступних джерел даних);

FILEDSN - ім'я .dsn файлу, з якого буде прочитаний рядок з'єднання;

DRIVER - опис драйвера (список доступних драйверів повертається функцією SQLDrivers);

UID - ідентифікатор користувача;

PWD - для зазначеного ідентифікатора користувача або при відсутності пароля порожній рядок (PWD=;);

SAVEFILE - ім'я .dsn файлу, у який буде записаний рядок з'єднання, що використовується для даного успішного підключення до джерела даних.

Ключові слова DSN й FILEDSN є в рядку з'єднання взаємовиключними: буде використане перше із зазначених. З іншими ключовими словами FILEDSN не є взаємовиключним: пріоритет має значення, зазначене безпосередньо в рядку стану. Очевидно, що значення ключового слова PWD не зберігається в .dsn файлі.

Функція SQLBrowseConnect реалізує ітераційний метод запиту значень атрибутів, необхідних для підключення до БД, повертаючи щоразу код відповіді SQL_NEED_DATA й ідентифікатор чергового запитуваного атрибута. Після визначення значень всіх необхідних атрибутів функція встановлює з'єднання з БД і при успішному завершенні операції повертає код відповіді, такий, що дорівнює SQL_SUCCESS або SQL_SUCCESS_WITH_INFO.

Функція SQLBrowseConnect має такий формальний опис:

SQLRETURN SQLBrowseConnect(;
SQLHDBC ConnectionHandle;
SQLCHAR * InConnectionString, - описує рядок підключення або її частину, зазначену при попередньому виклику функції;
SQLSMALLINT StringLength1, - задає довжину буфера;
SQLCHAR * OutConnectionString, - визначає покажчик на буфер, що містить інформацію про відсутній атрибут рядка з'єднання;
SQLSMALLINT BufferLength, - задає довжину буфера;
SQLSMALLINT *StringLength2Ptr); - указує загальне число байтів, що повинне бути повернуте в буфері.

***Пул з'єднань***

Організація пула з'єднань дозволяє додатку вибирати з'єднання з пула і без необхідності переустановлювати їх для кожного використання.

Використання з'єднань, поміщених у пул, може значно збільшити продуктивність додатків, які багаторазово встановлюють й розривають з'єднання. Прикладом таких додатків можуть служити серверні Інтернет-додатки середньої ланки, що постійно повторно встановлюють і розривають з'єднання.

З'єднання з пула можуть бути використані декількома компонентами в одному процесі. Це означає, що автономні компоненти в одному процесі можуть взаємодіяти один з одним без повідомлення один одного. З'єднання з пула може бути використане повторно декількома компонентами.

При роботі з пулом з'єднань використаний драйвер ODBC повинен бути повністю потокозахищеним, що дозволить одночасно виконувати різні виклики з різних потоків (наприклад, виконати приєднання в одному потоці, використати з'єднання в іншому потоці, а від'єднання виконати в третьому потоці).

Пул з'єднань управляється менеджером драйверів. З'єднання вибирається з пула при виклику додатком функції SQLConnect або функції SQLDriverConnect, а повертається в пул при виконанні функції SQLDisconnect. Розмір пула змінюється динамічно: якщо з'єднання не було використано протягом певного періоду часу, то воно віддаляється з пула.

Для використання додатком пула з'єднань необхідно:

для включення режиму пула з'єднань викликати функцію SQLSetEnvAttr з атрибутом середовища SQL_ATTR_CONNECTION_POOLING, такими, що дорівнює значенню SQL_CP_ONE_PER_DRIVER або значенню SQL_CP_ONE_PER_HENV. При виклику функції SQLSetEnvAttr дескриптор середовища вказується таким, що дорівнює значенню NULL, що визначає атрибут SQL_ATTR_CONNECTION_POOLING як атрибут рівня процесу. Значення SQL_CP_ONE_PER_DRIVER визначає, що окремий пул з'єднань підтримується для кожного драйвера. При необхідності мати один пул для різних драйверів указується значення SQL_CP_ONE_PER_HENV (окремий пул з'єднань підтримується для кожного середовища).

Створити дескриптор з'єднання, викликавши функцію SQLAllocHandle зі значенням параметра типу дескриптора, таким, що дорівнює SQL_HANDLE_DBC. Менеджер драйверів буде шукати існуюче розподілене середовище з відповідними атрибутами середовища (при знаходженні необхідного середовища воно повертається додатку й менеджер драйверів збільшує значення лічильника на 1). Якщо такого середовища немає, то менеджер драйверів створює його й установлює значення лічильника таким, що дорівнює 1.

Для одержання з'єднання з пула викликати функцію SQLConnect або SQLDriverConnect. Менеджер драйверів використовує значення параметрів і значення атрибутів з'єднання для визначення необхідного з'єднання з пула з'єднань. При цьому враховується значення атрибута SQL_ATTR_CP_MATCH (відповідність необхідного з'єднання з'єднанню з пула).

Для розриву з'єднання викликати функцію SQLDisconnect. При цьому з'єднання повертається назад у пул і робиться доступним для подальшого використання.

## 3.2 Об'єктна модель OLE DB

### 3.2.1 Основні поняття

OLE DB являє собою набір COM-інтерфейсів (Component Object Model), які надають клієнту-додатку-клієнтові уніфікований доступ до різних джерел даних. Можна сказати, що OLE DB - це метод доступу до будь-яких даних через стандартні COM-інтерфейси, поза залежністю від типу даних і місця їх розміщення. Як дані можуть виступати БД, прості документи, таблиці Excel і будь-які інші джерела даних. На відміну від доступу, наданого за допомогою драйверів OBDC, OLE DB дозволяє реалізовувати доступ до джерел даних, як із застосуванням мови SQL (до SQL-серверів), так і до будь-яких інших довільних джерел даних.

Засоби, що надають доступ до джерела даних із використанням технології OLE DB, називаються OLE DB провайдерами. Програми-клієнти, що використовують для доступу OLE DB провайдери, називаються споживачами даних. У тому випадку, якщо існує тільки ODBC-драйвер для доступу до конкретного джерела даних, то для застосування технології OLE DB можна використати OLE DB провайдер, призначений для доступу до ODBC-джерела даних.

Через те, що архітектура OLE DB заснована на COM, те механізм створення результуючих наборів складається з послідовностей кроків типу: 1. створення об'єкта → 2. запит покажчика на інтерфейс створеного об'єкта → 3. виклик методу інтерфейсу.

Аналогічно комплексу дій, якому слідують після створення результуючого набору при застосуванні технології ODBC - виконанню зв'язування, у технології OLE DB використовується механізм аксесорів. Аксесори описують, яким чином дані записуються в область пам'яті споживача даних, установлюючи адресну відповідність між областю пам'яті в буфері споживача даних і стовпцями даних у результуючому наборі. Іноді такий набір зв'язків називають картою стовпців (column map).

***Об'єктна модель OLE DB***

Специфікація OLE DB описує набір інтерфейсів, реалізованих об'єктами OLE DB. Кожен об'єктний тип визначений як набір інтерфейсів. Специфікація OLE DB визначає набір інтерфейсів базового рівня, які повинні реалізовуватися будь-якими OLE DB провайдерами.

У базову модель OLE DB входять такі об'єкти:

- об'єкт DataSource (джерело даних), використовується для з'єднання з джерелом даних і створення одного або декількох сеансів. Цей об'єкт управляє з'єднанням, використовує інформацію про повноваження й аутентифікацію користувача;
- об'єкт Session (сеанс) управляє взаємодією з джерелом даних - виконує запити й створює результуючі набори. Сеанс також може повертати метадані. У сеансі може створюватися одна або кілька команд;
- об'єкт Rowset (результуючий набір) являє собою дані, що витягають у результаті виконання команди або створювані в сеансі.

На наступній схемі (рис. 3.2) наведений приклад використання інтерфейсів базового рівня для створення результуючого набору.

Специфікація OLE DB визначає об'єкт Command (команда), призначений для виконання текстової команди. Як така команда може виступати й SQL-оператор. При цьому виконання команди може створювати результуючий набір (у випадку SQL-оператора - це оператор SELECT).

Деякі OLE DB провайдери підтримують роботу зі схемою (Schema), що надає метадані по БД. Метадані стають доступні як звичайні результуючі набори. У заголовному файлі oledb.h утримуються унікальні ідентифікатори всіх доступних типів результуючих наборів схеми даних (наприклад, для одержання інформації з таблиць БД варто вказати унікальний ідентифікатор DBSCHEMA_TABLES). Стовпець результуючого набору з ім'ям TABLE_NAME містить ім'я таблиці, стовпець TABLE_TYPE указує один із таких типів

таблиці: ALIAS, TABLE, SYNONYM, SYSTEM TABLE, VIEW, GLOBAL TEMPORARY, LOCAL TEMPORARY, SYSTEM VIEW.

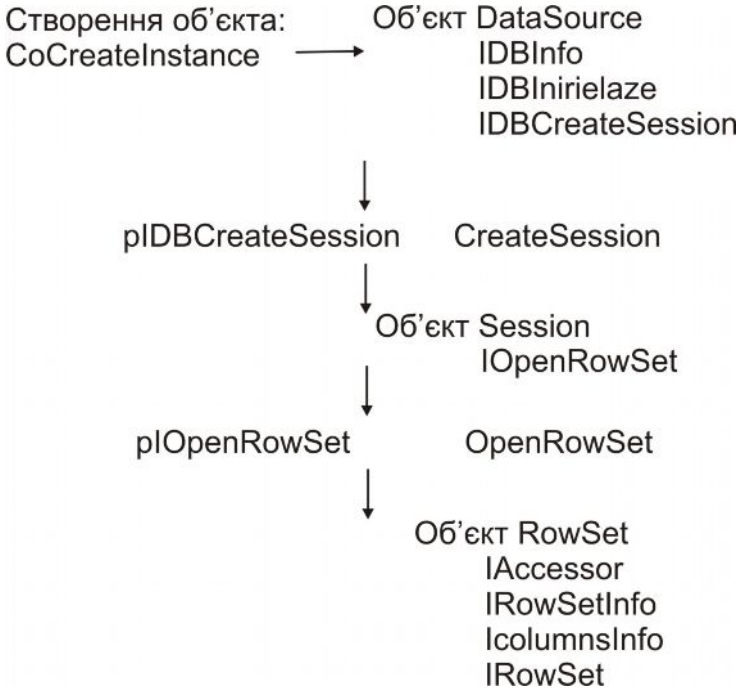



Рисунок 3.2 – Схема створення результуючого набору

Подання (View) визначає підмножину рядків і стовпців із набору даних, але саме не містить їх. Подання не можуть поєднувати дані з декількох наборів даних.

Для забезпечення розширених можливостей керування транзакціями об'єктна модель OLE DB включає об'єкт Transaction.

Для кожного об'єктного типу специфікація OLE DB визначає набір інтерфейсів, що повинен обов'язково бути реалізований для даного об'єкта. Такі інтерфейси відзначаються як [mandatory]. Інтерфейси, які можуть бути відсутніми, відзначаються як [optional].

Усі об'єкти об'єктного типу Rowset повинні реалізовувати такі інтерфейси:

інтерфейс IRowset, який використовується для добування рядків;

інтерфейс IAccessor, який використовуєтсья для визначення зв'язування;

інтерфейс IColumnsInfo, що надає інформацію про стовпці результуючого набору;

інтерфейс IRowsetInfo, що надає інформацію про самий результуючий набір;

інтерфейс IConvertType, що надає інформацію про перетворення типів даних, підтримуваних у результуючому наборі.

### 3.2.2 Створення результуючого набору

При реалізації доступу до БД за допомогою OLE DB провайдера спочатку варто створити об'єкт даних й установити з'єднання з БД. Далі необхідно створити об'єкт "сеанс". І тільки потім можна створювати результуючий набір.

Механізм створення об'єкта "сеанс" наведений на схемі (рис. 3.3).

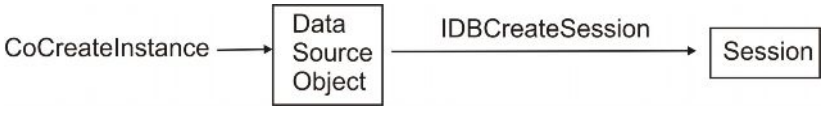



Рисунок 3.3 – Схема механізму створення об'єкта «сеанс»

Результуючий набір може бути створений одним із таких способів:

Для об'єкта "сеанс" викликається метод IOpenRowset::OpenRowset, що виконує безпосереднє створення результуючого набору (інтерфейс IOpenRowset повинен підтримуватися будь-яким провайдером);

Для об'єкта "сеанс" викликається метод IDBCreateCommand::CreateCommand, що створює об'єкт Command. Далі для об'єкта "команда" викликається метод ICommand::Execute. (при використанні інтерфейсу IMultipleResults можна працювати з декількома результуючими наборами);

Викликається один із наступних методів IColumnsRowset::GetColumnsRowset, IDBSchemaRowset::GetRowset, IViewRowset::OpenViewRowset або ISourcesRowset::GetSourcesRowset.

Щоб результуючий набір, збережений на сервері, можна було використовувати, необхідно виконати зв'язування й добування даних. Для цього слід визначити структури типу DBBINDING, що описують стовпці, і створити аксесор. Далі для одержання рядків результуючого набору можна використати один з наступних методів:

IRowset::GetNextRows;

IRowsetLocate::GetRowsByBookMarks;

IRowsetLocate::GetRowAt;

IRowsetScroll:: GetRowAtRatio.

На закінчення для запису даних у структуру, визначену аксесором, викликається метод IRowset::GetData.

Після одержання й обробки рядків їх варто звільнити, викликавши метод IRowset::ReleaseRows.

Після перегляду всього результуючого набору слід також звільнити аксесор, викликавши метод IRowset::ReleaseAccessor, і звільнити сам результуючий набір, викликавши метод IRowset::Release.

Інтерфейс IAccessor визначає такі методи:

AddRefAccessor - збільшує число посилань на даний аксесор;

CreateAccessor - створює аксесор з набору зв'язувань;

GetBindings - повертає зв'язування, установлені даним аксесором;

ReleaseAccessor - звільняє аксесор.

Об'єкт Command повинен реалізовувати такі інтерфейси:

Icommand;

Iaccessor;

IcommandText;

IcolumnInfo;

ICommandProperties.

Для створення команди викликається метод IDBCreateCommand::CreateCommand об'єкта "сеанс".

Алгоритм виконання команди наведений на схемі (рис. 3.4).

### 3.2.3 Об'єкти TRANSACTION

Застосування OLE DB дозволяє підтримувати прості, вкладені й розподілені транзакції.

Об'єкт Session для роботи із транзакціями підтримує наступні інтерфейси:

інтерфейс ITransactionLocal.

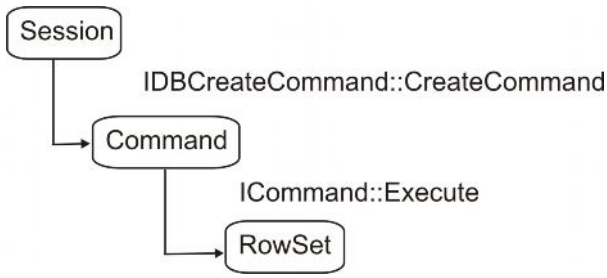



Рисунок 3.4 – Схема виконання методу IDBCreateCommand::CreateCommand

Для початку транзакції викликається метод ITransactionLocal::StartTransaction(). Якщо цей метод викликається з активної транзакції, то відкривається нова вкладена транзакція;

інтерфейс ITransaction, що підтримує методи Abort, Commit й GetTransactionInfo;

інтерфейс ITransactionJoin, що реалізує підтримку розподілених транзакцій.

Об'єкт Transaction дозволяє реалізовувати більш широкі можливості керування транзакціями, підтримуючи такі інтерфейси:

ITransaction, що дозволяє виконати переривання транзакції (методи Abort, Commit, GetTransactionInfo);

IConnectionPointContainer, що підтримує керування крапками з'єднання для об'єктів, що з'єднують.

## 3.3 Реалізація доступу до БД у середовищі DELPHI

### 3.3.1 Механізми доступу до БД

VCL-бібліотека класів середовища проектування Delphi надає ряд класів, що дозволяють швидко й ефективно розробляти різні додатки БД. Ці класи подані такими групами:

- компоненти для доступу до даних, реалізуючі: доступ через машину БД BDE (Borland Database Engine), який надає доступ через ODBC-драйвери або через внутрішні драйвери машини БД 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 вилученого SQL-сервера InterBase (компоненти сторінки InterBase); доступ за допомогою легковагих драйверів dbExpress; доступ до БД при багатоланковій архітектурі (компоненти сторінки DataSnap);
- візуальні компоненти, що реалізують інтерфейс користувача;
- компоненти для зв'язку джерел даних із візуальними компонентами, які надають інтерфейс користувача;
- компоненти для візуального проектування звітів.

Основними механізмами доступу до даних, підтримуваними 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 використовують легковагі драйвери БД;
- засоби доступу до розподілених наборів даних у багатоланковій архітектурі.

Найпростіший механізм керування даними, що використовують ODBC-драйвери, може бути реалізований за такою 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 зазначений одним із трьох способів: по імені БД, 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 й 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 розміщені 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 джерела даних, що вказується властивістю DataSource. Ім'я поля набору даних визначається властивістю DataField.

Графічно схему роботи з базами даних для дволанкових архітектур у середовищі Delphi можна представити в спосіб, який показано на рис. 3.5.

Для збереження даних із БД в XML-форматі або двійковому форматі, і назад, для формування набору даних з XML або двійкового файлу застосовується провайдер даних.

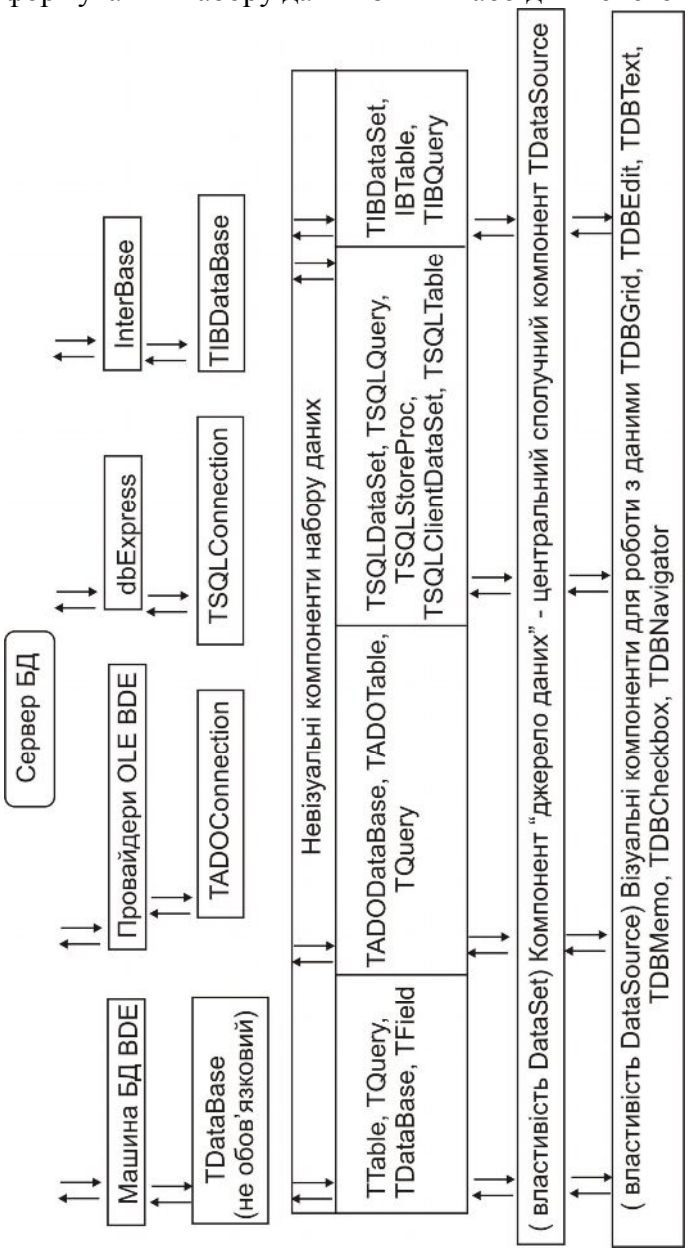



Рисунок 3.5 – Схема роботи з БД у середовищі Delphi

### 3.3.2 Набори даних

Базою всіх класів наборів даних є клас TDataSet. Він визначає основу структури всіх наборів даних - масив компонентів типу TField (кожен елемент масиву відповідає стовпцю таблиці).

*Набір даних* - це впорядкована послідовність рядків, витягнутих із джерела даних. Кожен рядок набору даних складається з полів, що вказують у властивостях класу.

Залежно від механізму доступу, який використовується додатком, базовими класами набору даних можуть бути:

TTable, TQuery, TStoredProc - для одноланкових або дволанкових додатків, які використовують машину БД BDE. Клас TQuery додатково дозволяє виконувати параметричні запити;

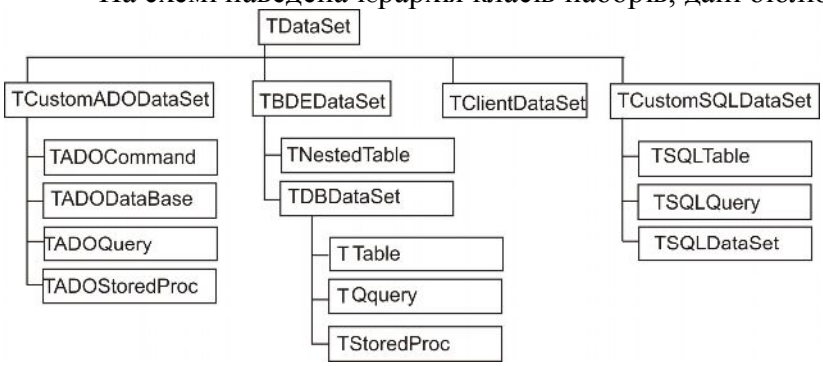
TClientDataSet - для реалізації клієнтського набору даних і для багатоланкової архітектури, яка використовує розподілений доступ;

TADODataSet - для додатків, які використовують ADO-об'єкти;

TSQLDataSet - для доступу до БД 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 й First. Редагування записів у спрямованому наборі даних можливо тільки явним виконанням SQL-оператора UPDATE або при установці з'єднання з клієнтським набором даних через провайдера;

TSQLTable й TSQLQuery - для доступу до БД за допомогою dbExpress.

На схемі наведена ієрархія класів наборів, дані бібліотеки VCL:




Для визначення набору даних необхідно задати такі властивості:

- для класу TTable - значення властивостей DatabaseName й TableName;
- для класу TQuery - значення властивості SQL й, 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 із наступних способів: установити значення властивості Active набору даних рівним True під час виконання додатка (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 таким, що дорівнює 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 й AfterOpen; установку стану набору даних у 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, які реалізують з'єднання з БД. Однак деякі можливості, такі як керування транзакціями або кешировані відновлення, неможливі без компонентів типу TDatabase або TADOConnection. Компонент "БД" TDatabase застосовується для з'єднання із джерелом даних через драйвери BDE або зовнішні ODBC-драйвери. Компонент TADOConnection використовується для створення об'єкта "з'єднання" при доступі через OLE DB, що інкапсулюються за допомогою ADO-об'єктів VCL-бібліотеки.

За замовчуванням при переході від одного запису набору даних до іншої відбувається запис всіх зроблених змін у БД. Для того, щоб можна було скасовувати зроблені зміни або виконувати відновлення декількох записів, застосовують кешировані 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 іншого інформація у БД не передається. Щоб включити режим кешированого відновлення, необхідно встановити значення властивості CachedUpdates таким, що дорівнює True для компонента набору даних. Для присвоєння кешированного відновлення викликається метод ApplyUpdates, а для скасування - CancelUpdates.

### 3.3.3 Класи бібліотеки VCL

***Клас TDATASET*** є базовим для всіх класів наборів даних, які успадковують загальні властивості й методи цього класу, включаючи такі:

- Active - властивість, що визначає, чи відкритий набір даних;
- CurrentRecord - властивість, що визначає номер поточного запису набору даних;
- DataSource - властивість, що вказує батьківську таблицю (для таблиць, зв'язаних відношенням батьківська - дочірня);
- Bof - властивість, що визначає, чи перебуває курсор на першому записі набору даних;
- Eof - властивість, що визначає, чи досягнуть кінець набору даних;
- FieldCount - властивість, що вказує кількість полів у наборі даних;
- Bookmark - властивість, що вказує поточну закладку в наборі даних. Закладка відзначає позицію в наборі даних. Використовуючи методи TDataSet.GetBookmark й TDataSet.GotoBookmark, додаток може запам'ятовувати й швидко переходити на потрібну позицію в наборі даних;
- Fields - властивість, що представляє собою масив полів набору даних і використовуване для доступу до цих полів. Властивість Fields дозволяє одержати ім'я поля в поточній структурі запису:

```
var  S: String;
begin
S := Fields[0].FieldName;// Ім'я першого поля
S := Fields[1].FieldName;// Ім'я другого поля
...
end;
```

записати в змінну значення поля.

```
var  s: String; i: Integer; d: TDateTime;
s := Fields[0].AsString;
i := Fields[0].AsInteger;
d := Fields[0].AsDate;
```

- Filter - властивість, в яку заноситься рядок, що визначає фільтр для набору даних. Фільтр визначає умова, якій повинні задовольняти доступні записи.

Визначення фільтра повинне задовольняти таким правилам:

фільтр складається з умов для полів набору даних, об'єднаних логічними операціями AND й OR. Наприклад: F2 > 10 AND F2 <50;

якщо ім'я поля містить пробіли, то воно повинне бути укладене у квадратні дужки або подвійні лапки. Наприклад: [Field Name1] > 50;

Filtered - властивість, що вказує, чи використовується фільтр, заданий властивістю Filter;

Found - властивість, що визначає, чи успішно виконаний пошук методами FindFirst, FindLast, FindNext або FindPrior;

Modified - властивість, що визначає, чи був змінений активний запис;

RecordCount - властивість, що містить загальне число записів у наборі даних;

- State - властивість, що вказує поточний стан набору даних. Ця властивість може приймати такі значення:

dsInactive - набір даних закритий;

dsBrowse - дані доступні тільки для перегляду;

dsEdi - можна змінювати активний запис;

dsInsert - активним записом є новий запис доти, поки не буде збережена;

dsSetKey - перегляд обмеженої безлічі записів (SetRange) або пошук запису;

dsCalcFields - виконується оброблювач події OnCalcFields;

dsFilter - виконується оброблювач події OnFilterRecord;

dsOpening - набір даних перебуває в процесі відкриття.

- Append - метод, що додає в кінець набору даних новий запис;
- Delete - метод, що видаляє поточний запис із БД. Якщо в момент видалення запису набір даних перебуває в неактивному стані, то ініціюється виключення;
- Edit - метод, що переводить поточний запис у режим редагування;
- Cancel - метод, що скасовує зміни, зроблені в поточному записі;
- Post - метод, що виконує внесення змін у БД;
- Refresh - метод, що виконує відновлення результуючого набору шляхом повторного добування даних із БД;
- Insert - метод, що вставляє в набір даних новий запис;
- InsertRecord - метод, що вставляє в набір даних новий запис зі значеннями, які зазначені параметрами методу;
- Close - метод, що закриває набір даних;
- Open - метод, що відкриває набір даних;
- First - метод, що встановлює курсор на перший запис набору даних і робить цей запис активним;
- Last - метод, що встановлює курсор на останній запис набору даних і робить цей запис активним;
- Next - метод, що переміщає курсор на наступний запис набору даних і робить цей запис активним;
- Prior - метод, що переміщає курсор на попередній запис набору даних і робить цей запис активним.

***Клас TDATASOURCE*** реалізує зв'язок між компонентами - наборами даних й елементами керування, які використовуються для відображення даних.

При побудові відношення між таблицями "батьківська-дочірня" компонентів "джерело даних" служить для зв'язування наборів даних, указуючи батьківський набір даних.

Клас TDataSource містить набір властивостей і методів, що використовуються для доступу до набору даних, включаючи наступні:

- AutoEdit - властивість, що визначає, чи буде автоматично викликатися метод Edit набору даних при одержанні фокуса елементом керування, асоційованим із джерелом даних;
- DataSet - властивість, що вказує на використовуваний набір даних.

Змінюючи значення властивості DataSet під час виконання, можна ефективно перемикатися на роботу з різними наборами даних, відображаючи різні набори даних у тих самих елементах керування.

DataSource.DataSet := Table1;.

- Enabled - властивість, що визначає, чи буде елемент керування відображати асоційовані з ним дані, або буде відображатися порожнім;
- State - властивість, що дозволяє визначити стан використовуваного набору даних.

```
if DataSource1.Dataset <> nil then
  //Кнопка доступна тільки в тому випадку, якщо набір
  //даних перебуває в стані редагування
  //або вставки нового запису
```

```
BtnPost1.Enabled := DataSource1.State in [dsEdit, dsInsert];
```

***Клас TTABLE*** використовується для доступу до БД за допомогою визначення джерела даних DSN й імені таблиці БД. При цьому допускається вибір всіх полів таблиці або тільки частини полів, а також завдання фільтра, що визначає, які рядки таблиці будуть доступні.

Компоненти типу TTable можуть використовувати всі властивості й методи, наслідувані від класу TDataSet, а також властивості й методи класу TTable для набору даних, включаючи такі:

- DatabaseName - властивість, що визначає ім'я джерела даних DSN;
- CanModify - властивість, що визначає, чи може додаток виконувати вставку, редагування й видалення записів у таблиці;
- DefaultIndex - властивість, що визначає, чи повинні дані в таблиці бути впорядковані при її відкритті. Якщо значення властивості дорівнює True (за замовчуванням), то виконується впорядкування за первинним ключем або унікальним індексом;
- IndexName - властивість, що дозволяє визначити вторинний індекс, використовуваний для сортування набору даних, які відкриваються;
- Exclusive - властивість, що дозволяє встановити винятковий режим доступу до таблиці (значення властивості повинне бути визначене до відкриття таблиці);
- MasterSource - властивість, що визначає ім'я компонента "джерело даних" батьківської таблиці для встановлення відносини між таблицями "батьківська-дочірня";
- MasterFields - властивість, що визначає одне або кілька полів із батьківської таблиці, службовців для зв'язку з відповідними полями даної дочірньої таблиці (це задає відношення між батьківською й дочірньою таблицями. Поля в списку розділяються крапкою з комою);
- ReadOnly - властивість, що дозволяє встановити для таблиці режим доступу "тільки для читання";
- TableName - властивість, що вказує використовувану таблицю БД;
- RecNo - властивість, що вказує номер поточного запису набору даних;
- FindKey - метод, що виконує пошук значення або значень, перерахованих у списку, для ключового поля;
- FindNearest - метод, що переміщає курсор на запис, що містить значення, найбільш близьке до зазначеного значення ключового поля (пошук може виконуватися як по одному значенню, так і по декількох, якщо використовується складений індекс);
- Locate - метод, який використовується для пошуку першого входження значення зазначеного поля або набору полів (якщо запис знайдений, то вона стає поточною).

***Клас TQUERY*** дозволяє виконувати будь-який SQL-оператор, припустимий по синтаксису ODBC-драйвером. Якщо використовується SQL-оператор SELECT, то компонент повертає набір даних (результуючий набір). На відміну від класу Ttable, клас TQuery дозволяє створювати набори даних з декількох таблиць, а також обмежувати одержуваний набір даних певними умовами. Це скасовує необхідність добування всіх записів таблиці в набір даних, що, у свою чергу, заощаджує пам'ять, скорочує мережний трафік для вилучених БД і зменшує час доступу.

Для визначення набору даних TQuery варто встановити значення властивості SQL й, можливо, властивості DatabaseName (властивість DatabaseName визначає ім'я джерела даних, але для деяких БД можна задати повне ім'я таблиці, що включає місце розміщенння файлу, у тексті SQL-оператора, - у цьому випадку властивість DatabaseName не використовується). Найбільш правильним підходом все-таки варто вважати той, при якому ім'я DSN джерела даних указується властивістю DatabaseName, а в SQL-операторі визначається тільки ім'я таблиці без визначення її місця розміщення.

За замовчуванням набір даних, сформований компонентом типу TQuery, не редагується. Для того, щоб значення в створеному наборі даних можна було редагувати, необхідно виконати одне з таких дій:

зв'язати компонент TQuery з компонентом типу TUpdateSQL (наприклад: Query1. Query1.UpdateObject:= UpdateSQL1;) і визначити для останнього значення властивості ModifySQL (наприклад: update TBL1 set F1 = :F1, F2 = :F2 where F3 = :OLD_F3);

установити значення властивості RequestLive таким, що дорівнює True (підтримка цієї можливості залежить від використовуваної БД).

- Клас TQuery містить властивості й методи, які використовуються для роботи з набором даних, включаючи такі:
- DataSource - властивість, що дозволяє вказати батьківський набір даних (для відношення "батьківський-дочірній").

Наприклад, якщо властивість SQL містить значення 'SELECT * FROM Tbl1 t WHERE (t.FNo = :FNo)', те значення змінного зв'язку :FNo буде визначатися із джерела даних, зазначеного властивістю DataSource.

- Params - властивість, що містить список параметрів для SQL-оператора.
- RequestLive - властивість, що визначає, чи буде можливість редагувати створюваний набір даних (можливість одержання що модифікує результуючого набору залежить від використовуваного SQL-сервера);
- SQL - властивість, що містить текст SQL-оператора (для автоматичного формування SQL-оператора можна викликати з контекстного меню компонента TQuery діалог SQL Builder);
- DatabaseName - властивість, що визначає ім'я джерела даних, які підключаються, (ім'я DSN джерела даних або ім'я, уведене класом типу TDatabase);
- ExecSQL - метод, що виконує SQL-оператор, зазначений властивістю SQL (для SQL-оператора, що створює набір даних, замість ExecSQL використовується метод Open);
- ExecSQL можна викликати для таких SQL-операторів, як INSERT, UPDATE, DELETE, CREATE TABLE і т.п.

Якщо перед викликом ExecSQL не був викликаний метод Prepare, то SQL-оператор буде одночасно й відкомпільований, і виконаний.

- Prepare - метод, що виконує компіляцію SQL-оператора.

Виклик цього методу перед ExecSQL збільшує швидкість виконання запиту при багаторазовому повторенні викликів ExecSQL для того самого оператора (наприклад, параметризованого запиту). Це дозволяє відкомпілювати SQL-оператор тільки один раз, а потім багаторазово його виконувати.

***Клас TSQLTABLE*** подає таблицю БД, доступну для клієнта як спрямований набір даних. Такий набір містить всі записи для полів, певних у класі TSQLTable. Об'єкт типу TSQLTable повинен бути пов'язаний з об'єктом типу TSQLConnection, що визначає з'єднання із джерелом даних. Для відображення такого набору даних не можна використовувати таблицю, тому що в клієнта відсутній кеш пам'яті для набору даних. Значення полів таблиці можна відображати компонентами типу TDBText або TDBEdit. Для переміщень по набору записів доступні тільки методи First й Next.

Після розміщення в модулі даних або на формі компонента треба виконати такі дії: установити значення властивості SQLConnection компонента TSQLTable, вибравши доданий раніше компонент типу TSQLConnection із пропонованого списку; визначити ім'я таблиці БД, використаної для побудови набору даних, визначивши властивість TableName компонента TSQLTable.

***Клас TUPDATESQL*** дозволяє для наборів даних, створених з доступом "тільки для читання", підтримувати можливість їхнього відновлення за допомогою виконання SQL-оператора.

Клас TUpdateSQL реалізує такі властивості й методи:

- DeleteSQL - властивість, що визначає SQL-оператор DELETE;

- InsertSQL - властивість, що визначає SQL-оператор INSERT;
- ModifySQL - властивість, що визначає SQL-оператор UPDATE;
- ExecSQL - метод, що виконує один із заданих SQL-операторів (залежно від значення параметра, що вказується такими константами: ukDelete, ukInsert, ukModify).

***Клас TDATABASE*** реалізує роботу з об'єктом "БД" і надає засоби контролю над з'єднанням з БД. Компонент типу TDatabase дозволяє управляти транзакціями.

Для роботи з компонентом TDatabase необхідно встановити значення властивостей AliasName й DatabaseName. Якщо значенням властивості AliasName зазначений DSN існуючого джерела даних, то розроблювач може сам визначити будь-який внутрішній (для додатка) псевдонім БД і задати його у властивості DatabaseName. У цьому випадку для будь-якого набору даних у списку значень властивості DatabaseName буде відображатися поряд з усіма доступними DSN джерелами даних і внутрішній псевдонім, заданий властивістю DatabaseName компонента TDatabase.

У тому випадку, якщо DSN не визначений, то властивість DatabaseName повинна містити повне ім'я файлу БД, а властивість DriverName - указувати використовуваний ODBC-драйвер.

Компонент типу TDatabase дозволяє управляти режимами роботи з наборами даних і транзакціями, використовуючи такі властивості й методи:

- Exclusive - властивість, що дозволяє додатку одержати винятковий доступ до БД (якщо це підтримується SQL-сервером);
- InTransaction - властивість, що вказує, чи був виконаний для БД виклик методу StartTransaction;
- ReadOnly - метод, що вказує, чи встановлений для з'єднання з БД доступ "тільки читання";
- TransIsolation - метод, що задає рівень ізоляції при керуванні транзакціями. Рівень ізоляції визначає, як дана транзакція буде взаємодіяти з іншими транзакціями, що працюють із тими самими таблицями. Властивість TransIsolation може бути зазначена одним із таких значень:

tiDirtyRead - транзакція може читати дані, які були змінені іншою транзакцією, але для яких не був виконаний виклик Commit (фіксація змін);

tiReadCommitted - дозволяє в одній транзакції читати фіксовані зміни, зроблені в базі даних іншою транзакцією;

tiRepeatableRead - істинність даних гарантується на увесь час читання, і транзакція не бачить ніяких змін, зроблених іншою транзакцією. Прочитаний запис залишається постійним, поки в ньому не будуть зроблені зміни усередині самої транзакції;

- StartTransaction - метод, що відкриває нову транзакцію;
- Commit - метод, що виконує фіксацію поточної транзакції;
- Rollback - метод, що виконує відкіт поточної транзакції;
- Execute - метод, що виконує зазначений параметром SQL-оператор, що не повертає результуючого набору.

***Клас TADOCONNECTION*** забезпечує з'єднання з даними, доступ до яких реалізується через ADO-об'єкти. ADO-об'єкти дозволяють працювати з різними сховищами даних, які можуть і не бути SQL-операторами. Об'єкти типу TADOConnection використовують для доступу до даних OLE DB провайдери.

Компоненти TADOCommand й TADODataSet зв'язуються із джерелом даних за допомогою об'єкта TADOConnection, указуючи посилання на нього як значення властивості Connection.

Для ідентифікації з'єднання необхідно визначити значення властивості ConnectionString (рядок з'єднання) компонента TADOConnection, що може ґрунтуватися на вказівці: datalink-файлу; рядка з'єднання.

Якщо як значення властивості ConnectionString зазначене ім'я UDL-файлу, то настроювання з'єднання можна виконувати автономно від додатка (наприклад, указуючи ім'я БД Microsoft SQL Server на поточному ПК).

### 3.3.5 Класи компонентів керування даними

Компоненти керування даними розміщені на сторінці Data Controls палітри компонентів. Багато хто із цих компонентів аналогічні елементам керування сторінки Standard, з тією лише відмінністю, що зв'язано через джерело даних (компонент типу TDataSource) з певним полем (або полями) з набору даних (компонентів типу TTable або TQuery).

Бібліотека VCL надає такі класи компонентів керування даними:

- TDBGrid - клас, що дозволяє відображати запису набору даних у вигляді таблиці й управляти цими записами.
- TDBNavigator - клас, що надає засоби навігації по набору даних, а також можливості додавання нових записів, включення режиму редагування, присвоєння й скасування зроблених змін. Для того, щоб програмно ініціювати дію, виконувану по щиглику на кнопці навігатора, варто викликати метод BtnClick: DBNavigator1.BtnClick(nbPost); // Присвоєння зроблених змін.

Компонент TDBNavigator може відображати кнопки, що вказуються такими константами:

nbFirst - перехід до першого запису;
nbPrior - перехід до попереднього запису;
nbNext - перехід до наступного запису;
nbLast - перехід до останнього запису;
nbInsert - вставка перед поточним записом нового запису й перехід на неї;
nbDelete - видалення поточного запису;
nbEdit - перехід у режим редагування поточного запису;
nbPost - внесення змін поточного запису в БД;
nbCancel - скасування змін, зроблених у поточному записі;
nbRefresh - повторне зчитування значень полів із джерела даних.

- TDBText - клас, що дозволяє, як напис, відображати значення поля поточного запису набору даних.
- TDBEdit - клас, що реалізує роботу з однорядковим полем редагування.
- TDBMemo - клас, що реалізує багаторядкове поле редагування, у якому можна відображати й змінювати значення поля набору даних.
- TDBImage - клас, що реалізує об'єкт "рисунок", у якому можна відображати й змінювати значення поля набору даних формату BLOB.
- TDBRadioGroup - клас, що реалізує групу радіокнопок, які пов'язані з полем БД. Застосування такого об'єкта надає користувачеві зручну можливість встановлювати значення поля БД, вибираючи його із пропонованих опцій.
- TDBCheckBox - клас, що реалізує компонент "прапорець", що пов'язаний з полем БД.
- TDBListBox - клас, що реалізує компонент "список", який використовується для відображення значень поля набору даних. Значення, відображувані в списку, утримуються у властивості Items.
- TDBComboBox - клас, що реалізує компонент "список, що розкривається", який використовується для відображення значень поля набору даних. Значення, відображувані в списку, утримуються у властивості Items.
- TDBLookupListBox - клас, що дозволяє виконувати перегляд списку, заповненого значеннями полів з іншого набору даних. Набір даних, які переглядаються, вказується властивістю ListSource, що переглядає поле (або поля) - властивістю ListField. Властивість KeyField указує поле набору, що переглядається, відповідному полю поточного набору

даних, що вказується властивостями DataField й DataSource. Даний клас дозволяє вибирати значення поля поточного набору даних з іншого набору даних, які переглядаються;

- TDBCtrlGrid - клас, що реалізує особливий вид таблиці, у якій кожен запис відображається на окремій панелі (кількість панелей у компоненті вказується значенням властивості RowCount).

### 3.3.6 Події, які ініціюються для наборів даних

AfterCancel і BeforeCancel - відбувається після/до скасування в додатку всіх змін, зроблених для поточного запису.

AfterClose і BeforeClose - відбувається після/до закриття набору даних і перекладу БД у стан dsInactive.

AfterDelete і BeforeDelete - ініціюється після/до видалення додатком поточного запису, перекладу набору даних у стан dsBrowse і переміщення позиції курсору на попередній запис.

AfterEdit і BeforeEdi - відбувається після/до початку редагування додатком поточного запису.

AfterInsert і BeforeInsert - відбувається після/перш ніж додаток вставить новий запис.

AfterOpen і BeforeOpen - відбувається після/перш ніж додаток відкриє набір даних, але до того, як які-небудь доступні дані будуть відображені.

AfterPost і BeforePost - відбувається до завершення переносу значень активного запису в БД або внутрішній кеш.

AfterRefresh і BeforeRefresh - відбувається після/до відновлення набору даних.

AfterScroll й BeforeScroll - відбувається після/до переміщення позиції курсору на інший запис.

OnCalcFields - відбувається при відкритті набору даних, перекладу його в стан dsEdit, переміщенні фокуса уведення від одного компонента до іншого або від одного стовпця до іншого, при змінах запису або при добуванні запису з БД, але тільки в тому випадку, якщо значення властивості AutoCalcFields дорівнює True;

OnDeleteError - ініціюється, якщо при спробі видалення рядка відбулася помилка - було зроблене виключення.

OnEditError - ініціюється, якщо при спробі зміни або вставки запису відбулася помилка - було зроблене виключення.

OnPostError - ініціюється, якщо при спробі передати зміну або вставку нового запису відбувається помилка - робиться виключення.

OnFilterRecord - відбувається при зміні активного запису й тільки в тому випадку, якщо властивість State набору даних установлено таким, що дорівнює dsFilter, а властивість Filtered дорівнює True. Щоб запис був включений у набір даних, для нього варто встановити параметр Accept таким, що дорівнює True.

OnNewRecord - відбувається при вставці або додаванні нового запису.

## 3.4 Проектування модулів додатків

### 3.4.1 Аналіз функціональної моделі ПО БД

Вхідними даними для вирішення завдання проектування модулів додатків БД є ієрархія функцій. На виході розробник повинен отримати опис (специфікацію) модулів додатків, а в процесі проектування модулів розробник будує відображення бізнес - вимог у специфікації модулів.

Алгоритм дій розробника БД полягає у такому: спочатку розробник намагається сформулювати бізнес - вимоги (функції) у самому загальному вигляді, а потім виконує декомпозицію кожної такої бізнес-функції доти, поки не буде отримана деяка функція, яку можна вважати атомарною функцією.

Розглянемо методологію проектування на прикладі. Розглянемо фрагмент ієрархії функцій для обробки заяв про виплату страхового відшкодування. На спрощеній схемі рис. 3.6 показана функція "2. Обробити заяву". Виконання цієї функції включає виконання чотирьох функцій такого рівня: "2.1. Зареєструвати заяву", "2.2. Ухвалити рішення щодо заяви", "2.3. Здійснити платіж за заявою", "2.4. Закрити заяву".

На рис. 3.6 показана подальша декомпозиція функції "2.2. Ухвалити рішення щодо заяви". Отримана на цьому етапі функція "2.2.5. Дозволити ремонт", є атомарною функцією. Ремонт дозволяється або не дозволяється.

При розгляді ієрархії функцій розробникові БД варто звернути увагу на такі моменти:

- у функціональній моделі БД описуються бізнес-функції, і не всі вони будуть безпосередньо підтримуватися додатком БД;
- при розгляді ієрархій нерідко виникає ситуація, коли екземпляри однієї й тієї ж функції будуть мати різні номери.

Якщо в першому випадку додаткову інформацію про те, які бізнес-функції будуть реалізовані в системі, можна одержати від керівника проекту, то в другому випадку розробник БД, найімовірніше, має справу з помилкою аналітика у визначенні функції.

### 3.4.2 Визначення функцій

Під час розроблення ієрархії функцій аналітик повинен надати текстовий опис до кожної функції, принаймні для

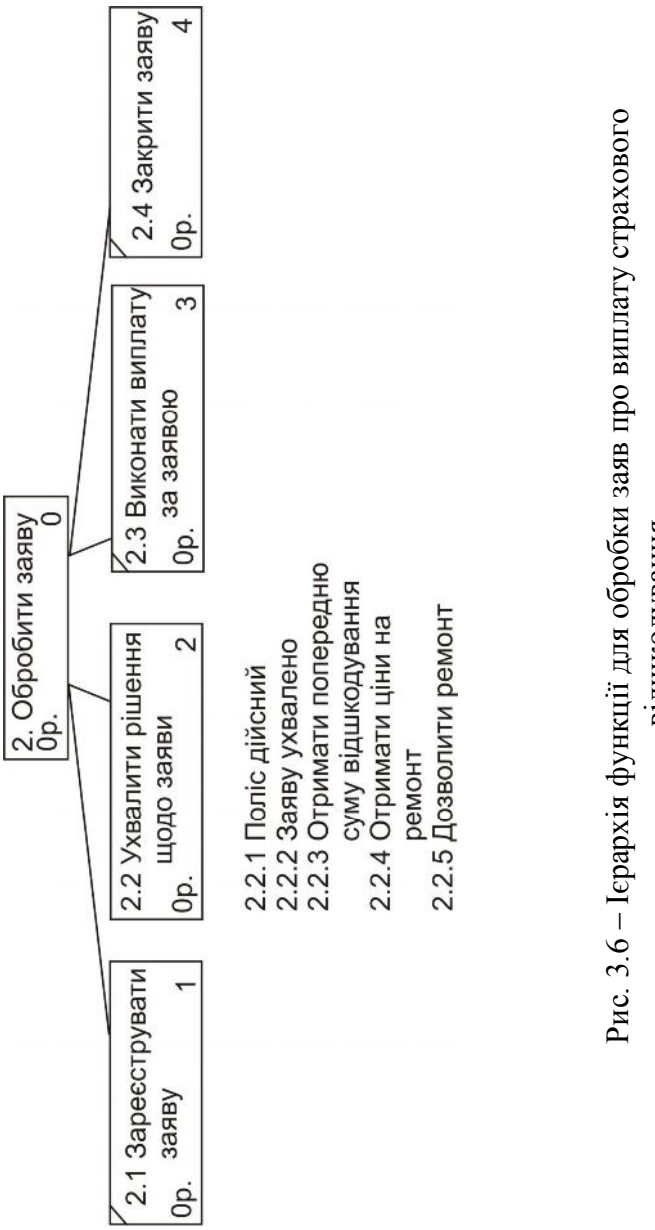



Рис. 3.6 – Ієрархія функції для обробки заяв про виплату страхового відшкодування

верхнього й самого нижнього рівнів ієрархії. Бажано, щоб у цьому описі аналітики виділяли сутності ПО. Це важливо для того, щоб знати з якими сутностями ПО працює функція, тобто які потенційні об'єкти реляційної БД будуть використовуватися в кожній функції. Якщо це не зроблено, то розробникам БД потрібно робити це самостійно.

*Приклад. Визначення функції "2.2.2. Перевірити чи ухвалена заява".*

*"Одержати й зареєструвати всі необхідні страховою компанією відомості про заяву (ВІДОМОСТІ ПРО ЗАЯВУ), включаючи всі докладні відомості про треті сторони (СТОРОННІ ЮРИДИЧНІ ОСОБИ) і свідках (ФІЗИЧНІ ОСОБИ).*

*Вивчити страховий поліс (ПОЛІС) на предмет наявності виняткових ситуацій (ВИКЛЮЧЕННЯ) і визначити, чи діють ці ситуації у випадку даної заяви (ЗАЯВА).*

*Якщо є виключення, то закрити заяву й скласти стандартний лист заявникові про відмову у виплаті (ЛИСТ) заявникові (ЗАЯВНИК).*

*Якщо ніяких виключень ні, то змінити статус заяви на очікування оцінки, призначити й повідомити оцінювача (ОЦІНЮВАЧ)."*

Із приклада видно, які сутності ПО беруть участь у виконанні функції (виділені в дужках), як міняється стан сутності (виділено курсивом) і який алгоритм роботи цієї функції.

Із приклада зрозуміло, що на цьому етапі розробник БД у якості вхідних даних використає також інформаційну модель ПО БД (опис сутностей).

При виконанні аналізу функцій корисно мати деяку таблицю (матрицю) "Функція-Сутність". Ця матриця повинна дати відповідь на такі запитання:

- чи має кожна сутність конструктор (функцію, що створює всі екземпляри сутності);
- чи має вона деструктор (функцію, що видаляє екземпляри сутності);
- чи є посилання на цю сутність (функції, які використовують цю сутність).

Процес аналізу взаємодії функції й сутності прийнято позначати абревіатурою CRUD (Create, Reference, Update, Delete - створення, посилання, модифікація, видалення).

Корисними для розуміння розробником бази даних призначення функцій і того, як дані функції беруть участь у процесі обробки даних у системі, можуть бути діаграми потоку даних і діаграми життєвих циклів сутностей, які були розглянуті нами в другій лекції. Останні, зокрема, дають ясну картину зміни стану сутності, що важливо у визначенні атрибутів статусу сутності.

### 3.4.3 Відображення функцій у модулі

Одним із основних завдань проектування модулів додатків є побудова відображення функцій у модулі. Під час вирішення цього завдання розробник БД повинен акцентувати увагу на структурі БД, що становить основу додатка.

Як правило, вирішення завдання відображення функцій у модулі виконується в чотири етапи:

1. Аналіз роботи функції.
2. Побудова моделі сутностей, що підтримує ці функції.
3. Початок проектування фізичної структури зі створення схеми, що підтримує розроблену модель сутностей.
4. Завершення проектування розробленням специфікацій модулів, які реалізують функції на запропонованій схемі БД.

Із запропонованого вище підходу видно, як тісно переплітаються у процесі проектування процеси розроблення фізичної моделі БД і специфікацій модулів додатків. Таким чином, якщо розробником був розроблений чорновий варіант фізичної моделі БД по алгоритмах, розглянутий нами в попередніх лекціях, то на цьому етапі він повинен бути адаптований до реалізації функцій й, можливо, до певної мірі перероблений.

При відображенні функцій у модулі необхідно отримати схему, що ставить у відповідність кожної функції певний модуль.

Розглянемо нашу БД, що містить інформацію про співробітників, відділи й проекти організації. Припустимо, вона буде підтримувати бізнес-функцію "Керування проектами в організації". Функціональна модель ПО БД у термінах ієрархії функцій наведена на рис. 3.7, а на рис. 3.8 наведений перелік функцій керування проектами в організації.

Завдання полягає у відображенні функцій з переліку на рис. 3.8 у перелік модулів. Спочатку з переліку функцій повинні бути вилучені ті функції, які не будуть підтримуватися додатком БД. Розробник довідається в керівника проекту, що в додатку БД не будуть підтримуватися такі функції:

- призначити куратора проекту; сповістити керівників підрозділів;
- сповістити співробітників; зібрати нараду;
- приступити до виконання; скласти список робіт;
- визначити обсяг робіт; визначити вартість робіт;


Рис. 3.7 – Ієрархія бізнес-функції "Керування проектами в організації


Рис. 3.8 – Перелік функції керування проектами в організації

- визначити час робіт; визначити виробничі потужності;
- розподілити виробничі потужності; розподілити роботи між співробітниками;
- контролювати хід виконання проекту.

Таким чином, буде отриманий перелік функцій, що показаний у лівій колонці таблиці 3.2. Цьому переліку функцій повинен бути поставлений у відповідність перелік модулів додатка БД.

Керівник проекту передав розробникові БД характеристику додатки БД по керуванню виконанням проектів в організації. Цей додаток буде займатися обліком виконуваних і виконаних проектів в організації.

Розробник БД повинен встановити відображення функцій у модулі, як показано на рис. 3.9.

Наведений приклад показує загальний принцип побудови відображення бізнес-функцій у модулі.

Таблиця 3.2 – Переліки функцій і модулів

| Функції | Модуль |
|---|---|
| Призначити керівника проекту | Уведення інформації про проект |
| Визначити бюджет проекту | Уведення інформації про співробітників |
| Визначити список підрозділів | Пошук інформації про співробітників |
| Визначити список співробітників | Пошук інформації про проекти |
| Виконувати проект | Генерація звіту про виконані проекти |
| Здати проект | Генерація звіту про виконувані проекти |

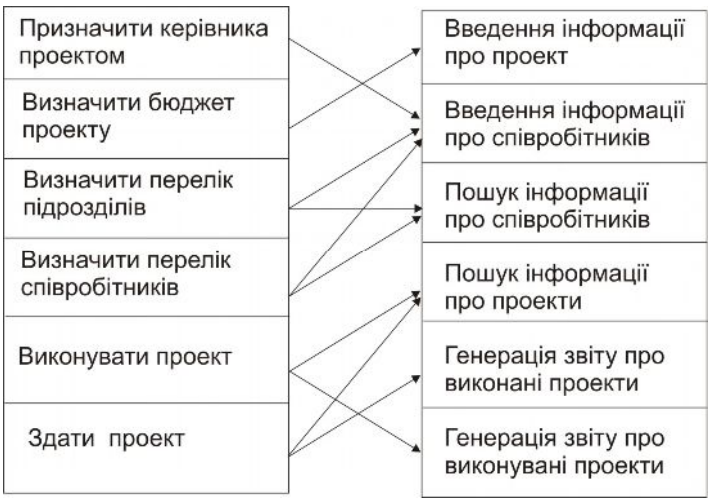



Рисунок 3.9 – Відображення функції в модулі

# 4 VBA відкритий інтерфейс доступу до баз даних – ODBC

## 4.1 Структура модуля. Вікно проекту й Вікно коду

Модулі VBA мають дуже просту синтаксичну структуру. На відміну від більшості мов програмування тут немає великої кількості розділів, немає поділу на інтерфейс і реалізацію. Усе зроблено гранично просто. Кожен модуль поза залежністю від його типу має всього два розділи:

- розділ оголошень змінних рівня модуля. Цей розділ іде першим й автоматично відокремлюється рискою від розділу методів. Завжди можна додати нове оголошення змінної в цей розділ. Область дії таких змінних поширюється на весь модуль, але вона може бути й розширена. Докладно про це ми поговоримо трохи пізніше;
- розділ методів модуля. У цьому розділі розміщуються процедури й функції. З погляду синтаксису нічого іншого крім процедур і функцій у цьому розділі бути не може. Звичайно є, у тому числі, і синтаксична різниця між макросом, методом - оброблювачем події й, наприклад, методом, що представляє процедуру з параметрами загального призначення. Проте, метод це завжди або процедура (Sub) або функція (Function).

## 4.2 Типи даних

Одні з перших запитань, що виникають при вивченні мови програмування: "Як у ньому влаштована система типів даних? Які є прості типи, як створюються складні, структурні типи, чи є можливість визначення власних типів, динамічних типів, чи можна в ньому визначати класи - "дійсні" типи, де визначається не тільки область можливих значень і структура даних, але й операції над ними?" Відповімо на ці запитання стосовно до VBA. Визначення типу T задає: область можливих значень типу; структуру організації даних; операції, певні над даними цього типу.

Типи даних прийнято поділяти на прості й складні залежно від того, як розміщені їхні дані. У простих (скалярних) типів можливі значення даних єдині й неподільні. Складні типи характеризуються способом структуризації даних, - одне значення складного типу складається з безлічі значень даних, що організують складний тип.

Є й інші критерії класифікації типів. Так, типи поділяються на вбудовані типи й типи**,** визначені програмістом (користувачем). Вбудовані типи споконвічно належать мові програмування й становлять його базис. В основі системи типів будь-якої мови програмування завжди лежить базисна система типів, вбудованих у мову. На основі вбудованих типів програміст може будувати власні, певні типи даних. Але способи (правила) створення таких типів є базисними, вбудованими в мову.

Типи даних поділяються також на статичні й динамічні типи . Для даних статичного типу пам'ять виділяється в момент оголошення, необхідний розмір даних відомий при їхньому оголошенні. Для даних динамічного типу розмір даних у момент оголошення не відомий і пам'ять їм виділяється динамічно в процесі виконання програми по запиту.

***Прості типи даних***

Як і всяка мова, VBA містить всі звичні вбудовані прості типи даних: логічні, арифметичні й строкові.

Таблиця 4.1 – Система простих типів мови VBA

| Ім'я типу | Можливі значення | Необхідна пам'ять |
|---|---|---|
| 1 | 2 | 3 |
| Boolean | True, False | 2 байти |
| Byte | 0.....255 | 1байт |
| Integer | -32768 …+32767 | 2 байти |
| Long | Приблизно: -2000 000 000…+2000000000 | 4байти |
| Decimal | Приблизно 30 десяткових цифр. Можна вказати число цифр після десяткової крапки . | 12 байтів |
| Single | - 3,4E38 …-1,4 E-45 для негативних значень 1,4E-45 … 3,4E38 для позитивних значень | 4 байти |
| Double | -1,7E308 … -4,9E-324 для негативних значень4,9E-324 … 1,7E308 для позитивних значень | 8 байтів |
| Currency | Десяткові числа з фіксованою позицією коми. Можливі 15 цифр до коми й 4 після | 8 байтів |

Продовження таблиці 4.1

| 1 | 2 | 3 |
|---|---|---|
| String | Є два види рядків: рядки фіксованої довжини мають до 216 символів. Рядки змінної довжини мають до 2 31 символів | 10 байтів +1 байт на символ у звичайному кодуванні й 2 байти в кодуванні Unicode |
| Date | Дати змінюються в діапазоні від 1 січня 100 р. до 31 грудня 9999 р. | 8 байтів |
| Object | Посилання на об'єкт (покажчик) | 4 байти |
| Variant | Універсальний тип, значенням | Залежить від контексту, але не |

| | | |
|---|---|---|
| | якого можуть бути дані кожного з перерахованих вище типів, об'єкти, значення NULL і значення помилок ERROR | менш 16 байтів |

***Масиви***

Найпростіший і найпоширеніший структурний тип - масив - упорядкована сукупність даних одного типу. Порядок на елементах масиву задається індексами його елементів. У VBA масиви можуть бути одномірними й багатомірними.

Синтаксис оголошення масивів розширений, - після ім’я змінної у круглих дужках вказується перелік розмірностей масиву:

{Dim | Private | Public | Static }<ім'я змінної> (<список розмірностей >)

[ As <ім'я типу>]

***Динамічні масиви***

Динамічні масиви VBA – це потужний засіб. Масив уважається динамічним , якщо при первісному оголошенні не вказується його розмірність, але вона може бути визначена й перевизначена надалі оператором ReDim. Розмірність визначається динамічно в тій процедурі й у той момент, коли вона стає фактично відомою. Зверніть увагу, у цьому операторі межі зміни індексів можна задати не тільки як константи, але і як вирази, залежні від змінних.

Якщо потім потрібно змінити межі або розмірність масиву, ви можете знову задати оператор перевизначення ReDim і почати новий цикл роботи з масивом. І ще одна "приємність" - можна зберегти усі раніше отримані елементи й розширити масив, додавши нові елементи. Для цього треба просто задати ключове слово Preserve при перевизначенні масиву. Динамічні масиви з успіхом можна застосовувати там, де необхідні динамічні структури даних, наприклад, списки, стеки, черги.

### 4.3 Оголошення

Оголошення можна давати на двох рівнях - модуля й процедури. На рівні модуля розділ оголошень іде першим й автоматично відокремлюється рискою від розділу методів. На рівні процедури оголошення й оператори можуть бути перемішані, потрібно лише щоб оголошення змінної передувало її використанню. Хорошим тоном вважається й у процедурах мати два чітко виділених розділи й всі оголошення розміщувати на початку процедури, так, щоб вони передували виконанню частини процедури - розділу операторів.

Давайте чітко виділимо основні частини розділу оголошень:

- Розділ опцій.
- Розділ констант.
- Розділ типів.
- Розділ змінних.
- Розділ Declare.

***Розділ опцій***. Опції є вказівками для транслятора. Вони можуть задаватися тільки на рівні модуля й повинні починати розділ оголошень. Опції задаються ключовим словом Option, після якого йде ім'я опції і можливо параметри. Перерахуємо їх складові:

- Explicit - при її вказівці транслятор вимагає, щоб всі змінні модуля були явно описані. Необхідно включити цю опцію раз і назавжди в опціях Редактора VBA;
- Base - ця опція має два значення: 0 і 1, що вказують нижню межу індексу масивів, що задає за замовчуванням. Правильно не користуватися цією опцією, а самому й завжди вказувати нижню межу.

- Private - цю опцію досить помістити в один з модулів проекту, зазвичай у головний модуль проекту, що неявно завжди виділяється програмістом. При її завданні проект робиться закритим і недоступний для інших проектів у системі документів;
- Compare - опція показує транслятору, як він повинен виконувати порівняння рядків у процедурах модуля. Параметр опції може брати одне із трьох можливих значень: {Binary | Text | DataBase}

***Розділи констант, типів і змінних***

Рекомендується створювати такі розділи, які будуть відокремлюватись коментарями один від іншого. Бажано, щоб ці розділи йшли в описаному нами порядку. Тоді вже оголошені константи будуть з'являтися при описі меж масивів, типи змінних будуть передувати оголошенню самих змінних.

Для констант і типів ключові слова Const й Type однозначно визначають описуваний об'єкт. Для змінних ключові слова можуть бути різними. Помітимо, що для змінних рівня модуля не доцільно використовувати ключове слово Dim, краще використовувати специфікатори Public або Private, що явно вказують область дії змінної.

Const має на увазі, що оголошена величина залишається незмінною. Синтаксис

Const *Ім'яСталоїВеличини = Значення*

Змінні, оголошені за допомогою службового слова Global, можуть адресуватися у будь-якому місці коду програми.

Global *Ім'яГлобальноїВеличини* As *ТипДаних*

***Розділ Declare***

Цей розділ з'являється в тих випадках, коли модулі проекту використовують бібліотеки, що приєднують динамічно, - DLL. Якщо DLL має бібліотеку типів TypeLib і вона доступна проекту, то немає необхідності описувати компоненти бібліотеки, вони будуть знайдені автоматично. Але якщо TypeLib недоступна або не визначена, то в цій ситуації кожна з функцій і процедур бібліотеки, яка викликається в модулі, повинна бути попередньо описана спеціальним оператором Declare. Розглянемо його синтаксис:

[Public | Private] Declare {Sub | Function} ім'я Lib "ім'я_бібліотеки" _

[Alias "псевдонім"] [([параметри])] [As що повертає_тип]

У ньому указується ім'я бібліотеки, ім'я процедури або функції, можливий псевдонім, параметри й значення функцій, яке повертається.

## 4.4 Правила іменування

Є кілька простих правил, які варто виконувати, щоб бути цивілізованим програмістом. Правила, про які ми зараз поговоримо, стосуються оформлення тексту програм. Наведемо основні правила:

- використовуйте коментарі,
- дотримуйтесь правил іменування,
- структуруйте текст,
- будуйте програми з модулів "піднімального" розміру.

Правило написання імен змінних полягає в такому: ім'я повинне відбивати основний зміст і складатися з одного або декількох разом написаних слів, кожне з яких починається з великої букви.

Розроблювачі від Microsoft рекомендують дотримуватися більш суворих правил. Ім'я повинне відбивати не тільки зміст, але й тип змінної, область дії. Тому ім'я повинне складатися із префікса й власного імені. Префікс також є складовим, дві його частини відбивають область дії й тип змінної.

## 4.5 Оператори

VBA - операторна мова. Це означає, що її програми (модулі) представляють послідовності операторів. Групу декларативних операторів VBA, з якими працює програма (типів, змінних, констант, об'єктів додатків й ін.), ми вже розглянули. Оператори іншої групи забезпечують присвоєння й зміну значень цих об'єктів, оператори третьої групи керують ходом обчислень, четвертої - роботою з каталогами й файлами й т.д.

### 4.5.1 Оператори й рядки

При записі тексту програм для спрощення читання, налагодження й модифікації програми кожен оператор слід розміщувати в окремому рядку тексту. Дотримуйтесь правила: "Один оператор - один рядок". Але дозволяється розміщувати на рядку й кілька операторів, на відміну від загальноприйнятого символу поділу операторів "крапки з комою", в VBA символом поділу двох операторів в одному рядку служить двокрапка. Помітьте, деякі оператори, наприклад, оператор If, можуть стояти лише на першому місці в рядку. І із цієї причини кожен оператор, як правило, варто починати з нового рядка, лише іноді, розумно, групу операторів присвоювання розміщувати в одному рядку. Частіше виникає інша ситуація, - оператор занадто довгий і його текст не видний повністю на екрані дисплея, що затруднює читання й розуміння програми. У цьому випадку оператор варто продовжити в одній або декількох рядках. Щоб продовжити (перенести) оператор на наступний рядок, використовується пара символів пробіл-підкреслення "_". Наприклад,

```
MyAddress = "будинок: " & Number & "вулиця: " & Street _
    & "місто: " & City
```

Перед оператором у рядку може стояти мітка - послідовність символів, що починається з букви й кінчається двокрапкою ":". Мітки можна розміщати й в окремих рядках перед тими операторами, які вони повинні позначати. Вони потрібні для операторів переходу типу GoTo, використання яких вважається "дурним тоном" у структурному програмуванні. Але іноді без міток і переходів на них обійтися важко - зокрема, для вказівки входів в оброблювачі помилок у деяких процедурах.

### 4.5.2 Оператор коментарю

Коментарі на виконання програми не впливають, але необхідні як ознака "гарного стилю". Офісні програми використовуються багаторазово й не раз модернізуються в процесі свого життя. Ви можете заощадити на коментарях і написати, налагодити невеликий модуль без них. Але вже через тиждень ніхто, у тому числі й автор, не зможе зрозуміти його дію й модифікувати потрібним способом. Коментар у VBA починається апострофом (') і включає будь-який текст, розміщений правіше в рядку. Звичайно в коментарях описують завдання, які розв'язуються модулями, функції, які виконуються процедурами, зміст основних змінних (якщо він неясний з імен), алгоритми роботи процедур. Корисно також коментувати оператори викликів зовнішніх для даного модуля процедур, пояснюючи їхні дії.

### 4.5.3 Оператор присвоювання

Оператори присвоювання - основний засіб зміни стану програми (значень змінних і властивостей об'єктів). У VBA кілька видів операторів присвоювання. Найпоширеніший серед них Оператор Let. За допомогою цього оператора відбувається "звичайне" присвоєння значення вирази змінної або властивості.

Синтаксис: *[Let] змінна = вирази*

Ключове слово Let, як правило, опускається. Змінна є ім'ям змінної або властивості; вираз задає значення, що привласнюється змінній. Його тип повинен відповідати типу змінної.

*Приклади*:

```
Public Sub Assign1()
  Dim MyStr As String, MyInt As Integer
  Let MyStr = "Доброго дня!"          ' Із ключовим словом Let
  MyInt = 5          ' Без нього. Звичайний варіант.
  Debug.Print MyStr, MyInt
End Sub
```

### 4.5.4 Керуючі оператори

Цикли із можливою перевіркою умови на початку, наприкінці й у середині роботи оператора, звичайний оператор If й оператор розбору випадків Case - всі ці засоби дозволяють організувати процес обчислень надійно й ефективно відповідно до кращих традицій програмування.

***Умовний оператор If Then Else End If***

Це загальноприйнятий у мовах програмування оператор керування обчисленнями дозволяє вибирати й виконувати дії залежно від істинності деякої умови.

Є два варіанти синтаксису: в один рядок й у формі блоку. У першому випадку він має вигляд:

*If умова Then [оператори1] [Else оператори2 ]*

У другому випадку оператор розміщений на декількох рядках:

*If умова Then*
*[оператори]*
*[ElseIf умова-n Then*
*[оператори-n]...*
*[Else*
*[ІншіОператори]]*
*End If*

Тут умова обов'язково в обох варіантах. Вона може бути числовим або строковим виразом зі значеннями True або False (Null трактується як False). Як умову можна використовувати й вирази виду TypeOf Ім’яОб’єкта Is ТипОб’єкта, де Ім’яОб’єкта - посилання на об'єкт, а ТипОб’єкта - довільний коректний тип об'єкта. Оператори1 й оператори2 - це послідовності з одного або декількох розділених двокрапкою операторів. Принаймні, одна із цих послідовностей повинна бути непустою. Якщо умова істинна (True), виконується послідовність оператора1, неправдива (False) - оператора2. Закриваючий оператор End If необхідний.

### 4.5.5 Оператор вибору Select Case

Цей оператор робить розгляд випадків, і залежно від значення аналізованого виразу вибирає й виконує одну з послідовностей операторів.

Синтаксис.

*Select Case Тест^-тест-вираз-тест*
*[Case ПерелікВиразів-n*
  *[оператори-n]]*
*[Case Else*
  *[ІншіОператори]]*
*End Select*

*Вираз-тест* повинен бути присутнім обов'язково. Воно може бути довільним виразом із числовим або строковим значенням. ПерелікВиразів-n повинен бути присутнім у рядку, що починається ключовим словом Case (Випадок).

***Цикл For Next***

Дозволяє повторювати групу операторів задане число раз.

Синтаксис:

*For лічильник_циклу = початок To кінець [Step крок]*

*тіло циклу*

*Next [лічильник_циклу]*

Тут *лічильник_циклу* - це числова змінна. На початку виконання циклу вона набуває значення, що задає числовим виразом початок (змінна *лічильник_циклу* не може мати тип Boolean або бути елементом масиву). Числовий вираз *кінець* задає заключне значення лічильника циклу. Воно обчислюється до початку виконання тіла циклу й не міняється, навіть якщо вхідні в нього змінні змінюють у тілі циклу свої значення. Числовий вираз *крок* необов'язковий. Його значення також обчислюється на початку циклу й додається до лічильника циклу щоразу, коли завершується виконання тіла циклу й обчислення досягає рядка *Next [лічильник_циклу]*. Якщо *крок* циклу явно не зазначений, за замовчуванням він дорівнює 1. *Тіло циклу* - це послідовність операторів, яка буде виконана задане число раз. При якому значенні змінної *лічильник_циклу* відбувається завершення циклу, залежить від знака параметра *крок*. Якщо *крок* додатний, цикл завершиться, коли вперше виконається умова: *лічильник_циклу* > *кінець*.

Якщо *крок* циклу від'ємний, умова його завершення: *лічильник_циклу* < *кінець*.

Ця умова перевіряється перед початком виконання циклу, а потім - після кожного додатка кроку до лічильника циклу в операторі *Next*. Якщо воно виконане, керування передається на оператор, що випливає за *Next*, немає - виконуються оператори з тіла циклу. Завершити цикл *For ... Next* можна й за допомогою оператора *Exit For*. Такі оператори можуть бути розміщені в тих місцях тіла циклу, де потрібно з нього вийти, не чекаючи виконання умови завершення.

*Приклад.*

У нашому прикладі три вкладених цикли For … Next використовуються для обчислення добутку двох цілочислених матриць з випадкових чисел. Потім результуюча матриця перевіряється на наявність у ній нульового значення:

```
Public Sub For1()
    Dim A(1 To 5, 1 To 5) As Integer
    Dim B(1 To 5, 1 To 5) As Integer
    Dim C(1 To 5, 1 To 5) As Integer
    Dim I As Integer, J As Integer, K As Integer
    Dim Res As String
    ' Ініціалізація матриць       A і B випадковими числами в интервалі [-10, +10]
    VBA.Randomize
    For I = 1 To 5
        For J = 1 To 5
        'Отримання випадкового числа Rnd і перетворення у ціле
        A(I, J) = Int(21 * Rnd) - 10
        Next J
    Next I
    For I = 1 To 5
        For J = 1 To 5
            B(I, J) = Int(21 * Rnd) - 10
        Next J
    Next I

    'Обчислення добутку матриць
    For I = 1 To 5
        For J = 1 To 5
            C(I, J) = 0
            For K = 1 To 5
            C(I, J) = C(I, J) + A(I, K) * B(K, J)
            Next K
```

```
            Next J
        Next I

        Res = "No"
        C(2, 2) = 0
        'Перевірка на нульове значення
        For I = 1 To 5
            For J = 1 To 5
                If C(I, J) = 0 Then
                Debug.Print "Індекси: ", I, J
                Res = "Yes"
                Exit For
                End If
            Next J
        Next I
        Debug.Print Res
End Sub
```

Зверніть увагу, оператор виходу Exit For припиняє виконання тільки внутрішнього циклу, так що перевірка на нуль буде здійснюватися в кожному рядку матриці, незалежно від існування нулів у попередніх рядках.

### 4.5.6 Цикл Do...Loop

Повторює блок операторів, поки задана умова є істиною або поки вона не стане істиною.

Синтаксис. Є чотири варіанти синтаксису цього циклу. У двох перших варіантах умова перевіряється на початку циклу:

*Do [{While | Until} умова]*
*тіло циклу*
*Loop*

В інших двох варіантах умова перевіряється наприкінці циклу:

*Do*
*тіло циклу*
*Loop [{While | Until} умова]*

Тут *умова* є числовим або строковим виразом зі значеннями True або False. Взагалі вона є необов'язковою. Значення Null умови трактується як False. *Тіло циклу* - це послідовність операторів, що буде виконуватися, поки умова залишається істиною, якщо перед ним стоїть ключове слово While або поки воно залишається помилковим - у варіантах циклу із ключовим словом Until.

***Цикл While...Wend***

Повторює виконання послідовності операторів, поки задана умова не стане помилковою.

Синтаксис:

*While умова*
*тіло циклу*
*Wend*

Тут *умова* й *тіло циклу* такі самі, як і для циклу Do…Loop... Тільки для цього виду циклу не передбачений оператор виходу Exit. Фактично цикл While…Wend-окремий випадок циклу Do…Loop-залишений у мові для сумісності з попередніми версіями.

### 4.5.7 Цикл For Each...Next

Повторює задану послідовність операторів для кожного елемента масиву або набору.

Синтаксис:
*For Each елемент In група*
*тіло циклу*
*Next [елемент]*

Тут *елемент* - змінна, яка виступає як значення елемента колекцій або масиву. Для колекцій *елемент* може бути змінної типу Variant, змінної типу Object або змінної (об'єктом) деякого класу. У випадку циклу по масиву *елемент* зобов'язаний бути змінного типу Variant. *Група* - це ім'я набору об'єктів (найчастіше це колекція об'єктів) або масиву, для елементів яких виконується цикл. Цикл не застосуємо для масивів, тип елементів яких визначений користувачем, тому що такі елементи не можуть бути значеннями змінної типу Variant. У таких масивах можна використовувати цикл виду For … Next... *Тіло циклу* - послідовність операторів, яка виконується для кожного елемента набору або масиву, - може містити оператори Exit For, що дозволяють перервати виконання циклу й передати керування операторові, що випливає за Next (звичайно такий вихід відбувається при виконанні деякої умови, що перевіряє в операторі If…Then…Else)... Указувати змінну елемент після ключового слова Next не обов'язково, але бажано.

### 4.6 Операції

У будь-якій мові програмування припустимі вирази. Потрібно вміти виражатися коректно. Вирази будуються зі змінних, констант, вбудованих функцій з використанням знаків операцій і дужок. Запис виразів задає правило (алгоритм) обчислення його значення і його типу. Природно, що тип і значення всіх його змінних повинні бути визначені до моменту обчислення виразу. Мови програмування розрізняються між собою тим, до якого ступеня вони допускають автоматичне перетворення типів даних у процесі обчислення виразу. Серед вбудованих функцій є велика кількість функцій, призначених для явного перетворення типів, що дозволяє програмувати в кращих традиціях надійних мов програмування, не довіряючи неявним перетворенням.

Якщо вирази містять операції різних категорій, то першими виконуються арифметичні операції, потім, операції порівняння й останніми - логічні.

Всі операції порівняння мають той самий пріоритет. Арифметичні й логічні операції виконуються відповідно до зазначеного пріоритету.

Дужки дозволяють змінити зазначений порядок обчислень виразу, оскільки вирази в дужках мають найвищий пріоритет й обчислюються першими. Усередині дужок діє звичайний порядок обчислення.

Операція конкатенації не є арифметичною, але тільки вона з'явиться у вираженні, відразу виконується після всіх арифметичних операцій, але до обчислення операцій порівняння.

Розглянемо основні вбудовані математичні функції.

#### 4.6.1 Математичні функції

Набір математичних функцій VBA досить стандартний. Перелічимо їх з короткими поясненнями:

- Abs(число) - абсолютне значення числа.
- Atn(число)- арктангенс (у радіанах) аргументу, що задає тангенс кута.
- Cos(число) - косинус кута. Аргумент числа задає кут у радіанах.
- Exp(число) - експонента, тобто результат зведення числа e (підстава натуральних логарифмів) у зазначений ступінь.
- Log(число) - натуральний логарифм числа.
- Rnd[(число)] - результат представляє рівномірно розподілене випадкове число в інтервалі [0 - 1]. Якщо аргумент числа не заданий або більше нуля, то породжується чергове випадкове число, якщо він дорівнює 0, то результатом буде попереднє випадкове число, а

якщо число менше нуля, то будь-який породжує те саме число, обумовлене аргументом. Помітимо, для формування значення випадкових чисел використовується таймер.

- Sgn(число) - знак числа (якщо число більше нуля - 1, дорівнює нулю - 0, менше нуля - -1).
- Sin(число) - синус кута. Аргумент число задає кут у радіанах.
- Sqr(число) - квадратний корінь.
- Tan(число) - тангенс кута. Аргумент число задає кут у радіанах.

У всіх цих описах під аргументом функції числа розуміються числові вирази.

### 4.6.2 Робота з рядками

Працювати з текстами програмістові, як правило, доводиться значно частіше, ніж працювати із числами. Тому варто добре представляти основні базисні операції над рядками, які в більшості випадків реалізуються за допомогою вбудованих функцій.

Звичайні операції порівняння можуть бути застосовані й до строкових даних. Ми вже говорили раніше про те, що інтерпретація цих операцій залежить від установки опції Option Compare.

- Якщо ця опція встановлена як Text, то порівняння на "більше - менше" представляє лексикографічне порівняння, коли рядки рівняються за їх розміщенням у словнику. Програмісти, як правило, розуміють, що порівняння рядків означає порівняння кодів їхніх символів, так що лексикографічний порядок визначається кодуванням символів алфавіту.
- Якщо ця опція встановлена як Binary, то порівняння йде побітно. У цьому випадку порівняння природно, відчутно до регістра.
- При роботі з рядками в Access за замовчуванням застосовується сортування, задана на рядках БД Access. Помітьте, при створенні модуля в Access за замовчуванням уставляється опція Option Compare Database. Звичайно ця опція застосовна тільки при роботі в Access.

Якщо потрібно локально перевизначити вид порівняння, заданий опцією для всього модуля, то можна використовувати вбудовану функцію StrComp, що повертає результат порівняння рядків. Її синтаксис:

*StrComp(string1, string2[, compare])*

Аргументи *string1* й *string2* - порівнювані рядки. Необов'язковий аргумент *compare* вказує спосіб порівняння рядків: значення за замовчуванням 0 використовується, щоб виконати двійкове порівняння, 1 задає посимвольне порівняння без обліку регістра.

Якщо string1 менше ніж string2, то результат дорівнює -1, якщо рядки рівні, то - 0, якщо друга менше, то дорівнює 1, якщо хоч один з рядків має значення Null, те результат також дорівнює Null.

***Оператори конкатенації рядків***

Існують два оператори додавання рядків. Вони позначаються символами *плюс* (+) і *амперсанд* (&). Обоє оператори бінарні. Відмінність оператора полягає в тому, що він здатний перетворювати значення будь-якого вбудованого типу даних у рядок.

*Приклад*:

```
1: Dim A As String
2: A = “Здраствуй, ” + “світ! ”
```

### 4.6.3 Робота з датами й часом

Для того, щоб забезпечити програмістові можливість коректно працювати з датами й часом, VBA надає спеціальний тип даних Date, що зберігає дату й час. Над даними цього типу можна виконувати деякі операції, але при роботі з ними найчастіше використовуються спеціальні вбудовані функції. Спробуємо коротко розглянути основні можливості роботи з датами. Насамперед, помітимо, що можливий діапазон дат охоплює дати від 1.1. 100 року до 1-го січня 9999 року. Якщо говорити про внутрішнє подання дат, що займають 4 байти

пам'яті, то ціла частина зберігає число днів від деякої початкової дати, дробова частина зберігає час від опівночі. Початковою датою є 30 грудня 1899 року. Завдяки такому внутрішньому поданню додавання й вирахування цілого числа сприймається як додаток або вирахування днів.

***Присвоювання значень***

При присвоюванні значень змінним типу дата необхідно містити дату в спеціальні обмежники "#" або задавати її як строкову константу. При завданні дати в обмежниках, наприклад, #9, May, 99 # вона автоматично перетвориться до стандартного формату #5/9/99# (місяць/день/рік). Наведемо приклад деяких дій над датами:

```
Public Sub WorkWithDates()

  'Робота з датами
  Dim dat1 As Date, dat2 As Date, dat3 As Date

  'Присвоювання дат
  dat1 = 12
  dat2 = 9/5 / 99
  dat3 = #9/5/1999#
  Debug.Print dat1, dat2, dat3
  dat1 = "15/7/99"
  dat2 = #5/9/1999#
  dat3 = dat3 + 100
  Debug.Print dat1, dat2, dat3
  If dat3 > dat2 Then
    Debug.Print dat3 - dat2
  Else
    Debug.Print dat2 - dat3
  End If
End Sub
```

Наведемо результати виконання цієї програми:

```
11.01.1900 0:26:11              05.09.99
15.07.99              09.05.99              14.12.99
 219
```

***Вбудовані функції для роботи з датами***

Як ми вже зазначали, основна робота з датами виконується з використанням вбудованих функцій. Основні з них:

- Date - повертає поточну дату.
- Time - повертає поточний час за годинниками комп'ютера.
- Now - повертає значення типу Variant (Date), що містить поточну дату й час за системним календарем й годинником комп'ютера.

***Обчислення над датами***

Функція DateAdd призначена для додавання або віднімання зазначеного тимчасового інтервалу зі значення дати. Зазначте, з її допомогою можна задавати часовий інтервал не тільки в днях, але й у місяцях, роках і так далі:

*DateAdd(interval, number, date)*

Аргумент *interval* - рядок, що вказує тип тимчасового інтервалу, що додається, *number* - число тимчасових інтервалів, на яке варто змінити дату, *date* - дата, до якої додається зазначений часовий інтервал. Для приклада, наведемо два виклики цієї функції у вікні налагодження:

```
? DateAdd("m", 1, "31-січня-95")
28.02.95
```

? DateAdd("m", -1, "31-січня-95")

31.12.94

Функція DateDiff призначена для визначення часу, що пройшов між двома датами. Наприклад, за допомогою цієї функції можна обчислити число днів між двома датами або число тижнів між поточною датою й кінцем року. Синтаксис:

*DateDiff(interval, date1, date2[, firstdayofweek[, firstweekofyear]])*

Аргумент *interval* задає тип тимчасового інтервалу при обчисленні різниці між датами *date1* й *date2*, - його можливі значення ті самі, що й для функції DateAdd, date1 й date2 - дві дати, різницю між якими варто обчислити. *firstdayofweek* - константа, що вказує перший день тижня (за замовчуванням уважається, що тиждень починається з неділі). *firstweekofyear* - константа, що вказує перший тиждень року (за замовчуванням першим тижнем уважається тиждень, що містить 1 січня).

Наведемо приклади виклику цієї функції у вікні налагодження:

? DateDiff("m", "18.10.55", "31-січня-95")

471

Функція DateSerial дозволяє обчислити значення дати типу Variant (Date) за її компонентами, - роком, місяцем й днем. Її синтаксис:

DateSerial(year, month, day)

Значення кожного аргументу повинне лежати у відповідному діапазоні: 100 -- 9999 для року, 1 - 31 для днів й 1 - 12 для місяців. Можна також використовувати для аргументів числові вирази для опису відносної дати.

Приклади:

? DateSerial(1997 - 25, 10 - 2, 18 - 1)

17.08.72

? DateSerial(Year(Now) - 3,Month(Now) - 2, Day(Now) - 1)

08.03.96

Функція DateValue переводить аргумент-рядок у дату. На відміну від функції перетворення CDate, функція DateValue правильно обробляє припустимі дати, що містять повні або короткі назви місяців.

Її синтаксис:

DateValue(date)

Аргумент date може задавати як дату, так і час. Можливі різні формати завдання дати, у тому числі й з назвами місяців.

Остання рівність пов'язана з тим, що при відсутності року, функція DateValue використає поточний рік за системним календарем комп'ютера.

Функції Day(date), Month(date), і Year(date) у деякому змісті є зворотними до двох попередніх. Вони за аргументом-датою визначають номер дня, місяця й року. Функція Weekday(date, [firstdayofweek]) повертає значення типу Variant (Integer), що містить ціле число, яке представляє день тижня. Другий аргумент задає перший день тижня (за замовчуванням - неділя). Значення 2 відповідає понеділку.

Функції Hour(час), Minute(час) і Second(час) за аргументом, який є числовим або строковим виразом, що представляє час, повертає утримуюче ціле число, що представляє, відповідно, годинники, хвилини й секунди в значенні часу.

Функція TimeSerial(hour, minute, second) обчислює результат Variant (Date), що містить значення часу, яке відповідає зазначеним годині, хвилині й секунді.

Функція TimeValue(час) повертає значення типу Variant (Date), що містить час. Аргумент час звичайно задається строковим виразом, що подає час від 0:00:00 (12:00:00 A.M.) до 23:59:59 (11:59:59 P.M.) включно.

## 4.7 Опис і створення процедур

Проекти різних документів пов'язані між собою й можуть мати загальні дані й загальні процедури й функції, доступні з різних проектів. Кожен проект складається з модулів різного типу. Кожен модуль містить оголошення змінних, процедур і функцій. Процедури й функції є мінімальними модульними конструкціями, з яких будується програмний проект.

*Процедура (функція)* - це програмна одиниця VBA, що включає оператори опису її локальних даних і операцій над даними. Звичайно в процедуру поєднують регулярно виконуючу послідовність дій, що вирішує окрему задачу або підзадачу.

### 4.7.1 Класифікація процедур

Процедури VBA можна класифікувати за декількома ознаками: за способом використання (виклику) у програмі, за способом запуску процедури на виконання, по способу створення коду процедури, за місцем перебування коду процедури в проекті.

Процедури VBA поділяються на підпрограми й функції. Перші описуються ключовим словом Sub, другі - Function. Ми дуже рідко використовуємо термін підпрограма, характерний для VBA, і замість нього використовуємо термін процедура.

За способом створення коду процедури поділяються на звичайні, розроблювальні "вручну", і на процедури, код яких створюється автоматично генератором макросів (MacroRecoder); їх називають також макро-процедурами або командними процедурами, оскільки їх код - це послідовність викликів команд відповідного додатка Office.

За способом запуску процедур на виконання можна виділити в окрему групу процедури, що запускають автоматично при настанні тієї або іншої події, - ми називаємо їхніми процедурами обробки подій.

За положенням у проекті розрізняються процедури, що перебувають у спеціальних програмних одиницях - стандартних модулях, модулях класів і модулях, пов'язаних з об'єктами, що реагують на події.

Ще один спеціальний тип процедур - процедури-властивості Property Let, Property Set й Property Get. Вони служать для завдання й одержання значень закритих властивостей класу.

При виклику процедури її аргументи, що відповідають вхідним параметрам одержують значення, так процедура одержує інформацію від зовнішнього середовища, у результаті роботи процедури формуються значення вихідних параметрів, переданих їй по посиланню, тим самим змінюється стан проекту й документів. Другий спосіб складається у використанні процедурою глобальних змінних й об'єктів, як для одержання, так і для передачі інформації.

### 4.7.2 Синтаксис процедур і функцій

Опис процедури Sub в VBA має такий вид:

*[Private | Public] [Static] Sub ім'я([перелік-аргументів])*
  *тіло-процедури*
*End Sub*

Ключове слово *Public* у заголовку процедури використовується, щоб оголосити процедуру загальнодоступною, тобто дати можливість викликати її із всіх інших процедур всіх модулів будь-якого проекту. Альтернативний ключ *Private* використовується, щоб закрити процедуру від всіх модулів, крім того, у якому вона описана. За замовчуванням процедура вважається загальнодоступною.

Ключове слово *Static* означає, що значення локальних (оголошених у тілі процедури) змінних будуть зберігатися в проміжках між викликами процедури (використані процедурою глобальні змінні, описані поза її тілом, при цьому не зберігаються).

Параметр *ім'я* - це ім'я процедури, що задовольняє стандартним умовам VBA на імена змінних.

Необов'язковий параметр *перелік-аргументів* - це послідовність розділених комами змінних, що задають передані процедурі при виклику параметри.

Послідовність операторів *тіло-процедури* задає програму виконання процедури. Тіло процедури може включати як "пасивні" оператори оголошення локальних даних процедури (змінних, масивів, об'єктів й ін.), так й "активні" - вони змінюють стани аргументів, локальних і зовнішніх (глобальних) змінних й об'єктів. У тіло можуть входити також оператори Exit Sub, що приводять до негайного завершення процедури й передачі керування в основну програму. Кожна процедура в VBA визначається окремо від інших, тобто тіло однієї процедури не може включати опису інших процедур і функцій.

Синтаксис визначення процедур-функцій схожий на визначення звичайних процедур:

*[Public | Private] [Static] Function ім'я [(перелік-аргументів)] [As тип-значення]*
*тіло^-функції*
*End Function*

Відмінність лише в тому, що замість ключового слова *Sub* для оголошення функції використовується ключове слово *Function*, а після списку аргументів слід вказати параметр *тип-значення*, який визначає тип значення, що повертає функцією.

### 4.8 Написання надійних програм

Помилки неминуче супроводжують будь-яку складну програму. Самі неприємні, дорогі помилки це ті, які допущені при визначенні основних завдань і цілей додатка, при проектуванні структури його керування й потоків передачі даних, а також помилки, пов'язані з невірною реалізацією алгоритмів. Часто вони не проявляються безпосередньо у вигляді збоїв у роботі програми, а виявляються після досить тривалого використання додатка й вимагають для свого виправлення істотних змін у проекті й програмі. Їх корінь може мати як об'єктивну природу (складність розв'язуваних завдань), так і суб'єктивну (нерозуміння замовником того, що йому необхідно). Як подолати такі помилки?

Виділимо два відомих підходи до цієї проблеми. Перший з них складається з підвищення рівня мов і систем програмування, щоб розроблювач міг оперувати при створенні системи поняттями ПО, для якої вона створюється. Інший підхід пов'язаний з ідеєю швидкого прототипування, тобто створення на ранній стадії розробки системи її працюючого прототипу, у ході експериментів з яким замовник може уточнити свої вимоги.

Одним із факторів, що впливають на надійність програм, є сама мова програмування. Відомо, що мова, у якій є оголошення змінних за замовчуванням, дозволені перетворення даних за замовчуванням у процесі обчислень, немає строгого контролю типів, - така мова є ненадійною, у ній значно легше створити ненадійну програму, яка містить помилку, що важко виявляється. Мову VBA важко вважати надійною мовою, вона скоріше займає по шкалі надійності серединне положення. Багато в чому, це пов'язане з історією його виникнення. Саме тому необхідно вживати ряд заходів для підвищення надійності. Звернімо увагу на деякі з них:

- простежте, щоб всі прапорці на вкладці Editor з меню Tools|Options були включені. Автоматична перевірка синтаксису в процесі написання програм, підказка про значення змінних, підказка про параметри функції, - всі вкрай корисні властивості. Особливу увага звертаємо на прапорець "Require Variable Declaration", при включенні якого в кожен модуль вставляється опція Option Explicit, що примушує явно повідомляти всі змінні. Із цим включеним прапорцем у мови VBA стає одним недоліком менше.
- при оголошенні змінних намагайтеся вказати точний тип змінної й об'єкта. Уникайте оголошень типу Variant й Object. У цьому випадку на Вашій стороні буде контроль типів, що дозволить уникнути багатьох можливих помилок.

- при оголошенні процедур явно вказуйте оператори ByRef й ByVal, пам’ятайте про особливості передачі аргументів по посиланню в VBA.
- не забувайте про розумні розміри модулів і процедур, намагайтеся створювати процедури, модулі й компоненти, що допускають перевикористання.
- гарні специфікації є засадою того, що програма допускає можливість зміни в процесі життєвого циклу, і що вона буде коректно працювати у кінцевого користувача. Тому коментарі в тексті програми, гарна довідкова система, - все це найважливіші фактори, що підвищують надійність програм.

### 4.9 Мистецтво налагодження

Принадність роботи програміста багато в чому пов'язана з налагодженням. Налагодження - це деякий детективний процес. Знов створену програму ми підозрюємо у тому, що вона працює не коректно. На жаль, налагодження не може гарантувати, що програма коректна, навіть якщо всі тести пройшли успішно. Налагодження може довести некоректність програми, але вона не може довести її правильності.

Мистецтво тестера полягає в тому, щоб створити по можливості повну систему тестів, що перевіряє всі можливі галузі обчислень. Пояснимо це на найпростішому прикладі. Нехай програма знаходить суму перших N елементів масиву X, що містить M елементів. Крім "нормального" тесту, що перевіряє ситуацію, у якій 1<N<M, варто перевірити й крайні випадки: N=1, N=M, N=0, N<0, N>M. Але це простий випадок, а цикли звичайно вкладені, і усередині них виробляється розбір випадків, усередині яких свої цикли.

Складність налагодження полягає й у тому, що, виявивши й виправивши помилку, ви одержуєте нову програму, для якої процес налагодження потрібно починати заново, знову пропустивши всі тести. Відомо, що в програмах зустрічаються зачаровані місця, - виправлення однієї помилки веде до появи нової. У таких випадках кращим виходом буває пошук іншого, принципово іншого вирішення завдання.

***Засоби налагодження***

Частина помилок програми ловиться автоматично ще на етапі компіляції. Сюди ставляться всі синтаксичні помилки, помилки невідповідності типів і деякі інші. Однак синтаксично коректна програма має потребу в налагодженні, оскільки, хоча результати обчислень й отримані, але вони не відповідають необхідним специфікаціям. Найчастіше, ще не налагоджена програма – на одних вихідних даних працює правильно, на інші - дає помилковий результат. Мистецтво налагодження полягає в тому, щоб виявити всі ситуації, у яких робота програми призводить до помилкових обчислень. VBA має досить витончені засоби, призначеними для налагодження програм, тобто для виявлення помилок у програмах (тестування) і їхнього виправлення. Є дві групи засобів VBA, що допомагають програмістові виявити й виправити помилки:

1. Перша група дозволяє контролювати хід обчислювального процесу, тобто порядок проходження операторів у процедурах, порядок виклику самих процедур. При необхідності в процесі налагодження дозволяється змінювати цей порядок, можна, наприклад, пропускати виконання деяких операторів, або повторно повертатися до їхнього виконання.
2. Друга група засобів дозволяє контролювати зміни стану обчислювального процесу (значень змінних і властивостей об'єктів) у процесі виконання. І тут можна вмішатися й змінити стан, залежно від того, як ідуть справи, нові значення для тих або інших змінних.

У ході налагодження програма може перебувати в одному з трьох станів: проектування, обчислення й переривання. Закінчивши проектування, можна запустити програму на виконання. Перервавши виконання програми в заданій точці, перейшовши в стан переривання, можна проконтролювати значення змінних і властивостей об'єктів у даній точці й, якщо потрібно, змінити ці значення "вручну". При цьому можна змінити порядок виконуваних операторів і редагувати програмний текст перед продовженням обчислення. Перехід зі стану обчислення в стан переривання може відбуватися через різні причини,

наприклад, по досягненні точки переривання, при виконанні однієї із численних умов переривання, через покрокове виконання програми.

### 4.10 Доказ правильності програм

Ми вже зазначили, що налагодження, засноване на побудові системи тестів, не може довести правильність програми. Тому в теоретичному програмуванні були початі більші зусилля з розроблення методів доказу правильності програм, такі ж строгі, як і методи доказу правильності теорем. На практиці ці методи не одержали широкого поширення з двох причин. По-перше, побудувати доказ правильності програми складніше, ніж написати саму програму. По-друге, помилки в доказі настільки ж можливі, як й у самій програмі. Проте, знання основ доказу правильності програм повинне бути частиною створення програміста.

Нехай P(X,Y) - програма, із заданими вхідними даними X і результатами Y. Предикат Q(X), визначений на вхідних даних, будемо називати предумовою програми P, а предикат R(X,Y), що пов'язує вхідні і вихідні змінні будемо називати постумовою програми P. Будемо також припускати, що в ході своєї роботи програма не міняє своїх вхідних змінних X.

Програма P(X,Y) коректна стосовно предумови Q(X) і постумови R(X,Y), якщо з істинності Q(X) до початку виконання програми необхідно, щоб, будучи запущеною, програма завершила свою роботу й по її завершенню предикат R(X,Y) буде істиною Умову коректності записують у вигляді тріади (тріади Хоора) - Q(X) {P(X,Y)} R(X,Y)

Вже з цього визначення стає ясно, що говорити про правильність треба не взагалі, а стосовно заданих специфікацій, наприклад, у вигляді предумови й постумови. Довести правильність тріади для складних програм, як уже зазначалось, досить складно. Один з методів (метод Флойда) полягає в тому, що програма розбивається на ділянки, розмічені предикатами - Q1, Q2, …QN, R... Перший із предикатів представляє предумову програми, останній - постумову. Тоді доказ коректності зводиться до доказу коректності послідовності тріад:

Q1{P1}Q2; Q2{P2}Q3; …QN{PN}R

Хоча використання цих тверджень не припускає проведення строгого доказу, але це практично реалізована спроба в цьому напрямкі. Все що може бути зроблене для підвищення надійності програми, повинне бути зроблене.

## Список використаної літератури


1. К. Дж. Кейт Введення в системи баз даних/ Пер. с англ. 8-е изд. М.: Издательский дом «Вильямс», 2006.– 1328 с.
2. Пушников А.Ю. Введение в системы управления базами данных. Часть 1. Реляционная модель данных: Учебное пособие/ Изд. Башкирского ун-та. - Уфа, 1999. - 108 с.
3. Пушников А.Ю. Введение в системы управления базами данных. Часть 2: Нормальные формы отношений и транзакции: Учебное пособие/Изд. Башкирского ун-та. - Уфа, 1999. - 138 с.
4. Мартин Грубер. Понимание SQL. /Пер. Лебедева В.Н. М., 1993.– 291 с.
5. Томас Коннолли, Каролин Бегг Базы данных. Проектирование, реализация и сопровождение. Теория и практика.– 3-е изд. М.: Издательский дом «Вильямс», 2003.– 1436 с.
6. Джен Л. Харрингтон Проектирование реляционных баз даннях. – М.: Издательство «Лори», 2006.– 230 с.
7. Киммел, Пол Освой самостоятельно программирование для Microsoft Access 2002 за 24 часа / Пер. с англ.- М.: Издательский дом «Вильямс», 2003.- 480 с.: іл.- парал. тит. англ.
8. Кириллов В.В. Основы проектирования реляционных баз данных: Учебное пособие. - СПб.: ИТМО, 1994. – 90 с.
9. В.В. Кириллов, Г.Ю. Громов. Учебное пособие по SQL: Структурированный язык запросов (SQL). http://www.citforum.ru/database/sql_kg/index.shtml
10. Пасічник В.В. Організація баз даних та знань: підручник для ВНЗ/ В.В. Пасічник, В.А. Резніченко.-К.: Видавнича група BHV,2006.-384с.
11. Бекаревич Ю.Б., Пушкина Н.В. Microsoft Access 2000.- СПб.: БХВ – Санкт- Петербург, 1999.- 480 с., ил.