МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ ЕКОНОМІЧНИЙ УНІВЕРСИТЕТ ІМЕНІ СЕМЕНА КУЗНЕЦЯ

ПРОЄКТУВАННЯ БАЗ ДАНИХ ТА БАЗ ЗНАНЬ

Конспект лекцій для студентів
спеціальності 186 "Видавництво та поліграфія"
першого (бакалаврського) рівня

Харків
ХНЕУ ім. С. Кузнеця
2022

УДК 004.65(042.034)
Г68

Рецензент – доцент кафедри інформаційних комп'ютерних технологій і математики Української інженерно-педагогічної академії, канд. пед. наук *Е. В. Громов*.

Затверджено на засіданні кафедри комп'ютерних систем і технологій.
Протокол № 4 від 10.11.2022 р.

Самостійне електронне текстове мережеве видання

Гордєєв А. С.

Г68 Проєктування баз даних та баз знань [Електронний ресурс] : конспект лекцій для студентів спеціальності 186 "Видавництво та поліграфія" першого (бакалаврського) рівня / А. С. Гордєєв. – Харків : ХНЕУ ім. С. Кузнеця, 2022. – 180 с.

Подано теоретичний матеріал з архітектури баз даних і баз знань систем опрацювання інформації стадій розроблення й експлуатації електронних мультимедійних видань, принципи їхньої організації та особливості використання. Розглянуто впровадження та використання технологічних рішень систем, що ґрунтуються на використанні систем управління баз даних. У результаті освоєння курсу студенти мають набути навичок у впровадженні й експлуатації інформаційних частин програмних систем, основаних на сучасному проєктному підході.

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

УДК 004.65(042.034)

Вступ

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

Мета навчальної дисципліни: формування у студентів фундаментальних теоретичних знань з архітектури баз даних і баз знань, принципів їхньої організації й особливостей використання; набуття навичок у проєктуванні технологічних систем, що ґрунтуються на використанні баз даних та баз знань.

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

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

Предметом вивчення навчальної дисципліни є бази даних та знань в інформаційні системи розроблення й експлуатації електронних мультимедійних видань.

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

Основною метою конспекту лекції з навчальної дисципліни "Проєктування баз даних та баз знань" є надання можливості закріплення студентами спеціальності 186 "Видавництво та поліграфія" таких компетентностей:

- Знання та розуміння предметної галузі та розуміння професійної діяльності.
- Здатність застосовувати знання у практичних ситуаціях.

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

Розділ 1
Базові концепції СУБД

1. Сучасні видавничі інформаційні системи

Мета – визначення понять "інформаційна система", "бази даних", "архітектура інформаційної системи" та їхніх основних складових. Розуміння принципів побудови сучасних видавничих інформаційних систем.

Основні питання

1.1. Класифікація інформаційних систем.
1.2. Архітектура інформаційної системи.
1.3. Системи управління базами даних.
1.4. Способи розроблення та виконання програм.
1.5. Життєвий цикл програмного забезпечення.

Ключові слова: інформація, система, архітектура, бази даних, адміністратор баз даних, клієнт-сервер.

1.1. Класифікація інформаційних систем

В основі вирішення багатьох завдань лежить опрацювання інформації. Для полегшення опрацювання інформації створюють інформаційні системи (ІС). У широкому розумінні під визначення ІС підпадає будь-яка система опрацювання інформації.

Інформаційна система становить сукупність технічного, програмного та організаційного забезпечення й навіть персоналу, призначена для здобуття права своєчасно забезпечувати людей необхідною інформацією.

Інформаційну систему в такому контексті слід розглядати як середовище, що забезпечує цілеспрямовану діяльність організації. Тобто вона становить сукупність таких компонентів, як персонал та інформація, а також процедури й обладнання, об'єднані інформаційними комп'ютерними технологіями, для формування організації як єдиного цілого та забезпечення її цілеспрямованої діяльності (рис. 1.1).

Рис. 1.1. **Основні компоненти інформаційної системи**

За сферою застосування ІС можна розподілити на системи, які використовують у виробництві, освіті, охороні здоров'я, науці, військовій справі, соціальній сфері, торгівлі та інших галузях (табл. 1.1).

Таблиця 1.1

Класифікація інформаційних систем

Інформаційні системи			
галузі застосування	цільові функції	у міру розподілу	за охопленням завдань (масштабності)
Виробництво	Керівні	Локальні (desktop)	Персональна
Освіта		Розподілені (distributed)	
Охорона здоров'я	Інформаційно-довідкові	Файл-серверні	Групова
Наука			
Військова справа			
Соціальна сфера	Підтримання ухвалення рішень	Клієнт-серверні	Корпоративна
Торгівля			
Інші			

За цільової функції ІС можна умовно розподілити на керівні, інформаційно-довідкові, підтримання ухвалення рішень.

Керівна інформаційна система – цифрова система контролю за деяким реальним об'єктом або управління ним. Цими системами вирішують завдання, не пов'язані з необхідністю в ухваленні рішення в реальному часі (розрахунку моделювання, офісні завдання). Ці ІС вирішують завдання, які мають чітко виявлену специфіку.

Завдання управління в поліграфії вирішували завжди. У докомп'ютерну еру, це зрозуміло, застосовували т. зв. паперові технології. Поява персональних комп'ютерів і створення локальних комп'ютерних мереж викликала бурхливу зміну технологій управління в поліграфічній галузі.

У кінці XX ст. інформаційно-керівні системи й інформаційні технології оформили в шість базових типів:

1. Системи управління, повністю засновані на паперових засобах документообігу та обліку.

2. Змішані паперово-комп'ютерні системи. Подібні системи побудовано на поєднанні цифрового опрацювання та зберігання даних і використання паперових технологій управління. Переважно як сховища даних найчастіше використовують MS Excel, рідше MS Access. На підтримання передавання даних впливають поштові або системи документообігу, наприклад, MS Exchange або Lotus Notes.

3. Повністю цифрові системи управління, що використовують різнорідні середовища зберігання й опрацювання інформації (наприклад, поєднань MS Excel, MS Project, MS Access Lotus Notes або будь-яких інших).

4. Системи виробничого управління й обліку, побудовані на основі єдиної неспеціалізованої загально використовуваної комп'ютерної системи (наприклад, Axapta/Navision або 1С).

5. Вузькоспеціалізовані галузеві системи, розроблені за індивідуальними замовленнями конкретних друкарень-замовників.

6. Спеціалізовані галузеві системи управління та обліку, орієнтовані на специфіку роботи поліграфічних підприємств (наприклад: Prinance/ Prinect, DISO, HiFlex, ASystem, Apler-Поліграфія, EFI-PrintSmith, SisTrade, PrintEffect).

Інформаційно-довідкова система – система, призначена для зберігання, пошуку та опрацювання інформації, і відповідні організаційні ресурси (людські, технічні, фінансові ресурси тощо), які забезпечують і поширюють інформацію. Ці системи призначені для своєчасного забезпечення конкретних інформаційних потреб у межах певної предметної галузі, за того ж результатом функціонування інформаційних систем є інформаційна продукція **–** документи, інформаційні масиви, бази даних та інформаційні послуги.

Активна комп'ютеризація видавничо-поліграфічного процесу (ВПП) привела до істотних його змін, насамперед це скорочення кількості використовуваних технологічних платформ. Ще 25 **–** 30 років тому ВПП будувався як послідовна зміна цілого ряду різнотипних технологічних платформ **–** редакційної, складальної, фотолітографії, верстальних, друкованої, брошурувальних та ін. **–** із формалізованими інтерфейсними процедурами.

У комп'ютеризованому ж ВПП домінують три платформи – додрукарська, подана здебільшого настільними видавничими системами; друкарська та післядрукарська (брошурувально-палітурна). Причому, як показує практика, інтерфейси між ними формалізовано недостатньою мірою.

Система підтримання ухвалення рішень (Decision Support System, DSS) – комп'ютерна автоматизована система, метою якої є допомога тим людям, хто ухвалює рішення у складних умовах для повного й об'єктивного аналізу предметної діяльності. Це означає, що вона видає інформацію (у друкованій формі, або на екрані монітора, або звуком), ґрунтуючись на вхідних даних, що допомагає людям швидко й точно оцінити ситуацію та ухвалити рішення. Ці системи виникли в результаті злиття управлінських інформаційних систем і систем управління базами даних.

У сучасних економічних умовах усе більше фірм та індивідуальних замовників прагнуть, використовуючи можливості настільних видавничих систем, узяти максимально можливу частину видавничого процесу на себе. Заміщення колективу вузьких фахівців одним-двома людьми, що мають, найчастіше, лише загальне уявлення про видавничий процес і його "підводні камені" та вимушені працювати у гранично стислі строки, не може не позначитися на якості продукції, що випускають. Одним із можливих шляхів виходу із ситуації, що склалася, є розроблення та впровадження у практику невеликих видавничих підрозділів проблемно-орієнтованих систем підтримання ухвалення рішень (СПУР), які акумулюють досвід і знання фахівців та на цій основі дозволяють навіть порівняно недосвідченому користувачеві успішно розв'язувати проблеми.

У міру розподілу розрізняють:

локальні інформаційні системи (desktop), у яких усі компоненти (БД, СУБД, клієнтські програми) працюють на одному комп'ютері;

розподілені інформаційні системи (distributed), у яких компоненти розподілено за кількома комп'ютерами.

Розподілені ІС, своєю чергою, розподіляють на: файл-серверні та клієнт-серверні.

У *файл-серверних ІС* база даних міститься на файловому сервері, а СУБД і клієнтські програми розміщено на робочих станціях.

У *клієнт-серверних ІС* база даних і СУБД містяться на сервері, а на робочих станціях розміщено клієнтські програми.

Зауважмо, що іноді використовують більш вузьке трактування поняття ІС як сукупності апаратно-програмних засобів, задіяних для вирішення

прикладного завдання. В організації, на приклад, можуть бути інформаційні системи, на які, відповідно, покладено завдання обліку кадрів і матеріально-технічних засобів, розрахунок із постачальниками та замовниками, бухгалтерський облік тощо.

Класифікацію інформаційних систем за *охопленням завдань* (масштабності) можна визначити так:

Персональна інформаційна система, призначена для вирішення певного кола завдань однієї людини. Прикладом такої системи є система опрацювання контенту.

Групова інформаційна система, орієнтована на колективне використання інформації членами робочої групи або підрозділу.

Корпоративна інформаційна система в ідеалі охоплює всі інформаційні процеси цілого підприємства, досягаючи їхньої повної погодженості, повноти та прозорості. Такі системи у видавничій галузі називають комплексними редакційно-видавничими системами.

Банк даних (БД) є різновидом ІС, у якій реалізовано функції централізованого зберігання й накопичення опрацьованої інформації, організованої в одну або кілька баз даних.

Банк даних у загальному випадку складається з таких компонентів: бази (кількох баз) даних, системи управління базами даних, словника даних, адміністратора, обчислювальної системи й обслуговчого персоналу.

Коротенько розгляньмо названі компоненти й деякі пов'язані з ними важливі поняття.

База даних (БД) становить сукупність спеціальним чином організованих даних, що зберігають у пам'яті обчислювальної системи та відображають стан об'єктів і їхні взаємозв'язки в цій предметній галузі.

Логічну структуру збережених у базі даних називають *моделлю подання даних*. До основних моделей подання даних зараховують такі: ієрархічну, мережеву, реляційну, постреляційну, багатовимірну та об'єктно-орієнтовану.

Словник даних (СД) є підсистемою БД, призначений для централізованого зберігання інформації, структури, взаємозв'язків файлів БД один з одним, типів даних і форматах їхнього подання, належності даних користувачам, кодів блокування та розмежування доступу тощо.

Функціонально СД наявний у всіх БД, але не завжди компонент, що виконує ці функції, має саме таку назву. Найчастіше функцій СД виконує СУБД.

Адміністратор бази даних (АБД) – це особа або група осіб, що відповідають за вироблення вимог до БД, її проєктування, створення, ефективне використання та супровід.

У процесі експлуатації АБД зазвичай стежить за функціонуванням інформаційної системи, забезпечує захист від несанкціонованого доступу, контролює надмірність, несуперечливість, збереження та достовірність інформації, що зберігають у БД. Для одного користувача інформаційних систем функції АБД зазвичай покладають на осіб, які безпосередньо працюють із застосунком БД.

Обчислювальна система (ОС) становить сукупність взаємопов'язаних і погоджено діючих ЕОМ або процесорів та інших пристроїв, що забезпечують автоматизацію процесів приймання, опрацювання та видавання інформації споживачам.

Оскільки основними функціями БД є зберігання й опрацювання даних, то використовувана ОС, разом із прийнятною потужністю центральних процесорів (ЦП), мусить мати достатній обсяг оперативної та зовнішньої пам'яті прямого доступу.

Обслуговчий персонал виконує функції підтримання технічних і програмних засобів у працездатному стані. Він виконує профілактичні, регламентні, відновлювальні та інші роботи, згідно із планами, а також у міру необхідності.

1.2. Архітектура інформаційної системи

Однією з найбільш поширених проблем більшості організацій є різноманітність інформаційних систем. Спочатку використання різних інформаційних систем, які іноді дублюють одна одну, не здається проблемою. Однак у якийсь момент під час спроби внести зміни в бізнес-процес, який стосується безлічі підрозділів і відомств, доводиться змінювати велику кількість наявних інформаційних систем, що не завжди можливо або потребує серйозних ресурсів.

Чи не менш гострою є проблема "клаптикової" автоматизації. Під час проведення аудиту проєкту впровадження інформаційної системи часто можна бачити, що за великого відсотка автоматизованих функцій наскрізної автоматизації бізнес-процесу немає. Це призводить до того, що під час переходу потоку робіт від одного підрозділу до іншого змінюються носії, формати та склад інформації. Прикладом цього може бути друкування

документа з однієї системи й передавання його в паперовій формі в інший підрозділ, де цей документ знову переводять у цифрову форму та заносять у свою локальну інформаційну систему, що сильно знижує ефективність автоматизації.

Зростальна залежність бізнес-процесів від якості й надійності ІТ-товарів потребує системного підходу до їхньої автоматизації та побудови архітектури інформаційних систем.

Слід зазначити, що нині термін "архітектура інформаційної системи" використовують дуже широко, але водночас він має настільки ж безліч різних трактувань.

Одне з популярних визначень архітектури наведено у стандарті ISO/IEC/IEEE 42010:2011 "Системна і програмна інженерія – опис архітектури" який є міжнародним стандартом для опису архітектури систем і програмного забезпечення.

ISO/IEC/IEEE 42010:2011 визначає вимоги щодо опису системи, програмного забезпечення та корпоративних архітектур. Вона покликана уніфікувати практику опису архітектури шляхом визначення стандартних термінів, надаючи концептуальну основу для вираження, спілкування й аналізу та виявлення вимог, що застосовують до описів архітектури, основи архітектури та мов опису архітектури.

ICO 42010 визначає ряд термінів:

- *Розроблення архітектури*: проєктування, підтримання та вдосконалення інформаційної системи протягом усього життєвого циклу системи.
- *Архітектура*: основні поняття та властивості системи, утілені у її елементах, відносинах, а також принципах її дизайну й еволюції.
- *Опис архітектури* (абревіатура OD): робочий продукт, який використовують для виявлення архітектури.
- *Мова опису архітектури* (абревіатура ADL): будь-яка форма виявлення для використання в описі архітектури.
- *Рамки архітектури*: принципи й методи опису архітектури, створеної для конкретної сфери застосування і/або спільнот зацікавлених сторін.

Відповідно до цього уявлення, система має архітектуру, якщо вона може бути описаною з різних поглядів зацікавлених осіб, що розглядають інформаційну систему. Кожному поглядові на архітектуру системи відповідає певне уявлення, основу якого становить набір моделей. Однак цей стандарт не визначає структуру власне архітектури підприємства. Наприклад, ідеться про те, що необхідно мати різні уявлення архітектури, але водночас не вказано, які це мають бути уявлення.

Для архітектури інформаційної системи традиційними є такі перспективи або рівні опису архітектури (рис. 1.2).

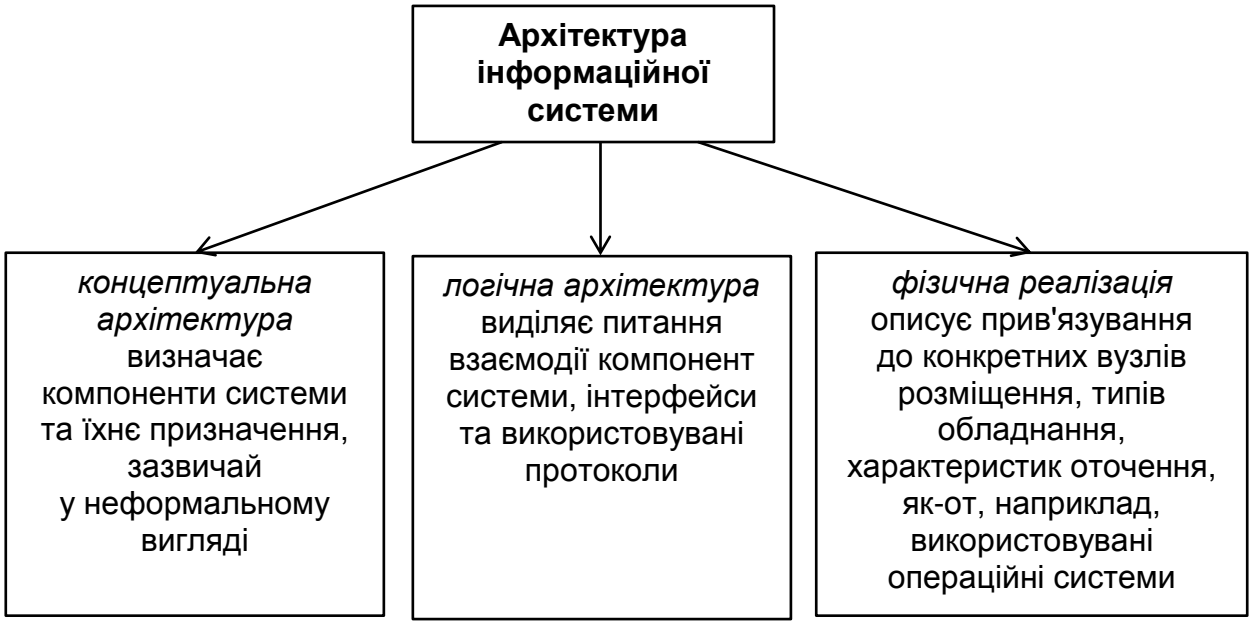


Рис. 1.2. **Архітектура інформаційної системи**

концептуальна архітектура визначає компоненти системи та їхнє призначення зазвичай у неформальному вигляді. Це уявлення часто використовують для обговорення з нетехнічними фахівцями, як-от керівництво, бізнес-менеджери та кінцеві користувачі функціональних характеристик системи (що система має вміти робити, переважно, із погляду кінцевого користувача);

логічна архітектура виділяє, насамперед, питання взаємодії компонент системи, інтерфейси та використовувані протоколи. Це уявлення дозволяє ефективно організувати паралельне розроблення;

фізична реалізація, яка описує прив'язування до конкретних вузлів розміщення, типів обладнання, характеристик оточення, як-от, наприклад, використовувані операційні системи.

Розглянуті раніше положення ISO/IEC/IEEE 42010:2011 задають лише рамкову модель розроблення архітектури, але є корисними для розуміння основ архітектурного підходу. Створення архітектури інформаційної системи починають із розроблення бізнес-архітектури.

Бізнес-архітектура виявляє цілісні, багатовимірні бізнес-погляди на можливості підприємства, інформацію й організаційну структуру, а також відносини між цими бізнес-поглядами та стратегіями, продуктами й зацікавленими сторонами.

Для переходу від опису архітектури бізнес-процесів до опису архітектури інформаційних систем необхідно формалізувати кілька додаткових предметних галузей. Насамперед, слід описати архітектуру даних, побудовану на підставі тієї інформації й документів, які використовують у бізнес-процесах, а потім необхідно сформувати архітектуру застосунку та архітектуру технологій (рис. 1.3).

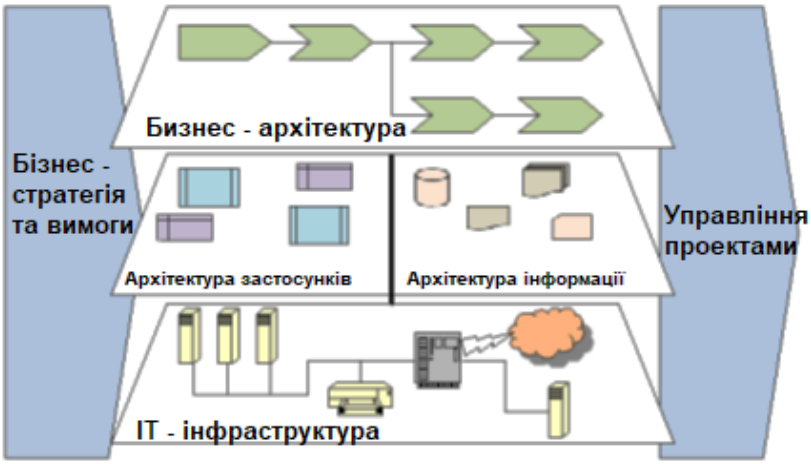


Рис. 1.3. **Перехід від бізнес-архітектури до архітектури інформаційних систем**

Для побудови архітектури даних необхідно виділити основні сутності й агрегувати на них усі кванти інформації, зібрані з опису бізнес-процесів. У результаті використання стандартної методології опису даних – моделі "сутність – зв'язок" (Entity – Relationship Model – ERM) – можна чітко структурувати всю інформацію, визначивши цим структуру таблиць бази даних, що однозначним чином формалізує структуру даних у компанії у прив'язуванні до наявних бізнес-процесів і буде зрозумілим для IT-фахівців.

Наступним етапом є перехід від архітектури бізнес-процесів та архітектури даних до створення архітектури застосунків. На цьому етапі необхідно визначити класи інформаційних систем, необхідних для автоматизації, а потім необхідні модулі для кожної інформаційної системи.

Після того як архітектуру застосунків сформовано, подальшим етапом є створення архітектури технологій, що становить елементи IT-інфраструктури, як-от сервери, мережеві елементи та інше обладнання, необхідне для підтримання функціонування застосунків.

Як уже зазначалося раніше, ефективність функціонування інформаційної системи багато в чому залежить від її архітектури. Історично першими виникли розподілені ІС із застосуванням *файл-сервера.* У таких ІС за запитами користувачів файли бази даних передають на персональні комп'ютери, де й здійснюють їхнє опрацювання. Недоліком такого варіанта архітектури є висока інтенсивність передавання опрацьованих даних. Причому найчастіше передають надлишкові дані: незалежно від того, скільки записів із бази даних потрібно користувачеві, файли бази даних передають повністю.

Нині перспективною є архітектура "клієнт – сервер" (рис. 1.4).

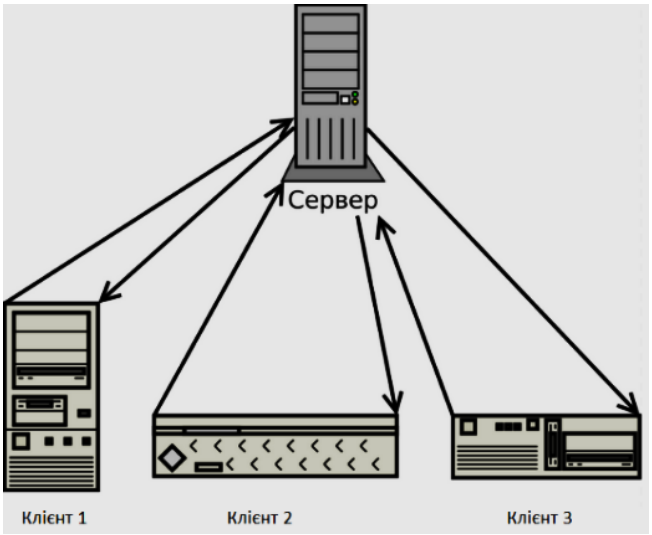


Рис. 1.4. **Архітектура "клієнт – сервер"**

"Клієнт – сервер" – обчислювальна або мережева архітектура, у якій завдання або мережеве навантаження розподілено між постачальниками послуг, названими *серверами*, і замовниками послуг, названими *клієнтами*.

Фактично клієнт і сервер – це програмне забезпечення. Зазвичай ці програми розташовано на різних обчислювальних машинах та взаємодіють між собою через обчислювальну мережу за допомогою мережевих протоколів, але вони можуть бути розташованими також і на одній машині. Програми-сервери очікують від клієнтських програм запитів і надають їм свої ресурси у вигляді даних (наприклад, завантаження файлів за допомогою HTTP, FTP, BitTorrent, потокове мультимедіа або робота з базами даних) або у вигляді сервісних функцій (наприклад, робота з електронною

поштою, спілкування за допомогою систем миттєвого обміну повідомленнями або перегляд вебсторінок у всесвітній павутині).

Оскільки одна програма-сервер може виконувати запити від безлічі програм-клієнтів, її розміщують на спеціально виділеній обчислювальній машині, налаштованій особливим чином, зазвичай, спільно з іншими програмами-серверами, тому продуктивність цієї машини має бути високою. Через особливу роль такої машини в мережі, специфіку її обладнання та програмного забезпечення, її також називають *сервером*, а машини, які виконують клієнтські програми, відповідно, *клієнтами*.

Перевагами організації інформаційної системи за архітектурою "клієнт – сервер" є такі:

- немає дублювання коду програми-сервера програмами-клієнтами;
- оскільки всі обчислення виконують на сервері, то вимоги до комп'ютерів, на котрих установлений клієнт, знижують;
- усі дані зберігають на сервері, який, зазвичай, захищений набагато краще за більшість клієнтів. На сервері простіше організувати контроль за повноваженнями, щоб діставати доступ до даних тільки клієнтам із відповідними правами доступу.

До недоліків організації інформаційної системи за архітектурою "клієнт – сервер" належать такі:

- непрацездатність сервера може зробити непрацездатною всю обчислювальну мережу. Непрацездатним сервером слід уважати сервер, продуктивності якого не вистачає на обслуговування всіх клієнтів, а також сервер, що є на ремонті, профілактиці тощо;
- підтримання роботи цієї системи потребує окремого фахівця – системного адміністратора;
- висока вартість обладнання.

1.3. Системи управління базами даних

Система управління базами даних (СУБД) – це комплекс мовних і програмних засобів, призначений для створення, ведення та сумісного використання БД багатьма користувачами.

Для роботи з інформацією, яку зберігають у базі даних, СУБД надає програмам і користувачам такі два типи мов, як-от:

• *мова опису даних* – високорівнева непроцедурна мова декларативного типу, призначена для опису логічної структури даних;

• *мова маніпулювання даними* – сукупність конструкцій, що забезпечують виконання основних операцій із роботи з даними: уведення, модифікацію та вибірку даних за запитами.

Названі мови в різних СУБД можуть мати відмінності. Найбільшого поширення набули дві стандартизовані мови: QBE (Que ry By Example) – мова запитів за зразком і SQL (Structured Query Language) – структурована мова запитів. QBE переважно має властивості мови маніпулювання даними, SQL поєднує в собі властивості мов обох типів – опису та маніпулювання даними.

Для створення й управління персональними БД і застосунків, що працюють із ними, використовують такі СУБД, як Access і Visual FoxPro фірми Microsoft, Paradox фірми Borland.

Корпоративну БД створюють, підтримують і вона функціонує під управлінням сервера БД, наприклад, Microsoft SQL Server або Oracle Server.

Використання архітектури "клієнт – сервер" дає можливість поступового нарощування інформаційної системи підприємства, по-перше, у міру розвитку підприємства; по-друге, у міру розвитку самої інформаційної системи.

Розподіл загальної БД на корпоративну БД і персональні БД дозволяє знизити складність проєктування БД, порівняно із централізованим варіантом, а значить, знизити ймовірність помилок у процесі проєктування та його вартість.

Найважливішою перевагою застосування БД в інформаційних системах є забезпечення незалежності даних від прикладних програм. Це дає можливість користувачам не займатися проблемами подання даних на фізичному рівні: розміщення даних у пам'яті, методів доступу до них тощо.

Такої незалежності досягають підтримуваним СУБД багаторівневим поданням даних у БД на логічному (для користувача) і фізичному рівнях. Завдяки СУБД і наявності логічного рівня подання даних, забезпечують відокремлення концептуальної (понятійної) моделі БД від її фізичного подання в пам'яті ЕОМ.

Розгляньмо класифікацію СУБД та основні їхні функції. Як основні класифікаційні ознаки можна використовувати такі: вид програми, характер використання, моделі даних (рис. 1.5).

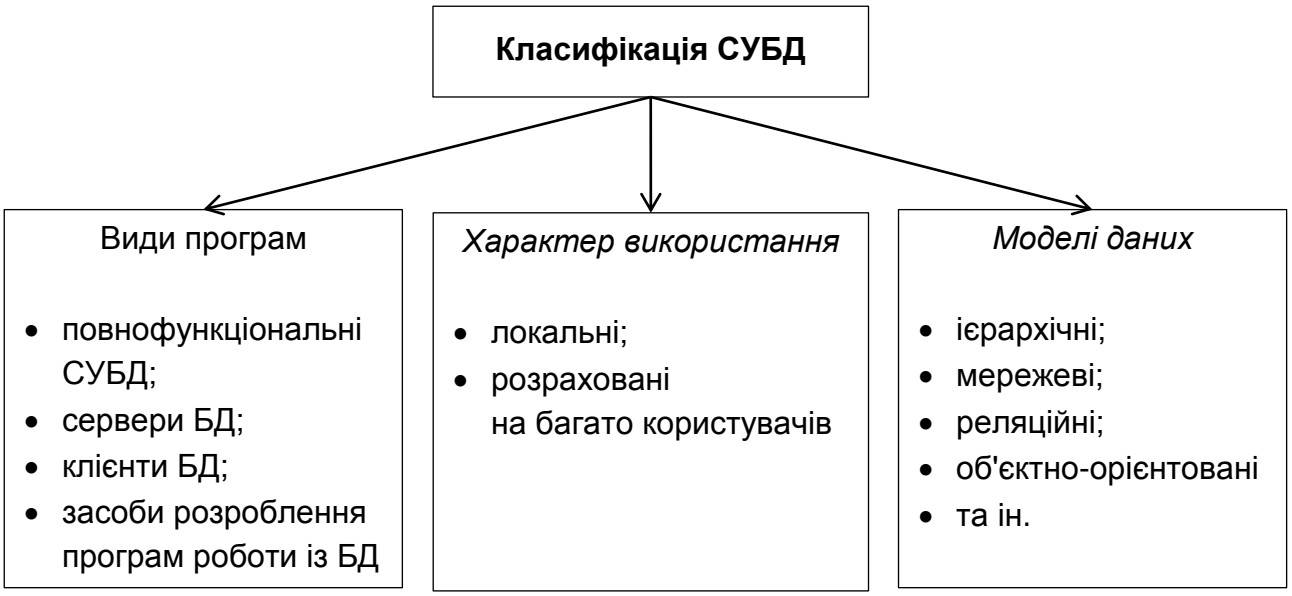


Рис. 1.5. **Класифікація СУБД**

До СУБД належать такі *основні види програм*:

- повнофункціональні СУБД;
- сервери БД;
- клієнти БД;
- засоби розроблення програм роботи із БД.

Повнофункціональні СУБД становлять традиційні системи, які спочатку виникли для великих машин, потім для мінімашин і персональних ЕОМ. До повнофункціональним СУБД належать такі пакети, як Clarion Database Deweidper, DataEase, DataFlex, dBase IV, Microsoft Access, Microsoft FoxPro і Paradox R: BASE. Зазвичай повнофункціональні СУБД мають розвинений інтерфейс, що дозволяє за допомогою команд меню виконувати основні дії із БД: створювати й модифікувати структуру таблиць, уводити дані, формувати запити, розробляти звіти, виводити їх на друкування тощо. Для створення запитів і звітів не обов'язкове програмування, а зручно користуватися мовою QBE.

Сервери БД призначено для організації центрів опрацювання даних у мережах ЕОМ. Ця група БД нині є менш численною, але їхня кількість поступово зростає. Сервери БД реалізують функції управління базами даних, запитувані іншими (клієнтськими) програмами зазвичай за допомогою операторів SQL. Прикладами серверів БД є такі програми: NetWare SQL (Novell), MS SQL Server (Microsoft), InterBase (Borland), SQLBase Server (Gupta), Intelligent Database (Ingress).

Як **клієнтські програми** для серверів БД у загальному випадку можна використовувати різні програми: електронні таблиці, текстові процесори, програми електронної пошти тощо. До того ж елементи пари "клієнт – сервер" можуть належати одному або різним виробникам програмного забезпечення.

Засоби розроблення програм роботи із БД можна використовувати для створення різновидів таких програм:

- клієнтських програм;
- серверів БД і їхніх окремих компонентів;
- програм, призначених для користувача застосунків.

Програми першого та другого виду є досить нечисленними, оскільки призначеними, головним чином, для системних програмістів. Пакетів третього виду є набагато більше, але менше, ніж повнофункціональних СУБД.

До засобів розроблення застосунків користувача належать системи програмування, наприклад, Clipper; різноманітні бібліотеки програм для різних мов програмування й навіть пакети автоматизації розроблень (зокрема систем типу "клієнт – сервер"). Серед найбільш поширених можна назвати такі інструментальні системи: Delphi і Power Builder (Borland), Visual Basic (Microsoft), SILVERRUN (Computer Advisers Inc.), S-Designor (SDP і Powersoft) і ERwin (LogicWorks).

За характером використання СУБД розподіляють на локальні та розраховано на багато користувачів.

Локальні СУБД зазвичай забезпечують можливість створення персональних БД і недорогих застосунків, що працюють із ними. Персональні СУБД або розроблені з їхньою допомогою застосунки часто можуть відігравати роль клієнтської частини, розрахованої на багато користувачів СУБД. До персональних СУБД, наприклад, належать Visual FoxPro, Paradox, Clipper, dBase, Access та ін.

Організація функціонування локальної ІС на одному комп'ютері в середовищі деякої операційної системи (ОС) є можливою за допомогою таких варіантів використання програмних засобів:

- повної СУБД;
- застосунків та усіченої (ядра) СУБД;
- незалежної програми.

Перший спосіб зазвичай застосовують у випадках, якщо в дисковій пам'яті комп'ютера поміщено всю СУБД і її часто використовують для доопрацювання застосунку.

Взаємодія користувача із СУБД відбувається безпосередньо через інтерфейс СУБД або за допомогою програми.

Основна перевага схеми – простота розроблення й супроводу БД та застосунків за наявності розвинених відповідних засобів розроблення й сервісних засобів. Недоліком цієї схеми є витрати дискової пам'яті на зберігання програми СУБД.

Застосунок із ядром СУБД використовують для досягнення таких цілей:

зменшення обсягу, займаного СУБД простору жорсткого диска й оперативної пам'яті;

підвищення швидкості роботи програми;

захисту програми від модифікації з боку користувача (зазвичай ядро не містить засобів розроблення застосунків).

Прикладом такого підходу є використання модуля FoxRun системи FoxBase+. Із сучасних СУБД зазначмо Microsoft Access, що містить додатковий пакет Microsoft AccessDeveloper's Toolkit.

Із його допомогою можна створювати та переносити на дискетах скорочену (run-time) версію Microsoft Access, яка не містить інструментів розроблення.

Перевагами використання ядра СУБД, порівняно з використанням повної версії СУБД, є такі: менше споживання ресурсів пам'яті комп'ютера, прискорення роботи програми й можливість захисту програми від модифікації. До основних недоліків можна зарахувати все ще значний обсяг дискової пам'яті, необхідної для зберігання ядра СУБД, і недостатньо високу швидкодію роботи застосунків.

За третього способу організації ІС вихідну програму попередньо компілюють – перетворюють на послідовність виконуваних машинних команд. У результаті виходить готова до виконання незалежна програма, яка не потребує для своєї роботи ні всієї СУБД, ні її ядра.

Основними перевагами цього варіанта, порівняно із двома попередніми, є економія зовнішньої й оперативної пам'яті комп'ютера, прискорення виконання програми та повний захист застосунків від модифікації. До недоліків можна зарахувати трудомісткість доопрацювання застосунків і відсутність можливості використовувати стандартні засоби СУБД з обслуговування БД.

Розраховані на багато користувачів СУБД містять сервер БД і клієнтську частину і, зазвичай, можуть працювати в неоднорідному обчислювальному середовищі (із різними типами ЕОМ та операційними системами).

До розрахованих на багато користувачів СУБД належать, наприклад, СУБД Oracle і Informix. База даних, зазвичай, містить дані, необхідні багатьом користувачам. Дістання одночасного доступу декількох користувачів до загальної бази даних можливе в разі установлення СУБД у локальній мережі ПК і створення бази даних, розрахованої на багато користувачів.

СУБД стежить за розмежуванням доступу різних користувачів до загальної бази даних і забезпечує захист даних за одночасної роботи користувачів із загальними даними. Автоматично забезпечено захист даних від одночасного їхнього коригування декількома користувачами-клієнтами. Розрізняють БД з архітектурою "файл – сервер" і "клієнт – сервер".

Файл-серверні СУБД. На сервері тільки зберігають файли бази даних, опрацювання яких переважно відбувається на комп'ютерах користувача. Сервер опрацьовує клієнтські запити й передає на робочі станції файли бази даних (рис. 1.6). Під час унесення змін до бази даних, СУБД із комп'ютера користувача блокує файли на сервері, щоб інші клієнти в цей момент не могли їх змінити.

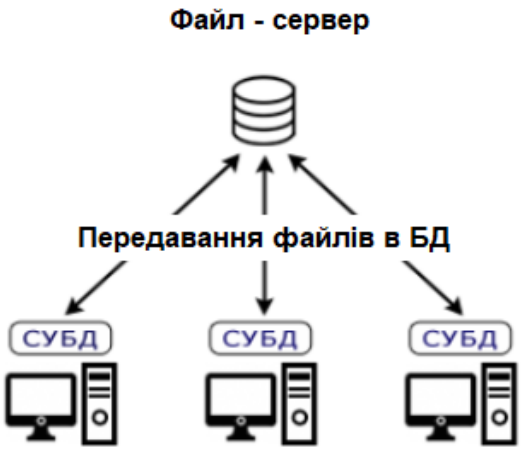


Рис. 1.6. **Файл-серверні СУБД**

Подібного роду технологія має певні недоліки:

- під час пошуку даних уся БД копіюється по мережі на комп'ютер користувача, що створює зайве мережеве навантаження;
- ненадійний захист від неправомірного доступу до даних;
- за великої кількості під'єднань знижується надійність у разі внесення змін до бази даних;
- знижується продуктивність системи загалом за високої інтенсивності доступу до одних і тих самих даних.

Клієнт-серверні СУБД. У цій концепції на сервері, крім зберігання централізованої бази даних, виконують усю роботу з опрацювання даних. На комп'ютері користувача функціонує клієнтська програма, яка відправляє запити до СУБД для виконання операцій із даними (рис. 1.7). Специфікою клієнт-серверної архітектури є використання мови структурованих запитів SQL для управління даними.

Рис. 1.7. **Клієнт-серверні СУБД**

За *використовуваною моделлю даних* СУБД розподіляють на ієрархічні, мережеві, реляційні, об'єктно-орієнтовані та інші типи. Деякі СУБД можуть одночасно підтримувати кілька моделей даних.

1.4. Способи розроблення та виконання програм

Застосунок становить програму або комплекс програм, що забезпечують автоматизацію опрацювання інформації для прикладної задачі. Застосунки можна створювати в середовищі або поза середовищем СУБД за допомогою системи програмування, що використовує засоби доступу до БД, наприклад, Delphi або C++ Builder. Застосунки, розроблені в середовищі СУБД, часто називають *застосунками СУБД*, а застосунки, розроблені поза СУБД, – *зовнішніми застосунками*.

Сучасні СУБД дозволяють вирішувати широке коло завдань із роботи з базами даних без розроблення програми. Проте є випадки, коли доцільно розробити застосунок. Наприклад, якщо потрібна автоматизація маніпуляцій із даними, термінальний інтерфейс СУБД є недостатньо розвиненим, або наявні в СУБД стандартні функції з опрацювання інформації не влаштовують користувача. Для розроблення застосунків СУБД

мусить мати програмний інтерфейс, основу якого становлять функції відповідної мови програмування.

Наявні СУБД підтримують такі технології розроблення застосунків:

- ручне кодування програм (Clipper, FoxPro, Paradox);
- створення текстів програм за допомогою генераторів (FoxApp у FoxPro, Personal Programmer у Paradox);
- автоматичну генерацію готового застосунку методами візуального програмування (Delphi, Access, Paradox for Windows).

За *ручного кодування* програмісти вручну набирають текст програм застосунків, після чого виконують їхнє налагодження.

Використання генераторів спрощує розроблення застосунків, оскільки водночас можна діставати програмний код без ручного набору. Генератори застосунків полегшують розроблення основних елементів застосунків (меню, екранних форм, запитів тощо), але часто не можуть повністю виключити ручне кодування.

Засоби візуального програмування застосунків є подальшим розвитком ідей використання генераторів застосунків. Застосунок водночас будують із готових "будівельних блоків" за допомогою зручного інтегрованого середовища. У разі потреби розробник легко може вставити в застосунок свій код. Інтегроване середовище, зазвичай, надає потужні засоби створення, налагодження та модифікації застосунків. Використання засобів візуального програмування дозволяє в найкоротші строки створювати більш надійні, привабливі й ефективні програми, порівняно із застосунками, побудованими першими двома способами.

Застосунок, що розробляють, зазвичай складається з одного або декількох файлів операційної системи.

Якщо основним файлом програми є виконуваний файл (наприклад, exe-файл), то цей застосунок, імовірніше, є незалежним застосунком, який виконують автономно від середовища СУБД. Розроблення незалежного застосунку на практиці здійснюють шляхом компіляції початкового програмного коду. Програмний код може бути визначеним різними способами: шляхом набору тексту вручну, здобутих за допомогою генератора застосунків або середовища візуального програмування.

Незалежні програми дозволяють визначати, наприклад, СУБД FoxPro і система візуального програмування Delphi. Зазначмо, що за допомогою засобів Delphi зазвичай незалежні програми не розробляють, оскільки це досить трудомісткий процес, а залучають процесор баз даних BDE

(Borland DataBase Engine), який відіграє роль ядра СУБД. Одним із перших розробок застосунків для персональних ЕОМ є система Clipper, що становить "чистий компілятор".

Здебільшого застосунок не можна виконувати без середовища СУБД. Виконання програми полягає в тому, що СУБД аналізує вміст файлів програми й автоматично будує необхідні машинні команди. Інакше кажучи, застосунок виконують методом інтерпретації.

Режим інтерпретації реалізовано в багатьох сучасних СУБД, наприклад, Access, Visual FoxPro і Paradox, а також у СУБД недавнього минулого, наприклад, FoxBase та FoxPro.

Крім цього, є системи, що використовують проміжний варіант між компіляцією й інтерпретацією, так звану псевдокомпіляцію. У таких системах вихідна програма шляхом компіляції перетвориться на проміжний код (псевдокод) і записується на диск. У цьому вигляді її в деяких системах дозволено навіть редагувати, але головна мета псевдокомпіляції – перетворити програму до вигляду, що прискорює процес її інтерпретації. Такий прийом широко застосовувався в СУБД, що працюють під управлінням DOS, наприклад, Foxbase+ і Paradox 4.0 / 4.5 for DOS.

У СУБД, що працюють під управлінням Windows, псевдокод частіше використовують для того, щоб заборонити модифікувати застосунок. Це корисно для захисту від випадкового або навмисного псування програми, що працює. Наприклад, такий прийом застосовано в СУБД Paradox for Windows, де допускають розроблені екранні форми та звіти перетворювати на відповідні об'єкти, що не підлягають редагуванню.

Під час вибору засобів для розроблення програми слід ураховувати три основні чинники: ресурси комп'ютера, особливості застосування (потреба в модифікації функцій програми, час на розроблення, необхідність у контролі за доступом та підтриманні цілісності інформації) і мета розроблення (відчужують програмний продукт або систему автоматизації своєї повсякденної діяльності).

Для користувача, що має сучасний комп'ютер і який планує створити нескладний застосунок більше підійде СУБД типу, що інтерпретує. Нагадаймо, що такі системи є досить потужними, мають високорівневі засоби, зручні для розроблення та налагодження, дозволяють швидко виконати розроблення та забезпечують зручний супровід і модифікацію програми.

Під час використання комп'ютера зі слабкими характеристиками краще зупинити свій вибір на системі із засобами розроблення незалежних застосунків. Водночас слід мати на увазі, що найменша зміна в застосунку тягне за собою циклічне повторення етапів програмування, компіляції та відлагодження програми.

1.5. Життєвий цикл програмного забезпечення

Життєвий цикл програмного забезпечення (ПЗ) – період часу, який починається з моменту ухвалення рішення про необхідність у створенні програмного продукту й закінчується в момент його повного вилучення з експлуатації.

Стандарт ISO/IEC 12207:2008 Information Technology – Software Life Cycle Processes є основним нормативним документом, який регламентує склад процесів життєвого циклу ПЗ.

Цей стандарт, використовуючи усталену термінологію, установлює загальну структуру процесів життєвого циклу програмних засобів, на яку можна орієнтуватися у програмній індустрії. Стандарт визначає процеси, види діяльності та завдання, які використовують під час придбання програмного продукту або послуги, а також постачання, розроблення, застосування за призначенням, супроводу та припинення застосування програмних продуктів.

Кожен процес розподілено на набір дій, кожна дія – на набір завдань. Кожен процес, дія або завдання ініціює та виконує інший процес у міру необхідності, причому не має заздалегідь визначених послідовностей виконання. Зв'язки за вхідними даними водночас зберігаються.

Кожен процес містить ряд дій. Наприклад, *процес придбання* охоплює такі дії:

- ініціацію придбання;
- підготовку заявкових пропозицій;
- підготовку й коригування договору;
- нагляд за діяльністю постачальника;
- приймання та завершення робіт.

Кожна дія містить ряд завдань. Наприклад, *підготовка заявкових пропозицій* має передбачати такі дії:

- формування вимог до системи;
- формування списку програмного продукту;

- установлення умов і угод;
- опис технічних обмежень (середовище функціонування системи та ін.).

Процеси життєвого циклу ПЗ розподіляють на основні, допоміжні та організаційні.

До *основних* належать такі:

Придбання – дії й завдання замовника, що набуває ПЗ.

Постачання – дії й завдання постачальника, що забезпечує замовника програмним продуктом або послугою.

Розроблення – дії й завдання, що виконує розробник: створення ПЗ, оформлення проєктної та експлуатаційної документації, підготовка тестових навчальних матеріалів та ін.

Експлуатація – дії й завдання оператора – організації, що експлуатує систему.

Супровід – дії й завдання, що виконує організація, яка займається супроводженням, тобто службою впровадження, унесення змін у ПЗ, із метою виправлення помилок, підвищення продуктивності чи адаптації до умов роботи, що змінилися, або вимог.

До *допоміжних* належать такі:

Документування – формалізований опис інформації, створеної протягом ЖЦ ПЗ.

Управління конфігурацією – використання адміністративних і технічних процедур протягом усього ЖЦ ПЗ для визначення стану компонентів ПЗ, управління його модифікаціями.

Забезпечення якості – забезпечення гарантій того, що ІС і процеси його ЖЦ відповідають заданим вимогам і затвердженим планам.

Верифікація – визначення того, що програмні продукти, які є результатами певної дії, повністю задовольняють вимоги або умови, обумовлені попередніми діями.

Атестація – визначення повноти відповідності заданих вимог і створеної системи їхньому конкретному функціональному призначенню.

Загальна оцінка – оцінювання стану робіт по проєкту, контроль за плануванням та управлінням ресурсами, персоналом, апаратурою, інструментальними засобами.

Аудит – визначення відповідності вимогам, планам та умовам договору.

Розв'язання проблем – аналіз і розв'язання проблем, незалежно від їхнього походження або джерела, виявлені під час розроблення, експлуатації, супроводу або інших процесів.

Організаційні процеси містять:

Управління – дії й завдання, які може виконувати будь-яка сторона, яка управляє своїми процесами.

Створення інфраструктури – вибір і супровід технології, стандартів та інструментальних засобів, вибір і встановлення апаратних та програмних засобів, які використовують для розроблення, експлуатації або супроводу ПЗ.

Удосконалення – оцінювання, вимірювання, удосконалення процесів ЖЦ і контроль за ними.

Навчання – первинне навчання та подальше постійне підвищення кваліфікації персоналу.

Модель життєвого циклу ПЗ – структура, що визначає послідовність виконання та взаємозв'язку процесів, дій і завдань протягом життєвого циклу. Модель життєвого циклу залежить від специфіки, масштабу та складності проєкту і специфіки умов, у яких система створюється та функціонує. На кожній стадії можна виконувати як кілька процесів, так і навпаки, один і той самий процес можна виконувати на різних стадіях. Співвідношення між процесами та стадіями також визначено використовуваною моделлю життєвого циклу ПЗ.

За структурою розрізняють такі моделі життєвого циклу ПЗ:

- каскадну;
- спіральну;
- ітераційну.

Каскадна модель передбачає послідовне виконання всіх етапів проєкту в точно фіксованому порядку. Перехід на наступний етап означає повне завершення робіт на попередньому етапі. Вимоги, визначені на стадії формування вимог, точно документують у вигляді технічного завдання та фіксують на весь час розроблення проєкту. Кожна стадія завершується випуском повного комплекту документації, достатньої для того, щоб розроблення могло бути продовженим іншою командою розробників.

Етапи проєкту, відповідно до каскадної моделі такі:

- формування вимог;
- проєктування;
- реалізація;
- тестування;
- упровадження;
- експлуатація та супровід.

Спіральна модель – ПЗ створюють у кілька ітерацій (витків спіралі) методом прототипування. Кожна ітерація відповідає створенню фрагмента або версії ПЗ, у ньому уточнюють цілі й характеристики проєкту, оцінюють якість визначених результатів та планують роботи наступної ітерації.

На кожній ітерації оцінюють:

- ризик перевищення строків і вартості проєкту;
- необхідність у виконанні ще однієї ітерації;
- міру повноти й точності розуміння вимог до системи;
- доцільності припинення проєкту.

Один із прикладів реалізації спіральної моделі – RAD (від англ. *Rapid Application Development* – метод швидкого розроблення застосунків).

Ітераційна модель – становить раціональне поєднання моделей, описаних раніше. Різні варіанти ітераційного підходу реалізовано в більшості сучасних технологій і методів (RUP, MSF, XP). Особливостями моделі процесів MSF є такі:

- підхід, заснований на фазах і віхах;
- ітеративний підхід;
- інтегрований підхід до створення та запровадження рішень;

Модель процесів містить такі основні фази процесу розроблення:

- вироблення концепції (envisioning);
- планування (planning);
- розроблення (developing);
- стабілізацію (stabilizing);
- упровадження (deploying).

Крім цього є велика кількість проміжних етапів, які показують досягнення в ході проєкту певного прогресу та розчленовують великі сегменти роботи на менші, досяжні ділянки.

У межах MSF програмний код, документація, дизайн, плани та інші робочі матеріали створюють, зазвичай, ітеративними методами. MSF рекомендує починати розроблення розв'язання з побудови, тестування та впровадження його базової функціональності. Потім до рішення додають усе нові й нові можливості. Таку стратегію називають стратегією *версіонування*. Незважаючи на те що для малих проєктів може бути достатнім випуск однієї версії, рекомендують не втрачати можливості створення на одне рішення ряду версій. Зі створенням нових версій еволюціонує функціональність рішення.

Ітеративний підхід до процесу розроблення потребує використання гнучкого способу ведення документації. "Живі" документи (living documents) мають змінювати в міру еволюції проєкту разом зі змінами вимог до кінцевого продукту.

Контрольні запитання

1. Дайте визначення поняття інформаційної системи.
2. Що становить банк даних і які компоненти входять до його складу?
3. Яке призначення СУБД?
4. Назвіть основні моделі даних.
5. Дайте визначення застосунку, укажіть, у яких випадках його розробляють.
6. Укажіть призначення словника даних.
7. Перелічіть функції адміністратора бази даних.
8. Що становить обчислювальна система?
9. Охарактеризуйте архітектуру "клієнт – сервер" і назвіть варіанти її реалізації, укажіть переваги й недоліки.
10. Зобразіть структуру інформаційної системи з файл-сервером.
11. Зобразіть структуру інформаційної системи із сервером баз даних.
12. Охарактеризуйте основні види програм, що належать до СУБД.
13. Назвіть основні способи роботи користувача з базою даних під час розв'язання прикладних задач.
14. Укажіть технології створення застосунків роботи з базами даних.
15. Охарактеризуйте способи виконання застосунків роботи з базами даних.
16. Розкрийте зміст понять "етапи життєвого циклу" та "стадії розроблення".
17. Розкрийте зміст поняття "архітектура інформаційної системи".
18. Наведіть основні класифікаційні ознаки інформаційних систем із погляду поширення контенту та особливостей життєвого циклу.
19. Розкрийте зміст поняття "життєвий цикл інформаційної системи".
20. Наведіть відмінні риси процесів, фаз та етапів життєвого циклу за IEC і MSF.
21. Охарактеризуйте роль об'єктного аналізу під час проєктування баз даних.

Рекомендована література: [1; 3; 5].

2. Моделі й типи даних

Мета – визначення понять "модель", "типи даних" та їхніх основних складових. Розуміння класичних і сучасних моделей подання даних, принципів інкапсуляції, поліморфізму та наслідування щодо об'єктно-орієнтованих баз даних.

Основні питання

2.1. Ієрархічна модель.
2.2. Мережева модель.
2.3. Реляційна модель.
2.4. Постреляційна модель.
2.5. Багатовимірна модель.
2.6. Об'єктно-орієнтована модель.
2.7. Типи даних.

Ключові слова: модель, тип даних, інкапсуляція, поліморфізм, наслідування, об'єктно-орієнтовані бази даних.

Збережені в базі дані мають певну логічну структуру, інакше кажучи, описують деякою моделлю подання даних, підтримуваної СУБД. До класичних належать такі моделі даних:

- ієрархічна;
- мережева;
- реляційна;
- постреляційна;
- багатовимірна;
- об'єктно-орієнтована;
- концептуальна.

Розробляють також усілякі системи, засновані на інших моделях даних, які розширюють відомі моделі. Серед них можна назвати об'єктно-реляційні, дедуктивно-об'єктно-орієнтовані, семантичні та інші моделі. Деякі із цих моделей слугують для інтеграції баз даних, баз знань і мов програмування.

2.1. Ієрархічна модель

В ієрархічній моделі зв'язок між даними можна описати за допомогою упорядкованого графа (або дерева). 1968 року було введено в експлуатацію першу промислову СУБД система IMS фірми IBM (рис. 2.1). Це була перша ієрархічна база даних, завдяки якій визначили ряд фундаментальних понять у теорії систем баз даних, які й досі є основними для мережевої моделі даних.

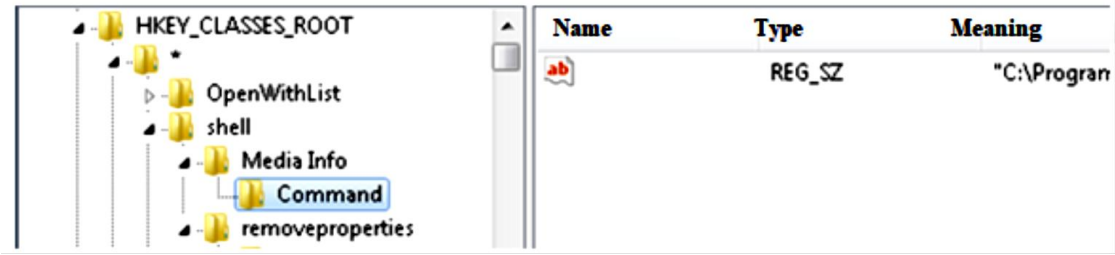


Рис. 2.1. **Деревоподібна структура файлової системи**

Таку форму використовують:

- сервери каталогів, як-от LDAP і Active Directory (допускають чітке уявлення у вигляді дерева);
- файлові, база налаштувань Windows WMI і реєстр Windows;
- Google App Engine Datastore API.

Ієрархічні моделі мають деревоподібну структуру, де кожному вузлові відповідає один сегмент, який становить названий лінійний кортеж полів даних. Кожному сегментові відповідає один вхідний і кілька вихідних сегментів. Кожен елемент структури лежить на єдиному ієрархічному шляху, що починається від кореневого. Ієрархічні бази даних є найбільш придатними для моделювання структур, які за своєю природою є ієрархічними. Як приклади, можна навести військові підрозділи або складні механізми, що складаються з більш простих вузлів, які, своєю чергою, теж можна піддати декомпозиції. Проте є значна кількість організацій, які не зведено до простої ієрархії. У цій моделі запит, спрямований униз по ієрархії, є простим, однак запит, спрямований угору по ієрархії, є більш складним. До переваг ієрархічної моделі даних належать ефективне використання пам'яті ЕОМ і якісні показники часу виконання основних операцій над даними. Ієрархічної базою даних є файлова система, що складається з кореневої директорії, у якій є деревоподібна структура піддиректорій і файлів.

Ієрархічні бази даних – найраніші моделі подання складної структури даних. Інформацію в ієрархічній базі організовано за принципом деревовидної структури, у вигляді відносин "предок – нащадок". Кожен запис може мати не більше ніж однин батьківський запис і кілька підлеглих. Зв'язки записів реалізують у вигляді фізичних покажчиків з одного запису на інший. Основний недолік ієрархічної структури бази даних – неможливість реалізувати відносини "багато-до-багатьох", а також ситуації, якщо запис має кілька предків.

Графічно таку структуру можна зобразити у вигляді дерева, що складається з об'єктів різних рівнів. Верхній рівень займає один об'єкт, другий – об'єкти другого рівня тощо.

Між об'єктами наявні зв'язки, кожен об'єкт може містити кілька об'єктів нижчого рівня. Такі об'єкти будуть у відношенні предка (об'єкт є більш близьким до кореня) до нащадка (об'єкт більш низького рівня), водночас можливо, щоб об'єкт-предок не мав нащадків або мав їх кілька, тоді як в об'єкта-нащадка обов'язково був тільки один предок. Об'єкти, що мають загального предка, називають *близнюками*.

Для організації фізичного розміщення ієрархічних даних у пам'яті ЕОМ можна використовувати такі групи методів:

- подання лінійним списком із послідовним розподілом пам'яті (адресна арифметика, лівоспискові структури);
- подання зв'язаними лінійними списками (методи, що використовують покажчики й довідники).

До основних операцій маніпулювання ієрархічно організованими даними належать такі:

- пошук зазначеного примірника БД;
- перехід від одного дерева до іншого;
- перехід від одного запису до іншого всередині дерева;
- уставлення нового запису в зазначену позицію;
- видалення поточного запису тощо.

Відповідно до визначення типу "дерево", між предками й нащадками автоматично постачається контроль за цілісністю зв'язків. Основне правило контролю за цілісністю формулюють у такий спосіб: нащадок не може бути без батька, а в деяких батьків може не бути нащадків. Ієрархічна модель даних є зручною для роботи з ієрархічно впорядкованою інформацією.

Недоліком ієрархічної моделі є її громіздкість для опрацювання інформації з досить складними логічними зв'язками, а також складність розуміння для звичайного користувача.

На ієрархічній моделі даних засновано порівняно обмежену кількість СУБД, серед яких можна назвати системи IMS, PC/Focus, Team-Up і Data Edge.

2.2. Мережева модель

Мережева модель даних є більш загальною структурою, порівняно з ієрархічною. Основні принципи мережевої моделі даних було розроблено в середині 1960-х рр., еталонний варіант мережевої моделі даних описано у звітах робочої групи з мов баз даних (COnference on DAta SYstem Languages) CODASYL (1971).

У мережевих БД разом із вертикальними реалізовано й горизонтальні зв'язки. Кожен окремий сегмент (осередок) може мати довільну кількість безпосередніх вихідних (старших) сегментів, а також довільну кількість породжених (молодших) (рис. 2.2). Це забезпечує подання відношення "багато-до-багатьох".

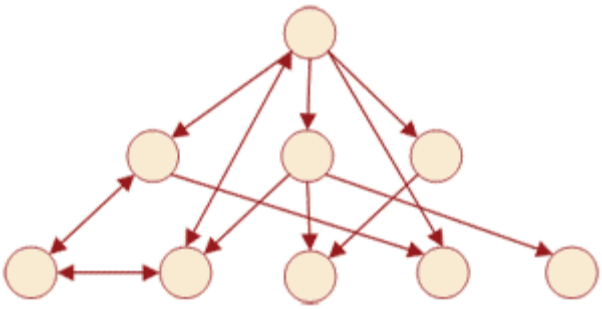


Рис. 2.2. **Мережева модель даних**

У мережевій моделі успадковано багато недоліків ієрархічної, і головний із них – необхідність чітко визначати на фізичному рівні зв'язки даних і настільки ж чітко дотримуватися цієї структури зв'язків у разі запитів до бази. Оскільки логіка процедури вибірки даних залежить від їхньої фізичної організації, то мережева модель не є повністю незалежною від програми. Інакше кажучи, якщо необхідно змінити структуру даних, то потрібно поміняти й застосунок.

Основна відмінність цих моделей полягає в тому, що в мережевій моделі запис може бути членом більш ніж одного групового відношення. Відповідно до цієї моделі, кожне групове відношення називають і відрізняють його тип та екземпляр. Тип групового відношення задано його назвою і визначає загальні властивості для всіх екземплярів цього типу. Примірник групового зв'язку є записом-власником і множиною (можливо порожньою) підлеглих записів. Водночас є таке обмеження: екземпляр запису не може бути членом двох примірників групових відношень одного типу (тобто співробітник, наприклад, не може працювати у двох відділах).

Для опису схеми мережевої БД використовують дві групи типів: "записи" та "зв'язок". Мережева БД складається з набору записів і набору відповідних зв'язків. На формування зв'язку особливих обмежень не накладають. Якщо в ієрархічних структурах запис-нащадок міг мати тільки один запис-предка, то в мережевий моделі даних запис-нащадок може мати довільну кількість записів-предків.

Фізичне розміщення даних у базах мережевого типу може бути організовано практично тими самими методами, що й в ієрархічних базах даних.

До найважливіших операцій маніпулювання даними баз мережевого типу можна зарахувати такі:

- пошук запису у БД;
- перехід від предка до першого нащадка;
- перехід від нащадка до предка;
- створення нового запису;
- видалення поточного запису;
- оновлення поточного запису;
- установлення запису у зв'язок;
- вилучення запису із зв'язку;
- зміна зв'язків тощо.

Перевагою мережевої моделі даних є можливість ефективної реалізації за показниками витрат пам'яті й оперативності. Порівняно з ієрархічною моделлю, мережева модель надає великі можливості в сенсі допустимості вибору довільних зв'язків.

Недоліком мережевої моделі даних є висока складність і жорсткість схеми БД, побудованої на її основі, а також складність для розуміння та виконання опрацювання інформації у БД звичайним користувачем. Крім того, у мережевий моделі даних ослаблено контроль за цілісністю

зв'язків, унаслідок допустимості встановлення довільних зв'язків між записами.

Системи на основі мережевої моделі не набули значного поширення на практиці. Найбільш відомими мережевими СУБД є IDMS, db_VistaIII.

2.3. Реляційна модель

Реляційна модель даних ґрунтується на понятті "відношення" (relation). Реляційна модель даних – це створена Едгаром Франком Коддом логічна модель даних, що описує:

- структури даних у вигляді (що змінюються в часі) наборів відношень;
- теоретико-множинні операції над даними: об'єднання, перетинання, віднімання та декартове множення;
- спеціальні реляційні операції: селекція, проєкція, з'єднання та розподіл;
- спеціальні правила, що забезпечують цілісність даних.

Едгар Франк Кодд – (23 серпня 1923 – 18 квітня 2003 року) – британський учений, роботи якого заклали основи теорії реляційних баз даних. Працюючи в компанії IBM, він створив реляційну модель даних. 1970 року видав працю A Relational Model of Data for Large Shared Data Banks, яку вважають першою з реляційної моделі даних.

Реляційна модель даних – це спосіб розгляду даних, тобто припис для способу подання даних (за допомогою таблиць) і роботи з таким поданням (за допомогою операторів). Її пов'язано із трьома аспектами даних: структурою (об'єктами), цілісністю й опрацюванням даних (операторами).

2002 року журнал Forbes помістив реляційну модель даних у список найважливіших інновацій останніх 85 років.

Цілі створення реляційної моделі даних такі:

- забезпечення більш високого ступеня незалежності від даних;
- створення міцного фундаменту для вирішення семантичних питань і розв'язання проблем несуперечності й надмірності даних;
- розширення мов управління даними шляхом уміщення операцій над множинами.

Реляційна модель даних передбачає структуру даних, обов'язковими об'єктами якої є такі:

- відношення;
- атрибут;

• домен;

• кортеж;

• ступінь;

• кардинальність;

• первинний ключ.

Відношення – це плоска (двовимірна) таблиця, що складається зі стовпців і рядків.

Атрибут – це названий стовпець відносини.

Домен – це набір допустимих значень для одного або декількох атрибутів.

Кортеж – це рядок відношення.

Ступінь – це кількість атрибутів, що він містить.

Кардинальність – це кількість кортежів, яке містить відношення.

Первинний ключ – це унікальний ідентифікатор для таблиці.

Відношення і їхня реалізація в реляційної моделі даних

Відношення *R* на множині доменів D_1, D_2, ..., D_n – це підмножина декартового множення цих доменів:

$$R \subseteq D_1 \times D_2 \times ... \times D_n.$$

Приклад. Визначено домени: D_1 – множина прізвищ викладачів; D_2 – множина аудиторій; D_3 – множина навчальних груп; D_4 – множина навчальних дисциплін. Запишіть відношення: 1) закріплення викладачів за навчальними курсами; 2) розклад занять у групах.

Вирішення.

1) закріплення викладачів за навчальними курсами:

$$R \subseteq D_1 \times D_4.$$

Це відношення визначає множину викладачів, які ведуть множину навчальних дисциплін;

2) розклад занять у групах:

$$R \subseteq D_2 \times D_3 \times D_4.$$

Це відношення визначає множину аудиторій, у яких проводять заняття із множини навчальних дисциплін для множини навчальних груп.

Властивості відношень:

- унікальна назва відношення;
- унікальна назва атрибута;
- немає однакових кортежів;
- кортежі не є впорядкованими згори вниз;
- атрибути не є впорядкованими зліва направо;
- усі значення атрибутів є атомарними (нормалізоване відношення).

Отже, реляційна база даних – це набір нормалізованих відношень. Для того щоб перейти до видів відношень, уведімо поняття змінного відношення. Мінливе відношення – це названий об'єкт, значення якого може змінюватися із плином часу. Мінливе відношення в різний час – це різні таблиці бази даних, у яких різні рядки, але однакові стовпці.

Види відношення:

- назване відношення;
- базове відношення;
- похідне відношення;
- відношення;
- подання (view);
- знімання (snapshot);
- результат запиту;
- проміжний результат.

Назване відношення – це змінне відношення, визначене в СУБД (системи управління базами даних) за допомогою оператора **CREATE** (CREATE TABLE, CREATE BASE RELATION, CREATE VIEW, CREATE SNAPSHOT).

Базове відношення – це назване відношення, яке не є похідним. Наявність базового відношення не залежить від наявності інших відношень.

Похідне відношення – це відношення, визначене через інші названі відношення. Похідне відношення залежить від наявності інших відношень.

Відношення – це відношення, яке можна дістати з набору названих відношень за допомогою деякого реляційного виявлення. Кожне назване відношення є висловлюваним відношенням, але не навпаки.

Подання – це назване похідне відношення. Подане в базі даних у вигляді визначення. Подання не зберігають у фізичній пам'яті системи управ-

ління базою даних (СКБД), а формують із використанням інших названих відношень.

Знімання (snapshot) – це те саме, що й подання, але з фізичним збереженням і періодичним оновленням.

Результат запиту – це неназване похідне відношення. СУБД не забезпечує постійної наявності результатів запитів. Для збереження результату запиту його можна надати будь-якому названому відношенню.

Проміжний результат – це відношення, яке є результатом підзапиту, укладеного в більший вираз.

Ключі відношення можуть бути такими, як-от:

- суперключ;
- потенційний ключ;
- первинний ключ;
- зовнішній ключ;
- сурогатний ключ.

Ключ відношення – це підсхема вихідної схеми відношення, що складається з одного або декількох атрибутів, для яких декларують умову унікальності значень у кортежі відношень. У разі оголошення схеми базового відношення може бути задано оголошення декількох ключів.

Ключ відношення може бути простим або складеним. Простий ключ – це ключ, що складається не більше ніж одного атрибута. Складений ключ – ключ, що складається із двох і більше атрибутів.

Суперключ – це атрибут або множина атрибутів, які єдиним чином ідентифікують кортеж цього відношення. Він може містити додаткові атрибути. Суперключ не має властивості ненадлишковості.

Потенційний ключ – це підмножина атрибутів відношення, що задовольняє вимоги унікальності й ненадлишковості. Він має такі властивості, як-от:

- унікальність – у таблиці немає двох різних рядків з однаковими значеннями в нашому потенційному ключі;
- ненадмірність – не можна прибрати один зі стовпців із ключа так, щоб він не втратив унікальності.

У відношенні може бути більше ніж один потенційний ключ.

Первинний ключ (primary key, PK) – це один із потенційних ключів відношення, вибраний як основний ключ. Припустімо оголошення одного й тільки одного первинного ключа. Атрибути первинного ключа не можуть набувати значень Null.

Зовнішній ключ (*foreign key, FK*) – це ключ, оголошений у базовому відношенні, який водночас посилається на первинний того самого або якогось іншого базового відношення.

Сурогатний ключ – це службовий атрибут, доданий до вже наявних інформаційних атрибутів відношень. Призначення сурогатного ключа – слугувати первинним ключем відношення. Значення цього атрибута генерують штучно.

Приклад. Є база даних мережі аптек. У ній є таблиця "Аптеки", у яку занесено всі аптеки мережі, і є таблиця "Препарати". Крім того, є таблиця "Наявність", у яку заносять дані про наявність препаратів у кожній аптеці. У таблиці "Наявність є такі поля: "Аптека" (у ній – ідентифікатори аптек), "Препарат" (у ній – ідентифікатори препаратів), "Кількість". Виникає проблема: у разі надходження в аптеку деякої кількості препарату можна не помітити, що в тій самій аптеці той самий препарат уже міститься в деякій кількості та зробити новий запис у таблиці, у якій аптека та препарат буде повторено. Як на рівні ключів уникнути цієї проблеми?

Вирішення. Можна оголосити первинним ключем таблиці "Наявність" складений ключ, що складається з ідентифікатора аптеки й ідентифікатора препарату. Тоді в таблиці неможливе повторення в різних записах поєднання "Аптека" та "Препарати". Первинний ключ може бути не тільки простим, але й складеним.

Цілісність даних у реляційної моделі даних

Поняття реляційної цілісності:

- визначник NULL;
- цілісність сутностей;
- посилальна цілісність;
- корпоративні обмеження цілісності.

Визначник NULL. Значення Null позначає той факт, що значення не є визначеним. Null не належить ніякому типу даних і може бути наявним серед значень будь-якого атрибута, визначеного на будь-якому типі даних. Двомісна арифметична операція з Null дає Null. Операція порівняння з Null дає UNKNOWN.

Цілісність сутностей. Вимога цілісності сутності означає, що первинний ключ має повністю ідентифікувати кожну сутність, а тому у складі будь-якого значення первинного ключа не допускають наявності невизначених значень. Значення атрибута має бути атомарним.

Посилальна цілісність. Вимога цілісності полягає в тому, що для кожного значення зовнішнього ключа, що з'являється в кортежі значення-відношення, має бути кортеж із таким самим значенням первинного ключа, або значення зовнішнього ключа має бути повністю не визначеним. Є правила видалення кортежу з відношення, на яке веде посилання.

Є три підходи видалення кортежу з відношення, на яке веде посилання:

1. Обмеження видалення – Delete: Restrict.
2. Каскадне видалення – Delete: Cascade.
3. Установлення значення NULL, переведення значення зовнішнього ключа в невизначений стан – Delete: Set NULL.

Обмеження видалення. Заборонено робити видалення кортежу, для якого наявні посилання. Спочатку потрібно або видалити ці кортежі, або відповідним чином змінити значення їхнього зовнішнього ключа.

Каскадне видалення. Під час видалення кортежу з відношення, на яке веде посилання, із відношення автоматично видаляються всі ці кортежі.

Установлення значення NULL. Під час видалення кортежу, на який є посилання, у всіх кортежів значення зовнішнього ключа автоматично стає повністю невизначеним.

Приклад. Є база даних порталу новин. У ній є таблиця "Рубрики" (політика, економіка, спорт тощо), таблиця "Автор" (прізвища та імена авторів). Є таблиця "Тексти", у якій в кожному записі про текст новини є поля "Рубрика" (з ідентифікаторами рубрик із відповідної таблиці) й "Автор" (з ідентифікаторами рубрик із відповідної таблиці). Якими способами можна домогтися, щоб під час видалення рубрики й автора було дотримано посилальної цілісності даних?

Вирішення.

Перший спосіб: установити заборону на видалення рубрики або автора з відповідних таблиць, у разі, якщо в таблиці "Тексти" є посилання на цю рубрику або на цього автора.

Другий спосіб: поставити автоматичне видалення з таблиці "Тексти" записів, у яких фігурують ця рубрика або цей автор.

Третій спосіб: у разі видалення рубрики або автора з відповідних таблиць кортежу таблиці "Тексти" значення ідентифікатора цієї рубрики або цього автора стають невизначеними (NULL).

Корпоративні обмеження цілісності – це додаткові правила підтримання цілісності даних, що визначають користувачі або адміністратори бази даних.

Перевага реляційної моделі даних полягає у простоті, зрозумілості та зручності фізичної реалізації на ЕОМ. Саме простота та зрозумілість для користувача стали основною причиною їхнього широкого використання.

Основними недоліками реляційної моделі є відсутність стандартних засобів ідентифікації окремих записів і складність опису ієрархічних та мережевих зв'язків.

Прикладами реляційних СУБД є такі: dBase III Plus і dBase IY (фірма Ashton-Tate); DB2 (IBM); R:BASE (Microrim); FoxPro і FoxBase (Fox Software); Paradox і dBASE for Windows (Borland); Visual FoxPro і Access (Microsoft); Clarion (Clarion Software); Ingres (ASK Computer Systems) і Oracle (Oracle).

2.4. Постреляційна модель

Класична реляційна модель передбачає неподільність даних, що зберігають у полях записів таблиць. Це означає, що інформацію в таблиці видають у першій нормальній формі. Є ряд випадків, коли це обмеження заважає ефективній реалізації застосунків.

Одним із найбільш важливих принципів реляційної моделі даних є нормалізація. Однак використання першої нормальної форми (1НФ) накладає обмеження атомарності на допустимі значення атрибутів (усі використовувані домени відношення мають містити тільки скалярні значення). Це знижує виразність реляційної моделі даних під час опису цілого ряду предметних галузей. Найбільш часто така проблема виникає, якщо атрибут має містити множинні значення або потрібна внутрішня структура даних атрибута.

Отже, у ряді випадків нормалізована реляційна модель предметної галузі є штучною. Крім того, нормалізація в таких випадках змушує розробляти досить складні запити для визначення, здавалося б, простих даних.

У ряді випадків можливі значення атрибута мають внутрішню структуру. Наприклад, дата народження, номер навчальної групи студента. Правила нормалізації реляційної моделі даних потребують декомпозиції такого складного атрибута на кілька простих (атомарних). Це не завжди зручно й наочно.

Для подолання такого роду недоліків реляційної моделі даних було розроблено постреляційну (post-relational) модель даних. Створення такої моделі даних, що допускає неатомарність значень атрибутів кортежів,

викликало розроблення нових правил нормалізації. Основою нормалізації в постреляційної моделі даних слугує так звана "не перша нормальна форма" – Non First Normal Form (NF 2). Сенс полягає в розширеному трактуванні поняття "атрибут". У постреляційній моделі атрибут може бути або атомарним (як у реляційній моделі), або множинним. Множинний атрибут описано вкладеним відношенням (множиною кортежів) з усіма наслідками, що випливають. Узагалі атрибути такого вкладеного відношення також можуть бути множинними. Це допущення не порушує принципів реляційної алгебри.

Для маніпулювання структурою й даними в постреляційних СУБД виробники створюють розширення мови SQL. Стандарту такого розширення немає, а в кожної постреляційної СУБД використовують свій синтаксис. Однак у будь-якому разі множинні атрибути подано або як вкладені таблиці, або як масиви даних (одновимірні або багатовимірні).

На рис. 2.3 на прикладі інформації про накладні й товари для порівняння показано подання одних і тих самих даних за допомогою реляційної (а) і постреляційної (б) моделей.

Накладні

Номери накладних	Номери покупців
0373	8723
8374	8232
7364	8723

Накладні-товари

Номери накладних	Назва товару	Кількість
0373	Сир	3
0373	Риба	2
8374	Лимонад	1
8374	Сік	6
8374	Печиво	2
7364	Йогурт	1

а)

Накладні

Номери накладних	Номери покупців	Назва товару	Кількість
0373	8723	Сир	3
		Риба	2
8374	8232	Лимонад	1
		Сік	6
		Печиво	2
7364	8723	Йогурт	1

б)

Рис. 2.3. **Структура даних реляційної (а) і постреляційної (б) моделей**

Таблиця "Накладні" містить дані про номери накладних і номери покупців. У таблиці "Накладні-товари" містяться дані про кожну з накладних: номер накладної, назва товару та кількість товару. Таблицю "Накладні" пов'язано з таблицею "Накладні-товари" за полем "Номери накладних".

Як видно з рис. 2.3, порівняно з реляційною моделлю, у постреляційній моделі дані зберігають більш ефективно, а під час опрацювання не потрібно виконувати операцію поєднання даних із двох таблиць.

На довжину та кількість полів у записах таблиці не накладено вимогу сталості. Це означає, що структура даних і таблиць мають велику гнучкість.

Перевагою постреляційної моделі є можливість подання сукупності пов'язаних реляційних таблиць однією постреляційною таблицею. Це забезпечує високу наочність подання інформації та підвищення ефективності її опрацювання.

Недоліком постреляційної моделі є складність розв'язання проблеми забезпечення цілісності й несуперечності даних, що зберігають.

Постреляційну модель даних підтримує СУБД uniVers, а також системи Bubba та Dasdb.

2.5. Багатовимірна модель

Джерелом для багатовимірних баз даних стали не сучасні технології баз даних, а алгебра багатовимірних матриць, яку використовували для ручного аналізу даних із кінця XIX ст.

У кінці 1960-х рр. компанії IRI Software і Comshare, незалежно одна від одної, почали розробляти те, що пізніше стали називати системами управління багатовимірними базами даних. IRI Express, популярний у кінці 1970-х і початку 1980-х рр. інструментарій маркетингового аналізу, став лідером ринку засобів оперативного аналітичного опрацювання й був придбаний корпорацією Oracle. На базі системи Comshare було створено інструментарій System W, у 1980-х рр. його активно застосовували для фінансового планування, аналізу та формування звітів.

Утворена 1991 року компанія Arbor Software (тепер Hyperion Solutions) як свою спеціалізацію вибрала створення багатокористувацьких серверів багатовимірних баз даних. Результатом цих робіт стала система Essbase. Пізніше Arbor ліцензувала базову версію Essbase корпорації IBM, яка інтегрувала її у DB2.

1993 року Е. Ф. Кодд увів термін OLAP. На початку 1990-х рр. склалася ще одна важлива концепція – великі сховища даних, які, зазвичай, ґрунтуються на схемах "зірка" та "сніжинка". За такого підходу для реалізації багатовимірних баз вдається використовувати технологію реляційних баз даних.

1998 року корпорація Microsoft випустила OLAP Server – першу багатовимірну систему, призначену для масового ринку, і тепер такі системи стають поширеними продуктами та їх пропонують без додаткової оплати разом із популярними системами управління базами даних.

Багатовимірні СУБД є вузькоспеціалізованими СУБД, призначеними для інтерактивного аналітичного опрацювання інформації. Розкриймо основні поняття, що використовують у цих СУБД: агрегованість, історичність, статичність і прогнозованість даних.

Агрегованість даних означає розгляд інформації на різних рівнях її узагальнення. В інформаційних системах ступінь детальності подання інформації для користувача залежить від його рівня: аналітик, користувач-оператор, керівник.

Історичність даних передбачає забезпечення високого рівня статичності (незмінності) власне даних і їхніх взаємозв'язків, а також обов'язковість прив'язування даних до часу.

Статичність даних дозволяє використовувати під час їхнього опрацювання спеціалізовані методи завантаження, зберігання, індексації та вибірки.

Тимчасове прив'язування даних необхідно для частого виконання запитів, що мають значення часу й дати у складі вибірки. Необхідність у впорядкуванні даних за часом у процесі опрацювання й подання даних користувачеві накладає вимоги на механізми зберігання та доступу до інформації. Так, для зменшення часу опрацювання запитів бажано, щоб дані завжди були розсортованими в тому порядку, у якому їх найбільш часто запитують.

Прогнозованість даних має на увазі завдання функцій прогнозування й застосування їх до різних тимчасових інтервалів.

Багатовимірність моделі даних означає не багатовимірність візуалізації цифрових даних, а багатовимірне логічне подання структури інформації під час опису та в операціях маніпулювання даними.

Порівняно з реляційною моделлю, багатовимірна організація даних має більш високу наочність та інформативність. Для ілюстрації на рис. 2.4

показано реляційне (а) й багатовимірне (б) подання одних і тих самих даних про обсяги продажів автомобілів.

Моделі	Місяці	Обсяги
Opel	червень	12
Opel	липень	24
Opel	серпень	5
Audi	червень	2
Audi	липень	18
Ford	липень	19

а

Моделі	червень	липень	серпень
Opel	12	24	5
Audi	2	18	–
Ford	–	19	–

б

Рис. 2.4. **Реляційне (а) і багатовимірне (б) подання даних**

Якщо мова йде про багатовимірні моделі з вимірністю більше ніж два, то не обов'язково інформацію можна надавати у вигляді багатовимірних об'єктів (трьох, чотирьох і більше вимірних гіперкубів). Користувачеві і в цих випадках більш зручно мати справу із двовимірними таблицями або графіками. Дані водночас становлять зрізи з багатовимірного сховища даних, виконані з різним ступенем деталізації.

Розгляньмо основні поняття багатовимірних моделей даних.

В основі багатовимірних сховищ даних лежить багатовимірна модель даних, яка спирається на концепцію гіперкубів. Такі гіперкуби є впорядкованими багатовимірними масивами.

Багатовимірне подання даних розподіляють на дві групи: *факти* – числові параметри та *вимірювання* – текстові параметри для надання максимально можливого контексту для фактів.

У багатовимірній базі даних параметри подають властивості факту, який користувач хоче вивчити.

Основною таблицею сховища даних є таблиця фактів. Зазвичай, такі таблиці містять відомості про об'єкти або події, сукупність яких будуть надалі аналізувати.

Таблиці вимірювань містять практично незмінні дані.

Вимірювання – упорядкований набір значень, прийнятих конкретним параметром, і відповідає одній із граней гіперкуба. Вимірювання якісно

описують досліджуваний процес, вони можуть бути числовими, але в будь-якому разі – це дані є дискретними, тобто їхнє значення належать обмеженому набору.

Вимірювання використовують для вибору й агрегування даних на необхідному рівні деталізації; організовують в ієрархію, що складається з декількох рівнів, кожен із яких подає рівень деталізації, необхідний для відповідного аналізу.

Факти – це дані, які кількісно описують процес, вони є безперервними, тобто можуть набирати нескінченну кількість значень.

Факти подають *суб'єкт* – якийсь шаблон або подію, які необхідно проаналізувати. У більшості багатовимірних моделей даних факти однозначно визначають комбінацією значень вимірювань. Факт наявний тільки тоді, коли клітинка для конкретної комбінації значень не є порожньою.

Кожен факт має деяку гранулярність, визначену рівнями, із яких створюється їхня комбінація значень вимірювань.

Сховища даних, зазвичай, містять такі три типи фактів:

1. *Події* (*event*) на рівні найбільшої гранулярності, зазвичай, моделюють події реального світу, водночас кожен факт подає певний екземпляр досліджуваного явища. Прикладами можуть слугувати продажі, клацання мишкою на вебсторінці або рух товарів на складі.

2. *Миттєві знімання* (*snapshot*) моделюють стан об'єкта наразі, як-от рівні наявності товарів у магазині або на складі та кількість користувачів вебсайта. Один і той самий екземпляр явища реального світу, наприклад, конкретна банка бобів, може виникати в декількох фактах.

3. *Сукупні миттєві знімання* (*cumulative snapshot*) містять інформацію про діяльність організації за певний відрізок часу. Наприклад, сукупний обсяг продажів за попередній період, включно з поточним місяцем, можна легко порівняти з показниками за відповідні місяці минулого року.

Сховище даних часто містить всі три типи фактів. Одні й ті самі вихідні дані, наприклад, рух товарів на складі, можуть міститися у трьох різних типах кубів: потік товарів на складі, список товарів і потік за рік до поточної дати.

Параметри – це властивості факту.

Параметри складаються із двох компонентів, як-от:

1. Числова характеристика факту, наприклад, ціна або дохід від продажів.

2. Формула, звичайно проста агрегативна функція, скажімо, сума, яка може об'єднувати кілька значень параметрів в одне.

Властивість і формулу вибирають таким способом, щоб подати осмислену величину для всіх комбінацій рівнів агрегування.

Класи параметрів такі:

1. *Адитивні параметри* можна комбінувати в будь-якому вимірюваннях. Наприклад, має сенс підсумувати загальний обсяг продажів для продукту, розташування й часу, оскільки це не викликає накладання серед явищ реального світу, які генерують кожне із цих значень.

2. *Напівадитивні параметри*, які не можна комбінувати в одному або декількох вимірюваннях. Наприклад, підсумовування запасів по різних товарах і складах має сенс, але підсумовування запасів товарів у різний час є безглуздим, оскільки одне й те саме фізичне явище можна враховувати кілька разів.

3. *Неадитивні параметри* не комбінують у будь-якому вимірюванні, зазвичай тому, що вибрана формула не дозволяє об'єднати середні значення низького рівня в середньому значенні більш високого рівня.

Адитивні та неадитивні параметри можуть описувати факти будь-якого роду, тоді як напівадитивні параметри, зазвичай, використовують із миттєвими зніманнями або сукупними миттєвими зніманнями.

Гіперкуб. Гіперкуб (рис. 2.5) розглядають як систему координат, у якому осями є вимірювання. Теоретично, куб може мати будь-яку кількість вимірювань. У такій системі кожному набору значень буде відповідати клітинка, у якій можна розмістити міри (числові показники / факти), пов'язані із цим набором.

Рис. 2.5. **Приклад гіперкуба**

Двовимірне подання куба можна дістати, "розрізавши" його поперек однієї або декількох осей: фіксуймо значення всіх осей, крім двох, і дістаймо звичайну двовимірну таблицю. У горизонтальній осі таблиці (заголовки стовпців) подано одне вимірювання, у вертикальній (заголовки рядків) – інше, а у клітинках таблиці – значення.

На рис. 2.6 зображено двовимірний зріз куба для однієї міри – "продано штук" і двох "нерозрізаних" мір – "країна" та "місяць".

	США	Канада	Мексика
Січень	20 000	4 000	2 000
Лютий	30 000	6 000	3 000
Березень	50 000	10 000	5 000

Рис. 2.6. **Двовимірний зріз куба для однієї міри**

Багатовимірну базу даних природним чином призначено для певних типів запитів.

1. Запити виду slice-and-dice виконують вибір, який скорочує куб (рис. 2.7).

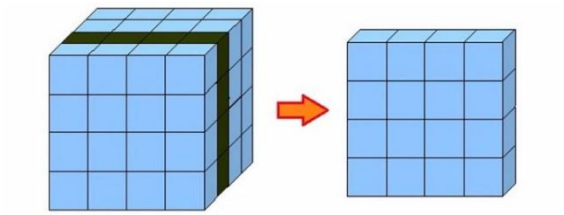


Рис. 2.7. **Зріз куба**

2. Запити виду drill-down і roll-up – взаємодоповнювальні операції, які використовують ієрархію вимірювань і параметри для агрегування. Узагальнення до більших значень відповідає вилученню вимірності (рис. 2.8 і рис. 2.9).

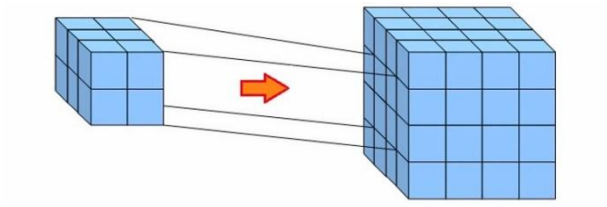


Рис. 2.8. **Деталізація куба**

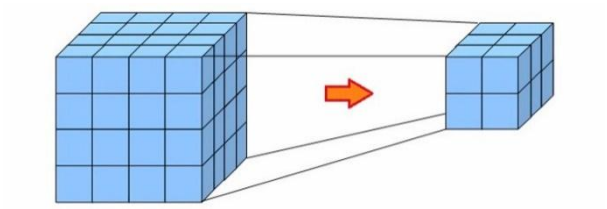


Рис. 2.9. **Консолідація куба (операція, обернена деталізації)**

3. Запити виду drill-across комбінують куби, які мають одне або кілька загальних вимірювань. Із погляду реляційної алгебри така операція виконує злиття (join).

4. Запити виду ranking повертає тільки ті клітинки, які з'являються у верхній або нижній частині упорядкованого певним чином списку.

5. Поворот (*rotating*) куба дає користувачам можливість побачити дані, згруповані за іншими вимірювання.

Багатовимірні бази даних реалізують у двох основних формах:

1. Системи багатовимірної оперативного аналітичного опрацювання (MOLAP) зберігають дані у спеціалізованих багатовимірних структурах. Системи MOLAP, зазвичай, містять засоби для опрацювання розріджених масивів і застосовують вдосконалену індексацію та хешування для пошуку даних під час виконання запитів. Дані в цьому разі організовано у вигляді впорядкованих багатовимірних масивів:

- гіперкубів (усі збережені у БД дані мусять мати однакову вимірність, тобто перебувати в максимально повному базисі);
- полікубів (кожну змінну зберігають із власним набором даних, і всі пов'язані із цим труднощі опрацювання перекладають на внутрішні механізми системи).

2. Реляційні системи OLAP (ROLAP) для зберігання даних використовують реляційні бази даних, а також застосовують спеціалізовані індексні структури, як-от бітові карти, щоб домогтися високої швидкості виконання запитів.

Моделювання багатовимірних кубів на реляційної моделі даних може відбуватися за двома схемами: "зірка" або "сніжинка".

Схема типу "зірка" – схема реляційної бази даних, яка слугує для підтримання багатовимірного подання даних, що містяться в ній. Ця схема бази даних має такі властивості:

- одна таблиця фактів, яка є сильно денормалізованою. Вона є центральною у схемі, може складатися з мільйонів рядків і містить

підсумовувані або фактичні дані, із допомогою яких можна відповісти на різні запитання;

- кілька денормалізованих таблиць вимірювань. Вони мають меншу кількість рядків, ніж таблиці фактів, і містять описову інформацію. Ці таблиці дозволяють користувачеві швидко переходити від таблиці фактів до додаткової інформації;
- таблиця фактів і таблиці вимірності пов'язані зв'язками, що ідентифікують, водночас первинні ключі таблиці вимірності мігрують у таблицю фактів як зовнішніх ключів. Первинний ключ таблиці повністю складається з первинних ключів усіх таблиць вимірності;
- агреговані дані зберігають спільно з вихідними.

Переваги. Завдяки денормалізації таблиць спрощується сприйняття структури даних користувачем і формулювання запитів, зменшується кількість операцій з'єднання таблиць під час опрацювання запитів.

Недоліки. Денормалізація таблиць дає надмірність даних, зростає необхідний для їхнього зберігання обсяг пам'яті.

Схема типу "сніжинка" – схема реляційної бази даних, яка слугує для підтримання багатовимірного подання даних, що містяться в ній, є різновидом схеми типу "зірка". Ця схема бази даних має такі властивості:

- одна таблиця фактів, яка є сильно ненормалізованою, центральною у схемі. Вона може складатися з мільйонів рядків і містити підсумовувані або фактичні дані, із допомогою яких можна відповісти на різні запитання;
- кілька таблиць вимірювань, які є нормалізованими, на відміну від схеми "зірка". Мають меншу кількість рядків, ніж таблиці фактів, і містять описову інформацію. Ці таблиці дозволяють користувачеві швидко переходити від таблиці фактів до додаткової інформації. Первинні ключі в них складаються з єдиного атрибута;
- таблиця фактів і таблиці вимірювань пов'язано зв'язками, водночас первинні ключі таблиці вимірювань мігрують у таблицю фактів як зовнішні ключі. Первинний ключ таблиці факту повністю складається з первинних ключів усіх таблиць розмірності;
- у схемі "сніжинка" агреговані дані можна зберігати окремо від вихідних.

Переваги. Нормалізація таблиць типу "сніжинка", на відміну від схеми "зірка", дозволяє мінімізувати надмірність даних і більш ефективно виконувати запити, пов'язані зі структурою значень.

Недоліки. За нормалізацію таблиць іноді доводиться платити часом виконання запитів.

Умови використання. Використання багатовимірних СУБД виправдано за таких умов:

- обсяг вихідних даних для аналізу є не дуже великим (не більше ніж декілька гігабайт), тобто рівень агрегації даних є досить високим;
- набір інформаційних вимірювань є стабільним;
- час відповіді системи на нерегламентовані запити є найбільш критичним параметром;
- потрібно широке використання складних убудованих функцій для виконання кросвимірних обчислень над клітинками гіперкуба, зокрема можливість написання призначених для користувача функцій.

Основною *перевагою* багатовимірної моделі даних є зручність та ефективність аналітичного опрацювання великих обсягів даних, пов'язаних із часом. У процесі організації опрацювання аналогічних даних на основі реляційної моделі відбувається нелінійне зростання трудомісткості операцій, залежно від вимірності БД і суттєве підвищення витрат оперативної пам'яті на індексацію.

Недоліком багатовимірної моделі даних є її громіздкість для найпростіших завдань звичайного оперативного опрацювання інформації.

Прикладами систем, що підтримують багатовимірні моделі даних, є Essbase (Arbor Software), Media Multi-matrix (Speedware), Oracle Express Server (Oracle) і Cache (InterSystems). Деякі програмні продукти, наприклад, Media/MR (Speedware), дозволяють одночасно працювати з багатовимірними та реляційними БД. У СУБД Cache, у якій внутрішньої моделлю даних є багатовимірна модель, реалізовано три способи доступу до даних: прямий (на рівні вузлів багатовимірних масивів), об'єктний і реляційний.

2.6. Об'єктно-орієнтована модель

В об'єктно-орієнтованій моделі під час подання даних є можливість ідентифікувати окремі записи бази. Між записами бази даних і функціями їхнього опрацювання встановлюють взаємозв'язки за допомогою механізмів, подібних відповідним засобам в об'єктно-орієнтованих мовах програмування.

Стандартизовану об'єктно-орієнтовану модель описано в рекомендаціях стандарту ODMG-93 (Object Database Management Group – група управління об'єктно-орієнтованими базами даних). Реалізувати в повному обсязі рекомендації ODMG-93 поки не вдається. Для ілюстрації ключових ідей розгляньмо спрощену модель об'єктно-орієнтованої БД.

Структуру об'єктно-орієнтованої БД графічно подано у вигляді дерева, вузлами якого є об'єкти. Властивості об'єктів, що описують деяким стандартним типом або типом, сконструйованим користувачем.

Кожен об'єкт-екземпляр класу є нащадком об'єкта, у якому його визначено як властивість. Об'єкт-екземпляр належить своєму класу та має одного батька. Родові відношення у БД утворюють зв'язну ієрархію об'єктів.

Приклад логічної структури об'єктно-орієнтованої БД бібліотечної справи показано на рис. 2.10.

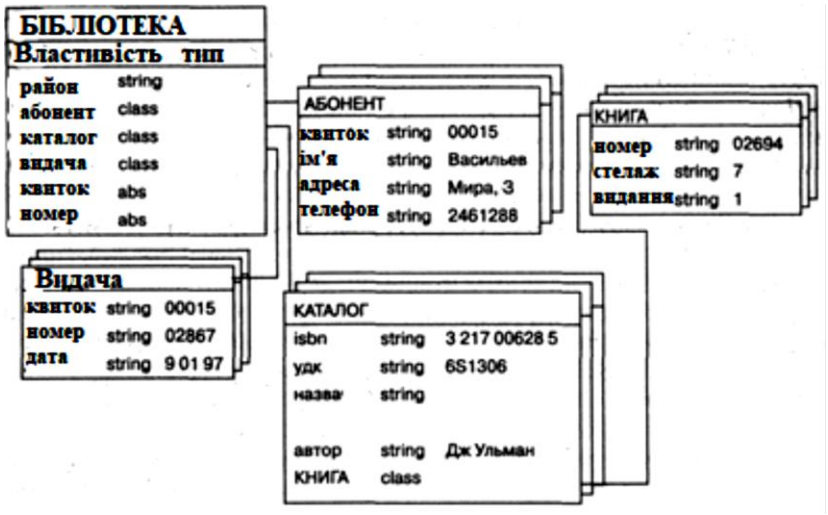


Рис. 2.10. **Логічна структура БД бібліотечної справи**

Тут об'єкт типу БІБЛІОТЕКА є батьківським для позначення об'єктів реалізованих класів АБОНЕНТ, КАТАЛОГ і ВИДАЧА. Різні об'єкти типу КНИГА можуть мати одного або різних батьків. Об'єкти типу КНИГА, які мають одного й того самого прабатька, мають відрізнятися принаймні інвентарним номером (унікальним для кожного екземпляра книга), але мають одинакові значення властивостей ISBN, УДК, назву й автора. Логічна структура об'єктно-орієнтованої БД зовні є схожою на структуру ієрархічної БД. Основна відмінність між ними полягає в методах маніпулювання даними.

Для виконання дій над даними в розглянутій моделі БД використовують логічні операції, посилені об'єктно-орієнтованими механізмами інкапсуляції, успадкування та поліморфізму. Обмежено можна застосовувати операції, подібні командам SQL (наприклад, для створення БД).

Створіння й модифікація БД супроводжується автоматичним формуванням і подальшим коригуванням індексів (індексних таблиць), що містять інформацію для швидкого пошуку даних.

Розгляньмо коротко поняття інкапсуляції, спадкування та поліморфізму щодо об'єктно-орієнтованої моделі БД.

Інкапсуляція обмежує зону видимості властивості справами того об'єкта, у якому його визначено. Так, якщо в об'єкт типу КАТАЛОГ додати властивість, що задає телефон автора книги й має назву "телефон", то ми дістаємо однойменні властивості в об'єктів АБОНЕНТІВ і КАТАЛОГ. Сенс такої властивості буде визначатися тим об'єктом, у якому його інкапсульовано.

Спадкування, навпаки, поширює зону видимості властивості на всіх нащадків об'єкта. Так, усім об'єктам типу КНИГА, що є нащадками об'єкта типу КАТАЛОГ, можна надати властивості об'єкта-батька: ISBN, УДК, назву й автора. Якщо необхідно розширити дію механізму успадкування на об'єкти, які не є безпосередніми родичами (наприклад, між двома нащадками одного з батьків), то у їхнього спільного предка визначають абстрактну властивість типу ABS. Так, визначення абстрактних властивостей "квиток" і "номер" в об'єкті БІБЛІОТЕКА приводить до спадкоємства цих властивостей усіма дочірніми об'єктами АБОНЕНТ, КНИГА та ВИДАЧА. Чи не випадково, тому значення властивості "квиток" класів АБОНЕНТ і ВИДАЧА, показаних на рис. 2.10, будуть однаковими – 00015.

Поліморфізм в об'єктно-орієнтованих мовах програмування означає здатність одного й того самого програмного коду працювати з різнотипними даними. Інакше кажучи, він визначає допустимість в об'єктах різних типів мати методи (процедури або функції) з однаковими назвами. Під час виконання об'єктної програми одні й ті самі методи оперують різними об'єктами, залежно від типу аргументу. Що стосується об'єктно-орієнтованої БД, то поліморфізм означає, що об'єкти класу КНИГА, мають різних батьків із класу КАТАЛОГ, можуть мати різний набір властивостей. Отже, програми роботи з об'єктами класу КНИГА можуть містити поліморфний код.

Пошук в об'єктно-орієнтованої БД полягає у з'ясуванні схожості між об'єктом, що задає користувач, та об'єктами, що зберігають у БД. Об'єкт,

що визначає користувач, у загальному випадку може становити підмножину всієї інформації, яку зберігають у БД. Об'єкт-мета, а також результат виконання запиту можуть зберігати в самій базі.

Основною *перевагою* об'єктно-орієнтованої моделі даних, порівняно з реляційною, є можливість відображення інформації про складні взаємозв'язки об'єктів. Об'єктно-орієнтована модель даних дозволяє ідентифікувати окремий запис бази даних і визначати функції їхнього опрацювання.

Недоліками об'єктно-орієнтованої моделі є висока понятійна складність, незручність опрацювання даних і низька швидкість виконання запитів.

До об'єктно-орієнтованим СУБД належать: POET (POET Software), Jasmine (Computer; Associates), Versant (Versant Technologies), ODB-Jupiter, а також Iris, Orion і Postgres.

2.7. Типи даних

Спочатку СУБД застосовували переважно для вирішення фінансово-економічних завдань. Водночас, незалежно від моделі подання, у базах даних використовували такі основні типи даних:

- символьні;
- числові;
- дату/час;
- виконавчі;
- призначені для користувача.

Розгляньмо кожну категорію типів окремо.

Символьні типи. Їх використовують для подання як рядків символів, так і окремих символів. Дані символьних типів подано двійковими кодами. Те, як вони відображаються на екранах моніторів або роздруківках принтера, визначено так званими кодовими таблицями. Вони дозволяють відображати текстову інформацію символами різних мов, починаючи від англійської, російської, грецької, іспанської та закінчуючи китайською та японською. Інформацію про наявні кодові таблиці зберігають у системній таблиці SYS.SYSCOLLATION.

Перелік символьних типів наведено в табл. 2.1.

Таблиця 2.1

Перелік символьних типів

Типи даних	Призначення	Розміри
CHAR	Рядковий тип	до 32 767 байт, за замовчуванням 1 байт
LONG VARCHAR	Символьний тип довільної довжини. Аналог MEMO-полів в dBase, FoxPro, Access	Довжина є довільною. Обмежена максимальним розміром файлів бази даних (2 гігабайти)
TEXT	Теж, що й LONG VARCHAR	Довжина є довільною

Під час використання символьних даних потрібно перевірити, як відображається символьна інформація, що зберігають у базі даних на вашій ЕОМ. На різних ЕОМ, у різних операційних системах і навіть різних застосунках можна візуально подавати по-різному.

Числові типи. Їх призначено для позначення цілих, дійсних і грошових типів. Представників числових типів наведено в табл. 2.2.

Таблиця 2.2

Перелік числових типів

Типи даних	Діапазони значень	Точність (кількість знаків після коми)
1	2	3
INTEGER	від -2 147 483 648 до +2 147 483 647	0
INT	Теж, що й INTEGER	0
SMALLINT	від -32 768 до +32 767	0
REAL	від -3,4 e - 38 до 3,4 e + 38	до 6
FLOAT	Теж, що й REAL	до 6
DOUBLE	від -1,797 e - 308 до +1,797 e + 308	до 15
TINYINT	від 1 до 255	0

Закінчення табл. 2.2

1	2	3
DECIMAL	Числа, що складаються з N цифр із M цифрами у дробовій частині. За замовчуванням N = 30, M = 6	M
NUMERIC	Теж, що й DECIMAL	M
MONEY	Для зберігання грошових величин. Припустімо значення NULL. Числа із 20 цифр із 4 цифрами після коми	4
SMALLMONEY	Для зберігання грошових величин. Припустімо значення NULL. Числа з 10 цифр із 4 цифрами після коми	4

Типи "дата/час". Їх призначено для зберігання часу, дат і дат спільно із часом. Такі типи наведено в табл. 2.3.

Таблиця 2.3

Перелік типів "дата/час"

Типи даних	Призначення
DATE	Тип для подання дати у вигляді сукупності року, місяця та числа. Значення року може змінюватися в діапазоні від 0001 до 9999 року
TIME	Тип для подання часу у вигляді сукупності години, хвилин, секунд і часток секунд. Частки секунд зберігають із точністю до 6 знаків
TIMESTAMP	Тип для подання моменту часу конкретної дати. Дані зберігають у вигляді сукупності року, місяця, числа, години, хвилин, секунд і часток секунд. Частки секунд зберігають із точністю до 6 знаків

Різні поєднання цих параметрів породжують велику кількість варіантів форматів. Однак це не має викликати особливого занепокоєння. Параметри бази даних установлюють для всієї бази даних і діють на всі застосунки та всіх користувачів. Формати даних типу "дата/час", що визначають вказаними раніше параметрами баз даних, наведено в табл. 2.4.

Таблиця 2.4

**Формати подання даних типу "дата/час",
що визначають за замовчуванням**

Типи даних	Формати, які використовують за замовчуванням
DATE	'YYYY-MM-DD'
TIME	'HH: NN: ss.SSS'
TIMESTAMP	'YYYY-MM-DD HH: NN: ss.SSS'
DATETIME	'YYYY-MM-DD HH: NN: ss.SSS'
SMALLDATETIME	'YYYY-MM-DD HH: NN: ss.SSS'

У табл. 2.4 використовують такі умовні скорочення:

YYYY – чотири цифри, що позначають рік;

MM – дві цифри, що позначають місяць;

DD – дві цифри, що позначають день;

HH – дві цифри, що позначають години;

NN – дві цифри, що позначають хвилини;

ss – дві цифри, що позначають секунди;

SSS – три цифри, що позначають частки секунд.

За замовчуванням складові часу HH, NN, ss, SSS беруть такими, що дорівнюють нулю, а DD – одиниці. Уміст рядків, що подають дані типу "дата/час", конвертуються автоматично.

Двійкові типи. Їх призначено для подання двійкових даних, включно із зображенням й іншою інформацією, не опрацьованою власними засобами СУБД. Усі двійкові типи наведено в табл. 2.5.

Таблиця 2.5

Двійкові типи

Типи даних	Призначення	Розміри
1	2	3
BIT	Тип для подання значень 0 і 1. Аналог полів типу Logical у dBase, FoxPro	1 байт

Закінчення табл. 2.5

1	2	3
BINARY	Теж, що й CHAR, за винятком операцій порівняння. На відміну від CHAR, дані цього типу за замовчуванням 1 байт	до 32 767 байтів
LONG BINARY	Тип для подання двійкових даних довільної довжини	Довжина є довільною, обмеженою максимальним розміром файлів бази даних (2 гігабайти)
IMAGE	Теж, що й LONG BINARY	

Призначені для користувача типи даних. У SQL користувачам надано можливість створювати свої типи даних. Їх створюють на базі наявних типів шляхом:

- заборони/дозволу запису значень NULL;
- визначення значень за замовчуванням (установлення DEFAULT);
- задавання умов на записувані значення (установлення CHECK).

Право створення типів даних користувачів мають тільки ті з них, які мають право створювати об'єкти бази даних (клас повноважень Resource) або володіють правами адміністратора (клас повноважень DBA). Користувач, який створив новий тип даних, стає його власником. Відразу після появи цього типу даних доступ до нього дістають усі користувачі, зареєстровані в базі даних.

Новий тип даних можна застосовувати під час визначення типів полів та опису змінних у збережених процедурах і тригерів. Видалити новий тип може його власник або користувач із класом повноважень DBA. Видалення цього типу даних можливо тільки в тому разі, якщо його ніде не використовують.

На завершення аналізу типів даних необхідно зазначити таке. Велика кількість "своїх" типів даних і можливість створення типів даних користувачів мають задовольнити запити самого вимогливого користувача. У сучасних СУБД із різними моделями даних можуть використовувати всі перелічені типи даних.

Контрольні запитання

1. Перелічіть класичні та сучасні моделі подання даних.
2. Укажіть переваги й недоліки ієрархічної моделі даних.
3. Як організовують фізичне розміщення даних у БД ієрархічного типу?
4. Охарактеризуйте мережеву модель даних.
5. Охарактеризуйте реляційну модель даних.
6. У чому відмінність постреляційної від реляційної моделі даних?
7. Укажіть переваги й недоліки постреляційної моделі.
8. Охарактеризуйте багатовимірну модель даних.
9. Назвіть і поясніть сенс операцій, здійсненних над даними в разі багатовимірної моделі.
10. Дайте визначення й наведіть приклади вияву принципів інкапсуляції, поліморфізму та наслідування щодо об'єктно-орієнтованих баз даних.
11. Укажіть переваги й недоліки об'єктно-орієнтованої моделі подання даних.
12. Охарактеризуйте типи даних, що використовують у сучасних СУБД.

Рекомендована література: [1; 5; 7].

3. Реляційні бази даних

Мета – Визначення понять "структура бази даних", "правило Кодда", "ключі" та їхніх складових. Розуміння сутності реляційної бази даних, створення ключів для зв'язку відношень.

Основні питання

3.1. Створення реляційної системи даних.
3.2. Сутність реляційної бази даних.
3.3. Створення ключів для зв'язку відносин.
3.4. Види зв'язків між таблицями.

Ключові слова: структура, схема відношення, кортеж, атрибут, первинний ключ, нормалізація.

3.1. Створення реляційної системи даних

Зазвичай, будь-який вебзастосунок можна розподілити на дві основні частини: фронтенд, де відображають усю інформацію сайта, і бекенд, де цю інформацію формують і розміщують (рис. 3.1). База даних зберігає записи у спеціально організованому вигляді, щоб інформацію можна було легко знайти та вилучити. Будь-яка БД складається з однієї або декількох таблиць. Електронна таблиця складається з рядків і стовпців. Усі рядки мають однакові стовпці, а кожен стовпець містить дані.

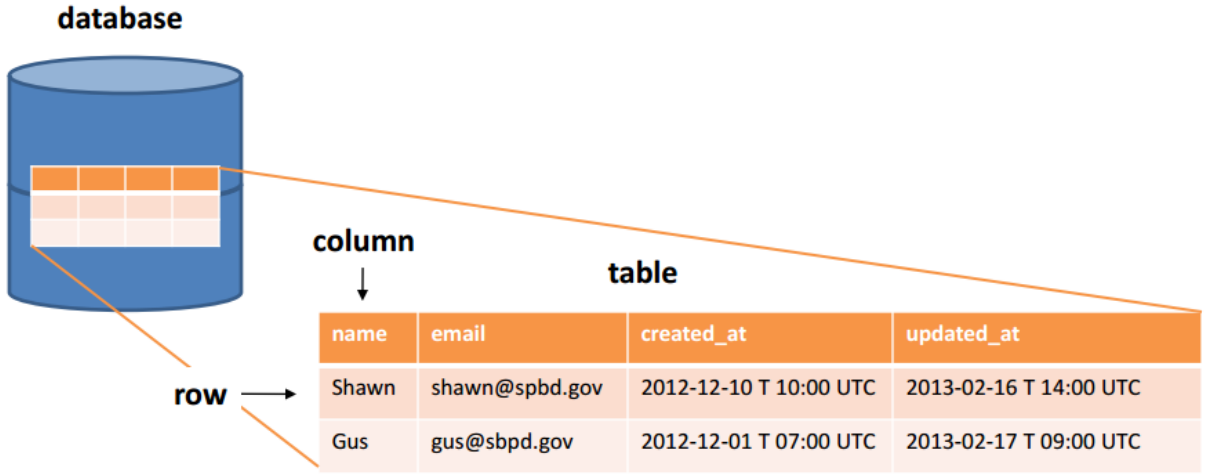


Рис. 3.1. **Структура бази даних**

Табличні дані можуть бути вставленими, відновленими, оновленими та видаленими. Для пакета цих операцій було створено спеціальну абревіатуру – CRUD (Create-Read-Update-Delete).

Реляційні бази даних – це бази, де всю інформацію зберігають у таблицях, пов'язаних одна з одною спеціальними відношеннями. Ці відношення дозволяють нам здобувати й об'єднувати дані з однієї або декількох таблиць за допомогою одного запиту.

Реляційна база даних – база даних, побудована на основі реляційної моделі. У реляційній базі кожен об'єкт задають відповідним записом (рядком) у таблиці. Реляційну базу створюють та потім управляють за допомогою реляційної системи управління базами даних. Фактично реляційна база даних – це інформація, що зберігають у двовимірних таблицях. Зв'язок між таблицями може мати своє відображення у структурі даних, а може тільки матися на увазі, тобто бути наявним на неформалізованому

рівні. Кожну таблицю БД подають як сукупність рядків і стовпців, де рядки відповідають екземпляру об'єкта, конкретної події або явища, а стовпці – атрибутам (ознаками, характеристиками, параметрами) об'єкта, події, явища. Реляційні бази даних надають більш простий доступ до складання звітів (зазвичай через SQL) і забезпечують підвищену надійність та цілісність даних, завдяки відсутності надлишкової інформації.

Реляційні системи беруть свій початок у математичної теорії множин. Едгар Кодд, співробітник дослідної лабораторії корпорації IBM, насправді, створив та описав концепцію реляційних баз даних. Нечіткість багатьох термінів, які використовують у сфері опрацювання даних, змусила Кодда відмовитися від них і придумати нові або дати більш точні визначення наявних. Так, він не міг використовувати значно поширений термін "запис", який у різних ситуаціях може означати екземпляр запису або тип записів; запис у стилі Кобола (який допускає повторювані групи) або плоский запис (який їх не допускає); логічний або фізичний запис, що зберігають, або віртуальний запис тощо. Замість цього він використовував термін "кортеж довжини *n*" або просто "кортеж", якому дав точне визначення.

Кодд запропонував модель, яка дозволяє розробникам розподіляти свої бази даних на окремі, але взаємопов'язані таблиці, що підвищує продуктивність, але водночас зовнішнє подання залишається тим самим, що й у вихідної бази даних. Відтоді Кодда вважають батьком-засновником галузі реляційних баз даних. Учений сформулював 13 правил для реляційних баз даних, більшість яких стосуються цілісності й поновлення даних, а також доступу до них.

Правила Кодда

Правило 0. Основне правило (Foundation Rule). Система, яку рекламують або позиціонують як реляційну систему управління базами даних, має бути здатною управляти базами даних, використовуючи виключно свої реляційні можливості.

Правило 1. Інформаційне правило (The Information Rule). Усю інформацію в реляційній базі даних на логічному рівні має бути явно подано єдиним способом: значеннями в таблицях.

Правило 2. Гарантований доступ до даних (Guaranteed Access Rule). У реляційній базі даних кожне окреме (атомарне) значення даних має бути логічно доступним за допомогою комбінації назви таблиці, значення первинного ключа та назви стовпця.

Правило 3. Систематичне підтримання відсутніх значень (Systematic Treatment of Null Values). Невідомі, або відсутні значення NULL, відмінні від будь-якого відомого значення, мають підтримувати для всіх типів даних під час виконання будь-яких операцій. Наприклад, для числових даних невідомі значення не мають розглядати як нулі, а для символьних даних – як порожні рядки.

Правило 4. Доступ до словника даних у термінах реляційної моделі (Active On-Line Catalog Based on the Relational Model). Словник даних мають зберігати у формі реляційних таблиць, і СУБД має підтримувати доступ до нього за допомогою стандартних мовних засобів, тих самих, які використовують для роботи з реляційними таблицями, що містять призначені для користувача дані.

Правило 5. Повнота підмножини мови (Comprehensive Data Sublanguage Rule). Система управління базами даних має підтримувати хоча б одну реляційну мову, що:

- має лінійний синтаксис;
- можуть використовувати як інтерактивно, так і в прикладних програмах;
- підтримує операції визначення даних, визначення подань, маніпулювання даними (інтерактивними та програмними), обмежувачі цілісності, управління доступом та операції управління транзакціями (begin, commit і rollback).

Правило 6. Можливість зміни подань (View Updati ng Rule). Кожне подання має підтримувати всі операції маніпулювання даними, які підтримують реляційні таблиці: операції вибірки, уставлення, зміни та видалення даних.

Правило 7. Наявність високорівневих операцій управління даними (High-Level Insert, Update, and Delete). Операції вставлення, зміни та видалення даних мають підтримувати не тільки щодо одного рядка реляційної таблиці, але й будь-якої множини рядків.

Правило 8. Фізична незалежність даних (Physical Data Independence). Застосунки не мають залежати від використовуваних способів зберігання даних на носіях, апаратного забезпечення комп'ютерів, на яких міститься реляційна база даних.

Правило 9. Логічна незалежність даних (Logical Data Independence). Подання даних у застосунку не має залежати від структури реляційних таблиць. Якщо у процесі нормалізації одну реляційну таблицю розподіляють

на дві, подання має забезпечити об'єднання цих даних, щоб зміна структури реляційних таблиць не позначалася на роботі застосунків.

Правило 10. Незалежність контролю за цілісністю (Integrity Independence). Уся інформація, необхідна для підтримання цілісності, має міститися у словнику даних. Мова для роботи з даними має виконувати перевірку вхідних даних та автоматично підтримувати цілісність даних.

Правило 11. Незалежність від розташування (Distribution Independence). База даних може бути розподіленою, може міститися на декількох комп'ютерах, і це не має впливати на застосунки. Перенесення бази даних на інший комп'ютер не має впливати на застосунки.

Правило 12. Погодження мовних рівнів (The Nonsubversion Rule). Якщо використовують низькорівневу мову доступу до даних, вона не має ігнорувати правила безпеки та цілісності, які підтримують мовою більш високого рівня.

3.2. Сутність реляційної бази даних

Реляційна база даних становить набір таблиць (сутностей). Таблиці складаються зі стовпців і рядків (кортежів). Усередині таблиць може бути визначено обмеження, між таблицями є відношення.

Під час створення інформаційної системи сукупність відношень дозволяє зберігати дані про об'єкти предметної галузі та моделювати зв'язок між ними. Елементи реляційної моделі даних і форми їхнього подання наведено в табл. 3.1.

Таблиця 3.1

Елементи реляційної моделі

Елементи реляційної моделі	Форми подання
Відношення	Таблиця
Схема відношень	Рядок заголовків колонок таблиці (заголовок таблиці)
Кортеж	Рядок таблиці
Сутність	Опис властивостей об'єкта
Атрибут	Назва стовпців таблиці
Домен	Множина допустимих значень атрибута
Значення атрибута	Уміст клітинки в запису
Первинний ключ	Один або кілька атрибутів
Тип даних	Тип значень елементів таблиці

Відношення є найважливішим поняттям і становить двовимірну таблицю, яка містить деякі дані.

Сутність є об'єкт будь-якої природи, дані про який зберігають у базі даних. Дані про сутність зберігають у відношенні.

Атрибути становлять властивості, що характеризують сутність. У структурі таблиці кожен атрибут називають і йому відповідає заголовок деякого стовпця таблиці.

За допомогою SQL можна виконувати запити, які повертають набори даних, здобутих з однієї або декількох таблиць. У межах одного запиту, дані виходять із декількох таблиць шляхом їхнього з'єднання (JOIN). Найчастіше для з'єднання використовують ті самі стовпці, які визначають відношення між таблицями.

Нормалізація – це процес структурування моделі даних, що забезпечує зв'язність і відсутність надмірності в даних. Метою нормалізації реляційної бази даних є усунення недоліків структури бази даних, що призводять до надмірності, яка, своєю чергою, потенційно призводить до різних аномалій і порушень цілісності даних. Теоретики реляційних баз даних у процесі розвитку теорії виявили й описали типові приклади надмірності та способи їхнього усунення. Реляційні сховища забезпечують найкраще поєднання простоти, стійкості, гнучкості, продуктивності, масштабованості та сумісності. Що стосується масштабованості, реляційні БД добре масштабувати тільки в тому разі, якщо їх розташовано на єдиному сервері.

Особливістю реляційної бази даних є використання в ній реляційної моделі даних і наслідків, що випливають із цього:

- модель даних у реляційних БД визначено заздалегідь. Вона є точно типізованою, містить обмеження та відношення для забезпечення цілісності даних;
- модель даних засновано на природному поданні даних, а не на функціональності застосунку;
- модель даних підлягає нормалізації, щоб уникнути дублювання даних. Нормалізація породжує відношення між таблицями. Відношення пов'язують дані різних таблиць.

У реляційній базі даних дані створюють, оновлюють, видаляють і запитують із використанням мови структурованих запитів (SQL). SQL-запити можуть здобувати дані як з одиночної таблиці, так і з декількох таблиць. Такі запити можуть містити агрегації та складні фільтри. Реляційна БД зазвичай містить убудовану логіку, таку як тригери, збережені процедури та функції.

Доступ до реляційних баз даних здійснюють через реляційні системи управління базами даних (РСУБД). Майже всі системи баз даних, які використовують, є реляційними, як-от Oracle, Microsoft SQL Server, MySQL, Sybase, DB2, TeraData тощо. Причини такого домінування є очевидними.

Як приклад розгляньмо таблицю, що містить дані про співробітника (рис. 3.2).

Рис. 3.2. **Подання відношення СПІВРОБІТНИК**

У загальному випадку порядок кортежів у відношенні, як і будь-якій множині, не визначено. Однак у реляційних СУБД для зручності кортежі все ж упорядковують. Найчастіше для цього вибирають деякий атрибут, за яким система автоматично сортує кортежі за зростанням або спаданням. Якщо користувач не призначає атрибута впорядкування, система автоматично надає номер кортежам у порядку їхнього введення.

Домен є множиною всіх можливих значень певного атрибута відношення. Відношення СПІВРОБІТНИК містить чотири домени. Домен 1 містить прізвища всіх співробітників, домен 2 – номери всіх відділів фірми, домен 3 – назви всіх посад, домен 4 – дати народження всіх співробітників. Кожен домен утворює значення одного типу даних, наприклад, числові або символьні.

Відношення СПІВРОБІТНИК містить три кортежі. Кортеж розглянутого відношення складається із чотирьох елементів, кожен із яких вибирають із відповідного домену. Кожному кортежу відповідає рядок таблиці (див. рис. 3.2).

Схема відношення (заголовок) становить список назв атрибутів. Наприклад, для наведеного прикладу схема відношення має вигляд СПІВРОБІТНИК (ПІБ, Відділ, Посада, Д_Народження). Множину власне кортежів відношення часто називають тілом відношення.

3.3. Створення ключів для зв'язку відношень

Первинним ключем (ключем відношення, ключовим атрибутом) називають атрибут відношення, якій однозначно ідентифікує кожен із його кортежів.

Наприклад, щодо СПІВРОБІТНИК (ПІБ, Відділ, Посада, Д_Народження) ключовим є атрибут "ПІБ".

Ключ може бути складеним (складним), тобто складатися з декількох атрибутів. Кожне відношення в цьому разі має комбінацію атрибутів, яка може слугувати ключем. Його наявність гарантовано тим, що відношення – це множина, що не містить однакових елементів – кортежів. Тобто у відношенні немає повторюваних кортежів, а це значить, що принаймні вся сукупність атрибутів має властивість однозначної ідентифікації.

Якщо вибраний первинний ключ складається з мінімально необхідного набору атрибутів, уважають, що він є ненадмірним.

Ключі зазвичай використовують для досягнення таких цілей:

1) виключення дублювання значень у ключових атрибутах (інші атрибути до уваги не беруть);

2) упорядкування кортежів. Можливе впорядкування за зростанням або спаданням значень усіх ключових атрибутів, а також змішане впорядкування (за одними – зростання, а за іншими – спадання);

3) прискорення роботи з кортежами відношення;

4) організації зв'язування таблиць.

За допомогою зовнішніх ключів установлюють зв'язки між відношеннями. Наприклад, є два відношення СТУДЕНТ (ПІБ, Група, Спеціальність) і ПРЕДМЕТ (Назв. Пр., Години), пов'язані відношенням СТУДЕНТ – ПРЕДМЕТ (ПІБ, Назв. Пр. Оцінка) (рис. 3.3). У пов'язаному відношенні атрибути ПІБ і Назв. Пр утворюють складений ключ. Ці атрибути є зовнішніми ключами, що є первинними ключами інших відношень.

Рис. 3.3. **Зв'язок відношень**

Реляційна модель накладає на зовнішні ключі обмеження для забезпечення цілісності даних, назване *посилальною цілісністю*. Це означає, що кожному значенню зовнішнього ключа мають відповідати рядки у відношеннях, що пов'язують.

Оскільки не будь-якій таблиці можна поставити у відповідність відношення, наведімо умови, виконання яких дозволяє таблицю вважати відношенням.

1. Усі рядки таблиці мають бути унікальними, тобто не може бути рядків з однаковими первинними ключами.

2. Назви стовпців таблиці мають бути різними, а значення їх – простими. Недопустимою є група значень в одному стовпці одного рядка.

3. Усі рядки однієї таблиці мусять мати одну структуру, відповідну назвам і типам стовпців.

4. Порядок розміщення рядків у таблиці може бути довільним.

Найбільш часто таблицю з відношенням розміщено в окремому файлі. У деяких СУБД одну окрему таблицю (відношення) уважають базою даних. В інших СУБД база даних може містити кілька таблиць.

У загальному випадку можна вважати, що БД містить одну або кілька таблиць, об'єднаних смисловим змістом, а також процедурами контролю за цілісністю й опрацювання інформації в інтересах розв'язання деякої прикладної задачі. Наприклад, під час використання СУБД Microsoft Access у файлі БД разом із таблицями зберігають й інші об'єкти бази: запити, звіти, форми, макроси та модулі.

Таблицю даних зазвичай зберігають на магнітному диску в окремому файлі операційної системи, тому з її назвою можуть бути обмеження. Назви полів зберігають усередині таблиць. Правила їхнього формування визначають СУБД, які, зазвичай, на довжину полів і використовуваний алфавіт серйозних обмежень не накладають.

Якщо таблиця відношення має ключ, то її називають *ключовою*, або *таблицею із ключовими полями*.

У більшості СУБД файл таблиці містить керівну частину (опис типів полів, назви полів та іншу інформацію) і зону розміщення записів.

До відношень можна застосовувати систему операцій, що дозволяє здобувати одні відношення з інших. Наприклад, результатом запиту до реляційної БД може бути нове відношення, обчислене на основі наявних відношень. Тому можна розподілити опрацьовувані дані на збережену

й обчислювану частини. Основною одиницею опрацювання даних у реляційних БД є відношення, а не окремі його кортежі (записи).

3.4. Види зв'язків між таблицями

Під час проєктування реальних БД інформацію зазвичай розміщують у декількох таблицях. Таблиці водночас пов'язано семантикою інформації. У реляційних СУБД для зв'язків таблиць виконують операцію їхнього зв'язування.

Багато СУБД під час зв'язування таблиць автоматично виконують контроль за цілісністю, тобто вводять у базу дані, відповідно до встановлених зв'язків. У кінцевому підсумку це підвищує вірогідність інформації, що зберігають у БД.

Крім того, установлення зв'язку між таблицями полегшує доступ до даних. Зв'язування таблиць під час виконання таких операцій, як пошук, перегляд, редагування, вибірка та підготовка звітів, зазвичай забезпечує можливість звернення до довільних полів пов'язаних записів. Це зменшує кількість явних звернень до таблиць даних і маніпуляцій у кожній із них.

Між таблицями можуть установлюватися бінарні (між двома таблицями), тернарні (між трьома таблицями) і, в загальному випадку, *n*-арні зв'язки. Розгляньмо бінарні зв'язки, що найбільш часто зустрічають.

Під час зв'язування двох таблиць виділяють основну й додаткову (підпорядковану) таблиці. Логічне зв'язування таблиць здійснюють за допомогою ключа зв'язку.

Ключ зв'язку складається з одного або декількох полів, які в цьому разі називають *полями зв'язку* (ПЗ).

Сенс зв'язування полягає у встановленні відповідності полів зв'язку основної та додаткової таблиць. Поля зв'язку основної таблиці можуть бути звичайними та ключовими. Як поля зв'язку підпорядкованої таблиці найчастіше використовують ключові поля.

Залежно від того, як визначено поля зв'язку основної й додаткової таблиць (як співвідносяться ключові поля з полями зв'язку), між двома таблицями в загальному випадку можуть встановлюватися такі чотири види зв'язку (табл. 3.2).

Таблиця 3.2

Характеристика видів зв'язків таблиць

Характеристика полів зв'язку за видами	Види зв'язку			
	1:1	1:M	M:1	M:M
Поля зв'язку основної таблиці	є ключем	є ключем	не є ключем	не є ключем
Поля зв'язку додаткової таблиці	є ключем	не є ключем	є ключем	не є ключем

Дамо характеристику названим видам зв'язку між двома таблицями й наведемо приклади їхнього використання.

Зв'язок виду 1:1 утворюється в разі, коли всі поля зв'язку основної та додаткової таблиць є ключовими. Оскільки значення у ключових полях обох таблиць не повторюються, забезпечується взаємно однозначна відповідність записів із цих таблиць. Самі таблиці, насправді, тут стають рівноправними.

Приклад. Нехай є основна О1 і додаткова Д1 таблиці. Ключові поля позначмо символом "*", що використовують для зв'язку поля позначмо символом "+" (рис. 3.4).

Таблиця О1

* +

Поле 11	Поле 12
а	10
б	40
в	3

Таблиця Д1

* +

Поле 21	Поле 22
а	стіл
в	книга

Рис. 3.4. **Зв'язок виду 1:1**

У наведених таблицях установлено зв'язок між записом (а, 10) табл. О1 і записом (а, стіл) табл. Д1. Аналогічна зв'язок є і між записами (в, 3) і (в, книга) цих же таблиць. Підставою для цього є збіг значень у полях зв'язку. У таблицях записи є відсортованими за значеннями у ключових полях.

Зіставлення записів двох таблиць насправді, означає утворення нових "віртуальних записів" (псевдозаписів). Так, першу пару записів логічно можна вважати новим псевдозаписом виду (а, 10, стіл), а другу пару – псевдозаписом виду (в, 3, книга).

На практиці зв'язок виду 1:1 використовують порівняно рідко, оскільки збережену у двох таблицях інформацію легко об'єднати в одну таблицю, яка займає набагато менше місця в пам'яті ЕОМ.

Зв'язок виду 1:М має місце в разі, коли одному запису основної таблиці відповідає декілька записів допоміжної таблиці.

Приклад. Нехай є дві зв'язані табл. О2 і Д2. У табл. О2 міститься інформація про види мультимедіа-пристроїв ПЕОМ, а в табл. Д2 – відомості про фірми-виробники цих пристроїв, а також про наявність на складі хоча б одного пристрою.

Табл. Д2 має два ключові поля, оскільки одна й та сама фірма може виробляти пристрої різних видів. У прикладі фірма Sony виробляє пристрої зчитування та перезапису з компакт-дисків.

Зіставлення записів обох таблиць за полем <Код> породжує псевдозаписи виду: (а, CD-ROM, Acer, є), (а, CD-ROM, Mitsumi, немає), (а, CD-ROM, NEC, є), (а, CD-ROM, Panasonic, є), (а, CD-ROM, Sony, є), (б, CD-Recorder, Philips, немає), (б, CD-Recorder, Sony, є) тощо (рис. 3.5).

Таблиця О2

* +

Коди	Види пристроїв
а	CD-ROM
б	CD-Recorder
в	Sound Blaster

Таблиця Д2

* + *

Коди	Фірми-виробники	Наявність
а	Acer	є
а	Mitsumi	немає
а	NBC	є
а	Panasonic	є
а	Sony	є
б	Philips	немає
б	Sony	немає
б	Yamaha	є
в	Creative Labs	є

Рис. 3.5. **Зв'язок виду 1:М**

Якщо звести псевдозаписи в нову таблицю, то здобудемо повну інформацію про всі види мультимедіа-пристроїв ПЕОМ, фірм, які їх виробляють, а також відомості про наявність конкретних видів пристроїв на складі.

Зв'язок виду М:1 має місце в разі, коли одному або кільком записам основної таблиці ставлять у відповідність один запис допоміжної таблиці.

Приклад. Розгляньмо зв'язок табл. О3 і Д3. В основній табл. О3 міститься інформація про назви деталей, види матеріалів, із якого ці деталі можна виготовити, і марки матеріалу. У додатковій табл. Д3 містяться дані про назву деталей, терміни виготовлення та вартість замовлень (рис. 3.6).

Таблиця О3

\+

Назви	Види матеріалу	Марки матеріалів
деталь 1	чавун	марка 1
деталь 1	чавун	марка 2
деталь 2	сталь	марка 1
деталь 2	сталь	марка 2
деталь 3	алюміній	–
деталь 4	чавун	марка 2

Таблиця Д3

* +

Назви	Терміни виготовлення	Вартість замовлення
деталь 1	04.03.2022 р.	90
деталь 2	03.01.2022 р.	35
деталь 3	17.02.2022 р.	90
деталь 4	06.05.2022 р.	240

Рис. 3.6. **Зв'язок виду М:1**

Зв'язування цих таблиць забезпечує таке встановлення відповідності між записами, яке еквівалентно утворенню таких псевдозаписів: (деталь 1, чавун, марка 1, 04.03.2022 р., 90); (деталь 1, чавун, марка 2, 04.03.2022 р., 90); (деталь 2, сталь, марка 1, 03.01.2022 р., 35); (деталь 2, сталь, марка 2, 03.01.2022 р., 35); (деталь 2, сталь, марка 3, 03.01.2022 р., 35); (деталь 3, алюміній; –, 17.02.2022 р., 90); (деталь 4, чавун, марка 2, 06.05.2022 р., 240).

Визначена псевдотаблиця може бути корисною під час планування або ухвалення управлінських рішень, якщо необхідно мати всі можливі варіанти виконання замовлень за кожним виробом. Зазначмо, що табл. О3 не має ключів і в ній можливе повторення записів.

Найбільший загальний **вид зв'язку М:М** виникає в разі, якщо декільком записам основної таблиці відповідає декілька записів додаткової таблиці.

Приклад. Нехай в основній табл. О4 міститься інформація про те, на яких верстатах можуть працювати робітники деякої бригади. Табл. Д4 містить відомості про те, хто із бригади ремонтників які верстати обслуговує.

Першому і третьому записам табл. О4 відповідає перший запис табл. Д4 (у всіх цих записах значення другого поля "верстат 1"). Четвертим записом табл. О4 відповідають другий і четвертий записи табл. Д4 (у другому полі цих записів міститься "верстат 3") (рис. 3.7).

Таблиця О4

*	* +
Працює	на верстаті
Іванов А. В.	верстат 1
Іванов А. В.	верстат 2
Петров Н. Г.	верстат 1
Петров Н. П.	верстат 3
Сидоров В. К.	верстат 2

Таблиця Д4

*	* +
Обслуговує	Верстат
Голубєв Б. С.	верстат 1
Голубєв Б. С.	верстат 3
Зиков А. Ф.	верстат 2
Зиков А. Ф.	верстат 3

Рис. 3.7. **Зв'язок виду М:М**

З огляду визначення полів зв'язку цих таблиць, можна скласти нову таблицю з назвою "О4 + Д4", записами якої будуть псевдозаписи. Записам визначеної таблиці можна надати сенс можливих змін, що виникають під час планування роботи. Для зручності поля нової таблиці перейменовано (до речі, таку операцію пропонують багато сучасних СУБД) (рис. 3.8).

Таблиця "О4 + Д4"

Робота	Верстати	Обслуговування
Іванов А. В.	верстат 1	Голубєв Б. С.
Іванов А. В.	верстат 2	Зиков А. Ф.
Петров Н. Г.	верстат 1	Голубєв Б. С.
Петров Н. П.	верстат 3	Голубєв Б. С.
Петров Н. Г.	верстат 3	Зиков А. Ф.
Сидоров В. К.	верстат 2	Зиков А. Ф.

Рис. 3.8. **Зведена таблиця**

Наведену таблицю можна використовувати, наприклад, для отримання відповіді на запитання: "Хто обслуговує верстати, на яких працює Петров Н. Г.?"

Очевидно, аналогічно зв'язку 1:1, зв'язок М:М не встановлює підпорядкованості таблиць. Для перевірки цього можна основну й додаткову таблицю поміняти місцями та виконати об'єднання інформації шляхом зв'язування. Результівні табл. "О4 + Д4" і "Д4 + О4" будуть відрізнятися порядком проходження першого та третього полів, а також порядком розташування записів.

Контрольні запитання

1. Розкрийте поняття "структура бази даних".
2. У чому полягає сутність реляційної бази даних?
3. Як створюють ключі для зв'язку відношень?
4. Перелічіть види зв'язків між таблицями.

Рекомендована література: [3; 4; 6].

4. Проєктування баз даних. Нормалізація

Мета – визначення понять "нормальна форма", "цілісність баз даних" та їхніх складових. Розуміння основних підходів під час проєктування структур даних.

Основні питання

4.1. Проблеми проєктування баз даних.
4.2. Метод нормальних форм.
4.3. Перша нормальна форма.
4.4. Друга нормальна форма.
4.5. Третя нормальна форма.
4.6. Четверта нормальна форма.
4.7. П'ята нормальна форма.
4.8. Забезпечення цілісності.

Ключові слова: нормальна форма, цілісність даних, дублювання, аномалії, функціональна залежність.

4.1. Проблеми проєктування баз даних

Проєктування інформаційних систем, що містять бази даних, здійснюють на фізичному та логічному рівнях.

Розв'язання проблем проєктування на *фізичному рівні* багато в чому залежить від СУБД, що використовують. У ряді випадків користувачі мають можливість налаштування окремих параметрів системи, яка не становить великої проблеми.

Логічне проєктування полягає у визначенні кількості та структури таблиць, формуванні запитів до БД, визначенні типів звітних документів, розроблення алгоритмів опрацювання інформації, створення форм для введення й редагування даних у базі та вирішенні ряду інших завдань.

Рішення завдання логічного проєктування БД переважно визначено специфікою предметної галузі. Найбільш важливої тут є проблема структуризації даних, на ній зосередьмо основний погляд.

Під час проєктування структур даних для автоматизованих систем можна виділити три основні підходи:

1. Збирання інформації про об'єкти вирішуваного завдання в межах однієї таблиці й подальша декомпозиція її на декілька взаємопов'язаних таблиць на основі процедури нормалізації відношень.

2. Формулювання знань про систему (визначення типів вихідних даних і їхніх взаємозв'язків) і вимог до опрацювання даних, побудова за допомогою CASE-системи (системи автоматизації проєктування й розроблення баз даних) готової схеми БД або навіть готової прикладної інформаційної системи.

3. Структурування інформації для використання в інформаційній системі у процесі здійснення системного аналізу на основі сукупності правил.

Розгляньмо перший із названих підходів, що є історично першим. Найперше охарактеризуймо основні проблеми, що мають місце під час визначення структур даних у відношеннях реляційної моделі.

Проблема 1. Надлишкове дублювання даних та аномалії.

Слід розрізняти просте й надлишкове дублювання даних. Надлишкове дублювання даних може призводити до проблем під час опрацювання даних. Наведімо приклади обох варіантів дублювання.

Приклад ненадлишкового дублювання даних відношення "Співробітник і Телефон" (С_Т) показано на рис. 4.1.

С_Т

Співробітники	Телефони
Іванов І. М.	37-21
Петров М. І.	43-28
Сидоров Н. Г.	43-28
Єгоров В. В.	43-28

Рис. 4.1. **Ненадлишкове дублювання**

Для співробітників, які перебувають в одному приміщенні, номери телефонів збігаються. Номер телефону 43-28 зустрічають кілька разів, хоча для кожного службовця номер телефону є унікальним. Тому жоден із номерів не є надлишковим. Дійсно, під час видалення одного з номерів буде загублено інформацію про те, за яким номером можна зателефонувати до одного із працівників.

Приклад надлишкового дублювання показано на рис. 4.2а. Відношення С_Т_Н доповнено атрибутом "номер кімнати співробітника". Природно припустити, що кожен, хто працює в одній кімнаті мають один і той самий номер телефону. Отже, у цьому відношенні є надлишкове дублювання даних. Так, у зв'язку з тим, що Сидоров і Єгоров перебувають в тій самій кімнаті, що й Петров, їхні номери можна дізнатися з кортежу з даними про Петрова.

С_Т_Н

Співробітники	Телефони	Н_кімн
Іванов І. М.	37-21	109
Петров М. І.	43-28	111
Сидоров Н. Г.	43-28	111
Єгоров В. В.	43-28	111

а

Співробітники	Телефони	Н_кімн
Іванов І. М.	37-21	109
Петров К. Ш.	43-28	111
Сидоров Н. Г.	–	111
Єгоров В. В.	–	111

б

Рис. 4.2. **Надлишкове дублювання**

На рис. 4.2б наведено приклад невдалого відношення С_Т_Н, у якому замість телефонів Сидорова та Єгорова поставлено прочерки. Не вдалість подібного способу усунення надлишкових даних полягає в такому:

по-перше, під час програмування доведеться прикласти додаткові зусилля на створення механізму пошуку інформації для таблиці;

по-друге, пам'ять все одно буде виділитися під клітинки із прочерками, хоча дублювання даних і усунено;

по-третє, що особливо важливо під час вилучення з колективу Петрова кортеж із відомостями про нього буде видалено з відношення, а значить, видалено інформацію про телефон 111-ї кімнати, що не припустимо.

Можливий спосіб виходу із цієї ситуації наведено на рис. 4.3. Тут показано два відношення С_Н і Н_Т, здобуті шляхом декомпозиції вихідного відношення С_Т_Н. Перше з них містить інформацію про номери кімнат, у яких розташовуються співробітники, а друге – інформацію про номери телефонів у кожній із кімнат. Тепер, якщо Петрова і звільнять з установи та, як наслідок цього, видалять будь-яку інформацію про нього з баз даних установи, це не призведе до втрати інформації про номер телефону в 111-й кімнаті.

Т_Н

Телефони	Н_кімн
37-21	109
43-28	111

С_Н

Співробітники	Н_кімн
Іванов І. М.	109
Петров М. І.	111
Сидоров Н. Г.	111
Єгоров В. В.	111

Рис. 4.3. **Вилучення надлишкового дублювання**

Процедура декомпозиції відношення С_Т_Н на два відношення С_Н і Н_Т є основною процедурою нормалізації відношень.

Надлишкове дублювання даних створює проблеми під час опрацювання кортежів відношення, названі *аномаліями проблеми відношення*. Ці проблеми виникають у разі спроби видалення, додавання або редагування кортежів.

Аномаліями називають таку ситуацію в таблицях БД, яка призводить до суперечностей у БД або істотно ускладнює опрацювання даних.

Виділяють три основні види аномалій: аномалії модифікації (або редагування), аномалії видалення та аномалії додавання.

Аномалії модифікації виявляють у тому, що зміна значення одного даного може спричинити перегляд усієї таблиці та відповідну зміну деяких інших записів таблиці.

Так, наприклад, зміна номера телефону в кімнаті 111 (див. рис. 4.2а) потребує перегляду всієї таблиці С_Т_Н і зміни поля Н_кімн, відповідно до поточного вмісту таблиці в записах, що належать до Петрова, Сидорова та Єгорова.

Аномалії видалення полягають у тому, що під час видалення будь-якого даного з таблиці може зникнути й інша інформація, яка не пов'язана безпосередньо з даними, що видаляють.

У тій самій таблиці С_Т_Н видалення запису про співробітника Іванова (наприклад, через звільнення) призводить до зникнення інформації про номер телефону, установленого у 109-й кімнаті.

Аномалії додавання виникають у разі, коли інформацію в таблицю не можна помістити до тих пір, поки вона є незаповненою, або вставлення нового запису потребує додаткового перегляду таблиці.

Прикладом може слугувати операція додавання нового співробітника в ту саму таблицю С_Т_Н. Очевидно буде протиприродним зберігання в цій таблиці відомостей тільки про кімнату й номер телефону в ній, поки нікого зі співробітників не вміщено в неї. Крім того, якщо в таблиці С_Т_Н поле "Співробітник" є ключовим, то зберігання в ній неповних записів із відповідним прізвищем співробітника просто неприпустимо через невизначеність ключового поля.

Другим прикладом виникнення аномалії додавання може бути ситуація вміщення в таблицю нового співробітника. Під час додавання таких записів для усунення суперечностей бажано перевірити номер телефону та відповідний номер кімнати хоча б з одним із співробітників, що перебувають із новим співробітником в тій самій кімнаті. Якщо ж виявиться, що в декількох співробітників, що перебувають в одній кімнаті, є різні номери телефонів, то взагалі незрозуміло, що робити (чи то в кімнаті є кілька телефонів, чи то якийсь із номерів є помилковим).

Проблема 2. Формування вихідного відношення.

Проєктування БД починається з визначення всіх об'єктів, відомості про яких буде вміщено в базу, і визначення їхніх атрибутів. Потім атрибути зводять в одну таблицю – вихідне відношення.

Приклад. Формування вихідного відношення.

Припустімо, що для навчальної частини факультету створюють БД про викладача. На першому етапі проєктування БД у результаті спілкування із завідувачем навчальної частини має бути визначено відомості про те, як базу даних мають використовувати, та яку інформацію замовник

хоче здобути у процесі її експлуатації. У результаті встановлюють атрибути, які мають міститися у відношеннях БД, і зв'язки між ними. Перелічмо назви виділених атрибутів і їхні короткі характеристики:

ПІБ – прізвище та ініціали викладача. Усуваймо можливість збігу прізвища та ініціалів у викладачів.

Посада – посада, яку займає викладачем.

Оклад – оклад викладача.

Стаж – викладацький стаж.

Д_Стаж – надбавка за додатковий стаж.

Каф – номер кафедри, на якій працює викладач.

Предм – назва предмета (навчальної дисципліни), що викладає викладач.

Група – номер групи, у якій викладач проводить заняття.

ВидЗан – вид занять, що проводить викладач у навчальній групі.

Одна з вимог до відношення полягає в тому, щоб всі атрибути відношення мали атомарні (прості) значення. У вихідному відношенні кожен атрибут кортежу також має бути простим. Приклад вихідного відношення ВИКЛАДАЧ показано на рис. 4.4.

ВИКЛАДАЧ

ПІБ	Посада	Оклад	Стаж	Д_Стаж	Каф	Предм	Група	ВидЗан
Іванов І. М.	викл.	5 000	5	1 000	25	СУБД	256	практ.
Іванов І. М.	викл.	5 000	5	1 000	25	ПЛ / 1	123	практ.
Петров М. І.	ст. викл.	8 000	7	1 000	25	СУБД	256	лекція
Петров М. І.	ст. викл.	8 000	7	1 000	25	Паскаль	256	практ.
Сидоров Н. Г.	викл.	5 000	10	1 500	25	ПЛ / 1	123	лекція
Сидоров Н. Г.	викл.	5 000	10	1 500	25	Паскаль	256	лекція
Єгоров В. В.	викл.	5 000	5	1 000	24	ПЕОМ	244	лекція

Рис. 4.4. **Початкове відношення ВИКЛАДАЧ**

Початкове відношення ВИКЛАДАЧ містить надлишкове дублювання даних, яке і є причиною аномалій редагування. Розрізняють надлишковість явну та неявну.

Явна надлишковість полягає в тому, що рядок із даними про викладачів, які проводять заняття в декількох групах, повторюють відповідну кількість разів. Наприклад, усі дані про Іванова повторюють двічі. Тому, якщо Іванов І. М. стане старшим викладачем, то цей факт має бути відображеним

в обох рядках. В іншому разі буде мати місце суперечність у даних, що є прикладом аномалії редагування, зумовленої явною надлишковістю даних у відношенні.

Неявна надлишковість є в однакових окладах у всіх викладачів і в однакових надбавках до окладу за однаковий стаж. Тому, якщо в разі зміни окладів за посаду із 5 000 на 5 100 це значення змінять у всіх викладачів, крім, наприклад, Сидорова, то база стане суперечливою. Це приклад аномалії редагування для варіанта з неявною надлишковістю.

Засобом усування надлишковості у відношеннях і, як наслідок, аномалій є нормалізація відношень, розгляньмо її більш детально.

4.2. Метод нормальних форм

Проєктування БД є одним з етапів життєвого циклу інформаційної системи. Основним завданням, що вирішують у процесі проєктування БД, є завдання нормалізації її відношень. Розглянутий далі метод нормальних форм є класичним методом проєктування реляційних БД. Цей метод засновано на фундаментальному понятті в теорії реляційних баз даних залежності між атрибутами відношень.

Розгляньмо основні види залежностей між атрибутами відношень: функціональні, транзитивні та багатозначні.

Поняття *функціональної залежності* є базовим, оскільки на його основі формулюють визначення всіх інших видів залежностей.

Атрибут B функціонально залежить від атрибута A, якщо кожному значенню A відповідає в точності одне значення B.

Математично функціональну залежність B від A позначають записом A→B. Це означає, що у всіх кортежів з однаковим значенням атрибута A атрибут B буде мати також одне й те саме значення. Зазначмо, що A й B можуть бути складеними, тобто складатися із двох і більше атрибутів.

На рис. 4.4 можна виділити функціональні відношення між атрибутами ПІБ→Каф, ПІБ→Посада, Посада→Оклад та ін. Наявність функціональної залежності у відношенні визначено природою речей, інформацію про яких подано кортежами відношення. У відношенні на рис. 4.4 ключ є складеним і складається з атрибутів ПІБ, Предмет, Група.

Якщо є функціональна залежність виду A → B і B→A, то між A і B є взаємно однозначна відповідність, або функціональна взаємозалежність.

Наявність функціональної взаємозалежності між атрибутами A і B позначмо як A↔B або B↔A.

Приклад. Нехай є деяке відношення, що містить два атрибути, функціонально залежні один від одного. Це серія, номер паспорта (N) та прізвище, ім'я та по батькові власника (ПІБ). Наявність функціональної залежності поля ПІБ від N означає не тільки той факт, що значення поля N однозначно визначає значення поля ПІБ, але й те, що одному й тому самому значенню поля N відповідає тільки єдине значення поля ПІБ. Зрозуміло, що в цьому разі діє й обернена функціональна залежність: кожному значенню поля ПІБ відповідає тільки одне значення поля N. У цьому прикладі передбачено, що ситуацію наявності повного збігу прізвищ, імен та по батькові двох людей виключено.

Якщо відношення є в нормальній формі, то всі неключові атрибути функціонально залежать від ключа з різним ступенем залежності.

Частковою функціональною залежністю називають залежність неключового атрибута від частини складеного ключа.

У цьому відношенні атрибут Посада є у функціональній залежності від атрибута ПІБ, що є частиною ключа. Тим самим атрибут Посада є в частковій залежності від ключа відношення.

Альтернативним варіантом є повна функціональна залежність неключового атрибута від усього складеного ключа. У нашому прикладі атрибут ВидЗан є в повній функціональної залежності від складеного ключа.

Атрибут C залежить від атрибута A транзитивно (є *транзитивна залежність*), якщо для атрибутів A, B, C виконують умови A→B і B→C, але оберненої залежності немає.

Між атрибутами може мати місце *багатозначна залежність*.

У відношенні R атрибут B багатозначно залежить від атрибута A, якщо кожному значенню A відповідає множина значень B, не пов'язаних іншими атрибутами з R.

Багатозначні залежності можуть бути "один-до-багатьох" (1:M), "багато-до-одного" (M:1) або "багато-до-багатьох" (M:M), що позначають, відповідно: A=>B, A<=B і A<=>B.

Наприклад, нехай викладач веде кілька навчальних предметів, а кожен цей предмет можуть вести декілька викладачів, тоді має місце залежність ПІБ⇔Предмет. Так, із рис. 4.4, видно, що викладач Іванов І. М. веде заняття із двох предметів, а навчальну дисципліну СУБД викладають два викладачі: Іванов І. М. і Петров М. І.

У загальному випадку між двома атрибутами одного відношення можуть бути залежності: 1:1, 1:M, M:1 і M:M. Оскільки залежність між атрибутами є причиною аномалій, намагаються розчленувати відношення із залежностями атрибутів на декілька відношень. У результаті утворюється сукупність зв'язаних відношень (таблиць) зі зв'язками виду 1:1, 1:M, M:1 і M:M. Зв'язки між таблицями відображають залежності між атрибутами різних відношень.

Два атрибути або більше називають *взаємно незалежними*, якщо жоден із цих атрибутів не є функціонально залежним від інших атрибутів.

Відсутність залежності атрибута A від атрибута B можна позначати так: A¬ → B. Випадок, якщо A¬ → B і B¬ → A, можна позначити A¬ =B.

Процес проєктування БД із використанням методу нормальних форм є ітераційним і полягає в послідовному переведенні відношень із першої нормальної форми до нормальної форми більш високого порядку за певними правилами. Кожна наступна нормальна форма обмежує певний тип функціональних залежностей, усуває відповідні аномалії під час виконання операцій над БД і зберігає властивості попередніх нормальних форм.

Виділяють таку послідовність нормальних форм:

- перша нормальна форма (1НФ);
- друга нормальна форма (2НФ);
- третя нормальна форма (3НФ);
- підсилена третя нормальна форма, або нормальна форма Бойса – Кодда (БКНФ);
- четверта нормальна форма (4НФ);
- п'ята нормальна форма (5НФ).

4.3. Перша нормальна форма

Відношення є в 1НФ, якщо всі його атрибути є простими (мають єдине значення). Початкове відношення побудовано таким способом, щоб воно було в 1НФ.

Переведення відношення в наступну нормальну форму здійснюють методом "декомпозиції без утрат". Така декомпозиція має забезпечити те, що запити (вибірка даних за умовою) до вихідного відношення й до відношень, що здобувають у результаті декомпозиції, дадуть однаковий результат.

Основною операцією методу є операція проєкції. Пояснімо її на прикладі. Початкове відношення ВИКЛАДАЧ, що використовують для ілюстрації методу, має складовою ключ ПІБ, Предм, Група та буде в 1НФ, оскільки всі його атрибути є простими.

У цьому відношенні можна виділити часткову залежність атрибутів Стаж, Д_Стаж, Каф, Посада, Оклад від ключа. Указані атрибути будуть у функціональній залежності від атрибута ПІБ, який є частиною складеного ключа.

Ця часткова залежність від ключа призводить до такого:

1. У відношенні наявне явне й неявне надлишкове дублювання даних, наприклад:

• повторення відомостей про стаж, посади й окладі викладачів, які проводять заняття в декількох групах і/або з різних предметів;

• повторення відомостей про оклади для однієї й тієї самої посади або про надбавки за однаковий стаж.

2. Наслідком надлишкового дублювання даних є проблема їхнього редагування. Наприклад, зміна посади у викладача Іванова І. М. потребує перегляду всіх кортежів відношення та внесення змін у ті з них, які містять відомості про цього викладача.

Частину надлишковості усувають під час переведення відношення у 2НФ.

4.4. Друга нормальна форма

Відношення є у 2НФ, якщо воно є в 1НФ і кожний неключовий атрибут функціонально повністю залежить від первинного ключа (складеного).

Для усунення часткової залежності й переведення відношення у 2НФ необхідно, використовуючи операцію проєкції, розкласти його на кілька відношень таким способом:

- побудувати проєкцію без атрибутів, які перебувають у частковій функціональної залежності від первинного ключа;
- побудувати проєкції на частини складеного первинного ключа й атрибут, що залежать від цих частин.

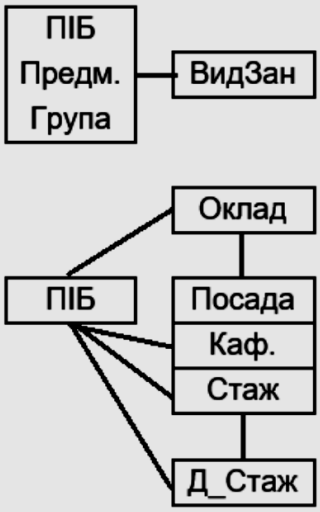
У результаті маємо два відношення R1 і R2 у 2НФ (рис. 4.5).

R1

ПІБ	Предм.	Група	ВидЗан
Іванов І. М.	СУБД	256	Практ.
Іванов І. М.	Java	123	Практ.
Петров М. І.	СУБД	256	Лекція
Петров М. І.	C#	256	Практ.
Сидоров Н. Г.	C#	123	Лекція
Сидоров Н. Г.	Java	256	Лекція
Єгоров В. В.	Java	244	Лекція

R2

ПІБ	Посада	Оклад	Стаж	Д_стаж	Каф
Іванов І. М.	виклад.	5 000	5	1 000	25
Петров М. І.	ст.викл.	8 000	7	1 000	25
Сидоров Н. Г.	виклад.	5 000	10	1 500	25
Єгоров В. В.	виклад.	5 000	5	1 000	24



а б

Рис. 4.5. **Відношення БД у 2НФ**

Щодо R1 первинний ключ є складеним і складається з атрибутів ПІБ, Предм, Група. Нагадаймо, що цей ключ щодо R1, визначений у припущенні, що кожен викладач в одній групі з одного предмета може або читати лекції, або проводити практичні заняття. Щодо R2 – ключ ПІБ.

Дослідження відношень R1 і R2 показує, що перехід до 2НФ дозволяє усунути явну надлишковість даних у табл. R2 – повторення рядків із відомостями про викладачів. У R2 як і раніше має місце неявне дублювання даних.

Для подальшого вдосконалення відношень необхідно перетворити його на третю нормальну форму.

4.5. Третя нормальна форма

Відношення є у 3НФ у тому й тільки тому разі, якщо всі неключові атрибути відношення взаємно незалежні та повністю залежать від первинного ключа.

Взаємна незалежність атрибутів означає, що немає будь-якої залежності між атрибутами відношення, зокрема й транзитивної залежності між ними.

Якщо щодо R1 транзитивних залежностей немає, то у відношенні R2 вони є такі:

ПІБ → Посада → Оклад;

ПІБ → Оклад → Посада;

ПІБ → Стаж → Д_Стаж.

Транзитивні залежності також породжують надлишкове дублювання інформації у відношенні. Для усунення їх виконують операцію проєкції на атрибути, які є причиною транзитивних залежностей (рис. 4.6).

а б

Рис. 4.6. **Відношення БД у 3НФ**

Графічно ці відносини показано на рис. 4.6б. Зауважмо, що відношення R2 можна перетворити по-іншому, а саме: у відношенні R3 замість атрибута Посада взяти атрибут Оклад.

На практиці побудова 3НФ здебільшого є достатньою. Дійсно, приведення відношення до 3НФ у нашому прикладі, привело до усунення надлишкового дублювання.

Якщо у плані є залежність атрибутів складеного ключа від неключових атрибутів, то необхідно перейти до підсиленої 3НФ.

4.6. Четверта нормальна форма

Розгляньмо приклад нового відношення ПРОЄКТИ, схема якого має такий вигляд: ПРОЄКТИ (Номер_проєкту, Код_співробітника, Завдання_співробітника). Первинним ключем відношення є вся сукупність атрибутів: Номер_проєкту, Код_співробітника та Завдання_співробітника.

Для кожного проєкту є список кодів співробітників-виконавців, а також список завдань, передбачених кожним проєктом. Співробітники можуть брати участь у декількох проєктах, і різні проєкти можуть містити однакові завдання. Передбачено, що кожен співробітник виконує всі завдання із цього проєкту.

За такого формулювання питання єдиним можливим ключем відношення є складений атрибут Номір_проєкту, Код_співробітника, Завдання_співробітника. Він, природно, і став первинним ключем відношення. Звідси випливає, що відношення ПРОЄКТИ є у 3НФ. Нехай вихідна інформація в цьому відношенні має такий вигляд (рис. 4.7):

ПРОЄКТИ

Номер_проєкту	Код_співробітника	Завдання_співробітника
001	05	1
001	05	2
001	05	3
004	03	2
004	05	1
004	05	2
007	06	1

Рис. 4.7. **Вихідна інформація відношення ПРОЄКТИ**

Головний недолік відношення ПРОЄКТИ полягає в тому, що в разі додавання/відсторонення від проєкту співробітника потрібно додавати/вилучати з відношення стільки кортежів, скільки завдань є у проєкті. Унесення або вилучення одного факту про деякого співробітника потребує серії елементарних операцій через дублювання значень у кортежі.

Звідси виникають такі запитання: "Навіщо зберігати в кортежі повторювані значення кодів співробітників?" "Чи потрібно перелічувати всі завдання за кожним проєктом та ще для кожного співробітника-виконавця цього проєкту?" "Чи не можна інформацію про прив'язування завдань

до проєкту помістити в окрему таблицю та вилучити повторення в основній таблиці?"

Зауважмо, що непряма ознака аномалії, як і раніше, – дублювання інформації в таблиці. Висловімо припущення, що причиною аномалії є наявність деякої залежності меж атрибутами відношення (як побачимо далі – багатозначною залежністю).

Дійсно, щодо ПРОЄКТИ наявні дві багатозначні залежності:

Номер_проєкту => Код_співробітника;

Номер_проєкту => Завдання_співробітника.

У довільному відношенні R (A, B, C) може одночасно бути багатозначна залежність A => B та A => C. Цю обставину позначають як A => B | C.

Під проєктуванням без утрат розуміють такий спосіб декомпозиції відношення, за якого початкове відношення повністю відновлюють шляхом з'єднання визначених відношень. Пояснімо проєктування без утрат на прикладі. Нехай є найпростіше відношення R (A, B, C), що має такий вигляд (рис. 4.8):

A	B	3
К	15	1
К	15	2
Л	10	1
М	20	1
М	20	2
М	20	3

Рис. 4.8. **Проєктування без утрат**

Побудуймо проєкції R1 і R2 на атрибути A, B і A, C, відповідно. Вони будуть мати такий вигляд (рис. 4.9):

R1

A	B
К	15
Л	10
М	20

R2

A	C
К	1
К	2
Л	1
М	1
М	2
М	3

Рис. 4.9. **Проєкції R1 і R2 на атрибути A, B і A, C**

Неважко побачити, що зв'язування R1 (A, B) і R2 (A, C) до точності породжує вихідне відношення R (A, B, C). Щодо R немає зайвих кортежів, немає і втрат.

Визначення четвертої нормальної форми. Відношення R є в четвертій нормальній формі (4НФ) у тому й тільки в тому разі, якщо є багатозначна залежність A => B, а всі фундаментальні атрибути R функціонально залежать від A.

Наведене раніше відношення ПРОЄКТИ можна подати у вигляді двох відношень: ПРОЄКТИ – СПІВРОБІТНИКИ та ПРОЄКТИ – ЗАВДАННЯ. Структура цих відношень і вміст відповідних таблиць має такий вигляд (рис. 4.10):

ПРОЄКТИ – СПІВРОБІТНИКИ

Номер_ проєкту	Код_ співробітника
001	05
004	02
004	03
004	05
007	06

а

ПРОЄКТИ – ЗАВДАННЯ

Номер_ проєкту	Завдання_ співробітника
001	1
001	2
001	3
004	1
004	2
007	1

б

Рис. 4.10. **Відношення ПРОЄКТИ – СПІВРОБІТНИКИ (а) та ПРОЄКТИ – ЗАВДАННЯ (б)**

Як легко побачити, обидва ці відношення будуть у 4НФ і вільними від помічених недоліків. Дублювання значень атрибутів кодів співробітників немає.

У загальному випадку не будь-яке відношення можна відновити до вихідного. У цьому разі відновлення можливе тому, що кожен співробітник виконував усі завдання за проєктом (саме це вкладають у принцип 1:M з'єднання відношень). Самі ж співробітники брали участь у декількох проєктах, і різні проєкти могли містити однакові завдання.

4.7. П'ята нормальна форма

Результатом нормалізації всіх попередніх схем відношень були два нові відношення. Іноді це зробити не вдається, або здобуті відношення

свідомо мають небажані властивості. У цьому разі виконують декомпозицію вихідного відношення на відношення, кількість яких перевищує два.

Розгляньмо відношення СПІВРОБІТНИКИ – ВІДДІЛИ – ПРОЄКТИ. Первинний ключ відношення містить усі атрибути: Код_співробітника, Код_відділу і Номер_проєкту. Нехай у цьому відношенні один співробітник може працювати в декількох відділах, причому в кожному відділі він може брати участь у декількох проєктах. В одному відділі можуть працювати декілька співробітників, але кожен проєкт виконує тільки один співробітник. Функціональних і багатозначних залежностей між атрибутами немає (рис. 4.11).

СПІВРОБІТНИКИ – ВІДДІЛИ – ПРОЄКТИ

Код_співробітника	Код_відділу	Номер_проєкту
01	РД	036
02	АТ	004
03	УП	004
04	АТ	019
05	ЛЗ	001
06	ЛЗ	004
07	УП	007
08	ВЦ	013
09	ВЦ	014
10	СЖ	013

Рис. 4.11. **Відношення СПІВРОБІТНИКИ – ВІДДІЛИ – ПРОЄКТИ**

З огляду на структуру відношення СПІВРОБІТНИКИ – ВІДДІЛИ – ПРОЄКТИ, можна зробити висновок, що воно є у формі. А втім, у відношенні можуть бути аномалії, пов'язані з можливістю повторення значень атрибутів у декількох кортежах. Наприклад, те, що співробітник може працювати в декількох відділах, у разі звільнення працівника потребує пошуку й подальшого видалення з вихідної таблиці декількох записів.

Визначення п'ятої нормальної форми. Відношення R є у 5НФ у тому й тільки тому разі, якщо будь-яка залежність з'єднання в R випливає з наявності деякого можливого ключа в R.

Утворити складові атрибути відношення СПІВРОБІТНИКИ – ВІДДІЛИ – ПРОЄКТИ:

СО = {Код_співробітника, Код_відділу};

СП = {Код_співробітника, Номер_проєкту};

ОП = {Код_відділу, Номер_проєкту}.

Покажімо, що якщо відношення СПІВРОБІТНИКИ – ВІДДІЛИ – ПРОЄКТИ спроєктувати на складові атрибути СО, СП і ОП, то поєднання цих проєкцій дає початкове відношення. Це означає, що в цьому відношенні є залежність з'єднання *(СО, СП, ОП). Проєкції на складові атрибути назвімо, відповідно, СПІВРОБІТНИКИ – ВІДДІЛИ, СПІВРОБІТНИКИ – ПРОЄКТИ та ВІДДІЛИ – ПРОЄКТИ.

Раніше виконували з'єднання двох проєкцій і відразу здобували шуканий результат. Для відновлення відношення із трьох (або декількох) проєкцій треба визначити всі попарні з'єднання (тому що інформації про те, яке з них найкраще, немає), над якими потім виконати операцію перетину множин. Перевіримо, чи це так (рис. 4.12).

СПІВРОБІТНИКИ – ВІДДІЛИ

Код_ співробітника	Код_ відділу
01	РД
02	АТ
03	УП
04	АТ
05	ЛЗ
05	ЛЗ
06	УП
08	ВЦ
09	ВЦ

СПІВРОБІТНИКИ – ПРОЄКТИ

Код_ співробітника	Номер_ проєкту
01	036
02	004
03	004
04	019
05	001
05	004
06	007
08	013
09	014

ВІДДІЛИ – ПРОЄКТИ

Код_ відділу	Номер_ проєкту
АТ	004
АТ	019
ВЦ	013
ВЦ	014
ЛЗ	001
ЛЗ	004
СЖ	013
РД	036
УП	004

Рис. 4.12. **Вихідні залежності**

Припустімо, необхідно дізнатися, де й які проєкти виконує співробітник із кодом 02. Для цього у відношенні СПІВРОБІТНИКИ – ВІДДІЛИ визначмо, що співробітник із кодом 02 працює у відділі АТ, а з відношення ВІДДІЛИ – ПРОЄКТИ – що у відділі АТ виконують проєкти 004 і 019. А це означає, що співробітник 02 має виконувати проєкти 004 і 019. На жаль, інформація про те, що співробітник із кодом 02 виконує проєкт 019, є хибною (див. вихідне відношення СПІВРОБІТНИКИ – ВІДДІЛИ – ПРОЄКТИ).

Здобудьмо попарні з'єднання трьох наведених раніше відношень, (рис. 4.13). Неважко побачити, що перетин усіх відношень дає вихідне відношення СПІВРОБІТНИКИ – ВІДДІЛИ – ПРОЄКТИ.

* (СВ, СП)

Код_ співробітника	Код_ відділу	Номер_ проєкту
01	РД	036
02	АТ	004
03	УП	004
04	АТ	019
05	ЛЗ	001
05	ЛЗ	004
06	УП	007
08	ВЦ	013
09	ВЦ	014

* (СВ, ВП)

Код_ співробітника	Код_ відділу	Номер_ проєкту
01	РД	036
02	АТ	004
02	АТ	019
03	УП	004
03	УП	007
04	АТ	004
04	АТ	019
05	ЛЗ	001
05	ЛЗ	004
06	УП	004
06	УП	007
08	ВЦ	013
08	ВЦ	014
09	ВЦ	013
09	ВЦ	014

* (СП, ВП)

Код_ співробітника	Код_ відділу	Номер_ проєкту
01	РД	036
02	АТ	004
02	ЛЗ	004
02	УП	004
03	АТ	004
03	ЛЗ	004
03	УП	004
04	АТ	019
05	ЛЗ	001
05	АТ	004
05	ЛЗ	004
05	УП	004
06	УП	004
06	УП	007
08	ВЦ	013
08	ВЦ	014
09	ВЦ	013
09	ВЦ	014

Рис. 4.13. **Попарні з'єднання відношень СВ, СП і ВП**

Відношення СПІВРОБІТНИКИ – ВІДДІЛИ, СПІВРОБІТНИКИ – ПРОЄКТИ та ВІДДІЛИ – ПРОЄКТИ є у 5НФ. Ця форма є останньою з відомих. Умови її досягнення є досить нетривіальними і тому її майже не використовують на практиці. Крім того, вона має певні недоліки.

Такі суперечності можна усунути тільки шляхом спільного розгляду всіх проєкцій основного відношення. Саме тому після з'єднання проєкцій і виконували операцію їхнього перетину.

На практиці зазвичай обмежуються структурою БД, відповідною 3НФ. Тому процес нормалізації, уставлений методом нормальних форм, передбачає послідовне видалення з вихідного відношення таких міжатрибутних залежностей:

- часткових залежностей неключових атрибутів від ключа (задоволення вимог 2НФ);
- транзитивних залежностей ключових атрибутів від ключа (задоволення вимог 3НФ);

залежність ключів (атрибутів складених ключів) від неключових атрибутів.

4.8. Забезпечення цілісності

Під **цілісністю** розуміють властивість бази даних, яка означає, що вона містить повну, несуперечливу предметної галузі інформації. Розрізняють фізичну та логічну цілісність.

Фізична цілісність означає наявність фізичного доступу до даних і те, що дані не є втраченими.

Логічна цілісність означає, що немає логічних помилок у базі даних, до яких належать порушення структури БД або її об'єктів, видалення або зміна встановлених зв'язків між об'єктами тощо.

Надалі мову будемо вести про логічну цілісність.

Підтримання цілісності БД містить перевірку цілісності та її відновлення в разі виявлення суперечностей у базі. Цілісний стан БД задають за допомогою обмежень цілісності у вигляді умов, яким мають задовольняти в базі даних.

Серед обмежень цілісності можна виділити два основні типи обмежень: обмеження значень атрибутів відношень і структурні обмеження на кортежі відношень.

Прикладом *обмежень значень атрибутів відношень* є вимоги неприпустимості порожніх або повторюваних значень в атрибутах, а також

контроль за належністю значень атрибутів заданому діапазонові. Так, у записах відношень про кадри значення атрибута Дата_народження не можуть перевищувати значення атрибута Дата_приймання.

Найбільш гнучким засобом реалізації контролю за значеннями атрибутів є збережені процедури та тригери, наявні в деяких СУБД.

Структурні обмеження визначають вимоги до цілісності сутності й цілісності посилань. Кожному екземплярові сутності, поданому у відношенні, відповідає тільки один його кортеж. Вимога до цілісності сутностей полягає в тому, що будь-який кортеж відношення має бути відмінним від будь-якого іншого кортежу цього відношення, тобто, інакше кажучи, будь-яке відношення має володіти первинним ключем.

Формулювання вимоги до цілісності посилань тісно пов'язано з поняттям зовнішнього ключа. Нагадаймо, що зовнішні ключі слугують для зв'язку відношень (таблиць БД) між собою. Водночас атрибут одного відношення називають *зовнішнім ключем* цього відношення, якщо він є первинним ключем іншого відношення (дочірнього). Кажуть, що відношення, у якому визначено зовнішній ключ, посилається на відношення, у якому цей самий атрибут є первинним ключем.

Вимога до цілісності посилань полягає в тому, що для кожного значення зовнішнього ключа батьківської таблиці має бути рядок у дочірній таблиці з таким самим значенням первинного ключа. Наприклад, якщо у відношенні R1 (рис. 4.14) містяться відомості про співробітників кафедри, а атрибут цього відношення має бути первинним ключем відношення R2, то в цьому відношенні для кожної посади з R1 має бути рядок із відповідним їй окладом.

Рис. 4.14. **Зв'язок відношень за допомогою зовнішнього ключа**

У багатьох сучасних СУБД є засоби забезпечення контролю за цілісністю БД.

Контрольні запитання

1. Назвіть підходи до проєктування структур даних.
2. У чому полягає надлишкове й ненадлишкове дублювання даних?
3. Назвіть та охарактеризуйте основні види аномалій.
4. Як формують вихідне відношення під час проєктування БД?
5. Наведіть приклади явної й неявної надлишковаості.
6. Назвіть основні види залежностей між атрибутами відношень.
7. Наведіть приклади функціональної й частково функціональної залежностей.
8. Наведіть приклади відношень із залежними атрибутами.
9. Охарактеризуйте нормальні форми.
10. Дайте визначення першої нормальної форми.
11. Дайте визначення другої нормальної форми.
12. Дайте визначення третьої нормальної форми.
13. Дайте визначення фізичної та логічної цілісностей БД.
14. Наведіть приклади обмежень значень і структурних обмежень.
15. Поясніть поняття зовнішнього та первинного ключів таблиць.

Рекомендована література: [1; 5; 6].

Розділ 2
Особливості програмного забезпечення систем баз даних

5. Мови запитів СУБД

Мета – визначення понять "вибірка", "групування записів", "об'єднання таблиць" та їхніх основних складових. Розуміння теоретичних мов запитів та можливостей маніпулювання даними під час опису запитів.

Основні питання

5.1. Теоретичні мови запитів.
5.2. Мова структурованих запитів SQL.
5.3. Вибірка даних – оператор SELECT.
5.4. Здобуття підсумкових значень.
5.5. Об'єднання таблиць.
5.6. Групування записів і функція COUNT().
5.7. Редагування, оновлення та видалення даних.

Ключові слова: мова запитів, реляційна алгебра, вибірка, оператор, групування, маніпуляція даними, транзакція.

5.1. Теоретичні мови запитів

У реляційних СУБД для виконання операцій над відношеннями використовують дві групи мов, які мають як свою математичну основу теоретичні мови запитів:

- реляційну алгебру;
- реляційне обчислення.

У *реляційній алгебрі* операнди й результати всіх дій є відношеннями. Мови реляційної алгебри є процедурними, оскільки відношення, що є результатом запиту до реляційної БД, обчислюють під час виконання послідовності операторів. Оператори складаються з операндів, у ролі яких виступають відношення, і реляційних операцій. Результатом реляційної операції є відношення.

Мови *реляційних обчислень*, на відміну від реляційної алгебри, є непроцедурними й дозволяють висловлювати запити за допомогою предиката першого порядку (висловлювання у вигляді функції), якому мають задовольняти кортежі або домени відношення.

Збережені в базі дані можна опрацьовувати вручну, послідовно переглядаючи й редагуючи дані в таблицях, за допомогою наявних у СУБД відповідних засобів. Множинне опрацювання даних дозволяє одночасно вводити, редагувати та видаляти множину записів, а також вибирати дані з таблиць.

Запит є спеціальним чином описана вимога, що визначає склад вироблених над БД операцій із вибірки, видалення або модифікацій збережених даних.

Для підготовки запитів за допомогою різних СУБД найчастіше використовують дві основні мови опису запитів:

QBE (Query By Example) – мову запитів за зразком;

SQL (Structured Query Language) – структуровану мову запитів.

За можливостями маніпулювання даними під час опису запитів зазначені мови є практично еквівалентними. Головна різниця між ними полягає у способі формування запитів: мова QBE передбачає ручне або візуальне формування запиту, тоді як використання SQL означає програмування запиту.

5.2. Мова структурованих запитів SQL

Структуровану мову запитів SQL засновано на реляційному обчислюванні зі змінними кортежами. Мова має кілька стандартів, найбільш поширеними з яких є SQL-89 і SQL-92.

Мова SQL становить набір операторів. Мова SQL є алгоритмічною та належить до мов, що інтерпретують. Це означає, що кожен оператор мови SQL виконують безпосередньо під час його появи.

У зв'язку із цим SQL автономно не використовують, зазвичай його занурено в середовище вбудованої мови програмування СУБД (наприклад, FoxPro СУБД Visual FoxPro, ObjectPAL СУБД Paradox, Visual Basic for Applications СУБД Access).

Усю множину операторів SQL розподіляють на підмножини, які вважають підмовою SQL. У кожній підмножині використовують певні оператори, задані їхніми ключовими словами. Підмовою SQL є такі групи операторів:

Оператори визначення даних (*Data Definition Language, DDL*):

• CREATE – створює об'єкт БД (саму базу, таблицю, подання, користувача);

• ALTER – змінює об'єкт;

• DROP – видаляє об'єкт.

Оператори маніпуляції даними (*Data Manipulation Language, DML*):

• SELECT – зчитує дані, що задовольняють заданим вимогам;

• INSERT – додає нові дані;

• UPDATE – змінює наявні дані;

• DELETE – видаляє дані.

Оператори визначення доступу до даних (*Data Control Language, DCL*):

• GRANT – надає користувачеві (групі) дозвіл на визначені операції з об'єктом;

• REVOKE – відкликає раніше видані дозволи;

• DENY – задає заборону, яка має пріоритет над вирішенням.

Оператори управління транзакціями (*Transaction Control Language, TCL*):

• COMMIT – застосовує транзакцію;

• ROLLBACK – скасовує всі зміни, зроблені в контексті поточної транзакції;

• SAVEPOINT – розподіляє транзакцію на більш дрібні ділянки.

Розрізняють два основні методи використання вбудованої мови SQL: статичний і динамічний.

За *статичного методу* використання мови в тексті програми є викликами функцій мови SQL, які жорстко додають у виконуваний модуль після компіляції. Зміни у функції, що викликають, можуть бути на рівні окремих параметрів за допомогою змінних мови програмування.

За *динамічного методу* використання мови передбачає динамічну побудову викликів SQL-функцій та інтерпретацію цих викликів, наприклад, звернення до даних лише у віддаленій базі в ході виконання програми. Динамічний метод зазвичай застосовують у разі, якщо в застосунку заздалегідь невідомий вид SQL-виклику і його будують у діалозі з користувачем.

Основним призначенням мови SQL є підготовка та виконання запитів. У результаті вибірки даних з однієї або декількох таблиць може бути визначено множину записів, названу поданням.

Подання, насправді, є таблицею, що формують, унаслідок виконання запиту.

Можна сказати, що воно є різновидом зберігання запиту. За однією таблицею можна побудувати кілька подань. Саме подання описують шляхом указівки ідентифікатора подання й запиту, який має бути виконаним для його здобуття.

Розгляньмо формат та основні можливості найважливіших операторів. Несуттєві операнди й елементи синтаксису (наприклад, прийняте в багатьох системах програмування правило ставити ";" у кінці оператора) будемо опускати.

1. Оператор створення таблиці має формат такого вигляду:

CREATE TABLE <назва таблиці>
(<Назва стовпця> <тип даних> [NOT NULL]
[<Назва стовпця> <тип даних> [NOT NULL]], ...)

Обов'язковими операндами оператора є назва створюваної таблиці й назва хоча б одного стовпця із зазначенням типу даних, що зберігають у цьому стовпці.

Під час створення таблиці для окремих полів можуть зазначати деякі додаткові правила контролю за значеннями, що вводять у них. Конструкція NOT NULL (не порожнє) слугує саме таким цілям і для стовпця таблиці означає, що в цьому стовпці має бути визначено значення.

2. Оператор зміни структури таблиці має формат такого вигляду:

ALTER TABLE <назва таблиці>
({ADD, MODIFY, DROP} <назва стовпця> <тип даних> [NOT NULL]
{ADD, MODIFY, DROP} <назва стовпця> <тип даних> [NOT NULL], ...)

Зміна структури таблиці може полягати в додаванні (ADD), зміні (MODIFY) або видаленні (DROP) одного або декількох стовпців таблиці. Правила запису оператора ALTER TABLE такі самі, як і оператора CREATE TABLE. Під час видалення стовпця вказувати <тип даних> не потрібно.

3. Оператор видалення таблиці має формат такого вигляду:

DROP TABLE <назва таблиці>

Оператор дозволяє видалити наявну таблицю.

4. Оператор створення індексу має формат такого вигляду:

CREATE [UNIQUE] INDEX <назва індексу>
ON <назва таблиці> (<назва стовпця> [ASC | DESC, <назва стовпця> [ASC | DESC] ...)

Оператор дозволяє створити індекс для одного або декількох стовпців заданої таблиці, із метою прискорення виконання запитальних і пошукових операцій із таблицею. Для однієї таблиці можна створити кілька індексів.

Задавши необов'язкову опцію UNIQUE, можна забезпечити унікальність значень. Насправді, створення індексу із зазначенням ознаки UNIQUE означає визначення ключа у створеній раніше таблиці.

Під час створення індексу можна задати порядок автоматичного сортування значень у стовпцях – у порядку зростання ASC (за замовчуванням), або в порядку убування DESC. Для різних стовпців можна задавати різний порядок сортування.

Приклад. Створення індексу.

Нехай для табл. EMP, що має поля: NAME (ім'я), SAL (зарплата), MGR (керівник) і DEPT (відділ), потрібно створити індекс main_indx для сортування імен в алфавітному порядку та зменшенням розмірів зарплати. Оператор створення індексу буде мати такий вигляд:

CREATE INDEX main_indx
ON EMP (NAME, SAL DESC)

5. Оператор видалення індексу має формат такого вигляду:

DROP INDEX <назва індексу>

Цей оператор дозволяє видаляти створений раніше індекс із відповідною назвою. Так, наприклад, для знищення індексу maifi_indx до табл. EMP достатньо записати оператор DROP INDEX main_indx.

6. Оператор створення подання має формат такого вигляду:

```
CREATE VIEW <назва подання>
[(<Назва стовпця> [, <Назва стовпця>] ...)]
AS <оператор SELECT>
```

Цей оператор дозволяє створити подання. Якщо назву стовпців у поданні не вказують, то будуть використовувати назви стовпців із запиту, описуваного відповідним оператором SELECT.

Приклад. Створення подання.

Нехай таблиця **companies** є описом виробників товарів із полями: comp_id (ідентифікатор компанії), comp_name (назва організації), comp_address (адреса) і phone (телефон).

А таблиця **goods** вироблених товарів із полями: type (вид товару), comp_id (ідентифікатор компанії), name (назва товару) і price (ціна товару). Таблиці пов'язані між собою за полем comp_id.

Потрібно створити подання **repr** із короткою інформацією про товари та їхніх виробників: вид товару, назва виробника й ціна товару. Оператор визначення подання може мати такий вигляд:

```
CREATE VIEW repr
AS SELECT goods.type, companies.comp_name, goods.price
FROM goods, companies
WHERE goods.comp_id = companies.comp_id
```

7. Оператор видалення подання має формат такого вигляду:

```
DROP VIEW <назва подання>
```

Оператор дозволяє видалити створене раніше подання. Зауважмо, що під час видалення таблиці подання, які беруть участь у запиті, видаленню не підлягають. Видалення подання repr виконує оператор такого виду:

```
DROP VIEW repr
```

8. Оператор вибірки записів має формат такого вигляду:

SELECT [ALL | DISTINCT]
<Список даних>
FROM <список таблиць>
[WHERE <умова вибірки>]
[GROUP BY <назва стовпця> [, <назва стовпця>] ...]
[HAVING <умова пошуку>]
[ORDER BY <специфікація> [, <специфікація>] ...]

Це найбільш важливий оператор з усіх операторів SQL. Функціональні можливості його є величезними.

Оператор SELECT дозволяє робити вибірку й обчислення над даними з однієї або декількох таблиць. Результатом виконання оператора є відповідна таблиця, яка може мати (ALL), або не мати (DISTINCT) повторюваних рядків. За замовчуванням у відповідну таблицю додають усі рядки, зокрема й повторювані. У відборі даних беруть участь записи однієї або декількох таблиць, перелічених у списку, операнда FROM.

Список даних може містити імена стовпців, що беруть участь у запиті, а також вирази над стовпцями. У найпростішому випадку у виразах можна записувати назви стовпців, знаки арифметичних операцій (+, −, *, /), константи та круглі дужки. Якщо у списку даних записано вираз, то разом із вибіркою даних виконують обчислення, результати котрого потрапляють у новий (створюваний) стовпець відповідної таблиці.

Під час використання у списках даних назв стовпців декількох таблиць для вказівки належності стовпця деякій таблиці застосовують конструкцію виду: <назва таблиці>.<назва стовпця>.

Операнд WHERE задає умови, яким мають задовольняти записи в результативній таблиці. Його елементами можуть бути назви стовпців, операції порівняння, арифметичні операції, логічні зв'язки (І, АБО, НІ), дужки, спеціальні функції LIKE, NULL, IN тощо.

Операнд GROUP BY дозволяє виділяти в результативній множині записів групи. Групою є записи з однаковими значеннями у стовпцях, перелічених за ключовими словами GROUP BY. Виділення груп потрібне для використання в логічних виразах операндів WHERE і HAVING, а також для виконання операцій (обчислень) над групами.

У логічних та арифметичних виразах можна використовувати такі групові операції (функції): AVG (середнє значення у групі), MAX (максимальне значення у групі), MIN (мінімальне значення у групі), SUM (сума значень у групі), COUNT (кількість значень у групі).

Операнд HAVING діє спільно з операндом GROUP BY і його використовують для додаткової селекції записів під час визначення груп. Правила запису <умови пошуку> є аналогічними правилам формулювання <умови вибірки> операнда WHERE.

Операнд ORDER BY задає порядок сортування результативного відношення. Зазвичай кожна <специфікація> є аналогічною відповідній конструкції оператора CREATE INDEX і становить пару виду: <назва стовпця> [ACS | DESC].

Приклад. Вибір записів.

Для табл. EMP, що має поля: NAME (ім'я), SAL (зарплата), MGR (керівник) і DEPT (відділ), потрібно вивести імена співробітників і розмір їхньої зарплати, збільшений на 100 одиниць. Оператор вибору можна записати таким чином:

```
SELECT name, sal + 100
FROM emp
```

Приклад. Вибір з умовою.

Вивести назви таких відділів табл. EMP, у яких наразі відсутні керівники. Оператор SELECT для цього запиту можна записати так:

```
SELECT dept
FROM emp
WHERE mgr is NULL
```

9. Оператор зміни записів має формат такого вигляду:

```
UPDATE <назва таблиці>
SET <назва стовпця> = {<вираз>, NULL}
[, SET <назва стовпця> = {<вираз>, NULL} ...]
[WHERE <умова>]
```

Виконання оператора UPDATE полягає у зміні значень у визначених операндом SET клітинках таблиці для тих записів, які відповідають умові, заданій операндом WHERE.

Нові значення полів у записах можуть бути порожніми (NULL) або їх визначають, відповідно до арифметичного виразу. Правила запису арифметичних і логічних виразів є аналогічними відповідним правилам оператора SELECT.

Приклад. Зміна записів.

Нехай необхідно збільшити на 500 одиниць зарплату тим службовцям, які отримують не більше ніж 6 000 (за табл. OMP). Запит, сформульований за допомогою оператора SELECT, може мати такий вигляд:

```
UPDATE emp
SET sal = +500
WHERE sal <= 6 000
```

10. Оператор уставлення нових записів має формати таких двох видів:

```
INSERT INTO <назва таблиці>
[(<Список стовпців>)]
VALUES (<список значень>)
і
INSERT INTO <назва таблиці>
[(<Список стовпців>)]
<Пропозиція SELECT>
```

У першому форматі оператор INSERT призначено для введення нових записів із заданими значеннями у стовпцях. Порядок переліку стовпців має відповідати порядку значень, перелічених у списку операнда VALUES. Якщо <список стовпців> опущено, то у <списку значень> мають бути переліченими всі значення в порядку стовпців структури таблиці.

У другому форматі оператор INSERT призначено для введення в таблицю нових рядків, відібраних з іншої таблиці за допомогою пропозиції SELECT.

Приклад. Уведення записів.

Увести в табл. EMP запис про нового співробітника. Для цього можна записати оператор такого вигляду:

```
INSERT INTO emp
VALUES ("Ivanov", 7 500, "Lee", "cosmetics")
```

11. Оператор видалення записів має формат такого вигляду:

```
DELETE FROM <назва таблиці>
[WHERE <умова>]
```

Результатом виконання оператора DELETE є видалення із зазначеної таблиці рядків, які задовольняють умову, визначену операндом WHERE. Якщо необов'язковий операнд WHERE опущено, тобто умови відбору видалених записів немає, видаленню підлягають усі записи таблиці.

Приклад. Видалення записів.

У зв'язку з ліквідацією відділу іграшок (toy), потрібно видалити з табл. EMP усіх співробітників цього відділу. Оператор DELETE для цього завдання буде мати такий вигляд:

```
DELETE FROM emp
WHERE dept = "toy"
```

Мова SQL є гібридом реляційної алгебри та реляційного обчислення. У ній є елементи алгебри (оператор об'єднання UNION) та обчислення (квантор наявності EXISTS). Крім того, мова SQL має реляційну повноту.

5.3. Вибірка даних – оператор SELECT

Припустімо є БД forum, у якій є три таблиці: users (користувачі), topics (теми) і posts (повідомлення). І ми хочемо подивитися, які дані в них містяться. Для цього в SQL наявний оператор *SELECT*. Синтаксис його виконання такий:

```
SELECT що_вибрати FROM звідки_вибрати
```

Замість "що_вибрати", маємо вказати або назву стовпця, значення якого хочемо побачити, або назви кількох стовпців через кому, або символ

зірочки (*), що означає вибір усіх стовпців таблиці. Замість "звідки_вибрати" слід указати назву таблиці.

Наприклад, нас цікавлять усі стовпці з табл. users:

```
SELECT * FROM users
```

Припустімо, що ми хочемо подивитися тільки стовпець id_user. Для цього в запиті вкажемо назву цього стовпця:

```
SELECT id_user FROM users
```

Ну, а якщо ми захочемо подивитися, наприклад, прізвища та e-mail наших користувачів, то перелічимо стовпці, які цікавлять через кому:

```
SELECT name, email FROM users
```

Що якщо ми хочемо, щоб наші дані виводили, наприклад, за алфавітом. Для цього у SQL є ключове слово *ORDER BY,* після котрого вказують назву стовпця, по якому буде відбуватися сортування. Синтаксис такий:

```
SELECT назва_стовпця FROM назва_таблиці
ORDER BY назва_стовпця_сортування
```

За замовчуванням сортування йде за зростанням, але це можна змінити, додавши ключове слово *DESC.* Тепер наші дані будуть відсортованими в порядку спадання.

Слід зазначити, що вертикальна вибірка може містити дублікати рядків у тому разі, якщо вона не містить потенційного ключа, який однозначно визначає запис. Якщо потрібно здобути тільки унікальні рядки, то можна використовувати ключове слово *DISTINCT*:

Якщо необхідно виконати вибірку з умовою, тоді використовують таку конструкцію:

```
SELECT назва_стовпця FROM назва_таблиці WHERE умова
```

Для нашого прикладу умовою є ідентифікатор користувача, тобто нам потрібні тільки ті рядки, у стовпці id_author яких буде 4:

```
SELECT * FROM topics WHERE id_author = 4
```

Або ми хочемо дізнатися, хто створив тему "фітнес":

SELECT * FROM topics WHERE topic_name ='фітнес'

Звичайно, було б зручніше, щоб замість id автора, виводили його ім'я, але імена зберігають в іншій таблиці. Далі ми дізнаємося, як вибирати дані з декількох таблиць. А поки дізнаємося, які умови можна задавати, використовуючи ключове слово *WHERE*.

= (Дорівнює)	Відбирають значення, що дорівнюють зазначеному, приклад: SELECT * FROM topics WHERE id_author = 4
> (Більше)	Відбирають значення, більші від зазначеного, приклад: SELECT * FROM topics WHERE id_author > 2
< (Менше)	Відбирають значення, менші від зазначеного, приклад: SELECT * FROM topics WHERE id_author < 3
>= (Більше або дорівнює)	Відбирають значення більші та такі, що дорівнюють зазначеному, приклад: SELECT * FROM topics WHERE id_author >= 2
<= (Менше або дорівнює)	Відбирають значення менші та такі, що дорівнюють зазначеному, приклад: SELECT * FROM topics WHERE id_author <= 3

!= (Не дорівнює)	Відбирають значення, що не дорівнюють зазначеному, приклад: SELECT * FROM topics WHERE id_author != 2
IS NOT NULL	Відбирають рядки, які мають значення в зазначеному полі, приклад: SELECT * FROM topics WHERE id_author IS NOT NULL
IS NULL	Відбирають рядки, які не мають значення в зазначеному полі, приклад: SELECT * FROM topics WHERE id_author IS NULL
BETWEEN (між)	Відбирають значення, що містяться між зазначеними, приклад: SELECT * FROM topics WHERE id_author BETWEEN 1 AND 3
IN	Відбирають значення, що відповідають зазначеним, приклад: SELECT * FROM topics WHERE id_author IN (1, 4)
NOT IN	Відбирають значення, крім зазначених, приклад: SELECT * FROM topics WHERE id_author NOT IN (1, 4)

LIKE (відповідність)	Відбирають значення, відповідні зразкам, приклад: SELECT * FROM topics WHERE topic_name LIKE 'фіт%' Можливі метасимволи оператора LIKE буде розглянуто далі.
NOT LIKE	Відбирають значення, що не відповідають зразку, приклад: SELECT * FROM topics WHERE topic_name NOT LIKE 'вів%'

Метасимволи оператора LIKE. Пошук із використанням метасимволів можна здійснювати тільки в текстових полях.

Найпоширеніший метасимвол – (%). Він означає будь-які символи.

Наприклад, якщо треба знайти слова, що починаються з літер "фіт", то напишемо LIKE 'фіт%', а якщо хочемо знайти слова, які містять символи "клуб", то напишемо LIKE '% клуб%'.

Ще один часто використовуваний метасимвол – (_). На відміну від (%), нижнє підкреслення позначає тільки один символ.

Предикат LIKE порівнює рядок, зазначений у першому виразі, для обчислення значення рядка, зі зразком, який визначено у другому виразі для обчислення значення рядка. У зразку дозволено використовувати два трафаретні символи:

- символ підкреслення (_), який можна застосовувати замість будь-якого одиничного символу в значенні, що перевіряють;
- символ відсотка (%) замінює послідовність будь-яких символів (кількість символів у послідовності може бути від 0 і більше) у значенні.

Якщо значення, що перевіряють, відповідає зразку з урахуванням трафаретних символів, то значення предиката буде TRUE. Далі наведено декілька прикладів написання зразків (табл. 5.1).

Таблиця 5.1

Метасимволи предиката LIKE

Зразки	Описи
'Abc%'	Будь-які рядки, які починаються з літер abc
'Abc_'	Рядки довжиною строго 4 символів, причому першими символами рядка мають бути abc
'% Z'	Будь-яка послідовність символів, яка обов'язково закінчується символом z
'% Rost%'	Будь-яка послідовність символів, що містить слово Rost у будь-якій позиції рядка
'% % %'	Текст, що містить не менше ніж 2 пробіли, наприклад, World Wide Web

Якщо шуканий рядок містить універсальний шаблон, то слід поставити керівний символ у реченні ESCAPE. Цей керівний символ мають використовувати у зразку перед трафаретним символом, повідомляючи про те, що останній слід трактувати як звичайний символ.

Наприклад, якщо в деякому полі слід відшукати всі значення, що містять символ (_), то шаблон (% _%) призведе до того, що будуть повернуті всі записи з таблиці. У цьому разі шаблон потрібно записати так:

'% # _%' ESCAPE '#'

Істинне значення предиката LIKE надають, відповідно до таких правил:

- якщо або значення, що перевіряють, або зразок, або керівний символ є NULL, істинне значення дорівнює UNKNOWN;
- в іншому разі, якщо значення що перевіряють, і зразок мають нульову довжину, істинне значення дорівнює TRUE;
- в іншому разі, якщо значення, що перевіряють, відповідає шаблону, то предикат LIKE дорівнює TRUE;
- якщо не дотримуються жодної з перелічених раніше умов, предикат LIKE дорівнює FALSE.

Предикат LIKE у його стандартній редакції не підтримує регулярних виразів, хоча ряд реалізацій (зокрема, Oracle) допускає їхнє використання, розширюючи можливості стандарту.

У SQL Server 2005/2008 використання регулярних виразів можливо через CLR, тобто за допомогою мов Visual Studio, які можуть використовувати для написання збережених процедур і функцій.

Однак у Transact-SQL, крім стандартних символів-шаблонів ("%" і "_"), є ще пара символів, які роблять цей предикат LIKE більш гнучким інструментом. Цими символами є такі:

- [ ] – одиночний символ із набору символів (наприклад, [zxy]) або діапазону ([az]), зазначених у квадратних дужках. Водночас можна перелічити відразу кілька діапазонів (наприклад, [0-9a-z]);
- ^ – який у поєднанні із квадратними дужками вилучає з пошукового зразка символи з набору або діапазону.

Якщо ми хочемо дізнатися, хто створив тему "фітнес", зробімо відповідний запит:

```
SELECT * FROM topics WHERE topic_name = 'фітнес'
```

Але замість імені автора, визначимо його ідентифікатор. Це й зрозуміло, адже робили запит до однієї таблиці – "Теми", а імена авторів зберігають в іншій таблиці – "Користувачі". Тому, дізнавшись ідентифікатор автора теми, нам треба зробити ще один запит – до таблиці "Користувачі", щоб дізнатися його ім'я.

У SQL передбачено можливість об'єднувати такі запити в один шляхом перетворення одного з них на підзапит (вкладений запит). Отже, щоб дізнатися, хто створив тему "фітнес", зробімо наступний запит:

```
SELECT name FROM users WHERE id_user IN
(SELECT id_author FROM topics WHERE topic_name = 'фітнес')
```

Тобто, після ключового слова *WHERE*, в умову запишімо ще один запит. SQL спочатку опрацьовує підзапит, повертає id_author = 3, і це значення передає в пропозицію *WHERE* зовнішнього запиту.

В одному запиті може бути кілька підзапитів, синтаксис у такого запиту такий:

```
SELECT назва_стовпця FROM назва_таблиці
WHERE частина умови IN
(SELECT назва_стовпця FROM назва_таблиці
```

```
WHERE частина умови IN
(SELECT назва_стовпця FROM назва_таблиці
WHERE умова))
```

Зверніть увагу, що підзапити можуть вибирати тільки один стовпець, значення якого вони будуть повертати зовнішнім запитам. Спроба вибрати кілька стовпців призведе до помилки.

Тепер, ускладнивши завдання, дізнаймося, у яких темах залишав повідомлення автор теми "нічні клуби":

```
SELECT topic_name FROM topics WHERE id_topic IN
(SELECT id_topic FROM posts WHERE id_author IN
(SELECT id_author FROM topics WHERE topic_name = 'нічні клуби'))
```

Розберімося, як це працює.

- Спочатку SQL виконає найглибший запит:

```
SELECT id_author FROM topics
WHERE topic_name = 'нічні клуби'
```

- Визначений результат (id_author = 2) передасть у зовнішній запит, який буде мати такий вигляд:

```
SELECT id_topic FROM posts WHERE id_author IN (2)
```

- Визначений результат (id_topic: 4,1) передасть у зовнішній запит, який буде мати такий вигляд:

```
SELECT topic_name FROM topics WHERE id_topic IN (4,1)
```

- І видасть остаточний результат (topic_name: про рибалку, про риболовлю). Тобто автор теми "нічні клуби" залишав повідомлення в темі "Про рибалку", створеній Сергієм (id = 1), і в темі "Про риболовлю", створеній Світланою (id = 4).

Є момент, на який варто звернути увагу: не рекомендовано створювати запити зі ступенем укладання більше за трьох. Це призводить до збільшення часу виконання та складності сприйняття коду.

5.4. Здобуття підсумкових значень

Як дізнатися кількість моделей ПК, що випускає той чи той постачальник? Як визначити середнє значення ціни на комп'ютери, які мають однакові технічні характеристики? На ці та багато інших запитань, пов'язані з деякою статистичною інформацією, можна дістати відповіді за допомогою підсумкових (агрегатних) функцій. Стандартом передбачено такі агрегатні функції (табл. 5.2).

Таблиця 5.2

Агрегатні функції

Функції	Описи
COUNT (*)	Повертає кількість рядків джерела записів
COUNT	Повертає кількість значень у вказаному стовпці
SUM	Повертає суму значень у зазначеному стовпці
AVG	Повертає середнє значення в зазначеному стовпці
MIN	Повертає мінімальне значення в зазначеному стовпці
MAX	Повертає максимальне значення в зазначеному стовпці

Усі функції повертають єдине значення. Водночас функції COUNT, MIN і MAX застосовні до даних будь-якого типу, тоді як SUM і AVG використовують тільки для даних числового типу. Різниця між функцією COUNT(*) і COUNT (назва стовпця | вираз) полягає в тому, що друга під час підрахунку не враховує NULL-значення.

Приклад. Знайти мінімальну й максимальну ціну на персональні комп'ютери:

```
SELECT MIN(price) AS Min_price
MAX(price) AS Max_price
FROM PC
```

Результатом буде єдиний рядок, що містить такі агрегатні значення: Min_price – 350,0; Max_price – 980,0.

Приклад. Знайти наявне кількість комп'ютерів, випущених виробником A:

```
SELECT COUNT(*) AS Qty
FROM PC
```

```
WHERE model IN (SELECT model
FROM Product
WHERE maker ='A')
```

У результаті маємо: Qty – 8.

Для того щоб під час визначення статистичних показників використовували тільки унікальні значення, при аргументі агрегатних функцій можна застосувати параметр DISTINCT. Інший параметр – ALL – задають за замовчуванням, і він передбачає підрахунок усіх значень, що повертають (не NULL) у стовпці.

Оператор

```
SELECT COUNT(DISTINCT model) AS Qty
FROM PC
WHERE model IN ( SELECT model
FROM Product
WHERE maker ='A')
```

дасть такий результат: Qty – 2.

Якщо ж нам потрібно визначити кількість моделей ПК, що виробляє кожен виробник, то буде потрібно використовувати пропозицію GROUP BY, синтаксично наступною після пропозиції WHERE.

5.5. Об'єднання таблиць

Для об'єднання запитів використовують службове слово UNION:

```
<запит 1>
UNION [ALL]
<запит 2>
```

Пропозиція UNION призводить до появи в результативному наборі всіх рядків кожного із запитів. Водночас, якщо визначено параметр ALL, то зберігають усі дублікати вихідних рядків, в іншому разі в результативному наборі наявні тільки унікальні рядки. Зауважмо, що можна пов'язувати разом будь-яку кількість запитів. Крім того, за допомогою дужок можна задавати порядок об'єднання.

Операцію об'єднання може бути виконано тільки в разі дотримання таких умов:

- кількість вихідних стовпців кожного із запитів має бути однаковим;
- вихідні стовпці кожного із запитів мають бути сумісними між собою (у порядку їхнього слідування) за типами даних;
- у результативному наборі використовують назви стовпців, задані в першому запиті;
- пропозицію ORDER BY застосовують до результату об'єднання, тому її може бути зазначено лише в кінці всього складеного запиту.

Приклад. Визначити номери моделей та ціни ПК та портативних комп'ютерів (рис. 5.1):

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

Model	Price
1750	1 200
1752	1 150
1298	1 050
1233	980
1321	970
1233	950
1121	850
1298	700
1232	600
1233	600
1332	400

Рис. 5.1. **Результат виконання запиту**

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

Зверніть увагу на дві речі:

- якщо в одній із поєднуваних таблиць є рядок з ідентифікатором, якого немає в іншій об'єднаній таблиці, то в результативній таблиці рядків із таким ідентифікатором не буде;

• у разі вказування умови назву стовпця пишуть після назви таблиці, у якій цей стовпець міститься (через крапку). Це зроблено, щоб уникнути плутанини, адже стовпці в різних таблицях можуть мати однакові назви, і SQL може не зрозуміти, про які конкретно клітинки йдеться.

Коректний синтаксис об'єднання з умовою має такий вигляд:

```
SELECT назва_таблиці_1.назва_стовпця1_таблиці_1,
назва_таблиці_1.назва_стовпця2_таблиці_1,
назва_таблиці_2.назва_стовпця1_таблиці_2,
назва_таблиці_2.назва_стовпця2_таблиці_2
FROM назва_таблиці_1, назва_таблиці_2
WHERE
назва_таблиці_1.назва_стовпця_за_яким_об'єднуємо =
назва_таблиці_2.назва_стовпця_за_яким_об'єднуємо
```

Якщо назва стовпця є унікальною, то назву таблиці можна опустити (як у прикладі), але робити це не рекомендовано.

Як розумієте, об'єднання дають можливість вибирати будь-яку інформацію з будь-яких таблиць, причому об'єднати можна і три, і чотири таблиці, та й умова для об'єднання може бути не одна.

Об'єднання, які раніше розглядали, називають *внутрішніми об'єднаннями*. Такі об'єднання пов'язують рядки однієї таблиці з рядками другої таблиці (а може, ще й третьої таблиці). Але бувають ситуації, коли необхідно, щоб у результат було додано рядки, які не мають пов'язаних.

Наприклад, коли створювали запит про те, які теми та якими авторами було створено, користувач Oleg у результативну таблицю не потрапив, тому що він тем не створював, а тому й пов'язаного рядка в об'єднаній таблиці не мав.

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

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

І маємо бажаний результат – усі користувачі й теми, ними створені. Якщо користувач не створював тему, то у відповідному стовпці стоїть значення NULL.

Отже, додано в запит ключове слово – *LEFT OUTER JOIN*, указавши тим самим, що з таблиці зліва треба взяти всі рядки та змінити ключове слово *WHERE* на *ON*. Крім ключового слова *LEFT OUTER JOIN,* може бути використано ключове слово *RIGHT OUTER JOIN*. Тоді будуть вибирати всі рядки із правої таблиці й наявні пов'язані з ними з лівої таблиці. І нарешті, можливе повне зовнішнє об'єднання, яке має всі рядки з обох таблиць і зв'яже між собою ті, які можуть бути пов'язаними. Ключове слово для повного зовнішнього об'єднання – *FULL OUTER JOIN*.

Змінімо в запиті лівостороннє об'єднання на правостороннє:

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

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

Синтаксис для зовнішнього з'єднання такий:

```
SELECT назва_таблиці_1.назва_стовпця, назва_таблиці_2.назва_стовпця
FROM назва_таблиці_1 ТИП ОБ'ЄДНАННЯ назва_таблиці_2
ON умова_об'єднання
де ТИП ОБ'ЄДНАННЯ або LEFT OUTER JOIN, або RIGHT OUTER JOIN
```

5.6. Групування записів і функція COUNT()

Згадаймо, які повідомлення та у яких темах у нас є. Для цього можна скористатися звичним запитом:

```
SELECT * FROM posts
```

А якщо нам треба лише дізнатися, скільки повідомлень є на форумі? Для цього можна скористатися вбудованою функцією COUNT(). Ця функція

підраховує кількість рядків. Причому, якщо як аргумент цієї функції є *, то підраховують усі рядки таблиці. А якщо як аргумент указують назву стовпця, то підраховують тільки ті рядки, які мають значення в зазначеному стовпці.

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

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

Отже, у темах є чотири повідомлення. Але якщо хочемо дізнатися, скільки повідомлень є в кожній темі? Для цього нам знадобиться згрупувати повідомлення за темами й обчислити для кожної групи кількість повідомлень. Для групування в SQL використовують оператор GROUP BY. Запит тепер буде мати такий вигляд:

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

Оператор GROUP BY указує СУБД згрупувати дані за стовпцем id_topic (тобто кожна тема – окрема група) і для кожної групи підрахувати кількість рядків.

Припустімо, що нас цікавлять тільки ті групи, у яких більше ніж два повідомлення. У звичайному запиті вказали б умову за допомогою оператора *WHERE*, але цей оператор уміє працювати тільки з рядками, а для груп ті самі функції виконує оператор *HAVING*:

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

Раніше розглядали, які умови можна задавати оператором *WHERE*, ті самі умови можна задавати й оператором *HAVING*, тільки треба запам'ятати, що *WHERE* фільтрує рядки, а *HAVING* – групи.

5.7. Редагування, оновлення та видалення даних

Припустімо, вирішено, що нашому форуму потрібні модератори. Для цього в табл. users треба додати стовпець із роллю користувача. Для додавання стовпців у таблицю використовують оператор *ALTER TABLE – ADD COLUMN*. Його синтаксис такий:

```
ALTER TABLE назва_таблиці
ADD COLUMN назва_стовпця тип
```

Додаймо стовпець role в табл. users:

```
ALTER TABLE users
ADD COLUMN role varchar (20)
```

Стовпець з'явиться в кінці таблиці. Для того щоб вказати місце розташування стовпця використовують ключове слова: *FIRST* – новий стовпець буде першим і *AFTER* – указує після якого стовпця помістити новий.

Додаймо ще два стовпці: один – kol – кількість залишених повідомлень, а другий – rating – рейтинг користувача. Обидва стовпці вставимо після поля password:

```
ALTER TABLE users
ADD COLUMN kol int (10) AFTER password,
ADD COLUMN rating varchar (20) AFTER kol
```

Тепер треба призначити роль модератора якомусь користувачеві, нехай це буде Sergey з id = 1. Для оновлення вже наявних даних слугує оператор *UPDATE*. Його синтаксис такий:

```
UPDATE назва_таблиці SET назва_стовпця = значення_стовпця
WHERE умова
```

Зробімо Сергія модератором:

```
UPDATE users SET role = 'модератор'
WHERE id_user = 1
```

Змінювати дані можна й відразу в декількох рядках і всій таблиці.

Наприклад, вирішено давати рейтинг, залежно від кількості залишених користувачем повідомлень. У таблицю спочатку внесімо значення стовпця kol:

```
UPDATE users SET kol = 50
WHERE id_user = 1

UPDATE users SET kol = 30
WHERE id_user = 2

UPDATE users SET kol = 45
WHERE id_user = 3

UPDATE users SET kol = 20
WHERE id_user = 4

UPDATE users SET kol = 2
WHERE id_user = 5

UPDATE users SET kol = 10
WHERE id_user = 6
```

А тепер задаймо рейтинг "Профі" тим, у кого кількість повідомлень більше ніж 30:

```
UPDATE users SET rating = 'Профі'
WHERE kol> 30
```

Дані змінилися у двох рядках, відповідно до заданих умов. Зрозуміло, що якщо в запиті пропустити умову, то дані буде оновлено у всіх рядках таблиці.

Припустімо, що не подобається назва "Рейтинг" у стовпці, і хочемо перейменувати стовпець на "Репутація" – reputation. Для зміни назви наявного стовпця використовують оператор *CHANGE*. Його синтаксис такий:

```
ALTER TABLE назва_таблиці
CHANGE стара_назва_стовпця нова_назва_стовпця тип
```

Змінімо rating на reputation:

```
ALTER TABLE users
CHANGE rating reputation varchar (20)
```

Зверніть увагу, що тип стовпця треба вказувати навіть, якщо його не змінюють. До речі, якщо знадобиться змінити тільки тип стовпця, то будемо використовувати оператор MODIFY. Його синтаксис такий:

```
ALTER TABLE назва_таблиці MODIFY назва_стовпця новий_тип
```

Останнє, що буде розглянуто – оператор DELETE, який дозволяє видаляти рядки з таблиці. Його синтаксис такий:

```
DELETE FROM назва_таблиці
WHERE умова
```

Із таблиці повідомлень видалімо ті записи, які залишав користувач Valera (id = 2):

```
DELETE FROM posts
WHERE id_author = '2'
```

Зрозуміло, якщо пропустити умову, то з таблиці буде видалено всі дані. Слід пам'ятати, що дані СУБД зможе видалити тільки в тому разі, якщо вони не є зовнішніми ключами для даних з інших таблиць (підтримання цілісності БД).

Наприклад, якщо захочемо видалити з табл. users користувача, який залишав повідомлення, то це не вдасться. Спочатку треба видалити його повідомлення, а вже потім і його самого.

Контрольні запитання

1. Назвіть основні мови опису запитів.
2. На чому засновано мову структурованих запитів SQL?
3. Назвіть оператори визначення даних (Data Definition Language, DDL).

4. Назвіть оператори маніпуляції даними (Data Manipulation Language, DML).

5. Перелічіть оператори визначення доступу до даних (Data Control Language, DCL).

6. Перелічіть оператори управління транзакціями (Transaction Control Language, TCL).

7. Для чого використовують оператор SELECT?

8. Які умови можна задавати, використовуючи ключове слово *WHERE*?

9. Перелічіть метасимволи оператора LIKE.

10. Перелічіть оператори здобуття підсумкових значень.

11. Опишіть способи об'єднання таблиць.

12. Перелічіть оператори групування записів і функції COUNT().

13. Перелічіть оператори редагування, оновлення та видалення даних.

Рекомендована література: [2; 6; 7].

6. Транзакції. Тригери. Індекси

Мета – визначення понять "транзакції", "тригери", "індекси" та їхніх складових. Розуміння архітектури застосунку баз даних і розташування транзакцій на шарі програмного забезпечення.

Основні питання

6.1. Вступ у транзакції.

6.2. Тригери.

6.3. Індекси.

Ключові слова: транзакція, тригер, індекс, інструкція, конкурентні транзакції, утрачене поновлення.

6.1. Вступ у транзакції

У повсякденному житті люди укладають різні види комерційних угод (транзакцій), наприклад: придбання продуктів, замовлення подорожей, зміна або скасування замовлень, придбання квитків на концерти, оплату рахунків за оренду, електрику, страхування тощо. Зрозуміло, угоди (транзакції)

мають відношення не тільки до комп'ютерів. Будь-який вид людської діяльності, що містить робочий логічний блок, який має бути виконаним або скасованим загалом, становить угоду, тобто транзакцію.

Неправильне управління транзакціями з боку прикладного програмного забезпечення може, наприклад, бути причиною:

- утрати замовлень, платежів клієнтів, відвантаження в інтернет-магазині, що не відбулося;
- перебою в реєстрації посадкових місць або виконання подвійної реєстрації пасажирів на поїзд чи літак;
- неможливості дзвінка для виклику оперативних служб у центрах із надзвичайних ситуацій тощо.

Надійний доступ до даних має бути заснованим на розроблених належним чином транзакціях SQL з урахуванням нульової толерантності (абсолютної неприпустимості) неправильних даних у базі даних.

Транзакція – група послідовних операцій із базою даних, яка становить логічну одиницю роботи з даними.

Якщо транзакцію виконано успішно, усі модифікації даних, зроблені протягом транзакції, приймають і вони стають постійною частиною бази даних. Якщо в результаті виконання транзакції відбуваються помилки й має бути виконано скасування або повернення, усі модифікації даних буде скасовано.

Є два типи транзакцій:

1. Неявні – кожного оператора, як-от INSERT, UPDATE або DELETE, виконано у транзакції.

2. Явні – група інструкцій мови Transact-SQL, початок і кінець якої позначають такими інструкціями, як BEGIN TRANSACTION, COMMIT і ROLLBACK.

Розгляньмо доступ до даних за допомогою транзакцій SQL під час виконання SQL-коду в інтерактивному режимі. Для дістання доступу до бази даних програма має ініціювати з'єднання з нею, своєю чергою, з'єднання встановлює контекст SQL-сесії. Простіше кажучи, SQL-сесію ініціює SQL-клієнт, а SQL-сервер містить сервер бази даних.

На рис. 6.1 показано спрощений вид архітектури типового застосунку баз даних із розташуванням транзакцій бази даних на шарі програмного забезпечення, відмінному від шару, призначеного для користувача інтерфейсу.

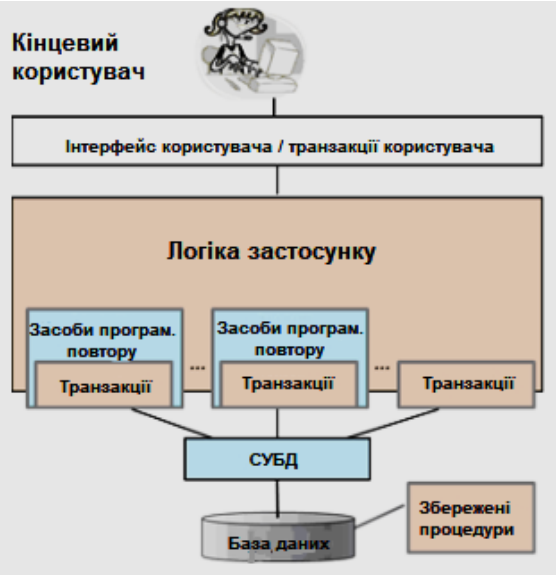


Рис. 6.1. **Розташування транзакцій SQL у протоколах прикладного рівня**

Команда SQL може містити одне або кілька пропозицій SQL. SQL-пропозиції опрацьовують, аналізують на основі метаданих бази даних, потім оптимізують і, нарешті, виконують. Для компенсації погіршення продуктивності через сповільненості операцій уведення/виведення диска, сервер зберігає всі недавно використані рядки в буферному пулі, що міститься в пам'яті RAM, де й відбувається опрацювання всіх даних.

Виконання введеної команди SQL не може бути розподіленим або здійсненим на сервері частково, команда SQL має бути виконаною повністю, інакше вся ця команда буде скасованою (виконано повернення). У відповідь на команду SQL, сервер посилає діагностичне повідомлення успішності або невдачі команди. Помилки виконання команди передають клієнту у вигляді послідовності винятків.

Принцип ACID (принцип ідеальної транзакції) визначає ідеал надійних транзакцій SQL у середовищі з багатьма користувачами. Акронім ACID походить від скорочення назв таких чотирьох властивостей транзакцій:

Неподільність (Atomic). Транзакція має бути неподільною послідовністю операцій ("усе або нічого"), які або успішно виконують до кінця та фіксують, або виконують повернення всіх цих операцій.

Погодженість (Consistent). Послідовність операцій буде передавати вміст бази даних з одного погодженого стану в інший. У момент фіксації транзакції не має бути виявлено ніяких обмежень бази даних, порушених операціями транзакції (первинні ключі, унікальні ключі, зовнішні ключі,

перевірки). У більшості СУБД обмеження застосовують безпосередньо щодо кожної операції. Наше більш точне тлумачення послідовності потребує, щоб прикладна логіка у транзакції була правильною і належним чином перевіреною (правильно побудована транзакція), включно з опрацюванням винятків.

Ізольованість (Isolated). Оригінальне визначення Гарднера і Рейтера, яке звучить як "події однієї транзакції має бути приховано від інших транзакцій, що виконують одночасно", не може бути повністю задоволено більшістю продуктів СУБД, але воно має бути взято до уваги розробниками застосунків. Поточні продукти СУБД використовують різні технології управління паралелізмом для захисту паралельних транзакцій від побічних ефектів, тому розробники застосунків мають знати, як користуватися ними належним чином.

Довговічність (Durable). Зафіксовані результати в базі даних будуть доступними на дисках, незважаючи на можливі перебої системи.

ACID-принцип потребує, щоб транзакція, яка не відповідає цим вимогам, не має бути зафіксованою; крім того, застосунок або сервер баз даних мають зробити повернення таких транзакцій. Властивість ізольованості у принципі ACID є складним завданням. Залежно від механізмів управління паралелізмом, це може призвести до конфліктів паралелізму й до занадто тривалого часу очікування, що знизить продуктивність бази даних.

Якщо логіка застосунку має виконати послідовність команд SQL у неподільному вигляді, то команди мають бути згрупованими в робочий логічний блок (англ. LUW – Logical Unit of Work). Такий блок, названий *транзакцією SQL*, під час опрацювання даних перетворює базу даних з одного погодженого стану на інший і його, отже, можна розглядати як елемент погодженості.

Будь-яке успішне виконання транзакції закінчують командою COMMIT (фіксація), тоді як невдале виконання має бути закінчено командою ROLLBACK (повернення), яка автоматично відновлює в базі даних усі зміни, унесені транзакцією. Отже, SQL-транзакцію можна також розглядати як елемент відновлення. Перевага команди ROLLBACK полягає в тому, що коли запрограмована у транзакції логіка програми не може бути завершеною, то немає ніякої необхідності у виконанні серії зворотних операцій окремими командами, робота може бути просто скасованою командою ROLLBACK, дію якої буде завжди успішно виконано.

Розгляньмо такий банківський рахунок:

```
CREATE TABLE Accounts (
acctId INTEGER NOT NULL PRIMARY KEY
balance DECIMAL (11,2) CHECK ( balance >= 0.00))
```

Типовим прикладом транзакцій SQL є переказ визначеної суми (наприклад, 100 євро) з одного рахунку на інший:

```
BEGIN TRANSACTION
UPDATE Accounts SET balance = balance -100
WHERE acctId = 101
UPDATE Accounts SET balance = balance +100
WHERE acctId = 202
COMMIT
```

У разі відмови системи або розриву з'єднання "клієнт-сервер" після першої команди UPDATE, протокол транзакції гарантує, що гроші з рахунку номер 101 не буде втрачено, оскільки буде виконано повернення транзакції. Однак, цей приклад транзакції далекий від того, щоб його вважати надійним. У разі, якщо б одного із цих двох банківських рахунків не було, команди UPDATE було б виконано, що з погляду SQL є успішним завершенням. Тому потрібно вивчити доступну діагностику SQL і перевірити кількість рядків, які було порушено кожною із цих двох команд UPDATE.

У разі, якщо перша команда UPDATE зазнає невдачі через негативний баланс рахунку номер 101, (оскільки мало місця порушення відповідного обмеження CHECK), то подальше продовження й успішне виконання другої команди UPDATE призведе до логічно помилкового стану в базі даних.

Замість простої послідовності завдань доступу до даних, деякі транзакції SQL можуть містити набір програмної логіки. У таких випадках логіка транзакції буде робити вибір під час виконання, залежно від інформації, здобутої з бази даних. Навіть у цьому разі транзакцію SQL можна розглядати як неподільний робочий логічний блок (LUW), який або успішно виконують, або скасовують. Проте перебій у транзакції зазвичай не генерує автоматично команду ROLLBACK. Після кожної команди SQL код

застосунку має перевірити діагностичні помилки, що видає сервер, і визначити, чи слід виконати команду ROLLBACK, чи ні.

Із цією метою ранній стандарт ISO SQL-89 визначив команду SQLCODE у вигляді відображення цілого числа, значення якого 0 у кінці кожної команди SQL указує на її успішне виконання, тоді як значення 100 вказує на те, що відповідні рядки не було визначено. Будь-які додатні значення вказують на попередження, а від'ємні – на різні помилки, на які надано пояснення у відповідних довідкових посібниках продукту.

Конкурентні транзакції. Прикладна програма, яка коректна працює в одного користувача, може несподівано зіткнутися із проблемами надійності роботи в багатокористувацькому навколишньому середовищі, одночасно обслуговуючи кілька клієнтів, як це показано на рис. 6.2.

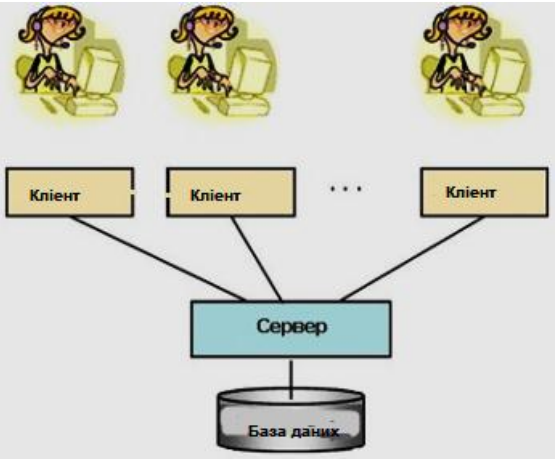


Рис. 6.2. **Доступ до однієї бази даних декількох клієнтів у багатокористувацькому середовищі**

Не маючи належних служб управління паралелізмом у базі даних, уміст бази даних або результати наших запитів можуть бути пошкодженими, тобто стати ненадійними.

Далі розгляньмо типові проблеми (аномалії) паралелізму:

- утраченого поновлення;
- зчитування "брудних" даних, тобто читання незафіксованих даних деяких паралельних транзакцій;
- неповторюваного читання, тобто читання, що повторюється, не повертаючи ті самі рядки і/або їхнє значення;
- фантомного читання, тобто під час транзакції деякі вибрані рядки може бути не видно транзакцією.

Після чого буде подано розв'язання цих проблем, відповідно до стандарту ISO SQL і реальних продуктів СУБД.

Проблема втраченого поновлення. Розгляньмо приклад, де два користувачі в різних банківських автоматах (ATM) знімають гроші з одного й того самого банківського рахунку, початковий баланс якого 1 000 $ (рис. 6.3).

Рис. 6.3. **Проблема втраченого поновлення**

Без управління паралелізмом, результат запису транзакції A за крок 5 значенням 700 $ буде втрачено у кроці 6, оскільки транзакція B наосліп записує новий баланс 500 $, який вона розрахувала. Оскільки це сталося до завершення транзакції A, то явище має назву "втрачене оновлення". Проте в кожному сучасному продукті СУБД реалізовано якийсь механізм управління паралелізмом, який захищає записи від перезапису в паралельних транзакціях як до кінця завершення.

Якщо сценарій раніше реалізовано як послідовність SELECT ... UPDATE і захищено схемою блокування, то замість утраченого поновлення, сценарій переходить до DEADLOCK, у результаті чого транзакцію B буде знижено до колишнього рівня системою управління базами даних і транзакцію A буде продовжено.

Якщо сценарій раніше реалізовано за допомогою вразливих оновлень (на основі поточних значень), як наприклад:

```
UPDATE Accounts SET balance = balance -200 WHERE acctID = 100
```

і захищено схемою блокування, то сценарій буде працювати без проблем.

Проблема зчитування "брудних" даних. Проблема зчитування "брудних" даних, зображена на рис. 6.4, становить зчитування транзакцією ненадійних (непідтверджених) даних, які можуть ще змінитися або ж щодо поновлення цих даних може бути застосовано повернення. Такого роду транзакції не мусять мати можливості робити якісь поновлення в базі даних, оскільки це призвело б до пошкодження вмісту бази даних. Справді, будь-яке використання непідтверджених даних є ризикованим і може призвести до неправильних рішень і дій.

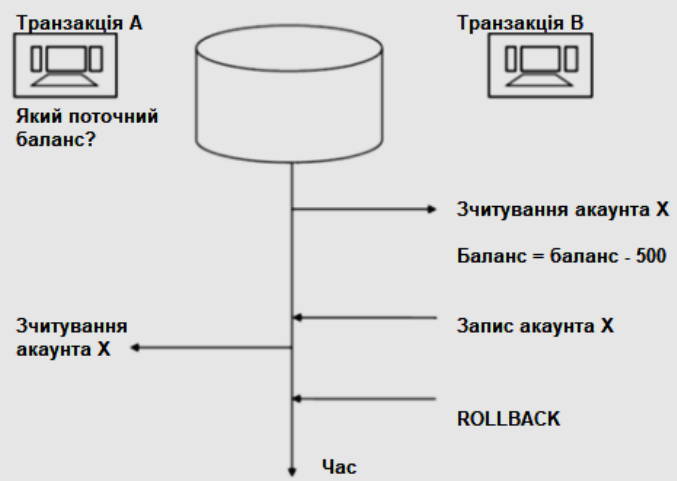


Рис. 6.4. **Приклад зчитування "брудних" даних**

Проблема неповторюваного читання. Проблема неповторюваного читання, зображена на рис. 6.5, становить нестабільність результатів запитів у транзакціях, і якщо запити має бути повторено, то деякі з раніше визначених рядків можуть бути недоступними в тому вигляді, якому вони були спочатку. Це також не виключає можливість, що в результатах повторних запитів виникають нові рядки.

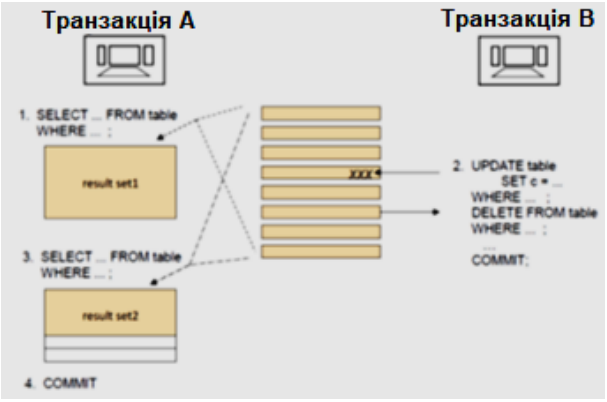


Рис. 6.5. **Проблема неповторюваного читання у транзакції А**

Проблема фантомного читання. Проблема фантомного читання, зображена на рис. 6.6, означає, що результат множини запитів у транзакції може містити нові рядки, якщо деякі запити буде повторена. Вони можуть містити знову додані рядки або ж оновлені рядки, у яких, із метою виконання умов пошуку, у запитах було змінено значення стовпців.

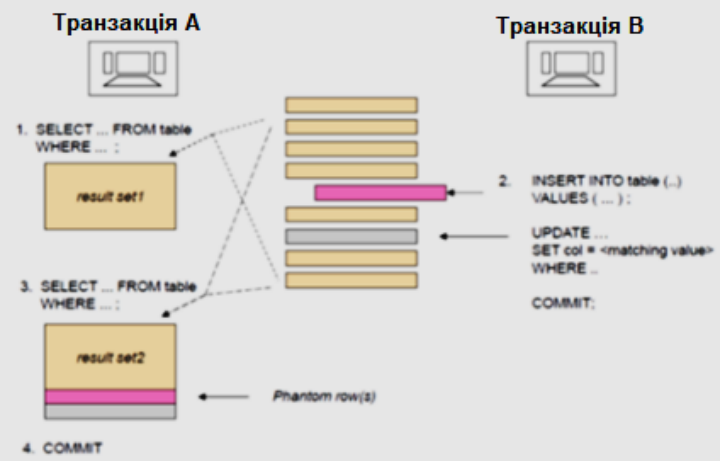


Рис. 6.6. **Приклад проблеми фантомного читання**

Для запобігання раніше описаним аномаліям використовують різні рівні ізольованості транзакцій (табл. 6.1).

Таблиця 6.1

Рівні ізольованості

Проблеми / Рівні ізольованості	Утрачене поновлення	"Брудне" читання	Неповторюване читання	Фантомне читання
READ UNCOMMITTED	Запобігає	Не запобігає	Не запобігає	Не запобігає
READ COMMITTED	Запобігає	Запобігає	Не запобігає	Не запобігає
REPEATABLE READ	Запобігає	Запобігає	Запобігає	Не запобігає
SERIALIZABLE	Запобігає	Запобігає	Запобігає	Запобігає

SERIALIZABLE – це найвищий рівень захисту, але другий користувач не зможе працювати із БД, поки перший закінчить свої операції. За замовчуванням установлюється рівень ізольованості READ COMMITTED.

6.2. Тригери

Тригер – це особливий різновид збереженої процедури, яка виконується автоматично під час виникнення події на сервері бази даних.

Є тригери на події DDL і DML.

Тригери DDL активуються у відповідь на різні події мови опису даних. Ці події, найперше, відповідають інструкціям Transact-SQL CREATE, ALTER, DROP і деяким системним збереженим процедурам, які виконують схожі з DDL операції.

Тригери DML виконуються, якщо користувач намагається змінити дані за допомогою подій мови опрацювання даних (DML). Подіями DML є процедури INSERT, UPDATE або DELETE, що застосовують до таблиці або подання. Ці тригери спрацьовують під час запуску будь-яких допустимих подій, незалежно від наявності та кількості відповідних рядків таблиці.

Тригери DML – це збережені процедури особливого типу, які автоматично вступають у силу, якщо відбувається подія мови опрацювання даних DML, яке зачіпає таблицю або подання, визначене у тригері. Тригери DML можуть використовувати для припису бізнес-правил і правил цілісності даних, виконання запитів до інших таблиць та додавання складних інструкцій Transact-SQL. Тригер та інструкція, під час виконання якої він спрацьовує, уважають однією транзакцією, яку можна повернути назад усередині тригера. Під час виявлення серйозної помилки (наприклад, брак місця на диску) уся транзакція автоматично повертається назад.

Тригери DML є аналогічними обмеженням у тому, що можуть наказувати цілісність сутностей або цілісність домену. Узагалі кажучи, цілісність сутностей має завжди приписувати на найнижчому рівні за допомогою індексів, які є частиною обмежень PRIMARY KEY і UNIQUE або створюваних незалежно від обмежень. Цілісність домену має бути запропоновано через обмеження CHECK, а посилальна цілісність – через обмеження FOREIGN KEY. Тригери DML є найбільш корисними в тих випадках, коли функції обмежень не задовольняють функціональних потреб застосунків.

Далі у списку наведено порівняння тригерів DML з обмеженнями та вказано, у чому тригери DML мають переваги.

1. Тригери DML дозволяють каскадно здійснювати зміни через пов'язані таблиці в базі даних, але ці зміни можна здійснювати більш ефективно з використанням каскадних обмежень посилальної цілісності.

Обмеженнями FOREIGN KEY можна перевірити значення стовпця тільки щодо точного збігу зі значеннями іншого стовпця, за винятком випадків, коли за допомогою пропозиції REFERENCES задають каскадні посилальні дії.

2. Для запобігання випадковим або неправильним операціям INSERT, UPDATE і DELETE та реалізації інших більш складних обмежень, ніж ті, які визначено за допомогою обмеження CHECK.

3. На відміну від обмежень CHECK, DML-тригери можуть посилатися на стовпці інших таблиць. Наприклад, тригер може використовувати інструкцію SELECT для порівняння вставлених або оновлених даних і виконання інших дій, наприклад зміни даних або відображення для користувача повідомлення про помилку.

4. Щоб оцінити стан таблиці до й після зміни даних і вжити заходів на основі цієї відмінності.

5. Кілька DML-тригерів однакового типу (INSERT, UPDATE або DELETE) для таблиці дозволяють зробити кілька різних дій у відповідь на одну інструкцію зміни даних.

6. Обмеження можуть повідомляти про помилки тільки за допомогою відповідних стандартних системних повідомлень. Якщо для призначеного для користувача застосунку потрібно складніші методи управління помилками та, відповідно, призначені для користувача повідомлення, то необхідно використовувати тригер.

7. Під час використання тригерів DML може статися повернення змін, які порушують цілісність посилань, що призводить до заборони модифікації даних. Подібні тригери можуть застосовувати під час зміни зовнішнього ключа у випадках, коли нове значення не відповідає первинному ключу. Зазвичай у зазначених випадках використовують обмеження FOREIGN KEY.

8. Якщо в таблиці тригерів є обмеження, то їхню перевірку здійснюють між виконанням тригерів INSTEAD OF та AFTER. У разі порушення обмежень виконують повернення дій тригерів INSTEAD OF, а тригер AFTER не спрацьовує.

Типи тригерів – AFTER або FOR. Тригери AFTER виконують після виконання дій інструкції INSERT, UPDATE, MERGE або DELETE. Тригери AFTER ніколи не виконуються, якщо відбувається порушення обмеження, тому ці тригери не можна використовувати для будь-якого опрацювання, яка могла б запобігти порушенню обмеження. Для кожної з операцій INSERT,

UPDATE або DELETE у зазначеній інструкції MERGE відповідний тригер викликається для кожної операції DML.

Тригер INSTEAD OF – тригери INSTEAD OF скасовують стандартні дії інструкції, що викликає тригер. Тому їх можна використовувати для перевірки на наявність помилок або перевірки значень на одному або декількох стовпцях і виконання додаткових дій, перш ніж уставити, оновленням або видаленням однієї або декількох рядків. Наприклад, якщо значення, що оновлюється, у стовпці погодинної оплати в таблиці облікової відомості починає перевищувати певне значення, то за допомогою цього тригера можна або задати висновок повідомлення про помилку та повернути транзакцію, або зробити вставлення нового запису в таблицю облікової відомості.

Головна перевага тригерів INSTEAD OF в тому, що вони дозволяють підтримувати поновлення для таких подань, які оновлювати неможливо.

Наприклад, у поданні, заснованому на декількох базових таблицях, мають використовувати тригер INSTEAD OF для підтримання операцій уставлення, оновлення та видалення, які посилаються на дані більше ніж в одній таблиці.

Інша перевага тригера INSTEAD OF полягає в тому, що він забезпечує логіку коду, за якої можна відкидати одні частини пакета та брати інші.

Інструкції тригерів DML використовують дві особливі таблиці: deleted та inserted. SQL Server автоматично створює ці таблиці й управляє ними. Ці тимчасові таблиці, що містяться в оперативній пам'яті, використовують для перевірки результатів змін даних і для встановлення умов спрацьовування тригерів DML. Не можна в цих таблицях змінювати дані безпосередньо або виконувати над ними операції мови опису даних DDL, наприклад інструкцію CREATE INDEX.

У тригерах DML табл. inserted і deleted переважно використовують для виконання таких операцій:

- розширення посилальної цілісності між таблицями;
- уставлення або оновлення даних у базових таблицях відповідного подання;
- перевірка на помилки та вживання відповідних заходів, у зв'язку з появою помилок;
- пошук відмінностей між станами таблиці до й після зміни даних і вживання відповідних заходів, залежно від наявності або відсутності відмінностей.

У табл. deleted містяться копії рядків, із якими працювали інструкції DELETE або UPDATE. Під час виконання інструкції DELETE або UPDATE відбувається видалення рядків із таблиці тригера і їхнє перенесення в табл. deleted. У табл. deleted зазвичай немає загальних рядків до таблиці-тригера.

У табл. inserted містяться копії рядків, із якими працювали інструкції INSERT або UPDATE. Під час виконання транзакції вставлення або поновлення відбувається одночасне додавання рядків до таблиці-тригера й табл. inserted. Рядки табл. inserted є копіями нових рядків таблиці-тригера.

Транзакція поновлення є аналогічною виконання операції видалення з подальшим виконанням операції вставлення. Спочатку старі рядки копіюють до табл. deleted, а потім нові рядки копіюють до таблиці-тригера та табл. inserted. Синтаксис створення тригера такий:

```
CREATE [ALTER] TRIGGER [ schema_name ] trigger_name
ON {table | view}
[WITH < dml_trigger_option > [... n]]
{FOR | AFTER | INSTEAD OF}
{[INSERT] [,] [UPDATE] [,] [DELETE]}
[WITH APPEND]
[NOT FOR REPLICATION]
AS { sql_statement [; ] [, ... n] | EXTERNAL NAME <method specifier [;
]>}
< Dml_trigger_option >: : =
[ENCRYPTION ]
[EXECUTE AS Clause]
< Method_specifier >: : =
assembly_name.class_name.method_name
```

де CREATE – створює тригер;

ALTER – змінює тригер тільки в тому разі, якщо він уже наявний;

schema_name – назва схеми, якій належить тригер DML. Дію тригерів DML обмежено схемою тієї таблиці або того подання, для яких їх створено. Аргумент schema_name не можна вказувати для тригерів DDL або тригерів входу;

trigger_nam – назва тригера. Аргумент trigger_name має відповідати правилам для ідентифікаторів з одним додатковим обмеженням: trigger_name не можна починати із символів # або ##;

table | view – таблиця або подання, у якому виконується тригер DML. Цю таблицю або подання іноді називають таблицею тригера або поданням тригера, відповідно. Указівка уточненого подання не є обов'язковою. Посилання на подання можна використовувати тільки у тригері INSTEAD OF. Не можна визначити тригери DML для локальної або глобальної тимчасових таблиць;

DATABAS – застосовує сферу тригера DDL до поточної бази даних. Якщо цей аргумент визначено, тригер спрацьовує щоразу в разі виникнення в базі даних події типу event_type або event_group;

ALL SERVER – застосовує сферу тригера DDL або тригера входу до поточного сервера. Якщо цей аргумент визначено, тригер спрацьовує щоразу в разі виникнення на поточному сервері події типу event_type або event_group;

WITH ENCRYPTION – маскує текст інструкції CREATE TRIGGER. Використання параметра WITH ENCRYPTION не дозволяє публікувати тригери як частину реплікації SQL Server. Параметр WITH ENCRYPTION не можна вказати для тригерів CLR;

EXECUTE AS – указує контекст безпеки, у якому виконується тригер. Дозволяє управляти своїм обліковим записом користувача, використовуваної екземпляром SQL Server для перевірки дозволів на будь-які об'єкти бази даних. Цей параметр є обов'язковим для тригерів у таблицях, оптимізованих для пам'яті;

NATIVE_COMPILATION – указує, що тригер компілюється у власному коді. Цей параметр є обов'язковим для тригерів у таблицях, оптимізованих для пам'яті;

SCHEMABINDING – гарантує, що таблиці, які використовує тригер, не будуть видаленими або зміненими. Цей параметр є обов'язковим для тригерів у таблицях, оптимізованих для пам'яті, і не підтримується для тригерів у звичайних таблицях;

FOR | AFTER – значення FOR або AFTER указує, що тригер DML спрацьовує тільки після успішного запуску всіх операцій в інструкції SQL, за якою спрацьовує тригер. Крім того, до запуску тригера мають успішно завершити всі каскадні дії й перевірки обмежень, на які є посилання. Не можна визначити тригери AFTER для подань;

INSTEAD OF – указує, що тригер DML виконується замість інструкції SQL, за якою він спрацьовує, тобто перекриває дії, що запускають інструкцій. Аргумент INSTEAD OF не можна використовувати для тригерів

DDL або тригерів входу. Для кожної інструкції INSERT, UPDATE або DELETE в таблиці або поданні можна визначити не більше за одного тригера INSTEAD OF. Також можна визначити подання, указавши для кожного їхнього рівня власний тригер INSTEAD OF. Тригери INSTEAD OF не можна визначати для оновлювання подань, які використовують параметр WITH CHECK OPTION. Така дія викличе помилку, якщо тригер INSTEAD OF додається до поновлюваного подання з параметром WITH CHECK OPTION. Щоб видалити цей параметр, виконайте інструкцію ALTER VIEW перед визначенням тригера INSTEAD OF;

{[DELETE] [,] [INSERT] [,] [UPDATE]} – визначає інструкції зміни даних, під час застосування яких до таблиці або подання спрацьовує тригер DML. Необхідно вказати хоча б один варіант. У визначенні тригера дозволено будь-які поєднання варіантів у будь-якому порядку. Для тригерів INSTEAD OF не можна використовувати параметр DELETE в таблицях із посилальним зв'язком, яка визначає каскадне дію ON DELETE. Аналогічно параметр UPDATE є неприпустимим у таблицях, у яких є посилальний зв'язок із каскадною дією ON UPDATE;

WITH APPEND – указує, що потрібно додати тригер наявного типу. Аргумент WITH APPEND не можна використовувати для тригерів INSTEAD OF і в тих випадках, якщо явно зазначено тригер AFTER. Для збереження зворотної сумісності аргумент WITH APPEND слід використовувати тільки за вказуванням параметра FOR без INSTEAD OF або AFTER. Не можна вказати WITH APPEND, якщо використовують EXTERNAL NAME (тобто тригер є тригером CLR);

event_type – назва мовної події Transact-SQL, запуск якої викликає спрацьовування тригера DDL;

event_group – назва стандартної групи мовних подій Transact-SQL. Тригер DDL спрацьовує після запуску будь-якої мовної події Transact-SQL, яка належить до групи event_group. Після завершення інструкції CREATE TRIGGER параметр event_group працює в режимі макросу, додаючи події, що охоплюють їхні типи, у поданні каталогу sys.trigger_events;

NOT FOR REPLICATION – указує, що тригер не має виконуватися, якщо агент реплікації змінює налаштовану для тригера таблицю;

sql_statement – умова й дії тригера. Умови тригера вказують додаткові критерії, що визначають, які події – DML, DDL або подія входу – викликають виконання тригера. Дії тригера, зазначені в інструкціях мови Transact-SQL, набувають чинності після спроби використання операції.

Тригери можуть містити будь-яку кількість інструкцій мови Transact-SQL будь-якого типу, за деякими винятками. Тригери призначено для перевірки або зміни даних під час виконання інструкцій модифікації або визначення даних. Не слід повертати з них дані користувачеві.

6.3. Індекси

Системи баз даних зазвичай використовують індекси для забезпечення швидкого доступу до реляційних даних. Індекс становить окрему фізичну структуру даних, яка дозволяє діставати швидкий доступ до однієї або кількох рядків даних. Отже, правильне налаштування індексів є ключовим аспектом поліпшення продуктивності запитів.

Індекс бази даних є багато в чому схожим з алфавітним покажчиком книги. Якщо потрібно швидко знайти будь-яку тему у книзі, спочатку слід подивитися в покажчику, на яких сторінках книги цю тему розглядають, а потім відразу ж відкрити потрібну сторінку. Подібним способом, під час пошуку певного рядка таблиці компонент Database Engine звертається до індексу, щоб дізнатися його фізичне місцезнаходження.

Але між індексом книги й індексом бази даних є дві істотні різниці:

1. Читач книги має можливість самому вирішувати, чи використовувати індекс у кожному конкретному випадку, чи ні. Користувач бази даних такої можливості не має, і за нього це вирішує компонент системи, названий *оптимізатором запитів*. Користувач може маніпулювати використанням індексів за допомогою підказок індексів, але ці підказки рекомендують застосовувати тільки в обмеженій кількості спеціальних випадків.

2. Індекс для певної книги створюють разом із книгою, після чого його більше не змінюють. Це означає, що індекс для певної теми завжди буде вказувати на один і той самий номер сторінки. На противагу, індекс бази даних може змінюватися за кожної зміни відповідних даних.

Якщо для таблиці немає відповідного індексу, для вибірки рядків система використовує метод сканування таблиці. Вираз сканування таблиці означає, що система послідовно вилучає й досліджує кожен рядок таблиці (від першої до останньої), і поміщає рядок у результівний набір, якщо для нього задовольняється умова пошуку в реченні WHERE. Отже, усі рядки вилучають, відповідно до їхнього фізичного розташування в пам'яті. Цей метод є менш ефективним, ніж доступ із використанням індексів, як пояснено далі.

Індекси зберігають у додаткових структурах бази даних, що називають *сторінками індексів*. Для кожного рядка, що індексують, є елемент індексу (index entry), який зберігають на сторінках індексів. Кожен елемент індексу складається із ключа індексу та покажчика. Ось тому елемент індексу є значно коротшим, ніж рядок таблиці, на яку він указує. Через це кількість елементів індексу на кожній сторінці індексів є набагато більшою, ніж кількість рядків на сторінці даних.

Індекси компонента Database Engine створюють, використовуючи структуру даних збалансованого дерева B+. B+-дерево має деревоподібну структуру, у якій усі найнижчі вузли містяться на відстані однакової кількості рівнів від вершини (кореневого вузла) дерева. Ця властивість підтримується навіть тоді, якщо в індексований стовпець додають або видаляють дані.

На рис. 6.7 показано структуру B+-дерева. На цьому рисунку можна бачити, що B+дерево складається з кореневого вузла, проміжних вузлів та вузлів листя.

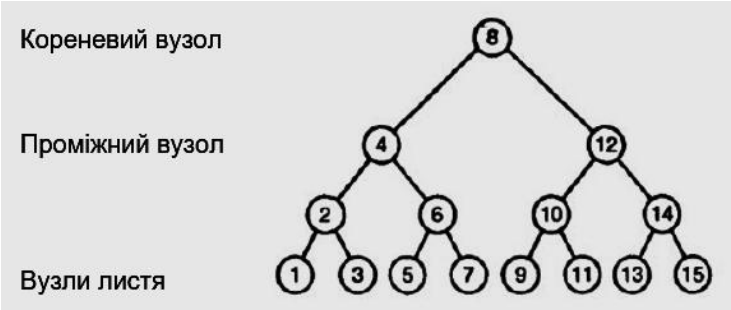


Рис. 6.7. **B+-дерево для стовпця Id**

Припустімо, що треба визначити дані під номером 13. Якщо немає індексів, пошук буде проходити за всіма 15 значеннями. Їх будуть всі послідовно переглядати.

За наявності індексу, пошук у цьому дереві будуть виконувати таким способом. У корені каталогу зазвичай розташовано серединне значення (у цьому разі 8). Нижчого за рівнем – середина середини тощо. Ліворуч розташовують менші значення, праворуч – більші.

Шукане 13 є більшим 8, тому спускаймося по правій гілці. Доходьмо до вузла зі значенням 12. 13 є більшим за 12 – спускаймося далі по правій гілці до значення 14. 13 є меншим за 14 – спускаймося по лівій гілці та визначаймо шукане значення.

Разом знадобилося 4 кроки для пошуку заданого значення замість 15, якщо немає індексу.

Індексований пошук зазвичай є найкращим методом пошуку в таблицях із великою кількістю рядків через його очевидну перевагу. Використовуючи індексований пошук, можна визначити будь-який рядок у таблиці за дуже короткий час, застосувавши лише кілька операцій уведення/виведення. А послідовний пошук (тобто сканування таблиці від першого рядка до останнього) потребує тим більше часу, чим далі міститься необхідний рядок.

Є два типи індексів: кластеризовані та некластеризовані.

Кластеризовані індекси. Кластеризований індекс визначає фізичний порядок даних у таблиці. Компонент Database Engine дозволяє створювати для таблиці лише один кластеризований індекс, тому що рядки таблиці можна впорядкувати фізично не більш ніж одним способом. Пошук із використанням кластеризованого індексу виконують від кореневого вузла B+-дерева у напрямку до вузлів дерева, пов'язаних між собою у двоспрямований зв'язаний список (doubly linked list), який має назву *ланцюжка сторінок* (page chain).

Важливою властивістю кластеризованого індексу є та особливість, що його вузли дерева містять сторінки даних. Вузли кластеризованого індексу всіх інших рівнів містять сторінки індексу. Таблиця, для якої визначено кластеризований індекс (явно або неявно), називають *кластеризованою таблицею*. Структуру B+-дерева кластеризованого індексу показано на рис. 6.8.

Рис. 6.8. **Фізична структура кластеризованих індексів**

Кластеризований індекс створюють за замовчуванням для кожної таблиці, для якої за допомогою обмеження первинного ключа визначено первинний ключ. Крім цього, кожен кластеризований індекс однозначний за замовчуванням, тобто у стовпці, для якого визначено кластеризований індекс, кожне значення даних можна зустріти тільки один раз. Якщо кластеризований індекс створюють для стовпця, що містить повторювані значення, система баз даних примусово забезпечує однозначність, додаючи чотирибайтовий ідентифікатор до рядків, що містять дублікати значень.

Групові індекси забезпечують дуже швидкий доступ до даних, якщо запит здійснює пошук у діапазоні значень.

Некластеризовані індекси. Структура некластеризованого індексу точно така сама, як і кластеризованого, але із двома важливими відмінностями:

1) некластеризований індекс не змінює фізичне упорядкування рядків таблиці;

2) сторінки вузлів некластеризованого індексу складаються із ключів індексу та закладок.

Якщо для таблиці визначити один або більше некластеризованих індексів, фізичний порядок рядків цієї таблиці не буде зміненим. Для кожного некластеризованого індексу компонент Database Engine створює додаткову індексну структуру, яку зберігають на індексних сторінках. Структуру B+-дерева некластеризованого індексу показано на рис. 6.9.

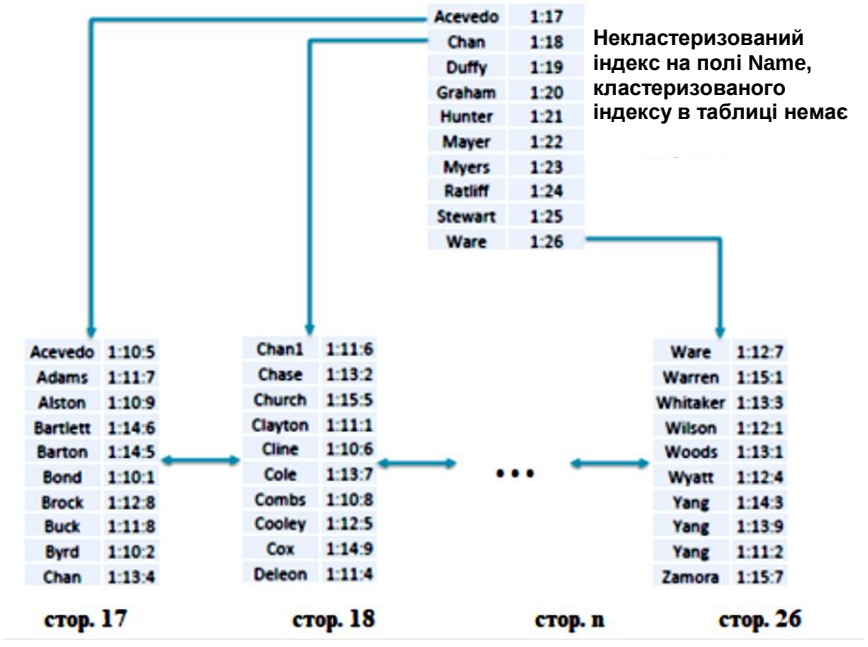


Рис. 6.9. **Структура некластеризованого індексу**

Закладка в некластеризованому індексі вказує, де міститься рядок, відповідний ключу індексу. Складова закладки ключа індексу може бути двох видів, залежно від того, чи є таблиця кластеризованою таблицею або купою (heap).

Відповідно до термінології SQL Server, *купою* називають таблицю без кластеризованого індексу. Якщо є кластеризований індекс, то закладка некластеризованого індексу показує B+-дерево кластеризованого індексу таблиці. Якщо таблиця не має кластеризованого індексу, закладка є ідентичною ідентифікатору рядка (RID – Row Identifier), що складається із трьох частин: адреси файлу, у якому зберігають таблицю; адреси фізичного блоку (сторінки), у якому зберігають рядок, і зміщення рядка на сторінці.

Як уже згадувалося раніше, пошук даних із використанням некластеризованого індексу можна здійснювати двома різними способами, залежно від типу таблиці, як-от:

1) купа – проходження під час пошуку за структурою некластеризованого індексу, після чого рядок вилучають, використовуючи ідентифікатор рядка;

2) кластеризована таблиця – проходження під час пошуку за структурою некластеризованого індексу, після чого проходження за відповідним кластеризованим індексом.

В обох випадках кількість операцій уведення/виведення є досить великою, тому слід підходити до проєктування некластеризованого індексу з обережністю, і застосовувати його тільки в тому разі, якщо є впевненість, що його використання істотно підвищить продуктивність.

Тепер, якщо познайомилися з фізичною структурою індексів, розгляньмо, як їх створювати, змінювати та видаляти, а також як здобувати інформацію про фрагментацію індексів і редагувати інформацію про індекси. Усе це підготує до подальшого обговорення використання індексів для поліпшення продуктивності системи.

Створення індексів. Індекс для таблиці створюються за допомогою інструкції CREATE INDEX. Ця інструкція має такий синтаксис:

```
CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED] INDEX index_name
ON table_name (column1 [ASC | DESC ], ...)
[INCLUDE ( column_name [...])]
[WITH
[FILLFACTOR = n]
```

```
[[ ,] PAD_INDEX = {ON | OFF}]
[[ ,] DROP_EXISTING = {ON | OFF}]
[[ ,] SORT_IN_TEMPDB = {ON | OFF}]
[[ ,] IGNORE_DUP_KEY = {ON | OFF}]
[[ ,] ALLOW_ROW_LOCKS = {ON | OFF}]
[[ ,] ALLOW_PAGE_LOCKS = {ON | OFF}]
[[ ,] STATISTICS_NORECOMPUTE = {ON | OFF}]
[[ ,] ONLINE = {ON | OFF}]]
[ON file_group | "default"]
```

Параметр index_name задає назву створюваного індексу. Індекс можна створити для одного або більше стовпців однієї таблиці, що позначають параметром table_name. Стовпець, для якого створюють індекс, указують параметром column1. Числовий суфікс цього параметра вказує на те, що індекс можна створити для декількох стовпців таблиці. Компонент Database Engine також підтримує створення індексів для подань.

Можна проіндексувати будь-який стовпець таблиці. Це означає, що стовпці, які містять значення типу даних VARBINARY (max), BIGINT і SQL_VARIANT, також можуть бути індексованими.

Індекс може бути простим або складеним. *Простий індекс* створюють за одним стовпцем, а *складений* – за кількома стовпцями. Для складеного індексу є певні обмеження, пов'язані з його розміром і кількістю стовпців. Індекс може мати максимум 900 байтів і не більше ніж 16 стовпців.

Параметр UNIQUE вказує, що проіндексований стовпець може містити тільки однозначні значення. В однозначному складеному індексі однозначною має бути комбінація значень усіх стовпців кожного рядка. Якщо ключове слово *UNIQUE* не вказують, то повторювані значення у проіндексованому стовпці дозволяють.

Параметр CLUSTERED задає кластеризований індекс, а параметр NONCLUSTERED (застосовують за замовчуванням) указує, що індекс не змінює порядок рядків у таблиці. Компонент Database Engine дозволяє для таблиці максимум 249 некластеризованих індексів.

Можливості компонента Database Engine були розширеними, дозволяючи створити підтримання індексів зі спадним порядком значень стовпців. Параметр ASC після назви стовпця вказує, що індекс створюють із зростальним порядком значень стовпця, а параметр DESC означає регресний порядок значень стовпця індексу. Отже, у використанні індексу

надано велику гнучкість. Із регресним порядком слід створювати складені індекси, значення яких упорядковано у протилежних напрямках.

Параметр INCLUDE дозволяє вказати неключові стовпці, які додають до сторінок вузлів некластеризованого індексу. Назви стовпців у списку INCLUDE не мають повторювати, і стовпець можна використовувати одночасно як ключовий і неключовий.

Щоб по-справжньому зрозуміти корисність параметра INCLUDE, потрібно розуміти, що становить покривальний індекс (covering index). Якщо всі стовпці запиту додано до індексу, то можна досягти значного підвищення продуктивності, тому що оптимізатор запитів може визначити місцезнаходження всіх значень стовпців по сторінках індексу, не звертаючись до даних у таблиці. Таку можливість називають *покривальним індексом*, або *покривальним запитом*. Тому вміщення у сторінки вузлів некластеризованого індексу додаткових неключових стовпців дозволить здобути більше покривальних запитів, водночас їхня продуктивність буде значно підвищеною.

Параметр FILLFACTOR задає заповнення кожної сторінки індексу у відсотках під час його створення. Значення параметра FILLFACTOR можна встановити в діапазоні від 1 до 100. За значення n = 100 кожну сторінку індексу заповнено на 100 %, тобто наявна сторінка вузла так само, як і сторінка, яка не належить до вузла, не буде мати вільного місця для вставлення нових рядків. Тому це значення рекомендовано застосовувати тільки для статичних таблиць. Значення за замовчуванням n = 0 означає, що сторінки вузлів індексу заповнено повністю, а кожна із проміжних сторінок має вільне місце для запису.

За значення параметра FILLFACTOR між 1 і 99 сторінки вузлів створюваної структури індексу будуть мати вільне місце. Чим більшим є значення n, тим менше вільного місця на сторінках вузлів індексу.

Наприклад, за значення n = 60 кожна сторінка вузлів індексу буде мати 40 % вільного місця для вставлення рядків індексу надалі. Рядки індексу вставляють за допомогою інструкції INSERT або UPDATE. Отже, значення n = 60 буде розумним для таблиць, дані яких піддають досить частим змінам. За значень параметра FILLFACTOR між 1 і 99 проміжні сторінки індексу мають вільне місце для запису кожна.

Після створення індексу у процесі його використання значення FILLFACTOR не підтримується. Інакше кажучи, воно тільки вказує обсяг зарезервованого місця з наявними даними в разі задавання відсоткового

співвідношення для вільного місця. Для відновлення початкового значення параметра FILLFACTOR застосовують інструкцію ALTER INDEX.

Параметр PAD_INDEX тісно пов'язано з параметром FILLFACTOR. Параметр FILLFACTOR переважно задає обсяг вільного простору у відсотках від загального обсягу сторінок вузлів індексу. А параметр PAD_INDEX указує, що значення параметра FILLFACTOR застосовують як до сторінок індексу, так і до сторінок даних в індексі.

Параметр DROP_EXISTING дозволяє підвищити продуктивність під час відтворення кластеризованого індексу для таблиці, яка також має некластеризований індекс.

Параметр SORT_IN_TEMPDB застосовують для вміщення в системну базу даних tempdb даних проміжних операцій сортування, що застосовують під час створення індексу. Це може підвищити продуктивність, якщо базу даних tempdb розміщено на іншому диску, ніж дані.

Параметр IGNORE_DUP_KEY дозволяє системі ігнорувати спробу вставлення значень, що повторюють в індексованих стовпцях. Цей параметр слід застосовувати тільки для того, щоб уникнути припинення виконання тривалої транзакції, якщо інструкція INSERT уставляє дублікат даних в індексований стовпець. Якщо цей параметр вимкнено, у разі спроби інструкції INSERT уставити в таблицю рядки, що порушують однозначність індексу, система бази даних, замість аварійного завершення виконання всієї інструкції просто видає попередження. Водночас компонент Database Engine не вставляє рядки з дублікатами значень ключа, а просто ігнорує їх і додає правильні рядки. Якщо саме цей параметр не встановлено, то виконання всієї інструкції буде аварійно завершеним.

Якщо параметр ALLOW_ROW_LOCKS активовано (має значення on), система застосовує блокування рядків. Подібним чином, коли в активному вікні ALLOW_PAGE_LOCKS, система застосовує блокування сторінок за паралельного доступу. Параметр STATISTICS_NORECOMPUTE визначає стан автоматичного перерахунку статистики зазначеного індексу.

Активований параметр ONLINE дозволяє створювати, перестворювати та видаляти індекс у діалоговому режимі. Цей параметр дозволяє у процесі зміни індексу одночасно змінювати дані основної таблиці або кластеризованого індексу й будь-яких пов'язаних індексів.

Наприклад, у процесі перевтілення кластеризованого індексу можна продовжувати оновлювати його дані та виконувати запити за цими даними.

Параметр ON створює зазначений індекс або на файловій групі за замовчуванням (значення default), або на зазначеній файловій групі (значення file_group).

Розгляньмо створення індексів на прикладах.

1. PRIMARY KEY за замовчуванням створює унікальний кластеризований індекс, але може створити первинний ключ, який задасть некластеризований індекс:

```
CREATE TABLE Persons
(Id INTEGER PRIMARY KEY
Name VARCHAR (255))
```

Якщо в таблиці вже є кластеризований індекс, то можна створити некластеризований індекс:

```
CREATE TABLE Persons
(Id INTEGER PRIMARY KEY NONCLUSTERED
Name VARCHAR (255))
```

2. Обмеження UNIQUE за замовчуванням створює унікальний некластеризований індекс:

```
CREATE TABLE Persons
(Id INTEGER
Name VARCHAR (255) UNIQUE)
```

але можна створити UNIQUE, який задасть кластеризований індекс:

```
CREATE TABLE Persons
(Id INTEGER
Name VARCHAR (255) UNIQUE CLUSTERED)
```

3. Створюючи індекси не через PRIMARY KEY або UNIQUE, можна створити неунікальний індекс (кластеризований або некластеризований).

За замовчуванням створюють *неунікальний некластеризований індекс*:

```
CREATE INDEX index_name
ON table_name (column_name)
```

Унікальний некластеризований індекс:

```
CREATE UNIQUE INDEX index_name
ON table_name (column_name)
```

Неунікальний кластеризований індекс:

```
CREATE CLUSTERED INDEX index_name
ON table_name (column_name)
```

Унікальний кластеризований індекс:

```
CREATE UNIQUE CLUSTERED INDEX index_name
ON table_name (column_name)
```

Рекомендації щодо створення та використання індексів. Хоча компонент Database Engine не накладає ніяких практичних обмежень на кількість індексів, через дві причини цю кількість слід обмежувати.

По-перше, кожен індекс займає певний обсяг дискового простору, отже, є ймовірність того, що загальна кількість сторінок індексу бази даних може перевищити кількість сторінок даних у базі.

По-друге, на відміну від отримання вигоди під час використання індексу для вибірки даних, уставлення та видалення даних такої вигоди не надають через необхідність в обслуговуванні індексу. Чим більше індексів має таблиця, тим більший потрібно обсяг роботи з їхньої реорганізації. Загальним правилом буде розумним вибирати індекси для частих запитів, а потім оцінювати їхнє використання.

Для кластеризованого індексу найбільш відповідним стовпцем буде унікальний стовпець, який не підтримує NULL-значень і який не буде оновлюватися. Ось чому первинний ключ часто використовують як кластеризований індекс.

Створюйте індекси на стовпці, за якими відбувається пошук, сортування, групування, об'єднання таблиць.

Не створюйте індекси на полях, які рідко використовують у запитах.

Не створюйте індекси на полях, які містять кілька унікальних значень, наприклад, стовпець, що містить тільки значення чоловічої або жіночої

статі. Чим більше дублікатів у стовпці, тим гірше працює індекс. Чим більше унікальних значення, тим вищою є працездатність індексу. Якщо можливо, використовуйте унікальний індекс.

Для невеликих таблиць пошук за індексом може зайняти більше часу, ніж просте сканування всіх рядків.

Для таблиць, які часто оновлюють, використовуйте якомога менше індексів.

Якщо пропозиція WHERE інструкції SELECT містить умову пошуку з одним стовпцем, то для цього стовпця слід створити індекс. Це особливо рекомендовано за високої селективності умови. Під *селективністю* (selectivity) умови мають на увазі співвідношення кількості рядків, що задовольняють умову, до загальної кількості рядків у таблиці. Висока селективність відповідає меншому значенню цього співвідношення. Опрацювання пошуку з використанням індексованого стовпця буде найбільш успішним за селективності умови, що не перевищує 5 %.

Стовпець не слід індексувати за постійного рівня селективності умови 80 % або вищого. У такому разі для сторінок індексу будуть потрібними додаткові операції введення/виведення, які знизять будь-яку економію часу, що досягають шляхом використання індексів. У цьому разі будуть швидше виконувати пошук скануванням таблиці, що й буде зазвичай вибрано оптимізатором запитів, роблячи індекс зайвим.

Контрольні запитання

1. Дайте визначення транзакції.
2. Які є типи транзакцій?
3. У чому полягає принцип ACID (принцип ідеальної транзакції)?
4. Дайте визначення конкурентним транзакціям.
5. У чому полягає проблема втраченого поновлення?
6. У чому полягає проблема зчитування "брудних" даних?
7. Дайте визначення тригерів.
8. Для чого використовують індекси?
9. Які є типи індексів?

Рекомендована література: [1; 3; 7].

7. Технологія роботи з базами даних на платформі NET Framework

Мета – визначення понять "постачальник даних", "автономна частина", "пул під'єднання" та їхніх основних складових. Розуміння основних напрямів технологій ADO.NET у складі роботи з базами даних.

Основні питання

7.1. Архітектура ADO.NET.

7.2. Під'єднання бази даних.

7.3. Здобуття даних. Об'єкт SqlCommand.

Ключові слова: інтерфейс, клас, бібліотека, постачальник даних, автономна архітектура, пул під'єднання.

7.1. Архітектура ADO.NET

Сьогодні велике значення має робота з даними. Для зберігання даних використовують різні системи управління базами даних: MS SQL Server, Oracle, MySQL тощо. І більшість великих застосунків, так чи інакше, використовують для зберігання даних ці системи управління базами даних. Однак, щоб здійснювати зв'язок між базою даних і застосунком на C # необхідний посередник. І саме таким посередником є технологія ADO.NET.

ADO.NET надає собою технологію роботи з даними, засновану на платформі .NET Framework. Ця технологія становить нам набір класів, через які можна відправляти запити до баз даних, установлювати під'єднання, діставати відповідь від бази даних і виробляти ряд інших операцій.

Причому важливо зазначити, що систем управління баз даних може бути множина. У своїй сутності вони можуть відрізнятися. MS SQL Server, наприклад, для створення запитів використовує мову T-SQL, а MySQL та Oracle застосовують мову PL-SQL. Різні системи баз даних можуть мати різні типи даних. Також можуть відрізнятися якісь інші моменти. Однак функціонал ADO.NET побудовано таким способом, щоб надати розробникам уніфікований інтерфейс для роботи з найрізноманітнішими СУБД.

Із початку 1990-х рр. лідируючі позиції на ринку програмного забезпечення міцно утримує компанія Microsoft. Першими інтерфейсами доступу до даних, випущеними компанією Microsoft, були DAO і ODBC.

Інтерфейс DAO – це об'єктна модель, заснована на бібліотеці Microsoft Jet Engine, використовуваній у Microsoft Access. Крім самого Access, технологія Jet Engine може бути застосованою до будь-якого сховища даних, що використовує індексного-послідовний метод доступу.

Інтерфейс ODBC становить класичний інтерфейс, заснований не на об'єктній моделі, а на використанні дескрипторів API у стилі мови С. Спочатку метод доступу до даних ODBC було призначено для використання корпоративними базами даних, як-от Oracle, DB 2, Sybase та власною базою даних Microsoft під назвою SQL Server.

Із погляду розробників реляційних баз даних інтерфейс ODBC виявився практично досконалим. І настільні, і корпоративні реляційні бази даних містили драйвери ODBC. Істотний недолік інтерфейсу ODBC полягав у тому, що в ньому не використовували модель COM (Component Object Model). Набагато більш удалими розробками Microsoft, призначеними для універсального доступу до даних і заснованими на використанні моделі COM, стали OLE DB та ADO.

У *технології OLE DB* доступ до даних дістають за допомогою набору чітко визначених абстрактних об'єктів, названих *котипами* (*cotypes*). Котипи складаються з набору добре структурованих інтерфейсів. В інтерфейсах OLE DB разом із користувачем інтерфейсами COM мають застосовувати вказівні знаки, структури та масиви мови С.

Бібліотека ADO насправді є таким собі "нашаруванням" на OLE DB, однак відрізняється від неї меншою кількістю об'єктів і тим, що кожному типу об'єктів відповідає однин головний інтерфейс.

Бібліотека ADO.NET підхоплює естафету, розпочату OLE DB та ADO. Код ADO.NET працює переважно в керованому оточенні .NET. Це дозволяє значно підвищити продуктивність застосунків. Бібліотеки ADO.NET може бути використано в будь-якій мові програмування, яку підтримує платформа .NET. Завдяки численним аспектам інтеграції ADO.NET і XML, комбінація ADO.NET та стека XML дозволяє діставати ефективний доступ до реляційних, нереляційних та інших моделей даних.

Двома основними компонентами ADO.NET є постачальник даних .NET Framework та автономна модель зберігання даних (рис. 7.1).

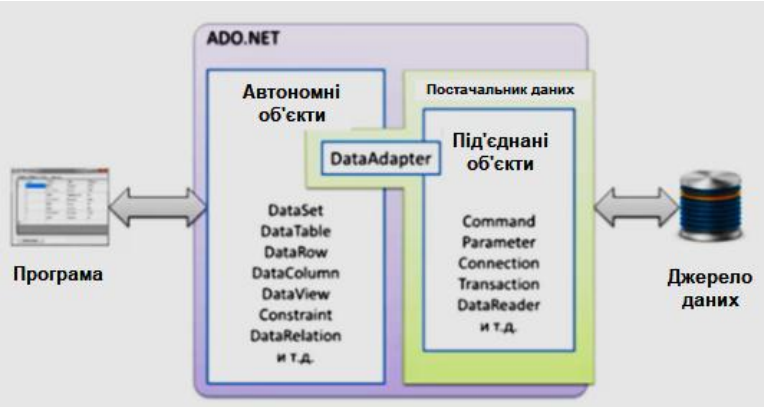


Рис. 7.1. **Архітектура ADO.NET**

Постачальник даних .NET Framework використовують для з'єднання з базою даних, виконання команд та здобуття результатів виконання команд.

Автономна частина архітектури, подана у вигляді класу DataSet, є розташованим в оперативній пам'яті кешем даних, для зберігання результатів, здобутих від постачальника даних.

Розподіл архітектури на дві незалежні частини дозволяє використовувати технологію ADO.NET для побудови багаторівневих застосунків, а так само для створення застосунків, що використовують різні джерела даних.

Усі класи, що надає технологія ADO.NET, можна зарахувати до частини що поєднується або є автономною. Єдиний виняток – клас DataAdapter, який є посередником між під'єднаною й автономною частинами ADO.NET.

Об'єкт DataAdapter становить набір команд для заповнення даними автономної частини, а так само для передавання відкладених змін назад у джерело даних.

Автономна частина. Головним об'єктом автономної частини ADO.NET є об'єкт DataSet. На абстрактному рівні об'єкт DataSet є віртуальною базою даних, що містить таблиці з даними та відношеннями між таблицями. Таблиці, що містяться в об'єкті DataSet, є екземплярами класу DataTable, а зв'язки – класу DataRelation. Автономна частина архітектури ADO.NET міститься у просторі назв System.Data.

Основні простори назв, які використовують в ADO.NET:

System.Data – визначає класи, інтерфейси, делегати, які реалізують архітектуру ADO.NET.

System.Data.Common – містить класи, спільні для всіх провайдерів ADO.NET.

System.Data.Design – визначає класи, які використовують для створення своїх власних наборів даних.

System.Data.Odbc – визначає функціональність провайдера даних для ODBC.

System.Data.OleDb – визначає функціональність провайдера даних для OLE DB.

System.Data.Sql – зберігає класи, які підтримують специфічну для SQL Server функціональність.

System.Data.OracleClient – визначає функціональність провайдера для баз даних Oracle.

System.Data.SqlClient – визначає функціональність провайдера для баз даних MS SQL Server.

System.Data.SqlServerCe – визначає функціональність провайдера для SQL Server Compact 4.0.

System.Data.SqlTypes – містить класи для типів даних MS SQL Server.

Microsoft.SqlServer.Server – зберігає компоненти для взаємодії SQL Server і середовища CLR.

Постачальники даних .NET Framework. Як уже зазначено, постачальник даних .NET Framework використовують для встановлення з'єднання з базою даних, виконання команд і здобуття результатів (рис. 7.2).

Рис. 7.2. **Постачальники даних .NET Framework**

Постачальники .NET Framework для різних джерел даних:

• Постачальник даних SQL Server – використовують для застосунків, що працюють із базами даних MS SQL Server.

• Постачальник даних OLE DB – використовують для джерел даних Microsoft Access і Microsoft Excel.

• Постачальник даних Oracle – використовують для застосунків, що працюють з Oracle джерелами даних.

Основні об'єкти частини, що під'єднують. Далі наведено основні класи, що під'єднують частини ADO.NET:

• Connection – клас, що дозволяє встановлювати під'єднання до джерела даних.

• Transaction – клас, що надає транзакцію для зазначеної команди.

• Command – клас, що становить виконувану команду в базовому джерелі даних.

• Parameter – клас, що надає параметри для зазначеної команди.

• DataReader – клас, що становить еквівалент конвеєрного курсора з можливістю тільки читання даних у прямому напрямку.

Класи автономної моделі ADO.NET. Можна виділити кілька основних класів автономної моделі ADO.NET, а саме:

1) DataSet є ядром автономного режиму доступу до даних в ADO.NET;

2) DataTable. Найбільше цей клас схожий на таблицю БД. Він складається з об'єктів DataColumn, DataRow, що становлять рядки та стовпці;

3) DataView – об'єкт подання бази даних;

4) DataRelation – клас дозволяє задавати відношення між різними таблицями, за допомогою яких можна перевіряти відповідність даних із різних таблиць.

7.2. Під'єднання бази даних

Для роботи з базами даних природно, найперше, треба мати якусь базу даних. У цьому разі будемо розглядати основні концепції ADO.NET переважно на прикладі MS SQL Server.

Об'єкт Connection

Для створення під'єднання до джерела даних в ADO.NET є спеціальний об'єкт Connection. Залежно від вибраного джерела даних, цей

об'єкт можуть називати по-різному. Для створення під'єднання до баз даних MS SQL Server слід використовувати об'єкт SqlConnection, який міститься у просторі назв System.Data.SqlClient.

Для роботи з об'єктом SqlConnection йому потрібно надати рядок з'єднання, який указує, яким способом потрібно під'єднатися до джерела даних.

Рядок з'єднання – рядок, що складається з пар "назва – значення", що містить відомості про ініціалізації, що передають у вигляді параметра від застосунку до джерела даних.

Насамперед треба визначити рядок з'єднання, який надає інформацію про базу даних та сервер, до яких належить установити під'єднання:

```
class Program
{
static void Main (string [] args)
{
string connectionString = @ "Data Source =. \ SQLEXPRESS
Initial Catalog = usersdb
Integrated Security = True"
SqlConnection connection = new SqlConnection (conStr)
}}
```

Рядок під'єднання становить набір параметрів у вигляді пар "ключ – значення". У цьому разі для під'єднання до раніше створеної бази даних usersdb визначаймо рядок під'єднання із трьох параметрів:

- Data Source – указує на назву сервера. За замовчуванням – це .\SQLEXPRESS. Оскільки в рядку використовують слеш, то на початку рядка ставлять символ @. Якщо назва сервера бази даних відрізняється, то, відповідно, її й треба використовувати.
- Initial Catalog – указує на назву бази даних на сервері.
- Integrated Security – установлює перевірку справжності. Параметр True означає під'єднання через обліковий запис Microsoft Windows.
- User Id – дозволяє вказати назву для під'єднання до сервера.
- Password – пароль входу у SQL Server.

Після створення рядка під'єднання можна створювати екземпляр SqlConnection.

Відкриття й закриття з'єднання. Для відкриття з'єднання із сервером в об'єкта SqlConnection є метод Open (), а для закриття – метод Close():

```
connection.Open ( )
...
connection.Close ( )
```

У конструктор об'єкта SqlConnection передається рядок під'єднання, який ініціалізує об'єкт. Щоб використовувати цей об'єкт і під'єднуватися до бази даних, мають виконати його метод Open(), а після завершення роботи з базою даних треба викликати метод Close() для закриття під'єднання.

Створювати екземпляр SqlConnection доцільно у блоці using. Під час виходу з блоку using на екземплярі створеного SqlConnection буде викликано метод Dispose, який автоматично закриє з'єднання із джерелом даних:

```
using (SqlConnection connection = new SqlConnection
(connectionStringBuilder.ConnectionString))
```

Пул під'єднання. Зазвичай, у програмі використовують одну або кілька одних і тих самих конфігурацій під'єднань. І щоб розробнику не доводилося створювати по кілька разів на коді програми фактично одне й саме під'єднання, в ADO.NET використовують механізм пулу під'єднань. До того ж сама по собі операція створення нового об'єкта під'єднань є досить витратною, і використання пулу дозволяє оптимізувати продуктивність програми.

Пул під'єднань дозволяє використовувати раніше створені під'єднання. Коли менеджер під'єднань, який управляє пулом, отримує запит на відкриття нового під'єднання за допомогою методу Open(), то він перевіряє всі під'єднання пулу.

Якщо менеджер під'єднань має у пулі доступне під'єднання, яке в поточний момент не використовують, то його повертають для використання. Якщо ж доступного під'єднання немає, і максимальний розмір пулу ще не перевищено (за замовчуванням розмір дорівнює 100), то створюють

нове під'єднання. Якщо доступного під'єднання немає, але водночас перевищено максимальний розмір пулу, то нове під'єднання додають у чергу і чекають, поки в пулі звільниться місце, і тоді воно стане доступним.

Після закриття під'єднання за допомогою методу Close() закрите під'єднання повертається в пул під'єднань, де воно є готовим до повторного використання за наступного виклику методу Open().

У пул можуть поміщати лише з'єднання з однаковою конфігурацією. ADO.NET підтримує кілька пулів одночасно, по одному для кожної конфігурації.

Технологія connection pooling дозволяє знизити витрати на відкриття та закриття з'єднання:

```
static void Main (string [] args)
{
string conStr = @ "Data Source = (localdb) \ MSSQLLocalDB
Initial Catalog = ShopDB
Integrated Security = true
Pooling = true"
```

Тут Pooling = true – включення або відключення пулу для цього під'єднання.

Якщо параметр Min Pool Size не вказано в рядку під'єднання або має значення 0, то під'єднання в пулі будуть закритими після періоду відсутності активності (4 – 8 хвилин), або якщо розірвано зв'язок із сервером бази даних. Але якщо значення параметра Min Pool Size більше за 0, пул під'єднань не видаляється, поки не буде вивантажено домен застосунку AppDomain і не завершиться процес.

Жорстке кодування рядка під'єднання (тобто його визначення в коді програми), зазвичай, рідко використовують. Набагато більш гнучкий шлях становить визначення його у спеціальних конфігураційних файлах програми. У проєктах десктопних застосунків – це файл App.config, а у вебзастосунках – це переважно файл Web.config.

App.config є XML-файлом із множиною зумовлених розділів конфігурації та підтримує розділи конфігурації, що настроюють. Розділ "Конфігурація" становить фрагмент XML зі схемою, призначеною для зберігання деякого типу інформації.

У цьому разі, оскільки створено проєкт консольного застосування, то у проєкті має бути файл App.config, який за замовчуванням має таке визначення:

```
<? xml version = "1.0" encoding = "utf-8"?>
< configuration >
< startup >
<supportedRuntime version = "v4.0" sku = ".NETFramework, Version =
v4.5" />
</ startup>
</ configuration >
```

Змінімо його, додавши визначення рядка під'єднання:

```
<? xml version = "1.0" encoding = "utf-8"?>
< configuration >
< startup >
<supportedRuntime version = "v4.0" sku = ".NETFramework, Version =
v4.5" />
</ Startup>
< ConnectionStrings >
<Add name = "DefaultConnection"
connectionString = "Data Source = (localdb) \ MSSQLLocalDB
Initial Catalog = usersdb
Integrated Security = True"
providerName = "System.Data.SqlClient" />
</ ConnectionStrings>
</ configuration>
```

Для визначення всіх під'єднань у програмі в межах вузла <configuration> додано новий вузол <connectionStrings>. У цьому вузлі визначають рядки під'єднання за допомогою елемента <add>. Можна використовувати в застосунку множину рядків під'єднання, і, відповідно, також у файлі визначити множину елементів <add>.

Кожен рядок під'єднання має назву, що визначають за допомогою атрибута name. У цьому разі рядок під'єднання називають DefaultConnection. Назва може бути довільною.

Атрибут connectionString власне зберігає рядок під'єднання, тобто весь той текст, який раніше визначали в методі Main. І третій атрибут providerName задає простір назв провайдера даних. Оскільки будемо під'єднуватися до бази даних MS SQL Server, то, відповідно, використовувати провайдер для SQL Server, функціональність якого полягає у просторі назв System.Data.SqlClient.

7.3. Здобуття даних. Об'єкт SqlCommand

Після встановлення під'єднання можна виконати до бази даних будь-які команди, наприклад, додати в базу даних об'єкт, видалити, змінити його або просто вилучити. Команди подано об'єктом інтерфейсу System.Data.IDbCommand. Провайдер для MS SQL надає його реалізацію у вигляді класу SqlCommand.

SqlCommand-об'єкт є частиною технології ADO.NET, що дозволяє виконувати інструкції T-SQL над джерелом даних. Для правильної роботи об'єкта SqlCommand потрібно надати під'єднання до джерела даних (екземпляр класу SqlConnection) (рис. 7.3).

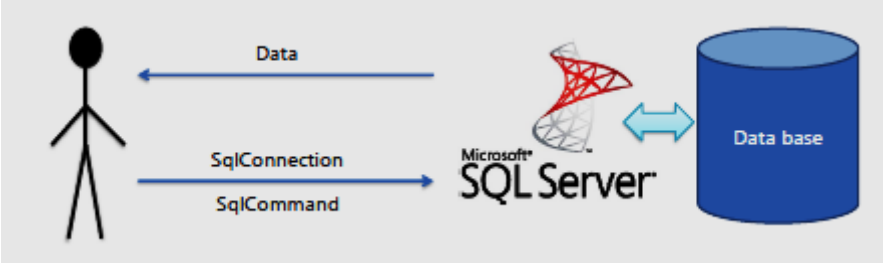


Рис. 7.3. **Під'єднання користувача до джерела даних**

Для виконання команди буде потрібно sql-вираз та об'єкт під'єднання.

Способи створення об'єкта SqlCommand:

1. Використання конструктора за замовчуванням:

```
SqlCommand cmd = new SqlCommand () //створюємо примірник
конструктора SqlCommand
cmd . Connection = connection //створюємо з'єднання
cmd . CommandText = "Some T - SQL Command" //створюємо текст
команди
```

2. Використання методу CreateCommand( ) об'єкта SqlConnection:

```
cmd = connection.CreateCommand ()
cmd.CommandText = "Some T-SQL Command"
```

3. Використання перевантаження конструктора із двома параметрами:

```
cmd = new SqlCommand ("Some T-SQL Command", connection)
```

Незалежно від способу створення об'єкта SqlCommand, він має дістати примірник SqlConnection. За допомогою властивості CommandText установлюють SQL-вираз, який будуть виконувати. За допомогою властивості Connection можна встановити об'єкт під'єднання SqlConnection.

Після створення об'єкта SqlCommand можна виконувати T-SQL інструкції над джерелом даних, для чого об'єкт SqlCommand має ряд методів, що дозволяють виконувати різні типи інструкцій.

Методи SqlCommand такі:

- ExecuteNonQuery() – просто виконує sql-вираз і повертає кількість змінених записів. Прийнятний для sql-виразів INSERT, UPDATE, DELETE.
- ExecuteReader() – виконує sql-вираз і повертає рядки з таблиці. Прийнятний для sql-виразів SELECT.
- ExecuteScalar() – виконує sql-вираз і повертає одне скалярне значення, наприклад, число. Прийнятний для sql-виразів SELECT у парі з одним із вбудованих функцій SQL, як наприклад, Min, Max, Sum, Count.

Об'єкт SqlDataReader

Об'єкт DataReader дозволяє переглядати табличні дані, які повертає команда. Визначити об'єкт DataReader можна за допомогою методу ExecuteReader () об'єкта Command:

```
SqlDataReader reader = cmd . ExecuteReader ( )
```

За допомогою об'єкта DataReader можна переглядати тільки один рядок в один момент часу та тільки в одному напрямку.

SqlDataReader має ряд методів, властивостей та індексаторів для здобуття інформації, яка надійшла від джерела даних. Далі наведено деякі з них:

- Read() – цей метод повертає значення true або false, залежно від того, чи досягнуто кінець набору рядків і в разі кожного його виклику переміщається до наступного рядка.
- FieldCount – властивість, що дозволяє визначити кількість полів у рядків, які надійшли від джерела.
- GetName (int index) – метод, що дозволяє визначити назву поля за індексом.

Для здобуття значень окремих полів рядків SqldataReader має два перевантаження індексатора із цілочисельним індексом та рядковим: точно типізовані методи Get <FieldType> (int index) і GetFieldValue <T> (int index).

Розгляньмо виконання команд, які повертають скалярні значення ExecuteScalar ():

```
using System
using System.Data.SqlClient

namespace CBS.ADO_NET.ExecuteCommand
{
class Program
{
static void Main (string [] args)
{
string conStr = @ "Data Source = (localdb) \ MSSQLLocalDB
Initial Catalog = ShopDB
Integrated Security = True" // створення рядка під'єднання
SqlConnection connection = new SqlConnection (conStr)
connection.Open ( ) // відкриття під'єднання
SqlCommand cmd = new SqlCommand ("SELECT Phone FROM Customers WHERE CustomerNo = 1", connection) // створення команди, яка повертає скалярне значення
string phoneNumber = (string) cmd.ExecuteScalar () //виконання команди. Команда ExecuteScalar повертає значення типу "об'єкт", тому треба перевизначити його надавши йому тип string
```

```
            Console.WriteLine (phoneNumber) //виводимо на консоль змінну phoneNumber
                }
            }
        }
```

Розглянувши, як виконувати команди за допомогою методу Execute NonOuery(), однак якщо захотіти зчитувати дані, які зберігають в таблиці, то буде потрібно інший метод – ExecuteReader(). Цей метод повертає об'єкт SqlDataReader, який використовують для читання даних і дозволяє переглядати табличні дані, що повертає команда.

Для вибірки даних із БД використовують sql-вираз SELECT. У цьому разі вибираймо всі стовпці всіх рядків таблиці. Діставши під час виконання запиту об'єкт SqlDataReader, можна вважати, що всі дані отримано.

Але спочатку перевіряймо, а чи є взагалі дані за допомогою властивості SqlDataReader. Якщо дані є, то виводьмо заголовки таблиці за допомогою методів reader.GetName (). Причому маємо стовпці у вибірці саме в тому порядку, у якому їх визначено в таблиці. Тобто якщо другим у таблиці йде стовпець Name, то щоб визначити його номер застосовують метод GetName (1) (оскільки нумерація стовпців починається з нуля).

Далі зчитуймо самі дані. За допомогою методу reader.Read () рідер переходить до наступного рядка та повертає логічне значення, яке вказує, чи є дані для зчитування.

У циклі while (reader.Read()) у порядку проходження стовпців визначаймо дані за допомогою методу GetValue(), який повертає дані у вигляді об'єкта типу object.

Після завершення роботи із SqlDataReader треба його закрити методом Close(). І поки один SqlDataReader не закрито, інший об'єкт SqlDataReader для одного й того самого під'єднання використовувати не зможемо:

```
using System
using System.Data.SqlClient

namespace CBS.ADO_NET.ExecuteTableCommands
{
```

```
class Program
{
static void Main (string [] args)
{
string conStr = @ "Data Source = (localdb) \ MSSQLLocalDB
Initial Catalog = ShopDB
Integrated Security = True"
SqlConnection connection = new SqlConnection (conStr)
connection . Open ( )
// Побудова команди, яка повертає дані в табличному поданні
SqlCommand cmd = new SqlCommand ("SELECT * FROM Customers",
connection)
SqlDataReader reader = cmd.ExecuteReader ()
```

За допомогою об'єкта SqlDataReder можна переглядати результати запиту рядок за рядком. Метод Read() повертає значення true або false, залежно від того, чи досягнуто кінець пакета рядків, які прийшли від сервера. Метод Read у разі кожного його виклику переміщається до наступного рядка пакета, який прийшов від сервера:

```
while (reader.Read ())
{
for (int i = 0; i <reader.FieldCount; i ++)
{
Console.WriteLine (reader.GetName (i) + ":" + reader [i])
}
Console.WriteLine (new string ( '_', 20))
}
reader.Close ( )
connection.Close ( )
}}}
```

Асинхронне виконання команд

Під час створення UI-застосунків, що використовують об'єкт SqlCommand, виконання команд слід виробляти асинхронно, щоб уникнути блокування призначеного для користувача інтерфейсу.

Спеціально для цього об'єкт Command має методи асинхронного виконання команд. Усі методи для асинхронного виконання команд мають префікс Async:

```
await cmd . ExecuteNonQueryAsync ()
await cmd.ExecuteReaderAsync ()
await cmd.ExecuteScalarAsync ()
```

Виконання пакетних запитів

Під час створення SqlCommand властивості CommandText можна привласнити відразу кілька операторів T-SQL. Під час використання об'єкта DataReader для такої команди після перегляду даних першого набору рядків потрібно перейти до наступного набору за допомогою методу NextResult () об'єкта DataReader.

Створення пакета запитів має такий синтаксис:

```
SqlCommand cmd = new SqlCommand ( "SELECT * FROM Customers
WHERE CustomerNo = 1; SELECT * FROM Employees WHERE Emp
loyeeID = 1;", connection)
```

Створення параметризованих запитів

Для початку створімо простий застосунок, який буде запитувати ID клієнта й після цього виводити його дані:

```
static void Main (string [] args)
{
string conStr = @ "Data Source = (localdb) \ MSSQLLocalDB
Initial Catalog = ShopDB
Integrated Security = True" //створення рядка під'єднання
Console.WriteLine ("Уведіть ID клієнта ")
var customerNo = Console . ReadLine ()
//для створення параметризованого запиту використовують метод string.Format
string commandStr = string.Format ("SELECT * FROM Customers WHERE CustomerNo = {0};", customerNo)
using (SqlConnection connection = new SqlConnection (conStr)) // створення під'єднання
{
```

```
connection.Open ( )
SqlCommand cmd = new SqlCommand (commandStr, connection)
using (SqlDataReader reader = cmd.ExecuteReader ()) // виконання запиту та читання результатів
{
while (reader.Read ())
      {
       for (int i = 0; i <reader.FieldCount; i ++)
            Console.WriteLine ( "{0}: {1}", reader.GetName (i), reader [i])
       }
    }
} } } }
```

Для відправлення запиту безпосередньо додавали дані у вираз SQL. У цьому разі мають на увазі, що значення для змінної CustomerNo вводить користувач. Microsoft не рекомендує використовувати конкатенацію рядків для запитів, щоб уникнути зміни структури запиту користувачем.

Конкатенація (лат. *concatenatio* "приєднання ланцюгами; зчеплення") – операція склеювання об'єктів лінійної структури, зазвичай рядків. Наприклад, конкатенація слів "мікро" і "світ" дасть слово "мікросвіт".

Якщо користувач хоче змінити запит, то слід запустити програму й замість ID клієнта ввести 1 OR CustomerNo = 2.

У підсумку в базу даних буде додано два об'єкти. Це відносно нешкідливий вид підміни sql-виразу, але реальні можливості вбудовування шкідливих скриптів такі, що можна взагалі втратити дані у БД, якщо надати користувачам можливість подібним способом додавати дані.

Щоб вийти із цієї ситуації, у SQL-командах використовують параметри. Для визначення параметрів застосовують об'єкт SqlParameter. Цей об'єкт має ряд конструкторів, але в цьому разі передають назву параметра та його значення. Причому назву параметрів починають зі знака @ і має збігатися з тією назвою, що використовують у sql-виразі (тобто в INSERT INTO Users (Name, Age) VALUES (@name, @age)). Після визначення параметра його додають у колекцію параметрів команди.

Під час виконання команди на місце параметрів у sql-вираз підставляють їхні значення. Водночас не важливо, що параметр @name у значенні визначає ще одну команду INSERT – усе його значення буде додано у стовпець name в табл. Users.

Параметри, які використовують у командах, можуть бути декількох типів. Тип параметра задають за допомогою властивості Direction об'єкта SqlParameter. Ця властивість набирає одне із значень переліку Parameter Direction:

- Input – параметр є вхідним, тобто призначеним для передавання значень у sql-вираз запиту. Це значення за замовчуванням для всіх параметрів.
- InputOutput – параметр може бути як вхідним, так і вихідним.
- Output – параметр є вихідним, тобто його використовують для повернення запитом будь-яких значень.
- ReturnValue – параметр становить результат виконання виразу або збереженої процедури.

Збережені процедури

Збережені процедури є ще однією формою виконання запитів до бази даних. Але, порівняно з раніше розглянутими запитами, які надсилають із програми базі даних, збережені процедури визначають на сервері, і мають велику продуктивність і є більш безпечними.

Об'єкт SqlCommand має вбудоване підтримання збережених процедур. Зокрема в нього визначено властивість CommandType, що набирає значення з переліку System.Data.CommandType. Значення System.Data. CommandType. StoredProcedure якраз указує, що буде використано збережену процедуру.

Контрольні запитання

1. На чому засновано архітектуру ADO.NET?
2. Перелічіть об'єкти автономної частини архітектури ADO.NET.
3. Які є постачальники даних .NET Framework?
4. Які об'єкти використовують для під'єднання бази даних?

Рекомендована література: [1; 5; 7].

8. Автономна частина архітектури ADO.NET

Мета – визначення понять "кешування даних", "реляційна таблиця", "клас-контейнер" та їхніх основних складових. Розуміння технології доступу до даних в автономній моделі ADO.NET/.

Основні питання

8.1. Клас DataSet.

8.2. Класи DataColumn, DataRow і DataTable.

8.3. Клас DataTable і його використання.

8.4. Ключі, відношення й обмеження.

8.5. Стан і версії рядків DataRow.

Ключові слова: кешування даних, реляційна таблиця, клас-контейнер, ключ, відношення, обмеження.

8.1. Клас DataSet

Розпочинаймо до вивчення автономної моделі доступу до даних і почнімо з розгляду класу DataSet. Об'єкт DataSet є об'єктом верхнього рівня в автономній моделі доступу до даних ADO.NET.

Клас DataSet призначено для кешування даних, які може бути використано тільки в режимі читання, за межами бази даних. Наявність подібного кеша позбавляє від необхідності зайвий раз звертатися до бази даних для пошуку таблиці *lookup table* – таблиці, що містить відносно постійну інформацію.

Крім того, об'єкти DataSet використовують для поновлення даних у режимі від'єднання. Набір даних вилучають із бази даних і поміщають в об'єкт DataSet, після чого вміст об'єкта DataSet оновлюється в автономному режимі, а зроблені зміни пізніше передаються назад у базу даних.

Оскільки об'єкт DataSet складається з таблиць і використовує стандартну реляційну термінологію, його часто порівнюють із реляційною базою даних, розташованою в оперативній пам'яті. Виконаймо це порівняння.

Отже, об'єкт DataSet має такі *подібності* з реляційною базою даних.

Набір даних, поданий об'єктом DataSet, складається з таблиць (об'єктів DataTable), які, своєю чергою, складаються зі стовпців (об'єктів DataColumn) і рядків (об'єктів DataRow). Метадані є тільки на рівні стовпців-зразків; інакше кажучи, кожен стовпець DataColumn може мати дані тільки одного типу.

Об'єкти DataTable можуть мати первинний ключ, а також зовнішні ключі, відношення й унікальні обмеження.

Об'єкт DataSet може враховувати регістр назв таблиць і стовпців, що входять до його складу, а також даних, що містяться в них.

Підтримуються обчислювані стовпці.

Чи є методи для виконання множинних змін такі, щоб траплялися всі зміни або вони не траплялися взагалі?

Наведімо *відмінності* класу DataSet від реляційних баз даних:

- не реалізовано концепцію автентифікації або авторизації;
- підтримується тільки підмножина операторів SQL-подібних мов запитів, і ці оператори можна застосовувати лише для вибірки наборів даних;
- не реалізовано концепцію ізоляції транзакцій або блокування записів у багатокористувацькому середовищі;
- деякі із властивостей об'єкта DataSet мають пряме відношення до відображення XML.

Об'єкт DataSet може відігравати роль кеша даних зі складною структурою, розташованого в оперативній пам'яті та поданого у вигляді універсальної об'єктної моделі, що не залежить від проблемної галузі.

Об'єктну модель класу DataSet засновано на використанні колекцій. На верхньому рівні цієї моделі розміщено клас DataSet. Структура класу DataSet і його основні колекції показано на рис. 8.1 (деякі класи й колекції до цієї схеми не ввійшли).

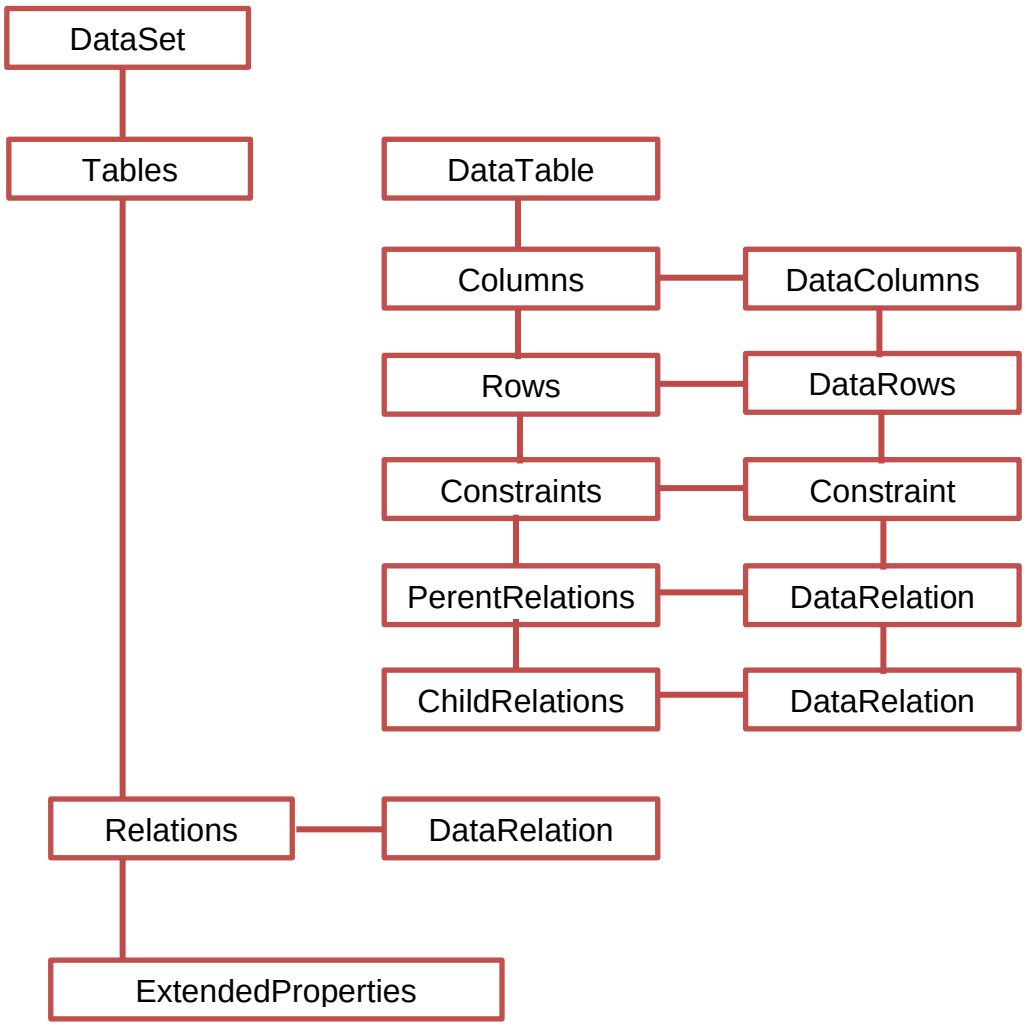


Рис. 8.1. **Класи-колекції, що входять до складу класів DataSet**

Наведімо короткий огляд цих класів:

DataSet – клас-контейнер верхнього рівня;

internalDataCollectionBase – базовий клас для класів-колекцій;

Constraint і Constraintcollection – обмеження, що накладають на дані, із метою забезпечення несуперечності окремої таблиці й несуперечливості між таблицями;

DataColumn і DataColumnCollection – реляційні стовпці (у реляційній теорії стовпець називають *атрибутом (attribute)* – тільки не плутайте його з атрибутами XML);

DataRelation і DataRelationCollection – відношення між таблицями (забезпечують несуперечливість даних);

DataRow і DataRowCollection – реляційні кортежі;

DataTable і DataTableCollection – таблиці, що складаються з рядків і стовпців;

ForeignKeyConstraint – клас, похідний від Constraint, забезпечує реляційну цілісність;

Uniqueconstraint – клас, похідний від Constraint, забезпечує унікальність стовпців таблиці;

Propertycollection – клас, похідний від HashTable (властивість класів DataSet, DataTable і DataColumn), містить визначені користувачем розширені властивості;

DataRowView – налаштовує подання будь-якого стовпця, поданого об'єктом DataRow;

DataView – налаштовує подання будь-якої таблиці, поданої об'єктом DataTable;

DataViewManager – властивість класу DataSet, призначена для управління колекцією подань таблиць (об'єктів DataTable) об'єкта DataSet;

DataViewSetting і DataViewSettingCollection налаштування, прийняті за замовчуванням для подань DataView, керованих об'єктом DataViewManager;

DataSysDescriptionAttribute – атрибут, який містить описовий текст, призначений для проєктувальників.

Клас DataView і супутні йому класи забезпечують пошук, сортування та фільтрацію даних в оперативній пам'яті, а також інтеграцію даних з елементами управління та засобами програмування WYSIWYG (What You See Is What You Get – що бачиш на екрані, то й дістанеш під час друкування).

8.2. Класи DataColumn, DataRow і DataTable

Оскільки об'єкт DataSet містить колекцію об'єктів DataTable, один зі способів заповнення порожнього об'єкта DataSet полягає у створенні стовпців DataColumn і додаванні цих стовпців до таблиці DataTable. Цей підхід передбачає ручне заповнення схеми таблиці DataTable. Таблиці DataTable можуть бути або доданими до колекції Tables об'єкта DataSet, або використаними самостійно. Після визначення схеми таблиці DataTable можна створити, а потім додати до цієї таблиці рядки (подані об'єктами DataRow). Доступ до окремих значень стовпців рядка DataRow, що входить до складу таблиці DataTable, можна дістати так, якщо б об'єкт DataTable був двовимірним масивом.

У класі DataColumn є цілий ряд конструкторів. Найбільш часто використовувані з них установлюють значення властивостей ColumnName і DataType. За замовчуванням значенням властивості ColumnName є порожній рядок, а значенням властивості DataType – System.String. Значення властивості ColumnName має бути унікальним у межах табл. DataTable, до якої входить цей стовпець. Теоретично значенням властивості DataType може бути будь-який тип даних, оскільки множиною допустимих типів даних є System.Type. На практиці, однак, є деякі обмеження. Доступні типи даних можна уточнити в документації до .NET, які є значеннями властивості System.DataColumn.DataType.

Як і стовпці в реляційних базах даних, об'єкти DataColumn підтримують поняття значення за замовчуванням і допустимості Null-значення. Властивість DataColumn.AllowDBNull показує, чи може стовпець містити об'єкт спеціального типу DBNull. Клас System.DBNull відповідає поняттю значення Null у реляційних базах даних і має єдине статичне поле Value. Отже, властивість AllowDBNull дозволяє привласнювати одну DataColumn значення System.DBNull.Value. Своєю чергою, у класі DataRow є метод isNull(n), що дозволяє визначити, чи містить n-й стовпець DataColumn значення System.DBNull.Value.

Об'єкт DataTable

DataTable – об'єкт автономної частини технології ADO.NET, що становить таблицю з даними. Основними складовими об'єкта DataTable є об'єкти DataRow, що становлять рядки таблиці з даними й об'єкти DataColumn, що становлять стовпці таблиці.

Об'єкти DataRow міститися в таблиці в колекції рядків (DataRowCollection), а об'єкти DataColumn у колекції стовпців (DataColumnCollection).

Об'єкт DataColumn

Об'єкт DataColumn є окремим стовпцем таблиці. Під час створення об'єкта DataColumn слід указати назву стовпця й тип даних, які буде містити стовпець. Якщо не вказати тип даних для стовпця, буде автоматично встановлено тип string.

Після створення екземпляра DataColumn можна додавати його до колекції стовпців примірника DataTable:

```
DataColumn firstColumn = new DataColumn ("First Column", typeof (int))
DataColumn secondColumn = new DataColumn ("Second column", typeof (string))
DataColumnCollection columnCollection = table.Columns
columnCollection.AddRange (new DataColumn [] {firstColumn, secondColumn})
```

Тепер таблиця table містить стовпець [First Column] типу int і [Second Column] типу string. Набір стовпців таблиці з їхніми назвами й типами даних називають схемою таблиці.

Об'єкт DataRow

Перш ніж використовувати таблицю DataTable як окремий об'єкт, її необхідно заповнити рядками. Клас DataTable містить колекцію (тип DataRowCollection) рядків DataRow. Об'єкт DataRow не можна створити за допомогою оператора new. Він має бути пов'язаним із конкретною табл. DataTable так, що б рядок, що створюється, уже "знав" свою структуру (тобто з яких стовпців DataColumn таблиця складається). Для створення об'єкта DataRow застосовують метод DataTable.NewRow. Цей метод створює рядок DataRow з потрібною кількістю стовпців потрібних типів, однак не додає цей рядок до колекції DataTable.Rows. Для додавання нового рядка до таблиці необхідно заповнити поля рядків даними й потім скористатися методом DataTable.Rows.Add. Є також перевантажений варіант методу DataTable.Rows.Add, що приймає як аргумент масив об'єктів, що є значеннями стовпців цього рядка. Однак за один раз до таблиці можна додати тільки один рядок DataRow. Після створення об'єкта DataRow

за допомогою методу DataTable.NewRow (і навіть ще до того, як новий рядок буде доданим до колекції DataTable.Rows) значенням властивості DataRow.Table стане назва табл. DataTable, до якої належить цей рядок.

Якщо в таблиці є стовпець зі встановленою властивістю AutoIncrement, під час додавання нового рядка в цьому стовпці можна вказати значення, яке перекриє автоматично сгенироване значення. Явна вказівка значення Null призведе до виникнення винятку. Однак якщо скористатися перевантаженим варіантом методу DataTable.Rows.Add, які мають як аргумент масив об'єктів, тоді стовпцям можна надати значення Null, указавши це значення як один з елементів масиву.

Об'єкт DataRow є окремим рядком таблиці. Для створення об'єкта DataRow не можна користуватися конструктором. Рядок може бути створеним на основі схеми наявної таблиці. Для створення рядків в об'єкта DataTable є метод NewRow (), який повертає об'єкт DataRow з полями, аналогічними стовпцям таблиці:

```
DataRow newRow = table.NewRow ()
```

Під час створення нового рядка за допомогою методу NewRow його не поміщають до колекції рядків таблиці.

На практиці часто доводиться створювати об'єкти DataTable з аналогічною схемою таблиці у джерелі даних. Об'єкт DataReader надає ряд методів для розв'язання цієї проблеми. Метод GetName дозволяє дізнатися назву стовпця таблиці у джерелі даних, а метод GetFieldType – визначити тип даних стовпця таблиці у джерелі даних.

Наступний метод на основі переданого йому об'єкта SqlDataReader створює новий екземпляр DataTable зі схемою, аналогічною таблиці, до якої звертається DataReader:

```
Private static DataTable CreateSchemaFromReader (SqlDataReader
reader, string tableName)
{
DataTable table = new DataTable (tableName)
For (int i = 0; i <reader.FieldCount; i ++)
table.Columns.Add (new DataColumn (reader.GetName (i),
reader.GetFieldType (i)))
return table
}
```

8.3. Клас DataTable і його використання

Щоб стовпці DataColumn можна було використовувати, їх має бути додано до об'єкта DataTable. Водночас один стовпець DataColumn не може бути доданим відразу до кількох таблиць DataTable. Значенням властивості DataColumn. Table є назва таблиці, до якої належить цей стовпець DataColumn. Клас DataTable інкапсулює в собі стандартну прямокутну таблицю. Найбільш поширений конструктор класу DataTable має як аргумент рядок із назвою нової таблиці. Після створення об'єкта DataTable до нього можна додавати стовпці DataColumn. Для зберігання стовпців в об'єкті DataTable використовують стандартну колекцію типу DataColumnCollection. Додавати стовпці до колекції можна по одному за раз (метод DataColumn Collection.Add) або ж відразу цілим масивом (метод DataColumnCollection. AddRange). Крім того, метод DataColumnCollection.Add містить дуже зручний перевантажений варіант, що передає необхідні аргументи в конструктор класу DataColumn, так що стовпець DataColumn може бути створеним і доданим до колекції стовпців за допомогою всього лише одного оператора.

Клас DataTable, як і DataColumn, містить колекцію властивостей ExtendedProperties. Вони можуть стати в нагоді, скажімо, для збереження дати створення таблиці, номера її версії або дати, на яку заплановано наступне оновлення таблиці DataTable (якщо вона становить таблицю бази даних).

Подібно до об'єкта ADO Recordset, об'єкт DataTable є самостійною сутністю і може використовуватися сам по собі.

Крім використання об'єкта DataTable як окремої сутності, із ним можна працювати у складі колекції таблиць (тип DataTableCollection), що входить до об'єкта DataSet. Це найбільш поширений спосіб застосування об'єкта DataTable. Крім того, у класі DataTable є властивість Defaultview, призначена для роботи зі складними елементами управління, як-от ListBox.

І, нарешті, у класі DataTable є захищений конструктор, що має як аргументи об'єкти Serializationinfo та Streamingcontext. Наявність цих параметрів є обов'язковою умовою, тому що клас DataTable реалізує власний механізм серіалізації та відповідний об'єкт може бути переданим іншому застосунку за допомогою класів із простору назв.

Розгляньмо метод створення таблиці яка має таку саму структуру, як і таблиця у джерелі бази даних. Інакше кажучи, розгляньмо створення методів, що дозволяють створювати нову таблицю на основі даних, що надає об'єкт DataReader.

Об'єкт CreateSemaFromReader приймає екземпляр SqlDataReader і на основі цих даних створює схему таблиці:

```
Private static DataTable CreateSchemaFromReader (SqlDataReader reader, string tableName)
{
DataTable table = new DataTable (tableName)
For (int i = 0; i <reader.FieldCount; i ++)
table.Columns.Add (new DataColumn (reader.GetName (i), reader.GetFieldType (i)))
return table
}
```

Метод WriteDataFromReader приймає два параметри. Перший параметр – це посилання на таблицю, а другий – посилання на екземпляр класу SqlDataReader. Цей метод дозволяє перенести дані з вихідної бази даних у нашу таблицю:

```
Private static void WriteDataFromReader (DataTable table, SqlDataReader reader)
{
While (reader.Read ())
{
DataRow row = table.NewRow ()
For (int i = 0; i <reader.FieldCount; i ++)
row [i] = reader [i]
table.Rows.Add (row)
}}
Static void Main (string[] args)
{
String conStr = @ "Data Source = (localdb) \ MSSQLLocalDB
Initial Catalog = ShopDB
Integrated Security = True"
```

```
SqlConnection connection = new SqlConnection (conStr)
connection.Open ( )
SqlCommand cmd = new SqlCommand ("SELECT * FROM Customers", connection)
SqlDataReader reader = cmd.ExecuteReader ()
DataTable table = CreateSchemaFromReader (reader, "Customers")
```

Наступні два рядки дозволяють перевірити схему, яку створює метод CreateSchemaFromReader. Тобто на консоль мають вивести назву полів і їхні типи:

```
Foreach (DataColumn column in table.Columns)
Console.WriteLine ("{0}: {1}", column.ColumnName, column.DataType)
WriteDataFromReader (table, reader)
Console.WriteLine ()
Foreach (DataRow row in table.Rows)
{
Foreach (DataColumn column in table.Columns)
Console.WriteLine ("{0}: {1}", column.ColumnName, row [column])
Console.WriteLine ()
}
reader.Close ()
connection.Close ()
}}}
```

Перше, що хотіли дізнатися, які поля в таблиці в нас є, а також їхні типи. Далі бачимо, що всі рядки новоствореної таблиці були заповненими.

8.4. Ключі, відношення й обмеження

Познайомилися з основними особливостями стовпців, рядків і таблиць. Настав час перейти до вивчення більш складних понять реляційної теорії, реалізованих у моделі DataSet.

Як і в реляційних базах даних, один або декілька стовпців табл. DataTable можуть відігравати роль первинного ключа. Наявність первинного ключа не є обов'язковою умовою, однак вона є необхідною для виконання деяких розширених функцій, використовуваних для поєднання

вмісту об'єктів DataSet. Первинний ключ має бути унікальним. Незважаючи на це додавання до стовпця DataColumn обмеження Uniqueconstraint або надання властивості DataColumn.Unique значення true ще не робить цей стовпець первинним ключем.

Якщо один стовпець таблиці буде позначеним як первинний ключ (властивість PrimaryKey), значення властивості цього стовпця AllowDBNull автоматично буде дорівнювати false, однак якщо для стовпця буде просто встановлено властивість Unique, то значення властивості AllowDBNull не буде дорівнювати false. Оскільки PrimaryKey – це властивість класу DataTable, його не можна використовувати як екземпляр самостійного класу; доступ до значення цієї властивості дістають за допомогою пари "get-метод/set-метод". Значення властивості PrimaryKey становить масив об'єктів DataColumn. Навіть якщо первинний ключ – це один стовпець, значення відповідної властивості все одно має бути масивом.

Об'єкт DataSet містить властивість Constraints, значенням якого є колекція типу Constraintcollection типізованих об'єктів Constraint. Клас Constraint – це абстрактний базовий клас, який має два спеціалізовані похідні класу: ForeignKeyConstraint і Uniqueconstraint. Клас Uniqueconstraint призначено для реалізації концепції унікальності, визначеної у стандарті SQL-92. Своєю чергою, об'єкт ForeignKeyConstraint описує "дочірню" половину відношення "батько – нащадок" між двома табл. DataTable.

Клас Uniqueconstraint – це оригінальний клас, який має п'ять варіантів конструктора, кожен із яких має свій набір параметрів. Цими параметрами можуть бути один стовпець DataColumn, масив стовпців DataColumn, назва обмеження, а також булева змінна, яка показує, що це обмеження Uniqueconstraint є первинним ключем таблиці. Значенням властивості Uniqueconstraint.Table є назва таблиці, якій належать стовпці.

Як раніше було зазначено, таблиці, що містяться в базі даних, мають ряд обмежень для даних, які можна зберігати в цих таблицях. Наприклад, деякі стовпці можуть зберігати лише унікальні дані, деякі заповнюються автоматично, а деякі не можуть зберігати порожні значення. Для створення таких обмежень в автономній частині ADO.NET об'єкти DataTable і DataColumn мають ряд властивостей.

Властивості об'єкта DataColumn. Для перевірки даних на рівні таблиць об'єкт DataColumn надає такі властивості:

1) Readonly – повертає або задає значення, яке вказує на допустимість зміни стовпця після додавання рядка до таблиці;

2) AllowDBNull – повертає або задає значення, яке вказує на допустимість нульових значень у цьому стовпці для рядків, що належать таблиці;

3) MaxLength – повертає або задає максимальну довжину текстового стовпця;

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

Обмеження об'єкта DataTable. Для об'єкта DataTable можна задати такі обмеження:

1) UniqueConstraint – надає обмеження на набір стовпців, у яких усі значення мають бути унікальними. Слід користуватися цим обмеженням у тому разі, якщо необхідно гарантувати унікальність комбінацій значень різних полів таблиці;

2) PrimaryKey – особливий вид обмеження на унікальність. Первинний ключ таблиці може бути тільки один;

3) ForeignKeyConstraint – обмеження, яке буде гарантувати, що не можна створити рядок у дочірній таблиці, яка посилається на рядок батьківської таблиці, якого немає.

Розгляньмо встановлення первинного ключа. PrimaryKey – особливий вид обмеження на унікальність. Первинний ключ таблиці може бути тільки один:

```
Static void Main (string[] args)
{
DataTable table = new DataTable ( )
DataColumn column1 = table.Columns.Add ("Column1", typeof (string))
DataColumn column2 = table.Columns.Add ("Column2", typeof (string))
```

Для створення первинного ключа створюймо обмеження UniqueConstraint на стовпець 1:

```
table.Constraints.Add (new UniqueConstraint (column1, true))
Console.WriteLine ("is unique:"+ table.Columns [0] .Unique)
Console.WriteLine ("Primary key columns count:"+ table.PrimaryKey.Length)
Console.WriteLine ("Primary key column name:"+ table.PrimaryKey [0]
.ColumnName)
}}}
```

Після запуску, бачимо, що є первинний ключ, і його встановлено на стовпець Column 1.

Об'єкт DataRelation задає відношення "батько – нащадок" між двома табл. DataTable на основі пари "первинний ключ – зовнішній ключ" (властивості PrimaryKey і ForeignKey, відповідно). Це значить, що для кожного рядка дочірньої таблиці значення, що стоять у певних стовпцях дочірньої таблиці, мають повторювати й у відповідній батьківській таблиці.

Первинний ключ батьківської таблиці мають створювати одночасно із зовнішнім ключем дочірньої таблиці. Кажучи формально, стовпець батьківської таблиці, що входить до відношення, що не обов'язково має бути первинним ключем, проте він має бути унікальним. Якщо стовпець визначено як первинний ключ (PrimaryKey) або як унікальний (Unique), створення відношення проходить успішно, і для стовпця батьківської таблиці додано обмеження Uniqueconstraint. Крім того, за замовчуванням додано й зовнішній ключ (властивість ForeignKey). Після цього під час додавання до таблиці нових рядків активізують обидва обмеження – UniqueConstraint і ForeignKeyConstraint, – якщо тільки не встановлено значення властивості DataSet. EnforceConstraints, що дорівнює false. Зазвичай, це тимчасовий захід, наприклад на час поєднання двох об'єктів DataSet або для того, щоб полегшити одночасне додавання деяких дочірніх рядків. Крім того, є перевантажений варіант методу DataRelationsCollection.Add, який дозволяє створювати відношення без створення відповідних обмежень.

Розгляньмо створення зовнішнього ключа. ForeignKeyConstraint – обмеження, яке буде гарантувати, що не можна створити рядок у дочірній таблиці, яка посилається на рядок батьківської таблиці, якщо немає.

```
Static void Main (string[] args)
{
DataSet ds = new DataSet(); //створення віртуальної бази даних
DataTable parentTable = new DataTable ( ); //батьківська таблиця
DataTable childTable = new DataTable ( ); //дочірня таблиця
DataColumn childColumn = childTable.Columns.Add ("ChildColumn",
typeof(int))
DataColumn parentColumn=parentTable.Columns.Add
("ParentColumn", typeof(int))
```

Обмеження ForeignKeyConstraint буде працювати, якщо батьківська й дочірня таблиця містяться в одному об'єкті DataSet:

```
ds.Tables.AddRange (new DataTable [] {childTable, parentTable})
```

На табл. parentTable установлюймо обмеження у вигляді первинного ключа UniqueConstraint на стовпець parentColumn:

```
parentTable.Constraints.Add (new UniqueConstraint (parentColumn))
```

На табл. childTable установлюймо обмеження у вигляді зовнішнього ключа ForeignKeyConstraint, котрий пов'язує стовпці parentColumn (батьківської таблиці) і childColumn (дочірньої таблиці):

```
childTable.Constraints.Add (new ForeignKeyConstraint (parentColumn, childColumn))
```

Заповнюймо табл. parentTable, додаючи до неї рядок зі значенням 1:

```
DataRow parentRow = parentTable.NewRow ( )
parentRow [0] = 1
parentTable.Rows.Add (parentRow)
```

Заповнюємо табл. childTable, додаючи до неї також значення 1:

```
DataRow childRow = childTable.NewRow ( )
childRow [0] = 1
```

Після створення обмеження ForeignKeyConstraint не можна додавати до дочірньої таблиці рядок, що посилається на рядки з батьківської таблиці, яких немає. Наступний запис додає рядок childRow до табл. childTable:

```
childTable.Rows.Add (childRow)
}}}
```

8.5. Стан і версії рядків DataRow

У кожного об'єкта DataRow є властивість Rowstate. Вона становить стан рядка DataRow щодо колекції рядків DataTable.Rows. Як простий приклад згадайте, що якщо рядок DataRow створюють за допомогою методу DataTable.NewRow, то його ще не пов'язано з таблицею DataTable;

додавання рядка до таблиці здійснюють за допомогою методу DataTable. Rows.Add. До того як рядок додано до колекції рядків, значення властивості цього рядка DataRow.Rowstate одне DataRowState.Detached.

Для того щоб передати відкладені зміни у джерело даних, автономна модель ADO.NET має документувати всі дії, які виконує над даними (об'єктами DataRow) користувач. Спеціально для цього в об'єкта DataRow є властивість RowState, що дозволяє дізнатися, що потрібно робити з рядком під час передавання відкладених змін.

Властивість RowState може мати елементи з переліку DataRowState.

Елементи переліку DataRowState такі:

1) Detached – рядок було створено, але він не є частиною будь-якої DataRowCollection. Об'єкт DataRow має цей стан відразу після свого створення й перед додаванням до колекції, а також якщо його було видалено з колекції;

2) Unchanged – рядок не було змінено з моменту останнього виклику AcceptChanges ();

3) Added – рядок було додано до колекції DataRowCollection і метод AcceptChanges () не було набрано;

4) Deleted – рядок було видалено за допомогою методу Delete () об'єкта DataRow;

5) Modified – рядок було змінено й метод AcceptChanges () не було набрано.

Для того щоб застосувати всі зміни, зроблені з рядками таблиці після передавання відкладених змін, об'єкт DataRow має метод AcceptChanges. Під час виклику методу AcceptChanges рядки, підготовлені до видалення, видаляють із колекції рядків таблиці, змінені й додані рядки стають Unchanged.

Метод AcceptChanges так само є в об'єктів DataSet і DataTable.

Під час виклику методу AcceptChanges на таблиці на всіх рядках цієї таблиці викликається метод AcceptChanges. Під час виклику методу AcceptChanges на DataSet викликається метод AcceptChanges на всіх таблицях цього DataSet.

На практиці іноді виникає потреба подивитися значення зміненого рядка до зміни або ж подивитися дані рядка, підготовленого до видалення. Для цього об'єкт DataRow має перевантаження індексатора, що має як другий індекс елемент переліку DataRowVersion.

Елементи переліку DataRowVersion такі:

1) Current – дозволяє дізнатися поточне значення рядка;

2) Original – дозволяє дізнатися початкове значення рядка;

3) Porposed – дозволяє дізнатися передбачуване значення поля (дійсне тільки під час редагування поля за допомогою методу BeginEdit ());

4) Default – версія за замовчуванням для рядка. Для значення DataRowState Added, Modified або Deleted версією за замовчуванням є Current. Для значення DataRowState Detached версією за замовчуванням є Proposed.

Контрольні запитання

1. Дайте опис класу DataSet. Для чого його використовують?
2. Які відмінності класу DataSet від реляційних баз даних?
3. На чому засновано об'єктну модель класу DataSet?
4. Для чого використовують класи DataColumn, DataRow і DataTable?

Рекомендована література: [1; 5; 6].

Використана та рекомендована література

1. Бакаєв А. А. Методи організації та обробки баз знань / А. А. Бакаєв. – Київ : Наукова думка, 2018. – 148 с.

2. Гайдаржи В. І. Бази даних в інформаційних системах / В. І. Гайдаржи, І. В. Ігор. – Київ : Університет "Україна", 2018. – 418 с.

3. Дейт К. Дж. Введення у системи баз даних / К. Дж. Дейт. 8-ме вид. – Київ : Діалектика, 2020. – 1328 с.

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

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

6. Оньон Ф. Основи ASP.NET із прикладами на C# / Ф. Оньон. – Київ : Діалектика, 2019. – 304 с.

7. Перевозчикова О. Л. Інформаційні системи і структури даних / О. Л. Перевозчикова. – Київ : Києво-Могилянська академія, 2017. – 288 с.

Зміст

НАВЧАЛЬНЕ ВИДАННЯ

Гордєєв Андрій Сергійович

ПРОЄКТУВАННЯ БАЗ ДАНИХ ТА БАЗ ЗНАНЬ

Конспект лекцій для студентів
спеціальності 186 "Видавництво та поліграфія" першого (бакалаврського) рівня

Самостійне електронне текстове мережеве видання

Відповідальний за видання *О. І. Пушкар*

Відповідальний редактор *О. С. Вяткіна*

Редактор *О. Г. Доценко*

Коректор *О. Г. Доценко*

План 2022 р. Поз. № 3-ЕК. Обсяг 180 с.

Видавець і виготовлювач – ХНЕУ ім. С. Кузнеця, 61166, м. Харків, просп. Науки, 9-А

Свідоцтво про внесення суб'єкта видавничої справи до Державного реєстру
ДК № 4853 від 20.02.2015 р.