Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут»

В. І. Гайдаржи
О. А. Дацюк

ОСНОВИ ПРОЕКТУВАННЯ ТА ВИКОРИСТАННЯ БАЗ ДАНИХ

2-ге видання, виправлене і доповнене

Рекомендовано Міністерством освіти і науки України як навчальний посібник для студентів вищих навчальних закладів

ПОЛІТЕХНІКА
НТУУ «КПІ»

Київ
2004

ФІРМА ПЕРІОДИКА

УДК 681.3.016
Г14

Гриф надано
Міністерством освіти і науки України
(лист № 14/18.2-520 від 20. 04 01)

Рецензенти: *В. Г. Писаренко*
Ю. І. Бадаєв
О. О. Гагарін

Гайдаржи В. І., Дацюк О. А.

Г14 Основи проектування та використання баз даних: Навч. посіб. – 2-ге вид., виправл. і доповн. – К.: ІВЦ "Видавництво «Політехніка»", ТОВ "Фірма «Періодика»", 2004. – 256 с.

ISBN 966-622-167-5

Розглянуто проектування баз даних для інформаційних систем. Подано теоретичні відомості про технології проектування баз даних для локальних та розподілених систем обробки інформації. Наведено основи реляційної алгебри, питання організації та нормалізації баз даних. Описано мову SQL – структуровану мову запитів до реляційних баз даних.

Для студентів вищих навчальних закладів.

УДК 681.3.016

ISBN 966-622-167-5

Передмова

Посібник присвячено розгляду базових питань проектування та використання баз даних для складних інформаційних систем. Наведено відомості про класичні та сучасні технології проектування баз даних та інформаційних систем. Особливу увагу приділено проектуванню реляційних баз даних.

Власне проектування складних інформаційних систем розглянуто стисло. Детальніше викладено проектування баз даних, на основі яких створюють такі системи. У першому розділі подано загальні відомості про інформаційні системи. Наведено їх класифікацію та розглянуто основні етапи проектування.

У другому розділі посібника описано взаємозв'язок інформаційних систем та відповідних баз даних, обґрунтовано потребу створювати системи саме на основі використання баз даних.

У третьому розділі розглянуто базові поняття систем керування базами даних (СКБД), їх основні функції, такі як керування даними, транзакціями, журналами змін тощо.

Четвертий розділ присвячено різним підходам до організації баз даних – ієрархічних та мережевих, наведено їх основні ознаки та відмінності.

У п'ятому розділі увагу приділено реляційним базам даних, подано основні визначення та загальну характеристику реляційної моделі даних.

У шостому розділі наведено теоретичну основу маніпулювання даними в реляційних базах даних – реляційну алгебру (алгебру Кодда). Розглянуто основні операції над елементами реляційної моделі – теоретико-множинні (операції об'єднання, перетину, різниці тощо) та реляційні (селекція, проекція, сполучення та ін.).

Сьомий розділ присвячено дуже важливим питанням оптимізації структури бази даних проведенням її нормалізації за певними правилами (формами нормалізації).

Восьмий розділ містить описання побудови семантичної моделі бази даних у процесі проектування інформаційної системи з метою побудови бази даних, структура якої як найкраще відповідає меті створення системи.

У дев'ятому розділі розглянуто особливості фізичної організації баз даних, тобто наведено відомості про спосіб зберігання відношень та індексів у зовнішній пам'яті, й організації доступу до них.

Десятий розділ присвячено описанню структурованої мови запитів до баз даних – мови SQL, зокрема визначення схеми баз даних, складних запитів, використання подань, тригерів, збережених процедур, керування транзакціями та підтримки деяких спеціальних функцій сервера баз даних.

В останньому, одинадцятому розділі розкрито питання проектування розподілених баз даних для систем з розподіленою обробкою інформації. Наведено основні принципи створення та функціонування систем із розподіленими базами даних, обробки даних за технологією *клієнт – сервер*. Розглянуто засоби проектування інформаційних систем із розподіленими базами даних із використанням мови UML.

Посібник розраховано на студентів вищих навчальних закладів за спеціальностями, які готують бакалаврів та спеціалістів у галузі конструювання комп'ютерних систем обробки інформації на основі реляційних баз даних. Автори посібника намагалися викласти навчальний матеріал мовою, доступною як для студентів вищих навчальних закладів, так і для широкого кола читачів, яких цікавлять питання проектуваннння та використання баз даних.

1. Інформаційні системи

1.1. Загальні положення

Традиційно *інформацією* прийнято називати відомості, що передаються людьми усною, письмовою або іншою формою.

Інформаційна система (ІС) слугує для збирання та накопичення інформації, її ефективного використання для різних цілей.

Автоматизація створення ІС відбувається за допомогою обчислювальної техніки. При цьому інформація подається у вигляді даних, які зберігаються у пам'яті ЕОМ. Отже, під час проектування ІС, з одного боку, вирішується питання про те, які відомості та для яких цілей будуть утримуватись у системі, з другого – як відповідні дані будуть організовані в пам'яті ЕОМ, як вони будуть підтримуватися та оброблятися під час експлуатації ІС.

Інформаційна система являє собою систему програмних, мовних, організаційних і технологічних засобів, призначених для централізованого накопичення та колективного використання даних.

Переважно ІС зберігають тим чи тим способом структуровані дані (бази даних (БД)). Але ІС може зберігати й неструктуровану інформацію (наприклад, документи вільного формату), однак її використання пов'язано з певними угодами та обмеженнями.

Основна функція ІС – це моделювання стану об'єктів частини реального світу, що розглядається (предметна область), та відображення зв'язків, які виникають між цими об'єктами під час розв'язання функціональних задач (проблемне середовище).

1.2. Класифікація інформаційних систем

Класифікація ІС за ознаками виконуваних функцій:

- системи керування базами даних (СКБД) – організація, збереження та поновлення БД;
- інформаційні системи обробки та накопичення (ІСОН) – це СКБД і засоби завантаження й актуалізації БД, а також засоби реалізації запитів і формування звітів;
- програмно-технічні комплекси (ПТК). Складаються з ІСОН та засобів автоматизації проектування БД і додатків користувача, наприклад CASE-засобів.

Приклади ПТК (як приклади типів ІС):

- САПР – системи автоматизованого проектування;
- АСУВ – автоматизовані системи управління виробництвом;

- АСУТП – автоматизовані системи керування технологічними процесами;
- АСНД – автоматизовані системи наукових дослідів.

Класифікація ІС за типом використання:

- локальні – ІС для одного користувача;
- мережеві – ІС колективного користування, характеризуються можливістю одночасного доступу до даних кількох користувачів; у свою чергу, поділяються на такі системи:
  - системи «файл/сервер» – у цьому випадку на окремій ЕОМ зберігається БД, а обробкою даних та їх поданням займаються прикладні системи;
  - системи «клієнт/сервер» – збереження та оброблення даних виконує сервер даних і додатків, а прикладні системи виконують тільки функції інтерфейсу між користувачем і сервером.

Класифікація ІС за типом завдань, які ними вирішуються:

Інформаційно-пошукові системи – зорієнтовані, як правило, на знаходження даних, які задовольняють вказаний критерій пошуку. Такі системи характеризуються потужними засобами реалізації запитів. Прикладом є довідкова служба міста, в якій зберігаються відомості про мешканців.

Системи цільової обробки та аналізу даних. Дані в таку систему, як правило, надходять від датчиків або інших ІС. Завдання ІС – аналіз цих даних, їх обробка та формування звітів. Вони характеризуються, переважно, потужними обчислювальними засобами. Прикладом є ІС керування оранжереєю або ІС обслуговування банку.

Фактографічні системи – зберігають відомості про об'єкти предметної області (ПО) (підприємства, підрозділи та ін.). Дані про кожен об'єкт, наприклад про співробітника, можуть надходити в систему з багатьох джерел. Основним завданням таких систем є підтримка актуальності БД та видача звітів про поточний стан об'єктів.

Документальні системи, в яких об'єкт зберігання – документи, тому що саме документи накопичуються і обробляються. Для обробки інформації не важливо, які відомості містяться в документах (наприклад, про витрати пального чи перевезені вантажі), у документальній системі можна знайти середнє значення, максимальне значення або суму за вказаним стовпчиком у всіх документах певної форми тощо.

Документально-фактографічні системи поєднують у собі характеристики класів ІС, описаних вище.

1.3. Користувачі інформаційних систем

Користувачів ІС умовно можна поділити на дві групи: внутрішні та кінцеві.

Внутрішні користувачі розробляють ІС та підтримують їх функціонування, кінцеві – це ті, заради яких створюються ІС, вони їх експлуатують.

Групу внутрішніх користувачів складають: адміністратор БД, адміністратор функціональних підсистем, системні та прикладні програмісти.

Функції адміністратора БД на стадії розробки та експлуатації ІС є різними і тому виконуються різними особами.

На стадії проектування адміністратор БД є ідеологом і конструктором системи, керує роботами зі створення програмного оточення БД.

На стадії експлуатації адміністратор БД – особа, відповідальна за функціонування ІС; вона керує режимом користування даними. Основні завдання адміністратора БД під час експлуатації – захист даних від руйнування, забезпечення достовірності, аналіз ефективності використання ресурсів ІС.

Адміністратор функціональних підсистем разом з адміністратором БД розробляють програмні «фільтри» для користувачів. Крім цього, адміністратори функціональних підсистем визначають алгоритми обробки даних, необхідних для проектування ІС.

Системні програмісти виконують генерацію СКБД, стежать за її функціонуванням у середовищі операційної системи, розробляють згідно з завданням адміністратора БД програмні компоненти, які розширюють програмне забезпечення СКБД.

Завдання прикладних програмістів полягає у розробці прикладних програм. Для цього їм необхідні знання алгоритмічних мов і мовних засобів СКБД.

Кінцевих користувачів, у свою чергу, поділяють на прямих і непрямих.

Непрямі кінцеві користувачі не спілкуються з ЕОМ безпосередньо. Вони надсилають свої запити службі адміністратора БД, а потім отримують відповідь на папері, яку до передачі замовнику інтерпретують фахівці. Слід зазначити, що з постійним розвитком засобів телекомунікацій та застосуванням розподілених БД кількість непрямих користувачів постійно зменшується.

Загальну схему використання ІС зображено на рис. 1.1.

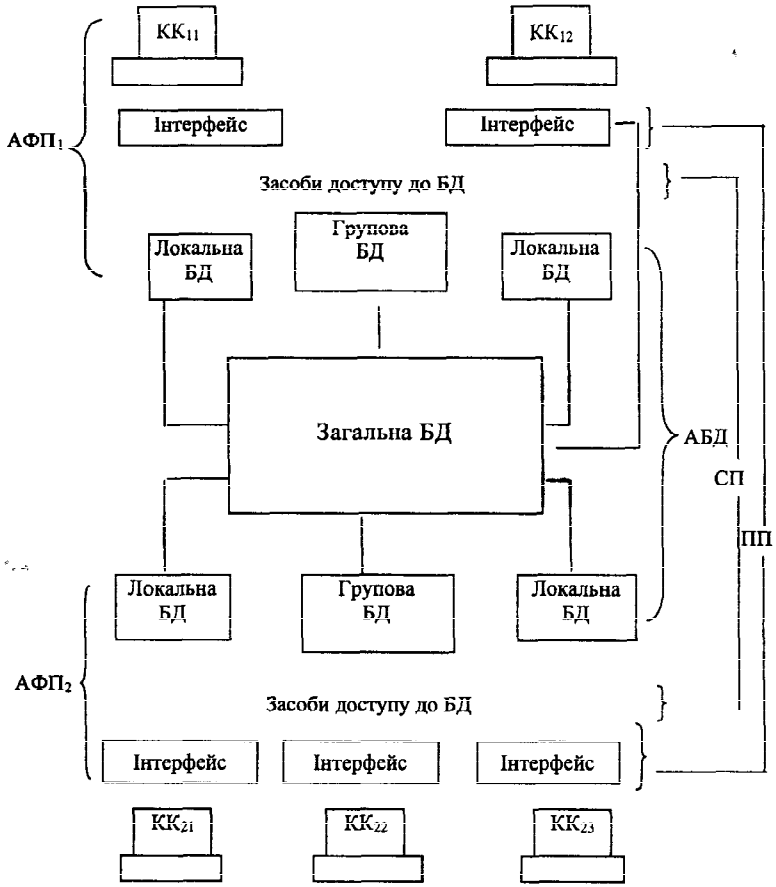


Рис. 1.1. Загальна схема використання ІС:
$КК_{ij}$ – j-й користувач i-ї системи; АБД – адміністратор БД; $АФП_i$ – адміністратор i-ї функціональної підсистеми; СП – системний програміст; ПП – прикладний програміст

Прямі кінцеві користувачі спілкуються з ІС в інтерактивному режимі. Деякі з них уміють користуватися існуючими програмними додатками та інтерпретувати відповіді ІС. Інші вміють самостійно розробляти нові додатки.

Сучасна тенденція розвитку СКБД полягає в розвитку мовних і програмних засобів, зорієнтованих на кінцевих користувачів, зокрема на тих, які готові розробляти нові додатки самостійно, не користуючись послугами прикладних програмістів.

1.4. Архітектура інформаційних систем

Особливість ІС – підтримка різноманітних уявлень користувачів про ІС.

Для кінцевих користувачів ІС – це сховище відомостей про мешканців та рахунки, угоди та поставки, документи і ціни. Але для внутрішніх користувачів ІС представляється у вигляді елементів даних, записів, сторінок, файлів.

Уявлення внутрішніх користувачів про систему не є однаковими. Прикладний програміст оперує елементами даних, записами, ключами, структурними асоціаціями, але він, як правило, не має уявлення про фізичну організацію даних, яка є прерогативою адміністратора та системних програмістів. Іншими словами, у сучасній ІС існує декілька рівнів подання даних. Їх різновиди прийнято асоціювати з поняттям архітектури ІС. На рис. 1.2 зображено рівні абстракції подання даних в ІС. Розрізняють: локальне, концептуальне, рівні формалізованого подання, фізичного зберігання та зовнішнього подання.

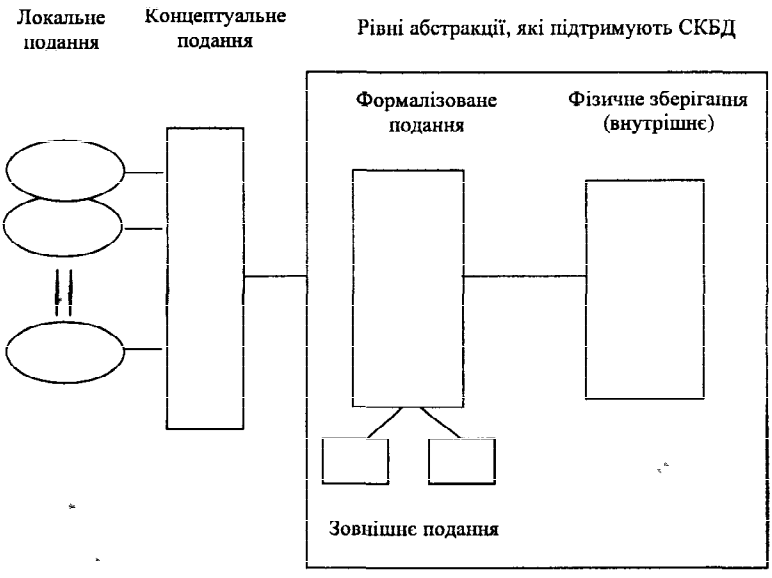


Рис. 1.2. Рівні абстракції подання даних в ІС

Під час аналізу та проектування ІС розглядають такі рівні абстракції:

- *локальне подання* – відповідає уявленням про ПО кінцевих користувачів – назвемо їх локальними уявленнями користувача (ЛУК);

- *концептуальне подання* являє собою інтеграцію ЛУК, що відповідають «погляду» на ПО її адміністратора (директора, міністра, ректора вищого навчального закладу тощо). Він володіє всією множиною інформаційних об'єктів і всіма можливими асоціаціями між ними, тоді як кожен кінцевий користувач переглядає лише обмежений фрагмент ПО. Відзначимо, що концептуальне уявлення про ПО існує поза яким-небудь зв'язком із засобами реалізації ІС. Воно представляє лише інформаційні потреби системи, яка розробляється та відображає особливості ПО, але не зачіпає питання, як відповідні дані будуть представлені у пам'яті ЕОМ;
- *формалізоване подання* – опис БД на рівні задається мовою опису даних, який відповідає уявленню про логічну організацію даних адміністратора БД. Цей рівень абстракції дуже схожий на концептуальний, але його відмінність полягає у прив'язці до засобів реалізації – СКБД. Опис БД на концептуальному рівні задається мовою опису даних, яку використовує СКБД, у термінах та обмеженнях, прийнятих у цій системі. Оскільки кожна СКБД на концептуальному рівні підтримує свою модель даних, то перетворення концептуального опису ПО на формалізований опис для БД різних систем приводить до різних результатів.
- *фізичне зберігання* – реалізація подання даних у пам'яті ЕОМ та організації їх зберігання. Зазначимо, що параметри внутрішнього рівня подання БД впливають на ефективність ІС, наприклад, на ємність зовнішньої пам'яті, час реакції системи. У процесі функціонування ІС адміністратор може змінювати внутрішній опис БД, залишаючи без змін описи, які задаються на інших рівнях. Це дозволить змінювати параметри ефективності системи, не змінюючи розроблені додатки;
- *зовнішнє подання* – уявлення користувача про дані, тобто спосіб, в який інформація надається користувачу засобами СКБД (відповідають початковим ЛУК).

На кожному рівні абстракції визначається своя модель ПО. Опис цих моделей прийнято називати *схемами*, тому кажуть, що схеми ЛУК відображаються в концептуальну схему ПО, остання відображається у формалізовану схему БД тощо.

Три рівні абстракції (концептуальний, зовнішній та фізичний) були запропоновані робочою групою CODASYL ще у 1971 р. Вони є центральною ідеєю звіту із СКБД дослідної групи Американського інституту стандартів ANSI/SPARC, який було опубліковано у 1975 р.

1.5. Життєвий цикл інформаційних систем

Як і будь-який інший продукт, ІС проходить певний життєвий цикл, котрий починається з рішення почати створення системи і закінчується припиненням експлуатації внаслідок морального старіння (фізичний знос для ІС є неактуальним). Життєвий цикл ІС містить декілька обов'язкових етапів:

- проектування (визначення та аналіз вимог);
- реалізація (програмний продукт);
- експлуатація та супроводження.

Проектування. Проектування виконують за допомогою вивчення ПО та вимог, які ставляться до ІС. На цій «паперовій» стадії життя системи вибирають:

- структуру даних і стратегію їх зберігання у пам'яті ЕОМ;
- технологію обслуговування ІС та взаємодію з нею кінцевих користувачів;
- технічні та стандартні програмні засоби, а також розробку оригінальних програмних засобів обслуговування ІС.

Реалізація. На цій стадії розробляють і налагоджують програмне забезпечення ІС, створюють первинну БД, розробляють необхідні програмні додатки. На стадії реалізації перевіряють і коригують технологію обслуговування ІС.

Експлуатація. Починається з наповнення системи реальною інформацією. Експлуатація охоплює весь комплекс дій для підтримки функціонування ІС, тобто: ведення словника-довідника даних, забезпечення захисту даних, організація колективного використання даних, аналіз і керування ефективністю системи тощо.

Крім цього, стадія експлуатації містить у собі розробку нових додатків, а також удосконалення та подальший розвиток ІС.

Існуючі моделі життєвого циклу відрізняються структурою та конкретним змістом етапів створення та впровадження. Від вибраної моделі життєвого циклу залежить, наскільки довго можна буде підтримувати працездатність ІС. Усі існуючі моделі являють собою спектр, на протилежних кінцях якого знаходяться так звані *каскадна* та *спіральна* моделі.

Каскадна модель характеризується суворою впорядкованістю стадій, з яких складаються етапи створення та впровадження. Така впорядкованість потребує досить ретельного виконання робіт, передбачених на кожній стадії, для того, щоб не виникало необхідності переглядати раніше прийняті рішення. На рис. 1.3 зображено каскадну модель життєвого циклу ІС.

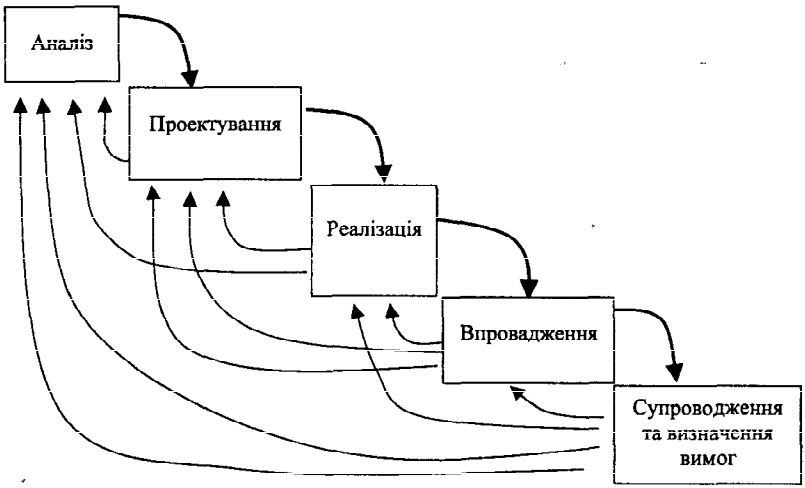


Рис. 1.3. Каскадна модель життєвого циклу ІС

Спіральна модель передбачає багаторазове проходження стадій розробки доти, доки отриманий продукт не буде повністю задовольняти замовника. Ця модель відображає ітераційний характер процесу проектування. На кожній ітерації створюється діючий прототип, який оцінюється, і на підставі цієї оцінки приймають рішення щодо подальшого удосконалення. На рис. 1.4 зображено спіральну модель життєвого циклу ІС.

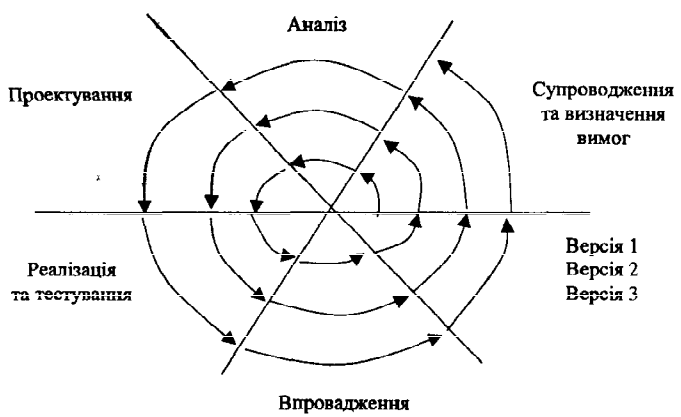


Рис. 1.4. Спіральна модель життєвого циклу ІС

Перевага каскадної моделі – її детермінованість і чітка регламентація робіт, що спрощує управління проектом. Її недоліком є те, що від затвердження технічного завдання до впровадження готового продукту проходить дуже багато часу. Існує ризик, що до цього часу реальні потреби користувача можуть змінитися і система повністю перестане задовольняти його вимоги. Крім цього, сам користувач у більшості випадків не може правильно сформулювати вимоги, поки сам не почне працювати з системою.

Спіральна модель – навпаки, вільна від цього недоліку, оскільки на кожному витку спіралі є можливість змінити проект з метою досягнення відповідності новим вимогам користувача. До недоліків такої моделі належить практична неможливість планування та контролю виконання проекту, оскільки завчасно невідомо, скільки витків спіралі буде потрібно для отримання остаточної версії системи. Тому не можна точно оцінити затрати на розробку.

Отже, в першому випадку замовник може проконтролювати строки виконання та якість отриманого продукту, визначити затрати та результат. Однак при цьому існує ризик, що створена система хоч і повністю відповідає розробленим спочатку вимогам, не відповідає реальним вимогам користувача. У другому випадку в результаті користувач все ж таки отримує таку систему, яка йому потрібна, але невідомо, скільки на це буде потрібно часу та ресурсів.

2. Інформаційні системи і бази даних

Інформаційні системи переважно зорієнтовані на зберігання, вибір і модифікацію інформації, яка постійно змінюється. Структура інформації зазвичай дуже складна, і хоча структури даних різноманітні в різних ІС, між ними є багато спільного. На початковому етапі використання обчислювальної техніки проблеми структуризації даних вирішувалися індивідуально в кожній ІС, але ці індивідуальні засоби керування даними складали істотну частину ІС, практично повторюючись (як програмні компоненти) від однієї системи до іншої. Прагнення виділити загальну частину ІС, відповідальну за керування структурованими даними, було причиною створення ряду СКБД, які складаються з бібліотеки програм, доступних кожній ІС.

Однак дуже скоро виявилось, що неможливо обійтися такою загальною бібліотекою програм, яка реалізує над стандартною базовою файловою системою більш складні засоби зберігання та оброблення даних. Поясним це на прикладі. Припустимо, що потрібно реалізувати просту ІС, яка підтримує облік співробітників деякої організації. Система має видавати списки співробітників згідно з означеними номерами відділів, підтримувати реєстрацію переходу співробітників з одного відділу до іншого, прийняття на роботу нових співробітників і звільнення тих, хто працює. Для кожного відділу має підтримуватися можливість одержання характеристик відділу, наприклад: імені керівника цього відділу, загальної чисельності відділу, загальної суми останньої зарплати тощо. Для кожного співробітника має існувати можливість видачі номера посвідчення за його повним іменем, надання повного імені за номером посвідчення, отримання інформації про поточну відповідність займаній посаді співробітника і про розмір зарплати.

Припустимо, що необхідно реалізувати цю ІС на основі файлової системи і користуватися при цьому одним файлом, розширивши базові можливості файлової системи за рахунок спеціальної бібліотеки функцій. Оскільки мінімальною інформаційною одиницею у цьому випадку є співробітник, природно зажадати, щоб у цьому файлі містився один запис для кожного співробітника. Очевидно, що поля таких записів мають містити такі атрибути (характеристики) співробітника: повне ім'я співробітника (СПІВ_ІМ'Я), номер його посвідчення (СПІВ_НОМЕР), інформацію про його відповідність займаній посаді (СПІВ_СТАТ – для простоти, «так» або «ні»), розмір зарплати (СПІВ_ЗАРП), номер відділу (СПІВ_ВІД_НОМЕР). Оскільки слід обмежитися одним файлом, цей самий запис має містити ім'я керівника відділу (СПІВ_ВІД_КЕР).

Для виконання вказаних функцій від ІС вимагається можливість реалізації доступу до цього файлу за унікальними ключами (атрибутами, які

визначають запис), а саме: СПІВ_ІМ'Я і СПІВ_НОМЕР. Крім того, має забезпечуватися можливість вибору всіх записів за заданим значенням атрибута СПІВ_ВІД_НОМЕР, тобто доступ за неунікальним ключем. Щоб отримати чисельність відділу або загальний розмір зарплати, ІС має кожного разу вибирати всі записи про співробітників відділу і підраховувати відповідні загальні значення.

Отже, під час реалізації навіть такої простої системи на базі файлової системи виникають певні проблеми:

- створення достатньо складної надбудови, що забезпечує багатоключовий доступ до файлів;
- дублювання даних, які зберігаються (для кожного співробітника відділу повторюється ім'я керівника відділу);
- виконання масової вибірки і обчислень для отримання зведеної інформації про відділи;
- необхідність повного перегляду файлу в разі необхідності вибірки за неключовим атрибутом, наприклад, якщо потрібно видати список співробітників, що отримують задану зарплату.

Можливе вирішення подібних проблем – почати підтримувати два багатоключові файли СПІВРОБІТНИКИ і ВІДДІЛИ: перший файл має містити поля: СПІВ_ІМ'Я, СПІВ_НОМЕР, СПІВ_СТАТ, СПІВ_ЗАРП і СПІВ_ВІД_НОМЕР, а другий – ВІД_НОМЕР, ВІД_КЕР, СПІВ_ЗАРП (загальний розмір зарплати) і ВІД_РОЗМІР (загальна кількість співробітників у відділі). Тоді можна уникнути більшості зазначених проблем. Кожний з файлів буде містити тільки недубльовану інформацію, необхідність у динамічних обчисленнях зведеної інформації не виникне. Але зазначимо, що після такого переходу ІС буде мати деякі нові можливості, що зближають її з СКБД.

Передусім, система має знати, що вона працює з двома інформаційно зв'язаними файлами (це крок у бік схеми БД), їй мають бути відомі структура і сенс кожного поля, а також розуміти, що в ряді випадків зміна інформації в одному файлі має зумовити модифікацію другого файлу, щоб загальний уміст файлів був погодженим.

Взагалі, погодженість даних є ключовим поняттям БД. Насправді, якщо ІС (навіть така проста, як у прикладі) підтримує погоджене зберігання інформації в декількох файлах, можна говорити про те, що вона підтримує БД. Якщо ж деяка допоміжна система керування даними дозволяє працювати з декількома файлами, забезпечуючи їхню погодженість, можна назвати її СКБД. Вимога підтримання погодженості даних у декількох файлах не дозволяє обійтися лише бібліотекою функцій: така система має володіти деякими власними даними (метаданими) і навіть знаннями, які

визначають цілісність даних. СКБД має також дозволяти сформулювати такий запит доступною для користувачів мовою. Такі мови називають *мовами запитів до баз даних*. Наприклад, мовою SQL цей запит можна було записати у такому вигляді:

```
SELECT ВІД_РОЗМІР
FROM СПІВРОБІТНИКИ, ВІДДІЛИ
WHERE СПІВ_ІМ'Я="ІВАН АРУТЮНОВИЧ ШЕПИТЬКО"
AND СПІВ_ВІД_НОМЕР=ВІД_НОМЕР
```

Цей запис означає:

Вибрати значення атрибутів ВІД_РОЗМІР

із записів файлів СПІВРОБІТНИКИ, ВІДДІЛИ

які задовольняють умову СПІВ_ІМ'Я="ІВАН АРУТЮНОВИЧ ШЕПИТЬКО"

та СПІВ_ВІД_НОМЕР=ВІД_НОМЕР

СКБД, які підтримують SQL-запити забезпечують збереження даних у вигляді спеціальних структур – відношень (RELATIONS). Відношення відповідає уявленню користувачів про опис сукупності однорідних об'єктів ПО як про окрему таблицю (наприклад, відношенню "СТУДЕНТ" відповідає таблиця з інформацією про студентів). Такі СКБД отримали загальну назву – *реляційні*.

Під час формування запиту до СКБД користувач не повинен знати, як буде виконуватися цей запит. Серед її метаданих буде міститися інформація про те, що поле СПІВ_ІМ'Я є ключовим для файлу СПІВРОБІТНИКИ, а ВІД_НОМЕР – для файлу ВІДДІЛИ, і система сама цим скористається. Якщо виникне потреба в отриманні списку співробітників, які не відповідають займаній посаді, то достатньо подати системі запит:

```
SELECT СПІВ_ІМ'Я, СПІВ_НОМЕР
FROM СПІВРОБІТНИКИ
WHERE СПІВ_СТАТ ='НІ'
```

Система виконає необхідний повний перегляд файлу СПІВРОБІТНИКИ, оскільки поле СПІВ_СТАТ не є ключовим.

Далі, уявімо, що на початку реалізації ІС, що базується на використанні бібліотек розширених засобів доступу до файлів, обробляється операція реєстрації нового співробітника. Для погодженої зміни файлів припустимо, що ІС уже вставила новий запис у файл СПІВРОБІТНИКИ і зібралася модифікувати запис файлу ВІДДІЛИ, але саме в цей момент відбулося аварійне вимкнення електричного живлення. Очевидно, що після перезапуску системи її БД буде порушено. Потрібно виконувати додаткові дії щодо приведення її в цілісний стан. Сучасні СКБД беруть таку роботу

на себе. Прикладна система зобов'язана знати, який стан даних є коректним, але всю технічну роботу виконує СКБД.

Нарешті, уявімо, що необхідно забезпечити паралельну (наприклад, багатотермінальну) роботу з БД співробітників. Якщо спиратися тільки на використання файлів, то для забезпечення коректності змін на весь час модифікації будь-якого з двох файлів доступ інших користувачів до цього файлу буде блоковано (згадаємо можливості файлових систем для синхронізації паралельного доступу). Отже, зарахування на роботу Петра Івановича Сидорова істотно загальмує отримання інформації про співробітника Івана Сидоровича Петрова, навіть якщо вони будуть працювати в різних відділах. Сучасні СКБД забезпечують значно тоншу синхронізацію паралельного доступу до даних.

Отже, СКБД вирішують більшість проблем, які важко або взагалі неможливо вирішити під час використання файлових систем. При цьому існують додатки, для яких цілком достатньо файлів; додатки, для яких необхідно вирішувати, який рівень роботи з даними у зовнішній пам'яті їм потрібен; і додатки, для яких безумовно потрібні БД.

Сучасні системи керування файлами і базами даних являють собою надто досконалі інструменти, кожний з яких можна дуже успішно застосовувати у відповідній сфері діяльності. Але завжди необхідно пам'ятати, що кожний інструмент приносить максимальну користь саме в тій сфері, для якої його створено.

3. Функції і структура систем керування базами даних

Традиційних можливостей файлових систем виявляється недостатньо для побудови навіть простих ІС. Вище наведено опис потреб, що не перекриваються можливостями систем керування файлами: підтримання логічно погодженого набору файлів, забезпечення мови маніпулювання даними, відновлення інформації після її порушень, реальна паралельна робота декількох користувачів. Можна вважати, що якщо прикладна ІС спирається на деяку систему керування даними, яка має ці властивості, то ця система є СКБД.

3.1. Основні функції систем керування базами даних

3.1.1. Безпосереднє керування даними у зовнішній пам'яті

Ця функція включає забезпечення необхідних структур зовнішньої пам'яті як для зберігання даних, що входять у БД, так і для службових цілей, наприклад, для забезпечення швидкого доступу до даних у деяких випадках (для цього використовують індекси). У деяких реалізаціях СКБД активно використовують тільки можливості існуючих файлових систем, в інших робота проходить на рівні приладів зовнішньої пам'яті. Але підкреслимо, що в розвинених СКБД користувачі у будь-якому випадку не зобов'язані знати, чи використовує СКБД файлову систему, а якщо використовує, то як організовано файли. Зокрема, СКБД підтримує власну систему іменування об'єктів БД (це дуже важливо, оскільки імена об'єктів БД відповідають іменам об'єктів ПО).

Існує багато різних засобів організації зовнішньої пам'яті БД. Як і всі рішення, що приймають під час організації БД, конкретні засоби організації зовнішньої пам'яті необхідно вибирати у тісному зв'язку зі всіма іншими рішеннями.

3.1.2. Керування буферами оперативної пам'яті

Як правило, СКБД працюють з БД великого розміру; принаймні цей розмір зазвичай істотно перевищує доступну ємність оперативної пам'яті. Зрозуміло, якщо у разі звернення до будь-якого елемента даних буде проходити обмін із зовнішньою пам'яттю, то вся система буде працювати зі швидкістю обміну із зовнішньою пам'яттю. Єдиним засобом реального збільшення цієї швидкості є буферизація даних в оперативній пам'яті. І навіть якщо операційна система виробляє загальносистемну буферизацію (як у випадку ОС UNIX), цього недостатньо для СКБД, яка має значно більшу інформацію про корисність буферизації тієї або іншої частини БД. Тому в розвинених СКБД підтримується власний набір буферів оперативної пам'яті з власною дисципліною заміни буферів. Під час керування буфера-

ми основної пам'яті потрібно розробляти і застосовувати погоджені алгоритми буферизації, журналізації та синхронізації. Відзначимо, що існує окремий напрям СКБД, зорієнтований на постійну наявність в оперативній пам'яті всієї БД. Такі СКБД передбачають, що в майбутньому ємність оперативної пам'яті комп'ютерів збільшиться.

3.1.3. Керування транзакціями

Транзакція – це послідовність операцій над БД, які розглядаються СКБД як єдине ціле. Після початку виконання транзакції можливі дві ситуації: або транзакція успішно виконується, і в цьому разі СКБД фіксує (COMMIT) зміни в БД, здійснені транзакцією, або виникла аварійна ситуація під час її виконання, у цьому разі СКБД жодну з таких змін ніяк не фіксує, тобто БД не змінюється. Поняття транзакції необхідно для підтримання логічної цілісності БД. Якщо згадати приклад ІС відділу кадрів з файлами СПІВРОБІТНИКИ і ВІДДІЛИ, то єдиним засобом не порушити цілісність БД під час виконання операції прийняття на роботу нового співробітника буде об'єднання елементарних операцій над файлами СПІВРОБІТНИКИ і ВІДДІЛИ в одну транзакцію. Отже, підтримання механізму транзакцій – обов'язкова умова СКБД (якщо, звичайно, така система заслуговує назви СКБД). Але поняття транзакції є значно складнішим для колективних СКБД. Той факт, що кожна транзакція починається при цілісному стані БД і залишає цей стан цілісним після завершення, робить дуже корисним використання поняття транзакції як одиниці активності користувача відносно БД. При відповідному керуванні кожний користувач може, як правило, відчувати себе єдиним користувачем СКБД (користувачі колективних СКБД часом можуть відчути присутність своїх колег).

З керуванням транзакціями у колективних СКБД пов'язані важливі поняття серіалізації транзакцій і серіального плану виконання суміші транзакцій. Під *серіалізацією транзакцій*, які виконуються паралельно, розуміють такий порядок планування їх роботи, за якого сумарний ефект суміші транзакцій дорівнює ефекту їх послідовного виконання. *Серіальний план виконання суміші транзакцій* – це такий засіб їх спільного виконання, який приводить до серіалізації транзакцій. Зрозуміло, що якщо вдається досягнути справді серіального виконання суміші транзакцій, то для кожного користувача, з ініціативи якого утворено транзакцію, наявність інших транзакцій буде непомітною (якщо не враховувати деяке уповільнення роботи для кожного користувача порівняно з режимом роботи систем одного користувача).

Існує декілька базових алгоритмів серіалізації транзакцій. У серіалізованих СКБД найбільш розповсюдженими є алгоритми, що базуються на синхронізаційних захопленнях об'єктів БД. Під час використання будь-якого алгоритму серіалізації можливі ситуації конфліктів між двома або більше транзакціями щодо доступу до

більше транзакціями щодо доступу до об'єктів БД. У цьому випадку для підтримання серіалізації необхідно виконати відкат (ліквідувати всі зміни, проведені в БД) однієї або більше транзакцій. Це один з тих випадків, коли користувач колективної СКБД може реально (і достатньо неприємно) відчути наявність у системі транзакцій інших користувачів.

3.1.4. Журнали змін

Одна з основних вимог до СКБД – надійне зберігання даних у зовнішній пам'яті. Під надійністю зберігання розуміють те, що СКБД має бути спроможна відновити останній погоджений стан БД колективного користування після будь-якої апаратної або програмної відмови. Зазвичай розглядають два можливі типи апаратних відмов: так звані *м'які відмови*, які можна трактувати як раптове зупинення роботи комп'ютера (наприклад, аварійне вимкнення живлення), та *жорсткі відмови*, що характеризуються втратою інформації на носіях зовнішньої пам'яті. Прикладами програмних відмов можуть бути аварійне завершення роботи СКБД через наявність помилки у програмі або аварійне завершення програми, в результаті чого деяка транзакція залишається незавершеною. У разі відмови системи потрібна ліквідація наслідків тільки однієї незавершеної транзакції.

Але в будь-якому випадку для відновлення БД потрібно володіти деякою додатковою інформацією. Підтримання надійного зберігання даних у БД вимагає зберігання надлишкових даних, причому та їхня частина, що використовується для відновлення, має зберігатися особливо надійно. Найбільш розповсюджений засіб підтримання такої надлишкової інформації – ведення журналу змін БД.

Журнал змін – це особлива частина БД, недосяжна для користувачів СКБД. Журнал підтримується особливо ретельно (інколи підтримуються дві копії журналу, які розташовують на різних фізичних дисках), сюди надходять записи про всі зміни в основній частині БД. У різних СКБД зміни в БД журналізуються на різних рівнях: інколи запис у журналі відповідає деякій логічній операції зміни БД (наприклад, операції вилучення рядка з таблиці реляційної БД), а часом запис відповідає мінімальній внутрішній операції модифікації сторінки зовнішньої пам'яті. У деяких системах водночас використовуються обидва підходи.

Під час ведення журналу змін дотримуються стратегії «попереджувального» запису в журнал (так званого протоколу Write Ahead Log – WAL). Ця стратегія полягає в тому, що запис про зміну будь-якого об'єкта БД має потрапити до журналу раніше, ніж змінений об'єкт буде записано до БД. Відомо, що якщо в СКБД коректно дотримується протокол WAL, то за допомогою журналу можна вирішити всі проблеми відновлення БД після будь-якої відмови.

Найпростішою ситуацією відновлення БД є індивідуальний відкат транзакції. Для нього не потрібний загальносистемний журнал змін БД. Достатньо для кожної транзакції підтримувати локальний журнал операцій модифікації БД, виконаних у цій транзакції, і виробляти відкат транзакції виконанням зворотних операцій, починаючи від кінця локального журналу. У деяких СКБД так і роблять, але в більшості систем локальні журнали не підтримують, а індивідуальний відкат транзакції виконують за загальносистемним журналом, для чого всі записи від однієї транзакції зв'язують зворотним списком (від кінця до початку).

У разі м'якої відмови у зовнішній пам'яті основної частини БД можуть знаходитися об'єкти, що модифікувалися транзакціями, які не закінчилися на час відмови, і може не бути об'єктів, що модифікувалися транзакціями, які на момент часу відмови успішно завершилися (завдяки використанню буферів оперативної пам'яті, уміст яких у разі м'якої відмови втрачається). Якщо дотримуватися протоколу WAL, у зовнішній пам'яті журналу мають гарантовано знаходитися записи, що стосуються операцій модифікації обох виглядів об'єктів. Метою процесу відновлення після м'якої відмови є стан зовнішньої пам'яті основної частини БД, в якому б вона перебувала в разі фіксації в зовнішній пам'яті змін усіх завершених транзакцій і який не містив би жодних слідів незавершених транзакцій. Щоб цього досягти, спочатку виробляють відкат незавершених транзакцій (*undo*), а потім повторно відтворюють (*redo*) операції тих завершених транзакцій, результати яких не відображені в зовнішній пам'яті. При цьому виникають труднощі, пов'язані із загальною організацією керування буферами і журналом.

Для відновлення БД після жорсткої відмови використовують журнал і архівну копію БД. *Архівна копія* – це повна копія БД на час початку заповнення журналу (є багато варіантів більш гнучкого тлумачення сенсу архівної копії). Звичайно, для нормального відновлення БД після жорсткої відмови необхідно, щоб журнал не знищився. Як уже зазначалося, до цілісності журналу в зовнішній пам'яті в СКБД висуваються особливо підвищені вимоги. Тоді відновлення БД полягає в тому, що виходячи з архівної копії, за журналом відтворюється робота всіх транзакцій, які закінчилися на час відмови.

3.1.5. Лінгвістичне оточення баз даних

Для роботи з БД використовують спеціальні мови. У ранніх СКБД підтримувалося декілька спеціалізованих за своїми функціями мов. Найчастіше виділялися дві: мова визначення схеми БД (SDL – Schema Definition Language) і мова маніпулювання даними (DML – Data Manipulation Language). SDL переважно використовували для визначення логічної структури БД, тобто структури БД, якою вона представляється користувачам.

DML містила набір операторів маніпулювання даними, тобто операторів, що дозволяють заносити дані в БД, вилучати, модифікувати або вибирати існуючі дані.

У сучасних СКБД зазвичай підтримується єдина інтегрована мова, що містить усі необхідні засоби для роботи з БД, починаючи від її створення, а також інтерфейс користувача, який забезпечує операції над БД. Стандартною мовою найбільш розповсюджених сучасних реляційних СКБД є мова SQL (Structured Query Language).

Передусім, мова SQL поєднує якості SDL і DML, тобто дозволяє визначати схему реляційної БД і маніпулювати даними. При цьому назви об'єктів БД (для реляційної БД – назви таблиць і їх стовпчиків) підтримуються на мовному рівні в тому сенсі, що компілятор мови SQL виробляє перетворення імен об'єктів на внутрішні ідентифікатори на підставі спеціальних службових таблиць-каталогів. Внутрішня частина СКБД (ядро) взагалі не працює з іменами таблиць і їх стовпчиків.

Мова SQL містить спеціальні засоби визначення обмежень цілісності БД. Обмеження цілісності зберігаються у спеціальних таблицях-каталогах. Забезпечення контролю цілісності БД виробляється на мовному рівні, під час компіляції операторів модифікації БД. Компілятор SQL на підставі наявних у БД обмежень цілісності генерує відповідний програмний код.

Спеціальні оператори мови SQL дозволяють визначати так звані представлення БД, які фактично є запитами, що зберігаються в БД (тому що результатом будь-якого запиту до реляційної БД є таблиця) з іменованими стовпчиками. Для користувача представлення є такою самою таблицею, як і будь-яка базова таблиця, що зберігається в БД, але за допомогою представлень можна обмежити або навпаки розширити вміст БД для конкретного користувача. Підтримання представлень виконується також на мовному рівні.

Нарешті, обмеження доступу до об'єктів БД виробляється на основі спеціального набору операторів SQL. Ідея полягає в тому, що для виконання операторів SQL різного вигляду користувач повинен мати різноманітні повноваження. Користувач, що створив таблицю БД, має повний набір повноважень для роботи з нею. До цього набору входить повноваження на передачу всіх або частини повноважень іншим користувачам. Повноваження користувачів описуються у спеціальних таблицях-каталогах, контроль повноважень підтримується на мовному рівні.

3.2. Типова організація сучасних систем керування базами даних

Природно, що організація типової СКБД і склад її компонентів відповідає набору основних функцій СКБД:

- керування даними в зовнішній пам'яті;

- керування буферами оперативної пам'яті;
- керування транзакціями;
- ведення журналів змін і відновлення БД після відмов;
- підтримання мов БД.

У сучасній реляційній СКБД можна виділити:

- внутрішню частину – ядро СКБД (часто його називають Data Base Engine);
- компілятор мови БД (зазвичай SQL);
- підсистему підтримки часу виконання;
- набір утиліт.

У деяких системах ці частини виділяються явно, в інших – неявно, але такий логічний розподіл можна здійснити для усіх СКБД.

Ядро СКБД відповідає за керування даними в зовнішній пам'яті, керування буферами оперативної пам'яті, керування транзакціями і ведення журналів змін. Відповідно можна виділити такі компоненти ядра (виділити логічно, хоча в деяких системах ці компоненти виділяють явно): менеджер даних, менеджер буферів, менеджер транзакцій і менеджер журналу. Функції цих компонентів взаємопов'язані. Для забезпечення коректної роботи СКБД всі ці компоненти мають взаємодіяти за перевіреними протоколами. Ядро СКБД має власний інтерфейс, який є недоступним для користувачів. Він використовується у програмах, які створюються компілятором SQL (або в підсистемі підтримки виконання таких програм), і утилітах БД. Ядро СКБД є основною резидентною частиною СКБД. У разі використання архітектури «клієнт/сервер» ядро є основною складовою сервера системи.

Головна функція компілятора мови БД – компіляція операторів мови в програму, що виконується.

Основною проблемою реляційних СКБД є те, що мови цих систем (а це, як правило, SQL) є непроцедурними. В операторах таких мов визначається деяка дія над БД, але ця специфікація не є процедурою, вона лише описує в деякій формі умови виконання бажаної дії. Тому компілятор має вирішити, як виконувати оператор мови, перш ніж виробити програму. Застосовуються достатньо складні засоби оптимізації операторів. Результатом компіляції є програма, що виконується. Вона представляється в деяких системах у машинних кодах, але найчастіше виконується у внутрішньому, незалежному від машини коді. В останньому випадку реальне виконання оператора виробляється із залученням підсистеми підтримки часу виконання, тобто являє собою інтерпретатор цієї внутрішньої мови.

Нарешті, в окремі утиліти БД зазвичай виділяють процедури, які зручно виконувати без використання мови БД, наприклад, завантаження і вихід з БД, збирання статистики, глобальна перевірка цілісності БД та ін. Утиліти програмують з використанням інтерфейсу ядра СКБД, а інколи навіть з проникненням усередину ядра.

4. Етапи створення систем керування базами даних

4.1. Основні особливості систем, що базуються на інвертованих списках

До найбільш відомих і типових представників таких систем належать Datacom/DB компанії Applied Data Research, Inc. (ADR), що зорієнтована на використання на машинах основного класу фірми IBM, та Adabas компанії Software AG.

Доступ до даних на базі інвертованих списків використовують практично в усіх сучасних реляційних СКБД, але в цих системах користувачі не мають безпосереднього доступу до інвертованих списків (індексів).

4.1.1. Структури даних

База даних, що організувалася за допомогою інвертованих списків, схожа на реляційну БД, але з тією різницею, що таблиці, які зберігаються, і шляхи доступу до них доступні для користувачів. При цьому:

1) рядки таблиць упорядковано системою в деякій фізичній послідовності;
2) фізична впорядкованість рядків усіх таблиць може визначатися і для всієї БД (так робиться, наприклад, в Datacom/DB);
3) для кожної таблиці можна визначити довільну кількість ключів пошуку, для яких будуються індекси. Ці індекси автоматично підтримуються системою, але доступні для користувачів.

4.1.2. Маніпулювання даними

Підтримуються два класи операторів:

1) оператори, що встановлюють адресу запису, серед яких:
   - прямі пошукові оператори:
     - LOCATE FIRST – знайти перший запис таблиці у фізичному порядку, повертає адресу запису;
     - LOCATE FIRST WITH SEARCH KEY EQUEL – знайти перший запис таблиці у порядку ключа пошуку із заданим значенням ключа, повертає адресу запису;
   - оператори, що знаходять запис у термінах відносної позиції від поточного запису:
     - LOCATE NEXT – знайти перший запис, наступний за записом із заданою адресою у заданому шляху доступу, повертає адресу запису;
     - LOCATE NEXT WITH SEARCH KEY EQUEL – знайти наступний запис таблиці у порядку шляху пошуку із заданим значенням, має бути встановлена відповідність між засобом сканування і ключем, повертає адресу запису;

– LOCATE FIRST WITH SEARCH KEY GREATER – знайти перший запис таблиці у порядку ключа пошуку зі значенням ключового поля, більшим від заданого значення ключа, повертає адресу запису;

2) оператори, що приводять до перетворення записів:
– RETRIVE – вибрати запис із визначеною адресою;
– UPDATE – відновити запис із визначеною адресою;
– DELETE – видалити запис із визначеною адресою;
– STORE – занести запис у визначену таблицю, операція генерує адресу запису.

4.1.3. Обмеження цілісності

Загальних правил визначення цілісності БД немає. У деяких системах підтримують обмеження унікальності значень деяких полів, але, як правило, все покладається на прикладну програму.

4.2. Ієрархічні системи

Типовим представником (найбільш відомим і розповсюдженим) є Information Management System (IMS) фірми IBM. Перша версія з'явилася у 1968 р. Досі підтримується багато БД, що створює істотні проблеми з переходом як до нової технології БД, так і до нової техніки.

4.2.1. Ієрархічні структури даних

Ієрархічна БД складається з упорядкованого набору дерев, точніше, з упорядкованого набору декількох примірників одного типу дерева.

Тип дерева складається з одного «кореневого» типу запису і упорядкованого набору з нуля або більше типів піддерев (кожне з яких є деяким типом дерева). Тип дерева взагалі являє собою ієрархічно організований набір типів запису. На рис. 4.1 зображено схему ієрархічної БД типу дерева.

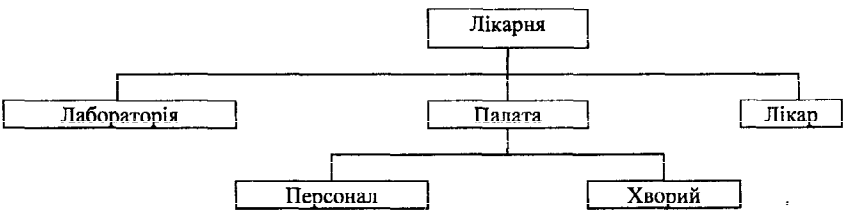


Рис. 4.1. Схема ієрархічної БД типу дерева

Усі екземпляри даного типу нащадка зі спільним екземпляром типу предка називаються *близнюками*. Для БД визначений повний порядок обходу – згори вниз, зліва направо.

4.2.2. Маніпулювання даними

Прикладами типових операторів маніпулювання над ієрархічно організованими даними можуть бути такі:

- знайти означене дерево БД (наприклад, палату 310);
- перейти від одного дерева до іншого;
- перейти від одного запису до іншого всередині дерева (наприклад, від палати – до першого хворого);
- перейти від одного запису до іншого в порядку обходу ієрархії;
- вставити новий запис у вказану позицію;
- видалити поточний запис.

4.2.3. Обмеження цілісності

Цілісність посилань між предками і нащадками підтримується автоматично. Основне правило: жодний нащадок не може існувати без свого предка. Відзначимо, що аналогічне підтримання цілісності посилань між записами, які не входять в одну ієрархію, не підтримується.

4.2.4. Особливості ієрархічної моделі

Можна виділити такі особливості ієрархічної моделі:

- дані організовано в ієрархічні структури;
- у разі моделювання мережевих структур ієрархічними необхідне дублювання даних;
- основна одиниця обробки – запис;
- обробка завжди починається з кореневого запису; доступ до некореневого запису здійснюється тільки ієрархічно;
- ієрархічна модель містить засоби для опису різноманітних поглядів користувачів на однакові дані.

4.3. Мережеві системи

Типовим представником мережевих систем є Integrated Database Management System (IDMS) компанії Cullinet Software, Inc., призначена для використання на машинах основного класу фірми IBM під керуванням більшості операційних систем. Архітектура системи базується на пропозиціях Data Base Task Group (DBTG) – Комітету з мов програмування Conference on Data Systems Languages (CODASYL). Звіт DBTG було опубліковано – 1971 р., а в 70-х роках з'явилося декілька систем, і серед них IDMS.

4.3.1. Мережеві структури даних

Для опису структур даних за пропозиціями робочої групи з БД (DBTG) використовують такі визначення:

- *елемент даних* – найменша одиниця структури даних;
- *агрегат даних* – іменована сукупність елементів або агрегатів даних;

- *запис* – агрегат, який не входить до складу ніякого іншого агрегату та використовується як основна одиниця обробки у БД. Якщо до запису входить декілька значень елемента одного типу, то вони визначаються як вектор. Існують вектори постійної та змінної довжини;
- *первинний ключ* – сукупність елементів, яка однозначно ідентифікує запис;
- *набір* – ієрархічне відношення між записами двох типів. Записи одного з типів визначають як предків, записи іншого – як нащадків.

Мережевий підхід до організації даних є розширенням ієрархічного. В ієрархічних структурах запис-нащадок повинен мати тільки одного предка, а у мережевій структурі даних нащадок може мати будь-яку кількість предків.

Мережева БД складається з набору записів і набору зв'язків між ними (групових відношень). Тип зв'язку визначається для двох типів запису: предка і нащадка. Тип зв'язку складається з одного екземпляра типу запису предка і упорядкованого набору типів запису нащадка. Для цього типу зв'язку L з типом запису предка P і типом запису нащадка C мають виконуватися дві умови:

- кожний екземпляр типу P є предком тільки в одному екземплярі L;
- кожний екземпляр типу C є нащадком не більш ніж в одному екземплярі L.

На формування типів зв'язку не накладено обмежень щодо використання типів записів. Можливі, наприклад, ситуації:

- тип запису нащадка в одному типі зв'язку L_1 може бути типом запису предка в іншому типі зв'язку L_2;
- даний тип запису P може бути типом запису предка у будь-якій кількості типів зв'язків;
- даний тип запису P може бути типом запису нащадка у будь-якій кількості типів зв'язків;
- може існувати будь-яка кількість типів зв'язків з одним і тим самим типом запису предка й одним і тим самим типом запису нащадка. Якщо L_1 і L_2 – два типи зв'язку з одним і тим самим типом запису предка P та одним і тим самим типом запису нащадка C, то утворюються різні типи зв'язку;
- типи запису X і Y можуть бути предком і нащадком в одному зв'язку і нащадком і предком – в іншому;
- предок і нащадок можуть бути одного типу запису.

Для представлення групових відношень використовують діаграми Бахмана. У них типи записів зображують як прямокутники – вершини, а типи зв'язків – зорієнтованими дугами.

Приклад фрагмента мережевої моделі даних зображено на рис. 4.2.

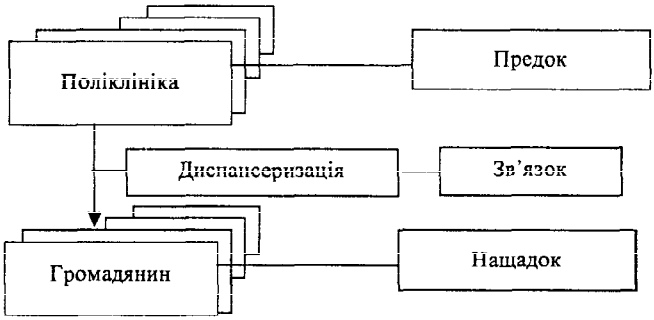


Рис. 4.2. Приклад фрагмента мережевої моделі даних

Кожний тип зв'язку можна схарактеризувати за такими ознаками:

- засіб упорядкування записів-нащадків. Він може бути будь-яким: хронологічним, нехронологічним, відсортованим;
- режим внесення записів-нащадків. Може бути автоматичним або ручним;
- режим вилучення записів-нащадків. Вилучення може бути:
  - фіксованим – запис-нащадок суворо закріплено до запису предка. Запис можна вилучити зі зв'язку тільки вилучивши його з БД;
  - обов'язковим – запис-нащадок має бути зв'язаний із записом-предком, але може бути підпорядкований іншому предку;
  - необов'язковим – запис-нащадок можна вилучити зі зв'язку, але залишити його в БД.

4.3.2. Маніпулювання даними

До типового набору операцій, які виконують СКБД, можна віднести такі:

- створити БД;
- створити таблицю БД;
- створити індекси для таблиці БД;
- забезпечити механізми захисту даних;
- знайти конкретний запис у наборі однотипних записів;
- перейти від предка до першого нащадка за деяким зв'язком;
- перейти до наступного нащадка за деяким зв'язком;
- перейти від нащадка до предка за деяким зв'язком;
- створити новий запис;
- знищити запис;
- модифікувати запис;
- внести у зв'язок;
- вилучити із зв'язку;
- переставити в інший зв'язок і т. д.

4.3.3. Обмеження цілісності

Обмежень цілісності, як правило, в мережевій системі не існує, але інколи вимагають цілісності за посиланням (як в ієрархічній моделі).

4.4. Переваги і недоліки ранніх систем керування базами даних

Переваги ранніх СКБД:

- розвинені засоби керування даними в зовнішній пам'яті на низькому рівні;
- можливість побудови вручну ефективних прикладних систем.

Недоліки:

- занадто складні у використанні;
- фактично необхідні знання про фізичну організацію;
- прикладні системи залежать від фізичної організації;
- логіка маніпулювання даними перевантажена деталями організації доступу до БД.

5. Реляційні бази даних

5.1. Загальний опис реляційного підходу

Реляційний підхід до побудови БД є найбільш розповсюдженим, незважаючи на те, що поряд із загальновизнаними перевагами містить ряд недоліків. До переваг реляційного підходу можна віднести:

- наявність невеликого набору абстракцій, що дозволяють порівняно просто моделювати більшу частину розповсюджених ПО і припускають точні формальні визначення, залишаючись інтуїтивно зрозумілими;
- наявність простого і водночас потужного математичного апарату, який спирається на теорію множин і математичну логіку та реляційний базис, який забезпечує теоретичний підхід до вирішення проблем щодо організації БД;
- можливість ненавігаційного маніпулювання даними без знання конкретної фізичної організації БД у зовнішній пам'яті.

Реляційні системи далеко не відразу набули широкого розповсюдження. Основні теоретичні результати в цій сфері було отримано ще в 70-х роках, і тоді ж з'явилися перші прототипи реляційних СКБД. Тривалий час вважали неможливим досягти ефективної реалізації таких систем. Однак зазначені вище переваги і поступове накопичування засобів та алгоритмів організації реляційних БД і керування ними зумовили те, що вже в середині 80-х років реляційні системи практично «перемогли» на світовому інформаційному ринку ранні СКБД. Нині основним предметом критики реляційних СКБД є не їх недостатня ефективність, а притаманна цим системам деяка обмеженість (наслідок простоти) під час використання в так званих нетрадиційних галузях застосування (найбільш розповсюдженими прикладами є САПР), в яких потрібні гранично складні структури даних. Ще одним недоліком реляційних БД є неможливість адекватного відображення семантики ПО. Іншими словами, можливості щодо подання знань про семантичну специфіку ПО в реляційних системах обмежені. Сучасні дослідження у сфері постреляційних систем присвячені саме усуненню цих недоліків.

5.2. Основні поняття реляційних баз даних

Реляційна СКБД (або просто реляційна система) – це система, в якій виконуються як мінімум дві умови:

1. Дані надаються користувачеві саме у вигляді таблиць.
2. Користувачі отримують спеціальні засоби (оператори мови SQL), за допомогою яких вони отримують дані з таблиці та виконують різні операції над таблицями.

Основні поняття реляційних БД:

– тип даних; – домен;
– атрибут; – кортеж;
– первинний ключ; – відношення.

У простому розумінні відношення є таблицею з інформацією про сукупність екземплярів певного об'єкта ПО. Заголовком таблиці є перелік назв характеристик об'єкта. Характеристики об'єкта створюють множину атрибутів відношення. Імена атрибутів є назвами стовпчиків цієї таблиці. Кожен рядок таблиці містить інформацію про окремий екземпляр об'єкта. Рядки таблиці створюють множину кортежів відношення, тому *інколи говорять «стовпчик таблиці», маючи на увазі «атрибут відношення».* Цієї термінології дотримуються в більшості комерційних реляційних СКБД. Реляційна БД – це набір відношень, імена яких збігаються з іменами схем відношень у схемі БД.

Отже, основні структурні поняття реляційної моделі даних (якщо не враховувати поняття домену) мають дуже просту інтуїтивну інтерпретацію, хоча в теорії реляційних БД всі вони визначаються абсолютно формально і точно.

Суть цих понять розглянуто на прикладі відношення СПІВРОБІТНИКИ (рис. 5.1), що містить таку інформацію про співробітників деякої організації: номер перепустки, прізвище, ім'я, ставка, номер відділу, дата народження.

Таке відношення може формуватися для різних ПО, наприклад для розв'язання задач обліку кадрового ресурсу на нарахування заробітної плати.

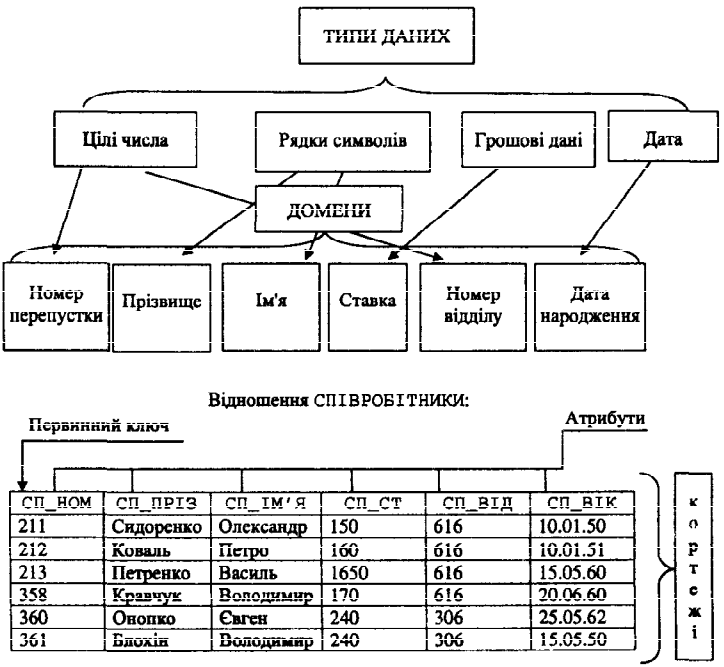


Рис. 5.1. Основні поняття реляційної моделі

5.2.1. Тип даних

Поняття *тип даних* у реляційній моделі даних повністю відповідає поняттю *тип даних* у мовах програмування. Зазвичай у сучасних реляційних БД допускається зберігання символьних і числових даних, бітових рядків, спеціалізованих числових даних (таких, як «гроші»), а також спеціальних *темпоральних* даних (дата, час, тимчасовий інтервал). Достатньо активно розвивається підхід до розширення можливостей реляційних систем абстрактними типами даних (відповідні можливості мають, наприклад, системи сім'ї Ingres/Postgres). У цьому прикладі маємо справу з даними трьох типів: рядок символів, ціле число і «грошовий».

5.2.2. Домен

Поняття *домен* більш специфічне для БД, хоча і має деякі аналогії з підтипами в деяких мовах програмування. У загальному вигляді домен визначається заданням деякого базового типу даних, до якого відносять елементи домену, і довільного логічного виразу, що застосовується до елемента типу даних. Якщо обчислення цього логічного виразу дасть результат «істина», то елемент даних є елементом домену. Найбільш правильним інтуїтивним тлумаченням поняття домену є розуміння його як допустимої потенційної множини значень даного типу. Наприклад, домен «Імена» в нашому прикладі визначений на базовому типі «рядок символів», але до його значень можуть входити тільки ті рядки, що можуть зображати ім'я (зокрема, такі рядки не можуть починатися з м'якого знака). У цьому прикладі логічним виразом є вираз «Чи є даний рядок символів іменем?». Слід відзначити також семантичне навантаження поняття домену: дані вважають порівняними тільки в тому випадку, коли вони належать до одного домену. У нашому прикладі значення доменів «Номери пропусків» і «Номери груп» належать до типу цілих чисел, але їх не можна порівнювати. Зазначимо, що в більшості реляційних СКБД поняття домену не використовують, хоча, наприклад, в Oracle V.7 воно вже підтримується.

5.2.3. Схема відношення, схема бази даних

Схема відношення – це іменована множина пар {ім'я атрибута, ім'я домену} (або типу, якщо поняття домену не підтримується). Ступінь, або «арність», схеми відношення визначається потужністю цієї множини. Ступінь відношення СПІВРОБІТНИКИ дорівнює 6, тобто відношення є 6-арним. Якщо всі атрибути одного відношення визначені на різних доменах, то доцільно використати для іменування атрибутів імена відповідних доменів (пам'ятаючи про те, що це є лише зручним засобом іменування і не усуває відмінності між поняттями домену та атрибута). Схема БД (у структурному розумінні) – це іменована множина іменованих схем відношень.

5.2.4. Кортеж, відношення

Кортеж, що відповідає схемі відношення, – це множина пар {ім'я атрибута, значення}. Кортеж містить одне входження для кожного імені атрибута, який належить схемі відношення. «Значення» є допустимим значенням домену атрибута (або типу даних, якщо поняття домену не підтримується). Завдяки цьому ступінь, або «арність», кортежу, тобто кількість елементів у ньому, збігається з «арністю» відповідної схеми відношення. Отже, кортеж – це іменований набір значень заданого типу, а відношення – це множина кортежів, що відповідають одній схемі відношення. Поняття *схеми відношення* ближче за все до поняття *структурного типу даних* у мовах програмування. Було б цілком логічно дозволяти окремо визначати схему відношення, а після цього – одне або декілька відношень з даною схемою. Однак у реляційних БД це не прийнято. Ім'я схеми відношення в таких БД завжди збігається з ім'ям відповідного відношення-екземпляра. У класичних реляційних БД після визначення схеми БД для кожної схеми-відношення обирають відповідні відношення-екземпляри. В них можуть з'являтися нові та вилучатися або модифікуватися існуючі кортежі. Проте у багатьох реалізаціях допускається зміна схеми БД: визначення нових і зміна існуючих схем-відношень. Це прийнято називати *еволюцією схеми бази даних*.

5.3. Фундаментальні властивості відношень

5.3.1. Відсутність кортежів-дублікатів

Та властивість, що відношення не містять кортежі-дублікати, виходить з визначення відношення як множини кортежів. У класичній теорії множин, згідно з визначенням, кожна множина складається з різноманітних елементів. З цієї властивості випливає наявність у кожного відношення так званого *первинного ключа* – набору атрибутів, значення яких однозначно визначають кортеж відношення. Для кожного відношення повний набір його атрибутів є первинним ключем. Однак при формальному визначенні первинного ключа потрібне забезпечення його «мінімальності», тобто до набору атрибутів первинного ключа не мають входити такі атрибути, які можна відкинути без втрат для основної властивості, – однозначно визначати кортеж. Поняття первинного ключа є важливим щодо поняття цілісності БД.

5.3.2. Відсутність упорядкованості кортежів

Властивість відсутності упорядкованості кортежів відношення також є наслідком визначення відношення-екземпляра як множини кортежів. Відсутність вимоги до підтримання порядку на множині кортежів відношення дає додаткову гнучкість СКБД у зберігання БД у зовнішній пам'яті та під час виконання запитів до БД. Це не суперечить тому, що при формулюванні за-

питу до БД, наприклад, мовою SQL, можна задати сортування таблиці-відповіді, яка створюється у відповідь на запит. Такий результат не є відношенням, а являє собою деякий упорядкований список кортежів.

5.3.3. Відсутність упорядкованості атрибутів

Атрибути відношень не є упорядкованими, оскільки відповідно до визначення схема відношення є множиною пар {ім'я атрибута, ім'я домену}. Для посилання на значення атрибута в кортежі відношення завжди використовують ім'я атрибута. Ця властивість теоретично дозволяє, наприклад, модифікувати схеми існуючих відношень не тільки шляхом додавання нових, але і шляхом вилучення існуючих атрибутів, однак у більшості існуючих систем така можливість не допускається. Хоча упорядкованість набору атрибутів відношення явно не вимагається, часто як неявний порядок атрибутів використовують їх порядок у лінійній формі визначення схеми відношення.

5.3.4. Атомарність значень атрибутів

Уведемо поняття простого і складного атрибута. *Простим* назвемо атрибут, значення якого є атомарними, тобто не мають внутрішньої структури. *Складний* атрибут може мати значення, які являють собою конкатенацію декількох значень одного або різних доменів. Складним атрибутом може бути вектор, що повторюється, або послідовність. У наведеному нижче прикладі у відношенні R_1 наявний складний атрибут *Діти*. Всі інші атрибути є простими. Значення всіх атрибутів у відношенні є атомарними. Це випливає з визначення домену як потенційної множини значень простого типу даних, тобто серед значень домену не можуть міститися множини значень.

Нормалізованим називається відношення, всі атрибути якого є простими. *Ненормалізоване* відношення (тобто відношення, атрибути якого складні) легко нормалізувати, якщо поділити його на групи (для тих атрибутів, які повторюються) за різними записами та створити нові структури цих атрибутів. Таке перетворення називається *нормалізацією*. Вона може призвести до збільшення потужності відношення і зміни первинного ключа.

Розглянемо приклад ненормалізованого відношення і зведення його до нормальної форми.

Відношення R_1:

ТАБ N	Прізвище	Оклад	Відділ	Телефон	Діти	
					Ім'я	Вік
211	Кузнецов	150	12	616	Петро	10
					Іван	7
					Сергій	3
358	Темкін	170	12	616	Василь	5
360	Кошкін	240	5	306	Євген	8
					Юрко	6

Ненормалізоване відношення R_1 перетворюється на відношення R_2, яке знаходиться у першій нормальній формі:

Відношення R_2:

ТАБ N	Прізвище	Оклад	Відділ	Телефон	Ім'я дитини	Вік
211	Кузнецов	150	12	616	Петро	10
211	Кузнецов	150	12	616	Іван	7
211	Кузнецов	150	12	616	Сергій	3
358	Темкін	170	12	616	Василь	5
360	Кошкін	240	5	306	Єген	8
360	Кошкін	240	5	306	Юрко	6

Розглянемо дві операції:

- зарахувати співробітника Кузнецова (табельний номер 300, зарплата 115, дитина Петро) у відділ номер 10;
- зарахувати співробітника Кузнецова (пропуск номер 300, зарплата 115, дитина Іван) у відділ номер 10.

Якщо інформація про співробітників подана у вигляді відношення R_2 (нормалізоване), обидва оператори будуть виконуватися однаково (вставити кортеж у відношення R_2). Якщо працювати з ненормалізованим відношенням R_1, то першим оператором буде – вставити кортеж у відношення, а другим – доповнити інформацію про Кузнецова в множинне значення атрибута *Діти* кортежу з первинним ключем 211.

У реляційних БД допускаються тільки нормалізовані відношення або відношення, наведені в першій нормальній формі.

5.4. Загальна характеристика реляційної моделі даних

Основні поняття реляційних баз, які були розглянуті в попередніх розділах, однаковою мірою стосуються будь-якої системи, при побудові якої використовувався реляційний підхід. Іншими словами, було неявно використано поняття так званої *реляційної моделі даних*. Модель даних описує деякий набір родових понять і ознак, які повинні мати всі конкретні СКБД, якщо вони базуються на цій моделі. Наявність моделі даних дозволяє порівнювати конкретні реалізації, використовуючи одну загальну мову. Хоча поняття моделі даних є загальним, і можна говорити про ієрархічну, мережеву, деяку семантичну та інші моделі даних, слід відзначити, що це поняття було введене стосовно до реляційних систем і найбільш ефективно використовується саме в цьому контексті.

Найбільш розповсюджене тлумачення реляційної моделі даних належить К. Дейту, суть якого полягає в тому, що реляційна модель складається з трьох частин, які описують різні аспекти реляційного підходу: структурної, маніпуляційної і цілісної частин.

У структурній частині моделі фіксується, що єдиною структурою даних, яку використовують в реляційних БД, є нормалізоване *n*-арне відношення.

У маніпуляційній частині моделі затверджуються два фундаментальних механізми маніпулювання реляційними БД – реляційна алгебра і реляційне обчислення. Перший механізм базується переважно на класичній теорії множин (з деякими уточненнями), а другий – на класичному логічному апараті обчислення предикатів першого порядку. Основною функцією маніпуляційної частини реляційної моделі є забезпечення міри реляційності будь-якої конкретної мови реляційних БД: мова називається *реляційною*, якщо вона забезпечує не меншу потужність маніпулювання, ніж реляційна алгебра або реляційне обчислення.

У цілісній частині реляційної моделі даних фіксуються дві базові вимоги цілісності, що мають підтримуватися у будь-якій реляційній СКБД.

Першу вимогу називають *вимогою цілісності сутності*. Об'єкту (або сутності) реального світу в реляційних БД відповідають кортежі відношень. Конкретно вимога полягає в тому, що будь-який кортеж будь-якого відношення має відрізнятися від будь-якого іншого кортежу цього відношення. Інакше, будь-яке відношення повинно мати первинний ключ. Ця вимога автоматично задовольняється, якщо в системі не порушуються базові властивості відношень.

Другу вимогу називають *вимогою цілісності за посиланням*, і вона є дещо складнішою. Очевидно, що у разі дотримання нормалізованості відношень складні сутності реального світу подаються в реляційній БД у вигляді декількох кортежів декількох відношень. Наприклад, уявімо, що потрібно відобразити в реляційній БД сутність ВІДДІЛ з атрибутами ВІД_НОМЕР (номер відділу), ВІД_КІЛЬК (кількість співробітників) і ВІД_СПІВ (набір співробітників відділу). Для кожного співробітника потрібно зберігати СПІВ_НОМЕР (номер співробітника), СПІВ_ІМ'Я (ім'я співробітника) і СПІВ_ЗАРП (заробітна плата співробітника). Якщо проектування відповідної БД правильне, у ній з'являться два відношення: ВІДДІЛИ (ВІД_НОМЕР, ВІД_КІЛЬК) (первинний ключ – ВІД_НОМЕР) і СПІВРОБІТНИКИ (СПІВ_НОМЕР, СПІВ_ІМ'Я, СПІВ_ЗАРП, СПІВ_ВІД_НОМ) (первинний ключ – СПІВ_НОМЕР). Як видно, атрибут СПІВ_ВІД_НОМ з'являється у відношенні СПІВРОБІТНИКИ не тому, що номер відділу є особистою характеристикою співробітника, а лише для того, щоб мати можливість відновити у разі необхідності повну сутність ВІДДІЛ. Значення атрибута СПІВ_ВІД_НОМ у будь-якому кортежі відношення СПІВРОБІТНИКИ має відповідати значенню атрибута ВІД_НОМ у деякому кортежі відношення ВІДДІЛИ. Атрибут такого роду називають *зовнішнім ключем*, оскільки його значення однозначно характеризують сутності, наведені кортежами деякого іншого відношення (тобто виз-

начають значення їхнього первинного ключа). Отже, відношення, в якому визначений зовнішній ключ, посилається на відповідне відношення, в якому такий самий атрибут є первинним ключем.

Суть вимоги цілісності за посиланням (вимога зовнішнього ключа) полягає в тому, що для кожного значення зовнішнього ключа, яке з'являється у відношенні, має знайтися кортеж з таким самим значенням первинного ключа, що і у відношенні, на яке посилаються, або значення зовнішнього ключа має бути повністю невизначеним (ні на що не вказувати). Для цього прикладу це означає, що якщо для співробітника вказано номер відділу, то цей відділ має існувати. Обмеження цілісності сутності і за цілісності посиланнями повинні підтримуватися СКБД. Для дотримання цілісності сутності достатньо гарантувати відсутність у будь-якому відношенні кортежів з однаковим значенням первинного ключа.

З цілісністю за посиланням справа більш складна. Зрозуміло, що у разі поновлення відношення, на яке виконують посилання (доповнення нових кортежів або модифікація значення зовнішнього ключа в існуючих кортежах), достатньо стежити за тим, щоб не з'являлися некоректні значення зовнішнього ключа. Але як бути у разі вилучення кортежу з відношення, на яке посилаються? Тут існують три підходи, кожний з яких підтримує цілісність за посиланнями:

- заборонено вилучати кортеж, на який існують посилання, тобто спочатку потрібно або усунути кортежі, на які є посилання, або відповідним чином змінити значення їх зовнішнього ключа;
- у разі вилучення кортежу, на який є посилання, в усіх кортежах, на які є посилання, значення зовнішнього ключа автоматично стає невизначеним;
- у випадку вилучення кортежу з відношення, на яке є посилання, автоматично вилучаються всі кортежі, на які є посилання.

У розвинених реляційних СКБД зазвичай можна вибрати засіб підтримання цілісності за посиланнями для кожної окремої ситуації визначення зовнішнього ключа. Як правило, для прийняття такого рішення необхідно аналізувати вимоги конкретної прикладної області.

6. Засоби маніпулювання реляційними даними

6.1. Загальний підхід

Як було вказано вище, у маніпуляційній частині реляційної моделі визначають два базові механізми маніпулювання реляційними даними:

- реляційна алгебра, яка спирається на теорію множин;
- реляційне обчислення, яке базується на математичній логіці (точніше, на обчисленні предикатів першого порядку). Розглядають два види реляційного обчислення – обчислення доменів та обчислення предикатів.

Усі ці механізми мають одну важливу властивість: вони є замкненими відносно поняття відношення. Це означає, що операції реляційної алгебри і формули реляційного обчислення проводять над відношеннями реляційних БД і результатами обчислень також є відношення. Як наслідок, будь-який вираз або формула можуть інтерпретуватися як відношення, що дозволяє використати їх в інших виразах або формулах. Отже, алгебра і обчислення мають велику виразну потужність: дуже складні запити до БД можуть бути виражені за допомогою одного виразу реляційної алгебри або однієї формули реляційного обчислення. Саме тому ці механізми включені в реляційну модель даних.

Конкретну мову маніпулювання реляційними БД називають *реляційно повною*, якщо будь-який запит, створений за допомогою одного виразу реляційної алгебри або однієї формули реляційного обчислення, можна записати за допомогою одного оператора цієї мови. Відомо (тому не будемо наводити доказ цього), що механізми реляційної алгебри і реляційного обчислення еквівалентні, тобто для будь-якого допустимого виразу реляційної алгебри можна побудувати еквівалентну (таку, що виробляє такий самий результат) формулу реляційного обчислення і навпаки. Чому ж у реляційній моделі даних наявні обидва механізми? Справа в тому, що вони відрізняються рівнем процедурності. Вирази реляційної алгебри будуються на базі алгебричних операцій (високого рівня) і мають процедурну інтерпретацію. Іншими словами, запит, наведений мовою реляційної алгебри, може бути обчислений на підставі обчислення елементарних алгебричних операцій з урахуванням їх порядку і можливої наявності дужок. Для формули реляційного обчислення такої інтерпретації немає. Формула тільки встановлює умови, які мають задовольняти кортежі відношення-результату. Тому мови реляційного обчислення є скоріше непроцедурними або декларативними. Оскільки механізми реляційної алгебри і реляційного обчислення еквівалентні, то в конкретній ситуації для перевірки ступеня реляційності деякої мови БД можна користуватися будь-яким з цих механізмів. Відзначимо, що дуже рідко алгебру або об-

числення беруть за повну основу якої-небудь мови БД. Зазвичай (як, наприклад, у випадку мови SQL) мова базується на деякій суміші алгебричних і логічних конструкцій. Однак знання алгебричних і логічних основ мов БД часто є корисним на практиці.

6.2. Реляційна алгебра

Основна ідея реляційної алгебри полягає в тому, що засоби маніпулювання відношеннями базуються на традиційних теоретико-множинних операціях, доповнених деякими спеціальними операціями, специфічними для БД. Існує багато підходів до визначення реляційної алгебри. Вони різняться набором операцій і засобами їх інтерпретації, але, загалом, більш-менш збігаються. Наведемо розширений початковий варіант алгебри, який було запропоновано Е. Ф. Коддом (E. F. Codd). У цьому варіанті набір основних алгебричних операцій складається з восьми операцій, поділених на два класи: теоретико-множинні і спеціальні реляційні операції.

До теоретико-множинних входять операції над відношеннями:

- об'єднання;
- перетин;
- різниця;
- декартовий добуток.

Спеціальні реляційні операції над відношеннями такі:

- селекція (обмеження, фільтрація);
- проекція;
- сполучення;
- ділення.

Крім того, до складу алгебри входить операція присвоювання, яка дозволяє зберегти у БД результати обчислення алгебричних виразів, і операція перейменування атрибутів, яка дає можливість коректно сформувати заголовок (схему) відношення, яке є результатом обчислення.

6.2.1. Загальна інтерпретація реляційних операцій

Два відношення будуть *сумісними за типом*, якщо вони мають ідентичні схеми, тобто:

- кожне з них складається з однакової множини атрибутів;
- відповідні атрибути (з однаковими назвами) визначено на однакових доменах.

Можна сказати, що сумісні за типом відношення можна представити як таблиці з однаковими назвами стовпчиків.

Операції об'єднання, перетину та різниці проводяться тільки над сумісними за типом відношеннями.

Для наведених операцій має місце така інтерпретація їх змісту:

- результатом операції об'єднання двох відношень буде нове відношення, до якого увійдуть усі кортежі, що входять хоча б в одне з відношень-операндів;
- результатом операції перетину двох відношень буде відношення, до якого увійдуть усі кортежі, що входять в обидва відношення-операнди;
- відношення, що є різницею двох відношень, містить усі кортежі, які входять у перше відношення-операнд, але жоден з них не входить у друге відношення-операнд;
- результатом виконання операції «прямий (декартовий) добуток» двох відношень є відношення, кортежі якого є конкатенацією (зчепленням) кортежів першого і другого відношень-операндів;
- результатом обмеження відношення за деякою умовою є відношення, до якого входять кортежі відношення-операнда, які задовольняють цю умову;
- результатом проекції відношення на заданий набір його атрибутів є відношення, кортежі якого будують шляхом взяття відповідних значень із заданих стовпчиків кортежів відношення-операнда;
- у разі сполучення двох відношень за деякою умовою утвориться відношення, кортежі якого є конкатенацією кортежів першого і другого відношень, що задовольняють цю умову;
- в операції реляційного ділення беруть участь два операнди – бінарне і унарне відношення. Відношення-результат складається з одноатрибутних кортежів, які містять такі значення першого атрибута кортежів першого відношення-операнда, для яких множина значень другого атрибута (при фіксованому значенні першого атрибута) збігається із множиною значень атрибута другого відношення-операнда;
- операція перейменування дає нове відношення, тіло якого збігається з тілом заданого відношення-операнда, але імена атрибутів змінені;
- операція присвоювання дозволяє зберегти результат обчислення реляційного виразу в існуючому відношенні БД.

Враховуючи той факт, що результатом будь-якої реляційної операції (окрім операції присвоювання) є деяке відношення, можна утворювати реляційні вирази, у яких на місці відношення-операнда знаходиться реляційна операція.

6.2.2. Замкненість реляційної алгебри та операція перейменування

Відомо, що кожне відношення характеризується схемою (або заголовком) відношення і набором кортежів (або тілом). Тому, якщо дійсно бажано мати алгебру, операції якої замкнені відносно поняття відношення, то результатом кожної операції має бути відношення, тобто воно повинно мати і тіло, і заго-

ловок. Тільки в цьому випадку можна будувати вкладені вирази. Заголовок відношення являє собою множину пар <ім'я – атрибута, ім'я – домену>. Якщо розглянути реляційні операції (див. підрозд. 6.2.1), то видно, що домени атрибутів відношення-результату однозначно визначаються доменами відношень-операндів. Однак з іменами атрибутів результату не завжди все так просто. Наприклад, уявімо собі, що у відношень-операндів, які є елементами операції декартового добутку, є однойменні атрибути з однаковими доменами. Яким має бути заголовок відношення-результату? Оскільки заголовок – це множина, у ньому не мають міститися однакові елементи. Але і втратити атрибут у результаті операції неприпустимо. А це означає, що в цьому разі взагалі неможливо коректно виконати операцію прямого добутку. Аналогічні проблеми можуть виникати і у випадках інших двомісних операцій. З метою їх дозволу у склад операцій реляційної алгебри вводять операцію перейменування. Її слід застосовувати у будь-якому випадку, коли виникає конфлікт назв атрибутів у відношеннях-операндах однієї реляційної операції. Тоді до одного з операндів спочатку застосовують операцію перейменування, а після цього основна операція виконується вже без ускладнень. Надалі будемо припускати застосування операції перейменування в усіх конфліктних випадках.

6.2.3. Особливості теоретико-множинних операцій реляційної алгебри

Незважаючи на те, що в основу теоретико-множиної частини реляційної алгебри покладено класичну теорію множин, відповідні операції реляційної алгебри мають деякі особливості. Почнемо з операції об'єднання (все, що стверджується для операції об'єднання, справедливе і для операцій перетину та різниці). Суть операції об'єднання в реляційній алгебрі в цілому залишається теоретико-множинною. Але, якщо в теорії множин операція об'єднання має сенс для будь-яких двох множин-операндів, то у випадку реляційної алгебри результатом операції об'єднання має бути відношення. Якщо припустити в реляційній алгебрі можливість теоретико-множинного об'єднання двох довільних відношень (з різними схемами), то результатом операції буде множина, але ця множина складатиметься з різнотипних кортежів, тобто вона не є відношенням. Якщо виходити з вимоги замкненості реляційної алгебри відносно поняття відношення, то така операція об'єднання є безглуздою. Усі ці міркування зумовлюють появу поняття сумісності відношень за об'єднанням: два відношення є сумісними за об'єднанням в тому і тільки в тому випадку, коли мають однакові заголовки. Більш точно це означає, що в заголовках обох відношень міститься один і тий самий набір імен атрибутів, і однойменні атрибути визначені на одному і тому самому домені. Якщо два відношення сумісні за об'єднанням, то у разі звичайного виконання над ними операцій об'єднання, перетину і знаходження різниці результатом операції є відношення з коректним заголовком, який збігається із заголовком кожного з відношень-операндів.

Нагадаємо, що якщо два відношення «майже» сумісні за об'єднанням, тобто сумісні в усьому, окрім імен атрибутів, то до виконання операції типу сполучення ці відношення можна зробити повністю сумісними за об'єднанням шляхом застосування операції перейменування. Відзначимо, що внесення до складу операцій реляційної алгебри трьох операцій об'єднання, перетину і знаходження різниці є очевидно зайвим, оскільки відомо, що будь-яка з цих операцій виражається через дві інші. Однак Кодд свого часу вирішив включити всі три операції, виходячи з інтуїтивних потреб потенційного користувача системи реляційних БД, далекого від математики.

Інші проблеми пов'язані з операцією визначення декартового (або прямого) добутку двох відношень. У теорії множин прямий добуток може бути отриманий для будь-яких двох множин, та елементами множини-результату є пари, складені з елементів першої та другої множин. Оскільки відношення є множинами, то і для будь-яких двох відношень можлива операція прямого добутку. Але результат не буде відношенням! Елементами результату будуть не кортежі, а пари кортежів. Тому в реляційній алгебрі використовується спеціалізована форма операції прямого добутку – розширений прямий добуток відношень. Під час виконання операції розширеного прямого добутку двох відношень елементом відношення-результату є кортеж, який є конкатенацією (або злиттям) одного кортежу першого відношення і одного кортежу другого відношення. Але тепер виникає нове запитання – як отримати коректно сформований заголовок відношення-результату? Очевидно, що проблемою може бути іменування атрибутів відношення-результату, якщо відношення-операнди мають однойменні атрибути. Ці міркування зумовлюють появу поняття сумісності для розширеного прямого добутку. Два відношення є сумісними для знаходження розширеного прямого добутку лише в тому разі, коли множини імен атрибутів цих відношень не перетинаються. Будь-які два відношення можна зробити сумісними для прямого добутку шляхом застосування операції перейменування до одного з цих відношень. Необхідно зазначити, що операція прямого добутку не часто використовується на практиці. По-перше, потужність її результату є дуже великою навіть за допустимих потужностей операндів, а по-друге, результат операції не буде більш інформативним, ніж отримані у сукупності операнди.

Основний сенс включення операції розширеного прямого добутку до складу реляційної алгебри полягає в тому, що на її основі визначається справді корисна операція сполучення (детальніше про це наведено нижче). Щодо теоретико-множинних операцій реляційної алгебри слід зазначити, що всі чотири операції є асоціативними, тобто, якщо визначити через OP будь-яку з чотирьох операцій, то (A OP B) OP C = A OP (B OP C) і, отже, без двозначності можна писати A OP B OP C (A, B і C – відношення, що мають властивості, необхідні для коректного виконання відповідної операції). Усі операції, окрім знаходження різниці, є комутативними, тобто A OP B = B OP A.

6.2.4. Спеціальні реляційні операції

Розглянемо спеціальні реляційні операції реляційної алгебри: селекція, проекція, сполучення і ділення.

6.2.4.1. Операція селекції

Операція селекції вимагає наявності двох операндів: відношення, яке підлягає селекції, і простої умови селекції. Проста умова селекції може мати вигляд (a comp-op b), де a і b – імена атрибутів відношення, для яких має сенс операція порівняння comp-op, або вигляд (a comp-op const), де a – ім'я атрибута відношення, а const – літерально задана константа. У результаті виконання операції селекції отримаємо відношення, заголовок якого збігається із заголовком відношення-операнда, а в тіло входять ті кортежі відношення-операнда, для яких значенням умови селекції є істина.

Уведемо такі позначення:

UNION	– операція об'єднання;
INTERSECT	– операція перетину;
MINUS	– операція знаходження різниці;
A WHERE comp	– конструкція для позначення операції селекції, де A – відношення, а comp – проста умова селекції.

Нехай $comp_1$ і $comp_2$ – дві прості умови селекції. Тоді згідно з позначеннями:

- A WHERE $comp_1$ AND $comp_2$ означає те саме, що і (A WHERE $comp_1$) INTERSECT (A WHERE $comp_2$);
- A WHERE $comp_1$ OR $comp_2$ означає те саме, що і (A WHERE $comp_1$) UNION (A WHERE $comp_2$);
- A WHERE NOT $comp_1$ означає те саме, що і A MINUS (A WHERE comp1).

З використанням цих визначень можна виконувати операції селекції, в яких умовою селекції є довільний булевий вираз, складений із простих умов з використанням логічних зв'язків AND, OR і AND та дужок. На інтуїтивному рівні операцію селекції найкраще подати як знаходження деякої «горизонтальної» вибірки з відношення-операнда.

6.2.4.2. Операція отримання проекції

Операція отримання проекції також потребує наявності двох операндів – відношення A, над яким виконується операція, та списку імен атрибутів, які входять до заголовка відношення A. Результатом проекції відношення A за списком атрибутів a_1, a_2, ..., a_n є відношення із заголов-

ком, який визначається множиною атрибутів a_1, a_2,..., a_n, та з тілом, яке складається з кортежів вигляду $<a_1: v_1, a_2: v_2, ..., a_n: v_n>$, таких, що у відношенні A є кортеж, атрибут a_1 якого має значення v_1, атрибут a_2 має значення v_2, ..., атрибут a_n – значення v_n. Завдяки цьому під час виконання операції проекції виділяється «вертикальна» вибірка відношення-операнда зі знищенням кортежів.

6.2.4.3. Операція сполучення відношень

Загальна операція сполучення (називається також *сполученням за умовою*) вимагає наявності двох операндів – відношень, що з'єднуються, і третього операнда – простої умови. Нехай сполучаються відношення A і B.

Як і у випадку операції обмеження, умова сполучення comp має вигляд або (a comp-op b), або (a comp-op const), де a і b – імена атрибутів відношень A і B, const – літерально задана константа, comp-op – допустима у цьому контексті операція порівняння. Тоді, за визначенням, результатом операції сполучення є відношення, одержуване шляхом виконання операції обмеження за умовою comp прямого добутку відношень A і B. Застосування умови сполучення істотно зменшить потужність результату проміжного прямого добутку відношень-операндів тільки в тому випадку, коли умова сполучення має вигляд (a comp-op b), де a і b – імена атрибутів різних відношень-операндів. Тому на практиці зазвичай вважають реальними операціями сполучення саме ті операції, що ґрунтуються на умові сполучення наведеного вигляду. Хоча операція сполучення не є примітивною (оскільки вона визначається використанням прямого добутку та проекції), завдяки її особливій практичній важливості вона входить у базовий набір операцій реляційної алгебри. Зазначимо також, що сполучення, як правило, не виконується саме як обмеження прямого добутку. Існують більш ефективні алгоритми, що гарантують отримання такого самого результату.

Можуть бути важливі окремі випадки сполучення:

- еквісполучення;
- просте, але важливе розширення операції еквісполучення – *природне сполучення*.

Операцію сполучення називають операцією *еквісполучення*, якщо умова сполучення має вигляд (a = b), де a і b – атрибути різних операндів сполучення. Цей випадок важливий, оскільки:

- він часто трапляється на практиці;
- для нього існують ефективні алгоритми реалізації.

Операція природного сполучення застосовується до пари відношення A і B, що мають (можливо складений) загальний атрибут C (тобто атрибут з одним і тим самим ім'ям, визначеним на одному і тому самому домені).

Нехай ab означає об'єднання заголовків відношень A і B (перелік атрибутів без повторів). Тоді природне сполучення A і B – це проекція на ab результату еквісполучення A та B по A/c і B/c. Зрозуміло, що основний сенс операції природного сполучення – можливість відновлення складної сутності, декомпозованої через вимогу першої нормальної форми. Операція природного сполучення не входить до набору операцій реляційної алгебри, але має дуже важливе практичне значення.

6.2.4.4. Операція ділення відношень

Ця операція є найменш очевидною з усіх операцій реляційної алгебри і тому потребує більш докладного пояснення. Нехай задано два відношення – A із заголовком $\{a_1, a_2, ..., a_n, b_1, b_2, ..., b_m\}$ і B із заголовком $\{b_1, b_2, ..., b_m\}$. Будемо вважати, що атрибут b_1 відношення A та атрибут b_1 відношення B не тільки мають одне і те саме ім'я, але і визначені на одному і тому самому домені. Назвемо множину атрибутів $\{a_j\}$ складеним атрибутом a, а множину атрибутів $\{b_j\}$ – складеним атрибутом b. Визначимо поняття реляційного ділення бінарного відношення A (a, b) на унарне відношення B (b). Результатом ділення A на B є унарне відношення C (a), яке складається з кортежів v, таких, що у відношенні A є кортежі <v, w>, для яких множина значень {w} містить множину значень атрибута b у відношенні B. Припустимо, що у БД співробітників підтримуються два відношення: СПІВРОБІТНИКИ (ІМ'Я, ВІД_НОМЕР) і ІМЕНА (ІМ'Я), причому унарне відношення ІМЕНА містить всі прізвища співробітників організації. Тоді після виконання операції реляційного ділення відношення СПІВРОБІТНИКИ на відношення ІМЕНА буде отримане унарне відношення, що містить номери відділів, співробітники яких мають усі імена, які трапляються в цій організації.

6.3. Реляційне обчислення

Припустимо, що працюємо з БД, яка визначається схемою: СПІВРОБІТНИКИ (СПІВ_НОМ, СПІВ_ІМ'Я, СПІВ_ЗАРП, ВІД_НОМ) і ВІДДІЛИ (ВІД_НОМ, ВІД_КІЛЬК, ВІД_НАЧ), і хочемо отримати імена і номери співробітників, що є начальниками відділів з кількістю співробітників більше ніж 50. Якщо для формулювання такого запиту використовувати реляційну алгебру, то необхідно створити такий алгебричний вираз:

- виконати сполучення відношень СПІВРОБІТНИКИ і ВІДДІЛИ за умовою СПІВ_НОМ = ВІД_НАЧ;
- обмежити отримане відношення за умовою ВІД_КІЛЬК>50;
- виконати проекцію результату попередньої операції на атрибут СПІВ_ІМ'Я, СПІВ_НОМ.

У прикладі чітко визначено послідовність кроків виконання запиту, кожний з яких відповідає одній реляційній операції. Якщо ж сформулювати тий

самий запит з використанням реляційного обчислення, про яке йдеться у цьому розділі, то отримаємо формулу, яку можна було б прочитати так: видати СПІВ_ІМ'Я та СПІВ_НОМ для тих співробітників, для яких існує відділ з відповідним значенням ВІД_НАЧ та значенням ВІД_КІЛЬК, більшим від 50. У другому формулюванні вказано лише характеристики відношення-результату, але нічого не сказано про засіб його формування. У цьому випадку система має сама вирішити, які операції і в якому порядку потрібно виконати над відношеннями СПІВРОБІТНИКИ і ВІДДІЛИ. Вважають, що алгебричне формулювання є процедурним, тобто таким, що задає правила виконання запиту, а логічне – описовим (або декларативним), оскільки воно тільки описує властивості бажаного результату. Насправді ці два формулювання є еквівалентними, і існують не дуже складні правила перетворення одного формулювання на інше.

6.3.1. Кортежні змінні і правильно побудовані формули

Реляційне обчислення є частиною формального механізму обчислення предикатів першого порядку. Базисними поняттями обчислення є поняття змінної з певною для неї областю допустимих значень і поняття правильно побудованої формули, що спирається на змінні, предикати і квантори. Залежно від того, що є областю визначення змінної, розрізняють обчислення кортежів і обчислення доменів. В обчисленні кортежів областями визначення змінних є відношення БД, тобто допустимим значенням кожної змінної є кортеж деякого відношення. В обчисленні доменів областями визначення змінних є домени, на яких визначені атрибути відношень БД, тобто допустимим значенням кожної змінної є значення деякого домену. Розглянемо більш докладно обчислення кортежів і коротко опишемо особливості обчислення доменів. Потрібні для опису синтаксичні конструкції будуть вводитися у міру необхідності. Для визначення кортежної змінної використовується оператор RANGE. Наприклад, для того, щоб визначити змінну СПІВРОБІТНИК, областю визначення якої є відношення СПІВРОБІТНИКИ, потрібно вжити конструкцію

```
RANGE СПІВРОБІТНИК IS СПІВРОБІТНИКИ.
```

З цього визначення випливає, що у будь-який момент часу змінна СПІВРОБІТНИК представляє деякий кортеж відношення СПІВРОБІТНИКИ. У разі використання кортежних змінних у формулах можна посилатися на значення атрибута змінної (це аналогічно тому, як, наприклад, під час програмування мовою Сі можна посилатися на значення поля структурної змінної). Наприклад, для того, щоб посилатися на значення атрибута СПІВ_ІМ'Я змінної СПІВРОБІТНИК, слід використати конструкцію

```
СПІВРОБІТНИК. СПІВ_ІМ'Я.
```

Правильно побудовані формули (WFF – Well-Formed Formula) використовують для вираження умов, що накладаються на кортежні змінні. Основою WFF є прості порівняння (*comparison*), які являють собою операції порівняння скалярних значень (значень атрибутів змінних або літерально заданих констант). Наприклад, конструкція "СПІВРОБІТНИК.СПІВ_НОМ = 140" є простим порівнянням. Просте порівнянням є WFF. Позначимо, що WFF, взята у круглі дужки, є простим порівнянням. Більш складні варіанти WFF будуються за допомогою логічних зв'язків: NOT, AND, OR і IF ... THEN. Так, якщо form – WFF, а comp – просте порівняння, то NOT form, comp AND form, comp OR form і IF comp THEN form є WFF. Нарешті, допускається побудова WFF за допомогою кванторів. Якщо form – це WFF, в якій бере участь змінна var, то конструкції EXISTS var (form) і FORALL var (form) представляють WFF. Змінні, що входять у WFF, можуть бути вільними або зв'язаними. Усі змінні, що входять у WFF, для побудови якої не використовувались квантори, є вільними. Фактично це означає, що якщо для якого-небудь набору значень вільних кортежних змінних після обчислення WFF отримано значення *true*, то ці значення кортежних змінних можуть входити у відношення-результат. Якщо ж ім'я змінної використано відразу після квантора під час побудови WFF вигляду EXISTS var (form) або FORALL var (form), то в цій WFF і в усіх WFF, побудованих за її участю, var – це зв'язана змінна. Це означає, що таку змінну не видно за межами мінімальної WFF, яка зв'язала цю змінну. Під час обчислення значення такої WFF використовується не одне значення зв'язаної змінної, а вся її область визначення. Нехай СПІВ1 і СПІВ2 – дві кортежні змінні, визначені на відношенні СПІВРОБІТНИКИ.

Приклад 1: WFF

```
EXISTS СПІВ2 (СПІВ1.СПІВ_ЗАРП > СПІВ2.СПІВ_ЗАРП)
```

Для поточного кортежу змінної СПІВ1 набуває значення *true* лише тоді, коли у всьому відношенні СПІВРОБІТНИКИ знайдеться такий кортеж (зв'язаний зі змінною СПІВ2), що значення його атрибута СПІВ_ЗАРП задовольнить внутрішню умову порівняння.

Приклад 2: WFF

```
FORALL СПІВ2 (СПІВ1.СПІВ_ЗАРП > СПІВ2.СПІВ_ЗАРП)
```

Для поточного кортежу змінної СПІВ1 набуває значення *true* в тому і тільки в тому випадку, якщо для усіх кортежів відношення СПІВРОБІТНИКИ (зв'язаних зі змінною СПІВ2) значення атрибута СПІВ_ЗАРП задовольняє умову порівняння. Насправді необхідно говорити не про вільні і зв'язані змінні, а про вільні і зв'язані входження змінних. Легко побачити, що якщо змінна var є зв'язаною у WFF form, то в усіх WFF, що містять цю змінну, може використовуватися ім'я змінної

var, яка може бути вільною або зв'язаною, але у будь-якому випадку це не стосується входження змінної var у WFF form.

Наприклад:

```
EXISTS СПІВ2 (СПІВ1.СПІВ_ВІД_НОМ = СПІВ2.СПІВ_ВІД_НОМ)
AND FORALL СПІВ2 (СПІВ1.СПІВ_ЗАРП > СПІВ2.СПІВ_ЗАРП)
```

Тут маємо два зв'язаних входження змінної СПІВ2 з цілком різним сенсом.

6.3.2. Цільові списки і вирази реляційного обчислення

WFF забезпечують засоби формулювання умови вибірки з відношень БД. Для того щоб можна було використати обчислення для реальної роботи з БД, потрібний ще один компонент, що визначає набір та імена стовпчиків відношення-результату. Цей компонент називається *цільовим списком* (target_list). Цільовий список будується з цільових елементів, кожний з яких може мати такий вигляд:

- var.attr, де var – ім'я вільної змінної відповідної WFF, attr – ім'я атрибута відношення, на якому визначена змінна var;
- var, що еквівалентно підпису $var.attr_1$, $var.attr_2$, ..., $var.attr_n$, де $attr_1$, $attr_2$, ..., $attr_n$ – імена всіх атрибутів відношення, що визначається;
- new_name = var.attr; new_name – нове ім'я відповідного атрибута відношення-результату.

Останній варіант потрібний в тих випадках, коли у WFF використовуються декілька вільних змінних з однаковою областю визначення. *Виразом реляційного обчислення кортежів* називають конструкцію вигляду

```
target_list WHERE wff.
```

Значенням виразу є відношення, тіло якого визначається WFF, а набір атрибутів і їхні імена – цільовим списком.

6.3.3. Реляційне обчислення доменів

В обчисленні доменів областю визначення змінних є не відношення, а домени. Стосовно БД СПІВРОБІТНИКИ-ВІДДІЛИ можна говорити, наприклад, про доменні змінні ІМ'Я (значення – допустимі імена) або СПІВ_НОМ (значення – допустимі номери співробітників). Основною формальною відмінністю обчислення доменів від обчислення кортежів є наявність додаткового набору предикатів, що дозволять виражати так звані *умови членства*. Якщо R – це *n*-арне відношення з атрибутами $<a_1, a_2, ..., a_n>$, то умова членства має вигляд R (a_{i1}: v_{i1}, a_{i2}: v_{i2}, ..., a_{im}: v_{im}) (m<=n), де v_{ij} – це або літеральна константа, що задається, або ім'я доменної змінної. Умова членства набуває значення *true* тоді і лише тоді, коли у відношенні R існує кортеж, який містить задані значення означених атрибутів. Якщо v_{ij} – констан-

та, то на атрибут a_{ij} накладається жорстка умова, яка не залежить від поточних значень доменних змінних; якщо ж v_{ij} – ім'я доменної змінної, то умова членства може набувати різних значень при різних значеннях цієї змінної. В усіх інших відношеннях формули і вирази обчислення доменів мають такий самий вигляд, що і формули та вирази обчислення кортежів. В окремих випадках, звичайно, розрізняють вільні та зв'язані входження доменних змінних. Для прикладу сформулюємо з використанням обчислення доменів запит «Видати номери та імена співробітників, які отримують мінімальну заробітну плату» (будемо вважати для простоти, що визначили доменні змінні, імена яких збігаються з іменами атрибутів відношення СПІВРОБІТНИКИ, а у випадку, коли потрібні будуть декілька доменних змінних, визначених на одному домені, будемо додавати в кінці імені цифри), наприклад:

```
СПІВ_НОМ, СПІВ_ІМ'Я
WHERE EXISTS ( СПІВ_ЗАРП1(СПІВРОБІТНИКИ (СПІВ_ЗАРП1)
    AND СПІВРОБІТНИКИ (СПІВ_НОМ, СПІВ_ІМ'Я, СПІВ_ЗАРП))
  AND СПІВ_ЗАРП>СПІВ_ЗАРП1)
```

Реляційне обчислення доменів є основою більшості мов запитів, що базуються на використанні форм. Зокрема, на цьому обчисленні базувалася відома мова Query-by-Example, що була першою (і найбільш цікавою) в сім'ї мов, основою яких є табличні форми.

7. Проектування реляційних баз даних за принципами нормалізації

7.1. Основні проблеми проектування

Під час проектування БД вирішують дві основні проблеми:

- проблему логічного проектування. Вирішення проблеми має забезпечити відображення об'єктів ПО в абстрактні об'єкти моделі даних так, щоб це відображення не суперечило семантиці ПО і було якомога кращим (ефективним, зручним);
- проблему фізичного проектування. Вирішення цієї проблеми має забезпечити ефективне виконання запитів до БД, тобто передбачає раціональне розташування даних у зовнішній пам'яті, створення корисних додаткових структур (наприклад, індексів) з урахуванням особливостей конкретної СКБД.

У випадку реляційних БД важко уявити які-небудь загальні рекомендації щодо фізичного проектування. Тут багато залежить від СКБД, яку використовують. Наприклад, під час роботи з СКБД Ingres можна вибирати один із запропонованих засобів фізичної організації відношень, під час роботи із System R слід було б передусім визначитись стосовно побудови кластерів відношень та необхідного набору індексів і т. д.

У посібнику буде розглянуто питання логічного проектування реляційних БД, яке є дуже важливими під час використання будь-якої реляційної СКБД з розглядом визначення обмежень цілісності тільки для первинного ключа. Справа в тому, що під час використання СКБД з розвиненими механізмами обмежень цілісності (наприклад, систем, орієнтованих на SQL) важко запропонувати будь-який загальний підхід до визначення обмежень цілісності. Ці обмеження можуть мати дуже загальний вигляд, і їх формулювання скоріше належить до сфери мистецтва, ніж інженерної майстерності. Загальний підхід – це автоматична перевірка несуперечливого набору обмежень цілісності.

Тому будемо вважати, що проблема проектування реляційної БД полягає в обґрунтованому прийнятті рішень про те, з яких відношень має складатися БД і які атрибути мають бути у цих відношеннях.

7.2. Принципи нормалізації. Перша нормальна форма

Спочатку розглянемо класичний підхід, за якого процес проектування реляційної БД виконується в термінах реляційної моделі даних засобом послідовних наближень до задовільного набору схем відношень. Початковим є подання ПО у вигляді одного або декількох відношень, і на кож-

ному етапі проектування виробляється деякий набір схем відношень, які мають кращі властивості. Процес проектування являє собою процес нормалізації схем відношень, причому кожна наступна нормальна форма має кращі властивості, ніж попередня.

Кожній нормальній формі відповідає деякий певний набір обмежень, і відношення знаходиться в деякій нормальній формі, якщо задовольняє властивий їй набір обмежень. Прикладом є обмеження першої нормальної форми: значення всіх атрибутів відношення мають бути атомарними, тобто не розподілятися на складові частини. Оскільки вимога першої нормальної форми є базовою вимогою класичної реляційної моделі даних, будемо вважати, що вхідний набір відношень вже відповідає цій вимозі.

У теорії реляційних БД, як правило, виділяють таку послідовність нормальних форм:

- перша нормальна форма (1NF);
- друга нормальна форма (2NF);
- третя нормальна форма (3NF);
- нормальна форма Бойса – Кодда (BCNF) (правильніше було б вважати цю нормальну форму третьою, однак з історичних причин третя сходинка виявилася зайнятою на час винаходу BCNF, через що вона і отримала нестандартну назву);
- четверта нормальна форма (4NF);
- п'ята нормальна форма, або нормальна форма проекції – сполучення (5NF або PJ/NF).

Основні властивості нормальних форм:

- кожна наступна нормальна форма в деякому розумінні поліпшує властивості попередньої;
- у разі переходу до наступної нормальної форми властивості попередніх нормальних форм зберігаються.

В основу класичного процесу проектування покладено засіб нормалізації, що спирається на декомпозицію (на основі проекції) відношення, яке знаходиться в попередній нормальній формі, в два або більше відношень, які задовольнять вимоги наступної нормальної форми.

Найважливіші на практиці нормальні форми відношень ґрунтуються на фундаментальному в теорії реляційних БД понятті функціональної залежності. Для подальшого викладення необхідно ввести ряд визначень.

Визначення 1. У відношенні R атрибут Y функціонально залежить від атрибута X (X і Y можуть бути складеними атрибутами, тобто реально складатися з декількох атомарних атрибутів) в тому і тільки в тому випадку, якщо кожному значенню X відповідає тільки одне значення Y: R. X -> R. Y.

Зауваження. Термін «функціональна залежність» не є випадковим, оскільки повністю відповідає математичному поняттю функції.

Визначення 2. Функціональну залежність R.X->R.Y називають *повною*, якщо атрибут Y не залежить функціонально від будь-якої точної підмножину X (*точною підмножиною* множини X називають будь-яку її підмножина, яка не збігається з множиною X).

Визначення 3. Функціональну залежність R.X->R.Y називають *транзитивною*, якщо існує такий атрибут Z, що існують функціональні залежності R.X->R.Z і R.Z->R.Y

Визначення 4. Неключовим атрибутом називають будь-який атрибут відношення, що не входить до складу первинного ключа.

Визначення 5. Можливим ключем відношення називають його атомарний або складений атрибут, значення якого повністю функціонально визначають значення всіх інших атрибутів відношення.

Визначення 6. Два або більше атрибутів називають *взаємно незалежними*, якщо жоден з них не є функціонально залежним від інших атрибутів.

7.2.1. Друга нормальна форма

Розглянемо такий приклад схеми відношення:

СПІВРОБІТНИКИ-ВІДДІЛИ-ПРОЕКТИ

(СПІВ_НОМЕР, СПІВ_ЗАРП, ВІД_НОМЕР, ПРО_НОМЕР, СПІВ_ЗАВД)

Первинний ключ:

СПІВ_НОМЕР, ПРО_НОМЕР

Функціональні залежності:

СПІВ_НОМЕР -> СПІВ_ЗАРП

СПІВ_НОМЕР -> ВІД_НОМЕР

ВІД_НОМЕР -> СПІВ_ЗАРП

СПІВ_НОМЕР, ПРО_НОМЕР -> СПІВ_ЗАВД

Відношення СПІВРОБІТНИКИ-ВІДДІЛИ-ПРОЕКТИ:

СПІВ_НОМЕР	СПІВ_ЗАРП	ВІД_НОМЕР	ПРО_НОМЕР	СПІВ_ЗАВД
001	100	001	001	001
002	100	001	001	001
003	100	001	002	001
004	100	001	002	002
005	200	002	001	001
006	200	002	001	002
007	200	002	002	001
008	200	002	002	002

Первинний ключ:

СПІВ_НОМЕР, ПРО_НОМЕР

Функціональні залежності:

СПІВ_НОМЕР -> СПІВ_ЗАРП

СПІВ_НОМЕР -> ВІД_НОМЕР

ВІД_НОМЕР -> СПІВ_ЗАРП

СПІВ_НОМЕР, ПРО_НОМЕР -> СПІВ_ЗАВД

Як видно, хоча первинним ключем є складений атрибут СПІВ_НОМЕР, ПРО_НОМЕР, атрибути СПІВ_ЗАРП і ВІД_НОМЕР функціонально залежать від частини первинного ключа, тобто атрибута СПІВ_НОМЕР. У результаті, не можна вставити у відношення СПІВРОБІТНИКИ-ВІДДІЛИ-ПРОЕКТИ кортеж, який описує співробітника, котрий ще не виконує жодного проекту (первинний ключ не може містити невизначене значення). Під час вилучення кортежу не тільки руйнується зв'язок даного співробітника з даним проектом, але втрачається інформація про те, що він працює в деякому відділі. У разі переведення співробітника в інший відділ, необхідно модифікувати всі кортежі, які описують цього співробітника, інакше отримаємо непогоджений результат. Такі неприємні явища називають *аномаліями схеми відношення*. Їх усувають шляхом нормалізації.

Визначення 7. Відношення R знаходиться у другій нормальній формі (2NF) тоді і лише тоді, коли знаходиться в 1NF, і кожний неключовий атрибут повністю залежить від первинного ключа.

Для зведення відношення R до другої нормальної форми можна провести наступну декомпозицію відношення СПІВРОБІТНИКИ-ВІДДІЛИ-ПРОЕКТИ у два відношення СПІВРОБІТНИКИ-ВІДДІЛИ і СПІВРОБІТНИКИ-ПРОЕКТИ.

Відношення СПІВРОБІТНИКИ-ВІДДІЛИ (СПІВ_НОМЕР, СПІВ_ЗАРП, ВІД_НОМЕР):

СПІВ_НОМЕР	СПІВ_ЗАРП	ВІД_НОМЕР
001	100	001
002	100	001
003	100	001
004	100	001
005	200	002
006	200	002
007	200	002
008	200	002

Первинний ключ:

СПІВ_НОМЕР

Функціональні залежності:

СПІВ_НОМЕР -> СПІВ_ЗАРП

СПІВ_НОМЕР -> ВІД_НОМЕР

ВІД_НОМЕР -> СПІВ_ЗАРП

Відношення СПІВРОБІТНИКИ-ПРОЕКТИ (СПІВ_НОМЕР, ПРО_НОМЕР, СПІВ_ЗАВД):

СПІВ_НОМЕР	ПРО_НОМЕР	СПІВ_ЗАВД
001	001	001
002	001	002
003	002	001
004	002	002
005	001	001
006	001	002
007	002	001
008	002	002

Первинний ключ:

СПІВ_НОМЕР, ПРО_НОМЕР

Функціональна залежність:

СПІВ_НОМЕР, ПРО_НОМЕР -> СПІВ_ЗАВД

Кожне з цих двох відношень знаходиться в 2NF, і в них усунені відзначені вище аномалії.

7.2.2. Третя нормальна форма

Розглянемо ще раз відношення СПІВРОБІТНИКИ-ВІДДІЛИ, яке знаходиться у 2NF. Відзначимо, що функціональна залежність СПІВ_НОМЕР -> СПІВ_ЗАРП є транзитивною, тобто вона є наслідком (у математичному розумінні) функціональних залежностей СПІВ_НОМЕР -> ВІД_НОМЕР та ВІД_НОМЕР -> СПІВ_ЗАРП. Інакше кажучи, заробітна плата співробітника насправді є характеристикою не співробітника, а відділу, в якому він працює (це не дуже природне припущення, але достатнє для прикладу).

У результаті, не можна занести у БД інформацію, що характеризує заробітну плату відділу, доки в цьому відділі не з'явиться хоча б один співробітник (первинний ключ не може містити невизначене значення). У разі вилучення кортежу, що описує останнього співробітника даного відділу, втрачається інформація про заробітну плату відділу. Для того, щоб за погодженням змінити заробітну плату відділу, необхідно заздалегідь знайти всі кортежі, що описують співробітників цього відділу. Тобто у відношенні СПІВРОБІТНИКИ-ВІДДІЛИ й досі існують аномалії. Їх можна усунути шляхом подальшої нормалізації.

Визначення 8. Відношення R знаходиться в третій нормальній формі (3NF) в тому і тільки в тому випадку, якщо воно знаходиться в 2NF і кожний неключовий атрибут нетранзитивно залежить від первинного ключа.

Еквівалентним альтернативним визначенням третьої нормальної форми є визначення 9.

Визначення 9. Відношення R знаходиться в третій нормальній формі (3NF) в тому і тільки в тому випадку, якщо всі неключові атрибути R взаємно незалежні і повністю залежать від первинного ключа.

Виконаємо декомпозицію відношення СПІВРОБІТНИКИ-ВІДДІЛИ у два відношення: СПІВРОБІТНИКИ і ВІДДІЛИ.

Відношення СПІВРОБІТНИКИ (СПІВ_НОМЕР, ВІД_НОМЕР):

СПІВ_НОМЕР	ВІД_НОМЕР
001	001
002	001
003	001
004	001
005	002
006	002
007	002
008	002

Первинний ключ:

СПІВ_НОМЕР

Функціональна залежність:

СПІВ_НОМЕР->ТД_НОМЕР

Відношення ВІДДІЛИ (ВІД_НОМЕР, СПІВ_ЗАРП):

ВІД_НОМЕР	СПІВ_ЗАРП
001	100
002	200

Первинний ключ:

ВІД_НОМЕР

Функціональна залежність:

ВІД_НОМЕР->СПІВ_ЗАРП

Кожне з цих двох відношень знаходиться в 3NF і не залежить від описаних аномалій.

На практиці третя нормальна форма схем відношень є достатньою у більшості випадків, і зведенням до третьої нормальної форми процес проектування реляційної БД, як правило, закінчується. Однак інколи доцільно продовжити процес нормалізації.

7.2.3. Нормальна форма Бойса – Кодда

Припустимо, що для відношення мають місце умови:

1) відношення має два (або більше) альтернативні ключі;

2) два альтернативні ключі є складеними;

3) складені ключі перекриваються (тобто мають спільний атрибут).

У цьому разі визначення третьої нормальної форми удосконалюється (для усунення аномалій).

Розглянемо такий приклад схеми відношення:

СПІВРОБІТНИКИ-ПРОЕКТИ(СПІВ_НОМЕР, СПІВ_ІМ'Я, ПРО_НОМЕР, СПІВ_ЗАВД)

У цьому прикладі припускаємо, що особистість співробітника повністю визначається як його номером, так і ім'ям.

Відношення СПІВРОБІТНИКИ-ПРОЕКТИ:

СПІВ_НОМЕР	СПІВ_ІМ'Я	ПРО_НОМЕР	СПІВ_ЗАВД
001	Кравчук	001	001
002	Іванов	001	002
003	Сидоренко	002	001
004	Слінченко	002	002
005	Дзюба	001	001
006	Ткаченко	001	002
007	Петренко	002	001
008	Корнієнко	002	002

Можливі ключі (зазначимо, що на цій стадії нормалізації слід звернути увагу на існування можливих ключів):

СПІВ_НОМЕР, ПРО_НОМЕР

СПІВ_ІМ'Я, ПРО_НОМЕР

Функціональні залежності:

СПІВ_НОМЕР->СПІВ_ІМ'Я

СПІВ_НОМЕР->ПРО_НОМЕР

СПІВ_ІМ'Я->СПІВ_НОМЕР

СПІВ_ІМ'Я->ПРО_НОМЕР

СПІВ_НОМЕР, ПРО_НОМЕР->СПІВ_ЗАВД

СПІВ_ІМ'Я, ПРО_НОМЕР->СПІВ_ЗАВД

Незалежно від того, який з можливих ключів обраний як первинний ключ, ця схема знаходиться в 3NF. Однак той факт, що існують функціональні залежності атрибутів відношення від атрибута, який є частиною первинного ключа, призводить до аномалій. Наприклад, для того, щоб змінити ім'я співробітника з даним номером за погодженням, необхідно модифікувати всі кортежі, які містять його номер.

Визначення 10. Детермінантом називають будь-який атрибут, від якого повністю функціонально залежить деякий інший атрибут.

Визначення 11. Відношення R знаходиться у нормальній формі Бойса – Кодда в тому і тільки в тому випадку, якщо кожний детермінант є можливим ключем.

З цього визначення випливає, що в цій формі кожний з неключових атрибутів повністю визначається первинним ключем і не залежить від його частини (у разі складеного первинного ключа).

Зауваження. Легко помітити, що якщо у відношенні є тільки один можливий ключ (і він є первинним ключем), то це визначення стає еквівалентним визначенню третьої нормальної форми.

Очевидно, що ця вимога не виконана для відношення СПІВРОБІТНИКИ-ПРОЕКТИ. Можна зробити його декомпозицію до відношень СПІВРОБІТНИКИ і СПІВРОБІТНИКИ-ПРОЕКТИ.

Відношення СПІВРОБІТНИКИ (СПІВ_НОМЕР, СПІВ_ІМ'Я).

СПІВ_НОМЕР	СПІВ_ІМ'Я
001	Кравчук
002	Іванов
003	Сидоренко
004	Сліпченко
005	Дзюба
006	Ткаченко
007	Петренко
008	Корнієнко

Можливі ключі:

СПІВ_НОМЕР

СПІВ_ІМ'Я

Функціональні залежності:

СПІВ_НОМЕР->СПІВ_ІМ'Я

СПІВ_ІМ'Я->СПІВ_НОМЕР

Відношення СПІВРОБІТНИКИ-ПРОЕКТИ

(СПІВ_НОМЕР, ПРО_НОМЕР, СПІВ_ЗАВД):

СПІВ_НОМЕР	ПРО_НОМЕР	СПІВ_ЗАВД
001	001	001
002	001	002
003	002	001
004	002	002
005	001	001
006	001	002
007	002	001
008	002	002

Можливий ключ:

СПІВ_НОМЕР, ПРО_НОМЕР

Функціональні залежності:

СПІВ_НОМЕР, ПРО_НОМЕР->СПІВ_ЗАВД

Можлива альтернативна декомпозиція, якщо взяти за основу СПІВ_ІМ'Я.

В обох випадках одержані відношення СПІВРОБІТНИКИ та СПІВРОБІТНИКИ-ПРОЕКТИ **знаходяться у** BCNF.

7.2.4. Четверта нормальна форма

Припустимо, що будь-який співробітник, що бере участь у проекті, виконує всі завдання, передбачені цим проектом.

Розглянемо приклад наступної схеми відношення.

Відношення ПРОЕКТИ (ПРО_НОМЕР, ПРО_СПІВ, ПРО_ЗАВД):

ПРО_НОМЕР	ПРО_СПІВ	ПРО_ЗАВД
001	001	001
001	001	002
001	002	001
001	002	002
002	003	001
002	003	002
002	004	001
002	004	002

Відношення ПРОЕКТИ містить номери проектів, для кожного проекту – список співробітників, які можуть виконувати проект, і список завдань, що передбачає проект. Співробітники можуть брати участь у декількох проектах, і різні проекти можуть містити однакові номери завдань.

Кожний кортеж відношення зв'язує деякий проект зі співробітником, що бере участь у цьому проекті, і завданням, яке співробітник виконує у рамках проекту. Враховуючи умови, сформульовані вище, єдиним можливим ключем відношення є складений атрибут ПРО_НОМЕР, ПРО_СПІВ, ПРО_ЗАВД, і немає жодних інших детермінантів. Отже, відношення ПРОЕКТИ знаходиться у BCNF. Але при цьому воно має недоліки: якщо, наприклад, деякий співробітник приєднується до проекту, необхідно вставити у відношення ПРОЕКТИ стільки кортежів, скільки завдань у ньому передбачено.

Визначення 12. У відношенні R (A, B, C) існує багатозначна залежність, яку позначають як R.A->->R.B тоді і лише тоді, коли множина значень B, яка відповідає парі значень A і C, залежить тільки від A і не залежить від C.

У відношенні ПРОЕКТИ існують дві багатозначні залежності:

ПРО_НОМЕР->->ПРО_СПІВ

ПРО_НОМЕР->->ПРО_ЗАВД

Фейджин показав, що у загальному випадку у відношенні R (A, B, C) існує багатозначна залежність R.A->->R.B лише в тому випадку, коли існує багатозначна залежність R.A->->R.C. Отже, багатозначні залежності завжди створюють зв'язані пари, що позначають виразом A->->B|C.

Подальша нормалізація відношень, подібних до відношення ПРОЕКТИ, ґрунтується на *теоремі Фейджина*:

Відношення R (A, B, C) можна спроектувати без втрат у відношення R_1 (A, B) і R_2 (A, C) лише тоді, коли існує багатозначна залежність A->->B|C.

Під проектуванням без втрат розуміємо такий засіб декомпозиції відношення, за якого вхідне відношення повністю і без надмірності відновлюється шляхом природного сполучення отриманих відношень.

Визначення 13. Відношення R знаходиться в четвертій нормальній формі (4NF) лише в тому разі, коли за умови існування багатозначної залежності A ->->B всі інші атрибути R функціонально залежать від A.

Інакше кажучи, четверта нормальна форма не містить багатозначної залежності між атрибутами.

У прикладі можна зробити декомпозицію відношення ПРОЕКТИ у два відношення: ПРОЕКТИ-СПІВРОБІТНИКИ і ПРОЕКТИ-ЗАВДАННЯ.

Відношення ПРОЕКТИ-СПІВРОБІТНИКИ (ПРО_НОМЕР, ПРО_СПІВ):

ПРО_НОМЕР	ПРО_СПІВ
001	001
001	002
002	003
002	004

Відношення ПРОЕКТИ-ЗАВДАННЯ (ПРО_НОМЕР, ПРО_ЗАВД):

ПРО_НОМЕР	ПРО_ЗАВД
001	001
001	002
002	001
002	002

Обидва відношення знаходяться в 4NF.

7.2.5. П'ята нормальна форма

В усіх розглянутих до цього моменту нормалізаціях проводилась декомпозиція одного відношення у два. Інколи це зробити не вдається, але можлива декомпозиція у більшу кількість відношень, кожне з яких має кращі властивості.

Припустимо, що один і той самий співробітник може працювати у декількох відділах і в кожному відділі брати участь у декількох проектах.

Розглянемо відповідне відношення:

СПІВРОБІТНИКИ-ВІДДІЛИ-ПРОЕКТИ (СПІВ_НОМЕР,ВІД_НОМЕР, ПРО_НОМЕР)

Первинним ключем цього відношення є повна сукупність його атрибутів. У відношенні немає функціональних і багатозначних залежностей, тому відношення знаходиться в 4NF. Однак у ньому можуть існувати аномалії, які можна усунути шляхом декомпозиції у три відношення.

Для відношення R із n атрибутів можна побудувати n проекцій по кожному з цих атрибутів. Сполучення отриманих відношень (всіх або деяких) являє собою нове відношення. Таке відношення може або збігатися з первинним (тобто відновлюється без втрат), або ні.

Визначення 14. Відношення R (X, Y, ..., Z) задовольняє *залежність сполучення* *(X, Y, ..., Z) в тому і тільки в тому випадку, коли R відновлюється без втрат шляхом сполучення своїх проекцій на X, Y, ..., Z.

Інакше кажучи, відношення залежить від сполучення своїх проекцій

Визначення 15. П'ята нормальна форма. Відношення R знаходиться у п'ятій нормальній формі (нормальній формі проекції-сполучення – PJ/NF) в тому і тільки в тому випадку, коли будь-яка залежність сполучення в R створюється з проекцій за ключовими атрибутами, які в сукупності є можливим ключем відношення.

Уведемо імена складених атрибутів:

```
СВ = {СПІВ_НОМЕР, ВІД_НОМЕР}
СП = {СПІВ_НОМЕР, ПРО_НОМЕР}
ВП = {ВІД_НОМЕР, ПРО_НОМЕР}
```

Припустимо, що у відношенні СПІВРОБІТНИКИ-ВІДДІЛИ-ПРОЕКТИ існує залежність сполучення: (СВ, СП, ВП)

На прикладах можна легко показати, що під час вставлення та вилучення кортежів можуть виникнути проблеми. Їх можна уникнути шляхом декомпозиції вхідного відношення у три нові відношення:

```
СПІВРОБІТНИКИ-ВІДДІЛИ (СПІВ_НОМЕР, ВІД_НОМЕР)
СПІВРОБІТНИКИ-ПРОЕКТИ (СПІВ_НОМЕР, ПРО_НОМЕР)
ВІДДІЛИ-ПРОЕКТИ (ВІД_НОМЕР, ПРО_НОМЕР)
```

П'ята нормальна форма – це остання нормальна форма, яку можна отримати шляхом декомпозиції з 4NF Її умови достатньо нетривіальні, і на практиці 5NF не використовується. Відзначимо, що залежність сполучення є узагальненням як багатозначної, так і функціональної залежностей.

8. Семантичне моделювання даних. ER-діаграми

8.1. Обмеженість реляційних моделей

Широке розповсюдження реляційних СКБД і їх використання у різноманітних додатках свідчать, що реляційна модель даних є достатньою для моделювання ПО. Однак проектування реляційної БД у термінах відношень на базі розглянутого механізму нормалізації часто являє собою дуже складний і незручний для проектувальника процес.

При цьому виявляється обмеженість реляційної моделі даних у таких аспектах:

- модель не надає достатніх засобів для подання сенсу даних. Семантика реальної ПО має відображатися незалежним від моделі засобом. Зокрема, це стосується проблеми подання обмежень цілісності;
- для багатьох додатків важко моделювати ПО на підставі плоских таблиць. На початковій стадії проектування проектувальнику важко описати ПО у вигляді однієї (можливо, навіть ненормалізованої) таблиці;
- незважаючи на те, що весь процес проектування відбувається з урахуванням залежностей, реляційна модель не надає будь-яких засобів для подання цих залежностей;
- незважаючи на те, що процес проектування починається з виділення деяких істотних для додатка об'єктів ПО («сутностей») і виявлення зв'язків між цими сутностями, реляційна модель даних не пропонує відповідного апарату для розподілу сутностей і зв'язків.

8.2. Семантичні моделі даних

Потреби проектувальників БД у більш зручних і потужних засобах моделювання ПО зумовили виникнення семантичних моделей даних. При тому, що будь-яка розвинена семантична модель даних, так само як і реляційна модель, містить структурну, маніпуляційну і цілісну частини, головним призначенням семантичних моделей є забезпечення можливості відображення семантики даних.

Розглянемо можливі застосування семантичних моделей. Найчастіше на практиці семантичне моделювання використовують на першій стадії проектування БД. При цьому в термінах семантичної моделі виробляється концептуальна схема БД, яка після цього перетворюється на реляційну (або будь-яку іншу) схему. Цей процес виконується за методиками, в яких достатньо чітко обумовлені всі етапи такого перетворення.

Рідше реалізується автоматизована компіляція концептуальної схеми в реляційну. Відомо два підходи: підхід, що базується на явному поданні концептуальної схеми як вхідної інформації для компіляції, і підхід, орієнтований на побудову інтегрованих систем проектування з автоматизованим створенням концептуальної схеми на основі інтерв'ю з експертами ПО.

В обох випадках у результаті виробляється реляційна схема БД у третій нормальній формі.

Нарешті третя можливість, яка ще не вийшла (або тільки виходить) за межі дослідних і експериментальних проектів, – це безпосередня робота з БД в семантичній моделі, тобто СКБД, що базується на семантичних моделях даних. При цьому знову розглядають два варіанти: забезпечення інтерфейсу користувача на основі семантичної моделі даних з автоматичним відображенням конструкцій у реляційну модель даних (це завдання приблизно такого самого рівня складності, як і автоматична компіляція концептуальної схеми БД у реляційну схему) і пряма реалізація СКБД, що грунтується на будь-якій семантичній моделі даних. Найбільш близькими до другого підходу є сучасні об'єктно-зорієнтовані СКБД, моделі даних яких за своїми параметрами подібні до семантичних моделей (хоча в деяких випадках вони є більш потужними, а в деяких – слабкішими).

8.3. Побудова семантичної моделі бази даних

За основу моделювання за методологією Gane/Sarson узято побудову ІС. Відповідно до методології модель системи визначають як ієрархію діаграм потоків даних (ДПД, або DFD – Date Flou Diagrames), які описують асинхронний процес перетворення інформації від її введення у систему до видачі користувачу. Розглядаються перетворення, які є необхідними для виконання функцій системи.

Діаграми верхніх рівнів ієрархії (контекстні діаграми) визначають основні процеси із зовнішніми входами і виходами. Вони деталізуються за допомогою діаграм нижнього рівня. Така декомпозиція продовжується, створюючи багаторівневу ієрархію діаграм, доти, доки не буде досягнутий такий рівень декомпозиції, на якому процеси стають елементарними і деталізувати їх далі не потрібно.

Джерела інформації (зовнішні сутності) породжують інформаційні потоки (потоки даних), що переносять інформацію до процесів. Останні, у свою чергу, перетворюють інформацію і породжують нові потоки, що переносять інформацію до інших процесів, накопичувачів даних чи зовнішніх сутностей – споживачів інформації. Отже, основні компоненти діаграм потоків даних такі:

- зовнішні сутності;
- процеси;
- накопичувачі даних;
- потоки даних.

Побудову семантичної моделі БД виконують поетапно:

- визначення призначення та кола завдань ІС;
- відокремлення зовнішніх сутностей;
- визначення процесів ПО;
- побудова діаграм потоків даних;
- проектування структури БД (концептуальної схеми).

8.3.1. Визначення призначення та кола завдань інформаційної системи

Для визначення призначення проводить обстеження реального об'єкта господарювання, для керування яким створюється ІС, в ході якого визначають основні проблеми, які має вирішити система. Залежно від ПО існує такий розподіл ІС:

- інформаційні системи організаційного типу, які проектують з метою вирішення завдань керування процесами організації виробничих процесів;
- навчальні інформаційні системи, призначенням яких є організація навчальних процесів;
- інформаційні системи керування технологічними процесами на виробництві, тобто керування безпосередньо процесами виготовлення різноманітної продукції;
- інформаційні системи, метою функціонування яких є створення технічної документації;
- інформаційні системи, завданням яких є організація обміну інформацією між її користувачами, наприклад, різноманітні системи класу Intranet.

Визначення призначення має складатися з одного речення, в якому стисло, але повно сформульовано призначення системи. Наприклад, «Система обслуговує керування орендою нерухомості». Після формування визначення, його необхідно погодити з користувачем системи.

Для визначення завдань аналізують документацію, яка регламентує функціонування реальної системи, а також співбесіди з майбутніми користувачами різного рівня. Останнім кроком попереднього вивчення є фіксація переліку запитань до системи. Після визначення головних завдань необхідно визначити функції, за допомогою яких система вирішуватиме ці завдання. Перелік функцій відповідає переліку завдань, не виходячи за межі призначення системи.

До визначених функцій формулюють такі вимоги:

- виробничі, наприклад, періодичність виконання;
- функціональні, наприклад, визначення вигляду звітів та екранних форм;
- технічні, наприклад, обмеження на тип операційної системи, інструментальне середовище або на використання протоколів мережевого сполучення.

8.3.2. Відокремлення зовнішніх сутностей

Зовнішня сутність являє собою матеріальний предмет або фізичну особу, яка є джерелом або приймачем інформації, наприклад, замовники, персонал, постачальники, клієнти, склад. Визначення деякого об'єкта або системи як зовнішньої сутності вказує на те, що вона перебуває за межами аналізованої ІС. У процесі аналізу деякі зовнішні сутності можуть бути перенесені всередину діаграми ІС, якщо це необхідно, або навпаки, частину процесів ІС можна винести за межі діаграми і подати як зовнішню сутність (рис. 8.1).

Рис. 8.1. Зовнішня сутність

Зовнішню сутність позначають квадратом, розташованим немов би «над» діаграмою з тінню, для того, щоб можна було виділити цей символ серед інших позначень.

8.3.3. Визначення функціональних процесів предметної області

Функціональний процес у ПО являє собою перетворення вхідних потоків даних на вихідні відповідно до визначеного алгоритму, для виконання деякої функції, притаманної даній області. Фізично процес може бути реалізований різними способами: у підрозділі організації (відділі), що виконує обробку вхідних документів і випуск звітів, програма, що виконує обробку даних, апаратно реалізований логічний пристрій і тощо.

Процес на діаграмі потоків даних показано на рис. 8.2.

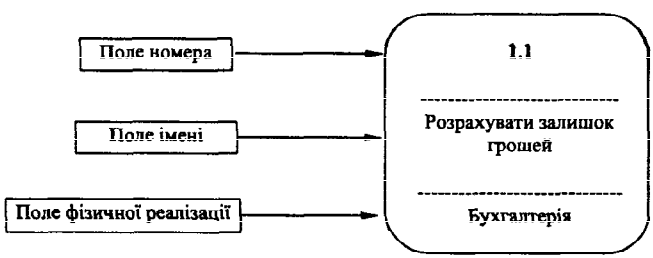


Рис. 8.2. Процес

Номер процесу використовують для його ідентифікації. У поле імені вводять найменування процесу у вигляді пропозиції з активним недвозначним дієсловом у неозначеній формі (обчислити, розрахувати, перевi-

рити, визначити, створити, одержати), за яким йдуть іменники у знахідному відмінку, наприклад:

«Увести відомості про клієнтів»;

«Видати інформацію про поточні витрати»;

«Перевірити кредитоспроможність клієнта».

Використання таких дієслів, як «обробити», «модернізувати» або «відредагувати» означає, як правило, недостатньо глибоке розуміння цього процесу і вимагає подальшого його аналізу.

Інформація в полі фізичної реалізації показує, який підрозділ організації, яка програма або апаратний пристрій виконують цей процес.

8.3.4. Побудова діаграм потоків даних

У процесі побудови діаграм потоків даних використовують поняття *накопичувача даних*.

Накопичувач даних являє собою абстрактний пристрій для збереження інформації, яку можна в будь-який момент занести в нього і через якийсь час витягти, причому способи занесення і отримання можуть бути різними.

Накопичувач даних може бути реалізований фізично у вигляді мікрофіші, шухляди в картотеці, таблиці в оперативній пам'яті, файлу на магнітному носії тощо. Накопичувач даних на діаграмі потоків даних наведено на рис. 8.3.

D11	Рахунки, які отримують

Рис. 8.3. Накопичувач даних

Накопичувач даних ідентифікують літерою D і довільним числом. Ім'я накопичувача вибирають з точки зору найбільшої інформативності для проектувальника.

Нагромаджувач даних у загальному випадку є прообразом майбутньої таблиці БД і опису даних, що зберігаються у ньому, та має бути ув'язаний з інформаційною моделлю.

Потік даних визначає інформацію, передану через з'єднання від джерела до приймача. Реальний потік даних може бути, наприклад:

- інформацією, переданою по кабелю між двома пристроями;
- листами, що пересилаються поштою;
- магнітними стрічками або дискетами, які переносять з одного комп'ютера на інший;
- іншими носіями.

Потік даних на діаграмі зображують лінією, яка закінчується стрілкою та показує напрямок потоку (рис. 8.4). Кожний потік даних має ім'я, що відображає його зміст.

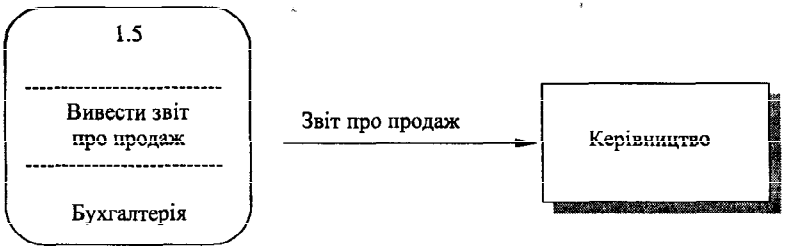


Рис 8.4. Потік даних

Далі будують *ієрархію діаграм потоків даних.* Першим кроком під час побудови ієрархії ДПД є побудова контекстних діаграм. Звичайно, під час проектування простих ІС будують єдину контекстну діаграму із зіркоподібною топологією, у центрі якої знаходиться так званий *головний процес*, з'єднаний із приймачами і джерелами інформації, за допомогою яких із системою взаємодіють користувачі й інші зовнішні системи.

Якщо для складної системи обмежитися єдиною контекстною діаграмою, то вона буде містити занадто велику кількість джерел і приймачів інформації, що важко розташувати на аркуші паперу нормального формату і, крім того, єдиний головний процес не розкриває структури розподіленої системи. Ознаками складності (за контекстом) можуть бути:

- наявність великої кількості зовнішніх сутностей (десять і більше);
- розподілена природа системи;
- багатофункціональність системи зі сформованим або виявленим групуванням функцій в окремі підсистеми.

Для складних ІС будується ієрархія контекстних діаграм. При цьому контекстна діаграма верхнього рівня містить не єдиний головний процес, а набір підсистем, з'єднаних потоками даних. Контекстні діаграми наступного рівня деталізують контекст і структуру підсистем.

Ієрархія контекстних діаграм визначає взаємодію основних функціональних підсистем проектованої ІС як між собою, так і з зовнішніми вхідними і вихідними потоками даних та зовнішніми об'єктами (джерелами і приймачами інформації), із якими взаємодіє ІС.

Розробка контекстних діаграм вирішує проблему суворого визначення функціональної структури ІС на початковій стадії її проектування, що особливо важливо для складних багатофункціональних систем, у розробці яких беруть участь різні організації і колективи розробників.

Після побудови контекстних діаграм отриману модель варто перевірити на повноту вихідних даних про об'єкти системи й ізольованість об'єктів (відсутність інформаційних зв'язків з іншими об'єктами).

Для кожної підсистеми, наявної на контекстних діаграмах, виконують її деталізацію за допомогою ДПД. Кожний процес на ДПД, у свою чергу, може бути деталізований за допомогою ДПД або мініспецифікації.

Під час деталізації мають виконуватися такі правила:

- правило балансування – означає, що під час деталізації підсистеми або процесу діаграма, яка деталізує як зовнішні джерела/приймачі даних, може мати тільки такі компоненти (підсистеми, процеси, зовнішні сутності, накопичувачі даних), із якими має інформаційний зв'язок підсистема або процес на батьківській діаграмі, що деталізується;
- правило нумерації – означає, що під час деталізації процесів має підтримуватися їхня ієрархічна нумерація. Наприклад, процеси, що деталізують процес із номером 12, одержують номери 12.1, 12.2, 12.3 і т. д.

Мініспецифікація (опис логіки процесу) має формулювати його основні функції так, щоб надалі фахівець, який виконує реалізацію проекту, зміг виконати їх або розробити відповідну програму.

Мініспецифікація є кінцевою вершиною ієрархії ДПД. Рішення щодо завершення деталізації процесу і використання мініспецифікації приймає аналітик, виходячи з таких критеріїв:

- наявність у процесі незначної кількості вхідних і вихідних потоків даних (2–3 потоки);
- можливість опису перетворення даних процесом у вигляді послідовного алгоритму;
- виконання процесом єдиної логічної функції перетворення вхідної інформації на вихідну;
- можливість опису логіки процесу за допомогою мініспецифікації невеликого обсягу (не більше 20–30 рядків).

Під час побудови ієрархії ДПД переходити до деталізації процесів необхідно тільки після визначення змісту всіх потоків і накопичувачів даних, що описуються за допомогою структур даних. Структури даних будуються з елементів даних і можуть містити альтернативи, умовні входження та ітерації. Умовне входження означає, що даного компонента може не бути у структурі. Альтернатива означає, що в структуру може входити один із перерахованих елементів; ітерація – входження будь-якої кількості елементів у зазначеному діапазоні. Для кожного елемента даних може вказуватися його тип (безперервні або дискретні дані). Для безперервних даних можна вказувати одиницю виміру (кг, см тощо), діапазон значень, точність подання і форму фізичного кодування. Для дискретних даних може вказуватися таблиця допустимих значень.

Після побудови кінцевої моделі системи її необхідно верифікувати (перевірити на повноту і узгодженість). У повній моделі усі її об'єкти (підсистеми, процеси, потоки даних) мають бути докладно описані і де-

талізовані. Виявлені недеталізовані об'єкти необхідно деталізувати, повернувшись на попередні кроки розробки. В узгодженій моделі для всіх потоків і накопичувачів даних має виконуватися правило збереження інформації: усі дані, що надходять, мають бути зчитані, а всі зчитані дані слід записати.

8.3.5. Проектування концептуальної схеми бази даних

Метою моделювання даних є забезпечення розробника ІС концептуальною схемою БД у формі однієї моделі або декількох локальних моделей, що відносно легко можуть бути відображені у будь-яку СКБД.

Перш ніж розпочати ER-моделювання, потрібно дати визначення деяких основних термінів. Наведений у табл. 8.1 перелік термінів не є вичерпним, достатній для виконання завдань практичного ER-моделювання. Деякі з наведених термінів стосуються ER-моделювання, а деякі є загальними термінами моделювання логічних структур даних.

Моделювання сутностей і зв'язків належить до моделювання логічних структур даних, яке базується на припущенні, що всі елементи ПО можна подати у вигляді ідеальних прототипів – абстрактних концепцій, або *сутностей*. Такі концептуальні сутності описуються в термінах їхніх характеристик, або *атрибутів* (*attribute*). Сутності пов'язані через дії, які вони виконують відносно одна до одної. Ці дії встановлюють *зв'язки* (relationship) між сутностями.

Таблиця 8.1. **Основні терміни моделювання сутностей і зв'язків**

Термін	Визначення
Сутність (*entity*)	Реальний об'єкт (фізична особа, підприємство, подія, предмет), дані про який зберігаються. Множину об'єктів однієї сутності також називають *класами сутностей* (*entity class*)
Екземпляр сутності (*entity instance*)	Конкретний представник класу сутностей. Наприклад, клієнт Кравченко є екземпляром сутностей КЛІЄНТ. Екземпляри сутностей називають також *втіленнями сутностей* (*entity occurrence*)
Підтип (*subtype*)	Клас сутностей, який є підмножиною більш великого типу сутностей, називають *супертипом*. Наприклад, клас сутностей ПОЖЕЖНИК може бути підтипом супертипу СЛУЖБОВЕЦЬ. Підтипи зазвичай успадковують атрибути і зв'язки супертипу, але можуть визначати і власні атрибути і зв'язки. Групу підтипів (наприклад, ПОЖЕЖНИК, ПОЛІЦЕЙСЬКИЙ, ДВІРНИК) називають *кластером* підтипів (*subtype cluster*)
Супертип (*supertype*)	Клас сутностей, що є надмножиною більш дрібних і вузьких класів сутностей, називають *підтипами*. Наприклад, клас сутностей АВТОМОБІЛЬ може бути супертипом сутностей ФОРД_АВТО, ДЖЕНЕРАЛ_МОТОРС_АВТО і КРАЙСЛЕР_АВТО. Супертипи і підтипи часто називають *S-типами* (*S-Type*)

Продовження табл. 8.1.

Термін	Визначення
Атрибут (*attribute*)	Характеристика сутності або зв'язку, призначена для кваліфікації, ідентифікації, класифікації, кількісної характеристики або стану. Атрибути докладно описують сутності. Наприклад, атрибут «Ідентифікаційний_код» може бути характеристикою класу сутностей СЛУЖБОВЦІ
Екземпляр атрибута (*attribute instance*)	Екземпляр атрибута визначається типом характеристики окремого екземпляра сутності і її значенням, яке називається *значенням атрибута*. Атрибути асоціюють з конкретними сутностями. Отже, екземпляр сутності повинен мати єдине визначене значення для асоційованого атрибута
Домен (*domain*)	Вказаний тип даних або діапазон значень, яких може набувати атрибут. Наприклад, домен *TDate* для атрибута Дата_Прийому вимагає, щоб усі значення цього атрибута являли собою правильно задані дати
Цілісність домену (*domain integrity*)	Правила, що визначають типи даних, які дозволяються доменом. Забезпечення цілісності домена може означати, наприклад, що значення, які зберігаються в атрибуті, до якого застосований домен *Date*, є правильними датами
Унікальний ідентифікатор сутності	Атрибут або сукупність атрибутів, призначена для унікальної ідентифікації кожного екземпляра сутності
Ідентифікатор сутності	Один з унікальних ідентифікаторів, обраний для унікальної ідентифікації кожного екземпляра сутності
Зв'язок (*relationship*)	Поіменована асоціація (з'єднання) двох сутностей, за якої кожний екземпляр однієї (батьківської) сутності асоційований із деякою кількістю екземплярів іншої (дочірньої) сутності. При цьому зміна стану батьківської сутності приводить до зміни стану дочірньої сутності. Кожний екземпляр сутності-нащадка є асоційованим з одним екземпляром сутності-батька. Отже, екземпляр сутності-нащадка може існувати тільки у разі існування екземпляра сутності-батька. Зв'язку надається ім'я, виражене граматичним зворотом дієслова, яке розміщується біля лінії зв'язку. Ім'я кожного зв'язку між двома даними сутностями має бути унікальним, але імена зв'язків у моделі не обов'язково мають бути унікальними. Ім'я зв'язку завжди формується з погляду батька, з'єднанням імені сутності-батька, імені зв'язку, ступеня зв'язку й імені сутності-нащадка
Цілісність зв'язків (*relational integrity*)	Правила, які забезпечують підтримку зв'язку між сутностями. Наприклад, цілісність зв'язків може перешкодити видаленню екземпляра сутності КЛІЄНТ, у якого є зв'язок з екземпляром сутності РАХУНОК
Потужність зв'язків (*connectivity*)	Відображення характеристики зв'язку між екземплярами зв'язаних сутностей. Наприклад, за визначенням потужності сутностей РАХУНОК і КЛІЄНТ можна зазначити, що кожний екземпляр сутності КЛІЄНТ може мати багато відповідних йому екземплярів класу сутностей РАХУНОК

За характером з'єднання розглядають чотири види зв'язків: «один до одного», «один до багатьох», «багато до одного», «багато до багатьох».

Розглядають такі типи зв'язків:

- *повний* – у зв'язку беруть участь усі екземпляри сутності;
- *необов'язковий* – у зв'язку беруть участь не всі екземпляри сутності;
- *обов'язковий* – екземпляри однієї сутності (залежної) можуть існувати тільки за наявності екземплярів іншої сутності (незалежної);
- *слабкий* – екземпляр дочірньої «слабкої» сутності можна ідентифікувати тільки за допомогою екземпляра батьківської «сильної» сутності, тобто ключ «сильної сутності» є частиною ключа «слабкої сутності»;
- «*супертип – підтип*» – загальні характеристики (атрибути) визначаються в батьківській сутності – супертипі, а дочірня сутність – підтип – успадковує атрибути супертипу;
- *асоціативний* – кожний екземпляр зв'язку (асоціативний об'єкт) може існувати тільки за умови існування окремо визначених екземплярів кожної із взаємозалежних сутностей. Асоціативний об'єкт – це об'єкт, що є одночасно сутністю і зв'язком. Асоціативний зв'язок – зв'язок між декількома «незалежними» сутностями і однією «залежною». Зв'язок між незалежними сутностями має атрибути, які визначаються у залежній сутності. Отже, залежна сутність визначається в термінах атрибутів зв'язку між іншими сутностями;
- *взаємовиключний* – зв'язок однієї сутності з декількома, за яких факт участі екземпляра в одному зв'язку веде до неможливості участі його в іншому;
- *рекурсивний* – зв'язок об'єкта із самим собою;
- *незсувний* – зв'язок, у якому екземпляр сутності не може бути зсунутий з одного екземпляра зв'язку в інший;
- *ідентифікуючий* – екземпляр сутності-нащадка однозначно визначається своїм зв'язком із сутністю-батьком;
- *неідентифікучий* – екземпляр сутності-нащадка не визначається однозначно своїм зв'язком із сутністю-батьком.

ER-моделювання проводиться з метою отримання зручного візуального представлення об'єктів і зв'язків між ними як концептуальної моделі БД. На базі цієї моделі проводиться наступне формування фізичної БД засобами СКБД. У наступних розділах розглянемо найпоширеніші підходи до процесу ER-моделювання.

8.3.5.1. ER-моделювання за методикою Чена

Найпоширенішим засобом семантичного моделювання даних є діаграми *сутність – зв'язок* (Entity Relations Diagram – ERD). За їх допомогою визначають важливі для ПО об'єкти (сутності), їх властивості (атрибути) та взаємо-

відношення (відношення між собою, тобто зв'язки). ERD можна безпосередньо використовувати для проектування реляційних БД.

Пітер Чен (Peter Chen) 1976 року опублікував специфікацію щодо підходу до реляційних структур як до набору зв'язків між сутностями. Ця праця та праці інших теоретиків, зокрема Хаммера (Hammer) та Мак-Леода (McLeod), які створили семантичну модель даних, стали основою для виникнення ERD, тобто ER-діаграм (діаграм *сутність – звязок*), які в наочній формі представляють зв'язки між сутностями і сьогодні є фундаментом моделювання логічних структур даних.

Існують два основні типи ER-діаграми, запропонованих П. Ченом.

Перший тип використовує *стандартну нотацію Чена*. У діаграмі сутність зображено прямокутником, у середині якого наведено ім'я сутності. Зв'язок показано ромбом, усередині якого – ім'я зв'язку. Сутність з'єднано із зв'язком за допомогою дуги, над якою вказано потужність зв'язку. Діаграми такого типу дуже спеціалізовані – кожна зазвичай представляє один зв'язок між двома сутностями. Приклад простої ER-моделі, що використовує стандартну нотацію Чена, показано на рис.8.5.

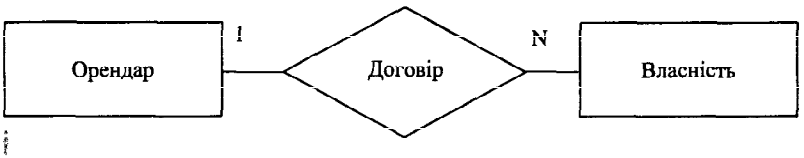


Рис. 8.5. ER-діаграма Чена стандартного типу

Другий тип діаграми називають *деталізованою ER-діаграмою*. Під час проектування громіздкої БД застосування методики Чена призводить до створення значної кількості діаграм. У разі застосування деталізованих ER-моделей цього, як правило, можна уникнути, тому що всі зв'язки окремої сутності наводяться на одній діаграмі. Указні діаграми ґрунтуються на сутностях, а не на зв'язках. Особливістю деталізованих діаграм також є інформації про атрибути (характеристики) сутностей. Якщо не вдаватися до подробиць, то сутності, які визначають за допомогою ER-діаграм, відповідають відношенням (таблицям) реляційний моделі БД та її наступної реалізації як фізичного сховища структурованої інформації. Саме завдяки цьому більшість популярних засобів створення подібних діаграм використовують не тільки елементи моделювання сутностей і зв'язків, але й і елементи реляційної моделі та фізичного проектування.

Приклад деталізованої ER-діаграми показано на рис. 8.6.

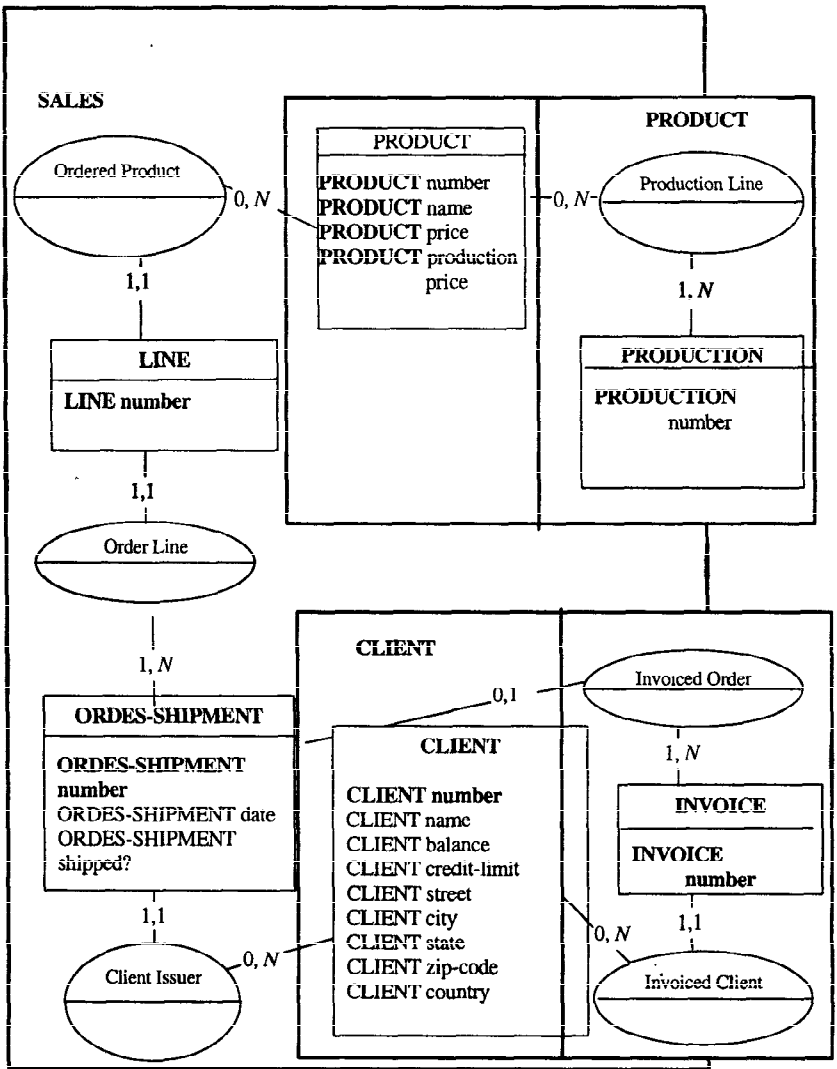


Рис. 8.6. Деталізована ER-діаграма Чена

8.3.5.2. ER- моделювання за методикою Баркера

Нотація ERD набула подальшого розвитку в роботах Баркера. Метод Баркера розглянемо на прикладі моделювання діяльності компанії з торгівлі автомашинами. Нижче наведений витяг з обов'язків персоналу компанії.

Головний менеджер: один з основних обов'язків – утримання автомобільного майна. Він повинен знати, скільки сплачено за машини і які є накладні витрати. Маючи цю інформацію, він може встановити нижню ціну, за якою можна продати даний зразок. Крім того, він відповідає за роботу продавців і йому слід знати, хто що продає і скільки машин продав кожний із них.

Продавець: повинен знати встановлену ціну і нижню ціну, за якою можна здійснити операцію продажу. Крім того, йому потрібна основна інформація про машини: рік випуску, марка, модель тощо.

Адміністратор: його діяльність зводиться до укладання контрактів, для чого потрібна інформація про покупця, автомашини і продавців, оскільки саме завдяки контрактам продавці отримують продаж.

Перший крок моделювання – одержання інформації під час бесіди з персоналом і виділення сутностей. За методом Баркера під час побудови ER-діаграми дотримуються наведених нижче узгоджень.

Графічне зображення сутності – прямокутник, в якому наводиться ім'я сутності (рис. 8.7).

Рис. 8.7. Графічне зображення сутності

Кожна сутність повинна мати унікальний ідентифікатор. Кожний екземпляр сутності має однозначно ідентифікуватися і відрізнятися від усіх інших екземплярів цього типу сутності. Будь-яка сутність має містити:

- унікальне ім'я. До імені слід завжди застосовувати одну і та саму інтерпретацію. Одна і та сама інтерпретація не може застосовуватися до різних імен;
- один або декілька атрибутів, які або належать сутності, або успадковуються через зв'язок;
- один або декілька атрибутів, що однозначно ідентифікують кожний екземпляр сутності;
- будь-яку кількість зв'язків з іншими сутностями моделі.

З наведеного вище прикладу ПО видно, що сутності, які можуть бути ідентифіковані з головним менеджером – це автомашини і продавці. Для продавця важливі автомашини і пов'язані з їхнім продажем дані, для адміністратора – покупці, автомашини, продавці та контракти. Виходячи з цього, виділяють 4 сутності (автомашина, продавець, покупець, контракт), які показано на рис. 8.8.

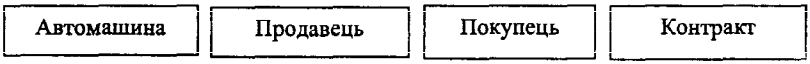


Рис. 8.8. Сутності ПО

Наступним кроком моделювання є ідентифікація зв'язків.

Зв'язок «продавець – контракт» може бути вираженим у такий спосіб:

- продавець може одержати винагороду за один або більше контрактів;
- контракт має бути ініційований лише одним продавцем.

Ступінь зв'язку й обов'язковість графічно показано на рис. 8.9.

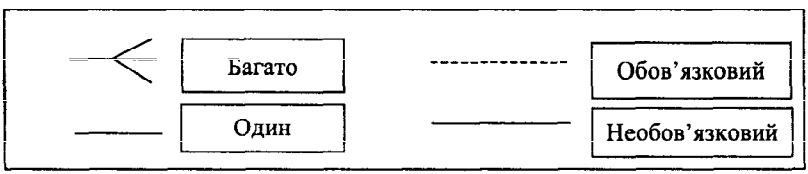


Рис. 8.9. Види зв'язків

Отже, опис зв'язку «продавець – контракт» матиме такий вигляд (рис. 8.10):

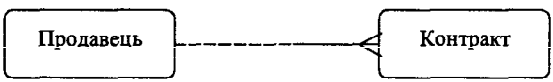


Рис. 8.10. Зв'язок «продавець – контракт»

Після опису зв'язків інших сутностей одержимо таку схему (рис. 8.11).

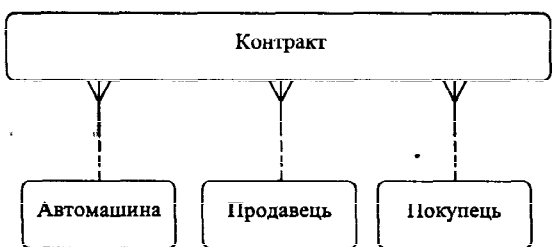


Рис. 8.11. Зразок схеми ERD

Останнім кроком моделювання є ідентифікація атрибутів.

Атрибут представляє тип характеристик або властивостей, асоційованих із безліччю реальних або абстрактних об'єктів (людей, місць, подій, станів, ідей, пар предметів тощо). Екземпляр атрибута – це визначена ха-

рактеристика окремого елемента. Атрибут може бути або обов'язковим, або необов'язковим (див. рис. 8.10). Обов'язковість означає, що атрибут не може набувати невизначених значень (*null values*). Атрибут може бути або описовим (тобто звичайним дескриптором сутності), або входити до складу унікального ідентифікатора (первинного ключа).

У разі повної ідентифікації кожний екземпляр даного типу сутності цілком ідентифікується своїми власними ключовими атрибутами, інакше в його ідентифікації беруть участь також атрибути іншої сутності-батька (рис. 8.12, 8.13).

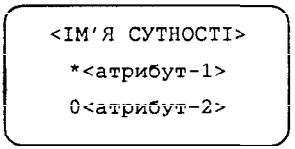
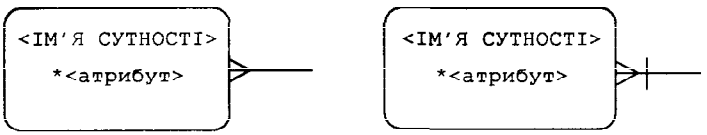


Рис. 8.12. Зображення атрибутів:
* – обов'язковий атрибут; 0 – необов'язковий атрибут



а *б*

Рис. 8.13. Типи ідентифікацій:
а – повна ідентифікація; *б* – ідентифікація через іншу сутність

Кожний атрибут ідентифікується унікальним ім'ям, що виражається граматичним зворотом іменника, та описує характеристику атрибута. Атрибути зображують у вигляді списку імен у середині блоку асоційованої сутності, причому кожний атрибут займає окремий рядок. Атрибути, що визначають первинний ключ, розміщують зверху списку і виділяють знаком «#».

Кожна сутність повинна мати хоча б один можливий ключ. Можливий ключ сутності – це один або декілька атрибутів, значення яких однозначно визначають кожний екземпляр сутності. У разі існування декількох можливих ключів, один із них позначається як первинний ключ, а інші – як альтернативні.

Ураховуючи наявну інформацію, доповнимо побудовану раніше схему (рис. 8.14). Крім перерахованих основних конструкцій, модель даних може містити ряд додаткових. Підтипи і супертипи: одна сутність є узагальнюючим поняттям для групи подібних сутностей (рис. 8.15). Взаємовиключні зв'язки: кожний екземпляр сутності бере участь тільки в одному зв'язку з групи зв'язків (рис. 8.16).

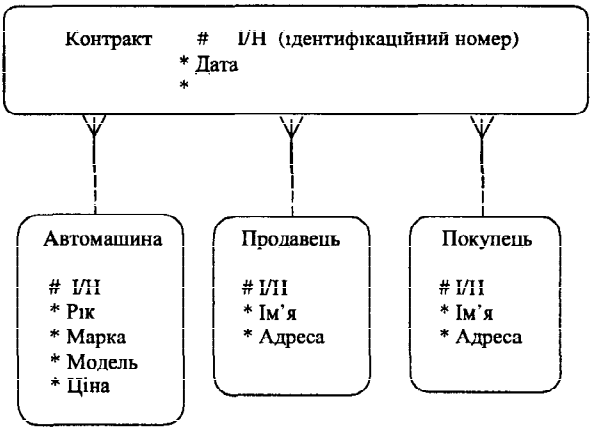


Рис. 8.14 Приклад схеми з атрибутами

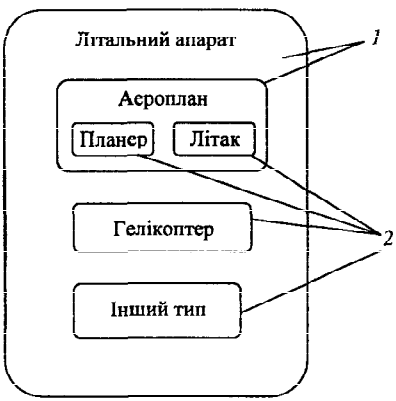


Рис. 8.15. Підтипи і супертипи:
1 – супертипи; *2* – підтипи

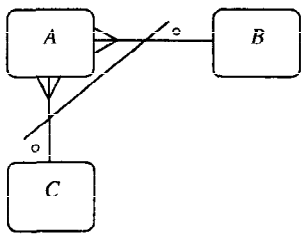


Рис. 8.16. Взаємовиключні зв'язки

Рекурсивний зв'язок: сутність може бути зв'язана сама із собою (рис. 8.17).

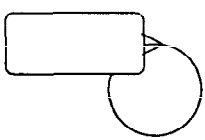


Рис. 8.17. Рекурсивний зв'язок

Незсувні (*non-transferrable*) зв'язки: екземпляр сутності не може бути перенесений з одного екземпляра зв'язку в інший (рис. 8. 18).

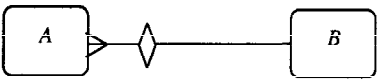


Рис. 8.18. Незсувний зв'язок

8.3.5.3. Методологія IDEF1-IDEF1X

Методологію IDEF1 (що також базується на підході П. Чена) розроблено Т. Ремом (T. Ramey). Вона дозволяє побудувати модель даних, еквівалентну реляційній моделі в третій нормальній формі. Після вдосконалення методології IDEF1 створено її нову версію – методологію IDEF1X. IDEF1X розроблено з урахуванням таких вимог, як простота вивчення і можливість автоматизації. IDEF1X-діаграми використовують разом з поширеними CASE-засобами (зокрема, ERwin, Design/IDEF).

Сутність у методології IDEF1X називають *незалежною від ідентифікаторів* (або просто *незалежною*), якщо кожний екземпляр сутності може бути однозначно ідентифікований без визначення його відношень з іншими сутностями. Сутність називають *залежною від ідентифікаторів* (або просто *залежною*), якщо однозначна ідентифікація екземпляра сутності залежить від його відношення до іншої сутності (рис. 8.19).

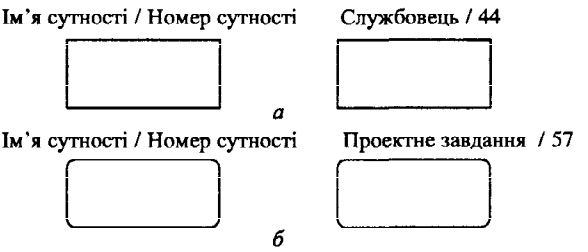


Рис. 8.19. Сутності: *а* – залежна;
б – незалежна від ідентифікатора

Кожній сутності присвоюють унікальне ім'я і номер, які відокремлюють навскісною рискою «/» і пишуть над блоком. Атрибути зображують у вигляді списку імен усередині прямокутника зображення сутності.

Атрибути, що визначають первинний ключ, розміщують зверху у списку і відокремлюють від інших атрибутів горизонтальною лінією (рис. 8.20).

Сутності, які беруть участь у зв'язках як сутності-нащадки, мають також і зовнішні ключі (*Foreign Key*), тобто атрибути, які є первинними ключами в сутностях-батьках. Для своєї сутності вони можуть бути первинним ключем, частиною первинного ключа або неключовим атрибутом. Зовнішній ключ зображують за допомогою запису імен атрибутів, після яких у дужках записано літери *FK* (рис. 8.21).

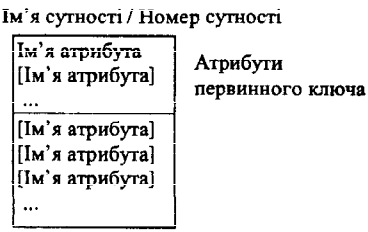


Рис. 8.20. Атрибути і первинні ключі

У IDEF1X зв'язок визначається з указанням ступеня або потужності (кількості екземплярів сутності-нащадка, що може існувати для кожного екземпляра сутності-батька). Розглядаються такі типи потужності зв'язків:

- кожний екземпляр сутності-батька може мати нуль, один або більше зв'язаних із ним екземплярів сутності-нащадка (потужність типу *N*);
- кожний екземпляр сутності-батька повинен мати не менше одного зв'язаного з ним екземпляра сутності-нащадка (потужність типу *P*);
- кожний екземпляр сутності-батька повинен мати не більше одного зв'язаного з ним екземпляра сутності-нащадка (потужність типу *Z*);
- кожний екземпляр сутності-батька зв'язаний із деякою фіксованою кількістю екземплярів сутності-нащадка (потужність фіксованого типу).

Якщо екземпляр сутності-нащадка однозначно визначається своїм зв'язком із сутністю-батьком (тобто в сутності-нащадку атрибут – зовнішній ключ, який входить до складу первинного ключа), то зв'язок називають *ідентифікованим*, у противному разі – *неідентифікованим*.

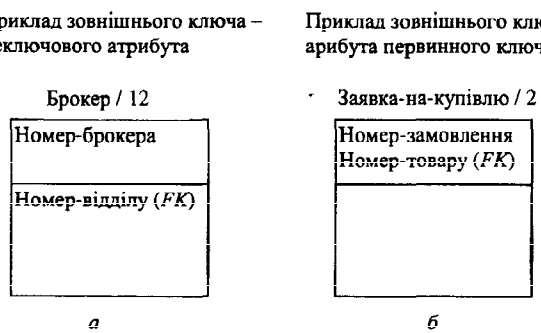


Рис. 8.21. Приклади зовнішніх ключів:
а – неключовий атрибут; *б* – частина первинного ключа

Рис. 8.22. Потужності зв'язків

Зв'язок зображують лінією, проведеною між сутністю-батьком і сутністю-нащадком із крапкою на кінці лінії в сутності-нащадка. Потужність зв'язку позначається, як показано на рис. 8.22 (потужність за замовчуванням – *N*).

Ідентифікований зв'язок між сутністю-нащадком і сутністю-батьком зображують суцільною лінією (рис. 8.23). Сутність-нащадок у ідентифікованому зв'язку є залежною від ідентифікатора сутністю. Сутність-батько в ідентифікованому зв'язку може бути як незалежною, так і залежною від ідентифікатора сутністю (це визначається її зв'язками з іншими сутностями).

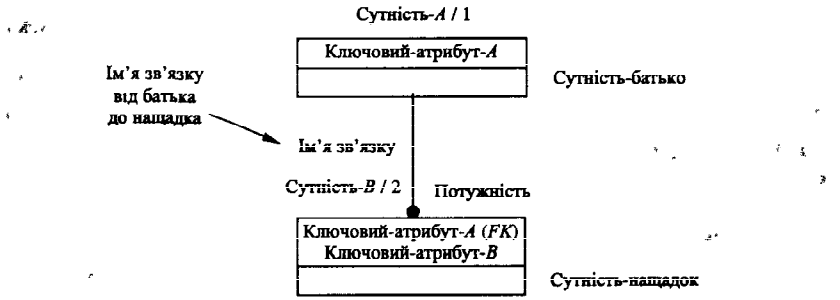


Рис. 8.23. Ідентифікований зв'язок

Неідентифікований зв'язок показано пунктирною лінією (рис. 8.24). Сутність-нащадок у неідентифікованому зв'язку буде незалежною від ідентифікатора, якщо вона не є також нащадком у якомусь іншому ідентифікованому зв'язку.

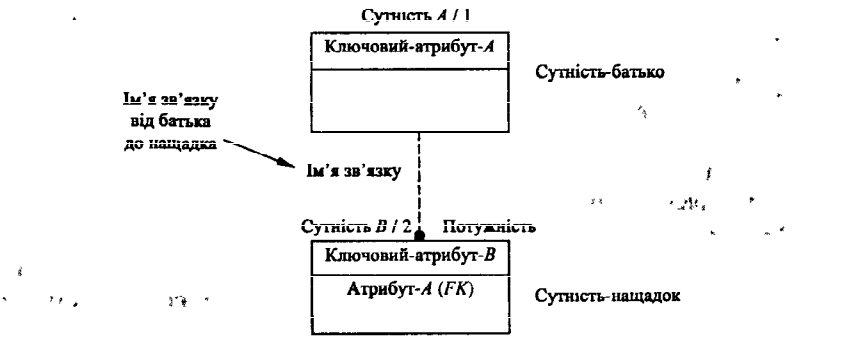


Рис. 8.24. Неідентифікований зв'язок

8.3.5.4. Семантична модель Vantage Team Builder

У семантичній моделі Vantage Team Builder використовується один із варіантів нотації П. Чена. На ER-діаграмах сутність позначено прямокутником, в який вписано ім'я, а зв'язок – ромбом (рис. 8.25). Числа над лініями вказують ступінь зв'язку.

Рис. 8.25. Позначення сутностей і зв'язків

Зв'язки є багатонапрямленими і можуть мати атрибути (за винятком ключових). Виділяють два види зв'язків:

- необов'язковий (*optional*);
- слабкий (*weak*).

У необов'язковому зв'язку (рис. 8.26) можуть брати участь не всі екземпляри сутності.

Рис. 8.26. Необов'язковий зв'язок

На відміну від необов'язкового зв'язку у повному (*total*) беруть участь усі екземпляри хоча б однієї із сутностей. Це означає, що екземпляри такого зв'язку існують тільки за умови існування екземплярів іншої сутності. Повний зв'язок може мати один із 4-х видів: обов'язковий зв'язок, слабкий, зв'язок «супертип – підтип» та асоціативний.

Обов'язковий (*mandatory*) зв'язок описує зв'язок між *незалежною* і *залежною* сутностями. Усі екземпляри залежної (обов'язкової) сутності можуть існувати тільки за наявності екземплярів незалежної (необов'язкової) сутності, тобто екземпляр обов'язкової сутності може існувати тільки за умови існування визначеного екземпляра необов'язкової сутності.

У прикладі (рис. 8.27) враховано, що кожен автомобіль має принаймні одного водія, але не кожен службовець керує машиною.

Рис. 8.27. Обов'язковий зв'язок

У слабкому зв'язку існування однієї із сутностей, яка належить деякій множині (*слабкій*), залежить від існування визначеної сутності, яка належить іншій множині (*сильній*), тобто екземпляр слабкої сутності може бути ідентифікований тільки за допомогою екземпляра сильної сутності. Ключ сильної сутності є частиною складового ключа слабкої сутності.

Слабкий зв'язок завжди є бінарним і допускає обов'язковий зв'язок для слабкої сутності. Сутність може бути *слабкою* в одному зв'язку і *сильною* в іншому, але не може бути слабкою більше, ніж в одному зв'язку. Слабкий зв'язок може не мати атрибутів.

Ключ (номер) рядка в документі може не бути унікальним і має бути доповнений ключем документа (рис. 8.28).

Рис. 8.28. Слабкий зв'язок

Зв'язок «супертип – підтип» зображено на рис. 8.29. Загальні характеристики (атрибути) типу визначаються у сутності-супертипі. Сутність-підтип успадковує всі характеристики супертипу. *Екземпляр підтипу існує тільки за умови існування визначеного екземпляра* супертипу. Підтип не може мати ключа (він імпортує ключ із супертипу). Сутність, що є супертипом в одному зв'язку, може бути підтипом в іншому. Зв'язок супертипу не може мати атрибутів.

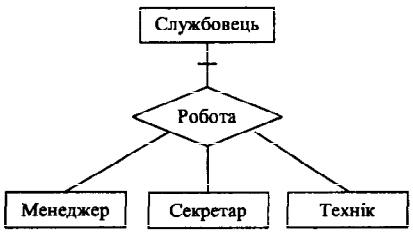


Рис. 8.29. Зв'язок «супертип – підтип»

В асоціативному зв'язку кожний екземпляр (асоціативний об'єкт) може існувати тільки за умови існування окремо визначених екземплярів кожної із взаємозалежних сутностей. Асоціативний об'єкт – це об'єкт, що є одночасно сутністю і зв'язком. Асоціативний зв'язок – це зв'язок між декількома незалежними сутностями і однією залежною. Зв'язок між незалежними сутностями має атрибути, які визначаються у залежній сутності. Отже, залежна сутність визначається в термінах атрибутів зв'язку між іншими сутностями.

У прикладі на рис. 8.30 літак здійснює посадку на злітну смугу у заданий час за відомої швидкості і напрямку вітру. Оскільки ці характеристики застосовуються тільки до конкретної посадки, вони є атрибутами посадки, а не літака або злітної смуги. Зв'язок «пілот – посадка» сильніший, ніж зв'язки «пілот – літак» і «пілот – злітна смуга».

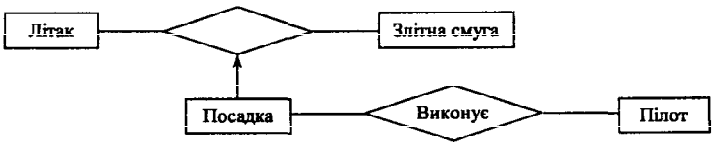


Рис. 8.30. Асоціативний зв'язок

Первинний ключ кожного типу сутності позначається зірочкою «*».

ER-діаграма має задовольняти такі правила:

- кожна сутність, кожний атрибут і кожний зв'язок повинні мати ім'я (але зв'язок супертипу або асоціативний зв'язок може не мати імені);
- ім'я сутності має бути унікальним у рамках моделі даних;
- ім'я атрибута має бути унікальним у межах сутності;
- ім'я зв'язку має бути унікальним, якщо для нього генерується таблиця БД;
- кожний атрибут повинен мати визначення типу даних;
- сутність у необов'язковому зв'язку повинна мати ключовий атрибут. Те саме стосується сильної сутності в слабкому зв'язку, супертипу у зв'язку «супертип – підтип» і необов'язкової сутності в обов'язковому (повному) зв'язку;
- підтип у зв'язку «супертип – підтип» не може мати ключового атрибута;
- в асоціативному або слабкому зв'язку може бути тільки одна асоціативна (слабка) сутність;
- зв'язок не може бути одночасно обов'язковим зв'язком, зв'язком «супертип – підтип» або асоціативним.

8.3.5.5. Семантична модель ORACLE

На використанні різновидів ER-моделі базується більшість сучасних підходів до проектування БД (переважно, реляційних). Один з різновидів ER-моделей застосовується у фірмі ORACLE. Розглянемо структурну частину цієї моделі.

Основними поняттями ER-моделі є тип сутності, тип зв'язку й атрибут. *Сутність* – це реальний або уявлений об'єкт, інформація про який має зберігатися і бути доступною. У діаграмах ER-моделі (ER-діаграми) сутність подається у вигляді прямокутника, що містить ім'я сутності. При цьому ім'я сутності – це ім'я типу, а не деякого конкретного екземпляра цього типу.

Кількість сутностей, наведених у ERD, позначено прямокутником та відповідає класам сутностей. Належність сутності до класу визначається предикатом. Сутність може входити до багатьох множин, наприклад *Лікар* може бути *Пацієнтом*.

Зв'язки в ERD позначають ромбами. Сутності, що беруть участь у зв'язку, з'єднано за допомогою дуг. На відміну від мережної та ієрархічної моделей, до яких входять лише функціональні зв'язки, до ER-моделі входять *n*-арні, взаємно однозначні, функціональні або типу «декілька до декількох». Допускаються також рекурсивні зв'язки.

Тип відображення позначається біля кожного об'єкта і залежить від максимальної кількості екземплярів сутності, що може брати участь у зв'язку.

Клас сутності може брати участь у зв'язку декілька разів у різних ролях. Наприклад у зв'язку *Управління* сутність *Персонал* бере участь перший раз у ролі *Керівника*, а другий – у ролі *Підлеглого*. У цьому разі роль сутності наводиться над дугою, яка поєднує сутність з указаним типом зв'язку.

Між двома множинами сутностей може бути наведено кілька типів зв'язків. Наприклад, між сутностями *Лікар* і *Пацієнт*. Кожен пацієнт може мати тільки одного лікаря та кількох консультантів. Кожен лікар може лікувати та консультувати кількох пацієнтів.

В ER-моделі можливі *n*-арні зв'язки, як, наприклад, зв'язок *Лікар-Пацієнт-Аналіз* Одному пацієнту може бути призначено одночасно декілька аналізів декількома лікарями

Участь множини сутностей у зв'язку може бути або жорсткою, або факультативною Під час вилучення екземпляра зв'язку вилучаються всі жорстко зв'язані сутності. Так, у разі вилучення зв'язку *Призначені аналізи* всі аналізи вилучаються, але лікарі та пацієнти лишаються. Відповідну ERD-діаграма даної ПО зображено на рис. 8.31.

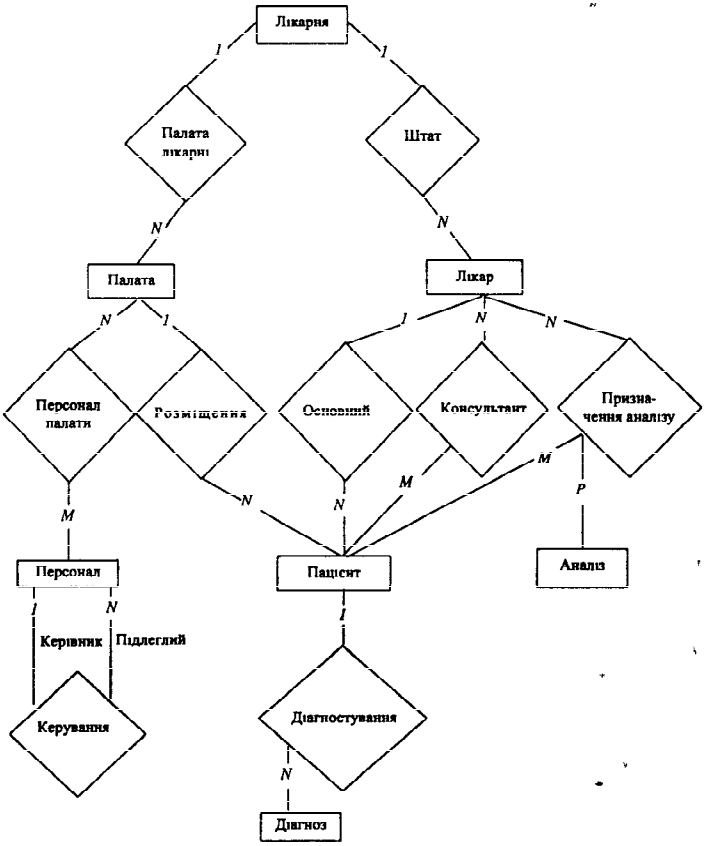


Рис. 8.31. Приклад ER-діаграми

Домен в *ER*-моделі називають *множиною значень*. Множина значень може відповідати множині сутностей або множині зв'язків. Множина значень на ER-моделі може позначатися еліпсом.

Атрибутом називають відображення множини сутностей або звязків на множину значень. Атрибут позначають дугою, яку напрямлено від сутності або зв'язку до значення.

Унікальним ідентифікатором сутності є атрибут, комбінація атрибутів, комбінація зв'язків або комбінація зв'язків і атрибутів, яка відрізняє будь-який екземпляр сутності від інших сутностей того ж типу. Приклад зображення атрибутів на ER-діаграмі наведено на рис. 8.32.

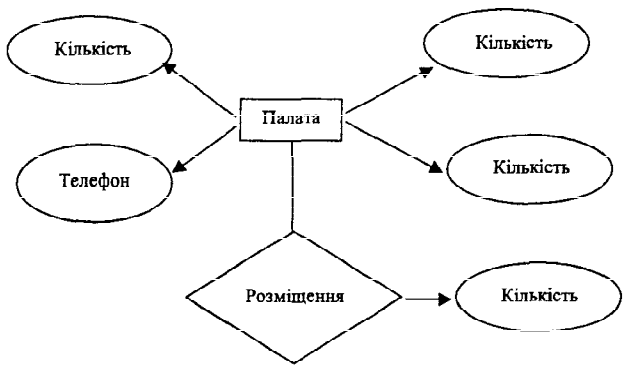


Рис. 8.32. Представлення атрибутів на ER-діаграмі

Аналогічно реляційним схемам БД, в ER-схемах уводять поняття нормальних форм, причому їх розуміння відповідає розумінню реляційних нормальних форм. Відзначимо, що формулювання визначень нормальних форм ER-схем роблять більш зрозумілим поняття нормалізації реляційних схем.

У першій нормальній формі ER-схеми вилучають атрибути або групи повторюваних атрибутів, тобто виявляють неявні сутності, що «замаскувалися» під атрибути.

У другій нормальній формі вилучають атрибути, які залежать тільки від частини унікального ідентифікатора. Ця частина унікального ідентифікатора визначає окрему сутність.

У третій нормальній формі вилучають атрибути, які залежать від атрибутів, що не входять в унікальний ідентифікатор. Ці атрибути є основою окремої сутності.

Було розглянуто тільки основні і найбільш очевидні поняття розглядуваної ER-моделі даних. Більш складними елементами моделі є такі:

- підтипи і супертипи сутності. У мовах програмування з розвиненими типовими системами (наприклад, у мовах об'єктно-зорієнтованого

програмування) вводиться можливість успадкування типу сутності, виходячи з одного або декількох супертипів;

- зв'язки «багато до багатьох». Інколи необхідно зв'язувати сутність так, що з обох кінців зв'язку можуть знаходитися декілька екземплярів сутності (наприклад, усі члени кооперативу спільно володіють майном кооперативу). Для цього вводиться різновид зв'язку «багато до багатьох»;
- ступені зв'язку, що уточнюються. Використовують, якщо потрібно визначити можливу кількість екземплярів сутності, що беруть участь у даному зв'язку (наприклад, що дозволяється брати участь не більше ніж у трьох проектах одночасно). Для вираження цього семантичного обмеження можна вказувати на кінці зв'язку його максимальний або обов'язковий ступінь;
- каскадні вилучення екземплярів сутності. Деякі зв'язки є настільки сильними (як правило, для зв'язку «один до багатьох»), що під час вилучення опорного екземпляра сутності (відповідного кінця зв'язку «один») потрібно вилучити і всі екземпляри сутності, які відповідають кінцю зв'язку *декілька*. Відповідну вимогу «каскадного вилучення» можна сформулювати під час визначення сутності.

Ці й інші більш складні елементи моделі даних «сутність – зв'язок» роблять її значно потужнішою, але водночас дещо ускладнюють використання.

8.3.6. Приклад побудови ER-моделі

Опис ПО. У цьому прикладі використовують методологію Yourdon, реалізовану в CASE-засобі Vantage Team Builder.

Як ПО використовують опис роботи відеобібліотеки, яка одержує запити від клієнтів на фільми, котрі вони повертають. Запити розглядає адміністрація відеобібліотеки, використовуючи інформацію про клієнтів і фільми. При цьому перевіряють і поновлюють список орендованих фільмів, а також записи про членство у бібліотеці. Якщо необхідний фільм наявний у бібліотеці, адміністрація інформує клієнта про орендну плату. Але, якщо клієнт невчасно повернув фільми, йому не дозволяється брати нові. Коли стрічку повертають, адміністрація розраховує орендну плату (плюс пеня за несвоєчасне повернення). Якщо клієнт не є членом бібліотеки, він не має права на оренду.

Відеобібліотека одержує нові фільми від своїх постачальників. Коли нові фільми надходять у бібліотеку, це фіксується. Інформація про членство у бібліотеці зберігається окремо від записів про оренду фільмів.

Адміністрація бібліотеки регулярно складає звіти за певний період часу про членів бібліотеки, нові надходження, оренду фільмів, їх постачальників.

Організація проекту. Увесь проект поділено на 4 фази: аналіз, глобальне проектування (проектування архітектури системи), детальне проектування і реалізація (програмування).

На фазі аналізу будується модель середовища, тобто проводиться:

- аналіз поведінки системи (визначення призначення ІС, побудова початкової контекстної діаграми потоків даних (DFD) і формування матриці списку подій (ELM – Event List Matrix), побудова контекстних діаграм);
- аналіз даних (визначення складу потоків даних і побудова діаграм структур даних (DSD – Data Structure Diagrames), конструювання глобальної моделі даних у вигляді ER-діаграми).

У цьому разі призначення ІС формулюється в такий спосіб: ведення БД про членів бібліотеки, постачальників, фільми та оренду. При цьому керівництво бібліотеки повинно мати можливість одержувати різні види звітів для виконання своїх завдань.

Перед побудовою контекстної DFD необхідно проаналізувати зовнішні події (зовнішні об'єкти), що впливають на функціонування бібліотеки. Ці об'єкти взаємодіють із ІС шляхом інформаційного обміну.

З опису ПО випливає, що в процесі роботи бібліотеки беруть участь такі групи людей: клієнти, постачальники і керівництво. Ці групи є зовнішніми об'єктами. Вони не тільки взаємодіють із системою, а також визначають її межі і зображуються на початковій контекстній DFD як термінатори (зовнішні сутності).

Початкову контекстну діаграму зображено на рис. 8.33. Зовнішні сутності позначено звичайними прямокутниками, а процеси – колом.

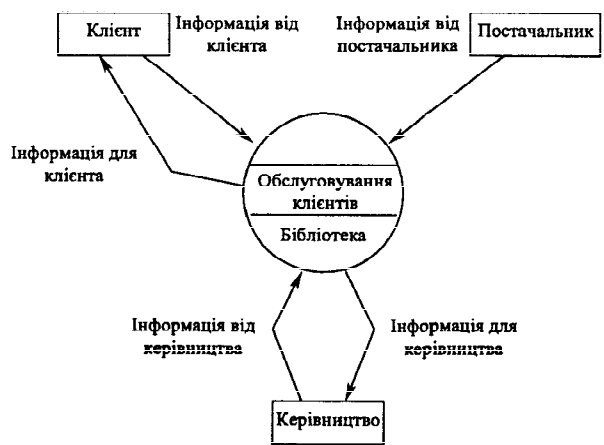


Рис. 8.33. Початкова контекстна діаграма

Список подій будується у вигляді матриці (ELM) і описує різні дії зовнішніх сутностей та реакцію ІС на ці дії. Вони являють собою зовнішні події, що впливають на бібліотеку.

Типи подій:

NC – нормальне керування;

ND – нормальні дані;

NCD – нормальне керування/дані;

TC – тимчасове керування;

TD – тимчасові дані;

TCD – тимчасове керування/дані.

Усі дії позначаються як нормальні дані. Ці дані є подіями, які ІС сприймає безпосередньо, наприклад, зміна адреси клієнта, що має бути відразу зареєстровано. Вони з'являються у DFD як вміст потоків даних.

Список подій наведено у табл. 8.2.

Таблиця 8.2. **Список подій**

Опис	Тип	Реакція
1. Клієнт бажає стати членом бібліотеки	*ND*	Реєстрація клієнта як члена бібліотеки
2. Клієнт повідомляє про зміну адреси	*ND*	Реєстрація нової адреси клієнта
3. Клієнт замовляє фільм	*ND*	Розгляд запиту
4. Клієнт повертає фільм	*ND*	Реєстрація повернення
5. Керівництво надає повноваження новому постачальнику	*ND*	Реєстрація постачальника
6. Постачальник повідомляє про зміну адреси	*ND*	Реєстрація нової адреси постачальника
7. Постачальник направляє фільм у бібліотеку	*ND*	Одержання нового фільму
8. Керівництво запитує новий звіт	*ND*	Формування необхідного звіту для керівництва

Для завершення аналізу функціонального аспекту поведінки системи будується повна контекстна діаграма, яка містить діаграму нульового рівня. При цьому процес *Бібліотека* поділяється на 4 процеси, які відображають основні види адміністративної діяльності бібліотеки. Існуючі «абстрактні» потоки даних між термінаторами і процесами трансформуються у потоки, які представляють обмін даними на більш конкретному рівні. Список подій показує, які потоки існують на цьому рівні: кожна подія зі списку має формувати деякий потік (подія формує вхідний потік, реакція – вихідний потік).

Один «абстрактний» потік може бути поділений на більше ніж один «конкретний» потік. Розглянемо розподіл потоків (табл. 8.3).

Таблиця 8.3. **Розподіл потоків**

Потоки на діаграмі верхнього рівня	Потоки на діаграмі нульового рівня
Інформація від клієнта	Дані про клієнта, Запит про оренду
Інформація для клієнта	Членська картка, Відповідь на запит про оренду
Інформація від керівництва	Запит звіту про нових членів, Новий постачальник, Запит звіту про постачальників, Запит звіту про оренду, Запит звіту про фільми
Інформація для керівництва	Звіт про нових членів, Звіт про постачальників, Звіт про оренду, Звіт про фільми
Інформація від постачальника	Дані про постачальника, Нові фільми

На наведеній на рис. 8.34 повній контекстній діаграмі накопичувач даних *бібліотека* є глобальним або абстрактним поданням сховища даних.

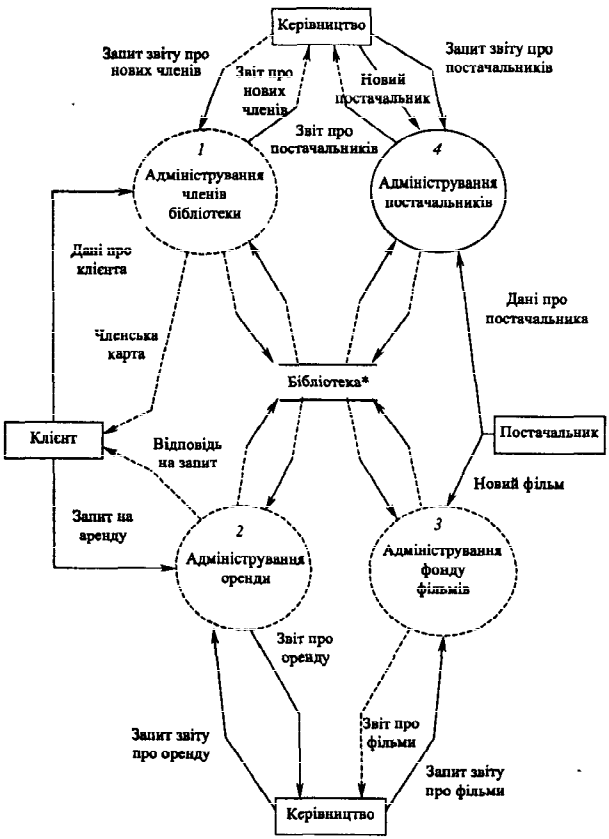


Рис. 8.34. Повна контекстна діаграма

Аналіз функціонального аспекту поведінки системи дає уявлення про обмін і перетворення даних у системі. Взаємозв'язок між «абстрактними» і «конкретними» потоками даних на діаграмі нульового рівня виражається у діаграмах структур даних (рис. 8.35).

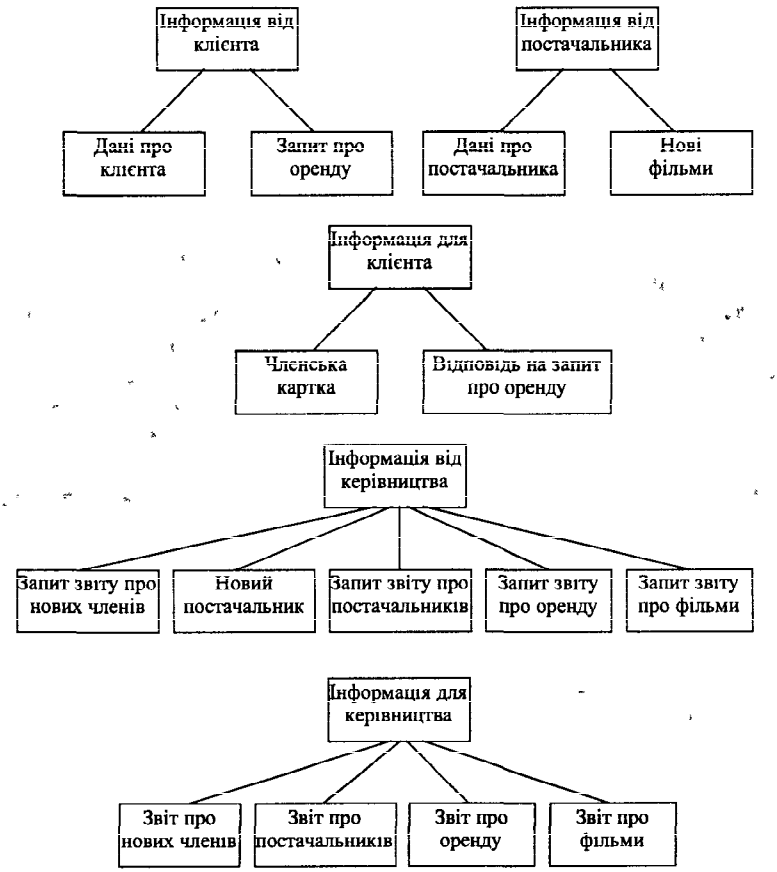


Рис. 8.35. Діаграма структур даних

На фазі аналізу будується глобальна модель даних, яка подається у вигляді діаграми «сутність – зв'язок» (рис. 8.36).

Між різними типами діаграм існують такі взаємозв'язки:

ELM – DFD: події – вхідні потоки, реакції – вихідні потоки;

DFD – DSD: потоки даних – структури даних верхнього рівня;

DFD – ERD: накопичувачі даних – ER-діаграми;

DSD – ERD: структури даних нижнього рівня – атрибути сутностей.

На етапі проектування архітектури будується предметна модель. Процес побудови предметної моделі містить у собі:

- детальний опис функціонування системи;
- подальший аналіз використовуваних даних і побудова концептуальної моделі даних для наступного проектування БД;
- визначення структури інтерфейсу користувача, специфікації форм і порядку їх появи;
- уточнення діаграм потоків даних і списку подій, виділення серед процесів нижнього рівня інтерактивних і неінтерактивних, визначення для них мініспецифікацій.

Результати проектування архітектури такі:

- модель процесів (діаграми архітектури системи (SAD – System Architecture Diagram)) та мініспецифікації структурованою мовою);
- модель даних (ERD та її підсхеми);
- модель інтерфейсу користувача (класифікація процесів на інтерактивні та неінтерактивні функції, діаграма послідовності форм (FSD – Form Sequence Diagram). Модель показує, які форми з'являються у додатку і в якому порядку. На FSD фіксується набір і структура викликів екранних форм. Діаграми FSD утворюють ієрархію, на вершині якої знаходиться головна форма додатка, що реалізує підсистему. На другому рівні знаходяться форми, які реалізують процеси нижнього рівня функціональної структури, зафіксованої на діаграмах SAD.

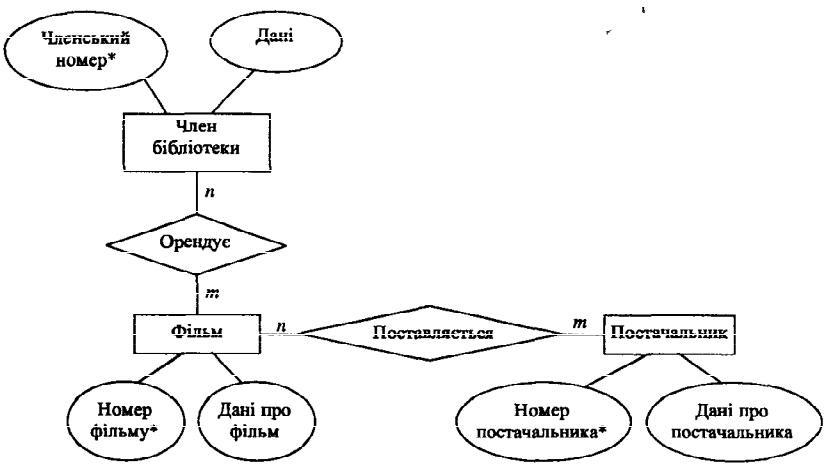


Рис. 8.36. Діаграма «сутність – зв'язок»

На фазі детального проектування будують модульну модель. Під *модульною моделлю* розуміють реальну модель прикладної системи, що проектується. Процес її побудови поєднує у собі:

- уточнення моделі БД для наступної генерації SQL-пропозицій;
- уточнення структури інтерфейсу користувача;
- побудову структурних схем, які відображають логіку роботи інтерфейсу користувача і модель бізнес-логіки (SCD – Structure Charts Diagram), та прив'язку їх до форм.

Результатами детального проектування є:

- модель процесів (структурні схеми інтерактивних і неінтерактивних функцій);
- модель даних (визначення в ERD усіх необхідних параметрів для додатків);
- модель інтерфейсу користувача (діаграма послідовності форм (FSD), яка показує, які форми з'являються у додатку та в якому порядку, взаємозв'язок між кожною формою і визначеною структурною схемою, взаємозв'язок між кожною формою й однією або більше сутностями в ERD).

На фазі реалізації будується реляційна модель. Процес її побудови такий:

- генерація SQL-пропозицій, які визначають структуру цільової БД (таблиці, індекси, обмеження цілісності);
- уточнення структурних схем (SCD) і діаграм послідовності форм (FSD) із наступною генерацією коду додатків.

На підставі аналізу потоків даних і взаємодії процесів зі сховищами даних здійснюється остаточне виділення підсистем (попереднє має бути зроблено та зафіксовано на етапі формулювання вимог у технічному завданні). Під час виділення підсистем необхідно керуватися принципами функціональної зв'язаності і мінімізації інформаційної залежності. Слід ураховувати, що на базі таких елементів підсистеми, як процеси і дані, на етапі розроблення має бути створений додаток, здатний функціонувати самостійно. Однак під час групування процесів і даних у підсистемі потрібно враховувати вимоги до конфігурування продукту, якщо вони були сформульовані на етапі аналізу.

8.4. Використання CASE-засобів для проектування баз даних

Історично склалося так, що розробники ІС завжди були останніми в ряду тих, хто використовував комп'ютерні технології для підвищення якості, надійності і продуктивності своєї діяльності (феномен «шевця без чобіт»). Однак «ручне» розроблення породжувало певні проблеми, а саме: неадекватна специфікація вимог, нездатність виявляти помилки в проектних рішеннях, низька якість документації (а отже, зниження експлуатаційної якості), затяжний цикл і незадовільні результати тестування.

Перераховані чинники сприяли появі програмно-технологічних засобів спеціального класу – CASE-засобів, що реалізують CASE-технологію створення і супроводу ІС. Термін CASE (Computer Aided Software Engineering) нині використовують дуже широко. Початкове значення терміна CASE, обмежене питаннями автоматизації розроблення лише програмного забезпечення (ПЗ), сьогодні набуло нового змісту: процес розроблення, що охоплює складні ІС у цілому.

Тепер під терміном CASE-*засоби* розуміють програмні засоби, які підтримують процеси створення і супроводу ІС, включаючи аналіз і формулювання вимог, проектування прикладного ПЗ (додатків) і БД, генерацію коду, тестування, документування, забезпечення якості, конфігураційне керування і керування проектом тощо. *CASE-засоби разом с системним ПЗ і технічними засобами утворюють повне середовище розроблення ІС.*

CASE-технологія являє собою методологію проектування ІС, а також набір інструментальних засобів, що дозволяють у наочній формі моделювати ПО, аналізувати цю модель на всіх етапах розроблення і супроводу ІС і розробляти додатки відповідно до інформаційних потреб користувачів. Більшість існуючих CASE-засобів базуються, як правило, на методологіях структурного (в основному) або об'єктно-зорієнтованого аналізу і проектування, що використовують програмні специфікації у вигляді діаграм або текстів для опису зовнішніх вимог, зв'язків між моделями системи, динаміки поведінки системи й архітектури програмних засобів. У зв'язку з поширенням CASE-технологій слід зазначити, що CASE-засоби не обов'язково дають негайний ефект, його можна досягти тільки через певний час; реальні витрати на впровадження таких засобів зазвичай значно перевищують витрати на їхнє придбання.

Наступні чинники ускладнюють визначення можливого ефекту від використання CASE-засобів: різноманітні якісний склад і можливості, відносно невеликий час використання у різних організаціях і через це нестача досвіду їхнього застосування, застосування у різних організаціях; широкий діапазон ПО проектів, різний ступінь інтеграції у різних проєктах.

Для успішного впровадження CASE-засобів організація має відповідати таким вимогам:

- розуміння обмеженості існуючих можливостей і здатність прийняти нову технологію;
- готовність до впровадження нових процесів і взаємовідносин між розробниками та користувачами;
- чітке управління виробництвом й організація процесу впровадження нових технологій.

Незважаючи на всі зазначені застереження і деякі труднощі, можна вдало використовувати CASE-засоби. Успішне впровадження CASE-засобів

має забезпечити такі вигоди: високий рівень технологічної підтримки процесів розроблення і супроводження ПЗ; позитивний вплив на деякі або усі з перерахованих чинників: продуктивність, якість продукції, дотримання стандартів, документування; прийнятний рівень ефективності від інвестицій у CASE-засоби.

Сучасні CASE-засоби підтримуються у багатьох технологіях: від простих засобів аналізу і документування до повномасштабних засобів автоматизації, що покривають весь життєвий цикл ІС. Найбільш трудомісткими етапами розробки ІС є етапи аналізу і проєктування, у процесі яких CASE-засоби забезпечують якість прийнятих технічних рішень і підготовку проектної документації. При цьому значну роль відіграють методи візуального подання інформації. Це припускає побудову структурних або інших діаграм у реальному масштабі часу, використання різноманітної колірної палітри, наскрізну перевірку синтаксичних правил. Графічні засоби моделювання ПО дозволяють розробникам у наочному вигляді вивчати існуючу ІС, перебудовувати її відповідно до поставлених цілей і наявних обмежень.

До CASE-засобів належить як відносно дешеві системи для персональних комп'ютерів із обмеженими можливостями, так і дорогі системи для неоднорідних обчислювальних платформ і операційних середовищ. Так, сучасний ринок програмних засобів нараховує майже 300 різних CASE-засобів, найпотужніші з яких так чи інакше використовують практично всі головні західні фірми.

До CASE-засобів відносять будь-який програмний засіб, що автоматизує ту або іншу сукупність процесів життєвого циклу ІС і має такі основні характерні риси: застосування потужних графічних засобів для опису і документування ІС; інтеграція окремих компонент CASE-засобів, що забезпечує керованість процесом розроблення ІС; використання спеціальним способом організованої бази проектних метаданих (репозитарія).

Інтегрований (що підтримує повний життєвий цикл ІС) CASE-засіб містить такі компоненти:

- репозитарій, що є основою CASE-засобу. Він має забезпечувати збереження версій проекту і його окремих компонентів, синхронізацію надходження інформації від різних розробників під час групового розроблення, контроль метаданих щодо повноти і несуперечливість;
- графічні засоби аналізу і проектування, що забезпечують створення та редагування ієрархічно зв'язаних діаграм (DFD, ERD тощо), які утворюють моделі ІС;
- засоби розроблення додатків, включаючи мови 4GL і генератори кодів;
- засоби документування;
- засоби тестування;
- засоби керування проектом.

CASE-засоби можна класифікувати за такими ознаками: застосовувані методології та моделі систем і БД, ступінь інтегрованості із СКБД, доступні платформи.

Класифікація за типами відображає функціональну орієнтацію CASE-засобів на ті або інші процеси життєвого циклу. Наведемо основні типи:

- засоби аналізу (Upper CASE) – призначені для побудови й аналізу моделей ПО (Design/IDEF (Meta Software), BPwin (Logic Works));
- засоби аналізу і проектування (Middle CASE) – підтримують найпоширеніші методології проектування і використовуються для створення проектних специфікацій (Vantage Team Builder (Cayenne), Designer/2000 (ORACLE), SilverRun (CSA), PRO-IV (McDonnell Douglas), CASE-аналітик (Macroproject)). Результатом роботи таких засобів є специфікації компонентів та інтерфейсів системи, архітектури системи, алгоритмів і структур даних;
- засоби проектування БД – забезпечують моделювання даних і генерацію схем БД (як правило, мовою SQL) для найпоширеніших СКБД. До них належать ERWin (Logic Works), S-Designor (SDP) і DataBase Designer (ORACLE). Засоби проектування БД є також у складі CASE-засобів Vantage Team Builder, Designer/2000, SilverRun і PRO-IV;
- засоби розроблення додатків – це засоби 4GL (Uniface (Compuware), JAM (JYACC), PowerBuilder (Sybase), Developer/2000 (ORACLE), New Era (Informix), SQL Windows (Gupta), Delphi (Borland) тощо) і генератори кодів, що входять до складу Vantage Team Builder, PRO-IV і частково – у SilverRun;
- засоби реінжинірингу – забезпечують аналіз програмних кодів і схем БД, а також формування на їх базі різних моделей і проектних специфікацій. Засоби аналізу схем БД і формування ERD входять до складу Vantage Team Builder, PRO-IV, SilverRun, Designer/2000, ERWin і S-Designor. У сфері аналізу програмних кодів найбільш поширеними є об'єктно-зорієнтовані CASE-засоби, що забезпечують реінжиніринг програм мовою C++ (Rational Rose (Rational Software), Object Team (Cayenne)).

Допоміжні типи містять засоби:

- планування та керування проектом (SE Companion, Microsoft Project та ін.);
- конфігураційного керування (PVCS (Intersolv));
- тестування (Quality Works (Segue Software));
- документування (SoDA (Rational Software)).

Для прикладу розглянемо структуру та функції одного з поширених комплексних CASE-засобів – пакет SilverRun.

До складу пакета входять і три окремі CASE-засоби:

1. SilverRun BMP – засіб для створення моделі процесів ПО. В інтерактивному режимі дозволяє визначити зовнішні об'єкти, процеси, накопичувачі даних і потоки. Після створення моделі процесів на моделі визначають структури даних, що переносяться в потоках та зберігаються в накопичувачах.

2. SilverRun ERX – засіб для створення ER-моделі ПО. В інтерактивному режимі дозволяє звернутися до моделі, отриманої за допомогою SilverRun BMP та на її основі виконати побудову ER-моделі, тобто:

- визначити початковий набір сутностей;
- провести їх декомпозицію до нормалізованих форм;
- з'єднати сутності за допомогою встановлення зв'язків.

Для полегшення процесу нормалізації можна використати вбудовану програму «Експерт нормалізації». Після створення ER-моделі проводиться перевірка зв'язків між сутностями на кардинальність (обмеження на потужність).

3. SilverRun RDM – засіб для створення концептуальної моделі реляційної БД ПО. В інтерактивному режимі дозволяє звернутися до ER-моделі, отриманої за допомогою SilverRun ERX та на її основі виконати побудову RDM-моделі, тобто:

- отримати RDM-модель в одній з вказаних нотацій;
- створити словник даних для встановлення параметрів відображення даних;
- створити текстовий опис моделі;
- генерувати опис специфікацій на зовнішні ключі відношень;
- перевірити цілісність моделі;
- генерувати сценарії DDL (Data Definition Language), тобто оператори мови SQL для створення об'єктів реляційної БД (доменів, таблиць тощо).

9. Зберігання інформації у базах даних

Реляційні СКБД мають наведені нижче властивості для впливу на організацію зовнішньої пам'яті:

1. Наявність двох рівнів системи: *рівня безпосереднього керування* даними у зовнішній пам'яті (а також, звичайно, керування буферами оперативної пам'яті, керування транзакціями і ведення журналу змін БД); *мовного рівня* (наприклад, рівня, що реалізує мова SQL). При такій організації підсистема нижнього рівня має підтримувати у зовнішній пам'яті набір базових структур, конкретна інтерпретація яких належить до функцій підсистеми верхнього рівня.
2. Підтримка відношень-каталогів. Інформація, пов'язана з іменуванням об'єктів БД і їхніми конкретними властивостями (наприклад, структура ключа індексу), підтримується підсистемою мовного рівня. Як відношення, відношення-каталог не відрізняється від звичайного відношення БД.
3. Регулярність структур даних. Оскільки основним об'єктом реляційної моделі даних є плоска таблиця, головний набір об'єктів зовнішньої пам'яті може мати дуже просту регулярну структуру. При цьому необхідно забезпечити ефективне виконання операторів мовного рівня як над одним відношенням (прості селекція та проекція), так і над декількома відношеннями (наприклад, сполучення декількох відношень). Для цього у зовнішній пам'яті мають підтримуватися додаткові керуючі структури – індекси.
4. Надмірність зберігання даних, яка необхідна для забезпечення надійного зберігання БД, необхідно підтримувати. Надмірність зазвичай реалізується у вигляді журналу змін БД.

Отже, розрізняють такі види об'єктів у зовнішній пам'яті БД:

- рядки відношень – основна частина БД, більшою мірою безпосередньо доступна користувачам;
- керуючі структури – індекси, створені з ініціативи користувача (адміністратора) або верхнього рівня системи з метою підвищення ефективності виконання запитів (як правило, автоматичні підтримуються нижнім рівнем системи);
- журнальна інформація, яка підтримується для задоволення потреби у надійному зберіганні даних;
- службова інформація, яка підтримується для задоволення внутрішніх потреб нижнього рівня системи (наприклад, інформація про вільну пам'ять).

Існують два альтернативні підходи до організації реляційної СКБД з точки зору розподілу функцій між різноманітними компонентами. На-

приклад, у СКБД System R існувала інтегрована підсистема керування даними, транзакціями і журналами змін, а в СКБД Ingres – керування даними було відокремлено від керування транзакціями і журналами змін.

У цих підходів є певні переваги та недоліки. Підхід System R дозволяє використати більш ефективні засоби за рахунок вирішення проблем фізичної та логічної синхронізації, використання загальних протоколів під час керування буферами і журналами змін тощо. Але при цьому у деякому розумінні підсистема нижнього рівня стає монолітом, компоненти залишаються зв'язаними загальними протоколами взаємодії. Локальні зміни одного компонента можуть призвести до фатальних наслідків для всієї системи. Підхід Ingres дозволяє спростити структуру системи і зробити її більш гнучкою, але це можливо тільки за рахунок втрати гнучкості алгоритмів: застосування більш «грубих» засобів керування транзакціями та жорстких протоколів ведення журналу змін.

9.1. Зберігання відношень

Існують два принципові підходи до фізичного зберігання відношень. *Найбільш розповсюдженим є покортежне зберігання відношень* (кортеж є одиницею фізичного зберігання). Природно, це забезпечує швидкий доступ до цілого кортежу, але при цьому у зовнішній пам'яті дублюються загальні значення різних кортежів одного відношення і виникають зайві обміни із нею, якщо потрібна частина кортежу.

Альтернативним (менш розповсюдженим) підходом є зберігання відношення стовпчиками, тобто одиницею зберігання є стовпчик відношення з виключеними дублікатами. Зрозуміло, що за такої організації сумарно витрачається менше зовнішньої пам'яті, оскільки дублікати значень не зберігаються; за один обмін із зовнішньою пам'яттю у загальному випадку можна одержати більше корисної інформації. Додатковою перевагою є можливість використання значень стовпчика відношення для оптимізації виконання операцій приєднання. Але при цьому потрібні істотні додаткові дії для збирання кортежу (або його частини).

Розглянемо алгоритм зберігання відношень кортежами. До основних характеристик цього алгоритму можна віднести такі: кожен кортеж має унікальний ідентифікатор (*tid*), що не змінюється протягом існування кортежу. Як правило, кожен кортеж зберігається цілком в одній сторінці. З цього випливає, що максимальна довжина кортежу будь-якого відношення обмежена розмірами сторінки. Виникає питання: як бути з «довгими» даними, що не вміщуються в одну сторінку? Застосовують декілька засобів.

Найпростішим рішенням є зберігання таких даних в окремих (поза БД) файлах із заміною «довгих» даних у кортежі на ім'я відповідного файлу. У деяких системах (наприклад, у одній із версій СКБД Informix) дані збе-

рігалися в окремому наборі сторінок зовнішньої пам'яті, зв'язаному фізичними посиланнями. Обидва рішення дуже обмежують можливість роботи з «довгими» даними (як, наприклад, усунути декілька байтів із середини двомегабайтного рядка?). В інших системах використовують засіб, запропонований декілька років тому у проекті Exodus, коли «довгі» дані організують у вигляді *B*-дерев послідовностей байтів.

Як правило, в одній сторінці даних зберігаються кортежі тільки одного відношення. Існують, однак, варіанти, коли можливе зберігання в одній сторінці кортежів декількох відношень. Це спричиняє деякі додаткові витрати щодо обробки службової інформації (при кожному кортежі потрібно зберігати інформацію про відповідне відношення), але інколи дозволяє різко скоротити кількість звернень до зовнішньої пам'ятті під час виконання операції приєднання.

Зміна схеми відношення із додаванням нового стовпчика не потребує фізичної реорганізації відношення. Достатньо лише змінити інформацію в описувачі відношення і поширювати кортежі тільки під час занесення інформації в новий стовпчик. Оскільки відношення можуть містити невизначені значення, необхідна відповідна підтримка на рівні зберігання. Як правило, цього досягають шляхом зберігання спеціальної шкали при кожному кортежі, який може містити невизначені значення.

Проблема розподілу пам'яті в сторінках даних, пов'язана з проблемами синхронізації і ведення журналу змін, не завжди є тривіальною. Наприклад, якщо в ході виконання транзакції деяка сторінка даних очищується, то її не можна перевести у статус вільних сторінок до кінця транзакції, оскільки у разі відкату транзакції усунені під час прямого виконання транзакції і відновлені під час її відкату кортежі мають отримати одні й ті самі ідентифікатори.

Розповсюдженим засобом підвищення ефективності СКБД є кластеризація відношення за значеннями одного або декількох стовпчиків. Корисною для оптимізації сполучень є спільна кластеризація декількох відношень.

З метою використання можливостей паралельного обміну із зовнішньою пам'яттю інколи застосовують схему декластеризованого зберігання відношень: кортежі із загальним значенням стовпчика декластеризації розміщують на різних дискових приладах, обмін з яким можна виконувати паралельно.

Що ж стосується зберігання відношень стовпчиками, то сутність полягає у спільному зберіганні всіх значень одного або декількох стовпчиків. Для кожного кортежу відношення зберігається кортеж одного і того самого ступеня, який складається з посилань на місця розташування відповідних значень стовпчиків. Одним з прийомів, що застосовують у розпо-

ділених системах БД, є так званий *вертикальний розподіл відношень*, коли у різних вузлах мережі зберігаються різні проекції даного відношення. Зберігання відношення стовпчиками в деякому розумінні є крайнім випадком вертикального розподілу відношень.

9.2. Індекси

Основним призначенням індексів є забезпечення ефективного прямого доступу до кортежу відношення за ключем. Зазвичай індекс визначається для одного відношення, і ключем є значення атрибута (можливо, складеного). Якщо ключем індексу є можливий ключ відношення, то індекс має бути унікальним, тобто не містити дублікатів ключа. На практиці ситуація звичайно виглядає інакше: під час оголошення первинного ключа відношенню автоматично надається унікальний індекс, а єдиним засобом оголошення можливого ключа, відмінного від первинного, є явне створення унікального індексу. Це пов'язано з тим, що для перевірки збереження унікальності можливого ключа так чи інакше потрібна індексна підтримка.

Оскільки під час виконання багатьох операцій мовного рівня потрібне сортування відношень відповідно до значень деяких атрибутів, корисною властивістю індексу є забезпечення послідовного перегляду кортежів відношення у діапазоні значень ключа, в порядку зростання або зменшення цих значень.

Нарешті, одним із засобів оптимізації виконання еквіпоєднання відношень є організація так званих *мультиіндексів* для декількох відношень, що мають загальні атрибути. Будь-який з цих атрибутів (або їх набір) може бути ключем мультиіндексу. Значенню ключа зіставляється набір кортежів усіх зв'язаних мультиіндексом відношень, значення виділених атрибутів яких збігаються зі значенням ключа.

Загальною ідеєю будь-якої організації індексу, що підтримує прямий доступ за ключем і послідовний перегляд у порядку зростання або зменшення його значень, є зберігання упорядкованого списку значень ключа з прив'язкою до кожного значення ключа списку ідентифікаторів кортежів. Одна організація індексу відрізняється від іншої, переважно, засобом *пошуку ключа із заданим значенням*.

9.3. B-дерева

Найбільш популярним підходом до організації індексів у БД є використання техніки *B*-дерев. *B*-дерево – це збалансоване, дуже розгалужене дерево у зовнішній пам'яті. Збалансованість означає, що довжина шляху від кореня дерева до будь-якого його листка однакова. Розгалуженість дерева – це властивість кожного вузла дерева посилатися на велику кількість вузлів-нащадків. З точки зору фізичної організації, *B*-дерево пода-

ється як мультиспискова структура сторінок зовнішньої пам'яті, тобто кожному вузлу дерева відповідає блок зовнішньої пам'яті (сторінка). Внутрішні і листові сторінки зазвичай мають різну структуру.

Як правило, структура внутрішньої сторінки має такий вигляд:

```
N1 ключ(1)N2 ключ(2) N3.........Nn ключ(n)N(n+1)ключ(n+1)
```

При цьому додержуються таких властивостей:

ключ (1) < = ключ (2) < =...< = ключ (*n*);

На сторінці дерева *Nm* знаходяться ключі *k* зі значеннями ключ (*m*) < = *k* < = ключ (*m*+1).

Листова сторінка має таку структуру:

```
Ключ(1) сп(1) Ключ(2) сп(2) ..... ключ(t) сп (t)
```

Листова сторінка має властивості:

- ключ (1) < ключ (2< ... < ключ (*t*);
- Сп (*r*) – упорядкований список ідентифікаторів кортежів (*tid*), які містять значення ключа (*r*);
- листові сторінки зв'язані одно- або двоспрямованим списком.

Пошук у *B*-дереві – це проходження від кореня до листка згідно із заданим значенням ключа. Зазначимо, що оскільки дерева є дуже розгалуженими і збалансованими, то для виконання пошуку за будь-яким значенням ключа потрібне одне і те саме (як правило, невелике) число обміну із зовнішньою пам'яттю. Більш точно, у збалансованому дереві, де довжини всіх шляхів від кореня до листка одні і ті самі, якщо у внутрішній сторінці зберігається *n* ключів, то при зберіганні *m* записів потрібне дерево глибиною $\log_n(m)$, де $\log_n$ обчислює логарифм на підставі *n*. Якщо *n* достатньо велике (звичайний випадок), то глибина дерева невелика, і пошук виконується швидко.

Основною властивістю *B*-дерев є автоматичне підтримання їх збалансованості. Розглянемо процес виконання операцій занесення і вилучення записів.

Під час занесення нового запису виконуються:

Пошук листової сторінки. Фактично здійснюється звичайний пошук за ключем. Якщо у *B*-дереві не міститься ключ із заданим значенням, то буде отриманий номер сторінки, в якій він має міститися, та відповідні координати всередині сторінки.

Природно, що вся робота виконується у буферах оперативної пам'яті. Листова сторінка, в яку потрібно занести запис, запам'ятовується у буфері, де і виконується операція вставлення. Розмір буфера має перевищувати розмір сторінки зовнішньої пам'яті.

Якщо після вставлення нового запису розмір використовуваної частини буфера не перебільшує розмір сторінки, то на цьому виконання операції занесення запису закінчується. Буфер може бути негайно записаний у зовнішню пам'ять або тимчасово збережений в оперативній пам'яті, залежно від алгоритму керування буферами.

Якщо ж буфер переповнюється (тобто його розмір перевищує розмір сторінки), то виконується розподіл сторінки. Для цього організовується нова сторінка зовнішньої пам'яті, частина буфера ділиться навпіл (так, щоб друга половина також починалася з ключа), і друга половина записується у знов виділену сторінку, а у попередній сторінці модифікується значення розміру вільної пам'яті. Природно, модифікуються і посилання за списком листових сторінок.

Для того щоб забезпечити доступ від кореня дерева до знов створеної сторінки, необхідно відповідно модифікувати внутрішню сторінку, яка є предком листової сторінки, тобто вставити у неї відповідне значення ключа і посилання на нову сторінку. Під час виконання цієї операції може знову відбутися переповнення вже внутрішньої сторінки, і її буде поділено на дві. У результаті необхідно вставити значення ключа і посилання на нову сторінку у внутрішню сторінку-предка вище за ієрархією і т. д.

Крайнім випадком є переповнення кореневої сторінки B-дерева. У цьому разі вона теж розподіляється на дві, і створюється нова коренева сторінка дерева, тобто його глибина збільшується на одиницю.

Під час *вилучення запису* виконують такі дії:

- пошук запису за ключем. Якщо запис не знайдено, то вилучати нічого не потрібно;
- реальне вилучення запису в буфері, до якого прочитана відповідна листова сторінка. Якщо після виконання цієї операції розмір зайнятої у буфері області виявляється таким, що його сума з розміром зайнятої області у листових сторінках, розташованих ліворуч або праворуч від даної сторінки, більше, ніж розмір сторінки, операція завершується, інакше, здійснюється злиття сторінок, тобто у буфері створюється новий образ сторінки, що містить загальну інформацію з даної сторінки і її сусіда праворуч або ліворуч. Звільнена листова сторінка заноситься у список вільних сторінок. Відповідно коригується список листових сторінок.

Для того, щоб уникнути доступу від кореня до вільної сторінки, потрібно усунути відповідне значення ключа і посилання на вільну сторінку із внутрішньої сторінки – її предка. При цьому може виникнути потреба у злитті цієї сторінки з її сусідами ліворуч або праворуч.

Крайнім випадком є повне звільнення кореневої сторінки дерева, що можливо після злиття останніх двох нащадків кореня. Тоді коренева сторінка звільнюється, а глибина дерева зменшується на одиницю.

Як видно, під час виконання операцій вставлення і вилучення властивість збалансованості *B*-дерева зберігається, а зовнішня пам'ять витрачається достатньо ощадливо.

Проблемою є те, що під час виконання операцій модифікації занадто часто можуть виникати розподіли і злиття. Щоб досягти ефективного використання зовнішньої пам'яті з мінімізацією розподілів і об'єднань, застосовують більш складні прийоми, а саме:

- попереджувальний розподіл, тобто розподіл сторінки не у разі її переповнення, а дещо раніше, коли ступінь заповнення досягає деякого рівня;
- «переливання», тобто підтримання рівномірного заповнення сусідніх сторінок;
- злиття «3 в 2», тобто породження двох листових сторінок на основі вмісту трьох сусідніх.

Слід зазначити, що під час організації мультидоступу до *B*-дерев, характерного у разі використання їх СКБД, потрібно вирішувати ряд складних проблем. Звичайне рішення, наприклад, – це монопольне захоплення *B*-дерева протягом виконання операції модифікації. Але існують і більш гнучкі рішення, розгляд яких виходить за межі цього курсу.

9.4. Хешування

Альтернативним і популярним підходом до організації індексів є використання техніки хешування. Загальною ідеєю засобів хешування є застосування до значення ключа деякої функції перетворення (хеш-функції), що виробляє значення меншого розміру. Отримане значення ключа після цього використовують для доступу до запису.

У найпростішому, класичному випадку згортку ключа застосовують як адресу в таблиці, що містить ключі та записи. Основною вимогою до хеш-функції є рівномірний розподіл значення перетворення. У разі виникнення колізій (одного і того самого результату для декількох значень ключа) утворяться ланцюжки переповнення. Головним обмеженням цього засобу є фіксований розмір таблиці. Якщо таблиця занадто заповнена або переповнена, то виникне дуже багато ланцюжків переповнення, і основну перевагу хешування – доступ до запису майже завжди за одне звернення до таблиці – буде втрачено. Розширення таблиці вимагає її повного перероблення на підставі нової хеш-функції (зі значенням функції більшого розміру).

Для БД такі дії є абсолютно неможливими. Тому зазвичай уводять проміжні таблиці-довідники, що містять значення ключів і адреси записів, а самі записи зберігають окремо. Тоді у разі переповнення довідника слід його переробити, що потребує менше накладних витрат.

Для того щоб уникнути повного перероблення довідників, під час їх організації часто використовують техніку *B*-дерев з розподілами та поєд-

наннями. Хеш-функція при цьому змінюється динамічно, залежно від глибини *B*-дерева. Шляхом додаткових технічних рішень можна зберегти порядок записів відповідно до значень ключа. У цілому, засоби *B*-дерев і хешування все більше стають подібними.

9.5. Журнальна інформація

Структура журналу, як правило, є суто індивідуальною для певної реалізації. Розглянемо тільки загальні властивості.

Журнал являє собою послідовний файл із записами змінного розміру, які можна переглядати у прямому або зворотному напрямку. Обмін здійснюється стандартними порціями (сторінками) з використанням буфера оперативної пам'яті. Структура (і тим більше, зміст) журнальних записів відома тільки компонентам СКБД, що відповідають за ведення журналів і їх відновлення. Оскільки вміст журналу є критичнім під час відновлення БД після відмов, до ведення файлу журналу ставлять особливі вимоги щодо надійності, наприклад, прагнуть підтримувати дві ідентичні копії журналу на різних пристроях зовнішньої пам'яті.

9.6. Службова інформація

Для коректної роботи підсистеми керування даними у зовнішній пам'яті необхідно підтримувати інформацію, яка використовується тільки цією підсистемою і є недоступною для підсистеми мовного рівня. Набір структур службової інформації залежить від загальної організації системи, але звичайно потрібне підтримання службових даних, наведених нижче.

Внутрішні каталоги. Описують фізичні властивості об'єктів БД, наприклад, кількість атрибутів відношення, їх розмір і, можливо, типи даних; опис індексів, визначених для даного відношення тощо.

Описувачі вільної і зайнятої пам'яті у сторінках відношення. Така інформація потрібна для знаходження вільного місця під час занесення кортежів. По-різному розв'язують задачу пошуку вільного місця для некластеризованих і кластеризованих відношень (в останньому випадку потрібно додатково використати кластеризований індекс). Нетривіальною є проблема звільнення сторінки в умовах мультидоступу.

Зв'язування сторінок одного відношення. Якщо в одному файлі зовнішньої пам'яті можуть розташовуватися сторінки декількох відношень (як правило, цього прагнуть), то потрібно зв'язати сторінки одного відношення. Тривіальній засіб використання прямих посилань між сторінками часто призводить до виникнення синхронізації транзакцій (наприклад, важко звільнювати і створювати нові сторінки відношення). Тому намагаються використати додаткове зв'язування сторінок з використанням службових індексів, наприклад, загальний механізм для опису вільної пам'яті і зв'язування сторінок на основі *B*-дерев.

10. Структурована мова запитів SQL

10.1. Уведення у мову SQL

Мова SQL (Structured Query Language – структурована мова запитів) орієнтована на операції з даними, поданими у вигляді логічно взаємозв'язаних сукупностей таблиць. Особливістю цієї мови є те, що вона орієнтована на кінцевий результат обробки даних, а не на процедуру самої обробки. Мова SQL сама визначає, де знаходяться дані, які індекси і найефективніші послідовності операцій потрібно використати для отримання даних. SQL – не процедурна мова, за допомогою якої програміст визначає тільки потрібний результат, не вказуючи алгоритм його виконання.

Мова SQL – всесвітній стандарт на засоби роботи з даними на всіх апаратних платформах. У SQL реалізовано концепцію операцій, орієнтованих на табличне подання даних. Ця мова може використовуватись як інтерактивна (для виконання запитів) і як вбудована (для побудови прикладних програм).

До мови SQL входять:

- оператори визначення даних (визначення БД, визначення і вилучення таблиць та індексів);
- запити на вибирання даних;
- оператори модифікації даних (доповнення, вилучення та редагування даних);
- оператори керування даними (надання та відміна привілеїв на доступ до даних, керування транзакціями та ін.).

Крім того, мова SQL дає можливість використовувати такі операції:

- арифметичні оброблення (включаючи функції перетворення даних), оброблення текстових рядків і виконання операцій порівняння;
- упорядкування рядків або полів (стовпчиків) під час виведення таблиць;
- створення представлень (віртуальних таблиць);
- запам'ятовування результату запиту з однієї або декількох таблиць в іншій таблиці (реляційна операція присвоєння);
- агрегування даних: групування даних і застосування до цих груп таких операцій, як середнє, сума, максимум, мінімум, кількість елементів тощо.

Перший міжнародний стандарт мови SQL було прийнято 1989 р. – SQL/89, який розроблено сумісно ANSI й ISO. У кінці 1992 р. прийнято новий міжнародний стандарт мови SQL – SQL/92, яку часто називають SQL2. Найближчим часом з'явиться SQL3, який буде суттєво розширено

щодо об'єктно-зорієнтованого підходу до роботи з даними і використання засобів мультимедіа.

Мову SQL застосовують для виконання запитів до СКБД; виконання операцій зі створення реляційних структур; збереження даних та їх оброблення. Виконання запитів здійснюють або стандартні (файлові) СКБД, для яких БД – це сукупність файлів у каталогах операційної системи (наприклад, СКБД FoxPro, PARADOX), або сервери БД, для яких уся БД розміщується на дисковому просторі, що з точки зору операційної системи є одним файлом (сервери INTERBASE, MS SQL SERVER, INFORMIX, ORACLE).

Загальні функції серверів БД. Розглянемо загальні функції серверів БД:

1. Забезпечення захисту даних шляхом використання механізму, за яким контролюють доступ до інформації, збережуваної у БД. Для сервера INTERBASE механізм захисту полягає в тому, що під'єднання до сервера вимагає введення імені користувача та, у разі необхідності, пароля. Користувачі створюються за допомогою спеціальної утиліти SERVER Manager. Для користувача визначають ім'я та пароль. Забезпечується обмеження множини операцій (оновлювати базу або виконувати процедури), які дозволено виконувати користувачу.
2. Організація фізичного простору для збереження даних, тобто створення та підтримка БД, точніше файлу БД.
3. Створення основних об'єктів БД: таблиць, представлень і курсорів.
4. Забезпечення для створених основних об'єктів підтримки виконання бізнес-правил, тобто обмеження на внутрішні та зовнішні значення полів таблиць.
5. Створення індексів для обслуговування первинних, вторинних і зовнішніх ключів для таблиці БД.
6. Надання можливості зберігання на сервері типових алгоритмів оброблення та забезпечення звернення до них. Їх оформлюють як збережені процедури сервера.
7. Надання можливості визначення та виконання стандартних дій, які супроводжують операції над записами БД, наприклад, протоколювання оновлення бази або створення копій; виведення повідомлень; побудова ланцюжка оновлень. Такі дії оформлюють як тригери БД.

Примітка. Процедури та тригери створюють за допомогою спеціальної мови програмування – мови процедур і тригерів.

8. Надання можливості оброблення ситуацій.
9. Створення спеціальних механізмів (генераторів) для отримання унікальних значень, як правило, для ідентифікації записів.

10. Підтримка спеціальних таблиць для збереження метаданих бази, тобто відомостей про об'єкти БД і спеціальних утиліт для їх перегляду та оброблення.
11. Забезпечення механізмів, що дозволяють відмінити оновлення БД за бажанням користувача та регулювати паралельні оновлення БД множиною користувачів. Для цього застосовують механізми кешованих оновлень і керування транзакціями.

Ураховуючи, що різні СКБД (сервери БД) підтримують свої діалекти мови SQL, надалі конструкції мови розглянемо на прикладі діалекту мови SQL для сервера INTERBASE.

10.2. Типи даних

У мові SQL підтримуються такі типи даних: `char`, `varchar`, `numeric`, `decimal`, `integer`, `smallint`, `float`, `smallfloat`, `real`, `double precision`, `date`, `datetime`, `text`, `money`, `bit`. Ці типи даних поділяють на типи рядків символів, точних і приблизних чисел.

До першого класу належать типи `char` та `varchar`.

Специфікатори:

`char (n)` – `n` задає довжину рядка даного типу. Літерні рядки символів зображаються у вигляді «послідовність_символів» (наприклад, «example»);

`varchar(n,r)` – `n` – максимальна довжина (не більше 255), `r` – необов'язковий параметр, який встановлює нижню межу довжини елементів даних.

Представниками другого класу є `numeric`, `decimal` (або `dec`), `integer` (або `int`) і `smallint`.

Специфікатори:

`numeric [(p[,s])]` – точні числа, які задаються довжиною числа `p` і кількістю десяткових знаків `s`. Якщо опущено `s`, то вважають, що `s` = 0, а якщо опущено `p`, то значення за замовчуванням визначають у реалізації;

`decimal [(p[,s])]` – аналогічний `numeric`;

`integer` – цілі числа в діапазоні від – 2147483647 до 2147483646 (займає 4 байт);

`smallint` – цілі числа в діапазоні від – 32,768 до 32,767 (займає 2 байт).

До типів даних приблизних чисел (чисел з плаваючою комою) належать `float`, `smallfloat`, `real` і `double precision`.

Специфікатори:

`float [(p)]`	– двійкові числа з подвійною точністю, з плаваючою комою і зі знаком, де p – довжина числа;
`real, smallfloat`	– двійкові числа з одинарною точністю, визначеною в реалізації, взяті за модулем (без знака);
`double precision`	– тип даних приблизних чисел з точністю, визначеною в реалізації і більшою, ніж точність типу `real`.

Зазначимо, що хоча і можна за допомогою мови SQL визначити схему БД, яка містить дані будь-якого з перерахованих типів, можливість використання цих даних у прикладних системах залежить від застосовуваної мови програмування.

Зазначимо також, що у більшості реалізацій SQL підтримуються деякі додаткові типи даних, наприклад `date`, `datetime` (або `time`), `interval`, `money`, `text`, `bit`. Деякі з цих типів специфіковані у стандарті SQL2, але в поточних реалізаціях синтаксичні та семантичні властивості таких типів можуть відрізнятися.

Опис додаткових типів:

`date`	– календарні дати (день, місяць, рік);
`datetime`	– календарні дати і час (день, місяць, рік, година, хвилина, секунда);
`interval`	– інтервали часу;
`money`	– грошовий вираз числової величини;
`text`	– блок текстової інформації;
`bit`	– булєвий тип даних (*true* або *false*).

10.3. Засоби визначення схеми

10.3.1. Визначення схеми

Відповідно до правил SQL кожна таблиця даної БД має просте й уніфіковане ім'я. Кваліфікатором імені є *ідентифікатор повноважень* таблиці, який зазвичай збігається з ім'ям деякого користувача даної БД. Уніфіковане ім'я таблиці має такий вигляд:

```
<ім`я БД>. <ідентифікатор повноважень (ім`я користувача)>.
<ім`я таблиці>
```

Підхід до визначення схеми в SQL полягає в тому, що всі таблиці з одним ідентифікатором повноважень створюють (визначають) шляхом виконання одного оператора визначення схеми. При цьому у стандарті не

визначено спосіб виконання оператора визначення схеми: чи має він виконуватися тільки в інтерактивному режимі, чи може бути вбудований у програму, написану традиційною мовою програмування.

В операторі визначення схеми міститься ідентифікатор повноважень і список елементів схеми, кожний з яких може бути визначенням таблиці, визначенням представлення (*view*) або визначенням привілеїв. Кожне з цих визначень представляється окремим оператором SQL, але всі вони мають бути вбудовані в оператор визначення схеми.

Для цих операторів у наступних розділах буде наведено синтаксис, оскільки це дозволить більш точно описати їхні особливості. Під час опису операторів SQL використовуватимемо оператори пошуку та поновлення даних у таблицях, а саме:

- `SELECT` – вибрати групу записів з таблиці відповідно до вказаного критерію;
- `UPDATE` – оновити групу записів у таблиці;
- `DELETE` – вилучити групу записів з таблиці.

Детальне пояснення цих операторів буде наведено далі.

10.3.2. Підтримка бази даних

Оператор створення БД CREATE DATABASE

Синтаксис:

```
CREATE {DATABASE | SCHEMA} "<filespec>"
[USER "username" [PASSWORD "password"]]
[PAGE_SIZE [=] int]
[LENGTH [=] int [PAGE[S]]]
[DEFAULT CHARACTER SET charset]
[<secondary_file>];
<secondary_file> = file "<filespec>" [<fileinfo>]
[<secondary_file>]
<fileinfo> = LENGTH [=] int [PAGE[S]] | STARTING [AT
[PAGE]] int [<fileinfo>]
```

Опис аргументів:

- `SCHEMA` – альтернативне ключове слово для DATABASE;
- `"<filespec>"` – специфікація імені файлу БД, формат імені залежить від ОС;
- `USER "username"` – ім'я користувача сервера БД;

PASSWORD "password"	–	пароль користувача сервера БД; ім'я та пароль для сервера встановлює утиліта *server manager*;
PAGE_SIZE [=] int	–	розмір сторінки БД. Може бути 1024 (за замовчуванням), 2048, 4096 або 8192;
DEFAULT CHARACTER SET <charset>	–	ім'я набору символів для правильного оброблення алгоритмів сортування. Для кириличних текстів необхідно встановити WIN1251 та вибрати мовний драйвер Pdox ANSI Cyrillic. За замовчуванням – значення NONE;
FILE "<filespec>"	–	назва одного або кількох додаткових файлів, в яких будуть зберігатися сторінки БД, якщо основний файл буде заповнено. Для віддалених серверів ім'я вузла не має вказуватись;
STARTING [AT [PAGE]] int	–	номер початкової сторінки;
LENGTH [=] int [PAGE[S]]	–	кількість сторінок для основного або додаткового файлу.

Наприклад, оператор створює БД з двох файлів і визначає ім'я набору символів за замовчуванням:

```
CREATE DATABASE "employee.gdb"
 DEFAULT CHARACTER SET "ISO8859_1"
 FILE "employee.gd1" STARTING AT PAGE 10001 LENGTH
 10000 PAGES;
```

Оператор визначення додаткових файлів до БД ALTER DATABASE
Синтаксис:

```
ALTER {DATABASE | SCHEMA}
ADD <add_clause>;
<add_clause> = FILE "<filespec>" [<fileinfo>]
[<add_clause>]
<fileinfo> = LENGTH [=] int [PAGE[S]] | STARTING
[AT [PAGE]] int [<fileinfo>]
```

Опис аргументів:

SCHEMA	– альтернативне ключове слово для DATABASE;
ADD FILE "<filespec>"	– опис додаткових файлів БД, які будуть використовуватись у разі заповнення основного файлу;
LENGTH [=] int [PAGE[S]]	– кількість сторінок для додаткового файлу;
STARTING [AT [PAGE]] int	– номер початкової сторінки додаткового файлу.

Наприклад, наступний оператор додає два файли до існуючої БД:

```
ALTER DATABASE
    ADD FILE "employee.gd1"
STARTING AT PAGE 10001
    LENGTH 10000
ADD FILE "employee.gd2"
    LENGTH 10000;
```

Оператор вилучення файлу БД (тобто активної БД)
DROP DATABASE

Синтаксис:

DROP DATABASE – вилучити поточну БД зі всіма пов'язаними файлами.

10.3.3. Підтримка доменів

Домен – структура шаблону БД, яка відповідає поняттю домену ПО та використовується під час створення таблиць БД.

Оператор створення домену CREATE DOMAIN

Синтаксис:

```
CREATE DOMAIN <domain> [AS] <datatype>
[DEFAULT {literal | NULL | USER}]
[NOT NULL] [CHECK (<dom_search_condition>)]
[COLLATE <collation>];
```

Примітка. Операнд COLLATE не можна використовувати для BLOB-полів.

```
<datatype> = {
{SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION}
[<array_dim>]
    | {DECIMAL | NUMERIC} [(precision [, scale])]
    [<array_dim>]
```

```
    | DATE [<array_dim>]
    | {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}
        [(1...32767)] [<array_dim>]
        [CHARACTER SET charname]
    | {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
        [VARYING] [(1...32767)] [<array_dim>]
    | BLOB [SUB_TYPE {int | subtype_name}] [SEGMENT SIZE n]
        [CHARACTER SET charname]
    | BLOB [(seglen [, subtype])]
    }
  <array_dim> = [x:y [, x:y ...]]
```

Примітка. Квадратні дужки необхідно включати до визначення масиву.

```
<dom_search_condition> = {
VALUE <operator> <val>
    | VALUE [NOT] BETWEEN <val> AND <val>
    | VALUE [NOT] LIKE <val> [ESCAPE <val>]
    | VALUE [NOT] IN (<val> [, <val> ...])
    | VALUE IS [NOT] NULL
    | VALUE [NOT] CONTAINING <val>
    | VALUE [NOT] STARTING [WITH] <val>
    | (<dom_search_condition>)
    | NOT <dom_search_condition>
    | <dom_search_condition> OR <dom_search_condition>
    | <dom_search_condition> AND <dom_search_condition>
    }
<operator> = {= | < | > | <= | >= | !< | !> | <> | !=}
```

Опис аргументів:

domain	– унікальне ім'я домену;
<datatype>	– SQL-тип даних;
DEFAULT	– визначення значення за замовчуванням для поля домену. Можливі значення: літерний рядок, число або дата;
NULL	– визначення значення NULL («порожнього» значення поля);
USER	– уведення імені поточного користувача як поля, значення поля за замовчуванням;
NOT NULL	– визначення, яке вказує, що поле не може мати значення NULL («порожнього» значення);

CHECK – визначення правил перевірки значень;
VALUE – визначення оператора порівняння;
COLLATE – визначення послідовності з'єднань текстових даних.

Приклади використання:

```
CREATE DOMAIN CUSTNO
    AS INTEGER
         DEFAULT 9999
         CHECK (VALUE > 1000);
CREATE DOMAIN PRODTYPE
    AS VARCHAR(12)
         CHECK (VALUE IN ("software", "hardware",
         "other", "N/A"));
```

Оператор модифікації домену ALTER DOMAIN

Синтаксис:

```
ALTER DOMAIN <name> {
[SET DEFAULT {literal | NULL | USER}]
| [DROP DEFAULT]
| [ADD [CONSTRAINT] CHECK (<dom_search_condition>)]
| [DROP CONSTRAINT]
};
<dom_search_condition> = {
VALUE <operator> <val>
| VALUE [NOT] BETWEEN <val> AND <val>
| VALUE [NOT] LIKE <val> [ESCAPE <val>]
| VALUE [NOT] IN (<val> [, <val> ...])
| VALUE IS [NOT] NULL
| VALUE [NOT] CONTAINING <val>
| VALUE [NOT] STARTING [WITH] <val>
| (<dom_search_condition>)
| NOT <dom_search_condition>
| <dom_search_condition> OR <dom_search_condition>
 <dom_search_condition> AND <dom_search_condition>
}
<operator> = {= | < | > | <= | >= | !< | !> | <> | !=}
```

Опис аргументів:

<name> – ім'я існуючого домену;

SET DEFAULT – визначення значення за замовчуванням;
DROP DEFAULT – вилучення значення за замовчуванням;
ADD [CONSTRAINT] CHECK <dom_search_cond> – додавання правил перевірки. До домену можна застосовувати тільки одне правило перевірки;
DROP CONSTRAINT – вилучення правил перевірки.

Приклад використання:

```
CREATE DOMAIN CUSTNO
    AS INTEGER
        CHECK (VALUE > 1000);
ALTER DOMAIN CUSTNO SET DEFAULT 9999;
```

Оператор вилучення домену DROP DOMAIN

Синтаксис:

DROP DOMAIN <name> – вилучити зі схеми БД домен з ім'ям name.

10.3.4. Підтримка таблиць

Поняття *таблиця* (TABLE), як правило, пов'язано з реальною таблицею, тобто з таблицею, кожний рядок якої зберігається у фізичній пам'яті машини. Таблиці містять відношення реляційної моделі ПО. Створення схеми таблиці полягає у визначенні структури таблиці (списку полів) та визначенні властивостей, які впливають на її використання.

Однак SQL використовує створені на базі існуючих (базових) таблиць віртуальні (начебто існуючі) таблиці, так звані *представлення* (*view*) та *курсори* (*cursor*).

Представлення – це таблиця, яка будується шляхом відбору інформації з існуючих таблиць на основі критеріїв відбору полів і рядків. Для користувача представлення майже не відрізняються від базових таблиць (є лише деякі обмеження під час виконання різних операцій маніпулювання даними). Їх можна використовувати як в інтерактивному режимі, так і в прикладних програмах.

Курсор – підмножина рядків і полів з базових таблиць, до якої надається послідовний доступ. Курсори створені для процедурної роботи з базовою таблицею у прикладних програмах.

Оператор створення схеми таблиці CREATE TABLE

Синтаксис:

```
CREATE TABLE <table> [EXTERNAL [FILE] "<filespec>"]
(<col_def> [, <col_def> | <tconstraint> ...]);
<col_def> = col {datatype | COMPUTED [BY] (<expr>) |
```

```
<domain>}
[DEFAULT {literal | NULL | USER}]
  [NOT NULL] [<col_constraint>]
  [COLLATE <collation>]
```

Примітка. Операнд COLLATE не можна використовувати для BLOB-полів.

```
<datatype> = {
{SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION}
[<array_dim>]
 | {DECIMAL | NUMERIC} [(precision [, scale])]
 [<array_dim>]
 | DATE [<array_dim>]
 | {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}
     [(int)] [<array_dim>] [CHARACTER SET charname]
| {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
     [VARYING] [(int)] [<array_dim>]
| BLOB [SUB_TYPE {int | subtype_name}] [SEGMENT SIZE int]
     [CHARACTER SET <charname>]
| BLOB [(seglen [, subtype])] }
<array_dim> = [x:y [, x:y ...]]
```

Примітка. Квадратні дужки необхідно включати до визначення масиву.

```
<expr> = правильний SQL-вираз, який повертає одне значення
<col_constraint> = [CONSTRAINT <constraint>] <constraint_def>
[<col_constraint>]
<constraint_def> = {UNIQUE | PRIMARY KEY
| CHECK (<search_condition>)
| REFERENCES <other_table>
[(<other_col> [, <other_col> ...])]}
  <tconstraint> = CONSTRAINT
  <constraint <tconstraint_def>
[<tconstraint>]
  <tconstraint_def> = {{PRIMARY KEY | UNIQUE}
  (col [, col ...])
| FOREIGN KEY (col [, col ...])
REFERENCES <other_table>
  | CHECK (<search_condition>)}
  <search_condition> =
{<val> <operator> {<val> | (<select_one>)}
  | <val> [NOT] BETWEEN <val> AND <val>
  | <val> [NOT] LIKE <val> [ESCAPE <val>]
  | <val> [NOT] IN (<val> [, <val> ...] | <select_list>)
```

```
  | <val> IS [NOT] NULL
  | <val> {[NOT] {= | < | >} | >= | <=}
      {ALL | SOME | ANY} (<select_list>)
  | EXISTS (<select_expr>)
  | SINGULAR (<select_expr>)
  | <val> [NOT] CONTAINING <val>
  | <val> [NOT] STARTING [WITH] <val>
  | (<search_condition>)
  | NOT <search_condition>
  | <search_condition> OR <search_condition>
  | <search_condition> AND <search_condition>}
<val> = {
<col> [<array_dim>] | <constant> | <expr> | <function>
  | NULL | USER | RDB$DB_KEY  } [COLLATE collation]
<constant> = число | "рядок" | ім'я набору символів "рядок"
<function> = {COUNT (* | [ALL] <val> | DISTINCT <val>)
  | SUM ([ALL] <val> | DISTINCT <val>)
  | AVG ([ALL] <val> | DISTINCT <val>)
  | MAX ([ALL] <val> | DISTINCT <val>)
  | MIN ([ALL] <val> | DISTINCT <val>)
  | CAST (<val> AS <datatype>)
  | UPPER (<val>)
  | GEN_ID (generator, <val>)      }
<operator> = {= | < | > | <= | >= | !< | !> | <> | !=}
```

<select_one> = оператор SELECT для одного поля, який повертає тільки одне значення.

<select_list> = SELECT для одного поля, який повертає список значень (може бути порожній).

<select_expr> = SELECT для списку полів, який повертає список значень (може бути порожній).

Опис аргументів:

Table	–	унікальне ім'я таблиці у БД;
EXTERNAL [FILE] "<filespec>"	–	специфікація шляху до зовнішнього файлу, з якого беруться значення;
<col>	–	унікальне ім'я поля;
<datatype>	–	SQL-тип даних для поля;
COMPUTED [BY] (<expr>)	–	вираз, який повертає значення поля;

<domain>	–	ім'я існуючого домену в схемі БД;
COLLATE <collation>	–	визначення правила поєднання текстових полів;
DEFAULT	–	визначення значення за замовчуванням. Можливі значення: літерний рядок, число або дата. NULL: визначити значення NULL. USER: визначити як значення ім'я користувача. Значення за замовчуванням, яке визначено в таблиці, перекриває значення, яке визначено в домені;
CONSTRAINT <constraint>	–	визначення імені обмеження перевірки цілісності.

Наведемо більш детальні пояснення до оператора CREATE TABLE. Визначення оператора можна навести у вигляді:

```
CREATE TABLE <Ім`я таблиці> (<Визначення поля>
[,<Визначення поля> [,...]]) | [<Обмеження цілісності>]
```

Крім імені таблиці, в операторі вказується список елементів таблиці, кожний з яких використовують або для визначення стовпчика (поля), або для визначення обмеження цілісності таблиці. Вимагається наявність хоча б одного визначення поля. Оператор CREATE TABLE визначає так звану *базову таблицю*, тобто реальне сховище даних.

Для визначення полів таблиці й обмежень цілісності використовують спеціальні оператори, які мають бути вкладені в оператор визначення таблиці.

Визначення поля описується такими синтаксичними правилами:

```
<Визначення поля > = <Ім`я поля> <Тип даних>
[(<Довжина поля>)] [DEFAULT <Вираз> | USER | NULL ]
[NOT NULL] | [<Обмеження поля> | CHECK (<вираз>)]
```

Як видно, крім обов'язкової частини, в якій задається ім'я поля і його тип даних, визначення поля може містити два необов'язкові розділи: розділ значення поля за замовчуванням опції DEFAULT і розділ обмежень цілісності поля.

У розділі значення за замовчуванням вказується, яке початкове значення слід помістити у поле. Значення за замовчуванням може бути вказане у вигляді літерної константи з типом, що відповідає типу поля; у вигляді ключового слова USER, що дає можливість занести ім'я поточного користувача у поле (у цьому разі поле повинно мати тип символьних рядків); шляхом задання ключового слова NULL, яке дозволяє набувати значення типу NULL або, навпаки, забороняти введення таких значень (NOT NULL).

Якщо значення поля за замовчуванням не специфіковано, і в розділі обмежень цілісності поля вказано NOT NULL (тобто наявність невизначе-

них значень заборонено), спроба занести у таблицю рядок з неспецифікованим значенням даного поля призведе до помилки.

Вказівка у розділі обмежень цілісності NOT NULL приводить до неявного породження перевірочного обмеження цілісності для всієї таблиці "CHECK (C IS NOT NULL)" (де C – ім'я даного поля). За допомогою опції CHECK можна вказати список можливих значень для введення даних. Якщо обмеження NOT NULL не вказане, і розділу замовчувань не буде, тоді неявно породжується розділ замовчувань DEFAULT NULL. Якщо вказано специфікацію унікальності, тоді породжується відповідна специфікація унікальності для таблиці.

Приклади визначення таблиці:

1. Визначення таблиці без правил обмежень.

```
CREATE TABLE tab1(
      field1  integer
      field2  char(10)
      field3  decimal);
```

2. Визначення таблиці з обмеженнями на значення полів.

```
CREATE TABLE tab2 (
      field1  integer          PRIMARY KEY
      field2  varchar(50,20)   NOT NULL UNIQUE
      field3  decimal          CHECK (field3<999)
      field4  decimal          CHECK (field4>field3*field1)
      field5  char(15)         DEFAULT "New record";
```

Обмеження PRIMARY KEY приводить до встановлення значень NOT NULL та UNIQUE.

3. Визначення таблиці з обмеженнями на таблицю.

```
CREATE TABLE tab1(
      field1  char(10)
      field2  char(15)
      field3  decimal
      field4  decimal
      PRIMARY KEY (field1,field2)
      UNIQUE (field3,field4);
```

4. Визначення таблиці з комбінованими обмеженнями.

```
CREATE TABLE tab1(
      field1  char(10)
      field2  char(15)
      field3  decimal CHECK   (field3<999)
      field4  decimal DEFAULT field3/35
```

```
PRIMARY KEY (field1,field2)
UNIQUE (field3,field4);
```

Під час створення таблиці можемо визначити *обмеження цілісності*. Існує три види визначень обмежень таблиць:

- визначення потенційного ключа;
- визначення зовнішнього ключа;
- визначення умови перевірки значень.

Ці обмеження можна задавати під час створення таблиці (опис обмеження наводиться одразу після імені поля і його типу в команді CREATE TABLE) або окремими командами (тоді перед цими визначеннями має бути інструкція вигляду "CONSTRAINT <ім`я обмеження>").

Розглянемо ці обмеження:

Визначення потенційного ключа має такий синтаксис: PRIMARY KEY (список полів) – визначення первинного ключа або UNIQUE (список полів) – визначення альтернативного унікального ключа В обох випадках список не може бути порожнім. Для однієї таблиці може існувати лише один первинний ключ PRIMARY KEY і велика кількість альтернативних ключів UNIQUE. У випадку первинного ключа для кожного з указаних полів, які його складають за замовчуванням, вказується специфікація NOT NULL.

Визначення зовнішнього ключа має такий синтаксис: FOREIGN KEY (список полів зовнішнього ключа) REFERENCES <ім'я батьківської таблиці> [(список полів батьківської таблиці)]. Список полів зовнішнього ключа визначає поля дочірньої таблиці, за якими будується зовнішній ключ. Ім'я батьківської таблиці визначає таблицю, в якій описаний первинний ключ (або поле). На цей ключ має посилатися даний зовнішній ключ для забезпечення цілісності. Список полів батьківської таблиці не є обов'язковим, якщо посилання здійснюється на первинний ключ. Якщо посилання виконується на поле з атрибутом UNIQUE, то цей список краще задати.

Визначення умов перевірки значень. Якщо вказано перевірне обмеження поля, то неявно породжується відповідне перевірне обмеження і для всієї таблиці. Для однієї таблиці можна задати декілька обмежень цілісності, в тому числі і ті, що неявно породжуються обмеженнями цілісності полів.

Зауваження. Наявність правильно обраного набору обмежень БД дуже важлива для надійного функціонування прикладної ІС. Разом із тим, у деяких СКБД обмеження цілісності практично не підтримуються. Тому під час проектування прикладної системи необхідно прийняти рішення про те, що є більш важливим: розраховувати на підтримку обмежень цілісності, але обмежити набір можливих СКБД, або відмовитися від їх використання на рівні SQL.

Розглянемо ці обмеження на прикладі таблиці *T*, для якої визначаються обмеження цілісності.

Обмеження унікальності. Кожне ім'я поля у списку унікальності має називати поле *T* і не має входити в цей список більше одного разу. Під час визначення поля, яке входить у список унікальності, необхідно вказувати обмеження поля NOT NULL. Серед обмежень унікальності *T* не може бути більше одного визначення первинного ключа (обмеження унікальності з ключовим словом PRIMARY KEY).

Дія обмеження унікальності полягає в тому, що в таблиці *T* не допускається поява двох або більше рядків, значення полів унікальності яких збігаються.

Обмеження за посиланнями. Обмеження за посиланнями від заданого набору полів *CT* таблиці *T* на заданий набір полів *CT1* деякої визначеної до цього моменту таблиці *T1* задає умову на вміст обох цих таблиць, за яким посилання можна вважати коректними.

Якщо список полів *CT1* є явно специфікованим у визначенні обмеження за посиланнями, то потрібно, щоб він явно входив у яке-небудь визначення унікальності таблиці *T1*. Якщо список *CT1* не специфіковано явно у визначенні обмеження за посиланнями таблиці *T*, потрібно, щоб у визначенні таблиці *T1* було наявне визначення первинного ключа. Список *CT1* тоді неявно вважають таким, що збігається зі списком імен полів із визначення первинного ключа таблиці *T1*. Імена полів у списках *CT* і *CT1* мають збігатися з іменами полів таблиць *T* і *T1* та з'являтися у списках по одному разу. Списки полів *CT* і *CT1* мають містити однакову кількість елементів. Крім того, поле таблиці *T*, що ідентифікується *i*-м елементом списку *CT*, повинно мати той самий тип, що й поле таблиці *T1*, яке ідентифікується *i*-м елементом списку *CT1*.

Визначення: таблиці *T* і *T1* задовольняють задане обмеження за посиланнями, якщо:

- для кожного рядка *s* таблиці *T* всі значення полів рядка *s*, які ідентифікуються списком *CT*, не є невизначеними;
- існує рядок *s1* таблиці *T1* такий, що значення полів рядка *s1*, які ідентифікуються списком *CT1*, позиційно дорівнюють значенням полів *s*, що ідентифікуються списком *CT*.

Більш точно це можна сформулювати так: обмеження за посиланнями задовольняється, якщо для будь-якого коректного посилання існує об'єкт, на який воно посилається, тобто обмеження за посиланнями не дозволяє робити «висячі» посилання, які не ведуть ні до якого об'єкта.

Перевірне обмеження. Це обмеження специфікує умову, яку має задовольняти окремо кожен рядок таблиці *T*. Умова не має містити параметри, підзапити, специфікації агрегатних функцій, а також посилання на зовнішні змінні. У неї можуть входити тільки імена полів даної таблиці та літерні константи.

Таблиця задовольняє перевірне обмеження цілісності лише в тому разі, якщо обчислення умови для кожного рядка таблиці дасть істину.

Зауваження. У деяких реалізаціях допускаються розширені механізми обмежень за посиланнями і перевірних обмежень. Треба бути уважними, щоб не виходити за межі можливостей стандарту.

Оператор модифікації структури таблиці ALTER TABLE

Синтаксис:

```
ALTER TABLE <table> <operation> [, <operation> ...];
<operation> = {ADD <col_def> | ADD <table_constraint>
| DROP col
| DROP CONSTRAINT <constraint>}
<col_def> = <col> {<datatype> | [COMPUTED [BY] (<expr>)
| domain}
[DEFAULT {literal | NULL | USER}]
 [NOT NULL] [<col_constraint>]
 [COLLATE collation]
```

Примітка. Операнд COLLATE не можна використовувати для BLOB-полів.

```
<col_constraint> = [CONSTRAINT constraint]
<constraint_def>
[<col_constraint>]
<constraint_def> = {PRIMARY KEY | UNIQUE
| CHECK (<search_condition>)
| REFERENCES <other_table> [(<other_col>
[, <other_col> ...])]}
  <datatype> = {
{SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION}
[<array_dim>]
| {DECIMAL | NUMERIC} [(<precision> [,<scale>])]
[<array_dim>]
| DATE [<array_dim>]
| {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}
    [(1...32767)]    [<array_dim>]    [CHARACTER    SET
    charname]
| {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
     [VARYING] [(1...32767)] [<array_dim>]
| BLOB [SUB_TYPE {int | <subtype_name>}]
[SEGMENT SIZE int]
     [CHARACTER SET <charname>]
| BLOB [(<seglen> [, <subtype>])]
}
```

```
<array_dim> = [x:y [, x:y ...]]
```

Примітка. Квадратні дужки необхідно включати до визначення масиву.

```
<table_constraint> = CONSTRAINT constraint
<tconstraint_opt>
[<table_constraint>]
<tconstraint_opt> = {
{PRIMARY KEY | UNIQUE} (col [, col ...])
| FOREIGN KEY (col [, col ...]) REFERENCES other_table
| CHECK (<search_condition>)
}
```

Примітка. Повний синтаксис <search_condition> наведено в операторі CREATE TABLE.

Опис аргументів:

<table>	– ім'я таблиці, опис якої змінюється;
<operation>	– дія щодо перетворення таблиці. Можливі значення: ADD – додати поле або обмеження таблиці; DROP – вилучити поле або обмеження таблиці;
<col_def>	– опис нового поля. Має містити ім'я, тип, значення за замовчуванням та обмеження значень;
<table_constraint>	– опис нового обмеження цілісності. До таблиці можна додати тільки одне обмеження цілісності;
<col>	– ім'я поля, що додається або вилучається. Має бути унікальним у таблиці;
<constraint>	– ім'я обмеження, що додається або вилучається. Має бути унікальним у таблиці;
COLLATE <collation>	– додавання правила об'єднання даних;
<datatype>	– тип даних поля, що додається;
<domain>	– ім'я домену, на який посилається визначення поля;
COMPUTED [BY] <expr>	– вираз, який повертає значення поля;
NOT NULL	– визначення, яке вказує, що поле не може мати значення NULL. Якщо таблиця не порожня, то не можна визначати такий атрибут для існуючого поля;

DEFAULT — визначення значення за замовчуванням. Можливі значення: літерний рядок, число або дата. NULL: визначити значення NULL. USER: визначити як значення ім'я користувача. Значення за замовчуванням, яке визначено в таблиці, перекриває значення, визначене в домені;

<constraint_def> — визначення обмеження цілісності;

CONSTRAINT — додати іменоване обмеження цілісності;

DROP CONSTRAINT — вилучити обмеження цілісності.

Приклади використання:

```
ALTER TABLE COUNTRY
 ADD CAPITAL VARCHAR(25),
 DROP CURRENCY;
ALTER TABLE COUNTRY
 ADD CAPITAL VARCHAR(25) UNIQUE,
 ADD LARGEST_CITY VARCHAR(25) NOT NULL;
```

Оператор вилучення таблиці DROP TABLE

Синтаксис:

DROP TABLE <name> – вилучити зі схеми БД таблицю з іменем name.

10.3.5. Підтримка індексів

Індекси (спеціальні допоміжні таблиці в БД) призначено для забезпечення потрібного порядку сортування даних, підтримування зв'язків між таблицями та для прискорення вибірки даних.

Оператор створення індексів CREATE INDEX

Синтаксис:

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

Опис параметрів:

UNIQUE — параметр, який визначає, що буде створено унікальний індекс, у якому ніколи не повторюється значення у двох різних записах по всіх полях, які входять в індекс;

DESC[ENDING] — параметр, який указує на необхідність сортування значень індексних полів за зменшенням;

<index> – унікальне ім'я індексу;
<table> – ім'я таблиці, для якої будується індекс;
<col> – ім'я поля, за яким будується індекс.

Приклади використання:

```
CREATE UNIQUE INDEX PRODTYPEX ON PROJECT (PRODUCT,
PROJ_NAME);
CREATE DESCENDING INDEX CHANGEX ON SALARY_HISTORY
(CHANGE_DATE);
CREATE INDEX NAMEX ON EMPLOYEE (LAST_NAME, FIRST_NAME);
```

Оператор модифікації індексів ALTER INDEX

Синтаксис:

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

Опис параметрів:

<name> – ім'я індексу;
ACTIVE – перевести індекс зі стану INACTIVE до стану ACTIVE;
INACTIVE – перевести індекс зі стану ACTIVE до стану INACTIVE.

Приклади використання:

```
ALTER INDEX BUDGETX INACTIVE;
ALTER INDEX BUDGETX ACTIVE;
```

Оператор вилучення індексів DROP INDEX

Синтаксис:

```
DROP INDEX <name>
```

– вилучити індекс з ім'ям name.

Коли модифікують багато – декілька сотень (тисяч) – рядків і після модифікації кожного рядка перебудовуються всі індекси таблиці, час модифікації може бути на порядок (декілька порядків) більше від часу модифікації рядків з неіндексованими полями. Тому перед модифікацією множини рядків таблиці доцільно знищити індекси її полів. Це можна зробити за допомогою команди DROP INDEX (знищити індекс).

Оскільки індекси можуть створюватися або знищуватися у будь-який час, то перед виконанням запитів доцільно будувати індекси лише для тих полів, які використовуються у фразах запиту WHERE і ORDER BY, а перед модифікацією значної кількості рядків таблиць з індексованими полями ці індекси потрібно знищити. Знищення індексу не змінює значення поля.

10.3.6. Підтримка представлень

Представлення – це іменована таблиця, що визначається переліком тих полів таблиць і ознаками тих їхніх рядків, які хотілося б у ній побачити, тобто представлення – це віртуальна таблиця, яка сама по собі не існує, але для користувача виглядає так, наче вона існує.

Представлення є немов би «вікном» в одну або декілька базових таблиць. Воно створюється за допомогою команди CREATE VIEW (створити представлення), докладний опис якої наведено нижче.

Представлення не підтримується його власними даними, які зберігаються фізично. Замість цього у каталозі таблиць зберігається визначення, яке показує, з яких полів і рядків інших таблиць воно має бути сформоване. Формується представлення під час реалізації SQL-команди на отримання даних або на модифікацію таких даних.

Механізм представлень (*view*) є потужним засобом мови SQL, що дозволяє приховати реальну структуру БД від деяких користувачів за рахунок створення представлення БД. Представлення реально є деяким запитом, який зберігається у БД з іменованими полями, а для користувача нічим не відрізняється від базової таблиці БД (з урахуванням технічних обмежень). Будь-яка реалізація має гарантувати, що стан таблиці, яка представляється, точно відповідає стану базових таблиць, на яких визначено представлення. Зазвичай обчислення таблиці, яка представляється, (матеріалізація відповідного запиту) проводиться кожного разу під час використання представлення.

Оператор створення представлень CREATE VIEW

Синтаксис:

```
CREATE VIEW <name> [(<view_col> [, <view_col> ...])]
  AS <select> [WITH CHECK OPTION];
```

Опис параметрів:

<name>	–	унікальне ім'я представлення;
<view_col>	–	унікальне ім'я поля в представленні: Необхідне, коли поле обчислюють за виразом, інакше – беруть з базової таблиці;
<select>	–	визначення критерію відбору рядків представлення;
WITH CHECK OPTION	–	визначення перевірки нових записів на виконання умови, яку вказано в операнді WHERE.

Опція AS потрібна для запису вибірки (підзапиту). Для створення вибірки можна використовувати запит за декількома таблицями.

Необов'язкова фраза WITH CHECK OPTION визначає, чи можна вводити записи до представлення, які не задовольняють умови формування представлення, тобто для операцій INSERT і UPDATE над цим представленням має здійснюватися перевірка, яка забезпечує виконання фрази WHERE підзапиту. Якщо WITH CHECK OPTION у визначенні представлення не буде, такий контроль не проводиться.

Створена таблиця представлення *V* є змінною (тобто щодо *V* можна використати оператори DELETE і UPDATE) лише тоді, коли виконуються такі умови для специфікації запиту:

- у списку вибірки не вказано ключове слово DISTINCT;
- кожен арифметичний вираз у списку вибірки являє собою одну специфікацію поля, і специфікація одного поля не з'являється більше одного разу;
- у розділі FROM вказано тільки одну таблицю, що є або базовою, або змінною таблицею, яка представляється;
- в умові вибірки розділу WHERE не використовуються підзапити;
- у табличному виразі не буде розділів GROUP BY і HAVING.

Зауваження. Ці обмеження є дуже сильними. У реалізаціях вони можуть бути послаблені. Але якщо прагнути до мобільності, не слід користуватися розширеними можливостями.

Приклади використання:

```
    CREATE VIEW cust_zak AS
    SELECT customer.name1, customer.adres, custpmer.tel,
    zakaz.model
    FROM customer, zakaz
    WHERE customer.nom_pokup=zakaz.nom_pokup
CREATE VIEW SNOW_LINE (CITY, STATE, SNOW_ALTITUDE) AS
    SELECT CITY, STATE, ALTITUDE FROM CITIES
    WHERE ALTITUDE > 5000;
CREATE VIEW RECENT_CITIES AS
    SELECT STATE, CITY, POPULATION FROM CITIES WHERE
    STATE IN
    (SELECT STATE FROM STATES WHERE STATEHOOD >
    "1-JAN-1850");
CREATE VIEW HALF_MILE_CITIES AS
    SELECT CITY, STATE, ALTITUDE FROM CITIES
         WHERE ALTITUDE > 2500
    WITH CHECK OPTION;
```

Оператор вилучення представлення DROP VIEW

Синтаксис:

`DROP VIEW <name>` – вилучити представлення з іменем name.

Операції SELECT, DELETE, INSERT і UPDATE можуть оперувати не тільки базовими таблицями, але й представленнями. Але, якщо з базових таблиць можна видаляти будь-які рядки, поновлювати значення будь-яких їхніх полів і вводити у такі таблиці нові рядки, то цього не можна сказати про представлення, не всі з яких можуть поновлюватися.

Представлення, отримані з однієї базової таблиці (наприклад, шляхом вибору деяких її рядків і (або) полів), можна поновлювати.

Наприклад, можна використовувати такі оператори:

```
UPDATE cust_zak SET adres='Київ' WHERE adres='Полтава'
UPDATE cust_zak SET nom_prod=5 WHERE adres='Полтава'
```

але не можна виконати оператор:

```
UPDATE cust_zak SET adres='Київ', nom_prod=5 WHERE
adres='Полтава'
```

Модифікації, які змінюють дані у двох чи більше таблицях, неприпустимі. Якщо ж це потрібно, то використовують кілька команд модифікації, кожна з яких працює зі своїм об'єктом. Оскільки під час визначення представлення може бути використаний будь-який допустимий підзапит, то вибірка даних може здійснюватися як з базових таблиць, так і з представлень. За створеними представленнями можна робити запити.

Наприклад, одержати всю необхідну інформацію можна за допомогою єдиного запиту: `SELECT * FROM cust_zak`

Одне з основних завдань, яке дозволяє вирішувати представлення, – забезпечення незалежності програм користувача від зміни логічної структури БД у разі її розширення і (або) зміни розміщення полів, що виникає, наприклад, при розщепленні таблиць. В останньому випадку можна створити представлення-з'єднання з ім'ям і структурою розщепленої таблиці, що дозволить зберегти програми, які існували до зміни структури БД.

Крім того, представлення дають можливість різним користувачам по-різному отримувати однакові дані, можливо, навіть одночасно. Це є особливо цінним під час роботи різних категорій користувачів з єдиною інтегрованою БД. Користувачам надають тільки ті дані, які їх цікавлять, в найбільш зручній для них формі (вікно у таблицю або в будь-яке з'єднання будь-яких таблиць).

Нарешті, від певних користувачів можуть бути приховані деякі дані, невидимі через запропоноване їм представлення. Отже, примус користувача здійснювати доступ до БД через представлення є простим, але ефективним механізмом для керування санкціонуванням доступу.

10.3.7. Підтримка курсорів

Курсор – це засіб мови SQL, який дозволяє з допомогою набору спеціальних операторів отримати послідовний доступ до рядків віртуальної таблиці, отриманої у результаті SELECT-запиту до БД. Для користуванням курсором, його необхідно спочатку описати, тобто визначити множину рядків, які пов'язуються з курсором, а потім (у циклі) звертатися до рядків, які визначено під час оголошення курсора. Слід зауважити, що різні сервери реалізують різні варіанти оголошення та використання курсорів.

Оператор оголошення курсора DECLARE CURSOR

У реалізаціях вбудованої SQL-мови потрібно, щоб оператор оголошення курсора текстуально передував множині інших операторів, які використовують курсор.

Синтаксис:

```
DECLARE <cursor> CURSOR FOR <select> [FOR UPDATE
OF    <col> [,<col >...]];
```

Опис параметрів:

<cursor>	– ім'я курсора;
<select>	– оператор SELECT для вибору рядків, які будуть пов'язані з курсором. Він не може містити оператори INTO або ORDER BY;
FOR UPDATE OF	– визначення можливості редагування та вилучення рядків з таблиці за допомогою операторів UPDATE і DELETE;
<col>	– ім'я поля таблиці.

Приклади використання:

```
DECLARE C CURSOR FOR
  SELECT CUST_NO, ORDER_STATUS
  FROM SALES
     WHERE ORDER_STATUS IN ('open', 'shipping');
```

Оператор активізації курсора OPEN

Оператор відкриття курсора має бути першим у серії операторів, що використовують дані, зв'язані із заданим курсором. Під час виконання цього оператора проводиться підготовка курсора до роботи над ним. У більшості реалізацій у разі вбудованої SQL-мови саме виконання оператора відкриття курсора приводить до компіляції специфікації курсора і підготовки плану виконання запиту. Можна вважати, що під час виконання оператора відкриття курсора будується тимчасова віртуальна таблиця, яка містить результат запиту, зв'язаного з цим курсором.

Синтаксис:

```
OPEN <cursor name>
```

– активізувати курсор з іменем `cursor name`.

Наприклад, `OPEN C`

Наступні оператори можна виконувати над відкритим курсором у довільному порядку.

Оператор отримання чергового запису курсора FETCH

Якщо курсор установлено в позицію перед рядком, то під час виконання оператора читання він встановлюється на цей рядок, і значення рядка присвоюються відповідним змінним.

Синтаксис:

```
FETCH <cursor name> [INTO : <hostvar>
[[INDICATOR ] : <indvar>]
[, :<hostvar> [[INDICATOR ] : <indvar>]...]];
```

Опис параметрів:

`<cursor name>` – ім'я курсора;

`<hostvar>` – ім'я змінної, яка отримує значення поля з рядка курсора. Множина змінних відповідає множині полів курсора за кількістю, порядком і типом;

`<indvar>` – ім'я змінної (індикатора), яка отримує значення відповідно до значення поля курсора, а саме: від'ємне значення індикаторної змінної (воно має бути цілого типу) відповідає невизначеному значенню поля або значенню `NULL`.

Якщо таблиця, на яку вказує курсор, порожня або позиція курсора знаходиться на останньому рядку чи за ним, то під час виконання оператора `FETCH` курсор переміщується в позицію після останнього рядка. У цьому разі коду завершення оператора `FETCH` системній змінній SQLCODE присвоюється значення 100, а змінні, указані в розділі INTO, не отримують ніяких значень.

Оператор вилучення рядка таблиці, на який позиційовано курсор, DELETE FROM

Якщо вказаний в операторі курсор відкрито і встановлено на деякий рядок, то поточний рядок курсора видаляється з таблиці, а курсор встановлюється перед наступним рядком. Таблиця, указана в розділі `FROM` оператора `DELETE`, має бути таблицею, указаною в розділі `FROM` специфікації курсора.

Синтаксис:

```
DELETE FROM <table name> WHERE CURRENT OF <cursor name>
```

Опис параметрів:

<table name> – ім'я таблиці;
<cursor name> – ім'я курсора.

Оператор вилучення модифікації таблиці, на якій позиційовано курсор, UPDATE

Якщо вказаний в операторі курсор відкрито і встановлено на деякий рядок, то поточний рядок курсора модифікується відповідно до розділу SET. Позиція курсора не змінюється. Таблиця, указана в розділі FROM оператора DELETE, має бути таблицею, указаною в зовнішньому розділі FROM специфікації курсора.

Синтаксис:

```
UPDATE <table name> SET {<column name> |
<value expression> | NULL} [{,<column name> | <value
expression> | NULL}...] WHERE CURRENT OF <cursor name>
```

Опис параметрів:

<table name> – ім'я таблиці;
<column name> – ім'я поля;
<value expression> – вираз для обчислення значення;
<cursor name> – ім'я курсора.

Оператор закриття курсора CLOSE

Якщо під час виконання цього оператора курсор знаходиться у відкритому стані, то оператор переводить курсор у закритий стан. Після цього над курсором можливе виконання тільки оператора OPEN.

Синтаксис:

```
CLOSE <cursor name> – закрити курсор з іменем cursor name.
```

10.3.8. Підтримка збережених процедур

Збережена процедура – це окрема програма, написана спеціальною мовою ISPL (мова процедур і тригерів СКБД). Вона зберігається як частина метаданих на сервері. Збережені процедури можуть отримувати вхідні параметри від програм користувача та повертати їм вихідні параметри.

Існують два типи збережених процедур.

Пошукові процедури – процедури, які повертають результат пошуку записів у БД у вигляді нової таблиці. Поля таблиці визначаються значеннями вихідних параметрів за допомогою ключового слова RETURNS в оголошенні процедури. Ім'я такої процедури можна використовувати замість імені таблиці в SELECT – конструкціях мови SQL.

Процедури, що виконують дії, – процедури, які виконують обробку БД, але вони не мають повертати які-небудь значення в програму, що викликала таку процедуру.

Мова ISPL (мова для розробки процедур і тригерів) INTERBASE-сервера містить SQL-пропозиції обробки даних, а саме: INSERT, UPDATE, DELETE і тільки один SELECT.

Синтаксис вбудованого в INTERBASE оператора SELECT має додаткові елементи порівняно зі стандартним, зокрема:

- INTO :пар, :пар, ... – для занесення результатів пошуку в наведені у списку вихідні параметри;
- SQL-оператори та вирази для обробки UDF (функцій, що визначені користувачем) та використання генераторів (спеціальні змінні для формування унікальних значень), наприклад, оператор надання значення змінної за форматом: змінна = вираз;
- оператори керування ходом виконання, а саме:
  - /* текст */ – коментар;
  - BEGIN...END – операторні дужки;
  - IF...THEN...ELSE – умовне виконання блоків операторів;
  - WHILE...DO – створення циклів;
  - FOR <operator SELECT> DO <body cycle> – створення циклів для обробки вибраних записів;
  - EXECUTE PROCEDURE <name_proc> [var[,...]] RETURNIG VALUES var [,var...]] – виконання іншої збереженої процедури з іменем <name_proc>, яка отримує вхідні параметри, занесені в список після імені процедури, та повертає вихідні параметри, занесені в список після RETURNIG VALUES. Допускається вкладення процедур та їх рекурсивний виклик;
  - EXIT – перехід до завершального END у процедурі;
  - POST_EVENT <name> – емуляція події за наведеним іменем;
  - SUSPEND – оператор повернення вихідних значень (для пошукових процедур). Він припиняє виконання процедури, повертає визначені параметри та очікує подальшого звернення до процедури. У процедурах, що виконуються, його не застосовують;
- оператори обробки винятків:
  - EXEPTION <name> – активізує вказаний виняток. *Виняток* – це визначена користувачем помилкова ситуація, яка може бути оброблена в операторі WHEN. Винятки створює оператор CREATE EXCEPTION, у якому вказується назва винятку та його текстове ви-

значення. Наприклад: CREATE EXCEPTION CUSTOMER_ON_HOLD "This customer is on hold.". Для активізації винятку виконують оператор EXEPTION, наприклад,

```
/* This purchase order has been already shipped. */
IF (ord_stat = "shipped") THEN
BEGIN
  EXCEPTION order_already_shipped;
  SUSPEND;
END
/* Customer is on hold. */
ELSE IF (hold_stat = "*") THEN
BEGIN
  EXCEPTION customer_on_hold;
  SUSPEND;
END;
```

- оператори оброблення помилок:
  - WHEN {[error[,<error...>]] ! ANY} DO <operators> – у разі, коли виникає вказана помилка, виконується складений оператор, вказаний за DO. Цей оператор має знаходитися в кінці процедури перед END.

Збережена процедура складається із заголовка процедури та її тіла.

Заголовок процедури містить:

- ім'я процедури;
- необов'язковий перелік вхідних параметрів;
- оператор RETURNS з переліком параметрів, що повертаються.

Тіло процедури містить:

- необов'язковий перелік локальних змінних процедури та опис їх типів;
- блок операторів мови ISPL в операторних дужках BEGIN. . . END.

Оператор створення збереженої процедури CREATE PROCEDURE

Синтаксис:

```
CREATE PROCEDURE <name>
[( <param> <datatype> [, <param> <datatype> ...])]
[RETURNS <datatype> [, <param> <datatype> ...])]
AS <procedure_body> [<terminator>]
<procedure_body> =
[ <variable_declaration_list>]
<block>
```

```
<variable_declaration_list> =
DECLARE VARIABLE <var> <datatype>;
[DECLARE VARIABLE <var> <datatype>; …]
< block> =
BEGIN
<compound_statement>
[ <compound_statement> …]
END;

<compound_statement> = {< block> | <statement>;}
< datatype> = {
{SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION}
| {DECIMAL | NUMERIC} [( <precision> [, <scale>])]
| DATE
| {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}
[( int)]
[CHARACTER SET <charname>]
| {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
[VARYING] [( int)]}
```

Опис параметрів:

<name>	–	унікальне ім'я процедури серед імен таблиць і представлень;
<param>	–	вхідні параметри, які програма, що викликає процедуру, використовує для передачі параметрів процедурі;
<datatype>	–	тип даних;
RETURNS <param> <datatype>	–	вихідні параметри, які процедура повертає програмі, що її викликала;
<datatype>:	–	дозволений сервером тип даних. Процедура повертає значення вихідних параметрів під час виконання оператора SUSPEND;
AS	–	ключове слово, яке відокремлює тіло та заголовок процедури;
DECLARE VARIABLE <var> <datatype>	–	оголошення локальних змінних процедури.

Приклад використання:

```
CREATE PROCEDURE sub_tot_budget (head_dept CHAR(3))
RETURNS (tot_budget DECIMAL(12, 2), avg_budget
DECIMAL(12, 2),
      min_budget DECIMAL(12, 2), max_budget
      DECIMAL(12, 2))
AS
BEGIN
    SELECT SUM(budget), AVG(budget), MIN(budget),
    MAX(budget)
        FROM department
        WHERE head_dept = :head_dept
INTO :tot_budget, :avg_budget, :min_budget, :max_budget;
            EXIT;
END;
```

Оператор модифікації збереженої процедури ALTER PROCEDURE

Синтаксис:

```
ALTER PROCEDURE <name>
[(<param> <datatype> [, <param> <datatype> ...])]
 [RETURNS (<param> <datatype> [, <param>
 <datatype> ...])]
 AS <procedure_body>
```

Опис параметрів:

<name>	–	ім'я існуючої збереженої процедури;
<param> <datatype>	–	перелік вхідних параметрів процедури. Типи параметрів мають відповідати списку параметрів, який визначено в операторі CREATE PROCEDURE;
RETURNS <param> <datatype>	–	перелік вихідних параметрів процедури. Типи параметрів мають відповідати списку параметрів, який визначено в операторі CREATE PROCEDURE;
<procedure_body>	–	тіло збереженої процедури. Складається з визначення локальних змінних і блоку операторів ISPL.

Приклад використання:

```
ALTER PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT)
RETURNS (PROJ_ID VARCHAR(20)) AS
```

```
BEGIN
      FOR SELECT PROJ_ID
      FROM EMPLOYEE_PROJECT
      WHERE EMP_NO = :emp_no
      INTO :proj_id
      DO
      SUSPEND;
END;
```

Оператор вилучення збереженої пропедури DROP PROCEDURE
Синтаксис:

```
DROP PROCEDURE <name>
```
– вилучити існуючу збережену процедуру з ім'ям name.

Оператор виконання збереженої процедури EXECUTE PROCEDURE
Синтаксис:

```
EXECUTE PROCEDURE <name> [<param> [, <param> ...]];
```

Опис параметрів:

<name> – ім'я збереженої процедури;
<param> – вхідні параметри (константи).

10.3.9. Підтримка тригерів

Тригер – це програмний блок, асоційований з таблицею БД, який автоматично виконує вказані в ньому дії, коли над зв'язаною з ним таблицею БД було здійснено окремо визначену подію.

З тригерами можна зв'язати такі події:

- доповнення нового запису;
- зміна існуючого запису;
- вилучення запису.

Виклик тригера для виконання можна визначити перед здійсненням або після здійснення події.

Тригер ніколи не викликають безпосередньо з програми користувача. Він активізується автоматично, якщо користувач виконав такі дії над таблицею, які передбачено в умовах активізації тригера.

Використання тригерів дає можливість:

- реалізації автоматичної перевірки правильності даних, які вносять у БД;
- реалізації обмежень цілісності даних;
- забезпечення компактності програм обробки таблиць БД;

- підтримки ведення таблиці реєстрації змін у БД;
- попередження про всі зміни в таблицях БД.

Тригер складається із заголовка та тіла.

Заголовок тригера містить:

- ім'я тригера, унікальне ім'я в межах БД серед імен таблиць, тригерів, збережених процедур і представлень;
- ім'я таблиці, для якої створюється тригер;
- умови, за яких активізується тригер.

Тіло тригера містить:

- необов'язковий список визначень локальних змінних (імена змінних та їх типи);
- блок операторів мови ISPL, який починається з BEGIN та завершується END.

Оператори цього блоку виконуються під час активізації тригера.

Оператор створення тригера CREATE TRIGGER

Синтаксис:

```
CREATE TRIGGER <name> FOR <tablename>
[ ACTIVE | INACTIVE]
{BEFOR | AFTER} { DELETE | INSERT | UPDATE }
[ POSITION <number>]
AS
<trigger body>
<trigger body> = [ <variable_declaration _list>] <block>
  <variable_declaration_list> =
DECLARE VARIABLE <variable> <data type>;
[DECLARE VARIABLE <variable> <data_type>;...]
<block> =
BEGIN
  <compound_statement>
[<compound_statement>...]
END
<compound_statement>= { <block> <statement> ;}
<data_type> = {
SMALLINT | INTEGER I FLOAT| DOUBLE PRECISION }
| { DECIMAL | NUMERIC } [ <precision> [, <scale>]])
| DATE
|{ CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}
```

```
[(int)]
  [ CHARACTER SET <charname>]
| { NCHAR | NATIONAL CHARACTER |NATIONAL CHAR}
  [ VARYING][(int)]}
```

Опис основних параметрів:

<name>	– ім'я тригера;
<tablename>	– ім'я таблиці, з якою зв'язано тригер;
ACTIVE	– визначає активність тригера;
INACTIVE	– визначає неактивність тригера;
BEFOR	– визначає, що тригер виконується перед асоційованою операцією;
AFTER	– визначає, що тригер виконується після асоційованої операції;
DELETE|INSERT|UPDATE	– визначає асоційовану з тригером операцію;
POSITION	– визначає номер активізації тригера в ряду декількох тригерів, тригер з найменшим номером активізується першим. Номер набуває значень від 0 до 32767;
DECLARE VARIABLE var <datatype>	– визначає локальні змінні, кожну змінну відокремлюють символом «;»;
<datatype>	– визначає тип змінної.

Усередині тіла тригера доступні для використання спеціальні змінні, імена яких відповідають формату NEW. <column> та OLD. <column>, де <column> – ім'я поля в таблиці. Ці імена дають змогу звертатися до нового (NEW. <column>) значення поля таблиці з іменем <column> та його попереднього (OLD. <column>) значення.

Приклад використання:

```
CREATE TRIGGER SAVE_SALARY_CHANGE FOR EMPLOYEE
AFTER UPDATE AS
BEGIN     IF (OLD.SALARY <> NEW.SALARY) THEN
      INSERT INTO SALARY_HISTORY
           (EMP_NO, CHANGE_DATE, UPDATER_ID, OLD_SALARY,
PERCENT_CHANGE)
      VALUES (OLD.EMP_NO, "now", USER, OLD.SALARY,
      (NEW.SALARY - OLD.SALARY) * 100 / OLD.SALARY);
END;
```

Оператор модифікації тригера ALTER TRIGGER

Синтаксис:

```
ALTER TRIGGER <name>
[ACTIVE | INACTIVE]
 [{BEFORE | AFTER} {DELETE | INSERT | UPDATE}]
 [POSITION <number>]
 [AS <trigger_body>]
```

Опис параметрів:

<name>	– ім'я існуючого тригера;
ACTIVE	– опція, що визначає активність тригера;
INACTIVE	– опція, що визначає неактивність тригера;
BEFORE	– визначає, що тригер виконується перед асоційованою операцією;
AFTER	– визначає, що тригер виконується після асоційованої операції;
DELETE|INSERT|UPDATE	– визначає асоційовану з тригером операцію;
POSITION <number>	– визначає номер активізації тригера в ряду декількох тригерів, тригер з найменшим номером активізується першим. Номер набуває значень від 0 до 32767;
<trigger_body>	– тіло тригера за правилами, визначеними в операторі CREATE TRIGGER.

Приклад використання:

```
ALTER TRIGGER SET_CUST_NO FOR CUSTOMER
BEFORE INSERT AS
    BEGIN
        NEW.CUST_NO = GEN_ID(CUST_NO_GEN, 1);
        INSERT INTO NEW_CUSTOMERS(NEW.CUST_NO, TODAY)
    END;
```

Оператор вилучення тригера DROP TRIGGER

Синтаксис:

```
DROP TRIGGER <name>
```

– вилучити існуючу збережену процедуру з іменем name.

10.3.10. Підтримка обробки особливих ситуацій (винятків)

Користувачі серверів БД здебільшого мають змогу передбачити створення та обробку виняткових ситуацій, визначених саме користувачем, з якими він поєднує формування відповідного повідомлення. Винятки (exceptions) використовують у збережених процедурах і тригерах.

Винятки є глобальними у БД. Повідомлення або набір повідомлень, що їм відповідають, доступні для всіх збережених процедур і тригерів у додатку. Коли тригер або збережена процедура активізують виняток, то:

- припиняється виконання тригера або процедури і знищуються будь-які дії, які було виконано;
- формується повідомлення, яке відповідає винятку.

Винятки можуть бути створені й оброблені за допомогою оператора WHEN у збереженій процедурі або тригері.

Оператор створення винятків CREATE EXCEPTION

Синтаксис:

```
CREATE EXCEPTION <name> "<message>";
```

Опис параметрів:

<name>	–	унікальне ім'я винятку;
"<message>"	–	текст повідомлення максимальною довжиною 78 символів.

Приклад використання:

```
CREATE EXCEPTION UNKNOWN_EMP_ID
"Invalid employee number or project id.";
```

Оператор активізації винятку EXCEPTION

Синтаксис:

```
EXCEPTION <name> - активізувати виняток з іменем name.
```

Приклад використання:

```
WHEN  SQLCODE -530 DO
EXCEPTION UNKNOWN_EMP_ID;
```

Оператор модифікації винятку ALTER EXCEPTION

Синтаксис:

```
ALTER EXCEPTION <name> "<message>";
```

Опис параметрів:

<name>	–	назва існуючого винятку;
"<message>"	–	текстове повідомлення.

Приклад використання:

```
ALTER EXCEPTION CUSTOMER_CHECK "Hold shipment for
customer remittance.";
```

Оператор вилучення винятку DROP EXCEPTION

Синтаксис:

```
DROP EXCEPTION <name> "<message>" – вилучити виняток з іме-
                                    нем name.
```

10.3.11. Підтримка генераторів

Користувачі мають можливість оголосити генератор БД (спеціальну змінну, яка підтримується сервером БД, використовується для отримання унікальних значень) і встановити її початкове значення. Поточне значення генератору повертає функція `GEN_ID()`. Воно є, наприклад, послідовним номером, який можна присвоїти значенню поля таблиці. Генератор часто використовують, щоб гарантувати унікальне значення в `PRIMARY KEY`, як, наприклад, номер рахунка-фактури. БД може містити набір генераторів. Генератори є глобальними у БД. Їх значення можна використовувати і змінювати під час оброблення запиту.

Оператор створення генераторів CREATE GENERATOR

Синтаксис:

```
CREATE GENERATOR <name> – створити генератор з іменем name.
```

Оператор установлення значення генератору SET GENERATOR

Після того, як генератор створено, `SET GENERATOR` може встановити або змінити поточне значення. Генератор задач можна використовувати під час розробки тригера, процедури або твердження SQL, яке викликає `GEN_ID()`.

Синтаксис:

```
SET GENERATOR <name> TO int;
```

Опис аргументів:

`<name>` – ім'я існуючого генератора;

`int` – значення генератора, ціле від мінус 231 до 231.

Використання функції GEN_ID()

```
GEN_ID (<generator>, <step>);
```

Опис параметрів:

`<generator>` – ім'я існуючого генератора;

`<step>` – крок (ціле або вираз). Поточне значення генератора повертається як результат функції, а значення кроку додається до поточного значення генератора. Значення кроку може бути від – 231 до 231.

Приклад використання генератора у тілі тригера:

```
CREATE GENERATOR EMP_NO_GEN;
SET GENERATOR EMP_NO_GEN TO 1000;
CREATE TRIGGER CREATE_EMPNO FOR EMPLOYEES
 BEFORE INSERT
     POSITION 0
 AS BEGIN
     NEW.EMPNO = GEN_ID(EMPNO_GEN, 1);
 END;
```

10.3.12. Підтримка фільтрів

Для спеціальної обробки даних типу BLOB можна встановити BLOB-фільтри, тобто визначену користувачем програму, яка перетворює дані поля BLOB на дані інших типів. Типи даних встановлює користувач. Розрізняють вхідний (INPUT_TYPE) та вихідний (OUTPUT_TYPE) типи. Разом вони визначають режим фільтра BLOB. Кожний фільтр, який оголошено у БД, повинен мати унікальну комбінацію цілих значень типів INPUT_TYPE і OUTPUT_TYPE. Визначені користувачем типи слід виразити як від'ємні значення.

Оператор визначення фільтра DECLARE FILTER

Синтаксис:

```
DECLARE FILTER <filter>
INPUT_TYPE <subtype> OUTPUT_TYPE <subtype>
ENTRY_POINT "<entryname>" MODULE_NAME "<modulename>";
```

Опис параметрів:

`<filter>`	–	унікальне ім'я фільтра;
`INPUT_TYPE <subtype>`	–	визначення вхідного типу;
`OUTPUT_TYPE <subtype>`	–	визначення вихідного типу;
`"<entryname>"`	–	літерний рядок, який визначає назву фільтра у бібліотеці. Якщо використовують BLOB-фільтр, то його пошук здійснюють за цим іменем;
`"<modulename>"`	–	літерний рядок, який визначає назву модуля, в якому збережено фільтр.

Приклад використання:

```
DECLARE FILTER DESC_FILTER
```

```
INPUT_TYPE 1
OUTPUT_TYPE -4
ENTRY_POINT "desc_filter"
MODULE_NAME "FILTERLIB";
```

Оператор вилучення фільтра DROP FILTER

Синтаксис:

```
DROP FILTER <name> – вилучити фільтр з іменем name.
```

10.3.13. Підтримка зовнішніх функцій користувача

Додатково до функцій, які виконує сервер БД, користувач має змогу визначити власні функції під загальною назвою UDF (User Defined Functions). Їх зберігають поза межами БД і використовують за бажанням користувачів.

Оператор визначення UDF DECLARE EXTERNAL FUNCTION

Синтаксис:

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

Опис параметрів:

`<name>`	– ім'я UDF;
`<datatype>`	– типи вхідних або вихідних параметрів. Усі вхідні параметри передаються до UDF за посиланням. Зворотні параметри можуть бути передані за значенням;
`RETURNS`	– опис значення, яке повертає функція;
`BY VALUE`	– вказує, що зворотне значення має бути передане значенням скоріше, ніж звертанням;
`CSTRING (int)`	– вказує, що UDF повертає *null-terminated*-рядки довжиною в `int` байтів;
`"<entryname>"`	– рядок, який визначає ім'я UDF, під яким вона збережена в бібліотеці UDF;
`"<modulename>"`	– рядок, що визначає ім'я модуля, в якому перебуває UDF.

Приклад використання:

```
DECLARE EXTERNAL FUNCTION TOPS
            CHAR(256), INTEGER, BLOB
```

```
RETURNS INTEGER BY VALUE
ENTRY_POINT "te1" MODULE_NAME "tm1";
```

Оператор вилучення UDF DROP EXTERNAL FUNCTION
Синтаксис:

```
DROP EXTERNAL FUNCTION <name> - вилучити UDF з іменем name.
```

10.3.14. Підтримка системи захисту

Адміністратори БД мають змогу визначити для користувачів або для інших даних об'єктів БД права для обробки даних. Право доступу до створеного об'єкта спочатку має тільки той, хто його створив, і тільки він може визначати права доступу до нього для інших користувачів або об'єктів.

Для отримання доступу до таблиці або представлення, користувач або об'єкт повинен мати одне (або кілька) з прав доступу до них. Існують права: SELECT, INSERT, UPDATE, DELETE або EXECUTE.

Опис прав користувача:

ALL	– право виконувати оператори SELECT, DELETE, INSERT, UPDATE і EXECUTE;
SELECT	– право виконувати оператори SELECT для таблиць або представлень;
DELETE	– право вилучати рядки таблиць або представлень;
INSERT	– право додавати нові рядки в таблиці або представлення;
EXECUTE	– право виконувати збережену процедуру.

Оператор визначення прав GRANT

Синтаксис:

```
GRANT{
{ALL [PRIVILEGES] | SELECT | DELETE | INSERT
  | UPDATE [(<col> [, <col> ...])]}
      ON [TABLE] {<tablename> | <viewname>}
      TO {<object> | <userlist>}
  | EXECUTE ON PROCEDURE <procname>
      TO {<object> | <userlist>}
  };
<object> = PROCEDURE <procname> | TRIGGER <trigname>
| VIEW <viewname>
| [USER] <username> | PUBLIC [, <object>]
<userlist> = [USER] <username> [, [USER] <username> ...]
[WITH GRANT OPTION]
```

Опис параметрів:

<col>	– поле таблиці, для якого визначаються права;
<tablename>	– ім'я таблиці;
<viewname>	– ім'я представлення;
<object>	– ім'я користувача або об'єкта;
<userlist>	– список користувачів або об'єктів;
WITH GRANT OPTION	– визначення можливості визначати права для інших об'єктів.

Приклади використання:

```
GRANT SELECT, DELETE ON COUNTRY TO CHLOE WITH GRANT
            OPTION;
GRANT EXECUTE ON PROCEDURE GET_EMP_PROJ
            TO PROCEDURE ADD_EMP_PROJ, LUIS;
```

Оператор вилучення прав REVOKE

Права можуть бути вилучені тільки користувачем, який їх визначив.
Синтаксис:

```
REVOKE [GRANT OPTION FOR]{
{ALL [PRIVILEGES] | SELECT | DELETE | INSERT
                  | UPDATE [(<col> [, <col> ...])]}
                  ON [TABLE] {<tablename> | <viewname>}
                  FROM {<object> | <userlist>}
                | EXECUTE ON PROCEDURE <procname>
                  FROM {<object> | <userlist>}
                };
                <object> = PROCEDURE <procname> |
                TRIGGER <trigname> | VIEW <viewname>
                | [USER] <username> | PUBLIC
                [, <object>]
                <userlist> = [USER] <username>
                [, [USER] <username> ...]
```

Опис параметрів:

GRANT OPTION FOR	– визначення прав, що вилучають зі списку <userlist>. Не можна вказувати з <object>;
<col>	– поле, для якого було надано права;
<tablename>	– назва таблиці, для якої було встановлено права;

<viewname>	– назва представлення, для якого було призначено права;
<object>	– ім'я користувача або об'єкта, для якого вилучаються права;
<userlist>	– список користувачів, для яких вилучаються права.

Приклад використання:

```
REVOKE EXECUTE ON PROCEDURE GET_EMP_PROJ
 FROM PROCEDURE ADD_EMP_PROJ, LUIS;
```

10.4. Формування запитів до бази даних

10.4.1. Оператор SELECT

Формування та виконання запитів до БД здійснюють за допомогою оператора SELECT.

Основу оператора складають ключові слова (специфікатори): DISTINCT, FROM, WHERE, GROUP BY, HAVING та інші. Результатом виконання оператора є таблиця-відповідь, структуру та вміст якої визначають специфікаторами оператора SELECT. Запит можна записати за загальним шаблоном:

SELECT <list_fields>	– визначення полів з таблиць БД, які стануть полями таблиці-відповіді;
FROM <list_tables>	– визначення таблиць БД, з яких отримають значення полів таблиці-відповіді;
[WHERE <search_condition>]	– визначення умов для відбору рядків таблиць БД, з яких отримають значення полів таблиці-відповіді.

Наприклад:

```
SELECT name_customer
FROM customer
WHERE key_customer=3 ;
```

Запит означає: сформувати таблицю-відповідь, до якої входить поле name_customer; значення поля вибирають з поля name_customer записів таблиці customer, але тільки тих записів, в яких поле key_customer має значення 3.

На практиці SQL-вираз для вибірки може бути досить складним.

Синтаксис:

```
SELECT [TRANSACTION <transaction>]
   [DISTINCT | ALL] {* | <val> [, <val> ...]}
   [INTO : var [, : var ...]]
```

```
FROM <tableref> [, <tableref> ...]
[WHERE <search_condition>]
[GROUP BY <col> [COLLATE <collation>] [, <col>
[COLLATE <collation>] ...]
[HAVING <search_condition>]
[UNION <select_expr> [ALL]]
[PLAN <plan_expr>]
[ORDER BY <order_list>]
[FOR UPDATE [OF <col> [, <col> …]]];
<val> = {
<col> [<array_dim>] | <constant> | <expr> |
<function>
| NULL | USER | RDB$DB_KEY
}
<array_dim> = [x:y [, x:y ...]]
```

Примітка. Квадратні дужки необхідно включати да визначення масиву.

```
<constant> = num | "string" | charsetname "string"
<expr> = A valid SQL expression that results in a
single value.
<function> = {
COUNT (* | [ALL] <val> | DISTINCT <val>)
  | SUM ([ALL] <val> | DISTINCT <val>)
  | AVG ([ALL] <val> | DISTINCT <val>)
  | MAX ([ALL] <val> | DISTINCT <val>)
  | MIN ([ALL] <val> | DISTINCT <val>)
  | CAST (<val> AS <datatype>)
  | UPPER (<val>)
  | GEN_ID (generator, <val>)
  }
<tableref> = <joined_table> | <table> | <view> |
<procedure>
[(<val> [, <val> ...])] [<alias>]
<joined_table> = <tableref> <join_type> JOIN <tableref>
ON <search_condition> | (<joined_table>)
<join-type> = {[INNER] | {LEFT | RIGHT | FULL }
[OUTER]} JOIN
<search_condition> = {<val> <operator>
{<val> | (<select_one>)}
  | <val> [NOT] BETWEEN <val> AND <val>
  | <val> [NOT] LIKE <val> [ESCAPE <val>]
  | <val> [NOT] IN (<val> [, <val> ...] | <select_list>)
```

```
  | <val> IS [NOT] NULL
  | <val> {[NOT] {= | < | >} | >= | <=}
      {ALL | SOME | ANY} (<select_list>)
  | EXISTS (<select_expr>)
  | SINGULAR (<select_expr>)
  | <val> [NOT] CONTAINING <val>
  | <val> [NOT] STARTING [WITH] <val>
  | (<search_condition>)
  | NOT <search_condition>
  | <search_condition> OR <search_condition>
  | <search_condition> AND <search_condition>)

<operator> = {= | < | > | <= | >= | !< | !> | <> | !=}
<select_one> = SELECT on a single column that returns
exactly one row.
<select_list> = SELECT on a single column that returns
zero or more rows.
<select_expr> = SELECT on a list of values that returns
zero or more rows.

<plan_expr> =
[JOIN | [SORT] MERGE] (<plan_item> | <plan_expr>
     [, <plan_item> | <plan_expr> ...])
 <plan_item> = {<table> | <alias>}
 NATURAL | INDEX (<index> [, <index> ...]) | ORDER
 <index>
<order_list> =
{<col> | int} [COLLATE <collation>] [ASC[ENDING] |
DESC[ENDING]]
 [, <order_list>]
```

Опис параметрів:

TRANSACTION <transaction>	– ім'я транзакції, під керуванням якої виконується оператор SELECT;
SELECT [DISTINCT | ALL]	– визначення режиму обрання значень; DISTINCT – заборона появи рядків, що дублюються, в таблиці-відповіді; ALL (за замовчуванням) – повернути всі рядки незалежно від дублювання значень.

{* | <val> [, <val> …]	– визначення полів для внесення в таблицю-відповідь, де * – внести всі поля визначених таблиць; <val> [, <val> ...] – перелік імен полів визначених таблиць, які необхідно внести до таблиці-відповіді;
[INTO : var [, : var …]]	– визначення переліку змінних збережених процедур чи тригерів або параметрів динамічних SQL-запитів для присвоєння значень полів одного рядка таблиці-відповіді;
FROM <tableref> [, <tableref> …]	– список таблиць, представлень або збережених процедур, з яких отримують значення полів таблиці-відповіді. Список може містити об'єднання, які можуть бути вкладеними;
<table>	– ім'я таблиці БД;
<view>	– ім'я представлення БД;
<procedure>	– ім'я збереженої процедури, яка повертає таблицю-відповідь;
<alias>	– альтернативне ім'я (псевдонім) таблиці або представлення, яке можна використовувати в тексті запиту замість основного імені таблиці;
<joined_table>	– визначення об'єднання таблиць, до якого надходить ключове слово JOIN;
<join_type>	– тип об'єднання. За замовчуванням: INNER (внутрішнє);
WHERE <search_cond>	– визначення умови пошуку, яка обмежує множину рядків таблиці-відповіді;

`GROUP BY <col> [, <col> …]`	– визначення групування таблиці-відповіді на групи рядків, які мають ідентичні значення вказаних полів. Кожна група представлена в таблиці-відповіді одним рядком. Для груп часто використовують функції агрегування (обчислення спеціальних значень для групи рядків);
`COLLATE <collation>`	– визначення порядку порівняння значень, що повертаються;
`HAVING <search_cond>`	– визначення додаткової умови для відбору рядків, які вносять до групи записів таблиці-відповіді. Використовується з `GROUP BY`;
`UNION`	– засіб для об'єднання результатів даного запиту з іншим в одну таблицю-відповідь. За `UNION` наводиться оператор `SELECT`, який визначає другий запит;
`PLAN <plan_expr>`	– визначення плану пошуку для оптимізатора виконання запиту;
`<plan_item>`	– визначення списку таблиць та індексів, які необхідно використати для створення оптимального плану пошуку відповіді на запит;
`ORDER BY <order_list>`	– визначення списку полів, за яким необхідно сортувати рядки таблиці-відповіді;
`[FOR UPDATE [OF col [, <col> …]]];`	– перелік полів у специфікаторі `SELECT` оператора `DECLARE CURSOR`, які будуть поновлюватися за допомогою специфікатора `WHERE CURRENT OF`.

У подальших розділах специфікатори оператора `SELECT` розглядатимемо більш детально.

Визначення заголовків полів. За замовчуванням заголовками полів у таблиці-відповіді будуть імена саме тих полів, на які здійснює запити

оператор SELECT (вони перелічені після ключового слова SELECT). Якщо потрібно змінити назву поля, то необхідно вказати заголовок поля після його імені через пробіл або ключове слово AS, тобто:

```
SELECT <ім`я поля> [AS] <заголовок поля> ...
FROM <ім`я таблиці>
```

Наприклад:

```
SELECT name_model Модель,
     top_speed AS Макс_швидкість, "км/год" Од_виміру
FROM model
```

Результат:

Модель	Макс_швидкість	Од_виміру
145 1.4	178.0	км/год
740 i 4.0	250.0	км/год

У деяких версіях сервера підтримується тільки англомовний синтаксис для визначення назв полів.

Якщо значення полів таблиці визначає генератор, то під час виконання запиту можна змінити значення генератора для вибраних рядків за допомогою функції GEN_ID ().

Синтаксис:

```
GEN_ID (generator, <val>)
```

Опис параметрів:

generator – ім'я існуючого генератора;

<val> – крок, з яким змінюється поточне значення генератора.

Наприклад, оператор SELECT gen_id(EMP_NO_GEN,2) FROM employee ORDER BY last_name змінює значення генератора на нові з кроком 2.

До всіх числових полів можна застосовувати арифметичні операції (додавання, віднімання, множення, ділення) та арифметичні функції (ABS, COS, SIN, EXP, FLOOR, LOG, LOG10, ROUND, SIGN, SQRT).

Наприклад, запит з використанням функції ROUND має такий вигляд:

```
SELECT Account Номер, Sum Сума, ROUND(sum*0.22)
Сума_ндс
FROM example
```

Аналогічно можна застосовувати символьні функції до символьних змінних, функції роботи з датами – до змінних типу DATE та ін.

Іноді необхідно перетворювати одні типи даних на інші. Для цього використовують відповідні функції, наприклад: CONVERT, STYLE.

Можна використовувати спеціальні ключові слова, для для визначення значення поля.

NULL	–	додати поле зі значенням NULL до кожного рядка таблиці-відповіді;
USER	–	додати поле з іменем власника таблиці до кожного рядка таблиці-відповіді;
RDB$DB_KEY	–	додати поле з базовим номером до кожного рядка таблиці-відповіді.

Вилучення дублікатів. Для вилучення дублікатів (однакових рядків) у таблиці-відповіді до запиту необхідно додати ключове слово DISTINCT (різноманітний, різний).

Наприклад, для визначення продавців, які мають замовлення у таблиці *Orders* без дублювання прізвищ (кількість замовлень несуттєва, а потрібний лише список продавців), необхідно сформувати запит:

```
SELECT DISTINCT Fio FROM Orders
```

Цей засіб доцільно використовувати для виключення надлишкових даних.

Альтернативою DISTINCT є ALL. ALL вказує на те, що в результат вибірки (тобто до таблиці-відповіді) включають всі рядки (навіть ті, що дублюються).

10.4.2. Специфікатор FROM

Специфікатор FROM вказує таблиці БД, на базі яких формується таблиця-відповідь.

Загальний синтаксис специфікатора:

```
FROM <джерело> [, <джерело> ...]
<джерело> = <приєднана таблиця>|<таблиця>|<представлення>|
<зб.процедура>[(<парам> [, <парам> ...])] [аліас]
<приєднана таблиця> = <джерело> <тип приєднання> JOIN
<джерело>
ON <умова зв'язування> | (<приєднана таблиця >)
<тип приєднання> = {[INNER] | {LEFT | RIGHT | FULL }
[OUTER]} JOIN
```

Результатом виконання запиту зі специфікатором FROM є розширений декартовий добуток таблиць, заданих списком таблиць специфікатора FROM. Розширений декартовий добуток визначають так: «Розширений добуток R є мультимножиною всіх рядків r, таких, що r є конкатенацією рядків з усіх ідентифікованих таблиць у тому порядку, у якому їх ідентифіковано. Потужність R є добутком потужностей ідентифікованих таблиць. Порядковий номер поля в R є $n + s$, де n – порядковий номер породжуючого поля в іменованій таблиці T, а s – сума степенів усіх таблиць, ідентифікованих до T у специфікаторі FROM».

Поруч з іменем таблиці можна вказувати ще одне ім'я – аліас (псевдонім). Фактично, аліас – це синонім імені таблиці, який можна використовувати в інших специфікаторах оператора SELECT замість імені цієї таблиці. У цьому разі звернення до поля формується за синтаксисом <аліас.ім'я_поля>. Одна і та сама таблиця може декілька разів брати участь у списку одного специфікатора FROM і входити у списки специфікаторів FROM декількох підзапитів.

Приклади використання:

```
SELECT name1, name2, name3 FROM customer
```
– вносяться до таблиці-відповіді) поля name1, name2, name3 з таблиці customer;

```
SELECT * FROM customer
```
– виводяться всі поля з таблиці customer;

```
SELECT firm.name, countre.* FROM firm, countri
```
– виводиться поле name з таблиці firm і всі поля з таблиці countri.

Для того щоб результати вибірки були зрозумілими для користувача, у рядки таблиці-відповіді можна включити поля-назви:

```
SELECT name_model, 'потужність (к.с.)',
capacity FROM model
```

Результат вибірки:

```
name_model                    capacity
 145 1.4   потужність (к.с.)    90.0
 740 4.0   потужність (к.с.)    286.0
```

У специфікаторі FROM можна встановити запит на об'єднання між таблицями за допомогою специфікаторів JOIN або WHERE.

10.4.3. Об'єднання таблиць

Об'єднання між таблицями за допомогою специфікатора WHERE. Результат об'єднання – це віртуальна таблиця, яку будують у два етапи: спочатку створюють декартовий добуток таблиць, а потім з результату вибирають рядки відповідно до умови специфікатора WHERE.

Приклад використання:

```
SELECT m.name_model, a.cost
FROM model m, automobile a
WHERE m.key_model=a.key_model
```

Об'єднання між таблицями за допомогою специфікатора JOIN. Результат об'єднання – віртуальна таблиця, яку будують у два етапи: спочатку створюють декартовий добуток таблиць, а потім з результату вибирають рядки відповідно до умови додаткового специфікатора об'єднання ON. Розрізняють зовнішні та внутрішні об'єднання.

Внутрішнє об'єднання.

Синтаксис:

```
<ліва таблиця> INNER JOIN <права таблиця> ON <умова об'єднання>.
```

Існують три типи внутрішнього об'єднання, а саме:

- еквіоб'єднання (об'єднання за еквівалентністю) – об'єднання таблиць *T1* і *T2* з умовою за форматом: T1.поле1 = T2.поле2

  Наприклад:

```
SELECT m.name_model, a.cost
FROM model m INNER JOIN automobile a ON m.key_model = a.key_model;
```

- порівняльне об'єднання (об'єднання за виразом порівняння) – об'єднання таблиць *T1* і *T2* з умовою за форматом: T1.поле1 <вираз_порівняння >T2.поле2.

  Наприклад:

```
SELECT m.cost, a.cost
FROM model m INNER JOIN automobile a ON m.cost > a.cost
```

- самооб'єднання (об'єднання таблиці із собою) – об'єднання таблиць *T1* і *T1* з умовою за форматом: T1.поле1 = T1.поле2

  Наприклад:

```
SELECT r1.river, r2.river, r2.source
FROM rivers r1 JOIN rivers r2
ON r2.outflow = r1.river
ORDER BY r1.river, r2.source
```

Зовнішнє об'єднання. Існують три типи зовнішнього об'єднання:

- зовнішнє об'єднання з лівого боку. Синтаксис:

```
<ліва таблиця> LEFT OUTER JOIN <права таблиця> ON <умова об'єднання>
```

Для об'єднання вибирають усі рядки лівої таблиці та поєднують або з тими рядками правої таблиці, які задовольняють умову об'єднання, або з NULL-значенням (у разі, коли не виконується умова об'єднання). Алгоритм об'єднання містить такі кроки:

1. Обирається перший рядок лівої таблиці.

2. Створюється множина пар: обраний рядок у лівій таблиці зчеплюється з усіма відповідними рядками правої таблиці (кількість пар дорівнює кількості рядків правої таблиці).
3. Для кожної пари перевіряється виконання умови об'єднання.
4. Усі пари, для яких умова задовольняється, включаються до результату.
5. Якщо умова не задовольняється для жодної пари, то до результату включається всі вказані поля рядка лівої таблиці та NULL-значення замість полів правої таблиці.
6. Кроки 1–5 повторюються для всіх рядків лівої таблиці.

• зовнішнє об'єднання з правого боку. Синтаксис:

```
<ліва таблиця> RIGHT OUTER JOIN <права таблиця> ON
<умова об'єднання>.
```

Для об'єднання вибирають по черзі всі рядки правої таблиці та кожен з них поєднують тільки з тими рядками лівої таблиці, які задовольняють умову об'єднання. Алгоритм об'єднання аналогічний наведеному вище, але застосовується до правої таблиці.

• повне зовнішнє об'єднання з правого боку. Синтаксис:

```
<ліва таблиця> FULL OUTER JOIN <права таблиця> ON
<умова об'єднання>.
```

Для об'єднання вибирають тільки ті рядки правої та лівої таблиць, які задовольняють умову об'єднання. Алгоритм об'єднання аналогічний наведеному вище, але застосовується до обох таблиць. Рядки в таблиці-відповіді не дублюються.

10.4.4. Специфікатор INTO

Специфікатор INTO вказує, куди занести для збереження таблицю-відповідь. Різні сервери надають різні варіанти синтаксису визначення та реалізації специфікатора.

Варіанти місць збереження: таблиця, список змінних, масив, текстовий файл, екран, принтер.

Зауваження:

1. Діалект SQL СКБД INTERBASE пропонує варіант оператора вибірки, зі специфікатором INTO, результатом якого є таблиця з одного рядка. Значення полів цієї таблиці надаються значенням локальних змінних збережених процедур і тригерів або параметрам динамічних SQL-запитів.
2. Діалекти SQL для СКБД Microsoft SQL Server і FoxPro (та деякі інші) пропонують варіант оператора SELECT, зі специфікатором INTO, який надає можливість завершувати виконання оператора

SELECT створенням у БД нової таблиці, структура та вміст якої збігається зі структурою та вмістом таблиці-відповіді на запит. Синтаксис специфікатора: INTO <ім'я таблиці>.

3. Діалект SQL СКБД Oracle пропонує розширений варіант оператора вибірки, результатом якого необов'язково є таблиця із одного рядка. Таке розширення не підтримується ані в SQL/89, ані в SQL/92.

10.4.5. Специфікатор WHERE

Специфікатор WHERE оператора SELECT дозволяє визначити предикат (умову вибору рядків), результат якого може бути «правда» (TRUE) або «неправда» (FALSE) для кожного рядка таблиці.

Виконання оператора SELECT проводиться у два етапи:

1. Відповідно до умов специфікатора FROM проводиться формування попередньої таблиці-відповіді *R*.
2. Проводиться додаткова перевірка рядків *R* на виконання умови пошуку специфікатора WHERE. Рядки, для яких результат є TRUE, включають в остаточну таблицю-відповідь.

Мова SQL передбачає використання таких предикатів: предикат порівняння, предикат BETWEEN, предикат IN, предикат LIKE, предикат NULL та предикат з квантором. Відразу зазначимо, що в усіх реалізаціях SQL на ефективність виконання запиту істотно впливає наявність в умові пошуку простих предикатів порівняння (предикатів, які задають порівняння поля таблиці з константою). Наявність таких предикатів дозволяє СКБД використовувати індекси під час виконання запиту, тобто уникати повного перегляду таблиці.

Операції, які використовуються в предикатах WHERE:

- операції порівняння — =, >, <, >=, <=, <>, !=, !<, !>;
- операції визначення інтервалу — BETWEEN, NOT BETWEEN;
- операції перевірки належності до списку — IN, NOT IN;
- операції порівняння рядків за шаблоном — LIKE, NOT LIKE;
- операції перевірки визначеності — IS NULL, IS NOT NULL;
- операції булевої алгебри — AND, OR, NOT

Предикат порівняння. Виконує перевірку вказаної умови порівняння.

Синтаксис:

```
<арифметичний вираз> <операція порівняння> <арифметичний вираз>
```

Арифметичні вирази лівої та правої частин предиката порівняння будуються за загальними правилами побудови арифметичних виразів і можуть містити у загальному випадку імена полів таблиць з розділу FROM та константи (не обов'язково літерні; замість літерної константи може використовуватися ім'я поля таблиці або ім'я змінної). Типи даних арифметичних виразів мають бути сумісними (наприклад, якщо тип поля a1 таблиці *A* є типом символьних рядків, то предикат "a1=5" є помилковим).

Значення предиката дорівнює TRUE в тому разі, коли визначена операцією порівняння умова виконується.

Приклад:

```
SELECT прізв FROM kadry WHERE стать = "ч"
```

Результатом є вибірка прізвищ з поля прізв усіх чоловіків з таблиці kadry. У прикладі використано предикат порівняння: стать = "ч".

Для формування складних запитів можна створювати складені предикати порівняння шляхом об'єднання предикатів порівняння такими логічними операціями:

- AND – якщо має виконуватися обидві умови, які з'єднуються за його допомогою;
- OR – якщо мають виконуватися одна з умов, з'єднаних за його допомогою;
- AND NOT – якщо має виконуватися перша і не виконуватися друга умова;
- OR NOT – коли має виконуватися перша умова або не виконуватися друга, причому існує пріоритет AND над OR (спочатку виконуються всі операції AND і тільки після цього – операції OR). Для одержання бажаного результату предиката WHERE умови слід увести у правильному порядку, що можна організувати введенням дужок.

Приклад:

```
SELECT прізв FROM kadry WHERE стать = "ч" AND вік < 50
```

Результатом є вибірка прізвищ з поля прізв усіх чоловіків з таблиці kadry, вік яких менше за 50 років. У прикладі використано складений предикат порівняння : стать = "ч" AND вік < 50.

Предикат BETWEEN. Виконує перевірку обмеженості значень поля за вказаними межами.

Синтаксис:

```
<вираз> [NOT] BETWEEN <нижнє значення> AND <верхнє значення>
```

Значення предиката дорівнює TRUE в тому разі, коли значення лівого операнда знаходиться між указаними межами.

За визначенням предиката встановлюють, що:

- результат "x BETWEEN y AND z" той самий, що й результат логічного виразу "x >= y AND x <= z";
- результат "x NOT BETWEEN y AND z" той самий, що і результат "NOT (x BETWEEN y AND z)".

Приклад використання:

```
SELECT name, cost FROM model WHERE cost BETWEEN 10000
AND 30000
```

Результатом є вибірка автомобілів з ціною від $10000 до $30000.

Предикат IN. Перевіряє, чи знаходиться значення виразу, що стоїть зліва від IN, у множині перерахованих справа від IN значень.

Синтаксис:

```
<вираз> [NOT] IN (<вираз1>, <вираз2>...)
```

Лівий операнд і значення зі списку правого операнда мають бути одного типу.

Значення предиката дорівнює TRUE в тому випадку, коли значення лівого операнда збігається хоча б з одним значенням зі списку правого операнда. В інших випадках (включаючи випадок, коли список правого операнда порожній) значення предиката IN дорівнює FALSE. Згідно з визначенням значення предиката "x NOT IN S" дорівнює значенню предиката "NOT (x IN S)".

Приклад:

```
SELECT прізв FROM kadr WHERE підр NOT IN ("КВ", "ОХР")
```

Результатом є вибірка прізвищ усіх співробітників, крім тих, що працюють у підрозділах КБ і ОХР.

Найбільш важливим застосуванням IN є підзапити, тобто вкладені запити.

Приклад:

```
SELECT name FROM customer WHERE key IN (SELECT key
FROM acc WHERE sold=0)
```

Результатом є список усіх замовників, які не сплатили свій рахунок; інформація про рахунки знаходиться в таблиці acc, інформація про замовників – у таблиці customer.

Предикат LIKE. Дозволяє побудувати умови порівняння за шаблоном (за маскою).

Синтаксис:

```
<вираз> [NOT] LIKE <"шаблон"> [ESCAPE <символ>]
```

Тип даних поля лівого операнда і зразка мають бути типами символьних рядків. У розділі ESCAPE має бути одиночний символ.

Значення предиката дорівнює TRUE, якщо шаблон є вкладеним рядком заданого поля. При цьому, якщо розділу ESCAPE небуде, то під час порівняння шаблону з рядком проводиться спеціальна інтерпретація двох символів шаблону: символ підкреслювання («_») означає будь-який одиночний символ; символ відсотка («%») – послідовність довільних символів довільної довжини (може бути нульовим).

Якщо ж розділ ESCAPE наявний і специфікує деякий одиночний символ *x*, то пари символів «*x*_» і «*x*%» представляють одиночні символи «_» та «%» відповідно.

Значення предиката "x NOT LIKE y ESCAPE z" збігається зі значенням "NOT x LIKE y ESCAPE z".

Наприклад, щоб знайти клієнтів з прізвищами, початкові літери яких "Іва", потрібно створити такий запит:

```
SELECT name FROM customer WHERE name LIKE "Іва%"
```

Предикат NULL. Використовується, якщо в поле таблиці не введено ніякого значення, СКБД занесе у нього NULL-значення. NULL-значення також можна ввести у поле таблиці, виконавши операцію зміни даних.

NULL-значення означає, що значення поля є недоступним або невідомим, тобто його не ввели. Для того, щоб вибрати записи зі значеннями NULL, використовують ключове слово IS NULL у фразі WHERE. При цьому потрібно пам'ятати:

- NULL означає повну відсутність значення. Це не те саме, що 0 чи рядок нульової довжини;
- з NULL неможливі ніякі порівняння. Наприклад, два значення NULL не дорівнюють одне одному.

Під час упорядкування таблиці-відповіді рядки, які містять NULL-значення у полі впорядкування, будуть виведені першими.

Предикат NULL описується за синтаксисом:

```
<вираз> IS [NOT] NULL
```

Значення предиката "x IS NULL" дорівнює TRUE тоді і тільки тоді, коли значення *x* не визначено, інакше – FALSE. Значення предиката "x NOT IS NULL" дорівнює значенню "NOT x IS NULL".

Приклад використання:

```
SELECT * FROM customer WHERE city IS NULL
```

Результатом є вибірка усіх записів, для яких не введено поле city.

Предикат STARTING. Дозволяє задати умову збігу значення перших символів текстових полів заданому шаблону.

Синтаксис:

```
<вираз> [NOT] STARTING [WITH] <"шаблон">
```

Типи даних поля лівого операнда і шаблону мають бути типами символьних рядків. Значення предиката дорівнює TRUE, якщо поле починається з шаблону. Значення предиката `"x NOT STARTING y"` збігається зі значенням `"NOT x STARTING y"`.

Наприклад, щоб знайти клієнтів з прізвищем, початкові літери яких `"Іва"`, потрібно створити такий запит:

```
SELECT name FROM customer WHERE name STARTING "Іва"
```

Предикат CONTAINING. Дозволяє задати для пошуку послідовності за вказаним шаблоном символів у текстовому полі.

Синтаксис:

```
<вираз> [NOT] CONTAINING <"шаблон">
```

Типи даних поля лівого операнда і зразка мають бути типами символьних рядків. Значення предиката дорівнює TRUE, якщо поле містить указану в шаблоні послідовність символів. Значення предиката `"x NOT CONTAINING y"` збігається зі значенням `"NOT x CONTAINING y"`.

Наприклад, щоб знайти клієнтів із прізвищем, до якого входить ланцюжок літер `"ван"`, потрібно створити такий запит:

```
SELECT name FROM customer WHERE name CONTAINING "ван"
```

У предикатах можна використовувати стандартні функції.

Функція UPPER. Перетворює (перед порівнянням) символи текстових полів на великі літери.

Примітка. Функцію можна використовувати не тільки в специфікації WHERE.

Синтаксис:

```
UPPER (<вираз>)
```

Наприклад, якщо потрібно знайти всіх замовників з Москви, а в полі `city` це значення записано як `"Москва"` або як `"МОСКВА"`, для впевненого пошуку потрібно скористатися функцією `UPPER ()`.

```
SELECT * FROM customer WHERE UPPER (city) ="МОСКВА".
```

Функція CAST. Використовують для перетворення типів змінних (полів).

Синтаксис:

```
CAST (<вираз> AS <тип даних>)
```

Функція зводить результат обчислення виразу до вказаного типу даних. При цьому типи полів, що використовувалися під час обчислення виразу, не змінюються.

Допустимі такі відповідності:

```
NUMERIC   - CHARACTER, DATE;
CHARACTER - NUMERIC, DATE;
DATE      - CHARACTER, NUMERIC.
```

Наприклад, потрібно знайти всіх замовників, які замовили 209 чи 309 одиниць товару:

```
SELECT * FROM customer WHERE CAST (kilkist AS CHAR(4))
LIKE "%09%"
```

10.4.6. Специфікатор GROUP BY

Цей специфікатор дозволяє поділити вибірку (таблицю-відповідь) на декілька груп за заданим критерієм.

Синтаксис:

```
GROUP BY <поле> [, <поле>...]
```

Якщо позначити через *R* таблицю, яка є результатом SQL-запиту зі специфікаторами FROM і WHERE, то результатом обробки специфікатора GROUP BY буде розподіл *R* на кілька груп рядків. До групи надходять рядки *R*, в яких збігаються значення всіх полів, указаних у специфікаторі GROUP BY. Остаточний результат групування – це таблиця, у якій кожна група представлена лише одним рядком.

Для додаткового вибору рядків, які входять до групи, використовують специфікатор HAVING.

10.4.7. Специфікатор HAVING

Специфікатор HAVING, як правило, використовують сумісно зі специфікатором GROUP BY. Умова пошуку HAVING є додатковою умовою щодо групування рядків таблиці-відповіді. Формально розділ HAVING може бути використано і у запиті, який не містить GROUP BY. У цьому разі результат запиту формується подібно до формування результату запиту зі специфікатором WHERE (з такою самого умовою вибору).

Синтаксис:

```
HAVING <умова відбору>
```

Умова пошуку специфікатора HAVING будується за тими самими синтаксичними правилами, що й умова пошуку специфікатора WHERE, і може містити ті самі предикати. Результатом виконання розділу HAVING є згрупована таблиця, що містить рядки – представники групи рядків, для яких результат обчислення умови HAVING є TRUE.

Наприклад, задано таблиці Dogovor, Org, Tovar.

Dogovor

N	Kod_org	Kod_tov	K_vo
1	1	1	2
2	1	1	4
3	2	1	5
4	2	1	7
5	1	2	4

Tovar

Kod_tov	Nazv	Price
1	Папір	7.00
2	Ручка	1.00
3	Олівець	0.50

Org

Kod_org	Nazv
1	Opr1
2	Opr2
3	Opr3

Припустімо, сформовано запит:

```
SELECT org.nazv Організація
FROM dogovor, org
WHERE dogovor.kod_org=org.kod_org
GROUP BY dog.kod_org
```

Результат:

Організація
Opr1
Opr2

Результатом є список організацій, з якими укладено договори.

Приклад запиту з HAVING:

```
SELECT org.nazv Організація
FROM dogovor, org
WHERE dogovor.kod_org=org.kod_org
GROUP BY dog.kod_org
HAVING dog.kod_org >1;
```

Результат:

Організація
Opr2

На відміну від попереднього прикладу, кожна група перевіряється на умову `dog.kod_org >1`.

10.4.6. Функції агрегування

Агрегатні функції потрібні для того, щоб обчислювати деяке значення для заданої множини рядків. Такою множиною рядків може бути група рядків, якщо до таблиці застосовано операцію групування, або вся таблиця.

У SQL визначено такі агрегатні функції:

`COUNT (аргумент)` – **кількість значень у полі;**

`SUM (аргумент)` – **сума значень;**

`AVG (аргумент)` – **середнє значення;**

`MAX (аргумент)` – **найбільше значення;**

`MIN (аргумент)` – **найменше значення.**

Кожна з цих функцій оперує сукупністю значень поля всіх рядків групи (або всієї таблиці) і створює єдине значення для кожної групи.

Для всіх агрегатних функцій, крім COUNT(), використовують такий порядок обчислень: на підставі параметрів агрегатної функції із заданої множини рядків (групи) виробляється список значень. Потім за цим списком значень проводиться обчислення функції. Якщо список виявився порожнім, значення функції COUNT() для нього дорівнює 0, а значення всіх інших функцій – NULL.

Синтаксис функції COUNT:

```
COUNT ( * | [ALL] <val> | DISTINCT <val>)
```

Опис параметрів:

*	– підрахувати кількість всіх без винятку рядків у таблиці (включаючи дублікати);
ALL	– підрахувати кількість рядків у таблиці, у яких значення не є NULL-значенням;
DISTINCT	– підрахувати кількість рядків у таблиці, у яких значення не є NULL-значенням, за винятком дублікатів;
<val>	– поле або вираз, що обчислюється.

Результат обчислення функції COUNT() – точне число з масштабом і точністю, визначеними у реалізації.

Приклад використання:

```
SELECT COUNT (DISTINCT CURRENCY) FROM COUNTRY;
```

Синтаксис інших агрегатних функцій:

```
SUM ([ALL] <val> | DISTINCT <val>);
AVG ([ALL] <val> | DISTINCT <val>);
MAX ([ALL] <val> | DISTINCT <val>);
MIN ([ALL] <val> | DISTINCT <val>);
```

Опис параметрів

ALL	– обробити значення всіх рядків таблиці;
DISTINCT	– обробити тільки унікальні значення рядків таблиці;
<val>	– поле або вираз, що обчислюється.

Для функцій SUM і AVG аргументом є поле таблиці, яке має містити числові значення. Слід відзначити, що тут поле – це поле віртуальної таблиці, в якому можуть міститися дані з поля базової таблиці і з поля, значення якого є результатом обчислення (поле, яке обчислюється на підставі значень полів таблиці). При цьому вираз обчислення може бути склад-

ним, але не має містити SQL-функції (вкладеність SQL-функцій не допускається). З SQL-функцій можна складати будь-які вирази. Тип результату значень функцій MAX() і MIN() збігається з типом поля. Під час обчислення функцій SUM() і AVG() тип поля не має бути типом символьних рядків, а тип результату функції – це тип точних чисел з масштабом і точністю, вказаними у реалізації.

Розглянемо різні випадки застосування агрегатних функцій у SQL-запитах.

Якщо результат табличного виразу *R* не є згрупованою таблицею, то поява хоча б однієї агрегатної функції від множини рядків *R* у списку вибірки приводить до того, що *R* неявно розглядається як згрупована таблиця, яка складається з однієї або нуля груп з відсутніми полями групування (критерій групування порожній). Тому в цьому випадку у списку вибірки не допускається використання специфікацій рядків *R*, тобто всі вони мають знаходитися всередині специфікацій агрегатних функцій. Результатом запиту є таблиця, яка складається не більше ніж з одного рядка, отриманого за допомогою застосування агрегатних функцій до *R*.

У разі, якщо *R* являє собою згруповану таблицю, тобто табличний вираз має розділ GROUP BY і вказано принаймні одне поле групування, правила формування списку вибірки повністю відповідають правилам формування умови вибірки розділу HAVING: допускається пряме використання імен полів групування, а імена інших полів *R* можуть з'являтися тільки всередині специфікацій агрегатних функцій. Результатом запиту є таблиця, кількість рядків у якій дорівнює кількості груп в *R*, і кожний рядок формується на підставі значень полів групування й агрегатних функцій для даної групи.

Розглянемо деякі приклади:

Під час групування даних часто підраховують результуючі значення для вказаної групи. Ці значення можна заносити в робоче поле вибірки.

Наприклад, для заданих таблиць Dogovor, Org, Tovar сформуємо більш розширений запит:

```
SELECT org.nazv AS Організація, tovar.nazv AS Товар,
tovar.price AS Ціна,
      SUM(dogovor.k_vo) AS Кількість,
      SUM(tovar.price*dogovor.k_vo)
AS Сума
FROM dogovor, org, tovar
WHERE dogovor.kod_org=org.kod_org
AND dogovor.kod_tov=tovar.kod_tov
GROUP BY dog.kod_org, tovar.kod_tov
```

Результат:

Організація	Товар	Ціна	Кількість	Сума
Орг1	Папір	7.00	5	35.00
Орг1	Ручка	1.00	4	4.00
Орг2	Папір	7.00	12	84.00

Розглянемо таблицю Podr:

N_podr	FIO	Szar
1	Іванов	120
1	Петров	110
2	Попов	100
2	Сокол	100
3	Павлов	150

Сформуємо запит для складання відомості, у якій по кожному підрозділу потрібно вказати кількість працівників та суму до сплати на кожен підрозділ.

```
SELECT podr, count(*) AS kv, sum(szar)
FROM podr
GROUP BY podr INTO TABLE vidom
```

Результат:

N_podr	Kv	SUM_Szar
1	2	230
2	2	200
3	1	150

Для того, щоб отримати список тільки тих клієнтів, які мають більше одного договору, необхідно виконати запит:

```
SELECT kod_klient, COUNT(*)
FROM dogovor
GROUP BY kod_klient
HAVING COUNT(*)>1
```

10.4.9. Специфікатор ORDER BY

Специфікатор надає можливість встановити бажаний порядок перегляду результату виразу запитів.

Синтаксис:

```
ORDER BY <поле>[ASC | DESC] [{, <поле>}] [ASC | DESC] ...
```

Як видно з цих синтаксичних правил, фактично задається список полів результату запитів і для кожного поля вказується порядок перегляду рядків результату залежно від значень цього поля (ASC – за зростанням (за замовчуванням), DESC – за зменшенням). Поля можна задавати їхніми іменами лише тоді, коли запит не містить операції UNION або UNION ALL. Не можна використовувати складені арифметичні вирази над іменами полів. Дозволяється у розділі ORDER BY вказувати порядковий номер поля в таблиці-результаті запиту.

Головне призначення ORDER BY – зробити результат вибірки найбільш зручним для сприйняття.

Розглянемо запит, який виводить прізвище, ім'я та по батькові замовників.

```
SELECT name1, name2,name3 FROM customer
```

Результат запиту:

```
Шалімов Борис Якович
Токарев Антон Васильович
Соколов Іван Петрович
Макашов Святослав Петрович
Андрушенко Олег Андрійович
```

Результати вибірки будуть (є) незручними, невідсортованими.

Сформуємо запит з ORDER BY:

```
SELECT name1, name2, name3 FROM customer
ORDER BY 1, 2, name3
```

Результат запиту:

```
Андрушенко Олег Андрійович
Макашов Святослав Петрович
Соколов Іван Петрович
Токарев Антон Васильович
Шалімов Борис Якович
```

10.4.10. Специфікатор PLAN

Специфікатор PLAN використовують для формування власного плану користувача щодо виконання запиту.

Синтаксис:

```
[PLAN <plan_expr>]
<plan_expr> =
[JOIN | [SORT] MERGE] (<plan_item> | <plan_expr>
     [, <plan_item> | <plan_expr> ...])
 <plan_item> = {table | alias}
     NATURAL|INDEX(<index>[, <index>...])|ORDER <index>
```

Опис параметрів:

JOIN – визначає необов'язкову процедуру з'єднання таблиць для збільшення швидкості виконання запиту, в якому обробляється кілька таблиць;

SORT MERGE – застосовують для збільшення швидкості виконання запиту, якщо не існує індексів, за якими можна з'єднувати таблиці;

NATURAL – визначає процедуру послідовного доступу до таблиць для виконання запиту;

INDEX – визначає перелік індексів, які необхідно використати під час виконання запиту;

ORDER – вимагає сортування таблиці, вказаної в <plan_item> за заданим індексом.

10.5. Підзапити

Мова SQL дає можливість використовувати вкладені SQL-запити. Вкладені запити називають *підзапитами*. Коли запит вкладено всередину другого запиту (оператора SELECT), то внутрішній запит виконується першим. Якщо підзапит повертає одне значення, то його можна використовувати в операціях порівняння.

Приклад використання підзапиту, який повертає одне значення:

```
SELECT name1, name2,
(SELECT SUM(summa) AS acc_sum FROM account
     WHERE key_customer = 2)
FROM customer
WHERE key_customer = 2
```

Результат вибірки:

```
 name1     name2     acc_sum
Соколов    Іван      10500.0
```

Запит виводить прізвище та ім'я покупця разом із сумою його рахунків (поле *summa*). Цей запит можна виконати і за допомогою двотабличного запиту, але тоді він буде виконуватися більш повільно.

Якщо підзапит повертає багато значень, то його можна використовувати тільки у специфікаторі WHERE. Як правило, більшість запитів із підзапитами можуть бути переписані як багатотабличні запити. Якщо умова WHERE містить підзапити, тоді кожний підзапит обчислюється окремо для кожного рядка таблиці. У специфікаторі WHERE підзапит використовують як аргумент предикатів IN, EXISTS і предикатів з квантором.

Предикат IN. Використовують, якщо необхідно перевірити, чи збігається значення конкретного поля з одним із значень визначеного дискретного набору.

Наприклад, запит «знайти всіх покупців, які зробили замовлення 3 жовтня 1990 року» формується так:

```
SELECT * FROM customer
WHERE 03/10/1990 IN
     (SELECT zdate FROM zakaz
```

```
WHERE customer.nom_pokup = zakaz.nom_pokup)
```

Предикат EXISTS. Підзапит є корельованим, якщо у своїй умові вибірки він посилається на дані зовнішнього запиту.

```
SELECT * FROM tab1
 WHERE tab1.field1= ( SELECT COUNT (tab2.field1)
      FROM tab2
     WHERE tab2.field2=tab1.field2);
```

Зверніть увагу на те, що в розділі WHERE підзапиту порівнюються поля з таблиць tab2 і tab1, тоді як у розділі FROM підзапиту вказана тільки таблиця tab2. Такі запити виконуються так:

1. Береться перший запис із таблиці tab1.
2. Виконується підзапит із застосуванням поточного значення поля зовнішнього підзапиту.
3. Отриманий результат підзапиту передається у зовнішній запит.
4. Виконується зовнішній запит.
5. Залежно від результату виконання зовнішнього запиту поточний запис таблиці tab1 заноситься чи не заноситься до таблиці-відповіді.
6. Береться наступний запис таблиці tab1 і процедура повторюється з пункту 2 до кінця перегляду таблиці tab1.

Отже, корельований підзапит буде виконуватись стільки разів, скільки записів у зовнішній таблиці. Корельовані підзапити є найбільш складними як для проектування, так і для виконання БД.

Під час виконання корельованих підзапитів часто використовують предикат EXISTS.

Предикат EXISTS дозволяє перевірити, чи існує в таблиці один або більше рядків, які задовольняють умову пошуку.

Предикат має такий синтаксис:

```
EXISTS <підзапит>
```

Значенням цього предиката є TRUE або FALSE. Значення TRUE встановлюється тоді і тільки тоді, коли результат обчислення підзапиту не є порожнім. Перед оператором EXISTS не має стояти ім'я поля або константи. Список полів підзапиту, який оброблюється EXISTS, зазвичай визначається як * (всі поля). Фраза WHERE у такому запиті перевіряє, чи існують вибрані записи.

Наприклад, запит «потрібно знайти тих продавців, які отримали товар зі складу» за допомогою предиката EXISTS формується так:

```
SELECT DISTINCT nom_prod, FIO
FROM customer out
WHERE EXISTS
```

```
( SELECT *
FROM customer inn
WHERE inn.nom_prod=out.nom_prod AND
    inn.nom_pokup<>out.nom_pokup )
```

Оскільки EXISTS генерує значення FALSE або TRUE, його можна використовувати разом з іншими булевими виразами за допомогою операторів AND, OR, NOT. Найчастіше з EXISTS використовують оператор NOT.

Наприклад, для виконання запиту «знайти продавців, які мають тільки одного покупця» достатньо сформувати запит:

```
SELECT DISTINCT nom_prod
FROM customer out
WHERE NOT EXIST
    ( SELECT *
    FROM customer inn
    WHERE inn..nom_prod=out.nom_prod AND
        inn.nom_pokup<>out.nom_pokup )
```

Предикати з кванторами SOME або ANY. Незалежно від застосування, SOME або ANY виконуються абсолютно однаково і є взаємозамінними.

Синтаксис:

```
<вираз для порівняння> <оператор порівняння> ANY | SOME
<підзапит>
```

Предикат SOME (ANY) дає результати TRUE, FALSE або UNKNOWN.

Позначимо через *x* результат обчислення арифметичного виразу лівої частини предиката, а через *S* – результат обчислення підзапиту.

Предикат "x <оператор порівняння> SOME S" має значення FALSE, якщо *S* є порожнім або значення предиката "x <оператор порівняння> s" дорівнює FALSE для кожного *s*, яке входить в *S*. Предикат "x <оператор порівняння> SOME S" має значення TRUE, якщо значення предиката "x <оператор порівняння> s" дорівнює TRUE хоча б для одного s, яке входить у *S*. В інших випадках значення предиката "x <оператор порівняння> SOME S" дорівнює UNKNOWN.

Наприклад, результатом обчислення виразу

```
WHERE stowbch > SOME (SELECT pole FROM tabl)
```

буде TRUE для рядків, в яких значення поля stowbch буде більше хоча б за одного з існуючих значень поля pole в таблиці tabl.

Наведемо кілька прикладів використання.

Запит «видати всі записи про відправлення товарів зі складу, для яких кількість товару більша за середнє значення відправлення хоча б одного товару».

Синтаксис запиту:

```
SELECT *
FROM pozhod r1
WHERE r1.kilk > SOME
      ( SELECT AVG(r2.kilk)
      FROM rozhod r2
      GROUP BY nom_pokup ).
```

Запит «обрати всі замовлення, величина яких більша за величину хоча б одного замовлення, зробленого 7 жовтня 1990 року».

Синтаксис запиту:

```
SELECT *
FROM zakaz
WHERE amt > ANY
      ( SELECT amt
      FROM zakaz
      WHERE zdate=10/07/1990 ).
```

Предикат з квантором ALL. Використовують для перевірки, чи виконується вказана умова для всіх рядків таблиці-відповіді деякого підзапиту.

Синтаксис:

```
<вираз для порівняння> <оператор порівняння> ALL <підзапит>
```

Предикат ALL дає результати TRUE, FALSE або UNKNOWN.

Позначимо через x результат обчислення арифметичного виразу лівої частини предиката, а через S – результат обчислення підзапиту.

Предикат "x <оператор порівняння> ALL S" має значення TRUE, якщо S є порожнім або значення предиката "x <оператор порівняння> s" дорівнює TRUE для кожного s, яке входить в S. Предикат "x <оператор порівняння> ALL S" має значення FALSE, якщо значення предиката "x <оператор порівняння> s" дорівнює FALSE хоча б для одного s, що входить у S. В інших випадках значення предиката "x <оператор порівняння> ALL S" дорівнює UNKNOWN.

Приклад: запит «визначити адресу покупця, який придбав найбільшу кількість товарів» формують за таким синтаксисом:

```
SELECT c.*
FROM  customer c
WHERE c.nom_pokup =
      ( SELECT rr.nom_pokup
      FROM rozhod rr
      GROUP BY rr.nom_pokup
      HAVING SUM(rr.kilk) > ALL
```

```
( SELECT SUM(rrr.kilk)
FROM rozhod rrr
GROUP BY rrr.nom_pokup ) )
```

DISTINCT у підзапитах. У деяких випадках використовують DISTINCT для гарантії того, що в результаті виконання підзапиту буде отримане одне значення. Наприклад, запит «потрібно знайти всі замовлення, з якими працює продавець, що обслуговує покупця з номером 2001» має такий синтаксис:

```
SELECT *
FROM zakaz
WHERE nom_prod =
      ( SELECT DISTINCT nom_prod
      FROM zakaz
      WHERE nom_pokup=2001 )
```

Функція SINGULAR. Використовують, якщо з таблиці потрібно вибрати тільки ті записи, для яких підзапит повертає тільки одне значення.

Синтаксис:

```
SINGULAR (<оператор_select>)
```

Наприклад, потрібно знайти всіх замовників, які замовили тільки один товар.

```
SELECT c.*
FROM customer c
WHERE SINGULAR
      ( SELECT *
      FROM conract r
      WHERE r.kod_customer=c.customer)
```

10.6. Реалізація операцій реляційної алгебри засобами мови SQL

Мову запитів SQL створено для надання можливості зручно виконувати операції над відношеннями, що зберігаються як таблиці БД. Мова проектувалася так, щоб забезпечити виконання повної множини операцій реляційної алгебри, хоча не всі операції застосовують під час формування запитів до БД внаслідок нормалізації відношень, наприклад, не має сенсу *виконання теоретико-множинних операцій. Розглянемо множину цих операцій* та визначимо відповідні запити.

10.6.1. Теоретико-множинні операції

Об'єднання відношень. Результатом операції об'єднання двох відношень буде нове відношення, до якого ввійдуть усі кортежі, що входять хоча б в одне з відношень-операндів.

```
SELECT * FROM <ім'я таблиці1> UNION SELECT * FROM
<ім'я таблиці2>
```

Наприклад

```
SELECT * FROM tab1 UNION SELECT * FROM tab2
```

Перетин відношень. Результатом операції перетину двох відношень буде відношення, до якого ввійдуть усі кортежі, що входять в обидва відношення-операнди. Його можна виконати за допомогою такого запиту:

```
SELECT * FROM <ім'я таблиці1>
 WHERE EXISTS
     ( SELECT * FROM <ім'я таблиці2>
     WHERE <умова рівності значень відповідних полів>)
```

Наприклад:

```
SELECT * FROM instrum1
WHERE EXISTS
(SELECT * FROM instrum2
  WHERE instrum1.kod = instrum2.kod
     and instrum1.naimenov= instrum2.naimenov
     and instrum1.kol_vo= instrum2.kol_vo
     and instrum1.czhena= instrum2.szhena
     and instrum1.n_sklada= instrum2.n_sklada )
```

Різниця відношень. Відношення, що є різницею двох відношень, містить усі кортежі, які входять у перше відношення-операнд, але жоден з них не входить у друге відношення-операнд.

Різницю можна знайти за допомогою такого запиту:

```
SELECT * FROM <ім'я таблиці1>
 WHERE NOT EXISTS
  ( SELECT * FROM <ім'я таблиці2>
     WHERE <умова рівності значень відповідних полів>)
```

Наприклад:

```
SELECT * FROM instrum1
WHERE NOT EXISTS
(SELECT * FROM instrum2
  WHERE instrum1.kod = instrum2.kod
     and instrum1.naimenov= instrum2.naimenov
```

```
and instrum1.kol_vo= instrum2.kol_vo
and instrum1.czhena= instrum2.szhena
and instrum1.n_sklada= instrum2.n_sklada )
```

Декартовий добуток відношень. Результатом виконання операції прямого (декартового) добутку двох відношень є відношення, кортежі якого є конкатенацією (зчепленням) кортежів першого і другого відношень-операндів. Його можна виконати за допомогою такого запиту:

```
SELECT * FROM <ім'я таблиці1>, <ім'я таблиці2>
```

Наприклад:

```
SELECT * FROM tab1, tab2
```

Простий декартовий добуток не має реального застосування, на практиці використовують добутки з умовами з'єднань.

10.6.2. Спеціальні реляційні операції

До таких операцій належать:

- селекція (обмеження, фільтрація) відношення;
- проекція відношення;
- сполучення відношень;
- ділення відношень.

Селекція відношення. Це вибір кортежів відношення-операнда, які задовольняють вказану умову.

Відповідний запит формується так:

```
SELECT * FROM <ім'я таблиці>;
 WHERE <умова відбору>
```

Наприклад:

```
SELECT * FROM таб1
 WHERE field1 = value
```

Результатом є вибірка (селекція) записів таблиці `tab1`, для яких поле `field1` дорівнює `value`.

Проекція відношення. Це взяття відповідних значень із заданих стовпчиків кортежів відношення-операнда.

Запит формується шляхом перерахування полів таблиці і внесенням ключового слова `DISTINCT` для запобігання відображенню записів-дублікатів:

```
SELECT DISTINCT <перелік полів> FROM <ім'я таблиці>
```

Наприклад:

```
SELECT DISTINCT field1, field2, field3 FROM tab1;
```

Результатом є всі унікальні значення (проекція) вказаних полів.

Комбінування селекції та проекції. У разі необхідності (що трапляється в більшості реальних ситуацій) селективний запит і запит на отримання проекції можна об'єднати:

```
SELECT UNIQUE field1, field2, field3 FROM tab1
WHERE field1 = value
```

Сполучення відношень. Під час сполучення двох відношень за деякою умовою утвориться відношення, кортежі якого є конкатенацією кортежів першого і другого відношень, які задовольняють вказану умову. Відповідний запит формується так:

```
SELECT * FROM tab1, tab2 WHERE <умова відбору>
```

Наприклад:

```
SELECT * FROM instrum1, instrum2 WHERE instrum1.kod
<> instrum1.kod
```

Ділення відношень. У операції реляційного ділення в загальному випадку приймають участь два операнди-відношення. Припустимо, що відношення *A* визначають за схемою {*X*1, *X*2, ...*Xn*, *Y*1, *Y*2, ...*Ym*}, а відношення *B* – за схемою {*Y*1, *Y*2, ...*Ym*}. Тоді відношення-результат операції *A*/*B* визначається схемою {*X*1, *X*2, ...*Xn*} та складається з тих кортежів відношення *A*, які містять такі значення атрибутів, для яких множина значень атрибутів {*Y*1, *Y*2, ...*Ym* } збігається із множиною значень атрибутів {*Y*1, *Y*2, ...*Ym* } відношення *B*.

Відповідний запит формується так:

```
SELECT <ім'я таблиці1>.X1,... <ім'я таблиці1>.Xn FROM
<ім'я таблиці1>
 WHERE NOT EXISTS
(SELECT <ім'я таблиці1>.Y1,... <ім'я таблиці1>.Ym
  FROM <ім'я таблиці1>
  WHERE <ім'я таблиці1>.Y1,... <ім'я таблиці1>.Ym <>
     ANY (SELECT <ім'я таблиці2>.Y1,… <ім'я таблиці2>.Ym
     FROM <ім'я таблиці2>))
```

Наприклад:

```
SELECT tab1.kod, tab1.name FROM tab1
 WHERE NOT EXISTS
(SELECT tab1.cena, tab1.sklad
  FROM tab1
  WHERE tab1.cena, tab1.sklad <>
     ANY (SELECT tab2.cena, tab2.sklad FROM tab2))
```

10.7. Модифікація вмісту таблиці

10.7.1. Додавання рядків до таблиці

Для додавання нових рядків до таблиці застосовують оператор INSERT.

Синтаксис:

```
INSERT [INTO] <ім'я таблиці> [(<список полів>)]
[DEFAULT] VALUES <список значень>
```

Специфікатор INSERT використовують для визначення таблиці, до якої будуть додані рядки. Якщо не потрібно заповнювати всі поля нового рядка, то у специфікаторі INSERT після імені таблиці вказують список полів, які набувають значення.

Специфікатор VALUES вказує значення полів нового рядка.

Наведемо приклади:

```
INSERT INTO customer (nom_pokup, name1, name2, name3, city)
     VALUES (2011, 'Сидоров', 'Ілля', 'Петрович', 'Київ')
INSERT INTO skalespeople
     VALUES (1001, 'Іванов', NULL, 12.5)
```

Специфікатор DEFAULT вказує на те, що доданий запис буде містити значення за замовчуванням, якщо воно визначене. Наприклад:

```
INSERT INTO body DEFAULT VALUES
```

Специфікатор DEFAULT трапляється у команді INSERT INTO ще й у списку значень, коли у разі необхідності ним можна замінити значення, яке вводять.

```
INSERT INTO body (name_body) VALUES (DEFAULT)
```

У предикаті оператора INSERT можна застосовувати разом із підзапитом для того, щоб вибрати значення з однієї таблиці і перенести їх в іншу, для цього потрібно замінити VALUES на підзапит.

Приклади:

```
INSERT INTO Londonstaff SELECT *
     FROM Salespeople
     WHERE cyti='London'
INSERT INTO Daytotals (date, total) SELECT zdate, SUM(amt)
     FROM zakaz
     GROUP BY zdate;
```

Відзначимо, що імена полів у двох таблицях не збігаються. Але якщо date і total – повний список полів таблиці і розташовані вони у ній в указаному порядку, то їх імена в INTO можна опустити.

10.7.2. Вилучення рядків з таблиці

Рядки з таблиці можна вилучити за допомогою оператора DELETE. Оператор вилучає повністю рядки, а не окремі поля.

Синтаксис:

```
DELETE [FROM] <ім'я таблиці> [WHERE <умова>]
```

Наприклад:

```
DELETE FROM customer
```

Якщо не вказано умови вибору, то вилучаються всі записи.

Найчастіше з таблиці потрібно вилучити тільки деякі рядки, ті, що відповідають заданій умові.

```
DELETE FROM customer
      WHERE city='Київ'
```

У предикаті оператора DELETE можна використовувати підзапити. Це дає можливість формувати досить складні критерії вилучення рядків, що потребує особливої уваги і обережності. Наприклад, для вилучення всіх покупців Лондонского офісу фірми достатньо використати такий запит:

```
DELETE FROM customer
WHERE snum=ANY
(SELECT snum FROM Salespeople WHERE city='London')
```

В операторі можна використовувати зв'язані підзапити. Наприклад, можна знайти найменше замовлення за кожний день і вилучити продавця, якому таке замовлення було адресовано.

```
DELETE FROM Salespeople
 WHERE snum IN
 (SELECT snum FROM Zakaz a
  WHERE amt=
     (SELECT MIN (amt)
      FROM Zakaz b
       WHERE a.zdate=b.zdate))
```

10.7.3. Зміна значень полів

Оператор UPDATE використовують для зміни деяких або всіх значень в існуючій таблиці.

Синтаксис:

```
UPDATE <ім'я таблиці> SET <поле> = <значення> [, <поле>
= <значення>...] [WHERE <умова>]
```

UPDATE дозволяє вказати ім'я таблиці, для якої буде виконана операція.

Специфікатор SET вказує, які заміни потрібно виконати для вказаного поля (полів).

Наприклад:

```
UPDATE Salespeople
 SET sname = 'Іванов', city = 'Москва'
 WHERE snom= 1004
UPDATE Salespeople
 SET comm = comm*2
 WHERE city = 'Москва'
```

У предикаті оператора UPDATE можна використовувати підзапити. Наприклад, підвищити комісійні для всіх продавців, які обслуговують більше ніж одного покупця:

```
UPDATE Salespeople
 SET comm = comm+0.02
 WHERE 2 <=
   (SELECT COUNT (nom_pokup)
     FROM customer
     WHERE customer.nom_prod= Salespeople.nom_prod)
```

10.8. Підтримка обробки транзакцій

10.8.1. Загальні відомості

Транзакція являє собою послідовність операцій над БД, яка має бути виконана або повністю, або не виконана зовсім. Ядро БД гарантує, що операції, які виконуються в рамках транзакції, будуть цілком і повністю зафіксовані на диску або (у разі помилки під час виконання транзакції) буде відновлено стан БД, в якому вона перебувала до початку транзакції. Транзакція є не тільки засобом захисту від непередбачених аварійних відмов, вона також надає можливість безпечного виходу з ситуації, коли програма виявляє логічну помилку.

Транзакції активізують різні користувачі БД незалежно один від одного, тому виникає ситуація паралельного виконання різних транзакцій.

Існують класичні проблеми паралельного виконання транзакцій.

Читання без перевірки, або «брудне читання» – програма може читати дані, які змінено, але зміни можуть бути не підтверджені, а пізніше – скасовані, тобто запис читається невірно.

Читання, що не повторюється – виникає як різне прочитання одного і того самого запису (під час виконання паралельної транзакції та після її завершення), внаслідок підтвердженої зміни запису паралельною транзакцією, тобто один запис читається в різних варіантах.

Фантомні читання – читання записів, які можуть бути вилучені паралельною транзакцією під час її завершення.

Втрачені зміни – виникають, коли транзакція змінює дані, які заздалегідь змінені паралельною транзакцією.

Порушення цілісності – ефекти, які виникають під час проведення каскадних змін однією транзакцією із паралельною зміною записів у батьківській таблиці іншою транзакцією.

СКБД усуває проблеми паралельного виконання транзакцій за допомогою використання системи блокування. Блокування являє собою деяку вимогу, яку висуває програма відносно обробки таблиць або груп рядків БД. СКБД гарантує, що поки дані заблоковано, ніяка інша програма не може оновити їх. Якщо інша програма запитує заблоковані дані, ядро БД або примушує цю програму чекати зняття блокування, або повертає повідомлення про помилку. Блокування виконується шляхом встановлення *рівня ізоляції транзакції.*

Існує декілька значень для рівнів ізоляції: *Dirty Read*, *Committed Read*, *Repeatable Read*. Вони відрізняються правилами використання даних програмою.

1. Рівень *Dirty Read* («брудне» читання або читання з робочих буферів) дозволяє транзакції читати всі поточні зміни, що проводяться іншими транзакціями, в тому числі не зафіксовані в БД.
2. Рівень *Committed Read* (підтверджене читання) дозволяє транзакції читати тільки зафіксовані в БД зміни інших транзакцій.
3. Рівень *Repeatable Read* (читання, що повторюється) не дозволяє транзакції після початку її виконання і до кінця читати зміни, зроблені іншими транзакціями, навіть якщо ці зміни зафіксовані в БД. Наступна транзакція додатка буде читати оновлені записи.

10.8.2. Керування транзакціями

Оператор визначення початку транзакції SET TRANSACTION

Синтаксис оператора містить параметри для встановлення рівня ізоляції транзакції :

```
SET TRANSACTION [READ WRITE | READ ONLY]
 [WAIT | NO WAIT]
 [[ISOLATION LEVEL] {SNAPSHOT [TABLE STABILITY]
   | READ COMMITTED [[NO] RECORD_VERSION]}]
 [RESERVING <reserving_clause>;
<reserving_clause> = table [, table ...]
[FOR [SHARED | PROTECTED] {READ | WRITE}]
[, <reserving_clause>]
```

Опис параметрів:

READ WRITE	–	дозвіл для транзакції на виконання операцій читання і запису;
READ ONLY	–	дозвіл для транзакції на виконання тільки операцій читання;
WAIT	–	вказує, що в разі блокування запису іншою транзакцією необхідно чекати її завершення;
NO WAIT	–	вказує, що в разі конфлікту блокування обробка запиту припиняється та повертається повідомлення про помилку;
ISOLATION LEVEL	–	визначає рівень ізоляції транзакції в умовах, коли до таблиць БД звертаються інші транзакції. Можливі значення: • SNAPSHOT TABLE STABILITY (*значення за замовчуванням*) – зміни, зроблені іншими транзакціями в таблиці, недоступні для даної транзакції (блокування таблиці); • READ COMMITED – показувати змінену версію рядка таблиці та дозволити неконфліктні зміни, при цьому: • NO RECORD_VERSION – читається останній непідтверджений варіант запису (але якщо включено очікування (WAIT), транзакція буде чекати завершення іншої транзакції і повторить читання). • RECORD_VERSION – читається остання підтверджена версія запису;
RESERVING <reserving_clause>	–	транзакція блокує доступ конкуруючих транзакцій до вказаних таблиць.

Завершення транзакції може проводитися або з підтвердженням змін у БД, проведених транзакцією (підтвердження транзакції), або з відмовою від змін (відкат транзакції).

Оператор підтвердження транзакції COMMIT

Синтаксис:

```
COMMIT [WORK] [TRANSACTION <name>] [RELEASE] [RETAIN
[SNAPSHOT]];
```

Опис параметрів:

`WORK`	– ключове слово для забезпечення сумісності версій СКБД;
`TRANSACTION <name>`	– ім'я транзакції, що завершується. За відсутності підтверджується транзакція, яка виконувалась за замовчуванням;
`RELEASE`	– ключове слово для забезпечення сумісності версій СКБД;
`RETAIN [SNAPSHOT]`	– підтверджує зміни і зберігає поточний контекст транзакції.

Приклади використання:

- підтвердження змін у БД:
  `COMMIT;`
- завершення іменованої транзакції:
  `COMMIT TR1;`
- завершення іменованої транзакції та збереження контексту транзакції:
  `COMMIT RETAIN.`

Оператор відкату транзакції COMMIT

Синтаксис:

```
ROLLBACK [WORK];
```

`WORK` – ключове слово для сумісності версій.

10.9. Підтримка спеціальних функцій сервера

10.9.1. З'єднання з базою даних

Сервер забезпечує обробку різних БД, як локальних (установлених на комп'ютері сервера), так і розміщених на інших комп'ютерах, які під'єднанні до однієї з сервером мережі. Для обробки конкретної БД спочатку необхідно виконати процедуру з'єднання.

Оператор `CONNECT` виконує процедуру з'єднання, тобто:

- визначає, чи є БД локальною або віддаленою;
- перевіряє безпомилковість мережі;
- перевіряє головну сторінку БД. Файл БД має містити правильну БД, створену тією самою версію сервера. У разі порушень вимог до бази повертається повідомлення про помилку;
- перевіряє правильність заданого імені користувача і паролю, які необхідні для доступу до БД;
- установлює з'єднання з БД.

Кожне з'єднання спричинює до роз'єднання з попередньою БД.

Синтаксис:

```
CONNECT ["]<filespec>["] [USER "username" [PASSWORD
"password"]];
```

Опис параметрів:

`"<filespec>"`	–	специфікація файлу БД, тобто визначення шляху до нього;
`USER "username"`	–	рядок, який визначає ім'я користувача для БД. Сервер перевіряє ім'я користувача в списку встановлених для сервера імен користувачів БД (файл захисту – isc.gdb);
`PASSWORD "password"`	–	рядок, який описує пароль для з'єднання користувача з БД. Сервер перевіряє ім'я користувача і пароль на наявність у файлі захисту БД.

Приклад використання:

```
CONNECT "employee.gdb" USER "ACCT_REC" PASSWORD "peanuts";
```

10.9.2. Підтримка паралельного резервування бази даних

Для того, щоб запобігти пошкодженням БД внаслідок помилок у роботі апаратних засобів або мережних пристроїв, проводиться процедура паралельного резервування, яка полягає в тому, що для основної БД створюється паралельна БД (тінь), вміст якої повністю збігається з основною базою. Будь-які операції над основною базою автоматично проводяться і з базою-тінню, тобто вони існують та поновлюються паралельно. Тінь БД може складатися з одного файлу (*single-file shadow*) або з кількох (*multi-file shadow*) файлів на дискових пристроях зовнішньої пам'яті. Кожна тінь визначається унікальним позитивним цілим числом.

У процесі паралельного резервування використовують такі компоненти:

- БД, що дублюється;
- системна таблиця RDB$FILES, яка містить список файлів тіні та іншу службову інформацію про БД;
- тінь, яка складається з одного або кількох файлів.

Оператор початку паралельного копіювання та створення тіні CREATE SHADOW

Синтаксис:

```
CREATE SHADOW set_num [AUTO | MANUAL] [CONDITIONAL]
"<filespec>" [LENGTH [=] int [PAGE[S]]]
 [<secondary_file>];
```

```
<secondary_file> = FILE "<filespec>" [<fileinfo>]
[<secondary_file>]
<fileinfo> = LENGTH [=] int [PAGE[S]] | STARTING [AT
[PAGE]] int [<fileinfo>]
```

Опис параметрів:

`set_num`	– позитивне ціле число, яке визначає тінь, до якої належать усі створювані файли;
`AUTO`	– визначає, що у разі помилки обробки тіні за замовчуванням буде реалізовано таку реакцію: всі операції з БД (підключення та звернення) продовжуються, але обробка тіні припиняється та файл тіні відключається;
`MANUAL`	– визначає, що в разі помилки обробки тіні підключення до БД та звернення припиняються, доки тінь не стає доступною або доки всі посилання до тіні не будуть вилучені з БД;
`CONDITIONAL`	– вказує, що в разі недоступності тіні або дискової помилки під час роботи з нею буде створено нову тінь, для продовження паралельного резервування;
`"<filespec>"`	– визначення специфікації (маршруту) файлу тіні;
`LENGTH [=] int [PAGE[S]]`	– довжина в сторінках БД додаткового файлу тіні. Розмір сторінки визначається розміром сторінки БД;
`<secondary_file>`	– визначення параметрів додаткового файлу тіні (якщо його визначено);
`STARTING [AT [PAGE]] int`	– номер початкової сторінки, з якої починається додатковий файл тіні.

Приклади:

- створення однофайлової автоматичної тіні для БД employee.gdb:
```
CREATE SHADOW 1 AUTO "employee.shd";
```
- створення однофайлової автоматичної тіні для БД employee.gdb з обробкою помилок звернень до тіні:
```
CREATE SHADOW 2 CONDITIONAL "employee.shd" LENGTH 1000;
```
- створення багатофайлової автоматичної тіні для БД employee.gdb з обробкою за замовчуванням помилок звернень до тіні:

```
CREATE SHADOW 3 AUTO        "employee.sh1"
 FILE "employee.sh2"     STARTING AT PAGE 1000
 FILE "employee.sh3"     STARTING AT PAGE 2000;
```

Оператор припинення паралельного копіювання та знищення файлу тіні DROP SHADOW

Синтаксис:

```
DROP SHADOW set_num – знищити файл тіні за номером set_num;
```

10.9.3. Визначення набору символів

Бази даних можна створювати з використанням різного мовного набору символів. Перед з'єднанням з БД, тобто перед оператором CONNECT, необхідно встановити той набір символів, на якому створено БД.

Оператор визначення активного набору символів SET NAMES

Синтаксис:

```
SET NAMES [charset] – встановити активним набір символів з іменем
                      charset
```

Приклад використання:

```
SET NAMES LATIN1
CONNECT "employee.gdb"
```

10.9.4. Підтримка показника селективності індексу

Для деяких індексованих таблиць спостерігається ситуація радикального зростання або зменшення кількості рядків з однаковим значенням індексних полів. Для підвищення продуктивності обробки таких таблиць можна використати *показник селективності (корисності)* індексу таблиці.

Показник селективності – це число, обчислення якого базується на кількості унікальних значень рядків у таблиці. Обчислення показника не є автоматичним, його виконує відповідний оператор. Частоту обчислення визначає користувач. Вона залежить від активності поновлень таблиці. Під час формування оптимального плану виконання запитів ураховують значення показників селективності індексів.

Оператор обчислення показника селективності SET STATISTICS INDEX

Синтаксис:

```
SET STATISTICS INDEX name – обчислити показник селективності для
                            індексу name.
```

Приклад використання:

```
SET STATISTICS INDEX MINSALX
```
– повторно обчислити показник селективності для індексу MINSALX

Зауваження. SET STATISTICS не поновлює індекс. Щоб поновити індекс, використовуйте ALTER INDEX.

11. Інформаційні системи з розподіленими базами даних

11.1. Інформаційне розподілене середовище

11.1.1. Середовище розподілених даних

Сукупність даних, у якій є хоча б два елементи з різними псевдонімами (аліасами), тобто такі, що мають різне адресне фізичне розміщення, і яка утворює середовище розподілених даних.

Кожний адресний елемент даних розглядають як розподілений інформаційний ресурс.

Переважно інформаційні ресурси є локальними реляційними БД, розміщеними в комп'ютерній мережі. Доступ до них забезпечує використання технологій *клієнт – сервер*.

Фактично псевдонім є мережевою адресою розподілених даних (інформаційних ресурсів) незалежно від того, як саме фізично реалізована мережеву архітектуру.

11.1.2. Середовище об'єктів доступу до розподілених реляційних даних

Доступ до розподілених даних виконується за допомогою спеціально розроблених інтеграційних об'єктів доступу до даних, які також є частиною інформаційного розподіленого середовища. Об'єкти доступу до даних подано у вигляді наборів драйверів (SQL LINKS, ODBC, OLE DB) і бібліотек програмних компонентів (COM, DCOM, Active X, ADO, CORBA).

Сукупність розподілених реляційних об'єктів і розподілених об'єктів доступу до даних утворює розподілене інформаційне середовище (рис. 11.1).

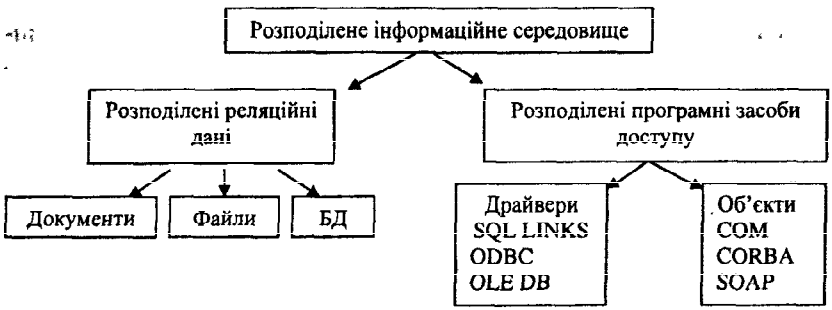


Рис. 11.1. Структура розподіленого інформаційного середовища

11.1.3. Огляд стратегій побудови розподіленої інформаційної системи

Розподілені інформаційні системи (РІС), зокрема системи, орієнтовані на застосування розподілених баз даних (РБД), розробляють у різних варіантах технології *клієнт – сервер*. Ця технологія спрямована на застосування об'єктно-орієнтованого підходу. У цій архітектурі будь-який об'єкт, який використовує ресурси іншого об'єкта, визначають як *клієнт*, а об'єкт, що надає ресурси, називають *сервером*.

Поведінка об'єкта-клієнта характеризується послугами, що надають йому об'єкти-сервери. Отже, що об'єкт-сервер має певні зобов'язання перед клієнтом, тобто для нього визначається деякий контракт (набір зобов'язань) стосовно об'єктів-клієнтів. Кінцеву множину операцій, передбачених контрактом, називають протоколом. Побудова розподілених інформаційних систем виконується за однією з таких стратегій:

- довільна;
- контейнерна;
- прикладна;
- технологічна.

Довільна стратегія – подання документів. Стратегія орієнтована на розробку інформаційних систем, спрямованих на забезпечення користувача інформацією, при цьому основою надання інформації є набір документів довільної структури. Основна ідея цієї стратегії така

розподілені об'єкти даних – документи користувача.

Основною характеристикою РІС, орієнтованих на таку стратегію, є те, що немає механізму перетворень розподілених об'єктів. Сервери в цих системах є серверами документальних розподілених об'єктів. При цьому зазвичай документ подається у вигляді однієї одиниці збереження.

Характерним прикладом є геоінформаційні системи (ГІС) (один об'єкт – одна карта – один документ у БД).

Інший приклад реалізації довільної технології – глобальна мережа інтернету, у якій одиницею подання інформації є документ, робота з яким можлива як з одним цілим без можливості перетворення його структури.

Контейнерна стратегія – подання інформаційних об'єктів. Контейнерна стратегія передбачає розробку набору інструментальних контейнерних об'єктів – спеціальних програмних об'єктів, призначення яких – доставка розподілених даних для програми користувача. Власне контейнерні об'єкти не містять безпосередньо даних, а тільки забезпечують маніпулювання ними.

Основний приклад – набір ActiveX компонентів доступу до даних (ADO – Active Database Object). Набір компонентів побудований на основі COM-технологій.

Технологія припускає наявність клієнтського додатка з клієнтськими компонентами і серверним додатком – контейнера серверних (COM, ActiveX) компонентів.

Інший приклад – набір CORBA (Common Object Request Broker Architecture) компонентів доступу до даних. Технологія припускає наявність клієнтського додатка з клієнтськими компонентами і серверним додатком – контейнера серверних (CORBA) компонентів. При цьому набір компонентів забезпечує роботу з різними платформами – використання різних операційних систем і апаратних рішень для клієнта і сервера.

Під час розробки компонентів формулюють цільові функції (контракти) для задоволення стратегічних інтересів клієнтських програм. Формулювання набору функцій визначають професійні вимоги до серверів і контейнерів. У технологіях COM і CORBA контракти реалізують у вигляді інтерфейсів серверних об'єктів.

Прикладна стратегія. Прикладна стратегія розробки розподілених інформаційних систем формується для задоволення стратегічного інтересу професійного використання. Вона базується на потребі створити інструментальні засоби, які забезпечують можливості перетворення представлень розподілених даних для розв'язання задач аналізу і прогнозування.

Прикладна стратегія чи стратегія професійного використання реалізується шляхом розробки набору інструментальних засобів (візуальних середовищ програмування), спроектованих для реалізації функцій користувача, тобто CASE-засобів для розробки програмного забезпечення. Класичні приклади – середовища візуального проектування, такі як Delphi, C++, Java, ASP, .NET. При цьому РБД розглядається як сукупний інформаційний ресурс, супроводжуваний установленими зв'язками логічної сумісності та відповідності між розподіленими даними.

Фізичний сенс застосування прикладної стратегії полягає у реалізації логічної схеми ІРБД у R-моделі та засобів проектування РБД, а також у розробці засобів і методів керування БД.

Технологічні стратегії. Технологічна стратегія передбачає розробку нових інформаційних технологій, які дозволяють користувачам генерувати нову інформацію і нові знання на підставі реалізації різних алгоритмів обробки інформації, що міститься в РБД. Така система, наприклад система екологічного моніторингу регіону, складається з ряду автоматизованих робочих місць (еколога, медика, економіста, юриста) та являє собою суперсистему, внутрішні компоненти якої призначені для реалізації проміж-

них операцій. Система реалізує ланцюг *вихідна розподілена інформація – проміжні перетворення – нова інформація.*

Технологічна стратегія (реалізація якої і є основним завданням прикладного програміста) заснована на таких методологічних ознаках:

1. Виділення сукупності технологічних розподілених даних. Технологічні дані не мають основної інформації, вони описують її структуру (склад і розподіл по мережі). Технологічні дані, будучи частиною сукупності розподілених даних, утворюють базу метаданих системи.

2. Формування інтегрованих даних системи. Принцип припускає створення «головної бази даних», що також є частиною РБД і містить інтегровані дані. Наприклад, головна БД містить усереднені показники забруднення.

3. Розробка засобів реалізації обмінної технології з метою забезпечення синхронності змін, несуперечності даних і функціонування алгоритмів формування головної БД.

4. Формування документального подання як фрагментів розподілених даних (вихідної інформації), так і нової створеної системою інформації. При цьому формування документального подання проводиться переважно у рамках професійного стандарту. Наприклад, подання інформації про екологічний стан регіону проводиться в рамках міжнародного стандарту опису екологічної ситуації відомого як DPSIR (рис. 11.2) – стандарт, що містить такі елементи:

- Driving forces – джерело впливу на навколишнє середовище (хто забруднює);
- Pressure – характеристики впливу джерела на середовище (чим забруднює);
- State – стан навколишнього середовища (як саме забруднює);
- Impact – наслідки забруднень (кому і як зашкоджує);
- Response – вжиті контрзаходи (що треба робити).

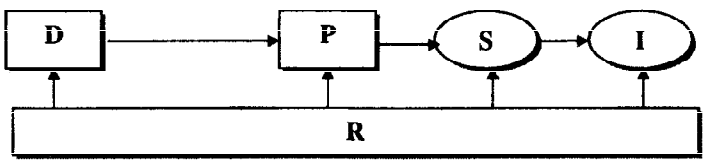


Рис. 11.2. Структура DPSIR – стандарту для опису екологічної ситуації

11.1.4. Мережеві рішення підтримки розподілених інформаційних систем

Для організації роботи комп'ютерних мереж використовують мережеве програмне забезпечення двох видів:

1. Для обслуговування мережевих комунікаційних протоколів:
   - мережі на основі протоколів сім'ї IPX/SPX; наприклад, операційні системи фірми Novel і OS/2 підтримує протокол SPX/IPX;
   - мережі на основі протоколів сім'ї TCP/IP. Мережі використовують під час розробки мережевого програмного забезпечення, яке підтримують засобами операційної системи Windows, Unix, SUN OS 4.1 і корпорації IBM у мережі Ethernet чи Token Ring.

2. Для передачі повідомлень між додатком і БД. Наприклад, пакет програм SQL*Net. Перша версія SQL*Net існує в декількох варіантах, які підтримують різні протоколи: TCP/IP, SPX/IPX, ASYN. SQL*Net із протоколом ASYNC забезпечує доступ до сервера по телефонних мережах із використанням модемів на обох кінцях.

Останнім часом спостерігається інтенсивний розвиток цифрових мереж інтегрального обслуговування, так званих ISDN-мереж (Integrated Services Digital Network).

Ці мережі мають такі характеристики:
- високу продуктивність;
- економічну ефективність;
- високу швидкість передачі даних;
- ширший спектр послуг.

Розрізняють два види цифрових мереж – вузькосмугові N-ISDN (N-Narrow) і широкосмугові B-ISDN (B-Broad). Для ефективного використання PIC застосовують широкосмугові цифрові мережі.

Такі мережі працюють на основі використання асинхронного режиму доставки – режиму ATM (Asynchronous Transfer Mode) із застосуванням режиму швидкої комутації пакетів (Fast Packet Switching).

Базові принципи, покладені в основу технології ATM, такі:

1. ATM забезпечує *трансляцію* (передачу по мережі) комірок. Інформація передається по мережі невеликими пакетами фіксованого розміру, які називають пакетами ATM чи комірками. Комірка складається з 5-байтового адресного поля і 48-байтового інформаційного поля.

2. ATM установлює *з'єднання*. Перед генерацією джерело спочатку встановлює з'єднання з одержувачем, при цьому для конкретного з'єднання резервується смуга пропуску і гарантується певна якість обслуговування.

3. ATM реалізує *комутацію*. У мережах ATM усі пристрої, такі як робочі станції, сервери, маршрутизатори, мости, приєднані до комутатора. У разі встановлення з'єднання комутатори визначають оптимальний маршрут для передачі інтегральної інформації, тобто виконують функції маршрутизатора. Після встановлення з'єднання комутатори пересилають дані

тільки наступному вузлу заздалегідь обраного маршруту, виконуючи функцію моста. Комутатор не коректує помилок і не виконує функції повторного запиту помилково переданої інформації. Протоколи захисту від помилок передачі переносяться на верхні рівні. Іншими словами, протокол АТМ є протоколом канального рівня.

В АТМ-мережах використовують такі основні поняття:

- віртуальний канал – однонаправлене з'єднання для передачі комірок з однаковим ідентифікатором каналу;
- віртуальний шлях – об'єднання групи однонаправлених віртуальних каналів, що мають свої унікальні ідентифікатори;
- віртуальне з'єднання – логічне з'єднання, що складається з одного або декількох віртуальних каналів, установлюється між двома пристроями в мережі. Віртуальне з'єднання є двонаправленим; це означає, що після встановлення з'єднання кожна кінцева станція може як надсилати повідомлення іншій станції, так і одержувати їх від неї.

Широкосмугові АТМ-мережі з'єднання мають такі характерні ознаки:

- створюються і закриваються функціями верхніх рівнів;
- підтримують якість обслуговування;
- можуть бути як комутованими, так і постійними;
- підтримують як одно-, так і двонаправлений потік ресурсів;
- забезпечують різні значення пропускної здатності в обох напрямках.

Найбільш перспективними є комутовані віртуальні з'єднання, оскільки вони:

- використовують пропускну здатність каналів зв'язку тільки у разі потреби;
- установлюються автоматично за потреби;
- забезпечують стійкість до відмов, тобто у випадку виходу з ладу комутатора вибирають альтернативний шлях.

Розглянемо структуру заголовка комірки:

1 байт	GFC	VPI		
2 байт	VPI	VCI		
3 байт	VCI	VCI		
4 байт	VCI	PT	Res	CLP
5 байт	HEC			

GFC (Generic Flow Control) – прапорці керування потоком;

VPI (Virtual Path Identifier) – ідентифікатор віртуального шляху;

VCI (Virtual Channel Identifier) – ідентифікатор віртуального каналу;
PT (Payload Type) – вид змісту; для пакетів користувачів PT = 00;
Res – резерв;
CLP (Cell Loss Priority) – пріоритет втрати комірки;
HEC (Header Error Control) – поліном корекції помилок у заголовку.

11.1.5. Класифікація методів маршрутизації

Класифікувати методи доставки інформаційних ресурсів у B-ISDN-мережах можна за декількома ознаками:

1. За характером використовуваної службової інформації для вибору маршруту:
   - ізольовані – кожен маршрутизатор робить маршрутні обчислення, спираючись тільки на інформацію про свій стан і стан суміжних каналів зв'язку; обміну службовою інформацією між маршрутизаторами немає;
   - глобальні – вибираючи віртуальний шлях, маршрутизатор користується інформацією про стан усієї мережі в цілому;
   - комбіновані – обмін частиною службової інформації може мати глобальний характер.
2. За місцем виконання маршрутних обчислень:
   - централізовані – у мережі є спеціальний центр керування мережею, який виконує усі маршрутні обчислення;
   - децентралізовані – кожен маршрутизатор розраховує свою таблицю маршрутизації, при цьому кожен маршрутизатор, розраховуючи власну таблицю, використовує результати аналогічних розрахунків, проведених на інших маршрутизаторах (алгоритм Дейкстри);
   - комбіновані (ієрархічні) – уся мережа розбивається на зони, у кожній визначено власний зоновий центр керування мережею.
3. За ступенем реагування на зміни умов (топології, навантаження):
   - адаптивні – алгоритми реагують на зміну умов під час визначення маршруту;
   - статичні – алгоритми на зміну умов не реагують (використовуються переважно для попередніх оцінок);
   - квазистатичні – алгоритми періодично перераховують маршрути і не реагують на зміну умов усередині інтервалу між перерахунками.
4. За кількістю обраних можливих маршрутів передачі:
   - одномаршрутні – після розрахунку таблиць маршрутизації вибирається єдиний віртуальний шлях для передачі комірок для вказаного адресата;
   - багатомаршрутні – визначається множина можливих віртуальних шляхів і комірки передаються віртуальним шляхом, вибраним із де-

якої підмножини з використанням спеціальних процедур вибору шляху трансляції (стохастичної, циклічної, такої, що враховує завантаження шляхів);

- комбіновані – мережа розбивається на зони; перший маршрутизатор вибирає шлях між зональними центрами керування, а в кожній зоні визначається множина паралельних шляхів. Це дозволяє досягти ефективного використання пропускної здатності каналів зв'язків, рівномірного розподілу навантаження, обходу ушкоджених шляхів.

5. За кількістю з'єднань у мережі:

- з'єднання «точка – точка», встановлюється між двома кінцевими станціями (алгоритм найкоротшої черги Дейкстри);
- з'єднання «точка – багатоточка», встановлюється між станцією і групою кінцевих станцій;
- з'єднання «багатоточка – багатоточка», встановлюється між групою початкових станцій і групою кінцевих станцій.

11.1.6. Модель доставки інформаційних ресурсів

Схему моделі показано на рис. 11.3, де використано такі позначення:

I_m – джерела службової інформації, які генерують повідомлення з постійною швидкістю;

$(T_{поч} - T_{кін})n$ – джерела інформації користувачів, які генерують повідомлення зі змінною швидкістю;

Мр – мультиплексори;

КС – комутаційна система;

$ВК_i$ – віртуальні канали, що входять до складу обчислювальної мережі.

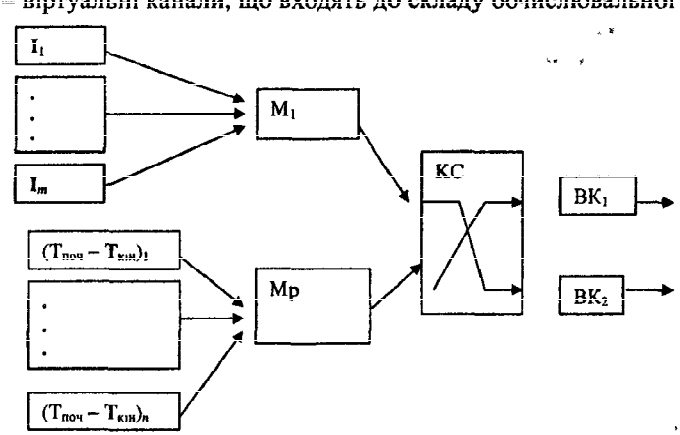


Рис. 11.3. Схема доставки інформаційних ресурсів у мережі

11.2. Розподілені бази даних

11.2.1. Розподілена система баз даних

Розподілена база даних, або Distributed DataBase (DDB) – це сукупність розподілених даних, представлених у рамках деякої реляційної моделі, яка відображає дані та зв'язки між ними.

Реляційна модель даних забезпечує однорідність подання даних і зв'язків між ними у вигляді концептуальної схеми БД.

Реалізація реляційної моделі приводить до створення РБД.

Розподілена база даних, яка створюється заново, – однорідна. Разом із цим, вона нерідко створюється як сукупність групи локальних БД, що уже функціонують у ряді систем. У цьому випадку виникає неоднорідна РБД. Обидва типи баз містяться в системі керування розподіленою базою даних (СКРБД).

У загальному випадку локальні БД, що складають РБД, не обов'язково мають бути однорідними (тобто вестися однією СКБД) чи оброблятися в середовищі однієї і тієї самої операційної системи і на комп'ютерах того самого типу. Наприклад, одна БД може бути базою Oracle на комп'ютері SUN з операційною системою SUN OS(UNIX), друга БД може вестися СКБД DB2 на мейнфреймі IBM3090 з операційною системою MVS, а для ведення третьої бази може використовуватися СКБД SQL/DS також на мейнфреймі IBM, але з операційною системою VM. Обов'язкова тільки одна умова – локальні БД зв'язані між собою логічно, але фізично розміщені на декількох машинах (вузлах), що входять в одну комп'ютерну мережу.

При цьому мережева підтримка РБД характеризується такими особливостями:

- кожен вузол має власні БД;
- вузли працюють узгоджено, тобто користувач може одержати доступ до даних на будь-якому вузлі мережі так, ніби вони знаходяться на його власному вузлі;
- функціонування РБД відбувається незалежно від типів використовуваних у системах пристроїв;
- забезпечується просторова прозорість, що дає можливість користувачу не знати, де розміщено компоненти бази;
- підтримка повної функціональності, тобто можливість виконання всіх можливих операцій у БД, яка знаходиться в одній системі;
- гарантія цілісності даних, яку забезпечують функції спостереження за даними, виправлення помилок.

Найважливіші функціональні характеристики РБД такі:

- автономність вузлів РБД, яка означає, що ведення кожної БД може відбуватися незалежно від інших;
- обробка розподілених запитів (SQL-команда), у ході виконання яких відбувається доступ до об'єктів (таблиць чи представлень) різних БД;
- виконання розподілених трансакцій передбачає узгоджене керування усіма залученими БД. При цьому часто використовують технологію двофазної передачі інформації.

Суть РБД полягає в організації доступу користувачів до великих обсягів інформації. Це дозволяє розмістити дані так, що вони, з одного боку, знаходяться в пунктах їх найбільшого використання, а з другого – за допомогою транзакцій забезпечується доступ до будь-яких даних незалежно від того, де вони знаходяться.

Сучасні комп'ютерні технології для проектування РБД розглядають у різних варіантах архітектури *клієнт – сервер* (дворівнева, трирівнева), орієнтованої на об'єктно-орієнтований підхід. У цій архітектурі будь-який об'єкт, що використовує ресурси іншого об'єкта, визначають як клієнт, а об'єкт, що поставляє ресурси, називають сервером відповідно до встановленого протоколу обміну.

11.2.2. Моделювання розподілених БД

Існують різні рішення побудови схем РБД. Завдяки схемі користувач розглядає РБД як єдину БД.

Різні автори пропонують різні підходи до побудови набору схем, які розроблюють під час проектування БД. Однією з найбільш поширених схем є схема Арсеньєва – Яковлєва, зображена на рис. 11.4.

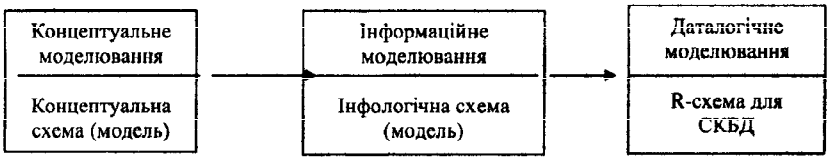


Рис. 11.4. Схема Арсеньєва – Яковлєва для моделювання РБД

Концептуальне моделювання складається з таких операцій:

- визначення джерел знань про предмет (закон евристики, існуючі рішення, документи);
- вибір системи класифікації об'єктів;
- виділення абстракції об'єктів і процесів, тобто виділення класів (типізація, абстрагування);
- побудова структури базових класів;
- побудова схеми взаємодії.

Мета концептуальної побудови – побудова фундаментальної семантики в поданій області.

Нотації опису різні, наприклад, у мові UML концептуальна схема може бути подана у вигляді діаграм класів без баз деталізації за атрибутами і методами.

Інфологічне моделювання виконується на базі концептуальної моделі як її уточнення і деталізація.

Основні операції:

- формування базових атрибутів класів на основі типізації;
- формування додаткових атрибутів;
- визначення сукупного спільного використання атрибутів, тобто створення прототипів таблиць;
- визначення зв'язків логічної цілісності та зв'язків навігації;
- побудова інформаційно-логічної моделі.

У мові UML – повна діаграма статичних класів.

Даталогічне моделювання. Даталогічний аналіз і проектування – це остаточне проектування реляційної схеми даних з урахуванням середовища обраної СКБД.

Проектування виконується на основі інфологічної схеми і полягає у виконанні таких пунктів:

- виділення таблиць (R-відношень);
- визначення фізичних форматів атрибутів на основі типів СКБД;
- визначення складу індексованих полів;
- визначення зв'язків навігації і логічної цілісності;
- установлення бізнес-правил;
- проектування представлень;
- проведення нормалізації реляційних схем (усунення надлишковості, багатозначності).

Результат даталогічного проектування – реляційна схема БД.

11.2.3. Розподілена обробка даних – Disdtributed Data Processing (DDP)

Виконання операцій із БД на одній машині і додатків на іншій і є розподілена обробка даних. Зв'язок між додатком і процесами виконання операцій із БД реалізується як апаратурою, так і програмним забезпеченням. Сутність DDP полягає у тому, що користувач одержує можливість працювати з мережевими службами і прикладними процесами, розміщеними в декількох взаємозв'язаних абонентських системах. При цьому можливі кілька видів робіт, які він може виконувати:

- віддалений запит, наприклад команда, що дозволяє посилати одиночну заявку на виконання обробки даних;
- віддалена транзакція, що направляє групи запитів до прикладного процесу;
- розподілена транзакція, що уможливлює використання декількох серверів і прикладних процесів, які виконуються у групі абонентських систем.

11.2.4. Обробка даних за технологією *клієнт – сервер*

Архітектура *клієнт – сервер*. Для використання РБД потрібно таке апаратне і програмне забезпечення:

- комп'ютер-сервер БД;
- комп'ютер(и)-клієнти;
- комунікаційна мережа;
- мережеве програмне забезпечення;
- прикладне програмне забезпечення.

Технологія *клієнт – сервер* є реалізацією розподіленої обробки даних. У системі архітектури *клієнт – сервер* обробка даних розподілена між комп'ютером-клієнтом і комп'ютером-сервером, зв'язок між якими відбувається по мережі. Цей розподіл процесів обробки даних засновано на групуванні функцій. Як правило, комп'ютер-сервер БД виділяється для виконання операцій із БД, а комп'ютер-клієнт виконує прикладні програми. На рис. 11.5 показано просту архітектуру системи *клієнт – сервер*, до складу якої входять комп'ютер, що діє як сервер, та комп'ютер, що діє як його клієнт. Кожна машина виконує різні функції і має власні ресурси.

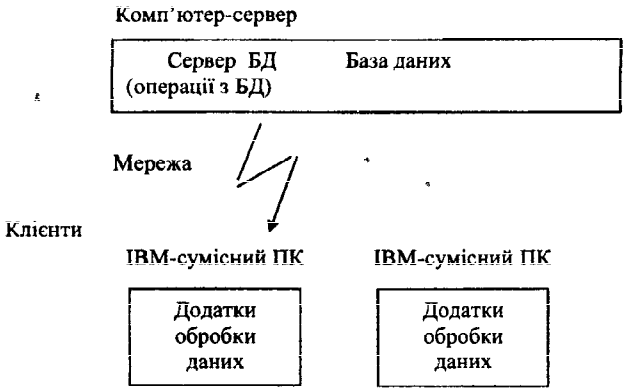


Рис. 11.5. Проста архітектура системи *клієнт – сервер*

Комп'ютери-клієнти. Основна функція комп'ютера-клієнта полягає у виконанні додатка (інтерфейсу з користувачем і відповідної логіки подання даних) і забезпеченні зв'язку із сервером, коли цього потребує додаток.

Зазвичай клієнти – це повільні і недорогі машини. Залежно від вимог, пропонованих додатками, можуть знадобитися потужніші комп'ютери-клієнти.

Комп'ютер-сервер бази даних. Головна функція комп'ютера-сервера полягає в обслуговуванні потреб клієнта. Термін «сервер» використовують для позначення двох різних груп функцій серверів: файл-сервер і сервер БД. *Файли-сервери не призначені для виконання операцій* із БД, їх основна функція – розподіл файлів між декількома користувачами, тобто забезпечення одночасного доступу багатьох користувачів до файлів на файлі-сервері. Прикладом такого сервера є операційна система NetWare компанії Novell.

Комп'ютер із сервером БД (наприклад, Oracle) виділяється тільки для виконання операцій із БД.

Основні функції сервера БД такі:

- зв'язок з клієнтом, аналіз і виконання запиту до БД, включаючи повернення клієнту результату запиту (набору рядків із БД);
- керування одночасним доступом до БД багатьох користувачів, перенаправлення запитів до інших серверів мережі;
- виконання реляційних операцій, керування словником-довідником даних і забезпечення захисту інформації.

Одна з важливих вимог до сервера – це багатозадачність операційної системи (бажано загального користування), у середовищі якої розміщений сервер БД. Багато різновидів систем UNIX, MVS, VM та інші операційні системи і багатозадачні, і загального користування.

Вибираючи комп'ютер-сервер БД, варто взяти до уваги потужність процесора, ємність оперативної пам'яті, обсяг дисків, швидкість роботи з дисками, кількість користувачів, що одночасно працюють, і вартість.

Принципи взаємодії між клієнтськими та серверними частинами. Доступ до БД від прикладної програми чи користувача відбувається шляхом звернення до клієнтської частини системи. Як основний інтерфейс між клієнтською і серверною частинами виступає мова БД SQL.

Ця мова являє собою стандарт інтерфейсу СКБД у відкритих системах. Назва «SQL-сервер» збірна для всіх серверів БД, заснованих на SQL. Сервери БД, інтерфейс яких заснований винятково мовою SQL, мають свої переваги і недоліки. Очевидна перевага – стандартність інтерфейсу. Клієнтські частини будь-якої SQL-орієнтованої СКБД могли б працювати з будь-яким SQL-сервером, хоча на практиці це не зовсім так. Недолік досить очевидний. За такого високого рівня інтерфейсу між клієнтською і серверною частинами системи на комп'ютері клієнта працює замало про-

грам СКБД. Це нормально, якщо як клієнт використовують малопотужну робочу станцію. Але якщо клієнтський комп'ютер має достатню потужність, то часто виникає бажання покласти на нього більше функцій керування БД, розвантаживши сервер, що є «вузьким» місцем усієї системи.

Одним із перспективних напрямів СКБД є гнучке конфігурування системи, за якого розподіл функцій між клієнтською і користувацькою частинами СКБД визначається в ході установлення системи.

Переваги технології *клієнт – сервер*:

1. Незалежність від платформ: доступ до різнорідних мережевих середовищ, до складу яких входять комп'ютери різних типів з різними операційними системами.

2. Незалежність від джерел даних: доступ до інформації різнорідних БД.

3. Економічність: доступність недорогого комп'ютерного устаткування й дедалі більше поширення локальних мереж роблять технологію *клієнт – сервер* вигіднішою за інші технології обробки даних.

4. Можливість залучення великих обчислювальних ресурсів для операцій із БД: оскільки додатки виконуються на комп'ютерах-клієнтах, на комп'ютері-сервері для операцій із БД вивільняються додаткові ресурси центрального процесора й оперативна пам'ять.

5. Формування з часто використовуваних процедур і функцій додатків вбудованих процедур БД. Для виклику цих процедур додатки можуть застосовувати механізм виклику віддалених процедур RPC (Remote Procedure Call). Такі процедури ще більше скорочують час і вартість розробки додатків, зменшують мережевий трафік і поліпшують характеристики функціонування додатків.

6. Підвищення продуктивності роботи кінцевих користувачів: на сьогодні багато кінцевих користувачів освоїли такі системи.

7. Продуктивність. Реальні характеристики, такі як пропускна здатність та час відгуку, можна буде з'ясовувати тільки на практиці. Наприклад, хоча технологія *клієнт – сервер* має ряд переваг перед технологією *термінал-хост*, вона може бути не оптимальною з погляду забезпечення виконання всіх додатків. Продуктивність технології *клієнт – сервер* може бути нижчою від продуктивності технології *термінал-хост*. За невеликої кількості одночасно працюючих користувачів технологія *термінал-хост* забезпечує ліпшу пропускну здатність і час відгуку, ніж технологія *клієнт – сервер*, однак у разі збільшення кількості користувачів продуктивність технології *термінал-хост* швидко погіршується. Таке саме комп'ютерне устаткування і програмне забезпечення за умови використання технології *клієнт – сервер* може підтримувати істотно більшу кількість користувачів за таку саму ціну з незначним погіршенням продуктивності порівняно з отриманими перевагами.

11.3. Принципи функціонування систем із розподіленою базою даних

11.3.1. Фундаментальний принцип розподіленої бази даних

Фундаментальний принцип РБД формулюють так. Для користувача розподілена система має виглядати так само, як і нерозподілена система.

Роботу користувачів у розподіленій системі варто організовувати в такий самий спосіб, як і у нерозподіленій. Усі пов'язані з розподіленими системами проблеми є чи мають бути внутрішніми і повинні виникати тільки на внутрішньому рівні (чи на рівні розробки), а не на зовнішньому рівні (чи на рівні користувачів).

У цьому випадку термін «користувачі» відносять до користувачів (чи розробників додатків), що виконують операції керування даними. Усі ці операції мають залишатися логічно незмінними, на відміну від операцій визначення даних, що, навпаки, можуть бути розширені в розподіленій системі. Наприклад, користувач на вузлі X може вказати, що збережене відношення можна розділити на «фрагменти» для фізичного зберігання на вузлах Y і Z. Поняття «фундаментальний принцип», або «правило нуль» (RuleZero), уперше запропонував К. Дж. Дейт. Цей принцип приводить до набору таких цілей:

Фундаментальний принцип (глобальна прозорість)			
Незалежність від конфігурації	**Незалежність від засобів**	**Незалежність від даних**	**Режими роботи**
Локальна автономія	Незалежність від апаратного забезпечення	Незалежність від фрагментації	Безупинне функціонування
Незалежність від центрального вузла	Незалежність від операційної системи	Незалежність від реплікації	Обробка розподілених запитів
–	Незалежність від мережі	Незалежність від розміщення	Керування розподіленими транзакціями
–	Незалежність від СКБД	–	–

Ці дванадцять цілей не залежать одна від одної, до того ж не всі вони рівнозначні (різні користувачі можуть надавати різного значення різним цілям у різному оточенні), і не вичерпують списку усіх можливих цілей. Однак вони потрібні для розуміння основ розподіленої технології і для загальної характеристики функціональності деякої розподіленої системи.

Слід розрізняти розподілені системи і системи, у яких підтримуються деякі способи віддаленого доступу до даних. У системах з віддаленим доступом до даних користувач може одночасно працювати з даними, роз-

міщеними на декількох віддалених вузлах, але у такому випадку будуть видні з'єднання між ними. У дійсно розподіленій системі, навпаки, усі з'єднання сховані від користувача.

11.3.2. Незалежність від конфігурації

Локальна автономія. У розподіленій системі вузли потрібно робити *автономними*. Локальна автономія означає, що операціями на цьому вузлі керує власне вузол, тобто функціонування будь-якого вузла X не залежить від успішного виконання деяких операцій на якомусь іншому вузлі *B* (інакше вихід з ладу вузла *B* може призвести до неможливості виконання операцій на вузлі X, навіть якщо з вузлом X нічого не трапилось). Із принципу локальної автономії також випливає, що зберігання даних і керування даними відбуваються локально. Дійсно, навіть якщо доступ до даних виконується з інших віддалених вузлів, усі вони відносяться до деякої локальної БД. Питання безпеки, цілісності та структури зберігання локальних даних залишаються під контролем цього локального вузла.

Насправді мета локальної автономності досягається неповністю, оскільки існує багато ситуацій, у яких вузол X має надавати деяку частину керування іншому вузлу Y. Тому ціль досягнення локальної автономії потребує точнішого формулювання, а саме: *вузли варто робити автономними максимально можливою мірою.*

Незалежність від центрального вузла. Під локальною автономією розуміють те, що *усі вузли мають розглядаються як рівні.* Отже, не має існувати ніякої залежності і від центрального «основного» вузла з деяким централізованим обслуговуванням, наприклад, централізованою обробкою запитів чи централізованим керуванням транзакціями. Таким чином, друга мета є логічним наслідком першої. Отже, навіть якщо локальна автономія не досягається повною мірою, то досягнення незалежності від центрального вузла саме по собі дуже важливе і може розглядатися як окрема мета.

Залежність від центрального вузла не бажана принаймні через дві причини. По-перше, центральний вузол може бути «вузьким» місцем усієї системи, а по-друге, важливіше те, що система в такому випадку стане дуже *уразливою*, тобто у разі виходу з ладу центрального вузла може відмовити вся система.

11.3.3. Незалежність від системних засобів

Незалежність від апаратного забезпечення. Використовувані сьогодні комп'ютери досить різні: це комп'ютери фірм IBC, DEC, HP, а також персональні комп'ютери і робочі станції інших типів. У зв'язку з цим іс-

нує реальна потреба інтеграції даних на всіх цих системах і створення для користувача «уявлення єдиної системи». Отже, дуже важливою є можливість запуску копій однієї і тієї самої СКБД на різному апаратному забезпеченні для того, щоб різні комп'ютери могли працювати в розподіленій системі як рівні партнери.

Незалежність від операційної системи. Ця мета є наслідком попередньої. Важливо запустити ту саму СКБД не тільки на різному апаратному забезпеченні, але й на різних операційних системах, причому навіть у тих випадках, коли різні операційні системи використовують на однотипному апаратному забезпеченні. Наприклад, для того щоб версії СКБД для операційної системи MVS, а також для систем UNIX і Windows могли спільно працювати в одній і тій самій розподіленій системі.

Незалежність від мережі. Якщо система підтримує кілька вузлів із різним апаратним забезпеченням і різними операційними системами, то бажано, щоб вона підтримувала також різні типи мереж.

Незалежність від СКБД. Ця мета передбачає використання дещо менш точного формулювання припущення про строгу однорідність, яке означає, що *всі екземпляри СКБД на різних вузлах підтримують той самий інтерфейс* (типи даних, реалізація запитів, транзакції та ін.), хоча вони необов'язково мають бути копіями того самого програмного забезпечення. Наприклад, якби системи INGRES і ORACLE підтримували офіційний стандарт SQL, то було б можливо організувати їх спільну роботу в контексті розподіленої системи. Інакше кажучи, розподілена система, принаймні деякою мірою, може бути неоднорідною.

Підтримка цієї неоднорідності дуже бажана. Річ у тім, що в реальному світі робота зазвичай організована не тільки на комп'ютерах різних типів і в різних операційних системах, але й за участю різних СКБД. Було б ідеально, якби такі СКБД могли працювати спільно в одній розподіленій системі. Іншими словами, в ідеальній розподіленій системі передбачається підтримка незалежності від СКБД.

Цій досить великій і дуже важливій з практичного погляду темі присвячено окремий розділ.

11.3.4. Незалежність від даних

Незалежність від розміщення. Основна ідея незалежності від розміщення (або прозорості розміщення) досить проста: користувачам не потрібно знати, де фізично зберігаються дані, навпаки, з логічної точки зору користувачам варто було б забезпечити такий режим, за якого створюється враження, що всі дані зберігаються на їх власному локальному вузлі. Ба-

жано забезпечувати незалежність від розміщення, оскільки при цьому істотно спрощуються програми користувачів і термінальна діяльність. Зокрема, це дозволяє виконувати міграцію даних від вузла до вузла без оголошення недійсними будь-яких програм користувача і видів термінальної діяльності. Процес міграції дуже корисний, оскільки дозволяє даним переміщатися по всій мережі у відповідь на зміну вимог до продуктивності.

Незалежність від фрагментації. Фрагментація даних означає, що дані розподіляються на частини так, щоб частини (фізичні фрагменти даних – таблиці) зберігалися в тому місці, де їх частіше використовують. За такої організації багато операцій будуть суто локальними, а обсяг даних, що пересилаються в мережі, знизиться. Фрагментація бажана для підвищення продуктивності системи. Однак з логічного погляду користувача забезпечують таким режимом роботи, за якого дані здаються зовсім не фрагментованими.

Розглянемо відношення співробітників EMP. У системі, де підтримується фрагментація, визначено два фрагменти, показані на рис. 11.6.

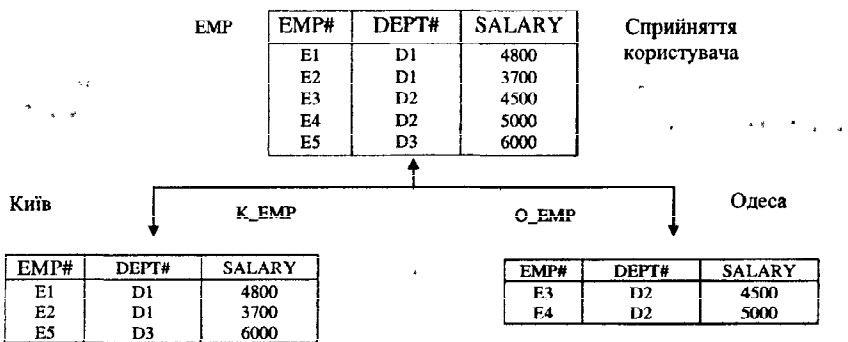


Рис 11.6. Фрагментація відношення EMP

Тепер оптимізатору відомо, що доступ потрібно здійснити тільки до вузла у Києві.

Отже, проблема підтримки операцій для фрагментованих відношень має деяку подібність до проблеми підтримки операцій для об'єднаних і сполучених представлень (фактично, це різні прояви однієї і тієї самої проблеми під час розгляду загальної архітектури системи з різних поглядів). Зокрема, проблема відновлення фрагментованих відношень – це відновлення в об'єднаннях і сполучення представлень. Звідси також випливає, що у разі відновлення заданий кортеж може мігрувати з одного фраг-

мента в інший за умови, що він більше не задовольняє предикат відношення для фрагмента, до якого він належав до відновлення.

Існує два основні типи фрагментації – *горизонтальна* і *вертикальна*, які відповідно пов'язані з реляційними операціями вибірки та проекції. Інакше кажучи, фрагментом може бути будь-яке довільне підлегле відношення, яке можна отримати з вихідного відношення за допомогою операцій вибірки і проекції. При цьому варто враховувати такі припущення:

- усі фрагменти даного відношення незалежні, тобто жоден із фрагментів не може бути виведений з інших фрагментів або мати вибірку чи проекцію, яка може бути виведена з інших фрагментів;
- проекції (фрагменти та їх об'єднання) не мають допускати втрати інформації.

Реконструкцію початкового відношення на основі його фрагментів можна виконати за допомогою операцій сполучення (для вертикальних фрагментів) і об'єднання (для горизонтальних фрагментів). Зверніть увагу, що у випадку об'єднання не потрібно виконувати операцію виключення дублікатів.

Зауваження. Відносно вертикальної фрагментації потрібні додаткові пояснення. Як уже згадувалося, за такої фрагментації не повинна допускатися втрата інформації, оскільки фрагментація відношення EMP на проекції (EMP#, DEPT#) і (SALARY) не буде достовірною. Але у деяких системах збережені відношення мають схований атрибут TI («tupleID» – ідентифікатор кортежу), тобто фізична чи логічна адреса цього кортежу. Атрибут TI є потенційним ключем для даного кортежу, тому якби відношення EMP містило такий атрибут, то воно могло б бути коректно фрагментовано на проекції (TI, EMP#, DEPT#) і (TI, SALARY), оскільки така фрагментація відбувається без втрати інформації.

Саме легкість виконання фрагментації і реконструкції – одна з багатьох причин використання реляційної моделі для розподілених систем. Нарешті, в системі, яка підтримує фрагментацію даних, варто також передбачити підтримку *незалежності від фрагментації* (чи прозорість фрагментації). Іншими словами, з логічного погляду користувачам варто забезпечити такий режим роботи, за якого дані здаються зовсім не фрагментованими. Незалежність від фрагментації (як і незалежність від розміщення) досить важлива, оскільки вона дозволяє в будь-який момент виконати дефрагментацію даних (і фрагменти можуть бути перерозподілені в будь-який момент часу) у відповідь на зміну вимог до продуктивності, причому без потреби відключення будь-яких програм користувача і термінальної діяльності.

Незалежність від фрагментації припускає, що дані будуть подані для користувачів у вигляді логічно комбінованих фрагментів на основі відпо-

відних об'єднань і сполучень. При цьому системний оптимізатор відповідає за визначення фрагментів, до яких слід забезпечити фізичний доступ для виконання будь-якого заданого користувачем запиту. Як приклад припустимо, що для фрагментації користувач задає такий запит:

```
EMP WHERE SALARY>40K AND DEPT#='D1'
```

У цьому випадку оптимізатор системи знатиме з визначень фрагментів (які зберігаються в системному каталозі), що всі результати можуть бути отримані на вузлі в Одесі і немає ніякої потреби здійснювати доступ до вузла в Києві.

Розглянемо цей же приклад ретельніше. Відношення EMP користувач сприймає як подання базових фрагментів O_EMP і K_EMP: EMP=O_EMP UNION K_EMP

Отже, оптимізатор перетворить вихідний запит користувача на такий:

```
(O_EMP UNION K_EMP) WHERE SALARY>40K AND DEPT#='D1'
```

А цей вираз, у свою чергу, можна перетворити до вигляду

```
(O_EMP WHERE SALARY>40K AND DEPT#='D1') UNION
(K_EMP WHERE SALARY>40K AND DEPT#='D1')
```

Із фрагмента L_EMP, який зберігається у каталозі визначення, оптимізатору відомо про те, що другий вираз дасть у результаті порожнє відношення (умова цілісності DEPT#='D1' AND DEPT#='D2' не може бути оцінена як «істина»). Отже, усі вирази в цілому можна звести до такого спрощеного вигляду:

```
O_EMP WHERE SALARY > 40K AND DEPT#='D1'
```

Тепер оптимізатору відомо, що доступ потрібно здійснити тільки до вузла в Одесі.

Незалежність від реплікації. Реплікація даних означає, що деякий фрагмент даних (таблиця) може бути поданий у БД різними копіями (репліками), збереженими на декількох різних вузлах.

Реплікації корисні з двох причин:

1) підвищення продуктивності системи, оскільки додатки можуть працювати з локальними даними без обміну інформацією з віддаленими вузлами;
2) збільшення доступності та надійності, оскільки об'єкт залишається *доступним для обробки доти, доки є хоча б одна його репліка.*

Головний недолік реплікації полягає в тому, що у разі відновлення реплікованого об'єкта всі копії цього об'єкта мають також поновлюватися. Цей недолік називають *проблемою розповсюдження поновлення.*

З логічного погляду користувачі повинні працювати так, ніби дані не є реплікованими.

Існує декілька механізмів виконання реплікації: множинна публікація, централізована публікація і централізована підписка.

Нижче наведено приклад реплікації, схематично показаний на рис. 11.7.

```
REPLICATE K_EMP OK_EMP AT SITE 'Одеса';
REPLICATE O_EMP KO_EMP AT SITE 'Київ';
```

Внутрішні імена реплік OK_EMP і KO_EMP.

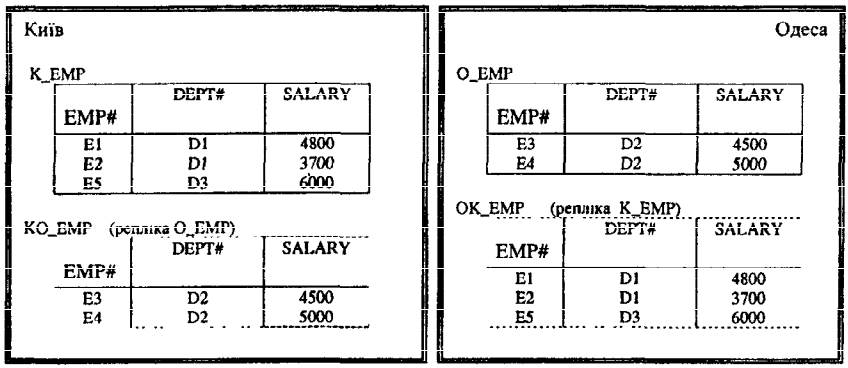


Рис. 11.7. Приклад реплікації

Слід зазначити, що реплікація в розподіленій системі являє собою особливий вид втілення ідеї керованої надлишковості. В ідеальному випадку реплікація, як і фрагментація, має бути «прозорою» для користувача. Інакше кажучи, у системі, у якій підтримується реплікація даних, повинна також підтримуватися незалежність від реплікації (чи прозорість реплікації), тобто користувачі, принаймні, з логічної точки зору, повинні працювати в такому режимі, ніби дані зовсім не репліковані. Незалежність від реплікації (як і незалежність від розміщення і незалежність від фрагментації) дуже бажана, оскільки дозволяє істотно спростити програми користувача і термін їх дії. Зокрема, при цьому дозволяється у будь-який момент створювати і видаляти репліки у відповідь на зміни вимог без відключення програм користувача.

Незалежність від реплікації припускає, що системний оптимізатор відповідає за визначення фізичного доступу саме до тих реплік, які потрібні для виконання заданого запиту користувача.

Приклад реалізації реплікації. База даних «Російські ВНЗ» має один головний сервер, що знаходиться в Москві, і кілька регіональних серверів. Московським сервером є MS SQL Server 6.5 під керуванням Windows NT4.0. Регіональними серверами є або MS SQL Server 6.5 під керуванням Windows NT4.0, або Oracle7(8) під керуванням Windows NT4.0 чи Linux.

У РБД застосовують сценарій централізованої підписки. Адміністратор призначає сервер реплікації та сервер підписки, визначає час, через який має відбуватися реплікація, і набір таблиць, що беруть участь у ній. Також визначаються кілька додаткових параметрів.

У цьому випадку реплікація відбувається відразу після завершення транзакції та поновлення таблиці. Адміністратор із клієнтського місця вносить зміни в таблицю, використовуючи для цього структуровану мову запитів SQL. Після виконання команди COMMIT WORK, що означає успішне завершення транзакції, сервер публікації/реплікації встановлює з'єднання із сервером підписки і перевіряє дані з таблиць, які беруть участь у реплікації, на ідентичність. У разі виявлення нових даних сервер публікації проводить реплікацію нових записів у відповідні таблиці сервера підписки. У випадку виникнення помилки в процесі реплікації (наприклад, розриву мережі) відбувається відкат виконання реплікації, і після відновлення працездатності мережі система автоматично переходить у нормальний стан і процес реплікації повторюється.

Тиражування – окремий випадок реплікації (на прикладі INFORMiX-OnLine DS).

Тиражування – це підтримка на іншій обчислювальній машині всіх копій об'єктів БД. При цьому реалізується прозоре тиражування даних з основного сервера БД на вторинний (чи підтримувальний) сервер, до якого дозволено доступ тільки на читання і який може знаходитися в іншому географічному пункті. У цій термінології сервер, що не бере участі у тиражуванні, називають стандартним.

Головна мета тиражування – забезпечення високої готовності (High Availability Data Replication – HDR). У разі відмови основного сервера вторинному серверу автоматично чи вручну надається статус стандартного з доступом на читання і запис. Прозоре перенаправлення клієнтів за відмови основного сервера не підтримується, але воно може бути реалізоване в межах додатків. Після відновлення основного сервера, залежно від значення параметра конфігурації, вибирають один із двох можливих сценаріїв:

1. Відновленому серверу знову надається статус основного. Вторинний сервер перед поверненням у режим доступу тільки на читання зупиняється, щоб забезпечити від'єднання клієнтів, які здійснювали доступ до запису.
2. Відновлений сервер стає вторинним, а колишньому вторинному, який уже функціонує в режимі читання – запису, надається статус основного; клієнти, підключені до нього, продовжують роботу. Цей сценарій забезпечує безперервний доступ до БД.

Тиражування виконується передачею інформації з журналу транзакцій (логічного журналу) у буфер тиражування основного сервера, звідки вона пересилається в буфер тиражування вторинного сервера. Таке пересилання може відбуватися або в синхронному, або в асинхронному режимі. *Синхронний режим* гарантує повну погодженість БД – жодна транзакція, зафіксована на основному сервері, не залишиться незафіксованою на вторинному, навіть у випадку збою основного сервера. *Асинхронний режим* не забезпечує абсолютної погодженості, але поліпшує робочі характеристики системи.

Дзеркальне відображення, що також є прозорим засобом підтримки високої готовності, забезпечує тільки копіювання дискових областей у межах однієї установки сервера INFORMIX-OnLine DS і захищає тільки від дискових збоїв. Механізм тиражування забезпечує підтримку повної віддаленої копії БД і захищає від усіх видів відмов, включаючи повну відмову однієї з установок.

Крім забезпечення безвідмовної роботи, тиражування дає такі переваги:

- оперативніший доступ до даних для локальних клієнтів вторинного сервера;
- можливість винести додатки користувача переважно на вторинний сервер, де вони виконуються з максимальним використанням даних, не подавлюючи додатків, які виконуються на основному сервері.

11.4. Принципи розподіленої обробки даних

В основу взаємодії прикладних програм – клієнтів і сервера БД, покладено ряд фундаментальних принципів, що визначають функціональні можливості сучасних СКБД у тій частині, яка стосується мережевої взаємодії і розподіленої обробки даних.

11.4.1. Безперервне функціонування

Однією з основних переваг розподілених систем є те, що вони забезпечують вищі надійність і доступність.

Надійність – імовірність того, що система справна і працює в будь-який заданий момент, – підвищується завдяки роботі розподілених систем не за принципом «усе чи нічого», а у постійному режимі, тобто робота системи продовжується (хоч і на більш низькому рівні) навіть у випадку несправності деякого окремого компонента, наприклад окремого вузла.

Доступність – імовірність того, що система справна і працює протягом деякого проміжку часу, – підвищується частково з тієї самої причини, а частково завдяки можливості реплікації даних.

Ці міркування можна застосовувати також для випадку незапланованого вимикання деякого компонента усередині системи, наприклад унаслідок якої-небудь несправності. Незаплановані вимикання вкрай небажані, але їх появи важко уникнути. В ідеальному випадку варто було б зовсім виключити заплановані вимикання, тобто не вимикати систему під час виконання будь-яких операцій, наприклад, додавання нового вузла чи відновлення СКБД на деякому вузлі.

11.4.2. Обробка розподілених запитів

Перевага реляційної моделі в продуктивності. Розглянемо запит «Одержати відомості про постачальників червоних деталей, що знаходяться в Харкові», причому припустимо, що користувач знаходиться в Києві, а дані зберігаються на вузлі в Лондоні. Припустимо також, що такий запит задовольняють *n* постачальників. Якщо система є реляційною, то в запиті будуть міститися два повідомлення – одне із запитом із Києва для Харкова, а інше з результатом запиту, тобто набором із *n* кортежів, що пересилаються з Харкова в Київ.

Якщо система не є реляційною, а працює в режимі послідовної обробки даних, то в запит буде включено $2n$ повідомлень – *n* з Києва в Харків із запитом відомостей про «наступного» постачальника, а також *n* з Харкова в Київ для повернення відомостей про «наступного» постачальника. Цей приклад ілюструє, що *реляційна система за продуктивністю може на кілька порядків перевершувати нереляційну* (принаймні на рівні множин).

Оптимізація розподілених запитів. Мета оптимізації запитів полягає у збільшенні пропускної здатності і поліпшенні часу відгуку за заданих обмежень на комп'ютерне устаткування і програмне забезпечення.

Основна проблема виконання запиту полягає в тому, що для виконання деякого запиту, який охоплює кілька вузлів, існує багато способів переміщення даних по мережі. У цьому випадку надзвичайно важливо знайти найефективнішу стратегію.

Оптимізація обробки запитів більш важлива для розподіленої системи, ніж для централізованої системи. Основна причина полягає в тому, що для виконання запиту, який охоплює кілька вузлів, існує досить багато способів переміщення даних по мережі. У такому випадку надзвичайно важливо знайти найбільш ефективну стратегію. Наприклад, запит на об'єднання відношення *Rx*, що зберігається на вузлі *X*, і відношення *Ry*, що зберігається на вузлі *Y*, міг би бути виконаний за допомогою переміщення:

- відношень *Rx* на вузол *Y*;
- відношень *Ry* на вузол *X*;
- цих двох відношень на третій вузол *Z* і т. д.

При цьому час відгуку може варіюватися від частки секунди до декількох годин. Для того щоб досягти прийнятного часу відгуку, треба розуміти поведінку оптимізатора. Наприклад, шоста версія Oracle включає евристичну, засновану на системі правил оптимізацію, тоді як Oracle7 пропонує обидва види оптимізації: як засновану на системі правил, так і побудовану на аналізі вартості. У будь-якому випадку оптимізація потребує аналізу планів виконання запитів. Для цього Oracle має кілька дуже корисних утиліт: EXPLAI PLAN, TKPROF. Під час виконання розподіленого запиту відбуваються такі процеси.

Декомпозиція запиту: якщо запит звертається до об'єктів, розміщених на кількох комп'ютерах-серверах, то цей запит розбивається на кілька SQL-пропозицій. Кожна пропозиція відсилається на відповідний вузол для виконання.

Сполучення: рядки, що повертаються SQL-пропозиціями, отриманими в результаті декомпозиції, збираються в одному пункті (вузлі мережі), де вони сполучаються. Тими пунктами, де виконується сполучення, можна керувати.

Кореляційні підзапити, використання яких украй невигідно, навіть якщо йдеться не про розподілений запит, оскільки такий підзапит буде оброблятися не одноразово за увесь час виконання вихідного запиту, а швидше за все під час одержання кожного нового рядка вихідного запиту. Якщо підзапит звертається до віддаленої таблиці, то для кожного отриманого рядка вихідного запиту підзапит посилається по мережі на віддалену базу даних для відповідної обробки.

Індекси: локальний вузол створює тимчасові таблиці для даних, отриманих із віддалених вузлів. Як і за звичайних (не розподілених) запитів, індекси не створюються для тимчасових таблиць. Навіть якщо віддалена таблиця має унікальний індекс, він все одно не використовується в операціях сполучення. EXPRESSION, GROUP BY, ORDER BY, DISTINCT обчислюються і виконуються на локальних вузлах. 'LOCAL' належить до того вузла, з яким додаток установлює зв'язок, а 'REMOTE' – до іншого вузла (чи інших вузлів) мережі.

Розглянемо таку ситуацію. Таблиця ORDER розміщена на вузлі LOCAL, має унікальний індекс на order_row і 100 тис. рядків. Таблиця CUSTOMER розміщена на вузлі REMOTE, має унікальний індекс на cust_id і 1 млн рядків. На вузлі LOCAL визначено зв'язок із БД (database link) з REMOTE і створено синонім «CUSTOMER» для таблиці CUSTONER. Наступний запит був виданий додатком, зв'язаним з вузлом LOCAL:

```
SELECT CUST_NAME, ORDER_NUM FROM ORDER, CUSTOMER
WHERE ORDER.CUST_ID = CUSTOMER.CUST_ID;
```

Під час виконання запит був декомпозований так:

```
SELECT CUST_ID, CUST_NAME FROM CUSTOMER;
<== executed at REMOTE.
SELECT CUST_ID, ORDER_NUM FROM ORDER;
<== executed at LOCAL.
```

У результаті 1 млн рядків таблиці CUSTOMER буде послано на вузол LOCAL, і відповідно мережевий трафік зросте. На вузлі LOCAL буде створена тимчасова таблиця для рядків, отриманих із вузла REMOTE. Оскільки звичайні тимчасові таблиці не індексуються, індекс таблиці клієнта не буде використаний. Як альтернативний варіант можна переслати таблицю ORDER на вузол REMOTE, з'єднати на вузлі REMOTE і повернути результат на вузол LOCAL. Це можна зробити, створивши LINK і VIEW на вузлі REMOTE:

```
CREATE DATABASE LINK LOCAL USING Connect-string;
CREATE VIEW CUST_ORDER AS SELECT ORDER_NUM, CUST_NAME
FROM ORDER@LOCAL O, CUSTOMER C WHERE O. CUST_ID =
C.CUST_ID;
```

Тепер може бути виконана така пропозиція вибору:

```
SELECT CUST_NAME, ORDER_NUM FROM CUST_ORDER@REMOTE;
```

У цьому прикладі варто брати до уваги не тільки те, яка таблиця чи скільки рядків будуть пересилатися по мережі, але і витрати на виконання з'єднання і кількість рядків, що повертаються після виконання запиту.

Технологічні поради:

- уникайте з'єднань великих таблиць із різних вузлів;
- на віддалених вузлах по можливості використовуйте опцію 'where clauses';
- аналізуйте й оптимізуйте декомпозицію запитів;
- по можливості застосовуйте тригери БД і збережені процедури;
- використовуйте подання віддалених таблиць для оптимізації функцій GROUP BY, DISTINCT;
- уникайте кореляційних підзапитів, які звертаються до об'єктів на декількох вузлах;
- обґрунтуйте застосування реплікації даних, де це тільки можливо і зручно. Ця ситуація докладно розглянута нижче, на цьому етапі слід зазначити, що на основі деяких правдоподібних пропозицій можна скласти шість різних стратегій обробки такого запиту.

11.4.3. Керування розподіленими транзакціями

Існує два основні аспекти керування обробкою транзакцій, а саме: керування відновленням і керування паралелізмом, кожному з яких у розподілених системах має приділятися підвищена увага.

У розподіленій системі виконання однієї транзакції може бути зв'язане з виконанням коду на декількох вузлах, зокрема вона може виконувати операції відновлення на декількох вузлах. У такому випадку говорять, що кожна транзакція складається з декількох *агентів*, тобто процесів, що виконуються на цьому вузлі за вказівкою заданої транзакції. Системі потрібно вказати, що деякі два агенти являють собою частини однієї і тієї ж транзакції, наприклад для того, щоб уникнути виникнення тупикової ситуації для двох агентів, які містить одна і та сама транзакція.

Керування відновленням. Протокол двофазної фіксації розподілених трансакцій. Для підтвердження того, що в розподіленому оточенні деяка транзакція є атомарною («усе чи нічого»), система повинна переконатися, що агенти заданого набору мають бути всі разом або завершені, або скасовані. Цієї мети можна досягти тільки за рахунок реалізації протоколу двофазної фіксації, суть якого полягає в тому, що виконанням транзакції в РБД керує системний компонент, названий координатором, під керуванням якого працюють незалежні адміністратори ресурсів вузлів мережі. Після виконання запиту на Commit координатор здійснює такий двофазний процес.

Кожен адміністратор ресурсів зберігає всі записи журналу реєстрації поза власними фізичними файлами (поза енергозалежною пам'яттю) реєстрації для своїх локальних ресурсів. Тепер за будь-яких умов адміністратор ресурсу не буде виконувати постійного запису від імені транзакції, а може за потреби передавати усі свої відновлення і скасовувати їх. Якщо примусовий запис пройшов успішно, то адміністратор відповідає координатору OK, інакше – Not OK.

Одержавши відповіді від усіх адміністраторів, координатор насильно заносить записи у свій фізичний файл реєстрації, вказуючи своє рішення щодо всієї транзакції. Якщо усі відповіді були OK, то рішенням буде «прийняти зміни», якщо була хоч одна відповідь Not OK – то «відкинути зміни». Потім координатор інформує кожного адміністратора про своє рішення, і кожен адміністратор згідно з прийнятим рішенням локально фіксує чи анулює транзакцію.

Керування паралелізмом. Для керування паралелізмом у розподілених системах, так само як і в нерозподілених, використовують метод блокування. (У деяких сучасніших системах уже застосовують різні способи керування, однак у більшості систем все-таки продовжують використовувати метод блокування.)

У розподіленій системі запити на перевірку, установлення і зняття блокувань є повідомленнями (передбачається, що розглянутий об'єкт знаходиться на віддаленому вузлі), що спричиняє додаткові накладні витра-

ти. Розглянемо, наприклад, транзакцію *T*, що має потребу у відновленні об'єкта, який має репліки на *n* віддалених вузлах. Якщо кожен вузол керує блокуваннями для об'єктів, збережених на цьому вузлі (як це було б у разі дотримання локальної автономії), то для найпростішого способу керування паралелізмом треба було б принаймні $5n$ повідомлень:

n запитів на блокування,
n дозволів на блокування,
n повідомлень про відновлення,
n підтверджень,
n запитів на зняття блокування.

Звичайно, можна легко удосконалити цей спосіб, використовуючи підтвердження, вкладені в блок даних зворотного напрямку. У такий спосіб можуть комбінуватися запит на блокування і повідомлення про відновлення, а також дозвіл на блокування і підтвердження. Але навіть у такому випадку загальний час відновлення може бути на кілька порядків більшим, ніж у централізованій системі.

Звичайний підхід до цієї проблеми полягає в прийнятті стратегії первинної копії, що стисло подано вище. Для об'єкта R вузол, що містить первинну копію об'єкта R, керуватиме всіма операціями блокування R (первинні копії різних об'єктів у загальному випадку можуть бути розміщені на різних вузлах). За такої стратегії множина усіх копій об'єкта з метою блокування може розглядатися як єдиний об'єкт, а загальна кількість повідомлень буде знижена від $5n$ до $2n + 3$ (один запит на блокування, один дозвіл на блокування, *n* відновлень, *n* підтверджень і один запит на зняття блокування). Однак слід звернути увагу, що у разі такого рішення знову втрачається автономність, тобто транзакція може виявитися неуспішною, якщо первинна копія недоступна (навіть за використання транзакції тільки для читання), а локальна копія доступна. (Для блокування первинної копії потрібні не тільки операції відновлення, а також й операції вилучення. Отже, неприємний побічний ефект використання стратегії первинної копії полягає в зниженні продуктивності та доступності даних як для операцій вилучення, так і для операції відновлення.)

Іншою проблемою блокування у розподіленій системі є те, що воно може призвести до глобального тупика, що охоплює два чи більше вузлів. Приклад виникнення такого тупика наведено на рис. 11.8.

Кожна з транзакцій виконується на кожному з вузлів у три етапи: блокування – виконання – розблокування.

Транзакція T1 стартувала на X, заблокувала вузол X і виконується, але для завершення їй потрібно чекати завершення T1 на вузлі *B*.

Транзакція T1 стартувала на вузлі *B*, але той заблокований T2.

Транзакція Т2 стартувала на вузлі *B*, заблокувала вузол Y і виконується, але для завершення їй потрібно чекати завершення Т2 на вузлі X.

Транзакція Т2 стартувала на вузлі X, але той заблокований Т1.

Агент транзакції Т1 на вузлі X очікує, коли закінчиться виконання транзакції Т1 на вузлі Y.

Агент транзакції Т1 на вузлі *B* очікує, коли агент транзакції Т2 зніме блокування на вузлі Y.

Агент транзакції Т2 на вузлі *B* очікує, коли закінчиться виконання транзакції Т2 на вузлі X.

Агент транзакції Т2 на вузлі X очікує, коли агент транзакції Т1 зніме блокування на вузлі X.

Тупикова ситуація!

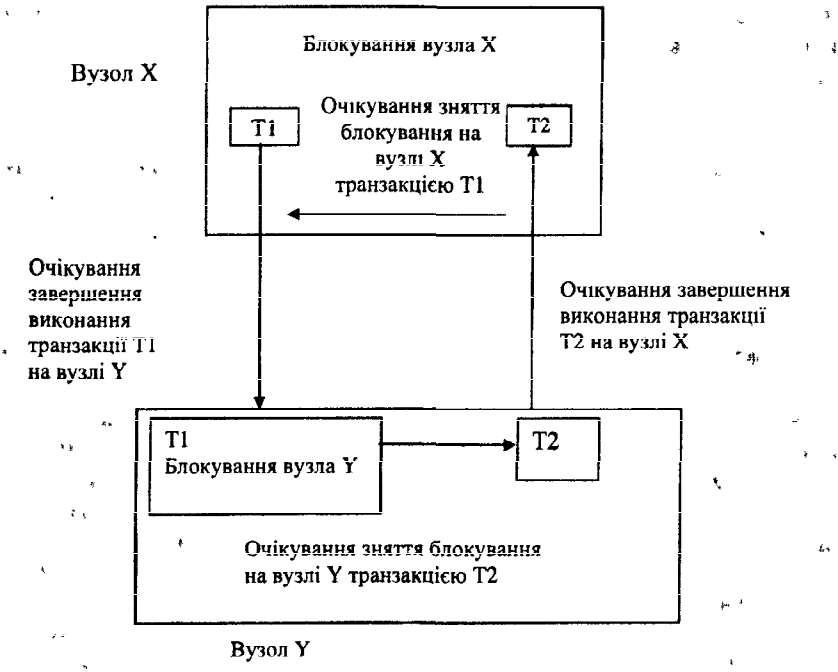


Рис. 11.8. Схема виникнення тупикової ситуації під час паралельного виконання транзакцій

Проблема тупика такого типу полягає у тому, що жоден із вузлів не може зайти у тупик, використовуючи тільки інформацію, зосереджену на цьому вузлі. Інакше кажучи, у локальних діаграмах чекання немає ніяких циклів, але вони з'являться у разі об'єднання локальних діаграм у глоба-

льну діаграму чекання. Звідси випливає, що виявлення глобальних тупиків зв'язано зі збільшенням накладних витрат, оскільки для цього потрібно додатково об'єднати окремі локальні діаграми.

11.4.4. Автоматичне перетворення форматів даних

Як тільки кілька комп'ютерів різних моделей під керуванням різних операційних систем з'єднуються в мережу, відразу виникає питання про узгодження форматів подання даних. Дійсно, у мережі можуть бути комп'ютери, що відрізняються розрядністю (16-, 32- і 64-розрядні процесори), послідовністю байтів у слові, поданням чисел із плавучою точкою і т. ін. Завдання комунікаційного сервера полягає в тому, щоб на рівні обміну даними забезпечити узгодженість форматів між віддаленим і локальним вузлами для того, щоб дані, витягнуті сервером із бази на віддаленому вузлі і передані по мережі, були правильно витлумачені прикладною програмою на локальному вузлі.

11.4.5. Автоматична трансляція кодів

У неоднорідному комп'ютерному середовищі під час взаємодії клієнта і сервера виникає також задача трансляції кодів. Сервер може працювати з однією кодовою таблицею (наприклад, EBCDI), клієнт – з іншою (наприклад, ASCII), при цьому відбувається неузгодженість трактування кодів символів. Тому якщо на локальному вузлі використовується одна кодова таблиця, а на віддаленому – інша, то для передачі запитів по мережі й одержання відповідей на них треба забезпечити трансляцію кодів. Вирішення цієї задачі також покладено на комунікаційний сервер.

Ми розглянули взаємодію однієї пари *клієнт – сервер*. Однак у реальній практиці сервер БД має обслуговувати одночасно безліч запитів від клієнтів. Отже, в один момент часу таких пар може бути кілька. І всі проблеми взаємодії, про які йшлося вище, комунікаційний сервер має вирішувати для всіх цих пар. Системи з архітектурою «один до одного» для одночасного обслуговування сервером БД декількох клієнтів змушені завантажувати окремий комунікаційний сервер для кожної пари *клієнт – сервер*. У результаті навантаження на операційну систему збільшується, різко зростає загальна кількість її процесів, що збільшує витрати обчислювальних ресурсів. Це один із недоліків архітектури «один до одного». Саме тому для сучасних розподілених СКБД важливо мати багатопотоковий комунікаційний сервер, що бере на себе задачі мережевої підтримки безлічі клієнтів, які одночасно звертаються до сервера. На кожному вузлі мережі він підтримує множину пар з'єднань *клієнт – сервер* і дозволяє існувати одночасно множині незалежних сеансів роботи з БД.

11.4.6. Міжоперабельність (шлюзи)

Для забезпечення незалежності від СКБД потрібно, щоб СКБД на різних вузлах підтримували однаковий інтерфейс. Розглянемо приклад: на вузлі X функціонує СКБД INGRESS, а на вузлі B СКБД Oracle, хоча в загальному випадку можна розглядати поєднання різних СКБД.

Припустимо, що деякий користувач U на вузлі X бажає переглянути дані з БД на вузлі X і вузлі Y. За визначенням він є користувачем INGRESS, а тому робота з РБД для нього має бути схожа на роботу з БД INGRESS. Ця проблема вирішується в такий спосіб: система INGRESS повинна мати спеціальну надбудовану програму – *шлюз*, яка взаємодіє із системою Oracle так, щоб створювалося враження, ніби робота Oracle виглядає так само, як і робота INGRESSI.

Шлюз має виконувати такі функції:

- реалізацію протоколу обміну інформацією між системами INGRESS і Oracle з урахуванням форматів даних, які використовують у різних системах;
- обробляти довільні незаплановані твердження мови SQL для Oracle, тобто має містити динамічну підтримку мови SQL;
- взаємне відображення типів даних між системами INGRESS і Oracle (наприклад, велика різноманітність способів збереження дати і часу);
- переклад діалекту мови SQL для INGRESS на діалект мови SQL Oracle;
- відображення інформації пересилається назад від системи Oracle до системи INGRESS;
- конвертація каталогу Oracle у відповідну структуру користувача INGRESS, щоб йому була зрозуміла структура БД Oracle;
- підтримувати протоколи двофазної фіксації транзакцій для обох систем;
- підтримувати потрібне блокування даних.

11. 5. Особливості проектування систем з розподіленими базами даних

11.5.1. Процес проектування інтегральних схем

Процес проектування ІС – це процес побудови такого опису цієї системи, який уможливлює її вивчення та створення. Такі описи називають поданнями системи.

Існують такі види подань системи:

1. Подання з погляду користувача. (Що робить система?)

2. Подання з погляду проектувальника. (З чого складається і як побудована система?)
3. Подання з погляду процесів. (Як поводиться система?)
4. Подання з погляду реалізації. (З яких кінцевих частин складається система (файли запуску, файли ініціалізації)?)
5. Подання з погляду розміщення. (Де встановлюються конкретні частини системи?)

Спроектувати систему – означає створити 5 цих подань.

Усі види подань у процесі проектування подаються у вигляді конструкцій деякої мови (у певній нотації).

Існує кілька мов проектування. Ці мови, як правило, графічні, тобто елементами мови є зображення. Більшість мов проектування мають інструментальну підтримку, що автоматизує процес проектування.

Наприклад:

1. BPWI, ERWI – програмні середовища. Дозволяють створити проект системи.
2. SilverRun:
   а) BMP – моделює процеси;
   б) ERX – будує схему *сутність – зв'язок*;
   в) RDM – будує концептуальну схему БД;
   г) WRM – розробляє форми додатків.

Для проектування інформаційних систем широко використовують мову UML – уніфіковану мову візуального моделювання. Це мова для візуалізації, специфікації, конструювання і документування програмних систем.

Мова UML під час розробки об'єктно-орієнтованих систем використовує власну стандартизовану нотацію. Для автоматизації візуального проектування систем мовою UML використовують :

- програмний продукт Rational Rose;
- програмний продукт Visual UML, який поставляють разом із системою DELPHI.

Візуальним моделюванням системи називають процес графічного подання системи за допомогою деякого стандартного набору графічних символів. Основною метою візуального моделювання є організація спілкування між користувачами, розробниками, аналітиками, тестувальниками та менеджерами. Діаграма показує взаємодію між користувачем і систе-

мою, взаємодію об'єктів усередині системи. Кожна зацікавлена сторона бере з моделі потрібну для себе інформацію:

- аналітики – склад об'єктів і взаємодію між ними;
- розробники – які об'єкти треба створювати і що вони мають робити;
- тестувальники – аналіз взаємодії між об'єктами і що потрібно тестувати;
- менеджери – як система улаштована, як організувати служби та їх взаємодію;
- керівники – як потрібно організувати роботу організації в цілому.

Мовою UML подається система у вигляді набору діаграм, на яких зображені сутності, зв'язані відношеннями. Різні автори називають ці діаграми по-різному.

Сутності – абстракції, що відображають об'єкти предметної області та системи (рис. 11.9).

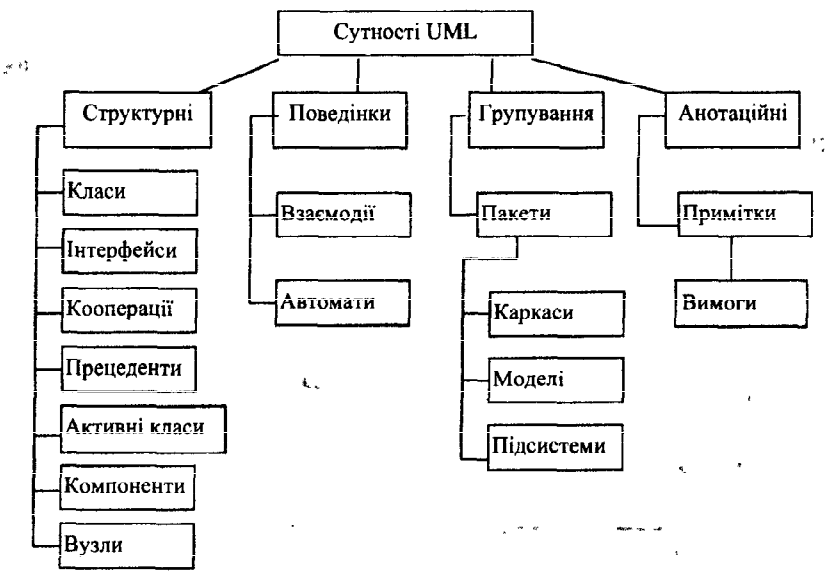
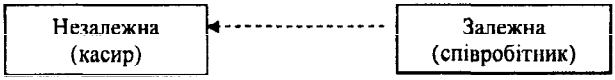


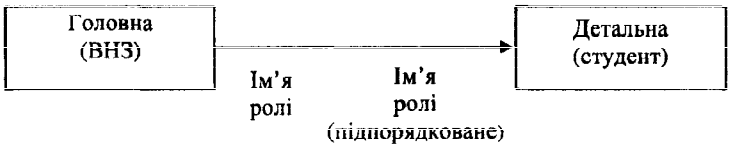
Рис. 11.9. Сутності UML

Відношення – відображення залежностей між об'єктами. У мові UML визначено чотири типи відношень між елементами.

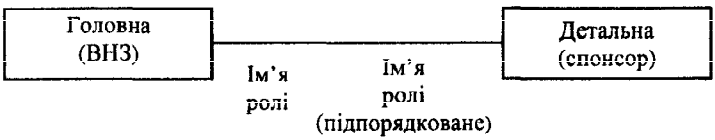
Відношення залежності – таке відношення між сутностями, за якого зміна стану першої (незалежної) впливає на стан іншої (залежної) сутності:



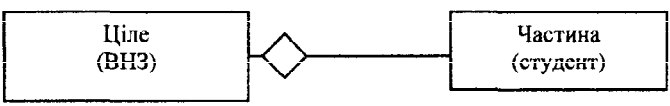
Відношення асоціації – структурне відношення, що описує зв'язок, тобто з'єднання між об'єктами (з різними типами зв'язків). На основі асоціацій виконується навігація за елементами відношення:



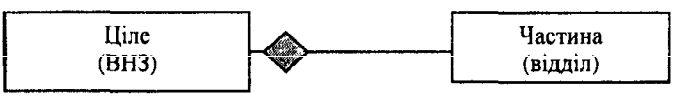
Можлива асоціація з подвійним напрямом навігації:



Відношення узагальнення (агрегування) – відношення показує, що один елемент (дочірній, частина) є структурною частиною іншого (батьківського, цілого):



Можливе композитне агрегування. Сутність є частина цілого і має з ним єдиний час життя:



Відношення реалізації – відношення показує, що один елемент визначає контракт (зобов'язання), а інший виконує цей контракт:

Діаграми UML. Мова UML передбачає використання сукупності діаграм (рис. 11.10). Наведемо загальний опис цієї сукупності. Діаграми будуються у вигляді зв'язаного графу, у якому сутності – вершини графу, а відношення – дуги графу.

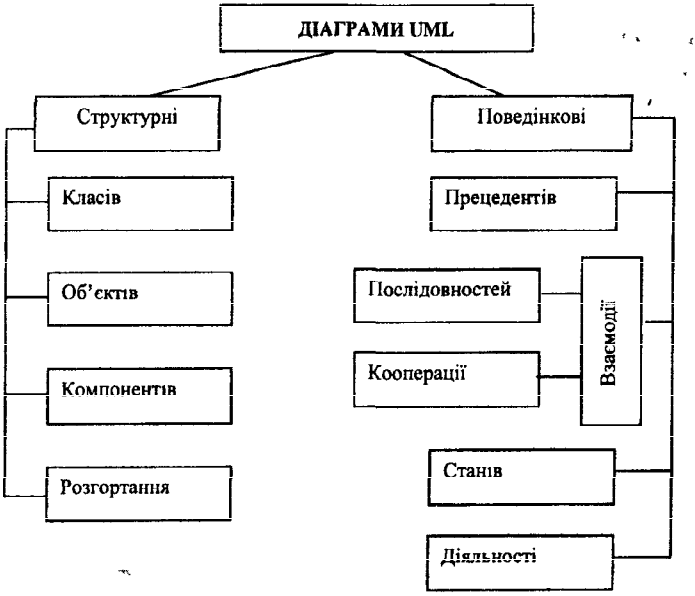


Рис. 11.10. Діаграми UML

11.5.2. Формування архітектури інформаційної системи

Архітектура інформаційної системи описується п'ятьма видами (поданнями) системи, кожний з яких є однією з можливих проекцій організації та структури системи і відповідає окремому аспекту її функціонування. Наведемо ці подання:

1. Вид з погляду *прецедентів використання*. Призначений для опису системи: як її сприймають кінцевий користувач, аналітик та тестувальник системи. У мові UML статичні аспекти цього виду подано діаграмами прецедентів, а динамічні – діаграмами взаємодій, станів і дій.

2. Вид з погляду *проектування*. Призначений для опису функціональних вимог до системи, тобто функцій, які вона має надавати користувачу. У мові UML статичні аспекти цього виду подано діаграмами класів і об'єктів, а динамічні – діаграмами взаємодій, станів і дій.

3. Вид з погляду *процесів*. Призначений для опису механізмів, що виконуються паралельно, і проблем їх синхронізації. У мові UML статичні аспекти цього виду подано діаграмами активних класів і об'єктів, а динамічні – діаграмами взаємодій, станів і дій.

4. Вид з погляду *реалізації*. Призначений для опису компонентів і файлів, що використовують для складання і кінцевої конфігурації системи. У

мові UML статичні аспекти цього виду подано діаграмами компонентів, а динамічні – діаграмами взаємодій, станів і дій.

5. Вид з погляду *розгортання системи*. Призначений для опису топології апаратних засобів, на яких виконується система. У мові UML статичні аспекти цього виду подано діаграмами розгортання, а динамічні – діаграмами взаємодій, станів і дій.

11.5.3. Основні діаграми мови UML, які використовують для проектування програмних систем із розподіленими базами даних

Діаграма прецедентів. На діаграмі прецедентів подано користувачі (актори) системи і варіанти використання системи (прецеденти), а також зв'язки між ними.

Таке подання – це загальний погляд на систему, незалежний від способу її реалізації. Головна мета подання – відобразити, що саме система може виконувати, а не те, як вона це буде робити. Подання містить:

- акторів (ACTORS) – діючі особи, що працюють із системою;
- прецеденти – (USE CASE) – варіанти використання, функції верхнього рівня, які система надає користувачу;
- зв'язки між акторами і варіантами використання;
- документацію про варіанти використання, що уточнює опис процесів, потрібних для виконання функцій системи, включно з процедурами обробки помилок.

Наведемо приклад діаграми на рис. 11.13.

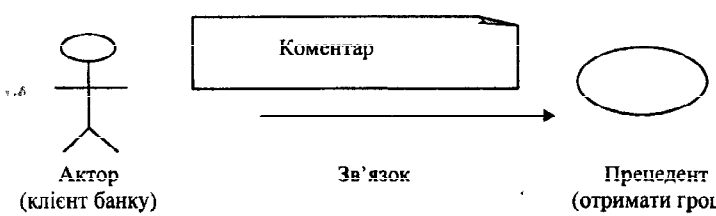


Рис. 11.11. Приклад діаграми прецедентів

Для елементів діаграми прецедентів задають відповідні специфікації.

Специфікації акторів: ім'я актора, кількість екземплярів, стереотип.

Специфікації прецедентів: ім'я прецеденту, опис, стереотип.

Мова UML передбачає використання декількох типів зв'язків між елементами діаграми прецедентів, а саме:

- зв'язок комунікації, тобто зв'язок між актором і прецедентом; напрямок стрілки визначає елемент, що є ініціатором комунікації;
- зв'язок використання – зв'язок, що показує, як один прецедент завжди використовує інший;
- зв'язок розширення – зв'язок, що показує, як один прецедент періодично використовує інший;
- зв'язок узагальнення актора.

Діаграми взаємодій і послідовностей. Діаграми взаємодії використовують для моделювання динамічних аспектів поведінки системи та відображають множину об'єктів, відношень між ними і повідомлення, якими вони обмінюються. Існують два види таких діаграм – діаграми послідовностей і діаграми кооперації.

На *діаграмі послідовностей* (рис. 11.11) увагу акцентовано на послідовності впорядкування повідомлень у часі. Графічно така діаграма являє собою таблицю, у якій об'єкти розміщено горизонтально, а повідомлення в порядку зростання часу – вертикально.

На *діаграмі кооперації* (рис. 11.12) показано структурну організацію об'єктів, що приймають або відправляють повідомлення. Графічно таку діаграму зображують у вигляді графу.

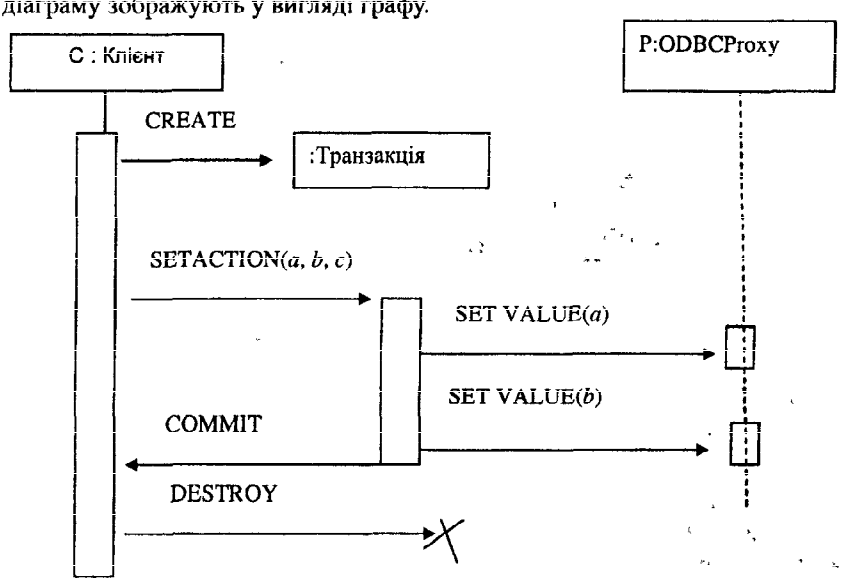


Рис. 11.12. Діаграма послідовностей повідомлень

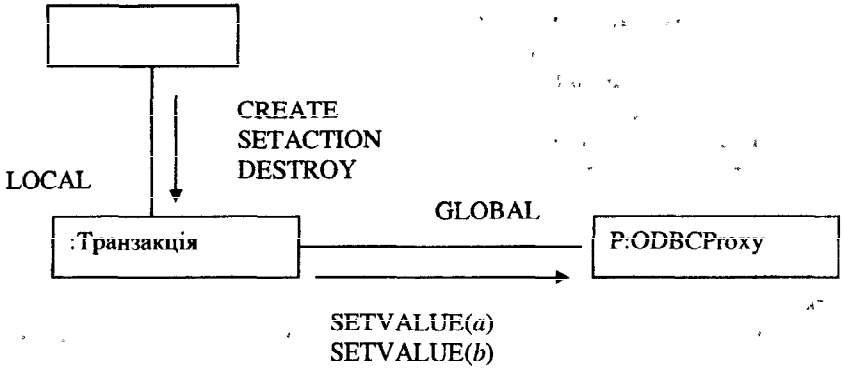


Рис. 11.13. Діаграма кооперації

Діаграми класів. На діаграмі взаємодії знайдемо об'єкти предметної області. Розглядаючи об'єкти більш узагальнено, можна виділити множину об'єктів, які мають однакові структуру й поведінку. Таке виділення називають типізацією, у результаті якої формують класи – абстракцію, що відображає певний набір об'єктів.

У мові UML для кожного проекту будується діаграма класів (рис. 11.14). Класи, а також зв'язки між класами можуть бути з різним рівнем деталізації.

Рис. 11.14. Діаграма класів

Створюючи клас, для нього визначають специфікації:

1) ім'я (name) класу;
2) область видимості (visible): public, private, package; множинність (cardinality) – зазначення кількості екземплярів класу;
3) ємність пам'яті, що виділяється під екземпляр класу (space);
4) стійкість класу: стійкий, тимчасовий.

Зі статичних класів формується БД програмної системи. І надалі множина статичних класів породжує концептуальну схему БД. Паралелізм: sequential (послідовний), synchronize (синхронний).

Стереотип класу:

1) <Boundary> – граничний клас для спілкування з іншими системами;

2) <Entity> – клас сутності, статичний клас; із таких класів створюється БД;
3) <Control class> – керувальний клас, що породжує потоки керування іншими класами.

Мова UML дозволяє користувачу визначати власні стереотипи, якщо це потрібно. Із класом пов'язана множина операцій над елементами класу. Під час програмування створюють методи класу.

Специфікація операцій:

1) значення, що повертається;
2) список параметрів;
3) видимість операції;
4) сигнатура (синтаксичний прототип);
5) час виконання;
6) паралелізм;
7) посада та передумова;
8) семантика (текстовий опис логіки операції).

Зв'язки між класами. Існує 4 види зв'язків між класами:

1. Зв'язок асоціації – смисловий зв'язок, що дозволяє забезпечити процес навігації (зв'язок типу *головна – детальна*). Ці зв'язки припускають наявність атрибутів зв'язку. Позначають їх просто лінією.
2. Зв'язки залежності свідчать про те, що екземпляри залежного класу залежать від екземплярів незалежного класу. Лінія зв'язку прямує від залежного класу до незалежного.
3. Зв'язок агрегації – зв'язок типу *ціле – частина*.
4. Зв'язок узагальнення (успадкування) – зв'язок між класами, один із яких батько, а інший спадкоємець.

Для кожного зв'язку також указують специфікації: ім'я зв'язку, стереотип, рольове ім'я, видимість, обмеження (описують, які саме екземпляри класу можуть брати участь у зв'язку).

Діаграма станів. Протягом життєвого циклу об'єкт знаходиться в різних станах, що змінюють один одний. Наприклад, об'єкт *співробітник* може знаходитись у таких станах: прийом на роботу, проходження випробного терміну, виконання роботи, перебування на лікарняному, перебування у відпустці, звільнення. У кожному з цих станів об'єкт має характерну для стану поведінку. Для моделювання життєвого циклу об'єкта використовують *діаграму станів* (рис 11.15).

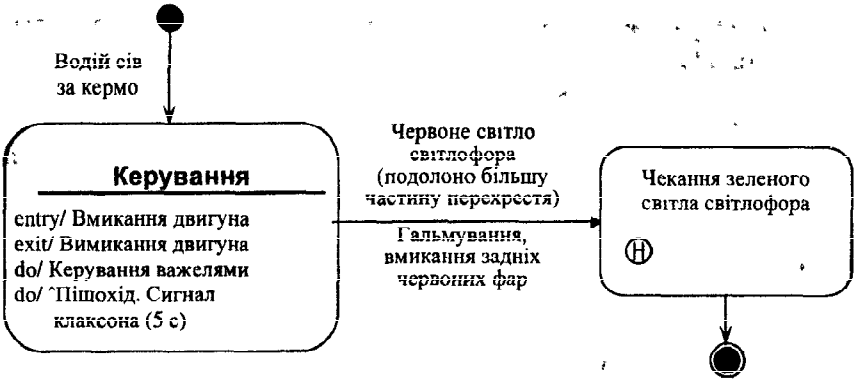


Рис. 11.15. Діаграма станів

Діаграма станів підказує програмісту алгоритм обробки об'єктів.

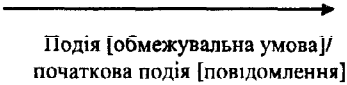
Діаграма починається спеціальною позначкою початкового стану – це стан об'єкта, у якому він знаходиться після створення.

Кожен стан має :

- ім'я (Name);
- діяльність (do: Activity) – поведінку об'єкта під час певного стану, яка не виводить його з цього стану; якщо діяльність переривається, то об'єкт виходить зі стану;
- вхідну дію, подію (entry: Event) – дію, з якої починається стан;
- вихідну подію (exit: Event) – подію, якою закінчується стан і яка призводить до виходу зі стану;
- *відправлення повідомлення* (do: ^Target.*Message*(Param)) *іншим* об'єктам (як правило викликає операції цього об'єкта).

Діаграма станів завершується кінцевим станом – станом об'єкта перед його видаленням:

Крім того, на діаграмі станів відображаються переходи об'єктів в інші стани. Переходом називають реакцію об'єкта на зовнішні події, що переводять його в інші стани (позначається стрілкою):



Специфікація переходу:

1. Event – *власне подія, що викликає перехід. Наприклад,* червоне світло світлофора.

2. *Обмежувальні умови*, тобто умови, за яких стан не зміниться, незважаючи на те, що подія переходу відбулася. Наприклад, червоне світло світлофора → керування закінчується → починається стан чекання іншого сигналу. Обмежувальна умова – подолана більша частина перехрестя.
3. *Початкова подія переходу*, наприклад гальмування.
4. Message – повідомлення, що посилається під час переходу. Наприклад, увімкнення задніх червоних фар.

Діаграма компонентів. Для розподілених систем, щоб відобразити розміщення частин системи на різних технічних пристроях, будують діаграми компонентів (рис. 11.16).

Компонент – фізична частина системи, яка може бути замінена, відповідає інтерфейсу і забезпечує його реалізацію. Компоненти не є абстракціями і тим самим відрізняються від класів.

Вузол – елемент фізичної системи, яка є обчислювальним ресурсом здебільшого комп'ютером.

На діаграмі зображають компоненти і класи. Залежності від інструментального середовища проектування і програмування набір компонентів різний. Мова UML припускає побудову діаграми розміщення з використанням елементів діаграм, показаних на рис. 11.17.

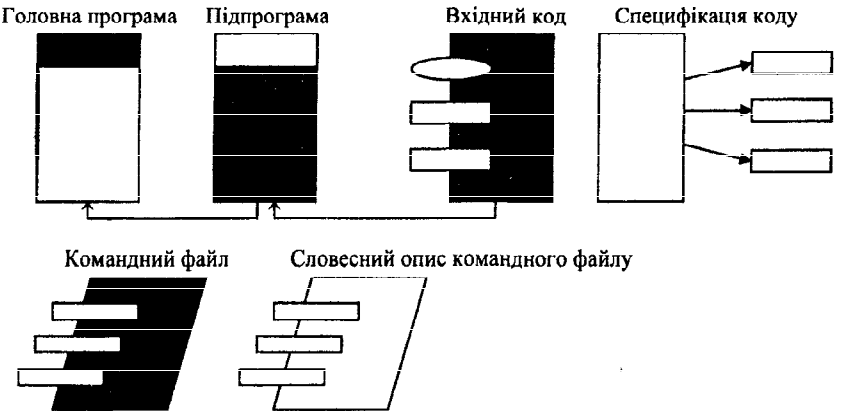


Рис. 11.16. Діаграма компонентів

Діаграма ілюструє зв'язки між компонентами – однотипні зв'язки залежності. Компонент А залежить від компонента В, якщо для роботи компонента А потрібний компонент В. Компонент А не транслюється без компонента В.

Для кожного елемента задають специфікації, найважливіші серед яких:

- стереотип;
- мова;
- адреса, місцезнаходження.

Діаграма компонентів потрібна для чіткого уявлення про розміщення системи. Це завдання стає дуже важливим та актуальним, якщо команда розробників часто змінюється. Друга причина – наявність варіантів реалізації системи. Основна проблема – чіткість збереження версій та підбір відповідних назв.

Діаграма розміщення проекту (діаграма розгортання). На діаграмі розгортання зображається розміщення компонентів системи на вузлах системи. Цю діаграму будують переважно для розподілених систем, вона зображає розміщення частин системи на різних технічних пристроях. Ці пристрої здебільшого або комп'ютери локальної мережі, або окремі мережеві пристрої (наприклад, мережеві принтери).

Зв'язки між елементами діаграми бувають двох видів:

- фізичні канали локальної мережі;
- інтернет-посилання, якщо система глобальна.

Для кожного елемента діаграми вказують специфікацію:

- *стереотип (ключове слово);*
- найважливіші технічні вимоги вузлів;
- адресна частина (місцезнаходження).

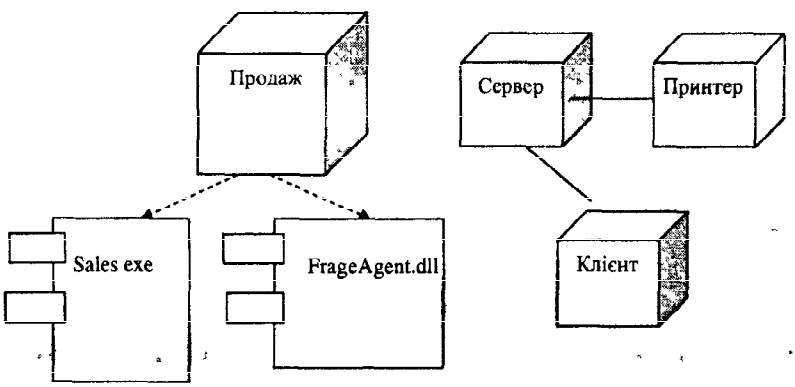


Рис. 11.17. Діаграма розміщення

Крім технічних пристроїв на діаграмах розміщення зображують потоки керування (наприклад, виконувані або пакетні командні файли).

Специфікація процесів:

- стереотип;
- пріоритетність.

11.5.4. Етапи проектування інформаційних систем

Побудову моделі виконують за такими етапами:

1. Сформулювати призначення системи однією пропозицією, що стисло і точно вказує на мету створення системи.
2. Зробити декомпозицію призначення за функціональним критерієм. Це є відповіддю на запитання, які функції має виконувати система для реалізації свого призначення. Не забути службові функції (наприклад, сервісне обслуговування, адміністрування):
   - визначити, які саме види системи треба подати в моделі (як правило, це вищезазначені основні 5 видів, можливі та додаткові);
   - вирішити, які діаграми потрібно створити і які елементи на них відобразити;
   - побудувати діаграми:
     а) діаграму прецедентів (діаграму варіантів використання);
     б) для кожного прецеденту побудувати документ – потік подій;
     в) кожний із потоків зобразити на діаграмах взаємодії: і на діаграмі послідовності повідомлень, і на діаграмі кооперації (об'єктів і повідомлень);
     г) розглядаючи діаграму взаємодії, виділити множину об'єктів, що мають однакову структуру і поведінку; сформувати класи; побудувати діаграму класів;
     д) для найважливіших об'єктів побудувати діаграму станів.
3. Відповідно до отриманої моделі системи побудувати відповідну концептуальну схему бази даних за відповідною нотацією.
4. Для відображення фізичного збереження системи побудувати діаграму компонентів.
5. Приступити до реалізації системи, тобто до побудови інтерфейсу користувача, розробки та заповнення бази даних, розробки та реалізації алгоритмів обробки даних.

11.6. Розподілена система екологічного моніторингу

11.6.1. Загальна постановка задачі

Для зниження ризику захворювань населення потрібна ефективна система контролю і керування екологічним станом навколишнього середовища. Основою системи є процес контролю за станом навколишнього се-

редовища на основі мережі регіональних автоматизованих систем моніторингу, які об'єднують у систему глобального керування екологічною ситуацією в країні.

Система передбачає реалізацію ряду напрямів моніторингу екологічного середовища, кожен із яких є її складовою частиною.

Забруднювання. Реалізація напряму передбачає:

- проведення класифікації екологічно небезпечних забруднювачів (ЕНЗ) та створення відповідних класифікаторів;
- вивчення зв'язків *виробництво – забруднювачі*;
- визначення нормативів ГДК для складових навколишнього середовища: атмосфери, гідросфери, літосфери;
- збереження інформації про забруднення довкілля;
- призначення заходів нейтралізації негативного впливу.

Здоров'я. Основою напряму є:

- визначення методів і моделей оцінки ризику генетичних аномалій та моделей розвитку віддалених наслідків для генофонду країни;
- розробка теоретичних методів та практичних заходів реабілітації здоров'я у випадках екологічних аномалій, включаючи медичний, соціальний та економічний аспекти;
- розробка методів визначення критеріїв оцінки здоров'я населення з розподіленням на групи ризику за медичними ознаками;
- дослідження впливу екологічних чинників на здоров'я людини.

Економіка. Напрям передбачає:

- створення макроекономічних моделей і методики оцінки збитків унаслідок забруднення навколишнього середовища;
- визначення джерел компенсації екологічних втрат;
- визначення моделі оцінки вартості витрат на проведення екологічних та медичних заходів щодо ліквідації екологічних аварій, реабілітації екологічного стану навколишнього середовища, профілактики та реабілітації здоров'я.

Право. Напрям передбачає:

- визначення правових основ для проведення засобів соціального захисту:
  а) покарання порушників;
  б) проведення профілактичних заходів;
  в) компенсації екологічних втрат;
- розгляд питань екологічного правового регулювання між організаціями.

11.6.2. Загальний опис інформаційної системи

Проект системи формувався на основі інформаційних діалогових моделей різних категорій користувачів деякої віртуальної глобальної екологічної інформаційної системи. У цих моделях розглядалися різні варіанти організації зв'язку, діалогу користувачів із системою і між собою під час вирішення загальних проблем, ставилися завдання, актуальні для різних категорій фахівців.

Відповідно до загальної концепції еколого-економічного моніторингу (ЕЕМ) навколишнього середовища інформаційна система складається з множини робочих місць фахівців, що спостерігають за станом навколишнього середовища та забезпечують проведення засобів мінімізації генетичного ризику для населення, яке потрапило в зону забруднення навколишнього середовища мутагенними речовинами. Передбачається, що система забезпечуватиме підтримку робочих місць таких спеціалістів.

Еколог – спостерігач за станом зони забруднення, який вирішує завдання формування інформаційної картини про стан контрольованої зони екологічного забруднення, оцінює рівень небезпеки (зокрема, за ГДК), прогнозує розвиток екологічної обстановки в зоні на найближче майбутнє і на більш віддалений період, *формує перелік заходів для нейтралізації наслідків забруднення,* наприклад, утилізації, поховання і нейтралізації ЕНЗ, фіксує і документує відповідно до законодавства про охорону навколишнього середовища різні екологічні надзвичайні ситуації, ініціює позови до порушників екологічної безпеки.

Лікар – вирішує завдання формування БД про стан здоров'я населення, ступеня ризику захворювань загального характеру, пов'язаних із порушеннями імунних систем і генетичних патологій нинішнього та майбутніх поколінь. Він користується інформацією про екологічний стан контрольованої зони і прогнозом його розвитку, клінічною картиною здоров'я населення різних груп ризику, *пропонує комплекс медичних і соціальних заходів щодо профілактики захворювань та реабілітації здоров'я.*

Юрист – вирішує завдання формування БД про множину законодавчих актів, присвячених правовому регулюванню проблем, що виникають унаслідок забруднень довкілля. Він користується інформацією про екологічний стан контрольованої зони і прогнозом його розвитку, *пропонує комплекс юридичних заходів щодо покарання порушників чинного законодавства і потрібних компенсаційних та інших заходів.*

Економіст – шукає оптимальний за вартістю варіант виконання заходів, рекомендованих екологами і медиками. Для цього він повинен мати відомості про ресурси, потрібні для виконання запропонованих заходів, та про ресурси, які знаходяться в розпорядженні відповідних міністерств (Міністерства надзвичайних ситуацій, Міністерства охорони здоров'я), місцевих адміністрацій, військових підрозділів та передбачені планами для залучення до подібних робіт тощо. Результатом його роботи є визначення переліку додаткових ресурсів для проведення заходів, пропонованих фахівцями, формування висновків про вартість заходів, можливих економічних витрат, про джерела компенсації екологічних збитків.

Особа, що приймає рішення – керівник, який приймає рішення щодо формування комплексного оптимального плану проведення заходів у зоні екологічного забруднення на основі аналізу рекомендацій предметних фахівців – еколога, медика, юриста та економіста, аналізує сформовану ситуацію і приймає остаточні рішення. Особа, що приймає рішення, може визначати обмеження на ресурси і терміни виконання робіт та вимагати повторної оцінки ситуації від предметних фахівців.

Можливе використання дворівневої системи прийняття рішень:

- регіональний рівень – для вирішення завдань, для яких достатньо ресурсів регіону;
- державний рівень – для вирішення завдань, які потребують мобілізації ресурсів державного резерву.

Керівник отримує інформацію про запропоновані заходи, потрібні ресурси та пріоритети, які визначають послідовність проведення цих заходів.

Адміністратор системи – фахівець, обов'язки якого полягають у забезпеченні функціонування РБД інформаційної системи та системи захисту інформації.

Сукупність наведених робочих місць створює комп'ютерну систему моніторингу та прогнозування генетичного ризику.

В основі функціонування інформаційної системи знаходиться інформаційна модель зони екологічного забруднення. Щодо типу забруднення розглядають локальні та комплексні зони.

Локальна зона – інформаційна модель забруднення частини навколишнього середовища одним забруднювачем. Її однозначно визначають за назвою забруднювача, діапазоном концентрації забруднювача, замкнутою межею (полігоном) та типом середовища (літосферою, гідросферою або атмосферою).

За алгоритмом визначення межі зони виділяють:

- статичні зони – зони навколо промислових підприємств, межа яких відома;
- динамічні зони – зони, межу яких еколог визначає на карті під час аналізу інформації про результати вимірів забруднень на місцевості.

Межу локальної зони визначають так, щоб зона знаходилася повністю у складі однієї адміністративно-територіальної одиниці (як правило, області) для того, щоб, формуючи план заходів, які потрібно провести в зоні, ураховувати можливості адміністративної одиниці.

Комплексна зона – інформаційна модель комплексного забруднення контрольованого регіону множиною забруднювачів, яку однозначно визначають замкнутою межею (полігоном) та типом середовища (літосферою, гідросферою або атмосферою). Визначають зону фахівці з метою її комплексного обстеження та формування загального плану проведення екологічних та медичних заходів. Про стан комплексної зони дізнаються за інформацією про локальні зони, з яких вона складається.

Інформаційну систему проектують так, щоб забезпечити проведення над зоною сукупності операцій, при цьому система реалізує технологію підключення нових операцій над зоною.

Множину операцій над зоною розподіляють на такі типи:

- операції актуалізації зони;
- операції візуалізації зони;
- аналітичні операції – різноманітні операції з аналізу інформації про стан зони, наприклад, прогнозування розвитку стану зони, проведення медичного аналізу захворювань, прогнозний економічний аналіз втрат та витрат на реабілітацію та ін.

Отже, система вирішує три основні завдання: збереження інформації про забруднення, формування комплексного плану заходів та проведення наукового аналізу стану довкілля.

Діаграму прецедентів системи комп'ютерного моніторингу показано на рис. 11.18. На діаграмі зображено користувачі системи та для кожного з них наведено прецеденти, тобто функції, для виконання яких користувач звертається до системи ЕЕМ

Структурну схему системи ЕЕМ, на якій зображено модульний склад системи з розподілом модулів системи за робочими місцями фахівців – користувачів системи показано на рис. 11.19.

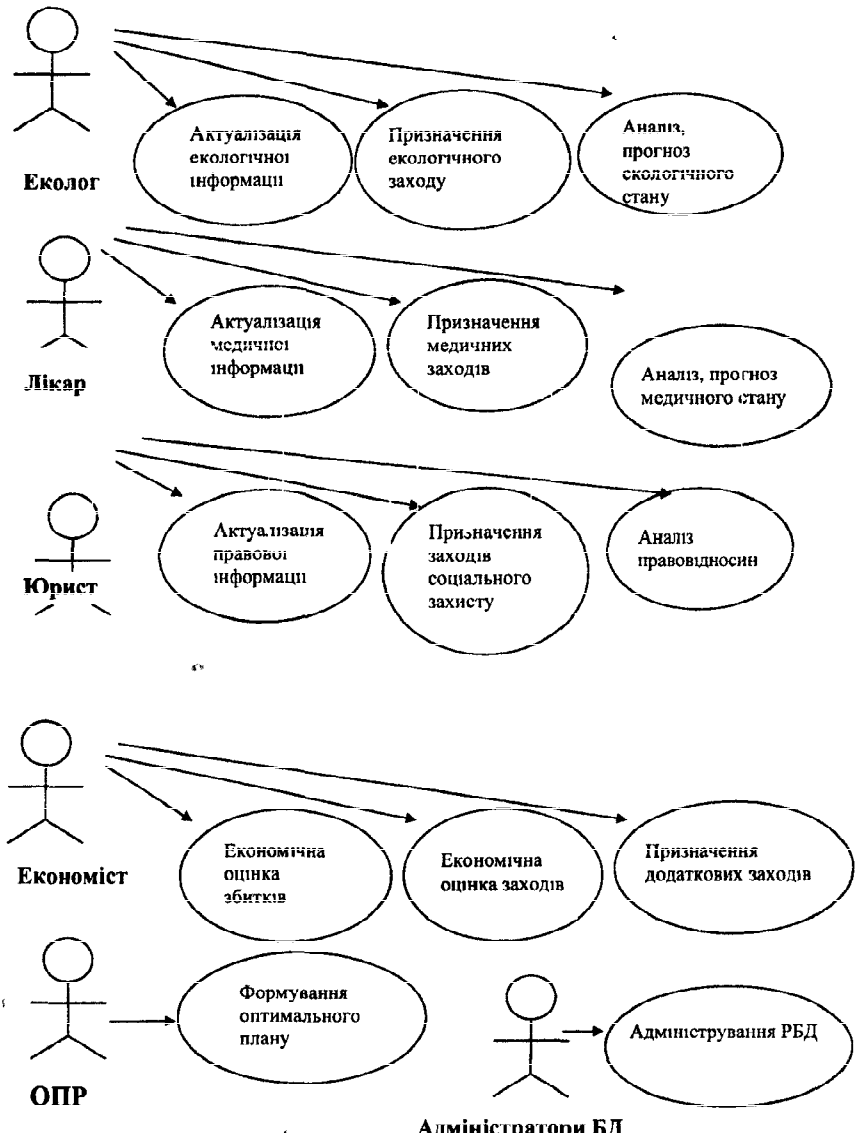


Рис 11 18 Діаграма прецедентів системи комп'ютерного моніторингу

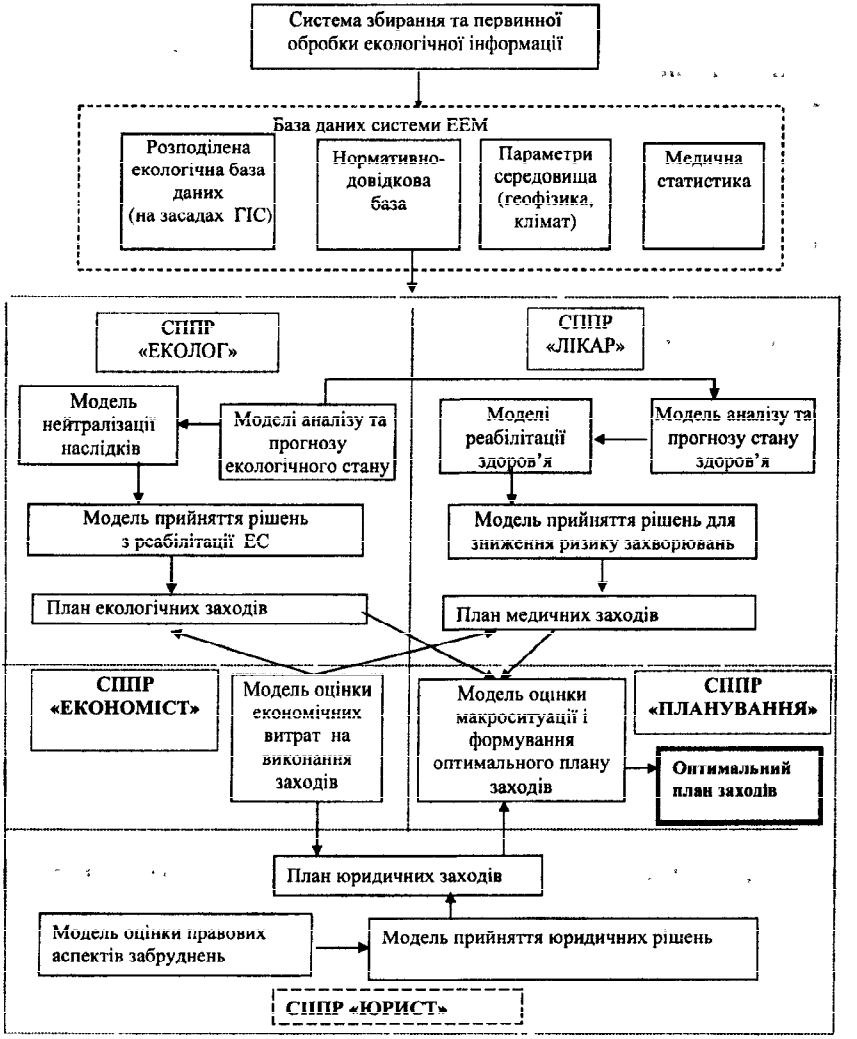


Рис. 11.19. Структурна схема системи ЕЕМ:
СППР – система підтримки прийняття рішень; ЕС – експертна система

11.6.3. Бази даних системи

Концептуальні рішення. Формування БД системи передбачає:

- створення в опорних центрах прогнозування і мінімізації генетичного ризику регіональних інформаційних БД про стан навколишнього середовища з урахуванням наслідків забруднення; для підтримки регіональних БД використовують SQL-сервери БД;
- створення головного центру прогнозування генетичного ризику для країни, у якому формується головна БД, яка містить інтегровану інформацію про екологічний стан країни;
- для забезпечення функціонування РБД системи створюється спеціальна база метаданих з інформацією про структуру РБД;
- використання глобальної мережі інтернету для забезпечення зв'язку головного центру з регіональними опорними центрами.

Отже, функціонування системи спирається на використання РБД, яка складається з множини регіональних БД.

Регіональні бази даних. Регіональні БД створюють у регіональних центрах мінімізації та прогнозування генетичного ризику. Регіональна БД містить інформацію про стан зон екологічного забруднення та заходи, що планують та проводять у зонах забруднення.

Регіональна БД містить такі таблиці:

- стану складових навколишнього середовища: літосфери, атмосфери, гідросфери;
- з інформацією про зони забруднення: «Зона», «Геометрія зони», «Збитки»;
- з інформацією про ресурси та заходи, що проводять або планують проводити в зоні: «Медичні заходи», «Екологічні заходи», «Юридичні заходи», «Ресурси екологічних заходів», «Ресурси медичних заходів», «Ресурси юридичних заходів»;
- з інформацією про захворювання в зоні: «Критерії захворювання», «Стан захворювання»;
- з інформацією про джерела забруднення: «Джерело забруднення» та акти аварійного забруднення «Акт забруднення» (Акт забруднення – це випадкова або аварійна ситуація, яка призвела до забруднення середовища.);
- зі службовою інформацією системи.

Концептуальну схему регіональної БД показано на рис. 11.20.

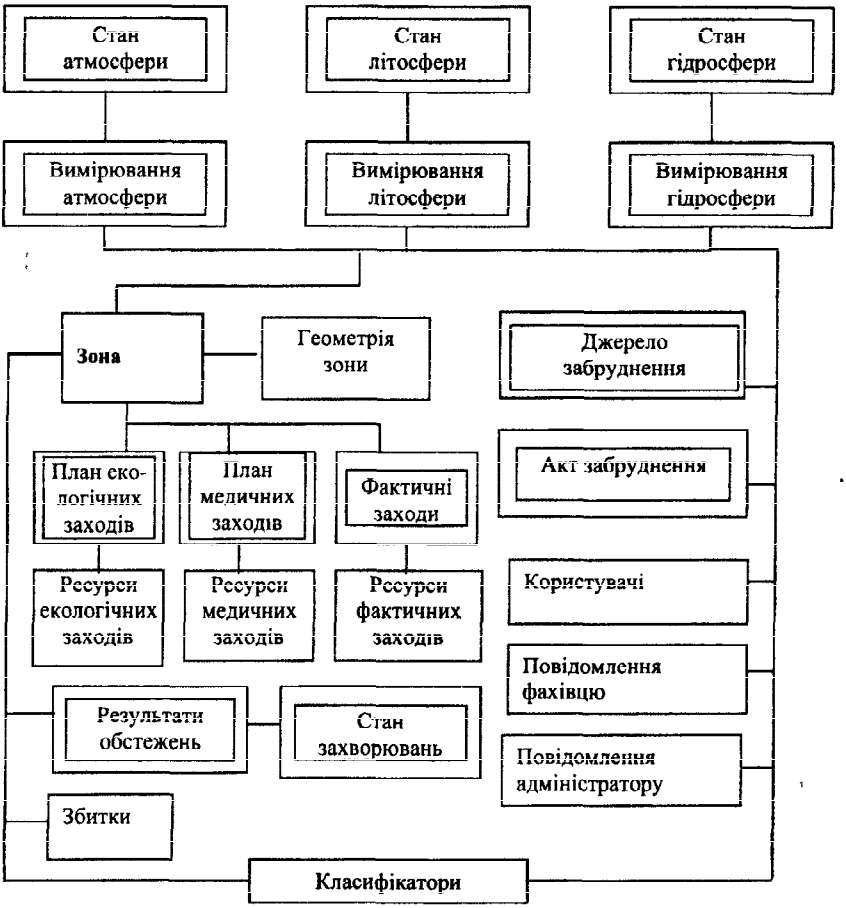


Рис. 11.20. Концептуальна схема регіональної БД

Головна база даних. Головну БД створюють у головному центрі мінімізації та прогнозування генетичного ризику. База даних містить інтегровану інформацію про стан зон екологічного забруднення, отриману в результаті розрахунку параметрів, отриманих із регіональних БД, та заходів, які планують проводити в зонах екологічного забруднення.

Концептуальну схему головної БД показано на рис. 11.21.

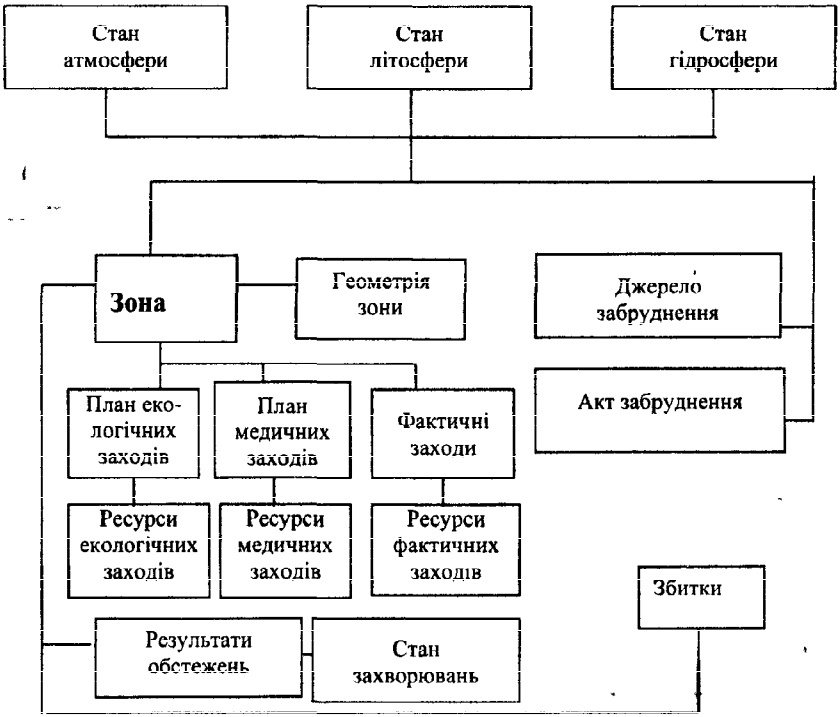


Рис. 11.21. Концептуальна схема головної БД

База метаданих. Базу метаданих створюють у головному центрі мінімізації та прогнозування генетичного ризику. Вона складається з таблиць, у яких зберігається інформація про структуру РБД, користувачів системи та їх права щодо доступу до інформації та про структури відповідних інформаційних об'єктів системи.

За допомогою цієї БД програмні засоби отримують інформацію про мережеві адреси регіональних БД та сервери БД, використані для підтримки регіональних БД, потрібну для проведення процедури формування БД головного центру прогнозування та моніторингу.

База метаданих містить також інформацію про множину користувачів, яка надає можливість керувати системою захисту інформації.

База метаданих містить такі таблиці:

- «Опис бази даних» – загальні характеристики регіональних БД;
- «Опис таблиці бази даних» – характеристики таблиць, що зберігаються в БД;
- «Опис поля таблиці» – характеристики окремих полів таблиць БД;

- «Опис міжтабличних зв'язків» – характеристики зв'язків між таблицями;
- «Опис категорії користувача» – характеристики категорій користувачів;
- «Опис користувача» – характеристики користувачів системи;
- «Опис зв'язку *категорія – користувач*» – характеристики зв'язку категорій та користувачів системи.

Концептуальну схему бази метаданих зображено на рис. 11. 22.

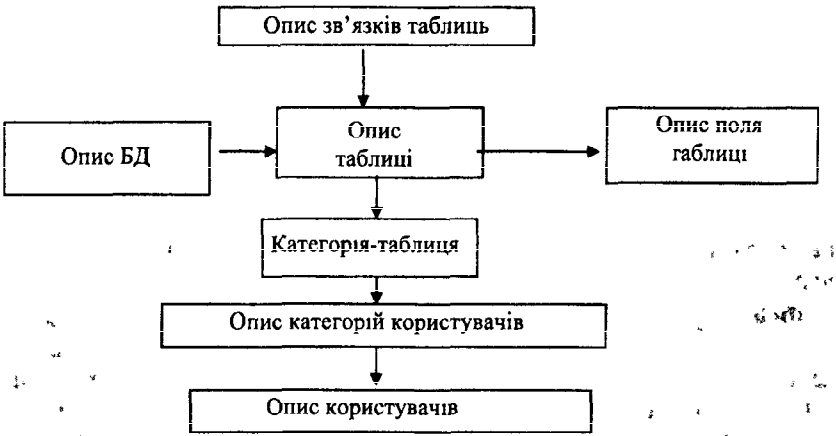


Рис. 11. 22. Концептуальна схема бази метаданих

11. 7. Типи баз даних. Дволанкова архітектура доступу до розподілених баз даних

11.7.1. Типи баз даних. Архітектури предметної області для доступу до РБД

Локальні БД. У локальних БД база даних розміщена на машині клієнта. Для зв'язку з БД клієнтський додаток використовує BDE. У цьому випадку здебільшого використовують одну зі стандартних БД:

- із файлами формату *.DBF (FoxPro);
- із файлами формату *.DB (Paradox).

Архітектура системи виглядає так, як показано на рис. 11.23.

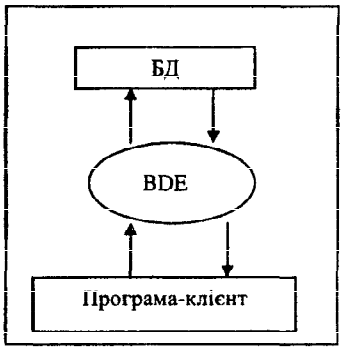


Рис. 11.23. Архітектура системи з локальною БД

Локальний варіант реально забезпечує доступ до даних лише одного користувача. У корпоративній роботі його не застосовують, оскільки в ньому важко синхронізувати вміст декількох копій БД, кількість яких має дорівнювати кількості користувачів.

Файл-серверні БД. У файл-серверних БД база даних зберігається на мережевому файл-сервері, який має бути доступним одночасно декільком користувачам, тому до таких БД можливий колективний доступ. Дані в БД зберігаються в єдиному екземплярі, а кожен клієнт у кожний момент часу працює з локальною копією БД, причому керування даними цілком покладається на клієнтські програми. Саме вони мають забезпечити синхронізацію вмісту локальних копій з основною БД. Для зв'язку з БД клієнтський додаток використовує BDE і разом із програмою клієнта утворює СКБД, кількість яких дорівнює кількості користувачів. У цьому випадку здебільшого використовують одну зі стандартних БД:

- із файлами формату *.DBF (FoxPro);
- із файлами формату *.DB (Paradox).

Архітектура системи виглядає так, як зображено на рис. 11.24.

Файл – серверний варіант реально забезпечує режим колективного доступу до даних. У корпоративній роботі його застосовують досить рідко, переважно у нескладних офісних системах, оскільки в ньому недостатньо засобів автоматичної синхронізації роботи користувачів.

Архітектура *клієнт – сервер* неефективна принаймні з трьох причин:

1) надзвичайно інтенсивне завантаження мережі, оскільки на машині клієнтів робляться локальні копії БД; результат – уповільнення роботи системи;
2) ускладнюється забезпечення цілісності БД;
3) не забезпечується належний захист даних.

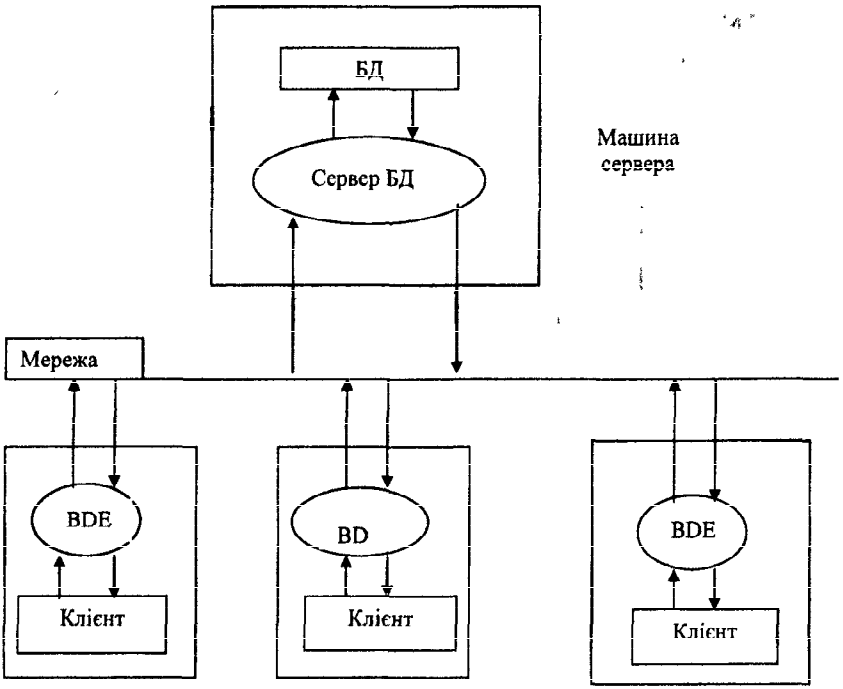


Рис. 11.24. Архітектура системи з файл-серверною БД

Клієнт-серверні БД. В архітектурі *клієнт – сервер* між BDE і БД з'являється важлива проміжна ланка – сервер БД – спеціальна програма, що керує БД. У такий спосіб архітектура *клієнт – сервер* дволанкова: першою ланкою є програма клієнта, а другою – сервер БД, що забезпечує доступ до БД. Клієнт формує запит до сервера мовою SQL, яка є промисловим стандартом для реляційних БД. SQL-сервер забезпечує інтерпретацію запиту, його виконання, формування результату і видачу цього результату користувачу. При цьому ресурси клієнтського комп'ютера не беруть участі у фізичному виконанні запиту, він лише відсилає запит і одержує результат. При цьому по мережі передаються тільки ті дані, які дійсно потрібні клієнту, у підсумку знижується навантаження на мережу. Крім того, SQL-сервер робить оптимізацію запиту так, щоб він виконувався за мінімально можливий час.

Архітектуру системи показано на рис. 11.25.

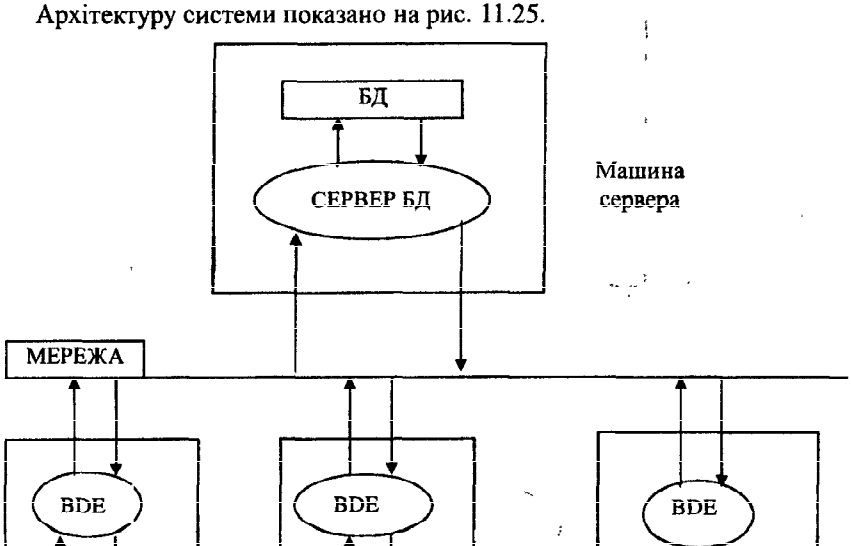


Рис. 11.25. Архітектура системи з клієнт-серверною БД

У разі виконання запитів сервером істотно підвищується ступінь безпеки, оскільки правила цілісності даних визначаються на сервері і є єдиними для всіх додатків, які використовують БД. Потужний апарат транзакцій, який підтримує SQL-сервер, блокує одночасну зміну одних і тих самих даних різними користувачами і надає можливість відкату до попередніх значень під час внесення змін у БД, якщо спроба змінити дані в БД закінчилася аварійно.

Дані в серверній БД зберігаються на диску у вигляді одного великого файлу, що разом з призначеним кожному користувачу паролем і привілеями доступу істотно підвищує захист даних від несанкціонованого доступу.

Архітектури із сервером додатків. Розвиток ідей архітектури *клієнт – сервер* зумовив появу триланкової архітектури доступу до БД (іноді її називають багатоланковою архітектурою, N-tier або Multi-tier архітекту-

рою). У триланковій архітектурі створюється допоміжна програма, яка містить усі компоненти для доступу до наборів даних (за дволанкової архітектури ці компоненти входили до складу програми-клієнта). Крім того, до них входять компоненти для керування сеансом роботи з БД TSession і компоненти керування доступом до конкретної БД – TDatabase. Цю програму розробляють як COM-сервер або як CORBA-сервер, реєструють у системі і вона стає сервером додатків.

Відповідно до програми клієнти є COM-клієнтами або CORBA-клієнтами, вони не містять компонентів доступу до БД. Для того щоб отримати дані, вони звертаються до інтерфейсів виділених серверів-додатків, що і реалізує потрібний обмін даними. У цій технології сервер додатків може бути розміщений на будь-якій машині в мережі, але найпопулярнішим є варіант розміщення сервера додатків на тій машині, на якій розміщений сервер БД і сама БД. Таке розміщення знижує навантаження на трафік мережі і забезпечує одночасну роботу сервера додатків і сервера БД.

За допомогою словників БД до компонентів можна перенести джерела і зв'язані з ними поля та частину бізнес-правил, які описують обмеження на вхідні дані. У цьому випадку неправильні дані сервер додатків відкидатиме і вони не передаватимуться на сервер БД.

У клієнтській програмі, яку у цій архітектурі називають полегшеним чи тонким клієнтом, розміщені:

- спеціальний зв'язувальний компонент – для з'єднання із сервером додатків;
- компоненти візуалізації даних;
- спеціальний компонент – клієнтський набір даних, що являє собою об'єкт, який дає доступ до копії частини даних із БД.

Усі зміни, які користувач вносить у дані, змінюють цю локальну копію і можуть до певного часу не передаватися в БД (режим відкладеної обробки даних). Крім того, під час роботи з громіздкими таблицями можна вимагати від сервера додатків передавати в локальний набір даних таблиці порціями, достатніми для одночасного відображення на екрані клієнта. Усі ці засоби істотно знижують навантаження на мережу і, отже, зменшують час чекання запиту.

Архітектура системи виглядає так, як показано на рис. 11.26.

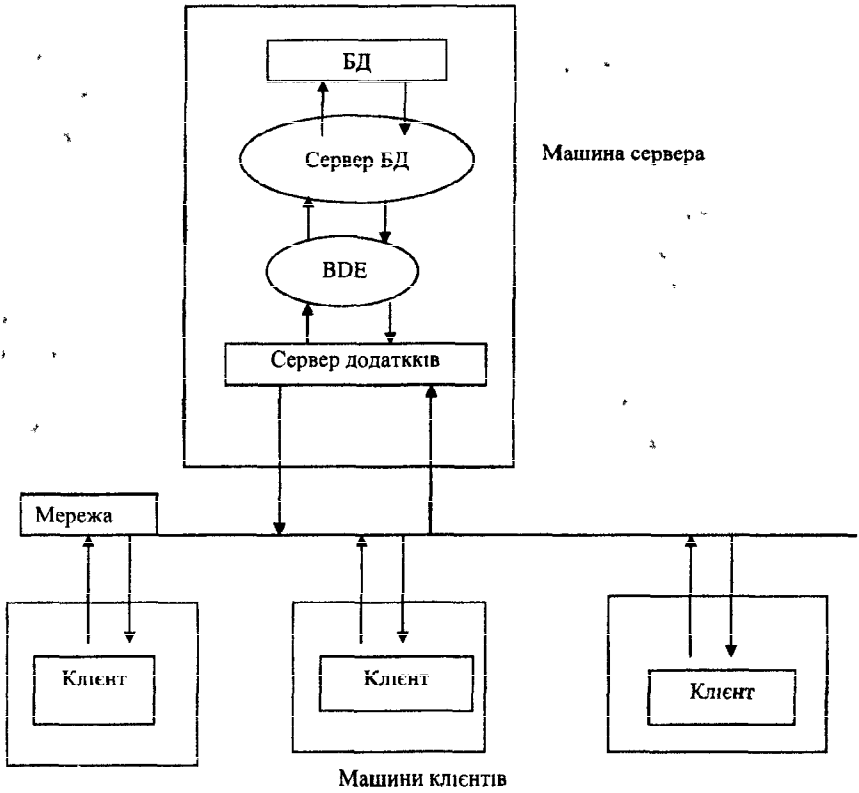


Рис. 11.26. Архітектура системи з використанням сервера додатків

11.7.2. Дволанкова архітектура *клієнт – сервер* у середовищі DELPHI і СКБД INTERBASE

Середовище візуального проектування DELPHI призначено для створення програм, які можуть працювати як з локальними, так і з віддаленими даними. При цьому можна використовувати як дволанкову, так і триланкову архітектуру системи.

Створення програм у дволанковій архітектурі *клієнт – сервер* має такі особливості:

- не рекомендується використовувати компонент TTable, оскільки він вимагає передачі всіх даних результату виконання запиту до сервера. Замість цього рекомендується використання компонента TQuery,

що отримує від сервера ту частину даних, яка має бути візуалізована;

- зміну записів БД варто робити не методами Insert, Edit, Delete класу TTable, а застосовувати відповідні команди мови SQL (INSERT, DELETE, UPDATE), використовуючи компонент TQuery;
- треба особливу увагу приділяти керуванню транзакціями і передусім – вибору рівня ізоляції транзакцій, адекватного потребам програми, при цьому варто використовувати способи явного керування транзакціями;
- бізнес-правила варто переносити на сервер додатків, розвантажуючи таким чином клієнтський додаток;
- із цього випливає, що потрібно намагатися використовувати збережені процедури і тригери для збільшення швидкості виконання передачі даних і зменшення навантаження на мережу;
- слід приділяти істотну увагу оптимізації запитів до БД, особливо під час читання даних оператором SELECT, оскільки оптимізований запит може виконуватися в кілька разів швидше.

Для з'єднання з віддаленим сервером у дволанковій системі досить установити псевдонім БД за допомогою утиліти BDE Administrator і потім використовувати цей псевдонім у компонентах TDataBase, TTable і TQuery. Працюючи з віддаленим мережевим набором даних, треба прописати його на комп'ютері, з якого відбувається звернення до сервера. У разі використання протоколу TCP/IP для цього потрібно:

- у файл HOSTS занести рядок з IP-адресою сервера і його ім'ям, наприклад:

196.182.20.2 server1

- у файл Services додати рядок з описом протоколу доступу до СКБД із зазначенням використовуваного мережевого протоколу, наприклад:

Interbase gds_db/tcp

11.8. Дволанкова реалізація розподіленої обробки

11.8.1. Постановка задачі

Припустимо, що існує фірма з центральним офісом у Києві та декількома філіями в інших містах України. На фірмі встановлено систему обліку кадрів, яка передбачає використання РБД. На кожній із філій є своя локальна БД. У центральному офісі також своя БД. Крім того, на машині

центрального офісу встановлено базу метаданих. В усіх локальних БД є таблиця «Personal» (табл. 11.1).

Таблиця 11.1. **Структура таблиці «Personal»**

Ім'я поля	Тип	Розмір	Коментар
Tab_Num	Символьний	10	Табельний номер
Id_Kod	Символьний	20	Ідентифікаційний код
Fam	Символьний	25	Прізвище
Imia	Символьний	15	Ім'я
Otch	Символьний	20	По батькові
Data_Rog	Дата	8	Дата народження
Data_Rab	Дата	8	Дата прийому на роботу
Oklad	Число	8,2	Місячний оклад
Adres	Символьний	80	Адреса
Tel	Символьний	15	Телефон

Звичайно, у БД є й інші таблиці, але для спрощення постановки задачі їх не розглядатимемо.

До складу бази метаданих входить таблиця «Loc DB» такої структури, як наведено в табл. 11.2.

Таблиця 11.2. **Структура таблиці «Loc DB»**

Ім'я поля	Тип	Розмір	Коментар
Gorod	Символьний	10	Назва міста
Filial	Символьний	10	Назва філії фірми
IP_adr	Символьний	20	IP адреса сервера
Adm	Символьний	25	Прізвище адміністратора БД
e-mail	Символьний	25	e-mail адміністратора БД

Як сервер БД використовують сервер INTERBASE.

Ставлять такі задачі:

- розробити програму, яка реалізує дещо незвичний запит «Одержати список співробітників, переглянувши локальні бази даних у Харкові і Львові, причому відібрати в харківській філії співробітників віком від 25 до 35 років, а у львівській філії співробітників з окладом від 500 до 700 гривень»; результат треба подати у вигляді єдиної таблиці на екрані користувача;
- розробити програму, яка реалізує розподілену обробку даних, тобто дає можливість користувачу, що одержав на екрані результат вищеописаного запиту, виконувати операції видалення, додавання і корегування записів у локальних БД.

Розв'язуючи такі задачі, потрібно звернути увагу на те, що стандарт мови SQL не має прямих засобів звернення до віддалених БД у тексті запиту. Надалі розглянемо засоби, надані діалектами мови SQL для різних СКБД,

що передбачають таку можливість. Відзначимо також, що для розв'язання цієї задачі можна було б використовувати можливість, надану BDE для реалізації гетерогенних запитів. Однак для кращого з'ясування сутності розподіленої обробки даних у цій задачі не будемо використовувати ці можливості. Для доступу до віддалених БД застосуємо аліаси (псевдоніми), у яких зазначено мережеві адреси локальних БД. Для спрощення задачі (але без обмеження узагальнення пропонованого рішення) припустимо, що локальні БД розміщені в локальній комп'ютерній мережі.

11.8.2. Реалізація розподіленого запиту

Поставлена задача передбачає, що користувач володіє навичками розробки DELPHI-додатків із локальними БД. Її можна розв'язувати різними способами. Розглянемо один із цих способів у вигляді послідовності таких кроків:

1. На машині головного офісу, на якій виконуватиметься розподілений запит, треба сформувати такі файли:
   - HOSTS – файл, що містить опис мережевих комп'ютерів, на яких зберігаються БД філій фірми, їх мережева адреса і мережеве ім'я серверів;
   - SERVICES – файл, що містить опис служби GDS_DB для мережевої роботи з СКБД Interbase із зазначенням використовуваного мережевого протоколу.
2. Створити псевдоніми до віддалених БД. Псевдоніми створюють за допомогою утиліти BDE ADMINISTRATOR так само, як і для локальних БД, але з тією відмінністю, що адресу БД записують у вигляді:

```
:мережеве_ім'я_сервера:шлях_до_файлу_бази_даних_на_сервері
```

3. Розробити типовий DELPHI-додаток для роботи з БД, розмістивши в його модулі даних такі компоненти:
   - TDATABASE – для кожної з локальних БД;
   - TDATABASE – для бази метаданих;
   - TQUERY – для кожного із запитів різної структури (до різних БД);
   - TTABLE – компоненти візуалізації для таблиці – результату запиту TDATASOURCE і TDBGRI.
4. Розробити додаток, що реалізує такий алгоритм:
   - вибір першої локальної БД (з використанням списку локальних БД, який зберігається в базі метаданих);
   - формування SQL-запиту до першої БД;
   - вибір другої локальної БД;
   - формування SQL-запиту до другої БД;

- створення в базі метаданих тимчасової таблиці для збереження результату розподіленого запиту (структура таблиці відповідає структурі таблиці «PERSONAL»);
- формування результуючої таблиці за результатами першого і другого запитів;
- візуалізація результату розподіленого запиту для користувача;
- після вказівки користувача видалення тимчасової таблиці – результату запиту з бази метаданих.

Отже, користувач цього додатка формує на власний розсуд розподілений запит і переглядає його на екрані комп'ютера. За потреби програма може бути ускладнена, наприклад, бажано до структури таблиці, яка містить результат розподіленого запиту, додати поле «Філія», у яке помістити назву філії і поле «Місто» з назвою міста, де знаходиться філія. Таке доповнення полегшить розв'язання наступної задачі, тобто реалізацію розподіленої обробки. Далі програму можна змінювати, варіюючи кількість локальних БД, задіяних у розподіленому запиті, і критерії добору записів у результуючу таблицю. Інший напрям модифікації програми – установлення виду результуючої таблиці та уточнення полів, які входять до її складу.

11.8.3. Реалізація розподіленої обробки

Реалізацію розподіленої обробки, тобто надання користувачеві можливості видалення, додавання і корегування записів у локальних БД, також можна виконувати по-різному. Опишемо реалізацію, що ґрунтується на модифікації вищеописаної програми розподіленого запиту. Модифікація програми полягає в такому:

1. Замість компонента TDBGRB, який використовували для візуалізації результатів розподіленого запиту, тепер використовуватимемо набір компонентів візуалізації, розміщених так, щоб користувач бачив на одному екрані не всю таблицю, а тільки один запис цієї таблиці у вигляді набору зручно розміщених на екрані полів таблиці. У цьому наборі полів передбачимо наявність поля – ідентифікатора локальної БД, з якої узято поточний запис.
2. Для виконання навігації за таблицею з результатами запиту розмістимо на формі відповідні елементи керування – кнопки з написами ПЕРШИЙ, ОСТАННІЙ, НАСТУПНИЙ, ПОПЕРЕДНІЙ.
3. Розробимо процедури навігації відповідно до кнопок.
4. Розмістимо в модулі даних додаткові компоненти TQUERY для формування запитів на доповнення, видалення і модифікацію записів у локальних БД.

5. На формі користувача розмістимо елементи керування розподіленої обробки – кнопки з написами ДОПОВНИТИ, ВИДАЛИТИ, ЗАМІНИТИ.

6. Розробимо процедури реалізації кнопок розподіленої обробки:

- за кнопкою ВИДАЛИТИ визначимо, до якої БД належить запис, що видаляється, далі сформуємо SQL-запит на видалення запису і викличемо для виконання; при цьому треба видалити відповідний запис і з таблиці з результатом запиту для синхронізації вмісту бази метаданих локальної БД;
- за кнопкою ДОПОВНИТИ зі списку локальних БД у базі метаданих визначимо БД, у яку потрібно внести новий запис, сформуємо SQL-запит на доповнення запису і викличемо для виконання; при цьому треба доповнити відповідний запис і в таблиці з результатом запиту для синхронізації вмісту бази метаданих локальної БД;
- за кнопкою ЗАМІНИТИ визначимо БД, до якої належить запис, який треба замінити, сформуємо SQL-запит на зміну запису і викликаємо для виконання; при цьому слід змінити відповідний запис і у таблиці з результатом запиту для синхронізації вмісту бази метаданих локальної БД.

Для перевірки роботи програми розподіленої обробки даних корисно після її виконання виконати програму розподіленого запиту, щоб переконатися в тому, що всі потрібні зміни в локальних БД проведено успішно.

11.8.4. Використання механізму гетерогенного запиту

Поняття гетерогенного запиту. Запити, що можуть звертатися до записів різних БД, називають *гетерогенними*. Можуть існувати запити не тільки до різних серверів, але і до різних типів серверів. Такі запити можливості стандартної мови запитів SQL не підтримують, їх реалізує програма BDE.

При цьому BDE виконує тільки такі гетерогенні запити, що створені за правилами синтаксису локального SQL.

Локальний SQL – підмножина мови стандартного SQL з такими обмеженнями:

1. Дозволяються такі команди маніпулювання даними:
   - SELECT – для пошуку даних;
   - INSERT – для доповнення даних до таблиці;
   - UPDATE – для зміни існуючих даних;
   - DELETE – для вилучення існуючих даних з таблиці.

2. У командах можна використовувати такі функції:

• агрегатні:

SUM( ) – одержати суму значень поля;

AVG( ) – одержати середньоарифметичне значення поля;

MIN( ) – одержати мінімальне значення поля;

MAX( ) – одержати максимальне значення поля;

COUNT( ) – одержати кількість записів відповідно до вказаного критерію;

COUNT(*) – одержати кількість записів з непорожнім (NOT NULL) значенням поля;

• рядкові:

UPPER( ) – перевести символи у верхній регістр;

LOWER( ) – перевести символи у нижній регістр;

TRI( ) – вилучити повторні входження символу;

SUBSTRING( ) – створити підрядок рядка;

• функції обробки дат. Локальний SQL надає функцію EXTRACT() для одержання частин значення дати, синтаксис якої має вигляд:

```
EXTRACT (частина, яку треба одержати) FROM ім'я_поля
```

Можна використовувати такі назви частин:

MONTH – одержати назву місяця;

DAY – одержати номер дня;

HOUR – одержати значення години;

MINUTE – одержати значення хвилини;

SECOND – одержати значення секунди.

Наприклад:

```
SELECT EXTRACT(YEAR FROM HIRE_DATE) FROM EMPLOYEE
```

3. У локальному SQL можна використовувати такі оператори:

• арифметичні:

+, -, *, /

• порівняння:

<, >, =, <>, IS NULL, IS NOTNULL, >=, =<

• логічні:

AND, OR, NOT

об'єднання рядків: ||

шаблони рядків: LIKE

4. Обмеження локального SQL під час формування запитів. Драйвери SQL Links підтримують виконання запитів як до однієї таблиці, так і до об'єднаних таблиць.

5. Обмеження під час виконання запитів. Запити до таблиць або до подань виконуються за таких умов:

- не дозволяються оператори типів JOIN, UNION, INTERSECT, MINUS;
- не дозволяється операнд DISTINCT в операторі SELECT;
- в операторі SELECT дозволяються тільки прості поля або поля, які не обчислюються з використанням агрегатних функцій;
- не дозволяються операнди GROUP BY і HAVING;
- не дозволяються запити, які звертаються до внутрішніх запитів;
- якщо для поля вказано операнд ORDER BY, обов'язково створити окремий індекс.

6. Обмеження на змінні, які визначають умови об'єднання таблиць:

- об'єднання можуть бути зовнішніми чи внутрішніми типу праворуч;
- об'єднання є об'єднаннями за еквівалентністю;
- для полів, які визначають об'єднання, потрібна наявність індексів (для Paradox і dBASE);
- не можна вимагати сортування результуючої таблиці.

7. Зазначені обмеження належать також до запитів на поновлення таблиць.

8. Дозволяються такі команди маніпулювання даними:

CREATE TABLE,
ALTER TABLE,
DROP TABLE,
CREATE INDEX,
DROP INDEX,
CREATE VIEW.

Для створення і виконання гетерогенного запиту треба:

- створити стандартний BDE псевдонім для кожної БД, що бере участь у запиті;
- розмістити на формі компоненти TQUERY, вказавши у властивості Database NAME один із цих псевдонімів;
- сформувати у властивості SQL оператор SQL, який виконує гетерогенний запит для пошуку.

Перед кожним ім'ям таблиці в запиті треба вказати в лапках назву BDE-псевдоніма для БД, у якій знаходиться таблиця та ім'я таблиці, тобто створити псевдонім таблиці. Перед назвою псевдоніма і назвою таблиці слід вставити символ ":" Наприклад, ":WORKDEMOS:CUSTOMER" CUSTOMER

- установити потрібні значення для параметрів запиту;

- до першого виконання реалізувати метод Prepare для підготовки запиту;
- виконати метод Open чи ExecSQL залежно від типу SQL-запиту (якщо повертає таблицю – метод Open , якщо ні – метод ExecSQL).

Приклад 1. Цей приклад містить текст гетерогенного запиту до сервера Informix, що має псевдонім INFORMX і містить набір даних EMPLOYEE, і до БД InterBase, що має псевдонім INTRBASE і містить таблицю ITEMS:

```
SELECT EMPLOYEE. EMPNO, ITEMS. ITEMNO
FROM ": INFORMX: EMPLOYEE"EMPLOYEE,
 ": INTRBASE: ITEMS"ITEMS
```

Приклад 2. Містить текст гетерогенного запиту до сервера INTERBASE, який має псевдонім IBLOCAL і містить набір даних EMPLOYEE , і до БД PARADOX, яка має псевдонім WORKDEMOS і містить таблицю CUSTOMER:

```
SELECT CUSTOMER.CUSTNO , CUSTOMER.COMPANY,
 EMPLOYEE.EMP_NO ,EMPLOYEE.FIRST_NAME
FROM ":WORKDEMOS:CUSTOMER" CUSTOMER,
 ":IBLOCAL: EMPLOYEE" EMPLOYEE
WHERE CustNo = :cnom
```

Список літератури

1. *Дейт К.* Введение в систему баз данных. – М.: Мир, 1998. – 846 с.
2. *Бойко В. В., Савинков В. М.* Проектирование баз данных информационных систем. – М.: Финансы и статистика, 1989. – 350 с.
3. *Джексон Г.* Проектирование реляционных баз данных. – М.: Мир, 1991. – 252 с.
4. *Олле Т.* Предложения КОДАСИЛ по управлению БД. – М.: Финансы и статистика, 1981. – 286 с.
5. *Озкарахан* Э. Машины баз данных и управление базами данных. – М.: Мир, 1989. – 695 с.
6. *Цикритзис Д., Лоховски Ф.* Модели данных. – М.: Финансы и статистика, 1985. – 343 с.
7. *Наумов А. Н.* Системы управления базами данных и знаний. – М.: Финансы и статистика, 1991. – 348 с.
8. *Артемов Д. В., Погульский Г. В., Альперович М. М.* Microsoft SQL Server 7.0 для профессионалов. – М.: Издат. отдел «Русская редакция» ТОО «Channel Trading Ltd», 1999. – 576 с.
9. *Горев А., Макашарипов С., Владимиров Ю.* Microsoft SQL Server 6.5 для профессионалов. – СПб: Питер, 1998. – 464 с.
10. *Шумаков П. В.* Delphi 3 и разработка приложений баз данных. – М.: НОЛИДЖ, 1998. – 704 с.
11. *Хендерсон К.* Руководство разработчика баз данных в Delphi 2. – К.: Диалектика, 1996. – 544 с.
12. *Попов А. А.* Программирование в среде СУБД FOXPRO 2.6. – М.: Радио и связь, 1998. – 350 с.
13. *Каратыгин С., Тихонов А., Тихонова Л.* Visual FoxPro 5. К вершинам мастерства. – М.: Вост. кн. компания, 1997. – 784 с.
14. *Сосински Б.* Разработка приложений в среде Visual FoxPro 5. – К.: Диалектика, 1997. – 441 с.

Зміст